Herança

Tiago Eugenio de Melo
tiago@comunidadesol.org

Herança

- Conceito:
  - Herança é a capacidade de especializar tipos de objetos (classes), de forma que os tipos especializados contenham, além de características estruturais e comportamentais já definidas pelos seus “ancestrais”, outras definidas para eles próprios.
- Forma:

tipo especializado —*é um tipo de*→ tipo ancestral

Herança

- A especialização pode ser feita tanto a partir de classes já construídas pelo próprio programador, como por classes de terceiros ou classes-padrão da linguagem Java.

Composição

- Conceito:
  - É a técnica de construir um tipo não pela derivação partindo de outra classe, mas pela junção de vários outros objetos de menor complexidade que fornecem ao objeto composto determinada funcionalidade quando em conjunto.
- Forma:

tipo composto —*é formado por*→ tipo fundamental ou tipo composto

Composição

- Não há nenhuma palavra-chave ou recurso especial para utilizar composição em Java, visto que esta técnica nada mais é do que um modo particular, para cada situação, de agrupar classes existentes de forma a criar novas classes com novas funcionalidades em determinado arranjo.

Composição

- Exemplo de composição

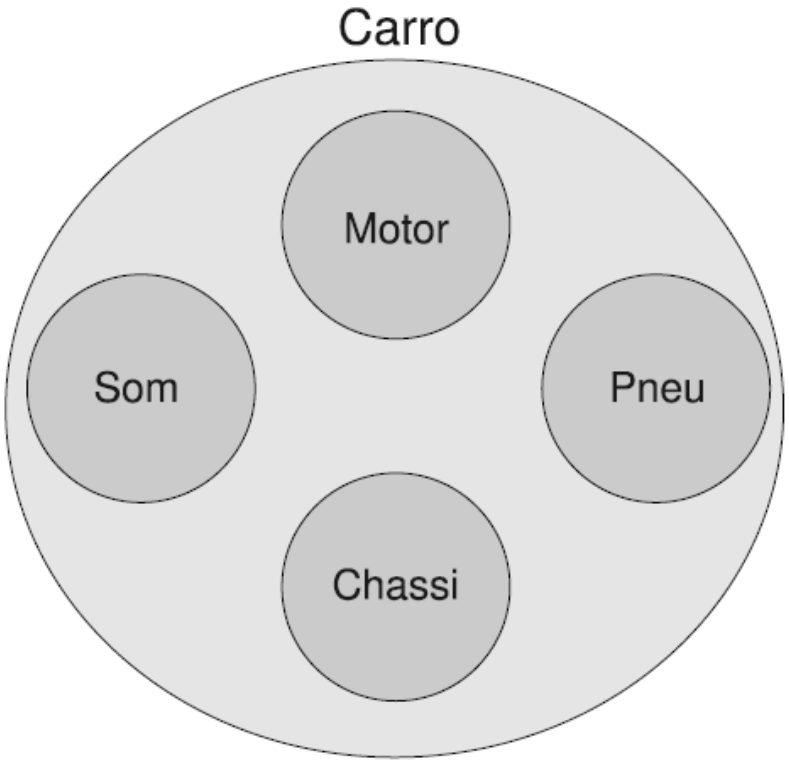

Figura 4.1: Composição entre diferentes níveis

Carro $\xrightarrow{\textit{é composto por}}$ {Motor, Som, Pneu, Chassi}

Composição

- Exemplo de composição em Java

Programa 4.1 Composição entre classes

```
01. // Composição entre classes
02.
03. class Carro {
04.     private String modelo, fabricante;
05.     private Chassi chassi;
06.     private Motor motor;
07.     private Pneu[] pneus;
08. }
09. class Chassi {
10.     private String numero, dataFabricacao;
11. }
12.
13. class Motor {
14.     private int potencia;
15.     private String numero;
16. }
17.
18. class Pneu {
19.     private int raio;
20.     private String fabricante;
21. }
22. class Som {
23.     private int potencia;
24.     private String marca;
25. }
```

Herança em Java

- Em Java, a palavra-chave utilizada para criar uma subclasse é `extends`.
- A sintaxe para uso de extends é a seguinte:

```
[Subclasse a ser criada] extends [Superclasse existente]
```

Herança em Java

- Uma característica da linguagem Java é que qualquer objeto que não herde explicitamente de outra classe, automaticamente será subclasse direta de `java.lang.Object`, que é a superclasse no nível mais alto da linguagem.

