JAVA 1

Introdução Prática

ESTE MÓDULO OFERECE UMA VISÃO GERAL DA LINGUAGEM JAVA, seus recursos e conceitos fundamentais. Apresenta a documentação e mostra como consultá-la. Apresenta as ferramentas do ambiente de desenvolvimento (JDK). No final, são criados pequenos programas em Java que serão compilados e executados.

Índice deste módulo

Objetivos

No final deste módulo você deverá ser capaz de:

- Conhecer as principais características da linguagem Java que a distinguem de outras linguagens.
- Identificar, ao analisar um programa fonte em Java, suas principais estruturas como classes, métodos e declaração de variáveis.
- Saber compilar e executar uma aplicação Java. Saber usar um applet Java (de maneira genérica).
- Saber usar o ambiente de desenvolvimento para editar, salvar, compilar, executar e depurar programas em Java.
- Saber usar a documentação em hipertexto distribuída pela Sun para encontrar classes e métodos.

1.1. O que é Java?

Por que aprender Java? Qual o sentido de se aprender mais uma linguagem de programação? As outras que já existem já não bastam? É verdade que Java só serve para fazer animações na Web? Programas em Java são muito lentos, não são? Afinal, o que é Java?

Talvez por ter surgido numa época em que as informações se espalham rapidamente e de forma caótica, Java tem sido objeto de discussões que ora exageram nas suas qualidades, ora desprezam suas potencialidades, sem contar as vezes que a linguagem é confundida com outras linguagens como C ou JavaScript. O objetivo desta seção é esclarecer essas dúvidas e definir de uma forma clara que é Java, de acordo com o que dizem seus criadores.

O nome Java é usado pela Sun – a empresa que a trouxe ao mundo – para se referir não só à linguagem de programação, mas a toda uma arquitetura que envolve, além da linguagem, um ambiente de execução interpretado (a Máquina Virtual Java ou *Java Virtual Machine* - JVM) e uma plataforma de execução e desenvolvimento (bibliotecas dinâmicas, compiladores, etc.).

Uma das características mais singulares da linguagem e plataforma Java é o fato de poderem ser usadas, no desenvolvimento e na execução, em praticamente qualquer plataforma. Seu código-fonte, pode ser compilado em qualquer plataforma que possua um ambiente de desenvolvimento (*Java Development Kit* – JDK). O código é compilado para uma linguagem de máquina *virtual* que roda sem modificações em um amplo universo de plataformas e sistemas operacionais

diferentes (basta ter uma plataforma Java com o JVM instalado). Essa característica é fundamental para o sucesso de Java no mundo das aplicações distribuídas.

Mas antes que falemos mais de Java, vamos antes conhecer melhor suas características. Iniciaremos com um pouco de história e depois deixaremos que a Sun descreva os adjetivos que usa para definir a linguagem.

A história

Java nasceu das cinzas de um projeto fracassado. Inicialmente foi uma linguagem desenvolvida para programar chips de aplicativos de consumo, como PDAs, fornos de microondas, etc. A equipe que desenvolvia o projeto na Sun Microsystems, liderado por James Gosling, tinha inicialmente pensado em utilizar C ou C++, mas descobriu logo cedo que a linguagem não se adequava ao projeto. Como resultado, em 1990, Gosling começou a projetar uma nova linguagem que ele batizou de Oak (segundo ele, por causa de um carvalho – *oak*, em inglês – que ele via da sua janela). O nome posteriormente teve que ser mudado por que já existia uma outra linguagem registrada com o mesmo nome. A lenda diz que a idéia do nome veio numa visita à lanchonete que servia café, que os americanos costumam batizar com os mais variados nomes. Na época, o apelido era *Java*, referindo-se à ilha de origem dos grãos (na verdade, na hora do consumo, pouco importava se o café era colombiano, brasileiro ou asiático – era Java).

O projeto avançou, mas o mercado decepcionou e a Sun decidiu alterar seu rumo, desta vez investindo no ramo da TV interativa. O programa durou dois anos e consumiu milhões de dólares. Consistia no desenvolvimento de um protótipo de controle para TV interativa, usando uma linguagem desenvolvida especialmente para este fim (Java). Foi gasto uma fortuna. Criaram até mascote (o *Duke*... um bicho que parece com um dente de cabeça para baixo e que agora é mascote do Java). Quando o projeto finalmente foi apresentado em 1993, descobriu-se que o mercado de TV interativa não existia, e não havia uma previsão para quando iria deslanchar.

O financiamento para o projeto estava para ser cortado e a equipe dissolvida e transferida para outros projetos quando a Sun decidiu mudar drasticamente o objetivo do projeto, e focar o seu desenvolvimento na Web –

uma novidade que tinha acabado de surgir e já começava a fazer sucesso. A partir daí, a equipe trabalhou em um ritmo alucinante e em segredo até liberar a primeira versão alfa da linguagem, em maio de 1995.

As primeiras notícias ouvidas de Java eram sobre uma tecnologia que possibilitava colocar animações em uma página Web (na época, ainda não existiam as imagens GIF animadas). Isto porém só era possível com um browser chamado *HotJava*. O browser permitia que um programa fizesse parte de uma página, assim como uma imagem é integrada nas páginas Web. O programa era transferido pela rede e rodava na máquina do usuário, aparecendo dentro de uma parte da tela gráfica do *HotJava*. Esses programas, que rodavam dentro de páginas Web, foram chamados de *Applets*. As pessoas começaram a ver que o mundo estava cheio de applets, quando o *Netscape Navigator* – browser usado por 86% da população da Web na época – passou a suportá-los também.

Muitos ainda achavam que Java era só mais um brinquedo para tornar a Web mais atraente. Mas logo surgiram alternativas mais simples e leves para realizar animações e outras coisas antes só possíveis com Java. O interesse pela linguagem, porém, não morreu, pois a estratégia já havia funcionado: as pessoas começavam a descobrir que Java era mais que uma mera tecnologia para animar a Web. Era uma poderosa linguagem que podia ser usada para desenvolver aplicações, como as que se faz hoje usando C++, Delphi ou VB. Tinha, ainda, uma série de vantagens que as outras não possuíam: era orientada à rede, era mais segura, tinha recursos que a tornavam mais robusta e, apesar disso ainda conseguia ser menos complicada que várias outras linguagens com bem menos recursos. Tinha também o que era mais importante para um mundo onde a plataforma deixava o desktop e passava a ocupar as redes mundiais: era independente de plataforma. Com Java, finalmente era possível desenvolver um único programa, compilá-lo uma vez e rodá-lo em quase qualquer lugar. E esse tornou-se o lema da linguagem: "Escreva uma vez. Rode em qualquer lugar."

Java segundo seus criadores

Segundo a Sun, Java pode ser definida com uma série de adjetivos. Java é:

- uma linguagem *simples* e *familiar*;
- *orientada a objetos*;
- *compilada* e *independente de plataforma*;

- com *suporte à programação concorrente*;
- com *coleta de lixo*;
- *robusta*, com '*tipagem' forte*,
- *segura*,
- *extensível* e *estruturada*.

Vejamos alguns desses conceitos em detalhe:

Simples e familiar

Java é muito parecida com C – a linguagem mais popular do mundo para o desenvolvimento de software básico. Se você já conhece C ou C++, irá dominá-la em pouco tempo. Esta característica é provavelmente uma das principais razões da popularidade da linguagem. Várias outras linguagens bem melhores que C++ foram inventadas nos últimos anos, mas não atraíram a mesma atenção pois exigiam que o programador aprendesse uma sintaxe completamente nova, o que exigia dedicação e paciência – recursos caros para a maioria dos programadores.

Apesar da semelhança, Java não herdou as complexidades do C e é muito mais simples que C++. Todos os recursos do C/C++ que podiam ser substituídos com construções mais simples (como *estruturas*, *unions* e *typedef*) foram deixados de lado. Diferente de C, Java não tem ponteiros, arquivos de cabeçalho, pré-processadores, estruturas, uniões, matrizes multidimensionais, gabaritos e nem suporta sobrecarga de operadores.

