

# JAVA e a Programação Orientada a Objetos

## SUMÁRIO

PARTE 1 – Fundamentos Básicos da Linguagem Java .....	3
PARTE 2 – Principais Estruturas da Linguagem Java .....	15
PARTE 3 – Funções Matemáticas, Funções de String, Formatação de números e de data/hora .....	23
PARTE 4 – Introdução à exceções em Java .....	31
PARTE 5 – Array e Matrizes .....	35
PARTE 6 – Orientação a Objetos .....	40

## PARTE 1 – Fundamentos Básicos da Linguagem Java

Introdução

James Gosling e outros desenvolvedores da Sun estavam trabalhando em um projeto de TV interativa em meados da década de 1990, quando Gosling descontente com a linguagem C++, trancou-se em seu escritório e criou uma nova linguagem adequada ao projeto e que focalizava alguns de seus motivos de frustração na C++. O esforço de TV interativa da Sun falhou, mas seu trabalho na linguagem Java que facilitava a reescrita de software rendeu frutos.

Java foi lançada pela Sun no segundo semestre de 1995, em um kit de desenvolvimento gratuito, do qual era possível fazer o download pelo Web site da empresa. O fato de programas Java chamados applets poderem ser executados como parte das páginas Web contribui para atrair centenas de milhares de desenvolvedores.

Java é uma linguagem orientada a objetos, independente de plataforma e segura, projetada para ser mais fácil de aprender do que C++ e mais difícil de abusar do que C e C++.

A Programação Orientada a Objetos (P.O.O) é um técnica de desenvolvimento de software em que um programa é percebido como um grupo de objetos que trabalham juntos. Os objetos são criados como modelos, chamados classes, e contem os dados e as instruções necessárias para usar esses dados. Java é completamente orientada a objetos.

Independente de plataforma, um programa Java é executado sem modificações em diferentes sistemas operacionais. Os programas Java são compilados para um formato chamado bytecode, que é executado por qualquer sistema operacional, software ou dispositivo com um interpretador Java.

Características:

- Java cuida automaticamente da alocação e desalocação de memória;
- Java não inclui ponteiros;
- Java só inclui herança única;
- Segurança.

A falta de ponteiros e a presença de gerenciamento automático de memória são dois dos elementos-chave para a segurança de Java. Outro é a forma como os programas Java executados em páginas Web são limitados a um subconjunto da linguagem, para evitar que um código malicioso prejudique o computador de um usuário.

Os recursos da linguagem que poderiam facilmente ser empregados para fins prejudiciais – como a capacidade de gravar dados em um disco e excluir arquivos – não podem ser executados por um programa quando ele é executado pelo interpretador Java de um browser na Internet.

SDK

Software Development Kit é um conjunto de programas da linha de comando, usados para criar, compilar e executar programas Java. Cada nova versão da Java é acompanhada por uma nova versão do kit de desenvolvimento. Ele não oferece uma interface gráfica com o usuário, editor de textos ou outros recursos. Para usar o kit, é necessário digitar os comandos em um prompt de texto, ou seja, um prompt de linha de comando.

No entanto, existem diversas IDEs (Integrated Development Environment), um ambiente integrado para desenvolvimento de software, tais como: JCreator, Eclipse, NetBeans, dentre outros.

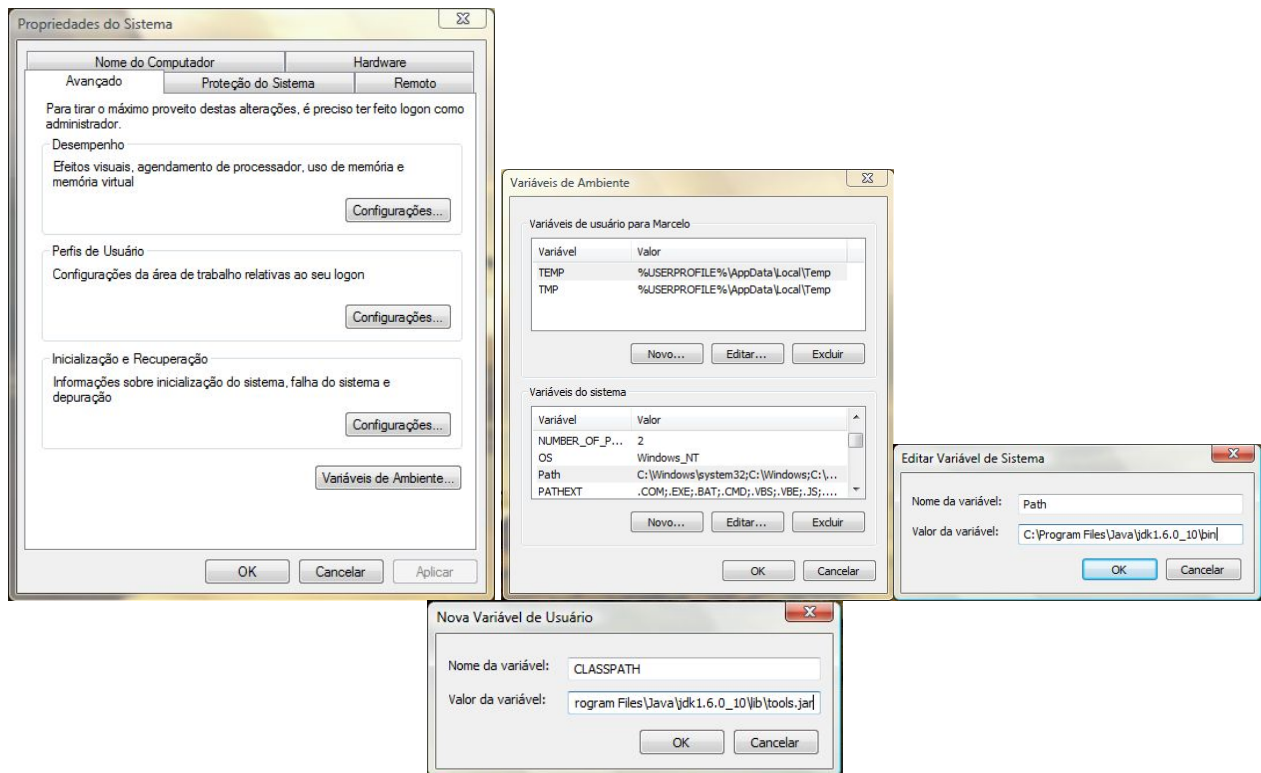
Exemplo:

- javac Soma.java – compila a classe Soma e gera o arquivo bytecode.
- Java Soma – interpreta o bytecode da classe Soma.

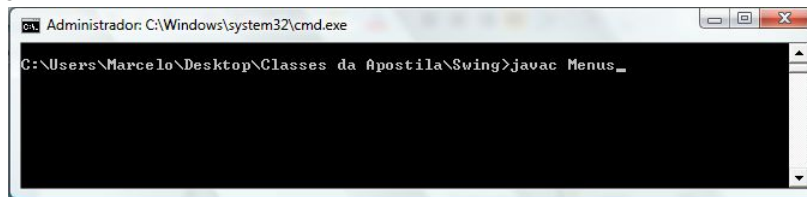
O download do Software Development Kit pode ser baixado no site da Sun (<http://java.sun.com>) e está disponível para diversos sistemas operacionais.

Depois da instalação do JDK, teremos que editar as variáveis de ambiente do computador para incluir as referências ao kit.

1. Edit a variável PATH do computador e acrescente uma referência à pasta Bin do Software Development Kit (C:\Program Files\Java\jdk1.6.0\_10\bin) se você instalou nesta pasta.
2. Edite ou crie uma variável CLASSPATH para que contenha uma referência à pasta ativa, seguida por uma referência ao arquivo tools.jar na pasta lib do kit (.;C:\Program Files\Java\jdk1.6.0\_10\lib\tools.jar). Não esqueça o ponto-e-vírgula.



O Java Software Development Kit exige o uso de uma linha de comando para compilar os programas Java, executá-los e tratar de outras tarefas. Uma linha de comando é um modo de operar um computador digitando comandos no seu teclado.



#### Dicas:

- Para abrir uma pasta, digite `cd` seguido pelo nome da pasta e pressione Enter;
  - Exemplo: `cd c:\temp`.
- Digite `cd \` para abrir a pasta raiz na unidade de disco;
- Digite `cd..` para retornar ao nível anterior;
- Para criar uma pasta, digite `md` e o nome da pasta;
  - Exemplo: `md c:\Teste`
- Para deletar a pasta, digite `rd` e o nome da pasta;
  - Exemplo: `rd c:\Teste`

Ao contrário das ferramentas de desenvolvimento Java mais sofisticadas, o Software Development Kit não inclui um editor de textos para criar códigos- fonte.

Para que um editor ou processador de textos funcione com o kit, ele precisa ser capaz de salvar arquivos de texto sem formatação. No Windows temos, por exemplo, o Bloco de Notas que só trabalha com arquivos de texto limpo. A desvantagem do uso de editores de texto simples, é que eles não exibem números de linha enquanto o código-fonte é criado.

Ver os números de linha ajuda na programação Java, pois muitos compiladores indicam o número de linhas em que ocorreu o erro.

Exemplo:

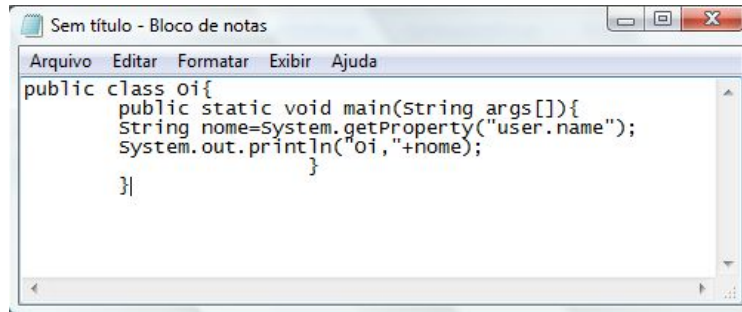
Menus.java:2: Class Font not found in type declaration

O número 2 depois do nome do arquivo fonte Java indica a linha que disparou o erro do compilador. Com um editor de textos que aceita numeração, podemos ir diretamente para essa linha e começar a procurar o erro.

Execute seu editor preferido e digite o programa Java abaixo:

Obs.: Certifique-se de que todos os parênteses, chaves e aspas na listagem sejam digitados corretamente e use maiúsculas e minúsculas no programa exatamente como aparecem.

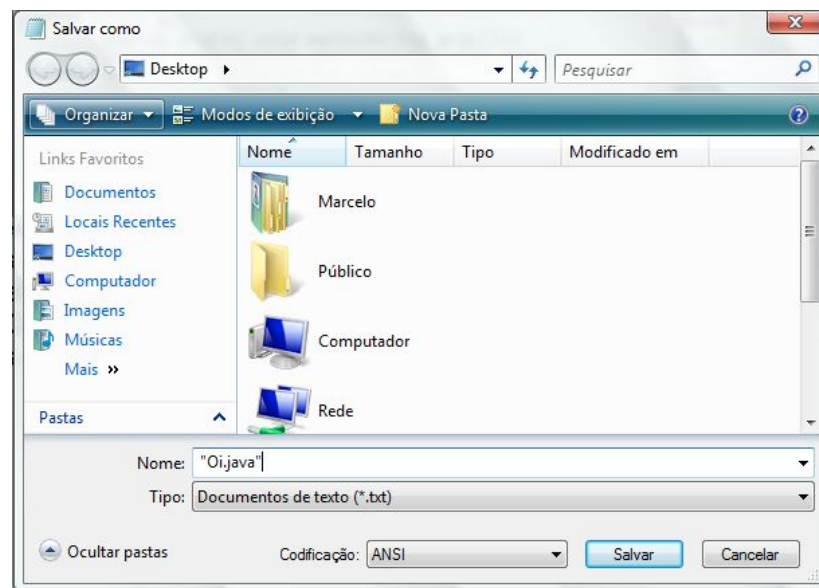
Exemplo 1: Classe Oi !



```
public class Oi{
    public static void main(String args[]){
        String nome=System.getProperty("user.name");
        System.out.println("oi,"+nome);
    }
}
```

Depois de digitar o programa, salve o arquivo em algum lugar do disco rígido com o nome Oi.java. Os códigos-fonte Java precisam ser salvos com nomes terminando em .java.

Se você estiver usando o Windows, um editor de textos como o Bloco de Notas poderá acrescentar uma extensão txt extra ao nome de arquivo de qualquer fonte Java salvo. Por exemplo, Oi.java será salvo como Oi.java.txt. Como uma alternativa para evitar este problema, coloque aspas em torno do nome de arquivo ao salvar um arquivo fonte.



Compilando e executando o programa

Abra uma janela da linha de comando, depois abra a pasta na qual o arquivo foi salvo. Quando estiver na pasta correta, poderemos compilar Oi.java digitando:

Exemplo: `javac Oi.java`

Neste momento é gerado o arquivo bytecode Oi.class.

Obs 1.: O compilador não apresenta qualquer mensagem se o programa for compilado com sucesso.

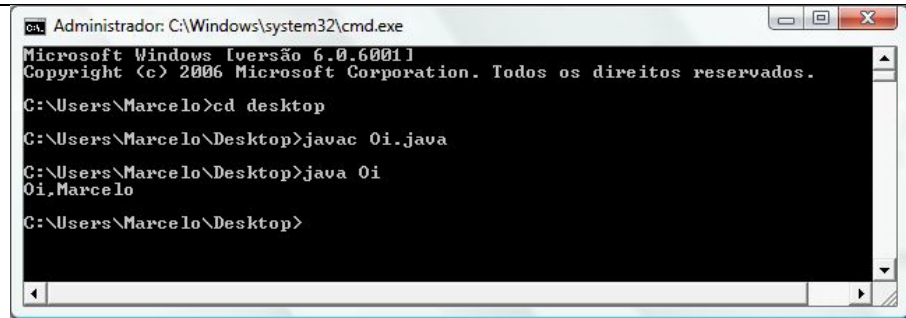
Obs 2.: Se houver problemas, o compilador lhe dirá exibindo cada erro junto com a linha que disparou o erro.

