PW 02

Introdução ao *Java*

Versão: 2.0.0

DIREITOS AUTORAIS

Todos os direitos são reservados para ***???***. Nenhuma parte deste material poderá ser reproduzida, extraída, copiada ou transmitida de qualquer forma em qualquer língua sem prévia autorização por escrito.

A Violação destes direitos é protegida pela Lei 5988 de 14/12/1973 e é tratado como crime no art. 184 e parágrafos, do Código Penal, CF Lei Nº 6895 de 17/12/1980.

Expediente:

Desenvolvimento do Material:
Arnaldo Gonçalves de Sousa

Java Certification

- Sun Certified Programmer For Java[Tm] 2 Platform
- Sun Certified Instructor For Java[Tm] 2 Platform (T3 Basic)
- Sun Certified Instructor For Java[Tm] 2 Platform (T3 Advanced)
- IBM Certified Instructor For Web Services (T3 – Webservices For J2EE Developers)
- ICS Java Developer - Prometric Test Center Recognize

Experiência Como Docente

IBM Corporation (Desde 2002) - ibm.com/services/learning/br
Sun Microsystems (Desde 1999) - sun.com.br
Universidade Presbiteriana Mackenzie (Desde 2003) - mackenzie.br

Editoração:
Elza F. Domingues

Revisão:
Arnaldo Gonçalves de Sousa

Direitos Autorais:
Todos os direitos são reservados para ***???***. Nenhuma parte deste material poderá ser reproduzida, extraída, copiada ou transmitida de qualquer forma em qualquer língua sem prévia autorização por escrito.

A Violação destes direitos é protegida pela Lei 5988 de 14/12/1973 e é tratado como crime no art. 184 e parágrafos, do Código Penal, CF Lei Nº 6895 de 17/12/1980.

Apresentação

Introdução

Nossa Metodologia

Objetivo do Curso

Descrição

Este curso fornece ao aluno conhecimento sobre como entender as regras e as estruturas de Programação da Linguagem *Java* bem como suas principais **API's** e a Implementação dos Conceitos da Orientação a Objetos na Linguagem.

Duração

A duração deste curso é de 40 horas dividido em módulos de 4 ou 8 horas diárias.

Público Alvo

Programadores que já possuam conhecimentos em Orientação a Objetos e pretendam trabalhar com a tecnologia *Java*. Esse curso é considerado o treinamento fundamental para aqueles que desejam aprender *Java*.

Pré-requisitos do Aluno

É exigido do aluno conhecimento gerais sobre:

- Lógica de Programação.
- Conhecimentos básicos de orientação a objetos.
- Operações básicas com arquivos texto (abrir, criar, salvar e etc.).
- Ter feito **PW01** ou ter conhecimentos equivalentes.

O não atendimento ou omissão destes pré requisitos isentam a *FastTraining* de qualquer responsabilidade por não aproveitamento.

Objetivo do Curso

Ao final deste curso você estará apto para:

- Conhecer os conceitos de portabilidade *Java*.
- Avaliar as inúmeras aplicações de sistemas em *Java*.
- Fazer comparativo com outras linguagens orientadas a objetos.
- Aplicar as conversões de variáveis (primitive *casting*).
- Uso de identificadores, palavras-chaves e tipos de dados.
- Entendendo as expressões e as estruturas de controle de fluxo.
- Trabalhando com *Arrays*.
- (*) Aprender a fazer conversões entre bases numéricas (decimal, binária, octal, hexadecimal).
- (**) Estudar profundamente os operadores da linguagem Java.
- Desenvolver programas *Java* orientados a objetos, utilizando recursos de herança, classes abstratas, interface, polimorfismo.
- Conhecer recursos de *File* **I/O**.
- Compilar e rodar programas gráficos (**API AWT**) com painéis, botões, títulos, campos de texto e áreas de texto.
- Desenvolver programas com uso de *Frames* e *Menus*, bem como fazer o tratamento de eventos.
- (**) Utilizar a **API** de *Thread*, para construir aplicativos com múltiplas linhas de execução.
- Aplicar os conceitos de comunicação por *Sockets* em uma rede **TCP/IP**.
- (**) Conhecer o *Java 2 Collections Framework* e implementar *collection*.

Legenda:

(*) Item não visto em outros cursos do mercado, mas que fazem parte da prova de certificação.

(**) Maior conteúdo do mercado em um curso.

Certificações

Se seu objetivo é a certificação **PCP JAVA**, deverá fazer também os cursos **PW01, PW02** e **PW100** e o exame oficial **310-035**.

Se seu objetivo é a certificação **DCP JAVA** deverá fazer também os cursos **PW01**, **PW02**, **PW100**, **PW03** e **PW04** e os exames oficiais **310-035** e **320-027**.

Se seu objetivo é a certificação **ACCP JAVA** deverá fazer os cursos **PW01**, **PW02**, **PW100**, **PW03, PW04, PW05, PW06** e **PW08** e os exames oficiais **310-035, 320-027** e **310-051**.

Observe que os cursos são cumulativos e você poderá ter 3 certificações *Microsoft* ao concluir os 8 cursos da formação **ACCP JAVA**.

Índice

Apresentação 3

Introdução 4

Nossa Metodologia 5

Objetivo do Curso 6

Introdução ao Java 13

Capítulo 1: A Plataforma *Java* 15
- Comparando *Java* com Outras Linguagens 16
- *Java Virtual Machine* 17
- ***JVM*** *for Devices* 19
- *Garbage Collector* 21
- Acrônimos Importantes *Java* 23
- Edições *Java 2 Platform* 24
  - *Hello World Java* 25
- Portabilidade 27

Capítulo 2:I ntrodução a *Object Oriented* com *Java* 29
- Introdução a *Object Oriented* com *Java* 30
  - Classe 30
  - Objeto 32
  - Método 34
- Ilustrando o Emprego de Classe, Objeto e Método 36
  - Classe 36
  - Objeto 38
  - Atributos 39
  - Métodos 42
  - Métodos de Inicialização de Atributos 42
  - Métodos de Impressão de Atributos 43
  - Resumindo Nosso Exemplo 44

Laboratório 1 - Capítulo 2: Introdução a *Object Oriented* com *Java* 47

Capítulo 3: Aprofundando Conceitos Iniciais 53
- Aprofundando Conceitos Iniciais 54
  - Construtor 54

Construtor *Default* ... 56
Encapsulamento ... 57
Java ***API*** *Documentation* ... 59
Packages ... 61
Imports ... 63
Fully Qualified Name - ***FQN*** ... 65
CLASSPATH ... 67

Laboratório 2 - Capítulo 3: Aprofundando Conceitos Iniciais ... 69

Capítulo 4: Sintaxe *Java 2 Platform* ... 75
Sintaxe *Java 2 Platform* ... 76
Comentários ... 76
Identificadores Válidos ... 78
Keywords Java ... 79
Variáveis Primitivas ... 81
Casting de Primitivas ... 84
Casting de Passagem por Refência ... 88
Passagem por Valor ... 90

Laboratório 3 - Capítulo 4: *Sintaxe Java 2 Platform* ... 93

Capítulo 5:Usando Operadores *Java* ... 99
Usando Operadores *Java* ... 100
Tipos de Operadores ... 102
Operadores Aritiméticos ... 104
Promoções Automáticas ... 108
Operadores Incremento ... 109
OperadoRelacionais ... 113
O Operador == quando usado em Objetos ... 114
O Método *Equals()* ... 114
O Operador *Instanceof* ... 118
Operadores *Boolean* ... 122
Bitwise Operators ... 124
Shift Operator ... 126
Bitwise Shifting Operator ... 128
Ternary Operator ... 131
Short cut Assigment Operators ... 132
Outros Operadores ... 134
Bases Numéricas ... 137
Conversão de Base 2 para Base 16 ... 140

Laboratório 4 - Capítulo 5: Usando Operadores *Java* 141

Capítulo 6: Controle de Fluxo 145
Controle de Fluxo 146

Laboratório 5 - Capítulo 6: Controle de Fluxo 151

Capítulo 7: *Arrays* 157
Arrays 158
Acesso aos Elementos do *Arrays* 160
Arrays Multidimendionais 162

Laboratório 6 - Capítulo 7: *Arrays* 165

Capítulo 8: Programação Avançada 169
Interface 170
Regras para a *Interface* 171
Herança 172
Regras para Herança 173
Generalização 174
Regras para a Generalização 175
Polimorfismo 176
Regra Básica para o Polimorfismo 177
Overloading de Construtores 178
Regra para o *Overloading* de Construtores 178
Overloading de Métodos 180
Regra para o *Overloading* de Métodos 180
Overriding 182
Regras para o *Overriding* 182
Encapsulamento 184
Acesso aos Membros 186
Membros *Static* 188
Membros Final 191
Access Modifiers 193

Laboratório 7 - Capítulo 8: Programação Avançada 195

Capítulo 9: *Java Exceptions* e Operações de I/O 203
Java Exceptions 204
Hierarquia de Classes 208
Java ***I/O*** 211

Laboratório 8 - Capítulo 9: *Java Exceptions* e Operações de I/O **217**

Capítulo 10: *Java Collections Framework* **221**
- *Java Collections Framework* 222
  - As *Collection Interfaces* e Classes 224
  - Hierarquias 225
  - Implementações de *Collection* 226
  - *Collection Interface* 228
  - Métodos Básicos da *Interface Collection* 230
  - Grupo de Elementos 232
  - A *Interface Iterator* 234
  - ListIterator Interface 236
  - *Set interface* 238
  - *List Interface* 241
  - *Map interface* 246
  - Os Métodos de *Map* 247
  - Importantes Informações sobre a *Interface Map* 248

Laboratório 9 - Capítulo 10: *Java Collections Framework* **253**

Capítulo 11: *Threads* **259**
- *Threads* 260
  - Como Criar *Threads* 262
  - Estendendo a Classe *Thread* 263
  - Implementando a *Interface Runnable* 266
  - O Ciclo de Vida de uma *Thread* 269
  - Escalonamento da **JVM** 270
- Escalonamento Preemptivo 270
  - Escalonamento Circular 271
  - Prioridades de *Thread* 272
  - Sincronização de *Threads* 275
  - Agrupamento de *Threads* 282

Laboratório 10 - Capítulo 11: *Threads* **285**

Capítulo 12: Aplicações Gráficas com AWT **293**
- Aplicações Gráficas com **AWT** 294
  - A Classe **AWT** *Compoment* 300
  - *Java Container* 302
  - Gerenciadores de *Layout* 304
  - *Frame* 306

Panel 308
List 310
TextArea 312
TextField 314

Laboratório 11 - Capítulo 12: Aplicações Gráficas com AWT 317

Capítulo 13: Eventos 321
Eventos 322
Classes de Eventos 324
Listeners 325
Tabela de *Interface* e Métodos 327
Classes Adaptadoras 328
Componentes e Eventos Suportados 330
Dialog 333
Menu 336

Laboratório 12 - Capítulo 13: Eventos 339

Gabaritodos dos Laboratórios 345

Gabarito Laboratório 1 - Capítulo 2: Introdução a *Object Oriented* com *Java* 347
Gabarito Laboratório 2 - Capítulo 3: Aprofundando Conceitos Iniciais 355
Gabarito Laboratório 3 - Capítulo 4: Sintaxe *Java 2 Platform* 361
Gabarito Laboratório 4 - Capítulo 5: Usando Operadores *Java* 367
Gabarito Laboratório 5 - Capítulo 6: Controle de Fluxo 371
Gabarito Laboratório 6 - Capítulo 7: *Arrays* 379
Gabarito Laboratório 7 - Capítulo 8: Programação Avançada 385
Gabarito Laboratório 8 - Capítulo 9: *Java Exceptions* e Operadores de **I/O** 399
Gabarito Laboratório 9 - Capítulo 10: *Java Collections Framework* 405
Gabarito Laboratório 10 - Capítulo 11: *Threads* 415
Gabarito Laboratório 11 - Capítulo 12: Aplicações Gráficas com **AWT** 423
Gabarito Laboratório 12 - Capítulo 13: Eventos 429

Anotações

Introdução ao *Java*

- Histórico cronológico de *Java*.
- O *HotJava*.
- As possibilidades de desenvolvimento com *Java*.
- O *Green Team.*
- *The next wave in computing*.

O início do desenvolvimento da plataforma *Java* iniciou-se em 1991. Em 1994 foi lançado o primeiro grande projeto feito totalmente em Java, um *web browser* chamado *HotJava*. Além de um importante marco para a plataforma *Java*, o *HotJava* também foi a primeira grande demonstração do potencial *Java* como uma linguagem de desenvolvimento.

A intenção deste projeto era apresentar o Java como sendo algo moderno e com grande potencialidade para web, mas Java enfrentou um outro obstáculo, pois ficou conhecido na época como uma linguagem que só era utilizada no âmbito da *internet*, e até hoje muitos ainda tem esta concepção errada de *Java*, a realidade é que Java pode ser utilizado para desenvolver qualquer tipo de sistema, seja ele *client-side*, *server-side*, ou *device application*.

Anotações

Em 23 de maio de 1995, John Gage, *director of the Science Office for Sun Microsystems*, e Marc Andreessen, co-fundador e vice presidente executivo da *Netscape*, anunciou em audiência para a *SunWorldTM* que *JavaTM technology* era real, oficial, e seria incorporado ao Netscape *NavigatorTM*. Ainda em 1995 foi lançado o primeiro *release* comercial do Java, a primeira *beta version* disponível foi a *jdk1.0.2a* .

O grupo criador de *Java*, chamado *Green Team*, era formado por 13 pessoas da *Sun*, sua finalidade era antecipar e planejar o plano que eles chamaram de *next wave in computing*. Suas metas eram criar uma convergência de controles digitais para controlar *devices* e *computers*.

Entre os membros e colaboradores do *Green Team* estão:

Al Frazier, Joe Palrang, Mike Sheridan, Ed Frank, Don Jackson, Faye Baxter, Patrick Naughton, Chris Warth, James Gosling, Bob Weisblatt, David Lavallee, and Jon Payne. Missing in action: Cindy Long, Chuck Clanton, Sheueling Chang, and Craig Forrest.

O projeto foi iniciado por Patrick Naughton, Mike Sheridan, e James Gosling.

Anotações

Capítulo 1:

A Plataforma *Java*

Comparando *Java* com Outras Linguagens

	Java	*SmallTalk*	*TCL*	*Perl*	*Shells*	*C*	*C++*
Simples	■	■	■	◧	◧	◧	□
Orientação Objetos	■	■	□	■	□	□	◧
Robusta	■	■	■	■	■	□	□
Segura	■	◧	◧	■	◧	□	□
Interpretada	■	■	■	■	■	□	□
Dinâmica	■	■	■	■	◧	□	□
Portável	■	◧	■	■	◧	◧	◧
Neutra	■	◧	◧	■	◧	□	□
Threads	■	□	□	■	□	□	□
Garbage Collection	■	■	□	□	□	□	□
Exceptions	■	■	□	■	□	□	◧
Performance	Alto	Médio	Baixo	Médio	Baixo	Alto	*High*

■ Característica Existente

◧ Característica Parcialmente Existente

□ Característica não Existe

Anotações

Java Virtual Machine

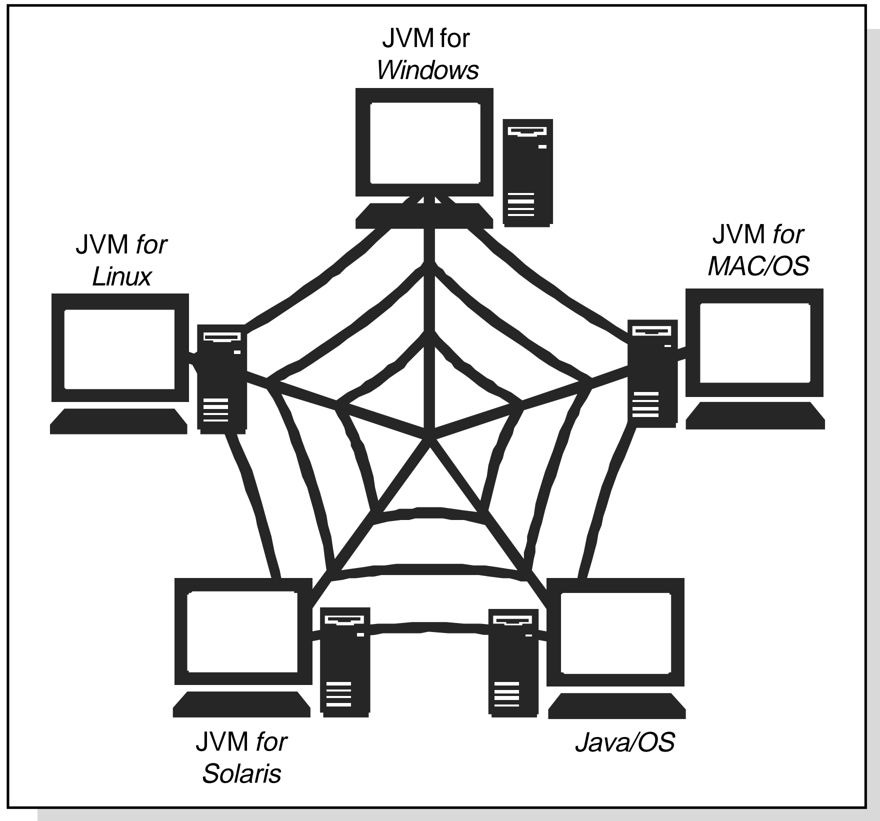
- A *Java Virtual Machine* (**JVM**) é na verdade um emulator para uma máquina real.
- O *byte code*.
- A **JVM** prove todas as especificações para o código pré-compilado *Java* (*byte code*).
- A independência de plataforma.

A *Java Virtual Machine* (**JVM**) é na verdade um emulator para uma máquina real. A **JVM** prove todas as especificações para que o código pré-compilado *Java* (*byte code*) seja independente de plataforma.

A **JVM** é totalmente dependente de plataforma, por exemplo, existe uma **JVM** para *Microsoft Windows*, outra para o *Linux* e, assim por diante. Quando dizemos que *Java* é independente de plataforma, isto não está completamente correto, é válido dizer que os *bytes* code *Java* são independentes de plataforma, mas dependentes de uma **JVM**.

Anotações

O exemplo abaixo ressalta a idéia da **JVM**:



Anotações

JVM *for Devices*

- Aplicações voltadas para dispositivos eletrônicos com *Java*.
- A portabilidade dos *byte codes* nas **JVM´s** dos *Devices*.
- *Java Cards, Java Concept Car, Java Mobile* etc.

Java também foi construído para desenvolver aplicações voltadas para dispositivos eletrônicos, periféricos, cartões, enfim, poderíamos até dizer que existe a possibilidade de construir aplicações *Java* para qualquer 'coisa eletrônica' e, o que torna isto possível é justamente a concepção da **JVM**, uma vez que é somente disto que um *byte code* necessita para ser interpretado.

O exemplo a seguir mostra a aplicação da **JVM** para dispositivos eletrônicos:

Anotações

JVM *for Java Cards*

Credit Card Club

Arnaldo Gonçalves de Sousa
2654 484 654564 56.56.

JVM for *Windows CE*

JVM *for*
Java Concept Car

JVM *for Printer*

Anotações

Garbage Collector

- O *Garbage Collector* se resume em uma *Thread* em baixa prioridade.
- Em *Java* a responsabilidade pelo *de-allocating* fica totalmente a critério da **JVM**.
- Apenas os objetos da memória que não são mais necessários para sua aplicação são coletados.
- Em **C++** é necessário que o programador faça o *de-allocating* dos objetos.
- *Memory overflow*.
- Forçando o *Garbage Collector* com o método *gc()* das classes *System* e *Runtime*.

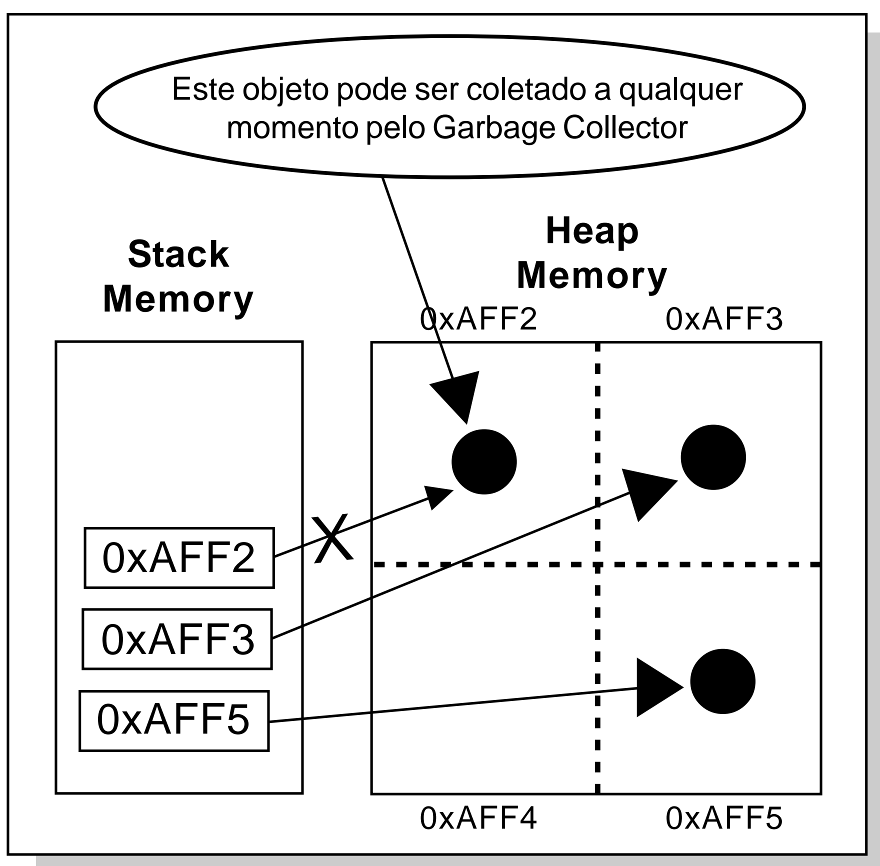
O *Garbage Collector* se resume basicamente em uma *Thread* em baixa prioridade, que por sua vez é lançada para coletar objetos da memória que não são mais necessários para sua aplicação.

Em **C++** é necessário que o programador faça o *de-allocating* dos objetos. Através de códigos escritos em **C++**, o programador retira os objetos que não são mais necessários, antes que haja um 'estouro de memória' ou *memory overflow*, ou seja, o programa fica sem recursos necessários para que o programa possa continuar a sua execução.

Anotações

Em *Java* a responsabilidade pelo *de-allocating* fica totalmente a critério da **JVM**. Embora existam métodos como *gc()* nas classes *System* e *Runtime*, que podem ser chamados com o intuito de coletar objetos, isto funcionará apenas como uma sugestão de coleta de objeto, e não como a coleta concreta do mesmo.

Abaixo um exemplo de funcionamento do *Garbage Collector*:



Anotações

Acrônimos Importantes *Java*

- JRE.
- JDK.
- J2SDK.
- API.
- OOP.
- OO.

JRE – *Java Runtime Environment*.

JDK – *Java Development Kit*.

J2SDK – *Java 2 Platform, Software Developer´S Kit*.

API – *Application Programming Interface*.

OOP – *Object Oriented Programming*.

OO – *Object Oriented*.

Anotações

Edições *Java* 2 *Platform*

- **J2SE** – *Java 2 Platform, Standard Edition*.
- **J2ME** – *Java 2 Platform, Micro Edition*.
- **J2EE** – *Java 2 Platform, Enterprise Edition*.

J2SE – *Java 2 Platform, Standard Edition*.

• Para *Standalone Application* e *Applets*.

J2ME – *Java 2 Platform, Micro Edition*.

• Atende a *mobile devices*.

J2EE – *Java 2 Platform, Enterprise Edition*.

• Atende **B2B** *Application*.

• Atende **B2C** *Application*.

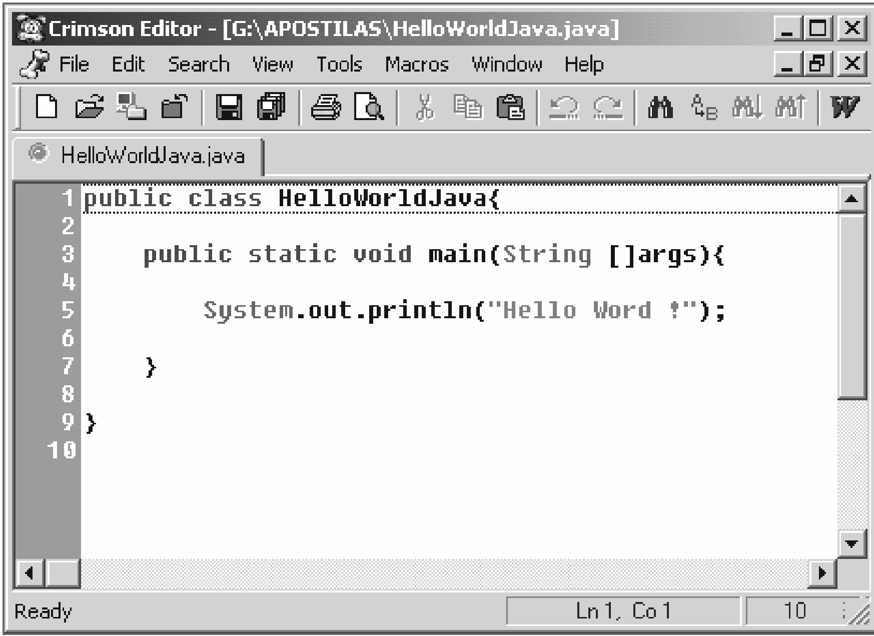
Anotações

Hello World Java

Não poderíamos deixar de aprender uma linguagem de programação sem começar por um *Hello World*, lembre sempre que *Java* é *case-sensitive*.

O editor de códigos usado neste curso é um *freeware*, que pode ser baixado em http://www.crimsoneditor.com/

Exemplo de um simples programa feito em *Java*:

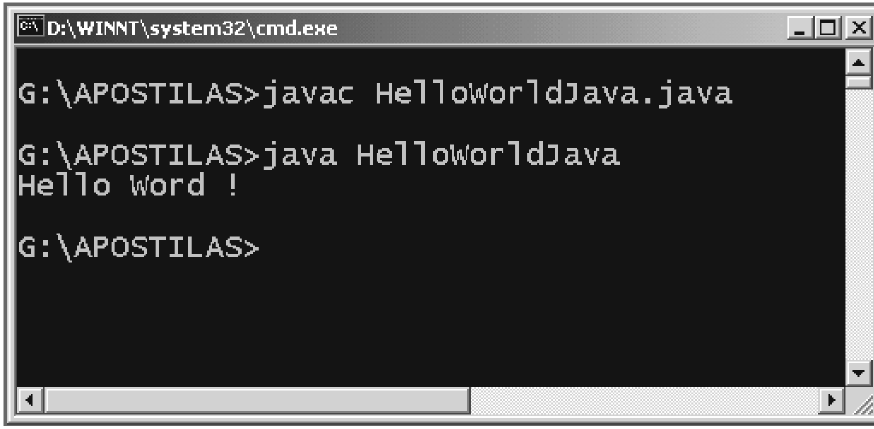


Anotações

Procedimentos para realizar este exemplo:

- O nome do arquivo seria ***HelloWorldJava.java***.
- A compilação seria *javac* ***HelloWorldJava.java***, que resulta em um ***HelloWorldJava.class***, que é nosso *byte code file*.
- E por fim, a execução assim:- ***java HelloWorldJava***.

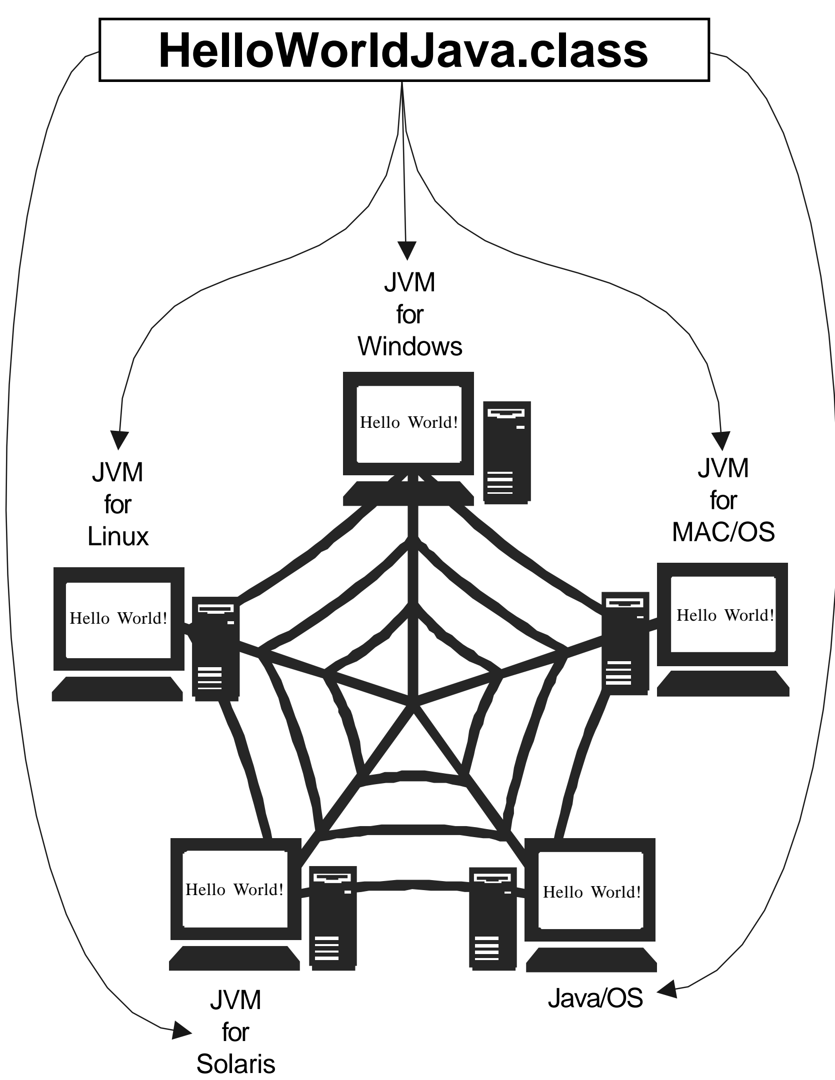
Saída:



Anotações

Portabilidade

Abaixo segue o exemplo para a distribuição do nosso primeiro programa feito em *Java*:



Anotações

Anotações

Capítulo 2:

Introdução a *Object Oriented* com *Java*

Introdução a *Object Oriented* com *Java*

Classe

- Podemos ver uma classe como sendo as especificações de um objeto.
- Dentro da classe teremos todas as operações.
- Classe é realmente uma "fôrma de bolo" para um objeto.

Podemos pensar em uma classe como se fosse uma fôrma para um bolo, assim como a fôrma para bolos, o molde da forma irá ditar o formato do bolo, e este último poderíamos associar a objetos.

Todas as especificações, comportamento e atributos dos objetos estão no design da classe. Portanto é correto afirmar que a classe é uma peça fundamental em uma programação orientada a objetos, pois você irá empregar na classe os mecanismos de operações que a mesma pode executar através de métodos a cada solicitação feita pelos objetos.

Anotações

Itens que você deve sempre recordar:

- Uma classe é uma definição de como irão ser os objetos. Podemos ver uma classe como sendo as especificações de um objeto.
- Dentro da classe teremos todas as operações, ou seja, todos os comportamentos que um objeto poderá ter.
- Como podemos observar, uma classe é realmente a fôrma para um objeto e, este por sua vez deverá obedecer às regras definidas na classe.

Anotações

Objeto

- O paradigma de programação orientado a objetos.
- Objeto, o centro das atenções
- É possível definir um objeto como algo que tenha atributos que serão manipulados.
- Objetos podem ser:
  - Uma caneta, suas propriedades seriam cor da tinta, peso e dimensões.
  - Um funcionário, com as propriedades nome, idade, endereço etc.

Dentro do paradigma de programação orientado a objetos, temos o objeto como o centro das atenções. É possível definir um objeto como algo que tenha atributos que serão manipulados.

Objetos podem ser:

- Uma caneta, suas propriedades seriam cor da tinta, peso e dimensões.
- Um funcionário, com as propriedades nome, idade, endereço etc.

Anotações

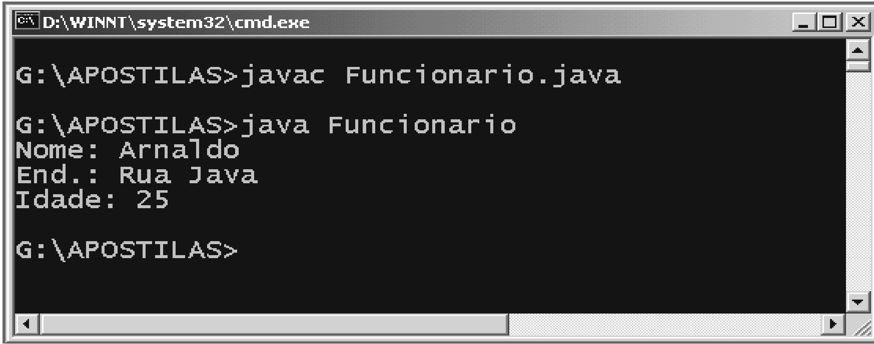
Código de exemplo de objeto:

Crimson Editor - [G:\APOSTILAS\Funcionario.java]

```java
public class Funcionario{
    String nome;
    String endereco;
    int idade;

    public static void main(String []args){

        Funcionario func = new Funcionario();

        func.nome = "Arnaldo";
        func.endereco = "Rua Java";
        func.idade = 25;

        System.out.println("Nome: " + func.nome);
        System.out.println("End.: " + func.nome);
        System.out.println("Idade: " + func.nome);

    }

}
```

Ready | Ln 1, Co 1 | 20

Saída para o exemplo anterior:



Anotações

Método

- No paradigma procedural tínhamos funções, no **OO** temos métodos.
- Podemos afirmar que o método é o equivalente a uma função em uma linguagem procedural, por exemplo, a linguagem de programação Pascal .
- Os métodos servirão para manipular os atributos dos objetos.
- É através dos métodos que os objetos se comunicam.

São considerados métodos, as funções que serão aplicadas em um ou mais objetos, ou seja, os métodos servirão para manipular os atributos dos objetos, o método é o equivalente a uma função em uma linguagem de programação procedural.

É através dos métodos que os objetos se comunicam e, utilizam a troca de mensagens para isto, que se dá com a chamada de métodos.

Anotações

Exemplo do uso de método:

```java
public class Funcionario{
    String nome;
    String endereco;
    int idade;

    public static void main(String []args){

        Funcionario func = new Funcionario();

        func.nome = "Arnaldo";
        func.endereco = "Rua Java";
        func.idade = 25;

        func.imprimir();
    }

    public void imprimir(){
        System.out.println("Nome: " + nome);
        System.out.println("End.: " + endereco);
        System.out.println("Idade: " + idade);
    }
}
```

Anotações

Ilustrando o Emprego de Classe, Objeto e Método

Classe

- Classe
  - Definimos a classe comemoração. Nesta classe ficarão métodos, atributos e objetos.

Neste exemplo, fugiremos um pouco dos tradicionais exemplos que utilizam uma classe Pessoa para explicar o significado de classe, objeto e métodos, pois, num mundo real iremos nos deparar com inúmeras situações onde nem sempre tudo é completamente concreto. Faremos então uma análise em uma classe com um nível de abstração maior que a classe Pessoa, utilizando para isto uma classe denominada Comemoração.

Anotações

Observe a seqüência de exemplos:

É definido então a classe comemoração. É na classe que ficam métodos, atributos e objetos:

Classe: Comemoração

Anotações

Objeto

- Objeto
  - Uma instância da classe comemoração poderia ser usada para representar o natal, já que o natal é uma comemoração.

Uma instância da classe comemoração poderia ser usada para representar o natal, já que o natal é uma comemoração.

Objeto: Natal (é uma comemoração)

Anotações

Atributos

- Atributos
  - Características dos objetos.
  - Particularidades dos objetos.

Uma classe pode ter inúmeros atributos, estes atributos descrevem as características de cada instância (objeto).

Cada palavra identificadora que possa atribuir ao objeto uma particularidade é um atributo do mesmo.

Anotações

O primeiro atributo a ser preenchido é o símbolo da comemoração:

Atributo 1 - SÍMBOLO: Árvore de Natal.

O segundo atributo a ser preenchido é o personagem da comemoração, como se fosse uma mascote:

Atributo 2 - PERSONAGEM: Papai Noel

Anotações

O terceiro, e mais importante atributo é o significado da comemoração:

Atributo 3 - SIGNIFICADO: Nascimento de Jesus Cristo de Nazaré, filho de Deus segundo o cristianismo.

Anotações

Métodos

Agora é o momento de estabelecer onde ficaria o método nesta nossa história. Partindo-se do princípio básico de que o método é o responsável por manipular os atributos:

Métodos de Inicialização de Atributos

- inserirAtributoSimbolo (“Árvore de Natal”).

- inserirAtributoPersonagem (“Papai Noel”).

- inserirAtributoSignificado (“Nascimento de Jesus”).

Anotações

Métodos de Impressão de Atributos

- imprimeAtributoSimbolo ( ) >>> imprime "Árvore de Natal'.
- imprimeAtributoPersonagem ("Papai Noel") >>> imprime "Papai Noel".
- imprimeAtributoSignificado ("Nascimento de Jesus") >>> imprime "Nascimento de Jesus Cristo".

Anotações

Resumindo Nosso Exemplo

- Resumo:
  - Classe: Comemoração.
  - Atributo 1 – SÍMBOLO: “Árvore de Natal”.
  - Atributo 2 – PERSONAGEM: “Papai Noel”.
  - Atributo 3 – SIGNIFICADO: “Nascimento de Jesus Cristo”.
  - Métodos de inicialização de atributos.
  - Métodos de impressão de atributos.

Fazendo a contabilidade de nossa classe, podemos contar:

- 1 classe.
- 3 atributos.
- 6 métodos.

Porém deve-se ficar muito claro que uma classe pode ter inúmeros atributos e métodos.

