Curso de Java Básico

Fábio Mengue – fabio@unicamp.br
Centro de Computação - Unicamp

História Rápida da Linguagem

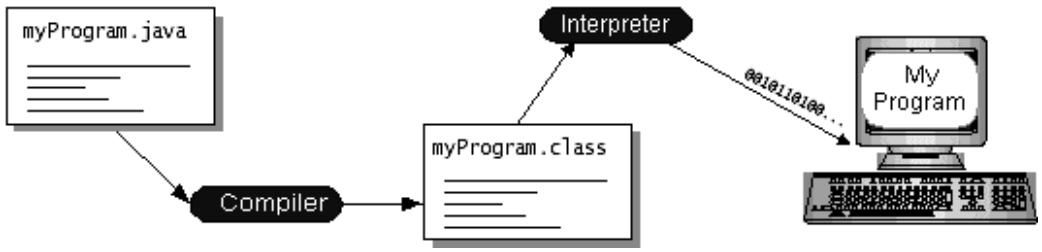
Em 1991, um grupo de engenheiros da Sun Microsystems foi encarregado de criar uma nova linguagem que pudesse ser utilizada em pequenos equipamentos como controles de TV, telefones, fornos, geladeiras, etc. Essa linguagem deveria dar a esses aparelhos a capacidade de se comunicar entre si, para que a casa se comportasse como uma federação. Deveria ainda ser capaz de gerar códigos muito pequenos, que pudessem ser executados em vários aparelhos diferentes, e praticamente infalível.

Os engenheiros escolheram o C++ como ponto de partida. Orientada a objetos, muito poderosa e gerando pequenos programas, parecia a escolha correta. Para solucionar o problema da execução em várias arquiteturas, eles utilizaram o conceito da máquina virtual, onde cada fabricante iria suportar algumas funções básicas que os programas utilizariam.

Até hoje a linguagem resultante deste projeto não é utilizada em aparelhos eletrodomésticos. Ao invés disso, o Java se tornou um das linguagens de programação mais utilizadas no planeta.

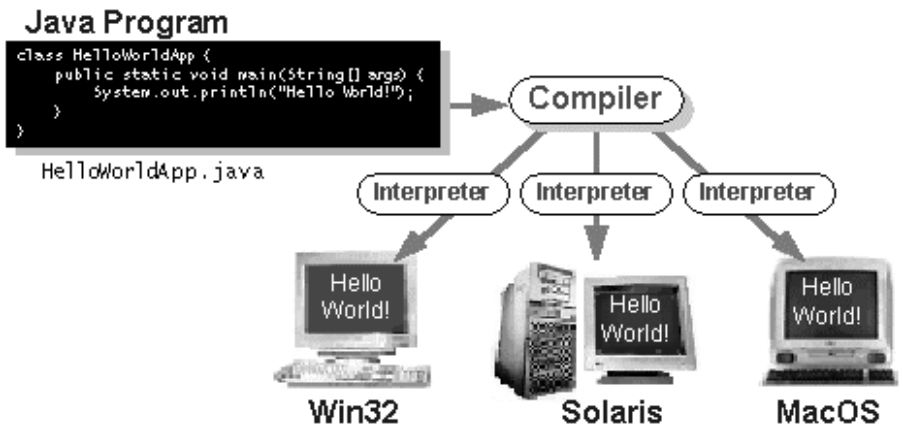
Vantagens do Java (Por Que Estudar Essa Linguagem ?)

Na maioria das linguagens de programação, você precisa compilar ou interpretar um programa para que ele seja executado em seu computador. A linguagem Java é diferente, pois seus programas são compilados **E** interpretados. Com o compilador, você inicialmente transforma seu programa em uma linguagem intermediária, chamada *bytecode*. Esse código é independente de plataforma, e é mais tarde interpretado por um interpretador Java. A compilação acontece apenas uma vez; a interpretação acontece todas as vezes que seu programa é executado. A figura abaixo mostra como isso acontece.



Você deve pensar nos *bytecodes* como instruções de máquina para a *Java Virtual Machine* (ou JVM). Todos os produtos que conseguem executar programas em Java (como um browser que executa applet's) possuem uma cópia da JVM.

Bytecodes Java tornam possível a tecnologia "escreva uma vez, execute em qualquer lugar". Você pode compilar seu programa Java em qualquer plataforma que possua um compilador. Os *bytecodes* gerados podem ser interpretados em qualquer plataforma que possua uma JVM. Veja na figura abaixo:



A Plataforma Java

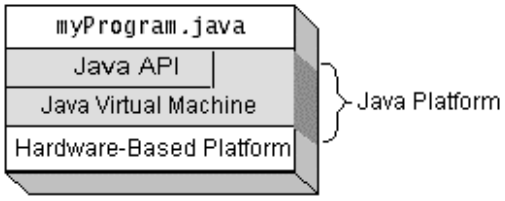
Por plataforma, entendemos o conjunto de hardware e software no qual um programa executa. Alguns exemplos de plataformas muito usadas são o Windows, o Linux, o MacOS. A plataforma Java é diferente, pois não envolve hardware; ela utiliza a plataforma de hardware das outras.

A plataforma Java tem dois componentes:

- Java Virtual Machine (Java VM ou JVM)
- Java Application Programming Interface (Java API)

A API Java é uma coleção de componentes de software prontos, que incluem desde estruturas para manipulação de arquivos até a construção de aplicativos gráficos. A API é organizada como um grupo de bibliotecas com classes e interfaces; essas bibliotecas são chamadas de pacotes.

A figura abaixo mostra o esquema de funcionamento da plataforma Java:



Erros Comuns de Conceito Sobre Java

Muita gente pensa muita coisa sobre Java, e muita coisa errada. A seguir listamos algumas idéias erradas mais comuns.

Java é uma linguagem fácil de aprender.

Nenhuma linguagem poderosa como o Java é fácil de aprender. É sempre fácil escrever programas do tipo “Alo Mundo”. Aprender a lidar com if, while e tipos do Java é uma tarefa simples; a parte complexa vem da orientação a objetos e das classes presentes na linguagem. Temos mais de 1.500 classes e interfaces diferentes. A descrição de cada uma cabe em um livro de 600 páginas.

O ambiente Java facilita a programação.

Você vai aprender Java usando o notepad e o prompt do DOS. Não é o melhor ambiente do mundo, especialmente se comparado às linguagens visuais que temos hoje, como o Visual Basic. Em um ambiente de alta produtividade, o uso do Java deve ser muito racional, levando em conta as dificuldades da codificação.

Java será a linguagem universal no futuro.

É possível, em teoria. Mas existem muitos sistemas com códigos nativos que são perfeitos da maneira que estão hoje, e por isso não devemos “mexer”. O Java não é recomendado para solucionar todos os problemas.

Java é apenas mais uma linguagem como qualquer outra.

A linguagem Java, apenas pelo fato de permitir que um programa seja executado em qualquer plataforma, já realizou revolução suficiente. Além disso, a linguagem foi desenhada para se utilizar da rede, e os conceitos de ambiente multitarefa auxiliam o produto a ter poucas comparações no mercado hoje.

Todos os programas Java tem que ser executados dentro de um navegador.

Um dos usos do Java é a confecção de applets. Java também serve para programar aplicativos, servlets, JavaBeans, componentes e uma grande gama de produtos.

Javascript é uma versão simplificada do Java.

Javascript é uma linguagem utilizada em navegadores. Ela foi inventada pela Netscape, e sua sintaxe é semelhante à da linguagem Java. Com exceção do nome, as semelhanças terminam ai.