Obs 3.: Se nenhum pacote tiver sido criado o arquivo Oi.class estará na mesma pasta do arquivo Oi.java

Para executar o programa digite:

`java Oi`

Se o arquivo for interpretado corretamente pela máquina virtual Java, aparecerá a palavra Oi,<Nome do Usuário> no prompt de comando.



```
Administrator: C:\Windows\system32\cmd.exe
Microsoft Windows [versão 6.0.6001]
Copyright (c) 2006 Microsoft Corporation. Todos os direitos reservados.

C:\Users\Marcelo>cd desktop
C:\Users\Marcelo\Desktop>javac Oi.java
C:\Users\Marcelo\Desktop>java Oi
Oi, Marcelo
C:\Users\Marcelo\Desktop>
```

## Introdução à programação Java

Java é uma linguagem sensível ao caixa, ou seja, distingue maiúsculas de minúsculas: nome é diferente de Nome, por exemplo.

### Comentários

São os textos usados para anotar explicações no próprio programa e desconsiderados pelo compilador.

Símbolo	Descrição
//	Comentário de uma linha
/* */	Comentário de bloco (pode conter várias linhas)
/** /	Comentário de documentação

Obs 1.: Os comentários de documentação, colocados antes da declaração dos elementos do programa, podem ser extraídos pela ferramenta javadoc e produzir páginas de documentação em HTML.

### Operadores

#### Operadores Aritméticos

Operador	Descrição
+	Adição
-	Subtração
*	Multiplicação
/	Divisão
%	Resto da divisão

#### Operadores de Incremento e decremento

Operador	Descrição
++	Incremento unitário (adiciona uma unidade)
--	Decremento unitário (subtrai uma unidade)

Obs1.: Se usado como prefixo (antes da variável), a operação é realizada antes do uso do valor da variável.

Exemplo:

```
int a=1,b;
```

```
b=++a;// b recebe 2, a passa a valer 2.
```

Obs 2.: Se usado como sufixo (depois da variável), a operação é realizada depois do uso do valor da variável.

Exemplo:

```
int a=1,b;
```

```
b=a++;//b recebe 1, a passa a valer 2;
```

Operadores Relacionais

Operador	Descrição
==	Igual
!=	Diferente
>	Maior que
>=	Maior ou igual a
<	Menor que
<=	Menor ou igual a

Obs 1.: O operador de igualdade "==" não deve ser confundido com o operador de atribuição "=".

Obs 2.: Não podem ser usados para comparar strings (literal).

Operadores Lógicos

Operador	Descrição
&&	E lógico (and)
	Ou lógico (or)
!	Negação (not)

Operador Ternário

É um valor simples que permite usar o valor de duas expressões diferentes conforme o resultado da condição.

Sua sintaxe é:

<cond>?<expressão\_se\_verdadeiro>:<expressão\_se\_falso>

Se a expressão cond é verdadeira, é retornado o resultado de expressão\_se\_verdadeiro, senão é retornado o resultado de expressão\_se\_falso.

Operadores de atribuição compostos

Expressões assim:

variável = variável operador expressão

a=a+1 ;

Podem ser escritas assim:

variável operador = expressão

a+=1 ;

Operador	Descrição
+=	Soma e atribui
-=	Subtrai e atribui
*=	Multiplica e atribui
/=	Divide e atribui
%=	Calcula o resto e atribui



Precedência dos operadores

São as regras que determinam a ordem com que os diferentes operadores serão processados nas expressões:

Nível	Operadores
1	()
2	++ - -
3	* / %
4	+ -
5	== !=
6	&&
7	
8	?:
9	= += - + *= /= %=

Obs.: Parênteses são usados para alterar a ordem natural das expressões.

Variáveis

A declaração de variáveis em Java requer um tipo de dados, um nome (identificador) e, opcionalmente, um valor inicial.

Exemplo:

```
int a;
```

Uma constante, ou variável constante, é uma variável com um valor que nunca muda. Para declarar uma constante, use a palavra-chave final antes da declaração da variável e inclua um valor inicial para essa variável, como a seguir:

```
final Double PI=3.141592;  
final boolean DEBUG = false;  
final int voltas=25;
```

Tipos de dados primitivos:

Definem o conjunto de valores que podem ser armazenados em uma variável e também as operações sobre seus valores. A linguagem Java possui oito tipos primitivos de dados:

Tipo	Descrição	Tamanho	Intervalo
Byte	Inteiros positivos e negativos	1	-128 a + 127
Short	Inteiros positivos e negativos	2	-32.768 a +32.767
Int	Inteiros positivos e negativos	4	-2.147.483.648 a 2.147.483.647
Long	Inteiros positivos e negativos	8	-2 <sup>64</sup> a +2 <sup>64</sup> -1
Float	Valores em ponto flutuante	4	1.40239846E-45 a 3.40282347E+38
Double	Valores em ponto flutuante	8	4.9406564E-324 a 1.79769313E+308
Char	Caracteres individuais	-	-
Boolean	Tipo lógico, só assume false ou true	-	-

Obs.: String é uma classe em Java.

**Exemplo 2: Classe Oi! comentada**

```
public class Oi{  
    public static void main(String args[]){  
        System.out.println("Oi!");  
    }  
}
```

- a) **public** – é um especificador do método que indica que este é acessível externamente a esta classe (para outras classes que eventualmente seriam criadas).
- b) **class** – é a palavra reservada que marca o início da declaração de uma classe.
- c) **static** – qualificador ou “specifier”, que indica que o método deve ser compartilhado por todos os objetos que são criados a partir desta classe. Os métodos static podem ser invocados, mesmo quando não foi criado nenhum objeto para a classe.
- d) **void** – é o valor de retorno da função, quando a função não retorna nenhum valor ela retorna void, uma espécie de valor vazio que tem ser especificado.
- e) **main** – este é um nome particular de método que indica para o compilador o início do programa. É dentro deste método e através das iterações entre os atributos, variáveis e argumento visíveis nele que o programa se desenvolve.
- f) **(String args[])** – argumento do método main, ele é um vetor de strings que é formado quando são passados ou não argumentos através da invocação do nome do programa na linha de comando do sistema.
- g) **System.out.println** – chamada do método println para o atributo out da classe System, o argumento é uma constante do tipo String, para imprimir a cadeia “Oi!” e posicionar o cursor na linha abaixo.

**Exemplo 3: Operadores**

```
import javax.swing.JOptionPane;  
public class Operadores {  
    public static void main(String[] args) {  
        /* declarando e inicializando 3 variáveis inteiras */  
        int i1 = 7;  
        int i2 = 3;  
        int i3;  
        i3 = i1 + i2; // adição  
        i3 = i1 - i2; // subtração  
        i3 = i1 * i2; // multiplicação  
        i3 = i1 / i2; // divisão inteira, pois n1 e n2 são do tipo int  
        i3 = i1 % i2; // resto da divisão inteira  
        /* declarando e inicializando 3 variáveis float */  
        float f1 = 12.8f; // conversão explícita para float  
        float f2 = 6.4f;  
        float f3;  
        f3 = i1 * f2; // conversão implícita para float  
        f3 = f2 / i2; // divisão float, pois o numerador é float  
        f3++; // incremento  
        f3--; // decremento  
        f1 = ++f2 + f3; //a variável i2 será incrementada antes da atribuição  
        f1 = f2++ + f3; //a variável i2 será incrementada após a atribuição  
        /* Operador relacional */  
        System.out.println( f1 > f2 ? f1 : f2 );  
        System.out.println( "f1 = " + f1 );  
    }  
}
```

```
System.out.println( "f2 = " + f2 );
/* Calculo do preço de venda um produto baseado no preço
de compra e
    e no percentual de lucro */
float preçoCompra;
float percentualLucro;
System.out.print( "Preço de Compra : " );
preçoCompra
Float.parseFloat(JOptionPane.showInputDialog("Digite o valor de
compra"));
System.out.print("Percentual de Lucro : ");
percentualLucro
Float.parseFloat(JOptionPane.showInputDialog("Digite o percentual de
lucro"));
float lucro = preçoCompra * (percentualLucro/100);
float preçoVenda = preçoCompra + lucro;
System.out.println("Preço de Compra : " + preçoCompra +
"\nPercentual de Lucro : " +
percentualLucro +
"\nLucro : " + lucro +
"\nPreço de Venda : " +
preçoVenda);

int i = 10;
System.out.println( " i " + i );
i = i << 1;
System.out.println( " i " + i );
i = i >> 2;
System.out.println( " i " + i );
int a = 5;
int b = 10;
int c = a | b;
int d = a & b;
System.out.println( "a = " + a + "\nb = " + b +
"\nc = " + c + "\nd = " + d);
}
}
```

## JOptionPane

A classe JOptionPane oferece vários métodos que podem ser usados para criar caixas de diálogos padrão. As quatro caixas de diálogo padrão são os seguintes:

- `ConfirmDialog` – Faz uma pergunta, com botões para respostas Yes/No/Cancel.
- `InputDialog` – Pedidos de entrada de texto.
- `MessageDialog` – Apresenta uma mensagem.
- `OptionDialog` – Compreende todos os três outros tipos de caixa de diálogo.

Se configurarmos um estilo para usar com qualquer uma dessas caixas, deveremos estabelecer antes que a mesma seja aberta.

Inicialmente, estudaremos apenas as caixas de entrada e saída.

### CAIXAS DE DIÁLOGO DE ENTRADA

Uma caixa de diálogo de entrada faz uma pergunta e usa um campo de texto para armazenar a resposta.

O modo mais fácil de criar uma caixa de diálogo de entrada é com uma chamada ao método `showInputDialog(Componente, Objeto)`. Os argumentos são o componente pai e a string, o componente ou o ícone a exibir a caixa.

A chamada de método da caixa de diálogo de entrada retorna uma string que representa a resposta do usuário.

Exemplo:

```
String resposta = JOptionPane.showInputDialog(null, "Entre com o seu nome:");
```

Podemos criar uma caixa de diálogo de entrada com o método `showInputDialog (Componente, Objeto, String, int)`. Os dois últimos argumentos são:

- O título a exibir na barra de título da caixa de diálogo.
- Uma das cinco constantes de classe descrevendo o tipo de caixa de diálogo: `ERROR_MESSAGE`, `INFORMATION_MESSAGE`, `PLAIN_MESSAGE`, `QUESTION_MESSAGE`, ou `WARNING_MESSAGE`.

Exemplo:

```
String resposta = JOptionPane.showInputDialog(null, "Qual é o seu CEP?", "Entre com o CEP",  
JOptionPane.QUESTION_MESSAGE);
```

### CAIXAS DE DIÁLOGO DE MENSAGEM

É uma janela simples, que mostra informações para tanto invocamos o método `showMessageDialog (Componente, Objeto)`.

Exemplo:

```
JOptionPane.showMessageDialog(null, "O programa foi desinstalado");
```

Podemos utilizar também o método `showMessageDialog (Componente, Objeto, String, int)`.

Exemplo:

```
JOptionPane.showMessageDialog(null, "Um asteróide irá destruir a Terra.", "Alerta de destruição",  
JOptionPane.WARNING_MESSAGE);
```

---

Exemplo 4: JOptionPane- showMessageDialog e showInputDialog

---

```
import javax.swing.JOptionPane;
public class EntradaString{
    public static void main(String args[]){
        String s= JOptionPane.showInputDialog("Digite um texto");
        JOptionPane.showMessageDialog(null,s);
    }
}
```

a) import javax.swing.JOptionPane – Importação da classe gráfica(swing) JOptionPane do pacote estendido de Java(javax). Necessário para utilização da classe que não está no pacote nativo de Java.

Obs. 1: Tudo aquilo que se digita em uma janela da classe JOptionPane.showInputDialog é tratado como uma String em Java.

Obs 2.: Geralmente para utilizarmos um método em Java podemos:

- |                            |                                       |
|----------------------------|---------------------------------------|
| a) Classe.método;          | Exemplo: JOptionPane.showInputDialog. |
| b) Classe.atributo.método; | Exemplo: System.out.println           |
| c) Objeto.método.          | Exemplo: janela.setVisible(true);     |

### WRAPPER CLASS

Existem classes especiais para cada um dos tipos primitivos, sua função é dotar cada tipo com métodos para que possamos resolver problemas do tipo: Como converter um objeto String para o tipo primitivo int?

O método principal é o parseTipo. Exemplos:

```
byte a = Byte.parseByte("1");
short b=Short.parseShort("1");
int c=Integer.parseInt("1");
long d=Long.parseLong("1");
float e= Float.parseFloat("1");
double f= Double.parseDouble("1");
```

### Exemplo 5: Wrapper class

```
import javax.swing.JOptionPane;
public class Soma{
    public static void main(String args[]){
        String numero1, numero2;
        float n1, n2, resultado;
        numero1=JOptionPane.showInputDialog("Digite o primeiro número:");
        numero2=JOptionPane.showInputDialog("Digite o segundo número");
        n1=Float.parseFloat(numero1);
        n2=Float.parseFloat(numero2);
        resultado=n1+n2;
        JOptionPane.showMessageDialog(null,"Resultado:"+resultado);
    }
}
```

- 1) Em épocas de pouco dinheiro, os comerciantes estão procurando aumentar suas vendas oferecendo desconto. Faça um algoritmo que possa entrar com o valor de um produto e imprima o novo valor tendo em vista que o desconto foi de 9%.
- 2) Criar um algoritmo que efetue o cálculo do salário líquido de um professor. Os dados fornecidos serão: valor da hora aula, número de aulas dadas no mês e percentual de desconto do INSS.
- 3) Calcular e apresentar o valor do volume de uma lata de óleo, utilizando a fórmula:  
volume:  $3.14159 * r^2 * \text{altura}$ .
- 4) Efetuar o cálculo da quantidade de litros de combustível gastos em uma viagem, sabendo-se que o carro faz 12 km com um litro. Deverão ser fornecidos o tempo gasto na viagem e a velocidade média. Utilizar as seguintes fórmulas:  
distância = tempo \* velocidade  
litros usados = distancia /12

## PARTE 2 – Principais Estruturas da Linguagem Java

**ESTRUTURAS CONDICIONAIS****IF**

Um dos principais aspectos da programação é a capacidade de um programa de decidir o que ele fará. Isso é tratado por um tipo especial de instrução, chamada condicional.