Herança em Java

- Exemplo:

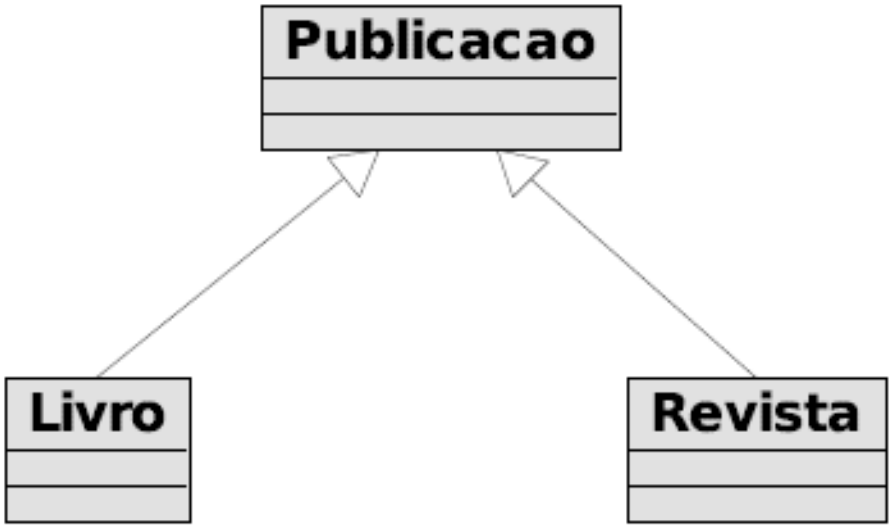

Figura 4.3: Relação entre a superclasse *Publicacao* e suas subclasses.

Herança em Java

- Implementação:

Programa 4.2 Diferentes classes herdam de Publicacao.

```
01. class Publicacao {
02.     private String nome, dataPublicacao, editora;
03. }
04.
05. class Livro extends Publicacao {
06.     private String ISBN;
07. }
08.
09. class Revista extends Publicacao {
10.     private String periodicidade;
11. }
```

Herança em Java

- Palavra-chave `protected`
  - A linguagem Java permite que certos dados sejam encapsulados de forma a estarem disponíveis apenas entre classes de uma mesma hierarquia, estando protegidos de acesso público.
  - `protected` é a palavra-chave que permite atribuir tal característica de encapsulamento a atributos e métodos de uma classe.

Herança em Java

- Observação: Vale relembrar a semântica relacionada aos símbolos utilizados para representar o nível de encapsulamento de um atributo ou método, de acordo com a terminologia da linguagem UML[RBJ06].

–	privado
#	protegido
+	público

Herança em Java

- Exemplo:

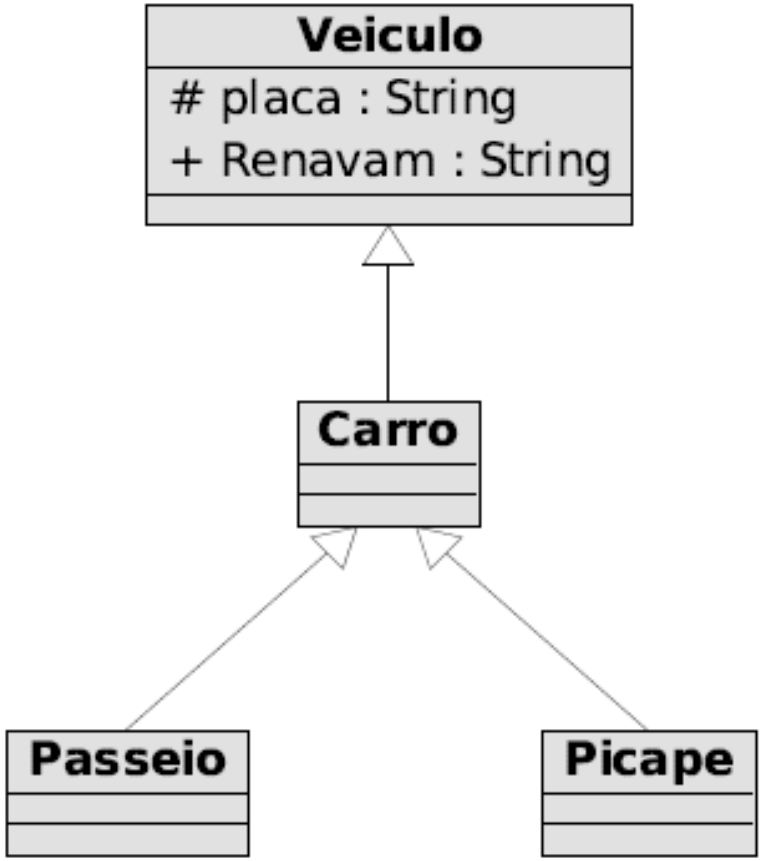

Figura 4.4: Atributos privado e protegido.

