Orientação a Objetos em Java

9ª aula

Prof. Douglas Oliveira
douglas.oliveira@prof.infnet.edu.br

Classe Object

- Quando o compilador Java compila uma classe que não estende nenhuma outra classe, ele inclui automaticamente a cláusula:

```
extends Object
```

Classe Object

- Por exemplo, a classe abaixo :

```
public class Disciplina {
    private String codigo;
    private String nome;
}
```

é compilada como:

```
public class Disciplina extends Object {
    private String codigo;
    private String nome;
}
```

Classe Object

- Assim, a classe Object é a super classe de todas as classes.
- Logo, uma variável do tipo Object pode referenciar qualquer objeto:

```
Object obj;

obj = new Aluno(123, "João");

obj = new int[10];

obj = new Retangulo(1, 1, 10, 20);
```

Classe Object

- A classe Object implementa alguns métodos importantes:
  - ✔ boolean equals(Object obj): retorna *true* se o objeto corrente (this) for igual ao objeto referenciado por obj. Esse método é chamado automaticamente por vários métodos da API do Java quando é necessário verificar se dois objetos são iguais.
  - ✔ String toString(): permite retornar uma descrição textual do objeto. Esse método é chamado automaticamente quando é preciso converter um objeto em uma String.

Classe Object

- Pelo princípio do polimorfismo, podemos sobrescrever esses métodos em nossas classes, de forma que eles sejam implementados de uma forma específica.
- Exemplo: implemente a classe Aluno contendo a matrícula e nome do aluno. Implemente também os métodos:
  - ✔ boolean equals(Object obj): para verificar a igualdade entre dois alunos;
  - ✔ String toString(): gerar uma descrição textual do aluno com matrícula e nome;

Classe Object

```
public class Aluno {
  private int matricula;
  private String nome;
  public Aluno(int matricula, String nome) {
     this.matricula = matricula;
     this.nome = nome;
  }


  @Override
  public boolean equals(Object obj) {
     if (obj == null)
        return false;
     if (! (obj instanceof Aluno))
        return false;
     if (this == obj)
        return true;

     return (this.matricula == ((Aluno)obj).matricula)
  }
```

```
     @Override
     public String toString() {
         return String.format("Matricula: %d
                 Nome: %s",
                 matricula, nome);
     }
}
```

Classe Object - equals

```
@Override
public boolean equals(Object obj) {

    if (obj == null)
        return false;

    if (! (obj instanceof Aluno))
        return false;

    if (this == obj)
        return true;

    return (this.matricula == ((Aluno)obj).matricula)
}
```

1-Verifica se o parâmetro passado é NULL.

Por que esse teste é necessário?

Classe Object - equals

- Esse teste é necessário porque é possível passar NULL como parâmetro de equals, e não faz sentido comparar um objeto com NULL.

```
public class TesteAluno {
  public static void main(String[] args) {
     Aluno a = new Aluno(100, "João da silva");


     if (a.equals(null))
        System.out.println("VERDADEIRO? Não faz sentido...");
     else
           System.out.println("FALSO. Esse é o resultado esperado");
  }
}
```

Compara aluno **a** com NULL

Classe Object - equals

```
@Override
public boolean equals(Object obj) {
    if (obj == null)
        return false;

    if (! (obj instanceof Aluno))
        return false;

    if (this == obj)
        return true;

    return (this.matricula == ((Aluno)obj).matricula)
}
```

2-Verifica se obj é um Aluno. Lembre-se que obj pode referenciar qualquer objeto.

Por que esse teste é necessário?

Classe Object - equals

- Esse teste é necessário porque é possível passar qualquer objeto como parâmetro de equals, e não faz sentido comparar um objeto com outro de um tipo diferente.

```
public class TesteAluno {
  public static void main(String[] args) {
     Aluno a = new Aluno(100, "João da silva");
     Disciplina d = new Disciplina("INF100", "POO");
     if (a.equals(d))
        System.out.println("VERDADEIRO? Não faz sentido...");
     else
           System.out.println("FALSO. Esse é o resultado esperado");
  }
}
```

Compara aluno **a** com disciplina **d** !

