nemo

ontology & conceptual modeling research group

# Desenvolvimento OO com Java

# 5 – Reuso de Classes


Vítor E. Silva Souza

(vitorsouza@inf.ufes.br)

http://www.inf.ufes.br/~vitorsouza



Departamento de Informática

Centro Tecnológico

Universidade Federal do Espírito Santo


# Objetivos da parte 5

- Apresentar os conceitos de reutilização de código em programação Java:
  - Composição e herança;
  - Sobrescrita de métodos;
  - A classe `java.lang.Object`;
  - O especificador `protected`;
  - A palavra-chave `final`.
- Desta forma:
  - Aumentar a capacidade de desenvolvimento de código para e com reuso, tornando-se mais produtivo.

# Reuso / reutilização

- Para entregar software de qualidade em menos tempo, é preciso reutilizar;
  - "Copy & paste" não é reuso!
- Reuso é uma das principais vantagens anunciadas pela Orientação a Objetos;
- Mecanismo baseado no conceito de classes:
  - Composição ("tem um");
  - Herança ou derivação ("é um").

# Composição

- Criação de uma nova classe usando classes existentes como atributos;
- Relacionamento "tem um": uma conta tem um dono (cliente), um cliente tem um nome (*string*);
- Vimos como fazer isso no capítulo passado:
  - Atributos primitivos e referências a objetos;
  - Operadores de seleção;
  - Inicialização (zerar e atribuir valor inicial);
  - O valor `null`;
  - Atributos estáticos.

# Composição

```
class Aleatorio {
  int numero;
  Aleatorio(int max) {
     numero = new Random().nextInt(max);
  }
}
public class NumeroAleatorio {
  private String nome;
  private Aleatorio valor;
  NumeroAleatorio(String nome, int valor) {
     this.nome = nome;
     this.valor = new Aleatorio(valor);
  }
  public static void main(String[] args) {
     NumeroAleatorio n;
     n = new NumeroAleatorio("Número secreto", 50);
  }
}
```

# Herança

- Criação de novas classes derivando classes existentes;
- Relacionamento "é um [subtipo de]": um livro é um produto, um administrador é um usuário;
- Uso da palavra-chave `extends`;
- A palavra-chave é sugestiva – a classe que está sendo criada "estende" outra classe:
  - Partindo do que já existe naquela classe...
  - Pode adicionar novos recursos;
  - Pode redefinir recursos existentes.

# Sintaxe

- Sintaxe:

```
class Subclasse extends Superclasse {
}
```

- Semântica:
  - A subclasse herda todos os atributos e métodos que a superclasse possuir;
  - Subclasse é uma derivação, um subtipo, uma extensão da superclasse.

# Herança

```
public class Produto {
  protected String nome;
  protected double preco;

  public Produto(String nome, double preco) {
     this.nome = nome;
     this.preco = preco;
  }

  public boolean ehCaro() {
     return (preco > 1000);
  }

  /* Métodos de acesso ... */
}
```

# Herança

```
public class Livro extends Produto {
  private String autor;
  private int paginas;

  public Livro(String nome, double preco,
               String autor, int paginas) {
     super(nome, preco);
     this.autor = autor;
     this.paginas = paginas;
  }

  public boolean ehGrande() {
     return (paginas > 200);
  }
}
```

# Herança

```
public class Loja {
  public static void main(String[] args) {
     Livro l;
     l = new Livro("Linguagem de Programação",
          74.90, "Flávio Varejão", 334);

     System.out.println(l.ehCaro());
     System.out.println(l.ehGrande());
  }
}
```

# Subclasses herdam membros

- Livro possui `autor` e `paginas` (definidos na própria classe);
- Livro possui `nome` e `preco` (definidos na superclasse);
- Livro pode receber mensagens `ehGrande()` (definida na própria classe);
- Livro pode receber mensagens `ehCaro()` (definida na superclasse).

# Sinônimos

- Superclasse;
- Classe base;
- Classe pai/mãe;
- Classe ancestral;
- Etc.

- Subclasse;
- Classe derivada;
- Classe filha;
- Classe descendente;
- Etc.

