Material de Apoio 5

Herança

Observe o código das classes Fatorial e Fibonacci apresentados abaixo.

```
class Fatorial {
   private int n, res;

   public Fatorial( int aux ) {
     n = aux;
     res = 1;
   }

   public void calcula() {
     int i;

     for( i = 1 ; i < n ; i++ )
       res = res * i;
   }

   int getRes() {
      return res;
   }
}
```

```
class Fibonacci {
   private int n, res;

   public Fibonacci( int aux ) {
     n = aux;
     res = 0;
   }

   public void calcula() {
     int i, t, a = 0, b = 1;

     for( i = 1 ; i < n ; i++ ) {
       res = a + b;
       a = b;
       b = res;
     }
   }

   int getRes() {
      return res;
   }
}
```

O que estas classes possuem em comum?
1) ____________________
2) ____________________
3) ____________________

Modelagem de classes

A programação orientada a objetos implica na necessidade do programador conceber uma representação para o problema tratado em termos de classes e de relações entre classes. Duas destas relações são as relações *dependência* e *associação.* A dependência denota uma relação de **uso**: caracterizada pelo aplicação de funcionalidades de uma classe por outra. A relação de associação reflete uma **composição**, onde uma classe utiliza uma outra para definir um atributo.

O mecanismo de herança implementa uma relação entre classes chamada de generalização ou especialização. Este tipo de relação pode ser considerado uma relação **é um**. Uma classe que herda uma outra classe herda, além do código, toda a concepção da classe original, em particular a especificação de serviços. O termo generalização exprime a idéia que a superclasse contém uma especificação mais genérica do modelo que está sendo criado. O termo especialização, por sua vez, exprime a idéia de que uma classe derivada refina uma especificação para caracterizar as funcionalidades específicas a um caso particular do modelo criado.

É importante observar que a relação de herança permite oferecer diferentes níveis de abstração do problema tratado: quanto mais alta estiver uma classe na hierarquia de herança, mais abstrato são os conceitos envolvidos e mais genérico é o tratamento apresentado.

Herança: hierarquia entre classes para reaproveitamento de código

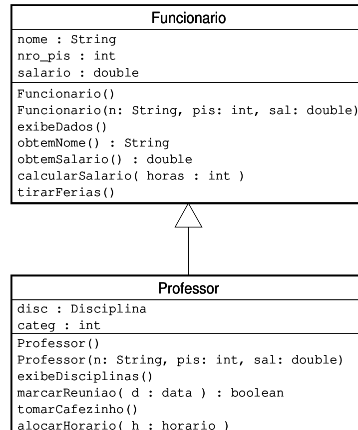
A herança é um recurso de programação do paradigma orientado a objetos que permite o reaproveitamento de esforço já despedido no desenvolvimento de um sofware. Este recurso permite que uma classe seja definida herdando características de uma outra classe já existente. Da-se a denominação de superclasse a classe original e de subclasse a nova classe. Outras nomenclaturas populares: classe base e classe derivada, classe mãe e classe

filha (note que não existem outros parentescos, tipo: classes irmãs). Uma das grandes vantagens da herança é de diminuir a necessidade de replicar código em um programa, permitindo que trechos de códigos definidos para uma classe sejam reaproveitados na construção de outras.

É importante ressaltar que quando uma classe é herdada, mais do que atributos e métodos, também é herdada a estrutura desta classe. Assim, a herança entre duas classes define uma relação de **é um**. Ou seja, considere que a classe `Y` seja subclasse da classe `X`. Caso seja criado um objeto `y1` da classe `Y`, este objeto é um objeto da classe `Y` e também é um objeto da classe `X`. O inverso, no entanto, não é verdadeiro: um objeto criado da classe `X` não é necessariamente um objeto da classe `Y`.

