Básicos da Orientação a Objetos em Java[1]

Artigo de Jeff Friesen, publicado originalmente na revista JavaWorld (www.javaworld.com),
Copyright Itworld.com, Inc., Novembro de 2000. Reimpresso, adaptado e traduzido com permissão do autor.
(www.javaworld.com/javaworld/jw-04-2001/jw-0406-java101_p.html)

Traduzido e adaptado por Leandro Krug Wives

Parte I: Aprenda a declarar classes e criar objetos

Existem, basicamente, duas metodologias de desenvolvimento de software: a Programação Estruturada (PE) e a Programação Orientada a Objetos (POO).

Na primeira, o objetivo consiste em separar os dados da funcionalidade do programa. Neste caso, as entidades são representadas de forma tal que satisfaçam as restrições da linguagem de programação utilizada. Isso acaba gerando programas que são difíceis de manter e compreender, principalmente se eles forem muito grandes.

Na segunda, a linguagem de programação é que tem de satisfazer as restrições e requisitos das entidades. Isso permite a construção de modelos mais realísticos do mundo, e, portanto, mais fáceis de serem mantidos e compreendidos.

Vamos tomar como exemplo a representação de um carro. Vamos descobrir que cada metodologia modela de forma diferente esse carro.

Na PE, por exemplo, um carro é representado por um conjunto de funções (trechos de código) responsáveis por ligar, frear, estacionar, acelerar e assim por diante. Um conjunto separado de variáveis define a cor do carro, seu modelo, fabricante e demais características. Você deve então inicializar (atribuir valores) às variáveis do carro e chamar algumas das funções para manipular estas variáveis para que o carro funcione.

Já na POO, o carro é visto como uma integração de seus comportamentos (suas funções) e atributos (variáveis) que mantêm o estado atual do carro. A integração destes atributos e comportamentos resulta em um objeto. A criação deste objeto é simples e faz com que ele seja inicializado ao mesmo tempo sem comandos adicionais. Neste caso, um programa orientado a objetos pensa em objetos e não em funções e variáveis separadas e soltas (na PE o programador é que sabe quais funções e quais variáveis pertencem ao carro; na POO, o próprio programa controla isso para o programador).

A integração destes atributos e comportamentos em objetos é chamada de "encapsulamento". O encapsulamento também permite com que as informações dos objetos fiquem escondidas, limitando o acesso a variáveis e funções que não necessitam ser manipuladas ou acessadas fora do objeto.

A orientação a objetos já existe há 15 anos, porém, eram poucas as linguagens que suportavam-na corretamente e por completo. Atualmente existem linguagens, tais como o Java, que suportam completamente a orientação a objetos. Devido a isso, todo programa escrito em Java necessita da presença de pelo menos uma classe.

Mas o que é uma classe? Bem, os objetos não surgem do nada. Assim como um mestre de obras necessita de um esquema de engenharia para realizar a construção de um prédio (a "planta do prédio"), um programa necessita de uma classe para criar objetos baseados nesta classe. A classe é como se fosse o esquema (a receita) que indica como criar objetos do seu tipo. Esse esquema ou receita indica quais são os comportamentos, as características e o estado (valor) destas características nos objetos criados a partir dela.

Declarando Classes

A classe é a planta ou esquema (em formato de código Java) que indica como os objetos são criados, quais os seus comportamentos e variáveis de estado. Para criar declarar uma classe você deve utilizar a seguinte sintaxe[2]:

```
[ palavra-chave ] 'class' nome_da_classe
'{'
  // comportamentos e variáveis são declarados e descritos entre o '{' e o '}'
'}'
```

[1] By Jeff Friesen. It was originally published by JavaWorld (www.javaworld.com), copyright ITworld.com, Inc., November 2000. Reprinted with permission.

[2] Preste atenção na convenção de sintaxe utilizada: os itens entre colchetes são opcionais e, geralmente, devem ser substituídos por palavras-chave detalhadas em seguida ao código; os itens entre apóstrofes são obrigatórios e correspondem aos comandos ou estruturas da linguagem; já os itens em itálico são obrigatórios, mas você pode utilizar qualquer identificador (palavra, desde que não seja reservada) para representá-los.

Logo, para declarar uma classe, coloque a palavra **class** seguida de um identificador (representado por *nome_da_classe*) que irá servir de nome para a classe. O identificador pode ser qualquer palavra, com exceção de alguma palavra reservada da linguagem (ou seja, algum comando). Por exemplo, **class Conta** introduz a declaração de uma nova classe onde **Conta** é o nome da classe. Note que, por convenção, o nome de uma classe inicia sempre com uma letra maiúscula.

A **palavra-chave** é opcional (pois está entre colchetes), e pode ser qualquer uma das seguintes: '**public'**, '**abstract', 'public abstract'** ou **'final'**.

Utilize a palavra-chave '**public**' (sem as apóstrofes) para criar um nome de classe que possa ser utilizado em qualquer outra classe de qualquer programa ou pacote[3]. Se você não colocar **public** na frente de uma declaração de classe, ela só poderá ser utilizada dentro do código de programa ou pacote atual. É importante salientar que você pode declarar mais de uma classe por arquivo, porém, somente uma delas poderá ser **public** (pública). Neste caso, o nome do arquivo do código-fonte deverá ser o mesmo da classe pública.

Exemplo:

```
public class MinhaClasse
{ }
```

Neste caso, o nome do arquivo onde **MinhaClasse** foi definida deve se chamar **MinhaClasse.java**. Declarar essa classe (pública) em outro arquivo não é permitido e deve gerar um erro do compilador. Note que o Java considera diferentes letras maiúsculas e minúsculas. Logo, o nome da classe e o nome do arquivo devem ser exatamente iguais (em termos de letras maiúsculas e minúsculas). Isso significa que o nome do arquivo deve ser escrito com o 'M' maiúsculo (de Minha) e 'C' maiúsculo (de Classe). As demais letras devem ser minúsculas.

Utilize a palavra-chave 'abstract' para indicar que a classe corresponde a uma classe abstrata. Classes abstratas não podem ser utilizadas para a criação de objetos. Elas só servem como "classes-pai", para que "classes-filhas" sejam criadas a partir delas.

Utilize a palavra-chave 'final' para declarar uma classe final. Classes finais não podem ser utilizadas como classes-pai, impedindo com que classes-filhas sejam criadas a partir delas (e herdem seu comportamento e estados).

Essas palavras-chave podem ser utilizadas em qualquer ordem. Você só não pode especificar uma classe como sendo abstrata e final ao mesmo tempo, pois isso não faz sentido, já que classes abstratas só servem para a criação de classes-filhas e classes finais só funcionam como classes filhas.

Cada classe possui um "corpo" onde você declara os comportamentos (métodos) e variáveis de estado (atributos ou campos). O corpo começa com o caractere ' **{** ' e termina com o caractere ' **}** '.

Criando Objetos

Um objeto é uma instância de uma classe. Para criar um objeto, utilize a seguinte sintaxe:

```
'new' construtor
```

O comando **new** (que significa novo em Inglês), também conhecido como operador de criação, cria um novo objeto, alocando memória para o objeto e inicializando essa memória para valores default (padrão). O comando **new** necessita de um operando: o **construtor**, que é o nome de um método especial que constrói o objeto. Uma vez construído, o objeto deve ser atribuído a uma variável, para que possa ser utilizado e referenciado no futuro. Preste atenção ao exemplo seguinte:

```
class Pessoa{
 }
class ProgramaExemplo{
   public static void main(Strings parametros[]){
      Pessoa umaPessoaQualquer; // declara uma variável que pode armazenar objetos
                                // do tipo Pessoa (definido na classe anterior)
      umaPessoaQualquer = new Pessoa(); // cria uma nova pessoa e coloca-a na
                                        // variável umaPessoaQualquer
   }
}
```

[3] Pacote é o nome dado a uma biblioteca de classes desenvolvidas na linguagem Java.

No exemplo anterior foi definida uma classe chamada **Pessoa** e após, no programa principal, foi criada uma variável chamada **umaPessoaQualquer** para receber um objeto do tipo pessoa (note que o tipo da variável é Pessoa, pois a variável é declarada de forma similar à declaração de variáveis do tipo **int** qualquer outro tipo natural da linguagem). Após, um novo objeto do tipo Pessoa é criado, através do comando **new Pessoa()**. Esse objeto é colocado na variável **umaPessoaQualquer** através do comando de atribuição (sinal de igual).

Em Java, você pode criar e atribuir um novo objeto a uma variável, simultaneamente. Basta colocar tudo na mesma linha da seguinte forma:

```
Pessoa umaPessoaQualquer = new Pessoa();
```

Destruindo objetos

A linguagem Java assume a responsabilidade de destruir qualquer objeto que você tenha criado e não esteja mais usando. Ela faz isso através de um processo especial chamado **coletor de lixo** (garbage collector), que é executado em intervalos regulares, examinando cada objeto para ver se ele ainda é referenciado por alguma variável. Caso o objeto não seja utilizado ao menos por uma variável, ele é destruído e sua memória é liberada.

Parte II: Declarando e acessando atributos e métodos

Em Java, as variáveis podem ser divididas em três categorias: **atributos** (também chamados de campos), **parâmetros** e **variáveis locais**. No início desta seção será explorado o primeiro tipo – o atributo. Os demais tipos serão discutidos em conjunto com os métodos, apresentados em seguida.

Declarando atributos

Um **atributo** ou **campo** é uma variável declarada no corpo de uma classe. Ele serve para armazenar o **estado de um objeto** (e neste caso é chamado de atributo de instância) ou o **estado de uma classe** (atributo de classe). A sintaxe de declaração de um atributo é a seguinte:

```
[ ( 'public' | 'private' | 'protected' ) ]
  [ ( 'final' | 'volatile' ) ]
    [ 'static' ] [ 'transient' ]
       tipo_de_dado nome_do_atributo [ '=' expressão ] ';'
```

A declaração de um atributo especifica o seu nome, seu tipo (que pode ser um dos tipos naturais, tais como **int**, **char**, **double**, **float**, ou o nome de uma classe, como **String** ou **Pessoa**), uma expressão opcional (que inicializa o atributo), especificadores de acesso e modificadores. Você pode dar qualquer nome ao atributo, desde que este nome não corresponda a uma palavra-reservada da linguagem. Por questões de padronização, o nome de um atributo sempre começa com uma letra minúscula. Considere o exemplo:

```
class Empregado{
   String nome;     // o nome do empregado
   double salario;  // o salário do empregado
   int    idCargo   // identificador do código da função que ele exerce
}
```

Neste exemplo, a classe chamada **Empregado** declara três atributos: **nome**, **salário** e **idCargo** (identificador do cargo). Assim, quando você cria um objeto do tipo Empregado, o campo nome mantém uma referência para uma string contendo o nome do empregado. O campo salário armazena o seu salário, e o campo idCargo armazena um número que identifica a categoria ou função que ele exerce na empresa.

