



# Programação de Computadores em Java

## Informações em

<http://www.dcc.ufmg.br/~camarao/ipcj/>

<b>Prefácio</b>	<b>Sumário</b>	<b>Errata</b>
-----------------	----------------	---------------

<b>Transparências</b>
-----------------------

<b>Livros de referência à linguagem Java</b>
--

<b>Tutoriais</b>	<b>Bibliotecas (<i>APIs</i>)</b>
------------------	----------------------------------

<b>Ambientes de Programação</b>
---------------------------------

<b>Artigos sobre POO e Java</b>
---------------------------------





---

---

**Visão geral**  
**contextualização**



---

Cap. 1: Computadores e Programas

Cap. 2: Paradigmas de Programação

Programação baseada em objetos

---

**Visão geral**  
**contextualização**



---

Cap. 1: Computadores e Programas

Cap. 2: Paradigmas de Programação

Programação baseada em objetos

**Visão geral**  
**contextualização**

---

**Programação imperativa**  
**passo**  
**a passo**

---



---

Cap. 1: Computadores e Programas

Cap. 2: Paradigmas de Programação

Programação baseada em objetos

**Visão geral**  
**contextualização**

---

Cap. 3: Primeiros Problemas

Cap. 4: Entrada e Saída: Parte I

Cap. 5: Recursão e Iteração

**Programação imperativa**  
**passo**  
**a passo**

---



---

Cap. 1: Computadores e Programas

Cap. 2: Paradigmas de Programação

Programação baseada em objetos

**Visão geral**  
**contextualização**

---

Cap. 3: Primeiros Problemas

Cap. 4: Entrada e Saída: Parte I

Cap. 5: Recursão e Iteração

**Programação imperativa**  
**passo**  
**a passo**

---

**Visão geral: POO**

---





---

Cap. 1: Computadores e Programas

Cap. 2: Paradigmas de Programação

Programação baseada em objetos

**Visão geral**  
**contextualização**

---

Cap. 3: Primeiros Problemas

Cap. 4: Entrada e Saída: Parte I

Cap. 5: Recursão e Iteração

**Programação imperativa**  
**passo**  
**a passo**

---

Cap. 6: Classes, Subclasses e Herança

---

**Visão geral: POO**



---

Cap. 1: Computadores e Programas

Cap. 2: Paradigmas de Programação

Programação baseada em objetos

**Visão geral  
contextualização**

---

Cap. 3: Primeiros Problemas

Cap. 4: Entrada e Saída: Parte I

Cap. 5: Recursão e Iteração

**Programação imperativa  
passo  
a passo**

---

Cap. 6: Classes, Subclasses e Herança

**Visão geral: POO**

---

**Mais sobre  
programação  
imperativa**

---

Cap. 1: Computadores e Programas

Cap. 2: Paradigmas de Programação

Programação baseada em objetos

**Visão geral  
contextualização**

Cap. 3: Primeiros Problemas

Cap. 4: Entrada e Saída: Parte I

Cap. 5: Recursão e Iteração

**Programação imperativa  
passo  
a passo**

Cap. 6: Classes, Subclasses e Herança

**Visão geral: POO**

Cap. 7: Exceções

Cap. 8: Entrada e Saída: Parte II

Cap. 9: Arranjos

**Mais sobre  
programação  
imperativa**



# Computadores e Programas

- Algoritmos e programas
- Organização de computadores
- Linguagens, compiladores e interpretadores



# Algoritmo e Programa

- **Algoritmo:** Conjunto de regras especificando como realizar uma tarefa
- **Programa:** Representação desse conjunto de regras

Exemplos de “programas”:



# Algoritmo e Programa

- **Algoritmo:** Conjunto de regras especificando como realizar uma tarefa
- **Programa:** Representação desse conjunto de regras

Exemplos de “programas”:

receitas culinárias, instruções de montagem (brinquedos, aparelhos),  
partituras musicais, programas de computadores.



# Organização de computadores

**Arquitetura de von-Nëumann:**



# Organização de computadores

## Arquitetura de von-Nëumann:

- programa a ser executado armazenado na memória





# Organização de computadores

## Arquitetura de von-Nëumann:

- programa a ser executado armazenado na memória
- instrução buscada na memória, armazenada em um registrador



# Organização de computadores

## Arquitetura de von-Nëumann:

- programa a ser executado armazenado na memória
- instrução buscada na memória, armazenada em um registrador
- e executada



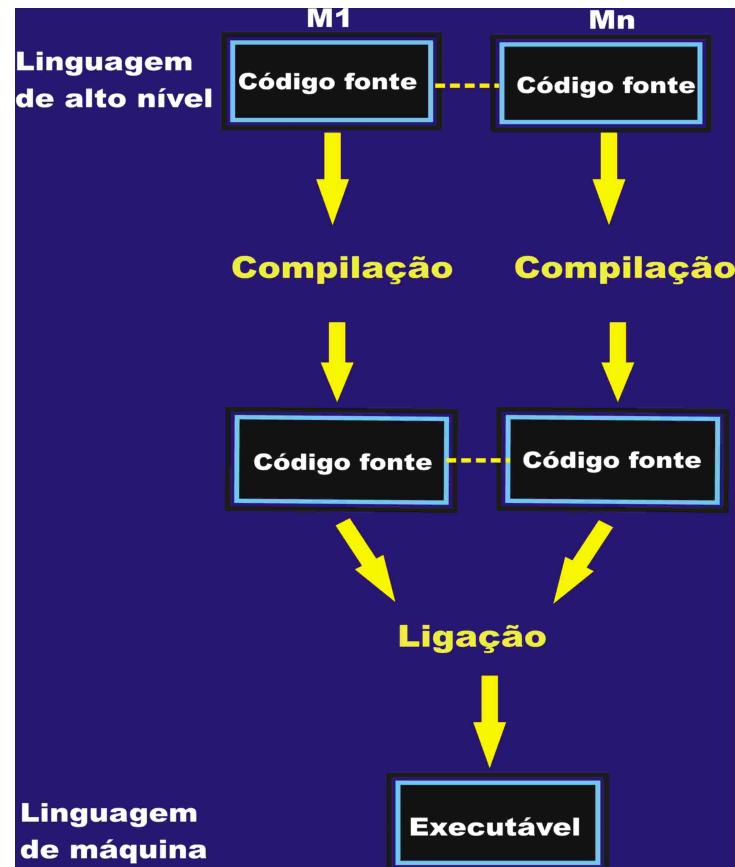
# Organização de computadores

## Arquitetura de von-Nëumann:

- programa a ser executado armazenado na memória
- instrução buscada na memória, armazenada em um registrador
- e executada
- (endereço da próxima instrução a ser executada, armazenado em outro registrador, atualizado)

 Computadores e programas

# Linguagens, Compiladores e Interpretadores





# Linguagens, Compiladores e Interpretadores

Compiladores são tradutores

- Programas gerados geralmente em linguagem de mais baixo nível.
- Compilação inclui em geral “montagem”: tradução de linguagem de montagem para linguagem de máquina.
- Interpretação pode ser usada em vez de ligação + execução.
- Execução pode ser vista como *interpretação em hardware*

# Linguagens, Compiladores e Interpretadores

Ambientes de programação usuais
editar–compilar–ligar–executar ou
editar–compilar–interpretar

Outras “ferramentas” de software úteis:

- **Depurador:** Ajuda na correção de erros de execução (funções para acompanhamento e impressão de dados de programa em execução)
- **Profilador:** ajuda na análise de gastos em tempo e memória

# Paradigmas



# Paradigmas



**Tradicional** Fortran, Algol, Algol-68, Pascal, C, Cobol, PL/I



# Paradigmas



**Tradicional OO** Fortran, Algol, Algol-68, Pascal, C, Cobol, PL/I  
Simula-67, Smalltalk, C++, Eiffel, Object Pascal, Java, C#

# Paradigmas





<b>Tradicional</b>	Fortran, Algol, Algol-68, Pascal, C, Cobol, PL/I
<b>OO</b>	Simula-67, Smalltalk, C++, Eiffel, Object Pascal, Java, C#
<b>Funcional</b>	Lisp, ML, Scheme, Miranda, Haskell

# Paradigmas



<b>Tradicional</b>	Fortran, Algol, Algol-68, Pascal, C, Cobol, PL/I
<b>OO</b>	Simula-67, Smalltalk, C++, Eiffel, Object Pascal, Java, C#
<b>Funcional</b>	Lisp, ML, Scheme, Miranda, Haskell
<b>Lógico</b>	Prolog, Mercury



# Paradigma Imperativo

## visão global / conceituação

- Variável e atribuição
- Comandos
  - ★ Composição seqüencial
  - ★ Seleção
  - ★ Repetição
- Funções e procedimentos

# Modificação do valor de variáveis: base da programação imperativa

- **Variável:** *lugar* (posição na memória) que contém um certo *valor* (difere do usual em matemática!)
- Valor armazenado em uma variável pode ser modificado por meio de um **comando de atribuição**
- Execução baseada em **comandos**, que modificam / controlam a modificação de valores de variáveis



# Declaração de variáveis

```
boolean x; int y = 10;
```

# Declaração de variáveis

```
boolean x; int y = 10;
```

- Em Java (e LPs em geral), toda variável deve ser *declarada*.

# Declaração de variáveis

```
boolean x; int y = 10;
```

- Em Java (e LPs em geral), toda variável deve ser *declarada*.
- Declaração especifica nome e *tipo*



# Declaração de variáveis

```
boolean x; int y = 10;
```

- Em Java (e LPs em geral), toda variável deve ser *declarada*.
- Declaração especifica nome e *tipo*
- **Tipo determina conjunto de valores que podem ser armazenados na variável**

# Declaração de variáveis

```
boolean x; int y = 10;
```

- Em Java (e LPs em geral), toda variável deve ser *declarada*.
- Declaração especifica nome e *tipo*
- **Tipo determina conjunto de valores que podem ser armazenados na variável**
- Declaração pode especificar valor inicial (valor armazenado no instante da criação)



# Atribuição

$$v = e;$$

# Atribuição

$$v = e;$$

- Execução: expressão  $e$  é avaliada e valor resultante atribuído à variável  $v$

# Atribuição

$$v = e;$$

- Execução: expressão  $e$  é avaliada e valor resultante atribuído à variável  $v$
- Após atribuição, valor anterior de  $v$  “é perdido”  
(não pode ser mais obtido usando  $v$ , a não ser que nova atribuição seja feita)



# Atribuição

$$a = b = b + 1;$$

# Atribuição

```
a = b = b + 1;
```

- Comando de atribuição é expressão em Java

# Atribuição

```
a = b = b + 1;
```

- Comando de atribuição é expressão em Java
- Não confundir:  $a = b$  com  $a == b$



# Comandos

Composição de comandos estabelece ordem de execução  
(determina ordem de modificação do valor de variáveis)

- Composição seqüencial

$$c_1; c_2;$$

Execução de  $c_1$  e, em seguida,  $c_2$

```
 $a = 10; b = \text{true}; c = 2*a;$ 
```

# Comandos

- Seleção (comando if) 

<code>if ( <i>b</i> ) <i>c</i><sub>1</sub>; else <i>c</i><sub>2</sub>;</code>
---

Se a avaliação de *b* retornar `true`, *c*<sub>1</sub> é executado;  
se `false`, *c*<sub>2</sub> é executado.

Cláusula `else` opcional: ausência  $\Rightarrow$   
nenhum comando é executado se avaliação de *b* retornar `false`.

```
if ( a > 10 ) { a = a + 10; b = b + 1; }  
    else { b = 0; if ( c > 1 ) a = a + 5; }
```

# Comandos

- Repetição

```
while ( b ) c;
```

Expressão *b* é avaliada; se resultado for true, *c* é executado, e o processo se repete; se false, execução termina

```
soma = 0; i = 1;  
while ( i <= n )  
{ soma = soma + i; i = i + 1; }
```



# Funções e Procedimentos

## Mecanismos de **abstração**

- **Funções**: fornecem um resultado, de acordo com argumentos  
Ex: + fornece resultado da adição, de dois argumentos
- **Procedimentos**: modificam valores de variáveis, de acordo com argumentos
- Em Java (e LOOs em geral), funções e procedimentos são casos especiais de **métodos**

 Paradigma Imperativo

# Primeiros Problemas

```
class PrimeirosExemplos
{ static int quadrado (int x) { return x*x; }

```

escute

```
static int somaDosQuadrados (int x, int y)
{ return (quadrado(x) + quadrado(y)); }

```

analise

```
static boolean tresIguais (int a, int b, int c)
{ return ((a==b) && (b==c)); }

```

pergunte

```
static boolean eTriang (int a, int b, int c)
// a, b e c positivos e cada um menor do que a soma dos outros dois
{ return (a>0) && (b>0) && (c>0) &&
        (a<b+c) && (b<a+c) && (c<a+b); }

```

pense

# Continuação: *PrimeirosExemplos*

```
static int max (int a, int b)
{ if (a >= b) return a; else return b;}

static int max3 (int a, int b, int c)
{ return (max(max(a,b),c)); }

} // fim PrimeirosExemplos
```

## Associe itens nos quadros abaixo:

- |  |                               |
|--|-------------------------------|
| <b>a.</b> <code>return e</code>  | <b>b.</b> <code>static</code> |
| <b>c.</b> <code>static tipo nome(lista-de-parâmetros) { corpo }</code> |                               |
| <b>d.</b> <code>nome(lista-de-argumentos)</code>                       |                               |

- |  |
|--|
| <b>1.</b> forma (sintaxe) de chamada de função ou procedimento<br>(na mesma classe em que é definida(o)) |
| <b>2.</b> pode ser entendido por enquanto como indicação de definição de<br>função/procedimento          |
| <b>3.</b> forma (sintaxe) de definição de função ou procedimento   |
| <b>4.</b> deve ocorrer no corpo de função; especifica<br>resultado de chamada                            |

# Construções sintáticas

- $e$  representa uma expressão
- lista-de-parâmetros é elemento sintático na forma  $\text{tipo}_1 \text{nome}_1, \text{tipo}_2 \text{nome}_2, \dots, \text{tipo}_n \text{nome}_n$  ( $n \geq 0$ )
- corpo é uma lista de comandos  $c_1; c_2; \dots c_n;$
- lista-de-argumentos é da forma  $\text{exp}_1, \text{exp}_2, \dots, \text{exp}_n$  (para  $n \geq 0$ )



# Operadores de comparação

Operador	Significado	Exemplo	Resultado
==	Igual a	1 == 1	true
!=	Diferente de	1 != 1	false
<	Menor que	1 < 1	false
>	Maior que	1 > 1	false
<=	Menor ou igual a	1 <= 1	true
>=	Maior ou igual a	1 >= 1	true

 Primeiros Problemas

# Expressão condicional

Expressão condicional:  $e \ ? \ e_1 \ : \ e_2$

Comando condicional: `if (e) c1; else c2;`

Definições alternativas de *max* e *max3*:

```
int max (int a, int b)
{ return (a >= b ? a : b); }
```

```
int max3 (int a, int b, int c)
{ return (a >= b ? max(a, c) : max(b, c)); }
```

# Constantes

```
static final int NaoETriang = 0, Equilatero = 1,  
               Isosceles     = 2, Escaleno    = 3;  
  
static int t_ou_nao_t (int a, int b, int c)  
{ if (eTriang(a,b,c))  
    if (a==b && b==c) return Equilatero;  
    else if (a==b || b==c || a==c) return Isosceles;  
    else return Escaleno;  
else return NaoETriang; }
```

# Operadores aritméticos

Operador	Significado	Exemplo	Resultado
+	Adição	2 + 1	3
		2 + 1.0	3.0
		2.0 + 1.0	3.0
-	Subtração	2 - 1	1
		2.0 - 1	1.0
-	Negação	-1	-1
		-1.0	-1.0
*	Multiplicação	2 * 3	6
		2.0 * 3	6.0
/	Divisão	5 / 2	2
		5 / 2.0	2.5
%	Resto	5 % 2	1
		5.0 % 2.0	1.0

