Universidade Federal do Cariri

INTERFACES

Profª Paola Accioly
Material baseado nos slides de Thaís Alves Burity Rocha

O que vimos aula passada?

- Classes abstratas
- Métodos abstratos
- Quando usar classes e métodos abstratos?

Hoje falaremos sobre interfaces

- Como fazemos para que os módulos em Java consigam ser facilmente intercambiáveis?
- Aumentando coesão
- Diminuindo acoplamento
- E...
- Obedecendo a contratos

Contratos e herança

- Em alguns casos, como no sistema bancário, a herança se aplica
  - Comportamentos comuns podem ser reusados
  - Comportamentos distintos podem ser ajustados através da sobrescrita de métodos
  - Há polimorfismo
- Em outros casos, é desejável haver **contrato**, mas não faz sentido reusar comportamento
  - Exemplo: Hierarquia de formas geométricas

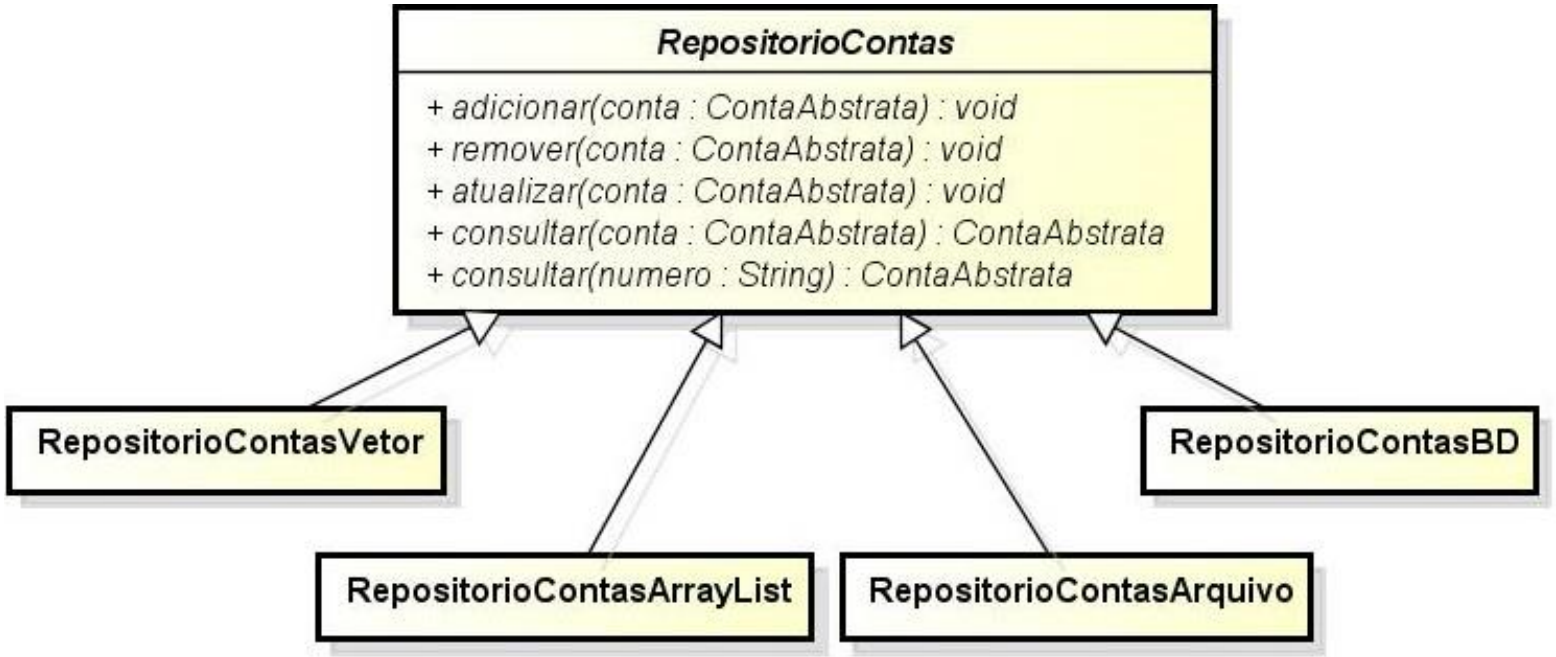
Repositório de dados

- Contrato: Operações de CRUD
  - Independente do meio de armazenamento, as operações são as mesmas
- Diferentes formas de armazenamento
  - Memória: Vetor ou ArrayList
  - Disco: Arquivos (texto, binário, CSV, JSON, XML, …)
  - Banco de dados
  - Não dá para reusar o código, pois cada meio tem uma implementação

Uma possível solução

Na prática, só está havendo herança de tipo

```
public abstract class RepositorioContas {
 public abstract void adicionar(ContaAbstrata conta);
 public abstract void remover(ContaAbstrata conta);
 public abstract void atualizar(ContaAbstrata conta);
 public abstract ContaAbstrata consultar(ContaAbstrata conta);
 public abstract ContaAbstrata consultar(String numero);
}
```