Classe Object - equals

```
@Override
public boolean equals(Object obj) {
    if (obj == null)
        return false;

    if (! (obj instanceof Aluno))
        return false;

    if (this == obj)
        return true;

    return (this.matricula == ((Aluno)obj).matricula)
}
```

3-Verifica se um objeto está sendo comparado com ele mesmo.

Para que serve isso?

Classe Object - equals

- Esse teste não é estritamente necessário, mas pode ser implementado para não perder tempo comparando um objeto com ele mesmo.

```
public class TesteAluno {
  public static void main(String[] args) {
     Aluno a = new Aluno(100, "João da silva");

     if (a.equals(a))
       System.out.println("VERDADEIRO. Esse é o resultado esperado");
     else
       System.out.println("FALSO? Não faz sentido...");
  }
}
```

Compara aluno **a** com ele mesmo !

Classe Object - equals

```
@Override
public boolean equals(Object obj) {
    if (obj == null)
        return false;
    if (! (obj instanceof Aluno))
        return false;
    if (this == obj)
        return true;

    return (this.matricula == ((Aluno)obj).matricula)
}
```

Por fim, compara um Aluno com outro usando a **matrícula**.

Classe Object - equals

- Esse teste não é estritamente necessário, mas pode ser implementado para não perder tempo comparando um objeto com ele mesmo.

```
public class TesteAluno {
  public static void main(String[] args) {
     Aluno a = new Aluno(100, "João da Silva");
     Aluno b = new Aluno(200, "Ana de Almeida");
     Aluno c = new Aluno(100, "João da Silva");
     if (a.equals(b))
       System.out.println("VERDADEIRO. Matrículas iguais. ");
     else
       System.out.println("FALSO. Matrículas diferentes!");
    if (a.equals(c))
       System.out.println("VERDADEIRO. Matrículas iguais.");
     else
       System.out.println("FALSO. Matrículas diferentes! ");
  }
```

Classe Object - toString

```
@Override
public String toString() {
    return String.format("Matricula: %d  Nome: %s",
                         matricula, nome);
}
```

Retorna uma string formada pela **matrícula** e **nome** do aluno.

- Observação:
  - ✓ O método String.format() é idêntico ao método System.out.printf() e deve ser usado da mesma forma.
  - ✓ A diferença é que System.out.printf() imprime a string formatada e String.format() retorna a string formatada.

Classe Object

```
public class TestaAluno{
    public static void main(String[] args) {
        Aluno a1, a2;
        a1 = new Aluno(1, "Joao");

        a2 = new Aluno(2, "Maria");

        System.out.println(a1);
        System.out.println("Aluno: " + a2);
        System.out.println(a1.toString() + "\n" +
                           a2.toString());
}
```

Nesses casos o método **toString()** é chamado automaticamente para converter o objeto em uma String.

Exercícios

- Exercício 41: implemente os métodos equals() e toString() na classe Data.
- Exercício 42: implemente os métodos equals() e toString() na classe Carro. Um carro é igual a outro se tem o mesmo modelo e mesmo motor. Um motor é igual a outro se tem a mesma cilindrada.

Interfaces

- Interface é um recurso muito utilizado em Java.
- Boa parte da API do Java é formada por Interfaces.
- A definição de uma Interface é muito parecida com a de uma classe, só que ela define somente os métodos sem implementá-los !

```
public interface Autenticavel {
    public boolean autenticar(String login, String senha);
}
```

- Assim como as classes, uma interface deve estar em um arquivo próprio, com o mesmo nome da interface.

Interfaces

- De uma forma geral, pode-se dizer que uma Interface define um contrato.
- Assim, as classes que assinam esse contrato se comprometem a implementar esses métodos (ou seja, cumprir o contrato).
- Em outras palavras, uma Interface "obriga" uma classe a implementar o conjunto de métodos definidos nela.
  - ✔ Note que esse conceito é bastante semelhante ao conceito de métodos abstratos.

