

INSTITUTO DE SOFTWARE DO CEARÁ – INSOFTEC

XI Semana da Tecnologia da Informação: na onda das férias

Java

básico e intermediário

XI SEMANA DA TECNOLOGIA DA INFORMAÇÃO

Java básico e intermediário



60.150-160 • Av Santos Dumont, 1180 • Fortaleza - CE
Tel.: (85) 488.5200 • Fax: (85) 488.5210

Instrutor:

Felipe Gaúcho

gaucho@atlantico.com.br

• todos os direitos reservados •

Índice analítico

<i>A tecnologia Java</i>	4	<i>Operadores de deslocamento (>>, <<, >>>)</i>	24
O que é Java ?	4	Circuitos lógicos	25
O que é máquina virtual ?	5	Exercícios	26
O que é garbage collection ?	6	<i>Fluxo de controle</i>	27
A estrutura de um código em Java.	6	Ramificação if, else	27
Exercícios	7	Ramificação switch	28
<i>Implementando o primeiro programa</i>	8	Repetição for	29
O ambiente de desenvolvimento	8	Repetição while	30
O código fonte	9	Comandos especiais de controle de fluxo: break, continue e label	31
Compilando e executando o programa	10	Lendo valores do teclado	32
Erros mais comuns	11	Exercícios	33
Exercícios	12	<i>Agrupamento de dados (Arrays)</i>	34
<i>Sintaxe da linguagem Java</i>	14	Declarando e criando Arrays	34
Delimitadores em Java	14	Inicializando os valores de um Array	35
Comentários	14	Array multi-dimensional	36
Identificadores e palavras reservadas	15	Array esparsos	36
Declarando variáveis em Java	16	Exercícios	37
Tipos primitivos da linguagem Java	17	<i>Introdução à Orientação a Objetos</i>	38
Tipos lógicos: <i>boolean</i>	17	Motivação ao uso de um novo paradigma, orientado a objetos.	38
Tipos textuais: <i>char</i> e <i>String</i>	17	Tipos agregados de dados	39
Tipos numéricos inteiros: <i>byte</i> , <i>short</i> , <i>int</i> e <i>long</i>	18	Criando objetos	41
Tipos numéricos de ponto flutuante: <i>float</i> e <i>double</i>	19	Alocação de memória durante a criação de objetos	42
Convenções de codificação	19	Atribuição de referências a uma variável	42
Exercícios	19	Termos básicos em Orientação a Objetos	43
<i>Expressões</i>	21		
Operadores lógicos e aritméticos	21		
Concatenação de <i>Strings</i> com o operador +	22		
Promoção e Casting	23		

<i>Abstração de dados</i>	44	<i>O que são exceções ?</i>	73
Tipos abstratos de dados	44	Diferença entre exceções e erros	73
Definição de métodos	45	Tratamento de exceções	74
A referência this	48	Tratamento de exceções em Java	75
Gerando a documentação de um programa - javadoc	49	A hierarquia das exceções	76
Documentação da API Java	50	Tratamento pendente de exceções	77
Exercícios	50	Exceções implementadas pelo programador	77
<i>Encapsulamento e sobrecarga de métodos</i>	52	Sinalizando uma exceção (throw)	78
Acessibilidade dos membros de uma classe	52	Throwable.printStackTrace() e	
O modificador final	53	Throwable.getMessage()	78
O modificador private	54	Exceções mais comuns	79
Encapsulamento	55	Exercícios	80
Sobrecarga de métodos	55	<i>Interface gráfica com o usuário</i>	81
Exercícios	56	Componentes gráficos – o pacote AWT	81
<i>Construtores</i>	57	Código do exemplo de componentes AWT	83
Processo de instanciação de objetos	57	Gerenciadores de Layout	86
Inicialização explícita de membros variáveis	57	Containers	86
Construtores	59	Flow layout	87
Exercícios	60	CardLayout	88
<i>Herança e polimorfismo</i>	61	BorderLayout	91
O conceito de classificação	61	GridLayout	93
A palavra reservada extends	62	GridBagLayout	94
Herança em Java	63	<i>Applets</i>	97
A superclasse Object	64	O que é um Applet ? (java.awt.Applet)	97
Polimorfismo	64	Restrições de segurança em applets	97
Argumentos e coleções heterogêneas	65	O primeiro applet	98
O operador instanceof	66	Ciclo de vida de um applet	100
Exercícios	67	Contexto gráfico AWT	101
<i>Classes abstratas e interfaces</i>	68	Aprendendo a usar o appletviewer	103
Classes abstratas e concretas	68	O código HTML de carga do applet	103
O modificador abstract	69	Lendo parâmetros do Html com um applet	105
Restrição de herança pelo modificador final	70	Manipulando imagens	106
As referências this e super	70	Exercícios	107
Exercícios	71	<i>Interfaces gráficas baseadas em behaviorismo – O pacote Swing</i>	108
<i>Tratamento de exceções</i>	73	O que é Swing ?	108
		Containers Swing	108

Introdução a padrões de projeto (design patterns)	109
Programação behaviorista - o paradigma modelo-visão-controle (MVC)	110
Implementando um modelo em Java (Observable)	113
Implementando visões para um modelo	114
Criando o primeiro aplicativo MVC	114
<i>Acesso a dispositivos de entrada e saída - I/O</i>	<i>116</i>
O que são I/O Streams ?	116
Leitura de dados (java.io.InputStream)	117
Escrita de dados (java.io.OutputStream)	118
As classes de manipulação de Streams em Java (o pacote java.io)	118
Leitores e Escretores de dados em Streams com buffer	120
Conversão entre bytes e caracteres	121
O que é UNICODE?	121
Manipulação de arquivos seqüenciais	122
Manipulação de arquivos randômicos	123
Serialização de objetos	124
O que são Grafos de Objetos?	124
Lendo objetos serializados de um arquivo	125
Exercícios	126



A tecnologia Java

Esta seção contextualiza a tecnologia Java no mercado de informática e enumera suas principais características e termos.

O que é Java ?

Java é:

- ?? Uma linguagem de programação
- ?? Um ambiente de desenvolvimento
- ?? Um ambiente de aplicação

Java é uma linguagem de programação desenvolvida pela SUN com o objetivo de manter o poder computacional de C++, agregando características de segurança, robustez e portabilidade.

Os objetivos primários da SUN ao desenvolver a linguagem Java foram:

- ?? Criar uma linguagem orientada a objetos
- ?? Prover um ambiente de desenvolvimento com duas características básicas:
 - o Velocidade de desenvolvimento, eliminando o ciclo *compilar-ligar-carregar-testar* tradicional em outras linguagens como C++, Pascal, etc.
 - o Portabilidade – prover um ambiente meta interpretado, permitindo que o mesmo código rode em diversos sistemas operacionais sem a necessidade de adaptação ou uso de bibliotecas específicas.
- ?? Eliminar exigências de programação que tradicionalmente afetam a robustez de um código de computador:
 - o Aritmética de ponteiros (comum em ANSI C/C++)
 - o Controle de alocação/liberação de memória (comum em Pascal, C, C++, Basic, ...)
- ?? Permitir a programação multitarefa, mesmo em sistemas operacionais que não dêem suporte nativo a *Threads*.
- ?? Permitir que um programa seja dinamicamente modificado através da carga de componentes via redes de computadores, como a Internet.
- ?? Prover um modo de checar a integridade de um programa quanto à sua origem, garantindo a segurança do sistema operacional e de dados durante a sua execução.

A arquitetura Java é formada pelas seguintes características:

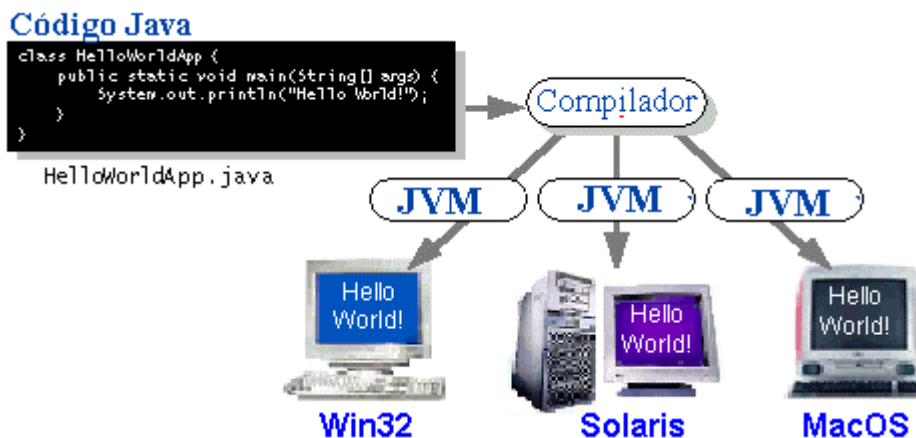
- ?? A máquina virtual Java (JVM – Java Virtual Machine, atualmente na versão 1.4)
- ?? Gerenciador de alocação/liberação de memória (Garbage Collection)
- ?? [Sand box](#) – módulo de garantia de segurança de código (é impossível criar um vírus em Java)

O que é máquina virtual ?

De acordo com a especificação da SUN, a máquina virtual do Java pode ser vista como:

Uma máquina imaginária que é implementada via software ou hardware. Um código a ser executado por essa máquina deve ser gravado em um arquivo com extensão *.class*, e possuir um código compatível com as instruções Java.

Para um programa Java ser executado, ele precisa passar pelo processo ilustrado na figura abaixo:



O código é compilado, gerando um conjunto de instruções chamado de *byte-code*. Esse *byte-code* é aplicado à **Máquina Virtual Java (JVM)** que se encarrega de interpretar os comandos para o sistema operacional onde o programa está rodando. Ou seja, a máquina virtual **traduz** as instruções do código Java para instruções válidas no sistema operacional em que está rodando. Se essa portabilidade fosse requerida em C, o código deveria ser compilado várias vezes – uma para cada sistema operacional desejado. No caso do Java, o código é compilado apenas uma vez, gerando o *byte-code*. Esse *byte-code* poderá então ser interpretado por qualquer máquina virtual Java, rodando em Linux, Windows, Palm OS, Solaris ou qualquer outro sistema operacional que possua uma máquina virtual Java implementada. (*Compile once, run anywhere*).

Uma JVM possui definições concretas para a implementação dos seguintes itens:

- ?? Conjunto de instruções (equivalentes às instruções da CPU)
- ?? Conjunto de registradores
- ?? Formato padrão de classes
- ?? Pilha de memória
- ?? Pilha de objetos coletados pelo *garbage-collector*
- ?? Área de memória

IMPORTANTE: a JVM não permite que um programa Java acesse recursos de hardware diretamente, protegendo o computador de operações perigosas, como acesso à regiões protegidas da memória ou formatação física do disco rígido.

Um programa Java só é executado caso o seu *byte-code* passe pela verificação de segurança da JVM, que consiste em dizer que:

- ?? O programa foi escrito utilizando-se a sintaxe e semântica da linguagem Java
- ?? Não existem violações de áreas restritas de memória no código
- ?? O código não gera *Stack Overflow*

?? Os tipos de parâmetros dos métodos são corretos

?? Não existe nenhuma conversão ilegal entre dados do programa, como a tentativa de conversão de inteiros em ponteiros

?? O acesso a objetos está corretamente declarado

Caso alguma das condições acima não seja satisfeita, a máquina virtual Java causará um erro de execução (*runtime error*).

O que é garbage collection ?

Durante a execução de um programa de computador, ocorre a alocação e liberação dinâmica de memória RAM. Dados são escritos e lidos da memória do computador satisfazendo os requisitos de cada programa. Em linguagens tradicionais como Pascal, Basic e C/C++, o programador é responsável por controlar essa alocação, impedindo o estouro de memória (*stack overflow*) e outros problemas, como o acesso indevido a áreas reservadas de memória. Para facilitar a vida dos programadores, e evitar os erros comuns associados à alocação de memória, a linguagem Java introduziu um novo conceito: o *garbage-collection*.

Garbage-collection é um mecanismo de controle automático de alocação e liberação de memória.

Quando uma variável é declarada em um código de computador, a JVM cria um ponteiro para uma área de memória equivalente ao tamanho do tipo de dado utilizado por essa variável. Quando essa variável é associada a outra região de memória, a JVM coloca o espaço alocado anteriormente em uma pilha de objetos em desuso. Caso o computador fique com pouca memória disponível, a JVM remove objetos dessa pilha, permitindo que esse espaço de memória seja re-alocado.

O processo de garbage-collection ocorre automaticamente durante a execução de um programa Java. O programador não precisa se preocupar com aritmética de ponteiros (grande dificuldade em linguagens como C e Pascal).

A estrutura de um código em Java.

Como todas as outras linguagens de programação, Java possui um formato básico para a escrita de códigos. Tal formato é demonstrado abaixo:

```

1. // Duas barras significam comentário
2. /* comentários também podem seguir o formato de C++ */
3.
4. public class NomeDoPrograma
5. {
6.     // O método main sempre deve estar presente para que um código
7.     // Java possa ser executado:
8.     static public void main(String[] args)
9.     {
10.         // aqui virão os comandos, que são parecidos com C++
11.     }
12. }
```

Compreendendo o código Java:

?? linhas 1 e 2: representam comentários. Um comentário pode conter qualquer informação relevante ao comportamento do programa, autor, versão, etc.

?? linha 3: está em branco, pois Java permite linhas em branco entre os comandos

- ?? linha 4: é a declaração do "nome do programa", que é **case-sensitive** (existe diferença entre maiúsculas e minúsculas). O arquivo que contém o código Java deve ser salvo com o mesmo nome que aparece após a declaração **public class** e mais a extensão .java (o exemplo acima deveria ser salvo como *NomeDoPrograma.java*).
- ?? linha 5 e 9: a abertura de chave { indica início de bloco (tal qual *begin* em Pascal)
- ?? linha 8: essa linha deve aparecer em todos os códigos Java. Quando um programa Java é executado, o interpretador da JVM executa os comandos que estiverem dentro do bloco indicado pelo método "static public void main(String)".
- ?? Linha 10: aqui seria escrito o código propriamente dito. Instruções como for-next, print, etc.
- ?? Linha 11 e 12: o fechamento de chave } indica início de bloco (tal qual *end* em Pascal)

Exemplo de código Java:

```
/**
 * Instituto de Software do Ceará - INSOFT
 * XI Semana tecnológica de férias
 * Primeiro programa - escrever a mensagem alô mundo na tela.
 */
public class AloMundo
{
    static public void main(String[] args)
    {
        System.out.println("Alo Mundo");
    }
}
```

Como rodar o programa acima ?

1. Salve o código acima em um arquivo nomeado: AloMundo.Java (não esqueça o case-sensitive)
2. Digite no console:
 - a. C:\fic>javac AloMundo.java
3. Caso não ocorra nenhuma mensagem de erro, digite:
 - a. C:\fic>java AloMundo

Exercícios

- a. Verifique se o sjdk1.4.1 está corretamente instalado em sua máquina (digite java no console e observe. Caso não aconteça nada, procure a documentação que vem junto ao sjdk1.4.1)
- b. Escreva um programa em Java para imprimir seu nome na tela do computador. Compile e rode esse programa.

Dica: o comando em Java para imprimir mensagens na tela é o seguinte:

`System.out.println("mensagem");` ✍ Não esqueça do ponto e vírgula no final



Implementando o primeiro programa

Ao final da seção anterior, o aluno foi apresentado ao formato de programas Java. O texto abaixo orienta a edição, compilação e execução destes programas.

O ambiente de desenvolvimento

☞ Para que você possa compreender o conteúdo desta aula, é necessário que você já tenha instalado o Java em sua máquina. As instruções de instalação do j2sdk1.4.1 serão apresentadas na aula de laboratório. Para verificar se o Java foi corretamente instalado em sua máquina, faça o seguinte:

- abra um console DOS em seu computador (no Win2000, clique iniciar/executar e digite o comando **cmd**).
- depois que o console estiver ativo, digite o seguinte comando : `Java -version <enter>`
- deverá aparecer uma mensagem parecida com esta:

```
C:\>java -version
java version "1.4.1"
Java(TM) 2 Runtime Environment, Standard Edition (build 1.4.-b92)
Java HotSpot(TM) Client VM (build 1.4.1-b92, mixed mode)

C:\>
```

Caso não esteja funcionando, revise o processo de instalação do j2sdk1.4.1 **antes de continuar essa aula**. Um site que pode lhe ajudar nessa instalação está em:

<http://geocities.com/canaljava/olamundo.html>

Não esqueça de configurar as variáveis de ambiente de seu sistema operacional. Em ambiente Windows, as configurações são as seguintes:

```
path = %path%;c:\j2sdk1.4.1\bin
classpath = %classpath%;.;c:\j2sdk1.4.1\jre\lib
```

* Em ambientes UNIX/Linux, o delimitador das variáveis de ambiente é o dois pontos (:) ao invés de ponto e vírgula.

** Detalhes sobre a instalação podem ser encontrados na página da SUN: [Your First Cup of Java](http://java.sun.com/docs/books/tutorial/getStarted/cupojava/index.html)

<http://java.sun.com/docs/books/tutorial/getStarted/cupojava/index.html>

O código fonte

Como qualquer outra linguagem de programação, Java é usada para criar aplicações de computador. O texto que contém os comandos a serem executados pela JVM é chamado de **código-fonte**, ou simplesmente fonte. O conjunto mínimo de instruções necessário para que um código-fonte seja considerado um programa Java aparece no quadro abaixo:

```
/**
 * Instituto de Software do Ceará - INSOFT
 * XI Semana tecnológica de férias
 * Primeiro programa - escrever a mensagem alô mundo na tela.
 */
public class AloMundo
{
    static public void main(String[] args)
    {
        System.out.println("Alô mundo");
    }
}
```

Analisando o código acima:

- ?? As primeiras 5 linhas representam um bloco de comentário, que tem por objetivo identificar a função do programa, seu autor, versão, etc. Mais adiante na disciplina aprenderemos a utilizar a ferramenta de documentação do Java, chamada *javadoc*.
- ?? A linha seguinte (`public class AloMundo`) declara uma classe chamada AloMundo. Após compilado, esse código gerará um arquivo *AloMundo.class* no mesmo diretório em que se encontra o código fonte. Um código fonte em Java pode descrever mais de uma classe. Após a compilação, cada descrição de classe gerará um arquivo *.class* separado. Observe que pode haver no máximo uma classe *public* dentro de cada código-fonte Java. Caso você inadvertidamente declare mais de uma classe como *public* dentro de um código-fonte Java, ocorrerá um erro de compilação. O corpo da classe (o código que define a classe) deve ser delimitado por chaves, assim como toda a estrutura de dados, decisão ou controle em Java.
- ?? A seguir encontra-se a declaração do método inicial de um programa Java. Todo programa Java começa a ser executado pelo método **main** (tal qual C/C++ e várias outras linguagens). Note que o método main é declarado com uma série de modificadores e com uma matriz de *Strings* como parâmetro. Não se preocupe se no momento esses conceitos parecem confusos, mais tarde você aprenderá o motivo de cada um deles. Por enquanto basta saber o seguinte sobre o método main: `static public void main(String[] args)`
 - o **static**: um modificador utilizado pelo compilador para identificar métodos que podem ser executados apenas no contexto da classe AloMundo, sem a necessidade que um objeto dessa classe seja instanciado.
 - o **public**: o método main pode ser executado por qualquer processo ativo no sistema operacional, incluindo o interpretador Java.
 - o **void**: indica o tipo do valor (*int*, *char*, etc.) a ser retornado pelo método main. Quando um tipo de retorno é declarado como void, significa que o método não retorna nenhum valor. **O método main sempre deverá ser declarado static public void.** Caso contrário o programa não poderá ser executado (*Exception in thread "main" java.lang.NoSuchMethodError: main*).

- o `String[] args`: um array de objetos do tipo `String`, que serve para armazenar a lista de argumentos digitados na linha de comando após o nome da classe a ser executada:

```
C:\al o>java AloMundo nome numero ...

C:\al o>_
```

Esses argumentos são acessados pelo índice da matriz `args`, ou seja, `args[0] = nome`, `args[1] = numero`, etc.

- ?? Dentro do método `main`, existe um exemplo de comando em Java. Nosso próximo passo será identificar o conjunto de comandos mais comuns em Java. Por hora, usaremos esse comando apenas para treinar a compilação e execução de programas Java. O método `System.out.println` é utilizado para escrever uma mensagem no dispositivo padrão de saída do Java, que é a tela do computador (mais tarde, aprenderemos como configurar dispositivos de entrada e saída, que podem ser: a tela do computador, uma conexão da Internet, o disco rígido, etc.)

Compilando e executando o programa

Para compilar o programa `AloMundo` proceda da seguinte maneira:

1. Digite o código do `AloMundo`, conforme aparece na figura acima, utilizando um editor de textos padrão Ascii. (Para ter certeza que o seu texto será salvo em Ascii, no windows, use o notepad. Ou então utilize o [editor recomendado pela disciplina](#))
2. Salve o código-fonte do `AloMundo` em um arquivo chamado: `AloMundo.java`
3. Ative um console DOS (no Windows, iniciar/executar/CMD)
4. Vá para o diretório em que você salvou o código-fonte do `AloMundo` (`cd <diretório>`)
5. Execute o compilador Java, passando como parâmetro o nome do arquivo a ser compilado:

```
C:\>_

C:\>cd al o

C:\al o>javac AloMundo.java

C:\al o>_
```

6. Se após executar o comando `javac` o sistema não acusou nenhum erro, é porque o programa `AloMundo.java` foi compilado com sucesso.
7. Para verificar se o byte-code do seu programa foi gerado com sucesso, liste o conteúdo diretório em que você está trabalhando:

```
C:\al o>dir
O volume na unidade C é WINDOWS
O número de série do volume é B4ECEDD8

Pasta de C:\al o

19/02/2002  15:54      <DIR>      .
19/02/2002  15:54      <DIR>      ..
19/02/2002  15:54                190 AloMundo.class
```

```

19/02/2002  12:52                191 AloMundo.java
                2 arquivo(s)                381 bytes
                2 pasta(s)          931.860.480 bytes disponíveis
C:\alo>_

```

8. Observe que o compilador Java criou um arquivo chamado AloMundo.class em seu diretório. Esse arquivo contém o byte-code, o conjunto de instruções executáveis pela JVM.

Para rodar o programa AloMundo, basta carregar a máquina virtual Java (JVM) passando como parâmetro o nome da classe a ser executada:

```

C:\alo>java AloMundo
Al mundo
C:\alo>_

```

Note também que o nome do arquivo aparece **sem a extensão .class**. Esse é um detalhe importante, pois causa problemas à maioria dos iniciantes em Java. O ponto em nome de arquivos obedece ao padrão UNIX, ou seja, se você executar o programa através de seu **nome + . + extensão** o JVM vai pensar que se trata de um subdiretório:

```

C:\alo>java AloMundo.class
Exception in thread "main" java.lang.NoClassDefFoundError: AloMundo/class
C:\alo>_

```

Erros mais comuns

Erros de compilação:

Variável de ambiente (PATH) mal configurada: um dos erros mais freqüentes entre iniciantes Java é a configuração incorreta das variáveis de ambiente PATH e CLASSPATH. Lembre-se que o diretório que contém os arquivos executáveis que compõem a JVM deve estar indicado na variável PATH do sistema operacional (na instalação padrão, esse diretório é o **..\j2sdk1.4.1.1\bin**) Caso contrário ocorrerá o seguinte erro:

```

C:\alo>javac AloMundo.java
'javac' não é reconhecido como um comando interno
ou externo, um programa operável ou um arquivo em lotes.
C:\alo>_

```

SOLUÇÃO: você deve incluir o diretório onde o Java foi instalado na definição da variável de ambiente PATH. No Windows 2000, abra o painel de controle, clique no ícone Sistema e depois na aba Avançado. Você verá um botão chamado Variáveis de ambiente... Clique nesse botão e inclua o diretório da instalação Java na variável PATH. Importante: você vai ter que fechar o console atual e abrir um outro para que a alteração tenha efeito. Para entender melhor o que é uma variável de ambiente, consulte o Help do Windows.

Erro de digitação: muito comum quando ainda não se está acostumado com a sintaxe de uma linguagem. No exemplo abaixo, ao invés de digitar println o usuário digitou printl.

```
C:\alo>javac AloMundo.java
AloMundo.java:9: cannot resolve symbol
symbol  : method printl (java.lang.String)
location: class java.io.PrintStream
        System out.printl("Al¶ mundo");
                        ^
1 error
C:\alo>_
```

SOLUÇÃO: revisar o código. Note que o compilador acusa em quais linhas ocorreram erros.

case-sensitive ou nome de arquivo diferente do nome da classe pública: Outra fonte de erros é o **case-sensitive**, ou seja, no Java palavras minúsculas são diferentes das maiúsculas. Se, por exemplo, o programador salvar o arquivo com o nome de `alomundo.java`, o compilador procurará uma classe chamada `alomundo` e não `AloMundo` ≠ esses dois nomes, em Java, são diferentes.

```
C:\alo>javac AloMundo.java
AloMundo.java:5: class aloMundo is public, hould be declared in a file
named aloMundo.java
public class aloMundo
        ^
1 error
C:\alo>_
```

SOLUÇÃO: revisar o código. Note que o compilador acusa em quais linhas ocorreram erros – no exemplo acima, o erro ocorreu na linha 5.

Erros de execução:

Variável de ambiente (CLASSPATHPATH) mal configurada: durante o processo de compilação, a má configuração da variável `PATH` causava um erro de compilação. Aqui o erro é de execução, e pode ser causado pela ausência da variável de ambiente `CLASSPATH`.

```
C:\alo>java AloMundo
Exception in thread "main" java.lang.NoClassDefFoundError: AloMundo
C:\alo>_
```

SOLUÇÃO: faça o mesmo procedimento de configuração de variáveis de ambiente e ajuste a variável `CLASSPATH` para o diretório que contém as bibliotecas padrão Java (na instalação padrão: `../j2sdk1.4.1/jre/lib`).

Exercícios

- Crie o arquivo `AloMundo.java` com algum editor de textos. Em seguida compile e execute o programa `AloMundo`.
- Modifique o programa `AloMundo` para que ele desenhe os seguintes formatos na tela:

```
C:\alo>java AloMundo

* * * * *      *
* * * * *      * *
* * * * *      * * *
* * * * *      * *
```

IMPLEMENTANDO O PRIMEIRO PROGRAMA

```
* * * * *
```

```
C:\al o>_
```

- c. Dê uma olhada nos arquivos de documentação do j2sdk1.4.1.



Sintaxe da linguagem Java

Nesta seção será apresentado o formato de codificação da linguagem Java como códigos de computador devem ser escritas aluno deve reconhecer Desde o seu surgimento, a tecnologia Java tem contribuindo no amadurecimento de processamento remoto e ligado à Internet. o mercado a se adaptar , nos anos 90, Reconhecida como tecnologia

Delimitadores em Java

Em um código-fonte Java, alguns símbolos são utilizados pelo compilador para diferenciar comandos, blocos de comandos, métodos, classes, etc. Tais símbolos, chamados de **delimitadores**, são enumerados abaixo:

Comentários

Comentários servem para identificar a função de um comando, um trecho de código, ou um método. Além disso, os comentários podem ser utilizados para documentar aspectos de desenvolvimento de um programa: como o nome da empresa que o desenvolveu, nome dos programadores, versão, etc. Existem dois tipos de comentário: o simples e o de documentação.

O comentário simples é delimitado por duas barras // e termina ao final de uma linha.

O comentário de documentação é iniciado pelo símbolo /** e encerrado pelo símbolo */, podendo conter várias linhas de texto e linhas em branco.

```
// comentário simples: não será incluído na documentação
// do programa (note que comentários simples exigem
// um par de barras para cada linha).

/**
FIC - Sistemas Orientados a Objetos I
Exercício de aula
Esse tipo de comentário será utilizado pela ferramenta geradora de
documentação javadoc, que acompanha o ambiente de desenvolvimento
sjdk1.3.1. O uso do javadoc será comentado em aula. A descrição das tags
utilizadas pelo javadoc pode ser encontrada na documentação da Java.

@author FIC - Faculdade Integrada do Ceará
@version 1.0 beta
@date fevereiro de 2002
*/
```

Ponto e vírgula, blocos e espaços em branco: em Java, todo comando é terminado por um ponto e vírgula (;). Exemplo: `System.out.println("note o ponto e vírgula no final 🐼");`

Um bloco é formado por um conjunto de instruções delimitadas por chaves, como no exemplo abaixo:

```
/** Exemplo de bloco */
{
    // Chamamos essa chave de início de bloco
    int ano;    // Note o ponto e vírgula
    ano = 2002; // sempre ao final de um comando Java
}
// Chamamos essa chave de final de bloco
```

o espaço em branco, quebra de linha e caracteres de tabulação são permitidos em qualquer trecho do código-fonte, devendo ser utilizados para realçar o aspecto visual de seu código.

✎ apesar do aspecto visual do código-fonte não ter nenhum impacto no desempenho de um programa Java, o uso de **indentação** é uma característica de bons códigos-fonte. Lembre-se que você não será o único a ler o código-fonte de seus programas, portanto escreva-o da maneira mais organizada e legível possível.

Identificadores e palavras reservadas

Em Java, um identificador é uma sequência de símbolos UNICODE (64K símbolos) que começa com uma letra, um símbolo subscrito `_`, ou o caractere `$`. Os demais símbolos de um identificador podem conter também números. **Identificadores são case-sensitive** e não tem um tamanho máximo estabelecido. Apesar da tabela UNICODE ser bastante extensa, um bom hábito de programação é utilizar somente letras do alfabeto (a-Z) e números para nomear identificadores.

Exemplo de identificadores válidos em Java:

```
?? data
?? _data
?? $data
?? data_do_mês
?? data1
?? uma_variável_pode_SER_bastante_extensa_e_conter_Numeros234876238476
?? data_public_class_NoteQueEsselIdentificadorContemPalavrasReservadas
```

Apesar desta "liberdade" de opções para nomes de identificadores, algumas palavras não são permitidas. Tais palavras são ditas **palavras reservadas**, e representam o conjunto de comandos que forma a sintaxe da linguagem Java. O **conjunto de palavras reservadas** em Java é o seguinte:

abstract	double	int	strictfp
boolean	else	interface	super
break	extends	long	switch
byte	final	native	synchronized
case	finally	new	this
catch	float	package	throw
char	for	private	throws
class	goto	protected	transient
const	if	public	try
continue	implements	return	void
default	import	short	volatile

do

instanceof

static

while

📖 você não precisa se preocupar em memorizar o nome das palavras reservadas em Java. À medida que você for praticando a programação em Java, isso se tornará natural. Além disso, o compilador acusa um erro de nomenclatura quando você tenta utilizar uma palavra reservada para nomear uma variável, um método ou uma classe.

