Programação Orientada a Objetos
Prof. Rone Ilídio – UFSJ/CAP

1) Introdução

Programação Orientada a Objetos é um paradigma de programação bastante antigo. Entretanto somente nos últimos anos foi aceito realmente pelas grandes empresas de desenvolvimento de software. Alguns exemplos de linguagens modernas utilizadas por grandes empresas em todo o mundo que adotaram essas idéias são: Java, C#, Object Pascal (Delphi), PHP, Visual Basic, entre outras. Algumas linguagens adotam os conceitos parcialmente, como por exemplo o C++. Nesta linguagem, o programador pode criar programas na forma **procedural** tradicional ou orientada a objetos. Entretanto, as linguagens mais modernas são totalmente orientadas a objetos, como é o caso do Java.

Os conceitos de orientação a objetos, além de serem utilizados em linguagens de programação, são utilizados por linguagens para projeto e modelagem de software e para construção de sistemas de gerencia de bancos de dados. A UML (*Unified Modeling Language*) é uma linguagem destinada ao projeto de sistemas. Toda a estrutura desta linguagem é baseada nos conceitos de orientação a objetos. Ela é formada por diagramas, sendo que um de seus principais é o Diagrama de Classe que é totalmente orientado a objetos. Nos últimos anos, um novo conceito de banco de dados surgiu, o qual é chamado de banco de dados orientado a objetos. Nesse tipo de banco de dados, a estrutura de armazenamento dos dados é definida tomando-se como base os conceitos de orientação a objetos. Bancos de dados bem conceituados como Oracle e Postgree adotaram essa tendência e lançaram versões orientadas a objetos. A seguir, são apresentados os principais conceitos de orientação a objetos.

1.1) Orientação a Objetos

Quando surgiram os primeiros computadores, a preocupação dos programadores era buscar a maior eficiência com pouco uso de memória, devido às limitações do hardware da época. Com a evolução do hardware, mais memória e maior poder de processamento, a preocupação voltou-se para a eficiência do desenvolvimento, ou seja, busca-se diminuir o tempo necessário para o desenvolvimento de programas. Por isso, foram elaboradas varias técnicas e metodologias que utilizam a estratégia de "dividir para conquistar". Tal estratégia implica em resolver um grande problema dividindo-o em vários problemas menores. A programação orientada a objetos auxilia essa divisão uma fez que facilita a divisão de um programa em módulos, cada um com tarefas bem específicas. Além disso, programação orientada a objetos facilita a reutilização de código uma vez que os módulos definidos para um programa podem ser utilizados em outros programas similares.

A programação orientada a objetos foi criada para tentar aproximar o mundo real do mundo virtual. Buscou-se simular o mundo real dentro do computador. Para isso, nada mais natural do que criar “objetos” nos computadores, que representam a abstração de objetos reais. Ao utilizar uma linguagem orientada a objetos, o programador é responsável por definir quais informações e ações são importantes para cada objeto, além de definir como os objetos interagem entre si. Com isso, cada objeto fica “responsável” em resolver os problemas referentes a ele, ou seja, a técnica dividir para conquistar é empregada na definição dos objetos.

Vamos a um exemplo prático: imagine que você está desenvolvendo um software para um supermercado e ele possui diversos clientes. Como estamos tentando modelar um sistema baseado no sistema real, nada mais obvio do que existirem objetos do tipo **Cliente** dentro do nosso programa. Tais objetos "simulam" no mundo virtual características e ações que um cliente pode realizar no mundo real. O mesmo pode acontecer para objetos do tipo Produto, Fornecedor, Empregado e vários outros.

2) Principais Conceitos de Orientação a Objetos

2.1) Objeto

Um objeto real é tudo aquilo que existe, como um cliente, um carro, um produto. Em termos computacionais, um objeto é a representação na memória do computador de um objeto real. O programador deve abstrair um objeto real para representá-lo computacionalmente. Com a definição eficiente de um conjunto de objetos no computador, os programas passam a ser formados por vários objetos que conversam entre si. Por isso, os programadores devem verificar como cada objeto trabalha e criar as regras para a iteração entre tais objetos.