Interfaces

- Uma classe implementa uma interface através da palavra implements.

```
public class Cliente implements Autenticavel {
}
public class Funcionario implements Autenticavel {

}
```

- Nesse exemplo, a classe Cliente e a classe Funcionário implementam a interface Autenticável.

Interfaces

- Ao implementar um interface, a classe é obrigada a implementar todos os métodos definidos na interface (semelhante ao conceito de método abstrato).

```
public class Cliente implements Autenticavel {
    public boolean autenticar(String login, String senha) {
      // aqui escreve o código que implementa o método
      // para a classe Cliente
    }
}
public class Funcionario implements Autenticavel {
    public boolean autenticar(String login, String senha) {
      // aqui escreve o código que implementa o método
      // para a classe Funcionario
    }
}
```

Interfaces

- Importante:
  - ✔ Uma classe pode implementar várias interfaces. É só separar o nomes das várias interfaces por vírgula.

```
public interface Climatizado {
  public void ligarArCondicionado();
  public void desligarArCondicionado();
}
public interface Sonorizado {
  public void ligarSom();
  public void desligarSom();
}
public class Carro implements Climatizado, Sonorizado {
  // Tem que implementar aqui os 4 métodos !
}
```

Interfaces

- Importante:
  - ✔ É permitido que uma mesma classe use o extends e o implements ao mesmo tempo.

```
public interface Turbinado {
  public void ligarTurbo();
  public void desligarTurbo();
}
public class CarroEsportivo extends Carro implements Turbinado {
  // Tem que implementar aqui os 2 métodos de Turbinado!
}
```

Interfaces

- Importante:
  - ✔ Não é possível criar objetos usando uma interface, pois interfaces não são classes.

```
Autenticavel a;
a = new Autenticavel();
```

Erro de compilação !

Interfaces

- Importante:
  - ✔ É possível ter variáveis do tipo da interface referenciando objetos de classes que implementam essa interface.

```
Autenticavel a;
a = new Cliente();
a = new Funcionario();
```

A variável **a** pode referenciar um Cliente ou Funcionario, porque as classes Cliente e Funcionario implementam a interface Autenticavel.

```
Sonorizado s;
s = new Carro();
```

A variável **s** pode referenciar um Carro, porque a classe Carro implementa a interface Sonorizado.

Interfaces

- Importante:
  - ✔ É possível chamar métodos usando variáveis do tipo da interface. Os métodos chamados devem ser aqueles definidos na interface.
  - ✔ Nesse caso, o mecanismo de polimorfismo funciona da mesma forma que na herança:

```
Autenticavel a;
a = new Cliente();
a.autenticar("Joao", "123");

a = new Funcionario()
a.autenticar("Ana", "456");
```

Chama o método **autenticar()** do Cliente

Chama o método **autenticar()** do Funcionario

Coleções

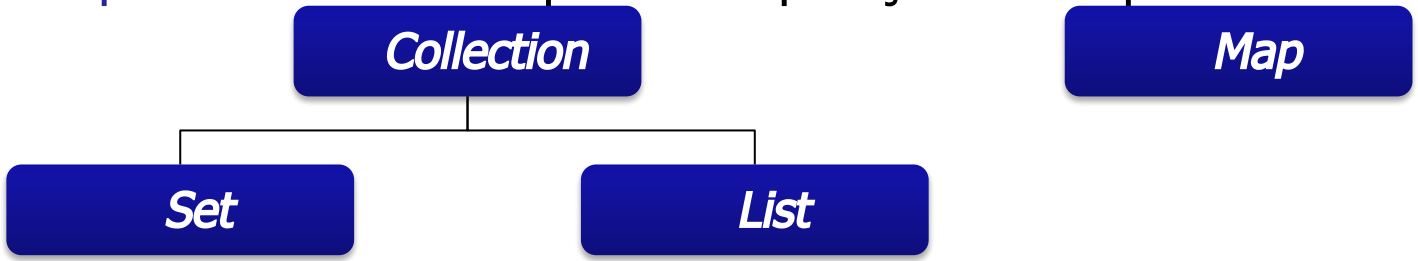
- A linguagem Java possui um conjunto de classes que servem para armazenar, na memória, coleções de objetos.
- Tais classes possuem a vantagem de não termos que saber, de antemão, a quantidade de elementos que iremos armazenar (que é uma grande desvantagem dos vetores).
- Dependendo de como declaramos nossas coleções, podemos armazenar objetos de diversos tipos de objetos em uma mesma coleção.
- Todas as coleções estão definidas no pacote java.util.

