

Reuso com Herança e Composição

Helder da Rocha
www.argonavis.com.br

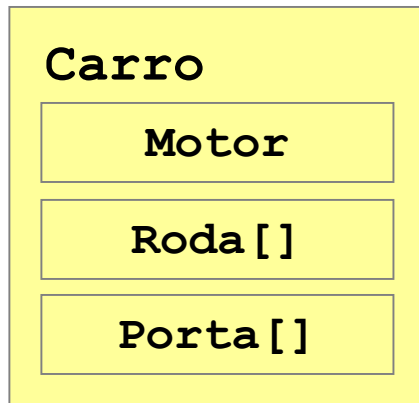
Como aumentar as chances de reuso

- Separar as partes que podem mudar das partes que não mudam. Exemplo: bibliotecas
 - *Programador cliente* deve poder usar o código sem a preocupação de ter que reescrever seu código caso surjam versões futuras
 - *Programador de biblioteca* deve ter a liberdade de fazer melhoramentos sabendo que o cliente não terá que modificar o seu código
- Em Java: esconder do cliente
 - Métodos que não fazem parte da interface de uso
 - Métodos que não fazem parte da interface de herança
 - Todos os atributos de dados

- *Quando você precisa de uma classe, você pode*
 - *Usar uma classe que faz exatamente o que você deseja fazer*
 - *Escrever uma classe do zero*
 - *Reutilizar uma classe existente com composição*
 - *Reutilizar uma classe existente ou estrutura de classes com herança*

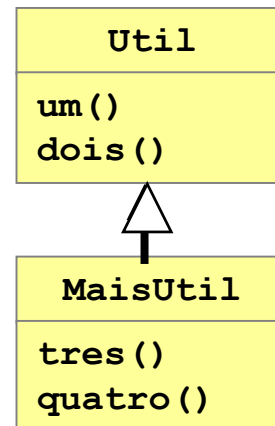
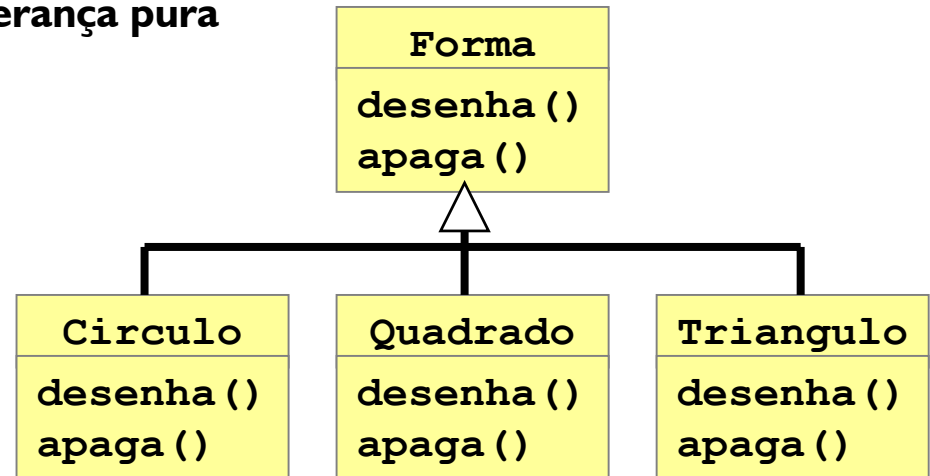
Composição vs. Herança