Instalando o ambiente

Para o desenvolvimento de aplicativos utilizando o Java, é necessário a instalação do compilador Java, das API's e da JVM. A instalação do ambiente segue o mesmo esquema da instalação de qualquer produto para Windows.

Devemos fazer o download da versão mais apropriada via ftp ou http e executar o arquivo, para que o produto se instale.

As versões para Windows, Linux e Solaris pode ser obtido em:

http://java.sun.com

Meu primeiro programa Java

Como a maioria das linguagens de programação, o fonte de seu programa em Java deve ser criado a partir de um editor de texto que gere arquivos em formato ASCII. É possível utilizar editores como o Word e o Wordpad, mas o texto deve ser salvo sem formatação. O editor ideal é o notepad.

O programa fonte em Java deve ser salvo obrigatoriamente com a extensão .java. Salve no notepad o arquivo utilizando aspas duplas, assim:

"AloMundo.java"

Vamos então criar uma pasta chamada CURSO (md curso) para que possamos organizar os programas e exercícios que faremos.

Execute o Notepad agora, e vamos digitar nosso primeiro programa em Java. Copie as linhas abaixo:

```
class AloMundo {

  public static void main(String args[]) {

  System.out.println("Alo Mundo !");

  }
}
```

Não se preocupe em entender o código; é apenas um exemplo, e explicaremos esses comandos mais tarde. Salve o arquivo como AloMundo.java (letras maiúsculas e minúsculas são importantes).

A seguir, vamos compilar o programa. A compilação irá gerar os *bytecodes*. Do prompt do DOS, execute:

javac AloMundo.java

Se não houver erro, depois de alguns segundos você deve ter acesso ao prompt novamente. Para executar o programa,

java AloMundo

Você deve ter recebido como resposta a frase “Alo Mundo !”. Isso significa que tudo está certo com seu ambiente e você acabou de criar seu primeiro programa em Java.

Exercícios

−Altere o programa AloMundo para que ele imprima seu nome completo.
−Altere o programa AloMundo e faça com que ele imprima seu nome em duas linhas separadas.

Objetos

O Que São Objetos ?

Quando temos um problema e queremos resolve-lo usando um computador, necessariamente temos que fazer um programa. Este nada mais é do que uma serie de instruções que indicam ao computador como proceder em determinadas situações. Assim, o grande desafio do programador é estabelecer a associação entre o modelo que o computador usa e o modelo que o problema lhe apresenta.

Isso geralmente leva o programador a modelar o problema apresentado para o modelo utilizado em alguma linguagem. Se a linguagem escolhida for LISP, o problema será traduzido como listas encadeadas. Se for Prolog, o problema será uma cadeia de decisões. Assim, a representação da solução para o problema é característica da linguagem usada, tornando a escrita difícil.

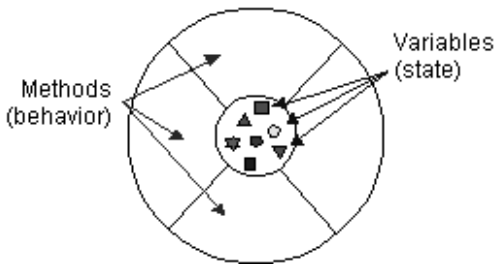
A orientação a objetos tenta solucionar esse problema. A orientação a objetos é geral o suficiente para não impor um modelo de linguagem ao programador, permitindo a ele escolher uma estratégia, representando os aspectos do problema em objetos. Assim, quando se "lê" um programa orientado a objeto, podemos ver não apenas a solução, mas também a descrição do problema em termos do próprio problema. O programador consegue "quebrar" o grande problema em pequenas partes que juntas fornecem a solução.

Olhando à sua volta é possível perceber muitos exemplos de objetos do mundo real: seu cachorro, sua mesa, sua televisão, sua bicicleta. Esses objetos tem duas características básicas, que são o estado e o comportamento.

Por exemplo, os cachorros tem nome, cor, raça (estados) e eles balançam o rabo, comem, e latem (comportamento). Uma bicicleta tem 18 marchas, duas rodas, banco (estado) e elas brecam, aceleram e mudam de marcha (comportamento).

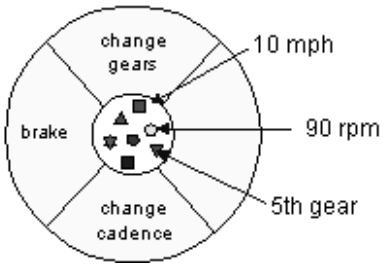
De maneira geral, definimos objetos como um conjunto de variáveis e métodos, que representam seus estados e comportamentos.

Veja a ilustração:



Temos aqui representada (de maneira lógica) a idéia que as linguagens orientadas a objeto utilizam. Tudo que um objeto sabe (estados ou variáveis) e tudo que ele podem fazer (comportamento ou métodos) está contido no próprio objeto.

No exemplo da bicicleta, poderemos representar o objeto como no exemplo abaixo:



Temos métodos para mudar a marcha, a cadência das pedaladas, e para brecar. A utilização desses métodos altera os valores dos estados. Ao brecar, a velocidade diminui. Ao mudar a marcha, a cadência é alterada.

Note que os diagramas mostram que as variáveis do objeto estão no centro do mesmo. Os métodos cercam esses valores e os "escondem" de outros objetos. Deste modo, só é possível ter acesso a essas variáveis através dos métodos. Esse tipo de construção é chamada de encapsulamento, e é a construção ideal para objetos.

Exercício

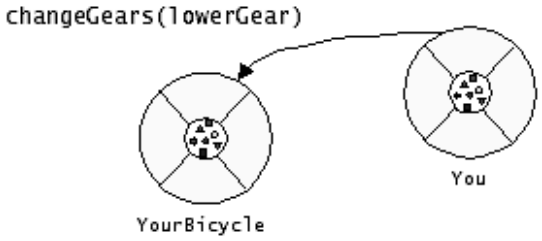
- Imagine um exemplo de objeto diferente dos vistos até agora. Para esse objeto, relacione possíveis métodos e atributos.

Mas um objeto sozinho geralmente não é muito útil. Ao contrário, em um programa orientado a objetos temos muitos objetos se relacionando entre si. Uma bicicleta encostada na garagem nada mais é que um pedaço de ferro e borracha. Por si mesma, ela não tem capacidade de realizar nenhuma tarefa. Apenas com outro objeto (você) utilizando a bicicleta, ela é capaz de realizar algum trabalho.

A interação entre objetos é feita através de mensagens. Um objeto "chama" os métodos de outro, passando parâmetros quando necessário. Quando você passa a mensagem "mude de marcha" para o objeto bicicleta você precisa dizer qual marcha você deseja.

A figura a seguir mostra os três componentes que fazem parte de uma mensagem:

- objeto para o qual a mensagem é dirigida (bicicleta)
- nome do método a ser executado (mudamarcha)
- os parâmetros necessários para a execução do método.



A troca de mensagens nos fornece ainda um benefício importante. Como todas as interações são feitas através delas, não importa se o objeto faz parte do mesmo programa; ele pode estar até em outro computador. Existem maneiras de enviar essas mensagens através da rede, permitindo a comunicação remota entre objetos.

Assim, um programa orientado a objetos nada mais é do que um punhado de objetos dizendo um ao outro o que fazer. Quando você quer que um objeto faca alguma coisa, você envia a ele uma "mensagem" informando o que quer fazer, e o objeto faz. Se ele precisar de outro objeto que o auxiliar a realizar o "trabalho", ele mesmo vai cuidar de enviar mensagem para esse outro objeto. Deste modo, um programa pode realizar atividades muito complexas baseado apenas em uma mensagem inicial.