Coleções

- As coleções em Java são divididas em 3 grandes grupos:
  - ✔ Set: coleções de objetos que representam conjuntos de objetos.
  - ✔ List: coleções de objetos que representam listas de objetos.
  - ✔ Map: coleções de objetos que representam mapas ou dicionários.

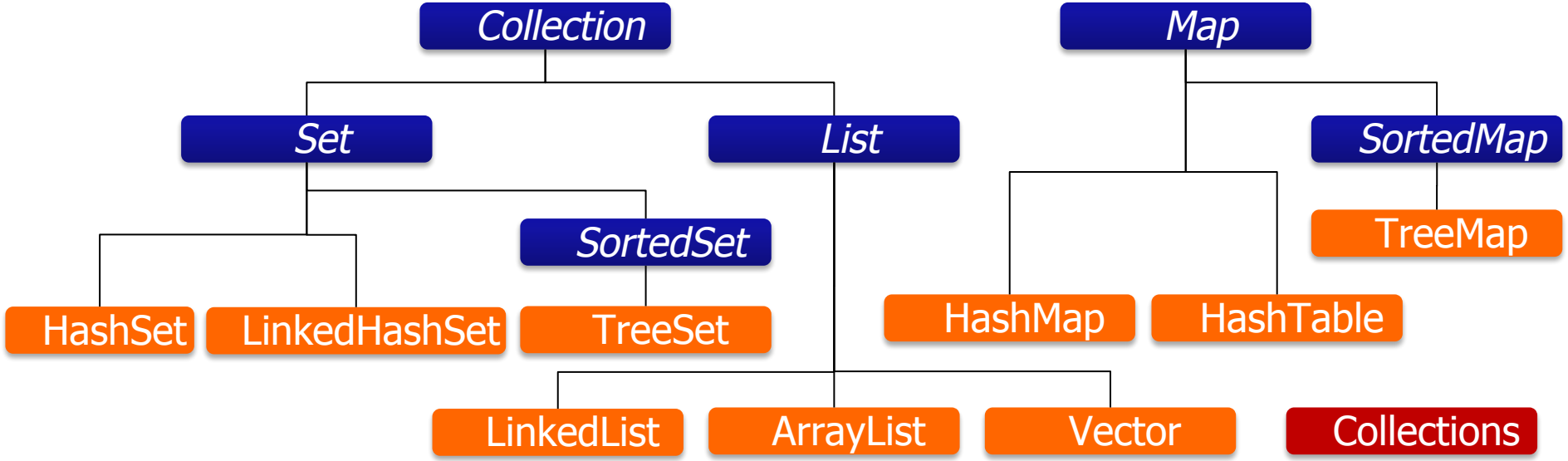
Coleções

- Essas coleções são definidas a partir de 4 interfaces principais:
  - ✔ Collection: define métodos comuns a conjuntos e listas.
  - ✔ Set: define métodos para manipulação de conjuntos de objetos.
  - ✔ List: define métodos para manipulação de listas de objetos.
  - ✔ Map: define métodos para manipulação de mapas ou dicionários.



Coleções

- O diagrama abaixo representa as principais classes que implementam essas interfaces, além da classe Collections.