Um condicional é uma instrução de programação executada apenas se uma condição específica for atendida.

A condicional mais básica é a palavra-chave `if`. A condicional `if` usa uma expressão Booleana para decidir se uma instrução deve ser executada. Se a expressão retornar um valor `true`, a instrução será executada.

**Exemplo:**

```
if(temperatura >38)
    System.out.println("Você está com febre);
```

Se quisermos que algo mais aconteça no caso em que a expressão `if` retorna um valor `false`, então uma palavra-chave `else` opcional poderá ser usada.

**Exemplo:**

```
if(temperatura>38)
    System.out.println("Você está com febre);
else
    Sytem.out.println("Sua temperatura está normal");
```

Usando `if`, é possível incluir apenas uma única instrução como código para executar se a expressão de teste for verdadeira, e outra instrução se a expressão for falsa.

Podemos testar uma variável em relação a algum valor e, se ele não combinar, testá-lo novamente com um valor diferente.

**Exemplo:**

```
if(operacao == '+')
    soma(objeto1, objeto2);
else if (operacao == '-')
    subtracao(objeto1,objeto2);
else if (operacao == '*')
    multiplicacao(objeto1,objeto2);
else if (operacao == '/')
    divicao(objeto1,objeto2);
```

Esse uso das instruções `if` é chamado instrução `if` aninhada, pois cada instrução `else` contém outro `if`, até que todos os testes possíveis tenham sido feitos.

**Lista – 2**

- 1) Ler um número e se ele for maior do que 20, então imprimir a metade do número.
- 2) Ler um número e imprimir se ele é par ou ímpar
- 3) Ler um número e imprimir se ele é positivo, negativo ou nulo.
- 4) Construir um algoritmo que leia dois números e efetue a adição. Caso o valor somado seja maior que 20, este deverá ser apresentado somando-se a ele mais 8; caso o valor somado seja menor ou igual a 20, este deverá ser apresentado subtraindo-se 5.
- 5) Entrar com um número e imprimir a raiz quadrada do número caso ele seja positivo e o quadrado do número caso ele seja negativo.
- 6) Entrar com um número e informar se ele é divisível por 3 e por 7.
- 7) Entrar com um número e informar se ele é divisível por 10, por 5, por 2 ou se não é divisível por nenhum destes.
- 8) A prefeitura do Rio de Janeiro abriu uma linha de crédito para os funcionários estatutários. O valor máximo da prestação não poderá ultrapassar 30% do salário bruto. Fazer um algoritmo que permita entrar com o salário bruto e o valor da prestação e informar se o empréstimo pode ou não ser concedido.



- 9) Entrar com nome, nota da PR1 e nota da PR2 de um aluno. Imprimir nome, nota da PR1, nota da PR2, média e uma das mensagens: Aprovado, Reprovado ou Prova Final (a média é 7 para aprovação, menor que 3 para reprovação e as demais em prova final).
- 10) Entrar com o salário de uma pessoa e imprimir o desconto do INSS segundo a tabela a seguir:
- |  |        |
|--|--------|
| menor ou igual a R\$ 600                       | isento |
| maior que R\$ 600 e menor ou igual a R\$ 1200  | 20%    |
| maior que R\$ 1200 e menor ou igual a R\$ 2000 | 25%    |
| maior que R\$ 2000                             | 30%    |
- 11) Segundo uma tabela médica, o peso ideal está relacionado com a altura e o sexo. Fazer um algoritmo que receba a altura e o sexo de uma pessoa, calcular e imprimir o seu peso ideal, utilizando as seguintes fórmulas:
- para homens -  $(72.7 * h) - 58$
  - para mulheres -  $(62.1 * h) - 44.7$
- Utilize o método `showOptionPane` da classe `JOptionPane` para dar entrada no campo de sexo

Switch

Em Java podemos agrupar ações com a instrução switch. Esta instrução é baseada em um valor da variável que pode ser qualquer um dos tipos primitivos char, byte, short ou int, que é comparada com cada um dos valores de case. Se houver uma combinação, a instrução ou as instruções após o teste serão executadas.

Se nenhuma combinação for encontrada, a instrução ou instruções default serão executadas. O fornecimento de uma instrução *default* é opcional – se ela for omitida e não houver combinação para qualquer uma das instruções case, a instrução switch será completada sem qualquer execução.

A implementação Java de switch é limitada:

- os testes e os valores só podem ser tipos primitivos simples, que podem ser convertidos para um int;
- não se pode usar tipos primitivos maiores, como um long ou float, strings ou outros objetos dentro de um switch,
- não se pode testar qualquer relacionamento além da igualdade.

Estas restrições limitam switch as casos mais simples. Ao contrário de if aninhadas funcionam para qualquer tipo de teste em qualquer tipo possível.

Exemplo:

```
switch(operation){
    case '+':
        soma(objeto1, objeto2);
        break;
    case '*':
        subtracao(objeto1, objeto2);
        break;
    case '-':
        multiplicacao(objeto1, objeto2);
        break;
    case '/':
        divisao(objeto1, objeto2);
        break;
}
```

Exemplo: Ler um número inteiro entre 1 e 12 e escrever o mês correspondente. Caso o usuário digite um número fora desse intervalo, deverá aparecer uma mensagem informando que não existe mês com este número.

```
import javax.swing.*;
public class Algoritmo138{
    public static void main(String args[]){
        int mes;
        mes=Integer.parseInt(JOptionPane.showInputDialog("Digite o
número do mês 1-12"));
        switch(mes){
            case 1:
                JOptionPane.showMessageDialog(null,"Janeiro");break;
            case 2:
                JOptionPane.showMessageDialog(null,"Fevereiro");break;
            case 3:
                JOptionPane.showMessageDialog(null,"Março");break;
            case 4:
                JOptionPane.showMessageDialog(null,"Abril");break;
            case 5:
                JOptionPane.showMessageDialog(null,"Maio");break;
```

```
case 6:
OptionPane.showMessageDialog(null,"Junho");break;
case 7:
OptionPane.showMessageDialog(null,"Julho");break;
case 8:
OptionPane.showMessageDialog(null,"Agosto");break;
case 9:
OptionPane.showMessageDialog(null,"Setembro");break;
case 10:
OptionPane.showMessageDialog(null,"Outubro");break;
case 11:
OptionPane.showMessageDialog(null,"Novembro");break;
case 12:
OptionPane.showMessageDialog(null,"Dezembro");break;
default: JOptionPane.showMessageDialog(null,"Não
existe mês correspondente para este número");
}

}

}
```

## ESTRUTURAS DE REPETIÇÃO

### FOR

Um loop for é usado para repetir uma instrução até que uma condição seja atendida. Embora os loops for sejam constantemente utilizados para a iteração simples, em que uma instrução é repetida por um certo número de vezes, os loops for podem ser usados para quase qualquer tipo de loop.

O loop for em Java possui o seguinte formato:

```
for(inicialização; condição; incremento){
instrução;
}
```

O início do loop for possui três partes:

- Inicialização- expressão que inicializa o início do loop. Se você tiver um índice de loop, essa expressão poderia declará-lo e inicializá-lo, como em `int i=0`. As variáveis declaradas nessa parte do loop for são locais ao próprio loop; elas deixam de existir depois que o loop terminar sua execução. Podemos inicializar mais de uma variável nesta seção, separando cada expressão com uma vírgula. A instrução `int i=0, int j=10` nesta seção declararia as variáveis `i` e `j`, e ambas seriam locais ao loop.
- Condição – teste que ocorre de cada passada do loop. O teste precisa ser uma expressão Booleana ou uma função que retorne um valor boolean, como `i<50`. Se a condição for true, o loop é executado. Quando a condição for false, o loop deixa de ser executado.
- Incremento – é qualquer expressão ou chamada de método. Normalmente, o incremento é usado para alterar o valor do índice do loop, para trazer o estado do loop para mais perto de retornar false e encerrar o loop. O incremento ocorre depois de cada passada do loop. Pode-se colocar mais de uma expressão nesta seção, separando cada expressão com uma vírgula.

Exemplo:

```
for(int i=0; i<10;i++)
    System.out.println("Número: "+i);
```

Imprime todos os números de zero a dez.

Obs.: qualquer parte do loop for pode ser uma instrução vazia; em outras palavras, você pode incluir um ponto-e-vírgula sem expressão ou instrução, e essa parte do loop for será ignorada.

Obs.: se utilizada uma instrução vazia no loop for, teremos que inicializar ou incrementar quaisquer variáveis de loop ou índices de loop em qualquer outro lugar no programa.

Podemos ter uma instrução vazia como corpo do seu loop for, se tudo o que quiser fazer estiver na primeira linha desse loop.

Exemplo:

```
for(i=4001;notPrime(i);i+=2);
```

Encontra o primeiro número primo maior do que 4000. Emprega um método chamado notPrime(), que retorna um valor Booleano, que deverá indicar se i é primo ou não.

### WHILE

É usado para repetir uma instrução se uma determinada condição for true.

Exemplo:

```
while(i<10){  
x=x*i++;  
}
```

A condição do while é uma expressão Booleana. Se a expressão for true, o loop while executará o corpo do loop e depois testará a condição novamente. Esse processo se repete até que a condição seja false.

Embora o loop apresentado utilize chaves de abertura e fechamento para tornar uma instrução em bloco, as chaves não são necessárias, pois o loop contém apenas uma instrução: x=x\*i++. O uso das chaves não cria quaisquer problemas, e as chaves serão exigidas se você acrescentar outra instrução dentro do loop depois.

### DO WHILE

O loop do é exatamente como um loop while com uma diferença principal – o lugar no loop em que a condição é testada.

Um loop while testa a condição antes do looping, de modo que, se a condição for false na primeira vez que for testada, o corpo do loop nunca será executado.

Um loop do executa o corpo do loop pelo menos uma vez antes de testar a condição, de modo que, se a condição for false na primeira vez em que for testada, o corpo do loop já teria sido executado uma vez.

Exemplo:

```
do{  
i*=2;  
System.out.println(i+ " ");  
}while (i<100);
```

### BREAK E CONTINUE

O comando break é usado para interromper a execução de um dos laços de iteração vistos acima ou de um comando switch. Este comando é comumente utilizado para produzir a parada de um laço mediante a ocorrência de alguma condição específica, antes da chegada do final natural do laço.

Exemplo:

```
// Achar i tal que v[i] é negativo  
for(i=0; i<n;i++)  
    if(v[i] <0) break;  
if(i == n)  
    system.out.println("elemento negativo não encontrado.");
```

E se isto se der dentro de um laço duplo? Nesse caso, o comando break provocará a interrupção apenas do laço em que o comando é imediatamente subjacente. Os outros laços continuam normalmente. O comando continue tem a função de pular direto para final do laço, mas em vez de interromper o laço como no break, ele continua executando o próximo passo do laço. O uso de continue é pouco usual na programação estruturada.



**Lista - 4****Estruturas repetição(while e do while)**

- 1) Entrar com números e imprimir o triplo de cada número. O algoritmo acaba quando entrar com o número -999
- 2) Entrar com números enquanto forem positivos e imprimir quantos números foram digitados.
- 3) Entrar com números e imprimir o quadrado de cada número até entrar um número múltiplo de 6 que deverá ter seu quadrado também impresso.
- 4) Chico tem 1,50 e cresce 2 centímetros por ano, enquanto Juca tem 1,10m e cresce 7 centímetros por ano. Construir um algoritmo que calcule e imprima quantos anos serão necessários para que Juca seja maior que Chico
- 5) Crie um algoritmo que entre com vários números inteiros e positivos e imprima a média dos números múltiplos de 3.
- 6) Uma das maneiras de se conseguir a raiz quadrada de um número é subtrair do número os ímpares consecutivos a partir de 1, até que o resultado da subtração seja menor ou igual a zero. O número de vezes que se conseguir fazer a subtração é a raiz quadrada exata(resultado 0) ou aproximada do número(resultado negativo).

Exemplo: Raiz de 16

$16-1=15-3=12-5=7-7=0 \rightarrow 4$

## PARTE 3 – Funções Matemáticas, Funções de String, Formatação de números e de data/hora

**FUNÇÕES MATEMÁTICAS**

A linguagem Java possui uma classe com diversos métodos especializados em realizar cálculos matemáticos. Para realizar esses cálculos, são utilizados os métodos da classe Math que devem apresentar a seguinte sintaxe:

- Math.<nome do método>(argumentos ou lista de argumentos)

Não é necessário importar a classe Math, pois o mesmo faz parte do pacote java.lang

A classe Math define duas constantes matemáticas:

- Math.PI – valor de pi (3,14159265358979323846)
- Math.E – logaritmos naturais (2.7182818284590452354)

**Método ceil**

Tem como função realizar o arredondamento de um número do tipo double para o seu próximo inteiro. Sua sintaxe é a seguinte:

- Math.ceil(<valor do tipo double>)

**Método floor**

É utilizado para arredondar um determinado número, mas para seu inteiro anterior. Sua sintaxe é:

- Math.floor(<valor do tipo double>);

**Método max**

Utilizado para verificar o maior valor entre dois números, que podem ser do tipo double, float, int ou long. A sua sintaxe é a seguinte:

- Math.max(<valor1>,<valor2>);

**Método min**

Fornece o resultado contrário do método max, sendo então utilizado para obter o valor mínimo entre dois números. Do mesmo modo que o método max, esses números também podem ser do tipo double, float, int ou long. A sua sintaxe é a mesma do método max mudando apenas para Math.min

**Método sqrt**

Utilizado quando há necessidade de calcular a raiz quadrada de um determinado número. O número que se deseja extrair a raiz deve ser do tipo double. Veja sua sintaxe:

- Math.sqrt(<valor do tipo double>);

**Método pow**

Assim como é possível extrair a raiz quadrada de um número, também é possível fazer a operação inversa, ou seja, elevar um determinado número ao quadrado ou a qualquer outro valor de potência. Os números utilizados deverão ser do tipo double. Sua sintaxe é a seguinte:

- Math.pow(<valor da base>,<valor da potência>);

**Método random**

É utilizado para gerar valores de forma aleatória. Toda vez que o método random é chamado, será sorteado um valor do tipo double entre 0.0 e 1.0 (o valor 1 nunca é sorteado). Nem sempre essa faixa de valores é suficiente numa aplicação real.

