Programação Orientada por Objetos usando Java

UNI-BH
Centro Universitário de Belo Horizonte

Denilson /Anderson / Ana Paula

Visão Geral de Java

- "Java é uma linguagem simples, orientada por objetos, distribuída, robusta, segura, independente de arquitetura, portável, interpretada, de alto desempenho, com suporte a múltiplas linhas de execução (*multi-threaded*) e dinâmica."

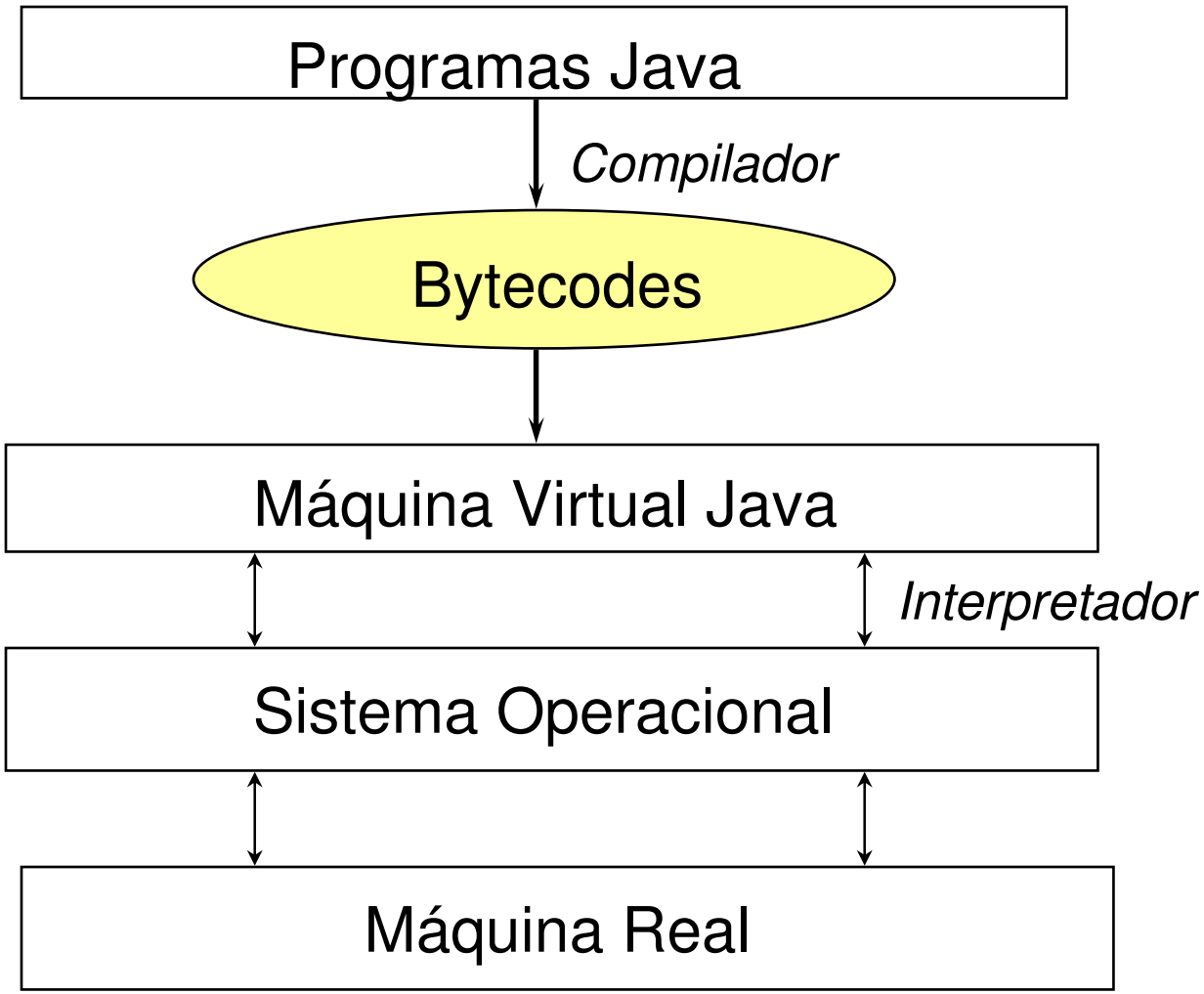
  SUN Microsystems, maio de 1995

- Java é uma linguagem de propósito geral:
  - pode ser usada para a construção de pequenos programas (Applets) que rodam em *browsers.*
  - pode ser usada para a construção de complexas aplicações.

Linguagem Java

- Linguagem *realmente* orientada por objetos
  - Menos que Eiffel
  - Mais que C++ ou Object Pascal (Delphi)
- Sintaxe baseada em C.
- Tipos de dados básicos similares a C.
- Remoção de várias características perigosas de C++ (pointers).
- Gerenciamento de memória automático.
- Grande biblioteca que inclui Web, Interfaces Gráficas (GUI) e Redes.

Filosofia de Programação



Filosofia de Programação

- Programas Java são compostos por classes, armazenadas em arquivos texto com extensão .java.
- Estes programas podem ser editados por um editor de texto convencional e são armazenados em disco como um arquivo convencional.
- Através do processo de compilação, um código objeto é gerado a partir do código fonte. Este código objeto, denominado *bytecode,* é armazenado em disco como um ou mais arquivos de extensão .class.
- Uma vez gerado o código objeto Java (*bytecodes*), o mesmo é interpretado por uma máquina virtual, que traduz cada instrução do *bytecode* para uma instrução que o computador nativo possa entender.

Filosofia de Programação

- Resumindo:
  - Compilador: atua no código fonte e gera código intermediário (*bytecodes*)
  - *Bytecodes*: Independente de arquitetura
  - Máquina virtual Java
    - Carrega os *bytecodes* na memória (*Class Loader*)
    - Verifica os *bytecodes* (*Bytecode Verifier*)
    - Interpreta *bytecodes* diretamente para a arquitetura da máquina real

Ambientes de Desenvolvimento

- Linha de Comando:
  - JDK: Java Developers Kit
- Ambientes Integrados:
  - Eclipse
  - Sun NetBeans
  - Borland JBuilder
  - Microsoft Visual J++

Livros de Java

- Core Java. Volume 1 - Fundamentos
  - Cay Horstmann & Gary Cornell
- Java Como Programar
  - Harvey Deitel & Paul Deitel
- Java in a Nutshell
  - David Flanagan
- Thinking in Java
  - Bruce Eckel
  - Disponível em www.bruceeckel.com

Plataforma Java

- Formada por três partes:
  - Máquina virtual Java (JVM)
  - Linguagem Java
  - Biblioteca de classes Java (API)
- API (*Application Programming Interface*) Java
  - Complementa a linguagem Java com um conjunto de rotinas específicas para diversas tecnologias.
  - Possui milhares de métodos.

JDK

- JDK (*Java Development Kit*) constitui o ambiente básico para desenvolvimento de software em Java. Ele é composto de:
  - um conjunto de ferramentas de desenvolvimento;
  - APIs que compõem o núcleo de funcionalidades da linguagem;
  - APIs que compõem a extensão padronizada ao núcleo;
  - ambiente *runtime* (a Máquina Virtual Java, ou JVM).

Ferramentas de Desenvolvimento

- Principais ferramentas de desenvolvimento incorporadas ao kit de desenvolvimento:
  - **javac** - compilador;
  - **java** - interpretador de aplicações;
  - **appletviewer** - interpretador de applets;
  - **javadoc** - gerador de documentação para programas;
  - **jar** - manipulador de arquivos comprimidos no formato Java Archive, juntamente com **extcheck**, o verificador de arquivos nesse formato;
  - **jdb** - depurador de programas;
  - **javap** - *disassembler* de classes Java;
  - **javah** - gerador de arquivos *header* para integração a código nativo em C.

PATH e CLASSPATH

- A variável de ambiente **PATH** deve incluir o diretório contendo as ferramentas de desenvolvimento
  - Em ambiente Windows, defina a variável PATH da seguinte forma:

    SET PATH=c:\jdk1.4\bin;%PATH%
- A variável de ambiente **CLASSPATH** deve incluir os diretórios contendo a estrutura de pacotes Java
  - SET CLASSPATH=c:\MeuPacoteJava;c:\prog\java;
  - Em algumas versões do JDK, o diretório corrente, ou seja, o diretório . precisa estar em CLASSPATH.

    SET CLASSPATH=.;%CLASSPATH%

Linha de Comando

- O programa fonte é um arquivo texto e pode ser digitado em qualquer editor de texto
  - NotePad (Bloco de Notas), TextPad ou outro qualquer.
- Cada programa fonte é compilado usando o compilador **javac**
  - Recebe como entrada um arquivo **.java**
  - Gera um ou mais arquivos **.class**
  - Ex.: javac MeuPrograma.java
- A interpretação e execução do programa são efetuadas usando o interpretador **java**
  - Recebe como entrada um arquivo **.class**
  - Executa o programa **.class** através da criação de uma máquina virtual. Ex.: java MeuPrograma

Programas em Java

- Um arquivo fonte é constituído por um conjunto de classes
  - Normalmente um arquivo contém apenas uma classe.
- Classes definem Tipos Abstratos de Dados
  - Podem conter variáveis (atributos), funções e procedimentos (métodos).
- A classe principal em um arquivo fonte é qualificada pela cláusula **public**
  - Somente uma classe é public em um arquivo.
  - A classe public deve ter o mesmo nome do arquivo.

Programas em Java

```
class Classe1 {
...
}
class Classe2 {
...
}
public class
Classe3 {
...
}
```

Nome do arquivo:
Classe3.java

Primeiro Programa

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

- O nome do arquivo deve ser PrimeiroPrograma.java.
- Java é sensível a letras maiúsculas e minúsculas.
- A classe compilada é armazenada em um arquivo .class.

Método main

- O método **main()** é o ponto de início de execução de uma aplicação Java (exceto para applets).
  - A primeira classe a ser chamada em uma aplicação possui o método main para iniciar a aplicação.
  - As demais classes podem implementar o método main para testes da própria classe.
- Assinatura do método main:

  public static void main(String[] args)

  ou

  static public void main(String args[])
  - O nome do parâmetro (args) poderia ser diferente, mas os demais termos da assinatura devem obedecer ao formato especificado.

Método main

- Parâmetro
  - O método main recebe como argumento um parâmetro do tipo arranjo de objetos String.
  - Cada elemento desse arranjo corresponde a um argumento passado para o interpretador Java na linha de comando que o invocou.
  - Ex.: java Teste aaaa 22 zzz
    - O método main(String args[]) da classe Teste vai receber, nessa execução, um arranjo de três elementos na variável args com os seguintes conteúdos:
      - args[0] - objeto String com conteúdo "aaaa";
      - args[1] - objeto String com conteúdo "22";
      - args[2] - objeto String com conteúdo "zzz".

Método main

- Valor de Saída
  - O método main é do tipo ***void***. Ele não tem valor de retorno.
  - Se for necessário retornar um valor deve-se usar o método ***System.exit(int)***.
    - A invocação desse método provoca o fim imediato da execução do interpretador Java.
    - Tipicamente, o argumento de exit() obedece à convenção de que "0" indica execução com sucesso, enquanto um valor diferente de 0 indica a ocorrência de algum problema.

Outros Detalhes do Primeiro Programa

- Observe no exemplo do PrimeiroPrograma o uso dos pares de chaves **{ }**. As chaves delimitam blocos de códigos.
  - Equivalente ao **begin end** do Pascal.
- O método main possui somente uma instrução

  System.out.println ("Alo, mundo!");
  - Por enquanto, saiba apenas que o método **println** imprime uma string na saída padrão.
  - Formalmente: a classe **System** possui um atributo estático **out** do tipo **PrintStream**, que por sua vez, possui o método **println** que imprime uma mensagem na saída padrão.
  - Uma string é delimitada por um par de aspas duplas.
- Toda instrução termina com um ponto-e-vírgula (;).

Convenção de Nomes em Java

- Pacote
  - Letras minúsculas. Ex.: meupacote.
- Classe
  - Primeira letra maiúscula, demais minúsculas. Nomes compostos iniciando com letras maiúsculas.
  - Ex.: MinhaClasse.
- Métodos e Atributos
  - Primeira letra minúscula, demais minúsculas. Nomes compostos iniciando com letras maiúsculas.
  - Ex1: meuMetodo. Ex2: meuAtributo.
- Constantes
  - Letras maiúsculas. Ex.: PI

Comentários em Java

- Java possui três tipos de comentários:
  - Comentário até o final da linha usando **//**

```
// isso é um exemplo de comentário até o final da linha.
```

  - Comentário em blocos usando delimitadores **/*** (início do comentário) e ***/** (fim do comentários)

```
/* o comentário começa aqui,
   continua aqui
   e termina aqui */
```

  - Comentário para documentação usando **javadoc**. O comentário começa com **/**** e termina com ***/**

```
/**  Comentário para a ferramenta javadoc
@version 1.0
@author Denilson Alves Pereira
*/
```

Tipos de Dados

- Há oito tipos primitivos (pré-definidos):
  - Números Inteiros
    - int 4 bytes
      - mais usado
    - short 2 bytes
    - long 8 bytes
      - possui sufixo L. Ex.: 4567L
    - byte 1 byte
      - com sinal: -128 a +127
  - Números em Ponto Flutuante
    - float 4 bytes
      - possui sufixo F. Ex.: 0.56F
    - double 8 bytes
      - duas vezes a precisão do tipo float
      - mais usado

Tipos de Dados

- Caractere
  - char 2 bytes
    - caracteres Unicode
      - » permite até 65536 caracteres (atualmente usados cerca de 35000).
      - » primeiros 255 caracteres idênticos ao código ASCII/ANSI.
      - » representados por '\u0000' a '\uFFFF'.
    - representado por aspas simples. Ex.: 'H'.
      - » "H" representa uma string contendo um único caractere.
    - caracteres especiais:

      \b (backspace) \r (carriage return) \\ (barra invertida)
      \t (tab) \" (aspas duplas)
      \n (linefeed) \' (apóstrofe)
- Lógico
  - boolean 1 bit (valores verdadeiro (*true*) e falso (*false*))

- O restante são objetos (exceto array).

Variáveis

- Java é uma linguagem fortemente tipada
  - toda variável precisa ter um tipo.
- Uma variável é declarada colocando-se o tipo seguido pelo nome da variável. Exemplos:

```
int umaVariavelInteira;
double variavelReal = 14.7;  //variável declarada e inicializada
boolean achou, b;    // declaração de duas variáveis do mesmo
                     // tipo
```

- Variáveis podem ser declaradas em qualquer lugar, desde que antes de seu uso.
  - Variáveis podem ser declaradas na primeira expressão de um loop **for**.

Variáveis

- Escopo de variáveis:
  - Um bloco ou instrução composta é qualquer número de instruções simples Java que são delimitadas por um par de chaves { ... }.
  - Variáveis são sempre locais a um bloco.
    - Não existe o conceito de variáveis globais a um programa.
  - Exemplo:

```
public static void main (String[] args) {
   int n;     // n é local ao método main
   ...
   { int k;
      ...     // k é local a este bloco
   }
   ...        // k não vale aqui
}
```

Atribuição

- Atribuição de valores a variáveis é feita pelo operador "=".
  - Exemplo:

```
char c;     // declaração
c = 'S';    // atribuição
```

- Múltiplas variáveis podem ser atribuídas em uma única expressão.
  - Exemplo:

```
int x, y, z;
x = y = z = 0;
```

- Toda variável deve ser explicitamente inicializada.
  - O compilador geralmente avisa se isso não ocorrer.