Especificadores de acesso

Opcionalmente, você pode utilizar uma palavra-chave modificadora de acesso ao declarar um atributo. As palavras-chave modificadoras de acesso são as seguintes: **public**, **private** ou **protected**. O modificador de acesso determina quão acessível o atributo é para o código de outras classes do seu programa. Essa acessibilidade pode variar de completamente inacessível até completamente acessível.

Se você não especificar um modificador de acesso, o Java assume o acesso padrão, que permite com que o campo seja acessível a classe no qual ele foi declarado e também a todas as classes existentes no mesmo arquivo (código-fonte) ou pacote (biblioteca de classes) da classe que o declarou.

Se você declarar um atributo como sendo privado (**private**), somente o código contido na sua classe pode acessá-lo. Este atributo permanece "escondido" para qualquer outra classe, mesmo para as classes-filhas da classe ao qual foi declarado (neste caso, elas conseguem herdá-lo, mas não conseguem manipulá-lo).

Se você declarar um atributo como sendo público (**public**), o código contido na sua classe e em todas as outras classes em qualquer pacote (biblioteca) pode acessá-lo.

Um atributo protegido (**protected**) possui o nível de acesso padrão, porém, as sub-classes (classes-filhas) da classe para o qual foi declarado também podem utilizá-lo.

```
class Empregado{

   public    String nome;     // seu nome (visível à qualquer classe)
   private   double salario;  // seu salário (inacessível fora da classe)
   protected int    idCargo   // código do cargo (acessibilidade padrão)

}
```

Mas, afinal, para que servem os modificadores de tipo? Bem, eles são utilizados para implementar o conceito de **esconderijo ou encoberta de informação** (information hiding) e encapsulamento. Para compreender melhor, vamos fazer uma analogia: suponha que você crie classes chamadas de **Corpo**, **Olho**, **Coração** e **Pulmão** para modelar o corpo humano, os olhos, o coração e os pulmões de uma pessoa. A classe **Corpo** declara atributos para referenciar os olhos, o coração e os pulmões, como demonstrado no exemplo seguinte:

```
public class Corpo{

   public  Olho    olhoEsquerdo,olhoDireito;
   private Coração coração;
   private Pulmão  pulmãoEsquerdo, pulmãoDireito;

}
```

Os atributos **olhoEsquerdo** e **olhoDireito** são públicos porque os olhos das pessoas são visíveis para qualquer observador. Já o **coração**, o **pulmãoEsquerdo** e o **pulmãoDireito** são privados porque os órgãos que estes atributos representam estão escondidos dentro do corpo de uma pessoa. Se eles tivessem sido declarados públicos, uma pessoa teria seu coração e pulmões à vista... Muito estranho, você não acha?

Modificadores

Você pode utilizar, opcionalmente, uma palavra-chave modificadora: **final** ou **volatile** e/ou **static** e/ou **transient**.

Se você utilizar o modificador **final** na declaração de um atributo o compilador assume que esse campo é uma constante (uma variável somente de leitura que não pode ser modificada). Neste caso o compilador faz ajustes e otimizações internas no código do programa, já que ele sabe que esta variável não será modificada em momento algum da execução do programa.

O exemplo seguinte declara três constantes. Note que as constantes são obrigatoriamente inicializadas na sua declaração. Além disso, todas as letras do nome de uma constante são maiúsculas, por convenção. Assim, você pode rapidamente saber se está trabalhando com uma variável comum ou com uma constante (já que as constantes possuem todas as suas letras em maiúsculo e as variáveis comuns não).

```
class Empregado{

   final int FAXINEIRO = 1;
   final int PROGRAMADOR = 2;
   final int GERENTE = 3;

   int idCargo = PROGRAMADOR;

}
```

Se você declara um atributo utilizando o modificador **volatile** (volátil), várias Threads (linhas ou processos de execução) podem acessar esse atributo. Neste caso, o próprio compilador realiza ajustes para que esse atributo seja acessado corretamente por cada uma delas, sem que ocorram problemas.

Se o atributo for declarado como sendo **static** (estático) todos os objetos compartilham (utilizam) a mesma cópia da variável. Se um dos objetos modifica esse atributo todos os outros objetos enxergam essa mesma modificação. Neste caso, o atributo é considerado um atributo de classe e não de objeto. Se você não especificar o modificador **static** cada objeto vai possuir a sua cópia específica deste atributo.

Finalmente o modificador **transient** indica que o atributo não será utilizado em uma operação de serialização do objeto (salvamento em arquivo ou envio de um objeto para outro computador ou programa). Esse processo não será abordado aqui, logo, para maiores informações, consulte a WEB ou um livro de Java.

Atributos de objetos (de instância)

Um atributo de objeto (também conhecido por atributo de instância) é uma variável declarada sem o modificador **static**. Os atributos de instância pertencem (são associados) aos objetos e não às classes. Quando um atributo de objeto é modificado, somente o objeto a que ele pertence enxerga a alteração (isso porque cada objeto possui o seu). Um atributo de objeto (ou instância) é criado quando o objeto é criado (comando **new**) e é destruído quando o objeto a que pertence for destruído.

Veja o exemplo:

```
class Empregado{

   public int cargo;

   public void contrataComoProgramador(void){
      cargo = 2; // Dois é o código de programador
   }

}
```

Para acessar o atributo de objeto em um método pertencente à mesma classe da qual o objeto foi criado (ou seja, um método também pertencente ao objeto), simplesmente especifique o nome do atributo (o nome da variável).

No exemplo anterior, o atributo cargo é acessado simplesmente pela utilização do seu nome (veja a linha grifada).

Se você quiser acessar um atributo em um método de outro objeto, você vai necessitar de uma referência (uma variável) ao objeto ao qual o atributo pertence. Utilize essa variável como prefixo do atributo, seguido do caractere '**.**' (ponto). Após, especifique o atributo desejado:

```
class Empregado{

   public int cargo;

   public void contrataComoProgramador(void){
      cargo = 2; // Dois é o código de programador
   }

}
class ProgramaExemplo{   // Esse programa utiliza a classe Empregado (anterior)

   public static void main(String parametros[]){

      Empregado empregado = new Empregado(); // cria um objeto empregado

      empregado.cargo = 2; // muda o atributo cargo de um empregado

   }

}
```

Note a diferença: os métodos da classe **Empregado** podem utilizar o atributo cargo **diretamente**; já os métodos da classe **ProgramaExemplo** devem sempre criar um objeto do tipo Empregado e utilizar esse objeto para ter acesso ao atributo (**empregado.cargo**). Isso serve para quando você tiver vários objetos (vários empregados) e necessitar alterar o atributo de um deles. Imagine que seu programa tenha quatro variáveis do tipo empregado e um deles tenha mudado de cargo. Essa mudança é muito fácil de ser feita, basta você informar qual objeto (qual empregado) que deve sofrer a alteração, seguido do seu atributo cargo, e pronto:

```
class ProgramaExemplo2{   // Esse programa utiliza a classe Empregado (anterior)

   public static void main(String parametros[]){

      Empregado empregado1 = new Empregado();
      Empregado empregado2 = new Empregado();
      Empregado empregado3 = new Empregado();
      Empregado empregado4 = new Empregado();

      empregado1.cargo = 3; // muda o cargo do empregado1 para gerente
      empregado2.cargo = 1; // muda o cargo do empregado2 para faxineiro
      empregado3.cargo = 2; // muda o cargo do empregado3 para programador
      empregado4.cargo = 2; // muda o cargo do empregado4 para programador

   }

}
```

Quando a Máquina Virtual Java (MVJ ou Virtual Java Machine – JVM, que é o computador virtual onde um programa em Java é executado) cria um objeto, ela aloca memória para todos os atributos do objeto e zera essa memória, inicializando-a. Esse valor inicial depende do tipo do atributo. Para **tipos naturais numéricos** o valor inicial é **zero** (0 ou 0.0). Os valores **booleanos** são inicializados em **false** (falso). Para os **demais tipos** de atributos (referências construídas a partir de outras classes) o valor inicial é **NULL** (nulo).

Atributos de classes

Um atributo de classe é um atributo declarado com o modificador **static**. Os atributos de classe são associados com a classe e não com os objetos (ou instâncias) criados a partir dela. Isso significa que quando um atributo de classe é modificado, todos os objetos criados a partir da mesma classe enxergam a alteração. Veja o exemplo:

```
class Empregado{
   static int numeroDeEmpregados; // atributo de classe
   public int cargo;              // atributo de instância (objeto)
}
class ProgramaExemplo3 {
   public static void main(String parametros[]){
      Empregado empregado1 = new Empregado(); // cria um objeto empregado
      empregado1.numeroDeEmpregados = 1;
      empregado1.cargo = 2;
      Empregado empregado2 = new Empregado(); // cria um novo empregado
      empregado2.cargo = 3;
      empregado2.numeroDeEmpregados = 2; // modifica a variável de classe
      System.out.println(empregado1.cargo); // imprime o cargo do empregado1
      System.out.println(empregado1.numeroDeEmpregados);     // vai mostrar 2
      // porque a variável numeroDeEmpregados é da classe Empregado (inteira)
      // e não somente do empregado1 ou do empregado2
      System.out.println(empregado2.cargo); // imprime o cargo do empregado2
   }
}
```

No exemplo anterior, quando o atributo numeroDeEmpregados é modificado em um dos objetos, todos sofrem a mesma alteração, pois, diferente do atributo de objeto (onde cada um possui uma cópia diferente do atributo), ele pertence à classe e não ao objeto.

Similarmente, você pode definir métodos de classe e métodos de objeto. A função **main** de um programa é um exemplo. Note que ela sempre é definida como um método de classe, pois possui o modificador **static**: public **static** void main(String parametros[]). **Saiba mais sobre métodos na seção seguinte: Declarando métodos**.

