Capítulo 1

Introdução

A programação orientada a objetos tem se tornado um importante instrumento para o desenvolvimento de sistemas computacionais complexos. Seu uso tem levado à construção de software com maior grau de qualidade. Dentre os principais aspectos de qualidade influenciados pela nova tecnologia podemos destacar a facilidade de modelagem, a reusabilidade e a manutenibilidade.

Uma das linguagens de programação orientada a objetos mais usadas no momento é Java. Esta linguagem apresenta uma série de características que a tornam muito atraente para a construção de sistemas computacionais, tais como, portabilidade, facilidade de integração, segurança e confiabilidade.

Além de incorporar o paradigma de orientação a objetos, o qual ainda é desconhecido por muitos programadores, Java possui uma grande variedade de características inovadoras. Embora seja uma linguagem muito bem projetada, a orientação a objetos somada a estas novas características acabam tornando o aprendizado da linguagem um interessante desafio intelectual aos programadores. Este curso procura facilitar o enfrentamento desse desafio através da introdução dos conceitos de Java e de orientação a objetos de um modo gradual e claro.

Este curso pressupõe um conhecimento prévio a respeito de construção de algoritmos e de alguma linguagem de programação estruturada como Pascal ou C. Nossa estratégia para explicar os conceitos de Java é utilizar exemplos simples e didáticos sempre que possível, somente partindo para exemplos mais complexos quando o conceito assim demandar. Outra pressuposição deste curso é que os alunos nunca viram ou ouviram falar de Java. Isto implica na necessidade de ensinar Java desde os seus conceitos mais elementares.

Neste capítulo apresentaremos alguns conceitos operacionais sobre linguagens de programação, discutiremos o que é a plataforma Java e introduziremos alguns exemplos iniciais de programas Java.

1.1 – Objetivos do Curso

O objetivo deste curso é ensinar os conceitos fundamentais de programação orientada a objetos e da linguagem Java de uma maneira aprofundada e clara. Nosso intuito não é apenas apresentar os diversas primitivas da linguagem e mostrar como elas podem ser usadas. Mais do que isso, pretende-se dar uma noção de como e porque elas foram incluídas na linguagem e como são implementadas. Acreditamos que com essa visão mais aprofundada, os programadores terão condições de utilizar esta poderosa linguagem de maneira a aproveitar todas as suas potencialidades.

Ao final do curso o aluno será capaz de criar programas em Java usando os conceitos de tipos abstratos de dados, encapsulamento, ocultamento de informação, herança e polimorfismo. Além disso, o aluno terá aprendido a modelar problemas de programação

de um modo diferenciado, enxergando o mundo a ser representado através dos objetos que o compõem.

1.2 – Conceitos Básicos de Linguagens de Programação

Nesta seção introduziremos alguns conceitos importantes para o estudo de linguagens de programação (LP). Estes conceitos permitirão ao aluno analisar criticamente os conceitos de Java e compreender as razões pelas quais certas características foram incluídas na linguagem. Apresentaremos, inicialmente, propriedades que são desejáveis em linguagens de programação para tornar os programas e a atividade de programação mais produtivos. Explicaremos brevemente o conceito de tradução de programas. Discutiremos porque a alocação dinâmica de memória é importante. Por fim, discutiremos a evolução do conceito de abstração nas linguagens de programação até a orientação a objetos.

1.2.1 – Propriedades Desejáveis em Linguagens de Programação

No início da era da computação, o principal fator que reduzia a produtividade no desenvolvimento de software eram os recursos computacionais limitados. Contudo, com o avanço da tecnologia, a tarefa de programação passou a ser o fator determinante da produtividade. A partir desta constatação, o aproveitamento do tempo do profissional de programação se tornou um conceito central no processo de desenvolvimento. Consequentemente, as propriedades desejáveis nas LPs devem enfatizar este aspecto. Apresenta-se, a seguir, uma lista de propriedades desejáveis em LPs.

a) **Legibilidade:** Deve ser possível seguir as instruções de um programa, de forma a entender o que está sendo feito, bem como facilitar a descoberta de erros que possam existir com a maior antecedência. LPs que requerem o uso extensivo do comando *goto* impossibilitam ler o programa em um único passo e incentivam a ocorrência de programação macarrônica.

b) **Redigibilidade:** Permitir que o programador se concentre nos algoritmos centrais, sem se preocupar com detalhes. LPs que requerem muita programação de entrada e saída tendem a obscurecer os algoritmos centrais nos programas.

c) **Confiabilidade:** Deve fornecer mecanismos que incentivem a construção de programas confiáveis. LPs podem promover a confiabilidade de programas facilitando a existência de ferramentas computacionais que verifiquem a ocorrência de erros nos programas. Por exemplo, LPs que requerem a declaração de variáveis, tais como C, Pascal, e Modula-2, facilitam a verificação de erros de digitação de nomes.

d) **Eficiência:** Uma LP pode promover a eficiência de programas incentivando o uso de mecanismos computacionalmente eficientes. De acordo com o tipo de aplicação, certas LPs não devem ser usadas. LPs com checagem de tipos durante a execução são menos eficientes. JAVA verifica índice de vetores em execução, enquanto que C não o faz.

e) **Facilidade de aprendizado:** Programador deve aprender a linguagem com facilidade. LPs com muitas características e múltiplas maneiras de realizar a mesma funcionalidade, tal como C++, tendem a ser mais difíceis de aprender.

f) **Reusabilidade de Código:** Deve ser possível reutilizar o mesmo código para diversas aplicações. São exemplos de mecanismos que permitem reuso: subprogramas com parâmetros e a modularização.

g) **Flexibilidade:** Uma LP pode promover a flexibilidade de programas fornecendo mecanismos que permitam a sua adaptação a diferentes contextos. Por exemplo, a declaração de constantes facilita a realização de modificações nos programas.

h) **Harmonia com a metodologia de projeto:** Isto possibilita que a implementação de um sistema reflita o seu projeto, evitando adaptações e distorções no projeto e perda de correspondência. Pode-se observar que algumas linguagens de programação são mais adequadas quando se utilizam certos métodos de projeto. Por exemplo, enquanto C é mais adequada ao método de projeto hierárquico-funcional, JAVA é mais adequada ao método orientado a objetos.

1.2.2 – Tradução de Programas

O processador de um computador só é capaz de executar um conjunto restrito de operações de hardware muito básicas. Instruções de máquina consistem de vários bytes armazenados na memória principal do programa que instruem o processador a executar uma operação de hardware. Uma coleção de instruções de máquinas na memória principal se chama programa em linguagem de máquina ou, como é mais conhecido, programa executável.

Programadores raramente escrevem programas em linguagem de máquina. Arquivos de programas executáveis contém tipicamente centenas de milhares ou mesmo milhões de instruções de máquina. Seria extremamente difícil e improdutivo criar programas assim.

A maior parte dos programassão criadas usando uma linguagem de programação de alto nível, como BASIC, C, C++ ou Java. Com este tipo de linguagem de programação, o programador pode utilizar operações muito mais poderosas que as instruções de máquina.

Por exemplo, a seguinte linha de programa em Java,

```
int sum = 0;
```

corresponde a centenas de operações de máquina que reservarão um espaço da memória principal capaz de armazenar um número, colocar o valor zero lá e ajustar as outras partes do programa de modo que elas possam usar este espaço. Obviamente, é muito mais fácil para um programador humano escrever essa operação em C do que em instruções de máquina.

Um programa em uma LP de alto nível consiste de linhas de texto escritas através de um editor e salvas normalmente em um arquivo em memória secundária. Como todo arquivo, ele consiste de uma sequência de bytes. Contudo, se tentarmos carregá-lo na memória principal e executá-lo, ele não realizará o que foi programado porque os seus bytes não correspondem às instruções de máquina que deveriam ser executadas. O arquivo que contém o texto do programa é chamado de arquivo fonte.

Para poder executar o programa é necessário traduzir o conteúdo do arquivo fonte para o correspondente programa executável, isto é, as instruções de máquina que são capaz de executá-lo. Isso é feito através de um outro programa, chamado de tradutor. O programa

executável é normalmente salvo em memória secundária num arquivo conhecido como executável. Este arquivo pode ser posteriormente carregado na memória principal e executado. Os programas tradutores são mais conhecidos como compiladores, embora, como veremos adiante, nem toda tradução corresponda a uma compilação.

1.2.2.1 – Compilação e Interpretação

Existem duas maneiras básicas de efetuar a tradução de programas: a compilação e a interpretação.

