

1. Como tudo começou:

A linguagem Java foi desenvolvida pela Sun Microsystems em 1991 em um projeto liderado por seus profissionais Patrick Naughton e James Gosling. Inicialmente a linguagem recebeu o nome de *7(star seven), mas esta denominação não foi bem aceita pela comunidade de desenvolvedores sendo, então, alterada para OAK, que significa carvalho em inglês. Porém, não muito depois, foi descoberto que já havia uma outra linguagem com este mesmo nome, sendo assim acabou recebendo o nome Java, uma ilha localizada na Indonésia, caracterizada por preparar um café de sabor muito especial.

A idéia primária da equipe era desenvolver uma linguagem de propósito específico destinada para dispositivos eletrônicos como decodificadores de TV a cabo. Para que fosse suportada por diferentes tipos de dispositivos ela deveria ter como uma de suas principais características a capacidade de ser uma linguagem portátil, que pudesse ser executada em qualquer destes dispositivos independentemente do hardware neles contidos. E, que devido à limitada capacidade computacional destes dispositivos os programas produzidos teriam que ser pequenos e eficazes. Assim foi feito. Desta forma simplória a linguagem Java iniciou sua trajetória.

Com a expansão da Internet, iniciada em meados da década de 90, a equipe de Naughton e Gosling vislumbrou a exploração deste segmento pela linguagem Java, orientando-a para o desenvolvimento de aplicações para a *web*. Surgem, então, os applets, pequenos programas escritos em Java que eram descarregados pelos *browsers* para as máquinas cliente, onde eram executados. Estes applets passaram a acrescentar elementos dinâmicos de conteúdo às páginas *web* que até então não existiam. Com a grande aceitação dos applets pelo mercado, em 1996, os mais populares *browsers* da época, o Netscape e o Microsoft Internet Explorer, passaram a ter a capacidade de executar applets. Assim, o Java ganha notoriedade e inicia sua meteórica inserção no mercado tecnológico mundial.

Em 2008, a Sun Microsystems foi adquirida pela Oracle Corporation.

Ainda que a linguagem Java tenha se voltado para o ambiente de redes, sua utilidade não se restringe somente a esta característica. O Java é, na verdade, uma linguagem de propósito geral que permite que programas sejam escritos explorando as seguintes facilidades:

1 Orientação a objetos.

Java atende a todos os requisitos para que seja considerada uma linguagem orientada a objetos: hereditariedade, encapsula atributos e métodos e suporta polimorfismo.

2 Multi-threading.

Oferece recursos que permitem o desenvolvimento de aplicações que explorem o paralelismo na execução de tarefas.

3 Manuseio estruturado de erros.

Como toda linguagem confiável o Java possui mecanismos para o tratamento das

possíveis exceções lançadas pelos programas.

4 Coleta de lixo.

A Java Virtual Machine oferece a facilidade de, periodicamente, realizar o expurgo daqueles objetos localizados na memória e não mais referenciados.

5 Carga dinâmica de classes.

Permite que classes possam ser carregadas em qualquer tempo do processamento, evitando reinícios indesejáveis dos aplicativos.

Uma outra importante virtude da linguagem Java é a sua independência de plataforma. Um sistema escrito, por exemplo, em uma plataforma Windows-Intel poderá ser executado, sem nenhuma intervenção de seus desenvolvedores, em uma plataforma Solaris-SPARC, comprovando o lema preconizado por James Gosling: “Escreva uma vez, execute em qualquer lugar”. Além disto, a escalabilidade do Java é outro fator essencial, uma vez que ele pode ser utilizado em uma vasta gama de máquinas, indo desde pequenos dispositivos embutidos até aos tradicionais *mainframes*. Portanto, os sistemas desenvolvidos em Java podem ser executados por máquinas de diferentes dimensões e capacidades.

2. Instalando o Java em seu Sistema:

A Oracle disponibiliza quatro plataformas distintas para o desenvolvimento de aplicações utilizando as funcionalidades oferecidas pelo Java, são elas:

1 Java 2 Platform, Micro Edition (J2ME)

- Plataforma orientada ao desenvolvimento de aplicações para dispositivos móveis tais como celulares, PDAs, pagers, e dispositivos embutidos. Inclui a Java Virtual Machine e um conjunto padrão de APIs pré-definidas pelas empresas fabricantes dos dispositivos e pelos provedores dos serviços de conteúdo.

2 Java 2 Platform, Standard Edition (J2SE)

- Plataforma orientada para desenvolver aplicações mais robustas, tais como aplicações para desktops e servidores de aplicações. É subdividida em dois principais produtos: Java Runtime Environment (JRE) e Java Development Kit (JDK). O JRE contém APIs Java, a Java Virtual Machine e outros componentes necessários para a execução de applets. O JRE precisa residir em toda e qualquer máquina que tenha a intenção de executar aplicativos em Java. O JDK abrange um conjunto maior de características. Além de incorporar todas aquelas já embutidas no JRE também contém o compilador Java e depuradores, portanto, precisa ser instalado em qualquer máquina voltada para o desenvolvimento de aplicativos em Java.

3 Java 2 Enterprise Edition (J2EE)

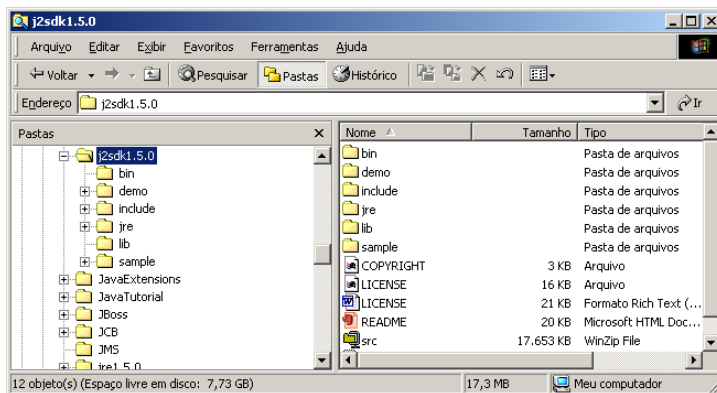
- É a plataforma mais robusta. Indicada para o desenvolvimento de aplicações corporativas de larga escala e de múltiplas camadas. Engloba todas as funcionalidades das plataformas J2ME e J2EE agregando todo o conjunto de elementos necessários para desenvolver sistemas voltados para a *web*.

4 Java Fx

- O JavaFX é uma biblioteca gráfica da plataforma Java. Permite criar: interfaces gráficas, animações, desenhar na tela, efeitos, gráficos, programar arrastando e soltando e tocar vídeo e áudio.

A seguir são apresentados os passos necessários para fazer a descarga e a instalação do Java:

1. No site da Oracle, java.com, selecione a plataforma de desenvolvimento de seu interesse. Em nosso caso J2SE.
2. Selecione o ambiente operacional onde a plataforma será instalada (Windows/Linux/Solaris).
3. Opte pela versão disponível do Java (Java 1.8).



Conforme ilustrado na figura ao lado, após o término do passo 3 são criados, no diretório de instalação (`jdk1.8.0_91`), os seguintes subdiretórios: `bin`, `include`, `jre` e `lib`.

No diretório `bin` estão contidos todos os arquivos necessários para a execução e teste de um programa escrito em Java, incluindo aplicações desktops.

O diretório `jre` contém os arquivos necessários para a execução dos programas Java já compilados, estes arquivos devem ser distribuídos juntamente com a aplicação desenvolvida uma vez que contém a Java Virtual Machine. Todo computador,

independentemente de seu porte, que deseja executar uma aplicação escrita em Java precisa abrigar uma Java Virtual Machine.

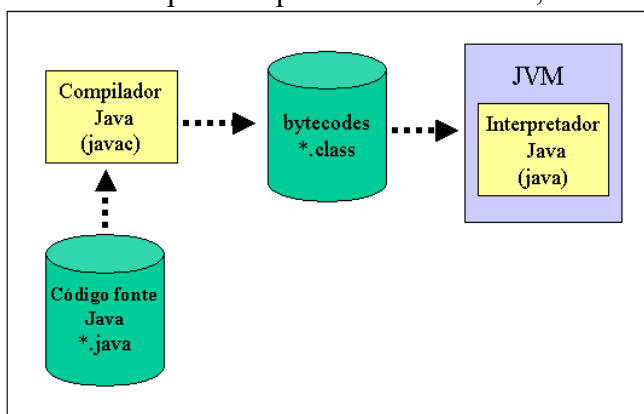
O diretório *include* contém arquivos da linguagem C que podem ser incorporados em um programa Java, explorando uma facilidade denominada Java Native Code em que um programa Java pode invocar programas escritos em outras linguagens como C, C++ ou Assembler. O diretório *lib* contém pacotes complementares de apoio ao desenvolvimento dos aplicativos.

O código fonte das classes que integram a plataforma descarregada encontra-se no arquivo *src.zip*.

Após o término da instalação, o sistema operacional que abrigou a plataforma precisa ser adequadamente configurado. O sub-diretório *bin*, gerado durante o processo de instalação do Java, e que contém o compilador e o interpretador da linguagem, deve ser incluído no *command path* para facilitar as etapas de compilação e testes dos aplicativos.

3. Fluxo de execução de um programa Java:

A figura abaixo apresenta o fluxo de execução de um programa escrito em Java. Todo programa fonte Java deve possuir um nome cuja extensão deve ser *.java*. e para que este programa tenha a capacidade de ser executado em qualquer plataforma operacional o compilador Java, denominado *javac*, gera um arquivo, de extensão *.class*, cujo conteúdo é chamado de *bytecodes*. Estes *bytecodes* são os códigos de máquina que serão interpretados e traduzidos pela máquina virtual do Java, a Java Virtual Machine (JVM).



A Java Virtual Machine lê e traduz os *bytecodes* para o código nativo da máquina onde ela está sendo executada, isentando do programador a preocupação em saber não só para qual hardware ele está programando como também para qual sistema operacional ele está desenvolvendo, estas são preocupações únicas da Java Virtual Machine. A Java Virtual Machine estabelece o elo de ligação

entre os programas desenvolvidos e o ambiente operacional onde eles serão executados. Um programa escrito na linguagem C ou C++, escrito originalmente para um ambiente Windows, precisa ser reescrito e re-compilado caso for necessário executá-lo em um Macintosh. Tais atividades consomem tempo do programador e muito dificultam a tarefa de manutenção de diferentes versões do mesmo programa. Caso haja uma mudança do hardware ou do sistema operacional, os sistemas desenvolvidos em Java existentes não sofrerão nenhuma mudança, no máximo, será necessária a descarga de uma versão do Java

específica para a nova plataforma operacional, contendo a Java Virtual Machine adequada.

4. Java e a programação orientada a objetos:

A programação orientada a objetos é um modelo de programação que se baseia na construção de classes e na criação de objetos destas classes. Estes objetos é que atenderão aos propósitos estabelecidos nas especificações dos programas de computador.

O relacionamento com objetos não é algo estranho para os seres humanos. Ao observar o que existe ao seu redor, o ser humano, normalmente, depara-se com objetos. Objetos possuem atributos e funcionalidades. Um automóvel, por exemplo, possui os atributos cor, potência do motor e o tipo de combustível e as funcionalidades como acelerar, girar o volante, trocar a marcha e frear. Existem vários tipos de automóveis: de corrida; de passeio; de carga. Todos estes tipos compartilham algumas funcionalidades idênticas, como, por exemplo, “frear”. Para quem somente dirige o automóvel o funcionamento interno de sua mecânica não precisa ser conhecido para que ele possa ser utilizado. Mesmo quando um novo modelo de automóvel é lançado por algum fabricante, o ser humano tem a capacidade imediata de identificá-lo, por mais exótico que seja.

Quais são as características comuns entre o nosso automóvel, um navio e uma bicicleta? Podemos afirmar que todos pertencem à categoria “veículos de transporte” e que possuem algumas funcionalidades idênticas, como “acelerar”, e outras particulares como “girar o leme”, no caso do navio.

Vamos analisar agora um ventilador. Através de controles externos podemos identificar as funcionalidades “ligar/ desligar” e “aumentar/reduzir” a velocidade. Mesmo desconhecendo o funcionamento interno do ventilador uma dona-de-casa poderá utilizá-lo plenamente. Um ventilador pode ser enquadrado na categoria “eletrodoméstico”, da mesma forma que uma geladeira. Tanto a geladeira quanto o ventilador possuem, também, algumas funcionalidades idênticas como, por exemplo, “ligar/desligar” e algumas particulares, como, “congelar”, no caso da geladeira.

Estes dois exemplos demonstram os seguintes paradigmas que constituem a programação orientada a objetos:

1 Classificação

- o Consiste em classificar os objetos para que possamos fazer uma leitura melhor do mundo real.

2 Encapsulamento

- o Significa que qualquer objeto pode ser utilizado apenas conhecendo-se a interface que interage com o meio externo. O funcionamento

interno pode ser desconhecido ou ignorado.

3 Herança

- o Através da herança os objetos podem herdar funcionalidades de outros objetos existentes, re-aproveitando algumas e especializando ou criando outras.

4 Polimorfismo

- o Refere-se à capacidade de um objeto possuir um comportamento diferente para uma mesma funcionalidade.

Embora aparentemente complexa esta forma de programar é muito mais intuitiva para os seres humanos, pois é aquela que melhor se aproxima de como estamos acostumados a interagir com os objetos do mundo real.

4.1. Definindo Classes e Objetos:

Uma classe pode ser definida como sendo um gabarito, ou um protótipo, que define um determinado tipo de objeto. As classes especificam as variáveis que irão conter os dados deste objeto e todas as funcionalidades que ele operacionaliza. As variáveis são chamadas de “atributos” do objeto e as funcionalidades são chamadas de “métodos do objeto”.

No exemplo do automóvel, a classe "automóvel" tem a capacidade de criar objetos do tipo automóvel, com os atributos de cor, potência do motor e tipo de combustível e definindo os métodos acelerar, frear, passar a marcha e girar o volante.

No caso do ventilador a classe "ventilador" pode criar objetos do tipo ventilador especificando os atributos número de pás, cor, voltagem e diâmetro e definindo os métodos aumentar velocidade, diminuir velocidade, ligar ventilador e desligar ventilador.

A criação de novos objetos de uma classe chama-se instanciação da classe, portanto podemos definir, dentro dos paradigmas da linguagem orientada a objetos, que um objeto é uma classe instanciada, qual seja, é a classe carregada em memória apta a executar as instruções e comandos inseridos em seus métodos. O objeto dá vida a uma classe, o objeto é a representação concreta de uma classe. A classe "automóvel" é uma representação de um modelo abstrato de um tipo de automóvel enquanto cada automóvel que existe fisicamente é uma instância desta classe.

automóvel
-cor:String -potência:int -combustível:String
+acelerar(valacelera:real) +frear(valfrea:real) +girarVolante(ângulo:real) +passarMarcha(valmarcha:int) +obterMarcha():int

A figura ao lado apresenta um diagrama de classes da classe "automóvel". Nesta representação é utilizada uma linguagem padrão de modelagem, denominada Unified Modeling Language(UML), que trabalha com todas as linguagens orientadas a objetos, inclusive o Java. Nesta figura são apresentados os três atributos e cinco métodos. Aqui, foi incluído mais um método denominado obterMarcha(), essencial para sabermos qual a marcha corrente do nosso automóvel. O sinal menos(-) identifica os atributos e os métodos que estão disponíveis apenas para a classe "automóvel", enquanto o sinal mais(+) indica aqueles atributos e

métodos que podem ser invocados também por outras classes. Na classe "automóvel", deste exemplo, todos os métodos podem ser utilizados por qualquer outra classe enquanto os atributos somente são visíveis para a própria classe. Podemos afirmar então que os atributos desta classe estão encapsulados. Outras classes somente poderão obter acesso a estes atributos através dos métodos da classe "automóvel".

Os métodos operacionalizam todas as funcionalidades oferecidas pelas classes. Os métodos permitem aos objetos exibirem, dinamicamente, os comportamentos especificados pelas classes. Nos métodos os programadores inserem os códigos que implementarão a lógica do negócio. Da mesma forma que os atributos, os métodos também podem ser encapsulados. Um método, com esta característica, somente poderá ser invocado por outros métodos pertencentes a esta mesma classe.