Como exemplo, vamos utilizar um rádio. Existem várias coisas que você pode fazer com ele, como: ligar ou desligar, aumentar o volume, escolher uma estação. Entretanto, você não precisa entender como tudo funciona para poder executar essas atividades. Em POO, uma vez se que tenha um objeto criado corretamente, somente é necessário saber como ele funciona para poder utilizá-los, seus detalhes internos não são necessários.

2.2 Classes e Seu Componentes

Classe é uma estrutura que abstrai um conjunto de objetos com características similares. Para se modelar computacionalmente objetos reais, criam-se classes. Elas contêm a definição de todas as informações importantes para os objetos que estão sendo modelados e as ações que eles realizam. Nas classes, os comportamentos de seus objetos são descritos por métodos e os estados possíveis destes objetos (informações que eles podem salvar) são armazenadas por atributos. Em outros termos, uma classe descreve os serviços providos por seus objetos e quais informações eles podem armazenar.

Importante mencionar que **classes são tipos definidos pelo programador**. Quando um programa é executado, ele utiliza a classe para “dar vida" ao objeto. Em outras palavras, são criadas “variáveis” cujo tipo é uma classe. Tais variáveis são chamadas de **objetos**. Objetos possuem um nome (identificador da variável) e uma região de memória onde são armazenadas todas as suas informações e seu comportamento, ou seja, tudo aquilo que foi definido na classe. Em geral **"criamos classes"** e a partir delas **"instanciamos objetos"**. A criação de uma classe é feita em tempo de projeto, quando o programador está codificando o programa. O objeto é criado em tempo de execução, ou seja, quando é feita uma instância da classe é criado um objeto. A classe é como se fosse um molde, você constrói o molde e depois o utiliza para fazer diversos objetos.

Uma observação importante é que tecnicamente o verbo **instanciar** significa alocar uma área de memória com determinadas características. Quando se define um objeto como uma “instancia de uma classe” (termo normalmente utilizado na literatura) quer dizer que uma área de memória foi reservada para o programa e todas as características da classe foram para tal área.

Os **Atributos** são os elementos que definem a estrutura de uma classe. Os atributos também são conhecidos como variáveis de classe e podem ser divididos em dois tipos básicos: atributos de instância e de classe. Os valores dos atributos de instância determinam o estado de cada objeto, ou seja, para um dado atributo seu valor muda de objeto para objeto. Um atributo de classe possui um estado que é compartilhado por todos os objetos de uma classe, ou seja, se um objeto muda o valor de um atributo de classe, todos os objetos do mesmo tipo acessarão esse novo valor. Atributos de classe podem ser chamados também de atributos estáticos.

Métodos definem as habilidades dos objetos, são as funções contidas dentro das classes. Como exemplo, Rex é uma instância da classe Cachorro, portanto tem habilidade para latir, implementada através do método deUmLatido(). Um método em uma classe é apenas uma definição. A ação só ocorre quando o método é invocado através do objeto. Dentro do programa, a utilização de um método deve afetar apenas um objeto em particular. Todos os cachorros podem latir, mas você quer que apenas Rex dê o latido. Normalmente, uma classe possui diversos métodos, que no caso da classe Cachorro poderiam ser sente(), coma() e morda().