Exemplo:

- (int) (Math.random()\*100)

Com isso seriam gerados números inteiros entre 0 e 99

**FUNÇÕES COM STRING**



### Método length

O método length é utilizado para retornar o tamanho de uma determinada string, incluindo também os espaços em branco presentes nela. Esse método retorna sempre um valor do tipo int. Veja sua sintaxe:

- `<String>.length();`

### Método charAt

Usado para retornar um caractere de uma determinada string de acordo com um índice especificado entre parênteses. Esse índice refere-se à posição do caractere na string, sendo 0 o índice do primeiro caractere. Sintaxe do método charAt é a seguinte:

- `<String>.charAt(<índice>);`

### Métodos toUpperCase e toLowerCase

São utilizados para transformar todas as letras de uma determinada string em maiúsculas ou minúsculas.

- O método toUpperCase transforma todos os caracteres de uma string em maiúsculos
- O método toLowerCase transforma todos os caracteres de uma string em minúsculos
- Sua sintaxe é a seguinte:
  - `<String>.toUpperCase()` ou `<String>.toLowerCase()`

### Método substring

Retorna um cópia de caracteres de uma string a partir de dois índices inteiros especificados, funcionando basicamente da mesma forma que o método charAt dentro de um looping.

- A sintaxe da substring é a seguinte:
  - `<String>.substring(<índice inicial>,[<índice final>]`

### Método trim

Seu objetivo é remover todos os espaços em branco que aparecem no início e no final de uma determinada string. Serão removidos apenas os espaços do início e do fim da string; não serão removidos os espaços entre as palavras. Sua sintaxe é a seguinte:

- `<String>.trim();`

### Método replace

É utilizado para substituição de caracteres, ou grupo de caracteres, em uma determinada string. Para seu funcionamento é necessário informar o(s) caractere(s) que deseja(m) substituir e por qual(is) caractere(s) ele será(ão) substituído(s). Caso não haja na string nenhuma ocorrência do caractere a ser substituído, a string original é retornada, isto é, não ocorre nenhuma alteração. Veja sua sintaxe:

- `<String>.replace(<caracteres a serem substituídos>, <substituição>)`

### Método valueOf

É usado para converter diversos tipos de dados em strings. Esse método aceita vários tipos de argumento (números ou cadeia de caracteres) e transforma-os em strings. Sintaxe:

- `String.valueOf(<nome da variável a ser convertida>)`

### Método indexOf

É usado para localizar caracteres ou substrings em uma String. Quando realizamos a busca de uma palavra em um texto, estamos usando algo parecido com o funcionamento de indexOf, isto é, ele busca uma palavra e retorna a posição onde ela se encontra. Caso haja sucesso na busca, é retornado um número inteiro referente a posição do texto onde o caractere foi encontrado, ou a posição do texto onde se inicia a substring localizada. Caso haja insucesso na busca é retornado o valor inteiro -1. A sintaxe é:

- `String.indexOf(<caractere ou substring a ser localizada, [posição inicial]>)`

## FORMATAÇÃO COM A CLASSE DECIMALFORMAT

Os cálculos matemáticos, em especial os que envolvem multiplicação e divisão, podem gerar resultados com muitas casas decimais. Isso nem sempre é necessário e esteticamente correto, pois apresentar um resultado com muitas casas decimais não é muito agradável e legível à maioria dos usuários. Por exemplo: considere duas variáveis do tipo `double` `x=1` e `y=6`. Ao realizar a divisão de `x` por `y`, aparece na tela o resultado `0.16666666666666666`. Esse resultado não é o mais adequado para se apresentar. Seria mais conveniente mostrar o resultado com duas ou três casas decimais.

Para realizar a formatação, é necessário definir um modelo de formatação, conhecido pelo nome de `pattern`. Considere `pattern` como o estilo de formatação que será apresentado sobre um valor numérico. Para definir o `pattern`, são usados caracteres especiais

Caractere	Significado
0	Imprime o dígito normalmente, ou caso ele não exista, coloca 0 em seu lugar. Exemplo: Seja as variáveis <code>int x=4, y=32</code> e <code>z=154</code> , ao usar o <code>pattern "000"</code> , o resultado impresso na tela seria <code>x→004, y→032</code> e <code>z→154</code>
#	Imprime o dígito normalmente, desprezando os zeros à esquerda do número. Exemplo: Sejam as variáveis <code>double x=0,4</code> e <code>y=01.34</code> , ao usar o <code>pattern "##.##"</code> , o resultado impresso na tela seria <code>x→.4, y→1.34</code>
.	Separador decimal ou separador decimal monetário
-	Sinal de número negativo

### CLASSE LOCALE

A linguagem Java tem como característica ser utilizada no mundo todo. Em função disso, um mesmo software feito em Java pode ser utilizado por usuários espalhados pelo globo. Cada país ou região adota certos formatos para representação monetária, apresentação de datas, etc. esses formatos são definidos pelo sistema operacional da máquina e ficam armazenados como configurações locais. O separador de casas decimais, por exemplo, pode ser um ponto ou uma vírgula, dependendo da região.

A classe `Locale` permite identificar certas propriedades da máquina em que o software está sendo executado.

#### Lista – 5

#### Funções Matemáticas e de String – Formatação com a classe `DecimalFormat`

- 1) Crie uma classe para fazer o arredondamento dos seguintes valores: 5.2, 5.6 e -5.8 para o valor inteiro mais próximo.
- 2) Crie uma classe para fazer o arredondamento dos seguintes valores: 5.2, 5.6 e -5.8 para o valor inteiro anterior.
- 3) Crie uma classe para calcular o maior entre dois números.
- 4) Crie uma classe para calcular o menor entre dois números.
- 5) Crie uma classe para calcular a raiz quadrada dos números 900 e 30.25.
- 6) Crie uma classe para calcular a potência de 5.5 elevado a 2 e 25 elevado a 0.5.
- 7) Crie uma classe que gere 5 cartões de loteria com seis números em cada um.
- 8) Formate os valores como se pede:
  - o Idade 38 para 038;
  - o Quantidade 9750 para 9.750;
  - o Estoque 198564 para 198.564
  - o Altura 1,74f para 1,74
  - o Peso 7025 para 70,25
  - o Valor 2583.75 para R\$ 2.583,75

9) Crie uma classe utilize os seguintes métodos existentes da classe Locale:

- `getCountry();`
- `getDisplayCountry();`
- `getDisplayLanguage();`
- `getDisplayName();`

10) Crie uma classe para obter o tamanho da frase: "Aprendendo Java".

11) Crie uma classe para obter os caracteres de 11 a 14 da frase: "Aprendendo Java".

12) Coloque a palavra ARROZ em minúscula, a palavra batata em maiúscula e a palavra SaLaDa em minúscula.

13) Crie uma classe que retorne trechos de caracteres definidos por intervalos pré-determinados.

14) Crie um programa que retire espaços em branco antes e depois de uma frase digitada pelo usuário.

15) Crie uma classe que utilize um método para substituir o caracter a por u e o carater n por N e trocar os espaços em branco por \_ de uma frase digitada pelo usuário.

16) Crie uma classe que converta valores números para string.

17) Crie uma classe que realize uma busca pelo caractere a.

### DATA/HORA

Os recursos de data e hora devem ser suficientemente flexíveis. O uso de datas e horas torna possível a criação de páginas que exibem informações de maneira dinâmica. Existem 11 classes diferentes para manipulação de datas e horas.

As classes disponíveis para a manipulação de data e hora pertencem a três pacotes diferentes

- `java.util` – `Date`, `Calendar`, `GregorianCalendar`, `TimeZone`, `SimpleTimeZone`
- `java.text` – `DateFormat`, `SimpleDateFormat`, `FormatSymbols`
- `java.sql` – `Date`, `Time`, `Timestamp`

A classe `Date` existe em dois pacotes (`util` e `sql`), ambos com características e comportamentos diferentes

A diferença básica entre as classes `Date`, `DateFormat`, `SimpleDateFormat` e `Calendar` é a seguinte:

- `Date` (pacote `util`) representa um instante de tempo, sem levar em consideração sua representação ou localização geográfica, com precisão de milissegundos
- `DateFormat` representa um data com formato `String` de acordo com um determinado fuso horário e calendário
- `SimpleDateFormat` permite a especificação de diferentes formatos para a data
- `Calendar` representar um instante de tempo de acordo com um sistema particular de calendário e fuso horário

### CLASSE DATE

Para utilizar uma classe externa, é necessário que ela esteja na mesma pasta da aplicação ou fazemos sua importação

- `import java.util.Date;`

O compilador compreende que data será um objeto declarado a partir da classe `Date`

- `Date data = new Date();`

Essa declaração indica que o objeto `data` será inicializado com a data e hora atuais do sistema(default)

Para marcar o tempo, Java considera o número de milissegundos decorridos desde 1º de janeiro de 1970.

Cada segundo possui 1.000 milissegundos, cada minuto possui 60 segundos, cada hora possui 60 minutos e cada dia possui 24 horas, para saber o correspondente em dias, basta multiplicar  $1000 \times 60 \times 60 \times 24$

`getTime()` – Esse método retorna um inteiro do tipo `long` que permite representar milissegundos decorridos

O uso de `getTime()` permite realizar o cálculo entre datas, bastando calcular a diferença entre os milissegundos

### CLASSE DATEFORMAT

A classe Date não fornece um mecanismo de controle sobre a formatação de uma data e não permite converter uma string contendo informações sobre uma data em um objeto Date

A classe DateFormat permite apresentar a data com diferentes formatações, dependendo das necessidades de utilização, tornando sua visualização mais agradável aos usuários

A classe DateFormat pode criar uma data a partir de uma string fornecida

Ao criar um objeto a partir de uma classe DateFormat, ele conterá informação a respeito de um formato particular no qual a data será apresentada

O método getDateInstance tem a seguinte sintaxe:

- getDateInstance( int estilo)

Ao invocar o método, deve ser passado um número inteiro que define o estilo de formatação

Métodos mais utilizados da classe DateFormat:

- Format(Date d) – formata a data em uma string de acordo com o estilo utilizado. Retorna uma String
- getInstance() – Retorna uma data e hora de acordo com o estilo SHORT. Retorna um DateFormat
- getDateInstance() – Retorna uma data de acordo com o estilo de formatação local. Retorna um DateFormat
- getTimeInstance() – Retorna um horário de acordo com o estilo de formatação local. Retorna um DateFormat
- parse(String s) – Converte a string em tipo Date. Retorna um Date

#### CLASSE SIMPLEDATEFORMAT

Permite criar formatos alternativos para a formatação de datas e horas

Em orientação a objeto dizemos que SimpleDateFormat extends DateFormat

Deve-se recorrer ao uso de um pattern para criar o próprio formato de data/hora

Principais letras para criação de patterns:

- G – designador de era. Formato texto. Exemplo: AD
- Y – ano. Formato Year. Exemplo: 2005;05
- M – mês do ano. Formato Month. Exemplo: Jul;07
- W – semana do ano. Formato Number. Exemplo: 15
- W – semana do mês. Formato Number. Exemplo: 3
- D – dia do ano. Formato Number. Ex.: 234
- D – dia do mês. Formato Number. Ex.: 5
- F – dia da semana no mês. Formato Number. Ex.: 2
- E – dia da semana. Formato Text. Ex.: Sex
- A – am/pm. Formato Texto. Ex.: PM
- H – hora do dia(0-23). Formato Number. Ex.: 0
- K – hora do dia(1-24). Formato Number. Ex.: 23
- K – hora em am/pm(0-11). Formato Number. Ex.: 2
- H – hora em am/pm(1-12). Formato Number. Ex.: 5
- M – minuto da hora. Formato Number. Ex.: 10
- S – segundo do minuto. Formato Number. Ex.: 30
- S – milissegundos. Formato Number. Ex.: 978

Métodos mais utilizados da classe SimpleDateFormat

- applyPattern(String p) – Aplica um pattern à data conforme definido na String p. Retorna void
- toPattern() – Fornece o pattern que está sendo usado no formato de data. Retorna uma String

#### CLASSE CALENDAR

Oferece mecanismos adequados para realização de cálculos com datas ou para identificação das propriedades de uma data.

A classe Calendar converte um tipo Date armazenado nela em um série de campos

Possui métodos para recuperar(get) ou para armazenar(set) os valores correspondentes a datas e horas, através de um argumento fornecido que identifica o campo a ser manipulado.

Exemplo: Recuperar o dia do ano.

