

# **TUTORIAL:**

**“A LINGUAGEM DE PROGRAMAÇÃO**

# **JAVA™,”**

**E**

**ORIENTAÇÃO A OBJETOS**

Instituto de Computação  
Julho 1996;  
Atualizado 2009.

André Augusto Cesta. [aacesta@yahoo.com](mailto:aacesta@yahoo.com)

Orientadora: Prof<sup>fa</sup> Dr<sup>a</sup> Cecília Mary Fischer Rubira



# INFORMAÇÃO SOBRE “COPYRIGHT”

Copyright 1996- IC - UNICAMP.

Este texto e documentos relacionados são protegidos por copyright.

Esta publicação é apresentada como está sem garantia de erros e mudanças, sejam nos endereços (URL's) , texto ou do código reutilizado nos programas exemplo.

Para obter a última versão deste texto consulte <http://www.ic.unicamp.br/~cmrubira/aacesta/>

## **Marcas Registradas:**

Sun, o logotipo da Sun, Sun Microsystems, Solaris, HotJava e Java são marcas registradas de Sun Microsystems, Inc. nos Estados Unidos e em alguns outros países. O personagem “Duke” é marca registrada de Sun Microsystems. UNIX é marca registrada nos Estados Unidos e outros países, exclusivamente licenciada por X/Open Company, Ltd.. Netscape Navigator é marca registrada de: “Netscape Communications Corporation”.



## Prefácio:

Este texto faz parte de um estudo comparativo de linguagens de programação orientadas a objetos. O conteúdo deste estudo também está disponível na Web sob o endereço <http://www.ic.unicamp.br/~cmrubira/aacesta/>. Neste endereço, você pode complementar seu aprendizado, rodando exercícios iterativos, acessando “links” para outros hipertextos sobre linguagens de programação, vendo exemplos de programas e interagindo com aplicações para a Internet.

A diferença entre este estudo e outros textos que você possa encontrar sobre o mesmo assunto é o caráter prático. Exemplos completos, dicas de programação, explicações sobre detalhes normalmente ignorados em livros, tornarão seu aprendizado mais fácil, principalmente na segunda parte onde tratamos da construção de aplicações para a Internet.

No início, os exemplos podem ser considerados fáceis, mas eles vão se complicando cada vez mais de modo que é importante que o leitor acompanhe o texto fazendo os exercícios. Forneceremos uma série de idéias de programas simples para que você possa testar seu conhecimento. Estes programas simples podem ser melhorados através do uso de interfaces gráficas, assunto que não é coberto neste tutorial.

## QUEM DEVERIA LER ESTE TUTORIAL

Qualquer leitor que tenha experiência com pelo menos uma linguagem de programação.

## DIAGRAMAÇÃO DESTE TEXTO

Apesar de **Java** ser uma linguagem que serve para vários propósitos, o seu sucesso atual (época do seu lançamento em 1996) se deve a possibilidade de elaboração de aplicações para a Internet. Dada a importância deste aspecto da linguagem, este texto está organizado de maneira semelhante as páginas encontradas na WWW, frequentemente você encontrará diagramas como o seguinte:



TUTORIAL

<http://www.dcc.unicamp.br/~cmrubira/aacesta>

*“Estudo comparativo de linguagens de programação orientadas a objetos”. Nesta página você encontrará tutoriais sobre diversas linguagens orientadas a objetos, dentre elas: C++, Modula-3, **Java**. Quanto a Java, você terá a oportunidade de ver código de programas e testá-los, além de poder adquirir a versão mais nova deste texto. Encontrará também links para sites sobre orientação a objetos.*

Este diagrama representa um hipertexto que pode ser acessado de modo a complementar seu estudo. A parte escrita em letra maior é o endereço, o texto em itálico faz um resumo do conteúdo desta página. Usando estes “links” ou diagramas você encontrará uma maneira ordenada de aprender sem se perder no mar de informações da Internet.



**ABC**

URL: “Uniform Resource Locator”, é o endereço de um computador na internet

O diagrama acima aparecerá toda vez que introduzirmos uma palavra nova. Caso você encontre alguma palavra desconhecida, basta usar o índice remissivo para obter sua definição.

É importante lembrar que os hipertextos citados neste tutorial, não são de nossa responsabilidade. Como eles estão sujeitos a mudanças, contamos com a sua ajuda para efetuarmos atualizações, conte você também com a nossa ajuda na internet.

Os programas exemplo deste texto são apresentados em caixas retangulares, e todo código é escrito em fonte diferente da usada neste texto comum. O trechos de código que aparecem desmembrados de seus arquivos, tem fundo cinza claro. Todos os arquivos presentes dentro dos retângulos, são arquivos “text-only”, qualquer formatação (negrito) tem apenas função didática.

Os resultados dos programas são indicados pelo diagrama ao lado:





## DIVISÃO DO TUTORIAL


Este tutorial contém uma sequência de tópicos que permite apresentar a linguagem sob a ótica da teoria de orientação a objetos. A apresentação do modelo de objetos da linguagem e conceitos relacionados tais como polimorfismo, tratamento de exceções está em primeiro plano. Ao longo dessa apresentação, em segundo plano, você aprenderá os aspectos básicos da linguagem tais como “loops”, desvios condicionais, etc.

Estes tópicos são frequentemente retomados, cada vez de maneira mais aprofundada. Quando terminamos o assunto métodos, você já está pronto para saber o que são construtores, e é exatamente isto que ensinamos. Porém o assunto construtores não é esgotado, você ainda vai aprender a usar construtores em conjunto com agregação e depois em conjunto com herança.

A maioria dos leitores fica ansiosa para aprender como criar aplicações para a Internet, mas depois de satisfeita esta ansiedade voltam para o ponto onde aprendem como programar na linguagem e não apenas experimentar com a criação de botões, caixas de diálogo, imagens, etc. Se esse é o seu caso, é recomendável um “tour” pela WWW antes de começar a programar, um bom “site” para começar a pesquisar com um “browser” compatível com Java (Netscape Navigator 2.0™ ou superior) é :

  
URL

  
APPLETS

  
ABC

<http://java.sun.com/applets/>  
“Links” para vários applets, divididos por categorias: games, sound, busines, animation...  
Divirta-se...

*APPLETS: São pequenos programas escritos em Java que podem ser embebidos em documentos hipertextos. São exemplos de applets: animações, imagens, botões, etc. Applets podem suportar efeitos de multimídia como sons, iterações com o usuário (mouse, teclado), imagens, animações, gráficos, etc.*

Ao longo do texto, você perceberá que Java é excelente para desenvolver aplicações comerciais e para ser usada em universidades. Java pode ser vista como uma fusão de várias tecnologias que vêm sendo desenvolvidas na área de computação, de modo que estudantes dessa linguagem tem a oportunidade de tomar contato com vários tópicos recentes: programação concorrente, sistemas distribuídos, orientação a objetos, protocolos da internet, business intelligence e uma série de outros assuntos fáceis de praticar nessa linguagem.

É claro que os resultados dependerão de seu esforço, portanto depois de ler esta introdução, descanse um pouco, tome um cafezinho<sup>1</sup>. Pois durante o restante do texto esperamos que você se envolva com a linguagem, reuse programas encontrados na Web, se comunique com colegas programadores, participe de listas de discussões, newsgroups (comp.lang.Java & alt.www.hotJava), e o mais importante: PROGRAME, PROGRAME, RE-PROGRAME!

<sup>1</sup> Um dos símbolos da linguagem é uma xícara de café que aparece em animações com sua fumaça quente tremulando.

## ADQUIRINDO O SOFTWARE NECESSÁRIO, PLATAFORMAS SUPORTADAS

Um ambiente de programação **Java** é normalmente composto de um kit de desenvolvimento de aplicações **Java** e um “browser compatível com esta linguagem (recomendável). Se você não tem acesso a esse ambiente de programação, tente estes endereços:



<http://Java.sun.com>

*Raiz do hipertexto montado pelos criadores da linguagem. Sob este endereço você pode obter o compilador e outras ferramentas de desenvolvimento de aplicações **Java** para a sua plataforma de programação. Fique atento! Outros desenvolvedores estão criando ambientes de programação Java como Eclipse.*



<http://www.netscape.com>

*Raiz do hipertexto montado pelos criadores do Netscape Navigator™. Sob este endereço você pode obter o browser “**Java** compatible” da “Netscape Communications INC’.. Outros desenvolvedores estão lançando “browsers” compatíveis com **Java**.*



<http://www.microsoft.com>

*A microsoft™ licenciou a tecnologia Java e a incorporou em seu novo browser: Internet Explorer versão 3.0 ou superior.*



***BROWSERS:** São uma categoria de programas que permitem você visualizar um documento criado em um certo padrão, no caso html (hypertext markup language). Atualmente os browsers tem se tornado complexos devido a quantidade de padrões existentes (ex. imagens .gif .jpg, etc). A linguagem Java pode contribuir para minimizar esta complexidade.*

## CARACTERÍSTICAS DA LINGUAGEM

### Parecida com C, C++:

**Java** tem a aparência de C ou de C++, embora a filosofia da linguagem seja diferente. Por este motivo estaremos frequentemente fazendo comparações com alguma destas linguagens. O leitor que programa em qualquer uma delas, ou em uma linguagem orientada a objetos, se sentirá mais a vontade e se tornará um bom programador **Java** em menos tempo.

**Java** também possui características herdadas de muitas outras linguagens de programação: Objective-C, Smalltalk, Eiffel, Modula-3, etc. Muitas das características desta linguagem não são totalmente novas. **Java** é uma feliz união de tecnologias testadas por vários centros de pesquisa e desenvolvimento de software.

### Distinta de Javascript:

Muita gente confunde **Java** com **JavaScript**. As duas linguagens são semelhantes na sintaxe, mas diferem no sistema de tipos, orientação a objectos e outros aspectos. Javascript tem sido combinada com sucesso com Java e dynamic HTML resultando em aplicações dynamicas conhecidas como “AJAX enabled applications”.

### Compilada:

Um programa em **Java** é compilado para o chamado “byte-code”, que é próximo as instruções de máquina, mas não de uma máquina real. O “byte-code” é um código de uma máquina virtual idealizada pelos criadores da linguagem. Por isso **Java** pode ser mais rápida do que se fosse simplesmente interpretada.

### **Portável:**

**Java** foi criada para ser portátil. O “byte-code” gerado pelo compilador para a sua aplicação específica pode ser transportado entre plataformas distintas que suportam **Java** (Solaris, Windows, Mac/Os etc). Não é necessário recompilar um programa para que ele rode numa máquina e sistema diferente, ao contrário do que acontece por exemplo com programas escritos em C e outras linguagens.

Esta portabilidade é importante para a criação de aplicações para a heterogênea Internet. Muitos dos programas exemplo deste tutorial foram escritos e compilados numa plataforma Windows-95® em 1996 e rodaram perfeitamente quando simplesmente copiados para uma plataforma Solaris®. Em **Java** um inteiro por exemplo, tem sempre 32 bits, independentemente da arquitetura. O próprio compilador **Java** é escrito em **Java**, de modo que ele é portátil para qualquer sistema que possua o interpretador de “byte-codes”. Um exemplo de programa escrito em **Java** é a aplicação de file sharing multi-plataforma chamada Limewire<sup>2</sup>.

### **Orientada a Objetos:**

A portabilidade é uma das características que se inclui nos objetivos almejados por uma linguagem orientada a objetos. Em **Java** ela foi obtida de maneira inovadora com relação ao grupo atual de linguagens orientadas a objetos.

**Java** suporta herança, mas não herança múltipla. A ausência de herança múltipla pode ser compensada pelo uso de herança e interfaces, onde uma classe herda o comportamento de sua superclasse além de oferecer uma implementação para uma ou mais interfaces.

**Java** permite a criação de classes abstratas. Outra característica importante em linguagens orientadas a objetos é a segurança. Dada a sua importância o tópico foi escrito a parte.

### **Segura:**

A presença de coleta automática de lixo, evita erros comuns que os programadores cometem quando são obrigados a gerenciar diretamente a memória (C, C++, Pascal). A eliminação do uso de ponteiros, em favor do uso de vetores, objetos e outras estruturas substitutivas traz benefícios em termos de segurança. O programador é proibido de obter acesso a memória que não pertence ao seu programa, além de não ter chances de cometer erros comuns tais como “reference aliasing” e uso indevido de aritmética de ponteiros. Estas medidas são particularmente úteis quando pensarmos em aplicações comerciais desenvolvidas para a internet.

Ser “strongly typed” também é uma vantagem em termos de segurança, que está aliada a eliminação de conversões implícitas de tipos de C++.

A presença de mecanismos de tratamento de exceções torna as aplicações mais robustas, não permitindo que elas abortem, mesmo quando rodando sob condições anormais. O tratamento de exceções será útil na segunda parte para modelar situações tais como falhas de transmissão e

---

<sup>2</sup> É importante dar exemplos para que o leitor curioso possa conferir o “look and feel” e qualidade das aplicações. Para mais aplicações escritas em Java visite: <http://java-source.net/>.



formatos incompatíveis de arquivos.

### **Suporta concorrência:**

A linguagem permite a criação de maneira fácil, de vários “threads” de execução. Este tópico será útil quando você estudar animações, e é particularmente poderoso nos ambientes em que aplicações **Java** são suportadas, ambientes estes que geralmente podem mapear os threads da linguagem em processamento paralelo real em máquinas que o suportam a exemplo das máquinas multi-core.

### **Eficiente:**

Como **Java** foi criada para ser usada em computadores pequenos, ela exige pouco espaço, pouca memória. **Java** é muito mais eficiente que grande parte das linguagens de “scripting” existentes, embora seja mais lenta que C. Com a evolução da linguagem, serão criados geradores de “byte-codes” cada vez mais otimizados que trarão as marcas de performance da linguagem mais próximas das de C++ e C. Além disso um dia **Java** permitirá a possibilidade de gerar código executável de uma particular arquitetura “on the fly”, tudo a partir do “byte-code”.

### **Suporte para programação de sistemas distribuídos:**

**Java** fornece facilidades para programação com sockets, remote method call, tcp-ip, etc. Estes tópicos não serão abordados neste texto.

# PROGRAMAÇÃO ORIENTADA A OBJETOS

## NOTA AOS PROGRAMADORES “C”

A leitura deste hipertexto é fortemente recomendada para os programadores que tem “C” ou “C++” como sua principal linguagem de programação, mesmo antes de iniciar a leitura deste tutorial:



<http://www.cafeaulait.org/javatutorial.html>

*“Brewing Java: A Tutorial”, Um tutorial de Java que segue o estilo do livro inicial sobre “C”, escrito por Kernighan & Ritchie [2], é a referência básica para este texto. Cobre principalmente sintaxe de loops e outros assuntos fáceis não abordados aqui.*

Iniciaremos com um pouquinho de teoria sobre orientação a objetos. Se você quiser algum hipertexto para aprofundamento ou para ler em paralelo com esta introdução sobre classes e objetos e use:



<http://java.sun.com/docs/books/tutorial/java/concepts/>

*“Introdução a orientação a objetos e Java em inglês”*

## CLASSES E OBJETOS

Uma classe é um tipo definido pelo usuário que contém o molde, a especificação para os objetos, algo mais ou menos como o tipo inteiro contém o molde para as variáveis declaradas como inteiros. A classe envolve, associa, funções e dados, controlando o acesso a estes, defini-la implica em especificar os seus atributos (dados) e seus métodos (funções).

Um programa que utiliza uma interface controladora de um motor elétrico provavelmente definiria a classe **motor**. Os atributos desta classe seriam: temperatura, velocidade, tensão aplicada. Estes provavelmente seriam representados na classe por tipos como **int** ou **float**. Os métodos desta classe seriam funções para alterar a velocidade, ler a temperatura, etc.

Um programa editor de textos definiria a classe **parágrafo** que teria como um de seus atributos uma **String** ou um vetor de **Strings**, e como métodos, funções que operam sobre estas strings. Quando um novo parágrafo é digitado no texto, o editor cria a partir da classe **Parágrafo** um objeto contendo as informações particulares do novo texto. Isto se chama instanciação ou criação do objeto.

## ESPECIFICANDO UMA CLASSE

Suponha um programa que controla um motor elétrico através de uma saída serial. A velocidade do motor é proporcional a tensão aplicada e esta proporcional aos bits que vão para saída serial e passam por um conversor digital analógico.

Vamos abstrair todos estes detalhes por enquanto e modelar somente a interface do motor como uma classe, a pergunta é que métodos e que atributos deve ter nossa classe, que argumentos e valores de retorno devem ter os métodos?

### Representação da velocidade:

A velocidade do motor será representada por um atributo inteiro (**int**). Usaremos a faixa de bits

que precisarmos, caso o valor de bits necessário não possa ser fornecido pelo tipo , usaremos então o tipo `long`, isto depende do conversor digital analógico utilizado.

### Representação da saída serial:

O motor precisa conhecer a sua saída serial, a sua ligação com o “motor do mundo real”. Suponha uma representação em hexadecimal do atributo endereço de porta serial, um possível nome para o atributo: `enderecomotor`. Não se preocupe em saber como usar a representação hexadecimal.

### Alteração do valor da velocidade:

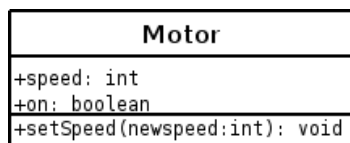
Internamente o usuário da classe `motor` pode desejar alterar a velocidade, cria-se então o método: `public void setSpeed(int newSpeed);`. O código anterior corresponde ao cabeçalho do método ele é definido junto com a classe `motor`, associado a ela. O valor de retorno da função que “implementa” o método é `void`, poderia ser criado um valor de retorno (`boolean`) que indicasse se o valor de velocidade era permitido e foi alterado ou não era permitido e portanto não foi alterado.

O ato de invocar um método também é chamado de passar uma mensagem para o objeto que está executando este método.

Não faz sentido usar, chamar, este método separado de uma variável do tipo `motor`, mas então porque na lista de argumentos da função não se encontra um `motor`? Este pensamento reflete a maneira de associar dados e código (funções) das linguagens procedurais. Em linguagens orientadas a objetos o código e os dados são ligados de forma diferente, a própria declaração de um tipo definido pelo usuário já engloba as declarações das funções inerentes a este tipo, isto será explicado em “CLASSES”. O objeto ao qual é aplicado o método é passado de outra forma.

Note que não fornecemos o código do método, isto não é importante, por hora a preocupação é com a interface definida pela classe: seus cabeçalhos de métodos e atributos. Apenas pense que sua interface deve ser flexível de modo a não apresentar entraves para a criação do código que seria feita numa outra etapa. Nesta etapa teríamos que imaginar que o valor numérico da velocidade deve ir para o conversor onde irá se transformar numa diferença de potencial a ser aplicada nos terminais do motor, etc.

### Um diagrama simplificado da classe `motor` com os atributos e métodos:



Este e outros diagramas deste texto foram elaborados com uma ferramenta case para “object oriented modeling and design” segundo a metodologia descrita em [1]

### Acessores de atributos:

Você pode se perguntar porque não usamos um método de nome `alteraVelocidade(nova velocidade)` envés de `setSpeed(newspeed)`, uma vez que estamos escrevendo o tutorial e também o código em língua portuguesa. A razão é um padrão de programação Java chamado “beans” que especifica que qualquer método que altere somente o valor de um atributo de nome X deve se chamar `setX` e qualquer método que retorne o valor de um atributo de nome X deve se chamar `getX`.

Embora esse padrão não seja obrigatório ele é usado pois Servlets, Reflexão, JSF e outras APIs fazem uso dele. Esses métodos são chamados em inglês de *accessors* or *getters and setters*.

## Exercícios:

1-

Lembre-se de algum programa em que você trabalhou, cite que tipos de classes seriam criadas se esse programa fosse escrito em **Java**, que atributos e que métodos estariam associados aos objetos dessas classes? Desenhe o diagrama de classe para as cada classe de objetos identificada.

**Exemplo:** “Eu trabalhei em um programa de contas a pagar e contas a receber. Se esse programa fosse escrito em **Java** eu definiria a classe **contaBancaria**. Os atributos seriam: **saldo**, **taxaDeJuros**, **limiteDeSaque**, etc. Minha opção seria por representá-los como variáveis do tipo **double** (não se preocupe em usar os tipos da linguagem inda)”. “Dentre os métodos desta classe estariam funções para efetuar saques, depósitos e computar juros.”

## OBJETOS EM JAVA

Objetos são instâncias de uma classe. Quando um objeto é criado ele precisa ser inicializado, ou seja para uma única classe de nome **EstudanteDeGraduacao** podemos ter vários objetos ativos em memória durante a execução de um programa.

Estudante de graduação **Andre**; Identificação **940718**; Curso **Computacao** | Estudante de graduação **Luiza** , Identificação **893249**, Curso **Medicina**... A classe representa somente o molde para a criação dos objetos, estes sim contém informação, veja tópico “CLASSES E OBJETOS”.

O atributo Identificação tem valor 940718 para a instância (objeto) André da classe Estudantes de Graduação.

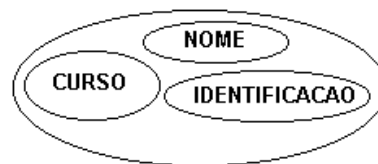


**INSTANCIAS:** Um objeto existente durante um momento da execução de um programa é uma instancia de uma classe.

Uma classe e suas instancias:



Cada estudante (ou instancia) poderia ser modelado, desenhado como:



Objetos podem conter objetos, ou seja os atributos de um objeto podem ser objetos, da mesma classe ou não. Objetos podem ser passados pela rede, armazenados em meio físico. Objetos possuem um estado e um comportamento. Métodos podem receber objetos como argumentos, podem declarar objetos como variáveis locais, podem chamar outros métodos. Você pode chamar um método (mandar uma mensagem) para objetos em outras máquinas através de sua rede.

Um objeto pode ser visto como um **RECORD** de Pascal ou **Strut** de C, só que com uma tabela de funções que podem ser chamadas para ele. Na verdade esta definição não é muito teórica, mas é um bom começo para os programadores que estão acostumados com linguagens procedurais. Na verdade podemos fazer com objetos muito mais do que fazemos com records e procedimentos em Pascal.

Em **Java**, ao contrário de C++ e Modula-3, não existem funções desvinculadas de classes, funções isoladas. Isto implica que todo trecho de código que for escrito deve pertencer a uma classe, mais precisamente deve ser um método desta. O programa mais simples em **Java** deve conter pelo menos uma classe e um método de início de programa, e é este programa que faremos agora.

Esta filosofia é simples e semelhante a adotada em Eiffel, tudo o que se pode fazer com procedimentos, funções isoladas e variáveis de procedimentos, também se pode fazer com classes e métodos. C++ tinha que permitir a criação de funções isoladas para manter a compatibilidade com “C”, mas **Java** não. Quando neste texto usarmos o termo função no lugar de métodos estaremos mais interessados em enfatizar a parte de implementação em detrimento da interface, você pensar que em **Java** toda função implementa um método de uma classe.

O leitor não acostumado com o paradigma de orientação a objetos, pode achar estranhas as afirmações acima, e a pergunta mais comum neste momento é: “Mas então como você sabe aonde vai começar o programa?”. Antes da resposta a essa pergunta, leia o primeiro programa exemplo, que é semelhante ao primeiro programa em C, “Hello World”, presente em [2].

## PROGRAMA HELLO INTERNET!

Este exemplo visa apresentar um programa simples para imprimir uma mensagem na tela, este provavelmente será seu primeiro programa em Java.

### COMPILANDO UM PRIMEIRO PROGRAMA:

1-Certifique-se de ter adicionado a sua lista de path's o path do compilador e interpretador **Java**. **Javac** e **Java** respectivamente.

2-Crie o arquivo ao lado em um diretório qualquer (“folder” para usuários mac) e salve com o nome: **HelloInternet.java**

3-Chame o compilador **Java** para este arquivo:  
**javac HelloInternet.java**

4-Seu diretório deve ter recebido um novo arquivo após essa compilação: **HelloInternet.class**

5-Chame o interpretador **Java** para este arquivo (omite a extensão .class de arquivo): **Java HelloInternet**

6-Observe o resultado na tela: **Hello Internet!**

### CÓDIGO

```
//Comentario de uma linha

public class HelloInternet {

    public static void main (String args[]) {

        System.out.println("Hello Internet!");

    }

}
```



**Hello Internet!**

### Resolvendo os eventuais problemas:

Compile este programa. Você pode decidir por não prosseguir enquanto não compilá-lo. Se você for como eu, e principalmente se tiver digitado tudo ao invés de usar “copy and paste”, é bem provável que ele não compile, se isto ocorrer, leia atentamente o programa. Você não esqueceu o ponto e vírgula? E as chaves? **Hellointernet** começa com letra maiúscula, e isso faz diferença<sup>3</sup>, você

<sup>3</sup> Identificadores em letras maiúsculas e minúsculas são diferentes, a linguagem é “case-sensitive”.

foi coerente com a convenção adotada de letras maiúsculas e minúsculas para seus identificadores?

Você estava no mesmo diretório de `HelloInternet.java` quando chamou o compilador? E quando chamou o interpretador? Também?

Se ainda não der certo, não se desespere, leia as explicações passo a passo do programa e depois recorra a um usuário mais experiente. Normalmente a **Java** é muito fácil de se programar, e você precisará de pouca ajuda, o seu interesse o fará cada vez mais familiar com esta linguagem.

### Explicação passo a passo do programa exemplo:

#### //Comentario de uma linha

Comentários em **Java** seguem a mesma sintaxe de C++, “//” inicia uma linha de comentário, todo o restante da linha é ignorado. Existe também um outro tipo de comentário formado por `/*` *Insira aqui o texto a ser ignorado* `*/`, este tipo de comentário pode ser intercalado em uma linha de código. Comentários são tratados como espaços em branco.

#### public class HelloInternet {

**class** é a palavra reservada que marca o início da declaração de uma classe. **Public** é um especificador, por enquanto guarde **public class** como o início da declaração de uma classe. Todas as classes serão declaradas assim até o tópico “ENCAPSULAMENTO DE ATRIBUTOS E MÉTODOS COM PACKAGESENCAPSULAMENTO DE CLASSES COM PACKAGES”.

#### HelloInternet

É o nome dado a esta classe. O “abre chaves” marca o início das declarações da classe que são os atributos e métodos. Esta classe só possui uma declaração, a do método **main**, note que um método, ao contrário de C++, só pode ser declarado {internamente} a classe a qual pertence, evitando as confusões sobre “escopo”. Desta forma, todo pedaço de código em **Java** deve pertencer ao abre chaves, fecha chaves da definição de uma classe.

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

#### public

É um qualificador do método que indica que este é acessível externamente a esta classe (para outras classes que eventualmente seriam criadas), não se preocupe com ele agora, apenas declare todos os métodos como **public**. Voltaremos a este assunto em “ENCAPSULANDO MÉTODOS E ATRIBUTOS”.

#### static

É um outro qualificador ou “specifier”, que indica que o método deve ser compartilhado por todos os objetos que são criados a partir desta classe. Os métodos **static** podem ser invocados, mesmo quando não foi criado nenhum objeto para a classe, para tal deve-se seguir a sintaxe: `<NomeClasse>.<NomeMetodoStatic>(argumentos);` ou seja `HelloInternet.main(args);`. Retornaremos a esta explicação mais tarde, por hora você precisa saber que particularmente o método **main** precisa ter essa qualificação porque ele é chamado sem que se crie nenhum objeto de sua classe (a classe `HelloInternet`).

### Curiosidade:

Se você gosta de paradoxos e já conhece um pouco de orientação a objetos, pense que se o método **main** tivesse que ser chamado para um objeto (o que não é o caso) este objeto teria que ter sido criado em algum outro lugar então este lugar seria o início do programa e **main**<sup>4</sup> deixaria de ter esta finalidade.

A linguagem de programação Eiffel adota uma técnica diferente para resolver este problema: todo programa começa com a criação de um objeto (e não mais a chamada automática de **main**), este objeto é chamado **ROOT**, ele pode conter atributos que são inicializados e um método de inicialização, construtor do objeto, que é o início do código do programa.

#### **void**

Semelhante ao **void** C++ ou C, é o valor de retorno da função, quando a função não retorna nenhum valor ela retorna **void**, uma espécie de valor vazio que tem que ser especificado.

#### **main**

Este é um nome particular de método que indica para o compilador o início do programa, é dentro deste método e através das iterações entre os atributos, variáveis e argumentos visíveis nele que o programa se desenvolve.

#### **(String args[])**

É o argumento de **main** e por consequência do programa todo, ele é um vetor de Strings que é formado quando são passados ou não argumentos através da invocação do nome do programa na linha de comando do sistema operacional, exemplo:

Java HelloInternet argumentotexto1 argumentotexto2

No nosso caso, ignoramos a possível passagem de argumentos via linha de comando, retornaremos a este assunto mais tarde.

```
{ ... }
```

“Abre chaves” e “fecha chaves”. Para quem não conhece C ou C++, eles podem ser entendidos como algo semelhante ao **BEGIN END** de Pascal ou Modula-3, ou seja: delimitam um bloco de código. Os programadores Pascal notarão que variáveis locais dos métodos podem ser declaradas em qualquer local entre as chaves. Mas por motivos de clareza do código declararemos todas no início do abre chaves.

```
System.out.println("Hello Internet!");
```

Chamada do método **println** para o atributo **out** da classe ou objeto **System**, o argumento é uma constante do tipo **String**. O **println** assim como **writeln** de Pascal, imprime a **String** na saída padrão e posiciona o cursor na linha abaixo, analogamente **print** não avança linha. Por hora você pode guardar esta linha de código como o comando para imprimir mensagens na tela, onde o argumento que vem entre aspas é a **String** a ser impressa. O ; “ponto e vírgula” separa operações.

```
}
```

Finalmente o fecha chaves termina com a declaração da classe **HelloInternet**.

## **Conclusão:**

---

<sup>4</sup> **main**, do inglês: Principal.

Normalmente o volume de conceitos presentes num primeiro programa de uma linguagem orientada a objetos como **Java** ou Eiffel é grande se comparado com o de um primeiro programa em C ou Pascal. Esses conceitos ainda serão aprofundados e são citados aqui apenas por curiosidade, é normal que você não tenha entendido tudo.

De agora em diante não explicaremos mais como compilar os programas.

## Exercícios:

1-

Experimente fazer modificações no programa **HelloInternet**. Imprima outras mensagens na tela, adicione comentários.

## ATRIBUTOS

No programa anterior, não observamos a criação de nenhum objeto<sup>5</sup>, apenas a declaração da classe **HelloInternet** que continha o método **main**. O nosso programa funcionou, porque o método **main** não precisa de um objeto específico para ser invocado.

Este exemplo declara uma classe (**Circulo**) e em seguida cria um objeto deste tipo em **main** e altera o conteúdo desta variável. Uma classe é parecida com um record de Pascal, a nossa representa um círculo com os atributos **raio** e **x**, **y**, que são coordenadas cartesianas. Note que este objeto não possui métodos ainda.

