

Universidade Federal do Cariri

CLASSES ABSTRATAS E HERANÇA

Profª Paola Accioly

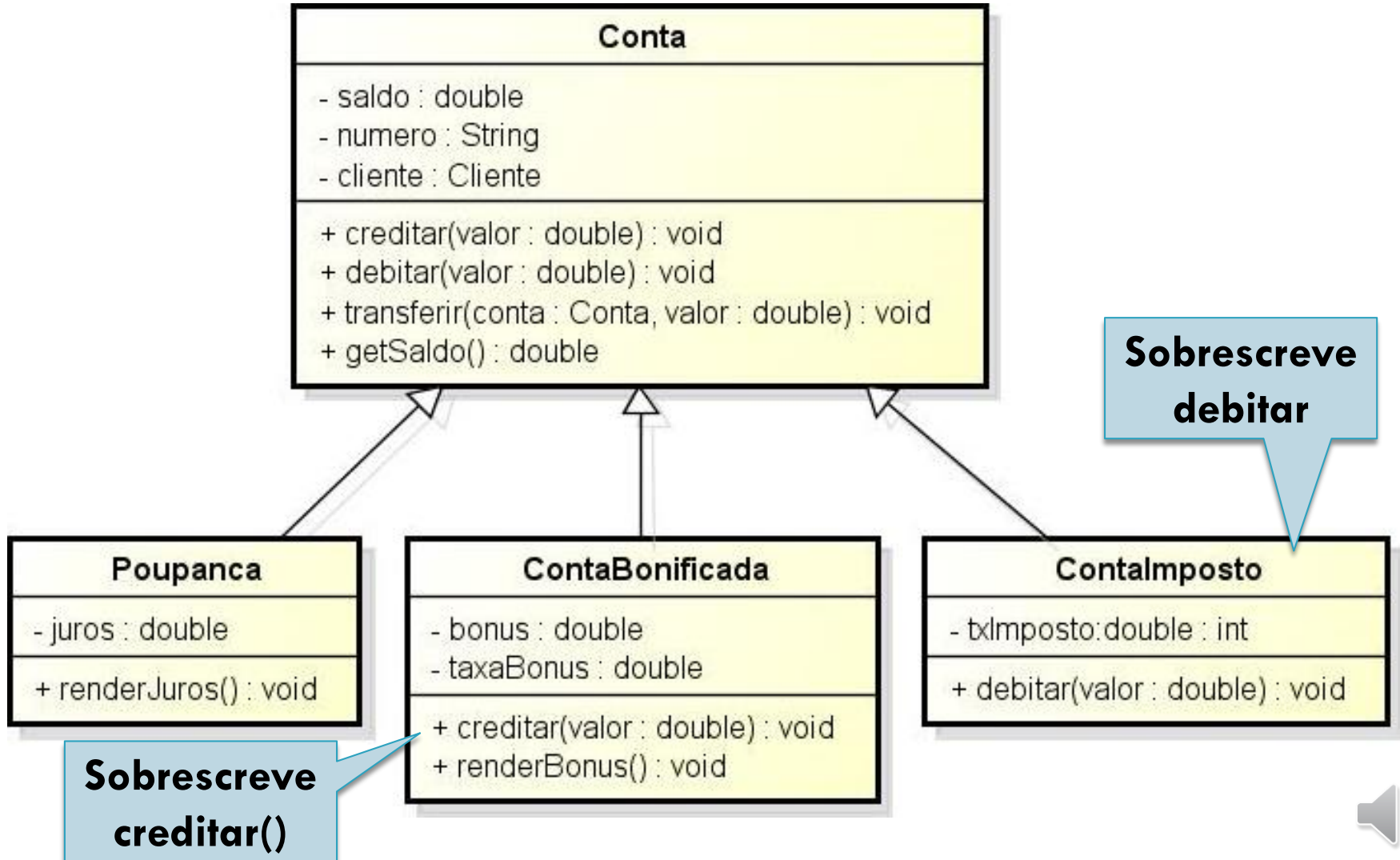
Material baseado nos slides de Thaís Alves Burity Rocha