Interface

- Semelhante à uma classe
- Todos os métodos são implicitamente **abstract** e **public**
  - Exceto se forem marcados com **default** ou **static**
- Serve para definir contratos (polimorfismo)
- Mais um mecanismo de encapsulamento de Java
- É declarada com a palavra-chave **interface**
- Não pode ser instanciada
- Não possui construtor
- Deve ser implementada

Declaração

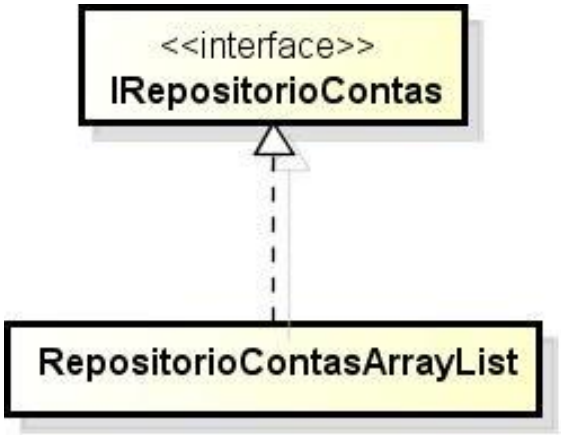
Convenção: Começar com I

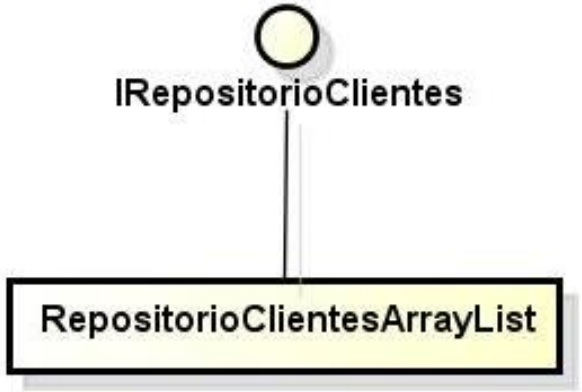
```
public interface IRepositorioContas {
    public abstract void adicionar(ContaAbstrata conta);
    void remover(ContaAbstrata conta);
    void atualizar(ContaAbstrata conta);
    ContaAbstrata consultar(ContaAbstrata conta);
    ContaAbstrata consultar(String numero);
}
```

- É redundante declarar métodos como **public** e **abstract**

Supertipo e subtipo

- **Supertipo:** Tipo definido pela interface
- **Subtipo:** Tipo definido pela classe que implementa à interface
  - Deve implementar **todos** os métodos definidos na interface (que não são **default** ou **static**)
  - Caso contrário, deve ser classe abstrata





Implementação de interface

```
public class RepositorioContasArrayList implements
IRepositorioContas {
   private ArrayList<ContaAbstrata> array;

   @Override
   public void adicionar(ContaAbstrata conta){
      int indice = array.indexOf(conta);
      if(indice == -1) array.add(conta);
      else{
         System.out.println("Não é possível adicionar
             uma conta igual a outra já existente!");
      }
   }

   /* demais métodos */
}
```

Regras de sobrescrita de métodos

- **Não pode** ter modificador de acesso mais restritivo

original
```
public void creditar(double valor) {...}
```

novo
```
private void creditar(double valor) {...}
```
✗

- **Não pode** mudar a lista de argumentos

original
```
public void creditar(double valor) {...}
```

novo
```
public void creditar(float valor) {...}
```
✗

- **Não pode** ter tipo de retorno incompatível

original
```
public double getSaldo() {...}
```

novo
```
public float getSaldo() {...}
```
✗

original
```
public Conta criarConta() {...}
```

novo
```
public Poupanca criarConta() {...}
```
✓

Referenciando interfaces

- Interfaces não podem ser instanciadas, mas podem ser referenciadas

```
IRepositorioContas rep = new RepositorioContasVetor();
ContaAbstrata conta = new Conta();
rep.adicionar(conta);
conta = new ContaBonificada();
rep.adicionar(conta);
```

E se quisermos usar outro repositório?

- Usar interface como argumento e tipo de retorno garante que poderá ser usado qualquer tipo que a implemente

Classe java.util.Arrays

- Provê métodos static para manipulação de vetores
- Ordenação de vetor: **void Arrays.sort(vetor)**
- Vetor de tipo primitivo: Ordenação natural
- Vetor de objetos: A classe do objeto deve implementar a interface **Comparable (java.lang.Comparable)**
- Método: **public int compareTo(Object obj)**
- Lógica: Dada a chamada **a.compareTo(b)**
  - Retornar inteiro <0, se **a** < **b**
  - Retornar inteiro >0, se **a** > **b**
  - Retornar 0, se **a** == **b**

