



Java básico 1

Paulo Henrique Cayres

Rio de Janeiro
Escola Superior de Redes
2013

Sumário

Lógica de Programação	1
Conceito de Algoritmo	2
Método para a Construção de Algoritmos	2
Instruções	4
Histórico da linguagem de programação Java	6
Características da Linguagem	8
Similaridade com C, C++	8
Compilação	8
Portabilidade	8
Orientação a Objetos	9
Segurança	9
Concorrência	9
Eficiência	9
Suporte para Programação de Sistemas Distribuídos	10
Fundamentos de Programação em Java	11
O que é Java	11
Recursos da Linguagem Java	12
Máquina Virtual Java	12
Coleta de Lixo	13
Segurança de Código	13
Primeiro Programa	14
Alguns cuidados ao escrever e executar programas Java	16
Comentários	16
Variáveis	19
Tipos de Dados	19
Inteiros	20
Ponto Flutuante	20
Tipo Caractere, char	20
Tipo boolean	21
Atribuições e Inicializações	21
Conversões entre Tipos Numéricos	22
Constantes	22
Operadores	23
Tipo String	25
Concatenação	25
Substrings	25
Tamanho	26
Teste de Igualdade	26
Manipulando dados em Java	26
Exercícios de Fixação 4 – Variáveis	27
Atividade 2 – Variáveis e manipulação de dados (Complementar)	28
Controle de Fluxo	29
Escopo de um Bloco de comandos	29
Estruturas de Decisão	30
Comando if	30

Comando switch	31
Estruturas de Repetição	33
Definição	33
Comando while.....	33
Comando do ... while.....	34
Comando for.....	34
Quebra de fluxo	35
Comando Break	35
Comando continue.....	36
Comando return	36
Arrays	39
Exercício de nivelamento 6 – Arrays	39
Definição	39
Tamanho de uma array	41
Cópia de Arrays	42
Arrays como Argumentos	42
Arrays como Valores de Retorno	42
Arrays Multidimensionais	42
Classes Especializadas	45
Wrappers de Objetos	45
Date, DateFormat e Calendar	46
Date	46
DateFormat	47
Calendar.....	48
NumberFormat	48
DecimalFormat	49
Padrão de Caracteres Especiais	49
Math.....	50
Classes e Objetos.....	53
Programação Orientada a Objetos	53
Classes	54
Relações entre Classes.....	54
Objetos.....	55
Objetos como Argumentos	55
Auto referência a um objeto	56
Atributos	56
Métodos	57
Getters e Setters.....	57
Sobrecarga	58
Construtores.....	58
Blocos de Inicialização.....	59
Métodos e Atributos static	59
Pacotes.....	63
Usando pacotes.....	63
Adicionando uma classe em um pacote	65
Herança.....	66
Hierarquia de Heranças	67

Evitar Herança	68
Atividade 3 - Herança.....	68
Polimorfismo	69
Classes Abstratas	70
Ligação	71
Classe Class	72
Reflexão.....	73
Análise de Recursos das Classes	73
Análise de Objetos em Tempo de Execução	73
Superclasse abstrata.....	73
Interfaces	74
Propriedades	76
Classes Internas	77
Collections.....	83
Exercício de nivelamento 9 – Collections	83
Definição	83
O que é o Java Collections Framework?	83
Benefícios do Java Collections Framework	84
Listas	85
Conjunto	86
Mapas	87
Exercício de Fixação 11 – Collections	89
Bibliografia	91

1

Lógica de Programação

Exercício de nivelamento 1 – Lógica de Programação

O que você entende por lógica de programação?

Você poderia descrever a sequência de passos necessários para realizar a troca de um pneu furado de um carro?

A lógica de programação é uma técnica desenvolvida para auxiliar as pessoas que desejam trabalhar com desenvolvimento de programas e sistemas. Ela permite definir e encadear pensamentos numa sequência lógica de passos necessários para o desenvolvimento de uma solução computacional para atingir um determinado objetivo ou solução de um problema (FARRER, 1999).

O resultado da sequência lógica de passos necessários para o desenvolvimento de uma solução computacional servirá de base para a escrita de programas que serão implementados utilizando-se uma linguagem de programação. Por sua vez, os programas serão armazenados nos computadores, equipamentos responsáveis por receber, manipular e armazenar dados, tendo como principal atividade o processamento de dados.

Quando queremos escrever um software para realizar qualquer tipo de processamento de dados, vamos escrever um programa ou vários programas interligados, mas para que o computador consiga compreender e executar esse programa ele deve estar escrito em uma

linguagem entendida pelo computador e pelo desenvolvedor de software, é o que chamamos de linguagem de programação.

As etapas para o desenvolvimento de um programa são: Análise, onde o enunciado do problema será estudado para a definição dos dados de entrada, do processamento e dos dados de saída; Algoritmo, onde ferramentas do tipo descrição narrativa, fluxograma ou português estruturado são utilizadas para descrever o problema com suas soluções; e Codificação, onde o algoritmo é transformado em códigos da linguagem de programação escolhida para se trabalhar (ASCENCIO, 2005).

Conceito de Algoritmo

A seguir serão apresentados alguns conceitos de algoritmos.

- ❑ “Algoritmo é uma sequência de passos que visam atingir um objetivo bem definido” (Forbellone, 1999).
- ❑ “Algoritmo é a descrição de uma sequência de passos que deve ser seguida para a realização de uma tarefa” (Ascêncio, 1999).
- ❑ “Algoritmo é uma sequência finita de instruções ou operações cuja execução, em tempo finito, resolve um problema computacional, qualquer que seja sua instância” (Salvetti, 2001).
- ❑ “Algoritmo são regras formais para obtenção de um resultado ou da solução de um problema, englobando fórmulas de expressões aritméticas” (Manzano, 1997).
- ❑ “Ação é um acontecimento que, a partir de um estado inicial, após um período de tempo finito, produz um estado final previsível e bem definido. Portanto, um algoritmo é a descrição de um conjunto de comandos que, obedecidos, resultam numa sucessão finita de ações” (Farrer, 1999).

Método para a Construção de Algoritmos

Podemos utilizar os passos a seguir para analisar e apresentar uma solução para qualquer tipo de problema que se deseja implementar um solução baseada em um programa de computador.

- ❑ **Análise preliminar:** entenda o problema com a maior precisão possível. Identifique os dados e os resultados desejados.
- ❑ **Solução:** desenvolva um algoritmo para resolver o problema.
- ❑ **Teste de qualidade:** execute o algoritmo desenvolvido com dados para os quais o resultado seja conhecido. O ideal é que o universo dos dados tenha todas as combinações possíveis.
- ❑ **Alteração:** se o resultado do teste de qualidade não for satisfatório, altere o algoritmo e submeta-o a um novo teste de qualidade.

Produto final: algoritmo concluído e testado, pronto para ser traduzido em programa de computador.

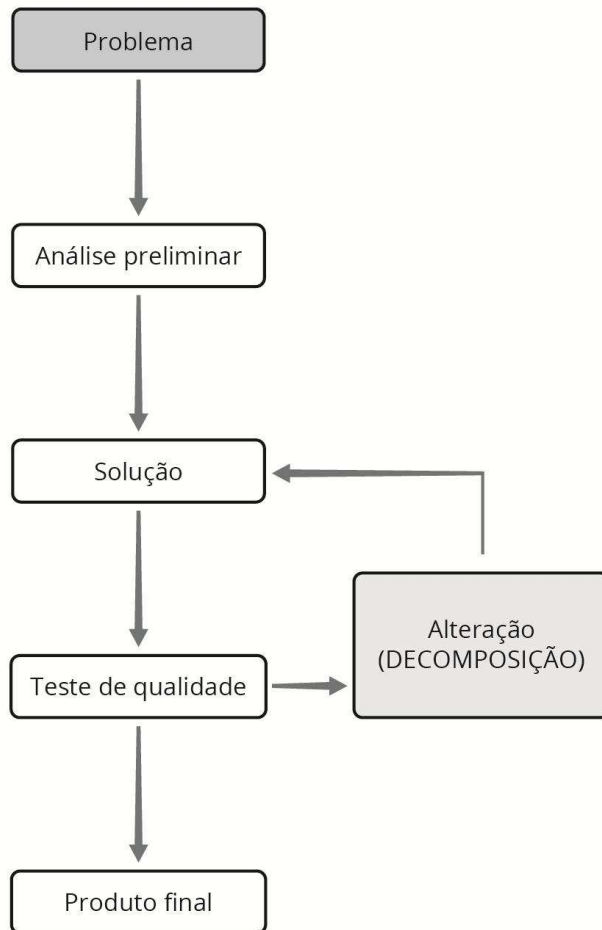


Figura 1.1 – Passos lógicos para desenvolvimento de programas.

Para a construção de qualquer tipo de algoritmo são necessários os passos descritos a seguir:

1. ler atentamente o enunciado, destacando os pontos mais importantes;
2. definir os dados de entrada, ou seja, quais dados serão fornecidos;
3. definir o processamento, ou seja, quais cálculos serão efetuados e quais as restrições para esses cálculos. O processamento é responsável por transformar os dados de entrada em dados de saída;
4. definir os dados de saída, ou seja, quais dados serão gerados depois do processamento;
5. construir o algoritmo;
6. testar o algoritmo realizando simulações.

De posse deste método, nós podemos observar várias situações do dia-a-dia e imaginarmos uma sequência de passos lógicos para solucionarmos, conforme os exemplos descritos a seguir:

■ Soma de dois números inteiros

Passo 1 – ler os dois números

Passo 2 – somar os dois números

Passo 3 – escrever o resultado do somatório dos dois números

■ Sacar dinheiro no caixa eletrônico

Passo 1 – Ir até um caixa eletrônico

Passo 2 – Inserir o cartão no leitor de cartões

Passo 3 – Informar a quantia que deseja sacar

Passo 4 – Retirar a quantia informada caso tenha saldo disponível

Passo 5 – Retirar o cartão

Passo 6 – Sair do caixa eletrônico


Instruções

Na linguagem comum, entende-se por instruções “um conjunto de regras ou normas definidas para a realização ou emprego de algo”. Em informática, instrução é a informação que indica a um computador uma ação elementar a executar (GUIMARÃES, 1994) (FARRER, 1999) (ASCENCIO, 2005).

Convém ressaltar que uma ação isolada não permite realizar o processo por completo, sendo necessário um conjunto de instruções colocadas em ordem sequencial lógica para implementar uma solução computacional adequada para um determinado problema.

Por exemplo, se quisermos trocar o pneu de um carro, precisaremos colocar em prática uma série de instruções: parar o carro, desligar o motor, pegar o pneu reserva, macaco e chave de rodas, etc...

É evidente que essas instruções têm que ser executadas em uma ordem adequada, ou seja, não se pode tirar o pneu furado sem antes erguer o carro com o macaco, por exemplo. Dessa maneira, uma instrução tomada em separado não tem muito sentido, ficando claro que, para obtermos um determinado resultado, precisamos colocar em prática um conjunto de instruções numa ordem lógica correta.

 Instruções são um conjunto de regras ou normas definidas para a realização ou emprego de algo. Em informática, é o que indica a um computador uma ação elementar a executar.

Exercícios de Fixação 1 – Lógica de Programação

Desenvolva algoritmos para dar solução a cada um dos problemas apresentados a seguir:

Calcular e escrever a média de três números inteiros.

Dadas 4 notas de um aluno, informar se ele está aprovado ou reprovado, considerando a média de aprovação igual ou superior a 7.

Calcular e escrever o resultado da multiplicação de três números.

2

Histórico da linguagem de programação Java

A linguagem Java começou em 1991, quando um grupo de engenheiros da Sun, liderados por Patrick Naughton, Sun Fellow e James Gosling, quiseram projetar uma pequena linguagem de programação que pudesse ser usada em equipamentos de consumo como caixas de comutação de TV a cabo. Com esses dispositivos não tinham uma grande capacidade de processamento ou memória, a linguagem tinha de ser pequena para gerar um código bem limitado em tamanho. Além disso, como fabricantes diferentes podiam escolher CPUs diferentes, era importante não ficar preso a uma única arquitetura. O projeto recebeu o nome de Green (HORSTMANN, 2003).

As exigências de um código pequeno e compacto levaram a equipe a ressuscitar o modelo de uma linguagem chamada UCSD Pascal que havia tentado nos primórdios dos PCs e da qual Niklaus Wirth havia sido pioneiro. O que Wirth havia desbravado, que o UCSD Pascal fez comercialmente e que os engenheiros do projeto Green também adoraram foi projetar uma linguagem portátil que gerava código intermediário para uma linguagem hipotética. (Estas são frequentemente chamadas de máquinas virtuais). Esse código intermediário poderia então ser usado em qualquer máquina que tivesse o interpretador correto. O código intermediário gerado nesse modelo é sempre pequeno e os interpretadores de código intermediário podem ser bastante pequenos, de modo que isso resolvia o problema principal que eles tinham.

Já o pessoal da Sun, que tinha experiência com UNIX, baseou sua linguagem no C++ em vez do Pascal. Em particular, eles fizeram a linguagem orientada a objetos em vez de orientada a procedimentos. Porém, como Gosling diz na entrevista, “Desde o início, a linguagem era uma ferramenta, não o fim”. Gosling decidiu chamar essa linguagem de Oak (presumivelmente porque ele gostava da aparência de um carvalho, *oak* em inglês, que ficava bem à frente de sua janela na Sun). O pessoal da Sun, mais tarde descobriu que Oak era o nome de uma linguagem de computador já existente, de modo que trocaram o nome para Java.

Em 1992, o projeto Green apresentava seu primeiro produto, chamado *7. Ele era um controle remoto extremamente inteligente. (Tinha o poder de uma SPARC station numa caixa de 15 x 10 x 10 cm). Infelizmente, ninguém estava interessado em produzi-lo na Sun e o pessoal do projeto Green teve que procurar outras formas de comercializar a tecnologia. Mesmo assim, nenhuma das empresas de equipamentos eletrônicos de consumo estava interessada. O grupo então ofereceu um projeto de construção de uma caixa de TV a cabo que pudesse lidar com novos serviços de cabo como vídeo sob demanda. Eles não conseguiram o contrato. (Ironicamente, a empresa que o fez era liderada por Jim Clark, que começou a Netscape – uma empresa que fez muito pelo sucesso da linguagem Java).

O projeto Green (com o novo nome de First Person Inc.) gastou todo o ano de 1993 e a metade de 1994 procurando quem comprasse sua tecnologia. Ninguém quis. (Patrick Naughton, um dos fundadores do grupo e a pessoa que acabou fazendo a maior parte da comercialização, afirma

que acumulou mais de 300000 milhas em viagens aéreas tentando vender a tecnologia). A First Person foi dissolvida em 1994.

Enquanto tudo isso ocorria na Sun, a parte World Wide Web da internet estava crescendo cada vez mais. A chave para a Web é o navegador (*browser*) que converte a página de hipertexto para a tela. Em 1994, a maioria usava Mosaic, um navegador Web não comercial que surgiu no centro de supercomputação da Universidade de Illinois, em 1993. (O Mosaic foi parcialmente escrito por Marc Andreessen por U\$ 6.85 a hora como estudante de pós-graduação em um projeto acadêmico. Ele passou para a fama e a fortuna como um dos co-fundadores e cabeça do grupo Netscape.)

Em uma entrevista para a SunWorld, Gosling disse que na metade de 1994 os desenvolvedores de linguagem perceberam que “Nós podíamos elaborar um navegador bom de verdade. Isso era uma das poucas coisas do processo cliente/servidor que precisava de algumas das coisas esquisitas que havíamos feito: neutralidade de arquitetura, tempo real, confiabilidade, segurança – questões que não eram terrivelmente importantes no mundo das estações de trabalho. Assim, construímos um navegador”.

O navegador em si foi elaborado por Patrick Naughton e Jonathan Payne e evoluiu até o navegador HotJava existente hoje em dia. Este foi escrito em Java para mostrar seu poder. Mas os responsáveis pela sua elaboração também tiveram em mente o poder do que agora chamamos de **applets**, de modo que eles fizeram o navegador capaz de interpretar os *bytecodes* intermediários. Essa demonstração de tecnologia foi exibida na SunWorld 95 em 23 de maio de 1995 e inspirou a moda Java que continua até hoje.

A grande jogada que espalhou a utilização da linguagem Java ocorreu na primavera de 1995 (hemisfério sul), quando a Netscape decidiu fazer a versão 2.0 do navegador Netscape com capacidade Java. O Netscape 2.0 foi lançado em janeiro de 1996 e tinha capacidade Java (assim como todas as versões posteriores). Outros licenciados incluíram empresas IBM, Symantec, Inprise e outras. Até mesmo a Microsoft foi licenciada e deu suporte à linguagem Java. O Internet Explorer tem capacidade Java e o Windows contém uma máquina virtual Java. (Note-se, porém que a Microsoft não suporta a versão mais recente da linguagem Java, e que sua implementação difere do Java padrão).

A Sun lançou a primeira versão da linguagem Java no início de 1996. Ela foi seguida pelo Java 1.02 alguns meses depois. As pessoas rapidamente perceberam que o Java 1.02 não seria usado no desenvolvimento de aplicativos sérios. Certamente alguém poderia usar o Java 1.02 para fazer um applet de texto nervoso (que move o texto ao acaso na tela). Mas não era possível sequer imprimir em Java 1.02! Para ser franco, o Java 1.02 não estava pronto para o horário nobre.

Os grandes anúncios sobre os futuros recursos da linguagem Java apareceram nos primeiros meses de 1996. Somente a conferência JavaOne, ocorrida em San Francisco em maio de 1996, apresentou um quadro completo mais claro de para onde o Java estava indo. No JavaOne, o pessoal da Sun Microsystems esboçou sua visão do futuro da linguagem Java com uma série aparentemente sem fim de aprimoramentos e novas bibliotecas. Havia a desconfiança de que isso levaria anos para acontecer. Não foi o que aconteceu porque num prazo relativamente curto a maioria das coisas prometidas tinham sido implementadas.

A grande novidade da conferência JavaOne de 1998 foi o anúncio do futuro lançamento do Java 1.2, que substitui a interface gráfica com o usuário e as ferramentas gráficas não profissionais por versões sofisticadas e expansíveis que se aproximaram muito mais da promessa de escreva uma vez, rode em qualquer lugar de seus antecessores. Três dias após seu lançamento, em dezembro de 1998, o nome foi alterado para Java 2. Novamente, não se acreditava que tudo seria implementado rapidamente. Surpreendentemente, grande parte foi implementada num tempo curto.

Características da Linguagem

Similaridade com C, C++

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

Também possui características herdadas de muitas outras linguagens de programação:

Objective-C, Smalltalk, Eiffel, Modula-3 etc. Muitas das características desta linguagem não são totalmente novas. Java é uma união de tecnologias testadas por vários centros de pesquisa e desenvolvimento de software.

Compilação

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

Além disso, existem compiladores de código nativo para Java (Asymetrix, Symantec, IBM etc). Há também outra forma de compilação, *just-in-time* (JIT). Esses compiladores trabalham compilando os *bytecodes* em código nativo uma vez, guardando o resultado e utilizando nas outras chamadas.

Portabilidade

Java foi criada para ser portátil. O *bytecode* gerado pelo compilador para a sua aplicação específica pode ser transportado entre plataformas distintas que suportam Java (Solaris 2.3®, Windows®, Mac/Os etc). Não é necessário recompilar um programa para que ele rode numa máquina e sistema diferentes, ao contrário do que acontece, por exemplo, com programas escritos em C e outras linguagens.

Esta portabilidade é importante para a criação de aplicações para a heterogênea Internet. Muitos dos programas aqui apresentados foram escritos e compilados numa plataforma Windows 98® e rodaram perfeitamente quando simplesmente copiados para uma plataforma Linux. Em Java um inteiro, por exemplo, tem sempre 32 bits, independentemente da arquitetura. O próprio compilador Java é escrito em Java, de modo que ele é portátil para qualquer sistema que possua o interpretador de *bytecodes*. Um exemplo de programa escrito em Java é o browser Hot Java.

Orientação a Objetos

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

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

Java também permite a criação de classes abstratas. Outra característica importante em linguagens orientadas a objetos é a segurança. Dada a sua importância o tópico foi considerado separadamente.

Segurança

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

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

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

Concorrência

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

Eficiência

Como Java foi criada para ser usada em computadores pequenos, ela exige pouco espaço, pouca memória. É muito mais eficiente que grande parte das linguagens de *scripting* existentes, embora seja muitas vezes mais lenta que C, o que não é um marco definitivo. Com a evolução da linguagem, existem geradores de *bytecodes* cada vez mais otimizados que levam as marcas de performance da linguagem mais próximas das de C++ e C. Além disso, um dia Java permite a possibilidade de gerar código executável de uma particular arquitetura *on the fly*, tudo a partir

do *bytecode*. São os compiladores JIT vistos acima que chegam a acelerar 10 a 20 vezes a execução de um programa.

Suporte para Programação de Sistemas Distribuídos

Java fornece facilidades para programação com *sockets*, *remote method call*, *tcp-ip* etc.

Exercícios de Fixação 2 – Histórico da linguagem de programação Java

Liste as principais características da linguagem de programação Java.

3

Fundamentos de Programação em Java

Exercício de nivelamento 2 – Fundamentos de Programação em Java

O que você entende por linguagem de programação?

Você já desenvolveu programas? Que linguagem de programação utilizou?

O que é Java

Java resulta da busca por uma linguagem de programação que englobe todas as características da linguagem C++, com a segurança de uma linguagem como SmallTalk.

Java é vista com uma linguagem de programação, um ambiente de desenvolvimento e um ambiente de aplicativos conforme ilustração da figura 01.

Existem duas grandes categorias de programas Java, os aplicativos, que rodam independentemente de um navegador e os applets, que dependem de um navegador para ser executado.

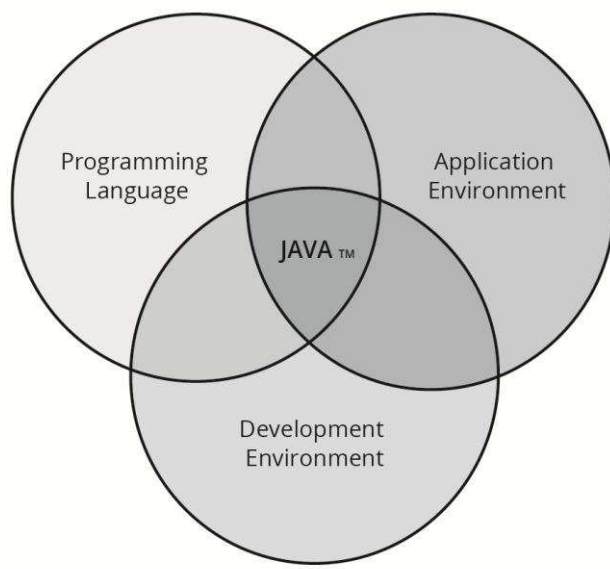


Figura 3.1 – Estrutura da linguagem Java.

Recursos da Linguagem Java

A arquitetura Java foi desenvolvida para atingir três objetivos através da implementação dos seguintes recursos: JVM (Máquina Virtual Java), Coleta de Lixo e Segurança de Código.

Máquina Virtual Java

A especificação da Máquina Virtual Java é definida como:

“Uma máquina imaginária que é implementada através de emulação em um software executado em uma máquina real. O código da Máquina Virtual Java está armazenado nos arquivo *.class*, que contém, cada, um código para, no máximo, uma classe pública.”

A especificação da Máquina Virtual Java fornece as especificações da plataforma de Hardware para a qual todo o código Java está compilado. Essa especificação permite que os programas Java sejam independentes de plataforma, já que a compilação é feita para uma máquina imaginária. Cabe ao interpretador Java de cada plataforma de hardware específica assegurar a execução do código compilado para a JVM. A figura seguinte ilustra o funcionamento da JVM.

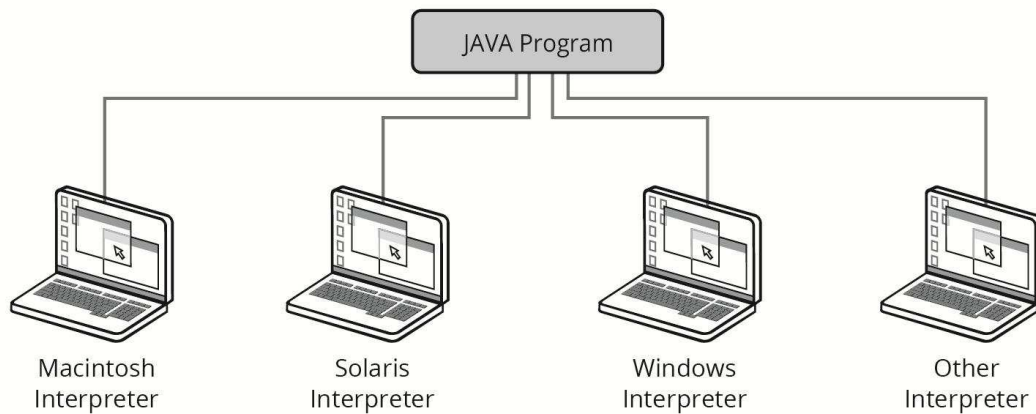


Figura 3.2 – Especificação da JVM Java.

Coleta de Lixo

A coleta de lixo é realizada automaticamente através de uma *thread* de segundo plano sendo executada no sistema operacional através da JVM específica desta plataforma de hardware. Nela os endereços de memória com referência 0 (zero) são removidos. Esta remoção ocorre durante o tempo de vida do programa em execução e tem como objetivo eliminar o vazamento de endereços de memória.

Segurança de Código

A segurança de código em Java é obtida através do processo de compilação e de verificação e execução de *bytecodes*.

A compilação tem como produto final um *bytecode*, independente de plataforma. A verificação e execução de *bytecodes* são feitas por interpretadores em tempo de execução. Este processo realiza três tarefas:

- Carregar o código: executado pelo utilitário de carga de classe.
- Verificar o código: executado pelo verificador de bytecode.
- Executar o código: executado pelo interpretador de tempo de execução.

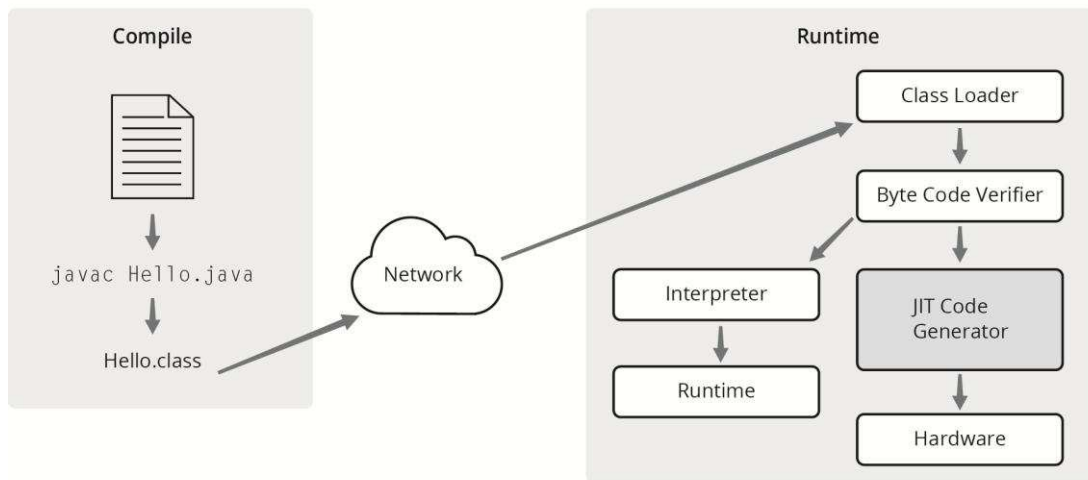


Figura 3.3 – Segurança de código em Java.

Primeiro Programa

Segue um programa simples em Java:

```

public class PrimeiroExemplo {
    public static void main(String[] args) {
        System.out.println("Meu primeiro Programa em Java!!!");
    }
}

```

A primeira coisa a ser observada é que Java é *case sensitive*, isto é, diferencia entre caracteres maiúsculos e minúsculos.

Analisando o programa acima, a palavra chave **public** é chamada **modificador de acesso**. Esses modificadores controlam o que outras partes do programa podem usar desse código. A outra palavra chave **class** diz que tudo está dentro de uma classe, como tudo em Java, e essa classe têm o nome de **PrimeiroPrograma**. Existe uma convenção para que os nomes das classes comecem sempre com letra maiúscula.

É necessário que o nome do arquivo tenha o mesmo nome da classe pública, nesse caso `PrimeiroPrograma.java`. Novamente não se pode trocar letras maiúsculas por minúsculas.

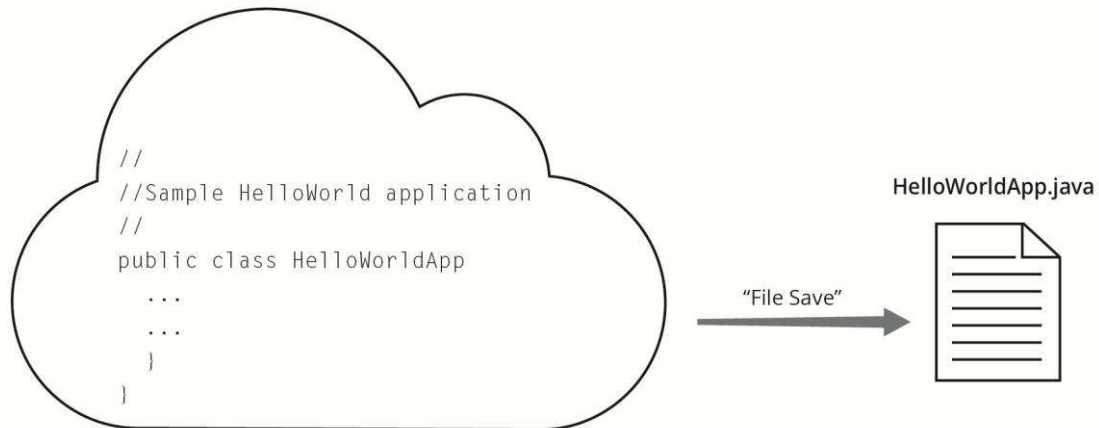


Figura 3.4 – Código-fonte em Java.

Ao ser compilado (`javac NomeDaClasse`) esse código gera um arquivo de bytecodes que é renomeado para `.class`. Para executar, basta dar o seguinte comando: `java NomeDaClasse`.

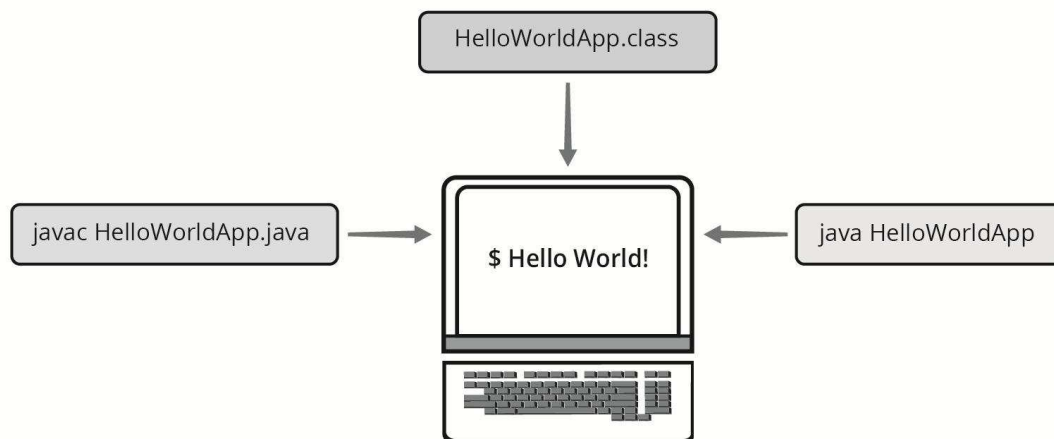


Figura 3.5 – Compilando e executando um programa Java.

Note que para executar o programa Java não é necessário informar a extensão `.class`. A execução começa com o código contido na função `main`, portanto sendo necessária sua presença. Os blocos de comandos são delimitados por chaves (`{` e `}`), assim como C/C++ e PHP.

O mais importante aqui é que todo programa em Java deve ter um método `main` com o cabeçalho apresentado na classe `PrimeiroExemplo`.

Dentro do método, existe uma linha de comando, terminada com ponto e vírgula (`;`). Todos os comandos terminam com esse sinalizador. Nessa linha, está sendo usado o objeto `System.out` e sendo executado o método `println`, que simplesmente imprime um conjunto de caracteres desejado na tela do computador. A forma geral de chamada de métodos de objetos em Java é: `Objeto.metodo(parametros);`. Se o método não possui parâmetros, usam-se parênteses vazios.

Alguns cuidados ao escrever e executar programas Java

Muitos erros podem ocorrer no momento que você rodar seu primeiro programa escrito em Java. Vamos ver alguns deles:

Código:

```
public class PrimeiroExemplo {  
public static void main(String[] args) {  
    System.out.println("Faltando ; no final da linha de comando!")  
}  
}
```

Erro:

PrimeiroExemplo.java:3: ';' expected

```
    }  
    ^
```

1 error

Esse é o erro de compilação mais comum. Lembre de que uma linha de comando em Java deverá terminar com o ponto e vírgula “;”. Repare que o compilador é explícito em dizer que a linha 3 esta com problemas. Outros erros de compilação podem ocorrer se você escrever de forma equivocada alguma palavra chave (que indicam comandos da linguagem), se esquecer de abrir e fechar chaves {}, dentre outros.

Durante a execução, outros erros podem aparecer:

- Se você declarar a classe Exemplo (primeira letra maiúscula), compilá-la e depois tentar usá-la como **e** minúsculo (java exemplo), o Java te avisa:

Exception in thread "main" java.lang.NoClassDefFoundError:

Exemplo (wrong name: exemplo)

- Se tentar acessar uma classe no diretório ou classpath errado, ou se o nome estiver errado, ocorrerá o seguinte erro:

Exception in thread "main" java.lang.NoClassDefFoundError: Exemplo

- Se esquecer de colocar static ou o argumento String[] args no método main:

Exception in thread "main" java.lang.NoSuchMethodError: main

- Se não colocar o método main como public:

Class Exemplo doesn't have a main method.

Comentários

Os comentários podem ser ou de uma linha (//) ou de várias linhas (/* e */). Por Exemplo:

```
// imprime na tela - comentário de uma linha  
System.out.printl("Hello World!!!");
```

```
/* Aqui sera impresso uma linha de texto na
tela: Hello World!!!
*/
System.out.println("Hello World!!!");
```

Existe ainda um terceiro tipo de comentário, utilizado para documentação automática do programa em desenvolvimento pela ferramenta javadoc do Java. Neste caso, devemos utilizar `/**` para começar e `*/` para terminar os textos que serão utilizados no processo de criação dos arquivos de documentação do programa em desenvolvimento.

Os comentários devem estar localizados imediatamente antes da definição da classe ou membro comentado. Cada comentário pode conter uma descrição textual sobre a classe ou membro, possivelmente incluindo tags HTML, e diretrizes para javadoc. As diretrizes para javadoc são sempre precedidas por `@`, como em:

- `@see nomeClasseOuMembro`: gera na documentação um link para a classe ou membro especificado, precedido pela frase "See also".

A documentação de uma classe pode incluir as diretrizes:

- `@author`: inclui informação sobre o autor na documentação.
- `@version`: inclui informação sobre a versão na documentação.

A documentação de um método pode incluir as diretrizes:

- `@param nome descrição` para a descrição de argumentos.
- `@return descrição` para a descrição do valor de retorno.
- `@exception nomeClasseExceção`: descrição para a descrição de exceções lançadas pelo método.

Apresentamos, a seguir, um exemplo de código java com comentários para geração de documentação pelo javadoc:

```
/**
 * @(#)PrimeiroExemplo.java
 *
 *
 * @author Paulo Henrique Cayres
 * @version 1.00 2010/3/2
 */

public class PrimeiroExemplo {
    public static void main(String args[]) {
        System.out.println("Meu primeiro Programa em Java.");
    }
}
```

Como resultado o javadoc irá gerar arquivos html com a estrutura de classes presente nos pacotes de classes do projeto em desenvolvimento. A figura seguinte apresenta o resultado da documentação de uma classe após a execução do javadoc.

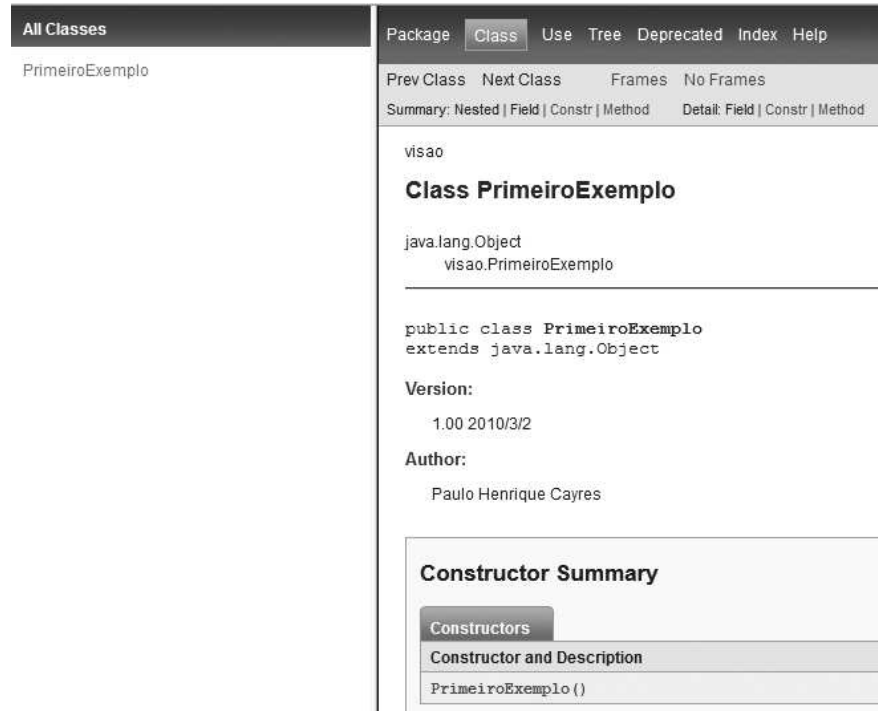


Figura 3.6 – Documentação de classes Java pelo javadoc.

Exercício de Fixação 3 – Linguagem de Programação

1. Crie uma classe executável Java para imprimir uma mensagem com o seu nome completo.
2. Crie uma classe executável Java para imprimir duas linhas de texto usando duas linhas de código `System.out`.
3. Crie a documentação das classes implementadas nos itens a e b utilizando o gerador de documentação do Java.
4. Qual a primeira impressão que você teve da linguagem de programação Java?

Variáveis

Exercícios de nivelamento 3 - Variáveis

Você sabe como o computador armazena informações na memória?

Em Java é necessária a declaração de variáveis para que possa usá-las. Eis alguns exemplos:

```
byte b;  
int umaVariavelInteira;  
long umaVariavelLong;  
char ch;
```

Deve-se prestar atenção ao ponto e vírgula no final das declarações, porque são comandos completos em Java.

Regras para nomeação de variáveis:

- Sequência de letras e dígitos;
- **Letras:** 'A' - 'Z', 'a' - 'z', '_' e qualquer caractere Unicode que denote uma letra em um idioma, por exemplo, em Alemão pode-se usar 'ä' nos nomes de variáveis, ou µ em grego. Apesar de em português se poder usar as vogais acentuadas ou cedilha, não é aconselhável, caso se abra o programa em outro sistema operacional, que pode gerar algum tipo de confusão;
- **Dígitos:** '0' - '9' e qualquer caractere Unicode que denote um dígito em uma linguagem;
- O comprimento do nome das variáveis é essencialmente ilimitado;
- Não se pode utilizar uma palavra reservada (como `if`, `switch`, etc).

Tipos de Dados

Java é fortemente tipada, o que significa que cada variável precisa ter seu tipo declarado para que possa ser usada. Há oito tipos primitivos. Seis numéricos (4 inteiros e 2 de ponto flutuante), um é o tipo `char` usado para caracteres no formato Unicode, e o outro é `boolean` para valores verdadeiro ou falso. Não existe nenhum tipo de dados equivalente ao `Variant`, que armazena qualquer tipo de dados básicos. Outros tipos mais elaborados, como `String`, não são tipos básicos e são objetos em Java.

Inteiros

São números sem parte fracionária. Veja a tabela a seguir:

Tipo	Armazenamento	Intervalo
Byte	8 bits – 1 byte	-128 a 127
short	16 bits – 2 bytes	-32.768 a 32.767
Int	32 bits – 4 bytes	-2.147.483.648 a 2.147.483.687
Long	64 bits – 8 bytes	-9.223.372.036.854.775.808L a - 9.223.372.036.854.775.807L

A parte mais interessante dos tipos em Java é que eles não dependem de plataforma, isto é, um valor `int` sempre terá 4 bytes, seja em Windows, seja em SPARC.