Você pode definir vários tipos de objetos diferentes para seu programa. Cada objeto terá suas próprias características, e saberá como interpretar certos pedidos. A parte interessante é que objetos de mesmo tipo se comportam de maneira semelhante. Assim, você pode ter funções genéricas que funcionam para vários tipos diferentes, economizando código.

Vale lembrar que métodos são diferentes de procedimentos e funções das linguagens procedurais, e é muito fácil confundir os conceitos. Para evitar a confusão, temos que entender o que são classes.

Exercícios
- Imagine dois objetos que se relacionam. Mostre como é possível através de mensagens o relacionamento entre eles.

Classes

Até agora, comentamos sobre o que é um objeto e como ele se relaciona com outros objetos. Falaremos agora a respeito de classes.

Sua bicicleta é apenas uma das várias que existem. A sua bicicleta específica foi feita por um fabricante, e tem certas características que seguem uma linha ou modelo. Assim, é possível ir em uma loja e comprar várias unidades de bicicletas essencialmente idênticas. Mas não podemos dizer que todas são o mesmo objeto, simplesmente por serem parecidas.

Em orientação a objetos, dizemos que um (ou mais) objetos pertencem a uma *classe* quando eles são muito semelhantes, de acordo com certos padrões. Assim, dependendo da situação, alguém pode dizer que sua bicicleta é um veículo, pois ele anda sobre rodas e carrega uma pessoa. Se seguirmos esse padrão, uma motocicleta também é um veículo, assim como um automóvel ou até um avião (ué, não anda sobre rodas quando decola ou pousa ? Não carrega uma pessoa ?). Na verdade, quando programamos em linguagens orientadas a objetos, antes de tudo nós construimos as classes, os modelos, para depois usá-las na criação de objetos daquele tipo.

Voltando ao nosso exemplo das bicicletas. Quando elas são construídas, os fabricantes tiram vantagem do fato que elas tem as mesmas características, e podem construir várias delas a partir de uma mesma planta. Seria muito custoso fazer um novo projeto para cada bicicleta que eles fossem construir.

Exercícios
- Agrupe em classes os objetos abaixo:

avião elefante computador tigre calculadora motocicleta formiga

Na orientação a objetos, quando programamos, tentamos visualizar os objetos necessários para resolver um problema. Se eu estou criando um programa de controle de estoque, eu vou necessitar de uma classe que represente os ítens que tenho nesse estoque. Outra classe que represente as ordens de compra. E assim por diante.

Nota: A classe é um modelo ou protótipo, que define as variáveis e comportamentos em comum que um certo tipo de objeto específico contém.

No nosso exemplo, depois de criada a classe bicicleta, nos é possível criar a partir dela objetos que serão cópias idênticas da classe, mas cada uma delas terá seus próprios estados. O modelo pode fornecer dados que serão compartilhados entre todos os objetos ou permitir que estes tenham dados própiros.

Nota: Objetos “derivados” de uma classe são chamados instância de classe.

Para tornar o entendimento mais fácil, vamos ver um exemplo mais fácil e prático. Inicialmente vamos pensar em um objeto bem simples, como uma lâmpada.

As lâmpadas podem ter diversos formatos, cores e tamanhos, mas basicamente elas tem apenas uma função, que é iluminar o ambiente. Assim, imagine uma classe lâmpada.

Essa classe pressupõe apenas 2 estados. Um objeto derivado dessa classe pode estar ligado ou desligado. Mas a lâmpada por si só não decide quando vai ligar ou desligar. Quem liga ou desliga esse objeto lâmpada é um interruptor, acionado normalmente por uma pessoa.

Pare resolver esse problema, teremos que definir um objeto baseado na classe lâmpada. Teremos ainda um objeto da classe interruptor e outro da classe pessoa. Pessoa irá enviar uma mensagem ao interruptor que por sua vez irá enviar uma mensagem para a lâmpada.

Uma coisa interessante é que não interessa para a pessoa que está ligando a lâmpada como o interruptor funciona: importa que quando pressionado, a lâmpada acenda. Esta é uma das grandes vantagens de se utilizar o conceito de objetos em informática. Um objeto é como uma caixa preta: não me importa como ele faz as coisas, apenas o que tenho que fazer para que as coisas acontecam.

Voltando ao exemplo da lâmpada, em Java eu poderia definir uma lâmpada assim:

```
lampada sala_de_aula = new lampada();
```

Se eu quiser ligar a lâmpada, é só informar a ela o meu desejo, através de uma mensagem:

```
sala_de_aula.ligar();
```

E a lâmpada vai acender. Não me importa como ela fez isso, me importa apenas que ela faça.

Sintaxe

Não basta um objeto estar definido para ser utilizado. Se eu pretendo usar um objeto, eu preciso cria-lo, e atribuir o controle desse objeto a um nome. A criação de objetos em Java é sempre necessária, e é chamada de instanciação (por isso o objeto é chamado de instância).

Quando eu crio o objeto sala_de_aula, eu digo que ele é do tipo lâmpada e que eu quero que ele seja instanciado nesse momento (new lampada()). A partir de agora, sala_de_aula é um objeto do tipo definido e tem como funções tudo que uma lâmpada pode fazer.

Classes, Atributos e Métodos

Uma das grandes vantagens do Java e nosso propósito de estudo é a capacidade de você definir seus próprios objetos. Como já comentado, um objeto contém atributos (dados) e métodos (funções para manipulação dos dados). Esses objetos irão ajudá-lo a resolver o problema apresentado e tornarão seu trabalho mais fácil.

Vamos inicialmente criar um objeto simples, que contenha apenas dados.

```
class meuObjeto
{
   String nome;
   int idade;
   String telefone;
}
```

Você acabou de definir um objeto chamado meuObjeto. Temos nome, idade e telefone como dados deste objeto.

Mas não basta defini-lo para que ele nos seja útil. Para isso, você deve instanciá-lo,

assim:

```
meuObjeto amigo = new meuObjeto();
```

A partir de agora, meu objeto amigo pode ser utilizado para guardar dados. Eu posso incluir dados assim:

```
amigo.nome = "Joao";
amigo.idade = 33;
amigo.telefone = "2223311";
```

E quando eu precisar dos valores, eu simplesmente os imprimo.

Nota: para facilitar nosso aprendizado, o comando para imprimir um dado é System.out.println, e pode ser utilizado assim:

System.out.println (amigo.nome);

Vamos agora criar um método para o meuObjeto. Esse método vai se chamar aniversario, e ele serve para aumentar em 1 ano a idade do objeto. Então, meu objeto ficaria:

```
class meuObjeto
{
   String nome;
   int idade;
   String telefone;

   public void aniversario()
   {
      idade = idade + 1;
   }
}
```

Agora, para um teste, poderíamos fazer:

```
meuObjeto amigo = new meuObjeto();
amigo.nome = "Joao";
amigo.idade = 33;
amigo.telefone = "2223311";
System.out.println ("Idade antiga"+amigo.idade);
amigo.aniversario();
System.out.println ("Nova idade"+amigo.idade);
```

Percebam que o valor da idade foi alterado.

Exercício
- Digitem o exemplo acima e façam o mesmo funcionar.

Nota: o esquema geral para a definição de um método é

tipo_do_dados_de_retorno **nome_do_metodo (*argumentos*)**
{
  corpo_do_metodo
}

No caso do método aniversario definido acima, não temos nenhum tipo de retorno, por isso ele é do tipo void.