# Ordem de avaliação de expressões

**Da esquerda para a direita,  
mas respeitando precedência de operadores e uso de parênteses**

Não é boa técnica de programação definir programas em que resultado dependa de efeitos colaterais e ordem de avaliação:

```
class Exemplo
{ public static void main (String[] a)
  { int i = 10;
    int j = (i=2) * i;
    System.out.println(j);
  }
}
```

# Precedência de operadores

Precedência maior	
Operadores	Exemplos
$*, /, \%$	$i * j / k \equiv$

# Precedência de operadores

Precedência maior	
Operadores	Exemplos
$*, /, \%$	$i * j / k \equiv (i * j) / k$
$+, -$	$i + j * k \equiv$

# Precedência de operadores

Precedência maior	
Operadores	Exemplos
$*, /, \%$	$i * j / k \equiv (i * j) / k$
$+, -$	$i + j * k \equiv i + (j * k)$
$>, <, >=, <=$	$i < j + k \equiv$



# Precedência de operadores

Precedência maior	
Operadores	Exemplos
$*, /, \%$	$i * j / k \equiv (i * j) / k$
$+, -$	$i + j * k \equiv i + (j * k)$
$>, <, >=, <=$	$i < j + k \equiv i < (j + k)$
$==, !=$	$b == i < j \equiv$

# Precedência de operadores

Precedência maior	
Operadores	Exemplos
$*, /, \%$	$i * j / k \equiv (i * j) / k$
$+, -$	$i + j * k \equiv i + (j * k)$
$>, <, >=, <=$	$i < j + k \equiv i < (j + k)$
$==, !=$	$b == i < j \equiv b == (i < j)$
$\&$	$b \& b1 == b2 \equiv$

# Precedência de operadores

Precedência maior	
Operadores	Exemplos
$*, /, \%$	$i * j / k \equiv (i * j) / k$
$+, -$	$i + j * k \equiv i + (j * k)$
$>, <, >=, <=$	$i < j + k \equiv i < (j + k)$
$==, !=$	$b == i < j \equiv b == (i < j)$
$\&$	$b \& b1 == b2 \equiv b \& (b1 == b2)$
$\wedge$	$b1 \wedge b2 \& b \equiv$

# Precedência de operadores

Precedência maior	
Operadores	Exemplos
$*, /, \%$	$i * j / k \equiv (i * j) / k$
$+, -$	$i + j * k \equiv i + (j * k)$
$>, <, >=, <=$	$i < j + k \equiv i < (j + k)$
$==, !=$	$b == i < j \equiv b == (i < j)$
$\&$	$b \& b1 == b2 \equiv b \& (b1 == b2)$
$\wedge$	$b1 \wedge b2 \& b \equiv b1 \wedge (b2 \& b)$
$ $	$i < j   b1 \& b2 \equiv$

# Precedência de operadores

Precedência maior	
Operadores	Exemplos
$*, /, \%$	$i * j / k \equiv (i * j) / k$
$+, -$	$i + j * k \equiv i + (j * k)$
$>, <, >=, <=$	$i < j + k \equiv i < (j + k)$
$==, !=$	$b == i < j \equiv b == (i < j)$
$\&$	$b \& b1 == b2 \equiv b \& (b1 == b2)$
$\wedge$	$b1 \wedge b2 \& b \equiv b1 \wedge (b2 \& b)$
$ $	$i < j   b1 \& b2 \equiv (i < j)   (b1 \& b2)$
$\&\&$	$i + j != k \&\& b1 \equiv$

# Precedência de operadores

Precedência maior	
Operadores	Exemplos
$*, /, \%$	$i * j / k \equiv (i * j) / k$
$+, -$	$i + j * k \equiv i + (j * k)$
$>, <, >=, <=$	$i < j + k \equiv i < (j + k)$
$==, !=$	$b == i < j \equiv b == (i < j)$
$\&$	$b \& b1 == b2 \equiv b \& (b1 == b2)$
$\wedge$	$b1 \wedge b2 \& b \equiv b1 \wedge (b2 \& b)$
$ $	$i < j   b1 \& b2 \equiv (i < j)   (b1 \& b2)$
$\&\&$	$i + j != k \&\& b1 \equiv ((i + j) != k) \&\& b1$
$  $	$i1 < i2    b \&\& j < k \equiv$

# Precedência de operadores

Precedência maior	
Operadores	Exemplos
$*, /, \%$	$i * j / k \equiv (i * j) / k$
$+, -$	$i + j * k \equiv i + (j * k)$
$>, <, >=, <=$	$i < j + k \equiv i < (j + k)$
$==, !=$	$b == i < j \equiv b == (i < j)$
$\&$	$b \& b1 == b2 \equiv b \& (b1 == b2)$
$\wedge$	$b1 \wedge b2 \& b \equiv b1 \wedge (b2 \& b)$
$ $	$i < j   b1 \& b2 \equiv (i < j)   (b1 \& b2)$
$\&\&$	$i + j != k \&\& b1 \equiv ((i + j) != k) \&\& b1$
$  $	$i1 < i2    b \&\& j < k \equiv (i1 < i2)    (b \&\& (j < k))$
$? :$	$i < j    b ? b1 : b2 \equiv$

# Precedência de operadores

Precedência maior	
Operadores	Exemplos
$*, /, \%$	$i * j / k \equiv (i * j) / k$
$+, -$	$i + j * k \equiv i + (j * k)$
$>, <, >=, <=$	$i < j + k \equiv i < (j + k)$
$==, !=$	$b == i < j \equiv b == (i < j)$
$\&$	$b \& b1 == b2 \equiv b \& (b1 == b2)$
$\wedge$	$b1 \wedge b2 \& b \equiv b1 \wedge (b2 \& b)$
$ $	$i < j   b1 \& b2 \equiv (i < j)   (b1 \& b2)$
$\&\&$	$i + j != k \&\& b1 \equiv ((i + j) != k) \&\& b1$
$  $	$i1 < i2    b \&\& j < k \equiv (i1 < i2)    (b \&\& (j < k))$
$? :$	$i < j    b ? b1 : b2 \equiv ((i < j)    b) ? b1 : b2$
Precedência menor	



# Tipos numéricos

Classificação	Nome	Tamanho <sup>1</sup>	Exemplos de literal
inteiro	byte	8	$\overline{A}$ <sup>2</sup>
	short	16	$\overline{A}$
	int	32	10
	long	64	10L    10l <sup>3</sup>
de ponto flutuante	float	32	2.718f    2e2f
	double	64	2.718    2e2

<sup>1</sup>Número de bits usado para representar valores do tipo.

<sup>2</sup>Quando um literal inteiro é usado em um contexto que requer um valor de tipo byte ou short, ocorre automaticamente (implicitamente) uma conversão de tipo.

<sup>3</sup>A letra l minúscula pode, mas em geral não deve, ser usada, por causar confusão com 1.

# Operadores booleanos

Operação "não"	Resultado
!true	false
!false	true

Operação	Resultado		
	( "ê" ) op = & ou &&	( "ou" ) op =   ou	( "ou exclusivo" ) op = ^
true op true	true	true	false
true op false	false	true	true
false op true	false	true	true
false op false	false	false	false

# Operadores booleanos

Operador	Significado
!	Negação (“não”)
& e &&	Conjunção estrita e não-estrita, resp.
e	Disjunção estrita e não-estrita, resp.
^	Disjunção exclusiva (“ou exclusivo”), estrita.

- Avaliação de  $e_1 \ \&\& \ e_2$  retorna false se a de  $e_1$  retornar false; caso contrário, resultado de  $e_2$ . Note:  $e_2$  só é avaliada se avaliação de  $e_1$  retornar true. Def. de  $||$  é análoga.
- Função estrita e não estrita: ►

# Funções estritas

- Considere que  $\perp$  representa “valor indefinido”: resultante de avaliação que nunca termina (como, veremos, é possível) ou que provoca a ocorrência de um erro (como, por ex., divisão por zero).
- **Função  $f$  estrita:**  $f(\perp) = \perp$
- $f$  não-estrita se avaliação puder fornecer resultado mesmo que avaliação de argumento não possa.
- **&& não-estrita** (como função que recebe dois valores).  
Ex: `(false && (0/0==0))` é igual a `false`.

# Caracteres

- Escritos entre aspas simples: `'a'` `'*'` `'3'` etc.
- “Caracteres de controle” (ex: fim de arquivo, tabulação, mudança de linha etc.), `'` e `\` escritos usando `\` :
  - `'\n'` indica terminação de linha      `'\t'` indica tabulação
  - `'\"'` indica o caractere `'`      `'\\'` indica o caractere `\`
- Representação binária: chamada de *código* do caractere.
- Codificação (“código”) *Unicode* usada em Java: associa um código para cada caractere. São usados dois bytes para cada caractere.

# Caracteres

```
static boolean minusc (char x)  
{ return (x >= 'a') && (x <= 'z'); }
```

```
static boolean maiusc (char x)  
{ return (x >= 'A') && (x <= 'Z'); }
```

```
static boolean digito (char x)  
{ return (x >= '0') && (x <= '9'); }
```

# Caracteres

Valor de tipo char pode ser usado como inteiro  
(de 16 bits, não-negativo)

```
static char minusc_maiusc (char x)  
{ int d = 'A' - 'a';  
  if (minusc(x)) return (x+d);  
  else return x;  
}
```



# Caracteres

Caracteres podem ser expressos por meio do valor da sua representação no código *Unicode*.

'\u0000'	caractere nulo
'\u0009'	caractere de tabulação ('\t')
'\u0097'	letra minúscula 'a'

 Primeiros Problemas





# Programação baseada em objetos: primeiras noções

**Projeto de sistemas** baseado em **paradigma de simulação**:

entidade do sistema  $\Leftrightarrow$  **objeto**

Objeto criado durante execução do programa, com atributos e comportamento descritos no programa, simula comportamento de entidade do sistema.

## Possíveis fases do desenvolvimento de software



# Classes e objetos

- **Classe:** parte de um programa que define “estrutura” e “comportamento” de grupo de objetos
- **Objeto** — *instância* de uma classe — existe durante a execução de um programa
- Estrutura definida por **variáveis de objeto** (ou variáveis de instância)
- Comportamento definido por **métodos e construtores**.



```
class Ponto
{ int x, y;

    Ponto (int a, int b) { x = a; y = b; }

    void move (int dx, int dy) {x+=dx; y+=dy; }

    double distancia (Ponto p)
    { int dx = this.x - p.x;
      int dy = this.y - p.y;
      return Math.sqrt(dx*dx + dy*dy); }
}

class TestePonto
{ public static void main(String[] a)
  { Ponto p1 = new Ponto(0,0);
    Ponto p2 = new Ponto(10,20);
    p1.move(3,25);
    p2.move(1,14);
    System.out.println(p1.distancia(p2)); }
}
```

escute

analise

pergunte

pense

# Criação de objetos

- Objeto criado por *efeito colateral* de expressão de criação de objeto — ex: `new Ponto(0,0)` — que retorna referência ao objeto criado.
- Referência ao objeto criado significa, em termos de implementação, endereço do início da área de memória alocada para as variáveis do objeto.

# Valor inicial de variáveis

**Variável de objeto ou de classe** tem valor inicial *default*, atribuído se nenhum valor inicial for especificado na declaração, que depende do tipo da variável:

byte, short, int, long	0
char	'\u0000'
boolean	false
float, double	0.0
classes	null

Ao contrário de variáveis locais de métodos.

# Chamada de método

$$\boxed{\boxed{exp.método(param_1, \dots, param_n)}}$$

- Tipo de *exp* deve ser alguma classe
- Deve existir definição do método *método* nessa classe
- Essa definição deve especificar tipos para os parâmetros formais que são compatíveis com os tipos dos parâmetros reais  $param_1, \dots, param_n$ , respectivamente.

## Associe itens nos quadros abaixo:

- a. variáveis de objeto
- b. método estático (método-de-classe) da classe *TestePonto*
- c. tipo do resultado da chamada *p1.distancia(p2)*
- d. referência ao objeto corrente
- e. parâmetro de método
- f. expressão de criação de objeto (retorna referência ao objeto criado)
- g. método estático da classe *Math*
- h. métodos

- |   |                          |
|---|--------------------------|
| 1. <i>main</i>                                  | 2. <i>sqrt</i>           |
| 3. <i>p</i> , do tipo <i>Ponto</i>              | 4. <i>new Ponto(0,0)</i> |
| 5. <i>x</i> , <i>y</i>                          | 6. <i>double</i>         |
| 7. <i>move</i> , <i>distancia</i> , <i>main</i> | 8. <i>this</i>           |



# Indique Falso ou Verdadeiro

- Definição de construtor não precisa especificar tipo do resultado, como no caso de métodos, porque “tipo = nome do construtor”
- Construtor tem sempre mesmo nome da classe em que ocorre
- Uma classe pode ter mais de um construtor
- Método estático funciona como função ou procedimento
- `static void` em método indica que tal método tem comportamento de procedimento: chamada sem especificação de *objeto alvo* e nenhum valor retornado.


## Associe itens nos quadros abaixo:

- a. resultado de `new Ponto(0,0)`
- b. valor *default* armazenado em variável de tipo-classe, se nenhuma inicialização for especificada na sua declaração
- c. avaliação de expressão de criação de objeto (`new`)
- d. objetivo de parâmetros de construtores
- e. denominação dada ao objeto denotado por `p1` em `p1.move(3,25)`

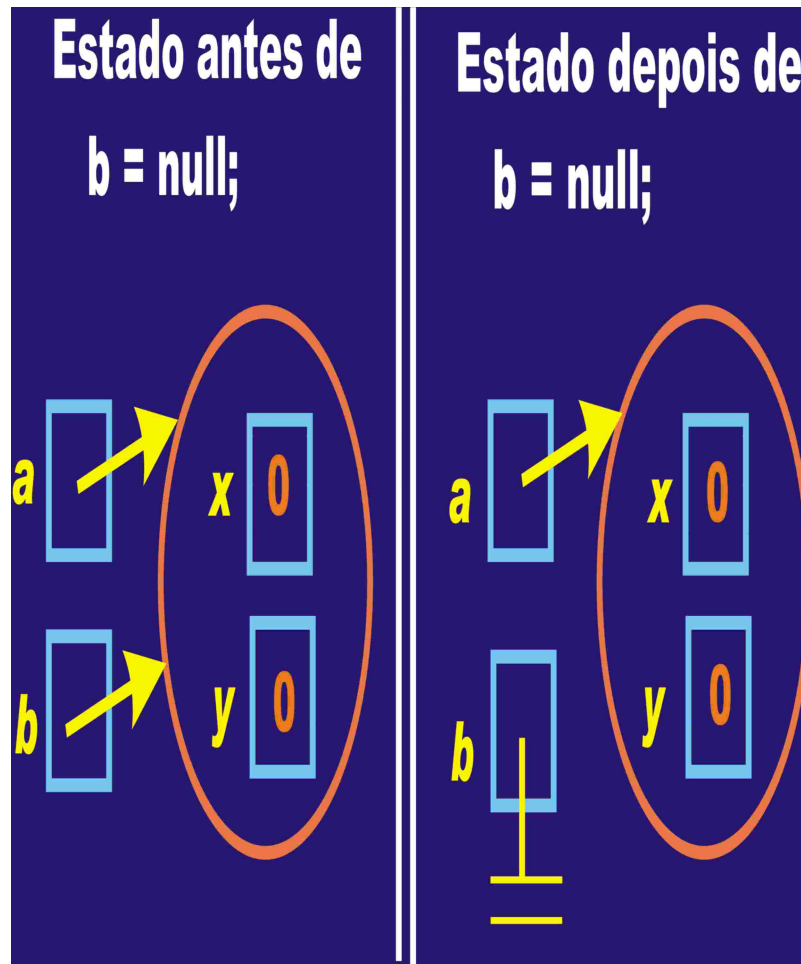
- 1. consiste em criar objeto (i.e. suas variáveis), executar corpo do construtor, e finalmente retornar referência ao objeto criado
- 2. `null`
- 3. referência ao objeto criado, e não o objeto propriamente dito
- 4. objeto alvo da chamada
- 5. permitir atribuição de valores iniciais a variáveis de objetos/classes



# Referências

- Em Java, uma variável de tipo-classe contém uma referência a um objeto (não o objeto propriamente dito)
- Atribuição apenas copia referências 

Atribuição de objetos = atribuição de referências



```
Ponto a =  
    new Ponto(0,0);  
Ponto b;  
  
b = a;  
b = null;
```

## Variáveis e métodos de objeto e de classe

Variáveis de classe (variáveis estáticas) armazenam valores (informação) comuns a todos os objetos da classe.

