Herança

Neste módulo veremos:

- Conceitos de herança em POO; e
- sobrecarga + coreção + uma citação de polimorfismo

O conceito de herança é um dos conceitos fundamentais de POO. **Herança**, na prática, significa a possibilidade de construir objetos especializados que herdam as características de objetos mais generalistas, ou ainda, a herança uma maneira de reutilizar código a medida que podemos aproveitar os atributos e métodos de classes já existentes para gerar novas classes mais específicas que aproveitarão os recursos da classe hierarquicamente superior.

O conceito de herança mimetiza as características hierárquicas de vários sistemas reais, como por exemplo, os sistemas de classificação em biologia que, pode determinar como uma hierarquia o seguinte:

- animais;
- vertebrados e invertebrados;
- mamíferos e aves;
- entre outras características mais específicas

Outras hierarquias são possíveis, por exemplo, na área da saúde a hierarquia dos agentes envolvidos poderia ser esta: pessoas, (empregados, terceirizados e pacientes), (médicos, biomédicos, enfermeiros...), (pacientes particulares, pacientes SUS) e assim, mais uma vez, a caminho de classes mais especializadas.
Notem como estas hierarquias podem ser facilmente descritas numa estrutura de **árvore**. Nesta estrutura a raíz da árvore é o agente da qual originam todos os outros agentes. Podemos dizer que cada agente **herda as características** dos seus antecessores.
Na visão POO podemos afirmar que pessoas é uma classe hierarquicamente superior e que dela são herdadas características para a formação de novas classes, como: empregados, pacientes, entre outras.

Superclasses e subclasses

Em POO todo objeto de uma classe construída pelo usuário da linguagem é também um objeto de outra classe. Podemos afirmar isso pois, por exemplo, em Java existe um objeto de qual todas as classes são originadas que é o `Object`.

Por exemplo, na hierarquia da área da saúde, podemos dizer que pessoa é uma **superclasse** e que empregados é uma **subclasse** de pessoa.

A herança normalmente produz subclasses mais especializadas, mais específicas, que as superclasses. Notem que, o termo subclasses pode trazer a idéia **errada** de que uma subclasse possa ter menos recursos que a superclasse mas ocorre exatamente o inverso, **uma subclasse é mais especializada que a superclasse**.

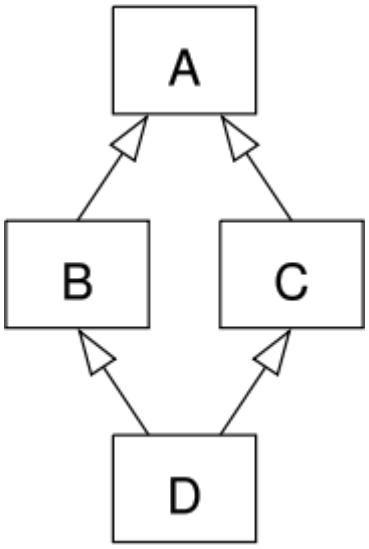
Guardem bem esta máxima:
Uma subclasse guarda a relação é um com a superclasse.

Herança simples e herança múltipla

Nossos cometárioa até o momento basearam-se num esquema de herança chamada de **herança simples** em que um objeto herda características somente de uma superclasse. No entanto, o conceito de herança não é tão restritivo assim, como em muitos casos da natureza, o conceito em si permite a **herança múltipla**, artifício que permite a um objeto herdar as características de mais de uma superclasse.

A herança múltipla é muito controversa na Computação pois pode gerar algumas situações intrincadas, para não dizer confusas. O ponto principal da discussão é que as subclasses são classes mais especializadas que as superclasses assim, herdar características muito específicas de outras classes que também podem ser específicas pode gerar mais dúvidas do que soluções. A herança oferece um poder imenso em POO mas lembrem-se que as agregações também.

Um dos problemas clássicos de ambiqüidade e complexidade apontados pela herança múltipla é o **problema do diamante**, como mostrado na figura abaixo (fonte: Wikipidia domínio público):



Na figura acima as classes B e C herdam de A, ou seja, são mais especializadas. Destas duas subclasses é gerada uma nova classe, D, que herda de B e C. Estranho mas é possível. Pergunta-se: se chamamos um método em D, conhecido em A por herança, de onde virá esta herança, de A ou de B?

Algumas linguagens de programação, como Java, resolvem este problema do seguinte modo: definem que uma classe pode herdar características de `interface` múltiplas mas compromete-se só com herança simples.

Herança dos rótulos

Até agora vimos que os membros (atributos e comportamentos) de uma classe poderiam ser rotulados como `public` ou `private`. Com a inclusão do conceito de herança podemos apresentar um novo rótulo, o **protected**.

O rótulo `protected` permite que os membros das subclasses, ou de outras classes de um mesmo **pacote**, podem acessar os membros `protected` da superclasse.