A Figura 1 exemplifica uma classe e suas características. São apresentados seus atributos e métodos. Para esse exemplo, foi utilizada a linguagem Java, a qual também será utilizada em todo restante do texto. Objetos do tipo Pessoa representarão no sistema pessoas reais. Nessa classe, pode-se verificar as seguinte características:

- Uma classe é semelhante a uma *struct* (C) ou *record* (Pascal), a diferença é que tais estruturas não contém métodos (ou funções) como existe na classe
- Os atributos são variáveis globais à classe.
- Os métodos são funções que normalmente manipulam os dados contidos nos atributos.
- Os métodos *get* e *set* são utilizados para leitura e escrita nos atributos (maiores detalhes em Encapsulamento).
- *Public* significa acesso externo e *Private* acesso interno. Maiores detalhes serão dados em Encapsulamento.

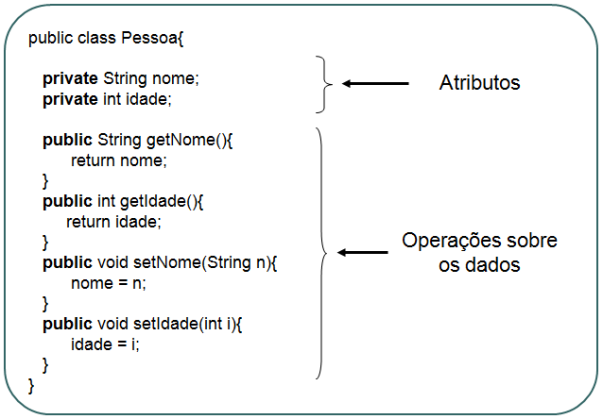


Figura 1: Definição de uma classe em Java.

A utilização de uma classe acontece como é exemplificado na Figura 2. Nela foi criado um aplicativo Java (que também é uma classe, mas que pode ser interpretada). Em seu método *main*, foi criada uma variável do tipo Pessoa (variável p). A instância da classe, ou seja, o objeto só é criado pelo operador *new*. Tal operador aloca uma área de memória com todas as características definidas na classe Pessoa (p=new Pessoa()). Após isso, o objeto pode receber valores (ex: p.setNome("José")) e tais valores podem ser lidos e exibidos na tela (ex: System.out.println("Nome="+ p.getNome());). A Figura 3 ilustra uma memória e a criação de um objeto do tipo Pessoa nela.

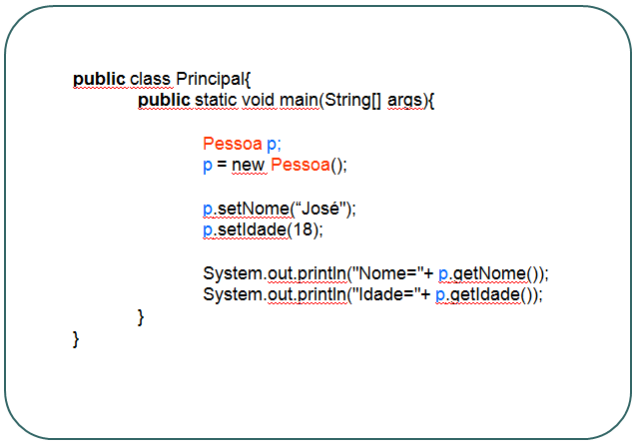


Figura 2: Aplicativo Java que cria um objeto do tipo Pessoa.

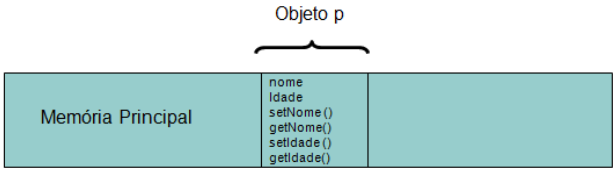


Figura 3: Ilustração de uma memória com criação de um objeto do tipo Pessoa.

Os métodos e atributos são denominados **membros da classe**. Ambos podem ser declarados como *static*. Métodos *static* podem ser chamados sem a criação de objetos da classe. Basta colocar o nome da classe, um ponto e o nome do método. Tais métodos só podem chamar métodos e variáveis que também sejam *static*. Os atributos *static* também podem ser chamados sem a criação de objetos da classe onde foram declarados. Uma característica importante desse tipo de atributo é que ele possui "escopo de classe", ou seja, um atributo *static* tem somente um valor para todos os objetos daquele tipo.

