



Apostila de Introdução ao Java

Prof. Jaqson Dalbosco

jaqson@upf.br

Prof. Willingthon Pavan

pavan@upf.br

SUMÁRIO

| | |
|---|----|
| 1. INTRODUÇÃO AO USO DA LINGUAGEM JAVA | 4 |
| 1.1. O que é Java? | 4 |
| 1.2. História da Tecnologia Java..... | 4 |
| 1.3. O Java pertence a quem?..... | 4 |
| 1.4. Vantagens | 4 |
| 1.5. Características..... | 5 |
| 1.6. Versões..... | 5 |
| 1.7. A Arquitetura Geral da Plataforma Java | 5 |
| 1.7.1. JDK, JVM e JRE..... | 6 |
| 1.8. As Plataformas do Java | 6 |
| 1.8.1. Java SE - Java Standard Edition | 6 |
| 1.8.2. Java EE - Java Enterprise Edition | 7 |
| 1.8.3. Java ME - Java Micro Edition | 8 |
| 1.9. Possibilidades para Desenvolvimento com Java..... | 8 |
| 1.10. Um pouco mais sobre Java | 9 |
| 1.10.1. Passos Básicos para Geração, Compilação e Execução..... | 9 |
| 1.10.2. Os Bytecodes Gerados | 9 |
| 1.10.3. Código Independente de Plataforma – Máquinas Virtuais | 10 |
| 1.10.4. Funcionalidades da Máquina Virtual Java | 10 |
| 2. INSTALAÇÃO DO JDK DA SUN | 11 |
| 2.1. Algumas ferramentas do diretório bin | 11 |
| 2.2. Variáveis de Ambiente..... | 11 |
| 3. PRIMEIROS PROGRAMAS EM JAVA | 12 |
| 3.1. Criando o Primeiro Programa em Java | 12 |
| 3.2. Programa Bem Vindo | 12 |
| 3.3. Programa Bem Vindo 2 | 12 |
| 4. AMBIENTES DE DESENVOLVIMENTO PARA JAVA | 13 |
| 4.1. JCcreator | 13 |
| 4.2. Eclipse..... | 14 |
| 4.3. NetBeans | 15 |
| 4.4. Egen Developer..... | 16 |
| 5. SINTAXE DO JAVA | 17 |
| 5.1. Variáveis | 17 |
| 5.1.1. Variáveis de Tipo Primitivo..... | 17 |
| 5.1.2. Variáveis Compostas / Referência | 18 |
| 5.2. Declarando Variáveis..... | 19 |
| 5.3. Atribuições a variáveis..... | 19 |
| 5.4. Conversões de Tipos por Cast..... | 19 |
| 5.5. Comentários | 20 |
| 5.6. Caracteres especiais | 20 |
| 5.7. Operadores | 21 |
| 5.7.1. Operadores Aritméticos | 21 |
| 5.8. Mais sobre atribuições | 21 |
| 5.9. Incrementos e decrementos..... | 22 |
| 5.10. Comparações | 22 |
| 5.11. Operadores de comparação | 22 |
| 5.12. Operadores lógicos..... | 22 |

| | |
|--|----|
| 5.13. Operadores de Atribuição | 22 |
| 5.14. Expressões..... | 22 |
| 5.15. Blocos de Código..... | 23 |
| 5.16. Controle de Fluxo | 23 |
| 5.17. Tomada de Decisões | 24 |
| 5.17.1. Estrutura de seleção if..... | 24 |
| 5.17.2. Estrutura de seleção switch..... | 24 |
| 5.18. Estruturas de Repetição..... | 25 |
| 5.18.1. while..... | 25 |
| 5.18.2. for..... | 25 |
| 5.18.3. do/while; | 25 |
| 5.19. Instruções break e continue;..... | 25 |
| 5.20. Instruções rotuladas | 26 |
| 5.21. Exercício 1 - Fatorial | 26 |
| 5.22. Exercício 2 - Fibonacci | 26 |
| 6. CAIXAS DE DIÁLOGO | 27 |
| 6.1. A classe JOptionPane..... | 27 |
| 6.1.1. Mensagens de Alerta..... | 27 |
| 6.1.2. Caixa de Confirmação..... | 28 |
| 6.1.3. Caixa de Opções | 29 |
| 6.1.4. Caixa de Entrada de Dados | 29 |
| 7. COVENÇÕES PARA NOMES DE CLASSES E VARIÁVEIS | 30 |
| 8. A CLASSE STRING | 31 |
| 8.1. Comprimento da String..... | 31 |
| 8.2. Concatenação | 31 |
| 8.3. Comparação | 31 |
| 8.4. Conversão de Dados Numéricos para String | 32 |
| 8.5. Conversão de String para Dados Numéricos | 32 |
| 8.6. Formatação de String | 33 |
| 8.7. Decomposição de Strings..... | 33 |
| 8.8. Mais Exemplos da Classe String..... | 34 |
| 9. DATA E HORA - CLASSE DATE..... | 36 |
| 9.1.1. A Classe Date..... | 36 |
| 9.1.2. A Classe SimpleDateFormat..... | 36 |
| 9.2. Convertendo Date para String..... | 36 |
| 9.3. Convertendo String para Date..... | 37 |
| 10. ARRAYS E MATRIZES | 38 |
| 10.1. Array | 38 |
| 10.1.1. Declarar e Criar um Array | 38 |
| 10.1.2. Inicialização de um Array com Tipos Primitivos | 38 |
| 10.1.3. Inicialização de um Array com Objetos..... | 38 |
| 10.1.4. Comprimento de um Array | 39 |
| 10.1.5. Mais Exemplos de Array | 39 |
| 10.2. Matrizes..... | 40 |
| 10.2.1. Mais Exemplos de Matrizes..... | 41 |
| 10.3. Passagem por Referência | 41 |
| 10.4. Exercício sobre Array | 42 |
| 11. COLEÇÕES | 43 |
| 11.1. HashSet | 44 |
| 11.2. ArrayList | 44 |
| 11.3. Vector..... | 44 |
| 11.4. Iterator..... | 44 |

| | |
|---|----|
| 11.5. Métodos para Manipular Coleções | 45 |
| 11.6. Coleções Tipadas – Uso de Generics | 45 |
| 11.7. Exercício ArrayList..... | 46 |
| 11.8. Exercício usando coleção..... | 47 |
| 12. A CLASSE CALENDAR | 48 |
| 13. CONTROLE DE ERROS E TRATAMENTO DE EXCEÇÕES | 50 |
| 13.1. Estrutura Básica do controle de exceção | 50 |
| 13.2. Exemplo de uma estrutura básica | 50 |
| 13.3. Classes Throwable e Exception | 51 |
| 13.4. Estrutura Básica do controle de exceção com finally | 51 |
| 13.5. Exemplo com uso do finally | 51 |
| 13.6. Captura de exceções com múltiplos blocos catch..... | 52 |
| 13.7. Lançamento de exceções definidas pelo usuário | 52 |
| 13.7.1. Captura de exceções definidas pelo usuário | 52 |
| 13.7.2. Exemplo com exceção definida pelo usuário..... | 53 |
| 14. PROGRAMAÇÃO ORIENTADA A OBJETOS NO JAVA | 54 |
| 14.1. Pacotes | 54 |
| 14.2. Classes..... | 55 |
| 14.2.1. Atributos / Variáveis de Instância..... | 55 |
| 14.2.2. Métodos..... | 56 |
| 14.2.3. Método Construtor | 57 |
| 14.3. Objetos | 57 |
| 14.4. Escopo de Variáveis, Atributos e Métodos..... | 59 |
| 14.4.1. Escopo de Variável | 59 |
| 14.4.2. Escopo de Atributos..... | 59 |
| 14.4.3. Escopo de Métodos | 60 |
| 14.5. Ecapsulamento | 61 |
| 14.6. Sobrecarga de Métodos..... | 62 |
| 14.7. Herança | 63 |
| 14.7.1. Substituição / Sobrescrição | 64 |
| 14.7.2. Especificação super..... | 64 |
| 14.7.3. Especificação this..... | 64 |
| 14.8. Agregação | 65 |
| 14.9. Composição..... | 65 |
| 14.10. Polimorfismo..... | 66 |
| 14.11. O Operador instanceof | 66 |
| 14.12. Interfaces e Classes Abstratas | 67 |
| 14.12.1. Interface | 67 |
| 14.12.2. Classes Abstratas | 68 |
| 14.13. Java Beans..... | 68 |
| 14.14. Recursividade..... | 69 |
| 14.15. Serialização de Objetos..... | 70 |
| 14.16. Cast de Objetos | 71 |
| 15. DISTRIBUINDO A APLICAÇÃO JAVA | 73 |
| 16. COMO USAR A DOCUMENTAÇÃO DO JAVA | 75 |
| 17. REFERÊNCIAS BIBLIOGRÁFICAS..... | 76 |

1. INTRODUÇÃO AO USO DA LINGUAGEM JAVA

Neste material são apresentados os conhecimentos básicos sobre a linguagem Java, suas bibliotecas e ferramentas, visando proporcionar o domínio da sintaxe da linguagem, padrões de codificação, orientação a objetos e algumas das principais classes do Java em sua versão atual.

1.1. O que é Java?

Java é uma linguagem de programação de distribuição gratuita, capaz de produzir softwares robustos e multiplataforma, que podem rodar em diversos tipos de microcomputadores e dispositivos como celulares, palmtops, PDAs, entre outros. Segundo(Silveira, 2003), “Java é uma linguagem de programação orientada a objetos, desenvolvida por uma pequena equipe de pessoas na Sun Microsystems”. A tecnologia Java é composta por uma gama de produtos, baseados no poder da rede e na idéia de que um software deve ser capaz de rodar em diferentes máquinas, sistemas e dispositivos. Por diferentes dispositivos entendemos: computadores, servidores, notebooks, handhelds, PDAs (Palm), celulares e tudo mais o que for possível.

1.2. História da Tecnologia Java

A linguagem Java começou a ser desenvolvida na Sun Microsystems, em 1991, como parte do projeto Green, seus pesquisadores estavam trabalhando em uma linguagem de programação para dispositivos inteligentes.

O primeiro nome dado a esta nova linguagem pelo seu criador James Gosling foi Oak que quer dizer *carvalho*, uma árvore que ele podia observar quando olhava pela janela de seu escritório na Sun. Em seguida descobriu-se que já havia uma linguagem chamada Oak. Quando uma equipe da Sun visitou uma cafeteria local, o nome Java (cidade de origem de um café importado) foi sugerido e aceito. Segundo (Hubbard, 2006, p15), ”Desde que a World Wide Web (Web) surgiu na Internet em 1993, a linguagem vem sofrendo melhorias para facilitar a programação para a Web“. A tecnologia Java foi lançada em 1995, e desde então tem crescido em popularidade, se tornado uma plataforma muito estável e madura.

1.3. O Java pertence a quem?

Apesar de a tecnologia ter sido criada pela Sun Microsystems, ela é mantida pelo JCP (Java Community Process), que estabelece um processo evolutivo da linguagem, coletando opiniões juntamente com os seus colaboradores como empresas, universidades e desenvolvedores.

1.4. Vantagens

A seguir são relacionadas algumas das vantagens da linguagem Java:

- Independente de sistema operacional, banco de dados, servidor Web, IDE's, etc. Isso quer dizer que, podemos programar com qualquer tipo de combinação;
- Multiplataforma – totalmente portátil, ou seja, escreva uma vez e rode muitas;
- Possibilita a criação de programas para diferentes ambientes computacionais como: PC's, Celulares, PDA's, Robots, Risc, Palmtops, etc;
- Uma linguagem moderna, orientada a objetos, segura, moderna e altamente preparada para rodar em rede;
- Proporciona uma forma mais eficiente de distribuição de software;
- Recursos utilizados para Internet e Intranet;
- Diversos Frameworks e IDE's de desenvolvimento para aumento da produtividade;
- Alto índice de adoção pelas Universidades.

1.5. Características

Dentre as muitas características que o Java possui podemos citar algumas:

- Semelhante às linguagens C e C++, porém o Java é mais evoluído que o C e sem a complexidade do C++;
- Controle de exceções (em alguns casos obrigatório);
- Coleta de lixo automática. Dispensa liberação de memória;
- Há verificação de limite de vetores e referências vazias pelo compilador;
- Bytecode (arquivos compilados do Java) contém informações que permitem a verificação da integridade do código (+segurança).

1.6. Versões

Com a evolução do Java verificaram-se alguns avanços da tecnologia no decorrer dos anos e as correções feitas em algumas das versões:

- Maio/1995 – Lançamento da Tecnologia Java;
- Janeiro/1996 – Lançamento do JDK 1.0 Bugfixes: 1.01, 1.0.2;
- Dezembro/1996 – Lançamento do JDK1.1 Bugfixes: 1.1.1, 1.1.2, 1.1.3, 1.1.4, 1.1.5, ...;
- Março/1997 – Lançamento do Java Web Server beta e Java Servlet Developers Kit;
- Março/1998 – Lançamento do JFC(Java Foundation Classes) - Projeto Swing;
- Dezembro/1998 – Formalização do Java Community Process (JCP);
- Junho/1999 – Lançamento do Java Server Pages(JSP);
- Junho/1999 – Sun anuncia três edições da plataforma Java: J2SE, J2EE e J2ME;
- Agosto/1999 – Lançamento do J2SE 1.3 beta;
- Setembro/1999 – Lançamento do J2EE beta;
- Junho/2001 – Lançamento do JDK1.4 Bugfixes : 1.4.1, 1.4.2, 1.4.3, 1.4.4, 1.4.5, ...;
- 2004 – Lançamento do Java 5 – JDK 1.5;
- 2006 – Lançamento do Java 6 – JDK 1.6;
- 2011 – Lançamento do Java 7 – JDK 7.

1.7. A Arquitetura Geral da Plataforma Java

A arquitetura geral da plataforma Java é formada basicamente pelo JDK(Java Development Kit) e JRE(Java Runtime Environment). O JDK é composto por compilador, Java debugger, responsável por facilitar a identificação de erros, entre outras ferramentas. A JRE é um componente da arquitetura Java, em que está incluída a JVM(Java Virtual Machine), as API's que possibilitam a execução dos programas, além de rodar em qualquer sistema operacional.

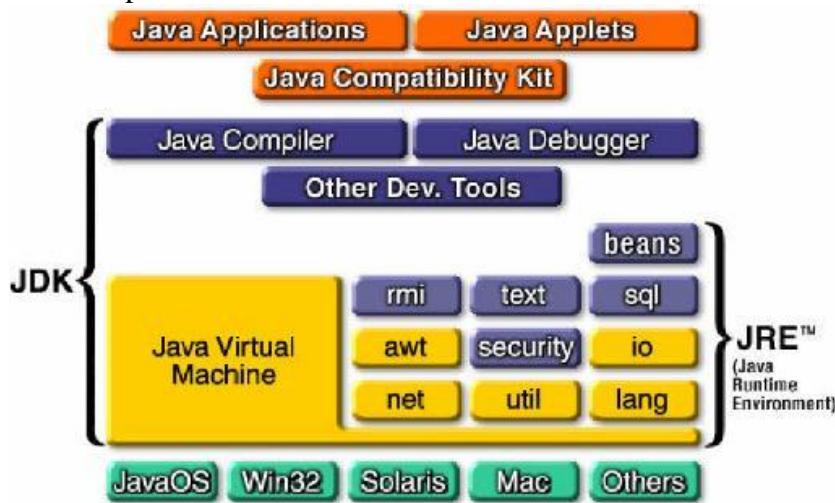


Figura 1 - Arquitetura Geral do Java

1.7.1. JDK, JVM e JRE

Para desenvolver e rodar aplicações com o Java, algumas ferramentas devem estar instaladas na máquina. Dentre estas ferramentas, destacam-se as três citadas a seguir:

JDK: O Java Development Kit (JDK) é o conjunto básico de ferramentas para o desenvolvedor Java. Fazem parte do JDK ferramentas importantes como o javac (compilador), java (a máquina virtual), javadoc (gerador automático de documentação), jdb (Java debugger), javap (decompilador) e diversas outras ferramentas importantes.

JVM: A Máquina Virtual Java (do inglês Java Virtual Machine - JVM) é um programa que carrega e executa os aplicativos Java, convertendo os bytecodes em código executável de máquina. A JVM é responsável pelo gerenciamento dos aplicativos, à medida que são executados. Graças à JVM, os programas escritos em Java podem funcionar em qualquer plataforma de hardware e software que possua uma versão da JVM, tornando assim essas aplicações independentes da plataforma onde funcionam.

JRE: O Java Runtime Environment (JRE) é basicamente composto da Máquina Virtual Java (JVM) e o conjunto de bibliotecas, ou seja, tudo o que você precisa para executar aplicações Java. O JRE pode ser distribuído a vontade, podendo inclusive ser embutido nas aplicações Java.

1.8. As Plataformas do Java

A tecnologia Java está organizada em três plataformas definidas de acordo com os objetivos específicos, como mostra a figura a seguir:

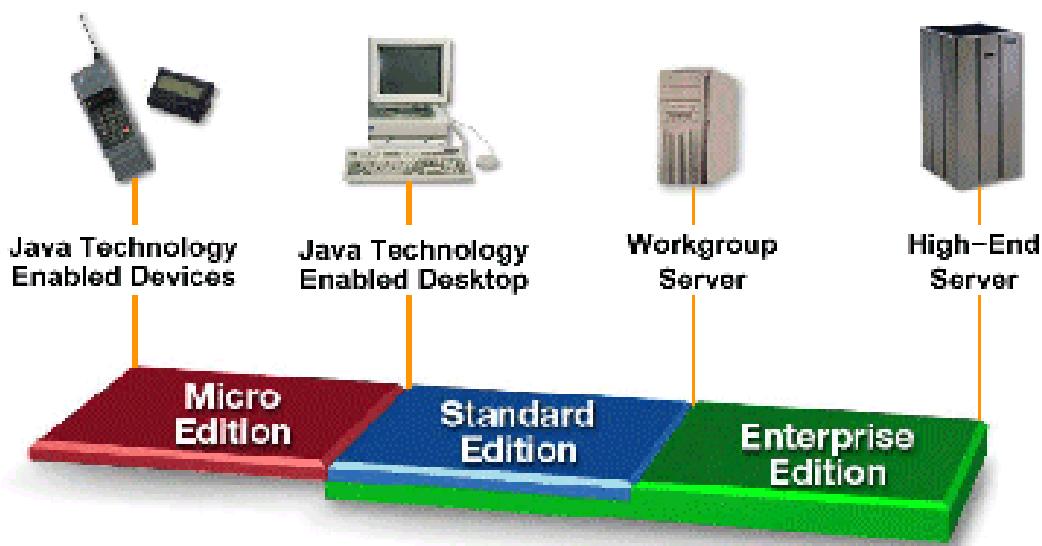


Figura 2 - Plataformas do Java

1.8.1. Java SE - Java Standard Edition

A Sun distribui o Java SE na forma de um SDK (Software Development Kit), em conjunto com uma JRE (Java Runtime Environment). O pacote do SDK da SE vem com ferramentas para: compilação, debugging, geração de documentação (javadoc), empacotador de componentes (jar) e a JRE, que contém a JVM e outros componentes necessários para rodar aplicações Java. A figura 2 mostra como é composta a plataforma Java SE.

| Java Language | | | | | | | | | |
|------------------------------|-----------------|-------|--------------------|----------------|-----------------------|------------|---------------------|---------------|---------------------|
| Tools & Tool APIs | java | javac | javadoc | apt | jar | javap | JPDA | jconsole | |
| | Security | Int'l | RMI | IDL | Deploy | Monitoring | Troubleshoot | Scripting | JVM TI |
| Deployment Technologies | Deployment | | | Java Web Start | | | | Java Plug-in | |
| User Interface Toolkits | AWT | | | Swing | | | | Java 2D | |
| JDK | Accessibility | | Drag n Drop | | Input Methods | | Image I/O | Print Service | Sound |
| | IDL | JDBC™ | | JNDI™ | | RMI | RMI-IIOP | | Scripting |
| JRE | Beans | | Intl Support | | I/O | | JMX | | Math |
| | Networking | | Override Mechanism | | Security | | Serialization | | Extension Mechanism |
| lang and util Base Libraries | lang and util | | Collections | | Concurrency Utilities | | JAR | | Logging |
| | Preferences API | | Ref Objects | | Reflection | | Regular Expressions | | Management |
| Java Virtual Machine | | | | | | | | | |
| Solaris™ | | Linux | | Windows | | Other | | | |

Figura 3 - Java Standard Edition

1.8.2. Java EE - Java Enterprise Edition

Java EE (Java Enterprise Edition) é uma plataforma de programação de computadores para desenvolver aplicações corporativas. A plataforma é formada por um conjunto de especificações que permitem o desenvolvimento de aplicações multi-camadas robustas, extensíveis e seguras, baseadas em componentes que são executados em um servidor de aplicações.