5.Tipos de Dados, Variáveis e Operadores :

5.1.Tipos de Dados:

Associado a cada variável existe um determinado tipo de dado. A linguagem Java suporta dois tipos de dados: o tipo primitivo e o tipo referência. Os tipos primitivos possuem tamanho e formato pré-definidos enquanto os tipos de referência são elementos de dados cujo conteúdo é um ponteiro para um valor ou para um grupo de valores de dados.

A linguagem Java suporta oito tipos primitivos de dados, que são agrupados em quatro categorias distintas: Tipos Inteiros, Tipos Ponto Flutuante, Tipo Caractere e Tipo Lógico. Uma vez que os tipos da linguagem Java são interpretados diretamente pela Java Virtual Machine, eles ficam independentes da plataforma operacional onde serão executados podendo, conseqüentemente, por isto possuir tamanho e formato pré-definidos.

Os Tipos Inteiros são divididos em quatro categorias distintas: *byte*, *short*, *int* e *long*.

Tipo	Ocupa	Valor Mínimo	Valor Máximo
byte	8 bits	-128	+128
short	16 bits	-32.768	+32.768
int	32 bits	-2.147.483.648	+2.147.483.648
long	64 bits	-9.223.372.036.854.775.808	+9.223.372.036.854.775.808

A tabela ao lado apresenta a relação dos tipos inteiros e o espaço, em bits, que cada um deles ocupa em memória.

Os tipos de ponto flutuante são divididos em: *float*

e *double*. O tipo *float* representa os números reais com precisão simples e o tipo *double* representa os números reais com precisão dupla.

Tipo	Ocupa
float	32 bits
double	64 bits

A tabela a lado apresenta os tipos ponto flutuante. Para a facilidade de escrita, expoentes podem ser representados pelas letras “e” ou “E”. Assim, o número 3,580,000,000 pode ser apresentado como 3.58E7 que é igual a 3.58×10^9 .

O tipo caractere é único e é denominado *char*. Este tipo ocupa dois bytes(16 bits) de espaço em memória e tem como propósito suportar o padrão UNICODE, que é um padrão internacional de caracteres que representam os principais tipos de textos do mundo assim como inúmeros símbolos técnicos. Com isto o Java oferece suporte à internacionalização, que é o processo em que os softwares projetados podem ser adaptados para várias línguas e regiões com um mínimo de esforço. A porção do software que cuida da lógica do negócio é isolada daquelas partes que dependem da linguagem, como por exemplo, as de interface com o usuário da aplicação.

O tipo lógico também é único e é denominado *boolean*. Este tipo pode assumir somente dois valores distintos: *false* ou *true*. É utilizado para representar situações que se complementam tais como: verdadeiro ou falso; ligado ou desligado; sim ou não; positivo ou negativo; caro ou barato etc.

5.2.Tipos de Variáveis e Comentários:

5.2.1. Tipos de Variáveis:

Podemos definir uma variável como um nome atribuído pelo programador a uma determinada posição de memória que conterá um dado que representa algum significado dentro da lógica do programa codificado. Este dado poderá possuir qualquer um dos tipos descritos na seção anterior.

Uma variável Java pode ser descrita por uma sequência de caracteres alfanuméricos, podendo iniciar por uma letra ou ainda pelos caracteres “_” ou “\$”. Pode conter até 32 caracteres e não pode conter espaços, símbolos ou qualquer operador. As letras maiúsculas, diferentemente de outras linguagens, não são consideradas iguais às letras minúsculas, pode ser dito, então, que a linguagem Java é *case sensitive*.

Assim como em outras linguagens de programação o Java possui algumas palavras, denominadas *palavras reservadas*, que não podem ser utilizadas como nomes de variáveis ocorrendo um erro de compilação.

abstract	continue	finally	interface	public	throw
boolean	default	float	long	return	throws
break	do	For	native	short	transient
byte	double	If	new	static	true
case	else	Implements	null	super	try
catch	extends	import	package	switch	void
char	false	instanceof	private	synchronized	while
class	final	int	protected	this	

A tabela ao lado apresenta a relação das palavras reservadas do Java.

Nas linguagens que seguem o paradigma da Orientação a Objetos, como é o caso do Java, recomenda-se que a declaração de variáveis siga determinados padrões de codificação que facilitem, somente pela leitura do nome da variável, a identificação de seu propósito.

Assim, a recomendação é que os nomes das variáveis devem ser iniciados com letras minúsculas, e caso o nome seja composto por mais de uma palavra, estas devem ser iniciadas por uma letra maiúscula. Temos, como exemplo de nomes adequados de variáveis: totalSalario; taxaDeJuros; mediaDaTurma; valorTotal.

Em Java declaramos as variáveis obedecendo-se a seguinte sintaxe:

tipo-da-variável variável1 [, variável2 [,variável3 [..., variávelN]]];

Inicialmente é declarado o tipo de dados em que as variáveis se enquadram, e logo a seguir são declarados os respectivos nomes das variáveis, a declaração é encerrada com um “;”, caractere que sempre indica o término de uma atribuição ou de um comando em Java.

As seguintes formas são válidas para declararmos variáveis em Java:

int valorSalario;

byte acumuladorAlunos, acumuladorTurmas;

float valorTemperatura;

boolean saldoNegativo, contaPaga, contaEncerrada;

No momento em que estamos declarando as variáveis de nosso programa também podemos atribuir valores para elas. Podemos codificar então:

```
int valorTotalSalario = 0;
```

```
char minhaLetra = 'A';
```

Aqui a variável **valorTotalSalario** é declarada como sendo do tipo inteiro e é iniciada com o valor zero e a variável **minhaLetra** é declarada como sendo do tipo caractere e é iniciada com o valor *A*. A atribuição simultânea de um mesmo valor para múltiplas variáveis não é permitida pela linguagem. No exemplo abaixo as variáveis **valorTotalSalario** e **valorTotalDesconto** são declaradas como sendo do tipo long mas somente a variável **valorTotalDesconto** é iniciada com o valor zero..

```
long valorTotalSalario, valorTotalDesconto = 0;
```

Podemos declarar as variáveis em qualquer ponto de nosso programa, desde que estas declarações estejam contidas no escopo de atuação das instruções que a elas fazem referências.

5.2.2. Tipos de Comentários:

Os comentários nos programas em Java podem ser inseridos de duas maneiras distintas. A primeira é digitando-se duas barras (//) antes do texto que deve ser tratado como comentário. A segunda utiliza uma dupla de barra asterisco (/*) que permite aumentar a abrangência do comentário já que pode ser utilizado em múltiplas linhas.

1. Exemplo de comentário com "//":

```
// comentário usado em uma única linha
```

```
a=b; // também pode ser usado assim
```

2. Exemplo de comentários com "/*":

```
/* Aqui podemos utilizar
```

```
múltiplas linhas
```

```
de comentários */
```

5.3. Tipos de Operadores:

A linguagem Java suporta seis tipos distintos de operadores; operadores aritméticos, operadores relacionais, operadores condicionais, operadores lógicos e de deslocamento e operadores de atribuição.

5.3.1. Operadores Aritméticos:

Os operadores aritméticos são aqueles utilizados para realizar as operações aritméticas sobre os tipos de dados inteiros e flutuantes. A figura abaixo apresenta os casos de utilização destes operadores onde op1 e op2 são os campos que representam os operandos.

<i>Operador</i>	<i>Como utilizar?</i>	<i>Como funciona?</i>
+	op1 + op2	Soma op1 a op2
-	op1 - op2	Subtrai op2 de op1
*	op1 * op2	Multiplica op1 por op2
/	op1 / op2	Divide op1 por op2
%	op1 % op2	Calcula o resto da divisão de op1 por op2
+	+op1	Transforma op em tipo <i>int</i> se ele for tipo <i>byte</i> , <i>short</i> ou <i>char</i> .
-	-op1	Nega, aritmeticamente, op
++	++op1 ou op1++	Incrementa em uma unidade op1.
--	--op1 ou op1--	Decrementa em uma unidade op1.

Cumpra observar que o operador (+) também pode ser utilizado para concatenar strings.

Quando um número inteiro e um número de ponto flutuante são utilizados como operandos de uma mesma operação, o resultado será um número de ponto flutuante.

Na operação de divisão envolvendo dois números inteiros o quociente também será um número inteiro, qual seja, se houver uma parte fracionária ela será truncada.

Dois operadores aritméticos considerados operadores de atalho são

disponibilizados em Java: “++” que incrementa o operando em uma unidade e o “- -” que decrementa o operando também em uma unidade. Tanto o “++” quanto o “- -” pode aparecer antes ou depois do operando e a diferença essencial em colocá-los antes ou depois é que:

- Colocando **antes**, a comparação do valor será feita **após** a operação de incremento ou decremento da unidade.
- Colocando **depois**, a comparação de valor será feita **antes** da operação de incremento ou decremento da unidade.

Exemplos:

totalAlunos++; // tem o mesmo efeito que **totalAlunos = totalAlunos + 1;**

++totalAlunos; // tem o mesmo efeito que **totalAlunos = totalAlunos + 1;**

totalAlunos--; // tem o mesmo efeito que **totalAlunos = totalAlunos - 1;**

--totalAlunos; // tem o mesmo efeito que **totalAlunos = totalAlunos - 1;**

5.3.2. Operadores Relacionais:

Os operadores relacionais comparam os valores de dois operandos retornando um valor booleano indicando se o resultado desta comparação é verdadeiro(*true*) ou falso(*false*). São utilizados para formar expressões do tipo “se *x* for maior ou igual a *y* então faça”.

<i>Operador</i>	<i>Como utilizar?</i>	<i>Como funciona?</i>
<i>></i>	<i>op1 > op2</i>	se op1 for maior do que op2 retorna <i>true</i>
<i>>=</i>	<i>op1 >= op2</i>	se op1 for maior ou igual a op2 retorna <i>true</i> .
<i><</i>	<i>op1 < op2</i>	se op1 for menor do que op2 retorna <i>true</i>
<i><=</i>	<i>op1 <= op2</i>	se op1 for menor ou igual do que op2 retorna <i>true</i> .
<i>==</i>	<i>op1 == op2</i>	se op1 for igual a op2 retorna <i>true</i> .
<i>!=</i>	<i>op1 != op2</i>	se op1 não for igual a op2 retorna <i>true</i> .

A figura ao lado descreve o funcionamento dos operadores relacionais.

Podemos escrever “se *x* == *y* faça” para testar uma condição de igualdade ou escrever “se *x* != *y* faça” para testar uma condição de desigualdade.

5.3.3. Operadores Condicionais:

O Java suporta seis diferentes operadores condicionais, cinco são binários e um é unário. Estes operadores são normalmente utilizados com os operadores relacionais para formar expressões do tipo “se *x* for maior do que *y* E se *k* for maior do que *w* então faça”. Também retorna um valor booleano indicando se o resultado desta comparação é verdadeiro(*true*) ou falso(*false*).

Operador	Como Utilizar?	Como Funciona?
&&	op1 && op2	se ocorrer op1 e se ocorrer op2 retorna <i>true</i>
 	op1 op2	se ocorrer op1 ou se ocorrer op2 retorna <i>true</i> .
!	!op1	se não ocorrer op1 retorna <i>true</i> .
&	op1 & op2	se op1 e op2 forem variáveis booleanas e ambas forem <i>true</i> retorna <i>true</i> .
 	op1 op2	se op1 e op2 forem variáveis booleanas e uma ou outra for <i>true</i> retorna <i>true</i> .
^	op1 ^ op2	retorna <i>true</i> se ambos operadores não são <i>true</i> . É um OR EXCLUSIVE.

A Java Virtual Machine, através de seu interpretador, somente analisa a condição do segundo operando quando a condição do primeiro operando é satisfeita, então no caso de “se *x* for maior do que 5 e se *y* for maior que 10” o valor de *y* somente será comparado se o valor de *x* for obrigatoriamente maior do que 5, com isto economiza-se tempo de computação, pois não será feita uma comparação que, de antemão, já é sabido que não será satisfeita.

5.3.4. Operadores Lógicos e de Deslocamento:

Os operadores lógicos e de deslocamento realizam operações de manipulação de bits sobre os operandos. São tratados os bits, um a um, daqueles operandos envolvidos nas operações.

Os operadores lógicos realizam as operações especificadas pela lógica booleana para os conectivos AND, OR, COMPLEMENTO e OR EXCLUSIVE. A figura abaixo apresenta o funcionamento destes operadores.

Operador	Como Utilizar?	Como Funciona?
&	op1 & op2	se os operadores forem números é feito um AND booleano, se forem booleanos é feito um AND condicional.
 	op1 op2	se os operadores forem números é feito um OR booleano, se forem booleanos é feito um OR condicional.
^	op1 ^ op2	é realizado um OR EXCLUSIVE entre os operandos.
~	~op1	o operando é complementado binariamente.

Como exemplo vamos realizar uma operação “numeroA & numeroB”. O numeroA é uma variável do tipo

inteiro cujo conteúdo é 18. O numeroB também é uma variável do tipo inteiro e seu conteúdo é 20. A configuração binária de numeroA é “00010010” e a configuração binária de numeroB é “00010100”. Ao aplicarmos o conectivo lógico AND, representado pelo símbolo “&”, teremos o seguinte resultado “00010000” que representa o número inteiro 16.

Realizando agora a operação “numeroA ^ numeroB”, que implementa o conectivo lógico OR EXCLUSIVE, teremos o resultado “00000110”, que representa o número inteiro 6. E se realizarmos a operação “~numeroA” teremos o resultado “11101101” que representa o número inteiro 237.

Os operadores de deslocamento realizam operações que deslocam os bits do primeiro operando uma quantidade de posições determinada pelo conteúdo do segundo operando. A tabela abaixo apresenta o seu funcionamento.

<i>Operador</i>	<i>Como Utilizar?</i>	<i>Como Funciona?</i>
<<	op1 << op2	desloca para a esquerda os bits de op1 pela quantidade especificada em op2. A quantidade remanescente é preenchida com zeros.
>>	op1 >> op2	desloca para a direita os bits de op1 pela quantidade especificada em op2. Completa com o bit de sinal.
>>>	op1 >>> op2	desloca para a esquerda os bits de op1 pela quantidade especificada em op2. A quantidade remanescente é preenchida com zeros.

Como exemplo vamos realizar uma operação “numeroA << 2”. O numeroA é uma variável do tipo inteiro cujo conteúdo é 15. O segundo operando é representado pelo número 2, portanto, o numeroA deve ter seu conteúdo deslocado 2 bits

para a esquerda. A configuração binária de numeroA é “00001111”, então, após o deslocamento, o conteúdo de numeroA será “00111100” que representa o número inteiro 60. Podemos afirmar que cada 1 bit deslocado para a esquerda implica em multiplicar o valor do primeiro operando por 2. Da mesma forma, cada bit deslocado para a direita implica em ter o valor do primeiro operando dividido por 2.

5.3.5. Operadores de Atribuição:

Os operadores de atribuição são utilizados para atribuir valores de dados àquelas variáveis que foram definidas em nosso programa, para tal é utilizado o símbolo padrão de igualdade “=”.

Para atribuir, por exemplo, o valor inteiro 1954 para uma variável de nome valorTotal escrevemos:

valorTotal = 1954;

Se quisermos incrementar a variável `valorTotal` de 10 unidades podemos escrever:

```
valorTotal = valorTotal + 10;
```

Os operadores de atribuição também suportam atalhos que permitem uma escrita mais sucinta e menos sujeita a equívocos. A mesma atribuição do exemplo anterior poderia ser codificada da seguinte forma:

```
valorTotal += 10;
```

A tabela abaixo apresenta os atalhos para os comandos de atribuição que envolvem os operadores aritméticos.

