Universidade Federal da Paraíba – Campus I
Centro de Informática
Departamento de Sistemas e Computação

Métodos e Projeto de Software

Material 7: Revisão de Conceitos Básicos de O.O - Parte 2

Prof. Raoni Kulesza
raoni@ci.ufpb.br

Objetivos

- Este módulo explora detalhes da construção de classes e objetos
  - Construtores
  - Implicações da herança
  - Palavras super e `this`, usadas como referências para o objeto corrente e a `super` classe
  - Instruções `super()` e `this()` usadas para chamar construtores durante a criação de objetos
  - Detalhes sobre a inicialização de objetos e possíveis problema

Criação e destruição de objetos

- Para a criação de novos objetos, Java garante que cada classe tenha um construtor
  - O construtor default recebe zero argumentos
  - Faz apenas inicialização da superclasse
- Programador pode criar um construtor explicitamente e determinar suas operações de inicializaçã
  - Inicialização pela superclasse continua garantida
  - Construtor default deixa de existir
- Objetos são destruídos automaticamente pelo sistema, porém, sistema não faz finalização
  - Método `finalize()`, herdado de Object, teoricamente permite ao programador controlar a finalização de qualquer objeto finalize() não funciona 95% das vezes -não use!
  - Se precisar de finalização, coloque seu código em um bloco `try {...} finally {...}`

Construtores e sobrecarga

- Construtores default (sem argumentos) só existem quando não há construtores definidos explicitamente no código
  - A criação de um construtor explícito substitui o construtor fornecido implicitamente
- Uma classe pode ter vários construtores (isto se chama sobrecarga de nomes)
  - Distinção é feita pelo número e tipo de argumentos (ou seja, pela assinatura do construtor)
- A assinatura é a identidade do método. É pela assinatura que ele se distingue dos outros métodos. Consiste de
  - Tipo de retorno
  - Nome
  - Tipo de argumentos
  - Quantidade de argumentos

Sobrecarga de métodos

- Uma classe também pode ter vários métodos com o mesmo nome (sobrecarga de nomes de métodos)
- Distinção é feita pela assinatura: tipo e número de argumentos, assim como construtores
- Apesar de fazer parte da assinatura, o tipo de retorno não pode ser usado para distinguir métodos sobrecarregados
- Na chamada de um método, seus parâmetros são passados da mesma forma que em uma atribuição
  - Valores são passados em tipos primitivos
  - Referências são passadas em objetos
  - Há promoção de tipos de acordo com as regras de conversão de primitivos e objetos
  - Em casos onde a conversão direta não é permitida, é preciso usar operadores de coerção(cast)

Distinção de métodos na sobrecarga

- Métodos sobrecarregados devem ser diferentes o suficiente para evitar ambigüidade na chamada
- Qual dos métodos abaixo …

```
int metodo (long x, int y) {...}

int metodo (int x, long y) {...}
```

- Será chamado pela instrução abaixo

```
int x = metodo (5,6)
```

- O compilador detecta essas situações

this()

- Em classes com múltiplos construtores, que realizam tarefas semelhantes, **`this()`** pode ser usado para chamar outro construtor local, identificado pela sua assinatura (número e tipo de argumentos)

```
public class Livro {
   private String titulo;
   public Livro() {
      titulo = "Sem titulo";
   }

   public Livro(String titulo) {
      this.titulo = titulo;
   }
}
```

```
public class Livro {
   private String titulo;
   public Livro() {
      this("Sem titulo");
   }

   public Livro(String titulo) {
      this.titulo = titulo;
   }
}
```

super()

- Todo construtor chama algum construtor de sua superclasse
  - Por default, chama-se o construtor sem argumentos, através do comando **`super()`** (implícito)
  - Pode-se chamar outro construtor, identificando-o através dos seus argumentos (número e tipo) na instrução **`super()`**
  - **`super()`**, se presente, deve sempre ser a primeira instrução do construtor (substitui o **`super()`** implícito)
- Se a classe tiver um construtor explícito, com argumentos, subclasses precisam chamá-lo diretamente
  - Não existe mais construtor default na classe

this e super

- A palavra **`this`** é usada para referenciar membros de um objeto
  - Não pode ser usada dentro de blocos estáticos (não existe objeto atual '**this**' em métodos estáticos)
  - É obrigatória quando há ambiguidade entre variáveis locais e variáveis de instância
- **`Super`** é usada para referenciar os valores originais de variáveis ou as implementações originais de métodos sobrepostos

```
class Numero {
    public int x = 10;
}
```

```
class OutroNumero extends Numero {
    public int x = 20;
    public int total() {
        return this.x + super.x;
    }
}
```

20

10

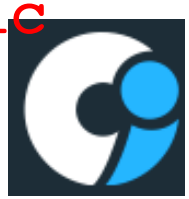
- Não confunda **`this`** e **`super`** com **`this()`** e **`super()`**
  - Os últimos são usados apenas em construtores!

