POO - Java

Unidade 2 – Parte 1

PROGRAMAÇÃO ORIENTADA A OBJETOS COM JAVA

Modelagem orientada a objetos

Abstração

O mecanismo básico utilizado para realização da análise do domínio da aplicação é a abstração, através da qual um indivíduo observa a realidade (domínio) e procura capturar sua estrutura (abstrair entidades, ações, relacionamento, etc, que forem consideradas relevantes para a descrição deste domínio). O resultado deste processo de abstração é conhecido como Modelo Conceitual.

Classes

Uma definição de classe é basicamente uma definição de um tipo de dado. Uma classe contém um conjunto de bits representando um estado e um conjunto de operações que permitem modificar o estado. Em geral, as operações são públicas e os dados internos da classe são privados -- as únicas modificações possíveis nos dados são realizadas através das operações que a classe deixa disponível para que o público use.

A informação que permanece escondida do usuário da classe está contida na seção privativa da classe. De fora da classe, é como se aqueles dados não existissem: eles não podem ser acessados ou modificados diretamente.

Objetos

Considere um domínio de aplicação específico como uma casa típica. Sob certo sentido ela pode ser vista como uma composição de entidades tais como:

João	o pai
Maria	a mãe
Pedro	o filho
Xpit	um Cachorro
Xbeta	uma Cadela

Nela há, também, vários outros objetos como:

cômodos	salas, quartos, banheiros...
Móveis	mesas, cadeiras...
Louças	xícaras, pratos, talheres...
decorações	quadros, tapetes...

Um objeto é cada uma das entidades identificáveis num dado domínio de aplicação. Alguns destes objetos são **objetos concretos**, a exemplo dos citados acima.

Distinção entre Classes e Objetos

Note que em frases como "*Xpit está lá no jardim brincando com Pedro*" há referências a objetos específicos. Neste caso não há dúvidas sobre a identidade dos objetos que estão sendo referenciados: o cão de nome Xpit e a criança de nome Pedro.
Uma segunda forma de se referenciar objetos pode ser obtida por construções como: "*o cão é amigo do homem*". Neste caso as palavras *cão* e *homem* não referenciam nenhum objeto específico como no caso anterior. Estas palavras estão sendo usadas para referenciar

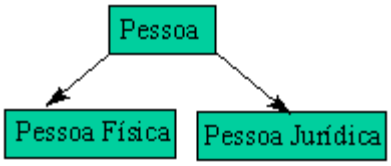
aqueles objetos que possuem características de cães e homens, respectivamente, ou seja, aqueles objetos que possam ser de alguma forma identificados como sendo um cão ou um homem.

Ações (Mensagens) - Encapsulamento / Ocultamento de Informações

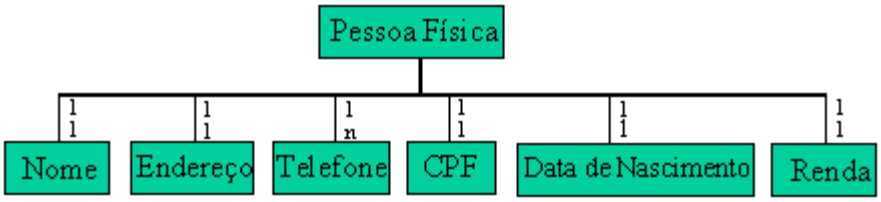
Um objeto pode ser descrito pela identificação de dois elementos básicos: sua estrutura e seu comportamento. A estrutura de um objeto é hierarquia de agregação/decomposição. Já seu comportamento pode ser descrito por identificação das ações a ele inerentes.
Considere um videocassete típico.
Quando visto sob a perspectiva de sua estrutura, nota-se que ele é composto por um número bastante grande de componentes eletromecânicos, os quais compõem seus mecanismos básicos de: tração da fita, gravação/reprodução de sinal, conexão com outros dispositivos, etc.
Note, contudo, que a estrutura interna do videocassete está protegida por uma carcaça. Este **encapsulamento** previne manipulações incorretas do equipamento, propiciando uma maior garantia da integridade interna do videocassete. Desde que o lacre de garantia não seja rompido, o fabricante garante seu correto funcionamento em condições normais de uso.
Este encapsulamento induz, também, um certo **ocultamento** da estrutura interna do videocassete. Em realidade boa parte dos usuários não faz idéia de como é realmente um videocassete ou de quais são os processos que estão envolvidos quando da reprodução de uma fita. Mas isto não os impede de utilizar o equipamento em sua plena funcionalidade, bastando apenas que ele saiba interagir com sua interface externa (botões e etc). O conhecimento da estrutura interna só se faz necessário quando se for proceder a reparos, ou nos casos em que explicitamente se deseje desvendar o equipamento.