Construída sobre a base sólida do Java SE, proporciona um ambiente favorável para a construção de aplicações distribuídas, web services e aplicações Web.

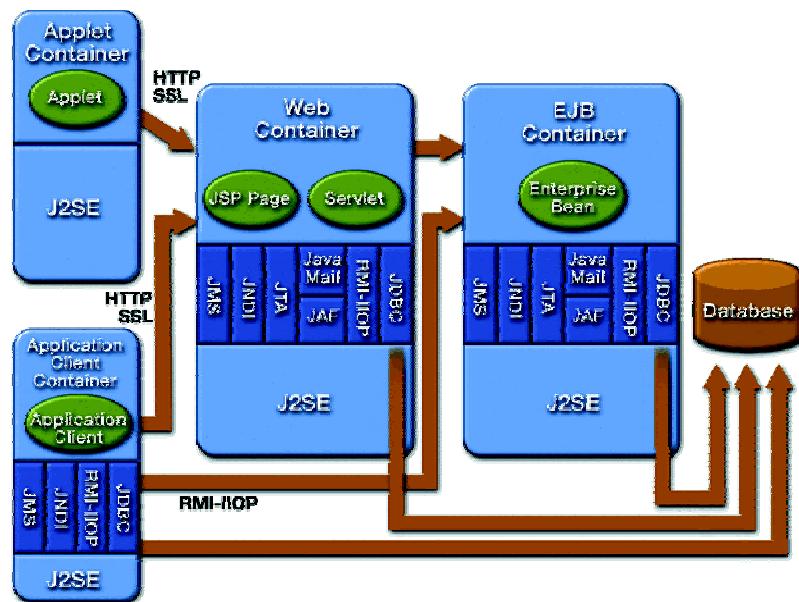


Figura 4 – Java Enterprise Edition

A plataforma J2EE, é uma plataforma completa, robusta, estável, segura e de alta performance, voltada para o desenvolvimento de soluções corporativas.

A plataforma J2EE é composta por muitas especificações entre elas:

EJB: permitem desenvolver a lógica de negócios e a camada de persistência de uma aplicação J2EE;

Servlets e JSP: para implementar interfaces web;

JNDI: acessar serviços de diretórios;

JavaMail: enviar e receber e-mail e mensagens JMS;

RMI: Invocação de método remoto.

1.8.3. Java ME - Java Micro Edition

A plataforma Java ME é voltada para aplicações que rodam em pequenos dispositivos como celulares, PDAs, smart cards, etc. Possui uma API bastante completa para o desenvolvimento de aplicações para dispositivos de pequeno porte.

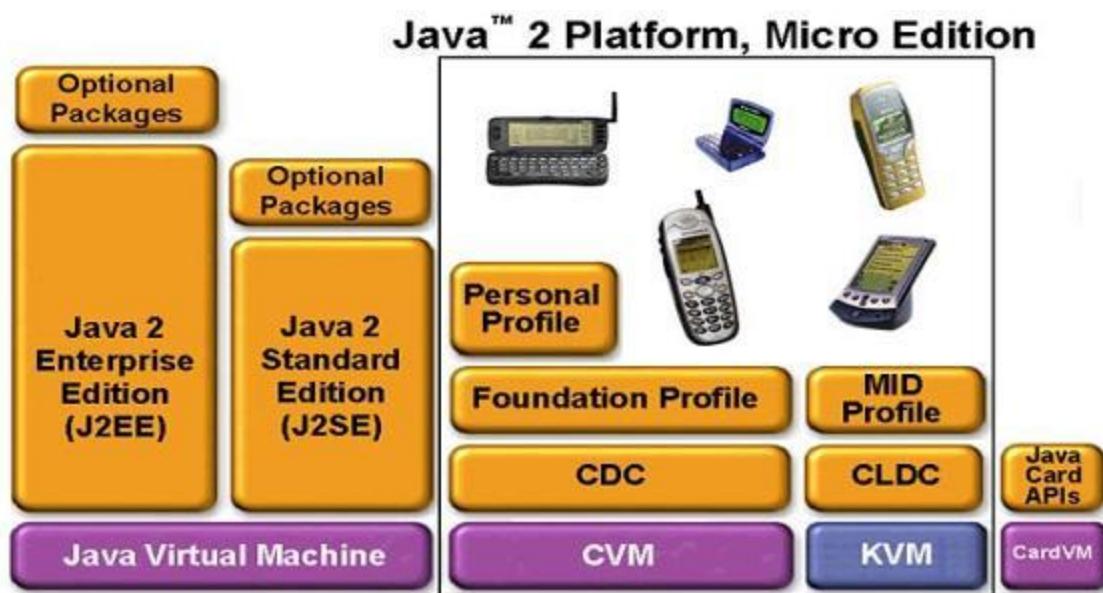


Figura 5 - Java 2 Micro Edition

1.9. Possibilidades para Desenvolvimento com Java

São diversas as aplicações que podem ser desenvolvidas com o uso da tecnologia Java. A seguir são descritos alguns dos tipos de aplicação que normalmente são desenvolvidas com essa tecnologia:

- Aplicativos gráficos;
- Soluções Web;
- Componentes de processamento de dados de larga escala de uso;
- Enterprise Application Integration (E.A.I.);
- Pequenos aplicativos para celulares, palmtops, cartões inteligentes, robots, entre outros;
- Desenvolvimento de jogos 3D;
- Tratamento de imagens;
- Grid computing;
- Código para banco de dados (Store Procedure EX: Oracle).

1.10. Um pouco mais sobre Java

1.10.1. Passos Básicos para Geração, Compilação e Execução

Para criar um programa em Java alguns passos básicos são necessários serem seguidos:

- Usa-se um editor qualquer para criar um arquivo com a extensão .java;
- Criar o programa dentro do arquivo com a extensão .java;
- Executa-se o compilador para gerar o arquivo compilado .class (chamados bytecodes);
- Utiliza-se um interpretador (JVM) para executar os bytecodes

A figura a seguir ilustra o fluxo completo para criar, compilar e executar um programa.

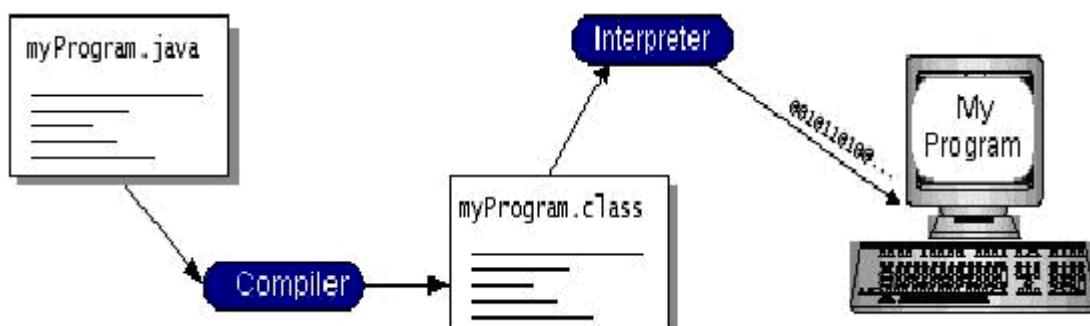


Figura 6 - Criar, compilar e executar

1.10.2. Os Bytecodes Gerados

Todos os compiladores Java geram bytecodes. Os bytecodes gerados pelo compilador Java podem ser executados em qualquer SO (Sistema Operacional) que tenha a JVM instalada. A maioria dos browsers Web também implementam a JVM para rodar applets Java. Logo, os bytecodes são altamente portáveis, podendo rodar em qualquer SO, e também em browsers web.

Na figura a seguir é ilustrado um pequeno comparativo entre um programa na linguagem C e na linguagem Java, demonstrando que ao escrever um programa na linguagem Java, para uma plataforma, o mesmo poderá rodar em qualquer outra, sem mudar uma linha de código, bastando para isso, ter instalado a JVM (Java Virtual Machine) no SO utilizado.

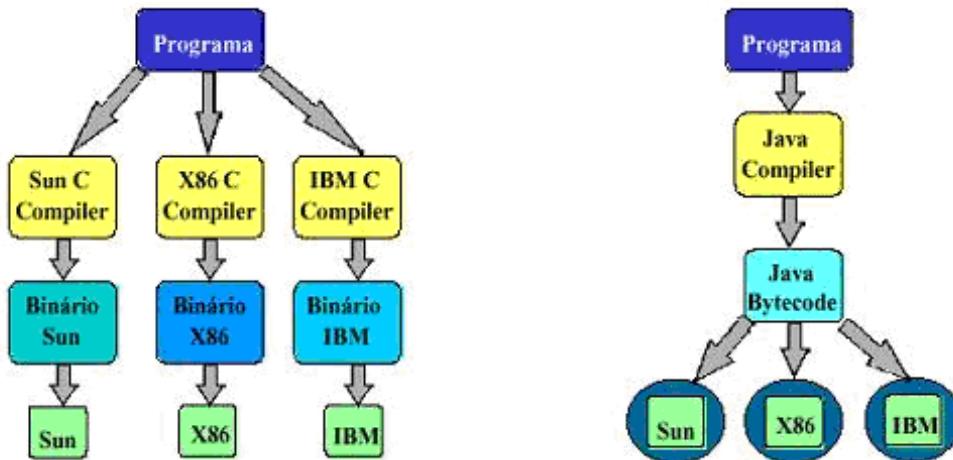


Figura 7 - Comparativo C e Java

1.10.3. Código Independente de Plataforma – Máquinas Virtuais

As Máquinas Virtuais (JVM) são responsáveis por prover a flexibilidade de sistema operacional para um software Java. Os programas Java na forma de bytecodes podem ser executados em diversos sistemas operacionais. O que muda em cada S.O. é a máquina virtual utilizada. A figura a seguir demonstra o caminho de um programa Java até a sua execução.

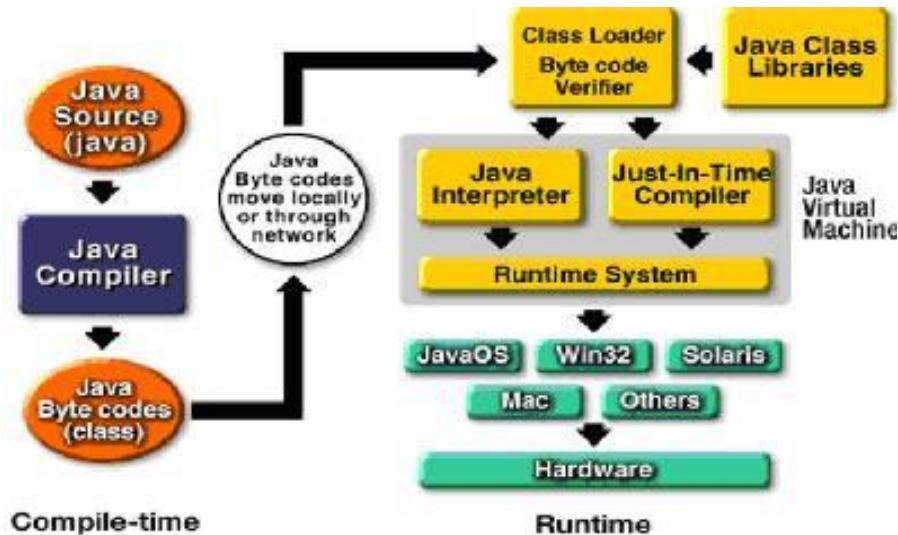


Figura 8 - Execução do Programa Java pela JVM

O código fonte passa pelo compilador do Java para que seja transformado em bytecode. Ao ser executado, passa pelas verificações das classes, é interpretado e compilado em tempo real pela JVM para o código de máquina do SO em uso, para que assim possa rodar em qualquer Sistema Operacional devido a sua característica de ser multiplataforma, ou seja, o mesmo código que é escrito para um ambiente roda em vários.

1.10.4. Funcionalidades da Máquina Virtual Java

A JVM apresenta uma série de funcionalidades. A seguir são apresentadas algumas das principais funcionalidades que a JVM proporciona:

- Interpretação de código inteligente;
- Mantém código mais acessado permanentemente traduzido para código nativo;
- Gerencia alocação e desalocação de memória;
- Class loader - carrega arquivos .class para memória;
- Segurança do código – responsável por garantir a não execução de códigos maliciosos (ex: applets);
- Verifica integridade do arquivo .class - Bytecode verifier.

2. INSTALAÇÃO DO JDK DA SUN

Para criar um programa em Java é preciso ter instalado no mínimo o Kit de Desenvolvimento, que pode ser baixado do site da Sun(www.java.sun.com).

- Versão Standard Edition disponível para Windows:
 - jdk-6-windows-i586.exe
- Versão Standard Edition disponível para Linux:
 - jdk-6-linux-i586-rpm.bin
 - jdk-6-linux-i586.bin

Ao executar este programa no computador, serão instalados todos os pacotes da edição Standard do Java para desenvolvimento de aplicações em uma pasta jdk1.?.? (de acordo com a versão instalada). A pasta de instalação da JDK possui o seguinte conteúdo:

Bin-> Compilador, interpretador, documentador, etc.

Demo -> Aplicativos de Exemplo

Include -> Arquivos .h para integração código nativo

Jre -> Ambiente de Execução (Java Runtime Environment)

Lib -> APIs da linguagem

Sample -> Exemplos de servidores

HTTP/HTTPS, CORBA, Servlets, etc.

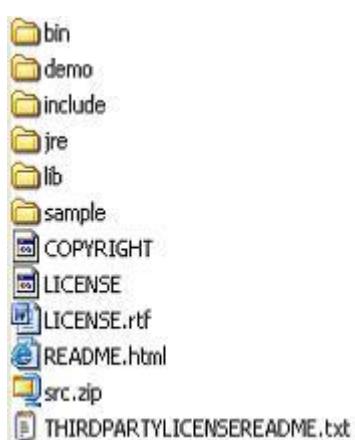


Figura 9 - Instalação JDK

2.1. Algumas ferramentas do diretório bin

- **appletviewer** (executa um applet)
- **java** alternativamente **jre** (executa uma aplicação (i.e. uma class com main))
- **javac** (compila um arquivo .java gerando um .class)
- **javadoc** (Gera documentação de arquivos .java)
- **javap** (disassembla classes - Para obter informações sobre a estrutura de uma classe use javap -c Classe)

2.2. Variáveis de Ambiente

Para facilitar o uso do compilador e interpretador, recomenda-se colocar o diretório C:\Java\jdk1.?.?\bin no PATH; Exemplo: **PATH=C:\Java\jdk1.6.0\bin**

Também é preciso criar uma variável de ambiente chamada **JAVA_HOME**, indicando o local de instalação do Kit; Exemplo: **JAVA_HOME=C:\Java\jdk1.6.0**

Esta variável é utilizada para aplicativos que dependem de Java poderem localizar a máquina virtual;

No Windows pode-se adicionar no AUTOEXEC ou no WinXP nas variáveis de ambiente acessíveis em “Meu Computador” -> “Propriedades” -> “Avançado” -> “Variáveis de Ambiente”. No Linux, essas variáveis podem ser adicionadas no profiles.

3. PRIMEIROS PROGRAMAS EM JAVA

A seguir, será apresentado, na prática, como os programas Java podem ser criados, compilados e executados.

3.1. Criando o Primeiro Programa em Java

1. Escrever o código em um arquivo texto com um editor;

```
public class programa1
{
    public static void main(string arg[])
    {
        System.out.println("Impresso pelo programa 1");
    }
}
```

Figura 10 - Primeiro Programa em Java

- 2) Salvar o arquivo com o nome "**programa1.java**" em uma pasta. EX: c:\java;



Figura 11 - Salvar o arquivo .java

3. Pelo prompt do DOS:

3.1. Compilar o código: **javac programa1.java** (gera o programa1.class)

3.2. Executar o aplicativo: **java programa1** (sem o .class)

```
C:\>javac programa1.java
C:\>java programa1
Impresso pelo programa 1
C:\>_
```

Figura 12 - Compilando e Executando

3.2. Programa Bem Vindo

```
public class BemVindo {
    public static void main( String args[ ] )  {
        System.out.println( "Bem vindo ao " );
        System.out.println( "mundo Java!" );
    }
}
```

3.3. Programa Bem Vindo 2

```
public class BemVindo2 {
    public static void main( String args[ ] )  {
        System.out.println( "Bem vindo\nao\nmundo\nJava!" );
    }
}
```

4. AMBIENTES DE DESENVOLVIMENTO PARA JAVA

Atualmente existem uma considerável quantidade de ambientes de desenvolvimento para Java como IDE's e ferramentas RAD (Rapid Application Development). Algumas delas são apresentadas a seguir.

4.1. JCreator

O Jcreator é um editor para a escrita do código Java. Permite editar códigos, tem funções de autocomplemento, proporciona a compilação e verificação de erros, além de ser um ambiente fácil de se trabalhar. O download pode ser efetuado na página <http://www.jcreator.com/download.htm>.

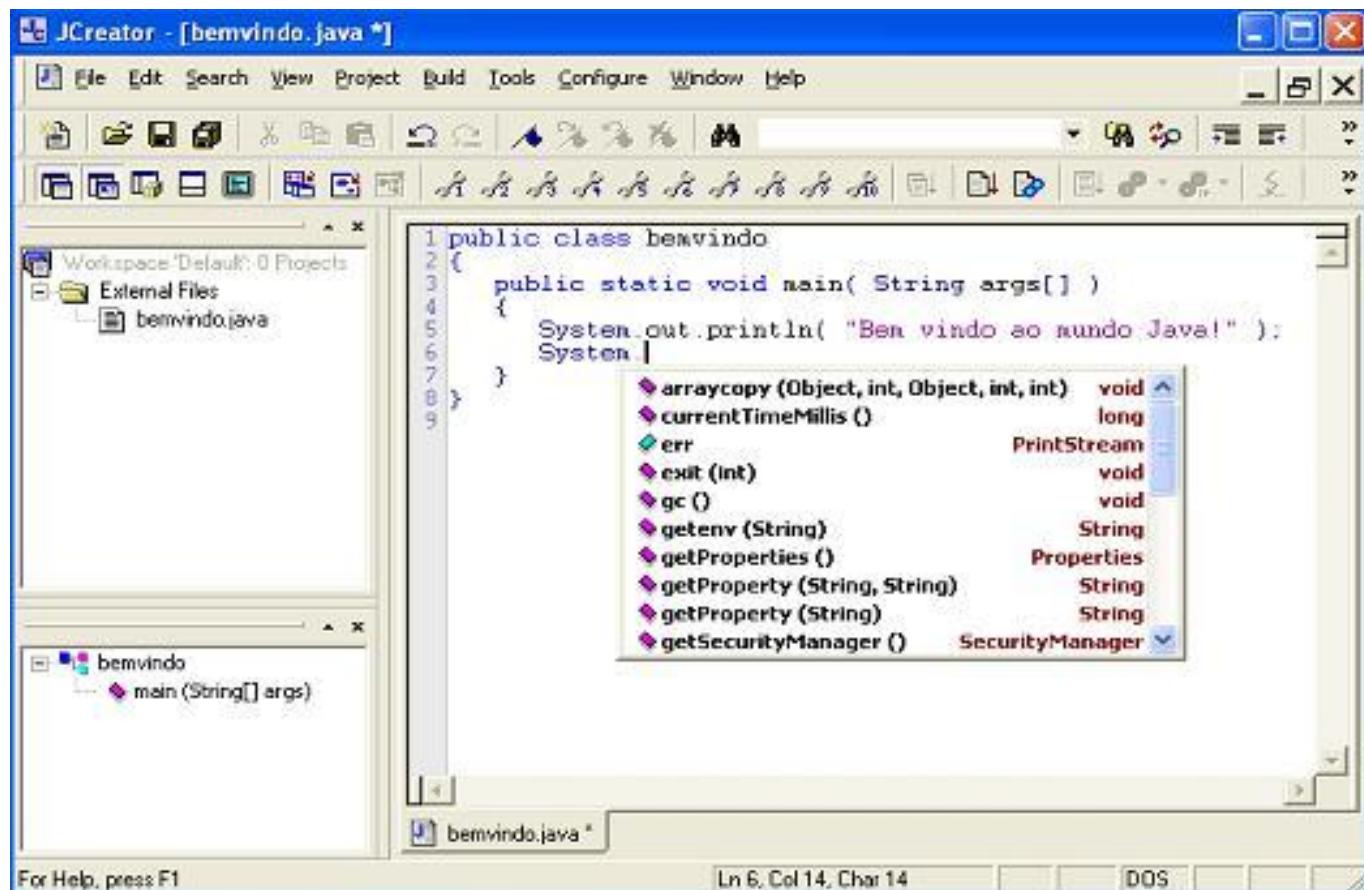


Figura 13 – Jcreator

4.2. Eclipse

Eclipse é uma IDE gratuita e open-source para o desenvolvimento de aplicativos em Java e em outras linguagens, proporciona o autocompleto dos códigos, permite a criação de códigos através da ferramenta, além de permitir estender as suas funcionalidades por meio da instalação de plugins. O download pode ser efetuado na página <http://www.eclipse.org/downloads>.

