

- *Explorar os conceitos fundamentais acerca do uso **herança** na linguagem Java*
 - *Como a herança reutiliza código, vantagens e desvantagens, o problema “weak base-class”, acoplamento com herança, o uso de herança e extensão em alguns cenários e boas práticas*
- *Discutir o uso de herança com outros mecanismos de reuso de código, como a **composição***
 - *Por que usar composição? Quais são os benefícios? regras de design, delegação (ou forward), acoplamento de código com composição, boas práticas e exemplos de códigos em Java*

- Quando você precisa de uma classe em Java, você pode escolher entre:
 - ① Usar uma classe que já faz exatamente o que você deseja fazer (API, Internet, colega, ...)
 - ② Escrever uma classe “do zero”
 - ③ Reutilizar uma classe existente ou estrutura (hierarquia) de classes com **herança**
 - ④ Reutilizar uma classe existente com **composição**

Parte I : reuso com herança

- *Permite reutilizar as características de uma classe na definição de outra classe*
 - *Reutilização direta de código previamente definido por alguém em uma **superclasse***
- *Terminologias relacionadas à Herança:*
 - *Classes mais generalizadas: **superclasses***
 - *Mais especializadas: **subclasses***
- *Na herança, classes estão ligadas à uma **hierarquia***
- *É a contribuição original do paradigma OO*
- *Linguagens como Java, C++ , Object Pascal, ...*

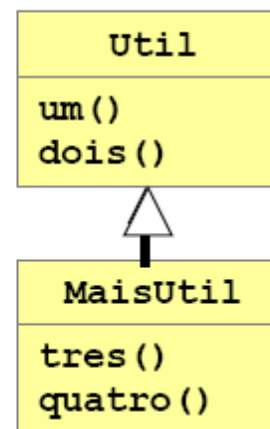
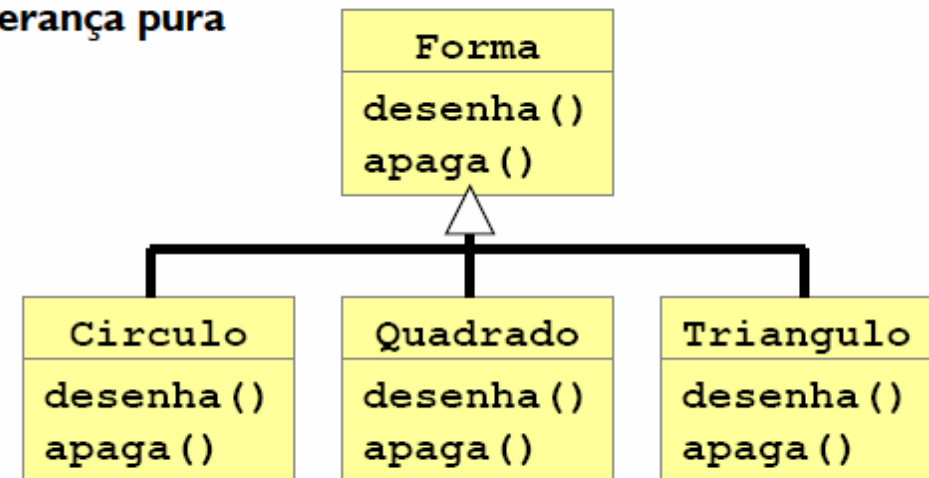
- *Propriedades, conexões a objetos e métodos comuns ficam na superclasse (classe de generalização)*
 - *Adicionamos mais dessas coisas nas subclasses (classes de especialização)*
- *A herança viabiliza a construção de sistemas a partir de componentes facilmente reutilizáveis*
 - *Suporte a reutilização até entre sistemas*
 - *Facilita a extensibilidade em um mesmo sistema*
- *A classe descendente não tem trabalho para receber a herança*
 - *Basta usar o **extends** e redefinir os métodos necessários (comportamentos mais especializados)*

Herança pura e extensão

- Pode-se reutilizar código com herança pura (note, no caso ao lado, que todos os métodos genéricos foram sobrepostos)