Um método de classe só pode usar (diretamente) variáveis de classe

```
class C
{ static int x; int y;

  static int m1 (int p)
  { x = p+1; }

  int m2 (int p)
  { x = p+2; y = p+1; }
}
```

# Variáveis de classe

Variáveis de classe

são comuns a

todos os objetos

```
class C
{ static int x=0;

    public static void main(String[] a)
    { C c1 = new C();
      C c2 = new C();

      c1.x = 1;
      System.out.println(c2.x);
    }
}
```

# Cadeias de caracteres

- Em Java (e LOOs em geral) cadeias de caracteres são objetos, da classe *String*.
- Literal entre aspas duplas (ex: `"a3*\n"`) cria objeto.
- Posição do primeiro caractere, em objeto do tipo *String*, é igual a zero, do segundo igual a um, e assim por diante.

# Operações sobre cadeias de caracteres

Indexação	<i>charAt</i>
Tamanho (nº de caracteres)	<i>length</i>

Sendo  $s$  expressão que representa objeto da classe *String*,  
 $e$  expressão que denota valor inteiro não-negativo  $n$ :

$s.charAt(e)$

retorna caractere na posição  $n$  de  $s$



# Operações sobre cadeias de caracteres

Indexação	<i>charAt</i>
Tamanho (nº de caracteres)	<i>length</i>

Sendo  $s$  expressão que representa objeto da classe *String*,  
 $e$  expressão que denota valor inteiro não-negativo  $n$ :

$s.charAt(e)$

$"xpto".charAt(0)$

retorna caractere na posição  $n$  de  $s$

# Operações sobre cadeias de caracteres

Indexação	<i>charAt</i>
Tamanho (nº de caracteres)	<i>length</i>

Sendo  $s$  expressão que representa objeto da classe *String*,  
 $e$  expressão que denota valor inteiro não-negativo  $n$ :

$s.charAt(e)$

"xpto". <i>charAt</i> (0)	'x'
"xpto". <i>charAt</i> (3)	

retorna caractere na posição  $n$  de  $s$

# Operações sobre cadeias de caracteres

Indexação	<i>charAt</i>
Tamanho (nº de caracteres)	<i>length</i>

Sendo  $s$  expressão que representa objeto da classe *String*,  
 $e$  expressão que denota valor inteiro não-negativo  $n$ :

$s.charAt(e)$

retorna caractere na posição  $n$  de  $s$

"xpto". <i>charAt</i> (0)	'x'
"xpto". <i>charAt</i> (3)	'o'
"xpto". <i>length</i> ()	

# Operações sobre cadeias de caracteres

Indexação	<i>charAt</i>
Tamanho (nº de caracteres)	<i>length</i>

Sendo  $s$  expressão que representa objeto da classe *String*,  
 $e$  expressão que denota valor inteiro não-negativo  $n$ :

$s.charAt(e)$

retorna caractere na posição  $n$  de  $s$

"xpto". <i>charAt</i> (0)	'x'
"xpto". <i>charAt</i> (3)	'o'
"xpto". <i>length</i> ()	4

# Classes invólucros

- Todo programa Java importa automaticamente a biblioteca *java.lang*.
- Essa biblioteca contém (dentre outras) as chamadas “classes invólucros”:

<i>Boolean</i>	<i>Byte</i>	<i>Character</i>
<i>Short</i>	<i>Integer</i>	<i>Long</i>
<i>Double</i>	<i>Float</i>	

## Conversão de cadeia de caracteres em valor básico

Para	Método	Exemplo de expressão
int	<i>parseInt</i>	<i>Integer.parseInt(s)</i>
long	<i>parseLong</i>	<i>Long.parseLong(s)</i>
float	<i>parseFloat</i>	<i>Float.parseFloat(s)</i>
double	<i>parseDouble</i>	<i>Double.parseDouble(s)</i>
boolean	<i>valueOf,</i> <i>booleanValue</i>	<i>Boolean.valueOf(s).booleanValue()</i>

## De cadeia de caracteres para objeto de classe invólucro

Seja  $C$  classe invólucro e  $s$  expressão do tipo *String*.

$C.valueOf(s)$

cria objeto da classe  $C$  e armazena nesse objeto resultado da conversão de cadeia de caracteres  $s$  para tipo básico correspondente (ou causa a exceção *NumberFormatException*, caso a cadeia de caracteres não represente um valor do tipo desejado).

Ex:  $Integer.valueOf(s)$  retorna um valor igual ao fornecido por:

$\text{new } Integer(Integer.parseInt(s))$

# Conversão para cadeia de caracteres

- método estático *toString* de classes invólucros

<i>Integer.toString</i> (123)
-------------------------------



# Conversão para cadeia de caracteres

- método estático *toString* de classes invólucros

<i>Integer.toString</i> (123)	"123"
<i>Double.toString</i> (0.1)	

# Conversão para cadeia de caracteres

- método estático *toString* de classes invólucros

<i>Integer.toString</i> (123)	"123"
<i>Double.toString</i> (0.1)	"0.1"
<i>Float.toString</i> (1e-1f)	

# Conversão para cadeia de caracteres

- método estático *toString* de classes invólucros

<i>Integer.toString</i> (123)	"123"
<i>Double.toString</i> (0.1)	"0.1"
<i>Float.toString</i> (1e-1f)	"0.1"

- Método sobrecarregado *valueOf* da classe *String*:

<i>String.valueOf</i> (10)	
----------------------------	--

# Conversão para cadeia de caracteres

- método estático *toString* de classes invólucros

<i>Integer.toString</i> (123)	"123"
<i>Double.toString</i> (0.1)	"0.1"
<i>Float.toString</i> (1e-1f)	"0.1"

- Método sobrecarregado *valueOf* da classe *String*:

<i>String.valueOf</i> (10)	"10"
<i>String.valueOf</i> (0.1)	

# Conversão para cadeia de caracteres

- método estático *toString* de classes invólucros

<i>Integer.toString</i> (123)	"123"
<i>Double.toString</i> (0.1)	"0.1"
<i>Float.toString</i> (1e-1f)	"0.1"

- Método sobrecarregado *valueOf* da classe *String*:

<i>String.valueOf</i> (10)	"10"
<i>String.valueOf</i> (0.1)	"0.1"
<i>String.valueOf</i> (1e-1f)	

# Conversão para cadeia de caracteres

- método estático *toString* de classes invólucros

<i>Integer.toString</i> (123)	"123"
<i>Double.toString</i> (0.1)	"0.1"
<i>Float.toString</i> (1e-1f)	"0.1"

- Método sobrecarregado *valueOf* da classe *String*:

<i>String.valueOf</i> (10)	"10"
<i>String.valueOf</i> (0.1)	"0.1"
<i>String.valueOf</i> (1e-1f)	"0.1"

# Classe *Character*

```
public static boolean  isDigit   (char c)
public static boolean  isLetter  (char c)
public static boolean  isLetterOrDigit (char c)
public static boolean  isLowerCase (char c)
public static boolean  isUpperCase (char c)
public static boolean  isSpace    (char c)
public static char      toLowerCase (char c)
public static char      toUpperCase (char c)
public char             charValue  ()
```

# Concatenação de cadeias de caracteres

- Operador +
- Quando um dos argumentos é um valor básico chama implicitamente método *toString* depois de criar objeto da classe invólucro

"abcd" + "ef"
---------------



# Concatenação de cadeias de caracteres

- Operador +
- Quando um dos argumentos é um valor básico chama implicitamente método *toString* depois de criar objeto da classe invólucro

"abcd" + "ef"	"abcdef"
"abcd" + 1	

# Concatenação de cadeias de caracteres

- Operador +
- Quando um dos argumentos é um valor básico chama implicitamente método *toString* depois de criar objeto da classe invólucro

"abcd" + "ef"	"abcdef"
"abcd" + 1	"abcd1"
"abcd" + 1.0	

# Concatenação de cadeias de caracteres

- Operador +
- Quando um dos argumentos é um valor básico chama implicitamente método *toString* depois de criar objeto da classe invólucro

"abcd" + "ef"	"abcdef"
"abcd" + 1	"abcd1"
"abcd" + 1.0	"abcd1.0"
"abcd" + 1e-1f	

# Concatenação de cadeias de caracteres

- Operador +
- Quando um dos argumentos é um valor básico chama implicitamente método *toString* depois de criar objeto da classe invólucro

"abcd" + "ef"	"abcdef"
"abcd" + 1	"abcd1"
"abcd" + 1.0	"abcd1.0"
"abcd" + 1e-1f	"abcd0.1"
"abcd" + 1 + 2	

# Concatenação de cadeias de caracteres

- Operador +
- Quando um dos argumentos é um valor básico chama implicitamente método *toString* depois de criar objeto da classe invólucro

"abcd" + "ef"	"abcdef"
"abcd" + 1	"abcd1"
"abcd" + 1.0	"abcd1.0"
"abcd" + 1e-1f	"abcd0.1"
"abcd" + 1 + 2	"abcd12"
1 + 2 + "abcd"	

# Concatenação de cadeias de caracteres

- Operador +
- Quando um dos argumentos é um valor básico chama implicitamente método *toString* depois de criar objeto da classe invólucro

"abcd" + "ef"	"abcdef"
"abcd" + 1	"abcd1"
"abcd" + 1.0	"abcd1.0"
"abcd" + 1e-1f	"abcd0.1"
"abcd" + 1 + 2	"abcd12"
1 + 2 + "abcd"	"3abcd"

# Comparação de cadeias de caracteres

- Método *equals* testa igualdade de cadeias de caracteres (ou seja, dos caracteres componentes das cadeias).
- Ao contrário, `==` determina se objetos comparados são iguais (ou seja, se são o mesmo objeto).

# Comparação de cadeias de caracteres

- Cada literal da classe *String* representa um dado objeto.

Ex: "abcd" == "abcd" retorna true (apesar de **primeiro** uso de aspas duplas significar criação de objeto da classe *String*).

- Para isso, a cada uso de um literal, é necessário determinar, em tempo de execução, se o objeto correspondente já foi anteriormente criado. Em caso positivo, uma referência ao objeto é retornada. Em caso negativo, um novo objeto é inserido no conjunto de objetos criados.



# Comparação de cadeias de caracteres

"ab". <i>equals</i> ("ab")
----------------------------

# Comparação de cadeias de caracteres

<code>"ab".<i>equals</i>("ab")</code>	<code>true</code>
<code>"ab" == "ab"</code>	

# Comparação de cadeias de caracteres

<code>"ab".<i>equals</i>("ab")</code>	<code>true</code>
<code>"ab" == "ab"</code>	<code>true</code>
<code>"ab" == new String("ab")</code>	

# Comparação de cadeias de caracteres

<code>"ab".<i>equals</i>("ab")</code>	<code>true</code>
<code>"ab" == "ab"</code>	<code>true</code>
<code>"ab" == new String("ab")</code>	<code>false</code>

# Conversão de Tipo

$(t) \ e$

- Converte expressão  $e$  para o tipo  $t$ , se possível
- Caso contrário, um erro é detectado (em geral durante a compilação)



# Conversão de Tipo

- Tipos numéricos: **extensão** ou **truncamento**.
- **Extensão**: simples atribuição de valores apropriados aos bits adicionais da representação.
- **Truncamento** pode envolver mudança de valor.

# Conversão de Tipo

- Extensões ocorrem implicitamente, sem nunca provocar ocorrência de erro.
- Uma extensão de  $t_e$  para  $t$  pode envolver dois tipos quaisquer nas seguintes cadeias ( $t_e$  precedendo  $t$  na cadeia);

<pre>byte--short--int--long--float--double char --int--long--float--double</pre>
--



## Conversão de tipo

Porque existem duas cadeias de conversão implícita

**byte–short–int–long–float–double**

**char –int–long–float–double**

e não apenas uma ?



## Conversão de tipo

Porque existem duas cadeias de conversão implícita

**byte–short–int–long–float–double**

**char –int–long–float–double**

e não apenas uma ?

Porque uma conversão implícita (uma extensão) nunca pode modificar o valor representado nem causar um erro; portanto, byte e short, que podem ser negativos, não podem ser convertidos implicitamente para char, que é sempre não-negativo.



## Conversão de tipo: Certo ou Errado?

- `byte b = 128;`



## Conversão de tipo: Certo ou Errado?

- `byte b = 128;` Errado: 128 não pode ser representado em 1 byte (não pode ser convertido de `int` para `byte`).
- `byte b = -128;`

## Conversão de tipo: Certo ou Errado?

- `byte b = 128;` Errado: 128 não pode ser representado em 1 byte (não pode ser convertido de `int` para `byte`).
- `byte b = -128;` Certo.
- `byte a, b = 1; byte c = a+b;`

## Conversão de tipo: Certo ou Errado?

- `byte b = 128;` Errado: 128 não pode ser representado em 1 byte (não pode ser convertido de `int` para `byte`).
- `byte b = -128;` Certo.
- `byte a, b = 1; byte c = a+b;` Errado: `byte` é estendido para `int` para realização de `a+b`.
- `byte a, b = 1; byte c = (byte)(a+b);`

## Conversão de tipo: Certo ou Errado?

- `byte b = 128;` Errado: 128 não pode ser representado em 1 byte (não pode ser convertido de `int` para `byte`).
- `byte b = -128;` Certo.
- `byte a, b = 1; byte c = a+b;` Errado: `byte` é estendido para `int` para realização de `a+b`.
- `byte a, b = 1; byte c = (byte)(a+b);` Correto.
- `float a=1.0, byte b = a;`

## Conversão de tipo: Certo ou Errado?

- `byte b = 128;` Errado: 128 não pode ser representado em 1 byte (não pode ser convertido de `int` para `byte`).
- `byte b = -128;` Certo.
- `byte a, b = 1; byte c = a+b;` Errado: `byte` é estendido para `int` para realização de `a+b`.
- `byte a, b = 1; byte c = (byte)(a+b);` Correto.
- `float a=1.0, byte b = a;` Errado: em geral, perda de precisão possível.
- `float a=1.0, byte b = (byte)a;`

## Conversão de tipo: Certo ou Errado?

- `byte b = 128;` Errado: 128 não pode ser representado em 1 byte (não pode ser convertido de `int` para `byte`).
- `byte b = -128;` Certo.
- `byte a, b = 1; byte c = a+b;` Errado: `byte` é estendido para `int` para realização de `a+b`.
- `byte a, b = 1; byte c = (byte)(a+b);` Correto.
- `float a=1.0, byte b = a;` Errado: em geral, perda de precisão possível.
- `float a=1.0, byte b = (byte)a;` Correto.





# Entrada e Saída Textual: Primeiras Noções

- Operação de saída (ou escrita, ou gravação) de dados envia dados para um dispositivo de saída conectado ao computador (ex: tela de terminal, impressora, disco, fita etc.).
- Operação de entrada (ou leitura) de dados obtém dados de um dispositivo de entrada (ex: teclado, disco etc.).
- Dados armazenados como seqüências de bytes ou caracteres.
- Dados podem ser armazenados em **arquivos**, organizados em estrutura hierárquica de conjuntos de arquivos (chamados de **diretórios**) para facilitar localização e uso de arquivos.

# Entrada e Saída Textual

- **Texto:** seqüência de *linhas*, separadas por caractere terminador de linha ( '`\n`' ).
- Escrevendo caracteres em dispositivo de saída padrão:

*System.out.println(e)*

- ★ *out*: variável estática declarada na classe *System*.
- ★ tipo de *out*: *PrintStream*, contém definição do método *println*.