Orientada a Objetos

Java é uma verdadeira linguagem *orientada a objetos*. Tudo em Java é objeto. A única exceção são os tipos simples como números e variáveis booleanas, que não são objetos por questões de desempenho. Mesmo assim, podem ser encapsulados em objetos quando necessário.

A orientação a objetos permite que programas sejam decompostos em estruturas de dados menores, com variáveis locais próprias e tarefas (operações) bem definidas. Um programa orientado a objetos consiste de várias dessas estruturas (os *objetos*) interligados que solicitam tarefas (através de *métodos*) entre si. Os programas podem ficar bem mais simples que os desenvolvidos usando decomposição orientada a procedimentos (procedural), pois um programa maior pode combinar vários objetos menores (e talvez bastante complexos) sem que

precise saber nada a não ser como usar seus métodos (que em geral são poucas e bem definidas).

Os arquivos em Java são organizados em *classes*, que são usadas para produzir objetos. As classes também podem herdar características (métodos e variáveis) de outras classes.

Java não suporta herança de múltiplas classes (característica presente em algumas linguagens orientadas a objeto como C++) já que o seu uso pode aumentar a complexidade sem trazer benefícios correspondentes. Mas existem casos onde a herança múltipla é benéfica. Para estes casos Java tem uma solução que permite o desenvolvimento de código com alto grau de reuso: as *interfaces*. Um programa construído usando interfaces pode ter módulos acrescentados no futuro sem que o seu código existente precise ser alterado (ou sequer compilado).

Em Java não existem variáveis globais ou funções independentes. Toda variável ou método pertence a uma classe ou objeto e só pode ser utilizada através dessa classe ou objeto.

Compilada e Independente de Plataforma

Um programa escrito em Java precisa ser compilado antes de ser executado. O compilador traduz o código-fonte e gera um código em *linguagem de máquina* para um *microprocessador virtual*. Esses arquivos são chamados *arquivos de classe* e sua linguagem de máquina é freqüentemente chamada de *bytecode*. Cada programa Java consiste da implementação de, no mínimo, *uma* classe. Depois de compilado, o programa-objeto pode ser executado em qualquer plataforma onde exista uma Máquina Virtual Java (JVM) instalada.

A compilação de um programa em C++ realiza a tradução do código-fonte da linguagem em instruções que são interpretadas pelo microprocessador da máquina onde foi compilado. O programa então só roda em outra máquina que tenha o mesmo tipo de processador. Já um programa em Java, quando compilado, gera instruções (os *bytecodes*) para o microprocessador da máquina virtual. Existem implementações desse microprocessador virtual para várias plataformas e o programa então rodará em qualquer uma delas.

O código Java consegue ser independente de plataforma porque a compilação é feita para uma máquina imaginária. Para que uma plataforma seja

capaz de rodar programas em Java, ela deve ter uma implementação que permita a emulação da máquina virtual.

Por ter suas instruções interpretadas por um software (processador virtual), os programas em Java são mais lentos que os escritos em C ou C++ (portanto, não é a *linguagem* que é lenta, mas a *plataforma*). Há, porém, vários meios de melhorar esse desempenho. Os interpretadores Java com *compiladores Just-In-Time* (JIT) já são praticamente onipresentes entre as plataformas Java mais populares (as primeiras não eram JIT). Esse tipo de sistema, converte as instruções em *bytecodes* para instruções do microprocessador na hora da execução, fazendo com que programas escritos em Java não percam em desempenho para programas escritos em C ou C++. Em alguns casos, porém, a compilação pode demorar e talvez não seja desnecessária a não ser naqueles trechos de código que exijam mais do processador. Para solucionar esse problema há uma nova geração de interpretadores-compiladores que se adaptam a essas condições. A Sun os chama de *HotSpot*. Já existem várias situações onde programas em Java rodam tão rápidos ou até mais rápidos que programas escritos em C++.

Robusta

Java é uma linguagem que tem *tipagem* de dados *forte*: ela exige que o *tipo* de objetos e números seja explicitamente definido durante a compilação. O compilador não deixa passar qualquer indefinição em relação ao tipo de dados. Esta característica garante uma maior segurança do código e o torna menos sujeito a erros, além de facilitar a depuração.

Programas em Java não provocam (sozinhos) core-dumps ou GPFs (*General Protection Fault*). Enquanto programas escritos em C ou C++ podem alterar qualquer posição da memória do computador, os programas escritos na linguagem Java não têm acesso direto à memória e deixam o controle a cargo do sistema operacional.

O sistema de gerenciamento de memória de Java libera os programadores da tarefa de se preocupar em liberar memória usada (veja tópico seguinte). Um processo chamado de coletor de lixo (*garbage collector*) opera sempre em uma rotina (*thread*) de baixa prioridade, liberando recursos que não são mais utilizados automaticamente.

E quando ocorrem erros ou situações inesperadas em tempo de execução, Java possui um meio de lidar com elas e se recuperar do erro se possível. O controle de exceções (condições excepcionais) é uma parte fundamental da linguagem e em muitos casos seu uso é obrigatório.

Todos esses recursos, tornam os programas escritos em Java mais robustos, pois torna os erros menos freqüentes tanto na fase de desenvolvimento como na fase de execução. O resultado é uma maior produtividade no desenvolvimento, reduzindo o custo para se ter programas com menos *bugs* e mais confiáveis.

Com coleta de lixo

A coleta de lixo é uma técnica automática de liberação de memória. Muitas linguagens permitem que ao programador alocar memória em tempo de execução. Esta alocação consiste geralmente no retorno de um ponteiro que indica o início do bloco de memória que foi alocado. Quando os dados armazenados naquela posição de memória não são mais necessários, o programa deve liberar os recursos para que ela possa ser reutilizada e evitar que o sistema pare por falta de memória.

Nas linguagens de programação comuns (como C e C++), é o programador que deve manter um controle da memória que foi alocada e liberá-la quando ela não mais for utilizada. Isto é uma tarefa complexa que exige disciplina e extrema atenção do programador. Geralmente, o programador que escreve o código esquece de liberar memória (ou libera quando ainda não pode liberar) e o problema só é descoberto quando o software já está em versão beta. Nesse estágio, uma dúzia de programadores vão passar noites tentando corrigir os bugs e o cronograma de entrega do software vai mais uma vez ser adiado, e o custo vai subir.

O sistema de coleta de lixo tira esta responsabilidade do programador, aumentando a produtividade, reduzindo os custos. Através de uma linha de execução (*thread*) de baixa prioridade, o sistema de coleta de lixo mantém um controle da memória alocada e conta o número de referências que existem para cada ponteiro de memória. Nos intervalos em que a JVM está inativo, o coletor de lixo (*Garbage Collector* – GC) verifica quais os ponteiros de memória que não têm mais referências apontando para eles marca a o endereço de memória correspondente como livre.

O GC é acionado automaticamente pelo sistema e o programador não tem acesso a ele. Ele pode, porém, influenciar o GC reinicializando as referências ou chamando o GC diretamente (através de função), mas a operação só ocorre quando a JVM permitir. Não há como forçar a coleta de lixo.

Dinâmica (extensível)

A linguagem Java foi projetada para se adaptar a um ambiente dinâmico, em constante evolução. Um programa pode, por exemplo, receber código binário (*bytecode*) pela Internet, carregá-lo e se estender automaticamente com novas classes (módulos). Possui uma representação (sintaxe) de tempo de execução que permite que o programa saiba a que classe pertence um objeto, na hora em que o recebe. Isto permite a inclusão dinâmica de classes que podem estar em qualquer lugar da rede.

