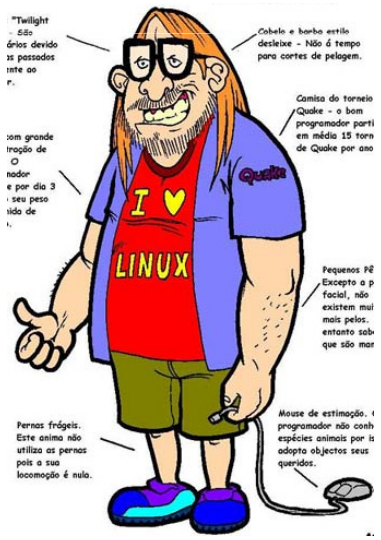
Introdução à Programação

Interface, Polimorfismo e Dynamic Binding

Interface

Programador Java PLENO

- Possuir sólida experiência em programação
- Desenvolvimento na linguagem JAVA
- Webservice, Struts ou JSF(desejável)
- Hibernate na linguagem JAVA
- Certificação: Java Programmer



Programadores bem diferentes fornecem o mesmo serviço: Interfaces!

Tópicos da Aula

- Hoje, aprenderemos a trabalhar com Interface
  - Motivação
  - Conceito
  - Utilização em Java
  - Interface e Herança Múltipla
  - Interface x Classe Abstrata
- Veremos, também, o que é Polimorfismo
  - Polimorfismo através de herança
  - Polimorfismo através de interface
  - Redefinição e Sobrecarga de Métodos
  - Dynamic Binding

Diferentes Classes de Repositório de Contas

```
public class RepositorioContasVector{
  private MiniJavaVector<Conta> contas;
  ...
  public void inserir(Conta c){
     if (c!= null) {
        contas.add(c);
     }else //Lança exceção;
     }
  }
```

Assinaturas idênticas

```
public class RepositorioContasArray{
  private Contas[] contas;
  private int proximaPosicaoVaga
  ...
  public void inserir(Conta c){
     if (c!= null) {
        if (this.arrayCheio()) {
           this.duplicarArray();
        }
        contas[proximaPosicaoVaga] = c;
        proximaPosicaoVaga++;
     }else //Lança exceção
  }
```

Implementações diferentes

Classe Cadastro de Contas

```
public class CadastroDeContasA {
  RepositorioContasArray contas;


  public CadastroDeContas(RepositorioContasArray repositorio) {
    contas = repositorio;
  }

  public void cadastrar(Conta c) throws ContaExistenteException{
    if (c!= null) {
        if (!this.existe(c.getNumero())) {
           contas.inserir(c);
        } else {
           throw new ContaExistenteException(c.getNumero());
        }
    }else {
        throw new IllegalArgumentException();
    }
  }
   ...
```

Utiliza repositório implementado com array

Outra Classe Cadastro de Contas

```
public class CadastroDeContasV {
  RepositorioContasVector contas;

  public CadastroDeContas(RepositorioContasVector repositorio) {
    contas = repositorio;
  }

  public void cadastrar(Conta c) throws ContaExistenteException{
    if (c!= null) {
        if (!this.existe(c.getNumero())) {
           contas.inserir(c);
        } else {
           throw new ContaExistenteException(c.getNumero());
        }
    }else {
        throw new IllegalArgumentException();
    }
  }
  ...
```

Utiliza repositório implementado com vector

Mesmo código em relação à implementação com array

Classes diferentes só por causa das diferentes implementações de repositório

Problemas

- Duplicação desnecessária de código
- O mesmo cadastro de contas deveria ser capaz de utilizar as operações fornecidas pelo repositório independente da implementação deste último
  - inserir
  - remover
  - procurar
  - consulta por critério

Solução: **Interfaces!**

Interfaces

- Caso especial de classes abstratas...
  - Todos os métodos são abstratos
    - Provêem uma interface para serviços e comportamentos
    - São qualificados como `public` por default
    - O uso do modificador **`abstract`** é opcional
  - Não definem atributos
    - Podem definir constantes
    - Por default todos os "atributos" definidos em uma interface são qualificados como `public`, `static` e `final`
  - Não definem construtores

