


Laboratório de Engenharia de Software

# INF1636 – PROGRAMAÇÃO ORIENTADA A OBJETOS

Departamento de Informática – PUC-Rio

Ivan Mathias Filho  
[ivan@inf.puc-rio.br](mailto:ivan@inf.puc-rio.br)




Laboratório de Engenharia de Software

## Programa – Capítulo 4

- Tipos Enumerados
- Generalização e Herança
- Visibilidade
- Chamada de Construtores
- A Classe `Object`

© LES/PUC-Rio 2

## Programa – Capítulo 4




Laboratório de Engenharia de Software

- **Tipos Enumerados**
- Generalização e Herança
- Visibilidade
- Chamada de Construtores
- A Classe `Object`

© LES/PUC-Rio

3

## Tipos Enumerados – C++



Laboratório de Engenharia de Software

- Tipos enumerados são usados para criar novos tipos de dados, que podem assumir apenas uma gama restrita de valores não-numéricos (enumeradores);
- Os valores (enumeradores) associados a um enumerado são identificadores definidos pelo programador;
- Os enumeradores são constantes do tipo **int**.

```

typedef enum {
    amarelo,vermelho,azul,verde,preto
} Cor;

int main(void) {
    Cor c=vermelho;
    int x=c;
    printf("valor de c: %d    valor de x: %d\n",c,x);
    return 0;
}

```


**valor de c: 1 valor de x: 1**

© LES/PUC-Rio

4

Laboratório de Engenharia de Software

## Tipos Enumerados – Motivação




- Antes da edição JSE5 (JDK 1.5) a linguagem Java não oferecia suporte direto à construção de tipos enumerados;
- A construção desses tipos era simulada usando-se classes, como no exemplo a seguir:

```
public class Cor {  
    static final int amarelo=0;  
    static final int vermelho=1;  
    static final int azul=2;  
    static final int verde=3;  
    static final int preto=4;  
}  
  
int vareta=Cor.amarelo;
```

© LES/PUC-Rio 5

Laboratório de Engenharia de Software


## Simulação de Enumerados – Desvantagens



- O uso de variáveis estáticas inteiras tem pelo menos duas desvantagens:
  - A variável `vareta` continuará a ser uma variável inteira. Logo, valores fora do domínio (por exemplo, 10) poderão ser atribuídos a ela;
  - A associação entre os nomes das variáveis e os valores das constantes só está representada na definição da classe, dificultando, assim, o entendimento do código escrito.

© LES/PUC-Rio 6

## Tipos Enumerados – Características




Laboratório de Engenharia de Software

- Os tipos enumerados resolvem estes problemas da seguinte maneira:
  - Os valores dos tipos enumerados são referenciados pelos seus nomes e não por valores inteiros;
  - Apenas valores pertencentes ao domínio de um enumerado poderão ser usados nos comandos de atribuição.

© LES/PUC-Rio 7

## Tipos Enumerados – Exemplo



Laboratório de Engenharia de Software


- O tipo `Cor`, visto anteriormente, pode ser definido da seguinte maneira, utilizando o construtor de enumerados, introduzido no JDK 1.5:

```
public enum Cor {  
    amarelo, vermelho, azul, verde, preto;  
}
```
- A atribuição de um valor enumerado a uma variável do tipo `Cor` tem a seguinte forma:

```
Cor vareta=Cor.amarelo;
```

© LES/PUC-Rio 8


## Tipos Enumerados – Vantagens



- As constantes que representam os valores enumerados não são compiladas para as classes que as usam;
- Cada constante de um enumerado é deixada como uma referência simbólica, que será ligada em tempo de execução, do mesmo modo que as referências para métodos e variáveis de instância;
- Desse modo, a retirada de uma constante que é usada por outra classe qualquer será sinalizada com uma mensagem de erro pela biblioteca de tempo de execução assim que a constante retirada for usada.