Inicialização estática

- Para inicializar valores estáticos, é preciso atuar logo após a carga da classe
  - O bloco 'static' tem essa finalidade
  - Pode estar em qualquer lugar da classe, mas será chamado antes de qualquer outro método ou variável

```
class UmaClasse {
    private static Point[] p = new Point[10];

    static {
        for (int i = 0; i < 10; i++) {
            p[i]= new Point(i, i);
        }
    }
}
```

- Não é possível prever em que ordem os blocos `static` serão executados, portanto: só tenha um!



Universidade Federal da Paraíba – Campus I
Centro de Informática
Departamento de Sistemas e Computação

Métodos e Projeto de Software

Slides 5: Herança

Prof. Raoni Kulesza
raoni@ci.ufpb.br

Objetivos

- Revisar os conceitos fundamentais e benefícios acerca do uso de *Herança* na linguagem Java
- Alguns cuidados com o uso *Herança* e construtores default
- Discutir problemas do uso irrestrito do uso de herança

Reuso de código

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

Reuso com Herança

Herança

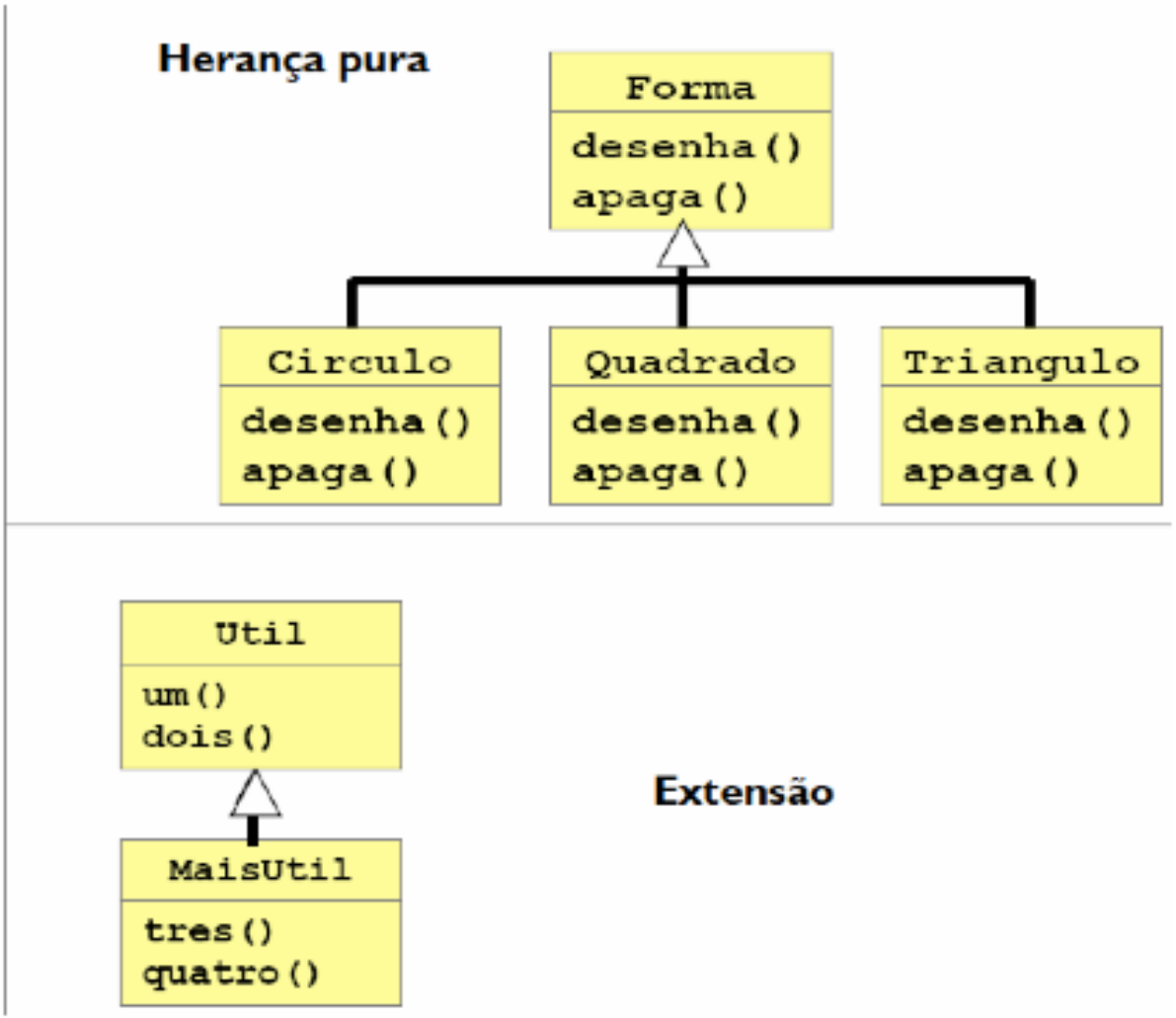
- Permite reutilizar as características de uma classe na definição de outra classe
- Reutilização direta de código previamente definido por alguém em uma superclasse
- Terminologias relacionadas à Herança:
  - Classes mais generalizadas: superclasses
  - Mais especializadas: subclasses
- Classes estão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