A compilação traduz todo o programa fonte para instruções de máquina e, depois, executa-as. A figura 1.1 ilustra o procedimento de compilação.

Figura 1. 1 - Compilação

A grande vantagem da compilação é a execução muito mais rápida, visto que durante a execução não é necessário perder tempo na tradução nem com a realização de muitas verificações. Outra vantagem é requerer apenas que o código executável esteja carregado na memória. A grande desvantagem da compilação é a falta de portabilidade. Se um programa é traduzido para um executável em um ambiente que usa o sistema operacional Windows ele não executará em um ambiente Linux. Será necessário recompilar o programa em cada ambiente onde se deseja executá-lo. Infelizmente, o processo de recompilação não funciona tão bem como o esperado. Normalmente, existem pequenas incompatibilidades que acabam requerendo mudanças no código fonte e um grande esforço do programador.

A interpretação traduz um comando de cada vez, executando-o antes de traduzir o comando seguinte. O interpretador age como um simulador de um computador virtual que entende as instruções da LP. A figura 1.2 ilustra o procedimento de interpretação.

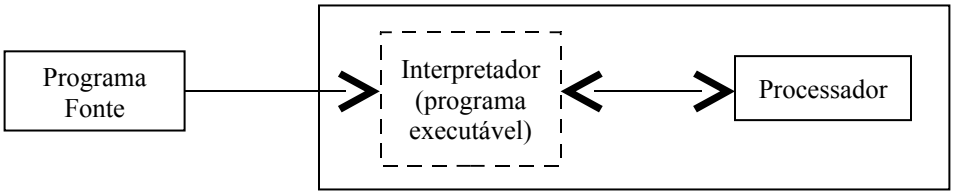

Figura 1. 2 - Interpretação

Nesta figura, o programa fonte poderia ter sido escrito em BASIC. Ele será interpretado por um interpretador BASIC, o qual está executando na máquina. O interpretador lerá cada comando do fonte e fazer o que ele determina. Durante a execução tanto o interpretador como o fonte estão carregados na memória principal. O interpretador consiste de instruções de máquina e o fonte de comandos na linguagem BASIC que o interpretador reconhece. Do ponto de vista do programa BASIC, parece que os seus comandos estão sendo diretamente executados por um tipo de máquina. A palavra virutal

tem sido usada em situações onde o software é utilizado para fazer algo parecer com a coisa real. Neste caso, parece que temos uma máquina que pode executar diretamente os comandos em BASIC, de modo que diríamos ter uma máquina virtual BASIC.

A vantagem da interpretação é a portabilidade. Como o interpretador é um programa que executa os comandos da LP e não, as instruções da máquina, basta que tenhamos um interpretador da LP em cada ambiente e o mesmo programa poderá ser executado em cada um deles. Os problemas da interpretação são a necessidade de manter em memória principal tanto o programa fonte quanto o interpretador e, principalmente, a lentidão da execução. O principal motivo da lentidão é que o interpretador necessita realizar a tradução de comandos muito complexos durante a execução. Além disso, em comandos de repetição, ele deve repetir a tradução dos comandos internos a repetição inúmeras vezes. De fato, essa ineficiência acaba comprometendo a implementação de tradutores como interpretadores puros.

Uma solução que vem sendo adotada por algumas linguagens é combinação dos métodos de compilação e interpretação. Este método, chamado de híbrido, se divide em duas etapas: compilação para código intermediário e interpretação deste código. As vantagens desta opção são que a execução é mais rápida que interpretação pura porque instruções do código intermediárias são muito mais simples e o código intermediário pode ser executado em várias plataformas diferentes, sendo, portanto, portável. JAVA adota o método híbrido. O código intermediário é chamado de byte code. O interpretador de byte code é a JVM (Java Virtual Machine). Cada plataforma possui a sua própria JVM.

A figura 1.3 apresenta os diversos tipos de modelos de tradução.

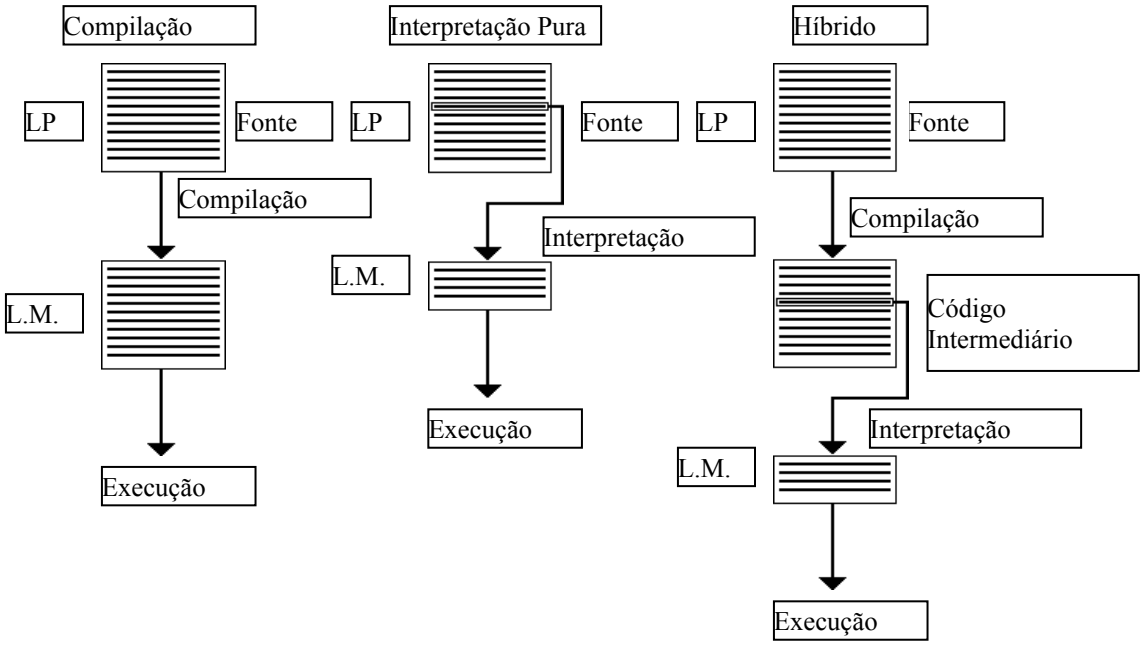

Figura 1. 3 - Modelos de Tradução

1.2.3 – Alocação Dinâmica de Memória

A memória principal de um computador consiste de uma enorme sequência contígua e finita de bits. Contudo, a menor unidade de memória que pode ser diretamente endereçada corresponde normalmente ao tamanho da palavra do computador (8 bits, 16 bits, 32 bits, etc.). Podemos imaginar, então, a memória como sendo um vetor de tamanho finito cujos elementos correspondem ao tamanho da palavra do computador, o qual chamaremos de vetor de memória.

Quando da execução de um programa, além do código executável, deve ser reservado um espaço na memória para armazenar os dados que serão manipulados pelo programa. Uma forma de se fazer isso seria identificar quais as variáveis que são utilizadas no programa, qual o seu tamanho e reservar um espaço na memória correspondente a soma dos tamanhos de todas as variáveis. A primeira linguagem de programação de alto nível, FORTRAN, adotou esta abordagem. Contudo, essa alternativa apresentava os seguintes problemas:

- Muitas vezes, a memória disponível acaba sendo mal utilizada. Isso ocorre porque o tamanho das variáveis que serão necessárias pode variar de execução para execução de programa. Tipicamente, a solução adotada por programadores destas linguagens é estimar e estabelecer como tamanho da variável o máximo que pode ser usado. Um problema desta solução é que, eventualmente, estas estimativas podem falhar, ocasionando erros inconvenientes. Outro problema é que se reserva mais espaço para as variáveis do que é necessário no caso geral. Além disso, frequentemente, na execução de um programa, somente um subconjunto dos subprogramas é invocado. Nestes casos, todo o espaço ocupado pelas variáveis dos subprogramas não utilizados foi reservada desnecessariamente. Toda essa má utilização da memória pode impedir a execução de programas que poderiam ser executados caso houvesse um melhor aproveitamento.
- Não é possível implementar subprogramas recursivos, em geral. Subprogramas recursivos são aqueles que invocam a si mesmo para serem executados. Se não se pode compartilhar a estrutura de dados usada pelo subprograma recursivo, cada chamada ao subprograma requer que seja alocada uma nova estrutura de dados a ser manipulada. Isso só pode ser realizado em tempo de execução.

