1 INTRODUÇÃO

LINGUAGENS: PROCEDURAIS X ORIENTADAS A OBJETOS

- Linguagens como C são procedurais, isto é, a programação tende a ser orientada para ação.
- No C, a unidade de programação é a função. Grupos de ação que realizam alguma tarefa comum são reunidos em funções e estas são agrupadas para formar os programas.
- No Java, a unidade de programação é a classe, a partir da qual os objetos são instanciados (criados).
- Classes Java contêm campos (que implementam atributos) e métodos (que implementam operações e são semelhantes a funções na linguagem C).

UML - Unified Modeling Language

- Na década de 1980, um número crescente de empresas começou a utilizar a POO para construir seus aplicativos e percebeu-se a necessidade de um processo-padrão de análise e projeto orientado a objetos (OOAD - Object-Oriented Analisys and Design).
- Em 1996, o grupo formado por James Rumbaugh, Grady Booch (Rational Software Corporation) e Ivar Jacobson liberou as primeiras versões da UML. Agora, em sua versão 2, a UML é o esquema de representação mais amplamente utilizado para modelar sistemas orientados a objetos.

OOD - Oriented Object Design

- O OOD (Projeto Orientado a Objetos) fornece uma maneira natural e intuitiva de visualizar o processo de projeto de software: modelar objetos por seus atributos e comportamentos da maneira como descrevemos objetos do mundo real. Alguns conceitos importantes:
  - Classes
  - Objetos
  - Atributos
  - Métodos
  - Encapsulamento
  - Herança
  - Polimorfismo

Classes

- Uma classe é um gabarito ou modelo para a definição de objetos.
- As classes estão para os objetos assim como as plantas arquitetônicas estão para as casas. Podemos construir várias casas a partir de uma planta, logo podemos instanciar (criar) muitos objetos a partir de uma classe. Você não pode fazer refeições na cozinha de uma planta; isto só é possível em uma cozinha real.
- Através da definição de uma classe descrevemos as propriedades (atributos) de seus objetos.
- A definição de uma classe descreve também as funcionalidades (métodos) aplicadas aos seus objetos.

- Na UML (Unified Modeling Language) a especificação de uma classe é composta por três regiões: o nome da classe, o conjunto de atributos e o conjunto de métodos:

NomeClasse
visibilidade nomeAtributo : tipo = valorDefault visibilidade nomeAtributo : tipo = valorDefault
visibilidade nomeMetodo(listaArgumentos) : tipoRetorno visibilidade nomeMetodo(listaArgumentos) : tipoRetorno

- O modificador de visibilidade (presente tanto para atributos como para métodos) apresenta três categorias de visibilidade:
  - **public** (representado em UML pelo símbolo +): os atributos ou métodos do objeto podem ser acessados por qualquer outro objeto (visibilidade externa total).
  - **private** (representado em UML pelo símbolo -): os atributos ou métodos do objeto não podem ser acessados por nenhum outro objeto que não pertença à própria classe (nenhuma visibilidade externa).
  - **protected** (representado em UML pelo símbolo #): os atributos ou métodos do objeto poderão ser acessados apenas por objetos de classes que sejam derivadas da classe através do mecanismo de herança.

Objetos

- Um objeto ou instância é uma materialização da classe, e assim pode ser utilizado para representar dados e executar operações.
- Para que os objetos ou instâncias possam ser manipulados, é necessária a criação de referências a estes objetos, que são basicamente variáveis do tipo da classe.
- Para criar novos objetos utilizamos o operador new:

```
Teste t = new Teste();
```

(atribuímos à variável **t** a referência ao objeto)

Atributos

- O conjunto de atributos descreve as propriedades da classe.
- Cada atributo é identificado por um nome e tem um tipo associado.
- O atributo pode ainda ter um valor default.

```
class Teste {
   int x = 0;
   char y = 'a';
   ...
}
```

Métodos

- Os métodos definem as funcionalidades da classe, ou seja, o que será possível fazer com os seus objetos.
- Cada método é especificado por uma assinatura composta pelo nome do método, o tipo para valor de retorno e uma lista de argumentos (cada argumento é identificado por seu tipo e nome).

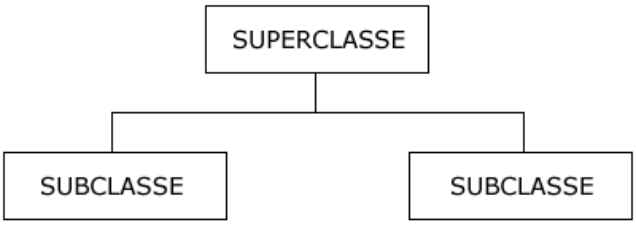
```
class Teste {
   static int soma(int x, int y) {
      return (x+y);
   }
   public static void main (String args[]) {
      Teste t = new Teste();
      System.out.println(t.soma(2,3));
   }
}
```

Encapsulamento

- É o princípio de projeto pelo qual cada componente de um programa deve agregar toda informação relevante para sua manipulação como uma unidade (uma cápsula).
- A estrutura de um objeto e a implementação de seus métodos devem ser tão privativos quanto possível.
- Cada objeto deve ser manipulado exclusivamente através de métodos públicos, dos quais apenas a assinatura deve ser revelada. O conjunto de assinaturas dos métodos públicos da classe constitui sua interface operacional.
- Esta técnica permite que o usuário do objeto trabalhe em um nível mais alto de abstração, sem preocupação com os detalhes internos da classe e simplifica a construção de programas com funcionalidades complexas, tais como interfaces gráficas ou aplicações distribuídas.

Herança

- É o mecanismo que permite que características comuns a diversas classes sejam fatoradas em uma classe base ou superclasse. A partir de uma classe base, outras classes podem ser especificadas.
- Cada classe derivada ou subclasse apresenta as características (estrutura e métodos) da classe base e acrescenta a elas outras particularidades.



- Há várias formas de relacionamentos em herança:
  - **Extensão:** a subclasse estende a superclasse, acrescentando novos membros (atributos e/ou métodos). A Superclasse permanece inalterada, motivo pelo qual este relacionamento é normalmente referenciado como **herança estrita**.
  - **Especificação:** a superclasse especifica o que uma subclasse deve oferecer, mas não implementa nenhuma funcionalidade. Apenas a **interface** (conjunto de especificações dos métodos públicos) da superclasse é herdada pela subclasse.
  - **Extensão e Especificação:** a subclasse herda a interface e uma especificação padrão de (pelo menos alguns) métodos da superclasse. A subclasse pode então redefinir métodos para especificar o comportamento em relação ao que é oferecido pela superclasse ou implementar métodos que tenham sido apenas declarados na superclasse. Este tipo de herança é denominado **herança polimórfica**.

Polimorfismo

- É o princípio pelo qual duas ou mais classes derivadas de uma mesma superclasse podem invocar métodos que têm a mesma assinatura mas comportamentos distintos, especializados para cada classse derivada, usando para tanto uma referência a um objeto do tipo da superclasse.
- Para que o polimorfismo possa ser utilizado é necessário que os métodos que estejam sendo definidos nas classe derivadas tenham exatamente a mesma assinatura do método definido na superclasse. Neste caso, está sendo utilizado o mecanismo de redefinição de métodos (overriding). Este mecanismo de redifinição é diferente do mecanismo de sobrecarga (overloading) de métodos, onde as listas de argumentos são diferentes.
- A decisão sobre qual dos métodos deverá ser selecionado, de acordo com o tipo de objeto, poderá ser tomada apenas em tempo de execução através do mecanismo de ligação tardia (conhecido em inglês pelos termos: late binding, dynamic binding ou run-time binding).

2 JAVA: CONCEITOS BÁSICOS

JAVA: 'WRITE ONCE, RUN ANYWHERE'

Slogan utilizado pela Sun: você escreve uma vez sua aplicação e poderá executá-la em 'qualquer lugar' (qualquer sistema operacional com uma JVM – Java Virtual Machine).

HISTÓRIA

- 1991: James Gosling e sua equipe da Sun Microsystems receberam a tarefa de criar aplicações para eletrodomésticos.
- Começaram a desenvolver utilizando C++ (uma linguagem atual naquela época).
- Perceberam que C++ não era a linguagem mais adequada para aquele projeto.
- Conclusão: era preciso criar uma nova linguagem.
- Gosling utilizou o C++ como modelo, aproveitando a sintaxe básica da linguagem e sua natureza orientada a objetos, retirando características que a tornavam mais complexa.
- Esta nova linguagem foi denominada Oak (pinheiro, em inglês) e foi utilizada pela primeira vez em um projeto chamado Green (objetivo: projetar um sistema de controle remoto que permitisse ao usuário controlar vários dispositivos (TV, vídeo-cassete, etc.) a partir de um computador portátil chamado Star Seven).
- O nome Oak já estava registrado por outra companhia. A Sun decidiu trocar o nome da linguagem para JAVA (nome de uma ilha na Indonésia e homenagem ao gosto da equipe por café).
- 1993: World Wide Web. A equipe percebeu que a linguagem era perfeita para programação Web. Surgiram os applets (pequenos programas que podiam ser incluídos em páginas Web).
- 1995: A Sun anunciou oficialmente o Java. Netscape e Microsoft adicionam suporte ao Java para os seus respectivos browsers.

VANTAGENS

- Atende às necessidades do desenvolvimento de aplicações em um ambiente distribuído (rede) e heterogêneo.
- Simples: os conceitos fundamentais de Java podem ser absorvidos rapidamente (principalmente pelos programadores que conhecem C++).
- Orientada a objetos desde sua criação.
- Possui um grande número de bibliotecas de classes que proporcionam várias funcionalidades (E/S, networking, interface gráfica, multimídia, etc.).
- Robusta e segura: provê verificações durante a compilação e em tempo de execução. Aplicações Java são seguras contra códigos não autorizados que tentam criar vírus ou invadir sistemas de arquivos.
- Gerência de memória simples: objetos são criados por meio de um operador new e a liberação de memória é feita por meio de um mecanismo chamado garbage collection (coleta de lixo).
- Arquitetura neutra e portável: As aplicações podem ser executadas em uma variedade de sistemas operacionais em diferentes arquiteturas de hardware. (Para acomodar esta diversidade de plataformas, o compilador Java gera bytecodes, um formato intermediário neutro projetado para transporte eficiente em múltiplos ambientes de software/hardware.)
- Alto desempenho: O interpretador da linguagem possui algumas otimizações que permitem a execução mais eficiente do código Java. Obviamente, pela característica interpretada da linguagem seu desempenho, em geral, é inferior ao de linguagens compiladas, porém existem soluções como compiladores Just-in-Time (JIT) que podem melhorar consideravelmente o desempenho de aplicações.
- Multithreaded: Para cada atividade é reservada uma thread exclusiva de execução. Todas as bibliotecas de sistema da linguagem Java foram desenvolvidas para serem "thread safe", ou seja, não existe possibilidade de ocorrência de conflitos caso threads concorrentes executem funções destas bibliotecas.
- Dinâmica: As classes (código) somente são ligadas (linked) a aplicação quando necessário. Novos módulos podem ser ligados sob demanda a partir de diversas fontes (inclusive através da própria rede). Esta característica proporciona a possibilidade de se atualizar transparentemente as aplicações.

ARQUITETURA JAVA

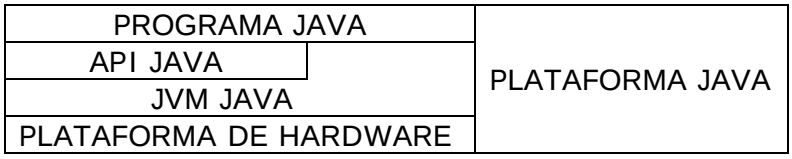
- Um programa Java é compilado e interpretado.
- O compilador faz a tradução de um programa Java para um código intermediário chamado bytecode.
- Os bytecodes são independentes de arquitetura de arquitetura de software/hardware.
- O interpretador analisa (parse) cada instrução do bytecode e a executa no computador.
- A compilação ocorre apenas uma vez, já a interpretação acontece cada vez que o programa é executado.

A plataforma Java possui dois componentes:

- A JVM – Java Virtual Machine (Máquina Virtual Java)
- A API – Application Programming Interface (Interface de Programação de Aplicação)

- Os programas Java podem ser compilados em qualquer plataforma que possuir um compilador Java.
- O produto da compilação (bytecodes) pode ser executado em qualquer implementação da Java Virtual Machine.
- Podemos imaginar os bytecodes como uma "linguagem assembly" da Java Virtual Machine.

- A API do Java é uma grande coleção de componentes de software, prontos para uso, que oferecem muitas funcionalidades úteis como elementos de interface gráfica (GUI).
- A API é agrupada em bibliotecas (packages) de componentes relacionados.



3 AMBIENTE DE DESENVOLVIMENTO

- SDK – Software Development Kit: formado pelas classes fundamentais da linguagem Java e por um conjunto de aplicativos que permite realizar a compilação e a execução de programas.
- Não é um ambiente integrado de desenvolvimento, não oferece editores ou ambiente de programação visual.
- Contém um amplo conjunto de APIs que compõem o núcleo de funcionalidades da linguagem Java.

FERRAMENTAS DO SDK

- javac: compilador Java
- java: interpretador
- appletviewer: interpretador de applets *
- javadoc: gerador de documentação
- jar: manipulador de arquivos comprimidos (formato Java Archive)
- jdb: depurador de programas

** Applets: programas que possuem características que permitem que sejam executados em browsers (com suporte a Java). Servlets: programas que são executados em servidores Java (TomCat, etc.).*

O JDK é gratuito para download e pode ser obtido em:

```
http://java.sun.com/javase/downloads/index_jdk5.jsp
```

NOTA: Para obter instruções sobre a instalação do JDK 1.5 veja o Apêndice A.