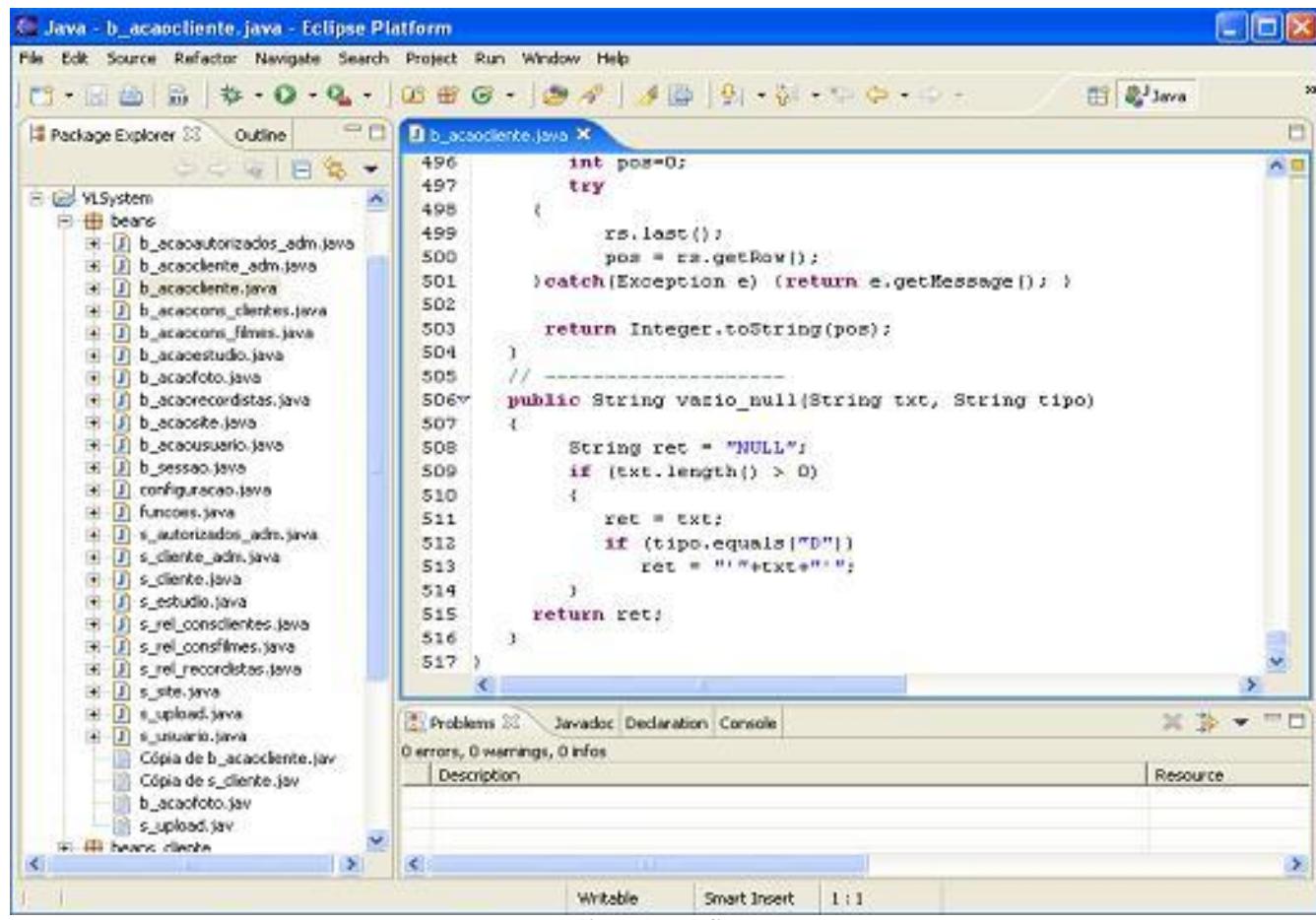


Figura 14 – Eclipse

4.3. NetBeans

O NetBeans, assim como a IDE Eclipse é uma ferramenta gratuita e open-source desenvolvida pela Sun Microsystems, para o desenvolvimento de aplicativos em Java, a ferramenta já possui na sua instalação tudo que será necessário para a escrita dos programa nas três áreas do Java, em suas últimas versões esta possibilita através do ambiente gráfico fazer conexões com banco de dados, criação de alguns componentes para a construção de páginas Web, além de facilitar a criação de ambientes swing como nas outras versões.

O download poderá ser efetuado na página <http://java.sun.com>.

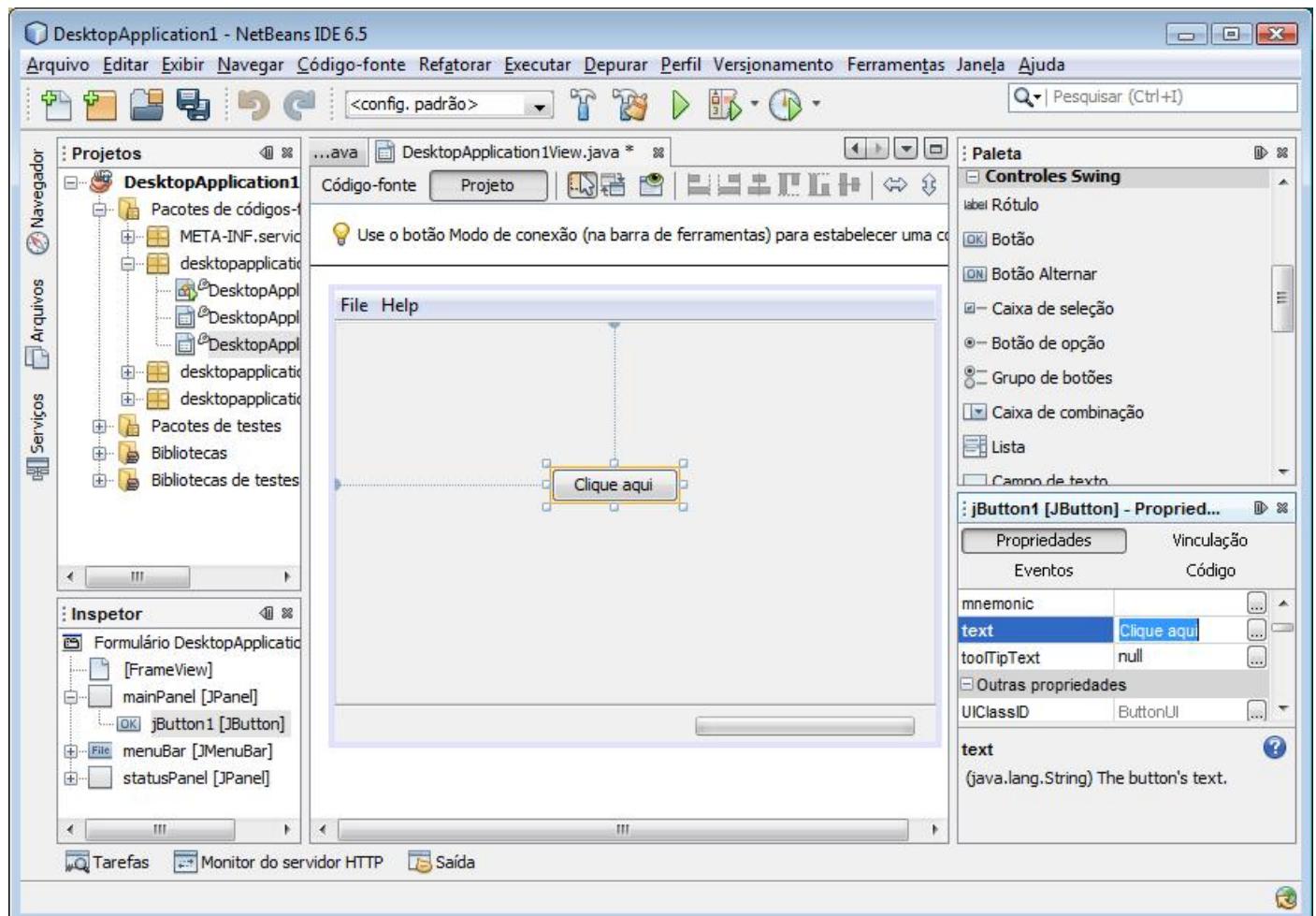


Figura 15 - NetBeans

4.4. Egen Developer

O Egen Developer é uma ferramenta RAD (Desenvolvimento Rápido de Aplicativos), gratuita e open-source (figura 11), proporciona o desenvolvimento rápido de aplicativos Java para Web, construindo sistemas apenas interagindo com a ferramenta, através de um ambiente gráfico, esta é baseada no framework MVC (Modelo Visualização) Struts, ou seja, organiza os códigos em camadas separadas . O download da ferramenta entre outras informações pode ser obtidas no site: <http://www.egen.com.br>.

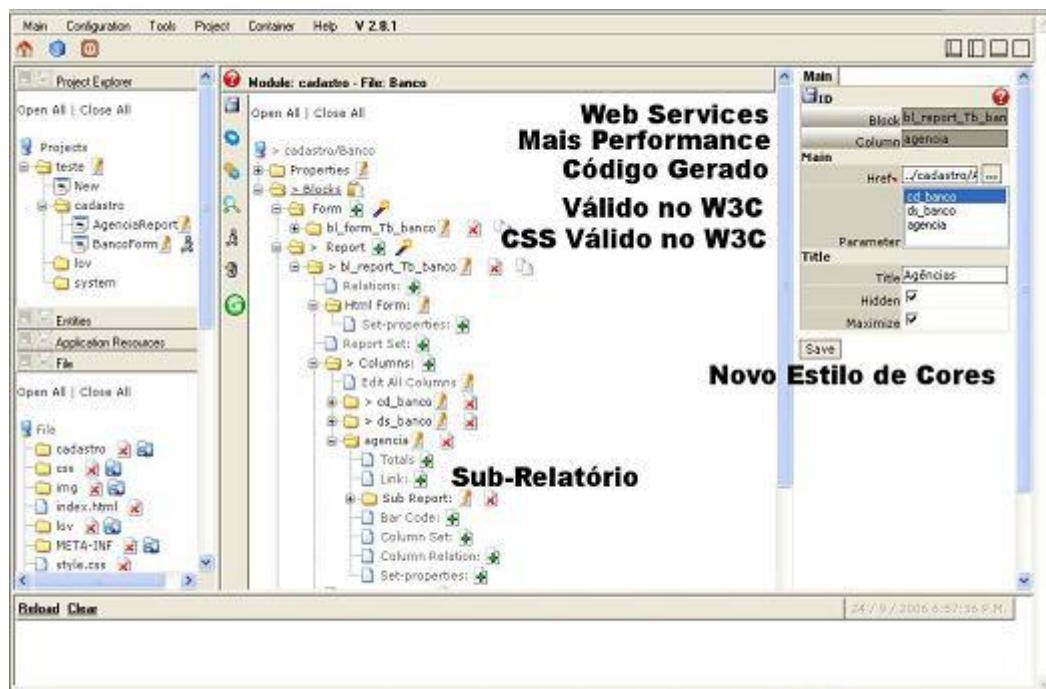


Figura 16 - Egen Developer

5. SINTAXE DO JAVA

5.1. Variáveis

No Java, uma variável pode ter basicamente três tipos.

- Primitivas: (valores atômicos) variáveis de “baixo nível”, como números inteiros, com casas decimais, caracteres e boleanas (sim/ não, verdadeiro / falso);
- Compostas / Por Referência: representam uma estrutura de dado já preparada para um determinado fim como Data, Conta, Produto, Moeda, String / Cadeia de caracteres, etc.;
- Arrays: representam listas que podem assumir qualquer tipo primitivo ou composto.

As variáveis tem seu acesso limitado de acordo com o seu escopo.

A sintaxe para declaração de uma variável é simples:

modificadores tipo nome;

A seguir será demonstrado um exemplo de declaração de variável:

private double m ;

O modificador private, especifica o acesso para a variável, nesse caso, a variável será acessível somente no local em que foi declarada. A palavra double representa o tipo da variável e m é o nome dado a variável.

5.1.1. Variáveis de Tipo Primitivo

As variáveis devem possuir nome, tipo e valor, sendo que toda vez que se necessite usar uma variável no Java, é preciso declará-la, para então atribuir valores a mesma. A seguir são apresentados os tipos primitivos usados para definir variáveis na linguagem Java:

| Tipo | Sorte | Valor mínimo | Valor máximo | Valor inicial default |
|----------------|--|----------------------------|----------------------------|-----------------------|
| boolean | Valores lógicos | true | false | false |
| byte | Inteiros de 8 bits | -128 | +127 | 0 |
| short | Inteiros de 16 bits | -32.768 | +32.767 | 0 |
| char | Caracteres codificados em Unicode de 16 bits | \u0000 | \uffff | \u0000 |
| int | Inteiros de 32 bits | -2.147.483.648 | +2.147.483.647 | 0 |
| long | Inteiros de 64 bits | -9.223.372.036.854.775.808 | +9.223.372.036.854.775.807 | 0 |
| float | Pto. Flut. de 32 bits | NEGATIVE_INFINITY | POSITIVE_INFINITY | 0 |
| double | Pto. Flut. de 64 bits | NEGATIVE_INFINITY | POSITIVE_INFINITY | 0 |

Figura 17 - Tipos primitivos do Java

```

public class TiposPrimitivos {
    public static void main(String[] args) {

        int numeroInteiro; // apenas declara a variável
        numeroInteiro = 5; // inicializa a variável

        double numeroDecimal = 2.34545; // declara e inicializa
        char letra = 'a'; // declarando e inicializando um char
        byte a = 127;
        short b = 32767;
        long c = 9223372036854775807L; // L no final
        float d = 1292.74F; // F (ou f) após o literal indica precisão simples
        boolean e = true;

        // Definindo mais de uma variável ao mesmo tempo
        int x, y, z;

        // Definindo e inicializando mais de uma variável
        int q = 55, w = 6, t = 44;
    }
}

```

Figura 18 - Exemplos de Tipos Primitivos

5.1.1.1 Classes Wrapper

Cada classe de tipo primitivo possui uma classe wrapper para operações sobre cada tipo de dado. Segue a lista de classes wrapper para cada tipo primitivo:

| | | |
|-------------------|--------------------|--------------|
| int -> Integer | boolean -> Boolean | byte -> Byte |
| char -> Character | short -> Short | long -> Long |
| float -> Float | double -> Double | |

Algumas operações comumente utilizadas:

Integer.parseInt(String s): Converte uma string para inteiro

Double.parseDouble(String s): Converte uma string para double

Float.parseFloat(String s): Converte uma string para float

5.1.2. Variáveis Compostas / Referência

Tipos que são definidos por uma classe. Exemplos: String, Date, List, Vector, etc.

```

public class TiposCompostos {
    public static void main(String[] args) {
        // Usando a classe String
        String palavra = "Teste";
        String frase;
        frase = "Teste de uma frase";
        String teste2 = new String("Inicializar pelo construtor da classe");

        // Usando a classe Date - é uma classe do pacote java.util
        // Pegando a data do Sistema Operacional
        java.util.Date hoje = new java.util.Date();

        // Inicializando uma data no construtor da classe
        java.util.Date dia = new java.util.Date("11/22/1974");

        // pode-se criar usando uma classe própria
        MinhaClasse objeto1 = new MinhaClasse();
        objeto1.meuMétodo();
    }
}

```

Figura 19 - Exemplos de Tipos Complexos

5.2. Declarando Variáveis

As declarações de variáveis consistem de um tipo e um nome de variável: como segue o exemplo:

```

int idade;
String nome;
boolean existe;
double salario;

```

Os nomes de variáveis podem começar com uma letra, um sublinhado (_), ou um cifrão (\$). Elas não podem começar com um número. Depois do primeiro caracter pode-se colocar qualquer letra ou número.

5.3. Atribuições a variáveis

Após declarada uma variável a atribuição é feita simplesmente usando o operador ‘=’:

```

idade = 18;
nome = "Fulano de Tal";
existe = true;
salario = 1550.50;

```

5.4. Conversões de Tipos por Cast

Pode-se atribuir uma variável primitiva usando um valor literal ou o resultado de uma expressão. Números inteiros literais (como 4 por exemplo) sempre serão implicitamente um tipo int, no capítulo anterior foi visto que um tipo int é um valor de 32 bits. Não haverá problema em atribuirmos um valor a uma variável int ou long, mas o que aconteceria se atribuíssemos um valor inteiro a uma variável byte de 8 bits? Um byte não poderá armazenar tantos bits quanto uma variável do tipo int, mas então como o seguinte código é executado?

```
byte b = 30;
```

Este código somente funciona porque o compilador java compacta automaticamente o valor literal (30) para o tipo byte. Em outras palavras o compilador executa o cast. Também poderíamos moldar explicitamente o valor da seguinte forma:

```
byte b = (byte) 30
```

O exemplo acima é exatamente igual ao anterior, porem com uma modelagem (cast) explicita sobre o valor 30 que originalmente é um valor int de 32 bits.

O mesmo ocorre com valores de ponto flutuante porem, com algumas diferenças. Com estes o tipo padrão de armazenamento é o tipo double, qualquer numero de ponto flutuante atribuído a uma variável será considerado como um double a menos que seja explicitamente declarado no código que deseja-se utilizar outro tipo, como um float. Mas ao contrario do que acontece nos tipos inteiros um double não é automaticamente moldado para os demais tipos de dados de menor precisão. O código abaixo não será compilado:

```
float f = 34.5;
```

Poderíamos imaginar que o valor 34.5 caberia tranquilamente em uma variável com o tamanho de um float, mas por segurança o compilador não permitirá isso, justificado pela perca de precisão de tal transformação. Para atribuir um valor literal de ponto flutuante a uma variável do tipo float precisamos modelar o valor ou então acrescentar um f ao final do literal, Mas lembre-se de que com isso você estará truncando os bits mais a esquerda do valor, neste caso específico de um cast de double para float metade dos bits serão descartados. Os exemplos abaixo serão compilados e executados corretamente:

```
float f = (float) 34.5;
float f = 34.5f;
float f = 34.5F;
```

5.5. Comentários

Java possui três tipos de comentário:

1. /* ... */: como no C e C++, tudo que estiver entre os dois delimitadores são ignorados:

```
/* Este comentário ficará visível somente no código o compilador ignorará completamente este
trecho entre os delimitadores
*/
```

2. Duas barras (//): também podem ser usadas para se comentar uma linha:

```
int idade; // este comando declara a variável idade
```

3. / ... **/:** Este comentário é especial e é usado pelo **javadoc** para gerar uma documentação do código.