A solução para lidar com esses problemas é permitir a alocação dinâmica de memória, isto é, somente reservar espaço para as variáveis quando o tamanho necessário for conhecido e quando elas forem efetivamente necessárias. Isto só pode ocorrer durante a execução do programa. Para realizar alocação dinâmica, os programas devem implementar e/ou usar um mecanismo de gerenciamento dinâmico de memória.

Uma possibilidade para implementação deste tipo de mecanismo seria alocar, na medida que se constate a necessidade e se identifique o tamanho de uma variável, o número de palavras correspondentes ao seu tamanho no vetor de memória. Para uma melhor utilização da memória, as diferentes variáveis seriam alocadas contiguamente no vetor. Esta solução, contudo, apresenta vários problemas. Primeiramente, pode ser necessário aumentar o tamanho das variáveis durante uma mesma execução de um programa. Além disso, variáveis podem ser úteis em apenas um intervalo da execução do programa. Estas situações demandariam que fossem feitas exclusões e movimentações de elementos no vetor. Tais operações demandam bastante esforço computacional para serem realizadas.

Ademais, elas requeririam que as referências as variáveis no código executável fossem atualizadas com o seu endereço absoluto sempre que elas fossem alocadas ou realocadas. Essas operações certamente prejudicariam a execução eficiente do programa.

A forma mais comum de implementação dos mecanismos de gerenciamento de memória de execução de programas subdivide a memória do programa em duas áreas: a área de pilha e a área de monte. A pilha é utilizada para resolver o problema de alocação de subprogramas recursivos e da alocação dinâmica dos dados específicos dos subprogramas. Toda vez que um subprograma é invocado, reserva-se no topo da pilha espaço suficiente para armazenar as variáveis locais e os parâmetros do subprograma. Ao se encerrar o subprograma, o espaço destinado a ele no topo da pilha é liberado. Isto só é possível porque o último subprograma a ser invocado é sempre o primeiro a ser encerrado. Com esta política de uso da pilha, as variáveis locais e parâmetros só ficam alocadas enquanto podem ser necessárias, isto é, durante a execução do subprograma. Por se tratar de uma pilha, a liberação da área de memória do subprograma envolve apenas o deslocamento do marcador do topo da pilha, não requerendo movimentações de elementos no vetor de memória. Isso torna esse mecanismo bastante eficiente. A figura 1.4 apresenta um exemplo de uso da pilha. Neste exemplo, o programa de nome *p* possui duas variáveis inteiras chamadas *a* e *b*, e o subprograma *f* possui dois parâmetros *x* e *y* e uma variável local *z*. O momento da execução retratado na figura ocorre durante a chamada de *f* pelo programa *p*.

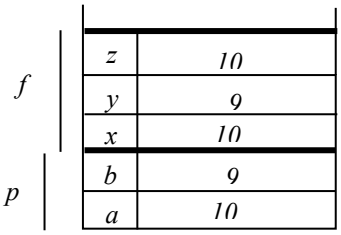

Figura 1. 4 - Estado da Área de Pilha

Contudo, o problema das variáveis cujo tamanho se modifica em tempo de execução ainda permanece. Se decidíssemos alocar estas variáveis na pilha, teríamos um grande problema quando fosse necessário aumentar o tamanho delas. Seria preciso retirar todas as variáveis que estariam acima dela na pilha, copiando-as em outra área de memória, aumentar o espaço reservado para a variável e, posteriormente, empilhar as variáveis retiradas na mesma ordem em que haviam sido alocadas originalmente. Isto seria claramente ineficiente.

A solução para o problema do aumento dinâmico de variáveis é o uso do monte. O monte é uma área de memória que não obedece a uma regra bem estabelecida como a pilha. Se existe necessidade de memória, reserva-se um espaço no monte onde existir espaço contíguo suficiente para alocar o tamanho de memória necessária. Neste momento, você deve estar se perguntando porque toda essa confusão de dividir a memória em pilha ou monte. Porque não usar toda a memória apenas como se fosse um monte? A resposta para essa pergunta é que as coisas não são tão simples assim. A alocação dinâmica de memória implica que subáreas da memória serão alocadas e desalocadas ao longo da execução do programa. Como vimos, alocar contiguamente essas áreas não é uma boa

solução. Se alocarmos as novas áreas sempre após o último bloco alocado e não reutilizarmos as áreas desalocados, rapidamente, a memória se esgotará. Se, por outro lado, tentarmos utilizar as áreas desalocadas, teremos de ficar fazendo várias movimentações de dados, o que fatalmente provocará ineficiência na execução.

Para contornar este problema e poder utilizar o monte de maneira razoável, embora sempre de forma menos eficiente do que a alocação via pilha, é necessário possuir uma estrutura de dados adicional responsável por controlar as áreas livres e já alocadas. De acordo com a linguagem de programação, o controle desta estrutura pode ser feito pelo programador (por exemplo, em C, C++ e Pascal) ou pelo próprio sistema que implementa a LP (neste caso, utiliza-se um mecanismo chamado de coletor de lixo, como é o caso de Java). Enquanto a primeira solução permite a construção de programas mais eficientes (o construtor do programa é quem diz em qual momento deve ser alocada e desalocada a memória), a segunda solução torna a vida do programador muito mais fácil (ele normalmente não precisa se preocupar com a gerência da memória).

Independentemente de ser o sistema ou o programador quem faz este gerenciamento de memória, o conceito que permite implementar esta estrutura de dados dinâmica é o de endereçamento indireto. Tal conceito permite acessar áreas de memória de maneira indireta, isto é, seguindo uma cadeia de endereços para chegar ao ponto desejado. A forma mais frequente de implementação do endereçamento indireto é através do uso de ponteiros. Ponteiros são variáveis que armazenam um endereço de memória. Com o uso de ponteiros é possível manter estruturas de dados (tipicamente, listas) contendo informações sobre as áreas livres e ocupadas da memória. Assim, ao invés de ter de alocar as áreas de memória no monte de modo contíguo, as variáveis são alocadas em qualquer ponto indicado pela lista de áreas livres, desde que caibam. Sempre que uma variável tem necessidade de ser aumentada, aloca-se em algum ponto do monte o novo tamanho necessário para a variável. Copia-se a antiga variável sobre a nova, acrescentando o que for preciso. Esta nova área é incluída na lista de áreas ocupadas ao mesmo tempo em que se libera a área anteriormente ocupada da lista de áreas ocupadas. Procedimento similar é realizado para a lista de áreas livres. A figura 1.5 mostra a realocação de uma variável na área de monte. Nesta figura, a mesma pilha da figura 1.4 é representada, mas substituindo o tipo da variável *z* de inteiro para ponteiro. Esta variável é usada para apontar para uma variável alocada no monte cujo valor é 10. Quando há necessidade de aumentar esta variável, um novo espaço no monte é alocado, a variável *z* passa a apontar para esta nova área e o espaço ocupado anteriormente por aquela variável é desalocado.

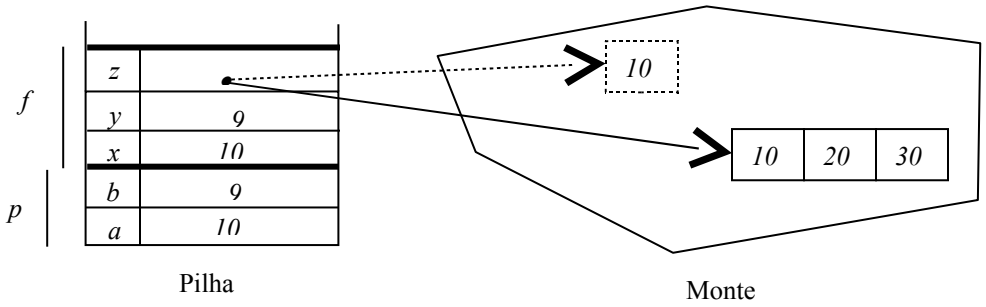

Figura 1. 5 - A Área de Monte

1.2.3.1 – Modelo de Memória de Programas

O modelo mais comum de implementação das linguagens de programação chamadas Algol-like (Java é uma delas) utiliza um modelo de memória particionado em três regiões: a área do código, a área da pilha e a área do monte. A figura 1.6 ilustra este modelo.