Java também suporta a integração com métodos (funções) nativos de outras linguagens. Dessa forma, pode-se desenvolver aplicativos híbridos em Java e C ou C++, aproveitando o grande volume de código existente hoje nessas linguagens. Usando a linguagem JavaIDL (parte do JDK) e CORBA (tecnologia de integração orientada a objetos), uma aplicação Java pode se comunicar com qualquer outro programa orientado a objetos através da rede.

Segura

Por ter uma *tipagem* de dados forte, somente permitir acesso a campos pelo nome (e não por endereço), não ter aritmética de ponteiros nem qualquer tipo de acesso direto à posições de memória, um programa compilado em Java pode ser verificado antes de ser executado. A verificação dos *bytecodes* é realizada nos browsers Web que suportam Java para garantir que os *applets*, transferidos pela Internet, não estejam violando as restrições da linguagem e não possam provocar danos no computador do usuário. Depois da verificação, os applets podem ser otimizados pelo sistema de execução, para garantir um melhor desempenho.

A linguagem também fornece um esquema de segurança baseado em certificados. Com eles, pode-se classificar certos applets como "confiáveis" com base em um certificado garantido pelo seu fabricante e ampliar as suas capacidades (applets normalmente têm várias restrições de segurança). Com esse artifício, será possível desenvolver applets mais úteis com capacidade de escrever em disco, imprimir, acessar mais de um endereço de rede, etc.

Como Java não dá acesso direto à memória, torna-se muito difícil o desenvolvimento de vírus, da forma como são feitos hoje, usando a linguagem Java. Pode haver, porém, programas hostis criados para causar transtornos aos usuários. A solução contra este tipo de programa pode estar nos certificados de segurança.

A segurança é garantida por uma classe Java chamada *Security Manager*. Nos browsers, o usuário não pode alterá-la, mas nas aplicações distribuídas desenvolvidas pelo programador, diferentes níveis de segurança poderão ser definidos. Nesses casos, a segurança é controlada exclusivamente pelo programador. Mas nos applets, não. Ao receber um applet em seu browser, a máquina virtual nele instalada sempre observa as três etapas abaixo:

- O carregamento do código
- A verificação do código
- A execução do código

A segunda etapa, de verificação, só ocorre com applets que vêm pela rede. Nas aplicações ou applets locais, apenas a primeira etapa e a última são realizadas. O *Class Loader* – carregador de classes sabe se a classe é local ou remota na hora em que recebe o programa Java (do disco local ou de um endereço remoto). Programas locais então são considerados confiáveis e têm menos restrições. Programas remotos têm que passar pelo *Bytecode Verifier* – o verificador de código, que irá dizer se o código foi corrompido ou não.

Suporte à programação concorrente (multithreading)

Programas em Java podem ter mais de uma linha de execução ocorrendo ao mesmo tempo. Os programadores podem definir quando e com que prioridade certas linhas de execução serão rodadas. O *multithreading* é essencial em aplicações gráficas. Sem ele não seria possível que os programas realizassem outras tarefas enquanto o usuário interagisse com ele. A programação de *threads* é trabalhosa em qualquer linguagem, inclusive em Java, mas Java a torna bem menos complexa ao permitir que ela seja realizada com classes e objetos da linguagem e não exigir que o programador faça chamadas ao sistema operacional (como ocorre com C ou C++). O resultado é que programadores são estimulados a usar threads sempre que o seu uso trouxer benefícios ao programa. Embora o controle e

agendamento de *threads* dependam do sistema operacional, é possível desenvolver programas *multithreaded* independentes de plataforma em Java.

Ambientes de execução

Java pode ser empregada em vários ambientes de execução. Os mais conhecidos são o sistema operacional (programas independentes), o browser (*applets*) e o servidor (*servlets*). Mas as plataformas que suportam Java não se limitam a *mainframes* e computadores desktop. Java pode ser usada em telefones celulares, fornos de microondas, controles remotos de TV, automóveis e até em anéis, colares, braceletes e cartões magnéticos! O melhor de tudo é que os programas desenvolvidos para essas plataformas podem ser criados e prototipados em computadores comuns, reduzindo imensamente o custo de desenvolvimento. Você pode, por exemplo, desenvolver software para cartões magnéticos (*smartcards*) baixando o *JavaCard* API do site da Sun e usando-o para compilar seus programas. Pode desenvolver software para telefones celulares usando a API *PersonalJava*. Vários aparelhos hoje em dia possuem máquinas virtuais Java embutidas e são clientes em potencial dos softwares que você poderá desenvolver com Java.

O mais recente avanço na tecnologia de plataformas Java é o *Jini* – uma arquitetura que permite o desenvolvimento de aplicações distribuídas entre quaisquer tipo de aparelho. Tem como objetivo fazer valer a máxima *Write once, run anywhere* (escreva uma vez, rode em qualquer lugar) e deverá ampliar de forma significativa o mercado para os desenvolvedores Java.

O que promete a linguagem Java?

Em uma rede heterogênea, que consiste de milhões de máquinas diferentes ligadas entre si, as características da linguagem Java a colocam em posição confortável como a linguagem ideal para o desenvolvimento de aplicações para a Internet e sistemas distribuídos.

Os *bytecodes* compilados são transferidos através da rede e rodam em qualquer máquina ligada à Internet, seja como uma applet, através de um browser, seja como uma aplicação independente. Se o seu aplicativo favorito foi escrito em Java, não importa se você tem uma máquina Solaris, um PC rodando Windows, um servidor NT ou um Macintosh. Ele irá rodar de forma quase igual em qualquer dessas máquinas.

Java deverá causar uma redução do custo de aplicativos que precisam rodar em várias plataformas. Com linguagens tradicionais, era necessário portar e compilar o código-fonte em cada plataforma diferente, e tratar os *bugs* que apareciam em cada uma delas individualmente. Depois, ainda ter documentação e suporte para cada plataforma, e embalagens diferentes para cada versão. Se o programa for escrito em Java, só haverá uma compilação, uma documentação e um produto comercial que poderá ser adquirido por qualquer usuário, independente de sua plataforma de trabalho.

A forma como se usa software também poderá mudar. Com uma rede suficientemente rápida, um usuário que precisar ocasionalmente usar um aplicativo qualquer poderia alugá-lo em vez de comprar todo um pacote de software. O programa seria transferido para a sua máquina, ele realizaria o seu trabalho e depois o descartaria, pagando apenas uma taxa pela utilização. Esta é a filosofia do *Network Computer* (NC) e dos provedores de aplicações.

Todas essas idéias são possíveis com Java. Em 1997 elas pareciam utópicas, irreais. A linguagem também só tinha dois anos e apresentava problemas de compatibilidade, a rede era lenta, entre outros problemas. Hoje a linguagem está bem mais estável com a plataforma Java 2 (JDK1.2) e muitas das utopias do passado já são realidade. Agora chegou o Jini com a promessa de colocar Java em todos os aparelhos do lar. E então, por que aprender Java? Será que vale a pena? Se você acha que sim, então, mãos à obra!

1.2. Introdução Prática

Nesta seção nós iremos introduzir os conceitos básicos de Java, como sua estrutura léxica, tipos de dados, expressões e outras características comuns à qualquer linguagem de programação. Como é impossível estudar Java sem levar em conta a sua natureza orientada a objetos (todo programa Java começa com uma declaração de classe), será inevitável depararmos com exemplos de criação de objetos, utilização de métodos, etc. Estes tópicos serão explicados aqui de forma superficial, mas merecerão uma abordagem mais detalhada em capítulos posteriores.

O Java Development Kit

A melhor forma de introduzir os conceitos de uma linguagem é com exemplos reais. É o que faremos nesta seção. Você pode testar todos os programas usados nos exemplos no seu próprio computador. Para isto é preciso que tenha no mínimo uma das configurações seguintes:

- O Java Delopment Kit (JDK 1.1 ou superior) e um editor de textos (Notepad, Vi, WinEdit, etc.); ou
- Um Ambiente Integrado de Desenvolvimento (*Integrated Development Environment* – IDE) para a linguagem Java como o Microsoft Visual J++, Sun Java Studio, Symantec Visual Café, Borland JBuilder, Oracle JDeveloper ou outro.