Ponto Flutuante

Os números de ponto flutuante denotam números com partes fracionárias. Há dois tipos:

Tipo	Armazenamento	Intervalo
Float	32 bits – 4 byte	Aproximadamente $\pm 3.40282347\text{E}+38\text{F}$ (6-7 dígitos decimais significativos)
Double	64 bits – 8 bytes	Aproximadamente \pm 1.79769313486231570E+308 (15 dígitos significativos)

Esses números seguem o formato IEEE 754. O tipo `float` é chamado de precisão simples. E o `double` de precisão dupla.

Tipo Caractere, char

Apóstrofes, ou aspas simples, são usados para denotar constantes `char`. Deve-se tomar cuidado com a diferença:

- ❑ `'H'` : caractere H
- ❑ `"H"` : string contendo um só caractere

Em Java, o tipo caractere é representado em **Unicode**. Normalmente não se precisa saber como funciona o Unicode, exceto se for necessário escrever aplicativos internacionais. Isto é, o Unicode foi desenvolvido para poder representar todos os caracteres de todos os idiomas possíveis do planeta. Para isso possui **2 bytes**, sendo assim, permitindo representar até 65536 caracteres, mas atualmente cerca de 35000 somente estão sendo usados.

No formato ASCII/ANSI, somente 1 byte é utilizado, com um total de 256 caracteres. Em Unicode, isso é representado pelos primeiros 255 caracteres.

Os caracteres Unicode seguem uma codificação em hexadecimal que vai de `'\u0000'` a `'\uffff'`, sendo os caracteres de `'\u0000'` até `'\u00ff'` os caracteres ASCII já conhecidos. O prefixo `\u` significa a codificação Unicode. Além dessa codificação, existe um conjunto de caracteres de controle válidos, conforme apresentado na tabela a seguir:

Sequência	Nome	Valor Unicode
<code>\b</code>	Backspace	<code>\u0008</code>
<code>\t</code>	Tabulação	<code>\u0009</code>
<code>\n</code>	Linefeed	<code>\u000A</code>
<code>\r</code>	Carriage return	<code>\u000D</code>
<code>\"</code>	Aspas duplas	<code>\u0022</code>
<code>\'</code>	Apostrofe (aspas simples)	<code>\u0027</code>
<code>\\</code>	Barra invertida	<code>\u005c</code>
<code>\b</code>	Backspace	<code>\u0008</code>

Em Java, valores do tipo `char` não são números, então sua conversão necessita uma conversão explícita de tipos.

Tipo boolean

O tipo lógico `boolean` tem dois valores: falso (`false`) e verdadeiro (`true`). É utilizado em testes lógicos.

Atribuições e Inicializações

Depois da declaração deve-se inicializar as variáveis. Não se deve ter variáveis não inicializadas, e o compilador geralmente avisa quando uma variável não foi inicializada.

Uma inicialização pode ser feita ou na hora da declaração ou depois, usando-se o nome da variável seguido do sinal de igual (=) e depois seguido do valor da inicialização. Por exemplo:

```
int teste;  
char ch;  
long i = 0;  
teste = 10;  
ch = 'S';
```

Outro ponto importante é que se pode declarar variáveis em qualquer lugar no código, mas cada variável (identificada por seu nome) somente uma vez em cada trecho.

Conversões entre Tipos Numéricos

Quaisquer operações em valores numéricos de tipos diferentes são aceitáveis e tratadas da seguinte maneira:

- Se qualquer um dos operandos for do tipo `double` então o outro será convertido para `double`;
- Caso contrário, se qualquer um dos operandos for `float`, o outro será convertido para `float`;
- Caso contrário, se qualquer um dos operandos for `long`, o outro será convertido para `long`.

Essa regra funciona também para `int`, `short` e `byte`.

Às vezes, conversões levam à perda de informação, como quando se quer transformar um `double` para `int`. Essas conversões são feitas através de *casts*, ou seja, conversões explícitas. A sintaxe é através de parênteses:

```
double x = 9.997;
int nx = (int) x;
```

Com isso a variável terá o valor 9, pois a conversão de um valor de ponto flutuante para um inteiro descarta a parte fracionária (e não é arredondado, como em VB). Quando se quer arredondar (`round`) um número fracionário para o inteiro mais próximo, usa-se `Math.round`:

```
double x = 9.997;
int nx = (int) Math.round(x);
```

No exemplo acima o valor 10 será retornado.

Outro ponto importante é que não se pode converter tipos `boolean` para outros tipos numéricos.

Constantes

Para se definir uma constante em Java, usa-se a palavra chave `final`:

```
public class UsaConstantes {
    public static void main(String[] args) {
        final double CM_POR_SOL = 2.54;
        double largPapel = 8.5;
        double altPapel = 11;
        System.out.println("Tamanho papel em cm: " +
            largPapel * CM_POR_SOL + " por " +
            altPapel * CM_POR_SOL);
    }
}
```

A palavra chave `final` significa que a variável só pode receber valor uma vez, e fica definido para sempre. É costume dar nomes totalmente em maiúsculas para constantes.

O mais comum em Java é declarar uma constante e fazer com que ela fique disponível para vários métodos dentro de uma classe. São geralmente chamadas de **constantes de classe**. Pode-se defini-las através das palavras-chave `static final`.

```
public UsaConstante2 {  
    public static final double G = 9.81;  
    public static void main(String[] args) {  
        System.out.println(G + " metros por Segundo ao quadrado.");  
    }  
}
```

Operadores

Segue uma tabela com os operadores aritméticos:

Operador	Significado	Exemplo
+	Soma	<code>x = 5 + 10;</code>
+=	Soma com atribuição	<code>x += 10; // x = x + 1</code>
++	Incremento de 1, após	<code>x++; // x = x + 1</code>
	Incremento de 1, antes	<code>++x; // x = x + 1</code>
-	Subtração	<code>x = 5 - 10;</code>
-=	Subtração com atribuição	<code>x -= 10; // x = x - 10</code>
--	Decremento de 1, após	<code>x--; // x = x - 1</code>
	Decremento de 1, antes	<code>--x; // x = x - 1</code>
*	Multiplicação	<code>x = 5 * 10;</code>
*=	Multiplicação com atribuição	<code>x *= 10; // x = x * 10</code>
/	Divisão, inteira	<code>x = 5 / 10;</code>
	Divisão com ponto flutuante	<code>x = 5.0 / 10.5;</code>
/=	Divisão com atribuição	<code>x /= 10; // x = x / 10</code>
%	Resto da divisão	<code>x = 5 % 10;</code>
%=	Resto da divisão com atribuição	<code>x %= 10; // x = x % 10</code>

A exponenciação é feita através da função `pow` da classe `Math`, encontrada em `java.lang`.
Exemplo:

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

O método `pow` recebe parâmetros `double` e retorna um `double` também.

Segue uma tabela com os operadores Lógicos e Relacionais:

Operador	Significado	Exemplo
==	Igual	if (x == 10) ...
!=	Diferente	if (x != 10) ...
>	Maior	if (x > 10) ...
>=	Maior ou igual	if (x >= 10) ...
<	Menor	if (x < 10) ...
<=	Menor ou igual	if (x <= 10) ...
&&	E lógico	if (x > 10) && (x < 20) ...
	Ou lógico	if (x > 10) (x < 20) ...
!	Não lógico	if (!(x > 10)) ...
?:	Condicional ternário	x = (y==10) ? 0 : 1; // se y==10, x recebe 0, senão recebe 1

Segue uma tabela com os operadores de bit a bit:

Operador	Significado	Exemplo
&	E bit a bit	x = y & 32;
&=	E bit a bit com atribuição	x &= 32; // x = x & 32
	Ou bit a bit	x = y 32;
=	Ou bit a bit com atribuição	x = 32; // x = x 32
^	Ou exclusivo (XOR)	x = y ^ 32;
^=	Ou exclusivo com atribuição	x ^= 32; // x = x ^ 32
~	Negação bit a bit	x = ~y;
>>	Deslocamento de bits à direita, estendendo o sinal	x = y >> 3; // desloca 3 bits à direita
>>=	Deslocamento de bits à direita, estendendo o sinal, com atribuição	x >>= 3; // x = x >> 3
<<	Deslocamento de bits à esquerda	x = y << 3; // desloca 3 bits à esquerda
<<=	Deslocamento de bits à esquerda, com atribuição	x <<= 3; // x = x << 3
>>>	Deslocamento de bits à direita, com preenchimento de zeros	x = y >>> 3;

Operador	Significado	Exemplo
>>>=	Deslocamento de bits à direita, com preenchimento de zeros, com atribuição	<code>x = >>>= 3; // x = x >>> 3</code>

Tipo String

Strings são sequências de caracteres, como "ola". O Java não possui um tipo nativo para o tratamento de strings, ao invés, possui uma classe pré-definida, chamada String. Cada string entre aspas é uma instância da classe String.

```
String e = ""; // string vazia
String ola = "Ola";
```

Concatenação

A linguagem Java permite que o sinal de + seja o operador para concatenação de strings.

```
String cur = "Curso";
String j = "Java";
String curso = cur + j;
```

O resultado do trecho acima é a atribuição de "CursoJava" para a variável `curso`. Perceba que nada é inserido nessa concatenação, por isso que não há nenhum espaço no resultado.

Ao concatenar uma string com um valor que não é uma string, este é convertido para string automaticamente:

```
String censura = 18 + "anos";
```

Substrings

Para se extrair uma substring de uma string, usa-se o método `substring` da classe String:

```
String saudacao = "Bem-vindo";
String s = saudacao.substring(0, 3);
```

Esse código cria uma string (`s`) que possui os caracteres "Bem". No comando `substring` acima, o primeiro parâmetro é o primeiro caractere a ser copiado (o primeiro caractere de uma string está na posição 0) e o segundo parâmetro é o índice do primeiro caractere que não se quer copiar. No exemplo, o caractere na posição 3 é o ' '.

Para se retornar somente um caractere da string usa-se:

```
String saudacao = "Bem-vindo";
char c = saudacao.charAt(2);
```

Que retornará o caractere 'm'.

Tamanho

O método **length** retorna o tamanho de uma string.

```
String saudacao = "ola";  
int n = saudacao.length(); // retorna 3
```

Teste de Igualdade

Para testar a igualdade de duas strings, usa-se o método `equals`. Por exemplo:

```
s.equals(t)
```

Retorna `true` se as strings `s` e `t` forem iguais e `false` caso contrário. Outra construção válida é:

```
"Ola".equals(comando)
```

Para testar se as strings são idênticas, sem levar em conta a diferenciação entre maiúsculas e minúsculas usa-se:

```
"Ola".equalsIgnoreCase("ola")
```

Não use o operador `==` para teste de igualdade, pois ele só compara se duas strings estão no mesmo local. E claro, se elas estiverem no mesmo local são idênticas.

Pode-se usar também o método `compareTo`:

```
if (saudacao.compareTo("Ola") == 0) ...
```

Manipulando dados em Java

O desenvolvimento de programas deve possibilitar, no mínimo, as funções básicas de entrada e saída de dados. Devemos nos preocupar em como iremos solicitar os dados aos usuários de um programa, bem como iremos imprimir os resultados que serão produzidos e que deverão ser mostrados aos usuários para que possam utilizá-los.

O Java possibilita o uso do console (uma janela DOS, por exemplo) ou o uso de interface gráfica (caixas de diálogos na plataforma Windows, por exemplo) para manipulação de dados.

Por exemplo, podemos lançar mão da classe `Scanner` para disponibilizar um canal de entrada de dados via console ou utilizarmos a classe `JOptionPane` para possibilitar a entrada de dados com o uso de ambiente visual gráfico. Outra classe muito útil neste processo é a `System` que nos possibilita um canal direto para os consoles padrão de entrada (teclado) e saída (tela do computador). Por exemplo, podemos utilizar `System.in` para obter os dados que estão sendo digitados no teclado e `System.out.print` para enviarmos uma mensagem para a tela do computador. Estas classes foram desenvolvidas para possibilitar interação entre o usuário e o programa que o mesmo necessita utilizar. O código Java a seguir ilustra o uso da classe `Scanner` e `JOptionPane` para possibilitar a entrada e saída de dados em modo texto ou visual gráfico.

Uso da classe `Scanner`:

```
Scanner sc = new Scanner(System.in);
```



```
String frase = sc.nextLine();
System.out.println(frase);

int i = sc.nextInt();
System.out.println(i);

double temperatura = sc.nextDouble();
System.out.println(temperatura);
```

Uso da classe JOptionPane:

```
String nome = JOptionPane.showInputDialog("Digite seu Nome: ");
JOptionPane.showMessageDialog(null, nome, "Mensagem",
JOptionPane.INFORMATION_MESSAGE);
```

Pode-se imprimir o resultado esperado como resposta a um problema apresentado no console (uma janela DOS, por exemplo) usando a instrução `System.out.print(x)`. Também podemos lançar mão de funcionalidades da classe `JOptionPane` presente no pacote `swing` do Java. O pacote `swing` do Java possibilita o uso de interface gráfica para desenvolvimento de aplicações. Neste curso iremos utilizar o `JOptionPane` apenas para nos auxiliar na construção de .

Exercícios de Fixação 4 – Variáveis

Liste os conceitos e recursos de programação que aprendeu até agora.

Quais são as três principais funcionalidades que um programa deve apresentar para possibilitar a manipulação e geração de informações tendo como base um conjunto de dados fornecidos?

Atividade 1 – Variáveis e manipulação de dados

1. Faça uma classe executável que dados o valor do depósito em uma poupança e a taxa de juros, informe a valor do rendimento (lucro) e o valor final (saldo resultante).

2. Faça uma classe executável que dados a distância percorrida (em Km) e o tempo gasto em uma viagem (em horas) e calcule e informe a velocidade média do carro, sabendo que $Velocidade = \Delta S / \Delta T$ (variação de distância / variação do tempo).
3. Faça uma classe executável que dado o salário de um funcionário, informe o valor de imposto de renda a ser pago, sabendo que o imposto equivale a 5% do salário.

Atividade 2 – Variáveis e manipulação de dados (Complementar)

1. Faça uma classe executável que dado o total de despesas realizadas em um restaurante, informe o valor total a ser pago com gorjeta, considerando a taxa de 10%.
2. Faça uma classe executável que dado o valor de um produto, informe o novo valor de venda sabendo-se que este sofreu um aumento de 25%.
3. Faça uma classe executável que dados o peso atual de uma pessoa e o peso desejado após processo de emagrecimento, informe o percentual do peso que deverá ser eliminado.
4. Faça uma classe executável que dados a quantidade de fitas que uma vídeo locadora possui e o valor que ela cobra por cada aluguel, informe:
 - ▣ Sabendo que um terço das fitas são alugadas por mês, o seu faturamento anual.
 - ▣ Sabendo que quando o cliente atrasa a entrega, é cobrada uma multa de 10% sobre o valor do aluguel e que um décimo das fitas alugadas no mês são devolvidas com atraso, o valor ganho com multas por mês.

4

Controle de Fluxo

Exercício de nivelamento 4 – Controle de Fluxo

Você poderia fazer uma breve descrição de como toma decisões para solucionar problemas do seu dia a dia?

Como você imagina que os computadores tomam decisões para solucionar nossos problemas do dia a dia?

Cite alguns problemas de que você necessita do auxílio do computador para solucionar.

Escopo de um Bloco de comandos

Um bloco de comandos consiste de vários comandos agrupados com o objetivo de relacioná-los com determinado comando ou função em busca da solução de um problema. Podemos utilizar blocos de comandos em instruções como `if`, `for`, `while`, `switch` e em declarações de funções para permitir que instruções façam parte de um contexto desejado que permita solucionar um problema. Blocos de comandos em Java são delimitados pelos caracteres `{` e `}`. A utilização dos delimitadores de bloco em uma parte qualquer do código não relacionada com os comandos citados ou funções não produzirá efeito algum, e será tratada normalmente pelo interpretador.

Pode-se inclusive aninhar blocos, isto é, blocos dentro de blocos. A restrição é que não se pode declarar variáveis com mesmo nome dentro de blocos aninhados. Exemplo:

```
public static void main(String[] args) {
    int n;
    ...
    {
        int k;
        int n;    // erro!!! Variáveis com mesmo nome
        ...
    }
}
```

Estruturas de Decisão

Também chamadas de estruturas condicionais, os comandos de seleção permitem executar instruções ou blocos de comandos com base em testes lógicos feitos durante a execução de um programa.

Comando if

O mais trivial das estruturas condicionais é o `if`. Ele testa uma determinada condição e executa o bloco de comandos indicado se o resultado da análise de uma condição for `true`. O comando `if` possui a seguinte sintaxe:

```
if (expressão)
    comando;

if (expressão) {
    comando_1;
    ...
    comando_n;
}
```

Para incluir mais de um comando na instrução `if` é preciso utilizar a marcação de bloco comandos, demarcando por chaves o início e fim das instruções a serem executadas caso o resultado da análise de uma condição for verdadeira (`true`).

O `else` é um complemento opcional para a instrução `if`. Se utilizado, o comando será executado se o resultado da análise de uma condição retornar o valor `false`. Suas duas sintaxes são:

```
if (expressão)
    comando_1;
else
    comando_2;

ou:

if (expressão)
```

```
    comando;
else {
    comando;
    ...
    comando;
}
```

A seguir, temos um exemplo do comando `if/else`:

```
int a, b, maior;
...
if (a > b)
    maior = a;
else
    maior = b;
```

O exemplo acima armazena na variável `maior` o maior valor entre `a` e `b`. Devido a um processo interno do Java o seguinte código:

```
if (x!=0 && 1/x + y > x)
```

não avalia $1/x$ caso `x` seja igual a 0 (divisão por 0).

Comando switch

O comando `switch` atua de maneira semelhante a uma série de comandos `if` na mesma expressão. Frequentemente o programador pode querer comparar uma variável com diversos valores, e executar um código diferente a depender de qual valor é igual ao da variável. Quando isso for necessário, deve-se usar o comando `switch`. O exemplo seguinte mostra dois trechos de código que fazem a mesma coisa, sendo que o primeiro utiliza uma série de `if`'s e o segundo utiliza `switch`:

```
if (i == 0)
    System.out.print("i é igual a zero");
else if (i == 1)
    System.out.print("i é igual a um");
else if (i == 2)
    System.out.print("i é igual a dois");

switch (i) {
case 0:
    System.out.print("i é igual a zero");
    break;
case 1:
    System.out.print("i é igual a um");
    break;
case 2:
    System.out.print("i é igual a dois");
    break;
```

```
default:
System.out.print("i diferente de tudo");
break;
}
```

A cláusula `default` serve para tratar valores que não estão em nenhum `case`. É opcional.

É importante compreender o funcionamento do `switch` para não cometer enganos. O comando `switch` testa linha a linha dos `cases` encontrados, e a partir do momento que encontra um valor igual ao da variável testada, passa a executar todos os comandos seguintes, mesmo os que fazem parte de outro teste, até o fim do bloco. Por isso usa-se o comando `break`, quebrando o fluxo e fazendo com que o código seja executado da maneira desejada. Segue outro exemplo:

```
switch (i) {
case 0:
    System.out.print("i é igual a zero");
case 1:
    System.out.print("i é igual a um");
case 2:
    System.out.print("i é igual a dois");
}
```

No exemplo acima, se `i` for igual a zero, os três comandos `System.out.print` serão executados. Se `i` for igual a 1, os dois últimos `System.out.print` serão executados. O comando só funcionará da maneira desejada se `i` for igual a 2.

Exercício de Fixação 5 – Estruturas de Decisão

1. Crie uma classe executável Java que leia dois números e imprima o menor.
2. Crie uma classe executável Java que receba dois números e execute as operações listadas abaixo de acordo com a escolha do usuário.

Escolha do usuário	Operação
1	Média entre os números digitados
2	Diferença do maior pelo menor
3	Produto entre os números digitados
4	Divisão do primeiro pelo segundo

Se a opção digitada for inválida, mostrar uma mensagem de erro e terminar a execução do programa. Lembre-se que na operação 4 o segundo número deve ser diferente de zero.

3. Uma empresa decide dar um aumento de 30% aos funcionários com salários inferiores a R\$ 500,00. Crie uma classe executável Java que receba o salário do funcionário e mostre o valor do salário reajustado ou uma mensagem caso o funcionário não tenha direito ao aumento.

Estruturas de Repetição

Exercício de nivelamento 5 – Estruturas de Repetição

Nós já vimos que necessitamos de variáveis para armazenar valores para possibilitar processamento de dados e impressão de resultados? Imagine que você tenha que desenvolver um programa para ler e imprimir notas de 5000 alunos. Que ideia você apresentaria para solucionar este problema com os recursos que você aprendeu até agora?

Definição

Estruturas de repetição são estruturas que repetem um bloco de instruções até que uma condição de parada especificada seja satisfeita. No Java, trabalharemos com as seguintes estruturas:

- `while`
- `do ... while`
- `for`

Comando `while`

O `while` é o comando de repetição (laço) mais simples. Ele testa uma condição lógica e executa um comando, ou um bloco de comandos, até que a condição testada seja falsa. Sintaxe:

```
while (<expressao>)
    <comando>;

while (<expressao>) {
    comando_1;
    ...
    comando_n;
}
```

A expressão só é testada a cada vez que o bloco de instruções termina, além do teste inicial. Se o valor da expressão passar a ser `false` no meio do bloco de instruções, a execução segue até o final do bloco. Se no teste inicial a condição for avaliada como `false`, o bloco de comandos não será executado.

O exemplo a seguir mostra o uso do `while` para imprimir os números de 1 a 10:

```
i = 1;
while (i <=10)
    System.out.print(i++);
```

Comando do ... while

O laço `do...while` funciona de maneira bastante semelhante ao `while`, com a simples diferença que a expressão é testada ao final do bloco de comandos. O laço `do...while` possui apenas uma sintaxe, que é a seguinte:

```
do {
    comando_1
    ...
    comando_n
} while (<expressao>);
```

O exemplo utilizado para ilustrar o uso do `while` pode ser feito da seguinte maneira utilizando o `do...while`:

```
i = 0;
do {
    System.out.print(++i);
} while (i < 10);
```

Comando for

O tipo de laço mais complexo é o `for`. Para os que programam em C, C++, a assimilação do funcionamento do `for` é natural. Mas para aqueles que estão acostumados a linguagens como Pascal, há uma grande mudança para o uso do `for`. As duas sintaxes permitidas são:

```
for (<inicializacao>; <condicao>; <incremento>)
    comando;

for (<inicializacao>; <condicao>; <incremento>) {
    comando_1;
    ...
    comando_n;
}
```

As três expressões que ficam entre parênteses têm as seguintes finalidades:

- **Inicialização:** Comando ou sequência de comandos a serem realizados antes do início do laço. Serve para inicializar variáveis;
- **Condição:** Expressão booleana que define se os comandos que estão dentro do laço serão executados ou não. Enquanto a expressão for verdadeira (valor diferente de zero) os comandos serão executados;
- **Incremento:** Comando executado ao final de cada execução do laço.

Um comando `for` funciona de maneira semelhante a um `while` escrito da seguinte forma:


```
<inicializacao>
while (<condicao>) {
    comandos
    ...
    <incremento>
}
```

Exemplo:

```
for (i=1; i<=10; i++)
    System.out.print(i);
```

Quebra de fluxo

Os comandos de quebra de fluxo servem para sair dos laços incondicionalmente, ou para executar o próximo passo.

Comando Break

O comando `break` pode ser utilizado em laços `do`, `for` e `while`, além do uso já visto no comando `switch`. Ao encontrar um `break` dentro de um desses laços, o interpretador Java para imediatamente a execução do laço, seguindo normalmente o fluxo do código:

```
while (x > 0) {
    ...
    if (x == 20) {
        System.out.println("erro! x = 20");
        break;
    }
    ...
}
```

No trecho de código acima, o laço `while` tem uma condição para seu término normal (`x <= 0`), mas foi utilizado o `break` para o caso de um término não previsto no início do laço. Assim o interpretador seguirá para o comando seguinte ao laço.

Existe outra modalidade de `break` que é o `break` rotulado. Eles são usados quando se necessita sair de blocos aninhados e não se quer inserir um monte de condições de saída, em cada um dos níveis dos laços. Por exemplo:

```
int n;
ler_dados:
while (...) {
    for (...) {
        n = Console.readInt(...);
        if (n < 0)    // nunca deveria ocorrer
            break ler_dados;
        ...
    }
}
```

```
}  
// verifica se a entrada de dados foi bem sucedida  
if (n<0) {  
    // situação inválida  
}  
else {  
    // entrada de dados normal  
}
```

Comando continue

O comando `continue` também deve ser utilizado no interior de laços, e funciona de maneira semelhante ao `break`, com a diferença que o fluxo ao invés de sair do laço volta para o início dele, executando a próxima iteração. Vejamos o exemplo:

```
for (i = 0; i < 100; i++) {  
    if (i % 2)  
        continue;  
    System.out.println(i);  
}
```

O exemplo acima é uma maneira ineficiente de imprimir os números pares entre 0 e 99. O que o laço faz é testar se o resto da divisão entre o número e 2 é 0. Se for diferente de zero (valor lógico `true`) o interpretador encontrará um `continue`, que faz com que os comandos seguintes do interior do laço sejam ignorados, seguindo para a próxima iteração. Pode-se usar o comando `continue` dentro de laços `for`, `while` e `do...while`.

Comando return

O comando `return` é usado para sair do método corrente, e também para retornar valores de métodos que têm retorno. Exemplo:

```
return ++cont;  
  
return;
```

Exercícios de Fixação 6 – Controle de fluxo

Liste os recursos de programação utilizados para implementar estruturas para tomada de decisão e de repetição de bloco de comandos disponíveis na linguagem de programação Java.

Você acredita que as estruturas de repetição facilitam o desenvolvimento de programas que manipulam grandes volumes de dados? Justifique sua resposta.

Atividade 1 – Controle de fluxo

1. Faça uma classe executável que calcule e imprima o somatório de todos os números inteiros existentes entre 200 e 400.
2. Faça uma classe executável que calcule e imprima a média do somatório dos números inteiros existentes entre 500 e 700.
3. Faça uma classe executável que dado um número inteiro, calcule e imprima o seu fatorial. O fatorial de um número n é $n * n-1 * n-2 \dots$ até $n = 1$. Por exemplo, o fatorial de 4 é 24 ($4 * 3 * 2 * 1$).

Atividade 2 – Controle de fluxo (complementar)

1. Faça uma classe executável que calcule e imprima o somatório dos números pares existentes entre 100 e 500.
2. Faça uma classe executável que calcule e imprima o somatório dos números ímpares existentes em 300 e 700.
3. Faça uma classe executável que dado um número, informe se ele é divisível por 5.
4. Uma prefeitura abriu uma linha de crédito para seus funcionários. O valor máximo da prestação não poderá ultrapassar 25% do salário bruto. Faça uma classe executável que dados o valor do salário bruto e o valor da prestação, informe se o empréstimo pode ou não ser concedido.

5. Faça uma classe executável que dados os limites de um intervalo $[a; b]$, informe a soma de todos os números naturais neste intervalo. Exemplo: $[4, 7] \Rightarrow 4+5+6+7=22$.

5

Arrays

Exercício de nivelamento 6 – Arrays

No início do capítulo anterior você pensou numa solução para resolver o problema de implementar um programa para ler e imprimir notas de 5000 alunos. Agora precisamos da mesma solução sendo que todas as notas lidas devem ser armazenadas na memória do computador. Que solução você apresentaria para solucionar este problema com os recursos que você aprendeu até agora?

Definição

Um array é uma variável composta homogênea unidimensional formada por uma sequência de valores, todas do mesmo tipo, com o mesmo identificador (mesmo nome) e alocadas sequencialmente na memória. Uma vez que os valores são alocados sequencialmente e tem o mesmo nome, a forma utilizada para acessá-los e através de um índice, que referencia a localização de um valor dentro da estrutura onde ele está armazenado.

Arrays são objetos em Java que possibilitam o armazenamento de dados primitivos. Arrays servem para manipular uma grande quantidade de dados sem que se precise declarar várias variáveis do mesmo tipo.

Aprender a usar arrays sempre é um problema em qualquer linguagem de programação. Isso porque envolve uma série de conceitos, sintaxe e outras definições de manipulação de endereço de memória no computador. No Java, muitas vezes utilizamos outros recursos em vez de arrays, em especial os pacotes de coleções do Java, que veremos mais a frente neste curso. Portanto, fique tranquilo caso não consiga digerir toda sintaxe dos arrays num primeiro momento.

Um array é uma estrutura de dados que armazena um conjunto de valores de mesmo tipo. Você acessa cada valor individual através de um índice inteiro. Por exemplo, se **x** é um array de inteiros, então **x[i]** é o **i**-ésimo inteiro no array.

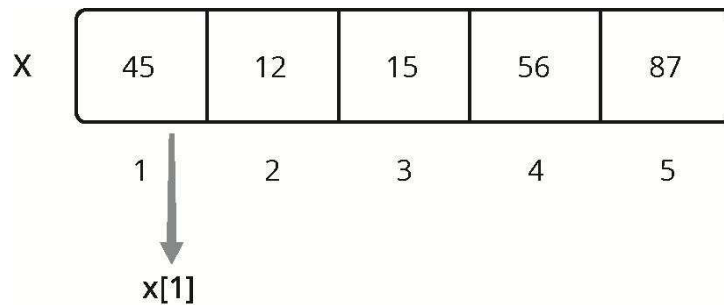


Figura 5.1 – Representação de array na memória do computador.

Você declara uma variável de array especificando o tipo primitivo de dados que ele armazenará seguido de [] (abre e fecha colchetes) e o nome da variável do tipo array. Por exemplo, **a** representa a declaração de um array de inteiros:

```
int[] a;
```

Entretanto, essa instrução apenas declara a variável **a** como sendo um array. Ela ainda não foi inicializada como um array, não sendo possível, ainda, armazenar valores em **a**. O operador *new* é utilizado para criar o array na memória do computador. Utilize a instrução abaixo para informar o total de endereços de memória que serão referenciados pelo identificador **a**.

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

Essa instrução configura um array que pode conter no máximo 100 números inteiros. As entradas do array são numeradas de 0 a 99 (e não de 1 a 100). Uma vez que o array foi criado, você pode preencher suas entradas (endereços de memória) usando, por exemplo, uma estrutura de repetição:

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

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

```
    a[i] = i; // preenche o array com 0 a 99
```

Atenção: Se você construir um array com 100 elementos e depois tentar acessar o elemento **a[100]** (ou qualquer outro índice fora do intervalo 0 . . . 99), o seu programa terminará com uma exceção *“array index out of bounds”* que indica que você está tentando acessar um endereço de memória que não pode ser referenciado pela variável **a**.

Você pode utilizar o Para encontrar o número de elementos de um array, use `arrayName.length`. Por exemplo,

```
for (int i = 0; i < a.length; i++)
```

```
    System.out.println(a[i]);
```

Uma vez que você crie um array, você não poderá mudar seu tamanho, embora você possa, é claro, mudar um elemento específico do array.

Você pode definir uma variável de array, como segue:

```
int[] a;
```

ou como:

```
int a[];
```

A maioria dos programadores Java prefere o primeiro estilo, pois ele separa corretamente o tipo `int[]` (array de inteiros) do nome da variável.

Tamanho de uma array

Uma vez criado o array, é complicado alterar o seu tamanho. A partir do momento que uma array foi criada num programa em execução ela não pode mudar de tamanho. A alteração poderá ser feita antes de se iniciar a execução do programa, durante o processo de criação do mesmo.

Vale lembrar que já tivemos um exemplo de array em Java, na clássica função `main(String[] args)`. Esse array referencia um conjunto de strings, que são os parâmetros passados para um programa na linha de comando quando iniciamos sua chamada para ser executada.

Arrays são o primeiro exemplo de objetos cuja criação o programador precisa lidar explicitamente, usando o operador `new`.

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

A linha de comando acima define um array de inteiros de 100 elementos, indexados de 0 a 99. Uma vez criado, pode-se preencher esse array da seguinte forma:

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

Existe ainda uma forma simplificada:

```
int[] smallPrimes = {2, 3, 5, 7, 11, 13};
```

Nesse caso não se usa o comando `new` que cria um array de inteiros e preenche com os dados dentro das chaves. Isso é usado quando se quer passar um array como parâmetro e não se quer criar uma variável para isso, por exemplo:

```
printLabels(new String[] {"Regiao", "Vendas"});
```

Utilize o atributo `length` para se descobrir o tamanho de um array da seguinte forma:

```
arrayName.length.
```

Cópia de Arrays

Pode-se copiar uma variável de array em outra, fazendo com que as duas variáveis façam referência ao mesmo array.

```
int [] luckyNumbers = smallPrimes;
```

Nesse exemplo os dois arrays irão acessar os mesmos elementos. Caso se queira realmente copiar os elementos de um para o outro, fazendo com que os elementos sejam duplicados, usa-se o método `arraycopy` da classe `System`. A sintaxe é:

```
System.arraycopy(de, indiceDe, para, indicePara, contagem);
```

Arrays como Argumentos

Os arrays podem ser usados como argumentos de um método definido pelo usuário, exatamente como qualquer outro tipo. Deve-se saber, contudo, que os métodos que recebem arrays como parâmetros podem alterar os elementos do array.

Arrays como Valores de Retorno

O tipo de retorno de um método pode também ser um array. Isso é útil quando se deseja retornar uma sequência de valores. Por exemplo:

```
public static int[] drawing(int high, int number)
```

Arrays Multidimensionais

Arrays multidimensionais, ou matrizes, para simplificar, são fáceis de declarar:

```
int[][] Tabuleiro;
```

Uma vez inicializada:

```
Tabuleiro = new int[8][8];
```

Pode-se utilizar cada um dos elementos da seguinte forma:

```
Tabuleiro[3][2] = 10;
```

Nota-se que a sintaxe e manipulação dos elementos é muito similar à usada em outras linguagens de programação. Mas um aspecto deve ser visto com calma. Os arrays multidimensionais são implementados com arrays de arrays.

No exemplo acima, `Tabuleiro` na verdade é um array com 8 elementos, cada um desses elementos sendo um array de oito elementos inteiros.

A expressão `Tabuleiro[3]` se refere ao subarray de índice 3, e `Tabuleiro[3][2]` se refere ao elemento (int) de índice 2 desse subarray.

Para inicializar uma matriz na declaração, usa-se como no exemplo:

```
int [][] vendas = {{1998, 1998}, {10000, 20000}};
```


Exercícios de Fixação 7 – Arrays

Você poderia descrever os ganhos e limitações que temos quanto implementamos programas utilizando array?

Você acredita que arrays facilitam o desenvolvimento de programas que manipulam grandes volumes de dados? Justifique sua resposta.

Atividade 1 – Arrays

1. Faça uma classe executável que dado um conjunto de 20 elementos numéricos, informe a soma de todos os números pares.
2. Faça uma classe executável que dado um conjunto com a idade de 50 pessoas, informe a percentual em cada faixa etária: de 0 a 15 anos, de 16 a 30 anos, de 31 a 45 anos, de 46 a 60 anos e mais que 60 anos.
3. Faça uma classe executável que receba por meio da linha de comando do programa (prompt) um número natural (de 1 a 10) e informe a sua tabuada.

Atividade 2 – Arrays (Complementar)

1. Faça uma classe executável que leia a altura, a idade e o sexo de 50 pessoas e forneça as seguintes informações:
 - a. A maior altura e a menor altura;
 - b. A média de altura de mulheres;
 - c. A idade do homem mais velho.

2. Faça uma classe executável que leia uma matriz quadrada de ordem 4x4 e forneça as seguintes informações:

- a. Imprima os elementos da diagonal principal;
- b. Imprima os elementos da diagonal secundária;
- c. Imprima a média aritmética de todos os elementos da matriz.

6

Classes Especializadas

Exercício de nivelamento 7 – Classes Especializadas

Listar padrões de formatação de informações que você consegue identificar no seu dia a dia?
Você conhece padrões de formatação que são diferentes entre continentes ou países?

Wrappers de Objetos

Ocasionalmente é necessário converter um tipo básico como `int`, para um objeto. Todos os tipos básicos têm correspondentes de classe. Por exemplo, existe um correspondente da classe `Integer` para o tipo básico `int`. Esses tipos de classes são normalmente chamadas de *wrappers de objeto*. As classes wrapper têm nomes óbvios: `Integer`, `Long`, `Float`, `Double`, `Short`, `Byte`, `Character`, `Void` e `Boolean`. As seis primeiras herdam do wrapper comum `Number`. As classes wrapper são `final`. Portanto, você não pode sobrepor o método `toString` em `Integer`, para exibir números usando algarismos romanos, por exemplo. Você também não pode alterar os valores que armazena no *wrapper* de objeto (HORSTMANN, 2003).

Frequentemente, você verá os wrappers de número por outro motivo. Os projetistas da linguagem Java acharam os wrappers um lugar conveniente para colocar certos métodos básicos, como aqueles para converter strings de dígitos para números.

Para converter uma string em um inteiro, você precisa usar a instrução a seguir:

```
int x = Integer.parseInt(s);
```

Analogamente, você pode usar o método `Double.parseDouble` para analisar números de ponto flutuante. A tabela abaixo apresenta as classes wrappers e seus tipos de dado primitivo correspondente.

Tipo primitivo	Classe wrapper	Parâmetro para o construtor
Byte	Byte	byte ou String
Short	Short	short ou String

Tipo primitivo	Classe wrapper	Parâmetro para o construtor
Int	Integer	int ou String
Long	Long	long ou String
Float	Float	float, double ou String
Double	Double	double ou String
Char	Character	char
Boolean	Boolean	boolean ou String

Outro uso importante dos *wrappers* é ser um lugar interessante para agrupar funcionalidades referentes ao tipo básico que ele empacota, como alguns tipos de conversões suportados pela linguagem Java.

Date, DateFormat e Calendar

Date

A classe *Date* representa um instante específico no tempo, com precisão de milissegundos.

Embora a classe *Date* pretenda reproduzir o tempo universal coordenado (UTC), ela não o faz exatamente. O processo de reprodução é dependente do ambiente onde a máquina virtual Java esta hospedado. Quase todos os sistemas operacionais modernos assumem que 1 (um) dia é igual a 86400 segundos (24 horas \times 60 minutos \times 60 segundos). No UTC, no entanto, cerca de uma vez a cada ano ou dois, há um segundo extra, chamada de "segundo bissexto". O saldo de um segundo sempre é adicionado como o último segundo do dia, e sempre em 31 de dezembro ou 30 de junho de um determinado ano. Por exemplo, o último minuto do ano de 1995 foi de 61 segundos de duração, graças ao salto de um segundo. A maioria dos relógios de computadores não é precisa o suficiente para ser capaz de acrescentar o salto de segundo previsto no UTC.

Alguns padrões de computador são definidos em termos do meridiano de Greenwich (GMT), o que equivale à hora universal (UT). GMT é o nome "civil" para o padrão; UT é o nome "científico" para o mesmo padrão. A distinção entre o UTC e UT é que UTC é baseado em um relógio atômico e UT é baseado em observações astronômicas, que para todos os efeitos práticos, é um fino de cabelo invisível para se dividir, devido à rotação da Terra não ser uniforme (que desacelera e acelera de maneiras complicadas), UT nem sempre flui de maneira uniforme. Os saltos de segundos são introduzidos conforme a necessidade em UTC, de modo a manter dentro de 0,9 segundo UTC de UT1, que é uma versão de UT com determinadas correções aplicadas. Há outra hora e sistemas de datas, bem como, por exemplo, a escala de tempo utilizado pelo sistema de posicionamento por satélite (GPS), que é sincronizada para UTC, mas não está ajustada para os segundos bissextos.

Em todos os métodos da classe *Date* que aceitar ou voltar ano, mês, dia, hora, minuto, segundo e os valores, as representações seguintes são utilizados:

- Um ano *y* é representado pelo número inteiro *y* - 1900.

- Um mês é representado por um número inteiro de 0 a 11; 0 é de Janeiro, 1 é de Fevereiro, e assim por diante, assim é 11 de Dezembro.
- A data (dia do mês) é representada por um inteiro de 1 a 31.
- Uma hora é representada por um número inteiro de 0 a 23. Assim, a hora a partir de meia-noite às 01:00 horas é 0, e na hora do meio-dia às 01:00 é de 12 horas.
- Um minuto é representado por um inteiro de 0 a 59.
- Um segundo é representado por um inteiro de 0 a 61; os valores 60 e 61 ocorrem apenas durante segundos bissextos e mesmo assim apenas em implementações Java que efetivamente controlam segundos intercalados corretamente.

DateFormat

DateFormat é uma classe abstrata para formatação de data/hora. Subclasses de formatação de data/hora, como SimpleDateFormat, permite a formatação (de data para texto), análise (de texto para data), dentre outras funcionalidades. A data é representada como um objeto Date ou como os milissegundos desde 1º de Janeiro de 1970, 00:00:00 GMT.

DateFormat fornece métodos de classe para a obtenção de muitos formataadores padrão de data/hora com base no padrão ou uma localidade específica e uma série de estilos de formatação. Os estilos de formatação incluem FULL, LONG, MEDIUM e SHORT.

DateFormat ajuda a formatar e analisar as datas para qualquer localidade. Seu código pode ser completamente independente das convenções de localização de meses, dias da semana, ou até mesmo o formato do calendário: lunar versus solar.