Observação: Atributos de classes são as coisas mais próximas à variáveis globais que a linguagem Java oferece.

Constantes

Uma **constante** é uma variável que só permite a leitura do seu conteúdo (valor). Uma vez definida, uma constante não pode mais ser modificada. As constantes são declaradas através da palavra-chave **final**. As constantes também podem ser de classe ou de objeto. Por questões de performance, sempre crie constantes de classe, como no exemplo:

```
class ProgramaExemplo4 {
  final static TOTAL = 10;
   public static void main(String parametros[]){
      int contador = 0;
      while(contador < TOTAL) contador++;
   }
}
```

Esse exemplo cria uma constante chamada **TOTAL** e inicializa-a com o valor 10. O valor de TOTAL não pode ser alterado em momento algum do programa. Caso você tente alterá-lo o compilador Java acusará um erro. Note que a constante foi definida com **todas as suas letras em formato maiúsculo**. Apesar de não ser obrigatório fazer isso, por questões de padronização as constantes são escritas com letras maiúsculas. Deste modo, você pode identificar no seu programa quais variáveis correspondem a atributos comuns e quais correspondem a constantes.

Utilize constantes pelas seguintes razões:

1) Constantes facilitam a leitura e o entendimento do código-fonte, pois fazem mais sentido do que um número sem explicação (TAMANHO_DO_VETOR, por exemplo, é mais significativo do que o número 20);
2) É mais fácil modificar o valor inicial de uma constante do que trocar todas as ocorrências de determinado valor no código-fonte.

Agora que você já aprendeu a declarar atributos, você vai precisar aprender a declarar e utilizar métodos.

Declarando métodos

A linguagem Java utiliza o termo **método** para se referir a trechos de código que são associados à classes. Se os atributos servem para manter ou armazenar o estado ou o valor de um objeto, os métodos servem para descrever os comportamentos de um objeto ou classe. Um método é muito similar a uma função em linguagem C. A maior diferença é que os métodos da linguagem Java são declarados completamente dentro de uma classe.

Para declarar um método utilize a seguinte sintaxe:

```
[ ( 'public' | 'private' | 'protected' ) ]
  ( [ 'abstract' ] | [ 'final' ] [ 'static' ] [ 'native' ] ['synchronized' ] )
      tipo_de_retorno nome_do_método '(' [ lista_de_parâmetros ] ')'
      corpo_do_método
```

A declaração de um método consiste de uma **assinatura** do método seguida por seu **corpo** ou **trecho de código**. A **assinatura do método** especifica o seu **nome**, **tipo de retorno**, **lista de parâmetros**, **especificadores de acesso**, **modificadores** e os **tipos de exceção** que o método pode gerar (exceções serão discutidas em um outro momento). O **corpo** do método é um grupo ou bloco de comandos que são executados quando o método é chamado por um outro método. O corpo de um método é sempre declarado dentro de colchetes **{ }**.

O **nome** do método identifica-o. Escolha um identificador (nome) que não corresponda a uma palavra-reservada da linguagem Java.

Precedendo o nome do método há sempre o **tipo de retorno**. Ele identifica o tipo de dado que o método retorna, podendo indicar também que o método não retorna nada. Os valores possíveis para ele são:

1) Um dos tipos de dados primitivos: **boolean**, **byte**, **char**, **double**, **float**, **int**, **long** ou **short**;
2) Uma referência a um objeto (nome da classe do objeto que será retornado);
3) O tipo **void**, indicando que o método não retorna valor algum.

Um par de parênteses (abre e fecha-parênteses) segue o nome do método. **Esse par de parênteses é obrigatório**. Opcionalmente você pode declarar, dentro destes parênteses, uma lista de variáveis separadas por vírgulas. Cada uma destas variáveis é conhecida como **parâmetro.** Assim, quando o método é chamado (em algum lugar do programa) uma lista de expressões separadas por vírgulas (chamadas de **argumentos**) pode ser passada para ele. **O número de argumentos e seus tipos de dados devem ser iguais ao número de parâmetros e o tipo de dados de cada um deles (da esquerda para a direita)**.

A seguir são apresentados alguns exemplos de assinaturas de métodos:

- `int contaAté(int limite)`
  Declara um método chamado **contaAté**, que recebe um parâmetro em uma variável chamada **limite** e retorna um valor decimal inteiro (**int**) após o processamento.
- `int soma(int a, int b)`
  Declara um método chamado **soma**, que recebe dois parâmetros decimais inteiros: o primeiro na variável **a** e o segundo na variável **b**. Após o processamento a função retorna um valor decimal inteiro (**int**).

- `void imprime(String texto)`
  Declara um método chamado **imprime**, que recebe um parâmetro do tipo String na variável **texto**. A função processa e não retorna valor para quem a chamou.
- `float sorteiaNota(void)`
  Declara um método chamado **sorteiaNota**, que não recebe parâmetro algum e retorna um número de ponto flutuante (natural, com vírgula), do tipo **float**, para quem a chamou.

Vamos ver um exemplo de um método completo, que inclua o seu corpo:

`int soma(int limite)`	*Assinatura do método*

`{ // início do corpo do método`
`int total;`
`for (int i = 1; i<= limite; i++)`
`total += i;`
`return total;`
`} // fim do corpo do método`
	Corpo do método

Neste último exemplo, a primeira linha corresponde à assinatura do método. Essa assinatura indica que o método retorna um valor do tipo **int** (um número inteiro decimal). Além disso, ela indica que o método se chama **soma** e recebe um parâmetro decimal inteiro, chamado **limite**.

O corpo do método inicia com o abre-colchete ' **{** '. Segue-se, em seguida, uma série de comandos que executam determinadas tarefas em função do parâmetro recebido (no caso, a soma dos números existentes entre 1 e o valor passado como parâmetro em **limite**). O corpo do método termina quando o fecha-colchete ' **}** ' correspondente é encontrado. Note que na linha anterior ao fecha-colchete o comando **return** é utilizado. Esse comando faz com que o valor da variável **total** seja retornado. O tipo da variável ou valor retornado deve ser o mesmo que foi declarado como tipo de retorno na assinatura do método (**int**, no caso).

Chamando (executando) um método

Para chamar um método, passe zero ou mais argumentos para o método. O método acessa estes argumentos (colocando-os nas variáveis correspondentes de parâmetros), processa-os e então, opcionalmente, retorna um valor para quem chamou esse método. Para fins de exemplo, vamos supor que você tenha criado um método para calcular a raiz-quadrada de um número e que agora você esteja fazendo um programa que necessite deste método. O código do seu programa chama o método de cálculo da raiz-quadrada passando o valor (número) do qual você deseja saber a raiz quadrada. O método de cálculo de raiz-quadrada recebe o argumento passado (o valor, que poderia até mesmo estar em uma variável) via o parâmetro correspondente. Após algum tempo de processamento, o método retorna a raiz-quadrada do valor solicitado para o programa que o chamou. Veja o exemplo:

```
class Matemática {
    float raizQuadrada(float numero){
        float b = numero * numero;

        // código que calcula a raiz quadrada do número passado como parâmetro
    }
}

class ProgramaExemplo {
   public static void main(String argumentos[])
   {
      Matematica funcoesMatematicas = new Matematica;
      float      resultado,
                 numero             = 20;

      // calcula a raiz-quadrada do número 10:
      resultado = funcoesMatematicas.raizQuadrada(10);
      System.out.println("A raiz-quadrada de 10 é: " + resultado);

      // calcula a raiz-quadrada da variável número (que vale 20):
      resultado = funcoesMatematicas.raizQuadrada(numero);
      System.out.println("A raiz-quadrada de " + numero + " é: " + resultado);
   }
}
```

Neste último exemplo, o **ProgramaExemplo** chama o método **raizQuadrada** (nas linhas grifadas), passando como parâmetro valores (um número, no primeiro caso, e o conteúdo de uma variável, no segundo). Durante a chamada, quando o método é executado, a Máquina Virtual Java (Java Virtual Machine – JVM) cria dentro dele uma variável chamada **numero** e atribui o valor passado como parâmetro para essa variável. Qualquer comando existente dentro do corpo do método **raizQuadrada** pode acessar e utilizar a variável **numero.** Isso porque a variável declarada como parâmetro é uma variável como outra qualquer do método, e pode ser utilizada dentro dele, em qualquer lugar, normalmente. Por ser uma variável do método, **numero** só pode ser utilizada dentro dele (os outros métodos não a enxergam, mesmo que tenham sido declarados dentro da classe **Matemática**). Quando a execução do método acaba a JVM destrói o parâmetro **numero**.

Apesar dos parâmetros e as variáveis locais de um método não poderem ser acessadas fora dele, e serem muito parecidas por esse motivo, existem diferenças entre o escopo de um parâmetro e o escopo[4] de uma variável local. O parâmetro vale dentro de todo o corpo do método, enquanto que uma variável local vale a partir do local onde ela foi declarada. Veja o exemplo seguinte:

```
01    class Matemática {
02       int soma(int limite){
03          System.out.println("O valor de limite é: " + limite); // OK
04          System.out.println("O valor de total é: " + total); // Erro!
05          int total;     // Problema!
06          for (int i = 1; i<= limite; i++){
07             total += i;
08          }
09          System.out.println("O valor de i é: " + i); // Erro!
10          return total;
11       }
12    }
```

Neste último exemplo a 4ª linha apresenta um erro, pois o programador tentou utilizar a variável **total** e ela ainda não havia sido declarada. **Nunca utilize uma variável local antes dela ter sido declarada.** Também há um erro na linha 9. Neste caso a variável **i** foi declarada dentro de um bloco interno (o bloco do comando **for**) e seu escopo (validade) consiste somente no corpo do comando for, entre o **{** e o **},** logo, na 9ª linha essa variável já não existe mais. **Nunca utilize uma variável de um bloco interno em um bloco externo**.

Variáveis locais também apresentam um outro problema em potencial: elas devem ser explicitamente inicializadas (sempre). Note que na linha 5 a variável total é declarada dentro do método **soma**. Isso significa que ela é uma variável local deste método e só pode ser utilizada dentro dele. Note também que ela não foi inicializada (algo do tipo: **int total = 0**). Isso ocasionaria um problema pois o laço (for) utiliza-a como acumulador do valor da variável **i** e seu valor inicial é desconhecido (e deveria ser zero). **Sempre inicialize suas variáveis locais a fim de evitar erros que podem surgir pela falta de um valor inicial de uma variável**.