Herança em Java

- Construtores em subclasses
  - A classe Aviao tem o seu construtor.

Programa 4.6 Código do Aviao.java

```
01. // Classe representando um Avião
02. public class Aviao {
03.     protected int cor;
04.     protected double coordenadaX, coordenadaY;
05.     public Aviao() {  // Construtor
06.         // Número randômico entre 0 e 255
07.         cor = (int) (255 * Math.random());
08.         // Números randômicos entre 0 e 65536
09.         coordenadaX = 65536 * Math.random();
10.         coordenadaY = 65536 * Math.random();
11.         System.out.println("Inicializado com Cor = " + cor +
12.            " e (x,y) = (" + coordenadaX + "," + coordenadaY + ")")
13.   }
14. } // final da classe Aviao
```

Herança em Java

- Construtores em subclasses.
  - A subclasse AviaoRadar tem o seu próprio construtor.

Programa 4.7 Código do AviaoRadar.java

```
01. // Especialização de Avião com Radar
02. public class AviaoRadar extends Aviao {
03.     private int raioRadar;
04.     public AviaoRadar() {   // Construtor
05.         // Número randômico entre 0 e 10000
06.         raioRadar = (int) (10000 * Math.random());
07.         System.out.println("Inicializado com Raio do Radar = " + raioRadar)
08.     }
09. } // final da classe AviaoRadar
```

Herança em Java

- Construtores em subclasses
  - A subclasse AviaoRadar define um construtor próprio, com o intuito de inicializar os seus próprios atributos.
  - Quando há herança entre classes, ao se criar um objeto da classe derivada, este contém um sub-objeto de sua superclasse.
  - Isto significa que este sub-objeto é equivalente a criar um objeto puro desta superclasse.
  - AviaoRadar contém um sub-objeto de Aviao.

Herança em Java

- Construtores em subclasses
  - A sequência de execução dos construtores sempre acontece no sentido *top-down*, ou seja, desde a superclasse, passando por todos os níveis da hierarquia, até chegar à subclasse em questão, como pode ser visto na Figura 4.5.