Para formatar uma data para a localidade atual, use um dos métodos de fábrica estáticos:

```
myString = DateFormat.getDateInstance().format(myDate);
```

Se você estiver formatando várias datas, o mais eficiente para obter o formato e usá-lo várias vezes para que o sistema não tenha que buscar as informações sobre o idioma local e convenções do país várias vezes.

```
DateFormat df = DateFormat.getDateInstance();  
for (int i = 0; i < a.length; i++) {  
    output.println (df.format(myDate [i]) + ";");  
}
```

Utilize a chamada para o método getDateInstance para formatar uma data para uma localidade diferente.

```
DateFormat df = DateFormat.getDateInstance (DateFormat.LONG, Locale.FRANCE);
```

Você pode usar um DateFormat para converter uma string em data também.

```
myDate = df.parse (myString);
```

Use `getDateInstance` para obter o formato da data para uma determinada localidade. Existem outros métodos disponíveis que podem ser utilizados. Por exemplo, use `getTimeInstance` para obter o formato de tempo de um determinado país. Use `getDateTimeInstance` para obter um formato de data e hora. Você pode passar em opções diferentes para estes métodos de fábrica para controlar o comprimento do resultado; de curto a médio e longo para FULL. O resultado exato depende da localidade, mas em geral:

SHORT é completamente numérica, como 12.13.52 ou 15:30

MEDIUM é mais longo, como 12 de janeiro de 1952

LONG é mais longo, como 12 janeiro de 1952 ou 03:30:32

FULL é mais completo, como terça-feira, 12 de abril, 1952 AD ou 03:30:42 PST.

Calendar

Calendar é uma classe abstrata de base para a conversão entre um objeto *Date* e um conjunto de campos inteiros, tais como ano, mês, dia, hora, e assim por diante. Um objeto *Date* representa um momento específico no tempo com precisão de milissegundos.

Subclasses de *Calendar* interpretam uma data de acordo com as regras de um sistema de calendário específico. A plataforma Java fornece uma subclasse concreta de *Calendar*: *GregorianCalendar*. Subclasses futuras poderiam representar os vários tipos de calendários lunares em uso em muitas regiões do mundo.

A classe *Calendar* utiliza o método *getInstance()*, para obter um objeto. O método *getInstance()* de *Calendar* retorna um objeto *Calendar* com campos inicializados com a data e hora atual:

```
Calendar agora = Calendar.getInstance();
```

Um objeto *Calendar* pode produzir todos os valores de um instante na linha do tempo em seus campos, necessários para implementar a formatação de data e hora para um determinado idioma e estilo de calendário, como, por exemplo, japonês-gregoriano, japonês tradicional. *Calendar* define o intervalo de valores devolvidos por certos campos, bem como o seu significado. Por exemplo, o primeiro mês do ano tem o valor para o campo MONTH igual a janeiro para todos os calendários.

NumberFormat

NumberFormat é a classe base abstrata para todos os formatos de número. Essa classe fornece a interface para formatação e análise de números. *NumberFormat* também fornece métodos para determinar quais locais têm formatos de números.

NumberFormat ajuda a formatar e analisar os números para qualquer localidade (*Locale*). Seu código pode ser completamente independente das convenções *Locale* para pontos decimais, milhares-separadores, ou até mesmo para a particularidade do uso de dígitos decimais, ou se o formato do número está na base decimal.

Para formatar um número para o Locale atual, use um dos métodos de classe:

```
myString = NumberFormat.getInstance().format(myNumber);
```

Para formatar um número para uma localidade diferente, especifique-o na chamada para `getInstance`.

```
NumberFormat nf = NumberFormat.getInstance(Locale.FRENCH);
```

Você também pode usar um `NumberFormat` para analisar números:

```
myNumber = nf.parse(myString);
```

Use `getInstance` ou `getNumberInstance` para obter o formato do número normal. Use `getIntegerInstance` para obter um formato de número inteiro. Use `getCurrencyInstance` para obter o formato de número de moeda. E usar `getPercentInstance` para obter um formato para apresentar porcentagens. Com `getPercentInstance`, por exemplo, a fração 0,53 será mostrada como 53%.

DecimalFormat

`DecimalFormat` é uma subclasse concreta de `NumberFormat` utilizada para formatar números com casas decimais. Ela tem uma variedade de características projetadas para tornar possível a análise e formatação de números em qualquer localidade (*Locale*), incluindo suporte para o idioma árabe e dígitos sânscritos (língua antiga utilizada na Índia). Ele também suporta diferentes tipos de números, incluindo inteiros (123), de ponto fixo números (123,4), notação científica (1.23E4), porcentagens (12%), e quantidade monetária (R\$ 123).

Para obter um `NumberFormat` para uma localidade específica, incluindo a localidade padrão definida no sistema operacional, chame um dos métodos da classe `NumberFormat`, tal como `getInstance()`. Em geral, não chame os construtores `DecimalFormat` diretamente, já que os métodos de `NumberFormat` pode retornar outras subclasses de `DecimalFormat`.

Um *DecimalFormat* compreende um padrão e um conjunto de símbolos. O padrão pode ser definido diretamente utilizando o método `applyPattern()`, ou indirectamente, utilizando os métodos de API. Os símbolos são armazenados em um objeto `DecimalFormatSymbols`.

Padrão de Caracteres Especiais

Symbol	Significado
0	Dígito
#	Dígito. Suprime a impressão de zeros à esquerda
.	Separador de casas decimais para números fracionários e valores monetários
-	Sinal de menos
,	Separador de agrupamento

Symbol	Significado
E	Separa mantissa e expoente em números com em notação científica
;	Separa subpadrões positivos e negativos
%	Multiplica por 100 e mostra como presencial
\u2030	Multiplica por 1000 e mostra como por milhagem
,	Usado para incluir caracteres especiais em um prefixo ou sufixo, por exemplo, "'#' #" formatos de 123 para "# 123".

Math

A classe *Math* contém uma variedade de funções matemáticas que você pode precisar ocasionalmente, dependendo do tipo de programação que realize.

Para extrair a raiz quadrada de um número, você usa o método `sqrt`:

```
double x = 4;
double y = Math.sqrt(x);
System.out.println(y); // imprime 2.0
```

A linguagem de programação Java não tem nenhum operador para elevar uma quantidade a uma potência: você deve usar o método `pow` da classe *Math*. A instrução

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

Configura `y` como sendo `x` elevado à potência `a` (x^a). O método `pow` tem parâmetros que são de tipo `double` e também retorna um valor `double`.

A classe *Math* fornece as funções trigonométricas normais: `Math.sin`, `Math.cos`, `Math.tan`, `Math.atan`, `Math.atan2` e a função exponencial e sua inversa, o logaritmo natural: `Math.exp`, `Math.log`. Finalmente, existem duas constantes: `Math.PI` e `Math.E`

Exercícios de Fixação 8 – Classes Especializadas

Para que servem as classes especializadas?

Atividade 1 – Classes Especializadas

1. Faça uma classe executável que dada uma string, informe:
 - a. Quantos espaços em branco existem.
 - b. A quantidade de vogais que aparecem e qual o percentual em relação aos caracteres digitados.
2. Faça uma classe executável que dada uma string, construa outra string substituindo a letra S por \$, E por 3, A por 4, G por 6, B por 8, O por * U por #.
3. Faça uma classe executável que verifique se uma cadeia de caracteres é um palíndromo. Uma cadeia de caracteres é um palíndromo se a inversão da ordem de seus caracteres resulta na mesma cadeia. Suponha que todas as letras da cadeia estejam em letras minúsculas. Exemplos: ovo, 1991, amor me ama em roma.

Atenção: Em alguns casos é necessário desconsiderar caracteres de pontuação, acentuação e espaço e, ainda, maiúscula de minúscula, para verificar se uma cadeia é um palíndromo. Exemplo: “Socorram-me, subi no ônibus em Marrocos.”.

Atividade 2 – Classes Especializadas (Complementar)

1. Faça uma classe executável que leia uma senha de no mínimo 6 caracteres e no máximo 12 caracteres, sendo que o primeiro caractere deve ser uma letra e deve conter pelo menos 2 números (em qualquer posição, exceto na primeira). Caso o usuário não forneça uma senha com estas especificações, mostrar a mensagem “senha não é válida” e solicitar que digite uma senha novamente (até ser válida).
2. Faça uma classe executável que informe uma das mensagens a seguir, de acordo com a hora do sistema. – 0h até 5:59h, “Boa madrugada” – 6h até 11:59h, “Bom dia” – 12h até 17:59h, “Boa tarde” e das 18h até 23:59h, “Boa noite”.
3. Faça uma classe executável que dado a data de nascimento de uma pessoa (dia, mês e ano), informe quantos anos ela tem.

4. Faça uma classe executável que dado a data de vencimento de uma fatura (dia, mês, ano) que já venceu (fatura em atraso) e seu valor, informe o número de dias em atraso, considerando a data atual para pagamento, e também o novo valor a pagar, considerando 0,5% de juros por dia de atraso.
5. Para evitar erros de digitação de sequências de números de importância fundamental, como a matrícula de um aluno, o CPF, o número de conta bancária, etc., geralmente se adiciona ao número um dígito verificador. Por exemplo, o número de matrícula 811057 é usado como 8110573, onde 3 é o dígito verificador, calculado da seguinte maneira:
- Cada algarismo do número é multiplicado por um peso começando de 2 e crescendo de 1, da direita para a esquerda: 8×7 , 1×6 , 1×5 , 0×4 , 5×3 , 7×2
 - Somam-se as parcelas obtidas: $56 + 6 + 5 + 0 + 15 + 14 = 96$
 - Calcula-se o resto da divisão desta soma por 11: $96 / 11$ dá resto 8 ($96 = 8 \times 11 + 8$)
 - Subtrai-se de 11 o resto obtido: $11 - 8 = 3$
 - Se o valor encontrado for 10 ou 11, o dígito verificador será 0; nos outros casos, o dígito verificador é o próprio valor encontrado.

Faça uma classe executável que leia uma string representando o número seguido de seu respectivo dígito verificador e informe se o número está correto.

7

Classes e Objetos

Exercício de nivelamento 8 – Classes e Objetos

Você poderia fazer uma breve descrição sobre o que você entende por objetos do mundo real? Cite alguns objetos que você consegue identificar dentro da sala de treinamento.

Programação Orientada a Objetos

A POO é o paradigma de programação predominante nos dias atuais, vindo substituir os paradigmas estruturados. Java é totalmente orientada a objetos. Ela surgiu da ideia de distribuição de funcionalidades e de certas propriedades que são comuns em unidades do mesmo tipo.

Um programa é feito de objetos com certas propriedades e operações que os objetos podem executar. O estado atual, ou seu interior pode variar, mas deve-se tomar cuidado para que os objetos não interajam de forma não documentada. Na POO você só se preocupa com o que o objeto expõe.

A POO inverte a maneira de pensar dos antigos programadores, que sempre pensavam primeiro nos algoritmos e depois nas estruturas de dados. Agora, as estruturas são mais importantes e depois sua manipulação.

A chave da produtividade em POO é tornar cada objeto responsável pela realização de um conjunto de tarefas relacionadas. Se um objeto depender de uma tarefa que não seja de sua responsabilidade, mas sim de outro, deve então pedir para o outro objeto, cujas responsabilidades incluam essa tarefa, realizá-la. Isso é feito por uma versão mais generalizada da chamada de funções usada na programação estruturada: a chamada de métodos.

Em particular, um objeto nunca deveria manipular diretamente dados internos de outro objeto. Isso é chamado de Encapsulamento (também diz respeito ao ocultamento da implementação de alguma tarefa, sendo essa somente visível através da sua interface, o método que a dispara). Toda comunicação deve ser através de mensagens, ou seja, chamada de métodos. Com isso, obtêm-se um alto grau de reusabilidade e pouca dependência entre os objetos.

Classes

Uma classe é um conjunto de variáveis e funções relacionadas a essas variáveis. Uma vantagem da utilização de programação orientada a objetos é poder usufruir o recurso de encapsulamento de informação. Com o encapsulamento o usuário de uma classe não precisa saber como ela é implementada, bastando para a utilização conhecer a interface, ou seja, os métodos disponíveis. Uma classe é um tipo, e, portanto não pode ser atribuída a uma variável. Para definir uma classe, deve-se utilizar a seguinte sintaxe:

```
<modificador_acesso> class Nome_da_classe {  
<atributos>;  
<métodos>;  
}
```

Onde <modificador_acesso> é uma palavra reservada que indica se a classe é visível em todos os sistemas ou não. Podendo ser pública ou privada.

Quando se cria um objeto a partir de uma classe, diz-se que foi criada uma **instância da classe**. Exemplo:

```
CursoJava curso = new CursoJava();
```

Como visto, tudo em Java está dentro de classes. Então, existem disponíveis várias classes em Java para realizar diversas tarefas, como por exemplo, formatação de números para o usuário. Mas mesmo assim, quando se está desenvolvendo um sistema, o programador deve desenvolver suas próprias classes, particulares, para realizar as tarefas próprias da aplicação que está desenvolvendo.

Outro recurso da POO é poder criar classes através de outras. Isso é chamado Herança, e em Java diz-se que uma classe estende a outra, quando ela herda seus dados e métodos. Todas as classes em Java estendem a classe básica `Object`. Inicialmente a classe nova contém todos os métodos e atributos da classe básica, e pode-se escolher por manter ou modificar esses objetos, e adicionar outros novos.

Relações entre Classes

As relações mais comuns entre classes são:

- **Uso:** mais genérica e mais óbvia, por exemplo, uma classe `Pedido` usa (precisa consultar) uma classe `Conta`, para consulta de crédito;
- **Inclusão (tem-um):** por exemplo, usando os pedidos, um objeto `Pedido` contém `Itens`;
- **Herança (é-um):** denota especialização, por exemplo, uma classe `Ranger` será herdeira da classe `Automovel`. A classe `Ranger` tem os mesmos métodos que a classe `Automovel` (porque é um automóvel), e outros especializados para o caso da `Ranger`, que ele representa.

Objetos

Como foi dito anteriormente, classes são tipos, e não podem ser atribuídas a variáveis. Variáveis do tipo de uma classe são chamadas de objetos, e devem ser criadas utilizando o operador `new`, seguindo o exemplo abaixo:

```
variavel = new nome_da_classe;
```

Por exemplo:

```
CursoJava curso = new CursoJava();
```

Onde `curso` é uma variável, uma instância, um objeto da classe `CursoJava`.

Para se trabalhar em POO, deve-se saber identificar três características-chave dos objetos:

- Qual é o comportamento do objeto?
- Qual é o estado do objeto?
- Qual é a identidade do objeto?

Todos os objetos, que são instâncias da mesma classe, têm uma semelhança no seu comportamento que é revelado pelas mensagens que ele aceita.

Em seguida, cada objeto guarda informações sobre o que ele é no momento e como chegou a isso. Isso é o que geralmente se chama de estado do objeto. O estado de um objeto pode mudar com o tempo, mas não espontaneamente. Uma mudança de estado num objeto só pode ser resultado de envio de mensagens, caso contrário, o encapsulamento estaria sendo violado.

Contudo somente o estado do objeto não descreve completamente o objeto, pois cada objeto tem uma identidade distinta. Por exemplo, num sistema de processamento de encomendas, duas encomendas são distintas, mesmo se pedirem a mesma coisa.

Em um programa tradicional, orientado a procedimentos, começa-se o processo de cima, com a função `main`. Ao projetar um sistema POO, não existe topo ou início, e os novatos em POO frequentemente ficam perguntando onde começar. A resposta é: primeiro se encontram ou determinam as classes e depois se acrescentam métodos a cada classe.

Objetos como Argumentos

Pode-se passar como argumento de métodos (funções) objetos de uma determinada classe. Por exemplo:

Na classe `Estacionamento`:

```
static void estacionar(Carro c) {  
    ...  
}
```

Em algum outro ponto do programa:

```
Carro carro = new Carro();  
estacionamento.estacionar(carro);
```

Nesse caso, o objeto `carro` da classe `carro` está sendo passado como parâmetro para o método `estacionar` do objeto `estacionamento`.

Auto referência a um objeto

Em certas ocasiões é necessário que se faça uma referência ao objeto no qual o método está sendo executado, como um todo, para isso existe o `this`. Por exemplo:

```
System.out.println(this);
```

Atributos

Atributos, ou propriedades, são os dados que fazem parte da classe. Para se manter o encapsulamento, os atributos não devem ser visíveis (mas podem), e quando se quer permitir a visualização ou alteração de um deles por outra parte do código, é recomendável que se use métodos de acesso e modificação. Mesmo assim, deve-se tomar cuidado para não escrever um método que retorna um objeto, que é atributo de outro. Pois assim se permitiria a manipulação direta dos dados dentro de um objeto, por outro.

Essa visibilidade dos atributos é dada por modificadores de acesso, como em classes:

- **private**: só é visível dentro da própria classe;
- **public**: é visível em todo o sistema;
- **protected**: só é visível na própria classe, nas classes que estendem dela e nas classes no mesmo pacote.

Por exemplo:

```
public class Evento {  
    private int Id;  
    private ItemEvento Item;  
    public int getId() {  
        return Id;  
    }  
  
    // implementação errada  
    public ItemEvento getItem() {  
        return Item;  
    }  
}
```

Acima, o método `getItem` foi implementado para prover a funcionalidade de alteração da data do item dentro do evento. Não é recomendado porque dá acesso a um dado diretamente, que está dentro da classe. Ao invés disso, prefere-se a seguinte implementação:

```
public class Evento {  
    private int Id;  
    private ItemEvento Item;  
    public int getId() {  
        return Id;  
    }  
}
```

```
}  
public void alterarDataItem(Date d) {  
    ...  
}  
}
```

Como visto, atributos podem ser de qualquer tipo no Java, inclusive outras classes.

Métodos

Métodos são as ações que um objeto provê como interface para o sistema. E não só isso, mas também ações internas, não visíveis para outros objetos. Basicamente são funções, com argumentos, retorno e visibilidade. A visibilidade pode ser `public`, `private` ou `protected`.

Um detalhe importante é que sempre, os atributos de uma classe, podem ser manipulados através de um método dela, obviamente, sejam privados ou públicos.

Os métodos públicos definem a interface da classe, as ações que ela pode executar. Os métodos privados fazem operações internas, que não devem estar disponíveis para os outros objetos. Por exemplo:

```
public class Evento {  
    private int Id;  
    private ItemEvento Item;  
    public int getId() {  
        return Id;  
    }  
    private boolean dataValidaItem(Date d) {  
        ...  
    }  
    public void alterarDataItem(Date d) {  
        if (dataValidaItem(d))  
            ...  
    }  
}
```

O método privado `dataValidaItem` serve como validação interna e quando a classe foi projetada não se tinha a intenção de torná-lo visível para os outros objetos, portanto, foi implementado como `private`.

Getters e Setters

Como já vimos, o ideal é que sejam implementados métodos para disponibilizar acesso aos valores de atributos dos objetos instanciados na aplicação em desenvolvimento. Uma prática bastante comum adotada é a criação de métodos de acesso, que possuem o prefixo *get*, e os métodos modificadores, que possuem o prefixo *set*. Estes métodos nos possibilitam a manutenção de acesso aos atributos privados, encapsulando-os e dando ao programador a

flexibilidade de acessar ou alterar valores de acordo com a necessidade da aplicação em desenvolvimento.

Por exemplo, podemos definir o métodos de acesso *getNome()* e o método modificador *setNome(String n)* para a classe Pessoa. Estes métodos serão utilizados pelo programador sempre que este necessitar consultar ou atualizar o valor do atributo nome de um objetivo instanciado da classe Pessoa.

Sobrecarga

Em Java pode-se ter vários métodos com o mesmo nome. Isso é chamado sobrecarga.

Mas cada um dos métodos tem uma assinatura, que é o que os difere uns dos outros, junto com seu nome. Sua assinatura é os parâmetros que ele espera receber. Por exemplo:

```
public class Evento {
    private int Id;
    private ItemEvento Item;
    public int getId() {
        return Id;
    }
    public void alterarDataItem(Date d) {
        ...
    }
    public void alterarDataItem(int dia, int mes, int ano) {
        ...
    }
}
```

No exemplo acima, para alterar a data do item do evento ou pode-se chamar *alterarDataItem* passando como parâmetro um objeto de data ou o dia, mês e ano, separadamente, da data.

Construtores

Construtores são métodos especiais que servem para fazer a inicialização de um objeto recém instanciado. Esses métodos são chamados toda vez que um objeto novo é construído, isto é, através do operador *new*. Seu nome é o mesmo que o nome da classe. Por exemplo:

```
class Cliente {
    public Cliente (String n) {
        nome = n;
    }
    ...
}
```

Onde *public Cliente* é um construtor da classe *Cliente*.

```
Cliente c = new Cliente("Microsoft");
```


Essa linha criará um objeto da classe `Cliente` e inicialmente chamará o construtor que recebe um `String` como parâmetro.

Existe sempre um construtor padrão que não recebe parâmetro nenhum. Deve-se lembrar que os construtores também permitem sobrecarga.

Blocos de Inicialização

São trechos arbitrários de códigos para fazer a inicialização de um objeto recém instanciado. Esses métodos são chamados toda vez que um objeto novo é construído. Por exemplo:

```
class Cliente {
    public Cliente (String n) {
        nome = n;
    }
    { // bloco de inicialização
        numConta = Conta.getNovoNumero();
    }
    ...
}
```

Esse mecanismo não é comum e não é necessário. Geralmente não é mais claro do que colocar a inicialização dentro de um construtor. Entretanto, para a inicialização de variáveis estáticas, um bloco estático pode ser útil, como será visto mais adiante.

Métodos e Atributos static

O modificador `static` é usado para criar constantes de classe, como vistas anteriormente, mas também é usado para criar atributos e métodos estáticos.

Os atributos estáticos não mudam de uma instância para outra, de modo que podem ser vistos como se pertencessem à classe. Da mesma forma, os métodos estáticos pertencem à classe e não operam nenhum dado de instância, objeto. Isso significa que se pode usá-los sem criar nenhuma instância.

Por exemplo, todos os métodos da classe `Console` são métodos estáticos, assim como todos os métodos da classe `Math`, embutida na linguagem Java. Exemplo:

```
double x = Console.readDouble();
double x = Math.pow(3, 0.1);
```

E a utilização de campos estáticos:

```
Math.PI
System.out
```

Pode-se ter campos e métodos tanto estático quanto não estáticos na mesma classe. As chamadas gerais para métodos e atributos estáticos são:

```
NomeClasse.metodoEstatico(parametros);
NomeClasse.NomeCampoEstatico;
```

Um campo estático é inicializado ou fornecendo-se um valor inicial a ele ou usando um bloco de inicialização estático:

```
public static final double CM_PER_INCH = 2.54;
```

ou

```
static double[] buffer = new double[1024];
static {
    int i;
    for (i=0; i<buffer.length; i++)
        buffer[i] = java.lang.Math.random();
}
```

A inicialização estática ocorre quando a classe é inicialmente carregada. Primeiro todas as variáveis estáticas são inicializadas na ordem que aparecem na declaração da classe.

Um outro exemplo:

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

Como `main` é estática, não é necessário criar uma instância da classe a fim de chamá-la. Por exemplo, se essa função `main` estivesse dentro da classe `Hipotecar`, para iniciar o interpretador Java usa-se:

```
java Hipotecar
```

Então o interpretador simplesmente inicia o método `main` sem criar a classe `Hipotecar`. Por isso, a função `main` só pode acessar dados estáticos da classe. Na verdade, não é raro que a função `main` crie um objeto de sua própria classe:

```
public class Application {
    public static void main(String[] args) {
        Application a = new Application();
        ...
    }
}
```

Exercícios de Fixação 9 – Classes e Objetos

Liste os recursos utilizados para implementar orientação a objetos na linguagem de programação Java.

Descreva a importância da identificação de atributos e métodos para a POO.

Atividade 1 – Classes e Objetos

1. Defina a classe `Empregado`, contendo os atributos `nome` e `salário`. Crie os métodos construtor, seletor e modificador (para cada atributo). Desenvolva as operações para:
 - a. Calcular e retornar o valor de imposto de renda a ser pago, sabendo que o imposto equivale a 5% do salário;
 - b. Faça um programa que crie um objeto da classe `Empregado` (leia o nome e o salário) e informe o nome do funcionário juntamente com o imposto de renda a ser pago.
2. Defina uma classe `Cliente`, contendo os atributos: `nome` e `total de despesas` que a mesma teve num restaurante. Crie os métodos construtor, seletor e modificador (para cada atributo). Desenvolva as operações para:
 - a. Calcular e retornar o valor total a ser pago com gorjeta, considerando a taxa de 10%;
 - b. Faça um programa que crie um objeto da classe `Cliente` (leia o nome e o total de despesas) e informe o nome do cliente e o valor a ser pago com gorjeta.

Atividade 2 – Classes e Objetos (Complementar)

1. Implemente a classe `Triângulo` representada abaixo. Crie um programa que leia os 3 lados do triângulo e diga se ele é equilátero, isósceles ou escaleno, utilizando a interface pública do objeto.

```
Triângulo    int lado1    int lado2          int lado3
              <getters...>, <setters...>
              Triangulo (int l1, int l2, int l3)
```

// **Propriedade:** cada lado de um triângulo é menor do que a soma dos outros dois lados

```
boolean ehEscaleno()    // tem o comprimento de seus três lados diferentes.
```

```
boolean ehEquilatero() // tem o comprimento dos três lados iguais.
```

```
boolean ehIsosceles()   // tem o comprimento de dois lados iguais.
```

2. Defina uma classe `Funcionário`, contendo os atributos: nome, salário-base e nível do cargo (I, II ou III). Crie os métodos construtor, seletor e modificador (para cada atributo). Desenvolva as operações para:

- a. Calcular o adicional por nível de cargo: não há para nível I, 10% para nível II e 15% para nível III;
- b. Calcular o desconto de Imposto de Renda sobre o salário-bruto: não há para salário-bruto até 1.499,99, 7,5% para salário-bruto até 1.999,99, 15% para salário-bruto até 2.999,99 e de 20% para salário-bruto acima de 3.000,00;
- c. Calcular o desconto de Previdência Social: 5% para salário-bruto até 999,99, 9% para salário-bruto até 1.799,99 e de 11% para salário-bruto até 3.500,00. A partir de 3.500,00 é descontado o teto, ou seja, 11% sobre 3.500,00 (que é 385,00).
- d. Mostrar o contracheque: exibe o nome do funcionário, seu salário-base, adicional, IRRF, INSS e o salário-líquido.
- e. Faça um programa que crie um objeto da classe `Funcionário` (leia o nome, o salário-base e o nível do cargo) e mostre seu contracheque.

3. Defina uma classe `Produto`, contendo os atributos: código, descrição, quantidade disponível e quantidade mínima (os atributos de quantidade não podem ser negativos) e também o preço de venda. Crie os métodos construtor, seletor e modificador (conforme necessário). Desenvolva as operações para:

- a. `reporEstoque(quantidade)`: dada uma quantidade comprada, aumentar a quantidade disponível do produto.
- b. `alterarPreco(taxa)`: dada uma taxa, atualizar o preço de venda do produto.
- c. `darBaixa(quantidade)`: dada uma quantidade vendida, diminuir a quantidade disponível do produto.
- d. `precisaRepor`: informa *verdadeiro* caso a quantidade disponível seja menor ou igual a quantidade mínima.

Faça um programa que crie um objeto da classe `Produto` (leia o código, a descrição e a quantidade mínima desejada) e mostre o menu:

1 – Registro da Compra (`reporEstoque`).

2 – Alterar Preço de Venda (`alterarPreco`)

3 – Registro da Venda (dada a quantidade desejada do produto, verificar se há estoque. Se sim, dar baixa e informar o valor total da venda; se não, informar 'estoque insuficiente').

4 – Verificar Reposição (`precisaRepor`)

5 – Fim

Exercícios de Fixação – Classes e Objetos

A proteção de atributos e métodos das classes, fazendo com que estas se comuniquem com o meio externo através da visibilidade permitida, define o conceito de:

- a. especialização
- b. polimorfismo
- c. encapsulamento
- d. herança
- e. agregação

Pacotes

A linguagem Java permite agrupar classes em uma coleção chamada pacote. Os pacotes são convenientes para organizar seu trabalho e para separá-lo das bibliotecas de código fornecidas por outros programadores.

A biblioteca Java padrão é distribuída em vários pacotes, incluindo `java.lang`, `java.util`, `java.net` etc. Os pacotes Java padrão são exemplos de pacotes hierárquicos. Assim como você tem subdiretórios aninhados em seu disco rígido, também pode organizar pacotes usando níveis de aninhamento. Todos os pacotes Java padrão estão dentro das hierarquias de pacote `java` e `javax`.

O principal motivo para se usar pacotes é a garantia da exclusividade dos nomes de classe. Suponha que dois programadores tenham a brilhante ideia de fornecer uma classe `Empregado`. Desde que ambos coloquem suas classes em pacotes diferentes, não haverá conflito. Na verdade, para garantir absolutamente um nome de pacote exclusivo, a Sun recomenda que você use o nome de domínio na Internet de sua empresa (que se sabe que é exclusivo), escrito ao inverso. Então, você usa subpacotes para diferentes projetos. Por exemplo, `cursojava.com` é um exemplo de domínio na internet. Escrito na ordem inversa, ele se transforma no pacote `com.cursojava`. Esse pacote pode então ser ainda subdividido em subpacotes, como `com.cursojava.exemplo.pacotes`.

O único objetivo do aninhamento de pacotes é para gerenciar nomes exclusivos. Do ponto de vista do compilador, não existe absolutamente nenhum relacionamento entre pacotes aninhados. Por exemplo, os pacotes `java.util` e `java.util.jar` não têm nenhuma relação entre si. Cada um deles tem sua própria coleção de classes independente.

Usando pacotes

Uma classe pode usar todas as classes de seu próprio pacote e todas as classes públicas de outros pacotes.

Você pode acessar as classes públicas de outro pacote, de duas maneiras. A primeira é simplesmente adicionar o nome do pacote completo na frente de todo nome de classe. Por exemplo:

```
java.util.Date today = new java.util.Date();
```

Obviamente, isso é maçante. A estratégia mais simples e comum é usar a palavra-chave `import`. O objetivo da instrução `import` é simplesmente fornecer um atalho para você fazer referência às classes do pacote. Quando você usa `import`, não precisa mais fornecer às classes seus nomes completos.

Você pode importar uma classe específica ou o pacote inteiro. Você coloca instruções `import` no início de seus arquivos-fonte (mas abaixo de todas as instruções `package`). Por exemplo, você pode importar todas as classes do pacote `java.util` com a instrução:

```
import java.util.*;
```

Então, você pode usar

```
Date today = new Date();
```

Sem um prefixo de pacote. Você também pode importar uma classe específica dentro de um pacote.

```
import java.util.Date;
```

Importar todas as classes de um pacote é mais simples. Isso não tem nenhum efeito negativo sobre o tamanho do código; portanto, geralmente não há razão para não usar.

Entretanto, note que você pode usar apenas a notação `*` para importar um pacote. Você não pode usar `import java.*` ou `import java.*.*` para importar todos os pacotes com o prefixo `java`.

Atenção: Você só pode importar classes e não objetos. Por exemplo, você nunca importaria `System.out`.

Na maioria das vezes, você importa apenas os pacotes que precisa, sem se preocupar muito com eles. A única vez em que você precisa prestar atenção nos pacotes é quando tem um conflito de nome. Por exemplo, os pacotes `java.util` e `java.sql` têm uma classe `Date`. Suponha que você escreva um programa que importa os dois pacotes.

```
import java.util.*;
```

```
import java.sql.*;
```

Se agora você usar a classe `Date`, então receberá um erro de tempo de compilação:

```
Date today; // ERRO--java.util.Date ou java.sql.Date?
```

O compilador não consegue descobrir qual classe `Date` você deseja. Você pode resolver esse problema adicionando uma instrução `import` específica:

```
import java.util.*;
```

```
import java.sql.*;
```

```
import java.util.Date;
```

E se você realmente precisar das duas classes `Date`? Então, precisará usar o nome completo do pacote, com cada nome de classe.

```
java.util.Date deadline = new java.util.Date();
```

```
java.sql.Date today = new java.sql.Date();
```

Localizar classes em pacotes é uma atividade do compilador. Os códigos de byte nos arquivos de classe sempre usam nomes de pacote completos para fazer referência a outras classes.

Em Java, você pode evitar completamente o mecanismo import, nomeando todos os pacotes explicitamente, como em `java.util.Date`.

A única vantagem da instrução import é a conveniência. Você pode se referir a uma classe por um nome mais curto do que o nome de pacote completo. Por exemplo, após uma instrução `import java.util.*` (ou `import java.util.Date`), você pode se referir à classe `java.util.Date` simplesmente como `Date`.

Adicionando uma classe em um pacote

Para colocar classes dentro de um pacote, você deve colocar o nome do pacote no início de seu arquivo-fonte, antes do código que define as classes no pacote. Por exemplo, o arquivo `Empregado.java` começa como segue:

```
package com.cursojava.empresa;  
public class Empregado  
{  
    . . .  
}
```

Se você não colocar uma instrução `package` no arquivo-fonte, então as classes desse arquivo-fonte pertencerão ao pacote padrão. O pacote padrão não tem nome de pacote. Até agora, todas as nossas classes de exemplo estavam localizadas no pacote padrão.

Você coloca os arquivos de um pacote em um subdiretório que corresponda ao nome do pacote completo. Por exemplo, todos os arquivos de classe do pacote `com.cursojava.empresa` devem estar no subdiretório `com/cursojava/empresa` (`com\cursojava\empresa`, no Windows).

Portanto, o arquivo `Empregado.java` deve estar contido em um subdiretório `com/cursojava/empresa`. Em outras palavras, a estrutura de diretório é a seguinte:

```
.(diretório corrente)  
  TesteEmpregado.java  
com/  
  cursojava/  
    empresa/  
      Empregado.java
```

Para compilar esse programa, basta mudar para o diretório que contém `TesteEmpregado.java` e executar o comando:

```
javac TesteEmpregado.java
```

O compilador encontra automaticamente o arquivo com `/cursojava/empresa/Empregado.java` e o compila.

Alerta: O compilador não verifica diretórios ao compilar arquivos-fonte. Por exemplo, suponha que você tenha um arquivo-fonte que começa com uma diretiva:

```
package com.mycompany;
```

Você pode compilar o arquivo, mesmo que ele não esteja contido em um subdiretório `com/mycompany`. O arquivo-fonte será compilado sem errors, mas a máquina virtual não encontrará as classes resultantes, quando você tentar executar o programa. Portanto, você deve usar a mesma hierarquia para arquivos-fonte e para arquivos de classe.

Pacotes são um recurso interessante em Java para o agrupamento de classes, com funcionalidades relacionadas.

A biblioteca padrão Java é distribuída com inúmeros pacotes incluindo `java.lang`, `java.util`, `java.net`, etc.

Herança

Herança basicamente é a técnica para derivar novas classes a partir das já existentes. Os conceitos por trás da herança são de alterar e reutilizar métodos de classes existentes, a fim de adaptá-los para uma nova situação.

Suponha a seguinte classe:

```
public class Empregado {
    private String nome;
    private double salario;
    public Empregado(String n, double s) {
        nome = n;
        salario = s;
    }
    public void aumentaSalario(double s) {
        salario = salario + s;
    }
    public double getSalario() {
        return (salario);
    }
    public void bonusSalario() {
        aumentaSalario(salario * 0.04);
    }
}
```

Essa é a classe mãe, que diz respeito às regras de tratamento de todos os empregados. Agora imagine que os gerentes, que também são empregados, tenham um tratamento especial.

Poderíamos implementar assim:

```
public class Gerente extends Empregado {
    private String departamento;
```



```
public Gerente(String n, double s) {
    super(n, s);
    departamento = "";
}
public void bonusSalario() {
    super.aumentaSalario(super.getSalario());
}
public String getDepartamento() {
    return departamento;
}
public void setDepartamento(String d) {
    departamento = d;
}
}
```

Claramente percebe-se que os gerentes têm uma informação a mais, o departamento e o tratamento do aumento de salários também é diferente.

Na implementação, a palavra chave `extends` faz com que a classe que está sendo criada derive da classe já existente cujo nome vai após essa palavra. Essa classe existente é chamada superclasse ou classe base. A nova classe é chamada de subclasse, classe derivada ou classe filha.

No construtor da classe `Gerente`, é usado o comando `super`, que se refere a uma superclasse, no caso `Empregado`. De modo que:

```
super(n, s);
```

É um modo sucinto de chamar o construtor da classe `Empregado`. Um detalhe importante é que todo construtor da subclasse tem que chamar o construtor da superclasse também. Caso contrário o construtor padrão será chamado. E se este não existir, um erro será mostrado. O compilador Java insiste que a chamada usando `super` seja a primeira no construtor da subclasse.

Se for analisada a classe `Gerente`, percebe-se que o método `aumentaSalario` não está presente. Isso acontece porque, a menos que especificado, uma subclasse sempre usa os métodos da superclasse.

Já no método `bonusSalario`, o tratamento das informações é diferente para os gerentes. Eles sempre têm bônus no valor do dobro do salário atual. O que não acontece com os funcionários normais, que é de 4%. Por isso, ele foi redefinido, e na sua implementação foram feitas as alterações devidas.

Note também que por ser privada, o atributo `salario` não é visível na classe derivada, então, para que se possa fazer as alterações, deve-se chamar os métodos da classe pai.

Hierarquia de Heranças

A herança não está limitada a um nível somente. Pode-se, por exemplo, fazer com que uma classe chamada `Executivo` derive da classe `Gerente`. Pode-se também fazer uma classe

Secretario e outra Programador derivarem de Empregado. Ficaríamos com a seguinte configuração:

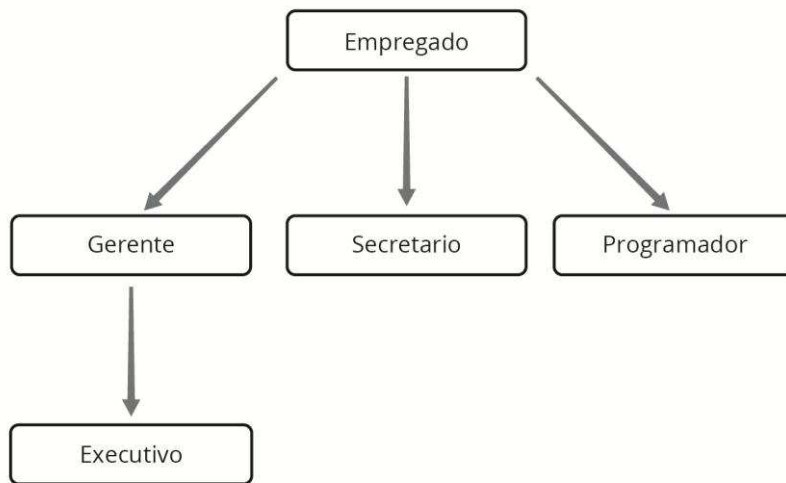


Figura 7.1 – Hierarquia de Heranças.

Evitar Herança

Para evitar herança em alguma classe, deve-se usar o modificador `final` na sua declaração:

```
final class Executivo {  
}
```

Faz com que não se possa definir nenhuma subclasse de `Executivo`. Se o modificador `final` for usado em um método da classe, este também não poderá ser sobreposto nas subclasses. Todos os métodos de uma classe `final` são também `final`.

Atividade 3 - Herança

1. Defina a classe `Funcionário`, contendo os atributos `nome`, `RG` e `total vendas em R$`. Crie o método construtor padrão e os métodos seletores e modificadores para cada um de seus atributos.
2. Defina a classe `Vendedor`, contendo o atributo `valor comissão`. Esta classe é subclasse da classe `Funcionário` e deverá implementar o método construtor padrão e os métodos seletores e modificadores para cada um de seus atributos. Implemente um método que calcule e retorne o valor da comissão do vendedor, sendo que o valor da comissão é de 5% sobre o valor total das vendas.
3. Defina a classe `Gerente`, contendo o atributo `valor comissão`. Esta classe é subclasse da classe `Funcionário` e deverá implementar o método construtor padrão e os métodos seletores e modificadores para cada um de seus atributos. Implemente um método que calcule e retorne o valor da comissão do gerente, sendo que o valor da comissão é de 7,5% sobre o valor total das vendas.
4. Desenvolva uma classe executável que instancie um objeto da classe `Vendedor` e depois calcule e escreva o valor da comissão a qual ele terá direito a receber.
5. Desenvolva uma classe executável que instancie um objeto da classe `Gerente` e depois calcule e escreva o valor da comissão a qual ele terá direito a receber.

Polimorfismo

É a capacidade do Java de descobrir, na hierarquia de herança, qual método usar para uma determinada chamada. Diferentemente da sobrecarga, todos esses métodos possuem a mesma assinatura, mas se encontram em lugares diferentes na hierarquia. Por exemplo, ao se chamar a função `getSalario`, na classe `Gerente`, o Java percebe que essa classe não possui esse método, então procura na superclasse para ver se o encontra. Se encontrado, executa-o, senão um erro é gerado.