Aula passada falamos de herança

- ❑ Superclasse e subclasses
- ❑ Operador instanceof
- ❑ Modificador protected
- ❑ Herança múltipla
- ❑ Sobrescrita de métodos
- ❑ Polimorfismo
- ❑ Modificador final
- ❑ Classe Object

Hoje teremos um novo tipo de conta



Classe ContaImposto

```
public class ContaImposto extends Conta {  
    private double txImposto;  
  
    public ContaImposto(Cliente c, String n, double s) {  
        super(c, n, s);  
        txImposto = 0.02;  
    }  
  
    @Override  
    public void debitar(double valor) {  
        double valorComImposto = valor + (valor*txImposto);  
        saldo -= valorComImposto;  
    }  
}
```



Análise do mecanismo de herança

- ❑ Em uma hierarquia, existem características **comuns** e **diferentes** entre as classes
- ❑ Planejar o que deve ser comum e como tratar as diferenças nem sempre é trivial
- ❑ A complexidade tende a aumentar com o tamanho da hierarquia



Sobrescrita de métodos

- ❑ Permite personalizar comportamentos
- ❑ Só é perceptível no código da subclasse ou na **execução** do programa
- ❑ **Seria melhor se fosse possível identificar na superclasse os comportamento comuns e diferentes entre as classes**



Contrato

- Ou seja, em algumas situações queremos definir apenas um **contrato**
 - ▣ O que deve ser feito e não como é feito
- A existência de um **contrato** viabiliza o uso de **polimorfismo**
- Java oferece dois recursos para a definição de **contratos**
 - ▣ **Classes abstratas**
 - ▣ Interfaces



