Interfaces, coleções

Prof. Renato Pimentel

2024/2

Sumário

Como visto, o conjunto de métodos disponíveis em um objeto é chamado **interface**:
É através da interface que se interage com os objetos de uma determinada classe – através de seus métodos.

Uma definição mais formal:

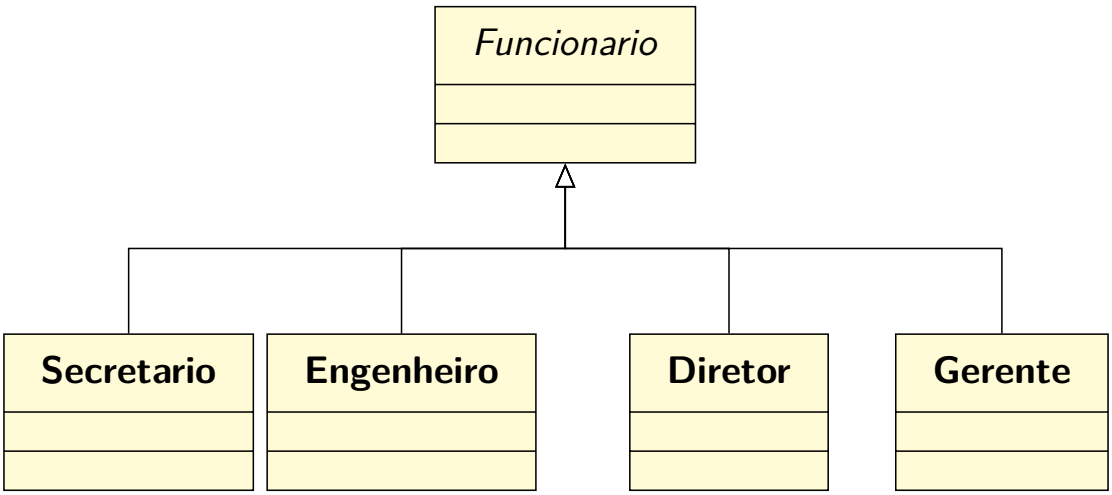
Interface

"Contrato" assumido por uma classe, de modo a garantir certas funcionalidades a suas instâncias.

Em outras palavras, uma interface define *quais* métodos que uma classe **deve implementar** visando tais funcionalidades.

Motivação I

Considere o exemplo:



Motivação II

As classes acima definem o organograma de uma empresa ou banco, ao qual criamos um *sistema interno* que pode ser acessado *somente* por diretores e gerentes:

```
1 class sistemaInterno {
2    void login(Funcionario funcionario) {
3       funcionario.autentica(...); //  erro de semâ
   ntica: nem todo funcionario autentica.
4    }
5 }
```

Engenheiros e secretários não autenticam no sistema interno.
Como resolver o problema, do ponto de vista de OO?

Motivação III

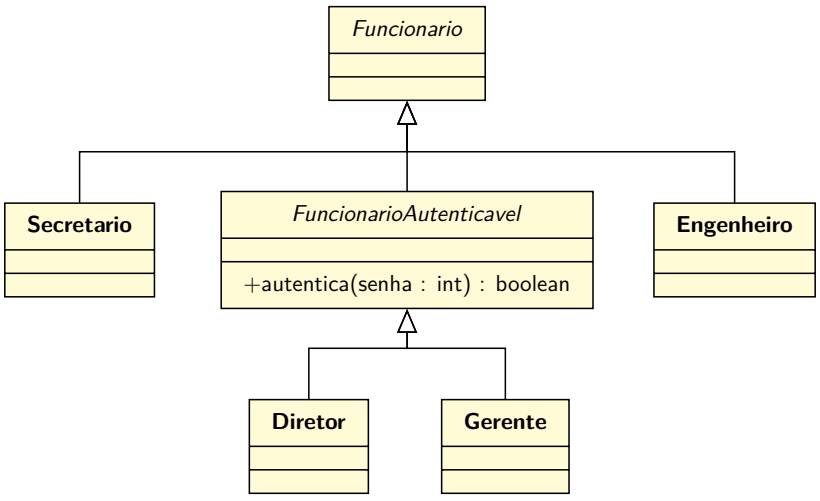
Alternativa (ruim):

```
1 class SistemaInterno {
2     void login(Diretor funcionario) {
3         funcionario.autentica(...);
4     }
5
6     void login(Gerente funcionario) {
7         funcionario.autentica(...);
8     }
9 }
```

Cada vez que outra classe que possa autenticar é definida, devemos adicionar um novo método `login()`.

Motivação IV

Possível solução: subclasse intermediária, com o método `autentica()`. Tal classe pode ou não ser abstrata:



Motivação V

Caso mais complexo: e se *cliente* da empresa ou banco também tiver direito a acessar o sistema interno, através de login?

`class Cliente extends FuncionarioAutenticavel { ... }` – **Herança sem sentido**!

Não há sentido `Cliente` herdar atributos e comportamentos de `Funcionario`, ex.: salário, bonificação, etc.

Herança é um tipo de relacionamento **"é um"**.

Motivação VI

Solução: usar o conceito de **interface**.
O fato é que as classes `Gerente`, `Diretor` e `Cliente` possuem um fator comum, o método `autentica()` – porém apenas as duas primeiras correspondem a funcionários no sistema.

Motivação VII

O "contrato", portanto, a ser assumido por tais classes, é que devem ser *autenticáveis* no sistema. Assim, cria-se a interface de nome, por exemplo, `Autenticavel`, possuindo a *assinatura* ou protótipo do método `autentica()`:

Interface I

```
1 interface Autenticavel {
2    boolean autentica(int senha);
3 }
```

Interface II

A interface diz **o que** o objeto deve fazer, mas não **como** fazer. Isto será definido na *implementação* da interface por uma classe.
Em Java: palavra reservada `implements`.