Herança

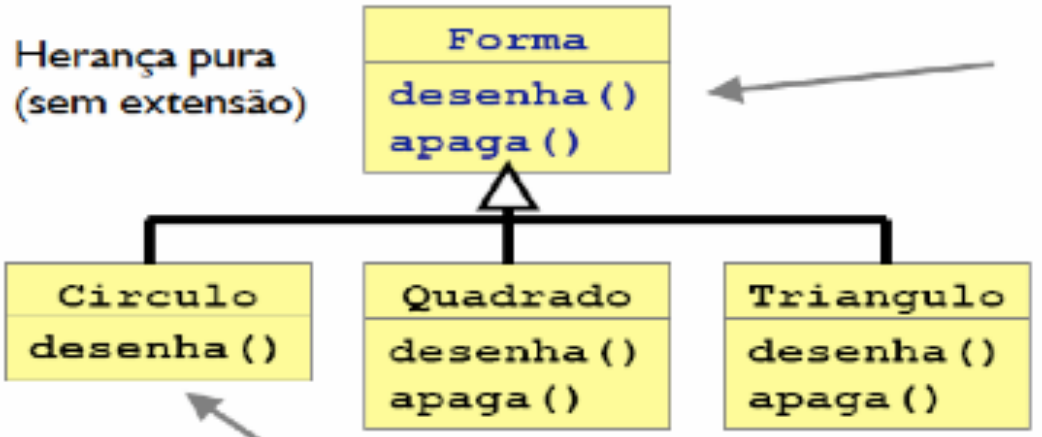
- Propriedades, conexões a objetos e métodos comuns ficam na superclasse (classe de generalização)
- Adicionamos mais dessas coisas nas subclasses (classes de especialização)
- A herança viabiliza a construção de sistemas a partir de componentes facilmente reutilizáveis
- A classe descendente não tem trabalho para receber a herança

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)



Árvore de Herança



interface original é automaticamente duplicada nas classes derivadas

Campos de dados da classe base também são duplicados

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)*

Herança

- Classes que herdam entre si gerando uma árvore de herança
  - Todos os objetos herdam características (gerais) definidas em Forma
  - Círculo, Quadrado e Triângulo são especializações de Forma (é uma Forma)
- Cada subclasse possui uma única superclasse
  - A isso, chamamos de herança simples
  - Em algumas linguagens, é possível herdar a partir de diversas superclasses

Herança

- 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 com classes! C++ sim!

Herança

- 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
  - Métodos declarados como private não serão herdados
- Construtores também não são herdados
  - Serão chamados (em cascata) na construção de objetos especializados(super()). Vimos isso na aula passada.

Herança – Benefícios (1)

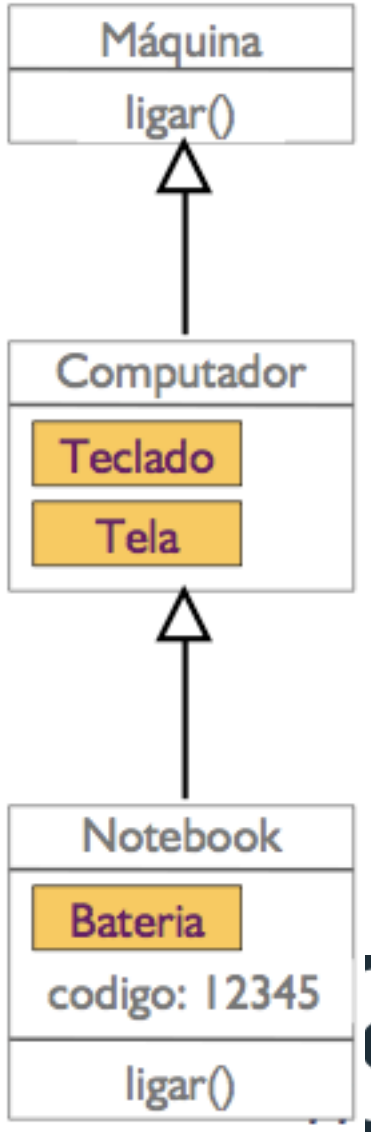
- Como código pode ser facilmente reutilizado, a quantidade de código a ser adicionado numa (sub)classe pode diminuir bastante
  - Subclasses provêem comportamentos especializados tomando como base os elementos comuns
- Potencializa a manutenção de sistemas
  - Maior legibilidade do código existente
  - A herança é vista diretamente no código

Problemas - Inicialização de instâncias

- O que acontece quando um objeto é criado usando **`new NomeDaClasse()`**?
  1. Inicialização default de campos de dados (0, null, false)
  2. Chamada recursiva ao construtor da superclasse (até **`Object`**)
     1. Inicialização default dos campos de dados da superclasse (recursivo, subindo a hierarquia)
     2. Inicialização explícita dos campos de dados
     3. Execução do conteúdo do construtor (a partir de **`Object`**, descendo a hierarquia)
  3. Inicialização explícita dos campos de dados
  4. Execução do conteúdo do construtor