# Sobrescrita de métodos

- Se um método herdado não satisfaz, podemos redefini-lo (sobrescrevê-lo):

```
public class Livro extends Produto {

  /* Definições anteriores... */

  // Livros acima de R$ 200 são caros!
  public boolean ehCaro() {
     return (preco > 200);
  }

}
```

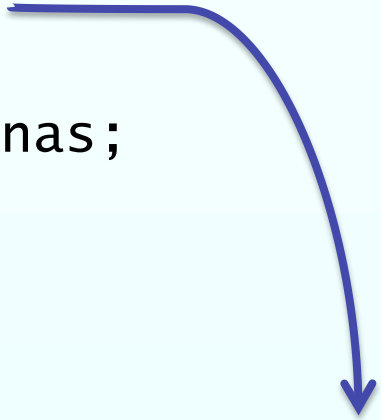
# Sobrescrita de métodos

- Métodos sobrescritos podem chamar sua versão na superclasse usando a palavra super:

```
public class Produto {
  /* ... */
  public void imprimir() {
    System.out.println(nome + "," + preco);
  }
}
public class Livro extends Produto {
  /* ... */
  public void imprimir() {
    super.imprimir();
    System.out.println(autor + "," + paginas);
  }
}
```

# Chamando o construtor da superclasse

```
public class Livro extends Produto {
  /* ... */
  public Livro(String nome, double preco,
               String autor, int paginas) {
    super(nome, preco);
    this.autor = autor;
    this.paginas = paginas;
  }
}
public class Produto {
  /* ... */
  public Produto(String nome, double preco) {
    this.nome = nome;
    this.preco = preco;
  }
}
```



# A classe Object

- Em Java, todos os objetos participam de uma mesma hierarquia, com uma raiz única;
- Esta raiz é a classe `java.lang.Object`.

```
class Produto { }

/* É equivalente a: */

class Produto extends Object { }
```

# A classe Object

- Possui alguns métodos úteis:
  - `clone()`: cria uma cópia do objeto (uso avançado);
  - `equals(Object o)`: verifica se objetos são iguais;
  - `finalize()`: chamado pelo GC (não é garantia);
  - `getClass()`: retorna a classe do objeto;
  - `hashCode()`: função hash;
  - `notify()`, `notifyAll()` e `wait()`: para uso com threads;
  - `toString()`: converte o objeto para uma representação como `String`.

# O método toString()

- `toString()` é chamado sempre que:
  - Tentamos imprimir um objeto;
  - Tentamos concatená-lo com uma *string*.

```
public class Loja {
  public static void main(String[] args) {
     Produto p = new Produto("CD", 30.0);
     System.out.println(p);
  }
}

// Resultado (toString() herdado de Object):
// Produto@10b62c9
```

# O método toString()

```
class Produto {
  /* ... */
  public String toString() {
     return nome + " (R$ " + preco + ")";
  }
}
public class Loja {
  public static void main(String[] args) {
     Produto p = new Produto("CD", 30.0);
     System.out.println(p);
  }
}

// Resultado (toString() sobrescrito):
// CD (R$ 30.0)
```

# Inicialização da classe base

- Se um `Livro` é um `Produto`, para criarmos um livro precisamos antes criar um produto.

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

class Notebook extends Computador {
  public Notebook() {
     System.out.println("Notebook()");
  }
}
```

# Inicialização da classe base

- O construtor base é chamado automaticamente;
- Chamada implícita à `super();`
- O construtor *default* também faz isso.

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

// Resultado:
// Computador()
// Notebook()
```

# Atenção à ordem de construção

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

class Notebook extends Computador {
  private int codigo;
  public Notebook() {
     System.out.println("Notebook()");
     codigo = 12345;
  }
  public void ligar() {
     System.out.println("Código " + codigo);
  }
}
```

# Atenção à ordem de construção

- O construtor da superclasse é chamado antes do código receber seu valor:

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

// Resultado:
// Computador()
// Código 0
// Notebook()
```

# Construtores sem parâmetros

```
class Pessoa {
  private String nome;
  public Pessoa(String nome) {
     this.nome = nome;
  }
}

class Aluno extends Pessoa { }

// Teste.java:28: cannot find symbol
// symbol  : constructor Pessoa()
// location: class Pessoa
// class Aluno extends Pessoa {
// ^
// 1 error
```

# Construtores com parâmetros

- `Aluno` não define construtor: ganha um *default*;
- `Pessoa` define um construtor com parâmetro: não ganha construtor default;
- Construtor default tenta chamar construtor sem parâmetro na superclasse (`Pessoa`);
- `Pessoa` não possui construtor sem parâmetro!