Interfaces

- Não se pode criar objetos
- Definem tipo de forma abstrata, apenas indicando a assinatura dos métodos
- Estabelecem um conjunto de métodos que serão implementados pelos subtipos (subclasses)
  - No caso de uma classe abstrata implementar uma interface, ela não precisa implementar os métodos

Definindo Interfaces

```
public interface RepositorioContas{

    void inserir(Conta c);
    void remover(String numero);
    Conta procurar(String numero);
    boolean existir(String numero);
}
```

Palavra reservada para indicar que é uma interface

Assinaturas dos métodos

Subtipos sem Herança de Código

```
public class RepositorioContasVector implements RepositorioContas{
  private MiniJavaVector<Conta> contas;
  ...
  public void inserir(Conta c){
     if (c!= null) {
        contas.add(c);
     }else //Lança exceção;
     }
  }
```

Nome da interface

```
public class RepositorioContasArray implements RepositorioContas{
  private Contas[] contas;
  private int proximaPosicaoVaga
  ...
  public void inserir(Conta c){
     if (c!= null) {
        if (this.arrayCheio()) {
           this.duplicarArray();
        }
        contas[proximaPosicaoVaga] = c;
        proximaPosicaoVaga++;
     }else //Lança exceção
  }
```

Palavra reservada que indica que a classe implementa a interface

Classe Cadastro de Contas Genérica

```
public class CadastroDeContas {
  RepositorioContas contas;

  public CadastroDeContas(RepositorioContas repositorio) {
    contas = repositorio;
  }

  public void cadastrar(Conta c) throws ContaExistenteException{
    if (c!= null) {
        if (!this.existe(c.getNumero())) {
           contas.inserir(c);
        } else {
           throw new ContaExistenteException(c.getNumero());
        }
    }else {
        throw new IllegalArgumentException();
    }
  }
  ...
```

Utiliza qualquer repositório que implemente a interface **RepositorioContas**

Utilizando Cadastro e Repositório

```
...
CadastroDeContas  cadastro;
RepositorioContasVector repositorio1;
repositorio1 = new RepositorioContasVector();
cadastro = new CadastroDeContas(repositorio1);
...
RepositorioContasArray repositorio2;
repositorio2 = new RepositorioContasArray();
cadastro = new CadastroDeContas(repositorio2);
...
```

implements

- *classe* implements

  interface1, interface2, ...

- *subtipo* implements

  supertipo1, supertipo2, ...

- Múltiplos supertipos:
  - uma classe pode implementar mais de uma interface (contraste com classes abstratas...)

implements

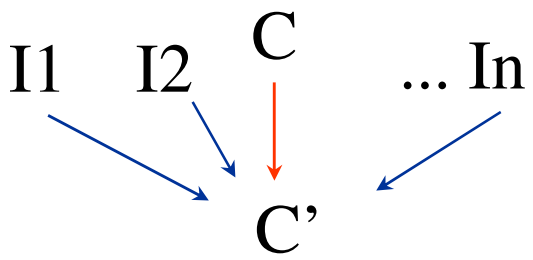
- Classe que implementa uma interface deve *definir* os métodos da interface:
  - classes concretas têm que implementar os métodos
  - classes abstratas podem simplesmente conter um ou mais métodos abstratos correspondentes aos métodos da interface

Interfaces e Reusabilidade

- Evita duplicação de código através da definição de um tipo genérico, tendo como subtipos várias classes não relacionadas
- Tipo genérico pode agrupar objetos de várias classes definidas de forma independente, sem compartilhar código via herança, tendo implementações totalmente diferentes
- Classes podem até ter mesma semântica...