Conversões entre Tipos de Dados

- As seguintes regras são válidas para operações binárias de tipos diferentes:
  - Se um dos operandos for *double*, o outro será convertido para *double*.
  - Senão se um dos operandos for *float*, o outro será convertido para *float*.
  - Senão se um dos operandos for *long*, o outro será convertido para *long*.
  - E assim sucessivamente para os tipos *int*, *short* e *byte*.

Conversões entre Tipos de Dados

- Em conversões onde houver possibilidade de perda de informação, as conversões devem ser explícitas (*cast*).
  - A sintaxe é dada pelo tipo resultante em parênteses, seguido pelo nome da variável.
  - Exemplos:

```
double x = 3.987;
int y = (int) x;          //  y = 3  (truncamento)
```

  - para arredondar, use o método Math.round:

```
double x = 3.987;
int y = (int) Math.round(x);      //  y = 4    (arredondamento)
// é necessária a conversão explícita pois round retorna um long.
```

Conversões entre Tipos de Dados

- Não há necessidade de conversão explícita se não houver possibilidade de perda. Da esquerda para a direta, as conversões possíveis são:

  byte --> short --> int --> long --> float --> double

  char --> int

  - Exemplos:

```
double d = 4.5; float f = 66.4F;
long l = 15L;     int i = 2;          char c = 'H';
d = d + f;   // correto
d = f + l;   // correto
f = d + l;   // incorreto, resultado deve ser double
l = 3 * i;   // correto
i = c;       // correto, resultado inteiro correspondente ao
             // código Unicode do caractere 'H'.
```

Constantes

- Uma constante é definida usando-se as palavras-chave ***static final***. Exemplo:

```
public class ExemploConstante {
   public static final double PI = 3.14;
   public static void main (String[] args) {
     System.out.println("O valor de Pi é "+ PI);
   }
}
```

  - A convenção de Java é sempre usar letras maiúsculas para constantes.

Operadores Aritméticos

- Operadores Aritméticos:
  - `+` adição
  - `-` subtração
  - `*` multiplicação
  - `/` divisão (inteira, se os operandos forem inteiros e ponto flutuante, caso contrário)
  - `%` módulo (resto de divisão inteira)

  - Não existe operador de exponenciação. Para isso, use o método *pow* da classe *Math* de *java.lang*.

    ```
    double y = Math.pow(x,a);  // y = x^a
    ```

  - A classe *Math* possui um grande número de funções matemáticas.

Operadores Aritméticos

```
public class TesteAritmetico {
    public static void main (Strings args[]) {
        short x = 6;
        int y = 4;
        double a = 12.6;
        double b = 3.0;
        System.out.println ("x é " + x + ", y é " + y );
        System.out.println ("x + y = " + (x + y) );
        System.out.println ("x - y = " + (x - y) );
        System.out.println ("x / y = " + (x / y) );
        System.out.println ("x % y = " + ( x % y ) );
        System.out.println ("a é " + a + ", b é " + b );
        System.out.println (" a / b = " + ( a / b ) );
        System.out.println (" 11.0 / 3 = " + (11.0 / 3 ) );
        System.out.println (" 11 / 3 = " + (11 / 3 ) );
    }
}
```

Operadores de Incremento e Decremento

- O operador de incremento adiciona 1 a uma variável numérica e o operador de decremento, subtrai 1.

```
int n = 5;
n++;        // faz n = 6
n--;        // faz n = 5
```

  - Forma pré-fixada: o incremento/decremento é executado antes da avaliação.
  - Forma pós-fixada: o incremento/decremento é executado depois da avaliação.

```
int m = 5;
int n = 5;
int a = 2 * ++m;     // faz a = 12  e  m = 6
int b = 2 * n++;     // faz b = 10  e  n = 6
```

Forma Reduzida de Operadores de Atribuição

- Forma reduzida de operadores aritméticos binários:

  x += y significa x = x + y

  x -= y significa x = x - y

  x *= y significa x = x * y

  x /= y significa x = x / y

- Exemplo:

```
int z = 4;
int w = 6;
z += w;    // faz  z = 10
```

Operadores Relacionais e Lógicos

- Operadores Relacionais:

==	Igual	x == 3
!=	Diferente	x != 3
<	Menor que	x < 3
>	Maior que	x > 3
<=	Menor ou igual	x <= 3
>=	Maior ou igual	x >= 3

- Operadores Lógicos:

&&	Operação lógica AND (E)
||	Operação lógica OR (OU)
!	Operação lógica NOT (Negação)

Operadores Bit a Bit

- Operam diretamente os bits de números inteiros.
  - & Operação lógica AND (E)
  - | Operação lógica OR (OU)
  - ^ Operação lógica XOR (OU exclusivo)
  - ~ Operação lógica NOT (Negação)
  - >> Deslocamento de bits à direita
  - << Deslocamento de bits à esquerda
- Exemplo:

```
int x = 6;          // 0110  em binário
int y = 13;         // 1101  em binário
int z = x & y;      // 0100  em binário,  z = 4
z = x << 1;         // 1100  em binário,  z = 12
```

Comandos Condicionais

- Comando **if** ... **else**
  - **if** (condição) comando;
  - **if** (condição) { bloco de comandos }
  - **if** (condição) comando1; **else** comando2;
  - **if** (condição) { bloco1 } **else** { bloco2 }
    - else é opcional.
    - condição entre parênteses
  - Exemplo 1:

```
if (nota >= 70)
   System.out.println("Aprovado");
else
   System.out.println("Reprovado");
```

Comandos Condicionais

- Exemplo 2:

```
if (x > 5) {
   if (y > 5)
      System.out.println("x e y maiores do que 5");
}
else     // sem o bloco, este else pertenceria ao segundo if
   System.out.println("x menor ou igual a 5");
```

- Operador ternário ?
  - condição ? e1 : e2
    - avalia e1 se condição for verdadeira ou e2 caso contrário.
  - Exemplo:

```
System.out.println (nota >= 70 ? "Aprovado" : "Reprovado");
```

Comandos Condicionais

- Comando **switch**

```
switch (opção) {
   case valor1: comandos1; break;
   case valor2: comandos2; break;
   ...
   default: comandosn; break;
}
```

  - opção deve ser do tipo char, byte, short ou int.
  - A execução começa no ***case*** que coincide com o valor da seleção realizada, e continua até o ***break*** seguinte ou o final do ***switch***. A cláusula ***default*** é opcional e será executada se nenhum valor coincidir na seleção.

Comandos de Repetição

- Comando **while**
  - **while** (condição) { bloco }
  - repete execução do bloco enquanto condição for verdadeira.
    - Se condição inicial é falsa, o bloco não é executado nenhuma vez.
  - Exemplo:

```
int cont = 1;
while (cont <= 10) {
   System.out.println ("contador = " + cont);
   cont++;
}
```

Comandos de Repetição

- Comando **do** ... **while**
  - **do** { bloco } **while** (condição)
  - repete execução do bloco enquanto condição for verdadeira.
    - O bloco é executado pelo menos uma vez.
  - Exemplo:

```
int cont = 1;
do {
   System.out.println ("contador = " + cont);
   cont++;
} while (cont <= 10);
```

Comandos de Repetição

- Comando **for**
  - **for** (comando; expressão1; expressão2) { bloco }
    - equivalente a:

```
{  comando;
   while (expressão1) {
      bloco;
      expressão2;
   }
}
```

  - O comando inicializa um contador (que pode ser declarado aqui), a expressão1 fornece a condição de teste antes de cada passagem pelo laço e a expressão2 determina a alteração do contador.
    - Qualquer expressão é válida nos segmentos do for, mas é aconselhável usar apenas o necessário para inicializar, testar e atualizar o contador.

Comandos de Repetição

- Exemplo1:

```
for (int cont = 1; cont <= 10; cont++)
   System.out.println (“contador = “ + cont);
// cont é válida somente dentro do bloco do for
```

- Exemplo2:

```
int i;
int j = 2;
for (i = 15; i > j; i -= 2) {
   System.out.println (“contador = “ + i);
   j++;
}
// i é válida depois do bloco do for
```

Comando break

- **break** não rotulado:
  - Usado para sair de um laço simples.
  - Exemplo:

```
while (i <= 100) {
   saldo = saldo + deposito + juros;
   if (saldo > meta)  break;
   i++;
}
// programa sai do laço while se i > 100 ou se saldo > meta.
// Obviamente, o mesmo efeito poderia ser obtido sem o uso
// do break.
```

Comando break

- **break** rotulado:
  - Usado para sair de laços aninhados.
  - Um rótulo (*label*) precisa preceder o laço mais externo do qual se deseja sair. Um rótulo é seguido de dois pontos (:).
  - Exemplo:

```
int n;
ler_dados:     // rótulo identificador do laço
while ( ... ) {
  ...
  for ( ... ) {
    n = Console.readInt( ... );     // um método para leitura de inteiros
    if (n < 0)   // uma condição que normalmente não deveria ocorrer
      break ler_dados;    // sai do laço ler_dados (while externo)
    ...
  }
}
```

Manipulação de Strings

- Strings são seqüências de caracteres.
- Java não tem um tipo string nativo.
- Java fornece duas classes para manipulação de strings:
  - String
    - operações somente de leitura
    - mais eficiente que StringBuffer (para leitura)
    - possui uma sintaxe mista entre objetos e tipos primitivos
  - StringBuffer
    - operações de leitura e escrita
- As constantes strings são escritas entre aspas.
  - Cada string entre aspas é uma instância da classe String.

Manipulação de Strings

- Exemplos:

  String s = “”; // string vazia

  String saudacao = “Ola”;

- A concatenação de string é feita pelo operador **+**.
  - Exemplo:

    String s = “Curso de ” + “Java”; // s = “Curso de Java”

- Ao concatenar uma string com um valor que não é string, este último é convertido para uma string.
  - Exemplos:

    String censura = 18 + “ anos”;

    int m = 20;

    System.out.prinln(“O valor de m e’ “ + m);

Operações da Classe String

- public String **substring**(int inicio, int fim)
  - Retorna uma substring.
  - Posição inicial é zero.
  - Especifica-se a posição inicial e a primeira posição não incluida.
  - Exemplo:

```
String str = "Curso de Java";
String j = str.substring(9,13);     //  j = "Java"
```

- public int **length**()
  - Retorna o tamanho da string (número de caracteres).
  - Exemplo (continuação):

```
int tam = str.length();      //  tam = 13
```

Operações da Classe String

- public char **charAt**(int indice)
  - Retorna o caractere da posição indice.
  - Exemplo (continuação):

```
char c = str.charAt(9);      //  c = 'J'
```

- public boolean **equals**(Object obj)
  - Retorna true se o conteúdo da string for igual ao conteúdo do objeto (string) obj, e false caso contrário.
  - Exemplo (continuação):

```
boolean b = "Java".equals(j);    //  b = true
```

  - **Obs.**: NÃO use o operador == para testar igualdade de strings.
    - == simplesmente compara se duas strings apontam para um mesmo local.

Operações da Classe String

- public int **compareTo**(String str)
  - retorna:
    - um inteiro menor que zero se a string vier antes (ordem alfabética) que str;
    - zero se a string for igual a str;
    - um inteiro maior que zero se a string vier depois (ordem alfabética) que str.
  - Exemplo (continuação):

```
int comp = j.compareTo("Java");      //  comp = 0
```

- A classe String possui mais de 50 métodos.
  - Consulte a documentação para maiores detalhes!

Operações da Classe StringBuffer

- public StringBuffer **append**(Object obj)
  - Adiciona o objeto (string) obj a uma string já existente.
  - Exemplo:

```
StringBuffer s1 = new StringBuffer("Gundamentos");
StringBuffer s2 = s1.append(" Java");  // s2 = "Gundamentos Java"
```

- public StringBuffer **insert**(int i, Object obj)
  - Insere o objeto (string) obj na posição i.
  - Exemplo (continuação):

```
StringBuffer s3 = s2.insert(11," de"); // s3 = "Gundamentos de Java"
```

- public void **setCharAt**(int i, char c)
  - Substitui o caractere da posição i por c.
  - Exemplo (continuação):

```
s3.setCharAt(0,'F');   // s3 = "Fundamentos de Java"
```

Classe Math

- Pacote: java.lang.Math
- Define constantes e funções matemáticas.
  - Constantes:
    - static double E // base de logaritmos neperianos (e = 2.718...)
    - static double PI // pi (pi = 3.14...)
  - Algumas funções:
    - static double abs(double x) // valor absoluto
    - static double log(double x) // ln(x) (logaritmo neperiano de x)
    - static double exp(double x) // e^x (e elevado a x)
    - static double sin(double x) // seno de x
    - static double cos(double x) // cosseno de x
    - static double sqrt(double x) // raiz quadrada de x
    - static double pow(double x, double y) // x^y (x elevado a y)

Arranjos

- Arranjos são objetos criados dinamicamente.
- O tamanho de um arranjo é definido na criação e não pode ser alterado.
  - se isso for necessário, deve-se usar o objeto *Vector*.
- O índices de um arranjo variam de 0 até tamanho-1.
- A criação de um arranjo é feita pelo operador **new**.
  - Exemplo de criação:

    ```
    int[] valor = new int[100];
    ```

    - define um arranjo que pode conter 100 elementos do tipo int.
    - os elementos são indexados de 0 a 99.
  - Exemplo de preenchimento:

    ```
    for (int i = 0; i < 100; i++)
      valor[i] = i;
    ```

Arranjos

- Arranjos podem ser criados e inicializados usando { e }.
  - int[] numerosPrimos = {2, 3, 5, 7, 11, 13} // sem o operador
    // new
  - new int[] {2, 3, 5, 7, 11, 13} // array anônimo
    - usado para passar um parâmetro para um método sem criar uma variável
- O número de elementos de um arranjo é dado por **length**.

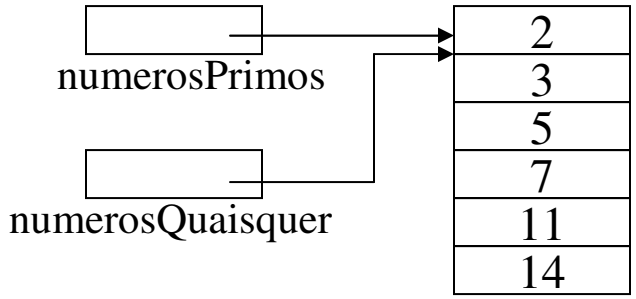
```
for (int i = 0; i < numerosPrimos.length; i++)
  System.out.println(numerosPrimos[i]);
```

- Acessar um índice fora dos limites gera exceção *IndexOutofBoundsException* durante a execução.