5.6. Caracteres especiais

| Caracter | Significado |
|----------|--|
| \n | Nova Linha |
| \t | Tab |
| \b | Backspace |
| \r | Retorno do Carro |
| \f | “Formfeed” (avança página na impressora) |
| \\\ | Barra invertida |
| \' | Apóstrofe |
| \” | Aspas |
| \ddd | Octal |
| \xdd | Hexadecimal |

5.7. Operadores

5.7.1. Operadores Aritméticos

| Operador | Significado | Exemplo |
|----------|---------------|-----------|
| + | soma | $3 + 4$ |
| - | subtração | $5 - 7$ |
| * | multiplicação | $5 * 5$ |
| / | divisão | $14 / 7$ |
| % | módulo | $20 \% 7$ |

Exemplo Aritmético:

```
class ArithmeticTest {
    public static void main ( Strings args[] ) {
        short x = 6;
        int y = 4;
        float a = 12.5f;
        float b = 7f;

        System.out.println ("x é " + x + ", y é " + y );
        System.out.println ("x + y = " + (x + y) );
        System.out.println ("x - y = " + (x - y) );
        System.out.println ("x / y = " + (x / y) );
        System.out.println ("x % y = " + ( x % y ) );

        System.out.println ("a é " + a + ", b é " + b );
        System.out.println (" a / b = " + ( a / b ) );
    }
}
```

A saída do programa acima é :

```
x é 6, y é 4
x + y = 10
x - y = 2
x / y = 1
x % y = 2
a é 12.5, b é 7
a / b = 1.78571
```

5.8. Mais sobre atribuições

Variáveis podem atribuídas em forma de expressões como:

```
int x, y, z;
x = y = z = 0;
```

No exemplo as três variáveis recebem o valor 0;

Operadores de Atribuição:

| Expressão | Significado |
|-----------|-------------|
| $x += y$ | $x = x + y$ |
| $x -= y$ | $x = x - y$ |
| $x *= y$ | $x = x * y$ |
| $x /= y$ | $x = x / y$ |

5.9. Incrementos e decrementos

Como no C e no C++ o Java também possui incrementadores e decrementadores:

```
y = x++;
y = - -x;
```

As duas expressões dão resultados diferentes, pois existe uma diferença entre prefixo e sufixo. Quando se usa os operadores (`x++` ou `x--`), y recebe o valor de x antes de x ser incrementado, e usando o prefixo (`++x` ou `--x`) acontece o contrário, y recebe o valor incrementado de x.

5.10. Comparações

Java possui vários operadores para testar igualdade e magnitude. Todas as expressões retornam um valor booleano (true ou false).

5.11. Operadores de comparação

| Operador | Significado | Exemplo |
|--------------------|------------------------|------------------------|
| <code>==</code> | Igual | <code>x == 3</code> |
| <code>!=</code> | Diferente (Não igual) | <code>x != 3</code> |
| <code><</code> | Menor que | <code>x < 3</code> |
| <code>></code> | Maior que | <code>x > 3</code> |
| <code><=</code> | Menor ou igual | <code>x <= 3</code> |
| <code>>=</code> | Maior ou igual | <code>x >= 3</code> |

5.12. Operadores lógicos

| Operador | Significado |
|-------------------------|-------------------------|
| <code>&&</code> | Operação lógica E (AND) |
| <code> </code> | Operação lógica OU (OR) |
| <code>!</code> | Negação lógica |

5.13. Operadores de Atribuição

Java oferece vários operadores de atribuição que abreviam as expressões de atribuição:

```
c = c + 3;           c += 3;           //   c = c * 3;   c *= 3;
c = c / 3;           c /= 3;           //   c = c - 3;   c -= 3;
c = c % 3;           c %= 3;
```

Java oferece operador de incremento unário, `++`, e o operador de decremento unário, `--`.

| Depois | Antes |
|-------------------------|-------------------|
| <code>c = c + 1;</code> | <code>c++;</code> |
| <code>C = c - 1;</code> | <code>++c;</code> |
| | <code>c--;</code> |
| | <code>--c;</code> |

Exemplo:

```
int c = 1;
System.out.println(c++); // Saída 1
System.out.println(c);  // Saída 2
```

5.14. Expressões

Expressões fazem combinações ordenadas de valores, variáveis, operadores, parênteses e chamadas de métodos, permitindo realizar cálculos aritméticos, concatenar strings, comparar valores, realizar operações lógicas e manipular objetos.

As expressões são avaliadas obedecendo as regras da Matemática. Um exemplo de regra a ser seguida é a da associatividade. Se um operador de mesmo nível aparece mais de uma vez em uma expressão, como em **a+b+c**, então o operador mais à esquerda é avaliado primeiro, sendo seguido pelo da direita, e assim por diante. A regra de associatividade pode ser visualizada no exemplo a seguir:

Exemplo 1: $((a+b)+c)$;

Em alguns casos é necessário utilizar operadores de diferentes níveis em uma mesma expressão fazendo com isso a aplicação da regra de precedência de operadores, como mostrado no exemplo a seguir:

Exemplo 2: $a+b*c$;

No exemplo acima multiplicou-se primeiro b com c e em seguida o somou-se o produto com o valor de a, pois a multiplicação e a divisão tem precedência sobre a adição e subtração.

5.15. Blocos de Código

Um bloco é definido por `({})` e contém um grupo de outros blocos. Quando um novo bloco é criado um novo escopo local é aberto e permite a definição de variáveis locais. As variáveis definidas dentro de um bloco só podem ser vistas internamente a este, e são terminadas ou extintas no final da execução do bloco.

```
void testabloco(){
    int x = 10, w=1;

    if (x> w)
    { // inicio do bloco
        int y=50;
        System.out.println("dentro do bloco");
        System.out.println("x: " + x);
        System.out.println("y: " + y);
    } // final do bloco
    System.out.println("w: " + w);
    System.out.println("y: " + y); // erro variável não conhecida
}
```

5.16. Controle de Fluxo

As linguagens de programação oferecem controles de fluxo (condicionais e de repetição), para que os comandos possam ser executados em diferentes partes de um programa, baseado em condições definidas.

Os comandos condicionais em Java são classificados em categorias como pode ser visto abaixo:

| Comando | Palavras-chave |
|--------------------------------------|--------------------------|
| Tomada de decisões | if-else, switch-case |
| Laços ou repetições | for, while, do-while |
| Apontamento e tratamento de exceções | try-catch-finally, throw |

5.17. Tomada de Decisões

5.17.1. Estrutura de seleção if

O comando if-else permite escolher alternadamente entre dois outros comandos a executar. Se o valor da expressão condicional for true, então o primeiro bloco/comando será executado, do contrário, o segundo.

Sintaxe:

```
if (expressão)
    comando
[else
    comando]
```

Exemplo 1:

```
if (resposta == true) {
    // comandos
}
```

Exemplo 2:

```
if (resposta == true) {
    // comandos
}
else {
    //comandos
}
```

5.17.2. Estrutura de seleção switch

Semelhante ao comando if, no entanto, permite a seleção múltipla, podendo testar vários valores em uma expressão.

Sintaxe:

```
switch (expressão) {
    case valor1: comando; [break];
    case valor2: comando; [break];
    ....
    [default: comando;]
```

Exemplo 1:

```
int mês = 3;
switch (mês)
{
case 1: System.out.println("Jan");
    break;
case 2: System.out.println("Fev");
    break;
    ....
}
```

5.18. Estruturas de Repetição

5.18.1. while

O *while* é utilizado para repetir um comando, ou um conjunto de comandos, enquanto a condição for verdadeira.

Sintaxe:

while (*expressão*)
 comando

Exemplo:

```
int i = 3;  
while (i <= 10) {  
    i++;  
    System.out.println(i);  
}
```

5.18.2. for

O laço de repetição *for* tem a função de executar um determinado número de vezes um determinado comando/bloco. A sintaxe do comando *for* pode ser vista abaixo.

Sintaxe:

for (*inicio*; *condição*; *incremento*)
 comando

Exemplo:

```
for (int i=1; i < 10; i++)  
    System.out.println(i);
```

5.18.3. do/while;

Este comando, não muito diferente do comando *while*, é um tipo de laço de repetição que executa um comando/bloco e em seguida avalia a expressão condicional.

Sintaxe:

do {
 comandos
}
while (*expressão*);

Exemplo:

```
do {  
    System.out.println("Índice:" +a);  
    a++;  
} while (a != 5);
```

5.19. Instruções break e continue;

Na execução de uma estrutura de repetição (for, while e do/while), a instrução **break** ocasiona a saída imediata da estrutura, enquanto a instrução **continue** pula as instruções restantes e prossegue com o teste para a próxima iteração do laço:

Exemplo de Continue:

```
for(int a=1; a<10; a++)
{
    if(a == 5)
        continue;
    saida += a + " ";
}
```

Exemplo de Break:

```
a = 10;
somatorio = 0;
do {
    if(a == 0)
        break;
    somatorio += 10 / a--;
} while (true);
```

5.20. Instruções rotuladas

Para que o fluxo de execução continue ou pare, respeitando blocos definidos pelo programador, utilizamos as instruções rotuladas:

```
stop: {
    for(int i=0; i<5; i++)
        for(int j=0; j<3; j++) {
            if(i == 2)
                break stop;
            saida += "[ "+i+" "+j+" ] ";
        }
}
```

5.21. Exercício 1 - Fatorial

Faça um programa em Java que solicite ao usuário um número que será utilizado para calcular o seu fatorial. Mostre o resultado ao usuário.

Utilize a estrutura de repetição for

Fatorial de 5:

$5 * 4 * 3 * 2 * 1 = 120$

5.22. Exercício 2 - Fibonacci

Fazer um programa que imprime a seguinte seqüência: 1, 1, 2, 3, 5, 8, 13, ... (serie de Fibonacci)

Utilize a estrutura de repetição while

Imprimir somente os valores menores que 100.

6. CAIXAS DE DIÁLOGO

O pacote javax.swing oferece um conjunto de classes para a criação de interfaces gráficas para interação com o usuário. A seguir será apresentada a classe JOptionPane que é usada para trabalhar com janelas de caixa de diálogo no Java.

6.1. A classe JOptionPane

A classe JOptionPane oferece diversos métodos para exibir mensagens para o usuário, desde simples mensagens de alerta a pequenas entradas de dados, ou ainda permite que sejam personalizados botões e imagens das mensagens.

6.1.1. Mensagens de Alerta

A forma mais simples para exibir um texto de feedback, pode ser obtida com a seguinte instrução:

```
import javax.swing.JOptionPane;  
...  
JOptionPane.showMessageDialog(null, "Mensagem !");
```

A execução da instrução acima gera uma tela como ilustrada na figura a seguir.



Figura 20 - Uso do JOptionPane para Mensagem

O primeiro parâmetro indica a que janela esta caixa de diálogo está relacionada, caso não esteja relacionada a nenhuma janela, pode-se passar null nesse parâmetro. Se for necessário quebrar a linha dentro da mensagem deve ser usado o caractere \n que indica uma nova linha, isto faz com que mensagens mais longas possam ser separadas em várias linhas.

Outra possibilidade é de trocar o tipo de mensagem e de alterar o título da caixa de dialogo. Exemplo:

```
JOptionPane.showMessageDialog(null, "Mensagem!", "Título", JOptionPane.WARNING_MESSAGE);
```

A execução da instrução acima gera uma tela como ilustrada na figura a seguir:

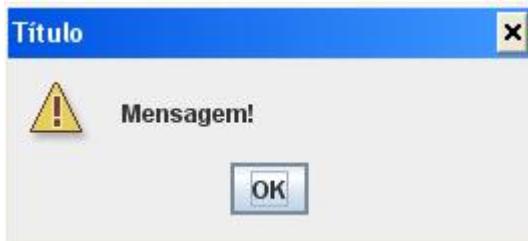


Figura 21 – Uso do JOptionPane para Mensagem

Mais alguns exemplos de uso do método showMessageDialog:

```
JOptionPane.showMessageDialog(null, "Mensagem!", "Título", JOptionPane.ERROR_MESSAGE);
JOptionPane.showMessageDialog(null, "Mensagem!", "Título", JOptionPane.PLAIN_MESSAGE);

ImageIcon icon = new ImageIcon("image/fig.gif");
JOptionPane.showMessageDialog(null, "Mensagem!", "Título", JOptionPane.INFORMATION_MESSAGE, icon);
```

6.1.2. Caixa de Confirmação

Além de exibir mensagens ao usuário pode-se também solicitar ao usuário confirmações. O exemplo a seguir permite ao usuário responde sim ou não para a caixa de diálogo e pode-se receber na variável n a sua resposta:

```
int n = JOptionPane.showConfirmDialog(null, "Confirma a exclusão?",
                                      "Confirmação", JOptionPane.YES_NO_OPTION);
if (n == JOptionPane.YES_OPTION) {
    JOptionPane.showMessageDialog(null, "confirmado");
}
```

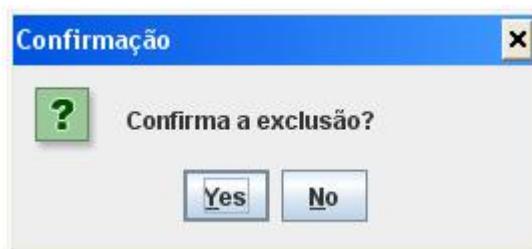


Figura 22 - Uso do JOptionPane para Confirmação

Pode-se ainda personalizar o texto dos botões a serem exibidos e assim mesmo receber como retorno um inteiro que representa o botão escolhido pelo usuário, sendo o primeiro botão zero e o segundo 1 e assim por diante, caso o usuário feche a janela o valor retornado será -1. Exemplo:

```
Object[] options = { "Confirmar ", "Cancelar" };
int n = JOptionPane.showOptionDialog(null, "Confirma exclusão ?",
                                      "Confirmação",
                                      JOptionPane.YES_NO_OPTION,
                                      JOptionPane.QUESTION_MESSAGE,
                                      null, // não usa um ícone
                                      options, // título dos botões
                                      options[0]); // opção default
if (n == JOptionPane.YES_OPTION) {
    JOptionPane.showMessageDialog(null, "confirmado");
}
```

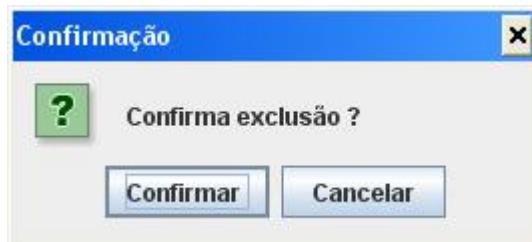


Figura 23 - Uso do JOptionPane para Confirmação

6.1.3. Caixa de Opções

Além de receber uma resposta do usuário através dos botões também é possível indicar uma lista de opções para o mesmo selecionar, e também, indicar que item deve estar selecionado inicialmente. Exemplo:

```
Object[] opcoes = { "Opção 1", "Opção 2", "Opção 3", "Opção 4" };
ImageIcon icon = new ImageIcon("image/pizza.gif");
String s = (String) JOptionPane.showInputDialog(null, "Escolha a opção",
    "Título", JOptionPane.PLAIN_MESSAGE, icon, opcoes, "Opção 2");
JOptionPane.showMessageDialog(null, s);
```

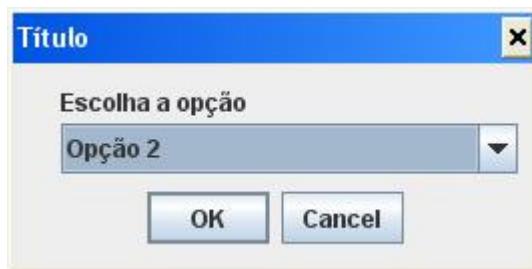


Figura 24 - Uso do JOptionPane para Opções

6.1.4. Caixa de Entrada de Dados

Também é possível receber uma String digitada pelo usuário usando este mesmo método. Exemplo:

```
String ler = (String) JOptionPane.showInputDialog(null,
    "Informe o nome ?", "Identificação",
    JOptionPane.PLAIN_MESSAGE,
    null, null, "");
```



Figura 25 - Uso do JOptionPane para Entrada de Dados

7. COVENÇÕES PARA NOMES DE CLASSES E VARIÁVEIS

Para a escrita de programas em Java, algumas convenções para declaração e escrita de classes e variáveis podem ser seguidas, para que assim mantenha-se um padrão de nomenclatura. Segundo Castela (2006), "Como o Java é case-sensitive, todo e qualquer comando, variável, nome de classes e objetos escritos em Java será diferenciado entre maiúsculas e minúsculas.", o Java impõem diferença de caixa alta de caixa baixa, como é ilustrado no exemplo a seguir:

Exemplo 1:

```
int Exemplo;
int exemplo;
int eXemplo;
```

Como mostra o exemplo acima, as variáveis são do mesmo tipo, porém diferenciam-se uma das outras, pela forma que foram escritas. As variáveis em Java são declaradas com letras minúsculas, porém se haver uma palavra composta, declara-se a primeira como minúscula e a segunda com a primeira letra do segundo nome em maiúscula, como exemplo tem-se as duas situações a seguir:

Exemplo 2:

| Variáveis | Descrição |
|-----------------------|--|
| int nome; | formato de declaração com apenas uma palavra, todas as letras minúsculas |
| int nomeComposto; | formato de declaração com duas palavras, com a inicial da segunda palavra em maiúsculo |
| int nomeCompostoDois; | formato de declaração com várias palavras, com o restante das outras palavras com inicial em maiúsculo |

Assim como nas variáveis, para criar uma classe no Java aconselha-se que algumas convenções sejam postas em prática. Toda classe começa com letras maiúsculas, mas se existir uma segunda palavra ou mais, estas deverão também iniciar com primeira letra em maiúscula. O exemplo a seguir demonstra como criar uma classe de acordo com a convenção:

Exemplo 3:

| Classe | Descrição |
|---|---|
| public class Pessoa{ } | Formato de declaração de Classe com uma palavra |
| public class PessoaFisica{ } | Formato de declaração de Classe com duas palavras |
| public class PessoaFisicaJuridica{ } | Formato de declaração de Classe com várias palavras |

8. A CLASSE STRING

Em Java string é uma classe ao invés de um tipo de dado, no caso um objeto String. Uma string caracteriza-se por ser uma seqüência de caracteres, como palavras, frases e nomes. A classe String fica no pacote java.lang, que é importado automaticamente pela aplicação.

Exemplo da Criação de uma String:

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

Objetos String são os únicos em Java que podem ser criados sem o operador new,, atribuindo diretamente um valor literal, pois seu esse objeto é tão usado que os implementadores de Java disponibilizaram uma forma mais simples de instanciar objetos String.

Outra forma de criar uma String:

```
String nome = "Paulo";
```

Mas quando um objeto String é criado dessa forma, nos bastidores o compilador está fazendo **String nome = new String("Paulo");**

8.1. Comprimento da String

O método length permite descobrir o número de caracteres contidos numa String, por exemplo:

```
String s = new String( "Olá!" );
int tamanho = s.length();
```

Resulta o valor 4 para o inteiro chamado tamanho.

8.2. Concatenação

Para concatenar duas Strings, pode-se utilizar o operador + ou o método concat. A concatenação resulta na criação de um novo objeto.

Exemplo:

```
String s1 = "Olá ";
String s2 = "Mundo!";
System.out.println( s1.concat( s2 ) );
System.out.println( s1 + s2 );
```

A saída gerada pelo exemplo acima será:

```
olá Mundo!
olá Mundo!
```

8.3. Comparação

Para determinar se duas Strings possuem o mesmo conteúdo, usa-se o método **equals()**. Este método considera maiúsculas e minúsculas como diferentes. Para não fazer esta distinção, pode-se usar o método **equalsIgnoreCase()**.

É importante não confundir estas comparações de conteúdo com o uso do operador `= =`, que, quando usado entre duas variáveis que são referências a objetos, serve para determinar se as duas variáveis apontam para o mesmo objeto.

Observe o exemplo abaixo:

```
String s1,s2,s3,s4,s5,s6;
boolean b1, b2, b3, b4, b5, b6, b7, b8, b9;

s1 = "Olá!";
s2 = "Olá!";
s3 = "olá!";
s4 = new String( "Olá!" );
s5 = s4;
s6 = new String( "Olá!" );

b1 = s1.equals( s2 );
b2 = s1.equals( s3 );
b3 = s1.equalsIgnoreCase( s3 );
b4 = s1.equals( s4 );
b5 = ( s1 == s2 );
b6 = ( s1 == s4 );
b7 = ( s4 == s5 );
b8 = ( s4 == s6 );
b9 = s4.equals( s6 );
```

Neste exemplo, vão ficar com o valor false somente b2, b6 e b8.

8.4. Conversão de Dados Numéricos para String

O método estático **valueOf** da classe String permite obter uma String que representa um dado tipo numérico.

Exemplo:

```
double a = 3.1415927;
int i = 123;
System.out.println( "a = " + String.valueOf( a ) );
System.out.println( "i = " + String.valueOf( i ) );
```

Este código resulta na impressão de `a = 3.1415927` seguido de `i = 123`.

O mesmo resultado pode ser obtido com uma sintaxe simplificada, pois numa concatenação de String com o operador `+`, números são transformados automaticamente em String. Assim, as duas últimas linhas acima podem ser substituídas por:

```
System.out.println( "a = " + a );
System.out.println( "i = " + i );
```

8.5. Conversão de String para Dados Numéricos

Para realizar a operação inversa, ou seja transformar Strings em tipos numéricos, podemos usar as chamadas "wrapper classes", que podem ser encontradas no pacote `java.lang`.