# Entrada e Saída Textual

- Lendo caracteres do dispositivo de entrada padrão:

```
int a = System.in.read();
```

armazena em *a* um único byte lido do dispositivo de entrada padrão.

- Valor entre 0 e 255 representa código do caractere lido.

# Entrada e saída textual

Leitura de um único caractere (representável em um byte), do dispositivo de entrada padrão, e escrita desse caractere lido no dispositivo de saída padrão:

```
import java.io.*;

class ESTextual
{ public static void main (String[] args) throws IOException
  { int a = System.in.read();
    System.out.println((char)a); }
}
```

`import java.io.*;` “importa” nomes definidos em “biblioteca” *java.io* (explicação detalhada mais à frente).

`throws IOException` também explicado mais adiante.

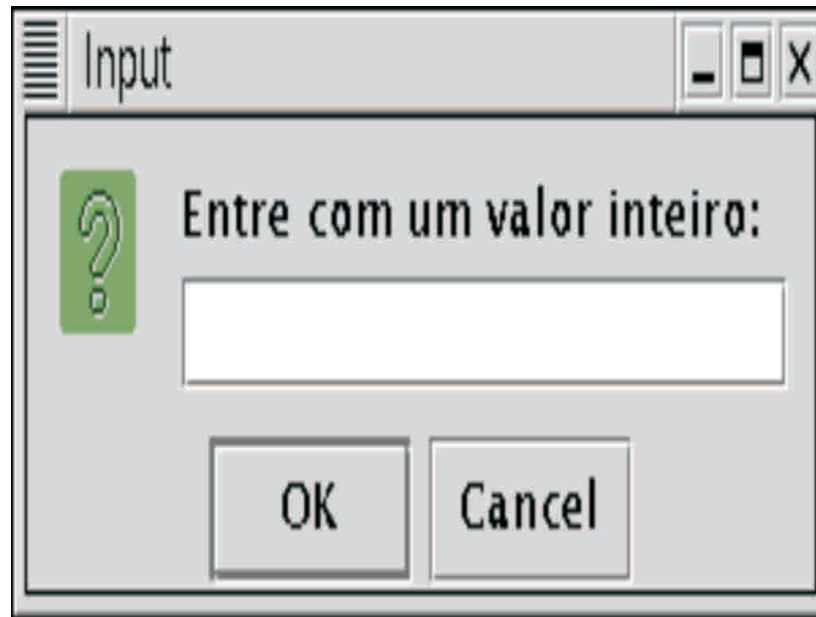


# Entrada e Saída em Janelas, Campos de Texto e Botões

- E/S em **componentes** de “interface gráfica” (ou “interface visual”)
- Modelo de **tratamento de eventos**: ações (ex: pressionar botão do *mouse*) originam chamadas a métodos.
- Bibliotecas **AWT** e **Swing** permitem tratamento de eventos em componentes de interface gráfica

## *showInputDialog*

```
String entrada = JOptionPane.showInputDialog  
    ("Entre com um valor inteiro:");
```



## *showInputDialog*

- *showInputDialog* cria janela e espera usuário digitar cadeia de caracteres no campo de texto.
- **Botão *OK* “clicado” ou tecla *Enter* pressionada:**  
⇒ cadeia digitada é retornada como objeto da classe *String* (e janela desaparece).
- **Botão *Cancel* “clicado”:** ⇒ valor `null` é retornado.

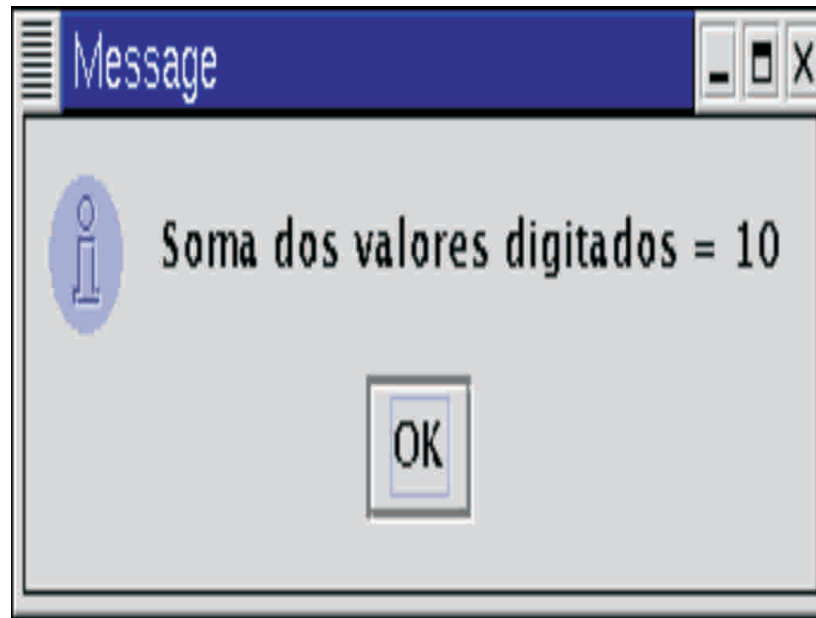
# *showMessageDialog*

```
import javax.swing.*;
class JanelasDeDialogo1
{ public static void main(String[] a)
  { String entrada = JOptionPane.showInputDialog
    ("Entre com um valor inteiro: ");
    int x = Integer.parseInt(entrada);
    entrada = JOptionPane.showInputDialog
    ("Entre com outro valor inteiro: ");
    int y = Integer.parseInt(entrada);

    JOptionPane.showMessageDialog
    (null, "Soma dos valores digitados = " + (x+y)); }
}
```



# *showMessageDialog*





## *showMessageDialog*

- 1<sup>o</sup> argumento especifica componente abaixo do qual a janela deve ser criada
- se `null` (isto é, nenhum componente) for especificado, a janela é criada no centro da tela
- 2<sup>o</sup> argumento especifica cadeia de caracteres a ser mostrada na janela

# Recursão e iteração

- Considere por exemplo que queremos definir a operação de multiplicação, em termos da operação mais simples de adição (apenas como exemplo ilustrativo de definições usando recursão e iteração, pois certamente essa operação está disponível na linguagem).
- A multiplicação de um número inteiro por outro inteiro maior ou igual a zero pode ser definida **por indução** como a seguir:

$$m \times 0 = 0$$

$$m \times n = m + (m \times (n - 1)) \quad \text{se } n > 0$$

# Iteração

- Mais informalmente, multiplicar  $m$  por  $n$  ( $n$  não-negativo) é somar  $m$ ,  $n$  vezes:

$$m \times n = \underbrace{m + \dots + m}_{n \text{ vezes}}$$

- Solução de problema que realiza operações repetidamente pode ser implementada (em linguagens imperativas) usando **comando de repetição** (também chamado de **comando iterativo** ou **comando de iteração**).



# Comandos de repetição

## Comando `while`

```
while (e) c
```



# Comandos de repetição

## Comando `while`

```
while (e) c
```

Avalia *e*; se false,



# Comandos de repetição

## Comando `while`

```
while (e) c
```

Avalia *e*; se false, termina;  
se true,

# Comandos de repetição

## Comando `while`

```
while (e) c
```

Avalia *e*; se `false`, termina;  
se `true`, executa *c* e repete o processo.





# Comandos de repetição

Comando for

```
for ( $c_0$ ;  $e$ ;  $c_1$ )  $c$ 
```

# Comandos de repetição

## Comando for

```
for ( $c_0$ ;  $e$ ;  $c_1$ )  $c$ 
```

Executa  $c_0$ .

Em seguida, faça o seguinte:

# Comandos de repetição

## Comando for

```
for ( $c_0$ ;  $e$ ;  $c_1$ )  $c$ 
```

Executa  $c_0$ .

Em seguida, faça o seguinte:

avalia  $e$ ;

se false,

# Comandos de repetição

## Comando for

```
for ( $c_0$ ;  $e$ ;  $c_1$ )  $c$ 
```

Executa  $c_0$ .

Em seguida, faça o seguinte:

avalia  $e$ ;

se false, termina;

se true,

# Comandos de repetição

## Comando for

```
for ( $c_0$ ;  $e$ ;  $c_1$ )  $c$ 
```

Executa  $c_0$ .

Em seguida, faça o seguinte:

avalia  $e$ ;

se false, termina;

se true, execute  $c$ , depois  $c_1$  e repita o processo.



# Comandos de repetição

## Comando do while

```
do c while (e)
```



# Comandos de repetição

## Comando do while

```
do c while (e)
```

Executa *c*

Avalia *e*; se false,



# Comandos de repetição

## Comando do while

```
do c while (e)
```

Executa *c*

Avalia *e*; se false, termina;  
se true,



# Comandos de repetição

## Comando do while

```
do c while (e)
```

Executa *c*

Avalia *e*; se false, termina;  
se true, repete o processo.

# Exemplo de iteração com comando `for`

```
static int mult (int m, int n)  
{ int r=0;  
  for (int i=1; i<=n; i++) r += m;  
  return r;  
}
```

# Iteração

- Exemplo a seguir segue passo a passo a execução de *mult(3,2)*
- São mostrados:
  - ★ **Comando a ser executado** ou **expressão a ser avaliada**
  - ★ **Resultado** (no caso de expressão)
  - ★ **Estado** (após execução do comando ou avaliação da expressão)

Detalhamento da execução de  $mult(3,2)$

Comando/ Expressão	Resultado (expressão)	Estado (após execução/avaliação)
$mult(3,2)$	...	$m \mapsto 3, n \mapsto 2$
<code>int r = 0</code>		$m \mapsto 3, n \mapsto 2, r \mapsto 0$
<code>int i = 1</code>		$m \mapsto 3, n \mapsto 2, r \mapsto 0, i \mapsto 1$
$i \leq n$	true	$m \mapsto 3, n \mapsto 2, r \mapsto 0, i \mapsto 1$
$r += m$	3	$m \mapsto 3, n \mapsto 2, r \mapsto 3, i \mapsto 1$
$i ++$	2	$m \mapsto 3, n \mapsto 2, r \mapsto 3, i \mapsto 2$
$i \leq n$	true	$m \mapsto 3, n \mapsto 2, r \mapsto 3, i \mapsto 2$
$r += m$	6	$m \mapsto 3, n \mapsto 2, r \mapsto 6, i \mapsto 2$
$i ++$	3	$m \mapsto 3, n \mapsto 2, r \mapsto 6, i \mapsto 3$
$i \leq n$	false	$m \mapsto 3, n \mapsto 2, r \mapsto 6, i \mapsto 3$
<code>for ...</code>		$m \mapsto 3, n \mapsto 2, r \mapsto 6$
<code>return r</code>		
$mult(3,2)$	6	



# Chamada de método

- Expressões — chamadas de **parâmetros reais** — são avaliadas, fornecendo valores dos **argumentos**.
- Argumentos são copiados para os **parâmetros** — também chamados **parâmetros formais** — do método.
- Corpo do método é executado.

# Chamada de método

- Chamada cria novas **variáveis locais**.
- Parâmetros formais são variáveis locais do método.
- Outras variáveis locais podem ser declaradas (ex: *r* em *mult*).
- Quando execução de uma chamada termina, execução retorna ao ponto da chamada.

# Comando `for`: terminologia

`for (c0; e; c1) c`

- *c<sub>0</sub>: comando de “inicialização”*

No caso em que é uma declaração, variável criada é comumente chamada de **contador de iterações**.

- *e: teste de terminação*
- *c<sub>1</sub>: comando de atualização*
- *c<sub>1</sub>: corpo*

# Recursão

Definição indutiva dá origem a implementação recursiva:

```
static int multr (int m, int n)  
{ if (n==0) return 0;  
  else return (m + multr(m, n-1)); }
```





# Recursão

- Cada chamada recursiva cria novas variáveis locais.
- Em chamadas recursivas, existem em geral várias variáveis locais de mesmo nome, mas somente as variáveis do último método chamado podem ser usadas (são acessíveis) diretamente.
- Quando execução de uma chamada recursiva termina, execução retorna ao método que fez a chamada.
- Assim, chamadas recursivas são executadas em **estrutura de pilha**.

# Recursão

- Exemplo a seguir ilustra a execução de *mult*(3,2)
- São mostrados, passo a passo:
  - ★ **Comando a ser executado** ou **expressão a ser avaliada**
  - ★ **Resultado** (no caso de expressão)
  - ★ **Estado** (após execução do comando ou avaliação da expressão)

$mult(3, 2)$	...	$m \mapsto 3$ $n \mapsto 2$		
$n == 0$	false	$m \mapsto 3$ $n \mapsto 2$		
return $m + mult(m, n - 1)$	...	$m \mapsto 3$ $n \mapsto 2$	$m \mapsto 3$ $n \mapsto 1$	
$n == 0$	false	$m \mapsto 3$ $n \mapsto 2$	$m \mapsto 3$ $n \mapsto 1$	
return $m + mult(m, n - 1)$	...	$m \mapsto 3$ $n \mapsto 2$	$m \mapsto 3$ $n \mapsto 1$	$m \mapsto 3$ $n \mapsto 0$
$n == 0$	true	$m \mapsto 3$ $n \mapsto 2$	$m \mapsto 3$ $n \mapsto 1$	$m \mapsto 3$ $n \mapsto 0$
return 0		$m \mapsto 3$ $n \mapsto 2$	$m \mapsto 3$ $n \mapsto 1$	
return $m + 0$		$m \mapsto 3$ $n \mapsto 2$		
return $m + 3$				
$mult(3, 2)$	6			



# Recursão

- Estrutura de pilha: último conjunto de variáveis (da pilha) são variáveis locais do último método chamado,
- penúltimo conjunto de variáveis são do penúltimo método chamado, e assim por diante.
- Espaço em memória de variáveis alocadas na pilha para um método é chamado de **registro de ativação** desse método.



# Valor inicial de variáveis locais

- Registro de ativação é alocado no início e desalocado no fim da execução de um método.
- **Variáveis locais a um método não são inicializadas automaticamente com valor *default*,**

# Valor inicial de variáveis locais

- Registro de ativação é alocado no início e desalocado no fim da execução de um método.
- **Variáveis locais a um método não são inicializadas automaticamente com valor *default*,**
- ao contrário de variáveis de objetos e de classes.

**Variável local tem que ser inicializada “em todos os caminhos até seu uso”**

**Por exemplo, programa a seguir contém um erro:**

```
import javax.swing.*;

class V
{ public static void main (String[] a)
  int x;
    boolean b = Boolean.valueOf(
        JOptionPane.showInputDialog("Digite \"true\" ou \"false\"")).booleanValue();
    if (b) x = Integer.parseInt(
        JOptionPane.showInputDialog("Digite um valor inteiro"));
    System.out.println(x); }

}
```

# Recursão simulando processo iterativo

```
static int multIter (int m, int n, int r)  
{ if (n == 0) return r;  
  else return multIter(m, n-1, r+m);  
}
```



# Recursão simulando processo iterativo

- Como na versão iterativa, a cada recursão valor de  $r$  (“acumulador”) é incrementado de  $m$ .
- Diferença:

versão recursiva	acumulador é nova variável a cada recursão
versão iterativa	acumulador é a mesma variável em cada iteração

# Exponenciação: implementações análogas

```
static int exp (int m, int n)  
{ int r=1;  
  for (int i=1; i<=n; i++) r*=m;  
  return r; }
```

```
static int expr (int m, int n)  
{ if (n==0) return 1;  
  else return (m * expr(m, n-1)); }
```

## *Math.pow* e *Math.exp*

- `public static double pow (double a, double b)` fornece como resultado valor de tipo `double` mais próximo de  $a^b$ .
- `public static double exp (double a)` fornece como resultado valor de tipo `double` mais próximo de  $e^a$  (sendo  $e$  a base dos logaritmos naturais).