Arranjos

- Cópia de arranjos:
  - Uma variável do tipo arranjo é um apontador para uma estrutura contendo os elementos do arranjo.
  - Copiar uma variável do tipo arranjo para outra significa apontar as duas variáveis para a mesma estrutura.
  - Exemplo:

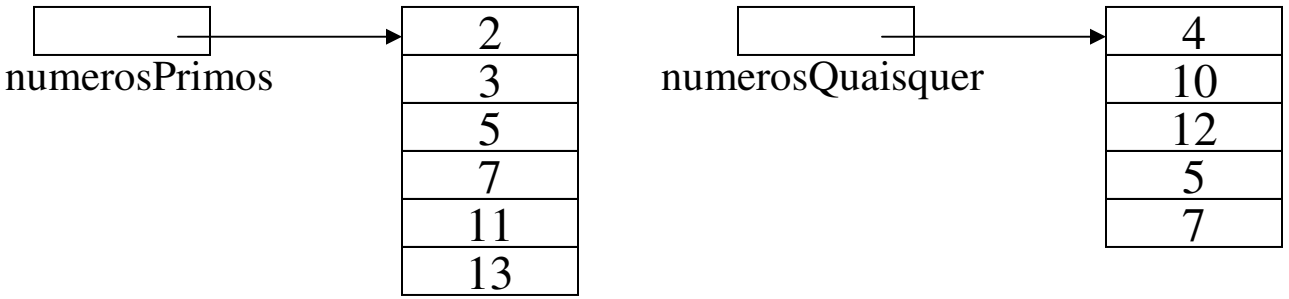
```
int[] numerosPrimos = {2, 3, 5, 7, 11, 13};
int[] numerosQuaisquer = numerosPrimos;
numerosQuaisquer[5] = 14;
```



Arranjos

- Cópia de arranjos:
  - Para realmente copiar valores de um arranjo para outro, pode-se usar o método *arraycopy* da classe *System*.
    - static void arraycopy(Object fonte, int posicaofonte, Object destino, int posicaodestino, int tamanho)
    - Exemplo:

```
int[] numerosPrimos = {2, 3, 5, 7, 11, 13};
int[] numerosQuaisquer = {4, 10, 12, 14, 20};
System.arraycopy(numerosPrimos,2,numerosQuaisquer,3,2);
```



Arranjos

- Arranjos Multidimensionais:
  - Declaração:
    - double[][] matriz; // arranjo bi-dimensional
    - double[][] matriz2 = new double[5][4]; // arranjo bi-dimensional 5x4
    - int[][] vendas = {{1999, 2000}, {100000, 120000}}; // inicialização
    - String[][][] triStr; // arranjo tri-dimensional
  - Prenchimento:
    - matriz[1][2] = 12;
    - triStr[3][3][2] = "Teste";
  - Arranjos multidimensionais são arranjos de arranjos.
    - Um arranjo contém elementos e cada um dos elementos pode ser um outro arranjo.
    - Pode-se criar arranjos "irregulares", ou seja, arranjos em que linhas diferentes têm comprimentos diferentes.

Métodos

- Método é uma função ou procedimento (módulo). Implementa uma operação realizada por uma classe.
- Definição de um método:

```
tipo_de_retorno   nome_do_método (lista_de_parâmetros)
{
   comandos;
}
```

  - tipo_de_retorno pode ser qualquer tipo de dado Java.
    - **void** não retorna nada.
  - lista_de_parâmetros contém zero ou mais parâmetros separados por vírgulas. Cada parâmetro possui um tipo e um identificador.
  - { e } indicam o início e o fim do método (corpo do método).

Métodos

- Variáveis declaradas dentro de um método são locais ao método.
  - O espaço de memória alocado a elas é liberado automaticamente ao término do método.
- O método retorna para o ponto onde foi chamado por:
  - return;
  - return expressão;
- Os parâmetros dos métodos são sempre passados por valor e não por referência.
  - Arranjos e objetos passam uma referência. O conteúdo apontado pela referência pode ser modificado pelo método.

Métodos

- A chamada a um método é dada pelo nome da classe ou do objeto seguido pelo operador . (ponto) e pelo nome do método.
  - Exemplos:
    - Math.sqrt(12);
    - System.out.println("Ola");
  - Chamadas a métodos locais a uma classe não precisam conter o nome da classe.
- Métodos podem ser recursivos (chamar a si próprio, direta ou indiretamente).

Métodos

- Java permite que vários métodos com o mesmo nome sejam definidos (sobrecarga de métodos).
  - As listas de parâmetros desses métodos devem ser diferentes.
  - Quando o método é chamado, o compilador Java seleciona o método pelo número, tipo e ordem dos parâmetros.

Métodos

- Exemplo de passagem de parâmetro:

```
public class Passagem1 {
  public static void divida(float y) {
     y /= 2.0;
     System.out.println("y: " + y);
  }
  public static void main (String args[]) {
     float x = 1.0f;
     System.out.println("primeiro x: " + x);
     divida(x);
     System.out.println("segundo x: " + x);
  }
}
```

  - resultado:

```
primeiro x: 1.0
y: 0.5
segundo x: 1.0
```

Métodos

- Exemplo de passagem de parâmetro:

```
public class Passagem2 {
   public static void altera(int w[]) {
      w[2] = 4;
   }
   public static void main (String args[]) {
      int y[] = {1,2,3};
      altera(y);
      for (int i = 0; i < y.length; i++)
          System.out.print(y[i]+" ");
   }
}
```

- resultado:

  1 2 4

Pacotes

- Pacote é um recurso para agrupar física e logicamente classes e interfaces relacionadas.
- Um pacote consiste de um ou mais arquivos.
- Um arquivo pode ter no máximo uma classe pública.
- Os tipos públicos definidos no pacote podem ser usados fora do pacote da seguinte maneira:
  - prefixados pelo nome do pacote.
  - importados diretamente via cláusula import.
- Pacote é uma coletânea de arquivos de classes individuais.
- O nome de um pacote corresponde a um nome de diretório.
- Como pacote é um diretório, pode haver hierarquia de pacotes.

Pacotes

- Um arquivo pertencente a um pacote inicia-se com a instrução **package**.
- Se um arquivo importa (usa) classes de um pacote, em seguida, vem a instrução **import**.
  - O pacote ***java.lang*** é importado automaticamente.
- Exemplo:

```
package cursojava;     // define um pacote de nome cursojava
import java.util.Arrays;  // importa classe Arrays do pacote java.util
import java.io.*;   // importa todas as classes do pacote java.io
public class Ordena {  // define uma classe pertencente ao
                       // pacote cursojava
   ...
}
```

Pacotes

- Cláusula **import**
  - A importação pode ser individual ou coletiva(*).
  - Importações supérfluas são descartadas.
  - O escopo de uma importação é a unidade de compilação.
- O acesso ao bytecode de uma classe precisa do:
  - nome da classe.
  - nome de seu pacote.
  - valor de CLASSPATH.
- CLASSPATH é uma variável de ambiente que informa ao carregador de classes onde procurá-las.
  - Exemplo: set CLASSPATH=c:\projeto\fonte;

Pacotes

- Para procurar por uma classe, o carregador de classes:
  - pega o nome completo da classe incluindo o pacote que a contém.
    - Exemplo: java.lang.Math
  - Substitui os pontos por separadores de diretórios.
    - java\lang\Math
  - Acrescenta o sufixo .class
    - java\lang\Math.class
  - Prefixa o resultado acima com cada um dos elementos de CLASSPATH.
  - Usa os caminhos obtidos para ter acesso ao arquivo de bytecode da classe desejada.

Pacotes

- Pacotes da API Java:
  - java.applet
    - criação de applets, interação de applets com o browser.
  - java.awt
    - criação e manipulação de interfaces gráficas de usuário.
  - java.awt.event
    - tratamento de eventos dos componentes da interface.
  - java.io
    - entrada e saída de dados.
  - java.lang
    - importado automaticamente por todos os programas, contém classes e interfaces básicas.

Pacotes

- Pacotes da API Java:
  - java.net
    - comunicação via Internet.
  - java.rmi
    - criação de programas distribuídos.
  - java.sql
    - interação com bancos de dados.
  - java.text
    - manipulação de textos, números, datas e mensagens.
  - java.util
    - manipulação de datas, hora, geração de números aleatórios etc.
  - javax.swing
    - componentes de interface gráfica independentes de plataforma.

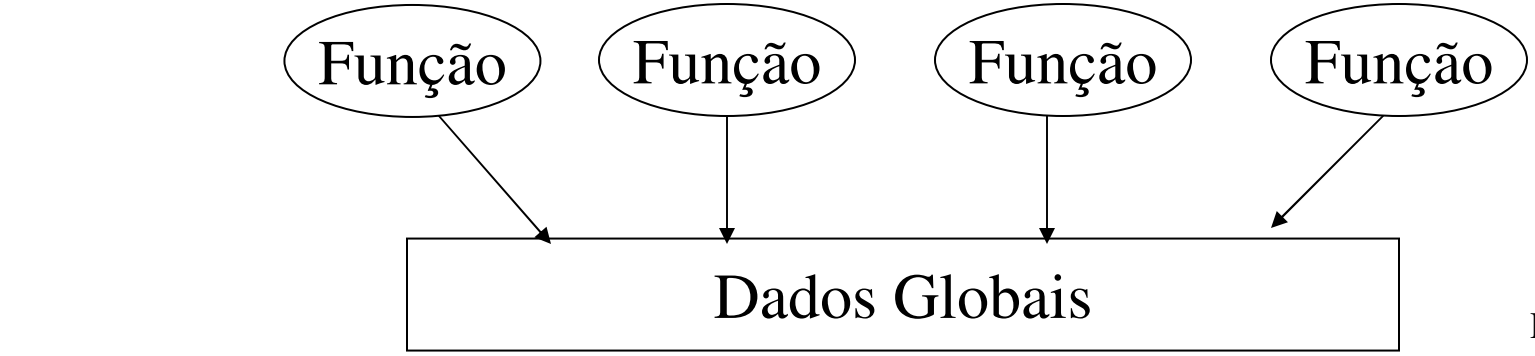
Programação Orientada por Objetos (POO)

- Programação Tradicional:
  - Na programação convencional, orientada a rotinas em uma linguagem como C, por exemplo, o problema é visualizado como uma seqüência de coisas a serem feitas. Os itens de dados relacionados são organizados em estruturas C (*structs*) e as funções necessárias (rotinas) são escritas para manipular os dados.
- Programação Orientada por Objetos:
  - A Programação Orientada a Objetos (POO) toma a vantagem da modularidade dos objetos para implementar um programa em unidades relativamente independentes que são mais fáceis de manter e de estender.

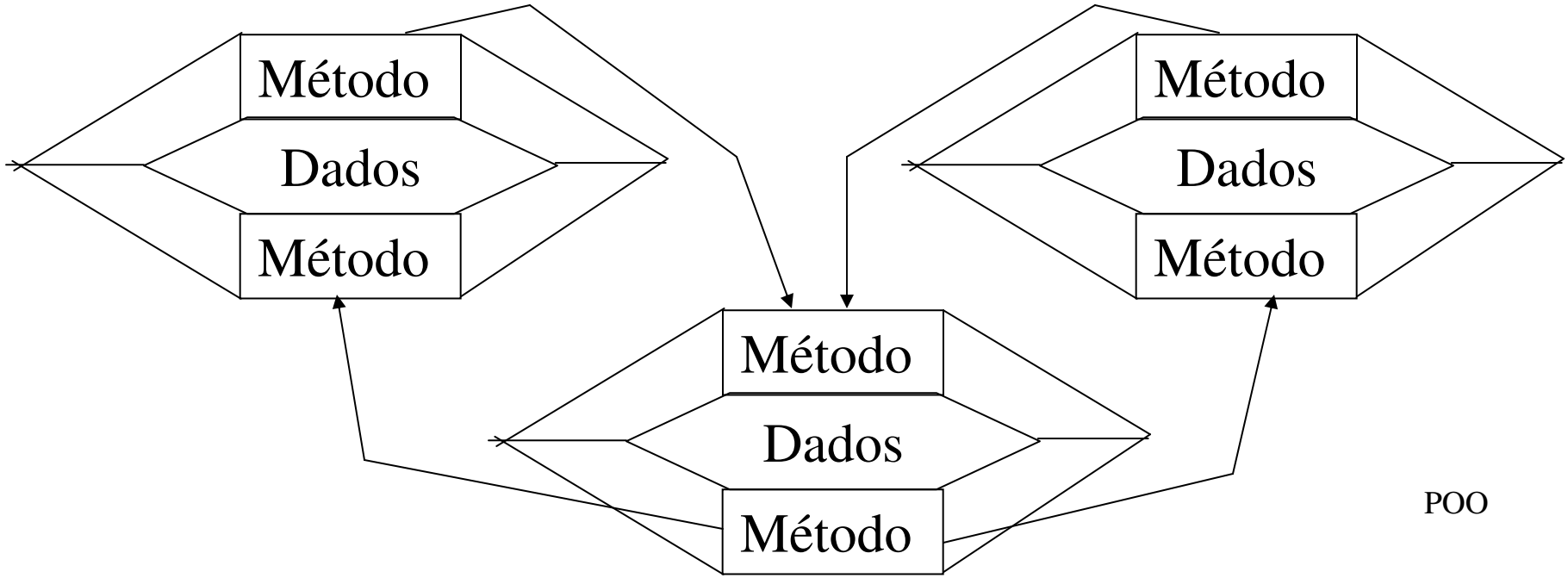
Programação Orientada por Objetos (POO)

- Programação Orientada por Objetos:
  - POO é apenas um método para projetar e implementar software. As técnicas usadas não acrescentam nada ao produto final visto pelo usuário. Entretanto, essas técnicas oferecem vantagens significativas para gerenciar problemas complexos, especialmente em grandes projetos.

Programação Procedimental vs. POO



Procedimental



POO

Objeto

- Um objeto representa uma entidade física ou uma entidade conceitual ou uma entidade de software. (definição informal)
- Um objeto é um conceito, abstração, ou algo com fronteiras bem definidas e significado para uma aplicação.
- Um Objeto é algo que possui:
  - Estado
    - armazena informação sobre o objeto.
  - Comportamento
    - define as mensagens aceitas pelo objeto.
  - Identidade
    - identifica unicamente o objeto.

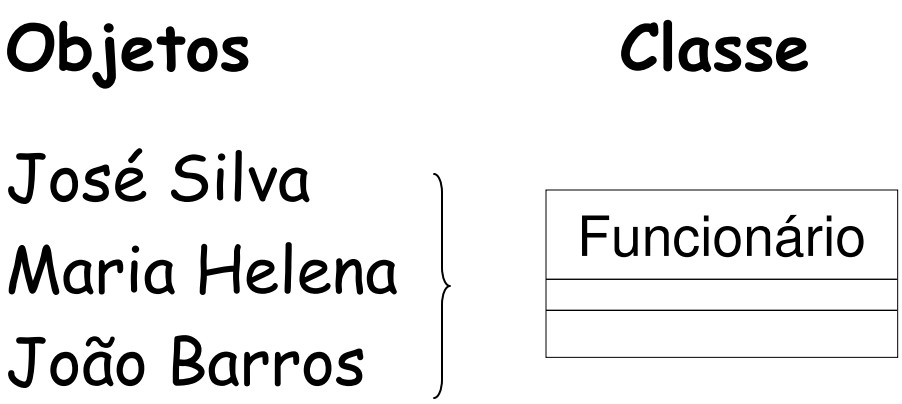
Classe

- Uma classe é a descrição de um grupo de objetos com propriedades similares (atributos), comportamento comum (operações), relacionamentos comuns com outros objetos e semânticas idênticas.
  - Um objeto é uma instância de uma classe.
- Exemplo:

Classe	**Atributos**	**Operações**
Pedido	Número	Adiciona item
	Data	Cancela
	Vendedor	Confirma venda

Classes e Objetos

- Uma classe é uma definição abstrata de um objeto.
  - Define a estrutura e o comportamento de qualquer objeto da classe.
  - Serve como um padrão para criação de objetos.
- Os objetos podem ser agrupados em classes.