<i>Atalho</i>	<i>Como Utilizar?</i>	<i>Equivale a que?</i>
<code>+=</code>	<code>op1 += op2</code>	<code>op1 = op1 + op2</code>
<code>-=</code>	<code>op1 -= op2</code>	<code>op1 = op1 - op2</code>
<code>*=</code>	<code>op1 *= op2</code>	<code>op1 = op1 * op2</code>
<code>/=</code>	<code>op1 /= op2</code>	<code>op1 = op1 / op2</code>
<code>%=</code>	<code>op1 %= op2</code>	<code>op1 = op1 % op2</code>

Esta forma de atalho também se aplica aos operadores lógicos e de deslocamento, cumprindo-se a mesma estrutura sintática e semântica da linguagem Java.

6. Definindo Constantes:

Uma constante é utilizada para definir um dado que não pode ser alterado durante a execução do programa. Em Java declaramos os valores constantes obedecendo a seguinte sintaxe:

```
final tipo-do-dado nome-da-constante = valor-da-constante;
```

A palavra-chave ***final*** é de codificação obrigatória e serve para indicar que está sendo especificado um valor constante. É declarado o tipo de dado da constante, o nome em que a constante será referenciada no programa e, por último, é informado o valor da constante. Qualquer tentativa de alteração do valor da constante provocará um erro em tempo de compilação. Abaixo, temos alguns exemplos de definição de constantes:

```
final float taxaDeJuros = 1.12;
```

```
final int diasEmSetembro = 30;
```

```
final char minhaNomeEmpresa = 'F';
```

7. Escrevendo as primeiras classes:

As classes que serão apresentadas a seguir têm como objetivo ilustrar os passos necessários para o desenvolvimento e execução de aplicativos escritos na linguagem Java. São todas de fácil entendimento já que não possuem nenhuma complexidade, e apenas estarão colocando em prática aquilo que já foi visto nas seções anteriores deste material.

A primeira classe, denominada AloMundo, simplesmente apresentará na tela de *prompt* a frase “Alo Mundo, estou aprendendo Java !”.

Para digitar o código desta classe pode ser utilizado qualquer editor de textos padrão devendo-se observar que o arquivo gerado deve ser do tipo “texto simples” e deve ter a extensão **.java**.

A classe AloMundo possui o seguinte código:

```
public class AloMundo {  
  
    public static void main(String[] args) {  
  
        System.out.println("Alo Mundo, estou aprendendo Java !");  
  
    }  
}
```

Após a classe AloMundo ser digitada e salva ela precisa ser compilada. Supondo que a classe foi salva em um diretório denominado “CursoJava”, residente em um disco rígido identificado como “C”, o seguinte comando deve ser emitido na tela de prompt para compilar a classe AloMundo:

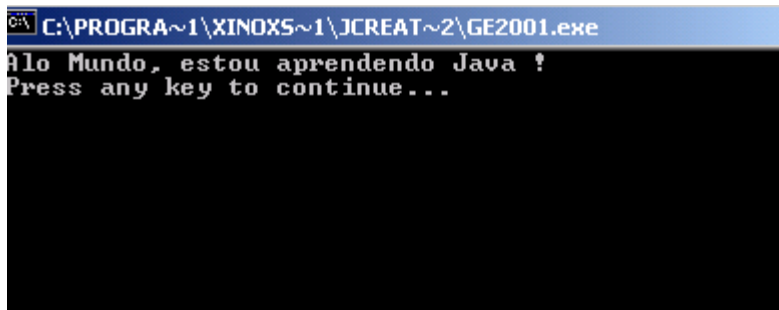
C:\CursoJava>javac AloMundo.java

É fundamental que os arquivos Java estejam sendo declarados no CLASSPATH do sistema operacional envolvido no processamento, pois caso contrário haverá uma mensagem de erro informando que o comando é inválido ou o arquivo não foi encontrado.

A palavra ***javac*** identifica o nome do compilador da linguagem Java. Após a compilação da classe AloMundo, caso não existam erros, será gerado o arquivo de nome AloMundo.class, este arquivo contém os *bytecodes* necessários para a Java Virtual Machine executar a classe, que é o nosso próximo passo. Então o seguinte comando deve ser emitido também na tela de prompt:

C:\CursoJava>java AloMundo

A palavra ***java*** identifica o nome da Java Virtual Machine. Observe que para a execução da classe AloMundo a codificação da extensão **.class não é necessária**.

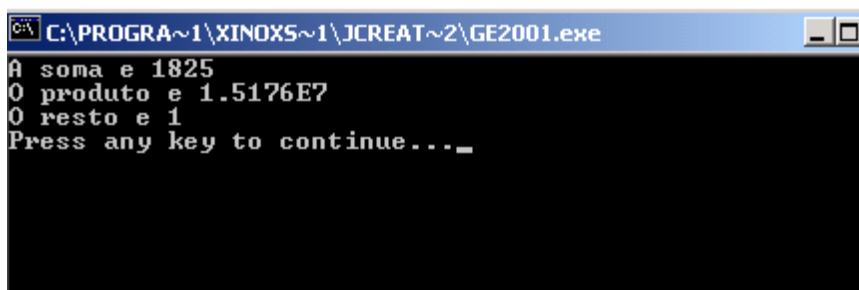


A figura ao lado apresenta o resultado fornecido após a execução da classe AloMundo.

A segunda classe, denominada RealizaContas, exercita algumas das operações aritméticas já anteriormente explicadas. São realizadas operações de somar três números inteiros (tipo *int*), multiplicar dois números de ponto flutuante (tipo *float*) e obter o resto da divisão de números inteiros (tipo *byte*).

A classe RealizaContas possui o seguinte código:

```
public class RealizaContas {  
    public static void main(String[] args) {  
        int parcela1 = 520;  
        int parcela2 = 380;  
        int parcela3 = 925;  
        float multiplicando = 3500;  
        float multiplicador = 4336;  
        byte dividendo = 13;  
        byte divisor = 3;  
        System.out.println("A soma e " + (parcela1+parcela2+parcela3));  
        System.out.println("O produto e " + (multiplicando * multiplicador));  
        System.out.println("O resto e " + (dividendo % divisor));  
    }  
}
```



Digitando, compilando e executando o código da classe RealizaContas

será apresentada a tela de prompt da figura ao lado.

A primeira linha de código das classes anteriores possui o seguinte formato:

```
public class AloMundo {
```

A palavra-chave **public** indica que a classe em questão é do tipo pública e que, portanto, pode ser instanciada por qualquer classe de qualquer aplicativo. A palavra-chave **class** é obrigatória e indica que está sendo escrito o código de uma classe. A palavra **AloMundo** indica o nome que estamos atribuindo a nossa classe. Cumpre observar que o nome do arquivo de extensão `.java`, que abrigará a classe, precisa chamar-se `AloMundo.java`, caso contrário a execução da classe não ocorrerá, pois a Java Virtual Machine irá verificar se o nome do arquivo e nome da classe são idênticos.

A linha seguinte identifica o primeiro e único método que integra a classe.

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

A palavra-chave **public** indica que o método pode ser invocado por qualquer outra classe ou método de qualquer aplicativo. A palavra-chave **static** indica que este método pode ser invocado sem que um objeto da classe que o abriga tenha sido criado. A palavra-chave **void** indica que o método invocado não retornará nenhum valor. As especificações entre parênteses, **String[] args**, indicam os argumentos que poderão ser passados para o método, sua utilização será vista posteriormente.

Como já foi visto em seções anteriores uma classe é constituída por atributos e métodos. Os métodos realizam as operações que implantarão a lógica determinada nos algoritmos elaborados pelos programadores. Os métodos podem ser comparados às tradicionais funções existentes nas inúmeras linguagens de programação.

Na linguagem Java o método *main* possui uma característica particular: quando a Java Virtual Machine executa uma classe (comando **java nome-da-classe**), conforme visto em exemplos anteriores, o método *main* será, obrigatoriamente, o primeiro método a ser processado pela Java Virtual Machine. Portanto, podemos afirmar que toda aplicação Java do tipo *desktop* possui um método *main*. Caso um aplicativo possua inúmeras classes, pelo menos uma delas deverá conter um método do tipo *main*. O método *main* é o método considerado como ponto-de-partida para a execução de um aplicativo Java do tipo *desktop*. As outras classes do aplicativo serão instanciadas no método *main*.

A próxima linha é utilizada para a apresentação de dados no *desktop*.

```
System.out.println(colocamos-aqui-o-que-queremos-apresentar);
```

A palavra chave **System.out** é um dos objetos utilizados para imprimir dados em um *desktop*. Este objeto possui os métodos **print**, **println** e **printf**.

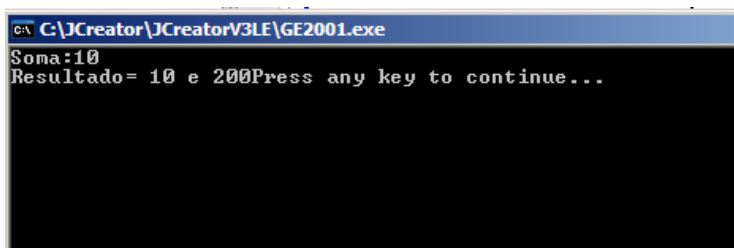
<i>Método</i>	<i>Descrição</i>
<code>println(dados)</code>	Imprime os dados e salta para a próxima linha no desktop.
<code>print(dados)</code>	Imprime os dados no desktop sem saltar de linha.

A tabela ao lado apresenta a descrição dos métodos **print** e **println**. Os dados que são passados como argumento podem ser agrupados utilizando-se o operador

de concatenação “+”. Neste caso, mesmo dados de tipos distintos podem ser concatenados.

Exemplos:

```
int valorSoma=10;
int valorSalarioTotal=200;
System.out.println("Soma:" + valorSoma);
System.out.print("Resultado= " + valorSoma + " e " + valorSalarioTotal);
```



A figura ao lado apresenta o resultado da execução do exemplo acima. Como pode ser observado o método **println** provocou o salto para a linha seguinte enquanto o método **print** não comandou o salto.

Caso exista no conjunto de dados que serão impressos uma barra invertida (\) o interpretador do Java entenderá que este caractere é um “caractere de escape”, e que associando este caractere a outros caracteres específicos, forma-se um par denominado “**seqüência de escape**”. Através da seqüência de escape as ações de impressão dos dados podem ser alteradas.

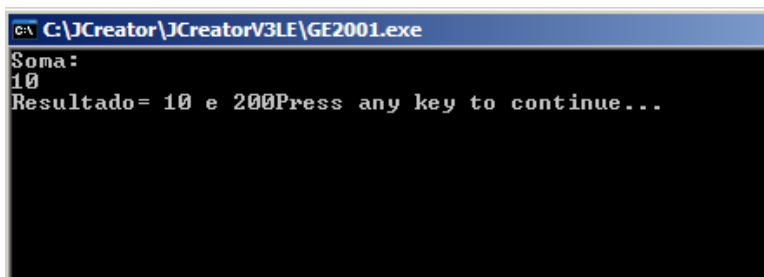
<i>Seqüência de Escape</i>	<i>Descrição</i>
<code>\n</code>	Salta para a próxima linha do <i>desktop</i> .
<code>\t</code>	Posiciona na próxima parada de tabulação.
<code>\r</code>	Retorna para início da linha do <i>desktop</i> .
<code>\\</code>	Utilizada para imprimir uma barra invertida, inibindo a ação de escape.
<code>\"</code>	Utilizada para imprimir um caractere com aspas duplas.

A figura ao lado apresenta algumas das seqüências de escape mais usuais.

Introduzindo no texto do programa do exemplo uma seqüência de escape “\n” será apresentado o resultado da figura abaixo.

Exemplo de utilização da seqüência de escape \n:

```
System.out.println("Soma:\n" + valorSoma);
```



```

C:\JCreator\JCreatorV3LE\GE2001.exe
Soma:
10
Resultado= 10 e 200Press any key to continue...

```

Como pode ser observado o resultado de Soma é apresentado na linha seguinte.

Uma outra maneira de apresentação de informações no *desktop* foi introduzida na recente versão 1.5 do Java, acrescentou-se ao objeto **System.out** um novo método denominado **printf**. Com este método podem ser feitas formatações nos dados apresentados, melhorando suas visualizações, o que nos métodos **print** e **println** não era possível. Este método possui a seguinte sintaxe:

System.out.printf (|indicadores-do-formato, |lista-de-variáveis)

Nos indicadores-do-formato são descritos os formatos das variáveis que serão apresentadas e seus escapes de sequência, se forem necessários. As variáveis, e nelas ficam incluídos os textos, complementam o comando.

Cada indicador de formato deve ser precedido pelo caractere “%” que servirá também como um delimitador para cada formato especificado. A tabela ao lado contém alguns dos formatos suportados pelo método **printf**.

<i>Indicador do Formato</i>	<i>Descrição</i>
d	Apresenta um número inteiro.
X ou x	Apresenta um número inteiro no formato hexadecimal.
E ou e	Apresenta um número ponto flutuante no formato exponencial.
f	Apresenta um número flutuante no formato decimal.
s	Apresenta o dado no formato string.
B ou b	Apresenta “true” ou “false” se o dado for um booleano.

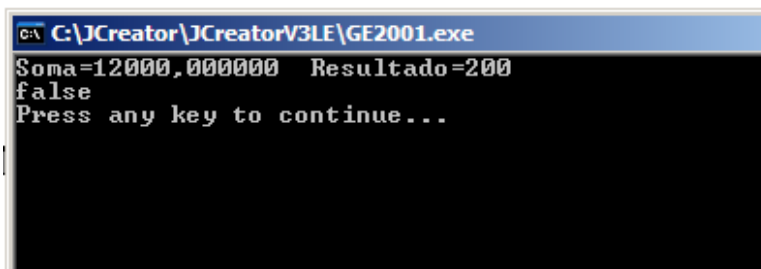
No código do exemplo abaixo é utilizado o método **printf** para formatar os dados de impressão.

```

public static void main(String[] args) {
    double valorSoma=1.2E4;
    int valorSalario=200;
    boolean ligado = false;

    System.out.printf("%s%f%s%d\n", "Soma=", valorSoma, " Resultado=", valorSalario );
    System.out.printf("%b\n", ligado);
}

```



A figura ao lado apresenta o resultado da execução do programa-exemplo.

8. Lendo dados do teclado ou de arquivos:

Também na versão 1.5 do Java foi disponibilizada uma classe, denominada **Scanner**, que tem como um de seus propósitos realizar a leitura de dados provenientes de várias origens distintas. Podem ser: dados digitados em teclado por algum usuário; dados contidos em uma string; ou dados de arquivos armazenados em discos magnéticos. A classe **Scanner** oferece um vasto elenco de métodos e nesta seção será tratado o método **next()**.

Para utilizar a classe **Scanner** para ler dados os seguintes passos devem ser seguidos:

1. Instanciar a classe **Scanner** especificando a origem dos dados a serem lidos.
2. Invocar o método **next()**, especificando o tipo do dado a ser lido.

Abaixo é apresentado o código completo da classe **NumeroTeste** que realizará a leitura de dados que serão digitados em um teclado.

```
import java.util.Scanner;

public class NumeroTeste{

    public static void main(String[] args) {

        int i1, i2;
        Scanner entrada = new Scanner (System.in);
        System.out.printf("Digite o primeiro numero: \t");
        i1 = entrada.nextInt();
        System.out.printf("Digite o segundo numero: \t");
        i2 = entrada.nextInt();
        System.out.printf("%s%d\n", "A soma: ", i1 + i2);
    }
}
```

As inúmeras classes que integram a plataforma J2SE são divididas em pacotes de acordo com seus propósitos e finalidades. Para evitar a carga de classes que não serão utilizadas pelos aplicativos, somente os pacotes que contém aquelas classes que executam funções consideradas básicas são carregadas sem a intervenção explícita do programador.

Outras classes, se utilizadas, precisam ser explicitamente carregadas.

A classe **Scanner** enquadra-se naquele conjunto de classes que precisam ser carregadas pelo programador. O pacote `Java.util.Scanner` contém a classe **Scanner** e, portanto, deve ser o pacote carregado. A carga de qualquer pacote é feita através do comando **import**, cuja sintaxe é apresentada abaixo.

```
import nome-do-pacote[.nome-da-classe];
```

O comando **import** deve preceder qualquer declaração realizada no código de uma classe.