Aplicação "Hello World"

Os seguintes passos são necessários para a criar uma aplicação Java "standalone":

1. Criar um arquivo fonte em Java

Usando um editor de texto qualquer (exemplo: bloco de notas), crie um arquivo (ASCII) chamado **HelloWorldApp.java** com o seguinte código Java:

```
/**
* A classe HelloWorldApp implementa uma aplicação que
* simplesmente mostra "Hello World!" na saída padrão (monitor).
*/
class HelloWorldApp {
    public static void main(String[] args) {
        System.out.println("Hello World!"); // imprime "Hello World!"
    }
}
```

2. Compilar o código fonte

No prompt de comando, execute o compilador Java:

```
javac HelloWorldApp.java
```

Se a compilação tiver êxito, o compilador vai criar um arquivo chamado **HelloWorldApp.class** no mesmo diretório do arquivo fonte (**HelloWorldApp.java**). O arquivo gerado contém os bytecodes Java, independentes de plataforma, que são interpretados pelo ambiente de execução do Java.

Se a compilação falhar, verifique se o programa foi digitado corretamente (cuidado com as letras maiúsculas/minúsculas).

3. Executar a Aplicação

No prompt de comando, execute o interpretador Java:

```
java HelloWorldApp
```

O argumento do interpretador Java é o nome da classe a ser executada e **não** o nome do arquivo. O nome da classe deve ser digitado da mesma maneira que foi definido no código fonte (maiúsculas/minúsculas).

Anatomia da Aplicação

Uma vez executada a primeira aplicação Java, vamos analisar esta aplicação standalone.

Comentários em Java

A aplicação "Hello World" tem dois blocos de comentários. O primeiro bloco, no início do programa, usa os delimitadores /** e */. Depois, uma linha de código é explicada por meio de um comentário marcado com os caracteres //. A linguagem Java suporta um terceiro tipo de comentário, estilo C, delimitado por /* e */.

Definindo uma Classe

Na linguagem Java, cada método (função) e variável deve existir dentro de uma classe ou objeto (uma instância de uma classe). A linguagem Java não suporta funções ou variáveis globais. Portanto, o esqueleto de qualquer programa Java é uma definição de classe.

Em Java, a forma mais simples de se definir uma classe é:

```
class NomeDaClasse {
. . .
}
```

A palavra-chave *class* começa a definição de uma classe denominada **NomeDaClasse**. As variáveis e os métodos da classe são delimitados por um par de chaves. A aplicação "Hello World" não possui variáveis e tem um único método chamado **main**.

O Método main

O ponto de entrada de qualquer aplicação Java é o método **main**. Quando se executa uma aplicação, via interpretador Java, na verdade, especifica-se o nome da classe a ser executada. O interpretador, então, invoca o método **main** definido dentro daquela classe. O método **main** controla o fluxo do programa, aloca recursos e executa quaisquer outros métodos que fazem parte da funcionalidade da aplicação.

Utilizando Classes e Objetos

Os outros componentes de uma aplicação Java são os objetos, classes, métodos e comandos (statements) da linguagem Java escritos para implementar as funcionalidades desejadas.

A aplicação "Hello World", por exemplo, utiliza uma outra classe, **System**, que é parte da API que acompanha o ambiente Java. A classe System oferece acesso a funções do sistema operacional.

Applet "Hello World"

Os seguintes passos são necessários para a criar um applet Java, que pode ser executado em um browser.

1. Criar um Arquivo Fonte em Java

Usando um editor de texto qualquer crie um arquivo (ASCII) chamado **HelloWorld.java** com o seguinte código Java:

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

public class HelloWorld extends Applet {
    public void paint(Graphics g) {
        g.drawString("Hello world!", 50, 25);
    }
}
```

2. Compilar o Código Fonte

No prompt de comando, execute o compilador Java:

```
javac HelloWorld.java
```

Se a compilação tiver êxito, o compilador vai criar um arquivo chamado HelloWorld.class no mesmo diretório do arquivo fonte (**HelloWorld.java**). O arquivo gerado contém os bytecodes Java, independentes de plataforma, que são interpretados pelo ambiente de execução.

Se a compilação falhar, verifique se o programa foi digitado corretamente (cuidado com as letras maiúsculas/minúsculas).

3. Criar um Arquivo HTML que Inclua o Applet

Utilizando um editor de texto qualquer, crie um arquivo chamado **hello.html** no mesmo diretório que contém a classe **HelloWorld.class**. Este arquivo HTML deve conter o seguinte código:

```
<HTML>
<HEAD>
  <TITLE> Um Programa Simples </TITLE>
</HEAD>
<BODY>
  Aqui está o meu applet:
  <APPLET CODE="HelloWorld.class" WIDTH=150 HEIGHT=25>
  </APPLET>
</BODY>
</HTML>
```

4. Executar o Applet

Para executar o applet, deve-se carregar o arquivo HTML em um ambiente que possa executar applets Java. Este ambiente pode ser um browser compatível com Java ou um programa para visualização de applets, como o Applet Viewer que acompanha o JDK.

No browser, para carregar o applet, pode-se digitar, no campo Endereço:

```
C:\hello.html
```

Para utilizar o Applet Viewer, digite no prompt do DOS:

```
appletviewer c:\hello.html
```

4 TIPOS PRIMITIVOS, OPERADORES E STRINGS

TIPOS PRIMITIVOS

Java oferece tipos literais primitivos (não objetos) para representar:

- valores booleanos
- caracteres
- numéricos inteiros
- numéricos em ponto flutuante

VALORES BOOLEANOS		
TIPO	TAMANHO	FAIXA DE VALORES
boolean	1 bit	true ou false

```
boolean a = true;
boolean b = false;
```

CARACTERES		
TIPO	TAMANHO	FAIXA DE VALORES
char	16 bits	**um** caractere UNICODE

```
char a = 'x'; // Apenas um caractere e utilizar aspas simples. Não aceita '' (vazio).
```

UNICODE é um padrão internacional para representação unificada de caracteres de diversas linguagens.

NUMÉRICOS INTEIROS		
TIPO	TAMANHO	FAIXA DE VALORES
byte	8 bits	-128 a +127
short	16 bits	-32.768 a +32767
int	32 bits	-2.147.483.648 a +2.147.483.647
long	64 bits	-9.223.372.036.854.775.808 a +9.223.372.036.854.775.807

NUMÉRICOS EM PONTO FLUTUANTE		
TIPO	TAMANHO	FAIXA DE VALORES
float	32 bits	±1,40239846432481707e-45 a ±3,40282346638528860e+38
double	64 bits	±4,94065645841246544e-324 a ±1,79769313486231570e+308

Os números do tipo **float** têm um sufixo f, por exemplo, 4.325f. Os números de ponto flutuante sem um sufixo f (como 3.325) serão sempre considerados do tipo **double**.

NOTA: O tipo **void** denota a ausência de valor. Este tipo é usado em situações especiais envolvendo métodos. Nenhuma variável pode ser do tipo void.

Lembre-se! O valor de um tipo primitivo é sempre copiado:

```
int x = 2; // x recebe uma cópia do valor 2
y = x; // y recebe uma cópia do valor x
x = x + 1; // x vale 3 e y continua 2
```

IDENTIFICADORES

Seqüências de caracteres UNICODE que obedecem às seguintes regras:

- Um nome pode ser composto por letras, por dígitos e pelos símbolos _ e $
- Um nome não pode ser iniciado por um dígito (0 a 9)
- Letras maiúsculas são diferenciadas de letra minúsculas
- Uma palavra reservada da linguagem Java não pode ser um identificador

OPERADORES

UNÁRIOS	
++	incremento
--	decremento

ATRIBUIÇÃO	
a=b	atribui o valor de *b* à variável *a*

EXERCÍCIO:

Digite, compile e execute os dois códigos a seguir e responda: quais serão os valores de **a** e **b** em cada caso? Explique.

```
public class Teste3a {
   public static void main(String args[]) {
      int a=1, b=1;
      a=++b; // pré incremento
      System.out.println("a = " + a);
      System.out.println("b = " + b);
   }
}
```

```
public class Teste3b {
   public static void main(String args[]) {
      int a=1, b=1;
      a=b++; pós incremento
      System.out.println("a = " + a);
      System.out.println("b = " + b);
   }
}
```

ARITMÉTICOS	
+	soma
-	subtração
*	multiplicação
/	divisão
%	resto de divisão (para inteiros)

```
public class Teste3c {
   public static void main(String args[]) {
      int x=5, y=2, a, b, c, d, e;
      a=x+y;
      b=x-y;
      c=x*y;
      d=x/y;
      e=x%y;
      System.out.println("x=" + x + " y=" + y);
      System.out.println("a=x+y=" + a);
      System.out.println("b=x-y=" + b);
      System.out.println("c=x*y=" + c);
      System.out.println("d=x/y=" + d);
      System.out.println("e=x%y=" + e);
   }
}
```

LÓGICOS	
|	OR (OU) retorna *true* se pelo menos um dos dois argumentos é *true*
||	OR (OU) se o primeiro argumento é *true* o segundo não é nem avaliado
&	AND (E) retorna *false* se pelo menos um dos dois argumentos é *false*
&&	AND (e) se o primeiro argumento é *false* o segundo não é nem avaliado
^	XOR (OU EXCLUSIVO) retorna *true* quando os dois argumentos têm valores lógicos distintos
!	NO (NEGAÇÃO) retorna *false* se o valor da expressão é *true* e vice-versa

RELACIONAIS	
<	menor que
>	maior que
==	igual
<=	menor ou igual
>=	maior ou igual
!=	diferente de

Exemplo:

```
public class Teste3d {
   public static void main(String args[]) {
      int x=4, y=2;
      System.out.println("x=" + x + " y=" + y);
      if ((x%2==0) && (y%2==0)) {
         System.out.println("Os dois numeros sao PARES"); }
      else if ((x%2==0) || (y%2==0)) {
         System.out.println("Apenas um dos numeros e PAR"); }
      else {
         System.out.println("Nenhum dos numeros e PAR"); }
   }
}
```

EXERCÍCIO:

Fizemos abaixo uma "pequena" alteração no código anterior. Ele continuará apresentando a resposta correta? Por que?

```
public class Teste3e {
   public static void main(String args[]) {
      int x=4, y=2;
      System.out.println("x=" + x + " y=" + y);
      if ((x%2==0) || (y%2==0)) {
         System.out.println("Apenas um dos numeros e PAR"); }
      else if ((x%2==0) && (y%2==0)) {
         System.out.println("Os dois numeros sao pares"); }
      else {
         System.out.println("Nenhum dos numeros e PAR"); }
   }
}
```

ATRIBUIÇÃO REDUZIDOS	
a+=b	a=a+b
a-=b	a=a-b
a*=b	a=a*b
a/=b	a=a/b
a%=b	a=a%b

CONDICIONAL	
exp1?a:b	retorna *a* se o valor da *exp1* for *true* e *b* se o valor da *exp1* for *false*

EXERCÍCIO

Qual será o resultado do código abaixo?

```
public class Teste3g {
   public static void main(String args[]) {
      int a=1, b=2;
         System.out.println(a>b?a+1:b+1);
   }
}
```

CASTING

```
int x = 3.14; // errado

double x = 3.14;
int y = x; // errado

double x = 3;
int y = x; // errado

long x = 10000;
int y = x; // errado

int x = 3;
double y = x; // correto
```

Às vezes precisamos que um número (float ou double) seja arredondado e armazenado num inteiro. Para fazer isso sem que haja erro de compilação, é preciso ordenar que o número seja moldado (casted) como um inteiro. Esse processo recebe o nome de **casting**.

```
double x = 3.14;
int y = (int) x; // compila


long x = 10000;
int y = (int) x; // compila


float x = 0.0; // errado: todos literais de ponto flutuante são tratados como double

float x = 0.0f; // errado: o f indica que o literal deve ser tratado como float


double x = 3;
float y = 2;
float z = x + y; // errado: o java armazena sempre no maior tipo (double)

double x = 3;
float y = 2;
float z = (float) x + y; // correto
```

STRINGS

A classe **String** é usada para representar cadeias de caracteres (strings). Não são tipos primitivos (nativos), mas instâncias da classe String.

Teste de igualdade de Strings

Para testar se duas strings são iguais utiliza-se o método **equals**:

```
public class Teste3f {
   public static void main(String args[]) {
      String a="Aluno";
      String b="Aluno";
      if (a.equals(b)) // ou if (a.equals("Aluno"))
         System.out.println("Sao iguais");
      else
         System.out.println("Sao diferentes");
   }
}
```

O método **equalsIgnoreCase** testa se duas strings são iguais, sem levar em conta a diferenciação entre maiúsculas e minúsculas.