Composição pura



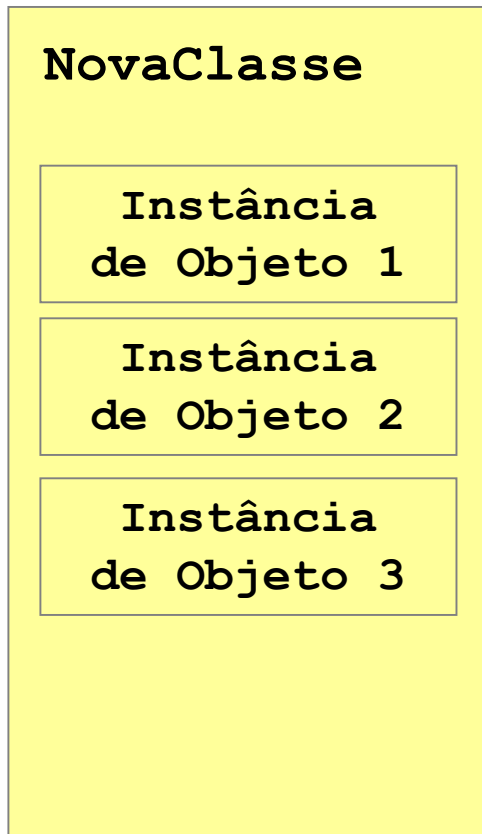
Ao reutilizar uma classe a composição deve ser sua escolha preferencial

Herança pura



Extensão

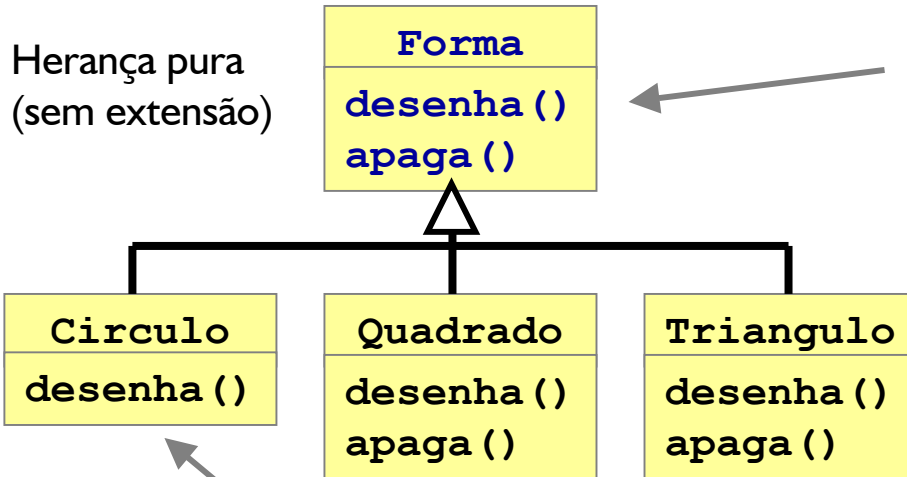
Composição em Java



```
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"*

Herança em Java



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)

Composição e Herança

- *Composição e herança não são mutuamente exclusivas*
 - *As técnicas podem ser usadas em conjunto para obter os melhores resultados de cada uma*
 - *No desenvolvimento, composição é a técnica predominante*
 - *Herança geralmente ocorre mais no design de tipos*

Quando usar?

Composição ou herança?

- 1. Identifique os componentes do objeto, suas partes
 - Essas partes devem ser agregadas ao objeto via composição (*é parte de*)
- 2. 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)

Modificadores relacionados

- No projeto de uma classe, é preciso definir **duas interfaces**
 - Interface para uso de classes via composição
 - Interface para uso de classes via herança
- A palavra **protected** deve ser usada para declarar os métodos, construtores e variáveis que destinam-se à interface de herança
- Elementos usados em interface para composição devem ser declarados **public**
- A palavra **final** é usada para limitar o uso das classes, variáveis e métodos quando existe a possibilidade de haver herança: impede que implementações ou valores sejam alterados

- Para declarar uma **constante**, defina-a com um modificador **final**