Herança

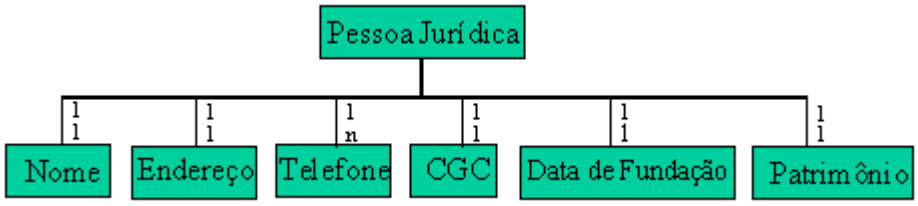
Para exemplificar o conceito de herança usaremos a classe Pessoa. Sob o ponto de vista administrativo (no contexto de um banco, por exemplo) a noção de *Pessoa* geralmente é subdividida (subclassificada) em: *Pessoa Física* (um ente individual) e *Pessoa Jurídica* (uma empresa ou sociedade).



Uma possível descrição da estrutura (hierarquia de agregação/decomposição) de cada uma destas classes poderia ser:



e



Note que as três propriedades mais à esquerda (Nome, Endereço e Telefone) são comuns às duas classes. Como ambas as classes são subclasses da classe *Pessoa*, estas propriedades podem ser mais bem descritas diretamente na classe *Pessoa*, evitando, desta forma, a repetição observada acima.

O efeito final é o mesmo, ou seja, os objetos pertencentes às classes *PessoaFísica* e *PessoaJurídica* terão as seis propriedades que lhe são características (as três comuns, mais as três particulares). Isto ocorre em função das classes *PessoaFísica* e *PessoaJurídica* serem especializações da classe *Pessoa* e, portanto, **herdarem** as suas propriedades.

Relação entre classes

Além da herança, outra relação comum entre classes é o "*uso*". A relação de uso é a mais óbvia e também a mais genérica. Por exemplo, uma classe *Pedido* usa a classe *Conta*, já que os objetos de *Pedido* precisam acessar os objetos *Conta* para verificar o crédito.

Ou seja, uma classe *A* usa uma classe *B* se:
Um método de *A* envia uma mensagem para um objeto da classe *B*, ou;
Um método de *A* cria, recebe ou retorna objetos da classe *B*.

Modificadores

Para modificar o modo como são declaradas classes, métodos e variáveis há algumas palavras-chave. Essas palavras-chaves são os modificadores.
Variáveis de instância são campos do objeto, declarados fora de qualquer método e inicializados quando é instanciado um objeto da classe. Também são chamadas de campos, propriedades ou atributos. Variáveis locais são as declaradas dentro de um método. Seu ciclo de vida está relacionado ao seu escopo (no máximo o do método) e elas sempre ficam no stack. Antes de tentar utilizá-las é necessário inicializá-las explicitamente, em uma parte atingível do código (fora de um if, por exemplo), senão o compilador retornará um erro. Variáveis locais também são chamadas de variáveis de stack, temporarias, automáticas ou de método.

Os modificadores de acesso aplicáveis são:
public: qualquer classe pode acessar
private: não pode ser acessado por nenhuma outra classe. Como não é visto pelas subclasses, não aplica-se regras de overriding.
protected: acessível por classes do mesmo pacote ou através de herança. Os membros herdados não são acessíveis a outras classes fora do pacote em que foram declarados.