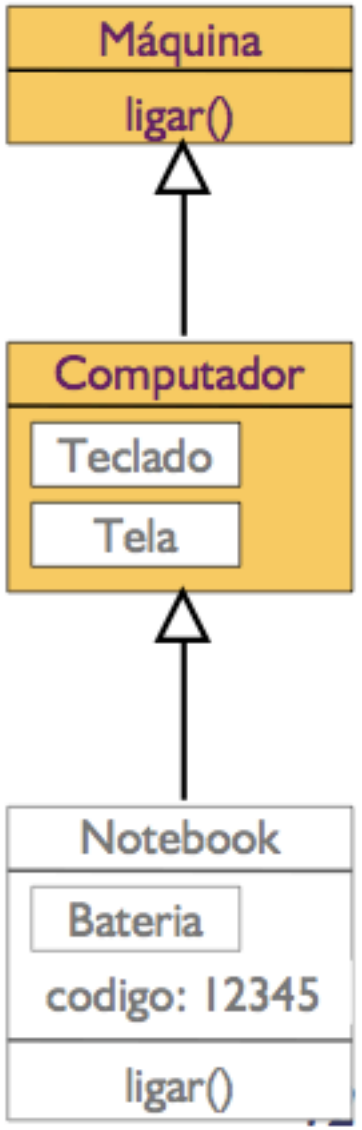
Problemas - Exemplo (1)

```
class Bateria {
  public Bateria() {
    System.out.println("Bateria()");
  }
}

class Tela {
  public Tela() {
    System.out.println("Tela()");
  }
}

class Teclado {
  public Teclado() {
    System.out.println("Teclado()");
  }
}
```



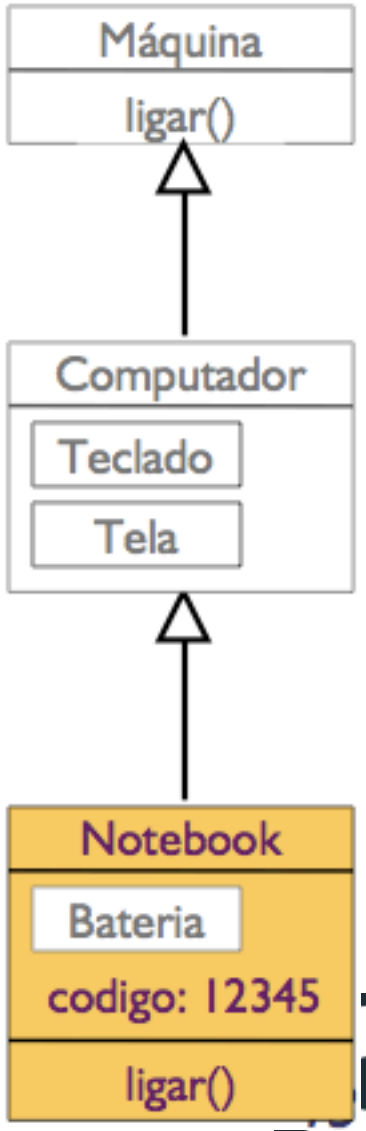
Problemas - Exemplo (2)

```
class Maquina {
  public Maquina() {
      System.out.println("Maquina()");
      this.ligar();
  }
  public void ligar() {
    System.out.println("Maquina.ligar()");
  }
}

class Computador extends Maquina {
  public Tela tela = new Tela();
  public Teclado teclado = new Teclado();
  public Computador() {
      System.out.println("Computador()");
  }
}
```