Declarando variáveis em Java

Uma variável é sempre declarada seguindo do seguinte esquema:

<tipo> + <espaço> + **identificador** + ;
 ou
<tipo> + <espaço> + **identificador** + = + **valor** + ;

onde:

<tipo>	é um tipo primitivo de dados ou o nome de uma classe ou interface
identificador	é o nome da variável
valor	é o valor atribuído à variável. Caso você declare uma variável e não atribua nenhum valor, ela não poderá ser utilizada em um código Java – a tentativa de utilizar uma variável não inicializada em Java gerará um erro de compilação.

exemplos:

```
/**
 * FIC - Faculdade Integrada do Ceará
 * Sistemas Orientados a Objetos I
 * Incluindo variáveis no programa AloMundo.
 */
public class AloMundo
{
    static public void main(String[] args)
    {
        boolean obrigatorio;
        int semestre = 2;
        String mensagem = "Alo mundo.";

        System.out.println(mensagem);
    }
}
```

📖 veremos mais adiante que Java possui um mecanismo de inicialização de variáveis de seus tipos primitivos, mas o aluno deve evitar considerar essa inicialização como prática de programação. De fato, esta inicialização automática não funciona para variáveis de tipos agregados ou abstratos de dados e também que o escopo das variáveis – de classe ou de instância - tem influência na sua inicialização. O aluno é fortemente recomendado a pensar em variáveis como espaços alocados na memória RAM, inicialmente podendo conter qualquer valor (conhecido como *lixo na memória*).

Tipos primitivos da linguagem Java

A linguagem Java utiliza oito tipos primitivos de dados e um tipo especial. Esses tipos primitivos podem ser utilizados para declarar constantes ou variáveis utilizadas em programas Java. Os tipos primitivos estão divididos em quatro categorias: lógicos, textuais, numéricos inteiros e numéricos de ponto flutuante.

Tipos lógicos: *boolean*

Valores lógicos possuem dois estados, normalmente ditos *verdadeiro/falso*, *sim/não* e *ligado/desligado*. Em Java um tipo lógico é definido pela palavra *boolean*, e pode assumir dois valores: *true* ou *false*.

```
// Exemplo de variável que suporta valores booleanos
boolean anoBissexto = false;
boolean anoPar = true;

// Apesar de uma variável poder ser declarada
// sem receber um valor, ela só poderá ser usada
// após a atribuição de algum valor a ela.
boolean valido;
```

Tipos textuais: *char* e *String*

Caracteres simples são representados pelo tipo *char*. Um *char* representa um caractere UNICODE, ou seja, um número inteiro sem sinal de 16 bits, no intervalo de 0 até $2^{16}-1$. O valor de um literal *char* deve ser delimitado por aspas simples:

```
// Exemplo de representação de caracteres UNICODE
char primeiraLetra = 'a';
char tabulacao = '\t';

// Código UNICODE para o caractere de interrogação
char unicode = '\u00A0';

// Lembre-se: Uma variável só poderá
// ser manipulada após receber um valor.
char inutil; // variável sem utilidade neste momento
inutil = '@'; // variável útil a partir de agora
```

Palavras são representadas por uma sequência de dados do tipo *char*, agrupadas em um tipo especial de dados: a classe *String*. Apesar de ser uma classe, uma variável do tipo *String* suporta operações como se fosse um tipo primitivo de dados. O valor de uma variável *String* deve ser delimitado por aspas duplas "valor".

```
// Exemplo de uso de variáveis do tipo String
String disciplina = "Sistemas Orientados a Objetos I";

// Uma variável pode receber o valor de outra
```

```
String outraVariavel = disciplina;

// A concatenação de Strings pode ser feita através do
// operador de soma (+)
disciplina = "Sistemas " + "Orientados a Objetos I";

// Concatenação de String com outro tipo de dados:
disciplina = "Sistemas Orientados a Objetos" + 'I';
disciplina = "Sistemas Orientados a Objetos" + 1;

// Para comparar duas variáveis do tipo String
// devemos usar o método equals():
// disciplina == "Sistemas orientados..." ✗ INCORRETO
// disciplina.equals("Sistemas orientados...") ✗ CORRETO
```

✚ a concatenação de qualquer tipo de dado com um dado do tipo String resulta em um novo dado do tipo String.

Tipos numéricos inteiros: *byte*, *short*, *int* e *long*

Existem quatro tipos primitivos de números em Java. Além disso, os valores numéricos podem ser representados de forma decimal, octal ou hexadecimal:

Valores numéricos inteiros em Java:

2	decimal
077	um número que começa com zero está representado de forma octal
0xBABE	representação hexadecimal

✚ todos os valores numéricos em Java tem sinal positivo ou negativo.

Um valor numérico é sempre considerado do tipo int, a menos que seja acompanhado do sufixo L, que representa um valor do tipo long. A diferença de um inteiro para um longo é a capacidade de dígitos que podem ser representados, conforme aparece no quadro abaixo.

Valores numéricos em Java, representados como long:

2L	decimal
077L	um número que começa com zero está representado de forma octal
0xBABEL	representação hexadecimal

```
// Valores inteiros representáveis pelos tipos
// numéricos em Java:
byte a    = 127;           // -27 ... 27 -1
short b   = 32767;         // -215 ... 215 -1
int c     = 2147483647;     // -231 ... 231 -1
long d    = 9223372036854775807L; // -263 ... 263 -1
```

Tipos numéricos de ponto flutuante: *float* e *double*

Um valor fracionário pode ser representado em Java através dos tipos *float* e *double*. A diferença entre esses dois tipos é o tamanho da mantissa:

<i>float</i>	32 bits
<i>double</i>	64 bits

Para um número ser considerado do tipo ponto flutuante, é necessário a inclusão de um ponto, do caractere E (de expoente) ou do sufixo D ou F, conforme mostra o quadro abaixo:

```
// Representação de valores numéricos de ponto flutuante
float pi    = 3.141516;
float taxa  = 6.02E23;
double valor= 123.4E+306D;
```

☞ todo o valor numérico de ponto flutuante é considerado do tipo *double*, a menos que o programador o declare explicitamente como *float*.

Convenções de codificação

Durante a nossa disciplina, adotaremos a seguinte convenção de codificação:

- ?? **Classes** – as classes devem ser designadas por nomes, começando por uma letra maiúscula e depois minúsculas. Cada nova palavra que formar o nome da classe deve ser capitalizada. Ex: *class* Calculadora, *class* CalculadoraCientifica, ...
- ?? **Interfaces** – igual às classes. Ex: *interface* Calculo, *interface* EquacaoLogaritmica, ...
- ?? **Métodos** – métodos devem nomeados por verbos, seguindo o mesmo formato de capitalização das classes. Entretanto, um método sempre deve começar com letra minúscula. Ex: *public void* calcular(*int* numero), *public void* extrairRaiz(*int* numero), ...
- ?? **Constantes** – constantes devem ter todas as suas letras em maiúsculo, com o símbolo de subscrito para separa as palavras. Ex: *final int* ANO = 2002, *final boolean* VERDADE = *true*, ...
- ?? **Variáveis** – tal qual os métodos, as variáveis devem começar com uma letra minúscula e depois alternar a cada palavra. Procure usar nomes significativos para variáveis. Evite declarar variáveis usando apenas um a letra.

Exercícios

- d. Altere o programa AloMundo.Java para que ele imprima na tela todos os tipos primitivos de dados suportados pela linguagem Java.
- e. Altere o programa AloMundo para que ele imprima os parâmetros recebidos pela linha de comando de execução do programa. (Dica: são os valores do array *args*: *args[0]*, *args[1]*, etc.)

- f. Crie um programa que receba três argumentos da linha de comando e imprima-os na mesma linha, em ordem inversa.
Exemplo: java Programa a1 a2 a3
Saída: a3 a2 a1

Expressões

Esta seção apresenta a manipulação de variáveis e implementação de tomada de decisões em programas Java.

Operadores lógicos e aritméticos

Os operadores Java são similares em estilo e função aos operadores da linguagem C/C++. A tabela abaixo enumera esses operadores **em ordem de precedência**:

Delimitadores	. [] () ; ,	Servem para delimitar partes distintas de um comando, método ou classe.
---------------	-------------	---

Ordem de leitura:	Operador :	Função:
Unário	++	Incrementa o valor da variável em uma unidade. Exemplo: i++; contador++;
Unário	--	Diminui o valor da variável em uma unidade. Exemplo: i--; contador--;
⌘	+ -	Operadores aritméticos
⌘	* / %	Multiplicação, divisão, resto
⌘	<< >> >>>	Operadores de deslocamento aritmético e lógico
⌘	== !=	Igualdade e desigualdade
⌘	^	Potência
⌘	&&	AND
⌘		OR
⌘	?:	Operador condicional. Exemplo: i=0; (i>2?i=0:i--);
⌘	= *= %= += -= <<= >>= >>>= &= ^= =	Operadores aplicados sobre a atribuição
⌘	instance of	Identificador de classes

Concatenação de *Strings* com o operador +

Quando o operador + é aplicado a dados do tipo String, ele cria um novo dado do tipo String, concatenando os dois operandos:

```
/**
 * Concatenação de Strings
 */

String sigla = "SOO-I";
String nome = "Sistemas Orientados a Objetos I";
String titulo = sigla + " - " + nome;

// Esse comando imprimirá na tela a frase:
// SOO-I - Sistemas Orientados a Objetos I
System.out.println(titulo);

int i = 10;
String legenda = "valor = ";

// campo é uma variável do tipo String
String campo = legenda + i;
```

Alguns métodos úteis em dados do tipo String:

```
/**
 * Strin: métodos úteis
 */

String disciplina = "Sistemas Orientados a Objetos I";

System.out.println("disciplina: " + disciplina);

// Isolando um caractere:
System.out.print("primeiro caractere: ");
System.out.println(disciplina.charAt(0));
System.out.print("segundo caractere: ");
System.out.println(disciplina.charAt(1));

// O primeiro caractere de uma String tem o
// índice 0, o segundo o índice 1 e assim por diante

// letra = 's';
char letra = disciplina.charAt(2);

// substrings:
System.out.print("primeiras cinco letras: ");
System.out.println(disciplina.substring(0, 5));
System.out.print("letras a partir da quarta: ");
System.out.println(disciplina.substring(4));
```



```
// número de caracteres em uma String:
System.out.print("tamanho da frase: ");
System.out.println(disciplina.lenght() + " letras");

// usando os caracteres de tabulação e quebra
// de linha:
System.out.println(""
    + disciplina.lenght()
    + " letras"
    + " \n"
    + " Nova linha\ttabulação"
    );
```

Promoção e Casting

A linguagem Java não suporta atribuições arbitrárias entre variáveis de tipos diferentes. Por exemplo, você não pode inicializar uma variável inteira com um valor de ponto flutuante sem explicitar isso através de um processo que chamamos de **casting**.

Quando atribuímos um valor a uma variável, e esse valor é incompatível com o tipo de dado definido para a variável, ocorrerá uma conversão. Em alguns casos, essa conversão será automática, em outros o programador deve indicar de que forma o valor será convertido ao tipo de dado da variável.

Quando o processo de conversão for automático, dizemos que ocorreu uma promoção, ou seja, um valor com um tipo de dado foi promovido a outro tipo de dado. Veja no exemplo abaixo:

```
//
// Promoção entre valores de tipos de dados distintos

// Apesar 6 ser um inteiro, o valor da variável grande
// continua sendo do tipo long
long grande = 6;

// Uma variável do tipo inteiro não possui
// espaço para armazenar um valor longo.
// A instrução abaixo é ilegal, e causará um erro de compilação.
int pequeno = 99L;

float a = 12.121F;    // correto
float b = 12.121;     // 12.121 é um double - incorreto
```

Como visto acima, algumas conversões não podem ser realizadas de forma automática, pois o compilador não pode assumir que tipo de conversão ele deve realizar (o tamanho do tipo de dado a ser recebido por uma variável é maior

que o tamanho pré-definido para o tipo dessa variável, logo o compilador não sabe como "ajustar" os bits excedentes). Nesse caso, o programador deve indicar ao compilador que tipo de conversão deverá ocorrer, digitando o tipo de dado que o valor deverá assumir entre parênteses:

```
//
// Casting entre valores de tipos de dados distintos
//

// Apesar 6 ser um inteiro, o valor da variável grande
// continua sendo do tipo long
long grande = 6;
int pequeno = (int)99L;      // sem problemas

float a = 12.121F;
float b = (float)a;          // sem problemas
```

Operadores de deslocamento (>>, <<, >>>)

Java provê operadores para a manipulação dos bits em variáveis de tipo numérico: o **deslocamento aritmético** >> e o **deslocamento lógico** >>>.

O operador de deslocamento aritmético >> executa um deslocamento de um bit para a direita de um número (na prática, o primeiro argumento desse operador é dividido por dois 'n' vezes – onde n é o segundo argumento do operador):

```
8 >> 2 = 2
128 >> 1 = 64
256 >> 4 = 16
```

* Notação em [complemento de dois](#): o operador >> mantém o sinal do bit mais significativo durante o deslocamento.

O operador de deslocamento lógico >>> executa um deslocamento no padrão dos bits ao invés do significado aritmético de um valor numérico. Esse operador sempre adiciona o valor 0 ao bit mais significativo:

```
1010 ... >> 2 = 111010 ...
1010 ... >>> 2 = 001010 ...
```

* Os operadores de deslocamento reduzem seus operandos à direita módulo 32 para um valor do tipo int e módulo 64 para um tipo long. Dessa forma, para qualquer valor do tipo int:

```
int x      ✎      x >>> 32 = x
```

✎ o operador de deslocamento lógico >>> só pode ser aplicado a valores inteiros, e não é efetivo em valores `int` e `long`. Se for aplicado a valor `short` ou `byte`, o valor será promovido a um `int` antes da aplicação do operador. Por isso, um deslocamento sem sinal acaba se tornando um deslocamento com sinal.

Circuitos lógicos

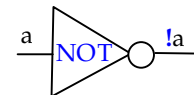
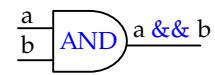
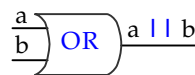
Java possui três operadores básicos para implementar circuitos lógicos :

NOT: operador **!**
 AND: operador **&&**
 OR: operador **||**

Esses operadores permitem a representação de expressões booleanas, que formam o argumento para comandos de decisão (IF), seguindo a seguinte tabela:

AND:	OR:
true && true = true;	true true = true;
true && false = false;	true false = true;
false && true = false;	false true = true;
false && false = false;	false false = false;

```
!true  = false;
!false = true;
!(a && b) = !a || !b
!(a || b) = !a && !b
```



Os comandos de controle (**if**, **while**, **switch**) utilizam o valor de expressões booleanas para guiar o fluxo de controle de um programa, como no exemplo abaixo:

```
/**
 * Comandos de decisão utilizando expressões booleanas
 */
int mes = 2;

if((mes == 12) || (mes == 1))
{
    System.out.println("férias :)");
}

if((mes > 1) && (mes < 12))
{
    System.out.println("semestre em andamento");
}

if((mes != 2))
{
    System.out.println("não tem carnaval ");
}
```

Exercícios

1. Implemente um programa para calcular a área de um trapézio, onde:
h = altura
b = base menor
B = base maior
 $\text{Área} = (h \cdot (b + B)) / 2$
2. Faça o programa acima calcular utilizando valores de ponto flutuante e depois imprima na tela duas informações:
Valor exato da área:
Valor arredondado para inteiro:
3. Calcule o valor das seguintes equações:
 - a. $3 - 2 - 1 + 2 + 1 + 3$
 - b. $2 * 3 - 4 * 5$
 - c. $2 + 6 - 3 / 7 * 9$
 - d. $3 \% 4 - 8$
4. Indique qual o valor verdade das seguintes expressões:
 - a. $(1 > 2)$ // exemplo: `false`
 - b. $(8 == 8)$ // exemplo: `true`
 - c. $((12 - 5) > 6)$
 - d. $(0 < 3) \&\& (8 < 9)$
 - e. $((i++) > i)$
 - f. $((10 * 90 / 50 - 2) == 16)$



Fluxo de controle

E sta seção demonstra como implementar tomada de decisões em códigos Java.

Ramificação **if, else**

A sintaxe básica para declarações **if, else** é a seguinte:

```
if (expressão_booleana)
{
    // bloco de comandos
}
else
{
    // bloco alternativo de comandos
}
```

exemplo:

```
/**
 * Usando if, else
 */
int maior = 10;
int menor = 5;

if (maior > menor)
{
    // (10 > 5) = true
    System.out.println(maior + ">" + menor);
}
else
{
    // (10 > 5) != true
    System.out.println(menor + ">" + maior);
}

// Lembre-se que o controle de fluxo é feito através
// do valor verdade de uma expressão booleana

boolean verdade = (10 > 5);

if (verdade)
{
```

```

        // (10 > 5) = true
        System.out.println(maior + ">" + menor);
    }
    else
    {
        // (10 > 5) != true
        System.out.println(menor + ">" + maior);
    }

```

Ramificação switch

Switch é uma declaração semelhante ao if, mas que usa valores inteiros para a tomada de decisões ao invés de expressões booleanas. (só pode ser usada em dados dos tipos `short`, `int`, `byte` ou `char`). Se o tipo de dado não for inteiro, o comando switch irá executar uma promoção desse valor ao tipo `int` para somente depois executar a ramificação.

A sintaxe básica para declarações `switch` é a seguinte:

```

Switch ((int) expressão)
{
    // bloco de comandos

    case ((int) valor_1):
        // bloco de comandos
        break;
    case ((int) valor_2):
        // bloco de comandos
        break;
    default :
        // bloco de comandos padrão.
        // Se nenhum dos valores acima corresponder à
        // expressão definida no comando switch, então
        // o programa executará o trecho default.
        // o trecho default é opcional.
        break;
}

```

exemplo:

```

/**
 * Usando switch
 */
// Considere valorDoTeclado() como um número
// inteiro digitado pelo usuário
int valor = valorDoTeclado();

switch (valor)
{
    case 0:
        System.out.println("cadastro de produto");
}

```

```

        break;
    case 1:
        System.out.println("emitir nota fiscal");
        break;
    case 2:
        System.out.println("cancelar compra");
        break;
    default:
        System.out.println("efetuar venda");
        break;
}

```

Repetição for

A declaração for é utilizada para definir que um bloco de comandos deve ser executado 'n' vezes, onde 'n' é um número inteiro. A sintaxe do comando for é a seguinte:

```

for (int i = 0; i < 10; i++)
{
    // bloco de comandos
}

```

exemplo:

```

/**
 * repetição de comandos usando FOR
 */

// Calculando o fatorial de um número:
int numero = 10;
int fatorial = 1;

for (int i = numero; i > 0; i--)
{
    fatorial = fatorial * i;
}
System.out.println("fatorial de " + valor
    + " = " + fatorial);

// Imprimindo os dias do ano:
for (int mes = 1; mes < 12; mes++)
{
    switch (mes)
    {
        case 1:
            System.out.println("janeiro");
            break;
        case 2:
            System.out.println("março");
            break;
        case 3:

```

```

        System.out.println("abril");
        break;
        // escreva aqui os demais comandos
    }
}

```

Repetição **while**

A declaração `while` é utilizada para definir que um bloco de comandos deve ser executado enquanto uma expressão booleana (condição de parada) não seja verdade.. A sintaxe do comando `while` é a seguinte:

```

While (expressão_booleana)
{
    // bloco de comandos executados enquanto a
    // expressão booleana tiver valor verdade = true
}

ou

do
{
    // bloco de comandos executados pelo menos uma vez
} While (expressão_booleana);

```

exemplo:

```

/**
 * repetição de comandos usando FOR
 */

// Calculando o fatorial de um número:
int numero = 10;
int fatorial = 1;

int i = numero;
while (i > 0)
{
    fatorial = fatorial * i;
    i--;
}
System.out.println("fatorial de " + valor
    + " = " + fatorial);

// Lendo a condição de parada do teclado:
int numero = 0;
while (numero < 10)
{

```



```

        numero = valorDoTeclado();
    }
    System.out.println("usuário digitou um número maior
    que 10");

```

Comandos especiais de controle de fluxo: **break**, **continue** e **label**

Os comandos de repetição em Java suportam dois tipos de desvios: o **break** e o **continue**. O **break** faz com que um laço seja interrompido, enquanto o **continue** é usado para "pular" uma execução e continuar a partir da próxima.

Exemplo:

```

// pulando a execução No 10 e parando na No 15
for (int i=0; i<20; i++)
{
    if(i == 10)
    {
        continue;
    }
    else if(i == 15)
    {
        break;
    }
    System.out.println("contador: " + i);
}

```

Além disso, esse desvio pode ser direcionado, ou seja, o programador pode indicar para qual linha do código a execução do programa deve ser desviada (uma espécie de GOTO em Java). Essa prática é altamente desaconselhada, e foi mantida no Java por motivos de compatibilidade com a linguagem C.

☞ um **label** em Java só pode ser usado para identificar um comando **for**, **while** ou **do**.

Exemplo:

```

// usando label para controlar o desvio da execução
// de um programa Java
desvio: for (int i=0; i<20; i++)
{
    if(i == 10)
    {
        continue desvio;
    }
    else if(i == 15)
    {
        // Essa linha nunca será executada
        // pois o laço sempre será reiniciado
    }
}

```

```

        // quando o valor de i for igual a 10
        break desvio;
    }
    System.out.println("contador: " + i);
}

```

Lendo valores do teclado

Mais tarde você aprenderá o conceitos de *Streams* e o acesso a dados lidos de arquivos, teclado, etc. Por enquanto basta que você consiga ler valores do teclado para facilitar a implementação dos exercícios propostos. A leitura do teclado será brevemente explicada em aula e, no decorrer da disciplina, revisaremos em detalhes a leitura de dados em dispositivos de entrada e saída.

```

/**
 * FIC - Faculdade Integrada do Ceará
 * Sistemas Orientados a Objetos I
 * Lendo valores do teclado
 */

// Tem que importar a biblioteca de acesso aos
// dispositivos de Entrada e Saída (I/O) do Java:
import java.io.*;

public class Teclado
{
    static public void main(String[] args)
    {
        // Tem que usar tratamento de exceções,
        // conforme explicado em aula.

        try
        {
            // Essas duas linhas criam um "leitor com buffer"
            // do dispositivo padrão de entrada do Java:
            // o teclado (System.in). Mais tarde você aprenderá que esse
            // leitor pode ser redirecionado para ler informações
            // de outros dispositivos, como uma conexão com a Internet,
            // um Socket, o mouse, etc.
            InputStreamReader dados = new InputStreamReader(System.in);
            BufferedReader teclado = new BufferedReader(dados);

            System.out.print("digite uma frase: ");
            String frase = teclado.readLine();
            System.out.println("Frase digitada:\t" + frase);

            System.out.print("\ndigite um numero inteiro: ");
            int numero = Integer.parseInt(teclado.readLine());
            System.out.println("Número digitado vezes dois =\t"
                + (numero * 2));

            System.out.print("\ndigite um numero fracionario: ");
            double flutuante = Double.parseDouble(teclado.readLine());

```

```

        System.out.println("Número digitado dividido por dois =\t"
            + (flutuante / 2));

        // outros métodos:

        // char letra = (char)teclado.read();
        // Float.parseFloat(teclado.readLine());
        // Long.parseLong(teclado.readLine());

    }
    catch(Exception error)
    {
        // Se houver um erro na leitura do teclado,
        // a execução do programa será desviada para
        // o bloco 'catch'
        System.out.println("[ERRO] - "
            + "voce digitou um valor invalido" );
    }
}
}

```

Exercícios

1. Escreva um programa que imprima na tela a soma dos números ímpares entre 0 e 30 e a multiplicação dos números pares entre 0 e 30.
2. Faça um programa para imprimir os números primos de 1 a 123.
3. Faça um programa para ler um número do teclado e imprimir na tela se ele é par ou ímpar. Imprima também se ele é primo.
4. O valor pago por um Hotel da Praia de Iracema para seus porteiros é de R\$ 10,25 por hora de trabalho. Faça um programa que pergunte ao usuário quantas horas ele trabalhou e imprima na tela o valor do salário a ser recebido por ele.
5. Modifique o programa anterior para que o sistema imprima uma mensagem de alerta quando o valor a ser pago ao funcionário seja inferior a R\$ 50,00: "Atenção, dirija-se à direção do Hotel!".
6. Existem 454 gramas em uma libra, e 1000 gramas em um quilo. Faça um programa que converta quilos para libras e vice-versa. (Dica: use um caractere indicando a ordem da conversão, exemplo "java q 1000" seria o comando para converter 1000 quilos para libra, e "java l 1000" seria o comando para converter 1000 libras para quilo)



Agrupamento de dados (Arrays)

Uma dos fatores de qualidade de software é o uso correto de estruturas de dados. Nesta seção analisaremos o agrupamento de dados, uma estrutura simples mas essencial para a compreensão de como Java manipula dados na memória.

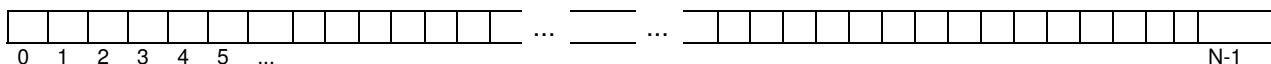
Declarando e criando Arrays

Arrays são tipos indexados de dados que permitem a representação de agrupamento de dados como vetores e matrizes.

Podemos declarar arrays de qualquer tipo de dado suportado em Java (primitivo ou agregado – tipos agregados de dados serão discutidos posteriormente). Essa declaração é feita pela adição do símbolo `[]` a um tipo de dado, que pode aparecer antes ou após o identificador da variável:

```
char[]    arrayDeCaracteres;  
String[]  arrayDePalavras;  
Ponto[]   arrayDeObjetosDoTipoPonto;  
int       números[];
```

Na memória, um array é um agrupamento de dados, indexados pelo tamanho do tipo de dado que o array suporta:



O primeiro índice de um array é sempre 0, e o último índice é o N-1, onde N é o número de elementos do array.

✎ apesar da posição do símbolo `[]` poder ser no início ou no final do nome da variável, é fortemente recomendado que você sempre aplique o `[]` após o tipo da variável. Isso torna o código muito mais legível.

Para criar um array, usamos a palavra chave **new**:

```
int[] números = new int[50];
```

A quantidade de elementos de um array sempre deve ser um valor inteiro. O comprimento de um array é dado pelo método **length**:

```
// Imprimindo o comprimento de um array
```

```
char [] alfabeto = new char[24];
int tamanhoDoAlfabeto = alfabeto.length;
System.out.println("alfabeto com " + tamanhoDoAlfabeto
                  + " letras");

// Muito importante: um array de 10 elementos tem os
// índices de 0 a 9
```

Inicializando os valores de um Array

A linha de comando acima cria uma referência para um vetor com 50 valores do tipo inteiro. No exemplo acima, o array de inteiros é inicializado automaticamente pelo JVM com o valor ZERO em todas as posições do array. Isso acontece porque o int é um tipo primitivo de dados em Java. Em outros tipos de dados (agregados), um array não será inicializado, ou seja, a declaração apenas criará uma referência a uma área de memória que não conterá valor algum.

Por exemplo, olhe o seguinte fragmento de código:

```
// Essas linhas causarão um erro de execução, pois a posição 0 do array
// contendo as frases não possui nenhum valor (null) e, portanto,
// não pode ser usada como parâmetro para nenhum tipo de operação.

String[] frases = new String[10];
int comprimentoDaPrimeiraFrase = frases[0].length();
```

Exception in thread "main" java.lang.NullPointerException

Para que possamos usar os dados contidos em um array, precisamos antes inicializá-los, conforme os exemplos abaixo:

```
// Inicializando arrays

String[] frases = new String[5];

frases[0] = "primeira frase";
frases[1] = frases[0];
frases[2] = frases[0] + frases[1];
frases[3] = "outro texto qualquer";
frases[4] = "último índice do vetor";

// inicialização no momento da declaração de um array:

String[] dias = {"segunda", "terça", "quarta",
                , "quinta", "sexta", "sábado", "domingo"};

int[] meses = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};
Ponto[] coordenadas = {new Ponto(5, 3), new Ponto(0, 2)};
```

☞ arrays de tipos primitivos de dados são inicializados no momento de sua criação. Arrays de tipos agregados de dados necessitam de inicialização explícita.

Array multi-dimensional

Quando declaramos um array através do símbolo [], estamos de fato criando um array unidimensional (vetor). Além de vetores, a linguagem Java permite a declaração de arrays n-dimensionais, conforme os exemplos abaixo:

```
// array unidimensional (VETOR)
String[] frases = new String[5];

// array bidimensional (MATRIZ)
String[][] tabela = new String[5][15];

// array n-dimensional
double[][][] densidade;
densidade = new double[10][10][10];

// observe que nem todos os valores de um array
// precisam estar preenchidos. Sempre que você for
// usar o valor contido em uma posição de um array
// você deve ter certeza que essa posição
// foi inicializada

densidade[0][0][0] = 35.034;
densidade[1][0][0] = 30.876;

System.out.println("densidade na coordenada 1,0,0 = "
    + densidade[1][0][0]);

// arrays com mais de três dimensões são raramente
// encontrados em programas de computador, mas
// são permitidos em Java
float[][][][][] espacoHexaDimensional;
int[][][][][] quartaDimensao = new int[564][2][200][1];
Ponto[][][][][][][] hiperespaco;
```

Array esparsos

Java permite a declaração de estruturas esparsas, ou seja, um agrupamento indexado de dados onde nem todas as dimensões são iguais:

```
// Matrizes esparsas podem ser representadas em Java
// através de arrays com dimensões heterogêneas:

int[][] esperso = new int[4][];
esperso[0] = new int[10];
esperso[1] = new int[2];
```

```

esparso[2] = new int[5];
esparso[3] = new int[1];

// Na memória, o espaço de dados ocupado pelo
// array acima seria aproximadamente assim:
// [ | | | | | ]
// [ | ]
// [ | | | ]
// [ ]

```

Exercícios

1. Faça um programa que enumere os parâmetros recebidos via linha de comando.
2. Faça um programa que leia cinco nomes do teclado, e depois imprima esses nomes na ordem inversa em que foram digitados pelo usuário.
3. Altere o programa acima para que ele continue a ler nomes do teclado até que o usuário digite a palavra "fim", e então imprima os nomes digitados na ordem em que foram digitados.
4. Implemente um jogo-da-velha, com as seguintes características:
 - a. O tabuleiro é formado por um array bidimensional, onde as posições são numeradas conforme abaixo:

0	1	2
3	4	5
6	7	8

- b. Para jogar, os jogadores devem digitar o número correspondente ao quadro que desejam jogar. Caso o quadro já esteja ocupado, ou o valor digitado pelo usuário não seja um número de 0 a 8, o programa deve acusar a jogada inválida e pedir que o jogador digite novamente.
- c. O programa deve considerar dois jogadores (que podem ser nomeados jogadorX e jogadorO), e utilizar dois símbolos distintos para representar a cada um deles.
- d. O programa deve perguntar alternadamente a próxima jogada, ou seja, primeiro pergunta a jogada do jogadorX, depois do jogadorO e assim sucessivamente.
- e. A cada nova jogada o programa deve mostrar o tabuleiro com a situação atual do jogo (seja criativo). Por exemplo:

O	-	-			[O]	[]	[]
X	-	X			[X]	[X]	[]
-	-	-			[]	[]	[]

Ou

Dica: você pode usar os caracteres `\t` para gerar tabulações tela de impressão, e o `\n` para gerar linhas em branco entre os tabuleiros.