A classe círculo é especificada em um arquivo separado do arquivo da classe que contém o método **main** (início do programa), um arquivo neste texto é representado pelo retângulo envolvendo um trecho de código, até o tópico “ENCAPSULAMENTO DE CLASSES COM PACKAGES” cada classe será especificada em um arquivo.

É importante entender este exemplo, quando você estudar interfaces gráficas, poderá usar a classe círculo pré-definida na linguagem para desenhar círculos que se movem na tela. Embora não tenhamos explicado com detalhes os tipos básicos da linguagem, usaremos neste exemplo o tipo **float** (real), e nas explicações o tipo **String** e o tipo **int** (inteiro). No final deste tópico forneceremos uma explicação detalhada sobre tipos.

## CÓDIGO

//Classe circulo, arquivo Circulo.Java

```
public class Circulo {  
    //so atributos entre as chaves  
  
    public float raio;           //atributo raio do circulo  
    public float x;  
    //posicoes em coordenadas cartesianas  
    public float y;  
  
}
```

//Classe principal, Arquivo Principal.Java

<sup>5</sup> Na verdade a string delimitada por aspas é um valor literal compilado em um objeto da classe **String** de Java.



```

public class Principal {

    public static void main(String args[]) {
        Circulo umcirc;    //declaracao de uma variavel circulo no metodo main.
        umcirc = new Circulo();    //alocacao dessa variavel
        System.out.println("(" + umcirc.x + "," + umcirc.y + "," + umcirc.raio + ")");
        umcirc.x = umcirc.x + 17;
        System.out.println("(" + umcirc.x + "," + umcirc.y + "," + umcirc.raio + ")");
    }
}

```



(0,0,0)  
(17,0,0)

### Porque os matemáticos não gostam de C, C++, Java e Basic:

A declaração `umcirc.x = umcirc.x + 17` presente em nosso programa deixa os matemáticos doidos pois de você subtrair `umcirc.x` de cada um dos lados da “igualdade” a expressão se torna `0 = 17`. Ocorre que `=` não é o operador de teste de igualdade e sim de atribuição, ele tem a mesma função do `:=` de Pascal a qual os matemáticos adoram. O operador de teste de igualdade em Java é : `==`

### Mais sobre arquivos:

Como pode ser observado, cada arquivo texto do programa está envolvido em uma moldura retangular. Neste caso as duas classes criadas: `Circulo` e `Principal`, estão em arquivos separados. A sequência de alterações em seu diretório ou folder é:

Início:	Após <code>javac AsDuasJuntas.Java</code> :
<pre> -    --- Circulo.java  --- Principal.java </pre>	<pre>  --- Circulo.java  --- Principal.java  --- Principal.class  --- Circulo.class </pre>

O compilador deve ser chamado para ambos arquivos. Ou você pode usar os chamados “wildcards”<sup>6</sup> `javac *.java`. Nesse caso a shell expande `*.java` em uma lista de strings contento todos os arquivos `.java` do diretório e passa essa lista como argumentos para o compilador `javac`.

### Sem Includes:

Se você já programa em alguma outra linguagem provavelmente deve estar com a seguinte

<sup>6</sup> Wild-Card, do inglês: Coringa , carta de baralho. O \* substitui os nomes de todos os arquivos, assim como o coringa pode substituir todas as cartas em jogos de baralho.

dúvida: “Vocês usaram o nome da classe **Circulo** na classe **Principal**, ambas estão em arquivos separados, como o compilador sabe em que diretório está a classe **Circulo**? Não é preciso fazer uma espécie de “include” ou referência para poder usar a classe **Circulo** em outro arquivo?”

A resposta é não. O compilador sabe achar se ela estiver sob o diretório de seu programa. Se você tiver que deixar esta classe em outro diretório então ela deve fazer parte de um package e você terá que importar explicitamente este package, veremos como fazê-lo mais tarde.

### Classe **Circulo** e declaração de atributos:

O primeiro arquivo deste exemplo contém o código da classe **Circulo**, esta classe contém três atributos . A declaração de atributos segue sintaxe semelhante a de C++ (haverão acréscimos a esta sintaxe):

```
EspecificadorModoDeAcesso NomeTipo NomeAtributo;  
public float raio;  
public float x;  
public float y;
```

Todos os atributos pertencentes a classe são do tipo **float** (Ponto flutuante 32-bit IEEE754, veja tabela de tipos básicos). Estão especificados como **public** o que significa que podem ser modificados a partir de uma classe que usa um objeto **Circulo** seja como variável de método ou como atributo (este tipo de modificação será exemplificado na classe **Principal**).

Existem outros especificadores que abordaremos mais adiante, por hora todos os métodos e atributos que criarmos deverão ser **public**. Vale lembrar que na declaração de variáveis simples em métodos, não faz sentido usar o **EspecificadorModoDeAcesso**. Esta classe **Circulo** não possui métodos.

Como exemplo de declaração de variáveis simples tome a declaração de uma variável **Circulo** no método **main**. Seguida de sua alocação: **umcirc = new Circulo(); //alocacao dessa variavel**. Sem a alocação a variável não pode ser usada. Note que os atributos são por default inicializados para zero.

### Classe **Principal** declaração e inicialização de atributos:

A classe **Principal** não possui atributos, porque nenhum objeto desta classe é criado ainda, apenas seu método **main** é chamado. O método **main** declara uma referência para objeto da classe **Circulo**: **Circulo umcirc;**. Note que diferentemente de C++ não são necessários “includes” ou “header files” para poder declarar essa variável de um “tipo” definido pelo usuário e existente em outro arquivo.

Antes da referência ao objeto ser usada, este precisa ser alocado na memória o que é feito através de **umcirc = new Circulo();**. Se você tivesse declarado um objeto da classe **String**, já definida na linguagem, e depois fizesse sua alocação, o código seria semelhante ao seguinte:

```
String umastring;  
umastring = new String(“Valor inicial”);
```

A diferença com a alocação do objeto **Circulo** é que entre os parênteses incluem um argumento “Valor inicial”, isto ocorre porque a linguagem permite que você aproveite a alocação de um objeto para inicializar alguns de seus atributos, isto será explicado em “CONSTRUTORES”. Neste nosso programa, os atributos contidos na área de memória alocada por **new**, são alterados de outra forma.

### Coleta automática de lixo:

“Coleta automática de lixo” é a reciclagem de áreas de memória não mais necessárias ao programa/software. A desalocação do objeto é feita pela linguagem, (“automatic garbage collection”), você não precisa se preocupar com ela como em C++. Após o fechamento de `main`, a memória do objeto `umcirc` já pode ser liberada, o que normalmente não ocorre de imediato, pois o ambiente da linguagem executa um “tread” em baixa prioridade que libera de tempos em tempos os espaços inutilizados de memória, tirando proveito por exemplo de eventuais pausas de interação do usuário com o programa.



*Thread: Por executar em um “tread” entenda paralelamente ou quase paralelamente, voltaremos a este tópico mais adiante em “Threads”.*

### Acesso aos atributos e métodos e alterações dos atributos:

O acesso aos atributos da variável (objeto) `umcirc` deve ser feito usando o nome do objeto e o nome do atributo deste, separados por um ponto: `umcirc.raio = 10.0;` . Note que `raio` sozinho não faz sentido no programa, precisa-se especificar de que objeto se deseja alterar ou obter o raio.

A sintaxe de chamadas de métodos é semelhante a sintaxe descrita acima, só que ao invés de nome do atributo temos o nome do método seguido dos parênteses que podem conter zero ou mais argumentos. Volte ao primeiro programa (`HelloInternet`) e verifique a declaração do método `main`, onde chamamos o método `println()` para o objeto `System.out` .

### Imprimindo variáveis do tipo ponto flutuante (`float`) e inteiro (`int`) na tela:

```
System.out.println("(" + umcirc.x + "," + umcirc.y + "," + umcirc.raio + "');
```

O argumento de `println` que conhecemos é uma única variável do tipo `String`, no entanto o código acima mistura nos argumentos desse método os seguintes elementos: `Strings`: “(”, operadores : +, e variáveis `float`: `umcirc.y`. Fica muito fácil de você entender porque isto funciona se pensar que + é também operador de concatenação de `Strings` e que os argumentos `int` ou `float` e de outros tipos básicos desse método são convertidos para `Strings`.

### Importante aos que não programam em C ou C++, os operadores = e ==:

Em `Java`, `C`, e `C++` o operador = (igual) tem a função de atribuição e o operador == (igual igual) tem a função de comparação entre valores, retornando o valor booleano verdadeiro se estes valores forem idênticos. Alguns podem não achar esta escolha de operadores sensata, principalmente os matemáticos, mas com um pouco de uso da linguagem ela se tornará automática.

### CURIOSIDADE (só para programadores C e C++), eliminação do problema que ocorre em C e C++ com os operadores = e == :

Em `C`, `C++` e também em `Java` é permitido escrever:

```
int i,j,k;
i=0;
j=k=i+3; //final, j vale 3, k vale 3 e i continua valendo 0
```

Essas atribuições em cadeia permitem escrever código mais compacto, elas fazem sentido porque atribuições nessas linguagens possuem valores, assim `k=i+3`; no final tem o valor de `k` que é igual a `i+3`. O valor de `j=...` no nosso exemplo é descartado, ignorado pois não há nada a esquerda desta atribuição. Em `C` e `C++` existe um problema grave que é frequentemente uma cilada para os

programadores Pascal e outros.

O problema é que nessas duas linguagens (C e C++) não existe o tipo boolean, no lugar deste tipo é usado o tipo int (inteiro), onde 1 significa verdadeiro e 0 falso. Assim em C e C++ (mas não em Java) `a==a` tem valor 1 com o significado **dado pelo programador** de true ou verdadeiro, mas `a=1+0`; também tem como resultado o valor 1, mas agora com o significado inteiro, numérico. O compilador não sabe distinguir entre os significados numérico e booleano.

Essa ambiguidade faz com que programadores (C ou C++) não acostumados com esta convenção de operadores `=` e `==`, incorram no erro de escrever `j=1` quando na verdade queriam dizer `j==1`, mas para o compilador ambas as expressões tem valor inteiro de modo que se esse engano ocorrer num teste de parada de um loop, pode ocorrer que ele nunca pare, pois 1 tem o mesmo valor de true.

Java elimina este tipo de erro introduzindo o tipo **boolean** com os valores **true** e **false**, que não tem nenhuma relação com o tipo **int** e qualquer outros tipos.

## TIPOS BÁSICOS OU TIPOS PRIMITIVOS E CONSTANTES OU VALORES LITERAIS

(para nossa sorte os tipos básicos, também conhecidos como tipos primitivos, são os mesmos para qualquer ambiente de programação Java, porque a linguagem é portátil):

<b>char</b>	Caractere UNICODE 16-bit
-------------	--------------------------

O tipo **char** (caractere UNICODE) é representado com 16-bits sem sinal, o que permite endereçar de 0 a 65535. O objetivo desta opção é permitir internacionalização da linguagem, bem como a padronização. Constantes do tipo caractere aparecem entre apóstrofes: 'a', '1', '\$'.

### Tabela de caracteres especiais:

(ao contrário de C/C++, não existe um caractere especial para o som de beep ou bell )

Representação visual:	Função, significado:
<code>\n</code>	Pula linha, linefeed
<code>\r</code>	Retorno de carro
<code>\b</code>	Backspace
<code>\t</code>	Tabulação
<code>\f</code>	Formfeed
<code>\'</code>	Apóstrofe
<code>\"</code>	Aspas
<code>\\</code>	Barra inversa
<code>\u223d</code>	Caractere unicode
<code>\gfa</code>	Octal
<code>\fff</code>	Hexadecimal

<b>boolean</b>	Valor true ou false, diferente representação de C++, sem conversão em outros tipos.
----------------	---

O tipo **boolean** não tem relação nenhuma com outros tipos (coerção). Eliminando problemas que surgiram por exemplo em C++ que usa inteiros para representar valores booleanos. Os possíveis valores são **true** e **false** que são os resultado dos testes lógicos.

```
boolean pertenceAoConjunto; //declara variavel
pertenceAoConjunto = true; //exemplo
```

byte	Inteiro 8-bit , complemento de 2, faixa:-128 até 127
short	Inteiro 16-bit, complemento de 2, faixa:-32768 até 32767
int	Inteiro 32-bit, complemento de 2, faixa:-2147483648 até 2147483647
long	Inteiro 64-bit, compl. de 2, faixa:-9223372036854775808 até 9223372036854775807

Não existem especificadores de tipos como unsigned, todos os tipos “inteiros” tem sinal. **Valores literais:** O que você vê nem sempre é o que você tem: Um valor como **299792458** é considerado int como padrão, se você quer atribuir esta constante a um long, faça o “type cast” explicitamente:

```
long a;
a = (long)299792458; //a recebe PI
```

Ou então use uma terminação em L para indicar que o número deve ser representado como long:

```
long a = 299792458L; //ou L minuscúlo
```

Para indicar valores octais anteceda-os com um zero: **0444** , já para valores hexadecimais anteceda-os com **0X** ou **0x**, exemplo: **0xBCD4**

float	Ponto flutuante 32-bit IEEE754
double	Ponto flutuante 64-bit IEEE754

Um valor como **3.14159265** é considerado double como padrão, se você quer atribuir esta constante a um float, faça o type cast explicitamente:

```
float a;
a = (float)3.14159265; //a recebe PI
```

Ou então usar uma terminação em f para indicar que o número deve ser representado como float:

```
float a=3.14159265f //ou F maiúsculo
```

Expoentes podem ser escritos usando o caracter e ou E: **6,02E23** ou **1.380658e-23** . Onde e-1, significa multiplicado por dez elevado a menos 1 ou **\*0.1**. O separador de casas decimais é o ponto. Apesar de todas estas regras para uso de valores literais, quando dois números de tipo diferentes como double e int são usados em um cálculo do lado direito de uma atribuição, você não precisa fazer o “type cast” desses valores. O compilador promove o número do tipo mais fraco para o tipo mais forte antes de fazer o cálculo (seria bom se funcionasse assim em empresas também onde o funcionário mais fraco seria promovido ao nível do mais forte no projeto e então eles trabalhariam juntos para expressar seu trabalho). Você pode desejar fazer o type cast do resultado, para atribuir a uma variável long por exemplo.

## Exercícios:

1-

Repita o mesmo exemplo só que agora mova o círculo alterando as componentes x e y. Coloque o círculo na posição (1.0, 1.0), através de atribuições do tipo **acirc.x = 1.0;** . Acompanhe todas as modificações do objeto imprimindo seus atributos na tela.

2-

Simplifique o programa anterior retirando o atributo **raio**. Você pode dar o nome de **Ponto** ou **PontoGeometrico** para esta classe. Não se esqueça de compilar, use o compilador como ferramenta para verificar se você aprendeu corretamente a sintaxe da linguagem. Agora escreva a classe **Circulo** contendo um ponto chamado centro.

3-

Reescreva a classe `Circulo` para trabalhar com atributos do tipo `int`.

## ATRIBUTOS E MÉTODOS

Os métodos determinam o comportamento dos objetos de uma classe. Quando um método é invocado, se diz que o objeto está recebendo uma mensagem (para executar uma ação). Como exemplo se meu carro é um objeto e desligar o alarme é um método poderíamos escrever em Java da seguinte forma a chamada para desligar o alarme: `meuCarro.desligarAlarme()`.

Programas complexos formam conjuntos de objetos que trocam mensagens entre si gerenciando inclusive os recursos do sistema.

O programa a seguir exemplifica chamadas de métodos, para tal define um objeto que serve como contador, a implementação representa a contagem no atributo `num` que é um número inteiro. Os métodos são simples: `incrementa` adiciona um ao contador em qualquer estado e `comeca` inicializa a contagem em zero. `Decrementa` faz o oposto de `incrementa`.



### CÓDIGO

//Classe Contador, arquivo Contador.Java



```
public class Contador {

    public int num; //este é o atributo
    //numero do contador

    public void incrementa() {
        num = num + 1; //acesso ao atributo
    }

    public void decrementa() {
        num = num - 1;
    }

    public void comeca(int n)
    //comeca a contar
    {
        num = n;
    }

}
```

//Classe principal, Arquivo Princ.Java

```
public class Principal {

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

```

Contador umcont;
//declaracao de atributo contador

umcont = new Contador();
//alocacao

umcont.comeca(0);
System.out.println(umcont.num);

umcont.incrementa();
System.out.println(umcont.num);
}
}

```



0  
1

### Exercícios:

1-

Defina um método chamado **mostra** para a classe contador. Este método deve imprimir o estado do contador na tela. A implementação deste método depende de onde você está imprimindo o estado do contador? Pense em programas de interfaces gráficas e de linha de comando.

2-

Crie uma classe contador cíclico, exemplo o contador de minutos de relógios digitais: 0,1,2,...,57,58,59,0,1,2,... . A operação modulo ou resto da divisão inteira, tem sintaxe semelhante a de C++:  $a \% b == a - ((\text{int})a/b)*b$ . Não tente usar a estrutura de decisão if ainda.

### Máquinas de estados:

Olhando para o desenho contido no canto direito superior do arquivo da Classe contador, você consegue imaginar este contador como uma máquina de estado? O uso de objetos como máquinas de estados é um dos conceitos que exemplificaremos deste texto.

### Sintaxe de declaração de métodos:

A sintaxe simplificada para a declaração de métodos de uma classe é:

**especificadordeacesso tipoderetorno nomedometodo(listaDeArgumentos) { /\*codigo \*/ }.**

Uma diferença do uso de funções comuns em linguagens não orientadas a objetos e do uso de métodos é que como o método está definido na classe, ele ganha acesso direto aos atributos, sem precisar usar o “ponto”, exemplo **um\_objeto.atributo;**. Lembre-se que as chamadas de métodos em um programa já se referem a um objeto específico, embora os métodos sejam definidos de uma forma geral e parametrizada para toda a classe. Volte agora ao método **main** e verifique sua sintaxe.

## this:

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

Suponha uma classe que possui a seguinte declaração de atributo: `public int qualquer;`. Se quisermos em um método desta classe alterar o atributo **qualquer** para o valor 3, basta escrever `qualquer = 3;`, mas este código escrito dentro de um método da classe que declara **qualquer**, é totalmente equivalente a `this.qualquer = 3;`, sendo o último uma opção mais clara e capaz de eliminar ambiguidades entre os nomes dos atributos de uma classe e os nomes dos argumentos de um dos métodos desta (quando estes nomes forem iguais). O uso de **this** também é válido fazer para chamadas de métodos para o objeto corrente.

## Sintaxe de chamada ou acesso a métodos:

A sintaxe de chamada ou acesso à métodos é semelhante a sintaxe de acesso aos atributos, com exceção dos parênteses que contém a lista de argumentos da função que implementa o método, mesmo que a lista seja vazia eles devem estar presentes: `umcontador.incrementa();`. Primeiro insere-se o nome do objeto e depois a chamada da função, estes são separados por um ponto. Cuidado para não esquecer os parênteses em programas futuros, este é um erro bastante comum.

## Nova versão do programa Circulo:

Agora reapresentaremos o programa **Circulo** baseado no exemplo acima, porém um pouco mais complicado:

### Método move:

O método **move** altera as coordenadas do objeto. O objeto tem suas coordenadas **x** e **y** somadas com os argumentos dessa função. Note que este método representa uma maneira mais segura, clara, elegante de alterar as coordenadas do objeto do que acessá-las diretamente da seguinte forma: `ac.x += dx;`. `ac.y += dy;`. Lembre-se que `ac.x += dx` é uma abreviação para `ac.x = ac.x+dx;`.

### Método mostra:

O método **mostra** imprime na tela, de forma compacta, os atributos do objeto.

## CÓDIGO

```
//Classe circulo

public class Circulo {

    public float raio;

    public float x;
    //posicoes em coordenadas cartesianas
    public float y;

    public void move(float dx,float dy) { //move o circulo de lugar
```



```
this.x += dx;
y += dy;
}

public void mostra()
//imprime na tela estado do objeto
{
System.out.println("(" + x + "," + y + "," + raio + ")");
}

} //fim da declaracao da classe
```

//Classe principal, Arquivo Principal.Java

```
class Principal {

public static void main(String args[]) {
Circulo umcirc;
//declaracao de atributo circulo
umcirc = new Circulo();
umcirc.x = 0;
umcirc.y = 0;
umcirc.raio = 12;
umcirc.mostra();
umcirc.move(10,10);
umcirc.mostra();
umcirc.x = 100;
umcirc.mostra();
}

}
```



(0,0,12)  
(10,10,12)  
(100,10,12)

### Como funcionam no compilador as chamadas de métodos:

É possível imaginar que as definições de métodos ocupam um grande espaço na representação interna dos objetos, mas lembre-se que elas são todas iguais para uma classe então basta manter para cada classe uma tabela de métodos que é consultada no momento da chamada. Os objetos só precisam ter uma referência para esta tabela.

### Exercícios:

1-

Faça as seguintes modificações no programa HelloInternet:

Adicione a declaração e inicialização de variável `String` logo após o abre chaves do método `main`:

```
String nomequalquer;  
nomequalquer = new String("Uma constante do tipo string");
```

Modifique o argumento de `println` para `nomequalquer`. Não use aspas em `nomequalquer`, temos uma variável agora. Execute o programa, qual o resultado?

2-

No programa deste exemplo, crie um método chamado `inicializa` para a classe `Circulo`. Este método deve ter como argumentos um valor para `x`, um para `y` e outro para o `raio`, e deve alterar os atributos inicializando-os com os valores passados. Você pode abstrair o uso desse método como uma maneira de inicializar o objeto de uma só vez embora isto seja feito seqüencialmente. Comente as vantagens desta abordagem, comparando com as outras opções, tenha sempre em mente a questão de segurança e consistência de dados quando avaliar técnicas diferentes de programação.

3-

No programa anterior, verifique que nada impede que você acesse diretamente os valores de `x`, `y` e `raio` e os modifique, como aliás foi feito nos exemplos anteriores. Como se pode criar um número enorme de métodos : `alteraX(float a)`; `moveRaio(float dr)`; seria desejável que somente essas métodos pudessem modificar `x`, `y` e `raio` garantindo que as operações executadas não corrompem o estado do objeto. Você verá que isso é possível em encapsulamento. Por hora, crie esses métodos se preocupando em permitir através chamadas a eles tudo o que for possível fazer com acesso direto aos atributos. Comente também as duas opções equivalentes de implementação abaixo (os nomes são auto explicativos), tenha em mente o número de métodos a serem criados para garantir flexibilidade a classe:

```
umcirculo.multiplicaAtributoX(10);
```

ou

```
umcirculo.setX(umcirculo.getX() * 10);
```

```
//chamadas aninhadas de metodos
```

```
//o resultado de uma chamada compoe o argumento da outra
```

4-

Teste o método `move` com argumentos negativos, exemplo `ac.move(-1.0,-1.5)`;. O resultado é coerente?

5-

“Há uma tendência em definir o maior número de métodos em uma classe, porque nunca se pode prever exatamente o seu uso em programas futuros”. Comente esta frase, tendo em vista o conceito de portabilidade. Você já é capaz de citar outras medidas que tornem suas classes mais portáveis, flexíveis e genéricas? Leia o exercício anterior e o exercício 3.

## MÉTODOS QUE RETORNAM VALORES.

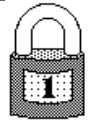
Até agora só havíamos visto métodos com valor de retorno igual a `void`. Um método, assim como uma função comum, pode retornar um único elemento de qualquer tipo, inclusive os definidos pelo usuário ou seja: objetos. Sendo assim, sua chamada no programa se aplica a qualquer lugar onde se espera um tipo igual ou equivalente ao tipo do seu valor de retorno, seja numa lista de argumentos de outro método, numa atribuição ou num operador `+` em `System.out.println( variavel + chamada_de_metodo_que_retorna_valor)`;

## Classe trava:

O estado da trava é representado no atributo `public boolean travado`; e pode ser obtido através de uma chamada ao método `estado()`. Este exemplo é bastante simples e não tem a pretensão de ser uma implementação completa de uma classe que modele uma trava que possa ser usada para evitar o acesso a um arquivo, por exemplo.

### CÓDIGO

```
public class Trava {  
  
    public String quem; //quem travou ou destravou por ultimo  
  
    public boolean travado;  
  
    public void trave(String q)  
    {  
        this.travado = true;  
        this.quem = q;  
    }  
  
    public void destrave(String q)  
    {  
        this.travado = false;  
        this.quem = q;  
    }  
  
    public boolean estado()  
    //estado(true/false) do objeto  
    {  
        return travado;  
    }  
}
```



```
//Classe principal, Arquivo Principal.Java
```

```
public class Principal {  
  
    public static void main(String args[]) {  
        Trava umatrava;  
  
        umatrava=new Trava();  
        umatrava.trave("ProgramaPrincipal");  
        System.out.println(umatrava.estado());  
        umatrava.destrave("ProgramaPrincipal");  
        System.out.println(umatrava.estado());  
    }  
}
```

```
}  
  
}
```



true  
false

## //COMENTARIOS

A classe trava é demasiadamente simples, mas se não o fosse não estaria na parte introdutória deste tutorial. Esta classe acaba tendo a funcionalidade de uma variável booleana e o custo de um objeto, mas neste ponto cabe a você melhorar este modelo para representar uma trava de arquivo (armazena o nome e “path” deste) ou uma trava de um objeto, evitando que este seja alterado.

## Exercícios:

1-

Melhore a classe Contador de “ATRIBUTOS E MÉTODOS”, defina um método que imprime o contador na tela. Se você estivesse fazendo um programa para funcionar em uma interface gráfica com o usuário este método para imprimir na tela seria o mesmo? Definir um método que retorna uma cópia do valor atual do contador garante maior portabilidade? Por quê? Para aprender a retornar valores consulte este tópico.

2-

Insira um novo método na classe Trava que tenha a função de alternar entre os estados do objeto, independente do estado atual (travado ou destravado). Não use a estrutura decisão if ainda, use o operador not que serve para negar um valor booleano: `a = !valorbooleano;` // `!true==false` ou `true == !false`.

3-

Implemente a checagem do estado de overflow no exemplo do contador, considere o uso de um método retornando um valor do tipo `boolean`. Considere a adição de um atributo.

## COMPARAÇÃO COM UM PROGRAMA EM PASCAL

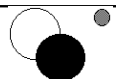
Este exemplo de programa em Java é semelhante ao exemplo da classe Circulo de “ATRIBUTOS”, ele será comparado com um programa escrito em Pascal, além disso introduzimos um novo método chamado `inicializa` e métodos `public float getRaio(void);` e `void setRaio(float a)`. `inicializa` coloca o ponto nas coordenadas passadas como seus argumentos. A introdução deste método prepara a explicação sobre construtores dada no próximo tópico. Para melhor entendimento leia os exemplos anteriores com a classe Circulo.



**CÓDIGO Java:** Programa exemplo círculo, baseado no exemplo de ATRIBUTOS:

```
//Classe circulo
```

```
public class Circulo {
```



```

public float raio,x,y;

public void inicializa(float ax,float ay,float ar)
//garante o estado do objeto
{
    this.x = ax; this.y = ay; this.raio = ar;
}

public void setRaio(float a)
{
    this.raio = a;
}

public float getRaio()
{
    return this.raio;
}

public void move(float dx,float dy) {
    this.x += dx; this.y += dy;
}

public void mostra()
{
    System.out.println("(" + this.x + "," + this.y + "," + this.getRaio()+")");
}

}

```

//Classe principal, Arquivo Principal.Java

```

public class Principal {

    public static void main(String args[]) {
        Circulo ac;
        ac = new Circulo();
        ac.inicializa(0,0,10);
        ac.mostra();
        ac.move(1,1);
        ac.mostra();
        ac.x = 100;
        ac.setRaio(12);
        ac.mostra();
    }
}

```

}

### Disposição dos métodos e atributos na declaração de uma classe:

Em uma declaração de uma classe normalmente se coloca a declaração de métodos depois da declaração dos atributos, porém podemos fazer intercalações ou adotar qualquer ordem que nos convenha.

Uma boa técnica de programação, que aconselhamos você adotar em seus programas é usar linhas de comentários para delimitar uma área do código de suas classes que englobe tudo o que interessa a um usuário desta. Nestes programas exemplos praticamente tudo é de interesse de um usuário (cliente) de sua classe, mas nos exemplos mais avançados isto não será verdade.

### Comentários:

Observe que o método `mostra` chama o método `public float getRaio()` que é da mesma classe. Fica claro da definição de `mostra` que `this.getRaio()` se aplica ao mesmo objeto instanciado que recebeu a chamada de `mostra`. Isto foi feito somente para revelar que chamadas aninhadas de métodos também são permitidas, pois nesse caso esta chamada de método poderia ser substituída pelo próprio atributo `raio`, o que seria mais eficiente.

### Programação orientada a objetos e interfaces gráficas com o usuário:

Existem “libraries” de classes que permitem o programador C++ desenvolver aplicações para ambientes como o Microsoft Windows® de uma maneira bastante abstrata, este é um exemplo claro de reuso de código, afinal não é preciso saber de detalhes da interface para programar nela.

Na segunda parte falaremos sobre a “Java Application Programming Interface” que permite programar de maneira bastante abstrata sistemas de interfaces gráficas com o usuário seja para aplicações para a Internet (rodando em browsers) ou para sistemas como o Windows ou Mac/Os e X-Windows .



(0,0,10)  
(1,1,10)  
(100,1,12)

PASCAL: Um programa parecido só que em PASCAL:

```
PROGRAM Comparacao;  
{COMPARACAO COM UM PROGRAMA Java}  
  
TYPE Circulo=RECORD  
    x:real;  
    {COORDENADAS X E Y}  
    y:real;  
    r:real;  
    {somente dados}
```

```

        END;

var ac:circulo;
    leitura:integer;

PROCEDURE Inicializa(var altereme:Circulo;ax,by,cr:real);
{COLOCA O CIRCULO EM DETERMINADA POSICAO}
BEGIN
    altereme.x:=ax;
    altereme.y:=by;
    altereme.r:=cr;
END;

PROCEDURE Altera_Raio(var altereme:Circulo;ar:real);
{ALTERA O RAO DO CIRCULO}
BEGIN
    altereme.r:=ar;
END;

FUNCTION Retorna_Raio(copieme:Circulo):real;
BEGIN
    Retorna_Raio:=copieme.r;
END;

PROCEDURE Move(var altereme:Circulo;dx,dy:real);
{MODE AS COORDENADAS X E Y ACRESCENTANDO DX E DY}
BEGIN
    altereme.x:=altereme.x+dx;
    altereme.y:=altereme.y+dy;
END;