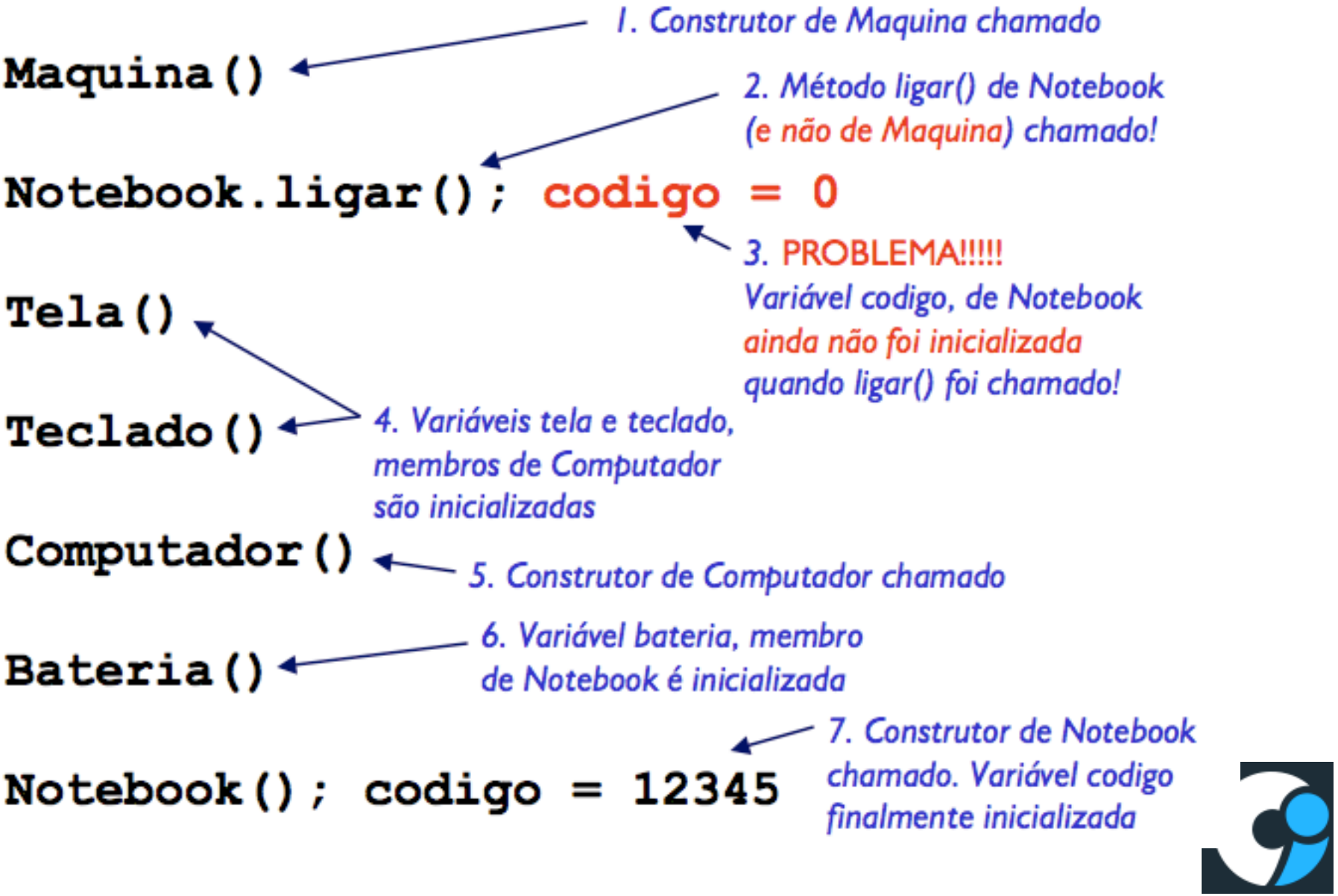


Problemas - Exemplo (3)

```
class Notebook extends Computador {
  int codigo = 12345;
  public Bateria bateria = new Bateria();
  public Notebook() {
    System.out.print("Notebook(); " +
                     "codigo = "+codigo);
  }
  public void ligar() {
    System.out.println("Notebook.ligar();" +
                       " codigo = "+ codigo);
  }
}

public class Run {
  public static void main (String[] args) {
  new Notebook();
  }
}
```



Resultado no new Notebook()



Detalhes

N1. new Notebook() chamado
N2. variável código iniciaizada: 0
N3. variável bateria iniciaizada: null
N4. super() chamado (Computador)

C1. variável teclado iniciaizada: null
C2. variável tela iniclaizada: null
C3. super() chamado (Maquina)

M2. super() chamado (Object)

M2. Corpo de Maquina() executado:
println() e this.ligar()

C4: Construtor de Teclado chamado

Tk1: super() chamado (Object)

C5. referência teclado inicializada
C6: Construtor de Tela chamado

Te1: super() chamado (Object)

C7: referência tela inicializada
C8: Corpo de Computador()
executado: println()

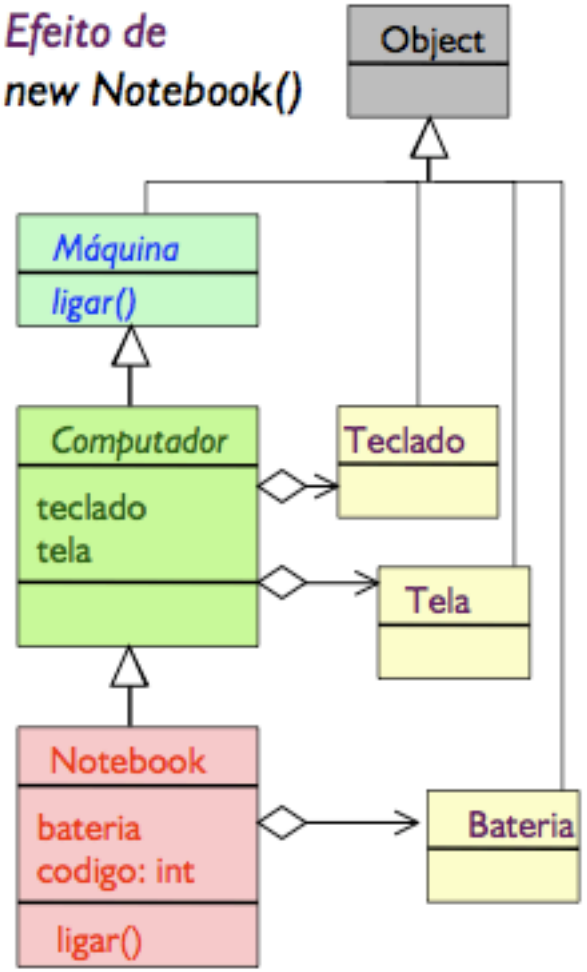
N5. Construtor de Bateria chamado

B1: super() chamado (Object)