Um IDE é ideal para desenvolver aplicações em Java com *produtividade*. Possui diversas ferramentas gráficas para gerenciar projetos, gerar código, depurar programas e reutilizar módulos dinâmicos. Um IDE, porém, não é o ambiente ideal para *aprender* Java. Usar o Borland JBuilder ou o Oracle JDeveloper, por exemplo, é fortemente recomendado no dia-a-dia do desenvolvimento de software em Java, mas é extremamente importante que você saiba Java e se familiarize com a linguagem antes! Se você sabe Java, não ficará dependente de uma ou outra ferramenta. Você terá condições de ir além da ferramenta quando e se for preciso.

Como o JDK é gratuito, e pode ser descarregado pela Internet, as instruções de compilação e execução mostradas aqui serão referentes a ele. Se você pretende acompanhar os exemplos deste capítulo, instale e configure o JDK agora (veja se ele já não está configurado na sua máquina). O funcionamento do JDK é dependente de plataforma. Leia o README e instruções para saber como usá-lo em sua máquina. As instruções deste capítulo supõem que o seu JDK irá rodar em um PC ou máquina Unix (se você estiver usando um Mac, consulte a documentação para saber como executar, compilar e editar arquivos).

O JDK 1.1 (para Windows e Solaris) consiste do seguinte:

- *Java API*: todas as classes e interfaces, organizadas em *pacotes*;
- Os *códigos-fonte* das classes, interfaces e métodos da API;
- *Applets e aplicações de demonstração* com exemplos de utilização;
- *Máquina Virtual Java* (java):
- *Visualizador de Applets* (appletviewer);

- *Ferramentas de desenvolvimento:* compilador Java (javac), gerador de métodos nativos C (javah), gerador de documentação (javadoc), debugger (jdb), disassembler (javap), profiler (javaprof), jar e outras.

A documentação do JDK

Além das ferramentas básicas, ainda usaremos a *documentação* em hipertexto para termos acesso a uma referência completa da linguagem Java. Essa referência está em HTML e pode ser consultada através de um browser. A documentação não faz parte do pacote distribuído pela Sun. Precisa ser baixada à parte e tem quase o mesmo tamanho em megabytes que o próprio JDK.

Descarregue a documentação e expanda o arquivo ZIP na raiz do disco onde o seu JDK estiver instalado. A descompressão automaticamente criará um subdiretório /docs/ dentro daquele diretório. Para ler a documentação, abra o arquivo index.html desse diretório no seu browser. Uma referência em hipertexto está disponível na opção "Java Platform API". Durante todo este curso, usaremos esta referência como guia para o desenvolvimento dos programas.

Como conseguir o JDK

Caso você deseje obter uma distribuição do JDK para uso próprio ou em outra plataforma que aquela do laboratório, poderá obtê-la pela Internet. Elas são gratuitas. O JDK1.2 (ou Java 2 platform) está disponível na Sun para os sistemas operacionais Solaris (SunOS), Windows e Macintosh. As versões para outras plataformas são distribuídas por outras empresas cujo endereço se obtém através das páginas da Sun.

Para descarregar a última versão do JDK para Solaris (SPARC ou x86), Windows ou Macintosh, visite a sua home page na JavaSoft em:

```
http://java.sun.com
```

O JDK também existe para outras plataformas. Informações sobre esses produtos podem ser obtidas no mesmo site acima. Algumas versões são:

- OS/2 e AIX:

  ```
  http://ncc.hursley.ibm.com/javainfo/
  ```

- Linux:

  ```
  http://www.blackdown.org
  ```


- NeXT:

  `http://www.next.com`

- HP/UX, AT&T Unix e DEC Alpha OSF/1:

  `http://www.gr.osf.org:8001/projects/web/java/`

Como Preparar e Usar o Ambiente Java da Sun

Depois que você tiver transferido o JDK para o seu disco, instale-o de acordo com as instruções para seu sistema (as instruções para *Windows*, *Solaris* e *Macintosh* estão disponíveis na home page do JDK).

O ambiente utiliza uma propriedade do sistema chamada `CLASSPATH` para informar a localização das bibliotecas de classes durante a compilação ou execução. Em geral, ela é configurada automaticamente, mas, se você estiver utilizando pacotes de terceiros, poderá ser necessário acrescentar o caminho dessas classes ao `CLASSPATH`. Você pode fazer isto a cada compilação ou execução, ou então definir esse caminho no seu sistema. No *Windows95*, você poderá acrescentar no arquivo **`AUTOEXEC.BAT`**, uma declaração do tipo:

```
CLASSPATH = %CLASSPATH%;c:\classes\adicionais\arquivo.jar;.
```

O `CLASSPATH` original, que você não deve sobrepor, contém os caminhos para as bibliotecas fundamentais, essenciais para a compilação e execução dos programas em Java. Bibliotecas são, em geral, distribuídas em arquivos `*.jar` ou `*.zip` que contém as classes organizadas em sistemas de diretórios.

No *Windows95*, você também deve incluir na sua variável `PATH`, o caminho `jdk1.2\bin\`, para que as ferramentas e aplicações do JDK possam ser executadas de qualquer lugar através do *MS-DOS Prompt*. No *Windows*, é interessante que você use o aplicativo `DOSKEY`, que guarda um histórico das linhas de texto digitadas, principalmente se você estiver usando o JDK apenas como ambiente de execução e compilação. Um `AUTOEXEC.BAT` típico para usar o JDK via linha de comando seria:

```
PATH=%PATH%;c:\jdk1.2\bin;
DOSKEY
```

Supondo que o seu ambiente Java foi instalado no drive C:

Depois de instalado o ambiente Java, você poderá rodar e compilar aplicações e applets, usando instruções da linha de comando. Por exemplo, para compilar um arquivo **`MeuProg.java`**, use o **`javac`** da seguinte maneira:

```
javac MeuProg.java
```

O compilador poderá responder com uma série de mensagens de erro ou gerar um ou mais arquivos objeto (`*.class`) com sucesso. Para executar o arquivo compilado **`MeuProg.class`** (se ele for um programa executável) use o programa interpretador **`java`**, omitindo (sempre) a extensão do arquivo:

```
java MeuProg
```

Isto vale até para aplicações gráficas (elas irão restaurar o ambiente gráfico do sistema para executar). Se seu arquivo de classe não for uma aplicação mas for uma applet, você só poderá executá-lo através de uma página HTML, onde ele deve ter sido embutido (adiante, veremos como fazer isto). Ele poderá ser visualizado em um browser ou com o **`appletviewer`** – ferramenta do JDK para testar applets. O argumento do appletviewer deve ser o nome de uma página HTML:

```
appletviewer TesteApplet.html
```

O **`appletviewer`** mostra apenas a applet, ignorando o restante da página HTML. Para ver toda a página com a applet incluída, você precisará de um Browser Web que suporte Java, como o Netscape Navigator ou o Internet Explorer com suporte Java.

Existem ferramentas de desenvolvimento para criar rotinas de instalação e execução profissionais para programas em Java. Elas são comerciais. As mais populares são o *InstallShield* e o *InstallAnywhere*. Elas instalam, na máquina do cliente, o JRE – *Java Runtime Environment* (parte do JDK sem as ferramentas de desenvolvimento) se o usuário já não tiver o ambiente na sua máquina e criam ícones da área de trabalho ou barra de ferramentas do sistema operacional do usuário. O JRE está incluído no JDK e você pode distribuir suas aplicações com ele livremente. É possível também, baixar o JRE da Sun sem o JDK.