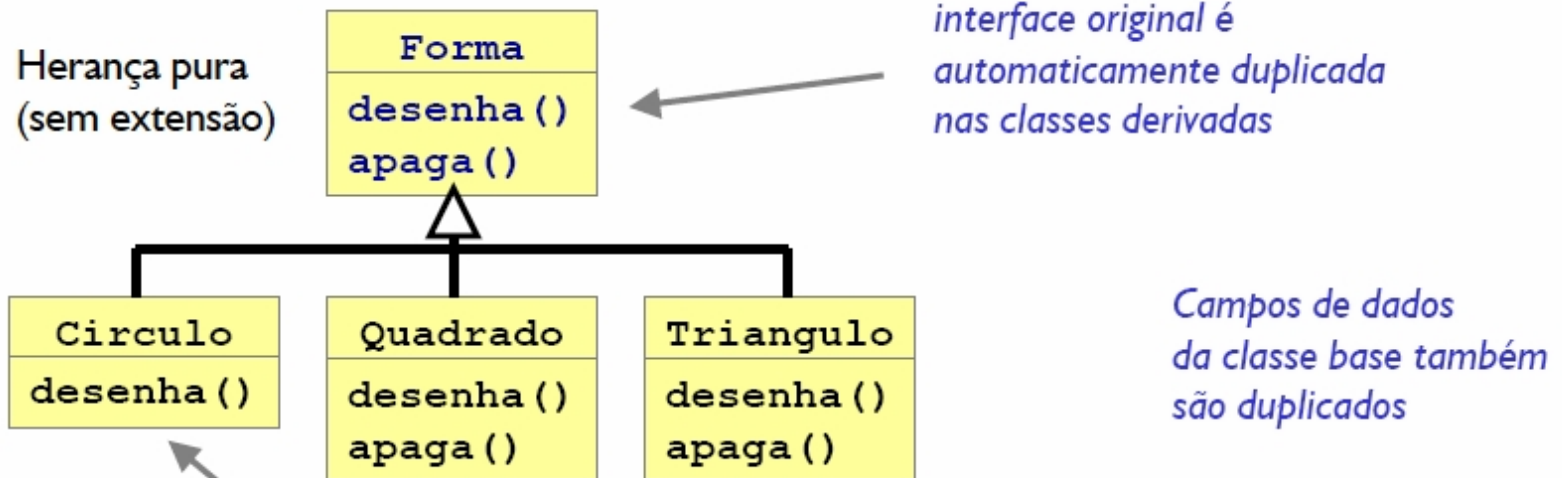
- Na extensão, novos comportamentos foram adicionados nas classes de especialização (os métodos genéricos são herdados)

Herança pura



Extensão

Árvore de Herança



Se membro derivado não for redefinido, implementação original é usada

```
class Forma {
    public void desenha() {
        /*...*/
    }
    public void apaga() {
        /*...*/
    }
}
```

```
class Circulo extends Forma {
    public void desenha() {
        /* nova implementação */
    }
}
```

Assinatura do método tem que ser igual ou sobreposição não ocorrerá (poderá ocorrer sobrecarga não desejada)

Árvore de Herança

- *União das classes que herdaram entre si gera uma árvore de herança (ou uma hierarquia de classes relacionadas)*
 - *Todos os objetos herdam características (gerais) definidas em Forma*
 - *Círculo, Quadrado e Triângulo são especializações de Forma (é uma Forma)*
- *Em todos os casos, cada subclasse possui uma única superclasse*
 - *A isso, chamamos de herança simples*
- *Em algumas linguagens, é possível herdar a partir de diversas superclasses*

- *Herança é uma técnica para prover suporte a especialização*
 - *Uma classe mais abaixo na hierarquia deve especializar comportamentos (tipo mais especializado de...)*
- *Métodos e variáveis internas são herdados por todos os objetos dos níveis mais abaixo*
- *Várias subclasses podem herdar as características de uma única superclasse*
- **Lembre-se:** *Java não possui herança múltipla!*
 - *Mas C++ sim!*

- Se *B* é uma subclasse de *A*, então:
 - Os objetos de *B* suportam todas as operações suportadas pelos objetos de *A*, exceto aquelas que foram redefinidas
 - Os objetos de *B* incluem todas as variáveis de instância de *B* + todas as variáveis de instância de *A*
- Lembre-se que métodos declarados como **private** não serão herdados, visto que o escopo é a classe que declarou este método em particular
- Construtores também não são herdados
 - Serão chamados (em cascata) na construção de objetos especializados (inicializar a parte que vem “de cima”)