Por exemplo, a classe wrapper de um tipo `int` é `Integer`, `double` é `Double`, etc. Para transformar uma String num `double`, pode-se usar o método estático **valueOf** em conjunto com o método **doubleValue** para transformar no tipo primitivo `double`. Ainda precisaremos "desembrulhar" o nosso número, ou seja traduzir o nosso objeto `Double` num tipo primitivo `double`. Isto é feito chamando o método `doubleValue` da classe

Double. O trecho de código abaixo ilustra este procedimento, assim como o semelhante para o caso de um tipo inteiro. (Outros tipos podem ser tratados também da mesma maneira.)

```
String s1 = "3.1415927";
String s2 = "123";
double d1 = Double.valueOf( s1 ).doubleValue();
double d2 = Double.parseDouble(s1);
Double d3 = new Double(s1);
Double d4 = Double.valueOf( s1 );
int i1 = Integer.valueOf( s2 ).intValue();
int i2 = Integer.parseInt( s2 );
```

Ao executar o código acima as variáveis ficarão assim:

```
d1 = 3.1415927
d2 = 3.1415927
d3 = 3.1415927
d4 = 3.1415927
i1 = 123
i2 = 123
```

8.6. Formatação de String

A transformação de números em String resulta numa representação completa do número, sem arredondamento nem controle do formato. Recursos para formatar texto, e em especial números, podem ser encontrados no pacote `java.text`. Formatos são objetos em Java. O exemplo a seguir apresenta o uso de um objeto de formatação da classe `DecimalFormat`:

```
import java.text.*;
...
double a = 3.1415927;
DecimalFormat formato = new DecimalFormat( "0.#####");
System.out.println( "a = " + meuFormato.format( a ) );
```

Saída: a = 3,1416

OBS: o método `format` faz o arredondamento para o número de casas decimais especificados no formato.

Exemplo para formatar um valor para duas casas decimais, no formato de moeda:

```
DecimalFormat formata = new DecimalFormat("R$ ###,###,##0.00");
double b = 134.5;
System.out.println( "Valor: "+formata.format(b) );
```

Saída: Valor: R\$ 134,50

8.7. Decomposição de Strings

Se uma String representa uma sucessão de palavras, pode ser necessário decompô-la em palavras individuais.

A método `split()` do pacote `java.lang` permite realizar este tipo de operação. Esse método recebe como parâmetro de entrada uma expressão regular e retorna um array de strings.

Exemplo:

```
String dados = "Real-How-To";
String[] vetor = dados.split("-");
for (int i = 0 ; i < vetor.length ; i++) {
    System.out.println(vetor[i]);
}
```

Valores das variáveis após a execução:

```
Real
How
To
```

8.8. Mais Exemplos da Classe String

Mais alguns exemplos de uso de métodos da classe String:

```
String texto = "classe ????"; // declara e inicializa a String texto
texto = texto.substring(0,6); // parte da String(posição inicial 0, até a 6)
System.out.println(texto);
System.out.println(texto.toUpperCase()); // converte String p/ MAIÚSCULO
texto += " String"; // concatenar
System.out.println(texto);
int tamanho = texto.length(); // tamanho da String
System.out.println("Tamanho da String: "+tamanho);
```

A saída gerada pelo programa acima é:

```
classe
CLASSE
classe String
Tamanho da String: 13
```

Há outros métodos que permitem trabalhar com Strings, por exemplo, descobrir qual o caráter numa dada posição, entre outras operações úteis. A figura a seguir apresenta diversas funções existentes na classe String:

| | |
|-----------------|--|
| int | lastIndexOf(String str) Returns the index within this string of the last occurrence of the specified substring. |
| int | lastIndexOf(String str, int fromIndex) Returns the index within this string of the last occurrence of the specified substring, searching backward starting at the specified index. |
| int | length() Returns the length of this string. |
| String | replace(char oldChar, char newChar) Returns a string resulting from replacing all occurrences of <code>oldChar</code> in this string with <code>newChar</code> . |
| String | replaceAll(String regex, String replacement) Replaces each substring of this string that matches the given regular expression with the given replacement. |
| String | replaceFirst(String regex, String replacement) Replaces the first substring of this string that matches the given regular expression with the given replacement. |
| String[] | split(String regex) Splits this string around matches of the given regular expression . |
| String[] | split(String regex, int limit) Splits this string around matches of the given regular expression . |
| String | substring(int beginIndex) Returns a string that is a substring of this string. |
| String | substring(int beginIndex, int endIndex) Returns a string that is a substring of this string. |
| char[] | toCharArray() Converts this string to a new character array. |
| String | toLowerCase() Converts all of the characters in this <code>String</code> to lower case using the rules of the default locale. |
| String | toUpperCase() Converts all of the characters in this <code>String</code> to upper case using the rules of the default locale. |
| String | trim() Returns a string whose value is this string, with any leading and trailing whitespace removed. |

Figura 26 - Classe String

9. DATA E HORA - CLASSE DATE

9.1.1. A Classe Date

Em Java informações de data e hora são representadas pela classe Date. A classe Date, encontrada no pacote java.util, encapsula um valor long que representa um momento específico no tempo. Um construtor útil é Date(), que cria um objeto Date representando a hora em que o objeto foi criado. O método getTime() retorna o valor long de um objeto Date.

O exemplo abaixo, ilustra o uso do construtor Date() para criar uma data e inicializar com a data e hora em que o objeto foi criado. O data e hora é obtida do sistema operacional onde o programa está executando. O método getTime() retorna o número de milisegundos que a data representa.

```
import java.util.*;  
  
public class HoraAtual {  
    public static void main(String[] args) {  
        Date agora = new Date();  
        long agoraLong = agora.getTime();  
        System.out.println("O valor é " + agoraLong);  
    }  
}
```

9.1.2. A Classe SimpleDateFormat

A classe SimpleDateFormat possibilita criar Strings que representam formatos em que se deseja trabalhar com data e hora. Alguém nos Estados Unidos pode preferir ver "December 25, 2000", enquanto que na França as pessoas estão mais acostumadas a "25 decembre 2000". Também é muito comum a data ser trabalhada no Brasil no formato "25/12/2009 14:35:22", ou seja, no formato "dd/MM/yyyy hh:mm:ss".

Para isso, é preciso criar uma instância de uma classe SimpleDateFormat. Este objeto contém informação a respeito de um formato particular no qual a data será tratada. Para usar o formato default do computador, pode-se aplicar o método getDateInstance() para criar o objeto DateFormat apropriado, como apresentado no exemplo abaixo:

```
SimpleDateFormat formatoData = SimpleDateFormat.getInstance();
```

A classe SimpleDateFormat é encontrada no pacote java.text.

9.2. Convertendo Date para String

Pode-se utilizar o método **format** para converter um objeto Date para uma string, conforme exemplo abaixo:

```
/*  
 * Definir uma variável de método para conter  
 * uma data de nascimento  
 */  
Date diaNascimento;  
// inicializar a data com a data do sistema  
diaNascimento = new Date();
```

```
// vamos criar um objeto de formatação para data
SimpleDateFormat formatoData = new SimpleDateFormat("dd/MM/yyyy");

// agora vamos mostrar a data formatada
System.out.println("A data formatada é: " + formatoData.format(diaNascimento));
```

O mesmo pode ser feito com a hora, conforme exemplo abaixo:

```
// vamos criar um objeto para formatar a hora
SimpleDateFormat formatoHora = new SimpleDateFormat("HH:mm:ss");
System.out.println("A hora é: " + formatoHora.format(diaNascimento));
```

9.3. Convertendo String para Date

Pode-se utilizar o método **parse** para converter uma String para um objeto Date, conforme exemplo abaixo:

```
// vamos criar um objeto de formatação para data
SimpleDateFormat formatoData = new SimpleDateFormat("dd/MM/yyyy");

// agora vamos montar uma data a partir de uma string
String dataLida = "15/02/1987";
try {
    // vamos agora converter a string para o tipo date
    diaNascimento = formatoData.parse(dataLida);
} catch (ParseException ex) {
    System.out.println("Não foi possível converter a string para data!");
}
```

Pode-se utilizar o método **parse** para converter uma String para um objeto Date, para armazenar uma informação de hora, conforme exemplo abaixo:

```
// vamos criar um objeto para formatar a hora
SimpleDateFormat formatoHora = new SimpleDateFormat("HH:mm:ss");

String horaLida = "20:32:40";
try {
    Date hora = formatoHora.parse(horaLida);
} catch (ParseException ex) {
    System.out.println("Erro na conversão!");
}
```

Pode-se utilizar o método **parse** para converter uma String para um objeto Date, para armazenar uma informação de data e hora, conforme exemplo abaixo:

```
// vamos criar um objeto para formatar a data e hora
SimpleDateFormat formatoDataHora = new SimpleDateFormat("dd/MM/yyyy HH:mm:ss");

String dataHoraLida = "15/02/1987 20:32:40";
try {
    Date dataHora = formatoDataHora.parse(dataHoraLida);
} catch (ParseException ex) {
    System.out.println("Erro na conversão!");
}
```

10. ARRAYS E MATRIZES

Arrays são utilizados para agrupar variáveis do mesmo tipo. O tipo pode ser qualquer tipo primitivo ou qualquer classe de objetos. Não é possível armazenar diferentes tipos em um único array. É possível ter um array de inteiros, ou um array de Strings, ou um array de array, etc.

10.1. Array

10.1.1. Declarar e Criar um Array

Para declarar um array, acrescenta-se colchetes após o nome da variável.

```
int c[];
```

Para criar o array utiliza-se a palavra-chave new e indicando o número de elementos do array entre colchetes:

```
c = new int[ 12 ];
```

Também pode-se declarar e já criar o array na mesma instrução, como a seguir:

```
int c[] = new int[ 12 ];
```

10.1.2. Inicialização de um Array com Tipos Primitivos

Quando um array é criado usando o operador new, todos os índices são inicializados (0 para arrays numéricos, falso para boolean, ‘\0’ para caracteres, e null para objetos).

O índice para especificar um elemento de um array é contado a partir de 0. Por exemplo, os comandos abaixo declaram e inicializam um array de 4 posições:

```
int b[] = new int[ 4 ];
b[ 0 ] = 5;
b[ 1 ] = 8;
b[ 2 ] = 3;
b[ 3 ] = 2;
```

O array pode ser criado e também inicializado através de uma lista de valores entre chaves. Os comandos acima podem ser substituídos por:

```
int b[] = { 5, 8, 3, 2 };
```

10.1.3. Inicialização de um Array com Objetos

No caso de um array de objetos, deve-se considerar que a criação do array não cria os objetos do array. Cria simplesmente uma lista de referências que ainda não apontam para nada (null).

Os objetos ainda precisam ser criados e as suas referências atribuídas aos elementos do array.

Por exemplo, o código a seguir cria um array de 3 posições e adiciona um objeto do tipo Date em cada posição:

```
Date datas[] = new Date[ 3 ];
datas[ 0 ] = new Date();
datas[ 1 ] = new Date("02/25/2007");
datas[ 2 ] = new Date("02/28/2007 21:14:30");
```

10.1.4. Comprimento de um Array

Qualquer array possui uma variável inteira **length** que fornece o número de elementos do array.

No exemplo abaixo, ao percorrer o array, utiliza-se length para determinar o seu tamanho, e assim, a condição de parada:

```
String palavras[] = new String[2];
palavras[0] = "palavra1";
palavras[1] = "palavra2";
// percorrer o array
for(int pos=0; pos < palavras.length; pos++)
    System.out.println("Índice "+pos+" : "+palavras[pos]);
```

Saída:

```
Índice 0 : palavra1
Índice 1 : palavra2
```

10.1.5. Mais Exemplos de Array

```
public class ExemploArray {
    public static void main(String[] args) {
        // Array de char
        char[] alfabeto = new char[26];
        alfabeto[0] = 'A';
        alfabeto[1] = 'B';    // ...
        alfabeto[25] = 'Z';
        // ou
        char[] alfabeto2 = {'A', 'B', '.', '.', '.', 'Z'};
        // Array de String
        String palavras[] = new String[2];
        palavras[0] = "palavra1";
        palavras[1] = "palavra2";
        // ou
        String[] palavras2 = new String[] { "outra", "forma", "de",
"inicializar" };
        // Array de int
        int[] a = new int[3];
        a[0] = 1;
        a[1] = 2;
        a[2] = 3;
        // ou
        int[] b = { 1, 2, 3 };
        // percorrer o array
        for(int pos=0; pos<palavras.length; pos++)
            System.out.println(palavras[pos]);
    }
}
```

10.2. Matrizes

Arrays de múltiplos índices podem ser definidos, sendo considerados como **arrays de arrays**. Os seguintes exemplos criam objetos que são essencialmente matrizes:

Exemplo 1:

```
int i[][]= new int[3][3];
i[0][0] = 1; // formato i[linhas][colunas]
i[0][1] = 2;
...
Ou
int i[][] = { { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 } };
```

| | | |
|---|---|---|
| 1 | 2 | |
| 4 | 5 | 6 |
| 7 | 8 | 9 |

Exemplo 2:

```
int i[][]= new int[3][2];
i[0][0] = 1; // formato i[linhas][colunas]
i[0][1] = 2;
...
ou
int i[][] = { { 1, 2 }, { 3, 4 }, { 5, 6 } };
```

| | |
|---|---|
| 1 | 2 |
| 3 | 4 |
| 5 | 6 |

10.2.1. Mais Exemplos de Matrizes

```
public class ExemplosMatriz {
    public static void main(String[] args) {
        int[][] matriz = new int[2][2];
        matriz[0][0] = 1;
        matriz[0][1] = 2;
        matriz[1][0] = 3;
        matriz[1][1] = 4;
        for (int i=0; i < matriz.length; i++)
        {
            for (int j=0; j < matriz[i].length; j++)
                System.out.println("[ "+i+", "+j+" ] = "+matriz[i][j]);
        }
        System.out.println();
        int[][] matriz2 = { {5,6},{7,8} };
        for (int i=0; i < matriz2.length; i++)
        {
            for (int j=0; j < matriz2[i].length; j++)
                System.out.println("[ "+i+", "+j+" ] = "+matriz2[i][j]);
        }
    }
}
```

Saída:

```
[0,0]=1
[0,1]=2
[1,0]=3
[1,1]=4

[0,0]=5
[0,1]=6
[1,0]=7
[1,1]=8
```

10.3. Passagem por Referência

Um array pode ser passado como argumento a um método. Deve-se notar que array são objetos e, na linguagem Java, objetos são sempre passados aos métodos por referência, ao passo que tipos primitivos são sempre passados por valor. Isto quer dizer que uma modificação ao array realizada no método chamado implicará na mesma modificação ao array definido no programa que chamou o método em questão. Veja no exemplo a seguir:

```
public static void main(String[] args) {
    // TODO code application logic here
    int v[] = new int[2];
    v[0] = 10;
    v[1] = 5;
    modifica(v);
    System.out.println( v[1] );
}

public static void modifica(int vet[]){
    vet[1] = 44;
}
```

A saída será 44

10.4. Exercício sobre Array

Ler dois arrays de 10 números (A e B) e:

- calcular S = (A[0] * B[9]) + (A[1] * B[8]) + ...
- calcular C, sendo $C[i] = A[i] / B[i]$
- imprimir os números pares de A
- imprimir o valor de S
- imprimir os números de C

- Controlar para que os números lidos sejam inteiros. Se não informar inteiro, ler o número novamente.

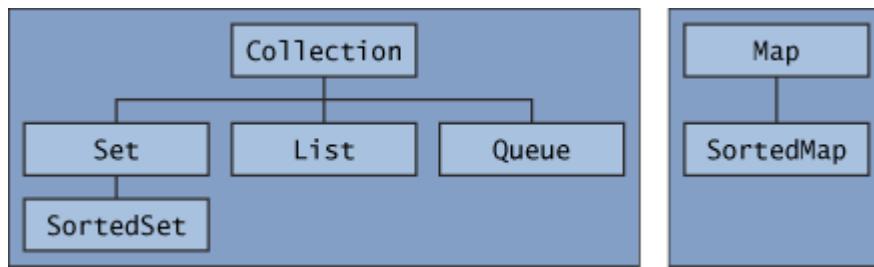
- Controlar para que se houver algum erro durante o cálculo, seja mostrada uma mensagem adequada ao usuário.

- Pode usar `System.out.println()` para gerar a saída.

11. COLEÇÕES

As coleções podem armazenar um número arbitrário de elementos, sendo que cada elemento é um outro objeto. O Java contém um framework (Java Collection Framework) com uma API de coleções do Java, organizada em uma hierarquia de interfaces e em uma hierarquia de implementação de classes em separado. **Interfaces:** permitem que as coleções sejam manipuladas independentes de suas implementações; **Implementações:** Classes que implementam uma ou mais interfaces.

O diagrama abaixo demonstra as interfaces de coleções no Java são organizadas:



Collection: Não existe uma implementação direta desta *interface*, porém, ela está no topo da hierarquia definindo operações que são comuns a todas as coleções;

Set: Está diretamente relacionada com a idéia de conjuntos. Assim como um conjunto, as classes que implementam esta interface não podem conter elementos repetidos. Podem ser usadas implementações de *SortedSet* para situações onde for necessário ordenar os elementos;

List: Também chamada de seqüência. É uma coleção ordenada, que ao contrário da interface *Set*, pode conter valores duplicados. Além disso, permite o controle total sobre a posição onde se encontra cada elemento da coleção, podendo acessar cada um deles pelo índice.

Queue: Normalmente utilizada quando for necessário uma coleção do tipo FIFO (First-In-First-Out), também conhecida como fila.

Map: Utilizada quando for necessária uma relação de chave-valor entre os elementos. Cada chave pode conter apenas um único valor associado. Utiliza-se *SortedMap* para situações onde for necessário ordenar os elementos.

A seguir são apresentadas classes de implementação de cada interface:

| Interfaces | Implementations | | | | | |
|------------|-----------------|-----------------|---------|-------------|--------------------------|---------------|
| | Hash table | Resizable array | Tree | Linked list | Hash table + Linked list | |
| Set | HashSet | | TreeSet | | | LinkedHashSet |
| List | | ArrayList | | LinkedList | | |
| Queue | | | | | | |
| Map | HashMap | | TreeMap | | | LinkedHashMap |

11.1. HashSet

HashSet é uma classe de implementação para coleções da interface Set.

```
import java.util.*;
public class ExHashSet {
    public static void main(String[] args) {
        HashSet lista = new HashSet(); // criar um HashSet
        lista.add("Linha 1"); // Adiciona uma String na coleção
        lista.add(new Date()); // Adiciona uma data na coleção

        for(Object o : lista) //percorre a coleção recuperando os objetos
            System.out.println(o);
    }
}
```

11.2. ArrayList

ArrayList é uma classe de implementação para coleções da interface List.

```
import java.util.*;
public class ExArrayList {
    public static void main(String[] args) {
        ArrayList lista = new ArrayList(); // criar um ArrayList
        lista.add("Linha 1"); // Adiciona uma String na coleção
        lista.add(new Date()); // Adiciona uma data na coleção

        for(Object o : lista) //percorre a coleção recuperando os objetos
            System.out.println(o);
    }
}
```

11.3. Vector

Semelhante ao ArrayList, uma classe Vector também realiza uma implementação da interface List, podendo ser usada de forma semelhante ao ArrayList. O exemplo a seguir ilustra como utilizar uma classe Vector.

```
import java.util.*;
public class ExVector {
    public static void main(String[] args) {
        Vector lista = new Vector(); // criar um Vector
        lista.add("Linha 1"); // Adiciona uma String na coleção
        lista.add(new Date()); // Adiciona uma data na coleção

        for(Object o : lista) //percorre a coleção recuperando os objetos
            System.out.println(o);
    }
}
```

11.4. Iterator

Um Iterador é um objeto que fornece funcionalidade para iterar por todos os elementos de uma coleção. O exemplo a seguir ilustra como adicionar e remover objetos da classe ArrayList e o uso do Iterator para percorrer todo o ArrayList e imprimir todos os objetos.

```

import java.util.*;
public class ExArrayList {
    public static void main(String[] args) {
        ArrayList objetos = new ArrayList(); // criar um ArrayList

        objetos.add("Primeiro"); // Adiciona um objeto String
        objetos.add(new Date()); // Adiciona um objeto Date
        objetos.add(new Double(10.45)); // Adiciona um objeto Double

        Iterator it = objetos.iterator(); // objeto para percorrer o ArrayList
        while(it.hasNext()) {
            System.out.println("Lista -> "+it.next()); // imprime o objeto
        }
    }
}

```

O exemplo a seguir ilustra como adicionar e remover objetos da classe ArrayList. Os objetos utilizados são gerados a partir de uma classe Cliente.

```

import java.util.*;
public class ExArrayList2 {
    public static void main(String[] args) {
        ArrayList objetos = new ArrayList(); // cria o ArrayList
        Cliente c1 = new Cliente(); // cria o objeto c1 da classe Cliente
        Cliente c2 = new Cliente(); // cria o objeto c2 da classe Cliente
        c1.nome = "Cliente 1"; // inicializa atributo nome de c1
        c2.nome = "Cliente 2"; // inicializa atributo nome de c2

        objetos.add(c1); // adicionar c1 no ArrayList
        objetos.add(c2); // adicionar c2 no ArrayList
        Cliente cr1 = (Cliente) objetos.get(1); // recupera o objeto da pos 1
        System.out.println("O cliente é "+cr1.nome);

        Iterator it = objetos.iterator(); // para pegar todos os objetos
        Cliente cr2;
        while(it.hasNext()){
            cr2 = (Cliente) it.next(); // recupera o objeto da posição
            System.out.println("Lista -> "+cr2.nome);
        }
    }
    class Cliente { // definição da classe Cliente
        String nome; // atributo da classe cliente
    }
}

```

11.5. Métodos para Manipular Coleções

As classes de coleções possuem métodos que premitem realizam uma série de operações sobre a coleção, tais como:

add(Object obj): adicionar um objeto na coleção

addAll(Collection<Object> obj): adicionar uma sub-coleção de objetos na coleção

clear(): limpar o conteúdo de uma coleção;

remove(int index): remover um objeto pelo índice de sua posição;

remove(Object obj): remover um objeto pela sua referência;

11.6. Coleções Tipadas – Uso de Generics

Antes dos tipos genéricos (isto é antes do Java 5.0), o compilador não se importava com o que era inserido em um conjunto como o ArrayList por exemplo, porque todas as implementações de conjuntos eram declaradas para conter o tipo Object.