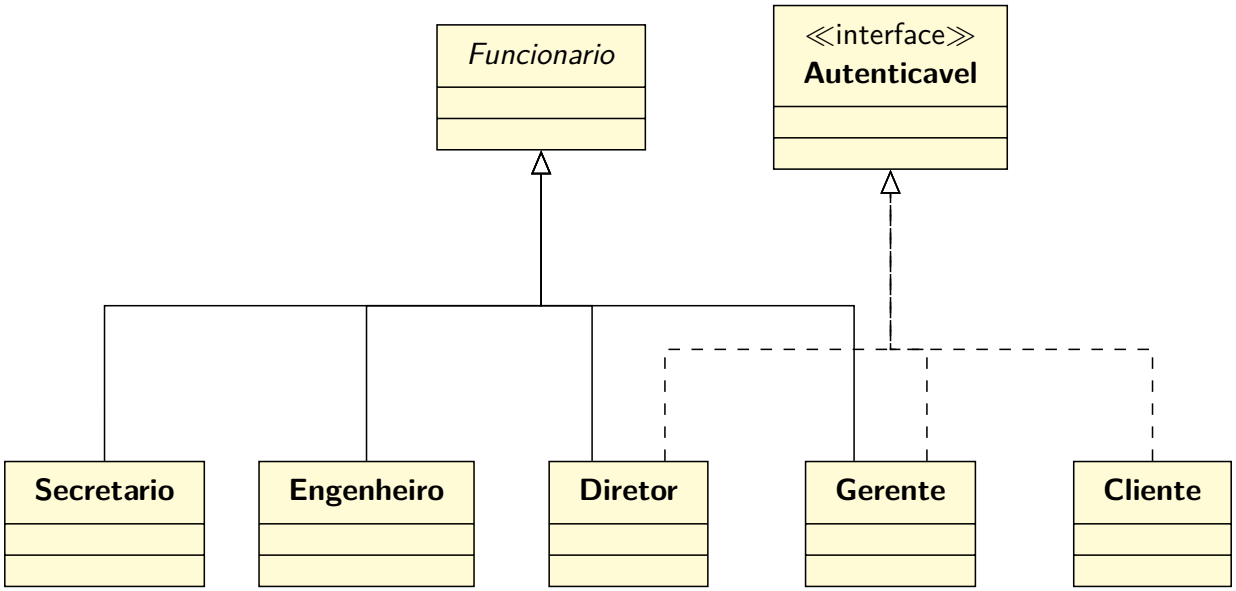
```
class Gerente extends Funcionario implements
    Autenticavel {
    private int senha;
    //...
    public boolean autentica(int senha) {
        // a implementação deste método é obrigatória
    na classe Gerente.
    }
}
```


Interface III

```
class Cliente implements Autenticavel {
    private int senha;
    //...
    public boolean autentica(int senha) {
        // a implementação deste método é obrigatória
    na classe Cliente.
    }
}
```

Interface IV

Diagrama de classes:



Interface V

O *polimorfismo* é uma vantagem ao se utilizar interfaces: podemos ter algo do tipo

`Autenticavel a = new Gerente();`

A variável do tipo `Autenticavel` pode se referir a objeto de *qualquer* classe que implemente `Autenticavel`. E o sistema interno visto fica como:

```
1 class SistemaInterno {
2    void login(Autenticavel a) {
3       // lê uma senha ...
4       boolean ok = a.autentica(senha);
5       // objeto que autentica não é mais necessariamente
     funcionário...
6    }
7 }
```

Usos da interface I

Vimos dois usos para interface:

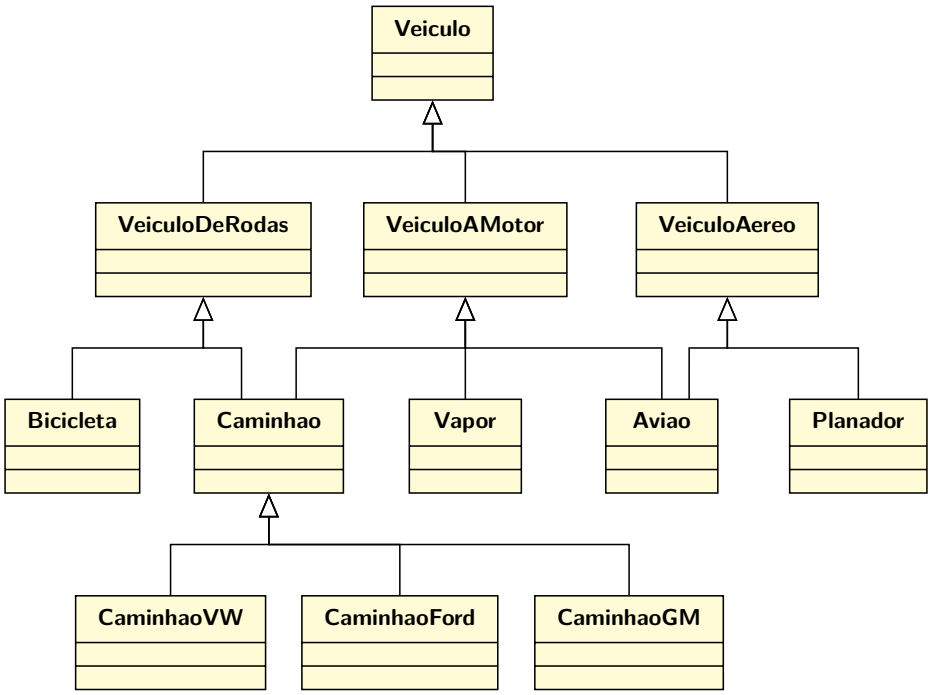
- Capturar *similaridades de comportamento* entre classes não relacionadas diretamente (definir superclasse abstrata não seria natural, pois não há "parentesco" forte entre as classes);
- Declarar métodos que uma ou mais classes não relacionadas *devem* necessariamente implementar.