```
      if (a.equalsIgnoreCase(b)) // ou if (a.equalsIgnoreCase("ALUNO"))
```

Comprimento de Strings

Para verificar o comprimento de uma String utilizamos o método **length**:

```
      String a="Aluno";
      int n=a.length(); // n vale 3
```

Obtendo caracteres individuais em uma String

Para obter um determinado caractere em uma String utilizamos o método **charAt**:

```
      String a="Aluno";
      char n=a.charAt(2); // n vale u: a primeira posição da String é 0 (zero)
```

Substrings

Para extrair uma substring (de uma String maior) utilizamos o método **substring**:

```
      String a="Aluno";
      String s=a.substring(0,3); // saída: Alu (posição inicial: 0 e tamanho: 3)
```

PALAVRAS RESERVADAS

abstract	boolean	break	byte	case	catch
char	class	const	continue	default	do
double	else	extends	false	final	finally
float	for	goto	if	implements	import
instanceof	int	interface	long	native	new
null	package	private	protected	public	return
short	static	strictfp	super	switch	syncronized
this	throw	throws	transient	true	try
void	volatile	while			

CARACTERES ESPECIAIS

\n	nova linha
\t	Tabulação
\r	retorno de carro
\b	retrocesso do cursor
\\	caractere \
\'	caractere apóstrofe
\"	caractere aspas

`public static void main (String args[])`

`public:` Indica que o método é público e acessível desde fora da classe que o define. O método **`main`** precisa ser invocado pela JVM, por isso deve ser público.

`static:` indica que o método é estático e que não é necessário que exista uma instância da classe para que possa ser executado. Isto também é necessário visto que, conforme mencionado, o método **`main`** é chamado a partir da JVM.

`void:` Informa ao compilador que o método **`main`** não devolve nenhum valor ao ser executado.

`String args []` ou **`String [] args:`** Define um parâmetro do método **`main`**. Neste caso **`args`** conterá os possíveis parâmetros da linha de comando da execução da classe.

5 ESTRUTURAS DE DECISÃO E DE REPETIÇÃO

ESTRUTURAS DE DECISÃO

if-else

A estrutura **if-else** serve para tomar decisões, permite decidir entre duas possíveis opções.

```
if (codição) {
   codigo;
}
else {
   codigo;
}
```

EXEMPLO:

```
int n = 5;
if (n % 2 == 0) {
   System.out.println("O número é PAR");
}
else {
   System.out.println("O número é ÍMPAR");
}
```

É possível também avaliar mais expressões (booleanas) na mesma estrutura:

```
int n = 5;
if (n < 10) {
   System.out.println("O número é menor que dez");
}
else if (n == 10) {
   System.out.println("O número é igual a dez");
}
else {
   System.out.println("O número é maior que dez");
}
```

switch

```
switch (expressão) {
   case valor1:
      codigo;
      break;
   case valor2:
      codigo;
      break;
   ...
   case valorN:
      codigo;
      break;
   default:
      codigo;
}
```

EXEMPLO:

```
int n = 2;
String extenso;
switch (n) {
   case 1:
      extenso = "um";
      break;
   case 2:
      extenso = "dois";
      break;
```

```
   case 3:
      extenso = "três";
      break;
   default:
      extenso = "maior que três";
}
System.out.println(extenso);
```

ESTRUTURAS DE REPETIÇÃO

while

A estrutura **while** serve repetir um conjunto de instruções enquanto o resultado de uma expressão lógica (uma condição) for verdadeiro.

```
while (codição) {
   codigo;
}
```

do-while

Avalia a expressão lógica no final do laço. Portanto, utilizando **do-while** o laço será executado pelo menos uma vez, mesmo que a condição não seja verdadeira.

```
do {
   codigo
}
while (codição);
```

EXEMPLOS

```
// while: não imprimirá nada
int n = 10;
while (n < 10) {
   System.out.println(n);
   n++;
}
```

```
// do-while: imprimirá 10
do {
   System.out.println(n);
   n++;
}
while (n < 10);
```

for

Utilizado quando queremos executar um conjunto de instruções um determinado número de vezes. Exemplo: imprimir todos os elementos de um array ou todos os registros retornados de uma consulta a um banco de dados.

```
for (inicialização; codição; operador)
{
  codigo;
}
```

EXEMPLO:

```
int n;
for (n=0; n<=100; n++) {
   if (n % 2 == 0) {
      System.out.println(n + " é PAR");
   }
}
```

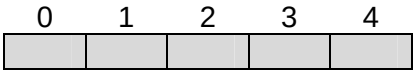
continue

Verifica a expressão condicional, mas não faz a iteração em curso.

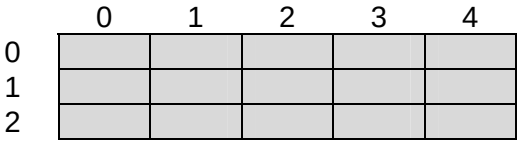
```
      int soma = 0;
         for (int i=0; i<=100; i++) {
            if (i%2==0) continue;
            soma+=i;
         }
      System.out.println("soma dos ímpares: " + soma);
```

6 ARRAYS

Um array (ou matriz) é um conjunto de dados homogêneos que ocupam posições de memória contigua e que pode ser referenciado através de um nome único. Cada um dos elementos que compõem o array pode ser acessado através de um índice que indica sua posição.



Matriz de uma dimensão



Matriz de duas dimensões

ARRAY UNIDIMENSIONAL

Para declararmos um array devemos especificar seu tipo seguido de [ ] e do nome da variável:

```
int[] a;
```

A seguir, temos que reservar a memória para armazenar o array. Fazemos isso utilizando o operador **new**:

```
a = new int[10];
```

Esta declaração cria um array de dez inteiros, referenciado através da variável **a**.

Resumindo:

```
int[] a;
a = new int[10];
```

ou

```
int[] a = new int[10];
```

Quando criamos um array, todas suas posições são inicializadas em 0 (zero). Porém, é possível na hora de criar o array dar-lhe valores diferentes de zero. Observe:

```
int[] a = {1,2,3,4,5};
```

O array acima conterá, ao finalizar sua criação, os inteiros de 1 a 5. Podemos observar também que não foi declarado seu tamanho. Java é capaz de calcular este dado a partir do número de elementos que encontrar na inicialização. No caso acima, criará um array de cinco posições.

O tamanho de um array é indicado no momento de sua definição e não poderá ser modificado posteriormente. Caso seja necessário alterar o tamanho, devemos criar um novo array e passar a este os elementos do array inicial.

É possível conhecer o número de elementos de um array utilizando a propriedade **length**:

```
int[] a = new int[10];
System.out.println(a.length);
// imprime 10
```

EXEMPLO:

```
int[] a = new int[5];
a[0]=1;
a[1]=2;
System.out.println(a[0] + ", " + a[1] + ", " + a[4]);
// imprime 1, 2, 0
```

Copiando de um array **origem** para um array **destino**:

Visto que copiar um array em outro é uma tarefa muito comum, existe um método na classe **System** que podemos usar para copiar os valores de um array em outro:

```
System.arraycopy(origem,indiceOrigem,destino,indiceDestino,nr_elementos);
```

EXEMPLO:

```
int[] a = {1,2,3,4,5};
int[] b = new int[10];
System.arraycopy(a,0,b,0,a.length);
for (int  i=0; i<b.length; i++) {
   System.out.println(b[i]);
}
```

ARRAY MULTIDIMENSIONAL

Quando pensamos em uma tabela (array de duas dimensões) imaginamos um conjunto de arrays de uma dimensão unidos. Isto é literalmente certo em Java. Assim, para definir um array bidimensional, definimos um array de arrays. Observe:

```
// declarar um array de 2 linhas e 3 colunas
int[][] a = new int[2][3];
```

Ao definir um array mutidimensional, é obrigatório somente indicar o número de linhas, depois se pode reservar memória para o resto de forma independente:

```
int[][] a = new int[2][]; // 2 linhas
a[0] = new int[3]; // primeira linha: 3 colunas (elementos)
a[1] = new int[3]; // segunda linha: 3 colunas (elementos)
```

É possível inicializar um array multidimensional no momento da declaração indicando os valores para cada dimensão entre chaves. Cada par de chaves corresponde a uma dimensão:

```
int[][] a = {{1,3,5},{2,4,6}};
```

O acesso a cada um dos elementos do array acima é similar ao acesso em uma dimensão, é necessário somente utilizar mais um grupo de colchetes para a nova dimensão:

```
System.out.println(a[0][1]); // imprime 3: linha 0 coluna 1
System.out.println(a[1][1]); // imprime 4: linha 1 coluna 1
```

7 CLASSES, OBJETOS, ATRIBUTOS E MÉTODOS

CLASSES

- Uma classe é um gabarito ou modelo para a definição de objetos.
- Através da definição de uma classe descrevemos as propriedades (atributos) de seus objetos.
- A definição de uma classe descreve também as funcionalidades (métodos) aplicadas aos seus objetos.
- Na UML (Unified Modeling Language) a especificação de uma classe é composta por três regiões: o nome da classe, o conjunto de atributos e o conjunto de métodos:

NomeClasse
visibilidade nomeAtributo : tipo = valorDefault visibilidade nomeAtributo : tipo = valorDefault
visibilidade nomeMetodo(listaArgumentos) : tipoRetorno visibilidade nomeMetodo(listaArgumentos) : tipoRetorno

O modificador de visibilidade (presente tanto para atributos como para métodos) apresenta três categorias de visibilidade:

- **public** (representado em UML pelo símbolo +): os atributos ou métodos do objeto podem ser acessados por qualquer outro objeto (visibilidade externa total).
- **private** (representado em UML pelo símbolo -): os atributos ou métodos do objeto não podem ser acessados por nenhum outro objeto que não pertença à própria classe (nenhuma visibilidade externa).
- **protected** (representado em UML pelo símbolo #): os atributos ou métodos do objeto poderão ser acessados apenas por objetos de classes que sejam derivadas da classe através do mecanismo de herança.

OBJETOS

- Um objeto ou **instância** é uma materialização da classe, e assim pode ser utilizado para representar dados e executar operações.
- Para que os objetos ou instâncias possam ser manipulados, é necessária a criação de referências a estes objetos, que são basicamente variáveis do tipo da classe.
- Para criar novos objetos utilizamos o operador **`new`**: **`Teste t = new Teste();`**
  (atribuímos à variável **`t`** a referência ao objeto)

ATRIBUTOS

- O conjunto de atributos descreve as propriedades da classe.
- Cada atributo é identificado por um **nome** e tem um **tipo** associado.
- O atributo pode ter ainda um valor default.

MÉTODOS

- Os métodos definem as funcionalidades da classe, ou seja, o que será possível fazer com os seus objetos.
- Cada método é especificado por uma **assinatura** composta pelo nome do método, o tipo para valor de retorno e uma lista de argumentos (cada argumento é identificado por seu tipo e nome).

EXEMPLO:

Liquidificador
- ligado : boolean = false
+ ligar() + desligar() + verificarEstado()

```
public class Liquidificador {

// criar atributo ligado tipo boolean
   private boolean ligado;

// criar método construtor
   public Liquidificador() {
      ligado = false;
   }

// criar método ligar
   public void ligar() {
      ligado = true;
   }

// criar método desligar
   public void desligar() {
      ligado = false;
   }

// criar método verificarEstado
   public boolean verificarEstado() {
      return ligado;
   }

   public static void main(String args[]) {

// criar um novo objeto da classe Liquidificador
// atribuir à variável l a referência do objeto
      Liquidificador l = new Liquidificador();

// imprimir o estado inicial do liquidificador
      System.out.println(l.verificarEstado());

// ligar o liquidificador e imprimir o estado
      l.ligar();
      System.out.println(l.verificarEstado ());

// desligar o liquidificador e imprimir o estado
      l.desligar();
      System.out.println(l.verificarEstado ());
   }
}
```

EXERCÍCIO

Criar mais um atributo para a classe Liquidificador: **velocidade : int = 0** e adicionar três métodos: **aumentarVelocidade(), diminuirVelocidade() e VerificarVelocidade()**. A cada chamada ao método **aumentarVelocidade()** incrementar em 1 o valor de **velocidade** e a cada chamada ao método **diminuirVelocidade()** decrementar em 1 o seu valor.

```
public class Liquidificador {
   private boolean ligado;
   private int velocidade;

   public Liquidificador() {
      ligado = false;
   }
   public void ligar() {
      ligado = true;
   }
   public void desligar() {
      ligado = false;
   }
   public boolean verificarEstado() {
      return ligado;
   }
   public void aumentarVelocidade() {
      velocidade = velocidade + 1;
   }
   public void diminuirVelocidade() {
      velocidade = velocidade - 1;
   }
   public int verificarVelocidade() {
      return velocidade;
   }
   public static void main(String args[]) {
      Liquidificador l = new Liquidificador();
      System.out.println(l.verificarEstado());
      System.out.println(l.verificarVelocidade());
      l.ligar();
      l.aumentarVelocidade();
      System.out.println(l.verificarEstado ());
      System.out.println(l.verificarVelocidade());
      l.diminuirVelocidade();
      System.out.println(l.verificarVelocidade());
      l.desligar();
      System.out.println(l.verificarEstado ());
   }
}
```

MÉTODOS CONSTRUTORES

- Para inicializar objetos devemos usar um tipo especial de método: os construtores.
- Esse tipo de método não possui tipo de retorno e recebe o nome da classe que o contém.
- Os construtores são utilizados para passar parâmetros de inicialização à instância no ato de sua construção (quando são chamadas por meio da instrução new).
- O construtor é executado apenas uma vez no ciclo de vida de um objeto.
- Toda classe Java deve possuir pelo menos um construtor. Ao detectar que não há construtores definidos no código de uma classe, o compilador javac automaticamente cria um construtor padrão:

```
public NomeClasse() {
}
```

- É possível escrever vários construtores para uma mesma classe, com parâmetros diferentes, com o objetivo de inicializar objetos de maneiras diferentes. Observe:

```
public class Cliente {
  private String nome;
  private String cpf;
  public Cliente() {

  }
```

```
    public Cliente(String nomeCliente) {
      nome = nomeCliente;
    }
    public Cliente(String nomeCliente, String cpfCliente) {
      nome = nomeCliente;
      cpf = cpfCliente;
    }
  }
```

SOBRECARGA DE MÉTODOS (OVERLOADING)

Dois métodos de uma classe podem ter o mesmo nome, desde que suas assinaturas sejam diferentes.

