Orientação a Objetos
Básica

Parte 04

Desenvolvimento de Programação Orientada a Objetos

Prof. Pedro Neto

Aracaju – Sergipe - 2011

Conteúdo

Orientação a Objetos Básica

i. Motivação
ii. Criando um tipo
iii. Uma Classe em Java
iv. Criando e usando um objeto
v. Métodos
vi. Método com retorno
vii. Objetos são acessados por referência
viii. Método transfere()
ix. Continuando com atributos
x. Para Saber mais: Uma fábrica de carros
xi. Um pouco mais...

Motivação e Vantagens

A **Orientação a Objetos** ajuda em muito em organizar e diminuir a quantidade de código, além de concentrar as responsabilidades nos pontos certos do programa, flexibilizando sua aplicação, **encapsulando** a lógica de negócios por exemplo. Outra enorme vantagem é o **polimorfismo** das referências, que veremos mais à frente.

Criando um Tipo

Considere um programa para um banco, é bem fácil perceber que uma entidade extremamente importante para o nosso sistema é a **Conta**. Nossa idéia aqui é generalizarmos alguma informação, juntamente com funcionalidades que toda conta deve ter.

O que toda conta tem e é importante para nós?

- número da conta
- nome do cliente
- saldo
- limite

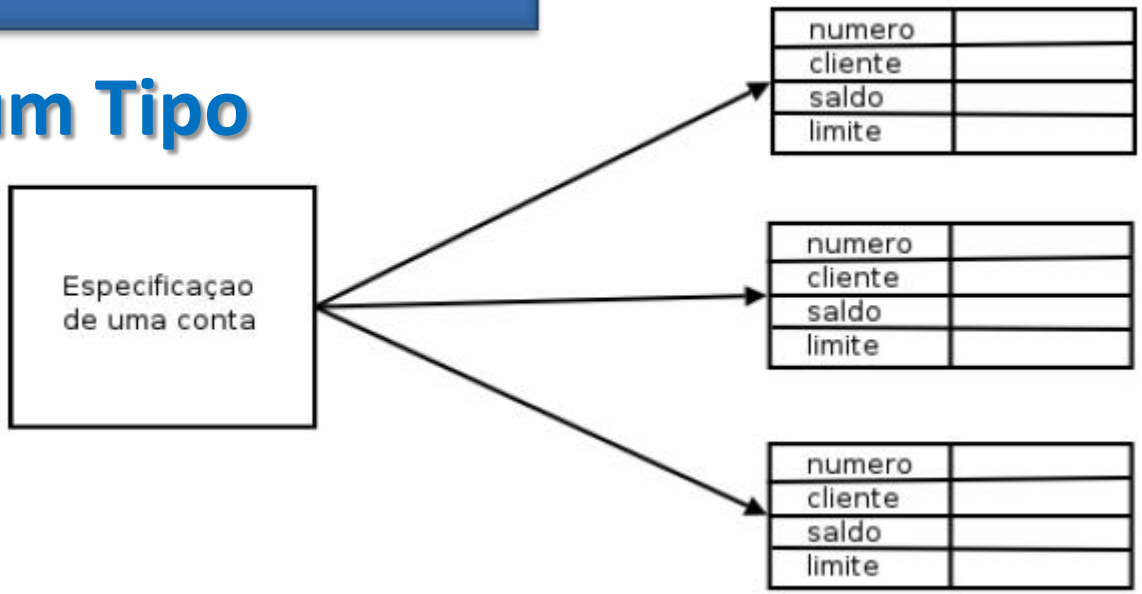
Criando um Tipo

Agora vamos definir o que toda conta faz e é importante para nós? Isto é, o que gostaríamos de "pedir a conta".

- saca uma quantidade x
- deposita uma quantidade x
- imprime o nome do dono da conta
- devolve o saldo atual
- transfere uma quantidade x para uma outra conta y
- devolve o tipo de conta

Com isso temos o projeto de uma conta bancária. Podemos pegar esse projeto e acessar seu saldo? Não. O que temos ainda é o projeto. Antes precisamos construir uma conta, para poder acessar o que ela tem, e pedir para ela fazer alguma coisa.

