Programação em JAVA

Daniela Barreiro Claro
João Bosco Mangueira Sobral

Prefácio

Este livro foi originado a partir de nossas experiências com o ensino da linguagem Java em laboratório nos Cursos de Extensão no Departamento de Informática e Estatística, Centro Tecnológico, Universidade Federal de Santa Catarina.

Desde sua primeira versão no primeiro semestre de 2000, algumas revisões foram realizadas e aplicadas nos cursos dos anos subseqüentes.

A partir de 1996, o intercâmbio de conhecimento utilizando a linguagem Java veio se consolidando no meio acadêmico e empresarial. A introdução da linguagem Java no Departamento de Informática e Estatística, mais especificamente no Curso de Pós-Graduação em Ciência da Computação, ocorreu no ano de 1999. A partir desta época, a linguagem Java foi sendo consolidada nos cursos de computação da UFSC, tanto graduação quanto pós-graduação, sendo utilizada em conjunto com outros paradigmas do meio computacional. Atualmente, a linguagem vem sendo bastante utilizada em implementações nos trabalhos de conclusão de curso de graduação, nas dissertações de mestrado e teses de doutorado deste Departamento.

No meio empresarial e corporativo, a cultura do Java vem se expandindo de uma forma gradual e constante, permitindo que diversas empresas desenvolvam aplicações utilizando esta linguagem.

Na literatura, já existem muitos livros explicando e utilizando a programação em Java, nos aspectos gerais, tanto em livros específicos, abordando APIs mais avançadas. Atualmente, em âmbito nacional, existem publicações periódicas referenciando detalhes da programação em Java.

Este livro aborda a linguagem Java de uma forma mais objetiva, considerando o histórico da linguagem, as características principais, os seus fundamentos, a orientação a objetos em Java, programação de interfaces gráficas, manipulação de programas para Web(Applets), simulação de processamento concorrente(Threads), programação em redes com Sockets e acesso a banco de dados utilizando o JDBC.

João Bosco M. Sobral
Daniela Barreiro Claro

Agradecimentos

Gostariamos também de agradecer aos autores de livros da área de Informática, especialmente aos da Linguagem Java que passaram o seu tempo tentando escrever nas mais diversas linhas os conceitos e definições para que nós, simples aprendizes, pudéssemos compreender e enfim utilizar a linguagem. Pontualmente, eu gostaria de agradecer aos autores do Livro Core Java 2 - Fundamentals, Sr. Horstmann e Sr. Cornell, pois considero este livro um dos melhores e mais bem explicados em relação a linguagem Java, assim também o recomendo a todos os interessados em se aprofundar nos conceitos da linguagem, e aproveito para mencionar que muitos dos exemplos e conceitos aqui escritos foram apreendidos do conteudo do citado livro.

Nós gostaríamos de agradecer aos alunos de graduação em computação da UFSC que ministraram a primeira experiência, a Daniela Vanassi de Oliveira (PPGCC-UFSC) pela contribuição inicial neste material e a Euclides de Moraes Barros Junior (PPGCC-UFSC), pelas participações nos cursos de extensão, quando continuamos nossas experiências de ensino com a linguagem Java.

Também agradecemos ao Departamento de Informática e Estatística de UFSC pela oportunidade de realizar os cursos de extensão. À FEESC (Fundação de Ensino de Engenharia em Santa Catarina) pelo apoio na parte financeira e certificados.

E àqueles que sempre nos apoiaram em todos os momentos, em especial, Olga Amparo Garcia de Barreiro (*in memorium*), Olga Maria Barreiro Claro e Antonio Carlos Claro Fernandez.

Também agradecemos aos Grupos de Usuários Java que hoje são numerosos em todo o país e que permitem uma troca de informações entre a comunidade Java.

Nossos sinceros agradecimentos.

Claro D. B. e Sobral J. B. M. PROGRAMAÇÃO EM JAVA. Copyleft Pearson Education. Florianópolis, SC.

Índice

1. Orientação a Objetos

Inovações tecnológicas surgidas na área de Informática têm criado uma necessidade de utilização e manipulação de informações que antigamente não eram utilizadas. Os tipos de dados complexos, como os objetos, passaram a ser manipulados através das linguagens de programação, que passaram a receber a conotação de Linguagem de Programação Orientada a Objetos.

A programação estruturada, em se tratando, principalmente, de manutenção de sistemas, possui taxas de reutilização muito baixas, dificultando a manutenção dos programas anteriormente desenvolvidos.

A orientação a objetos tem como objetivo principal modelar o mundo real, e garantir que as taxas de *manutenibilidade* (manutenção) serão maiores diante deste contexto. Isso é possível, pois utilizando uma linguagem de programação orientada a objetos consegue-se obter um desenvolvimento mais rápido, visto que este desenvolvimento ocorre em módulos, em blocos de códigos correspondentes aos objetos e seus acoplamentos. Através da orientação a objetos pode-se obter uma maior qualidade e agilidade no desenvolvimento, pois o fator *reusabilidade* (reutilização) permite que se re-utilize outros objetos que foram anteriormente desenvolvidos e podem ser facilmente incorporados na aplicação. A reusabilidade também garante uma manuseabilidade melhor do programa, pois os testes referentes aos componentes, já foram previamente executados, garantindo assim a utilização coesa dos objetos.

A orientação a objetos surgiu na década de 60, baseada na Teoria dos Tipos da Álgebra, mas somente na década de 90 começou a ser amplamente utilizada computacionalmente. Ela tem como principio fundamental representar o mundo real e a forma de se interagir com os objetos. Mas, o que é um objeto?

Atualmente, há milhares de milhares de objetos que se pode tocar e perceber. Agora mesmo, você está tocando em um objeto denominado livro e posso garantir que deste livro que você está nas mãos, somente há um exemplar. Há um modelo deste livro, que é responsável pela criação do mesmo em série, mas este livro, que está nas suas mãos, idêntico, que possui um grifo de lápis (caso você já tenha dado um) somente há um, este nas suas mãos. É esta a característica principal dos objetos, eles são únicos e somente há um único objeto no mundo.

Vamos pensar no mundo dos automóveis. Pensemos em uma montadora de veículo, onde há uma fábrica de automóveis. Cada automóvel que sai desta fábrica é único, é identificado por um número de chassi. O meu carro é único, é o único que possui um arranhão na parte traseira esquerda ... Por mais que tenha diversos carros iguais ao meu, com a mesma cor, da mesma marca, mesma categoria, o meu carro é único, por ter características que somente pertencem a ele. E assim são os objetos.

E as classes ? As classes são como se fossem as montadoras, é quem detém a “receita” de como montar, quem contém o molde dos carros, é uma abstração do carro. Na verdade a classe contém os moldes do carro, e não o carro propriamente dito. No caso de oreintação a objetos, as classes são as monytadoras dos objetos. As classes são vistas como agrupamento de objetos que contém as mesmas características e os mesmos comportamentos.

Os objetos ou as classes possuem características que determinam quem eles realmente são, pensemos em um bolo de chocolate, onde a receita possui características que determinam a receita como os ingredientes. Abstraímos um pouco este bolo, e imaginemos a receita como sendo a Classe, e o bolo pronto como sendo o objeto. Os ingredientes são parte da receita, assim são características de uma receita e poderemos denominar eles de atributos da Classe Receita. E o modo de preparo do bolo? O modo de preparo do bolo é método que vamos utilizar para elaborar o bolo. Assim, o modo de preparo é o método. E o bolo pronto, assado, cheirando, como já dissemos anteriormente ele é o objeto, e único, pois se você for fazer este mesmo bolo e seguir a mesma receita, provavelmente o seu bolo será melhor que o meu, pois ficará mais solto na assadeira, tem uma cor melhor, mais amarela, gosto saboroso. E estas características do bolo pertencem somente ao meu ou ao seu bolo, assim são atributos do objeto, também denominado de atributos de instância (CAMERA, 2002).

Assim, através destes exemplos podemos observar os fundamentos da Orientação a Objetos. Cada objeto também pode se comunicar entre si, através da troca de mensagem. A troca de mensagens entre objetos pode ser abstraída como a aplicação de métodos em determinados objetos. Por exemplo, quando um objeto A deseja que o objeto B execute um dos seus métodos, o objeto A envia uma mensagem ao objeto B, como segue na Figura 1.

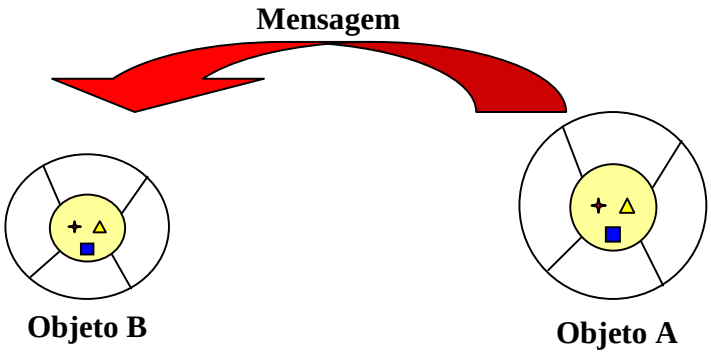


Figura 1. Troca de mensagens entre os objetos.

"Mensagens e métodos[operações] são os dois lados da mesma moeda. Métodos são os procedimentos que são invocados quando um objeto recebe uma mensagem " Greg Voss

Outro conceito muito importante da Orientação a Objetos é o encapsulamento das informações. Através do encapsulamento se pode ocultar informações irrelevantes para os outros objetos que interagem com estes objetos. O encapsulamento permite que os atributos de uma determinada classe somente sejam modificados utilizando métodos que interajam com o mesmo. Assim, as modificações através dos métodos garantem que não há manipulações indevidas aos atributos. Sempre que for possível garantir que os atributos sejam encapsulados e a manipulação deles ocorra através de métodos, o faça.

A herança é mais uma característica do mundo real incorporado no mundo orientado a objetos. É um mecanismo que define variáveis e métodos comuns a todos os objetos de um certo tipo. No mundo real há os nossos pais, nossos avós e nossos filhos e netos. Onde alguma pessoa sempre lhe diz que você se parece mais com o seu pai ou com a sua mãe, e seus filhos e netos sempre herdam algum comportamento do pai ou da mãe ou ainda dos avós paternos ou maternos...enfim, a herança é "uma transmissão dos caracteres físicos ou morais aos descendentes" (FERREIRA, 1993)

Assim, a partir de uma determinada classe, pode-se ter subclasses e superclasses. As subclasses não são limitadas aos estados e comportamentos definidos pela superclasse,

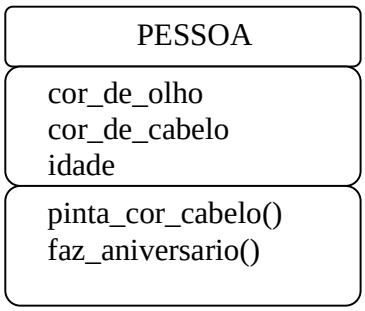
elas podem adicionar variáveis e métodos ao seu escopo. Entre estas características da herança, a manipulação das classes que herdam métodos e atributos permite que se modularize o código de tal forma que cada objeto tem uma funcionalidade própria e garante assim uma coesão das informações.

Outro conceito adicional da orientação a objetos utilizado em Java é o polimorfismo, onde pode-se definir duas abordagens como: sobreposição(override) e sobrecarga(overload). A sobreposição ocorre quando duas ou mais classes derivadas de uma mesma superclasse podem invocar métodos que possuem a mesma identificação(assinatura), mas comportamentos distintos. Enquanto que a sobrecarga dos métodos ocorre quando em uma mesma classe, métodos possuem os mesmos nomes, mas assinaturas diferentes, um exemplo disso são os métodos construtores do Java, que serão abordados mais adiante neste livro.

As interfaces, outro conceito importante e bastante utilizado nas implementações das Interfaces Gráficas no Java, "são invólucros que promovem a interação de contatos externos, com ou sem passagem de dados, com um processamento interno." (CAMARA, 2002)

Um usuário somente precisa saber o que o objeto é capaz de fazer, mas não interessa como ele deverá ser feito. Assim, a interface permite que se utilize o conceito do encapsulamento, onde somente as assinaturas dos métodos estão disponíveis para exibição aos usuários.

Podemos criar uma estrutura mais próxima da manipulação de classes e objetos dentro da linguagem Java, como podemos ter uma classe PESSOA que possui atributos (características) tais como: cor_de_olho, cor_de_cabelo, idade. Esta classe PESSOA também possui métodos (ações, comportamentos) tais como: faz_aniversario, pinta_cor_de_cabelo.



Instanciando esta classe PESSOA, obtemos o que chamamos de objeto, ou seja:

```
daniela   = new  PESSOA ( );
bosco     = new  PESSOA ( );
```

Cada objeto tem suas características (atributos) particulares, por exemplo:

```
daniela.cor_de_olho   = 'verde';
daniela.cor_de_cabelo= 'castanho escuro';
daniela.idade         = 27;
```

Enquanto que o objeto *antonio* possui outros valores para seus atributos.

```
bosco.cor_de_olho          = 'castanho';
```

```
bosco.cor_de_cabelo        = ‘preto;
bosco.idade                = 55;
```

Em relação aos métodos, estes são utilizados para modificar/manipular os valores dos atributos, segue um exemplo:

```
daniela.faz_aniversario ( );
```

O método faz_aniversario implementa um procedimento onde irá somar mais um no atributo idade de daniela. Exemplo:

```
faz_aniversario( ) {
        daniela.idade = daniela.idade+1;
}
```

```
daniela.pinta_cor_cabelo(‘preto’);
```

Este segundo método age sobre o objeto daniela e realiza a ação de pintar a cor do cabelo passando como parâmetro a cor desejada.

Assim, através de classes, objetos, atributos, métodos, dentre outras características da orientação a objetos, consegue-se modelar o mundo real e abstrair informações incorporando-as à linguagem Java.

2. Introdução ao JAVA

Java é a linguagem de programação orientada a objetos, desenvolvida pela Sun Microsystems, capaz de criar tanto aplicativos para desktop, aplicações comerciais, softwares robustos, completos e independentes, aplicativos para a Web. Alem disso, caracteriza-se por ser muito parecida com C++, eliminando as características consideradas complexas, dentre as quais ponteiros e herança múltipla.

2.1 Histórico

Em 1991, um pequeno grupo de funcionários da Sun incluindo James Gosling mudou-se para a San Hill Road, uma empresa filial. O grupo estava iniciando um projeto denominado Projeto Green, que consistia na criação de tecnologias modernas de software para empresas eletrônicas de consumo, como dispositivos de controle remoto das TV a cabo. Logo o grupo percebeu que não poderia ficar preso as plataformas, pois os clientes não estavam interessados no tipo de processador que estavam utilizando e fazer uma versão do projeto para cada tipo de sistema seria inviável. Desenvolveram então o sistema operacional GreenOS, com a linguagem de programação Oak. Eles se basearam no inventor do Pascal, através da linguagem USCD Pascal, que foi o pioneiro da linguagem intermediária ou máquina virtual.

Em 1993, surgiu uma oportunidade para o grupo Green, agora incorporado como FirstPerson a Time-Warner, uma empresa que estava solicitando propostas de sistemas operacionais de decodificadores e tecnologias de vídeo sob demanda. Isso foi na mesma época em que o NCSA lançou o MOSAIC 1.0, o primeiro navegador gráfico para Web. A FirstPerson apostou nos testes de TV da Time-Warner, mas esta empresa preferiu optar pela tecnologia oferecida pela Silicon Graphics.

Depois de mais um fracasso, a FirstPerson dissolveu-se e metade do pessoal foi trabalhar para a Sun Interactive com servidores digitais de vídeo. Entretanto, a equipe restante continuou os trabalhos do projeto na Sun. Apostando na Web, visto que os projetos estavam sendo todos voltados para a WWW, surgiu a idéia de criar um browser com independência de plataforma, que foi o HotJava.

Como a equipe de desenvolvimento ingeria muito café enquanto estavam trabalhando, várias xícaras de café foram inseridas até que o projeto estivesse pronto. Finalmente em maio de 1995, a Sun anunciou um ambiente denominado Java (homenagem às xícaras de café) que obteve sucesso graças a incorporação deste ambiente aos navegadores (browsers) populares como o Netscape Navigator e padrões tridimensionais como o VRML (Virtual Reality Modeling Language – Linguagem de Modelagem para Realidade VIrtual).

A Sun considera o sucesso do Java na Internet como sendo o primeiro passo para utilizá-lo em decodificadores da televisão interativa em dispositivos portáteis e outros produtos eletrônicos de consumo – exatamente como o Java tinha começado em 1991. Sua natureza portátil e o projeto robusto permitem o desenvolvimento para múltiplas plataformas, em ambientes tão exigentes como os da eletrônica de consumo.

A primeira versão da linguagem Java foi lançada em 1996.

2.2 Web x Aplicativos

Programas escritos em Java, podem ser Applets, Aplicativos ou ainda Servlets. Os aplicativos são programas que necessitam de um interpretador instalado na máquina. Enquanto que Applets são programas carregados juntamente com páginas HTML. O interpretador, no caso das Applets, é o próprio browser. Não necessita instalação, basta que o browser usado ofereça suporte a Java. Já no caso dos Servlets, são programas desenvolvidos em Java que são interpretados pelo Servidor Web. Os servlets são utilizados na geração dinâmica de páginas HTML. Atualmente, são muito utilizados na combinação com JSP (Java Server Pages) para a utilização do MVC (Model View Controller).

2.3 Java Developement Kit - JDK

O JDK é um kit de desenvolvimento Java fornecido livremente pela Sun. Constitui de um conjunto de programas que engloba compilador, interpretador e utilitários. A primeira versão deste Kit foi a 1.0. Atualmente, o JDK está na versão 1.4.x, tendo sido lançado recentemente a versão beta 1.5.x, também denominada Tiger. O JDK é separado em 3 edições: o Java 2 Standard Edition (J2SDK), o Java 2 Enterpise Edition (J2EE) e o Java 2 Micro Edition (J2ME). Cada uma engloba um conjunto de pacotes diferentes fornecendo aos usuarios uma forma organizada e diferenciada para desenvolver aplicações. Ou seja, os usuários que desejem desenvolver aplicações para Palm Tops, celulares, dispositivos pequenos, deve utilizar o J2ME para desenvolver as suas aplicações.

Os principais componentes do kit de desenvolvimento são:

- ✓ javac (compilador)
- ✓ java (interpretador)
- ✓ appletviewer (visualizador de applets)
- ✓ javadoc (gerador de documentação)
- ✓ jar (programa de compactação)

A utilização do JDK é feita da seguinte forma: primeiro escreve-se o programa fonte em Java, utilizando qualquer editor de texto ou IDE para Java como Eclipse, JCreator, JBuilder, JDeveloper, Bloco de Notas, TextPad(com pluggins), dentre outros. A seguir, o programa deve ser compilado utilizando o compilador **javac**:

javac <nomedoarquivo.java>

Exemplo: javac Teste.java

A compilação gera o arquivo em código binário (*bytecode*), com extensão *.class*.

Uma vez compilado, basta executar o programa. Se for uma aplicação, utilizar o interpretador **java**:

java <nomedaclasse>

Exemplo: java Teste

As IDEs para Java já tem o compilador (*javac*) e o interpretador (*java*) embutido no aplicativo, o que basta clicar em botões ou usar teclas de atalho para compilar e interpretar os programas desenvolvidos.

2.4 Manipulação do JDK(Java Developement Kit)

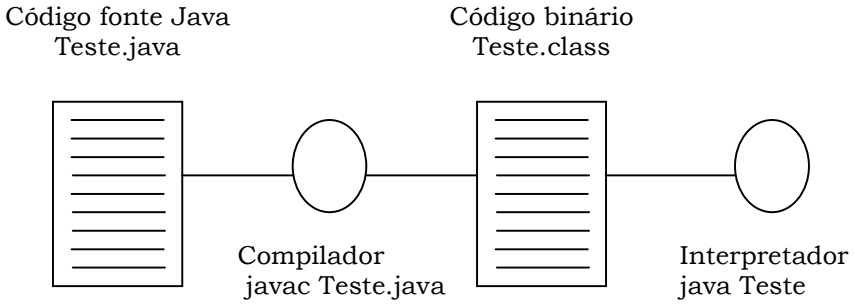


Figura 2 – Compilação e Interpretação no Java

Sendo um Applet, deve-se construir uma página HTML para abrigar o applet e carregá-la através de um browser ou do **appletviewer** (visualidador de applets Java):

appletviewer <nomedoarquivo.html>

Exemplo: appletviewer Teste.html

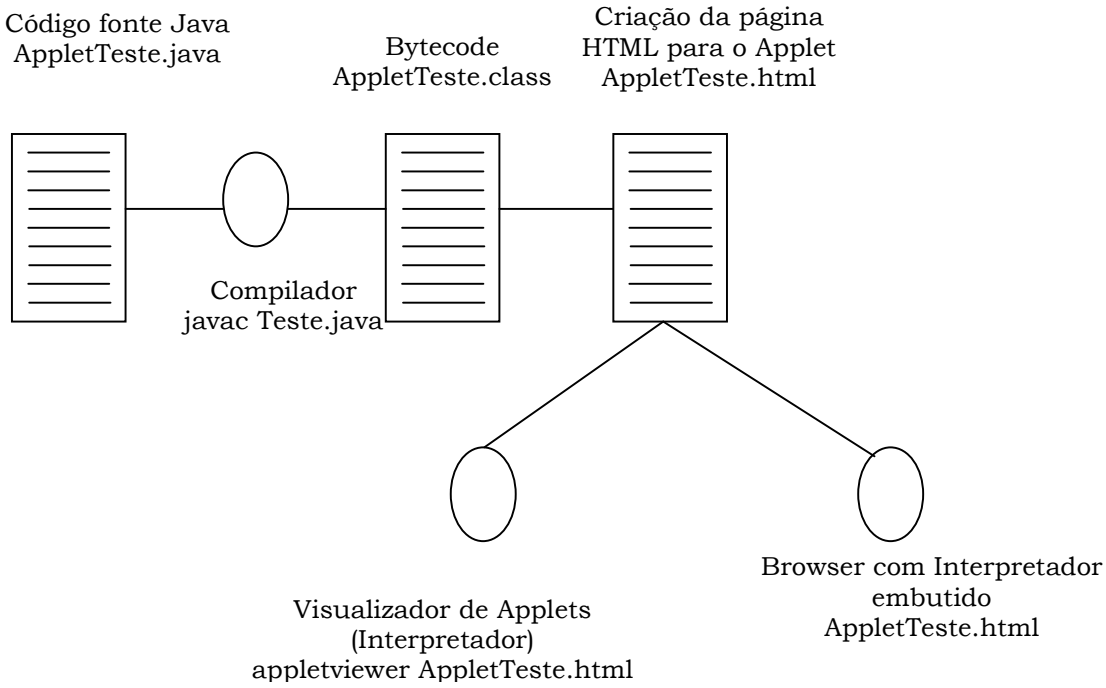


Figura 3 – Compilação e Interpretação da Applet

2.5 Características da Linguagem

Simples e familiar

Linguagem simples e de fácil manipulação, possui sintaxe muito parecida com C++ que é uma das mais conhecidas no meio. Java é muitas vezes considerada uma versão simplificada da linguagem C++, onde Java não possui características como arquivos *headers*, ponteiros, sobrecarga de operadores, classes básicas virtuais, dentre outras que somente aumentavam a dificuldade dos programadores com a linguagem C++.