Anotações

Resumindo nosso exemplo teremos:

- Classe: Comemoração.
- Atributo 1 – SÍMBOLO: “Árvore de Natal”.
- Atributo 2 – PERSONAGEM: “Papai Noel”.
- Atributo 3 – SIGNIFICADO: “Nascimento de Jesus Cristo”.
- Métodos de inicialização de atributos.
- Métodos de impressão de atributos.

Anotações

Anotações

Laboratório 1:

Capítulo 2: Introdução a *Object Oriented* com *Java*

Concluir o(s) exercício(s) proposto(s) pelo instrutor. O instrutor lhe apresentará as instruções para a conclusão do mesmo.

Laboratório 1 - Capítulo 2

1) Compile e rode o exemplo da página 33 (código de exemplo de objeto).

Anotações

2) Compile e rode o exemplo da página 35 (exemplo do uso de método).

Anotações

3) Implemente o código para a análise feita durante o módulo, contendo os itens abaixo:

Classe: Comemoracao.

Atributo 1 - SÍMBOLO: “Árvore de Natal”.

Atributo 2 - PERSONAGEM: “Papai Noel”.

Atributo 3 - SIGNIFICADO: “Nascimento de Jesus Cristo”.

Métodos de inicialização de atributos.

Métodos de impressão de atributos.

Anotações

4) Repita o exercício anterior, só que deste teste usando características de comemoração da páscoa, para analisar se este código também pode ser usado para outras comemorações.

Anotações

Anotações

Capítulo 3:

Aprofundando Conceitos Iniciais

Aprofundando Conceitos Iniciais

Construtor

- Inicializador de atributos.
- O Método e o construtor.
- Não retorna valor.

O construtor funciona como um Inicializador de atributos.

Embora possamos colocar em um construtor quase tudo o que podemos colocar em um método, não é correto dizer que o construtor também é um método, já que existe uma diferença crucial entre este dois elementos da programação orientada a objetos, o método retorna um valor e o construtor nunca retorna, e nem mesmo declara um tipo de retorno.

Anotações

Exemplo: Construtor

Crimson Editor - [G:\APOSTILAS\Construtor.java]

```
public class Construtor{
    String nome;
    String endereco;
    int idade;

    public Construtor(String a, String b, int c){
        nome = a;
        endereco = b;
        idade = c;
    }

    public static void main(String []args){

        Construtor func;
        func = new Construtor("Arnaldo", "Rua java", 25);

        func.imprimir();
    }

    public void imprimir(){
        System.out.println("Nome: " + nome);
        System.out.println("End.: " + endereco);
        System.out.println("Idade: " + idade);
    }
}
```

Anotações

Construtor *Default*

- O construtor que existe implicitamente.
- Provido pelo próprio *Java*.
- Caso você faça um construtor, o construtor *default* não existirá.

O construtor *default* é o construtor que existe implicitamente dentro de uma classe, ou seja, caso você não defina nenhum construtor dentro de uma classe, então *Java* irá prover um para você.

Caso o exemplo anterior não tivesse definido nenhum construtor, *Java* iria definir o seguinte construtor automaticamente:

```
public Construtor (){
      super (); // Esta keyword será estudada posteriormente.
}
```

Anotações

Encapsulamento

- Protegendo uma ou mais informações.
- Encapsulamento de atributos.
- Encapsulamento de métodos.
- Encapsulamento de construtores.
- Encapsulamento de classes internas.

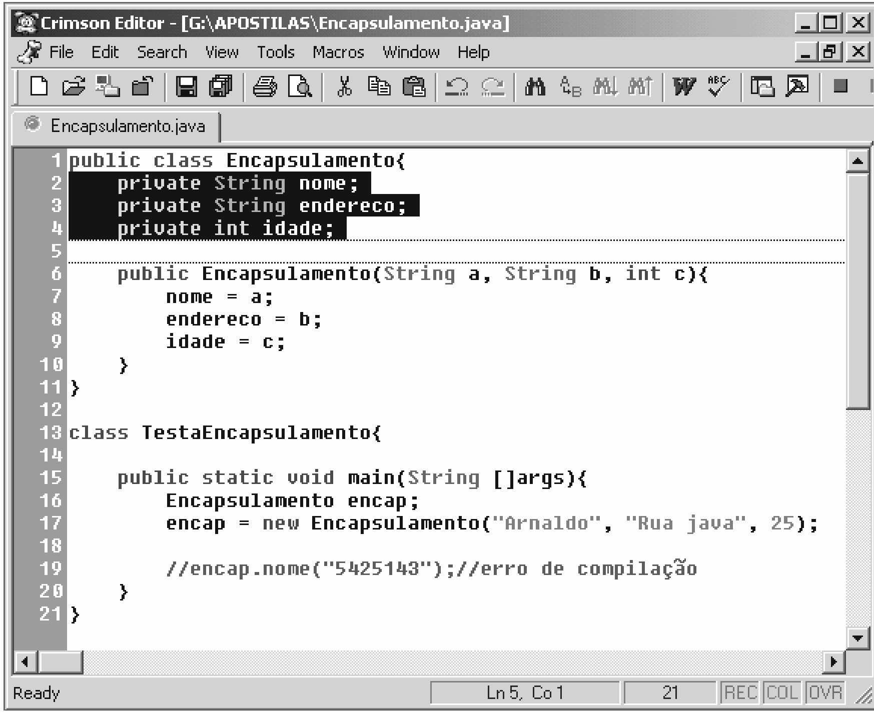
Quando dizemos que estamos Encapsulando um determinado atributo de um objeto, na verdade estamos protegendo uma ou mais informações que existem dentro de uma classe porque não queremos compartilhá-la.

Um bom exemplo de aplicação de encapsulamento seria sobre atributos, pois assim forçaria seu acesso através de métodos ou construtores, pois desta forma conseguiríamos avaliar um determinado valor antes de atribuí-lo.

Também é possível aplicar encapsulamento a construtores, métodos e classes internas.

Anotações

Abaixo encontraremos um exemplo de encapsulamento de atributos:



Anotações

Java API *Documentation*

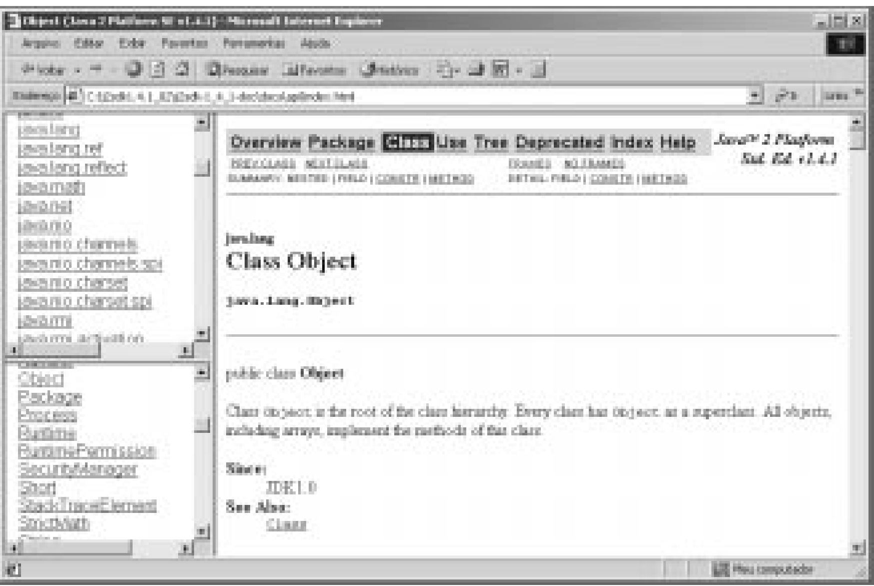
- Ferramentas.
- Recursos.
- Bibliotecas.
- *Links* importantes.
- Versão *on-line*.

Iremos encontrar dentro da **API** todas as informações sobre ferramentas, recursos, bibliotecas, *links* importantes etc, disponíveis para uma determinada edição *Java*.

Anotações

Esta documentação está disponível no *site* da *Sun Microsystems* para *download*, além de uma versão *on-line* para consulta.

API - *Application Programming Interface*



Anotações

Packages

- Organiza classes.
- Agrupamento lógico e físico das classes.
- Otimizar o *deployment*.

Anotações

Os pacotes servem para organizar classes através de um agrupamento lógico e físico, além de melhorar o *desing* de uma aplicação e otimizar o *deployment* da mesma.

Vamos observar abaixo:

Crimson Editor - [G:\APOSTILAS\RH.java]

```
package empresa.departamento;

public class RH{

    private String contratado;

    public RH (String a){
        contratado = a;
    }
}
```

Quando definimos o pacote acima como "empresa.departamento", devemos obrigatoriamente colocar esta classe dentro do diretório empresa\departamento.

Por exemplo:

`C:\empresa\departamento\RH.class`

Anotações

Imports

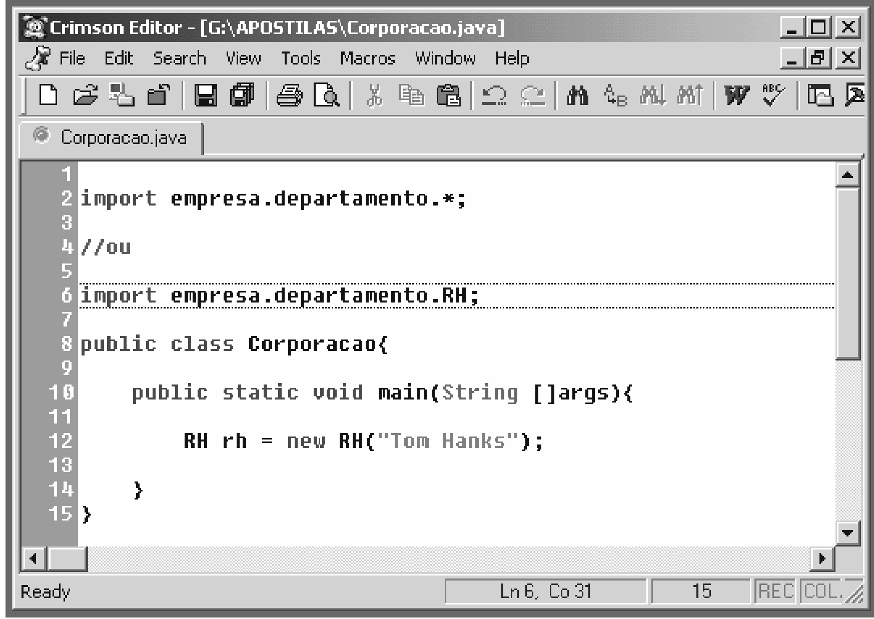
- Importação de uma Classe.
- Importação de todo o pacote de classes.
- Importação versus Carregamento de classes.

O import serve para importarmos uma determinada classe dentro de um *package*, ou para importarmos todas as classes dentro de um pacote.

É importante dizer que não importa quantas classes nos importamos, somente serão carregadas as que utilizarmos, em outras palavras, só serão carregadas aquelas que forem instanciadas.

Anotações

Exemplo: importando pacotes necessários.



Anotações

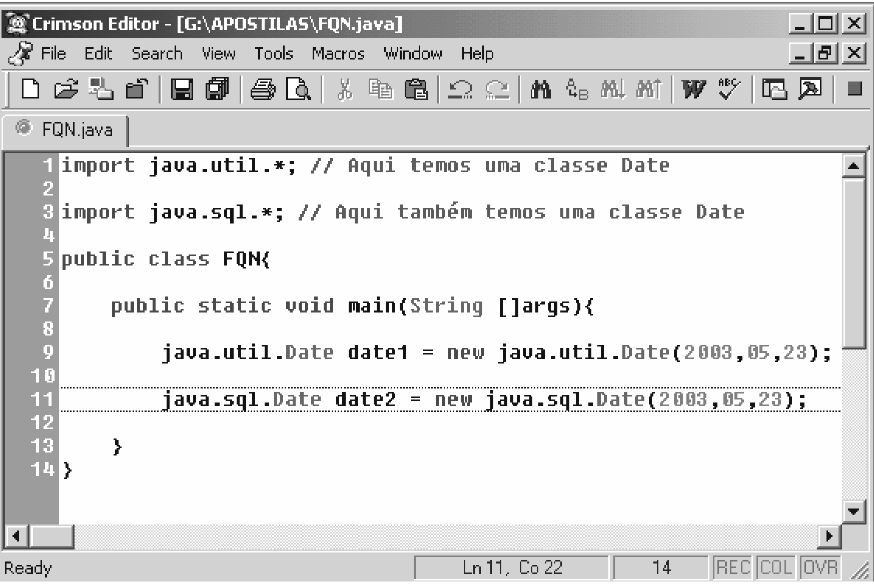
Fully Qualified Name - FQN

- Usando **FQN** não necessitamos fazer *imports*.
- Usado para abolir ambigüidade.

Usando o nome totalmente qualificado, não necessitamos fazer *imports*, porém esta forma é mais cansativa. A grande vantagem de se usar **FQN** é abolir ambigüidade.

Anotações

Veja o exemplo:



Anotações

CLASSPATH

■ O **CLASSPATH** é uma variável de ambiente que define qual será o ponto de início de busca por pacotes ou classes.

Exemplo:

SET CLASSPATH=.; C:\

■ A definição acima indica que a procura por pacotes ou classes deve ser a partir do diretório atual (.), ou a partir do diretório raiz (C:\ ).

O **CLASSPATH** é uma variável de ambiente que define qual será o ponto de início de busca por pacotes ou classes.

Exemplo:

SET CLASSPATH=.; C:

A definição acima indica que a procura por pacotes ou classes deve ser a partir do diretório atual (**.**), ou a partir do diretório raiz (**C:**\).

Anotações

Anotações

Laboratório 2:

Capítulo 3: Aprofundando Conceitos Iniciais

Concluir o(s) exercício(s) proposto(s) pelo instrutor. O instrutor lhe apresentará as instruções para a conclusão do mesmo.

Laboratório 2 - Capítulo 3:

1) Implemente um construtor e o conceito de encapsulamento na questão 2 do capítulo anterior.

Anotações

2) Compile e rode o exemplo de pacotes da página 62.

Anotações

3) Compile e rode o exemplo de *import* da página 64.

Anotações

4) Compile e rode o exemplo de **FQN** do capítulo 3, da página 66. Após fazer isso retire os **FQN's** (linhas 9 e 11) e, compile o código novamente para analisar o resultado desta alteração.

Anotações

Anotações

Capítulo 4:

Sintaxe Java 2 Platform

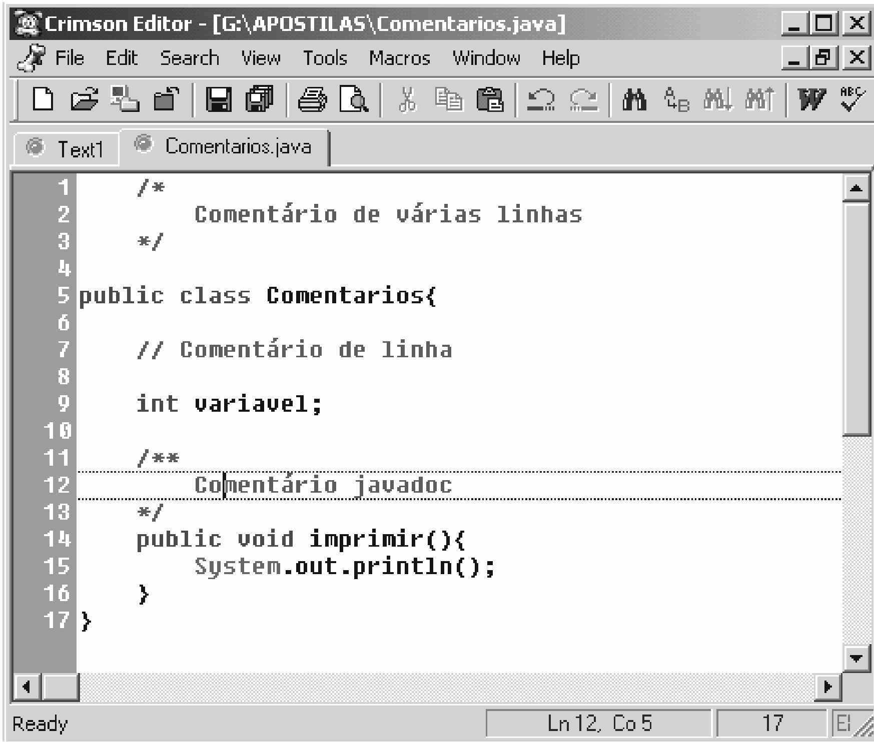
Sintaxe Java 2 Platform

Comentários

- Podem estar em qualquer parte do código.
- // Comentário de linha.
- /* Comentário de várias linhas */.
- /** Comentário de documentação *javadoc*.

Anotações

Exemplo:



Anotações

Identificadores Válidos

- Os nomes dados das variáveis, métodos, objetos etc, são denominados identificadores.
- Existem regras para a aplicação dos mesmos.

Os identificadores em *Java* devem seguir as seguintes regras:

1	A primeira posição deverá ser com uma letra, (_) ou (**$**).
2	Ser formado por caracteres **UNICODE**.
3	Não ser uma palavra reservada: *goto e const*.
4	Não ser igual a literal: *null*.
5	Não ser repetido dentro do seu *escopo*.
6	Não podem ser as *keywords* ou as *Boolean Literais*: *true e false*.
7	Não pode ser uma *keword Java*: *int, byte, class* etc.

Observações:

- O item 4 está de acordo com *Java Language Spefication Section 3.10.7.*
- O item 6 está de acordo com *Java Language Spefication Section 3.10.3.*

Anotações

Keywords Java

- A cada nova versão do *Java*, novas *keywords* são adicionadas ao **JDK**.
- Não podem ser utilizadas como identificadores.
- As *keywords goto* e *const*, embora sejam *keywords Java*, não possuem implementação em *Java*.

As *keywords Java* existem com múltiplas finalidades, que serão estudadas ao longo do curso. A cada nova versão do *Java*, novas *keywords* são adicionadas ao **JDK**.

Uma das características mais importantes das *keywords Java* é o fato de que as mesmas não podem ser utilizadas como identificadores.

As *keywords goto* e *const*, embora sejam *keywords Java*, não possuem nenhuma implementação em *Java*. Em **C++**, de onde *goto* e *const* foram "herdadas", o uso é normal.

Anotações

Tabela de Identificadores:

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

Anotações

Variáveis Primitivas

- Tipos integrais.
- Ponto flutuante.
- Tipo caractere.
- *Escape sequences*.
- Tipo lógico.

Abaixo serão apresentadas as variáveis comuns, ou seja, não são objetos.

Tipos Integrais: constituído por tipos numéricos inteiros.

Inicialização Automática: 0 (zero).

Tipo	Tamanho em *bits*	Faixa
byte	8	-128 até +127
short	16	-32,768 até +32,767
int	32	-2,147,483,648 até+2,147,483,647
long	64	-9,223,372,036,854,775,808 até +9,223,372,036,854,775,807

Anotações

Exemplos:

byte b = 10; // Inicialização Automática: 0 (zero).

short s = 5; // Inicialização Automática: 0 (zero).

int i = 80; // Inicialização Automática: 0 (zero).

// Inicialização Automática: 0L (zero+L).
long l = 800l; ou 800L (não é case-sensitive)

Ponto Flutuante: constituído por tipos numéricos reais.

Tipo	Tamanho em *bits*	Faixa
float	32	-3.40292347E+38 até +3.40292347E+38
double	64	-1.79769313486231570E+308 até +1.79769313486231570E+308

Inicialização Automática:

float - 0.0F (zero + F).

double - 0.0D (zero + D).

Exemplos:

float f = 10.44F; ou 10.44f (não é *case-sensitive*).

double d = 1.24; ou 1.24D ou 1.24d (não é *case-sensitive*).

Tipo Caractere:

Tipo	Tamanho em *bits*	Faixa
char	16	**UNICODE** - 65535 caracteres possíveis

Anotações

Escape Sequences:

'\u0000' a '\uFFFF'	Caracteres *Unicode*
'\b'	retrocesso
'\t'	tab
'\n'	avanço de linha
'\f'	form feed
'\r'	retorno de carro
'\" '	double quote
'\' '	single quote
'\\'	barra invertida

Inicialização Automática:

char – '\u0000'

Exemplo:

char c = 'a' ou 'A';

char c = '\n';

Tipo Lógico:

Tipo	**Faixa**
boolean	*true ou false*

Inicialização Automática:

boolean - false

Exemplo:

boolean b = false;

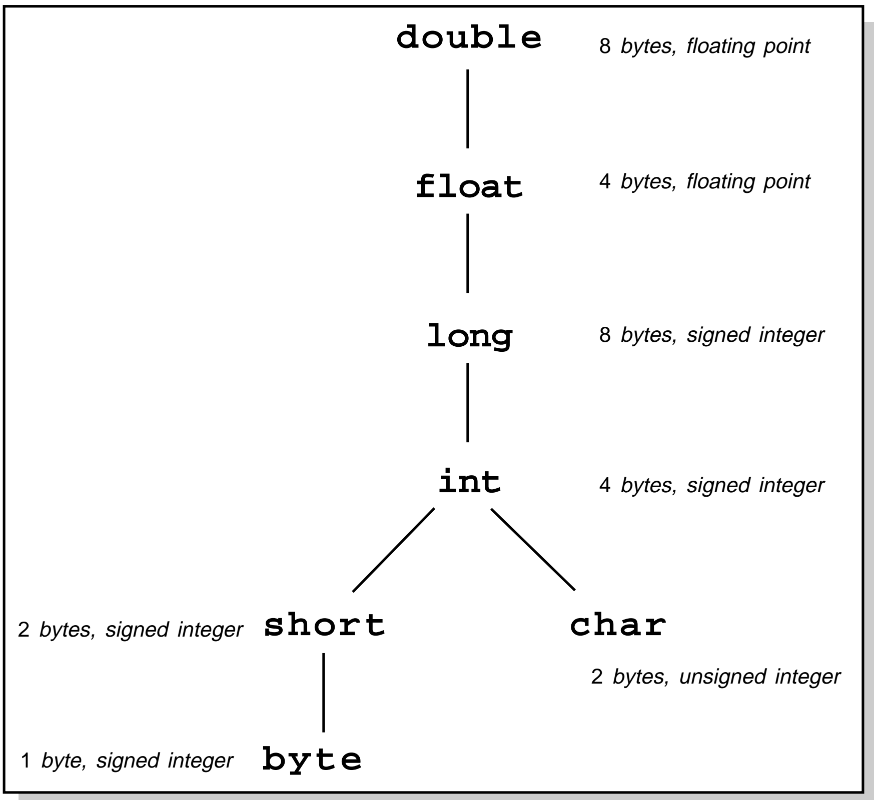
boolean b = true;

Anotações

Casting de Primitivas

- *Casting* implícito.
- *Casting* explícito.
- Promoções automáticas.

Anotações



Na figura anterior, considere *casting* automático entre os tipo que vão de baixo para cima, ou seja do *byte* ao double e do *char* ao *double*.

Quando necessitamos atribuir o valor de uma variável a uma outra e, elas não são compatíveis, como em tamanho (*bits*), por exemplo, então necessitamos fazer um *casting*. Abaixo segue um detalhamento sobre *casting* de variáveis.

Anotações

long → *int*:

discarta quatro *bytes*:

- *range* -2147483648 à 2147483647 são preservados.

int → *short*:

discarta dois *bytes*:

- *range* -32768 à 32767 são preservados.

short → *byte*:

discarta um *byte*:

- *range* -128 à 127 são preservados.

int → *char*:

discarta dois *bytes*:

- *range* 0 à 65535 são preservados.

char → *short* e *short* → *char*:

não precisa fazer nada, porém:

- somente números no *range* 0 à 32767 são preservados.

byte → *char*:

discarta dois *bytes*:

- somente números no *range* de 0 à 127 são preservados.

double → *float*:

O resultado é calculado por aproximação.

- números muito maior que sua capacidade irão ser mapeados para **POSITIVE_INFINITY** ou **NEGATIVE_INFINITY**.

double or *float* → *any integer number type*:

- A parte fractional é discartada.

Anotações

Exemplo: ***Casting***

// Operação válida

long *bigval* = 6;

// Operação inválida porque são de tipos diferentes.
int smallval = 99L;

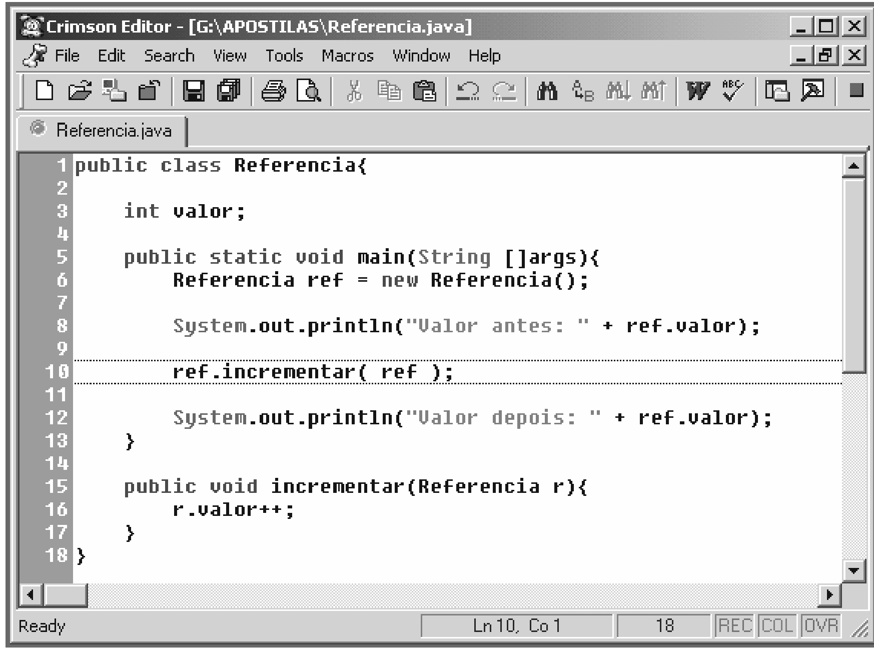
float z = 12.414F; // Operação válida

// Operação inválida porque está tentando atribuir um valor *double*.

float zp = 12.414;

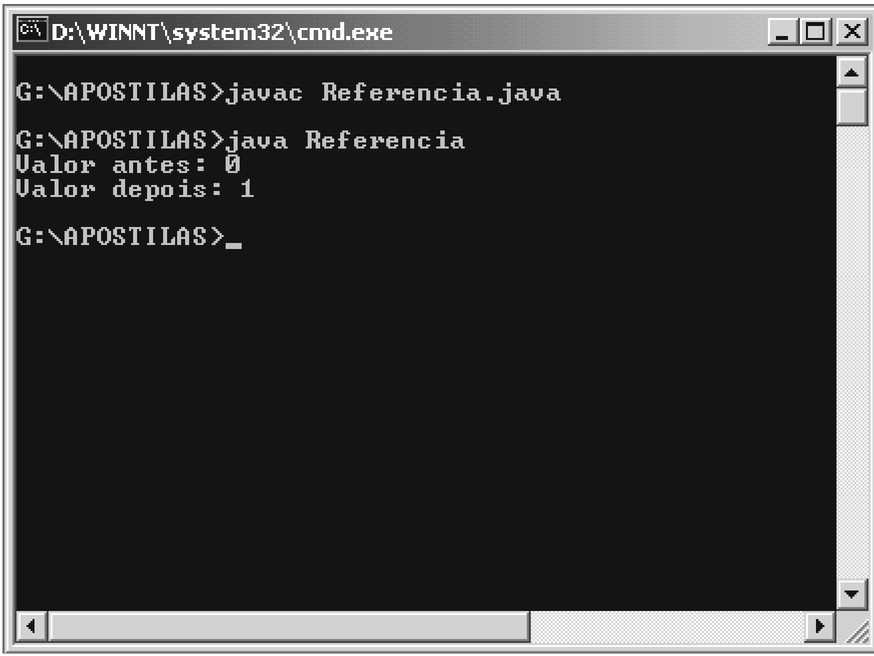
Anotações

Passagem por Referência



Anotações

Saída:



Anotações

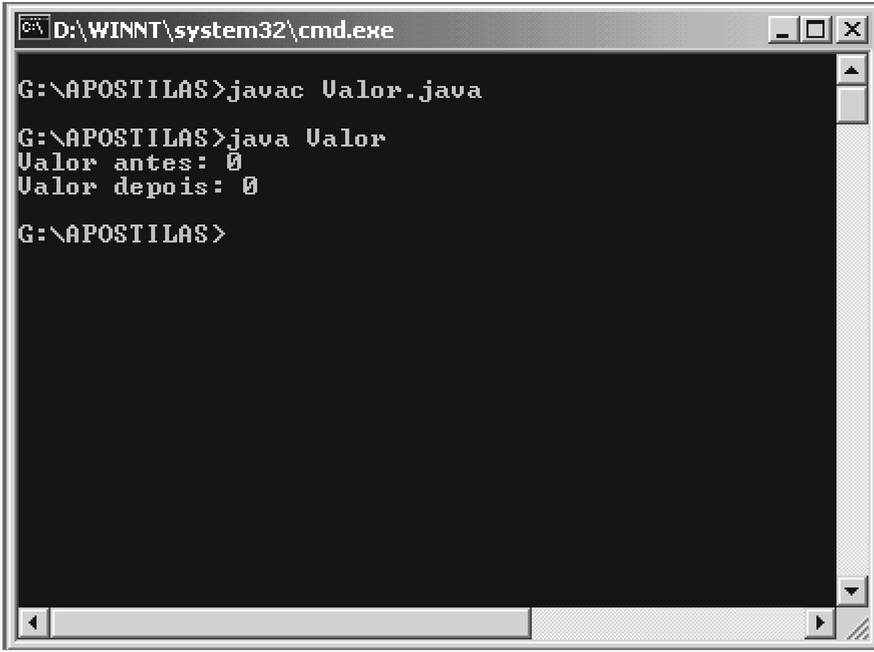
Passagem por Valor

Crimson Editor - [G:\APOSTILAS\Valor.java]

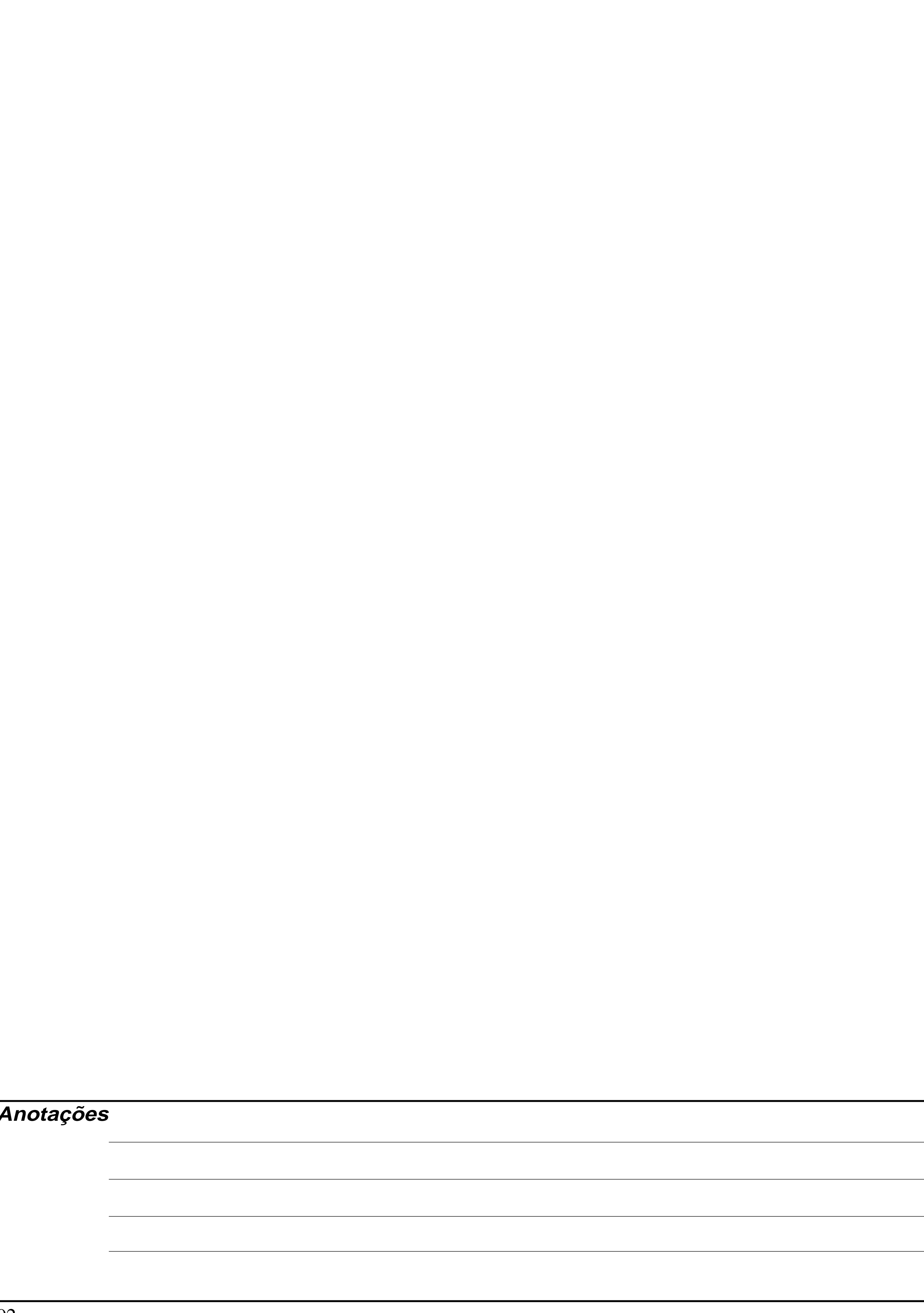
```
public class Valor{

    int valor;

    public static void main(String []args){
        Valor ref = new Valor();

        System.out.println("Valor antes: " + ref.valor);

        ref.incrementar( ref );

        System.out.println("Valor depois: " + ref.valor);
    }

    public void incrementar(Valor r){
        int valorLocal = r.valor;
        valorLocal++;
    }
}
```

Anotações

Saída:



Anotações



Anotações

Laboratório 3:

Capítulo 4: *Sintaxe Java 2 Platform*

Concluir o(s) exercício(s) proposto(s) pelo instrutor. O instrutor lhe apresentará as instruções para a conclusão do mesmo.

Laboratório 3 - Capítulo 4:

1) Teste os identificadores conforme a tabela que foi apresentada na página , a seguir segue uma cópia da mesma:

1	A primeira posição deverá ser com uma letra, "_" ou "$".
2	Ser formado por caracteres **UNICODE**.
3	Não ser uma palavra reservada: ***goto*** *e* ***const.***
4	Não ser igual a literal: ***null.***
5	Não ser repetido dentro do seu *escopo.*
6	Não podem ser as *keywords* ou as *Boolean Literais*: ***true*** *e* ***false***.
7	Não pode ser uma *keword Java: int*, *byte*, *class* etc.

Anotações

2) Teste os tipos de variáveis estudados neste capítulo, são elas:

- Tipos Integrais:
- Ponto Flutuante:
- Tipo Caractere:
- Dígrafos:
- Tipo Lógico:

Anotações

3) Compile e rode o exemplo de Passagem por Referência da página 88.

Anotações

4) Compile e rode o exemplo de Passagem por Valor da página 90.

Anotações

Anotações

Capítulo 5:

Usando Operadores *Java*

Usando Operadores *Java*

- Característica de funcionamento muito parecida a C/C++.
- *Instanceof* não existe em C++.
- Incremento e decremento são de comportamento idênticos aos de C/C++.

Neste capítulo serão apresentados os operadores *Java*. Muitos destes operadores tem sua característica de funcionamento muito parecida a **C/C++**.

Um grande destaque para este capítulo de operadores é exatamente o operador *instanceof* que não existe em **C++**, ou muito menos em **C**, que não é orientado a objetos.

O operador *instaceof* serve exclusivamente para testar tipos de objetos, e é um operador muito eficiente. Existe a possibilidade de uso deste operador até mesmo em objetos que estão em polimorfismo.

Anotações

Já os operadores extremamente versáteis e importantes como os de incremento e decremento são de comportamento idênticos aos de **C/C++**.

Outro grande destaque entre os operadores é o ternary, que permite uma excelente e elegante forma de inicializar variáveis e objetos.

Anotações

Tipos de Operadores

- Operator
  - *unary*
  - *binary*
  - *ternary*
- Notação
  - *prefix*
  - *postfix*
  - *infix*

Um operador pode executar uma função em um, dois ou três operandos. Um operador que requer apenas um operando, é chamado de *unary* operator. Por exemplo, ++ é um *unary* operator que incrementa um valor de um operando em 1. Um operador que requer dois operandos é chamado de *binary* operator. Por exemplo, = é um *binary* operator, que atribui o valor que está no lado direito para o operando que está no lado esquerdo. E por último, o *ternary* operator que requer três operandos. Este último operador, é um modo curto do *if-else*.

O *unary* operator suporta qualquer notação de prefixo ou sufixo. A notação do prefixo significa que o operador aparece antes de seu operando:

Anotações

Na notação *prefix*, o operador antecede o operando:

op x

A notação *postfix* significa que o operador aparece após seu operando:

op x

Todos os operadores binários usam a notação de *infix*, que significa que o operador aparece entre seus operandos:

op1 x op2

O operador ternário é também *infix*; cada componente do operador aparece entre operandos:

op1 ? op2 : op3

Além a executar a operação, um operador retorna um valor. O valor do retorno e seu tipo dependem do operador e do tipo de seus operandos. Por exemplo, os operadores aritméticos, que executam operações aritméticas básicas tais como a adição e a subtração, retornam números – resultantes da operação aritmética. O tipo de dados retornado por um operador aritmético depende do tipo de seus operandos.

Anotações

Operadores Aritiméticos

- Adição.
- Subtração.
- Multiplicação.
- Divisão.
- Módulo.

Java suporta vários operadores aritméticos para todos os números *floating-point* e do inteiro. Estes operadores são + (adição), **-** (subtração), ***** (multiplicação), **/** (divisão), e **%** (módulo). A tabela seguinte sumaria as operações aritméticas binárias na língua de programação de *Java*.