Definição de Classes: Forma Geral

```
class C'
   extends C
   implements I1, I2, ..., In {
      /* ... */
   }
```

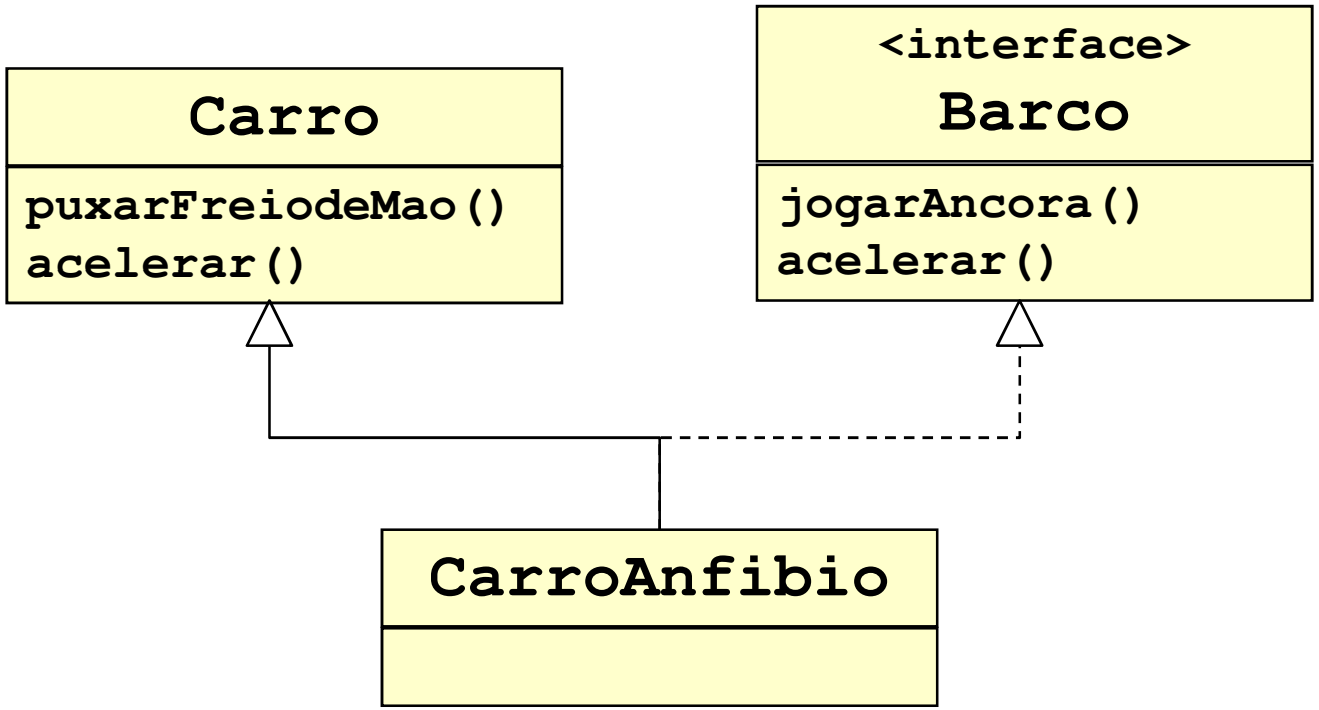


Subtipos com Herança Múltipla de Assinatura

```
interface I
  extends I1, I2, ..., In {
  /*... assinaturas de novos
        métodos ...
  */
}
```

Simulando Herança Múltipla com Interface

- Para simular herança múltipla, podemos combinar herança simples com implementação de interface



Subtipos com Herança Múltipla de Assinatura

- Classes (abstratas)
  - Agrupa objetos com implementações compartilhadas
  - Define novas classes através de herança (simples) de código
  - Só uma pode ser supertipo de outra classe
- Interfaces
  - Agrupa objetos com implementações diferentes
  - Define novas interfaces através de herança (múltipla) de assinaturas
  - Várias podem ser supertipo do mesmo tipo

