



# Programação Java

## Projeto UERJ - Faperj

Professor: Austeclynio Pereira - 2023

# Programação Java



- Início em 10/01/2023;
- Término em 23/02/2023;
- Horário: das 13h às 16h;
- Dias da semana: 3as e 5as;
- Conteúdo do curso em:  
**[www.inovuerj.sr2.uerj.br/portal/cursos\\_integra](http://www.inovuerj.sr2.uerj.br/portal/cursos_integra)**
- Preparação para o curso Desenvolvimento Java para Web;
- Teste ao final do curso;

# Bibliografia



1. *Head First* – Sierra, Kathy; Bates, Bert. O'Reilly, 2005
2. Core Java: Fundamentals Volume 1 – Horstmann, Cay S, 2021
3. Estrutura de Dados & Algoritmos em Java – Lafore, Robert, 2005
4. Conceitos de Computação com Java – Horstmann, Cay S., 2009
5. SCJP Sun Certified Programmer for Java 5 – Sierra, Kathy; Bates, Bert, 2006
6. Murach's Java Programming – Murach, Joel; Boehm, Anne, 2005
7. Object-Oriented Analysis and Design with Applications – Maksimchuk,Robert; Engle,Michael; Young,Bobbi; Conallen,Jim; Houston,Kelli, 2007
8. <https://docs.oracle.com/javase/tutorial/java/index.html>

# Ferramentas de trabalho



- Java 1.8.0\_91;
- IDE Jcreator;

# Introdução



- Foi desenvolvida pela Sun Microsystems em 1991, que foi comprada, em 2008, pela Oracle;
- Recebeu o nome de \*7 mudando depois para OAK;
- Havia outra linguagem com este nome, tornando-se Java em homenagem a ilha de Java, na Indonésia que servia um bom café;
- Ideia era desenvolver uma linguagem de propósito específico orientada para dispositivos eletrônicos como conversores de TV a cabo;
- Com a expansão da Internet, na década de 1990, vislumbraram a exploração deste segmento;
- Criam os *applets*. Pequenos programas, descarregados pelos navegadores, que dão conteúdo dinâmico às páginas *web*;

# Introdução



- Em 1996, com o NetScape e o Internet Explorer suportando *applets*, o sucesso do Java é meteórico;
- De 2001 a 2021 esteve sempre entre as 3 linguagens mais utilizadas no mundo. Em dez/2022, ocupava a 4a. posição(  
<https://www.tiobe.com/tiobe-index/>);
- Outra importante virtude da linguagem Java é ser multiplataforma;
- Altamente escalável;
- Sistemas desenvolvidos em Java podem ser executados por máquinas de diferentes capacidades e dimensões;

# Introdução



- Explora as seguintes facilidades:
  - Orientação a objetos;
  - *Multi-threading*;
  - Manuseio estruturado de erros;
  - Coleta de lixo(*Garbage collector*);
  - Carga dinâmica de classes;

# Introdução



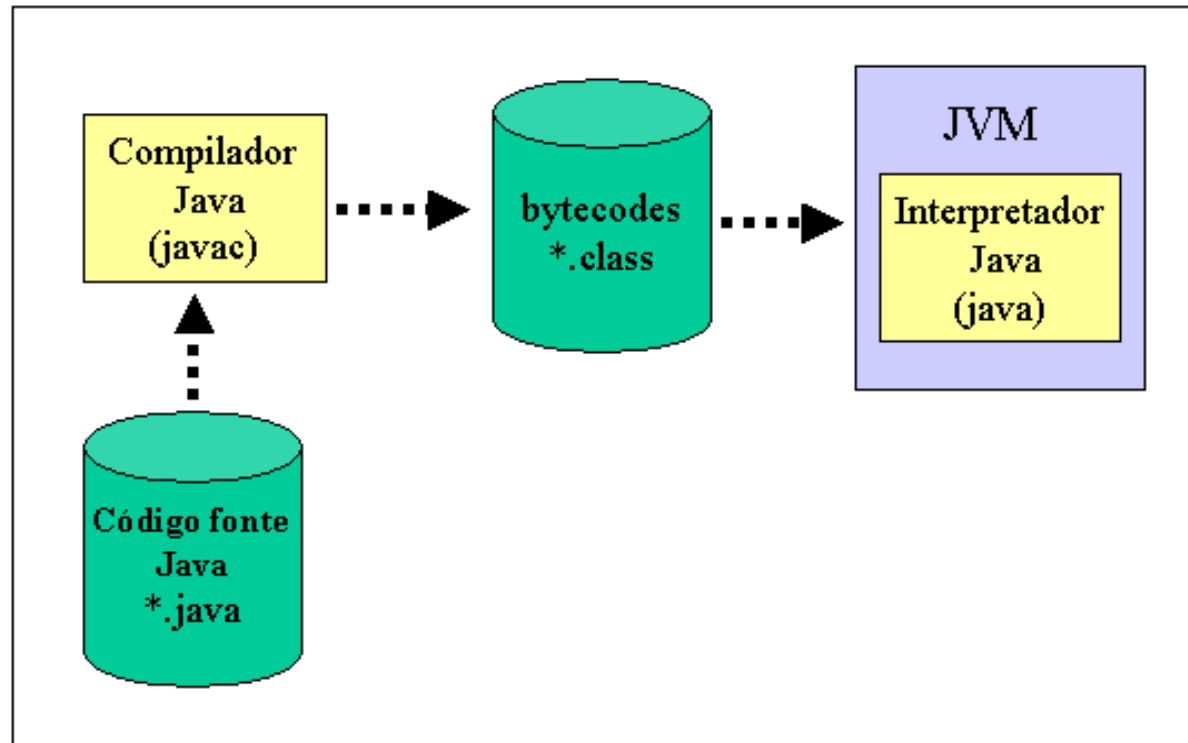
## ■ Plataformas Java:

- Java Micro Edition(JME);
- Java Standard Edition(JSE);
- Java Enterprise Edition(JEE);
- Java FX(JFX);



# Introdução

## Fluxo de execução de um programa



JVM - Java Virtual Machine

# Java e a Programação Orientada a Objetos



## ■ Classificação:

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

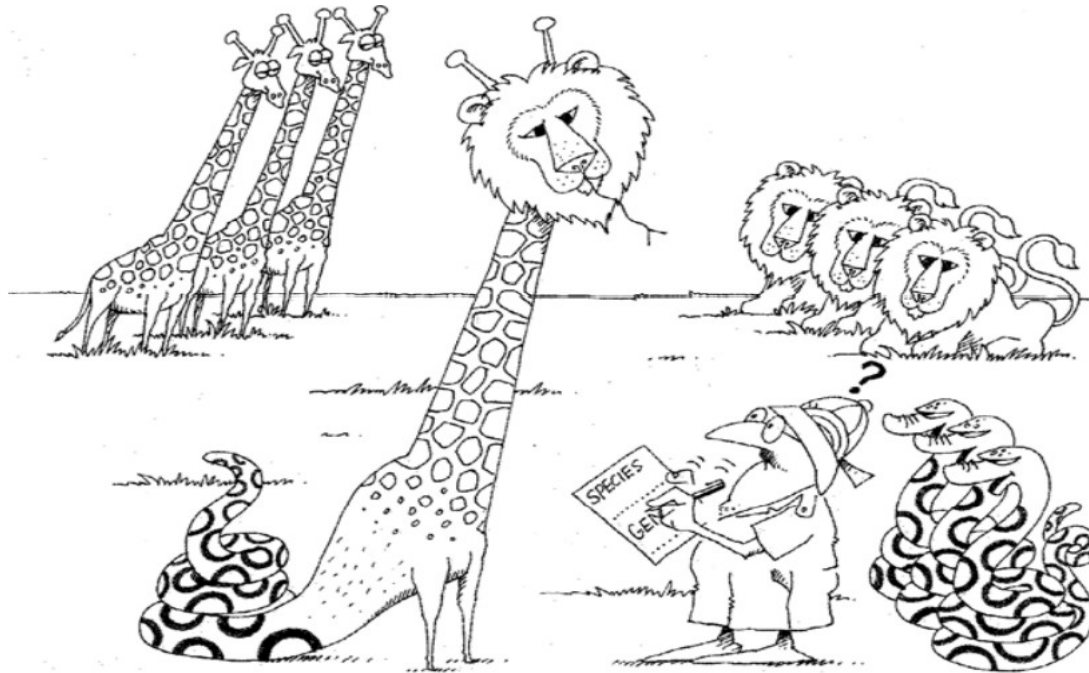


Figura extraída do livro Object-Oriented Analysis and Design with Applications

# Java e a Programação Orientada a Objetos



## ■ Encapsulamento:

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

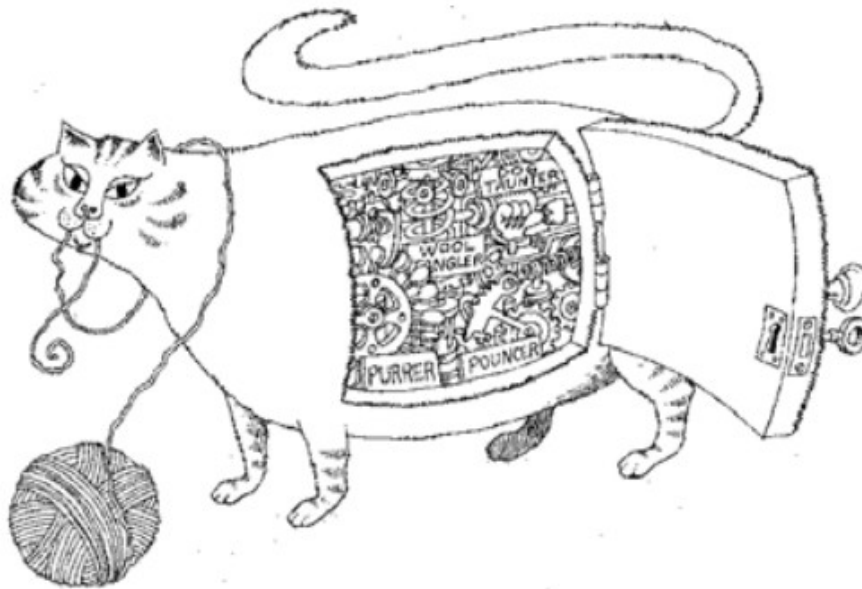


Figura extraída do livro Object-Oriented Analysis and Design with Applications

# Java e a Programação Orientada a Objetos



## ■ Herança:

- Objetos podem herdar funcionalidades de outros objetos, reaproveitando algumas e especializando ou criando outras.

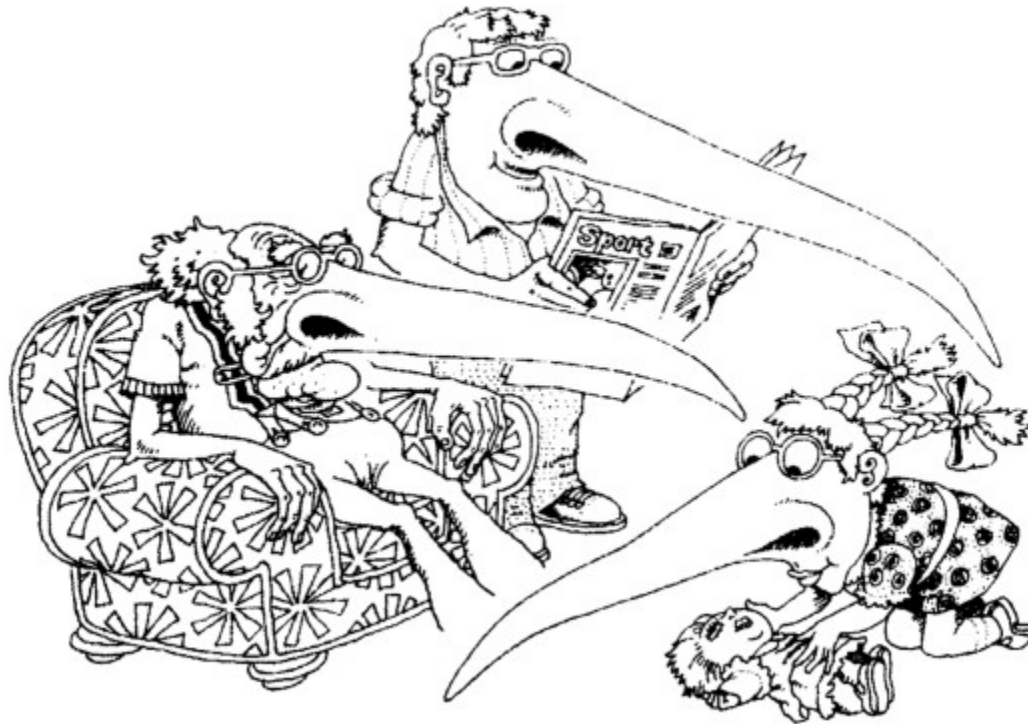


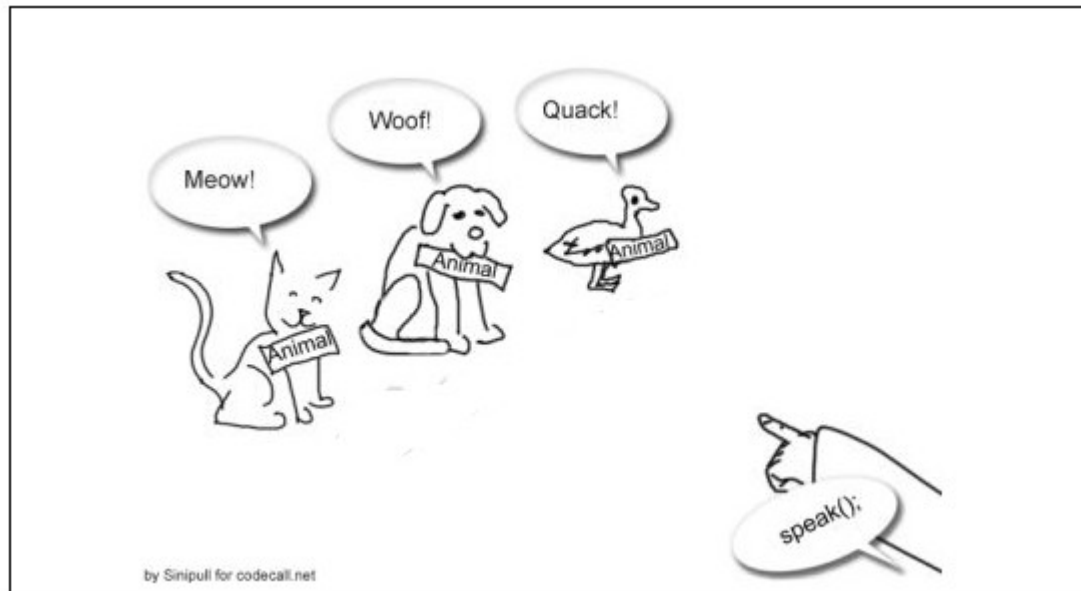
Figura extraída do livro Object-Oriented Analysis and Design with Applications

# Java e a Programação Orientada a Objetos



## ■ Polimorfismo:

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



# Classes e Objetos



## ■ Classes e seus objetos:

- Uma classe pode ser definida como sendo um gabarito, uma planta ou um protótipo que descreve um objeto;
- Um objeto possui estado, atributos ou propriedades e métodos ou comportamentos;
- Exemplos:
  - Os cães têm estados (nome, cor, raça, fome) e comportamentos (latir, buscar, abanar o rabo).
  - As bicicletas também têm estados (aro, cor, marcha atual, cadência atual do pedal, velocidade atual) e comportamentos (mudança de marcha, mudança da cadência do pedal, aplicação de freios);
- Programar orientado a objetos é identificar estados e comportamentos de objetos do mundo real;
- A criação de objetos, a partir de uma classe, chama-se instanciação da classe;
- O objeto dá vida a uma classe. A bicicleta construída é uma representação concreta do projeto bicicleta;
- Cada bicicleta construída é uma instância da classe bicicleta;

# Classes do Java



## ■ Classes com seus construtores e métodos(site da Oracle):

← → ↺ docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html

Trabalhando com J... https://www.ib7.bra... Se você tem banan... MySQL :: MySQL Co... Material UI - Overvi... Ro-Online Web

OVERVIEW PACKAGE **CLASS** USE TREE DEPRECATED INDEX HELP

PREV CLASS NEXT CLASS FRAMES NO FRAMES ALL CLASSES

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

compact1, compact2, compact3  
java.util

**Class ArrayList<E>**

java.lang.Object  
  java.util.AbstractCollection<E>  
    java.util.AbstractList<E>  
      java.util.ArrayList<E>

**All Implemented Interfaces:**  
Serializable, Cloneable, Iterable<E>, Collection<E>, List<E>, RandomAccess

**Direct Known Subclasses:**  
AttributeList, RoleList, RoleUnresolvedList

---

# Classes do Java



## ■ Classes com seus construtores e métodos:

## Constructor Summary

Constructors	
Constructor and Description	
<code>ArrayList()</code>	Constructs an empty list with an initial capacity of ten.
<code>ArrayList(Collection&lt;? extends E&gt; c)</code>	Constructs a list containing the elements of the specified collection, in the order they are returned by the collection's iterator.
<code>ArrayList(int initialCapacity)</code>	Constructs an empty list with the specified initial capacity.

## Method Summary

All Methods	Instance Methods	Concrete Methods
Modifier and Type		Method and Description
boolean		<code>add(E e)</code> Appends the specified element to the end of this list.
void		<code>add(int index, E element)</code> Inserts the specified element at the specified position in this list.
boolean		<code>addAll(Collection&lt;? extends E&gt; c)</code> Appends all of the elements in the specified collection to this list.
boolean		<code>addAll(int index, Collection&lt;? extends E&gt; c)</code> Inserts all of the elements in the specified collection into this list at the position indicated by the index.