Ambiente de desenvolvimento do laboratório

Neste curso, utilizaremos o JDK da Sun com um editor de código shareware chamado *WinEdit* (ambiente PC), que oferece uma interface tipo IDE

ao JDK. O *WinEdit* tem a vantagem de ser configurável, adequado à edição de código por converter tabulações em espaços além de outras vantagens. Ele permite que o usuário configure seus menus ainda para rodar o compilador, o interpretador e o *debugger* Java (JDB). Na versão que iremos utilizar, as palavras reservadas da linguagem Java aparecerão em azul se o arquivo tiver a extensão .java. Então, antes de escrever qualquer coisa em um arquivo do WinEdit, salve-o primeiro com a extensão `.java`. O *Kawa* é bem mais versátil e tem bem mais recursos. Escolha uma das duas ferramentas e mãos à obra.

Exemplos de Programas em Java

A seguir, mostraremos vários programas (didáticos[1]) em Java. Se você quiser digitá-los e testá-los em seu computador (altamente recomendável), tenha o cuidado de digitar o código *exatamente* da forma como é apresentado. Letras minúsculas e maiúsculas *são diferentes* em Java. Se na listagem aparece "String" e você digita "string", com "s" minúsculo, o compilador acusará um erro e o seu programa não será compilado.

Esta regra também vale para o *nome do arquivo*. Mesmo que o *Windows95* não faça distinção entre maiúsculas e minúsculas, quem efetivamente executa o programa compilado é o interpretador Java (seja via browser ou linha de comando) que considera maiúsculas diferentes de minúsculas. Então, se você salvar o arquivo como `Hello.java` e tentar compilar `hello.java`, o compilador acusará um erro, pois não será capaz de encontrar o arquivo. Use sempre os nomes do Windows e nunca os nomes do DOS (`HELLO~1.JAV` não funciona!).

HelloWorld Console

O seguinte programa é o clássico "Hello World" escrito como uma aplicação Java. Depois de compilado e executado ele imprime na tela (linha de comando) as palavras "Bom dia, javaneses!!!!" dez vezes.

```
import java.lang.Object;
import java.lang.String;
import java.lang.System;
```

[1] Programas didáticos geralmente são inúteis e programas muito úteis geralmente não são bons exemplos didáticos. Neste curso mostraremos programas mais úteis logo que tenhamos um pouco mais de experiência com a sintaxe de Java.


```
/* Programa Hello World */

class HelloWorld extends Object {

    public static void main(String[] args) {
        for(int x = 0; x < 10; ++x) {
            System.out.println("Bom dia, javaneses!!!");
        }
    }

}
```

Digite, compile e rode o programa acima no seu computador (os detalhes dependem do ambiente de desenvolvimento que você estiver usando). Se você estiver usando o JDK, faça o seguinte:

1) Crie um diretório para armazenar seus programas Java e salve o programa acima em um arquivo "`hello.java`", nesse diretório;
2) Abra uma janela de comando (MS-DOS Prompt) ou shell e mude para o diretório onde está o programa.
3) Para compilar, rode o compilador Java (`javac`) seguido do nome do seu programa:

   ```
   javac hello.java
   ```

   Se não houver erros, o programa foi compilado e o prompt C:> reaparecerá (se houver erros, volte e corrija-os). Se você listar o conteúdo do seu diretório agora, verá que o compilador criou um arquivo chamado "`HelloWorld.class`". Observe que este é o nome que demos para a classe no programa acima.
4) Finalmente, para rodar o seu programa, chame o interpretador Java seguido do nome da classe (sem a extensão):

   ```
   java HelloWorld
   ```

   O programa, então, deve exibir na tela o texto seguinte:

```
Bom dia, javaneses!!!
Bom dia, javaneses!!!
Bom dia, javaneses!!!
...
```

Como funciona?

O programa consiste de uma declaração de classe (que representa o programa-objeto) e de um método (função) especial (`main()`) que contém as

instruções da principal linha de execução do programa (onde ele começa). Você pode compilar código Java sem o `main()` mas ele não será um programa executável do S.O. É no `main()` que começa a execução da aplicação. Por `main()`, nos referimos na verdade ao bloco:

```
public static void main(String[] args) { ... }
```

Que deve ser digitado EXATAMENTE como acima, com exceção da palavra `args` que é nome de variável e pode ser substituída por qualquer outra (`x`, por exemplo). A *classe* é todo o programa:

```
class HelloWorld extends Object { ... }
```

Isto diz que `HelloWorld` é uma classe Java e que estende a classe `Object`. Todos os nomes em negrito são classes. Em Java, é uma convenção escrever nomes de classes com a primeira letra maiúscula (isto é regra na API). `Object` é uma classe (estrutura ou tipo de dados) que contém uma infraestrutura mínima para que outras classes possam existir. Qualquer classe Java, seja ela da biblioteca fundamental (API), seja ela criada por você, deriva e herda todas as características de `Object`. A extensão (`extends`) caracteriza a herança. Você pode estender `Object` ou outra classe (que descende de `Object`). Nós sempre construímos em cima de uma infraestrutura existente e assim fica mais fácil de escrever programas.

Mas onde está `Object`? Como é que o sistema encontra essa classe? A resposta a essas perguntas está nas três primeiras linhas do programa (as instruções `import`) e no `CLASSPATH`. Quando você instala um programa Java, o JDK, um browser ou qualquer aplicação que utilize a máquina virtual Java, essa aplicação define um `CLASSPATH` que informa ao ambiente de execução onde estão as classes da biblioteca fundamental. Esse `CLASSPATH` é definido automaticamente quando você instala o JDK. Pode ser algo como:

```
C:\jdk1.2\jre\lib\rt.jar
```

O arquivo `rt.jar` contém um sistema de arquivos e diretórios com as classes Java. Ele tem a seguinte estrutura de diretórios:

```
java\ ____ lang\
      |__ awt\ _____ event\
      |          |___ image\
      |       (...)
      |__ applet\
```



```
|  (...)
|__ sql\
```

Cada diretório é chamado de pacote. Dentro dos pacotes há várias classes, por exemplo, dentro de `java\lang\` existem, entre outras, as classes `Object.class`, `String.class` e `System.class` que importamos no início do nosso programa. Podemos concluir, então, que o separador "\" ou "/" dos diretórios é substituído por um ponto ".", quando lidamos com pacotes.

Dentro do pacote java.lang estão classes essenciais em qualquer programa Java. Por causa disto, mesmo que você não importe as classes do java.lang explícitamente, elas são importadas automaticamente pelo sistema. Toda classe também tem que herdar pelo menos de Object, portanto, se nós não tivéssemos declarado isto na cláusula extends, o sistema iria fazê-lo automaticamente. Em outras palavras, o programa funcionaria da mesma forma se tivéssemos programado apenas as linhas abaixo:

```
class HelloWorld {
    public static void main(String[] args) {
        for(int x = 0; x < 10; ++x) {
            System.out.println("Bom dia, javaneses!!!");
        }
    }
}
```

Só há um método (função) no nosso programa. É o `main()`. Neste programa, há no `main()`, uma estrutura de repetição `for`:

```
for(int x = 0; x < 10; ++x) {
   System.out.println("Bom dia, javaneses!!!");
}
```

O texto em sublinhado corresponde a palavras reservadas. A estrutura `for`, que vem seguida de argumentos entre parênteses, abre um bloco entre chaves que só contém uma instrução, que é repetida dez vezes. No próximo capítulo analisaremos em detalhes a sintaxe do `for` que *inicializa*, *testa* e *incrementa* uma variável enquanto é chamada. A instrução que é repetida no `for` é:

```
System.out.println("Bom dia, javaneses!!!");
```

Não vamos entrar em detalhes sobre a sintaxe desta instrução. No momento só precisamos saber que essa instrução imprime o conteúdo de uma cadeia de caracteres (entre aspas) que é passada entre parênteses. Outro detalhe

importante é o ponto-e-vírgula (`;`). Assim como várias outras linguagens de programação, em Java, toda instrução simples deve terminar em ponto-e-vírgula.