A classe **Scanner** precisa ser instanciada, e para tal foi codificado na classe `NumeroTeste` o seguinte comando:

```
Scanner entrada = new Scanner(System.in);
```

Para realizar a instanciação de qualquer classe é necessário que seja declarado o nome da classe de interesse e o nome do objeto que realizará as funcionalidades da classe, neste caso o nome do objeto é *entrada*.

A palavra-chave **new** é a indicação para a Java Virtual Machine que uma classe deve ser instanciada.

No caso da classe **Scanner** é necessário que seja identificado o local da origem dos dados que serão lidos. No exemplo, a especificação **System.in** indica que serão lidos os dados digitados no teclado.

Para que os dados sejam lidos é necessário invocar o método **next()** da classe **Scanner**. Este método possui diversas formas de uso, uma para cada tipo de dado que esteja sendo lido.

<i>Formas do next()</i>	<i>Descrição</i>
<code>nextInt()</code>	O dado a ser lido é do tipo <code>int</code> .
<code>nextLong()</code>	O dado a ser lido é do tipo <code>long</code> .
<code>nextDouble()</code>	O dado a ser lido é do tipo <code>double</code> .
<code>next()</code>	O dado lido é do tipo <code>string</code> .

Cada forma do método **next()** lê somente um dado, portanto, a leitura de vários dados implica em emitir vários comandos **next()** consecutivos, um para cada dado existente.

9. Manipulando strings:

Podemos definir uma string como sendo um tipo de variável que tem a capacidade de abrigar qualquer tipo de caractere incluindo números, letras e símbolos. Diferentemente

de outras linguagens o Java não possui o tipo *string*, como pôde ser observado nas seções em que foram especificados os tipos suportados pela linguagem.

Para a manipulação de *strings* em Java é disponibilizada uma classe **String**. Esta classe possui um amplo conjunto de métodos que realizam operações do tipo: comparação de *strings*, verificação do tamanho de uma *string*, informação da posição de determinado caractere em uma *string*, obtenção de um pedaço da *string* etc. Ao longo deste material serão vistos inúmeros casos em que estes métodos serão aplicados na resolução de algum tipo de problema.

Para ser utilizada é necessário que a classe **String** seja previamente instanciada. Esta instanciação pode ser feita das seguintes maneiras:

```
String nomeDoLivro = new String();   ou  
String nomeDoLivro;
```

Em ambos os casos o efeito produzido é o mesmo: está sendo instanciada uma classe **String** cujo objeto terá o nome **nomeDoLivro**. Durante a instanciação do objeto ele pode receber um valor inicial. Para tal, as seguintes formas são válidas:

```
String nomeDoLivro = new String("Java como Programar");   ou  
String nomeDoLivro = "Java como Programar";
```

Assim, o objeto **nomeDoLivro** será iniciado com o valor "Java como Programar".

A classe **String** disponibiliza inúmeros métodos, nesta seção serão detalhados os métodos `length()`, `charAt()`, `equals()`, `equalsIgnoreCase`, `compareTo`, `regionMatches()`, `indexOf()`, `lastIndexOf`, `startsWith()`, `endsWith()`, `substring()`, `concat()`, `trim`, `toLowerCase` e `toUpperCase`.

A tabela abaixo descreve a atuação de cada um destes métodos.

<i>Método</i>	<i>Como funciona?</i>
<code>charAt()</code>	Retorna qual caractere encontra-se em uma posição específica da <i>string</i> .
<code>concat()</code>	Concatena duas <i>strings</i> .
<code>endsWith()</code>	Retorna <code>true</code> se a <i>string</i> termina com determinado sufixo.
<code>equals()</code>	Retorna <code>true</code> se as duas <i>strings</i> comparadas têm o mesmo tamanho e são exatamente iguais.
<code>equalsIgnoreCase()</code>	Retorna <code>true</code> se as duas <i>strings</i> comparadas têm o mesmo tamanho e são exatamente iguais. Será ignorada a diferença entre maiúsculas e minúsculas.

<code>indexOf()</code>	Retorna um número inteiro informando a primeira posição em que se encontra um caractere ou uma substring em uma string.
<code>length()</code>	Retorna o tamanho da string. São considerados os espaços em branco no início e no fim da string, se existirem.
<code>lastIndexOf()</code>	Retorna um número inteiro informando a última posição em que se encontra um caractere ou uma substring em uma string.
<code>regionMatches()</code>	Retorna true se determinada região de uma string é igual a uma determinada região de outra string.
<code>replace()</code>	Substitui todos os caracteres de uma string por outro caractere.
<code>startsWith()</code>	Retorna true se a string inicia com determinado prefixo.
<code>substring()</code>	Copia para outra string uma porção da string a partir de determinada posição.
<code>toLowerCase()</code>	Retorna uma nova string com cada caractere sendo convertido para o correspondente minúsculo.
<code>toUpperCase()</code>	Retorna uma nova string com cada caractere sendo convertido para o correspondente maiúsculo.
<code>trim()</code>	Retorna uma string sem espaços em seu início e em seu fim.

Nos exemplos que seguem são ilustrados alguns casos de utilização destes métodos.

Exemplo 01

Informar qual a primeira letra de um determinado nome obtido pelo teclado:

```
import java.util.Scanner;

public class PrimeiraLetra{

    public static void main(String[] args) {

        String nome;

        Scanner entrada = new Scanner(System.in);

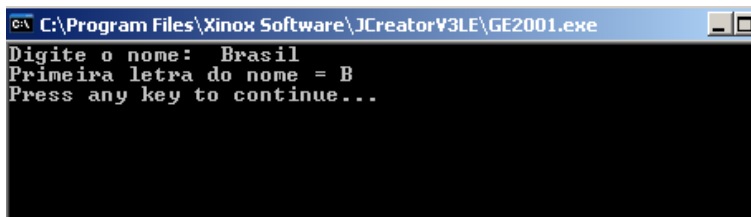
        System.out.printf("Digite o nome:\t");

        nome = entrada.next();

        System.out.printf("%s%s\n", "Primeira letra do nome = ", nome.charAt(0));

    }

}
```

A figura ao lado apresenta o resultado da execução do aplicativo. É muito importante ser observado que para a posição de partida de qualquer string é sempre considerada a posição

zero e não a posição um, daí ter sido especificado `nome.charAt(0)`. Se tivéssemos que capturar a segunda letra do nome, teria que ser escrito `nome.charAt(1)` e não `nome.charAt(2)`.

Exemplo 02

Concatenar duas strings distintas obtidas pelo teclado:

```
import java.util.Scanner;
```

```
public class ConcatenaPalavras{
```

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

```
        String nome,sobrenome;
```

```
        Scanner entrada = new Scanner(System.in);
```

```
        System.out.printf("Digite o nome:\t");
```

```
        nome = entrada.next();
```

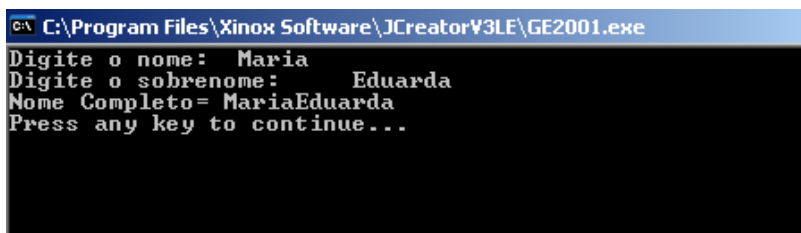
```
        System.out.printf("Digite o sobrenome:\t");
```

```
        sobrenome = entrada.next();
```

```
        System.out.printf("%s%s\n", "Nome Completo= ", nome.concat(sobrenome));
```

```
    }
```

```
}
```



A figura ao lado apresenta o resultado da execução do método `concat()`. Como já foi visto em seções anteriores também pode ser utilizado

o operador “+” para ser feita uma concatenação.

Exemplo 03

Informar, pesquisando a string da esquerda para a direita, em qual posição

encontra-se uma outra string. Para este exemplo é empregado o método `indexOf()`. Este método pode receber dois argumentos: o primeiro argumento informa o pedaço da string que se quer procurar; o segundo argumento, que é opcional, informa a posição inicial de pesquisa, caso seja omitido será assumida a posição zero.

```
import java.util.Scanner;

public class PesquisaString{

    public static void main(String[] args) {

        String nome,oQueProcurar;

        Scanner entrada = new Scanner(System.in);

        System.out.printf("Digite o nome a ser pesquisado:\t");

        nome = entrada.next();

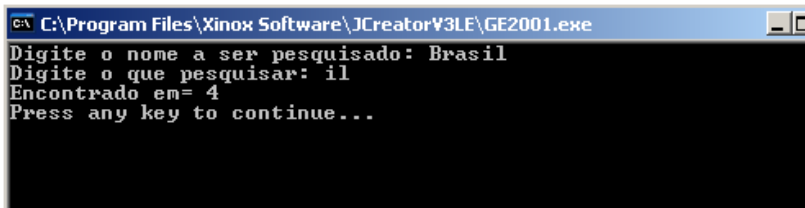
        System.out.printf("Digite o que quer pesquisar:\t");

        oQueProcurar = entrada.next();

        System.out.printf("%s%s\n", "Tamanho do nome= ", nome.indexOf(oQueProcurar));

    }

}
```



A figura ao lado apresenta o resultado da execução do método `indexOf()`. Deve ser observado que como a posição de partida é zero a

string “il” ocorre na quarta posição da string “Brasil”.

Exemplo 04

Informar qual o tamanho de determinada string obtida pelo teclado:

```
import java.util.Scanner;

public class VerificaTamanho{

    public static void main(String[] args) {

        String nome;

        Scanner entrada = new Scanner(System.in);

        System.out.printf("Digite o nome : \t");
```

```

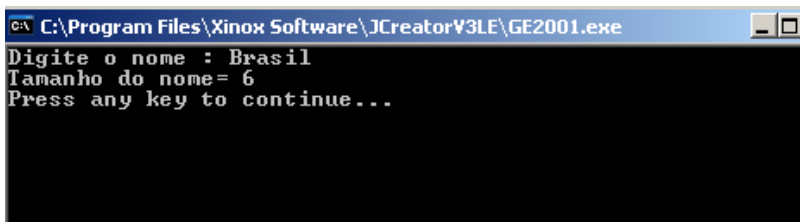
        nome = entrada.next();

        System.out.printf("%s%s\n", "Tamanho do nome= ", nome.length());

    }

}

```



A figura ao lado apresenta o resultado da execução do método `length()`.

Exemplo 05

Informar, pesquisando a string da direita para a esquerda, em qual posição encontra-se uma outra string. Para este exemplo é empregado o método `lastIndexOf()`. Este método pode receber dois argumentos: o primeiro argumento informa o pedaço da string que se quer procurar; o segundo argumento, que é opcional, informa a posição inicial de pesquisa, caso seja omitido será assumida a última posição.

```

import java.util.Scanner;

public class PesquisaString{

    public static void main(String[] args) {

        String nome,oQueProcurar;

        Scanner entrada = new Scanner(System.in);

        System.out.printf("Digite o nome a ser pesquisado:\t");

        nome = entrada.next();

        System.out.printf("Digite o que quer pesquisar:\t");

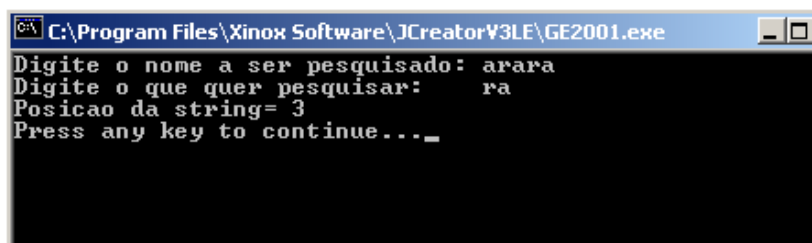
        oQueProcurar = entrada.next();

        System.out.printf("%s%s\n", "Posicao da string= ", nome.lastIndexOf(oQueProcurar));

    }

}

```



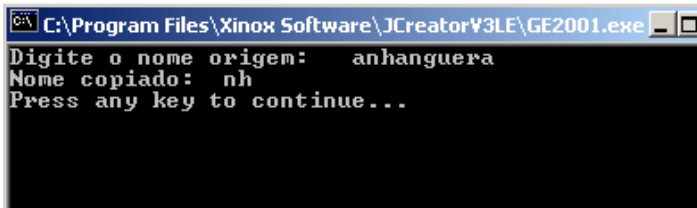
A figura ao lado apresenta o resultado da pesquisa. Deve ser observado que é indicada

a posição da última string “ra”.

Exemplo 06

Copiar uma porção de uma string para uma outra string. Neste exemplo será utilizado o método `substring()`. Este método pode receber dois argumentos: o primeiro argumento informa a posição inicial do pedaço da string que se quer copiar; o segundo argumento informa a posição final desta string adicionada de uma unidade. Caso o segundo argumento seja omitido será copiado até o final da string. `import java.util.Scanner;`

```
public class CopiaString{  
  
    public static void main(String[] args) {  
  
        String origem, destino;  
  
        Scanner entrada = new Scanner(System.in);  
  
        System.out.printf("Digite o nome origem:\t");  
  
        origem = entrada.next();  
  
        destino = origem.substring(1,3);  
  
        System.out.printf("%s%s\n", "Nome copiado: ", destino);  
  
    }  
  
}
```



A figura ao lado apresenta o resultado da execução. Neste exemplo está sendo copiada da posição um até a segunda posição, totalizando 2 caracteres.

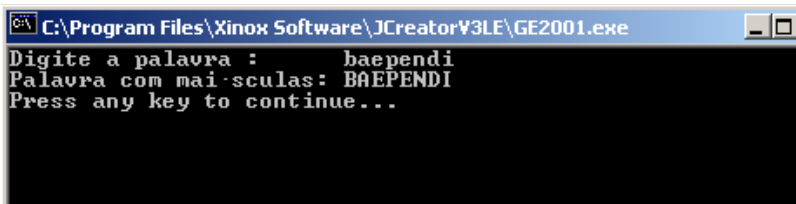
Exemplo 07

Converter todo o conteúdo de uma string para caracteres maiúsculos.

```
import java.util.Scanner;  
  
public class ConverteParaMaiusculo{  
  
    public static void main(String[] args) {  
  
        String palavra;  
  
        Scanner entrada = new Scanner(System.in);  
  
        System.out.printf("Digite a palavra :\t");  
  
        palavra = entrada.next();
```

```
System.out.printf("%s%s\n", "Palavra com maiúsculas: ", palavra.toUpperCase());
```

```
}
```



```
}
```

Exemplo 08

Converter todo o conteúdo de uma string para caracteres minúsculos.

```
import java.util.Scanner;
```

```
public class ConverteParaMinusculo{
```

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

```
        String palavra;
```

```
        Scanner entrada = new Scanner(System.in);
```

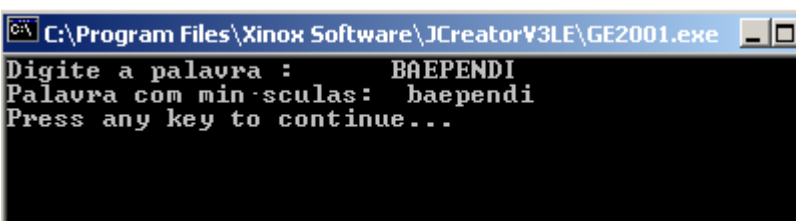
```
        System.out.printf("Digite a palavra :\t");
```

```
        palavra = entrada.next();
```

```
        System.out.printf("%s%s\n", "Palavra com minúsculas: ", palavra.toLowerCase());
```

```
    }
```

```
}
```



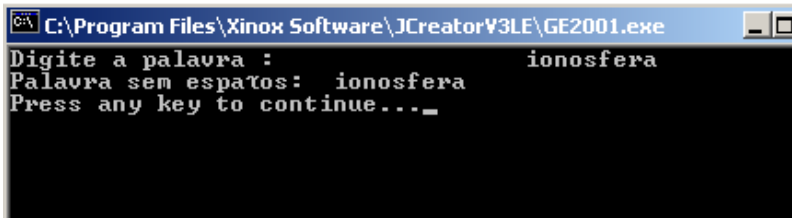
Exemplo 09

Suprimir todos os caracteres em branco que iniciam ou encerram uma string.

```
import java.util.Scanner;
```

```
public class SuprimeBranco{
```

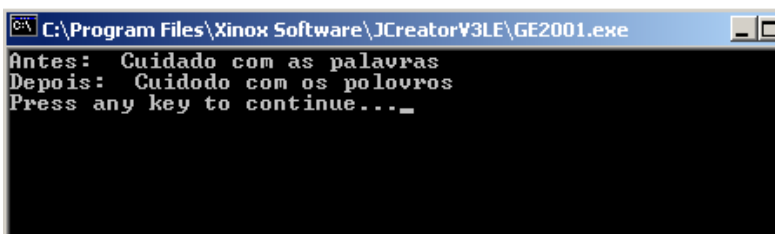
```
public static void main(String[] args) {  
    String palavra;  
    Scanner entrada = new Scanner(System.in);  
    System.out.printf("Digite a palavra :\\t");  
    palavra = entrada.next();  
    System.out.printf("%s%s\\n", "Palavra sem espaços: ", palavra.trim());  
}  
}
```



Exemplo 10

Substituir todos os caracteres existentes em uma string por outro.

```
import java.util.Scanner;  
  
public class SubstituiLetra{  
    public static void main(String[] args) {  
        String palavra = "Cuidado com as palavras";  
        System.out.printf("%s%s\\n", "Antes: ", palavra);  
        System.out.printf("%s%s\\n", "Depois: ", palavra.replace("a","o"));  
    }  
}
```



Neste exemplo, todos os caracteres “a” encontrados na string são substituídos pelo caractere “o”.