Classe abstrata

- É declarada com **abstract**

```
public abstract class ClasseAbstrata { }
```

- Pode conter **métodos concretos e abstratos**

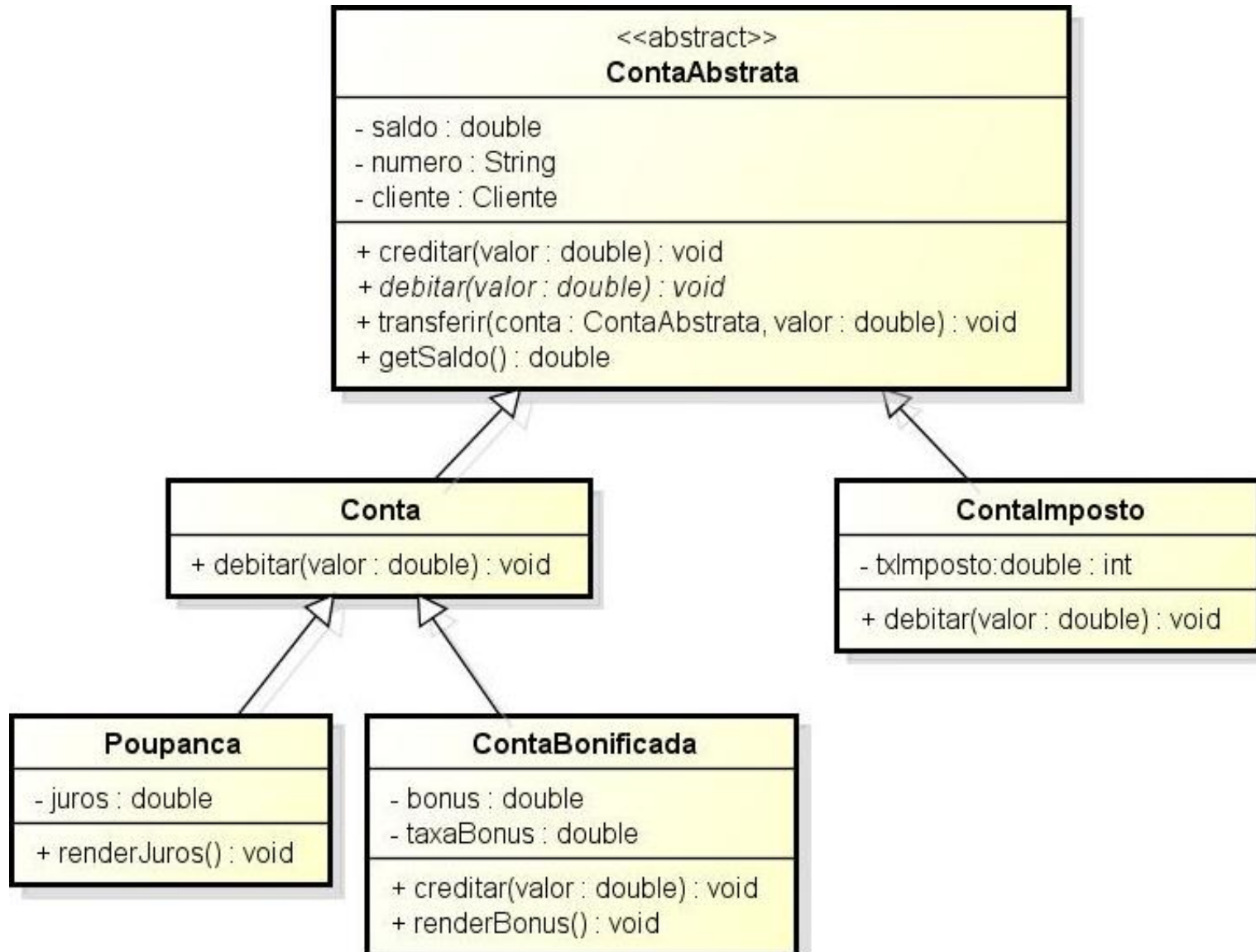
- Métodos abstratos

- ▣ Não possuem corpo, só assinatura
- ▣ Foram feitos para ser sobrescritos
- ▣ São declarados com **abstract**
- ▣ Não podem ser definidos em classes concretas

```
public abstract void metodoAbstrato() ;
```



Redefinição do sistema bancário



Classe ContaAbstrata

```
public abstract class ContaAbstrata {  
    protected Cliente cliente;  
    protected String numero;  
    protected double saldo;  
  
    public ContaAbstrata(Cliente c, String n, double s){  
        this.cliente = c;  
        this.numero = n;  
        this.saldo = s;  
    }  
  
    //getters e setters  
  
    public abstract void debitar(double valor);  
  
    public void creditar(double valor){  
        saldo += valor;  
    }  
}
```



Classe ContaAbstrata (cont.)

```
public void transferir(ContaAbstrata conta, double valor){  
    this.debitar(valor);  
    conta.creditar(valor);  
}  
@Override  
public boolean equals(Object obj){  
    if(obj instanceof ContaAbstrata){  
        ContaAbstrata conta2 = (ContaAbstrata)obj;  
        if(this.numero.equals(conta2.getNumero())) return true;  
    }  
    return false;  
}  
@Override  
public String toString(){  
    return "numero da conta: "+numero+"; titular: "+  
        cliente.getNome();  
}  
}
```

**Chamada de
método abstrato**



Classe Conta

```
public class Conta extends ContaAbstrata {  
    public Conta(Cliente c, String n, double s){  
        super(c, n, s);  
    }  
  
    @Override  
    public void debitar(double valor) {  
        if(saldo >= valor) saldo -= valor;  
        else System.out.println("Saldo insuficiente");  
    }  
}
```

**Sem cobrança
de juros**



Classe Poupança

```
public class Poupanca extends Conta {  
    private double juros = 0.2;  
  
    public Poupanca(Cliente c, String n, double s){  
        super(c,n,s);  
    }  
  
    public void renderJuros(){  
        this.creditar(saldo*juros);  
    }  
}
```