Um tipo diferente de método merece destaque, o **método Construtor**. Ele é chamado todas as vezes que se instancia um objeto de uma classe. Ou seja, quando o código "p=new Pessoa()" é executado, o new cria o objeto na memória e chama o métodos construtor (Pessoa()). Tal método tem o mesmo nome da classe e não possui tipo de retorno. Quando ele não é declarado, como é o caso da classe Pessoa, o compilador

“cria” um construtor que não possui tipo de retorno nem parâmetros. A Figura 4 ilustra o método construtor vazio que é considerado pelo compilador.

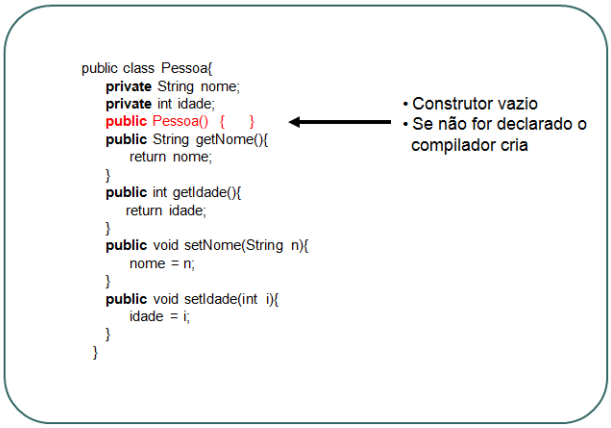


Figura 4: Método construtor quando não declarado, o compilador interpreta a existência de um construtor vazio.

Observação: na verdade, toda classe que não tem construtor herda um construtor vazio da classe Object. Isso será comentado posteriormente.

No exemplo da Figura 5 (A), a classe Pessoa foi alterada. Ela recebeu um construtor que recebe como parâmetros uma String e um inteiro, valores que serão passados para os atributos nome e idade, respectivamente. Com isso, a criação do objeto do tipo Pessoa muda, pois o construtor tem que obedecer ao que foi definido no construtor. A Figura 5 (B) ilustra como fica a criação de um objeto do tipo Pessoa com esse novo construtor.

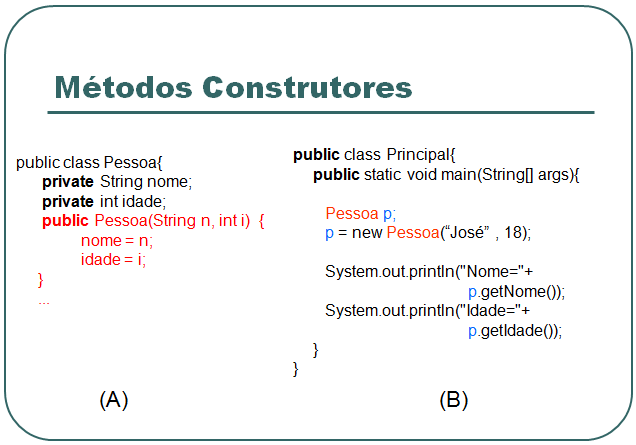


Figura 5: Criação de um objeto do tipo Pessoa com a criação de um construtor.

2.3) Herança

A **Herança** é um princípio da programação orientada a objetos que permite que as classes compartilhem atributos e operações baseados em um relacionamento, geralmente generalização. A herança permite a criação de subclasses que herdam alguns dos atributos e das operações (ou métodos) da classe mãe (super-classe ou classe base). A herança é um conceito aplicado no momento de criação das classes. Ela é usada na intenção de reaproveitar código ou comportamento generalizado ou especializar operações ou atributos.