# Java e a Programação Orientada a Objetos



## ■ Abstração:

- Capacidade de focar nas características essenciais do objeto, sob o ponto de vista do observador.

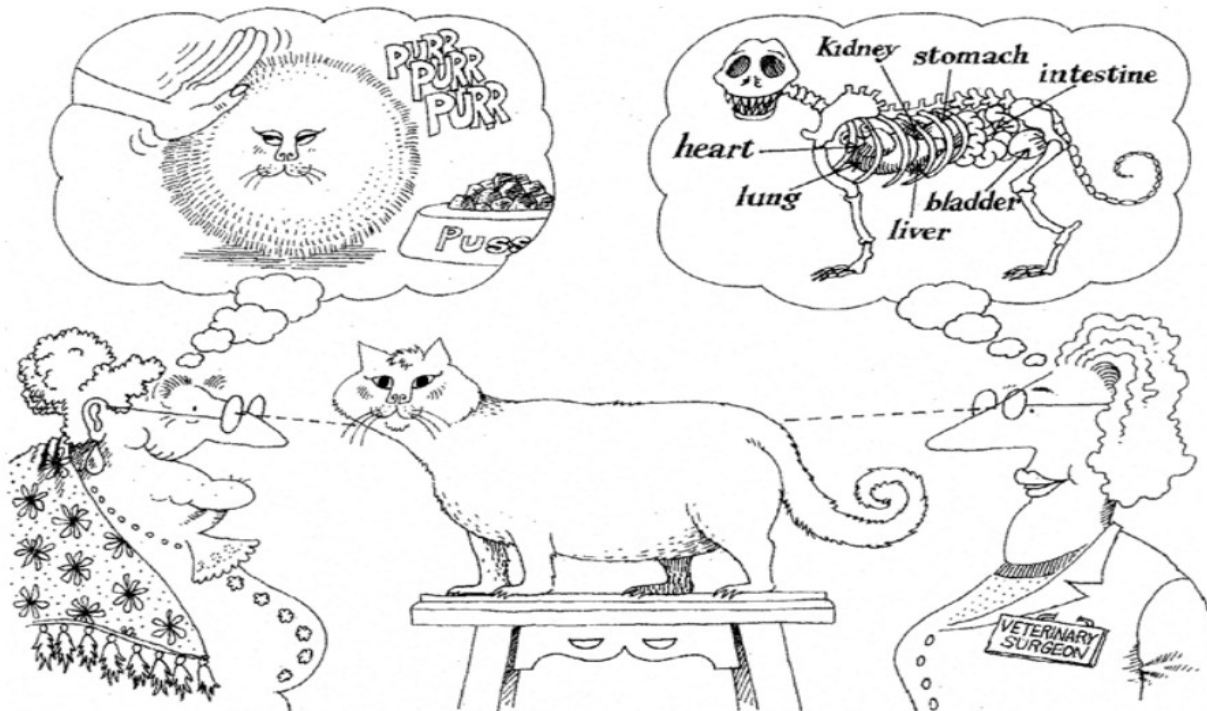


Figura extraída do livro Object-Oriented Analysis and Design with Applications

# Tipos de Dados



- Java é uma linguagem fortemente tipada;
- Abaixo, seus tipos primitivos:

Tipo	Ocupa	Valor Mínimo	Valor Máximo
byte	8 bits	-128	+128
short	16 bits	-32.768	+32.768
int	32 bits	-2.147.483.648	+2.147.483.648
long	64 bits	-9.223.372.036.854.775.808	+9.223.372.036.854.775.808
float	32 bits		
double	64 bits		
boolean	1 bit	false	true
char	16-bit Unicode	\u0000 (0)	\uffff (65535)

# Tipos de variáveis



- Descritas por uma sequência de caracteres alfanuméricos podendo iniciar por uma letra ou pelos caracteres “\_” ou “\$”;
- Podem conter até 32 caracteres;
- Não podem conter espaços, símbolos ou qualquer operador;
- Java é *case sensitive*;
- Abaixo, palavras reservadas da linguagem;

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

# Tipos de variáveis



## ■ Sintaxe para a declaração de variáveis:

Tipo-da-variável variável1[,variavel2[,variável3[...variávelN]]];

Exemplos:

int **valorSalario**;

byte acumuladorAlunos,acumuladorTurmas;

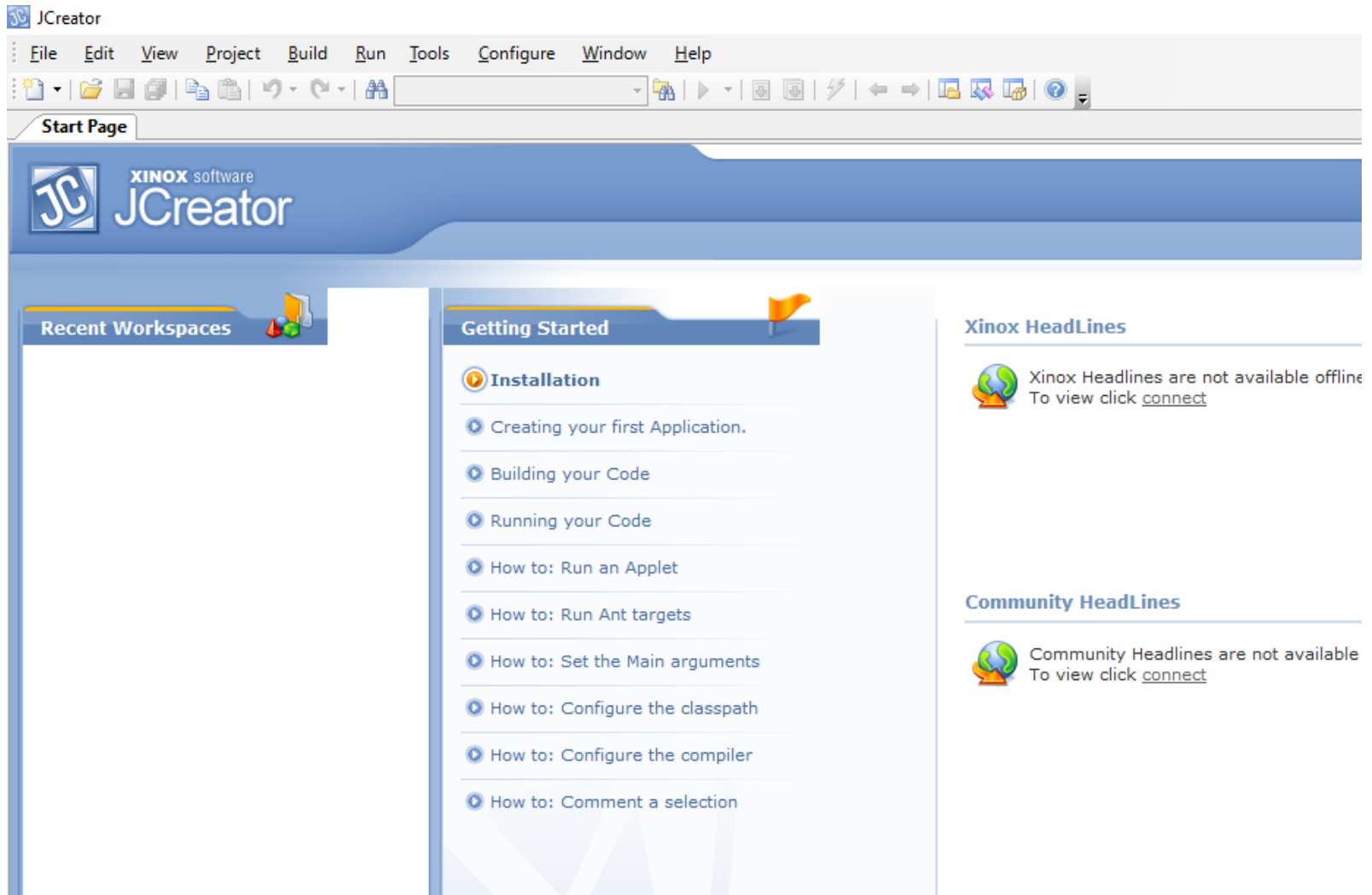
float valorTemperatura;

boolean saldoNegativo,contaPaga,contaEncerrada;

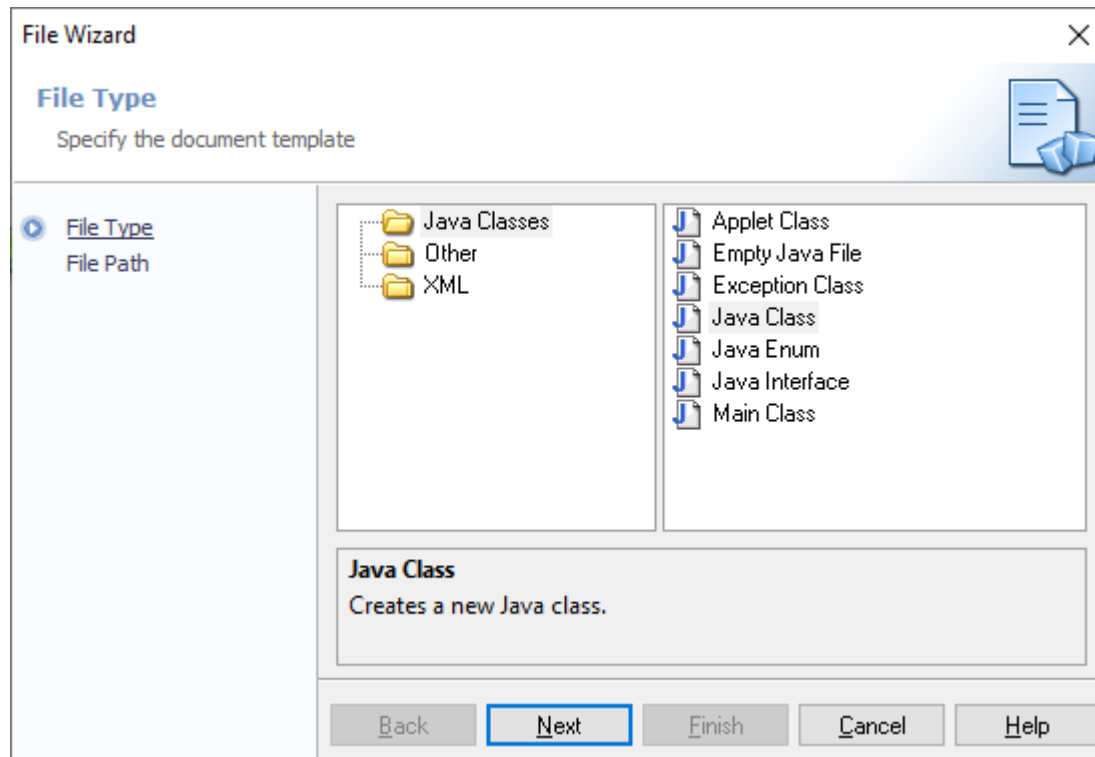
int valorToatalSalario=0;

char minhaLetra='A';

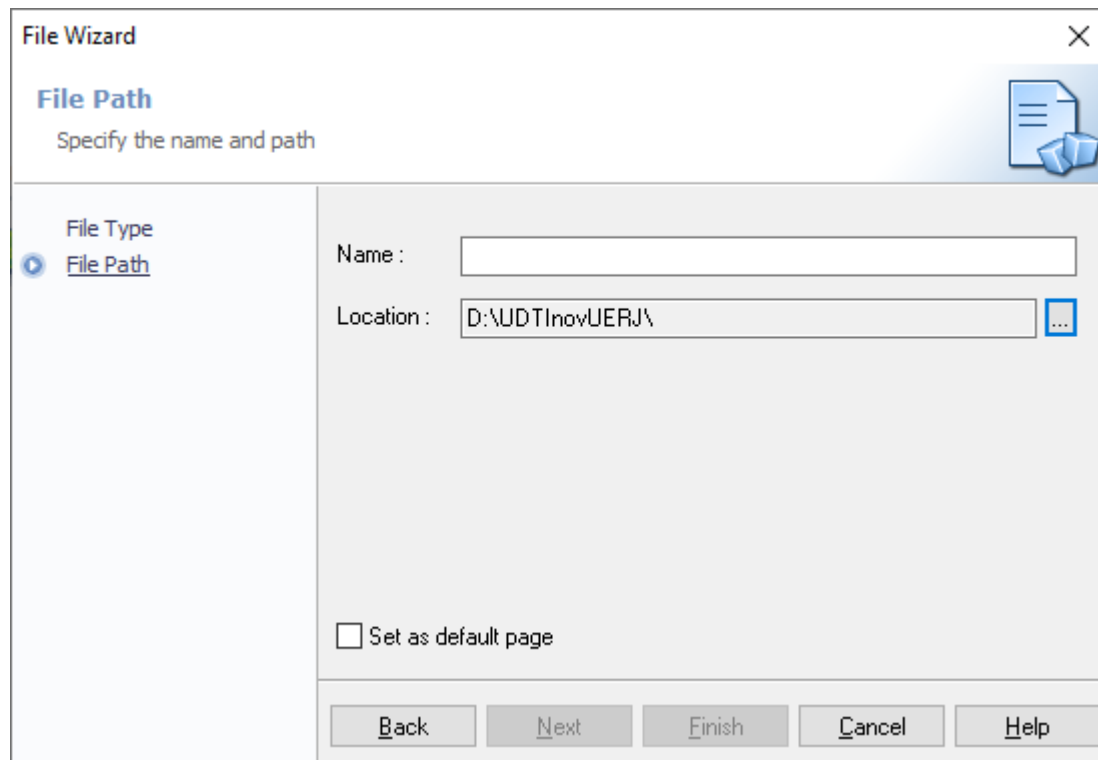
long valorTotalSalario,valorTotalDesconto=0;



- Sequência: File > New > File:



- Sequência: Java Classes > Main Class:



The image shows a 'File Wizard' dialog box with a close button (X) in the top right corner. The title bar reads 'File Wizard'. Below the title bar, the text 'File Path' is displayed in blue, followed by the instruction 'Specify the name and path'. On the left side, there are two radio buttons: 'File Type' and 'File Path', with 'File Path' being selected. The main area contains two text input fields: 'Name :' and 'Location :'. The 'Location :' field contains the text 'D:\UDTInovUERJ\'. To the right of the 'Location :' field is a small blue square button with three dots. At the bottom left, there is a checkbox labeled 'Set as default page' which is currently unchecked. At the bottom right, there are five buttons: 'Back', 'Next', 'Finish', 'Cancel', and 'Help'.

# Tipos de comentários



■ Podem ser inseridos de duas maneiras distintas;

- Digitando-se duas barras antes do texto comentário `//`;
- Utilizando-se de uma dupla barra-asterisco `/*`, abrangendo um conjunto de linhas de comentário;

■ Exemplos:

```
fator= 100; // Atribui o valor 100 a variável fator
```

```
/* A seguir o trecho da série
```

```
de Fibonacci que será
```

```
utilizada em nosso trabalho */
```

**Vamos praticar** : código `Usando_Variaveis_e_Underscore.java`



# Operadores Aritméticos



## Operadores aritméticos da linguagem:

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

# Operadores Aritméticos



■ Exemplos:

**Vamos praticar** : código Operadores\_Aritmeticos.java

código Mais\_Operadores\_Aritmeticos.java

# Operadores Relacionais



## ■ Operadores Relacionais da linguagem:

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

# Operadores Condicionais



## Operadores Condicionais da linguagem:

- Retorna um valor booleano indicando se o resultado da operação é *true* ou *false*;

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

# Operadores de Atribuição Atalhos



## Operadores de Atribuição - Atalhos:

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

# Definindo constantes



■ Utilizado para definir uma dado que não pode ser alterado durante a execução do código;

■ Sintaxe:

– **final** tipo-do-dado nome-da-constante = valor-da-constante;

Exemplo:

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

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

# Escrevendo uma classe



- Deverá ter o sufixo .java;
- Nome da classe é igual ao nome do arquivo;
- Pode conter n métodos;
- Sintaxe:

```
class nome-da-classe{  
    public static void main(String[] args) {  
        código  
    }  
}
```

# Apresentando dados System.out



- Classe utilizado para apresentar dados;
- Possui os métodos print, println e printf;

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

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



# Apresentando dados System.out



## ■ Sintaxe do printf:

- `System.out.printf([indicadores-do-formato,]lista-de-variaveis);`
- Cada indicador do formato deve ser precedido pelo caracter %;
- Exemplo:

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

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

# Lendo dados do teclado



- Funcionalidade oferecida pela classe Scanner;
- Instanciar a classe Scanner especificando a origem dos dados a serem lidos;
- A classe Scanner suporta também a leitura de dados de arquivos;
- Invocar o método `next()`, especificando o tipo do dado a ser lido;
- Deve ser importado o pacote `java.util.Scanner`;

# Lendo dados do teclado Scanner



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

■ Exemplo:

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

**Vamos praticar!!!**

# Manipulando *strings*



- Dado que tem a capacidade de abrigar qualquer tipo de caracter incluindo números, letras e símbolos;
- Java não possui o tipo string;
- Disponibilizada a classe String;
- Possui métodos que realizam diversas operações;
- Como uma classe, precisa ser instanciada:
  - `String nomeDoLivro = new String();`
  - `String nomeDoLivro;`
  - `String nomeDoLivro = new String("Java como Programar");` ou
  - `String nomeDoLivro = "Java como Programar";`