```
final XIS = 0;
```

```
public static final EPSILON = 12;
```

- Qualquer variável declarada como *final* tem que ser inicializada no momento da declaração
 - Exceção: argumentos constantes em métodos - valores não mudam dentro do método (uso em classes internas)
- Uma constante de tipo primitivo não pode receber outro valor
- Uma constante de referência não pode ser atribuída a um novo objeto
 - O objeto, porém, não é constante (apenas a referência o é). Os atributos do objeto podem ser alterados

- Método declarado como **final** não pode ser sobreposto
- Motivos para declarar um método **final**
 - **Design**: é a versão final (o método está "pronto")
 - **Eficiência**: compilador pode embutir o código do método no lugar da chamada e evitar realizar chamadas em tempo de execução: pode limitar o uso de sua classe
 - **Procedimentos chamados de dentro de construtores**: quaisquer métodos chamados de dentro de construtores devem ser final.
- Métodos declarados como **private** são implicitamente **final**

- A classe também pode ser declarada **final**

```
public final class Definitiva { ... }
```

- Se uma classe não-final tiver todos os seus métodos declarados como final, é possível herdar os métodos, acrescentar novos, mas não sobrepor
- Se uma classe for declarada como final
 - Não é possível estender a classe (a classe nunca poderá aparecer após a cláusula **extends** de outra classe)
 - Todos os métodos da classe são finais
- Útil em classes que contêm funções utilitárias e constantes apenas (ex: classe *Math*)

Modificadores de acesso

- *Em ordem crescente de acesso*
 - *private*
 - *"package-private"*
 - *modificador ausente*
 - *protected*
 - *public*

- **Acessível**
 - *na própria classe*
 - *nas subclasses*