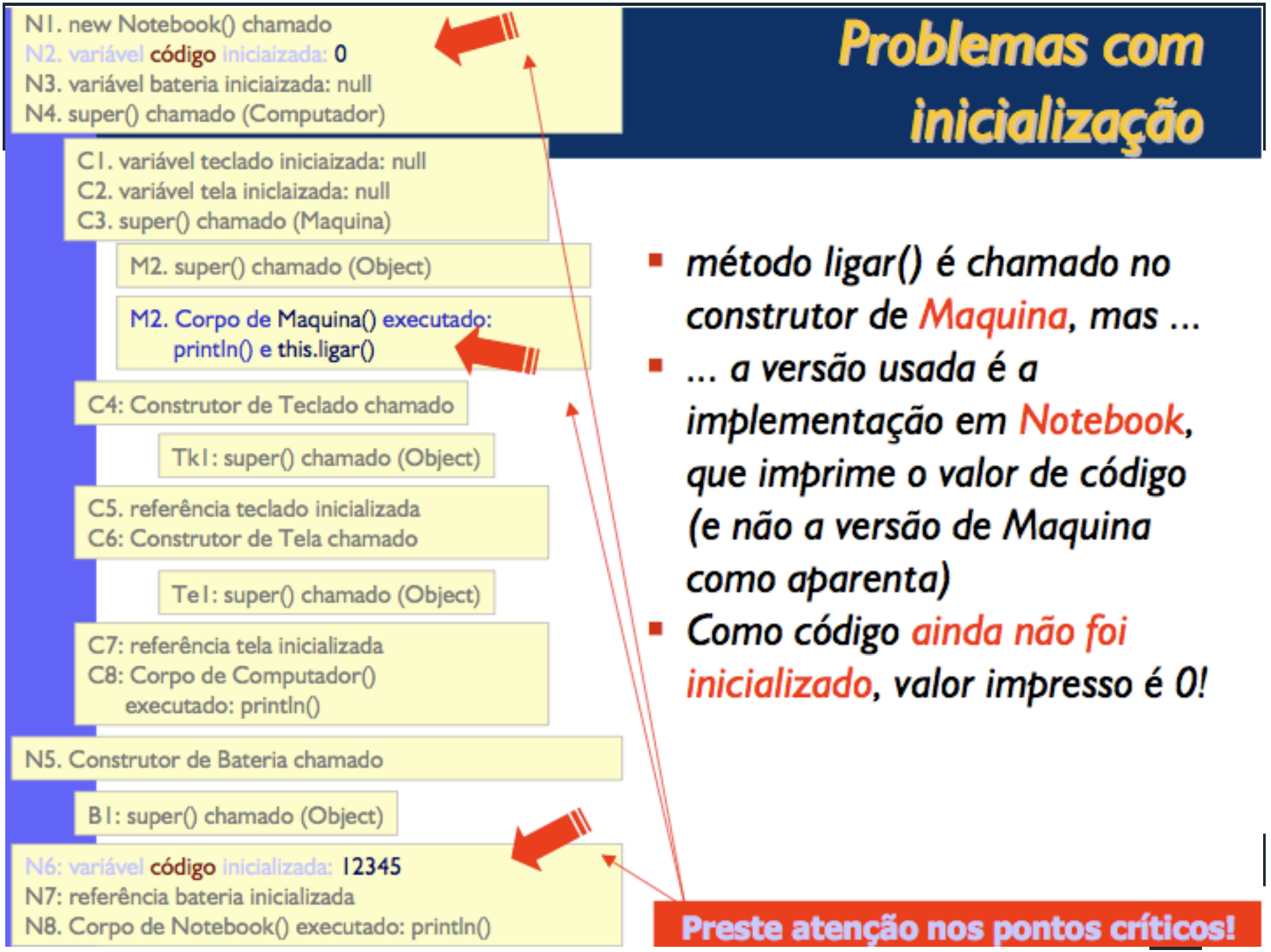
N6: variável código inicializada: 12345
N7: referência bateria inicializada
N8. Corpo de Notebook() executado: println()

O1. Campos inicializados
O2. Corpo de Object() executado

Efeito de
new Notebook()



Problemas com inicialização



- *método ligar() é chamado no construtor de Maquina, mas ...*
- *... a versão usada é a implementação em Notebook, que imprime o valor de código (e não a versão de Maquina como aparenta)*
- *Como código ainda não foi inicializado, valor impresso é 0!*