Classe ContaBonificada

```
public class ContaBonificada extends Conta {  
    private double bonus;  
    private double taxaBonus = 0.1;  
  
    public ContaBonificada(Cliente c,String n,double s){  
        super(c,n,s);  
    }  
    @Override  
    public void creditar(double valor) {  
        bonus += (valor*taxaBonus);  
        super.creditar(valor);  
    }  
    public void renderBonus() {  
        super.creditar(bonus); bonus = 0;  
    }  
}
```



Classe ContaImposto

```
public class ContaImposto extends ContaAbstrata {  
    private double taxaImposto = 0.02;  
  
    public ContaImposto(Cliente c, String n, double s){  
        super(c,n,s);  
    }  
    @Override  
    public void debitar(double valor){  
        double valorFinal = valor + (valor*taxaImposto);  
        if(valorFinal >= saldo) saldo -= valorFinal;  
        else System.out.println("Saldo insuficiente");  
    }  
}
```

**Com cobrança
de juros**




E o repositório de contas?

- ❑ A classe deve ser modificada para suportar a classe mais geral (o tipo de conta mais genérico que existe)
- ❑ Ou seja, onde havia **Conta**, substituir por **ContaAbstrata**



Cuidado!

- ❑ Classe concreta não pode ter método abstrato

```
public class ClasseQualquer {  
    public abstract void metodoAbstrato();   
}
```

- ❑ Classe abstrata não pode ser instanciada, apenas referenciada

```
public abstract class ClasseAbstrata {}  
public class ClasseConcreta extends ClasseAbstrata{}
```

```
ClasseAbstrata ref1 = new ClasseConcreta(); 
```

```
ClasseAbstrata ref2 = new ClasseAbstrata(); 
```



Construtor

- ❑ Classes abstratas **não podem** ser instanciadas
- ❑ Mas **possuem** construtor
- ❑ O construtor da classe abstrata só pode ser chamado por construtor(es) de sua(s) subclasse(s)



Significado e uso de classe abstrata

- ❑ **No geral, foi feita para ser herdada**
 - ❑ Logo, não pode ser marcada como **final**
- ❑ A subclasse **deve** sobrescrever os métodos abstratos
 - ❑ Cada subclasse define sua própria implementação
 - ❑ **A assinatura comum “garante” um comportamento**
- ❑ Caso contrário, a subclasse também deve ser abstrata



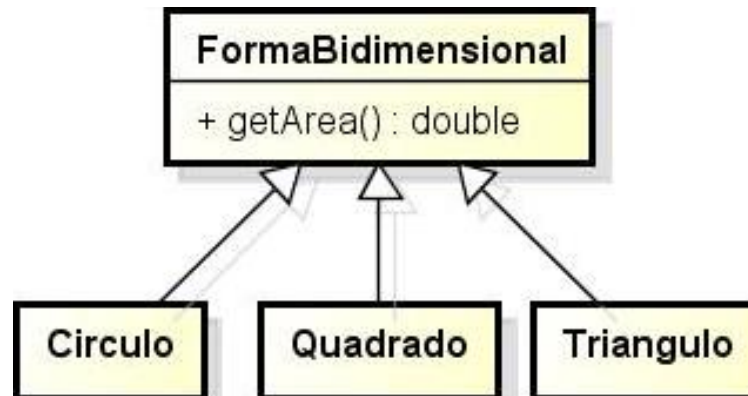
Boas práticas (1)

- ❑ **Considere usar classes abstratas quando não for possível generalizar comportamentos para toda hierarquia**
 - ▣ Define-se um contrato, mantendo-se a vantagem do reuso
 - ▣ O contrato viabiliza o uso de polimorfismo



Boas práticas (2)

- ❑ **Considere usar classes abstratas quando não fizer sentido instanciar a superclasse**