4. Estrutura de Desvio de Fluxo ou de Seleção:

A estrutura de desvio de fluxo, ou de seleção, é utilizada para desviar a seqüência de execução das instruções que integram um programa. Este desvio pode ser feito como resultado de uma comparação de uma expressão ou de um valor de determinada variável.

Na linguagem Java existem três tipos de instruções de seleção: a instrução ***if***; a instrução ***if...else***; e a instrução ***switch***.

6.1 A instrução *if*:

É caracterizada como uma instrução de seleção única, uma vez que ignora ou seleciona uma única seqüência de ações. Temos, como exemplo, no pseudo-código abaixo:

Se salário for menor ou igual do que R\$ 1.200,00

Imprimir não tem desconto de IR;

Caso o salário seja menor do que R\$ 1.200,00 não haverá desconto de Imposto de Renda.

Outro exemplo:

Se média final for maior ou igual a 5

Aluno aprovado;

Em Java a especificação da instrução ***if*** possui a seguinte sintaxe:

```
if (expressão de comparação)
    ação ;                ou
if (expressão de comparação) {
    ação1;
    ação2;.
    .
    ação n;
}
```

Esta última forma é utilizada quando existem inúmeras ações a serem seguidas. É importante observar que, neste caso, estas ações devem estar delimitadas pelos símbolos “{}”.

Escrevendo o pseudo-código em Java teríamos o seguinte código:

Para o primeiro exemplo:

```
if ( salario <= 1200 )  
    System.out.println(“Salário sem desconto de IR”);
```

Para o segundo exemplo:

```
if ( notaFinal >= 5 )  
    System.out.println(“Aluno Aprovado”);
```

Na segunda forma de utilização do *if* poderia ser feito:

```
if ( notaFinal == 10 ) {  
    System.out.println(“Aluno Aprovado”);  
    System.out.println(“Aluno Muito Bom”);  
    System.out.println(“Dar bolsa para o aluno”);  
}
```

6.2 As instruções *if.....else*:

Neste caso existe uma opção alternativa quando a condição testada no *if* não for satisfeita. Neste caso é realizada a ação especificada no *else*. No pseudo-código poderia ser feito:

Se a média final for maior ou igual a 5

Aluno aprovado

Caso contrário

aluno reprovado;

Escrevendo este código em Java teríamos:


```
if ( mediaFinal >= 5)
    System.out.println("Aprovado");
else
    System.out.println("Reprovado");

Ou

if ( mediaFinal >= 9) {
    System.out.println("Aprovado");
    System.out.println("Convidar para o Estágio");
}
else {
    System.out.println("Reprovado");
    System.out.println("Comunicar aos pais");
}
```

6.3 As instruções *if....else* aninhadas:

Consiste em utilizar instruções *if...else* dentro de outras instruções *if...else*, daí a denominação de *if..else* aninhados. No pseudo-código poderia ser feito:

```
Se salário menor ou igual do que R$ 1.200,00
    Imprimir não há desconto de IR
Senão se salário menor ou igual a R$ 1.500,00
    Imprimir desconto de IR de 10%
Senão se salário menor ou igual a R$ 2.500,00
    Imprimir desconto de IR de 20%
Senão
    Imprimir desconto de IR de 27.5%
```

Escrevendo este pseudo-código em Java teríamos:

```
if (salario <= 1200)

    System.out.println("não há desconto de IR");

else if (salario <= 1500)

    System.out.println("desconto de IR de 10%");

else if (salario <= 2500)

    System.out.println("desconto de IR de 20%");

else

    System.out.println("desconto de IR de 27.5%");
```

É sempre uma boa prática de programação utilizar as chaves “{}” mesmo que sob o *if* ou sob o *else* haja apenas uma única instrução.

6.4 A instrução switch:

A instrução ***switch*** é classificada como uma instrução de seleção múltipla pelo fato de permitir que diferentes ações possam ser tomadas em função do valor assumido por alguma variável ou por uma expressão. O ***switch*** trata apenas tipos enumerados e expressões ou variáveis do tipo inteiro. Variáveis do tipo ***float*** ou ***double*** não são tratadas por esta instrução. As comparações são feitas através da palavra-chave ***case***.

O ***switch*** possui a seguinte sintaxe:

```
switch (expressão ou variável){

    case valor-a-ser-testado1:

        instrução 1;

        instrução 2;

        instrução n;

        break;

    case valor-a-ser-testado2:

        instrução 1;

        instrução n;

        break;
```

```
default:  
    instrução 1;  
    instrução n;  
}
```

Vamos escrever um pseudo-código que testará qual o mês que está sendo fornecido para a aplicação:

verifica (mês)

```
caso seja 01 imprimir "Janeiro" e pára a comparação  
caso seja 02 imprimir "Fevereiro" e pára a comparação  
caso seja 03 imprimir "Março" e pára a comparação  
caso seja 04 imprimir "Abril" e pára a comparação  
caso seja 05 imprimir "Maio" e pára a comparação  
caso seja 06 imprimir "Junho" e pára a comparação  
caso seja 07 imprimir "Julho" e pára a comparação  
caso seja 08 imprimir "Agosto" e pára a comparação  
caso seja 09 imprimir "Setembro" e pára a comparação  
caso seja 10 imprimir "Outubro" e pára a comparação  
caso seja 11 imprimir "Novembro" e pára a comparação  
caso seja 12 imprimir "Dezembro" e pára a comparação  
caso não seja nenhum acima imprimir "Mês Errado"
```

O código Java correspondente é:

```
switch (mes){  
    case 1: System.out.println("Janeiro");break;  
    case 2: System.out.println("Fevereiro");break;  
    case 3: System.out.println("Março");break;  
    case 4: System.out.println("Abril");break;  
    case 5: System.out.println("Maio");break;  
    case 6: System.out.println("Junho");break;
```

```
case 7: System.out.println("Julho");break;
case 8: System.out.println("Agosto");break;
case 9: System.out.println("Setembro");break;
case 10: System.out.println("Outubro");break;
case 11: System.out.println("Novembro");break;
case 12: System.out.println("Dezembro");break;
default: System.out.println("Mês Errado");
}
```

O comando **break** faz com que as comparações sejam interrompidas, evitando que os **cases** subsequentes sejam processados mesmo se a comparação for satisfeita. O comando **default**, que é de uso opcional, será tratado somente se todas as comparações anteriores forem mal sucedidas.

Para melhor entendermos o comando **break**, a seguir é apresentado um pseudo-código para verificar quantos dias determinado mês possui.

Verifica(mês)

Caso seja mês 1 ou

Caso seja mês 3 ou

Caso seja mês 5 ou

Caso seja mês 7 ou

Caso seja mês 8 ou

Caso seja mês 10 ou

Caso seja mês 12

Imprimir "Mês com 31 dias"

Pára a comparação

Caso seja mês 4 ou

Caso seja mês 6 ou

Caso seja mês 9 ou

Caso seja mês 11 ou

Caso seja mês 12

Imprimir “Mês com 30 dias”

Parar a comparação

Caso seja mês 2

Se ano bi-sexto

Imprimir “Mês com 29 dias”

Senão

Imprimir “Mês com 28 dias”

Parar a comparação

Caso não seja nenhum acima imprimir “Mês Errado”

O correspondente código Java seria:

```
switch (mes){  
  
    case 1:  
  
    case 3:  
  
    case 5:  
  
    case 7:  
  
    case 8:  
  
    case 10:  
  
    case 12:  
        System.out.println("Este mês possui 31 dias");  
        break;  
  
    case 4:  
  
    case 6:  
  
    case 9:  
  
    case 11:  
        System.out.println("Este mês possui 30 dias");  
        break;  
  
    case 2:
```

```
if ( ano % 4 == 0)
    System.out.println("Este mês possui 29 dias");
else
    System.out.println("Este mês possui 28 dias");
break;
default: System.out.println("Valor Errado");
```

6.5 Exemplos com métodos de manipulação de strings:

A seguir são apresentados alguns casos de aplicação das instruções *if* e *else* com alguns dos métodos da classe `String`, que retornam um valor booleano do tipo *true* ou *false*. Inicialmente será utilizado o método *equals()*, que tem como propósito comparar em tamanho e conteúdo duas strings distintas.

Exemplo 1

Comparar se duas strings fornecidas pelo teclado são iguais.

`String nome1, nome2;`

```
Scanner entrada = new Scanner(System.in);
System.out.println("Digite a primeira string: ");
nome1 = entrada.next();
System.out.println("Digite a segunda string: ");
nome2 = entrada.next();
if (nome1.equals(nome2))
    System.out.println("Strings Iguais");
else
    System.out.println("Strings Diferentes");
```

No exemplo anterior o método *equals()* é case sensitive, logo letras maiúsculas serão consideradas diferentes de letras minúsculas, para que seja ignorada esta condição deve ser utilizado o método *equalsIgnoreCase()*.

O método *startsWith()* verifica se a string começa com determinado valor.

Exemplo2

Verificar se determinada palavra fornecida pelo teclado inicia com a string “Fut”.

String palavra;

```
Scanner entrada = new Scanner(System.in);  
System.out.println("Digite a palavra: ");  
palavra = entrada.next();  
if (palavra.startsWith("Fut"))  
    System.out.println("Inicia com Fut");  
else  
    System.out.println("Não inicia com Fut");
```

O método *endsWith()* verifica se a string termina com determinado valor.

Exemplo3

Verificar se determinada palavra fornecida pelo teclado termina com a string “bol”.

String palavra;

```
Scanner entrada = new Scanner(System.in);  
System.out.println("Digite a palavra: ");  
palavra = entrada.next();  
if (palavra.endsWith("bol"))  
    System.out.println("Termina com bol");  
else  
    System.out.println("Não termina com bol");
```

O método ***regionMatches()*** verifica se regiões de duas strings distintas são iguais. Este método pode receber quatro argumentos: o índice inicial da região da primeira string; nome ou conteúdo da primeira string a ser comparada; índice inicial da região da segunda string; tamanho da região a ser comparada.

Exemplo 4

Comparar se na segunda posição de uma string fornecida pelo teclado tem o valor “rasi”.

String palavra;

```
Scanner entrada = new Scanner(System.in);  
System.out.println("Digite a palavra: ");  
palavra = entrada.next();  
if (palavra.regionMatches(1,"rasi", 0, 4 ))  
    System.out.println("Possui rasi");  
else  
    System.out.println("Não possui rasi");
```

7. Estrutura de repetições:

A estrutura de repetição é utilizada para repetir, de forma controlada, a execução de determinado conjunto de instruções que integram o trecho de um programa. Esta repetição pode ser condicionada ao valor de determinada variável ou de uma expressão.

Na linguagem Java existem três tipos de instruções de repetição: a instrução ***for***; a instrução ***while***; e a instrução ***do...while***.

7.1. A instrução ***for***:

A instrução ***for*** executa repetidamente um determinado conjunto de instruções. Estas instruções integram um bloco único e o processo de repetição é denominado de ***loop***.

A instrução **for** possui a seguinte sintaxe:

```
for (valor-inicial; condição-de-repetição; incremento-ou-decremento-do-valor-  
inicial) {  
    (condicional) continue;  
    instrução 1;  
    instrução 2;  
    instrução n;  
    break; (opcional)  
}
```

O argumento valor-inicial especifica o valor de partida para o número de repetições do conjunto. O argumento condição-de-repetição estabelece a condição com que o número de repetições ficará delimitado. O argumento incremento-ou-decremento-do-valor-inicial estabelece o valor que será incrementado ou decrementado do valor inicialmente estipulado.

Para melhor entendermos o comando **for**, a seguir é apresentado um pseudocódigo para imprimir a frase “Estou aprendendo Java” 10 vezes consecutivas.

Início: Faça valor inicial igual a 1

Se o valor inicial for menor ou igual a 10 imprima:

“Estou aprendendo Java”

Incrementar valor inicial de uma unidade e voltar para o Início

O código correspondente em Java para este pseudocódigo é:

```
for (int i=1; i<=10; i++){  
    System.out.println("Estou aprendendo Java !");  
}
```

A instrução **break**, quando utilizada, interrompe o ciclo de repetições, enquanto a instrução **continue** salta todas as instruções que a sucedem retornando o controle para a próxima interação. Esta instrução deve ser utilizada sempre em uma instrução de controle do tipo **if**.

No Java 1.5 o comando **for** foi aprimorado, fazendo um tratamento especial para a

manipulação de *arrays* e de coleções como veremos em seções posteriores.

7.2. A instrução *while*:

A instrução *while* executa condicionalmente um determinado conjunto de instruções que também integram um bloco único. Enquanto a condição for verdadeira o bloco de instruções é repetidamente executado. Este processo de repetição também é denominado de *loop*.

A instrução *while* possui a seguinte sintaxe:

```
while (condição-a-ser-testada) {  
    (condicional) continue;  
    instrução 1;  
    instrução 2;  
    instrução n;  
    break; (opcional)  
}
```

No pseudocódigo abaixo é lido um conjunto de notas de alunos até for digitada uma nota negativa.

```
atribui a nota valor inicial zero  
enquanto a nota não for negativa  
    leia a nota do aluno  
    acumula a nota  
imprimir o somatório das notas
```

O código Java correspondente ao pseudocódigo seria:

```
Scanner entrada = new Scanner(System.in);  
  
int notaAluno = 0;  
  
int totalNotas = 0;
```

```
while(notaAluno>=0) {  
  
    totalNotas+= notaAluno;  
    System.out.println("Digite a nota:");  
    notaAluno = entrada.nextInt();  
}  
  
System.out.printf("%s%d\n","Total Notas: ", totalNotas);
```

7.3.A instrução *do...while*:

A instrução *do...while* difere-se da instrução *while* no fato de que a condição é testada ao fim da execução do bloco e não no início.