```
public class Sobrecarga {
  public static void main (String args[]) {
    System.out.println("Área do quadrado: " + area(3));
    System.out.println("Área do retântulo: " + area(3,2));
  }
  public static double area(int x) { // quadrado
    return x * x;
  }
  public static double area(int x, int y) { // retângulo
    return x * y;
  }
}
```

8

VARIÁVEIS PRIMITIVAS X VARIÁVEIS DE REFERÊNCIA

VARIÁVEIS PRIMITIVAS

- o valor da variável é o valor literal.

Exemplos:

```
int x = 5;
double y = 7.3;
char z = 'a';
```

Variáveis de instância e locais

- **Variáveis de instância:** são declaradas dentro de uma classe, mas não dentro de um método.
- **Variáveis locais:** são declaradas dentro de um método e sempre devem ser inicializadas.

```
public class Calcula {
  private int a, b = 3; // variáveis de instância
  public int soma() {
    // int total; (não inicializada) provocaria um erro
    int total = a + b; // total é uma variável local e foi inicializada
    return total;
  }
  public static void main(String args[]) {
    Calcula c = new Calcula();
    System.out.println(c.soma()); // imprime 3
  }
}
```

VARIÁVEIS DE REFERÊNCIA

- o valor da variável são bits que representam uma maneira de chegar a um objeto específico.

Exemplo (declaração, criação e atribuição de objetos):

```
Caixa minhaCaixa = new Caixa();
 (1)            (3)     (2)
```

(1) Caixa minhaCaixa: Aloca espaço para variável de referência e nomeia a variável

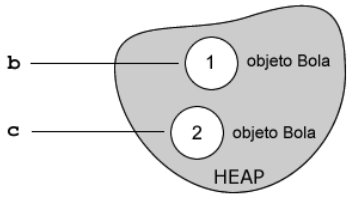
(2) new Caixa(): Aloca espaço para o novo objeto Caixa no HEAP (Pilha de Lixo Coletável)

(3) =: Atribui o novo objeto Caixa à variável minhaCaixa

- Declare duas variáveis de referência Bola **b** e **c**.
- Crie dois novos objetos Bola.
- Atribua os objetos Bola às variáveis de referência.

- Agora dois objetos Bola estão residindo na pilha.

```
Bola b = new Bola();
Bola c = new Bola();
```



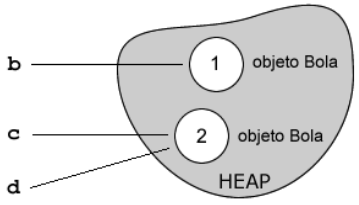
Referências: 2
Objetos: 2

```
public class Bola {
  private String cor = "verde";
  public static void main (String args[]) {
    Bola b = new Bola();
    Bola c = new Bola();
    System.out.println("Cor b: " + b.cor); // verde
    System.out.println("Cor c: " + c.cor); // verde
    c.cor = "amarela";
    System.out.println("Cor b: " + b.cor); // verde
    System.out.println("Cor c: " + c.cor); // amarela
  }
}
```

- Declare uma nova variável de referência Bola **d**.
- Em vez de criar um terceiro objeto Bola, atribua o valor da variável **c** à variável **d**.

- Tanto **c** quanto **d** referenciam o mesmo objeto.

```
Bola d = c;
```



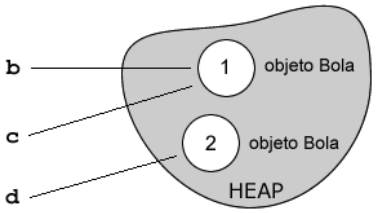
Referências: 3
Objetos: 2

```
public class Bola {
  private String cor = "verde";
  public static void main (String args[]) {
    Bola b = new Bola();
    Bola c = new Bola();
    Bola d = c;
    System.out.println("Cor b: " + b.cor); // verde
    System.out.println("Cor c: " + c.cor); // verde
    System.out.println("Cor d: " + d.cor); // verde
    d.cor = "amarela";
    System.out.println("Cor b: " + b.cor); // verde
    System.out.println("Cor c: " + c.cor); // amarela
    System.out.println("Cor d: " + d.cor); // amarela
  }
}
```

- Atribua o valor da variável **b** à variável **c**.

- Tanto **b** quanto **c** referenciam o mesmo objeto.

```
c = b;
```



Referências: 3
Objetos: 2

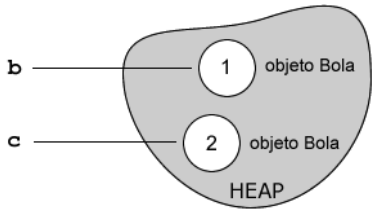
```
public class Bola {
  private String cor = "verde";
  public static void main (String args[]) {
    Bola b = new Bola();
    Bola c = new Bola();
    Bola d = c;
    System.out.println("Cor b: " + b.cor); // verde
    System.out.println("Cor c: " + c.cor); // verde
    System.out.println("Cor d: " + d.cor); // verde
    d.cor = "amarela";
    System.out.println("Cor b: " + b.cor); // verde
    System.out.println("Cor c: " + c.cor); // amarela
    System.out.println("Cor d: " + d.cor); // amarela
    c = b;
    b.cor = "vermelha";
    System.out.println("Cor b: " + b.cor); // vermelha
    System.out.println("Cor c: " + c.cor); // vermelha
    System.out.println("Cor d: " + d.cor); // amarela
  }
}
```

"VIDA" E "MORTE" NA PILHA DE LIXO COLETÁVEL

- Declare duas variáveis de referência Bola **b** e **c**.
- Crie dois novos objetos Bola.
- Atribua os objetos Bola às variáveis de referência.

- Agora dois objetos Bola estão residindo na pilha.

```
Bola b = new Bola();
Bola c = new Bola();
```

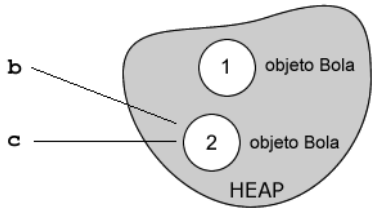


Referências ativas: 2
Objetos alcançáveis: 2

- Atribua o valor da variável **c** à variável **b**.

```
b = c;
```

- Tanto **b** quanto **c** referenciam o mesmo objeto.
- O objeto 1 será abandonado e estará qualificado para a Coleta de Lixo (GC – Garbage Colletion).

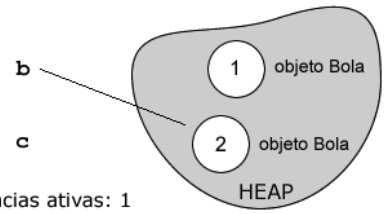


Referências ativas: 2
Objetos alcançáveis: 1
Objetos abandonados: 1

- Atribua o valor **null** à variável **c**. Isso a tornará uma referência nula, o que significa que ela não está referenciando nada. Mas continua a ser uma variável de referência e outro objeto Bola poderá ser atribuído a ela.

- O objeto 2 ainda tem uma referência ativa: **b** e, enquanto a tiver, não estará qualificado para a GC.

```
c = null;
```



Referências ativas: 1
Referências nulas: 1
Objetos alcançáveis: 1
Objetos abandonados: 1

PALAVRA CHAVE: **this**

A palavra-chave this pode ser utilizada para diferenciar atributos de instância e parâmetros de métodos. Assim é possível usar o mesmo nome de variável para parâmetro de método e atributo de instância, obtendo um código mais legível.
No exemplo abaixo, a classe **Produto** tem um único atributo: **codigo** e apenas um método **setCodigo** com um único parâmetro: **codigo**.
O objetivo do método **setCodigo** é receber um valor **(int codigo)** e atribuí-lo ao **codigo** (atributo).

```
public class Produto {
  int codigo; // codigo: atributo
  public void setCodigo(int codigo) { // codigo: parâmetro do método
    codigo = codigo; // ambigüidade (codigo: atributo ou parâmetro do método?)
  }
  public static void main(String args[]) {
    int meuCodigo = 5;
    Produto p = new Produto();
    p.setCodigo(meuCodigo);
    System.out.println(p.codigo); // imprime 0 (motivo: ambigüidade acima)
  }
}
```

Para resolver o problema da ambigüidade devemos utilizar a palavra-chave **this**:

```
public class Produto {
  int codigo;
  public void setCodigo(int codigo) {
    this.codigo = codigo; // this.codigo: atributo, codigo: parâmetro do método)
  }
  public static void main(String args[]) {
    int meuCodigo = 5;
    Produto p = new Produto();
    p.setCodigo(meuCodigo); // o método setCodigo recebeu o valor 5 (meuCodigo)
    System.out.println(p.codigo); // imprime 5
  }
}
```

MODIFICADOR: **static**

O modificador **static** permite criar uma **variável de classe** que poderá ser utilizada diretamente, sem a necessidade de criar uma instância.

Exemplo 1:

```
public class Teste {
  static int x = 5;
  public static void main (String args[]) {
    System.out.println(Teste.x); // imprime 5
  }
}
```

```
public class Teste {
  int x = 5;
  public static void main (String args[]) {
    System.out.println(Teste.x); // erro!
  }
}
```

```
public class Teste {
  int x = 5;
  public static void main (String args[]) {
    Teste t = new Teste();
    System.out.println(t.x); // imprime 5
  }
}
```

Exemplo 2:

```
public class Teste {
  static int x = 1;
  public static void main (String args[]) {
    Teste t1 = new Teste();
    t1.x = t1.x + 1; // t1 e t2 trabalham com a variável da classe
    Teste t2 = new Teste();
    System.out.println(t2.x); // imprime 2
  }
}
```

```
public class Teste {
  int x = 1;
  public static void main (String args[]) {
    Teste t1 = new Teste();
    t1.x = t1.x + 1; // t1 e t2 trabalham com suas respectivas cópias da variável
    Teste t2 = new Teste();
    System.out.println(t2.x); // imprime 1
  }
}
```

9 ENCAPSULAMENTO

Encapsulamento é o processo que impede que variáveis de classe sejam lidas ou modificadas por outras classes. A única maneira de usar essas variáveis é chamando métodos da classe, se estiverem disponíveis.

A linguagem Java oferece quatro níveis de controle de acesso: **public, protected, private** e um ***default*** (sem o uso de modificadores de controle de acesso).

Uma variável ou método declarado sem qualquer modificador de controle de acesso está disponível a qualquer outra classe do mesmo pacote (package).

Lembrete:

- **`public:`** podem ser acessadas por qualquer classe.
- **`protected:`** podem ser acessadas pelas subclasses e por outras classes do mesmo pacote.
- **`private:`** podem ser acessadas somente dentro da própria classe.
- ***`default`*****`:`** podem ser acessadas somente por classes do mesmo pacote.

Observe os exemplos abaixo:

```
public class Teste1 {
   public int a = 5;
}


public class Teste1a {
   public static void main (String args[]) {
      Teste1 t = new Teste1();
      System.out.println(t.a); // Imprime 5
   }
}
```

```
public class Teste2 {
   private int a = 5;
}


// não compila
public class Teste2a {
   public static void main (String args[]) {
      Teste2 t = new Teste2();
      System.out.println(t.a);
   }
}
```

```
public class Teste3 {
   private int a = 5;

   public int lerVariavel() {
      return a;
   }
}


public class Teste3a {

   public static void main (String args[]) {
      Teste3 t = new Teste3();
      System.out.println(t.lerVariavel()); // Imprime 5
   }
}
```

PACOTES (PACKAGES)

Pacotes são diretórios nos quais armazenamos um conjunto de classes que, geralmente, têm um mesmo propósito. Representam as bibliotecas (libraries) em outras linguagens de programação. Alguns exemplos de pacotes no Java: io, awt, beans, lang, net, etc. Todas as classes pertencem a algum pacote. Quando não especificado, a classe passa a pertencer ao pacote default, isto é, a própria pasta onde foi salva.
Um pacote, por convenção, deve ser identificado por letras minúsculas e apresentar o nome reverso do site em que se encontra armazenado: br.com.dominio.
A palavra reservada package seguida do nome do pacote devem ser inseridas na primeira linha de uma classe:

```
package br.com.dominio;
class NomeDaClasse {
  ...
}
```

Como trabalhar com pacotes (passo-a-passo):

1. Criar uma pasta com o nome br.com.dominio.pacotes.

2. Criar e compilar a seguinte classe dentro da pasta acima:

```
package br.com.dominio.pacotes;
public class Teste1 {
  public static int soma(int x, int y) {
    return x+ y;
  }
}
```

3. Fora da pasta criar, compilar e executar a seguinte classe:

```
import br.com.dominio.pacotes.*; // * especifica todas as classes do pacote
public class Teste2 extends Teste1{
  public static void main (String args[]) {
    System.out.println(soma(3,5));
  }
}
```

10 HERANÇA

- Herança é o mecanismo que permite que uma classe herde todo o comportamento e os atributos de outra classe.
- Uma classe que herda de outra classe é chamada de subclasse. A classe que dá a herança é chamada de superclasse.
- Uma classe pode ter somente uma superclasse, mas cada classe pode ter um número ilimitado de subclasses.
- Para executar determinado construtor de uma superclasse usamos a instrução: **super(argumentoDaSuperClasse1, argumentoDaSuperClasse2, ... argumentoDaSuperClasseN)**. Esta deve ser a primeira instrução do construtor da subclasse.

Exemplo 1:

A classe **motorista** herda os atributos e o comportamento (métodos) da classe **funcionario**:

```
public class Funcionario {
  private int matricula;
  private String nome;
  Funcionario (int matricula, String nome) {
    this.matricula = matricula;
    this.nome = nome;
  }
  public int getMatricula() {
    return matricula;
  }
  public String getNome() {
    return nome;
  }
}

public class Motorista extends Funcionario {
  int habilitacao;
  Motorista (int matricula, String nome, int habilitacao) {
    // chama o construtor da superclasse
    // super() deve ser a primeira linha do construtor da subclasse
    super(matricula,nome);
    this.habilitacao = habilitacao;
  }
  public int getHabilitacao() {
    return habilitacao;
  }
}

public class TesteMotorista {
  public static void main (String args[]) {
    Motorista m = new Motorista(1,"João Silva",12345);
    System.out.println("Matricula: " + m.getMatricula());
    System.out.println("Nome: " + m.getNome());
    System.out.println("Habilitação: " + m.getHabilitacao());
  }
}
```

A linguagem Java oferece dois mecanismos que permitem a criação de classes que contenham descrição de campos e métodos sem implementá-los: **classes abstratas** e **interfaces**.

CLASSES ABSTRATAS

Esse tipo de classe oferece um “lugar” para manter atributos e métodos comuns que serão compartilhados pelas subclasses.
Classes abstratas são criadas usando o modificador **abstract** e não podem ser instanciadas.
Uma classe que contém métodos abstratos deve ser declarada como **abstract** mesmo que contenha métodos concretos (não abstratos). Nota: métodos abstratos não têm corpo.
A primeira classe concreta da árvore de herança deverá implementar todos os métodos abstratos.