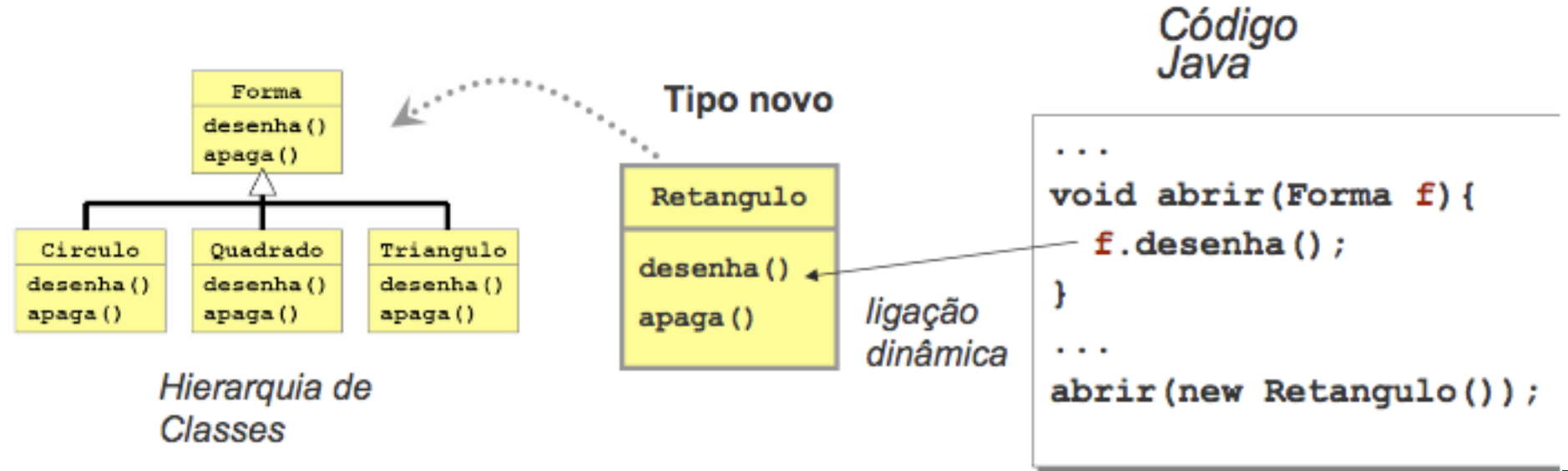
Preste atenção nos pontos críticos!

Como evitar este problema?

- Evite chamar métodos locais dentro de construtores
  - Construtor (qualquer um da hierarquia) sempre usa versão sobreposta do método
- Isto pode trazer resultados inesperados se alguém estender a sua classe com uma nova implementação do método que
  - Dependa de variáveis da classe estendida
  - Chame métodos em objetos que ainda serão criados (provoca `NullPointerException`)
  - Dependa de outros métodos sobrepostos
- Use apenas métodos finais em construtores
- Métodos declarados com modificador final não podem ser sobrepostos em subclasses

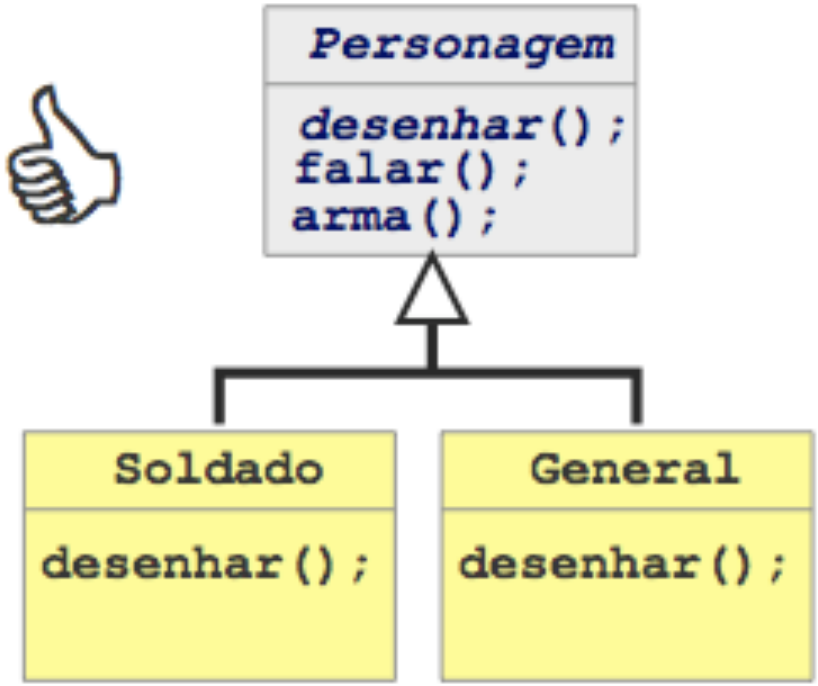
Herança – Benefícios (2)

- 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.)