 - *nas classes do mesmo pacote*
 - *em todas as outras classes*
- **Use para**
 - *construtores e métodos que fazem parte da interface do objeto*
 - *métodos estáticos utilitários*
 - *constantes (estáticas) utilitárias*
- **Evite usar em**
 - *construtores e métodos de uso restrito*
 - *campos de dados de objetos*

Classe
+ campoPublico: tipo
+ metodoPublico: tipo

- **Acessível**
 - *na própria classe*
 - *nas subclasses*
 - *nas classes do mesmo pacote*
- **Use para**
 - *construtores que só devem ser chamados pelas subclasses (através de super())*
 - *métodos que só devem ser usados se sobrepostos*
- **Evite usar em**
 - *construtores em classes que não criam objetos*
 - *métodos com restrições à sobreposição*
 - *campos de dados de objetos*

Classe
#campoProt: tipo
#metodoProt: tipo

package-private

- **Modificador ausente**
 - se não houver outro modificador de acesso, o acesso é "package-private".
- **Acessível**
 - na própria classe
 - nas classes e subclasses do mesmo pacote
- **Use para**
 - construtores e métodos que só devem ser chamados pelas classes e subclasses do pacote
 - constantes estáticas úteis apenas dentro do pacote
- **Evite usar em**
 - construtores em classes que não criam objetos
 - métodos cujo uso externo seja limitado ou indesejável
 - campos de dados de objetos

Classe
~campoAmigo: tipo
~metodoAmigo: tipo

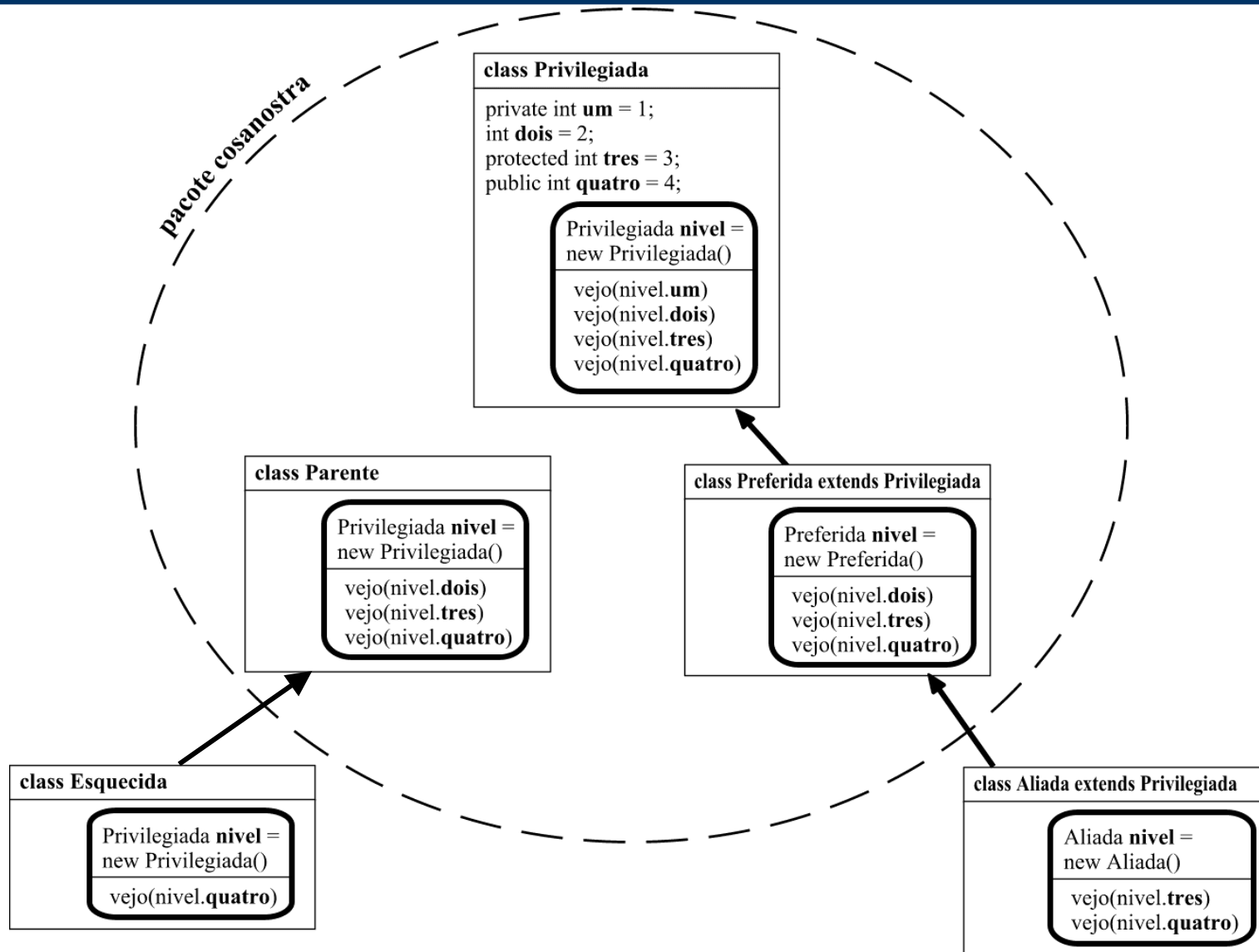
- **Acessível**
 - *na própria classe (nos métodos, funções estáticas, blocos estáticos e construtores)*
- **Use para**
 - *construtores de classes que só devem criar um número limitado de objetos*
 - *métodos que não fazem parte da interface do objeto*
 - *funções estáticas que só têm utilidade dentro da classe*
 - *variáveis e constantes estáticas que não têm utilidade ou não podem ser modificadas fora da classe*