Como exemplo, considere a classe Pessoa sem construtor (Figura 1). Pode-se observar as classes Aluno e Professor, onde ambas possuem atributos como nome e idade. Nesse caso, pode-se criar a classe Pessoa que contem as semelhanças entre as duas classes, fazendo com que as classes Aluno e Professor herdem as características de Pessoa. Desta maneira, pode-se dizer que Aluno e Professor são subclasses de Pessoa.

A Figura 6 exemplifica a criação da classe Aluno. A palavra **extends** define a relação de herança entre Pessoa e Aluno. Como pode ser visto, Aluno possui suas características próprias e recebe por herança as características de Pessoal. A Figura 7 exemplifica a criação de um objeto do tipo Aluno. Verifica-se que tanto as características definidas em Aluno, quanto a características definidas em Pessoa, podem ser utilizadas

nesse objeto.

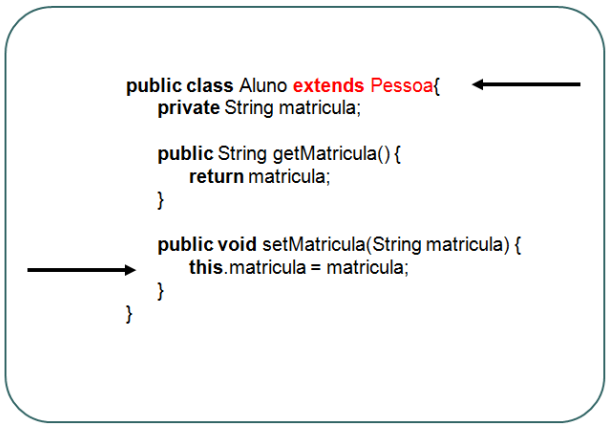


Figura 6: Classe Aluno filha de Pessoa.

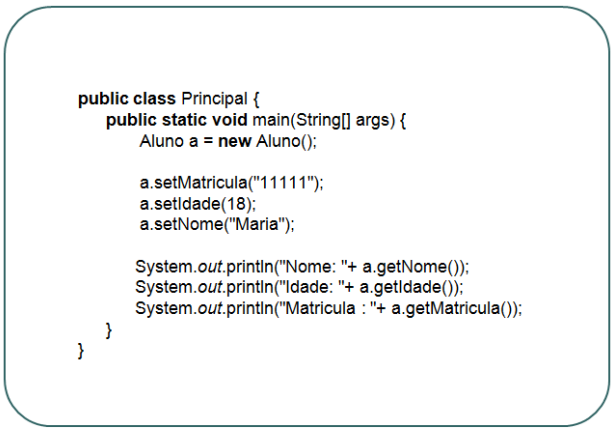


Figura 7: Aplicativo que cria um objeto do tipo Aluno.

Se considerarmos a classe Pessoa com construtor, suas filhas obrigatoriamente terão construtor cuja primeira linha deverá ser a chamada do construtor da mãe. Na Figura 8 temos a declaração da classe Pessoa com construtor. Na Figura 9, temos a declaração da classe Aluno com o construtor e a chamada para o construtor da mão. Essa chamada ocorre com a palavra reservada **super**. Essa palavra diz ao computador que o programador deseja acessar a mãe da classe em questão.

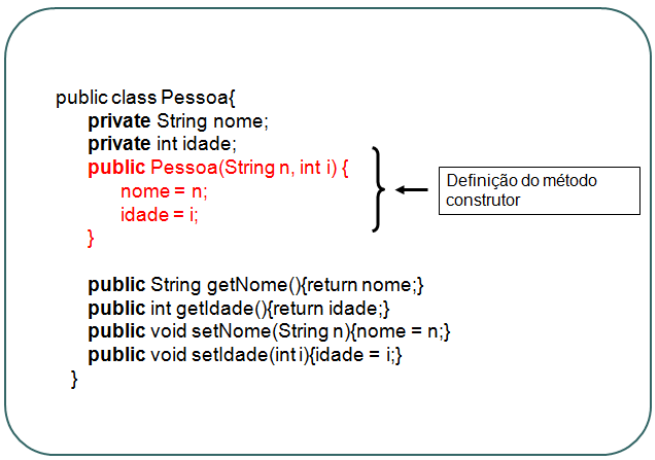


Figura 8: Classe Pessoa com construtor.