Classe e Tipo Abstrato de Dados

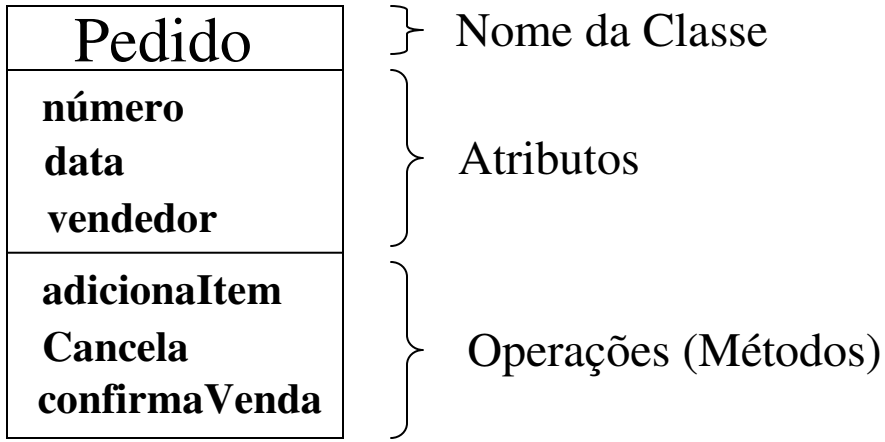
- Tipo Abstrato de Dados (TAD)
  - Abstração de Dados é o processo de definição de um tipo de dados chamado *tipo de dados abstrato* (TAD), usando ocultação de dados.
  - A definição de um TAD especifica não apenas a representação interna dos dados do TAD, mas funções que outros módulos de programa usarão para manipula-lo.
  - Ocultar dados garante que você pode alterar a estrutura interna do TAD sem quebrar programas que chamam funções operando com aquele TAD.
- Uma Classe é um Tipo Abstrato de Dados.

Guia para Encontrar Classes

- Uma classe deve capturar uma e apenas uma abstração chave.
  - Abstração Ruim: Classe Cliente que conhece as informações de um cliente e seus pedidos.
  - Boas Abstrações: Classes distintas para o cliente e para o pedido.
- Os nomes das classes devem vir diretamente do vocabulário do domínio do problema.
  - A dificuldade em nomear uma classe pode ser uma indicação de uma abstração mal definida.

Representação de Classes

- Um classe possui atributos e operações (métodos).
- Em UML, uma classe é representada utilizando-se um retângulo dividido em três seções:
  - A primeira seção contém o nome da classe.
  - A segunda seção mostra a estrutura (atributos).
  - A terceira seção mostra o comportamento (operações).
- Exemplo:



Atributos

- Os atributos definem o conjunto de propriedades de uma classe.
- Nome de atributos são substantivos simples ou frases substantivas.
- Cada atributo deve ter uma definição clara e concisa.
- Cada objeto tem um valor para cada atributo definido na sua classe.
- Para definir atributos, liste as propriedades de uma classe que sejam relevantes para o domínio em questão. Deve-se procurar um compromisso entre objetividade (procurar atender a determinado projeto, com o mínimo custo) e generalidade (permitir a reutilização da classe em outros projetos).

Atributos

- Um atributo é definido por:
  - **nome**: um identificador para o atributo.
  - **tipo**: o tipo do atributo (inteiro, real, caractere etc.)
  - **valor_default**: opcionalmente, pode-se especificar um valor inicial para o atributo.
  - **visibilidade**: opcionalmente, pode-se especificar o quão acessível é um atributo de um objeto a partir de outros objetos. Valores possíveis são:
    - privativo - nenhuma visibilidade externa;
    - público - visibilidade externa total;
    - protegido - visibilidade externa limitada.

Operações

- Uma classe incorpora um conjunto de responsabilidades que definem o comportamento dos objetos na classe.
- As responsabilidades de uma classe são executadas por suas operações.
- Uma operação é um serviço que pode ser requisitado por um objeto para obter um dado comportamento.
- Operações também são chamadas de métodos (Java) ou funções-membro (C++).

Identificação das Operações

- Siga os seguintes procedimentos para identificar operações:
  - Liste os papéis e as responsabilidades de cada classe.
  - Defina o conjunto de operações necessário para satisfazer estas responsabilidades.
  - Garanta que cada operação seja primitiva.
    - Uma operação primitiva é uma operação que pode ser implementada apenas usando o que é intrínseco, interno da classe. Exemplo:
      - Adicione um item a um conjunto -- operação primitiva.
      - Adicione quatro itens a um conjunto -- não primitiva (pode ser implementada com múltiplas chamadas à operação anterior).
  - Garanta a completeza do conjunto de operações.

Diretrizes para Escolha de Operações

- Cada operação deve realizar uma função simples.
- O nome deve refletir o resultado da operação, e não as suas etapas.
  - Ex.: Use **ObterSaldo()** ao invés de **CalcularSaldo()**. Esta última indica que o saldo deve ser calculado, o que é uma decisão de implementação.
- Evite excesso de argumentos de entrada e saída, o que geralmente indica a necessidade de partir as operações em outras mais simples.
- Evite chaves de entrada, que geralmente indicam funções não-primitivas.

Definição de Operações

- Uma operação (método) é definida por:
  - **nome**: um identificador para o método.
  - **tipo**: quando o método tem um valor de retorno, o tipo desse valor.
  - **lista de argumentos**: quando o método recebe parâmetros para sua execução, o tipo e um identificador para cada parâmetro.
  - **visibilidade**: como para atributos, define o quão visível é um método a partir de objetos de outras classes.

Encapsulamento

- Uma classe pode ser visualizada como consistindo de duas partes: interface e implementação.
  - A interface pode ser vista e usada por outros objetos (clientes).
  - A implementação é escondida dos clientes.
- Esconder os detalhes da implementação de um objeto é chamado **encapsulamento**.
- Encapsulamento oferece dois tipos de proteção. Protege:
  - O estado interno de um objeto de ser corrompido por seus clientes.
  - O código cliente de mudanças na implementação dos objetos.

Benefícios do Encapsulamento

- O código cliente pode usar a interface para uma operação.
- O código cliente não pode tirar vantagem da implementação de uma operação.
- A implementação pode mudar, por exemplo para:
  - Corrigir um erro (*bug*).
  - Aumentar a performance.
  - Refletir uma mudança no plano de ação.
- O código cliente não será afetado pelas mudanças na implementação, assim reduzindo o "efeito ondulação" no qual uma correção em uma operação força correções correspondentes numa operação cliente, o qual por sua vez, causa mudanças em um cliente do cliente...
- A manutenção é mais fácil e menos custosa.

Visibilidade e Encapsulamento

- O controle de acesso (visibilidade) é usado para garantir o encapsulamento.
  - A visibilidade é especificada para atributos e operações.
- Atributos e Operações devem ser tão privativos quanto possíveis.
  - **Recomendação: os atributos devem ser sempre privados.**
    - Para acessar os atributos, use operações get() e set().
  - Para um método, basta conhecer sua especificação. Não há necessidade de saber detalhes de sua implementação.

Definição de Classe em Java

```
class Nome_da_Classe
{
  // definição dos atributos
  [visibilidade] tipo nome_do_atributo  [= valor_inicial];

  // ...  outros atributos


  // definição dos métodos
  [visibilidade] tipo_de_retorno nome_do_método
  (lista_de_parâmetros)
  {
        comandos;
  }

  // ...  outros métodos
}
```

Sobrecarga (*Overloading*)

- Através do mecanismo de sobrecarga, dois ou mais métodos de uma classe podem ter o mesmo nome, desde que suas assinaturas sejam diferentes. Tal situação não gera conflito pois o compilador é capaz de detectar qual método deve ser escolhido a partir da análise dos argumentos do método.
- Exemplo 1:
  - métodos da classe *java.lang.Math*
    - esses métodos têm implementações alternativas para tipos de argumentos distintos.
    - Exemplos:
      - static double max (double a, double b)
      - static float max (float a, float b)
      - static int max (int a, int b)

Sobrecarga (*Overloading*)

- Exemplo 2:

```
public class Console {
  public static String readLine() {
        // comandos para leitura de uma string da entrada padrão
  }
  public static String readLine(String prompt) {
        System.out.println(prompt+" ");
        return readLine();
  }
  public static void main(String[] args) {
        String s = Console.readLine("Entre com o valor de s:");
        System.out.print("Entre com o valor de r:");
        String r = Console.readLine();
        // ...
  }
}
```

Criação de Objetos

- Conceito:
  - No paradigma de orientação por objetos, tudo pode ser potencialmente representado como um objeto. Um objeto não é muito diferente de uma variável normal.
  - Quando se cria um objeto, esse objeto adquire um espaço em memória para armazenar seu estado (os valores de seu conjunto de atributos, definidos pela classe) e um conjunto de operações que podem ser aplicadas ao objeto (o conjunto de métodos definidos pela classe).
  - Um programa orientado por objetos é composto por um conjunto de objetos que interagem através de "trocas de mensagens". Na prática, essa troca de mensagem traduz-se na aplicação de métodos a objetos.

Criação de Objetos

- A criação de um objeto é feita pelo operador **new**.

  new NomeDaClasse();

  - Essa expressão é uma invocação do **construtor**, um método especial que toda a classe oferece que indica o que deve ser feito na inicialização de um objeto.
  - A aplicação do operador new ao construtor da classe retorna uma **referência para o objeto**. Para que o objeto possa ser efetivamente manipulado, essa referência deve ser armazenada por quem determinou a criação do objeto:

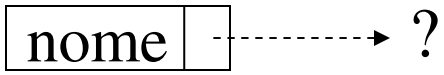
    NomeDaClasse minhaRef = **new** NomeDaClasse();

    - Nesse exemplo, minhaRef é uma variável que guarda uma referência para um objeto do tipo NomeDaClasse.

Manipulação de Objetos

- A declaração de uma variável cujo tipo é uma classe não cria um objeto. Cria-se uma **referência para um objeto** da classe, a qual inicialmente não faz referência a nenhum objeto válido.
- Exemplo:

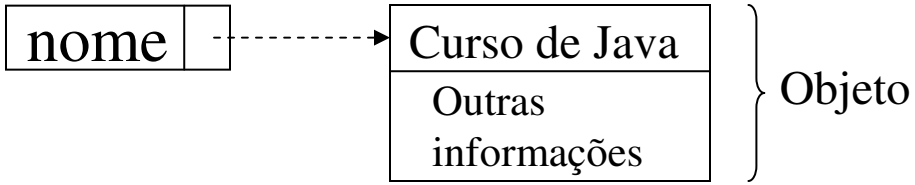
  String nome;



Manipulação de Objetos

- Ao se criar um objeto, usando o operador new, obtém-se uma referência válida, que é armazenada na variável do tipo da classe.
- Exemplo (continuação):

  nome = new String("Curso de Java");

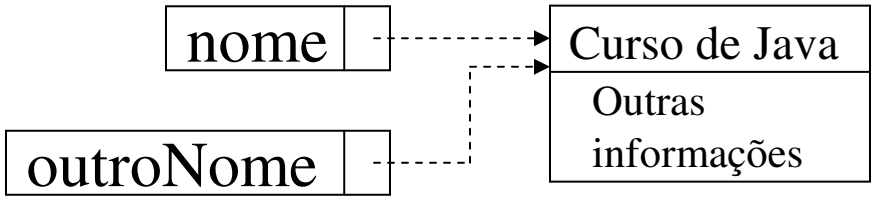


  - a variável ***nome*** armazena uma referência para o objeto cujo conteúdo é "Curso de Java".

Manipulação de Objetos

- A variável ***nome*** do exemplo anterior mantém apenas a referência para o objeto e não o objeto em si.
- O operador = (atribuição) não cria outro objeto. Ele simplesmente atribui a referência para o objeto.
- Exemplo (continuação):

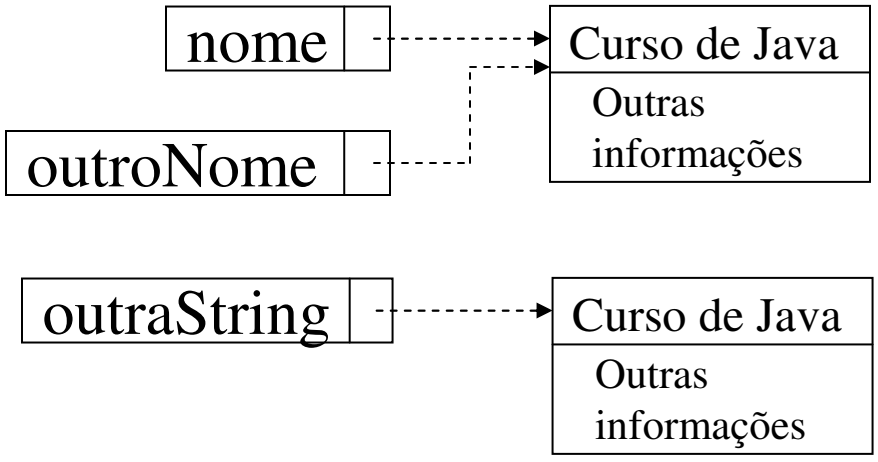
```
String outroNome = nome;
```



Manipulação de Objetos

- Para efetuar uma cópia de um objeto, criando um novo objeto com o mesmo conteúdo de um objeto já existente, é necessário usar o método **clone()** da classe **Object**, da qual todos os objetos descendem.
- Exemplo (continuação):

```
String outraString = nome.clone();
```



Manipulação de Objetos

- O operador == para objetos compara apenas se os dois objetos têm a mesma referência (apontam para o mesmo local).
  - Exemplo (continuação):

```
nome == outroNome          // resulta true
nome == outraString        // resulta false
outroNome == outraString   // resulta false
```

- O método **equals()** da classe **String** compara se o conteúdo de dois objetos são iguais.
  - Exemplo (continuação):

```
nome.equals(outroNome)     // resulta true
outroNome.equals(nome)     // resulta true
nome.equals(outraString)   // resulta true
outraString.equals(nome)   // resulta true
```

Exemplo OO

```
public class PrimeiroProgramaOO {
   private int x;
   public void atribuiValor(int valor) {
      x = valor;
   }
   public int obtemValor() {
      return x;
   }
   public static void main (String[] args) {
      PrimeiroProgramaOO  p = new PrimeiroProgramaOO();
      p.atribuiValor(15);
      System.out.println(p.obtemValor());
   }
}
```

Métodos Construtores

- **Construtor** é um método especial chamado quando um novo objeto é criado pelo operador **new()**.
- Métodos construtores possuem as seguintes características:
  - normalmente são métodos públicos;
  - possuem o mesmo nome da classe;
  - não possuem valor de retorno;
  - podem ter parâmetros;
  - pode haver sobrecarga (*overloading*) de construtores;
  - um construtor *default* sem parâmetros é gerado se nenhum construtor é fornecido pelo implementador da classe.
    - O construtor *default* inicializa todas os atributos da classe, não inicializados explicitamente, com seus valores padrão (números com zero, objetos com nulo e booleanos com falso).

Exemplo

```
public class Caixa {
  private double comprimento, largura,
    altura;
  public Caixa() {
    comprimento = 10;
    largura = 10;
    altura = 10;
  }
  public Caixa(double comp, double larg,
    double alt) {
    comprimento = comp;
    largura = larg;
    altura = alt;
  }
  public void setComprimento(double
    comp) {
    comprimento = comp;  }
  public void setLargura(double larg) {
    largura = larg; }
  public double getLargura() {
    return largura;
  }
  public double volume() {
    return (comprimento * largura * altura);
  }
  public static void main(String[] args) {
    Caixa c1 = new Caixa();
    System.out.println("Volume da caixa 1 = " +
      c1.volume());  // volume = 1000
    Caixa c2 = new Caixa(10,5,3);
    System.out.println("Volume da caixa 2 = " +
      c2.volume());  // volume = 150
    c2.setComprimento(8);
    c2.setLargura(c2.getLargura()-3);
    System.out.println("Novo volume da caixa
      2 = " + c2.volume());  // volume = 48
   }
}
```

this

- Suponha o seguinte código:

  Caixa c1 = new Caixa(); Caixa c2 = new Caixa(10,5,3);

  double v1 = c1.volume(); // v1 = 1000

  double v2 = c2.volume(); // v2 = 150

  - Como é que o método volume sabe de qual objeto ele deve obter o tamanho?
- **this** é uma referência implícita passada para os métodos para referenciar o objeto corrente.
  - usada quando um método precisa se referir ao objeto que o chamou;
  - pode ser usada dentro de qualquer método para se referir ao objeto corrente;
  - pode ser usada sempre que uma referência ao objeto for permitida.

this

- **this** é usada principalmente em dois contextos:
  - diferenciar atributos de objetos de parâmetros ou variáveis locais de mesmo nome.