Criando um Tipo



Repare na figura: Apesar do papel do lado esquerdo estar especificado uma Conta, essa especificação é uma Conta? Nós depositamos e sacamos dinheiro desse papel? Não. Utilizamos a especificação da Conta para poder criar **instâncias**, que realmente são contas, onde podemos realizar as operações que criamos. Apesar de declararmos que toda conta tem um saldo, um número e uma agência no pedaço de papel (como a esquerda na figura), são nas instâncias desse projeto que realmente há espaço para armazenar esses valores.

Criando um Tipo

CLASSE

Ao projeto da conta, isto é, a definição da conta, damos o nome de **classe**. O que podemos construir a partir desse projeto, que são as contas de verdade, damos o nome de **objetos**.

A palavra classe vem da taxonomia da biologia. Todos os seres vivos de uma mesma classe biológica tem uma série de atributos e comportamentos em comuns, mas não são iguais, podem variar nos valores desses atributos e como realizam esses comportamentos. Homo Sapiens define um grupo de seres que possuem características em comuns, porém a definição (a idéia, o conceito) de um Homo Sapiens é um ser humano? Não. Tudo esta especificado na classe Homo Sapiens, mas se quisermos mandar alguém correr, comer, pular, precisaremos de uma instância de Homo Sapiens, ou então de um objeto do tipo Homo Sapiens.

Uma Classe em Java

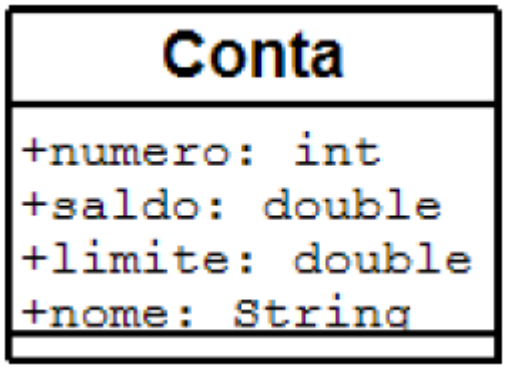
Vamos começar nossa codificação a partir do exemplo citado anteriormente, a Conta Bancária. Vamos definir uma que uma conta **TEM** e depois o que ela **FAZ.**

Um tipo desses (CONTA) pode ser facilmente traduzido para Java:

```
class Conta {
      int numero;
      String nome;
      double saldo;
      double limite;

      // ..

}
```



Uma Classe em Java

ATRIBUTO

Por enquanto declaramos o que toda conta deve ter. Estes são os **atributos** que toda conta, quando criada, vai ter. Repare que essas variáveis foram declaradas fora de um bloco, diferente do que a gente fazia quando tinha aquele **main**. Quando uma variável é declarada diretamente dentro do escopo da classe, e chamada de **variável de objeto**, ou **atributo**.

Criando e usando um objeto

NEW

Agora temos uma classe em Java, que especifica o que todo objeto dessa classe deve ter. Mas como usá-la?

Além dessa classe ainda teremos o **Programa.java**, e a partir dele é que iremos utilizar a classe Conta. Para criar (construir, instanciar) uma Conta, basta usar a palavra chave **new**, utilizamos também o parênteses, que descobriremos o que exatamente ele é mais à frente:

```
class Programa {
        public static void main(String[] args) {
                new Conta();
        }
}
```

Criando e usando um objeto

Bem, o código anterior cria um objeto do tipo Conta, mas como acessar esse objeto que foi criado? Precisamos ter alguma forma de nos referenciar a esse objeto. Precisamos de uma variável:

```
class Programa {
      public static void main(String[] args) {
            Conta minhaConta;
            minhaConta = new Conta();
      }
}
```

Pode parecer estranho escrevermos duas vezes Conta: uma vez na declaração da variável e outra vez no uso do **new**. Mas há um motivo que iremos entender também posteriormente.