Examine agora o comando **return** da linha 10. Esse comando retorna um valor para quem chamou o método **soma**. A sintaxe deste comando é a seguinte:

```
'return' [expressão]';'
```

Um comando de retorno consiste da palavra-chave **return** opcionalmente seguida de uma **expressão**. A menos que o tipo de retorno de um método seja **void** (significando que a função não retorna coisa alguma), você deve sempre colocar após o **return** uma expressão cujo tipo de dado seja igual ao tipo de retorno da função. Além disso, se o tipo de retorno não for **void**, a execução do método deve sempre terminar via um comando **return**. (o comando **return** pode ser colocado em qualquer lugar dentro do método).

Uma vez que o comando return tenha sido executado, para onde é que o programa vai? Se a chamada do método não fizer parte de uma expressão, a execução continua no comando (linha ou declaração) seguinte à chamada do método. Caso contrário, a expressão continua com o operador que avalia os outros operandos da expressão.

[4] **Escopo** é o local ou região onde uma variável é válida.

Especificadores de acesso

Você utilizar um dos seguintes especificadores de acesso na declaração de um método: **public**, **private** ou **protected**. O especificador determina o quão acessível esse método é para o código de outras classes de um programa. O acesso varia de totalmente acessível à totalmente inacessível. O funcionamento é muito similar ao dos atributos, vistos anteriormente.

Se você **não utilizar um especificador de acesso** o Java utiliza o nível de acesso padrão, fazendo com que o método só possa ser chamado por outros métodos declarados na mesma classe e em todas as classes do mesmo pacote.

Se você utilizar o especificador **private** (privado), somente o código contido na mesma classe pode chamar esse método:

```
class Matemática {
    private int fatorial(int numero){
        if(n == 0) return 1; // fatorial de zero é 1
        int produto = 1;
        for(int i = 2; i <= numero; i++) produto *= i;
        return produto;
    }
    int permutação(int n, int r){
       return fatorial(n) / fatorial(n – r);
    }
}
```

Somente os métodos da classe **Matemática** podem chamar o método **fatorial**. O método **fatorial** é um método de ajuda para o método **permutação** porque ele auxilia-o a executar a permutação. Como, neste caso, ele foi feito somente para ajudar o método **permutação**, não faz sentido que outras classes vejam-no e utilizem-no. Quando um método existe somente para ajudar os outros métodos de uma classe ele deve ser declarado privado (**private**). Declarar métodos deste tipo promove o ocultamento de informações não relevantes ou não importantes para quem está fora da classe, além de mantê-la organizada e fácil de dar manutenção.

Se você declarar um método utilizando a palavra-chave **public** o método torna-se acessível para toda e qualquer classe.

Um método **protected** (protegido) é um método com o nível padrão de acesso, com uma diferença: todas as classes filhas da classe na qual ele foi declarado podem acessá-lo.

Modificadores

Os modificadores de método disponíveis em Java são os seguintes: **abstract** ou **final** e/ou **static** e/ou **native** e/ou **synchronized**.

Se você utilizar a palavra-chave **abstract** (abstrato), você estará informando ao Java que a declaração consiste somente em uma assinatura do método. Isso porque, em Java, um método abstrato é um método sem **corpo** (sem bloco de comandos). Um método abstrato não pode ser declarado, simultaneamente, como sendo **final**, nem **static**, nem **synchronized** e nem **private**.

Se você declarar um método **final**, o compilador assume que ele não pode ser sobre-escrito (modificado) em sub-classes (classes-filhas).

Se você declarar um método como sendo **static** (estático), ele será considerado um método de classe e não um método de objetos (algo similar aos atributos de classe e de objetos, que foram discutidos na seção anterior). Neste caso ele só poderá acessar atributos de classe. Se essa palavra-chave não for utilizada, o método é considerado como sendo um método de instância (= de objeto), e poderá acessar qualquer espécie de atributo (de classe ou de objeto).

A palavra-chave **native** (nativo) indica para o Java que o corpo do método foi escrito em C++ e está armazenado em alguma biblioteca específica de plataforma (tal qual uma DLL). Neste caso, somente a assinatura do método deve ser declarada.

Finalmente, se você utilizar o modificador **synchronized** (sincronizada), somente uma thread (linha de execução) por vez pode executar o método.

Métodos de instância

Métodos de instância são os métodos de objetos. Estes métodos podem acessar atributos de instância e atributos de classe declarados na sua classe. Para declarar um método de instância, não especifique a palavra-chave **static**.

Se seu método de instância possuir um parâmetro que tenha o mesmo nome de um atributo de instância da sua classe, seria possível atribuir o valor do parâmetro para o atributo de mesmo nome? Sim, isso pode ser feito da seguinte forma:

```
01 class Empregado {
02    private double salario;
03    void ajustaSalario(double salario){ // observe que salario já existe
                                          // como atributo de classe!
04       this.salario = salario; // coloca o conteúdo do parâmetro no atributo
05    }
06 }
```

Na terceira linha o método **ajustaSalario** declara um parâmetro chamado **salario**. Note que uma variável (um atributo) contendo o mesmo nome já havia sido declarada anteriormente, na linha dois. Nesta situação, para que você possa colocar o valor do parâmetro no atributo (na variável já existente), você tem que utilizar o prefixo **this**. Essa palavra-chave identifica o objeto atual no código-fonte. Quando você utiliza esse prefixo, você diz para o Java: "acesse o atributo pertencente a esse objeto, cujo nome é o seguinte...". O nome do atributo vem após o ponto, como você pode ver na linha quatro do exemplo.

A chamada (utilização) de um método de instância depende se você está em um outro método da própria classe onde ele foi declarado ou em uma outra classe. Se você estiver em um outro método da mesma classe e também for um método de instância, basta você especificar o nome do método e sua lista de argumentos entre parênteses: **ajustaSalário(850.00)**. Exemplo:

```
class Empregado {
   private double salario;
   void ajustaSalario(double salario){
      this.salario = salario;
   }
   void aumentaDezPorcento(void) { // outro método de instância da mesma classe
      double novoSalario = salário + salário * 0.1;
      ajustaSalario(novoSalario);   // chama um método da mesma classe
   }
}
```

Se você, por acaso, necessitar chamar (utilizar) um método que pertence a uma outra classe **ou a um método de classe**, primeiro crie um objeto do tipo da classe que contém o método que você deseja chamar e depois utilize esse objeto para chamá-lo:

```
class ClasseUm {
   private boolean atributoQualquer;
   public void metodo(boolean valor){ // método a ser chamado por outra classe
      atributoQualquer = valor;
   }
}
class OutraClasse {
   void outroMetodo(void){
      ClasseUm objeto = new ClasseUm();
      objeto.método(true);             // chama o método através do objeto
      new ClasseUm().metodo(false);    // chama o método através do objeto
   }
}
```

Esse último exemplo também mostra que você não precisa colocar o objeto criado em uma variável para chamar um método de outro objeto. A expressão **new ClasseUm().método(false);** cria um objeto a partir da ClasseUm e, ao mesmo tempo (implicitamente), usa a referência ao objeto recém criado para chamar **metodo()**.

Similarmente, você também pode encadear chamadas de métodos de instância (desde que elas retornem referências para objetos):

```
class AlôTchau {

   public static void main(String listaDeParametros[]){
      new AlôTchau().alô().tchau(); // chamada encadeada de métodos
   }

   AlôTchau alô() {         // o método alô retorna um objeto do tipo AlôTchau
      System.out.println("Alô");
      return this;          // retorna este objeto
   }

   AlôTchau tchau() {     // o método tchau retorna um objeto do tipo AlôTchau
      System.out.println("Tchau");
      return this;        // retorna este objeto
   }

}
```

Métodos de Classe

Um método de classe é um método que acessa somente os **atributos de classe** definidos na sua própria classe. Para declarar um método deste tipo, especifique a palavra-chave **static:**

```
class Empregado {

   private static int numeroDeEmpregados;

   static void ajustaNumeroDeEmpregados(int numero){
      numeroDeEmpregados = numero;
   }

}
```

Se, no exemplo anterior, o parâmetro **numero** tivesse o mesmo nome do atributo **numeroDeEmpregados**, seria possível utilizar **this** para passar o valor do parâmetro para esse atributo? Veja o exemplo:

```
class Empregado {

   private static int numeroDeEmpregados;

   static void ajustaNumeroDeEmpregados(int numeroDeEmpregados){
      this.numeroDeEmpregados = numeroDeEmpregados; //ERRO!
   }

}
```

Você não pode utilizar **this** em um método de classe porque essa palavra-chave se refere à instância (ao objeto) de classe atual e, infelizmente, um método de classe não é associado a nenhum objeto (instância de classe).

Para chamar um método de classe por outro **método de classe ou de instância da mesma classe**, simplesmente utilize seu nome e sua lista de argumentos entre chaves:

```
class PrimeiroESegundo {
   static void primeiro() {
      System.out.println("Primeiro");
   }
   static void segundo() {
      primeiro();
      System.out.println("Segundo");
   }
}
```

Se a chamada estiver em um **método de classe de outra classe**, prefixe o método com o nome de sua classe seguido do operador de ponto:

```
class UsarPrimeiroESegundo {
   public static void main(String listaDeArgumentos[]) {
      PrimeiroESegundo.primeiro();   PrimeiroESegundo.segundo();
   }
}
```

Se o método da outra classe não for de classe, chame-o da mesma forma como você chamaria um método de instância de outra classe (primeiro crie um objeto e depois chame o método a partir dele).

Sobre a passagem de argumentos

Até o momento nada foi dito sobre como são passados os argumentos (parâmetros) para os métodos de classe ou de instância. Existem duas abordagens para tal: **chamada por valor** e **chamada por referência**.

As **chamadas por valor** fazem uma cópia de cada argumento e passa essa cópia para o método. Assim, o valor original não é modificado. Em Java, somente os **tipos primitivos** (int, double, float...) são passados por valor. Isso significa que, se durante seu processamento, o método realizar alguma mudança nos conteúdos dos parâmetros, as variáveis originais não serão afetadas (no método que realizou a chamada).

Em contraste, as **chamadas por referência** passam somente uma referência ao endereço do argumento original, tornando possível ao método chamar os métodos e acessar os atributos (elementos) deste argumento e modificá-los.