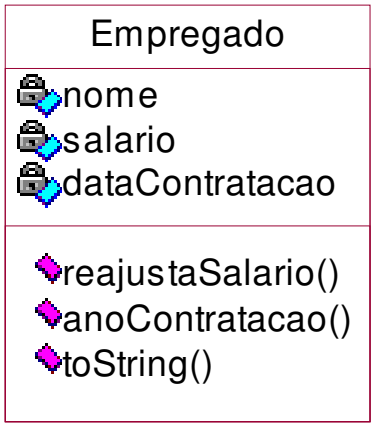
```
public class Caixa {
  private double comprimento, largura, altura;
  public Caixa(double comprimento, double largura, double altura) {
    this.comprimento = comprimento;
    this.largura = largura;
    this.altura = altura;
  }
}
```

  - Acessar o método construtor a partir de outros construtores.

```
public Caixa() {
  this (10, 10, 10)
}
```

Exemplo - Classe Empregado

```
import cursojava.*;

public class Empregado {
  private String nome;
  private double salario;
  private Day dataContratacao;

  public Empregado(String nome, double salario, Day dataContr) {
    this.nome = nome;
    this.salario = salario;
    dataContratacao = dataContr;      }

  public void reajustaSalario(double percentagem) {
    salario *= 1 + percentagem / 100;     }

  public int anoContratacao() {
    return dataContratacao.getYear();    }

  public String toString() {
    return nome + " " + salario + " " + anoContratacao()   }
}
```



Exemplo - Classe EmpregadoTeste

```
import cursojava.*;
public class EmpregadoTeste {
   public static void main(String[] args) {
      Empregado[] emp = new Empregado[3];

      emp[0] = new Empregado("Jose Silva", 3200, new Day(1998,10,1));
      emp[1] = new Empregado("Maria Oliveira", 1100, new Day(1997,12,15));
      emp[2] = new Empregado("Joao Barros", 300, new Day(2003,3,15));

      for (int i = 0; i < 3; i++) {
            emp[i].reajustaSalario(5);
            System.out.println(emp[i].toString());
      }
   }
}
```

Finalizadores

- Java não possui métodos destrutores
  - A coleta de lixo é automática.
- Não há necessidade de explicitamente liberar áreas de memória alocadas para um objeto
  - quando não há mais referências para uma área, ela pode ser liberada.
  - a liberação ocorre esporadicamente durante a execução, podendo até não ocorrer. A coleta de lixo é não-determinística.

Finalizadores

- Método finalize()
  - usado quando o objeto precisar realizar alguma ação antes de ser destruído.
  - antes de liberar o objeto, o método **finalize()** será chamado.

```
protected void finalize() {
  // corpo do método
}
```

  - como a coleta de lixo é não-determinística não se tem certeza quando o método finalize() será executado.

Finalizadores

- O programador não tem como atuar explicitamente na coleta de lixo. No entanto, o programador pode:
  - remover explicitamente a referência a um objeto para sinalizar ao coletor de lixo que o objeto não mais é necessário e pode ser removido;
  - sugerir (mas não forçar) que o sistema execute o coletor de lixo (*garbage collector*) através da invocação ao método **gc()** da classe java.lang.System.

    public static void gc()
  - sugerir que o sistema execute os métodos finalize() de objetos descartados através da invocação ao método **runFinalization()** da classe java.lang.System.

    public static void runFinalization()

Atributos Estáticos

- Quando se cria objetos de uma determinada classe, cada objeto tem sua cópia separada dos atributos definidos para a classe.
- Em situações em que é necessário que todos os objetos de uma classe compartilhem um mesmo atributo, semelhante ao que ocorre com variáveis globais em linguagens de programação tradicional, deve-se definir o atributo como estático. Neste caso, o atributo estático funciona como uma variável global da classe.
- Um atributo estático é declarado usando-se a palavra-chave **static** antes do nome do atributo.

Atributos Estáticos

```
public class QualStatic {
    private static int contEstatico = 0;
     private int contNaoEstatico = 0;

    public QualStatic(){
        contEstatico++;
         contNaoEstatico++;  }

    public void incContEstatico() {
       contEstatico++;   }

     public void incContNaoEstatico() {
        contNaoEstatico++;  }

     public int leContEstatico() {
         return contEstatico;  }

      public int leContNaoEstatico() {
         return contNaoEstatico;   }
```

Atributos Estáticos

```
  public static void main (String[] args){
    QualStatic obj1 = new QualStatic();
    System.out.println("Cont Estatico obj1 = " + obj1.leContEstatico());          // 1
    System.out.println("Cont Nao Estatico obj1 = " + obj1.leContNaoEstatico());  // 1
    QualStatic obj2 = new QualStatic();
    System.out.println("Cont Estatico obj1 = " + obj1.leContEstatico());          // 2
    System.out.println("Cont Nao Estatico obj1 = " + obj1.leContNaoEstatico());  // 1
    System.out.println("Cont Estatico obj2 = " + obj2.leContEstatico());          // 2
    System.out.println("Cont Nao Estatico obj2 = " + obj2.leContNaoEstatico()); // 1
    obj1.incContEstatico();
    obj1.incContNaoEstatico();
    System.out.println("Cont Estatico obj1 = " + obj1.leContEstatico());          // 3
    System.out.println("Cont Nao Estatico obj1 = " + obj1.leContNaoEstatico());  // 2
    System.out.println("Cont Estatico obj2 = " + obj2.leContEstatico());          // 3
    System.out.println("Cont Nao Estatico obj2 = " + obj2.leContNaoEstatico()); // 1
  }
}
```

Métodos Estáticos

- Assim como atributos estáticos, métodos estáticos não precisam de um objeto para serem ativados. Pode-se invocar diretamente um método estático sem necessidade de se criar um objeto.
- Por conseqüência, métodos estáticos:
  - só podem acessar dados estáticos;
  - não podem se referir a **this**;
  - só podem chamar outros métodos estáticos.
- Declaração
  - precedido da palavra-chave **static**.
  - Exemplo:

    public **static** double potencia(double x)

Métodos Estáticos

- Ativação:

  NomeDaClasse.método()
- Exemplos:
  - todos os métodos da classe java.lang.Math.

    double cosseno = Math.cos(60);

    double valor = Math.sqrt(144);
  - método **main**:

    public **static** void main (String[] args)
    - pode ser chamado antes de qualquer objeto existir.
    - geralmente instancia um objeto aplicação para acessar atributos não estáticos.

Reusabilidade

- Uma das principais vantagens de orientação por objetos é a facilidade de reuso de código.
  - Pode-se criar novas classes a partir de classes existentes que foram criadas e depuradas por terceiros.
- **Reusabilidade** é a habilidade de elementos de software servirem a diferentes aplicações.
- Outra característica interessante de orientação por objetos é a **extensibilidade**, que é a facilidade de adaptar produtos de software a mudanças de especificação.
- O reuso de código pode se dar por:
  - Composição;
  - Herança.

Reusabilidade

- Composição
  - Objetos de classes já existentes são criados dentro de uma nova classe.
  - A nova classe é composta de objetos de classes existentes.
  - Há um reuso da funcionalidade do código e não da sua forma.
- Herança
  - Permite a criação de novas classes com propriedades adicionais a uma classe já existente.
  - A nova classe herda as funcionalidades da classe já existente e adiciona novas funcionalidades.
  - Há um reuso da forma de uma classe.

Composição

- Exemplo:
  - Implementação de uma classe que descreve uma entidade Quadrado.
  - Quadrado é composto de pontos.
  - Ponto é uma entidade.
  - Logo, quadrado é construída através da composição de quatro entidades Ponto.
- A composição implementa o relacionamento TEM-UM (HAS-A).
  - Quadrado TEM-UM (*tem 4*) Ponto(*s*).
  - Quadrado é composto de Pontos.

Composição

```
public class Ponto {
   private double x;  // coordenada x
   private double y;  // coordenada y
   public Ponto (double x, double y) {
      this.x = x;    this.y = y;
   }
   . . .
}
```


```
public class Quadrado {
   private Ponto p1, p2, p3, p4;
   public Quadrado (Ponto p1, Ponto p2, Ponto p3, Ponto p4) {
      . . .
   }
   . . .
}
```

Composição

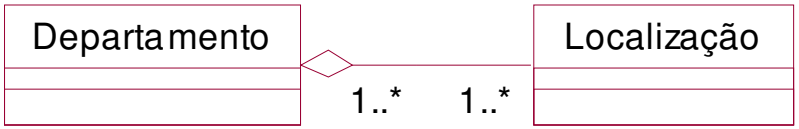
- Composição denota relacionamentos todo/parte, onde o todo é constituído de partes.
  - Exemplo: relacionamento entre Banco e Agência.

Banco é o todo e os objetos Agência são as partes. Os objetos Agência não podem existir independentemente do objeto Banco. Se o banco for excluído (encerrar suas atividades) as agências também serão excluídas. O inverso não é necessariamente verdadeiro.

- **Composição** é também chamada de **Agregação por Valor**.
  - Representada, em UML, por um losango fechado do lado todo.

Agregação

- A **Agregação por Referência** ou simplesmente **Agregação** também modela relacionamentos todo/parte (tem-um). Diferentemente da composição, os objetos em uma agregação podem existir independentemente uns dos outros.
  - Denota uma composição menos rigorosa.
  - Exemplo: relacionamento entre Departamento e Localização.

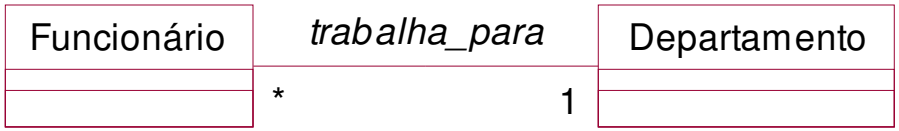


Um departamento possui várias localizações, mas uma localização pode pertencer a mais de um departamento e portanto não é excluída se um departamento for excluído.

  - Representada por um losango aberto do lado todo.

Associação

- A **Associação** indica que um objeto contém ou está conectado a outro objeto. Um objeto usa outro objeto.
  - Exemplo: relacionamento entre Funcionário e Departamento, indicando que um funcionário trabalha para um departamento e que em um departamento trabalham vários funcionários.



  Os objetos Funcionário e Departamento existem independentemente um do outro e não denotam todo/parte.
- Agregação e Composição são subtipos de Associação que ajudam a refinar mais os modelos.

Implementação dos Relacionamentos

- Os relacionamentos de Associação, e seus subtipos Agregação e Composição, são implementados da mesma forma: uma classe declara um atributo do tipo da outra classe.
  - O lado '1' do relacionamento é declarado como um atributo simples e o lado 'muitos' é declarado como uma coleção (array, Vector, ...).
  - Exemplo: Funcionário e Departamento

```
public class Funcionário {
   private Departamento depto;
   . . . }
```

```
public class Departamento {
   private Funcionário[] funcs;
   . . . }
```

  - Exemplo: Banco e Agência

```
public class Banco {
   private Agência[] agencias;
   . . . }
```

```
public class Agência {
   private Banco bco;
   . . . }
```

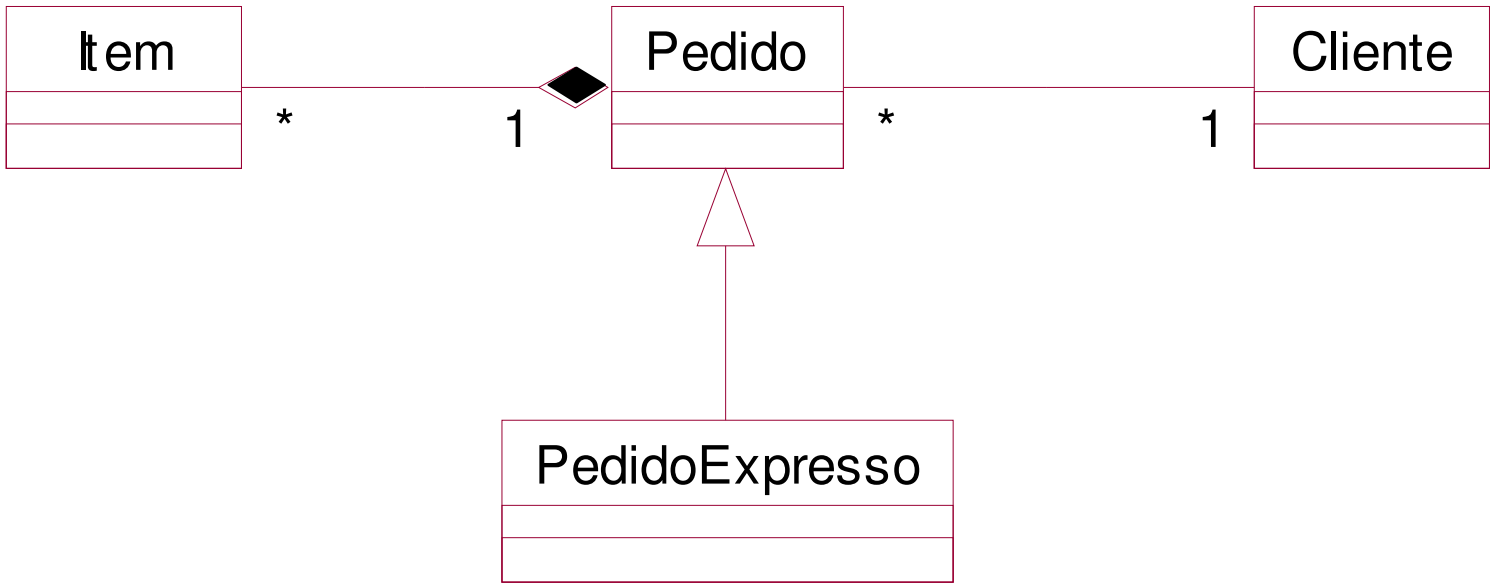
Herança

- Herança denota especialização.
- Herança permite a criação de novos tipos com propriedades adicionais ao tipo original.
- Em geral, se uma classe B estender (herdar) uma classe A, a classe B vai herdar métodos e atributos da classe A e implementar alguns recursos a mais.
- Herança implementa o relacionamento É-UM (IS-A).
- Exemplo: classes Pedido e PedidoExpresso
  - PedidoExpresso É-UM Pedido
  - PedidoExpresso é uma especialização de Pedido. Estende Pedido.
  - PedidoExpresso herda todas as definições (atributos e métodos) já implementadas na classe Pedido.

Herança

- Exemplos:
  - Classes Mamífero e Cachorro
    - Cachorro É-UM Mamífero
  - Classes Veículo e Carro
    - Carro É-UM Veículo
  - Classes Empregado e Gerente
    - Gerente É-UM Empregado
- A classe herdeira é chamada de subclasse (ou classe derivada) e a classe herdada é chamada de superclasse (ou classe base).
  - Carro é subclasse de Veículo
  - Veículo é superclasse de Carro
  - Carro é a classe derivada da classe base Veículo

Herança, Composição e Associação



- Herança: PedidoExpresso É-UM Pedido
- Composição: Pedido TEM-UM (tem vários) Item
- Associação: Pedido está associado a um Cliente

Herança

- Sintaxe:
  - Palavra-chave **extends**.
  - Utiliza-se na definição da subclasse a palavra-chave **extends** seguida pelo nome da super-classe.

```
class SuperClasse {
...
}
class SubClasse extends SuperClasse {
...
}
```

Classe Object

- **Object** é uma classe em Java da qual todas as outras derivam.
- Quando uma classe é criada e não há nenhuma referência a sua superclasse, implicitamente a classe criada é derivada diretamente da classe **Object**.
  - Todos os objetos podem invocar os métodos da classe Object.
- Alguns métodos da classe Object:

  boolean equals (Object obj)
    - Testa se os objetos são iguais (apontam para o mesmo local).