Coleções - Collection

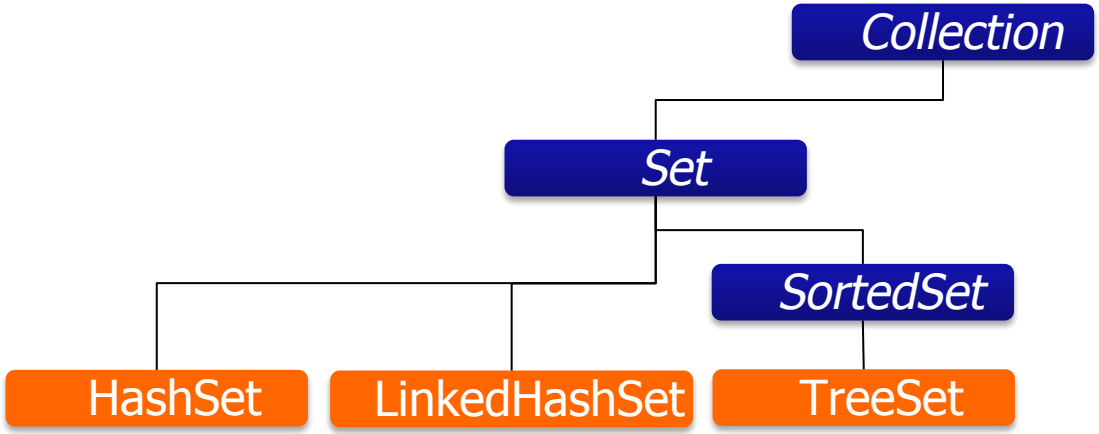
- A interface Collection define vários métodos básicos para manipulação de conjuntos e listas de objetos:
  - ✔ boolean add(Object): adiciona um objeto à coleção. Retorna *true* ou *false* para indicar se a operação foi bem sucedida ou não.
  - ✔ boolean remove(Object): remove o objeto especificado da coleção. Retorna *false* se o objeto não pertence à coleção.
  - ✔ boolean contains(Object): procura por um determinado objeto na coleção e retorna *true* ou *false*, se o objeto existir ou não. A comparação é feita pelo método equals().

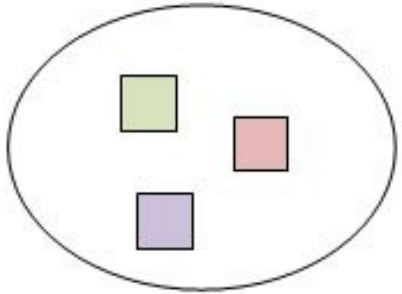
Coleções - Collection

- A interface Collection define vários métodos básicos para manipulação de conjuntos e listas de objetos:
  - ✔ int size(): retorna a quantidade de objeto presentes na coleção.
  - ✔ boolean isEmpty(): retorna *true* se a coleção está vazia ou *false* caso contrário.
  - ✔ void clear(): remove todos os objetos da coleção.

Coleções - Set

- Representa a mesma ideia de conjuntos da matemática, ou seja, um grupo de objetos sem duplicidade de elementos.
- Assim, se tentarmos armazenar mais de uma vez o mesmo objeto em um Set, não teremos o objeto duplicado.