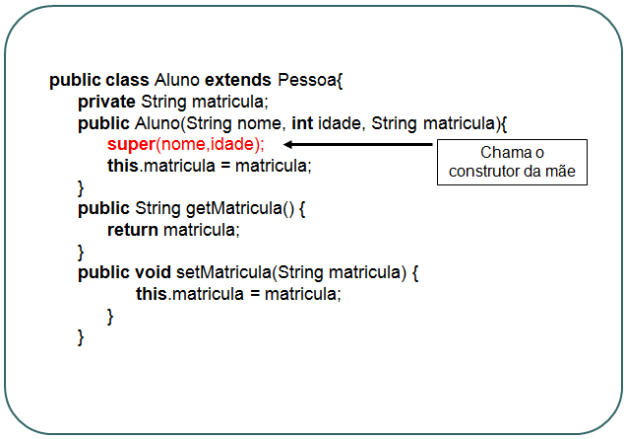


Figura 9: Classe Aluno com construtor e chamada do construtor da mãe (obrigatório).

Ao se criar uma classe que não possui uma mãe, o Java entende que ela é filha da classe Object. Com isso, ao se criar uma hierarquia de classes (criando-se uma classe mãe, uma filha, uma neta e assim por diante) a classe Object estará sempre no topo dessa hierarquia. Em outras palavras, todas as classes em Java são descendentes diretos ou indiretos da classe Object.

2.4) Encapsulamento

O termo encapsulamento vem de encapsular ou esconder. Ao utilizar tal conceito, separa-se o programa em partes, o mais independentes possível. A ideia é tornar o software mais flexível, fácil de modificar e de criar novas implementações. Para exemplificar, podemos pensar em uma dona de casa (usuário) utilizando um liquidificador (sistema). O usuário não necessita conhecer detalhes do funcionamento interno do sistema para poder utilizá-lo, precisa apenas conhecer a interface, no caso, os botões que controlam o liquidificador. Em programação, um programador não precisa saber os detalhes do funcionamento interno de um objeto, somente como utilizados. Por isso, as classes possuem métodos e atributos encapsulados, enquanto outros métodos e atributos ficam disponíveis para utilização. Isso garante o bom funcionamento dos objetos.

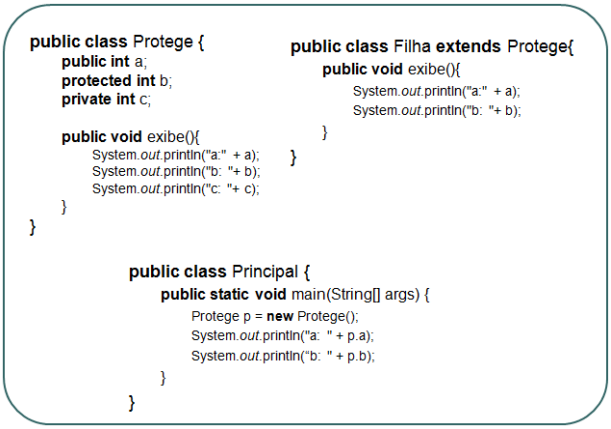
Uma grande vantagem do encapsulamento é que toda parte encapsulada pode ser modificada sem que os usuários da classe em questão sejam afetados. No exemplo do liquidificador, um técnico poderia substituir o motor do equipamento por outro totalmente diferente, sem que a dona de casa seja afetada - afinal, ela continuará somente tendo que pressionar o botão.