Orientada a Objetos

Paradigma atualmente mais utilizado na construção de softwares. Permite que se focalize o dado, enfim, o objeto. Java não é uma linguagem 100% orientada a objetos, como Smaltalk, onde qualquer elemento, (operadores, sinais, tipos de dados,...) são objetos. Em Java há os tipos primitivos de dados que não são objetos, mas foram criados e incorporados ao Java para permitir uma melhor forma de utilização da linguagem pelos programadores. Outra característica importante da linguagem Java em relação à linguagem C++, é que Java não suporta herança múltipla.

Compilada e Interpretada

Um programa desenvolvido em Java necessita ser compilado, gerando um *bytecode*. Para executá-lo é necessário então, que um interpretador leia o código binário, o *bytecode* e repasse as instruções ao processador da máquina específica. Esse interpretador é conhecido como JVM (Java Virtual Machine). Os *bytecodes* são conjuntos de instruções, parecidas com código de máquina. É um formato próprio do Java para a representação das instruções no código compilado.

Pronta para Redes

As funcionalidades que são fornecidas pela linguagem Java para desenvolver programas que manipulem as redes através das APIs são simples e de grande potencialidades. Através destas APIs pode-se manipular protocolos como TCP/IP, HTTP, FTP e utilizar objetos da grande rede via URLs.

Distribuído

Programas Java são "linkados" em tempo de execução. Os *bytecodes* gerados durante a compilação só serão integrados na execução. Um objeto X existente em um arquivo quando instanciado, somente será alocado na memória em tempo de execução. Se alguma alteração ocorrer na classe que define o objeto X, somente o arquivo da classe com a alteração necessita ser compilado.

Multiprocessamento (Multithread)

Suporta a utilização de *threads*. *Threads* são linhas de execução, executadas concorrentemente dentro de um mesmo processo. Diferentemente de outras linguagens, programar utilizando *Threads* é simples e fácil na linguagem Java.

Portabilidade

Pode ser executado em qualquer arquitetura de hardware e sistema operacional, sem precisar ser re-compilado. Um programa Java pode ser executado em qualquer plataforma que possua um interpretador Java (ambiente de execução). Além disso, não há dependência de implementação, como por exemplo, os tamanhos dos tipos primitivos não diferem entre si, são independentes da maquina em que está a aplicação. Assim, o tipo *int* possui sempre um tamanho de 32-bits em Java e em qualquer máquina que esteja sendo executado.

Coletor de Lixo – Garbage Colector

Se não houver nenhuma referência a um objeto que tenha sido criado na memória, o coletor de lixo destrói o objeto e libera a memória ocupada por ele.

O coletor de lixo é executado de tempos em tempos. Quando a JVM percebe que o sistema diminuiu a utilização do processador, ela-JVM- faz com que o coletor de lixo execute e este vasculha a memória em busca de algum objeto criado e não referenciado. Em Java nunca se pode explicitamente liberar a memória de objetos que se tenha alocado anteriormente. O método abaixo PROPOE/SUGERE que a JVM vai utilizar recursos para reciclar objetos que não são mais utilizados. Mas não garante que daqui a 1 milisegundo ou 100 milisegundos o Garbage Collection vai coletar todos os objetos em desuso.

```
Runtime.gc();
System.gc();
```

O Garbage Collector é uma grande vantagem para desalocação de memória, que é um grande inconveniente para programadores que trabalham com ponteiros e necessitam liberar o espaço alocado, visto que é o proprio sistema que se encarrega desta limpeza, evitando erros de desalocação de objetos ainda em uso.

Segura

O Java fornece uma série de mecanismos para garantir a segurança dos aplicativos. Um programa em Java não tem contato com o computador real; ele conhece apenas a máquina virtual (JVM). A máquina virtual decide o que pode ou não ser feito. Um programa Java nunca acessa dispositivos de entrada e saída, sistema de arquivos, memória, ao invés disso ele pede a JVM que acesse.

2.6 Compilação dos Programas

2.6.1 Compilação Java

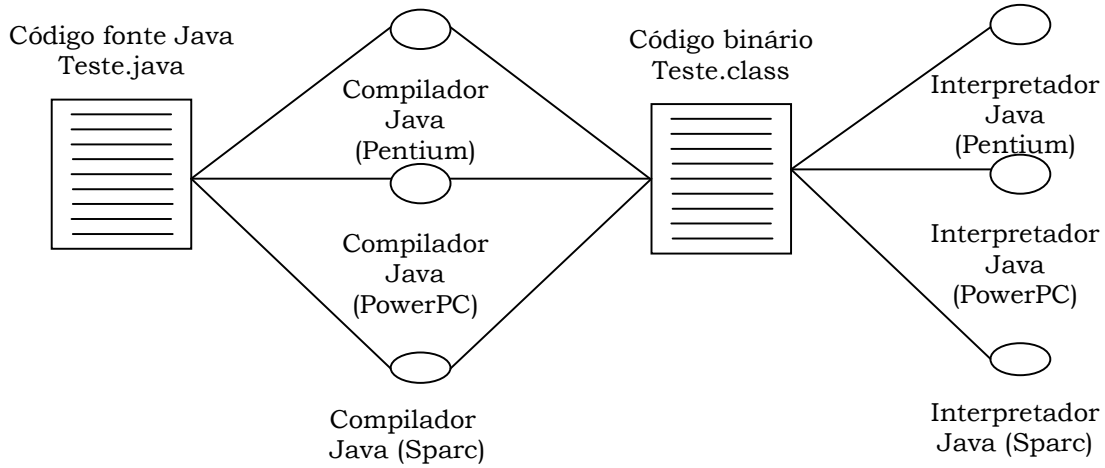


Figura 4. Compilação em Java

2.6.2 Compilação de Outras Linguagens

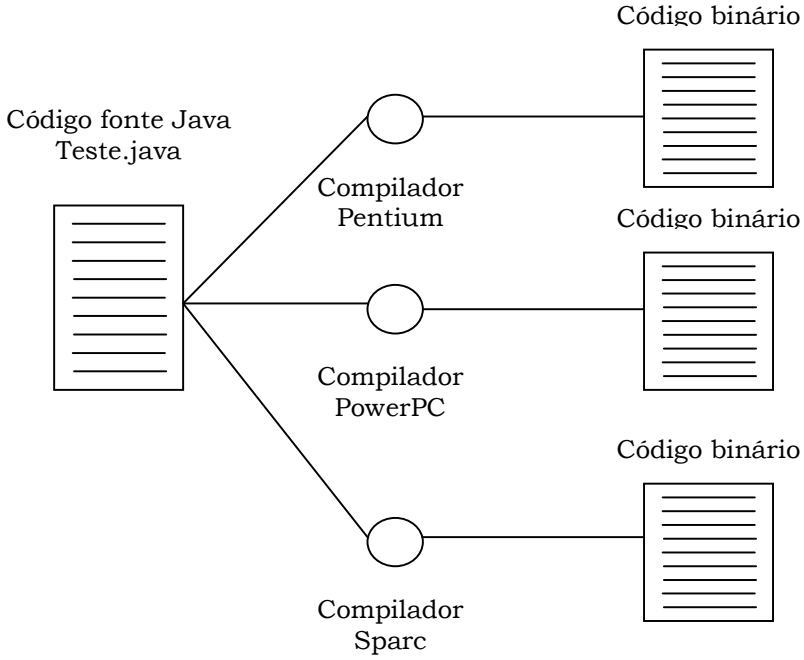


Figura 5. Compilação de outras linguagens

2.7 Ambientes Integrados de Desenvolvimento

Desde o lançamento de Java em 1996, várias empresas têm desenvolvido ambientes integrados de desenvolvimento, que agilizam a construção de programas em Java. Esses ambientes fornecem facilidades gráficas para manipulação dos recursos da linguagem. Entre as empresas e produtos existentes, destacam-se:

- ✓ VisualCafé - Symantec
- ✓ IBM Visual Age for Java - Sun
- ✓ Borland JBuilder – Borland
- ✓ Microsoft Visual J++ - Microsoft
- ✓ Oracle JDeveloper – Oracle
- ✓ Forte for Java – Sun
- ✓ Eclipse – Comunidade Aberta(IBM, Rational, Suse, Red Hat,..)
- ✓ JCreator - Xinox Software

Apesar das facilidades, a utilização destes ambientes pode trazer alguns inconvenientes como: a criação visual de componentes como botões, caixas de texto, barras de rolagem e outros, especificamente de um ambiente. Alguns ambientes incorporam grande quantidade de código na aplicação e uma hierarquia interna de organização dos pacotes, que obriga a aplicação final a carregá-los consigo. Particularidades dos produtos também criam uma dependência que contraria os "princípios" de Java.

Outro ponto a ser observado é a grande abstração provocada sobre o domínio da aplicação, ou seja, o programador acaba inserindo na aplicação possíveis bugs (erros de programação) dos ambientes e não tem controle para corrigí-los, e muitas vezes nem detectá-los. Assim, é mais razoável sempre ter domínio direto do código que se está desenvolvendo. Dessa forma, a utilização do JDK diretamente, gera aplicações mais robustas e enxutas.

2.8 Instalação do Compilador/Interpretador (JDK)

As versões do JDK estão disponíveis livremente na Internet no site da Sun Microsystems

http://java.sun.com

O JDK utiliza algumas variáveis de ambiente para indicar onde localizar as classes usadas pelo compilador e os arquivos necessários para compilar e interpretar os programas desenvolvidos. Para que o compilador fique visível em qualquer sessão de DOS ou por qualquer IDE utilizada e configurada, basta adicionar o código abaixo no PATH:

```
set path=c:\<diretorioJDK>\bin; %path%
```

Caso esteja sendo utilizado o Windows 2000,ou Windows XP, basta acrescentar esta variável dentro do Painel de Controle, no *System*, onde há a descrição das variáveis de ambiente. Nas ultimas versões de Java, a partir da 1.2.x este procedimento do path não é mais necessario visto que no momento da instalação estas configurações ja são realizadas.

Caso desejo rodar o interpretador Java dentro de uma janela DOS, você provavelmente necessitara setar a variael CLASSPATH para o diretorio local, como segue:

set classpath=%classpath%;.;

O ".;" significa justamente que você esta setando o diretorio corrente, ou seja o diretorio onde você esta localizado. Por exemplo, se você esta localizado no diretorio *c:\daniela\arquivos*, e você seta o classpath com o ".;" é como se voce estivesse setando o proprio diretorio, como segue:

set classpath=%classpath%; *c:\daniela\arquivos*;

A variavel de ambiente CLASSPATH serve justamente para identificar o caminho das classes.

2.9 Estrutura das Aplicações Java

O desenvolvimento de aplicações em Java sempre é realizado através da manipulação de classes. Uma aplicação Java sempre terá a seguinte estrutura:

```
class NomeDaClasse {
      // Atributos
      // Métodos
      public static void main( String[] args ) {
            //corpo principal do programa
      }
}
```

Claro que uma classe Java tera somente a classe, os métodos e seus atributos, não contendo a parte do método *main*.

Uma aplicação em Java é caracterizada por possuir um método **main().** O método main é o método chamado pelo interpretador Java quando executado. A declaração do método deve ser rigorosamente: **public static void main(String[] args)**, onde define um método de classe de acesso público, sem retorno, que recebe como parâmetro um array de Strings de tamanho indefinido, representado pela variável **args.** O **args** é a declaração do objeto do Array de String, por isso pode ser atribuído a ele qualquer nome.

A disposição dos atributos e métodos é feita de forma aleatória, ou seja, não importa a ordem, apenas necessitam respeitar regras de validade de escopo das variáveis, assunto abordado posteriormente.

Abaixo é demonstrada uma primeira aplicação que somente imprime na tela uma mensagem qualquer. Seguem os passos para construção da aplicação:

1. Abrir um editor de textos ou um IDE Java e escrever o programa

```
class OlaMundo {
      public static void main( String[] args ) {
            //corpo do programa
            //Escreve na tela:  Ola Mundo.
            System.out.println("Ola Mundo");
      }
}
```

2. Salvar o arquivo dando a ele o mesmo nome da classe e com a extensão **".java".** Salvar arquivo com nome "OlaMundo.java".

3. Compilar o arquivo

```
javac OlaMundo.java
```

A compilação gerou um arquivo "OlaMundo.class", que é o bytecode, o arquivo a ser executado.

4. Execute o programa utilizando o interpretador Java, e deve imprimir na tela "Ola Mundo"

```
java PrimeiraApp
```

2.9.1 Elementos da Aplicação

public

É um quantificador do método que indica que este é acessível externamente a esta classe (por outras classes que eventualmente seriam criadas). Este tópico será abordado posteriormente.

static

É um qualificador que indica que este método é um método de classe, ou seja, há uma cópia somente por classe. Os métodos *static* podem ser invocados, mesmo quando não for criado nenhum objeto para a classe, para tal deve-se seguir a sintaxe:

```
<NomeCasse>.<NomeMetodoStatic>(argumentos)
```

void

*S*emelhante ao *void* do C/C++, corresponde ao valor de retorno da função. Quando a função não retorna nenhum valor ela possui a palavra reservada *void* no local de retorno, uma espécie de valor vazio que tem que ser especificado.

main

É um nome particular do método que indica para o compilador o início do programa, é dentro deste método e através de interações entre os atributos e argumentos visíveis nele que o programa se desenvolve.

String[] args

É o argumento do método *main* e por conseqüência, do programa todo, é um array de *Strings* que é formado quando são passados ou não argumentos através da invocação do nome do programa na linha de comando do sistema operacional.

{ .... }

“Abre chaves” e “fecha chaves”: delimitam um bloco de código (semelhante a BEGIN e END em Pascal).

System.out.println

Chamada do método *println* para o atributo *out* da classe *System*. O argumento é uma constante do tipo *String*. *println* assim como o *writeln* de Pascal, imprime na saida padrão a *String* e posiciona o cursor na linha abaixo, analogamente *print* não avança a linha.

Passagem de Parâmetros da Linha de Comando

Aplicações em Java permitem que se passem parâmetros através da linha de comando. Os parâmetros são separados por espaço em branco. Para passar um parâmetro com espaços em branco deve colocá-lo entre aspas duplas.

Os parâmetros são passados para as aplicações através do array de Strings do método **main.** Através do método **length** pode-se verificar o número de parâmetros passados. O acesso é feito indicando-se a posição no array, sempre iniciando em 0. O método **length()** do parâmetro **args[x]**, permite que se obtenha o tamanho da String passada para este parâmetro. Segue um exemplo:

```
class AppParametro {
   public static void main (String[] args){
         System.out.println("Parametros:"+args[0]+""+args[1]+""+
args[2]);

         System.out.println("Tamanho terceiro parametro:"+
args[2].length());

         System.out.println("Quantidade Parametros:"+args.length);

   }
}
```

Caso o programa acima fosse executado com os seguintes parâmetros:

```
java AppParametro Daniela Claro “Menina bonita”
```

Os valores que seriam impressos na tela do DOS seriam :

```
Parametros:Daniela Claro Menina Bonita
Tamanho terceiro parametro:13
Quantidade Parametros:3
```

3. Fundamentos da Linguagem

3.1 Comentários

O Java suporta comentários como C. Qualquer informação especificada entre os caracteres de comentário será ignorada pelo compilador. Os tipos de comentários são os seguintes:

Comentário de bloco
/ texto */*
Todo o texto é ignorado. Este tipo de comentário pode ocupar várias linhas.

Comentário de Linha
// texto
Todo o texto depois de // até o final da linha será ignorado.

Comentário do JavaDoc
/** Comentário para documentação
*/

Este ultimo é um tipo especial de comentário que vai armazenar as diretivas para a geração automática da documentação das classes, utilizando o JAVADOC.

A regras que tratam dos comentários são:
- ✓ Comentários não podem ser aninhados
- ✓ Não podem ocorrer dentro de Strings ou literais
- ✓ As notações /* e */ não tem significado especial dentro dos comentários //.
- ✓ A notação // não tem significado especial dentro dos comentários /* e /**.

3.2 Palavras Chaves

Toda linguagem tem um grupo de palavras que o compilador reserva para seu próprio uso. Essas palavras-chaves não podem ser usadas como identificadores em seus programas.

São elas:

```
abstract    do          implements  package     throw
boolena     double      impor       private     throws
break       else        inner       protected   transient
byte        extends     instanceof  public      try
case        final       int         rest        var
cast        finally     interface   return      void
catch       float       long        short       volatile
char        for         native      static      while
class       future      new         super
const       generic     null        switch
continue    goto        operator    syncronized
default     if          outer       this
```

3.3 Tipos de Dados

Java é uma linguagem fortemente tipada, ou seja, todas as variáveis definidas na linguagem devem possuir um tipo de dados definido. Além disso, a linguagem possui os tipos primitivos de dados que permite que os mesmos tenham tamanhos pré-determinados (fixos) para cada tipo, independente da máquina que o programa está sendo compilado ou interpretado. Não há tamanhos dependentes de maquina para os tipos de dados como existe no C e no C++. Este fator é um dos mais importantes em relação a portabilidade.

A linguagem Java possui oito tipos primitivos: byte, short, int, long, float, double, char e boolean.

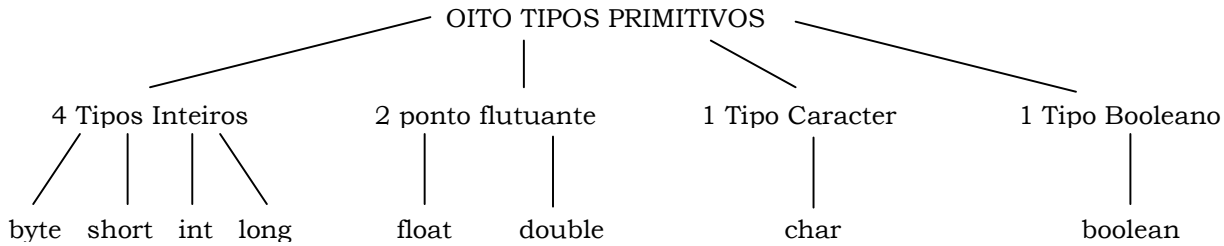


Figura 6. Tipos Primitivos do Java

Estes tipos primitivos possuem tamanhos distintos e são caracterizados como segue na Tabela 1.

Tipo	**Tamanho**	**Exemplos**
byte	8 bits	-128, 127, 0
short	16 bits	32768, -110, 49, 50
int	32 bits	2145848556, 23, 29
long	64 bits	987654367765, -1L
float	32 bits	0.2329, -654.213, 78.9
double	64 bits	35423.4589, 9999999
char	16 bits	‘i’, ‘*’, ‘!’, ‘d’
boolean		true, false

Tabela 1. Tipos primitivos de dados

3.4 Variáveis ou atributos

Em Java, toda variável ou atributo possui um tipo definido antes da declaração da variável. Os atributos ou variáveis são locais onde os dados são mantidos. O tipo de um atributo determina o tipo de informação que pode ser armazenada nele.

Atributos em Java podem ser declarados no corpo da classe ou podem ser declarados localmente em qualquer parte da implementação de um método. A declaração de atributos no Java segue a sintaxe:

```
<tipo> <identificador> [= <valor inicial>];
```

A atribuição é feita através do operador "=":

```
<identificador> = <valor>;
```

Exemplos:

```
int diasFerias, outroInteiro;
double salario;
float umFloat = 0.5;
char letra = 'i';
boolean achou = false;
diasFerias = 30;
....
```

Os tipos de dados juntamente com as variáveis podem ser declarados e somente depois inicializados com valores determinados, como segue o exemplo da variável diasFerias. Na primeira linha:

```
int diasFerias; //Somente a declaração da variável ou atributo
diasFerias = 30; //Inicialização da variável com o valor 30
```

3.5 Constantes

As constantes em Java são definidas através da palavra reservada FINAL. Esta palavra reservada tem por características atribuir somente uma vez o valor à variável. Para determinar que uma variável é uma constante, basta inserir a palavra reservada no inicio da definição da mesma. As constantes em Java têm por característica e estilo serem escritas em caixa alta e inicializadas com um determinado valor, como segue:

```
final double AUMENTO = 5,25;
```

3.6 Operadores

3.6.1 Operadores Aritméticos

Os operadores aritméticos são como outros operadores de outras linguagens tradicionalmente conhecidas. Exemplos deles são mostrados na tabela 2.