Como vimos nos exemplos acima, poderíamos inserir qualquer coisa em qualquer ArrayList, isso era possível por que ao armazenar o objeto no ArrayList ele o guardava em uma variável de referência do tipo Object, a qual como sabemos é a superclasse de que se derivam todas as demais classes do java, e justamente por causa deste “excesso de compatibilidade” dos objetos para com o ArrayList era necessária a conversão (cast) dos objetos retornados do ArrayList para que pudéssemos reutilizá-los.

Embora os tipos genéricos possam ser usados de outras maneiras, sua principal finalidade é permitir a criação de conjuntos com compatibilidade de tipos. Ou seja, dessa maneira o compilador o impede de inserir um objeto Dog em uma lista de objetos Duck por exemplo, e graças a declaração do tipo de objetos que nosso conjunto armazenará e já retornará o objeto correto sem a necessidade cast.

E qual a vantagem disso? Agora com os tipos genéricos, podemos inserir somente objetos Duck em um ArrayList<Duck> para que eles saiam com esse mesmo tipo de referência, deixando assim o programador livre de se preocupar com o fato de alguém inserir um objeto Dog na lista de patos, ou ainda de capturarmos algo que não possa ser convertido em uma referência de Duck, aumentando ainda mais a segurança da linguagem, pois todos os possíveis erros citados somente seriam descobertos em tempo de execução, e agora com os tipos genéricos todos estes problemas já são detectados na compilação do código.

Porem se você gosta de viver perigosamente ainda poderá declarar um ArrayList que possa armazenar qualquer tipo de objetos, o declarando como ArrayList<Object> que funcionará da mesma maneira que o antigo ArrayList.

No exemplo anterior sobre ArrayList, onde armazenamos um objeto Cliente em uma ArrayList antiga tínhamos que modelar para um formato de cliente todos os objetos que resgatávamos de nossa lista. Vejamos como ficaria nosso exemplo utilizando as novas listas implementadas a partir do java 5.

```
import java.util.*;
public class ExArrayList3 {
    public static void main(String[] args) {
        ArrayList<Cliente> objetos = new ArrayList<Cliente>(); // cria o ArrayList do tipo
        Cliente c1 = new Cliente(); // cria o objeto c1 da classe Cliente
        Cliente c2 = new Cliente(); // cria o objeto c2 da classe Cliente
        c1.nome = "Cliente 1"; // inicializa atributo nome de c1
        c2.nome = "Cliente 2"; // inicializa atributo nome de c2

        objetos.add(c1); // adicionar c1 no ArrayList<Cliente>
        objetos.add(c2); // adicionar c2 no ArrayList<Cliente>
        Cliente cr1 = objetos.get(1); // recupera o cliente, e o atribui sem a necessidade de
        System.out.println("O cliente é " + cr1.nome);
    }
}
class Cliente { // definição da classe Cliente
    String nome; // atributo da classe cliente
}
```

Mas lembre-se a classe ArrayList não foi a única que sofreu mudanças, outras classes de coleções como HashMap, TreeSet, LinkedHashMa, etc., também aderiram ao novo modelo, por isso para saber quais classes você pode utilizar com este novo formato aconselhamos consultar a documentação.

11.7. Exercício ArrayList

Altere o programa ExArrayList2 acrescentando na classe Cliente os atributos telefone e cidade, inicializando os atributos para ambos os clientes. Mostrar na listagem todos os atributos da classe Cliente.

11.8. Exercício usando coleção

Ler o Código, o Nome, Data de Nascimento e o Salário de 5 Pessoas. Ler um valor Inteiro que será o percentual de reajuste do salário que foi informado para cada pessoa.

Recalcular o salário e mostrar a relação atualizada com: Código, Nome, Data de Nascimento e Novo Salário.

12. A CLASSE CALENDAR

A classe abstrata Calendar encapsula um momento no tempo representado em milissegundos. Também provê métodos para manipulação desse momento. A subclasse concreta de Calendar mais usada é a GregorianCalendar que representa o calendário usado pela maior parte dos países.

Para obter um Calendar que encapsula o instante atual (data e hora), usamos o método estático **getInstance()** de **Calendar**. A partir de um Calendar, podemos saber o valor de seus campos, como ano, mês, dia, hora e minuto. Para isso, usamos o método **get()** que recebe um inteiro representando o campo.

Os valores possíveis estão em constantes na classe Calendar. Para trabalhar com dia, mês e ano, pode-se utilizar as constantes DAY_OF_MONTH, MONTH e YEAR respectivamente. Veja o exemplo a seguir:

```
import java.util.Calendar;
public class TestesCompleto {
    public static void main(String[] args) {
        // pegar uma instância de Calendar - pega a data atual do SO
        Calendar c = Calendar.getInstance();
        // imprimir dia/mes/ano
        System.out.println(c.get(Calendar.DAY_OF_MONTH)+"/"+
                           (c.get(Calendar.MONTH)+1)+"/"+
                           c.get(Calendar.YEAR));
    }
}
```

No exemplo acima, primeiramente é criada uma instância de Calendar chamada c. Em seguida, utiliza-se o método **get()** para obter o dia, mês e ano para imprimir a data. Note que o mês inicia em 0 para Janeiro até 11 para Dezembro.

Também é possível especificar uma data usando o método **set**. Veja o exemplo a seguir:

```
import java.util.Calendar;
public class EspecificarData {
    public static void main(String[] args) {
        // pegar uma instância de Calendar - pega a data do SO
        Calendar c = Calendar.getInstance(); // pegar data atual
        // setar dia/mes/ano para 11 de abril de 2007
        c.set(Calendar.DAY_OF_MONTH,11);
        c.set(Calendar.MONTH,3); // 0-Janeiro
        c.set(Calendar.YEAR,2007);
        System.out.println(c.get(Calendar.DAY_OF_MONTH)+"/"+
                           (c.get(Calendar.MONTH)+1)+"/"+
                           c.get(Calendar.YEAR)); // imprime dia/mês/ano
    }
}
```

É possível incrementar ou decrementar o dia, mês ou ano, usando o método **add()**, passando um inteiro positivo para incremento e negativo para decremento. Veja o exemplo a seguir:

```
import java.util.Calendar;
public class IncrementarDecrementar {
    public static void main(String[] args) {
        // pegar uma instância de Calendar - pega a data do SO
        Calendar c = Calendar.getInstance(); // pegar data atual
        // incrementar ou decrementar dia/mes/ano
        c.add(Calendar.DATE,-1); // decrementa um dia
        c.add(Calendar.MONTH,1); // incrementa um mês
        c.add(Calendar.YEAR,2); // incrementa dois anos
        System.out.println(c.get(Calendar.DAY_OF_MONTH)+"/"+
                           (c.get(Calendar.MONTH)+1)+"/"+
                           c.get(Calendar.YEAR)); // imprime dia/mês/ano
    }
}
```

Para trabalhar com hora, minuto e segundos, pode-se utilizar as constantes HOUR_OF_DAY, MINUTE e SECOND respectivamente. A constante HOUR_OF_DAY trabalhar a hora de 0 a 24. Também pode-se utilizar para hora as constantes HOUR e AM_PM para trabalhar no formato 0-12 AM/PM.

Veja o exemplo a seguir:

```
import java.util.Calendar;
public class ImprimiHoraMinSeg {
    public static void main(String[] args) {
        // pegar uma instância de Calendar - pega a data do SO
        Calendar c = Calendar.getInstance(); // pegar data atual
        // imprimir Hora:Min:Seg
        System.out.println(c.get(Calendar.HOUR_OF_DAY)+ ":" +
                           c.get(Calendar.MINUTE)+ ":" +
                           c.get(Calendar.SECOND));
        // ou
        String am_pm[] = {"AM", "PM"};
        System.out.println(c.get(Calendar.HOUR)+ ":" +
                           c.get(Calendar.MINUTE)+ ":" +
                           c.get(Calendar.SECOND)+ " " +
                           am_pm[c.get(Calendar.AM_PM)]);
    }
}
```

Também é possível setar, incrementar e decrementar hora, minuto e segundo como o uso do método add(). Veja o exemplo a seguir:

```
import java.util.Calendar;
public class SetarImcrementoDecremento {
    public static void main(String[] args) {
        // pegar uma instância de Calendar - pega a data do SO
        Calendar c = Calendar.getInstance(); // pegar data atual
        // setar hora, min e seg
        c.set(Calendar.HOUR,8); // 00-12
        c.set(Calendar.AM_PM,Calendar.PM); // PM
        // imprimir Hora:Min:Seg
        String am_pm[] = {"AM", "PM"};
        System.out.println(c.get(Calendar.HOUR)+ ":" +
                           c.get(Calendar.MINUTE)+ ":" +
                           c.get(Calendar.SECOND)+ " " +
                           am_pm[c.get(Calendar.AM_PM)]);
        // setar hora, min e seg
        c.set(Calendar.HOUR_OF_DAY,17); // 00-23
        c.set(Calendar.MINUTE,30); // 00-59
        // imprimir Hora:Min:Seg
        System.out.println(c.get(Calendar.HOUR_OF_DAY)+ ":" +
                           c.get(Calendar.MINUTE)+ ":" +
                           c.get(Calendar.SECOND));
    }
}
```

13. CONTROLE DE ERROS E TRATAMENTO DE EXCEÇÕES

Tratamento de exceções permite lidar com as condições anormais de funcionamento do programa. São exemplos de condições anormais: acesso a um índice inválido de um vetor, tentativa de uso de um objeto não inicializado, falha na transferência de uma informação, uma falha não prevista, etc.

Fazer uso deste recurso tornará o software mais robusto, seguro e bem estruturado, pois as exceções podem ser tratadas incluindo-se código adequado no programa; não são portanto erros fatais.

Naturalmente, exceções são objetos em Java. A classe `Exception` é a superclasse de todas as exceções. Existem vários tipos de exceções já definidos nas bibliotecas. É possível também construir as próprias exceções.

13.1. Estrutura Básica do controle de exceção

Para capturar uma exceção, é necessário no mínimo a seguinte estrutura de código:

```
try {
    // aqui vai código que pode gerar exceções
    // Exemplo: divisão por 0, erro de conexão, etc.
}
catch( Exception e ) {
    // aqui vai código para lidar com uma exceção
    // somente será executado se der erro em alguma instrução no try
}
```

No bloco `try` devem ser colocadas todas as instruções que podem gerar algum erro a ser tratado. Caso ocorra o erro, a execução do bloco é interrompida na linha que gerou o erro e é executado o bloco `catch`.

A o tipo de exceção gerada é recuperada pelo objeto da classe `Exception` ou alguma de suas subclasses.

Algumas classes da API do Java exigem que sejam utilizadas dentro de um controle de exceção, implementando a classe `Throwable`. Essas classes só poderão ser utilizadas se estiverem dentro de uma estrutura `try`. Exemplo: classes de conexão de rede, banco de dados, leitura e escrita de arquivos, etc.

13.2. Exemplo de uma estrutura básica

```
import javax.swing.JOptionPane;
public class Exemplo1 {
public static void main(String[] args) {
    int a = 0;
    int b = 5;
    try {
        int x = b / a;
        JOptionPane.showMessageDialog(null, "Valor de x : "+x);
    } catch (Exception e) {
        JOptionPane.showMessageDialog(null, "Erro: "+e.getMessage());
    }
    System.exit(0);
}}
```

13.3. Classes Throwable e Exception

Objetos que podem ser lançados para indicar que algo anormal aconteceu, e capturados para lidar com esta situação, devem estender a classe Throwable do pacote java.lang. Os métodos mais úteis desta classe são:

Método para retornar mensagem de erro:

```
public String getMessage()
```

Método para imprimir uma descrição da pilha no instante em que o problema ocorreu:

```
public void printStackTrace()
```

Existe um grande número de subclasses de Exception espalhadas pelos vários pacotes de biblioteca.

O programador pode estender a classe Exception ou uma das suas subclasses para construir as suas próprias exceções.

13.4. Estrutura Básica do controle de exceção com finally

Finally é um bloco da estrutura de exceção que é executado em ambas as situações. Dando erro no bloco try ou não.

```
try {
    // aqui vai código que pode gerar exceções
    // Exemplo: divisão por 0, erro de conexão, etc.
}
catch( Exception e ) {
    // aqui vai código para lidar com uma exceção
    // somente será executado se der erro em alguma instrução no try
}
finally {
    // não é obrigatório ter este bloco
    // aqui vai o código que deve ser executado em qualquer caso
    // em ambos os casos este bloco será executado
}
```

13.5. Exemplo com uso do finally

```
import javax.swing.JOptionPane;
public class Exemplo2 {
public static void main(String[] args) {
    int a = 0;      int b = 5;
    try {
        int x = b / a;
        JOptionPane.showMessageDialog(null, "Valor de x : "+x);
    } catch (Exception e) {
        e.printStackTrace();
        JOptionPane.showMessageDialog(null, "Erro: "+e.getMessage());
    } finally {
        JOptionPane.showMessageDialog(null,"Saiu do controle de exceção");
    }
    System.exit(0);
}}
```

13.6. Captura de exceções com múltiplos blocos catch

Para capturar diferentes tipos de exceção que podem ser geradas em um mesmo bloco try, é necessário montar a seguinte estrutura de código:

```
try {
    // aqui vai código que pode gerar exceções dos tipos
    // SQLException e IOException
}
catch( SQLException esql ) {
    // aqui vai código para lidar com uma exceção do tipo
    // SQLException
}
catch( IOException eio ) {
    // aqui vai código para lidar com uma exceção do tipo
    // IOException
}
finally {
    // aqui vai código que deve ser executado sempre
}
```

13.7. Lançamento de exceções definidas pelo usuário

Pode-se definir exceções próprias dentro das classes e dispará-las em determinadas ocasiões dentro dos métodos. Uma exceção pode ser definida da classe Exception e lançada dentro de um método usando-se a palavra chave **throw** seguida da referência a exceção a ser gerada.

Exemplo:

```
// Declara a exception erro1
Exception erro1 = new Exception( "deu zebra!" );
...
// lança a exception erro1
if( Deu_Erro ) throw erro1;
```

13.7.1. Captura de exceções definidas pelo usuário

Para capturar uma exceção definida pelo usuário, é necessário montar a seguinte estrutura de código.

```
try {
    // aqui vai código que pode gerar exceções dos tipos
    // ExceptionType1 e ExceptionType2
}
catch( ExceptionType1 erro1 ) {
    // aqui vai código para lidar com uma exceção do tipo
    // ExceptionType1
}
catch( ExceptionType2 erro2 ) {
    // aqui vai código para lidar com uma exceção do tipo
    // ExceptionType2
}
finally {
    // aqui vai código que deve ser executado em qualquer caso
}
```

13.7.2. Exemplo com exceção definida pelo usuário

```
import javax.swing.JOptionPane;
public class Exemplo3 {
    public static void testeException(boolean a) throws Exception {
        Exception errol = new Exception( "deu problema! " );
        if( a )
            throw errol;
        else
            JOptionPane.showMessageDialog(null, "Primeira chamada. Não gerou exceção");
    }
    public static void main(String[] args) {
        try {
            testeException(false); // chamada 1
            testeException(true); // chamada 2
        } catch (Exception e) {
            e.printStackTrace();
            JOptionPane.showMessageDialog(null, e.getMessage());
        } finally {
            JOptionPane.showMessageDialog(null,"Saiu do controle de exceção");
        }
    }
}
```

14. PROGRAMAÇÃO ORIENTADA A OBJETOS NO JAVA

O Java é uma linguagem orientada a objetos. Segundo (Hubbard, 2006, pg125) “Seus programas são organizados por meio de classes, as quais especificam o comportamento dos objetos, que controlam as ações do programa”. Desta forma, serão apresentados a seguir os principais recursos de programação orientada a objetos no Java.

14.1. Pacotes

Pacotes ou packages são conjuntos de declarações de classes e interfaces. Todo programa Java é organizado como um conjunto de pacotes. A maior parte dos sistemas armazena os pacotes em sistemas de arquivos, mas também há previsão para que os pacotes sejam armazenados em outros tipos de bases de dados.

Um pacote é composto pelos arquivos de programas Java. Cada arquivo que pertence ao mesmo pacote deve obrigatoriamente começar com a declaração do pacote, como mostra o exemplo a seguir:

```
package meuExemplo;
public class MinhaClasse{
    public static int contador;
    Public static void acao(){
    }
}
```

Todos os nomes declarados em unidades deste pacote devem ser precedidos do nome do próprio pacote, separado por ponto para serem utilizados.

Os nomes completos para acesso a essa classe são:

```
meuExemplo.MinhaClasse
meuExemplo.MinhaClasse.contador
meuExemplo.MinhaClasseacao()
```

Quando for necessário referenciar a variável contador, num método de uma outra classe, o correto é escrever como mostra o exemplo a seguir:

```
package outroPacote;
public class Outra{
    void metodoA(){
        meuExemplo.MinhaClasse.contador++;
    }
}
```

Para não escrever o nome completo, pode-se usar a declaração **import** nas classes que utilizarão o pacote. Dessa forma não será necessário escrever o caminho completo para usar as classes. Veja o exemplo que segue:

```
package outroPacote;
import meuExemplo.*; // disponibiliza todas as classes do pacote para uso
public class Outra{
    void metodoA(){
        MinhaClasse.contador++;
    }
}
```

O import permite que o programador não tenha que escrever o nome completo em todas as ocorrências do nome no texto do programa. Ao fazer **import pacote.***, todas as classes do pacote são disponibilizadas. Se desejar importar somente uma classe do pacote, pode-se usar **import pacote.Classe**.

14.2. Classes

Um programa em Java, assim como outros programas nas diversas linguagens, é uma arquivo texto, o qual possui ao menos uma definição de classe e um método main public, conforme o exemplo a seguir:

```
public class Programa1{  
    public static void main(String args []){  
        // Aqui vem os comandos  
    }  
}
```

Para definir uma classe use a palavra chave class e o nome da classe. Exemplo:

```
class Minhaclasse{  
...  
}
```

Para fazer herança de outra classe, permitindo definir a classe como subclasse de outra classe, utiliza-se extends para indicar a superclasse.

Exemplo:

```
class Minhaclasse extends SuperClasse{  
...  
}
```

Para fazer implementar uma interface, utiliza-se implements para indicar a classe de interface a ser implementada.

