Laboratório de Engenharia de Software

INF1636 – PROGRAMAÇÃO ORIENTADA A OBJETOS

Departamento de Informática – PUC-Rio

Ivan Mathias Filho
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`

Programa – Capítulo 4

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

Tipos enumerados – C++

- 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

Tipos enumerados – Motivação

- Antes da edição JSE5 (JDK 1.5) a linguagem Java não oferecia suporte direto a tipos enumerados;
- Enumerados eram simulados por meio de 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;
```


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.

Tipos enumerados – Características

- Os tipos enumerados resolvem estes problemas da seguinte maneira:
  - Os enumeradores 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.

Tipos enumerados – Exemplo

- O tipo `Cor`, visto anteriormente, pode ser definido da seguinte maneira, utilizando-se um enumerado:

```
public enum Cor {
  amarelo,vermelho,azul,verde,preto;
}
```

- A atribuição de um enumerador a uma variável do tipo `Cor` tem a seguinte forma:

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

Tipos enumerados – Vantagens

- Os enumeradores não são compilados para as classes que os usam;
- Cada enumerador é deixado 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 um enumerador usado por outra classe qualquer será sinalizada com uma mensagem de erro pela biblioteca de tempo de execução, assim que o enumerador retirado for referenciado.

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;
}
```

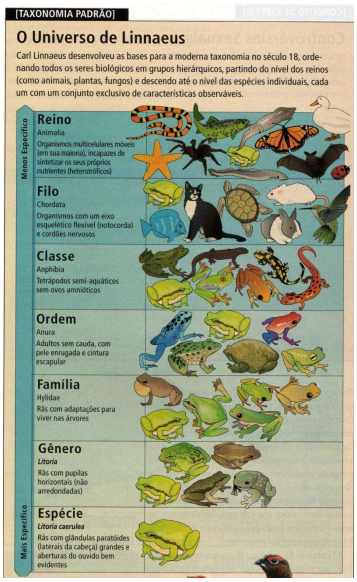
Programa – Capítulo 4

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

Taxonomia (1)

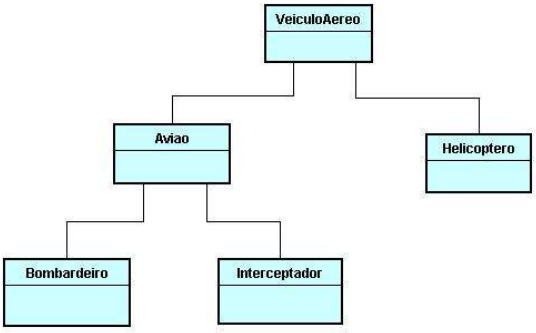
- Muito antes do alvorecer da ciência os seres humanos já nomeavam as espécies;
- Isso lhes permitia obter sucesso nas suas atividades de caça e coleta;
- A **Taxonomia**, palavra de origem grega cujo significado é "estudo das classificações", surgiu no século XVII;
- Ela ganhou força no século seguinte, graça ao trabalho do naturalista sueco **Carl Linnaeus**, que inventou um sistema para organizar os seres vivos em grupos cada vez menores;
- Nesse sistema, os membros de um grupo particular compartilham determinadas características.

Taxonomia (2)

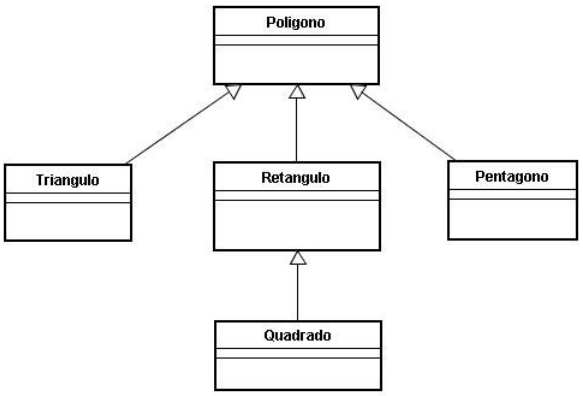


Taxonomia

- 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.