 - *campos de dados de objetos*

Classe
-campoPrivate: tipo
-metodoPrivate: tipo

Observações sobre acesso

- *Classes e interfaces (exceto classes internas)*
 - *Só podem ser package-private ou public*
- *Construtores*
 - *Se private, criação de objetos depende da classe*
 - *Se protected, apenas subclasses (além da própria classe e classes do pacote) podem criar objetos*
- *Váriáveis e constantes*
 - *O acesso afeta sempre a leitura e alteração. Efeitos "read-only" e "write-only" só podem ser obtidos por meio de métodos*
- *Variáveis locais*
 - *Usar modificadores de acesso dentro dos métodos é ilegal e não faz sentido pois variáveis locais só têm escopo local*

- *Métodos sobrepostos nunca podem ter **menos** acesso que os métodos originais*
 - *Se método original for **public**, novas versões têm que ser **public***
 - *Se método original for **protected**, novas versões podem ser **protected** ou **public***
 - *Se método original **não tiver modificador de acesso** (é "package-private"), novas versões podem ser declaradas sem modificador de acesso, com modificador **protected** ou **public***
 - *Se método original for **private**, ele não será visível da subclasse e portanto, jamais poderá ser estendido.*



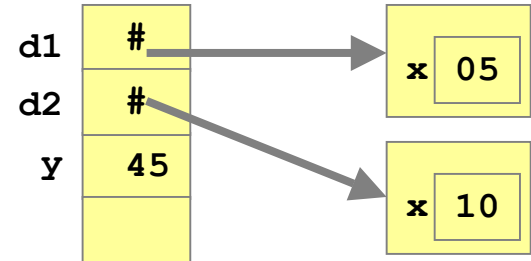
Mais sobre 'static'

- Variáveis declaradas como 'static' existem antes de existir qualquer objeto da classe
 - Só existe uma variável static, independente do número de objetos criado com a classe
 - Podem ser chamadas externamente pelo nome da classe
 - `Color.red`, `System.out`, `BorderLayout.NORTH`
 - Podem também ser chamadas através da referência de um objeto (evite usar este método)

Classe

campoStatic: tipo

metodoStatic: tipo