Usos da interface II

Um outro uso:

- Revelar uma *interface de programação*, sem revelar a classe que a implementa (apresenta a interface pública, mas não revela onde está a implementação).

Herança múltipla I



Herança múltipla II

Em Java – e em muitas linguagens OO, a **múltipla herança** – em que uma classe possui mais de uma superclasse – não é permitida nativamente.
Isto é, a instrução abaixo **não** é possível:

```
public class Aviao extends VeiculoAMotor,
VeiculoAereo {...}
```

A restrição evita conflitos que podem ocorrer devido aos *atributos* herdados das superclasses, que podem se sobrepor, afetando a definição do *estado* de um objeto da subclasse.
Mais detalhes em https://docs.oracle.com/javase/tutorial/java/IandI/multipleinheritance.html

Herança múltipla III

Interfaces:

- Não possuem atributos;
- Não possuem construtores;
- Contém **assinaturas de métodos**:
  - Métodos em interfaces são **abstratos** (e **públicos**).

Como não temos atributos sendo definidos nas interfaces, eliminamos a restrição vista anteriormente:
É possível que uma classe implemente várias interfaces (ideia de herança múltipla).

Herança múltipla IV

```
1 public interface VeiculoAMotor {
2    void ligarMotor();
3 }
```

```
1 public interface VeiculoAereo {
2    void voar();
3 }
```

Herança múltipla V

```java
public class Aviao implements VeiculoAMotor, VeiculoAereo {
    // atributos...

    public void ligarMotor() {
        // classe Aviao deve implementar este método
    }

    public void voar() {
        // classe Aviao deve implementar este método
    }

    ...
}
```


Herança múltipla VI

Uma interface também pode herdar características de outra(s) interface(s) – e *somente* de interface(s) – via herança:

```java
public interface Interface4 extends Interface1,
Interface2, Interface3 { ...
```

Outras observações

- Interfaces são codificadas em arquivo próprio, com mesmo nome da interface;
- Classes que usam uma interface têm que implementar **todos os métodos** definidos na interface – mas há exceções (a partir Java 8);
- Uma classe pode implementar mais de uma interface, desde que não haja **conflitos de nomes**.

Conflitos de nomes e herança múltipla I

```
1 interface A {
2    tipo metodo(pars);
3 }
```

```
1 interface B {
2    tipo metodo(pars);
3 }
```

Se uma classe implementa essas duas interfaces, haverá conflito de nomes.

Conflitos de nomes e herança múltipla II

Conflitos:

- Se os métodos têm o **mesmo nome**, mas parâmetros diferentes, não há conflito, há **sobrecarga**;
- Se os métodos tiverem a **mesma assinatura**, a classe implementará *um método apenas*;
- Se as assinaturas dos métodos diferirem **apenas** no tipo de retorno, a classe não poderá implementar as duas interfaces – este, na verdade, é o único conflito não-tratável.

Mais exemplos I

Supor uma classe `Classificador`, com um método `ordena()`, que faz a ordenação de objetos de outras classes.

O método `ordena()` implementa um algoritmo de ordenação que compara todos os elementos usando o método `eMaiorQue()`;

Toda classe que quiser ter ordenação de seus objetos **deve implementar** o método `eMaiorQue()`.

Mais exemplos II

```
1 public class Classificador {
2    void ordena(Object[] a) {
3       if (a[i].eMaiorQue(a[i+1])) {
4          ...
5       }
6    }
7 }
```

Como garantir que toda classe que necessite de ordenação implemente o método `eMaiorQue()`? **Usar interfaces.**

Mais exemplos III

```
1 public interface Classificavel {
2    boolean eMaiorQue(Classificavel obj);
3 }
```

Mais exemplos IV

```java
public class Produto implements Classificavel {
    String nome;
    double preco;
    ...
    boolean eMaiorQue (Classificavel obj)
    { ... }
    ...
}
```