Operador	Uso	Descrição
+	op1 + op2	Adiciona op1 e op2
-	op1 - op2	Subtrai op2 de op1
*	op1 * op2	Multiplica op1 por op2
/	op1 / op2	Divide op1 por op2
%	op1 % op2	Resto de op1 / op2

Anotações

No exemplo, *ArithmeticTest*, que definimos dois inteiros, dois números *double* e usamos os cinco operadores aritméticos executar operações aritméticas diferentes. Este programa usa também + para concatenar *Strings*.

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

      int i = 30;
      int j = 22;
      double x = 19.55;
      double y = 6.44;
      System.out.println("Valores de variaveis...");
      System.out.println("   i = " + i);
      System.out.println("   j = " + j);
      System.out.println("   x = " + x);
      System.out.println("   y = " + y);

      //Adição
      System.out.println("Adicao...");
      System.out.println("   i + j = " + (i + j));
      System.out.println("   x + y = " + (x + y));

      //Subtração
      System.out.println("Subtracao...");
      System.out.println("   i - j = " + (i - j));
      System.out.println("   x - y = " + (x - y));

      //Multiplicação
      System.out.println("Multiplicacao...");
      System.out.println("   i * j = " + (i * j));
      System.out.println("   x * y = " + (x * y));
```

Anotações

```
        //Divisão
        System.out.println("Divisao...");
        System.out.println("   i / j = " + (i / j));
        System.out.println("   x / y = " + (x / y));

        //Resto da divisão
        System.out.println("Resto da divisao...");
        System.out.println("   i % j = " + (i % j));
        System.out.println("   x % y = " + (x % y));

        //Tipos mistos
        System.out.println("Tipos mistos...");
        System.out.println("   j + y = " + (j + y));
        System.out.println("   i * x = " + (i * x));
    }
}
```

Saída:

```
Valores de variaveis...
    i = 30
    j = 22
    x = 19.55
    y = 6.44
Adicao...
    i + j = 52
    x + y = 25.990000000000002
Subtracao...
    i - j = 8
    x - y = 13.11
Multiplicacao...
    i * j = 660
    x * y = 125.90200000000002
```

Anotações

```
Divisao...
    i / j = 1
    x / y = 3.0357142857142856
Resto da divisao...
    i % j = 8
    x % y = 0.22999999999999954
Tipos mistos...
    j + y = 28.44
    i * x = 586.5
```

Anotações

Promoções Automáticas

- Os operadores + ou – em variáveis *byte*, *short* ou *char*.
- Exemplo:

```
byte b = 10;
b = -b; // erro de compilação!
int i = -b; // compilação ok
```

Caso sejam aplicados os operadores + ou – em variáveis *byte*, *short* ou *char*, irá ocorrer uma promoção do valor resultante para *int*.

Exemplo:

```
byte b = 10;
b = -b; // erro de compilação!

int i = -b; // compilação ok
```

Anotações

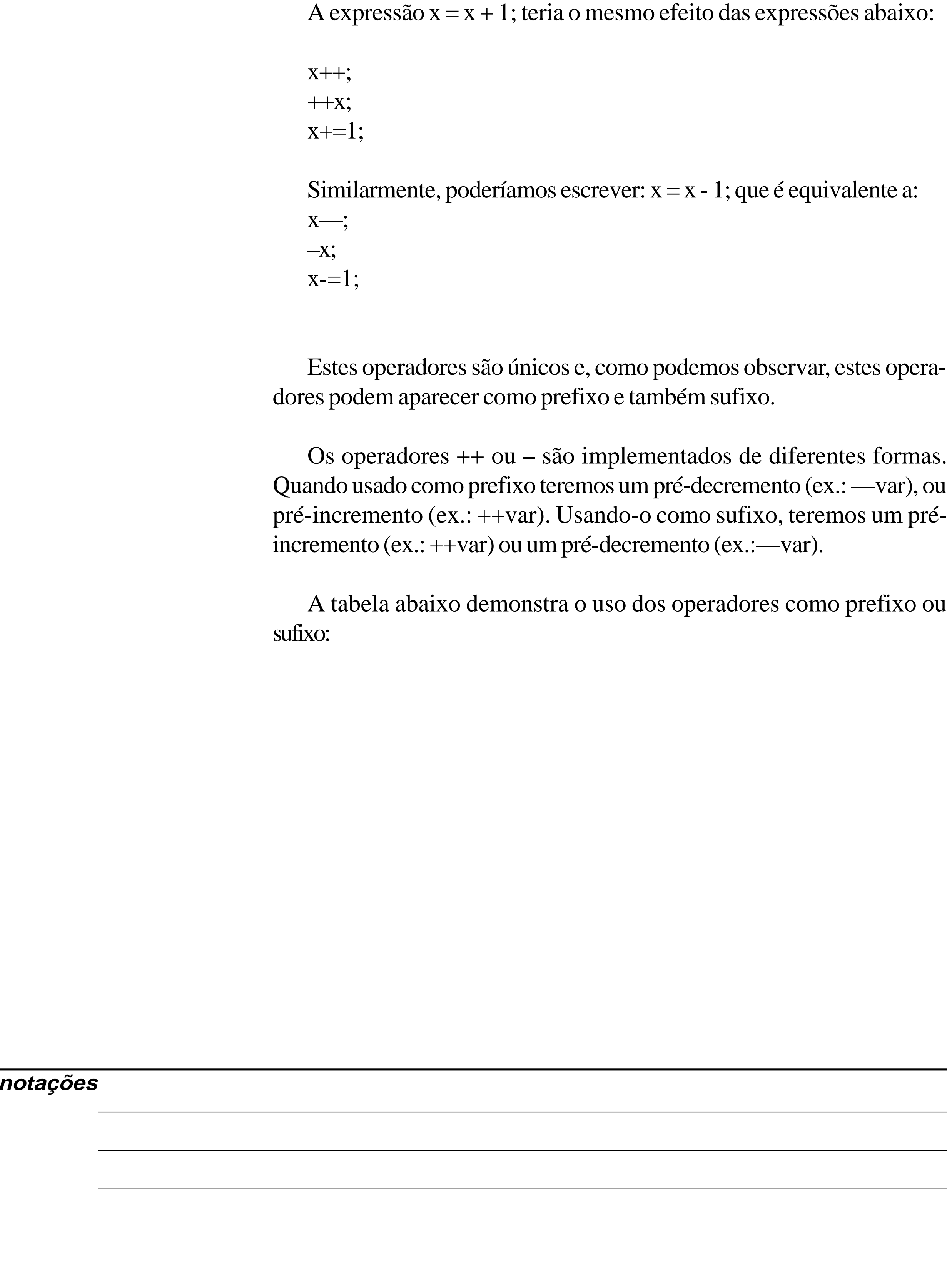
Operadores Incremento

- Incremento (++).
- Decremento (—).
- Pré-incremento.
- Pré-decremento.

Em Java temos os operadores de incremento (++) e decremento (—). O operador de incremento aumenta o operando em um. O operador de decremento diminui o valor de seu operando em um. Por exemplo:

Operador	Uso	Descrição
++	var++ ++var	Usa a variável e depois incrementa. Incrementa a variável e depois usa.
—	var— —var	Usa a variável e depois faz o decremento. Faz o decremento depois usa a variável.

Anotações



A expressão x = x + 1; teria o mesmo efeito das expressões abaixo:

x++;
++x;
x+=1;

Similarmente, poderíamos escrever: x = x - 1; que é equivalente a:
x—;
–x;
x-=1;

Estes operadores são únicos e, como podemos observar, estes operadores podem aparecer como prefixo e também sufixo.

Os operadores ++ ou – são implementados de diferentes formas. Quando usado como prefixo teremos um pré-decremento (ex.: —var), ou pré-incremento (ex.: ++var). Usando-o como sufixo, teremos um pré-incremento (ex.: ++var) ou um pré-decremento (ex.:—var).

A tabela abaixo demonstra o uso dos operadores como prefixo ou sufixo:

Anotações

Vamos declarar duas variáveis:

Pré-Decremento	Pré-Incremento
int a, b;	int a, b;
a = 10; b = —a;	a = 5; b = ++a;
Após isso os valores seriam:	Após isso os valores seriam:
a fica com o valor 9 b fica com o valor 9	a fica com o valor 6 b fica com o valor 6
Pós-Decremento	**Pós-Incremento**
int a, b;	int a, b;
a = 7; b = a—;	a = 2; b = a++;
Após isso os valores seriam:	Após isso os valores seriam:
a fica com o valor 6 b fica com o valor 7	a fica com o valor 3 b fica com o valor 2

A classe abaixo irá testar as condições de uso dos operadores ++ e —:

```
public class OperatorTest {
    public static void main(String args[]) {
        int a = 23;
        int b = 5;
        System.out.println("a & b : " + a + " " + b);
        a += 30;
        b *= 5;
        System.out.println("Novas atribuicoes para a & b: "+a+" "+b);
        a++;
        b—;
        System.out.println("Apos incremento & decremento a & b: "+a+" "+b);
```

Anotações

```
        System.out.println("a— />>/ —b : " + a— + " />>/ " + —b);

        System.out.println("a++ />>/ ++b : " + a++ + " />>/ " + ++b);

        }
}
```

Saída:

```
a & b : 23 5
Novas atribuicoes para a & b: 53 25
Apos incremento & decremento a & b: 54 24
a— />>/ —b : 54 />>/ 23
a++ />>/ ++b : 53 />>/ 24
```

Anotações

Operadores Relacionais

- Os operadores relacionais *Java*.
- O método *equals*().
- O uso do operador == em objetos.
- Características e exemplos de uso do método *equals*().

Estes operadores foram formulados para efetuar comparações entre valores. Lembrando que para efetuar comparações entre objetos, *Java* disponibiliza também outros recursos como o método *equals* e o *instanceof* por exemplo, que serão estudados neste curso.

Anotações

Operador	Exemplo	Descrição
==	A == B	Verifica se A é igual a B
!=	A != B	Verifica se A é diferente de B
<	A < B	Verifica se A é menor que B
>	A > B	Verifica se A é maior que B
<=	A <= B	Verifica se A é menor ou igual a B
>=	A >= B	Verifica se A é maior ou igual a B

O Operador == quando usado em Objetos

- Retorna *true* objetos estiverem apontando para a mesma área de memória, ou seja, duas referências iguais.
- Se dois objetos tiverem dados idênticos o operador == irá retornar *false*, pelo fato de residirem em diferentes áreas de memória.
- Uma mesma referência pode ter identificadores diferentes.

O Método *Equals*()

- O método *equals*() se encontra na classe *Object* e faz a mesma coisa que o operador ==, porém deve ser usado apenas objetos. Este é herdado da classe *Object*.
- Algumas classe da **API** fazem *overriding* no método *equals*, caso necessário você também pode fazer, este tipo de implementação visa facilitar a comparação de dados em diferentes objetos.

Anotações

• As classes abaixo fazem *overriding* no método *equals*():

String (em testes).
Todas as *Wrappers* (em testes).
Date (se a data e a hora forem iguais).
BitSet (se tiver a mesma seqüência de *bits*).
File (se o relative path name forem iguais.).

Exemplo:

```
Boolean b1 = new Boolean(true);
Boolean b2 = new Boolean(true);

System.out.println( "Test: "+ b1.equals( b2 ) );  = true
System.out.println( "Test: "+ ( b1 == b2 ) );  = false
```

O exemplo abaixo mostra o uso de Operadores Relacionais:

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

        int i = 27;
        int j = 22;
        int k = 22;
        System.out.println("Valores de Variaveis...");
        System.out.println("   i = " + i);
        System.out.println("   j = " + j);
        System.out.println("   k = " + k);

          //Maior que
        System.out.println("Maior que...");
        System.out.println("   i > j = " + (i > j));   //false
        System.out.println("   j > i = " + (j > i));    //true
        System.out.println("   k > j = " + (k > j));    //false, eles são iguais
```

Anotações

```
        //Maior ou igual
        System.out.println("Maior ou igual...");
        System.out.println("   i >= j = " + (i >= j));  //false
        System.out.println("   j >= i = " + (j >= i));  //true
        System.out.println("   k >= j = " + (k >= j));  //true

        //Menor que
        System.out.println("Menor que...");
        System.out.println("   i < j = " + (i < j));    //true
        System.out.println("   j < i = " + (j < i));    //false
        System.out.println("   k < j = " + (k < j));    //false

        //Menor ou igual
        System.out.println("Menor ou igual...");
        System.out.println("   i <= j = " + (i <= j));  //true
        System.out.println("   j <= i = " + (j <= i));  //false
        System.out.println("   k <= j = " + (k <= j));  //true

        //Igual
        System.out.println("Igual...");
        System.out.println("   i == j = " + (i == j));  //false
        System.out.println("   k == j = " + (k == j));  //true

        //Diferente
        System.out.println("Diferente...");
        System.out.println("   i != j = " + (i != j));  //true
        System.out.println("   k != j = " + (k != j));  //false

    }
}
```

Anotações

Saída:

```
Valores de Variaveis...
    i = 27
    j = 22
    k = 22
Maior que...
    i > j = true
    j > i = false
    k > j = false
Maior ou igual...
    i >= j = true
    j >= i = false
    k >= j = true
Menor que...
    i < j = false
    j < i = true
    k < j = false
Menor ou igual...
    i <= j = false
    j <= i = true
    k <= j = true
Igual...
    i == j = false
    k == j = true
Diferente...
    i != j = true
    k != j = false
```

Anotações

O Operador *Instanceof*

- *instanceof* que não existe em C ou C++.
- Usado para testar instâncias de classes.
- Uso permitido em objetos que estejam em polimorfismo.

Um grande destaque para este capítulo de operadores é exatamente o operador *instanceof* que não existe em **C++**, ou muito menos em **C**, que não é orientado a objetos.

O operador *instaceof* serve exclusivamente para testar instancias de classes, e é um operador muito eficiente. Existe a possibilidade de uso deste operador até mesmo em objetos que estão em polimorfismo.

Anotações

instanceof	**Testa se um objeto:** É uma instância de uma determinada classe ou subclasse. Ou se implementa uma *interface*.
	Notas: Todos os objetos são instâncias da classe Object. Todas as *numeric wrappers* são instâncias da classe *Number*. Todos os *applets* são instâncias da classe *Applet*. Qualquer classe que implemente uma determinada *interface* é uma instância desta *interface*. Qualquer classe que seja uma subclasse de uma outra classe é instância desta classe.
	Exemplos: System.out.println((new Integer(5)) *instanceof* Number));
	Saída: `true`
	Motivo: Todas as *wrappers classes extends Number class*.

Anotações

- Operadores relacionais condicionais.
- Construir expressões mais elaboradas.
- O operadores binários &&, ||, &, |, ^.
- O operador unário (!).

Os operadores relacionais abaixo são também operadores condicionais. Estes operadores servem para construir expressões mais elaboradas.

Abaixo encontraremos uma lista com seis operadores, sendo que cinco deles são operadores binários (**&&, ||, &, |, ^**) e um unário (**!**).

Anotações

<table>
<tr><th>Op</th><th>Exemplo</th><th>Tipo</th><th>Descrição</th></tr>
<tr><td>&&</td><td>(Exp1) && (Exp2)</td><td>binário</td><td>Se Exp1 e Exp2 forem verdadeiros, esta condicional irá retornar true. A expressão Exp2 não será avaliada se a Exp2 for falsa.</td></tr>
<tr><td>||</td><td>(Exp1)||(Exp2)</td><td>binário</td><td>Se Exp1 ou Exp2 for true, esta condicional retornará true. A expressão Exp2 não será avaliada se a Exp1 for true.</td></tr>
<tr><td>!</td><td>!(Exp)</td><td>unário</td><td>O valor boolean é invertido. Se Exp for true com este operador o retorno será false e vice-versa.</td></tr>
<tr><td>&</td><td>(Exp1) & (Exp2)</td><td>binário</td><td>Se Exp1 e Exp2 forem verdadeiras, esta condicional irá retornar true. A expressão Exp2 sempre é avaliada.</td></tr>
<tr><td>|</td><td>(Exp1) | (Exp2)</td><td>binário</td><td>Se Exp1 ou Exp2 for true, esta conditional retornará true. A expressão Exp2 sempre é avaliada.</td></tr>
<tr><td>^</td><td>(Exp1) ^ (Exp2)</td><td>binário</td><td>O único momento em que teremos o retorno de true será quando a Exp1 e a Exp2 forem diferentes, caso contrário retornará false. então retornaremos true e, se Exp1.</td></tr>
</table>

Anotações

Operadores *Boolean*

- *Boolean AND.*
- *Boolean OR.*
- *Short Circuit AND.*
- *Short Circuit OR.*

Os tipos são *boolean* e, os resultados são *boolean* também. Todas as condições serão avaliadas da esquerda para a direita.

Exemplo OR:
int x = 0;
if (true | (x—) == 5);
results: x = 6

Exemplo AND:
int x = 3;
if (false | (x—) == 5);
results: x = 2

Anotações

Short Circuit Logical Boolean - AND/OR

Os tipos são *boolean* e, os resultados são boolean também. Todas as condições serão avaliadas da esquerda para a direita.

Exemplo AND:

```
int z = 0;
if (false && (z++) == 5);
results:  z = 0
```

Exemplo OR:

```
int s = 0;
if (true || (s++) == 5);
results:  s = 0
```

Anotações

Bitwise Operators

- *Bitwise & (AND).*
- *Bitwise | (OR).*
- *Bitwise ~ (reverse bits).*
- *Bitwise ^ (XOR - eXclusive OR).*

Os *bitwise operators* são aplicados a números inteiros. Ambos os lados são avaliados.

Operador	Exemplo	Operação
&	A & B	*bitwise and*
|	A | B	*bitwise or*
^	A ^ B	*bitwise xor*
~	~A	*bitwise complement*

Anotações

Bitwise - AND/OR

Os tipos são inteiros. O Resultado é inteiro.
int Y = (1 | 2); Resultado: Y = 3
0001
0010
——-
0011 = 3 base 10

Exemplos:
(4 & 3) + (2 | 1) = 3 (na base 10)
4 = 1000 2 = 0010
3 = 0011 1 = 0001
AND —— OR ——
0000 + 0011
0 + 3 = 3 Base 10

Bitwise ~ reverse bits. Ele irá retornar o complemento

Ex: int i = 0x00000008; i = ~i; Resultado i = 0xFFFFFF7
Why: Binary 0000 = 0 reverse bits to 1111 = F
Why: Binary 1000 = 8 reverse bits to 0111 = 7.

Bitwise ^ - exclusive or – XOR

Retorna true se o valor Boolean das expressões forem diferentes.

(true ^ false); Retorna true!
(true ^ true); Retorna false!
(false ^ false); Retorna false!

Observação

Math.pow (***double a, double b***) – Este método calcula raiz quadrada. Onde **a** é a base e **b** o expoente. Algumas linguagens usam **^** mas não o *Java*.

Anotações

Shift Operator

- *Shift* aritmético.
- *Shift* aritmético à direita.
- *Shift* aritmético à esquerda.
- *Shift* lógico.

A tabela abaixo mostra o uso do *shift* operator aritmético e lógico:

Operador	Exemplo	Descrição
>>	X >> 2	*shift* de 2 *bits* no valor de **X**.
<<	Y << 3	*shift* de 3 *bits* no valor de **Y**.
>>>	Z >>> 4	*shift* de 4 *bits* no valor de **Z**, deslocando logicamente.

Anotações

Os operadores que fazem a manipulação de *bits* em um determinado dado, são chamados de *shift operators*. Estas manipulações deslocam *bits* para a direita ou para a esquerda. As operações com *shift* aritméticos são feitas mantendo o *bit* de sinal (*bit* responsável por definir um determinado número como positivo ou negativo).

Temos ainda um *shift* operator lógico que faz o deslocamento de *bits* desprezando o sinal o sinal.

Na composição de uma expressão que faz o deslocamento de *bits*, temos o operando seguido do operador de *shift* e o número de posições que serão deslocadas. O *shift* sempre ocorre no sentido indicado pelo operador.

Exemplo:

A seguinte expressão indica que o número 13 será deslocado uma casa a direita:

13 >> 1

A representação binária do número 13 é 1101. O resultado da operação é 1101 shifted a direita em uma posição é 110, ou 6 em decimal. O *bits* da direita são preenchidos por zero.

Anotações

Bitwise Shifting Operator

- << - *Bits shift* para a esquerda.
- >>> - *Bits shift* para a direita (incluindo o **bit** de sinal).
- >> - *Bits shift* para a direita.
- *Bitwise Shifting*.
- Regras para Negativos em Binário.

Anotações

Bitwise Shifting:

Bitwise Shifting:

<< - *Bits shift* para a esquerda.

>>> - *Bits shift* para a direita (incluindo o *bit* de sinal).

>> - *Bits shift* para a direita.

Bitwise shifting retorna um *int*. Se você fizer um *cast* para um valor menor, estará perdendo *bits*.

Exemplos:

var = “0010 0011”
var = var >> 1 , resultado = 0001 0001
var = var << 1 , resultado = 0010 0010
var = var >>> 1, resultado = 0001 0001

Quando aplicamos >> e >>> a números negativos, temos resultados diferenciados.

number = -8
11111111111111111111111111111000 = -8

number = number >> 1 (-4)
11111111111111111111111111111100 = -4
Note que o bit de sinal à esquerda permanece em 1.

number = number << 1 (volta a -8)
11111111111111111111111111111000 = -8
Note que temos um preenchimento de bit a direita com “0”.

number = number >>> 1
01111111111111111111111111111100 = 2147483644

Note que o *bit* de sinal foi alterado para “0”.

Esta é a diferença entre >> e >>>.

>> - o bit de sinal permanece em 1.

>>> - o bit de sinal é sempre 0.

Exemplos:

int i = 0x00000021; int y = 0; y = i >>2;

Resultado in base 10: i = 33 and y = 8

Resultado in base 2: i=0010 0001, y=0000 1000

Anotações

Negativos em Binário

Regras para Negativos em Binário

Se o *bit* de sinal for 1, então o número é negativo.

Passos para converter um número negativo em 32 *bytes* em binário.

- Aplicamos o complemento de *byte*. Para fazer isso você deve inverter o *bit*. Todos os 1's se tornam 0's e todos os 0's se tornam 1's.
- Adicionamos 1 *byte*.

Exemplos:

Integer (32 bits)	Binary
-1	11111111111111111111111111111111 00000000000000000000000000000000 (2's compliment) 00000000000000000000000000000001 Soma 1 00000000000000000000000000000001 Igual à 1 Resposta: -1 (Mantém o sinal).
-2	11111111111111111111111111111110 00000000000000000000000000000001 (2's compliment) 00000000000000000000000000000001 Soma 1 00000000000000000000000000000010 Igual à 2 Resposta: -2 (Mantém o sinal).
-3	11111111111111111111111111111101 00000000000000000000000000000010 (2's compliment) 00000000000000000000000000000001 Soma 1 00000000000000000000000000000011 Igual à 3 Resposta: -3 (Mantém o sinal).

Anotações

Ternary Operator

- Este operador usa a notação *infix*.
- Usado para inicializar variáveis ou objetos.

Outro grande destaque entre os operadores é o ternary, que permite uma excelente e elegante forma de inicializar variáveis e objetos.

Operador:

?:

(<boolean exp> ? <true exp> : <false exp>)

Exemplo:

Expressão lógica ? v_1 : v_2

Descrição:

Caso a expressão lógica seja verdade o v_1 será retornado, caso contrário será retornado v_2.

Anotações

Short cut Assigment Operators

- Operadores
  - +=
  - -=
  - *=
  - /=
  - %=
  - &=
  - |=
  - ^=
  - <<=

Abaixo encontraremos os operadores de atribuição resumidos. Estes operadores nos permitem efetuar uma operação e atribuir o resultado da mesma a um de seus operandos.

Anotações

Operador	Expressão Reduzida	Expressão Expandida
+=	A += B	A = A + B
-=	A -= B	A = A - B
*=	A *= B	A = A * B
/=	A /= B	A = A / B
%=	A %= B	A = A % B
&=	A &= B	A = A & B
|=	A |= B	A = A | B
^=	A ^= B	A = A ^ B
<<=	A <<= B	A = A << B
>>=	A >>= B	A = A >> B
>>>=	A >>>= B	A = A >>> B

Anotações

Outros Operadores

- Operadores dissolvidos ao longo do curso:
  - []
  - .
  - (params)
  - (type)

Abaixo temos uma tabela onde encontramos outros operadores suportados por *Java*, estes operadores são estudados conforme em suas respectivas seções:

Anotações

Operador	Descrição	Seção Pertinente
[]	Usado para declarar arrays, criar arrays, e acessar elementos de um array.	Este operador é estudad o no módulo de arrays
.	Usado para nomes qualificados.	Este operador é visto na seção que trata FQN
(params)	Delimitação para lista de parâmetros, separados por virgulas.	Estudamos este operador juntamente com métodos e construtores
(type)	Casts (coerção) de valores para os tipos especificados	Este operador é visto em casting de variáveis
new	Criação de novos objetos	Visto desde Passagem por Referência

Anotações

Tabela completa de *Operators Java*.

Operador	**Precedência**	**Associatividade**
() [] .	1	ESQUERDA para DIREITA
++ —	2	DIREITA para ESQUERDA
+ -	2	DIREITA para ESQUERDA
~	2	DIREITA para ESQUERDA
!	2	DIREITA para ESQUERDA
()	2	DIREITA para ESQUERDA
* / %	3	ESQUERDA para DIREITA
+	4	ESQUERDA para DIREITA
-	4	ESQUERDA para DIREITA
<< >> >>>	5	ESQUERDA para DIREITA
< > <= >=	6	ESQUERDA para DIREITA
instanceof	6	ESQUERDA para DIREITA
== !=	7	ESQUERDA para DIREITA
&	8	ESQUERDA para DIREITA
^	9	ESQUERDA para DIREITA
|	10	ESQUERDA para DIREITA
&&	11	ESQUERDA para DIREITA
||	12	ESQUERDA para DIREITA
(? :)	13	DIREITA para ESQUERDA
=, +=, /= %=, +=, -= <<=, >>=, >>>= &=, ^=, ~=	14	DIREITA para ESQUERDA

Anotações

Bases Numéricas

A conversão entre bases, e operações entre elas, como as que acontecem com bitwise operators são muito importantes, pois ela é um dos itens avaliados na prova de certificação para programadores Java. Será avaliado na prova se você sabe fazer conversões entre bases numéricas, por exemplo, transforma decimal em binário.

O motivo pelo qual as bases numéricas são itens de prova é o fato de que a linguagem de programação tem diversos recursos voltados às bases octal, binária e hexadecimal. Os entendimentos destes itens também são importantes para o conhecimento das variáveis primitivas, já que os tipos integrais suportam números em hexadecimal e octal.

Este curso não irá se aprofundar em bases numéricas, o objetivo é ensinar itens cruciais e elementares para a prova e entendimento da linguagem.

> **Nota:**
> Lembrem-se, conversões e operações entre bases binárias também são avaliados na prova de certificação *Java*.

O nome da base representa como os dígitos são representados em um sistema numérico.

Por exemplo:

Base 10 tem 10 dígitos (0-9)
Base 2 tem 2 (0-1)
Base 8 tem 8 (0-7)
Base 16 tem 16 (0-9, e A-F).

Como podemos ver a partir destes exemplos, a representação da base não inclui o digito da própria base para representá-lo. Por exemplo, a base octal não tem o número oito como representação, assim como a base decimal não tem o digito 10 na sua grafia.

Anotações

Exemplo dos números usados em quarto bases numéricas:

Base 2 – 0,1
Base 8 – 0,1,2,3,4,5,6,7
Base 10 - 0,1,2,3,4,5,6,7,8,9
Base 16 - 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F

Agora contando em diferentes bases:

Base 2 – 0,1,10,11,100,101,110,111, etc..
Base 8 - 0,1,2,3,4,5,6,7,10,11,12,13,14,15,16,17,20,21, 22,23,24,25,26,27,30, etc...
Base 10 - 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18, 19,20, etc...
Base 16 - 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F,10,11,12,13,14,15,16,17,18,19,1A,1B,1C, 1D,1E,1F,20, etc...

Por exemplo, temos a seguinte conversão de Base 16 to Base 10: A=10, B=11, C=12, D=13, E=14, F=15.

Um item que podemos observar é o fato de que a representação numérica para todas as bases é zero.

A seguir temos algumas tabelas de conversões de números para a base 10:

Anotações

Tabela 1:

10 (Base 10) = 10 (Base 10)	Fórmula: (1*10) + (0*1) = 10
10 (Base 2) = 2 (Base 10)	Fórmula: (1*2) + (0*1) = 2
10 (Base 8) = 8 (Base 10)	Fórmula: (1*8) + (0*1) = 8
10 (Base 16) = 16 (Base 10)	Fórmula: (1*16) + (0*1) = 16

Tabela 2:

21 (Base 10) = 10 (Base 10)	Fórmula: (2*10) + (1*1) = 21
101 (Base 2) = 5 (Base 10)	Fórmula: (1*4) + (0*2) + (1*1) = 5
21 (Base 8) = 8 (Base 10)	Fórmula: (2*8) + (1*1) = 17
21 (Base 16) = 16 (Base 10)	Fórmula: (2*16) + (1*1) = 33

Tabela 3:

1010 (Base 2) = 10 (Base 10)	Fórmula: (1*8)+(0*4)+(1*2) + (0*1) = 10
1111 (Base 2) = 15 (Base 10)	Fórmula: (1*8)+(1*4)+(1*2) + (1*1) = 15
1 0000 (Base 2) = 16 (Base 10)	Fórmula: (1*16)+(0*8)+(0*4)+(0*2) + (0*1) = 16
A (Base 16) = 10 (Base 10)	Fórmula: (10*1) = 10
FF (Base 16) = 255 (Base 10)	Fórmula: (15*16)+(15*1) = 255
100 (Base 16) = 256 (Base 10)	Fórmula: (1*256) (0*16)+(0*1) = 255

Anotações

Conversão de Base 2 para Base 16

Para converter a Base 2 (binária) para a Base 16 (hexadecimal), faça grupos de 4 bits e, então use a tabela abaixo para converter para base 16. Você pode usar o mesmo princípio para converter base 16 para a base 2.

Tabela de conversão da Base 10 e 16

Base 10	Base 2	Base 16	Base 10	Base 2	Base 16
0	0000	0	8	1000	8
1	0001	1	9	1001	9
2	0010	2	10	1010	A
3	0011	3	11	1011	B
4	0100	4	12	1100	C
5	0101	5	13	1101	D
6	0110	6	14	1110	E
7	0111	7	15	1111	F

Anotações

Laboratório 4:

Capítulo 5: Usando Operadores *Java*

Concluir o(s) exercício(s) proposto(s) pelo instrutor. O instrutor lhe apresentará as instruções para a conclusão do mesmo.

Laboratório 4 - Capítulo 5

1) Analise e encontre o erro no código abaixo:

```
public class exemplo03 {
  public static void main (String args[])
  {
  int x =10;
  int y =3;

  System.out.println("x =" + x);
  System.out.println("y =" + y);
  System.out.println("-x =" + (-x));
  System.out.println("x/y=" + (x/y));
  System.out.println("Resto de x por y=" + (x/y));
  System.out.println("inteiro de x por y =" + (x/y));
  System.out.println("x +1 =" + (x++));
  }
}
```

Anotações

2) Crie um programa que inicialize uma variável *double* de acordo com o resto de uma divisão qualquer, desde que esse seja ímpar, caso contrário inicialize com 1000. Utilize para isto o operador *ternary*.

Anotações

3) Faça a conversão dos números que estão na Base 16 para a Base 2 e para Base 10.

21 (base 16) = _____ _____ (base 2) = _____ (base 10)
08 (base 16) = _____ _____ (base 2) = _____ (base 10)
FF (base 16) = _____ _____ (base 2) = _____ (base 10)
1A3 (base 16) = _____ _____ _____ (base 2) = _____ (base 10)
0xFEDC123 (base 16)= _____ _____ _____ _____ _____ _____ _____ (Base 2)

Anotações

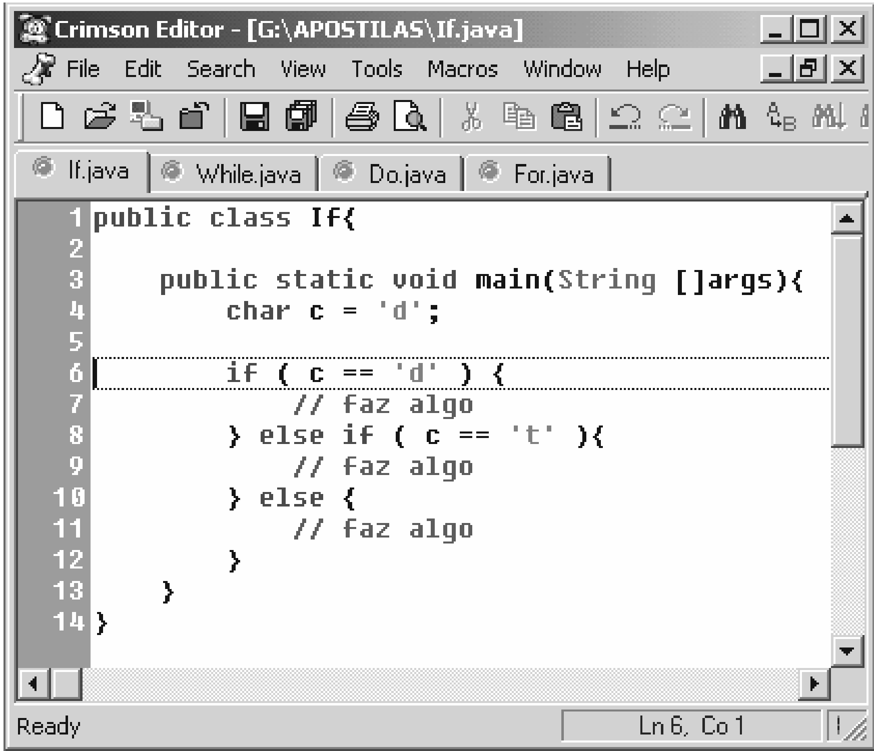
Capítulo 6:

Controle de Fluxo

Controle de Fluxo

A sintaxe dos comandos de controle de fluxo para *Java* se assemelha muitos aos de **C/C++**.

if / else if / else



Anotações

while

Crimson Editor - [G:\APOSTILAS\While.java]

File Edit Search View Tools Macros Window Help

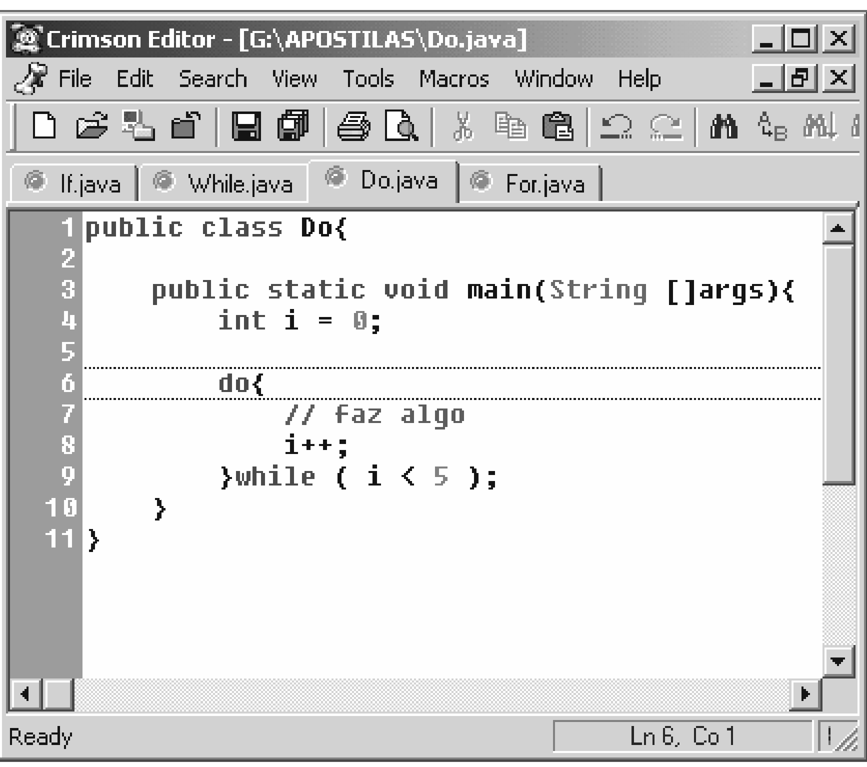
If.java | While.java | Do.java | For.java

```java
public class While{

    public static void main(String []args){
        int i = 0;

        while ( i < 5 ) {
            // faz algo
            i++;
        }
    }
}
```

Ready Ln 6, Co 1

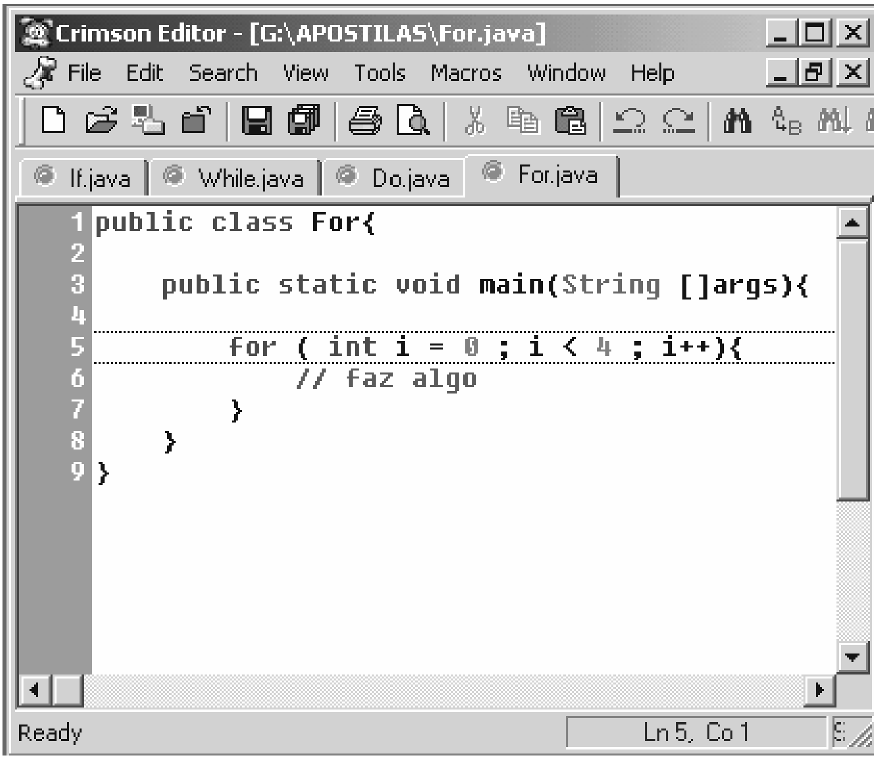
Anotações

do



Anotações

for



Anotações

switch/case/default

int op = 3

switch (*op*){

case **1**:
    //entra aqui se *op* for igual a **1**
    break; // termina o *loop*

case **2**:
    //entra aqui se *op* for igual a **2**
    break; // termina o *loop*

case **n**:
    //entra aqui se *op* for igual a **n**
    break; // termina o *loop*

default:
    //entra aqui se *op* for diferente de **2** e **2**

}

Anotações

Laboratório 5:

Capítulo 6: Controle de Fluxo

Concluir o(s) exercício(s) proposto(s) pelo instrutor. O instrutor lhe apresentará as instruções para a conclusão do mesmo.