Herança em Java

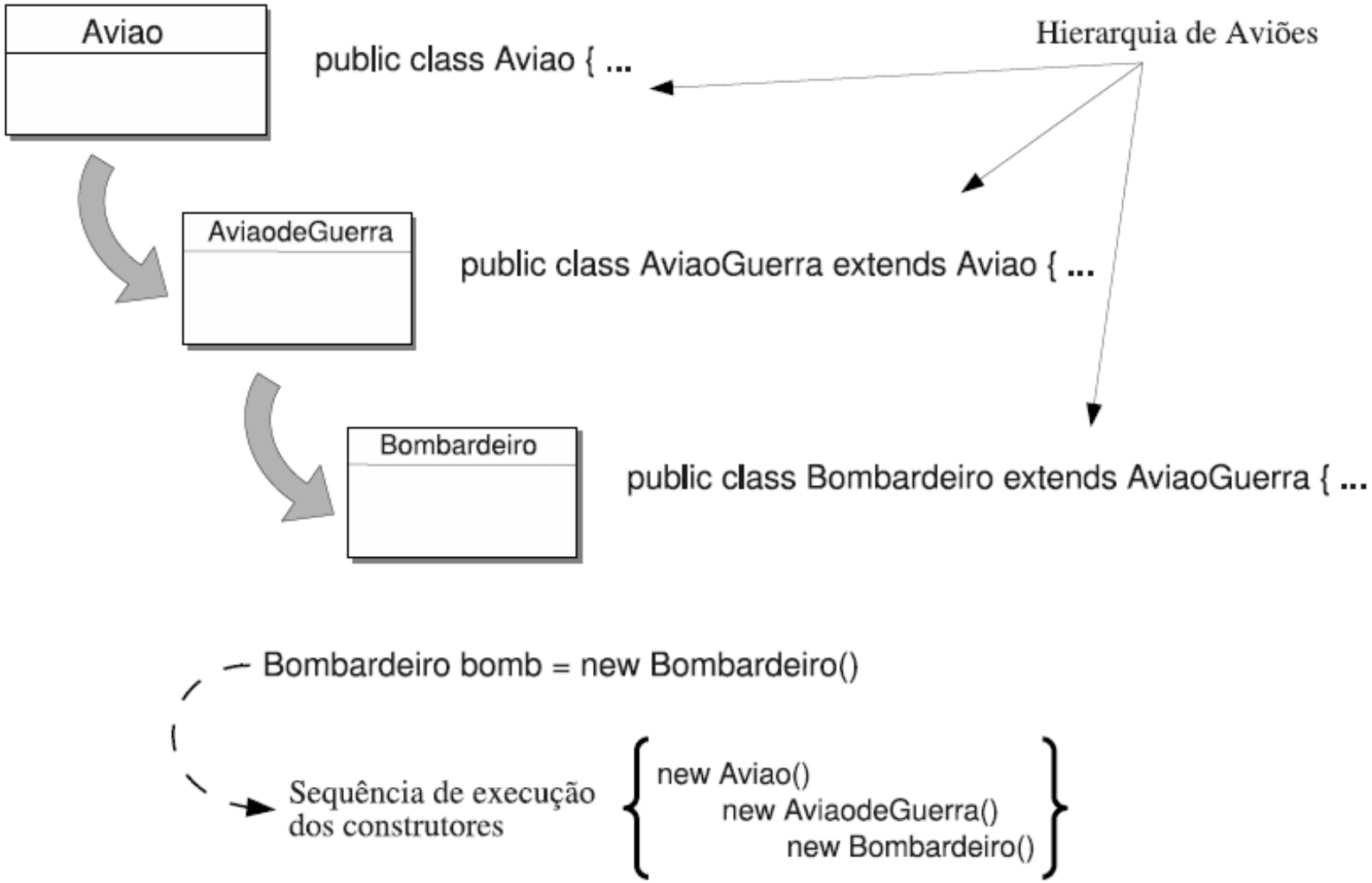

Figura 4.5: Sequência de chamada de construtores em uma hierarquia

Herança em Java

- Construtores parametrizados
  - A criação de construtores com nomes iguais, mas com diferentes argumentos, equivale à sobrecarga de métodos convencionais.
  - Na prática, projetar classes com construtores sobrecarregados permite ao programador inicializar objetos de diferentes formas, com diferentes parâmetros e opções.
  - Em se tratando de herança em Java, existe uma palavra-chave denominada `super` que permite às subclasses referenciar suas superclasses.

Herança em Java

- Construtores parametrizados
  - A palavra-chave super permite acessar a superclasse através de duas formas:
    - Um objeto implícito que representa a própria superclasse.
    - Um método que representa uma chamada aos construtores da superclasse.

Herança em Java

- Construtores parametrizados
  - No funcionamento do mecanismo de herança com um construtor padrão, os construtores da superclasse e subclasses não possuem argumentos, o compilador descobre facilmente que deve chamar implicitamente o construtor da superclasse.
  - Por outro lado, quando há construtores parametrizados, o compilador não faz este trabalho implícito, impondo ao programador que realize uma chamada ao construtor correto através da palavra-chave `super`.

Herança em Java

- A linha 15 do Programa 4.9 é desnecessária.

Programa 4.9 Classes de conta bancária - chamada desnecessária ao construtor da superclasse

```
01. // Conta bancária normal
02. class ContaBancaria {
03.     private double saldo;
04.     private String titular;
05.     public ContaBancaria() {
06.         // ... Inicialização da conta ...
07.     }
08. }
09.
10. // Conta bancária de clientes especiais
11. class ContaVIP extends ContaBancaria {
12.     private double chequeEspecial;
13.     private int diasSemJuros;
14.     public ContaVIP() {
15.        super()  // Chamada ao construtor da superclasse ContaBancaria
16.        // ... Inicialização da conta VIP ...
17.    }
18. }
```

Herança em Java

- Construtores parametrizados
  - Considere agora uma pequena modificação aplicada de forma que o construtor da classe ContaBancaria necessite de um parâmetro na inicialização do construtor, conforme o Programa 4.10.