© LES/PUC-Rio 9

## Tipos Enumerados – switch...case




- O comando `switch...case` foi alterado para ser usado com tipos enumerados. O exemplo a seguir mostra como isso é feito:


```
Cor vareta=Cor.amarelo;

int pontos;

switch(vareta) {
    case amarelo: pontos=5;break;
    case azul: pontos=15;break;
    case preto: pontos=50;break;
    case verde: pontos=20;break;
    case vermelho: pontos=10;
}
```


© LES/PUC-Rio 10

Laboratório de Engenharia de Software	<b>Programa – Capítulo 4</b>		
	<ul style="list-style-type: none"> <li>• Tipos Enumerados</li> <li>• <b>Generalização e Herança</b></li> <li>• Visibilidade</li> <li>• Chamada de Construtores</li> <li>• A Classe <code>Object</code></li> </ul>		
		© LES/PUC-Rio	11

Laboratório de Engenharia de Software	<b>Taxonomia (1)</b>		
	<ul style="list-style-type: none"> <li>• Muito antes do alvorecer da ciência os seres humanos já nomeavam as espécies;</li> <li>• Isso lhes permitia obter sucesso nas suas atividades de caça e coleta;</li> <li>• A <b>Taxonomia</b>, palavra de origem grega cujo significado é “estudo das classificações”, surgiu no século XVII;</li> <li>• Ela ganhou força no século seguinte, graça ao trabalho do naturalista sueco <b>Carl Linnaeus</b>, que inventou um sistema para organizar os seres vivos em grupos cada vez menores;</li> <li>• Nesse sistema, os membros de um grupo particular compartilham determinadas características.</li> </ul>		
		© LES/PUC-Rio	12

Laboratório de Engenharia de Software

Taxonomia (2)



TAXONOMIA PADRÃO

ESTRUTURA DE LINNAEUS

O Universo de Linnaeus

Carl Linnaeus desenvolveu as bases para a moderna taxonomia no século 18, ordenando todos os seres biológicos em grupos hierárquicos, partindo do nível dos reinos (como animais, plantas, fungos) e descendo até o nível das espécies individuais, cada um com um conjunto exclusivo de características observáveis.

Reino

Animalia

Organismos multicelulares móveis (em sua maioria), incapazes de sintetizar os seus próprios nutrientes (heterotróficos)

Filo

Chordata

Organismos com um eixo esquelético flexível (notocorda) e cordões nervosos

Classe

Amphibia

Terrestres semi-aquáticos sem ovos amnióticos

Ordem

Anura

Adultos sem cauda, com pele enrugada e cutícula escapular

Família

Hylidae

Rãs com adaptações para viver nas árvores

Gênero

Litoria

Rãs com pupilas horizontais (não arredondadas)

Espécie

Litoria caerulea

Rãs com glândulas paratoides (entre os cabos grandes e aberturas do ouvido bem evidentes)

Reino

Filo

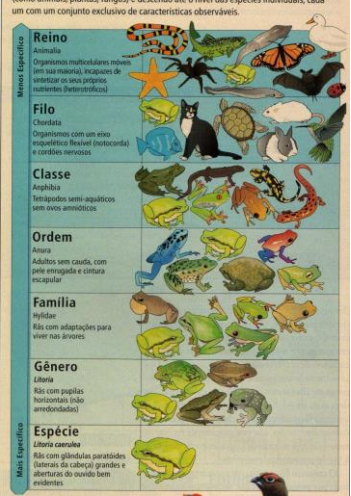
Classe

Ordem

Família

Gênero

Espécie




© LES/ PUC-Rio

13

Laboratório de Engenharia de Software

Taxonomia



- Posteriormente, a palavra **taxonomia** começou a ser usada em um sentido mais abrangente, podendo ser aplicada na classificação de quase tudo – objetos animados, inanimados, lugares e eventos.