Laboratório 5 - Capítulo 6

1) Faça um programa *Java* que imprima na tela todos os números ímpares em uma contagem até 25, use o *loop for*.

Anotações

2) Faça um programa *Java* que imprima na tela todos os números pares em uma contagem até 25, use o *loop do*.

Anotações

3) Faça um programa *Java* que imprima na tela todos os números em uma contagem até 25 exceto os números 8, 17, 21, use o *loop while*.

Anotações

4) Faça um programa *Java* que imprima na tela:

- Bom dia
- Boa tarde
- Boa noite

Use para isto uma variável *int*, com horas inteiras apenas. Implemente o código com *if/else/elseif*.

Exemplo:

- 00:00 as 11:00 - Bom dia
- 12:00 as 17:00 - Boa tarde
- 18:00 as 23:00 - Boa noite

Anotações

5) Repita o exercício anterior, porém ao invés de usar *if*, use *switch*. Use como seletor uma variável *char* conforme exemplo abaixo:

- 'D'- Bom dia
- 'T'- Boa tarde
- 'N'- Boa noite

Anotações

Capítulo 7:

Arrays

Arrays

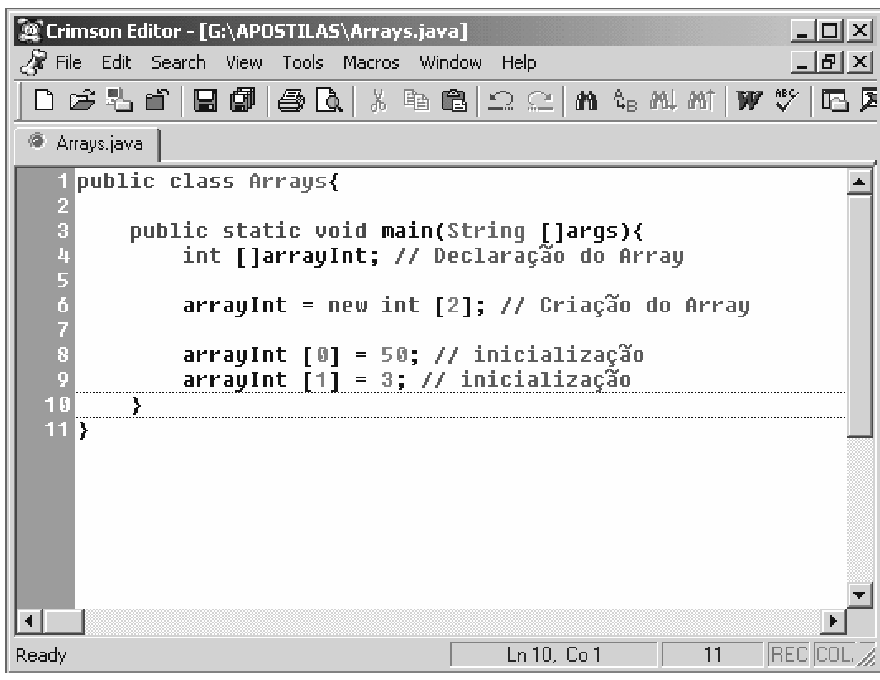
- *Array Java* são objetos.
- Objetos *arrays* não podem ser redimensionados.
- Assim como em **C/C++** o índice inicia em 0.
- O método *main* lê *strings* da linha de comando.

Os *arrays*, como a maioria dos tipos manipuláveis em *Java*, são objetos. Objetos *arrays* não podem ser redimensionados, caso seja necessário ter um *array* maior, deve-se então criar um novo *array* maior e copiar o conteúdo para este novo.

Assim como em **C/C++** o índice inicia em 0.

Anotações

Declaração / Criação / Inicialização de Atributos



Anotações

Acesso aos Elementos do *Array*

```
public class Arrays{

    public static void main(String []args){
        int []arrayInt; // Declaração do Array

        arrayInt = new int [2]; // Criação do Array

        arrayInt [0] = 50; // inicialização
        arrayInt [1] = 3; // inicialização

        /*
         * Acesso aos Elementos do Array
         */
        System.out.println("Elemento 1:" + arrayInt [0]);
        System.out.println("Elemento 2:" + arrayInt [1]);
    }
}
```

É possível ler *String* diretamente da linha de comando. O primeiro passo para isso é preparar nosso programa para receber e manipular os argumentos passados. *Java* uma o *array* de *Strings* recebidos pelo método main.

Anotações

```
public class Ex{
    public static void main(String []args){
        // vamos verificar a quantidade de elementos no array
        // através da variável length
        int tamanho = args.length;
        System.out.println("Tamanho do Array: " + tamanho);
        for (int i=0; i<tamanho; i++){
            System.out.println("Elemento No." + i + " = " + args[i]);
        }
    }
}
```

Saída:

Para testar o programa devemos passa palavras na mesma linha a qual o **JRE** Java esta sendo executado. Cada palavra representa um elemento no *array*.

```
C:\test>javac TestMain.java

C:\test>java TestMain Testamos o main

Tamanho do Array: 3
Elemento Nº 1 = Testamos
Elemento Nº 2 = o
Elemento Nº 3 = main
```

Anotações

Arrays Multidimendionais

- Em *Java* é feita uma simulação de *arrays* multidimensionais.
- Temos *arrays* multidimensionais de objetos e de primitivas:
  - double arrayMulti [][] = new double[4][];
  - arrayMulti [0] = new double[4];
  - arrayMulti [1] = new double[4];
  - arrayMulti [2] = new double[4];
  - arrayMulti [3] = { 0, 1, 2, 3 };

Em *Java* é feita uma simulação de *arrays* multidimensionais.

Temos *arrays* multidimensionais de objetos e de primitivas:
double arrayMulti [][] = new double[4][];
arrayMulti [0] = new double[4];
arrayMulti [1] = new double[4];
arrayMulti [2] = new double[4];
arrayMulti [3] = { 0, 1, 2, 3 };

Anotações

Temos uma simulação de *arrays* multidimensionais em *Java*, através da criação *arrays* de *arrays*.

```
double arrayMulti[][] = new double[4][4];
```

Outra maneira de fazer a mesma declaração é:

```
double arrayMulti [][] = new double[4][];
arrayMulti [0] = new double[4];
arrayMulti [1] = new double[4];
arrayMulti [2] = new double[4];
arrayMulti [3] = { 0, 1, 2, 3 };
```

Também podemos criar *arrays* de multidimensionais usando objetos:

```
String [][]s = new String [2][2];
s[0][0] = “Ola”;
s[0][1] = “java”;
s[1][0] = “2”;
s[1][1] = “Platform”;
```

E para acessa-los:

```
System.out.println(s[0][0]); //“Ola”
System.out.println(s[0][1]); //“java”
System.out.println(s[1][0] ); //”2”
System.out.println(s[1][1] ); //” Platform”
```

Anotações

Anotações

Laboratório 6:

Capítulo 7: *Arrays*

Concluir o(s) exercício(s) proposto(s) pelo instrutor. O instrutor lhe apresentará as instruções para a conclusão do mesmo.

Laboratório 6 - Capítulo 7

1) Compile e rode o exemplo da página 161.

Anotações

2) Faça um programa que leia um argumento da linha de comando e imprima na tela os itens:

- Quantos caracteres possuem este argumento.
- A palavra em maiúsculo.
- A primeira letra da palavra.
- A palavra sem as vogais.

Anotações

3) Faça um programa que leia um argumento da linha de comando, crie um segundo *array* e coloque nele todo o conteúdo do *array* anterior, mais uma cópia de cada elemento em maiúsculo. Use o método *System.arraycopy*() da *API Java* como auxílio.

Anotações

Capítulo 8:

Programação Avançada

Interface

- As *interfaces Java* servem como especificações
- Melhora o design de orientação a objetos
- Regras para as *interfaces*:
  - Não pode ser instanciada;
  - Uma *interface* pode implementar outras *interfaces*;
  - Todos os atributos dentro de uma interface são obrigatoriamente *public, final e static*;
  - Todos os métodos devem ser o mais acessível possível, ou seja, devem ser *public*;
  - Todos os métodos em uma *interface* são obrigatoriamente abstratos, mesmo que não seja aplicada a *keyword abstract*;
  - Todos os métodos de uma *interface* devem sofrer overriding na classe a qual foi implementada.

As *interfaces Java* servem como especificações para uma determinada classe, partindo-se deste princípio, podemos dizer que uma interface especifica comportamentos padrões de um ou mais objetos.

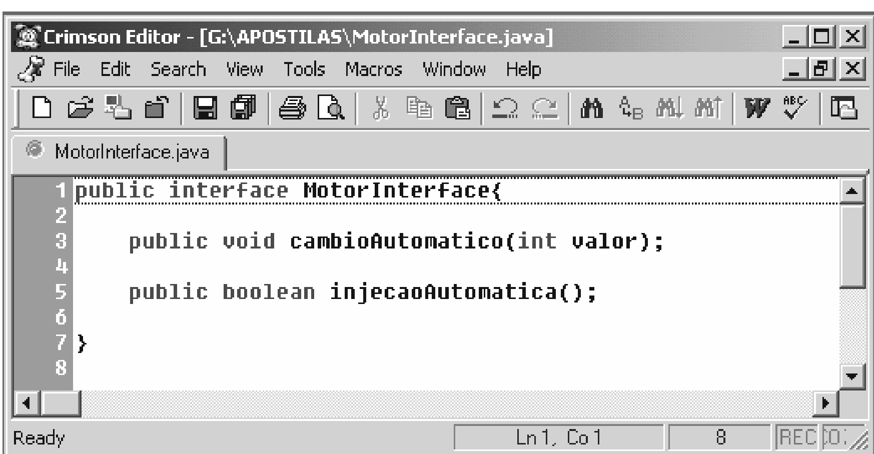
Através das *interfaces* podemos melhorar o *design* de orientação a objetos, pois desta forma é possível impor uma melhor organização de funcionalidades.

Anotações

Regras para as *Interfaces*

- Não pode ser instanciada.
- Uma *interface* pode implementar outras *interfaces*.
- Todos os atributos dentro de uma interface são obrigatoriamente *public*, final e *static*.
- Todos os métodos devem ser o mais acessível possível, ou seja, devem ser *public*.
- Todos os métodos em uma *interface* são obrigatoriamente abstratos, mesmo que não seja aplicada a *keyword abstract*.
- Todos os métodos de uma interface devem sofrer overriding na classe a qual foi implementada.

Exemplo - *Interface*



Anotações

Herança

- A herança possibilita a reutilização de código.
- *Java* possui uma enorme quantidade de dispositivos para fazer o controle de herança.
- Regra para herança:
  - Java trabalha com herança simples, diferentemente de **C++**, que por sua vez suporta herança múltipla.
  - É permitida a herança quando uma destas duas perguntas pode ser respondida:- é um? ou É um tipo de?
  - Construtores não são herdados.
  - Herança simples.

A herança é uma das características básicas de uma linguagem orientada a objetos. A herança foi inventada principalmente com o intuito de reutilização de código.

É possível construir grandes aplicações em um tempo muito hábil, utilizando a herança para reaproveitar funcionalidades rotineiras.

Java possui uma enorme quantidade de dispositivos para fazer o controle de herança, para podermos delimitar e selecionar o que devemos e o que não devemos reaproveitar destro do escopo de reuso de códigos.

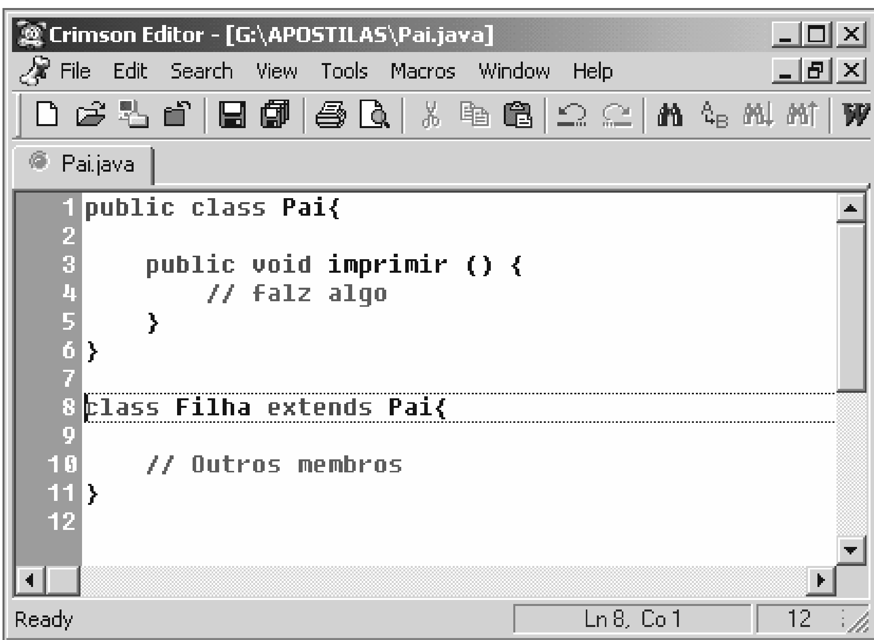
Anotações

Java trabalha com herança simples, diferentemente de **C++**, que por sua vez suporta herança múltipla. Isso implica em dizer que uma classe *Java* não pode ter duas super classes simultaneamente. Em Java construtores não são herdados.

Regra para Herança

- *Java* trabalha com herança simples, diferentemente de **C++**, que por sua vez suporta herança múltipla.
- É permitida a herança quando uma destas duas perguntas pode ser respondida:- é um? ou É um tipo de?
- Construtores não são herdados.
- Herança simples.

Exemplo - Herança



Anotações

Generalização

- A generalização em *Java* é feita através da utilização de classes abstratas e conseqüentemente apoiada no conceito de abstração através das *interfaces Java*.
- Regras para a generalização:
  - A generalização em *Java* é aplicada através da *keyword abstract*;
  - Uma classe abstrata deve obedecer ao conceito de abstração, ou seja, o conceito de generalização;
  - Uma classe abstrata não pode ser instanciada;
  - Classes abstratas podem ter métodos concretos, ou abstratos.

A generalização está entre os pontos mais fortes de uma linguagem orientada a objetos, e é claro que *Java* tem este conceito muito bem claro e excelentemente aproveitado.

A generalização em *Java* é feita através da utilização de classes abstratas e conseqüentemente apoiada no conceito de abstração através das *interfaces Java*.

Anotações

Regras para a Generalização

- A generalização em *Java* é aplicada através da *keyword abstract*.
- Uma classe abstrata deve obedecer ao conceito de abstração, ou seja, o conceito de generalização.
- Uma classe abstrata não pode ser instânciada.
- Classes abstratas podem ter métodos concretos, ou abstratos.

Exemplo - Generalização

Crimson Editor - [G:\APOSTILAS\Calculos.java]

```java
public abstract class Calculos{

    public abstract int soma (int a, int b);

    public abstract int div (int a, int b);

    public abstract int sub (int a, int b);

    public abstract int multi (int a, int b);

    public void imprimir () {
        // Método concreto
    }
}
```

Anotações

Polimorfismo

- O polimorfismo permite a um objeto ter várias formas.
- O polimorfismo é avaliado durante o *runtime*.
- Regra básica para o polimorfismo:
  - Um objeto se torna polimórfico quando é declarado como uma de suas super classes e se instancia uma de suas sub-classes.

Este conceito é com certeza o conceito mais poderoso de uma linguagem orientada a objetos e, conseqüentemente o item mais complexo também.

O polimorfismo permite a um objeto ter várias formas, ou seja, o mesmo objeto pode ter mais de um comportamento, dependendo da situação que este se encontre durante o *runtime*.

Devido ao fato de um objeto em polimorfismo ser avaliado durante o *runtime*, já que o mesmo pode tomar formas dependendo de interações dinâmicas, isto pode ocasionar problemas durante a utilização do sistema ao qual este objeto esteja agregado.

Anotações

Regra Básica para o Polimorfismo

- Um objeto se torna polimórfico quando é declarado como uma de suas super classes e se instancia uma de suas sub-classes.

Exemplo - Polimorfismo

Crimson Editor - [G:\APOSTILAS\PaiPolimorfismo.java]

```
public class PaiPolimorfismo{

    public void imprimir () {
        // falz algo
    }
}

class Filha extends PaiPolimorfismo{

    public void teste () {

        PaiPolimorfismo p = new Filha ();

        p.imprimir();
    }
}
```

Ready Ln 12, Co 18 17

Anotações

Overloading de Construtores

- Inicialização dinâmica de uma classe.
- Supri a necessidade de ter uma inicialização variada para os atributos.
- Pemiti que tenhamos quantas opções forem necessárias para inicializar uma classe.
- Regra para o *overloading* de construtores.
- A lista de argumentos deve ser diferente.

Através uma sobrecarga de construtores é possível fazer uma inicialização dinâmica de uma classe.

O que motiva fazer um *overloading* de contrutores é a necessidade de ter uma inicialização variada para os atributos.

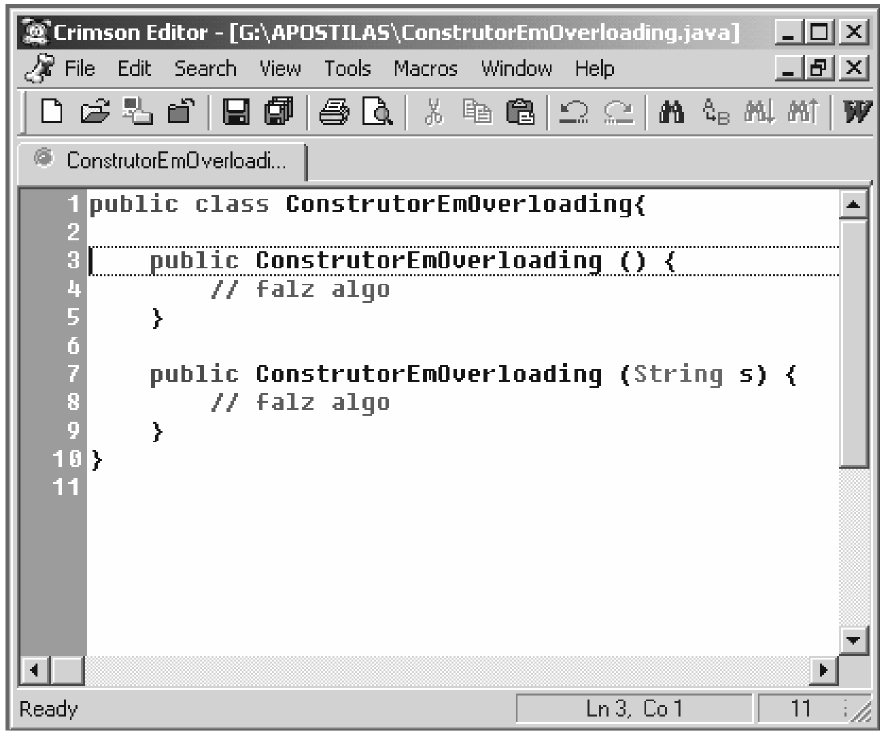
O uso de *overloading* ira permitir que tenhamos quantas opções forem necessárias para inicializar uma classe.

Regra para o *Overloading* de Construtores

• A lista de argumentos deve ser diferente.

Anotações

Exemplo - *Overloading* de Construtores



Anotações

Overloading de Métodos

- Através uma sobrecarga de métodos é possível fazer uma chamada dinâmica.
- Chamada é baseada na quantidade ou tipos recebidos.
- Regra para o *overloading* de métodos:
  - A lista de argumentos deve ser diferente;
  - Caracteriza-se overloading, métodos de lista de argumentos diferentes que estejam na mesma classe, ou ate mesmo em uma de suas super classes.

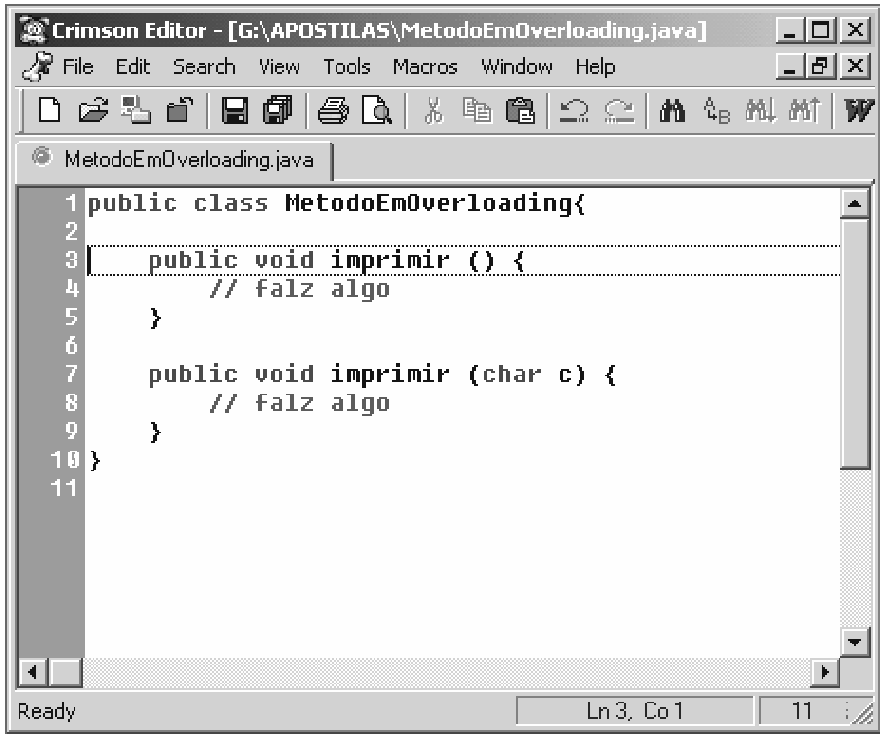
Através uma sobrecarga de métodos é possível fazer uma chamada dinâmica. Sua finalidade é ter o que podemos chamar de Atendimento sobre Medida, ou seja, você passa como argumento os elementos que tem em Mãos, e a chamada é baseada na quantidade ou tipos recebidos.

Regra para o *Overloading* de Métodos

- A lista de argumentos deve ser diferente.
- Caracteriza-se *overloading*, métodos de lista de argumentos diferentes que estejam na mesma classe, ou ate mesmo em uma de suas super classes.

Anotações

Exemplo - *Overloading* de Métodos



Anotações

Overriding

- Aplicação em métodos herdados que necessitam de ajustes.
- Regras para o *Overriding*.
  - O método deve ter o mesmo nome, tipo de retorno e lista de argumentos.
  - O método de overriding deve ser igualmente ou mais acessível que o de sua super classe.

Através da herança, podemos herdar métodos que possuem operações similares para instancias de uma classe pai ou filha, porém uma instância da classe filha pode precisar de alguns ajustes adicionais, é exatamente neste ponto onde podemos sobrescrever o método herdado, isso através do conceito *overriding*.

Regras para o *Overriding*

- O método deve ter o mesmo nome, tipo de retorno e lista de argumentos.
- O método de *overriding* deve ser igualmente ou mais acessível que o de sua super classe.

Anotações

Exemplo - *Overriding*

Crimson Editor - [G:\APOSTILAS\MetodoEmOverriding.java]

```
public class MetodoEmOverriding{

    public void imprimir () {
        // falz algo
    }

    public void imprimir (char c) {
        // falz algo
    }
}
class TestOverriding extends MetodoEmOverriding {

    public void imprimir () {
        // somente este sofreu overriding
    }
}
```

Anotações

Encapsulamento

- A *keyword private*.
- O membro se torna de uso exclusivo da classe a qual ele pertence.
- Faz a segurança das informações.
- Melhora o *design* da **OO**.

Anotações

O encapsulamento é constituído marcando-se um membro com a *keyword private*, desta forma o membro se torna de uso exclusivo da classe a qual ele pertence, sem possibilidade de acesso mesmo por sua sub-classe direta.

Crimson Editor - [G:\APOSTILAS\Encapsulando.java]

File Edit Search View Tools Macros Window Help

Encapsulando.java

```
1  public class Encapsulando{
2
3      private boolean valor; // membro encapsulado
4
5      public void imprimir () {
6          // falz algo
7      }
8
9      private void imprimir (char c) {
10         // membro encapsulado
11     }
12 }
13 class TestOverriding extends MetodoEmOverriding {
14
15     public void imprimir () {
16         // somente este sofreu overriding
17     }
18 }
```

Ready | Ln 9, Co 1 | 18

Anotações

Acesso aos Membros

- *Keyword this*.
- *Keyword super*.
- Em alguns momentos estas *keywords* são redundantes, porém ajudam na legibilidade do código.

Duas *keywords* responsáveis por viabilizar acesso a membros de super ou sub-classes, são elas *this* e *super*, a primeira delas para acesso de membros de mesma classe e, a segunda para acesso em super classe, em alguns momentos estas *keywords* são redundantes, porém ajudam na legibilidade do código.

Anotações

Exemplo - Acesso aos membros

Crimson Editor - [G:\APOSTILAS\Acesso.java]

```
public class Acesso{

    public Acesso () {
        // falz algo
    }

    public Acesso (int i) {
        this(); // acessa o outro construtor desta classe
        this.teste(); // acessa o metodo desta classe
    }

    public void teste () {
        // falz algo
    }
}
class TesteAcesso extends Acesso {

    public TesteAcesso () {
        super(10); // acessa o construtor pai
        super.teste(); // acessa o metodo pai
    }

    public void teste () {
        // falz algo
    }
}
```

Anotações

Membros *Static*

- Compartilhar valores agregados com diferentes instâncias.
- Um membro *static* também pode ser acessado através do nome de sua classe.
- O membro não pertence exatamente à uma classe, mas sim a um contexto de memória que esta ligado a uma determinada classe.

Marcando um membro como *static* o mesmo fica disponível para qualquer instância desta classe, compartilhando assim seus valores agregados com diferentes instancias.

Um membro static também pode ser acessado através do nome de sua classe, sem necessidade de qualquer instância, já que este membro não pertence exatamente a uma classe, mas sim a um contexto de memória que esta ligada a uma determinada classe para determinar sua existência dentro do âmbito do *runtime*.

Anotações

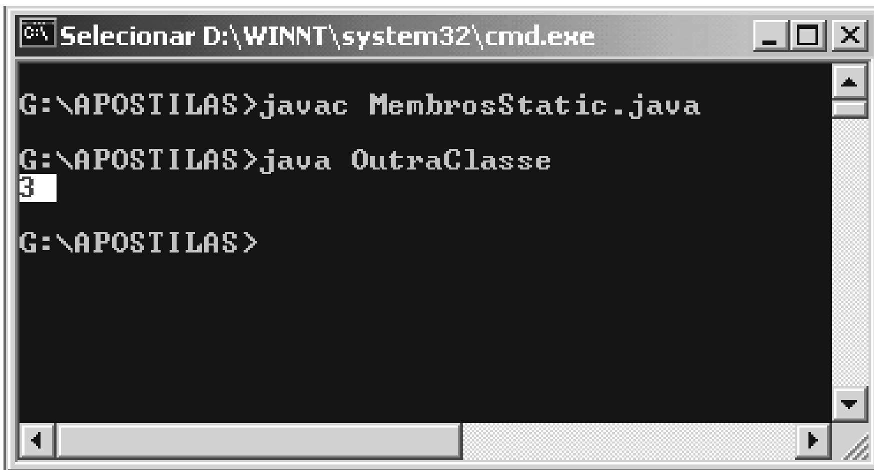
Exemplo - *Static*

Crimson Editor - [G:\APOSTILAS\MembrosStatic.java]

```
public class MembrosStatic{
    static int conta;

    public static void inc(){
        conta++;
    }
}

class OutraClasse {

    public static void main(String args[]) {
        MembrosStatic ms1 = new MembrosStatic();
        MembrosStatic ms2 = new MembrosStatic();

        ms1.conta++;

        MembrosStatic.conta++;

        ms2.conta++;

        System.out.println(MembrosStatic.conta);
    }
}
```

Anotações

Saída:



Anotações

Membros Final

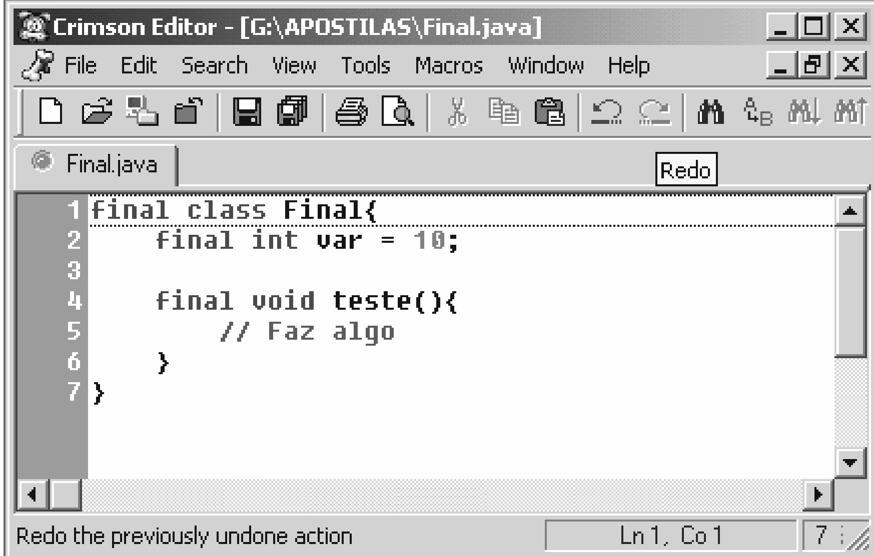
- Faz-se final um membro marcado com a *keyword* final.
- Aplicações de final.
  - Marcar uma classe como final resultará em uma classe que não poderá ser subclasseada;
  - Marcar uma variável como final a tornará uma constante;
  - Marcar um método como final, impedirá seu *overriding*.

Faz-se final um membro marcado com a *keyword* final.

Anotações

Aplicações de final:

- Marcar uma classe como final resultará em uma classe que não poderá ser subclasseada;
- Marcar uma variável como final a tornará uma constante;
- Marcar um método como final, impedirá seu *overriding*.



Anotações

Access Modifiers

- São os modificadores de acesso que filtram o acesso aos membros herdados.
- Eles também atuam no controle de acesso à classe.
- Formas de Acesso controladas pelos *access modifiers*.
  - Mesma Classe.
  - Mesmo Pacote.
  - Sub-Classe.
  - Universo.

São os modificadores de acesso que filtram o acesso aos membros e, o comportamento destes quando fazemos uma herança.

Eles também atuam no controle de acesso à classe, não só por herança, como também em momentos que a mesma é acessada por uma subclasse, por classes de outro ou do mesmo pacote.

Anotações

Tabela de acesso de membros:

Modifier	**Mesma Classe**	**Mesmo Pacote**	**Sub-Classe**	**Universo**
private	Sim	Não	Não	Não
default	Sim	Sim	Não	Não
protected	Sim	Sim	Sim	Não
public	Sim	Sim	Sim	Sim

Anotações

Laboratório 7:

Capítulo 8: Programação Avançada

Concluir o(s) exercício(s) proposto(s) pelo instrutor. O instrutor lhe apresentará as instruções para a conclusão do mesmo.

Laboratório 7 - Capítulo 8

1) Crie uma classe chamada carro e implemente a *interface* da página 171. Coloque mensagens de teste dentro dos métodos e, faça a chamada dos mesmos.

Anotações

2) Crie uma classe que estenda a classe abstrata da página 175. Faça os *overridings* necessários e implemente os métodos com suas respectivas funcionalidades.

Anotações

3) Faça um programa chamado Funcionario com as variáveis nome e idade. Após isso, implemente os seguintes *overloadings* de construtores e métodos:

public Funcionario ()
public Funcionario (String nome)
public Funcionario (int idade)
public Funcionario (String nome, int idade)
public void setInf (String nome)
public void setInf (int idade)
public void setInf (String nome, int idade)

Anotações

4) Faça o encapsulamento dos atributos do exercício anterior. E verifique se muda algo na compilação ou execução.

Anotações

5) Compile o código da página 187, retire o construtor abaixo da classe pai e resolva qualquer problema em virtude disto.

Contrutor:

```
public Acesso () {
    // falz algo
}
```

Anotações

6) Compile e rode o exemplo da página 189.

Anotações

7) Compile o programa da página 192. Teste da seguinte forma:

- Faça uma nova classe e tente estendê-la.
- Retire a *keyword* final da classe e faça uma subclasse, agora tente fazer um *overriding* do método final.
- Retire a *keyword* final do método, inclua um construtor na subclasse e tente mudar o valor da variável final.

Anotações

Capítulo 9:

Java Exceptions e Operações de I/O

Java Exceptions

- Uma *Java Exception* não é um erro. Os erros fazem parte da hierarquia das classes de *Error*.
- *Java Exception* são problemas que ocorrem durante o *runtime*.
- Instâncias da classe *exception*, ou de uma de suas inúmeras sub-classes.
- *Java Exceptions* são problemas que podem ser tratados.

São consideradas exceções *Java* todos os objetos que instanciam a classe *exception*, ou uma de suas inúmeras sub-classes. A definição correta para uma *exception Java* é um ou mais problemas que ocorrem durante o *runtime*.

Java Exceptions são todos aqueles problemas que podem ser tratados, a finalidade disto é que o programa continue sua execução mesmo após um problema ter ocorrido.

Anotações

- Tudo que esta dentro do corpo de *try* é chamado de código protegido.
- O bloco de *catch* é executado quando um objeto *exception* é lançado
- O bloco *finally* é usado quando se quer que uma tarefa seja sempre executada, independentemente da ocorrência de uma *exception*.

Abaixo segue um bom exemplo de tratamento de *exception*. Caso este mesmo problema tivesse, ocorrido em um programa feito em **C**, o mesmo seria terminado sem chance de prosseguir até que o programa seja modificado e compilado novamente.

Você pode forçar o lançamento de uma *exception* através da palavra *throw.*

Definições:

- Tudo que está dentro do corpo de *try* é chamado de código protegido
- O bloco de *catch* é executado quando um objeto *exception* é lançado.
- O último bloco é usado quando se quer que uma tarefa seja sempre executada, independentemente da ocorrência de uma *exception*.

Anotações

- Declarando uma *Exception* (*throws*).
- Lançando uma *Exception* (*throw*).

Quando não se deseja tratar uma *exception Java* podemos fazer a declaração da mesma. *Java* prove a *keyword throws* para isso. Embora não seja feito o tratamento onde a *exception* possa ocorrer, a mesma deverá ser tratada na chamada do método.

Anotações

Exemplo:

```
public class Ex{
    public static void main(String args[]) {
            Ex ex = new Ex();
            ex.test();
    }
    public void test(){
        try {
            declara (50);
            declaraLanca (50);
        } catch (ArithmeticException e) {
            System.out.println("uma divisão por zero ocorreu:" + e);
        }
    }
    //declaração da exception.
    public void declara (int i) throws ArithmeticException {
            int r = i/0;
    }
    //declaração da exception.
    public void declaraLanca (int i) throws ArithmeticException {
        throw new ArithmeticException ();
    }
}
```

Anotações

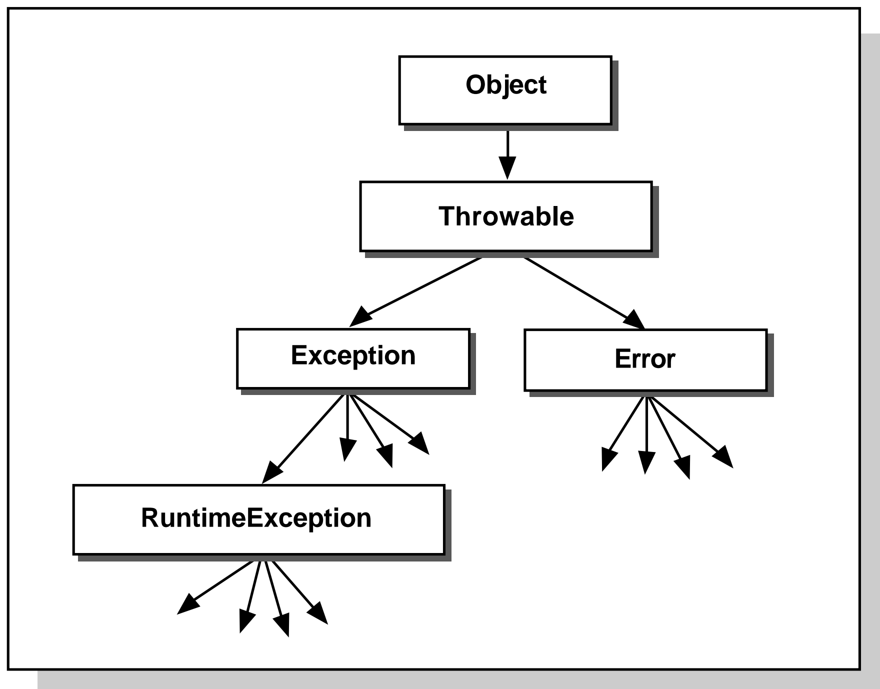
Hierarquia de Classes

- A classe *Throwable*.
- A hierarquia de *Exception*.
- A hierarquia de *Error*.