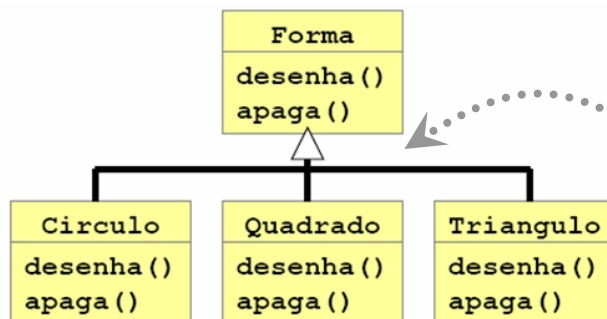
Benefícios da Herança

- *Como código pode ser facilmente reutilizado, a quantidade de código a ser adicionado numa (sub)classe pode diminuir bastante*
 - *Subclasses provêem comportamentos **especializados** tomando **como base os elementos comuns***
 - *A reutilização pode ser realizada mais de uma vez por outras (e até novas) subclasses*
- *Potencializa a manutenção de sistemas*
 - *Maior legibilidade do código existente*
 - *Quantidade menor de código a ser acrescentado*

Benefícios da Herança

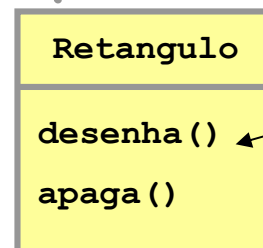
- Quando relacionamos duas classes via herança, podemos ter polimorfismo com ligação dinâmica
 - Se um fragmento de código usa uma referência de uma superclasse (Forma), esta pode manipular novos tipos concretos futuros (Retângulo, e.g.)
 - Isso porque ligação dinâmica irá garantir que os métodos certos de Retângulo (desenhar(), apagar(), ...) sejam chamados adequadamente

Código Java



Hierarquia de Classes

Tipo novo

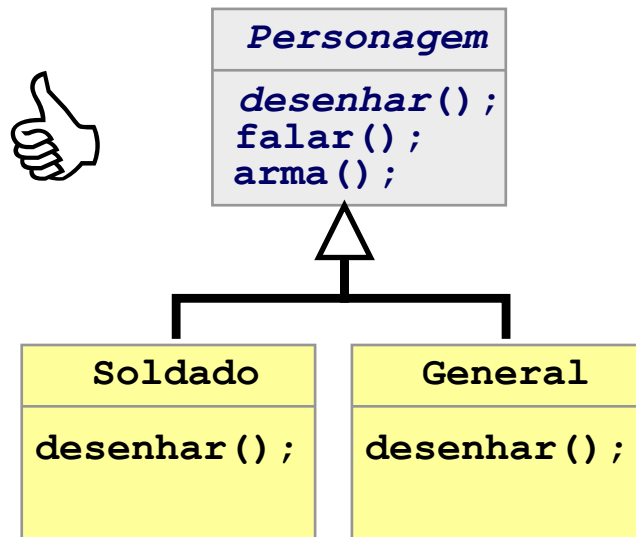


ligação
dinâmica

```
...
void abrir(Forma f) {
    f.desenha();
}
...
abrir(new Retangulo());
```

Cenário de uso(I)

- Imagine a modelagem de um sistema para um jogo de luta infantil para computador (com personagens)



Classe abstrata que possui a interface comum a todos os personagens

```
public abstract class Personagem {
    public abstract void desenhar();

    public void falar(){
        /* código comum para falar */
    }
    public void arma(){
        /* código comum para atirar */
    }
}
```

Subclasses redefinem comportamentos específicos

```
public class Soldado
    extends Personagem{
    public void desenhar(){
        /* desenha o soldado */;
    }
}
```