VeiculoAereo

Aviao

Bombardeiro

Interceptador

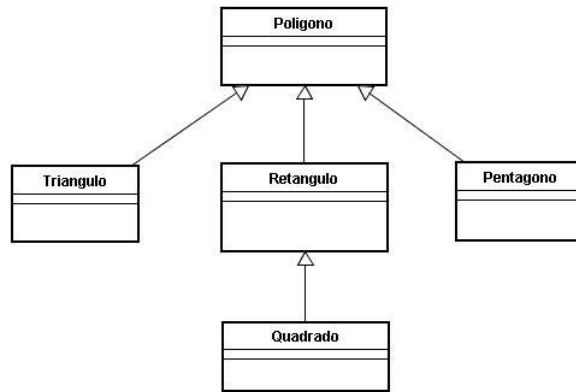
Helicoptero

© LES/ PUC-Rio

14

7

## Exemplo – Taxonomia de Polígonos



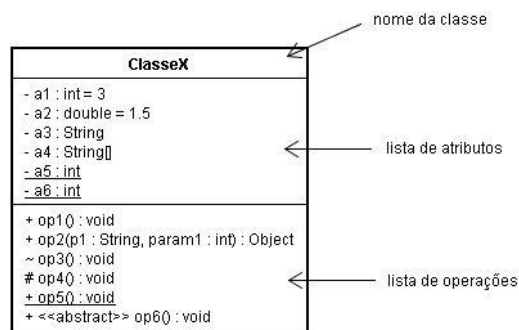
© LES/PUC-Rio

15

## Representação de Classes na UML



- Na UML, uma classe é graficamente representada por um retângulo dividido em três seções: nome, lista de atributos e lista das operações.




© LES/PUC-Rio

16



Laboratório de Engenharia de Software

## Generalização




- A **generalização** é atividade de identificar aspectos comuns e não comuns entre conceitos pertencentes a um domínio de aplicação;
- Ela nos permite definir relações entre superclasses – conceitos gerais – e subclasses – conceitos específicos;
- Tais relações formam uma **taxonomia** de conceitos de um certo domínio, que é representada por meio de uma hierarquia de classes.

© LES/PUC-Rio
17

Laboratório de Engenharia de Software

## Generalização em Java



- Em Java, a hierarquia de polígonos vista anteriormente pode ser definida da seguinte maneira:

```

public class Poligono { }

public class Triangulo extends Poligono { }

public class Retangulo extends Poligono { }

public class Pentagono extends Poligono { }

public class Quadrado extends Retangulo { }
```

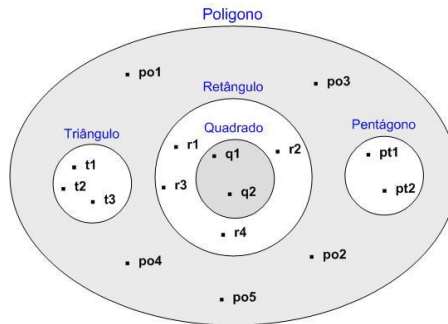
- A palavra reservada **extends** é usada para declarar que uma classe (por exemplo, Triangulo) é subclasse de outra (por exemplo, Poligono).

© LES/PUC-Rio
18

## Teoria dos Conjuntos



A generalização pode ser vista sob a ótica da **Teoria dos Conjuntos**. Desse ponto de vista, o diagrama de **Venn** abaixo é equivalente ao diagrama de classes do slide anterior.



© LES/PUC-Rio

19

## Benefícios




- Identificar superclasses e subclasses oferece os seguintes benefícios:
  - Permite compreender aspectos de um problema em termos mais gerais e abstratos;
  - Resulta em mais expressividade, melhoria na compreensão e redução das redundâncias de um modelo;
  - A utilização de superclasses e subclasses, juntamente com o mecanismo de polimorfismo, permite construir softwares bem organizados.

© LES/PUC-Rio

20

Laboratório de Engenharia de Software

## Regra "É-UM"




- Pode-se afirmar, informalmente, que toda instância de uma classe é também instância da sua superclasse;
- Isso é conhecido como a regra **é-um** (is-a): todo **quadrado** é um **retângulo**, e todo **retângulo** é um **polígono**;
- Logo, pode-se concluir que todas as propriedades válidas para os objetos de uma classe também são válidas para os objetos de suas classes descendentes.