Em Java uma subclasse não pode acessar diretamente os membros `private` de uma superclasse. Se isso fosse possível acabaria com os benefícios do ocultamento de informações.

Vejam que o rótulo `protected` serve como uma proteção intermediária entre os rótulos `private`, bastante restritivo, e `public`, aberto.

Mas o conceito de herança não tem só vantagens. Notem que um software mal projetado pode fazer com que uma subclasse possa herdar recursos que não seriam apropriados para ela. Este efeito de propagação de recursos pode ter efeitos interessantes se bem planejados mas destrutivos se mal planejados.
A herança promove a proparagação de TODOS os recursos.

Um pouco de prática

Vejamos o código abaixo que servirá como superclasse da classe `Circulo` a ser vista posteriormente.

```
/** Ponto versao 01
 * @author baseado em Deitel
 * @version 20.ago.2006
 */
 public class Ponto {
   protected int x,y;

   public Ponto()
   {
     x =0; y = 0;
   }
```

```
   public Ponto(int xc, int yc)
   {
           x = xc; y = yc;
   }

   public String toString()
   {
           return("(" +x +", " +y +")");
   }
 }
```

Prestem atenção no rótulo `protected` dos atributos da classe.

Sobrescrição

A primeira característica marcante da classe acima é que existem dois construtores para a classe `Ponto`. Obviamente com assinaturas diferentes. Esta possibilidade de definir métodos com o mesmo nome (e assinaturas diferentes) é chamada de **sobrescrição**.

Outra característica de herança interessante de ser observada esta no método `toString`. Na verdade este método existe na classe `Object` e aqui ele está sobrescrito. Lembre-se que, mesmo sem especificar, por `default` a classe `Ponto` é herdeira da classe `Object`.
Vejamos como este método pode ser carregado **dinâmicamente** pela classe `TestaPonto` descrita abaixo.

Esta sobrescrição é também um exemplo de **polimorfismo** (que veremos mais adiante), uma maneira de oferecer múltiplas formas a um mesmo método.

```
/** TestaPonto
 */
 public class TestaPonto {
   public static void main(String []xyz) {
     Ponto m, n;

     m = new Ponto();
     System.out.println(m);
     n = new Ponto(10,20);
     System.out.println(n);
   }
 }
```

Mais uma vez gostaria de chamar a atenção para a sobrescrição dos construtores e para a chamada dinâmica do método `toString` da classe `Ponto`.

A subclasse e as relações com a superclasse: Prática

Vejamos a classe abaixo que define uma subclasse da classe `Ponto`.

```
/** Circulo
 * @author baseado em Deitel
```

```
 * @version 20.ago.2006
 */
 public class Circulo extends Ponto{
   protected double raio;

   public Circulo()
   {
     setRaio(0);
   }

   public Circulo(double raio, int xc, int yc)
   {
           super(xc, yc);
           setRaio (raio);
   }

   public void setRaio(double raio)
   {
           if(raio>0)
             this.raio=raio;
           else this.raio=0;
   }

   public String toString()
   {
           return ("raio: " +raio+" "+ super.toString());
   }
 }
```

Logo na declaração da classe `Circulo` vemos o conceito de herança com a palavra `extends` que em Java deve ser lida como **herda**. Ou seja, esta nova classe será uma subclasse (uma classe mais especializada) da classe `Ponto`.

Chamada explícita para o construtor

Outra observação importante: no segundo construtor aparece uma chamada explícita ao construtor da superclasse `Ponto`. Os argumentos são passados para a classe mais acima hierarquicamente através desta chamada. Notem que, se não houvesse esta chamada explícita mesmo assim ocorreria uma chamada implicita para inicializar os valores das coordenadas e isto seria feito pelo construtor padrão, aquele sem argumentos de entrada, que simplesmente zera as variáveis `x` e `y`.

As vantagens da herança na reutilização do código são mais claras no método `toString` que, através novamente da chamada `super` completa a escrita de um objeto `Circulo` com o método já descrito na superclasse `Ponto`.

Utilizando herança em objetos: Prática

Veja a criação de objetos baseada no conceito de herança.

```
/** TestaCirculo
*/
public class TestaCirculo {
```

```
  public static void main(String []xyz) {
    Ponto p1, p2;
    Circulo c1, c2;

    p1 = new Ponto(10,10);
    c1 = new Circulo(1.34, 50,50);

    System.out.println("Ponto p1 "+ p1);
    System.out.println("Circulo c1 "+ c1);

    //coersao (conversao) de super para sub
    //usa conceito "eh um"
    p2 = c1;
    System.out.println("Ponto p2 (via Circulo) "+ p2.toString());

    // coersao de sub para super
    c2 = (Circulo)p2;
    System.out.println("Circulo c2 "+ c2);

     if(p1 instanceof Circulo)
    {
            c2 = (Circulo)p1;
    }
    else
      System.out.println("p1 nao eh uma instancia de Circulo");
  }
  }
}
```

Saída:

```
Ponto p1 (10, 10)
Circulo c1 raio: 1.34 (50, 50)
Ponto p2 (via Circulo) raio: 1.34 (50, 50)
Circulo c2 raio: 1.34 (50, 50)
p1 nao eh uma instancia de Circulo
```

Coersão na herança: detalhes

Como vimos, um objeto responde ao tipo com que foi criado, ou de forma mais completa, um objeto é do tipo que ele foi criado. Sabemos também que, em Java, todas as classes descendem da classe `Object` assim, todo objeto, além do seu próprio tipo é também um `Object`.