- f. Se um dos jogadores ganhar, o sistema deve mostrar uma mensagem de vitória. Se houver empate, o programa deve emitir a respectiva mensagem.
- g. .



Introdução à Orientação a Objetos

E sta seção visa estimular o aluno a pensar no conceito de classificação de objetos, atividade corriqueira do ser humano e que permite grande produtividade quando adotada como paradigma de programação. A migração da codificação estruturada para a orientada a objetos requer disciplina e paciência, mas capacita o aluno a compreender técnicas avançadas de análise e desenvolvimento de software.

Motivação ao uso de um novo paradigma, orientado a objetos.

Programadores iniciantes costumam visualizar variáveis como elementos isolados em um sistema de computador. Por exemplo, digamos que um programa deva representar a carteira de estudante dos alunos da FIC, onde essa identificação seja composta pelo nome e pelo número de matrícula.

Na forma estruturada de programação, o aluno poderia usar o seguinte conjunto de variáveis:

```
// Variáveis que armazenam as informações sobre o RG:  
String nome = "Fernando Henrique Gomes";  
int matricula = 78623423;
```

Apesar de simples, a forma de representação de dados acima permite dois grandes problemas:

Se quisermos representar informações sobre um número maior de alunos, teremos que declarar muitas outras variáveis:

```
// Variáveis que armazenam as informações sobre o RG:  
String nome0 = "Fernando Henrique Gomes";  
int matricula0 = 906231230;  
  
String nome1 = "Ciro Cardoso";  
int matricula1 = 903526345;  
  
String nome2 = "Luis Inácio Jereissat";  
int matricula2 = 003526388;
```

Imagine o exemplo de código acima necessário para manipular a informação de todos os alunos da FIC, cerca de oito mil alunos. Imagine a dificuldade de lidar com tantas variáveis ? Além disso, o número máximo de alunos suportados pelo sistema deveria ser previamente conhecido.

Ainda usando programação estruturada, uma forma um pouco melhor de representação dos dados seria o uso de informações indexadas em arrays:

```
String[] nome = new String[5000];
int[] matricula = new int[5000];

// Variáveis que armazenam as informações sobre o RG:
nome[0] = "Fernando Henrique Gomes";
matricula[0] = 903564756;

nome[1] = "Ciro Cardoso";
matricula[1] = 787564234;

// Mesmo assim seriam necessárias variáveis de indexação (confusas):
int fhg = 0;
int cardoso = 1;

// O uso de dados indexados era o paradigma vigente antes da popularização do
// uso de Orientação a Objetos. Linguagens antigas, como Cobol, tem a indexação
// como formato básico de acesso a dados na memória. Cobol usa inclusive o tamanho
// dos dados como o separador dos valores gravados na memória.
```

Outro problema, o principal, é a representação feita por tipos de dados, ou seja, o nome é uma variável do tipo String e o número de matrícula é uma variável do tipo inteiro. – não há nada no código que os vincule a uma carteira de estudante.

Note que a nível conceitual, existe uma associação entre as variáveis 'nome' e 'numero'. Elas são parte do mesmo "objeto", nesse exemplo, uma carteira de estudante.

No final dos anos setenta, com a popularização dos computadores pessoais e o aumento da presença da informática em nossa sociedade, pesquisadores começaram a notar a necessidade de representar os dados de forma mais fiel à forma como eles eram definidos em cada problema. Ou seja, uma carteira de estudante deveria ter um tipo de dado que representasse a sua estrutura, tal qual ela era conceituada no mundo real (e não um conjunto de variáveis alocadas na memória). Surgiu a Orientação a Objetos, paradigma que iremos estudar a partir de agora.

Tipos agregados de dados

A maioria das linguagens de programação suporta o conceito de variáveis tipadas, ou seja, uma variável pode ser do tipo `int`, `double`, `char`, etc. Embora essas linguagens possuam um bom número de tipos pré-definidos, seria mais interessante que o programador pudesse definir seus próprios tipos de dados, como, por exemplo, o tipo "carteira de estudante".

Essa questão é resolvida através da implementação de Tipos Agregados de Dados (em algumas linguagens, também conhecidos como Tipos Estruturados de Dados ou Registros).

Dados agregados são tipos de dados **definidos pelo programador** no código-fonte de um sistema. Uma vez que um tipo de dado agregado tenha sido definido pelo programador, ele pode ser usado normalmente para declarar variáveis.

Em Java, os tipos agregados de dados são definidos através da palavra reservada **class**:

```
class CarteiraDeEstudante
{
    String nome;
    int numero;
}
```

Note que a palavra **class** deve ser escrita em minúsculo, pois é uma palavra-reservada da linguagem Java. Já *CarteiraDoEstudante* é o nome do tipo de dado que está sendo criado, batizado conforme a preferência do programador. Essa preferência deve ser regida por dois fatores:

- ✍ O nome de um tipo de dado criado pelo programador deve ser fiel à função desse tipo de dado. No nosso exemplo, para representar uma carteira de estudante, não faria sentido criar um tipo de dado chamado: **class** Salario.
- ✍ O nome de uma classe deve obedecer ao padrão de codificação especificado na empresa em que o programador está trabalhando. Aqui na nossa disciplina, estamos utilizando o padrão sugerido pela SUN, onde o nome de toda a classe deve começar com uma letra maiúscula e ser alternado a cada palavra que compõe o nome. **O padrão de codificação é muito importante para evitar a confusão de tipos primitivos com tipos definidos pelo programador, entre outros detalhes que serão discutidos no decorrer da disciplina.**

☞ o conceito de classes é bem mais amplo que simplesmente um tipo agregado de dados. As características de uma classe serão apresentadas em detalhes ao longo do curso. Por hora, basta que o aluno tenha consciência da possibilidade de definir seus próprios tipos de dados em Java.

A partir do momento que um programador definiu um tipo agregado de dados, ele pode declarar variáveis desse tipo utilizando o nome da classe como o tipo da variável:

```
// Tipo de dado definido pelo programador
class CarteiraDeEstudante
{
    String nome;
    int numero;
}

// Variáveis declaradas a partir da classe CarteiraDeEstudante:
CarteiraDeEstudante carteira1;
CarteiraDeEstudante carteira2;
CarteiraDeEstudante[] carteirasDaFic = new CarteiraDeEstudante[8000];
```

As partes integrantes de um tipo agregado de dados pode ser acessadas através do operador ponto (.), como no exemplo abaixo:

```
carteira1.nome = "Fernando Inácio";
carteira1.numero = 02378946;
```

```
carteira2.nome = "Luis Gomes";
carteira2.numero = 03648919;
```

Na terminologia de Orientação a Objetos, chamamos os elementos que compõe uma classe de **membros** dessa classe. No nosso exemplo, as variáveis nome e numero são chamadas de membros da classe CarteiraDeEstudante.

Além disso, quando criamos uma instância de um tipo agregado de dados, chamamos essa instância de **objeto**.

Criando objetos

Quando declaramos variáveis de um tipo primitivo de dados (boolean, byte, short, char, int, long, float ou double) o interpretador Java aloca um espaço de memória para essa variável. Quando declaramos um a variável de um tipo de dado definido pelo programador (ou da API Java), o espaço de memória necessário para alocar o valor dessa variável não é alocado imediatamente.

De fato, uma variável declarada a partir da definição de uma classe não representa a informação propriamente dita, mas sim uma **referência** à informação.

☞ Se você preferir, você pode pensar no termo referência como um ponteiro. Muitas linguagens, como C/C++, utilizam a terminologia de ponteiros ao invés de referência.

Antes que você possa utilizar uma variável declarada a partir de uma classe, você deve alocar o espaço de memória necessário para guardar as informações referenciadas por essa variável. Em Orientação a Objetos dizemos que é necessário **criar uma instância de uma classe**. Isso é feito em Java pela palavra reservada **new**, conforme o exemplo abaixo:

```
// Tipo agregado de dados que modela uma classe
// de objetos conhecidos como carteiras de estudante
class CarteiraDeEstudante
{
    String nome;
    int numero;
}

// Um programa de testes, que usa uma variável
// do tipo carteira de estudante
public class Teste
{
    static public void main(String[] args)
    {
        // Usa-se a palavra reservada new para criar um objeto de um tipo
        // de dado definido pelo programador.
        // No nosso exemplo, é criado um objeto da classe CarteiraDeEstudante
        CarteiraDeEstudante carteira = new CarteiraDeEstudante();

        carteira.nome = "Fernando Gomes";
        carteira.numero = 89762347;
```

```

        System.out.println(carteira.nome);
    }
}

```

No exemplo acima, *carteira* é um objeto da classe *CarteiraDeEstudante*.

Note que, a partir do momento em que o objeto é instanciado, seus membros passam a ter um valor real, que pode ser manipulado com o operador ponto (.).

Alocação de memória durante a criação de objetos

Quando declaramos uma variável do tipo de uma classe, o JVM apenas aloca o espaço necessário para armazenar o endereço dessa referência. Quando um objeto é criado e atribuído a essa variável, através do comando **new**, é que a máquina virtual aloca o espaço necessário para armazenar os valores dos membros dos objetos.

Imagine o nosso exemplo das carteiras de estudante. A declaração da variável *carteira* gera a alocação de um espaço de memória necessário para guardar a referência "carteira":

CarteiraDeEstudante carteira;

carteira	???
----------	-----

Após declarar a variável, a construção do objeto força a alocação do espaço ocupado pelas informações da classe *CarteiraDeEstudante*:

```

CarteiraDeEstudante carteira;
carteira = new CarteiraDeEstudante();

```

carteira	???
nome	""
numero	0


Finalmente, quando atribuímos o objeto à variável "carteira", é criada a referência ao objeto recém construído:

```

CarteiraDeEstudante carteira;
carteira = new CarteiraDeEstudante();

```

carteira	CarteiraDeEstudante@113750
nome	""
Numero	0



referência

Atribuição de referências a uma variável

Java trata as variáveis declaradas a partir de uma classe como referências, ou seja, passamos a chamar essas variáveis apenas como referências a uma determinada classe.

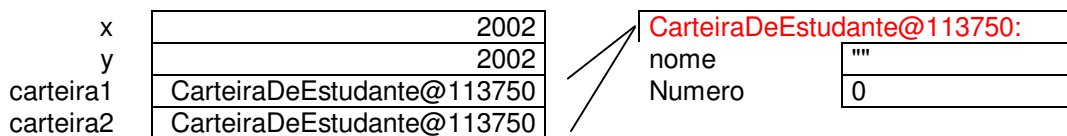
Veja o exemplo:

```
int x = 2002;
int y = x;
CarteiraDeEstudante carteira1 = new CarteiraDeEstudante();
CarteiraDeEstudante carteira2 = carteira1;
```

Quatro variáveis são criadas: duas do tipo `int` e duas referências a objetos do tipo `CarteiraDeEstudante`. O valor de `x` é 2002, e a segunda linha copia o valor da variável `x` na variável `y`. Ambas as variáveis inteiras continuam independentes entre si.

Já com as variáveis `carteira1` e `carteira2` o comportamento é diferente. Quando **instanciamos** a variável `carteira1`, um objeto da classe `CarteiraDeEstudante` é alocado na memória. Quando atribuímos o conteúdo dessa variável à `carteira2`, não estamos criando um novo objeto, mas sim copiando o endereço do objeto referenciado pela variável `carteira1`.

Na memória, o endereçamento ficará assim:



Termos básicos em Orientação a Objetos

Alguns termos são básicos quando estamos descrevendo um sistema desenvolvido a partir do paradigma de Orientação a Objetos. Abaixo enumeramos alguns desses termos:

Tipo agregado de dados: é um tipo de dado definido pelo programador. Agregado pelo fato de ser definido a partir da agregação de um ou mais tipos primitivos de dados em Java.

Classe: A tradução em linguagem orientada a objetos dos tipos agregados de dados. O conceito de classe pode ser visto com uma superclasse do conceito de tipos agregados de dados. Além de agregar tipos primitivos de dados, uma classe provê outras funcionalidades, que serão apresentadas em detalhes ao longo do semestre.

Objeto: Uma instância de uma classe. Podemos considerar uma classe como um gabarito, um modelo a partir do qual criamos objetos. Podemos declarar, por exemplo, uma classe que representa carteiras de estudantes. A classe representa um modelo de carteira de estudante. Já uma instância dessa classe é um conjunto de valores aplicados a esse modelo.

Membro: Um membro de uma classe é um dos elementos que formam o modelo representado pela classe. No nosso exemplo, as variáveis `nome` e `número` são membros da classe `CarteiraDeEstudante`.

Referência: Em Java, uma variável definida a partir de uma classe não contém as informações sobre um objeto dessa classe. Ao invés disso, a variável contém o endereço de memória no qual se encontram esses valores. Tais variáveis são chamadas de referências a um objeto. Um objeto pode ser referenciado por mais de uma variável. Porém, uma variável pode referenciar, no máximo, um objeto.



Abstração de dados

Uma das vantagens da programação orientada a objetos é a capacidade de representar um objeto e o comportamento esperado por esse objeto em um único trecho de código, conhecido como tipo abstrato de dados, ou simplesmente classe.

Tipos abstratos de dados

Quando definimos um tipo agregado de dados, podemos também definir um conjunto de operações que podem incidir sobre esse tipo de dado. Este não é um conceito novo. Quando uma linguagem de programação define um tipo primitivo, tal como um inteiro, ele também define um número de operações que pode ser aplicada a dados desse tipo, como a adição, a subtração, a multiplicação, a divisão, etc.

Algumas linguagens de programação, incluindo Java, permitem uma estreita associação entre a declaração de um tipo de dados e a declaração das operações que incidem sobre as variáveis deste tipo. Essa associação normalmente é descrita como um *tipo abstrato de dados*.

Em Java, você pode criar um tipo abstrato de dados através da implementação de **métodos**:

```
/**
 * Classe que modela o processo de avaliação dos alunos da disciplina
 * Sistemas Orientados a Objetos I
 * @author Felipe Gaúcho © 2002
 * @version exemplo
 */
public class Avaliacao
{
    public float[] trabalhos = new float[4]; // quatro trabalhos
    public float[] provas = new float[2]; // duas provas

    /**
     * Método que permite a inclusão da nota de um trabalho
     * @param numeroDoTrabalho O número do trabalho a ser atribuído a nota
     * @param nota A nota que o aluno tirou no trabalho
     */
    public void atualizarNotaDeTrabalho(int numeroDoTrabalho, float nota)
    {
        trabalhos[numeroDoTrabalho] = nota;
    }
}
```

```

* Método que permite a inclusão da nota de uma prova
* @param numeroDaProva O número da prova a ser atribuído a nota
* @param nota A nota que o aluno tirou na prova
*/
public void atualizarNotaDeProva(int numeroDaProva, float nota)
{
    provas[numeroDaProva] = nota;
}

/** @return A nota do aluno na primeira avaliação parcial */
public float primeiraParcial()
{
    float parcial = (2.0F * provas[0] + trabalhos[0] + trabalhos[1]) / 4.0F;
    return parcial;
}

/** @return A nota do aluno na segunda avaliação parcial */
public float segundaParcial ()
{
    float parcial = (2.0F * provas[1] + trabalhos[2] + trabalhos[3]) / 4.0F;
    return parcial;
}

/** @return A média final do aluno */
public float media()
{
    float media = (primeiraParcial() + segundaParcial() ) / 2.0F;
    return media;
}
}

```

📖 não se preocupe com os modificadores `public` que aparecem na definição dos métodos da classe `Avaliacao`. No decorrer da disciplina iremos estudar os tipos de modificadores de métodos e classes.

Definição de métodos

Linguagens como Java, que comportam tipos abstratos de dados, criam uma estreita associação entre os dados e o código. Não descrevemos um método como uma operação sobre objetos do tipo `Avaliação`. Ao invés disso, consideramos que objetos do tipo `Avaliação` "já sabem" como processar as operações relativas à avaliação de um aluno. Por exemplo:

```

// método inicial de um programa qualquer
static public void main(String[] args)
{
    // Cria uma instância da classe Avaliação e uma referência a esse objeto
    Avaliacao avaliacaoDoJoao = new Avaliacao();

    avaliacaoDoJoao.atualizarNotaDeTrabalho(0, 8.5F); // trabalho 1
    avaliacaoDoJoao.atualizarNotaDeTrabalho(1, 8.0F); // trabalho 2
    avaliacaoDoJoao.atualizarNotaDeTrabalho(2, 9.0F); // trabalho 3
}

```

```

avaliacaoDoJoao.atualizarNotaDeTrabalho(3, 7.5F);           // trabalho 4

avaliacaoDoJoao.atualizarNotaDeProva( 0, 7.2F);           // prova 1
avaliacaoDoJoao.atualizarNotaDeProva( 1, 9.5F);           // prova 2


// note que, nesse trecho do código, não foi necessário calcular a média
// propriamente dita. Bastou chamar o método média da classe Avaliação.
// Por isso, podemos interpretar que os objetos da classe Avaliacao
// "sabem como realizar a operação de cálculo da média das notas".
float mediaDoJoao = avaliacaoDoJoao.média();
System.out.println("Média do João = " + mediaDoJoao);
}

```

```

C:\Felipe\fic\Avaliacao>java Avaliacao
Média do João = 8.3

```

 A idéia de que métodos são uma propriedade dos dados ao invés de considerá-los como algo à parte é um passo importante para a construção de um sistema Orientado a objetos. Algumas vezes, você irá ouvir o termo *message passing*. Esse termo é usado em algumas linguagens para transmitir a noção de instrução para que um item de dados realize algo para si próprio. Na verdade, nas linguagens que usam convencionalmente essa terminologia, ela também reflete a natureza da implementação.

Em Java, os métodos são definidos através de uma abordagem muito semelhante a que é usada em outras linguagens, em especial C e C++. A **assinatura de um método** tem a seguinte forma:

```

<static / final> modificador tipoDeRetorno identificador (listaDeArgumentos)
{
    // bloco de comandos
}

```

A palavra **static** significa que o método não precisa estar associado a um objeto específico. O uso de **static** é opcional. Dizemos que métodos estáticos são métodos de classe, ou seja, podemos chama-lo a partir do nome da classe. Exemplo:

```

System.out.println();
System.exit(0);

```

Note que os métodos *out.println()*; e *exit(0)*; não estão associados a um objeto da classe System, apenas à definição da própria classe.

A palavra **final** também é opcional e será discutida mais tarde.

O **identificador** pode ser qualquer identificador legal, com algumas restrições baseadas nos nomes que já estão em uso. De acordo com o nosso padrão de codificação, o nome de um método sempre deve começar com uma letra minúscula e depois capitalizar a primeira letra das palavras que compõem o identificador.



O **tipoDeRetorno** indica o tipo de valor retornado pelo método. Se o método não retornar nenhum valor, ele deve ser declarado como **void**. O Java é rigoroso quanto aos valores retornados e, se a declaração estabelece que o método retorna um **int**, por exemplo, então você deve proceder assim em todos os caminhos possíveis de retorno.

O **modificador** é uma palavra reservada pertencente ao grupo de modificadores suportados em Java: **public**, **protected** ou **private**. O modificador **public** de acesso indica que o método pode ser chamado a partir de qualquer outro código e o **private** indica que um método só pode ser chamado por outros métodos da mesma classe onde ele está sendo declarado. Discutiremos o **protected** mais adiante.


A **listaDeArgumentos** permite que parâmetros sejam passados para um método. Os elementos da lista são separados por vírgula, enquanto cada elemento consiste em um tipo e um identificador. Por exemplo:

```
public void atualizarNotaDeTrabalho(int numeroDoTrabalho, float nota)
{
    trabalhos[numeroDoTrabalho] = nota;
}
```

a lista de argumentos do método *atualizarNotaDeTrabalho* é composta de dois argumentos:

-  numeroDoTrabalho: um argumento do tipo inteiro (**int**)
-  nota: um argumento do tipo ponto flutuante (**float**)

Note também que esse método não retorna nenhum valor, ou seja, o tipo de retorno é **void** (nulo).

 **todos os argumentos em Java são passados por valor.** Ou seja, o valor original do argumento não será modificado após o retorno do método. Quando a instância de uma classe é usada como argumento de um método, o valor que está sendo passado ao método é a referência a esse objeto – o valor dos atributos desse objeto podem ser modificados no método invocado, mas a referência a esse objeto não. Exemplo:

```
public class Valor
{
    static public void main (String[] args)
    {
        // Aqui a variável 'valor' tem o valor 2
        int valor = 2;

        somaDez(valor);

        // Aqui a variável 'valor' continua com o valor 2
        System.out.println("valor = " + valor);
    }

    // Note que esse método é private, ou seja, só pode ser acessado por
    // objetos da classe Valor. O contexto 'static' será discutido mais tarde
    static private void somaDez(int valor)
    {
        // Aqui temos outra variável , que recebeu o conteúdo do argumento 'valor'
```

```

    }
    valor += 10;
}

```

A referência **this**

Note nos exemplos anteriores que sempre usamos o operador ponto (.) para acessar os membros de uma classe. Exemplo:

```

// Cria uma instância da classe Avaliação e uma referência a esse objeto
Avaliacao avaliacaoDoJoao = new Avaliacao();

avaliacaoDoJoao.atualizarNotaDeTrabalho(0, 8.5F); // trabalho 1

```

Mas note também que dentro da classe Avaliacao, os membros são manipulados sem estarem associados a um objeto:

```

public void atualizarNotaDeTrabalho(int numeroDoTrabalho, float nota)
{
    // Note que a variável trabalhos não está sendo
    // associada a nenhum objeto
    trabalhos[numeroDoTrabalho] = nota;
}

```

Isso é possível porque em Java, os membros de uma classe possuem, dentro da classe, uma referência implícita identificada pela palavra reservada **this**:

```

public void atualizarNotaDeTrabalho(int numeroDoTrabalho, float nota)
{
    // Referência ao objeto corrente
    this.trabalhos[numeroDoTrabalho] = nota;
}

```

Essa referência, na verdade, está associando o membro a um objeto do tipo da classe em que o método se encontra. Em Java não é necessário que se use a referência **this** pois, caso uma referência não seja digitada, a referência **this** será assumida pelo interpretador Java (JVM). Além disso, a referência **this** pode ser usada como argumento para representar o objeto corrente na chamada de um método:

```

/**
 * Classe que modela o processo de avaliação dos alunos da disciplina
 */
public class Avaliacao
{
    public float[] trabalhos = new float[4]; // quatro trabalhos
    public float[] provas = new float[2];   // duas provas
}

```

```

    public void toString ()
    {
        System.out.println("Objeto atual = " + this);
    }

    //... demais métodos ...
}

```

Gerando a documentação de um programa - javadoc

Dentre os programas que compõem o ambiente de desenvolvimento Java, um dos mais interessantes é o gerador automático de documentação, o **javadoc**. Javadoc é um aplicativo que gera arquivos Html a partir de um ou mais códigos fonte Java. Essa documentação é útil para que outros desenvolvedores consultem a funcionalidade das classes implementadas por terceiros.

Por exemplo: suponha que você tenha implementado as classes de um sistema de avaliação de alunos da FIC. E suponha que um outro programador deve usar essas classes para gerar uma interface gráfica que permita que os professores da faculdade atualizem as notas dos alunos via Internet. Esse outro programador deverá ser informado sobre a funcionalidade das classes implementadas por você, ou seja, qual a classe que representa uma avaliação, um aluno, etc. e quais os métodos dessas classes devem ser usados para atualizar as notas, os dados cadastrais, etc.

Essa informação é encontrada na documentação das classes, gerada automaticamente pelo Java conforme mostra exemplo abaixo:

```

C:\Felipe\fic\Avaliacao>md docs

C:\Felipe\fic\Avaliacao>javadoc -d docs Avaliacao.java
Loading source file Avaliacao.java...
Constructing Javadoc information...
Building tree for all the packages and classes...
Building index for all the packages and classes...
Generating docs\overviewtree.html...
Generating docs\index-all.html...
Generating docs\deprecated-list.html...
Building index for all classes...
Generating docs\allclasses-frame.html...
Generating docs\index.html...
Generating docs\packages.html...
Generating docs\Avaliacao.html...
Generating docs\serialized-form.html...
Generating docs\package-list...
Generating docs\help-doc.html...
Generating docs\stylesheet.css...

C:\Felipe\fic\Avaliacao>

```

Os comandos acima executam as seguintes operações:

A primeira linha gera um novo diretório onde serão armazenados os arquivos da documentação: **md <diretório>** (em sistemas operacionais UNIX/Linux, o comando é o **mkdir <diretório>**)

Depois é chamado o aplicativo javadoc, passando como parâmetro da linha de comando o nome do código-fonte (ou dos códigos-fonte) a serem documentados.

Para ver a documentação recém criada, basta usar um browser para ler o arquivo que foi gerado automaticamente dentro do diretório de documentação: `./docs/index.html`

Note que a documentação realça partes da documentação, como o nome do autor, a versão da classe, etc. Esse destaque pode ser definido durante a implementação do código, através de TAGS reservadas pelo javadoc.

Algumas dessas tags, que devem ser usadas dentro de comentários delimitados por `/** */`, são enumeradas abaixo:

<code>@author</code>	o nome de quem implementou a classe (ou da empresa).
<code>@version</code>	a versão da classe
<code>@param</code>	o nome de um argumento usado por um método
<code>@return</code>	o retorno de um método
<code>@exception</code>	o tipo de exceção ao qual um método está sujeito

O aplicativo javadoc tem várias opções que permitem ao programador personalizar a documentação de seus sistemas. Para aprender mais sobre o javadoc, consulte a documentação que acompanha o ambiente de desenvolvimento Java. (normalmente, essa documentação se encontra no diretório: `../jdk1.3.1/docs/tool docs/javadoc/index.html`, mas pode variar conforme o sistema operacional e as opções utilizadas durante a instalação do jdk).


Documentação da API Java

Agora que você está familiarizado com o formato de documentação usado pelo Java, podemos investigar a API Java (Application Programming Interface – interface de programação de aplicativo).

A API Java é um conjunto de classes disponíveis aos programadores, desenvolvidas pela SUN e que visam agilizar a produção de softwares compatíveis com a linguagem Java.

Essas classes se encontram no diretório `../jdk1.3.1/jre/lib/rt.jar`. (Os arquivos `*.jar` podem ser "investigados" com aplicativos de compactação de arquivos, tais como o WinZIP)

A documentação dessas classes está publicada na documentação do ambiente de desenvolvimento Java (normalmente no diretório `../jdk1.3.1/docs/api/index.html`). Note que essa documentação foi criada também pelo javadoc, e que tem o mesmo formato da documentação das classes

 é vital que o aluno se habitue a consultar essa documentação durante o desenvolvimento de programas em Java. Nesse primeiro momento, procure navegar pela documentação da API Java, identificando o formato da informação, os termos, etc. Outra boa iniciativa é consultar também a documentação do ambiente de desenvolvimento Java, que normalmente se encontra no diretório `../jdk1.3.1/docs/index.html`. **Conhecer amplamente a API Java é prerrogativa para todo bom programador Java.**

Exercícios

1. Implemente um classe que represente um aluno da FIC, contendo três membros:
 - a. O nome do aluno
 - b. O número de matrícula do aluno
 - c. Um objeto da classe Avaliacao

- Depois implemente o método principal dessa classe com o seguinte comportamento:

```
/**
 * Perguntar ao usuário o nome, número de matrícula e as
 * seis notas de um aluno. Depois, imprima a identificação e a média do aluno,
 * dizendo se ele foi:
 *         aprovado (média > 6.9)
 *         dependente de prova final (3.9 < média <7.0)
 *         reprovado: (média < 4.0)
 */
static public void main(String[] args)
{
}
```

- Copie e salve a classe Avaliacao.java no mesmo diretório em que você gravou a sua classe Aluno. Compile as classes Avaliacao.java e Aluno.java, e depois execute o programa Aluno.class
- Comente os membros que você implementou na classe Aluno e gere a documentação das duas classes. Observe o resultado.



Encapsulamento e sobrecarga de métodos

E sta seção apresenta o controle sobre a acessibilidade dos membros de uma classe e a implementação de classes com comportamento dinâmico através da sobrecarga de métodos.

Acessibilidade dos membros de uma classe

Considere a criação de uma classe, por exemplo, a classe Avaliacao vista na aula passada. Quando instanciamos um objeto da classe Avaliacao, seus membros podem acessados por quaisquer outros objetos presentes na Máquina Virtual Java (JVM). Isso é definido pelo modificador **public** que antecede a declaração desses membros.

```
// Acessando os membros da classe carteira de estudante:

Avaliacao notas = new Avaliacao();

// Note que um valor inválido pode ser atribuído
// a um membro público de uma classe
notas.trabalhos[0] = -4.5F; // Nota menor que zero
notas.provas[1] = 20F;      // Nota maior que 10
```

Algumas vezes, entretanto, precisamos evitar que valores inválidos ou comportamentos inválidos sejam atribuídos aos membros de uma classe. No exemplo acima, uma nota jamais poderia ser negativa ou maior que dez. Como os membros da classe Avaliacao são todos públicos, a consistência desses membros deve ser garantida externamente ao código da classe, o que torna a consistência dos objetos dessa classe muito frágil.

Para contornar isso, a Orientação a Objetos define outras formas de acessibilidade aos membros de uma classe, através dos modificadores **final**, **protected** e **private**. Através desses modificadores, o programador pode **ENCAPSULAR** os membros de uma classe, de forma a garantir a sua consistência. O membro de uma classe pode ser:

- **Público** – declarado com o modificador **public**, e acessível por qualquer objeto instanciado na máquina virtual. Normalmente declaramos públicos apenas alguns métodos de uma classe. Os membros variáveis são todos declarados privados, salvo raras exceções.
- **Privado** – declarado com o modificador **private**, e acessível somente por métodos da de objetos da própria classe onde o membro foi declarado.

- **Protegido** – declarado com o modificador `protected`, e acessível somente por métodos de subclasses da classe onde o membro foi declarado ou da própria classe onde o membro foi declarado. Esse modificador será estudado junto com os conceitos de herança e polimorfismo, mais tarde em nossa disciplina.
- **Constante** – um membro declarado com o modificador `final` não pode ter a sua definição modificada por nenhum objeto, nem mesmo um objeto da classe onde o membro foi definido. Dizemos que um membro final é uma constante. No caso de herança, uma subclasse não pode sobrecarregar um método declarado como final na superclasse. (isso será estudado mais adiante).

O modificador `final`

Membros declarados com o modificador final são membros constantes, ou seja, cujo valor não pode ser mais modificado.