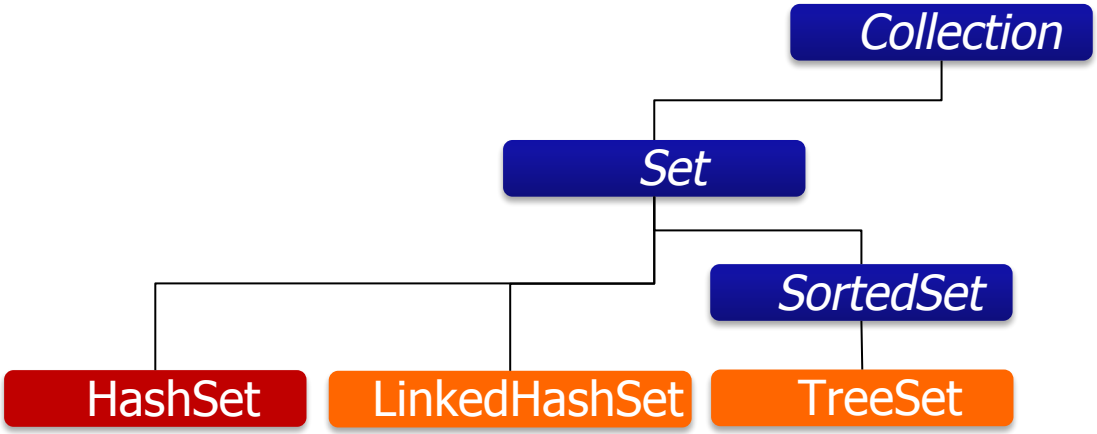


Coleções - Set

- Um conjunto pode ser mantido não ordenado (Set) ou ordenado (SortedSet).
- Em um conjunto não ordenado, ao recuperarmos os objetos não é possível prever em que ordem eles serão retornados (pode ser a mesma ordem em que foram armazenados ou não).
- Em um conjunto ordenado os objetos são sempre recuperados de forma ordenada. Um ou mais atributos do objeto podem ser usados para definir essa ordem.
- As interfaces Set e SortedSet não acrescentam nenhum método além daqueles já herdados da interface Collection.

Coleções - HashSet

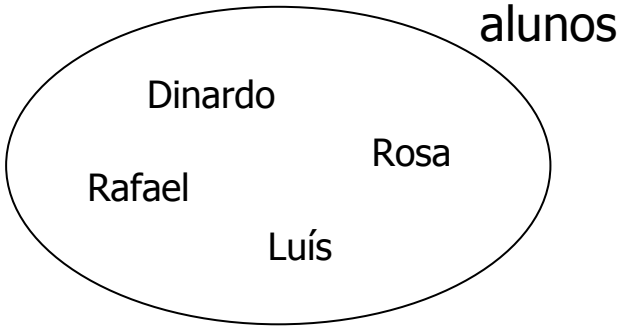
- HashSet é uma classe concreta que implementa a interface Set, ou seja, representa um conjunto matemático sem ordenação.



Coleções - HashSet

- Exemplo 1: não insere duplicatas

```
import java.util.HashSet;
public class ExemploSet {

  public static void main(String[] args) {
      HashSet alunos = new HashSet();
      alunos.add("Dinardo");
      alunos.add("Rosa");
      alunos.add("Dinardo");
      alunos.add("Rafael");
      alunos.add("Luís");
      alunos.remove("Rosa");
  }
}
```

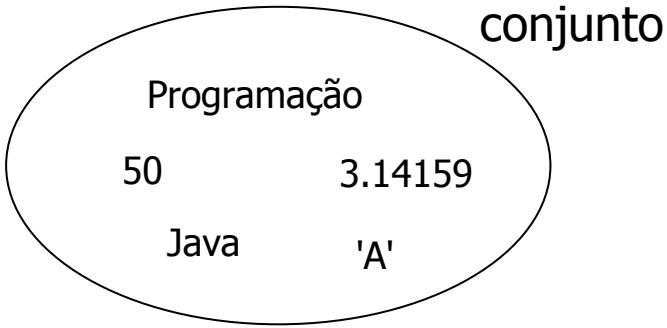


Coleções - HashSet

- Exemplo 2: aceita qualquer objeto

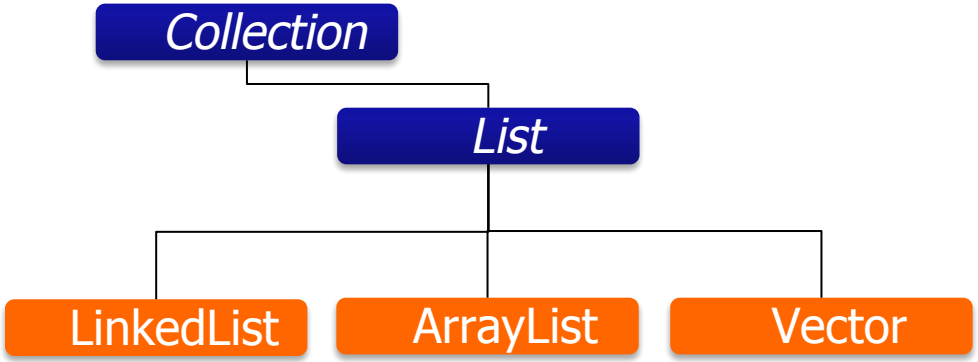
```
import java.util.HashSet;

public class ExemploSet {

  public static void main(String[] args) {
      HashSet conjunto = new HashSet();

      conjunto.add("Programação");
      conjunto.add(50);
      conjunto.add(3.14159f);
      conjunto.add('A');
      conjunto.add("Java");

      conjunto.remove("C++");
      conjunto.remove(50);
  }
}
```

Definido dessa forma, o Set armazena objetos do tipo **Object**.