Operador	**Nome**	**Exemplo**
+	Adição	23+29
-	Subtração	29-23
*	Multiplicação	0.5 * salário
/	Divisão	100/42
%	Módulo	57/5

Tabela 2. Operadores Aritméticos

3.6.2 Operadores Relacionais

Os operadores relacionais permitem que se realizem comparações entre os operadores.

Operador	Nome	Exemplo
==	Igual	10 == 10
!=	Diferente	3 != 2
<	Menor	4 < 10
>	Maior	10 > 6
>=	Maior ou igual	3 >= 3
<=	Menor ou igual	5 <= 6

Tabela 3. Operadores Lógicos

3.6.3 Operadores Lógicos

Operador	Nome	Exemplo
&&	AND	(0 < 2) && (10 > 5)
||	OR	(10 >11) || (10 < 12)
!	NOT	!(1= = 4)
^	XOR	(1!= 0) ^ (3< 2)
?:	Condicional	3>2? (comandoSe):(comandoSenão)
&	AND binário	3(00000011) & 2(00000010)= 2(00000010)
|	OR Binário	3(00000011) | 2(00000010)= 3(00000011)

A divisão retorna um inteiro se os argumentos forem inteiros, mas se forem de outro tipo retorna um ponto flutuante. Segue abaixo um exemplo:

```
15/2 = 7
15.0 / 2 = 7,5
```

3.6.4 Atribuição Composta

Para facilitar a programação, Java oferece um tipo de atribuição chamada atribuição composta. Esse tipo de atribuição pode ser utilizado quando se deseja atribuir a uma variável X, o valor de X adicionado a 10, por exemplo.

Exemplos:

```
x += 10;    // Equivale x = x + 10;
x += y;     // Equivale x = x + y;
a -= b;     // Equivale a = a – b;
a *= 3;     // Equivale a = a * 3;
```

3.6.5 Operadores Incremental e Decremental

Os operadores incremental e decremental também foram desenvolvidos para facilitar a programação. O Java oferece uma maneira simples para fazer o incremento ou decremento em variáveis. O incremento é dado pelo operador ++ e o decremento pelo operador --. Este decremento ou incremento pode ser feito antes ou depois da utilização da variável, dependendo da necessidade do programador em relação a atribuição do dado. Colocando os operadores antes das variáveis será realizado primeiro a operação para depois o valor da variável ser utilizado. Ao contrário, caso o operador esteja localizado

após a variável, será realizado primeiramente a variável para depois ser realizado o incremento ou o decremento da variável.

Exemplos:
```
i++;    //Equivalente a: i = i +1;
i--;    //Equivalente a: i = i -1;
++i;    //Equivalente a: i = i +1;
--i;    //Equivalente a: i = i -1;
```

Estes operadores quando utilizados isoladamente não oferecem problemas aos programadores. Mas quando utilizados dentro de expressões, eles podem causar confusões e mau entendimentos quanto aos resultados da apresentação. Abaixo segue um exemplo deste problema(CORNELL, 2002):

```
int a = 7;
int b = 7;
int x = 2 * ++a;
int y = 2 * b++;
```

Qual o valor dos resultados?
a=8, b=8, x=16 e y=14

3.6.6 Operador Ternário

O operador ternário permite que se realize operações simplificadas de condições. Uma condição trabalhada no Java como IF-THEN-ELSE, pode ser manipulada através do operador ternário.

O operador ternário é formado por uma condição, e em caso positivo, é executada uma ação e em caso negativo, outra ação pode ser executada.

`x<y ? e1:e2` onde,

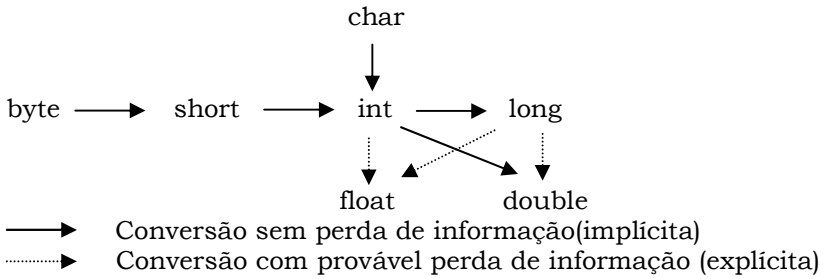
x<y é a condição, e em caso positivo será executado e1 e em caso negativo será executado e2. Segue abaixo um exemplo com valores:

```
String a = 1<2 ? “Menor” : “Menor”
```

O valor de a será `“Menor ”`, visto que 1 é menor que 2.

3.7 Conversões com Tipos Primitivos

A conversão entre tipos primitivos de dados é uma prática bastante comum e necessária. Deve-se ter bastante cuidado para não perder informações ou obter um resultado que não se esperava. Para efetuar uma conversão é necessário usar um mecanismo chamado *type cast*, onde o compilador vai converter de um tipo para outro. Isso é feito colocando o nome do tipo que se quer converter entre parênteses(CORNELL,2002).



Por exemplo, caso existam duas variáveis, uma do tipo short e outra do tipo int. Existem duas possibilidades para esta conversão:

```
Exemplo:
short  varshort;
int varInt;

varInt = varShort;
varShort = varInt;    //Tipo incompatível para conversão, short é
menor que int
```

Durante a compilação do código acima, um erro de tipos incompatível será gerado pelo compilador por tentar atribuir uma variável maior (varInt) para uma menor (varShort). Como esse tipo de operação faz perder valores ou informações de precisão, o Java faz você declarar explicitamente que você está consciente da conversão que deseja fazer. Neste caso, é necessário realizar uma conversão explícita, visto que para o compilador não será possível realizar a conversão de um *int* para *short*.

O código acima pode funcionar com o acréscimo do cast de tipo. O código correto seria:

```
short  varshort;
int varInt;

varInt = varShort;
varShort = (short) varInt;
```

Neste caso, varInt que tem 32 bits perderá 16 bits na conversão.

Conversões de Tipos Primitivos(CORNELL, 2002)

Tipo Original	**Tipo Destino**							
	byte	short	int	long	float	double	char	boolean
byte							C	X
short	C, P						C	X
int	C, P	C, P					C, P	X
long	C, P	C, P	C, P		C, P	C	C, P	X
float	C, P	C, P	C, P	C, P			C, P	X
double	C, P	C, P	C, P	C, P	C, P		C, P	X
char	C, P	C	C	C	C	C		X
boolean	X	X	X	X	X	X	X	

C – Utilizar cast explícito
P – perda de magnitude ou precisão
X – Java não permite conversão

3.8 Controle de Fluxo

A maioria dos programas toma decisões que afetam seu fluxo. As declarações que tomam essas decisões são chamadas de declarações de controle.

O controle de fluxo no Java pode ser utilizado tanto por sentenças condicionais, quanto por controle de estruturas de repetição. O controle de fluxo é similar ao C/C++, facilitando a utilização e manipulação destas estruturas.

3.8.1 Sentenças Condicionais

 # Declaração IF

As sentenças condicionais são determinadas pela estrutura IF-ELSE, onde no Java pode-se observar através do comando IF, como exemplificado abaixo:

Sintaxe:

```
if ( boolean )
    declaração1;
else
    declaração2;
```

Exemplo:

```
if (fim == true)
      System.out.println(“Término!”);
else
      System.out.println(“Continuando...”);
```

Para mais de um comando ser executado depois da declaração, utiliza-se o conceito de blocos, delimitados por chaves {}.
A declaração IF pode ser também utilizada quando a condição não é satisfeita como o ELSE, exemplificado abaixo:

Exemplo:

```
if (fim == true){
      cont = 0;
      System.out.println(“Término”);
} else {
      cont = cont +1;
      System.out.println(“Continuando...”);
}
```

A condição ELSE, sempre trabalha com o IF mais próximo delimitado por um bloco de código.

```
if (x>0)
      if (y>=0)
            sinal=0;
      else
            sinal=1;
```

3.8.2 Loops Indeterminados

Os loops indeterminados são representados por laços de repetição que contém uma condição de saída.

 # Declaração WHILE

Utilizada quando não se quer que o corpo do laço seja necessariamente executado. A expressão de comparação é avaliada antes que o laço seja executado.

Sintaxe:

```
while (booleano)
    declaração;
```

Exemplo:

```
while ( i != 0 ){
      salario = salario * 0.5;
      i--;
}
```

Declaração DO - WHILE

Utilizada quando se quer que o corpo do laço seja necessariamente executado, pelo menos uma vez. A expressão de comparação é avaliada depois que o laço for executado.

Sintaxe:

```
do
    declaração
while (booleano);
```

Exemplo:

```
 do {
      salario = salario * 0.5;
      i--;
 } while ( i != 0 );
```

3.8.3 Loops Determinados

Os loops determinados são responsáveis por executar um trecho de código por uma quantidade de vezes pré-determinada.

Declaração For

Fornece uma expressão para inicializar as variáveis, seguida por uma expressão de comparação e depois um lugar para incrementar ou decrementar as variáveis de laço. A declaração pode ser um comando simples ou um bloco de comandos.

Sintaxe:

```
for  (expressao; booleano; expressao)
    declaracao;
```

Exemplo:

```
for (i = 0; i < 20; i ++)
      salario = salario * 0.5;
```

As variáveis que são definidas dentro da sentença FOR, não podem usar os seus valores fora do loop. Isso ocorre, pois a variável perde o escopo, visto que está sendo definida para ser utilizada dentro do laço FOR.

3.8.4 Múltiplas Seleções

Declaração Switch

Aceita inteiro na expressão de comparação. Mas, como pode haver as conversões implicitas (cast implícitos) pelo compilador, ele passa a aceitar também os tipos char, byte, short ou int. Esse valor é procurado nas declarações `case` que vem depois da declaração `switch` e o código adequado é executado.

Sintaxe:

```
switch ( expressão ) {
      case valor: declaração;
      ...
      default: declaração;
}
```

Exemplo:

```
switch ( cmd ){
      case 1: System.out.println(“Item do menu 1”);
            break;
      case 2: System.out.println(“Item do menu 2”);
            break;
      case 3: System.out.println(“Item do menu 3”);
            break;
      default: System.out.println(“Comando invalido!”);
}
```

O comando *break* deve ser colocado ao final de cada cláusula case, pois senão todas as condições que forem satisfeitas, serão executadas. Ou seja, caso a condição do switch tenha uma condição que é satisfeita por mais de um *case*, todos que satisfizerem esta condição serão executados. Por isso, o `break` é importante para garantir que quando um case for satisfeito, mais nenhum deles serão executados também.

3.8.5 Manipulações diversas

 # Declaração Return

Declaração utilizada para transferir controle. Retorna informações de métodos, encerrando a execução e retornando para o local de onde o método foi chamado.

Sintaxe:

```
return expressão;
```

Exemplo:

```
class Teste{
      public int Potencia ( int base, int n){
            int result = base;
            for ( int i = 0; i < n-1;  i ++ )
```

```
                result = result * base;
            return result;
        }
    }
```

 # Declaração break e continue

Declarações de desvio usada para sair de um laço ou método antes do normal. O tipo determina para onde é transferido o controle. O break é utilizado para transferir o controle para o final de uma construção de laço (for, do, while ou switch). O laço vai encerrar independentemente de seu valor de comparação e a declaração após o laço será executada.

Exemplo:

```
int i = 0;
while (true) {
      System.out.println(i);
      i++;
      if ( i > 10 ) break;
}
```

A declaração *continue* faz com que a execução do programa continue voltando imediatamente para o início do laço, porém para a próxima interação.

Exemplo:

```
for (int i = -10; i<10; i++){
      if ( i == 0 )
            continue;
      System.out.println(1/i);
}
```

3.9 Arrays

Um array normalmente é usado para armazenar um grupo de informações semelhantes. Todos os itens de um array devem ser do mesmo tipo em tempo de compilação. Se o array for formado por tipos primitivos, eles devem ser todos do mesmo tipo.
Arrays são inicializados com o uso do operador *new*. Pense em cada elemento do array como um objeto distinto. O tipo mais simples de array é um array de dimensão de um tipo primitivo – por exemplo, um int. O código para criar e inicializar esse array segue abaixo:

```
int[] nums = new int [5];
```

Os colchetes depois do tipo de dado int, dizem ao compilador que é um array de inteiros. O operador new instancia o array e chama o construtor para cada elemento. O construtor é do tipo int e pode conter cinco elementos. Arrays podem ser multimensionais. Durante a instanciação, um array multidimensional deve ter pelo menos uma de suas dimensões especificadas. A seguir, exemplos de como criar um array bidimensional.

Exemplo:

```
int [][] numlist = new int [2][];
int[][] lista = new int[5][5];
```

Arrays podem ser inicializados na hora da criação, colocando-se os valores inicias desejados entre chaves {}. Não é necessário especificar o tamanho – Java irá inicializar o array com o número de elementos especificados.

Exemplo:

```
int[] nums = {1, 2, 3, 4, 5};
int[][] nums = {(1,1), (2,2), (3,3), (4,4), (5,5)};
```

Os arrays podem ser indexados por um valor byte, short, int ou char. Não se pode indexar arrays com um valor long, ponto flutuante ou booleano. Se precisar usar um desses tipos deve-se fazer uma conversão explícita. Os arrays são indexados de zero até o comprimento do array menos um.

```
long sum( int [] lista ){

        long result = 0;

        for ( int i = 0; i < lista.length; i++ ){
                result = result + lista[i];
        }
        return result;
}
```

3.10 Strings

Uma String é um tipo definido pela classe String e contém métodos e variáveis. Uma String é a unica classe que pode ser instanciado sem usar o operador *new*. A seguir exemplos da utilização de strings.

Exemplo:

```
//criação do String
String str = “Hello”;

int inteiro = 4;
String novo = str + “ valor do inteiro: ” + inteiro;
//Concatenação de strings

//extração dos cinco primeiros caracteres do string novo
String substr = novo.substring(5);
```

4. Orientação a Objetos em Java

Java é uma linguagem orientada a objetos e não é possível desenvolver nenhum software sem seguir o paradigma da orientação a objetos. Um sistema orientado a objetos é um conjunto de classes e objetos que interagem entre si, de modo a gerar um resultado esperado.

Java não é caracterizada como uma linguagem completamente Orientada a Objetos devido aos seus tipos primitivos de dados que não são caracterizados como classes e nem objetos. O Smaltalk é uma linguagem completamente Orientada a Objetos, onde qualquer manipulação e tipos de dados são classes. Mas, dizem alguns autores, que estes tipos primitivos permitiram que o Java fosse Orientada a Objeto, mas também facilitasse a manipulação dos dados realizada pelos programadores. E ainda, atribuem este fato ao sucesso da linguagem.

4.1 Vantagens da OO em Java

Ocultação de dados: No paradigma procedimental, os problemas causados pela utilização de variáveis globais são facilmente identificados. Se qualquer sub-rotina puder ter acesso a uma variável global, a perda ou erro de valores torna muito mais difícil a depuração e manutenção do sistema.

Em Java, atribuindo restrições de acesso às variáveis/atributos e métodos, é inserido um controle sobre o que e quem poderá manipular métodos e atributos. A utilização de modificadores de acesso (protected, public, private) permite a ocultação de dados.

Encapsulamento: O encapsulamento está intimamente relacionado com a ocultação de dados. Se um método fora da classe quiser alterar um atributo, ele tem que fazer isso chamando um dos métodos definidos na classe que se relaciona com este atributo. É a esta ocultação de informação e manipulação através de métodos que se denomina de encapsulamento.

Em Java quando quisermos obter o valor de um atributo de outra classe, esta classe deve oferecer um método com acesso liberado, que retorne o valor do atributo, denominado também de método GET. E ainda, caso se queira modificar um valor de um determinado atributo, também deve faze-lo através de métodos, os quais são denominados de métodos SET.

Facilidade de Manutenção: Normalmente sistemas legados precisam de atualizações. As linguagens devem fornecer recursos que garantam que os programas possam ser facilmente modificados para atender as novas necessidades de quem os mantém. Este é o principal objetivo da POO. Por exemplo, a facilidade de reutilização implica diretamente na facilidade de manutenção. A ocultação de dados torna o código muito mais simples ao entendimento. O encapsulamento facilita a distinção do relacionamento entre os dados e ação de nossos programas.

4.2 Componentes da Orientação a Objetos

4.2.1 Atributos (Variáveis)

Atributos ou variáveis são os locais onde os dados são mantidos. Como a maioria das linguagens de programação, os atributos são de tipos específicos. O tipo de um atributo determina o tipo de informação que pode ser armazenada nele. Normalmente utiliza-se a nomenclatura de atributo em relação às classes, às caracteristicas/propriedades das classes. E nomea-se variaveis em referência aos identificadores manipulados dentro dos métodos, ou seja, com um escopo diminuto e especifico. Neste ultimo temos como exemplo variaveis auxiliares, contadores, etc.

Atributos em Java podem ser declarados no corpo da classe ou podem ser declarados localmente em qualquer parte da implementação de um método, como acima denominamos aqui são as variaveis. A declaração de atributos no Java segue a sintaxe:

```
<tipo> <identificador> [= <valor inicial>];
```

A atribuição é feita através do operador "=":

```
<identificador> = <valor>;
```

Exemplos:

```
....
int umInteiro, outroInteiro;
float umFloat = 0.5;
char caracter = 'i';
boolean achou = false;
umInteiro = 90;
....
```

Os tipos podem ser primitivos ou objetos. Os primitivos representam valores numéricos, caracteres individuais e valores lógicos. Os tipos objetos são dinâmicos e definidos pelo usuário. Os objetos contêm uma copia da referência dos objetos.

Exemplo:

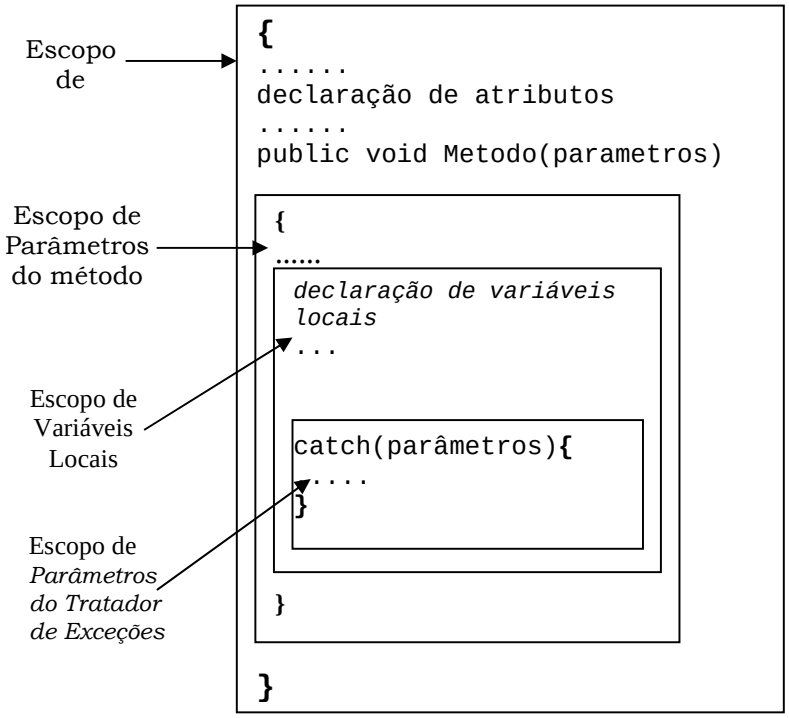
```
public class A{
      public static void main (String[] args){
            //atributo de tipo primitivo
            int a;
            a = 1;
            System.out.println("Saída do Valor de A:"+a);

            //atributo de tipo objeto
            int[] b;
            b = new int[3];
            b[0] = 10;
            b[1] = 20;
            b[2] = 30;
            for (int i=0; i<3; i++)
                System.out.println("Saída do Valor de B:"+b[i]);
      }
}
```

Para o tipo inteiro, o ambiente de execução sabe exatamente quanta memória alocar, no caso 4 bytes. Esta alocação sempre possui 4 bytes em Java, e não difere de máquina como na linguagem C/C++. Mas quando declaramos que queremos um array de inteiros,

o ambiente não sabe. Antes de colocarmos alguma coisa em a, temos que solicitar algum espaço de memória para a variável. Isso é realizado através do operador *new.*

4.2.2. Escopo de variáveis

O escopo de uma variável/atributo é o bloco de código dentro do qual ela/ele é acessível e determina quando foi criada e destruída. Basicamente, o que define o escopo de uma variável é o bloco onde ela se encontra. O bloco é determinado por chaves { }.
A figura abaixo ilustra o escopo de variáveis no Java:



4.2.3 Métodos

Um método em Java é uma sub-rotina – como as funções de C e Pascal. Os métodos têm um tipo de retorno e podem receber parâmetros. O modificador estático declara que o método terá um determinado tipo de comportamento dentro do programa orientado a objetos.