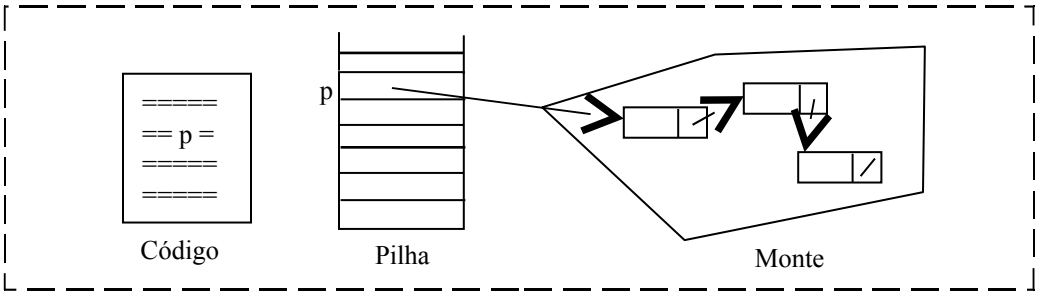

Figura 1. 6 - Modelo de Memória de Programas

No código existem referências estáticas e dinâmicas (isto é, aquelas que ficam pendentes até que se saiba o endereço das variáveis na pilha durante a execção) às váriaveis alocadas na pilha. Estas, por sua vez, ou contém valores, ou contém referências a variáveis alocadas no monte.

1.2.4 – Evolução do Conceito de Abstração nas Linguagens de Programação até a Orientação a Objetos

Pode-se dizer que a capacidade de lidar com a complexidade dos problemas é diretamente relacionada com o tipo e qualidade das abstrações utilizadas. Todas as linguagens de programação oferecem abstrações de diferentes tipos. Linguagens de montagem (*assemblers*) são uma pequena abstração da máquina subjacente. As LPs imperativas, tais como FORTRAN, BASIC e C, são abstrações das linguagens de montagem. Muito embora elas sejam um grande avanço em termos de abstração, elas ainda requerem que se pense em termos da estrutura do computador ao invés da estrutura do problema que se está tentando resolver. O programador deve fazer um mapeamento entre o modelo do problema a ser resolvido para o modelo da máquina que executará o programa. Normalmente, este mapeamento acaba provocando a construção de programas que são difíceis de escrever e manter. A alternativa para a modelagem da máquina é a modelagem do problema que se quer resolver.

Existem várias LPs que usam diferentes paradigmas que adotam estas alternativas (por exemplo, LISP, PROLOG). Dentro deste escopo de modelagem do problema, o paradigma que tem sido mais aceito é o de orientação a objetos. Neste paradigma os elementos do problema são representados como "objetos" (é importante esclarecer que também será necessário criar e operar objetos que não possuem uma analogia com os objetos do problema a ser modelado). A idéia é possibilitar a adaptação do programa à linguagem do problema pela adição de novos tipos de objetos, de modo que, quando se lê o código descrevendo a solução, se está lendo palavras que expressam o problema. Portanto, a programação orientada a objetos permite que se descreva o problema em

termos do problema, ao invés de em termos de sua solução. Ainda existe uma conexão com o computador. Cada objeto se assemelha a um pequeno computador (tem um estado e um conjunto de operações que pode realizar). No entanto, isto não parece ser uma analogia ruim. Os objetos do mundo também possuem características e comportamentos.

1.3 – A Plataforma Java

O mundo da computação sempre possuiu diferentes plataformas, tais como, Microsoft Windows, Macintosh, OS/2, UNIX ® e NetWare ®. Como vimos, programas nas linguagens de programação devem ser compilados separadamente para executar em cada plataforma. O programa executável que roda em uma plataforma não roda na outra.

A plataforma Java surgiu neste contexto para produzir sistemas de computação altamente interativos, dinâmicos e seguros que possam ser executados e distribuídos em rede. O que torna a plataforma Java diferente das demais é que ela se situa em um nível acima de todas as outras plataformas. Os programas executáveis na plataforma são escritos em bytecodes, os quais são instruções de uma máquina virtual. Assim, os programas executáveis não se tornam específicos de qualquer máquina física. Quando compilado, um programa escrito em Java gera um arquivo de bytecodes que pode ser executado em sistema operacional e/ou qualquer máquina onde a plataforma Java esteja presente.

Esta portabilidade é conseguida porque o núcleo da plataforma Java é a máquina virtual Java (JVM). Enquanto cada plataforma subjacente possui sua própria implementação da JVM, existe uma única especificação da máquina virtual. Por conta disto, a plataforma Java fornece uma interface padrão e uniforme para a construção de sistemas de computação sobre qualquer hardware. A plataforma Java se torna, portanto, ideal para a Internet, onde um mesmo programa deve ser capaz de executar em qualquer computador do mundo.

A plataforma Java foi projetada com o objetivo de fornecer a capacidade conhecida como "write once, run anywhere" (escreva uma vez, execute em qualquer lugar). A linguagem Java é a rampa de entrada para a plataforma Java. Programas escritos na linguagem Java e, posteriormente compilados, executarão na plataforma Java.

Portanto, a plataforma Java, introduzida no mercado em 1995 pela *Sun Microsystem*s, é mais do que uma linguagem de programação bem sucedida resultante de um trabalho consistente de pesquisa e desenvolvimento. Trata-se de um ambiente completo de desenvolvimento e execução de programas que exibe facilidades, tais como, orientação a objetos, portabilidade de código, características de segurança e facilidade de integração a outros ambientes, destacando-se a Internet.

Java foi projetada para enfrentar os desafios impostos pelo desenvolvimento de aplicações destinadas a serem executadas em ambientes distribuídos heterogêneos. Entre estes desafios, se destacam as necessidades de que as aplicações sejam distribuídas de modo seguro, que consumam o mínimo de recursos dos sistemas, que possam executar em qualquer plataforma de *hardware* ou *software* e que possam ser estendidas dinamicamente.

Figura 1. 7 - Logo do Java

1.3.1 – Pequeno Histórico

Java originou-se a partir de um projeto de pesquisa que objetivava desenvolver sistemas de computação para uma ampla variedade de dispositivos de rede e sistemas embutidos. Tudo começou em 1991, com um pequeno grupo de projeto da *Sun Microsystems* denominado *Green* que pretendia criar uma nova geração de computadores portáteis inteligentes, capazes de se comunicar de muitas formas, ampliando suas potencialidades de uso. A meta principal era construir uma plataforma de operação pequena, confiável, portável, distribuída e em tempo real. Quando o projeto se iniciou, a linguagem escolhida foi C++ por causa de suas características e para aproveitar a experiência dos integrantes do grupo no desenvolvimento de produtos. Mas, mesmo o C++ não permitia realizar com facilidade tudo aquilo que o grupo desejava..

James Gosling, coordenador do projeto, decidiu então pela criação de uma nova linguagem de programação que pudesse conter tudo aquilo que era considerado importante e que ainda assim fosse simples, portável e fácil de programar. Surgiu assim a linguagem interpretada *Oak* (carvalho em inglês) batizada assim dada a existência de uma destas árvores em frente ao escritório de Gosling. Para dar suporte a linguagem também surgiu o *Green OS* e uma interface gráfica padronizada.

Após dois anos de trabalho o grupo finaliza o *Star7* (ou *7), um avançado PDA *(Personal Digital Assistan*t) e em 1993 surge a primeira grande oportunidade de aplicação desta solução da *Sun* numa concorrência pública da *Time-Warner* para desenvolvimento de uma tecnologia para TV a cabo interativa, injustamente vencida pela SGI *(Silicon Graphics Inc.).*

O *Oa*k, então rebatizado Java devido a problemas de *copyrigh*t, continua sem uso definido até 1994, quando estimulados pelo grande crescimento da Internet, Jonathan Payne e Patrick Naughton desenvolveram o programa navegador *WebRunne*r, capaz de efetuar o *download* e a execução de código Java via Internet. Apresentado formalmente pela *Sun* como o navegador *HotJava* e a linguagem Java no *SunWorld'*95, o interesse pela solução se mostrou explosivo. Poucos meses depois a *Netscape Corp.* lança uma nova versão de seu navegador *Navigator* também capaz de efetuar o *download* e a execução de pequenas aplicações Java, então chamadas *applet*s. Assim se inicia a história de sucesso do Java.

Numa iniciativa também inédita a *Sun* decide disponibilizar o Java gratuitamente para a comunidade de desenvolvimento de *softwar*e, embora detenha todos os direitos relativos à linguagem e as ferramentas de sua autoria. Surge assim o *Java Developer's Kit* 1.0

(JDK 1.0). As plataformas inicialmente atendidas foram: *Sun Solaris* e *Microsoft Windows 95/N*T. Progressivamente foram disponibilizados *kits* para outras plataformas, tais como, *IBM OS/*2, *Linux* e *Applet Macintos*h.