Iterators

- Um ***iterator*** é um objeto que permite percorrer uma coleção de itens
  - Um de cada vez a cada passo
- Um objeto *iterator* deve possuir um método `hasNext()` que informa se existe ainda algum item a ser percorrido na coleção
- O método `next()` retorna o item atual e pula para o próximo item da coleção

Interface *Iterator*

- Como uma coleção pode ser implementada de várias formas diferentes (array, vector, etc), a implementação de um ***iterator*** depende da implementação da coleção
- Porém a assinatura de um *iterator* deve ser a mesma, independentemente da implementação
- Assim deve ser criada uma interface *iterator*, e esta interface deve ser implementada por cada subtipo de objeto *iterator*

Definindo e Implementando Interface IteratorConta

```
public interface IteratorConta {
        boolean hasNext();
        Conta next();

}
```

Nome da interface

```
public class IteratorContaVector implements IteratorConta {

    MiniJavaVector<Conta> contas;
    private int proximaPosicao;
    public IteratorContaVector(MiniJavaVector<Conta> vector){
        contas = vector;
        proximaPosicao = 0;
    }
    public boolean hasNext() {
        return proximaPosicao < contas.size();
    }
    public Conta next() {
        Conta conta = contas.get(proximaPosicao);
        proximaPosicao++;
        return conta;
    }
}
```

Uma implementação da interface

Classe RepositorioContasComVector

```
public interface RepositorioContas{

   void inserir(Conta c);
   void remover(String numero);
   Conta procurar(String numero);
   boolean existir(String numero);
   IteratorConta getContasVip();
}
```

Modifica a interface RepositorioContas

```
public class RepositorioContasVector implements RepositorioContas{
  private MiniJavaVector<Conta> contas;
  ...
  public IteratorConta getContasVip() {
     MiniJavaVector<Conta> resultado = new MiniJavaVector<Conta>();
     for(int i = 0; i < contas.size(); i++) {
        if (contas.get(i).getSaldo() > 10000){
           resultado.add(contas.get(i));
        }
     }
     return new IteratorContaVector(resultado);
  }
}
```

Classe Cadastro de Contas Genérica

```
public class CadastroDeContas {
  RepositorioContas contas;

  public void creditarBonusContaVip(){
    IteratorConta it = contas.getContasVip();
    double bonus = 1000;
    while (it.hasNext()) {
        Conta conta = it.next();
        conta.creditar(bonus);
    }

  }
  ...
```

Retorna item atual e passa para o próximo item

Laço utilizado para percorrer a coleção

Uso do `Iterator` permite esconder a forma em que o repositório é implementado

Polimorfismo

- A palavra polimorfismo significa “assumir muitas formas”
- Uma referência polimórfica é uma variável que pode se referir a objetos de tipos diferentes em intervalos de tempo diferentes
  - Em Java, qualquer referência é potencialmente polimórfica

Polimorfismo

- Um método chamado através de uma referência polimórfica pode ter comportamentos diferentes entre uma chamada e outra

- Polimorfismo em Java pode se dar através de **herança** ou **interfaces**

Polimorfismo via Herança

- Polimorfismo
  - Uma conta pode ser
    - Uma poupança
    - Uma conta especial
  - Herança permite a substituição do supertipo pelo subtipo no código
    - Onde é permitido o supertipo, o subtipo pode ser utilizado
    - Variável de supertipo pode em um determinado instante do tempo guardar uma referência de um subtipo

Polimorfismo via Interface

- Polimorfismo
  - Um repositório de contas pode ser
    - Um repositório de contas implementado com array
    - Um repositório de contas implementado com vector
  - Uma referência para o tipo de uma interface pode referenciar um objeto de qualquer classe que implementa a interface
    - Onde é permitida a interface, o objeto de uma classe que implementa a interface pode ser utilizado

Redefinição de Métodos

- Uma subclasse pode redefinir (***override***) um método herdado da superclasse
- O novo método deve ter a mesma assinatura do método da superclasse, mas pode ter um corpo diferente
- Semântica dos métodos redefinidos deve ser preservada