No exemplo acima, podemos dizer que todo `Circulo` é um `Ponto`. O reverso não é verdade pois, `Ponto` pode ser um `Circulo` mas não necessariamente pois `Ponto` poderia ter outras subclasses originárias dele.

A coersão mostra somente que uma referência a um objeto diferente pode ser usada no lugar da referência do mesmo tipo do objeto. O comando de atribuição `p2 = c1;` faz isso. Nesta linha estamos dizendo que `c1` pode atribuir a sua referência a `p2`. Notem que como são referências não estamos fazendo uma atribuição de objetos. Este tipo de coersão é chamada de **coersão implícita**.

Notem que, pela coersão implícita sempre podemos fazer:

```
  Object obj = new Circulo(10, 20, 30);
```

Outro modo de coersão é a coersão de super para sub, ou seja, da superclasse para a subclasse. Como uma superclasse pode ter, em tese, mais de uma subclasse, esta coersão deve ser uma **coersão explícita**, ou seja, devemos especificar qual a subclasse será usada como meio para utilizar a subclasse. Por isso tivemos no código acima a linha `c2 = (Circulo)p2`, ou seja, a referência `p2` pode ser usado como referência uma referência a uma subclasse mas esta deve ser especificada pois, como sabemos, podemos ter mais de uma subclasse a cada superclasse.

Coersão nos métodos: detalhes

Veremos agora como os métodos de instância e os métodos de classe são afetados pela coersão de tipos na linguagem Java.

Codifiquem e executem os códigos abaixo:

```
/** Moto.java
 */

public class Moto
{
    public void metodoInstancia()
    {
        System.out.println("Metodo de instancia de Moto (super).");
    }

    public static void metodoClasse()
    {
        System.out.println("Metodo de classe de Moto (super).");
    }
}
```

e a classe

```
/** Triciclo.java
 */

public class Triciclo extends Moto
{
    public void metodoInstancia()
    {
        System.out.println("Metodo de instancia de Triciclo (sub).");
    }

    public static void metodoClasse()
    {
        System.out.println("Metodo de classe de Triciclo (sub).");
    }

    public static void main(String[] args)
```

```
    {
        Triciclo tri;
        Moto biz;

        tri = new Triciclo();
        biz = tri;

        biz.metodoInstancia();
        biz.metodoClasse();
    }
}
```

No exemplo acima vimos que houve uma coersão implícita pela atribuição `biz = tri;`. Pela execução do código podemos perceber que, quando um objeto instanciado de uma subclasse recebe uma coerção para uma superclasse ele mantém os seus métodos de instância (ele ainda é um objeto sub) mas para os métodos de classe ele responde aos métodos da superclasse.

Podemos dizer então que, os métodos estáticos da superclasse prevalescem sob a subclasse.

Diferença entre sobrescrição e esconder

Podemos verificar pelas classes acima que os métodos de instância de uma subclasse sobrescrevem os métodos de instância da superclasse, ou seja, eles prevalescem sob os métodos de seus pais.

No entanto, quando temos uma coersão de superclasse para uma subclasse outra regra acontece. Podemos perceber nesta coersão que os métodos de classe das subclasses **escondem** os métodos de classe das superclasses. Estes métodos das superclasses só são revelados quando há uma coersão de sub para super.

Herança versus coposição

Como vimos, os relacionamentos tipo é um são os relacionamentos definidos por herança. Na aula em que vimos a classe `Relogio` a escrevemos como uma composição por duas outras classes chamadas `Mostrador`. Esta também é uma construção muito usada em Engenharia de Software e que revela o relacionamento tipo tem um, a associação. Há quem afirme que devemos sempre preferir a composição de objetos em detrimento a herança, mas isso nem sempre é possível além de ser uma afirmação ainda polêmica na Engenharia de Software

Exercícios

1. Escreva um método `getRaio` para a classe `Circulo`.

2. Acrescente um método para proporcionar a área do `Circulo`. Este método deve retornar um valor `double`. Para o valor da constante pi veja a classe `Math`.
3. Rescreva o método `toString` da classe `Circulo` para acessar diretamente os atributos `x` e `y` da classe `Ponto` que estão rotulados como `protected`.
4. Rescreva a classe `Circulo` usando composição de classes ao invés de herança.

Última modificação feita em: 12 de outubro de 2008
Evandro Eduardo Seron Ruiz, PhD *(evandro at usp ponto br)*