Exemplo:

```
/**
 * Classe que modela o processo de avaliação dos alunos da disciplina
 * Sistemas Orientados a Objetos I
 * @author Felipe Gaúcho © 2002
 * @version exemplo
 */
public class Avaliacao
{
    final int numeroDeProvas=2; // um membro final deve ser inicializado na declaração
    final int numeroDeTrabalhos=4; // ou no construtor (veremos construtores mais tarde)

    public float[] trabalhos = new float[numeroDeTrabalhos];
    public float[] provas = new float[numeroDeProvas];

    // Um método declarado FINAL não poderá ser sobrescrito em subclasses da
    // classe Avaliacao (veremos isso mais adiante)
    final public void teste()
    {
        // Essa linha causará um erro de compilação
        numeroDeProvas = 5;
    }
}
```

```
C:\Felipe\fic\Avaliacao>javac Avaliacao.java
Avaliacao.java:48: cannot assign a value to final variable numeroDeProvas
numeroDeProvas = 5;
^
1 error
C:\Felipe\fic\Avaliacao>
```

O modificador **private**

Membros declarados com o modificador `private` são acessíveis somente por métodos da própria classe onde foram definidos.

Aqui está o segredo da consistência: nos métodos públicos o programador codifica uma série de verificações à validade das operações e das modificações dos valores dos membros variáveis da classe (crítica). Caso a chamada ao método não agrida à consistência do objeto, o método atualiza os valores ou executa a requerida operação. Por exemplo:

```
public class Avaliacao
{
    final int numeroDeProvas = 2;
    final int numeroDeTrabalhos = 4;

    // Observe que os membros agora são privados. Só podem ser acessados
    // pelo objeto que os possui.
    private float[] trabalhos = new float[numeroDeTrabalhos];
    private float[] provas = new float[numeroDeProvas];

    /**
     * Método que permite a inclusão da nota de um trabalho
     * @param numeroDoTrabalho O número do trabalho a ser atribuído a nota
     * @param nota A nota que o aluno tirou no trabalho
     */
    public void atualizarNotaDeTrabalho(int numeroDoTrabalho, float nota)
    {
        // Verifica se o índice da nota é válido
        if(numeroDoTrabalho > -1
            && numeroDoTrabalho < numeroDeTrabalhos)
        {
            // verifica se a nota é válida
            if(nota >= 0F && nota <= 10F)
            {
                // Notas só serão atribuídas se forem válidas
                // e para o índice correto. Ou seja,
                // os membros variáveis de um objeto
                // dessa classe sempre
                // estarão consistentes
                trabalhos[numeroDoTrabalho] = nota;
            }
            else
            {
                System.out.println("nota inválida");
            }
        }
        else
        {
            System.out.println("número do trabalho inválido");
        }
    }

    // demais definições da classe (corpo da classe)
}
```



```
}
```

Encapsulamento

O acesso exclusivo por métodos aos membros variáveis de uma classe e a implementação da verificação de consistência nesses métodos formam o conceito de **encapsulamento**.

Encapsulamento é a garantia que os objetos de uma classe sempre serão consistentes quanto aos valores e comportamento esperados para esse objeto. O nome vem da idéia de que o objeto é protegido por uma cápsula, acessível somente através de métodos que possuem um controle rígido de consistência. Isso é muito importante na distribuição de classes, pois um programador pode declarar objetos de uma determinada classe em seus programas, confiando que o comportamento desse objeto obedecerá à documentação da classe.

Sobrecarga de métodos

Algumas vezes, um problema pode exigir que uma classe permita que uma mesma operação seja aplicada a diversos tipos de dados. Por exemplo: suponha que você esteja implementando uma classe chamada *Matematica*, com o intuito de prover as quatro operações aritméticas: soma, subtração, multiplicação e divisão. Note que essas operações podem ser aplicadas a números inteiros ou fracionários, e seria tedioso para o programador ficar diferenciando os métodos para cada tipo de argumento (ex: *somaDeInteiros()*, *somaDeDouble()*, etc..)


A solução para isso é conhecida em Orientação a Objetos como **sobrecarga de métodos**, exemplo:

```
public class Matematica
{
    /**
     * Método que soma dois inteiros
     * @param operando1 um número inteiro
     * @param operando2 um número inteiro
     * @return a soma dos dois operandos
     */
    public int soma (int operando1, int operando2)
    {
        return operando1 + operando2;
    }

    /**
     * Método que soma um número com ele mesmo
     * @param operando um número inteiro
     * @return a soma do numero com ele mesmo
     */
    public int soma (int operando)
    {
        return operando + operando;
    }

    /**
```

```
* Método que soma dois número de ponto flutuante
* @param operando1 um número de ponto flutuante
* @param operando2 um número de ponto flutuante
* @return a soma dos dois operandos
*/
public float soma (float operando1, float operando2)
{
    return operando1 + operando2;
}
```

 algumas linguagens, como C++, permitem a sobrecarga de operadores, mas **Java não permite a sobrecarga de operadores**.

Os critérios para a sobrecarga de métodos são os seguintes:

- A lista de argumentos deve ser diferente o suficiente para evitar a ambigüidade entre os métodos. O programador deve prever o uso de promoções e casting, que podem causar confusão na chamada dos métodos.
- O tipo de retorno dos métodos pode ser diferente, mas não basta para que seja caracterizada a sobrecarga. A lista de argumentos deve apresentar alguma diferença, como o número de argumentos ou o tipo desses argumentos.

Exercícios

7. Reescreva a classe Avaliacao.java da aula passada, garantindo que os valores das notas e o cálculo das médias seja sempre consistente. Discuta com seus colegas quais modificadores devem ser aplicados aos membros dessa classe para que os objetos sejam sempre consistentes.
8. Reescreva a classe Aluno.java da aula passada, realizando testes de consistência a partir da classe Avaliacao.java que você reescreveu acima
9. Generalize a classe Avaliacao.java permitindo que uma disciplina possa também adotar valores inteiros para as médias de seus alunos. Teste com a classe Alunos.java.



Construtores

Conforme avançamos no estudo de programação orientada a objetos, aumentamos o grau de abstração de nosso discurso - passamos a citar objetos como se eles fossem palpáveis dentro de nosso sistema. Entretanto, uma visão mais atenta nos lembra que os objetos continuam sendo abstrações de bits armazenados na memória RAM do computador. Nesta seção o aluno é orientado a pensar na forma como os sistemas criam objetos e como os membros de suas classes são organizados na memória em tempo de execução.

Processo de instanciação de objetos

Quando declaramos uma variável de um tipo abstrato de dado, uma classe, estamos apenas criando um a referência a um objeto dessa classe. Esse objeto só passará a existir quando o comando **new** for interpretado pela JVM. Esse comando instancia um objeto a partir do construtor definido para a classe em questão.

A instanciação de um objeto em Java obedece ao seguinte roteiro:

1. O espaço para o novo objeto é alocado na memória da máquina virtual e inicializado com **valores padrão**. Em Java, nenhum objeto pode ser instanciado com os valores de seus membros aleatórios (tal qual ocorre em C++). Por exemplo, quando declaramos um array de inteiros de 30 posições em Java, esse array será preenchido com o valor ZERO no momento de sua alocação na memória .
2. A seguir é executada a **inicialização explícita** dos membros do objeto, caso essa inicialização faça parte do código da classe desse objeto.
3. Finalmente, o **construtor** da classe é executado.

Inicialização explícita de membros variáveis

Quando instanciamos um objeto de um determinada classe, os membros variáveis desse objeto recebem os seguintes valores padrão:

Tipo:	Valor inicial:
int, byte ou long	0
float, double	0.0
Char	" (valor 0)
String	""
Demais classes	Null

Muitas vezes, porém, pretendemos definir os valores padrão para os membros dos objetos de uma classe. Tal definição é feita através da inicialização explícita de membros variáveis. Exemplo:

```
/**
 * Classe que modela o processo de avaliação dos alunos da disciplina
 * Sistemas Orientados a Objetos I
 * @author Felipe Gaúcho © 2002
 * @version exemplo
 */
public class Avaliacao
{
    // Membros de classe, constantes, normalmente são declarados
    // estáticos (static), pois a referência de todas as instâncias
    // dessa classe farão referência ao mesmo espaço de memória alocado
    // para esses membros. Padrão de codificação: variáveis finais são
    // declaradas sempre com todas as letras maiúsculas, e as palavras
    // são separadas por subscritos (_)
    static final int NUMERO_DE_PROVAS;
    static final int NUMERO_DE_TRABALHOS;

    private String nomeDaDisciplina = "SOO-I";

    // O uso do modificador private faz com que esse
    // membros variáveis sejam membros do objeto e não da classe
    private float[] trabalhos;
    private float[] provas;

    // Note que isso é um bloco inicializador e não um método, pois não
    // tem assinatura.
    {
        try
        {
            trabalhos = new float[NUMERO_DE_TRABALHOS];
            provas = new float[NUMERO_DE_PROVAS];
        }
        catch(Exception erro)
        {
            // O objetivo do bloco inicializador é evitar que eventuais
            // erros de inicialização explícita fiquem sem tratamento.
            erro.printStackTrace();
            System.exit(0);
        }
    }

    // Por ser estática, essa inicialização será executada antes de outras
    // inicialização que não sejam estáticas, como a inicialização acima.
    static
    {
        NUMERO_DE_PROVAS = 2;
        NUMERO_DE_TRABALHOS = 4;
    }
}
```

```

    // restante do corpo da classe....
}

```

Construtores

Algumas vezes, os valores dos membros de um objeto só serão conhecidos no momento de sua instanciação. Esse comportamento dinâmico é previsto em Orientação a Objetos através da implementação de construtores. Em Java, um construtor é um método, identificado com o nome da classe a que pertence, podendo conter argumentos. Exemplo:

```

/**
 * Classe que modela o processo de avaliação dos alunos da disciplina
 * Sistemas Orientados a Objetos I
 * @author Felipe Gaúcho © 2002
 * @version exemplo
 */
public class Avaliacao
{
    // inicializadores (vide exemplo anterior)

    // Construtor padrão, sem argumentos
    Avaliacao()
    {
        System.out.println("objeto da classe Avaliacao");
    }

    // Construtor com argumentos
    Avaliacao(int[] notas)
    {
        if(notas.length != (numeroDeProvas + numeroDeTrabalhos))
        {
            System.out.println("erro nos parâmetros do construtor");
            // O comando return pode ser usado para encerrar a
            // execução de um construtor
            return;
        }
        else
        {
            // A referência this é opcional (implícita)
            this.atualizarNotaDeProva(0, notas[0]);
            atualizarNotaDeProva(1, notas[1]);

            this.atualizarNotaDeTrabalho(0, notas[2]);
            this.atualizarNotaDeTrabalho(1, notas[3]);
            this.atualizarNotaDeTrabalho(2, notas[4]);
            this.atualizarNotaDeTrabalho(3, notas[5]);
        }
    }
}

```

```
}
```

☞ quando não declaramos nenhum construtor para uma classe, o interpretador da JVM considera um construtor padrão, onde todos os membros dos objetos criados serão inicializados com seus respectivos valores padrão.

Exercícios

1. Construa um sistema de cadastro de avaliações de alunos, utilizando os conceitos das aulas 8, 9 e 10. Lembre-se que os objetos do seu sistema devem ser todos consistentes, e que o sistema deve obedecer ao padrão de codificação. Seu programa deve ter pelo menos as três primeiras classes, identificadas abaixo (nomes sugeridos):
 - a. **Aluno.java** – uma classe que representa a classe dos alunos da FIC. Um aluno da FIC, para o nosso exercício, é um objeto que dentre os seus membros possui:
 - i. Carteira de estudante
 - ii. Um relatório de avaliação
 - b. **Avaliacao.java** – uma classe que representa uma avaliação padrão dos alunos da FIC, composta pela nota de duas provas e quatro trabalhos. (você pode definir seus próprios critérios de avaliação, mas eles tem que ser claros).
 - c. **CarteiraDeEstudante.Java** – uma classe que representa uma carteira de estudante, contendo pelo menos o nome e o número de matrícula do aluno. Você pode definir outros membros dessa classe, como o número do curso, o nome da faculdade, etc. Dica: crie um construtor que lhe permita preencher os dados da carteira de estudante do aluno no ato da instanciamento dos objetos dessa classe.
 - d. **CadastroDeAvaliacao.java** – é o sistema propriamente dito (a única classe que precisa ter método `static public void main(String[] args){}`), essa classe identifica e executa comandos digitados pelo usuário. Essa classe pode usar um array de tamanho pré-estabelecido para guardar os objetos que representam os alunos. Uma forma mais elegante é que esse array seja dimensionado no momento da instanciamento dessa classe. Dica: crie um construtor que lhe permita dimensionar o array que armazena os objetos.
2. Gere a documentação do seu sistema com o javadoc.



Herança e polimorfismo

A relação entre as classes de um sistema orientado a objetos.

O conceito de classificação

Quando implementamos uma classe, estamos modelando alguma coisa, um empregado, um automóvel, um animal, etc. Muitas vezes, porém, precisamos refinar esse modelo para representar subgrupos de objetos dentro dessa classe.

Por exemplo, suponha que você crie uma classe para representar os funcionários da FIC:

```
public class Funcionario
{
    private String nome;
    private String funcao;
    private Date dataDeAdmissao;
    private double salario;

    public void decimoTerceiro(Date dataAtual)
    {
        // cálculo do décimo terceiro salário
    }

    public void alterarSalario(double novoSalario)
    {
        salário = novoSalario;
    }

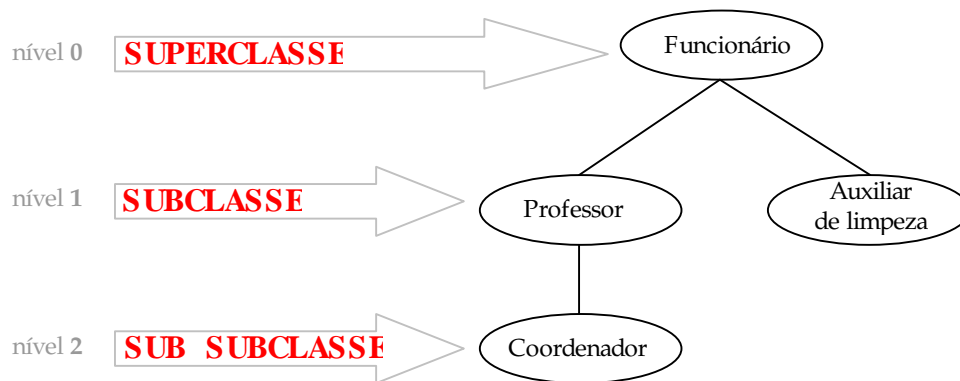
    // restante do corpo da classe
}
```

Agora imagine que você deseja representar os seguintes tipos de funcionários da FIC: um coordenador de curso, um professor e um auxiliar de limpeza. Note que essas três classes de funcionários possuem muitos membros variáveis em comum, como data de admissão, salário, função, nome, etc. E possuem também alguns métodos em comum, como o cálculo do décimo terceiro, alteração de salário, etc. Porém, cada uma dessas classes possui um conjunto de métodos exclusivos, ou seja, que só se aplicam a seus objetos e não a objetos de outras classes. Por exemplo: um auxiliar de limpeza tem o método `public void limpar();`, o coordenador tem o método `public void criarTurma(Turma aula);` e o professor tem o método `public void lecionar();`. Observe também que um coordenador é por sua vez um professor, e portanto possui todos os métodos de um professor e mais alguns exclusivos da classe Coordenador.

Se fôssemos criar uma classe para cada uma dessas categorias de funcionários, acabaríamos repetindo muito código, além do quê, perderíamos a importante relação existente entre os objetos dessas classes: todos são funcionários.

Para resolver isso, a Orientação a Objetos define o conceito de herança, permitindo que subclasses sejam criadas a partir de uma classe original, mais genérica e conhecida como superclasse.

No nosso exemplo, teremos a seguinte hierarquia de classes:



Uma das grandes vantagens do uso de subclasses é o aumento da **confiabilidade e** facilidade de **manutenção dos sistemas**. Se, por acaso, precisarmos mudar o comportamento da classe Funcionario, as demais classes refletirão essa mudança sem a necessidade de serem todas reimplementadas. Por exemplo, se a regra para o cálculo do décimo terceiro salário for alterada pelo governo, bastará ao desenvolvedor alterar um único trecho de código, justamente o método `public void decimoTerceiro(Date dataAtual)` da classe Funcionario. Caso contrário o desenvolvedor teria que alterar todas as classes que implementassem o cálculo de décimo terceiro salário.

Observe também que quanto mais abaixo na hierarquia, mais especializado se torna uma classe. A idéia é que as classes superiores sejam bastante genéricas, permitindo a representação mais abrangente possível das entidades que inspiraram o projeto dessa classe.

A palavra reservada **extends**

Em Java usamos a palavra reservada **extends** para definir uma subclasse. Essa palavra traduz a idéia de que uma classe está estendendo as características e a funcionalidade de sua superclasse. No nosso exemplo teríamos:

```
// Essa classe herda todos os atributos e a funcionalidade da classe Funcionario
public class AuxiliarDeLimpeza extends Funcionario
{
    public void limpar(Sala aula)
    {
        // método que limpa uma sala de aula
    }
}
```



```

    // restante do corpo da classe
}

```

```

// Essa classe herda todos os atributos e a funcionalidade da classe Funcionario
public class Professor extends Funcionario
{
    public void lecionar (Disciplina aula)
    {
        // método que processa o ensino da disciplina referenciada por 'aula'
    }
    // restante do corpo da classe
}

```

```

// Essa classe herda todos os atributos e a funcionalidade da classe Funcionário e também
// da classe Professor
public class Coordenador extends Professor
{
    public void criarTurma(Turma turma, Curso curso)
    {
        // método que cria uma nova turma de alunos
    }
    // restante do corpo da classe
}

```

Note que as classes estendidas de Funcionário herdam todas as características dessa classe. Por exemplo, um objeto da classe Professor tem os membros variáveis *nome*, *função*, etc. bem como os métodos *alterarSalario* e *decimoTerceiro*:

```

// Fragmento de código
{
    Coordenador sid = new Coordenador();
    Professor pardal = new Professor();

    // Esses seriam comandos válidos, considerando que o valor dos membros
    // já tenham sido inicializados
    System.out.println(pardal.decimoTerceiro());
    System.out.println(sid.funcao);
}

```

Herança em Java

Em Orientação a Objetos usamos a palavra herança para expressar a idéia de que uma classe herda todas as características e funcionalidade de uma outra classe. No nosso exemplo, as classes Professor e AuxiliarDeLimpeza herdam os membros da classe Funcionário. A classe Coordenador herda os membros da classe Funcionário e também herda os membros da classe Professor.

Um detalhe importante é que na linguagem Java não existe herança múltipla. Não que a herança múltipla seja um conceito errado, pelo contrário, linguagens como C++ implementam herança

múltipla e esse é um conceito bastante polêmico e discutido em congressos sobre linguagens de programação. Os criadores da Java optaram por exigir herança simples por acreditarem que assim o código de torna mais legível e confiável, embora reconheçam que, às vezes, isso causa um trabalho extra aos desenvolvedores Java. A modelagem de alguns sistemas se torna redundante sem herança múltipla, mas na grande maioria dos casos isso não ocorre. Uma alternativa diante da ausência de herança múltipla é o uso de interfaces, um recurso de modelagem que será discutido mais adiante na disciplina.

Outro aspecto importante é o fato de que os construtores não são herdados pelas subclasses. Uma classe estendida de outra herda todos os membros dessa superclasse menos os seus construtores.

A superclasse **Object**

Em Java, toda a classe é implicitamente estendida da classe **Object**, definida na API da linguagem como:

```
public class Object;
```

A superclasse do nosso exemplo poderia ter sido declarada assim:

```
public class Funcionario extends Object;  
é o mesmo que  
public class Funcionario
```

Polimorfismo

Descrever um professor como um funcionário não é apenas uma forma conveniente de expressar a relação entre essas duas classes, é também uma forma de reaproveitamento de modelos. Quando criamos um modelo de funcionários da FIC, dizemos que esses funcionários tem um método chamado *decimoTerceiro()*. Quando depois criamos um modelo de professores da FIC baseado no modelo dos funcionários, assumimos que os professores também são capazes de processar o método *decimoTerceiro()*, sem que para isso tenhamos que reimplementar esse método.

Essa relação leva à idéia de polimorfismo (do grego *poli* + *morfos* = várias formas). Um objeto polimórfico é um objeto que pode ter formas diferentes, funcionalidades diferentes mas continua a pertencer a uma única classe.

No nosso exemplo, um professor e um auxiliar de limpeza são entidades com características e comportamentos diferentes, porém ambos continuam sendo funcionários. Por esse motivo, dizemos que a classe Funcionário é polimórfica, ou seja, não existe um formato único para os indivíduos que constituem a classe Funcionário.

Java, como a maioria das linguagens orientadas a objeto, permite que um objeto seja referido com uma variável que é de um dos tipos da classe-pai:

```
// fragmento de código  
{  
    // Instanciação válida de objetos  
    private Funcionario cleto = new Coordenador();  
    private Funcionario gauchinho = new Professor();  
    private Object robo = new AuxiliarDeLimpeza();  
}
```

☞ quando criamos uma referência a um tipo abstrato de dados, uma classe, o compilador Java considera que o objeto atribuído a essa referência será do tipo declarado e não do tipo usado para criar o objeto. No exemplo acima, os objetos atribuídos às referências terão as seguintes características:

<i>cleto</i>	terá as características de um funcionário e não de um coordenador.
<i>gaucho</i>	terá as características de um funcionário e não de um professor.
<i>robo</i>	terá as características de um objeto e não de um auxiliar de limpeza.

Argumentos e coleções heterogêneas

Parece estranho criar uma referência para um objeto da classe `Funcionario` e deliberadamente atribuir-lhe uma instância da classe `Coordenador`. É verdade, mas há motivos para que você queira alcançar esse efeito.

Ao usar essa abordagem, você pode escrever métodos que aceitem um objeto "genérico", nesse caso, um `Funcionario`, e trabalhar adequadamente em qualquer subclasse dele. Assim, em uma determinada aplicação, você pode implementar um método que tem um objeto do tipo `Funcionario` como argumento e realiza operações sobre ele, independente da forma como esse objeto foi instanciado:

```
// fragmento de código
{
    static public void main(String[] args)
    {
        Funcionario[] funcionarios = new Funcionario[3];

        // coleção heterogênea
        funcionarios[0] = new Coordenador();
        funcionarios[1] = new Professor();
        funcionarios[2] = new AuxiliarDeLimpeza();

        // chamando um método com argumentos heterogêneos
        emitirRelatorio(new Professor()); // emitirRelatorio(Professor)
        emitirRelatorio(funcionarios[0]); // emitirRelatorio(Funcionario)
    }

    // Método que usa uma superclasse como argumento:
    // Note que esse método pode receber o argumento da classe
    // Funcionário ou de qualquer uma de suas subclasses.
    static public void emitirRelatorio(Funcionario empregado)
    {
        System.out.println(empregado.nome + ", " + empregado.salario);
    }
}
```

Uma coleção heterogênea é um agrupamento de coisas diferentes. Em Orientação a Objetos, você pode criar coleções de quaisquer objetos que tenham em comum uma classe ancestral. No exemplo acima, o array *funcionarios* é uma coleção heterogênea de objetos que compartilham a superclasse `Funcionario`. Podemos imaginar outras aplicações do uso de coleções heterogêneas, por exemplo, escrever um método que coloca os funcionários em ordem por salário ou data de admissão sem a preocupação com a função desses funcionários, se eles são coordenadores, professores ou auxiliares de limpeza.

☞ Como em Java toda classe é uma subclasse de `Object`, você pode usar um array de objetos da classe `Object` como uma coleção de quaisquer outros objetos. Os únicos membros variáveis que não podem ser colocadas em uma coleção de objetos do tipo `Object` são membros declarados como tipos primitivos de dados.

Mais tarde estudaremos em detalhes outras formas de usar coleções em Java. Veremos que a linguagem provê algumas classes específicas para o tratamento de coleções, cuja superclasse é a [java.util.AbstractCollection](#). Java é especialmente elegante no tratamento de coleções, o que justifica uma (futura) aula específica sobre esse assunto.

O operador `instanceof`

Sabendo-se que você pode circular objetos usando referências às suas classes-pai, muitas vezes você pode querer saber o que você realmente tem. Esse é o objetivo do operador `instanceof`. Imagine que nossa classe hierárquica seja estendida, de modo que temos:

```
// fragmento de código
{
    static public void main(String[] args)
    {
        cumprimentar(new Professor());
        cumprimentar(new Coordenador());
        cumprimentar(new AuxiliarDeLimpeza());
        cumprimentar(new Funcionario());
    }

    // Note que esse método usa a instância do objeto recebido
    // como argumento para adotar um comportamento diferenciado
    // para cada subclasse de Funcionário. Ou seja,
    // o operador instanceof permite a tomada de decisões
    // a partir da classificação dos objetos.
    static public void cumprimentar(Funcionario empregado)
    {
        if(empregado instanceof Coordenador)
        {
            System.out.println(
                "Seja bem vindo Sr. " + empregado.nome);
        }
        else if(empregado instanceof Professor)
        {
            System.out.println(
                "Bom dia professor " + empregado.nome);
        }
        else if(empregado instanceof AuxiliarDeLimpeza)
        {
            System.out.println(
                "Fala aí, grande " + empregado.nome);
        }
        else
        {
            // Funcionário genérico
        }
    }
}
```

```
        }  
    }  
}
```

```
        System.out.println("olá.");
```

Exercícios

1. Implemente uma superclasse `Animal`, e depois três subclasses: `Mamíferos`, `Peixes` e `Aves`. Depois crie um programa de testes que lhe permita criar instâncias dessas classes.
2. Modifique as classes do exercício anterior, implementando um método que imprime na tela o som de cada animal criado a partir dessas classes. Por exemplo, um cachorro deve latir, uma ave deve gralhar e um peixe não deve fazer barulho.
3. Altere o programa de testes do primeiro exercício para que ele crie aleatoriamente 10 animais, e depois imprima na tela o som de cada um deles, identificando se é um mamífero, um peixe ou uma ave.
4. Aumente o número de classes de seu exercício, especializando ainda mais os tipos de animais representáveis por elas. Por exemplo, inclua uma classe `Cachorro`, subclasse de `Mamífero`, ou uma classe `Acara`, subclasse de `Peixe`. Depois vá especializando ainda mais, criando as classes `Poodle` e `AcaraBandeira`.
5. Desenhe em um papel a hierarquia entre as classes que você criou no exercício anterior.

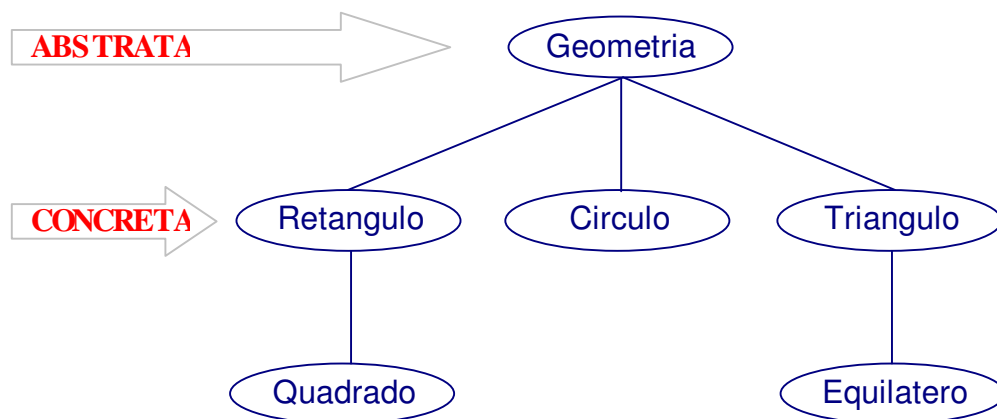


Classes abstratas e interfaces

Uma das contribuições maiores da Orientação a Objetos foi a estratificação funcional da mão de obra envolvida no desenvolvimento de um programa de computador. Hoje em dia os programadores não mais concebem e implementam seus códigos a partir de sua livre intuição - são orientados por sofisticados mecanismos de análise e modelagem de sistemas que freqüentemente geram esboços (esqueletos) de programas. Esses códigos incompletos representam a visão abstrata do sistema e dependendo do padrão de projeto utilizado podem ser fortemente baseados em interfaces.

Classes abstratas e concretas

Quando descrevemos uma hierarquia de classes, nem todas as classes são previstas de serem instanciadas. Por exemplo, observe a hierarquia de classes das figuras geométricas (vide a 2ª lista de exercícios):



Note que, apesar de Geometria ser uma classe, não faz muito sentido imaginar um "objeto geometria". No mundo real, sempre identificamos uma forma geométrica através de uma subclasse de Geometria: um quadrado, um triângulo, etc. Por isso dizemos que a classe Geometria é abstrata. De fato, o mais correto é dizer superclasse abstrata, pois a existência de objetos dessa classe sempre dependerá da implementação de alguma subclasse (concreta).

Uma **superclasse abstrata** possui as seguintes características:

- ?? Não pode ser instanciada, mesmo que possua construtor (algumas linguagens, como Java, permitem a implementação de construtores em classes abstratas, mas não faz sentido, pois esses construtores não serão acessíveis).

- ?? Pode possuir membros abstratos, ou seja, métodos cujo comportamento só será definido em suas subclasses.
- ?? Pode possuir membros concretos, ou seja, métodos comuns - com o comportamento já previsto na própria classe abstrata.

☞ métodos abstratos só podem ser implementados em classes abstratas. Por outro lado, métodos concretos também podem ser implementados em classes abstratas.

O motivo de implementarmos uma superclasse abstrata é permitirmos a representação correta da relação e hierarquia entre outras classes: concretas e/ou abstratas. No exemplo acima, se retirarmos a classe Geometria do diagrama, as demais classes continuarão a existir, mas não terão uma relação definida entre elas. Ou seja, poderemos construir objetos Quadrados, Triângulos, etc., mas não teremos tratar esses objetos como pertencentes a uma mesma superclasse.

O modificador **abstract**

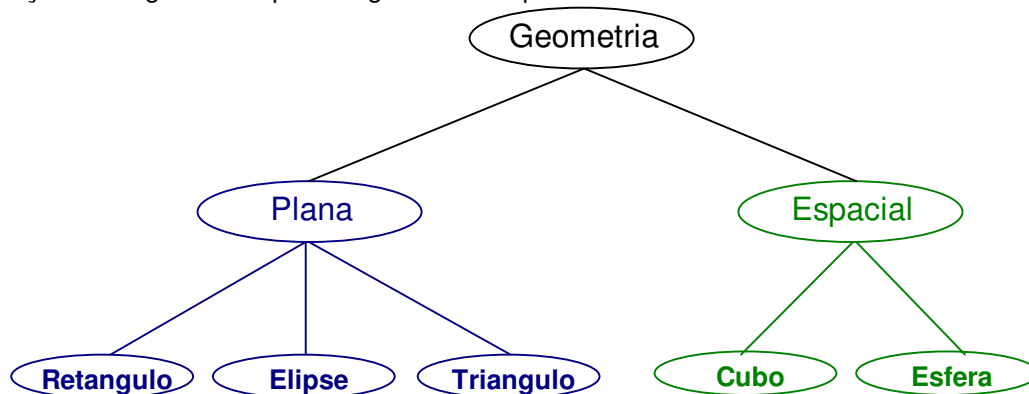
Em Java, usa-se o modificador **abstract** para definir uma classe abstrata:

```
/** Superclasse abstrata. */
abstract class Geometria
{
    abstract public double area();
    public abstract double perimetro();
}
```

Note que os métodos da classe Geometria foram declarados também abstratos. Isso significa que **as subclasses de Geometria deverão implementar esses métodos ou então serem igualmente abstratas**.