Tomemos como exemplo, a necessidade de definir a classe `Professor` em software de gestão de recursos humanos. Objetos da classe `Professor` necessitam manipular atributos específicos deste tipo de funcionário de uma empresa, como `categoria` (para enquadramento funcional) e `disciplinas` que leciona. Também são necessários alguns métodos: `alocarHorario`, `tomarCafezinho` e `marcarReuniao`. No entanto, objetos da classe `Professor` também necessitam manipular informações referentes a funcionários em geral, tal como `nome` e `nro_pis` e da mesma forma responder por ações genéricas a todos funcionários de uma empresa, como `tirarFerias` e `calcularSalario`. Estas características (atributos e métodos) podem ser regrupados em uma superclasse no software: a classe `Funcionário`. Desta forma, toda especificação desenvolvida para `Funcionario` seria reaproveitada na classe Professor e em qualquer outra classe necessária para outro tipo de funcionário da empresa.

Em uma forma gráfica, representamos a herança desta forma:



Sintaxe da herança em Java

Em Java, a herança pode ser utilizada através da palavra reservada `extends`, da seguinte forma:

```
class NovaClasse extends ClasseJaExistente {
  ...
}
```

Neste caso `NovaClasse` é a classe que está sendo definida a partir da classe original `ClasseJaExistente`. A classe `NovaClasse` é dita classe derivada e a class `ClasseJaExistente` é dita super-classe.

A criação de objetos se dá da seguinte forma:

```
NovaClasse obj1;        // Referencia para objetos NovaClasse
ClasseJaExistente obj2; // Referencia para objetos ClasseJaExistente
```

```
obj1 = new NovaClasse();        // Construcao valida
obj2 = new ClasseJaExistente(); // Construcao valida

obj2 = new NovaClasse(); // Construcao valida: um objeto da classe  NovaClasse tambem
                         // e' um objeto da classe ClasseJaExistente
obj1 = new ClasseJaExistente(); // Construcao invalida: um objeto da classe
                                // ClasseJaExistente nao e', necessariamente um objeto NovaClasse
```

Relação entre membros da super e da classe derivada

Quando uma nova classe estiver sendo criada a partir de uma classe já existente através do mecanismo de herança, a relação entre os membros se dá das seguintes forma:

- Métodos: O mecanismo de herança permite a exploração de três recursos de programação no desenvolvimento de novas classes. A **herança** propriamente dita é o primeiro recurso. A herança faz com que todo o código desenvolvido na classe original seja aplicado ao objeto criado da classe derivada. O segundo recurso é a **sobrescrita**, no qual um novo método é definido na classe derivada utilizando a mesma assinatura. A sobrescrita permite que métodos da superclasse sejam redefinidos (*especializados*) na classe derivada com novas funcionalidades. O programador deve atentar para manter o mesmo tipo de retorno. O terceiro recurso é a possibilidade de **estender** a definição de uma classe mais genérica (a superclasse) com novas funcionalidades especificas a nova classe criada.
- Atributos: Os atributos da(s) classe(s) original(is) sempre são herdados. Portanto, sempre ocuparão espaço de memória. Atributos não podem ser sobrescritos ou especializados. Note, no entanto, caso a classe derivada inclua um atributo com um mesmo identificador de atributo utilizado na classe derivada, cada atributo terá seu escopo restrito ao escopo do objeto que o define. Assim, o atributo definido na classe original somente poderá ser acessado quando forem executados métodos definidos e implementados nesta classe. Quando executados métodos definidos e implementados na classe derivada, o escopo válido é o mais interno. Assim, o identificador está se referindo ao atributo no contexto deste objeto.

Uma questão que deve ser bem compreendida é que a herança “incorpora” no corpo de um objeto o código da superclasse da classe da qual ele foi instanciado. Esta noção é importante pois explicita que o objeto criado a partir de uma classe derivada não é simplesmente um “objeto estendido com código de duas (ou mais) classes”. O escopo de cada classe continua preservado: um membro definido como privado na superclasse não será vizivel no escopo dos métodos implementados na classe derivada (esta discussão prossegue em Escopos de Visibilidade, abaixo). Também deve se observar que os membros (privados ou públicos) da classe derivada não são conhecidos pela sua superclasse. Isto é fácil de compreender, tendo em vista a organização hierárquica imposta pelo mecanismo de herança.

Outro aspecto importante: os construtores da superclasse não são executados implicitamente na construção de objetos das suas classes derivadas. Caso necessário, o programador deve introduzir a invocação explícita do construtor desejado da superclasse. Este invocação se dá através do uso da referência ao objeto da superclasse: `super`. Importante: caso a invocação ao construtor da superclasse deve ser realizada, o código correspondende deve se encontrar, obrigatoriamente, na primeira linha do construtor da classe derivada.