Em 1997, surge o JDK 1.1 que incorpora grandes melhorias para o desenvolvimento de aplicações gráficas e distribuídas e no início de 1999 é lançado o JDK 1.2, contendo muitas outras melhorias. A partir da versão do JDK 1.2, a plataforma Java passa a se chamar Java 2.

Atualmente a *Sun* vem liberando novas versões ou correções a cada nove meses bem como novas API *(Application Program Interfac*e) para desenvolvimento de aplicações específicas. A *Sun* passa a chamar o JDK de SDK A última versão disponível é o JDK 1.3.1. No momento de escrita desta apostila, a versão beta do JDK 1.4 já está disponível.

1.3.2 – Características Importantes

A linguagem Java e seu ambiente foi projetada para resolver um número variado de problemas que ocorrem na prática da programação. Para tanto, ela exibe importantes características que, em conjunto, diferenciam-na de outras linguagens de programação:

- **Orientação a Objetos**

Java é uma linguagem puramente orientada à objetos pois, com exceção de seus tipos primitivos de dados, tudo em Java são classes ou instâncias de uma classe. Java oferece todos os requisitos necessários para uma linguagem ser considerada orientada a objetos, isto é, possuir mecanismos de abstração, encapsulamento, hereditariedade e polimorfismo.

As necessidades de sistemas com arquiteturas distribuídas e/ou cliente-servidoras coincidem com as características de encapsulamento e passagem de mensagens do paradigma orientado a objetos. Para funcionar com ambientes baseado em redes cada vez mais complexos, os sistemas de programação necessitam adotar os conceitos de orientação a objetos. Java oferece uma plataforma que é ao mesmo tempo eficiente e elegante. Programadores Java podem acessar bibliotecas existentes de objetos previamente testados que fornecem desde tipos básicos de dados, passando por interfaces para redes e entrada e saída até interfaces gráficas com os usuários. Estas bibliotecas podem ser estendidas para fornecer novos comportamentos.

Um princípio importante que orientou os projetistas de Java foi manter a linguagem semelhante a C++, herdando muito da sua sintaxe. Isto acabou tornando a linguagem familiar aos programadores C++, enquanto ao mesmo tempo retirou algumas das fontes de complicação na programação em C++. Com isso, programadores de C++ podem migrar facilmente para Java e se tornar produtivos rapidamente. Outro princípio foi tentar manter a linguagem tão simples quanto possível. Assim, ela pode ser aprendida mais facilmente que C++ e os programadores podem se tornar produtivos mais rapidamente.

- **Portabilidade**

Como já vimos, Java é uma linguagem independente de plataforma pois os programas Java são compilados para uma forma intermediária de código denominada *bytecodes* que utiliza instruções e tipos primitivos de tamanho fixo, e uma biblioteca de classes padronizada. Os *bytecodes* são como uma linguagem de máquina destinada a uma única

plataforma, a máquina virtual Java (JVM – *Java Virtual Machin*e), um interpretador de *bytecode*s. Pode-se implementar uma JVM para qualquer plataforma. Assim, temos que um mesmo programa Java pode ser executado em qualquer arquitetura que disponha de uma JVM.

Dentro dos ambientes de rede heterogêneos, as aplicações Java devem ser executadas em cima de uma variedade de sistemas operacionais e interoperar com várias aplicações escritas em outras linguagens de programação.

Contudo, a neutralidade oferecida pela arquitetura da plataforma Java é apenas um fator para a construção de sistemas portáveis. Java vai mais adiante na questão da portabilidade estabelecendo um padrão rígido para a linguagem. Java fixa os tamanhos dos tipos básicos e o comportamento de seus operadores aritméticos. Ao contrário de C e C++, os programas fonte são os mesmos para as diferentes plataformas pois não existe incompatibilidades de tipos de dados entre as diferentes arquiteturas de hardware e software.

- **Confiabilidade**

Java foi projetada para a criação de programas altamente confiáveis. Java certamente não elimina a necessidade do programador adotar boas práticas de programação. Contudo, Java permite eliminar certos tipos de erros de programação, o que torna a tarefa de construir programas confiáveis muito mais fácil. Java realiza uma verificação extensiva do programa em tempo de compilação, seguida por um segundo nível de verificação em tempo de execução. As próprias características da linguagem guiam os programadores a se habituar com técnicas de programação mais confiáveis.

Uma das características que tornam os programas em Java mais confiáveis é a ausência do conceito de ponteiros. Já se constatou que a programação com ponteiros é uma das maiores causadoras de erros de programação. Por essa razão, Java simplesmente eliminou este conceito da linguagem. Como consequência, Java não permite a manipulação direta de endereços de memória. Por outro lado, isso não quer dizer que o sistema Java nunca utilize o mecanismo de ponteiros. A gerência de memória realizada pela JVM é toda baseada em ponteiros. No entanto, os ponteiros são utilizados apenas internamente pela JVM. O programador não precisa (nem pode) criar e manipular variáveis deste tipo. Adicionalmente, a JVM possui um mecanismo automático de gerenciamento de memória conhecido como coletor de lixo, que recupera a memória alocada para objetos não mais referenciados pelo programa, eliminando assim a necessidade do programador se preocupar com a liberação de memória não mais utilizada.

- **Dinâmica e Distribuída**

Java é uma linguagem dinâmica, isto é, qualquer classe de Java pode ser carregada a qualquer momento pelo interpretador Java. Mais interessante que isso, classes somente são carregadas quando são necessárias. Assim, mesmo que um programa crie objetos de várias classes, somente aquelas que forem efetivamente necessárias durante uma execução serão carregadas. O procedimento do interpretador Java é o de carregar as classes sob demanda, ou seja, quando é necessário criar um primeiro objeto daquela classe.

Java também é uma linguagem distribuída. Ela oferece suporte de alto nível para a criação de aplicações em rede. Por exemplo, a classe `URL` e as classes do pacote `java.net` tornam o uso de um arquivo ou recurso remoto quase tão fácil quanto ler um arquivo local. Java também possibilita a invocaçào de métods de objetos remotos, com se fossem objetos locais.

A natureza distribuída de Java se revela ainda mais quando combinada com sua característica dinâmica. Juntas, estas características tornam possível a um interpretador Java carregar e executar código de diferentes partes da Internet. Por exemplo, uma determinada aplicação pode identificar dinamicamente que necessita operar sobre um tipo de dados oferecido por um repositório de classes na rede. O interpretador Java pode, então, baixar dinamicamente o código deste tipo e usá-lo na aplicação sendo executada.

Estas características também permitem executar um *applet* Java. *Applets* são programas especiais que podem ser carregados e executados através de um *Web browser* que suporte Java, isto é, tenha uma JVM associada. Com o apoio dos *browsers*, pode-se garantir a evolução dos serviços fornecidos por uma aplicação *on-line* através da atualização transparente do código da aplicação. Em outras palavras, sempre que um novo serviço é prestado não é necessário fazer a distribuição da nova versão da aplicação para os consumidores. Basta alterar o código do *applet* no servidor e todos os usuários poderão imediatamente utilizar a nova versão.

- **Segurança**

Uma das mais importantes características de Java é a segurança que proporciona. Isto é especialmente importante por causa da natureza distribuída de Java. Sem garantias de segurança, você certamente não gostaria de baixar código de um site aleatório na Internet e deixá-lo executar em seu computador. Java foi projetado considerando a segurança como um dos fatores fundamentais. Java oferece várias camadas de controle da segurança para a proteção contra código malicioso. Com isso, Java impede a intrusão de código desautorizado tentando criar viroses ou invadir sistemas de arquivos.

Na camada mais inferior, a segurança caminha lado a lado com a confiabilidade. Programas em Java não podem manipular diretamente endereços de memória ou acessar posições de vetores além daquelas previamente declaradas.

A segunda linha de defesa é a verificação de bytecodes que o interpretador Java realiza sempre que código não confiável é carregado. Essa etapa assegura que o código é bem formado (isto é, que ele não estoura a pilha ou contém bytecodes ilegais, por exemplo). Se esta etapa não existisse, código malicioso ou corrompido inadvertidamente poderia tomar vantagens do interpretador de Java para causar danos ao ambiente onde o sistema está sendo executado.