```
public abstract class Figura {
  private String cor;
  public void setCor(String cor) { // método concreto
    this.cor = cor;
  }
  public String getCor() { // método concreto
    return cor;
  }
  public abstract void calculaArea(double x); // método abstrato

  // A linha abaixo provocaria um erro:
  // uma classe abstrata não pode ser instanciada
  // Figura f = new Figura();

}

public class Quadrado extends Figura {
  public void calculaArea(double x) { // método abstrato implementado
    System.out.println(x*x);
  }
  public static void main (String args[]) {
  Quadrado q = new Quadrado();
  q.setCor("azul");
  System.out.println(q.getCor());
  q.calculaArea(3);
  }
}

public class Circulo extends Figura {
  public void calculaArea(double x) { // método abstrato implementado
    System.out.println(3.14*x*x);
  }
  public static void main (String args[]) {
  Circulo c = new Circulo();
  c.setCor("vermelho");
  System.out.println(c.getCor());
  c.calculaArea(3);
  }
}
```

INTERFACES

As interfaces, similarmente às classes e métodos abstratos, oferecem modelos de comportamento que outras classes deverão implementar. Podemos dizer que interfaces são classes 100% abstratas.
Uma interface Java é uma coleção de comportamento abstrato que pode ser adotada por qualquer classe, sem ser herdada de uma superclasse.
Para usar uma interface devemos incluir a palavra-chave **implements** como parte da definição da classe.
Não podemos obter e escolher apenas alguns métodos ao implementar uma interface, é preciso implementar todos os seus métodos.
Ao contrário da hierarquia de classes, que utiliza a **herança única**, podemos incluir quantas interfaces precisarmos em nossas classes (simulando, de certa forma, a **herança múltipla**):

```
public class NomeDaClasse extends NomeDaSuperClasse
  implements NomeDaInterface1, NomeDaInferface2, ..., NomeDaInterfaceN {
  ...
}
```

Para estender uma interface, utilizamos a palavra-chave **extends**, assim como fazemos em uma definição de classe:

```
interface NomeDaInterface2 extends NomeDaInterface1 {
  ...
}
```

As “variáveis” deverão ser acompanhadas dos modificadores **public static final**, isto é, serão sempre tratadas como constantes. Mesmo que não sejam utilizados estes modificadores, ainda assim, serão consideradas constantes e deverão ser inicializadas.

```
public interface Figura {
  public static final String cor = "azul"; // constante inicializada obrigatoriamente
  // o método setCor não será utilizado visto que cor é agora uma constante
  public abstract String getCor(); // apenas métodos abstratos são permitidos
  public abstract void calculaArea(double x);
}

public class Quadrado implements Figura {
  public String getCor() { // método abstrato implementado
    return cor;
  }
  public void calculaArea(double x) { // método abstrato implementado
    System.out.println(x*x);
  }
  public static void main (String args[]) {
  Quadrado q = new Quadrado();
  // cor á uma constante não pode ter seu valor alterado
  // q.cor="Vermelho";
  System.out.println(q.getCor());
  q.calculaArea(3);
  }
}
```

11 POLIMORFISMO

Através desta técnica avançada uma classe mais genérica (superclasse) pode assumir diferentes comportamentos e gerar objetos diferentes, dependendo de certas condições.

Na prática isto quer dizer que um mesmo objeto pode executar métodos diferentes, dependendo do momento de sua criação.

Como um mesmo objeto pode ser gerado a partir de classes diferentes e classes diferentes possuem métodos diferentes, então o objeto criado poderá ter comportamentos diferentes, dependendo da classe a partir da qual ele foi criado.

O uso do polimorfismo pressupõe duas condições: a existência de herança entre as classes e redifinição de métodos em todas as classes. Todas as classes devem possuir métodos com a **mesma assinatura** (nome, argumentos e retorno), porém com funcionalidades diferentes. Esse mecanismo de redifinição de métodos entre superclasses e subclasses é chamado de **overriding**, diferente do mecanismo de sobrecarga (**overloading**) de métodos **com nomes iguais, mas assinaturas diferentes** que ocorre em uma mesma classe.

Exemplo 1:

```
public class Pessoa {
  public void imprimeClasse() {
    System.out.println("classe Pessoa");
  }
}


public class PessoaFisica extends Pessoa{
  public void imprimeClasse() {
    System.out.println("classe Pessoa Física");
  }
}


public class PessoaJuridica extends Pessoa{
  public void imprimeClasse() {
    System.out.println("classe Pessoa Jurídica");
  }
}


public class PessoaPolimorfa {
  public static void main (String args[]) {
    Pessoa p = null;
    int tipo = 1; // substitua por 2 ou 3 e observe o resultado
    switch (tipo) {
      case 1: p = new Pessoa(); break;
      case 2: p = new PessoaFisica(); break;
      case 3: p = new PessoaJuridica(); break;
    }
    p.imprimeClasse();
  }
}
```

Exemplo 2:

```
public class Conta {
  private int numero;
  protected double saldo;
  public void setSaldo(double saldo) {
    this.saldo = saldo;
  }
  public double getSaldo() {
    return saldo;
  }
}
```

```
public class ContaEspecial extends Conta {
  private double limite = 2000;
  public double getSaldo() {
    saldo = saldo + limite;
    return saldo;
  }
}


public class TesteConta {
  public static void main (String args[]) {
  Conta c = null;
  int tipo = 2;
  switch (tipo) {
    case 1: c = new Conta(); break;
    case 2: c = new ContaEspecial(); break;
  }
  c.setSaldo(1000);
  System.out.println(c.getSaldo());
  }
}
```

12 EXCEÇÕES

Exceção é uma condição anormal que surge em uma seqüência de código durante a execução de um programa. Neste caso, é criado um objeto que representa a exceção e este é enviado ao método que a provocou.

Alguns dos principais objetivos do tratamento de exceções são:

- Detectar e depurar possíveis erros
- Evitar que o programa termine de forma inesperada

Qualquer exceção em Java é uma subclase da classe **`Throwable`**, que por sua vez está dividida em:

- **Exception:** conjunto de exceções que o programa deverá capturar e resolver. Uma de suas subclasses mais importantes é a **`RuntimeException`**.
- **Error:** representam falhas muito graves que geralmente provocam a parada do programa em execução. Algumas de suas principais classes derivadas são: **`AWTError`**, **`LinkageError`**, **`ThreadDeath`**, **`VirtualMachineError`**.

Sintaxe:

```
try {
  código que pode produzir exceção
}

catch {
  código que trata a exceção
}
```

Dentro do **try** colocamos o código que pode produzir objetos do tipo exceção. Logo após, colocamos um bloco **catch**, que é justamente a rotina que manipula a exceção. Portanto, se dentro do **try** for produzido um objeto de tipo exceção, o correspondente **catch** será executado. Por outro lado, não havendo exceções, o código do bloco **catch** não será executado e o fluxo de execução prossegue a partir dos comandos que se seguem ao **catch**.

Na linguagem Java, as várias categorias de exceções são expressas através de uma hierarquia de classes que começa pela classe **Exception**. Por herança, são definidas classes mais específicas de exceções.

Apresentamos, a seguir, algumas destas subclasses:

- **`ClassNotFoundException:`** Exceções produzidas ao utilizar uma classe (construir um objeto, executar um método, etc.
- **`IOException:`** Exceções produzidas ao realizar tarefas de entrada e saída. Algumas classes derivadas: **`EOFException`**, **`FileNotFoundException`**, **`SocketException`**, etc.
- **`RunTimeException:`** Exceções produzidas em tempo de execução. Estão entre as mais utilizadas visto que representam uma grande quantidade de erros que podem aparecer durante a execução de um programa. Algumas de suas principais classes derivadas são: **`ArithmeticException`**, **`IndexOutOfBoundsException`**, **`NegativeArraySizeException`** e **`NullPointerException`**.
- **`SQLException:`** Exceções produzidas ao conectar ou utilizar um banco de dados.

Exemplo:

```
class Excecao1 {
  public static void main (String args[]) {
    int a, b, c;
    a = 2;
    b = 0;
    try {
      c = a / b; // divisão por zero!
    }
    catch (RuntimeException e) {
      System.out.println("Tratando a exceção");
    }
    System.out.println("Prosseguindo após a exceção");
  }
}
```

Para cada **try** podem estar associados vários **catch's** pois, em um bloco **try** podem ser produzidas exceções de classes diferentes. Observe a sintaxe:

```
try {
  // linhas de código que podem produzir exceções de classes diferentes
}
catch (Classe_de_execeção_1 objeto_1) {
  // comandos
}
catch (Classe_de_execeção_2 objeto_2) {
  // comandos
}
...
catch (Classe_de_execeção_n objeto_n) {
  // comandos
}
```

Exemplo:

```
class Excecao2 {
  public static void main(String args[]) {
    int a;
    int b[];
    int c = 1; // c = 0 provoca a Primeira Exceção
    try {
      a = 5 / c; // a > 5 provoca a Segunda Exceção
      System.out.println(a);
      b= new int[5];
      for (int i=0; i<a; i++) {
        b[i] = i;
        System.out.println(i);
      }
    }
    catch (ArithmeticException e) {
      System.out.println("Primeira exceção: divisão por zero " + e);
    }
    catch (ArrayIndexOutOfBoundsException e) {
      System.out.println("Segunda exceção: índice fora do limite " + e);
    }
  }
}
```

Podemos acrescentar a cláusula **finally** à estrutura **try-catch**. O bloco **finally** será **sempre** executado.

Exemplo:

```
class Excecao3 {
  static public void main (String args[]) {
    int a, b, c;
    a = 2;
    b = 1;
    try {
      c = a / b;
    }
    catch (RuntimeException e) {
      System.out.println("Tratando a exceção");
    }
    finally {
      System.out.println("Executando instruções do finally");
    }
    System.out.println("Prosseguindo após a exceção");
  }
}
```

Throws

- A cláusula throws informa que um método pode lançar uma exceção.
- É usada na assinatura do método seguida da exceção que ela deve levantar.

```
public class Teste {
  public void meuMetodo() throws Exception{
    ...
  }
}
```

- Para levantar duas exceções na assinatura do método devemos separá-las por vírgulas:

```
public class Teste {
  public void meuMetodo() throws IOException, ClassNotFoundException{
    ...
  }
}
```

Exemplo 1:

```
public class Excecao4 {
  public void metodo1() {
    try {
      metodo2(1);
      metodo2(0);
    }
    catch (Exception e) {
      System.out.println("Erro: " + e.getMessage());
    }
  }
  public void metodo2(int valor) throws Exception {
    if (valor == 0)
      throw new Exception ("recebi o valor 0");
    System.out.println("O valor " + valor + " é válido");
  }

  public static void main(String args[]) {
  Excecao4 x = new Excecao4();
    x.metodo1();
  }
}
// O valor 1 é válido
// Erro: recebi o valor 0
```

O metodo1 realiza uma chamada ao metodo2. O metodo2 avisa que pode lançar uma exceção: **`throws Exception`**. Caso realmente ocorra uma exceção o metodo2 lançará ela de volta ao metodo1 que, por sua vez, estará preparado para receber e tratar a exceção através da estrutura **`try/catch`**.

Exemplo 2:

```
public class Excecao5 {
  public void divide(int x, int y) throws Exception {
    if (y == 0)
      throw new Exception ("Divisão por zero");
      System.out.println("Resultado: " + x / y);
  }
  public static void main(String args[]) {
    Excecao5 x = new Excecao5();
    try {
      x.divide(2,1);
      x.divide(2,0);
    }
    catch (Exception e) {
      System.out.println("Erro: " + e.getMessage());
    }
  }
}
```

Lembre-se: A palavra chave **`throw`** lança uma Exception (diferente de **`throws`**, que apenas avisa da possibilidade daquele método lançá-la).

CRIANDO SUAS PRÓPRIAS EXCEÇÕES

Java permite definir suas próprias exceções gerando uma subclasse de Exception. O exemplo seguinte mostra como podemos definir um novo tipo de exceção:

```
class MinhaExcecao extends Exception {
  private int n;
  // construtor
  MinhaExcecao(int n) {
    this.n = n;
  }
  public String toString() {
    return "Minha exceção: " + n;
  }
}
```

Para criar uma nova exceção é necessário apenas elaborar uma nova classe que herde da classe Exception e implementar os métodos necessários. A exceção anterior poderá ser utilizada por qualquer outra classe da mesma forma que são utilizadas as outras exceções definidas na linguagem Java.

Observe, a seguir, como como é possível utilizar a exceção criada anteriormente:

```
class TestaExcecao {
  static void calcula(int x) throws MinhaExcecao {
    System.out.println("Valor: " + x);
    if (x > 10)
      throw new MinhaExcecao(x);
    System.out.println("Finalização correta");
  }
  public static void main(String args[]) {
    try {
      calcula(10);
      calcula(11);
    }
    catch (MinhaExcecao e) {
      System.out.println("Capturada: " + e);
    }
  }
}
```

13 ENTRADA DE DADOS

VIA CONSOLE

```
//importa pacote java.io para usar as classes InputStreamReader e BufferedReader
import java.io.*;
public class SeuNome {
  public static void main (String args[]) {
    // cria uma instância da classe InputStreamReader e atribui à referência ent
    // recebe um fluxo de entrada em bytes e converte em caracteres
    InputStreamReader ent = new InputStreamReader(System.in);
    // cria uma instância da classe BufferedReader e atribui à referência cons
    // recebe o leitor (classe InputStreamReader) e cria um buffer de caracteres
    BufferedReader cons = new BufferedReader(ent);
    String nome;
    System.out.println("Digite o seu nome: ");
    // o uso de try ... catch é obrigatório quando utilizamos classes do java.io
    try {
      // o método readLine da classe BufferedReader lê a entrada de dados
      nome = cons.readLine();
      System.out.println("Você digitou: " + nome);
    }
    catch(IOException ioex){
      System.out.println("Erro na entrada de dados");
      // o valor 1 indica que o programa terminou de maneira prematura
      System.exit(1);
    }
    System.exit(0);
  }
}
```

SWING

```
import javax.swing.JOptionPane;

public class Teste {

   public static void main (String args[]) {

      String strN1, strN2;
      int intN1, intN2, soma;

      strN1 = JOptionPane.showInputDialog ("Digite o 1o numero:");
      strN2 = JOptionPane.showInputDialog ("Digite o 2o numero:");

      intN1 = Integer.parseInt(strN1);
      intN2 = Integer.parseInt(strN2);

      soma = intN1 + intN2;

      JOptionPane.showMessageDialog(null, soma, "Total:", JOptionPane.INFORMATION_MESSAGE);
      System.exit(0);
   }
}
```

14 APPLETS

Applets são aplicações inseridas em páginas HTML que podem ser executadas pelo browser.

As applets são consideradas aplicações clientes porque rodam na máquina do usuário e não no servidor remoto (web server).

O ciclo de vida de um applet é constituído por métodos que são invocados de acordo com certas ações que ocorrem durante sua execução e é composto por seis estágios apresentados a seguir:

1. **Instanciação:** Ocorre no momento em que o applet é criado.
```
public class NomeDoApplet extends Applet {
  ...
}
```
2. **`init()`:** Inicia a execução do applet no browser.
3. **`start()`:** Torna visível o applet no browser.
4. **`stop()`:** Torna invisível o applet no browser.
5. **`paint()`:** Atualiza a exibição do applet no browser (janela: redimensionada, minimizada, maximizada).
6. **`destroy()`:** Libera os recursos de memória utilizados durante a execução.

Exemplo:

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

public class TesteApplet extends Applet
{
  String text;
  int cont = 1;
  TextField T1, T2;

  public void init()
   {
   T1 = new TextField("Testando o applet...");
   T2 = new TextField();
   setLayout(new GridLayout(2,1));
   add(T1);
   add(T2);
   }
  public void start()
   {
   System.out.println("Iniciando o applet ...");
   }
  public void stop()
   {
   System.out.println("Parando o applet ...");
   }
  public void destroy()
   {
   System.out.println("Destruindo o applet ...");
   }
  public void paint(Graphics g)
   {
    T2.setText("Atualizou " + cont + " vezes");
    cont++;
   }
}
```

```
<!–TesteApplet.html -->
<HTML>
<APPLET code="TesteApplet.class" width=300 height=70></APPLET>
</HTML>
```

Para visualizar utilize o appletviwer. Digite no prompt de comando:

```
appletviewer TesteApplet.html
```

Exemplo:

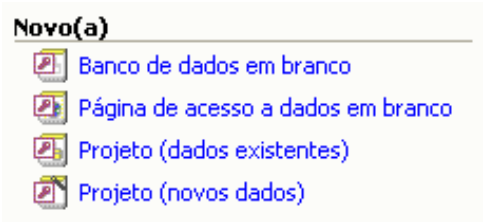
```
// Calculadora.java
import java.applet.Applet;
import java.awt.*;

public class Calculadora extends Applet {
  Label L1, L2, L3;
  TextField T1, T2, T3;
  Button B1, B2, B3, B4;
  double n1, n2, resultado;

  public void init() {
    L1 = new Label("N1:");
    T1 = new TextField(10);
    L2 = new Label("N2:");
    T2 = new TextField(10);
    L3 = new Label("Resultado:");
    T3 = new TextField(10);
    B1 = new Button(" + ");
    B2 = new Button(" - ");
    B3 = new Button(" x ");
    B4 = new Button(" / ");
    add(L1);
    add(T1);
    add(L2);
    add(T2);
    add(L3);
    add(T3);
    add(B1);
    add(B2);
    add(B3);
    add(B4);
  }

  public boolean action(Event evento, Object objeto) {
    try {
      n1 = Double.parseDouble(T1.getText());
      n2 = Double.parseDouble(T2.getText());
      if (evento.target == B1)
        resultado = soma(n1,n2);
      if (evento.target == B2)
        resultado = subtrai(n1,n2);
      if (evento.target == B3)
        resultado = multiplica(n1,n2);
      if (evento.target == B4)
        resultado = divide(n1,n2);
      T3.setText(Double.toString(resultado));
    }
    catch (Exception e) {
      T3.setText(e.toString());
    }
    return true;
  }

  public double soma (double x, double y) {
    return x + y;
  }
  public double subtrai (double x, double y) {
    return x - y;
  }
  public double multiplica (double x, double y) {
    return x * y;
  }
  public double divide (double x, double y) throws Exception {
    if (n2 == 0)
      throw new Exception();
    return x / y;
  }
}

<!—Calculadora.html -->
<HTML>
<APPLET code="Calculadora.class" width=600 height=100></APPLET>
</HTML>
```

15 BANCO DE DADOS

1. Criar um novo banco de dados

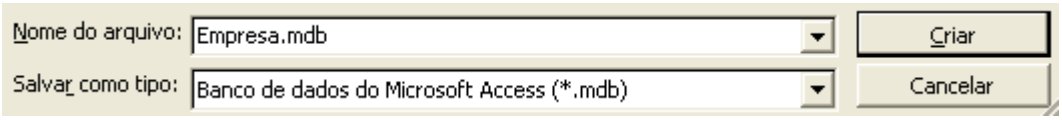
Vamos criar um novo banco de dados chamado **Empresa**:

- No Access XP escolha **Novo ➔ Banco de dados em branco**



- Digite o nome do banco de dados: **Empresa.mdb** e clique em **Criar**.

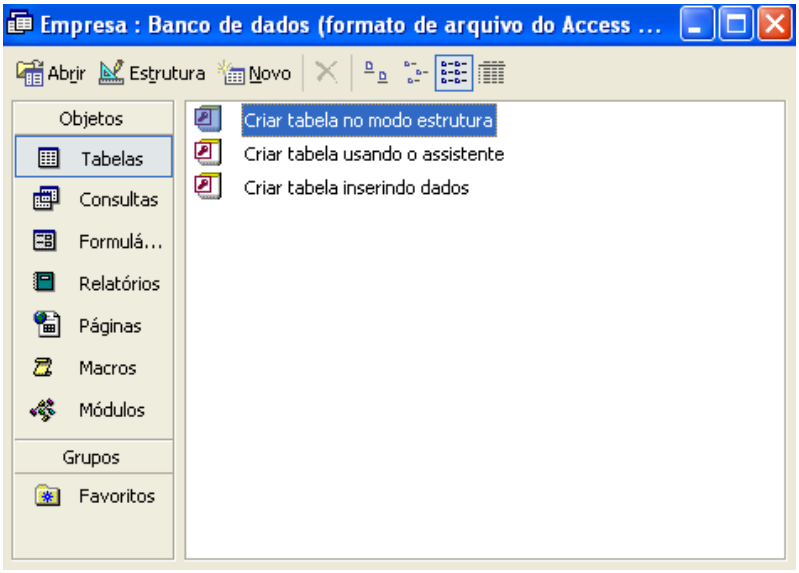
  Nota: Crie o arquivo acima na mesma pasta dos arquivos **.class**



2. Criar uma tabela

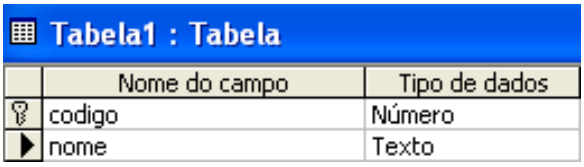
Agora vamos criar uma tabela chamada **Cliente**:

- Selecione **Criar tabela no modo estrutura**:

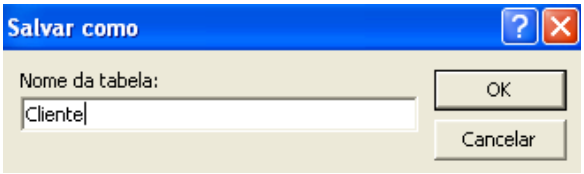


- Crie a tabela conforme a estrutura apresentada abaixo:

  Nota: Escolha o campo **codigo** como chave primária (selecione a linha e clique no ícone chave)



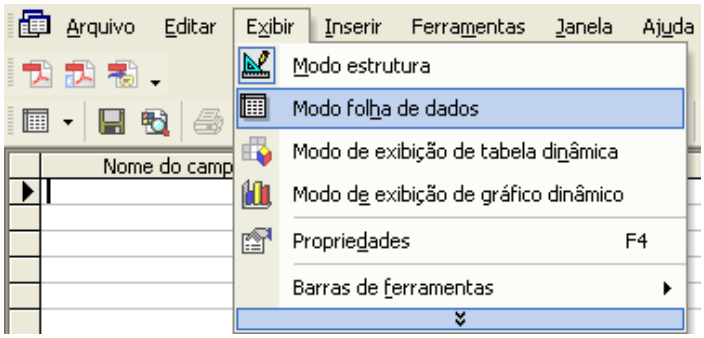
- Salve a tabela com o nome cliente:



3. Inserir dados na tabela

Agora vamos inserir algumas linhas na tabela **Cliente**:

- Selecione **Exibir ➔ Modo folha de dados**:



- Insira algumas linhas na tabela:

	codigo	nome
	1	Antonio
	2	Beatriz
	3	Claudio
	4	Daniela

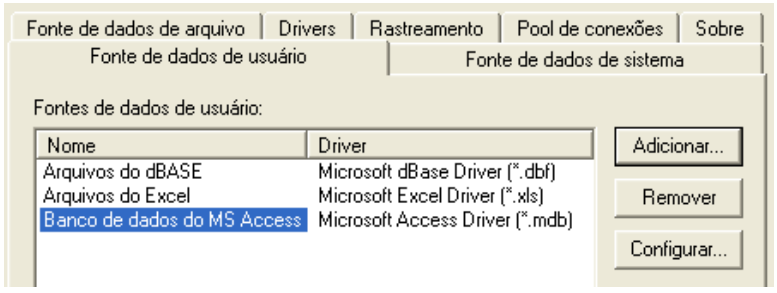
- Salve e feche o Access.

4. Criar fontes de dados ODBC

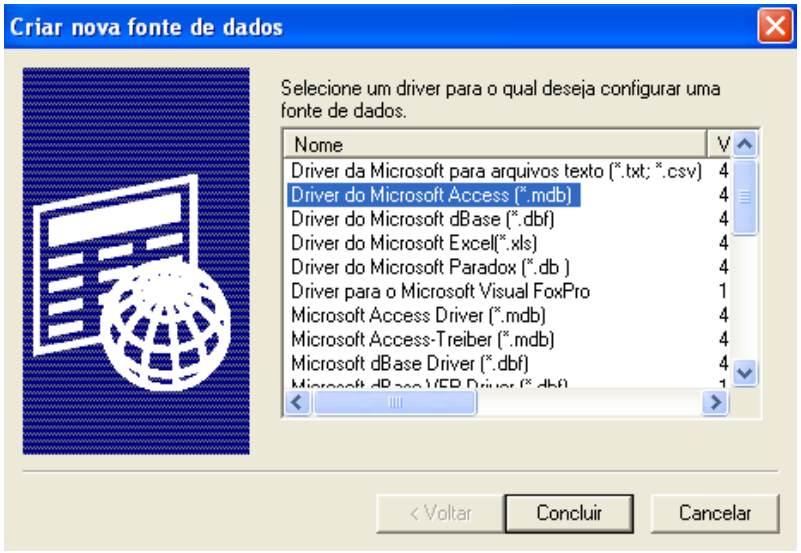
ODBC (Open Database Connectivity) é uma interface com grande disponibilidade de drivers para acesso a diferentes bancos de dados (dBaseIII, Access, SQL Server, Oracle, etc.).

JDBC (Java Database Conectivity) é a interface que possibilita que as aplicações Java acessem bancos de dados relacionais. É a forma mais prática e rápida de conectar uma aplicação Java a um arquivo de dados.