O modificador final impossiblita que a variável seja reinicializada com outro valor. Se a variável for de instância, é necessário fornecer um valor na hora de declará-la ou em todos os construtores da classe, caso contrário ocorrerá um erro de compilação. Variáveis de interface sempre são final e precisam ter seu valor declarado.

Variáveis de Classe (static)

Uma *variável de instância* é uma variável cujo valor é específico ao objeto e não à classe. Uma variável de instância em geral possui um valor diferente em cada objeto membro da classe.

Uma *variável de classe* é uma variável cujo valor é comum a todos os objetos membros da classe. Mudar o valor de uma variável de classe em um objeto membro automaticamente muda o valor para todos os objetos membros.

Uma variável é considerada como de instância por "default". Para declarar uma variável de classe, acrescenta-se a palavra-chave static.

Exemplos:

```
static int nuhmeroDeInstahnciasDestaClasse;
static int LEFT = 1;
```

Outro exemplo: A variável static serve para contar o número de objetos robôs instanciados.

```
//Classe Robo
class Robo
{
public int x;
public int y;
public static int quantos; //quantos foram instanciados
public Robo(int ax,int ay)
{
x=ax; y=ay;
quantos++;
```

```
}
}

//Classe principal, Arquivo Principal.java
class Principal
{
public static void main(String args[])
{
Robo.quantos=0; //inicializando a variavel static
Robo cnc,cnc2;
System.out.println(Robo.quantos);
cnc=new Robo(10,12);
System.out.println(Robo.quantos);
cnc2=new Robo(11,12);
System.out.println(Robo.quantos);
} //main method
} //class Principal
```

Polimorfismo

O polimorfismo é uma das características marcantes de linguagens orientadas a objetos. Ela permite que a mesma mensagem enviada a diferentes objetos resulte em um comportamento que depende da natureza dos objetos recebendo a mensagem, ou seja, permite a construção de abstrações que operem uniformemente sob uma família de tipos relacionados.

Quando uma mensagem é enviada para um objeto de uma subclasse em Java:

A subclasse checa se ela tem um método aquele nome e com exatamente os mesmos parâmetros. Se a resposta for positiva, ela usa esse método.

Caso contrário, Java envia a mensagem à classe imediatamente superior.

Se toda a hierarquia for caminhada sem que um método apropriado seja encontrado um erro em tempo de compilação ocorre.
A amarração dinâmica é a chave para o polimorfismo. O compilador não gera código em tempo de compilação para uma chamada de método, ele gera código para determinar que método chamar em tempo de execução usando a informação de tipo disponível para o objeto.

Interfaces

Interfaces podem ser vistas como projetos de classes. Desta forma interfaces apenas definem os métodos ou operações permitidas, sem a preocupação de como estes serão implementados. Quando se cria uma classe, especifica-se as propriedades assim como os métodos e suas implementações.
Exemplo:

```
interface NomeInterface
{
        declaração dos cabeçalhos dos métodos.
}
```

A implementação dos métodos de uma interface pode ter diferentes abordagens, de acordo com as necessidades do programador.
Classes podem implementar mais que uma interface. Contudo, uma classe pode estender somente uma classe.
Interfaces permitem múltipla herança, pelo fato de se poder implementar múltiplas interfaces.

Recomendações para o Projeto de Classes

Algumas recomendações são definidas para que suas classes sejam bem aceitas nos círculos de POO:

Sempre inicialize os dados;
Nunca use tipos de dados em demasia numa classe;
Nem todos os campos precisam de “acessadores” e “modificadores” de campos individuais;
Use uma forma padrão de definição de classes;
Divida classe com tarefas demais;
Dê nome às classes e métodos que representem suas tarefas.