Na declaração de um método os modificadores precedem o tipo de retorno, que precede o nome do método e a lista de parâmetros. O corpo do método é colocado entre chaves.

```
<modificadores do método> tipo_retorno nome_método (<parâmetros>) {
    corpo do método
}
```

O corpo do método pode conter declarações de variáveis e de comandos. As variáveis não ficam restritas a declaração somente antes dos comandos, como acontece em C, ao contrário, podem ser declaradas em qualquer lugar. O tipo de retorno de um método pode

ser primitivo ou um objeto de uma determinada classe, ou como no método *main*, pode ser *void*, ou seja, não retorna nada.

```
class Parametro {
      public int adicionaDois(int i){
            i = i+ 2;
            System.out.println("Valor de i " + i );
            return i;
      }
      public static void main (String[] args ){
            int i = 11;
            System.out.println("Valor inicial de i " + i );

            //cria objeto da classe
            Parametro oParam = new Parametro();

            //método AdicionaDois, resultado é colocado em j
            int j = oParam.adicionaDois(i);

            //Mostra os valores de i e de j
            System.out.println("Valor de j " + j );
            System.out.println("Valor final de i " + i );
      }
}
```

Características Especiais

 # THIS

Os métodos possuem características especiais e tem a principal funcionalidade de acessar os atributos privados do objeto. Segue um exemplo de um método abaixo:

```
public void aumentarSalario(double percentual){
      double aumento = salário*percentual/100;
      salário += aumento;
}
```

O método acima possui dois parâmetros:
- Implícito: o próprio objeto
- Explícito: percentual

O próprio objeto é sempre passado pelo método mas de uma maneira não visível ao usuário. Este mesmo método acima pode ser visto como sendo :

```
meuObjeto.aumentoSalario(double percentual)
```

Onde o "meuObjeto" se refere ao objeto ao qual está chamando este método e dentro do método pode ser referenciado por uma palavra reservada denominada ***this***. O mesmo método pode ser reescrito utilizando o *this* como segue abaixo, evidenciando assim manipulação do atributo corretamente.

```
public void aumentarSalario(double percentual){
      double aumento = this.salário*percentual/100;
      this.salário += aumento;
}
```

#NULL
O *null* representa um objeto não existente. Quando o objeto é nulo significa que não temos acesso a nenhum de seus métodos ou atributos. Uma maneira de evitar que uma exceção ocorra ao se tentar acessar o objeto (*NullPointerException)* é testando se este objeto é nulo.

Exemplo:

```
....
Empresa emp;    //objeto não foi criado, então é null
emp.Atualizar ( nome, cgc );
.....
```

O trecho de código acima causaria um NullPointerException.

O trecho seguinte resolveria o problema:

```
....
if ( emp = = null)
      System.out.println("Objeto empresa não foi criado");
else emp.Atualizar ( nome, cgc );
.....
```

#Modificadores de Acesso

Modificadores de acesso ou regras de acesso são utilizadas na declaração de classes, métodos e atributos. São classificados como modificadores de acesso e de cunho geral.

- *private*
  É a forma de proteção mais restritiva. Permite acesso apenas a mesma classe, ou seja, é visível somente na classe. Utilizado pelos atributos da classe, permitindo que outros objetos ou classes somente acessem estas caracteristicas atraves da manipulação de métodos.

- *public*
  Permite acesso por todas as classes, sejam elas do pacote corrente ou não. Normalmente utilizados pelos métodos dentro de uma classe. Permite que todos façam manipulações dos atributos através dos metodos publicos. Garantindo assim o encapsulamento das informações.

- protected
  Permite acesso por todas as classes que estão no mesmo pacote e às subclasses de fora do pacote.

- Sem modificador
  Permite acesso por todas as classes do mesmo pacote.

4.2.4 Classes

As classes contêm os atributos e os métodos e ainda são responsáveis pela estrutura dos objetos. Os objetos, por sua vez, são as instâncias das classes.

A criação de uma classe em Java é iniciada pela palavra reservada **class.** A sintaxe é a seguinte:

```
[modificadores] class <NomeDaClasse> [extends <Superclasse>]
                        [implements <Interface1>, <Interface2>, ... ]
{
        //corpo da classe
}
```

[modificadores]: são os modificadores que impõem regras de acesso (public, protected, private) e de tipo (abstract, final).

As classes podem estar organizadas cada uma em seu arquivo fonte separado, onde o nome da classe deve ser o nome do arquivo gerado com a extensão `*.java`. Caso necessite, pode colocar classes em um único arquivo fonte, porém somente pode ter uma classe com o modificador *public* e é esta classe com este modificador que determinará o nome do arquivo fonte.

4.2.5 Pacote

Pacotes contêm classes e duas outras entidades de Java que veremos posteriormente: exceções e interfaces.

Uma função inicial de um pacote é como a de um *container*. Eles têm o papel de fornecer ao compilador um método para encontrar classes que se precisa para compilar o código. No método *System.out.println*, *System* é uma classe contida no pacote *java.lang*, junto com String.

Se a declaração *import* não for utilizada e desejarmos utilizar alguma classe externa, será necessário colocar o caminho da classe a cada acesso.

Exemplo:

```
import java.lang.*;
```

O asterisco no final diz ao compilador para importar todas as classes do pacote *java.lang* Esse é um dos vários pacotes incluídos na API. O compilador Java define implicitamente um pacote para as classes no diretório atual e o importa implicitamente. Esse é o motivo pelo qual não precisamos colocar explicitamente as classes que escrevemos em um pacote.

Para colocarmos determinada classe em um pacote, devemos inserir a diretiva *package* com o nome do pacote no início da classe.

Exemplo:

```
package Teste;
class Simples {
        public void Metodo( ) {
        }
}
```

No nosso exemplo acima, o pacote `java.lang.*` é o unico pacote no Java que não precisamos fazer a importação dele. Ou seja, se quisermos utilizar por exemplo a classe

System como no nosso primeiro exemplo, nos não precisamos importar o pacote *java.lang.** antes, pois ele já é implicitamente importado.

4.2.6 Objetos

Os objetos são as instancias das classes e através deles podemos manipular os métodos e atributos de um objeto. A declaração de uma variável ocorre da seguinte maneira:

Modelo:

<Classe> <identificador>;

Exemplo:

```
Date data;
```

Esta declaração não é um objeto, mas simplesmente uma definição do tipo da variável `data`. Como a variável data não se trata de um objeto, a mesma não pode se referenciar aos métodos e atributos do objeto. Neste caso NÃO se pode realizar o seguinte código:

```
Date data;
data.toString();
```

Isso não pode ocorrer visto que data ainda não é um objeto. Para se obter um objeto, é necessário inicializar a variável e para inicializar esta variável é necessário utilizar o operador `new()`.

Modelo:

<identificador> = new <Classe> ([param1], [param2]);

Exemplo:

```
Data data = new Data();
```

O *new* cria uma referência para um objeto, e esta referencia é armazenada na variável do objeto. As variáveis de objetos locais não são automaticamente inicializadas para *null*, por isso devem ser inicializadas explicitamente com o operador *new()* ou com o valor *null*.

Claro que podemos declarar e instanciar um objeto simultaneamente:

<Classe > < identificador> = new <Classe> ([param1], [param2]);

Exemplos:

```
...
Empresa emp;
emp = new Empresa( nome, end );
....
ou
...
Empresa emp = new Empresa( nome, end );
....
```

O acesso a métodos e atributos é igualmente simples e realizado através do operador ponto ".".

<objeto>. <atributo>;
Exemplo:

```
Empresa emp = new Empresa( );
emp.nome = “Sun”;
```

<objeto>. <método> ([<valorParam1>, <valorParam2 > ….]);
Exemplo:

```
Empresa emp = new Empresa( );
emp.cadastrar ( “Sun”, “EUA” );
```

4.3 Atributos de Instancia

Atributos de instância ou de objetos, são declarados em Java diretamente no corpo da classe, como se fossem atributos globais da classe.

[modificadores] <Classe> <identificador> [= new <classe> (.....)];

[modificadores] <Tipo> <identificador> [= <valorInicial>];

Um atributo de instância pode ser tanto um tipo simples (int, char, etc) como um objeto de uma classe <Classe>. O atributo pode também ser inicializado ao ser declarado.
Exemplo:

```
class Teste {
      Pessoa umaPessoa;
      Pessoa outraPessoa = new Pessoa ("João");
      int  inteiro;
      int numero = 15;
}
```

4.4 Métodos de Instancia

Métodos de instância ou de objeto são declarados também no corpo da classe.

```
[modificadores]<tipoRetorno><nomeMetodo>([<tipoParamN>
<ident.ParamN>, ....   ] )   [throws <ClasseThrowableN>,....];
```

O tipo de retorno pode ser qualquer classe ou tipo primitivo. Um método pode lançar exceções, isto é indicado através da palavra **throws**, seguida das classes e exceções que podem ser lançadas, isso será abordado em um capitulo mais adiante. Se o método retornar algum valor, é necessário incluir a palavra **return** seguida do valor de retorno no corpo desse método. Durante a execução, se o computador encontrar um return, ele pára a execução do método atual e retorna o valor indicado.

```
class Exemplo{
      private float salario;

      //outros métodos

      public String retorneSalario( ){
            return salario;
      }
}
```

4.5 Atributos e Métodos Estáticos (Static Fields / Static Methods)

4.5.1 Atributos Estáticos

Os atributos estáticos, também conhecidos como atributos de classe, se caracterizam por serem únicos por classe. Enquanto que os atributos de instancia ou atributos de objetos possuem uma copia para cada objeto.

Semelhante a declaração de atributos de instância, com a única diferença de que devemos inserir a palavra reservada **static** antes de cada atributo de classe:

[modificadores] static <Classe> <identificador> [= new <classe> (.....)];

[modificadores] static <Tipo> <identificador> [= <valorInicial>];

Exemplo:

```
class Inteiro {
      static int ValorMax = 100000;
}

class Empregado {
      private int id;
      private static int nextid;
}
```

De acordo com a classe Empregado citada acima, podemos observar que há um atributo de instancia, denominado *id* e outro atributo de classe denominado *nextid*[1]. Neste caso, todo o objeto Empregado tem o seu atributo *id*, mas somente um atributo *nextid* é compartilhado por todas as instâncias da classe.

Se eu possuo 1000 objetos no meu programa, isso significa que eu terei 1000 atributos *id*, 1 atributo *nextId*.

Se não existir nenhum objeto, o *nextId* existirá, visto que o mesmo pertence à classe e não a um objeto individual.

```
public void setId(){
      id=nextid;
      nextId++;
}
pedro.setId();
daniela.setId();
```

Teste de Mesa:

id(pedro)	id(daniela)	nextId
0	0	1
1	1	2
		3

4.5.2 Métodos Estáticos ou Métodos de Classe

A declaração de métodos de classe é idêntica a declaração de métodos de instância, exceto pela inclusão da palavra reservada **static** precedendo cada um.

[1] Este exemplo, como muitos outros presentes neste livro, foi adaptado do livro Core Java 2 (CORNELL, 2002), por ser considerado, por mim(Daniela), um dos melhores livros de Java que ja utilizei. Inclusive adotado nas Universidades onde lecionei, por ser didatico e muito explicativo.

[modificadores] static <tipoRetorno> <nomeMetodo> ([<tipoParamN> <ident.ParamN>, ...]) [throws <ClasseThrowableN>,....];

Exemplo:

```
class Exemplo{
      static void ImprimeMsg( ){
            System.out.println("Requisição inválida!");
      }
}
```

Os métodos estáticos operam com as classes, não trabalham no nível de objetos. Assim, não se pode trabalhar com o THIS, visto que este contém o valor da referência do objeto. Ainda assim não se pode acessar atributos de instancia dentro de métodos estáticos, somente se pode manipular atributos estáticos.

```
public static int getNextId(){
      nextId++;
}
```

Os métodos de classe ou métodos estáticos são usados em duas situações:

- Quando um método não necessita acessar o estado dos seus objetos, pois todos os parâmetros necessários são parâmetros explícitos.
- Quando um método somente precisa acessar atributos de classe.

4.5.3 Métodos Destrutores

Cada classe pode ter um destrutor. O destrutor é chamado quando o objeto é jogado para a coleta de lixo, portanto não se sabe quando o destrutor será chamado. Esse é um bom local para fechar arquivos, liberar recursos da rede enfim encerrar algumas tarefas. O destrutor em Java é chamado *finalize*. Ele não possui tipo de retorno e não assume nenhum parâmetro.

```
class Teste{
      finalize ( ){
      //tarefas a serem encerradas
      }
}
```

4.5.4 Métodos Construtores

São métodos especiais, executados automaticamente toda vez que uma nova instância da classe é criada. São utilizados para realizar toda a inicialização necessária a nova instância da classe. A declaração é semelhante a qualquer outro método, a não ser pelo nome dos métodos construtores ser o mesmo da classe e eles não possuírem um tipo de retorno.

```
class Empresa{
      String nome;
      String endereço,
```

```
        Empresa ( ){
              nome = "não definido";
              endereço = "vazio";
        }

        Empresa ( String nm_empresa,   String end_empresa ){
              nome =  nm_empresa;
              endereço = end_empresa;
        }
  }
```

4.6 Passagem de Parâmetros: por valor e por referência

A passagem de parâmetros é diferenciada para variáveis de tipos primitivos de maneira diferente das variáveis de tipo referência (objetos) quando passadas para um método. Todas as variáveis primitivas são passadas para os métodos por valor. Isso significa que uma cópia da variável é feita quando ela é passada para o método. Se manipularmos a variável dentro do método, o valor original não é afetado - somente a cópia.

```
class Parametro{
      static int AdicionaQuatro( int i) {
            i = i + 4;
            System.out.println("Valor da cópia local: " + i );
            Return i;
      }
      static static void main( String S[ ] ) {
            System.out.println("Passagem de parâmetros! " );
            int i = 10;
            System.out.println("Valor original de i: " + i );
            int j = AdicionaQuatro(i);
            System.out.println("Valor de j: " + j );
            System.out.println("Valor corrente de i: " + i );
      }
}
```

Quando se executa este programa, obtém-se a seguinte saída:

```
Passagem de parâmetros!
Valor original de i: 10
Valor da cópia local: 14
Valor de j: 14
Valor corrente de i: 10
```

Observação
O valor de i não muda, embora tenha sido acrescentado 4 a i, no método AdicionaQuatro.

Em Java, considera-se que não há passagem por referência, visto que sempre é feita uma copia do valor em se tratando de manipulações de métodos. Assim, o método não altera o valor da referencia passada, visto que dentro de um método os parâmetros passados perdem o escopo quando mostrados os seus valores. Assim, em Java, considera-se que há somente passagem de parâmetro por valor. O método obtém uma copia do valor do parâmetro.

Na passagem por referência, as variáveis são alteradas se manipuladas dentro de um método, mas a passagem continua sendo por valor, ou seja, uma cópia do valor da referência é realizada (CORNELL, 2002). A seguir, um exemplo que ilustra essa abordagem:

```
            public class ParametroReferencia{
            static void MudaArray ( int[] variavel   ) {
                  variavel[2] = 100;
            }
            public static void main( String[] S ) {
                  int umArray[] = new  int [3];
                  umArray[2] = 10;
                  System.out.println("umArray[2] antes = " + umArray[2]);
                  MudaArray( umArray );
                  System.out.println("umArray[2]    depois    =    "    +
umArray[2]);
            }
      }
```

Quando se executa este programa, obtém-se a seguinte saída:

```
      umArray[2] antes = 10
      umArray[2] depois =  100
```

Os valores do array são alterados pois está se trabalhando com uma cópia do valor da referência. Mas se fossem passados dois objetos para o método MudaArray, neste caso o objeto externo não alteraria o seu valor, visto que foi passado internamente somente uma cópia do valor da referência, e não a referência propriamente dita. Assim, o Java sempre realiza as passagens dos parâmetros por valor.

Observação
A passagem de parâmetros em String, não segue as regras de tipos de referência quando passada como parâmetro. As variáveis String agem como tipos primitivos. Quando uma String é passada para um método, na realidade está se trabalhando com uma cópia. As alterações dentro do método não irão afetar o original passado como parâmetro.

4.7 Herança

O conceito de herança vem do mundo real, onde todos nós herdamos características dos nossos pais e parentes. Sempre é anexado à herança o conceito de "É-UM".

Gerente É – UM Empregado

onde, Gerente é-um Empregado. Em Java, nós simbolizamos o "é um" através da palavra reservada **extends**.

```
      class Gerente extends Empregado
```

As subclasses em Java possuem mais funcionalidades que suas superclasses, pois as subclasses possuem as características herdadas e as suas próprias características.

```
class Gerente extends Empregado{
      private double bônus;
      public void setBonus(double d){
```

```
            bonus=b;
      }
}
```

O método setBonus() pertence à classe Gerente e não ao Empregado. Na classe Gerente, podemos utilizar os métodos herdados da classe Empregado.

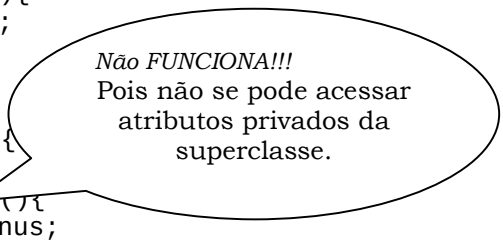
```
Gerente chefe = new Gerente();
chefe.setBonus(500);
```

Pode-se utilizar o `getNome()` e o `getCidade()` que o Gerente herda automaticamente da superclasse.

Imaginemos que o Empregado possui um método denominado `getSalario()` e um atributo privado salário. Porém o salário de um Empregado é X, mas de um Gerente é X + bônus.

Assim será necessário fazer uma sobreposição(override) do método `getSalario()`, onde este método na classe Gerente será reescrito e adicionado o valor do bônus.

```
class Empregado {
      priavate String nome;
      priavate double salario;

      public String getSalario(){
            return this.salario;
      }
}

class Gerente extends Empregado{
      private double bônus;
      public  double getSalario(){
            return salario + bonus;
      }
}
```



Os atributos privados definidos na classe Empregado, somente podem ser acessados pela classe Empregado. O Gerente, poderá acessar o método *getSalario()*. Isso garante o encapsulamento das informações. Para acessar o método getSalario() da superclasse, é necessário usar uma palavra reservada denominada SUPER.

SUPER é uma palavra chave utilizada para acessar métodos da superclasse. O *super* direciona o compilador para invocar um método da superclasse. Normalmente ele é utilizado no caso de herança, e que se deseja que a classe filha utilize os recursos da classe Pai.

```
      public class Carro{
            public void LavaCarro( ){
                  //comandos para lavar o carro
            }
      }
      public class Gol extends Carro{
            public void LavaCarro( ){
                  super.LavaCarro( );//Acessa método da classe Carro
            }
      }
```

Também se pode utilizar o **super** com o construtor, porém ele deve sempre ser a primeira sentença do construtor, como no exemplo a seguir:

```
public class Carro{
      String carro, sabao, local;
      public Carro( String nome){   //Método construtor
            carro = nome;
            sabao = "marca";
            local = "lavajato";
      }
}
public class Gol extends Carro{
      public Gol ( ){
            super ( "gol" );  //Acessa construtor da classe Carro
      }
}
```

Quando nenhuma superclasse é declarada explicitamente, Java assume automaticamente que a *Superclasse* é a classe **Object**, ou seja, toda classe Java é filha, no minimo, da classe **Object.**

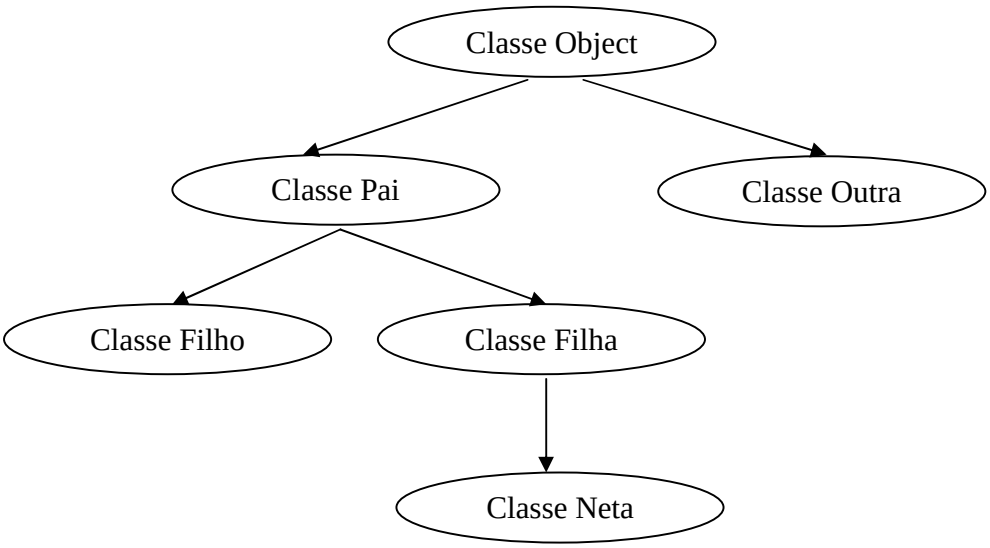


Figura 7. Demonstração da Hierarquia de Classes

Segue abaixo a descrição em codigo Java do esquema mostrado acima na Figura 7.

```
class Pai {
.......
}
class Filho extends Pai {
.......
}
class Filha extends Pai {
.......
}
class Neta extends Filha {
```

```
.......
}
```

O Java, automaticamente reconhece de qual classe um objeto é instanciado, se de uma classe filha ou de uma classe pai. E devido a tal, caso tenhamos uma sobreposição de métodos(override), o Java consegue diferenciar qual o método de qual classe aquele objeto esta se referenciando. Segue um exemplo:

```
Gerente chefe = new Gerente("Daniela",8000);
chefe.setBonus(500);
```

Bem, neste exemplo acima acabamos de denifir e instanciar um objeto chamado chefe da classe Gerente. Voltamos ao exemplo do Gerente, onde o salario do o chefe deve ser acrescido do bonus. Neste exemplo acima se solicitarmos o metodo salario de chefe, a nossa resposta seria 8000+500 = 8500.