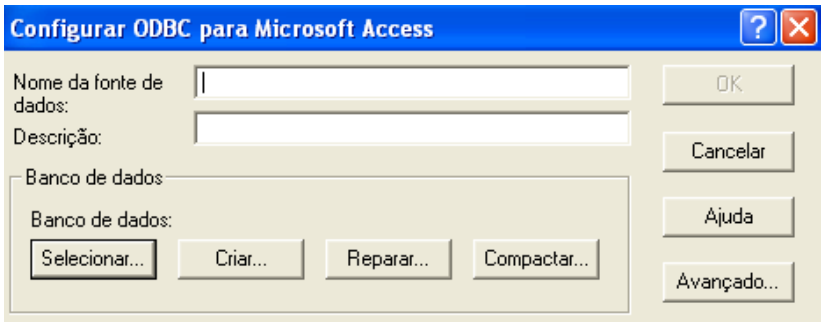
- Selecione **Iniciar ➔ Painel de Controle ➔ Ferramentas administrativas ➔ Fontes de Dados (ODBC)**
- Na aba **Fonte de dados de usuário** selecione **Banco de Dados do MS Access** e clique em **Adicionar**:



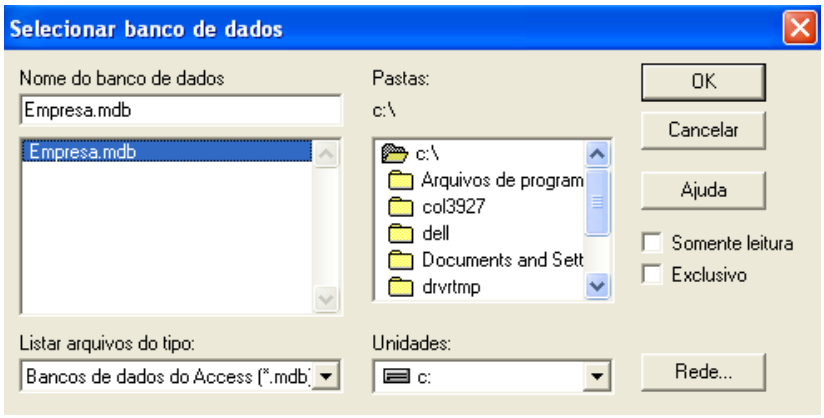
- Na próxima janela selecione **Driver do Microsoft Access (*.mdb)** e clique em **Concluir**:



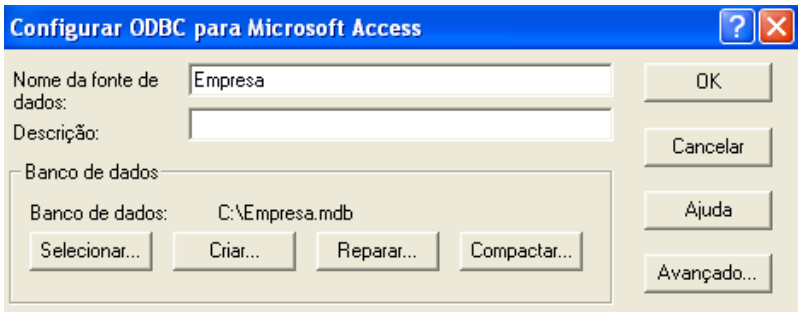
- Para escolher o **Banco de Dados** clique em **Selecionar**:



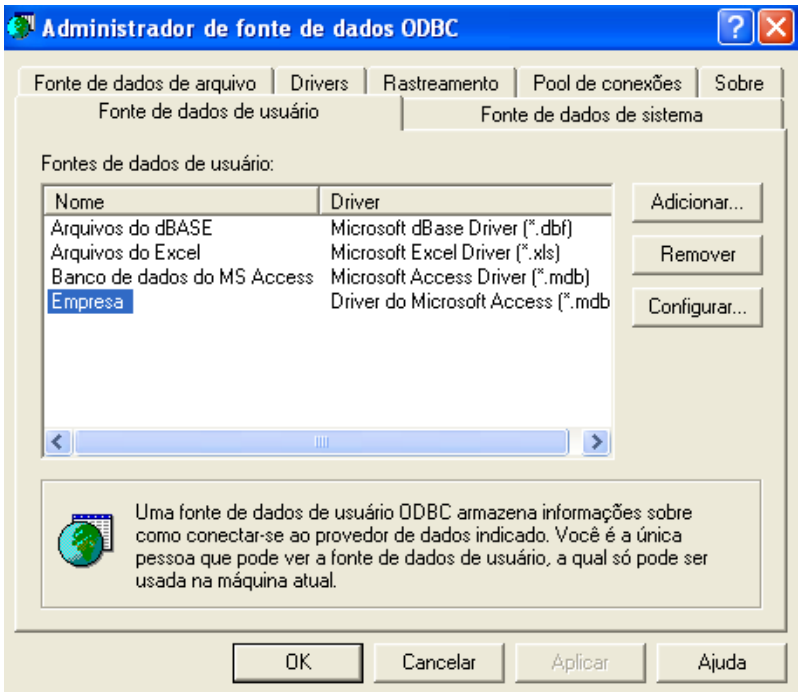
- Selecione a pasta adequada e o banco de dados **Empresa.mbd** e clique em **OK**:



- Digite o Nome da fonte de dados: **Empresa** e clique em **OK**:



- A fonte de dados: **Empresa** foi criada. Para concluir clique em **OK**.



5. Testar a conexão

```
import java.sql.*;
class Conecta {
  public static void main (String args[]) {
    try {
      String url = "jdbc:odbc:Empresa";
      String usuario = "";
      String senha = "";
      Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
      Connection con;
      con = DriverManager.getConnection(url,usuario,senha);
      System.out.println("Conexão realizada com sucesso.");
      con.close();
    }
    catch(Exception e) {
      System.out.println("Problemas na conexão.");
    }
  }
}
```

6. Selecionar dados

```
import java.sql.*;
class Seleciona {
  public static void main (String args[]) {
    try {
      String url = "jdbc:odbc:Empresa";
      String usuario = "";
      String senha = "";
      Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
      Connection con;
      con = DriverManager.getConnection(url,usuario,senha);
      Statement st = con.createStatement();
      ResultSet rs = st.executeQuery("SELECT * FROM Cliente");
        while (rs.next()) {
          System.out.print(rs.getString("Codigo") + " - ");
          System.out.println(rs.getString("Nome"));
        }
      System.out.println("Operação realizada com sucesso.");
      st.close();
      con.close();
    }
    catch(Exception e) {
      System.out.println("Problemas na conexão.");
    }
  }
}
```

7. Inserir dados

```
import java.sql.*;
class Insere {
  public static void main (String args[]) {
    try {
      String url = "jdbc:odbc:Empresa";
      String usuario = "";
      String senha = "";
      Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
      Connection con;
      con = DriverManager.getConnection(url,usuario,senha);
      Statement st = con.createStatement();
      st.executeUpdate("INSERT INTO Cliente VALUES (5,'Ernesto')");
      System.out.println("Operação realizada com sucesso.");
      st.close();
      con.close();
    }
    catch(Exception e) {
      System.out.println("Problemas na conexão.");
    }
  }
}
```

8. Alterar dados

```
import java.sql.*;
class Altera {
  public static void main (String args[]) {
    try {
      String url = "jdbc:odbc:Empresa";
      String usuario = "";
      String senha = "";
      Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
      Connection con;
      con = DriverManager.getConnection(url,usuario,senha);
      Statement st = con.createStatement();
      st.executeUpdate("UPDATE Cliente SET nome = 'Fabiana' WHERE codigo = 5");
      System.out.println("Operação realizada com sucesso.");
      st.close();
      con.close();
    }
    catch(Exception e) {
      System.out.println("Problemas na conexão.");
    }
  }
}
```

9. Excluir dados

```
import java.sql.*;
class Exclui {
  public static void main (String args[]) {
    try {
      String url = "jdbc:odbc:Empresa";
      String usuario = "";
      String senha = "";
      Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
      Connection con;
      con = DriverManager.getConnection(url,usuario,senha);
      Statement st = con.createStatement();
      st.executeUpdate("DELETE FROM Cliente WHERE codigo = 5");
      System.out.println("Operação realizada com sucesso.");
      st.close();
      con.close();
    }
    catch(Exception e) {
      System.out.println("Problemas na conexão.");
    }
  }
}
```

16 BIBLIOGRAFIA

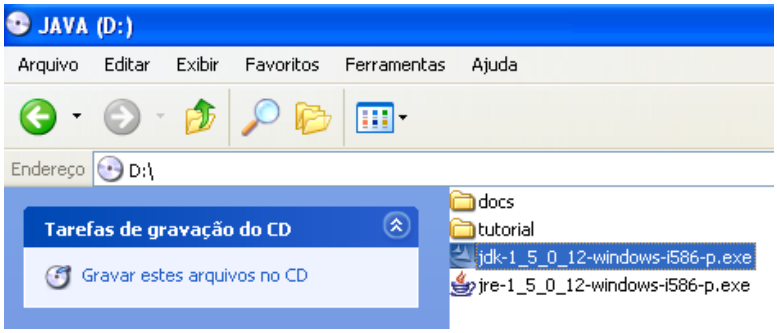
- DEITEL. Java - como programar. 6ª ed. São Paulo: Pearson / Prentice Hall, 2005.
- SANTOS, RAFAEL. Introdução à programação orientada a objetos usando Java. Rio de Janeiro: Elsevier, 2003.

17 SITES RECOMENDADOS

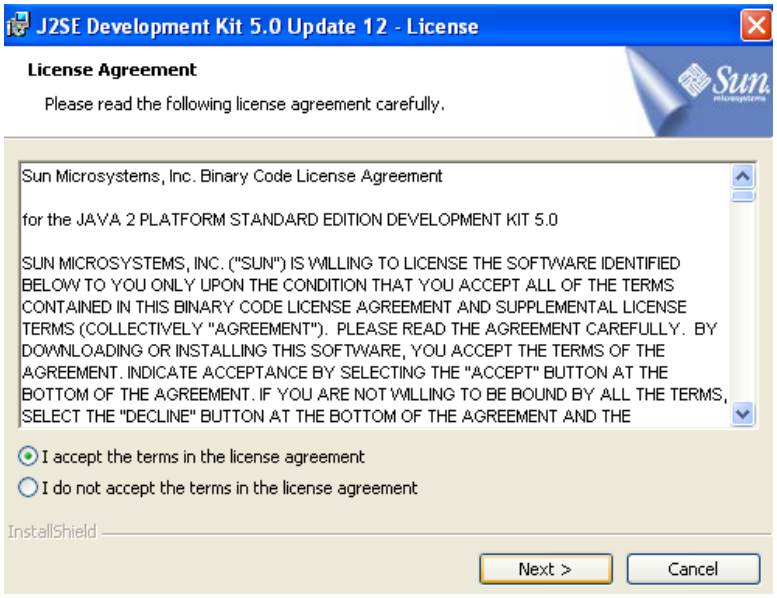
- The Java Tutorials: http://java.sun.com/docs/books/tutorial
- Java Free: http://www.javafree.org/index.jf

APÊNDICE A: INSTALAÇÃO DO JDK 1.5

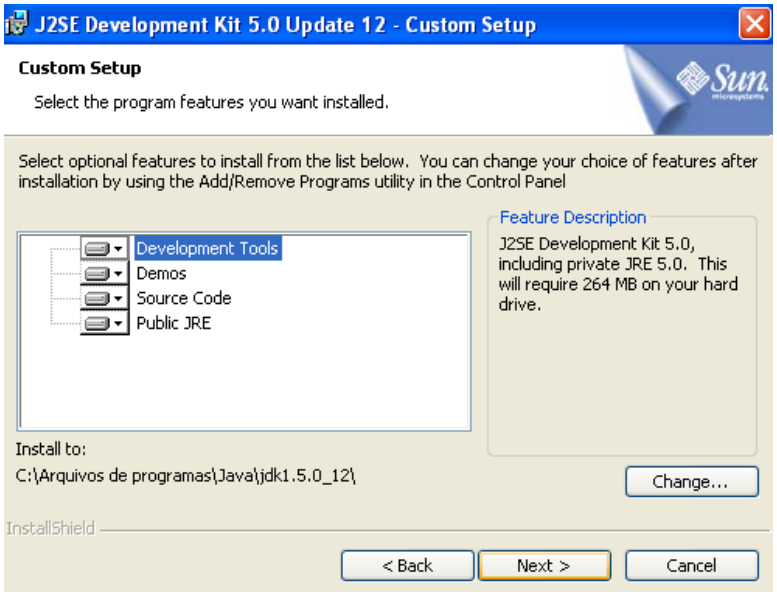
Duplo clique no arquivo: jdk-1_5_0_12-windows-i586-p.exe



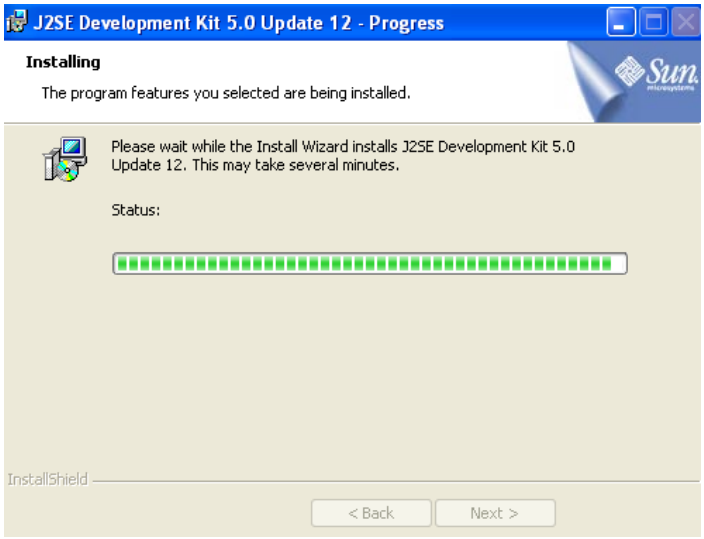
Clicar em "I accept the terms in the license agreement" e clicar em "Next".