Se precisarmos algum resultado de retorno, temos que indicar qual o tipo desse resultado.

Vamos definir outro método, que me retorna o numero de meses baseado na idade. O método poderia ser definido assim:

```
int idadeEmMeses()
{
  return (idade * 12);
}
```

A instrução return vai indicar qual o valor a ser retornado na chamada do método. Como temos um retorno, ele dever ser utilizado ou apresentado. Podemos apresentá-lo assim:

```
System.out.println (amigo.idadeEmMeses());
```

Nota: Se o valor for utilizado para outros fins, devemos definir uma variável do mesmo tipo do retorno para conter o resultado:

```
int idade_em_meses = amigo.idadeEmMeses();
```

Às vezes, necessitamos do envio de argumentos (também chamados parâmetros) para um método para que ele possa executar seu trabalho. A passagem de parâmetro é feita na hora da chamada, e temos que criar o método já levando em consideração a quantidade de parâmetros que iremos passar.

Assim, caso eu queira alterar ao atributo idade do objeto, eu poderia criar um método assim:

```
void alteraIdade(int nova_idade)
{
```

```
  idade = nova_idade;
}
```

E a chamada ao método ficaria:

```
  amigo.alteraIdade(30);
```

Exercício
- Implemente o exemplo acima.

Nota: Caso eu tenha mais de um argumento, eles devem ser separados por virgulas.
Nota2: Na definição do método com mais de um argumento, eu tenho que prever as variáveis a serem recebidas.

Exemplo:
void qualquerCoisa (String nome, int idade, String telefone)

A chamada é

amigo.qualquerCoisa ("paulo", 24, "2221133");

Construção De Programas Em Java

Vocês podem estar se perguntando qual a finalidade de criar métodos para alterar valores dentro de um objeto. Nesses exemplos fica fácil perceber que é muito mais fácil fazer uma atribuição simples (amigo.idade=34) do que criar um método só para alterar a idade. Mas isso tem sentido de ser, em linguagens orientadas a objetos.

A idéia é que o objeto deve gerenciar seus próprios dados, que só devem ser acessíveis ao "mundo exterior" através de seus métodos (excetuando-se aqui os métodos e variáveis estáticas). Então, pelo menos em teoria, cada atributo do meu objeto deve ter um método para gravar dados e outro para devolver o dado gravado. Isso vai permitir que esse objeto seja utilizado por qualquer um, a qualquer tempo.

Vamos passar então um pequeno código de um programa completo em Java para que possamos ir comentando e esclarecendo.

```
import java.util.*;
public class Propriedades
{
  public static void main (String[] args)
  {
    System.out.println ("Bom dia... Hoje é dia\n");
    System.out.println(new Date());
   }}
```

Logo na primeira linha temos "import...". É normal nas primeiras linhas de um programa em Java a colocação da instrução import. Essa instrução serve para que nosso programa possa utilizar qualquer classe externa que ele necessite. No caso, meu programa usa uma função que retorna a data do sistema, e essa função faz parte de java.util. Assim, eu preciso importar essas funções para que meu programa funcione. Isso será explicado melhor quando falarmos a respeito de API's.

Na segunda linha, definimos o nome desse objeto (em Java, chamado de classe). É importante notar que o objeto deve ter o mesmo nome do arquivo no disco, caso contrário o programa não irá funcionar.

Notamos ainda nessa mesma linha que essa classe é do tipo publica (public). Isso quer dizer que esse objeto é acessível por outros objetos, que ele pode ser importado e usado.

Na próxima linha, temos a definição do método main. Vemos novamente o public, e a palavra static, indicando que esse método não pode ser instanciado, apenas usado do jeito que esta. Vemos que o método main admite uma lista de Strings como argumentos (chamado de args). Esses argumentos não são necessários nesse programa, mas caso necessitássemos de passar algum tipo de informação para o programa, essas informações estariam armazenadas na variável args.

Na próxima linha, temos a impressão de um texto na tela. A única novidade é a presença de um sinal \n no final do texto. Ele indica ao Java para pular de linha depois de escrever o texto indicado.

Na ultima linha temos outra impressão. Dessa vez, temos a instanciação de um objeto do tipo Date, e seu valor é imediatamente impresso. Como não necessitamos do objeto, apenas queremos um valor impresso, não é necessário criar uma variável apenas para isso. Note que nem sempre isso é possível.

Métodos Construtores e Overloading

Como já apresentado, é sempre necessário instanciar um objeto para poder utilizá-lo. Existe um método especial em uma classe que fornece instruções a respeito de como devemos instanciar o objeto. Esse método é chamado de construtor.

A função do construtor é garantir que o objeto associado à variável definida será inciada corretamente. Sempre é necessário ter um construtor, e como na maioria das vezes esse construtor não faz nada (além de instanciar o objeto), não é necessário declara-lo. O Java faz isso automaticamente para nos.

Nota: O método construtor tem exatamente o mesmo nome da classe. Assim, no exemplo:

```
class meuObjeto
{
   String nome;
   int idade;
   String telefone;
   public void aniversario()
   {
      idade = idade + 1;
   }
}
```

Não temos construtor definido.

Mas existem casos onde teremos necessidade de um construtor que faz algo, como na definição de String. Eu posso definir uma String assim:

```
String nome = new String();
```

Ou assim:

```
String nome = new String ("Joao");
```

Isso quer dizer que o objeto String tem pelo menos 2 construtores; um que inicia o objeto sem argumentos e outro que inicia com argumentos. Esse tipo de artificio é chamado de sobrecarga (em inglês, Overloading).

Se eu quiser que meuObjeto tenha o mesmo tipo de funcionalidade do String, eu defino dois construtores, assim:

```
class meuObjeto
{
   String nome;
   int idade;
   String telefone;
   meuObjeto()               // Esse é o construtor sem argumentos
   {
   }
   meuObjeto(String _nome, int _idade, String _telefone)  // Construtor com argumentos
   {
      nome = _nome;
      idade = _idade;
      telefone = _telefone;
   }
   public void aniversario()
   {
      idade = idade + 1;
   }
```

}

Agora meuObjeto pode ser instanciado como

meuObjeto amigo = new meuObjeto();

ou

meuObjeto amigo = new meuObjeto("Joao", 32, "2223311");

A sobrecarga é um dos recursos mais interessantes da orientação a objetos. E não esta restrito aos construtores; podermos definir o mesmo nome de método para qualquer método. Assim, tornamos o programa mais legível e temos menos nomes para "inventar".

Nota: Como os dois (ou mais) métodos tem o mesmo nome, a diferenciação de qual método é executado depende da quantidade e do tipo dos argumentos enviados.

Nota2: Quando não definimos construtores, o Java cria um sem argumentos para nos. Quando eu escolho definir o construtor, tenho que definir para todos os tipos, inclusive o sem argumentos.

Utilização das API's

Como já falado, existem muitos objetos prontos que a linguagem Java nos proporciona. Como todos os objetos, eles podem conter dados e tem vários métodos para que possamos usa-los. É importante para um programador Java conhecer o máximo desses objetos que puder; eles facilitam o trabalho na medida que evitam que nos tenhamos o trabalho de inventar algo que já está pronto. Lembre que normalmente nós não temos acesso aos atributos dessa classe; apenas a seus métodos.

Quando executamos um import ... no inicio do nosso programa, estamos informando ao compilador Java quais as classes que desejamos usar.