```
class Duas {  
    int x;  
    static int y;  
}
```

```
(...)  
Duas d1 = new Duas();  
Duas d2 = new Duas();  
d1.x = 5;  
d2.x = 10;  
//Duas.x = 60; // ilegal!  
d1.y = 15;  
d2.y = 30;  
Duas.y = 45;  
(...)
```

mesma variável!

Use esta notação apenas

Métodos static

- Métodos *static* **nunca** são sobrepostos
 - Método *static* de assinatura igual na subclasse apenas "**oculta**" original
 - Não há polimorfismo: método está sempre associado ao tipo da **classe** (e não à instância)
- Exemplo: considere as classes abaixo

```
class Alfa {  
    static void metodo() {  
        System.out.println("Alfa!");  
    }  
}
```

```
class Beta extends Alfa {  
    static void metodo() {  
        System.out.println("Beta!");  
    }  
}
```

- O código a seguir

```
Alfa pai = new Alfa ();  
pai.metodo();  
  
Beta filho1 = new Beta ();  
filho1.metodo();  
  
Alfa filho2 = new Beta ();  
filho2.metodo();
```

irá imprimir:

```
Alfa!  
Beta!  
Alfa!
```

e não... Alfa!
Beta!
Beta!

como ocorreria **se** os métodos fossem de instância

- Não chame métodos *static* via referências! Use sempre:

`Classe.metodo()`

Classe que só permite um objeto (Singleton pattern)

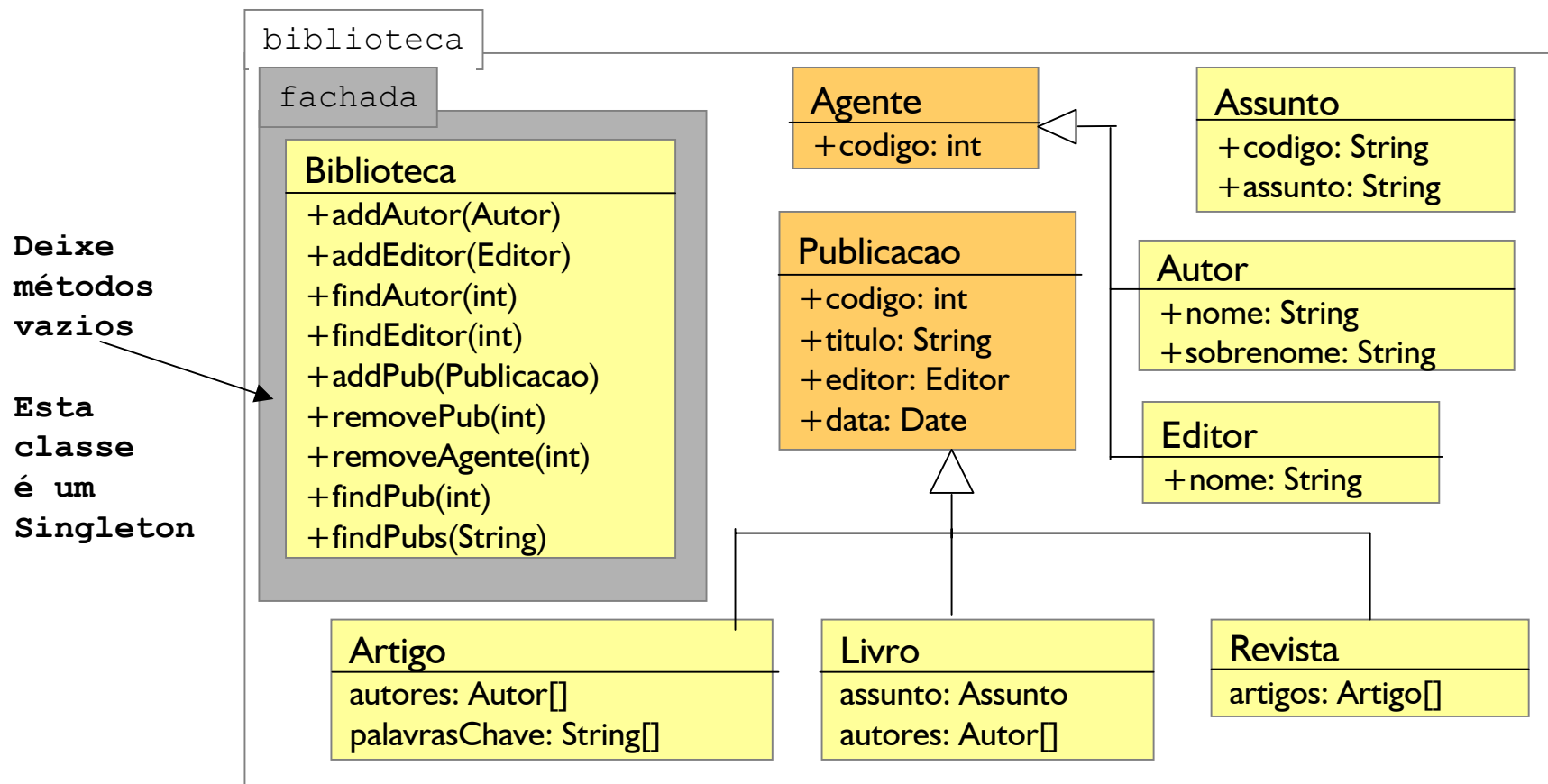
```
public class Highlander {
    private Highlander() {}
    private static Highlander instancia;
    public static Highlander criarInstancia() {
        if (instancia == null)
            instancia = new Highlander();
        }
        return instancia;
    }
}
```

Esta classe implementa o padrão de projeto Singleton