# Construtores com parâmetro

- A solução é chamar o construtor explicitamente;
- Assim como `this()`, `super()` deve ser o primeiro comando do construtor.

```
class Aluno extends Pessoa {
  public Aluno() {
     super("Sem nome");
  }
  public Aluno(String nome) {
     super(nome);
  }
}
```

# Composição vs. herança

- Use herança quando:
  - Uma classe representa um subtipo de outra classe;
  - Construção de famílias de tipos;
  - Use com cuidado!
- Use composição quando:
  - Uma classe representa algo que faz parte de outra;
  - Prefira composição à herança.
- Os dois conceitos são utilizados em conjunto a todo momento!

# Composição vs. herança

```
class Lista {
  public void adic(int pos, Object obj) { }
  public Object obter(int pos) { }
  public void remover(int pos) { }
}

// Uma pilha é uma lista?
class Pilha1 extends Lista { }

// Ou uma pilha tem uma lista?
class Pilha2 {
  private Lista elementos;
  public void empilha(Object obj) { }
  public Object desempilha() { }
}
```

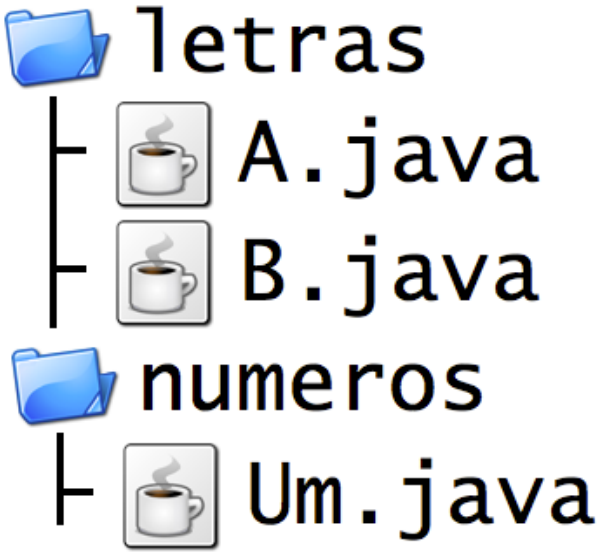
# O especificador de acesso protected

- Como já vimos, `protected` é um especificador de acesso que:
  - Permite acesso interno (à própria classe);
  - Permite acesso às classes do mesmo pacote;
  - Permite acesso às subclasses (do mesmo ou de outro pacote);
  - Não permite acesso às demais classes.

# Membros protegidos

```
class A {
  int x = 10;
  protected void print() {
     System.out.println(x);
  }
  protected void incr() {
     x++; }
}
```

```
import letras.*;
public class Um
              extends A {
  public void g() {
     incr(); // OK!
     print(); // OK!
     // Erro: x++;
  }
}
```




```
public class B {
  A a = new A();
  public void f() {
     a.x = 15; a.print();
  }
}
```

# Regra de bolso do ocultamento

- De maneira geral:
  - Atributos de uma classe devem ser privados;
  - Se a classe possui filhas, atributos podem ser protegidos ou possuir métodos de acesso protegidos;
  - Métodos que pertencem à interface devem ser públicos;
  - Alguns métodos podem ser utilizados internamente e, portanto, serem privados ou protegidos.

# Vantagens da herança

- Suportar do desenvolvimento incremental;
  - Classes já testadas podem ser reutilizadas;
  - Economia de tempo.
- Relacionamento "é um":
  - Permite substituir a classe base por uma subclasse quando a primeira é esperada;
  - Propriedade que chamamos de polimorfismo;
  - Veremos em mais detalhes no próximo capítulo.