A instrução *do....while* possui a seguinte sintaxe:

```
do {  
    continue; (condicional)  
    instrução 1;  
    instrução 2;  
    instrução n;  
    break; (opcional)  
} while (condição-a-ser-testada)
```

No pseudocódigo abaixo é impresso 10 vezes a frase “Estou aprendendo Java”

```
Inicia contador com valor 1  
faça  
    imprimir “Estou aprendendo Java”  
    incrementa contador de uma unidade  
enquanto contador for menor ou igual a 10
```

O código Java correspondente ao pseudocódigo é apresentado a seguir:

```
int contaImprime = 1;
do{
    contaImprime++;
    System.out.println("Estou aprendendo Java");
} while(contaImprime<=10);
```

Se desejarmos inibir uma única impressão utilizando a instrução *continue* podemos fazer:

```
int contaImprime = 1;
while(contaImprime<=10) {
    contaImprime++;
    if (contaImprime == 4) continue;
    System.out.println("Estou aprendendo Java");
}
```

Aqui, o número quatro, na expressão da instrução *if*, foi escolhido aleatoriamente, qualquer número entre 2 e 10 poderia ser o escolhido.

Tanto no *while* quanto no *do while* a instrução *break* interromperá a sequência de interações.

8. Invocando métodos de outras classes:

Nesta seção aprenderemos como invocar métodos que estão definidos em outras classes. Criaremos, inicialmente, duas classes:

- GeraMensagem
 - Esta classe contém um método denominado apresentaMensagem que, simplesmente, apresentará no desktop a mensagem “Estou aprendendo Java”.
- TestaMensagem

- Esta classe instanciará a classe `GeraMensagem`, e acionará o método `apresentaMensagem`.

O código da classe `GeraMensagem` é apresentado a seguir:

```
public class GeraMensagem{  
    public void apresentaMensagem() {        // método apresentaMensagem  
        System.out.println("Estou aprendendo Java");  
    }  
}
```

O código da classe `TestaMensagem` é:

```
public class TestaMensagem{  
    public static void main(String[] args) {  
        GeraMensagem gera = new GeraMensagem();    //instancia a classe  
                                                    // GeraMensagem  
        gera.apresentaMensagem();                // aciona o método  
                                                    // da classe  
    }  
}
```

Ao ser executada a classe `TestaMensagem` o seu método *main* é, automaticamente, acionado pela Java Virtual Machine. No método será instanciada a classe `GeraMensagem` criando o objeto `gera`. Como esta classe define o método `apresentaMensagem`, ele é invocado através do comando `gera.apresentaMensagem()`, produzindo o resultado desejado. A palavra-chave *void* no método `apresentaMensagem` indica que o método nada retornará para aquele objeto que o acionou.

8.1.Passando argumentos para os métodos de outras classes:

Na seção anterior a classe `TestaMensagem` não passava nenhum argumento para o método `apresentaPassagem`. Ela simplesmente invocava o método que então processava o conjunto de instruções. Vamos modificar as duas classes, adicionando algumas novas funcionalidades:

- GeraMensagem
 - O método **apresentaMensagem** desta vez receberá uma string como argumento de entrada. Esta string será apresentada no desktop.
- TestaMensagem
 - Irá capturar a string do teclado e invocará o método **apresentaMensagem()** passando a string como argumento.

Novo código da classe **GeraMensagem**:

```
public class GeraMensagem{  
    public void apresentaMensagem(String mensagem) {  
        System.out.println(mensagem);  
    }  
}
```

Novo código da classe **TestaMensagem**:

```
import java.util.Scanner;  
  
public class TestaMensagem{  
    public static void main(String[] args) {  
        String mensagem;  
        Scanner entrada = new Scanner(System.in);  
        System.out.println("Digite a mensagem:");  
        mensagem = entrada.nextLine();  
        GeraMensagem gera = new GeraMensagem();  
        gera.apresentaMensagem(mensagem);    // aqui a mensagem está sendo  
                                              // passada para o método  
    }  
}
```

Então podemos observar que para passarmos um argumento para um método basta colocar o argumento entre os parênteses que constituem o método.

Na classe **TestaMensagem** foi utilizada um outro método da classe **Scanner**, o **nextLine()**, este método lê integralmente a linha em que os dados foram digitados, sendo considerados todos os espaços em branco digitados, tanto à direita quanto à esquerda de qualquer palavra.

8.2.Passando argumentos para os métodos de outras classes e recebendo retorno:

Nesta seção serão criadas duas classes distintas:

- **Calcular**
 - Esta classe possui o método **calculaProduto** que tem como propósito receber dois números inteiros como argumentos, calcula o produto destes dois números e retorna o resultado para quem o a acionou. O retorno de um resultado, em Java, é representado pela instrução **return**.
- **TestaCalcular**
 - Esta classe captura dois números inteiros do teclado e passa-os, como argumentos, para o método **calculaProduto**, da classe **Calcular**, o resultado do cálculo, retornado pelo método **calculaProduto**, é apresentado no desktop.

Segue o código da classe **Calcular**:

```
public class Calcular{  
    public int calcularProduto(int multiplicando, int multiplicador) {  
        return multiplicando * multiplicador;  
    }  
}
```

E o código da classe **TestaCalcular**:

```
import java.util.Scanner;

public class TestaCalcular{

    public static void main(String[] args) {

        int a,b;

        Scanner entrada = new Scanner(System.in);

        System.out.println("Digite o multiplicando:");

        a = entrada.nextInt();

        System.out.println("Digite o multiplicador:");

        b = entrada.nextInt();

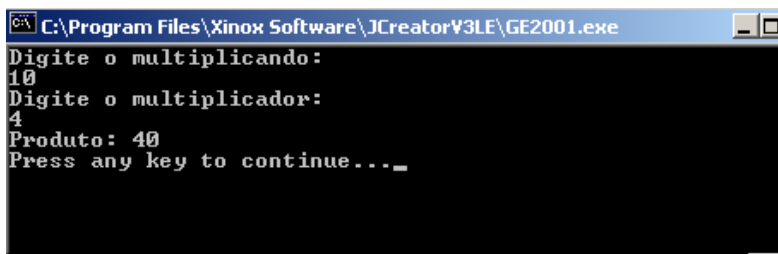
        Calcular multiplicar = new Calcular();

        System.out.printf("%s%d\n","Produto:" ,multiplicar.calcularProduto(a,b));

    }

}
```

Deve ser observado que na especificação do método **calcularProduto**, da classe **Calcular**, a palavra-chave **void** foi substituída pela palavra-chave **int**. Isto significa que o método retornará, para quem o invocar, um valor do tipo inteiro.



A figura ao lado apresenta o resultado da execução da classe **TestaCalcular** invocando o método **calcularProduto** da classe **Calcular**.

8.3. Variáveis de instância:

As variáveis que são definidas no interior de um método são denominadas variáveis locais. Este tipo de variável é visível apenas para o método que as contém, qual seja, outros métodos da classe não tem acesso às variáveis locais deste método.

As variáveis de instância são aquelas que são definidas no corpo da classe e não no corpo dos métodos. Elas têm como principal característica o fato de serem visíveis para todos aqueles métodos que integram a classe.

Normalmente as variáveis de instância são utilizadas para preservar valores comuns aos métodos e que são importantes durante toda a execução da aplicação, onde podem ser modificadas ou recuperadas.

Para modificar ou recuperar os valores de uma variável de instância utiliza-se, segundo a boa prática da POO, métodos distintos. Os métodos que modificam o conteúdo das variáveis devem ter seu nome precedido pela palavra **set**, enquanto aqueles métodos que recuperam os valores devem ser precedidos pela palavra **get**. Então, um método que recupera a nota de um aluno deve chamar-se, por exemplo, **getNota** enquanto o método que modifica a nota deve chamar-se **setNota**. Cumpra observar que este padrão é apenas uma recomendação, porém poderá ser útil quando for utilizada uma facilidade do Java denominada Javabeans.

No exemplo a seguir serão criadas duas classes:

- ContaCorrente
 - Esta classe possui dois métodos que administram a conta corrente de uma pessoa realizando operações de consulta a saldo, depósito e retirada de valores. Os métodos são, respectivamente, **getSaldo()** e **setSaldo()**. Esta classe possuirá uma variável de instância relativa ao saldo que será comum aos dois métodos.
- TestaContaCorrente
 - Esta classe inicialmente iniciará a variável de instância com o valor de saldo e depois fará operações de débito, crédito e consultas ao saldo.

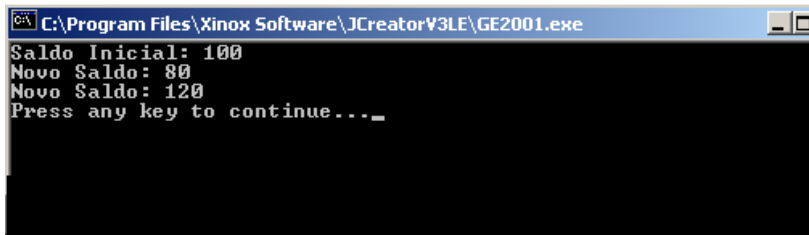
A classe **ContaCorrente** possui o seguinte código:

```
public class ContaCorrente{  
    public int saldoConta;  
    public int getSaldo(){  
        return saldoConta;  
    }  
    public void setSaldo(int valor){  
        saldoConta+= valor;  
    }  
}
```

Pode ser observado que o método **getSaldo()** apenas retorna o valor atualizado do saldo. Este valor é somente alterado pelo método **setSaldo()**. Por ser uma variável de instância, **saldoConta()** fica visível para ambos os métodos da classe **ContaCorrente**.

A seguir é apresentado o código da classe **TestaContaCorrente**. O valor do saldo é iniciado com R\$ 100 e depois são feitas uma operação de débito de R\$ 20,00 e uma operação de crédito de R\$ 40,00, em ambas as operações o resultado é apresentado no desktop:

```
public class TestaContaCorrente{  
    public static void main(String[] args){  
        ContaCorrente minhaconta = new ContaCorrente(); // instancia a classe  
        minhaconta.setSaldo(100); // inicia a conta com R$ 100,00  
        System.out.printf("%s%d\n", "Saldo Inicial: ", minhaconta.getSaldo()); // obtém saldo  
        minhaconta.setSaldo(-20); // debita R$ 20,00 da conta  
        System.out.printf("%s%d\n", "Novo Saldo: ", minhaconta.getSaldo()); // obtém saldo  
        minhaconta.setSaldo(40); // deposita R$ 40,00 na conta  
        System.out.printf("%s%d\n", "Novo Saldo: ", minhaconta.getSaldo()); // obtém saldo  
    }  
}
```



A figura ao lado apresenta o resultado da execução da classe **TestaContaCorrente**.

9.Arrays:

Um **array** é uma área de memória em que pode ser armazenado um grupo fixo e homogêneo de elementos. Estes elementos podem ser valores primitivos de dados ou referências para objetos. Vamos, por exemplo, supor que desejamos ler as notas dos alunos de determinada turma e imprimi-las em ordem ascendente. Sem o auxílio dos **arrays** haveria um grau de complexidade maior no algoritmo para realizar tal tarefa. Para declararmos um **array** precisamos saber:

- Quantos elementos terá o **array** ?
- Que tipo de elemento haverá no **array** ?

Para armazenar ou localizar os elementos dentro do **array** são utilizados índices. Estes índices são necessariamente números inteiros, fazendo com que um **array** de uma única dimensão possa abrigar até 2.147.483.647 elementos. Em Java os índices iniciam sempre com o valor zero e terminam com o tamanho do **array** - 1 e, uma vez declarados, os tamanhos dos **arrays** não podem ser modificados. Caso um **array** de tipos primitivos numéricos tenha sido declarado, mas não iniciado, todos os seus elementos terão o valor zero enquanto se o tipo for *booleano* os elementos serão iniciados como *false*.

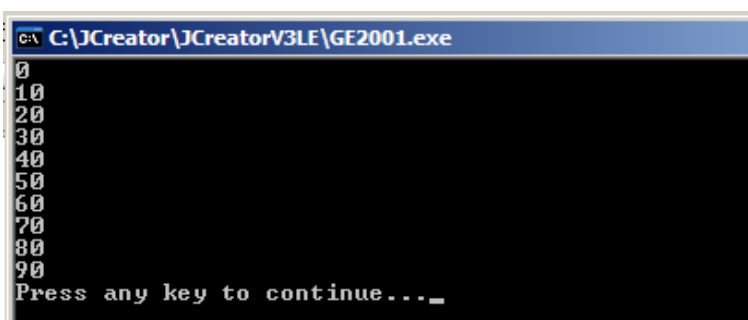
A declaração de um **array**, com tipos primitivos, possui a seguinte sintaxe:

```
tipo-primitivo[] nome-do-array = new tipo-primitivo[número-de-elementos];
```

Deve ser observado que em Java **array** é uma classe e que, portanto, deve ser sempre instanciada.

No exemplo a seguir é declarado um **array** com 10 elementos. Cada um dos elementos conterà um número inteiro positivo iniciando em 0 e terminando com 90.

```
public class TesteArray{  
    public static void main(String[] args){  
        int tabela[] = new int[10];  
        for (int i=0; i<= 9 ; i++){  
            tabela[i] = i * 10 ;  
            System.out.println(tabela[i]);  
        }  
    }  
}
```



A figura ao lado apresenta o resultado da execução da classe TesteArray.

O código também poderia ter sido escrito da seguinte forma:

```
public class TesteArray{  
    public static void main(String[] args){  
        int tabela[];  
        int N = 10;  
        tabela = new int[N];  
        for (int i=0; i<= 9 ; i++){  
            tabela[i] = i * 10 ;  
            System.out.println(tabela[i]);  
        }  
    }  
}
```

Os elementos de um *array* também podem ser iniciados durante a sua declaração

como no exemplo a seguir.

```
public class TesteArray{  
    public static void main(String[] args){  
        int tabela[] = {10 ,20, 30, 40, 50};  
        for (int i=0; i<= 4 ; i++){  
            System.out.println(tabela[i]);  
        }  
    }  
}
```

Neste exemplo o *array* denominado tabela é iniciado com cinco elementos valendo cada um 10, 20, 30, 40 e 50, respectivamente.

Vamos supor que nossa aplicação aciona um método de um objeto que retornará, por exemplo, um *array* de tipos numéricos inteiros. Para trabalharmos este *array* provavelmente teremos que saber quantos elementos ele possui. Para tal, existe uma variável da classe *array* denominada *length* que fornecerá esta informação. No código a seguir temos um exemplo de emprego desta variável.

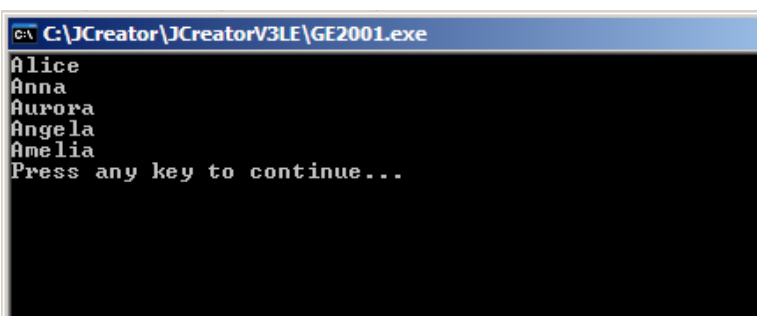
```
public class TesteArray{  
    public static void main(String[] args){  
        int tabela[] = {1 ,2, 3, 4, 5};  
        for (int i=0; i< tabela.length ; i++){  
            System.out.println(tabela[i]);  
        }  
    }  
}
```

É importante ser observado que a palavra reservada *length* não é sucedida por parênteses, uma vez que aqui ela é apenas uma variável de instância da classe *array* e não um método.

9.1.Array de strings:

No *array* de strings a declaração somente cria um *array* de referências para as strings. O objeto string não será criado cabendo-nos, portanto, realizar a tarefa de criar um objeto para cada elemento do *array*. No código a seguir é apresentado um exemplo de criação de um *array* de strings.

```
public class TesteArrayString{  
    public static void main(String[] args){  
        String strTabela[] = new String[5]; // declara o array de strings  
        strTabela[0] = "Alice";  
        strTabela[1] =new String ("Anna"); // forma alternativa de iniciar  
        strTabela[2] = "Aurora";  
        strTabela[3] = "Angela";  
        strTabela[4] = "Amelia";  
        for (int i=0; i< strTabela.length ; i++){  
            System.out.println(strTabela[i]);  
        }  
    }  
}
```



A figura ao lado apresenta o resultado da execução da classe TestaArrayString.