Assim, um import java.util.*; quer dizer: "eu vou utilizar alguns objetos do pacote java.util". Um programa pode ter tantos import quanto necessário.

Isso permite que você utilize componentes de pacotes baixados da Internet, com utilização mais restrita.

Nota: Uma lista completa das API's existentes pode ser achada na lista de links no final do documento.

Conceito De Pacote

Um pacote pode ser entendido como uma serie de objetos agrupados por afinidade. Eles ficam "juntos" pois tem funções semelhantes (como por exemplo manipular texto).

Quando nós vamos criar nossos próprios objetos para resolver um problema, normalmente esses objetos ficam todos no mesmo sub-diretorio; entre eles estabelecemos uma relação de "amizade", e podemos considera-los como parte do mesmo pacote (default). Estando no mesmo sub-diretorio, certas restrições de acesso entre esses objetos mudam. Explicamos restrição de acesso no tópico a seguir.

Tipos De Métodos: Públicos, Privados E Protegidos

As restrições de acesso em Java existem por dois motivos. O primeiro é não permitir que o programador utilize meus objetos da maneira que quiser. Eu dito o que pode e o que não pode ser utilizado. E o segundo motivo é que tudo que não for publico (e consequentemente não permite acesso) pode ser alterado na hora que eu quiser. Contanto que os métodos de acesso (que são públicos) fiquem inalterados, ninguém vai perceber a diferença.

Como exemplo, vejamos o objeto String. O String na verdade é um vetor de caracteres. Eu não tenho permissão para acessar os conteúdos internos desses caracteres; eu tenho que fazer isso utilizando um método. Assim, existem situações onde é interessante que meus objetos tenham um certo controle sobre o que o programador pode fazer com eles, para que não ocorram problemas. Para isso, temos três tipos de acesso a métodos e atributos.

Quando utilizamos a palavra public, liberamos o acesso do atributo/método para ser utilizado por qualquer um que importe o pacote ou o objeto.

No caso do private, ninguém pode acessar o método/atributo daquele objeto, nem mesmo os objetos daquele pacote ou objetos que herdam suas características. Isso evita que certos métodos sejam acessados diretamente, evitando erros.

Métodos/atributos protected podem ser herdados, mas não alterados. Iremos comentar a respeito de herança mais à frente.

Exemplo:

```
class exemplo {
        public metodo1() { ... }
        private metodo2() { ... }
}
```

Nessa classe, o metodo1 pode ser executado por qualquer outro objeto. O método metodo2 só pode ser executado pelo próprio objeto.

Composição e Herança

Um dos conceitos mais interessantes das linguagens orientadas a objeto é a reutilização de código. Mas para isso realmente funcionar, você tem que conseguir fazer mais do que simplesmente copiar código e altera-lo. Você deve ser capaz de criar uma nova classe usando outra classe já existente.

Existem duas maneiras diferentes de fazer isso. Uma é chamada composição. A composição é geralmente utilizada quando se deseja utilizar as características de um objeto mas não sua interface.

Quando eu tenho um objeto do tipo carro e uma outra pessoa vai utilizar um carro com as mesmas funcionalidades, é mais fácil ela construir um carro utilizando as "peças" (que já estão prontas). Assim, ela pode importar a classe carro e usar:

```
carro meuCarro = new carro();
```

A partir de agora, dentro do seu objeto, você tem um objeto do tipo carro. O que você fez aqui foi compor um objeto. Seu objeto agora é do tipo composto, já que ele possui mais do que um objeto dentro dele.

Mas existe situações onde a composição não basta. É quando eu desejo utilizar o objeto existente para criar uma versão melhor ou mais especializada dele. Isso é chamado herança.

Imagina que eu tivesse uma classe chamada Empregado. Essa classe tem como atributos o nome, seção e salário do empregado. Existe também um método para alterar o salário.

```
class Empregado
{
	public Empregado (String _nome, String _secao, double _salario)
	{
		nome = _nome;
		secao = _secao;
		salario = _salario;
	}

	public void aumentaSalario (double percentual)
	{
		salario *= 1 + percentual / 100;
	}
	String nome;
	String secao;
	double salario;
}
```

Vamos supor agora que temos que necessitamos de um tipo especial do objeto Empregado, que é o Gerente. O Gerente tem secretária, e a cada aumento ele recebe a mais 0,5% a título de gratificação.

Mas o Gerente continua sendo um empregado; ele também tem nome, seção e salário. Assim, fica mais fácil utilizar a classe Empregado já pronta como um modelo, e aumentar as funcionalidades. Isso é chamado de herança.

Veja a classe Gerente no exemplo abaixo:

```
class Gerente extends Empregado
{
	public Gerente (String _nome, String _secao, double _salario, String _secretaria)
	{
		super (_nome,_secao,_salario);   // Aqui eu chamo a super classe do Gerente
		secretaria = _secretaria;
	}

	public void aumentaSalario (double percentagem)
	{
		super.aumentaSalario (percentagem+0,5);
	}

	public String getSecretaria ()
	{
		return (secretaria);
	}

	public void setSecretaria (String _secretaria)
	{
		secretaria = _secretaria;
	}

	private String secretaria;
}
```

O que fizemos aqui foi literalmente nos aproveitarmos do código de outro programa, especializando o objeto Empregado. Adicionamos métodos a essa classe e mesmo alteramos um de seus métodos (aumentaSalario) para refletir uma nova condição.

Uma palavra que apareceu nessa classe foi a *super*. Ela referencia a classe da qual essa se originou, a classe que foi estendida ou herdada. Assim, super.aumentaSalario invoca o método aumentaSalario da classe Empregado. A classe Gerente poderia ter substituído completamente o método ou fazer alterações, como fizemos.

Assim, é sempre uma boa idéia sempre pensar em construir objetos que possam ser genéricos o suficiente para que possam ser reaproveitados.

Nota: Como referencia para nos auxiliar a determinar se devemos utilizar composição ou herança para construir um objeto, sempre pense:

é para herança (um carro é um veiculo).
contém para composição (um carro contém motor, freio, etc.).

Nota 2: Quando não desejamos que um método ou atributo seja redefinido, utilizamos a palavra reservada *final*.

Polimorfismo

O conceito de herança nos leva a discutir outro: o polimorfismo. Podemos traduzir essa palavra pela capacidade de um objeto em saber qual o método que deve executar. Apesar da chamada ser a mesma, objetos diferentes respondem de maneira diferente.

Assim, quando chamamos aumentaSalario da chasse Gerente, é esse método que será executado. Se a classe Gerente não tivesse esse método, o método da classe Empregado seria executado. Caso a classe Empregado também não tivesse esse método, a classe da qual ele veio seria objeto do pedido.

Como exemplo, imagine uma classe chamada Figura. Essa classe tem a habilidade de desenhar a si mesma na tela. Se eu definir uma classe chamada Triangulo que estenda a classe Figura, ela pode usar o método da super classe para desenhar a si mesma, sem necessidade de criar um método apenas para isso.

Usando ainda nosso exemplo de Funcionário e Gerente. Se a classe Funcionário tivesse um método para mudar a seção do funcionário, não seria necessário definir um método igual para a classe Gerente. Entretanto, quando eu invocasse o método Gerente.musaSecao(nova_secao), o objeto saberia como se comportar, pois ele herdou essa “sabedoria” de sua superclasse.

Nota: Como todos os objetos definidos são subclasses de *Object*, todas as classes que usamos ou definimos, por mais simples que sejam, tem certas capacidades herdadas desta classe. Para maiores detalhes, veja a API.