Continuando, nos vamos criar um array de Empregados com três elementos, tal como:

```
Empregado emp = new Empregado[3];//Array de empregados
```

Para cada indice deste array nos vamos colocar uma instancia de empegado., como segue:

```
emp[0]  = chefe;
emp[1]  = new Empregado("Pedro",5000);
emp[3]  = new Empregado("João",3000);
```

Neste caso, o primeiro indice nos populamos com um empregado especial que foi chefe, pois devido à herança nos vimos que Chefe é um Empregado. Nos podemos fazer estas associações pois Empregado é uma super classe de Gerente, então é como se nos tivéssemos uma caixa grande Empregado, e conseguimos colocar uma caixa menor chamada Gerente dentro, como segue abaixo:

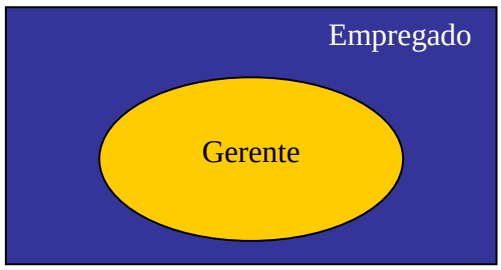


Figura 8: Representação das superclasses e subclasses

Assim, se nos mandarmos executar o metodo getSalario que contem nas duas classes sobrescrito, onde na classe Gerente ele é alterado para somar mais o bonus, quais seriam os nossos resultados?[CORNELL, 2002]

```
for(int i..){
      Empregado e = emp[i];
      System.out.println(e.getNome + e.getSalario());
}
```

Neste caso ele imprimiria *Daniela* com *8500*, *Pedro* com *5000* e *João* com *3000*. Mas como o Java reconhece que ele tem que pegar *Daniela* e executar o metodo *getSalario*() de *Gerente*?

O Java chama isso de Ligação Dinâmica ou Dynamic Binding, ou seja, a maquina virtual sabe qual o tipo atual do objeto. Assim faz uma ligaçao dinâmica e quando o objeto vem da classe Gerente, o *getSalario()* também vem da classe Gerente. A maquina virtual cria uma tabela de métodos, onde para cada classe ha as assinaturas correspondentes. Quando executamos um exemplo como acima, a JVM simplesmente acessa esta tabela.

Observações:

1- Todo objeto de uma subclasse é um objeto da superclasse, assim chefe pode ser um objeto da classe Empregado.
2- Em Java os objetos são polimorficos, ou seja, um objeto de Empregado pode se referenciar a Empregado ou qualquer uma das suas subclasses.
3- Não se pode através de Empregado chamar um método da subclasse, por exemplo, Empregado não pode chamar o método *getBonus().*
4- Não se pode associar ula referencia da superclasse a um objeto da subclasse, por exemplo, não podemos associar um objeto "*emp*" de Empregado a um objeto de Gerente, pois neste caso é como se eu quisesse colocar o nosso retangulo acima dentro do circulo.

4.8 Classes e Métodos Final

A palavra reservada FINAL em java pode significar o fim da implementaçao. Assim, em relação aos métodos, quando eles são definidos como final, significa que eles não podem ser sobrescritos(override) em classes herdadas. E em relação às classes, quando elas são definidas como final, elas nao podem ser herdadas.

Assim por exemplo:

```
final class Gol extends Carro{ }
//Evita que criem subclasses da classe Gol
```

Em relação aos métodos, eles não poderao ser sobrescritos.

```
public final void lavaCarro(){
      ...
}//Esse metodo nao pode existir em nenhuma outra classe filha.
```

Os atributos também podem ser final, como segue:

```
final String TESTE= "teste";
```

O *final* quando utilizado com os atributos, representa uma constante. Eles não permitem que os atributos sejam modificados depois que o objeto foi criado. Se uma classe é FINAL, automaticamente os métodos desta classe também são FINAL. Os atributos não seguem a mesma regra.

4.9 Classes e Métodos Abstratos

Segundo Cornell(2002), a medida que generalizamos na hierarquia da arvore, as classes vão ficando mais abstratas.

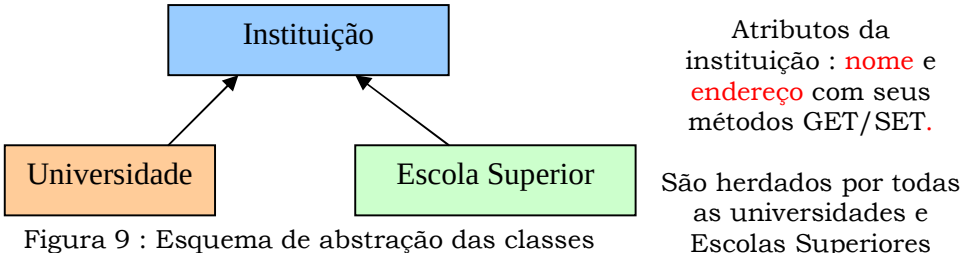


Figura 9 : Esquema de abstração das classes

Porém no exemplo da Figura 9, instituição é uma entidade ficticia que existe no papel e não fisicamente. Instituição é uma abstração de Universidades e Escolas Superiores. Assim, em relação aos atributos, sabemos que uma instituição tem nome e endereço, mas como ela é abstrata nao poderemos implementar ou seja, efetivamente dar o valor à estes atributos. Neste caso, nos utilizamos a nomenclatura *abstract* para designar que estes metodos não serão implementados na classe Instituição, mas sim nas classes filhas, como Universidades e Escola Superior. Neste caso, na classe Instituição nos declararemos a assinatura do métodos, como segue:

```
public abstract String getNome();
public abstract String getEndereco();
```

E nas classes filhas poderemos implementar o método. Se um método dentro de uma classe é abstrato, necessariamente a classe deve ser abstrata. Mas se uma classe ela é abstrata, ela não precisa ter metodos abstratos, podem ter todos os métodos concretos. Quando se herda de uma classe abstrata, nos podemos seguir dois caminhos:
1 –Não utilizar nenhum método abstrato, assim a subclasse deve ser tambem uma classe abstrata
2- Pode-se implementar todos os métodos, dai a subclasse não precisa ser abstrata.

Porém, uma classe abstrata ela não podera ser instanciada, ou seja, nenhum objeto pode ser criado como instancia desta classe. Os objetos somente podem ser criados baseado em classes concretas, mas eles podem ser definidos.

Vamos considerar o nosso exemplo da Figura 9, onde definiremos que instituição é uma classe abstrata, e as duas filhas, Universidade e EscolaSuperior são classes concretas. Assim nos poderemos ter:

```
Instituicao ins = new Universidade();
```

Classe abstrata

Classe concreta

4.10 Conversões Explicitas e Implicitas

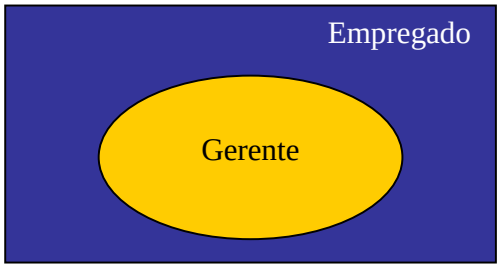
Embora tenhamos tratado no capitulo de Orientação a Objetos questões sobre Conversões explicitas e implicitas, neste secção do capitulo nos vamos voltar ao assunto em relação aos objetos.

Nos vimos nas conversões com tipos primitivos, podemos converter explicitamente os valores ou implicitamente, onde a propria JVM se encarrega ultimo deste processo. Então, no caso de conversões explicitas é o que chamamos de *cast*, como segue no exemplo:

```
double a= 3,4567;
int novoA= (int)a;
```

Converte para um inteiro descartando a parte fracional.

Em se tratando dos objetos o processo é o mesmo. Em Java todo objeto tem um tipo. Se desejo fazer uma conversão de uma subclasse para uma superclasse, é chamada conversão implicita, pois o complilador mesmo o fara. Assim é como se quisessemos converter (colocar dentro) do nosso retangulo(superclasse Empregado) a elipse(subclasse Gerente), como visto anteriormente:



O cast em relação aos objetos é necessario quando ocorre o processo inverso, ou seja, nos desejamos converter de uma subclasse para uma superclasse. Como se no nosso exmplo quisessemos colocar a classe Empregado(retangulo) dentro da classe Gerente(elipse). Neste caso o compilador nos mostrara uma mensagem onde informara que isso nao é permitido. Mas se voce desejar fazer isso de qualquer jeito, você devera então utilizar o CAST, ou seja, a conversão explicita. O *cast* so pode ser utilizado com às classes que possuem uma relação de hierarquia.

```
Gerente chefe = (Gerente) empregado[1];
```

Uma boa pratica de programação é utilizar o método `instanceOf` para garantir se um objeto é uma instancia de determinada classe e se positivo dai faz o cast, a conversão explicita.

```
if(emp[1] instanceOf Gerente){
      chefe = (Gerente)emp[1];
      ...
}
```

Através desta boa maneira de programação, citada por Cornell(2002), evitamos erros de querer fazer um CAST de uma classe Instituição, por exemplo, para uma classe Gerente, o que não é possivel, visto que não existe uma relação de hierarquia entre elas. Segue um exemplo que NÃO é possivel:

```
Instituição c = (Instituição)emp[1];
```

Isso não é possivel !

A utilização de CAST é muito utilizada em classes e métodos genéricos, que retornam objetos genéricos, visto dai que se utiliza o CAST para fazer a conversão para o tipo que você necessitar. Lembre-se sempre que não é necessario fazer a conversão de uma superclasse para uma subclasse somente para acessar os metodos das mesmas, pois neste caso o polimorfismo juntamente com a ligação dinâmica ja fazem isso sem precisar de CAST.

4.11 Interfaces

As interfaces em Java são responsaveis por descrever as assinaturas dos métodos e não como eles serão implementados. Além disso, a utilização das interfaces permite que simulemos a herança multipla, pois uma classe pode implementar uma ou mais interfaces. Assim, uma interface é a reunião das assinaturas dos métodos, sem definições reais. Isto indica que uma classe tem um conjunto de comportamentos (vindo da interface), além dos que recebe de sua superclasse. Uma interface não é uma classe, mas pode ser considerada um conjunto de requisitos que uma classe deve estar em conformidade. Segue um exemplo de criação de uma interface:

```
public interface Festa{
      void fazBolo(int tamanho);
}
```

Neste caso, toda classe que implementar a interface Festa, deve implementar o método fazBolo(). Além disso, todos os métodos da interface são automaticamente publicos, não sendo necessario colocar a palavra reservada PUBLIC neles. Em uma classe, a interface sera definida depois da palavra reservada ***implements.***

Uma classe pode implementar diversas interfaces, como dito acima, simulando assim a herança multipla. Porém, a classe deve implementar todos os métodos de cada interface, pois eles nunca são implementados na propria Interface, mas sempre nas classes que herdam delas. Outrossim, as interfaces não possuem atributos de instancia ou também chamados de atributos de objetos, e essa é uma das diferenças ja encontradas para as classes abstratas, onde os métodos também não são implementados na classe abstrata, mas nas classes que as herdam. Além disso, uma diferença muito importante entre as interfaces e as classes abstratas é que nas interfaces somos obrigados a implementar todos os métodos, mesmo que so utilizemos um deles. E nas classes abstratas, podemos implementar somente o método necessario a nossa utilizaçao. Segue um exemplo desta diferença:

```
public interface Festa {
      void fazBolo(int tamanho);
      boolean encherBolas(int quantidade);
      int convidarPessoa(String nome);
}

class Aniversario implements Festa {
...
}
```

Assim, a classe Aniversario obrigatoriamente tera que implementar todos os três métodos propostos pela interface Festa, mesmo que para este Aniversario você so deseje fazer o bolo.

5. Tratamento de Exceções

A elaboração de programas utilizando a linguagem Java, envolvendo assim o desenvolvimento das classes e interfaces com seus métodos e objetos retratam a maneira normal de desenvolvimento e não inclui qualquer caso especial ou excepcional. Na verdade, o compilador nada pode fazer para tratar condições excepcionais além de enviar avisos e mensagens de erro, caso um método seja utilizado incorretamente.

A utilização de exceções permite manipular as condições excepcionais do programa, tornando mais fácil o tratamento e a captura de erros. Assim, o tratamento de uma exceção é uma indicação de um erro ou uma exceção. Os erros são tratados nos lugares que eles podem ocorrer, dentro do proprio codigo.

O tratamento de exceção é projetado para lidar com erros sincronos, que vêm de dentro do programa. Não tratam de erros assincronos, como o término de uma operação E/S em um disco.

O bloco elementar do tratamento de exceção é o TRY – CATCH, como segue o exemplo abaixo:

```
try{
        ...
}catch(Tipo da Exceção) {
        ...
}
```

O bloco TRY pode ter um ou mais CATCH. Se nenhuma exceção for disparada, todos os CATCH são pulados e o processamento volta para a primeira linha de excução depois do tratamento, ou seja, depois do bloco TRY-CATCH. A instrução *try* diz basicamente "Tente executar o código dentro destas chaves e se exceções forem lançadas, vou captura-las de acordo com os tipos definidos. Cada um permite manipular qualquer e todas as exceções que são instâncias das classes listadas entre parênteses. Segue um exemplo de um tratamento de exceção basico:

```
class ExcecaoBasico{
        public static void main(String[] args){
                try {
                        int a = 2/0;
                }catch (Exception e ){
                        System.err.println("Aconteceu um problema:"+e);
                }
        }
}
```

Quando uma exceção é capturada, o bloco CATCH é executado. E a classe EXCEPTION, como é a superclasse das exceções, deve ficar por ultimo, em se tratando da captura de multiplas exceções.

O bloco TRY-CATCH pode ainda conter outro elemento e se transformar no bloco TRY-CATCH-FINALLY. A cláusula FINALLY é utilizada quando há uma alguma ação que não importa o que ocorra, a mesma terá que ser executada. Geralmente isso se refere a liberar algum recurso externo após sua aquisição, com a finalidade de fechar um arquivo, uma conexão com o banco de dados, assim por diante. E ela também pode ser usada juntamente com a cláusula CATCH, se referindo ao fato de uma não conformidade tenha ocorrido A diferença do FINALLY é que ele SEMPRE sera executado, caso HAJA uma exceção ou caso NÃO HAJA uma exceção. Porém, o FINALLY é opcional, ou seja, o TRY-

CATCH é obrigatorio, e podera ter ou não a clausula FINALLY. Segue abaixo um exemplo com o FINALLY:

```
try{
        System.out.println("Tentando executar o codigo.");
}catch (Exception e){
        System.err.println("Excecao capturada:"+e);
}finally{
        System.err.println("Finally executado!");
}
```

Segue um exemplo completo:

```
try {
  a = Integer.valueOf(numero1.getText()).intValue();
  b = Integer.valueOf(numero2.getText()).intValue();
  resultado = a / b;
  if (b < 0)
    throw new Exception();
  Message.texto("Este é o resultado:"+ resultado);
  Message.setVisible(true);
  }catch (ArithmeticException e){
    Message.texto("Gerou uma exceção Aritmetica - Divisão por zero");
    Message.setVisible(true);
  }catch (NumberFormatException e){
    Message.texto("Gerou uma exceção no Numero da divisão");
    Message.setVisible(true);
  }catch (Exception e){
    Message.texto("Gerou uma exceção Geral:"+e);
    Message.setVisible(true);
  }finally {
    MessageBox msgGeral = new MessageBox();
    msgGeral.texto("Um resultado, ou uma exceção, o programa está
funcionando!!!");
    msgGeral.setVisible(true);
  }
```

5.1 Criando uma exceção

Utilizamos a palavra reservada **`throw`** para gerar uma exceção. O throw determina um objeto a ser disparado, ou seja, especifica uma exceção. Por exemplo, se dentro de um bloco IF-ELSE, você desejar criar uma exceção, o fara utilizando a palavra throw seguida do nome da exceção, como segue abaixo:

```
import java.awt.*;
import java.awt.event.*;
import MessageBox;

class GeraExcecao extends Frame implements ActionListener{
int resultado;
int a;
int b;
MessageBox Message;
TextField numero1;
TextField numero2;
Button btnOk;

GeraExcecao (){
  setLayout (null);
  setSize(400,400);

  Message= new MessageBox();

  numero1 = new TextField();
  numero1.setBounds(60,40,80,30);
  add(numero1);
```

```
  numero2 = new TextField();
  numero2.setBounds(60,100,80,30);
  add(numero2);

  btnOk = new Button("Execute!!");
  btnOk.setBounds(160,100,70,30);
  add(btnOk);

 btnOk.addActionListener(this);
}

public void teste(){
  try {
     a = Integer.valueOf(numero1.getText()).intValue();
     b = Integer.valueOf(numero2.getText()).intValue();
     resultado = a / b;
     if (b < 0)
        throw new Exception();
     Message.texto("Este é o resultado:"+ resultado);
     Message.setVisible(true);
  }  catch (ArithmeticException e){
     Message.texto("Gerou uma exceção Aritmetica - Divisão por zero");
     Message.setVisible(true);
  }catch (NumberFormatException e){
     Message.texto("Gerou uma exceção no Numero da divisão");
     Message.setVisible(true);

  }catch (Exception e){
     Message.texto("Gerou uma exceção Geral:"+ e);
     Message.setVisible(true);
  }
}

public void actionPerformed(ActionEvent e){
  if (e.getSource() == btnOk){
     teste();
  }
}

public static void main (String args[]){
  GeraExcecao form = new GeraExcecao();
  form.setVisible(true);

}
}//Fechamento da classe
```

A classe Throwable é a classe responsavel pelos erros e exceções gerados no Java. Ela possui como subclasses ERROR e EXCEPTION. Os erros gerados da classe ERROR não são capturados, somente os erros da classe EXCEPTION deve ser capturados dentro da execução do programa.

5.2 Clausula Throws

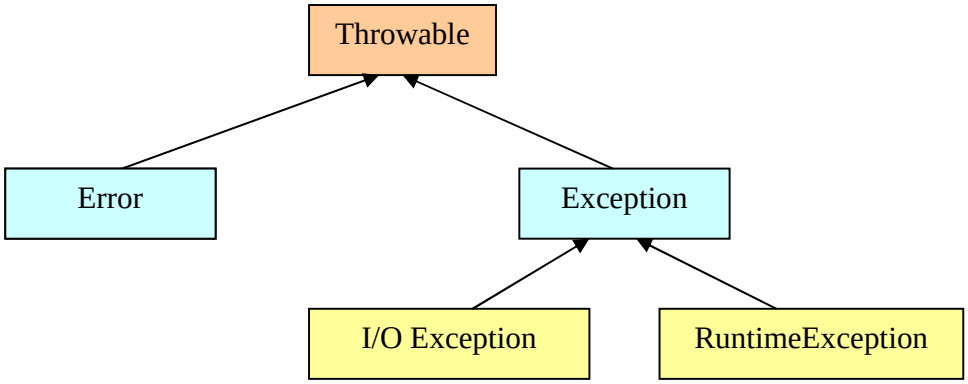
A clausula Throws é responsavel por listar todas as exceções que podem ser disparadas por um método. Porém, nem todos os erros e exceções disparados são listados na clausula Throws, as exceções geradas em tempo de execução(RuntimeException), devem ser tratadas no corpo do programa diretamente, dentro do TRY-CATCH. Se o método dispara qualquer exceção que não seja RuntimeException, elas devem ser capturadas na clausula Throws. Segue um exemplo:

```
class DivisaoPorZero extends ArithmeticException{
       DivisaoPorZero(){
```

```
            super("Foi gerada uma excecao devido a Divisao por zero");
      }
      DivisaoPorZero(String msg){
            super(msg);
      }
}
public class Excecao2{
      public static void main(String[] args) throws DivisaoPorZero {
            try {
                  int a=0, b = 0;
                  if (b!=0){
                        a = a/b;
                  }else{
                        throw new DivisaoPorZero();
                  }
            }catch (DivisaoPorZero e ){
                  System.err.println("Aconteceu um problema:"+e);
                  System.err.println("PrintMessage:"+e.getMessage());
                  e.printStackTrace();
            }catch (Exception f ){
                  System.err.println("Capturou excecao normal:"+f);
            }
      }
}
```

A programação Java foi desenvolvida para trabalhar com exceções, com a classe *Exception*. As exceções herdadas da classe Runtime Exception são exceções geradas normalmente pela programação mal feito do programador. O importante é capturar as exceções para que elas não sejam lançadas fora dos métodos.

Segue a estrutura hierarquica das exceções(Cornell,2002):



Em relação às classes e métodos herdados, caso um método de uma superclasse seja sobrescrito em uma subclasse, as exceções lançadas pelo método da superclasse são herdadas pela subclasse. O método na subclasse somente pode lançar as exceções da superclasse ou menos, mas não podera lançar mais exceçoes que existem na superclasse.

6. Abstract Window Toolkit - AWT

A AWT contém todas as classes para criação de interfaces do usuário e para trabalhos gráficos e com imagens. Um objeto de interface como um botão ou um scrollbar é chamado um componente. A classe Component é a raiz de todos os componentes AWT. Alguns componentes disparam eventos quando o usuário interage com eles. A classe **AWTEvent** e suas subclasses, são usadas para representar eventos que componentes AWT possam disparar.

Um container é um componente que pode conter componentes e outros containers. Um container pode também possuir um gerenciador de layout que controla a localização visual de componentes no container. O pacote AWT contém várias classes gerenciadoras de layout e uma interface para construir seu próprio gerente de layout.

Dentre as principais características de AWT destacam-se:

- ✓ Um amplo conjunto de componentes de interface para o usuário
- ✓ Um modelo de manipulação de eventos robusto
- ✓ Ferramentas gráficas e de visualização como classes de formatos, cores e fontes
- ✓ Gerenciadores de layout, para layouts de janelas flexíveis que não dependam de um tamanho de tela específico ou da resolução da tela.
- ✓ Classes de transferência de classes, para cortar e colar através de uma plataforma nativa.

O AWT fornece um conjunto de elementos de interface gráfica padrão (botões, janelas, menus, campos de edição, campos de seleção e outros) incluindo o sistema de tratamento de eventos que ocorrem nestes elementos e outras classes complementares.
O AWT foi projetado para que cada máquina virtual Java implemente seu elemento de interface. Por isso, um botão dentro de uma aplicação Java rodando no Windows, vai ter uma aparência diferente que um botão em uma outra aplicação no ambiente UNIX, porém a funcionalidade sera a mesma.