Uma vez que a variável é uma instância da class String todos os métodos associados a esta classe podem ser utilizados tais como *trim()*, *substring()*, *indexOf()* etc.

9.2.Array de arrays:

Em Java, *array* de *arrays* também são chamados de *arrays* multidimensionais, aqui cada elemento de um *array* contém uma referência para um outro objeto *array*

Um *array* multidimensional de duas dimensões é declarado da seguinte maneira:

```
tipo-do-dado [][] nome-do-array = new tipo-do-dado[][];
```

Para declararmos um *array* de string de uma dimensão 3 por 2 escreveríamos:

```
String [][] tabela = new String[3][2];
```

Que é semelhante a escrevermos:

```
String [][]tabela = new String[3][];
```

```
tabela [0] = new String[2];
```

```
tabela [1] = new String[2];
```

```
tabela [2] = new String[2];
```

Se desejarmos iniciar este *array* o código seria o seguinte:

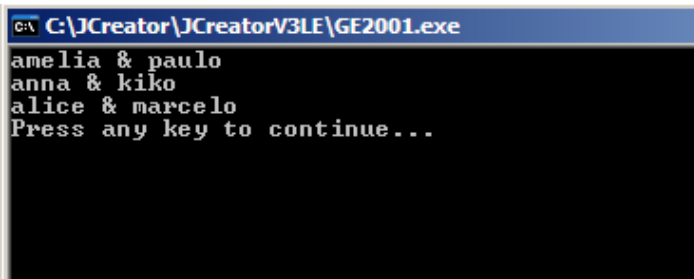
```
tabela [0] = new String[]{"alice","paulo"};
```

```
tabela [1] = new String[] {"anna", "kiko"};
```

```
tabela [2] = new String[] {"alice","marcelo"};
```

O código a seguir apresenta este código mais completo:

```
public class TesteArrayDimensional{  
    public static void main(String[] args){  
        String [][]tabela = new String[3][];  
        tabela [0] = new String[]{"amelia","paulo"};  
        tabela [1] = new String[] {"anna", "kiko"};  
        tabela [2] = new String[] {"alice","marcelo"};  
        for (int i=0; i< tabela.length ; i++){  
            System.out.println(tabela [i][0] + " & " + tabela [i][1]);  
        }  
    }  
}
```



A figura ao lado apresenta o resultado da execução da classe TestaArrayDimensional.

9.3.A classe Arrays:

A classe Arrays faz parte do pacote **java.util.Arrays** e fornece um conjunto de métodos que permite comparar, classificar e preencher *arrays*. Aqui veremos a algumas aplicações desta classe.

9.3.1 Arrays.equals:

Utilizado para comparar o conteúdo de dois *arrays*. Retornará uma variável do tipo *booleano* informando se os *arrays* são iguais ou não. Aplicável para tipos primitivos e para strings.

No código abaixo são comparados os conteúdos de dois *arrays* de strings.

```
import java.util.Arrays;

public class TestaArrays{

    public static void main(String[] args){

        String []tabela1 = new String[3];

        tabela1 [0] = new String ("amelia");

        tabela1 [1] = new String ("anna");

        tabela1 [2] = new String ("alice");


        String []tabela2 = new String[3];

        tabela2 [0] = new String ("amelia");

        tabela2 [1] = new String ("anna");
```



```
tabela2 [2] = new String ("alice");

if (Arrays.equals(tabela1,tabela2))
    System.out.println("São iguais");
}
```

9.3.2 Arrays.fill:

Utilizado para preencher com dados determinado *array*. Aplicável para tipos primitivos e para strings.

No código abaixo todos os elementos do *array* irão conter a string “maria”.

```
import java.util.Arrays;

public class TestaArrays{

    public static void main(String[] args){

        String []tabela1 = new String[3];

        tabela1 [0] = new String ("amelia");

        tabela1 [1] = new String ("anna");

        tabela1 [2] = new String ("alice");

        Arrays.fill(tabela1,"maria");

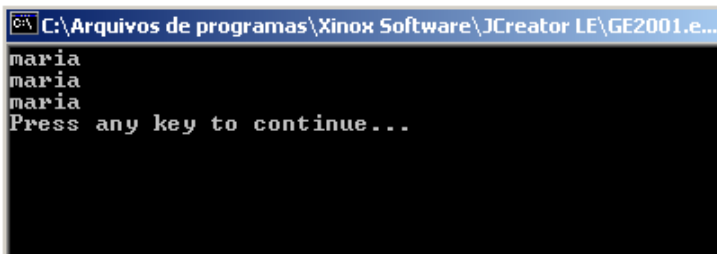
        for (int i=0; i<tabela1.length ; i++){

            System.out.println(tabela1[i]);

        }

    }

}
```



```
C:\Arquivos de programas\Xinox Software\JCreator LE\GE2001.e...
maria
maria
maria
Press any key to continue...
```

A figura ao lado apresenta o resultado da execução da classe TestaArrays onde todos os elementos do *array* passaram a ter seu conteúdo igual a string “maria”.

9.3.3 Arrays.sort:

Utilizado para classificar, em ordem ascendente, os dados de um *array*. Aplicável somente para os tipos primitivos.

No código abaixo são classificados os elementos de um *array* de números inteiros.

```
import java.util.Arrays;

public class TestaArrays{

    public static void main(String[] args){

        int[] tabela1 = {10,3,80,6,7,-3};

        Arrays.sort(tabela1);

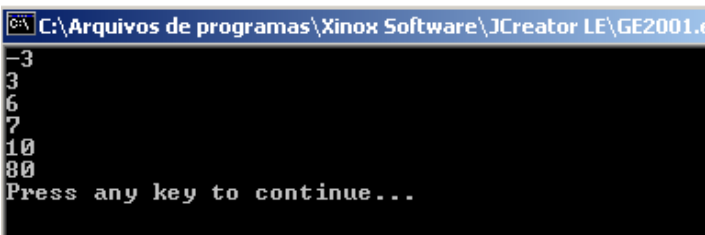
        for (int i=0; i<tabela1.length ; i++){

            System.out.println(tabela1[i]);

        }

    }

}
```



```
C:\Arquivos de programas\Xinox Software\JCreator LE\GE2001.e...
-3
3
6
7
10
80
Press any key to continue...
```

A figura ao lado apresenta o resultado da execução da classe TestaArrays, onde o *array* tabela1 foi classificado em ordem ascendente.

10. Pacotes, Acessibilidade e Construtores:

10.1 Pacotes:

Na linguagem Java todo o código existente é armazenado em classes. Na API (Application Programming Interface) do Java grupos de classes são organizados em pacotes e apenas as classes armazenadas no pacote `java.lang` são automaticamente disponibilizadas, dispensando-nos da sua importação.

Aqueles pacotes que não integram o grupo pertencente ao `java.lang` precisam ser importados pelo desenvolvedores caso as classes que os integram precisam ser utilizadas.

Um pacote pode ser definido como sendo um diretório onde residirão as classes do aplicativo que iremos desenvolver.

Para importar um pacote é utilizado o comando ***import***, que deve ser incluído antes da definição da classe, e que possui a seguinte sintaxe:

```
import nome-do-pacote.Nome-da-Classe;
```

ou

```
import nome-do-pacote.*;
```

Na primeira forma do comando está sendo importada apenas uma classe do pacote enquanto na segunda forma o “*” indica que estão sendo importadas todas as classes integrantes do pacote.

A boa prática de POO recomenda que as classes que integram uma aplicação devem ser organizadas em pacotes e para tal os seguintes passos devem ser seguidos:

- Criar um diretório para o pacote. O nome do diretório é o mesmo nome do pacote.
- Mover todos os arquivos *.java para o diretório.
- Adicionar o comando ***package*** que identifica o pacote das classes no início do código de cada classe que integra o pacote.

O comando ***package*** tem a seguinte sintaxe:

```
package nome-do-pacote;
```

Caso alguma classe não pertencente ao pacote precise utilizar alguma de suas classes deverá ser codificado o comando ***import***.

No código a seguir a classe **PrimeiraLetra** faz parte do pacote Estacio/POO.

```
package Estacio.POO;

public class PrimeiraLetra{

    private float altura,largura;

    public PrimeiraLetra(){

    }

    public PrimeiraLetra(float alt, float larg){

        altura = alt;

        largura = larg;

    }

    public void testa() {

        System.out.printf("%s%.2f%s%.2f\n","Valores fornecidos ", altura, " e " ,
        largura);

    }

}
```

10.2 Acessibilidade:

A acessibilidade de uma classe, de um método ou de um atributo especifica como estes elementos serão ou não vistos e utilizados por outras classes. A acessibilidade vem formalizar um dos paradigmas da POO que é o encapsulamento de dados. Os atributos e métodos de determinada classe podem ser ocultados de qualquer outra classe, restringindo o acesso e agregando integridade aos dados manipulados pela classe.

A acessibilidade é controlada pelas palavras-chave **public**, **private** e **protected** que também são chamadas de *modificadores de acesso*.

Estes modificadores possuem o seguinte significado:

- **public**
 - Aplica-se às classes, métodos e atributos. Permite que qualquer outra classe, de qualquer pacote, tenha acesso a estes elementos.
- **private**
 - Aplica-se às classes, métodos e atributos. Permite que apenas a própria classe tenha acesso aos seus elementos, nenhuma classe do pacote pode acionar um elemento definido como *private*.

- **protected**
 - Aplica-se aos métodos e atributos. Permite que apenas as classes do mesmo pacote tenham acesso a estes elementos.

10.3 Construtores:

Como descrito na seção 8 para criarmos uma instância de uma determinada classe utilizamos o operador *new*. Quando encontra este operador a Java Virtual Machine irá procurar, no código da classe que está sendo instanciada, um método especial denominado método construtor que é efetivamente quem dará vida ao objeto. Neste método podem ser iniciadas as variáveis de instância, podem ser acionados métodos da própria classe, ou também podem ser instanciadas outras classes.

Um construtor deve ser definido sempre como público e precisa possuir o mesmo nome da classe que o define. Pode existir mais de um construtor em uma única classe desde que possua, conforme veremos a seguir, assinaturas diferentes. Caso o método construtor não seja codificado, a Java Virtual Machine assumirá a existência de um construtor *default*. Este construtor *default* iniciará todas as variáveis de instância do tipo numérico com o valor zero, as variáveis lógicas receberão o valor *false* e as variáveis objeto são iniciadas como *null*.

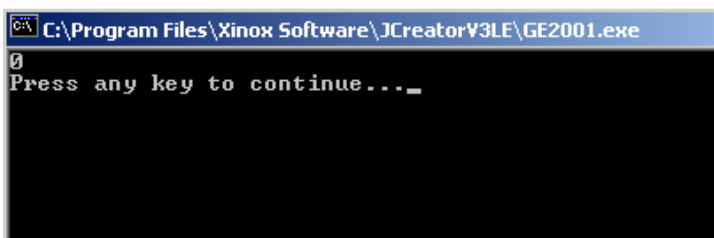
Em nosso primeiro exemplo é instanciada uma classe denominada **PrimeiraLetra** onde não foi especificado, inicialmente, nenhum construtor. Esta classe possui uma variável de instância numérica e um método denominado **testa()**. A classe **TestaConstrutor** instanciará a classe **PrimeiraLetra** e acionará o método **testa()**. Como a classe **PrimeiraLetra** não possui um construtor será acionado o seu construtor *default*. Este construtor iniciará a variável de instância com o valor zero, comprovando uma de suas funcionalidades.

A classe **PrimeiraLetra** possui o seguinte código:

```
public class PrimeiraLetra {  
    private int numero;  
    public void testa(){  
        System.out.println(numero);  
    }  
}
```

A classe **TestaConstrutor** possui o seguinte código:

```
public class TestaConstrutor{  
    public static void main(String args[]) {  
        PrimeiraLetra entrada = new PrimeiraLetra();  
        entrada.testa();  
    }  
}
```



A figura ao lado apresenta o resultado da execução da classe **TestaConstrutor**. Deve ser observado que o valor impresso da variável de instância *numero* é zero,

comprovando a ação do construtor *default*.

No próximo exemplo a classe **PrimeiraLetra** possui um construtor que iniciará a variável de instância *numero* com um valor passado durante a sua instanciação pela classe **TestaConstrutor**.

A classe **PrimeiraLetra** possui agora o seguinte código:

```
public class PrimeiraLetra{  
    private int numero;  
    public PrimeiraLetra(int onumero){  
        numero = onumero;  
    }  
    public void testa() {  
        System.out.printf("%s%d\n", "O numero vale ", numero);  
    }  
}
```

E a classe **TestaConstrutor**:

```
public class TestaConstrutor{
```

```

    public static void main(String[] args){

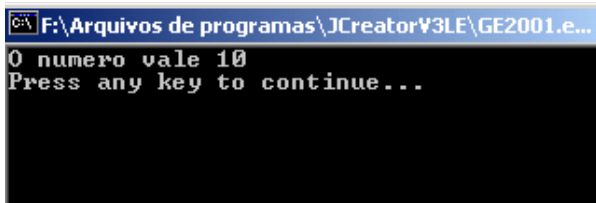
        PrimeiraLetra entrada = new PrimeiraLetra(10); // inicia com o valor 10

        entrada.testa();

    }

}

```



A figura ao lado apresenta o resultado da execução da classe **TestaConstrutor**. Na instanciação da classe **PrimeiraLetra** ela recebe o valor dez. Este valor será passado para o seu método construtor que iniciará a variável

de instância da classe.

No próximo exemplo a classe **PrimeiraLetra** possui duas variáveis de instância denominadas *altura* e *largura*. A classe também possui dois construtores: um que não modificará os valores destas variáveis e outro que permitirá que elas sejam iniciadas com valores passados pela classe **TestaConstrutor**.

A classe **PrimeiraLetra** possui agora o seguinte código:

```

public class PrimeiraLetra{

    private float altura,largura;

    public PrimeiraLetra(){

    }

    public PrimeiraLetra(float alt, float larg){

        altura = alt;

        largura = larg;

    }

    public void testa() {

        System.out.printf("%s%.2f%s%.2f\n","Valores fornecidos ", altura, " e " , largura);

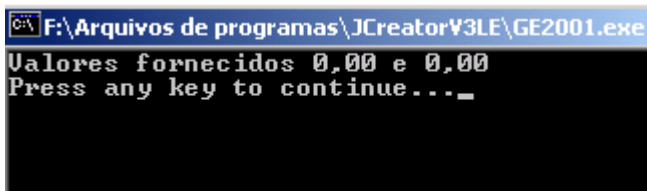
    }

}

```

A classe **TestaConstrutor** inicialmente não passará nenhum valor para a classe **PrimeiraLetra**, logo as variáveis de instância serão iniciadas com o valor zero. Seu código é o seguinte:

```
public class TestaConstrutor{  
  
    public static void main(String[] args){  
  
        PrimeiraLetra entrada = new PrimeiraLetra();  
  
        entrada.testa();  
  
    }  
  
}
```

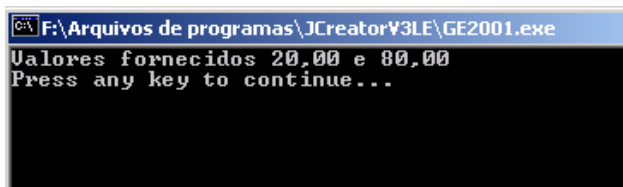


A figura ao lado apresenta o resultado da execução da classe **TestaConstrutor** sem passar valores para a classe **PrimeiraLetra**, conseqüentemente os valores de altura

e largura são zero.

No exemplo a seguir a classe **TestaConstrutor** passa os valores 20 e 80 para as variáveis de instância *altura* e a *largura*. O código ficou da seguinte maneira:

```
public class TestaConstrutor{  
  
    public static void main(String[] args){  
  
        PrimeiraLetra entrada = new PrimeiraLetra(20, 80);  
  
        entrada.testa();  
  
    }  
  
}
```



A figura ao lado apresenta o resultado da execução da classe **TestaConstrutor**. As variáveis de instância *altura* e *largura* foram iniciadas pelo construtor da classe

PrimeiraLetra com os valores passados por **TestaConstrutor**.