Sobrecarregando métodos

Declarar múltiplos métodos que possuam o mesmo nome, porém com listas de parâmetros diferentes, é uma técnica denominada **sobrecarga de método** (method overloading). Neste caso, o compilador seleciona o método apropriado escolhendo aquele cujo nome, número e tipo de argumentos coincidir com a chamada de um método com o mesmo nome, número e tipo de argumentos (através da comparação de assinaturas). Veja o exemplo seguinte:

```
void imprimir(){
   System.out.println("Impressão padrão");
}

void imprimir(String argumento){
   System.out.println(argumento);
}
```

O exemplo anterior demonstra dois métodos **imprimir** sobrecarregados. Os dois métodos possuem o mesmo nome, porém, parâmetros diferentes. Ao se chamar **imprimir()** o primeiro é chamado (executado). Ao se chamar **imprimir**("Olá"), o segundo é chamado.

Você pode declarar e implementar quantas sobrecargas você achar conveniente. Porém, é importante lembrar que os argumentos devem ser sempre diferentes. Não basta você modificar o tipo de retorno, mantendo o resto da assinatura igual. Veja o exemplo:

```
int somar(int x, int y){
   return x + y;
}

lont somar(int x, int y){
   return x + y;
}
```

Neste último caso o compilador não tem como decidir qual método chamar, e acusará um erro.

Construtores (constructors)

Construtor é um método especial que é chamado pelo operador **new** quando um novo objeto necessita ser criado. Dentro do construtor você pode colocar o seu próprio código "customizado" de inicialização do objeto. Por serem métodos, os construtores podem ser declarados com listas de parâmetros e especificadores de acesso. Entretanto, um construtor **deve ter o mesmo nome da classe em que for declarado**. Além disso, um construtor **não deve ter um tipo de retorno** em sua declaração.

Você se lembra da classe **Empregado?**

```
class Empregado {
   private int numeroDeEmpregados;
   private double salario;

   public void ajustaNumeroDeEmpregados(int numeroDeEmpregados){
      this.numeroDeEmpregados = numeroDeEmpregados;
   }
   void ajustaSalario(double salario){
      this.salario = salario;
   }
}
```

Objetos desta classe podem criados através da utilização do operador **new**:

```
class Exemplo {

   public static void main(String listaDeArgumentos[]){
      Empregado e = new Empregado();
   }

}
```

A linha **Empregado e = new Empregado()** cria um novo objeto do tipo **Empregado** e coloca-o na variável **e**, que também é do tipo **Empregado**. O **Empregado()** que vem depois do operador **new** é um construtor (sempre depois do **new** deve-se colocar um construtor). Note, porém, que **no código da classe Empregado não há a declaração de construtor algum**! O que está acontecendo? Basicamente, quando você não declara um construtor para uma classe o compilador cria um construtor-padrão (default), com uma lista vazia de parâmetros. Olhe o exemplo:

```
class X
{

}
```

Essa classe parece vazia, mas ela não está. Na realidade, o compilador enxerga a classe **X** da seguinte maneira:

```
class X
{

   X ()
   {
   }

}
```

O Compilador somente cria o construtor padrão quando nenhum outro for explicitamente declarado na classe. Logo, se você declarar algum outro construtor, o compilador **não vai gerar** um construtor-padrão sem argumentos:

```
class X
{
   private int i;

   X (int i) // Explicitamente declara um construtor com argumentos
   {
      this.i = i;
   }

}

class UsaX
{
   public static void main(Strings listaDeArgumentos[]){
      X x1 = new X(2);

      X x2 = new X(); // ERRO: não existe construtor sem argumentos na classe X!
   }

}
```

Neste ultimo exemplo, a tentativa de chamar o construtor-padrão (sem argumentos) gera um erro de compilação por que não há um construtor-padrão.

Você pode declarar múltiplos construtores em uma classe. Múltiplos construtores tornam possível a realização de diferentes tipos de inicialização:

```
class Circulo
{

   private int x, y, r;

   Circulo()  // declara um construtor sem argumentos
   {
      // System.out.println("Antes");
      this(0,0,1);
      System.out.println("Depois");
   }
```

```
   Circulo(int x, int y, int r)// declara um construtor que recebe 3 argumentos
   {
      this.x = x;
      this.y = y;
      this.r = r;
   }

   int retornaX() {
      return x;
   }

   int retornaY() {
      return y;
   }

   int retornaRaio() {
      return r;
   }

}

class Exemplo
{
   public static void main(String listaDeArgumentos[])
   {
      Circulo c1 = new Circulo();

      System.out.println("Centro X: " + c1.retornaX());
      System.out.println("Centro Y: " + c1.retornaY());
      System.out.println("Raio    : " + c1.retornaRaio());

      Circulo c2 = new Circulo(5,6,7);

      System.out.println("Centro X: " + c2.retornaX());
      System.out.println("Centro Y: " + c2.retornaY());
      System.out.println("Raio    : " + c2.retornaRaio());

   }

}
```

Este último programa declara um par de classes: **Exemplo** e **Circulo**. A classe **Círculo** declara um par de construtores. O primeiro construtor não recebe argumento algum e contém a seguinte linha de código: **this(0,0,1)**, que nada mais faz do que chamar o segundo construtor que possui 3 argumentos (dentro de uma classe, utilize **this()** ao invés do nome do seu construtor).

Uma chamada ao construtor via o comando **this()** deve ser a primeira linha de código dentro de um construtor (caso contrário o compilador gera um erro). Por esse motivo, a linha **System.out.println("Depois"),** que aparece no primeiro construtor da classe **Circulo,** está comentada. Também, devido a esse mesmo motivo, você não pode chamar dois construtores em um construtor.

Parte III: Composição – Construindo objetos a partir de outros objetos

Suponha que seu instrutor peça para você escrever uma aplicação Java que modele um carro. De acordo com as especificações dele, esta aplicação deve declarar uma classe chamada **Motor** (que modela o motor do carro) e uma classe chamada **Carro** (que modela carros). Finalmente, a aplicação deve declarar uma classe chamada **DemonstracaoCarro**, que possua uma função **main** a fim de executar o programa.

Mas afinal, neste caso, o que é que o método <u>main</u> faz realmente? Neste caso o método **main,** por ser um método de classe especial, fica encarregado de criar objetos do tipo **Carro** (que possuam objetos do tipo **Motor** em cada um deles), colocar informações apropriadas em cada objeto (via chamadas a seus construtores) e (para cada carro) imprimir as informações de cada carro. Depois de um tempo escrevendo essa aplicação, você deve terminar com o seguinte código-fonte:

```
class Motor
{
   private String tipo;
   Motor(String tipo)
   {
      this.tipo = tipo;
   }
}
```

```
class Carro
{
   private String marca;
   private String modelo;
   private String fabricante;

   Carro(String marca, String modelo, String fabricante)
   {
      this.marca = marca;
      this.modelo = modelo;
      this.fabricante = fabricante;
   }

   public String retornaMarca() { return marca; }
   public String retornaModelo() { return modelo; }
   public String retornaFabricante() { return fabricante; }
}

class DemostracaoCarro
{
   public static void main(String listaDeArgumentos[]){
      Motor m1 = new Motor("3.8L V6");
      Carro c1 = new Carro("Mustang", "Conversível", "Ford");

      Motor m2 = new Motor("Zetec Roçam 1.0");
      Carro c2 = new Carro("Ka", "Popular", "Ford");

      System.out.println( c1.retornaFabricante() + " " +
                          c1.retornaMarca()      + " " +
                          c1.retornaModelo()     + " " + m1.retornaTipo() );

      System.out.println( c2.retornaFabricante() + " " +
                          c2.retornaMarca()      + " " +
                          c2.retornaModelo()     + " " + m2.retornaTipo() );
}
```

Muito bem! O código anterior segue as especificações e requisitos solicitados pelo instrutor. Por outro lado, esse código apresenta um problema de modelagem importante: as classes **Carro** e **Motor** estão desconectadas ou desacopladas uma da outra.

No mundo real, é comum dizer que um carro possui um motor, assim como pneus e outros itens. Em outras palavras, vários itens, incluindo um motor, compõem um carro. Em contraste, o código anterior mostra que um carro não compõe um carro e, neste caso, é possível criar objetos do tipo **Carro** sem um motor associado – uma situação estranha. Você já imaginou alguém comprando um carro sem motor!? O erro de se criar um novo **Carro** sem um **Motor** resulta do fato das duas classes estarem desconectadas. A solução para esse erro consiste em conectar estas duas classes. É exatamente esse o assunto desta seção: **composição de objetos**.

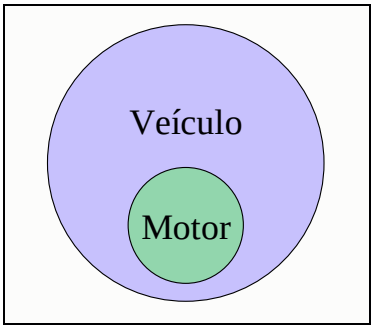
O que é composição?

Em nosso mundo, entidades compõem outras entidades. Só para dar um exemplo, um chassi, algumas portas, pneus, um motor, uma transmissão, um sistema de escapamento, e similares, compõem um veículo. Pelo fato de objetos servirem como representações de software de entidades do mundo real, é possível que vários objetos sejam utilizados para compor um outro objeto. O ato de compor um objeto a partir de outros objetos é chamado de **composição** ou **agregação**.