Herança em Java

Programa 4.10 Classes de conta bancária - chamada ao novo construtor

```
01. // Conta bancária normal
02. class ContaBancaria {
03.     private double saldo;
04.     private String titular;
05.     public ContaBancaria(double saldoInicial) {  // novo construtor
06.         saldo = saldoInicial;
07.         // ... Inicialização da conta ...
08.     }
09. }
10.
11. // Conta bancária de clientes especiais
12. class ContaVIP extends ContaBancaria {
13.     private double chequeEspecial;
14.     private int diasSemJuros;
15.     public ContaVIP(double saldoInicial) {
16.         super(saldoInicial)  // Chamada ao novo construtor da superclasse
17.         // ... Inicialização da conta VIP ...
18.     }
19. }
```

Herança em Java

- Neste caso do Programa 4.10, a chamada ao construtor da superclasse através da palavra-chave `super` é necessária, caso contrário não seria possível inicializar a subclasse da forma definida na superclasse, e haveria um erro em tempo de compilação.

Herança em Java

- Sobrescrita de métodos
  - Na herança de classes, existe a possibilidade de se redefinir métodos nas subclasses, mesmo quando estes estão implementados nas superclasses.
  - Isto se chama sobrescrita de métodos (*overriding).*
  - A utilidade se dá no fato do programador poder criar um comportamento próprio a um método após sobrescrevê-lo, quando o respectivo método da superclasse não é suficiente.

Herança em Java

- Sobrescrita de métodos

Programa 4.11 Código de Conexao.java

```
01. // Conexão convencional
02. public class Conexao {
03.     protected String endereco;
04.     public Conexao(String enderecoRemoto) {  // Construtor
05.         endereco = enderecoRemoto;
06.     }
07.     public void transmitir(byte[] dados) {
08.         // ... Código para transmissão convencional ...
09.     }
10.     public byte[] receber(int quantidade) {
11.         // ... Código para recepção convencional ...
12.     }
13. }
```

Herança em Java

- Sobrescrita de métodos

Programa 4.12 Código de ConexaoSegura.java

```
01. // Conexão segura
02. public class ConexaoSegura extends Conexao {
03.     private String algoritmo;
04.     public Conexao(String enderecoRemoto, String algoritmo) {  // Construtor
05.         super(enderecoRemoto); // Chamada ao construtor da superclasse
06.         this.algoritmo = algoritmo;
07.     }
08.     // Redefinição dos método transmitir e receber para incluir criptografia
09.     public void transmitir(byte[] dados) {
10.         // ... Transmissão com criptografia ...
11.     }
12.     public byte[] receber(int quantidade) {
13.         // ... Recepção com criptografia ...
14.    }
15. }
```

Interfaces

- Java possui o recurso de definição de classes abstratas.
- Uma classe abstrata serve como base para uma hierarquia de classes derivadas, podendo implementar parte do código comum a todas elas.
- Ao mesmo tempo que pode implementar parte do código, uma classe abstrata não pode ser instanciada.
- A instanciação deve ser feita sempre em uma classe concreta derivada.

Interfaces

- Java possui, além das classes abstratas, um outro recurso relacionado denominado de interface.
- Uma interface é um contrato onde a classe que a implementa assume uma espécie de compromisso em implementar os métodos no formato que a interface define.
- Como define [HC00], uma interface é uma maneira de descrever **o que** a classe vai fazer, ao invés de **como** ela faria.

Interfaces

- Ao contrário das classes abstratas, as interfaces não podem ter uma implementação própria.
- Isso significa que elas não possuem atributos nem implementação de métodos, apenas as assinaturas dos mesmos.

Interfaces

- Uma característica importante das interfaces é que uma classe Java é capaz de implementar quantas interfaces forem necessárias, enquanto que ao utilizar classes abstratas, o programador está restrito ao fato de herdar de apenas uma classe ancestral.
- O formato para uso de interface é descrito a seguir:

```
[Classe implementadora] implements [Interface 1], [Interface 2], ... , [Interface N]
```

Interfaces

- Exemplo:

Programa 4.13 Código da interface e da implementação de uma pilha

```
01. // Interface Pilha
02. interface Pilha {
03.     // Constante estática - máximo de elementos
04.     // Único tipo de membro que uma interface pode ter
05.     public final static int MAX_ELEM = 100;
06.     // Empilhar e desempilhar - Apenas a assinatura dos métodos
07.     public void empilhar(Object dado);
08.     public Object desempilhar();
09. }
10.
11. // Implementação concreta da interface Pilha
12. class PilhaImplementacao implements Pilha {
13.       // Pilha capaz de conter 100 objetos
14.     Object[] dadosPilha = new Object[Pilha.MAX_ELEM];
15.     int ULTIMO = -1; // Índice do último elemento da pilha
16.     // A classe deve implementar os dois métodos definidos
17.     // Atenção: os códigos não estão fazendo checagem de erros
18.     public void empilhar(Object dado) {
19.         ULTIMO++;
20.         dadosPilha[ULTIMO] = dado;
21.     }
22.     public Object desempilhar() {
23.         Object obj = dadosPilha[ULTIMO];
24.         dadosPilha[ULTIMO] = null;
25.         ULTIMO--;
26.         return obj;
27.     }
28. }
```

Herança Múltipla

- A herança múltipla é o fato de uma classe possuir mais de uma superclasse.
- Em outros termos, significa construir uma classe que herde as características de mais de uma classe ao mesmo tempo.
- Linguagens de programação como C++ permitem tal recurso, embora Java não o permita diretamente.
- Java permite apenas que haja herança múltipla de várias interfaces.

Herança Múltipla

- Exemplo

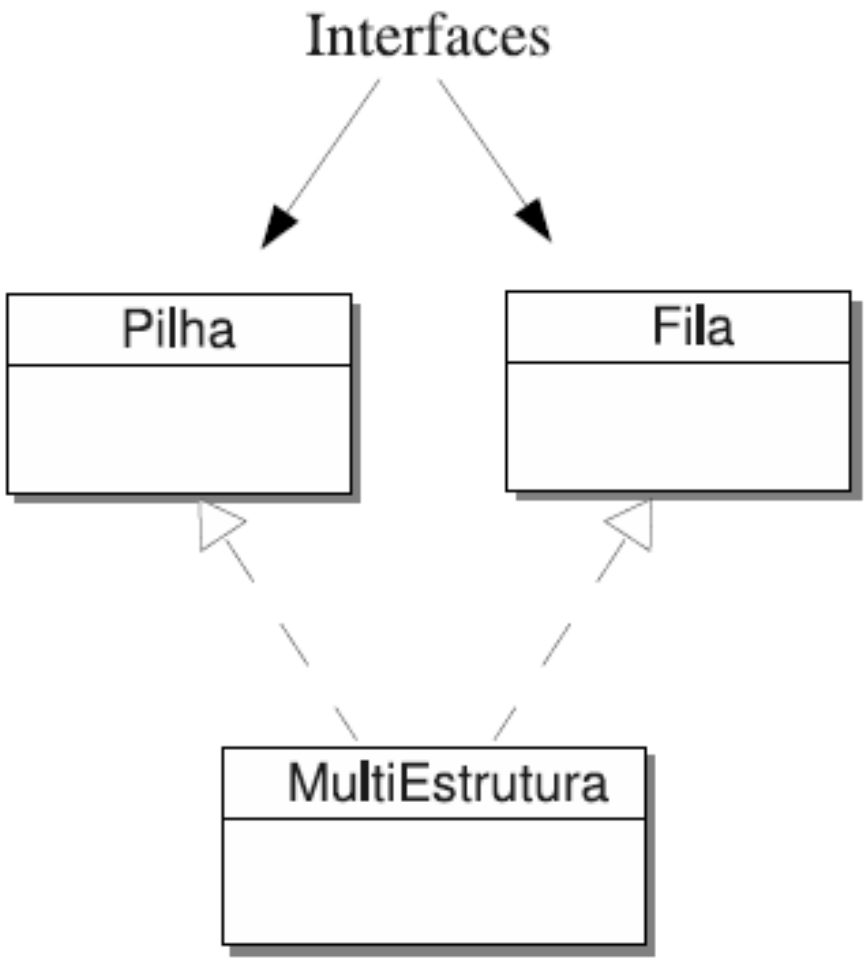

Figura 4.6: Implementação de diferentes interfaces

Herança Múltipla

- Exemplo

Programa 4.14 Código da implementação de pilha e fila ao mesmo tempo

```
01. // Interface Fila
02. interface Fila {
03.     public void enfileirar(Object dado);
04.     public Object desenfileirar();
05. }
06.
07. // Implementação concreta das interfaces Pilha e Fila
08. // Todos os métodos de ambas as interfaces são implementados
09. public class MultiEstrutura implements Pilha, Fila {
10.     public void empilhar(Object dado) {
11.           // Código para empilhar
12.     }
13.     public Object desempilhar() {
14.           // Código para desempilhar
15.     }
16.     public void enfileirar(Object dado) {
17.           // Código para enfileirar
18.     }
19.     public Object desenfileirar() {
20.           // Código para desenfileirar
21.     }
22. }
```