As classes estão no pacote **java.awt** . Pode-se citar o s seguintes elementos:

- ✓ ***Labels (rótulos) – classe java.awt.Label***
- ✓ ***Botões - classe java.awt.Button***
- ✓ ***Campos de texto - classe java.awt.TextField***
- ✓ ***Áreas de texto - classe java.awt.TextArea***
- ✓ ***Caixas de Verificação e Radio Buttons - classe java.awt.CheckBox***
- ✓ ***ComboBoxes - classe java.awt.Choice***
- ✓ ***ListBoxes - classe java.awt.List***
- ✓ ***Barras de Rolagem - classe java.awt.ScrollBar***
- ✓ ***Canvas (Telas de Pintura) - classe java.awt.Canvas***
- ✓ ***Frames - classe java.awt.Frame***
- ✓ ***Diálogos - classe java.awt.Dialog***
- ✓ ***Painéis - classe java.awt.Panel***
- ✓ ***Menus - classe java.awt.MenuBar, classe java.awt.Menu, classe java.awt.MenuItem***
- ✓ ***Cores - classe java.awt.Color***
- ✓ ***Fontes - classe java.awt.Font***

- ✓ ***Imagens - classe java.awt.Image***
- ✓ ***Cursores - classe java.awt.Cursor***

Classe java.awt.Component

```
java.lang.Object
  |
  +--java.awt.Component
```

A classe java.awt.Component fornece um conjunto de comportamentos padrão para a grande maioria dos elementos de interface. A classe Component é a superclasse da maioria dos elementos de interface da AWT. Podemos destacar alguns métodos:

setBounds(int x, int y, int width, int height)
Define a posição x, y, a largura e altura do componente.

setLocation(int x, int y)
Define a posição x, y do componente.

setSize(int width, int height)
Define a largura e altura do componente.

setEnable(boolean b)
Habilita/desabilita o foco para este componente.

setVisible(boolean b)
Mostra/esconde o componente.

setFont(Font f)
Define a fonte do componente.

setBackground(Color c)
Define a cor de fundo do componente.

setForeground(Color c)
Define a cor de frente do componente.

Classe java.awt.Container

```
java.lang.Object
  |
  +--java.awt.Component
        |
        +--java.awt.Container
```

Normalmente é preciso agrupar os objetos visuais antes de colocá-los na tela. Para agrupar componentes são utilizados Containers. Containers são componentes que podem abrigar outros componentes, inclusive outros Containers. Exemplos: Panel, Applet, Frame, Dialog. Os Containers herdam da classe Component, mas são usados como objeto para conter outros elementos visuais. O método **add()** adiciona componentes ao Container (diferentes opções de utilização estão disponíveis na API).

A disposição dos componentes adicionados ao Container depende da ordem em que foram adicionados e do Gerenciador de Layout definido para ele. Gerenciadores de Layout são classes que são responsáveis por gerenciar o layout do Container, organizando e dispondo os componentes segundo uma metodologia ímpar de cada gerenciador. O método **setLayout()** da classe Container é o método que define o gerenciador de layout a ser usado pelo Container. Dentre os gerenciadores de layout destacam-se:

FlowLayout
Layout de fluxo. Os componentes são adicionados linha por linha. Acabando o espaço na linha, o componente é adicionado na próxima linha. É o gerenciador de layout default para panels.

GridLayout
O Container é dividido em linhas e colunas formando uma grade. Cada componente inserido será colocado em uma célula desta grade, começando a partir da célula superior esquerda.

BorderLayout
Componente é dividido em direções geográficas: leste, oeste, norte, sul e centro. Quando um componente é adicionado deve-se especificar a direção em que será disposto. È o layout default para componentes Window.

Outra maneira de gerenciar o layout de um container é definir o layout como "null", mover cada componente para uma posição (x,y), e ajustar o tamanho (largura, altura) de cada um.

6.1 Elementos da Interface AWT

A seguir serão apresentados os principais elementos da interface AWT e os métodos mais utilizados de cada classe. A API Java fornece informações mais detalhadas.

Label
setText(String l)
Define o texto do rótulo.

String getText()
Retorna o texto do rótulo

Button
setLabel(String l)
Define o texto do botão

String getLabel()
Retorna o texto do botão

TextField
setText(String t)
Define o texto do campo

String getText()

Retorna o texto do campo

TextArea
setText(String t)
Define o texto da área

String getText()
Retorna o texto da área

setEditable(boolean b)
Define se a área pode ser editada ou não

appendText(String s)
Adiciona a string ao final do texto
Checkbox
setLabel(String l)
Adiciona a string ao final do texto

String getLabel()
Retorna o texto do checkbox

setState(boolena b)
Define o estado do checkbox true = on, false = off

boolean getState()
Retorna o estado do checkbox

Choice
addItem(String i)
Adiciona um item ao choice

String getItem(int pos)
Retorna o item da posição pos

int getItemCount()
Retorna o número de itens no choice

int getSelectedIndex()
Retorna a posição do item selecionado

String getSelectedIntem()
Retorna o item selecionado como um String

removeAll()
Remove todos os itens

List
addItem(String i)
Adiciona um item a lista

String getItem(int pos)
Retorna o item da posição pos

int getItemCount()
Retorna o número de itens na lista

int getSelectedIndex()
Retorna a posição do item selecionado

String getSelectedIntem()
Retorna o item selecionado

removeAll()
Remove todos os itens da lista

Frame
setTitle(String t)
Define o titulo do frame

setResizable(boolean b)
Define se o frame pode ser redimensionado ou não

setIconImage(Image img)
Define o icone para o frame

setMenubar(Menubar mb)
Define a barra de menu para o frame

Dialog
setTitle(String t)
Define o titulo do dialogo

setResizable(boolean b)
Define se o diálogo pode ser redimensionado ou não

setModal(boolean b)
Define se a janela é modal ou não

6.2 Tratamento de Eventos

O objetivo de uma interface gráfica com o usuário é permitir uma melhor interação homem x máquina. Quando ocorre uma ação do usuário na interface um evento é gerado. Um evento pode ser um movimento, um clique no mouse, o pressionamento de uma tecla, a seleção de um item em um menu, a rolagem de um scrollbar e outros. Eventos são pacotes de informações gerados em resposta a determinadas ações do usuário. Eventos também podem ser gerados em resposta a modificações do ambiente – por exemplo, quando uma janela da applet é coberta por outra janela.

Um evento sempre é gerado por um componente chamado fonte (source).Um ou mais objetos tratadores de eventos (listeners) podem se registrar para serem notificados sobre a ocorrência de eventos de um certo tipo sobre determinado componente (source). Tratadores de eventos ou listeners podem ser objetos de qualquer classe, entretanto devem implementar a interface correspondente ao(s) evento(s) que deseja tratar.

6.2.1 Classes de Eventos

Vários eventos podem ser gerados por uma ação do usuário na interface. As classes de tratadores de eventos foram agrupadas em um ou mais tipos de eventos com características semelhantes. A seguir a relação entre classes e tipo de eventos.

java.awt.event.ActionEvent - Evento de ação
java.awt.event.AdjustmentEvent - Evento de ajuste de posição
java.awt.event.ComponentEvent - Eventos de movimentação, troca de tamanho ou visibilidade
java.awt.event.ContainerEvent - Eventos de container se adicionado ou excluído
java.awt.event.FocusEvent - Eventos de foco
java.awt.event.InputEvent - Classe raiz de eventos para todos os eventos de entrada
java.awt.event.InputMethodEvent - Eventos de método de entrada com informações sobre o texto que está sendo composto usando um método de entrada
java.awt.event.InvocationEvent – Evento que executa métodos quando dispara uma thread
java.awt.event.ItemEvent - Evento de item de list, choice e checkbox se selecionado ou não
java.awt.event.KeyEvent - Eventos de teclado
java.awt.event.MouseEvent - Eventos de mouse
java.awt.event.PaintEvent - Eventos de paint
java.awt.event.TextEvent - Evento de mudança de texto
java.awt.event.WindowEvent - Eventos de janela

6.2.2 Tratadores de Eventos ou Listeners

Tratadores de Eventos ou Listeners são objetos de qualquer classe que implementem uma interface específica para o tipo de evento que deseja tratar. Essa interface é definida para cada classe de eventos. Então para a classe de eventos **java.awt.event.FocusEvent** existe a interface **java.awt.event.FocusListener.** Para a classe de eventos **java.awt.event.WindowEvent** existe a interface **java.awt.event.WindowListener** e assim sucessivamente.
As interfaces para tratamento de evento da AWT (pacote java.awt.event) são apresentadas a seguir com seus respectivos métodos (chamados conforme o tipo de evento).

Interface	**Métodos**
ActionListener	actionPerformed(ActionEvent)
AdjustmentListener	adjustmentValueChanged(AdjustmentEvent)
AWTEventListener	EventDispatched(AWTEvent)
ComponentListener	componentHidden(ComponentEvent) componentMoved(ComponentEvent) componentResized(ComponentEvent) componentShow(ComponentEvent)
ContainerListener	componentAdded(ContainerEvent) componentRemoved(ContainerEvent)
FocusListener	focusGained(FocusEvent) focusLost(FocusEvent)
InputMethodListener	caretPositionChanged(InputMthodEvent) inputMethodTextChanged(InputMethodEvent)
ItemListener	itemStateChanged(ItemEvent)

KeyListener	keyPressed(KeyEvent) keyReleased(KeyEvent) keyTyped(KeyEvent)
MouseListener	mousePressed(MouseEvent) mouseReleased(MouseEvent) mouseClicked(MouseEvent) mouseEntered(MouseEvent) mouseExited(MouseEvent)
MouseMotionListener	mouseDragged(MouseMotionEvent) mouseMoved(MouseMotionEvent)
TextListener	textValueChanged(TextEvent)
WindowListener	windowOpened(WindowEvent) windowActived(WindowEvent) windowDeactivated(WindowEvent) windowIconified(WindowEvent) windowDeiconified(WindowEvent) windowClosing(WindowEvent) windowClosed(WindowEvent)

Um componente que deseja que um evento seja tratado, deve especificar quem são os tratadores para o evento, através do método **add<tipo>Listener( ).**Quando um evento ocorrer, a Máquina Virtual Java identificará o tipo de evento e, se houver algum listener para o evento, repassará o evento para ele.
Por exemplo, o tratamento para o pressionamento de um botão que é um evento de ação. Deve-se escolher quem vai tratar o evento e implementar a interface **java.awt.event.ActionListener** nesta classe. Ainda, deve-se adicionar o listener aos tratadores de evento de ação do botão, utilizando **botao.addActionListener(<objetoActionListener>).**

6.2.3 Classe Adapter

Quando construirmos classes específicas para o tratamento de eventos e estas não possuem uma superclasse, podemos utilizar classes Adapter para facilitar a implementação. Classes Adapter são classes abstratas que implementam os métodos das interfaces que possuem dois ou mais métodos.

A grande vantagem de utilizar classes Adapter é que não é necessário implementar todos os métodos, o que na interface eles são obrigatorios, mesmo que não sejam utilizados. Utilizando as classes abstratas, deve-se apenas sobrescrever os métodos que tenham que realizar alguma tarefa. As classes Adapters e seus respectivos tratadores de eventos são ilustradas a seguir:

Listener	**Classe Adapter**
ComponentListener	ComponentAdapter
ContainerListener	ContainerAdapter
FocusListener	FocusAdapter
KeyListener	KeyAdapter
MouseListener	MouseAdapter
MouseMotionListener	MouseMotionAdapter
WindowListener	WindowAdapter

6.3 Componentes e Eventos Suportados

A seguir os principais elementos de interface da AWT e os eventos que suportam.

Componente	**Eventos Suportados**
Applet	ComponetEvent, ContainerEvent, FocusEvent, KeyEvent, MouseEvent, MouseMotionEvent
Button	ActionEvent, ComponetEvent, FocusEvent, KeyEvent, MouseEvent, MouseMotionEvent
Canvas	ComponetEvent, FocusEvent, KeyEvent, MouseEvent, MouseMotionEvent
Checkbox	ComponetEvent, FocusEvent, KeyEvent, MouseEvent, MouseMotionEvent
Choice	ComponetEvent, FocusEvent, KeyEvent, MouseEvent, MouseMotionEvent
Component	ComponetEvent, FocusEvent, KeyEvent, MouseEvent, MouseMotionEvent
Container	ComponetEvent, ContainerEvent, FocusEvent, KeyEvent, MouseEvent, MouseMotionEvent
Dialog	ComponetEvent, ContainerEvent, FocusEvent, KeyEvent, MouseEvent, MouseMotionEvent, WindowEvent
Frame	ComponetEvent, ContainerEvent, FocusEvent, KeyEvent, MouseEvent, MouseMotionEvent, WindowEvent
Label	ComponetEvent, FocusEvent, KeyEvent, MouseEvent, MouseMotionEvent
List	ActionEvent, ComponetEvent, FocusEvent, ItemEvent, KeyEvent, MouseEvent, MouseMotionEvent
MenuItem	ActionEvent
Panel	ComponetEvent, ContainerEvent, FocusEvent, KeyEvent, MouseEvent, MouseMotionEvent
TextArea	ComponetEvent, FocusEvent, KeyEvent, MouseEvent, MouseMotionEvent, TextEvent
TextField	ActionEvent, FocusEvent, KeyEvent, MouseEvent, MouseMotionEvent, TextEvent
Window	ComponetEvent, ContainerEvent, FocusEvent, KeyEvent, MouseEvent, MouseMotionEvent, WindowEvent

7. Interface Gráfica - Swing

A interface gráfica AWT permite que criemos programas que rodem em qualquer plataforma com a aparência da plataforma corrente, literalmente o “write once, run anywhere”.

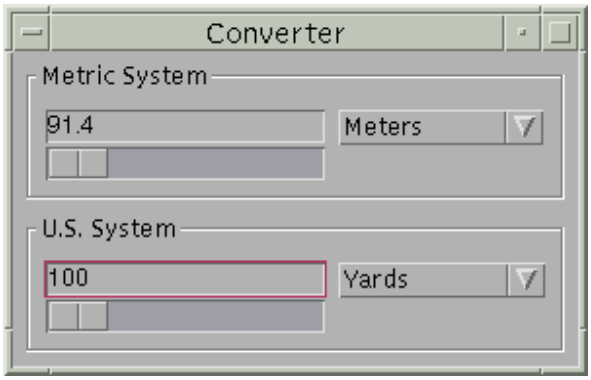
Isso era otimo para aplicações simples. Porém, alguns ambientes gráficos não tinham uma grande coleção de componentes gráficos, como o Windows, por exemplo, e tão pouco as funcionalidades que os usuários esperavam. Além do mais, os erros e “bugs” eram diferentes nas plataformas, e como dizia o Cornell(2002), “Write once, debug everywhere”.

Em 1996, a Netscape criou uma interface gráfica – IFC (Internet Foundation Class) onde os componentes eram “pintados” nas janelas. A Sun depois trabalhou junto para aperfeiçoar e deu o nome de Swing(Cornell,2002). Hoje, o Swing é o nome oficial para as interfaces que não são baseadas em ponto-a-ponto, que nem o AWT. O Swing é parte do JFC (Java Foundation Class). O JFC é vasto e contém muito mais coisas do que somente o Swing.

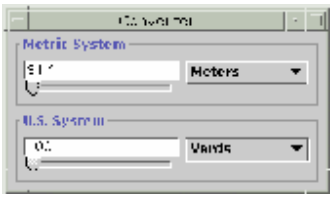
O Swing não substitui completamente o AWT, simplesmente fornece melhores componentes para a Interface Gráfica. Além disso, a captura de eventos houve uma grande mudança entre o Java 1.0 e o Java 1.1, e o Swing utiliza o modelo de captura de Eventos na versão 1.1.

Look & Feel: possibilita a qualquer programa que use componentes do JFC escolher o padrão como os componentes serão apresentados na tela. Os padrões incluídos junto com JFC são: **Java Look & Feel (Metal), Windows Look & Feel e CDE/Motif look & Feel.** A partir do Java 1.4 há também o **Mac Look & Feel.** A partir do Java 1.4 também foram lançadas outras tecnologias suportadas pelo Swing, como Drag & Drop e o suporte a API 2D. A seguir são mostrada três figuras de GUI que utilizam componentes Swing. Cada um mostra o mesmo programa, mas com aparência diferente.

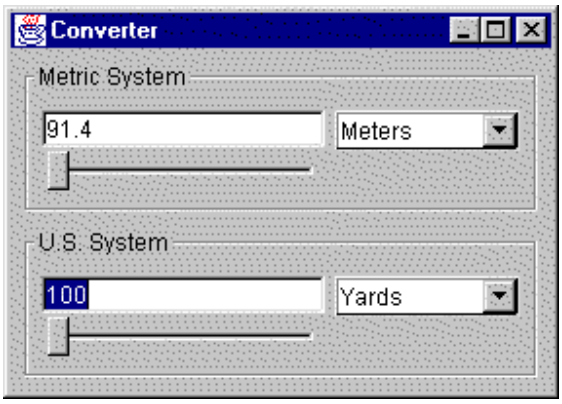
CDE/Motif Look & Feel



Java Look & Feel (Metal)



Windows Look & Feel



No Swing, os componentes são mais lentos para aparecer na tela que os componentes AWT. Porém, nas máquinas atuais, estas diferenças não chega a ser um problema.

Razões para utilizar o Swing:

- ✓ Maior e melhor conjunto de elementos para Interfaces do usuário
- ✓ Menos dependente da plataforma
- ✓ Menos "bugs" especificos da plataforma
- ✓ Mais robusto, mais caractristicas, maior portabilidade

É importante destacar que não se deve misturar componentes AWT com Swing.

7.1 Frames

Os frames são a janela mais alta da hierarquia, ou seja, não está inserida em nenhuma outra janela. Porém, os frames são containers, ou seja, eles podem conter outros componetes da Interface do usuário, como botoes, campos de texto, etc. Na versão do Swing, o nome do componente é JFrame que estende a classe Frame. Segue um exemplo de manipulação:

```
class PrimeiraInterface extends JFrame{
      public PrimeiraInterface(){
            setSize(300,200);
      }
      public static void main(String[] args){
            PrimeiraInterface frame = new PrimeiraInterface();
            frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            frame.show();
      }
}
```

Descreve o que deve ser feito quando o usuário fechar o frame.

Os frames iniciam sua vida invisiveis(para permitir adicionar componentes, antes de mostrar eles pela primeira vez).Logo, o *show()* deixa o frame visivel e coloca sempre em primeiro plano.

A manipulação de componentes dentro do AWT e do Swing são diferentes. No AWT para adicionar componentes ao Frame, nos utilizamos diretamente o método ADD. No Swing, todos os componentes devem ser adicionados dentro de um CONTENT PANE, como segue:

```
Container contentPane = frame.getContentPane();
frame.setContentPane(c);
```
→ onde 'c' é o componente.

Em relação ao tratamento de eventos, o Swing manipula a versão a partir do Java 1.1. Para isso aplicações Swing utilizam as classes Adapter para manipulação dos eventos e também as InnerClasses, ou seja, as classes que não possuem instancias declaradas, são classes que as instancias existem, mas não são definidas em um objeto pre-determinado. Segue abaixo um exemplo de tratamento de eventos utilizando a nova versão com as classes Adpter:

```
frame.addWindowListener(new WindowAdapter(){
      public void windowClosing(WindowEvent e){
            System.exit(0);
      }
});
```

Neste exemplo acima, graças a classe Adapter utilizada, nos não precisamos tratar todos os 7 métodos da WindowListerner, visto que nos estamos manipulando a WindowAdapter que é uma classe abstrata. Se estivessemos manipulando a interface WondowListener, nos teriamos que tratar cada um dos 7 metodos pertencente nesta interface, mesmo se nos fossemos utilizar somente o método *WindowClosing*.

8. Applets

Applets são programas Java embutidos em páginas HTML. Quando uma página HTML é carregada e é identificada a presença de um applet nessa página, o browser carrega também a classe Java do applet e o executa. Todas as classes utilizadas pelo applet também são carregadas pelo browser.

8.1 Hierarquia

```
java.lang.Object
  |
  +--java.awt.Component
        |
        +--java.awt.Container
              |
              +--java.awt.Panel
                    |
                    +--java.applet.Applet
```

8.2 Estrutura

Um applet deve possuir a seguinte estrutura:

```
import java.applet.Applet;

public class <NomeDaClasse> extends Applet
{
                    //corpo do Applet
}
```

A classe Applet está no pacote **java.applet,** devendo portanto ser incluida para acessar seus métodos. As applets também podem pertencer ao pacote do Swing, e o nome da mesma passa a ser JApplet. Alguns métodos são chamados automaticamente pelo interpretador:

init(): chamado quando o método é carregado ou descarregado
star(): chamado toda vez que a página que contém o applet é visitada
paint(): chamado sempre que for necessário desenhar o applet
stop(): chamado quando a página que contém o applet for substituída
destroy(): chamado quando todos os recursos do sistema precisam ser liberados. Por exemplo: quando o browser é fechado.

Esse métodos estão definidos na classe **java.applet.Applet** , porém não têm funcionalidade. Dessa forma, deve-se redefinir esses métodos ou alguns deles. Ao contrário das aplicações, quando um applet é carregado é criada uma instância da classe a que ele pertence. Quando um applet é carregado, é chamado o construtor do applet, depois os métodos init(), start() e paint(). Quando abandona-se a página, o método stop() é chamado e posteriormente o método destroy(). Os métodos init() e start() podem ser considerados os pontos de entrada de um applet.

8.3 Arquivos HTML

Como um applet está sempre embutido em uma página HTML, faz-se necessária a construção de uma página HTML, referenciando o applet. A inclusão de um applet em uma página HTML é feita através da tag **<APPLET>.**

```
<HTML><BODY>
…
<APPLET CODE = NomeDaClasse.class
[CODEBASE=diretório das classes]
[NAME = nome do applet]
WIDTH = 200
HEIGHT = 200 >
</APPLET>
…
</BODY></HTML>
```

Na tag <APPLET> devem obrigatoriamente ser definidos os atributos:
CODE – identifica o nome da classe do applet
WIDTH – largura do applet
HEIGHT – altura do applet

Opcionalmente pode-se definir:
CODEBASE: indica o diretório onde estão as classes. Por exemplo, a página html está no diretório /java e se deseja colocar as classes no diretório /classes.
NAME: um nome para o applet. Importante quando se deseja fazer comunicação entre applets.