☞ não existe uma ordem rígida entre os modificadores de uma classe ou método. No exemplo acima, por exemplo, a ordem dos modificadores **public** e **abstract** foi digitada propositalmente diferente nos dois métodos. Isso foi apenas para exemplificar a liberdade que existe na ordem dos modificadores. Entretanto, é fundamental que o programador escolha uma única ordem de aplicar os modificadores, tornando o código mais elegante e legível.

Existem subclasses abstratas ? sim, existem. Em alguns casos, subclasses de uma superclasse abstrata serão também abstratas. Por exemplo, imagine que a hierarquia Geometria vista anteriormente seja uma generalização entre geometria plana e geometria espacial:



Nessa nova hierarquia, as classes Plana e Espacial também deverão ser abstratas, apesar de serem subclasses da classe Geometria.

* O aluno deve exercitar bastante a idéia de generalização e especialização proposta pela hierarquia de classes. Essa é a idéia central do paradigma de Orientação a Objetos e é inspirada no que se acredita ser a base do pensamento humano.

Restrição de herança pelo modificador **final**

Nas seções anteriores, vimos o uso do modificador final para definirmos membros variáveis com valores constantes. Além de definir valores constantes, o modificador **final** pode ser usado para restringir a herança de classes e métodos:

```
/** Classe final – que não pode ser estendida. */
final class Quadrado extends Retângulo
{
    // O construtor Quadrado é implicitamente final
    Quadrado(double lado)
    {
        super(lado, lado);
    }
}
```

O exemplo acima define a classe Quadrado como final, ou seja, nenhuma outra classe poderá estender a classe Quadrado. O construtor da classe não foi implementado com o modificador final. Apesar disso, o construtor é final porque todos os métodos de uma classe final são implicitamente final. Também podemos ter métodos final, ou seja, uma classe normal possui um dos seus métodos modificados como final. Nesse caso, essa classe pode ser estendida, mas a subclasse não poderá sobrecarregar esse método final.

As referências **this** e **super**

Até aqui acessamos objetos através de referências explicitamente criadas no código dos programas. Entretanto, quando um objeto é instanciado na memória, são criadas duas referências implícitas: super e this. A referência super diz respeito à superclasse de um objeto e a referência this referencia o próprio objeto.

Para referenciarmos a superclasse de um objeto usamos a palavra reservada **super**. Note no exemplo anterior, que o construtor da classe Quadrado chama um método **super(lado, lado)**;. Isso significa que o construtor da classe Quadrado está chamando o construtor de sua superclasse Retângulo, e passando os argumentos exigidos por esse construtor.

Outro exemplo de utilização da referência super é com o operador ponto:

```
/** Classe final – que não pode ser estendida. */
final class Quadrado extends Retângulo
```



```

{
    Quadrado(double lado)
    {
        // Chamando o construtor da superclasse
        super(lado, lado);
    }

    public double area ()
    {
        // Chamando um método da superclasse com o uso do operador ponto
        return super.area();
    }
}

```

O uso da referência **this** é justificado para definir o contexto de uma variável e também para permitir sobrecarga de construtores:

```

/** Exemplo inspirado na hierarquia de classes apresentada na aula 11 */
class Professor extends Funcionario
{
    String nome;
    double salário;

    // Semi default
    Professor ()
    {
        // Acessando o construtor principal, passando parâmetros padrão
        this("", 0.0);
    }

    // Construtor com argumentos
    Professor (String nome, double salario)
    {
        // A referência this é usada para distinguir o escopo das variáveis
        this.nome = nome;
        this.salario = salario;
    }
}

```

Exercícios

- h. Desenhe uma hierarquia de classes capaz de representar os animais de um zoológico, considerando as seguintes restrições:
 1. O zoológico deve ter pelo menos três zonas diferentes, por exemplo: zona dos felinos, das aves, dos répteis, etc.
 2. Cada seção deve ter pelo menos três tipos diferentes de animais.

3. Defina uma hierarquia de classes que inclua métodos relativos à alimentação dos animais, à locomoção, etc. Que permita a modelagem fiel das características de cada animal.
 - i. Implemente as classes da hierarquia acima, incluindo o uso dos conceitos de classe abstrata, e as referências `this` e `super`.
 - j. Implemente um programa de testes, que lhe permita criar um zoológico a partir da instanciação dos vários animais presentes na hierarquia anterior.



Tratamento de exceções

Quando usamos um software, esperamos que ele seja tolerante às falhas – pelo menos às falhas mais comuns relativas à sua função. Quando um programador identifica uma falha possível de ocorrer durante a execução de seus programas, ele pode codificar estratégias de tolerância e/ou correção desta falha, em um processo conhecido como tratamento de exceções.

O que são exceções ?

Uma exceção é um evento ou valor ilegal, porém previsto na modelagem de um sistema. Por exemplo, quando criamos um programa que pede ao usuário a digitação de um valor inteiro e esse usuário digita um valor fracionário, dizemos que o valor digitado pelo usuário irá **gerar uma exceção**. Entretanto, essa exceção pode, e deve, ser previsto na implementação desse programa.

Diferença entre exceções e erros

A diferença entre exceção e erro é que uma exceção é prevista pelo programador durante a implementação de um sistema, enquanto o erro é algo inesperado, não tratado e que, na maioria das vezes, aborta o sistema.

Exceção:	Erro:
O usuário digitou um valor inválido. (previsível)	O teclado sofreu algum avaria física e parou de enviar sinais ao computador. O usuário não pode mais digitar valores com o teclado. (imprevisível)
Solução: a implementação de um tratamento para a exceção gerada pela digitação de um valor inválido.	Solução: não tem solução, pois um programa comum de computador não pode reparar um periférico de um computador – um teclado em curto, por exemplo. Um erro, caso seja detectado, geralmente causa o encerramento do programa que o detectou.

🔒 os erros são sinalizados pelo sistema operacional no qual a máquina virtual Java está rodando. Ou seja, geralmente, a máquina virtual não permite o acesso de um código Java à origem de um erro (uma das funções do sandbox), tornando a tentativa de tratamento dos erros impossível.

Um exemplo de exceção pode ser visto no fragmento de código abaixo, onde o programa tenta acessar um índice inexistente no array alunos:

```

public class Sistema
{

    // Um cadastro com no máximo cinquenta alunos
    private Aluno[] alunos = new Aluno[50];

    public void relatorio(int indice)
    {
        // Suponha que o argumento usado na chamado desse método
        // tenha sido maior do que 50. Isso geraria uma exceção.
        System.out.println(alunos[indice]);
    }
}

```

```

Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException
    at Sistema.relatorio(Sistema.java:10)

```

No exemplo acima, o método `relatorio()`; contém apenas um comando, o que tornaria possível a prevenção de exceções através de um comando de decisão:

```

public void relatorio(int indice)
{
    // Prevenção de falhas através de comandos de decisão
    if(indice>-1 && indice<50)
    {
        System.out.println(alunos[indice]);
    }
    else
    {
        System.out.println("aluno inexistente");
    }
}

```

Imagine, porém, que um determinado método realize dezenas de operações com os argumentos recebidos. As exceções, caso fossem tratadas via comandos de decisão, acarretariam em um código pouco legível e de difícil manutenção. Esse código seria cheio de decisões que nada teriam a ver com a lógica do método em si, mas apenas serviriam para evitar erros. Uma forma mais elegante e eficiente de fazer esse tratamento é conhecida pelo termo *tratamento de exceções*.

Tratamento de exceções

Quando uma falha ocorre na execução de um programa, o trecho de código onde essa falha ocorreu pode gerar uma exceção. Gerar uma exceção é a sinalização ao processo ativo na máquina virtual de que alguma falha ocorreu. Esse processo então pode "capturar" a execução do programa e, quando possível, tornar a execução do código novamente válido dentro do comportamento previsto para esse programa.

Tratamento de exceções em Java

O tratamento de exceções em Java segue o mesmo estilo de C++, visando um código mais tolerante a falhas. Existem três comandos básicos que permitem esse tratamento de exceções: try, catch e finally. Esses comandos são usados em conjunto, suprimindo o programador de recursos que garantam o desenvolvimento de códigos robustos:

- ?? **try**: é o bloco de comandos que são passíveis de gerarem uma exceção. Quando o programador sabe que um determinado conjunto de instruções pode gerar algum tipo de exceção, ele agrega esses comandos dentro de um bloco try. O código contido dentro de um bloco try é chamado de *código protegido*.
- ?? **catch**: é o bloco de comandos alternativos ao try, ou seja, se os comandos do bloco try gerarem uma exceção então os comandos do bloco catch serão executados no lugar deles. O bloco catch é opcional, mas normalmente é implementado junto a um bloco try.
- ?? **finally**: é um comando utilizado para delimitar um grupo de instruções que será sempre executada ao final de um bloco try-catch, independente de qual dos blocos anteriores foi processado pela máquina virtual.

A lógica dos blocos acima descritos é simples:

1. Tenta executar os comandos do bloco try
2. Se os comandos do try geraram uma exceção, então executa os comandos do bloco catch. Um bloco de comandos catch sempre deve seguir um bloco de comandos try, e o programador deve ter o cuidado de garantir que o bloco catch não irá gerar uma exceção.
3. Independente das exceções geradas na execução dos blocos try ou catch, os comandos do bloco finally serão executados.

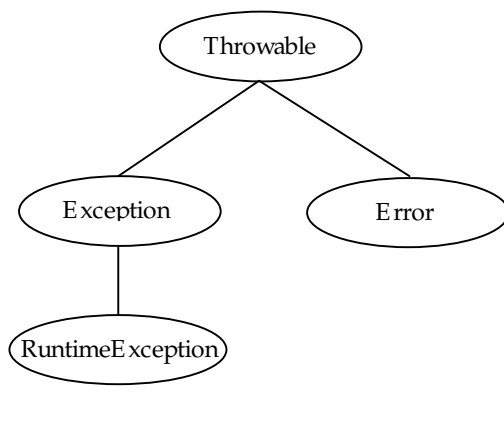
```
public void relatorio(int indice)
{
    // Tratamento de exceções
    try
    {
        // Tenta executar esse comando
        System.out.println(alunos[indice]);
    }
    catch(ArrayIndexOutOfBoundsException erro)
    {
        // Se houver algum erro no bloco anterior,
        // esse comando será executado
        System.out.println("aluno inexistente");
    }
    catch(Exception erro)
    {
        // Pode-se usar um catch para cada tipo de erro
        // possível no bloco try
        System.out.println("exceção inesperada");
    }
    finally
    {
        // Esse comando será executado sempre
        System.out.println("executou finally");
    }
}
```

Um exemplo mais comum do uso do bloco `finally` é quando o código do bloco `try` acessa recursos de entrada e saída do sistema, como uma conexão a um servidor, por exemplo. Nesse caso, se o bloco `try` abrisse uma sessão no servidor e depois gerasse uma exceção antes de fechar a sessão, essa sessão permaneceria aberta até algum `timeout`. No caso, o bloco `finally` teria o comando de encerramento de sessão `http`, garantindo que a sessão seria encerrada, independente de ter havido algum erro no bloco `try` ou não.

🔒 o único caso em que os comandos do bloco `finally` não serão executados é quando um comando `System.exit()`; for executado em um dos blocos `try` ou `catch`.

A hierarquia das exceções

Observe que, no exemplo acima, que a exceção "capturada" no bloco `catch` é uma referência a um objeto da classe `ArrayIndexOutOfBoundsException`. De fato, todas as exceções em Java são objetos instanciados a partir da seguinte hierarquia:



Exemplo: hierarquia da classe `ArrayIndexOutOfBoundsException`.

```

java.lang.Object
|
+--java.lang.Throwable
|
+--java.lang.Exception
|
+--java.lang.RuntimeException
|
+--java.lang.IndexOutOfBoundsException
|
+--java.lang.ArrayIndexOutOfBoundsException
  
```

Onde:

- ?? **Error:** Indica uma falha de difícil recuperação durante a execução de um programa. Um exemplo é a tentativa de instanciamento de uma classe em uma máquina virtual que não possui mais memória livre. Não é previsto que um programa recupere esse tipo de falha, embora seja possível.
- ?? **RuntimeException:** é usado para representar um problema de implementação ou de projeto de um sistema. Ou seja, é usado para indicar condições que jamais deveriam ocorrer durante a execução normal do programa. Pelo fato de que esses erros jamais devem ocorrer em condições normais, o seu tratamento é feito em tempo de execução – a decisão sobre o melhor tratamento é deixado a cargo do usuário. Por exemplo, a tentativa de alocação de um objeto em uma máquina virtual cuja memória está totalmente ocupada retorna um erro de execução. O sistema normalmente avisa ao usuário que não há memória disponível, dando-lhe a chance de fechar outros aplicativos ou optar por outros comandos do programa.
- ?? **Exception:** são erros plausíveis, previsíveis. Erros causados normalmente pelo contexto em que o programa será rodado, por exemplo: arquivo não encontrado, URL mal digitada, etc. Esses erros são normalmente gerados por falhas do usuário. Muitas vezes é complicado imaginar todas

as exceções possíveis quando o correto comportamento de um programa depende do usuário, mas tratar o maior número possível dessas exceções é uma obrigação do bom programador.

☞ se os métodos de um programa tratarem todas as exceções possíveis, dizemos que o programa é robusto. Evidentemente, nem sempre é possível prevenir todas as exceções, mas o grau de robustez de um código tem uma relação direta com a qualidade desse código. O programador deve sempre programar pensando em "quanto mais robusto melhor".

Tratamento pendente de exceções

Caso um método possa gerar uma exceção, esse método deve deixar claro o tipo de ação que deve ser tomada para contornar essa exceção.

Existem duas formas de fazer isso: um bloco try-catch, conforme visto anteriormente ou então a simples sinalização das exceções, deixando o seu tratamento para fora do trecho de código que gerou essa exceção.

Para não misturar o tratamento de exceções com a lógica de um programa, uma técnica muito utilizada é a declaração de métodos com exceções pendentes, ou seja, cujo tratamento deverá ser feito pelo programa que chamar esse método.

Em Java, essa pendência do tratamento de uma exceção é definida pela palavra reservada **throws**:

```
// A palavra reservada throws é seguida pela lista de exceções que podem
// ocorrer na execução do código do método. Esse método só poderá ser
// chamado de dentro de um bloco try-catch, pois a pendência deverá ser
// tratada no código que está chamando esse método.
public void relatorio(int indice) throws ArrayIndexOutOfBoundsException
{
    // A vantagem do tratamento de exceções pendentes é que o código dos
    // métodos fica restrito à lógica do programa.
    System.out.println(alunos[indice]);
}
```

Exceções implementadas pelo programador

Como as exceções em Java são instâncias de alguma subclasse de Throwable, é possível implementar outras classes estendidas de Throwable ou de uma de suas subclasses. Essas subclasses implementadas pelo programador serão igualmente exceções.

Tal qual vimos na [aula 11](#), usaremos a palavra reservada *extends* para implementarmos novas exceções:

```
public class AlunoInexistente extends Exception
{
    // Esse método sobrecarrega o método homônimo da classe Exception
    public String getMessage()
    {
        return "índice do aluno não existe";
    }
}
```

Após compilarmos o código acima, poderemos usar a nova exceção normalmente em outros programas:

```
public void relatorio(int indice)
{
    // Tratamento de exceções
    try
    {
        // Tenta executar esse comando
        System.out.println(alunos[indice]);
    }
    catch(AlunoInexistente erro)
    {
        // Se houver algum erro no bloco anterior,
        // esse comando será executado
        System.out.println(erro.getMessage());
    }
}
```

Sinalizando uma exceção (throw)

O programador pode optar em alguns casos por "forçar" uma exceção, ou seja, sinalizar de uma exceção em determinados casos previamente conhecidos. Essa sinalização programada deve ser feita em métodos definidos com tratamento de exceções pendentes:

```
public void relatorio(int indice) throws AlunoInexistente
{
    if(indice > -1 && indice < 50)
    {
        throw new AlunoInexistente();
    }
    else
    {
        System.out.println(alunos[indice]);
    }
}
```

Throwable.printStackTrace() e Throwable.getMessage()

O aluno deve consultar a documentação da classe `java.lang.Throwable` e identificar os métodos possíveis de serem chamados e/ou sobrecarregados quando instanciamos ou implementamos exceções. Dois desses métodos são particularmente úteis e devem ser utilizados sempre que possível:

- ?? `printStackTrace()`: esse método imprime na tela as informações de depuração da exceção: o tipo de exceção, em que linha, em qual método e em qual classe a exceção foi gerada.
- ?? `getMessage()`: é uma versão simplificada do `printStackTrace()`. Imprime na tela apenas a mensagem padrão da exceção que foi gerada. No caso da implementação de uma nova exceção pelo programador, é possível definir qual mensagem será gerada.

```
public class Teste
{
    static public void main(String[] args)
    {
        new Teste();
    }

    Teste()
    {
        Sistema s = new Sistema();

        // Tratamento de exceções
        try
        {
            s.relatório();
        }
        catch(ArquivoInexistente erro)
        {
            // O método getMessage retorna uma String
            System.out.println(erro.getMessage());
        }
        catch(Exception erro)
        {
            // O método printStackTrace imprime direto na tela
            erro.printStackTrace();
        }
        finally
        {
            // Descartando o objeto Sistema
            s = null;
        }
    }
}
```

Exceções mais comuns

À medida que o aluno for praticando o tratamento de exceções, e principalmente a herança de classe da API Java, ele passará a conhecer a série de exceções mais comuns da linguagem Java. Para facilitar essa familiaridade, algumas dessas exceções são descritas abaixo:

- ?? **ArithmeticException** – problemas com operações numéricas, tipicamente divisão por zero:
`int i = 10 / 0;`

- ?? **NullPointerException** – gerado pela tentativa de acessar um objeto através de uma referência nula, ou seja, antes do objeto ser instanciado.

```
Image[] imagens = new Image[100];
System.out.println(imagens[1]);
```

- ?? **NegativeArraySizeException** – gerado pela tentativa de criar um array de tamanho negativo:

```
Image[] imagens = new Image[-100];
```

- ?? **ArrayIndexOutOfBoundsException** – gerado pela tentativa de acesso a um índice fora dos limites de um array:

```
Image[] imagens = new Image[100];
System.out.println(imagens[200]);
```

- ?? **SecurityException** – gerado pelo sandbox, quando um programa tenta acessar um recurso que a máquina virtual protege. Por exemplo, o gerenciador de segurança da JVM gera uma exceção desse tipo para applets* que tentam:

- Acessar um arquivo local
- Abrir uma conexão socket diferente da conexão a qual o applet está vinculado
- Executar outro programa dentro do ambiente de execução (Runtime).

* veremos applets mais tarde na disciplina.

Exercícios

- k. Reescreva os [exercícios sobre OO](#), usando o tratamento de exceções para garantir a robustez dos programas.

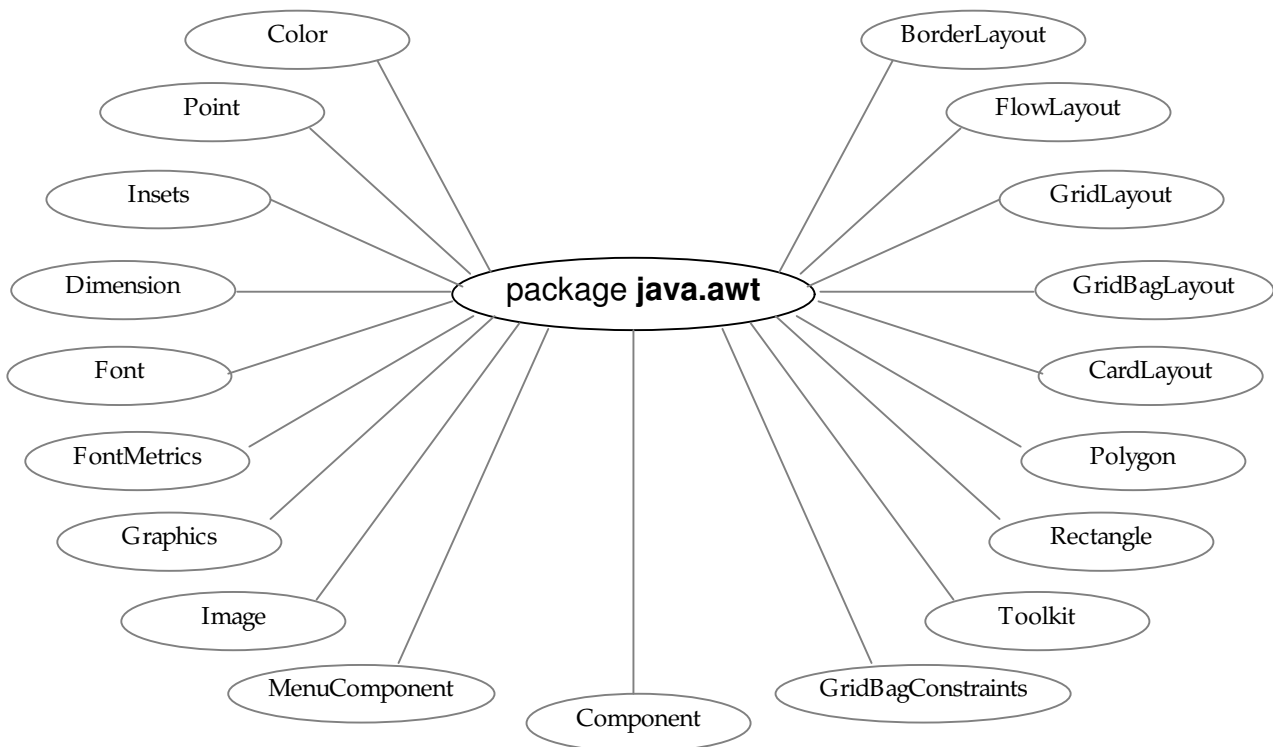
Interface gráfica com o usuário

Controlando o posicionamento de componentes visuais em programas que possuem interface gráfica como o

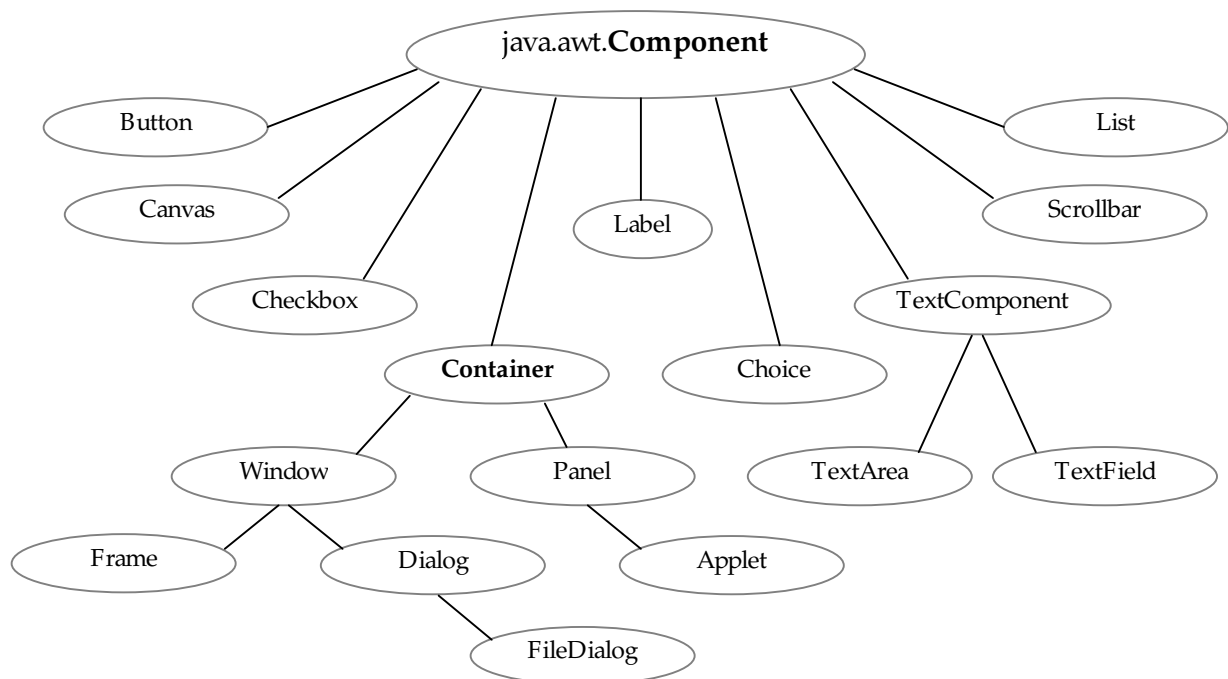
A API Java conta com um pacote chamado AWT – Abstract Window Toolkit, que permite ao desenvolvedor criar interfaces gráficas com o usuário (GUI – Graphical User Interface) a partir da instanciação ou especialização de classes de componentes gráficos. As versões atuais do jdk contém um grande número de componentes gráficos que serão enumerados em sala de aula. Nesta seção, o aluno encontra as diretrizes básicas da construção de interfaces gráficas com Java.

Componentes gráficos – o pacote AWT

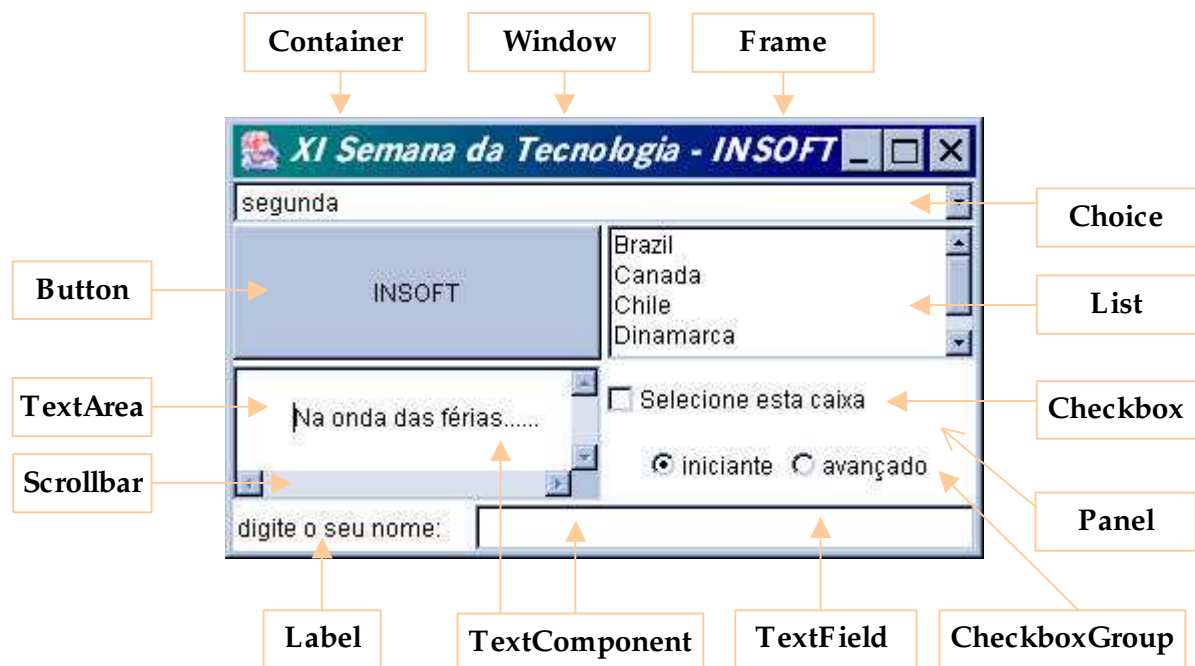
Componentes gráficos são classes que contêm membros associados a informações visuais, como cores, dimensões e bordas. Quando pensamos em componentes gráficos, estamos pensando no usuário – não faz sentido criar um programa com interface gráfica se ele não tiver interação com um ser humano. Aspectos de ergonomia de software são importantes, tais como combinação de cores e fontes utilizadas na interface gráfica. As principais classes do pacote AWT são apresentadas na figura abaixo:



A classe `java.awt.Component`, representada na elipse mais abaixo do diagrama acima, representa um importante papel na construção de interfaces gráficas: é a superclasse dos componentes visuais AWT. O próximo diagrama mostra a hierarquia da classe `Component` e suas especializações.



Os detalhes do uso de cada um desses componentes serão apresentados em aula a partir do exemplo abaixo. Observe que alguns componentes têm duas ou mais classificações. Isto ocorre porque o componente pertence à todas as classes de sua hierarquia, a partir da raiz – lembre-se que todos as classes em Java são especializações de *Object*.