Outro exemplo de polimorfismo é o visto abaixo:

```
abstract public class Poligono {
    abstract public void desenhar();
}

public class Retangulo extends Poligono{
    public void desenhar() { .... }
}

public class Icosaedro extends Poligono{
    public void desenhar() { .... }
}

Poligono[100] v;    // array heterogêneo ...
v[15] = new Retangulo();
v[16] = new Icosaedro();
v[15].desenhar();    // Retangulo.desenhar()
```

No exemplo acima, a chamada do método `desenhar`, faz com que seja chamado o método da classe `Retangulo`, apesar de ser um `Poligono` a estar armazenado ali.

Quando não se usa polimorfismo deve-se fazer:

```
Se funcionario = Gerente
    Gerente.aumentarSalarioGerente();
Se funcionario = Programador
    Programador.aumentarSalarioProgramador();
```

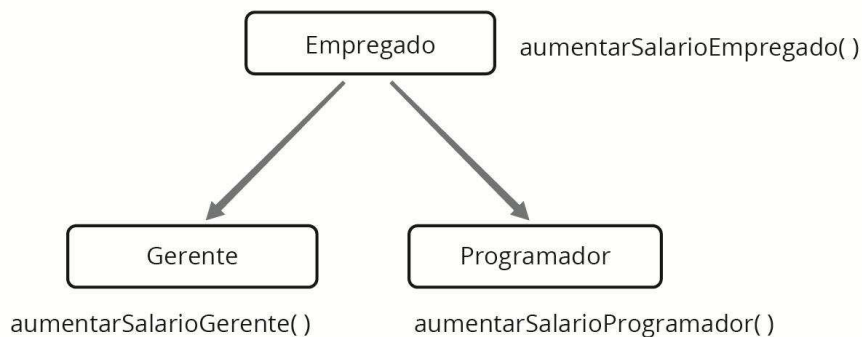


Figura 7.2 – Herança sem o emprego de polimorfismo.

Quando se usa polimorfismo, basta fazer:

```
Empregado.aumentarSalario();
```

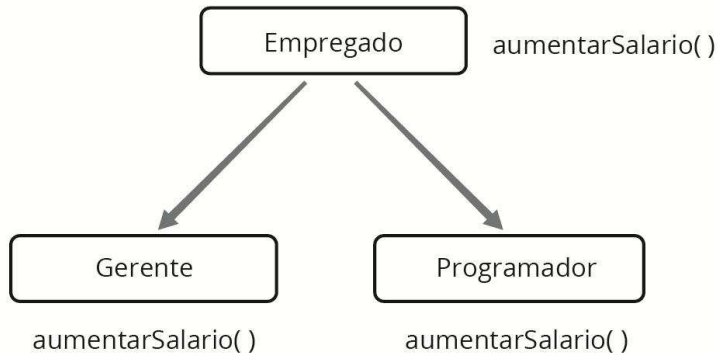


Figura 7.3 – Herança com o emprego de polimorfismo.

Atividade 4 - Polimorfismo

1. Defina a classe `Empregado`, contendo os atributos *nome* e *salário*. Crie o método construtor padrão e os métodos seletores e modificadores para cada um de seus atributos. Implemente o método `aumentarSalario()` que aplique e retorne 5% de aumento no salário.
2. Defina a classe `Programador`. Esta classe é subclasse da classe `Empregado`. Implemente o método `aumentarSalario()` que aplique e retorne 10% de aumento no salário.
3. Defina a classe `Gerente`. Esta classe é subclasse da classe `Empregado`. Implemente o método `aumentarSalario()` que aplique e retorne 15% de aumento no salário.
4. Desenvolva uma classe executável que instancie um objeto de cada uma das classes implementada nos exercícios de 1 a 3 e depois o escreva o nome e o salário que terão direito a receber.

Classes Abstratas

Ao subir na hierarquia de classes, estas vão se tornando mais genéricas. Chega um ponto que elas são tão genéricas que acabam se tornando somente um modelo de como devem ser implementadas.

Suponha um sistema de gerenciamento de mensagens por voz, fax ou texto. Criamos então as classes `VoxMessage`, `FaxMessage` e `TextMessage`. Como o sistema terá que manter e manipular uma mistura desses três tipos de mensagens, então cria-se um classes mãe comum a todas as demais, chamada `Message`.

Mas quando se manda reproduzir uma mensagem, através da classe mãe `Message`, não se sabe o que reproduzir, ou voz, ou fax ou texto. Então se usa a palavra reservada `abstract` para dizer

que esse método não pode ser definido nessa classe, mas sim em uma subclasse. Um detalhe é que se uma classe possui um ou mais métodos abstratos, então ela mesma tem que ser abstrata:

```
public abstract class Message {  
    public abstract void play();  
}
```

Além dos métodos abstratos, as classes abstratas podem ter métodos e atributos concretos. Por exemplo, a classe `Message` pode armazenar o remetente da mensagem postal e ter um método concreto que retorne o nome do remetente:

```
public abstract class Message {  
    private String sender;  
    public Message(String s) {  
        sender = s;  
    }  
    public abstract void play();  
    public String getSender() {  
        return sender;  
    }  
}
```

Não é possível criar objetos a partir de classes abstratas. Ou seja, se uma classe é declarada abstrata, então nenhum objeto dessa classe pode ser criado. Será necessário ampliar essa classe a fim de criar uma instância da classe. Observe que ainda assim podem-se criar variáveis objeto de uma classe abstrata, embora essas variáveis tenham de fazer referência a um objeto de uma subclasse não abstrata. Por exemplo:

```
Message msg = new VoiceMessage("greeting.au");
```

Aqui `msg` é uma variável do tipo abstrato `Message` que faz referência a uma instância da subclasse não abstrata `VoiceMessage`.

Ligação

Dependendo das características da linguagem e do tipo de polimorfismo suportado por ela, a implementação do polimorfismo pode exigir facilidades proporcionadas pela ligação dinâmica. De um modo geral, uma ligação (*binding*) é uma associação, possivelmente entre um atributo e uma entidade ou entre uma operação e um símbolo. Uma ligação é dita estática se ocorre em tempo de compilação ou de interligação (*linking*) e não é mais modificada durante toda a execução do programa. A ligação dinâmica é feita em tempo de execução ou pode ser alterada no decorrer da execução do programa.

Com a ligação dinâmica, a ligação do operador a uma particular operação pode ser feita em tempo de execução, isto é, o código compilado para um tipo abstrato de dados pode ser usado para diferentes tipos de dados, aumentando consideravelmente sua reusabilidade. Entretanto, o uso de ligação dinâmica compromete a eficiência de implementação.

Por default, Java faz ligação dinâmica (*late binding*) dos métodos das classes. Para que um método seja implementado com ligação estática (*early binding*) deve-se torná-la final. Deve-se ter em mente que isso não mais permitirá que um método seja sobrescrito.

Classe Class

A classe `Class` serve para fazer identificação de tipos em tempo de execução. Essa informação acompanha a classe à qual o objeto pertence e é usada pela linguagem para selecionar os métodos corretos a executar em tempo de execução.

Essas informações estão acessíveis para o programador através do método `getClass` da classe `Object`, que retorna uma instância de `Class`.

```
Empregado e;  
...  
Class c1 = e.getClass();
```

Provavelmente o método mais usado da classe `Class` é o `getName`, que retorna o nome da classe.

```
System.out.println(e.getClass().getName());
```

Pode-se obter um objeto `Class` também através de uma string contendo o nome da classe:

```
String className = "Gerente";  
Class c1 = Class.forName(className);
```

Outra maneira é usar o atributo `.class` de um tipo qualquer em Java:

```
Class c1 = Gerente.class;  
Class c2 = int.class;  
Class c3 = Double[].class;
```

Pode-se também criar uma nova instância de uma classe a qualquer momento:

```
e.getClass().newInstance();
```

Cria uma nova instância da mesma classe que `e`, e chama o construtor padrão, aquele sem parâmetros.

No exemplo abaixo, consegue-se criar uma instância de uma classe tendo o seu nome numa string:

```
String s = "Gerente";  
Object m = Class.forName(s).newInstance();
```

Caso se queira chamar outro construtor que não o padrão, deve-se usar o método `newInstance` da classe `Constructor`, que será visto na sessão sobre reflexão.

Reflexão

A classe `Class` fornece um conjunto de ferramentas muito rico e elaborado para escrever programas que manipulam código dinamicamente em Java. Isso é importante quando se deseja manipular recursos de classes, em tempo de execução.

Um programa que consegue analisar recursos de classes dinamicamente é chamado reflexivo. O pacote que dá essa funcionalidade é o `java.lang.reflect`.

Pode-se usar reflexão para:

- Analisar os recursos das classes em tempo de execução;
- Inspeccionar objetos em tempo de execução, por exemplo, escrever um único método `toString` para todas as classes;
- Implementar código genérico para tratamento de arrays;
- Aproveitar os objetos `Method` que funcionam exatamente como ponteiros para funções em C++.

Análise de Recursos das Classes

As três classes do pacote `java.lang.reflect`, `Field`, `Method` e `Constructor` descrevem os campos de dados, as operações e os construtores de uma classe, respectivamente.

Com essas classes pode-se, por exemplo, descobrir os modificadores dos métodos, retornar nomes dos construtores etc.

Análise de Objetos em Tempo de Execução

Através do método `get` da classe `Field`, consegue-se, por exemplo, retornar o valor de um atributo em tempo de execução:

```
Empregado paulo = new Empregado("Paulo Cayres", 1000);
Class c1 = paulo.getClass()
Field f = c1.getField("nome");
Object v = f.get(paulo); // retorna em v a string "Paulo Cayres"
```

Mas como o campo `nome` não é acessível, um erro ocorrerá. Para contornar isso, basta fazer uma chamada a:

```
f.setAccessible(true);
antes de chamar o f.get(paulo).
```

Superclasse abstrata

Suponha o exemplo típico de modelar classes de ordenamento genérico. Vamos inicialmente implementar um algoritmo de ordenação, e uma classe abstrata `Sortable`, contento o método único `compareTo`, que determina se um objeto ordenável é menor, igual ou maior que o outro.

```
abstract class Sortable {
```

```
    public abstract int compareTo(Sortable b);
}

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

Essa rotina pode ser usada em qualquer classe que estende `Sortable`, sobrepondo o método `compareTo`. Isso porque cada classe terá sua maneira de comparar elementos, que será definida nela, e como a classe de ordenação usa `Sortable`, no final o método `compareTo` que será usado é o sobreposto.

Por exemplo:

```
class Empregado extends Sortable {
    ...
    public int compareTo(Sortable b) {
        Empregado eb = (Empregado) b;
        if (salário < eb.salário) return -1;
        if (salário > eb.salário) return 1;
        return 0;
    }
}
```

Perceba que na classe acima, o método certo de comparação foi implementado.

Interfaces

Na linguagem de programação Java, uma interface não é uma classe, mas um conjunto de *requisitos* para classes que queiram se adequar à interface.

Normalmente, o fornecedor de algum serviço diz: “se sua classe obedecer a uma interface em particular, então executarei o serviço.” Vamos ver um exemplo concreto. O método `sort` da classe

Arrays promete ordenar um array de objetos, mas sob uma condição: os objetos devem pertencer às classes que implementam a interface Comparable.

Aqui está como é a interface Comparable:

```
public interface Comparable
{
    int compareTo(Object other);
}
```

Isso significa que qualquer classe que implemente a interface Comparable é obrigada a ter um método compareTo e o método deve receber um parâmetro Object e retornar um inteiro.

Todos os métodos de uma interface são automaticamente public. Por isso, não é necessário fornecer a palavra-chave public ao se declarar um método em uma interface.

É claro que existe um requisito adicional que a interface não pode explicar detalhadamente: ao chamar `x.compareTo(y)`, o método compareTo deve realmente poder comparar dois objetos e retornar uma indicação dizendo se `x` ou `y` é maior. O método deve retornar um número negativo, se `x` for menor do que `y`; zero, se eles forem iguais; e, caso contrário, um número positivo.

Essa interface em particular tem um único método. Algumas interfaces têm mais de um método. Conforme você verá posteriormente, as interfaces também podem definir constantes. O que é mais importante, entretanto, é o que as interfaces *não podem* fornecer. As interfaces nunca têm campos de instância e os métodos nunca são implementados na interface. Fornecer campos de instância e implementações de método é a tarefa das classes que implementam a interface. Você pode considerar uma interface como sendo semelhante a uma classe abstrata sem campos de instância. Entretanto, existem algumas diferenças entre esses dois conceitos.

Agora, suponha que queiramos usar o método sort da classe Arrays para ordenar um array de objetos Empregados. Então, a classe Empregado deve *implementar* a interface Comparable.

Para fazer uma classe implementar uma interface, você precisa executar dois passos:

1. Você declara que sua classe pretende implementar a interface dada.
2. Você fornece definições para todos os métodos na interface.

Para declarar que uma classe implementa uma interface, use a palavra-chave implements:

```
class Empregado implements Comparable
```

É claro que, agora, a classe Employee precisa fornecer o método compareTo. Vamos supor que queiramos comparar os funcionários pelos seus salários. Aqui está um método compareTo que retorna -1, caso o salário do primeiro funcionário for menor do que 0, 0 se for igual e 1, caso contrário.

```
public int compareTo(Object otherObject)
{
    Employee other = (Employee)otherObject;
    if (salary < other.salary) return -1;
    if (salary > other.salary) return 1;
```

```
    return 0;
}
```

Alerta: na declaração da interface, o método `compareTo` não foi declarado como `public`, porque todos os métodos em uma *interface* são automaticamente públicos. Entretanto, ao implementar a interface, você deve declarar o método como `public`. Caso contrário, o compilador vai supor que o método tem visibilidade de pacote, o padrão para uma *classe*. Então, o compilador reclama que você está tentando fornecer um privilégio de acesso mais fraco.

Dica: o método `compareTo` da interface `Comparable` retorna um inteiro. Se os objetos não forem iguais, não importa qual valor negativo ou positivo você retorne. Essa flexibilidade pode ser útil ao se comparar campos inteiros. Suponha, por exemplo, que cada funcionário tenham uma *id* inteira exclusiva e que você queira ordenar pelo número da *ID* do funcionário. Então, você pode simplesmente retornar `id - other.id`. Esse valor será negativo, se o primeiro número de *ID* for menor do que o outro, 0 se eles forem iguais e um valor positivo, caso contrário. Entretanto, existe um problema: o intervalo dos inteiros deve ser suficientemente pequeno para que a subtração não cause estouro. Se você souber que as *IDs* estão entre 0 e $(\text{Integer.MAX_VALUE} - 1) / 2$, pode ficar tranquilo.

É claro que o truque da subtração não funciona para números de ponto flutuante. A diferença `salário / outro.salário` pode ser arredondada para 0, se os salários forem próximos, mas não idênticos.

Existe um problema sério ao usar classes abstratas para expressar uma propriedade genérica, é que em Java, uma classe pode ter somente uma superclasse. Em vez de herança múltipla, o Java introduz o conceito de interfaces.

Interface nada mais é que uma promessa que uma classe vai implementar certos métodos com certas características. Usa-se inclusive a palavra-chave *implements* para indicar que a classe vai manter essas promessas. A maneira como esses métodos serão implementados depende da classe, evidentemente.

A maneira de dizer que será usada uma interface é:

```
class Empregado extends Pessoa implements Comparable ...
```

Como `Comparable` é uma interface:

```
public interface Comparable {
    public int compareTo(Object b);
}
```

Então a classe `Empregado` acima deve implementar o método `compareTo`.

Deve-se perceber que somente métodos (ocos, sem implementação) e constantes (atributos `final`) podem ser definidos dentro de uma interface.

Propriedades

Embora as interfaces não sejam instanciadas com `new`, pode-se ter uma variável da interface:

```
Comparable x = new Empregado(...);
Empregado y = new Empregado(...);
```

```
if (x.compareTo(y) < 0) ...
```

Pode-se também usar `instanceof` para saber se uma classe implementa uma interface:

```
if (anObject instanceof Comparable) { ...}
```

Também é possível estender uma interface para criar outra, permitindo cadeias múltiplas de interfaces:

```
public interface Moveable {
    public void move(double x, double y);
}
public interface Powered extends Moveable {
    public String PowerSource();
}
```

Não se pode colocar campos de instância numa interface, mas pode-se colocar constantes:

```
public interface Powered extends Moveable {
    public String PowerSource();
    public static final int SPEED_LIMIT = 95;
}
```

Classes podem implementar interfaces múltiplas:

```
class Tile extends Rectangle implements Cloneable, Comparable
```

Nota: `Cloneable` é uma interface para implementar o método `clone()`, que faz uma cópia bit a bit dos objetos da classe.

Classes Internas

Classes internas são classes definidas dentro de classes. Há quatro motivos para seu uso:

- Um objeto de uma classe interna pode acessar a implementação do objeto que a criou, incluindo dados que seriam, de outra forma, privados;
- As classes internas podem ser invisíveis para outras classes do mesmo pacote;
- As classes internas anônimas são práticas quando se quer definir callbacks em tempo de execução;
- As classes internas são muito convenientes quando se quer escrever programas orientados por eventos.

Aqui está um exemplo típico: uma classe de lista encadeada define uma classe para conter os links e uma classe para definir a posição de um iterador.

```
class LinkedList
{
    public class Iterator // uma classe aninhada
    {
        public:
```

```
        void insert(int x);
        int erase();
        . . .
    };
    . . .
private class Link // uma classe aninhada
{
    public:
        Link* next;
        int data;
    };
    . . .
};
```

O aninhamento é um relacionamento entre *classes* e não *objetos*. Um objeto `LinkedList` *não* tem sub-objetos de tipo `Iterator` ou `Link`.

Existem duas vantagens: *controle de nome* e *controle de acesso*. Como o nome `Iterator` está aninhado dentro da classe `LinkedList`, ele é conhecido externamente como `LinkedList::Iterator` e não pode entrar em conflito com outra classe chamada `Iterator`. Em Java, essa vantagem não é tão importante, pois os *pacotes* fornecem o mesmo tipo de controle de nome. Note que a classe `Link` está na parte *privada* da classe `LinkedList`. Ela fica completamente oculta de todo outro código. Por isso, é seguro tornar seus campos de dados públicos. Eles podem ser acessados pelos métodos da classe `LinkedList` (que tem a necessidade legítima de acessá-los) e não são visíveis em outro lugar. Em Java, esse tipo de controle não era possível até que as classes internas fossem introduzidas.

Entretanto, as classes internas Java têm um recurso adicional que as torna mais ricas e úteis do que as classes aninhadas em C++. Um objeto proveniente de uma classe interna tem uma referência implícita para o objeto da classe externa que o instanciou. Através desse ponteiro, ele tem acesso ao estado total do objeto externo.

Apenas as classes internas `static` não têm esse ponteiro adicional. Elas são as análogas Java exatas das classes aninhadas em C++.

Por exemplo:

```
class UmaClasse {
    ...
    private class InnerClass {
        ...
    }
    ...
}
```

Pode-se criar classes internas dentro de métodos, e também criar classes anônimas. Classes internas não são um conceito comum e Java as implementa como classes diferentes, inserindo métodos acessadores à classe que contém a classe interna.

Qualquer classe declarada dentro do corpo de outra classe ou interface é conhecido como uma classe aninhada. Se a classe aninhada não tem um nome, então ela é considerada uma classe

anônima. Se a classe aninhada tem um nome, então não é anônima. Se a classe aninhada tem um nome e não é declarada dentro de um método, construtor ou qualquer bloco, então ela é uma classe membro. Se uma classe membro não for declarada como estática, então ela é uma classe interna. Se uma classe não estiver aninhada, então é uma classe de nível superior.

Exercício de Fixação 10 – Classes e Objetos

Qual sua impressão sobre o uso de POO no desenvolvimento de aplicações Java?

Para você, os recursos de POO facilitam a organização, proteção e acesso aos recursos de uma aplicação Java em desenvolvimento?

Atividade 5 – Classes e Objetos

1. Um endocrinologista controla a saúde de seus pacientes através do IMC (Índice de Massa Corporal), calculado por $\text{peso (Kg)}/\text{altura}^2(\text{m})$. Defina uma classe *Pessoa*, contendo os atributos: nome, peso e altura. Crie os métodos construtor, seletor e modificador (para cada atributo). Desenvolva uma operação para:

- a. Determinar e retornar a faixa de risco, considerando a seguinte tabela:

IMC	Faixa de risco
Abaixo de 20	Abaixo do peso
A partir de 20 até 25	Normal
Acima de 25 até 30	Excesso de peso
Acima de 30 até 35	Obesidade
Acima de 35	Obesidade mórbida

- b. Desenvolva uma interface para instanciar um objeto da classe *Pessoa*, calcular e escrever na tela a faixa de risco do mesmo do mesmo.
2. Defina uma classe abstrata *Veículo*, cujos objetos representam veículos, contendo as seguintes informações: número de cilindradas, preço e combustível. Implemente construtor(es), e um método abstrato que possibilite a impressão de todos os atributos do objeto instanciado.
 - a. Defina uma classe *Moto*, subclasse da classe *Veículo*, que contenha as seguintes informações adicionais: tipo de motor (2 ou 4 tempos, carburado, etc.) e tipo de freio (ventilado, a disco etc.). Redefina o construtor de objetos da classe e de impressão de suas informações de maneira apropriada.
 - b. Defina uma classe *Carro*, subclasse da classe *Veículo*, que contenha as seguintes informações adicionais: número de passageiros e capacidade do porta-malas. Redefina o construtor de objetos da classe e de impressão de suas informações de maneira apropriada.
 - c. Defina uma classe *Caminhão*, subclasse da classe *Veículo*, que contenha as seguintes informações adicionais: peso do veículo e peso da carga. Redefina o construtor de objetos da classe e de impressão de suas informações de maneira apropriada.
 - d. Desenvolva uma interface para instanciar um objeto de cada uma das classes implementadas e que imprima uma frase composta por todos os valores de seus atributos próprios e herdados. A classe de interface deverá estar no pacote visão (visão).

Atividade 6 – Classes e Objetos (Complementar)

1. Defina uma classe abstrata *Conta*, cujos objetos representam contas bancárias, contendo as seguintes informações: número da conta e saldo. Defina um construtor de objetos dessa classe, que tenha como parâmetros essas informações, exceto saldo que será igual a zero inicialmente. Defina um método para depósito que irá atualizar o valor do saldo da conta através de um parâmetro passado ao objeto instanciado na interface. Defina também os métodos abstratos para saque.
 - a. Defina uma classe *ChequeEspecial*, subclasse da classe *Conta*, que contenha o valor do limite para o saldo da conta. Redefina o construtor da classe e o método de saque, de forma que possa utilizar o valor do limite, caso seja necessário.
 - b. Desenvolva uma interface para instanciar um objeto da classe *ChequeEspecial*, faça e imprima uma frase composta por todos os valores de seus atributos próprios e herdados. A classe de interface deverá estar no pacote visão (visao).
2. Defina uma classe *Veículo* com os atributos: número da placa, cor, ano de fabricação, tanque (quantidade de combustível em litros), quantidade de litros da reserva e consumo médio (Km/l). Implemente os métodos:
 - ▣ andar, dado o número de Km a percorrer. Atualize a quantidade de combustível no tanque, considerando o consumo médio do veículo e, também, informar quando o veículo entrar no consumo da reserva de combustível e, caso o veículo não tenha mais combustível no tanque, informar que o veículo parou.
 - ▣ abastecer, dado o número de litros de combustível. Atualize a quantidade de combustível no tanque.

- a. Defina duas subclasses, *Caminhão* – com o atributo capacidade de carga (em toneladas) e *Ônibus* – com o atributo capacidade de passageiros (número de lugares). Objetos dessas classes mudam o comportamento de consumo médio (reduzem em 10%) quando sua capacidade de carga supera 50%. Implemente os métodos de *alterarOcupação* (de carga ou pessoas) para cada uma delas e faça o veículo andar novamente.

8

Collections

Exercício de nivelamento 9 – Collections

Pesquise na internet o conceito de *framework* e diga como você acha que este pode auxiliar no desenvolvimento de programas Java.

Definição

Collections, às vezes chamado de container, é simplesmente um objeto que agrupa vários elementos em uma única estrutura. Collections são usados para armazenar, recuperar, manipular e comunicar dados agregados. Normalmente, eles representam itens de dados que formam um grupo natural, como uma mão de poker (uma coleção de cartas), uma pasta de correio (uma coleção de cartas), ou uma lista telefônica (um mapeamento de nomes para números de telefone).

O que é o Java Collections Framework?

O Java Collections Framework é uma arquitetura unificada para representar e manipular coleções de dados. Todas as estruturas das coleções conter o seguinte:

- **Interfaces:** Estes são os tipos de dados abstratos que representam coleções. Interfaces permitem coleções para ser manipulado independentemente dos detalhes de sua representação. Em linguagens orientadas a objeto, interfaces geralmente formam uma hierarquia.
- **Implementações:** Estas são as implementações concretas das interfaces de coleta. Em essência, são estruturas de dados reutilizáveis.
- **Algoritmos:** Estes são os métodos que executam cálculos úteis, como pesquisa e classificação, em objetos que implementam interfaces de coleta. Os algoritmos seriam polimórficos: isto é, o mesmo método pode ser usado em muitas aplicações diferentes da interface. Em essência, os algoritmos são funcionalidades reutilizáveis.

Além do Java Collections Framework, os exemplos mais conhecidos de estruturas Collections são o C++ Standard Template Library (STL) e hierarquia Collections do Smalltalk.

Historicamente, estruturas das coleções têm sido bastante complexo, o que lhes deu uma reputação de ter uma curva de aprendizagem difícil. O Java Collections Framework veio para quebra esta tradição.

Benefícios do Java Collections Framework

O Java Collections Framework oferece os seguintes benefícios:

- **Redução do esforço de programação:** Ao fornecer estruturas úteis de dados e algoritmos, a estrutura Collections libera o programador para concentrar nas partes importantes de seu programa. Ao facilitar a interoperabilidade entre APIs não relacionadas, o Java Collections Framework libera-o de escrever objetos adaptadores ou código de conversão para conectar APIs.
- **Aumenta a velocidade e a qualidade do programa:** Este Collections Framework fornece alto desempenho e alta qualidade de implementações de estruturas úteis de dados e algoritmos. As várias implementações de cada interface são intercambiáveis. Com isso, programas pode ser facilmente ajustado trocando implementações de coleta, livrando o programador do trabalho de escrever suas próprias estruturas de dados, ficando com mais tempo para se dedicar a melhorar a qualidade e desempenho dos programas.
- **Permite a interoperabilidade entre APIs não relacionadas:** As interfaces de coleta são o vernáculo pelo qual passa coleções APIs e para trás. Se a minha administração de rede API fornece uma coleção de nomes de nó e se o seu kit de ferramentas GUI espera uma coleção de títulos de coluna, nossas APIs vão interoperar perfeitamente, mesmo que eles foram escritos de forma independente.
- **Reduz o esforço de aprender e usar novas APIs:** APIs naturalmente tomam coleções na entrada e as fornece como saída. No passado, cada API, tinha uma pequena sub-API dedicada a manipular suas coleções. Houve pouca consistência entre essas coleções ad hoc sub-APIs, assim, você teve que aprender cada um a partir do zero, e era fácil cometer erros quando utilizá-las. Este problema não existe mais com o advento de interfaces padrão de Collections.
- **Reduz o esforço de projetar novas APIs:** Este é o outro lado da vantagem anterior descrita. Designers e implementadores não tem que reinventar a roda cada vez que criar uma API que se baseia em coleções. Em vez disso, eles podem usar interfaces padrão de Collections.
- **Promove a reutilização de software:** Novas estruturas de dados que estejam em conformidade com as interfaces padrão de Collections são, por natureza reutilizável. O mesmo vale para novos algoritmos que operam em objetos que implementam essas interfaces.

Com esses e outros objetivos em mente, a Sun criou um conjunto de classes e interfaces conhecido como Collections Framework, que reside no pacote `java.util`. A API do Collections é robusta e possui diversas classes que representam estruturas de dados avançadas. Por exemplo, não é necessário reinventar a roda e criar uma lista ligada, mas sim utilizar aquela que a Sun disponibilizou.

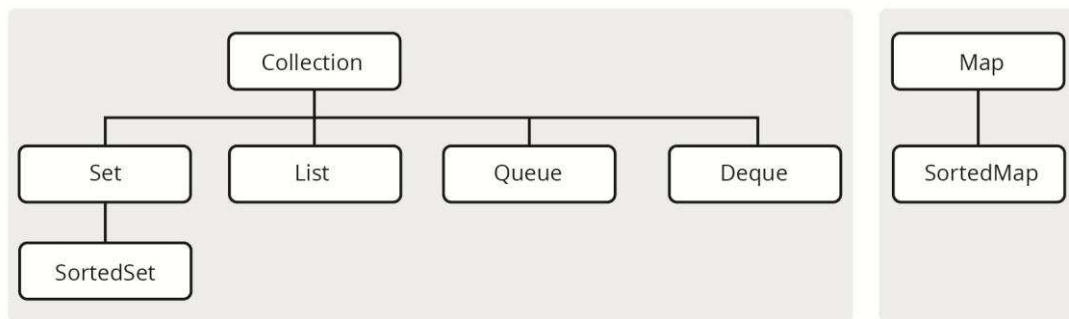


Figura 8.1 – API do Collections.

Listas

Um primeiro recurso que a API de `Collections` traz são as listas. Uma lista é uma coleção que permite elementos duplicados e mantém uma ordenação específica entre os elementos.

Em outras palavras, você tem a garantia de que, quando percorrer a lista, os elementos serão encontrados em uma ordem pré-determinada, definida na hora da inserção dos mesmos. Ela resolve todos os problemas que levantamos em relação ao array (busca, remoção, tamanho "infinito",...). Esse código já está pronto!

A API de `Collections` traz a interface `java.util.List`, que especifica o que uma classe deve ser capaz de fazer para ser uma lista. Há diversas implementações disponíveis, cada uma com uma forma diferente de representar uma lista.

A implementação mais utilizada da interface `List` é a `ArrayList`, que trabalha com um array interno para gerar uma lista. Portanto, ela é mais rápida na pesquisa do que sua concorrente, a `LinkedList`, que é mais rápida na inserção e remoção de itens nas pontas.

A interface `List` possui dois métodos `add`, um que recebe o objeto a ser inserido e o coloca no final da lista, e um segundo que permite adicionar o elemento em qualquer posição da mesma. Note que, em momento algum, dizemos qual é o tamanho da lista; podemos acrescentar quantos elementos quisermos que a lista cresce conforme for necessário.

Toda lista (na verdade, toda `Collection`) trabalha do modo mais genérico possível. Isto é, não há uma `ArrayList` específica para `Strings`, outra para `Números`, outra para `Datas` etc. Todos os métodos trabalham com `Object`.

```

List lista = new ArrayList();
lista.add("Manoel");
lista.add("Joaquim");
lista.add("Maria");
  
```

A interface `List` e algumas classes que a implementam podem ser vistas no diagrama a seguir:

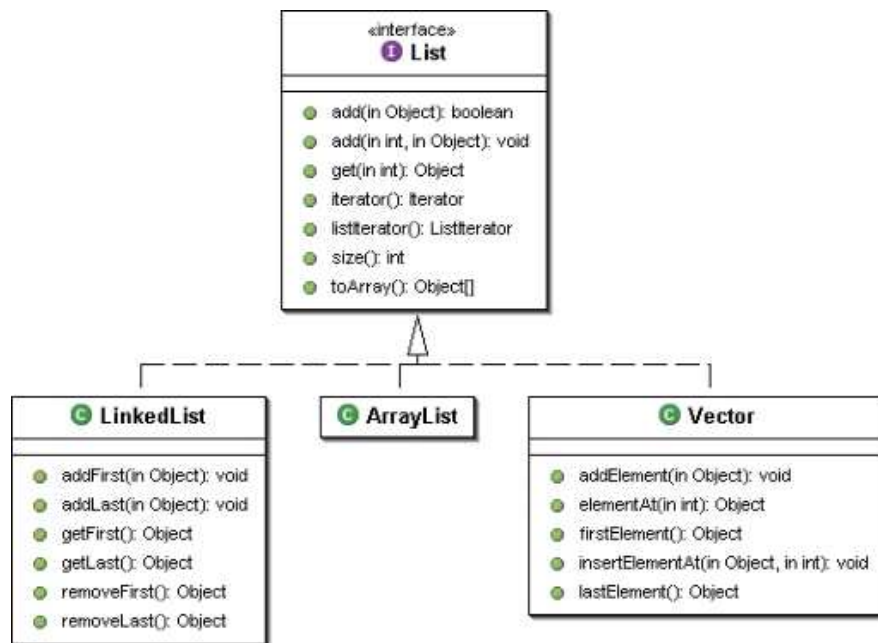


Figura 8.2 – Interface List e classes que a implementam.

Conjunto

Um conjunto (**Set**) funciona de forma análoga aos conjuntos da matemática, ele é uma coleção que não permite elementos duplicados.

Outra característica fundamental dele é o fato de que a ordem em que os elementos são armazenados pode não ser a ordem na qual eles foram inseridos no conjunto. A interface não define como deve ser este comportamento. Tal ordem varia de implementação para implementação.

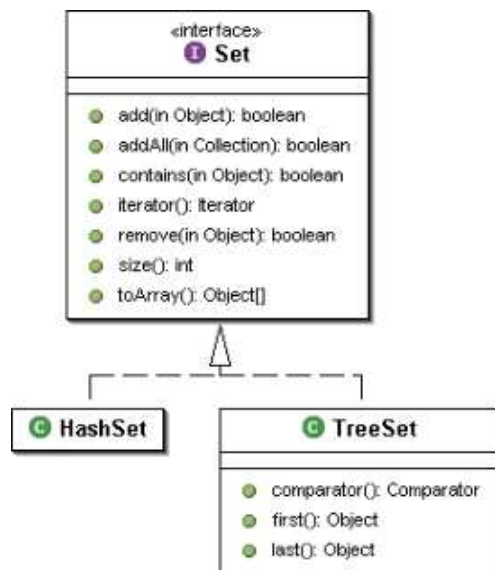


Figura 8.3 – Interface Set e classes que a implementam.

Um conjunto é representado pela interface `Set` e tem como suas principais implementações as classes `HashSet`, `LinkedHashSet` e `TreeSet`.

```
public static void main(String[] args) {
    Set<String> nomes = new HashSet<String>();
    nomes.add("João");
    nomes.add("José");
    nomes.add("Maria");
    nomes.add("Bianca");

    System.out.println("Número de elementos: "+nomes.size());

    if(nomes.contains("Maria")){
        System.out.println("Contém Maria!!!");
    }

    for(String temp : nomes){
        System.out.println(temp);
    }

    Iterator<String> i = nomes.iterator();
    while(i.hasNext())
        System.out.println(i.next());
}
```

O uso de um `Set` pode parecer desvantajoso, já que ele não armazena a ordem, e não aceita elementos repetidos. Não há métodos que trabalham com índices, como o `get(int)` que as listas possuem. A grande vantagem do `Set` é que existem implementações, como a `HashSet`, que possui uma performance incomparável com as `Lists` quando usado para pesquisa (método `contains` por exemplo).

Mapas

Muitas vezes queremos buscar rapidamente um objeto dado alguma informação sobre ele. Um exemplo seria, dada a placa do carro, obter todos os dados do carro. Poderíamos utilizar uma lista para isso e percorrer todos os seus elementos, mas isso pode ser péssimo para a performance, mesmo para listas não muito grandes. Aqui entra o mapa.

Um mapa é composto por um conjunto de associações entre um objeto chave a um objeto valor. `java.util.Map` é um mapa, pois é possível usá-lo para mapear uma chave a um valor, por exemplo: mapeie à chave "empresa" o valor "Nome da Empresa", ou então mapeie à chave "rua" ao valor "Endereço da Empresa". Semelhante a associações de palavras que podemos fazer em um dicionário.

O método `put(Object, Object)` da interface `Map` recebe a chave e o valor de uma nova associação. Para saber o que está associado a um determinado objeto-chave, passa-se esse

objeto no método `get(Object)`. Sem dúvida essas são as duas operações principais e mais frequentes realizadas sobre um mapa.