`this` e `super`

No escopo de um objeto existem duas referências implícitas: `this` e `super`. Enquanto a primeira diz respeito a uma referência ao próprio objeto, super é uma referência ao código instanciado para a componente do objeto representando o código da superclasse corresponde (caso exista uma superclasse). Desta forma, super pode ser utilizado para acessar os membros (publicos e/ou protegidos) da superclasse, permitindo, por exemplo, resolver questões de duplo emprego de nomes de membros.

Exemplo:

Implemente duas classes: `Carro` e `Carrao`. Ambas classes devem suportar operações para abastecimento do veículo e deslocamento. No entanto, objetos da classe `Carrao` possuem ar-condicionado, que pode ser ligado ou desligado através de métodos próprios. Quando o ar estiver ligado, o consumo de combustível aumenta em 10%. Implemente estas duas classes utilizando uma estrutura baseada em herança.

```
class Carro {
   private double comb, // total de combustivel no tanque
                  cons; // consumo de combustivel por quilometro
   private int    cont; // contador de quilometragem percorrida

   public Carro() {
     comb = 0;          // carro de tanque vazio
     cont = 0;          // carro zero quilometro
     cons = 10;         // consome 1 litro de gasolina a cada 10 km
   }

   public Carro( double c ) { // recebe quantidade de combustivel inicial
     if( c > 55 ) comb = 55;  // o reservatorio do tanque e' limitado em 55 litros
     else  comb = c;    // primeiro abastecimento realizado
     cont = 0;          // carro zero quilometro
     cons = 10;         // consome 1 litro de gasolina a cada 10 km
   }

   public void setConsumo( double c ) {
     cons = c;
   }

   public double getConsumo() {
     return cons;
   }

   public boolean abastece( double c ) {
     if( (c+comb) > 55 ) return false; // quantidade de combustivel cabe no tanque?
     comb += c;         // caso caiba, abastece
     return true;
   }

   public boolean anda( int q ) {
     if( (comb/cons) > q ) return false; // nao ha combustivel suficiente para andar q quilometros
     comb -= comb/cons;  // consome combustivel
     cont += q;          // atualiza contador de quilometros
     return true;        // o carro andou
   }
   public String status() {
     String s = new String("Combustivel: " + comb + "\nQuilometros percorridos: " + cont +
                           "\nPode andar: " + cons*comb + " quilometros com o combustivel disponivel\n");
   }

  public void status() {
    System.out.println("Eu tenho um carro.");
  }
}

Carrao extends Carro {
  private boolean ar;       // ar-condicionado: true para ligado, false para desligado

  public Carrao() {
    super();
    ar = false;
  }

  public Carrao( double c ) {
    super( c );
  }
```

```
  public void ligaAr() {
    double novocons;

    if( ar == false ) {
      novocons = getConsumo() * 1.1; // calcula mais 10% do consumo padrao
      setConsumo( novocons );        // seta novo consumo
      ar = true;
    }
  }

  public void desligaAr() {
    double novocons;

    if( ar == true ) {
      novocons = getConsumo() / 1.1; // retorna ao consumo padrao
      setConsumo( novocons );        // seta novo consumo
      ar = false;
    }
  }

  public void status() {
    System.out.println("Eu tenho um carrao!!!!");
  }
}
```

Escopo de visibilidade

Importante ressaltar que membros privados em uma classe continuam sendo privados a objetos de outras classes. Assim, sendo criado um objeto de uma classe derivada, os métodos deste objeto não possuem acesso aos membros privados de sua superclasse. Veja no exemplo anterior a implementação do método `ligaAr()`. Este método não acessa diretamente o atributo privado `cons`. O acesso é realizado através de métodos definidos na interface da classe `Carro`. A justificativa desta restrição de visibilidade de escopo está associada ao encapsulamento de dados: o escopo de visibilidade um membro privado deve ser restrita ao objeto que o contém.