- get(Calendar.YEAR)

## Principais campos usados pela classe Calendar:

- DAY\_OF\_MONTH – Dia do mês(1-31)
- DAY\_OF\_WEEK – Dia da semana(0=domingo, 6=sábado)
- DAY\_OF\_WEEK\_IN\_MONTH – Semana do mês(1-5) corrente. Diferente em relação a WEEK\_OF\_MONTH porque considera apenas a semana cheia
- DAY\_OF\_YEAR – Dias decorridos no ano corrente
- HOUR – Hora do dia(manhã ou tarde)(0 a 11)
- HOUR\_OF\_DAY – Hora do dia(0 a 23)
- MILLISECOND – Milissegundos em relação aos segundos
- MINUTE- Minutos em relação à hora corrente
- MONTH – Mês em relação ao ano corrente
- SECOND – Segundos em relação ao minuto corrente
- WEEK\_OF\_MONTH – Semana em relação ao minuto corrente (1 a 5)
- WEEK\_OF\_YEAR – Semana em relação ao ano corrente
- YEAR – Ano corrente
- JANUARY, FEBRUARY, MARCH, APRIL, MAY, JUNE, JULY, AUGUST, SEPTEMBER, OCTOBER, NOVEMBER, DECEMBER – Mês correspondente ao ano
- MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY – Dia correspondente à semana

## Resumo dos métodos mais utilizados da classe Calendar:

- Add(int field, int valor) – Função aritmética para objetos Date que adiciona o valor inteiro ao campo (Field) determinado. Retorna void
- after(Object x) – Verifica se o tempo (data e hora) do objeto x (pode ser Calendar ou outro tipo) é superior ao armazenado no objeto Calendar. Retorna boolean
- before(Object x) – Idem anterior, porém verifica se o tempo é anterior ao objeto
- clear() – Zera o conteúdo de data e hora armazenando 1º de janeiro de 1970 00:00:00. Retorna void
- getFirstDayOfWeek() – Fornece o primeiro dia da semana, dependendo da localidade
- getTime() – Fornece o tempo corrente
- getTimeMillis() – Fornece o tempo corrente em milissegundos
- roll() – Função aritmética para objetos Date aplicando o efeito de rolagem de datas. Retorna void
- toString() – Fornece uma representação em formato String para a data armazenada. Retorna String

## Lista – 5

- 1) Crie uma classe que imprima um objeto data em uma JOptionPane.
- 2) Execute o seguinte exercício que utiliza a seguinte informação: Java considera o número de milissegundos decorridos desde 1º de janeiro de 1970. Cada segundo possui 1.000 milissegundos, cada minuto possui 60 segundos, cada hora possui 60 minutos e cada dia possui 24 horas, para saber o correspondente em dias, basta multiplicar 1000 x 60 x 60 x 24.
- 3) Crie uma classe que utilize os seguintes métodos:
  - getHours(); getMinutes(); getSeconds(); getTime(); getTimezoneOffset().Altere os valores da hora, dos minutos e dos segundos com seus respectivos métodos.
- 4) Crie uma classe que formate as datas conforme os exemplos abaixo:
  - Formato Default = 18/05/2009
  - Formato SHORT = 18/05/09
  - Formato MEDIUM = 18/05/2009
  - Formato LONG = 18 de Maio de 2009
  - Formato FULL = Segunda-feira, 18 de Maio de 2009
- 5) Crie uma classe que formate a data conforme os exemplos abaixo:
  - dia/mês/ano com quatro dígitos, horas:minutos:segundos
  - dd/MM/yyyy, hh:mm:ss -->18/05/2009, 10:30:29

- Dia da semana, mês por extenso, dia, ano com dois dígitos
  - `EEE, MMM d, "yy-->Seg, Mai 18,' 09`
- 6) Crie uma classe que se utilize dos métodos da classe Calendar.
- 7) Crie uma classe que pegue a hora do sistema e imprima Bom dia, Boa tarde, Boa noite ou Boa madrugada conforme a hora capturada do sistema.
- 8) Execute a classe.

## PARTE 4 – Introdução à exceções em Java

### EXCEÇÕES

Exceções são objetos que representam erros que podem ocorrer em um programa Java.

Java define uma série de recursos da linguagem para lidar com exceções, incluindo os seguintes:

- Como tratar de exceções no código e recuperar-se elegantemente de problemas em potencial;
- Como dizer à Java e aos usuários dos métodos que você está esperando uma exceção em potencial;
- Como criar uma exceção se você a detectar;
- Como o seu código é limitado, embora tornado-se mais robusto pelas exceções.

As exceções em Java são instâncias de classes que herdam da classe Throwable. Uma instância de uma classe Throwable é criada quando uma exceção é lançada.

Throwable possui duas subclasses: Error e Exception

- As instâncias de Error são erros internos envolvendo a máquina virtual Java (o ambiente runtime). Esses erros são raros e normalmente fatais.
- A classe Exception caem em dois grupos gerais:
  - Exceções de runtime (subclasses da classe RuntimeException), como ArrayIndexOutOfBoundsException, SecurityException e NullPointerException.
  - Outras exceções, como EOFException e MalformedURLException

As exceções de runtime normalmente ocorrem por causa do código.

Uma ArrayIndexOutOfBoundsException nunca deveria ser lançada se verificássemos os limites de um array.

Uma NullPointerException não acontecerá a menos que tentássemos usar uma variável antes que ela tenha sido configurada para conter um objeto.

Uma EOFException acontece, por exemplo, quando se está lendo um arquivo e o mesmo termina de forma inesperada.

Uma MalformedURLException acontece quando uma URL não está no formato correto.

A maioria das classes de exceção faz parte do pacote java.lang (incluindo Throwable, Exception e RuntimeException).

Muitos dos outros pacotes definem outras exceções, e essas são usadas por toda a biblioteca de classes.

Em muitos casos, o compilador Java impõe o gerenciamento de exceção quando tentamos usar métodos que utilizam exceções, então precisamos lidar com essas exceções no código ou ele simplesmente não será compilado.

- Exemplo de um erro:
  - XMLParser.java:32:Exception java.lang.InterruptedException must be caught or it must be declared in the throws clause of this method
  - No erro acima é preciso pegar a exceção java.lang.InterruptedException ou então declará-la na cláusula throws deste método.

Um método pode indicar os tipos de erros que possivelmente poderia lançar. Por exemplo, métodos que lêem de arquivos poderiam potencialmente gerar erros IOException.

Pegar exceções em potencial exige:

- Que protejamos o código que contém o método que pode lançar uma exceção dentro de um bloco try
- Que coloquemos a exceção dentro de um bloco catch

#### TRY-CATCH

Significa: "Experimente este trecho de código, que poderia causar uma exceção. Se ele for executado corretamente, prossiga o programa. Se o código lançar uma exceção, apanhe-a e trate dela."

Exemplo:

```
try{
    float in = Float.parseFloat(input);
}
catch (NumberFormatException nfe){
    System.out.println(input+ " is not a valid number.");
}
```

Na instrução acima o método de classe Float.parseFloat() poderia potencialmente lançar uma exceção do tipo NumberFormatException, significando que o thread foi interrompido por algum motivo



Para tratar dessa exceção, a chamada a `parseFloat()` é colocada dentro de um bloco `try`, e um bloco `catch` associado foi configurado. Esse bloco `catch` recebe quaisquer objetos `NumberFormatException` que sejam lançados dentro do bloco `try`.

A parte da cláusula `catch` dentro dos parênteses é semelhante à lista de argumentos da definição de um método. Ela contém a classe da exceção que devemos pegar e um nome de variável. Podemos usar a variável para nos referirmos a esse objeto dentro de um bloco `catch`.

- `getMessage()` está presente em todas as exceções e apresenta uma mensagem detalhada do erro
- `printStackTrace()` apresenta a sequência de chamadas de método que levou à instrução que gerou a exceção.

Exemplo:

```
try{
    float in = Float.parseFloat(input);
}
catch (NumberFormatException nfe){
    System.out.println("Oops:" + nfe.getMessage());
}
```

Pegando exceções de uma superclasse, por exemplo, `IOException`, também pegamos instâncias de suas subclasses: `EOFException` e `FileNotFoundException`.

Para pegar várias exceções que não estejam relacionadas por herança podemos usar vários blocos `catch` para um único `try`.

- Exemplo:

```
try{
} catch (IOException ioe){
    System.out.println("Input/output error");
    System.out.println(ioe.getMessage());
} catch (ClassNotFoundException cnfe){
    System.out.println("Class not found");
    System.out.println(cnfe.getMessage());
} catch (InterruptedException ie){
    System.out.println("Program interrupted");
    System.out.println(ie.getMessage()); }
```

#### CLÁUSULA FINALLY

Utilizada quando desejamos realizar uma ação não importando o que aconteça.

Server para liberar algum recurso externo depois de adquiri-lo

- Exemplo:

```
try{
    readTextfile();
} catch (IOException e){
    System.out.println("Input/output error");
    System.out.println(e.getMessage());
} finally{
    closeTextfile(); }
```

Exemplo 1: Classe que gera erro de divisão por zero.

```
import javax.swing.JOptionPane;
class Excecoes{
    public static void main (String args[]) {
        int x=10, y=0, z=0;
        try{
            z = x / y; }
        catch(Exception e){
            JOptionPane.showMessageDialog(null,e.getMessage()); }
```

```
}  
}
```

**Exemplo 2:**

```
class HexRead {  
    String[] input = { "000A110D1D260219 ", "78700F1318141E0C ",  
"6A197D45B0FFFFFF " };  
    public static void main(String[] arguments) {  
        HexRead hex = new HexRead();  
        for (int i = 0; i < hex.input.length; i++)  
            hex.readLine(hex.input[i]);  
    }  
    void readLine(String code) {  
        try {  
            for (int j = 0; j + 1 < code.length(); j += 2) {  
                String sub = code.substring(j, j+2);  
                int num = Integer.parseInt(sub, 16);  
                if (num == 255)  
                    return;  
                System.out.print(num + " ");  
            }  
        } finally {  
            System.out.println("***");  
        }  
        return;  
    }  
}
```

**Exemplo 3:**

```
public class CalorieCounter {  
    float count;  
    public CalorieCounter(float calories, float fat, float fiber) {  
        if (fiber > 4) {  
            fiber = 4;  
        }  
        count = (calories / 50) + (fat / 12) - (fiber / 5);  
        assert count > 0 : "Adjusted calories < 0";  
    }  
    public static void main(String[] arguments) {  
        if (arguments.length < 2) {  
            System.out.println("Usage: java CalorieCounter calories fat  
fiber");  
            System.exit(-1);  
        }  
        try {  
            int calories = Integer.parseInt(arguments[0]);  
            int fat = Integer.parseInt(arguments[1]);  
            int fiber = Integer.parseInt(arguments[2]);  
            CalorieCounter diet = new CalorieCounter(calories, fat,  
fiber);  
            System.out.println("Adjusted calories: " + diet.count);  
        } catch (NumberFormatException nfe) {  
            System.out.println("All arguments must be numeric.");  
            System.exit(-1);  
        }  
    }  
}
```

## PARTE 5 – Array e Matrizes

Os arrays são uma maneira de armazenar uma lista de itens que possuem o mesmo tipo de dado primitivo, a mesma classe ou uma classe pai comum. Cada item na lista entra em seu próprio local numerado, para acessarmos a informação com facilidade.

Os arrays podem conter qualquer tipo de informação armazenada em uma variável, mas, quando o array é criado, podemos usá-lo apenas para esse tipo de informação. Por exemplo, pode-se ter um array de inteiros, um array de objetos String ou um array de arrays, mas não pode ter um array que contenha objetos String e inteiros.

Para criar um array em Java, precisa-se:

1. Declarar uma variável para manter o array.
2. Criar um novo objeto de array e atribuí-lo à variável de array.
3. Armazenar informações neste array.

O primeiro passo na criação do array é declarar uma variável que o manterá. As variáveis de array indicam o objeto ou tipo de dado que o array manterá e o nome do array. Para diferenciar das declarações de variável normal, um par de colchetes vazio([]) é acrescentado ao objeto ou tipo de dado, ou ao nome da variável.

Exemplo:

```
String [] nomes;  
float[] números;
```

Obs.: também podemos declarar um array colocando os colchetes após o nome da variável, em vez do tipo de informação.

Exemplo:

```
String nomes[];
```

Depois de declarar a variável de array, o próximo passo é criar um objeto de array e atribuí-lo a essa variável.

Para fazer isso:

- Use o operador new;
- Inicialize o conteúdo do array diretamente.

Como os arrays são objetos em Java, podemos usar o operador new para criar uma nova instância de um array.

Exemplo:

```
String[] jogador = new String[10];
```

Quando criamos um objeto de array usando new, todos os seus slots recebem automaticamente um valor inicial (0 para arrays numéricos, false para Booleanos, '\0' para arrays de caractere, e null para objetos).

Exemplo:

```
Integer[] serie = new Integer[3];  
series[0]= new Integer(10);  
series[1]= new Integer(3);  
series[2]= new Integer(5);
```

Podemos criar e inicializar um array ao mesmo tempo, delimitando os elementos do array entre chaves, separados por vírgulas.

Exemplo:

```
Ponto[] marcados={new Ponto(1,5), new Ponto(3,3), new Ponto(2,3),};
```

Obs.: Cada um dos elementos dentro das chaves precisa ter o mesmo tipo da variável que mantém o array.

Obs.: Quando criamos um array com valores iniciais, o tamanho será definido com o número de elementos incluídos dentro das chaves.

Exemplo:

```
String[] num={"Um", "Dois", "Três", "Quatro", "Cinco", "Seis"};
```

Obs.: todos os arrays possuem uma variável de instância chamada length que contém o número de elementos no array.

Obs.: O primeiro elemento de um array está na posição zero. Para acessar um elemento devemos acessar a posição do mesmo.

Exemplo:

```
num[0]= "Primeiro número";
```

Obs.: Em Java, é impossível acessar ou atribuir um valor a uma posição do array fora dos limites deste.

Obs.: Colocar mais elementos em um vetor gera um erro do tipo

`ArrayIndexOutOfBoundsException`.

Exemplo:

```
public class HalfDollars {
    public static void main(String[] arguments) {
        int[] denver = { 15000006, 18810000, 20752110 };
        int[] philadelphia = new int[denver.length];
        int[] total = new int[denver.length];
        int average;
        philadelphia[0] = 15020000;
        philadelphia[1] = 18708000;
        philadelphia[2] = 21348000;
        total[0] = denver[0] + philadelphia[0];
        total[1] = denver[1] + philadelphia[1];
        total[2] = denver[2] + philadelphia[2];
        average = (total[0] + total[1] + total[2]) / 3;
        System.out.println("1993 production: " + total[0]);
        System.out.println("1994 production: " + total[1]);
        System.out.println("1995 production: " + total[2]);
        System.out.println("Average production: " + average);    }    }
```

Ao lidar com arrays, podemos percorrer os elementos do array, em vez de lidar com cada element individualmente. Isso torna o código muito mais fácil de ler.