© LES/PUC-Rio

21

Laboratório de Engenharia de Software

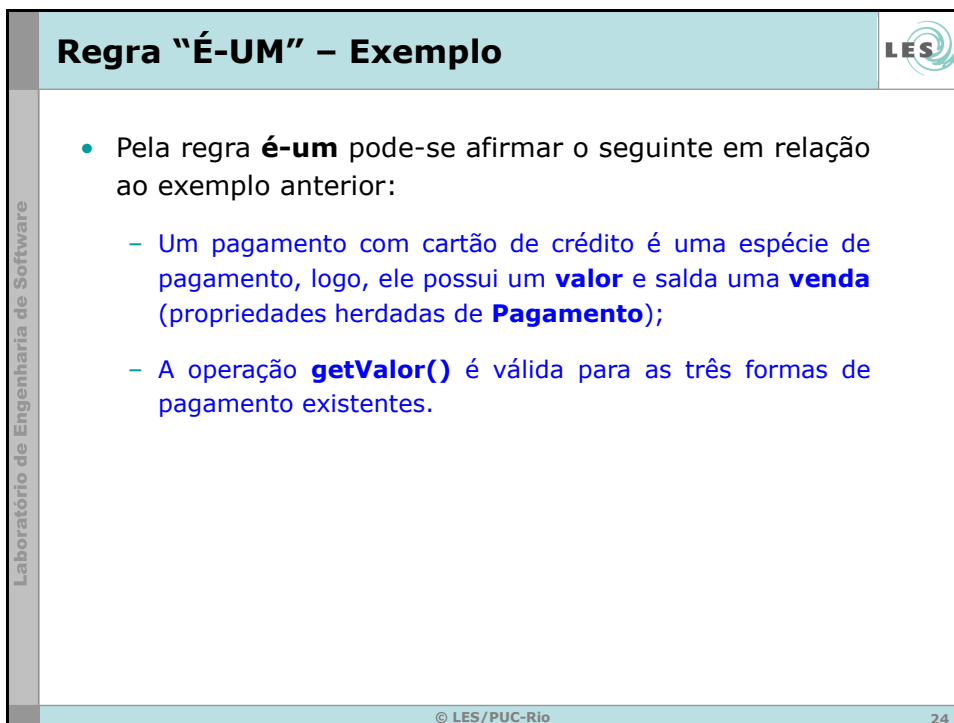
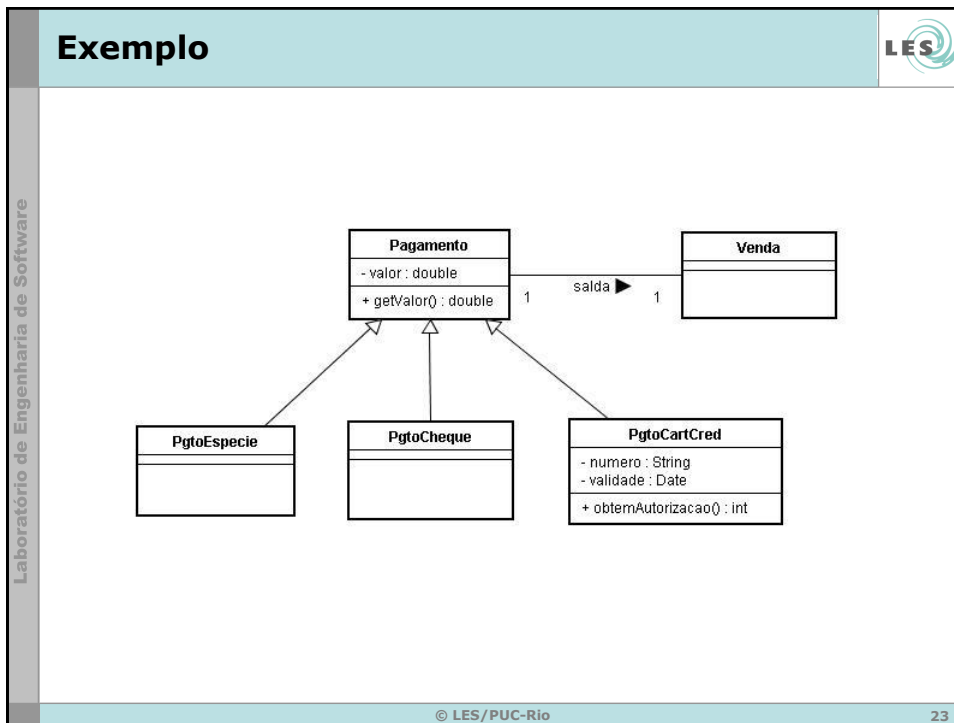
## Propriedades de uma Classe