Exemplo:

```
class Minhaclasse implements SuperClasse{  
...  
}
```

14.2.1. Atributos / Variáveis de Instância

Os atributos, ou variáveis de instância, aparentemente, são declaradas e definidas quase exatamente da mesma forma que as variáveis locais em métodos, a principal diferença é que são definidos na classe.

Em uma classe podem ser declarados quantos atributos forem necessários para tratar todos os dados do objeto. A sintaxe de declaração de atributos tem seis partes básicas, como pode ser visto a seguir:

- Visibilidade
- Modificador
- Tipo do Atributo;
- Nome do Atributo;
- Inicialização

Exemplo:

```
Public class Bicicleta {  
    String tipo;  
    private int correia = 1;  
    public static int pedal;  
}
```

14.2.2. Métodos

Os métodos no Java são pequenos trechos de código localizados internamente em uma classe, em outras linguagens de programação são conhecidos como funções, procedimentos, subrotinas , os quais desempenham alguma ação.

Toda classe que julga-se por principal tem um método main, com o seguinte formato:

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

Este é o formato padrão do método main, onde **public**, quer dizer que pode ser acessado por todos os seus membros, **static** a mesmas variáveis pode ser usadas para todas as instâncias da classe, **void**, significa que este método não retonará valores, **main**, o nome do método principal, **String**, vai receber como argumentos vetor do tipo String chamado args. O colchete poderá ser posto tanto em **String[]** quanto no **args[]**.

Em uma classe podem ser declarados quantos métodos forem necessários para tratar todas as operações do objeto. Um método pode ou não ter parâmetros, assim como pode ou não ter valor de retorno. Quando um método não tem retorno, deve-se especificar **void** como retorno. Quando o método retorna algum tipo de dado, utiliza-se a palavra **return** no corpo do método para retornar o valor.

A sintaxe de declaração de um método têm seis partes básicas, como pode ser visto a seguir:

- Visibilidade
- Modificador
- O tipo de retorno;
- O nome do método;
- Uma lista de parâmetros;
- O corpo do método.

A definição básica de um método tem esta aparência:

```
[Visibilidade][Modificador]TipoDeRetorno nomeDoMetodo(tipo1 arg1, tipo2 arg2, ...){  
    // corpo do método ... implementação  
}
```

Exemplo:

```
public int soma(int a, int b) {  
    return a + b;  
}
```

A seguir será demonstrado um exemplo de um programa que testa um método chamado cubo que devolve o cubo do inteiro para ele:

```
package meupacote;
public class TesteCubo {
    public static void main(String[] args) {
        for (int i = 0; i < 6; i++)
            System.out.println(i + "\t" + cubo(i));
    }
    static int cubo(int n) {
        return n*n*n;
    }
}
```

Saída:

```
0      0
1      1
2      8
3      27
4      64
5      125
```

14.2.3. Método Construtor

Um método construtor é um método como outro qualquer, com a diferença de ser automaticamente executado quando o objeto é criado. Uma das principais funções dos métodos construtores é inicializar os atributos do objeto, ou seja, definir os valores iniciais dos atributos do objeto.

Dentro do método construtor, pode-se além de inicializar os atributos do objeto, realizar outras tarefas para inicializar o objeto. Por exemplo, se sua classe representa uma impressora, você pode verificar se existe uma impressora conectada ao seu micro, etc.

Características de métodos construtores:

- Tem o mesmo nome da classe;
- Pode ou não ter argumentos;
- Não deve retornar nenhum tipo de valor, nem mesmo void;
- Não se pode chamar um construtor. Somente ao instanciar o objeto;
- Os construtores podem ser sobrecarregados;
- São sempre públicos.

Exemplo de uso do método construtor:

```
public class Data {
    private int dia, mes, ano;
    public Data(int dial, int mes1, int anol) {
        dia = dial; mes = mes1; ano = anol;
    }
}

Data d1 = new Data(); // construtor vazio
Data d1 = new Data(1,1,2005); // construtor parametrizado
```

14.3. Objetos

Em um programa em Java os objetos são a instância de uma classe, ou seja, são eles que fazem a ação de uma classe, de acordo com o seu procedimento acontecerá algum evento para aquela classe.

Para criar uma instância de uma classe (criar o objeto), pode-se usar a sintaxe a seguir:

```
NomeDaClasse nomeObjeto; // declara o objeto da classe
nomeObjeto = new NomeDaClasse(); // instancia o objeto
```

Essa mesma instrução também pode ser feita em uma única linha, como a seguir:

```
NomeDaClasse nomeObjeto = new NomeDaClasse(); // declara e instancia
```

Aparece duas vezes o nome da classe, no entanto, após o new, é a chamada ao método construtor da classe.

No exemplo a seguir é apresentada uma classe chamada Pessoa, que será utilizada por uma outra classe chamada Professor:

Exemplo 5

```
package meupacote;

public class Pessoa {
    private String nome;
    private String sobreNome;

    public Pessoa (){

    }

    public void setNome(String nome){
        this.nome = nome;
    }
    public void setSobreNome (String sobreNome){
        this.sobreNome = sobreNome;
    }
    public String getNome () {
        return nome;
    }

    public String getSobreNome() {
        return sobreNome;
    }
}
```

```
package meupacote;

public class Professor {
    public static void main(String[] args) {
        Pessoa p = new Pessoa();
        p.setNome("Fulano");
        p.setSobreNome("De Tal");
        System.out.println("Nome: " + p.getNome());
        System.out.println("Sobre Nome: " + p.getSobreNome());
    }
}
```

As classes representadas acima estão no mesmo pacote, chamado meupacote, e a primeira classe chamada Pessoa tem dois métodos get e dois métodos set para pegar e setar os valor dos atributos.

Na classe Professor, a classe Pessoa é instanciada para criar um objeto chamado p e terá por objetivo setar através dos métodos set um nome e um sobre nome para o professor e após pegar estes valores através dos métodos get.

A saída gerada pelo programa é demonstrada a seguir:

```
Nome: Fulano
Sobre Nome: De Tal
```

14.4. Escopo de Variáveis, Atributos e Métodos

O escopo determina o nível de acessibilidade de variáveis, atributos e métodos dentro de um programa em Java, dessa forma alguns conceitos são aplicados especificamente para um atributo e para um método, de modo a melhor conceituar os limites de acesso para cada um.

14.4.1. Escopo de Variável

Em Java uma variável só é válida no contexto onde foi declarada, ou seja, se uma variável é declarada dentro do método, esta será acessível somente dentro do método, tais variáveis são chamadas locais. Elas existem somente durante a execução daquele método ou bloco que a variável, porém se declarada dentro da classe (como atributo) ela vale na classe inteira incluindo no método. Um exemplo de variável local pode ser demonstrado no trecho do código a seguir:

Exemplo1:

```
int i = 10;
if (i > 0) {
    int j = 15;
    System.out.println(i + j); // 25
}
j= i + j; // Erro: variável está fora de escopo!
```

A variável j está declarada dentro de um bloco if , então a variável terá seu acesso válido somente entre as chaves {} do comando if. Fora deste escopo, ou seja, fora do local onde a mesma foi declarada, não terá um acesso válido.

14.4.2. Escopo de Atributos

As variáveis que são declaradas fora de qualquer método (usualmente no cabeçalho da classe), são acessíveis por qualquer método da classe. Tais variáveis são chamadas de atributos e são globais a classe em que estão declaradas. O escopo de um atributo pode ser melhor compreendido no exemplo a seguir:

Exemplo 2:

```
public class NomeDaClasse{
    private TipoDoAtributo atributo1;
    ...
    public TipoDoAtributo1 getAtributo1(){
        return atributo1;
    }
    public void setAtributo1( TipoDoAtributo1 valor ){
        /* aqui pode-se colocar testes para determinar se o valor é aceitável */
        ...
        atributo1 = valor;
    }
}
```

No exemplo acima o atributo de nome atributo1 , esta declarado no cabeçalho da classe, então, será acessível em qualquer dentro dessa classe. Assim, o escopo de um atributo é a classe inteira.

Também é possível controlar o escopo dos atributos através de restrição de acesso, para isso algumas palavras-chaves são disponíveis para ampliar ou restringir o acesso a um atributo. Estas palavras-chaves são acrescentadas à declaração do atributo.

- **public:** pode ser acessado por qualquer outra classe.

Exemplo:

```
public String nome;
```

- **private:** pode ser acessado somente por métodos da própria classe.

Exemplo:

```
private String endereco;
```

- **protected:** além de poder ser acessado por todas as classes do mesmo pacote, também pode ser acessado pelas subclasses da classe na qual.

Exemplo:

```
protected int numero;
```

- **default:** Um atributo para o qual não foi especificada nenhuma destas palavras-chaves apresentadas e que pode ser chamado por todas as classes que pertencem ao mesmo pacote. Este modo de acesso é também chamado de “friendly” ou “package”.

Exemplo:

```
int contador;
```

14.4.2.1 Modificadores de Atributos

Existem ainda outros modificadores para atributos.

static: indica que o atributo é compartilhado por todas as instâncias da classe. O atributo pode ser acessado diretamente na classe, mesmo sem haver a instância;

final: impossibilita que o atributo tenha seu valor alterado. Para atributos, é necessário fornecer um valor na hora de declará-lo. Atributos de interface devem ser sempre final e precisam ter seu valor declarado;

transient: é aplicável apenas a atributos, e faz a JVM ignorar o atributo quando tenta serializar o objeto;

volatile: indica que um thread acessando o atributo precisa sempre sincronizar o valor de sua cópia local com a cópia principal da memória. Tem seu valor alterado assincronamente por segmentos que estejam sendo executados ao mesmo tempo. É diferente do static, pois o static vale para todas as instâncias, já o volatile, vale somente para as threads associadas a instâncias do objeto.

14.4.3. Escopo de Métodos

Os métodos em Java definem o seu escopo através de restrição de acesso, para isso algumas palavras-chaves são disponíveis para ampliar ou restringir o acesso a um método. Estas palavras-chaves são acrescentadas à declaração do método.

- **public:** pode ser acessado por qualquer outra classe.

Exemplo:

```
public String retornaNome (){  
}
```

- **private:** Um método que pode ser acessado somente por métodos da própria classe.

Exemplo:

```
private String retornaNome (){  
}
```

- **protected:** Um método que, além de poder ser acessado por todas as classes do mesmo pacote, também pode ser acessado pelas subclasses da classe na qual ele é declarado.

Exemplo:

```
protected int retornaInt (){  
}
```

- **default:** Um método para o qual não foi especificada nenhuma destas palavras-chaves apresentadas e que pode ser chamado por todas as classes que pertencem ao mesmo pacote.

Exemplo:

```
int retornaInt (){  
}
```

14.4.3.1 Modificadores de Métodos

Existem ainda outros modificadores de métodos.

static: indica que o método pode ser utilizado sem a necessidade de instanciar o objeto;

final: impede que o método seja sobreescrito (overriding) na subclasse. Ele também pode ser utilizando nos argumentos do método, significando que estes não poderão ter seu valor ou referência alterados.

abstract: indica que o método é abstrato, não implementado. Neste caso é necessário utilizar ponto e vírgula ao invés das chaves na sua declaração. Existindo algum método abstrato, a classe também deve ser declarada abstrata. A primeira subclasse concreta será obrigada a implementar os métodos abstratos, seguindo exatamente a mesma assinatura do método. Métodos abstratos não podem ser static, final, private, synchronized, strictfp ou native.

synchronized: só pode ser acessado por uma thread de cada vez.

14.5. Encapsulamento

O encapsulamento caracteriza-se por ser um mecanismo que permite fazer parte do funcionamento de uma interface. Um exemplo prático de encapsulamento é o de um liquidificador, não precisamos saber detalhes do funcionamento de seu motor, o que se conhece são os botões, pois são eles que fazem o motor funcionar.

Uma grande vantagem do encapsulamento é que toda parte encapsulada pode ser modificada sem que os usuários da classe em questão sejam afetados. Ainda no exemplo do liquidificador, um técnico poderia substituir o motor do equipamento por um outro totalmente diferente, sem que alguém perceba,

continuando com o mesmo procedimento de pressionar o botão, além de proteger o acesso direto, ou seja a referência aos atributos de uma instância fora da classe onde estes foram declarados.

Esta proteção consiste em se usar modificadores de acesso mais restritivos sobre os atributos definidos na classe. Depois devem ser criados métodos para manipular de forma indireta os atributos da classe. Encapsular atributos também auxilia a garantir que o estado e o comportamento de um objeto se mantenha coeso.

Para melhor ilustar a seguir será demonstrado um exemplo prático:

Exemplo 1 sem encapsulamento:

```
class NaoEncapsulado {  
    int semProtecao;  
}  
  
public class TesteNaoEncapsulado {  
    public static void main(String[] args) {  
        NaoEncapsulado ne = new NaoEncapsulado();  
        ne.semProtecao = 10; //acesso direto ao atributo  
        System.out.println("Valor sem proteção: " + ne.semProtecao);  
    }  
}
```

No exemplo acima não está sendo usada nenhum tipo de modificador de acesso como private, protected, não garantindo a percedência dos objetos.

Exemplo 2 com encapsulamento:

```
class Encapsulado {  
    private int comProtecao;  
  
    public void setComProtecao(int comProtecao) {  
        this.comProtecao = comProtecao;  
    }  
  
    public int getComProtecao() {  
        return this.comProtecao;  
    }  
}  
  
public class TesteEncapsulado {  
    public static void main(String[] args) {  
        Encapsulado e = new Encapsulado();  
        e.comProtecao = 10; // dá erro !  
        e.setComProtecao(10); // assim permite acessar pelo método  
        System.out.println("Valor com proteção: " + e.getComProtecao());  
    }  
}
```

No exemplo 2 é usado modificadores garantindo a coesão entre os objetos.

14.6. Sobrecarga de Métodos

O Java permite que tenha métodos com o mesmo nome, mas com assinaturas diferentes, isto chama-se sobrecarga. O interpretador determinará qual método deve ser invocado pelo tipo de parâmetro passado. O exemplo a seguir demonstra como pode se fazer uma sobrecarga de métodos.

Exemplo:

```
public void imprime( int i ) { ... }
public void imprime ( float f ) { ... }
public void imprime ( String s) { ... }
```

Neste caso métodos com o mesmo nome, definidos dentro de uma mesma classe, podem receber três valores diferentes, um inteiro um float e uma string.

14.7. Herança

Uma das maiores vantagens da OO reside na possibilidade de haver herança entre classes. A herança consiste na capacidade de construir novas classes a partir de outras existentes. Quando há herança, os atributos e métodos de uma classe existente, chamada superclasse, são herdados pela nova classe, chamada subclasse ou classe derivada. Para ser possível que uma classe extenda uma outra qualquer, tem se que utilizar o a palavra **extends**:

Exemplo:

```
public class Classe2 extends Classe1
```

Quando há herança, trabalha-se com os conceitos de superclasse e subclasse, sendo:

superclasse: a classe que está sendo herdada, no exemplo acima a Classe1;

subclasse: a classe que herda, no exemplo anterior a Classe2.

Novos atributos e métodos podem ser adicionados na subclasse. No exemplo a seguir será demonstrado um exemplo com duas classes uma chamada Conta e a outra chamada ContaEspecial, que herda Conta, ou seja, que além de suas próprias cacterísticas possui também as características da classe Conta.

Classe Conta:

```
package meupacote.heranca;

public class Conta {
    public String titular;
    protected double saldo;
    private int controle;
    public void deposito(double vlr) {
        saldo += vlr;
    }
    public boolean saque(double vlr) {
        if (saldo > vlr){
            saldo -= vlr;
            return true;
        }else
            Return false;
    }
}
```

Classe ContaEspecial:

```
package meupacote.heranca;

public class ContaEspecial extends Conta {
    private double limite;
    static private double taxaJuro;
}
```

superclasse: Conta;

subclasse: ContaEspecial.

A classe ContaEspecial esta herdando os seguintes atributos e métodos, da classe Conta:

Atributos: titular, saldo

Métodos: deposito e saque

14.7.1. Substituição / Sobrescrição

O Java permite que métodos herdados sejam sobreescritos na subclasse, substituindo o que foi herdado por um método com novas funcionalidades na programação. Para haver substituição, deve-se reescrever o método herdado na subclasse, com o mesmo nome e assinatura.

Por exemplo, se desejar substituir na classe ContaEspecial o método saque herdado da classe conta para que este considere o limite para permitir ou não o saque, poderia ser feito como no código a seguir:

```
package meupacote.heranca;

public class ContaEspecial extends Conta {
    private double limite;
    static private double taxaJuro;

    public boolean saque(double vlr) {
        if ((saldo + limite) > vlr){
            saldo -= vlr;
            return true;
        }else
            Return false;
    }
}
```

Neste caso, ao chamar o método saque de uma instância de ContaEspecial, será executado o método que foi sobreescrito, e não o que foi herdado.

14.7.2. Especificação super

A palavra **super** provê acesso a partes de uma superclasse a partir de uma subclasse. É muito útil ao sobrepor um método, pois pode-se ainda fazer uso do método escrito na superclasse o que em alguns casos elimina a necessidade de reescrevê-lo por completo.

O exemplo poderia ser escrito também da seguinte forma.

```
package meupacote.heranca;

public class ContaEspecial extends Conta {
    private double limite;
    static private double taxaJuro;

    public boolean saque(double vlr) {
        if (super.saque(vlr) == false){ //tenta usar o saque da superclasse
            // se não passar, faz os testes para o limite
            ...
        }
    }
}
```

14.7.3. Especificação this

A palavra **this** provê uma referência a própria classe. No exemplo a seguir, o método setLimite recebe uma variável limite como parâmetro, que deve ser atribuído ao atributo da classe chamado limite. Dentro do método, ambos, variável recebida por parâmetro e atributo da classe são visíveis, porém os dois

possuem o mesmo nome. Então, pode-se referenciar ao atributo da classe adicionando a palavra this na frente.

```
package meupacote.heranca;

public class ContaEspecial extends Conta {
    private double limite;
    static private double taxaJuro;

    public boolean setLimite(double limite) {
        this.limite = limite;
    }
}
```

Assim, o valor do parâmetro limite recebido pelo método será atribuído ao atributo da classe chamado limite.

14.8. Agregação

Agregação é a criação de uma classe usando outra classe como seu componente de dados. A instância da classe que é agregada na nova classe é criada fora do contexto da classe atual.

Exemplo:

```
public class Nome {
    protected String nome;

    public Nome (String nome) {
        this.nome = nome;
    }
}
```

Usando a classe:

```
public class UsaNome {
    public static void main(String arg[]){
        String a = "Paulo Silva";
        Nome n = new Nome( a );
    }
}
```

A classe Nome possui um método construtor que recebe um objeto String, que será “agregado” a classe UsaNome pelo construtor. Ao destruir o objeto da classe Nome, o objeto String continuará a existir, pois sua instância foi criada fora do objeto da classe Nome.

14.9. Composição

Composição é a criação de uma classe usando outra classe como seu componente de dados. A instância da classe que é agregada na nova classe é criada dentro do contexto da classe atual.

Exemplo:

```
public class Nome {
    protected String nome;

    public Nome (String nome) {
        this.nome = nome;
    }
}
```

Usando a classe:

```
public class UsaNome {  
    public static void main(String arg[]){  
        Nome n = new Nome("Paulo Silva" );  
    }  
}
```

A classe Nome possui um método construtor que recebe um objeto String, que será “agregado” a classe UsaNome pelo construtor. Ao destruir o objeto da classe Nome, o objeto String deixará de existir, pois sua instância foi criada no contexto do objeto da classe Nome.

14.10. Polimorfismo

Polimorfismo é a capacidade de assumir formas diferentes, em termos de programação, representa a capacidade de um objeto poder ser referenciado de várias formas. Não quer dizer que o objeto fica se transformando, pelo contrário, um objeto nasce de um tipo e morre daquele tipo, o que muda é a maneira como referenciamos a ele.

Outra característica de polimorfismo reside no fato de que “as subclasses são consideradas do mesmo tipo de sua superclasse”. O polimorfismo permite a realização de uma mesma operação sobre diferentes tipos de classes, desde que mantenham algo em comum.

Por exemplo, considere a classe **Conta** e suas derivadas **ContaAplicacao** e **ContaEspecial**. Apesar de **ContaAplicacao** e **ContaEspecial** serem diferentes, elas ainda são consideradas contas. Assim, qualquer coisa que fosse permitido fazer com uma instância da classe **Conta**, seria também permitida para as instâncias das classes **ContaEspecial** e **ContaEmprestimo**.