# Eficiência

Definição indutiva da exponenciação:

$$\begin{aligned} m^0 &= 1 \\ m^n &= m \times m^{n-1} \quad \text{se } n > 0 \end{aligned}$$

Definição alternativa (também indutiva):

$$\begin{aligned} m^0 &= 1 \\ m^n &= (m^{n/2})^2 \quad \text{se } n \text{ é par} \\ m^n &= m \times m^{n-1} \quad \text{se } n \text{ é ímpar} \end{aligned}$$

# Eficiência

**Definição alternativa** dá origem a implementação **mais eficiente**:

```
static int exp2 (int m, int n)
{ if (n == 0) return 1;
  else if (n % 2 == 0) // n é par
    { int x = exp2(m, n/2);
      return x*x; }
  else return m * exp2(m, n-1);
}
```

# Eficiência

Diferença em eficiência é significativa:

- **Chamadas recursivas** na avaliação de  $exp2(m, n)$   
**dividem o valor de  $n$  por 2 a cada chamada**
- na avaliação de  $exp(m, n)$   
**valor de  $n$  é decrementado de 1 a cada iteração**
- assim como na avaliação de  $expr(m, n)$ , valor de  $n$  é decrementado de 1 a cada chamada recursiva

# Eficiência

- Exemplo: chamadas recursivas durante avaliação de  $\text{exp2}(2, 20)$ :

$\text{exp2}(2, 20)$	$\text{exp2}(2, 10)$	$\text{exp2}(2, 5)$	
$\text{exp2}(2, 4)$	$\text{exp2}(2, 2)$	$\text{exp2}(2, 1)$	$\text{exp2}(2, 0)$

- Quanto maior  $n$ , maior a diferença em eficiência.
- São realizadas da ordem de  $\log_2(n)$  chamadas recursivas durante avaliação de  $\text{exp2}(m, n)$  —  $n$  é em média dividido por 2 em chamadas recursivas**
- ao passo que avaliação de  $\text{exp}(m, n)$  requer  $n$  iterações.

# Fatorial recursivo

$$\begin{array}{ll} n! = 1 & \text{se } n = 0 \\ n! = n \times (n - 1)! & \text{em caso contrário} \end{array}$$

```
static int fatr (int n)  
{ if (n == 0) return 1;  
  else return n * fatr(n-1); }
```



# Fatorial iterativo

$$n! = n \times (n - 1) \times (n - 2) \times \dots 3 \times 2 \times 1$$

```
static int fat (int n)  
{ int f=1;  
  for (int i=1; i<=n; i++) f *= i;  
  return f; }
```

# Fatorial recursivo que se espelha no algoritmo iterativo

```
static int fatIter (int n, int i, int f)  
    // fatIter(n,1,1) = n!   i funciona como contador de recursões e  
    //                          f como acumulador (de resultados parciais)  
{ if (i > n) return f;  
  else return fatIter(n,i+1,f*i); }  
  
static int fatr1 (int n)  
{ return fatIter (n,1,1); } }
```

# Progressão aritmética de passo 1

## Implementação baseada em iteração

```
static int pa1 (int n)  
{ int s = 0;  
  for (int i=1; i<=n; i++) s += i;  
  return s; }
```

# Progressão aritmética de passo 1

## Implementação recursiva



```
static int par (int n)  
{ if (n==0) return 0;  
  else return n + par(n-1); }
```

# Progressão aritmética de passo 1

## Recursão espelhando algoritmo iterativo

```
static int pa1Iter (int n, int i, int s)  
{ if (i > n) return s;  
  else return pa1Iter(n, i+1, s+i); }
```

```
static int pa1rIter (int n)  
{ return pa1Iter(n,1,0); }
```



# Progressão aritmética

- Exemplos apenas ilustrativos: seriam implementações mal feitas na prática, pois ineficientes. . .
- Uma vez que . . .

# Progressão aritmética

- Exemplos apenas ilustrativos: seriam implementações mal feitas na prática, pois ineficientes. . .
- Uma vez que . . .

$$\sum_{i=1}^n i = \frac{n(n+1)}{2}$$



# Progressão geométrica: $\sum_{i=0}^n x^i$

- Implementação iterativa:



# Progressão geométrica: $\sum_{i=0}^n x^i$

- Implementação iterativa:

```
static int pg (int n, int x)
{ int s = 1, parc = x;
  for (int i=1; i<=n; i++)
    { s += parc; parc *= x; }
  return s; }
```

- *parc* usada para evitar cálculo de  $x^i$  a cada iteração.

# Progressão geométrica

Mostre analogia  $pa$ - $pg$ :

- Implemente  $pgr$  e  $pgrIter$ .
- Mostre que  $pg$ ,  $pgr$  e  $pgIter$  são ineficientes. . .

# Progressão geométrica

Mostre analogia  $pa$ - $pg$ :

- Implemente  $pgr$  e  $pgrIter$ .
- Mostre que  $pg$ ,  $pgr$  e  $pgIter$  são ineficientes. . . deduzindo fórmula para cálculo direto de  $pgs$ .
- Dica?

# Progressão geométrica

Mostre analogia *pa-pg*:

- Implemente *pgr* e *pgrIter*.
- Mostre que *pg*, *pgr* e *pgIter* são ineficientes. . . deduzindo fórmula para cálculo direto de *pgs*.
- Dica? multiplique  $s = \sum_{i=0}^n x^i$  por

# Progressão geométrica

Mostre analogia *pa-pg*:

- Implemente *pgr* e *pgrIter*.
- Mostre que *pg*, *pgr* e *pgIter* são ineficientes. . . deduzindo fórmula para cálculo direto de *pgs*.
- Dica? multiplique  $s = \sum_{i=0}^n x^i$  por  $-x$  e

# Progressão geométrica

Mostre analogia *pa-pg*:

- Implemente *pgr* e *pgrIter*.
- Mostre que *pg*, *pgr* e *pgIter* são ineficientes. . . deduzindo fórmula para cálculo direto de *pgs*.
- Dica? multiplique  $s = \sum_{i=0}^n x^i$  por  $-x$  e some a  $s$ .

# Implementação de somatórios

- Usar variável para armazenar soma.
- Decidir se parcela a ser somada vai ser obtida da parcela anterior ou do contador de iterações.
- No 1º caso, usar variável para armazenar valor calculado na parcela anterior (como *parc* em *pg*).
- Exemplo do 2º caso:  $\sum_{i=1}^n \frac{1}{i}$

# Implementação de somatórios

- Em vários casos, cálculo não usa a própria parcela anterior, mas valores usados no cálculo dessa parcela.
- Exemplo: cálculo aproximado do valor de  $\pi$ , usando:

$$\pi = 4 * \left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \dots\right)$$

- Precisamos guardar não valor mas **sinal** e **denominador** da parcela anterior.



# Implementação de somatórios

```
static float piAprox (int n)  
{ float s = 0.0f, denom = 1.0f; int sinal = 1;  
  for (int i=1; i<=n; i++)  
    { s += sinal/denom;  
      sinal = -sinal; denom += 2; }  
  return 4 * s; }
```

# Implementação de somatórios

$$e^x = 1 + (x^1/1!) + (x^2/2!) + \dots$$

```
static float eExp (float x, int n)  
{ float s = 1.0f; int i=1;  
  float numer = x; int denom = 1;  
  while (i<=n)  
    { s += numer/denom;  
      i++;  
      numer *= x; denom *= i; }  
  return s; }
```

# Não-terminação

Podem ocorrer programas cuja execução, em princípio, não-termina:

```
static int infinito()  
{ return infinito() + 1; }
```

```
static void cicloEterno()  
{ while (true) ; }
```

# Não-terminação

Existem programas cuja execução não termina apenas em alguns casos (para alguns valores de entrada). Exemplo:

```
static int fat (int n)  
{ int f=1;  
  for (int i=1; i!=n; i++) f *= i;  
  return f; }
```

# Seleção múltipla

(seleção de um dentre vários casos)

```
switch (e)
{ case e1:  c1;
  case e2:  c2;
  ...
  case en:  cn;
}
```

Expressão  $e$  avaliada.

Executado então primeiro comando  $c_i$  ( $1 \leq i \leq n$ ), caso exista, para o qual  $e = e_i$ .

Se não for executado comando de “saída anormal” (como `break`), são também executados comandos  $c_{i+1}, \dots, c_n$ , se existirem, nessa ordem.

# Comando break e caso default

- Execução de qualquer  $c_i$  pode ser finalizada (e geralmente deve ser) por meio do comando break.
- Se  $e \neq e_i$ , para todo  $1 \leq i \leq n$ , caso *default* pode ser usado.

Veja exemplo a seguir: ►

# Comando break com caso default

```
static double op (char c, double a, double b)
{ switch (c)
  { case '+': { return a + b; }
    case '*': { return a * b; }
    case '-': { return a - b; }
    case '/': { return a / b; }
    default: { System.out.println
              ("Caractere diferente de +,*,-,/");
              return 0.0; } // convenção
  }
}
```

## Caso default

- default pode ser usado no lugar de case  $e_i$ , para qualquer  $i = 1, \dots, n$ .
- Em geral usado depois do último caso.
- Se default não for especificado, execução de switch pode terminar sem que nenhum dos  $c_i$  seja executado (isso ocorre se resultado da avaliação de  $e \neq e_i$ , para  $i = 1, \dots, n$ ).



# Comando `switch`: crítica e restrições

- Necessidade de uso de `break` sempre que se deseja executar apenas uma alternativa em comando `switch` considerada ponto fraco de Java (herança de C).
- Expressão  $e$  deve ter tipo `int`, `short`, `byte` ou `char`, e deve ser compatível com tipo de  $e_1, \dots, e_n$ .
- Expressões  $e_1, \dots, e_n$  têm que ser valores constantes e distintos.

# break seguido de nome de rótulo

- Comandos `switch` e comandos de repetição podem ser precedidos de **rótulo**: nome seguido do caractere “:”.
- `break` pode ser seguido de nome de rótulo.
- Ao ser executado, tal comando causa terminação da execução do comando precedido pelo rótulo.

# Exemplo: break seguido de nome de rótulo

```
static String ident (String s)
/* Procura marca em s; se encontrar, retorna
 * marca encontrada; caso contrário, null.
 * Supõe: marca = cadeia de caracteres que:
 *     começa com caractere 'i'
 *     segue cadeia de caracteres não contendo 'i','j'
 *     termina com caractere 'j'
 */
```

```

{ int i = 0; String marca;
  while (i < s.length())
  { pesq: while (true)
    { while (s.charAt(i) != '<') { inci }
      marca = "<"; inci
      while (s.charAt(i) != '<' && s.charAt(i) != '>')
      { marca += Character.toString(s.charAt(i)); inci }
      if (s.charAt(i) == '>')
      { marca += ">"; return marca; }
      else break pesq;
    } return null; } }

```

Abreviação: `inci = i++; if (i >= s.length()) return null;`

# Classes e Objetos

- Em LOOs, classes são usadas para representar conjuntos de valores compostos.
  - ★ pontos no plano cartesiano (com coordenadas  $x$  e  $y$ )
  - ★ datas
  - ★ dados sobre produtos de uma loja
  - ★ etc. etc. etc.
- Em LOOs, classes são usadas em vez de produtos cartesianos ou registros (usados em outras linguagens) ►

# Classes e Objetos

- **Tipo:** Classe



```
class Ponto { int x, y;  
             Ponto(int a, int b) { x=a; y=b; } }
```

- **Valor:** Objeto

- ★ **Criação:** `new Ponto`

- ★ **Acesso:** `int f(Ponto p) { return p.x; }`

- **Usados tipicamente em LOOs (ex: Java)**



# **Estruturas semelhantes a classes e objetos em outras linguagens**

- Produtos e Tuplas
- Registros e Registros

# Produtos e Tuplas

- **Tipo:** Produto (cartesiano)

```
type Ponto = (Int, Int)
```

- **Valor:** Tupla
  - ★ **Criação:**  $(10, 20)$
  - ★ **Acesso:**  $f(x, y) = x$
- Usados tipicamente em linguagens funcionais (ex: Haskell)



# Registros

- **Tipo:** Registro

```
type Ponto = record { x:integer, y:integer };
```

- **Valor:** Registro

- ★ **Criação:** `var p:Ponto; p.x=10; p.y=20;` (em geral valor *default* criado em declaração de variável e modificado “seletivamente” — componente a componente)
- ★ **Acesso:** `function f(p:Ponto) begin f := p.x end;`

- **Usados tipicamente em linguagens imperativas (ex: Pascal; chamado de “estrutura” em C)**

# Exemplo de uso de classe como produto cartesiano ou registro

Classe *Equacao2* a seguir:

- Define tipo de objetos que representam equação de segundo grau  $ax^2 + bx + c$ .
- Construtor recebe argumentos que definem coeficientes da equação e define valores de variáveis do objeto criado: coeficientes, n<sup>o</sup> de raízes e valores das raízes (reais).

```
class Equacao2
```

```
{ float a, b, c, r1, r2;  
  int num_raizes;
```

```
    Equacao2 (float a, float b, float c)
```

```
    { if (a==0) { num_raizes = 1; r1 = r2 = c/b; }  
      else { float d = b*b - 4*a*c;
```

```
          if (d<0) num_raizes = 0;
```

```
          else { if (d==0) num_raizes = 1;
```

```
                else num_raizes = 2;
```

```
                r1 = (-b + d)/(2*a);
```

```
                r2 = (-b - d)/(2*a); } }
```

```
    }
```

```
}
```

## Criação e uso de objetos

```
class ExemploEquacao2
{ static void main(String[] args)
  { Equacao2 eq = new Equacao2(1.0f,2.0f,-8.0f);
    if (eq.num_raizes == 0)
      System.out.println("Equação não tem raízes reais");
    else if (eq.num_raizes == 1)
      System.out.println("A raiz da equação é " + eq.r1);
    else
      System.out.println("As raízes da equação são "
                          + eq.r1 + " e " + eq.r2); }
}
```

# Subclasse

- Classe pode ser definida como **subclasse** de outra classe
- Indicação feita explicitamente na definição da subclasse

```
class B extends A  
// indica que B é subclasse de A  
{ ... }  
...  
class A { ... }
```



# Subclasses e herança

- Subclasses visam facilitar aproveitamento de classes já desenvolvidas na definição de novas classes
- Subclasse **herda** métodos e variáveis das superclasses

Veja exemplo a seguir



```
class Ponto3D extends Ponto
{ int z;

  Ponto3D (int a, int b, int c)
  { super(a,b); z = c; }

  void move (int a, int b, int c)
  { super.move(a, b); z += c; }

  // continua . . .
```

# Subclasse e herança

- Objetos da classe *Ponto3D* representam pontos em espaço tridimensional (isto é, pontos que podem ser representados com 3 coordenadas cartesianas).
- Classe *Ponto3D* é subclasse de *Ponto* — objetos da classe *Ponto* representam pontos do plano (espaço bidimensional, isto é, pontos que podem ser representados com 2 coordenadas cartesianas).
- *Ponto3D* herda variáveis ( $x$  e  $y$ ) e métodos de *Ponto*.





# super

- `super` usada para executar corpo do construtor da superclasse: `super(a, b)`
- `super` usada para chamar método da superclasse: `super.move(a, b)`

## *Ponto3D (continuação)*

```
class Ponto3D
{ ...
  double distancia (Ponto3D p)
  { int dx = x - p.x;
    int dy = y - p.y;
    int dz = z - p.z;
    return Math.sqrt(dx*dx + dy*dy + dz*dz); }
}
```

# Subclasses: relação

- Se  $A$  é subclasse de  $B$ , então  $B$  é **superclasse** de  $A$  (relação inversa)
- **Relações** de subclasse e superclasse são **transitivas**
- **Herança simples**: podem existir várias superclasses de uma classe, mas apenas uma superclasse **direta**
- Podem existir várias subclasses diretas de uma classe



# Subtipagem

- Subclasse é **subtipo**: objeto de subclasse  $C$  pode ser usado em qualquer lugar em que objeto de superclasse de  $C$  pode.
- Subtipagem estudada mais adiante (Capítulo 10).

# Associação dinâmica

- Em chamada  $e.m(\dots)$ , onde:  $m$  definido em  $C$  e redefinido em subclasse de  $C$  com mesma assinatura

**método chamado depende do tipo do objeto denotado por  $e$**  (método de  $C$  se  $\text{tipo} = C$  e da subclasse se  $\text{tipo} = \text{subclasse}$ ).
- Em geral, tipo do objeto denotado por  $e$  só pode ser determinado dinamicamente (por causa da subtipagem).
- Associação dinâmica também abordada no Capítulo 10.