Coleções - List

- Representa uma lista de objetos com a possibilidade de duplicidade de elementos.
- Assim, se tentarmos armazenar mais de uma vez o mesmo objeto em um List, ele ficará duplicado na lista.



- As listas mantém a ordem em que os elementos foram adicionados, permitindo que esses sejam recuperados de forma indexada, ou seja, através de um índice.

Coleções - List

- A interface List define vários métodos importantes:
  - ✔ void add(índice, Object): adiciona o objeto à coleção na posição do índice.
  - ✔ Object get(índice): recupera o objeto de determinada posição da lista (da mesma forma como é feito com vetores).
  - ✔ Object set(índice, Object): substitui o objeto da posição do índice pelo novo objeto. Retorna o objeto que estava armazenado anteriormente.
  - ✔ Object remove(índice): remove o objeto de determinada posição da lista. Retorna o objeto removido.

Coleções - List

- A interface List define vários métodos importantes:
  - ✔ int indexOf(Object): retorna o índice da primeira ocorrência de um objeto ou –1 se ele não existir na lista. A comparação é feita pelo método equals().
  - ✔ int lastIndexOf(Object): retorna o índice da última ocorrência de um objeto ou –1 se ele não existir na lista. A comparação é feita pelo método equals().

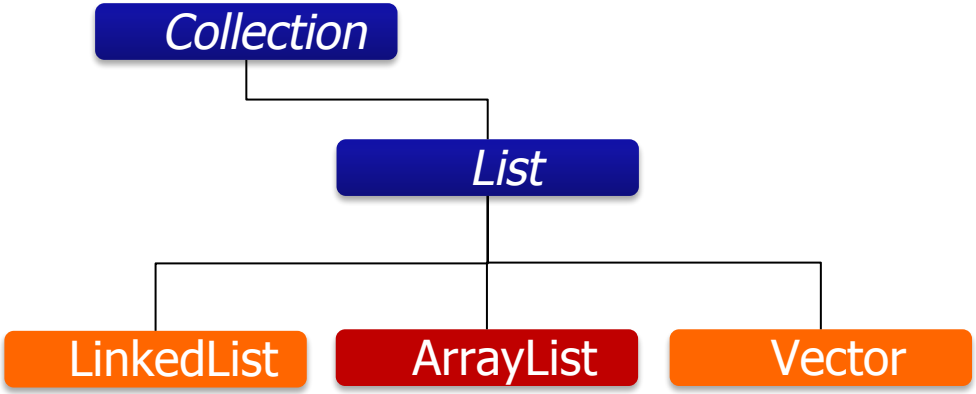
Coleções - List

- É importante notar o efeito dos métodos na lista de objetos:

Método	Descrição
boolean add(Object)	Adiciona o objeto sempre no final da lista.
void add(índice, Object)	Adiciona um objeto na i-ésima posição e move os objetos subsequentes para a posição posterior (índice deve ter valor de 0 a size()).
boolean remove(Object)	Remove o objeto da lista e move os objetos subsequentes para a posição anterior.
Object remove(índice)	Remove o objeto da i-ésima posição da lista e move os objetos subsequentes para a posição anterior (índice deve ter valor de 0 a size()-1).
Object set(índice, Object)	Substitui o elemento da i-ésima posição pelo novo objeto (índice deve ter valor de 0 a size()-1).

Coleções - ArrayList

- O ArrayList é uma classe concreta que implementa a interface List, ou seja, uma lista de objetos.
- Cada objeto armazenado no ArrayList possui um índice e através desse índice, é possível manipular esse objeto.