  Class getClass ()
    - Retorna a classe do objeto.

  Object clone ()
    - copia o conteúdo de um objeto para outro ("clone raso").

Construção de Objetos Derivados

- Processo de execução dos métodos construtores envolvendo herança:
  - Durante a construção de um objeto de uma classe derivada, o construtor de sua superclasse é executado (implicitamente ou explicitamente) antes de executar o corpo de seu construtor.
  - Assim, ao se construir um objeto de uma classe derivada, o método construtor da superclasse será inicialmente invocado. Este por sua vez, invocará o construtor de sua superclasse, até que o construtor da classe raiz de toda a hierarquia de objetos -- a classe **Object** -- seja invocado. Como **Object** não tem uma superclasse, seu construtor é executado e a execução retorna para o construtor de sua

Construção de Objetos Derivados

classe derivada. Então executa-se o restante do construtor de sua classe derivada. E a execução retorna para o construtor de sua classe derivada e assim sucessivamente, até que finalmente o restante do construtor da classe para a qual foi solicitada a criação de um objeto seja executado.

- Construtores da superclasse podem ser explicitamente invocados usando a palavra-chave **super**.

super

- **super** é usado para invocar explicitamente o construtor da superclasse imediatamente superior.
- Exemplo:

```
class Ponto2D {
   private double x, y;
   public Ponto2D (double x, double y) {
        this.x = x;   this.y = y }
}
class Ponto3D extends Ponto2D {
   private double z;
   public Ponto3D (double x, double y, double z) {
        super (x, y);
        this.z = z;   }
}
```

super

- A invocação do método **super()**, se presente, deve estar na primeira linha.
- Se o método **super()** não for usado, implicitamente o compilador faz a invocação do construtor **super()** *default* (sem argumentos) para cada construtor definido.
  - É sempre interessante ter o construtor *default* definido.
  - A invocação direta pode ser interessante quando se deseja invocar algum construtor que não o *default*, como no exemplo anterior.
- **Super** é também usado para acessar membros (atributos e métodos) da superclasse imediatamente superior.

  super.nomeDoAtributo ou super.nomeDoMétodo()

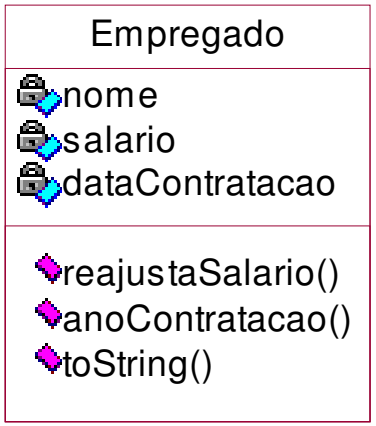
Exemplo - Classes Hora e HoraLocal

```
public class Hora {
   private int horas, minutos,
    segundos;
   public Hora (int hor, int min, int
    seg) {
     horas = hor;
     minutos = min;
     segundos = seg;
   }
   public void exibeHorario() {
     System.out.println(horas + ":" +
    minutos + ":" + segundos);
   }
}
```

```
public class HoraLocal extends Hora {
   private String local;
   public HoraLocal (int hor, int min,
    int seg, String l) {
     super(hor, min, seg);
     local = l;
   }
   public void exibeHorario() {
     super.exibeHorario();
     System.out.println(" "+local);
   }
   public static void main (String[]
    args) {
     HoraLocal hl = new
          HoraLocal(21,30,22,"Brasilia");
     hl.exibeHorario();
   } }
```

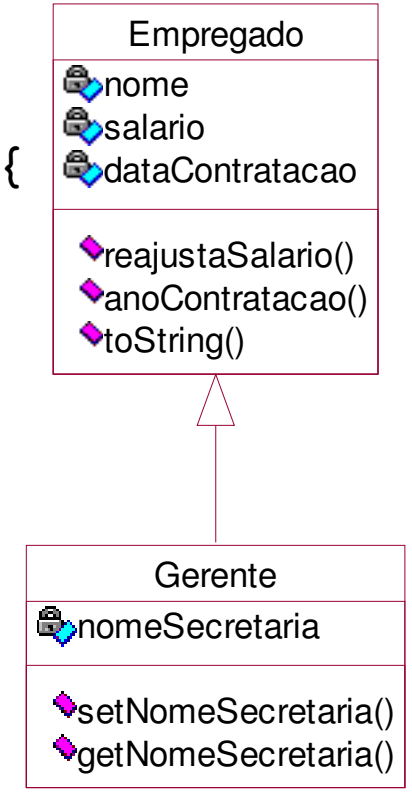
Exemplo - Classe Empregado

```
import cursojava.*;

public class Empregado {
  private String nome;
  private double salario;
  private Day dataContratacao;

  public Empregado(String nome, double salario, Day dataContr) {
    this.nome = nome;
    this.salario = salario;
    dataContratacao = dataContr;      }

  public void reajustaSalario(double percentagem) {
    salario *= 1 + percentagem / 100;     }

  public int anoContratacao() {
    return dataContratacao.getYear();    }

  public String toString() {
    return nome + " " + salario + " " + anoContratacao()   }
}
```



Exemplo - Classe Gerente

```
import cursojava.*;

public class Gerente extends Empregado {
  private String nomeSecretaria;

  public Gerente(String nome, double salario, Day dataContr) {
    super(nome, salario, dataContr);
    nomeSecretaria = "";    }

  public void reajustaSalario(double percentagem) {
    // adiciona 1/2% de bônus para cada ano de serviço
    Day hoje = new Day();
    double bonus = 0.5 * (hoje.getYear() - anoContratacao());
    super.reajustaSalario(percentagem + bonus);    }

  public void setNomeSecretaria(String nome) {
    nomeSecretaria = nome;     }

  public String getNomeSecretaria() {
    return nomeSecretaria;     }
}
```



Restrições de Acesso

- Restrições de acesso especificam o quão acessível é um membro (atributo ou método) de um objeto a partir de outros objetos.
- Existem 4 formas de restrição de acesso em Java:
  - **public**: o membro é visível (pode ser acessado) a todos os objetos;
  - **protected**: o membro é visível dentro da própria classe, dentro de suas subclasses e dentro do pacote ao qual a classe pertence;
  - **package**: o membro é visível dentro da própria classe e dentro do pacote ao qual a classe pertence. Não se usa a palavra **package**: restrição ***default***;
  - **private**: o membro é visível somente dentro da própria classe.

Restrições de Acesso

Especificador	Classe	Subclasse	Pacote	Mundo
public	X	X	X	X
protected	X	X	X	
package	X		X	
private	X			

Dicas:

- sempre mantenha os atributos privados.
- crie métodos acessores (get) e modificadores (set) para os atributos visíveis fora da classe
  - nem todos os atributos necessitam acessores e modificadores.

Mais sobre Herança

- SubClasse pode ser usada sempre que a SuperClasse é esperada (contrário não vale).
  - A subclasse possui todos os métodos e atributos da superclasse. Subclasse É-UM superclasse.

```
. . .
SuperClasse  sp;
SubClasse  sb = new SubClasse();
sp = sb;       // correto
SuperClasse sp2 = new SubClasse();  // correto
sb = sp2;     // incorreto
```

- Todos os métodos na SuperClasse são herdados sem modificação na SubClasse.
- Todos os atributos que formam a SuperClasse formam também a SubClasse.

Atributos Herdados

- Subclasses podem adicionar novos atributos.
- Subclasses não podem remover atributos herdados.
- Subclasses podem criar atributos com os mesmos nomes dos atributos originais (herdados). Neste caso, os atributos originais são escondidos (*shadow*).

Métodos Herdados

- Subclasses podem adicionar novos métodos.
- Subclasses não podem remover métodos herdados.
- Subclasses podem redefinir (*override*) métodos originais (herdados), usando a mesma assinatura.
- Subclasses não podem reduzir o nível de acesso de métodos herdados. O contrário é válido.
  - Exemplo:
    - de *public* para *private*. // incorreto
    - de *public* para *protected* // incorreto
    - de *protected* para *private* // incorreto
    - de *private* para *public* // correto
    - . . .

Override X Shadowing

```
class SuperClasse {
   int i = 2;
   int getI() { return i; } }

class SubClasse extends SuperClasse {
   int i = 1;
   int getI() { return i; } }

public class OverrideShadowing {
   public static void main (String[] args) {
      SubClasse b = new SubClasse();
      System.out.println(b.i);            // 1
      System.out.println(b.getI());       // 1
      SuperClasse a = b;
      System.out.println(a.i);            // 2
      System.out.println(a.getI()); }  // 1
}
```

final

- **final** é uma palavra chave em Java que indica que um atributo, um método ou uma classe não podem ser modificados ao longo do restante da hierarquia de descendentes.
- Atributos **final** são usados para definir constantes.
  - Apenas valores dos tipos primitivos podem ser usados para definir constantes.
  - O valor é definido na declaração ou nos construtores da classe, e não pode ser alterado.
  - Exemplo: **final** double PI = 3.14;
  - Para objetos e arranjos, apenas a referência é constante (o conteúdo do objeto ou arranjo pode ser modificado).

final

- Na lista de parâmetros de um método, **final** indica que o parâmetro não pode ser modificado.

  public void exemplo (**final** int par1, double par2) { ... }

  par1 não pode ser modificado dentro do método.

- Para método, **final** indica que o método não pode ser redefinido em classes derivadas.

  final public void exemplo () { ... }

  método exemplo não pode ser redefinido em classes herdeiras.

- Para classe, **final** indica que a classe não pode ser derivada.

  final public class Exemplo { ... }

  public class subExemplo extends Exemplo { ... } //ERRO

  classe Exemplo não pode ser derivada.

Polimorfismo

- **Polimorfismo** significa muitas formas. Em termos de programação, significa que um único nome de classe ou de método pode ser usado para representar comportamentos diferentes dentro de uma hierarquia de classes.
- A decisão sobre qual comportamento utilizar é tomada em tempo de execução.
- As linguagens que suportam polimorfismo são chamadas de *linguagens polimórficas*. As que não suportam são chamadas de *linguagens monomórficas*. Nestas, cada nome é vinculado estaticamente ao seu código.

Polimorfismo

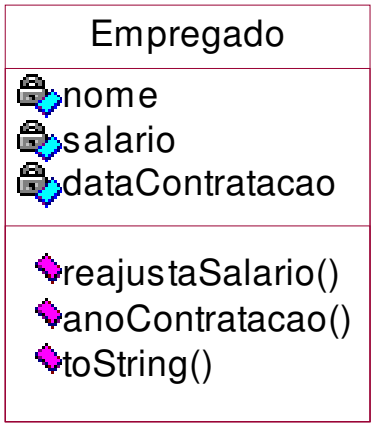
- Clientes podem ser implementados genericamente para chamar uma operação de um objeto sem saber o tipo do objeto.
- Se são criados novos objetos que suportam uma mesma operação, o cliente não precisa ser modificado para suportar o novo objeto.
- O polimorfismo permite que os clientes manipulem objetos em termos de sua superclasse comum.
- O polimorfismo torna a programação orientada por objetos eficaz, permitindo a escrita de código genérico, fácil de manter e de estender.

Ligação Tardia (*late binding*)

- Quando o método a ser invocado é definido durante a compilação do programa, o mecanismo de **ligação prematura** (*early binding*) é utilizado.
- Para a utilização de polimorfismo, a linguagem de programação orientada por objetos deve suportar o conceito de **ligação tardia** (*late binding*), onde a definição do método que será efetivamente invocado só ocorre durante a execução do programa. O mecanismo de ligação tardia também é conhecido pelos termos ligação dinâmica (*dynamic binding*) ou ligação em tempo de execução (*run-time binding*).

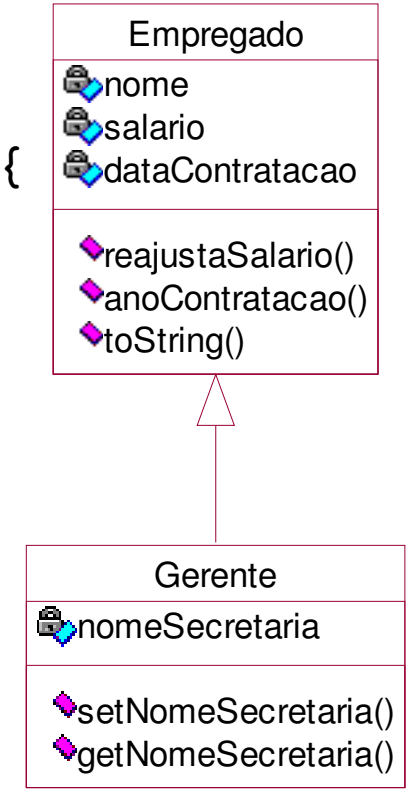
Exemplo - Classe Empregado

```
import cursojava.*;

public class Empregado {
   private String nome;
   private double salario;
   private Day dataContratacao;

   public Empregado(String nome, double salario, Day dataContr) {
      this.nome = nome;
      this.salario = salario;
      dataContratacao = dataContr;      }

   public void reajustaSalario(double percentagem) {
      salario *= 1 + percentagem / 100;     }

   public int anoContratacao() {
      return dataContratacao.getYear();    }

   public String toString() {
      return nome + " " + salario + " " + anoContratacao()   }
}
```



Exemplo - Classe Gerente

```
import cursojava.*;

public class Gerente extends Empregado {
  private String nomeSecretaria;

  public Gerente(String nome, double salario, Day dataContr) {
    super(nome, salario, dataContr);
    nomeSecretaria = "";    }

  public void reajustaSalario(double percentagem) {
    // adiciona 1/2% de bônus para cada ano de serviço
    Day hoje = new Day();
    double bonus = 0.5 * (hoje.getYear() - anoContratacao());
    super.reajustaSalario(percentagem + bonus);    }

  public void setNomeSecretaria(String nome) {
    nomeSecretaria = nome;     }

  public String getNomeSecretaria() {
    return nomeSecretaria;     }
}
```



Exemplo - Classe GerenteTeste

```
import cursojava.*;

public class GerenteTeste {
   public static void main(String[] args) {
      Gerente chefe = new Gerente("Pedro Paulo", 7500, new Day(1998,12,15));
      chefe.setNomeSecretaria("Patricia Moreira");
      Empregado[] emps = new Empregado[4];
      emps[0] = chefe;
      emps[1] = new Empregado("Jose Silva", 3200, new Day(1998,10,1));
      emps[2] = new Empregado("Maria Oliveira", 1100, new Day(1997,12,15));
      emps[3] = new Empregado("Joao Barros", 300, new Day(2003,3,15));
       for (int i = 0; i < 4; i++) {
          emps[i].reajustaSalario(5);
          System.out.println(emps[i].toString());
      }
      System.out.println("A Secretaria do Departamento e' " +
                                 chefe.getNomeSecretaria());     }
}
```

Polimorfismo

- Polimorfismo no exemplo anterior:
  - O objeto emps é do tipo Empregado[], portanto uma posição do vetor pode receber um objeto do tipo Empregado ou do tipo Gerente (ou do tipo de qualquer subclasse de Empregado que venha a existir).
  - O método reajustaSalario() existe tanto na classe Empregado quanto na classe Gerente. No código

```
for (int i = 0; i < 4; i++) {
    emps[i].reajustaSalario(5);
```

  o compilador executa o código de reajustaSalario() da classe Empregado se o objeto da posição emps[i] for do tipo Empregado ou o código da classe Gerente se o objeto emps[i] for do tipo Gerente.