- Por propriedades entende-se:
  - Atributos (variáveis)
  - Operações (métodos)
  - Relações (variáveis)


© LES/PUC-Rio

22



Laboratório de Engenharia de Software

Regra “É-UM” – Comentários




- Nem todas as propriedades válidas para os objetos de uma classe são válidas para os objetos da sua superclasse;
- No exemplo anterior, o **número** e a **validade** de um cartão são propriedades exclusivas de pagamentos feitos com cartão de crédito, não sendo aplicáveis a outras formas de pagamento;
- Pode-se, também, associar um pagamento com cartão a uma classe que represente a operadora do cartão usado;
- Tal associação será válida apenas para os pagamentos com cartão.

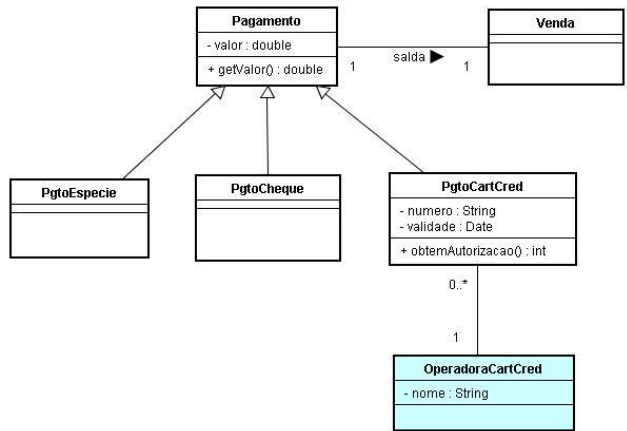
© LES/PUC-Rio

25

Laboratório de Engenharia de Software

Exemplo – Pagamento com Cartão





```
classDiagram
    class Pagamento {
        -valor : double
        +getValor() : double
    }
    class Venda {
    }
    class PgtoEspecie {
    }
    class PgtoCheque {
    }
    class PgtoCartCred {
        -numero : String
        -validade : Date
        +obtemAutorizacao() : int
    }
    class OperadoraCartCred {
        -nome : String
    }

    Pagamento <|-- PgtoEspecie
    Pagamento <|-- PgtoCheque
    Pagamento <|-- PgtoCartCred
    Pagamento "1" -- "1" Venda : salda
    PgtoCartCred "0..*" -- "1" OperadoraCartCred
```


The diagram illustrates a payment system. It features a base class **Pagamento** with attributes `- valor : double` and a method `+ getValor() : double`. Three subclasses inherit from **Pagamento**: **PgtoEspecie**, **PgtoCheque**, and **PgtoCartCred**. **PgtoCartCred** has additional attributes `- numero : String` and `- validade : Date`, and a method `+ obtemAutorizacao() : int`. An association labeled `salda` connects **Pagamento** (multiplicity 1) to **Venda** (multiplicity 1). Another association connects **PgtoCartCred** (multiplicity 0..\*) to **OperadoraCartCred** (multiplicity 1). The **OperadoraCartCred** class has an attribute `- nome : String`.

© LES/PUC-Rio

26

Laboratório de Engenharia de Software

## Generalização Simples




- Em todos os exemplos de generalização apresentados até agora foram usadas, apenas, generalização simples;
- Generalização simples significa que uma classe só pode ser descendente direta de uma única superclasse;
- Além da generalização simples, todos esses exemplos contêm apenas generalizações não-sobrepostas;
- Sob a ótica da **Teoria dos Conjuntos**, uma generalização não-sobreposta significa que a interseção entre duas classes (conjuntos) quaisquer é sempre vazia, exceto quando uma for subclasse da outra – direta ou transitivamente;
- Nesse caso, a interseção será a própria subclasse.

© LES/PUC-Rio 27

Laboratório de Engenharia de Software

## Generalização e Herança




- Não se deve confundir generalização com herança;
- Generalização é uma relação entre classes, em que uma classe mais especializada (subclasse) é definida em termos de uma classe mais geral (superclasse);
- A herança é o mecanismo que permite que todas as propriedades não privadas de uma classe também façam parte do espaço de nomes de suas subclasses.