Herança entre Interfaces

- Além da própria definição e uso de interfaces, Java também permite realizar herança entre interfaces.
- O conceito de herdar uma interface pode parecer estranho à primeira vista, já que as interfaces não possuem implementação.
- Entretanto, justamente por isso, é um artifício de simples compreensão.
- Uma interface, ao herdar de outra, automaticamente assume os métodos desta de forma implícita.

Herança entre Interfaces

- Exemplo

Programa 4.16 Herança de interfaces e implementação correspondente

```
01. // Interface Usuario
02. interface Janela {
03.     public int getLargura();
04.     public int getAltura();
05. }
06.
07. // Interface JanelaMensagem - herança de Janela
08. interface JanelaMensagem extends Janela {
09.     public String getMensagem();
10. }
11.
12. // A classe JanelaConfirmacao implementa apenas JanelaMensagem,
13. // mas também tem que implementar os métodos de Janela
14. // por causa da herança das interfaces
15. public class JanelaConfirmacao implements JanelaMensagem {
16.     public int getLargura() { /* Largura */ }
17.     public int getAltura() { /* Altura */ }
18.     public String getMensagem() {
19.         return "Por favor confirme!";
20.     }
21. }
```

Classes Anônimas

- A linguagem Java permite criar determinadas classes apenas para servir a um contexto muito específico, como argumento de um método, por exemplo.
- Para este tipo de situação, Java oferece a opção de se criar as chamadas classes anônimas.
- Uma classe anônima, como o próprio nome diz, não tem um nome atribuído.

Classes Anônimas

- A classe anônima é criada necessariamente como subclasse de uma outra classe ou como implementação de uma interface, e tem a função de suprir uma necessidade pontual.
- Um exemplo corriqueiro e bastante utilizado de classes anônimas é a programação com janelas visuais em Java

DICA: As classes anônimas não devem ser usadas indiscriminadamente em programas Java, pois acabam tornando os programas mais difíceis de interpretar por programadores diferentes. Desta forma, apenas em casos especiais como a programação em *Swing* elas devem ser utilizadas.

Classes Anônimas

- Exemplo

Programa 4.17 Código de Janela.java

```
01. // Janela visual com botão clicável
02. public class Janela extends JFrame {
03.     private JButton botao = new JButton();
04.     public Janela() {
05.         // Uma classe anônima é criada na passagem de parâmetros do botão
06.        // Perceba que a declaração [ new ActionListener() { ]
07.         // define uma classe sem nome!
08.         botao.addActionListener(new ActionListener() {
09.              // Código para fazer uma ação no clique do botão
10.         }); // Final da classe e da chamada ao método addActionListener
11.     }
12. }
```

Evitando a Herança

- A programação em Java envolve algumas situações peculiares onde além de não fazer uso de herança, o programador também necessita evitá-la por motivos diversos, como por exemplo para garantir que determinadas características da classe base não sejam modificadas por quaisquer subclasses geradas.

Evitando a Herança

- A palavra-chave `final` possui basicamente três tipos de utilização na linguagem Java:
  1. Criação de atributos constantes em classes, de forma que ao ser declarado como final, o atributo tem de receber um valor no ato da declaração, que não vai poder ser modificado ao longo do programa (esta é a forma que a linguagem permite definir constantes).
  2. Marcação de métodos de tal forma que eles que não possam ser sobrescritos em subclasses, ou seja, o comportamento do método permanece o mesmo em qualquer subclasse derivada da classe que o definiu.

Evitando a Herança

- A palavra-chave final possui basicamente três tipos de utilização na linguagem Java:

  3. Marcação de classes para que as mesmas não possam servir como base de uma herança. Isso significa que uma classe final não pode ser superclasse de nenhuma outra.