Criando e usando um objeto

Através da variável **minhaConta** agora podemos acessar o objeto recém criado para alterar seu nome, seu saldo etc.:

```
1.class Programa {
2.    public static void main(String[] args) {
3.          Conta minhaConta;
4.          minhaConta = new Conta();
5.
6.          minhaConta.nome = "Duke";
7.          minhaConta.saldo = 1000.0;
8.
9.          System.out.println("Saldo atual: " + minhaConta.saldo);
10.   }
11.}
```

E importante fixar que o ponto foi utilizado para acessar algo em **minhaConta**. Agora, **minhaConta** pertence ao Duke, e tem saldo de mil reais.

Métodos

Dentro da classe, também iremos declarar o que cada conta faz, e como isto é feito. Os comportamentos que cada classe tem, isto é, o que ela faz. Por exemplo, de que maneira que uma Conta saca dinheiro? Iremos especificar isso dentro da própria classe Conta, e não em um local desatrelado das informações da própria Conta. É por isso que essas "funções" são chamadas de **método**, pois é a maneira de fazer uma operação com um **objeto**.

Queremos criar um método que saca uma determinada quantidade e não retorna nenhuma informação para quem acionar esse método:

```
void saca(double quantidade) {
        double novoSaldo = this.saldo - quantidade;
        this.saldo = novoSaldo;
}
```

A palavra chave **void** diz que, quando você pedir para a conta imprimir o nome do banco, nenhuma informação será enviada de volta a quem pediu.

Métodos

ARGUMENTO / PARÂMAETRO

Quando alguém pedir para sacar, ele também vai dizer quanto quer sacar. Por isso precisamos declarar o método com algo dentro dos parênteses, o que vai aí dentro é chamado de **argumento do método** (**ou parâmetro**). Essa variável é uma variável comum, chamada também de temporária ou local, pois ao final da execução desse método, ela deixa de existir.

Dentro do método, estamos declarando uma nova variável. Essa variável, assim como o argumento, vai morrer no fim do método, pois este é seu **escopo**. No momento que vamos acessar nosso atributo, usamos a palavra chave **this** para mostrar que esse é um atributo, e não uma simples variável. (veremos depois que é opcional)

Repare que nesse caso, a conta pode estourar o limite fixado pelo banco. Mais para frente iremos evitar essa situação, e de uma maneira muito elegante. Da mesma forma, temos o método para depositar alguma quantia:

```
void deposita(double quantidade) {
        this.saldo += quantidade;
}
```

Métodos

Observe que, agora, não usamos uma variável auxiliar e ainda usamos a abreviação **+=** para deixar o método bem simples. O **+=** soma quantidade ao valor antigo do saldo e guarda no próprio saldo o valor resultante.

```
void deposita(double quantidade) {
        this.saldo += quantidade;
}
```

Métodos

INVOCAÇÃO DE MÉTODO

Para mandar uma mensagem ao objeto, e pedir que ele execute um método, também usamos o ponto. O termo usado para isso é uma **invocação de método.**

O código a seguir saca um dinheiro e depois deposita outra quantia na nossa conta:

```
1.class SacaEDeposita {
2.    public static void main(String[] args) {
3.          // criando a conta
4.          Conta minhaConta;
5.          minhaConta = new Conta();
6.
7.          // alterando os valores de minhaConta
8.          minhaConta.nome = "Duke";
9.          minhaConta.saldo = 1000;
10.
11.         // saca 200 reais
12.         minhaConta.saca(200);
13.
14.         // deposita 500 reais
15.         minhaConta.deposita(500);
16.         System.out.println(minhaConta.saldo);
17.   }
18.}
```

Uma vez que seu saldo inicial é 1000 reais, se sacamos 200 reais, depositamos 500 reais e imprimimos o valor do saldo, o que será impresso?