PROCEDURE Mostra(copieme:Circulo);
{MOSTRA O CIRCULO NA TELA}
BEGIN
    writeln('X:',copieme.x,' Y:',copieme.y,' R:',copieme.r);
END;

BEGIN
{TESTES}
    Inicializa(ac,0.0,0.0,10.0);
    Mostra(ac);
    Move(ac,1.0,1.0);
    Mostra(ac);
    ac.x:=100.0;
    Altera_Raio(ac,12.0);
    Mostra(ac);
    read(leitura);

```

END.



X: 0.0000000000E+00 Y: 0.0000000000E+00 R: 1.0000000000E+01  
X: 1.0000000000E+00 Y: 1.0000000000E+00 R: 1.0000000000E+01  
X: 1.0000000000E+02 Y: 1.0000000000E+00 R: 1.2000000000E+01

## //COMENTARIOS JAVA:

As classes em **Java** são compostas de atributos e métodos. Para executar uma ação sobre o objeto ou relativa a este basta chamar um método : **ac.mostra()**; O método não precisa de muitos argumentos, porque é próprio da classe e portanto ganha acesso aos atributos do objeto para ao qual ela foi associado:

```
public float getRaio(void)
{ return raio; //tenho acesso direto a raio. }
```

{ COMENTARIOS PASCAL: }

Em Pascal os procedimentos e os dados são criados de forma separada, mesmo que só tenham sentido juntos. A junção entre os dados e procedimentos se dá através de passagem de parâmetros. No caso de uma linguagem procedural, o que normalmente é feito se assemelha ao código seguinte: **Move(ac, 1.0,1.0);**. Nesse caso **AC** é um “record”, algo semelhante ao struct de C (não C++). **Move**, acessa os dados do “record” alterando os campos. O parâmetro é passado por referência e o procedimento é definido a parte do registro, embora só sirva para aceitar argumentos do tipo **Circulo** e mover suas coordenadas.

## Segurança:

Em ambos programas (Pascal, **Java**) o programador pode obter acesso direto aos dados do tipo definido pelo usuário: **ac.x := 100.0;** (Pascal) ou **ac.x = 100.0;** (Java).

Veremos em “ENCAPSULAMENTO” maneiras de proibir este tipo de acesso direto ao atributo, deixando este ser modificado somente pelos métodos. Isto nos garante maior segurança e liberdade pois podemos permitir ou não o acesso para cada atributo de acordo com nossa vontade.

## Eficiência:

Alguém pode argumentar que programas que usam bastante chamadas de métodos podem se tornar pouco eficientes e que poderia ser melhor obter acesso direto aos dados de um tipo definido pelo usuário ao invés de passar por todo o trabalho de cópia de argumentos, inserção de função na pilha, etc.

Em verdade não se perde muito em eficiência, por que tal metodologia de programação nos leva a organizar o código de maneira mais compacta. E além disso muitas vezes não se deseja permitir sempre o acesso direto aos dados de um tipo definido pelo usuário por razões de segurança.

## Exercícios:

1-



Implemente outros métodos do estilo `public void setRaio(float a)` e `float getRaio()` para os atributos X e Y. Mas leve o seguinte padrão de programação Java em conta. Para uma data variável ou atributo X os métodos altera e retorna devem ser chamados na verdade de `setX()` and `getX()`. Veja o tópico anterior chamado “Acessores de atributos” para detalhes sobre esse padrão ou convenção.

2-

Faça um programa simples para testar uma classe que representa um mouse e que fornece a posição na tela, os indicadores de estado dos botões e os métodos: `clica_botao()`; `move(float dx, float dy)`; . Não é preciso fazer a ligação do objeto com o mouse real, embora o leitor interessado possa encontrar na segunda parte subsídios para esta tarefa.

Seu mouse deve ser capaz de caminhar para qualquer lugar da tela através de chamadas de métodos, não deve ultrapassar os limites estabelecidos, deve indicar o botão está pressionado ou não através de um método semelhante ao `mostra()` deste exemplo. O mouse **deve** ter de 1 botão. Você pode substituir método `move(float dx, float dy)` por combinações de `move_x(float dx)`; e `move_y(float dy)`;

Você pode achar o fato de o mouse ser obrigado a ter um botão um tanto estranho, mas não é. Os mouses dos machintoshes têm um botão, alguns mouses de pc's têm dois botões e mouses de workstations têm três botões. Para seu programa ser portátil ele deve contar apenas com a existência de um botão e isso vale a pena quando se fala em portabilidade. O que tem sido usado em casos extremos é a combinação de pressionamento dos botões e pressionamento de teclas para diferenciar certas ações.

3-

Verifique que em `main()` você pode modificar o atributo x do objeto da classe `Circulo` da seguinte forma: `a.x = 12.2`; . Isto pode não ser muito útil, imagine-se criando uma library (em Java package) que implementa a classe `Circulo` e uma série de métodos relacionados, por certas razões você gostaria que o usuário se limitasse ao uso da interface do objeto, como fazê-lo será explicado em encapsulamento. Por hora, apenas crie métodos que sirvam como alternativas a este tipo de acesso direto.

## CONSTRUTORES

Construtores são métodos especiais chamados pelo sistema no momento da criação de um objeto. Eles não possuem valor de retorno, porque você não pode chamar um construtor para um objeto, você só usa o construtor no momento da inicialização do objeto. Construtores representam uma oportunidade de inicializar seus dados de forma organizada, imagine se você esquece de inicializar corretamente ou o faz duas vezes, etc.

Um construtor tem sempre o mesmo nome da classe a qual pertence. Para a classe `String`, pré-definida na linguagem o construtor tem a forma `String(“Constante do tipo String”)`; com o argumento entre aspas que é especificado pelo programador como um valor literal de string. Ele seria chamado automaticamente no momento da criação, declaração de uma `String`, sem necessidade de uso do nome do construtor como método, apenas dos argumentos:

```
String a;  
a=new String(“Texto”); //alocacao e inicializacao atraves do construtor  
a.mostra(); //mostra so pode ser chamada depois do construtor
```

Nos exemplos anteriores também usávamos construtores no momento de inicializar nossos objetos, só que eles não possuíam argumentos.

Existem variações sobre o tema que veremos mais tarde: sobrecarga de construtor, “copy constructor”, construtor de corpo vazio. O exemplo a seguir é simples, semelhante aos anteriores, preste atenção no método com o mesmo nome que a classe, este é o construtor:

## CÓDIGO



```
//Classe ponto

public class Ponto {

    public float x,y;

    public Ponto(float ax,float ay) // sempre omita o valor de retorno!
    //garante o estado do objeto
    {
        this.x = ax; this.y = ay;
    }

    public void move(float dx,float dy)
    {
        this.x += dx; this.y += dy;
    }

    public void mostra()
    {
        System.out.println("(" + this.x + "," + this.y + ")");
    }

}
```

```
//Classe principal, Arquivo Principal.java
```

```
public class Principal {

    public static void main(String args[]) {
        Ponto ap;
        ap = new Ponto((float)0.0, (float)0.0);
        ap.mostra();
        ap.move(1,1);
        ap.mostra();
        ap.x = 100;
        ap.mostra();
    }

}
```



(0,0)  
(1,1)  
(100,1)

## //COMENTARIOS:

Note que com a definição do construtor, você é obrigado a passar os argumentos deste no momento da alocação do objeto. Se você precisa ter a opção de não passar esses valores ou passar outros, as possíveis soluções serão dadas em “POLIMORFISMO, CLASSES ABSTRATAS”.

(float)0.0 indica que é para ser feita a conversão de 1.0 para ponto flutuante. 1.0 sozinho é considerado double. (int)1.0 é igual a 1. (int) 2.3 é igual a dois. Esta operação indicada por (nometipo)tipo\_a\_ser\_convertido é também chamada de “type cast”.

A ocorrência de rotinas de criação de objetos em diversos locais de um programa é muito comum. Objetos podem ser criados dentro de estruturas condicionais, armazenados em arquivos, passados como parâmetros, inseridos em estruturas dinâmicas dentro de outros objetos, etc.

## Exercícios:

1-

Um método pode chamar outro método da mesma classe. Parta do exemplo de e crie um construtor que chama o antigo método `inicializa(float a,float b)` repassando os argumentos do construtor:

```
ponto(float a, float b)
{ inicializa(a,b); }
```

Na chamada de `inicializa()` fica implícito que ela se aplica ao objeto cujo construtor foi chamado. Isto é válido também para outros métodos que chamam métodos, ou seja, não é necessário o operador `identificador.inicializa(a,b);`, veja o uso de `this`.

Este exercício é útil para mostrar outro recurso de Java, o ponteiro `this`. `this` é uma palavra reservada e dentro de qualquer método `this` é um “ponteiro” para o objeto em questão, então o código descrito acima poderia também assumir a seguinte forma equivalente:

```
ponto(float a, float b)
{ this.inicializa(a,b); } //Verifique!
```

É lógico que neste exemplo `this` não tem muita utilidade, mas existem casos onde um objeto precisa passar seu “ponteiro” para alguma função que o modifica, ou fica possuindo uma referência para ele, então usa-se `this`. Veremos outras aplicações mais adiante.

2-

Introduza mensagens no construtor tipo: `System.out.println(“Objeto instanciado”);` introduza trechos parecidos em outros pontos de seu programa. O objetivo é acompanhar visualmente a sequência de criação e modificação dos objetos.

3-

Esta classe `Ponto` pode ser adaptada para funcionar como representação de vetores em duas dimensões, para tal forneça outros métodos úteis: métodos para tornar o vetor unitário, retornar o módulo do vetor, a componente `x` e `y`. Para tal você terá que fazer uso de `Math.cos(double a);`, `Math.sin(double a);` e `Math.sqrt(double a);`, respectivamente o cosseno, o seno e a raiz quadrada de um ponto flutuante de dupla precisão. Não é necessário fazer nada semelhante a um `include` de C++

+ para usar esses métodos, basta adicioná-los ao seu código, outro método útil é `Math.max(a,b)`; que retorna o maior valor entre `a` e `b` (`float`, `double`, `int`, `long`, etc). Use “type cast” explicado nos comentários deste exemplo.

4-

Crie uma classe `reta` que tem como atributos dois objetos da classe `ponto`. Dica: **Não use construtores**, use funções do tipo `inicializa()`, já apresentadas. Quando seu programa ficar pronto acrescente métodos para esta `reta` tais como inclinação, coeficiente linear, etc. Para acrescentar construtores leia mais adiante. Existem maneiras mais eficientes e compactas de representar uma `reta`, porém faça como foi sugerido, neste caso o objetivo não é eficiência.

## CONSTRUTORES E AGREGAÇÃO

Este exemplo é o resultado do exercício anterior (exercício 4 tópico “CONSTRUTORES”), ele cria uma classe `Reta` com dois atributos da classe `Ponto`. Ou seja você estará reutilizando a classe `Ponto` na classe que representa uma `Reta`, a forma como essa reutilização de código ocorre é chamada de **agregação**.

O nome reutilização de código é bastante apropriado, pois a classe `Reta` estará utilizando todos os métodos escritos para a classe `Ponto`. Em casos como este é comum dizer que a classe `Reta` é cliente da classe `Ponto` em uma analogia com as relações comerciais.

Este programa busca chamar sua atenção para o fato de que objetos são alocados dinamicamente e que caso você se esqueça de alocá-los, eles ficam possuindo o valor `null`. Obter acesso a uma variável com valor `null` é um erro que geralmente é verificado em tempo de execução.

Nos programas anteriores alocar os objetos era fácil, mas agora que temos objetos funcionando dentro de outros objetos há necessidade de adotar técnicas melhores. Veremos a seguir duas alternativas para alocar os atributos `Ponto` presentes na classe `Reta`:

### Alternativa 1:

Primeiro aloca-se o objeto da classe `Reta`, depois chama-se para cada atributo da classe `Ponto` a rotina de alocação:

```
Reta r1;  
r1 = new Reta();  
r1.ponto1 = new Ponto((float)1.0, (float)2.0);  
r1.ponto2 = new Ponto((float)3.0, (float)4.0);
```

Esta alternativa é em muitos casos pior que a seguinte.

### Alternativa 2:

Passamos para o construtor da classe `Reta` a tarefa de alocar e inicializar os atributos de forma coerente. Os argumentos do construtor da classe `Reta` passam a conter valores úteis para a chamada dos construtores de seus atributos. Esta alternativa é executada pelo programa a seguir:

### CÓDIGO

```
//Reutilize o arquivo da classe Ponto definida anteriormente.  
//Classe Reta  
  
class Reta {
```



```

public Ponto a,b;

public Reta(float ax, float ay, float bx, float by) //construtor com argumentos
{
    a = new Ponto(ax, ay); //chamadas dos construtores da classe Ponto
    b = new Ponto(bx, by);
}

public void mostra()
{
    a.mostra();
    b.mostra();
}
}

```

```

//Classe principal, Arquivo Principal.java

class Principal {

    public static void main(String args[]) {
        Reta ar;
        ar=new Reta((float)0.0,(float)0.0,(float)1.0,(float)1.0);
        ar.mostra();
    }
}

```

Alguns podem argumentar que esta maneira de representar uma reta não é muito eficiente, mas não é do escopo deste texto tratar de problemas dessa natureza, o que aliás complicaria muito o código, desviando-nos do objetivo principal: simplicidade na apresentação de conceitos de orientação a objetos. O leitor deve ampliar os modelos aqui sugeridos em aplicações na sua área de interesse. Como ainda faltam conceitos importantes para serem apresentados este tipo de aplicação deve ser feita em paralelo com a leitura.

### Exercícios:

1-

Implemente um programa que use a mesma lógica do exemplo anterior para criar uma classe composta de outras . Você estará usando agregação. Por exemplo um triângulo precisa de três pontos para ser definido ou uma pessoa possui um endereço residencial.

2-

Uma implementação mais eficiente da classe **Reta** seria feita armazenando apenas um coeficiente angular e um linear, mas esta reta deveria prover também um construtor que aceitasse dois pontos como argumentos. Como você não aprendeu sobrecarga de construtores (definir vários construtores), use só um construtor que tem coeficiente angular e linear como argumentos e

implemente esta nova classe **reta** (sem usar agregação agora).

3-

Defina um método para a classe **Reta** que retorna o ponto de intercessão com outra **reta**: `public Ponto intercessao(Reta a);`. Não se esqueça de tratar os casos degenerados, tais como retas paralelas e coincidentes, use por exemplo mensagens de erro ou valor de retorno igual a `null`. Verifique que um método de uma **reta** recebe outra **reta** (mesmo tipo) como argumento. Dentro do método os atributos do argumento **a** devem ser acessados do seguinte modo:

```
ap1.x;
```

Mais tarde, em tratamento de exceções, veremos maneiras melhores de lidar com esses casos degenerados.

4-

Defina um método chamado **move** para a classe **Reta**, lembre-se de mover os dois pontos juntos (a inclinação não deve mudar).

5-

Defina um método `public void gira(tipo x angulo);` para a classe **Reta**. Este método recebe um ângulo como argumento, você pode optar por representar o ângulo em radianos (**float**) ou criar a classe ângulo (graus, minutos, segundos). Resolva também o problema da escolha do ponto em torno do qual a **reta** deve ser girada (você pode usar mais de um argumento). Use funções matemáticas (seno cosseno) e outras descritas no exercício 3 anterior.

## DESTRUTORES OU “finalizers”

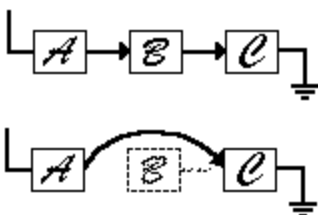
A presença de coleta automática de lixo torna o conceito de destrutores um pouco diferente de seus equivalentes em outras linguagens orientadas a objetos. Em **Java** destrutores são métodos chamados pelo sistema quando a memória de um objeto está para ser liberada pelo coletor automático de lixo (não quando está para ser coletada).

A sintaxe dos destrutores é a seguinte:

```
protected void finalize() {  
    //codigo para arrumar a casa, antes que o objeto seja apagado  
}
```

Note que o que caracteriza o construtor é ter o mesmo nome da classe, já o destrutor é caracterizado por possuir o nome **finalize**. Você pode chamar o destrutor, mas isso não implica que o objeto será deletado. Ao contrário dos construtores, os destrutores não tem argumentos, mas possuem valor de retorno que é igual a **void**.

Os destrutores são muito úteis para “limpar a casa” quando um objeto deixa de ser usado. Um exemplo bastante comum do uso de destrutores é o de um objeto que lida com arquivos que devem ser fechados quando o objeto for destruído. Existem outros casos onde um objeto deve comunicar aos outros objetos que será inutilizado, destruído, ou seja : sairá do programa.



Em **Java**, que possui coleta automática de lixo, um objeto passa a não existir mais no programa quando nenhuma variável faz referência a ele. Nesse ponto, ele é armazenado no coletor automático de lixo, onde receberá o tratamento adequado. Um nó de uma lista ligada pode ser apagado simplesmente fazendo o nó anterior apontar para o posterior a ele. Os programadores não acostumados com esse conceito precisam ouvir a seguinte frase: “Você não é obrigado a liberar explicitamente a

memória de um objeto como é feito em C++ e outras linguagens”.

Se estivéssemos por exemplo em C++, que não possui coleta automática de lixo, o destrutor seria chamado sempre que a memória de um objeto fosse desalocada, o que é feito pelo programador através de uma chamada a `delete nomeDoObjeto();` .

Em **Java** a liberação de memória que é feita pelo coletor automático de lixo que é executado de modo assíncrono com o restante do programa, ou seja você não pode contar com o fato de que o destrutor será chamado imediatamente após o momento em que seu objeto sai de escopo, a não ser que o programa seja executado com o modo assíncrono do coletor automático de lixo desligado (`java -noasyncgc NomePrograma`).

## PONTEIROS, “POINTERS”, REFERÊNCIAS E OBJETOS

Alguém pode ter achado estranho que não foram discutidos ponteiros em **Java**, ocorre que eles não existem nessa linguagem. Existem estudos que afirmam que erros com ponteiros são um dos principais geradores de “bugs” em programas, além disso com todos os recursos que temos não precisaremos deles.

Os programadores acostumados ao uso de ponteiros (e aos erros decorrentes desse uso), acharão muito natural e segura a transição para **Java** onde passarão a usar principalmente vetores e classes. A estruturação de seu código deverá agora ser feita em um modelo que se baseia no uso de objetos (vetores e **Strings** também são objetos). Objetos superam a representatividade obtida com records, funções isoladas e ponteiros.

De certo modo você estará usando referências, mas de forma implícita. Por exemplo: objetos são alocados dinamicamente com `new`, eles são referências ou ponteiros para posições na memória, mas a linguagem mascara este fato por razões de segurança. Como objetos são ponteiros (só que transparentes para você), nos depararemos com o problema de reference aliasing quando discutirmos cópia de objetos com outros objetos como atributos.

## PASSAGEM POR REFERÊNCIA

Linguagens como Pascal ou C criam meios de passar parâmetros por valor ou por referência. Como **Java** não possui ponteiros, a passagem por referência deve ser feita através de objetos. Se o parâmetro já é um objeto, então a passagem dele é obrigatoriamente por referência.

No caso de tipos simples, podemos passá-los dentro de vetores que são objetos, ou então criar classes para embalar, empacotar estes tipos. Dada a necessidade destas classes, elas já foram definidas na linguagem. São classes definidas para conter cada tipo básicos e permitir certas conversões entre eles, falaremos destas classes conforme necessitarmos de seu uso. As vezes estas classes são chamadas de “wrappers”.

### Exercícios:

1-

```
class X {  
  
    public void troca(X valores)  
    {  
        int a;
```

```

        a = val;
        val = valores.val;
        valores.val = a;
    }
    //imprima os atributos de um objeto OBJ da classe X
    //chame o metodo troca de outro objeto, usando OBJ como argumento
    //verifique a alteracao nos atributos desse argumento

```

O código acima mostra como definir um método que usa a existência implícita de ponteiros na linguagem para criar um método que troca os atributos de dois objetos da mesma classe. Implemente este método e faça os testes sugeridos nos comentários desse código.

## VETORES E MATRIZES

Vetores são objetos, eles possuem papel importante no estilo de programação desta linguagem que exclui ponteiros. Por serem objetos, vetores são obrigatoriamente alocados de maneira dinâmica. O exemplo a seguir aloca um vetor de inteiros com três posições, seguindo uma sintaxe semelhante a de alocação de objetos:

### CÓDIGO

```

class VetorTest {
    public static void main (String args[]) {
        int vetor[] = new int[3];
        vetor[0] = 0;                               //indexacao semelhante a C , C++
        vetor[1] = 10;
        vetor[2] = 20;
        System.out.println(vetor[0] + " " + vetor[1] + " " + vetor[2] + " ");
    }
}

```



0 10 20

### Resumo da sintaxe de vetores:

```

int a[]; //declara vetor de inteiros a
a = new int[10]; //aloca vetor a com dez posicoes
//as duas linhas anteriores podem ser abreviadas por:
int a[] = new int[10];
//alem disso se voce quiser inicializar o vetor a, ja' na declaracao:
int a[3] = {0,10,20};

```

O análogo para matrizes é:

```

int a[][]; //declara matriz de inteiros a
a = new int[3][3]; //aloca matriz 3x3, 9 celulas
//as duas linhas anteriores podem ser abreviadas por:
int a[] = new int[3][3];

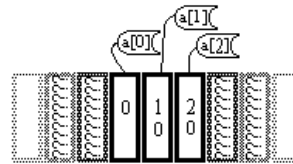
```



```
//alem disso se voce quiser inicializar a matriz a ja na declaracao:  
int a[3][3] = {{0,10,20},{30,40,50},{60,70,80}};
```

Em métodos, argumentos e valores de retorno que são vetores, são escritos da seguinte forma:  
`int[]` , ou `tipo[]` nomedavariavel //no caso de argumentos.

Diagrama do vetor:



Perceba que a faixa útil do vetor vai de 0 até (n-1) onde n é o valor dado como tamanho do vetor no momento de sua criação, no nosso caso 3. O mesmo ocorre com matrizes. Esta convenção pode confundir programadores Pascal onde a indexação vai de 1 até n.

Java checa se você usa corretamente os índices do vetor. Se ocorrer um acesso ao `vetor[i]` onde i é um índice inválido (fora da faixa de valores permitidos), você receberá uma mensagem parecida com: `java.lang.ArrayIndexOutOfBoundsException: #`. Retornaremos ao assunto desta mensagem mais adiante, ela é uma exceção gerada pelo código que acompanha a linguagem.

Existe um atributo muito útil quando se trabalha em um vetor de dados:

```
a.length; //armazena o numero de elementos do vetor a
```

Declarar um vetor de objetos, por exemplo objetos da classe `Ponto`, não implica que os objetos de cada posição do vetor já estarão inicializados, para inicializar os elementos, siga seguinte sintaxe:

```
Ponto a[];           //declaracao, todas as posicoes com null  
a = new Ponto[3];    //alocacao  
for (int i = 0; i < a.length(); i++) { a[i] = new Ponto(0,0); } //inicializacao
```

(o código acima representa um dos usos de vetores no lugar de ponteiros)

## Exercícios:

1-

Escreva um programa simples que toma um vetor de preços e um de descontos (50%=.5), e altera o vetor de preços de modo que estes produtos já incluam os descontos no seu valor de venda.

Exemplo:

Preços :	40,5	30,3	12,6	100
Descontos:	.5	.3	.2	.5
Vetor calculado:	20.25	9.09	2.52	50

A linguagem Java será muito usada no comércio da internet, tabelinhas assim, usadas para cálculos de preços de produtos já estão se tornando comuns. Os nomes que você pode dar as classes e métodos são:

Classe:	TabelaPrecos
Atributo:	boolean ComDesconto;
Atributo:	double[] Precos;
Método:	void AplicaDescontos(int[] descontos);

## COPIA, COMPARAÇÃO E DETERMINAÇÃO DA CLASSE EM OBJETOS

Objetos são implicitamente referências, portanto sua cópia (através do operador =) está sujeita ao problema de “reference aliasing” e efeitos colaterais. A comparação de objetos através do operador == tem o significado da comparação de referências, ou seja ela verifica se os objetos<sup>7</sup> compartilham o mesmo espaço alocado na memória.

Observe que com as operações conhecidas até agora, não conseguimos comparar dois objetos quanto ao conteúdo a não ser que comparemos atributo por atributo, o mesmo vale para a cópia.

Seria útil dispor de um conjunto de operações de igualdade, desigualdade e cópia que se aplicasse ao conteúdo dos objetos e não ao endereço de memória de suas variáveis. Uma outra necessidade seria verificar a classe de um objeto em tempo de execução. Antes de ver a solução procure sentir o problema no exemplo a seguir:

Usaremos o arquivo da classe **Ponto**, já apresentado em “CONSTRUTORES”, mas com uma modificação no método **mostra**. No lugar de **mostra** criaremos um método chamado **public String toString()**. Este método é padrão em muitas classes e deve ser definido de modo a retornar uma **String** descritiva do objeto. Fazendo isto você pode concatenar uma variável **Ponto** com uma **String** no argumento do método **System.out.println(meuPonto1 + “ Na vizinhanca de “ + meuPonto2);**. Esta decisão de implementação é certamente mais genérica que **mostra**, visto que nem sempre estaremos imprimindo através de **System.out.println()**, por exemplo na segunda parte ocorrerão casos em que temos que “pintar” **Strings** em áreas especiais na tela.

### **CÓDIGO** Apresentação do problema

```
//Classe ponto

class Ponto {

public float x,y;

public Ponto(float ax,float ay) { //omita o valor de retorno! garante o estado do objeto
    this.x = ax; this.y = ay;
}

public void move(float dx,float dy) {
    this.x += dx; this.y += dy;
}

public String toString() {
    return "(" + this.x + "," + this.y + ")"; //(x,y)
}

}
```

```
//Classe principal, Arquivo Principal.java
```

```
class Principal {
```

<sup>7</sup> Strings e vetores são objetos de classes pré definidas na linguagem, portanto as afirmações feitas aqui se aplicam a eles também.

```

public static void main(String args[]) {

//preparacao das variaveis copia de objetos
    Ponto pOriginal, pAlias, pCopia;
    pOriginal = new Ponto((float)0.0,0.0f); //(float)0.0 ou 0.0f; 0.0 eh double
    pAlias = pOriginal; //copiando atraves de atribuiçao
    pCopia = new Ponto(pOriginal.x, pOriginal.y); //copiando atributo por atributo

//resultados
    System.out.println("Original:" + pOriginal);
    System.out.println("Alias:" + pOriginal);
    System.out.println("Modificando Alias.x para 2");
    pAlias.x = 2.0f;
    System.out.println("Veja como o original ficou:" + pOriginal);
    System.out.println("pCopia nao se modifica:" + pCopia);

//comparacao de objetos
    System.out.println("Original==Alias:" + (pOriginal == pAlias) );
    System.out.println("Copia==Original:" + (pCopia == pOriginal) );
    System.out.println("Deixando atributos de Copia iguais aos de Original");
    pCopia.x = 2;
    System.out.println("Copia==Original:" + (pCopia == pOriginal) );
    System.out.println("Original.x==Copia.x:" + (pCopia.x == pOriginal.x) );
    System.out.println("Original.y==Copia.y:" + (pCopia.y == pOriginal.y) );

}

```



```

Original:(0,0)
Alias:(0,0)
Modificando Alias.x para 2
Veja como o original ficou:(2,0)
pCopia nao se modifica:(0,0)
Original==Alis:true
Copia==Original:false
Deixando atributos de copia iguais aos de Original
Copia==Original:false
Original.x==Copia.x:true
Original.y==Copia.y:true

```

## //COMENTARIOS

### //preparacao das variaveis, copia de objetos:

pAlias é uma referência para o mesmo local de memória que pOriginal, por este motivo quando pAlias é alterado, pOriginal se altera por “efeito colateral”, eles compartilham o mesmo objeto pois a atribuição pAlias = pOriginal, copia o endereço de pOriginal.

Já pCopia, é o resultado de uma nova alocação de memória, portanto um novo endereço, um objeto independente dos outros.

### //comparacao de objetos:

pOriginal == pAlias e outras comparações equivalentes têm o significado de comparação do endereço de memória e não do conteúdo.

pOriginal.x == pCopia.x tem o significado de comparação do valor desses atributos, assim como uma comparação entre inteiros. Se esses atributos por sua vez fossem objetos (tipo Integer ao invés de int), esta operação teria o significado de comparação entre endereços de memória dos objetos.



## CÓDIGO

As possíveis soluções, comentadas incluindo verificação da classe dos objetos

```
//Classe Ponto
```

```
class Ponto {

    public float x,y;

    public Ponto(float ax,float ay) { //omita o valor de retorno!
        //garante o estado do objeto
        this.x = ax; this.y = ay;
    }

    public void move(float dx,float dy) {
        this.x += dx; this.y += dy;
    }

    public String toString() {
        return "(" + this.x + "," + this.y + ")"; //(x,y)
    }

    public boolean equals(Ponto outro) { //equals significa igual
        return ((outro.x == this.x) && (outro.y == this.y)); //this==outro?
    }

    public void copy(Ponto outro) {
        this.x = outro.x;
        this.y = outro.y;
    }

    public Ponto clone() {
        Ponto cloned = new Ponto(this.x, this.y);
        return cloned;
    }
}
```

```
//Classe principal, Arquivo Principal.java
```

```
class Principal {

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

```

//preparacao das variaveis
Ponto pOriginal, pCopia1, pCopia2;
pOriginal = new Ponto((float)0.0,0.0f); //(float)0.0 ou 0.0f; 0.0 eh double
pCopia1 = new Ponto(0.0f,0.0f);
pCopia1.copy(pOriginal); //pCopia1 copia conteudo de pOriginal nele mesmo
pCopia2 = (Ponto) pOriginal.clone();

//copia de objetos
System.out.println("Original:" + pOriginal);
System.out.println("Copia1:" + pCopia1);
System.out.println("Copia2:"+pCopia2);
System.out.println("Modificando Copia1.x para 2");
pCopia1.x=2.0f;
System.out.println("Veja como o original ficou:"+pOriginal);
System.out.println("Copia2 nao se modifica:"+pCopia2);

//comparacao de objetos
System.out.println("Original==Copia2:"+(pOriginal==pCopia2) );
System.out.println("Original.igual(Copia2):" + pOriginal.equals(pCopia2) );
System.out.println("Deixando atributos de Copia iguais aos de Original");

//verificacao da classe dos objetos
System.out.println("Obtendo a classe dos objetos");
System.out.println(pOriginal.getClass().getName());
System.out.print("Original e da classe Ponto?");
boolean result = (pOriginal instanceof Ponto);
System.out.println(result);
}
}

```



Original:(0.0,0.0)  
 Copia1:(0.0,0.0)  
 Copia2:(0.0,0.0)  
 Modificando Copia1.x para 2  
 Veja como o original ficou:(0.0,0.0)  
 Copia2 nao se modifica:(0.0,0.0)  
 Original==Copia2:false  
 Original.igual(Copia2):true  
 Deixando atributos de Copia iguais aos de Original  
 Obtendo a classe dos objetos  
 Ponto  
 Original e da classe Ponto?true

Com o seu conhecimento dos tópicos anteriores é possível entender perfeitamente o que faz esse programa. E após entender você terá praticado três conceitos importantes em orientação a objetos: 1. teste de igualdade entre objetos (equals method); 2. cópia de objetos (copy method); 3. clone de

objetos (clone method).

```
boolean result = (pOriginal instanceof Ponto);
```

Este é seu primeiro contato com o operador **instanceof** que leva do lado esquerdo uma instância de objeto e do lado direito o nome de uma classe e retorna **true** se o objeto pertence à essa classe e **false** caso contrário.