Redefinição de Métodos

- Visibilidade dos métodos redefinidos deve ser preservada
  - Em Java, é possível aumentar a visibilidade de um método
    - Diminuir a visibilidade gera um erro de compilação
- Só é possível acessar a definição dos métodos da superclasse imediata (via *super*)
- Se modificador **`final`** vier na frente de método, este não pode ser redefinido

Sobrecarga de Métodos

- Sobrecarga (***overloading***) de método é o processo de dar a um método múltiplas definições
- Isto quer dizer que somente o nome do método não é suficiente para determinar qual o método que se deseja utilizar
- A assinatura de cada método *overloaded* deve ser única
  - O que vai diferenciar os métodos é o número, tipo e ordem dos parâmetros
    - Não podem somente ser diferenciados pelo tipo de retorno

Sobrecarga de Métodos

- Construtores são comumente *overloaded*
  - Útil para permitir várias formas de inicialização de um objeto
- O compilador checa qual é o método que está sendo chamado, analisando os parâmetros

Chamada

```
conta = new Conta("2", 50);
```

```
public class Conta{
    ...
    public Conta(String num, double saldo) {
        this.numero = num;
        if (saldo >= 0){
            this.saldo = saldo;
        } else {
            throw new IllegalArgumentException();
        }
}
    public Conta(String num) {
        this(num,0);
    }
    ...
```

Sobrecarga x Redefinição

- **Sobrecarga** trata múltiplos métodos com o mesmo nome, mas assinaturas diferentes
- **Redefinição** trata de dois métodos, um na superclasse e outro na subclasse, que possuem a mesma assinatura
- **Sobrecarga** permite que se defina uma operação similar em diferentes formas para diferentes parâmetros
- **Redefinição** permite que se defina uma operação similar em diferentes formas para diferentes tipos de objeto

Dynamic Binding: `debitar` de Conta e ContaEspecial

```
public class Conta{
  public void debitar(double valor)
                throws SaldoInsuficienteException{
    if (valor <= getSaldo()){
       setSaldo(getSaldo() - valor);
    } else {
       throw new
       SaldoInsuficienteException(getNumero());
    }
 }
```

```
public class ContaEspecial extends Conta{
  double limite = 1000;
  public void debitar(double valor)
                throws SaldoInsuficienteException{
    if (valor <= (getSaldo()+ limite){
       setSaldo(getSaldo() - valor);
    } else {
       throw new
       SaldoInsuficienteException(getNumero());
    }
 }
```

Dynamic Binding

```
...
MiniJavaSystem sys = new MiniJavaSystem();
Conta  ca;
ca = new ContaEspecial(¨21.342-7¨);
ca.creditar(500);
ca.debitar(600);
sys.println(ca.getSaldo());
ca = new Conta(¨21.111-7¨);
ca.creditar(500);
ca.debitar(600);
sys.println(ca.getSaldo());
...
```

Qual versão de debitar é utilizada?

Em tempo de execução é decidida qual versão de debitar deve ser utilizada

Dynamic Binding

- Uma chamada de um método dentro do código deve ser *ligada* a definição do método chamado
- Se esta ligação (binding) é feita em tempo de compilação, então sempre a chamada do método será ligada ao mesmo método
  - *Static Binding*
- Java só liga a chamada de métodos às definições de métodos correspondentes em tempo de execução
  - *Dynamic Binding* ou *Late Binding*

Dynamic Binding

- Portanto, se existir dois métodos com o mesmo nome e tipo(definição e redefinição), em Java, o código é escolhido dinamicamente
- Escolha é feita com base na classe do objeto associado à variável destino da chamada do método

Resumindo ...

- Interface
  - Motivação
  - Conceito
  - Utilização em Java
  - Padrão Iterator
  - Interface e Herança Múltipla
  - Interface x Classe Abstrata
- Polimorfismo
  - Polimorfismo através de herança
  - Polimorfismo através de interface
  - Redefinição x Sobrecarga de Métodos
  - Dynamic Binding