



Interfaces (e exemplos Comparable, Comparator)



Prog Orientada a Objetos
Prof. João Paulo A. Almeida
Baseado em slides do Prof. Vítor Souza
(usado com permissão)

- Uma classe **abstrata** é **pura** quando:
 - Possui métodos **abstratos**;
 - Não possui métodos **concretos**;
 - Não possui **atributos**.
- Java **oferece** a palavra reservada **interface**:
 - Cria uma classe **abstrata pura**;
 - Chamaremos pelo nome de **interface**;
 - Ao conversar com outros programadores, **cuidado** para não **confundir** com “interface com o usuário”.

- Estabelece a interface (o contrato) de um conjunto de classes;
- Permite a construção de código genérico:
 - Trabalha com qualquer objeto que implemente a interface;
 - Obriga programadores a implementar determinados métodos em suas classes para usar seu código.
- Classes utilizam implements ao invés de extends para implementar uma interface.

Interfaces

```
interface Forma {  
    double getPerimetro();  
}  
  
abstract class Poligono implements Forma {  
    private int nLados;  
    public Poligono(int nLados) {  
        this.nLados = nLados;  
    }  
    public int getNumeroLados() { return nLados; }  
    public abstract double getPerimetro();  
}
```

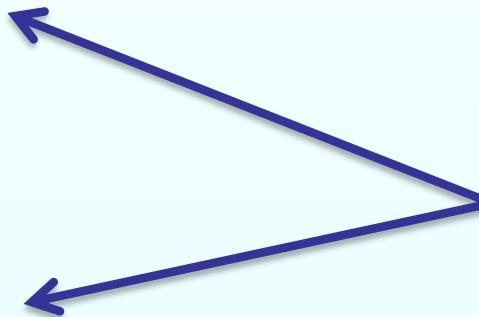
Interfaces

```
class SegmentoDeLinha implements Forma {  
    private Ponto v1, v2;  
  
    public double getPerimetro()  
    {  
        return v1.distancia(v2);  
    }  
}
```

Membros da interface

- Métodos definidos na interface são **automaticamente públicos**;
- Atributos definidos na interface são **automaticamente públicos e estáticos**.

```
interface Forma {  
    int x;  
    void desenhar();  
}  
  
interface Forma {  
    public static int x;  
    public void desenhar();  
}
```