Problemas - Cenário de Uso 1

- Imagine a modelagem de um sistema para um jogo de luta para computador (com personagens) Neste Jogo existem dois tipos de personagens
  - Soldado
  - General



Problemas - Cenário de Uso 1

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 */
 }
}
```

Implementações de falar e arma serão usadas da superclasse

Subclasses redefinem comportamentos específicos

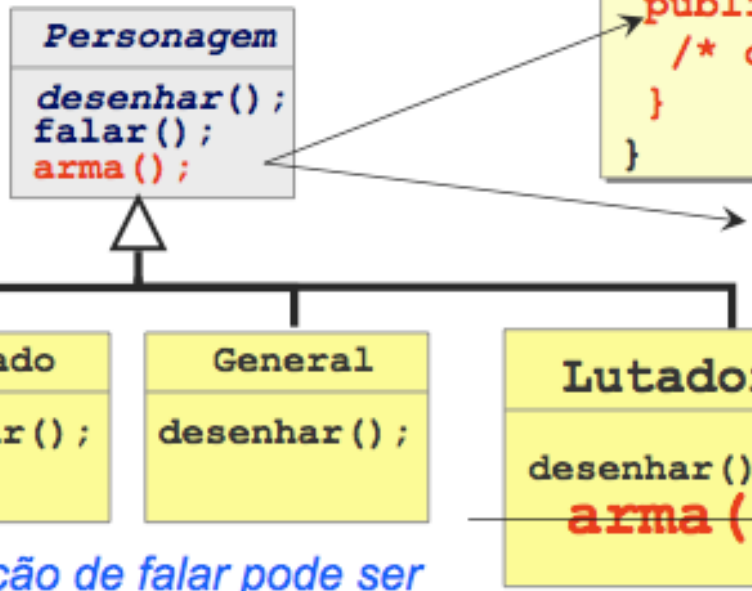
```
public class Soldado
        extends Personagem{
 public void desenhar(){
   /* desenha o soldado */;
 }
}
```

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

Problemas - Cenário de Uso 1

```
public class LutadorSumo
          extends Personagem {
 public void desenhar(){
    /* desenha o lutador */;
 }
 public void arma(){
   /* deixa corpo em branco? */
 }
}
```

Um novo personagem:



Temos comportamentos que não são comuns a todos os objetos!

- ☑ *Implementação de falar pode ser herdada da superclasse Personagem.*
- ☑ *Desenhar pode ser redefinido, mas...*
- ☒ ***o lutador de SUMÔ não deve atirar!***

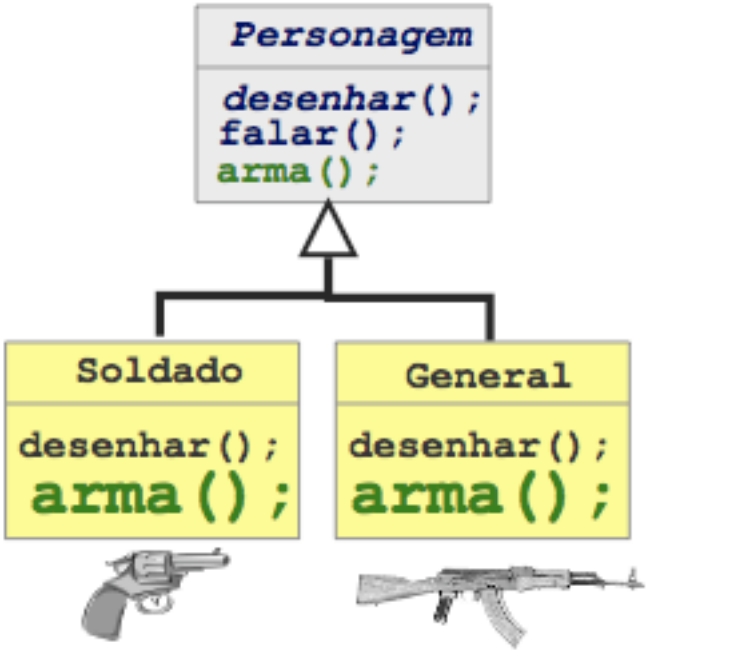
Tentativa: ***E se tirarmos o método arma da superclasse Personagem e colocá-lo em cada uma das subclasses? (duplicação!)***

Herança – Problemas

- Encapsulamento entre classes e subclasses é fraco (forte acoplamento)
  - Mudanças na superclasse podem afetar todas a subclasses
- Outro problema é a mudança de comportamento dinamicamente (em tempo de execução)

Problemas - Cenário de uso 2

- 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")
 }
}
```

Problemas - Cenário de uso 2

- Problema: trocar a arma 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?

Herança – Como resolver os problemas?

- Não estamos tendo sucesso nestes cenários 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 dificulta a criação de novas armas para o jogo e não permite que um personagem mude de arma em tempo de execução.
- O que fazer?

Dúvidas?

raoni@ci.ufpb.br