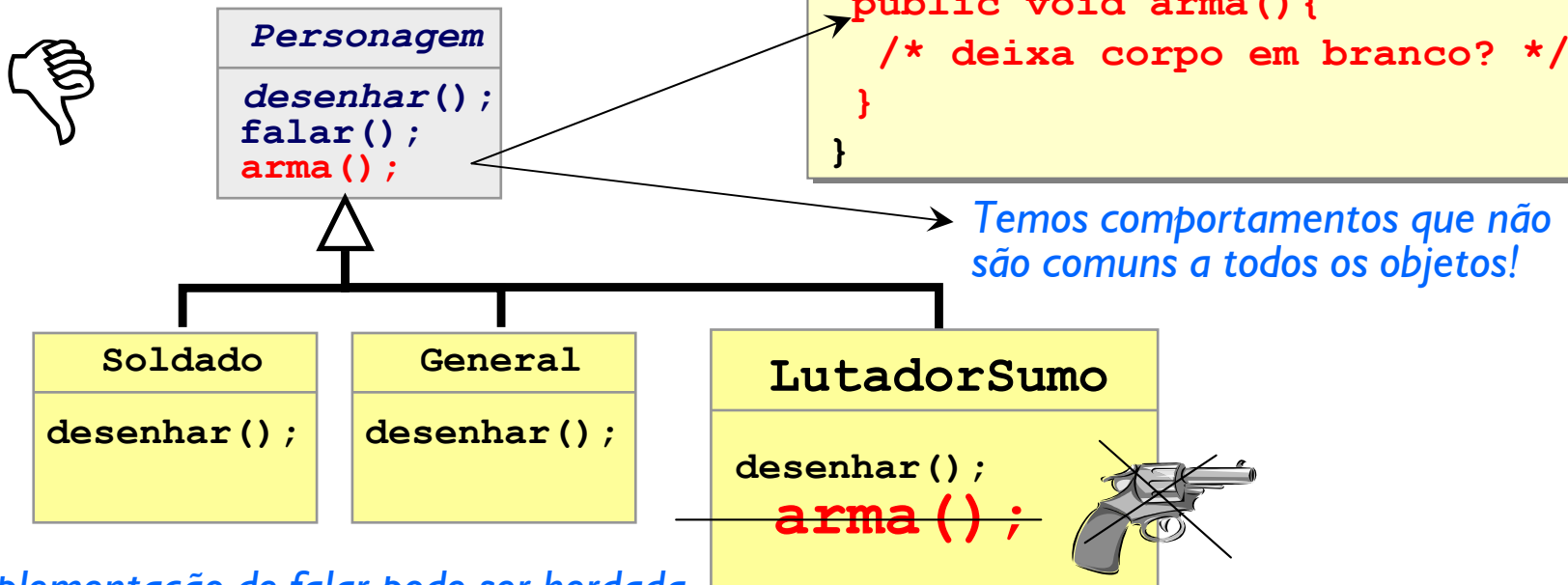
Implementações de falar e arma
serão usadas da superclasse

```
public class General
    extends Personagem {
    public void desenhar(){
        /* desenha o general */;
    }
}
```

Implementações de falar e arma
serão usadas da superclasse

Cenário de uso(I)

■ Um novo personagem:



Implementação de falar pode ser herdada da superclasse Personagem.



Desenhar pode ser redefinido, mas...

❌ o lutador de SUMÔ não deve atirar!

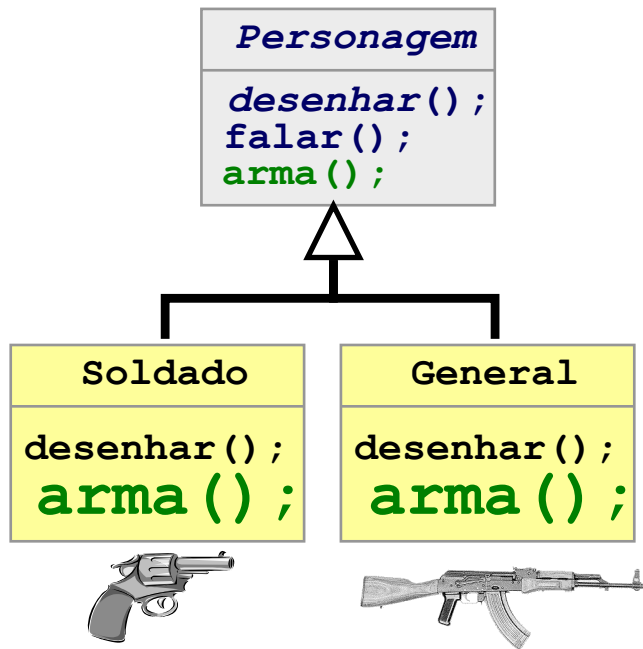
Tentativa: E se tirarmos o método arma da superclasse Personagem e colocá-lo em cada uma das subclasses? (duplicação!)

Problemas com a Herança

- Alguns autores afirmam que a Herança “fere” o princípio do encapsulamento
 - Mudanças na superclasse podem ser difíceis
 - Subclasses recebem/enxergam os dados e a implementação da superclasse
- Mesmo assim, hoje, muitos programadores consideram ainda a herança como a ferramenta básica de extensão e reuso de código
- Outro grande problema é **a mudança de comportamento dinamicamente** (em tempo de execução)
 - Veja no próximo slide...

