Java 2 Certified Programmer

Enviada por:

Alex Sandro de Lima

Java 2 Certified Programmer

SUN Java 2 Certified Programmer

O que você **pode fazer** com este texto:

- Copiar e distribuir gratuitamente ou comercialmente.
- Alterar e criar novas versões, desde que sigam a licença GNU Free Documentation License e mantenham as informações do copyright do autor original.

O que você **não pode fazer** com este texto:

- Alterar as mensagens de copyright e autoria do mesmo. Respeite o direito autoral.
- Distribuir este texto sem os arquivos de exemplo e sem a licença de distribuição GNU.

Este curso é acompanhado de um diretório “exemplos” contendo 09 (nove) subdiretórios. Cada um deles contém vários programas-exemplo em Java. Todos fazem parte do trabalho e estão contemplados com a licença GNU.

SUMÁRIO

Objetivos do Exame

O exame para obter a certificação SUN Certified Java 2 Programmer é baseado em uma série de perguntas versando sobre os temas:

1. Language Fundamentals (Fundamentos da Linguagem Java 2)
2. Operators and Assignments (Operadores e atribuição de valores)
3. Modifiers (Modificadores)
4. Convert and Casting (Conversão e transformação de tipos de dados)
5. Flow Control, Exceptions and Assertions (Controle de Fluxo, Exceções e Assertivas de depuração)
6. Objects and Classes (Objetos e Classes)
7. Threads (Segmentos)
8. Java.Lang and Java.Util Packages (Pacotes Java.Lang e Java.Util)
9. Input and Output (Entrada e Saída de dados)

Você deverá adquirir um "voucher" para prestar o exame na própria SUN (www.sun.com.br), escolhendo um Test Provider para efetuar seu exame.

Um exemplo do tipo de pergunta que você poderá ver na prova é:

"Analise o seguinte trecho de código Java e selecione a alternativa correta:

```
import java.io.*;

public class Cliente implements Serializable {
        private int codigo;
        int tipo;
        public String nome;
        private transient int senha;
}
```

A – "Código" é uma propriedade da Classe "Serializable"
B – A variável "tipo" pode ser acessada a partir de qualquer classe descendente desta
C – Não se pode ter Objetos como propriedades de qualquer classe
D – A variável "senha" não será serializada
E – A classe não compilará"

Neste caso, a resposta correta é a letra "D".

A estratégia para passar no exame é estudar detalhadamente cada capítulo e fazer alguns exercícios simulados. Junto com este material você receberá algumas perguntas para exercitar o que aprendeu no texto. Sugerimos que você faça e refaça os exercícios antes de marcar seu exame. Outra fonte para obter exames simulados é a empresa "MeasureUp", em seu web site: www.measureup.com.

Apesar de existirem alguns exames em português, os mais recentes sempre serão em inglês.

Obtendo o JDK

Uma das boas táticas de sucesso é fazer os exercícios e, para isto, você necessitará do Java 2 developer kit, ou JDK.

O JDK inclui um compilador de linha de comando chamado "javac". Você pode obter o JDK a partir do Site oficial do Java: www.java.sun.com. Baixe o pacote e instale em sua plataforma (Windows, Linux, Solaris etc).

Após instalar o JDK coloque o diretório C:\j2sdk1.x.x_0x\bin na sua variável "path". Assim você poderá invocar os programas "**javac**", "**java**" e "**jar**" de qualquer diretório.

Para testar se tudo está ok, digite o seguinte código em Java utilizando um editor de textos:

```
import java.util.*;

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

1. Salve o arquivo como "testeargs.java"
2. Abra um prompt de comandos (ou janela Terminal) e compile com o comando: "**javac** testeargs.java".
3. Não são esperados erros. Se houver, pode desconfiar de algum problema na instalação.
4. Execute a classe com o comando: "java testeargs banana laranja abacaxi"

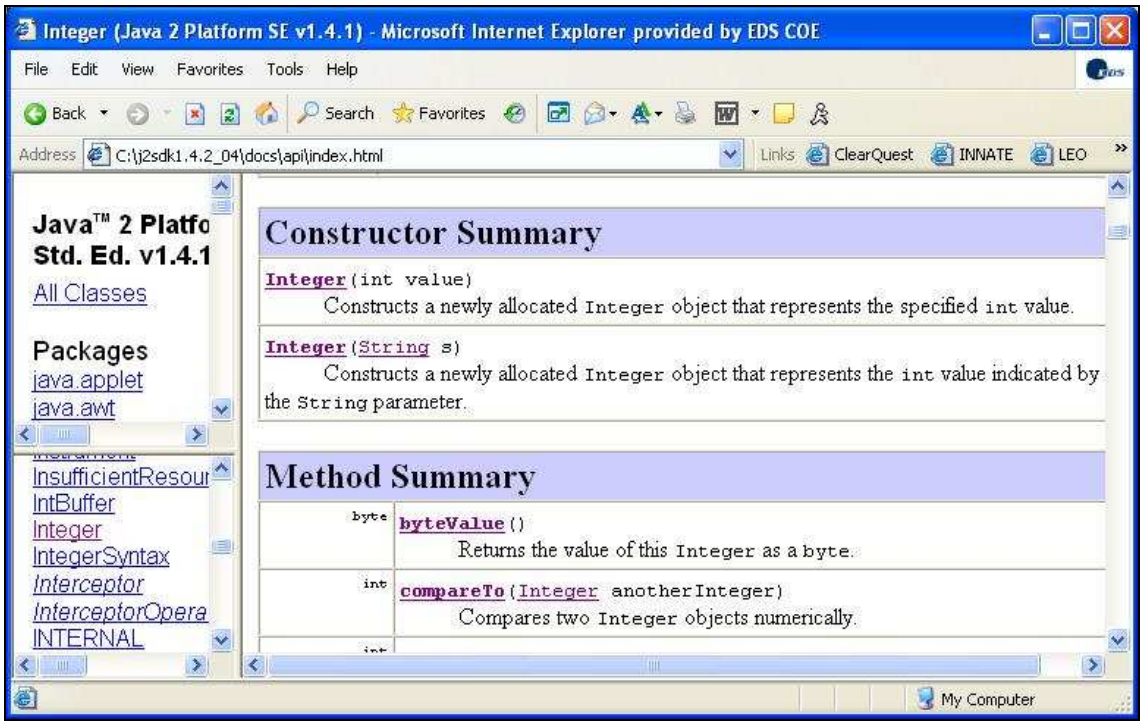
O que você precisa para se beneficiar deste material

Espera-se que você já conheça alguma linguagem de programação, além de algoritmos e estruturas de dados. É bom também conhecer Orientação a Objetos, pois Java é totalmente OOP.

Documentação extra

O website http://www.java.sun.com é a referência oficial para o Java. Você pode, inclusive, baixar a documentação da API em formato HTML, padrão "Javadoc", contendo todas as classes de todos os pacotes do Java.

Recomendo expressamente que, ao ler sobre uma nova classe ou método, você pesquise na documentação da API por maiores detalhes sobre o assunto.



É muito importante que você se familiarize com as classes e métodos. Estude as exceptions que eles levantam, os valores de retorno etc.

Um assunto final é sobre IDE (Integrated Development Environment). Existem várias IDEs para Java, muitas delas gratuitas, como a Eclipse (http://www.eclipse.org), mas para desenvolver em Java você não precisa de nada além do JDK e de um editor de arquivos.

Eu recomendo que neste início você só use Notepad (ou o equivalente do Linux), compilando os seus programas com o comando "javac". Se precisar utilizar classes que estão em outros packages você pode usar o "javac –cp <classpath> arquivo.java".

Fundamentos da Linguagem Java

Neste capítulo serão explicados os fundamentos do Java2. Arquivos-fonte, variáveis e outros detalhes serão vistos. Os exemplos estão no path “exemplos\01”.

Classes

Todo programa Java consiste na definição de uma ou mais classes. Em outras palavras, você define uma classe e, posteriormente, a instancia para executar seus métodos.

Só para relembrar, a OOP preconiza a existência de Classes, que são os gabaritos para criação dinâmica de Objetos, operação conhecida como “instanciamento”.

Uma Classe contém Membros que pertencem a ela. Os Membros podem ser privativos da Classe ou Públicos. Se forem Públicos, podem ser acessados por outros programas que instanciem Objetos da Classe.

Um exemplo:

```
public class Aluno {
     int matricula;
     String nome;
     String endereco;

     Aluno(int nMatricula)
     {
          this.matricula = nMatricula;
     }

     public static void main(String[] args)
     {
          Aluno A1 = new Aluno(Integer.parseInt(args[0]));
          A1.nome = args[1];
          A1.endereco = args[2];

          System.out.println("Nome: " + A1.nome +
                         " endereco: " + A1.endereco +
                         " matricula: " + A1.matricula);
     }
}
```

Neste exemplo temos uma Classe Pública, chamada “Aluno”, que possui 5 membros: três propriedades (“matricula”, “nome” e “endereco”) e dois métodos (“Aluno” e “main”). O prefixo **“this”** antes da variável “matrícula” indica que é **DESTA** classe. Em Java o valor de retorno de um método deve ser declarado imediatamente ANTES do nome do mesmo. Se ele nada retorna, deve declarar **“void”**.

Digite o programa anterior (ou abra de “exemplos\01\Aluno.java”) e compile-o:

1. Abra uma janela de Prompt de comandos.
2. Vá para o diretório do curso (“exemplos\01”).
3. Compile o programa: “**javac** Aluno.java”.
4. Execute o programa: “**java** Aluno 1234 jose rua”.

Se quiser informar nome ou endereço com espaços, informe-os entre aspas duplas.

Observações:

- Java é sensível a maiúsculas e minúsculas. Se você criou a classe como “Aluno” (com “A” maiúsculo) deve roda-lo no “java” como “Aluno”, caso contrário dará um erro de Exceção (Exception).
- Um arquivo que contenha uma classe Pública deverá ter o mesmo nome dessa classe pública, com maiúsculas e minúsculas.
- O Método “main” é declarado como “static” ou estático e é ele que cria a instância da classe definida no arquivo. Ele nada retorna, logo é declarado como **“void”.** Sintaxe de declaração de método:
  `[modificador] [retorno] nome_metodo([argumentos]) {[comandos]}`

Arquivo fonte em Java

Um arquivo fonte em Java pode conter uma Classe Pública, que deve lhe dar seu nome principal.

Você pode criar arquivos com classes privadas ou arquivos com classes que serão invocadas a partir de outras classes.

Se quiser criar um programa “stand alone”, ou seja, que pode ser executado a partir da linha de comando, deve criar o arquivo com o mesmo nome (sem contar a extensão) da classe Pública que ele contém.

Assim, quando chamar a máquina virtual do Java (através do comando “java”) ele saberá procurar o arquivo e qual classe deve ser criada. Na verdade, ele chama o método “main” da classe.

O seguinte arquivo contém um erro:

Arquivo: “Nanobot.java”:

```
public class NanoBot {
     public int id;
     public String nome;
}
```

Qual é o erro?
A – faltou o comando “import”
B – A classe não contém um método “main”

C – O tipo de dados “String” não existe, o correto seria “string”
D – Nenhuma das alternativas anteriores
A letra correta seria a “D”, pois o erro está no nome do arquivo: “Nanobot” (sem a extensão), que é diferente do nome da classe pública que contém (“NanoBot”).

Prefixos “this” e “super”

O prefixo “this” é como se fosse um “ponteiro” que indica ESTA classe. Podemos utilizá-lo para diferenciar propriedades e métodos DESTA classe das propriedades e métodos de sua ANCESTRAL.

Se a classe ANCESTRAL tiver métodos ou propriedades com mesmo nome (ou assinatura, no caso de método) podemos indicar se queremos o desta classe ou da ancestral com o prefixo “this”.

Já o prefixo “super” indica a classe ANCESTRAL desta. Se tivermos propriedades ou métodos iguais, ao utilizar “super” estamos indicando que queremos acessar os membros da classe ANCESTRAL.

Veja o exemplo “this_e_super.java”:

```
class mae {
     public int x;
     void mostrar() { // nada retorna e nada recebe
          System.out.println("\nDa mae: " + x);
     }
}
class filha extends mae {
     public int x;
     void mostrar() {
          System.out.println("\nDa filha: " + x);
     }
     void alterar(int y) {
          this.x = y;
          super.x = y + 1;
          this.mostrar();
          super.mostrar();
     }
}

public class this_e_super {
     public static void main(String args[]) {
          mae m = new mae();
          filha f = new filha();
          f.alterar(5);
     }
}
```

Elementos de um arquivo fonte

Um arquivo fonte pode conter três elementos básicos:

- Declaração de pacote (package)
- Comando para importar código fonte (import)
- Definições de classes (class)

Não é obrigatório que os três elementos existam, mas, se ocorrerem, devem vir exatamente nesta ordem: “package”, “import” e “class”.

Criando Packages

O Java foi criado para ser multiplataforma, logo, deve lidar com diferentes tipos de estrutura de arquivos (File Systems).

Quando você cria um projeto costuma juntar todos os arquivos-fonte em um só pacote, que pode estar em um só diretório no disco. Os arquivos-fonte de um pacote, normalmente, possuem as mesmas permissões de acesso.

Em Java todas as classes cujos arquivos ficam em um mesmo diretório pertencem a um mesmo “pacote” de aplicativo.

Se quiser pode incluir um comando indicando que o programa pertence a um pacote de aplicativos, isto ajudará a identificar o sistema ao qual pertence. Considere o exemplo:

```
package Cleuton.teste;

import java.util.*;

public class abc {
     private int m = 0;
     public int retorna() {
          return m;
     }
}
```

O que é um “package”?

É um conjunto de programas que pertencem a uma mesma unidade de projeto. Essa unidade é ditada por duas características: compartilham o mesmo diretório físico e/ou possuem um comando “package”.

O comando “package” indica que este programa faz parte de um aplicativo e deve ser colocado no diretório apropriado. Neste caso o programa deve estar no diretório: “Cleuton\teste”.

Todos os programas que estão em um diretório, tenham ou não o comando “package”, fazem parte de um pacote. Quando informamos o “package” estamos reforçando esta situação e permitindo ao compilador detectar se o programa está no diretório correto.

Note que a estrutura de diretórios em Java é ditada pelo parâmetro “classpath”. Se você possui uma variável de ambiente “classpath”, então o Javac irá procurar a classe dentro destes diretórios.

No exemplo anterior o package é “Cleuton.teste”. Se você tem uma variável de ambiente “classpath” o Javac irá procurar a classe dentro de um diretório chamado “\Cleuton\teste”, dentro de todos os diretórios especificados no “classpath”.

Se você não tem uma variável de ambiente “classpath”, pode especificar nos parâmetros do Javac. Supondo que o diretório físico seja: “C:\Cleuton\teste”, então o “classpath” deverá ser: “C:\”.

Classes que residem no mesmo “package” não necessitam ser importadas. Os arquivos “abc.java” e “usaabc.java” estão no mesmo “package” e no mesmo diretório:

```
package Cleuton.teste;

import java.util.*;

public class abc {
     private int m = 0;
     public int retorna() {
          return m;
     }
}

package Cleuton.teste;

import java.util.*;

public class usaabc {
     public void metodo() {
          abc t = new abc();
          System.out.println(t.retorna());

     }
     public static void main(String args[]) {
          usaabc xpto = new usaabc();
          xpto.metodo();
     }
}
```

Note que no arquivo da classe “usaabc” não estamos usando o comando “import” para o Package.

O problema de criar classes Públicas com método “main” dentro de packages é que na hora de rodar temos que informar o nome completo:

```
“java –cp c:\ Cleuton.teste.usaabc”
```

A SUN recomenda que se crie packages utilizando o nome de domínio (DNS) da empresa ao contrário. Por exemplo: “cleuton.com.Br” é o meu domínio e “teste” o nome do meu package. Então o correto seria:

```
“package br.com.cleuton.teste“
```

Importando arquivos

Quando criamos um sistema, normalmente, utilizamos diversos componentes, localizados em arquivos separados.

É uma forma de criarmos bibliotecas de código separadas, a partir das quais podemos instanciar objetos.

O procedimento é simples:

1. Crie um pacote (“package”) colocando os arquivos fonte no mesmo diretório
2. Compile cada arquivo criando sua classe (arquivo “.class”)
3. No programa que desejar utilizar uma determinada classe, informe o comando “import”.

Para utilizar uma classe que está dentro de um “package” temos que importa-la para o nosso programa com o comando “import”:

```
“import <package>.<classe>”
```

É comum importarmos alguns “packages” em nossos programas, dependento do que vamos fazer:

```
“import java.lang.*”
“import java.util.*”
```

Quando especificamos “*” no final do “import”, estamos dizendo ao Java para importar TODAS as classes pertencentes ao pacote.

Nomes em Java

Os nomes em Java podem começar por uma letra, um caracter sublinha "_" ou por um cifrão. Todos os nomes são sensíveis a maiúsculas e minúsculas (case-sensitive).

Um nome deve começar por letra, "_" ou "$" e pode conter letras, algarismos, "_" ou "$". Exemplos de nomes válidos:

- $Dollar
- $1
- _345
- Teste
- Teste

Note que "Teste" e "teste" são nomes diferentes em Java.

Palavras-chave

Segundo o manual do Java nenhum identificador (nome) pode ser igual a uma das palavras-chave:

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

O manual do Java afirma que:

"*The keywords `const` and `goto` are reserved, even though they are not currently used. This may allow a Java compiler to produce better error messages if these C++ keywords incorrectly appear in programs.* "

Tipos primitivos

Os tipos primitivos são associados às variáveis comuns em Java. O nome "primitivo" serve para diferenciar das classes. Uma variável "primitiva" não é uma referência para um Objeto, mas para um lugar na memória (stack ou heap).

Os seguintes tipos primitivos existem em Java (traduzido do manual do Java):

Números inteiros:

- `byte`, de –128 (-2^7) a 127 ($2^7 - 1$), inclusive (8 bits)
- `short`, de –32768 (-2^{15}) a 32767 ($2^{15} - 1$), inclusive (16 bits)
- `int`, de –2147483648 (-2^{31}) a 2147483647 ($2^{31} - 1$), inclusive (32 bits)
- `long`, de –9223372036854775808 (-2^{63}) a 9223372036854775807 ($2^{63} - 1$), inclusive (64 bits)
- `char`, de `'\u0000'` a `'\uffff'` inclusive, ou seja, de 0 a 65535 (16 bits)

Note que o tipo “char” é o único sem sinal.

Números reais:

- float, com 32 bits
- double, com 64 bits

Ambos sinalizados

Lógico

- boolean, não sinalizado, 1 bit, valores “true” ou “false”

Valores não numéricos reais

As variáveis “float” e “double” possuem situações onde um número não pode ser representado. Elas são “NaN” – Not a Number, “NEGATIVE_INFINITY” – Menos Infinito e “POSITIVE_INFINITY” – Mais Infinito.

Podemos saber se uma variável possui NEGATIVE_INFINITY ou POSITIVE_INFINITY com as funções:

- Float.POSITIVE_INFINITY
- Float.NEGATIVE_INFINITY
- Double.POSITIVE_INFINITY
- Double.NEGATIVE_INFINITY

Mas não podemos usar o valor “Float.NaN” ou “Double.NaN” para efetuar comparações. Para isto temos a função “Float.isNaN” ou “Double.isNaN”.

Comentários

Em Java podemos colocar comentários com duas barras "//" se forem em uma só linha:

```
// classe de teste
```

Se o comentário for maior que uma linha podemos inseri-lo entre "/*" e "*/":

```
/* este programa
     é fácil */
```

Final de comando

O Ponto-E-Vírgula deve encerrar todos os comandos em Java. Exemplo:

```
X = 123;
```

Variáveis

Podemos declarar variáveis informando o tipo e o nome, seguidos por ponto-e-vírgula:

```
int xTeste;
char nome;
long Valor;
long valor;
```

Literais

Um literal é a representação de um valor discreto em um programa. Os literais podem ser atribuídos às variáveis, impressos, gravados em disco ou utilizados em comparações.

Existem formas de representar literais para cada tipo de dados em Java:

- Inteiros: 5, 17, 32, 0xED, **todos "int"** (32 bits).
- Inteiros longos (64 bits): 100L, 0x1ab3L.
- Reais (32 bits): 1.44F.
- Duplos (64 bits): 126.77 ou 126.77D (sem nada o **default é "double"**).
- Booleanos: true ou false (sem aspas).
- Char (16 bits): 'x', '\n'
- Nulo: null.

Os literais caracter podem possuir seqüências de escape que permitem introduzir caracteres especiais:

```
EscapeSequence:
  \b                    /* \u0008: backspace BS */
  \t                    /* \u0009: horizontal tab HT */
  \n                    /* \u000a: linefeed LF */
  \f                    /* \u000c: form feed FF */
  \r                    /* \u000d: carriage return CR */
  \"                    /* \u0022: double quote " */
  \'                    /* \u0027: single quote ' */
  \\                    /* \u005c: backslash \ */
  OctalEscape           /* \u0000 to \u00ff: from octal value */
```

(do manual do Java)

Literais String

Embora “String” não seja um tipo de dados primitivo em Java, a linguagem está preparada para reconhecer literais string, como: “Isto é um teste\nABCD”.

O tipo de dados String é fornecido pela classe “String”:

```
String nome = “Cleuton”.
```

Arrays

Um Array é uma lista seqüencial de valores de um mesmo tipo. Você pode ter Arrays de qualquer tipo primitivo, referências a Objetos ou a outros Arrays.

Só há uma restrição: todos os elementos do Array tem que ser do mesmo tipo.

Para definir um Array:

```
int[] notas;
```

Inicializar o Array:

```
notas = new int[10];
```

Para usar um Array:

```
notas[4] = 10; // quinta nota do array–começa em zero
```

Todo Array começa com a posição ZERO. O primeiro elemento é ZERO e o último é o número de elementos menos 1. Por exemplo:

```
char[] tipos;
tipos = new char[7];
```

O primeiro elemento é ZERO e o último é 6.

Um Array pode ter mais de uma dimensão:

```
int[][] matriz;
matriz = new int[3][10];
matriz[0][7] = 12;
```

Podemos inicializar um Array dentro da própria definição:

```
int[] notas = {5, 7, 6, 10, 8, 2};
```

Programas com Classes Públicas

Em Java um programa aplicativo é um arquivo que contém uma Classe Pública com um método estático “main”.

Desta forma é possível utilizar o Java VM para executar a classe. Por exemplo:

```
public class Aluno {
     int matricula;
     String nome;
     String endereco;

     Aluno(int nMatricula)
     {
          this.matricula = nMatricula;
     }

     public static void main(String[] args)
     {
          Aluno A1 = new Aluno(Integer.parseInt(args[0]));
          A1.nome = args[1];
          A1.endereco = args[2];

          System.out.println("Nome: " + A1.nome +
                             " endereco: " + A1.endereco +
                             " matricula: " + A1.matricula);
     }
}
```

Este arquivo possui uma Classe Pública chamada “Aluno” (Seu nome é “Aluno.java”). Como ela possui um método estático “main”, pode ser chamada a partir do Java VM (“java <classe>”).

Os métodos são as funções que uma classe possui. Eles são diferenciados entre si pela sua Assinatura, que é o conjunto composto pelo seu:

1. nome
2. tipo e ordem dos argumentos
3. valor de retorno

A assinatura padrão do método “main” que o Java invoca é:

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

Note que a assinatura não inclui o nome dos argumentos, mas apenas o seu tipo e ordem. Logo, as seguintes assinaturas são válidas para o Java VM:

- “public static void main(String tipos[])”
- “public static void main(String[] argumentos)”

Ao executar a classe Alunos utilizamos o Java VM:

```
“java Alunos 1234 cleuton parque”
```

O primeiro argumento para o Java VM é o nome da classe (é case-sensitive). Depois informamos os argumentos que devem ser passados para dentro do método “main”, na ordem correta. Neste caso:

```
args[0] recebe “1234”
args[1] recebe “cleuton”
args[2] recebe “parque”
```

O Java VM vai invocar o método “main” da classe o qual, se espera, crie uma instância dela:

```
          Aluno A1 = new Aluno(Integer.parseInt(args[0]));
```

Inicialização e Escopo das variáveis

Uma variável pode ser Membro de uma classe ou privativa a um método. Exemplos:

```
public class teste {
     public int codigo;
     public void marca() {
          int x;
     }
}
```

Neste caso a variável “codigo” é membro da classe “teste” e a variável “x” é local ou privativa do método “marca”.

As variáveis Membros são criadas quando um Objeto é instanciado e destruídas quando ele é destruído. As variáveis locais ou Automáticas são criadas quando o método é chamado e destruídas quando a execução do método acaba.

Todas as variáveis membros que não forem inicializadas recebem um valor padrão:

- byte: 0
- int: 0
- short: 0
- long: 0L
- float: 0.0F
- double: 0.0D
- char: \u0000
- boolean: false
- referência a objetos: null

Assim como as variáveis Membros, os elementos de um Array também são inicializados com os mesmos valores.

Variáveis automáticas não são inicializadas automaticamente. Uma boa opção para evitar que o compilador reclame é inicializá-las na sua definição:

```
int numero = 0;
```

Referências de Objetos

Uma variável pode ser apenas um ponteiro para a instância de uma Classe. Suponha o seguinte código:

```
public class Aluno {
     int matricula;
     String nome;
     String endereco;

     Aluno(int nMatricula)
     {
          this.matricula = nMatricula;
     }

     public static void main(String[] args)
     {
          Aluno A1 = new Aluno(Integer.parseInt(args[0]));
          A1.nome = args[1];
          A1.endereco = args[2];
```

```
            System.out.println("Nome: " + A1.nome +
                               " endereco: " + A1.endereco +
                               " matricula: " + A1.matricula);
        }
}
```

Qual é o tipo primitivo da variável “A1”? Nenhum! Ela não é uma variável primitiva, mas uma referência para uma instância da classe “Aluno”! Podemos dizer, a grosso modo, que uma referência é um endereço de Objeto na memória.

As referências, quando declaradas, sempre contém null. Para que apontem para um Objeto válido precisamos cria-lo com o comando “new”, como na linha:

```
            Aluno A1 = new Aluno(Integer.parseInt(args[0]));
```

Será criada uma nova instância da Classe Aluno e seu endereço apontado por “A1”. Podemos chamar métodos ou propriedades do novo objeto através da variável “A1”.

Se atribuirmos o mesmo Objeto a duas variáveis, as duas apontarão para ele:

```
        Aluno A1 = new Aluno(5);
        Aluno A2 = A2;
```

Métodos e argumentos

Na assinatura de um método pode haver argumentos sendo passados para ele. O Java sempre passa POR VALOR.

Existem duas maneiras básicas de se passar um argumento: por valor ou por referência. Quando passamos por valor, uma cópia dos dados é criada e passada para o método. Essa é a maneira que o Java faz. Por referência é passado o endereço da variável original.

Em MS Visual Basic isto pode ser feito com as opções “By Val” ou “By Ref”.

Em Java apenas existe o “By Val”.

Suponha o método:

```
public long calcular(int numero, int fator, long ultprod) {
        ultprod = numero * fator;
        return ultprod;
}
```

Agora vamos ver a chamada deste método:

```
        int x  = 5;
        int y = 3;
```

```
long z = 0;
long t = 0;

t = calcular(x,y,z);
```

Qual será o valor de “z” após rodar a chamada da função? A resposta é ZERO. Apenas o valor de “z” foi passado para a função “calcular”.

Quando passamos Referências de Objeto para um Método é passada uma cópia do seu endereço. Não podemos alterar o endereço do Objeto. Por exemplo:

```
Aluno A1 = new Aluno(123);

calcular(A1);

void calcular(Aluno x) {
     x = new Aluno(13);
}
```

Apesar da função alterar a referência apontada por “x”, ao retornar, a variável “A1” ainda estará apontando para o aluno 123 e não para o 13.

Porém podemos alterar propriedades do Objeto:

```
Aluno A1 = new Aluno(123);
A1.nome = “José”;

calcular(A1);

void calcular(Aluno x) {
     x.nome= “Maria”;
}
```

Ao retornar, a variável “A1” ainda estará apontando para o Aluno 123, só que seu nome agora é “Maria”.

Coleta de Lixo

Para finalizar esta parte falta falar sobre Garbage Collection ou Coleta de Lixo.

Quando você cria referências, espaço em memória é alocado a elas. Quando não as utiliza mais, este espaço precisa ser liberado.

Em Java isto é responsabilidade do Garbage Collector, um módulo de baixa prioridade que vai liberando memória de objetos que não estão mais em uso.

A liberação não é determinística, ou seja, ao atribuirmos null a uma referência de objeto ela não será imediatamente liberada. Quando o GC rodar liberará memória de objetos em desuso.

Questões e Exercícios:

1) Considere o seguinte código em Java:

```
Arquivo: teste11.java
-------------------------
1> import java.io.*;
2> public class teste11 {
3> int serializable;
4> }
5> package teste;
```

a) A classe compilará mas dará exception ao ser executada.
b) Dará erro na linha 3 porque “serializable” é uma palavra reservada.
c) Não compilará porque falta o construtor.
d) Não compilará porque a ordem dos elementos está errada.

2) Considere o seguinte código em Java:

```
Arquivo: teste12.java
-------------------------
1> import java.io.*;
2> private class teste12 {
3>   public static void main(String args[]) {
4>        System.out.println("OLA");
5>   }
6> }
```

a) Dará erro na linha 2 porque falta a especificação do “package”.
b) Compilará normal e rodará normal.
c) Compilará normal e dará exception ao rodar.
d) Dará erro de compilação.

3) Considere o seguinte código do arquivo “teste13.java”:

```
Arquivo: teste13.java
-------------------------
1> import java.io.*;
2> public class Instanceof {
3>   char _29382938232932;
4>   public static void main(String args[]) {
5>        System.out.println("OLA");
6>   }
7>   public int $main() {
8>        return 2;
9>   }
10> }
```

a) Dará erro de compilação por uso de palavra reservada.
b) Está com erro de sintaxe nas linhas 2 e 3.
c) Compilará e rodará sem problemas.
d) Haverá um erro de compilação sem ser por uso de palavra reservada.

Respostas:

1) A resposta correta é a letra “D”. A diretiva “package” está fora de ordem.

2) A resposta é “D”, pois você não pode ter o modifier “private” em classes top level. Elas podem ser “default” ou “public”.

3) A resposta é “D”. Dará erro de compilação porque a classe “Instanceof” é pública, logo, tem que ficar em um arquivo “Instanceof.java”.

Operadores, Atribuição e Avaliação

Neste capítulo serão mostrados os principais Operadores em Java, além de regras de avaliação e atribuição de expressões.

Os exemplos deste capítulo estão no diretório: “exemplos/02”.

Atribuição

Em Java podemos alterar o conteúdo de uma variável através da Atribuição (Assignment). Esta operação é demarcada pelo uso do operador “=”. Veja alguns exemplos:

```
int x;
x = 5;
long z = 107L;
```

Repare que podemos declarar uma variável e depois atribuir um valor ou então atribuir no momento em que declaramos. Isto é útil porque já inicializamos as variáveis. Lembre-se que variáveis automáticas (declaradas em módulos) não são inicializadas automaticamente.

Atribuição incremental

Um tipo especial de atribuição é quando desejamos atribuir a uma variável o seu próprio valor incrementado. Exemplo:

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

Podemos escrever:

```
x += 1;
```

Estas operações tomam a forma “<variável> op= <incremento/variável2>”, veja o exemplo “atrib”:

```
public class atrib {
     static int x; // inicializadas automaticamente
     static int y;
     public static void main(String[] args) {
          int a = 0; /* variável automática tem que ser
inicializada */
          int b = 0;
          int c = 0;
          a = a + 1;
          b += 1;
          c *= 2;
          x -= 10;
          y = 10;
          y /= 2;
          System.out.print("a = " + a + '\n' +
                    "b = " + b + '\n' +
                    "c = " + c + '\n' +
                    "x = " + x + '\n' +
                    "y = " + y + '\n');
     }
}
```

Os resultados deste programa são