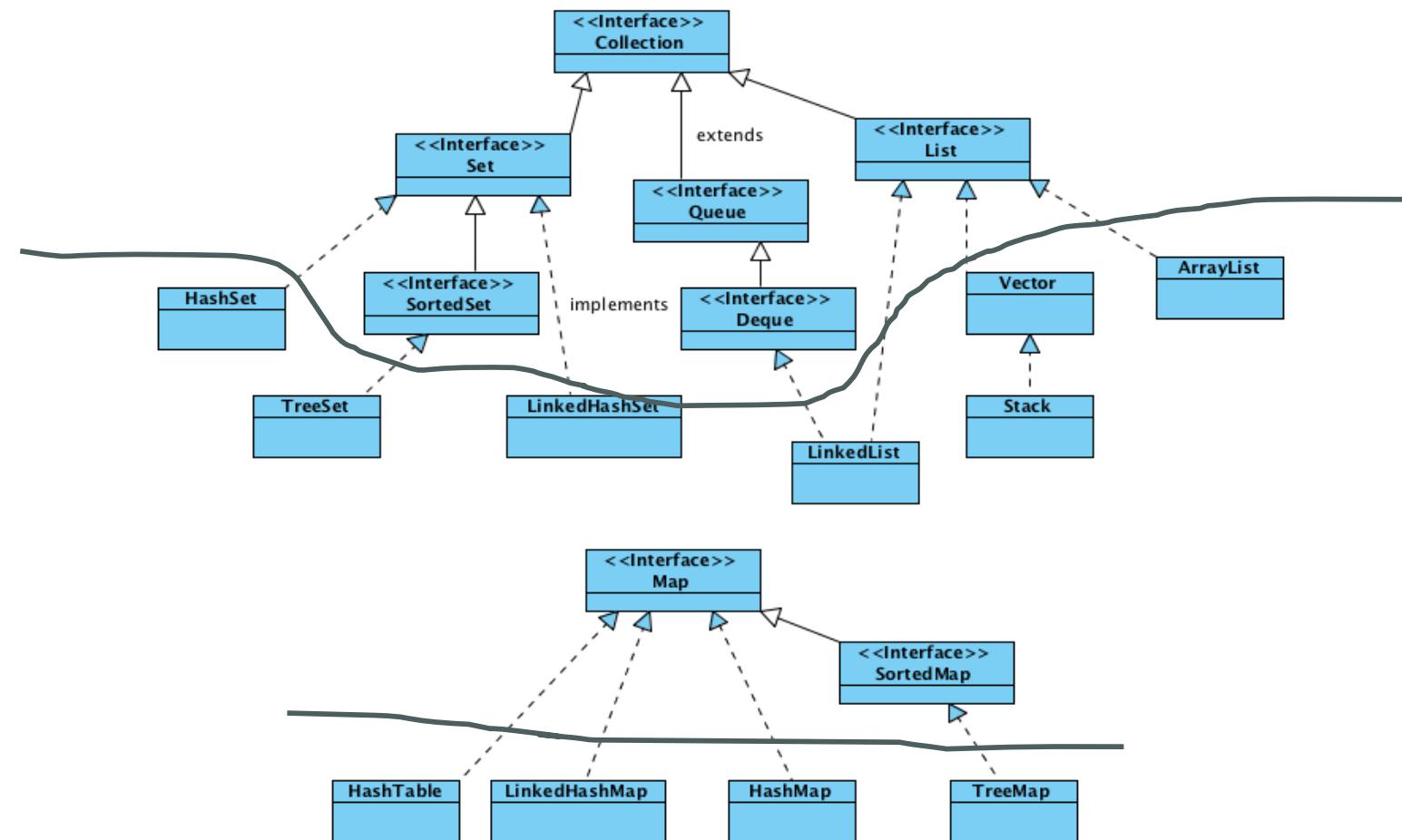
Definições
equivalentes

Membros da interface

```
interface Forma {  
    void desenhar();  
    void aumentar(int t);  
}  
  
class Linha implements Forma {  
    // Erro: reduziu de público para package-private!  
    void desenhar() {  
        /* ... */  
    }  
  
    // Erro: reduziu de público para privado!  
    private void aumentar(int t) {  
        /* ... */  
    }  
}
```