Analise o código seguinte:

```
class Motor
{
   // atributos e métodos que identificam a estrutura de um motor
}

class Veículo
{
   Motor motor; // define uma variável que armazena um objeto do tipo Motor

   Veículo(Motor m)
   {
      motor = m;
   }
}
```

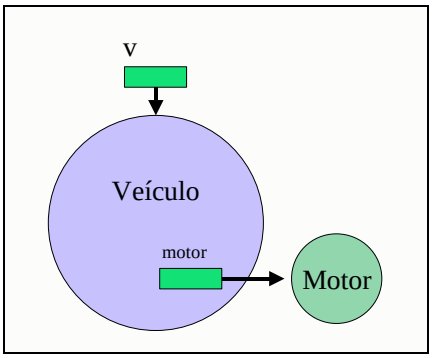
Esse exemplo de código declara duas classes: **Motor** e **Veículo**. A classe **Veículo** declara um atributo chamado motor, que é do tipo **Motor**, e utiliza esse atributo para referenciar um objeto que descreve o motor de um veículo. O construtor **Veículo(Motor m)** inicializa o atributo motor para referenciar um objeto do tipo **Motor** atribuindo o conteúdo do parâmetro **m** para esse atributo. Depois de fazer a atribuição, **motor** possui (referencia, na verdade) o mesmo que **m**. A figura seguinte demonstra a conceitualização um **Veículo** composto de um **Motor**:



Agora que você sabe alguma coisa sobre a estrutura das classes **Veículo** e **Motor**, como você faz para criar um objeto do tipo **Motor** e um objeto do tipo **Veículo** que seja composto do objeto **Motor**? A resposta está no exemplo seguinte:

```
Motor   m = new Motor(); // cria um novo motor e coloca na variável motor
Veículo v = new Veículo(m); // cria um novo veículo, passando o motor para ele
```

No exemplo acima, a primeira linha cria o objeto **Motor** e atribui seu endereço à variável de referencia de objetos **m**, do tipo **Motor**. A segunda linha, que cria o objeto **Veículo**, passa o endereço do objeto (que está na variável **m**) para o construtor do **Veículo**. Com isso, o construtor salva o conteúdo da variável **m** dentro do seu atributo privado **motor**. Neste momento, o objeto **Veículo** referenciado pela variável **v** é composto do objeto **Motor** referenciado pelo atributo interno do **Veículo**. A relação entre os dois objetos recém criados aparece na figura seguinte:

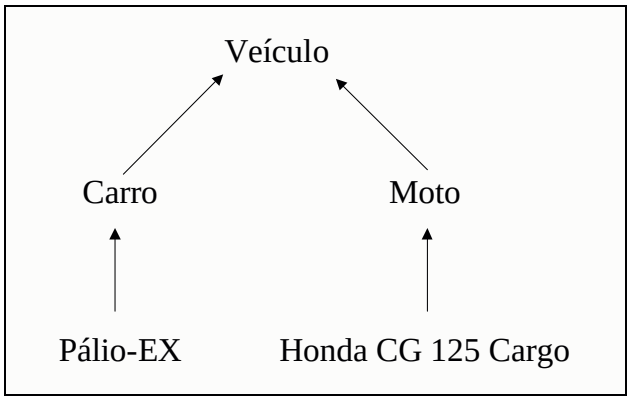


Nesta figura você pode notar que existem dois objetos: o objeto do tipo **Veículo**, que está referenciado na variável **v**, e o objeto do tipo **Motor**, que é referenciado pelo atributo **motor** do objeto **Veículo**. A princípio eles existem em locais diferentes de memória, porém, devido ao atributo **motor**, eles estão ligados, fazendo com que o objeto **Veículo** seja composto por um **Motor**.

Parte IV: Herança – Construindo objetos em camadas

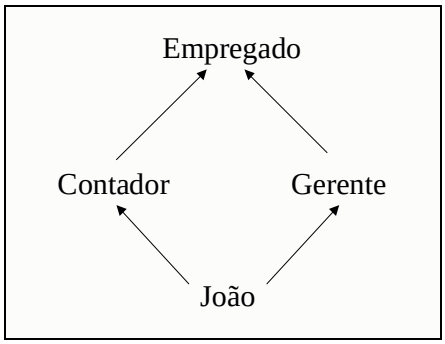
O que é Herança?

Herança é a habilidade de se derivar alguma coisa específica a partir de algo mais genérico. Nós encontramos essa habilidade ou capacidade no nosso cotidiano. Um **Pálio-EX**, estacionado na frente da garagem do seu vizinho, é uma instância específica da categoria **Carro**, mais genérica, por exemplo. Da mesma forma, uma **Honda CG 125 Cargo** é uma instância específica da categoria mais genérica **Moto**. Se levarmos as categorias **Carro** e **Moto** para um outro nível mais elevado, as duas se relacionarão uma com a outra por serem instâncias específicas de uma categoria mais genérica ainda do que elas: a categoria **Veículo**. Em outras palavras, carros e motos são veículos. A figura seguinte esquematiza as relações entre uma entidade **Pálio-EX**, a categoria **Carro**, uma entidade **Honda CG 125 Cargo**, a categoria **Moto**, e a supercategoria **Veículo**.



Entidades as são coisas tangíveis com as quais nós interagimos (o Pálio-EX ou a CG 125, por exemplo). As categorias relacionam entidades com outras entidades, em uma hierarquia: o seu carro (se você tiver um) se relaciona com outros carros através da categoria **Carro**. E os **Carros** se relacionam com **Motos** através da supercategoria **Veículo**. Se utilizarmos uma metáfora de árvore invertida (onde a raiz desta árvore encontra-se no topo e as folhas em baixo), as categorias são os ramos ou galhos da árvore, e as entidades são as folhas desta árvore (conforme você pode ver na figura anterior). A hierarquia de galhos de categorias organiza a forma da árvore (arquitetura de entidades), e as folhas (as entidades) são instâncias dos galhos mais baixos (categorias).

Esse exemplo ilustrou a **herança simples**. Neste caso, uma entidade herda estados e comportamentos (atributos e métodos) de uma e somente uma categoria. O **Pálio**, por exemplo, herda da categoria **Carro** (e somente dela, diretamente). Em contraste, **herança múltipla** permite com que uma entidade herde diretamente comportamentos e estados de duas ou mais categorias ao mesmo tempo. Imagine o **Empregado** chamado **José** de uma empresa qualquer. Para ser mais específico, pense no **José** como sendo **Gerente** e **Contador** ao mesmo tempo. Ele herdaria (possuiria) as capacidades de um **Gerente** e de um **Contador** (por que não?). Veja a próxima figura:



Estendendo classes

Na linguagem Java a herança se manifesta de duas maneiras: ou pela **extensão** de classes (chamada de **herança de implementação**) já existentes ou pela **implementação de interfaces** (**chamada de herança de interface**). **Nesta seção nós vamos estudar somente a herança de implementação.**

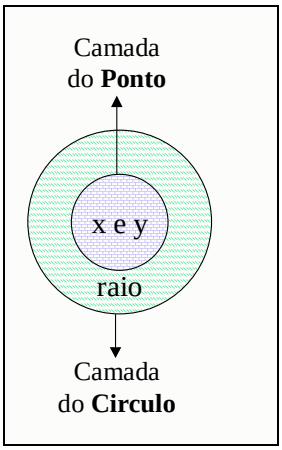
De acordo com a **herança de implementação**, novas classes derivam capacidades (expressas como atributos e métodos) de classes já existentes. Isso faz com que o tempo de desenvolvimento de uma aplicação seja bem menor, pois classes já existentes e comprovadamente funcionais (livres de erros e já testadas) são reaproveitadas (ou reutilizadas). A sintaxe que expressa o conceito de extensão de classes é a seguinte:

```
'class' identificadorDeClasse1 'extends' identificadorDeClasse2
'{'
      // Campos e métodos
'}'
```

Essa sintaxe pode ser lida da seguinte forma: o **identificadorDeClasse1** estende o **identificadorDeClasse2**. Em outras palavras, o **identificadorDeClasse1** herda (ou deriva) capacidades (expressas através de atributos e métodos) do **identificadorDeClasse2**. O **identificadorDeClasse1** é conhecido como **subclasse**, **classe derivada** ou **classe-filha**. O **identificadorDeClasse2** é conhecido como **superclasse**, **classe básica** ou **classe-pai**.

A palavra-chave **extends** (estende) faz com que uma subclasse herde (receba) todos os campos e métodos declarados na classe-pai (desde que ela não seja **final**), incluindo todas as classes-pai da classe-pai. A classe-filha pode acessar todos os atributos e métodos não privados. Ela não pode acessar (ao menos diretamente) os métodos e atributos privados (declarados com a palavra-chave **private**). Considere o exemplo seguinte:

```
class Ponto
{
      private double x, y;

      Ponto(double x, double y)
      {
         this.x = x;
         this.y = y;
      }

      double retornaX()
      {
         return x;
      }

      double retornaY()
      {
         return y;
      }
}

class Circulo extends Ponto
{
      private double raio;

      Circle(double x, double y, double raio)
      {
         super(x, y); // Chama Ponto(double x, double y)

         this.raio = raio;
      }

      double retornaRaio() {
      {
         return raio;
      }

}
```



O código acima consiste de duas classes: **Ponto** e **Circulo**. **Ponto** é uma classe que descreve um ponto, que não é nada mais do que um par de coordenadas (x, y), expresso por valores de ponto-flutuante de dupla-precisão numérica (tipo **double**). **Circulo** descreve um círculo (um círculo é um ponto que possui um raio). Se você fosse criar um objeto a partir da classe **Circulo**, você acabaria com o objeto de multicamadas representado na figura ao lado.

Essa figura mostra que o objeto **Circulo** de duas camadas possui três atributos: **x, y** e **raio**. Entretanto ele somente declara somente o atributo **raio**. Os outros dois atributos são apresentados na camada mais interna, do **Ponto**. As subclasses (classes-filha) são geralmente mais largas do que as superclasses (classes-pai). Isso porque elas costumam adicionar novos atributos ou métodos. Neste caso, por exemplo, a subclasse **Circulo** adicionou um atributo chamado **raio** e um método chamado **retornaRaio**().

Uma subclasse pode estender uma superclasse, desde que a superclasse não tenha sido declarada **final**. Por exemplo, se a declaração da classe **Ponto** tivesse sido **final class Ponto**, o compilador teria acusado um erro. Logo, por razões de segurança, se você quiser garantir que uma classe não vai ser estendida, declare-a como sendo **final**.

A sintaxe da herança sugere que você pode estender uma e somente uma classe. A linguagem Java não suporta a implementação de herança múltipla. Isso porque, segundo os projetistas da linguagem, esse tipo de implementação poderia gerar confusão. Imagine, por exemplo, duas classes-base declarando um atributo que possua o mesmo nome mas com tipos diferentes. Qual dos dois a classe-filha deveria herdar? O mesmo poderia acontecer com um método. Se dois métodos possuíssem o mesmo nome mas diferentes listas de parâmetros ou tipos de retorno, qual deles a subclasse deveria herdar? Para prevenir tais problemas a linguagem Java rejeita a implementação de herança múltipla.