Cenário de uso(2)

- Imagine a mesma modelagem com General e Soldado usando armas de fogo (nada novo)



- ✓ As subclasses podem redefinir métodos para um comportamento específico

```
public class Soldado
    extends Personagem{
    public void desenhar(){
        /* desenha o soldado */;
    }
    public void arma(){
        System.out.print("Tiro")
    }
}
```

```
public class General
    extends Personagem{
    public void desenhar(){
        /* desenha o general */;
    }
    public void arma(){
        System.out.print("Rajada")
    }
}
```


Cenário de uso(2)

- *Problema: trocar a arma do soldado por uma outra qualquer (inclusive futuras) dinamicamente*

```
public class UsaPersonagem {  
    public static void main(String[] args) {  
        Personagem p;  
  
        p = new Soldado();  
        p.desenha();  
        p.arma(); // imprime "Tiro"  
    }  
}
```

A arma (revólver) está parafusada no código da classe Soldado

```
public class Soldado  
    extends Personagem{  
    public void desenhar(){  
        /* desenha o soldado */;  
    }  
    public void arma(){  
        System.out.print("Tiro")  
    }  
}
```

```
public void arma(int arma){  
    if(arma == 0){  
        // imprime "Tiro"  
    }else {  
        // imprime "Rajada"  
    }  
}
```



O que acontece quando novas armas surgirem no jogo em cada uma das classes de personagens?

Tentativa: E se colocarmos um IF dentro do método arma na classe Soldado para decidir qual arma ele deve usar dentre as possíveis armas do jogo?

Problemas com a Herança

- Não estamos tendo sucesso nestes cenários (1 e 2) do Jogo com herança. Por que?
 - No primeiro cenário, existem comportamentos na superclasse que **não são comuns a todos** os personagens do jogo.
 - No segundo cenário, **o código da arma** específica está **parafusado** em cada uma das classes. Isso não apenas dificulta **a criação de novas armas** para o jogo, mas também não permite que um personagem **mude de arma em tempo de execução**.
- **O que fazer?**

Parte 2 : reuso com composição

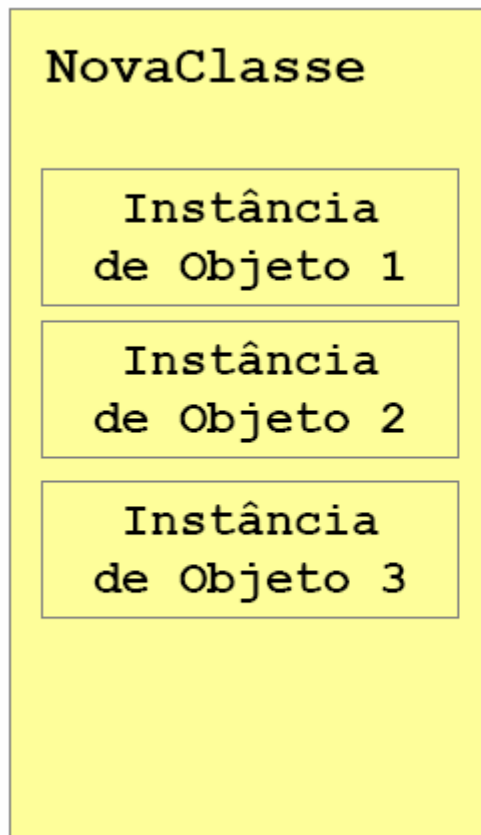
Como aumentar as chances de reuso?

- ***Separar as partes que podem mudar das partes que não mudam***
 - *Depois de descobrir o que irá mudar, a idéia é encapsular isso, trabalhar com uma interface e usar o código sem a preocupação de ter que reescrever tudo caso surjam versões futuras (fraco acoplamento.)*
- ***Programar para uma interface, e não para uma implementação (concreta)***
 - *A chave é explorar o polimorfismo e a ligação dinâmica para usar um supertipo (interface ou classe abstrata) e poder trocar objetos distintos (que de fato irão responder as mensagens) em tempo de execução*

Composição

- *A composição estende uma classe pela delegação de trabalho para outro objeto*
 - *Em vez de codificar um comportamento estaticamente, definimos e encapsulamos pequenos comportamentos padrão e usamos **composição para delegar** comportamentos*
- *Muitos autores já consideram a composição muito superior à herança em grande parte dos casos*
 - *Posso mudar a associação entre classes em tempo de execução (estou trabalhando com uma interface!)*
 - *Permite que um objeto assuma mais de um comportamento*
- *A herança deve ser utilizada em alguns (relativamente poucos) contextos*
 - *Herança acopla as classes demais e engessa o programa*

Composição em Java



```
class NovaClasse {
    Um um = new Um();
    Dois dois = new Dois();
    Tres tres = new Tres();
}
```