Polimorfismo

- Se não houvesse polimorfismo:
  - O vetor emps só receberia objetos do tipo Empregado e portanto em tempo de compilação já se saberia qual código de reajustaSalario() seria executado. A classe GerenteTeste mudaria para:

```
. . .
emps[0] = new Empregado("Jose Silva", 3200, new Day(1998,10,1));
emps[1] = new Empregado("Maria Oliveira", 1100, new Day(1997,12,15));
emps[2] = new Empregado("Joao Barros", 300, new Day(2003,3,15));
for (int i = 0; i < 3; i++) {
    emps[i].reajustaSalario(5);   // executa código da classe Empregado
   System.out.println(emps[i].toString()); }
chefe.reajustaSalario(5);       // executa código da classe Gerente
System.out.println(chefe.toString());
. . .
```

Polimorfismo

- Considere o seguinte trecho de código:

```
public class Empresa {
    private static final int MAXEMPS = 1000;
    private Empregado[] emps = new Empregado[MAXEMPS];
    private numEmps = 0;  // número de empregados na lista
    . . .
    public void adicionaEmpregado (Empregado emp) {
       if (numEmps = MAXEMPS) return;  // lista cheia
       emps[numEmps] = emp;
       numEmps++;
    }
    public void reajustaSalarios(double perc) {
       for (int i=0; i<numEmps; i++)
           emps[i].reajustaSalario(perc);
    }
    . . .
}
```

Polimorfismo

- Com o uso de polimorfismo, o código da classe Empresa é genérico:
  - O método adicionaEmpregado() pode receber como parâmetro tanto um objeto do tipo Empregado quanto do tipo Gerente e adicioná-lo à lista emps que é do tipo Empregado[].
  - O método reajustaSalarios() chama o método reajustaSalario() da classe Empregado ou da classe Gerente dependendo do tipo do objeto na posição emps[i].
  - Se for criada uma nova subclasse de Empregado, por exemplo, a classe Diretor, o código da classe Empresa não precisaria ser modificado para inserir e reajustar salários dos objetos do tipo Diretor.

Polimorfismo

- Se não houvesse polimorfismo, a classe Empresa seria algo do tipo:

```
public class Empresa {
    private Empregado[] emps = new Empregado[1000];
    private Gerente[] gers = new Gerente[50];
    . . .
    public void adicionaEmpregado (Empregado emp) { . . . }
    public void adicionaGerente (Gerente ger) { . . . }
    public void reajustaSalariosEmps(double perc) { . . . }
    public void reajustaSalariosGers(double perc) { . . . }
}
```

  - Se fosse criada a classe Diretor, a classe Empresa também deveria ser alterada, incluindo uma lista de diretores e métodos para adicionar e reajustar salários de diretores.

Polimorfismo

- O tipo de polimorfismo mostrado nos exemplos anteriores é conhecido como **Polimorfismo de Inclusão** ou **Polimorfismo Puro**.
- A **Sobrecarga** de métodos também é um tipo de polimorfismo (**Polimorfismo ad-hoc**). Ela permite que vários métodos tenham o mesmo nome, desde que o tipo e/ou o número de parâmetros sejam diferentes. Exemplo: os métodos da classe Math:
  - static int max (int a, int b)
  - static long max (long a, long b)
  - static float max (float a, float b)
  - static double max (double a, double b)

Polimorfismo

- Se não houvesse sobrecarga cada método teria de ter um nome diferente, mesmo tendo o mesmo significado. Exemplo:
  - static int maxInt (int a, int b)
  - static long maxLong (long a, long b)
  - static float maxFloat (float a, float b)
  - static double maxDouble (double a, double b)
- Com polimorfismo, pode-se chamar simplesmente max() e passar os parâmetros. O compilador chamará o método correto internamente.

Conversão de Tipo Explícita (*Cast*)

- Considere o seguinte trecho de código do exemplo da classe GerenteTeste:

```
Gerente chefe = new Gerente("Pedro Paulo", 7500, new Day(1998,12,15));
chefe.setNomeSecretaria("Patricia Moreira");
Empregado[] emps = new Empregado[4];
emps[0] = chefe;
emps[1] = new Empregado("Jose Silva", 3200, new Day(1998,10,1));
. . .
```

- Neste exemplo, o objeto chefe é do tipo Gerente (subclasse) e o objeto emps é do tipo Empregado[] (superclasse). Portanto a atribuição abaixo é legal:

```
emps[0] = chefe        // atribuição legal
```

Porém, o verdadeiro tipo está sendo subestimado.

Conversão de Tipo Explícita (*Cast*)

- Ao objeto emps[i] somente se aplicam os métodos da classe Empregado. Assim,

```
emps[0].setNomeSecretaria("Patricia Ferreira");     // erro
```

  resulta em erro, mesmo sendo o objeto emps[0] do tipo Gerente.

- Para converter novamente o objeto para sua classe original, usa-se a conversão de tipo explícita (*cast*), semelhante ao que foi feito com a conversão de tipos primitivos:

```
Gerente chefe2 = (Gerente) emps[0];     //  legal
((Gerente) emps[0]).setNomeSecretaria("Patricia Ferreira");  //  legal
```

Conversão de Tipo Explícita (*Cast*)

- Converter um objeto de uma superclasse para uma subclasse pode resultar em erro:

```
Gerente chefe3 = (Gerente) emps[1];   // erro em tempo de execução
```

emps[1] não é um Gerente!

- Pode-se fazer uma conversão de tipo explícita somente dentro de uma hierarquia de classes.

```
Ponto2D  p1 = (Ponto2D) emps[1];      //  erro de compilação
```

resulta em erro de compilação, pois Ponto2D não é subclasse de Empregado.

Conversão de Tipo Explícita (*Cast*)

- Para descobrir se um objeto é ou não instância de uma classe, usa-se o operador

  instanceof

Exemplo:

```
for (int i = 0; i < 4; i++) {
    emps[i].reajustaSalario(5);
    System.out.println(emps[i].toString());
    if (emps[i] instanceof Gerente)
        System.out.println("A Secretaria do Gerente e' " +
                    ((Gerente)emps[i]).getNomeSecretaria());
}
```

Conversão de Tipo Explícita (*Cast*)

- **Advertência**: não use conversão de tipo explícita de forma indiscriminada! Sempre que possível, coloque os métodos genéricos nas superclasses e use polimorfismo.
  - No exemplo anterior, a única razão para se fazer um *cast* é usar um método que seja único para os gerentes como getNomeSecretaria(). Para os demais, como reajustaSalario() e toString(), deve-se usar o polimorfismo.
- As conversões de tipo explícitas são usadas habitualmente com contêineres como a classe ***Vector***.
  - Ao ler o valor de um contêiner, seu tipo somente é conhecido como o genérico *Object* e torna-se necessário usar uma conversão de tipo explícita para casá-lo com o tipo de objeto que foi colocado no recipiente.

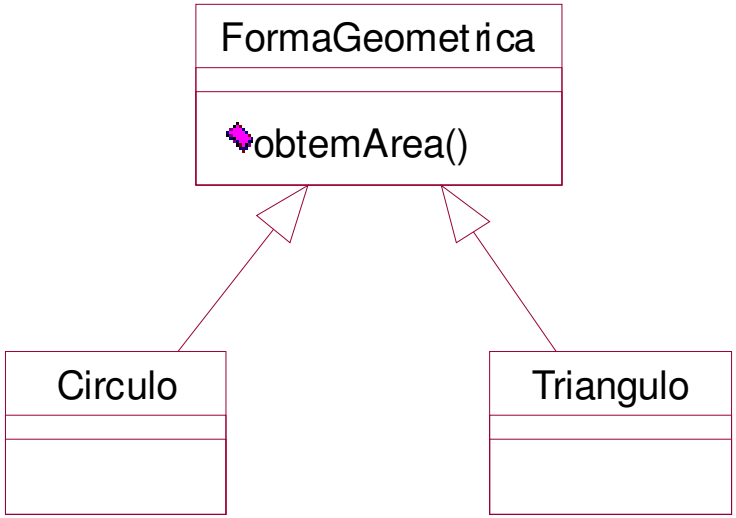
Classe Abstrata

- **Classe Abstrata** é uma classe que não pode ser instanciada, ou seja, não se pode criar objetos diretamente de uma classe abstrata.

```
abstract public class Exemplo {
    public static void main (String[] args) {
        Exemplo obj = new Exemplo();     //  ERRO
    }
}
```

- **abstract** é a palavra-chave em Java que indica que uma classe ou um método não tem definição concreta.
- Classes abstratas correspondem a especificações genéricas, que deverão ser concretizadas em classes derivadas.

Classe Abstrata



- Classes abstratas devem possuir pelo menos uma subclasse para serem utilizáveis.
- No exemplo acima, FormaGeometrica é uma classe abstrata. Todos os objetos ou são círculos ou são triângulos, não existem instâncias diretas de FormaGeometrica.

Classe Abstrata

- Um **método abstrato** cria apenas uma declaração de um método que deverá ser implementado em uma classe derivada. É um método que só faz sentido para as subclasses.
  - Exemplo, o método obtemArea() da classe FormaGeometrica é um método abstrato. Só faz sentido obter área de uma forma geométrica concreta como um círculo ou um triângulo.
- Classe com pelo menos um método abstrato é uma classe abstrata.
- Uma subclasse de uma classe abstrata permanece abstrata, mesmo não sendo declarada explicitamente, até que redefina e implemente todos os métodos abstratos.

Classe Abstrata

```
abstract public class FormaGeometrica {
    abstract public double obtemArea();
}
```

```
public class Circulo extends FormaGeometrica {
  private double raio;
  . . .
  public double obtemArea() {
    return Math.PI * Math.pow(raio,2);
}}
```

```
public class Triangulo extends FormaGeometrica {
  private double lado1, lado2, lado3;
  . . .
  public double obtemArea() {
    double sp = (lado1 + lado2 + lado3) / 2;
    return Math.sqrt(sp*(sp-lado1)*(sp-lado2)*(sp-lado3));
}}
```

Classe Abstrata e Polimorfismo

- Um método abstrato é uma promessa de implementação que será cumprida pelas subclasses. Outras classes podem confiar nessa promessa e implementar códigos genéricos usando o conceito de polimorfismo.

```
public class GeoPoli {
    private FormaGeometrica[] fgs;
    . . .
    public void adicionaFormaGeo (FormaGeometrica fg) { . . . }
    public void imprimeAreas() {
        for (int i=0; i<numFigs; i++)
            System.out.println (fgs[i].obtemArea());
    }
}
```

Regras de Projetos para Herança

- Operações e Atributos comuns pertencem à superclasse.
- Use herança para modelar relacionamento "É-UM" somente.
- Não use herança a menos que todos os métodos herdados façam sentido.
- Use polimorfismo ao invés de informação de tipo.

Classe Abstrata - Exemplo

```
public abstract class Sortable {
    /** @return 1 se o objeto corrente for maior do b, 0 se for igual e -1 se for menor*/
   public abstract int compareTo (Sortable b);
}
```

```
public class ArrayAlg {
    public static void shellSort(Sortable[] a) {
        int n = a.length;
        int incr = n / 2;
        while (incr >= 1) {
            for (int i = incr; i < n; i++) {
                Sortable temp = a[i];
                int j = i;
                while (j >= incr && temp.compareTo(a[j - incr]) < 0)  {
                    a[j] = a[j - incr];
                    j -= incr; }
                a[j] = temp; }
            incr /= 2; }
}}
```

Classe Abstrata - Exemplo

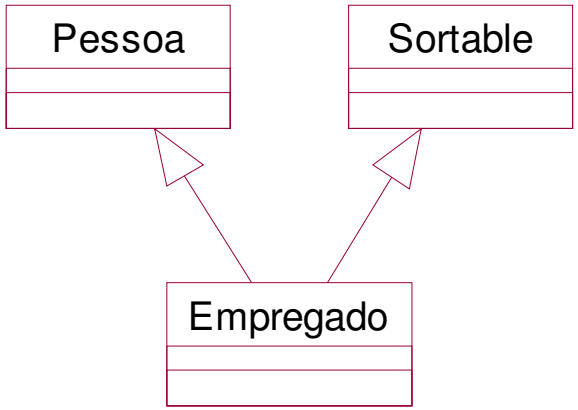
```
import cursojava.*;
public class Empregado extends Sortable {
    private String nome;
    private double salario;
    private Day dataContratacao;
    public Empregado (String n, double s, Day d) {
        nome = n;
        salario = s;
        dataContratacao = d;
    }
    . . .
    public int compareTo (Sortable b) {
         Empregado emp = (Empregado) b;
         if (salario < emp.salario) return -1;
         if (salario > emp.salario) return 1;
         return 0;
    }
}
```

Classe Abstrata - Exemplo

```
import cursojava.*;
public class EmpregadoSortTeste {
   public static void main(String[] args) {
       Empregado[] emps = new Empreago[3];
       emps[0] = new Empregado("Jose Silva", 3200, new Day(1998,10,1));
       emps[1] = new Empregado("Maria Oliveira", 1100, new Day(1997,12,15));
       emps[2] = new Empregado("Joao Barros", 300, new Day(2003,3,15));
       ArrayAlg.shellSort(emps);
       for (int i = 0; i < emps.length; i++)
           System.out.println(emps[i]);
   }
}
```

Herança Múltipla

- No exemplo anterior, Empregado é subclasse de Sortable. Suponha que Empregado seja também descendente de Pessoa. Então tem-se a seguinte hierarquia:



- Neste caso, Empregado tem duas superclasses, o que caracteriza uma herança múltipla.
- **Herança múltipla** ocorre quando uma classe herda de mais de uma superclasse.

Interface

- Java não permite herança múltipla de classes, como no exemplo anterior.
- Java possui apenas um tipo restrito de herança múltipla, onde no máximo uma das superclasses é uma "classe" e as outras são "interfaces".
- **Interface** é uma classe "totalmente" abstrata. É uma classe que possui somente métodos abstratos e constantes.
- Uma interface é declarada usando-se a palavra-chave ***interface***. Não se usa a palavra-chave *abstract* na declaração dos métodos (todos os métodos já são abstratos).

Interface

- Uma interface é herdada usando-se a palavra-chave ***implements*** ao invés de extends. Diz-se que a subclasse implementa a interface, ou seja, implementa os métodos abstratos da interface.
- Na herança múltipla do exemplo anterior, tem-se:

```
public interface Sortable {
    public int compareTo (Sortable[] b);
}

public class Empregado extends Pessoa implements Sortable  { . . . }
```

- Se empregado também for gravável, tem-se:

```
public class Empregado extends Pessoa implements Sortable,
   Serializable  { . . . }
```

- Uma inteface pode herdar de outra(s) interface(s)

Interface

- O exemplo anterior, de ordenação de empregados, pode ser mudado para interface da seguinte forma:

```
public interface Sortable {
    public int compareTo (Sortable[] b);
}

public class Empregado implements Sortable  { . . . }

/* As classes ArrayAlg e EmpregadoSortTeste não são alteradas */
```

- Java já tem pronta uma interface de nome **Comparable** no pacote java.lang, equivalente a Sortable. Também, a classe **Arrays** do pacote java.util tem um método de ordenação de nome **sort**.

Entrada e Saída

- Em Java, um objeto do qual se pode ler uma seqüência de bytes é chamado de um fluxo de entrada (implementado pela classe abstrata ***InputStream***).
- Um objeto no qual se pode escrever uma seqüência de bytes é chamado de um fluxo de saída (implementado pela classe abstrata ***OutputStream***).
- Esses objetos fonte e destino de seqüências de bytes são normalmente arquivos, mas também podem ser conexões de rede e mesmo blocos de memória.
  - vantagem dessa generalidade: informações armazenadas em arquivos, por exemplo, são tratadas essencialmente da mesma forma que aquelas obtidas de uma conexão de rede.