A palavra reservada "class" precede o nome da classe que chamamos de `HelloWorld` e todo o código do programa fica dentro desta classe, delimitado pelas chaves `{` e `}`. Dentro da classe, está o método (main) que também envolve com chaves toda a sua implementação. Todo programa em Java tem esta estrutura. Procedimentos (instruções, controle de fluxo, etc.) só podem ocorrer dentro de métodos. Métodos só podem ocorrer dentro de classes. Fora de uma classe só se admite *comentários*, zero ou mais declarações *import* (antes da classe), zero ou uma declaração *package* (antes dos *import*) e outras definições de classe (pode haver mais de uma por programa-fonte).

O nome do método é apenas `main`, mas a sua declaração traz outras palavras: `public`, `static` e `void`. São chamadas de *modificadores* do método. Main é um método especial e deve estar presente em toda rotina *executável* através do S.O. Quando o interpretador Java roda o programa, o primeiro método que ele chama é o `HelloWorld.main(`*args de linha de comando*`)`. Esta notação `Classe.método()` é usada para se referir a *funções públicas* (métodos declarados como `public` e `static`).

Vamos mostrar mais dois exemplos que você pode compilar e rodar. É recomendável que você os digite e compile para ir se familiarizando com o seu ambiente de desenvolvimento.

HelloWorld Gráfico

Esta outra aplicação faz o mesmo que a anterior mas imprime as palavras "Saudações Javanesas!" em uma janela aberta na tela. É uma aplicação escrita para usar a interface gráfica (GUI). Tudo está dentro da declaração de classe `HelloWorld2`, exceto uma declaração "`import`" que importa a biblioteca onde estão as classes básicas do sistema de janelas (desta vez ela é necessária). Apenas as classes do pacote `java.awt` estão em negrito. Observe que o programa consiste de dois arquivos!

```
// HelloWorld Gráfico – classe que define a interface (não executável)
import java.awt.*;
public class HelloWorld2 extends Frame {
   private Label hello;
```



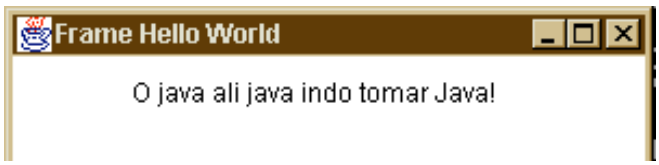
```
   public HelloWorld2(String nome) {      // Isto é um construtor
      super(nome);
      hello = new Label("O java ali java indo tomar Java!");
      FlowLayout flow = new FlowLayout(); // cria política de layout
      this.setLayout(flow);               // aplica política de layout
      this.add(hello);           // add() obedece pol. de layout atual
   }
}
```

```
// classe executável que cria objeto do tipo HelloWorld2

import HelloWorld2;  // importa do CLASSPATH (que inclui diretório atual)

class WinHello {

      public static void main(String[] args) {
            HelloWorld2 javali;
            javali = new HelloWorld2("Frame Hello World");
            javali.setSize(300, 100);
            javali.setVisible(true);
     }
}
```

Detalhes sobre a implementação do programa acima serão vistos no decorrer do curso, nos capítulos sobre applets e sobre programação da interface gráfica. Observe que a primeira classe tem um método chamado `HelloWorld2()` (o mesmo nome que a sua classe). Esse método especial chama-se *construtor* e é utilizado para inicializar objetos quando são criados. E é só isso que o `main` faz aqui: cria um objeto.

No exemplo anterior você podia utilizar qualquer nome para o seu arquivo `*.java`. Neste exemplo, a classe `HelloWorld2` não compila a não ser que o arquivo `.java` se chame `HelloWorld2.java` (lembre-se do `H` e `W` maiúsculos!). Isto é necessário porque a classe foi declarada pública (para que possa ser usada por outras classes fora do diretório atual). O javac exige que só haja uma classe pública por arquivo-fonte. Para garantir isto, exige que o nome do arquivo-fonte seja o mesmo da classe.

Se você rodar o programa (a classe executável `WinHello`) verá saltar na tela uma janela do *Windows* com o texto "O javali já vai indo tomar Java!" dentro dela, como na figura abaixo:



Java chama esse tipo de janela de "`Frame`" e essa aplicação, como qualquer outra aplicação de janelas, é um "`Frame`". Se ela "é", é porque estende `Frame` como uma subclasse (os detalhes sobre orientação objetos serão abordados mais adiante). Isto está representado na declaração da classe `HelloWorld2` através da palavra reservada "`extends`". Este é novamente um exemplo da reutilização. Nós construímos em cima de uma estrutura já existente: o Frame, e com isto fazemos, com poucas linhas, uma aplicação gráfica.

No nosso exemplo um programa executável usou uma classe que nós mesmo criamos (`HelloWorld2`) para construir um *objeto* (`javali`) chamar seus métodos (para torná-la visível e definir seu tamanho). Os métodos de `HelloWorld2` não foram definidos por nós. Nós apenas temos um método que é o construtor, e este foi chamado usando `new` no `main()` do executável. De onde vêm, então, os métodos `setSize()` e `setVisible()`?

A resposta pode estar na herança. Se podemos usar um método que não está disponível na nossa classe, certamente este método foi herdado de uma classe mais acima na hierarquia! Procure na documentação Java e tente descobrir em que classe os métodos `setSize()`, `setVisible()`, `setLayout()` e `add()` estão definidos. A palavra `this` é um ponteiro para o objeto corrente da própria classe. `this.add()`, portanto, chama um método que faz parte dos métodos de `HelloWorld2`.

HelloWorld Applet

Uma terceira versão do "Hello World" é um programa que roda como uma applet, dentro de uma página Web (todas as classes em negrito):

```
import java.awt.Graphics;
import java.applet.Applet;

public class HelloWorld3 extends Applet {

    private String mensagem;

    public void init() {
        setBackground(java.awt.Color.white);
    }

    public void paint(Graphics g) {
        mensagem = getParameter("msg");
        g.setColor(java.awt.Color.red);
        g.drawString(mensagem, 20, 20);
    }
```



```
}
```

Novamente, se você tentar rodar este programa a partir da linha de comando, o interpretador acusará um erro, pois não encontrará o método `main`. Esta versão não contém método `main` porque é um applet e só pode ter a sua execução iniciada por um browser. O applet contém métodos especiais que regulam o seu ciclo de vida, eventos, apresentação, etc. como veremos no capítulo sobre applets mais adiante. Um deles, chamado `paint()`, é invocado automaticamente pelo browser e fornece um contexto gráfico (espaço na tela do browser) onde se pode desenhar linhas, definir cores e fontes, e imprimir texto como no programa acima. O método `init()` é chamado uma vez para inicializar o applet. No nosso programa, nós *sobrepusemos* os dois métodos, isto é, definimos uma nova funcionalidade para eles. A *assinatura* (tipo de retorno, nome do método, número e tipo dos argumentos) dos dois tem que ser igual à assinatura dos métodos originais para realizar a sobreposição. Só assim o browser saberá quem chamar e quais parâmetros passar. Quando o browser chama `paint()`, automaticamente passa como argumento um contexto gráfico para o applet.

Assim como o programa anterior era um "`Frame`", este *é um* "`Applet`". Todo applet é uma subclasse de `java.applet.Applet`. A classe `Applet`, está organizada abaixo de uma biblioteca (pacote) chamada de "`java.applet`". `Frame` e `Label` são classes que pertencem à biblioteca "`java.awt`". Usa-se então "`import`" no início do programa para informar a localização das classes utilizadas.