8.4 Executando um Applet

Para executar um applet, deve-se carregar a página HTML em um browser que suporte Java, como o Internet Explorer, Netscape ou Mozilla. O JDK fornece o **appletviewer** que permite a visualização de applets.

```
appletviewer <nomeDoArquivoHTML.html>
```

8.4.1 Passagem de Parâmetros

Parâmetros são passados de páginas HTML para applets através da tag <**PARAM**> que deve estar entre <APPLET> e </APPLET>. A tag <PARAM> possui os atributos:

NAME: indica o nome do parâmetro
VALUE: indica o valor do parâmetro

```
<APPLET CODE=...>
<PARAM NAME="NomeParam1" VALUE="ValorParam1" >
<PARAM NAME="NomeParam2" VALUE="ValorParam2" >
…
</APPLET >
```

Dentro do applet, os parâmetros são recuperados através do método **getParameter().** Geralmente, mas não necessariamente, o código para recuperação dos parâmetros passados é feito no método **init().**

```
...
public void init()
{
      ...
      param1 = getParameter(“NomeParam1”);
      param2 = getParameter(“NomeParam2”);
      ...
}
...
```

Segue um exemplo mais completo da applet:

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

public class ParamApplet extends Applet{
      String codigo;
      public void init(){
            codigo=getParameter("codigo");
      }

      public void paint(Graphics g){
            g.drawString("Codigo: "+codigo ,20,20);
      }
}
```

Exemplo do arquivo HTML:

```
<HTML>
 <BODY background="#000000">
  <APPLET CODE=ParamApplet.class WIDTH=300 HEIGHT=100>
   <PARAM NAME="codigo" VALUE="55010">
  </APPLET>
 </BODY>
</HTML>
```

8.6 Restrições de Segurança

Por questões de segurança, applets não podem:
- ✓ Carregar bibliotecas ou definir métodos nativos
- ✓ Ler ou escrever arquivos no host onde está sendo executado
- ✓ Estabelecer conexões de rede com outro host diferente daquele de onde veio
- ✓ Executar programas no host onde está sendo executado
- ✓ Ler propriedades do sistema

Quem gerencia esta política de segurança é o browser através de um SecurityManager, que fiscaliza ações do applet. Quando ocorre uma violação de segurança, uma exceção do tipo SecurityException é gerada.

8.7 Principais Métodos

String getParameter(String nomeParam)

Retorna o valor do parâmetro *nomeParam*

ShowStatus(String msg)
Mostra a msg na barra de status do browser

String getDocumentBase()
Retorna o endereço da página HTML

String getCodeBase()
Retorna o endereço do applet

Image getImage(URL url)
Retorna a imagem da url especificada

GetAppletContext()
Retorna o contexto onde o applet está inserido

8.8 Objeto gráfico – Classe java.awt.graphics

A maioria dos elementos de interface da AWT são subclasses de **java.awt.Component.** A classe Component define um método chamado **paint()** utilizado para pintar o componente. Este método a principio, não faz nada. Sempre que um componente precisar ser repintado, **paint()** é chamado automaticamente.
Quando o método **paint()** é chamado, passa como parâmetro o objeto gráfico para o componente. Este objeto gráfico sempre será um objeto da classe **java.awt.Graphics.** A classe java.awt.Graphics incorpora os recursos gráficos do Java. Permite desenhar linhas, retângulos, círculos, imagens, pontos, e outros. As coordenadas do sistema gráfico do Java tem origem no ponto (0,0) no canto superior esquerdo e tem valores positivos a direita e abaixo da origem. A pintura é realizada pelo método **paint()** que é chamado automaticamente toda vez que o componente precisa ser repintado. Pode-se forçar a pintura do componente através do método **repaint()**.

8.9 Fontes

A classe utilizada para representar fontes é a classe **java.awt.Font**. A descrição de uma fonte é feita especificando-se o nome, estilo e tamanho da fonte.

```
Font f = new Font (<nomeFonte>, <estilo>, <tamanho>);
```

O nome da fonte pode ser: Dialog, DialogInput, Monospaced, Serif, SansSerif ou Symbol. O estilo da fonte pode ser: Font.PLAIN, Font.BOLD ou Font.ITALIC. A maioria dos componentes AWT possuem os métodos setFont() e getFont() para definir e para retornar a fonte do componente. Trocando a fonte do objeto gráfico:

```
public void paint(Graphics g){
      g.setFont(new Font("Symbol", Font.BOLD, 14));
}
```

8.10 Cores

A classe Java para representar cores é a classe **java.awt.Color.** Uma maneira de aproveitar a classe Color é aproveitar as cores já definidas na classe. Essas cores são basicas como: Color.white, Color.blue, Color.black, Color.lightGray, Color.red e etc.Outra maneira é criar uma instancia de Color, passando valores entre 0.0 e 1.0 para compor uma cor no padrão RGB. A seguir as três maneiras:

```
Color c1 = Color.white;
Color c2 = new Color(255, 0, 0);
Color c3 = new Color( 0.0f, 1.0f, 0.0f );
```

A maioria dos componentes AWT possuem métodos setForeground(), getForeground(), setBackground() para definição e retorno das cores de frente e de fundo utilizadas. Para trocar a cor do objeto gráfico:

```
public void paint(Graphics g ){
      g.setColor(Color.red);
}
```

9. Processamento Concorrente - Threads

No mundo da Informática, o processamento de multitarefas, ou seja, a capacidade de ter mais de um programa funcionando, parecendo que é simultaneo, dando uma impreséao de paralelismo, é relativamente recente. Imagine que o usuário esteja usando o processador de textos com um arquivo muito extenso. Ao iniciar, o programa necessita examinar o arquivo todo antes de utilizar. Com a utilização de multitarefas, o programa abre a primeira página, permitindo a sua manipulação, enquanto que continua o processamento no resto do documento. Um outro exemplo de processamento multitarefas é através da Web. Neste caso, o navegador carrega as figuras, os sons, enquanto está carregando a página.

Cada tarefa é chamada de linha de execução ou Thread. Cada programa que trabalha com uma interface gráfica tem uma linha de execução separada para reunir eventos da interface.

Para se utilizar das Threads há uma classe chamada *java.lang.Thread* que contém um método run() onde será embutido o código fonte da Thread.

```
public class PrimeiraThread extends Thread {

  public void run(){
    ....
  }

}
```

Para iniciar uma linha de execução ou thread deve-se executar o método *start()*. Este método quando chamado, fará com que o método *run()* seja executado, como segue abaixo:

```
PrimeiraThread pt = new PrimeiraThread()
pt.start();//executa o método run()
```

A maioria das vezes trabalha-se com Threads dentro de uma implementação, como em Java não podemos trabalhar com herança multipla, assim não poderemos estender à suas classes, há a interface Runnable, que permite também manipularmos as Threads.. A interface Runnable especifica apenas um método: *run()*. Assim podemos ter:

```
class X extends Thread
```

ou

```
class X extends JFrame implements Runnable
```

Como em PrimeiraThread se especifica a criação de uma instância da Thread, essa instância transfere a execução ao construtor que está elaborando a nova Thread (PrimeiraThread). Sempre que esta Thread inicia, o seu método *run()* é executado. Quando *start()* é chamado, a Thread indiretamente, chama o método run().

As Threads possuem vários estados que elas podem passar. Dentre eles, encontra-se o estado de bloqueio de uma Thread, ou seja, as Threads são classificadas como bloqueadas quando:

- ✓ Ocorre uma chamada do método SLEEP()
- ✓ A Thread chama o método WAIT()

- ✓ A Thread chama uma operação que está bloqueando E/S, ou seja, não retornará o controle enquanto a operação de E/S não sejam concluidas.

Para uma Thread sair de um estado de bloqueio, uma destas ações abaixo precisam ocorrer:

- ✓ Expirar a quantidade de tempo do método SLEEP()
- ✓ Uma operação de E/S terminou
- ✓ Se chamou o método WAIT(), então outra linha deve chamar o NOTIFYALL().

As Threads possuem prioridades e no Java definir estas prioridades é uma tarefa simples e fácil. A prioridade da Thread define a ordem de execução da mesma. Quando a CPU (schedule) escolhe uma Thread, ela verifica a prioridade mais alta. As Threads de prioridades iguais são escolhidas aleatoriamente pela CPU. Pode-se alterar as prioridades das Threads utilizando o método *setPriority(PRIORIDADE)*. Há constantes que definem MIN_PRIORITY, etc.

A Thread abaixo é executada em uma Applet e tem como objetivo mostrar um relógio em funcionamento ao mesmo tempo que a aplicação está sendo executada, visto que a própria aplicação é uma Thread.

```
import java.applet.Applet;
import java.awt.*;
import java.awt.Graphics;
import java.util.*;
import java.text.DateFormat;
import java.awt.event.*;
import MessageBox;

public class Thread1 extends Applet implements ActionListener,
Runnable{
  Button btnOk;
  TextField edNome;
  Label lbTitulo;
  Label lbMensagem;
  Panel pnSaida;
  MessageBox mensagem;

  Thread appClock;

  public void init(){
    setLayout(null);
    setSize(400,300);
    setBackground(Color.gray);

    pnSaida = new Panel();
    pnSaida.setBounds(20,20,330,40);
    pnSaida.setBackground(Color.yellow);
    add(pnSaida);

    lbTitulo = new Label("Gerador de Mensagens");
    lbTitulo.setBounds(170,25,200,40);
    pnSaida.add(lbTitulo);

    lbMensagem = new Label("Mensagem:");
    lbMensagem.setBounds(20,80,80,30);
    add(lbMensagem);
```

```
    edNome = new TextField("Daniela Claro");
    edNome.setBounds(20,110,200,24);
    add(edNome);

    btnOk = new Button("Mostra Mensagem!");
    btnOk.setBounds(250,90,120,40);
    add(btnOk);

    btnOk.addActionListener(this);
  }
  public void start(){
   appClock = new Thread(this, "Clock");
   appClock.start();
  }
  public void run(){
     while (appClock == Thread.currentThread()) {
        repaint();
   }
  }
  public void paint(Graphics g){
   Calendar cal = Calendar.getInstance();
   Date data = cal.getTime();
   DateFormat dateFormatter = DateFormat.getTimeInstance();
   g.drawString(dateFormatter.format(data),5,10);
  }
  public void actionPerformed(ActionEvent e){
    if (e.getSource() == btnOk){
      mensagem = new MessageBox(edNome.getText());
      mensagem.setVisible(true);
      showStatus(edNome.getText());
    }
  }
}
```

O método de classe *currentThread()* pode ser chamado para obter a thread na qual um método está sendo presentemente executado. Freqüentemente, dentro dos métodos chamados pelo método *run()* necessitará saber qual a thread que está em atual execução.

```
public class Sequencia implements Runnable{
  public void run(){
   while(true){
     System.out.println(Thread.currentThread().getName());
   }
  }
  public static void main (String args[]){
   Sequencia form = new Sequencia();
   new Thread(form, "Primeira Sequencia").start();
   new Thread(form, "Segunda Sequencia").start();
  }
}
```

O método *yield()* (ceder) dá uma chance de iniciar a execução, a qualquer outra Thread que queira ser executada. Se não houver threads esperando para serem executados, o thread que realiza o *yield()* simplesmente continuará.

```
public void run(){
   while(true){
     System.out.println(Thread.currentThread().getName());
     Thread.yield();
   }
}
```

9.1 Sincronização

Aplicativos com multiplas linhas de execução tem que compartilhar o acesso aos mesmos objetos, ou seja, cada linha pode chamar um método que modifica o estado do objeto. Neste caso, o paralelismo é desejável e poderoso, mas pode introduzir muitos problemas. Esta situação é chamada de condição de corrida. A integridade dos estados dos objetos deve ser mantida, e para evitar o acesso simultaneo, deve-se sincronizar o acesso à estes métodos. Baseado nos monitores, toda operação que não deva ser interrompida deve ser declarada como SYNCHRONIZED.

Abaixo há um exemplo de um código que sofre problema de sincronização. O método contar() será executado diversas vezes e em intervalos ligeiramente diferentes, podendo ocorrer em um erro do valor que esta sendo resgatado por uma outra Thread. Neste caso há o uso do *synchronized* que informa ao Java para tornar o bloco de código à prova de conflitos entre Threads. Apenas uma thread de cada vez será permitida dentro deste método, e as outras precisarão esperar até que a atual linha em execução termine de executar o método, para que outras Threads possam começar a executá-lo.

```
public class ContadorSeguro {
int ValorCrucial;
  public synchronized void contar(){
    ValorCrucial += 1;
  }
}
```

Neste caso, fica garantido que o método concluirá sua execução antes que outro possa executar. Se uma linha de execução chama *contar()* e outra linha de execução também chama, a segunda fica esperando.

Em contrapartida, a sincronização de um método extenso e demorado, é quase sempre uma idéia ruim. Todas as Threads acabariam sendo deixadas nesse gargalo, aguardando em uma fila simples pela sua vez, nesse método lento. Ainda neste contexto, caso tenhamos uma determinada operação que depende de outra, por exemplo, dentro de uma operação bancária, poderemos fazer transferência. Porém, somente poderemos fazer transferência de uma conta bancaria que tenha saldo. Assim, se o método de transferência for *synchronized*, e se não tiver saldo, esta Thread não sairá do método até que o mesmo tenha executado, mas para o mesmo executar ele precisará que outra Thread efetue o deposito, porém o objeto está “bloqueado” para a primeira Thread. Neste caso ocorre um Deadlock, ou seja, um ‘bloqueio mutuamente exclusivo’ visto que ninguém resolverá este imparse.

Para tal há uma solução que é a utilização dos métodos WAIT() e NOTIFY() dentro dos métodos declarados como SYNCHRONIZED.

O método WAIT() é utilizado quando se precisa esperar dentro de um método SYNCHRONIZED, ou seja, ele bloqueia a Thread corrente, que entra para uma lista de

espera. Para ser novamente ativada, alguma outra Thread deve chamar o método NOTIFY() ou NOTIFYALL().

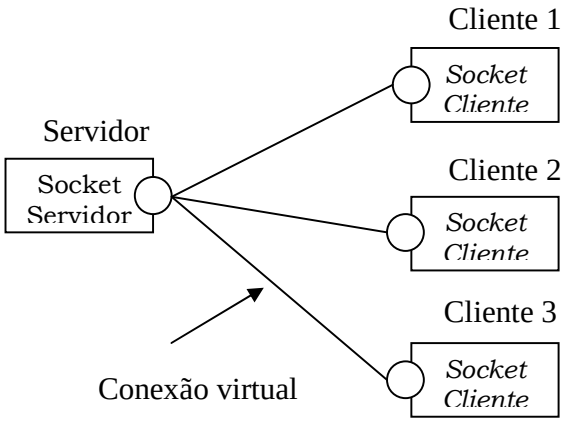
A diferença entre o NOTIFY e o NOTIFYALL é que o NOTIFY remove uma unica Thread da lista de espera, e o NOTIFYALL remove todas. O problema de trabalhar com o NOTIFY somente, é que pode-se desbloquear a Thread que não resolverá o problema, e isso também pode ocasionar em um problema de DEADLOCK.Neste caso, aconselha-se sempre a utilizar o NOTIFYALL, pois dará às Threads que estão esperando uma chance de verificar se as circunstancias mudaram.

Dentro do SWING, cada execução de um programa gera uma Thread, e a segunda Thread é criada para tratamento de eventos, ou seja, as chamadas ao método actionPerformed são executadas nesta Thread.

10. Sockets

A idéia de um socket faz parte do TCP/IP, o conjunto de protocolos usado pela Internet. Um socket é uma conexão de dados transparente entre dois computadores em uma rede. Ele é identificado pelo endereço de rede dos computadores, seus pontos finais e uma porta em cada computador. Os computadores em rede direcionam os streams de dados recebidos da rede para programas receptores específicos, associando cada programa a um número diferente, a porta do programa. Da mesma forma, quando o tráfego de saída é gerado, o programa de origem recebe um número de porta para a transação. Caso contrário, o computador remoto poderia não responder à entrada. Determinados números de portas são reservados no TCP/IP para protocolos específicos – por exemplo, 25 para o SMTP e 80 para o HTTP. Todos os número de porta abaixo do 1024 são reservados para o superusuário de cada computador.

Um servidor é um programa que fica executando em uma máquina, possui um nome (host) e este fica aguardando conexões de clientes em uma porta que é identificada por um número. Se criamos um programa cliente que deve se comunicar com um programa servidor, precisamos dizer ao nosso programa cliente em que máquina está sendo executado o servidor e em que porta ele está "escutando". Isso é feito através de sockets. Num mesmo instante, vários clientes podem estar solicitando a aplicação do servidor. Para cada um teremos um novo socket. Teremos um socket do lado do servidor e um socket em cada um dos clientes. Essa conexão não é física, podemos chamá-la de virtual ou lógica. Os pacotes continuarão circulando pela rede em caminhos diversos para depois serem reunidos numa mensagem.



A classe *Socket* do Java reúne todas estas informações em um único lugar e ainda fornece uma grande variedade de métodos para trabalharmos com o sockets. Todas as classes relacionadas a Sockets encontram-se no pacote "java.net".

Hierarquia

```
java.lang.Object
  |
  +--java.net.Socket
```

10.1 Classe ServerSocket

Quando se implementa sockets existe uma diferença entre o servidor e o cliente: o servidor precisa ficar "ouvindo" uma porta, até que chegue uma requisição de conexão por parte de um cliente. Quando a requisição chega, o servidor então estabelece conexão.

A classe ServerSocket implementa sockets no lado do servidor. Um socket servidor aguarda requisições que chegam pela rede. Ele executa algumas operações baseadas naquela requisição e então possivelmente retorna um resultado ao cliente.

Hierarquia

```
java.lang.Object
  |
  +--java.net.ServerSocket
```

Segue um exemplo de manipulação de Sockets no Servidor:

```
import java.net.*;
import java.io.*;

public class Servidor{
        public static  void main(String[] args) throws IOException{

        ServerSocket serverSocket = null;
        try {
          serverSocket = new ServerSocket(4445);
        }catch (IOException e){
                System.err.println("O servidor nao pode ouvir a porta");
                System.exit(1);
        }

        Socket clientSocket = null;
        try {
          System.out.println("Servidor esperando conexão!");
          clientSocket = serverSocket.accept();
        }catch (IOException e){
                System.err.println("A conexao falhou!");
                System.exit(1);
        }

        PrintWriter out = new PrintWriter ( clientSocket.getOutputStream(),true);
        BufferedReader in = new BufferedReader(
                        new InputStreamReader(clientSocket.getInputStream()));
                out.println(in.readLine());
        }
}
```

Inicialmente é criado o ServerSocket que irá ouvir a porta 3333. Após isso através do método accept() do SocketServidor, o servidor aguarda uma solicitação de conexão de um cliente. Quando isso acontece, um Socket é criado. Os objetos da classe PrintWriter e BufferedReader são utilizados para efetuar a comunicação.

10.2 Classe Socket

A classe Socket implementa o Socket no lado do Cliente. Ele é responsavel por estabelecer a conexão que esta à espera pelo Servidor através da porta 4445. Através deste canal estabelecido pode-se enviar dados e receber. Ainda se pode manipular multiplas conexões do Cliente para o Servidor, onde cada canal aberto Servidor podera tratar como uma

Thread independente e paralela. Assim, consegue-se obter maiores beneficios da manipulação dos Sockets.

Segue um exemplo de Socket no cliente. Estes dois programas permitem que se faça uma aplicação Echo, onde tudo que é enviado do Cliente para o Servidor, o Servidor responde a mesma coisa.