Programação orientada a objetos com JAVA

Declaração de classes

A forma geral da declaração de uma classe é a seguinte:

[modificadores] class [nome classe] extends [nome super]
implements [nome interface]

Como podemos notar, há quatro diferentes propriedades de uma classe definidas por essa declaração:
Modificadores
Nome de classe
Super classes
Interfaces

Ex:

```
class ComputadorSeguro extends Computador {
  private boolean executando = true;
  public void Desligar() {
    if ( executando )
```

```
        System.out.println("Há programas rodando. Não desligue!");
      else
        ligado = false;
    }
}
```

Construtores

Um construtor é um método especial, definido para cada classe. O corpo desse método determina as atividades associadas a inicialização de cada objeto criado. Assim, o construtor é apenas invocado no momento da criação do objeto através do operador *new*.

Ex:

```
class Pessoa {
        public String Nome;
        public String Telefone;

        Pessoa()
        { /* construtor default */
                Nome = "";
                Telefone = "";
        }

        Pessoa(String N, String T)
        {
                Nome = N;
                Telefone = T;
        }
}
```

A sobrecarga de construtores também é possível em Java. Ela ocorre se vários métodos tem o mesmo nome mas com argumentos diferentes. O interpretador Java decide qual método chamar.

Instância da Classe / Objetos

O exemplo abaixo exemplifica a instanciação de uma classe no Java. A classe “Pessoa” é instanciada na variável “p” através da palavra “new’. A partir daí todos os métodos e atributos públicos podem ser acessados.

```
class chamaClasse
{
        public static void main(String[] args)
        {
                Pessoa p = new Pessoa("Luis","2212121");
                System.out.println("Nome:"+p.Nome);
                System.out.println("Fone:"+p.Telefone);
        }
}
```

Definição de Métodos / Controle de Acesso

Em Java, métodos são definidos usando uma aproximação que é muito similar à usada em outras linguagens, como por exemplo C e C++. A declaração é feita da seguinte forma:

<modifiers> <tipo de retorno> <nome> (< lista de argumentos >) <bloco>

< modifiers > : segmento que possuem os diferentes tipos de modificações incluindo public, protected e private.

< tipo de retorno > : indica o tipo de retorno do método.

< nome > : nome que identifica o método.

< lista de argumentos > : todos os valores que serão passados como argumentos.

public void adicionaDias (int dias)

Ações (Mensagens)

Métodos são a forma com que os objetos interagem entre si. Na orientação a objetos, para se atingir encapsulamento e proteção, todas as interações entre objetos (ditas mensagens) devem ser feitas através dos métodos de uma classe. Não há funções e procedimentos em Java, haja vista que a programação procedural foi totalmente substituída. Mesmo assim, a forma de se invocar métodos em Java se assemelha à de funções em C e C++. Qualquer método pertence a uma classe, e deve ser definido dentro do corpo da mesma.
Exemplo:

```
class Pessoa {

        private String Nome; /* variável de instância */
        private String Telefone; /* variável de instância */

        Pessoa() { /* construtor default */
                Nome = "";
                Telefone = "";
                 }

        Pessoa(String N, String T) {
                Nome = N
                Telefone = T;
                }

        public void mudaNome(String Nome) {
                this.Nome = Nome;
        }

        public String leNome() {
                return Nome;
        }

        public String leTelefone() {
                return Telefone;
        }
}
```

Para invocar um método é necessário instanciar a classe e então chamar o método através do objeto.

```
Pessoa p = new Pessoa();
p.mudaNome(“Luis”);
```

Herança

Abaixo um exemplo de herança utilizando a linguagem Java.

```
class Funcionario extends Pessoa {
        // Novos métodos e instâncias podem ser inseridos aqui
}

class Secretaria extends Funcionario {
        ...
}

class Gerente extends Funcionario {
        ...
}
```

A palavra chave extends diz que Funcionario é uma subclasse de Pessoa, chamada de classe base.
Dizemos que Funcionario herda as características de Pessoa, já que este reutilizará boa parte do código e funcionalidade descrita na classe Pessoa.