## OBTENDO VALORES DO USUÁRIO

Este tópico foi introduzido porque os programas seguintes são mais complexos e precisam ser testados iterativamente. Mesmo que você só vá fazer programas em interfaces gráficas é interessante saber como testar certas classes via teclado ou linha de comando, sem contar que sem o conteúdo deste tópico os programas seguintes não serão possíveis. Além do que você pode desejar fazer algum programa de linha de comando (batch) em **Java**, por exemplo um programa de comunicação entre as diversas máquinas de uma rede ou um programa que receba argumentos via linha de comando.

## LENDO DO TECLADO

Este tópico contém dois exemplos que ensinam como ler **Strings** do teclado, como convertê-las para números, e como controlar melhor a entrada e saída de dados.

## LEITURA DE **STRINGS** USANDO UM VETOR DE BYTES.

Em **Java** é praticamente um padrão ler dados em bytes, seja do teclado, da rede, do disco ou de qualquer outro lugar. Por este motivo o primeiro exemplo lê em um vetor de bytes. Como sabemos que você quer mais do que isso( ex. ler outros tipos da linguagem ), o segundo exemplo tratará desse caso.

### **CÓDIGO**

```
1 import java.io.*;
2
3 class EntraDados {
4
5     public static void main (String args[])
6     {
7
8         byte vetortexto[] = new byte[200]; //declaracao de um vetor de bytes
9         int byteslidos = 0;
10
11         System.out.println("Escreva algo:");
12
13         try {
14             byteslidos = System.in.read(vetortexto);
15             System.out.print("Voce escreveu:");
```

```

16      System.out.write(vetortexto,0,byteslidos);
17      }
18      catch (IOException e) {
19          // Alguma acao de recuperacao da falha
20      }
21  }
22  }

```



Escreva algo:  
 Como assim escreva algo?  
 Voce escreveu: Como assim escreva algo?

## //COMENTARIOS:

Este programa usa o método `System.in.read(vetortexto);` para ler do teclado. Este método precisa de um vetor de bytes como argumento (onde serão lidos os caracteres) e além disso retorna o número de bytes lidos, para que você possa usar o vetor corretamente.

Para descarregar o vetor no vídeo use o método `System.out.write(vetortexto,0,byteslidos);` que imprime na tela as posições de 0 até `byteslidos` do vetor de bytes passado como o primeiro argumento.

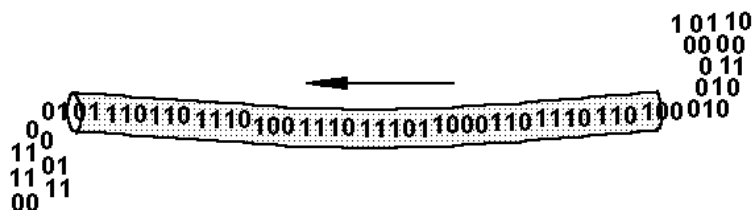
Você deve estar se perguntando qual a função dos blocos de código `try {} catch {}` deste programa exemplo. Eles são importantes no tratamento de exceções, tópico que será abordado no final deste texto. Por enquanto apenas veja estes blocos de código como necessários para escrever dentro do bloco `try {}` as operações de leitura de teclado, que são operações que podem gerar exceções.

## UMA VISÃO GERAL SOBRE PACKAGES E STREAMS

A maneira ensinada nos tópicos anteriores (leitura de bytes) é suficiente para que você leia do teclado, mas aproveitaremos agora que você já está mais experiente para ensinar uma outra maneira elegante que serve para ler do teclado, de um arquivo ou de uma conexão da rede que aceita por exemplo a leitura direta para uma `String` e outros tipos desta linguagem e não só para um vetor de bytes.

Nestes exemplos usaremos um conceito que somente serão explicados a fundo nos próximos tópicos, são os conceitos de packages e Streams. Packages são conjuntos de classes. Até agora só tínhamos utilizados elementos da package `java.lang` que é importada implicitamente em todos os programas, permitindo por exemplo escrever na tela (`System.out` é uma classe presente em `java.lang`).

A package `java.io` fornece a abstração de streams para lidar com a complexidade de entrada e saída de dados. Esta abstração é poderosa e será utilizada aqui para ler do teclado. Um stream é como um cano por onde passam os bytes, a vantagem deste cano é que não precisamos nos preocupar de onde vem esses bytes, para nós eles vem do cano. Para quem escreve no stream também vale a mesma idéia do que agora do outro lado, empurrando os bytes para dentro.



Neste exemplo usamos um stream para ler do teclado, esta poderosa abstração será estendida mais tarde para tratar a entrada e saída de dados através de portas de comunicação e programação cliente servidor, explorando as capacidades de “networking” da linguagem, além da leitura e escrita em arquivos, no tópico adequado.



## CÓDIGO

Este programa simplesmente lê uma linha do teclado e a imprime novamente.

```

1  import java.io.DataInputStream; //classe DataInputStream para a entrada de dados
2
3  public class ReadString {
4
5      public static void main(String args[]) {
6
7
8          String linha="";
9
10         DataInputStream meuDataInputStream;
11
12         meuDataInputStream = new DataInputStream(System.in);
13
14         try{
15             linha = meuDataInputStream.readLine();
16         }
17         catch (Exception erro) { System.out.println("Erro de leitura"); }
18         //antes de imprimir ou armazenar a string, e' obvio que voce poderia executar algum
19         //processamento, mas nao estudamos a classe string ainda, por isso tenha paciencia.
20         System.out.println(linha);
21
22     }/
23
24
25 }
```



Voce pode andar rapido com paciencia  
Voce pode andar rapido com paciencia

## //COMENTARIOS

O cano de nosso exemplo é mais parecido com uma mangueira de jardim, nós ficamos com a ponta de onde sai a água.



1:Nesta linha dizemos que desejamos usar, importar, um stream de entrada de dados de nome `DataInputStream` presente na package `java.io`

8: Nesta linha declaramos o que seria o balde para recolher a água da mangueira (Uma `String`). O balde começa vazio ("").

10:Esta linha diz para pegar a mangueira (`DataInputStream`) do depósito (package `java.io`).

12:Esta linha conecta a outra ponta da mangueira na torneira (`System.in` é a torneira, que significa teclado em quase todos os computadores).

15:Encha o balde com o que sair da torneira até haver uma interrupção no fornecimento. (Leia o que puder do stream até encontrar um caractere de nova linha '\n'. Daí o nome `readline` que significa leia linha).

20:Esvazie o balde no cano de saída (Copie o valor da `String` lida na entrada do stream `System.out`, que significa vídeo na maioria das máquinas).

As linhas 14,15 e 17 são necessárias para escrever `line = myDataInputStream.readLine();`. Essas linhas permitem ações de recuperação de falhas que possam ocorrer no processo de leitura, abordaremos este tópico bem mais adiante, de modo que aqui apenas daremos uma visão geral só a título de curiosidade:

No bloco `try { }`, deve-se inserir as operações passíveis de falhas. Este bloco deve ser seguido por `catch (Exception erro) { /* nao faco nada */ }`. No lugar de `/* nao faco nada */` deveriam estar as possíveis ações de recuperação de falhas como no caso adotamos: `System.out.println("Erro de leitura");`.

`line = myDataInputStream.readLine();` é o método que faz com que o `Stream` leia da entrada de dados em que foi associado uma linha. Este método obviamente bloqueia a execução do programa até que se digite "carriage return", o que não é problemático pois isto poderia ser feito paralelamente, usando "threads".

### Como converter **Strings** para inteiros:

Se você inserir o seguinte método na nossa classe `ReadString`, poderá ler valores inteiros do teclado.

```
static int leInteiro() {  
  
    String line;  
  
    DataInputStream in = new DataInputStream(System.in);  
    try {  
        line = in.readLine();  
        int i = Integer.valueOf(line).intValue();  
        return i;  
    }  
    catch (Exception e) {  
        return -1;  
    }  
}
```

```
int i = Integer.valueOf(line).intValue();
```

É um exemplo de chamadas consecutivas ou aninhadas de métodos. Olhe da esquerda para a

direita. Primeiro o método `valueOf(line)` é chamado para a classe<sup>8</sup> `Integer` retornando um objeto da classe `Integer`. Depois o objeto da classe `Integer` tem seu método `intValue()` invocado retornando um valor do tipo `int` que é correspondente a `String line`.

Você se lembra que nós falamos que a linguagem fornece uma auxiliar (“wrapper”) para cada tipo básico da linguagem? A classe `Integer` é uma classe que mapeia em termos de objetos o tipo básico `int` de 32 bits. Agora que você já conhece o nome da classe, pesquise mais sobre ela na `JAVA API`.

## Exercícios:

1-

Voltaremos a falar sobre `Strings`, porém por hora, você já tem condições de ler as bibliografias adicionais sobre eles.

## ARGUMENTOS DE LINHA DE COMANDO

Na segunda parte veremos que os programas `Java` que rodam embebidos em hipertextos podem receber parâmetros iniciais de funcionamento do hipertexto. Os programas que estamos descrevendo nesta parte são aplicações executadas via linha de comando. De modo análogo, essas aplicações podem receber argumentos de linha de comando.

Exemplo de passagem de três argumentos de linha de comando para um programa `Java`:

`java teste “Texto” 2 Nome`

Interrompa sua leitura agora e experimente passar argumentos para um dos programas criados anteriormente, os argumentos obviamente serão ignorados pelo programa e este se comportará da mesma forma que antes.

O vetor de `Strings` é o argumento do método `main` que normalmente chamávamos de `String args[]`:

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

## CÓDIGO

```
class MostraArgumentos {  
    public static void main(String args[]) {  
        for (int i=0; i < args.length; i++) {  
            System.out.println("Argumento" + i+": "+ args[i]);  
        }  
    }  
}
```

Resultado do programa para a chamada: `java MostraArgumentos Passando 3 “argumento s”`



```
Argumento0: Passando  
Argumento1: 3  
Argumento2: argumento s
```

`args.length`; Retorna o comprimento do vetor de `Strings`, este valor é usado para iterar sobre

<sup>8</sup> É possível criar métodos para serem chamados para classes e não só para objetos, são os chamados “class methods” em contraposição a “instance methods”.

os argumentos que são **Strings** armazenadas em `args[i]`. Perceba que as aspas fazem com que nomes separados por espaços sejam considerados só um argumento.

Os argumentos passados para seu programa são gravados em um vetor de **Strings**, para usar o argumento 3 que ficou guardado como **String**, na forma de inteiro, é preciso primeiro convertê-lo para inteiro. Isto pode ser feito no caso do nosso exemplo através de uma chamada a:

```
Integer.parseInt(args[1]);
```

A classe **Integer** usada no código acima é um dos “wrappers” que descrevemos em . A operação contrária a que foi feita é `Integer.toString(int a)`; O método `parseInt` aceita também um segundo argumento que é a base em que o número está representado na **String**.

### Exercícios:

1-

Mude o primeiro programa em **Java** (**HelloInternet**) para imprimir “Hello” seguido do primeiro argumento de linha de comando (se existir: `args.length>0` ).

2-

Construa um programa simples que recebe argumentos da linha de comando e os imprime através de `cout`. Normalmente é isso que deve ser feito antes de usar um recurso da linguagem pela primeira vez, experimentá-lo em programas simples.

## ENCAPSULAMENTO COM **PRIVATE**, **PUBLIC**, “**PACKAGE**” e **PROTECTED**

Encapsulamento, “data hiding” é um conceito bastante importante em orientação a objetos. Neste tópico vamos falar das maneiras de restringir o acesso as declarações de uma classe e a própria classe, isto é feito através do uso das palavras reservadas **public**, **private** e **protected**<sup>9</sup> <sup>10</sup> que são qualificadores.

Alguém pode estar se perguntando o porquê de se restringir o acesso a certas partes de uma classe. A idéia é simples, devemos fornecer ao usuário, cliente de uma classe, o necessário e somente o necessário para que ele tire proveito da funcionalidade desta classe. Os detalhes devem ser omitidos, somente a lista de operações a qual uma classe deve atender fica visível.

Os benefícios são muitos: clareza do código, minimização de erros, facilidade de extensão. Talvez a facilidade de modificação seja o mais importante dos benefícios. Como a classe é conhecida pela sua interface, é muito fácil mudar a representação interna sem que o cliente, usuário, perceba a diferença. Estaremos preocupados em separar design de implementação, **Java** é uma linguagem boa de se programar em termos de design e em termos de implementação.

Programar tendo em vista o design é também chamado de “programming in the large”, enquanto que programar tendo em vista implementação, codificação é chamado de “programming in the small”. Alguns programadores experientes afirmam que **Java** se parece com **C** quando estamos preocupados com codificação, mas quando estamos preocupados com design, **Java** se assemelha a **Smalltalk**.

Com encapsulamento você será capaz de criar componentes de software reutilizáveis, seguros, fáceis de modificar.

---

<sup>9</sup> **Protected** será explicada juntamente com herança na seção 2.

<sup>10</sup> **PACKAGE** será explicado juntamente com packages.

## ENCAPSULANDO MÉTODOS E ATRIBUTOS

Até agora não nos preocupávamos com o modo de acesso de declarações de uma classe pois, mesmo sem saber porque, você foi avisado para qualificar todos os atributos e métodos de suas classes como **public** o que significa que eles são acessíveis, visíveis, em qualquer local de seu código. Por visível entenda o seguinte: se o atributo **x** do objeto **UmPonto** não é visível por exemplo fora de sua classe, então não faz sentido escrever em **main**: **UmPonto.x=0;** .

Mas então como controlar o acesso de atributos e métodos em uma classe? Simples, através das palavras reservadas **private**, **public** e **protected** cujos significados quando qualificando métodos e atributos (**private** e **public** podem também qualificar classes) são descritos abaixo:

<b>public</b>	Estes atributos e métodos são sempre acessíveis em todos os métodos de todas as classes. Este é o nível menos rígido de encapsulamento, que equivale a não encapsular.
<b>private</b>	Estes atributos e métodos são acessíveis somente nos métodos(todos) da própria classe. Este é o nível mais rígido de encapsulamento.
<b>protected</b>	Estes atributos e métodos são acessíveis nos métodos da própria classe e suas subclasses, o que será visto em Herança .
Nada especificado, equivale “package” ou “friendly”	Estes atributos e métodos são acessíveis somente nos métodos das classes que pertencem ao “package” em que foram criados. Este modo de acesso é também chamado de “friendly”.

(existem outros qualificadores, não relacionados com encapsulamento que serão explicados depois)



### ABC

*Package e friendly: Aparecem entre aspas porque não são palavras reservadas da linguagem, são apenas nomes dados para o tipo de encapsulamento padrão (default), que ocorre quando não existe um especificador. São nomes fáceis de memorizar. Friendly significa amigável, no sentido de que as classes que permitem este tipo de acesso possuem um encapsulamento mais relaxado com relação as classes do mesmo package (amigas). Package é um grupo de classes relacionadas.*

**Protected** será explicada em pois está relacionada com herança, por hora vamos focalizar nossa atenção em **private** e **public** que qualificam os atributos e métodos de uma classe quanto ao tipo de acesso (onde eles são visíveis) . **Public**, **private** e **protected** podem ser vistos como qualificadores ou “specifiers”.

Para facilitar a explicação suponha a seguinte declaração de uma classe:

```
1)
class Ponto {
  private float x
  private float y
  public void inicializa(float a,float b) {x=a; y=b;};
  public void move (float dx,float dy);
}
```

Área private.

Fica fácil entender essas declarações se você pensar no seguinte: esses qualificadores se aplicam aos métodos e atributos que vem imediatamente após eles. Os elementos da classe qualificados como

**private** aparecem com fundo cinza escuro indicando que sua “visibilidade” é mais limitada que os atributos qualificados como **public** (cinza claro).

Agora vamos entender o que é **private** e o que é **public**. Vamos supor que você instanciou (criou) um objeto do tipo **Ponto** em seu programa:

```
Ponto meu; //instanciacao  
meu=new Ponto();
```

Segundo o uso da definição da classe **Ponto** dada acima você **não** pode escrever no seu programa:

```
meu.x=(float)5.0; //erro !
```

,como fazíamos nos exemplos anteriores, a não ser que **x** fosse declarado como **public** na definição da classe o que não ocorre aqui. Mas você pode escrever **x=5.0;** na implementação (dentro) de um método porque enquanto não for feito uso de herança, pode-se dizer que um método tem acesso a tudo que é de sua classe, veja o programa seguinte.

Você pode escrever: **meu.move(5.0,5.0);** ,porque sua declaração (**move**) está como **public** na classe, em qualquer lugar se pode escrever **meu.move(5.0,5.0);**.

Visibilidade das declarações de uma classe, fora dela ,de sua hierarquia e de seu package. Veja que só a parte **public** é visível neste caso:

PUBLIC
PRIVATE
PROTECTED
“PACKAGE”

Visibilidade das declarações de uma classe, dentro dela mesma:

PUBLIC
PRIVATE
PROTECTED
“PACKAGE”

## ATRIBUTOS **PRIVATE**, MÉTODOS **PUBLIC**

Aplicando encapsulamento a classe **ponto** definida anteriormente, deixando os atributos encapsulados e definindo a interface publica da classe somente através de métodos.

```
//Classe ponto  
  
public class Ponto {  
  
    private float x,y;           //atributos private  
  
    public Ponto(float ax,float ay) //omita o valor de retorno!  
    //garante o estado do objeto  
    {  
        this.x=ax; this.y=ay;  
    }  
  
    public void move(float dx,float dy)  
    {  
        this.x+=dx; this.y+=dy;  
    }  
}
```

```

}

public float retorna_x()
{
return x;
}

public void mostra()
{
System.out.println( "(" + this.x + "," + this.y + ")" );
}
}

```

//Classe principal, Arquivo Principal.java

```

class Principal {

public static void main(String args[]) {
    Ponto ap;
    ap=new Ponto((float)0.0,(float)0.0);
    ap.mostra();
}

}

```



(0,0)

### //COMENTARIOS:

Este programa não deixa você tirar o ponto de (0,0) a não ser que seja chamada inicializa novamente. Fica claro que agora, encapsulando **x** e **y** precisamos de mais métodos para que a classe não tenha sua funcionalidade limitada. Novamente: escrever **ap.x=10;** em **main** é um erro! Pois **x** está qualificada como **private**. Sempre leia os exercícios, mesmo que não vá fazê-los. O que ocorre em classes mais complicadas, é que ao definir mais métodos na interface da classe, estamos permitindo que a representação interna possa ser mudada sem que os usuários tomem conhecimento disto.

### Exercícios:

1-

Implemente os métodos **public void altera\_x(float a)** , **public float retorna\_x(void)**, **public void move (float dx,float dy );** .Implemente outros métodos que achar importantes exemplo **public void distancia(ponto a) { return dist(X,Y,a.X,a.Y); }** , onde **dist** representa o conjunto de operações matemáticas necessárias para obter a distância entre (X,Y) (a.X,a.Y). Você provavelmente usará a função **Math.sqrt()** que define a raiz quadrada de um **double**, não é preciso fazer nenhum **import**

para usar `Math.sqrt()`, mas é preciso converter os argumentos de `float` para `double` e o valor de retorno de `double` para `float`.

Veja que no método `distancia`, podemos obter acesso aos atributos `private X` e `Y` do argumento `a`, isto é permitido porque `distancia` é um método da mesma classe de `a`, embora não do mesmo objeto, uma maneira de prover este tipo de acesso para outras classes (`distancia` de reta a ponto) é dotar a classe `Ponto` de métodos do tipo `float retorna_x(void)`;

2-

Escolha um programa implementado anteriormente e aplique encapsulamento, analise a funcionalidade de sua classe. Você teve que implementar mais métodos? Quais?

## UM ATRIBUTO É PUBLIC

Neste tópico pedimos que o leitor faça uma variante do programa anterior, a única diferença é que `Y` deve ser colocado na parte `public` da definição da classe podendo ser acessado diretamente. Para escrever a variante, tome o código do programa anterior e substitua

```
private float x,y;  
por  
private float x;  
public float y; //y pode ser acessado
```

### Comentários:

Observe que agora nada impede que você acesse diretamente `y`: `ap.y=100.0`, porém `ap.x=10.00` é um erro. Observe em que parte (área) da classe cada um desses atributos foi declarado.

### Exercícios:

1-

Crie os métodos `float retorna_x(void)`, `void altera_x(float a)`; que devem servir para retornar o valor armazenado em `x` e para alterar o valor armazenado em `x` respectivamente. Crie também os respectivos métodos `retorna` e `altera` para o atributo `y`.

2-

Qual das seguintes declarações permite que se acesse em `main` **somente** os métodos `move` e `inicializa`, encapsulando todos os outros elementos da classe? Obs.: A ordem das declarações `private` e `public` pode estar invertida com relação aos exemplos anteriores.

a)

```
public class Ponto {  
    public float x;  
    public float y;  
    public void inicializa(float a, float b) {x=a; y=b;};  
    public void move(float dx, float dy) ; {x+=dx; y+=dy; };  
};
```

b)

```
public class Ponto {
```

```

public void inicializa(float a, float b) {x=a; y=b;};
public void move(float dx, float dy) ; {x+=dx; y+=dy; };
private float x;
private float y;
};

```

c)

```

public class Ponto {
    public void inicializa(float a, float b) {x=a; y=b;};
    private void move(float dx, float dy) ; {x+=dx; y+=dy; };
    public float x;
    private float y;
};

```

## ENCAPSULAMENTO E “PACKAGES”

Neste tópico explicaremos o recurso de packages com ênfase nas opções de encapsulamento relacionadas, em herança explicaremos este recurso com ênfase no reuso de código das packages oferecidas com a linguagem. Packages são conjuntos de classes relacionadas, estes conjuntos são determinados incluindo uma linha no topo de cada arquivo indicando a qual package pertencem as classes ali declaradas. Se nenhuma linha é inserida assume-se que todas as classes pertencem a uma package só.

## ENCAPSULAMENTO DE ATRIBUTOS E MÉTODOS COM PACKAGES

O encapsulamento de atributos e métodos atingido com o uso de packages é muito semelhante ao encapsulamento com **private** e **public**, só que agora o “limite de visibilidade” é mais amplo do que a classe. A questão aqui levantada é se a classe é visível fora do “package” ou não. Só para lembrar, a questão que do item anterior: , era se os métodos e atributos eram visíveis fora da classe ou não.

Visibilidade das declarações de uma classe, dentro de seu package:

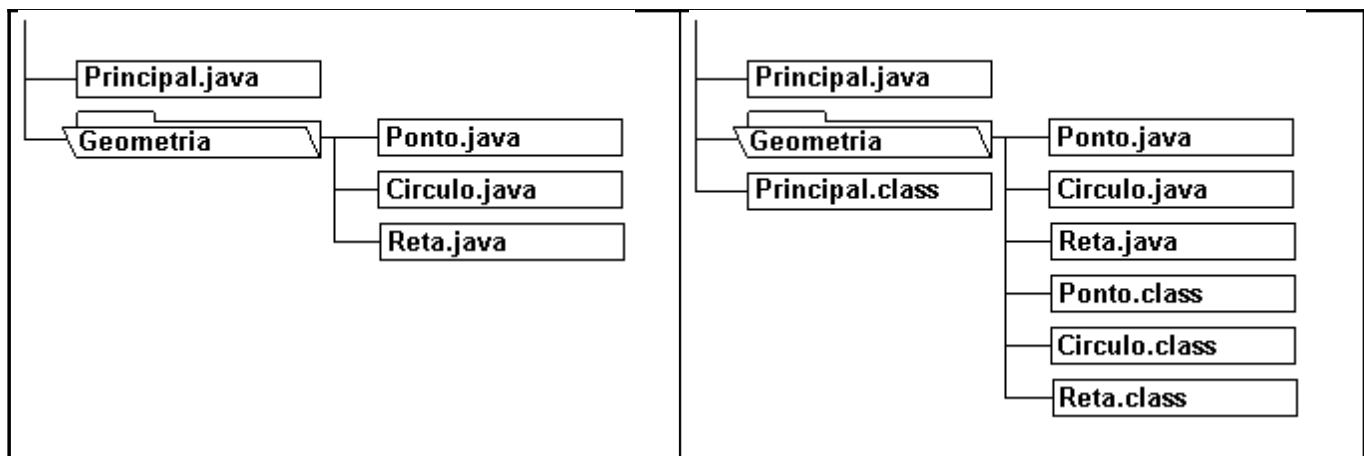
<b>PUBLIC</b>
<b>PRIVATE</b>
<b>PROTECTED</b>
<b>“PACKAGE”</b>

Quando explicarmos **protected** forneceremos um diagrama completo dos modos de encapsulamento de métodos e atributos, este diagrama é muito útil para a memorização.

“Packages” é um recurso da linguagem que permite formar grupos de classes relacionadas entre si de forma que elas ofereçam facilidades umas as outras. Facilidades estas que nem sempre são oferecidas ao usuário. Vamos montar uma package de nome **Geometria** contendo classes que representam elementos gráficos tais como retas, círculos e pontos. A estrutura de diretório abaixo descreve a disposição dos arquivos deste exemplo:

<b>Início:</b>	<b>Após Javac:</b>
----------------	--------------------





Note que a package `Geometria` está toda sob um diretório ou folder de mesmo nome, isto ocorre porque a estrutura de packages deve ser mapeada em uma estrutura de diretórios para que suas classes possam ser achadas. Assim a classe `java.awt.Color` está dois níveis, portanto dois diretórios abaixo na hierarquia de packages fornecida com a linguagem.

## CÓDIGO

```
//Classe ponto
package Geometria;

public class Ponto {

float x,y; //nenhu especificador, default=package

public Ponto(float ax,float ay) //omita o valor de retorno!
//garante o estado do objeto
{
    this.x=ax; this.y=ay;
}

public float retornaX()
{
    return x;
}

public void move(float dx,float dy)
{
    this.x+=dx; this.y+=dy;
}

public void mostra()
{
    System.out.println("("+this.x+","+this.y+");");
}

}
```

```

//Classe circulo
package Geometria;

public class Circulo {

float raio,x,y; //nenhum especificador, default=package

public Circulo(float ax,float ay,float ar)
//garante o estado do objeto
{
    this.x=ax; this.y=ay; this.raio=ar;
}

public void altera_raio(float a)
{
    this.raio=a;
}

public float retorna_raio()
{
    return this.raio;
}

public void move(float dx,float dy)
{
    this.x+=dx; this.y+=dy;
}

public float distancia(Ponto ap)
{
    float dcp; //distancia do centro do circulo ao ponto
    dcp=(float)Math.sqrt((double) ((x-ap.x)*(x-ap.x)+(y-ap.y)*(y-ap.y)) ); //acesso direto
    //aos atributos de ap, isto porque as classes pertencem ao mesmo package
    if (dcp<raio) {return raio-dcp; }
    else {return dcp-raio; }
}

public void mostra()
{
    System.out.println("(" +this.x+", "+this.y+", "+this.raio+"");
}

}

```

```

//Arquivo Reta.java

```

```
//Classe Reta
package Geometria;

public class Reta {

    Ponto a,b; //idem , sao encapsulados como package

    public Reta(float ax,float ay,float bx,float by)

    {
        a=new Ponto(ax,ay);
        b=new Ponto(bx,by);
    }

    public float distancia(Ponto ap)
    {
        //metodo nao implementado
        //acesse livremente os atributos do argumento Ponto
        //de modo a calcular sua distancia a esta reta, ex copia=ap.x;
        return 0.0f;
    }

    public void mostra()
    {
        a.mostra();
        b.mostra();
    }

}
```

```
//Classe principal, Arquivo Principal.java
import Geometria.*;

public class Principal {

    public static void main(String args[]) {
        Circulo acirc;
        //acirc.x=(float)10.0; erro! atributo encapsulado (modo package)
        Ponto apto;
        acirc=new Circulo((float)0.0,(float)0.0,(float)1.0);
        acirc.mostra();
        apto=new Ponto((float)4.0,(float)3.0);
        apto.mostra();
        System.out.println("Dist:" + acirc.distancia(apt));
    }
}
```

```
}
```



(0,0,1)  
(4,3)  
Dist:4

As declarações em negrito dos arquivos acima: **package Geometria.\*;** e **import Geometria.\*** devem ser inseridas logo no início do arquivo, só podendo ser precedidas por comentários.

Existe uma quantidade grande de “packages”, já definidas na linguagem, as quais abordaremos na segunda parte. Por hora basta saber que já estamos usando declarações da “package” `java.lang`, mesmo sem especificar `import java.lang.*;` no início de nossos arquivos. Isto ocorre porque este package é importado implicitamente em todos os programa Java.

O `.*` usado em `import Geometria.*;` tem o significado de todas as classes da package `Geometria`, o asterisco é usado como um coringa, um valor que pode substituir todos os demais, assim como em alguns sistemas operacionais.

Embora de fácil utilização, este conceito constitui uma inovação com relação a C++. Em Modula-3 pode-se obter um efeito semelhante utilizando declarações de classes (tipos objetos) no mesmo módulo e interfaces que não exportam tudo o que existe no módulo, assim as classes declaradas no mesmo módulo oferecem facilidades umas as outras.

O modo de acesso de atributos e métodos chamado “friendly” ou package é proporcionado pelo uso de “packages” e permite programar de maneira eficiente (sem muitas chamadas de métodos) e segura (com encapsulamento).

Se os atributos da classe `Ponto` forem especificados como “friendly” ou package (equivale a nenhum especificador), eles poderão ser acessados **diretamente** (sem chamadas do tipo `ap.retorna_x();`) pela classe `Reta` que pertence a mesma “package”, mas continuam encapsulados para as classes externas ao “package” ou seja para o usuário. Enfatizamos o termo “diretamente” porque obter acesso a um atributo via método é bem mais custoso do que acessá-lo diretamente, mas se você programar usando packages, ganhará o benefício da eficiência sem perder o do encapsulamento.

## ENCAPSULAMENTO DE CLASSES COM PACKAGES

Até agora, tínhamos declarado todas as classes como **public** (**public class Nomeclasse {}**), embora sem explicar exatamente porque. Este tópico trata da possibilidade de o programador desejar criar uma classe para seu uso próprio e não fornecê-la para o usuário.