Exemplo:

```
public class Polimorfismo {  
    public static void main(String[] args) {  
        Conta conta1, conta2, conta3;  
        conta1 = new conta();  
        conta2 = new ContaEspecial(); // Isto é válido, pois contaespecial é uma conta  
        conta3 = new ContaAplicacao(); // Isto é válido, pois contaaplicacao é uma conta  
    }  
}
```

Dessa forma usa-se uma variável de um tipo único (normalmente do tipo da superclasse) para referenciar objetos variados do tipo das subclasses, envolve o uso automático do objeto armazenando na super-classe para selecionar um método de uma das subclasses. A escolha do método a ser executado é feita dinamicamente em tempo de execução.

Polimorfismo é uma das ferramentas mais poderosas da programação orientada a objetos, além de fundamental para o paradigma de Orientação a Objetos.

14.11. O Operador instanceof

O operador instanceof é um mecanismo de RTTI (Runtime Type Information - Informação dinâmica de tipo) que permite consultar se um objeto é de uma determinada classe.

Sintaxe de uso do operador instanceof:

```
<objeto> instanceof <Classe>
```

Retorna true se o objeto for instância (direta ou indireta) da classe especificada, retorna false caso contrário.

Exemplo:

```
class Arvore{}
class Pinheiro extends Arvore{}
class Carvalho extends Arvore{}
public class TestaClasse {
    public static void main( String[] args ) {

        Arvore a1 = new Pinheiro();

        if( a1 instanceof Pinheiro )
            System.out.println( "É da classe Pinheiro" );
        if( a1 instanceof Arvore )
            System.out.println( "É da classe Árvore" );
        if( a1 instanceof Carvalho )
            System.out.println( "É da classe Carvalho" );
    }
}
```

14.12. Interfaces e Classes Abstratas

Interfaces e Classes abstratas são conceitos parecidos mas apresentam características específicas: Uma classe pode estender uma única classe (que pode ser abstrata ou não), mas pode implementar várias interfaces.

A seguir são apresentadas as diferenças entre interfaces e classes abstratas.

14.12.1. Interface

Java não permite que uma classe herde recursos de mais de uma outra classe, ou seja herança múltipla. Por outro lado, a herança múltipla é um recurso poderoso, cujo total abandono limitaria bastante a linguagem.

A saída encontrada é o conceito de interface, que nada mais é que um conjunto de declarações de métodos desprovidos de implementação. Cabe ao programador que deseja implementar a interface em questão providenciar uma implementação destes métodos na classe que ele está desenvolvendo.

Desta forma, além de estender alguma superclasse, a classe em desenvolvimento pode implementar várias interfaces.

Interface é uma classe que não pode ser instanciada, tem por objetivo especificar um conjunto de requisitos que devem ser implementados pela classe que a implementa. A palavra chave **implements** é utilizada para indicar que uma classe implementa uma determinada interface.

Uma interface diferencia-se de uma classe nos seguintes pontos:

- Declara somente cabeçalhos de métodos e constantes public;
- Não possui contrutores;
- Não pode ser instanciada;
- Pode ser implementada por uma classe;
- Não pode implementar uma interface;
- Não pode estender uma classe;
- Ao criar uma classe usando uma interface, o corpo dos métodos da interface devem ser escritos para que façam alguma operação.
- Pode estender várias outras interfaces.

Exemplo de uma classe Interface:

```
interface Funcionario{
    public double reajusteSalario(double percentual);
}
```

Pode-se então definir uma classe gerente que implementa esta interface:

```
public class Gerente implements Funcionario{
    public double reajusteSalario(double percentual){
        // programar o cálculo
    }
}
```

14.12.2. Classes Abstratas

A linguagem Java ainda oferece um meio termo entre uma classe normal e uma interface na qual nenhum dos métodos declarados está implementado. Trata-se da classe abstrata, na qual alguns dos métodos declarados estão implementados e outros não.

Algumas características de classes abstratas:

- Uma classe abstrata deve ser declarada tal usando-se a palavra chave **abstract**;
- Uma classe abstrata não pode ser instanciada diretamente;
- Pode herdar outra classe;
- Pode-se construir subclasses da classe abstrata, nas quais os métodos declarados mas ainda não implementados devem receber uma implementação.

Exemplo: uma alternativa ao exemplo anterior seria definir uma classe abstrata Forma:

```
abstract Funcionario{
    // Método Abstrato
    public abstract double reajusteSalario(double percentual);

    // Método não abstrato
    public double getSalario(){
        // implementação do método
        return salário;
    }
}
```

Métodos Abstratos: Em uma classe abstrata, podemos escrever que determinado método será sempre escrito pelas classes filhas. Isto é, um método abstrato.

Ele indica que todas as classes filhas (concretas, que não forem abstratas) devem reescrever esse método, ou não compilão. É como se você herdasse a responsabilidade de ter que escrever aquele método.

```
public class Gerente implements Funcionario{
    public double reajusteSalario(double percentual){
        // programar o cálculo
    }
}
```

14.13. Java Beans

JavaBeans são definidos como sendo componentes de software escritos na linguagem de programação Java.

Para ser considerado um JavaBean, uma classe precisa seguir algumas convenções de nomenclatura de métodos, construtores e comportamento. Estas convenções permitem a existência de ferramentas que podem utilizar e manipular os JavaBeans.

As convenções definem que a classe:

- Deve implementar a interface `java.io.Serializable` (que possibilita a persistência e restauração do estado do objeto da classe);
- Deve possuir um construtor sem argumentos;
- Seus atributos devem estar encapsulados com `private`;

- Seus atributos devem ser acessíveis através de métodos "get" e "set", seguindo um padrão de nomenclatura.

Exemplo:

```
public class JavaBeanPessoa implements java.io.Serializable {
    private String nome;
    private int idade;

    // O constructor sem argumentos
    public JavaBeanPessoa() {
    }

    public void setNome(String nome) {
        this.nome = nome;
    }

    public void setIdade(int idade) {
        this.idade = idade;
    }

    public String getNome() {
        return this.nome;
    }

    public int getIdade() {
        return this.idade;
    }
}
```

14.14. Recursividade

Recursividade é a capacidade de um método fazer uma chamada a ele mesmo. Veja no exemplo abaixo:

```
public long fatorial( long number ) {
    if ( number <= 1 )
        return 1;
    else
        return number * fatorial( number - 1 );
}
```

A execução desse método irá calcular o fatorial do número passado na chamada ao método. Ao chamar **fatorial(5)**, o método fatorial será chamado recursivamente até que o valor passado como argumento seja ≤ 1 . A execução pode ser interpretada como ilustrado a seguir:

| | | | | | | | | | |
|----------|--|----|--------|----|-------|----|-------|----|---|
| Chamada: | fatorial(5) -> fatorial(4) -> fatorial(3) -> fatorial(2) -> fatorial(1) | | | | | | | | |
| Retorno: | 5*24=120 | <- | 4*6=24 | <- | 3*2=6 | <- | 2*1=2 | <- | 1 |

14.15. Serialização de Objetos

De forma genérica a serialização é uma técnica usada para persistir objetos, ou seja, gravar objetos em disco, para fazer a transmissão remota de objetos via rede, armazenar os objetos em um banco de dados e/ou arquivos (binários, xml, etc.).

Serializar nada mais é do que colocar os valores que o objeto está utilizando juntamente com suas propriedades de uma forma que fique em série (sequencial). Fazendo isto estamos tornando o objeto Serializable, e, tornando um objeto Serializable, estamos atribuindo essa qualidade a ele, e dando privilégios para que o mesmo possa ser gravado em disco ou enviado por rede.

A serialização é o processo de armazenar um objeto, incluindo todos os atributos públicos e privados para um stream. Se você faz a serialização naturalmente vai querer fazer o processo inverso - deserialização, que seria restaurar os atributos de um objeto gravado em um stream.

O stream é um objeto de transmissão de dados, onde um fluxo de dados serial é feito através de uma origem e de um destino. Pode-se utilizar dois tipos de stream, o FileOutputStream e o FileInputStream para manipular objetos serializados.

O FileOutputStream é um fluxo de arquivo que permite a gravação em disco. Já o FileInputStream é justamente o contrário, permitindo a leitura de um arquivo em disco.

No exemplo a seguir, é feita a gravação / leitura de objetos serializados em um arquivo. Também serão utilizadas duas classes chamadas ObjectInputStream e ObjectOutputStream. Elas são responsáveis por inserir e recuperar objetos serializados do stream.

Primeiro é criada uma classe chamada Genero, que será utilizada como Objeto serial de armazenamento:

```
class Genero implements java.io.Serializable
{
    int genCodigo;
    String genDescricao;
    // definição dos métodos get e set ...
}
```

O exemplo a seguir ilustra como persistir um objeto Gênero em um arquivo:

```
import java.io.*;
public class Teste_Serializacao {
    public static void main(String[] args) {
        Genero g = new Genero();
        g.genCodigo = 1;
        g.genDescricao = "Teste";
        File arq = new File("c:/arquivo.ser");
        try {
            // cria um outputstream para escrever no arquivo
            FileOutputStream fos = new FileOutputStream( arq );
            // cria um mapeamento do objeto para o fos
            ObjectOutputStream oos = new ObjectOutputStream( fos );
            // mapeia o objeto para o arquivo
            oos.writeObject( g );
            /* OBS: a classe do obj deve implementar a classe java.io.Serializable
            */
            oos.close();
            fos.close();
        }catch (Exception e) {
            System.out.println("Erro ao persistir objeto: "+e.getMessage());
        }
    }
}
```

O exemplo a seguir ilustra como recuperar um objeto Gênero em um arquivo:

```
import java.io.*;
public class Teste_Serializacao {
    public static void main(String[] args) {
        File arq = new File("c:/arquivo.ser");
        if (!arq.exists())
            System.out.println("Arquivo não encontrado!");
        try {
            FileInputStream fis = new FileInputStream( arq );
            ObjectInputStream ois = new ObjectInputStream( fis );
            Genero g = (Genero) ois.readObject();
            System.out.println("Código: "+g.genCodigo+
                               " Descrição: "+g.genDescricao);
            ois.close();
            fis.close();
        }catch(Exception e)
        {
            System.out.println("Erro ao recuperar objeto: "+e.getMessage());
        }
    }
}
```

Para excluir o arquivo de persistência:

```
import java.io.*;
public class Teste_Serializacao {
    public static void main(String[] args) {
        File arq = new File("c:/arquivo.ser");
        if (!arq.exists())
            System.out.println("Arquivo não encontrado!");
        try {
            arq.delete();
        }catch(Exception e)
        {
            System.out.println("Erro excluir arquivo: "+e.getMessage());
        }
    }
}
```

14.16. Cast de Objetos

Os mesmos conceitos que são utilizados para o cast de variáveis também podem ser aplicados as variáveis de referência a objetos. Considere a seguinte situação:

```
class Animal{
    public void comer(){
        System.out.println( "comendo...." );
    }
}

class Dog extends Animal{
    public void latir(){
        System.out.println( "au, au, au ...." );
    }
}
```

Neste exemplo possuímos uma classe chamada Animal que é capaz de se alimentar e uma segunda chamada Dog que extende da classe Animal podendo assim além de se alimentar também latir. Observe o trecho de código abaixo:

```
public static void main(String arg[]){
    Dog rex = new Dog();
    Animal anim01 = rex;
}
```

A variável de referencia anim01 segundo as regras e vantagens da orientação a objetos, pode armazenar tanto objetos do tipo Animal quanto objetos do tipo Dog, o que é abordado em polimorfismo, porem quando armazenamos nosso objeto Dog em uma classe superior como neste caso, todas as características próprias da classe Dog serão perdidas, ou melhor, ocultadas, pois um objeto Animal não sabe como é que um objeto Dog deve se comportar, no caso específico do nosso exemplo, mesmo o objeto sendo criado como um Dog, quando for atribuído a uma variável de referencia Animal não poderá mais “latir” por que isso é uma característica que apenas os cães possuem, então como podemos recuperar as características do objeto Dog já que não podemos atribuir um objeto do tipo Animal (superclasse) a um objetos Dog (subclasse de Animal)? A resposta é simples, cast.

```
public static void main(String arg[]){
    Dog rex = new Dog();
    Animal anim01 = rex;

    Dog rex2 = (Dog) anim01; // recupera como Dog
    rex.latir();
}
```

Da mesma maneira que faríamos um cast entre variáveis primitivas podemos modelar também referencias de objetos como mostrado no exemplo acima. Mas lembre-se o cast somente será compilado quando as classes utilizadas possuem algum tipo de vínculos como a herança por exemplo, caso contrario, se você quiser transformar objetos não relacionados como por exemplo, transformar um Tree em um Dog, causará um erro de compilação.

15. DISTRIBUINDO A APLICAÇÃO JAVA

Um programa Java é um conjunto de classes. Esse será o resultado do seu desenvolvimento, mas como distribuiremos nossa aplicação em outros sistemas que não são idênticos ao sistema em que o projeto foi construído?

A solução para este problema são os arquivos JAR.

Um arquivo JAR é um Java ARchive. Ele se baseia no formato de arquivo pkzip e permitirá que você empacote todas as suas classes para, que em vez de fornecer a seu cliente dezenas de arquivos de classes, passe apenas um único arquivo JAR, que pode inclusive se comportar como um arquivo executável, independente da plataforma em que é executado.

Isso significa que o usuário não precisará extrair os arquivos de classes para executar o programa, o usuário final poderá executar seu aplicativo com os arquivos de classes ainda no arquivo JAR, para isso devemos criar um arquivo de declaração que seja inserido dentro do JAR contendo informações sobre as classes que o compõe, entre elas sem dúvida a mais importante para que seu JAR fique executável é a declaração de qual das classes deste JAR possui o método main().

Poucos passos são necessários para a criação de seu arquivo JAR executável.

Primeiro, certifique-se que todos os seus arquivos de classes fiquem no mesmo diretório, o ideal é seguir o padrão de estrutura de diretórios para projetos, justamente por isso as ferramentas gráficas em Java seguem um padrão ao gerarem novos projetos.

Apos a organização de todas as classes utilizadas pela sua aplicação em um único diretório, o próximo passo é a criação do arquivo que indica para o JAR qual das classes que ele possui é a primeira que deve ser executada, ou em outras palavras qual é a classe que possui o método main(). Para isso basta criarmos um arquivo de texto com o nome de manifest.txt que contenha pelo menos a linha:

```
Main-Class: ClasseComMetodoMain
```

Onde, ClasseComMetodoMain é o arquivo .class que possui entre os seus métodos o método main() responsável pelo inicio do programa, dois detalhes fazem parte desta linha: a extensão do arquivo (.class) não deve ser escrita, e também é aconselhável pressionar enter ao final da linha para que seu arquivo de declaração não corra o risco de funcionar incorretamente.

Por ultimo é preciso executar a ferramenta jar, a qual faz parte do JDK, para criar o arquivo JAR que contenha tudo o que existe no diretório de classes mais o arquivo de declaração (manifest.txt) da seguinte forma:

```
> cd /diretório/MinhasClasses  
> jar -cvmf manifest.txt MeuAplicativo.jar *.class
```

Na primeira linha entramos dentro do diretório onde encontram-se as classes que desejamos inserir no arquivo jar, as quais já organizamos no primeiro passo. Em seguida chamamos a ferramenta jar indicando a ela qual o arquivo de declaração (manifest.txt), o nome que queremos dar ao nosso .jar (MeuAplicativo.jar) e por ultimo quais as classes que farão parte do Java Archive, que em nosso exemplo (*.class) selecionamos todos os arquivos .class do diretório atual.

Assim, será criado o arquivo executável, permitindo distribuir os aplicativos escritos em Java. Dependendo do seu sistema operacional basta dois cliques para a execução do programa, ou então pela linha de comando com uma pequena modificação em relação a execução de uma classe simples, a adição do flag –jar apos a chamada da JVM.

```
>java -jar MeuAplicativo.jar
```

As ferramentas gráficas para desenvolvimento em Java possuem métodos simples para gerar os arquivos .jar. As imagens abaixo mostram formas de geração do arquivo JAR no NetBeans e no Eclipse, em que pode-se gerar o JAR apenas selecionando o projeto desejado.



Figura 27 - Distribuição no NetBeans

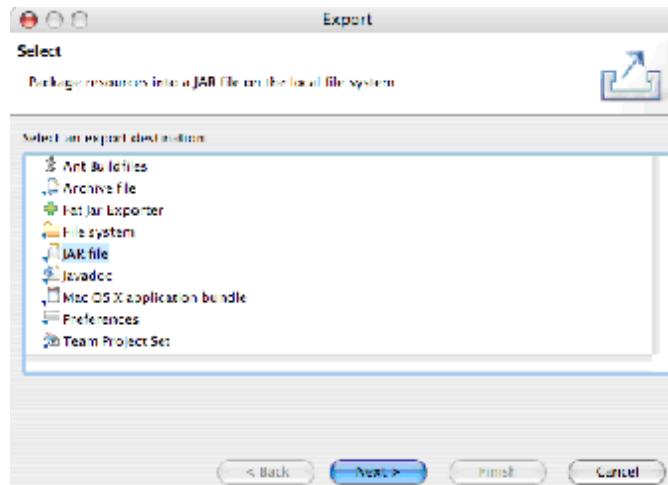


Figura 28 - Distribuição no Eclipse

16. COMO USAR A DOCUMENTAÇÃO DO JAVA

Para poder ter acesso a docmunetação da API do Java, tem que acessar a página através do link : <http://java.sun.com/j2se/1.4.2/docs/api/>. Onde pode ser encontrada uma página como mostra a figura a seguir:

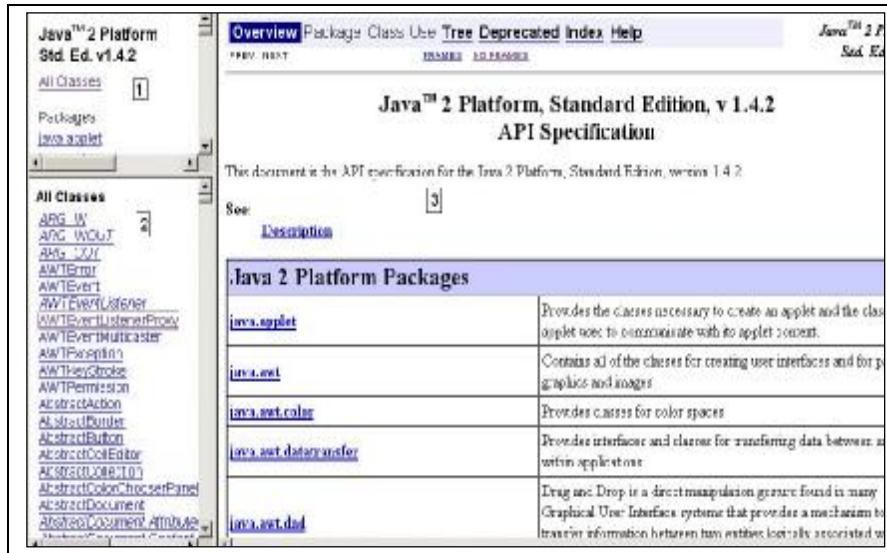


Figura 29 - Documentação do Java

Como mostra a figura esta é a tela inicial da documentação, onde é possível encontrar todos os pacotes e classes do Java e as suas descrições

A figura a seguir ilustra que ao selecionar uma classe do lado esquerdo da página, será mostrado no lado direito a descrição da mesma, como também de qual classe esta é herdada, as suas implementações, atributos e métodos que possui.



Figura 30 - Documentação de uma Classe

17. REFERÊNCIAS BIBLIOGRÁFICAS

CADENHEAD, Rogers. Aprenda em 21 dias Java 2: professional reference. Rio de Janeiro: Elsevier, 2003. 576 p.

CADENHEAD, Rogers. Aprenda em 21 dias Java 2. Rio de Janeiro: Elsevier, 2005. 525 p.

DEITEL, H. M.. Java: como programar. Porto Alegre: Bookman, 2003. 1386 p.

FURGERI, Sérgio. Java 2: ensino didático: desenvolvendo e implementando aplicações. São Paulo: Érica, 2003. 372 p.

FURGERI, Sérgio. Java 2: ensino didático. São Paulo: Érica, 2005. 372 p.

HUBBARD, John Programação com Java.2.ed. São Paulo: Ed ARTMED Editora S.A, 2006.

SANTOS, Rafael. Introdução à programação orientada a objetos usando Java. Rio de Janeiro: Campus,2003. 317 p.