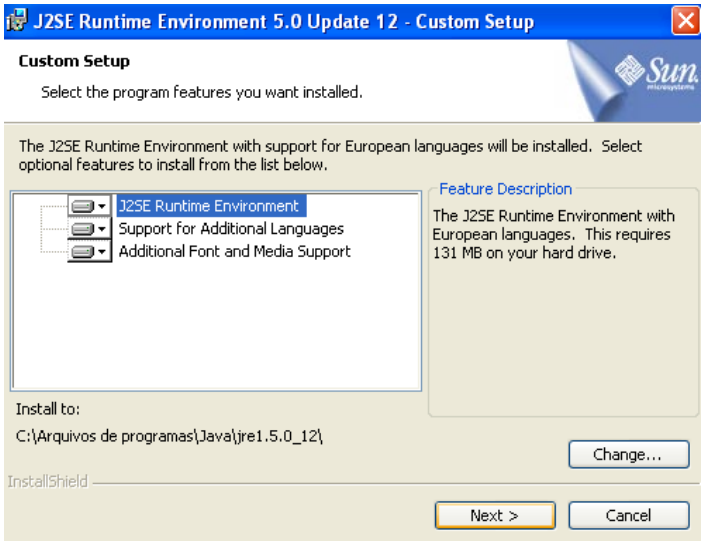
Aceite o local sugerido para instalação (C:\Arquivos de programas\Java\jdk1.5.0_12)



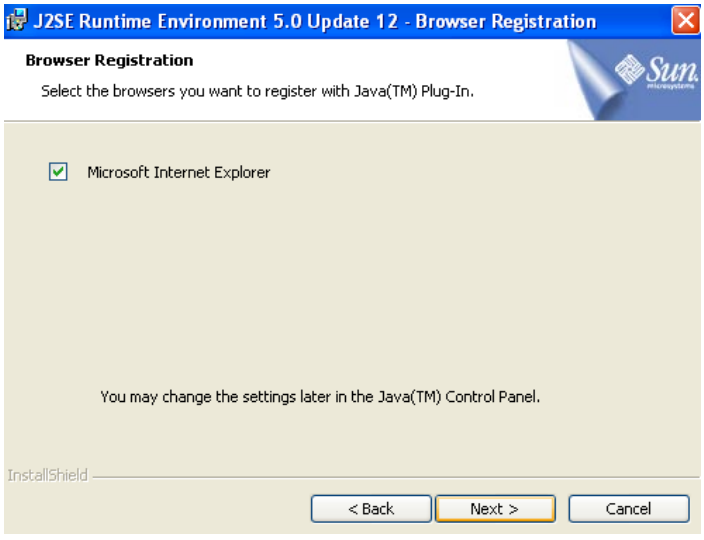
Aguarde o término da instalação.



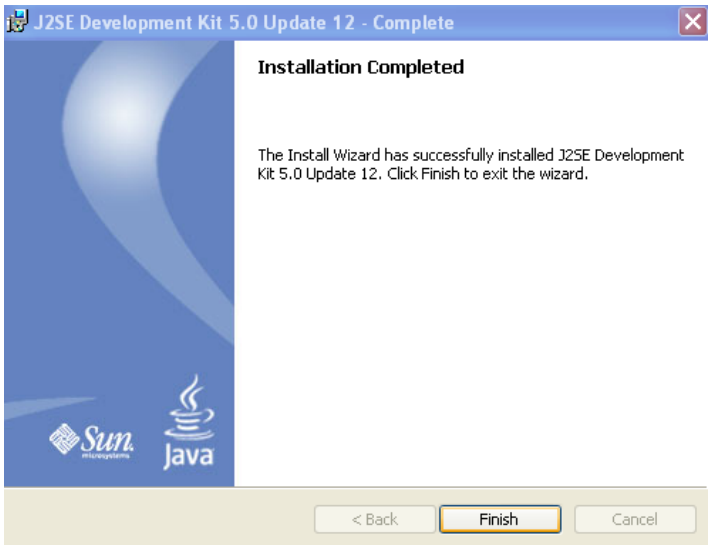
Aceite o local sugerido para instalação do JRE - J2SE Runtime Environment



Selecione o browser que oferecerá suporte ao Java.



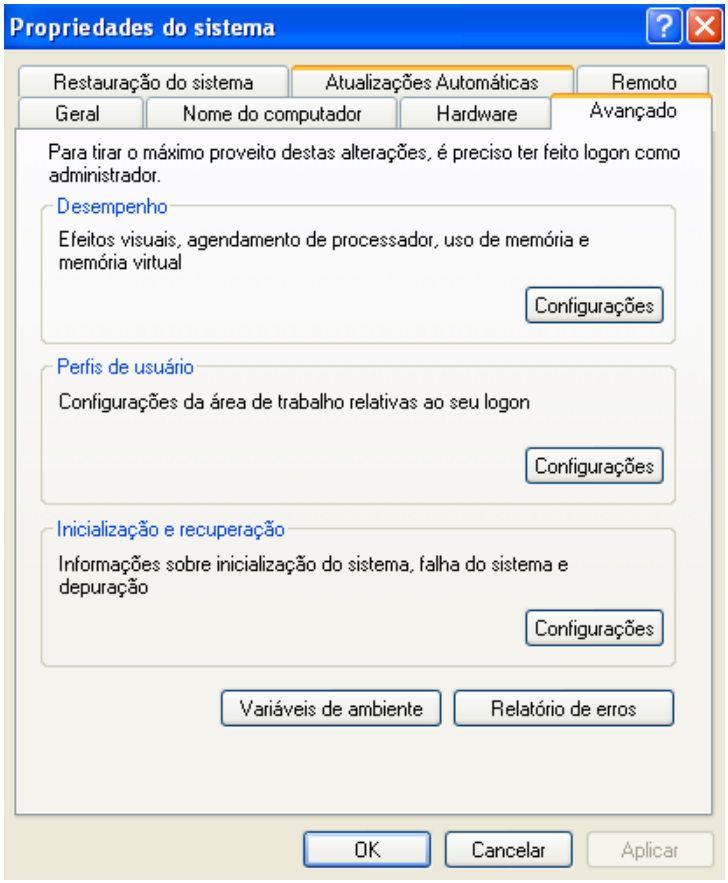
A instalação está completa! Clique em "Finish".



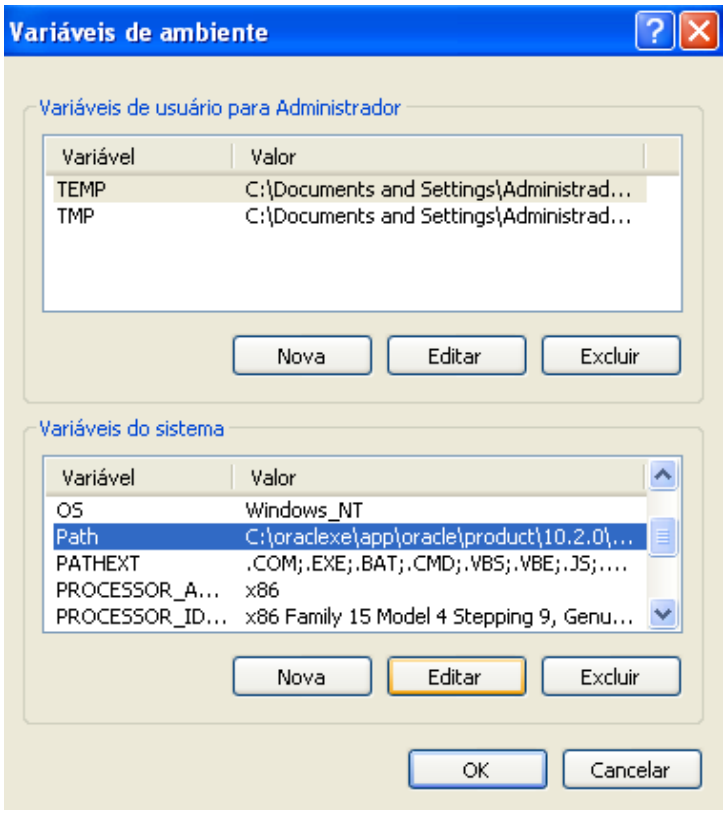
Configurar a variável de ambiente Path (para indicar onde o JDK está instalado)

Selecione: "Iniciar" e clique com o botão direito do mouse em "Meu Computador". Clique em "Propriedades".

Selecione a aba "Avançado" e clique na opção "Variáveis de ambiente".

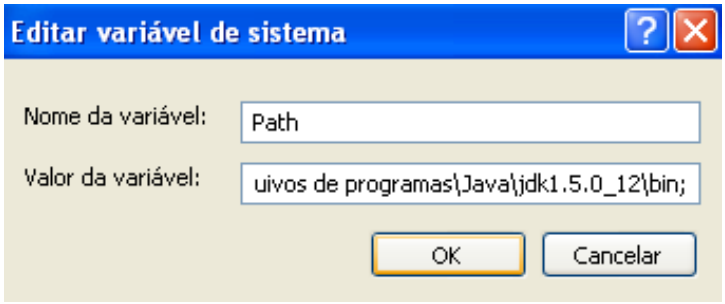


Na janela "Variáveis do sistema" selecione a opção "Path" e clique em "Editar".



No campo "Valor da variável" faça o seguinte:

1. Não apague nada.
2. Verifique se a linha termina com um ; (ponto e vírgula).
3. Se não terminar, acrescente-o e digite logo após o seguinte:
   `C:\Arquivos de programas\Java\jdk1.5.0_12\bin;`



Clique em OK ... OK ... OK

Terminamos! O ambiente está preparado.

APÊNDICE B: EXERCÍCIOS

ESTRUTURAS DE DECISÃO E REPETIÇÃO

Criar uma classe chamada Temperatura para apresentar as temperaturas em graus Celsius entre -50 e +50 e, ao lado, os valores correspondentes em graus Fahrenheit. Fórmula de conversão: f = c*9/5+32. Veja a saída (parcial):

```
Celsius: -50.0 Fahrenheit: -58.0
...
Celsius: 50.0 Fahrenheit: 122.0
```

Criar uma classe chamada Tabuada para apresentar as tabuadas de 1 a 10. Veja a saída (parcial):

```
1 x 1 = 1
...
10 x 10 = 100
```

Criar uma classe chamada Relogio para apresentar as horas e minutos de um dia: das 0:00 até 23:59.

Criar uma classe chamada Bissexto para apresentar os anos bissextos entre 2000 e 2100. Um ano será bissexto quando for possível dividi-lo por 4, mas não por 100 ou quando for possível dividi-lo por 400.

Criar uma classe chamada Cilindro para apresentar os volumes de cilindros cujos raios e alturas variem de 1 a 10.São dados: volume = Π * raio^2 * altura. Sendo Π = 3.14. Veja a saída (parcial):

```
raio: 1 altura: 1 volume: 3.14
...
raio: 10 altura: 10 volume: 3140.0
```

Criar uma classe chamada Tabuleiro para elaborar um tabuleiro de damas ou xadrez binário. As casas brancas deverão ser representadas por 0 (zero) e as pretas por 1 (um). Dica: utilizar para saída: **`System.out.print`**. Veja o resultado:

```
0 1 0 1 0 1 0 1
1 0 1 0 1 0 1 0
0 1 0 1 0 1 0 1
1 0 1 0 1 0 1 0
0 1 0 1 0 1 0 1
1 0 1 0 1 0 1 0
0 1 0 1 0 1 0 1
1 0 1 0 1 0 1 0
```

Criar uma classe chamada Fibonacci para apresentar a seqüência até o décimo termo: **0 1 1 2 3 5 8 13 21 34**

HERANÇA

Criar uma superclasse chamada Produto com os seguintes atributos: int codigo, double preco e com os seguintes métodos getCodigo (retorna codigo) e getPreco (retorna preco). Em seguida, criar duas subclasses que herdem de Produto. Classe Livro com os seguintes atributos: String autor e String editora e com os seguintes métodos: getAutor (retorna autor) e getEditora (retorna editora). Classe Disco com os seguintes atributos: String artista e String gravadora e com os seguintes métodos: getArtista (retorna artista) e getGravadora (retorna gravadora). E, finalmente, criar uma classe TestaProduto com duas instâncias: uma para classe Livro e outra para classe Disco. Ambas deverão receber valores que correspondam a cada um de seus atributos e, em seguida, imprimi-los.

APÊNDICE C: EXEMPLOS COMPLEMENTARES

```
// Operador + com dois inteiros e um inteiro e uma String

class A {
  int a = 1;
}

class B {
  int b = 2;
}

class C {
  String c = "2";
}

public class Duas {
  public static void main (String args[]) {
    A a = new A();
    B b = new B();
    C c = new C();
  System.out.println(a.a + b.b);  // dois inteiros: imprime 3
  System.out.println(a.a + c.c);  // um inteiro e uma String: imprime 12
  }
}
```

NOTAS:

- Observar que no exemplo acima foram usados identificadores iguais para variáveis de referência e variáveis primitivas (a, b, c), prática não recomendada, mas tratada adequadamente, sem conflitos, pelo Java.
- O arquivo tem quatro classes, mas apenas uma **public**.