# Manipulando strings

## Alguns métodos



<i>Método</i>	<i>Como funciona</i>
<code>charAt()</code>	Retorna qual caractere encontra-se em uma posição específica da string.
<code>concat()</code>	Concatena duas strings.
<code>endsWith()</code>	Retorna true se a string termina com determinado sufixo.
<code>equals()</code>	Retorna true se as duas strings comparadas têm o mesmo tamanho e são exatamente iguais.
<code>equalsIgnoreCase()</code>	Retorna true se as duas strings comparadas têm o mesmo tamanho e são exatamente iguais. Será ignorada a diferença entre maiúsculas e minúsculas.
<code>indexOf()</code>	Retorna um número inteiro informando a primeira posição em que se encontra um caractere ou uma substring em uma string.
<code>length()</code>	Retorna o tamanho da string. São considerados os espaços em branco no início e no fim da string, se existirem
<code>lastIndexOf()</code>	Retorna um número inteiro informando a última posição em que se encontra um caractere ou uma substring em uma string.
<code>regionMatches()</code>	Retorna true se determinada região de uma string é igual a uma determinada região de outra string
<code>replace()</code>	Substitui todos os caracteres de uma string por outro caractere.
<code>startsWith()</code>	Retorna true se a string inicia com determinado prefixo
<code>substring()</code>	Copia para outra string uma porção da string a partir de determinada posição.
<code>toLowerCase()</code>	Retorna uma nova string com cada caractere sendo convertido para o correspondente minúsculo.
<code>toUpperCase()</code>	Retorna uma nova string com cada caractere sendo convertido para o correspondente maiúsculo.
<code>trim()</code>	Retorna uma string sem espaços em seu início e em seu fim

Vamos praticar, detalhes em <https://docs.oracle.com/javase/7/docs/api/java/lang/String.html>

# Estrutura de fluxo, controle ou de seleção



- Utilizada para desviar a sequência de execução das instruções que integram um programa;
- Este desvio pode ser feito como resultado de uma comparação de uma expressão ou de um valor de determinada variável;
- Em Java existem três tipos de instruções de seleção: a instrução **if**; a instrução **if...else**; a instrução **switch**;

## ■ Sintaxe do **if**:

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

# Estrutura de fluxo, controle ou de seleção



## ■ Exemplos:

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

```
if ( notaFinal == 10) {
```

```
    System.out.println("Aluno Aprovado");
```

```
    System.out.println("Aluno Muito Bom");
```

```
    System.out.println("Dar bolsa para o aluno);
```

```
}
```

# Estrutura de fluxo, controle ou de seleção



## ■ Instruções **if...else**:

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

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



# Estrutura de fluxo, controle ou de seleção



## ■ Instruções **if...else** aninhados:

```
if (salario <= 1200)
    System.out.println("não há desconto de IR");
else if (salario <= 1500)
    System.out.println("desconto de IR de 10%");
else if (salario <= 2500)
    System.out.println("desconto de IR de 20%");
else
    System.out.println("desconto de IR de 27.5%");
```

# Estrutura de fluxo, controle ou de seleção



■ A instrução **switch** é classificada como uma instrução de seleção múltipla pelo fato de permitir que diferentes ações possam ser tomadas em função do valor assumido por alguma variável ou por uma expressão;

■ Aceita os tipos primitivos byte, short, char, and int e também String e Enum;

■ Sintaxe:

```
switch (expressão ou variável){  
    case valor-a-ser-testado1:  
        instrução 1;  
        instrução 2;  
        instrução n;  
        break;  
    case valor-a-ser-testado2:  
        instrução 1;  
        instrução n;  
        break;  
    default:  
        instrução 1;  
        instrução n;  
}
```

# Estrutura de fluxo, controle ou de seleção



- O comando *break* faz com que as comparações sejam interrompidas;
- O comando *default*, que é de uso opcional, será tratado somente se todas as comparações anteriores forem insatisfeitas;

# Estrutura de fluxo, controle ou de seleção



■ Exemplo, código para verificar mês digitado:

```
switch (mes){  
    case 1: System.out.println("Janeiro");break;  
    case 2: System.out.println("Fevereiro");break;  
    case 3: System.out.println("Março");break;  
    case 4: System.out.println("Abril");break;  
    case 5: System.out.println("Maio");break;  
    case 6: System.out.println("Junho");break;  
    case 7: System.out.println("Julho");break;  
    case 8: System.out.println("Agosto");break;  
    case 9: System.out.println("Setembro");break;  
    case 10: System.out.println("Outubro");break;  
    case 11: System.out.println("Novembro");break;  
    case 12: System.out.println("Dezembro");break;  
    default: System.out.println("Mês Errado");  
}
```

# Estrutura de fluxo, controle ou de seleção



■ Exemplo, código para indicar quantos dias determinado mês possui:

```
switch (mes){
    case 1:
    case 3:
    case 5:
    case 7:
    case 8:
    case 10:
    case 12:
        System.out.println("Este mês possui 31 dias");
        break;
    case 4:
    case 6:
    case 9:
    case 11:
        System.out.println("Este mês possui 30 dias");
        break;
    case 2:
        if ( ano % 4 == 0)
            System.out.println("Este mês possui 29 dias");
        else
            System.out.println("Este mês possui 28 dias");
            break;
    default: System.out.println("Valor Errado");
}
```

# Estrutura de fluxo, controle ou de seleção



- Executar as listas de exercícios 1 e 2;

# Estrutura de repetição



- A estrutura de repetição é utilizada para repetir, de forma controlada, a execução de determinado conjunto de instruções que integram o trecho de um programa;
- Na linguagem Java existem três tipos de instruções de repetição: a instrução **for**; a instrução **while**; e a instrução **do...while**;
- A instrução **for** executa repetidamente um determinado conjunto de instruções;
- Este processo de repetição é denominado **loop**;
- Sintaxe do **for**:

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

# Estrutura de repetição



- O argumento valor-inicial especifica o valor de partida para o número de repetições do conjunto;
- O argumento condição-de-repetição estabelece a condição com que o número de repetições ficará delimitado;
- O argumento incremento-ou-decremento-do-valor-inicial estabelece o valor que será incrementado ou decrementado do valor inicialmente estipulado;
- A instrução **break**, quando utilizada, interrompe o ciclo de repetições;
- A instrução **continue** salta todas as instruções que a sucedem retornando o controle para a próxima interação;
- No Java 1.5 o comando **for** foi aprimorado, fazendo um tratamento especial para a manipulação de *arrays* e de coleções, como veremos em seções posteriores;
- Exemplos:

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



# Estrutura de repetição



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

```
while (condição-a-ser-testada) {
```

```
    (condicional) continue;
```

```
    instrução 1;
```

```
    instrução 2;
```

```
    instrução n;
```

```
    break; (opcional)
```

```
}
```

# Estrutura de repetição



■ A instrução ***break***, quando utilizada, interrompe o ciclo de repetições;

■ A instrução ***continue*** salta todas as instruções que a sucedem retornando o controle para a próxima interação;

■ Exemplo:

```
while(notaAluno>=0) {  
    totalNotas+= notaAluno;  
    System.out.println("Digite a nota:");  
    notaAluno = entrada.nextInt();  
}  
while (true){           // Loop Infinito  
    // Código aqui  
}
```

# Estrutura de repetição



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

■ Sintaxe do **while**:

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

# Estrutura de repetição



## ■ Exemplo 1:

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

## ■ Exemplo 2:

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

- **Vamos praticar:** código Estrutura\_de\_Repeticao

- É uma área de memória em que pode ser armazenado um grupo fixo e homogêneo de elementos;
- Estes elementos podem ser dados primitivos ou referências para objetos;
- Para declarar um *array* necessário saber o número de seus elementos e seus tipos;
- Para localizar os elementos são utilizados índices;
- *Array* de única dimensão pode suportar até 2.147.483.647 elementos;
- O índice inicia com valor zero;
- Uma vez declarado, o tamanho do *array* não pode ser modificado;
- *Arrays* de tipos primitivos, não iniciados, **assume o valor inicial zero**;
- *Arrays* de tipos booleanos, não iniciados, **assume o valor inicial *false***;

■ Em Java, *array* é uma classe e precisa ser instanciada;

■ Sintaxe para declarar um *array* de tipos primitivos:

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

■ Exemplos:

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

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

■ O atributo *length*, da classe *array*, permite saber o número total de elementos nele contidos;

■ Exemplo:

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

# Array de Strings



- A declaração somente cria um *array* de referências para as *strings*;
- Todos os métodos da classe *String* podem ser utilizados, tais como: `trim()`, `substring()`, `indexOf()` etc;
- Exemplos:

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



# Array de arrays



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

■ Sintaxe para declarar um *array* multidimensional:

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

■ Formas de declarar:

```
String [][] tabela = new String[3][2]; // ou
String [][]tabela = new String[3][];
tabela [0] = new String[2];
tabela [1] = new String[2];
tabela [2] = new String[2];
```

- Se desejar iniciar o array:

```
tabela [0] = new String[]{"alice","paulo"};
tabela [1] = new String[] {"anna", "kiko"};
tabela [2] = new String[] {"alice","marcelo"};
```

# Array de arrays



## ■ Exemplo:

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

# A classe *Arrays*



- A classe *Arrays* faz parte do pacote `java.util.Arrays`;
- Fornece um conjunto de métodos estáticos que permite comparar, classificar e preencher *arrays*;

**`Arrays.equals`**(nome-do-array1,nome-do-array2);

- Utilizado para comparar o conteúdo de dois *arrays*;
- Aplicável para tipos primitivos e para *strings*;

■ Exemplo 01 – comparando arrays:

```
import java.util.Arrays;
public class TestaArrays{
    public static void main(String[] args){
        String []tabela1 = new String[3];
        tabela1 [0] = new String ("amelia");
        tabela1 [1] = new String ("anna");
        tabela1 [2] = new String ("alice");
        String []tabela2 = new String[3];
        tabela2 [0] = new String ("amelia");
        tabela2 [1] = new String ("anna");
        tabela2 [2] = new String ("alice");
        if (Arrays.equals(tabela1,tabela2))
            System.out.println("São iguais");
    }
}
```

# A classe *Arrays*



■ **Arrays.fill**(nome-do-array,dado-de-preenchimento);

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

■ Exemplo02 – preenchendo arrays:

```
import java.util.Arrays;
```

```
public class TestaArrays{  
    public static void main(String[] args){  
        String []tabela1 = new String[3];  
        tabela1 [0] = new String ("amelia");  
        tabela1 [1] = new String ("anna");  
        tabela1 [2] = new String ("alice");  
        Arrays.fill(tabela1,"maria");  
        for (int i=0; i<tabela1.length ; i++){  
            System.out.println(tabela1[i]);  
        }  
    }  
}
```

# A classe *Arrays*



■ **Arrays.sort**(nome-do-array[,indice-inicial-inclusive,[indice-final-exclusive]]);

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

■ Exemplo03 – classificando arrays:

```
import java.util.Arrays;
```

```
public class TestaArrays{  
    public static void main(String[] args){  
        int[] tabela1 = {10,3,80,6,7,-3};  
        Arrays.sort(tabela1);  
        for (int i=0; i<tabela1.length ; i++){  
            System.out.println(tabela1[i]);  
        }  
    }  
}
```

# A classe *Arrays*



■ Exemplo04 - classificando partes de uma array:

```
import java.util.Arrays;

public class TestaArrays{
    public static void main(String[] args){
        int[] tabela1 = { 54, 9, 18, 44, 26, 5, 99, 220,580,4 };

        Arrays.sort(tabela1, 1, 6);

        for (int i=0; i<tabela1.length ; i++){
            System.out.println(tabela1[i]);
        }
    }
}
```

# Pacotes, Acessibilidade e Construtores



## ■ Pacotes:

- Na linguagem Java todo o código existente é armazenado em classes;
- Na API (Application Programming Interface) do Java grupos de classes são organizados em pacotes;
- Apenas as classes armazenadas no pacote `java.lang` são automaticamente disponibilizadas, dispensando-nos da sua importação;
- Aqueles pacotes que não integram o grupo pertencente ao `java.lang` precisam ser importados, caso as classes que os integram precisam ser utilizadas;
- Um pacote pode ser definido como sendo um diretório onde residirão as classes do aplicativo que iremos desenvolver;
- Sintaxe de importação:
  - **`import nome-do-pacote.Nome-da_Classe;`**     `// ou`
  - **`import nome-do-pacote.*;`**

# Pacotes, Acessibilidade e Construtores



## ■ Pacotes:

- A boa prática de POO recomenda que as classes que integram uma aplicação devam ser organizadas em pacotes;
- Os seguintes passos devem ser seguidos:
  - Criar um diretório para o pacote. O nome do diretório é o mesmo nome do pacote;
  - Adicionar o comando ***package*** que identifica o pacote das classes no início do código de cada classe que integra o pacote;
  - Sintaxe do ***package***:
    - **package nome-do-pacote;**



# Pacotes, Acessibilidade e Construtores



## ■ Pacotes:

- Exemplo:

```
package UERJ.CursoJava; // Diretório e subDiretório
```

```
import java.util.Arrays;
```

```
public class TestaArrays{  
    public static void main(String[] args){  
        int[] tabela1 = { 54, 9, 18, 44, 26, 5, 99, 220,580,4 };  
  
        Arrays.sort(tabela1, 1, 6);  
  
        for (int i=0; i<tabela1.length ; i++){  
            System.out.println(tabela1[i]);  
        }  
    }  
}
```

# Pacotes, Acessibilidade e Construtores



## ■ Acessibilidade:

- A acessibilidade de uma classe, de um método ou de um atributo especifica como estes elementos serão ou não vistos e utilizados por outras classes;
- A acessibilidade vem formalizar um dos paradigmas da POO que é o **encapsulamento de dados**;
- Os atributos e métodos de determinada classe podem ser ocultados de qualquer outra classe;
- A acessibilidade é controlada pelas palavras-chave ***public***, ***private*** e ***protected*** que também são chamadas de **modificadores de acesso**.
- Modificador ***public***:
  - Aplica-se às classes, métodos e atributos. Permite que qualquer outra classe, **de qualquer pacote**, tenha acesso a estes elementos;
- Modificador ***private***:
  - Aplica-se às classes, métodos e atributos. Permite que apenas a própria classe tenha acesso aos seus elementos, nenhuma classe do pacote pode acionar um elemento definido como private;

# Pacotes, Acessibilidade e Construtores



## ■ Acessibilidade:

### – Modificador ***protected***:

- Aplica-se aos métodos, atributos e construtores. Permite que apenas as classes do mesmo pacote tenham acesso a estes elementos;
- Para obter acesso a um modificador *protected*, fora do pacote, então é necessário herdar da classe protegida;
- Proteger um construtor impede que os usuários criem uma instância da classe, fora do pacote;

# Pacotes, Acessibilidade e Construtores



## ■ Acessibilidade:

### – Exemplos:

```
private salario;  
protected salario;  
public salario;
```

```
private int calculaDesconto(float salario){  
.....  
}
```

```
protected int calculaDesconto(float salario){  
.....  
}  
public int calculaDesconto(float salario){  
.....  
}
```

# Pacotes, Acessibilidade e Construtores



## ■ Construtores:

- Para criar uma instância de uma determinada classe utiliza-se o operador ***new***;
- Quando encontra este operador a JVM irá procurar, no código da classe que está sendo instanciada, um método denominado método construtor que é efetivamente quem dará vida ao objeto;
- É alocada a memória necessária para instanciar o objeto;
- Neste método podem ser iniciadas as variáveis de instância, podem ser acionados métodos da própria classe, e também podem ser instanciadas outras classes;
- Um construtor deve ser definido como ***public*** ou ***protected*** e possuir o mesmo nome da classe que o define;
- Pode existir mais de um construtor em uma única classe desde que possua assinaturas diferentes;
- Caso o método construtor não seja codificado, a JVM assumirá a existência de um construtor *default*;

# Pacotes, Acessibilidade e Construtores



## ■ Construtores:

- O construtor *default* iniciará todas as variáveis de instância do tipo numérico com o valor zero, as variáveis lógicas receberão o valor *false* e as variáveis objeto são iniciadas como *null*;
- Exemplo, comprovando a iniciação de uma variável de instância:

```
public class PrimeiraLetra {  
    private int numero;    // Variável de instância  
    public void testa(){  
        System.out.println(numero);  
    }  
}  
  
public class TestaConstrutor{  
    public static void main(String args[]) {  
        PrimeiraLetra entrada = new PrimeiraLetra();  
        PrimeiraLetra entrada;    // outra  
        entrada = new PrimeiraLetra(); // maneira  
        entrada.testa();  
    }  
}
```

# Pacotes, Acessibilidade e Construtores



## ■ Construtores:

- Exemplo, classe com dois construtores:

```
public class PrimeiraLetra{
    private float altura,largura;

    public PrimeiraLetra(){
    }
    public PrimeiraLetra(float alt, float larg){
        altura = alt;
        largura = larg;
    }
    public void testa() {
        System.out.printf("%s%.2f%s%.2f\n","Valores fornecidos ", altura, " e ",largura);
    }
}

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

# Pacotes, Acessibilidade e Construtores



## ■ Construtores:

- Exemplo, classe com dois construtores:

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



# Variáveis de Instância



- As variáveis que são definidas **no interior de um método**, ou de de um bloco de instruções, são denominadas **variáveis locais**;
- Este tipo de variável é visível apenas para o método, ou bloco, que as contém, outros métodos da classe não tem acesso às variáveis locais deste método;
- As **variáveis de instância** são aquelas que **são definidas no corpo da classe** e não no corpo dos métodos;
- Têm como característica o fato de serem visíveis para todos aqueles métodos que integram a classe;
- Normalmente as variáveis de instância são utilizadas para preservar valores comuns aos métodos **onde podem ser modificadas ou recuperadas**;

# Métodos Codificação



## ■ Sintaxe:

```
modificador tipo_retornado(ou void) nome-do-método([tipo1 parametro1[,...[,tipoN parametroN]]){  
    ....  
    ....  
    código  
    ...  
    return valor; // se tipo_retornado != void  
}
```

■ Para o nome dos métodos usa-se, por padrão, o camelCase;

## ■ Exemplo:

```
private String nome; // variável da instância  
  
public String getNome() {  
    return this.nome;  
}  
public void setNome(String nome) {  
    this.nome = nome;  
}
```

# Métodos Codificação



■ Métodos podem ter várias assinaturas(*overloading*);

■ Exemplo:

```
public class Desenhar {  
    ...  
    public void draw(String s) {  
        ...  
    }  
    public void draw(int i) {  
        ...  
    }  
    public void draw(double f) {  
        ...  
    }  
    public void draw(int i, double f) {  
        ...  
    }  
}
```

# Métodos *get* e *set*



■ Para modificar ou recuperar os valores de uma variável de instância utiliza-se, **segundo a boa prática da POO**, métodos distintos;

■ Os métodos que modificam o conteúdo das variáveis devem ter seu nome precedido pela palavra **set**;

■ Os métodos que recuperam os valores devem ser precedidos pela palavra **get**;

■ Então, um método que recupera a nota de um aluno deve chamar-se, por exemplo, **getNota**;

■ O método que modifica a nota deve chamar-se **setNota**;

■ Exemplo:

```
private float nota;  
  
public float getNota(){  
    return this.nota;  
}  
  
public void setNota(float nota){  
    this.nota = nota;  
}
```

Obs.: A palavra-chave *this* refere-se ao objeto atual em um método ou construtor.  
O uso mais comum da palavra-chave *this* é eliminar a confusão entre atributos da classe e parâmetros com o mesmo nome.

# Métodos *get* e *set*



## Exemplo

```
public class Lance {  
    private int id;  
    private int idCliente;  
    private int idLeilao;  
    private double valorLance;  
  
    public int getId(){  
        return this.id;  
    }  
    public void setId(int id){  
        this.id=id;  
    }  
    public int getIdCliente(){  
        return this.idCliente;  
    }  
    public void setIdCliente(int idCliente){  
        this.idCliente=idCliente;  
    }  
    public int getIdLeilao(){  
        return this.idLeilao;  
    }  
    public void setIdLeilao(int idLeilao){  
        this.idLeilao=idLeilao;  
    }  
    public double getValorLance(){  
        return this.valorLance;  
    }  
    public void setValorLance(double valorLance){  
        this.valorLance=valorLance;  
    }  
}
```

# Exercícios



■ Fazer lista de Exercícios 3(pode ser em dupla!);

# Métodos estáticos



- São aqueles associados a uma classe em vez de a um objeto;
- Uma razão para usar métodos estáticos é quando reutilizamos um comportamento padrão em instâncias de diferentes classes;
- Utilizados quando não é necessário saber o estado do objeto;
- A classe não precisa ser instanciada;
- Métodos estáticos não podem ser alterados;
- São declarados usando a palavra-chave ***static***;
- Sintaxe:

```
modificador static tipo_retornado(ou void) nome-do-método(tipo1 parametro1,...,tipoN parametroN){  
    código  
    return valor; // se tipo_retornado != void  
}
```

# Métodos estáticos



- Métodos estáticos não podem acessar ou alterar os valores das variáveis de instância;
- Podem acessar ou alterar os valores das variáveis estáticas;
- Não podem chamar métodos não estáticos;
- Uma classe não estática pode conter métodos e propriedades estáticas;
- O elemento estático pode ser chamado em uma classe mesmo quando nenhuma instância da classe foi criada;
- Uma variável estática é compartilhada por todas as instâncias de sua classe;



# Métodos estáticos



## Exemplo com variável *static*;

```
public class Temperature{  
    private double temperature;  
  
    public static double maxTemp = 0;  
  
    public Temperature(double t)  {  
        temperature = t;  
        if (t > maxTemp)  
            maxTemp = t;  
    }  
  
    public double getTemperature(){  
        return this.temperature;  
    }  
  
    public static void main(String[] args)  {  
  
        Temperature t1 = new Temperature(75);  
        Temperature t2 = new Temperature(100);  
        Temperature t3 = new Temperature(65);  
  
        System.out.println("Temp: " + t1.getTemperature());  
        System.out.println("Temp: " + t2.getTemperature());  
        System.out.println("Temp: " + t3.getTemperature());  
        System.out.println("Max Temp: " + Temperature.maxTemp);  
    }  
}
```

**Questão: O que será impresso?**

# Métodos estáticos



## Exemplo com método *static*;

```
class Alienado {  
    public static String nomeAlienado = "";  
    public static void alien(String nome)  {  
        nomeAlienado = nome;  
    }  
}  
  
class TestaAlienado {  
    public static void main(String[] args)  {  
        Alienado.alien("Jojojo");  
        System.out.println(Alienado.nomeAlienado);  
        Alienado obj = new Alienado();  
        obj.alien("Jajaja");  
        System.out.println(obj.nomeAlienado);  
    }  
}
```

# Herança



- Capacidade que uma classe possui de herdar atributos ou estados e métodos ou comportamentos de outra classe;
- Todos os atributos e métodos definidos como ***public*** ou ***protected*** podem ser alterados por qualquer outra classe;
- É usada a palavra-chave ***final*** em uma declaração de método para indicar que o método não pode ser substituído por subclasses;
- A classe que disponibiliza seus atributos e métodos é chamada de **superclasse**;
- A classe que herda é chamada de **subclasse**;
- O Java somente suporta herança simples;
- Sintaxe:

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

- Os campos herdados podem ser usados diretamente, assim como qualquer outro campo;
- Você pode declarar um campo na subclasse com o mesmo nome da superclasse (não recomendado);
- Você pode declarar novos campos na subclasse que não estão na superclasse;
- Os métodos herdados podem ser usados diretamente como estão;
- Você pode escrever um novo método de instância na subclasse que tenha a mesma assinatura da superclasse, substituindo-o;
- Você pode escrever um novo método estático na subclasse que tenha a mesma assinatura da superclasse;
- Você pode declarar novos métodos na subclasse que não estão na superclasse;
- Você pode escrever um construtor de subclasse que invoque o construtor da superclasse, implicitamente ou usando a palavra-chave `super`;

# Herança



## Exemplo:

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

```
public class HerdaPrincipal extends ClassePrincipal {
    public static void main(String[] args){
        HerdaPrincipal objeto = new HerdaPrincipal ();
        objeto.calculaArea(10,20);
        System.out.printf("%s%.2f\n", "Area calculada:", objeto.getArea());
    }
}
```

```
public class HerdaPrincipal extends ClassePrincipal { // Aqui o método herdado é modificado
    public void calculaArea(float a, float b){
        area= b - a; // Código modificado
    }
    public static void main(String[] args){
        HerdaPrincipal objeto = new HerdaPrincipal ();
        objeto.calculaArea(10,20);
        System.out.printf("%s%.2f\n", "Area calculada:", objeto.getArea());
    }
}
```

## Exemplo:

```
public class Bicycle {  
  
    public int cadence;  
    public int gear;  
    public int speed;  
  
    public Bicycle(int startCadence, int startSpeed, int startGear) { // Construtor da superclasse  
        gear = startGear;  
        cadence = startCadence;  
        speed = startSpeed;  
    }  
  
    public void setCadence(int newValue) {  
        cadence = newValue;  
    }  
  
    public void setGear(int newValue) {  
        gear = newValue;  
    }  
  
    public void applyBrake(int decrement) {  
        speed -= decrement;  
    }  
  
    public void speedUp(int increment) {  
        speed += increment;  
    }  
}
```

## ■ Exemplo (continuação):

```
public class MountainBike extends Bicycle {  
  
    public int seatHeight;  
  
    public MountainBike(int startHeight, int startCadence, int startSpeed, int startGear) {  
        super(startCadence, startSpeed, startGear); // Herdando o construtor  
    }  
  
    public void setHeight(int newValue) {  
        this.seatHeight = newValue;  
    }  
}
```

Exemplo real da UDT, herdando AcessoBanco.

# Polimorfismo



- Conceitualmente podemos dizer que polimorfismo é “o que se apresenta sob numerosas formas”;
- Na linguagem Java o polimorfismo se apresenta tão somente nas chamadas de seus métodos;
- Um método de mesmo nome pode retornar resultados de diversas formas, e quem decide as formas é o objeto que contém o método;



# Polimorfismo



## Exemplo 01:

```
public class Animal{
    public void emitirSom(){
        System.out.println("Emitindo Som Generico");
    }
}
public class Cachorro extends Animal{
    public void emitirSom(){
        System.out.println("Latindo....");
    }
}
public class Gato extends Animal{
    public void emitirSom(){
        System.out.println("Miando....");
    }
}
public class Vaca extends Animal{
    public void emitirSom(){
        System.out.println("Mugindo");
    }
}
```

# Polimorfismo



## ■ Exemplo 01 (continuação):

```
public class TestaAnimal{  
  
    public static void main(String[] args){  
  
        Animal bicho1 = new Gato();  
        bicho1.emitirSom();  
  
        Animal bicho2 = new Cachorro();  
        bicho2.emitirSom();  
  
        Animal bicho3 = new Vaca();  
        bicho3.emitirSom();  
  
    }  
}
```

# Polimorfismo



## Exemplo 02:

```
public class Veterinario{  
    public Veterinario(Animal animal){  
        animal.emitirSom();  
    }  
}  
  
public class TestaVeterinario{  
  
    public static void main(String[] args){  
        Animal animal = new Gato();  
        Veterinario vet = new Veterinario(animal);  
        animal = new Cachorro();  
        vet = new Veterinario(animal);  
    }  
}
```

# Classes e Métodos Abstratos



- Uma classe abstrata serve como um modelo que pode ser herdado pelas subclasses;
- Uma classe abstrata pode conter métodos abstratos e métodos não abstratos;
- Uma classe abstrata não pode ser instanciada, apenas herdada;
- Quando ela define métodos abstratos as suas subclasses **precisam implementar estes métodos**;
- Um método abstrato não contém um corpo, só a assinatura;
- Os métodos de uma classe abstrata podem ser do tipo *public* ou do tipo *protected*;
- Se uma classe inclui métodos abstratos ela deve ser declarada abstrata;

# Classes e Métodos Abstratos



■ Exemplo:

```
abstract class GraphicObject {
    int x, y;
    ...
    void moveTo(int newX, int newY) {
        ...
    }
    abstract void draw();
    abstract void resize();
}

class Circle extends GraphicObject {
    void draw() {
        ...
    }
    void resize() {
        ...
    }
}

class Rectangle extends GraphicObject {
    void draw() {
        ...
    }
    void resize() {
        ...
    }
}
```

# Interface



- Pode ser definida como sendo um contrato onde todas as classes que a implementam têm que codificar métodos nela definidos;
- Possui o sufixo .java;
- Determinam comportamentos que a classe implementa;
- Não é instanciada, é implementada;
- Uma classe pode implementar **n** interfaces;
- Uma interface pode herdar de outras interfaces;
- Pode conter constantes, assinaturas de métodos(abstratos), métodos *default*, métodos estáticos e tipos aninhados;
- Os métodos estáticos contém conteúdo;
- Métodos *default* contém conteúdo e podem ou não ser implementados;
- Sintaxe:

```
public interface nome-da-interface{  
    constantes e métodos...  
}  
public class nome-da-classe implements nome-da-interface{} // Classe implementando
```

# Interface



## Exemplo 01 – implemetando conteúdo:

```
public interface OperateCar {
    int turn(Direction direction, double radius, double startSpeed, double endSpeed);
    int changeLanes(Direction direction, double startSpeed, double endSpeed);
    int signalTurn(Direction direction, boolean signalOn);
    .....
    // Mais assinaturas de métodos
}

public class OperateBMW760i implements OperateCar {
    public int turn(Direction direction, double radius, double startSpeed, double endSpeed) {
        // implementação
    }
    public int changeLanes(Direction direction, double startSpeed, double endSpeed) {
        // implementação
    }
    public int signalTurn(Direction direction, boolean signalOn) {
        // implementação
    }
    // Mais Implementações
}
```

# Interface



## Exemplo 02 – utilizando constantes:

```
interface OlympicMedal {  
    static final String GOLD = "Gold";  
    static final String SILVER = "Silver";  
    static final String BRONZE = "Bronze";  
}  
  
public final class OlympicAthlete implements OlympicMedal {  
  
    private String medal;  
  
    public OlympicAthlete(int id){  
        //..  
    }  
  
    //..  
  
    public void winEvent(){  
        medal = GOLD;  
    }  
  
}
```



# Interface



## Exemplo 03 – utilizando métodos estáticos:

```
interface PrintDemo {  
    static void hello() {  
        System.out.println("Called from Interface PrintDemo");  
    }  
}  
  
public class InterfaceDemo implements PrintDemo {  
    public static void main(String[] args) {  
        PrintDemo.hello();  
        hello("Epa!!!");  
    }  
  
    static void hello(String texto){  
        System.out.println("Called from Class");  
        System.out.println(texto);  
    }  
}
```

# Interface



## Exemplo 04 – utilizando o método *default*:

```
interface Person {
    String getName();
    default void greet() {
        System.out.println("Hello");
    }
}

class John implements Person { // Implementou Person mas não invocou/modificou greet()
    public String getName() {
        return "John";
    }
}

class Mary implements Person { // Implementou Person e modificou greet()
    public String getName() {
        return "Mary";
    }
    public void greet() {
        System.out.println("Hi");
    }
}

public class Padrao {
    public static void main(String[] args) {
        John john = new John();
        Mary mary = new Mary();
        System.out.print(john.getName() + " says: ");
        john.greet();
        System.out.print(mary.getName() + " says: ");
        mary.greet();
    }
}
```

# Interface Usada como um tipo



## ■ Exemplo 05 –

```
interface MinhaInterface {  
    void callback(int param);  
}
```

```
class Cliente implements MinhaInterface{  
    public void callback(int parametro) {  
        System.out.println(parametro);  
    }  
}
```

```
public class Principal {  
    public static void main(String args[]) {  
        MinhaInterface minha = new Cliente();  
        minha.callback(18);  
    }  
}
```

# Exercícios



- Executar a lista de Exercícios 4(pode ser em dupla!);

# Classe Enum



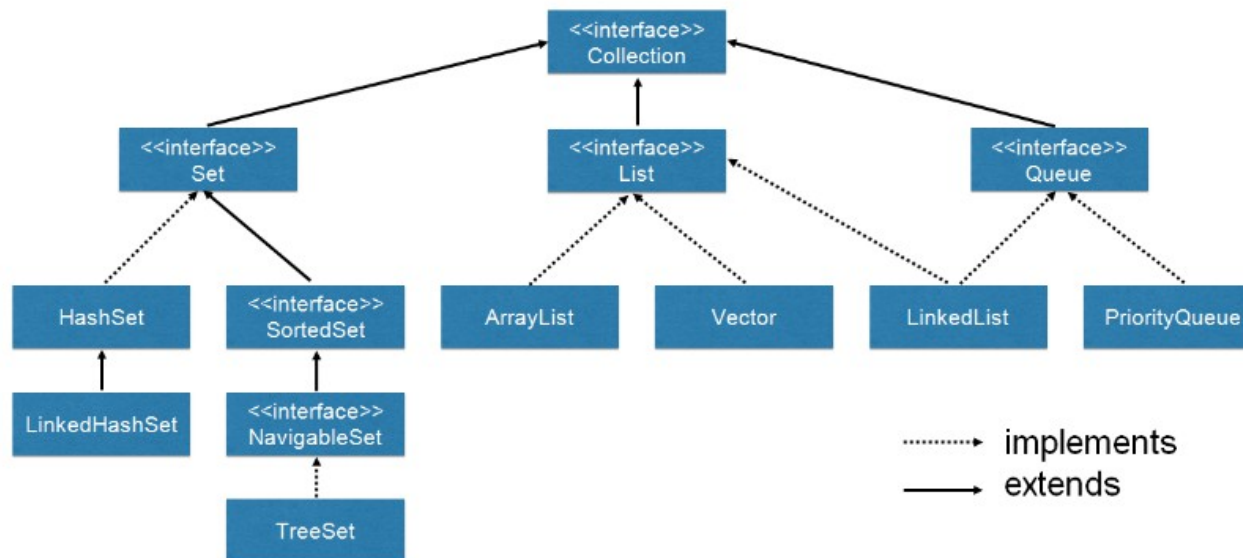
- Enum é uma "classe" especial que representa um grupo de constantes (variáveis imutáveis, como variáveis finais);
- Para criar um enum, utiliza-se a palavra-chave **enum** e as constantes são separadas por vírgulas.
- Assim como classes e interfaces o código fonte terá o sufixo .java;
- Exemplo 01:

```
enum Level {  
    PEQUENO,  
    MEDIO,  
    GRANDE  
}  
- Para acessar:  
  public class Principal {  
      public static void main(String[] args) {  
          Level myVar = Level.GRANDE;  
          System.out.println(myVar);  
          // ou  
          System.out.println(Level.GRANDE);  
      }  
  }
```

# Coleções



- É uma estrutura de dados que fornece uma arquitetura para armazenar e manipular um grupo uniforme de objetos;
- Realiza operações como pesquisa, classificação, inserção, manipulação e exclusão de objetos;
- As coleções possuem interfaces que definem todas as operações que podem ser realizadas;



# Coleções

## ArrayList



- Implementa uma estrutura de dados semelhante a um *array* sendo que, desta vez, os únicos elementos que ela pode armazenar são objetos;
- Estes objetos podem ser acessados através de várias formas e não apenas através de seus índices como acontece nos *arrays*;
- Tem a capacidade de armazenamento variável;
- Seu tamanho aumenta conforme os objetos são inseridos;
- Como somente armazena objetos, tipos primitivos de dados não são suportados;
- Para que os tipos primitivos possam ser armazenados em um ArrayList é necessário que antes eles sejam convertidos para objetos;
- Possui somente uma única dimensão;
- Necessário importar o pacote `java.util.ArrayList`;

# Coleções ArrayList



## ■ Alguns Métodos:

<i><b>Método</b></i>	<i><b>Descrição</b></i>
add(objeto)	Adiciona um objeto ao ArrayList.
remove(índice)	Remove um objeto em determinado índice.
remove(objeto)	Remove o objeto.
contains(objeto)	Retorna <i>true</i> se o objeto existe.
isEmpty()	Retorna <i>true</i> se o ArrayList não possui elementos.
indexOf(objeto)	Retorna o índice do objeto procurado ou -1 se não encontrado.
size()	Retorna o número de elementos no ArrayList.
get(índice)	Retorna o objeto de determinado índice.

## ■ Sintaxe:

```
ArrayList nome-do-Array = new ArrayList (); ou  
ArrayList nome-do-Array = new ArrayList(capacidade); ou  
ArrayList<objeto> nome-do-Array = new ArrayList <objeto>(); ou  
ArrayList<objeto> nome-do-Array = new ArrayList <objeto>(capacidade)
```



# Coleções ArrayList



## Exemplo 01 – Adicionando itens:

```
import java.util.ArrayList;

public class Principal {
    public static void main(String[] args) {
        ArrayList<String> cores = new ArrayList<String>();
        cores.add("Verde");
        cores.add("Amarelo");
        cores.add("Azul");
        cores.add("Branco");
        System.out.println(cores);
    }
}
```

## Exemplo 02 – Acessando um item:

```
import java.util.ArrayList;

public class Principal {
    public static void main(String[] args) {
        ArrayList<String> cores = new ArrayList<String>();
        cores.add("Verde");
        cores.add("Amarelo");
        cores.add("Azul");
        cores.add("Branco");
        System.out.println(cores.get(0));
    }
}
```

# Coleções ArrayList



## Exemplo 03 – Modificando um item:

```
import java.util.ArrayList;

public class Principal {
    public static void main(String[] args) {
        ArrayList<String> cores = new ArrayList<String>();
        cores.add("Verde");
        cores.add("Amarelo");
        cores.add("Azul");
        cores.set(0,"Branco");
        System.out.println(cores.get(0));
    }
}
```

## Exemplo 04 – Removendo um item:

```
import java.util.ArrayList;

public class Principal {
    public static void main(String[] args) {
        ArrayList<String> cores = new ArrayList<String>();
        cores.add("Verde");
        cores.add("Amarelo");
        cores.add("Azul");
        cores.add("Branco");
        cores.remove(0);
        System.out.println(cores.size());
    }
}
```

# Coleções ArrayList



## Exemplo 05 – Removendo todos os itens:

```
import java.util.ArrayList;

public class Principal {
    public static void main(String[] args) {
        ArrayList<String> cores = new ArrayList<String>();
        cores.add("Verde");
        cores.add("Amarelo");
        cores.add("Azul");
        cores.add("Branco");
        cores.clear(); // Mais rápido que o removeAll()
        System.out.println(cores.size());
    }
}
```

# Coleções ArrayList



## Exemplo 06 – Visitando o ArrayList:

```
import java.util.ArrayList;
```

```
public class Principal {  
    public static void main(String[] args) {  
        ArrayList<String> cores = new ArrayList<String>();  
        cores.add("Verde");  
        cores.add("Amarelo");  
        cores.add("Azul");  
        cores.add("Branco");  
        for (int i = 0; i < cores.size(); i++) {  
            System.out.println(cores.get(i));  
        }  
    }  
}
```

```
import java.util.ArrayList;
```

```
public class Principal {  
    public static void main(String[] args) {  
        ArrayList<String> cores = new ArrayList<String>();  
        cores.add("Verde");  
        cores.add("Amarelo");  
        cores.add("Azul");  
        cores.add("Branco");  
        for (String i : cores) {  
            System.out.println(i);  
        }  
    }  
}
```

**// Outra forma do for**

# Coleções ArrayList



## Exemplo 07 – Testando conteúdo:

```
import java.util.ArrayList;
public class Principal {
    public static void main(String[] args) {
        ArrayList<String> cores = new ArrayList<String>();
        cores.add("Verde");
        cores.add("Amarelo");
        cores.add("Azul");
        cores.add("Branco");
        if(cores.contains("Vermelho")){ // O containsAll() verifica a presença de um conjunto
            System.out.println("Contém Vermelho");
        }
        else{
            System.out.println("Não"); }
        }
    }
```

## Exemplo 08 – Verificando o índice do elemento(1a. ocorrência!):

```
import java.util.ArrayList;

public class Principal {
    public static void main(String[] args) {
        ArrayList<String> cores = new ArrayList<String>();
        cores.add("Verde");
        cores.add("Amarelo");
        cores.add("Azul");
        cores.add("Branco");
        System.out.println(cores.indexOf("Amarelo")); //Utilizar lastIndexOf() para a última ocorrência
    }
}
```

# Coleções ArrayList



## Exemplo 09 – Utilizando Objetos:

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

# Coleções ArrayList



## Exemplo 09 (continuação) – Utilizando Objetos:

```
import java.util.ArrayList;
public class Principal {
    public static void main(String[] args) {

        Cachorro cachorro1 = new Cachorro();
        cachorro1.setNome("Danka");
        cachorro1.setIdade(5);
        Cachorro cachorro2 = new Cachorro();
        cachorro2.setNome("Saga");
        cachorro2.setIdade(3);

        ArrayList<Cachorro> dog = new ArrayList<Cachorro>();
        dog.add(cachorro1);
        dog.add(cachorro2);

        System.out.println(dog.get(0).getIdade());
        System.out.println(dog.get(0).getNome());
        System.out.println(dog.get(1).getIdade());
        System.out.println(dog.get(1).getNome());

        for (int i=0; i<dog.size(); i++){
            System.out.println(dog.get(i).getNome());
            System.out.println(dog.get(i).getIdade());
        }

    }
}
```

**// Forma usual de recuperar o dado**  
**// Forma usual de recuperar o dado**  
**// Forma usual de recuperar o dado**

- A classe Collections oferece vários métodos estáticos que auxiliam na manipulação de listas;
- Implementam alguns algoritmos que são de extrema utilidade no cotidiano computacional;
- Deve ser importado o pacote em `import java.util.Collections;`
- Abaixo, alguns de seus métodos;

<i>Método</i>	<i>Descrição</i>
sort	Classifica os elementos de uma lista.
binarySearch	Localiza um objeto em uma lista ordenada.
reverse	Inverte os elementos de uma lista.
shuffle	Embaralha os elementos de uma lista
copy	Copia as referências de uma lista para outra lista.
min	Retorna o menor elemento de uma coleção.
max	Retorna o maior elemento de uma coleção
addAll	Copia todos os elementos de um <i>array</i> para uma coleção
frequency	Informa quantos elementos na coleção são iguais a um elemento específico.
disjoint	Verifica se duas coleções não possuem nenhum elemento em comum.



## Exemplo 01 – classificando:

```
import java.util.ArrayList;
import java.util.Collections;

public class Principal {

    public static void main(String[] args) {
        ArrayList<String> cores = new ArrayList<String>();
        cores.add("Verde");
        cores.add("Amarelo");
        cores.add("Azul");
        cores.add("Branco");
        Collections.sort(cores);
        System.out.println(cores);
        Collections.reverse(cores);
        System.out.println(cores);
    }
}
```

- Analisar e fazer os exemplos do material  
Programação\_Java\_Volume\_2(início na página 35);

■ Executar Lista de Exercícios 5;

# Conceitos Gerais



■ Executar Lista de Exercícios 6 (Impressa – fazer agora!);

# Wrapper Classes



■ As classes *wrapper* fornecem uma maneira de usar tipos de dados primitivos como objetos;

Primitive Data Type	Wrapper Class
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
boolean	Boolean
char	Character

# Wrapper Classes



■ Os seguintes métodos são usados para obter o valor associado ao objeto *wrapper* correspondente: `intValue()`, `byteValue()`, `shortValue()`, `longValue()`, `floatValue()`, `doubleValue()`, `charValue()`, `booleanValue()`;

■ O método `toString()` é usado para converter objetos *wrapper* em *strings*;

■ Exemplos:

```
Integer myInt = 5;
Double myDouble = 5.99;
Character myChar = 'A';
Boolean chaveligada = false;
System.out.println(myInt); ou System.out.println(myInt.intValue());
System.out.println(myDouble); ou System.out.println(myDouble.doubleValue());
String myString = myInt.toString();
int quantidade = Integer.parseInt(valorString); // Converte uma String para um valor inteiro
boolean habilitado = Boolean.parseBoolean(valorString); // Converte uma String para um valor booleano
ArrayList<Integer> numeros = new ArrayList<Integer>();
ArrayList<int> numeros = new ArrayList<int>(); // Inválido
```

# Coleções Vector



- Implementa uma estrutura de dados semelhante a um *array* sendo que, desta vez, os únicos elementos que ela pode armazenar são objetos;
- Comportamento análogo ao ArrayList;
- Grande diferença: O Vector é sincronizado;
- Se uma implementação **thread-safe** não for necessária, é recomendável usar ArrayList no lugar de Vector;
- Necessário importar o pacote `java.util.Vector`;
- Todas as APIs do java em:
  - <https://docs.oracle.com/javase/8/docs/api/overview-summary.html>

# Coleções Vector



## ■ Sintaxe:

- `Vector nome-do-vetor = new Vector (int valorInicial); // ou`
- `Vector nome-do-vetor = new Vector (int valorInicial, int valorIncremento);`
- `Vector<Objeto> nome-do-vetor = new Vector<Objeto>(int valorInicial, int valorIncremento); // ou`
- `Vector<Objeto> nome-do-vetor = new Vector<Objeto>(int valorInicial); // ou`
- `Vector<Objeto> nome-do-vetor = new Vector<Objeto>();`



# Coleções Vector



■ Alguns de seus Métodos:

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

# Coleções Vector



## Exemplos 01:

```
import java.util.Vector;

public class TesteVetor{

    public static void main(String[] args){

        Vector vetor = new Vector(12); // Cria o vetor com capacidade para 12 objetos
        vetor.add("anna") ; // Aqui é adicionado uma String
        vetor.add(new Integer(2)); // Aqui é adicionado um inteiro
        vetor.add(new Temperature(89)); // Aqui é adicionado o objeto Temperature
        System.out.printf("%s%d\n","Tamanho: ", vetor.size());
        System.out.printf("%s%d\n","Capacidade: ",vetor.capacity());
        System.out.printf("%s%s\n","Primeiro Elemento: ",vetor.get(0));
        System.out.printf("%s%s\n","Segundo Elemento: ",vetor.get(1));
        System.out.printf("%s%s\n","Terceiro Elemento: ",(((Temperature)vetor.get(2)).getTemperature()));

    }
}
```

# Coleções Vector



## Exemplo 02:

```
import java.util.Vector;

public class TesteVetor {

    public static void main(String args[]){

        Vector<String> vetor = new Vector<String>();

        vetor.add("Vermelho");
        vetor.add("Verde");
        vetor.add("Violeta");
        vetor.add("Ciano");

        System.out.println(vetor);

        vetor.add(2,"Magenta");
        System.out.println(vetor);

        System.out.println(vetor.get(2));
        System.out.println(vetor.firstElement());
        System.out.println(vetor.lastElement());
        System.out.println(vetor.isEmpty());

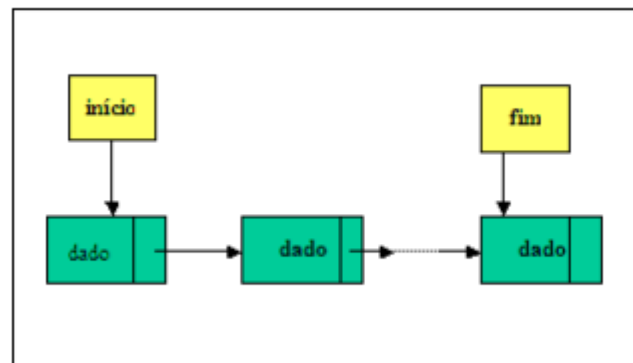
    }
}
```

# Coleções

## Lista Vinculada



- Uma lista vinculada ou ligada é uma coleção de nós conectados por endereços de referência;
- Cada nó contém o dado de interesse e o endereço do nó seguinte;
- O último nó da lista contém um endereço null uma vez que não referencia nenhum outro nó;
- Os dados contidos em um nó podem ser de qualquer tipo e uma lista vinculada não possui um número fixo de elementos;
- Uma lista vinculada é um caso específico de uma lista linear, qual seja, os dados são colocados em sequência;



# Lista Vinculada LinkedList



- A classe `LinkedList` fornece métodos que permitem obter, remover e inserir elementos tanto no início quanto no fim de uma lista;
- Torna-se apropriada para criar estruturas de dados do tipo pilha, fila ou lista encadeada;
- Todas as operações que envolvem a `LinkedList` são assíncronas, portanto, caso necessite de sincronismo terá que implementá-lo explicitamente em seu código;
- Se várias *threads* acessarem uma lista simultaneamente e pelo menos um das threads modificar a lista estruturalmente, ela deverá ser sincronizada externamente;
- Para utilizar a classe `LinkedList` é necessário importar o pacote `import java.util;`
- Sintaxe:

`LinkedList nome-da-lista = new LinkedList ();` ou

`LinkedList <objeto> nome-do-Array = new ArrayList <objeto>()`

# Lista Vinculada LinkedList



■ Descrição em <https://docs.oracle.com/javase/8/docs/api/java/util/LinkedList.html>;

■ Alguns de seus métodos:

<i>Método</i>	<i>Descrição</i>
<code>add(objeto)</code>	Adiciona um objeto ao fim da lista.
<code>add(indice, objeto)</code>	Adiciona um objeto em determinada posição da lista.
<code>addFirst(objeto)</code>	Adiciona um objeto ao início da lista.
<code>addLast(objeto)</code>	Adiciona um objeto ao fim da lista.
<code>clear()</code>	Remove todos os elementos da lista.
<code>contains(objeto)</code>	Verifica se a lista contém o objeto.
<code>element()</code>	Obtém um elemento da lista sem removê-lo.
<code>get(indice)</code>	Retorna o objeto de determinado índice.
<code>getFirst()</code>	Obtém o primeiro elemento da lista.
<code>getLast()</code>	Obtém o último elemento da lista.
<code>indexOf(objeto)</code>	Indica o índice da 1ª ocorrência do objeto ou -1 se não encontrar.
<code>lastIndexOf(objeto)</code>	Indica o índice da última ocorrência do objeto ou -1 se não encontrar.
<code>remove()</code>	Remove o primeiro elemento da lista.
<code>remove(indice)</code>	Remove o elemento localizado por índice.
<code>remove(objeto)</code>	Remove um objeto específico.
<code>removeFirst()</code>	Remove o 1º objeto da lista.
<code>removeLast()</code>	Remove o último objeto da lista.
<code>set(indice, objeto)</code>	Substitui o elemento em índice pelo objeto.
<code>size()</code>	Retorna o número de elementos na lista.

# Lista Vinculada LinkedList



## Exemplo:

```
import java.util.LinkedList;

public class TesteVetor {

    public static void main(String args[]){

        LinkedList<Object> lista = new LinkedList<Object>();
        lista.add("Anna");
        lista.add("Raquel");
        lista.add(new Integer(2)); // Aqui é incluído um inteiro
        lista.add("Marilia");

        System.out.printf("%s%d\n", "Tamanho: ", lista.size());
        System.out.printf("%s%s\n", "Primeiro Elemento: ", lista.get(0));
        System.out.printf("%s%s\n", "Primeiro Elemento: ", lista.getFirst());
        System.out.printf("%s%s\n", "Tamanho: ", lista.size());
        System.out.printf("%s%s\n", "Ultimo Elemento: ", lista.getLast());
        if (lista.contains("Rose")) System.out.printf("%s\n", "Ocorre");

    }

}
```

# Lista Vinculada Pilha



- Uma pilha é uma forma limitada de lista vinculada uma vez que novos nós somente podem ser inseridos ou removidos da parte superior da pilha;
  - O último elemento armazenado em uma pilha deve ser configurado como null para indicar que é o fim da pilha;
  - Uma pilha também é um caso específico de uma lista linear;
  - O conteúdo de uma pilha pode variar dinamicamente e a linguagem Java oferece a classe **Stack**, integrante do pacote `java.util`, para a sua implementação;
  - As operações definidas para uma pilha incluem:
    1. Verificar se a pilha está vazia.
    2. Inserir um elemento na pilha (empilhar ou 'push'), no lado do topo.
    3. Remover um elemento da pilha (desempilhar ou 'pop'), do lado do topo.
- Obs. Uma vez que o último elemento que entrou na pilha será o primeiro a sair, a pilha é também conhecida como uma estrutura do tipo **LIFO (Last In, First Out)**.



# Lista Vinculada Classe Stack



- Necessário importar o pacote `java.util.Stack`;
- Ela herda da classe `Vector` incluindo operações que permitem que um vetor seja tratado como uma pilha;
- Sintaxe:
  - `Stack nome-da-pilha = new Stack (); // ou`
  - `Stack<Objeto> nome-da-pilha = new Stack<Objeto>();`
- Alguns de seus métodos:

<i>Método</i>	<i>Descrição</i>
<code>pop()</code>	Remove um objeto do início da pilha.
<code>push(objeto)</code>	Adiciona um objeto na pilha.
<code>isEmpty()</code>	Verifica se a pilha está vazia.
<code>peek()</code>	Retorna o elemento do topo da pilha sem removê-lo.

# Lista Vinculada

## Classe Stack



### Exemplo:

```
import java.util.Stack;

public class TestePilha{

    public static void main(String[] args){
        Stack pilha = new Stack();
        pilha.push("anna") ;
        pilha.push(new Integer(200));
        pilha.push(new Integer(5));
        pilha.push("Raquel");

        System.out.printf("%s%d\n","Tamanho: ", pilha.size());
        System.out.printf("%s%s\n","Posição na pilha? ",pilha.search("anna"));
        System.out.printf("%s%s\n","Ver Primeiro Elemento: ",pilha.peek());
        System.out.printf("%s%s\n","Ver e Retira Primeiro Elemento: ",pilha.pop());
        System.out.printf("%s%s\n","Pilha Tamanho? ",pilha.size());
    }
}
```

# Lista Vinculada

## Fila



- As filas são as estruturas de dados mais populares em nosso cotidiano;
- Neste tipo de estrutura os nós são removidos somente da cabeça da fila e novos nós são inseridos apenas ao fim da fila;
- Também conhecida como estrutura de dados do tipo **FIFO(First in, First out)**;
- Uma fila também é um caso específico de uma lista linear;
- A linguagem Java, a partir da versão 1.5, oferece a interface **Queue** e a classe **PriorityQueue**, ambas integrantes do pacote `java.util`, para a implementação de filas;
- A implementação não é sincronizada;
- Como **Queue** é uma interface, necessário instanciar uma implementação concreta da interface para usá-la;
- Podem ser escolhidas as seguintes implementações de fila da API de coleções Java:
  - `java.util.LinkedList`
  - `java.util.PriorityQueue`

# Lista Vinculada Queue



- LinkedList é uma implementação de fila padrão;
- Faz com que seja mais eficiente inserir elementos no final da lista e remover elementos do início da lista;
- PriorityQueue armazena seus elementos internamente de acordo com sua ordem natural (se eles implementarem Comparable), ou de acordo com um Comparator passado para o PriorityQueue;

## ■ Sintaxe:

```
Queue queue1 = new LinkedList();  
Queue queue2 = new PriorityQueue();  
Queue<Objeto> queue = new LinkedList<Objeto>();
```

# Lista Vinculada Queue



## Exemplo:

```
import java.util.Stack;
import java.util.LinkedList;
import java.util.Iterator;
public class TestePilha{
    public static void main(String[] args){
        Queue<String> queue=new LinkedList<String>();
        queue.add("Cascadura");
        queue.add("Madureira");
        queue.add("Bento Ribeiro");
        queue.add("Oswaldo Cruz");
        queue.add("Marechal Hermes");
        System.out.println("Início:"+queue.element());
        System.out.println("Início:"+queue.peek());
        System.out.println("Iteragindo:");
        Iterator itr=queue.iterator(); // iterator() é um método herdado!
        while(itr.hasNext()){
            System.out.println(itr.next());
        }
        queue.remove();
        queue.poll();
        System.out.println("Após remoções");
        itr=queue.iterator();
        while(itr.hasNext()){
            System.out.println(itr.next());
        }
    }
}
```

# Coleções Iterator



- Iterator é um objeto que pode ser usado para percorrer coleções, como uma `ArrayList` ou uma `Queue`;
- É chamado de "iterador" porque "iterativo" é o termo técnico para *loop*;
- Para usar um Iterator, importá-lo do pacote `java.util`;
- O método `iterator()` é usado para obter um Iterator para qualquer coleção;
- Para percorrer uma coleção, são utilizados os métodos `hasNext()` e `next()`;
- A classe `LinkedList` oferece os métodos `descendingIterator()` e `listIterator(índice)` para apoiar a iteração;

# Coleções Iterator



## ■ Exemplo01 - iteragindo:

```
import java.util.ArrayList;
import java.util.Iterator;

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

        ArrayList<String> cores = new ArrayList<String>();
        cores.add("Vermelho");
        cores.add("Verde");
        cores.add("Azul");
        cores.add("Magenta");

        Iterator<String> itr = cores.iterator();

        while(itr.hasNext()) {
            System.out.println(itr.next());
        }
    }
}
```

# Coleções Iterator



## Exemplo02 – iteragindo e removendo:

```
import java.util.ArrayList;
import java.util.Iterator;

public class Principal {
    public static void main(String[] args) {
        ArrayList<Integer> numeros = new ArrayList<Integer>();
        numeros.add(18);
        numeros.add(5);
        numeros.add(100);
        numeros.add(8);
        numeros.add(25);
        Iterator<Integer> itr = numeros.iterator();
        while(itr.hasNext()) {
            Integer i = itr.next();
            if(i < 10) {
                itr.remove();
            }
        }
        System.out.println(numeros);
    }
}
```



# Coleções Iterator



## ■ Exemplo03 – iteragindo em ordem inversa:

```
import java.util.Iterator;
import java.util.LinkedList;

public class Principal {

    public static void main(String[] args) {
        LinkedList<String> nomes = new LinkedList<String>();

        nomes.add("JAVA");
        nomes.add("PASCAL");
        nomes.add("C++");

        Iterator<String> iterator = nomes.descendingIterator();

        while(iterator.hasNext()) {
            System.out.println("Lista de nomes = "+iterator.next());
        }
    }
}
```

# Coleções Iterator



## ■ Exemplo04 – iteragindo a partir de determinado índice:

```
import java.util.LinkedList;
import java.util.Iterator;

public class Principal {

    public static void main(String[] args) {

        LinkedList<String> cores = new LinkedList<String>();

        cores.add("Red");
        cores.add("Green");
        cores.add("Black");
        cores.add("White");
        cores.add("Pink");

        Iterator p = cores.listIterator(2);

        while (p.hasNext()) {
            System.out.println(p.next());
        }
    }
}
```

# Coleções Exercícios



Executar Lista de Exercícios 7 (Impressa – fazer agora!);

# Coleções Exercícios



Exercício 01:

```
import java.util.*;
public class Exercicio01 {
    public static void main(String[] args) {

        List<String> list_Strings = new ArrayList<String>();
        list_Strings.add("Red");
        list_Strings.add("Green");
        list_Strings.add("Orange");
        list_Strings.add("White");
        list_Strings.add("Black");
        System.out.println("Antes sort: "+list_Strings);
        Collections.sort(list_Strings);
        System.out.println("Apos sort: "+list_Strings);
    }
}
```

# Coleções Exercícios



Exercício 02:

```
import java.util.*;
public class Exercicio02 {
    public static void main(String[] args) {

        List<String> list_Strings = new ArrayList<String>();
        list_Strings.add("Red");
        list_Strings.add("Green");
        list_Strings.add("Orange");
        list_Strings.add("White");
        list_Strings.add("Black");
        System.out.println("Antes shuffling:\n" + list_Strings);
        Collections.shuffle(list_Strings);
        System.out.println("Apos shuffling:\n" + list_Strings);
    }
}
```

# Coleções Exercícios



## Exercício 03:

```
import java.util.ArrayList;
import java.util.Collections;

public class Exercicio03 {
    public static void main(String[] args) {
        ArrayList<String> c1= new ArrayList<String>();
        c1.add("Red");
        c1.add("Green");
        c1.add("Black");
        c1.add("White");
        c1.add("Pink");

        System.out.println("Array list antes Swap:");
        for(String a: c1){
            System.out.println(a);
        }
        Collections.swap(c1, 0, 2);
        System.out.println("Array list apos swap:");
        for(String b: c1){
            System.out.println(b);
        }
    }
}
```

# Coleções Exercícios



## Exercício 04:

```
import java.util.ArrayList;
import java.util.Collections;
public class Exercicio04 {
    public static void main(String[] args) {
        ArrayList<String> c1= new ArrayList<String>();
        c1.add("Red");
        c1.add("Green");
        c1.add("Black");
        c1.add("White");
        c1.add("Pink");
        System.out.println("Lista do 1o array: " + c1);
        ArrayList<String> c2= new ArrayList<String>();
        c2.add("Red");
        c2.add("Green");
        c2.add("Black");
        c2.add("Pink");
        System.out.println("Lista do 2o array: " + c2);
        ArrayList<String> a = new ArrayList<String>();
        a.addAll(c1);
        a.addAll(c2);
        System.out.println("Novo array: " + a);

    }
}
```

# Coleções Exercícios



## Exercício 05:

```
import java.util.*;

public class Exercicio05 {

    public static void main(String[] args) {

        LinkedList<String> l_list = new LinkedList<String>();

        l_list.add("Red");
        l_list.add("Green");
        l_list.add("Black");
        l_list.add("Pink");
        l_list.add("orange");

        System.out.println("Original linked list:" + l_list);
        for(int i=0; i < l_list.size(); i++) {
            System.out.println("Elementos no indice "+ i + ": " +l_list.get(i));
        }
    }
}
```



# Coleções Exercícios



## Exercício 06:

```
import java.util.*;

public class Exercicio06 {

    public static void main(String[] args) {
        LinkedList <String> c1 = new LinkedList <String> ();
        c1.add("Red");
        c1.add("Green");
        c1.add("Black");
        c1.add("White");
        c1.add("Pink");
        System.out.println("Original linked list: " + c1);
        String x = c1.peekFirst();
        System.out.println("Primeiro elemento na lista: " + x);
        System.out.println("Original linked list: " + c1);
    }
}
```

# Coleções Exercícios



## Exercício 07:

```
import java.util.LinkedList;
import java.util.Iterator;

public class Exercicio07 {

    public static void main(String[] args) {

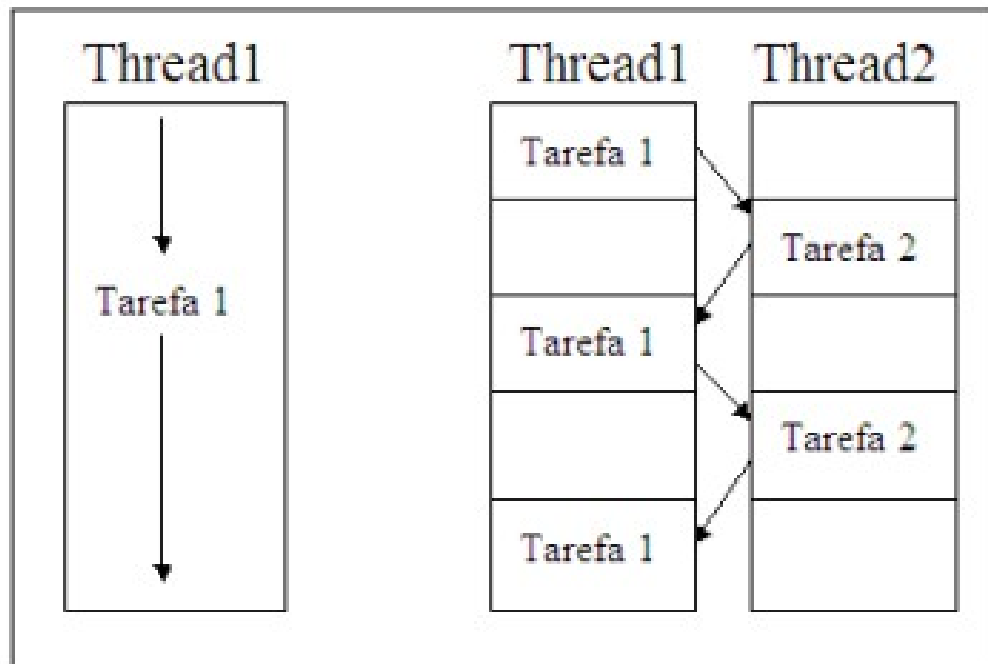
        LinkedList<String> l_list = new LinkedList<String>();

        l_list.add("Red");
        l_list.add("Green");
        l_list.add("Black");
        l_list.add("Pink");
        l_list.add("orange");

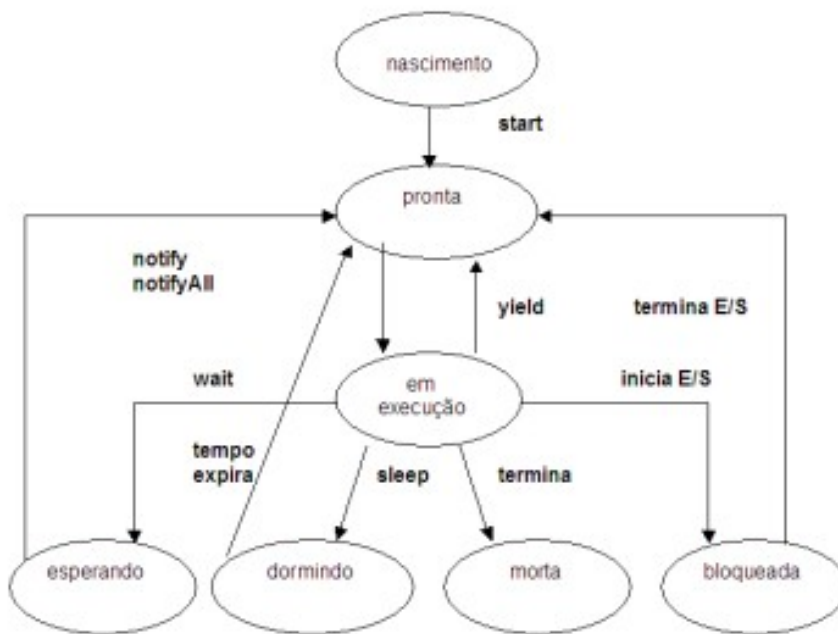
        System.out.println("Original linked list:" + l_list);
        Object first_element = l_list.getFirst();
        System.out.println("Primeiro Elemento: "+first_element);
        Object last_element = l_list.getLast();
        System.out.println("Ultimo Elemento: "+last_element);
    }
}
```

- Uma das principais características da linguagem Java é a sua capacidade de executar múltiplas threads;
- Em computação threads podem ser definidas como uma sequência de linhas de execução que executam tarefas específicas;
- O principal propósito de se utilizar múltiplas threads é explorar o paralelismo nas aplicações;
- Desta forma, é atingida mais eficazmente a capacidade de processamento das CPUs;
- Threads permitem que um programa opere com mais eficiência fazendo várias coisas ao mesmo tempo;

# Threads



# Threads Ciclo de Vida



- **Nascimento** – compreende o processo de criação da thread.
- **Pronta** – a thread já foi criada e está esperando ser despachada para o processamento.
- **Em Execução** – a thread está sendo executada pela CPU.
- **Bloqueada** – a thread requisitou alguma operação ao sistema operacional e fica aguardando ser atendida.
- **Dormindo** – a thread colocou-se voluntariamente para dormir por uma determinada quantidade de tempo.
- **Esperando** – a thread está esperando que algum recurso por ela compartilhado com outra thread seja liberado.
- **Morta** – a thread encerrou a sua tarefa e liberará todos os recursos alocados por ela.

- Quem trata dos despachos das threads é o **Thread Sheduler** integrante da Java Virtual Machine;
- Importante observar que as **threads residem em processos** e que os processos são despachados e tratados pelo sistema operacional;
- Quando a JVM chama o método main() ele cria uma thread principal e a coloca em uma pilha específica;
- Todos os métodos acionados no método main() são colocados nesta pilha;
- As threads criadas voluntariamente por nós desenvolvedores são colocadas em uma outra pilha , denominada de **pilha do usuário**;
- Cabe a JVM selecionar quais threads despachar das diferentes pilhas;

# Threads Criando



- Para criar uma thread deve ser utilizada a classe Thread;
- Esta classe implementa a interface Runnable que define um único método abstrato denominado run();
- Todo aplicativo que deseja utilizar threads precisa implementar este método;

# Threads

## Criando exemplo



### Exemplo 01 – implementando a interface Runnable:

```
public class MinhaThread implements Runnable{
    public void run(){
        vai();
    }
    public void vai(){
        System.out.println("Thread Disparada !!");
        imprimeAlgo();
    }
    public void imprimeAlgo(){
        System.out.println("Imprimindo Algo !!");
    }

    public static void main(String[] args){
        Runnable disparaThread = new MinhaThread();
        Thread testaThread = new Thread(disparaThread);
        System.out.println("Disparou !!");
        testaThread.start();
        System.out.println("A vida continua 01!!");
        System.out.println("A vida continua 02!!");
    }
}
```

Obs.: Executem o código.



# Threads Criando Exemplo



- A instrução `Runnable disparaThread = new MinhaThread()` identifica qual o objeto que deverá ser executado como uma nova thread;
- Em nosso exemplo será o `objeto disparaThread`;
- A instrução `Thread testaThread = new Thread(disparaThread)` passa para o objeto Thread o que será executado;
- Podemos dizer que o objeto Thread é o trabalhador e o objeto Runnable identifica o serviço a ser executado por este trabalhador;
- A instrução `testaThread.start()` faz com que a thread seja colocada no estado de pronta onde aguardará a JVM para ser despachada para execução;
- Os métodos `run()`, `vai()` e `imprimeAlgo()` são colocados na pilha de threads dos usuários;

# Threads

## Criando exemplo



### Exemplo 02 – 1 serviço com 2 trabalhadores:

```
public class MinhaThread implements Runnable{
    public void run(){
        vai();
    }
    public void vai(){
        System.out.println("Thread Disparada !!");
        imprimeAlgo();
    }
    public void imprimeAlgo(){
        System.out.println("Imprimindo Algo !!");
    }

    public static void main(String[] args){
        Runnable disparaThread = new MinhaThread();
        Thread testaThread1 = new Thread(disparaThread);
        Thread testaThread2 = new Thread(disparaThread);
        System.out.println("Disparou !!");
        testaThread1.start();
        testaThread2.start();
        System.out.println("A vida continua 01!!");
        System.out.println("A vida continua 02!!");
    }
}
```

Obs.: Executem o código.

# Threads Criando exemplo



## Exemplo 03 – herdando da classe Thread:

```
public class MinhaThread extends Thread{
    public void run(){
        vai();
    }
    public void vai(){
        System.out.println("Thread Disparada !!");
        imprimeAlgo();
    }
    public void imprimeAlgo(){
        System.out.println("Thread Disparada !!");
    }
    public static void main(String[] args){
        MinhaThread testaThread = new MinhaThread();
        System.out.println("Disparou !!");
        testaThread.start();
        System.out.println("A vida continua 01!!");
        System.out.println("A vida continua02 !!");
    }
}
```

Obs.: Executem o código.

# Threads Criando exemplo



- Neste código a classe MinhaThread herda da classe Thread;
- Quando a classe MinhaThread é instanciada, o objeto Thread já está pronto para ser ativado;
- Basta acionar o método start();
- O grande inconveniente é que a linguagem Java não suporta herança múltipla, assim, se estamos herdando da classe Thread não poderemos herdar de mais nenhuma outra classe;

# Threads Criando exemplo



## Exemplo 04 – 2 serviços e 2 trabalhadores:

```
public class TestaThreads implements Runnable{
    private String palavra;
    private int delay;
    public TestaThreads(String oQueFalar){    // Construtor
        palavra=oQueFalar;
    }
    public void run() {
        System.out.print(palavra + " ");
        System.out.print(palavra + " ");
        System.out.print(palavra + " ");
        System.out.print(palavra + " ");
        System.out.print(palavra + " ");
    }
    public static void main(String[] args) {
        Runnable ping=new TestaThreads("PING");
        Runnable pong=new TestaThreads("PONG");
        new Thread(ping).start();
        new Thread(pong).start();
    }
}
```

Obs.: Executem o código (pode deixar o JCreator não interrompível!!!).

# Threads

## Colocando para dormir



- Para garantir que outras threads tenham a oportunidade de serem processadas ou para garantir o sincronismo de eventos pode-se colocar uma thread para dormir;
- Para tal é disponibilizado o método `sleep()`;
- Este método recebe como argumento um número inteiro especificando o tempo, em milisegundos, que a thread ficará dormindo;
- Este método deverá estar dentro de uma estrutura de exceção `try/catch`;

# Threads Dormindo exemplo



## Exemplo 04:

```
public class TestaThreads implements Runnable{
    private String palavra;
    private int delay;
    public TestaThreads(String oQueFalar, int tempoDelay){    // Construtor
        palavra=oQueFalar;
        delay=tempoDelay;
    }
    public void run() {
        try {
            for (;;) {
                System.out.print(palavra + " ");
                Thread.sleep(delay);
            }
        } catch (Exception e) {
            System.err.println("Erro");
        }
    }
}

public static void main(String[] args) {
    Runnable ping=new TestaThreads("PING",500);
    Runnable pong=new TestaThreads("PONG",500);
    new Thread(ping).start();
    new Thread(pong).start();
}
```

Obs.: Executem o código.

# Threads

## Atribuindo e obtendo nomes



- Os métodos `setName()` e `getName()` permitem, respectivamente, que sejam atribuídos nomes às threads e que estes nomes possam ser posteriormente resgatados;
- O método `currentThread()` retorna uma referência ao objeto thread atualmente em execução;
- Utilizado quando a classe não herda de `Thread`;



# Threads exemplo get and setName()



## Exemplo 05:

```
public class TestaThreads implements Runnable{
    private String palavra;
    private int delay;
    public TestaThreads(String oQueFalar, int tempoDelay){
        palavra=oQueFalar;
        delay=tempoDelay;
    }
    public void run() {
        try {
            for (;;) {
                System.out.println(Thread.currentThread().getName());
                Thread.sleep(delay);
            }
        } catch (Exception e) {
            System.err.println("Erro");
        }
    }
}
```

# Threads exemplo get and setName()



## ■ Exemplo 05 continuação:

```
public static void main(String[] args) {  
  
    Runnable ping=new TestaThreads("PING",500);  
    Runnable pong=new TestaThreads("PONG",500);  
    Thread ping1 = new Thread(ping);  
    Thread pong1 = new Thread(pong);  
    ping1.setName("Primeira");  
    pong1.setName("Segunda");  
    ping1.start();  
    pong1.start();  
  
}
```

Obs.: Executem o código.

# Threads

## Atribuindo prioridades



- Uma thread possui, inicialmente, a mesma prioridade da thread que a criou;
- Esta prioridade pode ser alterada através do método `setPriority()`;
- Seu valor pode variar entre `MIN_PRIORITY` (valor padrão=1) e `MAX_PRIORITY` (valor padrão=10);
- Se nenhuma prioridade é especificada o valor `NORM_PRIORITY` (valor padrão=5) é assumido;
- O método `getPriority()` obtém o valor corrente da prioridade;

# Threads

## Exemplo prioridades



### Exemplo 06:

```
public class TestaThreads implements Runnable{
    private String palavra;
    private int delay;
    public TestaThreads(String oQueFalar, int tempoDelay){
        palavra=oQueFalar;
        delay=tempoDelay;
    }
    public void run() {
        try {
            for (;;) {
                System.out.println(Thread.currentThread().getName());
                System.out.println(Thread.currentThread().getPriority());
                Thread.sleep(delay);
            }
        } catch (Exception e) {
            System.err.println("Erro");
        }
    }
}
```

# Threads

## Exemplo prioridades



### ■ Exemplo 06 Continuação:

```
public static void main(String[] args) {  
  
    Runnable ping=new TestaThreads("PING",500);  
    Runnable pong=new TestaThreads("PONG",500);  
    Thread ping1 = new Thread(ping);  
    Thread pong1 = new Thread(pong);  
    ping1.setName("Primeira");  
    ping1.setPriority(Thread.NORM_PRIORITY+5);  
    pong1.setName("Segunda");  
    pong1.setPriority(Thread.NORM_PRIORITY+2);  
    ping1.start();  
    pong1.start();  
  
}
```

Obs.: Executem o código.

# Threads

## Renunciando à CPU



- O método `yield()` permite que uma thread renuncie ao uso da CPU, permitindo que outras threads possam ser disparadas;
- O método `yield()` informa ao thread scheduler que a thread não necessita, momentaneamente, ser executada;
- A thread emissora do `yield()` fica em estado de pronta e, caso não exista nenhuma outra thread para ser despachada, ela entra novamente em execução;
- Este método não precisa de uma estrutura `try/catch`.

# Threads

## Renunciando à CPU



### ■ Exemplo 06:

```
private String palavra;  
private int delay;  
private boolean renuncia;  
public TestaThreads(String oQueFalar, int tempoDelay, boolean renuncia){  
    palavra=oQueFalar;  
    delay=tempoDelay;  
    this.renuncia=renuncia;  
}  
public void run() {  
    try {  
        for (;;) {  
            System.out.println(palavra);  
            if (renuncia) {  
                Thread.yield();  
            }  
            Thread.sleep(delay);  
        }  
    } catch (Exception e) {  
        System.err.println("Erro");  
    }  
}
```

# Threads

## Renunciando à CPU



### ■ Exemplo 06 continuação:

```
public static void main(String[] args) {  
    Runnable ping1=new TestaThreads("um",500,true);  
    Runnable ping2=new TestaThreads("dois",500,false);  
    Runnable ping3=new TestaThreads("tres",500,false);  
    Runnable ping4=new TestaThreads("quatro",500,false);  
    Runnable ping5=new TestaThreads("cinco",500,false);  
    Runnable ping6=new TestaThreads("seis",500,false);  
    Thread ping11 = new Thread(ping1);  
    Thread ping22 = new Thread(ping2);  
    Thread ping33 = new Thread(ping3);  
    Thread ping44 = new Thread(ping4);  
    Thread ping55 = new Thread(ping5);  
    Thread ping66 = new Thread(ping6);  
  
    ping11.start();  
    ping22.start();  
    ping33.start();  
    ping44.start();  
    ping55.start();  
    ping66.start();  } }
```

Obs.: Executem o código.



# Threads

## Interrompendo



- O processo de cancelamento sem riscos de uma thread é feito através dos métodos `interrupt()` e `interrupted()`;
- O processo de cancelamento envolve: a ordem da thread que está comandando o cancelamento e o consentimento da thread que está sendo cancelada;
- Uma interrupção não força a parada de uma thread, embora ela interrompa o “descanso” de uma thread em `wait` ou `sleeping`, ela apenas ativa o estado de interrupção da thread;
- Quando uma thread emite um `interrupt()` para uma outra thread ela apenas irá ativar este estado de interrupção;
- A outra thread se desejar ser interrompida deverá utilizar o método `interrupted()` para verificar se o estado de interrupção foi ativado;
- Caberá a esta thread decidir se interromperá ou não o seu processamento;

# Threads

## Interrompendo



■ Exemplo 07:

```
public class InterrompeThread implements Runnable{

    public void run() {
        try {
            Runnable filha=new ThreadFilha();
            Thread filha1 = new Thread(filha);
            filha1.start();
            Thread.sleep(5); // Tempo para respirar!!!
            System.out.println("Mande parar!");
            filha1.interrupt();
        }catch (Exception e) {
            System.err.println("Erro");
        }
    }
}
```

# Threads Interrompendo



Exemplo 07 continuação:

```
public static void main(String[] args) {  
  
    Runnable ping1=new InterrompeThread();  
    Thread ping11 = new Thread(ping1);  
    ping11.start();  
  
}  
class ThreadFilha implements Runnable{  
  
    public void run() {  
        int i=0;  
        while(!Thread.currentThread().interrupted()) {  
            i++;  
            System.out.println( i + " ");  
        }  
    }  
}
```

Obs.: Executem o código.

# Threads

## Esperando terminar



- Às vezes precisamos esperar o término da execução de várias threads que foram disparadas para produzirmos um resultado final;
- Para tal deve ser utilizado o método `join()`;
- Este método precisa ser utilizado em conjunto com uma estrutura try/catch;
- Também pode ser utilizado o método `isAlive()`;

# Threads

## Esperando terminar



### Exemplo 08:

```
public class TestaJoin implements Runnable {
```

```
    int[] array;
```

```
    static int soma =0;
```

```
    public TestaJoin(int[] array){  
        this.array=array;
```

```
    }
```

```
    public void run() {
```

```
        for (int i=0; i<array.length; i++){  
            soma = soma + array[i];
```

```
        }
```

```
    }
```

# Threads

## Esperando terminar



### Exemplo 08 continuação:

```
public static void main(String[] args) {  
  
    int[] array1 = new int[]{1,2,3,4,5,6,7,8,9,10};  
    int[] array2 = new int[]{11,12,13,14,15,16,17,18,19,20};  
  
    Runnable ping1=new TestaJoin(array1);  
    Thread ping11 = new Thread(ping1);  
    Runnable ping2=new TestaJoin(array2);  
    Thread ping22 = new Thread(ping2);  
    ping11.start();  
    ping22.start();  
    try {  
        ping11.join();  
        ping22.join();  
        System.out.println("Esperei o término !");  
        System.out.println(soma);  
    }catch (Exception e) {  
        System.err.println("Erro");  
    }  
}
```

Obs.: Executem o código.

# Threads

## Esperando terminar



### ■ Exemplo 08.1:

```
public class Principal extends Thread {
    public static int amount = 0;

    public static void main(String[] args) {
        Principal thread = new Principal();
        thread.start();
        while(thread.isAlive()) {
            System.out.println("Esperando...");
        }
        System.out.println("Principal: " + amount);
        amount++;
        System.out.println("Principal: " + amount);
    }

    public void run() {
        amount++;
    }
}
```

# Threads

## Sincronização



- Para sincronizar o acesso a recursos compartilhados a linguagem Java utiliza monitores (semáforos);
- Todo o objeto contendo métodos com o atributo **synchronized** possui um monitor;
- O monitor permite que apenas uma thread por vez execute um método **synchronized** sobre o objeto;
- O objeto fica atomicamente bloqueado quando o seu método **synchronized** é invocado;
- As demais threads que tentarem invocar este método ficarão em estado de espera;
- As variáveis de instância pertencem unicamente a uma instância de determinada classe;



# Threads

## Sincronização



- Se esta classe implementa threads, cada thread disparada poderá ter a capacidade de compartilhar estas variáveis, o que poderá provocar sérios danos ao aplicativo;
- No código a seguir temos um exemplo desta possível situação;
- São apresentadas as classes MinhaClasse e MinhaThread;
- MinhaClasse define uma variável de instância denominada saldo que tem o seu valor incrementado a cada invocação de seu método calcula;
- A classe MinhaThread disparará duas threads sobre uma única instância da classe MinhaClasse;

# Threads

## Sincronização



### ■ Exemplo 09:

```
public class MinhaClasse {  
  
    private double saldo;  
  
    public MinhaClasse(){  
    }  
  
    public void calcula(){  
        saldo++;  
        System.out.println("Saldo: " + saldo);  
    }  
}
```

# Threads

## Sincronização



### ■ Exemplo 09 - continuação:

```
public class MinhaThread implements Runnable {  
  
    private MinhaClasse soma = new MinhaClasse();  
  
    public void run() {  
        soma.calcula();  
    }  
  
    public static void main(String[] args) {  
        Runnable dispara = new MinhaThread();  
        Thread uma = new Thread(dispara);  
        Thread outra = new Thread(dispara);  
        uma.start();  
        outra.start();  
    }  
}
```

Obs.: Executem o código.

# Threads

## Sincronização



- O valor do saldo foi incrementado com a participação das duas threads;
- Portanto, podemos concluir que as duas threads tiveram acesso à mesma variável de instância;
- Imaginando a situação de gravação de alguma informação em um disco magnético, esta operação de concorrência poderia provocar algum resultado inesperado;
- Da mesma forma seria se dois correntistas, que compartilham a mesma conta-corrente, pudessem ter acesso simultâneo para realizar operações de retirada;
- No código exemplo vamos utilizar o método **synchronized** para termos a garantia que apenas uma thread terá acesso ao recurso que queremos controlar;

# Threads

## Sincronização



■ Exemplo 10 – sincronizando o acesso:

```
public class MinhaClasse {  
  
    private double saldo;  
  
    public MinhaClasse(){  
    }  
  
    public synchronized void calcula(){  
        saldo++;  
        System.out.println("Saldo: " + saldo);  
    }  
}
```

Obs.: Executem o código do Exemplo 9 com a modificação acima.

# Threads Daemon



- A thread daemon em Java é um thread que fornece serviços às threads do usuário;
- Sua vida depende da vida das threads do usuário, ou seja, quando todos as threads do usuário morrem, a JVM encerra essa thread automaticamente;
- Ela fornece serviços para threads de usuários para tarefas de suporte em *background*;
- Ela não tem nenhum papel na vida além de servir às threads de usuários;
- É uma thread de baixa prioridade;
- Suportada pelos métodos `setDaemon()` e `isDaemon()`;

# Threads Daemon



## Exemplo 11:

```
public class TestaThreads extends Thread{
    public void run(){
        if(Thread.currentThread().isDaemon()){
            System.out.println("Thread Daemon");
        }
        else{
            System.out.println("Thread Usuario");
        }
    }
}

public static void main(String[] args){

    TestaThreads t1=new TestaThreads();
    TestaThreads t2=new TestaThreads();
    TestaThreads t3=new TestaThreads();

    t1.setDaemon(true);
    t1.start();
    t2.start();
    t3.start();
}
}
```

Obs.: Executem o código.

# Threads

## Pool de Threads



- Representa um pool de threads que estão aguardando uma tarefa e são reutilizadas várias vezes;
- Um pool de threads de tamanho fixo é criado;
- É atribuída uma tarefa, à uma thread do pool, pelo provedor de serviços;
- Após a conclusão da tarefa, a thread do pool fica disponível para uma nova tarefa;
- Economiza-se tempo porque não há a necessidade de criar-se uma nova thread;
- É usado em Servlets e JSPs onde o container cria um pool de threads para processar a solicitação;



# Threads

## Pool de Threads



- As tarefas são enviadas ao pool por meio de uma fila interna, que armazena tarefas extras sempre que houver mais tarefas ativas do que threads;
- Surgem as figuras das classes `Executors` e `ExecutorService` pertencentes aos pacotes `java.util.concurrent.Executors` e `java.util.concurrent.ExecutorService`, respectivamente;
- Suportado pelos métodos `newFixedThreadPool(int s)`, `newCachedThreadPool()` e `newSingleThreadExecutor()`;

# Threads

## Pool de Threads

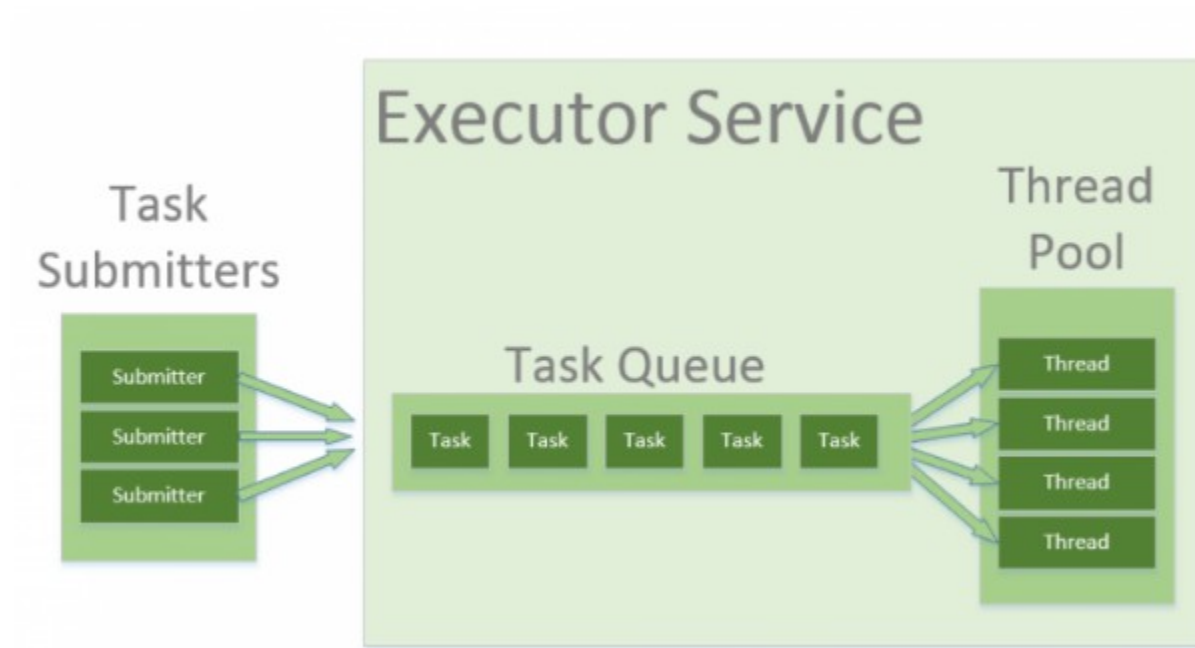


Figura em <https://www.baeldung.com/thread-pool-java-and-guava>

# Threads

## Pool de Threads



### ■ Exemplo 12:

```
class TestaThreads implements Runnable {
    private String message;
    public TestaThreads(String s){
        this.message=s;
    }
    public void run() {
        System.out.println(Thread.currentThread().getName()+" (Inicio) mensagem = "+message);
        processmessage();
        System.out.println(Thread.currentThread().getName()+" (Fim)");
    }
    private void processmessage() {
        try {
            Thread.sleep(2000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

# Threads

## Pool de Threads



### ■ Exemplo 12 continuação:

```
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;

public class TestaThreadPool {
    public static void main(String[] args) {
        ExecutorService executor = Executors.newFixedThreadPool(5);
        for (int i = 0; i < 20; i++) { // Manda trabalho 20 X, para as 5 threads criadas
            Runnable worker = new TestaThreads("" + i);
            executor.execute(worker);
        }
        executor.shutdown();
        while (!executor.isTerminated()) { }

        System.out.println("Todas as threads encerradas");
    }
}
```

Obs.: Executem o código.