```
public static void main(String[] args) {  
  
    HashMap<String, String> nomes = new HashMap<String, String>();  
    nomes.put("joao", "João da Silva");  
    nomes.put("jose", "José Matos");  
    nomes.put("maria", "Maria das Dores");  
    nomes.put("bianca", "Bianca Patrícia");  
  
    System.out.println("Número de elementos: "+nomes.size());  
  
    String temp = nomes.get("bianca");  
    System.out.println(temp);  
  
}
```

Um mapa é muito usado para "indexar" objetos de acordo com determinado critério, para podermos buscar esses objetos rapidamente. Um mapa costuma aparecer juntamente com outras coleções, para poder realizar essas buscas!

Ele, assim como as coleções, trabalha diretamente com `Objects` (tanto na chave quanto no valor), o que tornaria necessário o casting no momento que recuperar elementos. Usando os generics, como fizemos aqui, não precisamos mais do casting. Suas principais implementações são o `HashMap`, o `TreeMap` e o `Hashtable`.

Apesar do mapa fazer parte do framework, ele não estende a interface `Collection`, por ter um comportamento bem diferente. Porém, as coleções internas de um mapa são acessíveis por métodos definidos na interface `Map`.

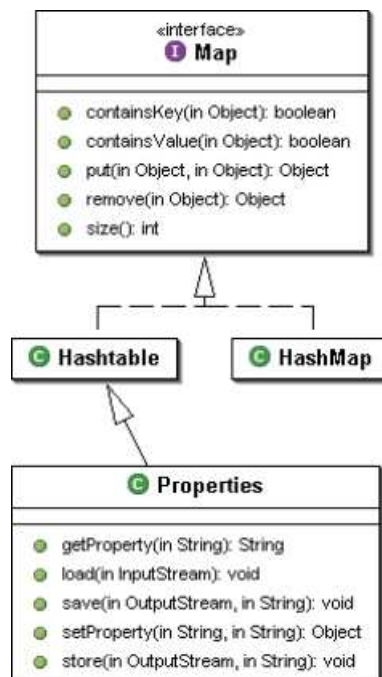


Figura 8.4 – Interface Map e classes que a implementam.

O método `keySet()` retorna um `Set` com as chaves daquele mapa e o método `values()` retorna a `Collection` com todos os valores que foram associados a alguma das chaves.

Exercício de Fixação 11 – Collections

Qual sua impressão sobre o uso de Collections no desenvolvimento de aplicações Java?

Para você, os recursos de Collections facilitam a organização e manipulação de conjuntos de dados em aplicações Java?

Atividade 1 – Collections

1. Desenvolva uma aplicação Java que gere e armazene todos os números inteiros entre 1 e 1000 randomicamente e depois ordene os números gerados em ordem crescente e em ordem decrescente, utilizando um `ArrayList` e implementando interfaces `Comparable`.
2. Crie uma classe `Conta Bancária` contendo os atributos número da conta, nome cliente e saldo atual. Implemente construtores, getters e setters. Desenvolva aplicações Java que apresente as seguintes funcionalidades:
 - a. 1 – Incluir (solicitação dos dados de uma conta bancária em um `ArrayList`);
 - b. 2 – Listar (objetos armazenados no `ArrayList`) em ordem alfabética;
 - c. 3 – Listar (objetos armazenados no `ArrayList`) em ordem decrescente de saldo atual;
 - d. 4 – Pesquisar e mostrar informações de um determinado cliente através do número da conta ou seu nome, utilizando coleções (`HashMap`);

Atividade 2 – Collections (Complementar)

1. Uma empresa decidiu fazer um levantamento de informações em relação aos candidatos que se apresentaram para o preenchimento de vagas no seu quadro de funcionários. Para resolver este problema, é necessário ter os seguintes dados sobre cada candidato: número de inscrição, nome, ‘código da vaga’, a idade (em anos), o sexo (M–masculino ou F–feminino) e a experiência no serviço (S–sim ou N–não). Escreva uma classe principal que leia os dados dos candidatos e armazene em um ArrayList de Objetos da classe Candidato e informe:

- a. O total de candidatos do sexo feminino
- b. A média de idades entre candidatos que tem experiência.
- c. O código da vaga mais concorrida (Interface Comparable).
- d. O percentual de candidatos sem experiência.

Bibliografia

- ASCÊNCIO, Ana Fernanda Gomes; CAMPOS, Edilene Aparecida Vaneruchi de. Fundamentos de Programação de Computadores: algoritmos, pascal, e c/c++. São Paulo: Prentice-Hall, 2002.
- DEITEL. Java Como Programar. 6ª. ed. São Paulo: Pearson Prentice Hall, 2005.
- FARRER, Harry, [Et. Al.]. Algoritmos Estruturados: programação estruturada de computadores. 3ª d. Rio de Janeiro: LTC, 1999.
- FORBELLONE, A. L. V.; EBERSPÄCHER, H. F. Lógica de Programação. São Paulo: Pearson Education/ Makron Books, 1999.
- GUIMARÃES, Ângelo de Moura. Algoritmos e Estruturas de Dados. Rio de Janeiro: LTC, 1994.
- HORSTMANN, Cay. CORNELL, Gary. Core Java 2 – Fundamentals. The Sun Microsystems Press Java Series. Vol. I. 2003.
- JUNIOR, Peter Jandl. Java Guia do Programador. São Paulo: Novatec, 2007.
- MANZANO, J. A. N. G. Algoritmos. São Paulo: Érica, 1997.
- ORACLE. The Java Tutorials. Disponível em: <http://docs.oracle.com/javase/tutorial/>. Acessado em: 29 jan. 2013.
- SIERRA, Kathy; BATES, Bert. Certificação Sun Para Programador Java 6 Guia de Estudo. São Paulo: Rio de Janeiro, 2008.
- SALVETTI, D. D.; BARBOSA, L. M. Algoritmos. São Paulo: Makron Books, 2001.



Java básico 2

Paulo Henrique Cayres

Rio de Janeiro

Escola Superior de Redes

2013

Sumário

Programação para interfaces gráficas, componentes e controle de eventos – sessões 1, 2 e 3..... 3

Introdução.....	4
Janelas	5
Exibição de Informações	7
Objetos Gráficos e o Método <code>paintComponent</code>	7
Texto, Fontes e Cores.....	9
Formas Geométricas.....	10
Imagens	10
Eventos	10
Hierarquia de Eventos do AWT	12
Eventos de Foco	12
Eventos de Teclado	13
Eventos de Mouse	13
Componentes visuais-gráficos do pacote Swing	14
Padrão MVC.....	14
Gerenciamento de Layout de Visualização	15
Bordas.....	17
Painéis	18
Grade	19
Caixa.....	20
Bolsa de Grade	21
Não usar Gerenciador de Layout.....	23
Campos de Entrada de Texto.....	25
Campos de Texto.....	25
Campos de Senha	26
Áreas de Texto.....	26
Seleção de Texto	27
Rótulos.....	27
Caixas de Seleção	28
Botões de Rádio.....	29
Borda	30
Listas	30
Combos.....	32
Menus	34
Caixas de Diálogo	37
Ordem de Travessia: <i>Tab Order</i>	38

Applets – sessão 4.....40

Ciclo de Vida de um Applet.....	41
Um primeiro exemplo de Applet.....	42
A classe Applet.....	42
Invocando um Applet no browser	43
Obtendo parâmetros no Applet.....	45
Passando parâmetros para o Applet.....	46

Applets Java versus Aplicativos.....	47
Conversão de aplicação em Applet:	47
Manipulação de eventos no Applet	48
Exibindo imagens no Applet	50
Reprodução de áudio no Applet	51
Tratamento de Erros – sessão 5.....	56
Descrever Exceções em Métodos.....	57
Levantar uma Exceção	58
Criar Novas Exceções	59
Capturando Exceções	59
Controle de entrada e saída, streams e tratamento de arquivos	
– sessão 6	62
Stream em Java.....	64
Entrada e Saída baseada em Bytes	65
Serialização de Objetos	67
JDBC – sessão 7, 8 e 9	72
Exemplo de Acesso	74
Obtendo Conexões	75
Executando Consultas	75
Executando Alterações	78
Recuperando Resultados	78
Instruções Preparadas	79
Stored Procedures.....	81
Fechando Recursos.....	81
Transações	81
Execução em Batch	82
Threads – sessão 10	86
Múltiplas Linhas de Execução (<i>Thread</i>)	87
O que são?	87
Início do <i>Thread</i> e Programação de <i>Thread</i>	87
Teste de um <i>Thread</i>	90
Suspendendo Threads.....	90
Implementar Runnable Versus Estender Thread	92
As Três Partes de um <i>Thread</i>	92
Implementando uma <i>Thread</i> – Método Mais Comum	93
Tratamento de Regiões Críticas com Threads.....	95
Um Resumo do Funcionamento do Mecanismo de Sincronismo	97
Bibliografia	100

1

Programação para interfaces gráficas, componentes e controle de eventos – sessões 1, 2 e 3

Exercício de nivelamento 1 – Interfaces gráficas

Você já desenvolveu aplicações desktop que utilizaram Interfaces Gráficas com Usuários?
Com qual linguagem de programação?

Você poderia descrever alguns componentes visuais gráficos que utilizou para
desenvolvimento destas interfaces?

▣ Breve histórico:

- ▣ AWT (Abstract Window Toolkit)

Introdução

Quando a linguagem Java 1.0 foi lançada, ela continha somente a biblioteca de classes AWT (Abstract Window Toolkit) para programação GUI (Graphical User Interface) básica. A maneira que o AWT funciona é a delegação de criação de seus componentes GUI nativos. Na verdade, quando a criação de uma caixa de texto era feita, a linguagem criava por trás uma caixa **semelhante** (*peer*) que fazia todos os tratamentos. Dessa maneira conseguia-se rodar os programas em várias plataformas, tendo o programa com o mesmo aspecto (HORSTMANN, 2003).

Em aplicações mais complexas, essa metodologia não era muito produtiva, pois era realmente difícil escrever determinados códigos. Outro problema era que o comportamento de determinados componentes mudava de plataforma para plataforma, ou determinados ambientes não tinham uma vasta biblioteca de componentes gráficos, o que levava a um esforço muito grande para a padronização das aplicações.

Em 1996 a Netscape criou a biblioteca GUI chamada IFC – Internet Foundation Classes – que usava uma metodologia completamente diferente. Aqui, todos os componentes eram desenhados numa tela em branco. Assim, os componentes gráficos se tornavam parecidos com os nativos das plataformas. A Sun trabalhou junto com a Netscape para aprimorar essa biblioteca, e criaram a biblioteca **Swing**.

Assim, o conjunto swing foi tornado o kit de ferramentas GUI oficial da JFC (Java Foundation Classes).

Evidentemente, os componentes do swing são um tanto quanto mais lentos que os criados usando AWT, mas o uso em máquinas mais modernas mostrou que a diferença não é mais tão problemática.

O problema da aparência em relação ao sistema nativo foi resolvido, usando o conceito de *look and feel*, onde se pode escolher a aparência que se quer que o sistema tenha. Mesmo assim, a Sun desenvolveu uma aparência independente de plataforma, chamada **Metal**. A biblioteca swing se encontra em: `javax.swing.*`.

Na versão atual a linguagem Java suporta duas bibliotecas gráficas: a AWT e a Swing. A biblioteca AWT foi a primeira API voltada para interfaces gráficas a surgir no Java. Posteriormente surgiu a biblioteca Swing (a partir do Java 1.2), que possui algumas melhorias com relação à antecessora. De uma forma geral, apesar de reza o senso comum, as bibliotecas gráficas são bastante simples no que diz respeito a conceitos necessários para usá-las.

Os sentimentos de complexidade e dificuldade visualizados no aprendizado de interfaces gráficas em Java é devido ao tamanho das bibliotecas e na enorme gama de possibilidades, as quais certamente assustarão o desenvolvedor de primeira viagem. As bibliotecas AWT e Swing as bibliotecas gráficas oficiais, sempre inclusas em qualquer pacote distribuído pela Sun. Porém, além destas, existem algumas outras bibliotecas de terceiros, sendo a mais famosa o SWT, desenvolvido pela IBM.

Outra questão que vale a pena ser ressaltada é o fato que a linguagem Java, antes de tudo prima pelo lema da portabilidade e as API de interface gráfica do Java não poderiam ser

diferentes. A aparência das janelas construídas com a API Swing é única em todas as plataformas onde roda (Windows, KDE, Gnome). Com isso pode-se afirmar que a aplicação terá exatamente a mesma interface e aparência com pequenas variações geradas por peculiaridades dos sistemas operacionais, como por exemplo, as fontes utilizadas nas janelas. Grande parte da complexidade das classes e métodos do Swing está no fato da API ter sido desenvolvida tendo em mente o máximo de portabilidade possível. Favorece-se, por exemplo, o posicionamento relativo de componentes em detrimento do uso de posicionamento relativo que poderia prejudicar usuários com resoluções de tela diferentes da prevista.

Janelas

Uma janela de mais alto nível, que não está contida em nenhum outro componente, é chamada **quadro (frame)**. Em AWT a classe é `Frame`. Em Swing é `JFrame`. Essa classe não é desenhada dentro de uma **tela de desenho (canvas)**, portanto a moldura toda é desenhada pelo gerenciador de janelas nativo.

Os quadros são exemplos de contêineres, isto é, podem conter componentes de interface de usuário dentro deles. Por exemplo:

```
import javax.swing.*;
class FirstFrame extends JFrame {
    public FirstFrame() {
        setTitle("FirstFrame");
        setSize(300, 200);
    }
}
public class FirstTest {
    public static void main(String[] args) {
        JFrame frame = new FirstFrame();
        frame.show();
    }
}
```

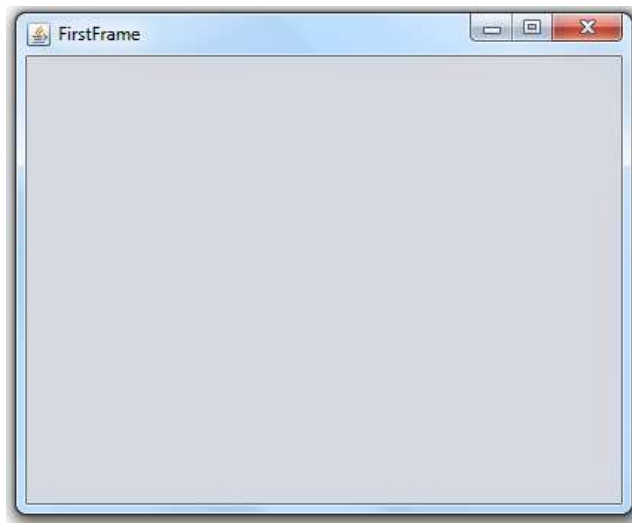



Figura 01 – Exemplo JFrame.

Nesse exemplo, a classe `FirstFrame` estende a classe `JFrame`. Quando a classe `FirstTest` instancia um `FirstFrame` (`new FirstFrame()`) e chama `show()`, esse frame será mostrado, fazendo com que seja aberta uma nova janela.

No construtor de `FirstFrame`, a chamada de `setTitle()` seta o título da janela. A chamada de `setSize()` seta o tamanho da janela, em pixels. Quando se quiser setar a localização dessa janela, pode-se chamar o método `setLocation()`.

Para se encerrar um programa, não adianta somente fechar a janela, deve-se sair do programa e desligar a *virtual machine*. Para isso, deve-se tratar o **evento** de saída do programa. Eventos serão vistos com mais cuidado mais adiante, mas por agora temos que saber como tratar com esse evento básico, o de fechamento de aplicativo.

No modelo de eventos do AWT, deve-se informar ao *frame* **qual é o objeto que quer saber quando ele for fechar**. Esse objeto (o ouvinte, o que tratará o evento) deve ser uma instância de uma classe que implementa a interface `WindowListener`. No nosso exemplo, essa classe será chamada `Terminator`. Em seguida, precisa-se adicionar um objeto da classe `Terminator` ao *frame*, para que ele seja ouvinte da janela.

```
class FirstFrame extends JFrame {  
    public FirstFrame() {  
        Terminator exitOnClose = new Terminator();  
        AddWindowListener(exitOnClose);  
        ...  
    }  
}
```

Mas se for analisada a interface `WindowListener`, percebe-se que existem sete métodos que devem ser sobrepostos:

```
public void windowActivated(WindowEvent e)  
public void windowClosed(WindowEvent e)  
public void windowClosing(WindowEvent e)
```

```
public void windowDeactivated(WindowEvent e)
public void windowDeiconified(WindowEvent e)
public void windowIconified(WindowEvent e)
public void windowOpened(WindowEvent e)
```

No nosso caso, só precisamos tratar o método `windowClosing`. Fica incômodo ter que implementar seis métodos vazios. Para isso existe a classe `WindowAdapter`, que implementa todos esses sete métodos como sendo vazios. Com isso, pode-se só sobrepor os métodos que se quer tratar:

```
class Terminator extends WindowAdapter {
    public void windowClosing(WindowEvent e) {
        System.exit(0);
    }
}
```

No modelo de eventos do Swing, deve-se definir a constante pertencente a classe `WindowConstants` como parâmetro do método `setDefaultCloseOperation()` para que a instância de `JFrame` consiga tratar o evento que correspondente ao fechamento da janela da aplicação em desenvolvimento. A configuração da chamada do método seria `setDefaultCloseOperation(javax.swing.WindowConstants.EXIT_ON_CLOSE);`.

Exibição de Informações

Desenhar uma string diretamente num quadro não é considerada uma boa técnica de programação, apesar de possível. Mas esse componente é projetado para ser um contêiner de outros componentes, estes sim podendo possuir dados.

A estrutura de um `JFrame` é complexa, mas o que nos interessa é a **área de conteúdo**. Essa é a área onde os componentes serão adicionados.

Suponha que se queira inserir no frame um painel (`JPanel`) para se imprimir uma string dentro. O código seria:

```
// obtenção da área de conteúdo
Container painelConteudo = frame.getContentPane();
// criação do componente
JPanel p = new JPanel();
// adição do componente ao frame
painelConteudo.add(p);
```

Objetos Gráficos e o Método `paintComponent`

Sempre que se precisa de um componente gráfico parecido com um componente do swing, pode-se usar herança para criar uma nova classe e depois sobrepor ou adicionar métodos para satisfazer as necessidades.

Para desenhar um painel, deve-se:

- Definir uma classe que estenda `JPanel`;
- Sobrepor o método `paintComponent` dessa classe.

Por exemplo:

```
class MyPanel extends JPanel {  
    public void paintComponent(Graphics g) {  
        // o código de desenho vem aqui  
    }  
}
```

Um detalhe importante é nunca chamar o método `paintComponent` diretamente. Ele é chamado automaticamente toda vez que uma parte do sistema tenha que ser refeita, redenhada. Ao invés, deve-se chamar o método `repaint`.

A exibição do texto se dá por:

```
g.drawString("Hello World", 75, 100);
```

Que deve ser colocado dentro do `paintComponent`. Por exemplo:

```
class MyPanel extends JPanel {  
    public void paintComponent(Graphics g) {  
        super.paintComponent(g);  
        g.drawString("Hello World", 75, 100);  
    }  
}
```

A classe `Graphics` é a responsável por fazer desenhos de texto e imagens, com fontes e cores.

O exemplo acima completo fica:

```
import java.awt.*;  
import java.awt.event.*;  
import javax.swing.*;  
  
class HelloWorldPanel extends JPanel {  
    public void paintComponent(Graphics g) {  
        super.paintComponent(g);  
        g.drawString("Hello World", 75, 100);  
    }  
}  
  
public class HelloWorldFrame extends JFrame {  
    public HelloWorldFrame() {  
        setTitle("Hello World");  
        setSize(300, 200);  
        addWindowListener(new WindowAdapter() {
```

```
        public void windowClosing(WindowEvent e) {
            System.exit(0);
        };
        Container contentPane = getContentPane();
        contentPane.add(new HelloWorldPanel());
    }
}
public class HelloWorld {
    public static void main(String[] args) {
        JFrame frame = new HelloWorldFrame();
        frame.show();
    }
}
```

Texto, Fontes e Cores

Pode-se selecionar fontes diferentes deve-se informar ao objeto `Graphics` qual fonte usar. A classe responsável por criar fontes é `Font`. Por exemplo:

```
Font sans = new Font("SansSerif", Font.BOLD, 14);
```

Onde 14 é o tamanho da fonte. Para setar uma fonte dentro do `paintComponent`, antes de imprimir faz-se:

```
Font sans = new Font("SansSerif", Font.BOLD, 14);
g.setFont(f);
g.drawString("Hello World", 75, 100);
```

Um detalhe importante é que através da classe `FontMetrics` se consegue saber qual é a altura de uma determinada fonte (`Font`) e também o comprimento de uma string usando uma determinada fonte (`Font`). Isso é muito útil quando se quer imprimir várias strings na tela, alinhado, e também não tão dispersos.

Cores também são facilmente alteradas através da classe `Color`. Por exemplo:

```
g.setColor(Color.pink);
g.drawString("Hello", 75, 100);
g.setColor(new Color(0, 128, 128));
g.drawString("World", 75, 125);
```

Para especificar a cor de fundo de um painel, pode-se usar o seguinte exemplo:

```
JPanel p = new MeuPainel();
p.setBackground(Color.white);
contexPane.add(p, "Center");
```

Formas Geométricas

Para desenhar formas geométricas, usam-se os métodos `drawLine`, `drawArc` e `drawPolygon`, do pacote `java.lang.Graphics`. Por exemplo, um polígono:

```
Polygon p = new Polygon();
p.addPoint(10, 10);
p.addPoint(10, 30);
p.addPoint(20, 20);
g.drawPolygon(p);
```

Para retângulos e elipses usa-se `drawRect`, `drawRoundRect`, `draw3DRect` e `drawOval`. Para preencher as formas geométricas uma série de funções são disponibilizadas: `fillRect`, `fillOval`, `fillPolygon`, etc.

Imagens

Para mostrar um gráfico num aplicativo, usa-se a classe `Toolkit`. Por exemplo:

```
String nome = "blue-ball.gif";
Image img = Toolkit.getDefaultToolkit().getImage(nome);
g.drawImage(img, 10, 10, null);
```

Eventos

A manipulação de eventos é ponto imprescindível no desenvolvimento de aplicativos com interface gráfica para o usuário. Frequentemente é dito que a aplicação é **Orientada a Eventos**.

Exemplos de eventos são teclas pressionadas, recebimento de mensagens pela rede, cliques no mouse, etc.

Em Java pode-se controlar totalmente como os eventos são transmitidos desde as **origens** dos eventos (como barras de rolagem ou botões) até os **ouvintes** de eventos. Pode-se designar **qualquer** objeto para ser um ouvinte de um evento, desde que ele o trate convenientemente.

Existem subclasses para cada tipo de eventos, como `ActionEvent` e `WindowEvent`.

Uma visão geral da manipulação de eventos é:

- Um objeto ouvinte é uma instância de uma classe que implementa uma interface especial, chamada naturalmente de **interface ouvinte**;
- Uma origem do evento é um objeto que pode registrar objetos ouvintes e envia a estes os objetos eventos;
- A origem do evento envia objetos eventos para todos os ouvintes registrados quando esse evento ocorre;
- Os objetos ouvintes vão então usar a informação do objeto evento recebido para determinar sua reação ao evento.

O objeto ouvinte é registrado no objeto de origem com um código que segue o modelo a seguir:

```
objetoFonteEvento.addEventListener(objetoOuvinteEvento);
```

Por exemplo:

```
MeuPanel painel = new MeuPanel();  
JButton botao = new JButton("Limpar");  
botao.addActionListener(painel);
```

Aqui, o objeto `painel` será notificado sempre que um evento ocorrer em botão. A classe `MeuPainel` deve possuir uma construção parecida com:

```
public class MeuPainel extends JPanel implements ActionListener {  
    public void actionPerformed(ActionEvent evt) {  
        ...  
    }  
}
```

Sempre que o usuário clicar no botão, o objeto `JButton` cria um objeto `ActionEvent` e chama `painel.actionPerformed`.

Quando se tem muitos botões, que possuem o mesmo ouvinte, se torna necessário saber qual objeto fonte gerou o evento. Existem duas maneiras fáceis de se tratar isso, dentro do método `actionPerformed`:

```
String command = evt.getActionCommand();  
if (command.equals("OK")) ...
```

Que retorna o *label*, o texto do botão que foi impresso. Ou:

```
Object source = evt.getSource();  
if (source == btnOk) ...
```

Que retorna o objeto que gerou o evento.

Hierarquia de Eventos do AWT

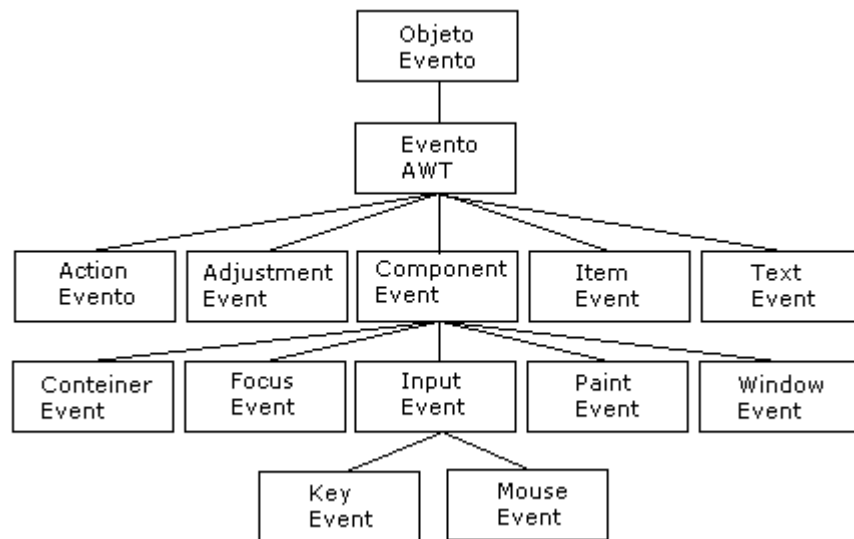


Figura 02 – Hierarquia de eventos AWT.

Vendo a hierarquia percebe-se que o AWT faz distinção entre eventos **semânticos**, que expressam o que o usuário está fazendo, e de **baixo nível**, que tornam os eventos semânticos possíveis. Por exemplo, um clique de um botão, na verdade é uma sequência de outros pequenos eventos, como pressionamento do botão, movimentação do mouse enquanto pressionado e liberação do botão.

Eventos de Foco

Na linguagem Java, um componente tem o **foco** se pode receber pressionamento de teclas. Esse componente pode **receber o foco** ou **perder o foco**. Também pode-se pressionar TAB para mover o foco sequencialmente componente a componente, de cima para baixo, da esquerda para direita.

Pode-se evitar que um componente receba o foco simplesmente sobrepondo o método `isFocusTransversable` para que retorne `false`.

Um ouvinte de foco precisa implementar dois métodos: `focusGained` e `focusLost`. Esses eventos são acionados quando a origem ganhar ou perder o foco, respectivamente. Cada um desses métodos tem um parâmetro `FocusEvent`. Há dois métodos úteis nessa classe: `getComponent` que retorna o componente que recebeu ou perdeu o foco, e `isTemporary` que retorna `true` se a mudança do foco foi temporária. Uma mudança temporária do foco acontece quando um componente perde o controle temporariamente para depois recuperá-lo automaticamente.

Exemplo:

```

public void focusLost(FocusEvent evt) {
    if (evt.getComponent() == ccField && !evt.isTemporary()) {
        If (!checarFormato(ccField.getText()))
  
```

```
        ccField.requestFocus();
    }
}
```

Onde o método `requestFocus` coloca o foco num controle específico.

Eventos de Teclado

Eventos de teclado são ouvidos por classes que implementam a interface `KeyListener`. O evento `keyPressed` informa a tecla pressionada:

```
public void keyPressed(KeyEvent evt) {
    int keyCode = evt.getKeyCode();
    ...
}
```

Outros métodos podem ser sobrepostos como `keyReleased`, que diz quando uma tecla foi liberada.

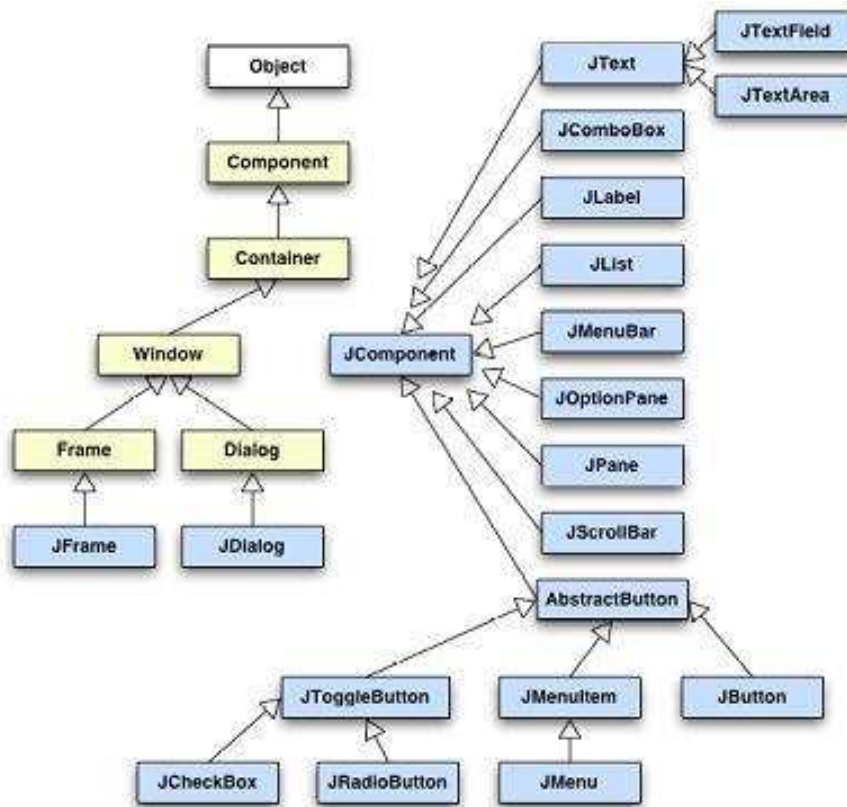
Eventos de Mouse

Não é necessário processar explicitamente os eventos de mouse caso se queira tratar cliques em botões ou menus. Essas são tratadas internamente.

Para o tratamento de eventos de mouse, a classe ouvinte deve implementar a interface de `MouseMotionListener`. Para os eventos de pressionamento dos botões do mouse, deve-se usar um `MouseAdapter`. Por exemplo:

```
public MousePanel() {
    addMouseListener(new MouseAdapter() {
        public void mousePressed(MouseEvent evt) {
            int x = evt.getX();
            int y = evt.getY();
            current = find(x, y);
            if (current < 0) // not inside a square
                add(x, y);
        }
        public void mouseClicked(MouseEvent evt) {
            int x = evt.getX();
            int y = evt.getY();
            if (evt.getClickCount() >= 2) {
                remove(current);
            }
        }
    });
    addMouseMotionListener(this);
}
```


Componentes visuais-gráficos do pacote Swing



Padrão MVC

O MVC é um padrão de projeto (Modelo-Visualização-Controlador ou *Model-View-Controller*), que se baseia nos princípios de projetos orientados a objetos: não construir um objeto responsável por tarefas em demasia. Isto é, num exemplo de componente de botão, não se deve criar um botão como uma única classe que cuida da aparência visual e das particularidades de como um botão funciona.

O padrão de projeto MVC ensina como realizar isso. Deve-se implementar três classes separadas:

- **Model:** o modelo, que armazena o conteúdo;
- **View:** a visualização, que exibe o conteúdo;
- **Controller:** o controlador, que processa entrada de dados do usuário.

O padrão especifica precisamente como os três objetos interagem. O **Modelo** armazena o conteúdo e **não tem interface**. Para um botão, o conteúdo é bem trivial, contém apenas um

conjunto de flags indicando se está pressionado ou não, ativo ou não. O modelo precisa implementar métodos para alterar e ler o conteúdo. Deve-se ter em mente que ele é totalmente abstrato, não possui representação visual.

Uma das grandes vantagens do padrão MVC é que um modelo pode ter várias visualizações, cada uma mostrando partes ou aspectos diferentes do conteúdo. Por exemplo, um editor HTML pode mostrar o conteúdo de duas formas, uma como a página aparece no browser, com opção de edição (WYSIWYG) e outra mostrando as tags todas do arquivo html para edição.

O controlador manipula os eventos de entrada do usuário, como cliques do mouse e pressionamento de teclas. Ele decide se deve traduzir ou não esses eventos em alterações do modelo ou na visualização. Por exemplo, numa caixa de edição de texto, quando um caractere é pressionado, o controlador deve informar ao modelo que este foi alterado. Então o modelo avisa a visualização para atualizar a interface. Mas quando uma tecla de navegação é pressionada, então o controlador somente avisa a visualização, já que o modelo não foi alterado.

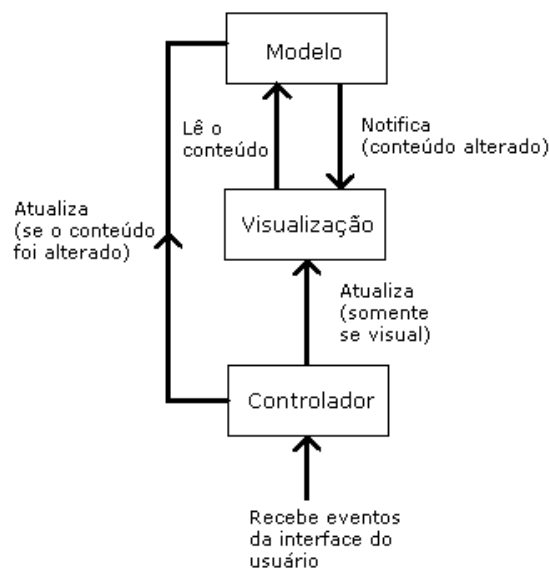


Figura 04 – Interação do padrão MVC.

Gerenciamento de Layout de Visualização

Até agora utilizamos componentes para desenvolvimento de interfaces gráficas sem se preocupar com o modelo de visualização. Isto é, a maneira como estes estão distribuídos e como se apresentam seus controles dentro de um frame.

Usaremos o botão (JButton) por ser um componente visual gráfico comumente utilizado para interação com usuários de programas e também por ser simples de controlar, será usado nos exemplos.

Por exemplo, podemos desenvolver um programa que imprime três botões que alteram as cores de fundo de um frame, quanto clicados pelo usuário. Veja o código java abaixo que pode ser utilizado para implementar esta funcionalidade:

```
// pega o panel
Container contentPane = getContentPane();
// adiciona controles
aux = new JButton("Amarelo");
contentPane.add(aux);
aux.addActionListener(this);

aux = new JButton("Azul");
contentPane.add(aux);
aux.addActionListener(this);

aux = new JButton("Vermelho");
contentPane.add(aux);
aux.addActionListener(this);

contentPane.setLayout(new FlowLayout(FlowLayout.CENTER));
```

Neste exemplo, a última linha seta o layout que será usado na impressão dos controles na tela. Nesse caso, os controles serão centralizados:

```
contentPane.setLayout(new FlowLayout(FlowLayout.CENTER));
```

A função `setLayout` é um método do contêiner, que pode receber qualquer uns dos layouts definidos. Em particular, o `FlowLayout` é usado para alinhar os controles, centralizado, esquerda ou direita.

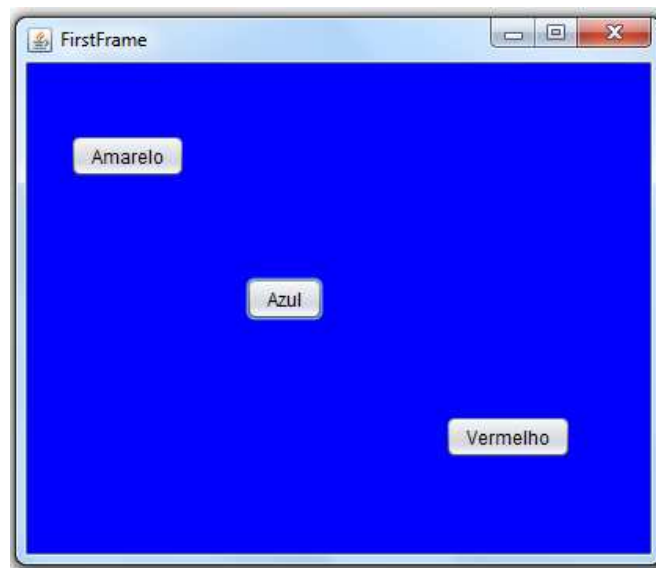


Figura 05 – Exemplo de uso de botões em interfaces gráficas.

Bordas

O gerenciador de layout de visualização de bordas (BorderLayout) é o gerenciador padrão do `JFrame`. Este gerenciador permite escolher em que posição se deseja colocar o controle.



Figura 06 – Exemplo BorderLayout.

Por exemplo:

```
Class MyPanel extends JPanel {
    ...
        jButton1.setText("Norte");
        getContentPane().add(jButton1,
java.awt.BorderLayout.PAGE_START);

        jButton2.setText("Sul");
        getContentPane().add(jButton2,
java.awt.BorderLayout.PAGE_END);

        jButton3.setText("Oeste");
        getContentPane().add(jButton3,
java.awt.BorderLayout.LINE_START);

        jButton4.setText("Leste");
        getContentPane().add(jButton4,
java.awt.BorderLayout.LINE_END);

        jButton5.setText("Centro");
        getContentPane().add(jButton5, java.awt.BorderLayout.CENTER);
    ...
}
```

Os componentes das bordas são colocados primeiro. O que sobra é ocupado pelo centro. Pode-se posicionar como “North”, “South”, “East”, “West” e “Center”. O Layout de borda aumenta todos os componentes para que ocupem todo o espaço disponível. O Layout de Fluxo deixa todos os componentes com seu tamanho preferencial.

Painéis

O problema do Layout de Borda, quanto ao redimensionamento dos componentes para que ocupem o espaço disponível em cada uma das regiões disponíveis, pode ser contornado através do uso de painéis adicionais. Esses painéis agem como contêineres para controles. Dentro desses painéis pode-se utilizar outro layout, podendo assim fazer uma mescla de vários tipos.

Por exemplo:

```
Container contentPane = getContentPane();
Jpanel panel = new JPanel();
panel.add(yellowButton);
panel.add(blueButton);
panel.add(redButton);
contentPane.add(panel, "South");
```

Grade

O gerenciador de layout de visualização de grade (GridLayout) organiza todos os componentes em linhas e colunas como uma planilha. Neste layout as células são todas do mesmo tamanho.

No construtor do objeto layout de grade, deve-se especificar quantas linhas e colunas se quer:

```
panel.setLayout(new GridLayout(5, 4);
```

Pode-se também especificar os espaçamentos horizontal e vertical (respectivamente e em pixels) desejados:

```
panel.setLayout(new GridLayout(5, 4, 3, 3);
```

Adicionam-se componentes começando com a primeira entrada da primeira linha, depois a segunda entrada da primeira linha e assim sucessivamente.

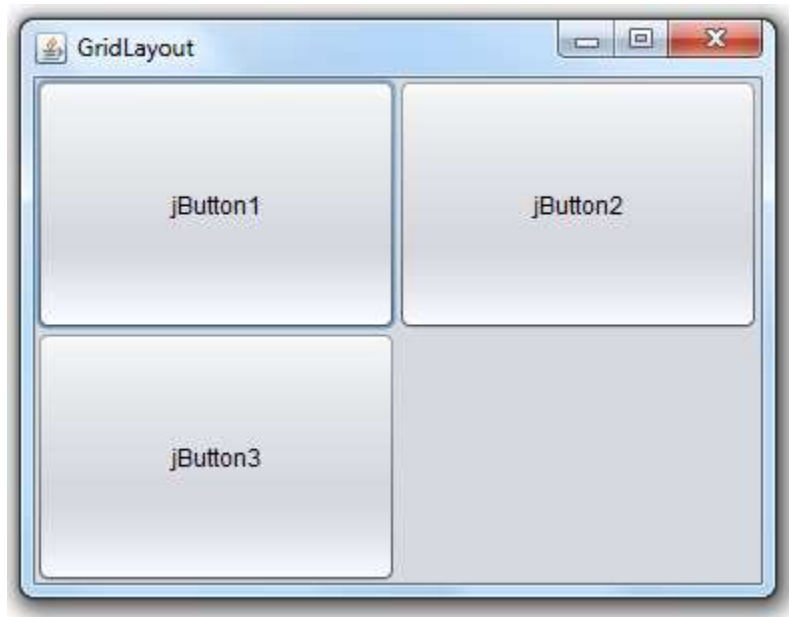


Figura 07 – Exemplo GridLayout.

Caixa

O gerenciador de layout de visualização de Caixa (BoxLayout) permite colocar uma única linha ou coluna de componentes com mais flexibilidade de que o layout de grade. Existe um contêiner (Box) cujo gerenciador padrão de layout é o BoxLayout, diferentemente da classe Panel, cujo gerenciador é o FlowLayout.

Exemplos. Para criar um novo contêiner com layout de caixa, faz-se:

```
Box b = Box.createHorizontalBox();
```

ou

```
Box b = Box.createVerticalBox();
```

Depois, adicionam-se elementos de maneira usual:

```
b.add(okButton);  
b.add(cancelButton);
```

Na caixa horizontal, os componentes são organizados da esquerda para a direita. Na vertical de cima para baixo.

Para não se ter problemas com um controle que dentro de um layout de caixa fique com uma altura não desejada, faz-se com que seu tamanho máximo seja o tamanho preferencial. Por exemplo:

```
textField.setMaximumSize(textField.getPreferredSize());
```

Para se adicionar algum espaço entre, por exemplo, botões OK e Cancel numa tela, pode-se fazer:

```
b.add(okButton);  
b.add(Box.createHorizontalStrut(5));  
b.add(cancelButton);
```

Que adiciona uma **escora** vertical, numa caixa vertical, ou uma escora horizontal numa caixa vertical, separando os componentes em exatamente 5 pixels (no exemplo).

Pode-se usar também uma área rígida invisível, sendo similar a um par de escoras, separando os componentes adjacentes, mas também adicionando uma altura ou largura mínima na outra direção. Por exemplo:

```
b.add(Box.createRigidArea(new Dimension(5, 20)));
```

adiciona uma área invisível com largura mínima, preferencial e máxima de 5 pixels e altura de 20 pixels, mas alinhamento centralizado.

Adicionar **cola** faz o contrário que uma espora, separa os componentes tanto quanto possível.

```
b.add(okButton);  
b.add(Box.createGlue());  
b.add(cancelButton);
```

Bolsa de Grade

O gerenciador de layout de visualização de Caixa (GridBagLayout) é a **mãe** de todos os gerenciadores de layouts. É um layout de grade sem limitações. Pode-se unir células adjacentes para abrir espaço para componentes maiores, etc. Os componentes não precisam preencher toda a área da célula e pode-se especificar seu alinhamento dentro delas.

Mas a sua utilização pode ser bem complexa. É o preço que se paga pela flexibilidade máxima e possibilidade de se trabalhar com vários tipos de situações diferentes.

Para descrever o layout do gerenciador de bolsa de grade, é necessário percorrer o seguinte procedimento:

- Criar um objeto do tipo `GridBagLayout`. Não se informa quantas linhas e colunas a grade tem. Em vez disso, o gerenciador de layout irá tentar descobrir isso a partir das informações fornecidas mais tarde;
- Especificar esse objeto `GridBagLayout` como sendo o gerenciador de layout do componente;
- Criar um objeto do tipo `GridBagConstraints`. Esse objeto irá especificar como os componentes são colocados dentro da bolsa de grade;
- Para **cada componente**, preencher o objeto `GridBagConstraints`. Depois, adicionar o componente com as restrições.

Por exemplo:

```
GridBagLayout layout = new GridBagLayout();
```



```
panel.setLayout(layout);
GridBagConstraints restricoes = new GridBagConstraints();
restricoes.weightx = 100;
restricoes.weighty = 100;
restricoes.gridx = 100;
restricoes.gridy = 100;
restricoes.gridwidth = 100;
restricoes.gridheight = 100;
JList estilo = new JList();
panel.add(estilo, restricoes);
```

Os parâmetros `gridx`, `gridy` especificam a posição (linha e coluna) do componente em relação ao canto superior esquerdo. Os valores `gridWidth` e `gridHeight` determinam quantas colunas e linhas o componente ocupa.

Os campos `weightx` e `weighty` especificam se os campos podem ou não ser ampliados. Se o peso for 0, então o campo terá seu tamanho fixo, nunca será ampliado ou reduzido. Isso quando a janela for redimensionada. Eles não definem tamanho, mas definem a folga que um campo pode ter para ser ampliado.

Os parâmetros `fill` e `anchor` são usados para que um componente não se estenda toda a área. Para `fill`, tem-se quatro possibilidades: `GridBagConstraints.NONE`, `GridBagConstraints.HORIZONTAL`, `GridBagConstraints.VERTICAL`, `GridBagConstraints.BOTH`.

O campo `anchor` é usado quando o componente não ocupa o espaço todo da área. Pode-se **ancorá-lo** em algum espaço dentro do campo. Pode ser: `GridBagConstraints.CENTER`, `GridBagConstraints.NORTH`, `GridBagConstraints.NORTHEAST`, `GridBagConstraints.EAST`, etc.

Para tornar a confecção desse layout mais tranquilo, pode-se usar a seguinte receita:

1. Esboçar o layout do componente numa folha de papel;
2. Encontrar uma grade tal que os pequenos componentes fiquem contidos, cada um, em uma célula, e os componentes grandes ocupem diversas células;
3. Rotular as linhas e as colunas da grade com 0, 1, 2, 3, Com isso, consegue-se os valores de `gridx`, `gridy`, `gridwidth` e `gridheight`;
4. Para cada componente, deve-se saber se ele precisa preencher sua célula horizontalmente ou verticalmente. Caso não, como fica alinhado? Isso determina `fill` e `anchor`;
5. Fazer todos os pesos iguais a 100. Contudo, se for preciso que uma linha ou coluna fique no tamanho padrão, deve-se fazer `weightx` ou `weighty` iguais a 0 em todos os componentes que pertençam a essa linha ou coluna;
6. Escrever código. Conferir cuidadosamente os valores de `GridBagConstraints`, pois um único valor errado poderá arruinar todo o layout.

Não usar Gerenciador de Layout

Para não usar o gerenciador de layout, ou para usar posicionamento absoluto deve-se setar o gerenciador de layout para null e posicionar os controles manualmente:

```
panel.setLayout (null);  
panel.add(ok);  
ok.setBounds (10, 10, 30, 15);
```

Sessão 01 – GUI em Java

Exercícios de Fixação 1 – GUI em Java

O que são gerenciadores de layout? Quais são os gerenciadores de layout do Java?

Atividade 1 – GUI em Java

1. Escreva uma aplicação Swing Java com uma interface que possibilite a digitação de três números e que mostre em uma mensagem (*JOptionPane*) apenas o maior dentre estes. A mensagem será visualizada através de um clique num botão.
2. Escreva uma aplicação Swing Java com uma interface que possibilite a digitação de nome e sobrenome de uma pessoa, separadamente. O programa deverá produzir como mensagem (*JOptionPane*) com o resultado da concatenação dessas duas cadeias de caracteres.
3. Escreva uma aplicação Swing Java com uma interface que possibilite a digitação de dois números inteiros e depois exiba uma mensagem (*JOptionPane*) com o somatório de todos os números inteiros existentes entre os números digitados, inclusive.
4. Escreva uma aplicação Swing Java com uma interface que possibilite a digitação de um número inteiro e depois exiba uma mensagem (*JOptionPane*) com o resultado informando se este número é par ou ímpar.

Atividade 2 – GUI em Java (complementar)

1. Escreva uma aplicação Swing Java com uma interface que possibilite a digitação do nome e idade de uma pessoa. Estas informações serão utilizadas para instanciar um objeto da classe Pessoa através do clique do mouse em um botão. Um segundo botão deverá retornar uma mensagem (*JOptionPane*) se esta pessoa é maior ou menor de idade.
2. Escreva uma aplicação Swing Java que, dada uma palavra ou frase fornecida pelo usuário em uma caixa de texto (*TextField*), determine as seguintes informações que deverão ser exibidas numa área de texto (*TextArea*): número de caracteres, quantidade de vogais, quantidade de constantes, quantidade de dígitos. A operação deverá ser acionada por meio de um clique num botão (*Button*).

Campos de Entrada de Texto

Em Java existem dois tipos de controle para tratamento de texto: **campo de texto** e **área de texto**. A diferença é que campos de texto admitem uma só linha, enquanto áreas de texto admitem várias linhas. As classes são `TextField` e `TextArea`.

Como métodos mais usados tem-se:

- `void setText(String t);` :: seta o texto do controle;
- `String getText();` :: retorna o texto dentro do controle;
- `void setEditable(boolean b);` :: seta se o controle é editável.

Campos de Texto

A maneira usual de inserir um campo de texto é adicioná-lo a um contêiner ou a um painel.