Entrada e Saída

- Os dados, no final das contas, são armazenados como uma série de bytes, mas pode-se pensar neles, em um nível mais alto, como sendo uma seqüência de caracteres ou de objetos.
- Java fornece uma hierarquia de classes de entrada e saída cuja base são as classes InputStream e OutputStream.
- Pode-se manipular fluxos de diversos formatos: compactados, textos, registros de tamanho fixo com acesso aleatório e objetos.
- A seguir, será apresentada uma forma para se manipular fluxos de objetos usando o mecanismo de serialização de objetos.

Fluxos de Objetos

- Usando-se as classes ***ObjectOutputStream*** e ***ObjectInputStream*** pode-se gravar e ler objetos em um fluxo, de forma automática, através de um mecanismo chamado serialização de objetos.
- Classe java.io.ObjectOutputStream:
  - ObjectOutputStream (OutputStream saida)
    - cria um *ObjectOutputStream* de modo que se possa escrever objetos no *OutputStream* (fluxo de saída) especificado.
  - void writeObject (Object obj)
    - escreve o objeto especificado no *ObjectOutputStream*. A classe do objeto, a assinatura da classe e os valores dos campos não marcados como *transient* são escritos, bem como os campos não estáticos da classe e de todas as suas superclasses.

Fluxos de Objetos

- Classe java.io.ObjectInputStream:
  - ObjectInputStream (InputStream entrada)
    - cria um *ObjectInputStream* para ler informações de objetos do *InputStream* (fluxo de entrada) especificado.
  - Object readObject ()
    - lê um objeto do *ObjectInputStream*. Em particular, lê a classe do objeto, a assinatura da classe e os valores dos campos não transientes e não estáticos da classe e de todas as suas superclasses. Ele faz a desserialização para permitir que múltiplas referências de objetos possam ser recuperadas.
- Classe java.io.FileInputStream:
  - FileInputStream (String nome)
    - cria um novo fluxo de entrada de arquivo, usando o arquivo cujo nome de caminho é especificado pela string *nome*.

Fluxos de Objetos

- Classe java.io.FileOutputStream:
  - FileOutputStream (String nome)
    - cria um novo fluxo de saída de arquivo especificado pela string *nome*. Os nomes de caminhos que não são absolutos são interpretados como relativos ao diretório de trabalho.

      Atenção: apaga automaticamente qualquer arquivo existente com esse nome.
  - FileOutputStream (String nome, boolean anexar)
    - cria um novo fluxo de saída de arquivo especificado pela string *nome*. Os nomes de caminhos que não são absolutos são interpretados como relativos ao diretório de trabalho. Se o parâmetro *anexar* for *true*, então os dados serão adicionados ao final do arquivo. Um arquivo existente com o mesmo nome não será apagado.

Fluxos de Objetos

- Passos para salvar dados de um objeto:
  - abra um objeto ObjectOutputStream

    ObjectOutputStream out = new ObjectOutputStream (new FileOutputStream("empregado.dat");
  - salve os objetos usando o método writeObject

    Empregado e1 = new Empregado("João",2500,new Day(1996,12,10));

    Gerente g1 = new Gerente("Maria",3800,new Day(1995,10,15);

    out.writeObject(e1);

    out.writeObject(g1);
- Passos para ler dados de um objeto:
  - abra um objeto ObjectInputStream

    ObjectInputStream in = new ObjectInputStream (new FileInputStream("empregado.dat"));

Fluxos de Objetos

- leia os objetos usando o método readObject

  Empregado e1 = (Empregado) in.readObject();

  Gerente g1 = (Gerente) in.readObject();

  - os objetos são gravados como *Object* e podem ser convertidos para o seu tipo original usando *cast*.
  - cada chamada de readObject() lê um objeto na mesma ordem em que foram salvos.

- Para que uma classe possa ser gravada e restaurada num fluxo de objetos é necessário que ela implemente a interface **Serializable**. Esta interface não tem métodos, portanto não é necessário alterar nada na classe.

  public class Empregado **implements Serializable** { . . . }

Fluxos de Objetos

- Pode-se salvar/restaurar um array de objetos em uma única operação:

```
Empregado[] emp = new Empregado[3];
. . .
out.writeObject(emp);
. . .
Empregado[] empNovo = (Empregado[]) in.readObject();
```

- Ao salvar um objeto que contém outros objetos, os mesmos são gravados automaticamente.
  - quando um objeto é compartilhado por vários outros objetos, somente uma cópia é gravada.
    - a técnica de serialização atribui um número de série a cada objeto. Após um objeto ser gravado, uma outra cópia compartilhada apenas recebe uma referência ao número de série já gravado anteriormente.

Tratamento de Exceções

- Considere o seguinte trecho de código:

```
public class Estoque {
  private int codProduto;
  private int qtdeEstoque;
  public void decrementaEstoque (int qtde) {
        qtdeEstoque = qtdeEstoque - qtde;
  }
}
```

- O que fazer se o estoque não for suficiente?
  - Desconsiderar operação
  - Mostrar mensagem de erro
  - Retornar código de erro

Tratamento de Exceções

- Opção 1: desconsiderar operação

```
public class Estoque {
  private int codProduto;
  private int qtdeEstoque;
  public void decrementaEstoque (int qtde) {
        if (qtdeEstoque >= qtde)
              qtdeEstoque = qtdeEstoque - qtde;
  }
}
```

- Problema:
  - Não há como saber se a operação foi realizada. Nenhuma informação é retornada ao cliente (outros métodos da classe ou de outras classes).

Tratamento de Exceções

- Opção 2: mostrar mensagem de erro

```
public class Estoque {
  private int codProduto;
  private int qtdeEstoque;
  public void decrementaEstoque (int qtde) {
        if (qtdeEstoque >= qtde)
              qtdeEstoque = qtdeEstoque - qtde;
        else System.out.println("Estoque Insuficiente!");
  }
}
```

- Problema:
  - Gera dependência entre a classe e sua interface de usuário;
  - Não retorna informação ao cliente.

Tratamento de Exceções

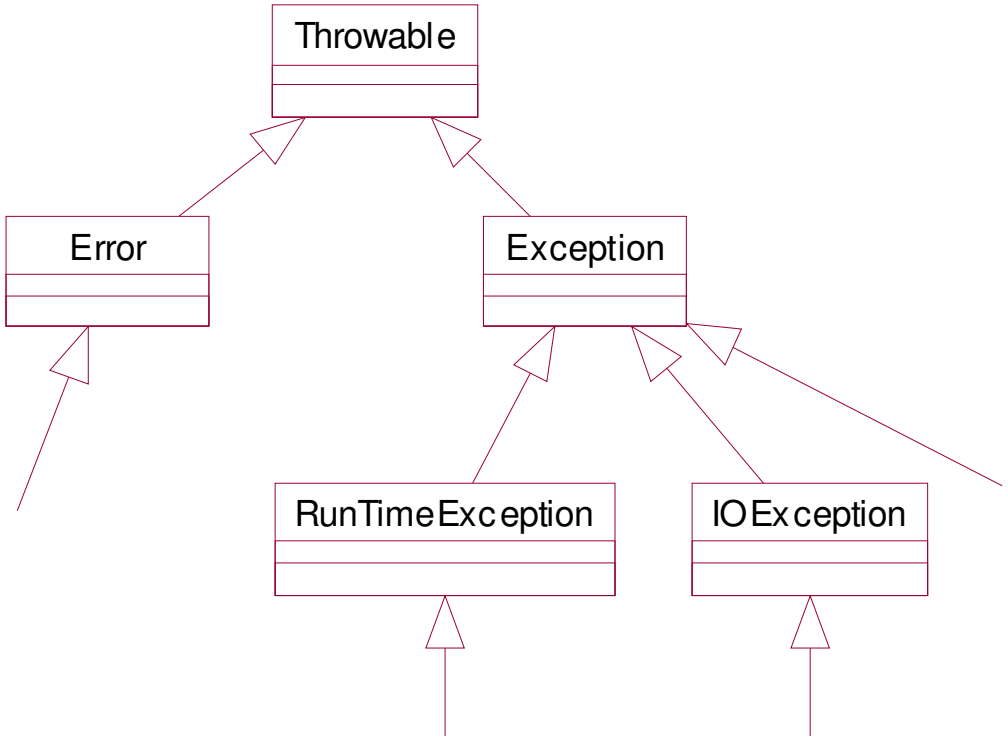
- Opção 3: retornar código de erro

```
public class Estoque {
  private int codProduto;
  private int qtdeEstoque;
  public boolean decrementaEstoque (int qtde) {
        if (qtdeEstoque >= qtde) {
              qtdeEstoque = qtdeEstoque - qtde;
              return true; }
        else return false;
  } }
```

- Problema:
  - Complica definição e uso do método (clientes têm que fazer testes);
  - Pior para métodos que já retornam valores.

Tratamento de Exceções

- Solução:
  - Usar esquema de tratamento de exceções.
- Hierarquia de exceções em Java:



Tratamento de Exceções

- Error
  - Descreve erros internos. Ex: falta de memória, disco cheio.
  - Não se deve lançar um objeto desse tipo.
  - Pouco se pode fazer se um erro interno ocorre, além de notificar o usuário e tentar finalizar o programa.
- Exception
  - RunTimeException
    - Ocorre porque houve um erro de programação (culpa do programador).
    - Ex: conversão explícita de tipo (cast), acesso a elemento de array além dos limites, acesso de apontador nulo.
  - Outras exceções
    - Ex: tentar ler além do final de um arquivo, tentar abrir URL incorreta.

Tratamento de Exceções

- As exceções que derivam da classe ***Error*** ou da classe ***RunTimeException*** são chamadas exceções *não verificadas*. Todas as demais são chamadas *verificadas*.
- Um método precisa declarar todas as exceções verificadas que ele lança. Palavra-chave ***throws*** no cabeçalho.

```
public void gravaDados() throws IOException {
    ObjectOutputStream out = new ObjectOutputStream(
          new FileOutputStream("rh.dat"));
    out.writeObject(funcionarios);
    out.close();
}
```

  - Se o método não conseguir criar o arquivo ou gravar os dados, ocorrerá uma exceção do tipo ***IOException***, que deverá ser tratada.

Tratamento de Exceções

- A hierarquia de exceções possui várias classes para diversos tipos de exceções que podem ser lançadas pelos programas.
- Para lançar uma exceção, use a palavra-chave ***throw***.

```
public String lerDados (BufferedReader ent) throws
  EOFException {
  . . .
  while ( . . . ) {
        if (ch == -1)
              if (n < compr)
                    throw new EOFException();
  } }
```

  - a classe EOFException sinaliza que ocorreu um fim de arquivo inesperado durante a entrada de dados.

Tratamento de Exceções

- Como criar suas próprias classes de Exceção?
  - Um programa pode ter um problema que não está descrito adequadamente em nenhuma das classes de exceção padrão.
  - Para criar um classe de exceção basta derivá-la de ***Exception*** ou de uma classe descendente de ***Exception***.
  - É habitual criar um construtor padrão e um construtor que contenha uma mensagem detalhada.
  - Construtores e um método da classe ***Throwable***:
    - *Throwable()* : constrói um novo objeto sem mensagem detalhada.
    - *Throwable (String mensagem)* : constrói um novo objeto com a mensagem detalhada especificada.
    - *String getMessage()* : obtém a mensagem detalhada do objeto *Throwable*.

Tratamento de Exceções

- Criação de uma classe de exceção

```
public class EIException extends Exception {
  private int codProd;
  private int qtdeEst;
  public EIException () {
        super("Estoque Insuficiente!"); }
  public EIException (int cod, int qtde) {
        super("Estoque Insuficiente!");
        codProd = cod;
        qtdeEst = qtde; }
  public int getCodProduto() { return codProd; }
  public int getQtdeEstoque() {return qtdeEst; }
}
```

Tratamento de Exceções

- Lançamento de uma exceção criada

```
public class Estoque {
  private int codProduto;
  private int qtdeEstoque;
  public void decrementaEstoque (int qtde) throws EIException
  {
        if (qtdeEstoque >= qtde)
              qtdeEstoque = qtdeEstoque - qtde;
        else throw new EIException(codProduto, qtdeEstoque);
  }
}
```

Tratamento de Exceções

- Exceções levantadas indiretamente também devem ser tratadas

```
public class Pedido {
  . . .
  public void adicionaItem (Estoque est, int qtde) throws
  EIException {
        . . .
        est.decrementaEstoque (qtde);
        . . .
  }
}
```

Tratamento de Exceções

- Se não se desejar criar uma classe de exceção própria pode-se levantar uma exceção com uma mensagem personalizada através de uma classe pré-existente

```
public class Estoque {
  private int codProduto;
  private int qtdeEstoque;
  public void decrementaEstoque (int qtde) throws Exception {
        if (qtdeEstoque >= qtde)
              qtdeEstoque = qtdeEstoque - qtde;
        else throw new Exception("Estoque Insuficiente!");
}}
```

  - Mas lembre-se que a classe ***Exception*** é muito genérica e captura qualquer exceção!

Tratamento de Exceções

- Captura de exceções
  - As exceções devem ser capturadas e tratadas adequadamente.
  - Se ocorrer uma exceção e esta não for capturada em nenhum lugar, o programa termina mostrando uma mensagem de erro.
  - Para capturar uma exceção, especifica-se um bloco **try catch finally** (sendo catch e finally opcionais).

```
try {
   comando1;
   comando2;  . . .  comandon;
} catch (TipoExceção1 e1)  {tratamento de e1}
   catch (TipoExceção2 e2)  {tratamento de e2}  . . .
   catch (TipoExceçãon en)  {tratamento de en}
finally {finalizações}
```

Tratamento de Exceções

- Exemplo de captura de exceção

```
public class Interface {
  . . .
  public void insereItemPedido {
        . . .
        try {
            pedido.adicionaItem(est, qtde);
        } catch (EIException eie) {
            System.out.println(eie.getMessage()+
                  " Produto: "+eie.getCodProduto()+
                  " Estoque: "+eie.getQtdeEstoque());
        }
        . . .
  }
```

Tratamento de Exceções

- Se o código dentro do bloco ***try*** não lançar nenhuma exceção
  - o programa executa o código dentro de ***try***, pula a cláusula ***catch***, executa o código dentro da cláusula ***finally*** e o restante do código.
- Se qualquer parte do código dentro do bloco ***try*** lançar uma exceção da classe especificada na cláusula ***catch***
  - o programa pula o restante do código do bloco ***try*** a partir do ponto onde a exceção foi lançada, executa o código da cláusula ***catch*** correspondente e, então, o código da cláusula ***finally***.
    - Se a cláusula ***catch*** não lançar nenhuma exceção, o programa executa a primeira linha depois do bloco ***try***. Senão, a exceção é lançada de volta para o chamador do método.

Tratamento de Exceções

- Se código dentro do bloco ***try*** lançar uma exceção que não é capturada por nenhuma cláusula ***catch***
  - o programa pula o restante do código do bloco ***try*** a partir do ponto onde a exceção foi lançada, executa o código na cláusula ***finally*** e lança a exceção de volta para o chamador do método.

Tratamento de Exceções

- Para capturar múltiplas exceções
  - usam-se cláusulas ***catch*** separadas, uma para cada tipo de exceção.
    - Deve-se colocar as classes mais específicas primeiro.
- Exceções podem ser relançadas dentro da cláusula ***catch***.

```
. . .
Graphics g = image.getGraphics();
try {
   código que pode lançar exceções
} catch (MalformedURLException e) {
          g.dispose();
          throw e;
}
```