### Arrays multidimensionais (matrizes)

Um uso comum de matrizes é para representar os dados em uma grade x,y de elementos de array.

Java não aceita arrays multidimensionais, mas podemos conseguir a mesma funcionalidade declarando um array de arrays. Esses arrays também podem conter arrays, e assim por diante, por tantas dimensões quantas forem necessárias.

Exemplo:

Registrar um valor inteiro a cada dia por um ano e organizar esses valores por semana.

```
int[][] dias= new int[52][7];
```

Esse array de arrays contém um total de 364 inteiros, um para cada dia em 52 semanas. Poderíamos definir o valor para o primeiro dia da décima semana com a seguinte instrução:

```
dias[9][0]=14200;
```

Exemplo: Crie uma matriz 3 por 4 e inicialize seus valores.

```
public class Matriz {
    public static void main(String[] args) {
        /* declarando dois arrays do tipo int*/
        int[] a1;
        int[] a2;
        /* instanciando os dois arrays */
        a1 = new int[ 10 ]; // array com 10 posições de int
        a2 = new int[ 30 ]; // array com 30 posições de int
        /* imprimindo o tamanho dos array a1 e a2 */
        System.out.println("a1 tamanho : " + a1.length);
        System.out.println("a2 tamanho : " + a2.length);
    }
}
```

```
/* declarando, instanciando e atribuindo valores ao array
notas */
float[] notas = { 7.5f , 9.0f , 10.0f };
/* imprimindo as posições do array notas */
System.out.println("Notas : " + notas[0] + " - " +
                    notas[1] + " - " +
                    notas[2]);

/* simulando array bidimensional*/
int[][] matriz = new int[ 3 ][ 4 ];
System.out.println("Linhas da Matriz : " + matriz.length);
System.out.println("Colunas da Matriz : " + matriz[0].length);
matriz[0][0] = 1;
matriz[0][1] = 2;
matriz[0][2] = 3;
matriz[0][3] = 4;
matriz[1][0] = 5;
matriz[1][1] = 6;
matriz[1][2] = 7;
matriz[1][3] = 8;
matriz[2][0] = 9;
matriz[2][1] = 10;
matriz[2][2] = 11;
matriz[2][3] = 12;
/* Imprimindo o array */
for (int i = 0; i < matriz.length; i++){
    for (int j = 0; j < matriz[0].length; j++){
        System.out.print(matriz[i][j] + "\t");
        System.out.println();
    }
}
}
```

### Lista - 7 VETOR – MATRIZES

- 1) Criar um algoritmo que entre com dez nomes e imprima uma listagem contendo todos os nomes.
- 2) Criar um algoritmo que armazene nome e duas notas de 5 alunos e imprima uma listagem contendo nome, as duas notas e a média de cada aluno. Mostre a situação de cada aluno, ou seja, aprovado ou reprovado.
- 3) Criar um algoritmo que armazene 5 nomes em um vetor. Ordenar e imprimir uma listagem em ordem crescente.
- 4) Armazenar 15 números inteiros em um vetor NUM e imprimir uma listagem numerada contendo o número e uma das mensagens: par ou ímpar.
- 5) Armazenar nome e salário de 5 pessoas. Calcular o novo salário sabendo-se que o reajuste foi de 8%. Imprimir uma listagem com nome, salário antigo e novo salário.
- 6) Ler um vetor vet de 5 elementos e obter um vetor w cujos componentes são os fatoriais dos respectivos componentes de v.
- 7) Crie um algoritmo que armazene 5 números e gere um vetor com os quadrados destes números.
- 8) Crie um algoritmo que armazene 5 nomes e faça a ordenação destes em ordem crescente.

9) Criar um algoritmo que armazene nomes de duas disciplinas, sabendo-se que as turmas têm sempre 5 alunos. Armazene também a matrícula e as notas do alunos em matrizes de ordem cinco. Imprima a saída da seguinte maneira:

```
/*           Nome da disciplina:
*           Mat.           Nota: */
```

10) Entrar com valores para uma matriz A 3x4. Gerar e imprimir uma matriz B que é o triplo da matriz A.

11) Entrar com valores inteiros para um matriz A[4][4] e para uma matriz B[4][4]. Gerar e imprimir a matriz SOMA[4][4].

## PARTE 6 – Orientação a Objetos



## Pacotes

É um modo de organizar grupos de classes. Um pacote contém qualquer quantidade de classes relacionadas em finalidade, em escopo ou por herança.

Se os programas forem pequenos e usarem um número limitado de classes, você poderá descobrir que não precisa explorar pacote algum. Todavia, quando começar a criar projetos mais sofisticados com muitas classes relacionadas umas as outras por herança, descobrirá o benefício de organizá-las em pacotes.

Os pacotes são úteis por vários motivos gerais:

- Os pacotes permitem a organização de classes em unidades. Assim temos pastas ou diretórios no disco rígido para organizar seus arquivos e aplicações, os pacotes permitem a organização das classes em grupos, de modo que possamos usar o que for necessário em um programa.
- Os pacotes reduzem problemas de conflitos em relação a nomes. À medida que o número de classes Java cresce, cresce também a probabilidade de usar o mesmo nome de classe de outro desenvolvedor, aumentando a possibilidade de conflitos de nomes e de mensagens ao tentar integrar grupos de classes em um único programa. Os pacotes fornecem um meio de referência específica à classe desejada, mesmo que ela compartilhe um nome com uma classe de outro pacote.
- Os pacotes permitem proteger classes, variáveis e métodos de maneiras mais amplas do que para cada classe individual.
- Os pacotes podem ser usados exclusivamente para identificar o seu projeto.

Todas as vezes que usamos o comando `import` e toda vez que nos referimos a uma classe por seu nome de pacote inteiro `java.awt.Font`, por exemplo, estamos usando pacotes.

Para usar uma classe contida em um pacote, use um dos três mecanismos:

1. Se a classe que desejamos usar estiver no pacote `java.lang` (por exemplo, `System` ou `Date`), poderemos simplesmente usar o nome da classe para se referenciar a essa classe. As classes `java.lang` estão disponíveis de maneira automática.
2. Se a classe que deseja usar estiver em algum outro pacote, poderemos nos referir a ela por seu nome completo, incluindo quaisquer nomes de pacotes (por exemplo, `java.awt.FlowLayout`).
3. Para as classes que usa com frequência a partir de outros pacotes, precisamos importar classes individuais ou um pacote inteiro de classes. Depois que uma classe ou pacote tiver sido importado, podemos nos referir a esta classe somente pelo seu nome.

Para se referir a uma classe em algum outro pacote, podemos usar o nome completo, que é o nome de classe precedido por quaisquer nomes de pacotes.

Obs.: Não é necessário importar classe ou pacote se formos utilizá-lo desta maneira:

```
java.awt.Font texto = new java.awt.Font();
```

Para classes utilizadas apenas uma ou duas vezes no programa, o uso de nomes completos faz sentido. Contudo, se formos usar várias vezes, ou se o nome do pacote for muito longo, com muitos subpacotes, então importe essa classe, para economizar tempo de digitação.

Obs.: Ao se criar o pacote do exemplo abaixo, será criada uma subpasta chamada `curso` para armazenar o arquivo `bytecode`, ou seja, os `(.class)`.

```
package curso;
```

Para importar classes de um pacote, use a declaração `import`.

```
import curso;
```

Obs.: podemos importar um pacote inteiro de classes, usando um asterisco(\*).

```
import javax.swing.*;
```

Obs.: Contudo o asterisco não permite a importação de vários pacotes com nomes semelhantes.

- `java.util`; `java.util.jar`; `java.util.prefs`.
- Não podemos importar os três pacotes com a seguinte instrução:  

```
o import java.*;
```
- Para tornar os três disponíveis em uma classe, as seguintes instruções são necessárias:

```
o import java,util.*;  
o import java,util.jar.*;  
o import java,util.prefs.*;
```

Obs.: não se pode iniciar nomes de classes parciais, por exemplo, J\* para importar todas as classes que comecem por J.

Obs.: a importação de um grupo de classes não torna o seu programa mais lento nem o aumenta; somente as classes realmente usadas no código serão carregadas, conforme a necessidade.

Java permite se referir às constantes de uma classe pelo seu nome. Sendo assim, devemos fazer uma import static o que torna disponíveis as constantes específicas nesta classe.

Exemplo:

```
Color.black; Math.PI e File.separator.
```

Após o import static java.lang.Math.\*;

Exemplo:

```
import static java.lang.Math.*;  
public class Constantes{  
    public static void main(String args[]){  
        System.out.println("PI: "+PI);  
        System.out.println(" "+(PI*3));  
    }  
}
```

Cuidado: Ao importar classes com mesmo nome de pacotes diferentes será necessário se referir a esta classe pelo seu caminho completo. Exemplo: a classe Date existe nos pacotes: java.util e java.sql, sendo assim ao se referir a classe Date do pacote java.util, utilize: java.util.Date.

### Classe

Em Java, um programa é composto de uma classe principal e quaisquer outras classes que sejam necessárias para dar suporte a ela. Essas classes de suporte incluem qualquer uma daquelas na bibliotecas de classes Java de que você precisar.

Por padrão, as classes herdam da classe Object. Ela é a superclasse de todas as classes na hierarquia Java.

A palavra-chave extends é usada para indicar a superclasse de uma classe, ou seja, fazer herança.

**Exemplo:**

```
public class Cadastro extends JFrame{//corpo da classe}
```

### Objeto

Um objeto é uma representação abstrata de uma entidade do mundo real, que tem um identificador único, propriedades embutidas e a habilidade de interagir com outros objetos e consigo mesmo. O estado do objeto é um conjunto de valores que os atributos do objeto podem ter em um determinado instante do tempo.

As classes definem a estrutura e o comportamento de um tipo de objeto, elas atuam como templates.

Todos os objetos criados a partir de uma classe são idênticos.

Quando declaramos variáveis de qualquer tipo primitivo o espaço em memória é alocado como parte da operação.

A declaração de uma variável referência a um objeto não aloca espaço na memória. A alocação de memória é feita somente quando o objeto é criado.

Exemplo:

```
Cadastro cad;  
cad = new Cadastro();
```

O primeiro comando, a declaração, aloca apenas o espaço suficiente para a referência. O segundo, aloca o espaço para os atributos do objeto cad, somente após a criação do objeto é que seus membros atributos e métodos podem ser referenciados.

Uma classe pode ter construtores especializados ou somente o construtor default.

Um construtor recebe sempre o mesmo nome da classe.

O método construtor gera uma instância do objeto em memória e o seu código é executado imediatamente após a criação do objeto provendo-lhe um estado inicial.

Obs.: Valores de parâmetros são passados no momento da criação.

### Herança

Significa ser capaz incorporar os dados e métodos de uma classe previamente definida. Assim como a herança de todas as operações e dados, você pode especializar métodos da classe ancestral e especificar novas operações e dados, para refinar, especializar, substituir ou estender a funcionalidade da classe progenitora.

Terminologia:

- Estender- criar uma nova classe que herda todo o conteúdo da classe
- Superclasse – classe progenitora
- Subclasse – classe filha

Em Java, a classe "Object" é a raiz de todas as classes.

### Variáveis de instância - atributos

As são consideradas variáveis de instância se forem declaradas fora de uma definição de método, e não são modificadas pela palavra-chave static.

Por um costume de programação, a maioria das variáveis de instância é definida logo após a primeira linha da definição da classe, mas também podem ser definidas no final.

Exemplo:

```
public class Cadastro extends JFrame{
    String nome; //variável de instância
    String sexo; //variável de instância
    int idade; //variável de instância
}
```

**Obs.:** São definidas fora de um método.

### Variáveis de classe

As variáveis de classe se aplicam a uma classe como um todo, em vez de serem armazenadas individualmente em objetos de classe.

As variáveis de classe são eficazes na comunicação entre diferentes objetos da mesma classe ou para registrar informações por toda a classe entre um conjunto de objetos.

A palavra-chave static é usada na declaração da classe para declarar uma variável de classe.

static – qualificador ou "specifier", que indica que o método deve ser compartilhado por todos os objetos que são criados a partir desta classe.

Exemplo:

```
public class Cadastro extends JFrame{
    static String nome= "Cliente"; //variável de classe
    String sexo; //variável de instância
    int idade; //variável de instância
    static final double rendaMinima=1000; /*variável de classe –é
uma constante em Java*/
}
```

As variáveis de classe são variáveis definidas e armazenadas na própria classe. Seus valores se aplicam à classe e a todas as suas instâncias.

Com variáveis de instância, cada nova instância da classe recebe uma nova cópia das variáveis de instância que a classe define. Cada instância, então, pode alterar os valores dessas variáveis de instância sem afetar quaisquer outras instâncias. Com as variáveis de classe, só existe uma cópia dessa variável. A alteração do valor dessa variável o altera para todas as demais instâncias.

Cada instância da classe Cadastro possui seus próprios valores para sexo e idade. A variável de classe nome e rendaMinima, no entanto, possui apenas um valor para todos os membros do Cadastro: "Cliente". Mude o valor de nome, e todas as instâncias de Cadastro são afetadas.

No caso da variável de classe rendaMinima ela é uma constante e não pode ser modificada, isto é feito colocando-se a palavra-chave final após static.