```
JPanel panel = new JPanel();
TextField textField = new TextField("Entrada Padrão", 20);
panel.add(textField);
```

O segundo parâmetro do construtor do `TextField` é o tamanho, no caso 20 colunas. Pode-se alterar o tamanho depois de criado, usando `setColumns(15)`, por exemplo. Logo após, deve-se executar um `validate()`.

Para adicionar um campo em branco:

```
TextField textField = new TextField(20);
```

Para alterar a fonte, usa-se `setFont`.

O tratamento de eventos de alteração de texto pode ser feito através da instalação de um ouvinte na interface `Document`.

```
textField.getDocument().addDocumentListener(ouvinte);
```

Quando um texto é alterado, um dos três métodos a seguir é chamado:

```
void insertUpdate(DocumentEvent e);  
void removeUpdate(DocumentEvent e);  
void changeUpdate(DocumentEvent e);
```

Os dois primeiros métodos são chamados quando caracteres tiverem sido inseridos ou removidos. O terceiro método é chamado quando mudanças como formatação, ocorrem, e também quando alterações no texto são feitas.

Campos de Senha

Para que ninguém veja o que está sendo digitado, usa-se um campo de senha, que no Java é implementado através de `JPasswordField`. Usam-se as seguintes funções:

- `void setEchoChar(char eco);` : informa qual o caractere que será mostrado quando algo foi digitado;
- `char[] getPassword();` : retorna o texto digitado.



Figura 10 – Exemplo `JPasswordField`.

Áreas de Texto

Áreas de texto podem ser criadas através do seguinte exemplo:

```
JTextArea textArea = new JTextArea(8, 40); // 8 linhas e 40 colunas  
GetContentePane().add(textArea);
```

Pode-se setar as linhas e colunas através de `setColumns` e `setRows`. Se houver mais texto a ser mostrado, ele será truncado.

Quando se quer que as linhas maiores sejam quebradas, usa-se `textArea.setLineWrap(true)`. Essa quebra de linha é completamente visual, o texto continua inalterado.

No Swing área de texto não possui barra de rolagem, se a quiser, deve-se usar um **ScrollPane**, como a seguir:

```
TextArea = new JTextArea(8, 40);
JScrollPane scrollPane = new JScrollPane(textArea);
getContentPane().add(scrollPane, "Center");
```

Seleção de Texto

Usam-se os métodos da superclasse `JTextComponent`.

- `void selectAll()` ; : seleciona todo o texto;
- `void select(int inicio, int fim)` ; : seleciona o texto com início e fim especificados;
- `int getSelectionStart()` ; : retorna início da seleção;
- `int getSelectionEnd()` ; : retorna o fim da seleção;
- `String getSelectedText()` ; : retorna o texto selecionado.

Rótulos

São componentes que possuem uma linha de texto simples, que não reagem às opções do usuário. São usados como informativos, ou como nome de campos na tela, etc.

```
JLabel label = new JLabel("Nome: ", JLabel.LEFT);
```

Onde `JLabel.LEFT` é o alinhamento do texto.



Figura 12 – Exemplo JLabel.

Caixas de Seleção

É usado quando se quer uma seleção do tipo **Sim** ou **Não**. As caixas de seleção precisam de um Label para descrever sua finalidade. Por exemplo:

```
JPanel p = new JPanel();
JCheckBox bold = new JCheckBox("Bold");
bold.addActionListener(this);
p.add(bold);
JCheckBox italic = new JCheckBox("Italic");
italic.addActionListener(this);
p.add(italic);
```

Usa-se o método `setSelected` para marcar ou desmarcar uma caixa de seleção.

Quando o usuário clica no **check box** um evento de ação é gerado. Com o método `isSelected` consegue-se descobrir se o controle está ou não selecionado. Por exemplo:

```
public void actionPerformed(ActionEvent evt) {
    Int m = (bold.isSelected() ? Font.BOLD : 0) +
    (italic.isSelected() ? Font.ITALIC : 0);
    panel.setFont(m);
}
```



Figura 13 – Exemplo JCheckBox.

Botões de Rádio

Nas caixas de seleção pode-se selecionar nenhuma, uma ou várias opções. Em várias situações é necessário que o usuário escolha uma dentre várias opções. E quando uma opção é selecionada, as outras são desmarcadas automaticamente. Para isso são usados botões de rádio. Para que isso funcione, usam-se grupos. Isso é conseguido criando-se um controle de grupo e adicionando-se todos os botões de **radio**.

```
smallButton = new JRadioButton("Small", true);
mediumButton = new JRadioButton("Medium", false);
...
ButtonGroup group = new ButtonGroup();
group.add(smallButton);
group.add(mediumButton);
...
```

Novamente, quando o **radio** é pressionado, um evento de ação é gerado. Por exemplo:

```
public void actionPerformed(ActionEvent evt) {
    Object source = evt.getSource();
    If (source == smallButton)
        panel.setSize(8);
    Else if(source == mediumButton)
        panel.setSize(12);
    ...
}
```

Para saber se um botão de rádio está ou não selecionado pode-se usar `isSelected`.



Figura 14 – Exemplo `JRadioButton` e `buttonGroup`.

Borda

São usados para delimitar partes da interface, como diferentes grupos de rádio. Para isso, pode-se seguir os seguintes passos:

1. Chame um método estático de `BorderFactory` para criar uma borda. Pode-se escolher entre os seguintes estilos:
 - a. Relevo abaixado (lowered bevel)
 - b. Relevo destacado (raised bevel)
 - c. Entalhe (etched)
 - d. Linha (line)
 - e. Fosco (matte)
 - f. Vazio (empty)
2. Se quiser, adicione um título à borda, passando-o ao método `BorderFactory.createTitledBorder(String)`.
3. Se você quiser experimentar de tudo, pode-se combinar bordas com `BorderFactory.createCompoundBorder()`.
4. Adicione a borda resultante ao componente usando o método `setBorder` da classe `JComponent`.

Por exemplo:

```
Border etched = BorderFactory.createEtchedBorder();
Border titled = BorderFactory.createTitledBorder(etched, "Titulo");
Panel.setBorder(titled);
```

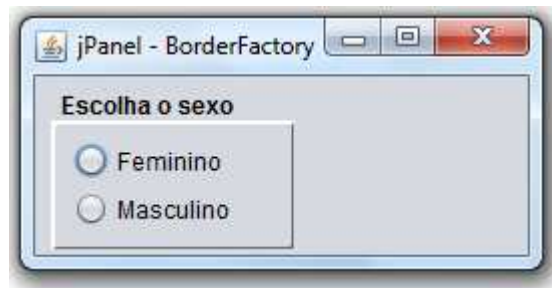


Figura 15 – Exemplo JPanel e BorderFactory.

Listas

Listas são usadas para escolhas com várias opções. O Swing oferece um objeto muito completo para se gerenciar esse tipo de lista, permitindo tanto strings quanto objetos quaisquer.

O componente `JList` é usado para manter um conjunto de valores dentro de uma só caixa para seleção, permitindo tanto seleções únicas quanto múltiplas. Exemplo:

```
String[] words = {"quick", "brown", "hungry", "wild"};
JList wordList = new JList(words);
```

As listas não possuem barra de rolagem automaticamente, para isso deve-se adicioná-la dentro de um painel de rolagem:

```
JScrollPane scrollPane = new JScrollPane(wordList);
```

Depois, deve-se adicionar o painel de rolagem, e não a lista, no painel que a conterá.

Para fazer com que um número definido de elementos seja mostrado a cada momento, usa-se:

```
wordList.setVisibleRowCount(10); // mostra 10 itens
```

Por default a lista permite múltipla seleção, para usar seleção única, usa-se:

```
wordList.setSelectionModel(ListSelectionModel.SINGLE_SELECTION);
```

O método de notificação, tratamento de eventos, não se dá através do monitoramento de eventos de ação, mas sim eventos de seleção de lista. Deve-se implementar o método:

```
public void valueChanged(ListSelectionEvent evt)
```

no ouvinte, e a interface é `ListSelectionListener`.

Por exemplo:

```
class ListFrame extends JFrame implements ListSelectionListener {
    public ListFrame() {
        ...
        JList wordList = new JList(words);
        wordList.addListSelectionListener(this);
    }
    public void valueChanged(ListSelectionEvent evt) {
        JList source = (JList)evt.getSource();
        Object[] values = source.getSelectedValues();
        String text="";
        for (int i=0; i<values.length; i++)
            text += (String)values[i] + " ";
    }
}
```

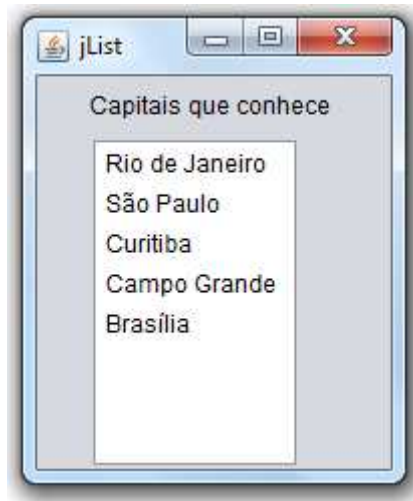


Figura 16 – Exemplo jList.

Combos

É uma lista que fica escondida, e quando o usuário clica no campo, a lista se abre. Se a caixa estiver configurada para ser editável, pode-se editar a seleção como se fosse uma caixa de texto.

```
style = new JComboBox();
style.setEditable(true);
style.addItem("Serif");
style.addItem("SansSerif");
...
```

Esse método (`addItem`) adiciona a string no final da lista., pode-se também inserir em qualquer lugar:

```
style.insertItemAt("Monospaced", 0);
```

Para remover itens usa-se:

```
style.removeItem("Monospaced");
style.removeItemAt(0); // remove o primeiro item
style.removeAllItems();
```

Para tratar dos eventos, captura-se um evento de ação, logo, tratando-se o método `actionPerformed`:

```
public void actionPerformed(ActionEvent evt) {
    JComboBox source = (JComboBox) evt.getSource();
    String item = (String) source.getSelectedItem();
    panel.setStyle(item);
}
```

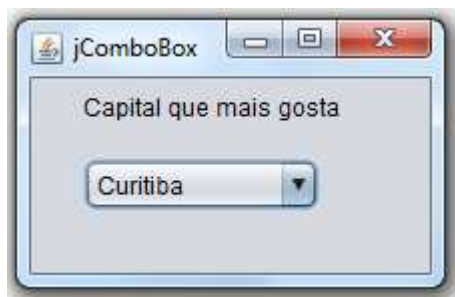


Figura 17 – Exemplo jList.

Sessão 02 – GUI em Java

Exercícios de Fixação 2 – GUI em Java

Liste os recursos que o pacote swing do java disponibiliza para desenvolvimento de interfaces gráficas que você aprendeu até o momento.

Atividade 1 – GUI em Java

1. Escreva uma aplicação Swing Java com uma interface que tenha duas caixas de listas JList, do mesmo tamanho, uma ao lado da outra, denominadas lista 01 (a esquerda) e lista 02 (a direita). Implemente uma entrada de texto (JTextField) que permitirá a digitação de palavras que serão adicionadas apenas à lista 01. Por meio de botões JButton deve ser possível: mover os itens selecionados da lista 01 para a lista 02; mover todos os itens da lista 01 para a lista 02; mover os itens selecionados da lista 02 para a lista 01; e mover todos os itens da lista 02 para lista 01. Por fim, implemente um botão que possibilite remover os itens selecionados de cada uma das listas (um botão para cada lista).
2. Escreva uma aplicação Swing Java com uma interface que permita a seleção (JComboBox) de uma entre 5 opções de cores que serão utilizadas para preencher a cor de fundo do JFrame onde a caixa de seleção foi disponibilizada.
3. Escreva uma janela JDialog que exiba uma pergunta (em um rótulo JLabel) e cinco alternativas de resposta (usando botões de opções de JRadioButton). A pergunta e suas alternativas devem ser especificadas por métodos próprios, bem como a resposta correta. Apenas uma alternativa poderá ser selecionada. Por meio de um botão JButton, a aplicação deverá avaliar se a resposta está ou não correta, impedindo a partir de seu

acionamento a alteração da resposta. Como opção ajustável, a janela deve exibir uma mensagem de diálogo (com JOptionPane) com a indicação de acerto ou erro.

4. Escreva uma aplicação Swing Java que implemente uma sequência de questões a serem apresentadas pela janela implementada no exercício 3. Esta aplicação deverá calcular e escrever uma mensagem (JOptionPane) com o total de acertos e erros obtidos.

Atividade 2 – GUI em Java (complementar)

1. Escreva uma aplicação Swing Java que implemente uma interface onde o usuário deverá informar seu usuário (JTextField) e senha (JPasswordField) e, ao clicar no botão de acesso (JButton), seja feita uma verificação um array com a lista de usuários e senhas que tenham acesso ao sistema. A mensagem **ACESSO PERMITIDO** deverá ser exibida caso seja encontrado um usuário com a senha informada. Exiba a mensagem **ACESSO NEGADO** caso não seja encontrado o usuário e senha informados no array de usuários.

Menus

Não é necessário um gerenciador de layout para se colocar um menu numa janela. Sua elaboração é direta:

```
JMenuBar menuBar = new JMenuBar();
```

Uma barra de menu pode ser colocada onde se queira, mas normalmente aparece na parte superior do frame. Usa-se:

```
frame.setJMenuBar(menuBar);
```

Para cada menu cria-se um objeto menu:

```
JMenu editMenu = new JMenu("Edit");
```

Então se adicionam os itens, separadores e submenus:

```
JMenuItem pasteItem = new JMenuItem("Paste");
editMenu.add(pasteItem);
editMenu.addSeparator();
...
```

Também se adiciona menus de nível mais alto à barra de menus:

```
menuBar.addMenu(editMenu);
```

Quando o usuário seleciona um menu ocorre um evento de ação. É necessário instalar um ouvinte de evento de ação para cada item do menu:

```
pasteItem.addActionListener(this);
```

A geração desse código todo é um tanto quanto trabalhosa, por exemplo:

```
JMenu menu = new JMenu("Edit");
item = new JMenuItem("Cut");
```

```
item.addActionListener(this);
menu.add(item);
item = new JMenuItem("Copy");
item.addActionListener(this);
menu.add(item);
item = new JMenuItem("Paste");
item.addActionListener(this);
menu.add(item);
menuBar.add(menu);
```

Para responder a eventos de menu usa-se `actionPerformed`:

```
public void actionPerformed(ActionEvent evt) {
    if (evt.getSource() instanceof JMenuItem) {
        String arg = evt.getActionCommand();
        if (arg.equals("Open")) ...
        else if (arg.equals("Save")) ...
        ...
    }
}
```

As teclas de atalho podem ser especificadas no construtor dos itens de menu:

```
JMenuItem cutItem = new JMenuItem("Cut", 'T');
```

Para teclas de atalho para o menu principal, deve-se usar o seguinte:

```
JMenu helpMenu = new JMenu("Help");
HelpMenu.setMnemonic('H');
```

Para acessar as teclas de atalho usa-se `ALT`+ letra.

Quando se quer teclas de atalho, do tipo `CTRL`+`S` para salvar, isto é, para que itens de menu sejam chamados, sem sequer abrir o menu onde estão, então usa-se `setAccelerator`.

```
openItem.setAccelerator(KeyStroke.getKeyStroke(KeyEvent.VK_O,
InputEvent.CTRL_MASK));
```

Para inserir ícones nos itens de menu, usa-se o seguinte construtor:

```
JMenuItem cutItem = new JMenuItem("Cut", new ImageIcon("cut.gif"));
```

Para criar itens de menu usando botões de rádio e *check boxes*, usa-se outros objetos:

```
JCheckBoxMenuItem readOnlyItem = new JCheckBoxMenuItem("Read-only");
OptionsMenu.add(readOnlyItem);
```

Os itens de menu em forma de botões de rádio funcionam exatamente como controles comuns, isto é, deve-se criar um grupo para eles, para possuam o efeito esperado:

```
ButtonGroup group = new ButtonGroup();
JRadioButtonMenuItem insertItem = new JRadioButtonMenuItem("Insert");
```

```
insertItem.setSelected(true);
JRadioButtonMenuItem overtypeItem = new
JRadioButtonMenuItem("Overtime");
group.add(insertItem);
group.add(overtimeItem);
optionsMenu.add(insertItem);
optionsMenu.add(overtimeItem);
```

Menus flutuantes são facilmente implementados também:

```
JPopupMenu popup = new JPopupMenu();
```

Os itens são adicionados da forma usual:

```
JMenuItem item = new JMenuItem("Cut");
item.addActionListener(this);
popup.add(item);
```

Para abrir explicitamente um menu flutuante usa-se o método `show`.

```
popup.show(painel, x, y);
```

Onde `painel` é o controle que será pai no menu, `x` e `y` é a localização.

Os menus flutuantes normalmente estão associados ao pressionamento de um botão particular do mouse, como no Windows. Para que isso aconteça, deve-se:

```
public void mouseReleased(MouseEvent evt) {
    if (evt.isPopupTrigger())
        popup.show(evt.getComponent(), evt.getX(), evt.getY());
}
```

Para habilitar e desabilitar itens de menu, pode-se usar:

```
saveItem.setEnabled(false);
```

Mas isso pode levar o código a ficar cheio de instruções lidando com os menus. Então pode-se usar um tratador de eventos de menu. Existe o `MenuListener`, com três métodos:

```
void menuSelected(MenuEvent evt); // chamado antes do menu ser
exibido
void menuDeselected(MenuEvent evt); // depois que o menu é removido da
tela
void menuCanceled(MenuEvent evt); // qdo a seleção é cancelada
```

Caixas de Diálogo

São janelas pré-definidas que são usadas para informação para o usuário. Existem dois tipos: **modais** que quando aparecem, o usuário não pode fazer mais nada antes de fechá-la e as **não modais** que o usuário pode continuar o que está fazendo.

Algumas janelas pré-definidas são:

- `JOptionPane.showMessageDialog` : Exibe uma mensagem e espera o usuário clicar OK;
- `JOptionPane.showConfirmDialog` : Exibe uma mensagem e obtém confirmação (OK/Cancelar);
- `JOptionPane.showOptionDialog` : Exibe uma mensagem e obtém uma opção do usuário dentre várias;
- `JOptionPane.showInputDialog` : Exibe uma mensagem e obtém uma linha digitada pelo usuário.

Para se criar uma janela de diálogo própria, como a janela **sobre**, deve-se derivar uma classe de `JDialog`.

Ordem de Travessia: *Tab Order*

Quando uma janela possui muitos componentes, deve-se pensar na ordem de travessia desses componentes, isto é, a ordem na qual os componentes perdem e ganham o foco enquanto o usuário pressiona TAB.

O Swing tenta percorrer esses elementos de uma forma coerente, de cima para baixo, da esquerda para direita, pulando os rótulos. No Swing, a ordem de inserção dos elementos não influencia no *tab order*, mas sim o layout usado.

O problema ocorre quando um contêiner contiver outros contêineres e assim por diante. Quando o foco vai para outro contêiner, automaticamente ele é colocado no componente mais acima e mais à esquerda desse contêiner, para depois percorrer os outros componentes.

Para alterar essa ordem pode-se usar duas formas. A primeira é chamando:

```
okButton.requestFocus();
```

Mas só funciona como resposta a um dado evento, isto é, não tem efeito permanente na ordem de travessia.

A segunda é explicitamente mudar a ordem de travessia, chamando:

```
style.setNextFocusableComponent(size);
```

Que faz com que o próximo componente a receber o foco, depois do `style` seja o `size`.

Sessão 03 – GUI em Java

Exercícios de Fixação 3 – GUI em Java

O que são componentes e o que são os event listeners?

Atividade 1 – GUI em Java

1. Construa uma aplicação que permita efetuar um teste de cor, por meio de três controles deslizantes JSlider, onde o usuário poderá ajustar uma cor por meio de seus componentes RGB expressos com valores inteiros entre 0 e 255. A cor a ser ajustada deve ser visualizada por meio de um JPanel sem conteúdo.

2. Escreva uma aplicação Swing Java que permita a construção de uma árvore com o componente JTree. Uma caixa de texto JTextField deve fornecer os valores para os nós da árvore. O primeiro nó adicionado constituirá sua raiz. A partir daí, os novos nós deverão ser adicionados como filhos do nó que esteja selecionado na árvore no momento da nova inserção. A inserção de um novo nó será feita por meio do clique num botão JButton. Implemente também um botão para remover um nó que esteja selecionado. A exclusão deverá excluir todos os nós filhos pertencentes ao nó selecionado quando do clique no botão de exclusão.

Atividade 2 – GUI em Java (complementar)

3. Construa uma aplicação Java que permita a seleção (JFileChooser) e visualização de uma imagem em formato GIF, JPG ou PNG. A seleção do arquivo e o encerramento da aplicação devem constituir opções de um menu suspenso (JMenu) denominado arquivo e uma barra de menus (JMenuBar).

2

Applets – sessão 4

Exercício de nivelamento – Applets

Você já desenvolveu aplicações para internet? Que recurso e/ou linguagem de programação você já utilizou para desenvolvimento?

▣ Applets Java

- ☐ Browser com suporte a tecnologia Java
- ☐ Podem ser baixado pela internet

- ☐ subclasse
 - ☐ java.applet.Applet ou javax.swing.JApplet
- ☐ Applet ou JApplet fornece a interface padrão
 - ☐ Entre o applet e blowser

Os Applets são aplicações java que utilizam recursos da máquina virtual java dentro de navegadores web com algumas limitações de segurança.

Um applet é um programa em Java que é executado em um browser. Um applet pode ser uma aplicação Java totalmente funcional, pois tem toda a API Java à sua disposição.

Existem algumas diferenças importantes entre um applet e uma aplicação Java standalone, incluindo o seguinte:

- ▣ Um applet é uma classe Java que estende a classe java.applet.Applet.
- ▣ Um método main () não é chamado em um applet, e uma classe do applet não vai definir main ().

- Applets são projetados para ser incorporado em uma página HTML.
- Quando um usuário visualiza uma página HTML que contém um applet, o código do applet é baixado para a máquina do usuário.
- A JVM é necessário para visualizar um applet. A JVM pode ser um plug-in do navegador da Web ou um ambiente de tempo de execução separado.
- A JVM na máquina do usuário cria uma instância da classe do applet e invoca vários métodos durante a vida do applet.
- Applets têm regras rígidas de segurança que são aplicadas pelo browser. A segurança de um applet é muitas vezes referida como área de segurança, comparando o applet para uma criança brincando em uma sandbox com várias regras que devem ser seguidas.
- Outras classes que as necessidades de applets podem ser baixados em um único arquivo Java Archive (JAR).

Ciclo de Vida de um Applet

Quatro métodos da classe Applet disponibilizam o framework necessário a partir do qual podemos construir programas java que é executado em um browser. São eles:

- **init()**: Este método destina-se a qualquer inicialização é necessário para o seu applet. Ele é chamado após as tags **PARAM** dentro da tag do **APPLET** foram processadas.
- **start()**: Este método é chamado automaticamente após o navegador chama o método init. Ele também é chamado sempre que o usuário retorna para a página que contém o applet depois de ter ido para outras páginas.
- **stop()**: Este método é chamado automaticamente quando o usuário navega para uma página em que o applet não foi referenciado.
- **destroy()**: Este método só é chamado quando o browser for fechado normalmente. Applets são feitos para serem executados em uma página HTML, sendo que os recursos alocados para a sua execução, serão removidos pelo garbage collector após a sinalização do método destroy(), chamado quando a instância do browser que estiver executando a página HTML que contém um applet for fechada.
- **paint()**: Invocado imediatamente após o método start(), e também a qualquer momento o applet precisa se redesenhar no próprio navegador. O método do paint() é herdado do java.awt.

Antes de Java, você usava HTML (a linguagem de marcação de hipertexto) para descrever o layout de uma página da Web. A HTML é simplesmente um veículo para indicar elementos de uma página de hipertexto. Por exemplo, <TITLE> indica o título da página e todo texto após essa tag se torna o título da página. Você indica o final do título com a tag </TITLE>.

Um primeiro exemplo de Applet

A seguir apresentamos um código java simples que implementa um Applet que irá escrever a mensagem “Primeiro exemplo de Applet Java” no browser quando a página html que ele estiver inserido for acessada.

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

public class HelloWorldApplet extends Applet
{
    public void paint (Graphics g)
    {
        g.drawString ("Primeiro exemplo de Applet Java ", 25, 50);
    }
}
```

Estas declarações de importação trazer as classes no âmbito da nossa classe applet:

- java.applet.Applet;
- java.awt.Graphics.

Sem essas declarações de importação, o compilador Java não reconheceria a classe Applet, bem como os objetos visuais gráficos, que a classe do applet se refere.

A classe Applet

Cada applet é uma extensão da classe java.applet.Applet ou da classe javax.swing.applet.JApplet. A classe Applet base fornece métodos que uma classe Applet derivada pode chamar para obter informações e serviços a partir do contexto browser. Estes incluem métodos que fazem o seguinte:

- Obter parâmetros de applet
- Obter a localização de rede do arquivo HTML que contém o applet
- Obter a localização de rede do diretório da classe do applet
- Imprimir uma mensagem de status no navegador
- Buscar uma imagem
- Buscar um clipe de áudio
- Reproduzir um clipe de áudio
- Redimensionar o applet

Além disso, a classe Applet fornece uma interface pela qual o usuário ou o navegador obtém informações sobre o applet e controla sua execução. O usuário poderá:

- pedido de informação sobre o autor, a versão eo copyright do applet
- solicitar uma descrição dos parâmetros do applet reconhece
- inicializar o applet
- destruir o applet
- iniciar a execução do applet
- interromper a execução do applet

A classe Applet fornece implementações padrão de cada um desses métodos. Estas implementações podem ser substituídas quando necessário.

O applet apresentado no item 2.2 está completo, tal como está. O único método substituído é o método de pintura **paint()**.

Invocando um Applet no browser

Um applet pode ser invocada por directivas de inclusão em um arquivo HTML e visualizar o arquivo através de um visualizador de applets ou navegador habilitado para Java.

A tag <applet> é a base para a incorporação de um applet em um arquivo HTML. Abaixo está um exemplo que chama do applet implementado no item 2.2 no arquivo fonte Exemplo01.java:

```
<html>
<title>The Hello, World Applet</title>
<hr>
<applet code="Exemplo01.class" width="320" height="120">
</applet>
<hr>
</html>
```

É necessário o atributo código do tag <applet>. Ele especifica a classe Applet para ser executado. Também são necessários largura e altura para especificar o tamanho inicial do painel em que um applet é executado. A directiva applet deve ser fechado com a tag </applet>.

Se um applet usa parâmetros, os valores podem ser passados para os parâmetros adicionando marcas <param> entre <applet> e </ applet>. O browser ignora texto e outras marcas entre as tags applet.

Navegadores não-habilitados para Java não processam <applet> e </applet>. Portanto, qualquer coisa que aparece entre as tags, não relacionada com o applet, é visível em navegadores não-habilitados para Java.

O usuário ou o navegador procura o código Java compilado no local do documento. Para especificar caso contrário, use o atributo codebase da tag <applet> como mostrado:

```
<applet codebase="http://amrood.com/applets"
code="HelloWorldApplet.class" width="320" height="120">
```

Se um applet for implementado em um pacote diferente do padrão, o cominho para este pacote deverá ser especificado no atributo de código usando o caractere ponto (.) para separar os componentes do pacote / classe. Por exemplo:

```
<applet code="mypackage.subpackage.TestApplet.class"
width="320" height="120">
```

A ideia básica de como usar applets em uma página Web é simples: a página HTML deve dizer ao navegador quais applets deve carregar e, então, onde colocar cada applet na página da Web. Como se poderia esperar, a tag necessária para usar um applet deve dizer ao navegador o seguinte:

- De onde obter os arquivos de classe;
- Como o applet se situa na página da Web (tamanho, localização etc.).

A exibição de um applet em uma página HTML pode ser feita com o uso da tag e “APPLET” e pela tag “OBJECT”.

```
<applet code="Exemplo1.class" width="800px" height="600px">
  <param name="message" value="Exemplo de Applet Java!!!" />
</applet>
<object codetype="application/java" classid="java: Exemplo1.class"
width="800px" height="600px">
  <param name="message" value="Exemplo de Applet Java!!!" />
</object>
```

A maneira original de incorporar um applet Java era através de uma tag APPLET com parâmetros que forneciam as informações listadas anteriormente. O W3 Consortium sugeriu trocar para a tag OBJECT, mais versátil, e a antiga tag APPLET foi desaprovada na especificação HTML 4.0. Os navegadores mais recentes reconhecerão as duas tags, mas você deve lembrar que os navegadores mais antigos não reconhecerão.

Utilizaremos estas tags sempre que quisermos que o navegador recupere os arquivos de classe da Internet (ou de um diretório na máquina do usuário) e executa o applet automaticamente, usando sua máquina virtual Java.

Além do applet em si, a página da Web pode conter todos os outros elementos HTML que você viu em uso nas páginas da Web: múltiplas fontes, listas com bullets, elementos gráficos, links etc. Os applets são apenas uma parte da página de hipertexto. É sempre interessante lembrar que a linguagem de programação Java *não* é uma ferramenta para projetar páginas

HTML; ela é uma ferramenta para *dar vida a elas*. Isso não quer dizer que os elementos de projeto de GUI em um applet Java não são importante, mas que eles devem trabalhar com (e, na verdade, são subservientes a) o projeto HTML subjacente da página da Web.

Obtendo parâmetros no Applet

O exemplo abaixo demonstra como fazer um applet responder aos parâmetros adicionais especificados no documento. Este applet exibe uma mensagem dizendo de um número recebido como parâmetro durante a carga do applet na página html e par ou impar.

VerificarParImparApplet obtém seus parâmetros no método `init()`. Ele também pode ter seus parâmetros no método `paint()`. No entanto, obter os valores e salvar as configurações de uma vez no início do applet, em vez de a cada atualização, é conveniente e eficiente.

O Applet Viewer ou o navegador chama o método `init()` de cada applet que é executado. O usuário chama `init()` uma vez ao carregar a página html.

O método `Applet.getParameter()` obtém um parâmetro dado o nome do parâmetro (o valor de um parâmetro é sempre uma string). Se o valor for numérico ou outros dados não-caracteres, a sequência deve ser analisada e convertida através de classes Wrappers, por exemplo.

```
package visao;

import java.awt.BorderLayout;
import javax.swing.JApplet;
import javax.swing.JLabel;

/**
 *
 * @author Paulo Henrique Cayres
 */
public class VerificarParImparApplet extends JApplet {

    //Executado quando o applet for carregado no browser.
    @Override
    public void init() {

        setSize(250, 100);
        JLabel lbl = new JLabel("Exemplo com passagem de
parâmetros", JLabel.CENTER);
        setLayout(new BorderLayout());
        add(lbl, BorderLayout.CENTER);

        int num = Integer.parseInt(getParameter("numero"));
```



```
        if(num%2 == 0)
            javax.swing.JOptionPane.showMessageDialog(rootPane, num+"
é um número
par!", "Mensagem", javax.swing.JOptionPane.INFORMATION_MESSAGE);
        else
            javax.swing.JOptionPane.showMessageDialog(rootPane, num+"
é um número
par!", "Mensagem", javax.swing.JOptionPane.INFORMATION_MESSAGE);
    }
}
```

Neste exemplo, o applet chama `getParameter()` informando qual o nome do parâmetro definido na tag `<PARAM>` presente na chamada `<APPLET>` na página html. O método `Integer.parseInt` (Classe Wrapper), converte uma string e retorna um inteiro.

Na sequência o applet realiza um teste de condição e chama o método `showMessageDialog()` de `javax.swing.JOptionPane` para exibir uma mensagem informando se o número é par ou ímpar. Note que estamos utilizando os mesmos componentes visuais gráficos utilizados para desenvolver aplicativos desktop em Java.

Passando parâmetros para o Applet

O exemplo abaixo apresenta um arquivo HTML com o applet `VerificarParImparApplet` incorporado. O arquivo HTML especifica o parâmetro para o applet por meio da marcação `<PARAM>`.

```
<HTML>
<HEAD>
    <TITLE>Applet HTML Page</TITLE>
</HEAD>
<BODY>

<APPLET codebase="classes" code="visao/VerificarParImparApplet.class"
width=350 height=200>
    <PARAM name="numero" value="10">
</APPLET>

</BODY>
</HTML>
```

Os parâmetros especificados numa página html serão repassados ao applet que poderão tratá-los para executar alguma funcionalidade desejada. Durante a escrita do código java no applet, o programador deverá se preocupar com a conversão dos parâmetros, que são Strings, para o tipo necessário para utilização desta informação.

Applets Java versus Aplicativos

Java possibilita a criação de dois grandes tipos de programas: um applet ou um aplicativo, sendo que uma applet java é um programa que roda a partir de um browser web e um aplicativo java, por sua vez, é um programa independente do browser que roda como um programa isolado.

Uma vez que um applet é executado a partir de um browser Web, ela tem a vantagem de já ter uma janela e a capacidade de responder aos eventos da interface com o usuário, por meio do browser. Também pelo fato de as applets serem projetadas para uso em redes, Java é muito mais restritivo nos tipos de acesso que as applets poderão ter no seu sistema de arquivos do que com os aplicativos que não estão necessariamente rodando em rede.

Conversão de aplicação em Applet:

É fácil converter um aplicativo Java gráfica (isto é, um aplicativo que usa o AWT ou SWING e que você pode começar com o lançador de programa java) em um applet que você pode inserir em uma página web.

Aqui estão os passos necessários para a conversão de um aplicativo para um applet:

- Faça uma página HTML com a tag `<APPLET></APPLET>` apropriada para carregar o código do applet.
- Fornecer uma subclasse da classe `JApplet`. Esta subclasse deve ser declarada como pública. Caso contrário, o applet não pode ser carregado.
- Eliminar o método principal na aplicação. Não construa um `Frame` para a aplicação. A sua aplicação será exibida dentro do navegador.
- Mova qualquer código de inicialização do construtor para o método `init()` do applet. Você não precisa construir explicitamente uma instância do objeto para o navegador. O applet irá instanciar-lo para você por meio de uma chamada ao método `init()`, quando você solicitar o carregamento de uma página HTML que tenha referência a tag `<APPLET>`.
- Retire a chamada para `setSize()`. O dimensionamento dos applets é feito com a especificação da largura e altura como parâmetros da tag `<APPLET>` no arquivo HTML.
- Retire a chamada para `setDefaultCloseOperation()`. Um applet não pode ser fechado. Sua execução termina quando o usuário navega para uma próxima página ou fecha o navegador.
- Eliminar a chamada para o método `setTitle()`. Applets não pode ter barras de título. Você pode, é claro, utilizar o título da própria página web, usando o título de tag HTML.
- Não chame `setVisible(true)`. O applet é exibido automaticamente durante a carregamento da página HTML.

Manipulação de eventos no Applet

Applets herdam um grupo de métodos da classe Container de manipulação de eventos. A classe Container define vários métodos, tais como `processKeyEvent` `processMouseEvent` e, para o tratamento de tipos particulares de acontecimentos, e, em seguida, um método chamado `processEvent` todos `catch`.

Para possibilitar controle de eventos, um applet deve implementar a classe de interface específica do evento apropriado e reescrevê-lo para atender a especificidade da aplicação em desenvolvimento.

O código abaixo implementa a interface `MouseListener` para controlar eventos do mouse no applet implementado:

```
package visao;

import java.awt.Graphics;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
import javax.swing.JApplet;

/**
 *
 * @author Paulo Henrique Cayres
 */

public class ExemploControleEventoMouseApplet extends JApplet
implements MouseListener {

    StringBuffer strBuffer;

    public void init() {
        addMouseListener(this);
        strBuffer = new StringBuffer();
        addItem("initializing the apple ");
    }

    public void start() {
        addItem("starting the applet ");
    }

    public void stop() {
        addItem("stopping the applet ");
    }

    public void destroy() {
```

```
        addItem("unloading the applet");
    }

    void addItem(String word) {
        System.out.println(word);
        strBuffer.append(word);
        repaint();
    }

    public void paint(Graphics g) {
        //Draw a Rectangle around the applet's display area.
        g.drawRect(0, 0,
            getWidth() - 1,
            getHeight() - 1);
        //display the string inside the rectangle.
        g.drawString(strBuffer.toString(), 10, 20);
    }

    @Override
    public void mouseClicked(MouseEvent e) {
        addItem("mouse clicked! ");
    }

    @Override
    public void mousePressed(MouseEvent e) {

    }

    @Override
    public void mouseReleased(MouseEvent e) {

    }

    @Override
    public void mouseEntered(MouseEvent e) {
        addItem("mouse entered! ");
    }

    @Override
    public void mouseExited(MouseEvent e) {
        addItem("mouse exited! ");
    }
}
```

Exibindo imagens no Applet

Um applet pode exibir imagens no formato GIF, JPEG, BMP, e outros. Para exibir uma imagem dentro do applet, você usa o método `drawImage ()` encontrado na classe `java.awt.Graphics`. O exemplo abaixo implementa um applet com todos os passos necessários para visualizar imagens:

```
package visao;

import java.applet.AppletContext;
import java.awt.Graphics;
import java.awt.Image;
import java.net.MalformedURLException;
import java.net.URL;
import javax.swing.JApplet;

/**
 *
 * @author Paulo Henrique Cayres
 */
public class ExemploImgApplet extends JApplet {

    private Image image;
    private AppletContext context;

    public void init()
    {

        context = this.getAppletContext();
        String imageURL = this.getParameter("image");
        if(imageURL == null)
        {
            imageURL = "alert.gif";
        }
        try
        {
            URL url = new URL(this.getDocumentBase(), imageURL);
            System.out.println(url);
            image = context.getImage(url);
        } catch (MalformedURLException e)
        {
            e.printStackTrace();
            // Display in browser status bar
            context.showStatus("Could not load image!");
        }
    }
}
```

```
public void paint(Graphics g)
{
    context.showStatus("Displaying image");
    g.drawImage(image, this.getWidth()/2, this.getHeight()/2, 33,
34, null);
    repaint();
}
}
```

Reprodução de áudio no Applet

Um applet pode reproduzir um arquivo de áudio representado pela interface AudioClip no pacote java.applet. A interface AudioClip tem três métodos, incluindo:

- **public void play():** Reproduz o clipe de áudio uma vez, desde o início.
- **public void loop():** Faz com que o clipe de áudio seja reproduzido continuamente.
- **public void stop():** Para a reprodução do clipe de áudio.