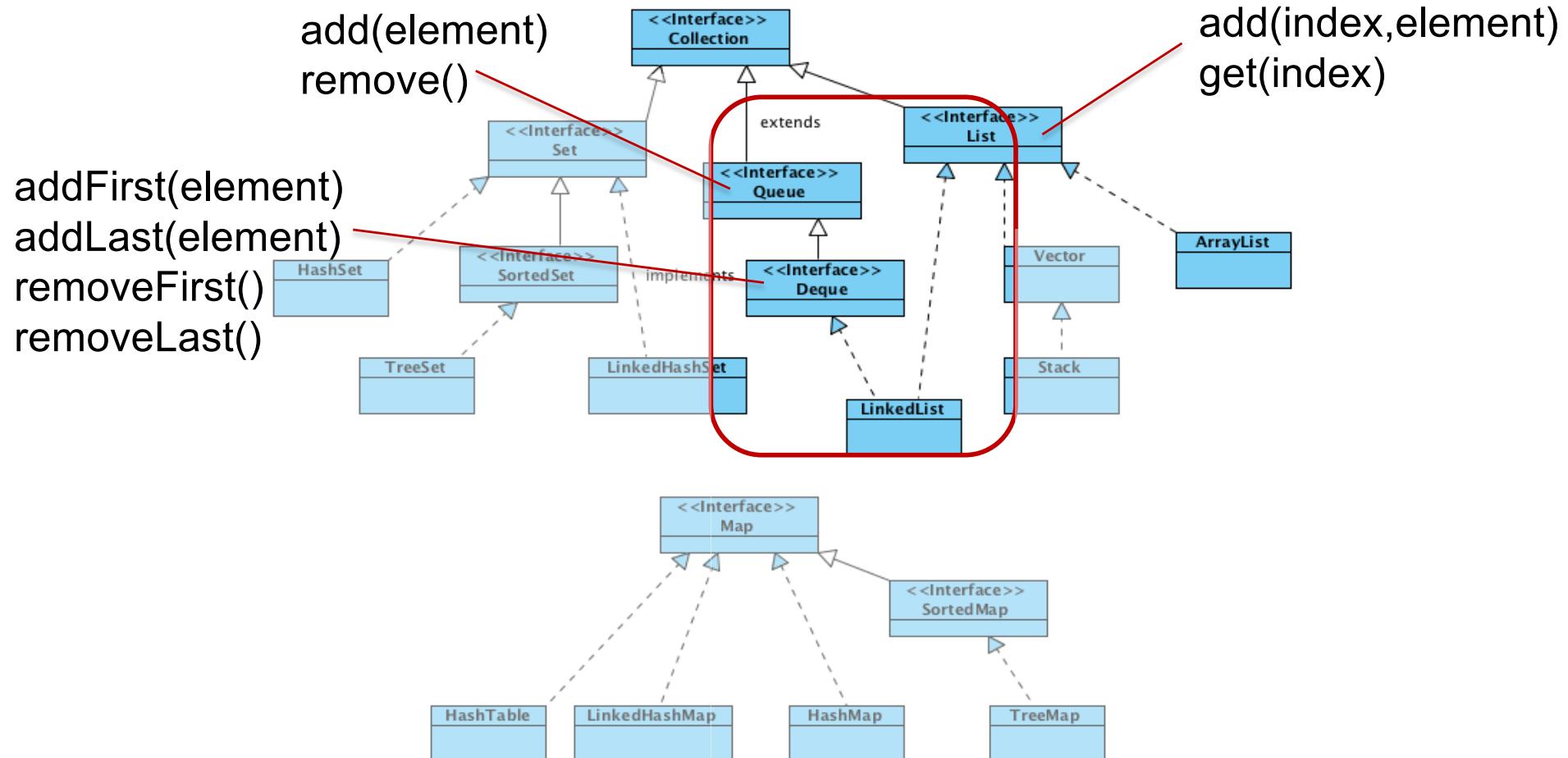
Separação de Implementação e Interface

- Permite alteração futura da classe que implementa a interface



Separação de Implementação e Interface

- Permite alteração futura da classe que implementa a interface
- Permite **herança múltipla de interface**



Interface ou classe abstrata?



- Sempre que possível, use **interfaces**;
- Lembre-se que Java não suporta herança múltipla:
 - Quando um objeto **estende** uma classe, não poderá **estender** nenhuma outra;
 - Não “**queime**” a herança sem **necessidade!**

Colisão de nomes em interfaces

- Assim como na herança múltipla, pode haver colisão de nomes em interfaces:
 - Não há colisão de implementação (simplifica);
 - Especificadores de acesso podem colidir;
 - Tipos de retorno podem colidir.

```
interface Forma {  
    void desenhar();  
    void inverter();  
}  
  
class UmaForma implements Forma {  
    protected void desenhar() { }      // Erro!  
    public int inverter() { }         // Erro!  
}
```

Hierarquias de interface

- Uma interface pode estender outra:

```
interface Forma {  
    void desenhar();  
    void aumentar(int t);  
}  
  
interface FormaInversivel extends Forma {  
    void inverter();  
}  
  
class UmaForma implements Forma { /* ... */ }  
  
class OutraForma implements FormaInversivel  
{ /* ... */ }
```

Exemplo de Interface: Comparable



- Java já implementa algoritmo de ordenação:
 - `Collections.sort()` para coleções;
 - `Arrays.sort()` para vetores.
 - (Coleções ordenadas: `TreeSet`, `TreeMap` ;)
- Para que a ordenação funcione, é preciso que os objetos implementem a interface `Comparable`
- As classes `Arrays` e `Collections` possuem outros métodos úteis: busca binária, cópia, máximo, mínimo, preenchimento, trocas, etc.