Como vimos, o programa acima não roda sozinho. Applets não são aplicações independentes; precisam de um outro programa que os inicie. Para utilizar a applet é necessário "chamá-la" através de uma página Web usando um descritor HTML `<applet>` como mostrado a seguir:

```
<html>
  <head>
       <title>O Famigerado Hello World em forma de Applet</title>
  </head>

  <body bgcolor=white>
       <h1 style="font-color: red; font-family: sans-serif">
       O Famigerado Hello World...</h1>
       <p>... em forma de Applet!</p>
       <p align=center><hr>
```



```
        <applet code=HelloWorld3.class height=50 width=400>
          <param name=msg value="Leve-me ao seu mestre, terráqueo!">
        </applet>
      </p>
  </body>
</html>
```

O programa lê parâmetros da página HTML através de um outro descritor HTML `<param>`, ou seja, o texto da applet pode ser mudado pelo autor da página HTML que não precisa entender nada de Java.

Digite o programa da applet e salve-o em um arquivo com o nome `HelloWorld3.java`. (novamente, é importante que o programa-fonte tenha *o mesmo nome* que a classe). Depois carregue o HTML em seu browser (*Netscape*, *Internet Explorer*, *HotJava* ou outro que suporte Java 1.1 em diante) ou isoladamente, usando o *AppletViewer* – programa distribuído com o JDK. Para rodá-lo digite:

`appletviewer` *`nome-da-página`*`.html`

Onde `nome-da-página.html` é o nome que você deu ao arquivo HTML onde está a chamada para a sua applet. A figura ilustra a applet acima rodando dentro de uma página em um browser.

1.3. Análise dos programas

Pudemos perceber que o compilador Java cria um arquivo de classe (`.class`), formado por instruções de *bytecodes* (linguagem de máquina virtual Java), para *cada classe* declarada no programa fonte. O programa fonte também é chamado de *unidade de compilação*. Nos nossos exemplo, apenas um arquivo de classe por unidade de compilação. Se uma única unidade de compilação tiver 10 classes declaradas, o compilador produzirá 10 arquivos *bytecode*. Cada classe irá dar origem a um arquivo de mesmo nome, com a extensão `class`.

Há, porém, uma restrição quanto ao nome que pode ser dado ao programa fonte. A linguagem Java estabelece que só poderá haver *uma* classe pública (declarada com o modificador "`public`") em cada unidade de compilação. Existindo tal classe pública, o nome do arquivo fonte deverá ser o *mesmo* que o nome dessa classe, acrescentado naturalmente da extensão `.java` (veja como exemplo a classe `HelloWorld3` que poderia ser compilada no mesmo arquivo que `WinHello`, que não é pública).

Por exemplo, suponha que uma unidade de compilação possui várias classes declaradas `Cachorro`, `Gato` e `Rato` e uma classe *pública* chamada `Animal`. O arquivo fonte deverá chamar-se necessariamente `Animal.java`. Se for usado outro nome o compilador irá exibir uma mensagem de erro e não realizará a compilação. Usando-se o nome correto, o resultado será o esperado: quatro arquivos de classe: `Animal.class`, `Cachorro.class`, `Gato.class`, e `Rato.class`.

Normalmente, as classes que definem programas são públicas, pois desejamos que elas possam ser executadas fora do seu diretório de origem (o que não seria possível se não fossem públicas). Não usamos `public` nos nossos executáveis mais por questões didáticas (para mostrar que o nome da classe deriva da declaração "class" e não do nome do arquivo).

O Método main() e outros métodos

O método `main()`, que está presente nas classes executáveis, sempre tem a sintaxe a seguir:

```
public static void main (String[] args) { ... }
```

Modificadores | Tipo de Retorno | Tipo do Argumento | Argumento | Implementação do Método

Qualquer definição de método Java (veja também os métodos `paint()` e `init()` nos exemplos acima) tem uma estrutura semelhante ao `main()`. Entre as chaves **`{`** e **`}`** está *toda* a *implementação* do método (instruções). É o único lugar[2] do programa onde pode existir um procedimento algorítmico. Após o *nome do método* sempre há um par de *parênteses* que podem ou não conter um ou mais argumentos. O *tipo* de cada argumento sempre é declarado e a variável do argumento pode ser usada apenas localmente. Os argumentos são separados por vírgulas:

```
public int calcula(int a, int b, double limite) {
    int z = a + b;    // z, a e b são variáveis locais
    (...)
    return z;      // z é int!
}
```

O método acima retorna um valor inteiro (tipo `int`). Caso ele não retorne algum valor, o valor deve ser `void`. Isto tem que ser declarado antes do *nome* de cada método. O uso de modificadores é opcional e pode afetar a qualidade ou acessibilidade de um método.

No caso do `main`, os modificadores (neste caso, apenas, obrigatórios) são `public` e `static`. Como `main` não retorna valor, ele é declarado `void`. Ele também pode receber como argumento um *vetor de objetos* do tipo `String`, que chamamos, neste exemplo, de `args`.

Os colchetes `[` e `]` indicam que `args` é um *vetor*. `String` é uma classe que pertence a biblioteca fundamental da linguagem Java e caracteriza uma cadeia de caracteres. Toda classe define um *tipo de dados*. A variável `args`, então, pode conter um vetor de cadeias de caracteres, que podem ser palavras, por exemplo. No caso específico do `main`, `args` é utilizado pelo interpretador para armazenar as palavras digitadas após o nome do programa na linha de comando.

Variáveis e campos de dados

Dentro do método `calcula()` que aparece como exemplo acima, a variável `z` é declarada como sendo do tipo int. Variáveis declaradas dentro de métodos são locais àqueles métodos e só existem dentro do bloco formado pelas chaves.

[2] Há algumas exceções mas não convém mencioná-las no momento.

Modificadores Tipo da Variável valor de inicialização (definição)

Em geral, não são precedidas de modificadores. O mesmo ocorre com as variáveis declaradas como argumentos do método.

```
private static int valor = 15;
```

Variáveis também podem ser declaradas e inicializadas fora dos métodos. Neste caso elas passam a ser chamadas de campos de dados pois definem propriedades do objeto ou classe à qual pertencem. Quase sempre têm modificadores. Métodos (ou construtores) ou variáveis que são declarados com o modificador private só podem ser usados dentro da mesma classe. Se forem declarados com modificadores public, pode ser usados de uma outra classe.
Uma *declaração* de variável tem, portanto, a seguinte sintaxe:

```
modificadores tipo nome_da_variável;
```

A *definição* de uma variável ocorre através de uma expressão de atribuição:

```
variável = expressão;
```

que consiste em se atribuir o resultado da expressão que está do lado direito ao nome de variável que está do lado esquerdo. A expressão do lado direito pode ser uma expressão aritmética, booleana ou um método que retorne um tipo de dados compatível com o tipo declarado para a variável.

Toda instrução simples, seja declaração, atribuição ou invocação de método deve, necessariamente, terminar em ponto-e-vírgula. Expressões mais complexas que consistem de várias instruções simples, como as expressões de controle de fluxo, devem ser organizadas dentro de blocos, delimitados por chaves ({ }). A tabela abaixo lista estes e outros separadores usados em Java:

SEPARADOR	DESCRIÇÃO
(...)	separa blocos de argumentos em métodos
[...]	define vetores (arrays)
{ ... }	separa blocos de código
,	separa argumentos ou variáveis em uma declaração
;	separa linhas de código (marca o final)

As variáveis que são declaradas no corpo da classe não são as únicas usadas pelo programa, mas são as únicas que podem ter acesso externo e serem manipuladas pelos métodos. São chamadas de *variáveis de instância* (pertencem aos

objetos criados com aquela classe), se não tiverem modificador `static`; ou *variáveis de classe*, se tiverem modificador `static` (este modificador será explicado mais adiante).