Para acessar variáveis de classe, use a mesma notação de ponto que as variáveis de instância. Para obter ou alterar o valor da variável de classe, podemos usar a instância ou o nome da classe no lado esquerdo do ponto. As duas linhas de saída no exemplo a seguir apresentam o mesmo valor:

Exemplo:

```
Cadastro cad = new Cadastro();
System.out.println("O nome no Cadastro é:"+cad.nome);
System.out.println("O nome no Cadastro é:"+Cadastro.nome);
```

Exemplo:

```
public class MembrosEstaticos {
    /* Atributo estático */
    private static int contObjetos = 0;
    public MembrosEstaticos () {
        contObjetos++;
    }
    /* Método estático */
    public static void numeroDeObjetos () {
        System.out.println("Número de objetos instanciados : " +
        contObjetos);
    }
}
```

Exemplo:

```
public class MembrosEstaticosApp {
    public static void main (String[] args) {
        for (int i = 1; i <= 10; i++) {
            MembrosEstaticos me = new MembrosEstaticos();
            /* chamada ao método estático */
            me.numeroDeObjetos();
        }

        /* chamada ao método estático */
        MembrosEstaticos.numeroDeObjetos();
    }
}
```

As definições de método possuem quatro partes básicas:

- O nome do método;
- Uma lista de parâmetros;
- O tipo de objeto ou primitivo retornado pelo método;
- O corpo do método.

Obs.: o nome do método e sua lista de parâmetros correspondem a sua assinatura.

Obs.: Em Java podemos ter vários métodos na mesma classe com o mesmo nome, mas com diferenças nas assinaturas. Essa prática é chamada de sobrecarga de método.

A menos que um método tenha sido declarado como void como seu tipo de retorno, o método retorna algum tipo de valor quando é completado. O valor precisa ser retornado explicitamente em algum ponto de saída dentro do método, usando a palavra-chave return.

Exemplo:

```
tipoRetorno nomeMétodo (tipo1 arg1, tipo2 arg2, tipo3 arg3...){  
    //corpo do método}
```

Quando chamamos um método com parâmetro de objeto, os objetos que passam para o corpo do método são passados por referência. Qualquer modificação feita nos objetos dentro do método persistem fora dele.

### Métodos de classe

Os métodos de classe estão disponíveis a qualquer instância da própria classe e podem estar disponíveis a outras classes. Além disso, ao contrário de um método de instância, uma classe não exige uma instância da classe para que seus métodos sejam chamados.

Exemplo:

```
System.exit(0);
```

A classe System define um conjunto de métodos úteis durante a exibição do texto, recuperando informações de configuração e executando outras tarefas.

O método exit(int) fecha uma aplicação com um código de status que indica sucesso (0) ou falha(qualquer outro valor).

Para definir métodos de classe, use a palavra-chave static na frente da definição de método.

Exemplo:

```
double x=Double.parseDouble("42");
```

parseDouble é um método de classe, pois não precisamos criar nenhum objeto para utilizá-lo.

Obs.: para passar argumentos a um programa Java, os argumentos deverão ser anexados à linha de comando (prompt do MS-DOS) quando o programa for executado.

Exemplo:

```
java Cadastro Abril 450
```

// passados dois argumentos que são arrays de strings para o método main().

Sendo assim, args[0]=Abril e args[1]=450.

Dentro do método main() podemos tratar os argumentos que foram recebidos.

Duas características diferenciam os métodos com o mesmo nome:

- O número de argumentos;
- O tipo de dado ou objetos de cada argumento.

Essas duas características fazem parte da assinatura de um método. O uso de vários métodos com o mesmo nome e assinaturas diferentes é chamado de sobrecarga.

A sobrecarga de método pode eliminar a necessidade de métodos inteiramente diferentes, que realizam essencialmente a mesma coisa. A sobrecarga também torna possível que os métodos se comportem de modo diferente, dependendo dos argumentos que recebem.

Quando chamamos um método em um objeto, a Java combina o nome do método e os argumentos a fim de escolher qual definição de método deve ser executada.

Para criar um método sobrecarregado, crie diferentes definições de método em uma classe, cada uma com o mesmo nome, mas com diferentes listas de argumentos. A diferença pode ser o número, o tipo de argumentos ou ambos. Java permite a sobrecarga do método, desde que cada lista de argumentos seja exclusiva para o mesmo nome do método.

Exemplo:

```
public class SobreCarga {
    public static float media (int a, int b) {
        return ( a + b ) / 2;
    }
    public static float media (float a, float b) {
        return ( a + b ) / 2;
    }
    public static float media (int a, int b, int c) {
        return ( a + b + c ) / 3;
    }
    public static float media ( float a, float b, float c ) {
        return ( a + b + c ) / 3;
    }
    /* Não é possível ter um método com apenas o tipo de retorno
diferente */
    /*
    public static double media ( float a, float b, float c ) {
        return ( a + b + c ) / 3;
    }
    */
}
```

Obs.: Não é possível ter um método com apenas o tipo de retorno diferente

```
public class SobreCargaApp {
    public static void main (String[] args) {
        System.out.println( SobreCarga.media( 5, 7 ) );
        System.out.println( SobreCarga.media( 5, 7, 3 ) );
        System.out.println( SobreCarga.media( 8f, 2f ) );
        System.out.println( SobreCarga.media( 5.3f, 7.9f, 3.1f ) );
        /* Conversão explícita pra float */
        System.out.println( SobreCarga.media( 8f, 4 ) );
    }
}
```

São chamados automaticamente quando os objetos dessa classe são criados.

Ao contrário dos outros métodos, um construtor não pode ser chamado diretamente. Java faz três ações quando `new` é usada para criar uma instância de uma classe.

- Aloca memória para o objeto;
- Inicializa as variáveis de instância desse objeto, seja para valores iniciais ou para um padrão (0 para números, null para objetos, false para Booleanos ou '0' para caracteres).
- Chama o método construtor da classe, que pode ser um dentre vários métodos.

Se uma classe não possui quaisquer métodos construtores definidos, um objeto ainda será criado quando o operador `new` for usado em conjunto com a classe. Contudo, podemos ter de definir essas variáveis de instância ou chamar outros métodos que o objeto precise para inicializar-se.

Definindo métodos construtores em sua próprias classes, é possível definir valores iniciais de variáveis de instância, chamar métodos com base nessas variáveis, chamar métodos em outros objetos e definir propriedades iniciais de um objeto.

Também podemos sobrecarregar métodos construtores, como pode fazer com métodos normais, para criar um objeto que tenha propriedades específicas com base nos argumentos fornecidos a `new`.

Os construtores se parecem muito com os métodos normais, com três diferenças básicas:

- Eles sempre têm o mesmo nome da classe.
- Eles não tem um tipo de retorno.
- Eles não podem retornar um valor no método usando a instrução `return`.

Exemplo:

```
import javax.swing.*;
public class Cadastro extends JFrame{
    String nome;
    int idade;
    char sexo;
    Cadastro(String nome, int idade, char sexo){
        this.nome=nome;
        this.idade=idade;
        this.sexo=sexo;
    }
}
```

Poderíamos criar um objeto dessa classe com a seguinte instrução:

```
Cadastro cad = new Cadastro("Ana", 32, 'F');
```

O objeto teria os seguintes valores:

- `cad.Nome="Ana";`
- `cad.idade=32;`
- `cad.sexo='F'.`

### Polimorfismo por reescrita de métodos

Java procura a definição de um método na classe do objeto. Se não encontrar uma, ele passará para a chamada de método acima na hierarquia de classe, até que uma definição de método seja encontrada. A herança de método permite definir e usar métodos repetidamente em subclasses, sem ter de duplicar o código.

Contudo, pode haver ocasiões em que desejemos que um objeto responda aos mesmos métodos, mas que tenha um comportamento diferente quando esse método é chamado. Nesse caso, você pode redefinir o método. Para redefinir um método, defina um em uma subclasse com a mesma assinatura do método em uma superclasse. Depois, quando o método for chamado, o método da subclasse será encontrado e executado no lugar daquele que se encontra na superclasse.



Para reescrever um método, basta criar um método na sua superclasse que tenha a mesma assinatura (nome e lista de argumentos) do método definido pela superclasse da sua classe. Como Java executa a primeira definição de método que encontra, que combine com a assinatura, a nova assinatura esconde a definição do método original.

Exemplo:

```
class Imprima {
    int x = 0;
    int y = 1;
    void imprimir() {
        System.out.println("x é " + x + ", y é " + y);
        System.out.println("Eu sou uma instância da classe " +
            this.getClass().getName());
    }
}
```

Exemplo: Classe Executora

```
class SubImprima extends Imprima {
    int z = 3;
    public static void main(String[] arguments) {
        SubImprima obj = new SubImprima();
        obj.imprimir();    }    }
```

Exemplo: Correção com reescrita do método acrescentando a variável de instância z.

```
class SubImprima extends Imprima {
    int z = 3;
    void imprimir() {
        System.out.println("x é " + x + ", y é " + y + ",z é "+z);
        System.out.println("Eu sou uma instância da classe " +
            this.getClass().getName());
    }
    public static void main(String[] arguments) {
        SubImprima obj = new SubImprima();
        obj.imprimir();    }    }
```

## Super

Normalmente, existem dois motivos para querermos definir um método já implementado por uma superclasse:

- Para substituir completamente a definição desse método original;
- Para aumentar o método original com algum comportamento adicional.

A redefinição de um método e a entrega de uma nova definição ao método escondem a definição do método original. Entretanto, existem ocasiões em que o comportamento deve ser acrescentado à definição original, em vez de substituí-la completamente, em especial quando o comportamento é duplicado no método original e no método que o redefine. Chamando o método original no corpo do método que o redefine, podemos acrescentar apenas o necessário.

A palavra-chave `super` é usada para chamar o método original de dentro de uma definição de método. Essa palavra-chave passa a chamada de método para cima na hierarquia.

Exemplo:

```
class SubImprima extends Imprima {
    int z = 3;
    void imprimir() {
        System.out.println("x é " + x + ", y é " + y + ",z é "+z);
        System.out.println("Eu sou uma instância da classe " +
            this.getClass().getName());
        super.imprimir();//chama o método da superclasse
    }
}
```



```
    }  
    public static void main(String[] arguments) {  
        SubImprima obj = new SubImprima();  
        obj.imprimir();  
    }  
}
```

Obs.: A palavra-chave `super`, semelhante à palavra-chave `this`, é um marcador de lugar para a superclasse da classe. Pode-se usá-la em qualquer lugar em que utiliza `this`, mas `super` refere-se à superclasse, em vez do objeto atual.

## THIS

**Escopo** – é a parte de um programa em que uma variável ou outro tipo de informação existe, possibilitando o uso de variáveis em instruções ou expressões. Quando a parte que define o escopo tiver concluído sua execução, a variável deixará de existir.

**Variável com escopo local** – só pode ser usada dentro do bloco em que foi definida.

**Variável de instância** – se estende à classe inteira, de modo que podem ser usadas por qualquer um dos métodos de instância dentro da classe.

**Ordem de busca da variável** – verifica 1º no escopo atual (que poderia ser um bloco), depois em cada escopo mais externo e, finalmente, na definição do método atual. Se a variável não for uma variável local, então Java verificará uma definição dessa variável como uma variável de instância ou de classe, na classe atual. Se a definição da variável assim não for encontrada, ela procurará em cada superclasse, uma por vez.

**THIS** - É uma palavra chave usada num método como referência para o objeto corrente, ela tem o significado de: “o objeto para o qual este trecho de código está sendo executado.”

Refere-se ao objeto corrente quando usado no código de um método não estático. Usado com frequência para passar uma referência do objeto corrente num envio de mensagem para outro objeto.

A → this → B

A=this

Exemplo:

```
public class Teste{  
    int A=0;  
    public void Teste(int A){  
        this.A=A; } } //this.A - refere-se a variável de instância
```

A palavra-chave `this` é a única forma de garantir que o valor inteiro `A` do método `Teste()` será passado para a variável inteira `A` da classe `Teste`.

A palavra `this` é muito utilizada quando um dos parâmetros do método tem o mesmo nome que uma variável de instância.

```
void saca (double quantidade){  
    double novoSaldo = this.saldo-quantidade;  
    this.saldo=novoSaldo; }
```

A palavra-chave `this` refere-se ao objeto atual, e podemos usá-la em qualquer lugar onde uma referência a um objeto pudesse aparecer:

Em muitos casos, você pode não ter de usar explicitamente, pois ela será assumida. Por exemplo, você pode se referir a variáveis de instância e chamadas de método definidas na classe atual simplesmente pelo nome, pois o `this` é implícito nessas referências.

Exemplo:

```
t=x;  
resetData(this);
```

This é uma referência à instância atual de uma classe. Devemos usá-la apenas dentro do corpo de uma definição de método de instância.

Os métodos de classe – que são declarados com a palavra-chave static não podem usar this.

Exemplo:

```
class ScopeTest{
int test=10;
void printTest(){
int test=20;
System.out.println("Test:"+test);}
public static void main (String args[]){
ScopeTest st= new ScopeTest();
st.printTest();}}
```

Nessa classe, temos duas variáveis com o mesmo nome e definição. A primeira, uma variável de instância, possui o nome test e é inicializada com o valor 10. A segunda é uma variável local com o valor de 20.

A variável local test, dentro do método printTest(), esconde a variável de instância test. Quando o método printTest() é chamado de dentro do método main(), ele indica que test é igual a 20, embora haja uma variável de instância igual a 10. Para evitar esse problema use this.test para se referir à variável de instância e use apenas test para se referir à variável local.

Exemplo:

```
public abstract class Pessoa {
private String identificacao;//variável de instância
private String nome; //variável de instância
private String dataNascimento; //variável de instância
private char sexo;//variável de instância
public Pessoa (String id, String nome, String datNasc, char sexo) {
    this.nome = nome;//a variável de instância recebe nome
    this.dataNascimento = datNasc; //a variável de instância
recebe datNasc

    this.sexo = sexo;//a variável de instância recebe sexo
}
public void id (String id) {
    this.identificacao = id;
}
public void nome (String nome) {
    this.nome = nome;
}
public void dataNascimento (String dataNascimento) {
    this.dataNascimento = dataNascimento;
}
public void sexo (char sexo) {
    this.sexo = sexo;
}
public String id () {
    return identificacao;
}
public String nome () {
    return nome;
}
```

```
}  
public String dataNascimento () {  
    return dataNascimento;  
}  
public char sexo () {  
    return sexo;  
}  
public String toString () {  
    return(nome + " | " + dataNascimento + " | " + sexo);  
}  
}
```

## Encapsulamento

Mecanismo utilizado visando obter segurança, modularidade e autonomia para objetos;

Conseguido através da definição de visibilidade privada dos atributos, ganhando-se assim autonomia para definir o que o mundo externo ao objeto poderá visualizar e acessar, normalmente através de métodos públicos.