```
import java.io.*;
import java.net.*;

public class EchoClient {

   public static void main(String[] args) throws IOException {
      Socket echoSocket = null;
      PrintWriter out = null;
      BufferedReader in = null;

      try {
         echoSocket = new Socket("127.0.0.1", 4445);
         out = new PrintWriter(echoSocket.getOutputStream(), true);
         in = new BufferedReader(
                     new InputStreamReader(echoSocket.getInputStream()));
      } catch (UnknownHostException e) {
         System.err.println("Host desconhecido");
         System.exit(1);
      } catch (IOException e) {
         System.err.println("Nao foi possivel estabeler uma conexao.");
         System.exit(1);
      }

       for(int vc=0; vc<args.length; vc++) {
          out.println(args[vc]);
          System.out.println("echo: " + in.readLine());
       }

       out.close();
       in.close();
       echoSocket.close();
   }
}
```

11. Acesso a Banco de Dados - JDBC

A programação em bancos de dados tem se caracterizado por uma Torre de Babel. Este caso se exemplifica devido ao fato de existirem diversos produtos de banco de dados(BD) disponíveis no mercado sendo que cada um trabalha de acordo com uma linguagem diferenciada. Os produtos de BD não seguem completamente o padrão ANSI, criam métodos proprios com o intuito de melhorar a manipulação dos dados internos.

A API do Java, o JDBC, permite compartilhar uma única linguagem entre as diversas aplicações. Além disso, o JDBC - Java Database Connectivity - provê um conjunto de interfaces com o objetivo de criar um ponto em comum entre as aplicações e os mecanismos de acesso a um banco de dados.

Assim, juntamente com grandes líderes da área de banco de dados, a JavaSoft desenvolveu uma API simples para o acesso a base de dados - JDBC. Baseado neste processo, foi criado algumas metas de projeto tais como:

- JDBC deve ser uma API ao nível de SQL
- JDBC deve absorver as experiências em outras APIs de bases de dados.
- JDBC deve ser simples.

A versão utilizada atualmente é o JDBC2.0, apesar de já existir o JDBC 3.0. A partir da versão 2.0, novas características e suporte forem implementados, podemos citar: controle de atualizações em bloco, manipulação de registros através de programação, suporte a Scrollable Resultset, dentre outras características. O JDBC2.0 também suporta o SQL3, linguagem de consulta para banco de dados objeto-relacional.

Uma API ao nível do SQL permite construir sentenças SQL e colocá-las dentro das chamadas da API Java. Assim, JDBC permite uma transmutação transparente entre o mundo do Banco de Dados e as aplicações JAVA. Os resultados das bases de dados são retornados através de variáveis do Java e problemas com acessos são recebidos pelas exceções, que contém uma classe especial chamada de SQLException.

Devido aos problemas surgidos com a proliferação dos acessos as APIs dos bancos de dados proprietários, a idéia do acesso ao BD universal não é uma solução nova. De fato, a JavaSoft aproveitou todos os aspectos de uma API, a ODBC (Open DataBase Connectivity). O ODBC foi desenvolvido com o intuito de criar um padrão de acesso às bases de dados no ambiente Windows. Embora a indústria tenha aceitado o ODBC como um primeiro recurso, o mesmo é lento, deixando as aplicações com conexões aos bancos de dados dificeis de manipular.

Adicionando ao ODBC, o JDBC é fortemente influenciado pelas APIs existentes nos BD tais como X/Open SQL Call Level Interface. Assim, a JavaSoft se utilizou deste recurso e construiu uma simples API com o intuito de ser aceito pelas empresas de BD. Além disso, desenvolvendo uma API partindo dos produtos já existentes diminui o tempo de desenvolvimento.

Especificamente, a JavaSoft trabalhou em paralelo com o a Intersolv para criar uma ponte no ODBC que mapeasse as chamadas JDBC para o ODBC, permitindo que os acessos às aplicações Java sejam realizados por qualquer base de dados que suporte o ODBC.

Atualmente, há o que chamamos dos drivers nativos para Banco de Dados, ou seja, cada banco de dados possui classes Java que compoem o pacote JDBC especifico para cada Banco, onde nos, desenvolvedores, poderemos fazer o download e utilizar diretamente com o nosso BD de desenvolvimento. Por exemplo, se a aplicação que você esta desenvolvendo ela trabalha com o BD Oracle, há um driver JDBC nativo para BD Oracle. Você pode encontra-lo nas páginas da Oracle ou links nas páginas da Sun, e fazer o download do mesmo. Neste caso, provavelmente ele será um *.jar* ou *.zip* onde você poderá setar no *classpath* e manipular diretamente o seu banco de dados.

11.1 Versões do JDBC

A API JDBC possui três versões disponiveis. Elas foram assim classificadas de acordo com a disponibilidade de recursos que as mesmas manipulam e também devido às novas funcionalidades que foram incorporadas ao JDBC.

11.1.1 JDBC 1.0

O JDBC na versão 1.0 provê o básico para fornecer acesso aos dados. Ele é composto pelas principais interfaces e classes tais como Driver, DriverManager, Connection, Statement, PreparedStatement, CallableStatement e ResultSet.

11.1.2 JDBC 2.0

A partir do JDBC 2.0 a API foi dividida em duas partes: core API e um pacote adicional, que a partir da versão 1.4 do JDK já foi incorporado. Esta versão 2.0 adicionou algumas classes e funcionalidades, tais como:

- ✓ Scrollable Resultsets – permite a manipulação de *next()* e *prior()* dentro do ResultSet, o que antes so era possivel percorrer o resultado do inicio para o fim(método next()).
- ✓ Insert, delete e update através da programação – permite que façamos manipulações de inserção de registro dentro da propria aplicação, como acontece com linguagens como Delphi, VB.
- ✓ Suporte aos tipos de dados SQL-3 – permite que manipulemos dados vindos de Banco de Dados Objeto-Relacional, ou seja, que manipulemos os registros como objetos.

11.1.3 JDBC 3.0

Foram incluidas mais funcionalidades a partir do JDK 1.4.

11.2 Tipos de Drivers

Atualmente, há quatro tipos de drivers disponiveis para o JDBC. Estes tipos foram classificados devido a interação entre o banco de dados especifico e a aplicação Java. Eles permitem distinguir como será realizada a conexão entre a aplicação e o BD.

11.2.1 Driver JDBC Tipo 1

Este driver caracteriza a ponte JDBC-ODBC. Foi o primeiro tipo de driver desenvolvido e atualmente não é utilizado devido à presença dos drivers nativos. Também é caracterizado pela sua lentidão, visto que entre o BD e a aplicação Java temos duas camadas, sendo a primeiro o ODBC e a segunda o JDBC, como segue na ilustração:

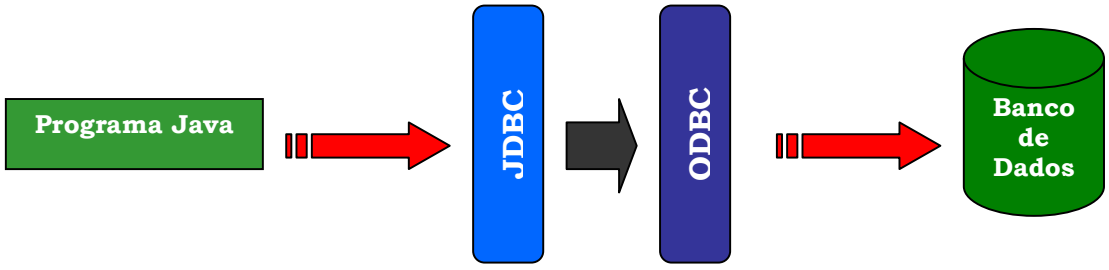


Figura 1. Exemplo do driver JDBC - ODBC

11.2.2 Driver JDBC Tipo 2

O driver do tipo 2 é caracterizado por possuir um software cliente do Banco de Dados em cada estação que possua a aplicação. Assim, por exemplo, se você esta desenvolvendo uma aplicação para Banco de Dados DB2, significa que quando for instalar nos seus 100 clientes, precisara antes instalar o cliente do DB2 em cada uma das 100 estações que vai colocar o seu aplicativo. Além disso, se a sua aplicação se tratasse de uma aplicação para a Web, cada vez que um cliente(usuario da Web) fosse executar a sua aplicação ele teria que baixar o cliente do BD. So para esclarecer, um cliente de um banco de dados é uma media de uns 100Mb, em se tratando por exemplo do DB2. Segue uma ilustração:

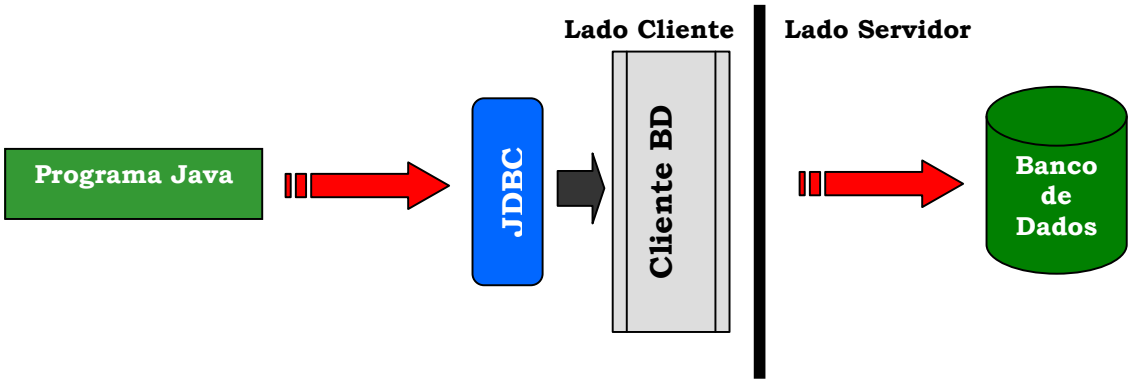


Figura 2. Exemplo do Driver Tipo 2

11.2.3 Driver JDBC Tipo 3

O driver tipo 3 é caracterizado por uma classe Java que pode ser carregada. Através desta classe Java, é criada uma porta para a comunicação com o Banco de Dados. Desta vez, o cliente do respectivo Banco de Dados estara instalado no lado do Servidor. Segue uma ilustração:

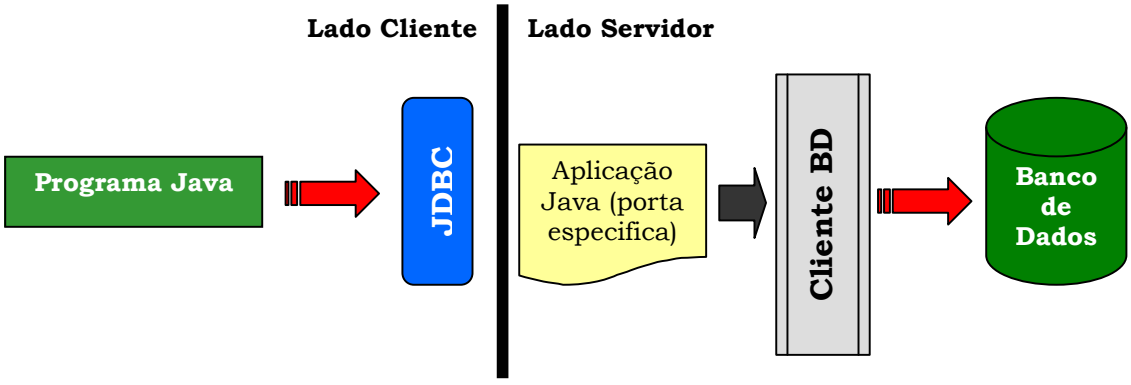


Figura 3. Exemplo do Driver tipo 3

11.2.4 Driver JDBC Tipo 4

O Driver tipo 4 é caracterizado como um driver puro Java, com protocolo nativo. Não necessita de Cliente do BD, o acesso é realizado diretamente ao banco de dados, é também chamado de Cliente Magro(Thin Client). Como segue a ilustração:

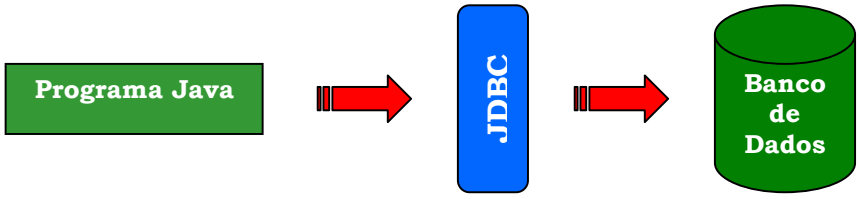


Figura 4. Exemplo do Tipo 4, driver nativo.

11.3 Classes do JDBC – Conexão com o BD

O JDBC realiza as suas tarefas implementando um conjunto de Interfaces, cada um deste desenvolvido por um fabricante distinto. O conjunto de classes que implementa estas interfaces é chamado de Driver do JDBC. No desenvolvimento de uma aplicação, não é necessário se preocupar com a construção destas classes, mas somente na sua utilização e implementação do código que será utilizado.

11.3.1 Driver

A aplicação que estiver sendo desenvolvida deve estar desprovida de detalhes contendo, estritamente, a implementação do Driver para esta base de dados. Assim, uma aplicação utiliza o JDBC como uma interface por onde trafegam todos os requerimentos feitos ao Banco. Através do método ***Class.forName(DriverClassName)*** é possivel dizer qual o tipo de Driver que se esta utilizando. O nome do Driver é definido por um tipo específico de URL, como segue um exemplo abaixo:

```
jdbc:<subprotocolo>:<subname>
```

onde podemos ter especificamente:

```
jdbc:odbc:JdbcOdbcDriver
```

O subprotocolo identifica qual o *Driver* que será utilizado e o subname identifica algum parâmetro necessário ao Driver.

11.3.2 DriverManager

A classe DriverManager obtém uma conexão utilizando a URL passada. O DriverManager seleciona o driver apropriado se existirem mais de um. Utilizando a URL da base de

dados, uma identificação de usuario, uma senha, e o nome da instancia da base de dados, a aplicação requisita uma conexão ao BD, como segue:

```
jdbc:odbc:cursoJDBC,"","
```

ou em MySQL

```
jdbc:mysql://localhost/JAVADB
```

Nome da instância do BD

11.3.3 java.sql.Connection

Esta classe *Connection* permite que se trabalhe com uma conexão do BD. A classe DriverManager devolve uma conexão, como segue:

```
Connection con = DriverManager.getConnection(url, userid, passwd);
```

Assim, para realizar a conexão, o JDBC utiliza de uma classe (java.sql.DriverManager) e duas interfaces (java.sql.Driver) e (java.sql.Connection). Enfim, a classe *Connection* representa uma transação de base de dados lógica. Esta classe é utilizada para enviar uma série de sentenças SQL para o banco de dados, gerenciando o *commit* ou *abort* destas sentenças.

```
import java.SQL.*;

try{
   Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
   Connection con = DriverManager.getConnection("jdbc:odbc:JavafichaDB", "", "");
}
catch (Exception e){
   System.out.println(e.toString());
   System.exit(0);
}

con.close();
```

Pacote do JDBC.

Nome do Driver do JDBC.

UserID

Caminho da Fonte de Dados.

Passwd

Fechando a conexão com o Banco para o envio das sentenças SQL.

11.4 Classes JDBC – Acesso ao BD

Uma vez inicializada a conexão através do objeto *Connection* é criada uma linha direta com a Base de Dados onde pode-se manipular as funções do SQL presentes na DML como alteração, ou simplesmente consultas. Assim, a utilização do objeto *Connection* permite gerar implementações da classe *java.sql.Statement* na mesma transação. Após a utilização das Sentenças do SQL é necessário realizar um *commit* (confirmação) ou *rollback* (cancelamento) dos objetos associados na classe *Connection*.

11.4.1 java.sql.Statement

A classe Statement permite manipular as sentenças onde serão inseridos os comandos SQL. Como citado anteriormente, o acesso a base de dados se inicializa com a conexão ao BD através dos objetos da classe *Connection*. Este objeto tem como objetivo armazenar implementações da classe java.sql.Statement em uma mesma transação. É importante se distinguir entre o tipo de sentença SQL se deseja utilizar, visto que o método de enviar

consultas(*query*) se difere do envio de atualizações(*update, insert, delete*) na base de dados.

```
Statement stmt = con.createStatement();
stmt.executeQuery("SELECT * FROM curso");
stmt.executeUpdate("DELETE FROM curso WHERE cpf=3333");
```

A principal diferença ocorre pois o método da consulta retorna uma instância da classe java.sql.*ResultSet* enquanto que a atualização retorna um número inteiro. Ainda há o método *Execute()* da classe Statement que é utilizado em situações onde não se tenham o conhecimento dos tipos das sentenças. Ele retorna TRUE, caso tenha afetado algum registro. E para obter os dados utiliza-se o método *getResultSet().*

11.4.2 java.sql.ResultSet

Um *ResultSet* é uma linha ou conjunto de dados resultante de uma Consulta (*Query*). Assim, através dela, nos poderemos manipular os resultados da nossa consulta à base de dados. Outra caracteristica desta classe, é que podemos armazenar neste conjnto de dados uma determinada coluna de um tipo de dados, mas mostrar este resultado como um tipo de dado diferente. Por exemplo, se eu tenho uma coluna chamada Data e no banco de dados ela esta armazenada como DateTime, eu posso retornar ela como uma String. Assim, a manipulação dos dados de retorno podem ser diferente do que os tipos armazenados.

A classe permite manipular os resultados de acordo com as colunas da base. Segue a estrutura:

Type get type (int | String)

Assim, a coluna pode ser referenciada por um numero ou o nome da coluna no BD, como segue:

```
ResultSet rs = stmt.executeQuery("SELECT * FROM curso");
rs.getString("nome");
rs.getString(2);
```

Neste caso a coluna 2 corresponde a coluna Nome.

Há ainda o método n*ext()* que permite navegar pelos registros de um *ResultSet*. Se o método *next()* retornar *TRUE* significa que há outra linha na seqüência, e se retornar *FALSE* significa que não falta nenhum registro no *ResultSet*.

11.4.3 java.sql.PreparedStatement

Esta classe é uma subinterface da classe *Statement*, e representa as sentenças que são enviadas ao Banco de Dados e são compiladas ou seja, preparadas antecipadamente. Com isso, elas ficam armazenadas em cache ou ele pode ser reutilizado, dependendo do BD, tornando a execução muito mais rápida. A classe *Statement* requer uma compilação em toda execução. Os valores variaveis para as colunas são passados como parâmatros através da cláusula IN, como segue um exemplo:

```
pstmt = con.prepareStatement("UPDATE pessoa SET nome=?");
```

Para setar o valor dos parametro nome, segue o seguinte:

```
pstmt.setString(1,"Bosco");
pstmt.executeUpdate();
```

A classe *CallableStatement* é um procedimento para a chamada de "procedures" internas do banco de dados.

11.5 JDBC 2.0

O JDBC 2.0 trouxe algumas mudanças significativas para a manipulação dos dados armazenados no BD. Vamos relembrar aqui as caracteristicas que foram acrescidas no JDBC:

- ✓ Navegar para frente e para trás e mover para uma linha especifica
- ✓ Realizar atualizaçoes utilizando métodos na linguagem Java e não utilizando SQL
- ✓ Utilizar SQL3

11.5.1 Scrollable ResultSet

Através do Scrollable ResultSet nos conseguimos manipular os resultset para frente e para trás. Para tal precisamos de dois argumentos:

1. Tratamento do tipo de Scroll
    a. TYPE_FORWARD_ONLY – Antigamente utilizado
    b. TYPE_SCROLL_INSENSITIVE – Não ve as atualizaçoes no ResultSet se o mesmo estiver "aberto"
    c. TYPE_SCROLL_SENSITIVE – Vê as atualizações no ResultSet se o mesmo estiver "aberto"
2. Somente leitura ou atualizável
    a. CONCUR_READ_ONLY – Somente leitura
    b. CONCUR_UPDATABLE – Pode ser atualizado através de métodos do Java

Segue um exemplo:

```
Statement stmt = con.createStatement( ResultSet.TYPE_SCROLL_INSENSITIVE,
ResultSet.CONCUR_READ_ONLY);
```

Os métodos disponiveis são: next(), previous() e getRow(). Segue um exemplo de uma aplicação:

```
if (ae.getSource()== btnConsulta){
        System.out.println("Realizando uma consulta...");
        stmt= con.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
ResultSet.CONCUR_READ_ONLY);
        rs = stmt.executeQuery(MetodoSQL.mConsultar());
        if (rs.last()){
                int nQtde = rs.getRow();
                lbQuantidade.setText("Quantidade: "+nQtde);
                rs.beforeFirst();
        }
        if (rs.next()){
                edCPF.setText(rs.getString("CPF"));
                edNome.setText(rs.getString("nome"));
                edCidade.setText(rs.getString("cidade"));
        }
}
```

11.5.2 Atualizações de Dados via Programação

As atualizações de dados normalmente são feitas utilizando a propria sentença SQL, como:

```
stmt.executeUpdate("UPDATE curso SET nome='Daniela'");
```

Através da programação em Java nos utilizamos o método *updateString* da classe ResultSet e passamos como parâmetros o nome do campo a ser atualizado e o valor. Depois disso, nos mandamos executar o método *updateRow*(), também da classe ResultSet. Segue um exemplo abaixo:

```
if (edCPF.getText().trim().equals("")|| edNome.getText().trim().equals("")||
    edCidade.getText().trim().equals("")){
        System.out.println("Os campos devem estar preenchidos");
} else {
        System.out.println("Campos OK - Realizar a inserção");
        stmt = con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
ResultSet.CONCUR_UPDATABLE);
        uprs = stmt.executeQuery("SELECT cpf,nome,cidade FROM PESSOA");
        uprs.moveToInsertRow();
        uprs.updateString("cpf",edCPF.getText());
        uprs.updateString("nome", edNome.getText());
        uprs.updateString("cidade", edCidade.getText());
        uprs.insertRow();
}
```

11.5.3 Consultas com o SQL 3

As consultas com a manipulação do SQL 3 não trazem inovações em relação à manipulação dos dados no Java, mas através do JDBC 2.0 é possivel que consultemos e façamos atualizaçoes em bancos de Dados Objetos relacionis que possuem estruturas de criação de tabelas e manipulação de dados diferente, como segue um exemplo abaixo:

```
CREATE TABLE Usuario OF objUsuario (UsuarioID PRIMARY KEY);
```

E a manipulação deste tipo de tabela ocorre como se estivessemos referenciando objetos, como segue:

```
SELECT Emp.UsuarioID.Nome as Nome, Emp.LivroID.Titulo as Titulo
FROM Emprestimo Emp;
```

Assim, o JDBC 2.0 permite manipularmos BDOR.

12. Referências Bibliográficas

Links

- ✓ SUN http://java.sun.com
- ✓ API On-line http://java.sun.com/api
- ✓ TheServerSide www.theserverside.com
- ✓ MundoOO www.mundooo.com.br

Livros

- ✓ HORSTMANN, Cay S., CORNELL, Gary; Core Java 2 – Fundamentals, 5ª Edição, 2003. Sun Press
- ✓ DEITEL & DEITEL; Java – Como Programar, 4ª Edição; Editora Bookman.
- ✓ Java Essential for C and C++ Programmers, Addison Wesley,1996
- ✓ CAMARA, Fabio; Orientação a objetos com .NET, 2002

13.Bibliografia dos Autores

Daniela Barreiro Claro é atualmente pesquisadora no Departamento de Ciência da Computação da Universidade Federal da Bahia, onde desenvolve pesquisas em Banco de Dados e Composições de Serviços Web.

Homepage Daniela Barreiro Claro: http://lattes.cnpq.br/9217378047217370
Email: danielabarreiroclaro@gmail.com

João Bosco Mangueira Sobral é professor associado II da Universidade Federal de Santa Catarina onde desenvolve pesquisas nas áreas de Segurança de Redes, Computação Ubíqua e Móvel e Auto-Organização em redes.

Homepage João Bosco Sobral: http://www.inf.ufsc.br/~bosco
Email: bosco@inf.ufsc.br