Exercícios
-Implemente o exemplo acima. Altere ou defina métodos usando o mesmo nome (sobrecarga), defina atributos e métodos private e public e teste sua segurança.

Métodos Estáticos

Um último conceito importante diz respeito a métodos especiais. Eles foram criados para resolver situações especiais na orientação a objetos, e também para simplificar certas operações.

Imagine uma classe, criado por outra pessoa, que tenha apenas um método. A classe se chama validaCPF, e serve para verificar se um CPF é valido ou não.

Segundo as "leis" da orientação a objetos, eu preciso instanciar um objeto desse tipo e utilizar seu método para a verificação do valor. Isso é um tanto quanto incômodo, pois eu simplesmente necessito de uma funcionalidade, e não do objeto todo. Da mesma maneira, temos dezenas de funções matemáticas, físicas, estatísticas e outras que não nos interessam. Gostaríamos apenas de enviar os parâmetros e receber resultados, como se esses métodos fossem funções.

Nesses casos (e em alguns outros) podemos criar esse método como estático (static). Um método estático presente em uma classe não obriga a instanciar um objeto para que eu tenha acesso a seus serviços; ele serve apenas para que possamos aproveita-lo em pequenas computações. Assim, a definição do validaCPF seria:

```
static boolean validaCPF(String numero_cpf)
{
  código...
}
```

Quando eu precisasse utilizar o código, eu faria algo do tipo:

```
boolean cpf_valido = validaCPF("123123123");
```

Nota: Os métodos estáticos são utilizados em casos específicos. Um programa orientado a objetos que é feito inteiramente de métodos estáticos não é orientado a objetos :)

Exercício
- Crie uma classe que possua um método estático chamado **Soma**. Esse método deve receber dois parâmetros e devolver o resultado da soma destes parâmetros. Use sobrecarga de métodos para que os parâmetros possam ser tanto do tipo inteiro (int) quanto do tipo fracionário (double).

Apêndice A – Programação Fundamental

Nessa seção falamos sobre os conceitos básicos do Java, como tipos de dados, loops, e métodos.

Como não é trivial escrever programas que utilizam a interface gráfica, iremos inicialmente aprender como criar aplicativos, ou programas que não utilizam outros recursos além do prompt do DOS.

Comentários

É sempre interessante a colocação de comentários em programas. Os comentários permitem que a manutenção posterior do código seja mais rápida e serve para indicar o que o programa faz. Os comentários em Java podem ser de dois tipos. Utiliza-se duas barras (//) em qualquer posição da linha. Tudo o que aparecer à direita das duas barras será ignorado pelo compilador. Exemplo:

```
a++;  // Incremento da variável
```

Existem ocasiões onde várias linhas de comentário são necessárias. Nesse caso, utilizamos os sinais de /* e */ para indicar início e fim de bloco de comentários, como no exemplo:

```
/* Programa de Exemplo – Esse programa não faz nada.
Criado por Fábio Mengue em Outubro de 2000
Versao 1.0
*/
```

Tipos de Dados

O Java é uma linguagem que necessita que todas as variáveis tenham um tipo declarado. Existem 8 tipos primitivos em Java. Seis deles são numéricos, um é o caracter e o outro é o booleano. Os tipos inteiros guardam valores numéricos sem parte fracionária. Valores negativos são permitidos.

Tipo	**Tamanho**	**Faixa de Valores**
Int	4 bytes	-2.147.483.648 até 2.147.483.647
Short	2 bytes	-32.768 até 32.767
Byte	1 byte	-128 até 127
Long	8 bytes	-9.223.372.036.854.775.808 até 9.223.373.036.854.775.807

Na maioria das ocasiões, o tipo int é suficiente. Não se pode esquecer que como Java é portável, esses valores são os mesmos para qualquer plataforma de hardware.

Os tipos primitivos que representam valores com ponto flutuante:

Tipo	**Tamanho**	**Faixa de Valores**
Float	4 bytes	+/- 3.40282347E+38F (aproximandamente 7 dígitos significativos)
Double	8 bytes	+/- 1.79769313486231570E+308 (15 dígitos significativos)

Normalmente utilizamos o double na maioria das situações onde é necessária a representação desse tipo de número, pois sua precisão é maior.

Nota: Números do tipo float aparecem com um sufixo F. (3.04F). Se o número aparecer sem o F, ele é considerado double.

O tipo caracter serve para representar apenas uma letra ou número.

Tipo	**Tamanho**	**Faixa de Valores**
Char	2 bytes	0 até 65536

Esse tipo serve para representar caracteres usando a tabela Unicode. Dessa tabela faz parte a tabela ASCII e mais alguns caracteres especiais.

Nota: O tipo caracter é sempre representado por aspas simples (‘h’). Caracteres representados por aspas duplas (“h”) na verdade são strings.

O tipo booleano pode assumir apenas dois valores, *true* ou *false*. Esse tipo é usado apenas para testes lógicos.

Tipo	**Tamanho**	**Faixa de Valores**
Boolean	1 bit	true ou false

Declaração de Variáveis

A declaração de variáveis em Java, como em várias outras linguagens, exige que o tipo da variável seja declarado. Você inicia a declaração indicando o tipo da variável e o nome desejado, como no exemplo:

int a;

```
byte b;
char ch;
```

Note que todas as declarações terminam com o ponto-e-vírgula. Os nomes das variáveis devem ser iniciados com qualquer letra, seguidas por uma seqüência de letras ou dígitos. O tamanho do nome da variável não tem limites.

É possível declarar várias variáveis em uma linha, bem como atribuir valores a elas na declaração, como nos exemplos abaixo:

```
int a,b;
int  a = 10;  // Isto é uma inicialização
```

Conversões Entre Tipos

Java não tem problemas para atribuir um tipo int para um double – ele vai tratar o valor como double. Assim sempre que uma atribuição for efetuada o tipo mais representativo será utilizado.

Entretanto, existem ocasiões onde queremos representar o valor inteiro de um tipo double, por exemplo. Assim, torna-se necessário converter o tipo, em uma operação chamada de *cast*. Essa conversão nada mais é do que indicar o tipo desejado, como no exemplo:

```
double x = 9.345
int z = (int)x;
```

A variável z terá como valor o número 9.

As conversões permitidas sem *cast* são:

byte->short->int->long->float->double e char-> int

Constantes

Você pode definir constantes em Java utilizando a palavra reservada *final*. Essa palavra indica que você definiu o valor da uma variável e que esse valor não pode ser modificado. Normalmente as constantes são definidas em caixa alta, como no exemplo:

```
final double TEMPERATURA = 25.4;
```

Operadores

Os operadores aritméticos + - * / são utilizados para a adição, subtração, multiplicação e divisão. A divisão retorna resultado inteiro se os operadores forem inteiros, e valores de ponto flutuante em caso contrário. Se for necessário ter o valor do resto da divisão, utilizamos o % (função mod).

É possível utilizar operadores na atribuição das variáveis, como no exemplo:

```
int n = 5;
int a = 2 * n;  // a = 10
```

Existe também a possibilidade de utilizar atalhos para operações:

```
x += 4;  // equivalente a x = x + 4;
```

A exponenciação é feita por uma função da biblioteca matemática. Essa biblioteca tem dezenas de operações específicas.

```
double y = Math.pow (x, b);  // x é elevado a b (x^b)
```

Incremento e Decremento

O uso de contadores em programas é muito comum. Existem maneiras de realizar incrementos e decrementos em variáveis utilizando o sinal ++ e o --. Veja nos exemplos:

```
int a = 12;
a++   // a agora vale 13
```

O uso do incremento e do decremento depende da posição onde eles se encontram na expressão. Existem ocasiões em que quero a expressão calculada e o valor seja incrementado depois. Em outros casos, o valor deve ser incrementado e a expressão avaliada ao final. Acompanhe o exemplo:

```
int m = 7;
int n = 7;
int a = 2 * ++m;  // a vale 16, m vale 8
int b = 2 * n++;   // b vale 14, n vale 8
```

Operadores Relacionais e Booleanos

Esses operadores servem para avaliar expressões. Para verificar a igualdade entre dois valores, usamos o sinal == (dois sinais de igual).