Encapsular atributos também auxilia a garantir que o estado e o comportamento de um objeto se mantenha coeso. Por exemplo, no caso da classe Semaforo poderíamos ter um método de acesso chamado lerCor(), e um modificador chamado proximaCor(). O estado é mantido pois os usuários da classe não podem alterar as cores de um semáforo ao seu bel prazer e, por exemplo, fazer a seguinte troca de cores: vermelho-amarelo-vermelho. É comum usar o padrão get<nomeDoAtributo> para o método que retorna o valor atual do atributo e set<nomeDoAtributo> para o método que modifica o valor de um atributo do objeto, como no exemplo abaixo: setComProtecao e getComProtecao.

A Figura 10 contém uma tabela com a síntese do encapsulamento em Java. A Figura 11 exemplifica as três formas de encapsulamento. A classe Protege contém todos os tipos de encapsulamento em seus atributos. Dentro de seu escopo, todos esses atributos podem ser utilizados. Somente os atributos *public* e *protected* podem ser acessados na classe Filha. No objeto criado na classe Principal, os atributos *public* podem ser acessados. Os atributos *protected* só serão acessados se as classes estiverem no mesmo pacote. (Observação: o tema “Pacotes” não será tratado neste texto.)

	Local	Filha	Objeto
public	Sim	Sim	Sim
protected	Sim	Sim	Sim/Não
private	Sim	Não	Não

Figura 10: Encapsulamento em Java.

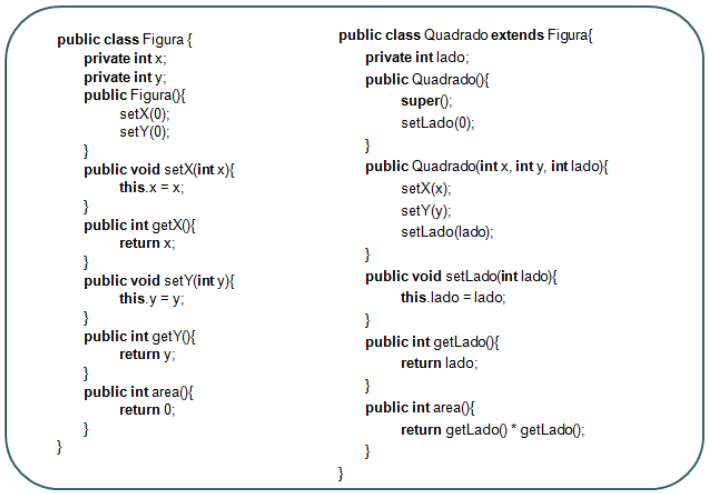


```
public class Protege {
    public int a;
    protected int b;
    private int c;

    public void exibe(){
        System.out.println("a:" + a);
        System.out.println("b: "+ b);
        System.out.println("c: "+ c);
    }
}

public class Filha extends Protege{
    public void exibe(){
        System.out.println("a:" + a);
        System.out.println("b: "+ b);
    }
}

public class Principal {
    public static void main(String[] args) {
        Protege p = new Protege();
        System.out.println("a: " + p.a);
        System.out.println("b: " + p.b);
    }
}
```

Figura 11: Exemplo de encapsulamento.