© LES/PUC-Rio 28

Laboratório de Engenharia de Software

## Programa – Capítulo 4




- Tipos Enumerados
- Generalização e Herança
- **Visibilidade**
- Chamada de Construtores
- A Classe `Object`

© LES/PUC-Rio

29

Laboratório de Engenharia de Software

## Visibilidade



- Determina o nível de restrição de acesso às construções de um programa Java;
- Aplicável a classes, a métodos e a variáveis;
- Tipos de visibilidade:
  - pública `“+”`
  - protegida `“#”`
  - privada `“-”`
  - *default* ou pacote `“~”`

© LES/PUC-Rio

30

## Modificador de Acesso public



- Todas as propriedades (variáveis e métodos) públicas de uma classe são visíveis em todas as classes de uma aplicação.

```
package poligonos;

public class Poligono {
    public int numLados;
}

----- pacote default -----
import poligonos.*;

public class Ex {

    public static void main(String[] args) {
        Poligono p=new Poligono();
        p.numLados=4; //correto: numLados é visível
    }
}
```

© LES/PUC-Rio

31

## Modificador de Acesso private



- Todas as propriedades (variáveis e métodos) privadas de uma classe não são visíveis em suas subclasses ou em quaisquer outras classes.

```
package poligonos;

public class Poligono {
    private int numLados;
}

package poligonos;

public class Retangulo extends Poligono {
    public Retangulo() {
        numLados=4; //erro: numLados não é visível
    }
}
```

© LES/PUC-Rio

32



## Modificador de Acesso protected (1)



- Todas as propriedades (variáveis e métodos) protegidas de uma classe são visíveis em suas subclasses ...

```
package poligonos;

public class Poligono {
    protected int numLados;
}

package meupacote;
import poligonos.Poligono;

public class Hexagono extends Poligono {
    public Hexagono() {
        numLados=6; //correto: numLados é visível
    }
}
```

© LES/PUC-Rio

33

## Modificador de Acesso protected (2)



- ... e nas classes definidas no mesmo pacote.

```
package poligonos;

public class Poligono {
    protected int numLados;
}

package poligonos;


public class OutraClasse {
    private Poligono p=null;

    public void umMetodo() {
        p.numLados=0; //correto: numLados é visível
    }
}
```

© LES/PUC-Rio

34

## Modificador de Acesso `protected` (3)




- Entretanto, elas se comportam como se fossem privadas nas classes pertencentes a outros pacotes.

Laboratório de Engenharia de Software

© LES/PUC-Rio 35

## Modificador de Acesso `protected` (4)



```
package poligonos;

public class Poligono {
    protected int numLados;
}

package p1;

import poligonos.*;

public class UmaClasse {
    Poligono p=new Poligono();

    public UmaClasse() {
        p.numLados=4; //erro: numLados não é visível
    }
}
```

Laboratório de Engenharia de Software

© LES/PUC-Rio 36

## Modificador de Acesso Default (1)



- Todas as propriedades de uma classe (variáveis e métodos) com acesso default (sem modificador) são visíveis em todas as classes do mesmo pacote ...

```
package poligonos;

public class Poligono {
    int numLados;
}

package poligonos;

public class OutraClasse {
    private Poligono p=null;

    public void umMetodo() {
        p.numLados=0; //correto: numLados é visível
    }
}
```

© LES/PUC-Rio

37

## Modificador de Acesso Default (2)



- ... e invisíveis nas classes definidas em outros pacotes.

```
package poligonos;

public class Poligono {
    int numLados;
}

package meupacote;
import poligonos.Poligono;

public class Hexagono extends Poligono {
    public Hexagono() {
        numLados=6; //erro: numLados não é visível
    }
}
```

© LES/PUC-Rio

38

Laboratório de Engenharia de Software

## Programa – Capítulo 4




- Tipos Enumerados
- Generalização e Herança
- Visibilidade
- **Chamada de Construtores**
- A Classe `Object`

© LES/PUC-Rio 39

Laboratório de Engenharia de Software

## Sequência de Chamada dos Construtores (1)



- Quando um objeto de uma classe é criado, todos os construtores das classes ancestrais serão invocados, implícita ou explicitamente;
- No exemplo a seguir, uma condição de erro será sinalizada pelo runtime de Java quando um quadrado for criado, pois o construtor default da classe `Poligono` não foi definido explicitamente.

```
import poligonos.*;

public class Main {
    public static void main(String[] args) {
        Quadrado p1=new Quadrado(4);
    }
}
```