Métodos com Retorno

RETURN

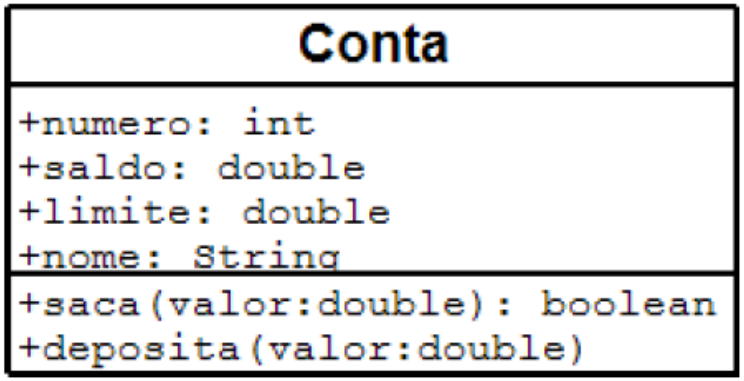
Um método sempre te que retornar alguma coisa, nem que essa coisa seja nada, como nos exemplos anteriores, estávamos usando o **void**.

Um método pode retornar um valor para o código que o chamou. No caso do nosso método **saca** podemos devolver um valor booleano indicando se a operação foi bem sucedida.

```
boolean saca(double valor) {
        if (this.saldo < valor) {
                return false;
        }
        else {
                this.saldo = this.saldo - valor;
                return true;
        }
}
```

Agora a declaração do método mudou! O método **saca** não tem **void** na frente, isto quer dizer que, quando e acessado, ele devolve algum tipo de informação. No caso, um boolean. A palavra chave **return** indica que o método vai terminar ali, retornando tal informação.

Métodos com Retorno



Exemplo:

```
minhaConta.saldo = 1000;
boolean consegui = minhaConta.saca(2000);
if(consegui){
      System.out.println("Consegui sacar");
}else{
      System.out.println("Não consegui sacar");
}
```

Métodos com Retorno

Um programa pode manter na memória mais de uma conta:

```
1.class TestaDuasContas {
2.    public static void main(String[] args) {
3.
4.          Conta minhaConta;
5.          minhaConta = new Conta();
6.          minhaConta.saldo = 1000;
7.
8.
9.          Conta meuSonho;
10.         meuSonho = new Conta();
11.         meuSonho.saldo = 1500000;
12.
13.   }
14.}
```

Objetos são acessados por referência

Quando declaramos uma variável para associar a um objeto, na verdade, essa variável não guarda o objeto, e sim uma maneira de acessá-lo, chamada de **referência**. É por esse motivo que, diferente dos *tipos primitivos* como **int** e **long**, precisamos dar **new** depois de declarada a variável:

```
1.public static void main(String args[]) {
2.     Conta c1;
3.     c1 = new Conta();
4.
5.     Conta c2;
6.     c2 = new Conta();
7.}
```

Objetos são acessados por referência

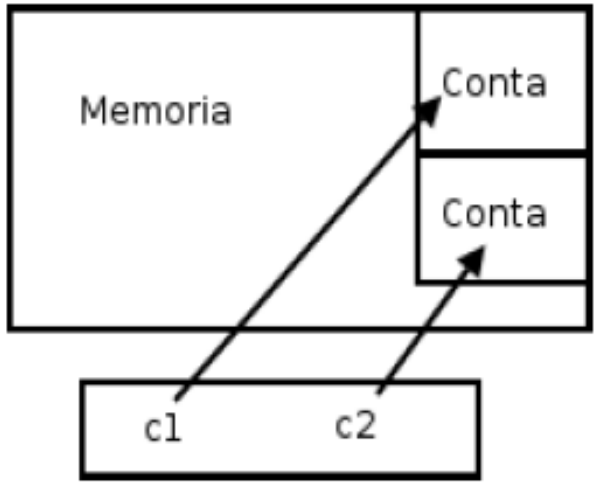
O correto aqui é dizer que c1 se refere a um objeto. Não é correto dizer que c1 é um objeto, pois c1 é uma variável referência, apesar de que depois de um tempo os programadores Java falem “Tenho um objeto c1 do tipo Conta”, mas apenas para encurtar a frase “Tenho uma referência c1 a um objeto do tipo Conta”. Basta lembrar que em Java uma variável nunca é um objeto. Não ha no Java uma maneira de criarmos o que e conhecido como “*objeto pilha*” ou “*objeto local*”, pois todo objeto em Java, sem exceção, sempre é acessado por uma *variável referência*.