2.5) Sobrecarga de métodos

Sobrecarga de métodos ocorre quando temos dois métodos com o mesmo nome. Isso pode ocorrer de duas formas distintas. A primeira dentro de uma mesma classe. Nesse caso, o compilador obriga que tais métodos tenham listas de parâmetros diferentes. A segunda forma ocorre quando uma classe mãe contém um método com o mesmo nome do método da filha. Nesse caso, o método da filha é sempre chamado. A Figura 12 tem a declaração de duas classes Figura e Quadrado. Figura, a classe mãe, tem um método denominado area(). A classe Quadrado, filha de Figura, **sobrescreve** tal método, ou seja, cria um método com o mesmo nome. Em Quadrado também existe sobrecarga do método construtor, pois existem dois construtores, um com parâmetros e outro sem parâmetros. Com isso, Quadrado possui os dois exemplos de sobrecarga de métodos. Importante mencionar que não é somente o construtor que pode receber sobrecarga, qualquer método pode ser sobrecarregado.



```
public class Figura {
    private int x;
    private int y;
    public Figura(){
        setX(0);
        setY(0);
    }
    public void setX(int x){
        this.x = x;
    }
    public int getX(){
        return x;
    }
    public void setY(int y){
        this.y = y;
    }
    public int getY(){
        return y;
    }
    public int area(){
        return 0;
    }
}

public class Quadrado extends Figura{
    private int lado;
    public Quadrado(){
        super();
        setLado(0);
    }
    public Quadrado(int x, int y, int lado){
        setX(x);
        setY(y);
        setLado(lado);
    }
    public void setLado(int lado){
        this.lado = lado;
    }
    public int getLado(){
        return lado;
    }
    public int area(){
        return getLado() * getLado();
    }
}
```

Figura 12: Sobrecarga do método area() na classe filha e sobrecarga de construtor na classe Quadrado.

2.6) Polimorfismo

Permite que referências de tipos de classes mais abstratas representem o comportamento das classes concretas que referenciam. Assim, é possível tratar vários tipos de maneira homogênea (através da interface do tipo mais abstrato). O termo polimorfismo é originário do grego e significa "muitas formas" (poli = muitas, morphos = formas). Suponha a seguinte classe escrita em Java:

```
public abstract class OperacaoMatematica {
  public abstract double calcular(double x, double y);
}
```

Esta é uma classe abstrata que representa qualquer operação matemática. Podemos imaginar diversas operações que se encaixam na sua interface, como soma, subtração, multiplicação ou divisão, entre outras. Note que, mesmo que a natureza do cálculo mude, a semântica do método calcular não muda, ou seja, ele sempre calculará o resultado da operação matemática que está sendo trabalhada. Definamos então, duas subclasses, Soma e Subtracao, que implementam a classe OperacaoMatematica:

```
public class Soma extends OperacaoMatematica {
  public double calcular(double x, double y) {
    return x+y;
  }
}

public class Subtracao extends OperacaoMatematica {
  public double calcular(double x, double y) {
    return x-y;
  }
}
```

O seguinte trecho de código demonstra o uso do polimorfismo:

```
public class Contas {
  public static void mostrar(OperacaoMatematica operacao, double x, double y) {
     System.out.println("O resultado é: " + operacao.calcular(x, y));
  }

  public static void main(String args[]) {
      //Primeiro calculamos uma soma
      Contas.mostrar(new Soma(), 5, 5); //Imprime o resultado é: 10
      //Depois uma subtração
      Contas.mostrar(new Subtracao(), 5, 5); //Imprime o resultado é: 0
  }
}
```

Note que, embora o método calcular tenha sido chamado duas vezes no interior de mostrar, os tipos (isto é, as classes das instâncias) utilizados como parâmetros eram diferentes. De fato, o comportamento de cada tipo era exatamente oposto. É comum definir sobrecarga de métodos ou simplesmente sobrecarga como uma forma de polimorfismo (chamado de polimorfismo *ad-hoc*). Nesse caso, implementa-se métodos com um mesmo nome, mudando apenas a lista de parâmetros que ele recebe. Digamos

```
  public static void mostrarCalculo(Soma soma, double x, double y) {
     System.out.println("O resultado é: " + soma.calcular(x, y));
  }

  public static void mostrarCalculo(Subtracao subtracao, double x, double y) {
     System.out.println("O resultado é: " + subtracao.calcular(x, y));
  }
```

Embora nesse caso o resultado possa ser o mesmo que aquele obtido com o uso de herança, no polimorfismo *ad-hoc* não existem garantias que os métodos sobrecarregados tenham o mesmo comportamento.