# Unidades de Compilação

- *Unidade de compilação*: parte de programa que pode ser compilada separadamente.
- Programas médios e grandes são geralmente divididos em várias unidades de compilação.



# Visibilidade de nomes

- Existe em geral mecanismo em linguagens para **controle da visibilidade** dos nomes definidos.
- Mecanismo permite- **encapsulamento** (restrição da visibilidade) e **exportação** e **importação** de nomes (extensão da visibilidade).

# Gerenciamento da visibilidade de nomes

Nomes exportados e importados definem **interface** de uma unidade de compilação ou de uma construção sintática da linguagem

Decomposição adequada de um programa em partes, com interfaces pequenas e propósitos bem definidos, é fundamental para facilitar o desenvolvimento de programas, e eventuais modificações do mesmo.





# Pacotes

- Construção sintática para divisão de programas em partes, **organizadas hierarquicamente**, com controle de visibilidade de nomes.
- Pacote consiste simplesmente de seqüência de definições (em geral definições de classes)
- Nomes exportados são definidos com atributo `public`.

# Pacotes: exemplo

```
package P;  
  
public class C  
{ public static int x = 1; }
```

`package P;` indica início de declaração de pacote de nome *P*.

# Cláusula de importação

```
// ausência de package nome;  
// corresponde a definição de pacote sem nome
```

```
import P.C;
```

```
class D  
{ static int y = C.x; }
```

# Nomes qualificados

```
// Cláusula de importação evita nomes qualificados  
// e vice-versa
```

```
//  $P.C.x$  em vez de import P.C; ... C.x
```

```
class  $D$   
{ static int  $y = P.C.x$ ; }
```

# Nomes qualificados

- Nomes qualificados evitam conflito de nomes iguais de classes definidas em pacotes diferentes.
- `import P.*;` indica importação de todos os nomes públicos de *P*.
- Pacotes podem ser organizados **hierarquicamente**. Ex: pacote *java* contém pacotes *java.io*, *java.lang* etc.

# Pacotes X unidades de compilação

- Pacote pode ser dividido em várias unidades de compilação.
- Estrutura hierárquica de pacotes e sub-pacotes segue estrutura hierárquica de diretórios.
- Compilador procura, para cada pacote, diretório de mesmo nome — em diretório predefinido (no caso de pacotes do sistema), diretório corrente, ou especificado pela variável de ambiente CLASSPATH.

# Exceções

- **Exceções:** Erro durante a execução para o qual pode existir “*tratamento*”.
- **Exemplo:** divisão por zero, falta de espaço suficiente para alocação de memória em dispositivo de armazenamento de dados, chamada de método na qual valor da expressão que denota objeto alvo é `null`, etc.



# Exceções: Objetivos

- Permitir que execução de programa continue mesmo depois da ocorrência de erro.
- Permitir estrutura mais adequada para programas, separando código que define comportamento “normal” (esperado) do código que realiza tratamento de exceções.



# Exceções em Java

Podem ser causadas por:

- Erro detectado pelo sistema de suporte a execução.
- Comando `throw` incluído pelo programador.

Em qualquer caso: exceção em Java é objeto que representa a situação de erro ocorrida, criado automaticamente no instante em que ocorre esse erro.



# Tratador de Exceção

- Comando ou método em que uma exceção pode ocorrer pode (ou tem que) especificar um **tratador**.
- **Tratador:** bloco de comandos (possivelmente parametrizado) a ser executado se e quando a exceção ocorrer.

# Tratamento de Exceções

Para especificar tratador, comando que pode provocar ocorrência da exceção deve ser colocado internamente a um comando try:

```
int x = 0;
try { System.out. print(x/0); }
catch (ArithmeticException e)
    { System.out.println
      ("Exemplo de tratamento de exceção"); }
```

# Classes *Exception* e *Throwable*

- Exceções são objetos de subclasses da classe *Throwable*
- *Exception*, subclasse de *Throwable*, contém:
  - ★ Construtor com parâmetro de tipo *String*, para receber informações relativas à exceção ocorrida.
  - ★ Método *getMessage*, que recupera essa cadeia de caracteres.

# Qual Tratador?

Exceção  $E$  ocorre durante execução de comando  $c$  em método (ou construtor)  $m$ ; então:

- Execução é transferida para tratador mais interno, no corpo de  $m$ , que engloba  $c$ , caso exista esse tratador;
- Caso contrário,  $E$  é **propagada** para (isto é, considerada como causada no) ponto de chamada a  $m$ .

# Exceções da classe *RuntimeException*

Considere método ou construtor  $p$ :

**Exceções podem precisar ou não ser especificadas no cabeçalho de  $p$  sempre que puderem ocorrer em algum comando e não existir tratador correspondente em  $p$ .**

- No 1<sup>o</sup> caso, cabeçalho de  $p$  deve conter cláusula `throws  $E$`  (que indica que chamada a  $p$  pode provocar ocorrência de  $E$ ).
- No 2<sup>o</sup> caso, devem ser exceções de tipo-classe *RunTimeException*.
- Objetivos: conveniência do programador e clareza dos programas.

# Exceções da classe *RuntimeException*

Nenhum tratador para *NumberFormatException* — subclasse de *RunTimeException* — que pode ser causada por *parseInt*:

```
class NenhumTratador_RunTimeException
{ public static void main (String[] a)
  { System.out.print (Integer.parseInt(a[0])); }
}
```

# Exemplo de Tratamento de Exceções

Programa a seguir lê dois algarismos e imprime maior inteiro que se pode escrever com esses dois algarismos.

Método *charPraNum* recebe como argumento um caractere e retorna valor inteiro correspondente, caso esse caractere seja algarismo (0 a 9). Caso contrário, exceção *AlgarismoEsperado* é gerada — usando comando `throw` — e propagada por esse método, por meio do uso da cláusula `throws` no cabeçalho do método.



# Exemplo: Gerando e Propagando Exceções

```
static int charPraNum (char c)  
    throws AlgarismoEsperado  
{ if (c >= '0' && c <= '9') return (c - '0');  
  else throw new AlgarismoEsperado(c); }
```


## Exemplo: subclasse de *Exception*

```
import javax.swing.*;  
  
class AlgarismoEsperado extends Exception  
{ AlgarismoEsperado(char c)  
    { super ("Caractere digitado: " + c + "\n" +  
            "Era esperado um algarismo."); }  
}
```



# Exemplo: código para exceções separado

Tratamento de exceções permite que trecho do programa que trata da execução normal (no caso, a entrada de dados corretos) fique separado do trecho de tratamento de exceções (contido na cláusula `catch`).



```

static void dialog ()
{ try
    { char c1 = (JOptionPane.showInputDialog ("Digite ...: ")) . charAt(0);
      int  d1 = charPraNum(c1);
      char c2 = (JOptionPane.showInputDialog ("Digite ...: ")) . charAt(0);
      int  d2 = charPraNum(c2);
      int  v   = maiorInt2Alg(d1, d2);

      JOptionPane.showMessageDialog (null, "Maior inteiro formado com " + c1 +
        " e " + c2 + " = " + v, "Resultado", JOptionPane.INFORMATION_MESSAGE); }
    catch (AlgarismoEsperado exc)
        { JOptionPane.showMessageDialog(null, exc.getMessage(),
          "Erro", JOptionPane.ERROR_MESSAGE); dialog(); } }

```

# Cláusula finally

- Comando em cláusula finally (de comando try) é sempre executado, ocorrendo ou não exceção.
- Se exceção  $E$  for causada e execução entrar em corpo da cláusula, pode ocorrer que essa execução termine:
  - ★ normalmente (após atingir fim da execução do último comando): nesse caso exceção  $E$  é propagada;
  - ★ anormalmente: nesse caso  $E$  é descartada. Pode ocorrer que nova exceção ocorra na cláusula finally: nesse caso ela é sinalizada (sobrepondo-se a  $E$ ).

```
class Exemplo_finally
{ public static void m1 () throws Ex2
  { try { throw new Ex1(); } finally {throw new Ex2();} }
  public static void m2 () throws Ex1
  { try { throw new Ex1(); } finally {return;} }

  public static void m3 () throws Ex3
  { try { throw new Ex1();} catch (Ex1 e) {throw new Ex2();}
    finally {throw new Ex3();} }
  public static void m4 () throws Ex2
  { try { throw new Ex1(); } catch (Ex1 e) {throw new Ex2();}
    finally { ; /* nada */ } }

  public static void main (String[] a)
  { try {m1();} catch(Ex2 e) { System.out.print('a');}
    try {m2();} catch(Ex1 e) { System.out.print('b');}
    try {m3();} catch(Ex3 e) { System.out.print('c');}
    try {m4();} catch(Ex2 e) { System.out.print('d');}} }
```

# Utilidade da cláusula finally

```
try { ... /* recursos alocados */  
    ... /* usados */ ...  
catch (...) { ... }  
catch (...) { ... }  
finally { ...  
    /* e liberados (com ou sem exceções) */  
    ... }
```

# Sobrecarga em Java

Dizemos que existe **sobrecarga** quando métodos de mesmo nome são visíveis em um mesmo escopo (chamada difere por tipo de parâmetros ou resultado).

Em Java, métodos sobrecarregados devem ter pelo menos um *parâmetro* de tipo diferente:

```
class ErroSobrecarga
{ int read() { ... }
  boolean read() { ... } }
```



# Classe *Console*

```
static boolean  readBoolean()  
static double   readByte()  
static short    readShort()  
static int       readInt()  
static long      readLong()  
static float     readFloat()  
static double    readDouble()  
static boolean   readBoolean()  
static String    readString()
```

# Classe *BufferedReader*

- Contém fonte de caracteres, de onde os caracteres são de fato lidos.
- Métodos (ex: *readLine*) usam áreas de armazenamento temporário ( “buffers” ) para melhorar o desempenho das operações de entrada de dados.
- Construtor recebe como parâmetro objeto (fonte de caracteres) da classe *InputStreamReader*.

# Classe *Console*: implementação

```
import java.io.*;
public class Console
{final static BufferedReader console =
    new BufferedReader(new InputStreamReader(System.in));
  final static PrintStream terminal = System.out;

  . . .
```

```
public static byte readByte()
{try { return Byte.parseByte(console.readLine());
  catch(IOException e)
    { terminal.println("IOException:tente de novo.");
      return readByte(); }
  catch(NumberFormatException e)
    { terminal.println("NumberFormatException \n" +
      "Valor entre -128 e 127 esperado: digite novamente");
      return readByte(); } }
```

```
public static String readString ()
{ try    { return console.readLine(); }
  catch (IOException e)
      { terminal.println( "IOException: tente de novo.");
        return readString(); }
}
```

# Classe *StreamTokenizer*

- *StreamTokenizer* provê suporte a *análise léxica*.
- **Análise léxica:** separação da entrada (contendo uma cadeia de caracteres) em *elementos léxicos* (“*tokens*”), descartando *caracteres delimitadores* (como espaços, caracteres de tabulação e de mudança de linha).
- Classe com comportamento típico de POO (métodos mudam *estado*, que pode então ser “consultado”).

## Método *nextToken*

Método *nextToken* de *StreamTokenizer* lê “próximo” elemento léxico de uma fonte (“*stream*”) de caracteres (objeto do tipo *StreamTokenizer*) e modifica variáveis desse objeto, de acordo com elemento léxico lido.

Se for lido	Modifica	de Tipo
valor numérico	variável <i>nval</i> (e <i>ttype</i> )	<code>double</code>
identificador	variável <i>sval</i> (e <i>ttype</i> )	<code>String</code>
caso contrário	variável <i>ttype</i>	<code>int</code>

# Variável *ttype*

*nextToken* retorna e armazena em *ttype* tipo do elemento léxico lido:

- *TT\_NUMBER* — indica que foi lido valor numérico;
- *TT\_WORD* — indica que foi lido um identificador;
- *TT\_EOF* — indica fim da entrada de caracteres;
- *TT\_EOL* — indica fim de linha;
- nenhuma das opções acima — tipo do elemento léxico é o valor da representação do caractere. Ex: se '\*' for lido, valor retornado e armazenado em *ttype* é igual ao código Unicode de '\*'.



# Especificando identificadores

- *identificador* caracterizado por: *ttype* = *TT\_WORD*
- Usualmente, iniciado com letra (ou outro caractere que indica início de um nome) e seguido por letras e dígitos
- *wordChars*, definido em *StreamTokenizer*, especifica caracteres de identificadores:

```
public void wordChars (int c1, int c2)
```

- caracteres com código na faixa de *c1* a *c2* passam a ser considerados como componentes de identificadores

# Especificando delimitadores

- `public void whiteSpaceChars(int c1, int c2)`
  - Define quais caracteres são delimitadores (separam identificadores), de maneira análoga a *wordChars*
- `public void eolIsSignificant(boolean b)`
  - Habilita ou desabilita atribuição de TT\_EOL a *ttype*
- `public void resetSyntax()`
  - Especifica que nenhum caractere é reconhecido como identificador ou delimitador.

# Usando *StreamTokenizer*: exemplo

Programa a seguir usa *StreamTokenizer* para contar número de caracteres, palavras e linhas na entrada padrão.

```
import java.io.*;

class ContaPalavras
{ public static int palavras    = 0;
  public static int linhas     = 0;
  public static int caracteres = 0;
  ...
}
```

```
public static void conta_palavras (InputStreamReader r)
                                throws IOException
{ StreamTokenizer st = new StreamTokenizer(r);
  st.resetSyntax(); st.eolIsSignificant(true);
  st.wordChars(33, 255); st.whitespaceChars(0, ' ');
  while (st.nextToken() != st.TT_EOF)
    switch (st.ttype) {
      case st.TT_EOL: linhas++; break;
      case st.TT_WORD: palavras++;
        caracteres += st.sval.length(); break;
      default: caracteres += st.sval.length(); break;
    } }
```

```
public static void main (String[] args)
{ try { conta_palavras(new InputStreamReader(System.in));
      System.out.println("\n" +
                          "linhas = "      + linhas      + "\n" +
                          "palavras = "    + palavras    + "\n" +
                          "caracteres = " + caracteres); }
  catch (IOException e) { };
}
```



# Arranjo

Estrutura de dados muito usada

— por questão de eficiência —

para representação de tabelas ou de seqüências de valores

# Arranjo é estrutura de dados:

- **composta**: por dados (componentes) “mais simples”
- **homogênea**: componentes têm sempre mesmo tipo (ao contrário por exemplo de classes)
- de tamanho fixo: tem (em qualquer instante da execução) **número limitado de componentes**
- **“eficiente”**: armazenamento em posições contíguas permite **tempo de acesso igual aos componentes**
- **acesso via indexação**: índice para cada componente

# Arranjo $\cong$ memória

Arranjo espelha funcionamento de  
memória de computador

Memória pode ser vista como arranjo  
no qual índices são endereços de posições da memória  
(chamadas de *palavras*)



# Variável e Objeto de tipo Arranjo

declaração  
de  
variável  
 $\underbrace{\text{int}[] \text{ v}}$  = criação  
de  
objeto  
 $\underbrace{\text{new int}[n];}$

0	0	...	0	...	0
0	1		$i$		$n-1$

$v[i]$  representa  $i$ -ésima variável de  $v$  ( $i = 0, \dots, n - 1$ )

# Arranjo: Indexação

$v$  arranjo de tamanho  $n$        $\left\{ \begin{array}{l} v[i] \text{ variável} \\ (i\text{-ésimo componente de } v) \end{array} \right.$   
 $i$  tem valor entre 0 e  $n-1$