Na figura abaixo podemos comprovar o que estava no *slide* anterior, ou seja, uma *exception* não é um *error* e vice-versa. Assim como a classe *error* a classe *exception* é a raiz de sua hierarquia de classes.

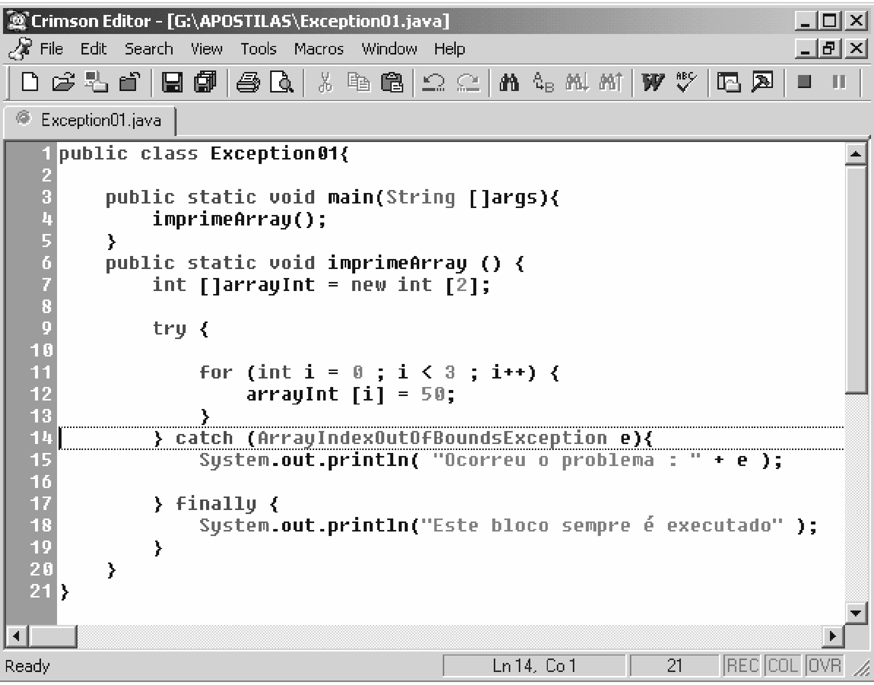
A classe *throwable* define as operações comuns de problemas que podem ser aplicados a *error* e *exception*. Desta forma, resta então as especializações feitas mais abaixo na hierarquia, ou seja, a classe de *exception* faz a especialização de problemas tratáveis e a classe *error* por sua vez, especializa condições de erros onde o sistema deve parar.

Anotações



Anotações

Exemplo: *Java Exception*



Anotações

Java I/O

- *InputStream.*
- *OutputStream.*
- *Source Stream.*
- *Sink Stream.*

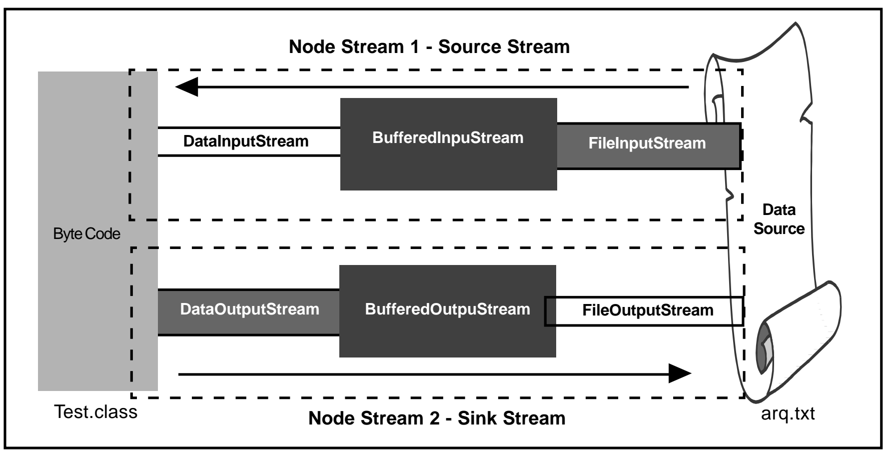
Operações de **I/O** em *Java* são totalmente componentizadas. E para que isso se tornasse possível, as classes de entrada e saída utilizam-se de todos os conceitos de orientação a objetos, isso implica em ter de conhecer bem os conceitos estudados até aqui, para que possamos obter uma maior compreensão destes componentes.

Cada *stream* abaixo representa um fluxo de dados que correm entre uma fonte de dados e o programa *Java*, as setas indicam o sentido deste fluxos.

Anotações

Definições:

- ***InputStream:*** Fluxo de entrada de *bytes*.
- ***OutputStream:*** Fluxo de saída de *bytes*.
- ***Source Stream:*** Nodo de fluxo de entrada de dados.
- ***Sink Stream:*** Nodo de fluxo de saída de dados.



Anotações

- *Java I/O libraries*:
  - *Network*.
  - *File*.
  - *Screen (Terminal, Windows, Xterm)*.
  - *Screen Layout*.
  - *Printer*.

Java suporta uma grande quantidade de bibliotecas de **I/O**. Esta quantidade se traduz em uma **API** recheada de opções para operações de **I/O**. Uma vez que temos disponível uma classe para cada tipo de *stream*, existe um tratamento mais minucioso não só para a troca de informações, mas também para o tratamento de *exceptions*.

Um fator muito interessante a ser levado em consideração, é que devido a grande componentização das *streams*, existem diversas formas de se estabelecer uma comunicação entre dois sistemas, para conhecer todas estas formas devemos consultar a **API** e analisar a hierarquia de classes, para verificarmos as heranças e, a partir delas, estabelecer as diversas aplicações de polimorfismo que podem ser feitas.

Anotações

Java I/O libraries:

- *Network*.
- *File*.
- *Screen (Terminal, Windows, Xterm)*.
- *Screen Layout*.
- *Printer*.

Anotações

Exemplo - *Java* I/O

O exemplo abaixo mostra uma troca de mensagens entre um mini-servidor **TCP/IP** e um mini-cliente **TCP/IP**, isso será feito através de uma conexão via *socket*, utilizando os conceitos de *streams* apresentados. O exemplo será explicado através de comentários inseridos no código.

Código fonte para o *Server.java*.

```
Crimson Editor - [G:\APOSTILAS\Server.java]
File Edit Search View Tools Macros Window Help
Client.java  Server.java

1  import java.net.*;
2  import java.io.*;
3
4  public class Server {
5    public static void main(String args[]) {
6      ServerSocket s = null;
7
8      // Registrando o serviço na porta 5432
9      try {
10       s = new ServerSocket(5432);
11     } catch (IOException e) {
12       // Faz algo
13     }
14
15    // Loop infinito para listen/accept
16     while (true) {
17       try {
18         // Aguarda neste ponto esperando uma conexão
19         Socket s1 = s.accept();
20
21         // Obtém um output stream associado a um socket
22         OutputStream s1out = s1.getOutputStream();
23         DataOutputStream dos = new DataOutputStream(s1out);
24
25         // Send um Ola Stream
26         dos.writeUTF("Ola Stream");
27
28         // Fecha a conexão, mas não o server socket
29         dos.close();
30         s1.close();
31       } catch (IOException e) {
32         // Faz Algo
33       }
34     }
35   }
36 }
37

Ready    Ln 4, Co 1    37    REC COL OVR
```

Anotações

Código Fonte para o *Client.java*.

Crimson Editor - [G:\APOSTILAS\Client.java]

```
import java.net.*;
import java.io.*;

public class Client {
  public static void main(String args[]) {
    try {
      // Abre a conexão no servidor, na porta 5432
      Socket s1 = new Socket("127.0.0.1", 5432);

      // Obtém um input stream para o socket
      InputStream is = s1.getInputStream();
      // Lê a informação como um data input stream
      DataInputStream dis = new DataInputStream(is);

      // Lê a entrada e imprime na tela
      System.out.println(dis.readUTF());

      // Fechamos agora a input stream e a conexão
      dis.close();
      s1.close();
    } catch (ConnectException connExc) {
      System.err.println("Não foi possível conectar.");
    } catch (IOException e) {
      // Faz algo
    }
  }
}
```

Anotações

Laboratório 8:

Capítulo 9: *Java Exceptions* e Operações de I/O

Concluir o(s) exercício(s) proposto(s) pelo instrutor. O instrutor lhe apresentará as instruções para a conclusão do mesmo.

Laboratório 8 - Capítulo 9

1) Compile e rode o código da página 207.

Anotações

2) Compile e rode o exemplo da página 210.

Anotações

3) Compile e rode o exemplo da página 215 e 216.

Anotações

Capítulo 10:

Java Collections Framework

Java Collections Framework

- *Java Collections Framework* foi incluído no Java 2.
- Estrutura de classes e *interfaces*.
- *Array versus Collections*.
- *Subset da collections* **API** para uso no **JDK 1.1**.

Java Collections Framework foi incluído no *Java 2 Platform Standard Edition* e prove uma estrutura de classes e interfaces para armazenamento e manipulação de dados e grupo de dados através de um único objeto.

Arrays Java não permitem redimensionamento, uma vez que um *array* foi criado com 10 posições, por exemplo, não será possível alterar seu tamanho para 11 nem reduzi-lo para 9, em outras palavras, não será possível alterar o tamanho de um *array* em uma posição sequer. As collections Java resolvem este problema, pois podem sofrer alteração de capacidade a qualquer momento.

Anotações

Embora *Java Collections Framework* tenha sido incluída na *Java 2 Platform Standard Edition*, a Sun Microsystem disponibiliza um *subset da collections* **API** para uso no **JDK 1.1**.

Anotações

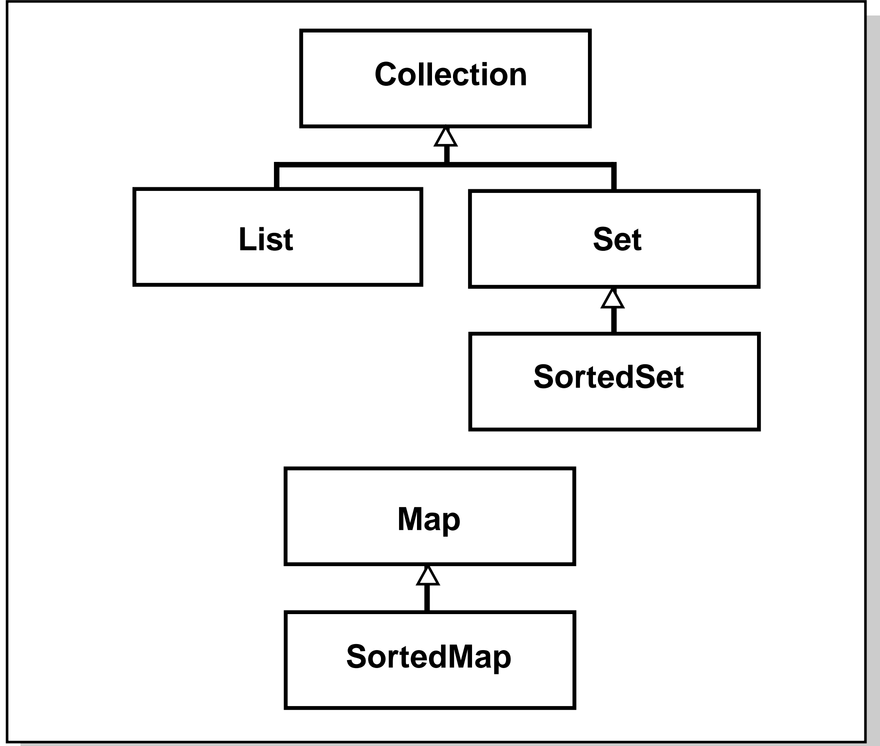
As *Collection Interfaces* e Classes

- A *interface Collection* é um grupo de objetos, onde as duplicatas são permitidas.
- A *interface Set extends Collections*, mas proíbe duplicatas:
- A *interface List extends Collection*, duplicatas são permitidas, e introduzidas em posições indexadas.
- A *interface Map* não *extends* nem *Set* nem *Collection*.

A *Collections Framework* é constituída através de um conjunto de *interfaces* para trabalhar com grupos de objetos. As diferentes *interfaces* descrevem diferentes tipos de grupos. Na maior parte, uma vez que você entenda as interfaces, você terá entendido o *framework*.

Anotações

Hierarquias



As relações hierárquicas e as quarto *interfaces* básicas da *collection framework* seguem abaixo:

- A *interface Collection* é um grupo de objetos, onde as duplicatas são permitidas;
- A *interface Set extends Collections*, mas proíbe duplicatas.
- A *interface List extends Collection*, duplicatas são permitidas, e introduzidas em posições indexadas.
- A *interface Map* não *extends* nem *Set* nem *Collection*.

Anotações

Implementações de *Collection*

- *Collections* que foram incluídas no *framework* do *Java 2*.
- Classes de *collection* vindas da primeira versão do *Java*.
- Comparando as classes históricas com o novo *framework* de *Java*, uma das principais diferenças que encontraremos é que todas as operações não são sincronizadas.
- O interessante é que podemos adicionar a sincronização às novas classes, porém você não pode remover das antigas.

A seguir será apresentada uma relação de seis implementações de collections que foram incluídas no *framework* do *Java 2*, e também as classes de *collection* vindas da primeira versão do *Java*.

Anotações

Interface	**Implementação**	**Históricas**
Set	*HashSet*	
	TreeSet	
List	*ArrayList*	*Vector*
	LinkedList	*Stack*
Map	*HashMap*	*Hashtable*
	TreeMap	*Properties*

Se formos comparar as classes históricas com o novo *framework* de *Java*, uma das principais diferenças que encontraremos é que todas as operações não são sincronizadas. O interessante é que podemos adicionar a sincronização às novas classes, porém você não pode remover das antigas.

Anotações

Collection Interface

- Disponível a partir do *jdk 1.2*.
- *Collection* suporta as operações básicas como adição e remoção de elementos.
- A *interface Collection* a raiz de toda a hierarquia de *collections*.
- Sua própria classe de *collections*.

Esta interface faz parte do *Collection Framework*, ou seja, está disponível a partir do *jdk* 1.2.

A *interface Collection* suporta as operações básicas como adição e remoção de elementos. Quando você tentar remover um elemento, somente uma instância de um elemento dentro da collection é removida, se ela existir.

Anotações

As classes que implementam esta *interface* são:

AbstractCollection, AbstractList, AbstractSet, ArrayList, BeanContext ServicesSupport, BeanContextSupport, HashSet, LinkedHashSet, LinkedList, TreeSet, Vector.

Todas estas classes herdam o comportamento genérico da *interface Collection* através de sua implementação, sendo assim a *interface Collection* a raiz de toda a hierarquia de collections. Desta forma todas estas classes irão herdar a característica da *interface collection*, que é a de representar grupo de objetos.

Devemos ressaltar que, apesar das classes citadas estarem todas no nível hierárquico abaixo da *interface Collection*, algumas permitem duplicatas e outras não, bem como, algumas são ordenadas e outras não possuem esta característica.

Caso você precise construir, seja por necessidade ou por motivos acadêmicos, sua própria classe de *collections*, você poderia implementar esta *interface* e fazer as especializações necessárias.

Anotações

Métodos Básicos da *Interface Collection*

- Remoção de elementos:
  - *boolean add*(*Object element*).
  - *boolean remove*(*Object element*).
- Operações de query:
  - *int size*().
  - *boolean isEmpty*().
  - *boolean contains(Object element)*.
  - *Iterator iterator*().

Os métodos para adição e remoção de elementos são:

- *boolean add(Object element).*
  Adiciona elementos.

- *boolean remove(Object element).*
  Remove elementos.

Anotações

A *interface Collection* também suporta as operações de query:

\- *int size*().
Retorna o tamanho da *collection*.

\- *boolean isEmpty*().
Testa se a *collection* esta vazia.

\- *boolean contains(Object element)*.
Verifica se um determinado elemento se encontra na *collection*.

\- *Iterator iterator*().
Retorna um objeto do tipo *Iterator*.

Anotações

Grupo de Elementos

- Operações aplicadas em grupo de elementos:
  - *boolean containsAll(Collection collection).*
  - *boolean addAll(Collection collection).*
  - *void clear*().
  - *void removeAll(Collection collection).*
  - *void retainAll(Collection collection).*

A *interface Collection* também suporta tarefas aplicadas em grupos de elementos, de uma só vez, ou seja, podemos executar operações em todos os elementos de uma *collection* de uma só vez. Veja abaixo:

- *boolean containsAll(Collection collection).*

- *boolean addAll(Collection collection).*

- *void clear*().

- *void removeAll(Collection collection).*

- *void retainAll(Collection collection).*

Anotações

Descrição:

\- *boolean containsAll(Collection collection)*.
Este método verifica se a *collection* atual contém todos os elementos da *collection* que foi passada por referência.

\- *boolean addAll(Collection collection).*
Este método adiciona todas os elementos à *collection* atual, fazendo uma união.

\- *void clear*().
Remove todos os elementos da *Collection*.

\- *void removeAll(Collection collection)*.
Remove da *collection* atual todos os elementos que forem iguais aos da *collection* especificada.

\- *void retainAll(Collection collection)*.
Retém somente os elementos da *collection* que estão contidos na *collection* especificada.

Anotações

A *Interface Iterator*

■ *Interface* também faz parte do *Collection Framework*, *jdk 1.2*.

- *public boolean hasNext*().
- *public Object next*().
- *public void remove*().

Esta *interface* também faz parte do *Collection Framework*, está disponível a partir do *jdk 1.2*.

O método *iterator*() da *interface Collection* retorna o tipo *Iterator*. Com os métodos da *interface Iterator*, você pode atravessar uma coleção do começo ao fim e com segurança remover os elementos da coleção subjacente. Abaixo temos os métodos desta *interface*:

- *public boolean hasNext*().
- *public Object next*().
- *public void remove*().

Anotações

Descrição:

- *public boolean hasNext*().
  Verifica se há mais elementos.

- *public Object next*().
  Retorna o próximo elemento da iteração.

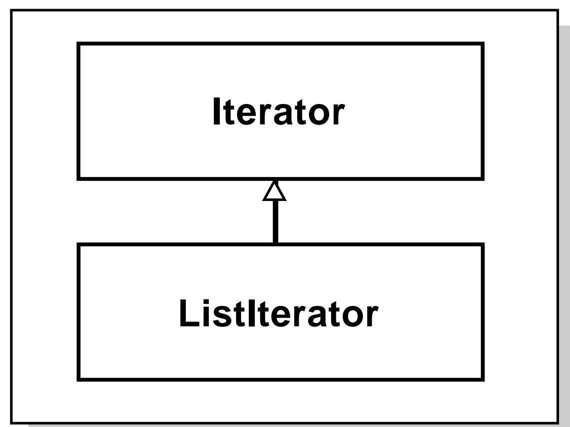
- *public void remove*().
  Remove o último elemento retornado pelo iterator.

```
Collection collection = ...;
Iterator iterator = collection.iterator();

while (iterator.hasNext()) {
	Object element = iterator.next();
	if (removalCheck(element)) {
		iterator.remove();
	}
}
```

Anotações

ListIterator Interface

- Esta *interface extends* a *interface Iterator*.
- Suportar acesso bi-direcional.
- Adicionar ou mudar os elementos em uma sub-coleção.



Anotações

Esta *interface extends* a *interface Iterator* para suportar acesso bi-direcional, bem como adicionar ou mudar os elementos em uma sub-coleção.

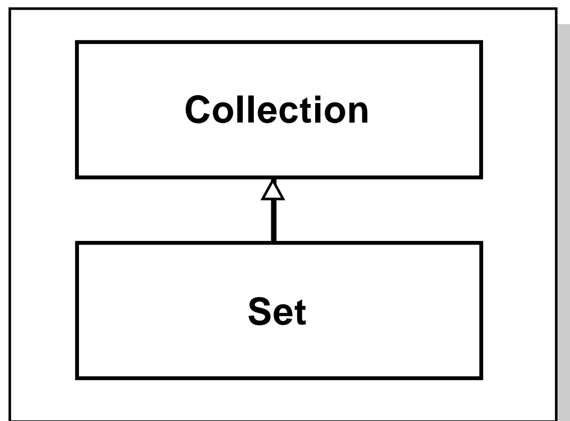
Abaixo temos um exemplo que mostra um *looping* inverso através de uma *List*. Note que o *ListIterator* é originalmente posicionado além do fim da lista (*list.size*()), como o índice do primeiro elemento é zero.

```
List list = ...;
ListIterator iterator = list.listIterator(list.size());
   while (iterator.hasPrevious()) {
           Object element = iterator.previous();
           // Processa o elemento.
   }
```

Anotações

Set interface

- A *interface Set extends Collection interface*.
- Proíbe qualquer duplicata.
- Os métodos da *interface Set* são os mesmos da *interface Collection*, a especialização de *Set* é a função da mesma não permitir duplicatas.



Anotações

A *interface Set extends Collection interface* e, por definição, ela proíbe qualquer duplicata, isso quer dizer que a classe de *collection* que implementa esta *interface* não poderá conter duplicatas. Isso é verificado através do método *equals*().

O *Collection Frameword* provê quatro implementações da *interface Set*, que são respectivamente *AbstractSet*, *HashSet*, *LinkedHashSet*, *TreeSet*.

Os métodos da *interface Set* são os mesmos da *interface Collection*, a especialização de *Set* é a função da mesma não permitir duplicatas.

Anotações

Acompanhe o exemplo de *Set*:

```
import java.util.*;

public class SetExample {

    public static void main(String args[]) {

            Set set = new HashSet(); //criação de um objeto HashSet
            set.add("1 Pato");
            set.add("2 Cachorro");
            set.add("3 Gato");
            set.add("4 Porco");
            set.add("5 Vaca");
            System.out.println("HashSet - " + set);

            //criação de um objeto TreeSet
            Set sortedSet = new TreeSet(set);
            System.out.println("TreeSet - " + sortedSet);

            //criação de um LinkedHashSet
            Set linkedSet = new LinkedHashSet (sortedSet);
            System.out.println("LinkedHashSet - " + linkedSet);
            }
    }
```

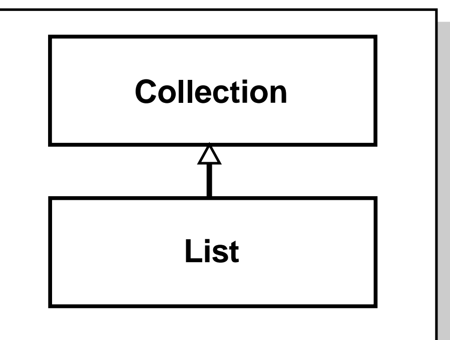
Saída:

```
HashSet - [3 Gato, 5 Vaca, 2 Cachorro, 1 Pato, 4 Porco]
TreeSet - [1 Pato, 2 Cachorro, 3 Gato, 4 Porco, 5 Vaca]
LinkedHashSet - [1 Pato, 2 Cachorro, 3 Gato, 4 Porco, 5 Vaca]
```

Anotações

List Interface

- *A interface List extends interface*.
- Definir uma collection ordenada.
- Duplicatas são permitidas.
- Métodos Comuns:
  - *void add*(*int index, Object element*).
  - *boolean addAll*(*int index, Collection collection*).
  - *Object get*(*int index*).
  - *int indexOf*(*Object element*).
  - *int lastIndexOf*(*Object element*).
  - *Object remove*(*int index*).
  - *Object set*(*int index, Object element*).
- Recursos para trabalhar com subconjuntos de *collection*, bem como executar operação em intervalos de índices.
  - *ListIterator listIterator*().
  - *ListIterator listIterator*(*int startIndex*).
  - *List subList*(*int fromIndex, int toIndex*).



Anotações

A *interface List extends interface Collection interface* para definir uma *collection* ordenada, e as duplicatas são permitidas. Estas interfaces trabalham com operações orientadas por posições.

Métodos de *List*

- *void add*(*int index, Object element*).
- *boolean addAll*(*int index*, *Collection collection*).
- *Object get*(*int index*).
- *int indexOf*(*Object element*).
- *int lastIndexOf(Object element*).
- *Object remove*(*int index*).
- *Object set*(*int index, Object element*).

Vejamos suas aplicações:

- *void add*(*int index*, *Object element*).
  Adiciona um elemento em uma posição específica.

- *boolean addAll*(*int index*, *Collection collection*).
  Adiciona todos os elementos da *collection* passada como argumento, a partir do índice especificado. O método ainda retorna *true* caso o à chamada altere a *List* atual.

- *Object get*(*int index*).
  Retorna o elemento do índice especificado.

- *int indexOf*(*Object element*).
  Retorna o índice do elemento.

Anotações

- *int lastIndexOf*(*Object element*).
  Retorna o índice da ultima ocorrência do elemento especificado.

Lembre se que em uma *List* podemos ter duplicatas.

- *Object remove*(*int index*).
  Remove o elemento que se refere ao índice especificado.

- *Object set*(*int index*, *Object element*).
  Substitui o elemento que estiver no índice especificado.

A *interface List* também prove recursos para trabalhar com subconjuntos de *collection*, bem como executar operação em intervalos de índices.

- *ListIterator listIterator*().
- *ListIterator listIterator*(*int startIndex*).
- *List subList*(*int fromIndex*, *int toIndex*).

Agora vejamos suas aplicações:

- *ListIterator listIterator*().
  Retorna o tipo *ListIterator* da *List* atual.

- *ListIterator listIterator*(*int startIndex*).
  Retorna o tipo *ListIterator* da *List* atual a partir do índice especificado.

- *List subList*(*int fromIndex, int toIndex*).
  Retorna um *List* contendo os elementos que estiverem dentro do intervalo de índices especificados.

Anotações

Exemplo - Usando *List*

```
import java.util.*;

public class ListExample {

    public static void main(String args[]) {
            List list = new ArrayList();
            list.add("1 Pato");
            list.add("2 Cachorro");
            list.add("3 Gato");
            list.add("4 Porco");
            list.add("5 Vaca");

            System.out.println(list); // Imprimindo a List
            System.out.println("1: " + list.get(1));
            System.out.println("0: " + list.get(0));

            // Usando LinkedList.
            LinkedList linkedList = new LinkedList();

            // Atribuindo os mesmos elementos de List.
            linkedList.addFirst("1 Pato");
            linkedList.addFirst("2 Cachorro");
            linkedList.addFirst("3 Gato");
            linkedList.addFirst("4 Porco");
            linkedList.addFirst("5 Vaca");
            System.out.println(linkedList); // Imprimindo a LinkedList.

            linkedList.removeLast();
            linkedList.removeLast();
            // LinkedList sem os dois últimos elementos.
            System.out.println(linkedList);
    }
}
```

Anotações

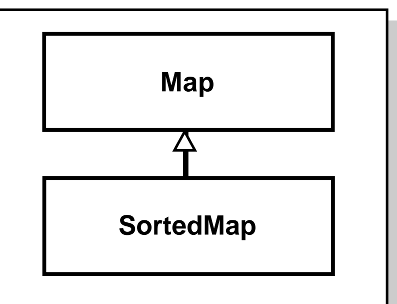
Saída:

```
[1 Pato, 2 Cachorro, 3 Gato, 4 Porco, 5 Vaca]
1: 2 Cachorro
0: 1 Pato
[5 Vaca, 4 Porco, 3 Gato, 2 Cachorro, 1 Pato]
[5 Vaca, 4 Porco, 3 Gato]
```

Anotações

Map interface

- A *interface Map* não *extends* a *interface Collection*.
- Esta *interface* inicia sua própria hierarquia de classes.
- O objetivo desta nova hierarquia é manter associações de *key-value*.
- Descreve um mapeamento de *keys to values* e, sem duplicatas por definição.
- Ambos, *key* e *value* podem seu *null*.
- A *interface Map* proíbe *keys* duplicatas.
- Os *values* podem estar duplicados.
- Cada *key* pode sofrer um map para apenas um *value*.



A *interface Map* não *extends* a *interface Collection*. Ao invés disto, a esta *interface* inicia sua própria hierarquia de classes. O objetivo desta nova hierarquia é manter associações de *key-value*. Esta interface descreve um mapeamento de *keys to values* e, sem duplicatas por definição.

Anotações

Os Métodos de *Map*

- **Métodos de adição e remoção:**
  - *Object put*(*Object key, Object value*).
  - *Object remove*(*Object key*).
  - *void putAll*(*Map mapping*).
  - *void clear*().
- **Métodos para checagem de conteúdo:**
  - *Object get*(*Object key*).
  - *boolean containsKey*(*Object key*).
  - *boolean containsValue*(*Object value*).
  - *int size*().
  - *boolean isEmpty*().
- **Métodos abaixo executam operações em grupos de *keys* ou *values*:**
  - *public Set keySet*().
  - *public Collection values*().
  - *public Set entrySet*().

As operações habilitam você a fazer *add* e *remove key-value pairs* de um *Map*. Ambos, *key* e *value* podem seu *null*.

As classes que implementam a *interface Map* são: *AbstractMap*, *Attributes*, *HashMap*, *Hashtable*, *IdentityHashMap*, *RenderingHints*, *TreeMap e WeakHashMap*.

Anotações

Importantes Informações sobre a *Interface Map*

- A *interface Map* proíbe *keys* duplicatas.
- Os *values* podem estar duplicados.
- Cada *key* pode sofrer um map para apenas um *value*.

Métodos de adição e remoção:

- *Object put*(*Object key, Object value*).
- *Object remove*(*Object key*).
- *void putAll*(*Map mapping*).
- *void clear*().

Métodos para checagem de conteúdo:

- *Object get*(*Object key*).
- *boolean containsKey*(*Object key*).
- *boolean containsValue*(*Object value*).
- *int size*().
- *boolean isEmpty*().

Métodos abaixo executam operações em grupos de *keys* ou *values*:

- *public Set keySet*().
- *public Collection values*().
- *public Set entrySet*().

Anotações

Descrições:

- *Object put*(*Object key, Object value*).
  Associa a *key* especificada com o argumento *value*. Caso já exista esta *key* no objeto *map*, o valor antigo será substituído. Retorna o *value* associado a *key*, ou *null* se a *key* estiver com *value null*.

- *Object remove*(*Object key*).
  Remove a *key* especificada. Retorna o valor associado a *key*, ou *null* se a *key* nao existir.

- *void putAll*(*Map mapping*).
  Copia todos os *mappings* do *map* especificado para o *map* atual.

- *void clear*().
  Remove todos os *mappings*.

As operações que habilitam você fazer checagem no conteúdo de um *Map* são:

- *Object get*(*Object key*).
  Retorna o valor associado a *key* especificada.

- *boolean containsKey*(*Object key*).
  Retorna *true* caso no *map* houver a *key*.

- *boolean containsValue*(*Object value*).
  Retorna *true* caso exista o *value* especificado para uma ou mais *key*.

- *int size*().
  Retorna o número de *key-value* no *map*.

Anotações

- *boolean isEmpty*().

  Retorna verdadeiro se não houver *key-value mapping*.

Os métodos abaixo executam operações em grupos de *keys* ou *values* como um *collection*.

- *public Set keySet*().
  Retorna um *set view das keys* contidas no *map* atual. Qualquer alteração no *map* se refletira no *set* e vice versa.

- *public Collection values*().
  Retorna uma *Collection* dos *mappings* contidos no *map* atual. Qualquer alteração no *map* se refletira na *Collection* e vice-versa.

- *public Set entrySet*().
  Retorna um *set view* dos *mappings* contidos no *map* atual. Qualquer alteração no *map* se refletira no *set* e vice-versa.

Exemplo - *Map*

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

public class MapExample {

   public static void main(String args[]) throws java.io.IOException{
          Map map = new HashMap();

          String key = null;
          String value = null;
```

Anotações

```
            System.out.println("\n Palavra digitada:  " + args[0] + '\n');
            for(int i=0; i<3; i++){

                switch(i){
                    case 0:
                        key = "length";
                        value=args[0].valueOf(args[0].length());
                        break;
                    case 1:
                        = "substring(0,1)";
                        value = args[0].substring(0,1);
                        break;
                    case 2:
                        key = "toUpperCase";
                        value = args[0].toUpperCase();
                }
                map.put(key, value);
            }

            System.out.println(map);

            System.out.println("\n SortedMap: \n");

            Map sortedMap = new TreeMap(map);
            System.out.println(sortedMap);
        }
    }
```

Anotações

Saída:

```
C:\test>java MapExample Objeto

Palavra digitada:  Objeto

{toUpperCase=OBJETO, substring(0,1)=O, length=6}

 SortedMap:

{length=6, substring(0,1)=O, toUpperCase=OBJETO}
```

Anotações

Laboratório 9

Capítulo 10: *Java Collections Framework*

Concluir o(s) exercício(s) proposto(s) pelo instrutor. O instrutor lhe apresentará as instruções para a conclusão do mesmo.

Laboratório 9 - Capítulo 10

1) Compile e rode o exemplo da página 240.

Anotações

2) Compile e rode o exemplo da página 244.

Anotações

3) Compile e rode o exemplo da página 250.

Exercício Opcional (somente se houver tempo).
- Tempo previsto 3 horas:
A *interface Collection* também suporta as operações de *query*:

- *int size*()
Retorna o tamanho da *collection*

- *boolean isEmpty*()
Testa se a *collection* esta vazia

- *boolean contains*(*Object element*)
Verifica se um determinado elemento se encontra na *collection*

- *Iterator iterator*()
Retorna um objeto do tipo *Iterator.*

Anotações

4) Estenda a classe *java.util.AbstractCollection* e faça sua própria classe de *collection*, faça *overriding* apenas nos métodos Métodos Básicos da *interface Collection* que estão na página 230. Para facilitar um pouco a tarefa, não há necessidade de diminuir a capacidade da *collection* após uma remoção de elemento.

Métodos Básicos da *interface Collection*:

Os métodos para adição e remoção de elementos são:

- *boolean add*(*Object element*)
Adiciona elementos

- *boolean remove*(*Object element*)
Remove elementos

Anotações

Anotações

Capítulo 11:

Threads

Threads

- O que é uma *Thread*.
- Sistemas *multithreads*.
- **API** dedicada a *Threads*.
- Itens do módulo:
  - Como criar *Threads*.
  - Ciclo de vida de uma *Thread Java*.
  - Prioridades dos *Threads*.
  - Sincronização de *Threads*.
  - Agrupamento de *Threads*.

Um *Thread* é uma linha de execução. Um sistema pode ser composto por muitas *Threads*, cada uma delas com uma função específica.

Com *Threads* é possível construir sistemas multitarefa, ou seja, sistemas multithread, onde podemos realizar varias tarefas simultaneamente.

Java possui ampla **API** dedicada a *Threads*, que irá permitir a construção de grandes e poderosos sistemas, que necessitem de vários tipos de tarefas executando ao mesmo tempo, vários trabalhos distintos.

Anotações

Veremos neste módulo os seguintes itens:

- Como criar *Threads*.

- Ciclo de vida de uma *Thread Java*.

- Prioridades dos *Threads*.

- Sincronização de *Threads*.

- Agrupamento de *Threads*.

Anotações

Como Criar *Threads*

- Existem duas formas distintas de criação de *Threads*:

1. Estender a classe *java.lang.Thread*.
2. Implementar a *interface java.lang.Runnable*.

Existem duas formas distintas de criação de *Threads*:

1. Extender a classe *java.lang.Thread*

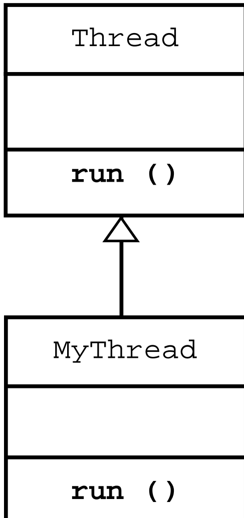
2. Implementar a *interface java.lang.Runnable*

Como podemos observar a classe *Thread* e a *interface Runnable* encontram-se no pacote *java.lang*. Desta forma não existe a obrigatoriedade de fazer qualquer tipo de *import*, para nenhumas das duas formas de se criar *Threads*.

Anotações

Estendendo a Classe *Thread*

```
class MyThread extends Thread {
    public void run ( )  {
        //. . . . . . .
    }
}
Thread x = new MyThread();
```



Anotações

Para criar uma nova *Thread*, precisamos fazer uma subclasse de *java.lang.Thread* e personalizar esta classe fazendo o *overriding* do método *public void run*().

- O método *public void run*() está vazio na **API**, somente existe para sofrer *overriding*.
- Método *public void run*() é onde são disparadas as ações dos *Threads*.
- Para iniciar a execução de uma *Thread*, basta chamar o método *start*().

Thread - Exemplo I

```
class MyThread extends Thread {
    private String nome, msg;

    public MyThread(String nome, String msg) {
        this.nome = nome;
        this.msg = msg;
    }
    public void run() {
        System.out.println(nome + " iniciou !");
        for (int i = 0; i < 4; i++) {
                System.out.println(nome + " disse: " + msg);
                try {
                        Thread.sleep(3000);
                }
                catch (InterruptedException ie) { }
        }//end for
        System.out.println(nome + " terminou de executar");
    }
}
```

Anotações

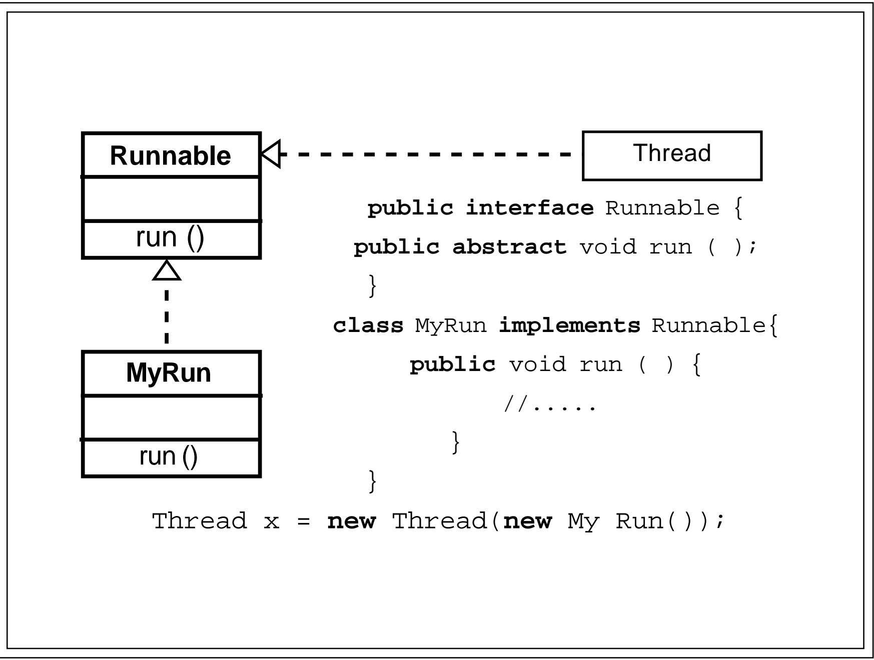
```
public class StartThread {
    public static void main(String[] args) {
        MyThread t1 = new MyThread("thread1", "ping");
        MyThread t2 = new MyThread("thread2", "pong");
        t1.start();
        t2.start();
    }
}
```

Saída:

```
thread1 iniciou !
thread1 disse: ping
thread2 iniciou !
thread2 disse: pong
thread1 disse: ping
thread2 disse: pong
thread1 disse: ping
thread2 disse: pong
thread1 disse: ping
thread2 disse: pong
thread1 terminou de executar
thread2 terminou de executar
```

Anotações

Implementando a *Interface Runnable*



- A criação de *Threads* através da implementação da *interface Runnable* é muito interessante, já que libera a classe para a mesma poder estender outra se necessária.
- Temos construtores na classe *Thread*, que recebem objetos *Runnable*, para criação de novas *Threads*:

  - *Thread*(*Runnable target*).

  - *Thread*(*Runnable target, String name*).
- A *interface Runnable* possui o método *run*(), o qual devera sofrer *overriding*.

Anotações

Thread - Exemplo II

```
class MyClass implements Runnable {
    private String nome;
    private A sharedObj;

    public MyClass(String nome, A sharedObj) {
        this.nome = nome;
        this.sharedObj = sharedObj;
    }
    public void run() {
        System.out.println(nome + " iniciou !");
        for (int i = 0; i < 4; i++) {
                System.out.println(nome + " disse: " + sharedObj.getValue());

                if (i==2) {
                        sharedObj.setValue("mudou a string!");
                }
                    try {
                        Thread.sleep(3000);
                }
                catch(InterruptedException ie) { }
        }//end for
        System.out.println(nome + " terminou de executar");
    }
}
class A {
    private String value;

    public A(String value) {
        this.value = value;
    }
    public String getValue() {
        return value;
    }
```

Anotações

```
    public void setValue(String newValue) {
        this.value = newValue;
    }
}
public class StartThread2 {
    public static void main(String[] args) {
        A sharedObj = new A("algum valor");
        Thread t1 = new Thread(new MyClass("thread1", sharedObj));
        Thread t2 = new Thread(new MyClass("thread2", sharedObj));
        t1.start();
        t2.start();
    }
}
```

Saída:

```
thread1 iniciou !
thread1 disse: algum valor
thread2 iniciou !
thread2 disse: algum valor
thread1 disse: algum valor
thread2 disse: algum valor
thread1 disse: algum valor
thread2 disse: mudou a string!
thread1 disse: mudou a string!
thread2 disse: mudou a string!
thread1 terminou de executar
thread2 terminou de executar
```

Anotações

O Ciclo de Vida de uma *Thread*

new thread ()
início
start () chama run ( )
Created
start()
Alive
stop(), ou
run () returns
stop()
Death
fim

O método *start* cria os recursos necessários para que a *Thread* possa rodar e, inicia a execução da mesma através da chamada do método *run*().

A *Thread* deixa entra em *Not Runnable*, em uma destas situações:

- Método *sleep*() é chamado.
- Método *wait*() é chamado.
- A *Thread* é bloqueada em **I/O**.
- A T*hread* morre sozinha quando o método *run*() termina sua execução.
- Não é possível "*ressuscitar*" uma *Thread*.

Anotações

Escalonamento da JVM

- Escalonamento preemptivo
- Escalonamento circular (*Round Robin*)

A **JVM** e o escalonamento.

A **JVM** trabalha com escalonamento preemptivo circular, os conceitos serão abordados abaixo:

Escalonamento Preemptivo

O escalonamento é dito preemptivo quando o sistema pode interromper um processo que está sendo executado. Em geral isso ocorre para que outro processo entre na área de execução.

Anotações

A principal função é dar prioridade a processos mais importantes, e também proporcionar melhores tempos de resposta em sistemas de tempo compartilhado.

Os algoritmos de preempção levam em consideração critérios que têm como objetivo evitar a sobrecarga do sistema causada pela interrupção da execução de processos.

Escalonamento Circular

Também chamado de *Round Robin*, é implementado por meio de um algoritmo projetado exclusivamente para sistemas de tempo compartilhado. O algoritmo estabelece um tempo limite de processamento, chamado de *time-slice*, ou quantum, e, quando este tempo termina, o processo deixa a **CPU** mesmo sem ter terminado o processamento.

Este mecanismo recebe o nome de Preempção por Tempo.

Anotações

Prioridades de *Thread*

- Concorrência de *Threads* controladas por prioridades.
- *Scheduling* ocorre em relação aos *threads* que estão em *runnable status*.
- O sistema escolhe a *Thread* de maior prioridade.
- *setPriority*().
- **MAX_PRIORITY** - A máxima prioridade que uma *Thread* pode ter.
- **MIN_PRIORITY** - A mínima prioridade que uma *Thread* pode ter.
- **NORM_PRIORITY** - Esta é a prioridade padrão de uma *Thread*.

A *Java Virtual Machine* (**JVM**), na verdade simula uma situação onde varias coisas acontecem ao mesmo tempo, quando na verdade uma coisa é processada de cada vez. Para que isso ocorra corretamente, a **JVM** prove uma concorrência de *Threads* controladas por prioridades.

A execução de múltiplos *threads* em uma única **CPU** é chamada de *scheduling*.

A **JVM** prove um algoritmo simples de *scheduling*, onde o *scheduling* ocorre em relação aos *threads* que estão em *runnable status*.

Anotações

Durante o *runtime*, o sistema escolhe a *thread* de maior prioridade para colocá-la em execução.

Quando uma *thread Java* é criada, esta irá herdar a prioridade da *thread* que a criou.

Você pode alterar a prioridade das *threads* a qualquer momento, a **JVM** prove mecanismos para isso, para configurar uma nova prioridade para uma *thread* basta usa o método *setPriority*().

Usamos um *int* para definir a prioridade. Na classe *Thread* temos as seguintes constantes:

MAX_PRIORITY
A máxima prioridade que uma *thread* pode ter.

MIN_PRIORITY
A mínima prioridade que uma *thread* pode ter.

NORM_PRIORITY
Esta é a prioridade padrão de uma *thread*.

Exemplo - Prioridades de *Thread*

```
class PriorityTest {
   public static void main (String args[]) {
      Thread t1 = new RandomPrintString("Primeira");
      Thread t2 = new RandomPrintString("Segunda");
      Thread t3 = new RandomPrintString("Terceira");
```

Anotações

```
        t1.setPriority(4);
        t2.setPriority(5);
        t3.setPriority(6);
        t1.start();
        t2.start();
        t3.start();
    }
}
class RandomPrintString extends Thread {
    public RandomPrintString(String str) {
            super(str);
    }
    public void run() {
            for (int i = 0; i < 3; i++) {
            System.out.println(getName());
            //sleep((int)(Math.random() * 1000));
            }
        //System.out.println(Thread.MIN_PRIORITY); // 1
        //System.out.println(Thread.NORM_PRIORITY); // 5
        //System.out.println(Thread.MAX_PRIORITY); // 10
    }
}
```

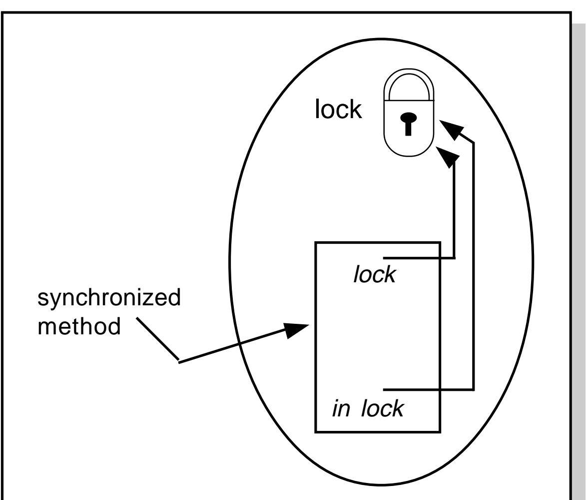
Saída:

```
Terceira
Terceira
Terceira
Segunda
Segunda
Segunda
Primeira
Primeira
Primeira
```

Anotações

Sincronização de *Threads*

- Porque sincronizar um método.
- *A keyword synchronized*.
- O sistema de *flag lock*.



Anotações

Podem existir situações em que diferentes threads compartilham dados.

Se duas *threads* executam dados que modificam o *state* de um objeto, então o método deveria ser declarado *synchronized*, pois isso irá garantir que somente uma *thread* executasse o método até o fim, sem que seja interrompido.

***Synchronized* - Exemplo:**

```
public synchronized void updateRecord() {
    // critical code place …
}
```

No exemplo acima, somente uma única *thread* poderá estar no corpo deste método. A segunda será bloqueada, até que a primeira chame *return*, ou *wait*() seja chamado dentro do método sincronizado.

Se você não precisa proteger todo o método, pode proteger apenas um objeto:

```
public void test() {
    // synchronized (this) { //critical code place …
    }
}
```

Anotações

Produtor e Consumidor

O Problema:

```
public class TestSync{
    int index;
    float [] data = new float[5];
    public float get() {
            index—;
            return this.data[index];
    }

    public void put(int value){
            this.data[index] = value;
            index++;
    }
}
```

O problema do programa acima se deve ao fato de que este código compartilha dados e, neste caso, deve-se proteger os dados. Devemos então sincronizar os métodos, pois desta forma não correremos o risco de que uma *thread* seja colocada on *hold* no meio da execução de um método, no exemplo acima, isto pode acontecer.

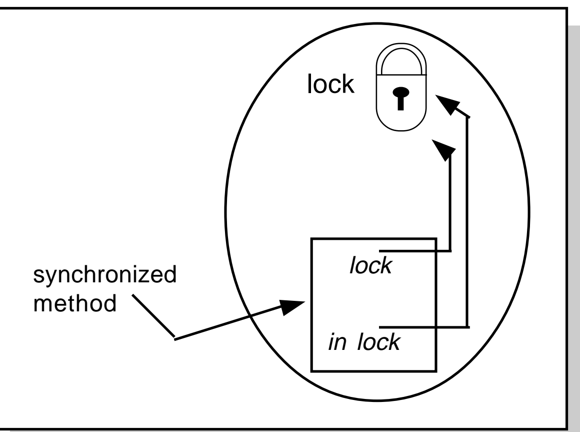
Imagine que uma *thread* **A** esteja executado a primeira linha do método *put*(), e ela seja colocada para dormir, daí vem a *thread* **B** e começa a executar o método *get*(), causando assim uma enorme inconsistência de dados.

Anotações

Produtor e Consumidor

A Solução:

```
public class TestSync{
    int index;
    float [] data = new float[5];

    public synchronized float get() {
            index--;
            return this.data[index];
    }
public synchronized void put(int value){
            this.data[index] = value;
            index++;
            }
}
```

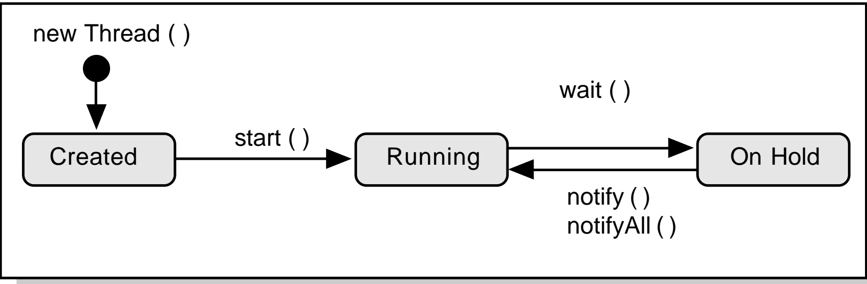


Agora sim temos um código confiável, pois foi feita a sincronização dos métodos. Desta forma, a *thread* que estiver executando um método *synchronized* irá tomar para si o que chamamos de *flag* de travamento (*lock flag*), quando uma outra *thread* vier executar o método, ira tentar pegar a *lock flag* e, não a encontrara, desta forma ela ira novamente para fila.

Anotações

A classe *Object* possui métodos ligados à sincronização de *Threads*:

- *wait*().
- *wait*(*long timeout*).
- *notify*().
- *notifyAll*().



Anotações

A classe *Object* possui métodos ligados à sincronização de *Threads*:

- *wait*().
- *wait(long timeout*).
- *notify*().
- *notifyAll*().

▲ Uma *thread* pode chamar *wait*() dentro de um método sincronizado. Um *timeout* pode ser provido. Se ele estiver faltando, ou for igual a zero, então a *thread* irá esperar ate que *notify*() ou *notifyAll*()seja chamado.

▲ *wait*() é chamado por uma *thread*, isso irá causar um *lock*, e esta *thread* ficara esperando eternamente, ate que você a libere.

▲ *wait*(*long timeout*) é chamado quando se deseja que a *thread* seja liberada em um determinado período de tempo caso não seja chamado *notify*()ou *notityAll*() por algum motivo.

▲ *notify*() e *notifyAll*() são somente chamados de métodos sincronizados. Um ou todos os threads são notificados respectivamente.

▲ *notifyAll*() é a forma mais segura de se trabalhar, pois não é possível escolher uma *thread* em especifico para notificá-la, então se muitas *threads* estiverem em wait, este método ira notificar todas.

Anotações

Exemplo sincronização de *Threads*:

```
public synchronized int get() {
    while (available == false) {
            try { //wait for Producer to put
            //value wait();
            } catch (InterruptedException e){ }

    }available = false;  //notify Producer that value has
                          //been retrieved notifyAll();
return contents;
}
public synchronized void put(int value){
    while (available == true) {
    try {   //wait for Consumer to get
            //value wait();
    } catch (InterruptedException e){ }

    }contents = value;
    available = true;  //notify Consumer that value has
                        //been set notifyAll();
}
```

A classe *thread* fornece o método *join*(), que abilita o objeto a esperar até que a outra *thread* termine.

```
public void myMethod() {
// Faz algo...
// Não pode prosseguir até que a outra thread termine
otherThread.join();
// Continua executando...
}
```

- Este método é equivalente a *waitFor*() *of java.lang.Process*.

Anotações

Agrupamento de *Threads*

- O que é um *Thread Groups*.
- A classe *java.lang.ThreadGroup*.
- *ThreadGroup main*.
- Uma nova *thread* em um *Thread Group*.
- Uma *thread* não pode ser movida para um novo grupo.
- *Public ThreadGroup*(*ThreadGroup parent*, *String name*).
- *ThreadGroup group = myThread.getThreadGroup*().

- *Thread groups* prove um mecanismo para coletar múltiplos *threads* dentro de um simples objeto e manipular todas estas *threads* de uma só vez.
- Java *thread groups* são implementadas pela classe *java.lang.ThreadGroup*.
- Quando uma aplicação java inicia, o *runtime java* cria *ThreadGroup* chamado *main*. A menos que seja especificado, todas as novas *threads* que você criar serão membros de *main thread group*.
- Para colocar uma nova *thread* em um *thread group* o group deveria ser explicitamente especificado quando a *Thread* é criada.

Anotações

- *public Thread*(*ThreadGroup group*, *Runnable runnable*).
- *public Thread*(*ThreadGroup group*, *String name*).
- *public Thread*(*ThreadGroup group*, *Runnable runnable*, *String name*).

• Uma *thread* não pode ser movida para um novo grupo depois que a *thread* já esta criada.

• Uma pode também conter o outro *ThreadGroups* permitindo a criação de uma hierarquia das *threads* e *thread groups*.

- *public ThreadGroup*(*ThreadGroup parent, String name*).

• Para obter a *thread group* de uma *thread use getThreadGroup* da classe *Thread*.

• *ThreadGroup group* = *myThread.getThreadGroup*().

Anotações

Exemplo - *ThreadGroup*:

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

       ThreadGroup myGroup = new ThreadGroup(“Meu grupo”);
       Thread myGroupThread1;
       myGroupThread1 = new Thread(myGroup,”Thread membro 1");
       Thread myGroupThread2;
       myGroupThread2 = new Thread(myGroup,”Thread membro 2");

       // configurando prioridade normal (5) para o grupo myGroup
       myGroup.setMaxPriority(Thread.NORM_PRIORITY);

       System.out.println(“Group’s priority = “ + myGroup.getMaxPriority());

       System.out.println(“Numero de Threads no grupo = “ +
       myGroup.activeCount());

       System.out.println(“Grupo Pai = “ + myGroup.getParent().getName());

       // Imprime as informações sobre o grupo
       myGroup.list();

   }
}
```

Saída:

```
Group’s priority = 5
Numero de Threads no grupo = 2
Grupo Pai = main
java.lang.ThreadGroup[name=Meu grupo,maxpri=5]
   Thread[Thread membro 1,5,Meu grupo]
   Thread[Thread membro 2,5,Meu grupo]
```

Anotações

Laboratório 10:

Capítulo 11:
Threads

Concluir o(s) exercício(s) proposto(s) pelo instrutor. O instrutor lhe apresentará as instruções para a conclusão do mesmo.

Laboratório 10 - Capítulo 11

1) Compile e rode o exemplo da página 264, que mostra a criação de *threads* com a opção de estender a classe *Thread*.

Anotações

2) Compile e rode o exemplo da página 267, que mostra a criação de *threads* com a opção de implementar a classe *Runnable*.

Anotações

3) Compile e rode o exemplo da página 273, sobre priorização de *threads*.

Anotações

4) Compile e rode o exemplo da página 277, sobre sincronização de *threads*, no caso de produtor e consumidor.

Anotações

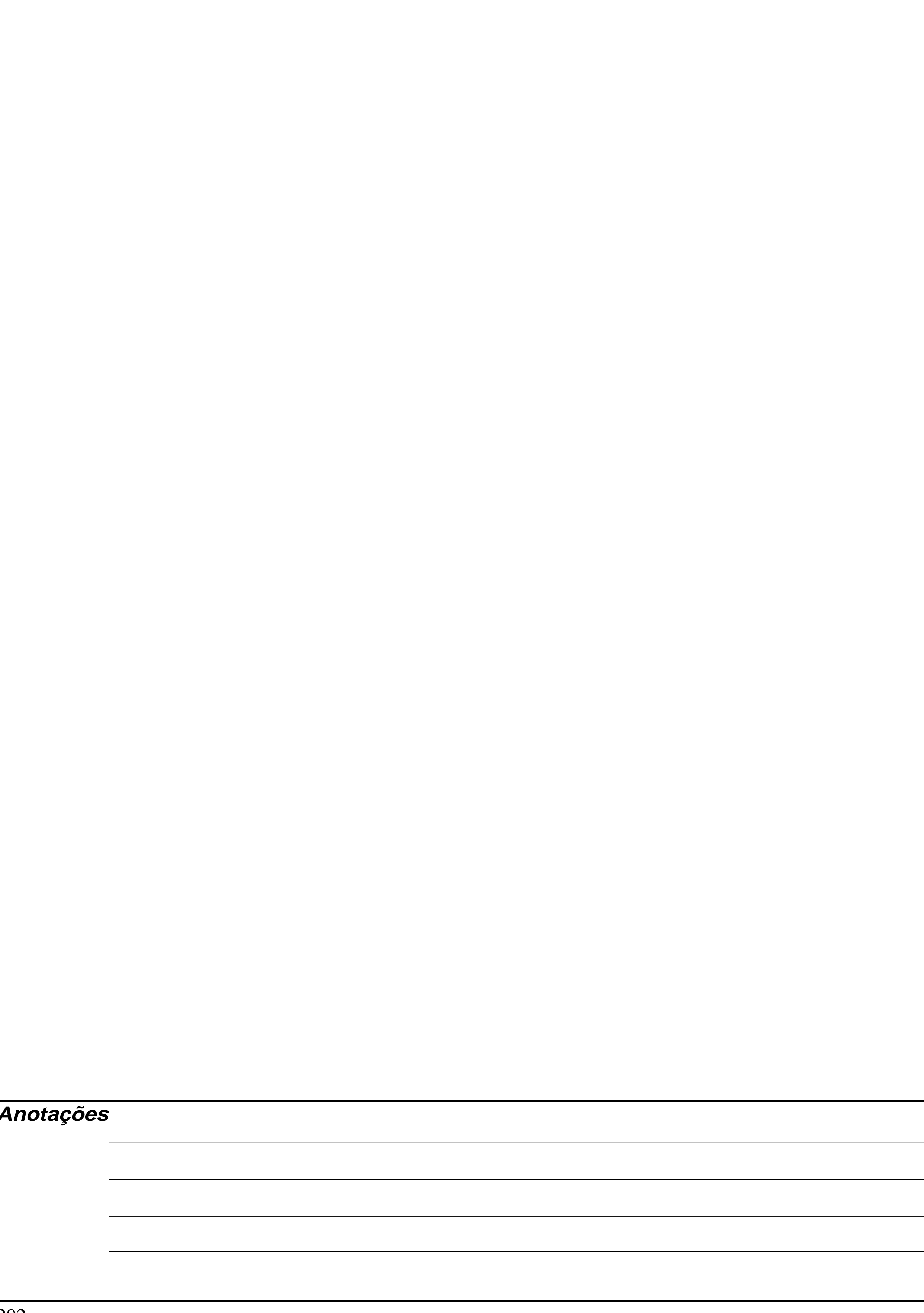
5) Compile e rode o exemplo da página 284, que mostra a criação de *ThreadGroup* e implementa a priorização em grupos de *threads*.

Anotações

6) Crie um programa *multithread* que recebe dois nomes pela linha de comando, cria uma *thread* para cada nome e coloque ambas para dormir comum tempo aleatório, a *Thread* que acordar primeiro é a vencedora.

Dica
Use o método *nextInt*(*int*) da classe *java.util.Random*(). Mas primeiramente você deve ler o que a **API** diz sobre este método, para melhor implementá-lo.

Anotações



Anotações

Capítulo 12:

Aplicações Gráficas com AWT

Aplicações Gráficas com AWT

- O pacote *java.awt*.
- Uso das definições de componentes gráfico do gerenciador de janelas que esta rodando.
- Características **AWT**:
  - Grande variedade de componentes de *interface* para o usuário;
  - Delegação de eventos;
  - Gerenciadores de *layout* de janelas flexíveis, de auto manipulação de tamanhos de componentes, ajustes e auto posicionamento dos mesmos.

Java possui uma grande quantidade de componentes preparados para a construção de sistemas com interfaces gráficas com o usuário (*Graphical User Interface* o **GUI**).

Uma característica muito interessante de sistemas feitos com **AWT** é que este usa as definições de componentes gráfico do gerenciador de janelas que esta rodando. Isso implica em uma tela aparecer de uma forma no sistema operacional *Windows* e de forma diferente no *Linux*.

Anotações

O **AWT** fornece um conjunto de elementos de *interface* gráfica padrão (botões, janelas, menus, campos de edição, campos de seleção e outros) incluindo o sistema de tratamento de eventos que ocorrem nestes elementos e outras classes complementares.

Características AWT:

- Grande variedade de componentes de *interface* para o usuário.
- Delegação de eventos.
- Gerenciadores de *layout* de janelas flexíveis, de auto manipulação de tamanhos de componentes, ajustes e auto posicionamento dos mesmos.

Anotações

Dentro do pacote java.awt podemos encontrar os seguintes componentes:

- **Labels (rótulos)** - classe *java.awt.Label*.
- **Botões** - classe *java.awt.Button*.
- **Campos de texto** - classe *java.awt.TextField*.
- **Áreas de texto** - classe *java.awt.TextArea*.
- **Caixas de Verificação e Radio *Buttons*** - classe *java.awt.CheckBox*.
- **ComboBoxes** - classe *java.awt.Choice*.
- **ListBoxes** - classe *java.awt.List*.
- **Barras de Rolagem** - classe *java.awt.ScrollBar.*
- **Canvas (Telas de Pintura)** - classe *java.awt.Canvas*.
- **Frames** - classe *java.awt.Frame*.
- **Diálogos** - classe *java.awt.Dialog*.
- **Painéis** - classe *java.awt.Panel*.
- **Menus** - classe *java.awt.MenuBar, java.awt.Menu, java.awt.MenuItem.*
- **Cores** - classe *java.awt.Color.*
- **Fontes** - classe *java.awt.Font*.
- **Imagens** - classe *java.awt.Image*.
- **Cursores** - classe *java.awt.Cursor*.

Anotações

Abaixo temos as principais características dos principais elementos de **AWT** e os métodos mais utilizados de cada classe. A **API** *Java* fornece informações mais aprofundadas a respeito destes e de outros componentes.

Label
setText(*String l*).
Define o texto do rótulo.

String getText().
Retorna o texto do rótulo.

Button
setLabel(*String l*).
Define o texto do botão.

String getLabel().
Retorna o texto do botão.

TextField
setText(*String t*).
Define o texto do campo.

String getText().
Retorna o texto do campo.

TextArea
setText(*String t*).
Define o texto da área.

String getText().
Retorna o texto da área.

setEditable(*boolean b*).
Define se a área pode ser editada ou não.

appendText(*String s*).
Adiciona a *string* ao final do texto.

Anotações

Checkbox

setLabel(*String l*).
Adiciona a *string* ao final do texto.

String getLabel().
Retorna o texto do *checkbox*.

setState(*boolena b*).
Define o estado do *checkbox true* = *on*, *false* = *off*.

boolean getState().
Retorna o estado do *checkbox*.

Choice

addItem(*String i*).
Adiciona um item ao *choice*.

String getItem(*int pos*).
Retorna o item da posição pos.

int getItemCount().
Retorna o número de itens no *choice*.

int getSelectedIndex().
Retorna a posição do item selecionado.

String getSelectedIntem().
Retorna o item selecionado como um *String*.

removeAll().
Remove todos os itens.

List

addItem(String i).
Adiciona um item a lista.

String getItem(*int pos*).
Retorna o item da posição pos.

Anotações

int getItemCount().
Retorna o número de itens na lista.
int getSelectedIndex().
Retorna a posição do item selecionado.

String getSelectedIntem().
Retorna o item selecionado.

removeAll().
Remove todos os itens da lista.

Frame
setTitle(*String t*).
Define o título do *frame*.

setResizable(*boolean b*).
Define se o *frame* pode ser redimensionado ou não.

setIconImage(*Image img).*
Define o icone para o *frame*.

setMenubar(*Menubar mb*).
Define a barra de menu para o *frame*.

Dialog
setTitle(*String t*).
Define o titulo do dialogo.

setResizable(boolean b).
Define se o diálogo pode ser redimensionado ou não.

setModal(*boolean b*).
Define se a janela é modal ou não.

Anotações

A Classe AWT *Compoment*

- A classe *java.awt.Component* fornece um conjunto de operações padrão.
- A classe *Component* é a superclasse da maioria dos elementos de *interface* da **AWT**.
- Métodos comuns:
  - *setBounds*(*int x, int y, int width, int height*).
  - *setLocation*(*int x, int y*).
  - *setSize*(*int width, int height*).
  - *setEnable*(*boolean b*).
  - *setVisible(boolean b*).
  - *setFont*(*Font f*).
  - *setBackground*(*Color c*).
  - *setForeground*(*Color c*).

Hierarquia da classe *java.awt.Component.*

```
java.lang.Object
 |
 +—java.awt.Component
```

A classe *java.awt.Component* fornece um conjunto de operações padrão para a quase todos os componentes **AWT**. A classe Component é a superclasse da maioria dos elementos de *interface* da **AWT**.

Anotações

Descrições dos métodos mais comuns:

setBounds(*int x, int y, int width, int height*).
Define a posição x, y, a largura e altura do componente.

setLocation(*int x, int y*).
Define a posição x, y do componente.

setSize(*int width, int height*).
Define a largura e altura do componente.

setEnable(*boolean b*).
Habilita/desabilita o foco para este componente.

setVisible(*boolean b*).
Mostra/esconde o componente.

setFont(*Font f*).
Define a fonte do componente.

setBackground(*Color c*).
Define a cor de fundo do componente.

setForeground(*Color c*).
Define a cor de frente do componente.

Anotações

Java Container

- *Containers* são componentes que podem abrigar outros componentes, inclusive outros *Containers*.
- Os *Containers* herdam da classe *Component*.
- Exemplos:
  - *Panel*;
  - *Applet*;
  - *Frame*;
  - *Dialog*.

Hierarquia da classe *java.awt.Container*.

```
java.lang.Object
  |
  +—java.awt.Component
        |
        +—java.awt.Container
```

Anotações

Normalmente é preciso agrupar os objetos visuais antes de colocá-los na tela. Para agrupar componentes são utilizados *Containers*. *Containers* são componentes que podem abrigar outros componentes, inclusive outros *Containers*. Exemplos: *Panel, Applet, Frame, Dialog*.

Os *Containers* herdam da classe *Component*, mas são usados como objeto para conter outros elementos visuais. O método *add*() adiciona componentes ao *Container* (diferentes opções de utilização estão disponíveis na **API**).

Anotações

Gerenciadores de *Layout*

- Gerenciadores de *Layout* são classes que são responsáveis por gerenciar a organização dos componentes dentro do *container*.
- O método *setLayout*().
- Aplicando o *setLayout*() em:
  - *FlowLayout*;
  - *GridLayout*;
  - *BorderLayout*;
  - *Null*.

A disposição dos componentes adicionados ao *Container* pode depender da ordem em que foram adicionados e do Gerenciador de *Layout* definido para ele.

Gerenciadores de *Layout* são classes que são responsáveis por gerenciar a organização dos componentes dentro do *container*.

O método *setLayout*() da classe *Container* é o método que define o gerenciador de *layout* a ser usado pelo *Container*.

Estudaremos os seguintes gerenciadores de *layout*:

Anotações

FlowLayout

Layout de fluxo. Os componentes são adicionados linha por linha. Acabando o espaço na linha, o componente é adicionado na próxima linha. É o gerenciador de *layout default* para panels e *applets*.

GridLayout

O *container* é dividido em linhas e colunas formando uma grade. Cada componente inserido será colocado em uma célula desta grade, começando a partir da célula superior esquerda.

BorderLayout

Componente é dividido em direções geográficas: leste, oeste, norte, sul e centro. Quando um componente é adicionado deve-se especificar a direção em que será disposto. É o layout default para componentes *Window* (exemplo: *Frame*).

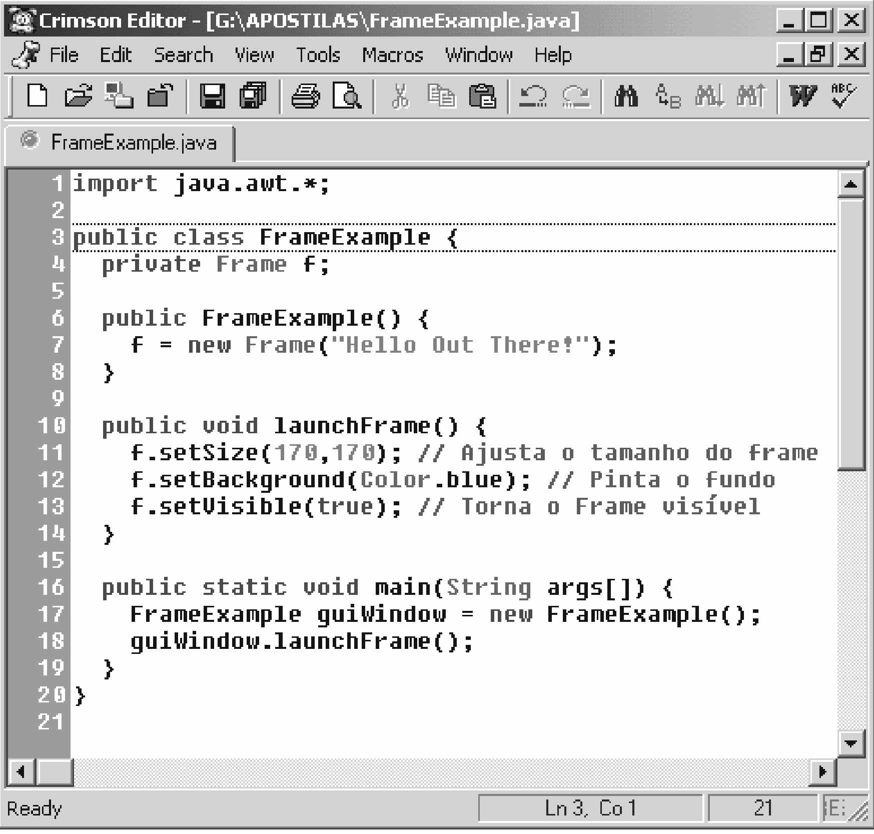
Outra maneira de gerenciar o *layout* de um container é definir o *layout* como "*null*", mover cada componente para uma posição (**x,y**), e ajustar o tamanho (largura, altura) de cada um.

A seguir será apresentada uma seqüência de códigos de estudo para a construção de uma aplicação **GUI**.

Anotações

Frame

O *frame* é o *container* onde iremos adicionar nossos objetos gráficos, ele serve e de moldura principal para nossa aplicação.



Anotações

Saída:

Anotações

Panel

O *Panel* também é um *container*, ele serve para ser colocado dentro de outro *container*. A forma de criação é similar ao *Frame*, a diferença está na linha 19, onde adicionamos o *Panel* dentro do *Frame*.

Crimson Editor - [D:\Documents and Settings\Administrador\Desktop\SL...

File Edit Search View Tools Macros Window Help

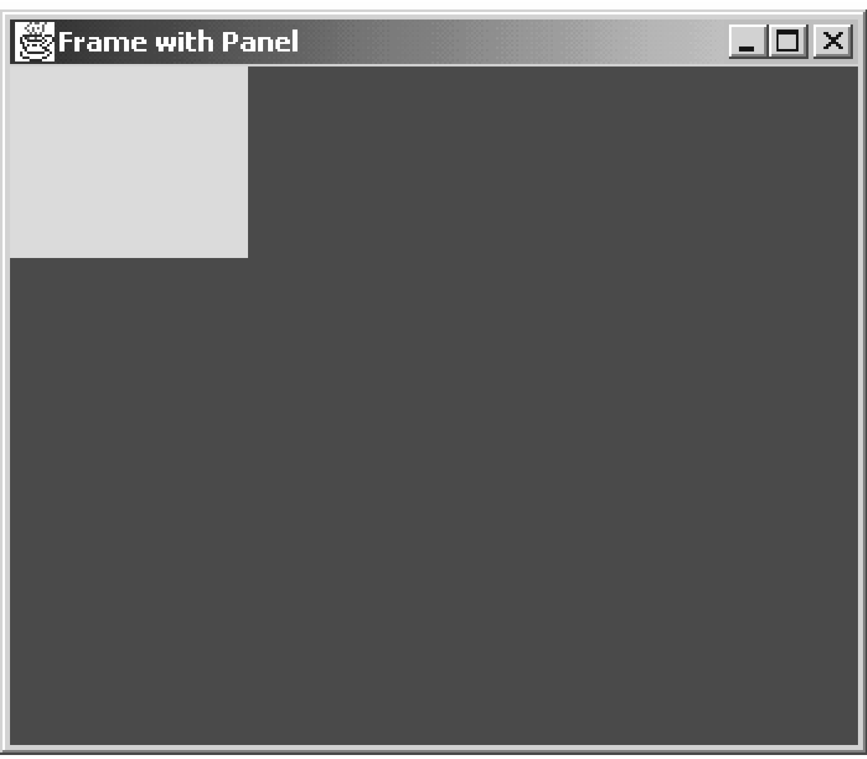
FrameWithPanel.java

```
1  import java.awt.*;
2
3  public class FrameWithPanel {
4    private Frame f;
5
6    public FrameWithPanel(String title) {
7      f = new Frame(title);
8    }
9
10   public void launchFrame() {
11     f.setSize(200,200);
12     f.setBackground(Color.blue);
13     f.setLayout(null);  // Override default layout mgr
14
15     Panel pan = new Panel();
16     pan.setSize(100,100);
17     pan.setBackground(Color.yellow);
18
19     f.add(pan);
20     f.setVisible(true);
21   }
22
23   public static void main(String args[]) {
24     FrameWithPanel guiWindow =
25         new FrameWithPanel("Frame with Panel");
26     guiWindow.launchFrame();
27   }
28 }
29
```

Ready   Ln 15, Co 1   29   REC

Anotações

Saída:



Anotações

List

Veja abaixo como monta uma *list*:

Crimson Editor - [G:\APOSTILAS\awt\examples\MyList.java]

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

public class MyList {
  private Frame f;
  private List list;

  public void go() {
    f = new Frame("List");
    list = new List(4, true);
        list.add("Hello");
        list.add("there");
        list.add("how");
    f.add(list, BorderLayout.CENTER);
    f.pack();
    f.setVisible(true);
  }

  public static void main (String args[]) {
    MyList ml = new MyList();
    ml.go();
  }
}
```

Anotações

Saída:

Anotações

TextArea

Exemplo para uma área de texto:

Crimson Editor - [G:\APOSTILAS\awt\examples\SampleTextArea.java]

File Edit Search View Tools Macros Window Help

SampleTextArea.java

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

public class SampleTextArea {
  private Frame f;
  private TextArea ta;

  public void go() {
    f = new Frame("TextArea");
    ta = new TextArea("Hello!", 4, 30);
    f.add(ta, BorderLayout.CENTER);
    f.pack();
    f.setVisible(true);
  }

  public static void main (String args[]) {
    SampleTextArea sampleTextArea = new SampleTextArea();
    sampleTextArea.go();
  }
}
```

Ready | Ln 4, Co 1 | 21 | REC

Anotações

Saída:

Anotações

TextField

Agora uma área de texto:

Crimson Editor - [G:\APOSTILAS\awt\examples\SampleTextField.j...