Outra camada de proteção é frequentemente chamada de modelo *sandbox* (caixa de areia). Código não confiável é colocado nesta caixa de areia, onde ele pode ser executado livremente, sem contudo poder causar prejuízos ao ambiente externo a esta caixa, isto é, o ambiente completo da plataforma Java. Quando um *applet,* ou qualquer outro código não confiável, está rodando nesta caixa, existem um número de restrições sobre o que ele pode fazer. A mais óbvia dessas restrições é que ele não pode ter acesso ao sistema de arquivos local do ambiente onde está executando. Existem várias outras restrições, tais

como, somente tipos bem restritos de operações na rede podem ser realizadas, não é possível criar novos processos ou acessar a fila de eventos do sistema.

Finalmente, a partir da versão 1.1 do JDK, pode-se associar uma assinatura digital criptografada ao código Java de modo que se possa definir que o código provindo de uma pessoa ou organização seja considerado confiável mesmo quando carregado via rede.

Infelizmente, segurança de sistemas não é algo totalmente determinístico. Da mesma maneira que não se pode garantir que um programa seja 100% correto, nenhuma linguagem ou ambiente pode ser garantidamente 100% seguro. Tendo-se dito isto, Java parece oferecer um nível prático de segurança para a maioria das aplicações.

- **Desempenho**

Java foi projetada para ser compacta, independente de plataforma e para utilização em rede. Isto levou a decisão de utilizar o esquema de interpretação de *bytecode*s. Como uma linguagem interpretada, seu desempenho é razoável. Contudo, ela nunca será tão rápida como uma linguagem compilada como C. Estima-se que programas Java são cerca de 20 vezes mais lentos que programas equivalentes escritos em C. Antes que você fique desanimado, é importante saber que esta velocidade é mais do que suficiente para a maioria das aplicações, em particular, daquelas que fazem uso de interfaces gráficas interativas ou de acessos através da rede. Nestes tipos de aplicações, existe uma ociosidade de processamento causada pela espera da ação do usuário ou de um dado obtido através da rede.

Além disso, interpretadores de Java podem incluir compiladores de bytecodes, chamados de *just in time* (JIT), que podem traduzir os *bytecodes* para código executável nativo da plataforma durante a execução do programa. Neste caso ocorre uma melhora significativa do desempenho de programas Java. O formato de bytecodes de Java foi especialmente projetado para que o processo de geração de código de máquina seja bastante eficiente e otimizado. Ainda, se estivermos dispostos a sacrificar a portabilidade do código para ganhar rapidez, pode-se escrever partes do programa usando C, C++ ou outra linguagem e usar os métodos nativos de java para interagir com estes códigos.

- **Concorrência**

Java oferece recursos para o desenvolvimento de aplicações capazes de executar múltiplas rotinas concorrentemente assim como dispõe de elementos para a sincronização destas várias rotinas. Cada um destes fluxos de execução é o que se denomina um *thread*, um importante recurso de programação de aplicações mais sofisticadas. Por exemplo, aplicações gráficas interativas normalmente demandam que várias ações sejam realizadas simultaneamente. Cada uma dessas ações pode ser relacionada a um ou mais *threads*.

1.3.3 – Recursos Necessários e Disponíveis

Para trabalharmos com o ambiente Java é necessário um ambiente de desenvolvimento. Recomendamos o uso do *Java Developer's Kit* em versão superior ao JDK 1.2 e um navegador compatível com o Java tais como o *Netscape Communicator* 4.5 ou o

Microsoft Internet Explorer 4 ou versões superiores. O JDK, em diversas versões e plataformas oficialmente suportadas pela *Su*n, pode ser obtido gratuitamente no *sit*e:

http://www.javasoft.com/j2se/

1.3.3.1 Instalação do JDK

No *site* mencionado acima são encontradas as instruções atualizadas de como fazer a instalação da versão corrente do JDK da *Sun*. A seguir, apresentaremos os passos de instalação do JDK 1.3.1 para a plataforma Windows e para a plataforma Linux. Recomendamos, contudo, que esses passos sejam verificados no site da *Sun*, antes de uma instalação, para que se verifique se continuam atualizadas.

1. Na plataforma Windows

- Requisitos do Sistema:
  - Pentium 166 MHz ou superior (recomenda-se 300 MHz ou superior)
  - Mínimo de 32 Mbytes de RAM (recomenda-se 64 Mbytes ou mais)
  - Mínimo de 70 Mbytes de espaço livre em disco para SDK
  - 120 Mbytes de espaço livre adicional para instalação da documentação (o que é fortemente recomendado)

- Instruções de Instalação
  1. Faça o *download* do arquivo instalador `j2sdk-1_3_1-win.exe` no site http://www.javasoft.com/j2se/1.3/download-windows.html .
  2. Verifique se o o download foi bem sucedido checando o tamanho do arquivo (34440521 bytes).
  3. Execute o instalador.
  4. Se a instalação for bem sucedida, desligue e religue o computador (dê um *reboot)*.
  5. Verifique se a estrutura de diretórios tem a seguinte forma:

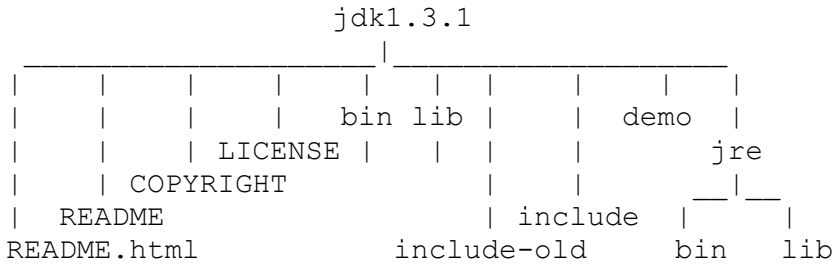

Figura 1. 8 - Árvore de Diretórios do JDK

  6. Você pode, agora, remover o arquivo instalador para recuperar o espaço livre em disco.
  7. Inclua a trilha completa do diretório jdk1.3.1\bin (tipicamente, c:\jdk1.3.1\bin) na variável PATH. Vc deve consultar as instruções específicas da versão do

Windows (95, 98, NT, etc.) utilizada sobre como adicionar trilhas a variável PATH.

8. Verifique a variável CLASSPATH. Caso ela tenha sido definida em uma instalação prévia da plataforma Java, vc precisa apenas garantir que o diretório corrente (".") faça parte dela. Se existir uma trilha para o arquivo classes.zip, você pode removê-la. Caso ela não tenha sido definida, você não deve fazer nada (continue deixando-a indefinida). Para verificar a variável CLASSPATH, use o comando set do DOS. Se a variável aparecer na lista, ela já foi definida. Neste caso, proceda como agiu no passo 7 para atualizá-la.
9. Utilize alguns exemplos simples para verificar se as ferramentas de desenvolvimento estão funcionando apropriadamente.
10. Caso tenha algum problema de instalação, consulte a seção de *troubleshooting* no site:
    http://java.sun.com/j2se/1.3/install-windows.html
11. Para desinstalar o SDK, use o utilitário de adição e remoção de programas do painel de controle. Para tanto, basta selecionar a opção Java 2 SDK.

2. Na plataforma Linux

- Requisitos do Sistema:
  - Plataforma Pentium rodando o Linux kernel v 2.2.12 e glibc v2.1.2-11 ou superior. Para verificar a sua versão da glibc, use o comando:
    ls /lib/libc-*
  - Mínimo de 32 Mbytes de RAM (recomenda-se 64 Mbytes ou mais)
  - Mínimo de 75 Mbytes de espaço livre em disco para SDK
  - 125 Mbytes de espaço livre adicional para instalação da documentação (o que é fortemente recomendado)
  - A plataforma Linux oficial do J2SDK 1.3.1 é o RedHat Linux 6.2, embora ele também já tenha sido testado de forma limitada em outras sistemas operacionais Linux.

- Instruções de Instalação

1. Faça o *download* do arquivo instalador j2sdk-1_3_1-linux-i386.bin no site
   http://www.javasoft.com/j2se/1.3/download-linux.html
2. Verifique se o o download foi bem sucedido checando o tamanho do arquivo (26924751 bytes).
3. Copie o arquivo para o diretório que você deseja instalar o Java 2 SDK.
4. Execute o j2sdk-1_3_1-linux-i386.bin

   De dentro do diretório onde o arquivo foi copiado, use os seguintes comandos

   ```
   chmod a+x j2sdk-1_3_1-linux-i386.bin
   ./j2sdk-1_3_1-linux-i386.bin
   ```

   Concorde com o acordo de licensa apresentada.
   O Java 2 SDK será instalado em um diretório chamado jdk1.3.1 dentro do diretório corrente.