Objetos são acessados por referência

O último código nos deixa a seguinte situação:

```
Conta c1;
c1 = new Conta();

Conta c2;
c2 = new Conta();
```



Objetos são acessados por referência

Internamente, c1 e c2 vão guardar um número que identifica em que posição da memória, aquela Conta se encontra. Dessa maneira, ao utilizarmos o "." para navegar, o Java vai acessar a Conta que se encontra naquela posição de memória, e não uma outra. Para quem conhece, é parecido com um ponteiro, porém você não pode manipulá-lo e utilizá-lo para guardar outras coisas.

Agora vamos a um outro exemplo:

```
1.class TestaReferencias {
2.    public static void main(String args[]) {
3.          Conta c1 = new Conta();
4.          c1.deposita(100);
5.
6.          Conta c2 = c1;  // linha importante!
7.          c2.deposita(200);
8.
9.          System.out.println(c1.saldo);
10.         System.out.println(c2.saldo);
11.   }
12.}
```

Objetos são acessados por referência

new

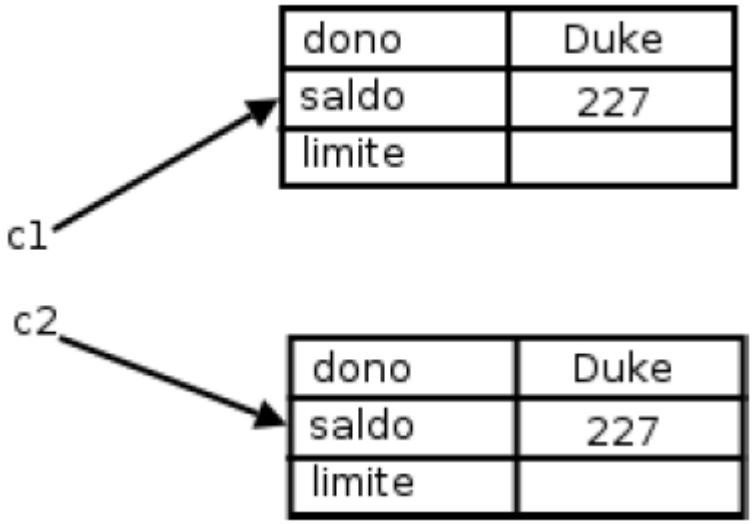
O que exatamente faz o new? O new executa uma serie de tarefas, que veremos mais adiante. Mas, para melhor entender as referências no Java, saiba que o new, depois de alocar a memória para esse objeto, devolve uma “flecha”, isto e, um valor de referência. Quando você atribui isso em uma variável, essa variável passa a se referir para esse mesmo objeto. Podemos ver outra situação:

```
1.public static void main(String args[]) {
2.     Conta c1 = new Conta();
3.     c1.nome = "Duke";
4.     c1.saldo = 227;
5.
6.     Conta c2 = new Conta();
7.     c2.dono = "Duke";
8.     c2.saldo = 227;
9.
10.    if (c1 == c2) {
11.          System.out.println("Contas iguais");
12.    }
13.}
```

Objetos são acessados por referência

O operador == compara o conteúdo das variáveis, mas essas variáveis não guardam o objeto, e sim o endereço em que ele se encontra. Como em cada uma dessas variáveis guardamos duas contas criadas diferentemente, eles estão em espaços diferentes da memória, o que faz o teste no if valer false. As contas podem ser equivalentes no nosso critério de igualdade, porem eles não são o mesmo. Quando se trata de objetos, pode ficar mais fácil pensar que o == compara se os objetos (referencias na verdade) são o mesmo, e não se são iguais.

Para saber se dois objetos tem o mesmo conteúdo, você precisa comparar atributo por atributo. Veremos uma solução mais elegante para isso também.