Dica: sempre defina os atributos de uma classe como privados, a não ser que tenha uma boa justificativa para não serem

### Modificadores de Visibilidade

Os modificadores usados com mais frequência nos programas são aqueles que controlam o acesso a métodos e variáveis: public, private e protected. Esses métodos modificadores determinam quais variáveis e métodos de uma classe são visíveis a outras classes.

Usando o controle de acesso, pode-se controlar como outras classes usarão suas classes. Algumas variáveis e métodos em uma classe serão úteis apenas dentro da própria classe e deverão ficar escondidos de outras que possam interagir com essa classe. Esse processo é chamado de encapsulamento. Um objeto controla o que o mundo exterior pode saber a respeito dele e como o mundo exterior pode interagir com ele.

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

Java oferece quatro níveis de controle de acesso: public, private, protected e um nível padrão especificado sem o uso desses modificadores de controle de acesso.

- Public – estes atributos e métodos são sempre acessíveis em todos os métodos de todas as classes. Este é o nível menos rígido de encapsulamento, que equivale a não encapsular.
- Private – estes atributos e métodos são acessíveis somente nos métodos (todos) da própria classe. Este é o nível mais rígido de encapsulamento.
- Protected – estes atributos e métodos são acessíveis no pacote, nos métodos da própria classe e suas subclasses
- <default> - Visível no pacote e na classe. Variáveis e métodos declarados sem quaisquer modificadores.

TABELA RESUMO

Visibilidade	Public	Protected	Default	Private
Da mesma classe	Sim	Sim	Sim	Sim
De qualquer classe no mesmo pacote	Sim	Sim	Sim	Não
De qualquer classe fora do pacote	Sim	Não	Não	Não
De uma subclasse no mesmo pacote	Sim	Sim	Sim	Não
De uma subclasse fora do mesmo pacote	Sim	Sim	Não	Não

## Classes, métodos e variáveis finais

O modificador final é usado com classes, métodos e variáveis para indicar que não serão alterados. Ele possui diferentes significados para cada elemento que pode se tornar final, como a seguir:

- Uma classe final não pode ser subclassificada(extendida);
- Um método final não pode ser redefinido por quaisquer subclasses;
- Uma variável final não pode mudar de valor.

Exemplo: Variável final (constantes)

```
public static final int voltas=10;  
static final String titulo= "Revista";
```

Exemplo: Método final

```
public final void get Assinatura(){  
    //corpo do método}
```

Obs.: O motivo mais comum para declarar um método final é fazer com que a classe seja executada com mais eficiência. Normalmente, quando o ambiente de runtime Java executa um método, ele primeiro verifica a classe atual para encontrar o método, depois verifica sua superclasse e prossegue subindo na hierarquia de classes, até que o método seja encontrado. Esse processo sacrifica alguma velocidade em nome da flexibilidade e facilidade de desenvolvimento.

Se um método é final, o compilador Java pode colocar o bytecode executável do método diretamente em qualquer programa que chame o método. Afinal, o método nem sequer mudará devido a uma subclasse que o redefine.

Exemplo: Classe final

```
public final class Cadastro{  
    //corpo da classe}
```

Obs.: Todos os métodos em uma classe final são automaticamente finais.

### Classes e métodos abstratos

Em uma hierarquia de classes, quanto maior a classe, mais abstrata é sua definição. Uma classe no topo de uma hierarquia de outras classes só pode definir o comportamento e os atributos comuns a todas as classes. Comportamentos e atributos mais específicos entrarão em algum lugar mais baixo na hierarquia.

Quando estivermos fatorando o comportamento e os atributos comuns durante o processo de definição de uma hierarquia de classes, às vezes podemos nos deparar com uma classe que nem sequer precisar ser instanciada diretamente. Em vez disso, esse tipo de classe serve como um lugar para manter comportamentos e atributos comuns, compartilhados por suas subclasses.

Essas classes são chamadas classes abstratas e são criadas usando o modificador abstract.

Exemplo:

```
public abstract class Pessoa{  
    //....  
}
```

Um exemplo de uma classe abstrata é java.awt.JComponent, a superclasse dos componentes da interface gráfica com o usuário. Como diversos componentes herdam dessa classe, ela contém métodos e variáveis úteis a cada uma delas. Contudo, não existe algo como um componente genérico que possa ser acrescentado a uma interface, de modo que nunca precisa criar um objeto Component em um programa.

As classes abstratas podem conter tudo o que uma classe normal pode conter, incluindo métodos construtores, pois suas subclasses ter de herdar os métodos. As classes abstratas também podem conter métodos abstratos, que são assinaturas de método sem implementação. Esses métodos são implementados em subclasses de classe abstrata. Os métodos abstratos são declarados com o modificador abstract. Não podemos declarar um método abstrato em uma classe não abstrata. Se uma classe não tiver nada além de métodos abstratos, será melhor usar uma interface.

Exemplo:

```
public abstract class Pessoa {
    private String identificacao;
    private String nome;
    private String dataNascimento;
    private char sexo;
    public Pessoa (String id, String nome, String datNasc, char sexo) {
        this.nome = nome;
        this.dataNascimento = datNasc;
        this.sexo = sexo;
    }
    public void id (String id) {
        this.identificacao = id;
    }
    public void nome (String nome) {
        this.nome = nome;
    }
    public void dataNascimento (String dataNascimento) {
        this.dataNascimento = dataNascimento;
    }
    public void sexo (char sexo) {
        this.sexo = sexo;
    }
    public String id () {
        return identificacao;
    }
    public String nome () {
        return nome;
    }
    public String dataNascimento () {
        return dataNascimento;
    }
    public char sexo () {
        return sexo;
    }

    public String toString () {
        return(nome + " | " + dataNascimento + " | " + sexo);
    }
}
```

Exemplo:

```
public interface SituacaoAcademica {
    public static final float MEDIA = 5.0f;
    public static final int FALTAS = 18;
    public static final String AP = "Aprovado";
    public static final String RF = "Reprovado por Falta";
    public static final String RM = "Reprovado por Média";

    public abstract void faltas (int faltas);
    public abstract void notas (float n1, float n2, float n3);
}
```

```
        public abstract void nota1 (float n);
        public abstract void nota2 (float n);
        public abstract void nota3 (float n);

        public abstract String situacaoFinal ();
        public abstract float media ();
        public abstract int faltas ();
        public abstract float nota1 ();
        public abstract float nota2 ();
        public abstract float nota3 ();
    }
}
```

Exemplo:

```
public class Alunos {
    private Aluno[] alunos;
    private int posNovoAluno = 0;
    public Alunos() {
        alunos = new Aluno[10]; }
    public void insere (int mat, String nome, float n1, float n2, float n3){
        alunos[posNovoAluno] = new Aluno(mat, nome, n1, n2,n3);
        posNovoAluno++; }
    public void escreve () {
        String linha = "*****";
        System.out.println(linha);
        for (int i = 0; i < posNovoAluno; i++) {
            alunos[i].escreve(); }
        System.out.println(linha);
    }
    /* Classe Interna : Aluno */
    class Aluno {
        int matricula;
        String nome;
        float nota1, nota2, nota3;
        Aluno (int mat, String nome, float n1, float n2, float n3) {
            this.matricula = mat;
            this.nome = nome;
            this.nota1 = n1;
            this.nota2 = n2;
            this.nota3 = n3;
        }
        void escreve() {
            String linha = "_____";
            System.out.println(linha +
                "\nMatrícula : \t" + matricula +
                "\nNome : \t" + nome +
                "\nNotas : \t" + nota1 + "\t" + nota2 + "\t" + nota3 +
                "\n" + linha);
        }
    }
}
```

GETTERS E SETTERS

Um padrão que surgiu com o desenvolvimento de Java foi de encapsular, isto é, esconder todos os membros de uma classe. Para permitir o acesso aos atributos de uma maneira controlada, a prática mais comum é de criar dois métodos, um que retorna valor e outro que muda valor.

O padrão para esses métodos é de colocar a palavra get ou set antes do nome do atributo.

Exemplo 1: classe com os atributos públicos.

```
package oo;
public class Cliente{
    public String nome;
    public String escolaridade;
    public int idade;
    public Cliente(){
    }
}
```

Exemplo 1: classe executora de cliente

```
package oo;
public class PrincipalCliente{
    public static void main(String args[]){
        Cliente cliente1 = new Cliente();
        cliente1.escolaridade= "Superior Completo";
        cliente1.idade=25;
        cliente1.nome= "Mario dos Santos";
        System.out.println(pl.escolaridade);
        System.out.println(pl.idade);
        System.out.println(pl.nome);
    }
}
```

Exemplo 2: modificar os atributos da classe Cliente para private

```
package oo;
public class Cliente{
    private String nome;
    private String escolaridade;
    private int idade;
    public Cliente(){
    }
}
```

Quando tentamos executar novamente a classe PrincipalCliente todas as tentativas de acesso aos atributos da classe Cliente apresentarão erros na classe PrincipalCliente, porque agora os atributos são privados. É possível somente acessá-los dentro da própria classe Cliente.

Para solucionarmos este problema deveremos criar os métodos set() e get() públicos para cada atributo que desejemos tornar público. O método set() é o modificar, seu objetivo é alterar o valor. O método get() é o método acessor, pega o valor do atributo.

Exemplo 3: Classe Cliente modificada incluindo os métodos get e set.

```
public class Cliente{
    private String nome;
    private String escolaridade;
    private int idade;
    public Cliente(){
    }
    public String getNome(){
        return nome;
    }
    public void setNome(String nome){
        this.nome=nome;
    }
    public String getEscolaridade(){
        return escolaridade;
    }
    public void setEscolaridade(String escolaridade){
        this.escolaridade=escolaridade;
    }
    public String getIdade(){
        return idade;
    }
    public void setIdade(int idade){
        this.idade=idade;
    }
}
```

Exemplo 3: nova classe executora que acessa os método get e set da classe Cliente,

```
package oo;
public class PrincipalCliente{
    public static void main(String args[]){
        Cliente cliente1 = new Cliente();
        cliente1.setEscolaridade("Superior Completo");
        cliente1.setIdade(25);
        cliente1.setNome("Mario dos Santos");
        System.out.println(pl.getEscolaridade());
        System.out.println(pl.getIdade());
        System.out.println(pl.getNome());
    }
}
```

Os métodos get() e set() podem ser usados para validar um valor de entrada.

Exemplo:

```
public void setIdade(int idade){
    if(idade >=0)
        this.idade=idade;
}
```

Neste caso, só daria entrada se o idade fosse maior que 0.

Para não termos que repetir várias vezes os comandos de impressão, podemos criar um método que faça este trabalho na classe Cliente.

Exemplo 4: incluindo um método para impressão dos atributos.



```
        public class Cliente{
            private String nome;
            private String escolaridade;
            private int idade;
            public Cliente(){
            }
            public String getNome(){
                return nome;
            }
            public void setNome(String nome){
                this.nome=nome;
            }
            public String getEscolaridade(){
                return escolaridade;
            }
            public void setEscolaridade(String escolaridade){
                this.escolaridade=escolaridade;
            }
            public String getIdade(){
                return idade;
            }
            public void setIdade(int idade){
                this.idade=idade;
            }
            public void imprime(){
                System.out.println(pl.getEscolaridade());
                System.out.println(pl.getIdade());
                System.out.println(pl.getNome());
            }
        }
    }
```

Exemplo 4: Classe PrincipalCliente que utiliza o método imprime().

```
package oo;
public class PrincipalCliente{
    public static void main(String args[]){
        Cliente clientel = new Cliente();
        clientel.setEscolaridade("Superior Completo");
        clientel.setIdade(25);
        clientel.setNome("Mario dos Santos");
        clientel.imprime();
    }
}
```

Desenvolver as classes abaixo e fazer um principal e criar 2 objetos de cada classe, alimentar os objetos e exibir o relatório.

Na classe Funcionario, o método getSalario() calcula o salário do funcionário. O método exibeDados() exibe as informações do funcionário, inclusive o salário aproveitando o método getSalario().

Funcionário
String nome int horastrabalhadas float valorhora
void setNome(String nome) String getNome() void setHorasTrabalhadas(int ht) int getHorasTrabalhadas() void setValorHora(float vh) float getValorHora() float getSalario() void exibeDados()

Na classe conta, o método deposito(float qtd) adiciona a quantidade passada por parâmetro ao saldo atual. O método saque(float qtd) retira do saldo a quantidade indicada no parâmetro, caso o saldo seja insuficiente deve retornar o valor 1, caso contrário retorna 0.

Conta
String conta String cliente float saldo
void setCoonta(String cont) String getConta() void setCliente(String cliente) String getCliente() void setSaldo(float sal) float getSaldo() void deposito(float qtd) int saque(float qtd) void relatorio()

Na classe Produto, o método getSituacao() retorna 0 se o estoque estiver acima do estoque mínimo permitido, caso contrário retorna 1. No método relatorio(), caso o estoque esteja abaixo do permitido deve se informado que necessita comprar mais produto.

Produto
int codigo String nome float preço int estoque int estoquemin
void setCodigo(int cod) int getCodigo() void setNome(String nome)

```
String getNome()
void setPreco(float p)
float getPreco()
void setEstoque(int est)
int getEstoque()
void setEstoqueMinimo(int estmin)
int getEstoqueMinimo()
int getSituacao()
void relatorio()
```