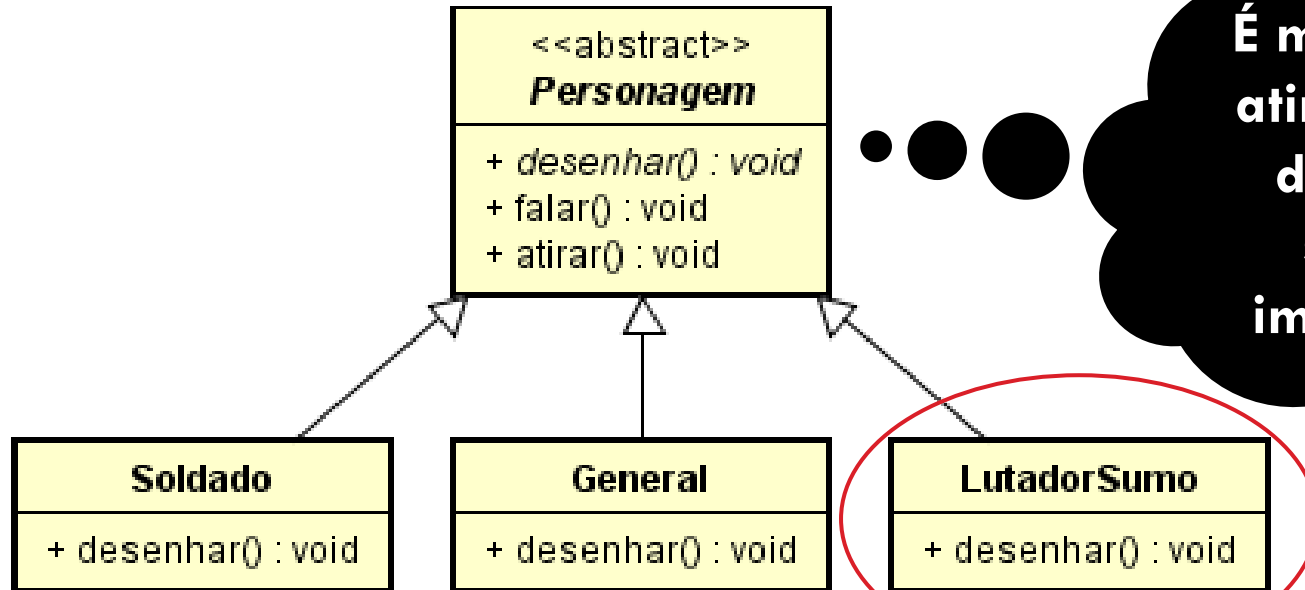


Benefícios de herança

- ❑ Reuso de código
 - ▣ Menos código precisa ser adicionado na subclasse
- ❑ Facilidade de manutenção
 - ▣ Melhoria de legibilidade
 - ▣ Menor volume de código
 - ▣ Polimorfismo: Se o código usa uma referência de `ContaAbstrata`, ele pode manipular qualquer tipo de conta (atual ou futura)



Problema: Nem todo personagem atira



É melhor tirar o `atirar()` daqui e deixar cada subclasse implementar?

Deve sobrescrever o método para não fazer nada?



Acoplamento e coesão

- ❑ Um bom projeto OO deve se preocupar em definir **classes coesas e pouco acopladas**
- ❑ Classes coesas têm um propósito bem definido
 - ▣ Não assumem mais responsabilidade do que devem
- ❑ Classes pouco acopladas são pouco dependentes entre si
 - ▣ As classes se relacionam, mas conseguem evoluir sem afetar muito outras classes



Problemas de herança

- ❑ Herança “fere” a ocultação de informação e encapsulamento
 - ▣ Mudanças na superclasse podem ser difíceis
 - ▣ Subclasses recebem/enxergam os dados e a implementação da superclasse
 - ▣ Ou seja, subclasses são fortemente acopladas à superclasse
- ❑ Herança não permite mudança dinâmica de comportamento (em tempo de execução)