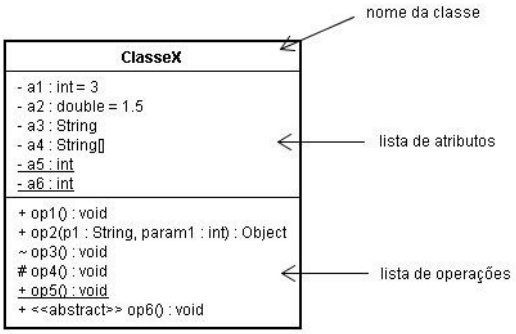


Exemplo – Taxonomia de polígonos



Representação de classes em UML

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



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.

Generalização em Java

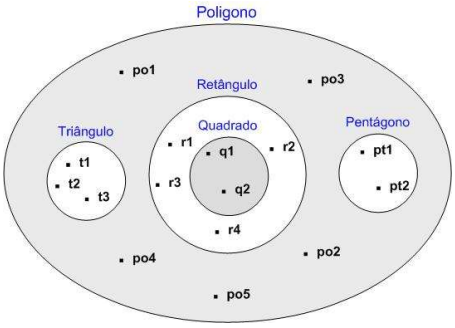
- Em Java, a hierarquia de polígonos vista anteriormente pode ser definida da seguinte maneira:

```
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`).

Teoria de conjuntos

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



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.

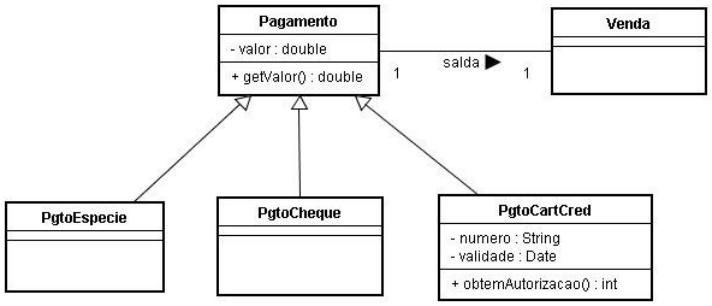
Regra "É-UM"

- 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.

Propriedades de uma classe

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

Exemplo



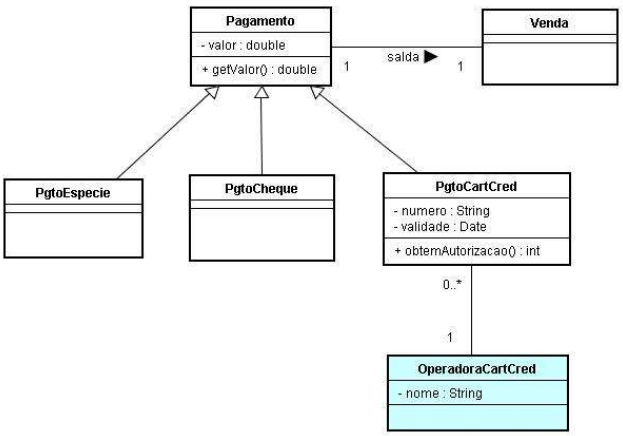
Regra "É-UM" – Exemplo

- Pela regra **é-um** pode-se afirmar o seguinte em relação ao exemplo anterior:
  - Um pagamento com cartão de crédito é uma espécie de pagamento, logo, ele possui um **valor** e salda uma **venda** (propriedades herdadas de **Pagamento**);
  - A operação **getValor()** é válida para as três formas de pagamento existentes.

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.

Exemplo – Pagamento com cartão



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 de 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.

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.

Programa – Capítulo 4

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

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 "**~**"

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
  }
}
```


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
  }
}
```

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
  }
}
```


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
   }
}
```

Modificador de acesso `protected` (3)

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

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
  }
}
```

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
   }
}
```


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
  }
}
```

Programa – Capítulo 4

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

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 compilador Java quando a classe `Retangulo` for compilada, 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);
   }
}
```

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.

A palavra reservada `super` (1)

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

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;
  }
}
```

Programa – Capítulo 4

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

A classe Object

- 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 **==**.

O operador ==

- O operador relacional **==** determina 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");
  }
}
```