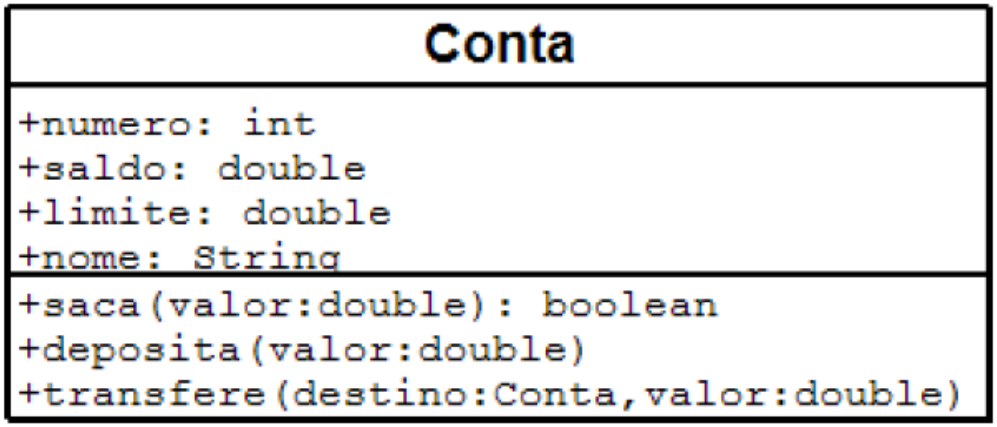
O método transfere()

A idéia e que quando chamarmos o método transfere, já teremos um objeto do tipo Conta (o this), portanto o método recebe apenas um parâmetro do tipo Conta, a Conta destino (alem do valor):

```
class Conta {

      // atributos e metodos...

      void transfere(Conta destino, double valor) {
            this.saldo = this.saldo - valor;
            destino.saldo = destino.saldo + valor;
      }
}
```



O método transfere()

Para deixar o código mais robusto, poderíamos verificar se a conta possui a quantidade a ser transferida disponível. Para ficar ainda mais interessante, você pode chamar os métodos deposita e saca já existentes para fazer essa tarefa:

```
class Conta {

      // atributos e metodos...

      boolean transfere(Conta destino, double valor) {
            boolean retirou = this.saca(valor);
            if (retirou == false) {
                  // não deu pra sacar!
                  return false;
            }
            else {
                  destino.deposita(valor);
                  return true;
            }
      }
}
```

O método transfere()

Quando passamos uma Conta como argumento, o que será que acontece na memória? Será que o objeto e *clonado?* No Java, a passagem de parâmetro funciona como uma simples atribuição como no uso do "=". Então esse parâmetro vai copiar o valor da variável do tipo Conta que for passado como argumento. E qual é o valor de uma variável dessas? Seu valor é um endereço, uma referência, nunca um objeto. Por isso não há cópia de objetos aqui.

Esse último código poderia ser escrito com uma sintaxe muito mais sucinta. Como?

Continuando com atributos

As variáveis do tipo atributo, diferentemente das variáveis temporárias (declaradas dentro de um método), recebem um valor padrão. No caso numérico, valem 0, no caso de boolean, vale false. Você também pode dar valores default, como segue:

```
1.class Conta {
2.     int numero = 1234;
3.     String dono = "Duke";
4.     String cpf = "123.456.789-10";
5.     double saldo = 1000;
6.     double limite = 1000;
```

Continuando com atributos

Nesse caso, quando você criar um carro, seus atributos já estão “populados” com esses valores colocados. Imagine que agora começamos a crescer nossa classe Conta e adicionar nome, sobrenome e cpf do cliente dono da conta. Começaríamos a ter muitos atributos... e se você pensar direito, uma Conta não tem nome, nem sobrenome nem cpf, quem tem esses atributos é um Cliente. Então podemos criar uma nova classe e fazer uma composição Seus atributos também podem ser referências para outras classes. Suponha a seguinte classe Cliente:

```
1.class Cliente {
2.     String nome;
3.     String sobrenome;
4.     String cpf;
5.}
```

```
1.class Conta {
2.     int numero;
3.     double saldo;
4.     double limite;
5.     Cliente cliente;
6.     // ..
7.}
```