Problema: Como variar a arma?

```
public class Soldado extends Personagem {
    public void desenhar(){
        ...
        //desenha o soldado
    }
    public void atirar(){
        System.out.println("tiro"); //a arma é um revólver
    }
}
```

```
public class General extends Personagem {
    public void desenhar(){
        ...
        //desenha o general
    }
    public void atirar(){
        //a arma é uma metralhadora
        System.out.println("rajada");
    }
}
```



Solução problemática

```
public class Soldado extends Personagem {  
    public void desenhar(){  
        ...  
        //desenha o soldado  
    }  
  
    //variação estática  
    public void atirar(int arma){  
        if(arma==0) System.out.println("tiro");  
        else System.out.println("rajada");  
    }  
}
```

**E se surgirem
novos tipos
de arma?**



○ que fazer para aumentar o reuso?

□ **Encapsular as partes que podem mudar**

- ▣ ○ que varia deve ser tratado em uma classe
- ▣ Ou seja, devemos ter uma classe Arma

□ **Programar para contratos**

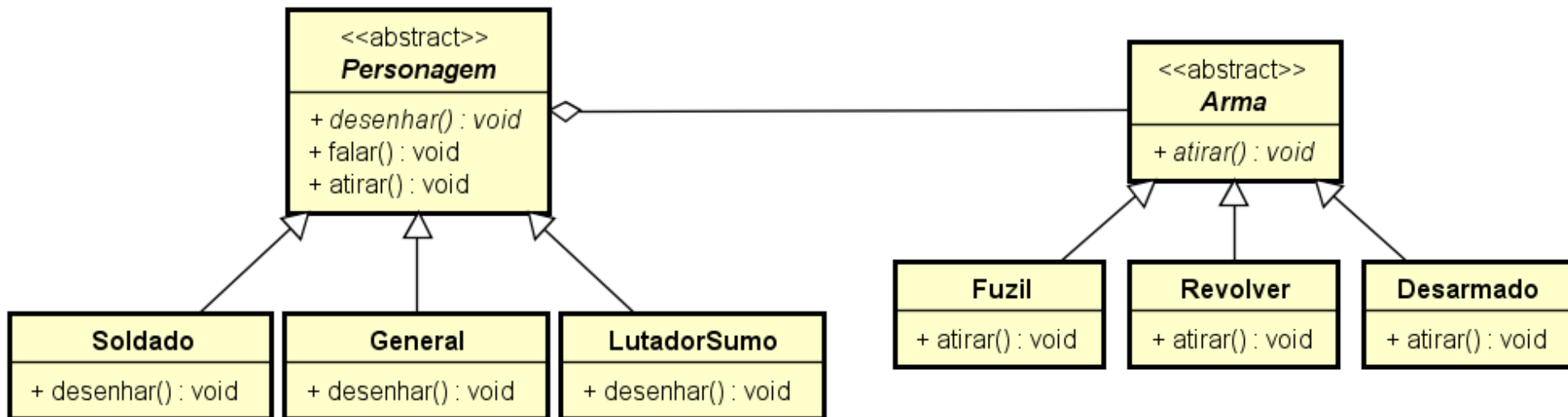
- ▣ A chave é explorar o polimorfismo para poder trocar objetos em tempo de execução
- ▣ Ou seja, podemos ter uma variedade de armas, desde que elas satisfaçam o mesmo contrato, podemos substituí-las



Solução

//evita chamada com essa cara:
personagem.getArma().atirar();

```
public abstract class Personagem {  
    private Arma arma;  
    public Personagem (Arma arma){  
        this.arma = arma;  
    }  
    public void atirar(){  
        arma.atirar(); //delegação  
    }  
    public void setArma(Arma arma){  
        this.arma = arma;  
    }  
    ...  
}
```



LutadorSumo usa arma do tipo desarmado, que tem implementação “morta” para atirar