# Polimorfismo

```
class Forma {
  public void desenhar() {
     System.out.println("Forma");
  }
}
class Circulo extends Forma {
  public void desenhar() {
     System.out.println("Círculo");
  }
}
class Quadrado extends Forma { /* ... */ }
class Triangulo extends Forma { /* ... */ }
```

# Polimorfismo

```
public class Teste {
  private static void desenha(Forma[] fs) {
     for (int i = 0; i < fs.length; i++)
        fs[i].desenhar();
  }

  public static void main(String[] args) {
     Forma[] formas = new Forma[] {
        new Circulo(), new Forma(),
        new Quadrado(), new Triangulo()
     };
     desenha(formas);
  }
}
```

# Upcasting (ampliação)

- Nome dado à conversão implícita de uma subclasse para uma superclasse:

```
class Teste {
  public static void inverter(Forma f) {
     System.out.println("Inverte " + f);
  }

  public static void main(String[] args) {
     Circulo c = new Circulo();
     inverter(c);                    // Upcasting!
     Forma f = new Quadrado();       // Upcasting!
  }
}
```

# Recapitulando...

- O que já aprendemos:
  - Podemos fazer reuso com composição ou herança;
  - Os dois conceitos são muito usados em conjunto;
  - Subclasses herdam membros da superclasse;
  - Subclasses podem sobrescrever métodos;
  - Classe `java.lang.Object` é raiz da hierarquia;
  - A subclasse chama o construtor da superclasse;
  - `protected` é um *friendly* extensível às subclasses.
- Prosseguindo...
  - É possível definir uma classe como não herdável?

# A palavra reservada final

- Significa “Isto não pode ser mudado”;
- Dependendo do contexto, o efeito é levemente diferente;
- Pode ser usada em:
  - Dados (atributos / variáveis locais);
  - Métodos;
  - Classes.
- Objetivos:
  - Eficiência;
  - Garantir propriedades de projeto.

# Dados finais

- Constantes são comuns em LPs;
  - Constantes conhecidas em tempo de compilação podem adiantar cálculos;
  - Constantes inicializadas em tempo de execução garantem que o valor não irá mudar.
- Em Java, utiliza-se a palavra `final`:

```
public static final int MAX = 1000;
private final String NOME = "Java";
final double RAD = Math.PI / 180;
```

# Referência constante

- Um primitivo constante nunca muda de valor;
- Uma referência constante nunca muda, mas o objeto pode mudar internamente:

```
public class Teste {
  public static final int MAX = 1000;
  private final Coordenada C = new Coordenada();
  public static void main(String[] args) {
     // Erro: MAX = 2000;
     // Erro: C = new Coordenada();
     C.x = 100;  // OK, se x for público!
  }
}
```

# Dados finais não inicializados

```
class Viagem { }
class DadoFinalLivre {
  final int i = 0; // Final inicializado
  final int j;     // Final não inicializado
  final Viagem p;  // Referência final não inicializada

  // Finais DEVEM ser inicializados em
  // todos os construtores e somente neles
  DadoFinalLivre () {
     j = 1;
     p = new Viagem();
  }
  DadoFinalLivre (int x) {
     j = x;
     p = new Viagem();
  }
}
```

# Argumentos finais

- Um parâmetro de um método pode ser `final`:
  - Dentro do método, funciona como constante.

```
public class Teste {
  public void soImprimir(final int i) {
     // Erro: i++;
     System.out.println(i);
  }
}
```

# Métodos finais

- Métodos finais não podem ser sobrescritos por uma subclasse;
- Chamada do método *inline* (maior eficiência).

```
class Telefone {
  public final void discar() { }
}

// Não compila: discar() é final!
class TelefoneCelular extends Telefone {
  public void discar() { }
}
```

# Métodos privados são finais

- Métodos privados não podem ser acessados;
- Portanto, são finais por natureza (as subclasses não têm acesso a ele).

```
class Telefone {
  private final void checarRede() { }
}

// OK. São dois métodos diferentes!
class TelefoneCelular extends Telefone {
  private final void checarRede() { }
}
```

# Classes finais

- Classes finais não podem ter subclasses;
- Por consequência, todos os métodos de uma classe final são automaticamente finais.

```
class Telefone { }

final class TelefoneCelular extends Telefone { }

// Erro: TelefoneCelular é final!
class TelefoneAtomico extends TelefoneCelular { }
```


**http://nemo.inf.ufes.br/**