O que a linguagem Java permite neste sentido é:

1- A criação de classes que só podem ser usadas dentro de packages, ou seja, você não pode declarar um objeto desta classe externamente ao package.

2- A criação de classes que podem ser usadas somente pelas classes presentes no mesmo arquivo, embora não tenhamos mostrado, um arquivo pode conter a declaração de mais de uma classe desde que somente uma delas seja **public**.

O encapsulamento de classes segue uma sintaxe semelhante ao encapsulamento de métodos e atributos, ou seja: através de qualificadores ou modificadores:

```
QualificadorDaClasse class NomeDaClasse { /*Atributos e metodos */ }
```

Os qualificadores são:

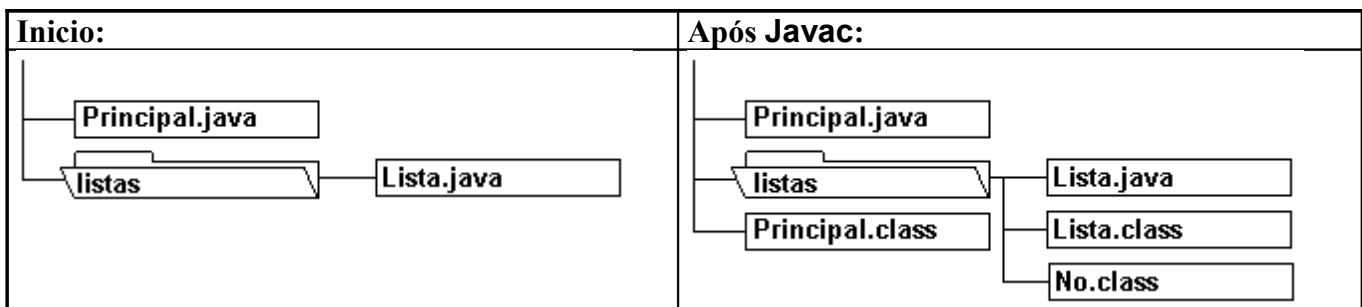
public	Estas classes são sempre acessíveis em todos os packages do seu código. Somente uma classe publica é permitida por arquivo, e o arquivo deve ter o mesmo nome da classe.
private	Estas classes são acessíveis somente pelas classes declaradas no mesmo arquivo. Um arquivo pode possuir várias classes <b>private</b> , mas uma única classe <b>public</b> .
Nada especificado "package"	Estas classes podem ser acessadas no package que elas pertencem, se nenhum package é especificado, elas pertencem ao programa.

(existem outros qualificadores, **não** relacionados com encapsulamento que serão explicados depois)

Seus programas não precisam necessariamente fazer uso destes recursos de encapsulamento, mas todo código escrito para terceiros deve utilizá-los intensamente.

Você teve ter notado que sempre definimos uma classe por arquivo, isto é feito porque a linguagem só permite uma classe **public** por arquivo, as outras tem que ser **private** ou **package**.

Um exemplo de utilização: ao criar uma classe lista ligada alguém pode achar conveniente definir uma classe nó para ser usada somente pela classe **listaligada**. Uma opção é definir a classe nó com modo de encapsulamento: **package**. O cliente de sua classe lista ligada não precisa saber que esta classe se baseia em uma classe **No**. Vamos implementar esta idéia no exemplo a seguir, tomarei o cuidado de fazê-lo bem simples pois estamos no começo do tutorial:



## CÓDIGO

```
package listas;
//classe Lista, classe No arquivo Lista.java

class No { //sem especificador de modo de acesso na classe

private char info; //se eu ja tivesse ensinado protected usaria em lugar de private
private No prox; //"ponteiro" para o proximo no

No(char i,No p) //construtor
{
    info=i;
    prox=p;
}

char retorna_info()
```

```

//retorna valor do campo
{
    return info;
}

void altera_info(char i)
//altera valor do campo
{
    info=i;
}

void altera_prox(No p)
//altera valor do proximo no
{
    prox=p;
}

No retorna_prox()
//retorna referencia ao proximo no
{
    return prox;
}

}

public class Lista {

private No cabeca;    //inicio da lista
private int elementos; //numero de nos na lista


public Lista()
//construtor
{
    cabeca=null;
    elementos=0;
}

public void insere(char a)
{ //realizada em muitos mais pacos para facilitar apredizado
    elementos++;
    No temp;
    if (cabeca==null) cabeca=new No(a,null);
    else{
        temp=new No(a,null);
        temp.altera_prox(cabeca);
        cabeca=temp;
    }
}
}

```

```

    }

    //se cabeca == null tambem funciona
}

public char remove()
{
    No removido;
    if (cabeca==null) return '0'; //elementos==0
    else
    {
        elementos--;
        removido=cabeca;
        cabeca=cabeca.retorna_prox();
        return removido.retorna_info();
    }
}

public int retorna_elementos()
{
    return elementos;
}

public void mostra() //nao deveria estar aqui, e so para debugar
{
    No temp=cabeca;
    while (temp!=null)
    {
        System.out.print( "[" + temp.retorna_info() + "]"-"    ");
        temp=temp.retorna_prox();
    }
    System.out.print("null");
    System.out.println();
}
}

```

Agora você pode escolher entre duas versões do programa principal, a versão da esquerda implementa um loop de entradas do teclado que permite testar iterativamente a classe **Lista**, não explicaremos ainda detalhes deste loop, apenas como usá-lo. Dê uma olhada no código desta versão de programa principal, se você acha-lo complicado pode usar a segunda versão presente a direita deste arquivo e em uma segunda leitura retornar a versão da esquerda.

Na versão com o loop (1-esquerda) a letra **'i'** indica o comando **inserção**, se você digitar **i<enter>** inserirá o caractere **<enter>** na sua lista, o que normalmente não é desejado, digite **ic<enter>** para inserir o caractere **c**. **'r'** indica **remoção**, e **'m'** indica que a lista deve ser **mostrada** na tela. Pelo resultado do programa você entenderá melhor esses comandos rudimentares de teste. Você pode achar rudimentar programar assim, mas é um bom método, verá como esta mesma lista depois de testada assim pode ficar bonita se inserida em um **applet** e mostrada graficamente.

Escolha uma implementação para entender e compilar:

```
//Classe principal, Arquivo Principal.java versao 1
import java.io.DataInputStream;
import listas.*;

class Principal {

    public static void main(String args[]) {
        Lista ml=new Lista(); //ml=minhalista
        char o,e; //o=opcao, e=temporario

        DataInputStream meuDataInputStream
        =new DataInputStream(System.in);

        try{
            do
            {
                o =(char)meuDataInputStream.read();
                switch (o) {
                    case 'i':
                        e=(char)meuDataInputStream.read();
                        ml.insere(e);
                        break;
                    case 'r':
                        e=ml.remove();
                        System.out.println(e);
                        //System.out.flush();
                        break;
                    case 'm':
                        ml.mostra();
                        System.out.println();
                        //System.out.flush();
                        break;
                    default: ;
                }

            } while (o!='q');}
        catch (Exception erro) { /* nao faco nada */ }
    }
}
```

```
//Classe principal, Arquivo Principal.java versao 2
import java.io.DataInputStream;
import listas.*;

class Principal {

    public static void main(String args[]) {
        char e;
        Lista ml=new Lista(); //ml=minhalista
        ml.insere('a');
        //mesma sequencia da versao com menu
        //de programa principal
        ml.mostra();

        ml.insere('v');
        ml.insere('a');
        ml.insere('j');
        ml.mostra();

        e=ml.remove();
        System.out.println(e);
        ml.mostra();

        ml.remove(); //embora o metodo
        //remove retorne um valor
        ml.remove(); //nestas chamadas,
        //este valor de retorno e'
        ml.remove(); //ignorado
        ml.mostra();

    }
}
```



ia  
m  
**[a]-null**

iviai  
m  
**[j]-[a]-[v]-[a]-null**

r



[a]-null  
[j]-[a]-[v]-[a]-null  
j  
[a]-[v]-[a]-null  
null



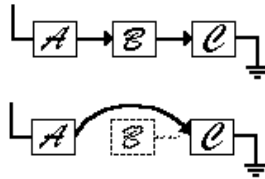
```
j  
m  
[a]-[v]-[a]-null
```

```
r  
a  
r  
v  
r  
a  
m  
null
```

```
q
```

## //COMENTARIOS

Os programadores que não estão acostumados com coleta automática de lixo podem achar estranho que retiramos o primeiro elemento da lista simplesmente perdendo propositalmente a referência para ele ao emendar o nó cabeça com o restante:



Não se preocupem, a linguagem cuida da desalocação.

## TIPO ABSTRATO DE DADOS

Tipo abstrato de dados, TAD, se preocupa em proporcionar uma abstração sobre uma estrutura de dados em termos de uma interface bem definida. O encapsulamento mantém a integridade do objeto evitando acessos inesperados. O fato de o código estar armazenado em um só lugar cria um programa modificável, legível, coeso.

A idéia é criar classes que ocultem a sua representação interna as quais podem geralmente ser modeladas através de “shopping list approach”. Por “shopping list approach” entenda que você pode modelar sua classe através das operações que ela suporta, ou seja, antes de começar a implementar é recomendável que se faça uma lista (semelhante a uma lista de compras) das operações desejadas para a classe e depois se escolha a representação interna adequada. Completando este conceito amplo podemos dizer que uma classe implementa um tipo abstrato de dados.

São exemplos de tipos abstratos de dados:

- Uma árvore binária com as operações usuais de inserção, remoção, busca ...
- Uma representação para números racionais (numerador, denominador) que possua as operações aritméticas básicas e outras de conversão de tipos.
- Uma representação para ângulos na forma (Graus, Minutos, Segundos). Também com as operações relacionadas, bem como as operações para converter para radianos, entre outras.

Tipo abstrato de dados é um conceito muito importante em programação orientada a objetos e por este motivo é logo apresentado neste tutorial. Os exemplos seguintes são simples por não podermos usar todos os recursos da linguagem ainda. Dada esta importância, a medida em que formos introduzindo novos conceitos exemplificaremos com aplicações na implementação tipos abstratos de dados.

## Exercícios:

1-

Use a estratégia da lista de compras (“shopping list approach”) para modelar a interface do tipo abstrato de dados **Ponto**, tente pensar nas operações que geralmente se aplicam a pontos em geometria, tais como distância a outros elementos, rotação em torno de outro ponto. Reimplemente este TAD adicionando as inúmeras alterações.

## TAD FRAÇÃO

Neste exemplo implementaremos o tipo abstrato de dados fração. Baseado no conceito de número racional do campo da matemática. Algumas operações não foram implementadas por serem semelhantes às existentes.

Uma aplicação deste TADs consiste em alguns cálculos onde pode ocorrer muita perda de precisão ao longo do programa devido ao uso de aritmética de ponto flutuante. Por exemplo: faça exatamente o seguinte em sua calculadora:  $5 / 3 * 3$ , qual é o resultado? Ao terminar este programa teste a seguinte operação com frações  $(5/3)*(3/1)$ , qual será o resultado?

## RESUMO DAS OPERAÇÕES MATEMÁTICAS ENVOLVIDAS:

- Simplificação de fração:  $(a/b) = (a/\text{mdc}(a,b)) / (b/\text{mdc}(a,b))$
- Onde  $\text{mdc}(a,b)$  retorna o máximo divisor comum de  $ab$ .
- Soma de fração:  $(a/b) + (c/d) = (a.d + c.b) / b.d$  simplificada.
- Multiplicação de fração:  $(a/b) * (c/d) = (a*c) / (b*d)$  simplificada.
- Igualdade:  $(a/b) == (c/d)$  se  $a*d == b*c$ .
- Não igualdade:  $(a/b) != (c/d)$  se  $a*d != b*c$
- Maior ou igual que:  $(a/b) \geq (c/d)$  se  $a*d \geq b*c$

## “SHOPPING LIST APPROACH” PARA O TAD FRAÇÃO:

(O conjunto de operações implementadas está marcado com ☒. A finalização dessa shopping list bem como do programa é deixada como exercício, o qual não deve ser difícil pois vendo a implementação da soma, o leitor obtém quase que de maneira direta a implementação da subtração, o mesmo ocorre para as demais operações):

- ☒ Construtor (recebe dois argumentos numéricos inteiros)
- ☒ Simplificação da fração (divisão do numerador e denominador por máximo divisor comum)
- ☒ Soma de fração (método recebendo argumento do próprio tipo fração)
- ☒ Subtração de fração
- ☒ Multiplicação

- ☒ Divisão
- ☒ Teste de igualdade
- ☒ Teste de desigualdade
- ☒ Teste de maior ou igual que
- ☒ Teste de menor ou igual que
- ☒ Teste de maior que
- ☒ Teste de menor que
- ☒ Impressão na tela
- ☒ Rotina de criação com entrada de numerador e denominador pelo teclado
- ☒ Conversão para **double**
- ☒ Conversão para **long**
- ☒ Operação de alteração do numerador
- ☒ Operação de alteração do denominador
- ☒ Retorno do valor do numerador e denominador
- ☒ Outras operações que o leitor julgar necessárias

## TÓPICOS ABORDADOS:

Construtores em geral, criação de métodos de conversão de tipos, chamadas de métodos do mesmo objeto, operador % que retorna o resto da divisão de dois inteiros.

## CONSIDERAÇÕES DE PROJETO:

A representação escolhida para o numerador e o denominador da fração será baseada no tipo `int`.

O formato escolhido para os métodos que implementam as operações é:

**TipoDoValorDeRetorno NomeDaOperacao(TipoDoOperando ValorDoOperando);**

Nesse formato um dos operandos é a própria fração que está recebendo a chamada de método o outro é passado como argumento. Outros formatos equivalentes poderiam ter sido adotados. Um dos possíveis formatos faz com que os dois operandos sejam passados como argumentos e a fração que está recebendo a chamada de método executa a operação para esses argumentos retornando o valor. Voltaremos a discutir essas alternativas de implementação.

Se durante o processo de construção de seu programa, ocorrer a repetição de um certo trecho de código nos diversos métodos (repetição da rotina de simplificação nos métodos de soma, subtração), considere a opção de definir este trecho de código como um método em separado. Se este método não for um método que deva compor, participar, da interface, mas que ainda assim tem seu padrão muito repetido, considere a possibilidade de defini-lo como **private** (método `mdc`).

Não existe uma regra de ouro que diga exatamente como projetar as suas classes para que elas preencham requisitos de portabilidade, robustez, flexibilidade. Todavia uma recomendações importantes podem ser feitas para evitar reformulações durante o processo de programação: “Não economize tempo na fase de projeto. Procure antes de programar, simular o uso de uma classe, seja mentalmente ou através de um protótipo.”

O cálculo do máximo divisor comum (`mdc`) de dois inteiros não tem nada a ver com as operações a serem oferecidas por frações, teria a ver com as operações oferecidas por um objeto facilidades de cálculos (`mdc(a,b)` , `fatorial(b)` , `fibonaci(x)`, `combinacoes(n,k)` ). No entanto a classe fração precisa da operação `mdc`. Ocorre que já estudamos uma maneira de implementar um método em uma classe e não oferecê-lo através da interface, é o qualificador **private**. Em C++

provavelmente o programador implementaria **mdc** como uma função isolada.

## IMPLEMENTAÇÃO:

### CÓDIGO

$$\frac{12}{3} + \frac{12}{7}$$

```
//TAD fracao.  
//File Fracao.java  
  
class Fracao {  
  
    private int num,den;                                //numerador, denominador  
  
    private int mdc(int n,int d)                        //metodo private maximo divisor comum  
                                                    //metodo de Euclides +- 300 anos AC.  
    {  
        if (n<0) n=-n;  
        if (d<0) d=-d;  
        while (d!=0) {  
            int r=n % d;                                //%=MOD=Resto da divisao inteira.  
            n=d;  
            d=r;  
        }  
        return n;  
    }  
  
    public Fracao(int t,int m)                          //construtor comum  
    {  
        num=t;  
        den=m;  
        this.simplifica();                             //chamada para o mesmo objeto.  
    }  
  
    public void simplifica()                            //divide num e den pelo mdc(num,den)  
    {  
        int commd;  
        commd=mdc(num,den);                             //divisor comum  
        num=num/commd;  
        den=den/commd;  
        if (den<0) { den=-den; num=-num;};             //move sinal para cima  
    }  
  
    //operacoes matematicas basicas  
    public Fracao soma (Fracao j)  
    {  
        Fracao g;  
        g=new Fracao((num*j.den)+(j.num*den),den*j.den);  
    }  
}
```

```

return g;
}

public Fracao multiplicacao(Fracao j)
{
    Fracao g;
    g=new Fracao(num*j.num,den*j.den);
    return g;
}

//operacoes de comparacao

public boolean igual(Fracao t)
{ return ((num*t.den)==(den*t.num)); }

public boolean diferente(Fracao t)
{ return ((num*t.den)!=den*t.num); }

public boolean maiorouigual(Fracao t)
{ return ((num*t.den)>=(t.num*den)); }

//operacoes de input output

public void mostra() //exibe fracao no video
{ System.out.println("(" + num + "/" + den + ")"); }

//operacoes de conversao de tipos

public double convertedbl() //converte para double
{
    double dbl;
    dbl=((double)num/(double)den);
    return dbl;
}

public int converteint() //converte para int
{
    int itng;
    itng=num/den;
    return itng;
}

```

```
public void altera_num(int nn)
{ num=nn; }
```

```
public void altera_den(int nd)
{ den=nd; }
```

```
public int retorna_num()
{ return num; }
```

```
public int retorna_den()
{ return den; }
```

```
}
```

```
class Principal {
```

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

```
{
```

```
    Fracao a,b,c;
```

```
    a=new Fracao(5,3);
```

```
    b=new Fracao(2,6);
```

```
    System.out.print("Esta e' a fracao a: ");
```

```
    a.mostra();
```

```
    System.out.print("Esta e' a fracao b: ");
```

```
    b.mostra();
```

```
    c=a.soma(b);
```

```
    System.out.print("c de a+b: ");
```

```
//c(a+b)
```

```
    c.mostra();
```

```
    System.out.print("a*b: ");
```

```
    c=a.multiplicacao(b);
```

```
    c.mostra();
```

```
    System.out.print("a+b: ");
```

```
    c=a.soma(b);
```

```
    c.mostra();
```

```
    System.out.print("a>=b: ");
```

```
    System.out.println(a.maiorouigual(b));
```

```
    System.out.print("a==b: ");
```

```
    System.out.println(a.igual(b));
```

```
    System.out.print("a!=b: ");
```

```
    System.out.println(a.diferente(b));
```

```

System.out.print("(int)a ");
System.out.println(a.converteint());

System.out.print("(double)a ");
System.out.println( a.convertedbl());

}

}

```



Esta e' a fracao a: (5/3)  
 Esta e' a fracao b: (1/3)  
 c de a+b: (2/1)  
 a\*b: (5/9)  
 a+b: (2/1)  
 a>=b: true  
 a==b: false  
 a!=b: true  
 (int)a 1  
 (double)a 1.66667

## //COMENTARIOS:

Uma implementação completa do tipo de dados fração tem que checar por “overflow” do numerador e denominador o que ocorre frequentemente quando se trabalha com números primos entre si (não podem ser simplificados), uma possível solução para este problema é fazer uma aproximação e/ou alterar a representação interna da fração para um tipo com maior numero de bits (de int para long). De qualquer forma estas extensões são exercícios avançados pois a última delas envolve uso de herança.

## Exercícios:

1-

Complete o tipo fração com os métodos faltantes da “shopping list approach”

```

long retorna_den(void) {return den;}
long altera_den(int a) {den=a;}

```

Considerando que os atributos declarados em **private** não são acessíveis fora da classe, descreva a utilidade desses métodos. Eles são úteis se usados pelos próprios métodos de fração?

<sup>11</sup>2-

Implemente o tipo abstrato de dados número complexo com as operações matemáticas inerentes. Faça antes um projeto dos métodos que serão implementados, descreva (detalhadamente) as operações matemáticas necessárias. Que forma de representação você escolherá: coordenadas polares

<sup>11</sup> Estes exercícios são considerados difíceis. É recomendável somente esboçar o projeto deles e depois, a implementação pode ser deixada como exercício das próximas seções.

ou retangulares?

3-

Pesquise sobre matrizes em **Java**: . Crie um tipo abstrato de dados matriz que suporte atribuições e leituras de células contendo elementos do tipo **float**. Crie outros métodos para este tipo abstrato de dados como multiplicação por uma constante.

4-

Implemente o tipo abstrato de dados relógio, pesquise as operações normalmente oferecidas por relógios reais, o único objetivo é marcar as horas. ☺ Se você precisar de inspiração para este exercício, consulte o exemplo da classe **Contador** e seus exercícios.

## STRINGS, UM MODELO DE CLASSE

Agora que já estamos programando alguns tipos abstratos de dados, está na hora de apresentar um exemplo que mostre como comportamento é importante para um TAD. Só que desta vez ficaremos do lado do cliente, do usuário desse tipo abstrato de dados e não do lado do programador. Estudaremos a classe **String** oferecida pela linguagem.

Nos tutoriais desta série, feitos para outras linguagens (C++, Modula-3), recomendávamos como exercício a implementação do tipo abstrato de dados string, que deveria suportar operações de concatenação, substring, acesso a elemento, etc. Este exercício não faz sentido em **Java** porque o tipo **String**, é fornecido com a linguagem como uma classe da package **java.lang** que é importada implicitamente em todos os programas além disso a sua implementação desta classe é bastante completa.

A declaração de **Strings** se dá da mesma forma que os outros objetos: **String minhaString**; . O compilador oferece uma facilidade sintática para a inicialização com valores literais:

```
String teste="Ola meu amigo"; //objeto instanciado com valor Ola meu amigo
```

Para concatenar **Strings** use o operador **+**. Os operandos podem não ser **Strings**, nesse caso serão convertidos para objetos desta classe, por exemplo se um dos argumentos for um inteiro, o objeto **String** correspondente conterá o valor literal deste inteiro.

```
System.out.println(teste + " Andre!"); //Ola meu amigo Andre!
```

```
teste+=" Andre!"; //atalho para concatenacao seguida de atribuicao: teste=teste+" Andre!"
```

```
System.out.println(teste); //totalmente equivalente a primeira
```

Para obter o comprimento em número de caracteres de uma **String**, chame o método **length()** para a **String** em questão. Para obter o caractere presente na posição 6 da **String**, chame o método **charAt()**; . Note que o primeiro caractere da **String** está na posição zero:

```
char umChar=teste.charAt(6); //um char recebe 'u'
```

Para obter uma substring, chame o método **substring(int a,int b)**; onde o primeiro argumento é o índice do início da substring e o segundo é o índice do fim da substrings, os caracteres em a e b também são incluídos:

```
String aStr=teste.substring(0,2); //aStr recebe ola
```

Para transformar todos os caracteres de uma **String** em letras maiúsculas basta chamar o método **toUpperCase()**;

```
teste=teste.toUpperCase(); //teste fica igual a OLA MEU AMIGO
```

Um método interessante para usar em checagem de padrões em texto é **indexOf(String busque)**; . Este método retorna o índice posição inicial de ocorrência de busque na **String** para a qual foi chamado o método:

```
teste.indexOf("MEU"); //retorna 4
```



Analogamente, `lastIndexOf(String busque)`, retorna o índice de ocorrência da substring, só que agora do procurando do fim para o começo.

```
teste.indexOf("M"); //resulta em 9 (logo a seguir do ultimo A que esta na posicao 8)
```

Para comparação de igualdade use:

```
test.equals("OLA MEU AMIGO"); //retorna valor booleano
```

Além disso, a classe `String` define métodos para conversão dos tipos básicos para seus valores na forma de `String`, você pode achar esses métodos um pouco estranhos, pois eles tem todos os mesmos nomes e não precisam de um objeto para serem chamados, eles são chamados para a classe:

```
String.valueOf(3); //argumento e naturalmente um inteiro, retorna "3"
```

```
String.valueOf(3.1415); //argumento e double, retorna "3.1415"
```

Os métodos de nome `valueOf` são uma padronização de métodos de conversão entre tipos encontrados em algumas das classes pré-definidas na linguagem, principalmente nas classes “wrappers” que foram exemplificadas com a classe `Integer`.

## CÓDIGO

```
class StringTest {
    public static void main (String args[]) {
        String teste="Ola meu amigo";

        System.out.println(teste + " Andre!"); //Ola meu amigo Andre!

        teste+=" Andre!"; //atalho para concatenacao seguida de atribuicao

        System.out.println(teste); //totalmente equivalente a primeira

        char umChar=teste.charAt(5); //um char receber 'e'

        System.out.println("Andre "+umChar+teste.substring(3,13));

        teste=teste.toUpperCase(); //teste fica igual a OLA MEU AMIGO ANDRE!

        for (int i=0;i<teste.length();i++) //imprimindo caracteres um a um
        {
            System.out.print(teste.charAt(i));
        }

        System.out.println(); //pula uma linha

        System.out.println(teste.indexOf("AMIGO")); //retorna 8

        System.out.println(teste.indexOf("biba")); //nao acha, retorna -1

        System.out.println(teste.lastIndexOf("AMIGO")); //retorna 8

        System.out.println(String.valueOf(3.1415f)); //Metodo chamado para a classe
    }
}
```



```
Ola meu amigo Andre!  
Ola meu amigo Andre!  
Andre e meu amigo  
OLA MEU AMIGO ANDRE!  
8  
-1  
8  
3.1415
```

## //COMENTARIOS

O código fonte disponível com o compilador constitui uma ótima fonte de aprendizado sobre como construir componentes de software reutilizáveis no estilo de programação orientada a objetos, leia-o sempre que estiver disponível e quando não estiver, preste atenção na sua interface.

Você deve ter notado que exceto pelo método `toUpperCase()` a classe `String` não permite alterações no seu estado depois de ter sido criada. Isto decorre da decisão do “Java team” de implementar duas classes para garantir a funcionalidade e segurança no uso de strings, são elas: `String` e `StringBuffer`. `StringBuffer` permite ser modificada, abaixo apresentamos uma tabela de seus principais métodos, para maiores informações consulte a documentação de `java.lang.*`

Tabela de métodos da classe <code>StringBuffer</code>
<code>StringBuffer endereco="http://www.java.com";</code> //Erro! Esta facilidade para instanciação não é permitida para a classe <code>StringBuffer</code> , ao invés disso use o construtor descrito abaixo.
<code>StringBuffer endereco=new StringBuffer("http://www.java.com");</code> //agora sim esta' correto
<code>endereco.append("/fim.html");</code> //concatena e atribui, forma: “http://www.java.com/fim.html”
<code>endereco.charAt(5);</code> //retorna o caractere ‘l’
<code>System.out.println(endereco + "/outro.html");</code> //o operador+ também funciona para <code>StringBuffer</code>
<code>endereco.insert(15,"soft");</code> //forma “http://www.javasoft.com/fim.html”
<code>endereco.setCharAt(25,'O');</code> //forma “http://www.javasoft.com/fOm.html”
<code>System.out.println(endereco.toString());</code> //retorna objeto <code>String</code> equivalente

## Exercícios:

1-

Melhore um dos exercícios anteriores introduzindo um nome para o objeto. Este nome pode ir nas saídas de tela executadas pelo objeto e é o nome dado para o construtor como uma `String`. Não se esqueça de prover métodos como `String getName()`.

\*2-

Este é um exercício avançado. Busca de padrões em texto é uma operação bastante comum em editores. A busca de informações na internet quando feita por palavras chave, é executada por aplicativos que podem ter como estratégia a contagem do número de ocorrências de certas palavras nos textos cadastrados.

Neste exercício você fará parte desse trabalho de busca de informações, eu vou te orientar como: Crie um método que aceita como argumento uma `String` grande que pode ter vindo de várias fontes (não importa quais). Este método pode pertencer a uma classe de nome `EstatisticasDeTexto` e deve quebrar o argumento `String` em palavras separadas por espaços (você já pode fazer isso com os métodos conhecidos, mas talvez queira usar a classe `StreamTokenizer`). Para cada uma destas palavras este método faz uma busca na estrutura de armazenamento de palavras da classe, se a palavra

não constar, deve ser incluída com multiplicidade 1, caso ela conste na estrutura, apenas sua multiplicidade deve ser incrementada.

Um problema pode surgir. A estrutura deve ser capaz de crescer de tamanho durante as várias chamadas de métodos com Strings para serem processadas. Se você usar vetor uma opção é realocar o vetor com o dobro de espaços. Outra opção é usar uma lista ligada.

3-

Crie uma classe que agrega um **stream** de entrada de dados ligado ao teclado. Coloque esta classe como intermediária na recepção dos dados de modo que ela testa para todo caractere lido se ele é igual ao caractere '\n'. Se o caractere lido for igual a '\n' ele deve ser retirado da String lida (use os métodos substring e concatenação ensinados).

Faça com que a chamada do método **readline** de sua classe dispare o a chamada do método **readline** do **stream** agregado nela.

Resumindo : sua classe atuará como um filtro de dados do teclado. Mantendo a analogia do exemplo introdutório sobre **Streams**, estaremos lidando com conexões entre canos, ou seja para os dados chegarem do teclado ao seu programa eles devem passar pela sua classe filtro.

## TAD E ALOCAÇÃO DINÂMICA.

Este exemplo cria um tipo abstrato de dados matriz bidimensional de inteiros (int). O leitor pode achar estranho que para representar esta matriz usamos um vetor, mas é que isto traz algumas vantagens:

### Representação linear de uma matriz:

Pode-se representar uma matriz de qualquer dimensão em um vetor. Veja o exemplo de uma matriz bidimensional de inteiros mostrada no formato **indiceLinear:valorArmazenado**. Os índices lineares vão de 1 até  $n^2$ .

Matriz:

1:3	2:32	3:1
4:23	5:90	6:12
7:21	8:08	9:32

Vetor equivalente:

1:3	2:32	3:1	4:23	5:90	6:12	7:21	8:08	9:32
-----	------	-----	------	------	------	------	------	------

Vantagem da representação linear (vetor): para referenciar uma posição gasta-se somente um inteiro contra dois da representação matriz. Pode-se considerar posições que apontam para outras posições, basta interpretar o conteúdo do vetor como um índice linear. Este tipo de construção pode ser útil em **Java**, pois a linguagem não possui ponteiros este é um dos motivos de estarmos ensinando esta técnica.

Desvantagem da representação linear (vetor): é necessário criar funções de conversão de índice na forma (linha,coluna) para (índice linear) e de (índice linear) para (coluna) ou (linha). São as funções **lin** e **col** e **linear** deste exemplo. Em uma primeira leitura, não é preciso entender os cálculos com índices, apenas o uso dos métodos que oferecem estes cálculos.