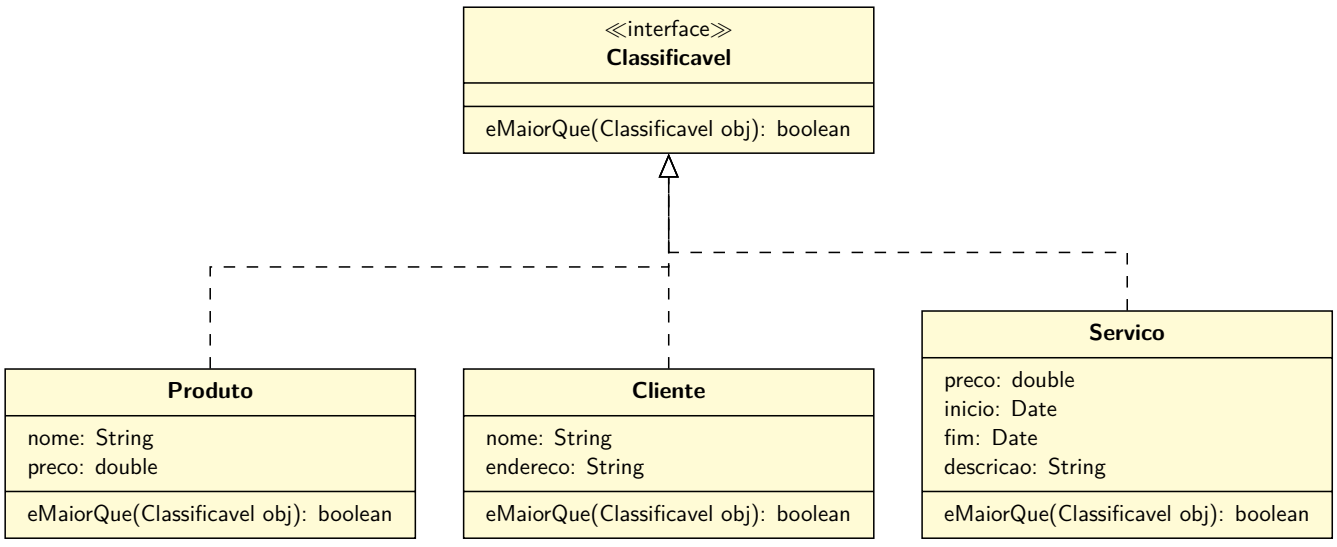

Mais exemplos V

```java
public class Cliente implements Classificavel {
    String nome;
    String endereco;
    ...
    boolean eMaiorQue (Classificavel obj)
    { ... }
    ...
}
```

Mais exemplos VI

```java
public class Servico implements Classificavel {
    double preco;
    Date inicio, fim;
    String descricao;
    ...
    boolean eMaiorQue (Classificavel obj)
    { ... }
    ...
}
```


Mais exemplos VII



Outros membros da interface I

Foi visto que interfaces não contêm atributos, e sim **assinaturas de métodos** (isto é, métodos abstratos).
Além de tais membros, as interfaces também podem conter *constantes*; e métodos *estáticos* e *default*[1] (implementação padrão)

Outros membros da interface II

Constantes em interfaces:
Implicitamente, são `public`, `static` (i.e., da classe) e `final` (não é necessário escrever tais modificadores).

- `final`: indica que não pode ser alterado (equivalente a `const` em C)

```
1 public interface A {
2    // base of natural logarithms
3    double E = 2.718282;
4
5    // method signatures
6    void doSomething (int i, double x);
7    int doSomethingElse(String s);
8 }
```

E: valor *constante* – note que o valor é atribuído logo na declaração.

Outros membros da interface III

Por extensão, **enumerações** (listas de constantes) também são permitidos:

```
1 public interface B {
2    enum diaSemana {
3       DOMINGO, SEGUNDA, TERCA, QUARTA, QUINTA, SEXTA
4    }
5
6    // assinaturas de métodos...
7
8 }
```

Constantes em Java, por convenção, sempre em **caixa alta** (maiúsculas) – veja http://www.oracle.com/technetwork/java/javase/documentation/codeconventions-135099.html. Mais de uma palavra: separam-se com _ (exemplo: SEGUNDA_FEIRA).

Outros membros da interface IV

Antes da versão 8 do Java:
Toda vez que se quisesse atualizar a especificação de uma interface, adicionando-se à mesma um novo método, **todas as classes** que previamente a implementavam deveriam também implementar tal método, ou não compilariam.

Uma opção: colocar o novo método numa nova interface, **sub-interface** da anterior:

```
1 public interface B extends A {
2    tipo novoMetodo (...);
3 }
```

(uso agora opcional: quem quiser usar nova interface implementa B, mas quem desejar manter apenas requisitos anteriores continua implementando A).

Java 8 – Processo simplificado com novo recurso:
Métodos *default*: a implementação *padrão* **é feita na interface** (não são abstratos); (re-)implementação nas classes torna-se *facultativa* –
Classes que implementam uma interface contendo um ou mais métodos *default* pode, mas não precisa implementá-los – já os terá definidos.

```
1 public interface A {
2    // outros métodos...
3
4    default tipo novoMetodo (...) {
5       // implementação aqui
6    }
7 }
```

Agora, `novoMetodo` pode ser chamado para qualquer objeto de classe que implemente A.

Outros membros da interface VII

Herança entre interfaces e métodos *default*:
Se B é sub-interface de A, que define método *default* `novoMetodo`:

- B não menciona tal método, apenas o herdando como método *default*;
- **Redeclara** o método, **tornando-o abstrato**:
  - Toda classe que implementar B, e não A, precisará implementar `novoMetodo`;
- **Redefine** o método, reimplementando-o (será *default* também em B):
  - Toda classe que implementar B, e não A, terá como implementação padrão (*default*) de `novoMetodo` a dada em B – e não em A.

Outros membros da interface VIII

Também em Java 8: **Métodos estáticos**.

- Associados a classe/interface que os define, e não a objetos.
- Não podem ser redefinidos via polimorfismo dinâmico (sobrescrita).

```
1 public interface A {
2    static double sqrt2() {
3       return Math.sqrt(2.0);
4    }
5 }
```

Outros membros da interface IX

```java
public class Classe implements A {
    // ...
    public void mostraSqrt2() {
        System.out.println("Raiz de 2 = " + A.sqrt2()); // sqrt2 é está
     tico, dado pela interface A
    }
}
```

[1] Java 8

Classes abstratas × interfaces

Classe abstrata	**Interface**
Uma classe pode ser subclasse de apenas uma classe abstrata;	Uma classe pode implementar múltiplas interfaces;
Faz parte de uma hierarquia que possui correlação ("é-um").	Não faz parte de uma hierarquia (classes sem relação podem implementar uma mesma interface).