© LES/PUC-Rio 40

## Sequência de Chamada dos Construtores (2)



```
public class Poligono {
    protected int numLados;
    protected double perimetro;

    public Poligono(int x) {
        numLados=x;
    }
}

public class Retangulo extends Poligono {

}

public class Quadrado extends Retangulo {

    public Quadrado(int x) {
        numLados=x;
    }
}
```


## Sequência de Chamada dos Construtores (3)



- O erro é causado pela invocação implícita do construtor default de **Retangulo**, que por sua vez invoca, implicitamente, o construtor default de **Poligono**;
- Como o construtor default de **Poligono** não foi definido, embora exista outro construtor em **Poligono**, o erro é sinalizado.

Laboratório de Engenharia de Software

## A Palavra Reservada `super` (1)



- A solução é definir um construtor em **Retangulo**, que por sua vez irá invocar o construtor de **Poligono**;
- Tal invocação é feita por meio da palavra reservada `super`.

© LES/PUC-Rio 43

Laboratório de Engenharia de Software

## A Palavra Reservada `super` (2)



```
public class Poligono {
    protected int numLados;
    protected double perimetro;


    public Poligono(int x) {
        numLados=x;
    }
}

public class Retangulo extends Poligono {
    public Retangulo() {
        super(0);
    }
}

public class Quadrado extends Retangulo {
    public Quadrado(int x) {
        numLados=x;
    }
}
```

© LES/PUC-Rio 44

Programa – Capítulo 4




Laboratório de Engenharia de Software

- Tipos Enumerados
- Generalização e Herança
- Visibilidade
- Chamada de Construtores
- **A Classe Object**

© LES/PUC-Rio

45

A Classe Object



Laboratório de Engenharia de Software

- A classe **java.lang.Object** é a derradeira ancestral de qualquer classe em Java;
- Isto é, toda classe definida em Java herda implicitamente as propriedades de **Object**;
- Ela possui alguns métodos bastante úteis, que podem ser invocados sobre qualquer objeto;
- Por exemplo, o método `equals()`, definido na classe **Object**, nos permite implementar o conceito de igualdade de maneira um pouco mais sofisticada do que o simples uso do operador relacional `==`.

© LES/PUC-Rio

46

## O Operador ==



- O operador relacional `==` compara apenas se duas variáveis de objeto referenciam o mesmo objeto;
- No exemplo abaixo, a mensagem **p1 e p2 referenciam o mesmo objeto** será exibida no console como resultado da comparação de p1 com p2.

```
import poligonos.*;

public class Ex {
    public static void main(String[] args) {
        Quadrado p1=new Quadrado(),p2;
        p2=p1;
        if(p2==p1)
            System.out.println("p1 e p2 referenciam o mesmo objeto");
    }
}
```

## O Método `Object.equals()`



- O método `equals()` implementado em **Object** produz o mesmo resultado que o operador `==`;
- Caso se queira implementar outro conceito de igualdade, deve-se sobrescrever o método `equals()` e fornecer uma nova implementação;
- No exemplo a seguir, o método `equals()` é usado para definir um novo conceito de igualdade;
- Nesse caso, dois quadrados são considerados iguais se tiverem perímetros iguais.



## Implementação da Igualdade (1)



```
package poligonos;

public class Quadrado extends Retangulo {

    public Quadrado(Double p) {
        perimetro=p;
    }

    public boolean equals(Quadrado q) {
        if(this.perimetro==q.perimetro)
            return true;
        else
            return false;
    }
}
```

## Implementação da Igualdade (2)



- No exemplo a seguir, a mensagem **p1 e p2 são diferentes** será exibida no console como resultado da comparação entre **p1** e **p2**.

```
import poligonos.*;

public class EX {
    public static void main(String[] args) {
        Quadrado p1=new Quadrado(16.0),p2=new Quadrado(17.0);

        if(p2.equals(p1))
            System.out.println("p1 e p2 são iguais");
        else
            System.out.println("p1 e p2 são diferentes");
    }
}
```