Para obter um objeto AudioClip, você deve chamar o método da classe Applet getAudioClip (). O método getAudioClip () retorna imediatamente, ou não resolve o URL para um arquivo de áudio real. O arquivo de áudio não é baixado até que seja feita uma tentativa de reproduzir o clipe de áudio.

O mecanismo de som que reproduz os clipes de áudio suporta vários formatos de arquivo de áudio, incluindo:

- formato de arquivo Sun Audio (.au)
- Windows wave (.wav)
- Macintosh AIFF (.aif ou .AIFF)
- Musical Instrument Digital Interface – MIDI (.mid ou .rmi)

Segue-se o exemplo mostrando todos os passos para ter um áudio:

```
package visao;

import java.applet.AppletContext;
import java.applet.AudioClip;
import java.net.MalformedURLException;
import java.net.URL;
import javax.swing.JApplet;

/**
 *
```

```
* @author Paulo Henrique Cayres
*/
public class ExemploAudioApplet extends JApplet {

    private AudioClip clip;
    private AppletContext context;
    public void init()
    {
        context = this.getAppletContext();
        String audioURL = this.getParameter("audio");
        if(audioURL == null)
        {
            audioURL = "teste.wav";
        }
        try
        {
            URL url = new URL(this.getDocumentBase(), audioURL);
            clip = context.getAudioClip(url);
        } catch (MalformedURLException e)
        {
            e.printStackTrace();
            context.showStatus("Could not load audio file!");
        }
    }

    public void start()
    {
        if(clip != null)
        {
            clip.loop();
        }
    }

    public void stop()
    {
        if(clip != null)
        {
            clip.stop();
        }
    }
}
```

Exercícios de Fixação 4 – Applets

Qual a diferença entre lançar uma exceção e capturar uma exceção?

Atividade 1 – Applets

1. Faça uma aplicação Java Applet que recebe como parâmetro de inicialização da página HTML um número inteiro, calcule e escreva num JTextArea o resulta da tabuada do número informado.
2. Faça uma aplicação Java Applet que possibilite a digitação de 3 números (JTextField) e que mostre em uma mensagem (JOptionPane) com o maior números. A mensagem será exibida quando o usuário clicar no botão (JButton) Mensagem. Após a visualização da mensagem os campos de entrada de dados deverão ser limpos e o primeiro campo de entrada de dados deverá receber o foco.
3. Utilizando o exemplo de definição de classes em Java, crie a classe de aplicação Pessoa descrita abaixo. Os métodos seletores e modificadores deverão ser criados para cada um de seus atributos.

Pessoa
String nome int idade
Pessoa() Pessoa(String, idade)

4. Construa uma Applet que apresente uma tela com campos de entrada de dados (JTextField) para cada um dos atributos da classe Pessoa.
5. Implemente na interface do exercício 3 botões de ação (JButton) para possibilitar inclusão, consulta e exclusão de objetos instanciados da classe Pessoa em um ArrayList. A exclusão será realizada por meio do nome da Pessoa.
6. Implemente na interface do exercício 3 um JTextArea onde serão listados os objetos em ordem alfabética. A ordenação será executada após clique num botão.

Atividade 2 – GUI em Java (complementar)

1. Construa um Applet Java que apresente uma tela com uma campo de entrada de dados (JTextField) para que o usuário possa informar um número inteiro. O Applet deverá implementar um botão (JButton) que, quando clicado, informe se o número digitado é par ou ímpar.
2. Construa um Applet Java que recebe um número inteiro positivo como parâmetro da página html e calcule e escreva a tabuada deste número dentro de um JTextArea.

3

Tratamento de Erros – sessão 5

Exercício de nivelamento – Tratamento de erros

Você já desenvolveu alguma aplicação que utilizou tratamento de erros? Em que linguagem?

■ Tratamento de Erros

- ☐ Durante a execução de métodos
 - ☐ Erro de lógica
 - ☐ Uso inadequado da API
- ☐ Uso de classes especializadas
 - ☐ Não compromete a inteligibilidade do código
 - ☐ Implementado por meio de interfaces

Os erros em Java são tratados como em C++, através de **exceções**. Em Java, todas as exceções descendem de `Throwable`, mas a hierarquia se divide em `Error` e `Exception`.

Os erros da classe `Error` descrevem erros internos e de exaustão de recursos. Pouco se pode fazer numa situação dessas, além de informar ao usuário e deixar o programa de forma segura. Essas situações são raras.

Ao programar em Java, deve-se focalizar a hierarquia `Exception`. Esta também se divide em exceções que derivam de `RuntimeException` e as que não. A regra geral é a seguinte:

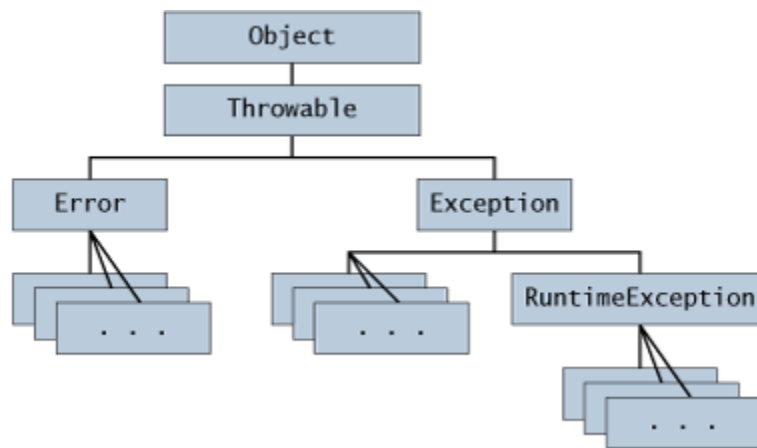
- Uma `RuntimeError` ocorre porque houve um erro de programação. Qualquer outra exceção ocorre devido a algo errado, como erro de I/O que tenha ocorrido no programa.

As exceções que derivam da `RuntimeError` incluem problemas como:

- ❑ Conversão explícita de tipo (*cast*);
- ❑ Acesso a elemento de array além dos limites;
- ❑ Acesso de ponteiro nulo.

As exceções que não derivam do `RuntimeError` incluem:

- ❑ Tentar ler além do final de um arquivo;
- ❑ Tentar abrir uma URL incorreta;
- ❑ Tentar encontrar um objeto `Class` através de uma string que não denota uma classe existente.



Descrever Exceções em Métodos

Para que um método possa gerar uma exceção tratável pelo usuário, deve-se descrever quais ele pode **levantar**. Isto é, além das informações que um método pode retornar, também se informa o que pode dar errado.

Essa informação das exceções é feita no cabeçalho dos métodos, por exemplo:

```
public String readLine() throws IOException
```

Se dentro do método `readLine` ocorrer uma exceção `IOException`, essa poderá facilmente ser tratada no método que chamou essa função. Mas não há necessidade de se tratar todo e qualquer tipo de exceção. Basta ter em mente as ocasiões em que uma exceção é levantada:

- ❑ Na chamada de um método que lança (*throws*) uma exceção;
- ❑ Se um erro for detectado e for lançada uma exceção (*throw*);

- Num erro de programação;
- Num erro interno do Java.

Se ocorrer um dos dois primeiros casos, então o programador deverá informar a quem usar o método que ele levanta uma exceção. Caso contrário, se não tiver nenhum mecanismo para tratar essas exceções, o programa termina a execução.

```
class Animation {  
    ...  
    public Image loadImage(String s) throws IOException {  
        ...  
    }  
}
```

Se o método pode lançar mais de uma exceção, separa-se por vírgula:

```
class Animation {  
    ...  
    public Image loadImage(String s) throws EOFException,  
        MalformedURLException {  
        ...  
    }  
}
```

Não se deve anunciar (`throws`) erros internos do Java (derivadas de `Error`) nem erros em tempo de execução (`RuntimeException`), porque estão fora do alcance do programador. Essas exceções são chamadas **não verificadas** (*unchecked*).

As exceções não verificadas não podem ser tratadas, e devem ser evitadas no programa, através de verificações. Um método precisa declarar todas as exceções verificadas que ele pode levantar.

Levantar uma Exceção

Para um método levantar uma exceção, é usado o comando `throw`. Só se pode levantar uma exceção se ela foi declarada no cabeçalho do método (ou uma superclasse). Exemplo:

```
throw new EOFException();
```

Ou, se preferir:

```
EOFException e = new EOFException();  
throw e;
```

E para juntar tudo que foi visto nesta sessão:

```
String lerDados(BufferedReader ent) throws EOFException {  
    ...  
    while (...) {
```

```
        if (ch == -1)    // encontrou EOF
            throw new EOFException();
    }
}
```

A `EOFException` também tem um outro construtor, que recebe uma string, que serve para descrever a condição do erro mais detalhadamente. Exemplo:

```
throw new EOFException("Conteúdo completo recebido.");
```

Criar Novas Exceções

Para criar uma nova classe de exceção basta criar a classe e derivá-la de `Exception` ou de qualquer uma de suas descendentes.

```
class ExecucaoTamanhoArquivo extends IOException {
    public ExecucaoTamanhoArquivo() { }
    public ExecucaoTamanhoArquivo(String str) {
        super(str);
    }
}
```

Agora pode-se usá-la normalmente nos programas:

```
String lerDados(BufferedReader ent) throws ExecucaoTamanhoArquivo {
    ...
    while (...) {
        if (ch == -1)    // encontrou EOF
            throw new ExecucaoTamanhoArquivo();
    }
}
```

Capturando Exceções

Para capturar exceções usa-se um bloco `try...catch`, com a seguinte construção:

```
try {
    <Código que pode gerar erro>
}
catch(TipoExceção_1 e) {
    <Manipulador da exceção tipo 1>
}
...
catch(TipoExceção_n e) {
    <Manipulador da exceção tipo n>
}
finally {
    <Código finalizador>
}
```

```
}
```

Dentro do trecho `try` coloca-se o código que será executado e verificado por um determinado erro. Se uma exceção ocorrer, ela será de um determinado tipo, e conforme o tipo, a execução será desviada para um trecho `catch` que trata esse tipo de exceção. Pode-se ter um ou mais trechos `catch`, conforme a necessidade.

O trecho `finally` é usado quando se deseja, por exemplo, liberar um recurso reservado por um código que gerou exceção. O trecho de código que estiver aí dentro sempre será executado, seja uma exceção tratada (`catch`) ou não. A cláusula `finally` pode ser usada mesmo sem o `catch`, assim, se uma exceção foi levantada, o trecho `finally` é executado e a exceção é rejeitada.

```
Graphics g = image.getGraphics();
try {
    // código que pode gerar erros
}
catch(IOException e) {
    // tratamento de erros
}
finally {
    // libera recursos, mesmo se nenhuma exceção for levantada
    g.dispose();
}
```

Exercícios de Fixação 5 – Tratamento de Erros

Qual a diferença entre lançar uma exceção e capturar uma exceção?

Atividade 1 – GUI em Java

1. Faça uma classe executável que receba por meio da linha de comando do programa (prompt) dois números e um operador aritmético (+ , - , x , /), informe o resultado da operação. Atenção: não é possível divisão por zero. Então, lance uma exceção caso isso ocorra.
2. Faça uma classe executável que receba por meio da linha de comando do programa (prompt) um número natural (de 1 a 10) e informe a sua tabuada. Atenção: Se não informado, assumo o valor 7. Então, utilize a cláusula `finally` em sua solução.

3. Crie uma classe de exceção que implemente o método `toString()` e retorne a mensagem "Impossível converter. Favor informar apenas dígitos na entrada de dados"
4. Crie uma classe que implemente o método `retornaDouble(String)` que lança a exceção implementada no enunciado 3 caso o String de entrada seja vazia.
5. Faça uma aplicação Java com uma interface que possibilite a digitação de um número e que mostre em uma mensagem (`JOptionPane`) com resultado da multiplicação deste número por 5. Utilize try-catch para tratar a exceção por meio do método de conversão de String para double implementado no enunciado 4 para tratar a entrada de dados.

Atividade 2 – GUI em Java (complementar)

1. Defina a classe `ContaCorrente` contendo os atributos número da conta, nome do cliente e saldo para lançar exceções (definidas por você) quando a conta é construída com saldo negativo e quando se saca uma quantidade maior que o saldo atual.
2. Faça uma aplicação Java com uma interface que possibilite a digitação de informações para instanciar objetos da classe `ContaCorrente` (criada no enunciado anterior) e armazenar num `ArrayList`.
3. Faça uma aplicação Java com uma interface que possibilite informar o número de uma conta corrente e o valor com dois botões (`JButton`), sendo um para depósito e outro para saque. O botão de saque deverá lançar uma exceção para tratar possível saldo negativo em caso de saque de valores superiores ao saldo atual da conta.
4. Defina uma classe `MyInput`, contendo métodos estáticos abaixo (todos os métodos devem retornar o tipo especificado), de forma que exceções sejam lançadas caso um valor incorreto tenha sido fornecido pelo usuário.
 - a. **`getInt`** para leitura de um inteiro, com parâmetros que representem os valores mínimo e máximo aceitos.
 - b. **`getFloat`** para leitura de um flutuante, com parâmetros que representem os valores mínimo e máximo aceitos.
 - c. **`getChar`** para leitura de um caractere, com um parâmetro do tipo `String` contendo um conjunto de caracteres aceitos.
5. Faça uma classe executável Java que teste entradas de dados para cada um dos métodos implementados no enunciado 4 e trate possíveis exceções (problemas de conversão com a estrutura try-catch).

4

Controle de entrada e saída, streams e tratamento de arquivos – sessão 6

Exercício de nivelamento – I/O

Você já desenvolveu alguma aplicação que utilizou manipulou arquivos em disco? Em que linguagem? Trabalhou com gravação de objetos? Que recurso utilizou?

▣ Controle de entrada e saída

- ☐ Leitura/Escrita de Dados
 - ☐ Orientada a bytes (modo 'binário')
 - ☐ Orientada a caracteres (modo 'texto')
- ☐ Serialização de Objetos
- ☐ Outros
 - ☐ Acesso Aleatório: classe `RandomAccessFile`
 - ☐ Informações sobre arquivos e diretórios: classe `File`

Uma stream de I/O gera o caminho por meio do qual seus programas podem enviar uma sequência de bytes de uma fonte até um destino. Uma stream de entrada é uma fonte (ou produtor) de bytes e uma stream de saída é o destino (ou consumidor).

Todas as funções de I/O stream tratam os arquivos de dados ou os itens de dados como uma corrente ou um fluxo (stream) de caracteres individuais. Se for escolhida uma função de stream apropriada, a aplicação poderá processar dados em qualquer tamanho ou formato, desde simples caracteres até estruturas de dados grandes e complicadas. Tecnicamente

falando, quando um programa usa uma função stream para abrir um arquivo para I/O, o arquivo que foi aberto é associado com uma estrutura do tipo FILE que contém informações básicas sobre o arquivo.

Uma vez aberta a stream, é retornado um ponteiro para a estrutura FILE. Este ponteiro FILE – algumas vezes chamado de ponteiro stream ou stream – é usado para fazer referência ao arquivo para todas as entradas/saídas subseqüentes.

Todas as funções de I/O stream fornecem entrada e saída bufferizada, formatada ou não formatada. Uma stream bufferizada proporciona uma localização de armazenamento intermediária para todas as informações que são provenientes de stream e toda saída que está sendo enviada a stream. I/O em disco é uma operação que toma tempo, mas a bufferização da stream agilizará sua aplicação. Ao invés de introduzir os dados stream, um caractere ou uma estrutura a cada vez, as funções I/O stream acessam os dados em um bloco de cada vez.

À medida que a aplicação necessita processar a entrada, ela simplesmente acessa o buffer, o que é um processo muito mais rápido. Quando o buffer estiver vazio, será acessado um novo bloco de disco. O inverso também é verdadeiro para saída stream. Ao invés de colocar fisicamente na saída todos os dados, à medida que é executada a instrução de saída, as funções de I/O stream colocam todos os dados de saída no buffer. Quando o buffer estiver cheio, os dados serão escritos no disco.

Dependendo da linguagem de alto nível que estiver sendo utilizada, pode ocorrer um problema de I/O bufferizado. Por exemplo, se o programa executar várias instruções de saída que não preencham o buffer de saída, condição necessária para que ele fosse descarregado no disco, aquelas informações serão perdidas quando terminar o processamento de programa.

A solução geralmente envolve a chamada de uma função apropriada para “limpar” o buffer. É claro que uma aplicação bem escrita não deverá ficar dependendo destes recursos automáticos, mas deverá sempre detalhar explicitamente cada ação que o programa deve tomar. Mais uma observação: se a aplicação terminar de maneira anormal quando se estiver usando stream I/O, os buffers de saída podem não ser esgotados (limpos), resultando em perda de dados.

O último tipo de entrada e saída é chamado “low-level I/O” – I/O de baixo nível. Nenhuma das funções de I/O de baixo nível executa bufferização e formatação. Ao invés disso, elas chamam diretamente os recursos de I/O do sistema operacional. Estas rotinas permitem acessar arquivos e dispositivos periféricos em um nível mais básico do que as funções stream.

Os arquivos abertos desta maneira retornam um manipulador de arquivo, um valor inteiro que é utilizado para fazer referência ao arquivo em operações subseqüentes. Em geral, é uma prática ruim de programação misturar funções de I/O stream com rotinas de baixo nível.

Como as funções stream são bufferizadas e as funções de baixo nível não, a tentativa de acessar o mesmo arquivo ou dispositivo por dois métodos diferentes leva à confusão e à eventual perda de dados nos buffers.

Stream em Java

As bibliotecas Java para I/O (`java.io` e `java.nio`) oferecem numerosas classes de stream. Mas todas as classes de stream de entrada são subclasses da classe abstrata `InputStream`, e todas as classes stream de saída são subclasses da classe abstrata `OutputStream`. A figura abaixo ilustra a hierarquia de classes do pacote `java.io`.

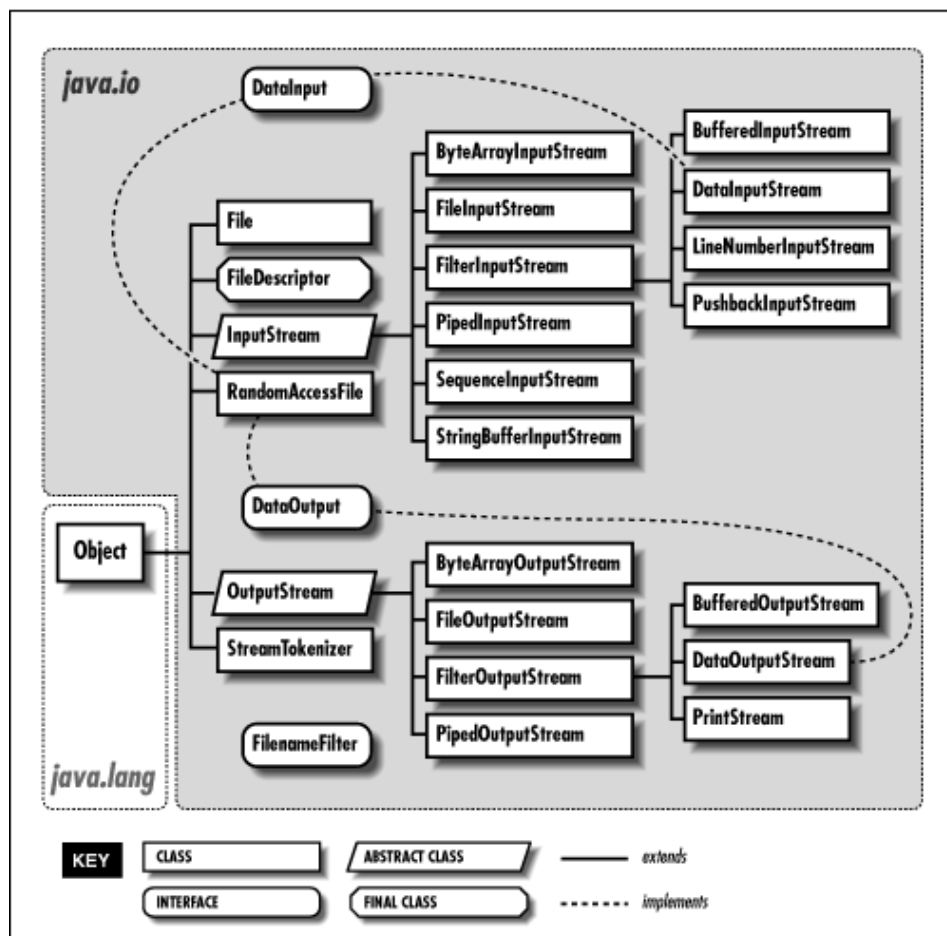


Figura 20 – Biblioteca Java IO.

O recurso de serialização permite que um objeto seja transformado em um stream de bytes, que podem ser armazenados em um arquivo e enviados via rede. Através desses bytes torna-se possível recriar a classe serializada, desde que esta implemente a interface `Serializable` ou a `Externalizable`.

Somente objetos que implementem essas interface podem ser “desintegrados” em bytes, ou “integrados” a partir dos bytes. A interface Externalizable, por exemplo, deve implementar os

métodos `writeExternal` e `readExternal`. As variáveis serão salvas ou lidas por esses métodos. Essas duas chamadas recebem uma classe do tipo `ObjectOutput` e `ObjectInput`, respectivamente. O principal método da classe `ObjectOutput` é `writeObject` e o da classe `ObjectInput` é o `readObject`.

Esses métodos serão os responsáveis pela transformação da classe em bytes e vice-versa. Existem classes concretas para essas interfaces, que são `ObjectOutputStream` e `ObjectInputStream`. Este recurso, serialização, pode ser utilizado em aplicações visuais, por exemplo, para permitir que os usuários salvem seus trabalhos em um arquivo. Neste caso, antes de escrever um objeto em uma stream, os objetos deverão ser serializados.

Deve-se considerar a serialização de objetos semelhante à divisão de um objeto em pequenos pedaços para poder escrevê-los em uma stream de bytes. O termo para este processo é serialização de objetos.

Entrada e Saída baseada em Bytes

A classe abstrata `OutputStream` contém definições de métodos `write` para gravação de bytes de um arquivo. O método de gravação de byte é abstrato, devendo ser redefinido em subclasses, como `FileOutputStream` e `DataOutputStream`.

O código abaixo ilustra a utilização dos métodos `writeInt()` e `writeDouble()` utilizados para gravação de bytes de um arquivo informado pelo usuário.

```
import java.io.*;

public class EscritaBytes {
    public static void main(String a[]) {
        // verifica presença do argumento obrigatório com nome do
arquivo
        if (a.length==0) {
            System.out.println("uso:\njava EscritaBytes <nomeArquivo>
[quant]");
            System.exit(-1);
        }
        // efetua abertura do arquivo
        DataOutputStream dos = null;
        try {
            dos = new DataOutputStream(new FileOutputStream(a[0]));
            System.out.println("Arquivo '" + a[0] + "' aberto.");
        } catch (FileNotFoundException e) {
            System.out.println(e);
            System.exit(-1);
        }
        // converte quantidade dada pelo usuário ou usa default
        int quant = 10;
        try {
```

```
        quant = Integer.parseInt(a[1]);
    } catch (Exception e) {
        System.out.println("Usando quantidade default 10");
    }
    // gera quantidade indicada de valores aleatórios
    try {
        System.out.println("Gerando conteudo...");
        for(int i=0; i<quant; i++) {
            dos.writeInt(i);
            double aux = Math.random();
            dos.writeDouble(aux);
            System.out.println("#" + i + " : " + aux);
        }
        dos.close();
        System.out.println("Arquivo '" + a[0] + "' fechado.");
    } catch (Exception e) {
        System.out.println(e);
    }
}
}
```

Da mesma forma a classe abstrata `InputStream` contém definições de métodos `read` para leitura de bytes de um arquivo. O método de leitura de byte é abstrato, devendo ser redefinido em subclasses, como `FileInputStream`.

O código abaixo ilustra a utilização dos métodos `readInt()` e `readDouble()` utilizados para leitura de bytes de um arquivo informado pelo usuário.

```
import java.io.*;

public class LeituraBytes {
    public static void main(String a[]) {
        // verifica presença de argumento com nome do arquivo
        if (a.length==0) {
            System.out.println("uso:\njava LeituraBytes <nomeArquivo>");
            System.exit(-1);
        }
        // efetua a abertura do arquivo, para leitura de bytes
        DataInputStream dis = null;
        try {
            dis = new DataInputStream(new FileInputStream(a[0]));
            System.out.println("Arquivo '" + a[0] + "' aberto.");
        } catch (FileNotFoundException e) {
            System.out.println(e);
            System.exit(-1);
        }
        // efetua a leitura do arquivo enquanto existirem dados
```

```
try {
    System.out.println("--- Conteudo ---");
    while(dis.available()>0) {
        int i = dis.readInt();
        double aux = dis.readDouble();
        System.out.println("#" + i + " : " + aux);
    }
    System.out.println("--- Conteudo ---");
    dis.close();
    System.out.println("Arquivo '" + a[0] + "' fechado.");
} catch(IOException e) {
    System.out.println(e);
}
}
```

Serialização de Objetos

A serialização é um mecanismo que permite transformar um objeto em uma sequência de bytes, a qual pode ser restaurada, em um momento posterior, em um objeto idêntico ao original.

As classes `ObjectOutputStream` e `ObjectInputStream` implementam o protocolo de serialização do Java. Estes streams são capazes de serializar e desserializar objetos instanciados em aplicações Java. Por exemplo, podemos ter uma classe da qual podemos instanciar objetos que representam produtos.

Note que, para um objeto ser serializado, sua classe deve implementar a interface `Serializable`, que não exige a implementação de qualquer método, servindo apenas como uma tag interface (marcação) que autoriza a serialização.

```
import java.io.*;

public class Produto implements Serializable {
    String nome;
    int codigo;

    public Produto(String n, int c) {
        nome = n;
        codigo = c;
    }

    public String toString() {
        return "Produto[nome=" + nome + ", codigo=" + codigo + "];"
    }
}
```

```
}
```

Depois de instanciar objetos da classe produto nós podemos ter uma aplicação Java que possibilite a serialização destes objetos e gravá-los em um arquivo em disco.

```
import java.io.*;
import java.util.*;

public class SerializaProduto {
    public static void main(String a[]) {
        if (a.length==0) {
            System.out.println("uso:\njava SerializaProduto
<nomeArquivo>");
            System.exit(-1);
        }
        ObjectOutputStream oos = null;
        try {
            oos = new ObjectOutputStream(new FileOutputStream(a[0]));
            System.out.println("Arquivo '" + a[0] + "' aberto.");
        } catch(IOException e) {
            System.out.println(e);
            System.exit(-1);
        }
        try {
            System.out.println("Gerando conteudo...");
            Produto p = new Produto("Nome do Produto", 123456);
            oos.writeObject(p);
            System.out.println(p);
            Date d = new Date();
            oos.writeObject(d);
            System.out.println(d);
            oos.close();
            System.out.println("Arquivo '" + a[0] + "' fechado.");
        } catch(Exception e) {
            System.out.println(e);
        }
    }
}
```

Por fim, podemos realizar o processo inverso através da leitura de uma sequência de bytes de um arquivo em disco, que representa um objeto, e desserializá-la para que possamos novamente ter um objeto instanciado da classe produto em uma aplicação Java.

Exercícios de Fixação 6 – Java I/O

Qual a diferença entre lançar uma exceção e capturar uma exceção?

Atividade 1 – Java I/O

1. Desenvolva os problemas (aplicação em três camadas) descritos abaixo:
 - 1.1. Crie uma classe com o nome de produto com os atributos código, nome, preço de compra e preço de venda que possibilite a serialização de objetos.
 - 1.2. Crie uma classe de controle que irá possibilitar a gravação de objetos em disco e demais consultas que são solicitadas;
 - 1.3. Utilizar Coleções para busca e ordenação;

- 1.4. Desenvolva uma aplicação GUI em Java que disponibilize as seguintes funcionalidades:
 - 1.4.1. Menu principal que possibilite acesso as seguintes funcionalidades:
 - 1.4.1.1. Tela que possibilite a digitação dos dados de produtos e sua inclusão num arquivo em disco;
 - 1.4.1.2. Tela que possibilite a digitação de um intervalo de preços de compra necessários para a geração de uma listagem (jTextArea) com produtos que tenham esta característica informada pelo usuário do sistema;
 - 1.4.1.3. Tela que possibilite a listagem de todos os produtos cadastrados;
 - 1.4.1.4. Tela que possibilite a visualização de dados do produto com o maior preço de compra e também o que tem o menor preço de venda;
 - 1.4.1.5. Tela que possibilite a consulta de um produto pelo seu nome. A tela deverá mostrar todos os dados cadastrados do produto em jTextFields desabilitados ou uma mensagem (JOptionPane) com a informação de que não existe produto com a descrição informada;

2. Atividade 2 – Java I/O (complementar)

1. Uma indústria deseja manter um banco de dados sobre a folha de pagamento mensal de seus funcionários. Para isso, é lido de cada funcionário: o nome, o salário-base e uma categoria funcional (numérica: 1-operário, 2-administrativo ou 3-gerência). Os dados devem ficar armazenados em um **arquivo** (folha.dat) e a cada vez que for solicitado, o contra-cheque de um funcionário deve ser exibido, com as seguintes informações: **nome do funcionário**, **salário bruto** (se da categoria 1, o próprio salário-base; se da categoria 2, o salário-base + 5%; se da categoria 3, o salário-base + 15%), **descontos** (se salário bruto maior ou igual a 1000, 3% sobre o salário bruto; caso contrário, 1% sobre o salário bruto) e **salário líquido** (salário bruto – descontos). Desenvolva um aplicação GUI em Java que apresente as seguintes opções:
 - a) Incluir (solicitação dos dados de um funcionário e gravação de um registro com os dados da folha de pagamento no arquivo)
 - b) Listar (leitura dos dados da folha de pagamento de um funcionário do arquivo, mostrando as informações do contra-cheque)
6. A comissão organizadora de um campeonato de atletismo decidiu apurar os resultados da competição através de um processamento eletrônico. De cada atleta são lidos: código de identificação, nome e o tempo gasto em cada uma das 5 provas que formam a competição. Esses dados são armazenados em um **arquivo**. Ao final de cada prova, a pontuação é atribuída de acordo com o tempo realizado em cada prova (quem gastou menos tempo está em 1º lugar) e é feita da seguinte maneira: 1º colocado: 5 pontos, 2º colocado: 3 pontos e 3º colocado: 1 ponto. Essa pontuação é atualizada na pontuação geral do campeonato. Desenvolva uma aplicação GUI em Java que apresente as seguintes opções:
 - a) Inclusão dos dados (solicitação dos dados de um atleta e gravação de um registro no arquivo)
 - b) Classificação (leitura dos dados dos atletas e processamento dos tempos e pontos)

5

JDBC – sessão 7, 8 e 9

Exercício de nivelamento – JDBC

Você já desenvolveu alguma aplicação que utilizou acesso a banco de dados? Em que linguagem? Que banco de dados utilizou?

■ JDBC

- Drivers de acesso a banco de dados

- Drivers

- Classes especializadas

JDBC (*Java Database Connectivity*) é a API padrão para conectividade de bancos de dados, tais como banco de dados SQL e outras fontes de dados tabulares, como planilhas e arquivos simples, com a linguagem de programação Java.

JDBC fornece em sua API um conjunto de classes genéricas para acesso a qualquer tipo de banco de dados baseado em SQL.

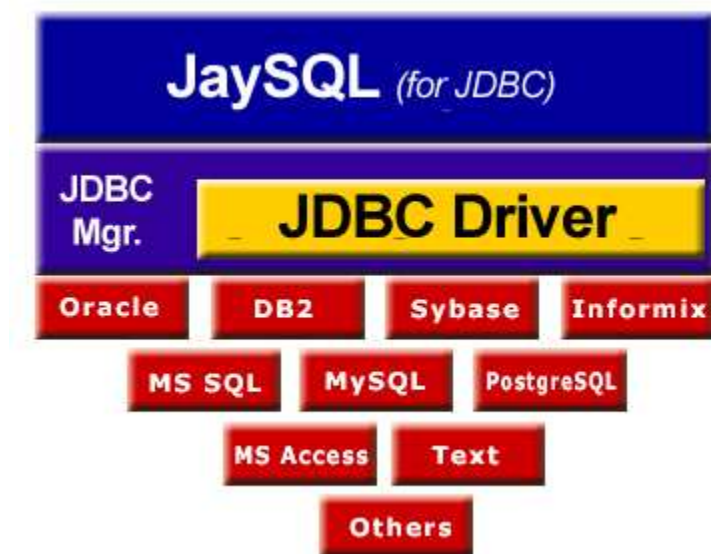


Figura 21 – JDBC Driver.

O JDBC está no pacote `java.sql.*`. Pode ser encontrado no site da Oracle. A instalação do J2SE já trás consigo o JDBC.

Além disso, precisa-se de um driver para acesso ao banco de dados que se quer. Atualmente os drivers de acesso a banco de dados para aplicações desenvolvidas em java estão disponíveis no site do fornecedor do SGBD que você necessita acessar. Por exemplo, para acesso ao PostgreSQL, usa-se o driver **JDBC para PostgreSQL**. O driver deve ser colocado no CLASSPATH do projeto para que possa ser referenciado.

Os passos para o acesso são:

1. Instalação do driver e inserção no CLASSPATH;
2. Registro do driver no código
3. Criação de uma conexão
4. Executar consultas SQL.

Além de acessos usando-se drivers, pode-se usar uma ponte que vai desde o JDBC ao ODBC, para acesso de bases Win32.

A arquitetura JDBC pode ser melhor visualizada na seguinte figura:

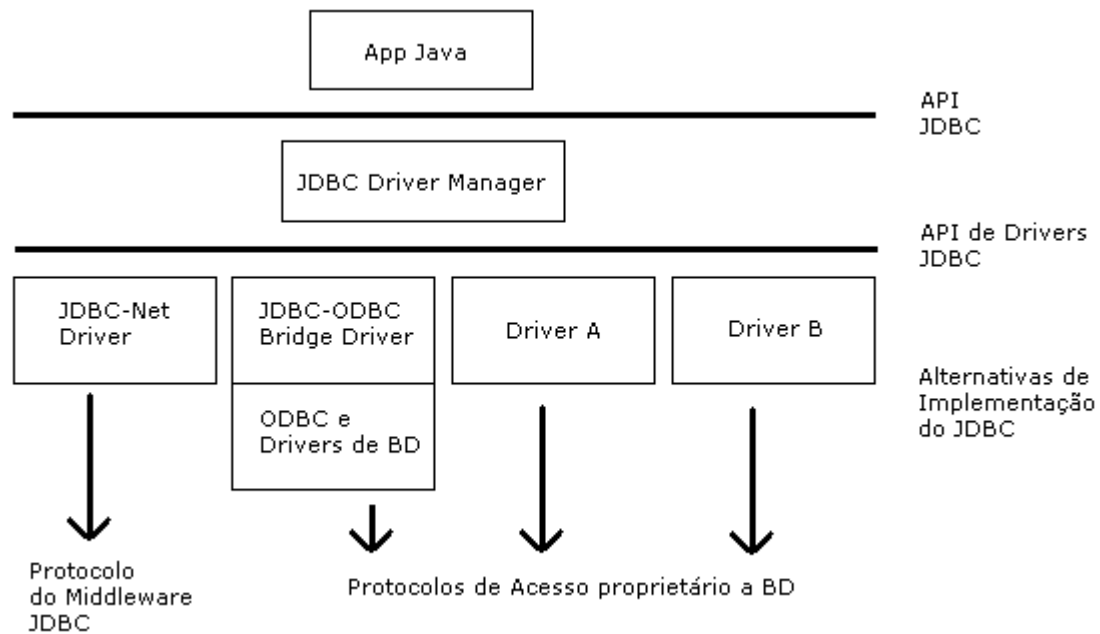


Figura 22 – Arquitetura JDBC.

Exemplo de Acesso

Segue um exemplo de acesso a um banco de dados PostgreSQL. Perceba que a variável `strDriver` deve conter o nome da classe do driver, para que possa ser registrada, e `url` deve conter a localização do banco de dados. Essa separação é feita de maneira com a que foi implementada, permitindo que se possa colocá-las num arquivo de configuração, tornando assim fácil a alteração, se a plataforma mudar.

```

public void pesquisar() {
    Connection con;
    Statement st;
    ResultSet rs;
    String url, user, password, strDriver;
    user = "usr";
    password = "pwd";
    url = "jdbc:postgresql://localhost:5432/postgres";
    strDriver = "org.postgresql.Driver";
    try {
        // registra o driver
        Class.forName( strDriver );
        // acesso
        con = DriverManager.getConnection(url, user, password);
        st = con.createStatement();
    }
}

```

```
rs = st.executeQuery("select * from tb_cliente");
while (rs.next()) {
    System.out.println(rs.getString("nome_cliente"));
}
st.close();
con.close();
}
catch(Exception e) {
    System.out.println(e.getMessage());
}
}
```

Para acessar a mesma base de dados usando uma ponte ODBC pode-se alterar somente o seguinte na função acima:

```
url = "jdbc:odbc:banco_cliente";
strDriver = "sun.jdbc.odbc.JdbcOdbcDriver";
```

Onde `banco_cliente` é exatamente o DSN criado no ODBC do Windows.

Obtendo Conexões

Para obter uma conexão como banco de dados usa-se o método:

```
Connection con = DriverManager.getConnection(url, user, pwd);
```

Onde **url** é o endereço do servidor de banco de dados, **user** é o usuário de conexão e **pwd** sua senha. Esse método retorna um objeto da classe **Connection**. Caso algum erro ocorra, uma exceção será levantada.

O objeto `Connection` também é responsável pela dinâmica de atualização das alterações no banco: **commit** e **rollback**.

Executando Consultas

Para executar uma consulta simples (select) deve-se ter um objeto da classe `Statement`, e deste objeto chamar o método `executeQuery`. Para se conseguir um `Statement`, usa-se `createStatement` da conexão:

```
st = con.createStatement();
rs = st.executeQuery("select * from tb_cliente");
```

Percebe-se que o método `executeQuery` retorna um objeto **ResultSet** que contém o resultado da seleção.

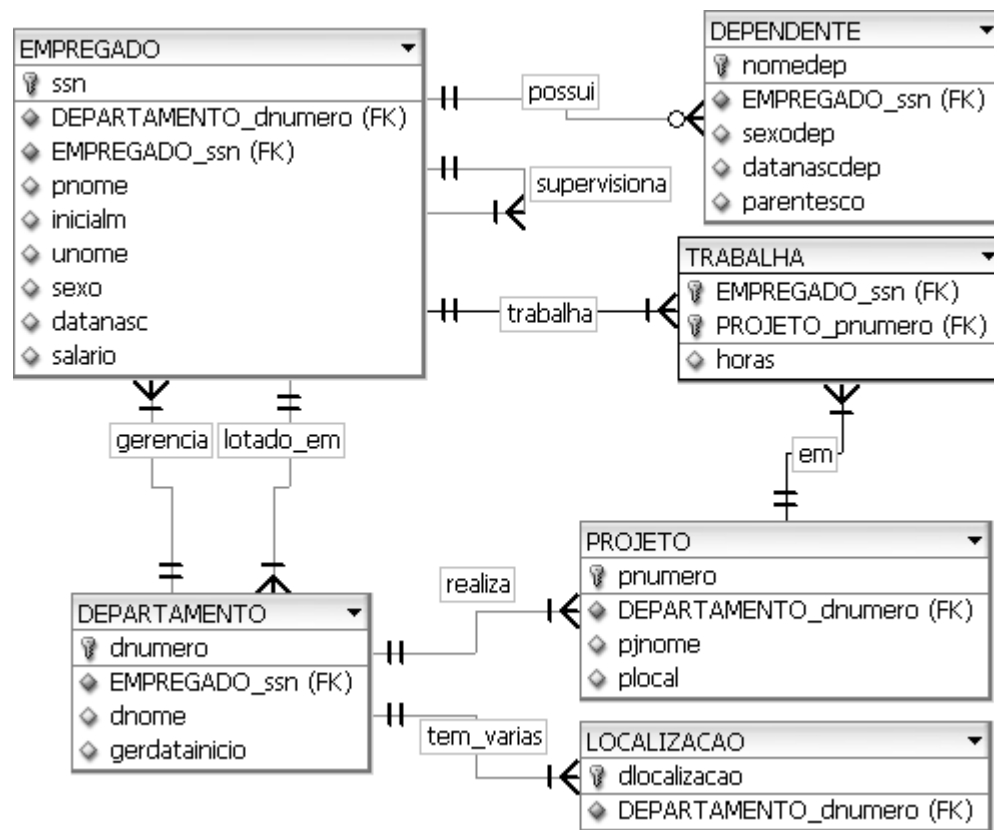
Sessão 07 – JDBC

Exercícios de Fixação 7 – Java JDBC

Quais as principais diferenças dos objetos Statement e PreparedStatement?