Coleções – motivação I

Em Java é possível armazenar um conjunto de valores, primitivos ou objetos, utilizando variáveis compostas homogêneas (vetores, matrizes, etc)

Mas e se quisermos:

- Criar estruturas que aloquem dinamicamente espaço em memória (aumentar ou diminuir o espaço **em tempo de execução**)?
- Criar estruturas de dados mais complexas com disciplinas de acesso, através da implementação de tipos abstratos de dados como **listas**, **pilhas** e **filas**?

Coleções – motivação II

Estas questões são tratadas comumente em disciplinas específicas de **estruturas de dados**.

Na linguagem de programação Java, estas estruturas são oferecidas através do **Java Collections Framework**.

Estruturas de dados comuns I

Arrays são estruturas de dados poderosas, mas com utilização específica:

- Inadequados para excluir/incluir elementos frequentemente.

Em geral, não existe uma estrutura de dados que tenha desempenho excelente para várias operações que se possa realizar, como:

- Incluir, excluir, alterar, listar, ordenar, pesquisar, etc.

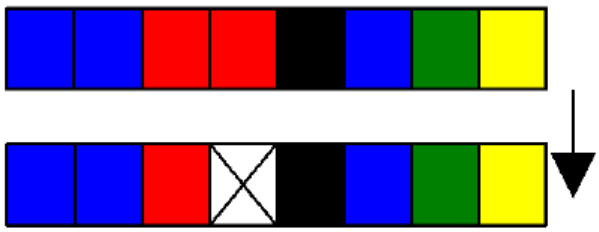
Estruturas de dados comuns II

Além disso, manipular os *arrays* do Java é bastante trabalhoso:

- Não se pode redimensionar;
- A busca direta por um determinado elemento cujo índice não se conhece não é possível;
- Não se sabe quantas posições do *array* foram efetivamente usadas, sem uso de recursos auxiliares, como *contadores*.

Estruturas de dados comuns III

Exemplo: remoção em posição intermediária de um *array*.



- Ao inserir novo elemento: procurar posição vazia?
- Armazenar lista de posições vazias?
- E se não houver espaço vazio? (`arraycopy()` não é bom)
- E qual o tamanho da estrutura? (posições de fato usadas?)

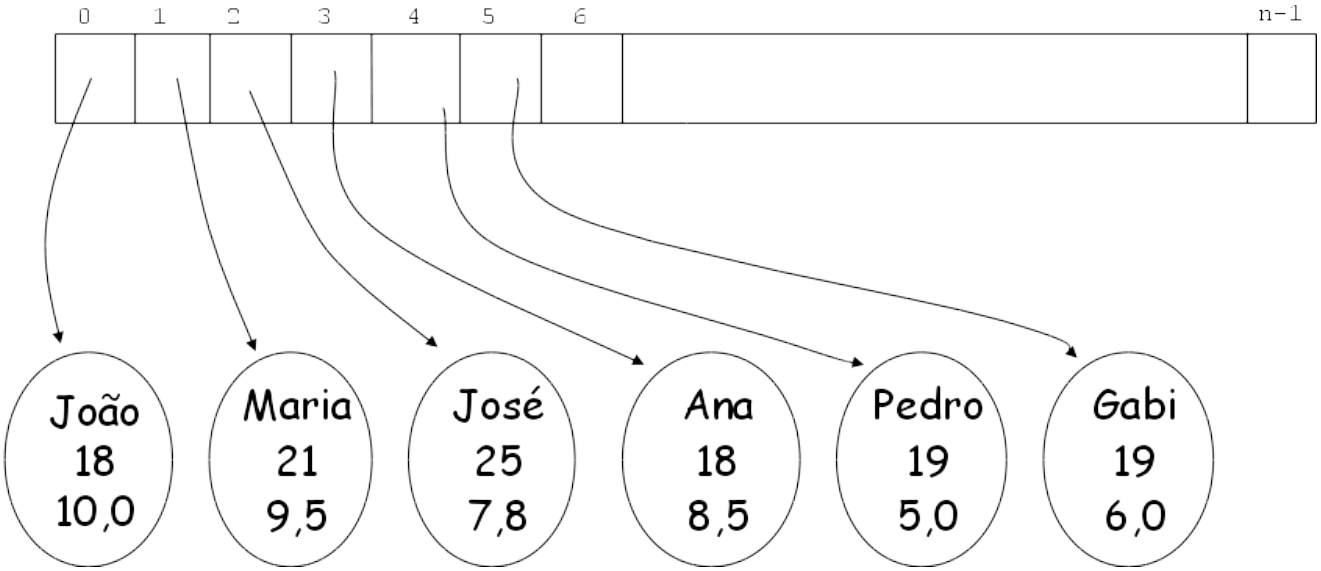
Estruturas de dados comuns IV

Um outro ponto importante:
ordenação.