Coleções - ArrayList

- Exemplo 1: adicionando e removendo elementos da lista.

```
import java.util.ArrayList;
public class ExemploArrayList {

  public static void main(String[] args) {

      ArrayList lista = new ArrayList();
      lista.add("Dinardo");
      lista.add("Rosa");
      lista.add("Dinardo");
      lista.add("Rafael");
      lista.add(2, "Carlos");
      lista.add("Luís");
      lista.remove("Carlos");
      lista.remove("Rafael");
      lista.remove("Dinardo");
      lista.remove(2);
  }
}
```

Cria uma lista vazia.

Insere no fim da lista.

Insere em uma posição específica.

Remove o objeto.

Remove o objeto dessa posição.

Coleções - ArrayList

- Exemplo 2: podemos armazenar objetos de tipos diferentes.

```
import java.util.ArrayList;
public class ExemploArrayList {

  public static void main(String[] args) {

      ArrayList lista = new ArrayList();

      lista.add("Dinardo");
      lista.add("Rosa");
      lista.add(10);
      lista.add(2465);
      lista.add(3.14159);
      lista.add('A');
  }
}
```

Definido dessa forma, o ArrayList armazena objetos do tipo **Object**.

Coleções - ArrayList

- Exemplo 3: podemos recuperar um elemento da lista pelo seu índice.

```
import java.util.ArrayList;
public class ExemploArrayList {
  public static void main(String[] args) {
      ArrayList lista = new ArrayList();
      lista.add("Dinardo");
      lista.add("Rosa");
      lista.add(10);
      lista.add(2465);
      lista.add(3.14159);
      lista.add('A');

      System.out.println(lista.get(0));
      System.out.println(lista.get(3));
      System.out.println(lista.get(4));
      System.out.println(lista.get(5));
  }
}
```

O método **get(i)** recupera o elemento da i-ésima posição.

A primeira posição é ZERO, como no vetor.

Coleções - ArrayList

- Exemplo 4: podemos alterar um elemento da lista pelo seu índice.

```
import java.util.ArrayList;
public class ExemploArrayList {
  public static void main(String[] args) {
      ArrayList lista = new ArrayList();
      lista.add("Dinardo");
      lista.add("Rosa");
      lista.add(10);
      lista.add(2465);
      lista.add(3.14159);
      lista.add('A');
      lista.set(1, 200);
      lista.set(2, "Andre");
  }
}
```

O método **set(i, obj)** altera o elemento da i-ésima posição.

Coleções - ArrayList

- Para percorrer um ArrayList podemos adotar uma das 3 estratégias:

1. for
2. for each
3. Iterator ou ListIterator

Coleções - ArrayList

- Exemplo 5: usando um for tradicional.

```
import java.util.ArrayList;
public class ExemploArrayList {
  public static void main(String[] args) {
      ArrayList lista = new ArrayList();
      lista.add("Dinardo");
      lista.add("Rosa");
      lista.add(10);
      lista.add(2465);
      lista.add(3.14159);
      lista.add('A');
      for (int i = 0; i < lista.size(); i++)
          System.out.println(lista.get(i));
  }
}
```

Coleções - ArrayList

- Exemplo 6: usando um for each.

```
import java.util.ArrayList;
public class ExemploArrayList {
   public static void main(String[] args) {
      ArrayList lista = new ArrayList();
      lista.add("Dinardo");
      lista.add("Rosa");
      lista.add(10);
      lista.add(2465);
      lista.add(3.14159);
      lista.add('A');
      for (Object obj : lista)
          System.out.println(obj);
  }
}
```

Exercícios

- Exercício 43: leia strings e armazene-as em uma lista até que o usuário digite uma string vazia. Não permita que sejam armazenadas strings duplicadas na lista. Ao final, imprima a lista na ordem inversa.

- Exercício 44: leia uma string e imprima quantas vezes as letras de A a Z aparecem na string. Dica: use uma lista para armazenar as letras lidas e outra lista para contar quantas vezes as letras aparecem.