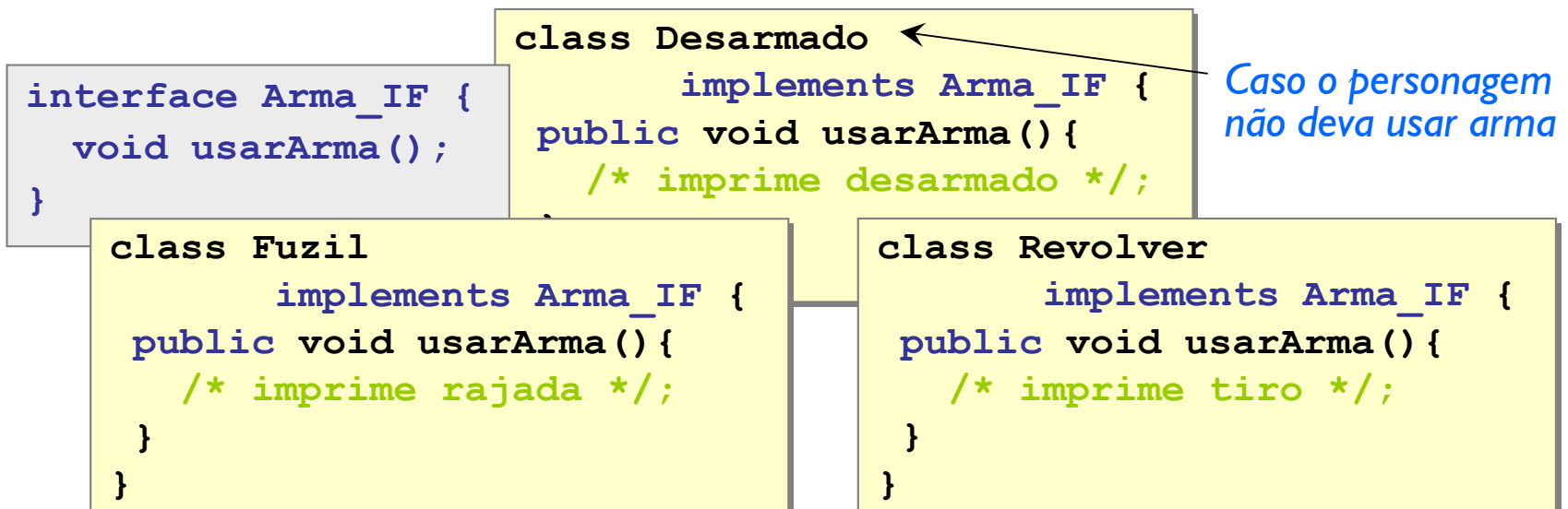
- *Objetos podem ser inicializados no construtor*
- *Flexibilidade*
 - *Pode trocar objetos durante a execução!*
- *Relacionamento*
 - *"TEM UM"*

O que se procura?

- *O sistema deve permitir o aparecimento de novas armas no Jogo sem grandes impactos negativos (if-else's) em todas as subclasses concretas*
- *Além do surgimento de novas armas, o sistema deve permitir que um personagem mude de arma em tempo de execução (passagem de fases, e.g.)*
 - *Em vez de já definir a arma estaticamente no código*
- *Alguns comportamentos não devem ser compartilhados por todos os personagens*
 - *Ex.: atirar com um revólver ser executado por um objeto do tipo lutador de Sumô*

Cenário de uso(3)

- Se a armas devem mudar, a partir de agora, deve-se separá-las das classes concretas (soldado, general)
 - Elas irão viver em uma classe que implementará uma **interface** em particular. Dessa forma, Soldado e General não vão precisar saber nenhum detalhe de implementação de seus comportamentos dinâmicos



Cenário de uso(3)

- Como foi dito, todas os personagens concretos devem trabalhar com uma **referência para alguma coisa que implemente a interface** de armas
 - Cada personagem irá **delegar seu comportamento** de arma em vez definir isso estaticamente no código

```
public abstract class Personagem {  
    Arma_IF arma;   
  
    public abstract void desenhar();  
    public void falar() {  
        /* código comum para falar */  
    }  
    public void setArma(Arma_IF a) {  
        arma = a;  
    }  
    public void arma() {  
        arma.usarArma();  
    }  
}
```

