INSTITUTO FEDERAL DE
EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
BAHIA

Collections Framework

Arrays

INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA BAHIA

- Manipular **array** é bastante **trabalhoso**.
- **Dificuldades** aparecem 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**.
- Para suprir essas **dificultdades** existe na linguagem **Java** um conjunto de **classes** e **interfaces** conhecido como **Collections Framework;**

Collections

INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA BAHIA

- A **API** do **Java Collections Framework (JFC)** 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 a **Sun** disponibilizou.

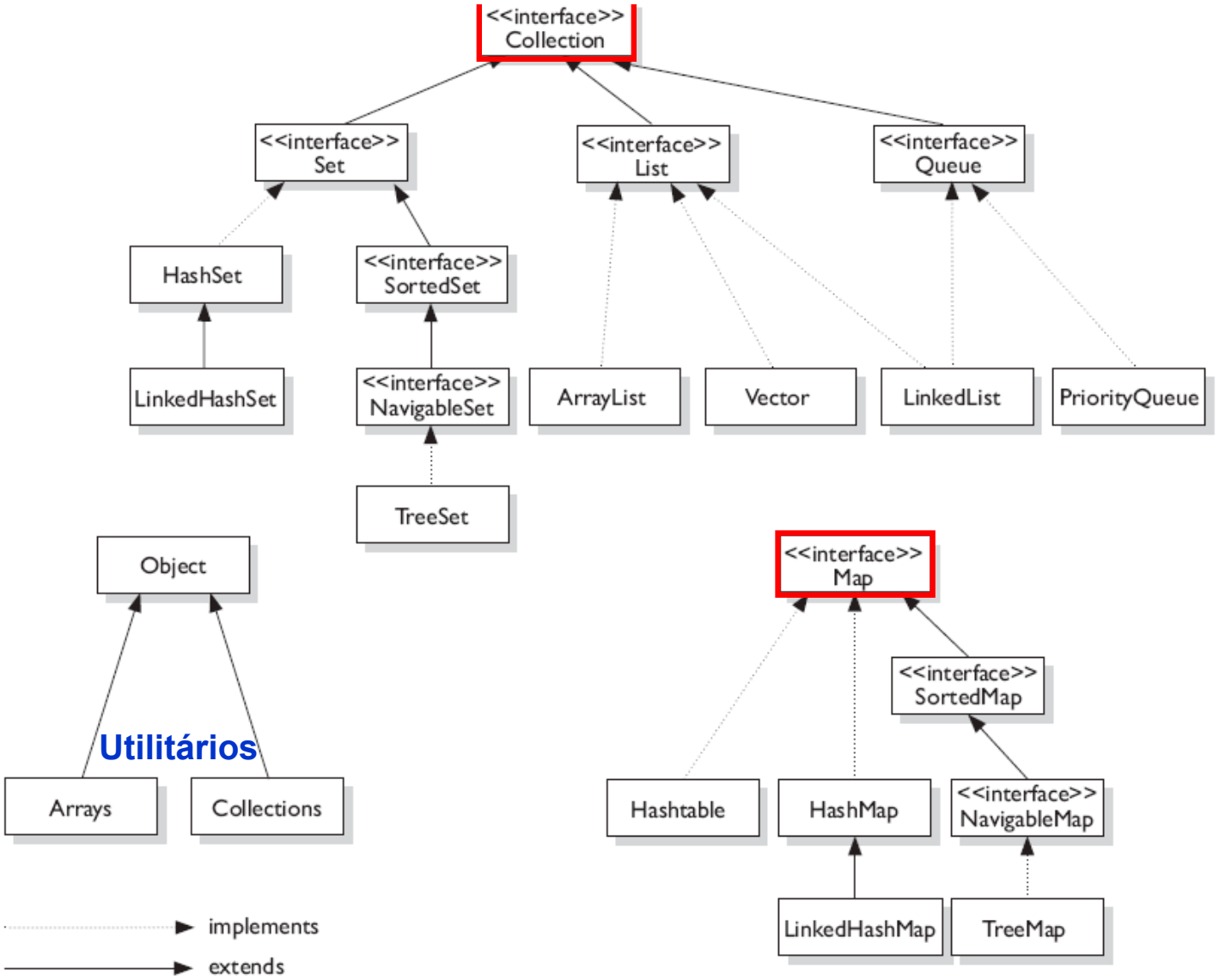
Collections

INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA BAHIA

- **Java Collections Framework (JCF)**
  - Existem duas **interfaces principais** são elas:
    - **java.util.Collection:** uma **coleção** de **objetos**
    - **java.util.Map:** uma **coleção** de **chave/objeto**

- Toda a **estrutura** da **JCF** é **baseada** e **descendem** da **estrutura** destas duas **interfaces.**

Collections



Listas:java.util.List

- Uma lista é uma **coleção** que **permite elementos duplicados** e mantém uma **ordenação específica** entre os elementos.
- Ela resolve **todos os problemas** que levantamos em relação ao **array** (busca, remoção, tamanho "infinito",...).
- 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**.
- A implementação **mais utilizada da interface List** é a **ArrayList**, que trabalha com um **array interno** para gerar uma lista.