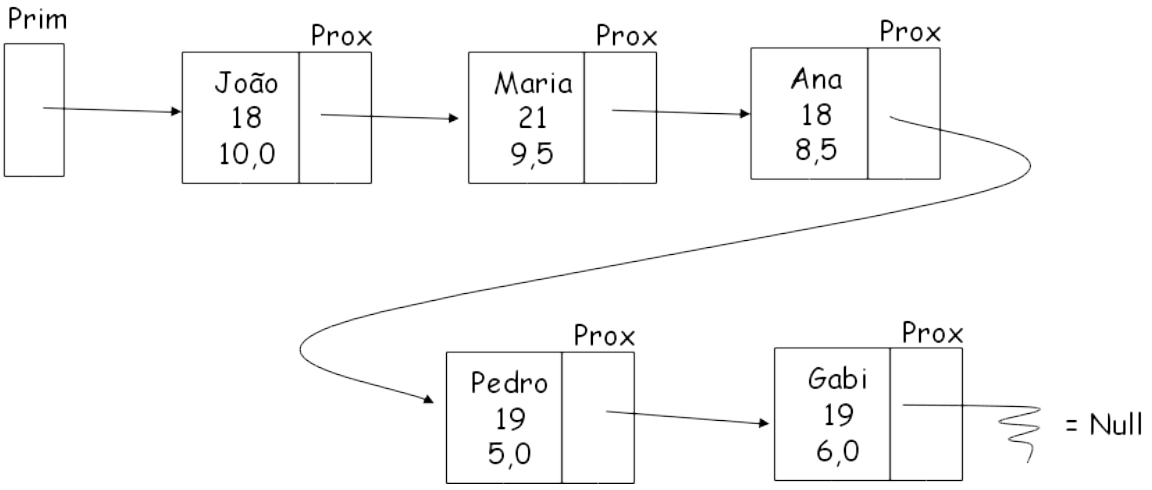
O que existem são estruturas de dados que são **"melhores"** para algumas operações:
A decisão depende do problema específico.

Algumas estruturas de dados:

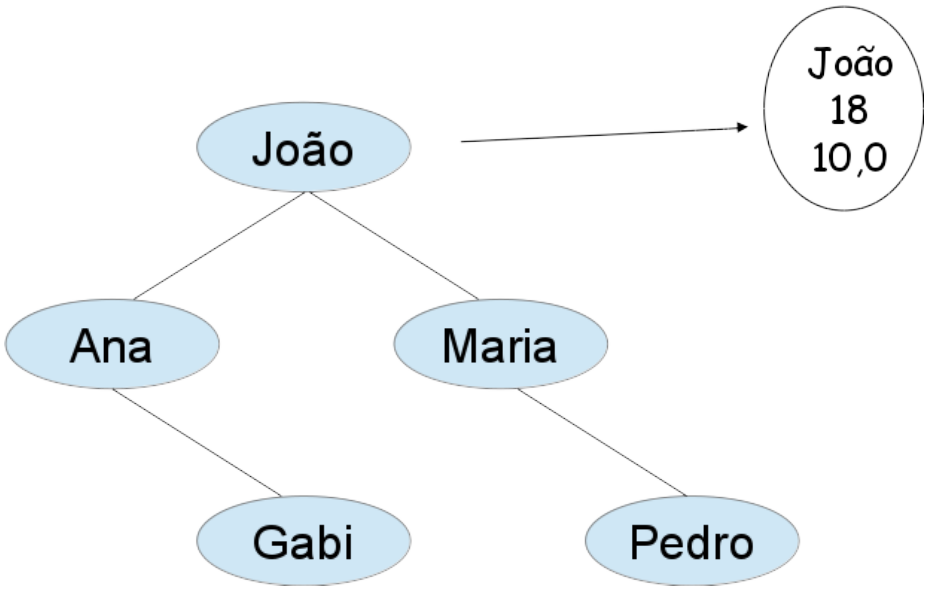
- Vetores;
- Listas encadeadas;
- Pilhas;
- Árvores;
- Tabelas *hash*;
- Etc.



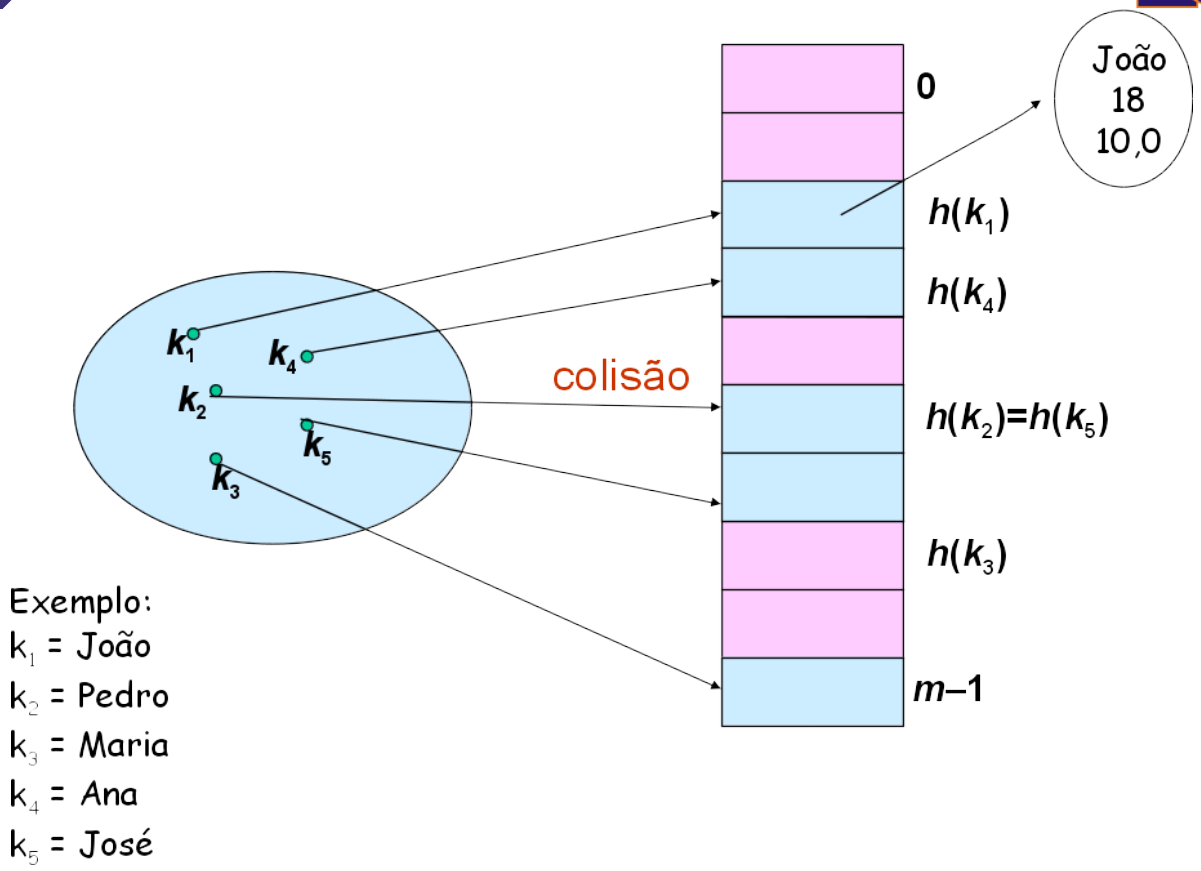
Exemplo de lista (encadeada)