- Um exemplo de interface na API Java é a interface Comparable;
- Define o método `compareTo(Object obj)`:
 - Compara o objeto atual (`this`) com o objeto informado (`obj`);
 - Retorna `0` se `this = obj`;
 - Retorna um número negativo se `this < obj`;
 - Retorna um número positivo se `this > obj`.
- Métodos genéricos a utilizam para ordenar coleções de elementos.

A interface Comparable

```
class Valor implements Comparable {  
    int valor;  
    public Valor(int v) { valor = v; }  
  
    public int compareTo(Object obj) {  
        return valor - ((Valor)obj).valor;  
    }  
  
    public String toString() {  
        return "" + valor;  
    }  
}
```

A interface Comparable

```
public class Teste {  
    static void imprimir(Object[] vetor) {  
        for (int i = 0; i < vetor.length; i++)  
            System.out.print(vetor[i] + " ");  
        System.out.println();  
    }  
    public static void main(String[] args) {  
        Valor[] vetor = new Valor[] {  
            new Valor(10), new Valor(3),  
            new Valor(15), new Valor(7)  
        };  
        imprimir(vetor); // 10; 3; 15; 7;  
        Arrays.sort(vetor);  
        imprimir(vetor); // 3; 7; 10; 15;  
    }  
}
```

Comparadores (interface Comparator)

- Quando existe **mais de uma forma de ordenar objetos**, podemos criar comparadores;
- Implementam `java.util.Comparator`;
- Método `compare(Object a, Object b)` retorna:
 - Número negativo, se o primeiro $a < b$;
 - Zero, se $a == b$;
 - Número positivo se $a > b$.

Comparadores

```
public class Pessoa implements Comparable {  
    private String nome;  
    private int idade;  
  
    public Pessoa(String n, int i) {  
        nome = n;  
        idade = i;  
    }  
  
    public String toString() {  
        return nome + ", " + idade + " ano(s)";  
    }  
  
    public int compareTo(Object o) {  
        return nome.compareTo(((Pessoa)o).nome);  
    } // compara pessoa por nome  
  
    /* Continua... */
```

Comparadores

```
public class Pessoa implements Comparable<Pessoa> {
    private String nome;
    private int idade;

    public Pessoa(String n, int i) {
        nome = n;
        idade = i;
    }

    public String toString() {
        return nome + ", " + idade + " ano(s)";
    }

    public int getIdade() { return idade; }

    public int compareTo(Pessoa p) {
        return nome.compareTo(p.nome);
    } // compara pessoa por nome
}
```

Comparadores

```
public class ComparadorIdade
    implements Comparator {

    public int compare(Object o1, Object o2) {
        return (((Pessoa)o1).getIdade() -
                ((Pessoa)o2).getIdade());
    }
}
```

Comparadores

```
public class ComparadorIdade
    implements Comparator<Pessoa, Pessoa> {

    @Override
    public int compare(Pessoa p1, Pessoa p2) {
        return (p1.getIdade() - p2.getIdade());
    }

}
```

Comparadores

```
import java.util.*;  
  
public class Teste {  
    public static void main(String[] args) {  
        List<Pessoa> pessoas = new ArrayList<>();  
        pessoas.add(new Pessoa("Fulano", 20));  
        pessoas.add(new Pessoa("Beltrano", 18));  
        pessoas.add(new Pessoa("Cicrano", 23));  
  
        Collections.sort(pessoas);  
        for (Object o : pessoas) System.out.println(o);  
  
        Collections.sort(pessoas, new  
                            ComparadorIdade());  
        for (Object o : pessoas) System.out.println(o);  
    }  
}
```

Comparadores (mais um exemplo)

```
import java.util.Arrays;
import java.util.Comparator;

public class Student {
    private int grade;
    public Student(int grade) {
        this.grade = grade;
    }
    public int getGrade() {
        return grade;
    }
    public void setGrade(int grade) {
        this.grade = grade;
    }

    public static void main(String[] args)
    {
        Student s[]={ new Student(0), new Student(1), new Student(10), new Student(2)};
        Arrays.sort(s, new GradeComparator());
        for(Student i : s)
        {
            System.out.println(i.getGrade());
        }
    }
}

class GradeComparator implements Comparator<Student> {
    @Override
    public int compare(Student o1, Student o2) {
        return o2.getGrade() - o1.getGrade();
    }
}
```