Para nós, clientes da classe **Matriz2DInt**, os elementos serão indexados de 1 até **m** (arbitrário) em termos de índice linear. Em termos de linhas e colunas, eles serão indexados de (1,lmax) e de

(1, cmax). O fato da linguagem adotar índices de 0 até m-1 para matrizes e vetores não importa, nós construímos em volta dessa representação para que nosso objeto forneça, trabalhe na convenção mais natural de indexação para humanos: 1 até m. Quanto as operações de índices, apenas verifique a veracidade para valores arbitrários de uma matriz como a desenhada anteriormente tentar entendê-las leva tempo e eu sei que você é capaz de programa-las.

Dentre os objetivos de um programador de uma linguagem orientada a objetos podemos citar: escrever pouco código, escrever código correto, tornar o seu código reutilizável. A criação de componentes de software reutilizáveis, é enormemente facilitada pela portabilidade da linguagem Java. Programas exemplo posteriores (segunda parte) mostrarão como reutilizar esta classe matriz para a criação de um jogo de quebra cabeça de quadradinhos deslizantes. Os quadradinhos devem ser movidos na moldura de modo a formar uma imagem onde um dos quadrados é vazio.

### “SHOPPING LIST APPROACH” PARA O TAD MATRIZ2DInt:

(As operações implementadas estão marcadas com ☑. As operações marcadas com ☒ devem ser implementadas como exercício avançado cujo objetivo é completar um de seus primeiros componentes de software reutilizável, estamos incluindo desta vez os atributos também).

```
private int linhas; //numero de linhas da matriz
private int colunas; //numero de colunas da matriz
private int tam; //linhas*colunas
private int lc[]; //new int[linhas*colunas]=vetor[0..(tam-1)]=~matriz[l][c]
```

- ☑ Construtor (recebe dois argumentos numéricos inteiros, número de linhas e de colunas)
- ☑ Conversão de linha e coluna para índice linear
- ☑ Conversão de índice linear para coluna.
- ☑ Conversão de índice linear para linha.
- ☑ Operação de troca de elementos da matriz com dois argumentos do tipo índice linear.
- ☒ Operação de troca de elementos da matriz com argumentos do tipo (linha e coluna).
- ☑ Operação de atribuição a elemento da matriz indicado por índice linear.
- ☒ Operação de atribuição a elemento da matriz indicado por linha e coluna.
- ☑ Operação de retorno do conteúdo de posição da matriz indicada por um índice linear.
- ☒ Operação de retorno do conteúdo de posição da matriz indicada por linha e coluna.
- ☑ Representação do número de colunas, permitindo acesso de leitura ao cliente.
- ☑ Representação do número de linhas, permitindo acesso de leitura ao cliente.



### CÓDIGO Programa exemplo da classe Matriz2DInt.

```
class Matriz2DInt {
    private int linhas; //numero de linhas da matriz
    private int colunas; //numero de colunas da matriz
    private int tam; //linhas*colunas
    private int lc[]; //new int[linhas*colunas]=vetor[0..(tam-1)]=~matriz[l][c]

    public Matriz2DInt(int l,int c)
    //cria matriz LxC
    {
        lc=new int[l*c]; //l,c dimensoes ; lc vetor[l*c]
```

```
linhas=l;
colunas=c;
tam=linhas*colunas;
}
```

//qualquer uma das funcoes abaixo retorna int negativo se nao obteve sucesso

```
public int linear(int alin,int acol)
//ind linear a partir de linha e coluna
//nao modifica nenhum atributo
{
    int result; //valor de retorno para todos os metodos ...
    if ( (0<alin) && (alin<=linhas) && (0<acol) && (acol<=colunas) )
        { result=(alin-1)*colunas+acol; }
    else
        { result=-1; }
    return result;
}
```

```
public int col(int indlin)
//coluna a partir do indice linear
//nao modifica nenhum atributo da classe
{
    int result;
    if ( (0<indlin) && (indlin<=tam) )
        { result=(indlin % colunas);
          if (result==0)
            { result=colunas; }
        }
    else
        { result=-1; }
    return result;
}
```

```
public int lin(int indlin)
//linha a partir do indice linear
//nao modifica nenhum atributo da classe
{
    int result;
    if ( (0<indlin) && (indlin<=tam) )
        { result=(int)( ( (indlin-1)/colunas )+1 ); }
    else
        { result=-1; }
    return result;
}
```

```
public boolean trocaindlin(int i,int j)
```

```

//argumentos: 2 indices lineares
// retorna se conseguiu,ou nao conseguiu(false)
{
    int aux; //auxiliar na troca
    if ( (0<i) && (i<=tam) && (0<j) && (j<=tam) )
    {
        aux=lc[i-1];    //efetua a troca
        lc[i-1]=lc[j-1]; //embora para usuario a matriz vai de 1 ate l*c
        lc[j-1]=aux;    //para mim vai de 0 ate l*c-1
        return true; //sucesso
    }
    else
    { return false; } //falhou
}

public boolean atribuiindlin(int i,int v)
//atribui v ao indice i
//retorna true se conseguiu, false nao conseguiu
{
    if ( (0<i) && (i<=tam) )
    {
        lc[i-1]=v; //efetua a atribuicao
        return true;
    }
    else
    { return false; } //falhou
}

public int retornaindlin(int indlin)
//retorna conteudo do indice i
//retorna -1 se nao conseguiu
{
    int result;
    if ( (0<indlin) && (indlin<=tam) )
    { result=lc[indlin-1]; }
    else
    { result=-1; }
    return result;
}

public int getl()
//retorna numero de linhas
{
    return linhas;
}

public int getc()
//retorna numero de colunas

```

```

{
    return colunas;
}

public int gett()
//retorna tamanho
{
    return tam;
}
}

```

//Classe principal, Arquivo Principal.java

```

class Principal {

    public static void main(String args[]) {

        Matriz2DInt teste;
        teste=new Matriz2DInt(5,10); //5 linhas 10 colunas
        for(int i=1;i<teste.gett();i++) {teste.atribuiindlin(i,0); }
        System.out.println("linear(5,5)="+ teste.linear(5,5) );
        System.out.println("Atribuindo 2 a posicao (5,5)");
        teste.atribuiindlin(teste.linear(5,5),2);
        System.out.println("Atribuindo 4 a posicao (4,2)");
        teste.atribuiindlin(teste.linear(4,2),4);
        System.out.println("Trocando estas posicoes");
        teste.trocaindlin(teste.linear(5,5),teste.linear(4,2));
        System.out.println("Conteudo da posicao (5,5):"+teste.retornaindlin(teste.linear(5,5)));

    }

}

```



linear(5,5)=45  
 Atribuindo 2 a posicao (5,5)  
 Atribuindo 4 a posicao (4,2)  
 Trocando estas posicoes  
 Conteudo da posicao (5,5):4

### Matrizes definidas na linguagem:

As matrizes definidas pela linguagem seguem uma sintaxe de declaração e uso semelhante a sintaxe de vetores:

```

int custos[][]=new int [20,30]; //vinte por trinta, nao importa qual e linha qual e coluna
custos[0][0]=145;
int a=custos[0][0];

```

### Curiosidade:

Houve um caso de vírus na internet que se baseava no acesso a índices fora de um vetor para gravar por cima de instruções do sistema operacional o código que garantisse a sua multiplicação. Não por acaso, **Java**, impede o acesso a posições fora do vetor.

### **Dica de programação:**

Saiba que uma prática bastante útil na fase de testes de um programa é introduzir mensagens informativas em pontos convenientes. Quando trabalhando com objetos tal prática pode ser usada de vários modos, por exemplo pode-se inserir uma mensagem no construtor de uma classe para verificar quando os objetos são criados e se são criados corretamente.

### **Exercícios:**

1-

Melhore a classe matriz para aceitar nas suas funções argumentos do tipo (linha,coluna) e não só índices lineares.

2-

Note que não foram impostas restrições para índices inválidos da matriz de modo que nosso componente de software falha se não usado corretamente. Adote uma estratégia de detecção de índices inválidos e use-a de modo a evitar o travamento do programa. Veremos como fazer esta checagem de índices de forma segura e uniforme em “Exception handling”.

3-

Implemente um método chamado ordena para o tipo abstrato de dados matriz definido acima. Use qualquer algoritmo de ordenação para deixar a matriz ordenada como um vetor quebrado em várias linhas. De que forma a facilidade que a interface oferece de enxergar a matriz como um vetor facilita este processo?

Será que este método ordena é realmente imprescindível para esta classe? Os métodos criados até o momento já não formam um modelo computacional suficiente para que o cliente desta classe possa definir em seu código os métodos de ordenação de matrizes que desejar? A dúvida sobre a necessidade ou não de um método em uma classe é bastante frequente.

Guardadas as devidas proporções, programação orientada a objetos é como brincar com blocos de encaixar, ou legos. Existem blocos que não se encaixam, existem blocos que podem ser construídos através de outros menores, mas que ainda sim existem porque são bastante usados, e a tarefa de compo-los a partir de outros blocos torna-se inconveniente. Existem blocos sem os quais não se pode construir praticamente nada.

4-

Crie um método de nome **preenche**, que inicializa todas as posições da matriz com o valor de um de seus argumentos. Este método pode ser composto pelos métodos já implementados?

\*5-

Defina um programa chamado grandes que implementa o tipo abstrato de dados números grandes e inteiros, este tipo deve usar um vetor do tipo numérico que você achar conveniente para representar os algarismos. Talvez estudar circuitos lógicos (somadores, multiplicadores) o ajude a implementar as quatro operações matemáticas básicas para estes tipo em termos de “look ahead carrier” e outras técnicas de performance de implementação de operações. Se sua preocupação é com eficiência e espaço, você pode usar cada posição do vetor para representar mais de um dígito, mas haverá muito trabalho em termos de tratar overflow e underflows.



6-

Note que na alocação da matriz:

```
public Matriz2DInt(int l,int c)
```

, não é checado se os valores passados são maiores que 0, faça este teste.

Quando explicarmos “exception handling” você terá métodos melhores de lidar com esses “erros”.

## HERANÇA

Existe uma visão um pouco tacanha de orientação a objetos como uma simples maneira de organizar melhor o seu código. Essa visão é facilmente desmentida pelos conceitos de encapsulamento, interfaces, packages e outros já apresentados. Neste tópico apresentaremos o conceito de herança, fundamental para programação orientada a objetos e um dos fatores de sucesso desta como muito mais que uma simples maneira de organizar melhor seu código.

Um dos aspectos que distinguem objetos de procedimentos e funções é que o tempo de existência de um objeto pode ser maior do que o do objeto que o criou. Isto permite que em sistemas distribuídos objetos criados em um local, sejam passados através da rede para outro local e armazenados lá quem sabe na memória ou mesmo em um banco de dados.

### Curiosidade:

Existem classes que podem ser obtidas na Internet para fazer interface com bancos de dados SQL, servindo principalmente para facilitar esta faceta da programação na internet que é bastante limitada pelas restrições de segurança da linguagem.

## HIERARQUIAS DE TIPOS

Neste tópico mostraremos como construir hierarquias de tipo por generalização / especialização. Para entender o que é generalização especialização e as regras de atribuição entre elementos dessas hierarquias, acompanhe a seguinte comparação:

Se você vai a um restaurante e pede o prato de frutos do mar, é natural que você aceite uma lagosta com catupiry, ou então filé de badejo. Mas se o garçom lhe serve uma salada de tomates isto não se encaixa no pedido. Por outro lado, se o seu pedido for peixe, uma lagosta com catupiry, embora muito saborosa não serve mais<sup>12</sup>, assim como a salada. Note que peixe e lagosta são especializações de frutos do mar.

Generalização e Especialização são ferramentas para lidar com complexidade, elas são abstrações. Os sistemas do mundo real apresentam complexidade muito maior que ordenar um prato listado em um cardápio. O uso de generalização e especialização permite controlar a quantidade de detalhes presente nos seus modelos do mundo real, permite capturar as características essenciais dos objetos e tratar os detalhes de forma muito mais organizada e gradual.

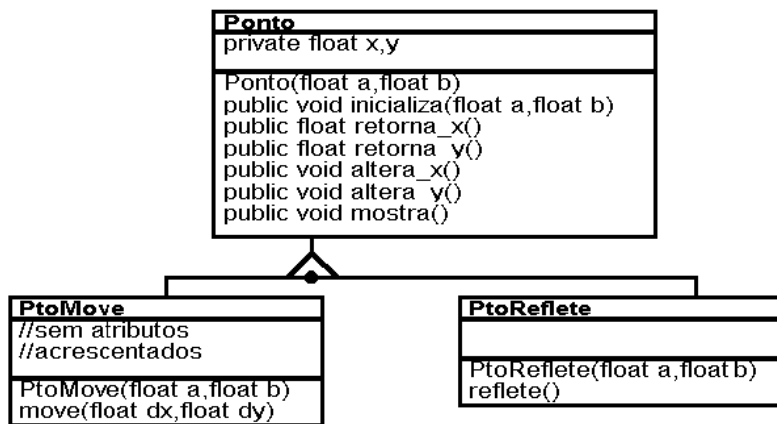
Existe muito mais para falar sobre herança, principalmente no que diz respeito a polimorfismo de inclusão e acoplamento dinâmico de mensagens, tópicos estes que serão abordados em separado.

## UMA HIERARQUIA SIMPLES.

Construiremos uma hierarquia de tipos simples para demonstrar herança pública em Java.

---

<sup>12</sup> lagosta é crustáceo não peixe.



### Comentários:

O diagrama acima representa a hierarquia de classes implementada neste exemplo e foi obtido a partir da janela de edição de uma ferramenta case para programação orientada a objetos.

A classe **ponto** que está no topo da hierarquia é chamada de classe base, enquanto que as classes **ponto\_reflete** e **ponto\_move** são chamadas classes filhas ou herdeiras. As classes da hierarquia são simples, **ponto\_move** apresenta o método **move**, já abordado neste tutorial em , **ponto\_reflete** apresenta o método (**reflete**) que inverte o sinal das coordenadas.

Dada a simplicidade das classes o leitor poderia se perguntar, porque não juntar as três em uma só. A pergunta faz sentido, mas e se quiséssemos criar uma classe **Ponto** que não se movesse, apenas refletisse e outra que só se movesse? E se quiséssemos projetar nosso programa segundo uma hierarquia de especialização / generalização da classe **Ponto**? O exemplo mostra como fazê-lo.

Na herança as classes filhas passam a atender pelos mesmos métodos e atributos **public** da classe pai, as classes filhas podem acrescentar métodos, atributos e até redefinir métodos herdados (veremos mais tarde). Por isso é que se diz que as classes subclasses garantem pelo menos o comportamento “behaviour” das superclasses, podendo acrescentar mais características. Os atributos encapsulados (**private**) da classe pai não são acessíveis diretamente na classe filha a não ser que sejam qualificados como **protected** ou **public**, veja .


Diagrama de acesso, visibilidade, dos elementos da classe pai para uma classe filha ou herdeira. Os atributos e métodos da classe pai são classificados quanto ao encapsulamento. A parte sombreada significa não visível, encapsulado.

\*As duas são consideradas como sendo do mesmo package.

PUBLIC
PRIVATE
PROTECTED
“PACKAGE”*

### Construtores e herança:

No construtor de uma classe filha o programador pode incluir a chamada do construtor da classe pai existente nela. Para referenciar a classe pai use a “keyword “ **super** de modo analogo a **this** (objeto corrente).

 **CÓDIGO** Hierarquia de generalização e especialização.

```
//Classe Ponto
```

```
class Ponto {
```

```
    private float x,y;
```

```

public Ponto(float ax,float ay) //omita o valor de retorno!
//garante o estado do objeto
{
    this.x=ax; this.y=ay;
}

public void inicializa(float a,float b)
{
    this.x=a; this.y=b;
}

public float retorna_x()
{
    return x;
}

public float retorna_y()
{
    return y;
}

public void altera_x(float a)
{
    this.x=a;
}

public void altera_y(float b)
{
    this.y=b;
}

public void mostra()
{
    System.out.println( "(" + this.x + "," + this.y + ")" );
}
}

```

```

//Classe PtoMove

class PtoMove extends Ponto {

//adicione algum atributo private se quiser

public PtoMove(float a,float b)
{
    super(a,b); //chamada do construtor da classe pai
}
}

```

```
public void move(float dx,float dy)
{
    this.altera_x(retorna_x()+dx);
    this.altera_y(retorna_y()+dy);
}
}
```

```
//Classe PtoRelete

class PtoRelete extends Ponto {

//adicione algum atributo private se quiser

public PtoRelete(float a, float b)
//construtor
{
    super(a,b); //chamada de construtor da classe pai ou superclasse
}

void relete()
//troca o sinal das coordenadas
{
    this.altera_x(-retorna_x());
    this.altera_y(-retorna_y());
}

}
```

```
//Classe principal, Arquivo Principal.java

class Principal {

public static void main(String args[]) {
    PtoRelete p1=new PtoRelete(3.14f,2.72f);
    System.out.println("Criando PontoRelete em 3.14,2.72");
    p1.relete();
    System.out.println("Refletindo este ponto.");
    p1.mostra();
    PtoMove p2=new PtoMove(1.0f,1.0f);
    System.out.println("Criando PontoMove em 1.0,1.0");
    p2.move(.5f,.5f);
    System.out.println("Movendo este ponto de 0.5,0.5");
    p2.mostra();
}
}
```

```
}
```



Criando PontoReflete em 3.14,2.72  
Refletindo este ponto.  
(-3.14,-2.72)  
Criando PontoMove em 1.0,1.0  
Movendo este ponto de 0.5,0.5  
(1.5,1.5)

## //COMENTARIOS

Os atributos `x` e `y` da classe `Ponto` estão declarados como **private**. Pelo diagrama anterior ao programa, atributos **private** não são visíveis aos descendentes na hierarquia. E de fato nós alteramos esses atributos (reflete e move) através de chamadas a métodos **public** da classe pai, chamadas de métodos que impliquem em acesso indireto a atributos da mesma classe parecem ser um pouco ineficientes. Lembre-se que os métodos **public** sempre são visíveis.

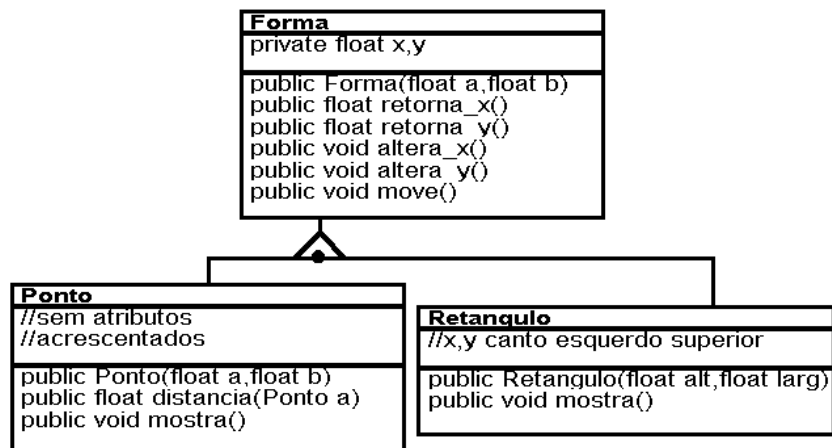
## Nota aos programadores C++:

Java não permite herança **private**, a clausula **extends** equivale a herança pública de C++ e só. Para obter o efeito de herança **private** use agregação.

## Exercícios:

1-

Programa e teste a hierarquia representada abaixo:



A classe `forma` não tem um significado prático, é uma abstração, você não pode desenhar (mostrar) uma forma, no entanto neste programa você poderá instanciá-la. Esta classe está na hierarquia, somente para capturar as características comuns a `Ponto` e `Retangulo`. Em aprenderemos como definir estas classes abstratas de forma mais condizente com o paradigma de orientação a objetos.

O método `mostra` deve imprimir na tela os atributos destas classes.

**PROTECTED**

Quando vimos o tópico encapsulamento, foi mencionado que **private** era o modo de encapsulamento mais restritivo, seguido de **protected**, **package** e depois **public** (o mais aberto). Naquele tópico mostramos um exemplo para cada tipo de encapsulamento, exceto **protected** que depende da existência de uma hierarquia para ser demonstrado.

Igual ao exemplo anterior, mas agora tornando os atributos da classe pai acessíveis para as classes filhas através do uso de **protected**. **Protected** deixa os atributos da classe pai visíveis, acessíveis “hierarquia abaixo”. Mas para o restante do programa tem o mesmo efeito que **private**.

Outra frase sobre **protected**: “A herança permite que uma subclasse ganhe acesso a declarações **protected** de sua superclasse, mas o usuário não percebe isso, para o usuário (uma classe externa) o que continua existindo é o que é **public**”.

Diagramas de acesso, visibilidade, de atributos e métodos de uma classe pai para uma classe filha ou herdeira:

Para uma classe filha em outro package (você herdando de uma classe pronta em Java)

PRIVATE
PROTECTED
“PACKAGE”
PUBLIC

O que o restante do programa vê das declarações da classe pai na classe filha.  
(por restante do programa entenda: outros packages e outras hierarquias)

PRIVATE
PROTECTED
“PACKAGE”
PUBLIC

 **CÓDIGO** O mesmo exemplo só que usando **protected**.

```
//Classe Ponto

class Ponto {

    protected float x,y;

    public Ponto(float ax,float ay) //omite o valor de retorno!
    //garante o estado do objeto
    {
        this.x=ax; this.y=ay;
    }

    public void inicializa(float a,float b)
    {
        this.x=a; this.y=b;
    }

    public float retorna_x()
    {
        return x;
    }

    public float retorna_y()
```

```
{
    return y;
}

public void altera_x(float a)
{
    this.x=a;
}

public void altera_y(float b)
{
    this.y=b;
}

public void mostra()
{
    System.out.println( "(" + this.x + "," + this.y + ")" );
}
}
```

```
//Classe PtoMove

class PtoMove extends Ponto {

//adicione algum atributo private se quiser

public PtoMove(float a,float b)
{
    super(a,b);
}

public void move(float dx,float dy)
{
    x=x+dx; //aqui continuam acessiveis, em main nao
y=y+dy; //acesso direto, sem passar por metodo
}

}
```

```
//Classe PtoReflete

class PtoReflete extends Ponto {

//adicione algum atributo private se quiser

public PtoReflete(float a, float b)
{

```



```

super(a,b); //chamando o construtor da classe pai
}

void reflete()
{
    x=-x;
    y=-y;
}

}

```

```

//Classe principal, Arquivo Principal.java

class Principal {

    public static void main(String args[]) {
        PtoReflete p1=new PtoReflete(3.14f,2.72f);
        System.out.println("Criando PontoReflete em 3.14,2.72");
        p1.reflete();
        System.out.println("Refletindo este ponto.");
        p1.mostra();
        PtoMove p2=new PtoMove(1.0f,1.0f);
        System.out.println("Criando PontoMove em 1.0,1.0");
        p2.move(.5f,.5f);
        System.out.println("Movendo este ponto de 0.5,0.5");
        p2.mostra();
    }

}

```

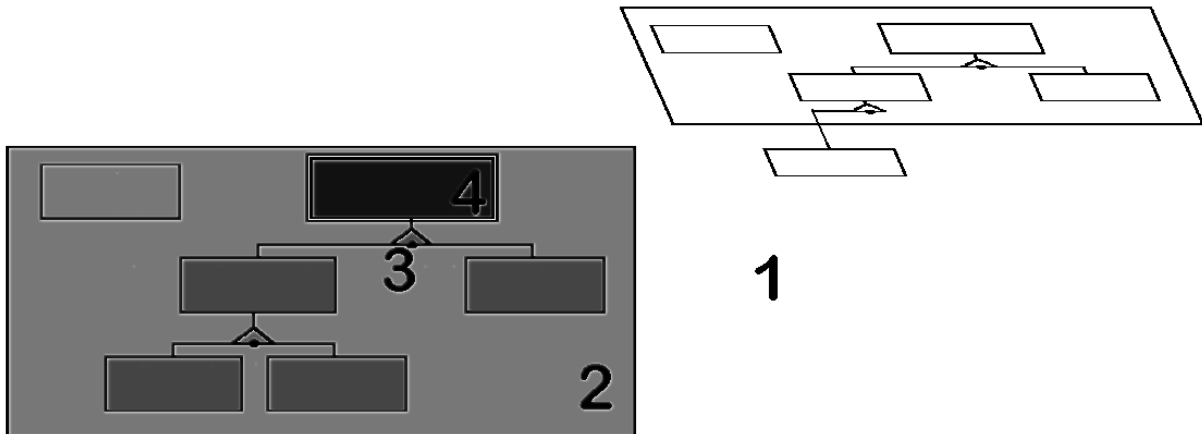


Criando PontoReflete em 3.14,2.72  
 Refletindo este ponto.  
 (-3.14,-2.72)  
 Criando PontoMove em 1.0,1.0  
 Movendo este ponto de 0.5,0.5  
 (1.5,1.5)

O qualificador **protected** termina com os modos de encapsulamento de atributos e métodos, portanto faremos uma revisão final, utilize os números dessa tabela para checar visualmente o tipo de encapsulamento no diagrama a seguir. Você vai aprender a olhar para este diagrama e enxergar tudo o que aprendemos nesse assunto, os atributos e métodos que estamos preocupados em encapsular são os da classe mais escura:

MODO	REPRESENTAÇÃO	LIMITE DE VISIBILIDADE
4)private	Representação no diagrama: a	Este é o nível de encapsulamento mais restritivo. A visibilidade das declarações limita-se ao envoltório da classe.

3) <b>protected</b>	Representação no diagrama: a hierarquia abaixo da classe escura.	A visibilidade das declarações se limita própria classe e as classes herdeiras dela.
2) Nada especificado “package”	Representação no diagrama: retângulo envolvendo as classes pintadas.	A visibilidade das declarações se limita a própria classe e as classes do mesmo package, mas não às classes herdeiras, . Classes herdeiras não precisam ser do mesmo package.
1) <b>public</b>	Representação no diagrama: todas as classes.	Estas declarações são sempre acessíveis.



O diagrama acima mostra as áreas de visibilidade de cada tipo de modificador aplicado aos atributos da classe destacada. Os retângulos grandes, representam os packages, optamos por construir a hierarquia da esquerda dentro do mesmo package o que nem sempre é feito. Normalmente quando você estende uma classe das packages que vem com a linguagem, sua classe herdeira não pertence a aquela package, mas pertence a hierarquia, saindo fora dos retângulos maiores como na hierarquia da direita. Suponha que todas as classes são declaradas como **public**. Existem algumas declarações de qualificadores de atributos que não fazem sentido com classes **private** e são erros de compilação.

## REDEFINIÇÃO DE MÉTODOS HERDADOS

Uma classe filha pode fornecer uma outra implementação para um método herdado, caracterizando uma redefinição “overriding” de método. Importante: o método deve ter a mesma assinatura (nome, argumentos e valor de retorno), senão não se trata de uma redefinição e sim sobrecarga “overloading”. A redefinição garante que o método terá o mesmo comportamento que o anterior isto faz com que as subclasses possam ser atribuídas a variáveis da superclasse pois atendem a todas as operações desta.

Este exemplo é igual ao exemplo anterior, mas agora redefinindo o método **mostra** para a classe filha **PtoReflete**. Na verdade este exemplo deveria pertencer ao tópico de polimorfismo, contudo, nos exemplos seguintes usaremos também redefinições de métodos, portanto faz-se necessário introduzi-lo agora. Teremos mais explicações sobre o assunto.

No nosso exemplo a classe **PtoReflete** redefina o método **mostra** da classe pai, enquanto que a classe herdeira **PtoMove** aceita a definição do método **mostra** dada pela classe **Ponto** que é sua classe pai.

## CÓDIGO

```
//Insira aqui o arquivo da classe Ponto do exemplo anterior: PROTECTED
```

```
//Insira aqui o arquivo da classe PtoMove do exemplo anterior: PROTECTED
```

```
//Classe PtoReflete

class PtoReflete extends Ponto {

//adicione algum atributo private se quiser

public PtoReflete(float a, float b)
{
    super(a,b); //chamando o construtor da classe pai
}

public void mostra()
{
    System.out.println( "X:" + this.x + " Y:" + this.y );
}

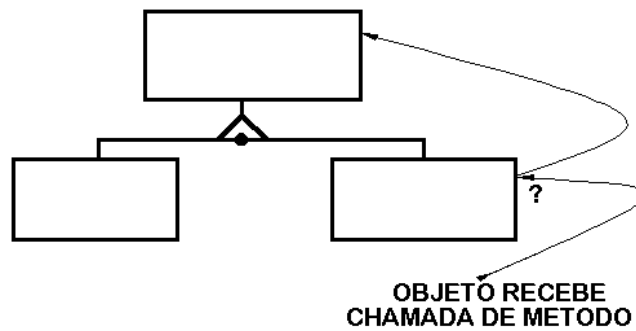
void reflete()
{
    x=-x;
    y=-y;
}

}
```

```
//Insira aqui o arquivo da classe Principal do exemplo anterior: PROTECTED
```

## //COMENTARIOS

No caso de redefinição de métodos, a busca da implementação do método a ser executado ocorre de baixo para cima na hierarquia. Exemplo: se você tem um objeto **PtoReflete** e chama o método **retorna\_x()** para ele, primeiro o compilador procura se **PtoReflete** possui ou não uma implementação para este método, no caso não possui, então a busca é feita nos métodos **public** da superclasse, onde a implementação de **retorna\_x()** é achada.



## Exercícios:

1-

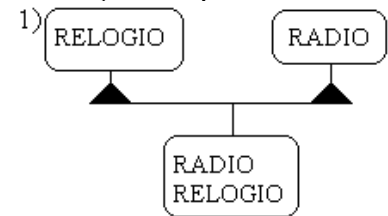
Teste redefinição de métodos colocando “System.out’s” em métodos da hierarquia, tais como:  
 System.out.println(“Metodo redefinido na classe X, chamado.”) ;

## INTERFACES, UMA ALTERNATIVA PARA HERANÇA MÚLTIPLA

Herança múltipla é a capacidade de uma classe herdar de duas ou mais classes, por exemplo a classe radio-relógio herdar da classe rádio e da classe relógio. C++ apresenta herança múltipla, e também maneiras de tratar os problemas decorrentes de seu uso. Um dos problemas que podem surgir é o conflito de nomes de atributos ou métodos herdados desse tipo de herança. Uma das estratégias adotadas para resolver estes conflitos é o “renaming” ou renomeamento desses nomes iguais presentes nas superclasses.

Tendo o seguinte significado: A classe herdeira tem comportamento, “behaviour”, semelhante ao das duas classes pais. Um outro exemplo de interface seria a classe audio-vídeo que herda da classe audio e da classe vídeo.