Código do exemplo de componentes AWT

```
import java.awt.*;
import java.awt.event.*;

/**
 * Classe demonstrativa de componente gráficos AWT
 * XI Semana da Tecnologia da Informação - INSOFT
 *
 * @author Felipe Gaúcho
 * @version demo
 */
public class ComponentesAWT extends Frame implements WindowListener {

    /** Botão */
    private Button botao = new Button("INSOFT");

    /** Checkbox - botão de seleção */
    private Checkbox checkbox = new Checkbox("Selecione esta caixa");

    /** Os botões de opção podem ser organizados em grupo */
    private CheckboxGroup grupoDeOpcoes = new CheckboxGroup();

    /**
     * Choice - Caixa de seleção. Observe que o choice será inicializados
     * posteriormente, pois depende de vários comandos para ser construído.
     */
    private Choice choice = null;

    /**
     * List - uma lista de opções, semelhante ao Choice porém com a caixa de
     * opções fixa. Este componente também não pode ser totalmente construído
     * totalmente com uma linha de código.
     */
    private List list = null;

    /**
     * Label - um componente não editável de texto, usado normalmente como
     * título de outros componentes.
     */
    private Label label = new Label("digite o seu nome: ");

    /** TextField - campo de entrada de textos. */
    private TextField nome = new TextField();

    /** TextArea - área para edição de textos. */
    private TextArea texto = new TextArea("");

    /** Panel - um painel que pode conter componentes gráficos */
    private Panel painel1 = new Panel();

    /** Panel - um painel que pode conter componentes gráficos */
    private Panel painel2 = new Panel();
```

```

static public void main(String[] args) {
    ComponentesAWT gui = new ComponentesAWT();
    gui.setSize(400, 300);
    gui.setVisible(true);
    gui.addWindowListener(gui);
}

/** Construtor da interface gráfica */
ComponentesAWT() {
    super("XI Semana da Tecnologia - INSOFT");

    // Inicializando um componente Choice:
    choice = new Choice();
    choice.add("segunda");
    choice.add("terça");
    choice.add("quarta");
    choice.add("quinta");
    choice.add("sexta");

    // Inicializando um componente List: você diz o número de opções
    // que devem aparecer e se o componente aceita seleção múltipla
    list = new List(3, false);
    list.add("Brazil");
    list.add("Canada");
    list.add("Chile");
    list.add("Dinamarca");
    list.add("Espanha");

    painel1.setLayout(new GridLayout(2, 2, 3, 3));
    painel1.add(botao);
    painel1.add(list);
    painel1.add(texto);

    // Exemplo de aninhamento de containers:
    Panel container1 = new Panel();
    container1.setLayout(new FlowLayout());
    container1.add(new Checkbox("iniciante", grupoDeOpcoes, true));
    container1.add(new Checkbox("avançado", grupoDeOpcoes, false));

    Panel container2 = new Panel();
    container2.setLayout(new GridLayout(2, 1));
    container2.add(checkbox);
    container2.add(container1); // ⚡ Veja, um container dentro de outro

    painel1.add(container2);

    painel2.setLayout(new BorderLayout(2, 2));
    painel2.add(label, BorderLayout.WEST);
    painel2.add(nome, BorderLayout.CENTER);

```

```

        this.setLayout(new BorderLayout(2, 2));
        add(choice, BorderLayout.NORTH);
        add(painel2, BorderLayout.SOUTH);
        add(painel1, BorderLayout.CENTER);
    }

    /**
     * Executado quando a janela se torna ativa no contexto do gerenciador de janelas
     * do sistema operacional, ou seja, a janela se tornou o foco de interação com
     * o usuário. Isto acontece no Windows quando uma janela é selecionada pelo
     * usuário.
     */
    public void windowActivated(WindowEvent e) { }

    /** Executado após o fechamento da janela */
    public void windowClosed(WindowEvent e) { }

    /**
     * Executado quando o usuário requisita o fechamento da janela
     */
    public void windowClosing(WindowEvent e) {
        // Esta linha representa a saída padrão de um programa.
        System.exit(0);
    }

    /**
     * Executado quando a janela deixa de ser o foco de interação
     * com o usuário.
     */
    public void windowDeactivated(WindowEvent e) { }

    /**
     * Executado quando a janela recupera o foco de interação
     * com o usuário, normalmente quando o usuário retorna a aplicação de
     * minimizada para normal.
     */
    public void windowDeiconified(WindowEvent e) { }

    /** Executado quando a janela for minimizada */
    public void windowIconified(WindowEvent e) { }

    /** Executado após a abertura da janela */
    public void windowOpened(WindowEvent e) { }
}

```

Gerenciadores de Layout

Uma das principais características da linguagem Java é a sua portabilidade, que dispensa os desenvolvedores de preocupações com aspectos de hardware. Interfaces gráficas, entretanto, possuem dependência dos dispositivos nos quais serão exibidas – a resolução, cores e suporte a eventos são exemplos de aspectos relevantes em um projeto que envolve interface gráfica com o usuário (GUI). Na maioria das linguagens, o programador define previamente a aparência da GUI, incluindo o tamanho e posicionamento dos componentes, e este aspecto é fixo e imutável a menos que haja uma mudança no código.

Imagine um programa codificado para rodar em um monitor com resolução de 800 x 600 pixels sendo executado em um monitor de apenas 640 x 400 pixels. Provavelmente isto acarretará problemas de posicionamento dos componentes ou eventualmente a perda de visibilidade destes componentes. Linguagens compiladas como C++ ou Delphi exigem que o programador saiba de antemão as características de hardware para os quais ele está programando, ou então adotar estratégias de verificação destas características no momento da abertura ou instalação dos programas – o que agrega complexidade ao algoritmo e reduz a portabilidade dos programas.

Em Java não tratamos o posicionamento e dimensionamento dos componentes gráficos rigidamente, mas sim através de processos independentes chamados de *gerenciadores de layout*. Isto permite que um código gerado no sistema operacional Windows, em uma resolução alta, seja executado sem perda de forma ou função em outros sistemas operacionais, como Linux ou Machintosh – ou até mesmo em dispositivos especiais, como Palms ou telefones celulares.

Containers

Interfaces gráficas em Java são construídos a partir da idéia de containers, repositórios de componentes gráficos que possuem um processo gerenciando a disposição e dimensão destes componentes. O interpretador Java requisita ao sistema operacional uma região de vídeo e, a partir desta região, calcula a dimensão que cada componente deve assumir em tempo de execução – esta organização é chamada de *Layout* (disposição em inglês). Este processo de gerenciamento é transparente ao usuário e normalmente só é executado na primeira vez em que o programa entra em execução ou após um redimensionamento da janela. Abaixo exemplificaremos os principais gerenciadores de layout para containers AWT:

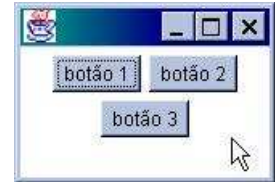
- FlowLayout
- CardLayout
- GridLayout
- BorderLayout
- GridBagLayout

Flow layout

Exemplo de gerenciador de componentes baseado em fluxo:

java.awt.FlowLayout

idéia: adicionar componentes da esquerda para a direita, centralizados em relação à largura do container. Caso os componentes ultrapassem a largura do container, então o gerenciador adiciona os componentes excedentes na linha de baixo, e assim sucessivamente. É como se a interface fosse formada por várias linhas, e existisse uma corrente formada por componentes (uma fila). Cada novo componente adicionado à interface é um novo elo dessa corrente, que fica distribuída sobre as linhas da interface. *Component e1* \rightsquigarrow *Component e2* \rightsquigarrow *Component e3* \rightsquigarrow ...



```
/**
 * @author Felipe Gaúcho
 * @version exemplo.SOO-I
 */
public class Fluxo extends Frame
{
    Fluxo()
    {
        // Ajusta o gerenciador de layouts baseado em corrente de componentes
        setLayout(new FlowLayout());

        // Criando três componentes gráficos (botões)
        Button botao1 = new Button("botão 1");
        Button botao2 = new Button("botão 2");
        Button botao3 = new Button("botão 3");

        // adicionando os componentes
        add(botao1);
        add(botao2);
        add(botao3);

        // Tornando a janela (o Frame) visível
        setSize(200,200);
        setVisible(true);
    }

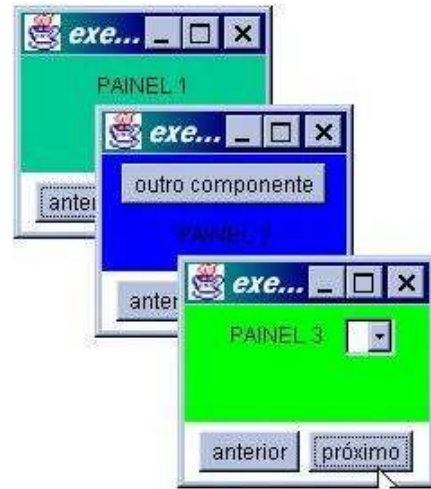
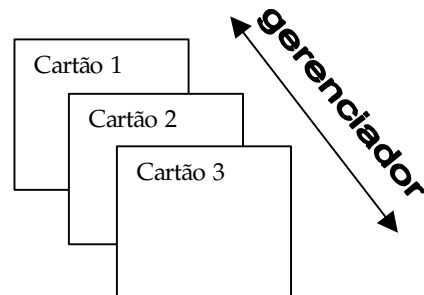
    /** disparador da aplicação */
    static public void main(String[] args)
    {
        new Fluxo();
    }
}
```

CardLayout

Exemplo de gerenciador de cartões:

java.awt.CardLayout

idéia:



```
import java.awt.*;
import java.awt.event.*;
```

```
/** * O gerenciador de cartões atua como se existisse uma pilha de cartões, cada cartão sendo
 * um container. O usuário somente pode observar/manipular o cartão
 * do topo da pilha. Cada cartão tem um nome, definido pelo programador
 * Para se trocar de cartão, usa-se o comando 'show()', conforme mostra o código abaixo.
 *
 * @author Felipe Gaúcho
 * @version exemplo.SOO-I
 */
```

```
public class Cartao extends Frame
{
    private String[] paineis = {"first", "second", "third"};
    int painelAtual = 0;
    private Button proximo = new Button("próximo");
    private Button anterior = new Button("anterior");
    private CardLayout gerenciador = new CardLayout();
    private Panel painelCentral = new Panel();

    Cartao()
    {
        // chamando o construtor da superclasse Frame
        super("exemplo de CardLayout");

        painelCentral.setLayout(gerenciador);

        Panel painel1 = new Panel();
        Label textoDoPainel1 = new Label("PAINEL 1", Label.CENTER);
        painel1.setBackground(new Color(0, 200, 148));
        painel1.add(textoDoPainel1);
```

```
Panel painel2 = new Panel();
Label textoDoPainel2 = new Label("PAINEL 2", Label.CENTER);
painel2.setBackground(Color.blue);
painel2.add(textoDoPainel2);
painel2.add(new Button("outro componente"));
painel2.add(textoDoPainel2);
```

```
Panel painel3 = new Panel();
Label textoDoPainel3 = new Label("PAINEL 3", Label.CENTER);
painel3.setBackground(Color.green);
painel3.add(textoDoPainel3);
painel3.add(new Choice());
```

```
painelCentral.add(painel1, "first");
painelCentral.add(painel2, "second");
painelCentral.add(painel3, "third");
```

```
Panel controles = new Panel();
controles.add(anterior);
controles.add(proximo);
```

```
setLayout(new BorderLayout());
add(painelCentral, BorderLayout.CENTER);
add(controles, BorderLayout.SOUTH);
```

// Programação inline é altamente desaconselhável. Só foi usada aqui
 // para o exemplo não ficar muito grande. O ideal é declarar uma classe
 // que implemente o controlador de eventos.

```
proximo.addActionListener(
    new ActionListener()
    {
        public void actionPerformed(ActionEvent evento)
        {
            mostrarProximoPainel();
        }
    }
);
```

// Programação inline é altamente desaconselhável. Só foi usada aqui
 // para o exemplo não ficar muito grande. O ideal é declarar uma classe
 // que implemente o controlador de eventos.

```
anterior.addActionListener(
    new ActionListener()
    {
        public void actionPerformed(ActionEvent evento)
        {
            mostrarPainelAnterior();
        }
    }
);
```

// Tornando a janela (o Frame) visível

```

        setSize(300, 275);
        setVisible(true);
    }

    public void mostrarProximoPainel()
    {
        if(painelAtual < 2)
        {
            painelAtual++;
            gerenciador.show(painelCentral, paineis[painelAtual]);
        }
    }

    public void mostrarPainelAnterior()
    {
        if(painelAtual > 0)
        {
            painelAtual--;
            gerenciador.show(painelCentral, paineis[painelAtual]);
        }
    }

    /** disparador da aplicação */
    static public void main(String[] args)
    {
        new Cartao();
    }
}

```

BorderLayout

Exemplo de gerenciador de componentes baseado em moldura:

java.awt.BorderLayout

idéia: considerar a interface como uma moldura, dividida em cinco partes:

uma borda superior, uma borda inferior, uma borda esquerda, uma borda direita e uma área central.

A área central prevalece sobre as demais quando esta "moldura" for redimensionada, ou seja, o componente que está no centro da interface é redimensionado em igual proporção ao redimensionamento do container enquanto os demais componentes apenas preenchem os espaços que forem adicionados em suas respectivas bordas.



```
/**
 * Observe que o componente que está no centro da interface é o "âncora", ou seja
 * este componente ocupará sempre a maior parte da interface. Os demais serão também
 * redimensionados, mas sempre respeitando o espaço do componente central.
 *
 * @author Felipe Gaúcho
 * @version exemplo.SOO-I
 */
public class Borda extends Frame
{
    Borda()
    {
        // chamando o construtor da superclasse Frame
        super("exemplo de BorderLayout");

        // Ajusta o gerenciador de layouts para "grade"
        // setLayout(new GridLayout(ESPACO_ENTRE_COLUNAS, ESPACO_ENTRE_LINHAS));
        setLayout(new BorderLayout(2, 2));

        // Item central da interface
        TextArea entradaDeTexto = new TextArea("escreva aqui o seu texto...");
        add(entradaDeTexto, BorderLayout.CENTER);

        // Item da borda superior da interface
        Choice escolha = new Choice();
        escolha.addItem("texto jurídico");
        escolha.addItem("poesia");
        escolha.addItem("tese de mestrado");
        add(escolha, BorderLayout.NORTH);

        // Item da borda direita da interface
    }
}
```

```

Label leste = new Label("Leste");
add(leste, BorderLayout.EAST);

// Item da borda esquerda da interface
Label oeste = new Label("Oeste");
add(oeste, BorderLayout.WEST);

// Item da borda inferior da interface
CheckboxGroup grupoDeSeletores = new CheckboxGroup();
Checkbox opcao1 = new Checkbox("itálico", grupoDeSeletores, false);
Checkbox opcao2 = new Checkbox("negrito", grupoDeSeletores, false);
Checkbox opcao3 = new Checkbox("normal", grupoDeSeletores, true);
// Alterando a fonte de componentes AWT
opcao1.setFont(new Font("Roman", Font.ITALIC, 14));
opcao2.setFont(new Font("Roman", Font.BOLD, 14));
opcao3.setFont(new Font("Roman", Font.PLAIN, 14));

Panel painelInferior = new Panel();
painelInferior.add(opcao1);
painelInferior.add(opcao2);
painelInferior.add(opcao3);

add(painelInferior, BorderLayout.SOUTH);

// Tornando a janela (o Frame) visível
setSize(300, 275);
setVisible(true);
}

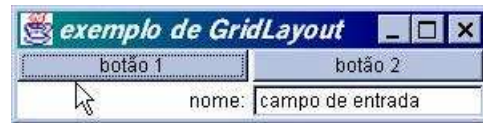
/** disparador da aplicação */
static public void main(String[] args)
{
    new Borda();
}
}

```

GridLayout

Exemplo de grade de componentes:

java.awt.GridLayout



idéia: criar uma grade na qual os componentes são adicionados, da esquerda para a direita e de cima para baixo.

?? Todos os componentes da grade são redimensionados junto com as células.

?? Todas as células possuem o mesmo tamanho.

```
/**
 * Observe que as células do layout em forma de grade ocupam sempre o
 * mesmo espaço, e os componentes são redimensionados junto com as células
 *
 * @author Felipe Gaúcho
 * @version exemplo.SOO-I
 */
public class Grade extends Frame
{
    Grade()
    {
        // chamando o construtor da superclasse Frame
        super("exemplo &nbsp;de &nbsp;GridLayout");

        // Ajusta o gerenciador de layouts para "grade"
        // setLayout(new GridLayout(NUMERO_DE_LINHAS, NUMERO_DE_COLUNAS,
        //                           ESPACO_ENTRE_COLUNAS, ESPACO_ENTRE_LINHAS));
        setLayout(new GridLayout(2, 2, 3, 2));

        // Criando três componentes gráficos (botões)
        Button botao1 = new Button("botão &nbsp;1");
        Button botao2 = new Button("botão &nbsp;2");
        Label label1 = new Label("nome:", Label.RIGHT);
        TextField campo = new TextField("campo &nbsp;de &nbsp;entrada");

        // adicionando os componentes
        add(botao1);
        add(botao2);
        add(label1);
        add(campo);

        // Tornando a janela (o Frame) visível
        setSize(300, 75);
        setVisible(true);
    }
    /** disparador da aplicação */
    static public void main(String[] args) {
        new Grade();
    }
}
```

GridBagLayout

Exemplo de gerenciador de grade assimétrica de componentes:

java.awt.GridBagLayout

idéia: segue a mesma idéia adicionar os componentes a uma grade, porém uma grade assimétrica, ou seja, o espaço ocupado pelos componentes não é obrigatoriamente igual.

Para fazer esse agrupamento assimétrico de componentes, o gerenciador GridBagLayout adota um conjunto de restrições a cada componente. Essas restrições são definidas pelo programador em um objeto da classe: [java.awt.GridBagConstraints](#).

O gerenciamento assimétrico de componentes se baseia na idéia que existem as dimensões do espaço disponível para a visualização de um determinado componente e as dimensões desse componente. Esses dois espaços não são necessariamente iguais e, dependendo do caso, o gerenciador deve calcular de que forma o espaço disponível será preenchido pela representação gráfica do componente. O conjunto de restrições reconhecido pelo gerenciador GridBagLayout é enumerado abaixo:

Variáveis de instância da classe java.awt.GridBagConstraints:

- ?? **anchor:** define onde o componente será posicionado dentro da área disponível para ele: GridBagConstraints.CENTER, GridBagConstraints.NORTH, GridBagConstraints.NORTHEAST, GridBagConstraints.EAST, GridBagConstraints.SOUTHEAST, GridBagConstraints.SOUTH, GridBagConstraints.SOUTHWEST, GridBagConstraints.WEST e GridBagConstraints.NORTHWEST.
- ?? **gridx e gridy:** indica a célula onde o componente deverá ser inserido, onde os valores gridx=0 e gridy=0 indicam a linha e coluna dessa célula – lembre-se que o GridBagLayout, mesmo assimétrico, continua sendo um grid e, portanto, utiliza a idéia de células. Use GridBagConstraints.RELATIVE para indicar que o componente deve ser colocado à direita da posição atual ou abaixo da posição atual.
- ?? **gridwidth e gridheight:** indica o número de células em uma linha da grade e quantas linhas serão ocupadas por um componente. O valores padrão para essas variáveis é 1. Use GridBagConstraints.REMAINDER para indicar que o componente será o último de uma linha e GridBagConstraints.RELATIVE para indicar que o componente deve preencher a linha até o próximo e último componente.
- ?? **fill:** usado quando a área de visualização do componente é maior que o tamanho do componente. Nesse caso, o programador pode indicar de que forma o componente deve ser visualizado:
 - GridBagConstraints.NONE – não muda o tamanho do componente (padrão)
 - GridBagConstraints.HORIZONTAL – preenche todo o espaço horizontal disponível com o componente.
 - GridBagConstraints.VERTICAL – preenche todo o espaço vertical disponível com o componente.
 - GridBagConstraints.BOTH – preenche todo o espaço disponível com o componente, horizontal e vertical.
- ?? **ipadx e ipady:** define o espaçamento entre as dimensões previstas de um componente e as dimensões reais, mostradas na tela – uma borda interna e invisível do componente. A largura do componente será, no máximo, a sua largura menos ipadx*2 (uma vez que a borda interna é aplica



aos dois lados do componente). Similarmente, a altura do componente será a sua altura menos $ipady*2$.

?? **insets**: é parecido com o `ipad`, porém define a borda externa do componente, ou seja, o espaçamento entre o componente e o espaço disponível para a sua visualização.

?? **weightx e weighty**: determina o "peso" dos componentes, ou seja, como eles se comportarão durante o redimensionamento do container gerenciado pelo `GridBagLayout`. Quando o container for redimensionado, o gerenciador calcula as novas dimensões dos componentes de acordo com o seu peso. É necessário indicar o peso de pelo menos um componente em cada linha, caso contrário, todos os componentes ficarão centralizados na linha, sem peso.

```
/**
 * Exemplo de GridBagLayout
 * @author Felipe Gaúcho
 * @version exemplo SOO-I
 */
public class GradeAssimetrica extends Frame
{
    /**
     * Método que adiciona componentes na interface
     * @param componente O componente a ser adicionado à interface
     * @param gerenciador O gerenciador de layouts utilizado
     * @param restricoes O objeto com as restrições associadas ao componente
     */
    private void adicionarComponente(Component componente,
                                     GridBagLayout gerenciador, GridBagConstraints restricoes)
    {
        // Registra os restrições do componente no gerenciador de layouts
        gerenciador.setConstraints(componente, restricoes);
        add(componente); // Adiciona o componente
    }

    GradeAssimetrica()
    {
        super("FIC - exemplo de GridbagLayout");

        GridBagLayout gerenciador = new GridBagLayout();
        GridBagConstraints restricoes = new GridBagConstraints();

        // Define o objeto 'gerenciador' como o gerenciador de componentes da Janela
        this.setLayout(gerenciador);

        // GridBagConstraints.BOTH faz o componente ocupar toda a área disponível
        // horizontal e vertical.
        restricoes.fill = GridBagConstraints.BOTH;
        restricoes.weightx = 0.5; // Componente com peso horizontal 0.5x
        adicionarComponente(new Label("nome: "), gerenciador, restricoes);

        // Componente ocupando a linha até o final - REMAINDER = nova linha
        restricoes.gridwidth = GridBagConstraints.REMAINDER;
        restricoes.weightx = 3.0; // Componente com peso horizontal 2x
        adicionarComponente(new TextField( ), gerenciador, restricoes);

        // O RELATIVE faz o componente ocupar a linha até o próximo componente
    }
}
```

```

restricoes.gridwidth = GridBagConstraints.RELATIVE;
restricoes.weightx = 0.0; // Componente com peso padrão - igual aos outros
restricoes.weighty = 3.0;
adicionarComponente(new Button("botão 1"), gerenciador, restricoes);
// depois de um RELATIVE sempre deve vir um REMAINDER
restricoes.gridwidth = GridBagConstraints.REMAINDER;
adicionarComponente(new Button("botão 2"), gerenciador, restricoes);

restricoes.gridwidth = 1; // recupera os padrões de alocação de espaço
restricoes.gridheight = 2; // na grade
restricoes.weightx = 0.0;
restricoes.weighty = 1.0;

Button botao3 = new Button("botão 3");
botao3.setBackground(Color.cyan);
adicionarComponente(botao3, gerenciador, restricoes);
adicionarComponente(new Button("botão 4"), gerenciador, restricoes);
adicionarComponente(new Button("botão 5"), gerenciador, restricoes);
adicionarComponente(new Button("botão 6"), gerenciador, restricoes);
restricoes.gridwidth = GridBagConstraints.REMAINDER;

// Adicionando um container dentro do de uma célula
Panel painel = new Panel();
painel.setLayout(new GridLayout(2,2,2,2));
painel.add(new Label("campo 1"));
painel.add(new TextField());
painel.add(new Label("campo 2"));
painel.add(new TextField());
adicionarComponente(painel, gerenciador, restricoes);

restricoes.weighty = 1.7;
restricoes.fill = GridBagConstraints.BOTH;
Label fic = new Label("FACULDADE INTEGRADA DO CEARÁ", Label.CENTER);
fic.setBackground(Color.black);
fic.setForeground(Color.white);
fic.setFont(new Font("Serif", Font.BOLD, 16));
adicionarComponente(fic, gerenciador, restricoes);

setSize(375,300);
setVisible(true);
}

static public void main(String[] args)
{
    new GradeAssimetrica();
}
}

```




Applets

*Um dos maiores apelos da linguagem Java está na facilidade de implementação de programas executáveis através da Internet. Tais programas rodam em qualquer sistema operacional que possua um navegador web, e são conhecidos como Applets (do inglês **Application Lett**)*

O que é um **Applet** ? (java.awt.Applet)

Um applet é um programa Java que pode ser executado via Internet, através de um browser. A principal diferença entre um applet e os demais programas implementados em Java é a forma como esses programas serão inicializados. Em uma aplicação Java usamos o método `main()` para inicializar a aplicação, enquanto nos applets esse processo de inicialização é um pouco mais complexo.

Devido ao fato de um applet rodar dentro de um browser, ele não pode ser executado diretamente por linha de comando, tal qual uma aplicação Java. Ao invés disso, devemos criar um código HTML que contém informações sobre o código a ser carregado e executado pela máquina virtual contida no browser – os navegadores web mais populares geralmente contém uma máquina virtual embutida em seu código. Quando o navegador reconhece que o código html que ele está interpretando possui uma referência a um Java applet, ele ativa essa máquina virtual para executar esse applet.

 existem alguns detalhes sobre a compatibilidade de applets em navegadores que serão melhores apresentados em sala de aula. A maioria desses detalhes refere-se a questões comerciais polêmicas e de versões de produtos envolvendo a SUN, fabricante do Java, e as fabricantes dos navegadores: Netscape, Microsoft, etc. Ao aluno é importante apenas a informação de que para um applet ser executado por algum navegador web, esse navegador deve ter alguma máquina virtual compatível com Java dentro dele (ou acessível por ele).

Outro detalhe importante a ser observado é que **applets são programas completos** de computador, e **não scripts** a serem executados por um servidor ou interpretados pelo browser, como JavaScript, Asp e scripts interpretados por CGIs. Quando uma página Html que contém um applet é lida pelo browser, todo o código do Applet é carregado na memória da JVM do browser, para daí então ser executado. Quando implementamos um applet devemos lembrar que todo o seu código deverá ser transmitido via Internet para que ele possa rodar, logo, o programador deve ter o cuidado de não criar applets muito grandes.

Restrições de segurança em applets

Pelo fato de serem distribuídos via Internet, os applets representam aplicações potencialmente perigosas para o usuário. Imagine um applet que leia o seu disco rígido e, sempre que encontrar um arquivo contendo senhas, o transmita para o endereço de algum hacker. Esse hacker poderia usar esse arquivo para descobrir suas senhas pessoais e então fazer um grande estrago com isso.

Para prevenir tais problemas, a máquina virtual Java exerce um controle rígido de acesso ao sistema operacional quando executa applets, através da classe `SecurityManager`. Esse controle, realizado

através do já citado modelo de segurança **sandbox**, permite que uma aplicação applet rode em um contexto limitado quanto à sua liberdade de operações.

O quão seguro será a execução de um applet é configurável através dos browsers que, geralmente, **proíbem os seguintes processos**:

- ?? Execução de outros programas via Runtime
- ?? Acesso a arquivos
- ?? Chamada de métodos nativos do sistema operacional
- ?? Abrir uma conexão Socket com outro endereço senão o próprio endereço onde o applet reside.

☞ quando Frames ou caixas de diálogo (Dialog) são abertas a partir de um applet, um rodapé é adicionado a eles, informando o usuário que se trata de uma aplicação via web, insegura. Isso foi feito para evitar que uma aplicação applet "iluda" um usuário leigo a digitar, por exemplo, o número de seu cartão de crédito – que poderia ser então retornado ao servidor de origem do applet.

As restrições de segurança de um applet podem ser alteradas através de arquivos de configuração da máquina virtual em Intranets ou através de um certificado de segurança vinculado ao applet na Internet – chamado de **Applet assinado**. O uso de applets assinados será comentado em aula, mas está fora do escopo do nosso curso. Caso seja do seu interesse o uso de certificados de segurança, procure em sites de busca pelas palavras chave: Java Signed Applet. Existem dezenas de páginas bem detalhadas sobre esse assunto.

O primeiro applet

A criação de um applet é feita a partir do pacote `java.applet.*`; conforme mostra o exemplo abaixo:

```
import java.awt.*;
import java.applet.*;

/**
 * FIC - Faculdade Integrada do Ceará
 * O primeiro applet
 * @author Felipe Gaúcho
 * @version exemplo.applet
 */
public class AloWeb extends Applet
{
    // Uma variável de instância, do tipo String
    private String texto;

    // Método inicial do applet
    public void init ()
    {
        texto = "Alô Internet";
    }

    // Imprime o conteúdo de 'texto' na posição 25, 25 do applet
    public void paint(Graphics g)
```

```

    {
        g.drawString(texto, 25, 25);
    }
}

```

Para se criar um applet, é necessário a declaração de uma classe com a seguinte assinatura:

```

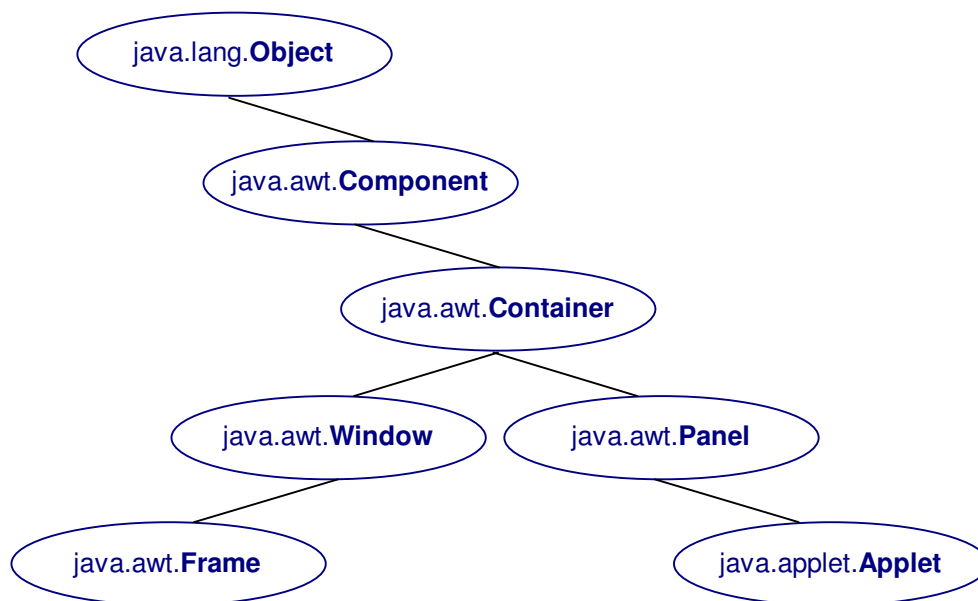
import java.applet.*;

public class <identificador> extends Applet
{
    // ...
}

```

Além disso, as demais considerações sobre programas Java permanecem valendo – o código do exemplo acima deve ser salvo em um arquivo AloWeb.Java e compilado antes de poder ser executado dentro de um browser.

É importante que o aluno pesquise a documentação sobre a classe java.applet.Applet para compreender melhor as características desses componentes, principalmente quanto a sua posição na **hierarquia de classes da API Java**:



Observe que um applet é uma extensão de um painel, portanto o comportamento de uma interface gráfica applet é o mesmo de um aplicativo Java que use painéis como container padrão. O gerenciador de layout padrão para o applet é herdado da classe Panel: FlowLayout.

✎ embora um applet seja uma subclasse de Panel, não faz sentido usa-lo como um componente de um programa Java. Independentemente disso, existem aplicativos híbridos, ou seja, quando um programa é codificado para ser executado tanto como aplicação de console como aplicativo web. As vantagens e desvantagens da implementação de programas híbridos serão discutidas em sala de aula.