$i$  fora do intervalo 0 a  $n-1 \Rightarrow$  exceção

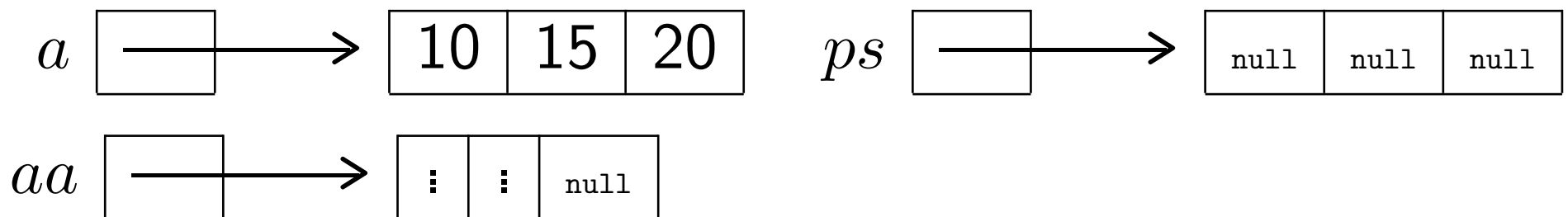
*IndexOutOfBoundsException*

# Tipo Arranjo

- Dado tipo  $T$  qualquer,  $T[]$  representa em Java tipo arranjo com componentes do tipo  $T$
- Exemplos:
  - `int[]` tipo de arranjos de inteiros
  - `int[][]` tipo de arranjos de arranjos de inteiros
- tamanho ( $n^o$  de elementos) não faz parte do tipo arranjo em Java: tamanho definido para valores do tipo arranjo (não necessariamente para variáveis e expressões)

# Criação de arranjos

```
int[] a      = { 10, 15, 20 };  
char[][] aa  = { { 'a', 'b' }, { 'c' }, null };  
Ponto[] ps  = new Ponto[4];
```



# Arranjos de tamanhos diferentes como elementos de arranjos

```
int [][] a;  
a = new int [2] [];  
...  
a[0] = new int [10];  
...  
a[1] = new int [40];  
...
```

# Exemplo: maior valor em arranjo de inteiros

```
int[] a;  
...  
int maior = a[0];  
for (int i=1; i<a.length; i++)  
    if (a[i] > maior) maior = a[i];
```

# Classe *Arrays*

*fill(a, v)* armazena valor denotado por *v* em todos os componentes do arranjo *a*;

*equals(a1, a2)* compara se valores em *a1* e *a2* são iguais

*sort(a)* ordena valores em *a* (ordem não-descendente)

*binarySearch(a, v)* pesquisa por *v* em *a*, supondo *a* ordenado (retorna `true` se encontrar e `false` caso contrário)



# Pesquisa em Arranjos

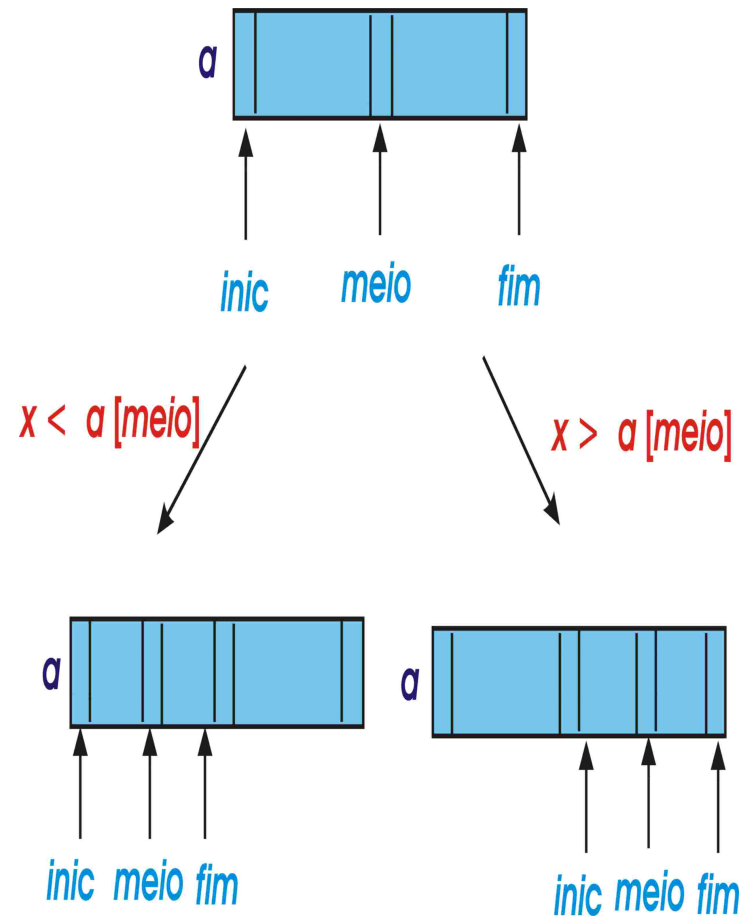
- **Pesquisa** por valor em seqüência de valores é problema comum em diversas aplicações
- Algoritmo mais simples: **pesquisa seqüencial**
- Percorre seqüência de valores, do primeiro ao último valor, testando igualdade ao valor desejado



# Pesquisa Seqüencial em Arranjo

```
static int pesqSeq (int[] a, int x)  
{ for (int i = 0; i < a.length; i++)  
    if (a[i] == x) return i;  
    return -1; // -1 indica valor não encontrado  
}
```

# Pesquisa Binária



# Pesquisa Binária: implementação recursiva

```
static int pesqBinr (int[] a, int x)
{ return pesqBRec (a, x, 0, a.length-1); }

static int pesqBRec (int[] a,int x,int inicio,int fim)
{ if (inicio > fim) return -1; // -1 = insucesso

  int meio = (inicio+fim)/2, v = a[meio];
  if      (x == v)    return meio;
  else if (x < v)      return pesqBRec(a,x,inicio,meio-1);
  else /* (x > v) */  return pesqBRec(a,x,meio+1,fim); }
```

# Pesquisa Binária: implementação iterativa

```
static int pesqBin (int[] a, int x)
{ int v, meio, inicio=0, fim=a.length-1;

  while (inicio <= fim)
  { meio = (inicio+fim)/2;
    v = a[meio];
    if      (x == v)      return meio;
    else if (x < v)      fim = meio-1;
    else /* (x > v) */ inicio = meio+1; }
  return -1; // -1 indica pesquisa sem sucesso
}
```

# Ordenação

- Dada seqüência de  $n$  valores, encontrar permutação  $v_1, \dots, v_n$  desses valores tal que  $v_i \leq v_{i+1}$ , para todo  $i$  entre 1 e  $n - 1$ .
- Problema clássico em computação
- Diversos algoritmos existentes (seleção, “bolha”, “quicksort”, “mergesort” etc.)

# Ordenação por seleção

- Determina posição do menor elemento e troca valor contido nessa posição com o contido na 1ª posição
- Em seguida, o 2º menor elemento é selecionado e trocado com valor contido na 2ª posição
- E assim sucessivamente.
- Ou seja: para arranjo de tamanho  $n$ ,  $i$ -ésima iteração, para  $i = 0, \dots, n - 2$ , determina menor valor dentre os contidos nas posições de índice  $i$  a  $n - 1$ , e troca valores contidos na posição  $i$  e na posição desse  $i$ -ésimo menor elemento.

# Ordenação por seleção

```
static void ordena(String[] a)
{ for (int i=0; i<a.length-1; i++)
    troca(i, indMenor(i)); }
```

## *indMenor e troca*

```
private int indMenor(int i)
int m = i, j; String menor = a[m];
    for (j = i+1; j<a.length; j++)
        if (a[j].compareTo(menor) < 0)
            { m = j; menor = a[m]; }
return m; }
```



## *indMenor e troca*

```
private int indMenor(int i)  
int m = i, j; String menor = a[m];  
    for (j = i+1; j<a.length; j++)  
        if (a[j].compareTo(menor) < 0)  
            { m = j; menor = a[m]; }  
    return m; }
```

```
private void troca (int i, int j)  
{ String t = a[i]; a[i] = a[j]; a[j] = t; }
```

## *quicksort*

- Escolhe elemento (*pivô*) e separa elementos em duas partes: aqueles com valor maior e aqueles com valor menor ou igual ao pivô.
- Aplica mesmo procedimento a cada parte, se ela tem mais de um elemento.

Como pesquisa binária, baseado em técnica muito usada na construção de algoritmos: *dividir para conquistar* (*“divide and conquer”*)

```
void ordena()  
{ quicksort(0, a.length-1); }  
  
private void quicksort (int i, int j)  
{ if (i < j)  
    { String pivo = a[(i+j)/2];  
      int m = divide(pivo, i, j);  
      quicksort(i, m); quicksort(m+1, j);  
    }  
}
```

```
private int divide (String pivo, int i0, int j0)
{
    int i = i0-1, j = j0+1;
    do {
        do { i++; } while (a[i].compareTo(pivo)<0);
        do { j--; } while (a[j].compareTo(pivo)>0);
        if (i < j) troca(i,j);
    } while (i < j);
    return j;
}
```



# Modularização em LOOs

- Estrutura de programas definida por **interfaces** entre classes
- Interface entre classes  $A$  e  $B$  especificada por conjunto de nomes de  $A$  que podem ser usados em  $B$  e vice-versa
- Modificação/extensão mais fácil de programas:
  - ★ divisão em classes com interfaces claras e bem definidas
  - ★ e com **encapsulamento** de dados: mudança na implementação não acarreta mudança em outras classes

# Facilidade de modificação

Mecanismos para facilitar reuso com possibilidade de extensão e especialização:

- **Herança:** aproveitamento de definições, com possibilidade de redefinição
- **Subtipagem:** definições existentes podem operar com instâncias das novas definições
- **Associação dinâmica:** redefinições automaticamente usadas se objeto é instância da redefinição; permite que análise de casos se restrinja a criação de objetos.

## Controle de visibilidade de nomes

<b>Acesso permitido</b>	<code>private</code>	<b>sem atributo</b>	<code>protected</code>	<code>public</code>
mesma classe	Sim	Sim	Sim	Sim
mesmo pacote, outra classe	Não	Sim	Sim	Sim
outro pacote, subclasse	Não	Não	Sim	Sim
outro pacote, fora de subclasse	Não	Não	Não	Sim

# Controle de redefinição em subclasse

- Atributo `final` em definição de método indica que método não pode ser redefinido em subclasse.
- Garante que comportamento de um método não vai ser modificado em nenhuma subclasse desse método. Ex: método para verificação da validade de senha secreta.
- Atributo `final` em declaração de classe especifica que nenhuma subclasse dessa classe pode ser definida.





# Classes e métodos abstratos

- Classe **abstrata** — declarada com atributo `abstract` — contém um ou mais métodos abstratos
- Método abstrato — declarado com atributo `abstract` — não é implementado: apenas assinatura especificada
- Implementação de método abstrato deve ser feita em subclasse não abstrata
- Classe abstrata provê comportamento de alguns métodos e especifica apenas interface de outros

# Classes e métodos abstratos: Exemplo

```
abstract class Benchmark
{
    abstract void benchmark();
    public long repita(int n)
    {
        long inicio = System.currentTimeMillis();
        for (int i=0; i<n; i++) benchmark();
        return (System.currentTimeMillis() - inicio);
    }
}
```

# Interfaces

- **Interface** contém assinaturas de métodos e constantes.
- Classes podem implementar uma ou mais interfaces, definindo implementações para métodos cujas assinaturas foram especificadas nessas interfaces.

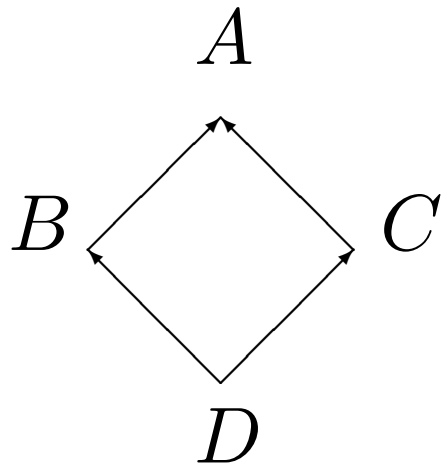
```
interface Ponto
{ float x(); float y();
  void move(float dx, float dy);
  float distancia(Ponto p); }
```

```
public class PCart implements Ponto
{ private float x, y;
  public float x() { return x; }
  public float y() { return y; }
  public PCart(float a, float b) {x=a; y=b; }
  public void move (float dx, float dy)
  { x+=dx; y+=dy; }
  public float distância (Ponto p)
  { float dx = x - p.x(), dy = y - p.y();
    return (float) Math.sqrt(dx*dx+dy*dy); }
}
```

# Herança de interfaces

```
interface CoresBasicas
{ int vermelho=1, verde=2, azul=3; }
interface CoresDoArcoIris extends CoresBasicas
{ int amarelo=4, laranja=5, anil=6, violeta=7; }
interface Cores extends CoresDoArcoIris
{ int carmim=8, ocre=9, branco=0; preto=10; }
interface PontoColorido extends Cores, Ponto
{ Cores cor(); void mudaCor(Cores cor); }
```

# Porque Java não provê suporte a herança múltipla de classes



Variável declarada/usada em classe herdada por caminhos distintos

Considere  $a$  tal variável: uso de expressão  $e.a$  em  $D$  — onde  $e$  tem tipo  $A$  — ambíguo.

# Ambigüidade no uso de constantes de mesmo nome em interfaces distintas

```
interface A { int x = 1; }
```

```
interface B { float x = 2.0f; }
```

```
class C implements A, B  
{ public static void main(String[] a)  
  { System.out.println(x); }  
}
```

# Uso de constantes de mesmo nome em interfaces distintas

```
interface A { int x = 1; }
```

```
interface B { float x = 2.0f; }
```

```
class C implements A, B  
{ public static void main(String[] a)  
  { System.out.println(A.x + B.x); }  
}
```



# Subtipagem

## Regra de subtipagem:

Se expressão tem tipo  $T$ ,  
tem também qualquer supertipo de  $T$

Implica que em qualquer ponto de um programa em que é permitido usar expressão de tipo  $T$  podemos usar expressão de subtipo de  $T$

(pois expressão de subtipo de  $T$  pode ser considerada como expressão de tipo  $T$ )

## Subtipagem: Exemplo

```
class A
{ int a = 1; void m1 { a++; } }

class B extends A
{ int b = 2;
  void m2()
  { A a = new A(); B b = new B();
    a = b; a.m1(); }
}
```

# Associação dinâmica

- Método a ser executado — em chamada “ $e.m()$ ” — determinado (dinamicamente) conforme tipo do objeto denotado por  $e$
- Associação dinâmica evita *análises de casos*: testes e chamada para cada tipo possível para o objeto
- Conseqüência: programa mais claro, conciso, e mais fácil de ser modificado

```
class A
{ int a = 1;
  void m() { System.out.println(a); } }
class B extends A
{ int a = 2;
  void m() { System.out.println(a); } }
class Main
{ public static void main(String[] args)
  { A a = new A(); B b = new B();
    a.m(); a = b; a.m(); } }
```



# Invariância

- **Invariância** na redefinição de um método em uma subclasse: método da subclasse e método de mesmo nome na superclasse têm a mesma assinatura.
- Em Java só é realizada associação dinâmica de nomes a métodos no caso em que há invariância na redefinição.

```
class A
{ int a=1;
  void m() { System.out.println(a); } }
class B extends A
{ int a=2;
  void m(int p) { System.out.println(a+p); } }
class Main
{ public static void main(String[] args)
  { A a = new A(); B b = new B();
    a.m(); a = b;
    a.m(1); // erro!  } }
```

# Covariância e Contravariância

- **Covariância:** Tipo de parâmetro ou do resultado de redefinição de método em subclasse é *subtipo* do tipo na definição original (e redefinição tem o mesmo número de parâmetros).
- **Contravariância:** Tipo de parâmetro ou do resultado de redefinição de método em subclasse é *supertipo* do tipo original.
- Co e contra-variância não provocam associação dinâmica em Java.