File Edit Search View Tools Macros Window Help

SampleTextField.java

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

public class SampleTextField {
  private Frame f;
  private TextField tf;

  public void go() {
    f = new Frame("TextField");
    tf = new TextField("Single Line", 30);
    tf.addKeyListener( new NameHandler() );
    f.add(tf, BorderLayout.CENTER);
    f.pack();
    f.setVisible(true);
  }

  class NameHandler extends KeyAdapter {
    public void keyPressed(KeyEvent e) {
      char c = e.getKeyChar();
      if ( Character.isDigit(c)) {
        e.consume();
      }
    }
  }

  public static void main (String args[]) {
    SampleTextField txtf = new SampleTextField();
    txtf.go();
  }
}
```

Ready | Ln 4, Co 1 | 31

Anotações

Saída:

Anotações

Anotações

Laboratório 11:

Capítulo 12: Aplicações Gráficas com AWT

Concluir o(s) exercício(s) proposto(s) pelo instrutor. O instrutor lhe apresentará as instruções para a conclusão do mesmo.

Laboratório 11 - Capítulo 12

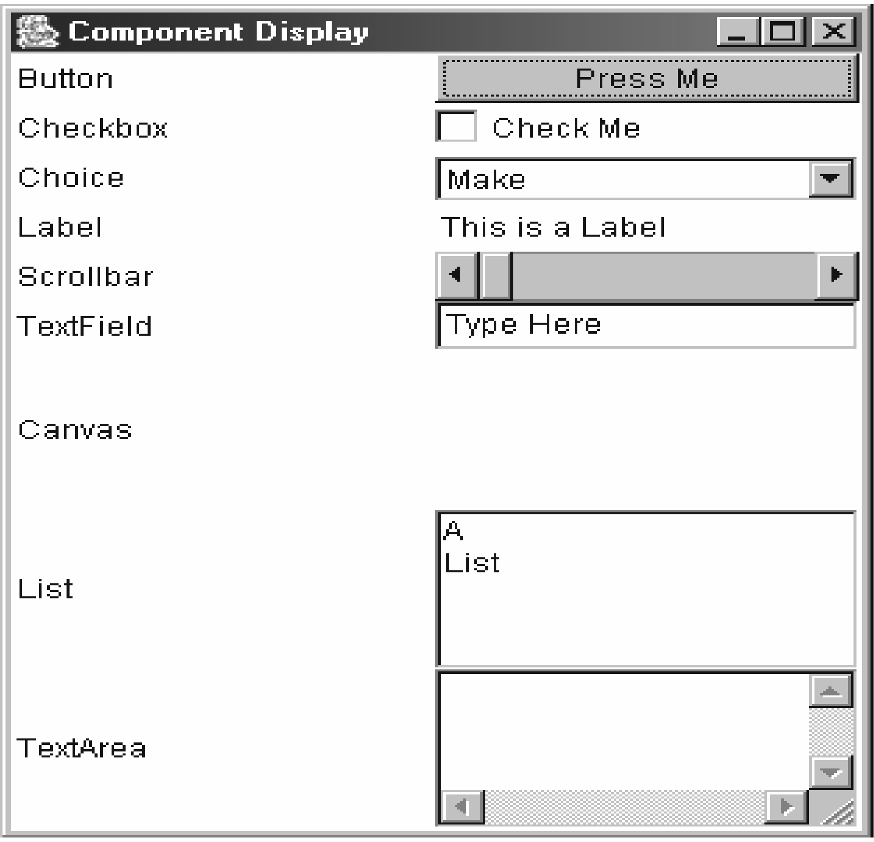
1) Compile e rode o exemplo da página 306.

Anotações

2) Compile e rode o exemplo da página 312.

Anotações

3) Faça o código fonte para a tela abaixo:



Anotações

Capítulo 13:

Eventos

Eventos

- Um evento pode ser um movimento, um clique no *mouse*, o pressionamento de uma tecla, a seleção de um item em um menu, a rolagem de um *scrollbar* e outros.
- Delegação de eventos para tratamento (a partir da *jdk 1.1*).
- Flexibilidade, componentização e conseqüentemente uma grande melhoria na **OOP**.
- Um mesmo componente pode ter vários ouvidores de eventos, desde que o ouvidor compatível com o componente gráfico.

A partir da *jdk1.1* tivemos uma grande diferença na forma de tratar de eventos. Foram implementadas umas séries de novos recursos e, modificados outros, para que fosse possível uma delegação de eventos para tratamento. O modelo anterior, para o tratamento de eventos, era o tratamento por hierarquia, que se tornou obsoleto.

Nesta nova forma de trabalhar com eventos, temos como destaque, uma maior flexibilidade, componentização e conseqüentemente uma grande melhoria na **OOP**.

Anotações

Um evento pode ser um movimento, um clique no mouse, o pressionamento de uma tecla, a seleção de um item em um menu, a rolagem de um scrollbar e outros.

Um ou mais objetos ouvidores de evento (tratadores de eventos ou *listeners*) podem registrar-se para serem notificados sobre a ocorrência de eventos de um certo tipo sobre determinado componente *(source)*.

Anotações

Classes de Eventos

Vários eventos podem ser gerados por uma ação do usuário na *interface.* As classes de tratadores de eventos foram agrupadas em um ou mais tipos de eventos com características semelhantes.

Classes e tipo de eventos:

- *java.awt.event.ActionEvent* - Evento de ação.
- *java.awt.event.AdjustmentEvent* - Evento de ajuste de posição.
- *java.awt.event.ComponentEvent* - Eventos de movimentação, troca de tamanho ou visibilidade.
- *java.awt.event.ContainerEvent* - Eventos de *container* se adicionado ou excluído.
- *java.awt.event.FocusEvent* - Eventos de foco.
- *java.awt.event.InputEvent* - Classe raiz de eventos para todos os eventos de entrada.
- *java.awt.event.InputMethodEvent* - Eventos de método de entrada com informações sobre o texto que está sendo composto usando um método de entrada.
- *java.awt.event.InvocationEvent* - Evento que executa métodos quando dispara uma thread.
- *java.awt.event.ItemEvent* - Evento de item de *list*, *choice* e *checkbox* se selecionado ou não.
- *java.awt.event.KeyEvent* - Eventos de teclado.
- *java.awt.event.MouseEvent* - Eventos de *mouse*.
- *java.awt.event.PaintEvent* - Eventos de *paint*.
- *java.awt.event.TextEvent* - Evento de mudança de texto.
- *java.awt.event.WindowEvent* - Eventos de janela.

Anotações

Listeners

■ O pacote *Java.awt.event*.

■ A implementação de uma *interface* ou uma subclasse de uma classe adaptadora pode ser usada para tratamento de eventos.

■ *Listeners* são objetos de qualquer classe que implementem uma interface específica para o tipo de evento que deseja tratar.

■ Os métodos add<tipo>*Listener*( ) e *remove*<tipo> *Listener*( ).

A implementação de uma *interface* ou uma subclasse de uma classe adaptadora pode ser usada para tratamento de eventos. Com certeza a melhor opção é o uso de *interfaces*, pois devido ao fato de *Java* trabalhar com heranças simples, uma mesmo componente poderá ter quantos *listeners* forem necessários, pois você pode implementar quantas *interface* quiser.

Logo, *listeners* são objetos de qualquer classe que implementem uma interface específica para o tipo de evento que deseja tratar. Essa interface é definida para cada classe de eventos. Então para a classe de eventos *java.awt.event.FocusEvent* existe a *interface java.awt.event.FocusListener*.

Anotações

Para a classe de eventos *java.awt.event.WindowEvent* existe a *interface* *java.awt.event.WindowListener* e assim sucessivamente.

O pacote onde encontraremos os recursos para tratamento de eventos é o *java.awt.event*.

Para adicionar um *listener* use o método *add*<tipo>*Listener*( ) ou para remover método *remove*<tipo>*Listener*( ) do componente ao qual se deseja ter o ouvidor. Por exemplo, se quiséssemos adicionar um ouvidor a um botão, usaríamos *botao.addActionListener*(*<objetoActionListener>*). E para remover *removeActionListener*(*<objetoActionListener>*).

Anotações

Tabela de *Interface* e Métodos

Interface	**Métodos**
ActionListener	*actionPerformed(ActionEvent)*
AdjustmentListener	*adjustmentValueChanged(AdjustmentEvent)*
AWTEventListener	*EventDispatched(AWTEvent)*
ComponentListener	*componentHidden(ComponentEvent)* *componentMoved(ComponentEvent)* *componentResized(ComponentEvent)* *componentShow(ComponentEvent)*
ContainerListener	*componentAdded(ContainerEvent)* *componentRemoved(ContainerEvent)*
FocusListener	*focusGained(FocusEvent)* *focusLost(FocusEvent)*
InputMethodListener	*caretPositionChanged(InputMthodEvent)* *inputMethodTextChanged(InputMethodEvent)*
ItemListener	*itemStateChanged(ItemEvent)*
KeyListener	*keyPressed(KeyEvent)* *keyReleased(KeyEvent)* *keyTyped(KeyEvent)*
MouseListener	*mousePressed(MouseEvent)* *mouseReleased(MouseEvent)* *mouseClicked(MouseEvent)* *mouseEntered(MouseEvent)* *mouseExited(MouseEvent)*
MouseMotionListener	*mouseDragged(MouseMotionEvent)* *mouseMoved(MouseMotionEvent)*
TextListener	*textValueChanged(TextEvent)*
WindowListener	*windowOpened(WindowEvent)* *windowActived(WindowEvent)* *windowDeactivated(WindowEvent)* *windowIconified(WindowEvent)* *windowDeiconified(WindowEvent)* *windowClosing(WindowEvent)* *windowClosed(WindowEvent)*

Anotações

Classes Adaptadoras

- Classes Adaptadoras implementam os métodos das *interfaces* de eventos.
- As vantagens de utilizar classes Adaptadoras.
- Sobrescrevendo os métodos das classes adaptadoras.
- A herança simples e os problemas refletidos nas classes que usam adaptadoras.

Classes Adaptadoras são classes abstratas que implementam os métodos das interfaces de eventos que possuem dois ou mais métodos.

A grande vantagem de utilizar classes Adapter é que não é necessário implementar todos os métodos da *interface*, pois já estão implementadas na classe abstrata. Deve-se apenas sobrescrever os método que tenham que realizar alguma tarefa.

Apesar de facilitar a implementação de um sistema de tratamento de eventos temos um grande problema, pois se no futuro você necessitar estender alguma classe e, já tiver estendido a adaptadora, então o código ira necessitar de uma remodelagem, isso significa retrabalho.

Anotações

Tabela de *interfaces* e classes adaptadoras correspondentes:

Listener	*Classe Adapter*
ComponentListener	*ComponentAdapter*
ContainerListener	*ContainerAdapter*
FocusListener	*FocusAdapter*
KeyListener	*KeyAdapter*
MouseListener	*MouseAdapter*
MouseMotionListener	*MouseMotionAdapter*
WindowListener	*WindowAdapter*

Anotações

Componentes e Eventos Suportados

Tabela de dos principais elementos de *interface* da **AWT** e os eventos suportados.

Componente	Eventos Suportados
Applet	*ComponetEvent, ContainerEvent, FocusEvent, KeyEvent, MouseEvent, MouseMotionEvent.*
Button	*ActionEvent, ComponetEvent, FocusEvent, KeyEvent, MouseEvent, MouseMotionEvent.*
Canvas	*ComponetEvent, FocusEvent, KeyEvent, MouseEvent, MouseMotionEvent.*
Checkbox	*ComponetEvent, FocusEvent, KeyEvent, MouseEvent, MouseMotionEvent.*
Choice	*ComponetEvent, FocusEvent, KeyEvent, MouseEvent, MouseMotionEvent.*
Component	*ComponetEvent, FocusEvent, KeyEvent, MouseEvent, MouseMotionEvent.*
Container	*ComponetEvent, ContainerEvent, FocusEvent, KeyEvent, MouseEvent, MouseMotionEvent.*
Dialog	*ComponetEvent, ContainerEvent, FocusEvent, KeyEvent, MouseEvent, MouseMotionEvent, WindowEvent.*
Frame	*ComponetEvent, ContainerEvent, FocusEvent, KeyEvent, MouseEvent, MouseMotionEvent, WindowEvent.*
Label	*ComponetEvent, FocusEvent, KeyEvent, MouseEvent, MouseMotionEvent.*
List	*ActionEvent, ComponetEvent, FocusEvent, ItemEvent, KeyEvent, MouseEvent, MouseMotionEvent.*
MenuItem	*ActionEvent.*
Panel	*ComponetEvent, ContainerEvent, FocusEvent, KeyEvent, MouseEvent, MouseMotionEvent.*
TextArea	*ComponetEvent, FocusEvent, KeyEvent, MouseEvent, MouseMotionEvent, TextEvent.*
TextField	*ActionEvent, FocusEvent, KeyEvent, MouseEvent, MouseMotionEvent, TextEvent.*
Window	*ComponetEvent, ContainerEvent, FocusEvent, KeyEvent, MouseEvent, MouseMotionEvent, WindowEvent.*

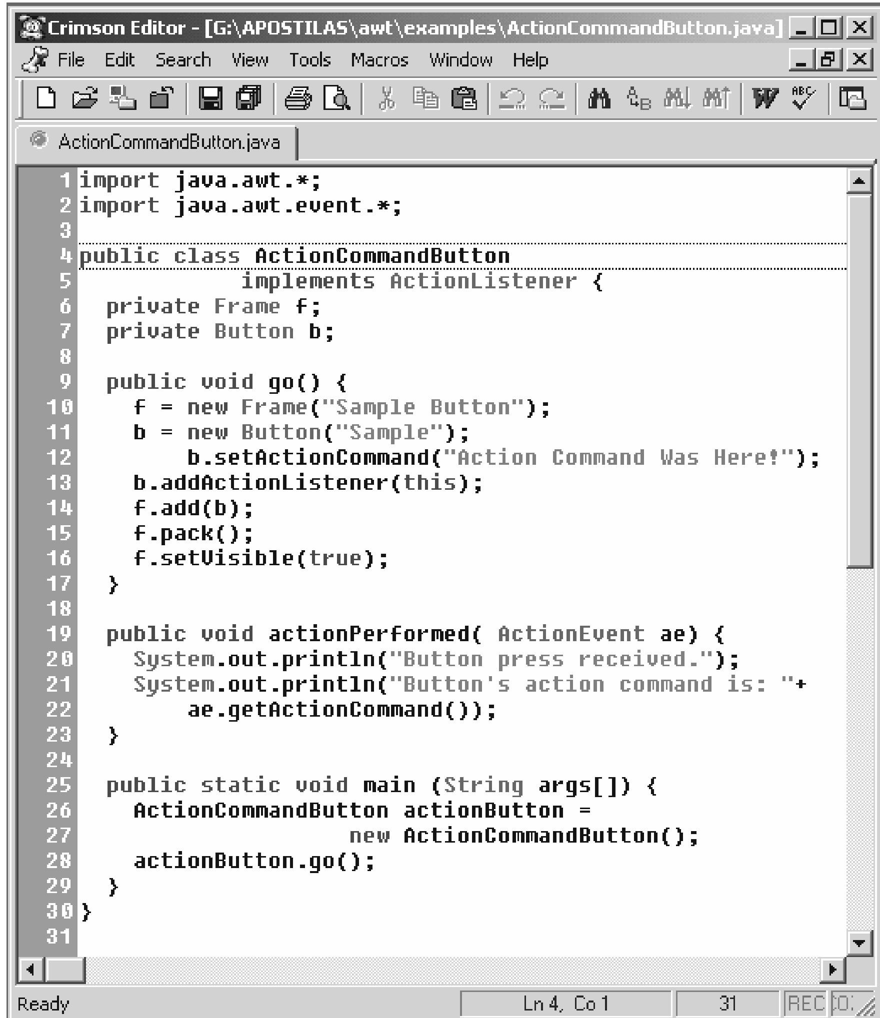
Anotações

Para fechar o *frame*, basta usar a *interface WindowListener* e colocar o método *System.exit*() dentro. Desta forma, quando o usuário clicar no botão sair do *frame*, o método *windowClosing*() será chamado automaticamente e fechará a janela.

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

public class TestWindow extends Frame implements WindowListener{

   public TestWindow(){

   this.addWindowListener(this);
   this.setVisible(true);
   this.setSize(300,300);

}

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

public void windowClosing     (WindowEvent event) {
   System.exit(0);
}

public void windowActivated        (WindowEvent event) { }
public void windowClosed           (WindowEvent event) { }
public void windowDeactivated      (WindowEvent event) { }
public void windowDeiconified      (WindowEvent event) { }
public void windowIconified        (WindowEvent event) { }
public void windowOpened           (WindowEvent event) { }

}
```

Anotações

Este exemplo mostra uma classe que implementa a *interface ActionListener*, logo isso irá torná-la uma classe ouvidora de eventos:



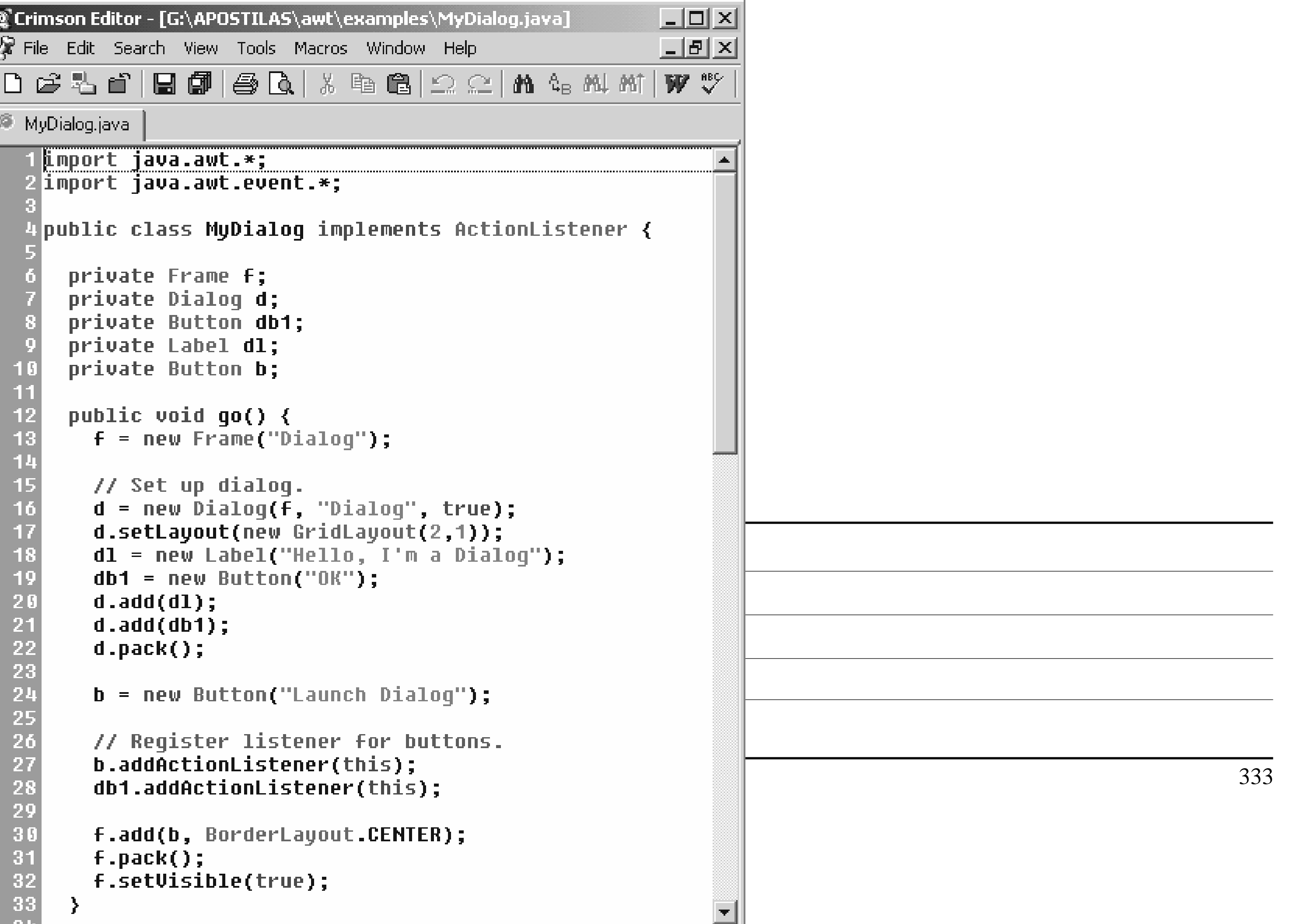

```java
import java.awt.*;
import java.awt.event.*;

public class ActionCommandButton
             implements ActionListener {
  private Frame f;
  private Button b;

  public void go() {
    f = new Frame("Sample Button");
    b = new Button("Sample");
        b.setActionCommand("Action Command Was Here!");
    b.addActionListener(this);
    f.add(b);
    f.pack();
    f.setVisible(true);
  }

  public void actionPerformed( ActionEvent ae) {
    System.out.println("Button press received.");
    System.out.println("Button's action command is: "+
        ae.getActionCommand());
  }

  public static void main (String args[]) {
    ActionCommandButton actionButton =
                      new ActionCommandButton();
    actionButton.go();
  }
}
```

Saída:

Anotações

Dialog

Abaixo o código para uma caixa de diálogo, o nome do arquivo é *SampleDialog.java*:



Continuação:

Crimson Editor - [G:\APOSTILAS\awt\examples\MyDialog.java]

File Edit Search View Tools Macros Window Help

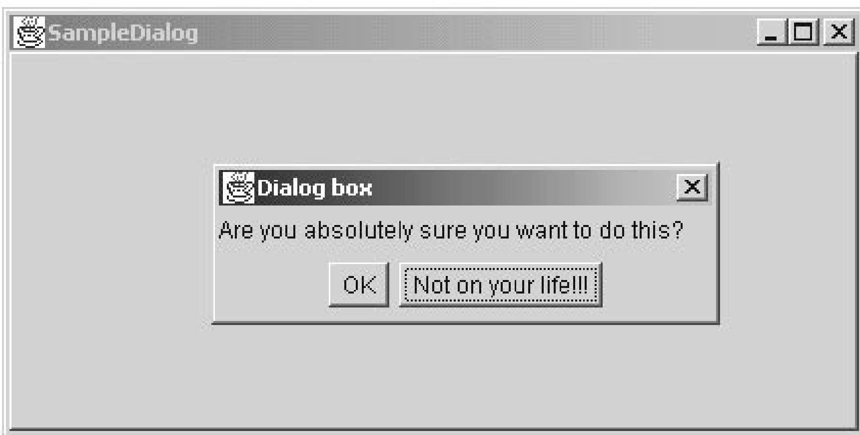
MyDialog.java

```
34
35    // Handler for all buttons.
36    public void actionPerformed( ActionEvent ae) {
37      String buttonPressed = ae.getActionCommand();
38      if (buttonPressed.equals("Launch Dialog")) {
39        d.setVisible(true);
40      } else if (buttonPressed.equals("OK")) {
41        System.out.println ("Process terminated!!!");
42        System.exit(0);
43      } else {
44        d.setVisible(false);
45      }
46    }
47
48    public static void main (String args[]) {
49      MyDialog myDialog = new MyDialog();
50      myDialog.go();
51    }
52  }
53
```

Ready Ln 1, Co 1 53

Anotações

Saída:



Anotações

Menu

Agora um exemplo de uso de *menus*.

Crimson Editor - [G:\APOSTILAS\awt\examples\SampleMenu.java]

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

public class SampleMenu
    implements ActionListener, ItemListener {
  private Frame f;
  private MenuBar mb;
  private Menu m1;
  private Menu m2;
  private Menu m3;
  private MenuItem mi1;
  private MenuItem mi2;
  private MenuItem mi3;
  private MenuItem mi4;
  private CheckboxMenuItem mi5;

  public void go() {
    f = new Frame("Menu");
    mb = new MenuBar();
    m1 = new Menu("File");
    m2 = new Menu("Edit");
    m3 = new Menu("Help");
    mb.add(m1);
    mb.add(m2);
    mb.setHelpMenu(m3);
    f.setMenuBar(mb);

    mi1 = new MenuItem("New");
    mi2 = new MenuItem("Save");
    mi3 = new MenuItem("Load");
    mi4 = new MenuItem("Quit");
    mi1.addActionListener(this);
    mi2.addActionListener(this);
    mi3.addActionListener(this);
    mi4.addActionListener(this);
    m1.add(mi1);
    m1.add(mi2);
    m1.add(mi3);
    m1.addSeparator();
    m1.add(mi4);
```

Anotações

Continuação:

Crimson Editor - [G:\APOSTILAS\awt\examples\SampleMenu.java]

File Edit Search View Tools Macros Window Help

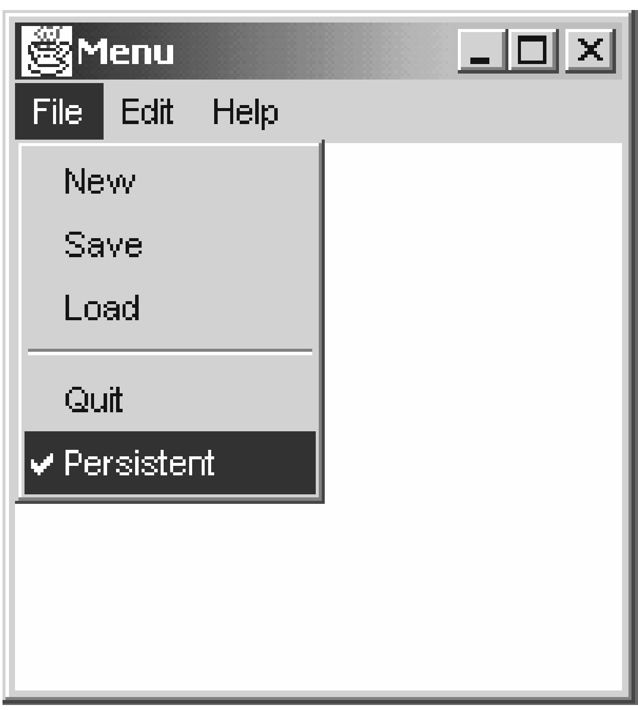
SampleMenu.java

```
41
42      mi5 = new CheckboxMenuItem("Persistent");
43      mi5.addItemListener(this);
44      m1.add(mi5);
45
46      f.setSize(200,200);
47      f.setVisible(true);
48    }
49
50    public void actionPerformed( ActionEvent ae) {
51      System.out.println("Button \"" +
52          ae.getActionCommand() + "\" pressed.");
53
54      if (ae.getActionCommand().equals("Quit")) {
55        System.exit(0);
56      }
57    }
58
59    public void itemStateChanged(ItemEvent ie) {
60      String state = "deselected";
61
62      if (ie.getStateChange() == ItemEvent.SELECTED) {
63        state = "selected";
64      }
65      System.out.println(ie.getItem() + " " + state);
66    }
67
68    public static void main (String args[]) {
69      SampleMenu sampleMenu = new SampleMenu();
70      sampleMenu.go();
71    }
72  }
73
```

Ready | Ln 4, Co 1 | 73

Anotações

Saída:



Anotações

Laboratório 12:

Capítulo 13: Eventos

Concluir o(s) exercício(s) proposto(s) pelo instrutor. O instrutor lhe apresentará as instruções para a conclusão do mesmo.

Laboratório 12 - Capítulo 13

1) Compile e rode o exemplo da página 331.

Anotações

2) Compile e rode o exemplo da página 332.

Anotações

3) Compile e rode o exemplo da página 333.

Anotações

4) Compile e rode o exemplo da página 336.

Anotações

5) Implemente a *interface WindowListener* nos exercícios anteriores para que o **GUI** possa ser fechado através do botão fechar. Use o código da página 331 como exemplo.

Anotações

Gabarito dos Laboratórios

Anotações

Gabarito Laboratório 1:

Capítulo 2: Introdução a *Object Oriented* com *Java*

1) Compile e rode o exemplo da página 33 (código de exemplo de objeto):

Gabarito

```
public class Funcionario{
      String nome;
      String endereco;
      int idade;
     public static void main(String []args){
 Funcionario func = new Funcionario();
    func.nome = "Arnaldo";
    func.endereco = "Rua Java";
    func.idade = 25;
    System.out.println("Nome: " + nome);
    System.out.println("End.: " + endereco);
    System.out.println("Idade: " + idade);
   }
}
```

Anotações

2) Compile e rode o exemplo da página 35 (exemplo do uso de método).

Gabarito

```
public class Funcionario{
        String nome;
        String endereco;
        int idade;

        public static void main(String []args){
     Funcionario func = new Funcionario();

     func.nome = "Arnaldo";
     func.endereco = "Rua Java";
     func.idade = 25;

     func.imprimir();
   }

   public void imprimir(){
     System.out.println("Nome: " + nome);
     System.out.println("End.: " + endereco);
     System.out.println("Idade: " + idade);
   }
}
```

Anotações

3) Implemente o código para a análise feita durante o módulo, contendo os itens abaixo:

Classe: Comemoracao

Atributo 1 – SÍMBOLO: “Árvore de Natal”

Atributo 2 – PERSONAGEM: “Papai Noel”

Atributo 3 – SIGNIFICADO: “Nascimento de Jesus Cristo”

Métodos de inicialização de atributos

Métodos de impressão de atributos

Gabarito

```
public class Comemoracao{
    String simbolo;
    String personagem;
    String significado;

    public static void main(String args[]){
        Comemoracao natal = new Comemoracao();

        natal.inserirAtributoSimbolo(“ Arvore de Natal “);
        natal.inserirAtributoPersonagem(“ Papai Noel “);
        natal.inserirAtributoSignificado(“ Nascimento de Jesus Cristo “);

        natal.imprimeAtributoSimbolo();
        natal.imprimeAtributoPersonagem();
        natal.imprimeAtributoSignificado();
    }

    //Métodos de inicialização de atributos:

    public void inserirAtributoSimbolo ( String param ) {
        simbolo = param;
    }
```

Anotações

```
    public void inserirAtributoPersonagem ( String param ) {
        personagem = param;
    }
    public void inserirAtributoSignificado ( String param ) {
        significado = param;
    }
    //Métodos de impressão de atributos:
    public void imprimeAtributoSimbolo (){
        System.out.println("Simbolo: " + simbolo);
    }

    public void imprimeAtributoPersonagem (){
        System.out.println("Personagem: " + personagem);
    }
     public void imprimeAtributoSignificado (){
        System.out.println("Significado: " + significado);
    }
}
```

Anotações

4) Repita o exercício anterior, só que desta teste usando características de comemoração da páscoa, para analisar se este código também pode ser usado para outras comemorações.

Gabarito

```
public class Comemoracao{
        String simbolo;
        String personagem;
        String significado;

        public static void main(String args[]){
                Comemoracao natal = new Comemoracao();

                natal.inserirAtributoSimbolo(“ Ovo de Pascoa “);
                natal.inserirAtributoPersonagem(“ Coelho da Pascoa “);
                natal.inserirAtributoSignificado(“ Ressureicao de Jesus Cristo
                “);

                natal.imprimeAtributoSimbolo();
                natal.imprimeAtributoPersonagem();
                natal.imprimeAtributoSignificado();

        }

        //Métodos de inicialização de atributos:

        public void inserirAtributoSimbolo ( String param ) {
                simbolo = param;
        }

        public void inserirAtributoPersonagem ( String param ) {
                personagem = param;
        }

        public void inserirAtributoSignificado ( String param ) {
                significado = param;
        }
```

Anotações

```
        //Métodos de impressão de atributos:
        public void imprimeAtributoSimbolo (){
            System.out.println("Simbolo: " + simbolo);
        }

        public void imprimeAtributoPersonagem (){
            System.out.println("Personagem: " + personagem);
        }
        public void imprimeAtributoSignificado (){
            System.out.println("Significado: " + significado);
        }
}
```

Anotações

Anotações

Gabarito Laboratório 2:

Capítulo 3: Aprofundando Conceitos Iniciais

1) Implemente um construtor e o conceito de encapsulamento no exercício 2 do capítulo anterior.

Gabarito

```
public class Comemoracao{
    private String simbolo; // Variável encapsulada
    private String personagem; // Variável encapsulada
    private String significado; // Variável encapsulada

    // Construtor
    public Comemoracao(String param1, String param2, String param3){
        simbolo = param1;
        personagem = param2;
        significado = param3;
    }

    public static void main(String args[]){
        Comemoracao natal = null;

      // Usando o construtor para inicializar as variáveis
      natal = new Comemoracao(" Ovo de Pascoa ",
      " Coelho da Pascoa "," Ressureicao de Jesus Cristo ");

    natal.imprimeAtributoSimbolo();
    natal.imprimeAtributoPersonagem();
    natal.imprimeAtributoSignificado();
    }

    //Métodos de inicialização de atributos:
     public void inserirAtributoSimbolo ( String param ) {
          simbolo = param;
    }

    public void inserirAtributoPersonagem ( String param ) {
```

Anotações

```
        personagem = param;
    }
    public void inserirAtributoSignificado ( String param ) {
        significado = param;
    }

    //Métodos de impressão de atributos:

    public void imprimeAtributoSimbolo (){
        System.out.println("Simbolo: " + simbolo);
    }


    public void imprimeAtributoPersonagem (){
        System.out.println("Personagem: " + personagem);
    }

    public void imprimeAtributoSignificado (){
        System.out.println("Significado: " + significado);
    }
}
```

Anotações

2) Compile e rode o exemplo de pacotes da página 62.

Gabarito

```
package empresa.departamento;
public class RH{
        private String contratado;

        public RH (String a){
            contratado = a;
        }
}
```

Anotações

3) Compile e rode o exemplo de *import* da página 64.

Gabarito

```
import empresa.departamento.*;
//ou
import empresa.departamento.RH;
public class Corporacao{
        public static void main(String []args){
            RH rh = new RH("Tom Hanks");
        }
}
```

Anotações

4) Compile e rode o exemplo de **FQN** do módulo 3, da página 66. Após fazer isso retire os **FQN´s** (linhas 9 e 11) e, compile o código novamente para analisar o resultado desta alteração.

Gabarito

```
import java.util.*; // Aqui temos uma classe Date
import java.sql.*; // Aqui também temos uma classe Date
public class FQN{
        public static void main(String []args){
            java.util.Date date1 = new java.util.Date(2003,05,23);
            java.sql.Date date2 = new java.sql.Date(2003,05,23);
          }
}
```

Retirando os FQNs:

```
import java.util.*; // Aqui temos uma classe Date
import java.sql.*; // Aqui também temos uma classe Date
public class FQN{
        Public static void main(String []args){
            Date date1 = new Date(2003,05,23);
            Date date2 = new Date(2003,05,23);

        }
}
```

Anotações

Gabarito Laboratório 3:

Capítulo 4: *Sintaxe Java 2 Platform*

1) Teste os identificadores conforme a tabela que foi apresentada na página 78, a seguir segue uma cópia da mesma:

1	A primeira posição deverá ser com uma letra, “_” ou “$”.
2	Ser formado por caracteres **UNICODE**.
3	Não ser uma palavra reservada: ***goto** e **const.***
4	Não ser igual a literal: ***null.***
5	Não ser repetido dentro do seu *escopo*.
6	Não podem ser as *keywords* ou as *Boolean Literais*: ***true** e **false***.
7	Não pode ser uma *keword Java: int*, *byte*, *class* etc.

Gabarito

```
public class TestIdentificadores{
        public static void main(String args[]){
                int _var; // OK
                char $var; // OK
                int VAR; // OK
                int Var1; // OK

                // int 1VAR; // erro
                // short goto; // erro
                // boolean const; // erro
                // byte null; // erro
                // float false; // erro
                // double true; // erro
                // int class; // erro
                // int int; // erro
        }
}
```

Anotações

2) Teste os tipos de variáveis estudadas neste módulo, são elas:

- Tipos Integrais.
- Ponto Flutuante.
- Tipo Caractere.
- Dígrafos.
- Tipo Lógico.

Gabarito

```
public class TestVar{
    public static void main(String args[]){
        // Tipos Integrais
        byte b = 127;
        short s = 16;
        int     i = 1;
        long L = 8L;
        // Ponto Flutuante
        float f = 1.1F;
        double d = 2.2D;
        // Tipo Caractere
        char c = 'A';
        // Dígrafos
        char dig = '\n';

        //Tipo Lógico
        boolean boo = false;
        System.out.println(i);
        System.out.println(dig);
        System.out.println(c);
```

Anotações

```
            System.out.println(b);
            System.out.println(s);
            System.out.println(boo);
            System.out.println(f);
            System.out.println(d);
            System.out.println(L);
        }
    }
```

Anotações

3) Compile e rode o exemplo de Passagem por Referência da página 88.

Gabarito

```
public class Referencia{
            int valor;
     public static void main(String []args){
            Referencia ref = new Referencia();
            System.out.println("Valor antes: " + ref.valor);
            ref.incrementar( ref );
            System.out.println("Valor depois: " + ref.valor);
     }
     public void incrementar(Referencia r){
            r.valor++;
            }
}
```

Anotações

4) Compile e rode o exemplo de Passagem por Valor da página 90.

Gabarito

```
public class Valor{
        int valor;
    public static void main(String []args){
        Valor ref = new Valor();
        System.out.println("Valor antes: " + ref.valor);

        ref.incrementar( ref );
        System.out.println("Valor depois: " + ref.valor);
    }
    public void incrementar(Valor r){
        int valorLocal = r.valor;
        valorLocal++;
        }
}
```

Anotações

Gabarito Laboratório 4:

Capítulo 5: Usando Operadores *Java*

1) Analise e encontre o erro no código abaixo:

```
public class exemplo03 {
 public static void main (String args[])
    {
    int x =10;
    int y =3;
    System.out.println("x =" + x);
    System.out.println("y =" + y);
    System.out.println("-x =" + (-x));
    System.out.println("x/y=" + (x/y));
    System.out.println("Resto de x por y=" + (x/y));////%
    System.out.println("inteiro de x por y =" + (x/y));
    System.out.println("x +1 =" + (x++));////++x
    }
}
```

Gabarito

```
public class exemplo03 {
    public static void main (String args[])
    {
    int x =10;
    int y =3;
    System.out.println("x =" + x);
    System.out.println("y =" + y);
    System.out.println("-x =" + (-x));
    System.out.println("x/y=" + (x/y));
    System.out.println("Resto de x por y=" + (x%y));// usar modulus (%)
    System.out.println("inteiro de x por y =" + (x/y));
    System.out.println("x +1 =" + (++x)); //colocar pré-incremento
    }
}
```

Anotações

2) Crie um programa que inicialize uma variável *double* de acordo com o resto de uma divisão qualquer, desde que esse seja ímpar, caso contrário inicialize com 1000. Utilize para isto o operador *ternary*.

Gabarito