Exemplo de árvore (árvore binária de busca)



Exemplo de tabela *hash*



Coleções I

Como podemos ver, existem ED especializadas em certas operações/funcionalidades.

E em Java?

Coleções II

A partir do Java 1.2 (Java 2):
Collections framework

Collections framework

Conjunto de **classes** e **interfaces** Java, dentro do pacote nativo `java.util`, que representam diversas estruturas de dados avançadas.

- em outras palavras, são implementações pré-existentes em Java para EDs bem conhecidas;
- Possuem métodos para *armazenar*, *recuperar*, *consultar*, *listar* e *alterar* dados que são tratados de forma agregada;

Coleções III

Uma definição para o que seria uma *coleção*:

Coleção

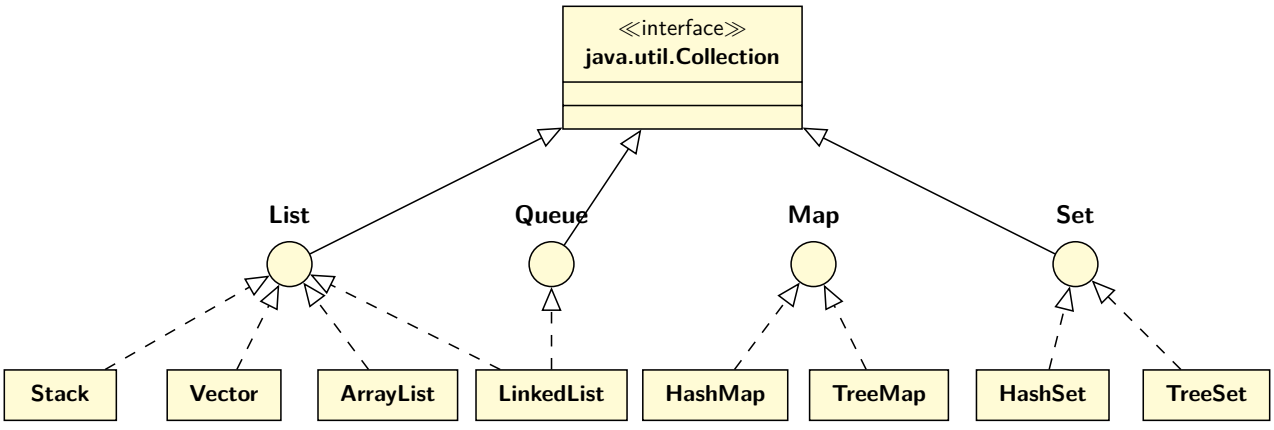
Um *objeto* que agrupa múltiplos elementos em uma estrutura única

Coleções IV

Exemplos de situações em que se usa coleções:

- Lista de disciplinas, lista de professores, lista de alunos, lista de turmas, lista de alunos por turma, etc;
- Relação de temperaturas atmosféricas de uma localidade para um determinado período;
- Conjuntos de dados que não apresentam elementos repetidos, como clientes que receberam presente de Natal (podem vir de listas de diferentes vendedores);
- Filas (por exemplo, clinica médica ou supermercado), onde o primeiro a chegar é o primeiro a ser atendido.

Diagrama UML



Incompleto: para estrutura completa, visitar: https://docs.oracle.com/javase/8/docs/api/java/util/Collection.html

Coleções: interfaces

Core Collections: principais coleções

- `Set` (java.util.Set)
- `List` (java.util.List)
- `Queue` (java.util.Queue)
- `Map` (java.util.Map)

Não oferecem nenhuma implementação diretamente: são **interfaces**.

Coleções: implementações

Interface Set (define uma coleção que não contém objetos duplicados). Como o nome sugere, modela a abstração de *conjunto* (matemática)

- `HashSet` é a implementação mais comum.

Interface List (define uma sequência de objetos onde é possível elementos duplicados; inserção qualquer lugar).

- `ArrayList` é a implementação mais comum.

Interface Map (define uma coleção em um algoritmo *hash*).

- `HashMap` é a implementação mais comum.

Interface Queue (define uma coleção que representa uma fila, ou seja, implementa o modelo FIFO – *first-in-first-out*).

- `LinkedList` é a implementação mais comum.