Ciclo de vida de um applet

O ciclo de vida de um applet prevê a execução dos métodos abaixo, na ordem em que estão enumerados:

- 1st. Construtor
- 2nd. `public void init()`
- 3rd. `public void start()`
- 4th. `public void paint(Graphics g)`
- 5th. `public void stop()`
- 6th. `public void destroy()`

onde:

Construtor	É um construtor normal de uma aplicação Java, que leva o nome da classe a que pertence e é geralmente usado para inicializar variáveis. Blocos de inicialização também podem ser usados em applets. O programador não deve incluir a lógica do programa no construtor, mas sim no método <code>start</code> explicado abaixo.
<code>init()</code>	Uma vez que o construtor já tenha sido executado, o objeto applet já existe na memória e a máquina virtual do browser executa o método <code>init()</code> . O método <code>init()</code> é usado para inicializar variáveis e, principalmente, para definir o aspecto gráfico do applet: mudar o gerenciador de layout e adicionar componentes. IMPORTANTE: o método <code>init()</code> é chamado apenas na primeira vez que o applet é carregado pelo browser. Se o usuário atualizar a página Html que contém um applet, esse método não será executado novamente.
<code>start()</code>	Esse método é chamado após o <code>init()</code> e é usado para usado para executar a lógica do applet. Esse método será chamado na primeira vez que um applet for carregado, sempre que a página for atualizada ou quando o browser for restaurado após ter sido minimizado. Por exemplo, se você implementar uma animação ou execução de sons em seu applet, essa animação ou som sempre será interrompida quando o browser for minimizado e reiniciará quando a janela do browser for restaurada. Dizemos que quando o método <code>start</code> é chamado o applet se torna vivo .
<code>paint(Graphics g)</code>	Veja a seção "contexto gráfico AWT" abaixo.
<code>stop()</code>	O método <code>stop</code> é chamado sempre que um applet deixar de estar vivo, ou seja, o browser for minimizado ou o usuário trocar de página. O programador pode usar esse método para liberar recursos de multimídia, como sons e animações. Dizemos que quando o método <code>stop</code> é chamado o applet deixa de estar vivo .
<code>destroy()</code>	Método chamado pelo browser automaticamente após o término do método <code>stop()</code> . Serve para indicar à máquina virtual vinculada ao browser que o applet não está mais ativo e pode ser desalocado da memória, bem como todos os recursos associados a ele. Pode ser sobrecarregado pelo programador para incluir uma determinada funcionalidade no encerramento da execução do applet, por exemplo, abrir uma janela de despedida, iniciar

ou suspender threads, etc.

Outros dois métodos importantes na implementação de applets são:

<code>getCodeBase ()</code>	Esse método retorna o endereço onde a classe principal do applet está armazenada.
<code>getDocumentBase ()</code>	Esse método retorna o endereço da página Html que contém o applet

Contexto gráfico AWT

Toda a interface gráfica AWT é baseada em um intrincado mecanismo de exibição de componentes, gerenciado por um processo isolado da aplicação chamado de *Thread AWT*.

Essa thread é ativada em duas situações:

- ?? Quando um componente gráfico for exibido na tela do computador. Isso ocorre quando um programa é ativado e está pronto para ser exibido ao usuário
- ?? Quando um componente gráfico precisar ser atualizado, ou seja, precisar ser redesenhado por alguma razão: mudanças de características gráficas do componente, redimensionamento, etc. Quando isso ocorre, primeiro a thread AWT precisa remover a imagem antiga para depois aplicar a nova imagem do componente.

Mais adiante no curso iremos estudar o conceito de threads. Por hora basta que o estudante saiba que existe um processo externo ao programa sendo executado, controlado automaticamente pela máquina virtual e chamado de thread AWT.

Apesar da imagem dos componentes ser gerada automaticamente pela thread AWT, o programador pode interferir no aspecto visual dos componentes através dos seguintes métodos:

```
?? public void paint(Graphics g)
?? public void repaint()
?? public void update(Graphics g)
```

onde:

`paint (Graphics g)` Sempre que a imagem de um componente gráfico AWT (ou estendido de AWT, como os componentes Swing) for ser exibida na tela de um computador, o método `paint()` será chamado pela máquina virtual. Se o programador não sobrecarregar esse método, a JVM irá usar uma imagem padrão para o componente em questão.

Uma facilidade da classe `java.awt.Graphics`, chamada de `clip rectangle`, é utilizada para que nem toda a área ocupada por um componente precise ser redesenhada a cada atualização desse componente. Por exemplo, quando você tem duas janelas sobrepostas no Windows, e parte da janela que está atrás aparece, se você minimizar a janela que está na frente apenas a parte que não aparecia da janela atrás será redesenhada.

Se o programador desejar alterar a imagem do componente ou desenhar

diretamente sobre ele, deverá sobrepor esse método.

Um applet é um componente AWT que não possui imagem gráfica padrão. Para que alguma imagem apareça quando o applet for executado, o programador deve sobrepor o método `paint(..)`. Caso contrário, o espaço ocupado pelo applet ficará vazio, com a cor do fundo cinza.

`repaint()`

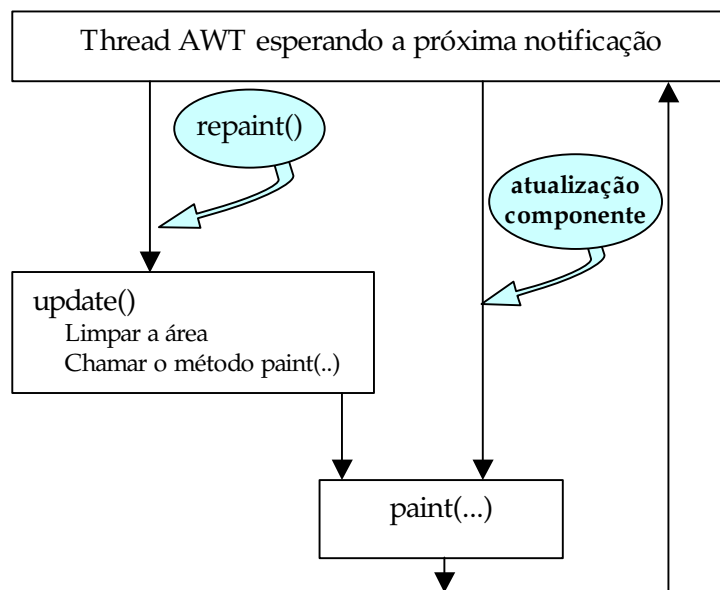
Esse método força a atualização da imagem do componenet, através de uma chamada ao método `update()`. É muito usado para prover uma forma interativa de modificação na imagem do componente, por exemplo, quando um usuário seleciona uma nova imagem de uma lista e essa imagem deve ser visualizada no centro de um applet.

`update(Graphics g)`

Esse método força a thread AWT a chamar um método nativo chamado `update()`, que tem o seguinte comportamento:

- ?? limpa a área ocupada pelo componente gráfico, preenchendo essa área com a cor de fundo do componente.
- ?? Ajusta a cor padrão do contexto gráfico para a cor de frente do componente gráfico.
- ?? Chama o método `paint()` do componente.

A ordem de execução dos métodos descritos acima é apresentada no esquema abaixo:



Aprendendo a usar o appletviewer

O appletviewer é um aplicativo que acompanha o ambiente de desenvolvimento jdk e visa facilitar os testes durante a implementação de applets Java. Ao invés de usar um browser para testar um applet, o

desenvolvedor pode ativar o applet diretamente a partir do aplicativo chamado appletviewer, com a seguinte linha de comando:

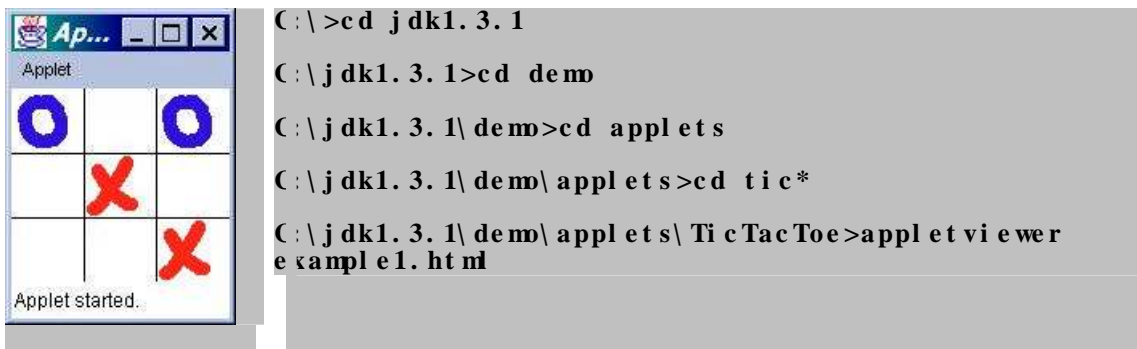
appletviewer [-debug] url

onde:

-debug: permite que o appletviewer mostre no console informações sobre a alocação dos objetos durante a execução do applet, e eventuais erros de execução.

url: o endereço onde o código Html que contém uma referência ao applet está armazenado.

Exemplo:



uma grande fonte de aprendizado para a implementação de applets é o conjunto de exemplos que acompanham a documentação do jdk, pois tem o código fonte disponível para que o aluno aprenda. Procure identificar em que diretório esses exemplos estão copiados e execute-os para ter uma boa idéia de como um applet se comporta. Além disso, procure em sites de busca pelas palavras chave: Java e Applets. Você verá que existem centenas de páginas com demonstrações de applets e dicas de como criar applets mais bonitos ou úteis.

O código HTML de carga do applet

Conforme citamos no início deste texto, para executar um applet via Internet é necessário que a referência a esse applet apareça em uma página Html. Ao interpretar uma página Html que contém um applet, o browser identifica a tag <applet> e ativa a máquina virtual vinculada a ele.

Essa tag <applet> é apresentada abaixo, com as definições opcionais em itálico:

<applet>

archive = listaDeArquivos

code = nome do applet

width = largura em pixels

height = altura em pixels

codebase = endereço onde o applet está armazenado

alt = código para ser exibido em browser sem suporte à applets

name = nome da instância do applet

align = alinhamento em relação ao espaço disponível para exibição

vspace = espaço vertical em pixels

hspace = espaço horizontal em pixels

```

        param name = atributo    value = valor
        alternate HTML
    </ applet >

```

onde:

Archive	Lista de arquivos contendo recursos a serem carregados antes da execução do applet. Normalmente são arquivos contendo imagens, sons ou classes implementados por terceiros. O nome dos arquivos deve ser separado por ','
code	Atributo obrigatório que indica o nome da classe principal do applet. O nome da classe não deve incluir o caminho da classe no servidor – se a classe se encontrar em um caminho diferente do Html que a chamou, esse novo caminho deve ser definido no atributo <code>codebase</code> . Caso o applet pertença a um pacote, o nome do pacote deve ser inserido ao nome da classe, por exemplo: <code>fic.jogo.jogoDaVelha.class</code>
width height	Esses atributos obrigatórios indicam as dimensões que o applet deverá ocupar na página Html, em pixels.
codebase	Especifica a url onde o código se encontra. Se esse atributo não for definido, o browser assume o endereço da página Html como sendo o mesmo da classe principal do applet.
alt	Se o browser suportar a execução de applets mas por algum motivo o applet não puder ser executado, o texto especificado pelo atributo <code>alt</code> será mostrado no lugar do applet.
name	Atributo opcional usado para nomear a instância do applet. Isso é útil para prover a comunicação entre dois mais applets rodando na mesma página Html.
align	Atributo opcional de alinhamento do applet em relação ao espaço disponível para a sua exibição na página Html: <code>left, right, top, texttop, middle, absmiddle, baseline</code> <code>bottom, absbottom</code>
vspace hspace	Atributos opcionais que especificam o número de pixels a serem deixados de espaço entre as bordas do applet e o espaço disponível para a sua exibição.
param name value	São atributos que permitem a passagem de parâmetros ao applet. Usado para configurar o comportamento de um determinado applet, que lê os parâmetros utilizando o método <code>getParameter()</code> ;

Como exemplo de Html que contém um applet, mostramos abaixo o Html responsável pela exibição do Tic-Tac-Toe, o exemplo que acompanha o jdk mostrado anteriormente:

```

<html>
    <head>
        <title>TicTacToe v1.1</title>
    </head>
    <body>

```

```

        <h1>TicTacToe v1.1</h1>
        <hr>
        <applet
            code=TicTacToe.class
            width=120
            height=120
        >
            alt=" o applet não está rodando, verifique suas configurações"
        </applet>
    <hr>
    <a href="TicTacToe.java">The source.</a>
</body>
</html>

```

Lendo parâmetros do Html com um applet

Uma das formas mais comuns de configurar o comportamento de um applet é a definição de parâmetros no código Html. Por exemplo, abaixo temos uma applet que mostra uma figura carregada a partir da url onde o applet está armazenado. Observe que o nome da figura a ser exibida pelo applet aparece como um parâmetro no Html e não no próprio código do applet, Isso é útil, pois se quisermos mudar o nome da figura, ou não soubermos de antemão o nome da figura, poderemos mudar apenas o código Html, sem a necessidade de modificar e compilar o código do applet.

```

<html>
    <head>
        <title>Mostrando uma imagem</title>
    </head>
    <body>
        <applet
            code=Figura.class
            width=200
            height=200
        >
            <param name=imagem value=foto.gif>
        </applet>
    </body>
</html>

```

```

-----

import java.awt.*;
import java.applet.*;
import java.net.*;

/**
 * Exemplo de leitura de parâmetro com applets
 * @author Felipe Gaúcho
 */
public class Figura extends Applet
{

```

```

Image figura;

// Aqui a figura será carregada
public void init()
{
    URL endereço = getDocumentBase();
    String nomeDaFigura = getParameter("imagem");
    figura = getImage(endereço, nomeDaFigura);
}

// Aqui a figura será desenhada na tela
public void paint(Graphics g)
{
    g.drawImage(figura, 0, 0, this);
}
}

```

Os parâmetros encontrados em páginas Html são sempre do tipo String, mas você pode convertê-los usando os método das classes wrapper, por exemplo:

```
int velocidade = Integer.parseInt(getParameter("velocidade"));
```

Manipulando imagens

Para que um applet use uma imagem gráfica, é necessário que essa imagem seja do tipo *.gif, ou *.jpg. Os aspectos relativos ao uso de imagens e sons serão discutidos em sala de aula, mas o aluno deve ter sempre em mente o fato de que applets são aplicativos carregados via Internet e, dependendo da velocidade de conexão com o servidor, o uso excessivo de imagens ou de imagens muito grandes deve ser evitado.

Abaixo mostramos um método bastante útil, que lê imagens a partir de um determinado endereço da internet. Você pode incluir esse método no código de seus applets. Esse código será melhor explicado em sala de aula.

```

/**
 * Método que carrega imagens na memória, a partir da URL onde o
 * applet está sendo carregado. O uso de um 'mediaTracker' é fundamental para
 * evitar problemas quanto à velocidade de carga de uma imagem. Se você não
 * usar um mediaTracker, o seu código pode tentar usar uma imagem antes dela
 * estar plenamente carregada na memória, o que causaria um erro de execução.
 * @param nomeDaImagem O nome da imagem a ser carregada
 * @return A imagem requisitada ou null caso não haja tal imagem na url do applet
 */
public Image carregarImagem(String nomeDaImagem)
{
    try
    {
        Image imagem = getImage(getDocumentBase(), nomeDaImagem);
        MediaTracker carregador = new MediaTracker(this);
        carregador.addImage(imagem, 0);
        carregador.waitForID(0);
    }
}

```

```
        return imagem;
    }
    catch(Exception erro)
    {
        erro.printStackTrace();
        System.exit(0);
        return null;
    }
}
```

Exercícios

10. Escreva um applet que mostre o seu nome no centro da tela.
11. Escreva o código Html necessário para rodar o applet do exercício 1 e teste o seu applet com o appletviewer.
12. Implemente um applet que desenhe círculos coloridos na tela. O tamanho e a cor dos círculos podem ser aleatórios.
13. Procure na web por exemplos de applets e procure identificar como esses applets foram implementados. Dica: você pode aproveitar a seção de apoio didático como ponto de partida. Boa viagem.



Interfaces gráficas baseadas em behaviorismo – O pacote Swing

Padrões de projeto permitem ao desenvolvedor de software o aumento da produção e a garantia da qualidade de seus programas. Nesta seção, o aluno é apresentado ao pacote Swing – um conjunto de componentes gráficos baseados no padrão MVC.

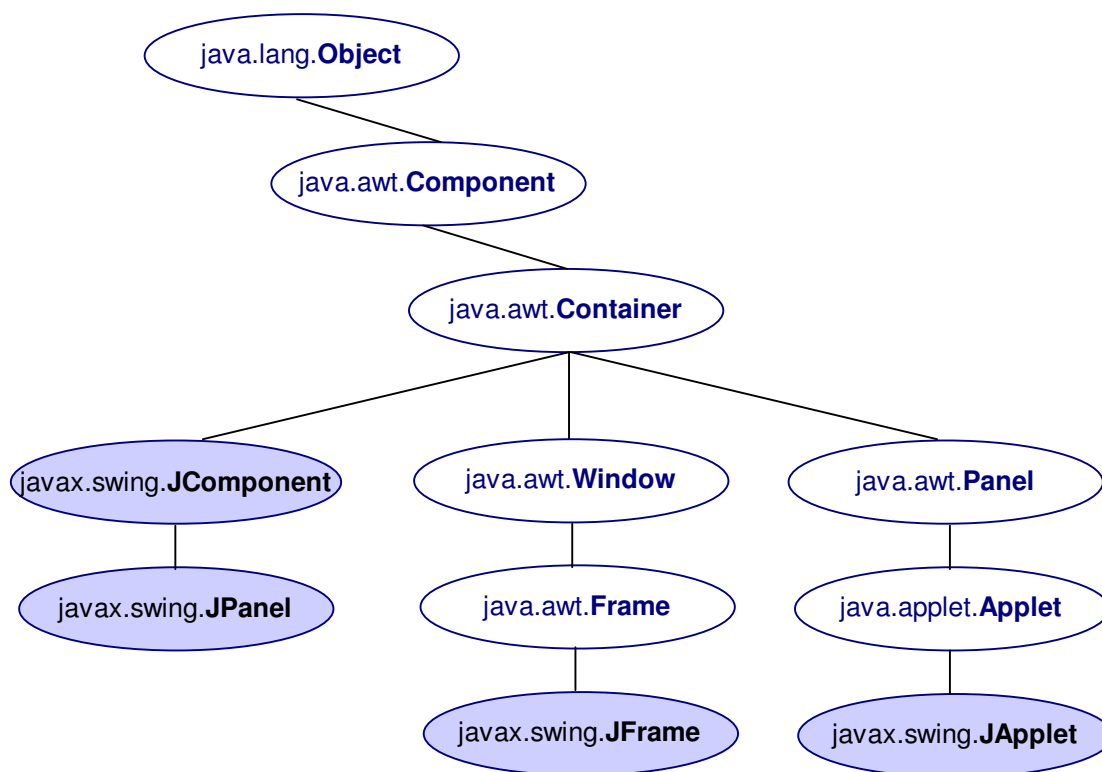
O que é Swing ?

Swing é o nome dado ao pacote de classes desenvolvidas pelo "Projeto Swing" – parte de um contexto maior chamado de JFC – Java Foundation Classes. O Swing consiste em um conjunto de componentes gráficos (extensões dos componentes AWT e novos componentes como representação de árvores e painéis tabulados), que agregam o conceito de **look and feel (L&F)**, ou seja, a capacidade de um mesmo componente assumir aparências diferentes sem a necessidade de mudanças no seu código. Por exemplo, você pode criar uma aplicação Java que se adapte à aparência gráfica do sistema operacional em que for executado – assumindo um aspecto semelhante as demais janelas Windows, Linux, Solaris, Macintosh, etc. Os componentes Swing são totalmente criados em Java (100% pure Java) e foram desenvolvidos no conceito de **interface peso-leve do usuário**. A ideia é codificar apenas a funcionalidade do componente e a sua relação com o modelo de dados ao qual está associado, deixando a sua aparência a cargo do gerenciador de interface do usuário (UI Manager), um novo recurso incorporado às máquinas virtuais a partir da versão 1.3 do ambiente de desenvolvimento Java (jdk1.3.1). Nesta aula não iremos detalhar a manipulação dos componentes Swing. Ao invés disso, focaremos a atenção no conceito de Modelo-Visão-Controle, o padrão de projeto adotado pela maioria dos componentes Swing. Aconselha-se fortemente ao aluno que leia a excelente documentação sobre Swing e analise os exemplos que acompanham o jdk.

Containers Swing

Começaremos a nossa visita ao mundo das interfaces Swing apresentando a classificação de seus principais containers em relação à API Java:

```
?? javax.swing.JFrame  
?? javax.swing.JPanel  
?? javax.swing.JApplet
```



Observe que a grande mudança de classificação em relação aos componentes AWT é que os painéis Swing passaram a ter uma superclasse distinta dos painéis AWT, embora ambos continuem a ser containers. Essa diferenciação foi promovida pela implementação da classe `JComponent`, que foi concebida a dar suporte ao uso de modelos separados da interface, conforme veremos mais adiante no curso. Os demais containers, `JFrame` e `JApplet`, apresentam variações apenas estéticas e, por isso mesmo, aparecem no mesmo ramo de seus pares AWT - `Frame` e `Applet` respectivamente.

Além dos containers tradicionais, apresentados acima, o pacote Swing introduziu uma série de novos containers que, por possuírem funções específicas na construção de interfaces inerentes a determinados tipos de problemas, não serão detalhados aqui:

- ?? `BasicSplitPaneDivider` – usado para interfaces onde os painéis são redimensionáveis.
- ?? `Box` – usado para
- ?? `CellRendererPane`
- ?? `DefaultTreeCellEditor.EditorContainer`

🔑 Observe que os containers Swing tem como superclasse o container AWT, logo, todos os conceitos sobre gerenciamento de componentes em containers AWT seguem valendo.

Introdução a padrões de projeto (design patterns)

No começo dos anos 90, um grupo de pesquisadores conhecidos como "A gangue dos 4", lançou o célebre livro **Design Patterns, Elements of Reusable Object-Oriented Software** (The Gang of Four -

GOF, Addison-Wesley: 1995). Nesta publicação, os autores Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides identificaram padrões no desenvolvimento de sistemas industriais, ou seja, observaram que vários programadores acabavam gerando soluções muito semelhantes para os problemas mais freqüentes no desenvolvimento de programas de computador. Detectaram também que o desenvolvimento de tais projetos exigia um grande esforço de reflexão acerca da ontologia dos problemas, consumindo muito tempo para ser desenvolvido e com a qualidade vinculada à experiência dos analistas responsáveis pelo projeto. Por essa razão, a gangue dos quatro resolveu documentar os padrões mais usados, permitindo aos novos programadores e analistas a redução do tempo necessário para identificar os aspectos centrais dos problemas mais comuns em computação. Ao todo, foram detectados 23 padrões de comportamento na modelagem de sistemas, subdivididos em três grandes grupos:

- ?? **Creational** design patterns – relacionado à instanciação dos objetos ou grupos de objetos. Por exemplo, um sistema pode necessitar limitar o número de instâncias ativas de uma determinada classe ou então permitir que as características de uma classe sejam definidas em tempo de execução. Padrões definidos: *Abstract factory*, *Builder*, *Factory Method*, *Prototype* e *Singleton*.
- ?? **Structural** design patterns – associado à relação entre as classes de um projeto, ou seja, a comunicação entre elas, ordem em que os objetos são instanciados, etc. Padrões definidos: *Adapter*, *Bridge*, *Composite*, *Decorator*, *Facade*, *Flyweight* e *Proxy*.
- ?? **Behavioral** design patterns – estudo do comportamento dos objetos em relação ao seu meio ambiente, tratando isoladamente as características de um objeto e o seu comportamento. Grande parte dos componentes Swing foram implementados usando o padrão behaviorista chamado *Observer*, que iremos descrever com mais detalhes na próxima seção. Padrões definidos: *Chain-of-responsability*, *Command*, *Interpreter*, *Iterator*, *Mediator*, *Memento*, *Observer*, *State*, *Strategy*, *Template* e *Visitor*.

📖 O termo *design pattern*, embora possa ser literalmente traduzido como padrão de projeto, deve ser visto como um jargão tecnológico, ou seja, usado em inglês, mesmo por falantes nativos de outras línguas. Além disso, o estudo de *design patterns* tornou-se referência de qualidade para todo profissional de informática. É indispensável que o aluno leia algum material completo sobre o assunto, que será retomado com mais profundidade na disciplina de Sistemas Orientados a Objetos II.

Programação behaviorista - o paradigma *modelo-visão-controle* (MVC)

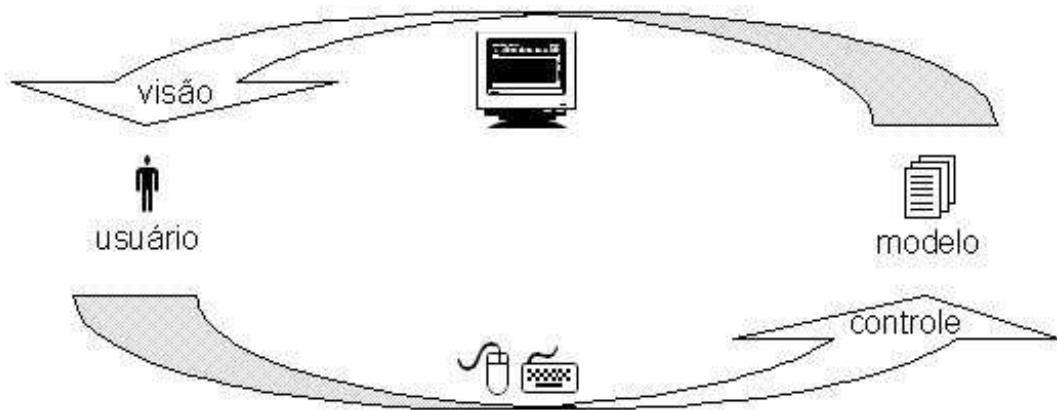
O termo behaviorismo remete à idéia de ação e reação, adotada por alguns estudiosos da mente (psicólogos, psiquiatras, etc.) como teoria de formalização do comportamento humano. A idéia é bem simples: você tem uma entidade que reage aos estímulos do meio ambiente, como por exemplo, uma pessoa gritar quando recebe um choque elétrico, uma bola se mover quando é chutada, uma lâmpada acender quando o seu interruptor é ligado, etc.

No ser humano, esse conceito é um tanto quanto polêmico, pois não se pode afirmar o quanto do nosso comportamento é puramente behaviorista e o quanto é de natureza emocional, ainda pouco conhecida. Já em entidade artificiais, como componentes de programação, o reflexo condicionado é perfeitamente aplicável, pois todos os estímulos e reações de um componente serão previamente conhecidos, não havendo espaço para reações anômalas, de natureza emocional ou caótica.

Exemplo: você pode criar um botão e associar a esse botão um determinado método em seu programa. Você sabe que, sempre que esse botão for pressionado, ele irá executar o método que está associado a ele. **Ação e reação**: o usuário clica sobre o botão e o programa executa algum método, sempre o mesmo método e quantas vezes o usuário quiser – a reação do botão ao estímulo "clique do usuário" é pré-definida e, portanto, consistente em relação ao modelo do sistema.

Tradicionalmente, a funcionalidade de um programa de computador é codificada junto ao seu modelo de dados e, às vezes, junto à sua interface gráfica. A proposta do paradigma controle-modelo-visão (**CMV**) é desassociar esses três aspectos do software, deixando os programas mais portáteis e simples de serem compreendidos. Um sistema behaviorista deve incluir três estruturas codificadas geralmente em separado:

- ?? **Modelo**: é a enumeração das estruturas de dados associadas ao problema, ou seja, uma classe que encapsule um conjunto de dados primitivos ou agregados. O modelo também deve prover métodos que permitam a manipulação de seus dados através da interação com processos computacionais controlados pelo usuário ou por sistemas de computação. Um modelo é dito **observável**, ou seja, outras entidades do sistema podem monitorar o seu comportamento, reagindo a mudanças nos valores de seus dados.
- ?? **Controle**: são processos computacionais que tem acesso aos dados de um modelo, podendo manipular esses dados de forma consistente. Por exemplo: se tivermos um sistema baseado em banco de dados relacional, as tabelas da base de dados serão o modelo do sistema, enquanto os processos de inserção/remoção/atualização dos dados serão o controle do sistema. Podemos dizer informalmente que o controle é forma de comunicação entre o modelo e a visão de um sistema CMV, sendo que, em muitos casos, encontramos o modelo e o controle codificados na mesma classe.
- ?? **Visão**: informa o valor dos dados do modelo a outros processos do sistema ou ao usuário, ou seja, é um processo **observador** do modelo. Em sistemas behavioristas, a visão é normalmente a interface gráfica do usuário



A figura acima representa a idéia geral de um sistema behaviorista, ou seja, o usuário manipula o modelo de dados através de mecanismos de controle e, cada vez que o valor dos dados do modelo for alterado, essa modificação é refletida na interface do usuário (visão). A figura acima mostra um exemplo simples de sistema CMV, mas é importante o aluno lembrar que podemos ter sistemas distribuídos, com múltiplos usuários e diferentes controles e visões para o mesmo modelo de dados. Sistemas multiusuários apresentam problemas complexos de sincronização de acesso ao modelo de dados que serão abordados

mais adiante, na aula sobre threads. Outro detalhe interessante a ser lembrado é o fato de que o modelo pode ser manipulado por outros sistemas de computador. Ou seja, no exemplo acima aparece um usuário como agente ativo no sistema, mas a figura do usuário poderia ser substituída por algum processo automático controlado por software e/ou hardware. Imagine um sistema bancário que aplica dinheiro automaticamente na poupança a cada vez que o saldo for superior a R\$ 1.000,00. O usuário, correntista, fica mexendo em sua conta bancária sem interagir diretamente com as aplicações financeiras. O próprio sistema bancário é que detecta o saldo suficiente e dispara um processo de aplicação bancária ou resgate de aplicação.

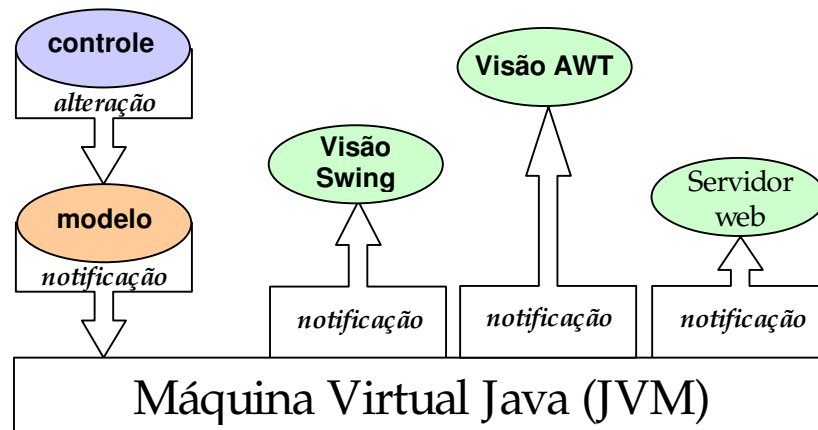
Muitas situações são imagináveis, a maioria delas prevista em design patterns, o que dá uma boa noção sobre a necessidade de um estudo detalhado sobre Design Patterns.

Qual a vantagem de se codificar o modelo, o controle e a visão em classes separadas?