Chamando construtores da classe-pai

No exemplo anterior, examine o construtor da classe **Circulo**. Veja que dentro dele há o seguinte código na linha grifada: **super(x, y)**. Essa linha utiliza a palavra-chave **super** para chamar o construtor **Ponto(double x, double y)** da sua superclasse (classe-pai) e passar para ele o conteúdo das variáveis **x** e **y**. É importante para a classe **Circulo** chamar esse construtor da classe **Ponto** para que os valores de **x** e de **y** inicializem corretamente. Pelo fato da classe **Ponto** declarar estes atributos como sendo privados (**private**), nenhuma classe (nem mesmo sua classe-filha **Circulo**) a não ser a própria classe **Ponto**, pode acessá-los. Logo, para inicializar os valores de **x** e de **y** passados como argumentos para o construtor da classe **Circulo**, essa classe necessita delegar essa tarefa para o construtor da classe **Ponto**, e isso é feito chamando-se **super(x, y**).

Suponha que você modifique o construtor da classe **Circulo** e retire a linha que contém **super(x, y)**; e após você tente compilar o programa. O que aconteceria? O compilador acusaria um erro. Isso porque um **construtor de subclasse deve sempre chamar um construtor de sua superclasse, mesmo que a superclasse não declare explicitamente um construtor**. Se você não colocar uma chamada para o construtor da superclasse, o compilador insere automaticamente na subclasse o código que chama o construtor-padrão (sem argumentos). No exemplo anterior, se você tivesse retirado **super(x, y)** o compilador teria inserido o comando **super()** para chamar o construtor-padrão da classe **Ponto()**. Note que, neste caso, a classe **Ponto** não tem nenhum construtor-padrão, sem argumentos, declarada dentro dela. A classe **Ponto** do exemplo possui somente o construtor que recebe dois argumentos do tipo double: **Ponto(double x, double y)**. E, por esse motivo, um erro seria gerado (lembre-se que quando você não especifica um construtor o compilador gera automaticamente o construtor padrão; mas esse não foi o caso ocorrido na classe **Ponto**).

Existem duas outras regras que devem ser levadas em consideração. A primeira delas diz que **você só pode chamar o construtor de uma superclasse se estiver dentro de um construtor de sua subclasse** (e em mais nenhum método). A segunda diz que, **dentro de um construtor, você nunca pode colocar código algum antes de chamar o construtor da superclasse**.

Sobre-escrevendo métodos (Overriding)

Uma subclasse pode sobrescrever, trocar ou re-escrever um método de sua superclasse. Para que isso ocorra, o método da subclasse deve possuir o mesmo nome, a mesma lista de parâmetros e o mesmo tipo de retorno do método da sua superclasse. Veja o exemplo:

```
class Ponto
{
   void desenha()
   {
      System.out.println("Eu sou um ponto.");
   }
}
class Circulo extends Ponto
{
   void desenha()  // re-declara ou re-escreve o método
   {
      System.out.println("Eu sou um círculo.");
   }
}
```

De acordo com o código presente no exemplo acima, se você criar um novo objeto do tipo **Circulo** e chamar o método **desenha()**, então a frase "**Eu sou um círculo."** irá aparecer:

```
Circulo c = new Circulo();
c.desenha();
```

Entretanto, se a classe Circulo não tivesse re-escrito o método **desenha()** com o seu próprio código, você teria visto "**Eu sou um ponto.**", ao invés.

Agora imagine que você tenha um atributo ou método, em uma subclasse, que seja idêntico a um atributo ou método de sua superclasse. Imagine também que você queira acessar esse atributo ou método que pertence à superclasse e não à subclasse. Como você poderia fazer isso? A resposta você vai encontrar no fragmento de código seguinte:

```
class Super
{
   double x = 2;
   void desenha()
   {
      System.out.println("Super");
   }
}
class Sub extends Super
{
   double x = 3;    // re-declara o atributo
   void desenha() { // re-declara / re-escreve o método
      System.out.println("Sub");
      System.out.println("x = " + x);
      System.out.println("Super.x = " + super.x);
      super.desenha();  // chama o método da superclasse
   }
}
```

O exemplo anterior demonstra um outro uso pra a palavra-chave **super**. Prefixando com "**super.**" um método ou um atributo faz com que o método da superclasse seja chamado ou com que o atributo da superclasse seja utilizado.

Lembre-se, porém, que **um método só pode ser sobre-escrito se ele não tiver sido declarado privado (private) na superclasse**, isto é, ele tem que ser acessível. Da mesma forma, o método não pode ter sido declarado **final**, pois essa palavra-chave faz com que o método não possa ser sobre-escrito.

Nomeando ou especificando o tipo de um objeto (type Casting)

Todo objeto criado a partir de uma subclasse é também um objeto do tipo da sua superclasse (um objeto do tipo **Carro** também é um objeto do tipo **Veículo**, e um objeto do tipo **Circulo** também é um objeto do tipo **Ponto**, conforme nossas declarações anteriores). Essa afirmação implica no fato de você poder atribuir um objeto de uma subclasse para uma referência criada ou declarada para um objeto de sua superclasse.

```
Ponto p = new Circulo(10.0, 20.0, 30.0);
```

A linha de código acima cria um objeto do tipo **Circulo** e atribui sua referência à variável **p**. Note que essa variável **p** é uma variável que armazena referências para objetos do tipo **Ponto**. Esse tipo de atribuição é perfeitamente possível, já que um **Circulo**, pelas definições de classe que foram descritas em exemplos anteriores, é uma subclasse de **Ponto** (e, neste caso, um círculo também é um ponto). Assim, através da variável **p** é possível chamar os métodos **retornaX()** e **retornaY()**, que pertencem ao tipo **Ponto**. Mas seria possível, também, chamarmos o método **retornaRaio()**?

A resposta é **não**, pois quem declara esse método é a camada do **Circulo** e não a camada do **Ponto** (você se lembra que os objetos podem ser construídos em mais de uma camada?). Em essência, **retornaRaio()** é um método que pertence à camada do **Circulo** e não do **Ponto**. Logo, se você declara uma variável do tipo **Ponto** ela enxerga somente a camada existente dentro da classe **Ponto**. Por outro lado, se você declara uma variável do tipo **Círculo** ela enxerga a toda a camada da classe Circulo; e a camada da classe **Circulo** engloba a camada da classe Ponto.

Neste caso, a única maneira de utilizar a variável **p** para acessar o método **retornaRaio()** é utilizando um recurso chamado **type cast**[5], que em Inglês significa algo do tipo **nomeação ou especificação de tipo**. Com esse recurso, uma variável pode assumir momentaneamente outro papel (ou tipo), para que você possa atribuí-la ou utilizá-la em alguma expressão. Em Java, assim como na linguagem C, o **cast** de uma variável ou expressão é realizado colocando-se entre parênteses o tipo desejado. Logo, se no exemplo anterior você quisesse utilizar o método **retornaRaio()** para poder retornar o valor do raio do círculo e colocá-lo em uma variável, você teria que fazer o seguinte:

```
double raio = ((Circulo) p).retornaRaio();
```

[5] A palavra **cast**, em Inglês, pode significar "elenco", que diz respeito à lista de atores de uma peça de teatro, novela ou filme. O elenco tem a ver com o "papel" que determinado ator ou pessoa representa. Analogamente, passando para o contexto de linguagens de programação, o significado da expressão **type cast** seria algo do tipo indicar/mudar momentaneamente o papel (ou o tipo) de determinada variável.

Lembre-se que as operações dentro de parênteses são executadas primeiro. Logo, primeiro o compilador troca o tipo da variável **p** para **Circulo** e depois, com esse resultado (que será um objeto do tipo **Circulo**), ele chama o método **retornaRaio()**. O valor retornado é atribuído à variável chamada **raio**.

É importante lembrar que você pode utilizar esse artifício de nomeação para transformar uma classe-filha em qualquer uma de suas classes-pai. **O inverso, porém, não é permitido**. Veja o exemplo:

```
Ponto  p    = new Ponto(10.0, 20.0);
double raio = ((Circulo) p).retornaRaio();
```

O problema, neste caso, é que um objeto do tipo **Ponto** não tem noção alguma de um método chamado **retornaRaio**. Isso porque quem introduz esse método é a classe **Circulo**. Logo, apesar desse código compilar sem acusar erro, quando você executá-lo, a JVM vai gerar uma exceção (uma falha) do tipo **ClassCastException (Exceção de Nomeação de Tipo)**, ou até mesmo travar, quando a chamada ao método **retornaRaio()** ocorrer.

Herança versus Composição

Herança e composição relacionam-se entre si da mesma forma que um lado se relaciona com o outro em uma moeda. Herança promove um objeto em camadas, como uma cebola. Em contraste, a composição promove objetos que possuem outros objetos, como mingau de aveia em flocos, onde cada floco é um objeto. Na seção (capítulo) anterior, você aprendeu que a composição resulta de relações do tipo "tem um", entre objetos (exemplo: carro **tem um** motor). A herança resulta de relações do tipo "é um". Logo, quando você encontrar um texto ou uma descrição do tipo "um carro **é um** tipo de veículo", você está lidando com herança. Para projetar ou construir objetos complexos você pode utilizar tanto a herança quanto à composição. Uma não exclui a outra, e você pode, inclusive, utilizá-las junto.

Um dos problemas mais difíceis que um projetista pode enfrentar, quando estiver desenvolvendo uma hierarquia de classes, é saber decidir quando utilizar composição e quando utilizar herança. Algumas vezes as relações do tipo "é um" ou "tem um" não são muito claras. Para exemplificar, suponha que você não saiba que um círculo seja um ponto sem raio. Ao contrário, você acredita que um círculo **tenha** um ponto **e** um raio. A partir desta premissa, você pode acabar implementando o seguinte fragmento de código:

```
class Ponto{
      private double x, y;

      Ponto(double x, double y){
         this.x = x;
         this.y = y;
      }

      double retornaX() {
         return x;
      }

      double retornaY() {
         return y;
      }
}

class Circulo{
      private Ponto  p;
      private double raio;

      Circle(double x, double y, double raio){
         p = new Ponto(x, y);

         this.raio = raio;
      }

      double retornaRaio() {
         return raio;
      }

      double retornaX() {
         return p.retornaX();
      }

      double retornaY() {
         return p.retornaY();
      }
}
```

A partir de uma interface ou código externos, um objeto criado a partir desta classe **Circulo** é equivalente ao da classe **Circulo** apresentada no início desta seção. Com este exemplo, pode parecer que não importa se a herança é ou não utilizada. Mas, na verdade, importa: preste muita atenção na classe **Circulo**. Note que há a definição dos métodos **retornaX** e **retornaY**. Estes métodos duplicam o código já existente na classe **Ponto** sem necessidade. Com isso, a reutilização de código não é feita (e uma das grandes vantagens da orientação a objetos é poder reutilizar código). Portanto, se você focalizar somente a composição de objetos você poderá estar adicionando redundância ao código. Em uma escala muito maior, esse tipo de redundância pode se tornar um pesadelo de manutenção.