Herança múltipla:



**Java** por motivos de simplicidade, abandona a idéia de herança múltipla, cedendo lugar ao uso de interfaces. Interfaces são um conjunto de métodos e constantes (não contém atributos). Os métodos definidos na interface são “ocos” ou desprovidos de implementação. Classes podem dizer que implementam uma interface, estabelecendo um compromisso, uma espécie de contrato, com seus clientes no que se refere a prover uma implementação para cada método da referida interface.. Ao cliente, pode ser dada a definição da interface, ele acaba não sabendo o que a classe é, mas sabe o que faz. Quem programa em Objective C, deve ver as interfaces como algo semelhante ao conceito de protocolos.

Neste exemplo usaremos uma interface de nome imprimível para capturar as características comuns as classe que podem ser imprimidas em algum dispositivo de saída de dados.

### CÓDIGO Interfaces

```
public interface Imprimivel { //alem das classes, so interfaces pode ocupar um arquivo
```

```

        final char nlin="\n"; //nova linha
    public String toString();

        //forma preferida para impressao na tela

    public void toSystemOut();
    }

```

```

public class Produto implements Imprimivel {
//um produto comercial qualquer
protected String descricao;
protected int quantidade;

    public Produto(String d,int q)
    {
        descricao=d;
        quantidade=q;
    }

    public String toString()
    {
        return new String(" "+descricao+" "+quantidade);
    }
        //forma preferida para impressao na tela

    public void toSystemOut()
    {
        System.out.print(descricao + quantidade);
    }
}

```

```

//Classe principal, Arquivo Principal.java

class Principal {

    public static void main(String args[]) {
        Produto ump=new Produto("macarrao", 100);
        ump.toSystemOut();
        System.out.println();
        System.out.println(ump.toString());
    }
}

```



macarrao100  
macarrao 100

## //COMENTARIOS

O paradigma de orientação a objetos está refletido na capacidade de herança e encapsulamento das interfaces. No caso deste exemplo, a interface foi declarada como **public**, mas se nada fosse especificado ela pertenceria ao package dela, ou seja os modos de encapsulamentos são semelhantes aos de classes.

Uma **interface** poderia **estender** a **interface** `Imprimivel`:

```
interface Imprimivel2 extends Imprimivel {  
}
```

Interfaces tem sido representadas por retângulos de bordas arredondadas ligadas as classes que as implementam por linhas tracejadas.

Muitos confundem interfaces com classes e o ato de implementar uma interface com o ato de estender ou herdar uma classe. Por isso a relação entre interfaces e herança será explicada só agora, depois que você já pensou no assunto.

Uma classe `Produto2` herda da classe `Produto`(nosso exemplo) que implementa a interface `Imprimivel`. A classe produto já fez a parte difícil que é implementar a interface, agora a classe `Produto2` pode optar por aceitar ou redefinir os métodos herdados, ou seja: “A interface é um dos itens que é herdado de uma classe, assim como os atributos e métodos”.

### Exercícios:

1-

Defina uma interface para um conjunto de classes que representam figuras geométricas que podem ser desenhadas na tela.

## **POLIMORFISMO, CLASSES ABSTRATAS**

Existem várias classificações e tipos de polimorfismo. C++ apresenta vários tipos de polimorfismo. Java apresenta um conjunto mais reduzido evitando principalmente polimorfismos ad-hoc.

Polimorfismo, do grego: muitas formas. Polimorfismo é a capacidade de um operador executar a ação apropriada dependendo do tipo do operando. Aqui operando e operador estão definidos num sentido mais geral: operando pode significar argumentos atuais de um procedimento e operador o procedimento, operando pode significar um objeto e operador um método, operando pode significar um tipo e operador um objeto deste tipo.

## **REDEFINIÇÃO DE MÉTODOS PARA UMA CLASSE HERDEIRA**

Este exemplo já foi apresentado em . Também trata-se de um polimorfismo, pode ser classificado como polimorfismo de inclusão. Um método é uma redefinição de um método herdado, quando está definido em uma classe construída através de herança e possui o mesmo nome, valor de retorno e argumentos de um método herdado da classe pai. A assinatura do método tem que ser idêntica, ou seja, teremos redefinição quando uma classe filha fornece apenas uma nova implementação para o método herdado e não um novo método.

Se a classe filha fornecer um método de cabeçalho ou assinatura parecida com a do método herdado (difere ou no número ou no tipo dos argumentos, ou então no tipo do valor de retorno) então não se trata mais de redefinição, trata-se de uma sobrecarga, pois criou-se um novo método. Uma chamada ao método herdado não mais é interceptada por esse novo método de mesmo nome. O método tem o mesmo nome, mas é ligeiramente diferente na sua assinatura (o corpo ou bloco de código {} não importa), o que já implica que não proporciona o mesmo comportamento (behaviour) do método da superclasse.

## **SOBRECARGA ( MÉTODOS E OPERADORES)**

Este tipo de polimorfismo permite a existência de vários métodos de mesmo nome, porém com assinaturas levemente diferentes ou seja variando no número e tipo de argumentos e no valor de retorno. Ficaria a cargo do compilador escolher de acordo com as listas de argumentos os procedimentos ou métodos a serem executados.

## **SOBRECARGA DE MÉTODOS, “COPY CONSTRUCTOR”**

No exemplo a seguir vamos sobrecarregar o construtor de uma classe, esta classe passará a ter duas versões de construtores, vale lembrar que assim como o construtor será sobrecarregado, qualquer outro método poderia ser. O compilador saberá qual método chamar não mais pelo nome, mas pelos argumentos.

O método `Ponto(Ponto ap);` é um “copy constructor”, pois tem o mesmo nome que `Ponto(float dx,float dy);`. Tal duplicação de nomes pode parecer estranha, porém Java permite que eles coexistam **para uma mesma classe** porque não tem a mesma assinatura (nome+argumentos). Isto se chama sobrecarga de método, o compilador sabe distinguir entre esses dois construtores.

Outros métodos, não só construtores poderão ser sobrecarregados para vários argumentos diferentes, esse recurso é um polimorfismo do tipo “ad-hoc”.

O que é interessante para nós é o fato de o argumento do construtor **Ponto(Ponto ap)**; ser da mesma classe para qual o construtor foi implementado, o que caracteriza um “copy constructor” é que inicializa um objeto a partir de outro da mesma classe. Outros métodos semelhantes seriam: **Circulo(Circulo a)**; **Mouse(Mouse d)**; . Implementar copy constructor pode ser muito importante, lembre-se dos problemas com cópias de objetos apresentados em .

Por questões de espaço, basearemos nosso exemplo no tipo abstrato de dados fração, apresentado em . Você deve modificar a classe **Fracao** para que ela tenha dois construtores, o que esta em negrito deverá ser acrescentado ao código original:

```
public Fracao(int umso) //sobrecarga do construtor original
{
    num=umso;
    den=1; //subentendido
}

public Fracao(Fracao copieme) //esse e' um copy constructor e uma sobrecarga
{
    num=copieme.retorna_num();
    dem=copieme.retorna_den();
}

public Fracao(int t,int m) //construtor original
{
    num=t;
    den=m;
    this.simplifica();           //chamada para o mesmo objeto.
}
```

## CÓDIGO

```
//O programa principal, sobre a modificacao em negrito
class Principal {

public static void main(String args[])
{
    Fracao a,b,c;
    a=new Fracao(5); //elimine o segundo argumento
    b=new Fracao(2,6);
    System.out.print("Esta e' a fracao a: ");
    a.mostra();
    System.out.print("Esta e' a fracao b: ");
    b.mostra();
    c=a.soma(b);
    System.out.print( "c de a+b: "); //c(a+b)
    c.mostra();
}
```



```

System.out.print("a*b: ");
c=a.multiplicacao(b);
c.mostra();
System.out.print("a+b: ");
c=a.soma(b);
c.mostra();

System.out.print("a>=b: ");
System.out.println(a.maiorouigual(b));

System.out.print("a==b: ");
System.out.println(a.igual(b));

System.out.print("a!=b: ");
System.out.println(a.diferente(b));

System.out.print("(int)a ");
System.out.println(a.converteint());

System.out.print("(double)a ");
System.out.println( a.convertedbl());

}
}

```



Esta e' a fracao a: (5/1)  
 Esta e' a fracao b: (1/3)  
 c de a+b: (16/3)  
 a\*b: (5/3)  
 a+b: (16/3)  
 a>=b: true  
 a==b: false  
 a!=b: true  
 (int)a 5  
 (double)a 5

Teste o “copy constructor” para o tipo abstrato de dados fração apresentado acima. Quando um só número for passado para o construtor desta classe, subentende-se que o construtor chamado é o de um só argumento inteiro e que portanto o denominador será igual a 1.

Agora vamos falar do “copy constructor”, que embora implementado, não foi testado em `main()`. Esse método, pertence a outro objeto que não o argumento `copiame`, então para distinguir o atributo `num` deste objeto, do atributo `num` de `copiame` usamos `copiame.num` e simplesmente `num` para o objeto local, objeto em questão, ou objeto dono do método chamado.

## Exercícios:

1-

Faça um “copy constructor” para uma das classes já implementadas neste texto.

2-

Sobrecarregue o método `move` da classe `Ponto` para aceitar um `Ponto` como argumento, subentendendo-se que devemos mover a distância `x` e a distância `y` daquele ponto a origem.

3-

Crie um método de nome `unitarizado` para a classe `Ponto`. Este método deve interpretar o `Ponto` como um vetor e retornar um novo `Ponto` que contém as coordenadas do vetor unitarizado. Unitarizar é dividir cada coordenada pelo módulo do vetor. O módulo é a raiz quadrada da soma dos quadrados das componentes.

## **SOBRECARGA DE OPERADOR**

**Java** não fornece recursos para sobrecarga de operador, o que é perfeitamente condizente com a filosofia da linguagem. Seus criadores que acreditavam que a linguagem deveria ser pequena, simples, segura de se programar e de se usar (simple, small, safe and secure).

A ausência de sobrecarga de operadores pode ser contornada definindo apropriadamente classes e métodos.

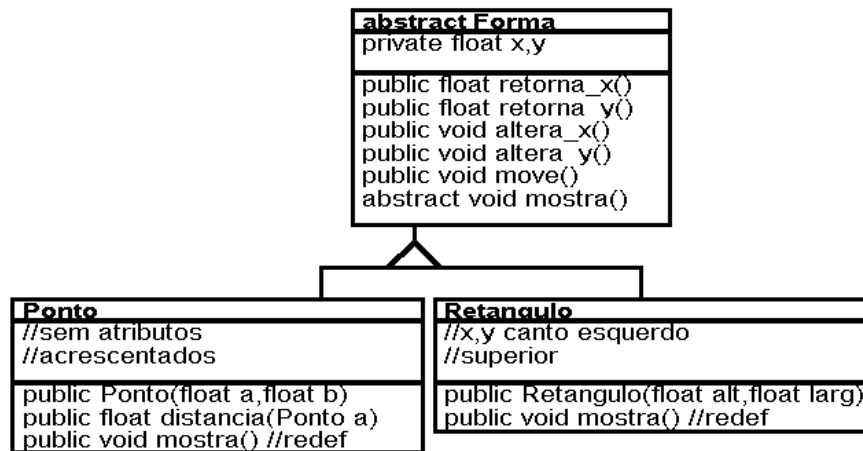
## **CLASSES ABSTRATAS E CONCRETAS**

Em um dos exercícios anteriores (no tópico sobre herança) pedíamos que você definisse uma hierarquia composta de três classes. A classe pai tinha o nome de `Forma`, e as classes herdeiras desta eram `Ponto` e `Retangulo`. Embora a classe `forma` não possuísse sentido prático, ela permitia certas operações como `move`, `altera_x(int nx)`, entre outras (retorne a este exercício).

Na verdade o que desejávamos era que esta classe `Forma` se comportasse como um esqueleto para as suas classes filhas, nós não queríamos instanciá-la. Classes abstratas permitem exatamente isto pois não podem ser instanciadas embora possam ser usadas de outras maneiras.

Classes abstratas são poderosas, elas permitem: criação de listas heterogêneas, ocorrência de “dynamic binding” e maior clareza no projeto de sistemas. Os packages que vem com a linguagem estão repletos de exemplos de classes abstratas.

Métodos abstratos, obrigatoriamente pertencem a classes abstratas, e são métodos desprovidos de implementação, são apenas definições que serão aproveitadas por outras classes da hierarquia. Voltando ao exemplo da hierarquia `Forma`, `Ponto` e `Retangulo`. O método `mostra` poderia ter sido definido na classe base abstrata (`Forma`) como um método abstrato.



## Classes abstratas

```

//Classe Forma

abstract class Forma {

protected float x,y; //visivel hierarquia abaixo

public void move(float dx,float dy)
{
    this.x+=dx; this.y+=dy;
}

abstract public void mostra(); //metodo abstrato
}
  
```

```

//Classe ponto

class Ponto extends Forma {

public Ponto(float ax,float ay) //omite o valor de retorno!
//garante o estado do objeto
{
    this.x=ax; this.y=ay;
}

//move nao precisa ser redefinido

public void mostra()
{
    System.out.println("(" + this.x + "," + this.y + ")");
}

}
  
```

```
//Classe Retangulo

class Retangulo extends Forma {

protected float dx,dy; //delta x e delta y
//protected acaba sendo menos inflexivel e mais eficiente que private

public Retangulo(float ax,float ay,float dx,float dy)
//garante o estado do objeto
{
x=ax; y=ay;
this.dx=dx; this.dy=dy; //this usado para eliminar ambiguidade
}

//metodo move precisa ser redefinido
public void move(float dx,float dy)
{
this.x+=dx; this.y+=dy;
this.dx+=dx; this.dy+=dy; //this distingue o argumento do atributo de mesmo nome
}

public void mostra()
{
System.out.println("("+this.x+","+this.y+")("+dx+","+dy+")");
}

}
```

```
//Classe principal, Arquivo Principal.java

class Principal {

public static void main(String args[]) {
Retangulo ar=new Retangulo(0,0,13,14);
ar.mostra();
ar.move(1,1);
ar.mostra();
}

}
```



(0,0)(13,14)  
(1,1)(14,15)

## //COMENTARIOS:

Observe que a classe **Forma** que é abstrata, não possui um construtor, porque não pode ser instanciada. Agora também temos um novo qualificador de classe e de métodos: **abstract**.

### **Classes abstratas X Interfaces:**

Você deve estar achando que classes abstratas e interfaces são conceitos parecidos e que podem ser usados com objetivos semelhantes. Cuidado! Uma classe pode estender uma única classe (que pode ser abstrata ou não), mas pode implementar várias interfaces. Além disso, interfaces não permitem declaração de atributos, enquanto que classes abstratas permitem. Interfaces estão mais ligadas a comportamento, enquanto que classes abstratas estão mais ligadas a implementação.

### **Exercícios:**

1-

Defina uma classe abstrata tipo numérico que deve servir como classe base para a montagem de classes como a classe fração ou a classe número complexo. Uma boa medida da qualidade de sua implementação é a quantidade de mudanças necessárias para por exemplo trocar a classe fração usada em um algoritmo de cálculo numérico pela classe número complexo. É bem verdade que existem operações que se aplicam a uma dessas classes, mas não a outra, mas essas disparidades deverão ser mantidas fora da classe base abstrata.

## **CLASSE ABSTRATA ITERADOR**

Neste exemplo iremos criar uma classe base abstrata iterador, que servirá como topo de uma hierarquia para iteradores de estruturas de dados como listas, vetores e árvores. O iterador de vetor é definido por herança da classe base abstrata de iteradores.

Perceba que alguns métodos da classe base são desprovidos de implementação, porém nada impede que você coloque como código desses métodos uma mensagem de erro do tipo “Erro, método deveria ter sido redefinido”, mas agora o compilador não pode mais te lembrar de redefini-los.

### **“SHOPPING LIST APPROACH” PARA A CLASSE ABSTRATA ITERADOR**

(As operações implementadas estão marcadas com ☒, existem outras operações úteis, não mencionadas por motivos de espaço. Esta classe base abstrata não tem a funcionalidade de uma classe que possa estar instanciada, perceba a ausência de um método para avançar na iteração).

- ☒ Método de inicialização da iteração
- ☒ Retorno do conteúdo da atual posição na iteração
- ☒ Atribuição de valor a atual posição da iteração
- ☒ Método que verifica se a iteração não chegou ao fim

### **“SHOPPING LIST APPROACH” PARA A CLASSE ABSTRATA ITERADOR VETOR**

(Uma tarefa desagradável quando iterando sobre vetores é manter os índices das iterações atualizados durante avanços e retrocessos. Note que retrocessos não fariam sentido em uma lista simplesmente ligada, por isso essa operação somente é definida neste nível da hierarquia.)

- ☒ Retorno do valor numérico ou índice da atual posição da iteração
- ☒ Retrocesso na iteração

- ☒ Avanço na iteração
- ☒ Avanço e retrocesso com saltos (inclusive é mais genérico que os dois anteriores).

## CÓDIGO

```
//Classe IteradorI
```

```
abstract class IteradorI {  
  
abstract public void começa();  
  
abstract public int retorna(); //metodos abstrato  
  
abstract public void atribui(int a);  
  
abstract public boolean fim();  
}
```

```
//Classe IteradorVetorI
```

```
class IteradorVetorI extends IteradorI {  
  
protected int[] vet; //itero sobre ele  
  
private int conta; //posicao atual da iteracao  
  
public IteradorVetorI(int[] itereme)  
{  
    vet=itereme;  
    conta=0;  
}  
  
public void começa()  
{  
    conta=0;  
}  
  
public void começa(int p)  
{  
    conta=(p%vet.length);  
}  
  
public void atribui(int novo)  
{  
    vet[conta]=novo;  
}  
  
public int retorna()  
{
```

```

        return vet[conta];
    }

    public boolean fim()
    {
        return conta==vet.length-1;
    }

    public int retorna_conta()
    {
        return conta;
    }

    public void avanca()
    {
        if (conta<(vet.length-1)) conta++;
    }

    public void retrocede()
    {
        if (conta>0) conta--;
    }
}

```

```

import java.io.DataInputStream;
//Classe principal, Arquivo Principal.java

class Principal {

    public static void main(String args[]) {

        int[] vet=new int[6];

        vet[0]=0; vet[1]=1; vet[2]=2; vet[3]=3; vet[4]=4; vet[5]=5;

        IteradorVetorI mit=new IteradorVetorI(vet);

        char o; //o=opcao,

        int e; //temporario

        String line; //linha a ser lida do teclado

        DataInputStream meuDataInputStream=new DataInputStream(System.in);

        try{
            do

```

```

{
    do { o=meuDataInputStream.readLine().charAt(0); }
    while (o!='\n');
    switch (o) {
        case 'a': //atribui
            line=meuDataInputStream.readLine();
            try {
                e=Integer.valueOf(line).intValue();
                mit.atribui(e);
            }
            catch (Exception erro) {
                System.out.println("Entrada invalida!");
            }
            break;
        case 'r': //retorna
            e=mit.retorna();
            System.out.println(e);
            break;
        case 'f': //frente
            mit.avanca();
            break;
        case 't': //tras
            mit.retrocede();
            break;
        case 'c': //comeca iteracao?
            mit.comeca();
            break;
        case 'e': //fim da iteracao?
            System.out.println(mit.fim());
            break;
        case 'v': //valor atual
            System.out.println("V:"+mit.retorna_conta());
            break;
        case 'm': //mostra vetor
            for(int j=0;j<vet.length;j++)
            { System.out.print "["+vet[j]+" " ); }
            System.out.println();
            break;
        default: ;
    } //switch

    } while (o!='q');
} //try block
catch (Exception erro) { /* nao faco nada */ }

} //main method

```



```
} //class Principal
```



```
m  
[0][1][2][3][4][5]  
a  
9  
m  
[9][1][2][3][4][5]  
f  
f  
f  
a  
33  
t  
a  
22  
m  
[9][1][22][33][4][5]  
c  
v  
V:0  
q
```

### Exercícios:

1-

Defina uma classe de nome **ArrayServices** que fornece serviços para vetores. Implemente os “serviços” de: ordenação de subvetor (vetor interno menor ou igual ao vetor em questão) , busca, troca de posições, etc. Esta classe opera sobre os vetores passados como argumentos de seus métodos (passagem implícita do ponteiro para o vetor). Os vetores devem ser de tipos numéricos definidos na linguagem (conte com a existência de operadores + - < ==, etc).

Você terá que definir uma versão desta classe para cada tipo da linguagem (**byte**, **float**, etc). Na verdade isto não é trabalhoso, basta você definir para um tipo, depois alterar só as partes necessárias e recompilar para os demais.

Pense como fazer a classe **ArrayServices** trabalhar em conjunto com a classe Iterador vetor. Não confunda estas duas classes, elas executam tarefas distintas.

## ACOPLAMENTO DINÂMICO DE MENSAGENS

Por acoplamento entenda a escolha certa de um método a ser executado para uma variável declarada como de uma classe, mas podendo conter um objeto de uma subclasse desta. Por dinâmico entenda em tempo de execução.

Já dissemos que um objeto de uma classe filha garante no mínimo o comportamento “behaviour” de seu pai. Por este motivo podemos atribuir um objeto da classe filha a uma variável da classe pai, mas não o contrário.

Acoplamento dinâmico mostrará que é possível fazer com que o compilador execute a

implementação desejada de um método redefinido para classes herdeiras, mesmo no caso de chamada de método ocorrer para uma variável de superclasse (classe pai) contendo um objeto de uma subclasse (classe filha). Isto nos permitirá construir listas heterogêneas .

Fazendo uma comparação com linguagens procedurais:

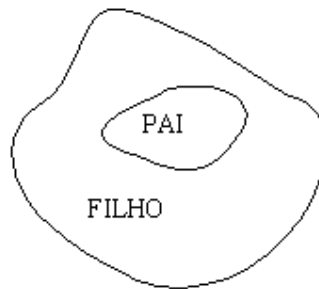
O trabalho do dynamic binding normalmente é feito desta forma em linguagens procedurais:

```
//ComputaContaBancaria
if type(a)==ContaCorrente then ComputaContaCorrente(a);
else if type(a)==Poupanca then ComputaPoupanca(a); //chamadas de procedimentos
```

Já em linguagens orientadas a objetos como Java temos o seguinte:

```
Conta ContVet=new conta[3];           //Vetor de objetos da classe conta
ContVet[0]=new Poupanca(1000,.10);     //toda Poupanca é uma Conta
ContVet[1]=new ContaCorrente(1000,.10,0); //toda ContaCorrente é uma Conta
ContVet[2]=new Poupanca(1030,.15);
ContVet[x].computa(); //nao importa para que classe da hierarquia de contas, computa
//computa definida e o compilador acopla dinamicamente, em tempo de execucao a
//mensagem
```

**Diagrama das classes:**



A classe filha garante no mínimo o mesmo comportamento, “behaviour” da classe pai, podendo acrescentar ou redefinir parte do que foi herdado. Por este motivo, uma variável da classe pai pode receber um objeto da classe filha, o comportamento da classe pai fica garantido e o restante (o que a classe filha acrescentou) é perdido. Já uma variável da classe filha não pode receber um objeto da classe pai, porque os métodos definidos para variáveis desta classe passam a não fazer sentido para o objeto contido nesta variável.

O diagrama reflete o aspecto das características acrescentadas pela classe filha a classe pai, mas não o fato de uma variável da classe pai poder receber um elemento da classe filha, isto porque como no desenho o pai é desenhado menor que o filho, o leitor tem a tendência de inferir que o pai cabe no filho o que é justamente o contrário do que acontece em termos de variáveis.

## UM EXEMPLO ESCLARECEDOR

O exemplo a seguir cria dois objetos (pai e filho) e faz atribuições e chamadas de métodos entre eles.



```
//Classe SuperClasse
```

```
class SuperClasse {  
  
    public void nome()  
    {  
        System.out.println("Metodo da superclasse");  
    }  
  
}
```

```
//Classe SubClasse
```

```
class SubClasse extends SuperClasse {  
  
    public int novoatributo;  
  
    public void nome()    //redefinicao  
    {  
        System.out.println("Metodo da subclasse");  
    }  
  
    public void novometodo()  
    {  
        System.out.println("Novo metodo:"+novoatributo);  
    }  
  
}
```

```
//Classe principal, Arquivo Principal.java
```

```
class Principal {  
  
    public static void main(String args[]) {  
  
        SubClasse esub1=new SubClasse();  
        SuperClasse esuper1=new SuperClasse(); //poderia ter alocado uma SubClasse aqui  
        esub1.nome();  
        esub1.novoatributo=10;  
        esuper1.nome();  
        esuper1=esub1;  
        esuper1.nome();  
    } //main method  
  
} //class Principal
```



Metodo da subclasse  
Metodo da superclasse  
Metodo da subclasse

### //COMENTARIOS:

Note que o método é escolhida de acordo com o conteúdo da variável e não de acordo com a classe desta.

## O QUE ACONTECE COM O QUE FOI ACRESCENTADO

Depois do programa anterior, você deve estar se perguntando o que acontece com o que foi acrescentado pela classe filha quando um objeto desta classe é atribuído a classe pai.

### CÓDIGO

```
//Insira aqui o arquivo da SuperClasse do exemplo anterior
```

```
//Insira aqui o arquivo da SubClasse do exemplo anterior
```

```
//Classe principal, Arquivo Principal.java
```

```
class Principal {
```

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

```
        SubClasse esub1=new SubClasse();
```

```
        SuperClasse esuper1=new SuperClasse(); //poderia ter alocado uma SubClasse aqui
```

```
        esub1.nome();
```

```
        esub1.novoatributo=10;
```

```
        esuper1.nome();
```

```
        esuper1=esub1;
```

```
        esuper1.nome();
```

```
        esub1=(SubClasse)esuper1; //cast
```

```
        esub1.nome();
```

```
        esub1.novometodo();
```

```
    } //main method
```

```
} //class Principal
```



Metodo da subclasse  
Metodo da superclasse  
Metodo da subclasse  
Metodo da subclasse  
Novo metodo:10

## //COMENTARIOS

Embora você seja capaz de recuperar os métodos e atributos acrescentados através do type casting, enquanto isto não for feito, estes métodos e atributos estão inacessíveis.

## Exercícios:

1-

Implemente em suas classes métodos que imprimem uma frase identificando o tipo da classe, por exemplo: “Eu sou a classe conta corrente, especialização de conta bancaria.”

2-

Modele e implemente uma hierarquia de CONTAS BANCÁRIAS, use os recursos que achar conveniente: classes abstratas, interfaces, variáveis static. Você deve definir classes semelhantes as contas bancárias como poupança, conta corrente, etc. Os métodos devem ter nomes como deposita, saca, computa, etc.

## LISTA HETEROGÊNEA DE FORMAS (geométricas)

Este exemplo lida com um vetor de objetos da classe forma definida em CLASSES ABSTRATAS E CONCRETAS , estes objetos são retângulos, pontos, etc. O objetivo é mostrar: subclasses contidas em variáveis (posições de vetor) da superclasse e acoplamento dinâmico de mensagens.

O nosso vetor de formas conterà objetos gráficos de classes heterogêneas. Trataremos todos de maneira uniforme, chamando os métodos **mostra** e **move**. Você já viu nos exemplos anteriores como recuperar o objeto em uma variável de sua própria classe e não superclasse.



**CÓDIGO** Recompile com esse novo método main:

```
//Insira aqui a definicao da classe Forma dada em CLASSES ABSTRATAS E CONCRETAS
```

```
//Insira aqui a definicao da classe Ponto dada em CLASSES ABSTRATAS E CONCRETAS
```

```
//Insira aqui a definicao da classe Retangulo dada em CLASSES ABSTRATAS E CONCRETAS
```

```
//Classe principal, Arquivo Principal.java

class Principal {

    public static void main(String args[]) {
        Forma vetorgrafico[]=new Forma[4];
        vetorgrafico[0]=new Retangulo(0,0,20,10);
        vetorgrafico[1]=new Ponto(0,1);
        vetorgrafico[2]=new Retangulo(100,100,20,20); //lados iguais
        vetorgrafico[3]=new Ponto(2,1);
        for(int i=0;i<4;i++)
        {
            vetorgrafico[i].mostra();
        }

        for(int j=0;j<4;j++)
        {
            vetorgrafico[j].move(12.0f,12.0f);
            vetorgrafico[j].mostra();
        }
    }
}
```



```
(0,0)(20,10)
(0,1)
(100,100)(20,20)
(2,1)
(12,12)(20,10)
(12,13)
(112,112)(20,20)
(14,13)
```

### //COMENTARIOS:

Observe que a implementação correta do método **mostra** é escolhida em tempo de execução de acordo com a classe do objeto que está naquela posição do vetor.

### Exercícios:

1

Considere as seguintes declarações em Java:

```
public class T
{
    public void f(void) { System.out.println( "Estou em T");}
};
```

```
public class S extends T
{
    public void f(void) { System.out.println("Estou em S");}
};

T x= new T();
S y=new S();
T p=new T();
```

e as seguintes invocações de operações:

```
p.f(); //primeira
p=y;
p.f(); //segunda
x.f(); //terceira
y.f(); //quarta
x=y;
x.f() //quinta
```

Responda qual é o resultado na tela de cada uma destas chamadas.

## CONCEITOS AVANÇADOS

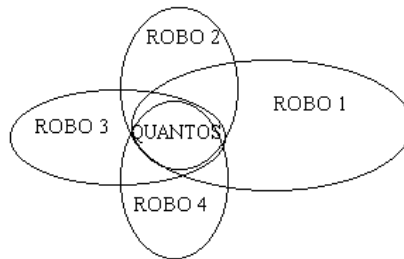
Neste tópico apresentaremos conceitos avançados da linguagem, estes conceitos são importantes se você deseja criar softwares relativamente grandes em **Java**.

### ATRIBUTOS STATIC

Até o momento só havíamos aprendido como definir atributos de instância. Cada objeto tinha seus próprios atributos e uma modificação nos atributos de um objeto não afetava os atributos de outros objetos.

Neste tópico iremos apreender como definir atributos de classe. Esses atributos são os mesmos para todos os objetos, eles são compartilhados. Uma mudança em um destes atributos é visível por todos os objetos instanciados. Atributos de classe também são chamados de atributos **static**.

Neste exemplo definiremos uma classe **robo** que usa atributos **static** para saber quantos **robos** foram criados (instanciados). Um outro uso de atributos **static** seria usar uma estrutura de dados que permitisse que um **robo** saber a posição(coordenadas) dos demais objetos de sua classe.



### CÓDIGO

```
//Classe Robo

class Robo {

public      int x;
public      int y;
public static int quantos; //quantos foram instanciados

public Robo(int ax,int ay)
{
    x=ax; y=ay;
    quantos++;
}

}
```

```
//Classe principal, Arquivo Principal.java
```



```

class Principal {

    public static void main(String args[]) {
        Robo.quantos=0; //inicializando a variavel static
        Robo cnc,cnc2;
        System.out.println(Robo.quantos);
        cnc=new Robo(10,12);
        System.out.println(Robo.quantos);
        cnc2=new Robo(11,12);
        System.out.println(Robo.quantos);
    } //main method

} //class Principal

```



0  
1  
2

### //COMENTARIOS:

Quando definimos atributo **static**, estávamos nos referindo ao sentido geral de atributo. Apesar de termos exemplificado com um inteiro, você poderia ter usado uma classe no lugar desse atributo, tomando o cuidado de chamar **new** antes de usá-lo.