Classes que implementam `java.util.List`

- `ArrayList`: implementação de lista usando para armazenar os dados um *array* redimensionável. Melhor performance para métodos `get` (acesso a elemento) e `set` (alterar);
- `LinkedList`: lista duplamente encadeada (melhor performance p/ métodos `add` e `remove` – inserção e remoção);
- `Stack`: pilha (estrutura LIFO: *last-in-first-out*);
- `Vector`: `ArrayList`, melhorado para trabalhar com **código paralelo**.

Exemplos de lista com ArrayList I

Criação de um `ArrayList` de objetos de uma classe chamada `Aluno`:

```
1 ArrayList<Aluno> alunos;
2 alunos = new ArrayList<Aluno>();
3
4 Aluno a = new Aluno();
5 alunos.add(a);
```

Exemplos de lista com ArrayList II

Utilização de um `ArrayList` na classe Turma:

```
1 public class Turma {
2    private  List<Aluno>  alunos;
3    Turma {
4       alunos = new ArrayList<Aluno>();
5    }
6 }
```


Exemplos de lista com ArrayList III

Outro exemplo:

```
1  import java.util.ArrayList;
2
3  public class Cores {
4     public static void criarCores() {
5        ArrayList<String> cores = new ArrayList<>();
6        cores.add("Vermelho");
7        cores.add("Verde");
8        cores.add("Azul");
9        cores.add("Amarelo");
10       for (int i = 0; i < cores.size(); i++) {
11          String str = cores.get(i);
12          System.out.println(str);
13       }
14       cores.remove(3);
15          cores.remove("Azul");
16       System.out.println("==================");
```

```
17          for (String s : cores) {
18             System.out.println(s);
19          }
20          int indice = cores.indexOf("Vermelho");
21          cores.set(indice, "Preto");
22          System.out.println("=======================");
23          for (String s : cores) {
24             System.out.println(s);
25          }
26       } //Fim do método criarCores()
27
28       public static void main(String args[]) {
29          criarCores();
30       }
31    }
```


Ordenação: `Collections.sort()` I

O *framework* de coleções do Java possui uma classe, também do pacote `java.util`, chamada `Collections` – não confundir com interface `Collection` vista previamente – que oferece, dentre outros métodos, um **método de ordenação**, o método `sort()`.
Basta importar `Collections`, na classe onde irá utilizar a ordenação,
`import java.util.Collections`

Ordenação: `Collections.sort()` II

Exemplo com *strings*:

```
1 List<String> lista = new ArrayList<>();
2 lista.add("verde");
3 lista.add("azul");
4 lista.add("preto");
5
6 System.out.println(lista);
7 Collections.sort(lista);
8 System.out.println(lista);
```


Ordenação: `Collections.sort()` III

No exemplo anterior, `ArrayList` de *strings* foi ordenado.

E se trabalharmos com objetos de outra classe? Como fica a ordenação?

Ordenação: Collections.sort() IV

Exemplo:

```
1  ContaCorrente c1 = new ContaCorrente();
2  c1.deposita(500);
3  ContaCorrente c2 = new ContaCorrente();
4  c2.deposita(200);
5  ContaCorrente c3 = new ContaCorrente();
6  c3.deposita(150);
7
8  List<ContaCorrente> contas = new ArrayList<>();
9  contas.add(c1);
10 contas.add(c2);
11 contas.add(c3);
12
13 Collections.sort(contas); // qual seria o critério para esta
       ordenação?
```


Ordenação: Collections.sort() V

Considere que a classe `ContaCorrente` possui um atributo chamado `saldo` e um método chamado `deposita()`, que altera o valor do saldo.

Neste caso, é preciso **instruir** o método `sort()` sobre como será o critério de ordenação, ou seja, como os elementos serão **comparados**.

Isto será feito *implementando-se a interface* `Comparable` do pacote `java.lang`.

Ordenação: `Collections.sort()` VI

A interface `Comparable` possui um **método abstrato** chamado `compareTo()`, que compara um objeto qualquer em relação a outro, e *retorna um inteiro* de acordo com a comparação:

- < 0, se o objeto que chama o método (`this`) é "menor que" o objeto passado por parâmetro do método;
- 0, se ambos são iguais;
- > 0, se `this` é "maior que" o objeto passado.

O método `sort()` de `Collections` chamará o método `compareTo()` internamente.

Ordenação: `Collections.sort()` VII

```
1  public class ContaCorrente implements Comparable<ContaCorrente>
       {
2      private double saldo;
3
4      // ... demais atributos, e outros métodos ...
5
6      public int compareTo(ContaCorrente outra) {
7          if (this.saldo < outra.saldo) {
8              return -1;
9          } else if (this.saldo > outra.saldo) {
10             return 1;
11         } else {
12             return 0; // saldos iguais
13         }
14     }
15 }
```

Outros métodos de java.util.Collection

- 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.

Outros métodos, e mais detalhes sobre a classe Collections: https://docs.oracle.com/javase/8/docs/api/java/util/Collections.html

EDs e respectivas classes

Interface	Tabela *hash*	Lista estática (*array*)	Árvore balanceada	Lista encadeada	Tabela *hash* com lista encadeada
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Queue		ArrayDeque		LinkedList	
Map	HashMap		TreeMap		LinkedHashMap

Referências

1. Apostila de Java e POO Caelum: disponível em https://www.caelum.com.br/download/caelum-java-objetos-fj11.pdf – acesso em: MAI/2017.
2. Documentação Java Oracle: https://docs.oracle.com/javase/tutorial/java/IandI/createinterface.html, https://docs.oracle.com/javase/8/docs/technotes/guides/collections/index.html

Os slides de parte desta seção foram cedidos por Marcelo Z. do Nascimento, FACOM/UFU

LaTeXagem e adaptações: Renato Pimentel, FACOM/UFU