Lembre-se que se você decidir instalar o SDK em uma locação visível por todos os usuários do sistema (tal como, /usr/local), você deve se tornar root primeiro para ter acesso as permissões necessárias. Se você não tem acesso a conta root, instale o SDK abaixo do seu diretório home.

5. Utilize alguns exemplos simples para verificar se as ferramentas de desenvolvimento estão funcionando apropriadamente.
6. Para maiores informações, consulte o site (lá se encontram também as instruções de como fazer a instalação do SDK através de arquivo rpm): http://java.sun.com/j2se/1.3/install-linux-sdk.html

1.3.3.2 Instalação da Documentação do Java 2 SDK

Como já mencionamos, é fortemente recomendado que se instale também a documentação do Java 2 SDK. Esta documentação é fundamental para que o programador Java possa consultar a API (*Aplication Programming Interface)* do Java. Ela pode ser obtida no site

http://java.sun.com/j2se/1.3/docs.html

- Instruções de Instalação

1. Faça o *download* do arquivo j2sdk-1_3_1-doc.zip a partir do site mencionado acima.
2. Verifique se o o download foi bem sucedido checando o tamanho do arquivo (22577838 bytes).
3. Realize a descompressão do arquivo (escolha o diretório desejado para guardar a documentação).
4. Verifique se a estrutura de diretórios tem a seguinte forma:

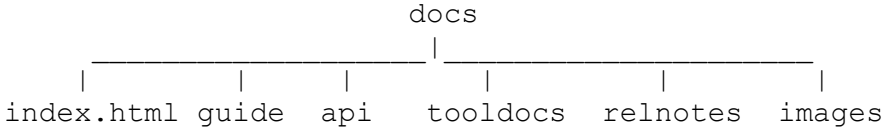

Figura 1. 9 - Árvore de Diretórios da Documentação

5. Consulte a documentação:

   Abra em um *browser* o arquivo index.html situado dentro do diretório docs.
   Clique no link Java 2 Platform API Specification.
   Veja as descrições dos pacotes Java do SDK. Selecione o pacote java.awt.
   Clique no link da classe Point.
   Observe a descrição da hierarquia de classes, dos campos e métodos desta classe.

6. Para maiores informações sobre a instalação, consulte o site:

   http://java.sun.com/j2se/1.3/install-docs.html

1.3.3.3 – Fontes de Consulta e Obtenção de Aplicações

Outras informações sobre a plataforma Java, artigos, dicas, exemplos, bibliotecas e aplicações podem ser obtidas nos sites:

http://www.javasoft.com/	Site oficial da *Sun* sobre o Java
http://www.javasoft.com/tutorial/	Tutoriais sobre o Java
http://java.sun.com/docs/books/tutorial/	Outros tutoriais sobre o Java
http://www.javaworld.com/	Revista *online*
http://www.javareport.com/	Revista *online*
http://www.jars.com/	*Applet*s, exemplos e outros recursos
http://www.gamelan.com/	Diversos recursos e exemplos Java
http://www.internet.com/	Diversos recursos e exemplos Java
http://www.javalobby.org/	Revista *online*
http://www.sys-con.com/java	Revista *online*
http://sunsite.unc.edu/javafaq/javafaq.html	Respostas de dúvidas comuns sobre Java
http://www.acm.org/crossroads/	Revista *online* da ACM
http://www.december.com/works/java.html	Diversos recursos e exemplos Java
http://www.javacoffeebreak.com	Diversos recursos e exemplos Java

1.3.4 – O Kit de Desenvolvimento de Java da *Sun*

O JDK é composto basicamente por:

- Um compilador (javac)
- Uma máquina virtual Java (java)
- Um visualizador de *applets* (appletviewer)
- Bibliotecas de desenvolvimento (os *packages* java)
- Um programa para composição de documentação (javadoc)
- Um depurador básico de programas (jdb)
- Versão *run-time* do ambiente de execução (jre)

Deve ser observado que o JDK não é um IDE (ambiente interativo de desenvolvimento), embora mesmo assim seja possível o desenvolvimento de aplicações gráficas complexas apenas com o uso do JDK. De fato, este é o padrão em termos da tecnologia Java. Outros fabricantes de software tais como *Microsof*t, *Borlan*d, *Symantec* e *IBM* oferecem comercialmente ambientes visuais de desenvolvimento Java, respectivamente, *Visual J++*, *JBuilde*r, *VisualCafe* e *VisualAge for Jav*a. A *Sun* oferece gratuitamente um ambiente interativo de desenvolvimento chamado *Forte for Java*.

1.3.4.1 – Usando as Ferramentas Básicas do JDK

Com o JDK adequadamente instalado em seu computador, apresentaremos um pequeno exemplo de aplicação para ilustrarmos a utilização básica das ferramentas do JDK. Observe o exemplo 1.1 a seguir, sem necessidade de compreendermos em detalhe o que cada uma de suas partes representa:

```
//Eco.java
import java.io.*;
public class Eco {
  public static void main(String[] args) {
  for (int i=0; i < args.length; i++)
  System.out.print(args[i] + " ");
  System.out.println();
  }
}
```

Exemplo 1. 1 - Primeiro Exemplo de Aplicação Java

Utilize um editor de textos qualquer para digitar o exemplo, garantindo que o arquivo será salvo em formato de texto simples, sem qualquer formatação e que o nome do arquivo será Eco.java (utilize letras maiúsculas e minúsculas exatamente como indicado!).

Para podermos executar a aplicação exemplificada devemos primeiramente compilar o programa. Para tanto, devemos utilizar uma janela de comandos do DOS ou do Linux. Devemos acionar, então, o compilador Java, ou seja, o programa javac, como indicado abaixo:

```
javac nome_do_arquivo.java
```

O compilador Java exige que os arquivos de programa tenham a extensão ".java" e que o arquivo esteja presente no diretório corrente, assim, numa linha de comando, dentro do diretório onde foi salvo o arquivo Eco.java digite:

```
javac Eco.java
```

Isto aciona o compilador Java transformando o código digitado no seu correspondente em *bytecode*s, produzindo um arquivo Eco.class. Não existindo erros, o javac não exibe qualquer mensagem, caso contrário serão exibidas na tela uma mensagem para cada erro encontrado, indicando em que linha e que posição desta foi detectado o erro.

Tendo compilado o programa, ou seja, dispondo-se de um arquivo ".class" podemos invocar uma máquina virtual Java (JVM), isto é, o interpretador Java, da seguinte forma:

```
java nome_do_arquivo
```

Note que não é necessário especificarmos a extensão ".class" para acionarmos a JVM. Digitando-se o comando abaixo executamos o programa Eco:

```
java Eco
```

Aparentemente nada aconteceu, mas se a linha digitada fosse outra contendo argumentos diversos tal como:

```
java Eco Testando 1 2 3
```

Obteríamos o seguinte resultado:

```
Testando 1 2 3
```

Na verdade o programa exemplo apresentado apenas imprime todos os argumentos fornecidos como um eco. Embora de utilidade duvidosa, esta aplicação nos mostra como proceder a compilação e execução de programas escritos em Java utilizando o JDK.

1.3.5 – A Arquitetura da Plataforma Java

O exemplo dado na seção anterior, além de ilustrar aspectos relacionados a programação de aplicações Java, serve também para caracterizar uma parte do ambiente de desenvolvimento da linguagem Java: a criação de programas, sua compilação e execução, como mostra a Figura 1.10.