Imagine que você desenvolva um sistema de controle de estoque de uma loja, formado por alguns formulários de entrada de dados, emissão de relatórios e que tenha também o controle fiscal. Agora imagine que o governo altere as normas fiscais vigentes no País. Caso você tenha codificado seu sistema em uma única classe, você terá que alterar os trechos de código relativos ao controle fiscal, procurando esses trechos entre outros códigos que nada tem a ver com o controle fiscal – como, por exemplo, aspectos da interface, dos relatórios, etc. Ou então imagine que o gerente de marketing de sua empresa sugeriu mudar drasticamente o visual do sistema, permitindo aos compradores da loja o acesso direto à consulta de preços através de uma nova interface que agrega informações com marketing. Mesmo que você tenha codificado seu sistema em módulos separados, se esses módulos possuem referências entre eles, ou seja, ponteiros entre as classes, você terá um grande trabalho de reformulação da interface gráfica e, principalmente, de garantir a consistência do sistema – talvez tenha que refazer tudo.

Embora um software possa ser modulado através de técnicas convencionais, a presença de referências entre os módulos impede uma dinâmica eficiente na manutenção desses sistemas. Se você tiver, por exemplo, uma classe de controle fiscal com um ponteiro para a classe que gera a interface gráfica, e essa interface gráfica for renomeada ou removida do sistema, será necessário modificar também a classe de controle fiscal para que o sistema continue consistente. Ou então se você quiser reaproveitar uma boa interface gráfica em um novo modelo, você terá uma complicada tarefa de verificar todos os ponteiros entre as duas classes.

O CMV vem justamente propor uma forma modular de atualização de sistemas, onde a modificação de um aspecto do sistema não interfere nos demais. Por exemplo, em um sistema CMV, podemos ter várias interfaces gráficas diferentes refletindo os dados de um único modelo, ou então diferentes formas de controle atuando sobre esse mesmo modelo – por exemplo: a interface dos usuários só permite consulta enquanto a dos gerentes permite modificação nos valores. Quem faz a comunicação entre os módulos de um sistema behaviorista não são mais os componentes do software, diretamente via referências, mas sim a própria máquina virtual que possui suporte a esse tipo de programação - baseada em eventos chamados de *notificações*. De fato, o comportamento de um sistema behaviorista gira em torno do modelo de dados – sempre que algum formato ou valor for modificado no modelo de dados, o objeto que representa esse modelo notifica a JVM sobre que tipo de alteração ele sofreu. A máquina virtual, por sua vez, se encarrega de repassar a notificação a todos os observadores que estão associados a esse modelo. Note que o modelo não sabe se existem observadores sobre ele e nem o quê esses eventuais observadores fazem com as notificações recebidas, pois não existe nenhuma referência direta entre o modelo (observável) e os observadores. A comunicação entre os módulos de um sistema CMV aparece na figura abaixo:



No diagrama acima, cada elipse representa uma classe diferente, ou seja, a remoção ou modificação de uma das classes do sistema não afeta o comportamento das demais. Um detalhe a ser lembrado é que não faz sentido pensar em observadores sem modelo, porém um modelo sem observadores é perfeitamente aceitável – normalmente em sistemas automáticos, com pouca ou nenhuma interferência do usuário. Outra vantagem do uso de CMV é que as partes que compõem o sistema podem ser desenvolvidas em separado, por pessoas/equipes diferentes e/ou sem a necessidade de sincronia.

🔧 Os módulos de um sistema podem também ser testados em separado, por softwares de teste. O uso de ferramentas de teste vem crescendo em popularidade entre as empresas de tecnologia, bem como a precisão dessas ferramentas. É interessante que o aluno saiba da existência dessas ferramentas e procure informações de como elas são utilizadas. Dica: dê uma olhada em www.junit.org.

Implementando um modelo em Java (Observable)

Um modelo em Java pode ser implementado de duas maneiras:

- ?? Contendo uma lista de observadores (Listeners)
- ?? Estendendo a superclasse `java.util.Observable`

A discussão entre essas diferentes abordagens será dispensada aqui, deixando ao aluno o cargo de aprofundar o estudo sobre implementação de objetos observáveis em Java. A principal diferença entre a lista de observadores e a classe Observable é relacionada à inexistência de herança múltipla em Java. Quando estendemos uma classe à superclasse Observable, estamos impedindo que a classe do modelo seja derivada de outra classe, com funcionalidades eventualmente relevantes ao sistema que está sendo implementado. Mas, na maioria das vezes, um modelo deve apenas representar os dados do sistema e não necessita de outras funcionalidades exceto de ser um objeto observável. Para tal, declaramos a classe que irá representar o modelo no seguinte formato:

```
import java.util.*; // Pacote Java.útil contém a classe Observable

/** Classe observável (modelo) */
public class <Identificador> extends Observable
{
```

```

private int valor = 0;

// Exemplo de método de acesso aos dados do modelo
public void ajustarValor (int valor)
{
    this.valor = valor;
    setChanged(); // ✗ Registro de modificação do modelo
    notifyObservers(); // ✗ Notificação de modificação nos valores do modelo
}

// restante do corpo da classe...
}

```

Implementando visões para um modelo

Uma visão é uma classe Java que implementa a interface `java.util.Observer`, conforme o exemplo abaixo:

```

import java.util.*; // Pacote Java.útil contém a classe Observer

/** Classe observável (modelo) */
public class <Identificador> implements Observer
{
    public void update(Observable modelo, Object component)
    {
        System.out.println("recebeu notificação de " + modelo);
    }

    // restante do corpo da classe...
}

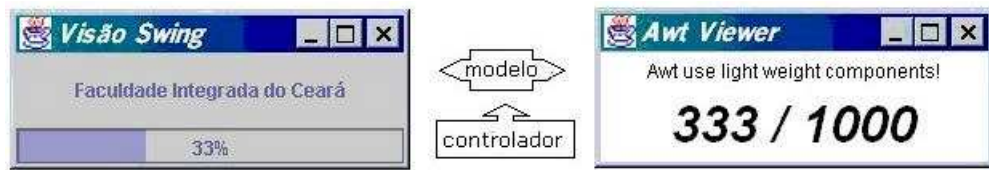
```

Criando o primeiro aplicativo MVC

Mostraremos um exemplo simples, com os seguintes componentes:

- ?? **Modelo:** contém um valor inteiro, limitado entre 0 e um determinado limite superior. Para facilitar o aprendizado de interação entre modelos e visões, iremos implementar a interface `javax.swing.BoundedRangeModel` no modelo. Isso faz com que os métodos de atualização dos valores do modelo sejam sobrecargas de métodos dessa interface:
 - o `setValue(int valor);`
que permite a um controlador modificar o valor atual do modelo.
 - o `getValue();`
que permite a um controlador saber o valor atual do modelo.
- ?? **Controle:** contém uma referência a um modelo do contador e, ao ser ativado, incrementa sistematicamente o valor do modelo até que seja alcançado o seu limite. Para ser de simples entendimento, o controlador possui apenas um método, construtor, que executa a toda a função pré-definida ao controlador.

- ?? **Visão:** implementaremos duas interfaces gráficas, uma Swing e outra AWT, permitindo ao aluno observar o comportamento de diferentes visões associadas a um mesmo modelo:



O código desse exemplo será detalhado em aula, mas para você rodar em casa faça o seguinte:

- ?? Baixe o arquivo zipado contendo os códigos fonte do exemplo
 ?? Descompacte os códigos em algum diretório de teste
 ?? No diretório de testes, digite os seguintes comandos:

- o **javac *.Java**
- o **java CMV**

🔍 Após rodar o exemplo, procure olhar o código fonte e descobrir como ele funciona, como foi implementado. Se você não compreender o funcionamento do CMV, ou parte dele, pergunte ao professor.

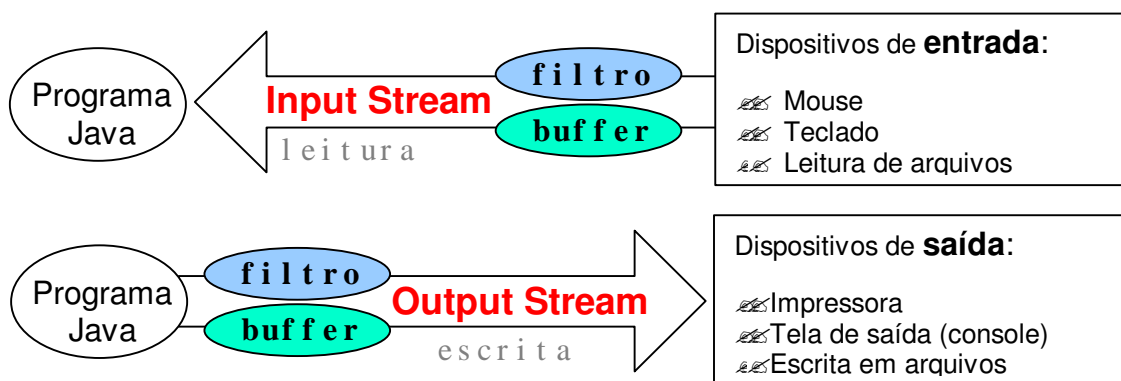


Acesso a dispositivos de entrada e saída - I/O

Dispositivos de entrada e saída são utilizados para persistência ou transferência de dados. Java apresenta uma elegante abordagem. Padrões de projeto permitem ao desenvolvedor de software o aumento da produção e a garantia da qualidade de seus programas. Nesta seção, o aluno é apresentado ao pacote *Swing* – um conjunto de

O que são I/O Streams ?


A linguagem Java não trata dispositivos de entrada e saída de forma específica, ou seja, com classes específicas para cada dispositivo. A invés disso, Java implementa o conceito de Streams, que podem ser vistos como canais por onde trafegam bytes entre um processo computacional e uma origem ou destino de dados: arquivos, impressora, mouse, teclado, vídeo, conexão via Socket com outros programas, etc. A ordem do fluxo de dados, entrada ou saída, é relevante na escolha do Stream a ser utilizado. Além disso, podemos ter Streams com buffer de dados e/ou com filtros de dados, que veremos a seguir. A figura abaixo mostra os dois tipos básicos de Streams utilizados pela linguagem Java:



☞ **Input Stream:** canal utilizado para a leitura de bytes a partir de um dispositivo de entrada.

☞ **Output Streams:** canal utilizado para a escrita de bytes em um dispositivo de saída.

☞ **Filtered Streams:** canal com um filtro acoplado a dispositivos de entrada ou saída, e que permite a escrita ou leitura de tipos de dados ao invés de simples bytes.


 **Buffered Streamas:** canal que permite a leitura ou escrita de dados através de um depósito de bytes, ou seja, o programador pode definir uma quantidade de bytes transferida a cada comando de escrita ou leitura.

Leitura de dados (`java.io.InputStream`)

Os métodos básicos em Streams de entrada são:

```
int read()
int read(byte[])
int read(byte[], int, int)
```

Esses três métodos são usados na leitura dos dados disponíveis em um Stream. Note que o retorno dos métodos é um número inteiro, indicando o byte lido do Stream ou então o número de bytes lidos do Stream. Caso não haja bytes disponíveis para a leitura, ou tenha ocorrido algum erro durante a leitura, o retorno desses métodos será `-1`. O parâmetro `byte[]` que aparece dentro dos dois últimos métodos representa a referência a um array de bytes onde o método deve guardar os bytes lidos do dispositivo de entrada – o número de bytes a ser lido é o tamanho desse array. No último método, os dois parâmetros inteiros representam o intervalo dentro do array onde os bytes devem ser armazenados, e o número de bytes a ser lido é a diferença entre os dois valores.

 para um melhor desempenho de seus programas, procure definir o tamanho do array de bytes usado para a leitura de dados com o máximo tamanho suportado pelo dispositivo de entrada.

```
void close()
```

Método que fecha um Stream, e que deve ser executado sempre que o seu programa não precisar mais ler dados de um dispositivo. A permanência de canais abertos a dispositivos de entrada sem necessidade prejudica o desempenho de seu programa e causa um risco à integridade dos dados desses dispositivos.

```
int available()
```

Método que retorna a quantidade disponível de bytes em um dispositivo de leitura, muito usado na leitura de arquivos cujo tamanho não é previamente conhecido.

```
void skip(long)
```

Esse método descarta um determinado número de bytes do Stream.

```
boolean markSupported()
```

Alguns dispositivos de entrada permitem operações **push back**, ou seja, a utilização de um marcador de posição do primeiro byte disponível no Stream. O método `markSupported()` é utilizado para detectar se o dispositivo ao qual o Stream está associado suporta `push back`, ou seja, retorno verdadeiro. Caso contrário, o retorno do método será falso.

```
void mark(int)
```

```
void reset()
```

Se o dispositivo suportar push back, o programador pode usar o método mark para definir a posição inicial de leitura ou o método reset para restaurar a ordem original dos bytes disponíveis no dispositivo de leitura. O exemplo mais comum de push back é verificado em arquivos de acesso randômico conforme descrito mais adiante nesse texto.

Escrita de dados (java.io.OutputStream)

Os métodos básicos em Streams de saída são:

```
void write(int)
void write(byte[])
void write(byte[], int, int)
```

Esses três métodos são usados na escrita de dados em um dispositivo de saída, com um comportamento semelhante aos métodos de leitura descritos na seção anterior.

```
void close()
```

Método que fecha o acesso a um dispositivo de saída. Sempre que o programa encerrar as operações de escrita em um dispositivo de saída, ele deve fechar o Stream associado a esse dispositivo.

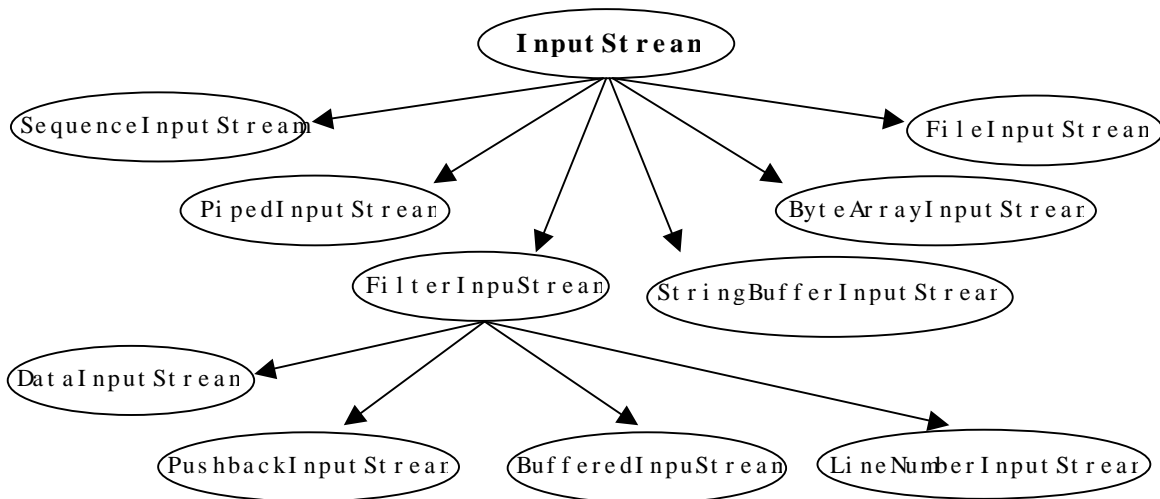
```
void flush()
```

Método que força o Stream a descarregar todos os bytes que já foram enviados ao dispositivo de saída. Algumas vezes, existe uma defasagem entre o momento que um programa chama uma escrita de dados em um dispositivo de saída e o Stream realmente escreve nesse dispositivo. O comando flush força a escrita imediatamente. Quando o método close é chamado sobre um Stream de saída, o método flush é automaticamente executado antes do fechamento do Stream, para garantir que todos os bytes enviados a um dispositivo de saída tenham efetivamente sido escritos nesse dispositivo.

As classes de manipulação de Streams em Java (o pacote

java.io)

A hierarquia de classes do pacote java.io, relativos à Streams de entrada, aparece na figura abaixo. Para Streams de saída vale a mesma hierarquia, só que você deve substituir a palavra Input por Output em todas as classes.



FileInputStream e FileOutputStream

São classes usadas para a manipulação de arquivos, permitindo que você especifique o caminho completo do arquivo como parâmetro do construtor dessas objetos classes. Para que um objeto FileInputStream seja construído com sucesso, o arquivo especificado no construtor deve existir e estar disponível para leitura (não bloqueado). No caso de um FileOutputStream, se o arquivo destino já existir, ele será sobrescrito. A forma de construção dos Streams associados a arquivos aparece abaixo:

```
FileInputStream entrada = new FileInputStream("c:/documentos/teste.txt");
```

```
FileOutputStream saida = new FileOutputStream("saida.txt");
```

o separador de pastas utilizados em endereços de Streams Java segue o padrão UNIX, ou seja, é o "/". Se você quiser adotar o padrão windows, você deve colocar duas barras: "\\" pois somente uma será confundida com a definição de algum caractere especial. Exemplo: "c:/teste/arq.txt" é o mesmo que "c:\\teste\\arq.txt". Por questão de portabilidade, acostume-se a sempre usar o padrão UNIX, com a barra invertida "/".

BufferedInputStream e BufferedOutputStream

Um dos problemas de se usar as classes InputStream e OutputStream, é que elas só permitem a transferência byte a byte dos dados, o que é lento devido ao processo físico de transferência de bytes entre dispositivos de armazenamento e a memória. Para contornar isso, foram definidos Streams com buffer, ou seja, que permitem a leitura ou escrita de quantidades maiores de bytes, normalmente controlados por algum delimitador. Isso faz com que o número de acessos ao dispositivo de armazenamento diminua - um disco, por exemplo - aumentando o desempenho geral do sistema. Em casos onde o número de bytes a ser lido ou escrito é pequeno (<1Mb), essa diferença é desprezível, mas em grandes transferências de dados a diferença é notória. Streams com buffer são normalmente utilizados como superclasses de leitores com buffer, apresentados na próxima seção.


`DataInputStream` e `DataOutputStream`

O raciocínio em bytes é pouco intuitivo, então a API Java provê classes que permitem a manipulação de **tipos primitivos** de dados, tornando a lógica de métodos de leitura e escrita mais clara em relação ao quê está sendo transferido entre um dispositivo de armazenamento e a memória da máquina virtual. Alguns métodos dessas classes aparecem descritos abaixo:

```
public class DataInputStream extends FilterInputStream implements DataInput
byte readByte()
long readLong()
double readDouble()
```

```
public class DataOutputStream extends FilterOutputStream implements DataOutput
void writeByte(byte)
void writeLong(long)
void writeDouble(double)
```

Apesar da Strings serem muitas vezes tratados como tipos primitivos de dados, e existir métodos de leitura e escrita de String nas classes `DataInputStream` e `DataOutputStream`, você deve sempre optar por manipular a transferência de Strings em Streams como serialização de objetos, que veremos na sequência desse texto.

 É fortemente recomendado que você leia as informações sobre conversão de caracteres que aparece na documentação da classe `DataInputStream`.

`PipedInputStream` e `PipedOutputStream`

Essas classes permitem a transferência de bytes entre Threads, ou seja, a origem e o destino da transferência não são dispositivos de armazenamento mas sim processos na memória. Além disso, por serem threads, esses processos tem comportamento diferenciado e, muitas vezes, obrigam a transferência de bytes a ser sincronizada para evitar problemas de inconsistência de dados e/ou dead-lock. A thread de origem deve instanciar um objeto `PipedOutputStream`, enquanto a thread destino deve instanciar um objeto `PipedInputStream`.

Leitores e Escritores de dados em Streams com buffer

As classes descritas anteriormente permitem que um processo computacional leia e escreva bytes em um dispositivo de armazenamento ou em outro processo. Porém, por questões de clareza de código e desempenho costuma-se utilizar essas classes na construção de objetos de leitura e escrita com um buffer, permitindo a transferência de grandes quantidades de dados entre a origem e o destino de um Stream. Tal técnica aparece no exemplo abaixo, originalmente introduzido na aula 6:

```
/**
 * FIC - Faculdade Integrada do Ceará
 * Sistemas Orientados a Objetos I
 * Lendo valores do teclado
 */
// Tem que importar a biblioteca de acesso aos
```

```
// dispositivos de Entrada e Saída (I/O) do Java:
import java.io.*;

public class Teclado
{
    static public void main(String[] args)
    {
        // Tem que usar tratamento de exceções,
        // conforme explicado em aula.
        try
        {
            // Essas duas linhas criam um "leitor com buffer"
            // do dispositivo padrão de entrada do Java:
            // o teclado (System.in).
            InputStreamReader dados = new InputStreamReader(System.in);
            BufferedReader teclado = new BufferedReader(dados);

            System.out.print("digite uma frase: ");
            String frase = teclado.readLine();
            System.out.println("Frase digitada\t" + frase);
        }
        catch(Exception erro)
        {
        }
    }
}
```

Conversão entre bytes e caracteres

Em Java, quando cria-se um leitor ou escritor a partir de um Stream, a conversão entre os bytes transferidos entre a origem e o destino do Stream serão convertidos segundo o padrão de caracteres da plataforma onde o sistema está rodando e a tabela UNICODE. Em lugares onde o padrão é o Latin-1 (Ascii), como no Brasil, a codificação de bytes utilizada será a ISO 8859-1 (para a nossa sorte, o padrão de codificação Java é o mesmo utilizado nos computadores brasileiros, então não há necessidade de preocupações com conversões entre símbolos – a menos que você venha a produzir programas para outros países). Você pode definir qual o padrão de codificação a ser adotado pela máquina virtual, devendo para isso consultar a documentação da ferramenta `native2ascii` que acompanha o jdk. Esse esquema de conversão permite que os programas Java sejam portáveis a computadores de todo mundo, uma vez que os processos da memória sempre trabalharão com o UNICODE. Você pode passar o padrão de codificação no construtor dos Streams, como mostra o exemplo abaixo:

```
InputStreamReader leitor = new InputStreamReader(System.in, "8859_1");
```

O que é UNICODE?

Java utiliza uma tabela, chamada de UNICODE, para representar os símbolos utilizados na maioria das línguas conhecidas no mundo. Essa tabela é usada para converter os símbolos utilizados pela plataforma onde a máquina virtual está rodando e os programas Java – que, na verdade, passam a reconhecer esses símbolos apenas como entradas na tabela UNICODE e não pelo significado real desses símbolos na linguagem padrão dessa plataforma.

Manipulação de arquivos seqüenciais

A manipulação de arquivos, seqüenciais ou randômicos, passa pela construção de objetos da classe `java.io.File`, que provê, além do acesso físico ao arquivo, uma série de métodos que facilitam o controle sobre o conteúdo e as informações básicas desses arquivos, como data da última modificação, tamanho, etc.

Criando objetos do tipo **File**:

```
// construtor de objeto arquivo no diretório corrente
File arquivo = new File("origem.txt");

// construtor com diretório especificado no construtor
File arquivo = new File("/dados/", "origem.txt");

// Uso de variáveis para a identificação do diretório
// e do arquivo a ser aberto
String diretório = "/";
String arquivo = "teste.txt";
File arquivo = new File(diretório, arquivo);
```

A partir de um objeto do tipo `File`, você pode criar Streams de entrada ou saída, conforme mostra o exemplo abaixo:

```
/**
 * Método de leitura de arquivos texto.
 * @param arquivo O objeto associado a um arquivo
 */
static public byte[] carregar(File arquivo)
throws Exception
{
    FileInputStream dispositivoDeEntrada =
        new FileInputStream(arquivo);
    byte[] conteudo = new byte[dispositivoDeEntrada.available()];
    dispositivoDeEntrada.read(conteudo);
    return conteudo;
}

/**
 * Método de gravação em arquivos texto.
 * @param arquivo O objeto associado a um arquivo
 * @param conteúdo O texto a ser escrito no arquivo
 */
static public void salvar(File arquivo, String conteudo)
throws IOException, Exception
{
    FileOutputStream streamDeSaída = new FileOutputStream(arquivo);
    streamDeSaída.write(conteudo.getBytes());
    streamDeSaída.close();
}
```

Apesar das classes `FileInputStream` e `FileOutputStream` permitirem a passagem do nome dos arquivos em seus construtores, é fortemente recomendado que você sempre crie um objeto da classe `File` para

referenciar um arquivo, evitando problemas de dependência à organização dos arquivos no sistema em que a máquina virtual está rodando. E lembre-se de usar sempre a barra invertida '\' como separador de diretórios. É fortemente recomendado que você leia a documentação da classe `java.io.File` acerca das facilidades de manipulação das informações sobre o conteúdo e as características básicas dos arquivos.

Manipulação de arquivos randômicos

Um dos problemas do uso de arquivos seqüenciais é a necessidade de ler ou escrever todo o conteúdo desses arquivos, mesmo que um determinado processo necessite apenas de uma parte de seu conteúdo. Imagine que você tenha um arquivo texto com quinhentos nomes, e precise verificar o centésimo nome – nesse caso, você terá que ler 99 nomes desnecessários até obter a informação que necessita. O ideal, nesse caso, é manipular arquivos como se fossem bancos de dados, ou seja, ter a capacidade de acessar um dado em uma posição específica dentro do arquivo, sem a necessidade de percorrer o resto desse arquivo. Java provê classes para a manipulação de arquivos de acesso randômico, ou seja, arquivos que possuem um ponteiro (**file pointer**) que permite a localização dos dados dentro do arquivo.

Para manipular um arquivo de acesso randômico, constrói-se um objeto da classe `java.io.RandomAccessFile` passando como parâmetro um objeto do tipo `File`:

```
File arquivo = new File("origem.txt");
RandomAccessFile manipulador = new RandomAccessFile(arquivo, "rw");
```

Note que o primeiro argumento do construtor do manipulador de arquivos randômicos é a referência ao arquivo, enquanto o segundo argumento é o tipo de acesso que será utilizado:

r Read Only, significando que o acesso ao arquivo será apenas para leitura
rw Read and Write, significando que o acesso permite a leitura e/ou modificação do conteúdo do arquivo.

Como a classe `RandomAccessFile` implementa as interfaces `DataInput` e `DataOutput`, o acesso aos dados de um arquivo de acesso randômico é feito através dos métodos definidos nessas interfaces, basicamente `read()` e `write()`. Leia a documentação da classe `java.io.RandomAccessFile` para a completa descrição dos métodos de leitura e escrita, bem como os parâmetros desses métodos.

O posicionamento da leitura ou escrita em arquivos randômicos é ajustado ou reconhecido através de dois métodos:

```
void seek(long)
long getFilePointer()
long length()
```

O método `seek` ajusta o ponteiro do arquivo a uma determinada posição, considerando o intervalo padrão do arquivo, que pode ser o tamanho de um tipo de dado ou definido pelo programador. O método `getFilePointer` retorna a posição atual desse ponteiro, e o método `length` retorna o tamanho total do arquivo. **Exemplo** de posicionamento em um arquivo de acesso randômico:

```
// Criando a referência ao arquivo de acesso randômico
File arquivo = new File("vendas.log");
```

```

RandomAccessFile manipulador = new RandomAccessFile(arquivo, "rw");
// Posicionando o file pointer no final do arquivo
manipulador.seek(manipulador.length());
// Todos os comandos write() a partir de agora escreverão dados
// no final do arquivo (append)

```



Arquivos de acesso randômico são uma interessante forma de acesso a dados quando lidamos com quantidades pequenas de informação (<5Mb), ou quando um sistema não justificar a compra e/ou configuração de um banco de dados. Quando o acesso aleatório a registros for muito freqüente ou a quantidade de registros for muito grande, o aconselhável é sempre o uso de um banco de dados. Informações detalhadas do uso de sistemas gerenciadores de banco de dados podem ser encontrados facilmente na Internet, como no seguinte endereço :

<http://developer.java.sun.com/developer/onlineTraining/Database/JDBC20Intro/JDBC20.html>

Serialização de objetos

Um dos aspectos mais elegantes da linguagem Java é a capacidade de serialização de objetos e a transferência desses objetos serializados através de Streams. A serialização de um objeto é a geração de uma seqüência de bytes que representem o valor de seus membros. Uma vez que tenhamos serializado um objeto, podemos gravá-lo em um dispositivo de armazenamento, como um disco rígido ou disquete – nesse caso dizemos que o objeto serializado é persistente.

Um objeto é dito persistente quando ele pode ser salvo em algum dispositivo de armazenamento ou transmitido entre processos computacionais, via rede de computadores.

O que são Grafos de Objetos?

Quando um objeto é serializado, apenas os valores de seus dados são preservados – métodos de classe e construtores não fazem parte da serialização. O processador de serialização usará a definição da classe como esquema de reconstrução do objeto na memória. Quando um dos membros de um objeto serializável for também um objeto, esse objeto será igualmente serializado. Esse processo recursivo de serialização gera um grafo, que representa os valores do objeto a serem persistidos.

Algumas classes de objetos não permitem a serialização de seus objetos, devido à natureza dinâmica desses objetos. Por exemplo, objetos das classes `FileInputStream`, `FileOutputStream` e `Threads` não são serializáveis. Um objeto que contém uma referência a um outro objeto não serializável não pode ser serializado, sob o risco de causar uma exceção do tipo: `java.io.NotSerializableException`

Java provê um modificador especial para permitir que você crie classes serializáveis contendo referências a objetos não serializáveis: a palavra-chave **transient**. Exemplo:

```

// Criando uma classe serializável
public class Teste implements Serializable
{
    // Esse membro será ignorado no momento da serialização
    transient private Thread relógio;
    // Os modificadores private, protected e public não
    // tem influência na serialização
    private String nome = "digital v1.0";
}

```

```

        // .... restante da classe
    }

    // IMPORTANTE: membros estáticos não são serializáveis.

```

Escrevendo objetos serializados em um arquivo - para os exemplos sobre serialização utilizaremos a classe `Java.util.Date`, que é serializável.

```

// O construtor vazio da classe Date assume a data e hora atuais
java.util.Date data = new Date();

File arquivo = new File("data.txt");
FileOutputStream saida = new FileOutputStream(arquivo);
ObjectOutputStream escritor = new ObjectOutputStream(saida);

// O processo de gravação e leitura de dados em arquivos sempre
// deve ser feito prevendo-se o tratamento de exceções
try
{
    escritor.write(data);
    escritor.close();
}
Catch(IOException erro)
{
    erro.printStackTrace();
}

```

Lendo objetos serializados de um arquivo

```

java.util.Date data = null;

File arquivo = new File("data.txt");
FileInputStream entrada = new FileInputStream(arquivo);
ObjectInputStream leitor = new ObjectInputStream(entrada);

// O processo de gravação e leitura de dados em arquivos sempre
// deve ser feito prevendo-se o tratamento de exceções
try
{
    // O casting sempre deve ser feito, porque o retorno
    // de uma leitura via ObjectInputStream é sempre
    // um objeto da superclasse Object, que deve ser convertido
    // conforme o tipo de dado utilizado
    data = (Date)leitor.readObject();
    leitor.close();
    System.out.println("objeto lido: " + data);
}
Catch(IOException erro)
{
    erro.printStackTrace();
}

```

Exercícios

14. Crie um aplicativo, com interface gráfica, que lhe permita ler, criar ou modificar arquivos texto, seqüenciais e randômicos. Dica: dê uma olhada na classe `javax.swing.JFileChooser`.
15. Modifique o seu trabalho 3, permitindo a gravação em disco do jogo como um objeto serializável. Faça com que o usuário possa armazenar o seu jogo em disco para poder continuar depois. Não esqueça de incluir também o método de recuperação desse objeto – a leitura do disco.