A Referência Super

A palavra-chave super sempre se refere a uma superclasse. Por exemplo:

```
class empregador
{
        private String nome;
        empregador (String n)
```

```
{
        nome  = n;
}
}

class gerente extends empregador
{
        private String secretaria;
        gerente(String n, String sec)
{
        super (n);
        secretaria = sec;
}
}
```

Evitando herança (Final)

Pode-se impedir que uma classe seja derivada a partir de outras. As classes que não podem ser classes mãe são chamadas de classes finais e usa-se o modificador final na definição da classe para indicar isso.

```
final class novaclasse
}
...
}
```

Pode-se também fazer final um método específico de uma classe. Se isso for feito então nenhuma subclasse poderá sobrepor ou substituir esse método (Todos os métodos de uma classe final são automaticamente “finais”).

A Referência This

É aplicado a métodos não estáticos.

O Java associa automaticamente a todas as variáveis e métodos referenciados com a palavra this. Por isso, na maioria dos casos torna-se redundante o uso em todas as variáveis da palavra this.

Existem casos em se faz necessário o uso da palavra this. Por exemplo, você pode necessitar chamar apenas uma parte do método passando uma instância do argumento do objeto. (Chamar uma classe de forma localizada);

Birthday bDay = new Birthday(this);

Polimorfismo

Abaixo se demonstra um exemplo de polimorfismo utilizando a linguagem Java. Foi definida uma classe “Mamífero” e cria-se duas classes (“Rato” e “Elefante”) estendidas da classe “Mamífero”.

```
abstract class Mamifero
{
   public abstract double obterCotaDiariaDeLeite();
} // class Mamifero

class Elefante extends Mamifero
{
   public double obterCotaDiariaDeLeite()
   {
      return 20.0;
   }
} // class Elefante

class Rato extends Mamifero
{
   public double obterCotaDiariaDeLeite()
```

```
    {
        return 0.5;
    }
} // class Rato

class Aplicativo
{
    public static void main(String args[])
    {
        System.out.println("Hierarquia e polimorfismo\n");
        Mamifero mamifero1 = new Elefante();
        System.out.println("Cota diária de leite do elefante: " +
mamifero1.obterCotaDiariaDeLeite());
        Mamifero mamifero2 = new Rato();
        System.out.println("Cota diária de leite do rato: " +
mamifero2.obterCotaDiariaDeLeite());
    }
} // class Aplicativo
```

Interfaces

Abaixo se demonstra um exemplo da criação de interfaces utilizando a linguagem Java. Uma interface “Relógio” foi definida e duas classes “RelogioAnalogico” e “RelogioDigital” foram implementadas a partir dela.

```
interface Relógio {
        void MostraHora();
        void MostraDia();
        void AjustaHora();
        void AjustaDia();
}
```

```
class RelogioAnalogico implements Relogio {
        hora HoraAtual;
        dia DiaAtual;

void MostraHora() {
        ...movimenta ponteiros...
        return;

void MostraDia() {
        ...
        return;
}

void AjustaHora() {
        ...
        return;
}

void AjustaDia() {
        ...
        return;
}
}
```

```
class RelogioDigital implements Relogio {
        hora HoraAtual;
        dia DiaAtual;
        display Display;

void MostraHora() {
        Display = HoraAtual;
        return;

void MostraDia() {
        Display = DiaAtual;
        return;
}

void AjustaHora() {
        ...
        return;
}

void AjustaDia() {
        ...
        return;
}
}
```

Modelo de Classes (abstract)

Ao subir na hierarquia de heranças as classes tornam-se mais genéricas e provavelmente mais abstratas, tornando-se tão geral que acaba sendo vista mais como um modelo de classe do que uma classe com instâncias.

```
abstract class nomeclasse
}
...
}
```

Um método abstrato promete que todos os descendentes não abstratos dessa classe abstrata irão implementar esse método abstrato. Os métodos abstratos funcionam como uma espécie de guardador de lugar para métodos que serão posteriormente implementados nas

subclasses. Uma classe com um ou mais métodos abstratos precisa, ela mesma ser declarada abstrata.