Parte V: A raiz de todas as classes

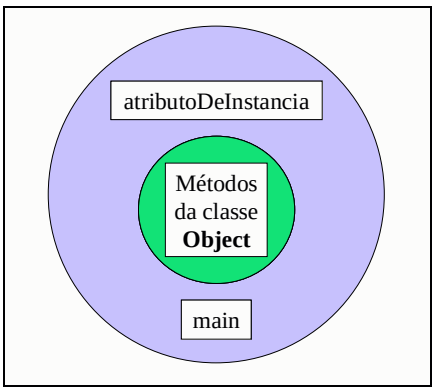
Muitas linguagens orientadas a objetos, como o C++, não suportam a noção de existir uma única classe a partir da qual todas as outras classes são derivadas, sem que isso seja um detrimento da linguagem. Entretanto, na linguagem Java, a falta desse tipo de classe tornaria a linguagem limitada.

A camada "Objeto"

Todos os objetos da linguagem Java são de múltiplas camadas. Cada classe, na hierarquia de classes, representa uma camada que adiciona diversas capacidades a um objeto. No topo desta hierarquia você sempre vai encontrar uma classe chamada de **Objet** (objeto). Imagine, por exemplo, que você tenha declarado uma classe chamada **DemoCamadaObjeto** que não estende, explicitamente, nenhuma outra classe. O código seguinte apresenta uma possível representação desta classe:

```
class DemoCamadaObjeto{

   int atributoDeInstancia = 1;

   public static void main(String listaDeArgumentos[]){
      DemoCamadaObjeto antigo = new DemoCamadaObjeto();
      System.out.println(antigo.atributoDeInstancia);
      System.out.println(antigo.toString());
   }

}
```

À primeira vista, você não nota nada de especial nesta classe. Ela declara um atributo de objeto e um método **main** que cria um objeto do tipo **DemoCamadaObjeto** e imprime o valor do atributo **atributoDeInstancia**. Entretanto, se você examinar com mais calma, você vai notar algo estranho: **antigo.toString()**. De alguma forma, a função **main** chama um método **toString**, mesmo que **DemoCamadaObjeto** não tenha declarado esse método ou tenha especificado uma classe-pai. As aparências podem ser decepcionantes. Na realidade **DemoCamadaObjeto** implicitamente estende a classe **Objet**. É como se o programador tivesse declarado **class DemoCamadaObjeto extends Object**, o que é perfeitamente permitido. Pelo fato de qualquer classe estender implicitamente (sem ser necessário declarar) a classe **Object**, a classe **DemoCamadaObjeto** é uma classe-filha da classe **Object**. Também pelo fato da classe **Object** possuir um método chamado **toString()** (que converte um objeto para uma String), é perfeitamente legal chamar esse método no método **main** do exemplo anterior. Para compreender melhor como o objeto criado a partir de **DemoCamadaObjeto** se parece, observe a figura seguinte.



No centro do objeto **DemoCamadaObjeto**, representado na figura anterior, você vai encontrar uma camada do tipo **Objet** (representada pelo círculo interno). Essa camada interna apresenta alguns métodos que nós vamos examinar em detalhes nesta seção. A camada externa, que representa o objeto **DemoCamadaObjeto,** herda

estes métodos. Além dos métodos herdados, a camada externa também apresenta dois novos componentes: o atributo **atributoDeInstancia** e o método **main**.

Em termos mais gerais, se você tiver uma classe-filha qualquer que estende alguma outra classe-pai qualquer, a classe-pai geralmente estende diretamente a classe raiz **Objet,** e a classe-filha estende-a também, indiretamente. Mas quais são estes métodos herdados por elas? Vamos descobrir!

Obtendo informações sobre uma classe

O primeiro método da classe **Object** que nós vamos estudar é o método **getClass** (retornar classe). Quando chamado, o método **getClass()** retorna uma referencia para um objeto do tipo **Class** (classe). O método **getClasse** é declarado **final**, logo, você não pode sobrescreve-lo em uma subclasse. Um objeto da classe **Class** contém diversos métodos que retornam diversas informações sobre uma classe, tais como seus construtores, atributos e informações sobre métodos.

Um objeto da classe **Class** permite com que o programador determine (descubra) dinamicamente o nome, os atributos, os métodos e etc, de um objeto qualquer. Para descobrir o nome da classe (tipo) de um objeto qualquer utilize o método **getName()**, que retorna uma String que contém esse nome.

O método getClass() não pode ser utilizado em um tipo de dado primitivo, tal como char, int ou float, porque eles não são objetos. Por outro lado, a linguagem Java permite com que você descubra o tipo de qualquer variável através do operador **.class**, que deve ser utilizado como sufixo da variável na qual se deseja descobrir o tipo. Esse operador pode ser utilizado mesmo que você ainda não tenha criado um objeto (já o getClasse exige que você tenha criado um objeto). Exemplo:

```
Class     classe1, classe2, classe3, classe4;
char      letra = 'c';
Empregado e     = new Empregado();

classe1 = Empregado.class; // válido: é um tipo de váriável
classe2 = char.class;      // válido: é um tipo de váriável
classe3 = e.getClass();    // válido: é um objeto
classe4 = Empregado.getClass(); // inválido: não é um objeto
classe5 = c.getClass();    // inválido: não é um objeto
classe6 = c.class;         // inválido: não é um tipo de variável
classe7 = e.class;         // inválido: não é um tipo de variável
```

Logo, quando você estiver trabalhando com um objeto, utilize o método **getClass**. Porém, se você estiver trabalhando com um tipo de variável, utilize o "**.class**". As classes **Object** e **Class** possuem uma série de outros métodos que você pode utilizar. Confira na tabela seguinte alguns destes métodos e para que eles servem:

classe Object		**classe Class**	
Método	**Utilidade**	**Método**	**Utilidade**
getClass()	Retorna uma referência do tipo **Class** que contém informações sobre a classe do objeto.	getName()	Retorna uma String contendo o nome da classe do objeto.
clone()	Retorna um clone (uma cópia) idêntica do objeto (veja seção seguinte).	getSuperClass()	Retorna uma referência do tipo **Class** que contém informações sobre a classe-pai da classe do objeto.
equals(objeto)	Compara o objeto atual com o objeto passado como parâmetro. Se eles forem iguais é retornado um valor **booleano true** (verdadeiro). Caso contrário **false** (falso).		
classe Field (atributo)		getDeclaredField(atributo)	Retorna uma referência do tipo **Field** (atributo) que contém informações sobre o **atributo** de classe solicitado. **Atributo** é uma String que contém o nome do atributo desejado.
Método	**Utilidade**		
getType()	Retorna uma referência do tipo **Class** que contém informações sobre o tipo (classe) do atributo.		

Clonagem

Clonagem é o processo de fazer duplicatas exatas de uma entidade. A linguagem Java suporta a clonagem através do método **clone()**, que pertence à classe **Object**. O método **clone** retorna uma referencia para um objeto que é exatamente a duplicata do objeto que a chama. Durante a operação de clonagem, o construtor do objeto não é chamado. Como resultado, o método **clone** cria uma duplicata de um objeto mais rapidamente que a utilização do comando **new**, seguido de uma chamada ao construtor do objeto.

O método **clone** retorna uma referência para um objeto do tipo **Object**. Logo, antes de colocar esse objeto clonado em uma variável você tem que nomeá-lo (**cast**) para o tipo adequado (tipo do objeto original, que foi clonado):

```
Empregado e1 = new Empregado();   // cria um empregado
Empregado e2 = (Empregado) e1.clone();
```

O método **clone** é um método protegido. Isso significa que somente aquelas classes filhas da classe **Object** podem utiliza-lo, fazendo com que objetos possam clonar a si mesmos, mas não possam clonar outros objetos.

A clonagem de um objeto pode ser superficial (**shallow**) ou profunda (**deep**). O método **clone** sempre realiza uma clonagem do tipo superficial. Na **clonagem superficial**, somente os conteúdos dos, atributos e não os objetos referenciados por estes atributos, é que são clonados.

A **clonagem profunda** envolve, primeiramente, a realização da clonagem superficial. Após, os atributos devem ser atribuídos para referências de novos objetos. Finalmente, estes objetos devem ser populados (preenchidos) com informações duplicadas (clonadas).

Igualdade

Comparar duas variáveis para verificar se o seu conteúdo é igual e comprar dois objetos para verificar se eles são iguais são tarefas diferentes. Quando você quer comparar o conteúdo de duas variáveis, você utiliza o operador de igualdade (==).

O que o operador de igualdade faz é comparar se o conteúdo de duas variáveis é igual. Se seus conteúdos forem iguais, ele retorna o **valor Booleano true** (verdadeiro). Se elas forem diferentes, o valor **false** é retornado. Esse operador funciona muito bem com tipos primitivos de dados (int, float, double...).

Já nos outros tipos de variáveis, quando você estiver trabalhando com objetos, é errado utilizar esse operador. Isso porque essas variáveis não contêm os objetos em si, mas, sim, referências para estes objetos. Logo, o comparador de igualdade estaria comparando duas referências (endereços) e não o conteúdo dos objetos.

Para comparar dois objetos você deve utilizar o método **equals**. Veja a diferença entre os dois:

```
String nome1 = "Leonardo";
String nome2 = "Leonardo";

System.out.println("nome1 == nome2     : " + nome1 == nome2);       // false
System.out.println("nome1.equals(nome2): " + nome1.equals(nome2) ); // true
```

Se você utilizar uma das classes já existentes da linguagem Java, o método equals funcionará muito bem. Porém, **se você criar uma nova classe ele não vai funcionar corretamente**. Isso porque o método **equals** padrão utiliza, internamente, o operador de comparação (==). **Para que ele funcione corretamente você deve redefinir (reescrever) este método dentro da sua classe** de modo que ele possa realizar a comparação correta.