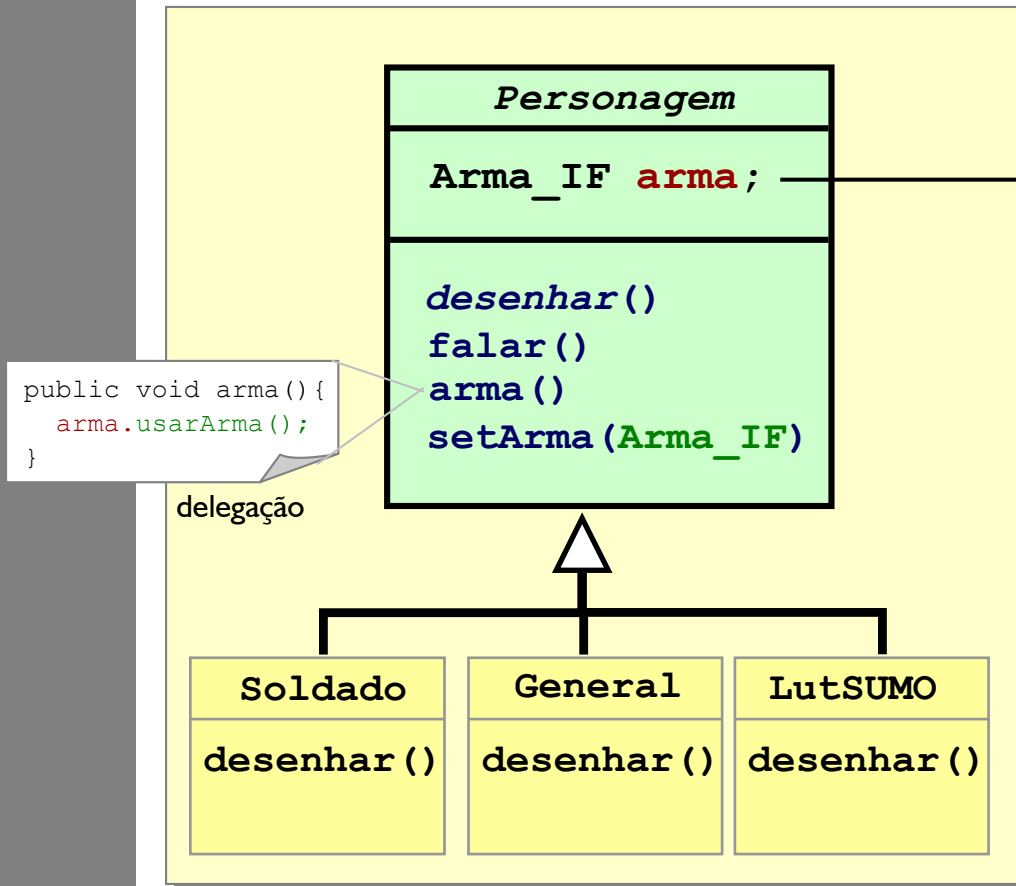
Reduz o acoplamento de código, já que os personagens interagem com interface (em vez de uma implementação)

Se um personagem deseja usar sua arma, ele simplesmente delega esta tarefa ao objeto (alguém que tem o método usarArma) que está sendo referenciado no momento

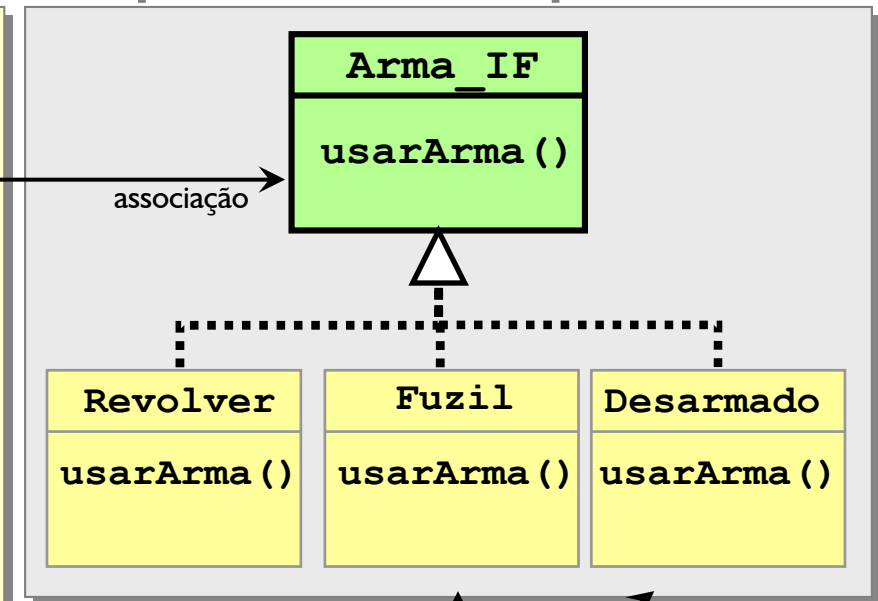
O polimorfismo e a ligação dinâmica irão cuidar de chamar o método (usarArma) correto