Neste último exemplo deve também ser observado que os métodos construtores têm o mesmo nome, mas como recebem argumentos distintos são tratados como se fossem métodos também distintos, diz-se que métodos deste tipo possuem **diferentes assinaturas**.

11. Herança e Polimorfismo:

11.1. Herança:

Um dos mais importantes paradigmas da Programação Orientada a Objetos refere-se à capacidade que uma classe possui de herdar atributos e métodos de outra classe. Todos os atributos e métodos definidos como *public* ou *protected* podem ser sobre-escritos por qualquer outra classe. A classe que disponibiliza seus atributos e métodos é chamada de **superclasse** e a classe que herda é chamada de **subclasse**.

Cumpre observar que o principal motivo de ser criada uma subclasse é explorar a capacidade de criar uma nova versão do método herdado, qual seja, poder alterar o conteúdo de qualquer método *public* ou *protected* da superclasse.

A sintaxe para ser declarada uma subclasse é:

```
public class nome-da-subclasse extends nome-da-superclasse{}
```

Para uma superclasse ser herdada é necessário que a subclasse utilize a palavra-chave ***extends*** e somente uma superclasse pode ser herdada por vez, por isto diz-se que a linguagem Java suporta apenas herança simples.

No exemplo a seguir é criada uma superclasse denominada **ClassePrincipal** e uma outra classe denominada **HerdaPrincipal**. A **ClassePrincipal** possui um método público **calculaArea** que, inicialmente, não será alterado pela classe **HerdaPrincipal**. A **ClassePrincipal** possui o seguinte código:

```
public class ClassePrincipal{  
  
    public float area;  
  
    public void calculaArea(float a, float b){  
  
        area = a * b;  
  
    }  
  
    public float getArea(){  
  
        return area;  
  
    }  
  
}
```

A subclasse **HerdaPrincipal** possui o seguinte código:

```
public class HerdaPrincipal extends ClassePrincipal {
```

```

    public static void main(String[] args){

        HerdaPrincipal objeto = new HerdaPrincipal ();

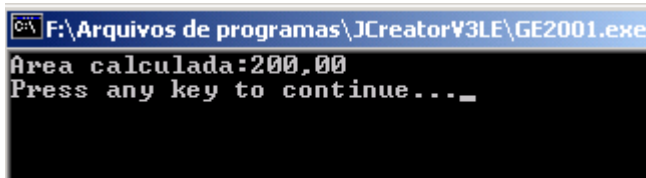
        objeto.calculaArea(10,20);

        System.out.printf("%s%.2f\n","Area calculada:", objeto.getArea());

    }

}

```



A figura ao lado apresenta o resultado da execução da classe **HerdaPrincipal**. São passados os valores 10 e 20 para ser calculada a área.

No próximo exemplo a classe **HerdaPrincipal** modifica o método **calculaArea**.

```

public class HerdaPrincipal extends ClassePrincipal {

    public void calculaArea(float a, float b){

        area= b - a;

    }

    public static void main(String[] args){

        HerdaPrincipal objeto = new HerdaPrincipal ();

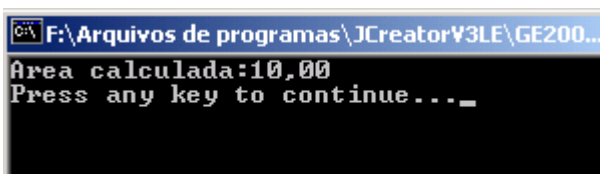
        objeto.calculaArea(10,20);

        System.out.printf("%s%.2f\n","Area calculada:", objeto.getArea());

    }

}

```



A figura ao lado apresenta o resultado da execução da classe **HerdaPrincipal**. Aqui o método **calculaArea** tem seu conteúdo alterado pela subclasse.

11.2. Polimorfismo:

Conceitualmente podemos dizer que polimorfismo é “o que se apresenta sob numerosas formas”. Na linguagem Java o polimorfismo se apresenta tão somente nas chamadas de seus métodos, qual seja, um método de mesmo nome pode retornar resultados de diversas formas, e quem decide as formas é o objeto que contém o método. Então se determinado objeto aciona um método de outro objeto é este quem determinará a forma com que o método se apresentará.

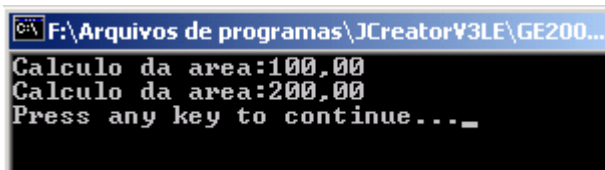
No próximo exemplo a classe **CalculaTudo** possui o método **calculaArea**. Este método possui duas assinaturas: a primeira calcula a área de um quadrado e a segunda calcula a área de um retângulo. A classe **TestaCalcular** invoca as duas assinaturas distintas do método.

Segue o código da classe **CalculaTudo**:

```
public class CalculaTudo{  
    public float area;  
    public void calculaArea(float a){  
        area = a * a;  
    }  
    public void calculaArea(float a, float b){  
        area = a * b;  
    }  
    public float getArea(){  
        return area;  
    }  
}
```

O código da classe **TestaCalcular** é:

```
public class TestaCalcular {  
  
    public static void main(String[] args){  
  
        CalculaTudo objeto = new CalculaTudo ();  
  
        objeto.calculaArea(10);  
  
        System.out.printf("%s%.2f\n","Calculo da area:", objeto.getArea());  
  
        objeto.calculaArea(10,20);  
  
        System.out.printf("%s%.2f\n","Calculo da area:", objeto.getArea());  
  
    }  
  
}
```



A figura ao lado apresenta o resultado da execução da classe **TestaCalcular** onde podem ser observados os retornos distintos do método **calculaArea**.

12. Classes Abstratas e Interfaces:

12.1. Classes Abstratas:

Uma classe abstrata serve como um modelo que pode ser herdado pelas subclasses. Uma classe abstrata não pode ser instanciada, ela apenas pode ser herdada. Uma classe abstrata pode conter métodos abstratos e métodos não abstratos. Quando ela define métodos abstratos necessariamente as suas subclasses precisam sobrecarregar estes métodos.

Os métodos não abstratos precisam conter um corpo e não precisam ser sobrecarregados nas subclasses. Os métodos de uma classe abstrata podem ser do tipo *public* ou do tipo *protected*.

A sintaxe da declaração de uma classe abstrata é:

```
public abstract class nome-da-classe{}
```

e a sintaxe da declaração de um método abstrato é:

```
public/protected abstract tipo-de-retorno nome-do-método(argumentos);
```

Deve ser observado que como o método é abstrato não existe nenhum código em

seu conteúdo.

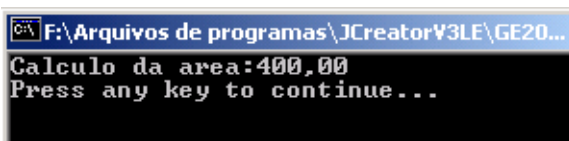
No exemplo a seguir é criada uma classe abstrata **ClasseAbstrata** que contém dois métodos **calculaArea** e **calculaTudo**. O método **calculaArea** é um método abstrato e, como tal, precisa ser implementado na subclasse. O método não abstrato **calculaTudo** é implementado na classe abstrata.

A seguir o código da classe **ClasseAbstrata**:

```
public abstract class ClasseAbstrata {  
    public abstract float calculaArea(float a);  
    public float calculaTudo(float a){  
        return calculaArea(a);  
    }  
}
```

A subclasse **TestaAbstrata** herda a classe **ClasseAbstrata** e implementa, obrigatoriamente, o método **calculaArea**, este método é acionado pelo método **calculaTudo** da classe **ClasseAbstrata**.

```
public class TestaAbstrata extends ClasseAbstrata{  
    public float calculaArea(float a){  
        return a * a;  
    }  
    public static void main(String[] args){  
        TestaAbstrata objeto = new TestaAbstrata ();  
        System.out.printf("%s%.2f\n","Calculo da area:",objeto.calculaTudo(20));  
    }  
}
```



A figura ao lado apresenta o resultado da execução da classe **TestaAbstrata**.

12.2.Interfaces:

Interfaces são muito semelhantes às classes abstratas. As principais diferenças são:

- Somente uma classe abstrata pode ser herdada por vez enquanto várias interfaces podem ser implementadas.
- Uma classe abstrata implementa métodos abstratos e não abstratos enquanto a interface somente implementa métodos abstratos.

Como a interface apenas implementa métodos abstratos, a interface também pode ser definida como sendo um contrato onde todas as classes que a implementam têm que codificar todos os métodos definidos.

A sintaxe para declarar uma interface é:

```
public interface nome-da-interface{}
```

Como todos os métodos definidos na interface são abstratos não existe a necessidade de explicitar esta palavra-chave quando estivermos declarando métodos em uma interface. Todos estes métodos são automaticamente declarados como públicos.

Para implementar uma interface a classe passa a ter a seguinte sintaxe:

```
public class nome-da-classe implements nome-da-interface{}
```

Aqui, a palavra-chave utilizada é **implements**.

No exemplo a seguir é declarada uma interface **InterfaceCalcula** que define um método **calculaArea**. A classe **TestaInterface** implementa esta interface e, como tal, tem que codificar o método **calculaArea**.

A seguir o código da interface **InterfaceCalcula**:

```
public interface InterfaceCalcula{  
    public float calculaArea(float a);  
}
```

E o código da classe **TestaInterface**:

```
public class TestaCoisas implements InterfaceCalcula{  
    public float calculaArea(float a){
```

```
        return a * a;
    }

    public static void main(String[] args){
        TestaCoisas objeto = new TestaCoisas();
        System.out.printf("%s%.2f\n","Calculo da area:",objeto.calculaArea(20));
    }
}
```

13.Vetores:

A classe **Vector** implementa uma estrutura de dados semelhante a um *array* sendo que, desta vez, os únicos elementos que ela pode armazenar são objetos. Estes objetos podem ser acessados através de várias formas e não apenas através de seus índices como acontece nos *arrays*. Na classe **Vector** os índices também iniciam com o valor zero. A classe **Vector** possui algumas características fundamentais tais como:

- Ter a capacidade de armazenamento variável, qual seja, o número dos objetos nela contidos pode variar dinamicamente, o tamanho do vetor aumenta conforme os objetos são inseridos.
- Como somente armazena objetos, tipos primitivos de dados não são suportados.
- Possui somente uma única dimensão.

A capacidade inicial de um vetor é sempre igual ou maior ao número de objetos que ele abriga. Se forem adicionados mais objetos, esta capacidade aumentará automaticamente de um valor especificado pelo desenvolvedor. Caso nada seja especificado a capacidade do vetor sempre dobrará de tamanho.

Para utilizar a classe **Vector** é necessário importar o pacote **Java.util.Vector**, e para instanciá-la o seguinte comando é utilizado:

```
Vector nome-do-objeto = new Vector (int valorInicial);    // ou
```

```
Vector nome-do-objeto = new Vector (int valorInicial, int valorIncremento);
```

Se nenhum argumento for passado para o construtor da classe **Vector**, é assumido um valor padrão de tamanho igual a dez e um incremento de capacidade padrão igual a zero.

A classe **Vector** possui inúmeros métodos e alguns dos mais importantes são apresentados na tabela abaixo:

<i>Método</i>	<i>Como funciona?</i>
add(objeto)	Adiciona um objeto ao fim do vetor.
capacity()	Retorna inteiro com a capacidade máxima corrente do vetor.
contains (objeto)	Retorna true se determinado objeto encontra-se armazenado no vetor.
get(indice)	Retorna o objeto que está armazenado na posição especificada em índice.
copyInto (array)	Copia todos os objetos do vetor para um array.
add (índice, objeto)	Adiciona um objeto na posição especificada pelo índice.
size()	Retorna um inteiro com o número de objetos armazenados no vetor.
remove(indice)	Remove o objeto localizado na posição especificada em índice.
clear()	Remove todos os objetos armazenados no vetor.
remove(objeto)	Remove a primeira ocorrência do objeto.
set(índice, objeto)	Substitui o objeto localizado na posição especificada em índice.
get(indice)	Copia para outra string uma porção da string a partir de determinada posição.
indexOf(objeto)	Retorna em que posição encontra-se determinado objeto.
isEmpty()	Retorna true se o vetor não contém nenhum objeto.
trimToSize()	Iguala a capacidade do vetor ao número de objetos armazenados.

Para que os tipos numéricos possam ser armazenados nos vetores é necessário que antes eles sejam convertidos para objetos. Para realizar esta conversão existem duas classes **Integer** e **Double**. A classe **Integer** é constituída dos métodos **parseInt()** e **toString()** e aplica-se para os tipos definidos como **int** e como **long** já a classe **Double** é constituída dos métodos **parseDouble()** e **toString()** se aplica para os tipos definidos como **double**. Os métodos **parseInt()** e **parseDouble()** convertem um objeto **String** para o tipo numérico correspondente enquanto o método **toString()** converte o tipo numérico para um objeto **String**.

Para converter uma *string* para um inteiro escrevemos, por exemplo:

```
int quantidade = Integer.parseInt(quantidadeString);
```

ou


```
double quantidade = Double.parseDouble(quantidadeString);
```

Para converter um número inteiro para uma *string* escrevemos:

```
String quantidadeString = Integer.toString(quantidade);
```

ou

```
String quantidadeString = Double.toString(quantidade);
```

No próximo exemplo um vetor é criado e são adicionados dois objetos. São apresentados: o tamanho do vetor; a sua capacidade inicial; o conteúdo do objeto armazenado na primeira posição do vetor.

```
import java.util.Vector;

public class TesteVetor{

    public static void main(String[] args){

        Vector vetor = new Vector(12); // Cria o vetor com capacidade para 12 objetos

        vetor.add("anna") ;

        vetor.add(new Integer(2)); // Aqui é adicionado um inteiro

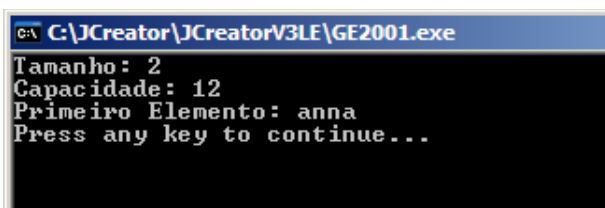
        System.out.printf("%s%d\n","Tamanho: ", vetor.size());

        System.out.printf("%s%d\n","Capacidade: ",vetor.capacity());

        System.out.printf("%s%s\n","Primeiro Elemento: ",vetor.get(0));

    }

}
```

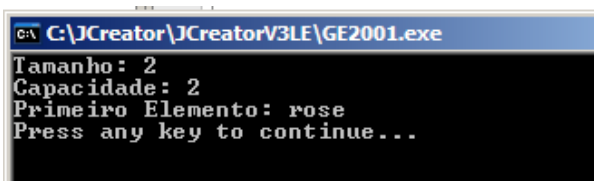


A figura ao lado apresenta o resultado da execução da classe **TestaVetor**. O segundo objeto incluído foi um tipo primitivo inteiro que foi transformado em um objeto através da instânciação da classe **Integer**.

No próximo exemplo um objeto substitui um outro e a capacidade de armazenamento do vetor é igualada ao número de objetos adicionados.

```
import java.util.Vector;
```

```
public class Teste{  
  
    public static void main(String[] args){  
  
        Vector vetor = new Vector(12);  
  
        vetor.add("anna") ;  
  
        vetor.add(new Integer(2)); // Aqui é incluído um inteiro  
  
        vetor.set(0,"rose");  
  
        vetor.trimToSize();  
  
        System.out.printf("%s%d\n","Tamanho: ", vetor.size());  
  
        System.out.printf("%s%d\n","Capacidade: ",vetor.capacity());  
  
        System.out.printf("%s%s\n","Primeiro Elemento: ",vetor.get(0));  
  
    }  
  
}
```



A figura ao lado apresenta a execução da classe **TestaVetor** onde pode ser observada a nova capacidade e o objeto substituto.