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");
```

- 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**.

ArrayList

- **Toda lista** (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**.

- **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, podemos usar o **método size()**:

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

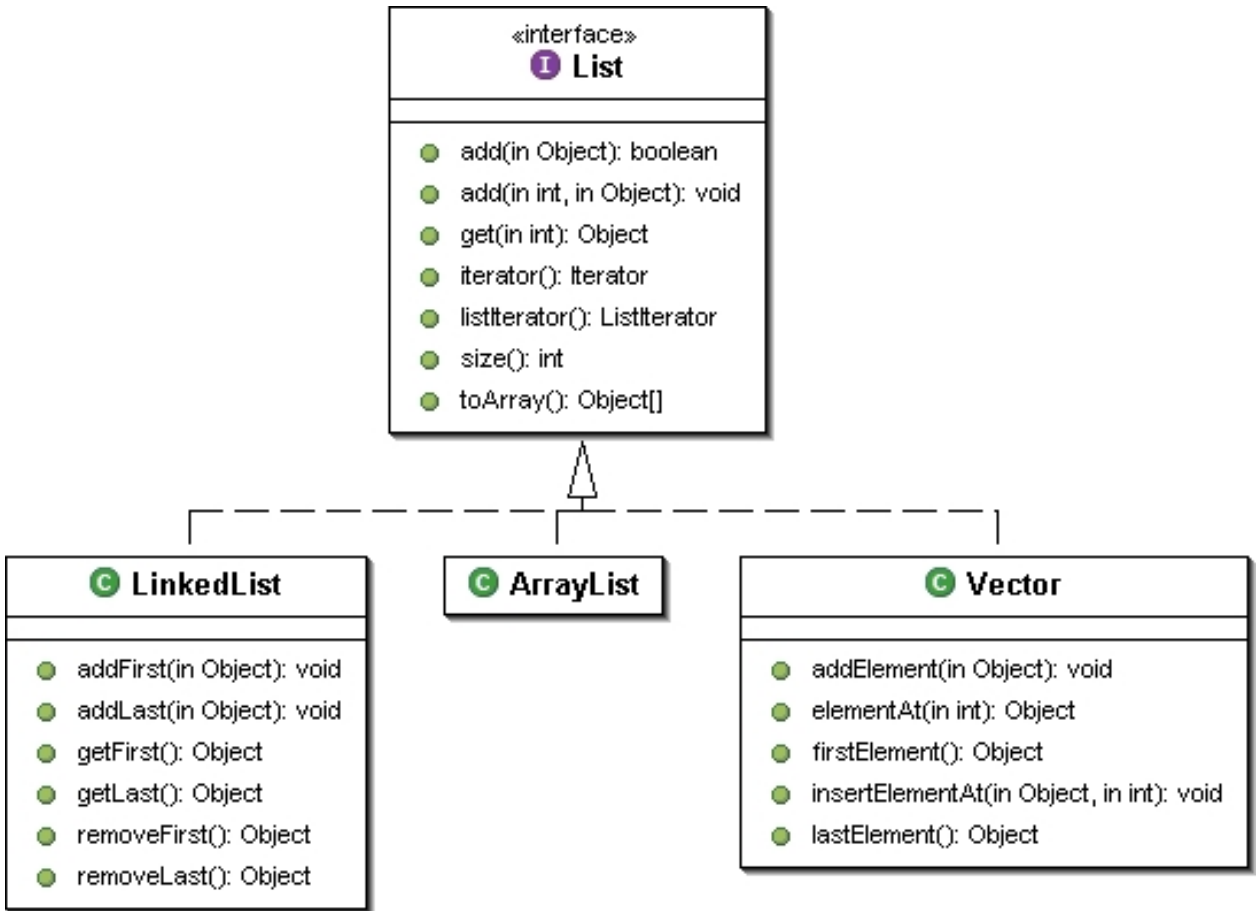
- O **método get(int)** recebe como **argumento** o **índice do elemento** que se quer **recuperar**. Assim podemos fazer um **for** para **iterar** na lista de **contas**:

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

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

Interface List

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



LinkedList

- A outra **implementação** muito usada (**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**;

Lista 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);
...
```

- 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**;

Lista com Generics

INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA BAHIA

- Geralmente, **não nos interessa** uma lista com **vários tipos** de **objetos misturados**;
- No **Java 5.0**, 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<ContaCorrente>();
contas.add(c1);
contas.add(c3);
contas.add(c2);
```

- 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!!
```

Lista com Generics

INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA BAHIA

- 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());
}
```

Importância das interfaces

- 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<Conta>();

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

        return contas;
    }
}
```

Importância das interfaces

INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA BAHIA

- O ideal é **sempre** trabalhar com a **interface mais genérica** possível:

```
class Agencia {

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

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

        return contas;
    }
}
```

- Assim como no **retorno**, é boa **prática** trabalhar com a **interface** em todos os **lugares possíveis**:

```
class Agencia {

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

Importância das interfaces

- Também declaramos **atributos** como **List** em vez de nos **comprometer** como uma ou outra **implementação**.

- Dessa forma **obtemos** um **baixo acoplamento:** podemos **trocar** a **implementação**, já que estamos **programando** para a **interface**!

```
class Empresa {

    private List<Funcionario> empregados = new ArrayList<Funcionario>();

    // ...
}
```

Ordenação: Collections.sort

- Listas são **percorridas** de maneira **pré-determinada** de acordo com a **inclusão dos itens.**

- E se **quisermos** **percorrer** 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**.

```
List lista = new ArrayList();
lista.add("Sérgio");
lista.add("Paulo");
lista.add("Guilherme");

System.out.println(lista); //repare que o toString de ArrayList foi sobrescrito!

Collections.sort(lista);

System.out.println(lista);
```

Ordenação: Collections.sort

INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA BAHIA

- 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**?

```
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<ContaCorrente>();
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**.

Ordenação: Collections.sort

INSTITUTO FEDERAL DE EDUCAÇÃO, CIÊNCIA E TECNOLOGIA BAHIA

- Como 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**!
- 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 codigo anterior fica aqui

 public int compareTo(ContaCorrente outra) {

   if (this.saldo < outra.saldo) {
     return -1;
   }

   if (this.saldo > outra.saldo) {
     return 1;
   }

   return 0;
 }
}
```

- 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.

Exercício