Evitando a Herança

- Exemplo

Programa 4.18 Classe com método bloqueado para herança

```
01. // Uma classe base capaz de gerar senhas para um sistema a partir
02. // de um dado fornecido de forma a ser definida em subclasses.
03. public abstract class GeradorDeSenhas {
04.     // Um determinado dado a ser lido (arquivo, banco de dados, rede, etc.)
05.     protected Object dado;
06.     // Método a ser implementado em subclasse que define
07.     // como o dado é carregado
08.     public abstract void carregarDado();
09.     // O código para gerar a senha é proprietário e deve ser
10.     // mantido fixo independente das implementações
11.    public final String getSenha() {
12.        // Código para gerar a senha não pode ser sobreescrito pelas subclasses
13.    }
14. }
```

Evitando a Herança

- Exemplo

Programa 4.19 Classe bloqueada para herança

```
01. // Classe de configuração de um sistema
02. public final class Configurador {  // Esta classe não pode ser derivada
03.     public void inicializarSistema() {
04.          // Código para inicializar os módulos do sistema
05.     }
06.     public void finalizarSistema() {
07.          // Código para finalizar o sistema
08.     }
09.     public void configurar(int modulo, String chave, Object valor) {
10.         // Configura uma determinada variável do sistema
11.     }
12.     public int obterStatus() {
13.         // Obtém o status do sistema
14.     }
15. }
```

Evitando a Herança

- Dica:
  - Mesmo quando uma classe é declarada como final, seus atributos continuam mutáveis, a menos que declarados explicita e individualmente como finais também.

Composição versus Herança

- Existem situações que levam o programador a ter de escolher entre projetar uma classe baseada em composição ou herança.
- Na maior parte das vezes, a solução mais comum é agrupar classes existentes em novas funcionalidades para criar novas classes, ou seja, utilizar composição.
- Em outras situações, uma análise levará à percepção de que o uso de herança será necessário.

Composição versus Herança

- O programador deve ter em mente que a utilização de herança não deve ser excessiva, mas que possa atingir a demanda correta que o seu projeto exige.

Composição versus Herança

- A utilização de herança ou composição é se perguntar se a classe a ser criada nunca necessitará de um *upcast* para a suposta superclasse, ou seja, se ela nunca precisará “assumir” o tipo da superclasse em alguma situação.
- Lembre-se que quando uma classe herda de outra, ela também é daquele tipo.

Composição versus Herança

- O *upcast* é justamente isso: em qualquer momento, um objeto da subclasse pode ser utilizado como se fosse um objeto da superclasse.
- O programador, desta forma, deve concluir que se a classe a ser criada nunca precisa assumir o referido tipo da superclasse, provavelmente esta situação será melhor modelada se esta superclasse for apenas um atributo, caracterizando assim uma situação de uso de composição.

Composição versus Herança

- Dica:
  - Classes ou métodos abstratos não podem ser definidos como final, já que, obrigatoriamente, são projetados para subclasses os implementarem.
  - Isso significa que as palavras-chave `abstract` e `final` são mutuamente excludentes.

Atividades

- Explique de que maneira o uso da herança promove a reutilização de código.
- Qual a utilidade de se definir métodos e atributos com o modificador de acesso `protected`?
- Qual as principais diferenças entre herdar interfaces e herdar classes?
- Explique, com suas palavras, as principais diferenças entre composição e herança.
- Qual a principal diferença entre uma classe final e uma classe convencional?

Atividades

- Qual é a diferença entre herança simples e herança múltipla? Como Java trata a herança múltipla?

Atividades

- Considere o Programa 4.20. Que mudanças seriam aplicadas à classe Usuario para que a classe Administrador pudesse implementar um método mostrarDados() de forma a imprimir a senha e o password ? Realize também a implementação de um método main() para a classe Administrador que a faça a chamada ao método mostrarDados().

Atividades

Programa 4.20 Exercício.

```
01. class Usuario {
02.     private String password;
03.     private String login;
04. }
05. class Administrador extends Usuario {
06. }
```

Atividades

- Não é possível compilar o código o Programa 4.23. Qual é a modificação que fará este código compilar corretamente?

Atividades

Programa 4.23 Exercício.

```
01. final class Aaa {
02.     int xxx;
03.
04.     void yyy() {
05.         xxx = 1;
06.     }
07. }
08.
09. class Bbb extends Aaa {
10.     final Aaa finalref = new Aaa();
11.
12.     final void yyy() {
13.         System.out.println("No método yyy()");
14.         finalref.xxx = 12345;
15.      }
16. }
```