Continuando com atributos

E dentro do main da classe do programa (teste):

```
1.class Teste {
2.    public static void main(String[] args) {
3.          Conta minhaConta = new Conta();
4.          Cliente c = new Cliente();
5.          minhaConta.cliente = c;
6.          // ...
7.    }
8.}
```

Aqui simplesmente houve uma atribuição. O valor da variável c é copiado para o atributo cliente do objeto a qual minhaConta se refere. Em outras palavras, minhaConta agora tem uma referência ao mesmo Cliente que c se refere, e pode ser acessado através de minhaConta.cliente.

Continuando com atributos

Você pode realmente navegar sobre toda essa estrutura de informação, sempre usando o ponto:

```
Cliente clienteDaMinhaConta = minhaConta.cliente;
clienteDaMinhaConta.nome = "Duke";
```

Ou ainda pode fazer isso de uma forma mais direta, e até mais elegante:

```
minhaConta.cliente.nome = "Duke";
```

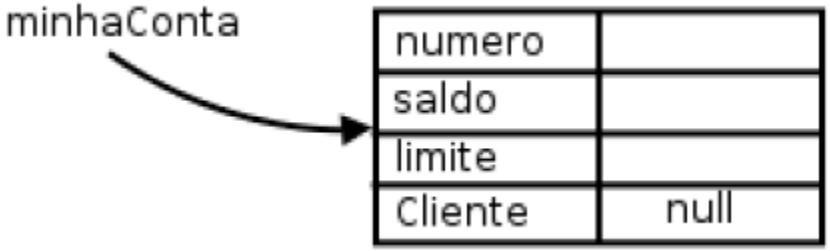
Mas e se dentro do meu código eu não desse new em Cliente e tentasse acessá-lo diretamente?

```
class Teste {
      public static void main(String[] args) {
            Conta minhaConta = new Conta();

                  minhaConta.cliente.nome = "paulo";
                  // ...
            }
      }
```

Continuando com atributos

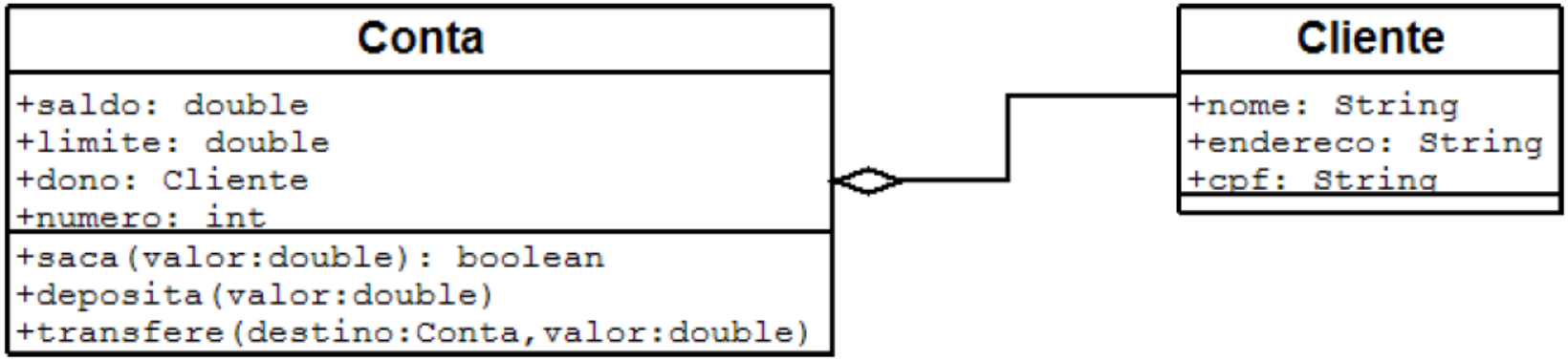
Quando damos new em um objeto, ele o incializa com seus valores default, 0 para números, false para boolean e null para referências. null é uma palavra chave em Java, que indica uma referência para nenhum objeto.