O operador usado para verificar a diferença (não igual) é o !=. Temos ainda os sinais de maior (>), menor (<), maior ou igual (>=), menor ou igual (<=).

Existem operadores lógicos AND (&&), OR (||).

Strings

Todos os outros valores que utilizamos em Java com exceção dos tipos explicados acima (ditos primitivos) são objetos. Um dos objetos mais utilizados é o String (com S maiúsculo);

O String é uma sequencia de caracteres.

```
String e = “”;  // String vazia. Note as aspas duplas.
String oi = “Bom dia”;
```

As strings podem ser concatenadas, utilizando o sinal de +, como no exemplo:

```
String um = “Curso”;
String dois = “Java”;
String result = um + dois;
```

Nota: Uma String não deve ser comparada com outra usando o sinal ==, pois elas são objetos. Existe um método especial para comparar objetos, utilizando o *equals*. Assim, a comparação da String a com a String b seria:

```
a.equals(b);
```

Nota 2: O objeto String em Java tem mais de 50 métodos diferentes.

Vetores e Matrizes

Vetores são estruturas utilizadas para armazenar dados afins. Esses dados podem ser de qualquer tipo, desde variáveis primitivas até objetos complexos.

Um vetor pode ser definido assim:

```
int[] vetor = new int[100];
```

Aqui temos um vetor de 100 posições (de 0 a 99) de valores inteiros. Os elementos do vetor podem ser acessados segundo sua posição:

```
vetor[30]
```

E podemos iniciar seus valores na própria inicialização, como abaixo:

```
int[] impares = {2,3,5,7,9,11,13}
```

É possível definir vetores de várias dimensões. É muito usado em matemática o conceito de matriz, que em Java é definida como:

```
int[][] matriz = new int[5][6];
```

E os valores são acessados da mesma maneira que os vetores.

Controle De Fluxo Do Programa

Apresentamos aqui os comandos que nos permitem controlar o fluxo do programa e expressões condicionais em Java. Mas antes temos que aprender a delimitar blocos e conceituar o escopo.

Um bloco nada mais é uma séries de linhas de código situadas entre um abre e fecha chaves ({ }). Podemos criar blocos dentro de blocos. Dentro de um bloco temos um determinado escopo, que determina a visibilidade e tempo de vida de variáveis e nomes. Por exemplo:

```
{
  int x = 10;
  // aqui eu tenho acesso ao x
  {
     int z = 20;
     // aqui eu tenho acesso ao x e ao z
  }
  // aqui eu tenho acesso ao x; o z esta fora do escopo
}
```

Assim, é permitido a definição de variáveis com mesmo nome, desde que elas não estejam compartilhando o mesmo escopo. A definição dos blocos ajuda a tornar o programa mais legível e a utilizar menos memória, além de indicar quais os comandos a serem executados pelas instruções condicionais e os loop, que veremos a seguir:

```
if (expressão)
  comando ou { bloco }
else                  // opcional
  comando ou { bloco }      // opcional
```

Desvia o fluxo de acordo com o resultado da expressão. A expressão pode ser algo simples ou composto. O else é opcional. Se for necessário mais de um comando, é necessário colocar o bloco das instruções entre { } .

return

O comando return serve para 2 propósitos: mostrar qual valor deve ser retornado do método (se ele não for void) e para encerrar a execução do método imediatamente.

Os comandos que utilizamos para executar a mesma porção de código várias vezes são chamados de comandos de iteração, ou comandos de loop.

```
while (expressão)
  comando ou { bloco }
```

A expressão é avaliada uma vez antes do comando. Caso seja verdadeira, o comando é executado. Ao final do comando, a expressão é avaliada novamente. Se for necessário mais de um comando, é necessário colocar o bloco das instruções entre { } .

```
do
  comando ou { bloco }
while (expressão);
```

O comando é executado, e a expressão é avaliada no final. A única diferença entre o do-while e o while é que no primeiro o comando é sempre executado pelo menos uma vez. Se for necessário mais de um comando, é necessário colocar o bloco das instruções entre { } .

```
for (inicialização; expressão; passo)
  comando ou { bloco }
```

Uma variável é iniciada na parte de inicialização. A expressão é testada a cada execução do comando, e enquanto for verdadeira, a(s) instrução(es) contidas no bloco é (são) executada(s). Ao final, passo é executado.

Nota: É possível a inicialização de mais de uma variável e a execução de mais de uma instrução no passo, dividindo as instruções com virgulas, como abaixo:

for (int i=0, j=1; i < 10 && j != 11; i++, j++)

break

O comando termina a execução de um loop sem executar o resto dos comandos, e força a sadia do loop.

continue

O comando termina a execução de um loop sem executar o resto dos comandos, e volta para o inicio do loop para uma nova iteração.

```
switch (variável)
{
case (valor1): comando ou { bloco } break;
case (valor2): comando2 ou { bloco2 } break;
...
default: comando_final ou { bloco final }
}
```

O comando switch serve para simplificar certas situações onde existem vários valores a serem testados. Assim, identificamos a variável a ser testada, e colocamos uma linha case para cada possível valor que a variável pode assumir. No final, nos é permitido colocar uma linha default para o caso da variável não assumir nenhum dos valores previstos. O break no final de cada comando serve para evitar comparações inúteis depois de encontrado o valor correto. Se for necessário mais de um comando, é necessário colocar o bloco das instruções entre { } .

Outras instruções

Ainda existem centenas de outras instruções que podem ser utilizadas em Java. A maioria delas faz referência a métodos de objetos e classes, que existem para realizar operações específicas.

Mas não precisamos nos limitar às funções definidas pela linguagem. Temos a possibilidade de criar nossas próprias funções, utilizando procedimentos simples para realizar tarefas complexas (tradicionalmente chamadas de funções). Além disso, podemos ainda estender a API, com classes específicas de outros fabricantes.

Apêndice B - Objetivos do Desenho da Linguagem Java

Simples – A linguagem Java é na verdade uma versão mais "limpa" do C++. A idéia era que a linguagem deveria evitar consumir um grande tempo para o treinamento de programadores; entretanto, ela deveria utilizar as técnicas mais modernas de construção de software.

Orientada a Objetos – Há trinta anos o conceito de orientação a objetos existe na programação. Hoje em dia, ele é sinônimo de modernidade, eficiência e extensibilidade, em um universo cujas expectativas mudam muito rapidamente.

Familiar – O Java se manteve o mais perto possível do C++. Removendo suas complexidades e mantendo sua sintaxe, é possível a uma grande gama de programadores iniciar diretamente a programação nessa linguagem.

Robusta – Java foi criada para desenhar programas confiáveis. O interpretador verifica continuamente a execução dos programas, protegendo o sistema de erros. A linguagem também evita que vícios prejudiciais por parte dos programadores possam causar instabilidade no sistema operacional. Não é necessária a alocação de memória, e uma série de erros de bibliotecas podem ser descobertos imediatamente, na própria compilação.