```
public class Ternary{
        public static void main(String args[]){
            double var = 11;
            double result = var % 2 != 0 ? var % 2 : 1000;
            System.out.println( result );
        }
}
```

Anotações

3) Faça a conversão dos números que estão na Base 16 para a Base 2 e para Base 10.

21 (base 16) = 0010 0001 (base 2) = 33 (base 10)
08 (base 16) = 0000 1000 (base 2) = 8 (base 10)
FF (base 16) = 1111 1111 (base 2) = 255 (base 10)
1A3 (base 16) = 0001 1010 0011 (base 2) = (1*256)+(10*16)+(3*1) = 419 (base 10)
0xFEDC123 (base 16)= 1111 1110 1101 1100 0001 0010 0011 (Base 2)

Anotações

Gabarito Laboratório 5:

Capítulo 6: Controle de Fluxo

1) Faça um programa *Java* que imprima na tela todos os números ímpares em uma contagem até 25, use o *loop for*.

Gabarito

```
public class Ex{
    public static void main(String args[]){
        for(int i=0; i<=25; i++){
            System.out.println( i%2 != 0 ? i+"" : "" );
        }
    }
}
```

Anotações

2) Faça um programa *Java* que imprima na tela todos os números pares em uma contagem até 25, use o *loop do*.

Gabarito

```
public class Ex{
    public static void main(String args[]){
        int i=0;
        do{
            i++;
            System.out.println( i%2 != 0 ? i+”” : “” );
        }while(i<=25);
    }
}
```

Anotações

3) Faça um programa *Java* que imprima na tela todos os números em uma contagem até 25 exceto os números 8, 17, 21, use o *loop while*.

Gabarito

```
public class Ex{
    public static void main(String args[]){
        int i=0;
        while( i<=25 ){
            i++;
            System.out.println( i!=8 && i!=17 && i!= 21 ? i+"" : "" );
        }
    }
}
```

Anotações

4) Faça um programa *Java* que imprima na tela:

- Bom dia
- Boa tarde
- Boa noite

Use para isto uma variável int, com horas inteiras apenas. Implemente o código com if/else/elseif.

Exemplo:

- 00:00 às 11:00 - Bom dia
- 12:00 às 17:00 - Boa tarde
- 18:00 às 23:00 - Boa noite

Gabarito

```
public class Ex{
    public static void main(String args[]){
        // - 00:00 às 11:00 - Bom dia
        // - 12:00 às 17:00 - Boa tarde
        // - 18:00 às 23:00 - Boa noite
        int hora=0, i=0;
        while( i<=24 ){
            if( hora>=0 && hora<12){
              System.out.println("Hora: " + hora + " - Bom dia" );
            }
            else if( hora>=12 && hora<18){
              System.out.println("Hora: " + hora + " - Boa tarde" );
            }
            else{
```

Anotações

```
                System.out.println("Hora: " + hora + " - Boa noite" );
               }
               hora++;
               i++;
          }
     }
}
```

Anotações

5) Repita o exercício anterior, porém ao invés de usar *if*, use *switch*. Use como seletor uma variável char conforme exemplo abaixo:

- ‘D’- Bom dia
- ‘T’- Boa tarde
- ‘N’- Boa noite

Gabarito

```
public class Ex{
    public static void main(String args[]){
        // - D - Bom dia
        // - T - Boa tarde
        // - N - Boa noite
        char hora = ‘D’;
        int i = 0;

        while( i<=2 ){

        switch ( hora){
          case ‘D’:
            System.out.println(“Hora: “ + hora + “ - Bom dia” );
            break;

            case ‘T’:
              System.out.println(“Hora: “ + hora + “ - Boa tarde” );
              break;

            default:
              System.out.println(“Hora: “ + hora + “ - Boa noite” );
              }
```

Anotações

```
                hora = (i == 1) ? 'T' : 'N';
                i++;
            }
        }
    }
```

Anotações

Gabarito Laboratório 6:

Capítulo 7: *Arrays*

1) Compile e rode o exemplo da página 161.

Gabarito

```
public class Ex{
        public static void main(String []args){
                // vamos verificar a quantidade de elementos no array
                // através da variável length
                int tamanho = args.length;
                System.out.println("Tamanho do Array: " + tamanho);
                for (int i=0; i<tamanho; i++){
                        System.out.println("Elemento No." + i + " = " + args[i]);
                }
        }
}
```

Anotações

2) Faça um programa que leia um argumento da linha de comando e imprima na tela os itens:

- Quantos caracteres possuem este argumento
- A palavra em maiúsculo
- A primeira letra da palavra
- A palavra sem as vogais

Gabarito

```
public class Ex{

    public static void main(String []args){
        // - Quantos caracteres possuem este argumento
           // - A palavra em maiúsculo
           // - A primeira letra da palavra
           // - A palavra sem as vogais

        int tamanho = args.length;

        // - Quantos caracteres possuem este argumento
        System.out.println("Tamanho do Array: " + tamanho);
        // - A palavra em maiúsculo
           System.out.println("Palavra em maiusculo: " +
           args[0].toUpperCase() );

        // - A primeira letra da palavra

        System.out.println("Primeira letra: " + args[0].charAt(0) );
        // - A palavra sem as vogais
```

Anotações

```
            String nova = args[0];
            nova = nova.replace('a', '\u0000');
            nova = nova.replace('e', '\u0000');
            nova = nova.replace('i', '\u0000');
            nova = nova.replace('o', '\u0000');
            nova = nova.replace('u', '\u0000');
            System.out.println("Palavra em as vogais: " + nova );
        }
}
```

Anotações

3) Faça um programa que leia um argumento da linha de comando, crie um segundo *array* e coloque nele todo o conteúdo do *array* anterior, mais uma cópia de cada elemento em maiúsculo. Use o método *System.arraycopy()* da **API** *Java* como auxílio.

Gabarito:

```
public class Ex{

    public static void main(String []args){

        String []array2 = new String [ args.length * 2];

        System.arraycopy(args, 0, array2, 0, args.length);

        for (int i=0; i<args.length; i++){
            args[i] = args[i].toUpperCase();
        }

        System.arraycopy(args, 0, array2, args.length, args.length);

        for (int i=0; i<array2.length; i++){
            System.out.println("Elemento No." + i + " = " + array2[i]);
        }
    }
```

Anotações

Gabarito Laboratório 6 - Capítulo 7:

Anotações

Gabarito Laboratório 7:

Capítulo 8: Programação Avançada

1) Crie uma classe chamada carro e implemente a interface da página 171. Coloque mensagens de teste dentro dos métodos e, faça a chamada dos mesmos.

Gabarito

```
public interface MotorInterface{
        public void cambioAutomatico(int valor);
        public boolean injecaoAutomatica();
}
public class Carro implements MotorInterface{
        public static void main(String []args){
            Carro carro = new Carro();
            carro.cambioAutomatico(10);
            carro.injecaoAutomatica();
        }
        public void cambioAutomatico(int valor){
            System.out.println("Cambio Automatico OK");
        }
        public boolean injecaoAutomatica(){
            System.out.println("Injecao Automatica OK");
            return true;
        }
}
```

Anotações

2) Crie uma classe que estenda a classe abstrata da página 175. Faça os *overridings* necessários e implemente os métodos com suas respectivas funcionalidades.

Gabarito

```
public abstract class Calculos{

    public abstract int soma (int a, int b);

    public abstract int div (int a, int b);

    public abstract int sub (int a, int b);

    public abstract int multi (int a, int b);

    public void imprimir () {
        // Método concreto
    }
}

class Calculando extends Calculos{

    public static void main(String []args){
        Calculando calc = new Calculando();

        System.out.println("soma: " + calc.soma(10,5));
        System.out.println("div: " + calc.div(10,5));
        System.out.println("sub: " + calc.sub(10,5));
        System.out.println("multi: " + calc.multi(10,5));
    }
```

Anotações

```
        public int soma (int a, int b){
                return (a+b);
        }

        public int div (int a, int b){
                return (a/b);
        }

        public int sub (int a, int b){
                return (a-b);
        }

        public int multi (int a, int b){
                return (a*b);
        }
}
```

Anotações

3) Faça um programa chamado Funcionario com as variáveis nome e idade. Após isso, implemente os seguintes *overloadings* de construtores e métodos:

```
public Funcionario ()
public Funcionario (String nome)
public Funcionario (int idade)
public Funcionario (String nome, int idade)

public void setInf (String nome)
public void setInf (int idade)
public void setInf (String nome, int idade)
```

Gabarito

```
public class Funcionario{
	String nome;
	int idade;

	public Funcionario (){ }

	public Funcionario (String nome){
		this.nome = nome;
	}

	public Funcionario (int idade){
		this.idade = idade;
	}

	public Funcionario (String nome, int idade){
		this.nome = nome;
		this.idade = idade;
	}
```

Anotações

```
        public void setInf (String nome){
                this.nome = nome;
        }

        public void setInf (int idade){
                this.idade = idade;
        }

        public void setInf (String nome, int idade){
                this.idade = idade;
                this.nome = nome;
        }
        public static void main(String []args){

        Funcionario func1 = new Funcionario("Arnaldo");
        Funcionario func2 = new Funcionario(26);
        Funcionario func3 = new Funcionario("Arnaldo", 26);

        Funcionario func4 = new Funcionario();
        func4.setInf("Arnaldo");
        func4.setInf(26);
        func4.setInf("Arnaldo", 26);

        func1.imprimir();
        func2.imprimir();
        func3.imprimir();
        func4.imprimir();
    }

    public void imprimir(){
        System.out.println("Nome: " + nome);
        System.out.println("Idade: " + idade);
    }
}
```

Anotações

4) Faça o encapsulamento dos atributos do exercício anterior. E verifique se muda algo na compilação ou execução.

Gabarito

```
public class Funcionario{
        private String nome; //encapsulado
        private int idade; //encapsulado

        public Funcionario (){ }

        public Funcionario (String nome){
                this.nome = nome;
        }

        public Funcionario (int idade){
                this.idade = idade;
        }

        public Funcionario (String nome, int idade){
                this.nome = nome;
                this.idade = idade;
        }

        public void setInf (String nome){
                this.nome = nome;
        }

        public void setInf (int idade){
                this.idade = idade;
        }
        public void setInf (String nome, int idade){
                this.idade = idade;
```

Anotações

```
                this.nome = nome;
            }

    public static void main(String []args){

            Funcionario func1 = new Funcionario("Arnaldo");
            Funcionario func2 = new Funcionario(26);
            Funcionario func3 = new Funcionario("Arnaldo", 26);

            Funcionario func4 = new Funcionario();
            func4.setInf("Arnaldo");
            func4.setInf(26);
            func4.setInf("Arnaldo", 26);

            func1.imprimir();
            func2.imprimir();
            func3.imprimir();
            func4.imprimir();
        }

        public void imprimir(){
            System.out.println("Nome: " + nome);
            System.out.println("Idade: " + idade);
        }
    }
```

Anotações

5) Compile o código da página 187, retire o construtor abaixo da classe pai e resolva qualquer problema em virtude disto.

Contrutor:

```
        public Acesso () {
                // falz algo
        }
```

Gabarito

```
public class Acesso{

        public Acesso (int i) {
                //COMENTADO A LINHA ABAIXO QUE
                //CHAMAVA O CONSTRUTOR REMOVIDO
                //this(); // acessa o outro construtor desta classe
                this.teste(); // acessa o metodo desta classe
        }

        public void teste () {
                // falz algo
        }
}
class TesteAcesso extends Acesso {

        public TesteAcesso () {
                super(10); // acessa o construtor pai
                super.teste(); // acessa o metodo pai
        }

        public void teste () {
                // falz algo
        }
}
```

Anotações

6) Compile e rode o exemplo da página 189.

Gabarito

```
public class MembrosStatic{
        static int conta;

        public static void inc(){
                conta++;
        }
}

class OutraClasse {

        public static void main(String args[]) {
                MembrosStatic ms1 = new MembrosStatic();
                MembrosStatic ms2 = new MembrosStatic();

                ms1.conta++;

                MembrosStatic.conta++;

                ms2.conta++;

                System.out.println(MembrosStatic.conta);
        }
}
```

Anotações

7) Compile o programa da página 192. Teste da seguinte forma:
- Faça uma nova classe e tente estendê-la

Gabarito

```
final class Final{
        final int var = 10;

        final void teste(){
                // Faz algo
        }
}

class ExtendsFinalClass extends Final{}
```

Resposta do Compilador:

```
Final.java:9: cannot inherit from final Final
public class ExtendsFinalClass extends Final{}
                                       ^
2 errors
```

- Retire a keyword final da classe e faça uma subclasse, agora tente fazer um overriding do método final.

Gabarito

```
class Final{
        final int var = 10;

        final void teste(){
```

Anotações

```
                // Faz algo
        }
}
class ExtendsFinalClass extends Final{
        final void teste(){
                // Faz algo
        }
}
```

Resposta do Compilador:

```
Final.java:10: teste() in ExtendsFinalClass cannot override teste() in Final; ov
erridden method is final
        final void teste(){
                 ^
1 error
```

- Retire a keyword final do método, inclua um construtor na subclasse e tente mudar o valor da variável final

Gabarito

```
class Final{
        final int var = 10;

        void teste(){
                // Faz algo
        }
}

class ExtendsFinalClass extends Final{
```

Anotações

```
    public ExtendsFinalClass(){
        var = 1000;
    }
    void teste(){
        // Faz algo
    }
}
```

Resposta do Compilador:

```
Final.java:12: cannot assign a value to final variable var
        var = 1000;
        ^
1 error
```

Anotações

Anotações

Gabarito Laboratório 8:

Capítulo 9: *Java Exceptions* e Operações de I/O

1) Compile e rode o código da página 207.

Gabarito:

```
public class Ex{
        public static void main(String args[]) {
                Ex ex = new Ex();
                ex.test();
        }

        public void test(){
                try {
                        declara (50);
                        declaraLanca (50);

                } catch (ArithmeticException e) {
                        System.out.println("Uma divisao por zero ocorreu: " + e);
                }
        }

        //declaração da exception
        public void declara (int i) throws ArithmeticException {

                int r = i/0;
        }

        //declaração da exception
        public void declaraLanca (int i) throws ArithmeticException {

        throw new ArithmeticException ();
        }
}
```

Anotações

2) Compile e rode o exemplo da página 210.

Gabarito:

```
public class Exception01{
        public static void main(String []args){
                imprimeArray();
        }
        public static void imprimeArray () {
                int []arrayInt = new int [2];

                try {
                        for (int i = 0 ; i < 3 ; i++) {
                                arrayInt [i] = 50;
                        }
                } catch (ArrayIndexOutOfBoundsException e){
                        System.out.println( “Ocorreu o problema : “ + e );
                } finally {
                        System.out.println(“Este bloco sempre é executado” );
                }
    }
}
```

Anotações

3) Compile e rode o exemplo da página 215 e 216.

Gabarito:

```
import java.net.*;
import java.io.*;

public class Server {
    public static void main(String args[]) {
        ServerSocket s = null;

    // Registrando o serviço na porta 5432
    try {
        s = new ServerSocket(5432);
    } catch (IOException e) {
    // Faz algo
    }

    // Loop infinito para listen/accept
    while (true) {
    try {
        // Aguarda neste ponto esperando uma conexão
        Socket s1 = s.accept();

        // Obtém um output stream associado a um socket
        OutputStream s1out = s1.getOutputStream();
        DataOutputStream dos = new DataOutputStream(s1out);

        // Send um Ola Stream
        dos.writeUTF("Ola Stream");

        // Fecha a conexão, mas não o server socket
        dos.close();
        s1.close();
        } catch (IOException e) {
```

Anotações

```
            // Faz Algo
          }
        }
      }
    }

    import java.net.*;
    import java.io.*;

    public class Client {
      public static void main(String args[]) {
      try {
        // Abre a conexão no servidor, na porta 5432
        Socket s1 = new Socket("127.0.0.1", 5432);

        // Obtém um input stream para o socket
        InputStream is = s1.getInputStream();
        // Lê a informação como um data input stream
        DataInputStream dis = new DataInputStream(is);

        // Lê a entrada e imprime na tela
        System.out.println(dis.readUTF());

        // Fechamos agora a input stream e a conexão
        dis.close();
        s1.close();
      } catch (ConnectException connExc) {
        System.err.println("Não foi possível conectar.");
      } catch (IOException e) {
      // Faz algo
      }
    }
    }
```

Anotações

Gabarito Laboratório 8 - Capítulo 9:

Anotações

Gabarito Laboratório 9:

Capítulo 10: *Java Collections Framework*

1) Compile e rode o exemplo da página 240.

Gabarito

```
import java.util.*;

public class SetExample {

    public static void main(String args[]) {

        Set set = new HashSet(); //criação de um objeto HashSet
        set.add("1 Pato");
        set.add("2 Cachorro");
        set.add("3 Gato");
        set.add("4 Porco");
        set.add("5 Vaca");
        System.out.println("HashSet - " + set);

        Set sortedSet = new TreeSet(set); //criação de um objeto TreeSet
        System.out.println("TreeSet - " + sortedSet);

        Set linkedSet = new LinkedHashSet(sortedSet);
        System.out.println("LinkedHashSet - " + linkedSet);
        }
}
```

Anotações

2) Compile e rode o exemplo da página 244.

Gabarito

```
import java.util.*;

public class ListExample {

    public static void main(String args[]) {
        List list = new ArrayList();
        list.add("1 Pato");
        list.add("2 Cachorro");
        list.add("3 Gato");
        list.add("4 Porco");
        list.add("5 Vaca");

        System.out.println(list); // Imprimindo a List
        System.out.println("1: " + list.get(1));
        System.out.println("0: " + list.get(0));

        LinkedList linkedList = new LinkedList(); // Usando LinkedList
        linkedList.addFirst("1 Pato"); // Atribuindo elementos
        linkedList.addFirst("2 Cachorro");
        linkedList.addFirst("3 Gato");
        linkedList.addFirst("4 Porco");
        linkedList.addFirst("5 Vaca");
        System.out.println(linkedList); // Imprimindo a LinkedList

        linkedList.removeLast();
        linkedList.removeLast();
        System.out.println(linkedList); // Sem os dois últimos elementos
    }
}
```

Anotações

3) Compile e rode o exemplo da página 250.

Gabarito

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

public class MapExample {

    public static void main(String args[]) throws java.io.IOException{
        Map map = new HashMap();

        String key = null;
        String value = null;

        System.out.println("\n Palavra digitada:  " + args[0] + '\n');

            for(int i=0; i<3; i++){

            switch(i){
                case 0:
                    key = "length";
                    value = args[0].valueOf( args[0].length() );
                    break;
                case 1:
                    key = "substring(0,1)";
                    value = args[0].substring(0,1);
                    break;
                case 2:
                    key = "toUpperCase";
                    value = args[0].toUpperCase();
            }
```

Anotações

```
                map.put(key, value);
            }

            System.out.println(map);

            System.out.println("\n SortedMap: \n");

            Map sortedMap = new TreeMap(map);
            System.out.println(sortedMap);
        }
}
```

Exercício Opcional (somente se houver tempo)
- Tempo previsto 3 horas:

Anotações

4) Estenda a classe *java.util.AbstractCollection* e faça sua própria classe de *collection*, faça *overriding* apenas nos métodos Métodos Básicos da *interface Collection* que estão na página 230. Para facilitar um pouco a tarefa, não há necessidade de diminuir a capacidade da *collection* após uma remoção de elemento.

Métodos Básicos da *interface Collection*:

Os métodos para adição e remoção de elementos são:

- boolean add(Object element)
Adiciona elementos

- boolean remove(Object element)
Remove elementos

A interface Collection também suporta as operações de query:

- int size()
Retorna o tamanho da collection

- boolean isEmpty()
Testa se a collection esta vazia

- boolean contains(Object element)
Verifica se um determinado elemento se encontra na collection

- Iterator iterator()
Retorna um objeto do tipo Iterator

Anotações

Gabarito:

```
import java.util.*;

public class MyBeautifulCollection extends AbstractCollection{
    private Object [] array = new Object[1];
    private Object [] array2 = new Object[1];
    private int counter;
    public static void main(String args[]) {
        MyBeautifulCollection my = new MyBeautifulCollection();

        System.out.println("Size: " + my.size()); //tamanho da collection

        System.out.println("Is empty? " + my.isEmpty()); //retorna true

        my.add("Look1"); //adiciona elemento
        my.add("Look2"); //adiciona elemento
        my.add("Look3"); //adiciona elemento
        my.add("Look4"); //adiciona elemento

        System.out.println("Size: " + my.size()); //tamanho da collection

        System.out.println(my.toString()); //testa overriding em toString()

        System.out.println("contain Look2 ? "+my.contains("Look2")); // true
        my.remove("Look2");
        System.out.println(my.toString()); //testa overriding em toString()

        System.out.println("Size: " + my.size()); //tamanho da collection

        System.out.println("contain Look2 ? "+my.contains("Look2")); // false

        System.out.println("Is empty? " + my.isEmpty()); //retorna false
        my.add("Look5"); //adiciona elemento
```

Anotações

```
        System.out.println(my.toString()); //testa overriding em toString()
    }

    private int getArrayInc(){
    return this.array.length + 1;
    }

private int getArrayDec(){
    return this.array.length - 1;
}

//Sobreescrevendo o método toString() da API
public String toString(){
    String string = "[";
    for (int i=0; i<array.length; i++){
        string += array[i] != null ? array[i] + ", ": "";
    }
    string = string.substring(0, string.lastIndexOf(','));
    string += "]";
    return string;
}

public boolean add(Object element){
    try{
        //Adiciona um elemento no último índice
        array [this.array.length - 1]= element;
        //Cria um novo array com um elemento a mais que o array antigo
        array2 = new Object[this.getArrayInc()];

    //Copia todos os elementos do array para o array2
    System.arraycopy(this.array, 0, array2, 0, this.array.length);

    array = array2;
```

Anotações

```
        counter++;

    }catch(ArrayIndexOutOfBoundsException e){
        return false;
    }
        return true;
    }

    //Remove o elemento, mas não diminue a capacidade da collection
    public boolean remove(Object element){
        try{
            array2 = new Object[this.getArrayDec()];

            for (int i=0; i<array.length; i++){
                array[i] = array[i] != null && array[i].equals(element) ? null :
                array[i];

        }
        }catch(ArrayIndexOutOfBoundsException e){
            return false;
        }
      return true;

    }

    public int size(){
        return this.array.length - 1;
    }
      public Iterator iterator(){
        ArrayList al = new ArrayList();

      for (int i=0; i<size(); i++){
        al.add(array[i]);
    }
```

Anotações

```
  return al.iterator();
}

public boolean isEmpty(){
        return size()==0 ? true : false;
}

public boolean contains(Object element){
    boolean contain = false;
    try{
        for (int i=0; i<array.length; i++){
            if (array[i] != null && array[i].equals(element)){
            contain = true;
            break;
                }
        }
    }catch(ArrayIndexOutOfBoundsException e){
                return false;
        }
    return contain;
  }
}
```

Anotações

Gabarito Laboratório 10:

Capítulo 11: *Threads*

1) Compile e rode o exemplo da página 264, que mostra a criação de *threads* com a opção de estender a classe *Thread*.

Gabarito:

```
class MyThread extends Thread {
	private String nome, msg;

	public MyThread(String nome, String msg) {
		this.nome = nome;
		this.msg = msg;
	}
	public void run() {
		System.out.println(nome + " iniciou !");
			for (int i = 0; i < 4; i++) {
			System.out.println(nome + " disse: " + msg);
			try {
				Thread.sleep(3000);
			}
			catch (InterruptedException ie) { }
		}//end for
		System.out.println(nome + " terminou de executar");
	}
}
public class StartThread {
	public static void main(String[] args) {
		MyThread t1 = new MyThread("thread1", "ping");
		MyThread t2 = new MyThread("thread2", "pong");
		t1.start();
		t2.start();
	}
}
```

Anotações

2) Compile e rode o exemplo da página 267, que mostra a criação de *threads* com a opção de implementar a classe *Runnable*.

Gabarito

```
class MyClass implements Runnable {
	private String nome;
	private A sharedObj;

	public MyClass(String nome, A sharedObj) {
		this.nome = nome;
		this.sharedObj = sharedObj;
	}
	public void run() {
		System.out.println(nome + " iniciou !");
		for (int i = 0; i < 4; i++) {
			System.out.println(nome + " disse: " + sharedObj.getValue());

			if (i==2) {
				sharedObj.setValue("mudou a string!");
			}
			try {
				Thread.sleep(3000);
			}
			catch(InterruptedException ie) { }
		}//end for
		System.out.println(nome + " terminou de executar");
	}
}

class A {
	private String value;
```

Anotações

```
        public A(String value) {
                this.value = value;
        }
        public String getValue() {
                return value;
        }
        public void setValue(String newValue) {
                this.value = newValue;
        }
}

public class StartThread2 {
        public static void main(String[] args) {
                A sharedObj = new A("algum valor");
                Thread t1 = new Thread(new MyClass("thread1", sharedObj));
                Thread t2 = new Thread(new MyClass("thread2", sharedObj));
                t1.start();
                t2.start();
        }
}
```

Anotações

3) Compile e rode o exemplo da página 273, sobre priorização de threads.

Gabarito

```
class PriorityTest {
    public static void main (String args[]) {
        Thread t1 = new RandomPrintString("Primeira");
        Thread t2 = new RandomPrintString("Segunda");
        Thread t3 = new RandomPrintString("Terceira");

        t1.setPriority(4);
        t2.setPriority(5);
        t3.setPriority(6);

        t1.start();
        t2.start();
        t3.start();

    }
}

class RandomPrintString extends Thread {
    public RandomPrintString(String str) {
                super(str);
    }
    public void run() {
                        for (int i = 0; i < 3; i++) {
                        System.out.println(getName());
                        //sleep((int)(Math.random() * 1000));
                        }
        //System.out.println(Thread.MIN_PRIORITY);  // 1
        //System.out.println(Thread.NORM_PRIORITY);// 5
        //System.out.println(Thread.MAX_PRIORITY); // 10
    }
}
```

Anotações

4) Compile e rode o exemplo da página 277, sobre sincronização de threads, no caso de produtor e consumidor

Gabarito

```
public class TestSync{
	int index;
	float [] data = new float[5];

	public synchronized float get() {
		index—;
		return this.data[index];
	}

	public synchronized void put(int value){
		this.data[index] = value;
		index++;
	}
}
```

Anotações

5) Compile e rode o exemplo da página 284, que mostra a criação de *ThreadGroup* e implementa a priorização em grupos de threads.

Gabarito

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

        ThreadGroup MyGroup = new ThreadGroup("Meu grupo");
        Thread MyGroupThread1 = new Thread(MyGroup,"Thread membro 1");
        Thread MyGroupThread2 = new Thread(MyGroup,"Thread membro 2");

        // configurando prioridade normal (5) para o grupo MyGroup
        MyGroup.setMaxPriority(Thread.NORM_PRIORITY);

        System.out.println("Group's priority = " + MyGroup.getMaxPriority());

        System.out.println("Numero de Threads no grupo = " + MyGroup.activeCount());

        System.out.println("Grupo Pai = " + MyGroup.getParent().getName());

        // Imprime as informações sobre o grupo
        MyGroup.list();

        }
}
```

Anotações

6) Crie um programa *multithread* que recebe dois nomes pela linha de comando, cria uma *thread* para cada nome e coloque ambas para dormir comum tempo aleatório, a *Thread* que acordar primeiro é a vencedora.

Gabarito

```
public class Ex {
    public static void main (String args[]) {
        Thread t1 = new MyThread(args[0]);
        Thread t2 = new MyThread(args[1]);

        t1.start();
        t2.start();

    }
}
class MyThread extends Thread {
    public MyThread(String str) {
        super(str);
    }
    public void run() {
        long time = new java.util.Random().nextInt(5000);
        System.out.println(getName() + " Vai tirar um cochilo de: " +
        time +" segundos");

        try{
                sleep(time);
                }catch(Exception e){ }

        System.out.println(getName() + " Venceu o jogo! ");
    }
}
```

Dica
Use o método *nextInt*(*int*) da classe *java.util.Random*(). Mas primeiramente você deve ler o que a **API** diz sobre este método, para melhor implementá-lo.

Anotações

Gabarito Laboratório 11:

Capítulo 12: Aplicações Gráficas com AWT

1) Compile e rode o exemplo da página 306.

Gabarito:

```
import java.awt.*;

public class FrameExample {
   private Frame f;

   public FrameExample() {
     f = new Frame("Hello Out There!");
   }

   public void launchFrame() {
     f.setSize(170,170); // Ajusta o tamanho do frame
     f.setBackground(Color.blue); // Pinta o fundo
     f.setVisible(true); // Torna o Frame visível
   }

   public static void main(String args[]) {
     FrameExample guiWindow = new FrameExample();
     guiWindow.launchFrame();
   }
}
```

Anotações

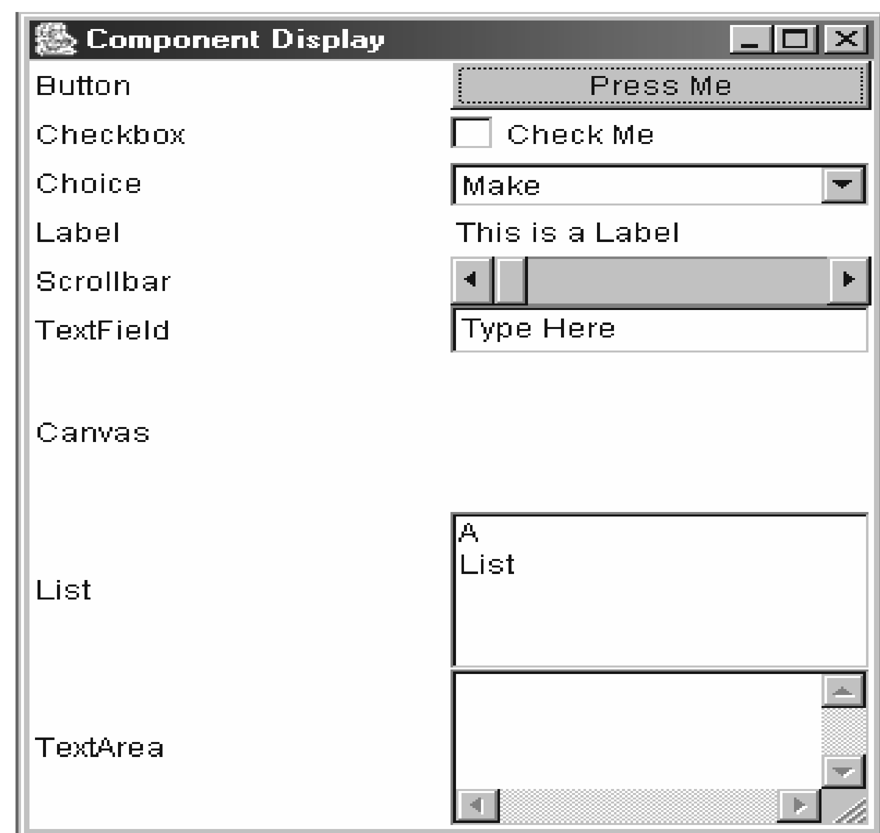
2) Compile e rode o exemplo da página 312.

Gabarito:

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

public class SampleTextArea {
   private Frame f;
   private TextArea ta;

   public void go() {
     f = new Frame("TextArea");
     ta = new TextArea("Hello!", 4, 30);
     f.add(ta, BorderLayout.CENTER);
     f.pack();
     f.setVisible(true);
   }

   public static void main (String args[]) {
     SampleTextArea sampleTextArea = new SampleTextArea();
     sampleTextArea.go();
   }
}
```

Anotações

3) Faça o código fonte para a tela abaixo:



Gabarito

```
import java.awt.*;

public class Ex extends Panel{

public void init(){
        setLayout(new BorderLayout());
```

Anotações

```
Panel p;

p = new Panel();

p.setLayout(new GridLayout(0, 2));

p.add(new Label("Button"));
p.add(new Button("Press Me"));

p.add(new Label("Checkbox"));
p.add(new Checkbox("Check Me"));

p.add(new Label("Choice"));
Choice c = new Choice();
c.addItem("Make");
c.addItem("A");
c.addItem("Choice");
p.add(c);

p.add(new Label("Label"));
p.add(new Label("This is a Label"));

p.add(new Label("Scrollbar"));
p.add(new Scrollbar(Scrollbar.HORIZONTAL));

p.add(new Label("TextField"));
p.add(new TextField("Type Here"));

add("North", p);

p = new Panel();

p.setLayout(new GridLayout(0, 2));
```

Anotações

```
        p.add(new Label("Canvas"));
        p.add(new Canvas());

        p.add(new Label("List"));
        List l = new List();
        l.add("A");
        l.add("List");
        p.add(l);

        p.add(new Label("TextArea"));
        p.add(new TextArea());

        add("Center", p);
    }

    public static void main(String [] args){
        Frame f = new Frame("Component Display");

        Ex ex = new Ex();

        ex.init();

        f.add("Center", ex);

        f.pack();
        f.setVisible(true);
    }
}
```

Anotações

Gabarito Laboratório 12:

Capítulo 13: Eventos

1) Compile e rode o exemplo da página 331.

Gabarito

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

public class TestWindow extends Frame implements WindowListener{

  public TestWindow(){

    this.addWindowListener(this);
    this.setVisible(true);
    this.setSize(300,300);

  }

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

  public void windowClosing (WindowEvent event) {
  System.exit(0);
  }

  public void windowActivated      (WindowEvent event) { }
  public void windowClosed         (WindowEvent event) { }
  public void windowDeactivated    (WindowEvent event) { }
  public void windowDeiconified    (WindowEvent event) { }
  public void windowIconified      (WindowEvent event) { }
  public void windowOpened         (WindowEvent event) { }
}
```

Anotações

2) Compile e rode o exemplo da página 332.

Gabarito

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

public class ActionCommandButton
                        implements ActionListener {
  private Frame f;
  private Button b;

  public void go() {
    f = new Frame("Sample Button");
    b = new Button("Sample");
    b.setActionCommand("Action Command Was Here!");
    b.addActionListener(this);
    f.add(b);
    f.pack();
    f.setVisible(true);
  }

  public void actionPerformed( ActionEvent ae) {
    System.out.println("Button press received.");
    System.out.println("Button's action command is: "+
        ae.getActionCommand());
  }

  public static void main (String args[]) {
    ActionCommandButton actionButton =
                                    new ActionCommandButton();
    actionButton.go();
  }
}
```

Anotações

3) Compile e rode o exemplo da página 333.

Gabarito

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

public class MyDialog implements ActionListener {

   private Frame f;
   private Dialog d;
   private Button db1;
   private Label dl;
   private Button b;

   public void go() {
     f = new Frame("Dialog");

     // Set up dialog.
     d = new Dialog(f, "Dialog", true);
     d.setLayout(new GridLayout(2,1));
     dl = new Label("Hello, I'm a Dialog");
     db1 = new Button("OK");
     d.add(dl);
     d.add(db1);
     d.pack();

     b = new Button("Launch Dialog");

     // Register listener for buttons.
     b.addActionListener(this);
     db1.addActionListener(this);
```

Anotações

```
    f.add(b, BorderLayout.CENTER);
    f.pack();
    f.setVisible(true);
  }
  // Handler for all buttons.
  public void actionPerformed( ActionEvent ae) {
    String buttonPressed = ae.getActionCommand();
    if (buttonPressed.equals("Launch Dialog")) {
        d.setVisible(true);
    } else if (buttonPressed.equals("OK")) {
    System.out.println ("Process terminated!!!");
    System.exit(0);
    } else {
    d.setVisible(false);
    }
  }

  public static void main (String args[]) {
    MyDialog myDialog = new MyDialog();
    myDialog.go();
  }
}
```

Anotações

4) Compile e rode o exemplo da página 336.

Gabarito

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

public class SampleMenu
   implements ActionListener, ItemListener {
   private Frame f;
   private MenuBar mb;
   private Menu m1;
   private Menu m2;
   private Menu m3;
   private MenuItem mi1;
   private MenuItem mi2;
   private MenuItem mi3;
   private MenuItem mi4;
   private CheckboxMenuItem mi5;

   public void go() {
     f = new Frame("Menu");
     mb = new MenuBar();
     m1 = new Menu("File");
     m2 = new Menu("Edit");
     m3 = new Menu("Help");
     mb.add(m1);
     mb.add(m2);
     mb.setHelpMenu(m3);
     f.setMenuBar(mb);

     mi1 = new MenuItem("New");
     mi2 = new MenuItem("Save");
```

Anotações

```
  mi3 = new MenuItem("Load");
  mi4 = new MenuItem("Quit");
  mi1.addActionListener(this);
  mi2.addActionListener(this);
  mi3.addActionListener(this);
  mi4.addActionListener(this);
  m1.add(mi1);
  m1.add(mi2);
  m1.add(mi3);
  m1.addSeparator();
  m1.add(mi4);

  mi5 = new CheckboxMenuItem("Persistent");
  mi5.addItemListener(this);
  m1.add(mi5);

  f.setSize(200,200);
  f.setVisible(true);
}

public void actionPerformed( ActionEvent ae) {
  System.out.println("Button \"" +
     ae.getActionCommand() + "\" pressed.");

if (ae.getActionCommand().equals("Quit")) {
  System.exit(0);
  }
}

public void itemStateChanged(ItemEvent ie) {
  String state = "deselected";

if(ie.getStateChange() == ItemEvent.SELECTED) {
```

Anotações

```
        state = "selected";
      }
      System.out.println(ie.getItem() + " " + state);
    }

    public static void main (String args[]) {
      SampleMenu sampleMenu = new SampleMenu();
      sampleMenu.go();
    }
  }
```

Anotações

5) Implemente a *interface WindowListener* nos exercícios anteriores para que o **GUI** possa ser fechado através do botão fechar. Use o código da página 331 como exemplo.

Gabarito

Basta apenas aproveitar o mesmo código da página 331 e fazer o comando abaixo em todas os *Frames*.

Frame. addWindowListener (TestWindow());

Não há necessidade de digitar mais nada para o fechamento da janela funcionar através do botão fechar do *frame*.

Anotações