Atividade 1 – Java JDBC

2. Todas as consultas a seguir se referem ao banco de dados “empresa” **empresa.sql**. Escreva o comando DML para cada uma delas e mostre o resultado obtido utilizando componentes visuais gráficos do pacote Swing do Java. Desenvolva a aplicação em camadas (padrão MVC). Cada enunciado deverá estar acessível através de um botão (JButton) . O resultado que conter apenas uma linha deverá ser visualizado em uma mensagem (JOptionPane). O resultado que conter várias linhas deverá ser visualizado em uma tabela (JTable).



3. Mostre o primeiro nome e endereço de todos os empregados que trabalham para o departamento 'Pesquisa'.
4. Encontre os nomes (primeiro nome) de todos os empregados que são diretamente supervisionados por "Joaquim".
5. Faça uma lista dos números e nomes de projetos que envolvam um empregado cujo último nome seja 'Will', mesmo que esse trabalhador seja o gerente do departamento que controla o projeto.
6. Para todo projeto localizado em 'Araucaria', liste o nome do projeto, o nome do departamento de controle e o último nome, endereço e data de nascimento do gerente do departamento.
7. Encontre o nome dos empregados que trabalham em *todos* os projetos controlados pelo departamento número 5.
8. Para cada projeto, liste o nome do projeto e o total de horas por semana (de todos os empregados) gastas no projeto.
9. Recupere os nomes de todos os empregados que trabalhem mais de 10 horas por semana no projeto 'Automatizacao'.
10. Recupere os nomes de todos os empregados que não trabalham em nenhum projeto.

11. Recupere a média salarial de todos os empregados do sexo feminino.
12. Para cada departamento, recupere o nome do departamento e a média salarial de todos os empregados que trabalham nesse departamento.

Atividade 2 – Java JDBC

1. Encontre o nome e o endereço de todos os empregados que trabalhem em pelo menos um projeto localizado em 'Curitiba', mas cujo departamento não se localiza em 'Curitiba'.
2. Liste os nomes de todos os empregados com dois ou mais dependentes.
3. Recupere o nome de empregados que não tenham dependentes.
4. Liste o último nome de todos os gerentes de departamento que não tenham dependentes.
5. Liste os nomes dos gerentes que tenham, pelo menos, um dependente.

Executando Alterações

Para executar alterações no banco, como **inserts**, **updates**, criação de tabelas, usa-se:

```
st = con.createStatement();  
st.executeUpdate("create table teste (cod int, nome char(50))");
```

Esse método não retorna resultado, somente gera exceção caso haja erro.

Recuperando Resultados

Os resultados de uma consulta ficam armazenados num objeto da classe `ResultSet`. Para buscar a primeira linha de resultados executa-se:

```
rs.next()
```

Se essa chamada retornar **false** é porque não existem mais registros, caso contrário, um registro foi recuperado.

Para recuperar os dados (campos) de um registro, usa-se a variedade das funções `get*` do `ResultSet`. Por exemplo:

```
String name = rs.getString("nome");  
int cod = rs.getInt("cod");
```

As strings passadas como parâmetro para as funções são os nomes dos campos. Pode-se passar também o índice do campo (0, 1, 2, ...), mas claro, o nome é muito mais legível.

Com essas funções também se pode ler toda uma grande variedade de dados, como longs, datas, etc.

Instruções Preparadas

Instruções preparadas permitem pré-compilar uma instrução para depois poder executá-la mais de uma vez, com uma performance melhor.

Como um objeto **Statement**, o objeto **PreparedStatement** é obtido a partir de um **Connection**:

```
PreparedStatement pstmt = con.prepareStatement(  
    "insert into teste (cod, nome) values (?, ?)");
```

Repare que há dois pontos de interrogação (?) na instrução. Isso são parâmetros da instrução, que devem ser preenchidos antes da sua execução:

```
pstmt.setInt(1, 1);  
pstmt.setString(2, "Camila");
```

Onde o primeiro parâmetro (1 e 2) é a posição do parâmetro. O segundo é o valor a ser colocado no lugar das interrogações. Assim como as funções `get*`, as funções `set*` podem inserir qualquer tipo de dados numa instrução preparada.

Depois de inseridos os parâmetros, pode-se executar a consulta:

```
pstmt.executeUpdate();
```

Se fosse uma query (select) pode-se também usar:

```
pstmt.executeQuery();
```

Sessão 08 – JDBC

Exercícios de Fixação 8 – Java JDBC

Liste as principais classes que o Java disponibiliza no pacote `java.sql` necessários para o desenvolvimento de aplicações java que possibilitem acesso a banco de dados?

Atividade 1 – Java JDBC

1. Uma indústria deseja manter um banco de dados sobre a folha de pagamento mensal de seus funcionários. Para isso, é lido de cada funcionário: o nome, o salário-base e uma categoria funcional (numérica: 1-operário, 2-administrativo ou 3-gerência). Os dados devem ficar armazenados em uma tabela no *PostgreSQL* e a cada vez que for solicitado, o contra-cheque de um funcionário deve ser exibido, com as seguintes informações: **nome do funcionário**, **salário bruto** (se da categoria 1, o próprio salário-base; se da categoria 2, o salário-base + 5%; se da categoria 3, o salário-base + 15%), **descontos** (se salário bruto maior ou igual a 1000, 3% sobre o salário bruto; caso contrário, 1% sobre o salário bruto) e **salário líquido** (salário bruto – descontos). Desenvolva uma aplicação GUI em Java que apresente as seguintes opções:
 - c) Incluir (solicitação dos dados de um funcionário e gravação de um registro com os dados da folha de pagamento no arquivo).
 - d) Listar (leitura dos dados da folha de pagamento de um funcionário do arquivo, mostrando as informações do contra-cheque).

Atividade 2 – Java JDBC

1. Defina um novo banco de dados no *PostgreSQL* que deverá possuir três tabelas. A primeira, denominada DEPARTAMENTO, deverá conter o código (valor inteiro) e seu nome (literal). A segunda, denominada OCUPAÇÃO, deverá conter o código (valor inteiro) e sua descrição (literal). A terceira, denominada FUNCIONÁRIO, deverá conter o número de registro do funcionário (valor inteiro), o nome (literal), o código de ocupação, o código do departamento em que esta lotado.
2. Desenvolva um aplicação GUI em Java que apresente as seguintes opções para manipulação do banco de dados implementado no enunciado 1 da atividade 2 – Java JDBC:
 - a) Uma tela (JFrame) que possibilite a manutenção de registros na tabela DEPARTAMENTO (inclusão, exclusão, consulta, alteração);
 - b) Uma tela (JFrame) que possibilite a manutenção de registros na tabela OCUPAÇÃO (inclusão, exclusão, consulta, alteração);
 - c) Uma tela (JFrame) que possibilite a manutenção de registros na tabela FUNCIONÁRIO (inclusão, exclusão, consulta, alteração).

Os dados das tabelas relacionadas deverão estar acessíveis em objetos JComboBox;

- d) Implementar uma classe executável que disponibilize um Menu com opções de acesso para as telas implementadas nos itens a, b e c.

Stored Procedures

São procedimentos armazenados no banco de dados, prontas para execução, que podem ou não ter parâmetros, e também podem ou não ter algum resultado.

Para isso, usa-se a classe `CallableStatement`. Supondo que exista uma *stored procedure* no nosso banco, chamada: **sp_teste**, que recebe dois parâmetros, um código de entrada e um nome de saída. Para chamá-la, faz-se:

```
CallableStatement cstmt = con.prepareCall("{call sp_teste(?, ?)}");
cstmt.setInt(1, 1);
cstmt.registerOutParameter(2, java.sql.Types.VARCHAR);
cstmt.executeUpdate();
System.out.println(cstmt.getString(2));
```

Quando se quer que a Stored Procedure retorne um valor, sem ser um parâmetro de retorno, a sintaxe da string dentro do `prepareCall` é:

```
{?= call [, , ...]}
```

Nesse caso, a primeira interrogação ({ ?= ...}) terá o índice 1 para o `registerOutParameter`.

Todos os parâmetros de saída (OUT) devem ser registrados com `registerOutParameter`, antes de chamar o procedimento.

Fechando Recursos

Depois de usados, as conexões, `Statements` e `ResultSets` devem ser fechadas:

```
con.close();
st.close();
pstmt.close();
cstmt.close();
rs.close();
```

Transações

Uma transação é um conjunto de um ou mais comandos que são executados, completados e então **validados** (commit) ou **cancelados** (rollback). Quando uma transação é validada ou cancelada, uma outra transação se inicia.

A conexão em java é, por default, **auto-commit**. Isso significa que sempre que um comando é executado, um **commit** é executado no banco de dados. Se o comando **auto-commit** for desabilitado, então deve-se fazer explicitamente o commit ou rollback das transações. A maioria dos drivers JDBC suportam transações.

Um exemplo de código:

```
con.setAutoCommit(false);
Statement st = con.createStatement();
....
con.commit();
con.setAutoCommit(true);
```

O comando **setAutoCommit** é aplicado à conexão, e define se os commits e rollbacks serão ou não efetuados automaticamente. Caso não sejam, a conexão disponibiliza os comandos **commit()** e **rollback()**.

Execução em Batch

É usado quando se tem muitas atualizações para serem feitas no banco de dados. Muitas vezes, atualizações em batch são mais rápidas do que atualizações feitas uma a uma.

Exemplo:

```
con.setAutoCommit(false);
Statement stmt = con.createStatement();

stmt.addBatch("INSERT INTO COFFEES " +
              "VALUES('Amaretto', 49, 9.99, 0, 0)");
stmt.addBatch("INSERT INTO COFFEES " +
              "VALUES('Hazelnut', 49, 9.99, 0, 0)");
stmt.addBatch("INSERT INTO COFFEES " +
              "VALUES('Amaretto_decaf', 49,
                    10.99, 0, 0)");
stmt.addBatch("INSERT INTO COFFEES " +
              "VALUES('Hazelnut_decaf', 49,
                    10.99, 0, 0)");

int [] updateCounts = stmt.executeBatch();
con.commit();
con.setAutoCommit(true);
```

A linha:

```
con.setAutoCommit(false);
```

é usada para desabilitar o auto-commit da conexão. Isso é usado para que não aconteçam erros imprevistos na execução do batch.

A linha:

```
int [] updateCounts = stmt.executeBatch();
```

executa todos os comandos que foram inseridos com **addBatch**, na ordem em que foram inseridos. Quando terminados, esse método retorna, para cada um dos comandos, a quantidade de linhas afetadas por eles (por isso que retorna um array de inteiros). Como o auto-commit foi desligado, deve-se explicitamente efetuar o commit.

Sessão 09 – JDBC

Exercícios de Fixação 9 – Java JDBC

Liste as principais recursos que você consegue identificar que sejam necessários para o desenvolvimento de aplicações java que possibilitem acesso a banco de dados?

Atividade 1 – Java JDBC

1. Uma loja decidiu fazer o levantamento de informações em relação aos produtos que disponibiliza para venda, juntamente com informações de seus fornecedores. Durante a fase de especificação de requisitos e entrevistas com o usuário os analistas conseguiram identificar a necessidade da criação da entidade Produto (nome, preço de compra, preço de venda, estoque e estoque mínimo) e também a entidade Fornecedor (nome, endereço, telefone, contato). Verificou-se também que a relação entre estas entidades é de (1:N), sendo necessária a criação de um relacionamento entre Produto e Fornecedor, visto que um produto pode ser fornecido por apenas um fornecedor e um fornecedor pode fornecer mais de um produto para esta empresa.

Desenvolva os problemas descritos abaixo:

- 2.1. Crie um banco de dados no *PostgreSQL* com o nome de **estoque**.
 - 2.1.1. Crie as tabelas, com base nas estruturas de dados descritas anteriormente, que irão compor o banco de dados no SGBD *PostgreSQL*.
- 2.2. Desenvolva uma aplicação GUI Java que disponibilize as seguintes funcionalidades:
 - 2.2.1. Classes modelos para cada uma das tabelas implementadas no banco de dados da aplicação;

- 2.2.2. Classes DAO que irão possibilitar o acesso ao banco de dados para manutenção de registros das tabelas implementadas no banco de dados da aplicação;
- 2.2.3. Menu principal que possibilite acesso as telas para manutenção de registros da tabela produto, fornecedor e também para relação entre produto-fornecedor;
- 2.2.4. Tela que possibilite a manutenção de dados de produtos (inclusão, alteração, consulta, exclusão).
- 2.2.5. Tela que possibilite a manutenção de dados de fornecedores (inclusão, alteração, consulta, exclusão).
- 2.2.6. Tela que possibilite, através de cliques em botões, a visualização dos resultados das seguintes consultas SQL:
 - 2.2.6.1. A média geral da quantidade em estoque dos produtos da empresa;
 - 2.2.6.2. O nome e o contato do fornecedor do produto que tem o menor preço de venda;
 - 2.2.6.3. O nome do produto que tem o maior preço de compra, juntamente com seu fornecedor e contato;
 - 2.2.6.4. Para cada fornecedor, o total de produto que ele fornece;
 - 2.2.6.5. A média dos preços de compra de cada um dos fornecedores
 - 2.2.6.6. O total em R\$ dos produtos armazenados na empresa, tendo como base o preço de compra.

3. Atividade 2 – Java JDBC

1. Uma indústria deseja manter um banco de dados sobre a folha de pagamento mensal de seus funcionários. Para isso, é lido de cada funcionário: o nome, o salário-base e uma categoria funcional (numérica: 1-operário, 2-administrativo ou 3-gerência). Os dados devem ficar armazenados em uma tabela no *PostgreSQL* e a cada vez que for solicitado, o contracheque de um funcionário deve ser exibido, com as seguintes informações: **nome do funcionário**, **salário bruto** (se da categoria 1, o próprio salário-base; se da categoria 2, o salário-base + 5%; se da categoria 3, o salário-base + 15%), **descontos** (se salário bruto maior ou igual a 1000, 3% sobre o salário bruto; caso contrário, 1% sobre o salário bruto) e **salário líquido** (salário bruto – descontos). Desenvolva um aplicação GUI em Java que apresente as seguintes opções:
 - a) Incluir (solicitação dos dados de um funcionário e gravação de um registro com os dados da folha de pagamento no arquivo).
 - b) Listar (leitura dos dados da folha de pagamento de um funcionário do arquivo, mostrando as informações do contracheque).
3. A comissão organizadora de um campeonato de atletismo decidiu apurar os resultados da competição através de um processamento eletrônico. De cada atleta são lidos: código de identificação, nome e o tempo gasto em cada uma das 5 provas que formam a competição. Esses dados são armazenados em um **arquivo**. Ao final de cada prova, a pontuação é atribuída de acordo com o tempo realizado em cada prova (quem gastou

menos tempo está em 1º lugar) e é feita da seguinte maneira: 1º colocado: 5 pontos, 2º colocado: 3 pontos e 3º colocado: 1 ponto. Essa pontuação é atualizada na pontuação geral do campeonato. Desenvolva uma aplicação GUI em Java que apresente as seguintes opções:

- a) Inclusão dos dados (solicitação dos dados de um atleta e gravação de um registro no arquivo)
- b) Classificação (leitura dos dados dos atletas e processamento dos tempos e pontos)

6

Threads – sessão 10

Exercício de nivelamento – JDBC

Você já desenvolveu alguma aplicação que utilizou acesso a banco de dados? Em que linguagem? Que banco de dados utilizou?

☐ Sistema Operacional e Processos

☐ Threads em Java

☐ Operações sobre Threads

☐ Prioridades de Threads

☐ Programação concorrente

Uma **thread** é uma ramificação do programa, ou trecho, em execução, que é executada paralelamente. A JVM permite que uma aplicação tenha múltiplas threads executando concorrentemente.

Toda thread tem uma prioridade. As que possuem alta prioridade são executadas preferencialmente àquelas com baixa prioridade. Quando um código rodando em alguma thread cria um novo objeto Thread, este terá sua prioridade inicialmente configurada igual à thread que o criou.

Existem dois tipos de threads: *daemon* e *user*. As threads *daemon* são do sistema e *user* são as criadas pelo usuário.

Quando JVM inicia, existe usualmente uma thread do usuário, que tipicamente chama o método `main` de alguma classe. O JVM continua a execução até:

- O método `exit` da classe `Runtime` tenha sido chamado e o gerenciador de segurança tenha permitido a operação de saída;
- Todas as threads que não são daemon tenham terminado, ou retornando da chamada `run`, ou levantando uma exceção que se propaga até o método `run`.

Múltiplas Linhas de Execução (*Thread*)

Muitos dos problemas de software propostos em situações reais podem ser resolvidos utilizando várias threads de controle. Por exemplo, programas interativos que exibem dados graficamente frequentemente precisam permitir aos usuários alterar parâmetros de exibição em tempo real. Os programas interativos geralmente têm um melhor comportamento dinâmico utilizando threads.

Esse capítulo discute profundamente os conceitos de thread e, através de exemplos detalhados, esclarece o modelo Java de threading.

O que são?

Uma perspectiva simples, mais útil, de um computador é a de que ele possui uma CPU, que realiza a computação, uma ROM, que contém o programa que a CPU executa, e uma RAM, que armazena os dados em que o programa opera. Nessa perspectiva simples, somente um trabalho é realizado por vez. Uma perspectiva mais completa da maioria dos sistemas de computação modernos permite a possibilidade de um ou mais trabalhos realizados ao mesmo tempo, ao menos, aparentemente.

É importante considerar as implicações do ponto de vista da programação. A realização de mais de um trabalho é semelhante à existência de mais de um computador. Sendo assim, considere um thread ou linha de execução como o encapsulamento de uma CPU virtual com seus próprios dados e código de programa. A classe `Java.lang.Thread` nas principais bibliotecas Java permite criar e controlar threads.

Início do *Thread* e Programação de *Thread*

Apesar do Thread ter sido criado sua execução não começa imediatamente. Para inicia-la torna-se necessária a utilização do método `start()`. Esse método pertence à classe `Thread` do Java e a forma de utilização é `t.start()`, onde `t` é um objeto instanciado da classe `Thread`. Nesse ponto, a CPU virtual, incorporada ao thread, torna-se executável. Você pode pensar nesse evento como a ligação da CPU virtual.

Apesar do thread se tornar executável, ele não é necessariamente iniciado de imediato. Em uma máquina que tenha efetivamente apenas uma CPU, é óbvio que ela só pode executar uma instrução por vez. Consideraremos, agora, a forma como a CPU é alocada quando mais de um thread puder estar trabalhando com utilidade.

Na tecnologia Java, os threads normalmente são preemptivos, mas não necessariamente fracionados no tempo, (é um erro comum acreditar que preemptivo seja uma palavra sofisticada para "fracionada no tempo"). O modelo de um programador preemptivo estabelece que vários threads podem estar prontos para execução, mas somente um está efetivamente sendo executado.

Esse processo continua a ser executado até que ele não seja mais executável ou que outro processo de prioridade maior se torne executável. Nesse caso, dizemos que o thread de menor prioridade é precedido pelo thread de maior prioridade.

Um thread pode deixar de estar pronto por uma série de motivos. Ele pode executar uma chamada a `Thread.sleep()`, que solicita deliberadamente uma pausa por um período, ou pode precisar aguardar um dispositivo externo mais lento, como uma unidade de disco ou de uma interação com o usuário.

Todos os threads executáveis, mas não executados, são mantidos em filas segundo as respectivas prioridades. O primeiro thread da fila de maior prioridade que não estiver vazia será executado. Quando um thread para de ser executado por causa de uma precedência, ele é tirado do estado de execução e é colocado no fim da fila de espera.

Analogamente, um thread que se torna executável após estar bloqueado (inativo ou aguardando operações de E/S, por exemplo) sempre se posiciona no fim da fila.

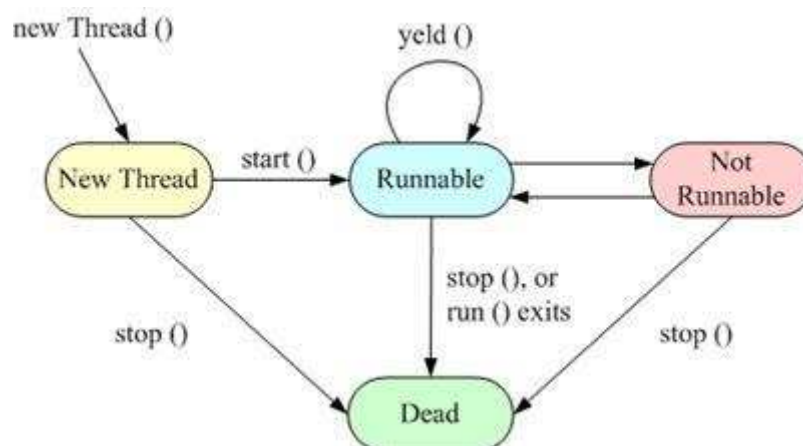


Figura 08 – thread, fila de executáveis.

Pelo fato dos threads em Java não dividirem necessariamente o tempo, na ausência de uma compreensão mais avançada e para tornar a estrutura do programa mais adequada, você deve garantir que o código dos seus threads dê chance a outros threads de serem executados de tempos em tempos. Como mostrado no exemplo a seguir, isso pode ser obtido emitindo a chamada `sleep()` em intervalos, como um laço principal.

```
1  public class xyz implements Runnable {
2      public void run() {
3          while (true) {
4              // faz várias coisas interessantes
5              ...
6              // Dá oportunidade a outros threads
7              try {
8                  Thread.sleep(10);
9              } catch (InterruptedException e) {
10                 // Esse thread foi interrompido por outro
11             }
12         }
13     }
14 }
```

Observe a utilização de try e catch. Observe também que a chamada `sleep()` é um método static na classe `Thread`, e por isso é referida como `Thread.sleep(x)`. O argumento especifica o número de milissegundos que o thread deve permanecer inativo. A execução do thread não será retomada até o fim desse período e mais algum tempo para que outros threads sejam bloqueados, permitindo que esse thread realmente inicie.

Outro método na classe `Thread`, o `yield()`, pode ser emitido para dar oportunidade a outros threads de serem executados sem interromper efetivamente o thread atual, a menos que seja necessário. Se outros threads de mesma prioridade forem executáveis, `yield()` coloca o thread chamador no fim da respectiva fila executável e permite que outro thread inicie. Se nenhum outro thread for executável com a mesma prioridade, `yield()` não atuará.

É importante observar que uma chamada `sleep()` pode conceder a threads de menor prioridade uma oportunidade de serem executados. O método `yield()` só dá a mesma oportunidade a threads de mesma prioridade.

Quando um thread retorna do fim do respectivo método `run()`, ele morre. Em seguida, ele não pode ser executado novamente.

O método `stop()` pode ser emitido para forçar a parada de um thread. Deve ser utilizado em uma instância específica da classe `Thread`. Na linha 11 do exemplo abaixo, o thread `t` é forçado a parar.

```
1  public class xyz implements Runnable {
2      // Algumas coisas que queremos em um thread
3  }
4  -----
5  public class tTest {
6      public static void main(String args[]) {
7          Runnable r = new xyz();
8          Thread t = new Thread(r);
9          t.start();
10         // faz outras coisas
11         if (time_to_kill)
12             t.stop();
13     }
14 }
```

Em um determinado trecho de código, é possível obter uma referência ao thread atual utilizando o método `currentThread()` do `Thread` estático, como mostrado no próximo exemplo na linha 6.

```
1  public class xyz implements Runnable {
2      public void run() {
3          while (true) {
4              // várias coisas interessantes
5              if (time_to_die)
6                  Thread.currentThread().stop();
7          }
8      }
9  }
```

Observe que, nesse caso, a execução de `stop()` destruirá o contexto de execução atual e, portanto, o laço `run()` não será mais executado nesse contexto.

Teste de um *Thread*

Às vezes, é possível para um thread estar em um estado desconhecido (isso pode ocorrer se o código não estiver controlando diretamente um determinado thread). É possível descobrir se um thread ainda é viável através do método `isAlive()`.

Estar vivo não implica que o thread esteja sendo executado. Apenas indica que ele foi iniciado e não foi interrompido por meio de `stop()`, nem atingiu o fim do método `run()`.

Suspendendo Threads

Existem vários mecanismos capazes de interromper a execução de um thread temporariamente. Após esse tipo de suspensão, a execução pode ser retomada como se nada tivesse acontecido. O thread aparenta simplesmente ter executado uma instrução muito devagar.

O método `sleep()` foi introduzido anteriormente e é utilizado para pausar um thread por um período de tempo. Lembre-se de que, normalmente, o thread não é retomado no instante em que o período de inatividade termina, pois é provável que outro thread esteja sendo executado nesse instante e não possa ser desprogramado, a menos que: a) o thread reativado seja de prioridade superior; b) o thread que está sendo executado seja bloqueado por algum motivo; ou c) a divisão do tempo esteja habilitada. Na maioria dos casos, nenhuma das duas últimas condições prevalece imediatamente.

Às vezes, é conveniente suspender a execução de um thread indefinidamente. Nesse caso, algum outro thread será responsável pela retomada da execução. `suspend()` e `resume()` foram criados com esse propósito.

Observe que, em geral, um thread pode ser suspenso por qualquer trecho de código que possua um handle do thread – ou seja, uma variável que o referencia.

Obviamente, contudo, o thread só pode ser reiniciado por outro thread que não seja o próprio, pois o thread suspenso não está executando nenhum código.

```
1  public class xyz implements Runnable {
2      public void run() {
3          // faz várias coisas interessantes...
4
5          // aguarda até receber ordem para continuar
6          Thread.currentThread().suspend ();
7          // faz mais coisas...
8      }
9  }
-----
10 ...
11 Runnable r = new xyz();
12 Thread t = new Thread(r);
13 t.start();
14 // pausa para xyz ser executado por um momento
15 // pressupor que atinge seu suspend()...
16 Thread.sleep(1000);
17
18 // torna xyz executável novamente
19 t.resume();
20
21 // executa efetivamente
22 Thread.yield();
```

Além dos métodos `sleep()`, `suspend()` e `resume()`, um outro método pode ser utilizado para interromper a execução de um thread temporariamente, o método `join()`.

O método `join()` faz com que o thread atual aguarde o término do thread em que o método `join` foi chamado. Como mostrado no exemplo a seguir, a chamada a `join()` na linha 6 provoca a espera do thread atual até que `TimerThread tt` termine.

Como mostrado na linha 6 do exemplo a seguir, `join()` também pode ser chamado com um valor de tempo de espera em milissegundos. O método `join()` suspenderá o thread atual pelo número de milissegundos do tempo de espera ou até que o thread `tt` termine, o que ocorrer primeiro.

```
1  TimerThread tt = new TimerThread (100);
2  tt.start ();
3  ...
4  public void timeout() {
5      // Aguarda aqui o thread temporizador concluir
6      tt.join ();
7      ...
8      // Continua nesse thread
9      ...
10 }
```

Implementar Runnable Versus Estender Thread

Dadas às possibilidades de criação de threads, como decidir entre elas? Há alguns pontos a favor de cada técnica:

▣ *A favor da implementação de `Runnable`:*

▣ *A favor da extensão de `Thread`:*

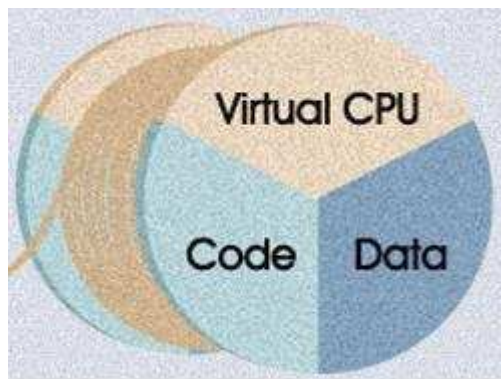


Figura 09 – As três partes de um thread.

Java é a única entre as linguagens de programação a de uso geral e popular, no sentido de que torna as primitivas de concorrência disponíveis para o programador de aplicativos. O programador especifica quais aplicativos contêm fluxos de execução (threads¹), cada thread designando uma parte de um programa que pode executar concorrentemente com outros threads. Essa capacidade, chamada multithreading, fornece recursos poderosos para o programador Java não disponíveis em outras linguagens como C ou C++, chamadas de linguagens de um único thread.

Uma dos exemplos mais contundentes de uma aplicação multithread é um browser web. Como você sabe, a maior parte dos browsers permite que você abra mais de uma janela e descarregue vários arquivos ao mesmo tempo. Para permitir que você execute várias tarefas dessa maneira, o browser utiliza as multithreds.

Implementando uma *Thread* – Método Mais Comum

Você pode implementar threads em programas Java de duas maneiras. Normalmente implementa-se threads através da instância de objetos que estendem a classe Thread. Como você já sabe, estender uma classe permite que o novo objeto herde as propriedades da classe progenitora. Logo após é só reescrever o método run() como ilustrado no trecho de código a seguir:

```
class ExemploThread extends Thread
{
    public void run(){
        //Implemente aqui o código para as threads da classe
    }
}
```

¹ Embora o termo *thread* venha sendo traduzido por “fluxo de execução”, preferimos mantê-lo no original por se tratar de uso já consagrado no meio acadêmico.

Esta primeira forma não é possível de implementação caso a sua classe já herde de qualquer outra classe que não a classe Thread. Isso poderá ocorrer uma vez que Java não permite herança múltipla. A solução é implementar a interface Runnable, que é o segundo método de criação de um thread. Lembre-se que de interfaces são o modo de Java proporcionar as vantagens da herança múltipla.

O exemplo a seguir implementa threads para executar um aplicativo que disponibiliza relógios com horários de 6 países diferentes. A execução deste aplicativo utiliza de threads para efetuar a movimentação do ponteiro de todos os relógios.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.util.*;

public class TimerTest
{   public static void main(String[] args)
    {   JFrame f = new TimerTestFrame();
        f.show();
    }
}

class TimerTestFrame extends JFrame
{   public TimerTestFrame()
    {   setSize(450, 300);
        setTitle("TimerTest");

        addWindowListener(new WindowAdapter()
        {   public void windowClosing(WindowEvent e)
            {   System.exit(0);
            }
        });

        Container c = getContentPane();
        c.setLayout(new GridLayout(2, 3));
        c.add(new ClockCanvas("Brasil", "GMT-2"));
        c.add(new ClockCanvas("Taipei", "GMT+8"));
        c.add(new ClockCanvas("Berlin", "GMT+1"));
        c.add(new ClockCanvas("New York", "GMT-5"));
        c.add(new ClockCanvas("Cairo", "GMT+2"));
        c.add(new ClockCanvas("Bombay", "GMT+5"));
    }
}

interface TimerListener
{   void timeElapsed(Timer t);
}

class Timer extends Thread
{   public Timer(int i, TimerListener t)
    {   target = t;
        interval = i;
        setDaemon(true);
    }

    public void run()
    {   try
        {   while (!interrupted())
            {   sleep(interval);
                target.timeElapsed(this);
            }
        }
        catch (InterruptedException e) {}
    }

    private TimerListener target;
    private int interval;
}
```

```
}

class ClockCanvas extends JPanel
    implements TimerListener
{
    public ClockCanvas(String c, String tz)
    {
        city = c;
        calendar = new GregorianCalendar(TimeZone.getTimeZone(tz));
        Timer t = new Timer(1000, this);
        t.start();
        setSize(125, 125);
    }

    public void paintComponent(Graphics g)
    {
        super.paintComponent(g);
        super.setBackground(Color.white);
        g.drawOval(0, 0, 100, 100);
        double hourAngle = 2 * Math.PI
            * (seconds - 3 * 60 * 60) / (12 * 60 * 60);
        double minuteAngle = 2 * Math.PI
            * (seconds - 15 * 60) / (60 * 60);
        double secondAngle = 2 * Math.PI
            * (seconds - 15) / 60;
        g.drawLine(50, 50, 50 + (int)(30
            * Math.cos(hourAngle)),
            50 + (int)(30 * Math.sin(hourAngle)));
        g.drawLine(50, 50, 50 + (int)(40
            * Math.cos(minuteAngle)),
            50 + (int)(40 * Math.sin(minuteAngle)));
        g.drawLine(50, 50, 50 + (int)(45
            * Math.cos(secondAngle)),
            50 + (int)(45 * Math.sin(secondAngle)));
        g.drawString(city, 0, 115);
    }

    public void timeElapsed(Timer t)
    {
        calendar.setTime(new Date());
        seconds = calendar.get(Calendar.HOUR) * 60 * 60
            + calendar.get(Calendar.MINUTE) * 60
            + calendar.get(Calendar.SECOND);
        repaint();
    }

    private int seconds = 0;
    private String city;
    private int offset;
    private GregorianCalendar calendar;

    private final int LOCAL = 16;
}
}
```

Tratamento de Regiões Críticas com Threads

O problema de regiões críticas em certas aplicações deve ser considerado para que não exista erro no processamento de dados. Tais erros ocorrem devido à falta de controle de acesso a recursos sem o devido controle de compartilhamento. Suponha que você esteja desenvolvendo uma aplicação java onde duas ou mais threads tentam executar simultaneamente uma instrução de adição da saldo monetário a uma conta corrente. Vale lembrar que estas são operações atômicas, sendo que a instrução de adição poderia ser processada da seguinte forma pelas threads:

1. Carregue **saldo da conta corrente** em um registrador;
2. Adicione o novo valor monetário ao **saldo da conta corrente**;

3. Mova o resultado do atualizado **saldo da conta corrente** para a variável em memória

O problema é que, durante a execução de uma das threads, por exemplo, a primeira, esta conseguisse executar apenas as etapas 1 e 2, sendo interrompida logo a seguir. Neste ponto uma segunda thread iniciaria a sua execução e modificaria os valores de entrada para a primeira thread. Quando a primeira thread voltar a ser executada, o total monetário para o saldo da conta corrente não estaria mais correto, gerando problemas de consistências de dados manipulados pela aplicação desenvolvida.

Para que esse tipo de problema não ocorra, devemos ter alguma garantia de que, quando uma thread começar a inserir um elemento no buffer, ela poderá completar a operação sem ser interrompida. A maioria dos ambientes de programação forçam os programadores a mexer com os chamados semáforos e regiões críticas para obter acesso sem interrupção a um recurso. Já a linguagem de programação Java disponibiliza um recurso inspirado nos monitores inventados por Tony Hoare. Para tanto, basta que qualquer operação compartilhada por várias threads seja marcada como *synchronized*, da seguinte forma:

```
Public synchronized void transferir_saldo(int de, int para, int quantia){  
    //implementar funcionalidades do método  
}
```

Com isso, a aplicação desenvolvida em Java consegue garantir que, qualquer thread que inicie a execução de um método sincronizado conclua a sua execução antes que outra thread passe a executar o mesmo método sobre o mesmo objeto. O método sincronizado desativa automaticamente a chamada da segunda thread e fica em espera até que o método seja liberado para execução. Essa espera ocorre numa fila de espera onde são colocadas as threads que estão esperando para manipular o objeto que está sendo utilizado através de métodos sincronizados. Esta fila obedece a uma ordem de prioridade existente dentre as threads que se encontram nela.

Em java, os responsáveis por bloquear e notificar quando um objeto está disponível são os objetos monitores. Esses objetos são instâncias de classes que implementam um ou mais métodos sincronizados.

O problema é que existem aplicações que possam querer que threads que estejam com acesso exclusivo a um determinado objeto de uma aplicação seja interrompida para que os valores de determinados atributos satisfaçam as necessidades da aplicação (por exemplo, só transferir saldo entre contas se a conta de origem tiver saldo suficiente). Para tanto, torna-se necessário o uso do método `wait()`.

Quando uma thread chama o método `wait()` dentro de um método sincronizado, ela é desativada sendo colocada na fila de espera para o objeto em questão. Com isso, outras threads passam a ter acesso ao objeto. Com isso, esperasse que dentro de um determinado tempo de execução da aplicação o objeto passe a ter saldo suficiente para executar a transferência e, através do método `notify()` o java desperta a thread original que esperava pela chance de executar sua funcionalidade.

O ponto a se considerar é que o método `notify()` deverá ser chamado por alguma outra thread que esteja trabalhando sobre o objeto em questão, pois as threads em espera não são

ativadas automaticamente quando nenhuma outra thread está trabalhando no objeto. Como resultado de implementação teríamos o teste de condição e ação de transferência dentro do mesmo método `synchronized`, como mostrado abaixo:

```
Public synchronized void transferir_saldo(int de, int para, int quantia){  
    While(saldo < quantia){  
        Wait();  
    }  
    //transferir fundos  
}
```

Um Resumo do Funcionamento do Mecanismo de Sincronismo

Apresentamos a seguir um resumo do funcionamento do mecanismo de sincronismo para controle de concorrência em threads sobre um determinado objeto da aplicação:

1. Para chamar um método `synchronized`, o parâmetro implícito não pode ser bloqueado. Chamar o método bloqueia o objeto. Retornar da chamada desbloqueia o objeto parâmetro implícito. Assim, apenas uma linha de execução por vez pode executar métodos `synchronized` em um objeto específico.
2. Quando uma linha de execução executa uma chamada `wait()`, ela renuncia ao bloqueio do objeto e entre em uma lista de espera.
3. Para remover uma linha de execução da lista de espera, alguma outra linha de execução deve fazer uma chamada a `notifyAll()` ou `notify()`, no mesmo objeto.

As regras de agendamento são incontestavelmente complexas, mas é realmente muito simples colocá-las em prática. Basta seguir estas quatro regras:

1. Se duas ou mais linhas de execução modificam um objeto, declare os métodos que realizam as modificações como `synchronized`. Os métodos somente de leitura que são afetados por modificadores de objeto também ser `synchronized`.
2. Se uma linha de execução precisa esperar que o estado de um objeto mude, ele deve esperar dentro do objeto, e não fora, entrando em um método `synchronized` e chamando `wait()`.
3. Quando um método muda o estado de um objeto, ele deve chamar `notifyAll()`. Isso dá às linhas de execução que estão esperando uma chance de ver se as circunstâncias mudaram.
4. Lembre-se de que `wait()` e `notify()/notifyAll()` são métodos da classe `Object` e não da classe `Thread`. Confira se suas chamadas a `wait()` correspondem a uma notificação no *nomes de objetos*.

Exercícios de Fixação 10 – Thread

Os comandos a seguir estão corretos ou não? Justifique.

```
Thread myThread = new Thread() ;  
myThread.run() ;
```

Atividade 1 – Thread

1. Escreva um programa que instancie e inicie duas threads: uma que conta até 10 e outra que conta até 20, através do método construtor.
2. Implemente uma classe Fila cujos métodos acrescentar e remover são sincronizados. O tamanho da fila é 10. Forneça uma thread, chamada produtor, que insira um dado de valor aleatório (de 1 a 50) continuamente. Quando a fila ficar cheia, a thread espera. Forneça uma segunda thread, chamada consumidor, que remova um dado da fila e imprima seu valor continuamente, contanto que a fila não esteja vazia. Quando a fila estiver vazia, a thread espera. Tanto a thread consumidor quanto a produtor devem executar 50 iterações.
3. Fazer um programa em Java que implemente uma classe Java que possibilite a instância de objetos da classe Pessoa (nome, idade e sexo) e posterior gravação dos mesmos num arquivo em disco. Os objetos instanciados deverão ser ativados com tempos aleatórios de pausa e possibilitar a gravação dos mesmos em arquivo.
4. Fazer um programa em Java que implemente métodos de leitura e cálculos serializados para objetos de consulta distintos que serão ativados através de uma interface JFrame que irão consultar o arquivo gerado no exercício 2 para calcular e responder o total de pessoas do sexo feminino e a média geral de idades das pessoas cadastradas.

Atividade 2 – Thread

1. Construa, na forma de Thread, uma classe capaz de realizar a ordenação de um array de valores do tipo double (passado como argumento ao construtor). A ordenação pode ser feita com o método sort da classe Array ou então implementada com qualquer método direto de ordenação (bubble sort, por exemplo). Escreva um programa que seja capaz de criar várias dessas threads, passando arrays de tamanho determinado pelo usuário e preenchidos com valores aleatórios entre 0 e o tamanho do próprio array.
2. Fazer um programa em Java que implemente uma classe Java que possibilite a instância de 4 objetos onde cada um possibilitará a realização de um dos quatro cálculos numéricos básicos entre dois números. Os objetos deverão ser instancias de uma classe extends Thread que deverá receber os números e o sinal da operação básica como parâmetro e escrever o resultado do cálculo. Durante a chamada deverão ser informados tempos aleatórios de pausa na execução dos objetos.

Bibliografia

- DEITEL. Java Como Programar. 6a. ed. São Paulo: Pearson Prentice Hall, 2005.
- HORSTMANN, Cay. CORNELL, Gary. Core Java 2 – Fundamentals. The Sun Microsystems Press Java Series. Vol. I. 2003.
- JANDL Junior, Peter. Java Guia do Programador. São Paulo: Novatec, 2007.
- POSTGRESQL. The Java Tutorials. Disponível em:
<http://docs.PostgreSQL.com/javase/tutorial/>. Acessado em: 29 jan. 2013.
- SIERRA, Kathy; BATES, Bert. Certificação Sun Para Programador Java 6 Guia de Estudo. São Paulo: Rio de Janeiro, 2008.