```
public class Fabrica {
    public static void main(String[] args) {
        Highlander h1, h2, h3;
        //h1 = new Highlander(); // nao compila!
        h2 = Highlander.criarInstancia();
        h3 = Highlander.criarInstancia();
        if (h2 == h3) {
            System.out.println("h2 e h3 são mesmo objeto!");
        }
    }
}
```

Esta classe cria apenas um objeto Highlander

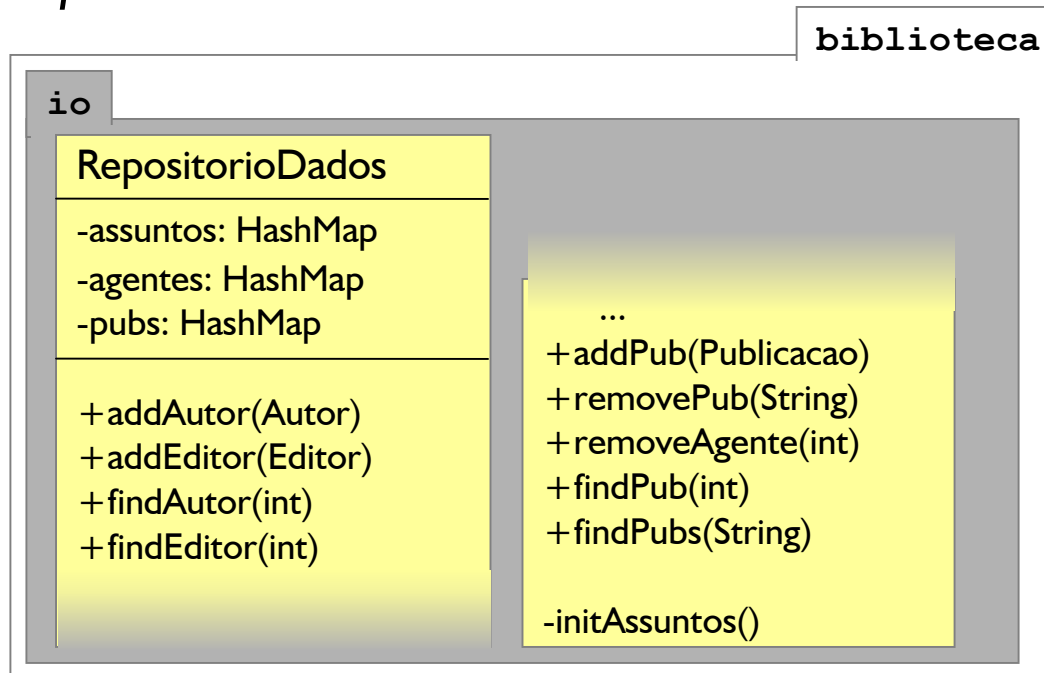
1. Implemente a seguinte hierarquia de classes



- a) Os dados representam propriedades (pares get/set) e não campos de dados (que devem ser private)
- b) Implemente equals(), toString() e hashCode() e construtores em cada classe biblioteca.*

Exercícios (2)

- 2. Implemente a classe abaixo



Inicie
previamente
os assuntos
(veja slide
seguinte)

- 3. Implemente os métodos de **Biblioteca** para que chamem os métodos de **RepositorioDados**.
- 4. Coloque tudo em um JAR.
- 5. Escreva uma classe que contenha um **main()**, importe os pacotes da biblioteca, crie uma **Biblioteca** e acrescente autores, livros, artigos, revistas, e imprima os resultados.

Apêndice: RepositorioDados (trecho)

```
private java.util.HashMap assuntos =
    new java.util.HashMap();

public void initAssuntos() {
    assuntos = new java.util.HashMap(10);
    assuntos.put("000", "Generalidades");
    assuntos.put("100", "Filosofia");
    assuntos.put("200", "Religião");
    assuntos.put("300", "Ciências Sociais");
    assuntos.put("400", "Línguas");
    assuntos.put("500", "Ciências Naturais");
    assuntos.put("600", "Ciências Aplicadas");
    assuntos.put("700", "Artes");
    assuntos.put("800", "Literatura");
    assuntos.put("900", "História");
}
```

Curso J100: Java 2 Standard Edition

Revisão 17.0

© 1996-2003, Helder da Rocha
(helder@acm.org)

 argonavis.com.br