Para lidar com ArrayList, usar a classe java.util.Collections

Ordenando contas pelo saldo

```
public abstract class ContaAbstrata implements Comparable{
    //construtores e métodos

    @Override
    public int compareTo(Object obj){
        ContaAbstrata conta = (ContaAbstrata) obj;
        if( this.saldo>conta.getSaldo() ) return 1;
        else if( this.saldo<conta.getSaldo() )return -1;
        else return 0;
    }
}
```

```
ContaAbstrata[] vetor = repConta.getContas(cliente);
Arrays.sort(vetor);
for(ContaAbstrata c: vetor){
    System.out.println(c);
}
```

No main

Modificadores de acesso

- Mesmos modificadores de classes
  - default
  - public

- Aplicam-se à interface apenas

Dados em interfaces

- Atributos são implicitamente constantes
  - **public, static** e **final**
- Classes que implementam a interface herdam todas as suas constantes
  - Podem usá-las como se fossem da própria classe
- Classes que não implementam a interface podem usar suas constantes
  - **<nome da interface>.<nome da constante>**

Evolução de interfaces

- Quando um novo método é incluído em uma interface...

```
public interface IRepositorioContas {
    void adicionar(ContaAbstrata conta);
    void remover(ContaAbstrata conta);
    void atualizar(ContaAbstrata conta);
    ContaAbstrata consultar(ContaAbstrata conta);
    ContaAbstrata consultar(String numero);
    ContaAbstrata[] consultar(Cliente cliente);
}
```

- Todas as classes que implementam a interface passam a ter erro de compilação

Métodos default

- Mecanismo que possibilita evoluir interfaces sem comprometer a compatibilidade
- Método concreto marcado como **default**
- O método é herdado por toda classe que implementa a interface
- Lógica: A classe deve implementar o método, mas caso não o faça, é provida uma implementação default

Exemplo de método default

```
public interface IRepositorioContas {

    void adicionar(ContaAbstrata conta);
    void remover(ContaAbstrata conta);
    void atualizar(ContaAbstrata conta);
    ContaAbstrata consultar(ContaAbstrata conta);
    ContaAbstrata consultar(String numero);

    default ContaAbstrata[] consultar(Cliente cliente){
        return null;
    }

}
```

Métodos static

- Interfaces podem conter métodos static
- Objetivo de facilitar a organização de métodos utilitários
  - Métodos utilitários específicos da interface são mantidos nela própria
  - Antigamente era necessário definir uma classe só para isso (geralmente abstrata)

Métodos private

- A partir de Java 9
- Interfaces podem conter métodos private, desde que não sejam default ou abstract
- Ou seja, métodos concretos (static ou não) que não podem ser sobrescritos
- Objetivo
  - Prover código utilitário que será reusado apenas por métodos default ou static da interface

Implementação de múltiplas interfaces

- Uma classe **pode** implementar mais de 1 interface
- Mecanismo de herança múltipla em Java

```
public interface Relogio{
  String getHoras();
}
```

```
public interface Radio{
  void ligar();
  void desligar();
  void trocarEstacao(int frequencia);
}
```

```
public class RadioRelogio implements Relogio, Radio {
  public String getHoras(){ ... }
  public void ligar(){ ... }
  public void desligar(){ ... }
  public void trocarEstacao(int frequencia){ ... }
}
```

Problema de ambiguidade

```
public interface A{
   int X = 10;
   default void m1(){
     System.out.println("Oi!");
   }
   void m2();
}
```

```
public interface B{
   int X = 20;
   default void m1(){
     System.out.println("Tchau!");
   }
   void m2();
}
```

```
public class MinhaClasse implements A, B {
    @Override
    public void m1(){ //é obrigado a sobrescrever
      A.super.m1(); //exemplo para usar versão herdada
    }

    @Override
    public void m2(){
      System.out.println(X); //erro compilação
      System.out.println(A.X);
      System.out.println(B.X);
    }
}
```

Herança e implementação de interfaces

- Uma classe pode herdar de outra classe e implementar uma ou mais interfaces, ao mesmo tempo

```
public class Superclasse {

}
```

```
public interface Interface {

}
```

```
public class Subclasse extends Superclasse implements Interface {

}
```

Quando criar uma classe, subclasse, classe abstrata ou interface?

- Não use herança quando a nova classe não passar no teste **É-UM** com nenhuma outra

- Use herança quando tiver que criar uma versão mais específica de uma classe e precisar sobrepor ou adicionar novos comportamentos

Quando criar uma classe, subclasse, classe abstrata ou interface?

- Use uma classe abstrata quando quiser definir um contrato para um grupo de subclasses e tiver algum código para ser reusado entre elas

- Use uma classe abstrata quando quiser garantir que ninguém possa criar objetos da classe

Quando criar uma classe, subclasse, classe abstrata ou interface?

- Use uma interface quando quiser definir uma função que outras classes possam desempenhar, independente de hierarquia de herança

- Lembre-se: **Contratos estão menos vulneráveis à mudanças, pois não há implementação**
  - E se for necessário adicionar métodos não previstos ao contrato, é possível usar métodos default