## MÉTODOS STATIC

Métodos **static** também são chamados de métodos de classes. Estes métodos só podem operar sobre atributos de classes. Nos já vínhamos usando estes métodos. Existem exemplos de chamadas de métodos **static** em nossos programas anteriores porque as packages da linguagem são repletas de exemplos de métodos **static**. Por este motivo eu escolhi uma das definições de métodos **static** da Java API (Sun Microsystems®) para transcrever:

*abs(double). Static method in class java.lang.Math  
Returns the absolute double value of a.*

Neste tópico não mostraremos um exemplo propriamente dito, apenas mostraremos como definir um método **static**, portanto é importante que você faça os exercícios.

Métodos **static** são definidos assim como atributos **static**:

```

public static int MDC(int a,int b) { //maximo divisor comum de a e b
}

```

No exemplo da classe **Fracao** (TAD FRAÇÃO), tivemos que lidar com o método **mdc**. Este método não fazia muito sentido para o usuário desta classe, embora fosse necessário para fazer certas operações que envolviam simplificação de fração. Naquele momento nossa opção foi por

implementar o método como **private** na própria classe **Fracao**. Alguém rigoroso poderia ter dito: “Você está errado, **mdc** não tem nada a ver com **Fracao** e eu vou implementá-lo em uma classe separada chamada **ServicosMatematicos**”. A minha resposta a esta afirmação seria: “Acho sua idéia boa, mas sem o uso de métodos **static** na classe **ServicosMatematicos**, você vai ter que declarar um objeto da classe **ServicosMatematicos** dentro de cada **Fracao** para poder usar o método **mdc**.”

### Exercícios:

1-

Defina uma classe com métodos **static** que permite cálculos matemáticos como MDC, FATORIAL, COMBINAÇÕES(n,k), PERMUTAÇÕES, etc.

\*2-

Existe uma linguagem de programação chamada Logo que é muito usada no Brasil em escolas de primeiro e segundo grau. Nesta linguagem você pode programar os movimentos na tela de uma tartaruguinha ou cursor. Este cursor descreve movimentos riscando ou não a tela, dentre os movimentos descritos estão círculos, quadrados e sequências repetitivas (loops). Modifique seu objeto robo para apresentar alguns desses recursos. Crie então um programa que lê de um arquivo texto instruções para este objeto robo executar movimentos na tela.

Logo foi criada por um cientista de computação do MIT.

3-

Você pode desejar armazenar as informações da classe **Robot** em outra classe, que computa outros cálculos, neste caso temos pelo menos duas alternativas a seguir:

a) Crie uma classe auxiliar externa de armazenagem e para todos objetos **robo** instanciados passe o ponteiro desta classe auxiliar como argumento do construtor. Assim esses objetos poderão mandar mensagens para esta classe de armazenagem. Essas mensagens, chamadas de métodos, podem ter vários significados, num sentido figurado podemos ter algo parecido com: “Classe auxiliar, armazene essa informação para mim”. “Classe auxiliar, me mande uma mensagem daqui a cinco segundos, estou passando o ponteiro para mim mesmo (**this**)”, etc. As mensagens vistas desse modo ficam mais interessantes, você pode até achar engraçado, mas é muito prático pensar assim.

4-

Em alguma classe que você criou anteriormente defina variáveis **static** com o seguinte objetivo: Contar, fazer estatísticas das chamadas de métodos da classe.

5-

Use o que foi aprendido sobre “**static variables**” no programa contas, o objetivo é armazenar informações sobre os movimentos de todas as contas num objeto **static**. Para que a modificação fique a contento você pode precisar tornar o exemplo mais próximo da realidade, adicionando no construtor de contas um argumento: número de conta. Comente se você usaria alocação dinâmica para essa classe de armazenagem agregada em conta, ou não. Tenha em mente a questão do tamanho em bytes do objeto.

## TRATAMENTO DE EXCEÇÕES

Se os conceitos de orientação a objetos dados até agora fossem suficientes para modelar qualquer atividade ou objeto do mundo real, todos os problemas de programação estariam resolvidos.

Ocorre que o mundo real é bem mais complexo que seu programa pode ser. Nem todos os objetos e iterações entre eles podem ser modelados ou previstos. Os mecanismos de tratamento de exceções se encaixam justamente nessa lacuna.

O modelo de tratamento de exceções adotado por **Java** é muito semelhante ao de C++. Se você já teve alguma experiência com tratamento de exceções, este tópico será bastante fácil.

## TRATANDO AS EXCEÇÕES GERADAS POR TERCEIROS

Tratamento de exceções permite lidar com as condições anormais de funcionamento de seu programa. Fazer uso deste recurso tornará seu software mais robusto, seguro e bem estruturado. São exemplos de condições anormais: acesso a um índice inválido de um vetor, tentativa de uso de um objeto não inicializado, falha na transferência de uma informação, uma falha não prevista, etc.

Sem as facilidades oferecidas pela linguagem seria muito difícil lidar com essas condições anormais, isto pode ser observado nos deficientes meios de tratamento de erros usados em linguagens que não fornecem “exception handling”.

**Java** é uma linguagem que faz forte uso do conceito de tratamento de exceções. Em algumas linguagens que implementam tratamento é perfeitamente possível programar sem usar esse recurso, mas em **Java** não. Isto pode ser comprovado pela necessidade de alguns blocos de código `try {} catch {}` em programas anteriores.

Um dos motivos de o programador **Java** ter que saber tratamento de exceções é que os métodos de classes definidas na linguagem podem gerar exceções e na maioria das vezes o compilador nos obriga a escrever tratadores (blocos `try{} catch{}`) para chamadas destes métodos.

Quando você for estruturar seu código dessa forma, haverá duas ações básicas que devem ser tomadas: levantar (jogar) uma exceção e tratar uma exceção. Uma exceção será levantada quando for verificada uma condição anormal de funcionamento do programa, então o método que esta sendo executado é imediatamente terminado e o controle passa para o método que o chamou, onde pode ocorrer um tratador da exceção ou não. Se ocorrer um tratador, na maioria dos casos a exceção para de se propagar ali mesmo. Se não ocorrer um tratador outras chamadas de métodos são “desfeitas”, encerradas, podendo culminar no término do programa se toda a cadeia de chamada de métodos for desfeita até chegar em `main` sem que se ache um tratador para esta exceção. Existem vários modelos de tratamento de exceções, o modelo adotado por **Java** recebe o nome de: “termination model”, justamente por essas terminações de métodos.

Mas o que é uma jogar uma exceção? É suspender a execução do método atual e passar um objeto para o bloco `catch` mais próximo na cadeia de chamadas de métodos atual. Isto é feito através da declaração:

```
throw nomedoobjeto; //ou throw new nomedaclasseodoobjeto(argumentos do construtor)
```

`throw` é como um `break` para métodos.


Como exceções são objetos, você pode definir hierarquias de classes de exceções, que mapeiem em termos de informações as condições anormais de seu programa contendo as mensagens de erro e as possíveis soluções

As exceções geradas pela linguagem pertencem a uma hierarquia cujo topo é a classe `Throwable`, imediatamente estendida por `Error` e `Exception`. Neste exemplo você verá que os tratadores de exceção são escolhidos comparando a classe da exceção jogada e a classe de exceções que o tratador diz tratar. Assim sendo o tratador:

```
try{ /*algo que possa gerar uma excecao*/}  
catch (Exception erro) { /* acoes de tratamento do erro com possivelmente nova tentativa de execucao dos metodos chamados*/ }
```

Seria capaz de tratar todas as exceções que estejam abaixo de **Exception** (na hierarquia) geradas em `try { }`. Dispondo mais de um tratador (bloco `catch`) em sequência onde os primeiros só tratam as classes exceções mais baixas da hierarquia, é possível escolher que código de tratamento usar com cada tipo de exceção gerada.

Por sorte, das exceções levantadas pela linguagem, você como programador só precisará tratar as da hierarquia de **Exception**. Neste exemplo iremos forçar o acontecimento da exceção **ArrayIndexOutOfBoundsException**, através de uma tentativa de acesso a um índice inválido de um vetor:

 **CÓDIGO** Acesso a índice inválido do vetor sem corromper o sistema, programadores C pasmem!

//Classe principal, Arquivo Principal.java

```
class Principal {  
  
    public static void main(String args[]) {  
  
        int a[]=new int[4];  
  
        try {  
            a[4]=10;  
            //linha acima gera excecao, os indices validos sao quatro:0,1,2,3  
            //qualquer codigo escrito aqui (depois de a[4]=10;)   
            //nunca sera executado  
        }  
        catch(Exception ae) {  
            //refaz a pergunta do indice a alterar ao usuario  
            //e descobre que ele queria alterar o valor no indice 3, escrevendo 12  
            a[3]=12;  
        }  
  
        System.out.println(a[3]);  
    }  
}
```



12

#### //COMENTARIOS:

O fato do código imediatamente após o ponto onde foi gerada a exceção não ser executado te preocupa? Você gostaria por exemplo de ter uma chance de liberar recursos do sistema (ex...fechar um arquivo) antes do método ser terminado? É para isso que existe em **Java** o bloco `try{ } catch{ } finally{ }` que não existe em **C++**, mas existe por exemplo em **Modula-3**.

A cláusula `finally{ }` é opcional, seu código vai ser executado ocorra ou não ocorra uma exceção no bloco `try{ }`.

Exemplo clássico de uso do bloco `try{ } catch{ } finally { }`:

```

try {
    //abre um arquivo
    //gera uma excecao com arquivos
}
catch (ExcecaoArquivo e){
    //tenta recuperar arquivo e informacoes perdidas
}
finally {
    arquivo.close();
}

```

finally tem sido usado para fechar arquivos, parar threads e descartar janelas.

## Exercícios:

1-

Em vários dos programas anteriores mencionamos que haveria uma maneira melhor de tratar situações anormais. Um desses programas era o do tipo abstrato de dados **matriz**. Leia este programa e adicione tratamento de exceções para as condições anormais que podem surgir, tais como acesso a índices inválidos. Complemente este exercício após ter lido os tópicos seguintes.

## GERANDO SUAS PRÓPRIAS EXCEÇÕES

O exemplo a seguir ensina como trabalhar com **throw**, a palavra chave usada para levantar exceções. Este exemplo se baseia na classe **Fracao** de TAD FRAÇÃO. Nossa exceção será gerada quando nas operações de frações ocorrer uma divisão por zero.

Em **Java**, exceções são instâncias de classes que pertencem a hierarquia que é iniciada, encabeçada, pela classe **Throwable**. Neste exemplo construiremos nossa exceção herdando de **Exception** que por sua vez herda de **Throwable**.

## CÓDIGO

```

public class DivisaoPorZero extends Exception
{
    public String info;
    public DivisaoPorZero(String i)
    {
        info=i;
    }
}

```

Você deve modificar a classe **Fracao** apresentada anteriormente na página 66, para aceitar o seguinte método:

```

public Fracao divisao(Fracao j) throws DivisaoPorZero
{
    Fracao g;
    if (j.den==0) throw new DivisaoPorZero("Na classe Fracao");
    //se for zero a execucao nao chega aqui
    g=new Fracao(num*j.den,den*j.num);
}

```

```
return g;  
}
```

```
class Principal {  
    public static void main(String args[])  
    {  
        Fracao a,b,c;  
        a=new Fracao(5,3);  
        b=new Fracao(2,0);  
        System.out.print("Esta e' a fracao a: ");  
        a.mostra();  
        System.out.print("Esta e' a fracao b: ");  
        b.mostra();  
        try {  
            c=a.divisao(b);  
            c.mostra();  
        }  
        catch(DivisaoPorZero minhaexcecao)  
        {  
            System.out.println("Nao posso dividir por zero");  
        }  
    }  
}
```



Esta e' a fracao a: (5/3)  
Esta e' a fracao b: (1/0)  
Nao posso dividir por zero

## //COMENTARIOS

Nós não apresentamos um exemplo de uma exceção propagando em uma cadeia longa de chamadas de métodos. Mas com os conhecimentos dados, você pode fazer isso.

Outro fato importante é que um bloco **catch** também pode gerar exceções, assim se você pegou uma exceção e resolveu que não consegue tratá-la você pode fazer um **throw** dela mesma ou mudar a classe da exceção e continuar propagando (**throw** de outra exceção), ou fazer o que você pode para reparar o erro e jogar uma exceção para que o que você fez seja completado por outros métodos. Lembre-se que se você pegou uma exceção, ela para de propagar.

## Exercícios:

1-

Implemente, tratamento de exceções completo para o exemplo de TAD FRAÇÃO. Antes faça um levantamento das exceções que podem ser geradas, lembre das restrições matemáticas para o denominador em uma divisão. Leve em conta também o overflow de variáveis **int** que são uma

representação com número de bits finito da sequência dos números inteiros (conjunto  $\mathbb{Z}$  da matemática). Compare este tratamento com o de outros programas por exemplo na divisão por zero, quais as vantagens que você pode apontar e as desvantagens?

## THREADS

threads são fluxos de execução que rodam dentro de um processo (aplicação). Normalmente os threads compartilham regiões de memória, mas não necessariamente. Lembre-se de encapsulamento. Processos, os avós dos threads permitem que o seu sistema operacional execute mais de uma aplicação ao mesmo tempo enquanto que threads permitem que sua aplicação execute mais de um método ao mesmo tempo.

Todos os programas que fizemos até agora só tinham um único caminho, fio, fluxo, de execução. Nenhum deles executava duas coisas (dois pedaços de código) simultaneamente. Grande parte do software de qualidade escrito hoje faz uso de mais de uma linha de execução, mais de um thread. São os chamados programas multithreaded.

O seu browser de hipertexto consegue fazer o download de vários arquivos ao mesmo tempo, gerenciando as diferentes velocidades de cada servidor e ainda assim permite que você continue interagindo, mudando de página no hipertexto enquanto o arquivo nem foi carregado totalmente? Isto não seria possível sem o uso de threads.

O seu editor de textos permite que você vá editando o começo do arquivo, enquanto ele está sendo carregado do disco? Editar e carregar do disco são atividades que não podem ser intercaladas de maneira simples em um pedaço de código. Seu editor está usando threads, essas atividades estão sendo feitas em paralelo.

Se sua máquina só possui um processador, esse paralelismo é um falso paralelismo. O processador tem seu tempo dividido em pequenos intervalos, em cada intervalo ele executa uma das atividades e você tem a sensação de que tudo está funcionando ao mesmo tempo, simultaneamente. Se você é um felizarado e sua máquina têm mais de um processador, então seu ambiente será capaz de mapear seus threads em hardware e você terá realmente processamento paralelo.

Se você olhar a tradução de threads no dicionário é até capaz que você encontre um desenho de um carretel de linha ou da rosca de um parafuso, este nome é bastante feliz. Imagine que seu programa é composto por várias linhas de execução que funcionam em paralelo (algumas vezes estas linhas podem se juntar, outras se dividir). Cada linha de execução cuida de uma tarefa: transferir um arquivo, tratar a entrada do usuário, mostrar sua janela na tela, etc.

**threads** é uma invenção recente se comparada com o restante da linguagem. Algumas outras linguagens (bem poucas) fornecem facilidades para lidar com threads, exemplo: Modula-3. Também conhecidos como lightweight processes, threads são um recurso extremamente difícil de se implementar, de modo que é possível dizer que ou seu ambiente de programação oferece facilidades para lidar com eles, ou você não vai querer implementá-los/usá-los.

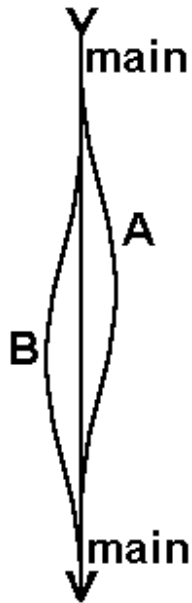
## CRIANDO THREADS USANDO INTERFACES OU HERANÇA

Existem duas maneiras básicas de criar threads em **Java**, usando interfaces e usando herança. Usando herança, sua classe já é um thread (is a relationship), que quando tiver seu método **start()** chamado vai executar tudo o que estiver no método **run()** em paralelo.

Usando interfaces, você define uma classe cujo método **run()** vai ser executado por um thread.

## HERANÇA USADA PARA CRIAR THREADS

### Threads ↗



Se você quer ter uma idéia do que vai acontecer no programinha abaixo, veja o desenho ao lado. Nós faremos uma corrida de Threads: A e B. Eles serão iniciados depois que o programa principal (main) começar. A sai com alguma vantagem pois é iniciado primeiro, depois sai B. A cada loop estes Threads são obrigados a fazer uma pausa por um intervalo aleatório até completarem 4 loops: 0,1,2,3,4.

No final, pedimos ao programa principal que espere os Threads terminarem seus ciclos para se juntar a eles (método `join()` da classe `thread`). O método `join` não retorna enquanto o seu threads não terminar.

Existem uma série de outras primitivas para lidar com Threads: pausa, parada, retorno a execução, etc. Não explicaremos todas aqui. Você deve fazer leituras complementares, é isso que temos aconselhado. Agora estas leituras podem ser mais técnicas, tipo guias de referência, uma vez que nós já fizemos a introdução do assunto.

Alguns assuntos, a exemplo de Threads exigem um conhecimento teórico forte, de modo que também aconselhamos que você adquira um livro sobre programação concorrente.

### CÓDIGO

```
public class MeuThread extends Thread {  
  
    public MeuThread(String nome)  
    {  
        super(nome);  
    }  
  
    public void run() // o metodo que vai ser executado no thread tem sempre nome run  
    {  
        for (int i=0; i<5; i++) {  
            System.out.println(getName()+ " na etapa:"+i);  
            try {  
                sleep((int)(Math.random() * 2000)); //milisegundos  
            } catch (InterruptedException e) {}  
        }  
        System.out.println("Corrida de threads terminada:" + getName());  
    }  
}
```

```
class CorridaThreads  
{
```



```

public static void main (String args[])
{
    MeuThread a,b;
    a=new MeuThread("Leonardo Xavier Rossi");
    a.start();
    b=new MeuThread("Andre Augusto Cesta");
    b.start();
    try {a.join(); } catch (InterruptedException ignorada) {}
    try {b.join(); } catch (InterruptedException ignorada) {}
}
}

```



thread A  
thread B

**Leonardo Xavier Rossi na etapa:0**

Andre Augusto Cesta na etapa:0

Andre Augusto Cesta na etapa:1

**Leonardo Xavier Rossi na etapa:1**

Andre Augusto Cesta na etapa:2

**Leonardo Xavier Rossi na etapa:2**

**Leonardo Xavier Rossi na etapa:3**

Andre Augusto Cesta na etapa:3

Andre Augusto Cesta na etapa:4

Corrida de threads terminada:Andre Augusto Cesta

**Leonardo Xavier Rossi na etapa:4**

**Corrida de threads terminada:Leonardo Xavier Rossi**

## INTERFACES USADAS PARA CRIAR THREADS

Colocar sua classe abaixo da classe Threads em uma hierarquia, as vezes é um preço muito grande para ter um método dela rodando em um Thread. É por isso que Threads pode ser criados usando interfaces. Você só tem que dizer que implementa a interface **Runnable**, que é composta do método **run()**. Quando um Thread é construído usando como argumento uma classe que implementa a interface **Runnable**, chamar o método **start** para este thread faz com que o método **run()** de nossa classe comece a ser executado neste thread paralelo.

### CÓDIGO

```

public class RodemeEmUmThread implements Runnable {

    public String str; //nome do objeto que vai ter o metodo run rodado em um
    //thread

    public RodemeEmUmThread(String nome)
    {
        str=nome;
    }

    public void run()

```

```

    {
        for (int i=0; i<5; i++) {
            System.out.println(str+ " na etapa:"+i);
        }
        //As mensagens de corrida terminada poderiam se postas aqui.
    }
}

```

```

class ThreadsRodadores
{
    public static void main (String args[])
    {
        Thread a,b;
        RodemeEmUmThread leo,andre;
        leo=new RodemeEmUmThread("Leonardo Xavier Rossi");
        andre=new RodemeEmUmThread("Andre Augusto Cesta");
        a=new Thread(leo);
        a.start();
        b=new Thread(andre);
        b.start();
        try { a.join(); } catch (InterruptedException ignorada) { }
        //espera thread terminar seu metodo run
        try { b.join(); } catch (InterruptedException ignorada) { }
    }
}

```



thread A  
**thread B**

Leonardo Xavier Rossi na etapa:0  
**Andre Augusto Cesta na etapa:0**  
**Andre Augusto Cesta na etapa:1**  
 Leonardo Xavier Rossi na etapa:1  
**Andre Augusto Cesta na etapa:2**  
 Leonardo Xavier Rossi na etapa:2  
 Leonardo Xavier Rossi na etapa:3  
**Andre Augusto Cesta na etapa:3**  
 Leonardo Xavier Rossi na etapa:4  
**Andre Augusto Cesta na etapa:4**

## Exercícios:

1-

As tarefas de transpor uma matriz ou fazer o “espelho” de uma imagem (que pode ser representada por uma matriz) são exemplos fáceis de tarefas que podem ser divididas em dois ou mais Threads. É lógico que aqui estaremos buscando tirar vantagem da possível existência de mais de um processador e também buscando liberar o Thread principal dessa computação, para que ele possa fazer outras atividades antes do join().

Escolha uma dessas tarefas e implemente-as usando Threads. Dica: os Threads devem ser construídos de modo a conter a referência para a matriz que vai armazenar o resultado, a referência para a matriz original e os valores que indicam em que área da matriz este Thread deve trabalhar. Por exemplo: no caso da inversão da imagem um Thread trabalharia em uma metade e o outro na outra metade.

Seria interessante imprimir na tela os instantâneos da matriz resultado para você ver o trabalho sendo feito em paralelo.

## PENSANDO MULTITHREADED

Este tópico discute alguns dos problemas que podem surgir quando lidando com threads e apresenta algumas das soluções da linguagem. Não nos aprofundaremos muito em threads.

Existe um exemplo clássico dos problemas que podem acontecer quando você está usando concorrência. Imagine que você tem um programa que lê dados em bytes de algum lugar (teclado/disco) e os transmite via rede. Você decidiu usar threads porque não quer ficar com um programa de um único fluxo de execução bloqueado porque está esperando o teclado ou o disco enquanto poderia estar tentando transmitir parte de seu buffer pela rede.

Para tal você dividiu o seu programa em dois threads que ficam repetindo o mesmo conjunto de ações:

Thread A, Enfileirando valores do teclado (Leitura):

- 1-Lê valor do fonte.
- 2-Consulta o número de elementos da fila.
- 3-Soma um a esse número.
- 4-Enfilera o valor lido.

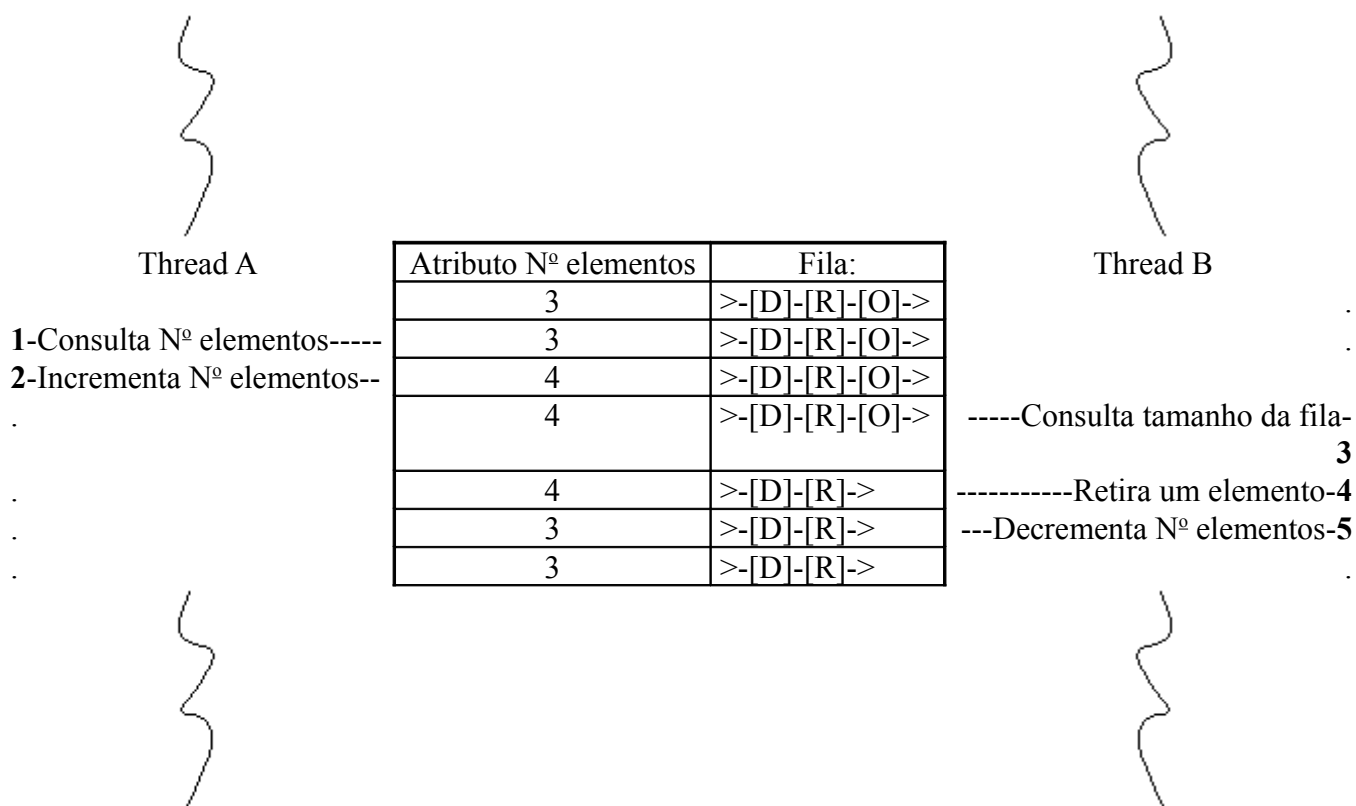
Thread B, Desenfileirando os valores (E escrevendo-os na saída):

- 1-Consulta o tamanho da fila se for maior que zero executa próximo passo
- 2-Retira elemento da fila.
- 3-Decrementa contador de elementos.

Os seus threads podem não rodar em intervalos definidos, regulares de tempo. Alguns problemas podem surgir pois a mesma região de memória é acessada por dois fluxos de execução distintos que podem ter executados apenas alguns de seus “passos”:

```
public class ThreadA {  
    //Leitura  
    while (!fimLeitura)  
    {  
        EnfileraUm();  
    }  
}
```

```
public class ThreadB {  
    //Escrita  
    while (!fimEscrita)  
    {  
        Escreve(DesenfileraUm());  
    }  
}
```



A sequência de operações **1,2,3,4,5** corrompe o estado de sua estrutura de dados. Se um só thread estivesse executando sobre a mesma estrutura, ainda assim poderiam também haver momentos em que o atributo número de elementos não reflete exatamente quantos estão na fila, mas isso logo seria consertado. Só que com dois threads, um outro thread pode usar essa informação inconsistente sem que haja tempo para ela ser reparada. Seu programa pode falhar quando você tentar retirar um elemento de uma fila vazia.

Alguns dos passos dos programas acima podem ser divididos em passos menores, mas o que importa nesse exemplo é que sem o uso dos recursos específicos da linguagem é impossível tornar este programa seguro. O ideal para este nosso exemplo seria que um **Thread** tivesse que esperar um ciclo do outro **Thread** terminar para iniciar o seu próximo ciclo e vice versa, ou seja os ciclos teriam que ser sincronizados, ocorrer um após o outro, podendo intercalar ciclos do **Thread A** com ciclos do **Thread B**, mas não pedaços desses ciclos. Em casos assim , costuma dizer que cada ciclo compõe um conjunto de ações indivisível.

Agora que você já sabe a solução, basta saber como implementá-la. Os métodos **EnfilaUm();** e **DesenfilaUm();** devem ser especificados como sincronizados. Evite especificar métodos muito grandes como sincronizados, pois outros **Threads** terão que esperar muito tempo para começar a executar.

```
public synchronized void EnfilaUm(byte a) {
    //esta parte voce ja sabe
}
```

```
public synchronized Byte DesenfilaUm() {
    //Byte e uma classe wrapper, ela e igual a null se nao ha elementos
    //o restante voce ja sabe
```

```
}
```

Agora sua classe é segura. Você só deve se preocupar com a sincronização dos métodos do seu programa. Os métodos e as classes da linguagem já são escritos para serem “Thread safe”, o que gerou muito trabalho para os programadores do “Java team”.

## **//COMENTARIOS**

Acabamos agora o tópico sobre threads e também o tutorial. Antes de você começar a estudar outros textos, um comentário final: Se você quiser rodar um método de uma classe já pronta em um thread, lembre-se de colocá-lo (a sua chamada) dentro de um método com o nome `run()`.

## **O QUE VOCÊ PODE ESTUDAR A PARTIR DE AGORA USANDO JAVA**

As opções são muitas: programação concorrente, interfaces gráficas, sistemas distribuídos, etc. O importante é que para todos estes assuntos, você vai ter que saber as técnicas ensinadas neste texto.



## **Índice remissivo:**





## **Conteúdo:**



## **Bibliografia:**

Alguns dos tutoriais aqui mencionados se tornarão livros, de modo que é importante que você faça uma busca pelos nomes dos autores também.

- [1]Rumbaugh, Blaha M., Premerlani W., Eddy F. & Lorensen W. ,“Object-Oriented Modeling and Design”. Prentice Hall, 1991.
- [2]Kernigham Brian W. , Ritchie Dennis M. , “The C Programming Language” , Englewood Cliffs, N.J.:Prentice-Hall Inc, 1978
- [3]Rubira, C.M.F. “Structuring Fault-Tolerant Object Oriented Systems Using Inheritance and Delegation”, Ph.D. Thesis, Department of Computing Science, University of Newcastle upon Tyne, October 1994, see Chapter 2.
- [4]Lemay Laura, Perkins Charles L., “Teach Yourself JAVA in 21 days”, samsnet, 1996
- [5]van Hoff A., Shaio S., Starbuck O., Sun Microsystems Inc, “Hooked on Java”, Addison-Wesley, 1996
- [6]Harold Elliotte Rusty, “Brewing Java: A Tutorial”, <http://sunsite.unc.edu/javafaq/javatutorial.html>
- [7]Campione Mary, Walrath Kathy, “The Java Tutorial!, Object-Oriented Programming for the Internet”, <http://www.aw.com/cp/javaseries.html>