# Threads

## Exercícios



■ Executar Lista de Exercícios 8 (Impressa – fazer agora!);

## ■ Exercício 08-01

```
public class Exercicio0801 {  
  
    public static void main(String[] args) {  
  
        Runnable dispara1 =new Exercicio0801_Conta_Vogal("Austeclynio");  
        Runnable dispara2 =new Exercicio0801_Conta_Palavras("Azul   Amarelo Vermelho   Preto   ");  
        Thread uma = new Thread(dispara1);  
        Thread outra = new Thread(dispara2);  
        uma.start();  
        outra.start();  
    }  
}
```

## ■ Exercício 08-01 - continuação

```
public class Exercicio0801_Conta_Vogal implements Runnable{
    String texto;
    public Exercicio0801_Conta_Vogal(String texto) {
        this.texto = texto;
    }
    public void run() {
        contaVogal();
    }
    public void contaVogal(){
        char[] vogais = {'A','E','I','O','U','a','e','i','o','u'};
        int count = 0;
        for (int i = 0; i < texto.length(); i++) {
            for (int j=0; j<vogais.length; j++){
                if (texto.charAt(i)== vogais[j]){
                    count++;
                }
            }
        }
        System.out.println("Vogais :" + count);
    }
}
```

# Threads Soluções



## Exercício 08-01 - continuação

```
public class Exercicio0801_Conta_Palavras implements Runnable{
    String texto;
    public Exercicio0801_Conta_Palavras(String texto) {
        this.texto = texto.trim();
    }
    public void run() {
        contaPalavras();
    }
    public void contaPalavras(){
        int count=0;
        int indice = 0;
        String pedaco="";
        int tamanho = texto.length();
        if (texto.length()!=0){
            if (texto.indexOf(" ")>=0){
                for (int i=0; i<tamanho; i++){
                    indice = texto.indexOf(" ");
                    if (indice>=0){
                        count++;
                        texto = (texto.substring(indice)).trim();
                    }
                    else {
                        count++;
                        break;
                    }
                }
            }
            else count = 1;
        }
        System.out.println("Palavras :" + count);
    }
}
```



## ■ Exercício 08- 02

```
public class Exercicio0802 {  
  
    public static void main(String[] args) {  
        Runnable dispara1 =new Exercicio0802_Menor_Numero(2,30,5);  
        Runnable dispara2 =new Exercicio0802_Media_Numero(2,30,5);  
        Thread uma = new Thread(dispara1);  
        Thread outra = new Thread(dispara2);  
        uma.start();  
        outra.start();  
    }  
  
}
```

## ■ Exercício 08-02 - continuação

```
public class Exercicio0802_Menor_Numero implements Runnable{
    int a;
    int b;
    int c;
    Exercicio0802_Menor_Numero(int a, int b, int c) {
        this.a = a;
        this.b = b;
        this.c = c;
    }
    public void run() {
        menorNumero();
    }
    public void menorNumero(){
        System.out.println("Menor Numero :" + Math.min(Math.min(a, b), c));
    }
}
```

## ■ Exercício 08-02 - continuação

```
public class Exercicio0802_Media_Numero implements Runnable{
    float a;
    float b;
    float c;
    Exercicio0802_Media_Numero(int a, int b, int c) {
        this.a = a;
        this.b = b;
        this.c = c;
    }
    public void run() {
        mediaNumero();
    }
    public void mediaNumero(){
        System.out.println("Media Numero :" + (a + b + c) / 3);
    }
}
```

■ A classe Math contém métodos para executar operações numéricas básicas, como exponencial elementar, logaritmo, raiz quadrada e funções trigonométricas;

■ Métodos em:

**[docs.oracle.com/javase/8/docs/api/java/lang/Math.html](https://docs.oracle.com/javase/8/docs/api/java/lang/Math.html)**

■ Exemplos:

```
Math.max(18,9);
```

```
Math.min(18,9);
```

```
Math.sqrt(54);
```

```
Math.abs(-14.8);
```

```
(int)(Math.random() * 101); // O Math.random retorna um valor double com sinal positivo, maior ou igual a 0,0 e menor que 1,0.
```

```
Math.pow(2,8); // Retorna o valor do primeiro argumento elevado à potência do segundo argumento.
```

# SecureRandom class



- Essa classe fornece um gerador de números aleatórios criptograficamente forte (RNG);
- Um número aleatório criptograficamente forte atende minimamente aos testes estatísticos de gerador de números aleatórios especificados no FIPS 140-2;
- Além disso, SecureRandom deve produzir saída não determinística;
- Todas as sequências de saída SecureRandom devem ser criptograficamente fortes, conforme descrito em RFC 1750;
- Herda da classe Random;
- Importar do pacote `import java.security.SecureRandom;`
- Sintaxe:
  - `SecureRandom random = new SecureRandom();`

# SecureRandom class



## ■ Exemplo:

```
import java.security.SecureRandom; //A partir do Java 8
import java.util.Base64;

public class CalculaToken {

    SecureRandom secureRandom = new SecureRandom();

    Base64.Encoder base64Encoder = Base64.getUrlEncoder();

    byte[] randomBytes = new byte[50];
    secureRandom.nextBytes(randomBytes);

    return(base64Encoder.encodeToString(randomBytes));

}
```

# Tratamento de Exceções



- Uma aplicação para ser robusta e confiável deverá se preocupar com o tratamento dos possíveis erros que poderão ocorrer durante a sua utilização;
- A apresentação destes erros para ao usuário final deve ser feita pela aplicação em si, de forma clara, e não por algum componente externo;
- Alguns possíveis problemas podem ser tratados já em tempo de compilação como, por exemplo, uma tentativa de se conectar a uma máquina servidora;
- Como existe a possibilidade de esta máquina estar indisponível, o método que trata desta conexão obriga a previsão desta possibilidade no código;
- Outro caso semelhante é a tentativa de ler algum dado gravado em um arquivo residente em um disco magnético. Este arquivo pode não mais existir;
- O compilador Java sempre avisará quando o código precisar se precaver, apresentando um erro de compilação;

# Tratamento de Exceções



- Nem sempre o compilador possuirá este poder;
- Alguns tipos de erros podem ser pouco prováveis como, por exemplo, não conseguir gravar um arquivo em um disco magnético por falta de espaço físico;
- O código compilará e somente em tempo de execução será percebido o problema.
- Exemplo de erro em tempo de execução:

```
import java.util.Scanner;

public class TestaCalculadora{

    public static void main(String[] args){
        int numero1,numero2, resultado;
        Scanner entrada = new Scanner(System.in);
        System.out.println("Digite o primeiro numero:");
        numero1 = entrada.nextInt();      // Digite um número maior do que 0
        System.out.println("Digite o segundo numero:");
        numero2 = entrada.nextInt();      // Digite o valor 0
        resultado = numero1 / numero2;
        System.out.printf("%s%d\n", "Resultado = ", resultado);
    }
}
```



# Tratamento de Exceções



- Neste caso será lançada a mensagem de exceção *java.lang.ArithmeticException*;
- Caso tenha sido digitado uma letra a mensagem de exceção será *java.util.InputMismatchException*;
- Os códigos passíveis de erros devem ficar contidos em blocos **try/catch**;
- Sob o **try** fica o código do risco e sob o **catch** fica o código para o tratamento do erro pela aplicação;
- Todas as classes de exceção do Java herdam da classe **Exception**;
- Alguns tipos de exceções: *RuntimeException*, *IOException*, *ThreadDeath*, *OutOfMemoryError*, *ArrayIndexOutOfBoundsException*, *InputMismatchException*, *ClassCastException*, *NullPointerException* e *ArithmeticException*;
- Em algumas situações algum código precisa ser executado mesmo que uma exceção seja lançada, para tal existe o bloco **finally**;
- Toda e qualquer instrução que integra este bloco será executada mesmo que uma exceção seja ou não lançada;

# Tratamento de Exceções



## Exemplo 01 try/catch:

```
import java.util.Scanner;
import java.util.InputMismatchException;
public class TestaCalculadora{
    public static void main(String[] args) {
        int numero1,numero2, resultado;
        boolean continua=true;
        Scanner entrada = new Scanner(System.in);
        do {
            try{
                System.out.println("Digite o primeiro numero:");
                numero1 = entrada.nextInt();
                System.out.println("Digite o segundo numero:");
                numero2 = entrada.nextInt();
                resultado = numero1 / numero2;
                System.out.printf("%s%d\n", "Resultado = ", resultado);
                continua = false;
            }
            catch(InputMismatchException erro1){
                System.err.printf("%s", "Você digitou errado. Entre com números inteiros\n");
                entrada.nextLine();
            }
            catch(ArithmeticException erro2){
                System.err.printf("%s", "Divisão por ZERO.\n");
                entrada.nextLine();
            }
        } while(continua);
    }
}
```

Obs.: Executem o código.

# Tratamento de Exceções



## Exemplo 02 try/catch/finally:

```
import java.util.Scanner;
import java.util.InputMismatchException;
public class TestaCalculadora{
    public static void main(String[] args) {
        int numero1,numero2, resultado;
        boolean continua=true;
        Scanner entrada = new Scanner(System.in);
        do {
            try{
                System.out.println("Digite o primeiro numero:");
                numero1 = entrada.nextInt();
                System.out.println("Digite o segundo numero:");
                numero2 = entrada.nextInt();
                resultado = numero1 / numero2;
                System.out.printf("%s%d\n", "Resultado = ", resultado);
                continua = false;
            }
            catch(InputMismatchException erro1){
                System.err.printf("%s", "Você digitou errado. Entre com números inteiros\n");
                entrada.nextLine();
            }
            catch(ArithmeticException erro2){
                System.err.printf("%s", "Divisão por ZERO.\n");
                entrada.nextLine();
            }
            finally{
                System.out.println("Passou por mim");
            }
        } while(continua);
    }
}
```

Obs.: Executem o código.

# Processando Arquivos



■ Os arquivos podem ser classificados em dois tipos:

- **Arquivos texto** – um arquivo que contém caracteres do tipo texto. Os registros ou campos são por delimitados por caracteres especiais definidos pelo próprio desenvolvedor da aplicação;
- **Arquivos binários** – são arquivos contendo dados primitivos e objetos de dados;

■ Quase todas as instruções que manipulam arquivos devem estar contidas em um bloco ***try/catch***, uma vez que vários tipos de exceções podem ser lançados;

■ As exceções nas operações de E/S mais comumente lançadas são:

- **IOException** - Indica que ocorreu algum problema na operação de E/S;
- **EOFException** - Indica que está sendo feita uma operação de leitura além do fim do arquivo;
- **FileNotFoundException** - Indica que o arquivo não foi localizado;

■ A classe **File** do pacote `java.io` permite o trabalho com arquivos;

# Processando Arquivos

## File class



- A classe **File** não lança exceções uma vez que ela apenas obtém o caminho para o diretório ou o arquivo desejado;
- Cabe ao desenvolvedor verificar se o diretório ou arquivo existe ou não;
- A classe **File** possui os seguintes construtores:

<i>Construtor</i>	<i>Descrição</i>
File(StringPathName)	Cria um objeto File que faz referencia um especifico path.
File(StringPathName, StringSubPath)	Cria um objeto File que referencia um path combinando um especifico path com um subpath.
File(File, StringSubPath)	Cria um objeto File que referencia um path combinando um outro objeto File com um subpath.

# Processando Arquivos

## File class



### Exemplos:

```
File arquivoLeitura = new File("teste.txt");  
File arquivoLeitura = new File("c:/Aplicações/teste.txt");  
File arquivoLeitura = new File("../Aplicações", "teste.txt");
```

### Alguns métodos da classe File:

<i>Método</i>	<i>Descrição</i>
<code>exists()</code>	Retorna <i>true</i> se o pathname existe.
<code>isDirectory()</code>	Retorna <i>true</i> se o pathname existe e é um diretório.
<code>isFile()</code>	Retorna <i>true</i> se o pathname existe e é um arquivo.
<code>getName()</code>	Retorna o nome do diretório ou do arquivo.
<code>length()</code>	Retorna o tamanho em bytes do arquivo
<code>mkdir()</code>	Cria um novo diretório para o objeto File.
<code>delete()</code>	Apaga o arquivo ou o diretório do objeto File.
<code>getAbsolutePath()</code>	Obtém o endereço absoluto do objeto File.

# Processando Arquivos

## File class



■ Exemplo 01 - informações a respeito de determinado arquivo:

```
import java.io.*;

public class TestaFiles{

    public static void main(String[] args){
        File arquivo = new File("TestaFiles.java");
        if (arquivo.exists()){
            System.out.println("Nome do arquivo: " + arquivo.getName());
            System.out.println("Nome do path: " + arquivo.getPath());
            System.out.println("Nome do path absoluto: " + arquivo.getAbsolutePath());
            System.out.println("Tamanho do arquivo: " + arquivo.length() + " bytes");
        }
    }
}
```

# Processando Arquivos

## File class



■ Exemplo 02 – criando um arquivo:

```
import java.io.*;

public class CriarArquivo {

    public static void main(String[] args) {
        try {
            File arquivo = new File("nomearquivo.txt");
            if (arquivo.createNewFile()) {
                System.out.println("Arquivo Criado: " + arquivo.getName());
            } else {
                System.out.println("Arquivo já existe.");
            }
        } catch (IOException e) {
            System.out.println("Arquivo não criado!!!");
            e.printStackTrace();
        }
    }
}
```



# Processando Arquivos FileWriter class



■ Exemplo 03 – gravando dados em um arquivo:

```
import java.io.FileWriter;
import java.io.IOException;

public class GravarArquivo {
    public static void main(String[] args) {
        try {
            FileWriter arquivoSaida = new FileWriter("d:/Curso_Java/meu_arquivo.txt");
            arquivoSaida.write("A primeira gravacao a gente nunca esquece.");
            arquivoSaida.close();
            System.out.println("Operação realizada com sucesso!");
        }
        catch (IOException e) {
            System.out.println("Algo deu errado!");
            e.printStackTrace();
        }
    }
}
```

# Processando Arquivos File e Scanner classes



Exemplo 04 – lendo dados de um arquivo:

```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

public class LerArquivo {
    public static void main(String[] args) {
        try {
            File meuArquivo = new File(d:/Curso_Java/meu_arquivo.txt);
            Scanner ledor = new Scanner(meuArquivo);
            while (ledor.hasNextLine()) {
                String data = ledor.nextLine();
                System.out.println(data);
            }
            ledor.close();
        } catch (FileNotFoundException e) {
            System.out.println("Algo deu errado!");
            e.printStackTrace();
        }
    }
}
```

# Processando Arquivos File



■ Exemplo 05 – deletando um arquivo:

```
import java.io.File;

public class ApagaArquivo {

    public static void main(String[] args) {

        File meuArquivo = new File(d:/Curso_Java/meu_arquivo.txt);

        if (meuArquivo.delete()) {
            System.out.println("Arquivo apagado: " + meuArquivo.getName());
        }
        else {
            System.out.println("Algo deu errado!");
        }
    }
}
```

# Processando Arquivos File



■ Exemplo 06 – deletando uma pasta:

```
import java.io.File;

public class DeleteFolder {

    public static void main(String[] args) {

        File meuArquivo = new File("d:/Curso_Java/");

        if (meuArquivo.delete()) {
            System.out.println("Pasta apagada: " + meuArquivo.getName());
        }
        else {
            System.out.println("Algo deu errado!");
        }
    }
}
```

Obrigado.