Como não é interessante tornar membros públicos quando estes não devam fazer parte da interface de serviços dos objetos. É possível, no entanto, relaxar esta restrição, permitindo que membros (atributos ou métodos) não pertençam a interface de serviços mas mesmo assim fiquem disponíveis para as classes derivadas. O uso do modificador `protected` permite definir este relaxamento. Um membro definido como `protected` não pertence a interface de serviços dos objetos, mas tem seu escopo de visibilidade em todas as classes derivadas da classe onde foi definido.

Exercícios:

1. Considere os métodos (lembre, os construtores também são métodos) abaixo. Indique se está sendo utilizada sobrecarga de método ou sobrescrita. Explique e diga com que outro método esta sendo realizada a sobrecarga ou sobrescrita (se for o caso).

```
public class A {
   ...
   public A() { ... }
   public A( int x ) { ... }
   public void m1() { ... }
   public void m1( int h ) { ... }
}
```

```
public class B extends A {
   ...
   public B() { ... }
   public void m1() { ... }
   public void m1( double x, double y ) { ... }
   public void m2() { ... }
}
```

2. Implemente uma classe genérica para reagrupar as partes comuns das classes Fatorial e Finonacci apresentadas como primeiro exemplo. Reimplemente estas classes.
3. Implemente um novo tipo de veículo: o jipe, o qual não possui ar-condicionado, mas pode ter acionado a tração em quatro rodas. A qual implica em aumentar o consumo de combustível em 10%.

4. Defina a estrutura de classes para representar contas em banco: conta genérica, conta corrente comum, conta corrente especial e conta-poupança. Represente o esquema de relacionamento entre classes e realize sua implementação.
5. Considere as definições abaixo.
   - `Carro c1;`
   - `Carrão c2;`

   As seguintes instanciações de objeto estão corretas ? Sim/Não, Justifique.

`c1 = new Carro();` `c1 = new Carrao();`	`c2 = new Carro()` `c2 = new Carrao()`

   E as seguintes invocações de métodos funcionam? Sim/Não, Justifique. Caso funcionem, qual método de qual classe será executado?

`c1.anda(10);` `c2.anda(10);`	`c1.status();` `c2.status();`	`c1.ligaAr();` `c2.ligaAr();`

6. **Trabalho opcional: Vale 1 ponto na Prova 1. Entregar no dia da prova.**

   1a parte:

   Implementar uma classe para armazenamento de números inteiros (`int`) – em geral, um objeto que armazena uma coleção de dados é chamada de *container*. Na construção de um objeto desta classe deve ser informado o número máximo de inteiros que serão armazenados. Objetos desta classe devem ainda oferecer as operações:

   - `boolean insere( int num )`: insere o valor `num` no container.
   - `boolean insere( int num, int p )`: insere o valor `num` na `p`-ésima posição do container.
   - boolean put`(int num, int p)`: atribui o valor `num` à `p`-ésima posição do container.
   - `boolean retira( int p )`: retira o elemento que esta na posição `p`-ésimo elemento da lista.
   - `boolean deleta( int num )`: retira a primeira ocorrência de `num` do container.
   - `int get( int p )`: retorna o valor do elemento que esta na `p`-ésima posição do container.
   - `int length()`: retorna o tamanho ocupado do container.
   - `int maxSize()`: retorna o tamanho máximo do container.
   - `void list()`: imprime (na tela) todos os container.
   - `int first()`: retorna o primeiro elemento do container.
   - `int last()`: retorna o último elemento do container.
   - `void sort()`: ordena o container.

   2a parte:

   Implementar uma estrutura de generalização/especialização permitindo a ordenação do container segundo diferentes estratégias de ordenação. Entre eles, no mínimo, implemente:

   - Merge Sort
   - Insertion Sort
   - Bubble sort

   Observações:

   - Outros métodos/atributos podem ser necessários. Implemente-os conforme a necessidade observada.
   - Os métodos que retornam boolean devem retornar true para uma execução bem sucedida e false caso a operação não tenha sido realizada (ex.: *overflow*).
   - A implementação deverá ser defendida pelo seu autor – o ponto extra é condicionado a avaliação desta defesa.
   - Pode ser realizado em grupo, no caso, a nota do trabalho será de 2 pontos, que será dividida entre todos os participantes do grupo. Um participante do grupo será sorteado para defender a implementação realizada – a defesa do programa valerá para todo grupo.