Segura – A tecnologia do Java foi desenhada para utilizar extensivamente a rede e os ambientes distribuídos. Nessas arquiteturas, segurança é um dos parâmetros principais. Um aplicativo em Java não pode ser invadido via rede, pois suas restrições de segurança não permitem acessos não autorizados.

Neutralidade – Java foi criada para funcionar em uma grande variedade de plataformas de hardware. seus *bytecodes* permitem a criação de um programa em qualquer plataforma e sua execução em qualquer plataforma.

Portabilidade – A neutralidade de arquitetura é apenas um dos pontos que indicam a portabilidade de um sistema. Além disso, o Java uniformiza os tipos de dados nas diferentes arquiteturas, de modo que um inteiro num PC representa a mesma quantidade de bits em uma estação de trabalho. Assim, um programa Java é totalmente independente de hardware e software.

Alta Performance – A performance de um programa Java é relacionada estritamente à performance do interpretador. A JVM Java permite que se execute o código do usuário na máxima velocidade possível; todas as outras tarefas ficam em segundo plano. Ainda assim, se for necessário uma performance ainda maior, é possível compilar o *bytecode* para código nativo da máquina.

Multi-Tarefa – Sistemas orientados à rede necessitam executar várias tarefas ao mesmo tempo. Java permite a construção de um modelo onde podem ser executadas *threads* concorrentes. Esse modelo tem controle de concorrência, sincronização e monitoramento presentes na própria linguagem. Além disso, o Java tem um sistema *threadsafe* que evita qualquer tipo de conflito entre as várias tarefas.

Apêndice C – Dicas para a Construção de Classes

Sempre mantenha seus dados como private.

É a dica principal. Se seus atributos forem todos *public*, você estará violando o encapsulamento. Você vai necessitar de métodos para alterar os valores, o que dá um pouco mais de trabalho, mas a experiência mostra que o formato dos dados pode ser alterado, mas a forma de alterá-los muda muito pouco.

Sempre inicialize seus dados.

Java não fará a inicialização de variáveis locais para você. Não confie nos valores padrão, sempre prefira fazê-lo você mesmo.

Não utilize muitos tipos básicos em uma classe.

A idéia aqui é substituir os tipos básicos que são relacionados entre si por outras classes. Como exemplo:

```
private String rua;
private String cidade;
private String estado;
private String cep;
```

Nesse caso, teremos uma maior legibilidade usando uma classe chamada endereco, que contenha todos esses atributos.

Nem todos os atributos necessitam de métodos.

Imagine uma classe que represente os empregados da sua empresa. Você provavelmente não necessita ter um método para alterar a data de contratação de uma pessoa, uma vez que o objeto está construído. Então isso pode ser resolvido no constructor, sem a necessidade de um método.

Use definições padrão para suas classes.

Sempre use um padrão. Isso torna muito mais fácil a manutenção e entendimento de código. Normalmente, o padrão usado em Java é:

Escopo público
Escopo de pacote
Escopo privado

E em cada uma dessas seções,

Constantes
Construtores
Métodos
Métodos Estáticos
Instâncias de variáveis
Variáveis Estáticas

Divida classes que estão muito complexas.

Essa dica é vaga, pois a complexidade depende de quem olha. Entretanto, se houver oportunidade de dividir uma grande classe em duas ou mesmo três, devemos aproveitar a oportunidade. Mas claro que isso tem um limite: dividir sua aplicação em 10 classes de 1 método cada normalmente vai fazer seu programa ser mais lento.

Faça o nome de suas classes e métodos fazer sentido.

Assim como as variáveis devem ter nomes que representam o dado que elas contém, classes devem seguir o mesmo princípio. Uma convenção usada é que a classe deve ser um substantivo seguido de um adjetivo ou de um gerúndio. A convenção dos métodos propõe que eles devem ser iniciados por letras minúsculas, e cada palavra envolvida com o método tem sua primeira letra maiúscula (como em converteTemperatura). Quando estamos falando de métodos cuja função é recuperar dados, iniciamos o método usando a palavra *set*, e quando o método recupera os dados, usamos *get* (setSalario, getSalario).

Apêndice D – Erros Mais Comuns e Suas Soluções

Problemas de Compilador

Erro: Bad command or file name (Windows 95/98) ou
The name specified is not recognized as an internal or external
command, operable program or batch file (Windows NT)

Caso essa mensagem de erro apareça quando da execução do javac ou do java, isso quer dizer que o seu sistema não está encontrando esses programas. O mais comum é a falta da informação no PATH do computador.

Para solucionar o problema, procure ajuda no menu do sistema operacional relacionado a como alterar ou adicionar elementos no PATH.

Erros de Sintaxe

Se você não digitou corretamente alguma instrução do seu programa, o compilador vai indicar o erro. A mensagem normalmente mostra o tipo do erro, a linha onde o erro foi detectado, o código naquela linha e a posição do erro na linha. Segue uma amostra:

```
testing.java:14: `;' expected.
System.out.println("Input has " + count + " chars.")
                                                ^
1 error
```

O erro indica a falta de um “;” no final da linha.

Às vezes podem ocorrer erros onde o compilador não consegue entender o que você pretendia, e mensagens de erro confusas são geradas. No exemplo abaixo, também temos a falta de um “;”, mas as mensagens são um tanto quanto diferentes:

```
while (System.in.read() != -1)
count++
System.out.println("Input has " + count + " chars.");
```

Ao processar esse código, teremos duas mensagens de erro:

```
testing.java:13: Invalid type expression.
            count++
                  ^
testing.java:14: Invalid declaration.
             System.out.println("Input has " + count + " chars.");
                        ^
2 errors
```

O compilador processa o count++, e como não há o sinal “;”, ele pensa que ainda esta avaliando uma expressão. Assim temos a primeira mensagem. A segunda acontece quando o compilador tenta “juntar” a primeira expressão com a segunda, e ao perceber que não existe sentido, apresenta a outra mensagem.

Quando erros sintáticos acontecem, o *bytecode* não é gerado. É necessário corrigir o problema e tentar novamente.

Erros Semânticos

Além dos erros sintáticos, existem também os erros semânticos. Por exemplo, se você tentou incrementar uma variável que não foi iniciada:

```
testing.java:13: Variable count may not have been initialized.
        count++
        ^
testing.java:14: Variable count may not have been initialized.
    System.out.println("Input has " + count + " chars.");
                                  ^
2 errors
```

Erros de Execução

Um erro comum é tentar executar uma classe utilizando-se da extensão. Quando quero compilar o programa, faço

```
javac MeuProg.java
```

Quando vou executá-lo, faço

```
java MeuProg
```

Não é necessária a extensão. Se o usuário a especificar, uma mensagem do tipo

```
Can't find class MeuProg.class
```

Outro problema comum é a criação de uma classe sem o método main(). Toda classe dita “executável” deve possuir esse método. A falta dele gera uma mensagem assim:

```
In class MeuProg: void main(String argv[]) is not defined
```

E por último, não esquecer que é necessário recompilar o programa a cada alteração. Se você altera seu código mas parece que isso não surte efeito, o problema pode estar aí.

Bibliografia

http://java.sun.com/docs/white/langenv/
Thinking In Java – Eckel, Bruce. Prendice Hall PTR. (www.phprt.com, www.bruceeckel.com)
Just Java – Linden, Peter Van Der. Makron Books, SunSoft Press.
Core Java 2 – Fundamentals – Horstmann, Cay S. – Cornell, Gary. The Sun Microsystems Press.