Diagrama de classes final

Cliente



Comportamento encapsulado



Estes algoritmos (comportamentos) são perfeitamente intercambiáveis

O cliente faz uso de uma família encapsulada de tipos de armas

Executando novo código

■ A troca de armas (dinamicamente):

```
public abstract class Personagem {
    Arma_IF arma;

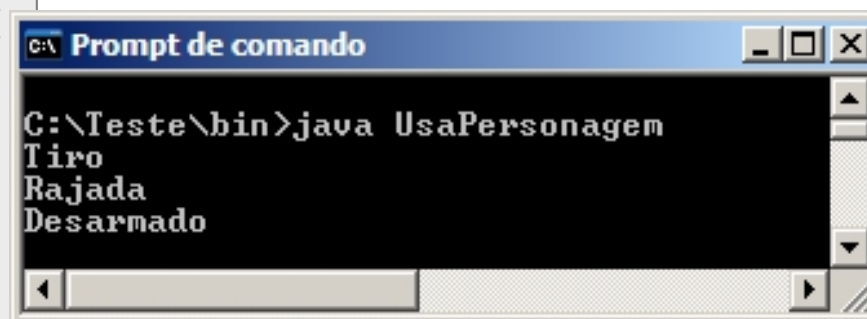
    public abstract void desenhar();
    public void falar() {
        /* código comum para falar */
    }

    public void setArma(Arma_IF a) {
        arma = a;
    }

    public void arma() {
        arma.usarArma();
    }
}
```

Note a presença de interfaces

Ligação dinâmica



```
C:\Teste\bin>java UsaPersonagem
Tiro
Rajada
Desarmado
```

```
public class UsaPersonagem {
    public static void main(String[] args) {
        Personagem p;

        p = new Soldado();
        p.desenhar();
        p.setArma(new Revolver()); // define arma
        p.arma(); // imprime "Tiro"
        p.setArma(new Fuzil()); // trocou arma
        p.arma(); // imprime Rajada

        p = new LutadorSumo();
        p.desenhar();
        p.setArma(new Desarmado()); // define arma
        p.arma(); // imprime "Desarmado"
    }
}
```

* E se amanhã surgisse uma nova arma (Faca, por ex.)?

Considerações: composição e herança

- *Uma das principais atividades em um projeto orientado a objetos é estabelecer relacionamentos entre classes*
 - *Duas formas básicas de relacionar classes são: herança e a composição*
- *Composição e herança não são mutuamente exclusivas*
 - *As técnicas podem ser usadas em conjunto para obter os melhores resultados de cada uma*
- *A herança, geralmente, ocorre mais no design de tipos (uma subclasse é um tipo de...)*

Considerações: composição e herança

- *No desenvolvimento, porém, a composição de objetos é a técnica predominante*
 - *Separar o que muda do que não muda (e encapsular estes pequenos comportamentos)*
 - *Trabalhar com uma interface para manipular estes pequenos comportamentos*
 - *Trocar comportamentos dinamicamente*
- *Programar para uma interface sempre que possível*
 - *Isso garante um fraco acoplamento entre o código-cliente e as classes concretas (até futuras)*

Quando usar? Composição ou Herança?

- ❶ *Identifique os componentes do objeto, suas partes*
 - *Essas partes devem ser agregadas ao objeto via composição (relacionamento do tipo **é parte de**)*
- ❷ *Classifique seu objeto e tente encontrar uma semelhança de identidade com classes existentes*
 - *Herança só deve ser usada se você puder comparar seu objeto A com outro B dizendo que A “**É UM tipo de...**” B*
 - *Tipicamente, herança só deve ser usada quando você estiver construindo uma família de tipos (relacionados entre si)*
 - **LEIA!** <http://www.javaworld.com/javaworld/jw-11-1998/jw-11-techniques.html>

① *Implemente e execute os exemplos mostrados*

- *Coloque texto em cada método (um `println()` para mostrar qual método foi chamado e descrever o que aconteceu)*
- *Faça experimentos: (a) insira uma nova arma para os jogadores (uma faca, por ex.); (b) insira novos personagens (um mago que possa soltar alguma magia como arma para se defender)*

② *Para os alunos de excepcional intelecto:*

- *O que deve ser alterado no sistema para acomodar um personagem DragãoAlado no sistema?*
- *Um dragão pode `falar()`? E qual é a `arma()` do dragão?*