# Covariância: considere *motorista.m(v)*

```
class Veiculo
{ String placa; ... }
class Caminhao extends Veiculo
{ int cargaMaxima; ... }
class Motorista
{ void m(Veiculo v) { ...v.placa ...} }
class MotoristaDeCaminhao extends Motorista
{ void m(Caminhao c) { ...c.cargaMaxima ...} }
```



## Associação dinâmica: Exemplo

```
import java.util.Arrays;
class Exemplo_compareTo
{ public static void main (String[] a)
  { Candidato[] cands =
    new Candidato[Integer.parseInt(a[0])];
    /* Cria objetos da classe Candidato e
       armazena-os em cands */
    imprime(cands);
    Arrays.sort(cands);
    imprime(cands); } ... }
```

```
class Candidato implements Comparable
{ String nome;
  int inscricao;

  public int compareTo (Object cand)
  { return nome.compareTo
           ((Candidato) cand.nome); }
}
```



# *Applets*

- Programas que podem ser executados por programas de navegação na Web ( “*browsers*” )
- possivelmente trazidos de computador remoto
- Iniciação por navegador a partir de uma página da Web
- em vez de pelo sistema operacional

# *E/S Gráfica*

- E/S em applets feita por (chamados) “componentes de interface gráfica”
- disponíveis na biblioteca *Swing*
- importação: `import javax.swing.*;`



# Páginas da Internet

- Descritas usando linguagem de descrição de hipertextos
- por meio de “marcas”.
- Chamadas de “*linguagens de marcação*”
- Linguagem *HTML* (*HyperText Markup Language*) é a linguagem mais usada para descrição de páginas na Web



# Linguagens de Marcação

- Contêm (além do texto propriamente dito) *marcas* que determinam estrutura e outras características que definem o aspecto do documento
- *Páginas dinâmicas* incluem também recursos para interação com usuários



# HTML

- Marca HTML começa com caractere <
- Segue nome da marca, e possivelmente parâmetros
- e finalmente o caractere >
- Texto seguinte modificado pela marca
- sendo fim do efeito especificado por marca com mesmo nome, só que precedido por /
- Ex: efeito da marca <strong> indica que texto seguinte deve ser escrito em negrito; terminado por </strong>

# Exemplo de página HTML

```
<html>
  <head> <title>MiniCalc</title> </head>
  <body> <h1>MiniCalc</h1> <hr>
    <applet code="MiniCalc.class" width=300 height=100>
      <param name=ops value="mult exp fat">
    </applet> <hr>
    <a href="http://www.dcc.ufmg.br/~camarao/ipcj/java/MiniCalc.java">
      Programa fonte</a>
  </body>
</html>
```



# Marcas HTML

- Conteúdo de página entre `<html>` e `</html>`
- Cabeçalho (opcional) entre `<head>` e `</head>`
- Cabeçalho pode especificar título da página entre marcas `<title>` e `</title>`
- Corpo da página especificado entre `<body>` e `</body>`
- Marca `applet` provava iniciação de *applet*, quando a página é mostrada por *navegador*

# Marca applet

```
<applet code="MiniCalc.class" width=300 height=100>  
...  
</applet>
```

- Parâmetro `code` da marca `applet` especifica arquivo contendo *bytecodes* a serem interpretados
- Definidos também largura e altura da janela a ser usada para interface com programa (em “*pixels*”)

# Interface com *applet*

```
<applet code="MiniCalc.class" width=320 height=120>  
  <param name=ops value="mult exp fat"> </applet>
```

- Parâmetro code deve ser sempre especificado
- Argumentos para *applet* passados por meio da marca param, que tem “atributos” name e value
- Valores passados de página HTML para *applet* como cadeias de caracteres
- Valor passado por ops é "mult exp fat"

# Usando argumentos em *applets*

- Método *getParameter* da classe *JApplet* obtém argumento passado por meio da marca `param`
- Chamada a *getParameter* deve especificar como argumento nome do parâmetro — no exemplo, "ops"
- Valor do tipo *String* retornado — no exemplo, "mult exp fat"

# *JApplet*

- Programa iniciado a partir de página da Internet deve conter subclasse da classe *JApplet* — da biblioteca *Swing*
- ou *Applet* — da biblioteca *AWT*
- Deve redefinir um ou mais dos métodos *init*, *start*, *stop* e *destroy*
- Definidos na classe *JApplet* e *Applet*

# Iniciação de *applets*

`public void init()` : primeiro método a ser chamado, quando *applet* é iniciado. Chamado uma única vez.

Usado para atribuição de valores iniciais a variáveis do programa e para criação de objetos.

`public void start()` : Chamado após *init* e toda vez que página é “*visitada*”.

# Terminação de *applets*

`public void stop()` : Chamado sempre que página deixa de ser mostrada

Ex: navegador mostra outra página

`public void destroy()` : Chamado, após o método *stop*, quando o programa termina sua execução



# *Applets* e segurança

- Código de *applet* (*bytecodes*) carregado a partir de página da Web, e interpretado localmente
- Sujeito a restrições por questão de segurança







# *Applets* e segurança

- *Applet* em geral obtém informações fornecidas diretamente pelo usuário
- e produz dados, que podem ser armazenados apenas por solicitação explícita do usuário
- não pode alterar sistema no qual é executado, nem obter dados de forma não autorizada pelo usuário
- Em geral, *applets* não podem criar nem ler arquivos, nem executar outros programas.

# *AWT e Swing*

- *AWT* (*Abstract Window Toolkit*) compõe núcleo da *JFC* (*Java Foundation Classes*),
- da qual *Swing* também faz parte
- *Swing* definida com base em *AWT*
- e usa classes de *AWT* para gerenciamento da disposição de componentes em janelas (*LayoutManagers*) e interfaces com métodos para tratamento de eventos

# *AWT e Swing*

- Em geral, para cada componente *AWT* existe um componente *Swing* análogo
- precedido do caractere “*J*”
- Ex: *Applet*   *JApplet*  
          *Frame*    *JFrame*
- Existem, no entanto, muitos componentes *Swing* para os quais não existe componente *AWT* análogo.



# Objetivos do Projeto *AWT*

- Objetivos até certo ponto conflitantes:
  - ★ permitir que interfaces gráficas fossem criadas da mesma maneira, independentemente de ambiente ou sistema (Unix, Windows, Mac etc.)
  - ★ permitir que essas interfaces tivessem a mesma aparência das interfaces nativas desses ambientes

# Motivação do Projeto *AWT*

- Fazer com que programador não precisasse se preocupar com detalhes específicos de cada ambiente na implementação de interface gráfica
- Usado então esquema de *pares*: para cada componente *AWT* existe um par equivalente, implementado especificamente para cada plataforma.
- Problema: funcionalidade de componentes limitada à encontrada em *todos* os ambientes nos quais é usada.

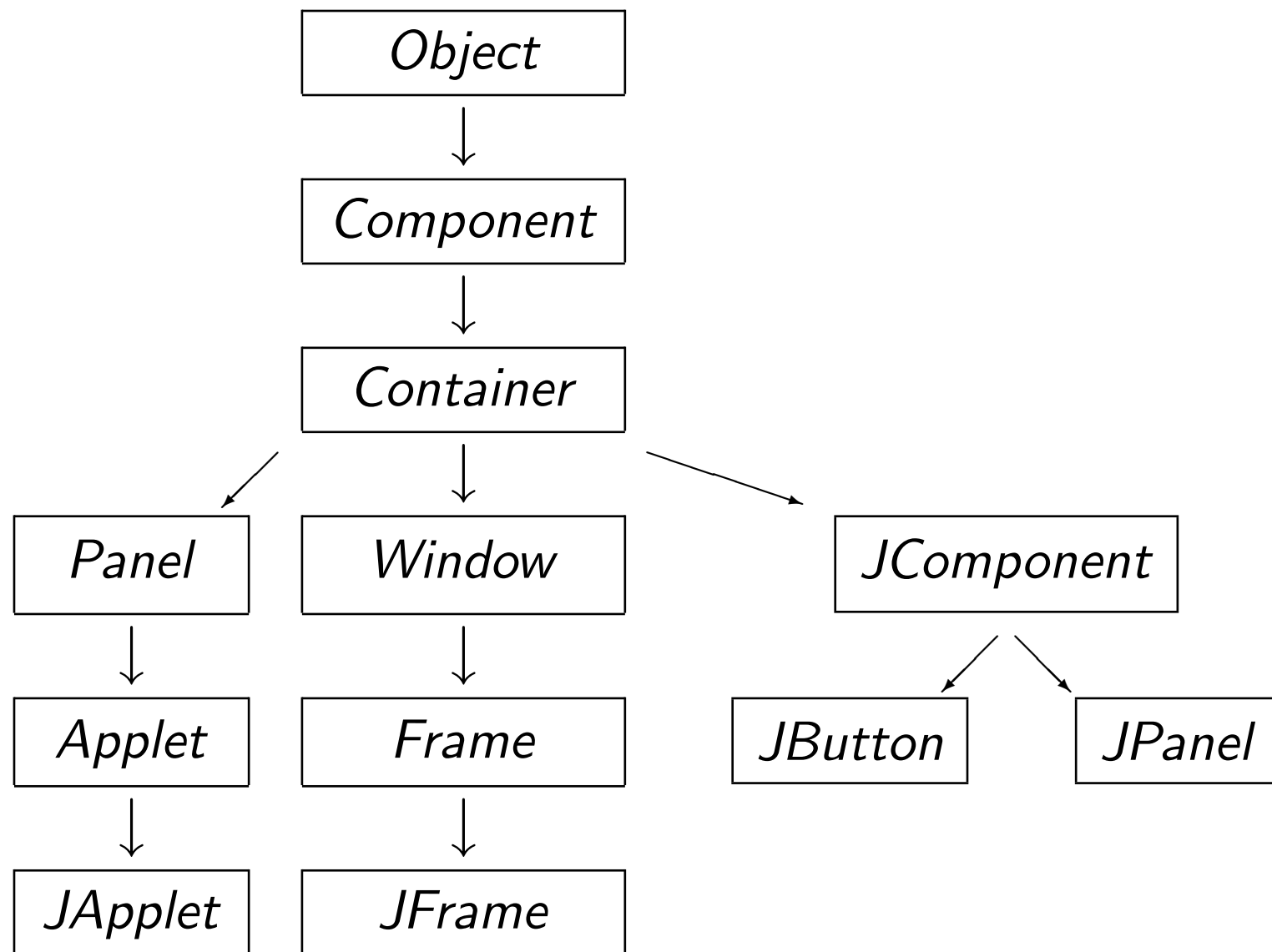


# Projeto *Swing*

- Escrita em Java, não dependendo assim de “código nativo” em dado ambiente, com dada funcionalidade
  - ★ Permite assim definição de mais componentes, com características e funcionalidades variadas
- Depende menos do sistema de interface de usuário nativo, e portanto está menos sujeita a erros na plataforma nativa
- Torna mais fácil usar interface de usuário com mesma aparência em plataformas diferentes, e permite mudança na aparência, para um mesmo código, devido ao uso da arquitetura *MVC*

# Arquitetura *MVC*

- Decomposição entre
  - ★ modelo: criação e manipulação de componentes
  - ★ visão: aspecto visual dos componentes
  - ★ controle: definição e tratamento de eventos associados aos componentes
- Arquitetura desenvolvida no final da década de 70
- Divulgada a partir da criação do ambiente de desenvolvimento da linguagem *Smalltalk*





# Eventos em Componentes

Método de delegação de tratamento de eventos:

1. Implementar método definido em interface, como:  
*ActionListener*, *WindowListener*, *MouseListener* etc.
2. Usar *addActionListener* para indicar objeto responsável por tratar eventos em componente:

```
cp.addActionListener(c);
```

# Implementando Interface *ActionListener*

Necessário implementar *actionPerformed*:

```
public class C implements ActionListener
{ ...
    public void actionPerformed (ActionEvent e)
    { /* código para tratamento do evento */
        ...e.getSource(); ... }
}
```





# *actionPerformed*

1. *actionPerformed* tem parâmetro do tipo *ActionEvent*, que descreve evento ocorrido.
2. Implementação envolve, em geral, obter o objeto que representa o componente no qual o evento ocorreu, usando *getSource*

# Disposição dos componentes

1. Disposição de componentes definida por meio do método *setLayout*, definido na classe *Container*
2. Bibliotecas *AWT* e *Swing* definem classes para suporte a diferentes modos de dispor componentes em painel



*FlowLayout*      *GridLayout*  
*BorderLayout*   *GridBagLayout*



# *FlowLayout*

1. Disposição seqüencial: esquerda para direita e de cima para baixo
2. Componente colocado na linha seguinte quando não cabe mais em uma linha
3. Construtor permite definir
  - número de espaços entre componente e o seguinte, na mesma linha, e número de espaços entre linhas adjacentes
  - alinhamento de componentes em cada linha (à esquerda, ao centro ou à direita)
  - se não especificado, alinhamento é centralizado (*CENTER*), e espaço entre componentes igual a 5 pontos

```
{ String texto = " Proemiatur apte, narrat aperte,"  
  "pugnat acriter, colligit fortifier, ornat excelse." +  
  "Postremo docet, delectat, afficit.";   
  getContentPane().setLayout(new FlowLayout(FlowLayout.RIGHT,10,5));  
  StringTokenizer tokens = new StringTokenizer(texto);  
  while (tokens.hasMoreTokens())  
    getContentPane().add(new  JButton(tokens.nextToken()));  
  setSize(400,200); } }
```



# *GridLayout*

1. Disposição em forma de tabela
2. Construtor define número de linhas e colunas

```
import java.awt.*; import java.applet.*;
import java.util.*;
public class ExemploGridLayout extends JApplet
{ public void init()
  { String texto = "Proemiatur apte, narrat aperte,"
    " pugnat acriter, colligit fortifier, ornat excelse,"
    " Postremo docet, delectat, afficit.";
  getContentPane().setLayout(new GridLayout(7,2));
  StringTokenizer tokens = new StringTokenizer(texto);
  while (tokens.hasMoreTokens())
    getContentPane().add(new JButton(tokens.nextToken()));
  setSize(400,200); } }
```







# *BorderLayout*

1. *BorderLayout* permite dispor componentes nas posições centro, norte, sul, leste e oeste de um painel
2. Classe *BorderLayout* implementa interface *LayoutManager2*, subtipo de *LayoutManager*





```
import java.awt.*; import javax.swing.*;
class ExemploBorderLayout extends JFrame
{ ExemploBorderLayout()
  { super("Exemplo de BorderLayout"); setSize(300,200);
    Container c = getContentPane();
    c.setLayout(new BorderLayout());
    c.add(new JButton("centro"), BorderLayout.CENTER);
    c.add(new JButton("norte"),      BorderLayout.NORTH);
    c.add(new JButton("sul"),        BorderLayout.SOUTH);
    c.add(new JButton("leste"),      BorderLayout.EAST);
    c.add(new JButton("oeste"),      BorderLayout.WEST);
    setVisible(true); }
  public static void main(String[] a) {new ExemploBorderLayout();}}
```



# *GridBagLayout*

1. *LayoutManager* mais poderoso e complexo
2. baseado em tabela (*grid*), mas
3. componentes podem ocupar várias linhas e colunas





# Usando *GridBagLayout*: Primeiro Passo

1. em um papel, divida o painel em linhas e colunas, contendo os componentes da interface gráfica
2. Componentes inseridos de acordo com número das linhas e colunas (números começam de zero)
3. Use *GridBagConstraints* para especificar posição



## *GridBagConstraints*

1. *gridx* / *gridy*: coluna / linha (nº) do canto superior esquerdo do componente
2. *gridwidth* / *gridheight* : nº de colunas / linhas ocupadas
3. *weightx* / *weighty*: espaço horizontal / vertical ocupado ao mudar de tamanho (em relação a outros componentes na mesma linha / coluna)



## *GridBagConstraints:fill*

Variável *fill* de objeto da classe *GridBagConstraints* controla mudança de tamanho:

1. *NONE*: não muda
2. *VERTICAL*: muda verticalmente
3. *HORIZONTAL*: muda horizontalmente
4. *BOTH*: muda em ambas as direções



## *GridBagConstraints.anchor*

Variável *anchor* de objeto da classe *GridBagConstraints* controla posição quando componente não ocupa toda área de linhas/colunas:

1. *NORTH*, *NORTHEAST*, *NORTHWEST*, *CENTER* (default), *EAST*, *SOUTHEAST*, *SOUTHWEST*.