Se em algum caso você tentar acessar um atributo ou método de alguém que está se referenciando para null, você receberá um erro durante a execução (NullPointerException, que veremos mais a frente). Dá para perceber então que o new não traz um efeito cascata, a menos que você dê um valor default (ou use construtores que também veremos mais a frente):

```
1.class Conta {
2.    int numero;
3.    double saldo;
4.    double limite;
5.    Cliente cliente = new Cliente();    // quando chamarem new Conta,
6.                                        //havera um new Cliente para ele.
7.}
```

Continuando com atributos



Com esse código, toda nova Conta criada criado já terá um novo Cliente associado, sem necessidade de instanciá-lo logo em seguida da instanciação de uma Conta. Qual alternativa você deve usar? Depende do caso: para toda nova Conta você precisa de um novo Cliente? E essa pergunta que deve ser respondida. Nesse nosso caso a resposta e não, mas depende do nosso problema.

Para saber um pouco mais: Uma Fábrica de Carros

```
1.class Carro {
2.    String cor;
3.    String modelo;
4.    double velocidadeAtual;
5.    double velocidadeMaxima;
6.
7.    //liga o carro
8.    void liga() {
9.          System.out.println("O carro está ligado");
10.   }
11.
12.   //acelera uma certa quantidade
13.   void acelera(double quantidade) {
14.         double velocidadeNova = this.velocidadeAtual + quantidade;
15.         this.velocidadeAtual = velocidadeNova;
16.   }
17.
18.   //devolve a marcha do carro
19.   int pegaMarcha() {
20.         if (this.velocidadeAtual < 0) {
21.               return -1;
22.         }
23.         if (this.velocidadeAtual >= 0 && this.velocidadeAtual < 40) {
24.               return 1;
25.         }
26.         if (this.velocidadeAtual >= 40 && this.velocidadeAtual < 80 {
27.               return 2;
28.         }
29.         return 3;
30.   }
31.}
```

Além do Banco que estamos criando, vamos ver como ficariam certas classes relacionadas a uma fábrica de carros. Vamos criar uma classe Carro, com certos atributos que descrevem suas características e com certos métodos que descrevem seu comportamento.

Para saber um pouco mais: Uma Fábrica de Carros

Agora vamos testar a nossa classe Carro em um programa:

```
1.class TestaCarro {
2.    public static void main(String[] args) {
3.          Carro meuCarro;
4.          meuCarro = new Carro();
5.          meuCarro.cor = "Verde";
6.          meuCarro.modelo = "Fusca";
7.          meuCarro.velocidadeAtual = 0;
8.          meuCarro.velocidadeMaxima = 80;
9.
10.         // liga o carro
11.         meuCarro.liga();
12.
13.         // acelera o carro
14.         meuCarro.acelera(20);
15.         System.out.println(meuCarro.velocidadeAtual);
16.   }
17.}
```

Um pouco mais...

1) Quando declaramos uma classe, um método, ou um atributo, podemos dar o nome que quiser, seguindo uma regra. Por exemplo, o nome de um método não pode começar com um número. Pesquise sobre essas regras.

2) Como você pode ter reparado, sempre damos nomes as variáveis com letras minúsculas. E que existem convenções de código, dadas pela Sun, para facilitar a legibilidade do código entre programadores. Essa convenção é *muito seguida. Pesquise sobre* ela no http://java.sun.com, procure por “code conventions”.

3) E necessário usar a palavra chave this quando for acessar um atributo? Para que então utilizá-la?

4) Existe um padrão para representar suas classes em diagramas que é amplamente utilizado chamado UML. Pesquise sobre ele.

Dados de Contato


79 9949 4098


http://pedroneto.yolasite.com/


pedro@pyxistec.com.br


psneto@emsergipe.com


pedro.pyxistec@gmail.com


http://www.facebook.com/pedro.neto.se


pedropyxis


http://lattes.cnpq.br/4891420246888248