As variáveis que não forem declaradas `static` pertencem a objetos criados por essa classe e não podem ser usadas dentro de métodos também declarados como `static` sem que isto ocorra através de um objeto.

```
public class Teste {
  public int x;          // pertence aos objetos criados com Teste
  public static int y;   // pertence à classe Teste
  public metodo() {
    int z = y + 1; // OK pois y é static!
    int k = x + 1;  // ERRO! variável não está disponível na classe!
    Teste t = new Teste();  // cria objeto desta classe
    int k = t.x + 1;        // usa x através de t.
}
```

Impressão na saída padrão

No nosso primeiro programa aparece o método:

```
System.out.println("cadeia de caracteres");
```

Em Java, pontos "`.`" são usados para separar classes e objetos de seus campos de dados e métodos. Vimos que `System` (`java.lang.System`) é uma classe da biblioteca fundamental da linguagem Java. `System.out` é uma variável pública de `System` que é um *objeto* que representa a saída padrão do seu computador. Objetos podem ter métodos. É o tipo do objeto (ou a classe) que define esses métodos. O método `println()`, portanto, é método definido na classe que criou out. Qual é esta classe? Procure na documentação! (seção a seguir).

Comentários

Há três formas de se representar comentários em Java:

`/* Texto */`

Comentários estilo C. Todo o texto após o "`/*`"e antes do "`*/`" é ignorado pelo compilador, inclusive novas-linhas. Não podem conter outros blocos de comentário do mesmo tipo, já que o comentário termina assim que encontra o primeiro "`*/`".

Exemplos de uso:


```
for (x /*x é um índice */ = 0; x < 5; x++);
```

Correto. O compilador irá ver: `for (x = 0; x < 5; x++);`

```
s = texto + /* plural */ "s";
```

Correto. Para o compilador: `s = texto + "s";`

```
/* s = texto + /* plural */ "s"; */
```

Errado. O compilador vai ler: `"s"; */`

`// Texto`

Comentários estilo C++. Todo o texto que segue o "`//`" é ignorado até o final da linha.

Exemplo:

```
// s = texto + /* plural */ "s";
```

Correto. O compilador ignorará toda a linha.

`/** Texto */`

Comentários especiais para geração de documentação. São idênticos aos comentários estilo C. O compilador ignora o bloco da mesma forma, mas um programa especial que faz geração automática de documentação, o *Javadoc* (parte do JDK) é capaz de extrair seu texto, comandos de escape especiais (iniciados com "`@`") e adicioná-los a um documento HTML. Além dos comentários, o documento HTML gerado contém outras informações extraídas da classe, como superclasses, sintaxe dos construtores, métodos e variáveis.

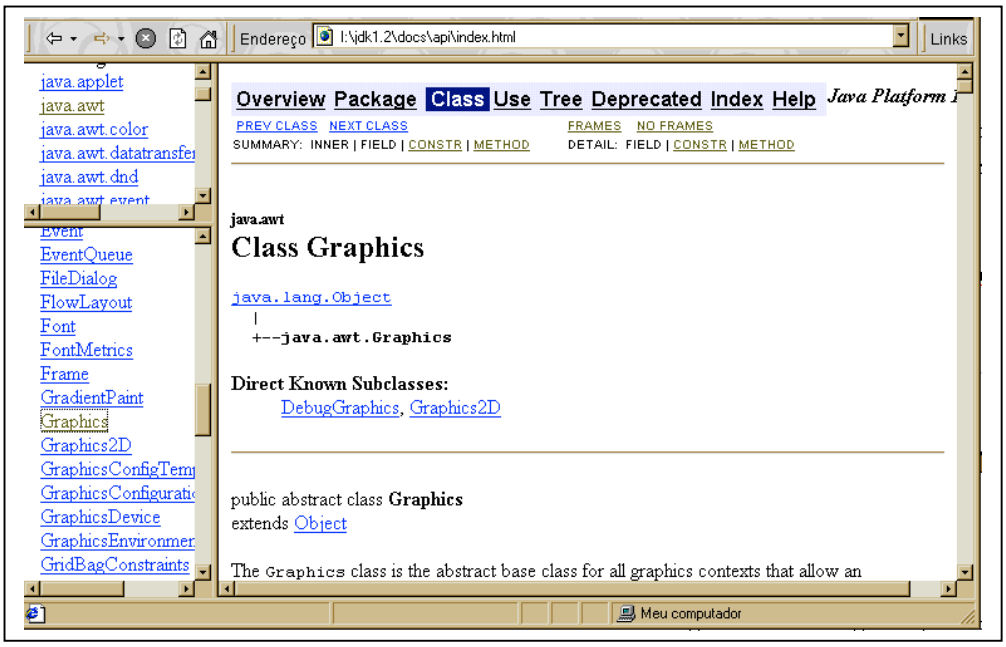
Vamos ver a geração de documentação na prática. Rode o `javadoc` no arquivo `HelloWorld3.java`. Ele vai gerar um arquivo `HelloWorld3.html`, entre outros. Abra esse arquivo no seu browser e veja seu conteúdo.

```
C:\> javadoc HelloWorld3.java
Generating packages.html
generating documentation for class HelloWorld3
Generating index
Sorting 5 items . . . done
Generating tree
```

A documentação gerada tem o mesmo estilo que a documentação da API Java. Se você acrescentar comentários de documentação antes dos métodos, eles aparecerão no HTML como descrições do seu código. Vamos dar uma olhada na documentação em hipertexto. É essencial que você saiba usá-la.

1.4. Como consultar a documentação

A documentação da linguagem Java é distribuída em um pacote à parte do JDK. Consiste de dezenas de milhares de arquivos interligados por hipertexto. Toda a documentação foi gerada com o Javadoc sobre os códigos fonte das classes da API Java. Para abrir a documentação, abra o arquivo `index.html` do subdiretório `/jdk1.2/docs/`. A partir dessa página, que possui vários tutoriais e outras explicações sobre os recursos de Java, procure o link *API & Language*, depois *Java Platform 1.2 Specification*. Você deve encontrar uma página como a seguinte:



A janela se divide em três partes. A do canto superior esquerdo permite que você selecione um pacote da API Java (`java.awt`, por exemplo). Na janela logo abaixo, você pode selecionar uma classe desse pacote (`Graphics`, por exemplo). Quando você fizer isto, na janela principal aparecerá uma página descrevendo todos os métodos, variáveis e construtores da classe selecionada, mostrando uma descrição e a hierarquia de classes em detalhes.

Experimente e explore a documentação. Aprenda a utilizá-la sempre e principalmente enquanto estiver aprendendo Java.

Exercícios

1. Pratique mais com os programas apresentados neste capítulo. Modifique linhas do código e veja os erros que você obtém. É importante se familiarizar com o ambiente de desenvolvimento antes de continuar.
2. Localize o método `drawString()` da classe `java.awt.Graphics` (veja terceiro exemplo) na documentação Java. Veja quais outros métodos a classe Graphics oferece. Tente usar outros como `fillRect()` e `drawOval()`. Veja como funciona o método `setColor()` e tente mudar a cor dos objetos desenhados.
3. Todas as classes que compõem a biblioteca fundamental da linguagem Java são escritas em Java. Se você instalou o JDK, pode encontrar as listagens do código-fonte de todas as classes no seu sub-diretório `src` (por exemplo `c:\java\src` ou `/java/src`). Veja, por exemplo, as listagens de `java/awt/Frame.java`, `java/awt/Label.java` e `java/applet/Applet.java`.
4. Tente entender como funciona o comando `System.out.println()`. Use a documentação e procure na classe `System` pela variável pública `out`. O que ela significa? Ela é uma referência para um objeto de um determinado tipo (ou classe). Que tipo? Qual a classe que foi usada para formar o objeto `out`? Onde está o método `println()`?
5. Descubra como usar o objeto `in` para ler texto da entrada padrão. Escreva um programa que leia uma linha da entrada padrão e escreva a mesma linha na saída.

Recapitulação

Verifique se você aprendeu os principais tópicos deste capítulo:

- Descreva as principais características da linguagem Java.
- Descreva o funcionamento da máquina virtual Java e da coleta de lixo.
- Defina pacote, classe e método.
- Escreva, compile e execute uma aplicação.
- Escreva, compile e execute um applet.
- Localize o método de uma determinada classe (por exemplo, `add()` de `Frame`) na documentação da linguagem.