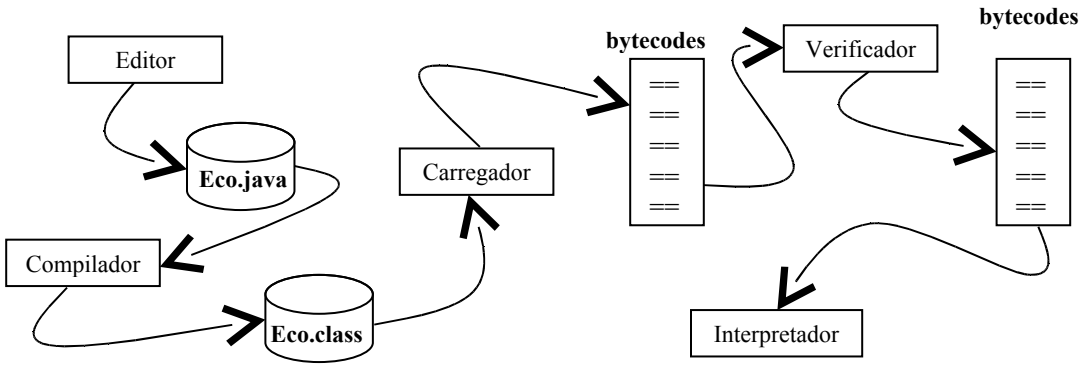

Figura 1. 10 - Ambiente de Desenvolvimento Java

Como visto, devemos utilizar um editor de textos ASCII simples para produzirmos o código fonte de um programa que, obrigatoriamente deve ser salvo como arquivos de extensão ".java". No nosso exemplo, o arquivo gerado se chama "Eco.java". Utilizando-se o compilador javac o arquivo fonte é transformado em *bytecodes* Java e automaticamente salvo em um arquivo ".class" (no exemplo, "Eco.class"). O arquivo ".class", composto por bytecodes, pode ser executados em qualquer plataforma computacional que disponha de uma máquina virtual Java como ilustrado na Figura 1.11.

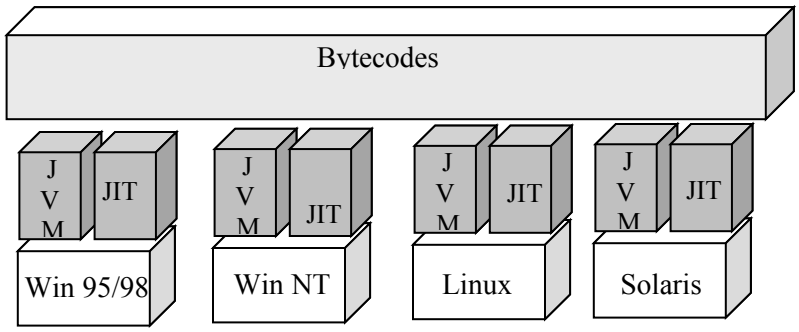

Figura 1. 11 - Ambiente de Execução Java

Quando solicitamos a execução do programa, tanto através do interpretador java como de sua versão *run-time* jre, o arquivo ".class" é primeiramente carregado na memória principal. Em seguida, o arquivo é verificado para se garantir os requisitos de segurança do sistema e só então é propriamente interpretado. No caso específico das *applets* o navegador *(browse*r) efetua o download de um arquivo ".class" interpretando-o ou

acionando algum interpretador associado *(plug-in*). Em ambientes onde existam compiladores JIT *(just in time*) os *bytecodes* já verificados são convertidos em instruções nativas do ambiente durante a execução, aumentando a performance das aplicações ou *applets* Java.

O resultado deste ambiente é que o Java, embora interpretado inicialmente, torna-se independente de plataforma, simplificando o projeto de aplicações de rede ou distribuídas que tenham que operar em ambientes heterogêneos, além de permitir a incorporação de vários mecanismos de segurança.

Cabe dizer ainda que já existem compiladores Java para gerar diretamente código nativo a partir do código fonte. Enquanto a eficiência é melhorada (uma vez que a compilação é feita antes da solicitação de execução), sacrifica-se a portabilidade do código compilado. De qualquer forma, o código fonte continua sendo portável, bastando recompilá-lo em cada plataforma específica.

1.3.6 – Programas em Java

Um programa em Java é composto por classes. Em capítulos subsequentes, apresentaremos o conceito de classe em profundidade. Por enquanto, basta saber que uma classe define uma seção de um programa. Programas pequenos normalmente são constituídos por apenas uma classe. O exemplo 1.2 mostra a implementação de um programa bastante simples.

```
class PrimeiroPrograma {
  public static void main ( String[] args )  {
    System.out.println("Isto é um programa muito
pequeno!");
  }
}
```

Exemplo 1. 2 - Um Programa Muito Simples

Como mencionado anteriormente, este programa será escrito em um arquivo texto com extensão ".java". Embora não seja o caso do exemplo acima, normalmente, Java requer que o nome do arquivo coincida com o nome da classe nele definida (deve ser escrito exatamente da mesma maneira). Por uma questão de bom hábito vamos utilizar esta mesma regra para a definição do nome do nosso arquivo. Então, o arquivo do exemplo deve se chamar "PrimeiroPrograma.java". Após sua compilação, portanto, será gerado um arquivo de bytecodes chamado "PrimeiroPrograma.class". A execução deste arquivo apresentará a mensagem *Isto é um programa muito pequeno!* na saída padrão.

Classes normalmente conterão várias sentenças da linguagem Java. Tudo que faz parte de uma classe deve ser colocado entre um abre e fecha chaves.

Existem dois tipos de programas em Java: Aplicações e *Applets*. Enquanto *Applets* necessitam de um *browser* para serem executados, aplicações podem ser executadas independentemente. Para que uma classe possa ser executada como aplicação, ela necessita implementar um subprograma chamado *main*. Este nome é derivado da função principal de C e C++ que indica onde deve se iniciar a execução de um programa. Este

mesmo papel é realizado pela função *main* da classe Java. Em outras palavras, a linha de código `public static void main ( String[] args )` indica onde o programa Java começará a ser executado. O exemplo 1.2 é, portanto, uma aplicação.

A título de curiosidade, apresentamos a seguir um exemplo de classe que implementa um *applet*. Embora possam existir *applets* que também sejam aplicações, a classe do exemplo 1.3 não é uma aplicação, pois não possui o subprograma *main*.

```
import java.applet.*;
import java.awt.*;

public class Rabisco extends Applet {
  private int last_x = 0, last_y = 0;
  public boolean mouseDown (Event e, int x, int y) {
     last_x = x;
     last_y = y;
     return true;
  }
  public boolean mouseDrag (Event e, int x, int y) {
     Graphics g = getGraphics();
     g.drawLine(last_x, last_y, x, y);
     last_x = x;
     last_y = y;
     return true;
  }
}
```

Exemplo 1. 3 - Exemplo de Applet Java

Para que se possa observar o que este *applet* realiza, é necessário criar um arquivo html com uma tag chamada APPLET, e chamar este arquivo html em um *browser* que tenha associada uma JVM. O exemplo 1.4 apresenta um arquivo html que pode ser usado. Para a correta execução, é importante que este arquivo se encontre no mesmo diretório do arquivo "Rabisco.class".

```
<HTML>
<HEAD>
<TITLE> O Applet Rabisco </TITLE>
</HEAD>
<BODY>
Por favor, faça rabiscos com o mouse na tela abaixo.
<P>
<APPLET code="Rabisco.class" width=500 height=300>
Sinto muito! Java não é suportado ou está
desabilitado no seu browser.
</APPLET>
</BODY>
</HTML>
```

Exemplo 1. 4 - Arquivo HTML com Applet

1.4 – Exercícios

1. Responda as seguintes perguntas:
   1.1. Quando estão em execução, programas em linguagem de máquina estão na memória principal juntamente com os dados?
   1.2. Geralmente, o custo envolvido com programação em uma empresa é alto. O que seria melhor para a empresa: os programadores utilizarem linguagem de máquina ou alguma linguagem de alto nível?
   1.3. Digamos que um programa fonte tenha sido traduzido para um executável. Após executá-lo algums vezes, o programador decide fazer modificações no programa. Onde deve ser feita a mudança? No fonte ou no executável?
   1.4. Considere que um programa fonte tenha sido escrito em C. O arquivo com o fonte foi copiado para um computador PC e para um Macintosh. O que necessita ser feito para que o programa execute nos dois computadores? É necessário alterar o programa fonte?
   1.5. Suponha que uma empresa necessita de programas que executem tanto em computadores PC quanto Macintosh. Seria melhor para esta empresas que os programas fossem feitos em Java ou em outra linguagem de alto nível?
   1.6. É sempre importante executar programas o mais rápido possível?
   1.7. É possível escrever tanto interpretadores quanto compiladores para traduzir programas de uma linguagem de alto nível qualquer?
   1.8. Qual o papel dos arquivos bytecode de Java?
2. Instale em sua máquina o JDK e a documentação de Java, seguindo os passos especificados neste capítulo.
3. Consulte a documentação da classe BigInteger de Java.
4. Utilize os exemplos 1.1 e 1.2 de programas Java deste capítulo para criar programas executáveis em Java.
5. Altere o programa PrimeiroPrograma para que imprima uma outra mensagem qualquer.