```
C:\Cleuton\Aulas\Java2Prog\Exemplos\02>java atrib
a = 1
b = 1
c = 0
x = -10
y = 5
```

Atribuição múltipla

Em Java podemos fazer a seguinte atribuição:

```
int x;
int z = 5;
int y;

z = x = y;
```

Ordem de avaliação de expressões

As expressões em Java são sempre avaliadas da esquerda para a direita, independente da sua ordem de execução. Considere o exemplo:

```
int[] codigo = {1,5,7,9};
int posic = 1;
codigo[posic] = posic;
```

Na terceira linha a expressão mais à direita é avaliada primeiro. Neste caso, ele começará a avaliar “codigo[posic]” em primeiro lugar, resultando no valor “codigo[1]" que é "5", depois ele avalia a expressão “posic”, à direita do sinal de atribuição. Logo, o segundo valor do Array será alterado para “1”.

O problema de avaliação começa quando digitamos expressões como esta:

```
codigo[posic] = posic = 0;
```

Qual é o elemento do array que será alterado? O primeiro? Vamos avaliar:

1. Primeiramente será valiada a expressão mais à esquerda; codigo[posic], que virará codigo[1].
2. Depois será avaliada a expressão do meio: posic.
3. Finalmente será avaliada a expressão do final: ZERO.

Quando a expressão for executada o segundo elemento do Array receberá o valor ZERO, que foi atribuído à variável “posic”.

A ordem de execução depende da precedência dos operadores envolvidos. No caso de atribuição esta ordem começa da direita para a esquerda.

Operadores Unários

Os operadores unários são aplicados sobre variáveis individuais segundo a tabela:

++x ou x++	Incrementa o valor de “x”
--x ou x--	Decrementa o valor de “x”
+x ou –x	Mudam o sinal de “x”
~x	Inversão de bits (0 para 1 e 1 para 0)
!x	Negação lógica (só para boolean)

Note-se que a função dos operadores “~” e “!” é a mesma, a diferença é que o operador “!” é exclusivo do tipo “boolean”.

Os operadores de incremento e decremento funcionam de maneira diferente dependendo da posição em relação à variável. Se colocados antes (pré-incremento) a variável é incrementada (ou decrementada) no momento de sua Avaliação. Se colocados após a variávei (pós-incremento) a variável é incrementada (ou decrementada) após sua avaliação. Exemplos:

```
int x = 10;
int y = 0;
int z = 0;

y = x++;
z = ++x;
```

Após a execução deste trecho de código o valor de “x” será o mesmo (11), mas os valores de “y” e “z” serão respectivamente “10” e “11”.

Outro operador que pode ser aplicado a variáveis é o “cast” ou transformação de tipo de dados. Para efetuar Cast em uma variável informamos o novo tipo entre parêntesis:

```
byte d;
d = (byte) 0xfe;
```

Neste caso tivemos que efetuar o Cast porque literais inteiros sempre são considerados como “short” (32 bits) e um byte só pode conter 8 bits. Se não fizéssemos o Cast o resultado seria:

```
C:\Cleuton\Aulas\Java2Prog\Exemplos\02>javac atrib.java
atrib.java:15: possible loss of precision
found   : int
required: byte
                d = 0xFE;
                    ^
1 error
```

Operadores Aritméticos

Em Java temos um conjunto de operadores aritméticos completos. Veja a tabela:

Soma	+
Subtração	-
Multiplicação	*
Divisão	/
Módulo (resto da divisão)	%

A potenciação é feita através das Classes do pacote java.math.*, método “pow” (arquivo “atrib.java”):

```
import java.math.*;
...
int a = 2;
BigInteger e = new BigInteger("5");
int f = 0;
f =  e.pow(a).intValue();
```

Neste pequeno trecho de código vemos a criação de um Objeto “BigInteger”, do pacote java.math, e na última linha calculamos o cubo do seu valor. Note que na última linha estamos calculando o cubo com o método “pow” e simultaneamente convertendo-o em Inteiro com o método “intValue”.

Considerações sobre cálculos

O único caso que retorna exceção é a divisão por zero, todos os outros casos não geram exceções.

Se alguma variável double ou float apresentar um valor não numérico, o resultado será “Double.NaN” ou “Float.NaN”. Neste caso você poderá testar com as funções “Float.isNaN” ou “Double.isNaN”. Comparações com “Double.NaN” ou “Float.NaN” nunca darão certo.

Operações de deslocamento (SHIFT)

O Shift é o deslocamento dos bits de um número para a direita ou esquerda. Serve para multiplicar ou dividir um número pelas potências de “2”.

Em Java temos três operações de deslocamento:

<<	Deslocamento sinalizado para a esquerda
>>	Deslocamento sinalizado para a direita
>>>	Deslocamento para a direita sem sinal

No arquivo de exemplo “testeshift.java” temos várias operações de deslocamento implementadas.

O Shift é o deslocamento de bits. Vamos supor a seguinte operação:

```
int a = 2;
int a_LeftShift = a << 4; // vezes 2 elevado a 4
```

Como a variável “a” é inteira, possui 32 bits. Seu conteúdo original é:

0000 0000 | 0000 0000 | 0000 0000 | 0000 0010

Ao deslocarmos um bit para a esquerda o valor será:

0000 0000 | 0000 0000 | 0000 0000 | 0000 0100

Ou seja: 4, que é a multiplicação do valor original (2) por 2^1.

Ao deslocarmos 4 bits para a esquerda obtemos o valor:

0000 0000 | 0000 0000 | 0000 0000 | 0010 0000

Que é 32 ou seja: valor original (2) x 2^4.

Agora considere o deslocamento para a direita. Veja o exemplo:

```
int b = 256;
int b_RightShift = b >> 4; // dividido por 2 elevado a 4
```

O valor original de “b” é:

0000 0000 | 0000 0000 | 0000 0001 | 0000 0000

Se deslocarmos um bit para a direita teremos:

0000 0000 | 0000 0000 | 0000 0000 | 1000 0000

Que é 128, ou seja, o valor original (256) dividido por 2^1.

Agora se deslocarmos 4 bits para a direita teremos:

0000 0000 | 0000 0000 | 0000 0000 | 0001 0000

Que é 16, ou seja, o valor original (256) dividido por 2^4.

Complemento a dois

Quando deslocamos para a esquerda os novos bits são sempre preenchidos com ZERO (bits da direita). Quando deslocamos para a direita, os novos bits à esquerda são preenchidos com o valor do último bit que estava na casa.

Isto é para preservar o sinal.

Em Java os números negativos são representados através do Complemento a dois. Para calcularmos a representação binária de um número negativo:

1. Pegamos a representação binária do número
2. Invertemos os bits através de negação
3. Somamos “1”

Veja alguns exemplos:

Número 16:

0000 0000 | 0000 0000 | 0000 0000 | 0001 0000

Negando os bits:

1111 1111 | 1111 1111 | 1111 1111 | 1110 1111

Somando “1” temos “-16”:

1111 1111 | 1111 1111 | 1111 1111 | 1111 0000

Números com o Bit de mais alta ordem ligado são negativos. Desta forma podemos deduzir os limites dos valores primitivos. Por exemplo o tipo “byte” é sinalizado, logo pode conter os números não sinalizados:

0000 0000 até 1111 1111

Como os números com o Bit de mais alta ordem ligado são negativos, podemos representar: -2^7 até $(2^7 – 1)$:

Número 128: 1000 0000
Número -128 (-2^7): 1000 0000
Número 127 ($2^7 – 1$): 0111 1111

Podemos representar de 1000 0000 até 0111 1111.

Deslocamento de valores negativos

Se quisermos deslocar um número negativo para a esquerda, o comportamento dos novos bits inseridos à direita será o mesmo, ou seja, serão inseridos ZEROS.

Agora, se quisermos deslocar um número negativo para a direita, qual deverá ser o comportamento dos novos bits inseridos à esquerda? A resposta é que serão inseridos bits de acordo com o último bit de mais alta ordem. Suponha o número “-32”:

Número –32:

1111 1111 | 1111 1111 | 1111 1111 | 1110 0000

Deslocado 2 bits para a direita:

1111 1111 | 1111 1111 | 1111 1111 | 1111 1000

Que é igual a “-8”, ou seja, o valor original (-32) dividido por 2^2.

Você pode ver isto no exemplo “testeshift.java”.

Deslocamento não sinalizado para a direita

Em Java as variáveis dos tipos “int”, “long”, “byte” são sempre sinalizadas, logo, o deslocamento para a direita “>>” é sempre sinalizado. Em outras palavras, sempre que você inserir um novo bit à esquerda, ele terá o valor do último bit de mais alta ordem.

E se você quiser deslocar um número para a direita sem se importar com a manutenção de sinal? Em C++ é possível fazer isto porque podemos declarar nossas variáveis como “unsigned”, mas em Java isto não existe.

Para deslocar um valor para a direita sem manter a equivalência, ou seja, novos bits inseridos à esquerda como ZERO, temos que usar o operador “>>>” ou Unsigned Right Shift.

No mesmo exemplo (“testeshift.java”) temos uma operação deste tipo:

```
int c_UnsignedRightShift = c >>> 2;
```

O valor original da variável “c” era –32:

1111 1111 | 1111 1111 | 1111 1111 | 1110 0000

Após a operação ficou: 1.073.741.816

0011 1111 | 1111 1111 | 1111 1111 | 1111 1000

Por que os dois bits incluídos à esquerda (sublinhados) ficaram como ZERO.

Redução do número de Bits de deslocamento

Em Java não é possível fazermos “loops” de deslocamento. Só podemos deslocar um número pelo total de bits – 1. Se temos uma variável “int” só poderemos deslocá-la até 31 bits.

Se informarmos um valor de deslocamento maior que o número de bits do operando, ele será reduzido pela fórmula: <bits reais> = <bits informados % <tamanho em bits>. Por exemplo, se quisermos deslocar uma variável “int” 33 bits para a direita, na verdade, o Java deslocará 33 % 32 bits, ou seja “1” bit.

Promoção de variáveis em expressões aritméticas

Antes de calcular uma expressão, e após avaliar os componentes, os valores são promovidos de modo a ficar com o tamanho do maior tipo envolvido ou, pelo menos, “int”.

Na seguinte expressão:

```
byte b = 5;
byte c = 2;
byte f;
f = (byte) (b * c);
```

Por que o Cast para “byte” foi necessário na última linha? Porque os valores de “b” e “c” foram promovidos para “int” antes de efetuar o cálculo e o resultado ficou como “int”. Logo, para atribuir um valor “int” a um valor “byte” temos que fazer o Cast.

Veremos mais adiante um capítulo sobre Conversões onde serão abordadas as regras de conversão automática do Java.

Se uma das variáveis for “long”, as outras também serão convertidas para “long”. O resultado sempre será do mesmo tamanho que o maior dos operandos. Se algum deles for “double” todos serão convertidos para “double”.

Isto tem conseqüências complicadoras no uso do Shift com variáveis “byte”. Suponha o exemplo (arquivo “expressoes.java”):

```
byte z = -64;
byte x;
x = (byte) (z >>> 4);
```

Neste caso você está esperando o valor 268.435.452, que seria o resultado do Unsigned Right Shift sobre o valor. Mas, como os bits inseridos serão retirados no Cast, você acabará com um resultado diferente do que esperava.

O sinal “+” em expressões com Strings

Quando o sinal “+” atua sobre Objetos String, ele efetua uma concatenação. Por exemplo (arquivo “expressoes.java”):

```
String nome1 = "teste ";
String nome2 = " de expressoes";

System.out.println(nome1 + nome2);
```

O resultado será a concatenação dos dois textos.

Quando temos objetos String misturados com variáveis numéricas ou outros Objetos na mesma Expressão, o Java os converte todos para String. Isto é feito de maneiras diferentes:

- Se o operando for um Objeto diferente de String, será chamado o método “toString”, que é implementado em java.lang.Object (pai de todos os Objetos em Java).

- Se o operando for um tipo primitivo, será utilizado o método “toString” da classe encapsuladora (veremos adiante). Se for um inteiro será utilizado o método “Integer.toString()”.

Operadores lógicos bitwise

Em Java temos os seguintes operadores lógicos:

~	NOT (Negação)
&	E (conjunção)
|	OU (disjunção)
^	XOR (Disjunção excusiva)

Esses operadores podem ser utilizados em qualquer tipo. Veja um exemplo (arquivo “expressoes.java”):

```
R_and = (byte) (b & c);
R_xor = (byte) (b ^ c);
```

Se os operandos forem “boolean” serão tratados como um único bit. Para valores true, o valor será “1” e para valores false será “0”.

Comparadores lógicos

Podemos efetuar comparações em Java com os operadores: “==” (dois sinais de igual), “<”, “>”, “<=”, “>=” e “!=”.

Eles retornam um valor “boolean” indicando se a comparação é verdadeira ou falsa. Veja um exemplo (arquivo “expressoes.java”):

```
byte c = 2;
byte b = 5;
```

System.out.println(“c > b = “ + (c > b));

O resultado será false pois “c” é menor que “b”. No mesmo programa (“expressoes.java”) temos um exemplo de negação, onde se obtém o valor oposto ao inicial (!(c > b)).

Para comparar se dois valores são iguais temos que utilizar o duplo igual “==”:

```
boolean x;
x = c == 2;
```

Podemos comparar variáveis de tipos diferentes, conforme o programa "expressoes.java" faz. Neste caso ele compara uma variável "char" com um literal "long". A variável "char" é promovida a "long" e depois a comparação é efetuada.

Podemos comparar duas variáveis objeto também, embora existam algumas restrições. Para saber se uma referência aponta para uma determinada classe podemos utilizar o operador "instanceof":

```
if (xAluno instanceof Aluno) {
}
```

Neste caso está se perguntando se a variável "xAluno" aponta para um Objeto da classe "Aluno".

Operador condicional

Em Java temos um operador semelhante à função "IIF" do VB. Trata-se do Operador Condicional "?". Sua sintaxe é:

<destino> = <expressão de teste> ? <expressão verdadeira> : <expressão falsa>

x ? a : b

Neste exemplo, se "x" for true o valor será "a", caso contrário será "b". Veja outro exemplo (arquivo "expressoes.java"):

```
teste2 = (letra == 65L) ? 10 : 2;
```

Neste caso a variável "teste2" receberá "10" se o resultado da expressão "letra == 65L" for true e "2" se for false.

Pode-se concatenar diversas expressões com operadores condicionais, como o exemplo:

```
teste3 = (c == 2) ? (b != 5) ? 100 : (letra == 'A') ? 200 : 300 : 50;
```

Embora, por critérios de legibilidade, recomenda-se evitar a todo custo esta prática.

Operadores Condicionais

Os operadores condicionais em Java são:

&&	E (Conjunção)
||	OU (Disjunção)
!	Não (Negação)

Os operadores condicionais são utilizados em expressões lógicas, da mesma forma que os operadores Bitwise (“&”, “|” e “^”), porém com uma pequena diferença: dependendo da avaliação da primeira expressão, a segunda poderá não ser avaliada.

No arquito “expressoes.java” temos um exemplo de operadores condicionais:

```
teste5 = (c == 3) && (teste4 = b == 5);
teste6 = (c == 3) &  (teste7 = b == 5);
```

Se a primeira e a segunda expressões forem verdadeiras as variáveis “teste5” e “teste6” receberão true. Como a expressão “c == 3” é falsa no programa (veja o código completo em “expressoes.java”), ambas as variáveis de resultado ficam como false.

A diferença é que quando usamos o operador condicional (&&) a segunda expressão não será avaliada se a primeira for false. Logo, “teste4” nunca recebera o valor true (pois no programa b é 5). Na segunda linha, apesar do resultado ser false, “teste7” recebe true.

Questões e Exercícios

1) Analise o seguinte código-fonte em java:

```
Arquivo: exop01.java
-------------------------
1>
2> public class exop01 {
3>    public static void main(String args[]) {
4>          byte b = 7;
5>          byte c = 0;
6>          c += b;
7>          byte z = (byte) (b + c);
8>          c = b + 1;
9>    }
10> }
```

a) Dará erro de compilação na linha 8
b) Dará erro de compilação nas linhas 7 e 8
c) Compilará e rodará sem problemas
d) Compilará mas dará exception na linha 6

2) Analise o seguinte programa em Java:

```
Arquivo: exop02.java
-------------------------
1>
2> public class exop02 {
3>    public static void main(String args[]) {
4>          int a = -5;
5>          int c = a >> 1;
6>          int d = a >>> 1;
7>          System.out.println(c);
8>          System.out.println(d);
9>    }
10> }
```

Quais serão os sinais dos valores impressos de “c” e “d”?

3) Qual é o resultado de “z” após o comando:

```
int z = (a > 6) && (++c <=7) ? 500 : 100;
```

4) Qual é o valor de “b” após os comandos:

```
int a = 5;
int b = 0;
boolean c = (a < 4) && (--b == 0);
```

Respostas

1) letra “a”. O resultado da expressão é “int” e a variável é “byte”. Note que existem algumas excessões, como a linha 6.

2) Negativo, “-3” e positivo, “2147483645” respectivamente.

3) O resultado é “100”. Porque a segunda expressão não será avaliada.

4) O resultado é 0 (zero). Porque a segunda expressão não será avaliada.

Modificadores

Os exemplos estão em “exemplos/03”.

Em Java, assim como em outras linguagens OOP, existem mecanismos para alterar a visibilidade ou a característica das classes, variáveis e métodos. É isto que os Modificadores fazer.

Existem modificadores comuns e modificadores de acesso.

Os modificadores comuns alteram a Natureza do elemento (classes, métodos ou variáveis, trechos de código) a que se aplicam. Os modificadores de acesso alteram a Visibilidade do elemento (classes, métodos, variáveis) a que se aplicam.

Os modificadores comuns são

abstract	final	native	static
synchronized	transient	volatile	

Os modificadores de acesso são:

“” (default)	public	protected	private

Modificadores de Acesso

Os modificadores de acesso do java são:

“” (default)	Se não for informado um modificador de acesso, o padrão é que os elementos definidos são visíveis a todas as classes definidas no mesmo Package. Note que “default” não é uma palavra-chave em Java.
private	Se aplicado a um elemento, determina que apenas por instâncias da mesma classe. Mesmo instâncias de sub classes não poderão acessar o elemento. Classes podem ser “private” desde que sejam “inner” (veremos o conceito de “inner classes” adiante). Classes Top-level não podem ser “private”.
protected	Só pode ser aplicado a variáveis e métodos. Um elemento “protected” pode ser acessado a partir da própria classe ou de suas sub-classes, estejam ou não no mesmo Package. Igualmente, os elementos “protected” estão visíveis a todas as classes no mesmo Package, sejam ou não sub-classes de quem os define.

public	Os elementos declarados como “public” são visíveis a todas a classes, sem restrições. Cuidado para não declarar propriedades desta forma. É sempre melhor declará-las como métodos.

Exemplos:

Suponha a classe “visibilidade” (arquivo “visibilidade”):

```
public class visibilidade {
     int numero;
     private int valor;
     protected int saldo;
     public int codigo;
     public visibilidade(int num,int val,int sal,int cod) {
          super();
          numero = num;
          valor = val;
          saldo = sal;
          codigo = cod;
     }
}
```

Agora abra o arquivo “outra.java”:

```
public class outra extends visibilidade {
     public outra() {
          super(5,4,3,2);
     }
     public void metodo() {
          System.out.println(this.numero + '\n' +
               // this.valor +  '\n' +
               this.saldo +  '\n' +
               this.codigo);
     }
}
```

Finalmente, temos o arquivo “teste.java”:

```
public class teste {
     outra obj;
     public teste() {
          obj = new outra();
     }
     public static void main(String args[]) {
          teste x = new teste();
          System.out.println(x.obj.numero + '\n' +
                         //   x.obj.valor +  '\n' +
                            x.obj.saldo +  '\n' +
                            x.obj.codigo);
     }
}
```

Da maneira que estão, se estiverem no mesmo diretório, compilarão sem erros e rodarão sem problemas. Mas se retirarmos os comentários ("//") dos arquivos "outra.java" e "teste.java", estes não compilarão.

O motivo é que a variável "valor" está definida em "visibilidade" como "private". As outras não são problemas porque:

- "numero" é "default", como os três arquivos estão no mesmo diretório, todos terão acesso a ela.
- "saldo" é "protected". A classe "outra" é sub-classe de "visibilidade" e a classe "teste" está no mesmo Package, logo, ambas têm acesso.
- "código" é "públic", logo, todos têm acesso a ela.

Por que não precisamos utilizar o comando "import" em "outra.java" e "teste.java"? Porque as três classes estão no mesmo diretório, logo, estão no mesmo Package.

Modificadores Comuns

Os modificadores comuns alteram a Natureza dos elementos aos quais são aplicados. Vamos apresentar a descrição sucinta:

abstract	O elemento é virtual e deve ser redefinido em sub-classes. Só pode ser aplicado a métodos e classes. Se uma classe possuir um método declarado como "abstract", ela deve ser declarada como "abstract". É redundante, mas é assim que deve ser. Você não pode implementar o método em uma classe abstrata, mas deve implementá-lo em qualquer sub-classe.
final	Significa que o elemento não pode ser alterado. Pode ser aplicado a classes, métodos ou variáveis. Uma classe "final" não pode ter sub-classes. Um método "final" não pode ser redefinido (sobrescrito) e uma variável "final" não pode ser alterada.
native	Só pode ser aplicado a métodos. Um método declarado como "native" possui sua implementação fora do JVM. É uma rotina escrita em C++, por exemplo, residente em uma DLL.
static	Pode ser aplicado a variáveis, métodos ou trechos de código (entre "{" e "}"). Se aplicado a uma variável, significa que ela pertence à classe e não à instância do Objeto. Se aplicado a um método, este passa a ser também da Classe e somente pode acessar suas variáveis Estáticas. Se aplicado a um trecho de código, este será executado no momento em que a Classe for carregada pelo JVM.

synchronized	Permite controlar acesso a código sensível em ambientes multi-thread. Pode ser aplicado a métodos ou a trechos de código (entre "{" e "}"). Veremos melhor este modificador quando falarmos sobre "threads".
transient	Pode ser aplicado a variáveis significando que elas não serão persistidas junto com o Objeto. Uma classe pode implementar a interface "Serializable", o que significa que uma instância pode ser persistida (ter seus dados gravados em disco). As variáveis marcadas com "transient" não serão salvas.
volatile	Uma variável pode ser declarada como "volatile", significando que os threads possuirão cópias locais da mesma. Quando um thread necessitar do valor, deverá reconciliar a sua cópia local com a cópia principal do programa.

Agora vamos ver alguns exemplos dos principais modificadores.

Abstract

Serve para declarar classes abstratas ou fundacionais. Este tipo de classe serve para "lembrar" ao programador os métodos que deve implementar em uma sub-classe. Veja um exemplo:

Você não pode criar instâncias da classe Abstrata. Veja um exemplo:

arquivo "pessoa.java":

```
public abstract class pessoa {
     protected String mNome;
     protected int mCodigo;
     protected String mEndereco;
     public abstract String getNome();
     public abstract int getCodigo();
     public abstract String getEndereco();
}
```

arquivo "chefe.java":

```
public class chefe extends pessoa {
     public String getNome() {
          return mNome;
     }
     public int getCodigo() {
          return mCodigo;
     }
```

```
     public String getEndereco() {
          return mEndereco;
     }
}
```

Se você tirar um dos métodos da classe “chefe”, o compilador dirá que ela deverá ser abstrata pois não implementa todos os métodos abstratos de “pessoa”.

Uma classe abstrata não pode ser instanciada. Veja um exemplo (arquivo “testapessoa.java”):

```
public class testapessoa {
     private pessoa x;
     public void testa() {
          x = new pessoa();
          System.out.println(x.getEndereco);
     }
     public static void main(String args[]) {
          testapessoa y = new testapessoa();
          y.testa();
     }
}
```

Final

Uma classe “final” não pode possuir sub-classes. É o contrário de uma classe Abstrata. Veja um exemplo:

```
public final class carro {
}

public class onibus extends carro {
}
```

A compilação da Segunda classe resultará em erro, pois “carro” foi definida como “final”.

Um método “final” não pode ser redefinido (override). Redefinir um método é reescrevê-lo em sub-classes. Por exemplo:

```
public class veiculo {
     protected boolean ligado;
     public final boolean ligar() {
          ligado = true;
          return ligado;
     }
}
```

```
public class carro extends veiculo {
     public boolean ligar() {
          ligado = true;
          return ligado;
     }
}
```

Como o método “ligar” está definido como “final” na classe “veículo”, a classe “carro”, não pode redefiní-lo.

Uma variável “final” não pode ser alterada. Em outras palavras:

- uma variável primitiva com “final” vira uma constante.
- uma variável referência de objeto com “final” não pode ter a referência alterada.

Porém, o objeto referenciado por uma variável “final” pode ter suas propriedades alteradas:

```
public final x = new chefe();
x = new chefe(); // ERRO
x.nome = “José”; // OK, sem problemas
```

Static

Uma variável de classe “static” pertence apenas à classe e não às suas instâncias. Em outras palavras existe apenas uma cópia da variável:

```
public class pessoa {
     static int quantidade = 0;
     int codigo;
     String nome;
}
```

Neste caso a variável “quantidade” pertence à classe “pessoa” e as variáveis “codigo” e “nome” pertencem às instâncias a serem criadas. Você pode acessar “quantidade” de duas maneiras:

```
     public pessoa x = new pessoa();
     System.out.println(x.quantidade);

     System.out.println(pessoa.quantidade);
```

Pode usar o nome de uma instância ou o nome da classe, não importa pois ambos apontam para a mesma variável. Só existirá uma “quantidade”. Ela será inicializada quando a classe for carregada no JVM.

Métodos declarados como “static” são independentes das instâncias da classe. Na verdade podem ser invocados sem que uma classe exista (Integer.parseInt é um

exemplo). Os métodos Estáticos somente podem acessar variáveis Estáticas da classe. Veja um exemplo:

```
public class estatica {
     static int x = 0;
     int z = 13;
     public static void main(String args[]) {
          estatica w = new estatica();
          System.out.println(x);
          // System.out.println(z);
          System.out.println(w.z);
     }
}
```

Se retirarmos o comentário ("//") a compilação dará erro, pois a variável "z" não é estática. Porém, ele pode acessá-la a partir de alguma instância criada, como na linha seguinte.

Podemos colocar código para ser executado quando uma Classe for carregada. São úteis para inicializar variáveis estáticas:

```
static {  x = 130; }
```

Parece estranho mas não é. O código será executado quando a classe ao qual pertence for carregada na memória. Não será repetido a cada instância criada!

Override de Métodos

Vamos adiantar um pouco e definir o conceito de "Override" de Métodos. Uma classe pode alterar a definição de um método herdada de sua ancestral. Exemplos (arquivo "testa.java"):

```
class pessoa {
     protected String nome;
     protected int id;
     public String ObterNome() {
          return nome;
     }
}

class profissional extends pessoa {
     protected String titulo;
     public profissional(String wnome, String wtitulo) {
          this.nome = wnome;
          this.titulo = wtitulo;
     }
     public String ObterNome() {
          return nome + "(" + titulo + ")";
```

```
     }
}

public class testa {
     public static void main(String args[]) {
          profissional x = new
profissional("cleuton","Professor");
          System.out.println(x.ObterNome());
     }
}
```

Neste exemplo o método “ObterNome”, da classe “pessoa”, foi redefinido na classe “profissional”. Observe que, apesar da “assinatura” do método ser a mesma, seu conteúdo é completamente diferente. Isto se chama “override” de método ou “redefinição” de método.

Você pode redefinir um método alterando seu modificador de acesso, sempre tornando-o mais visível. Se ele é “private” você pode redefiní-lo como “public”. Eis a escala correta:

```
private->default->protected->public
```

No exemplo anterior, se você alterar o “override” para “default” (retirando o modificador “public” do método “ObterNome”) na classe “profissional”, o Java dará o seguinte erro:

```
C:\Cleuton\Aulas\Java2Prog\Exemplos\03>javac testa.java
testa.java:15: ObterNome() in profissional cannot override
ObterNome() in pessoa
; attempting to assign weaker access privileges; was public
         String ObterNome() {
                ^
1 error
```

Questões e Exercícios:

1) Considere o seguinte código em Java:

```
Arquivo: teste31.java
-------------------------
1> public class teste31 {
2>   private float saldo;
3>   private String nome;
4>   public abstract float saldo();
5>   public String nome() {
6>        return this.nome;
7>   }
8>
9> }
```

a) Compilará e rodará sem problemas.
b) Dará erro de compilação porque métodos e propriedades não podem ter o mesmo nome.
c) Compilará mas dará run-time exception.
d) Dará erro de compilação.

2) Considere o seguinte código em Java:

```
Arquivo: teste32.java
-------------------------
1> abstract class cliente {
2>   float saldo;
3>   String nome = "teste";
4>   public abstract float saldo();
5>   public final String nome() {
6>        return this.nome;
7>   }
8>
9> }
10>
11> public class teste32 extends cliente {
12>  public float saldo() {
13>       return this.saldo;
14>  }
15>  public static void main(String args[]) {
16>       teste32 t = new teste32();
17>       System.out.println(t.nome() + "\r\n"+ t.saldo());
18>  }
19> }
20>
```

a) Compilará e rodará sem problemas.
b) Dará erro de compilação na linha 13.
c) Dará erro porque classes abstratas não podem ter métodos “final”.
d) Dará erro de compilação porque as variáveis “saldo” e “nome”, da classe “cliente” não podem ser “enxergadas” pela classe “teste32”.

3) Crie uma estrutura de classes para o seguinte problema:

- Uma pessoa tem como atributos: nome e endereço.
- Uma pessoa deve possuir métodos para obter e alterar seus atributos.
- Um cliente é uma pessoa que também possui o atributo saldo.
- Um cliente deve possuir métodos para obter e alterar seu saldo.

Use Abstract e Final quando necessário.
Crie também uma classe top level (com método “main”) que instancie “cliente” e use seus métodos.

Respostas:

1) Resposta correta: “D”. Uma classe que possui um método abstrato, ou que não faz “override” de todos os métodos abstratos que herdou, deve ser declarada como abstrata.

2) Resposta correta: “A”. Não há nada de errado em uma classe abstrata possuir métodos com “final”. Estes métodos não podem sofrer “override”.

3) A classe “pessoa” pode ser abstrata, mas seus métodos para obter e alterar nome e endereço devem ser “final”.

Conversão e transformação de tipos de dados

Os exemplos estão em “exemplos/04”.

Às vezes é necessário converter uma expressão para outro tipo de dados. Existem duas maneiras de fazer isto: conversão (Convert) e transformação (Cast).

O Java pode fazer a conversão, automática, de um tipo de dados para outro em certos casos. Considere o exemplo (arquivo “ex01.java”):

```
pessoa x = new pessoa();
Vector tab = new Vector();
tab.add(x);
```

Se observarmos a declaração do método “add”, da classe “Vector”, veremos que sua assinatura é:

```
boolean add(Object o);
```

Logo, como passamos uma referência a uma classe “pessoa”, o JVM fez a conversão de “pessoa” para “Object” automaticamente.

Embora isto seja possível de maneira automática, certas regras devem ser aplicadas. Por exemplo, o seguinte código daria erro:

```
pessoa x = new pessoa();
pessoa y;
x.nome = "Teste";
Vector tab = new Vector();
tab.add(x);

y = tab.elementAt(0);

ex01.java:14: incompatible types
found   : java.lang.Object
required: pessoa
                y = tab.elementAt(0);
```

Por que deu erro, se nós sabemos que o elemento da classe “Vector” é do tipo “pessoa”? Deu erro porque ao adicionarmos o objeto “pessoa” ao “Vector” “tab”, o JVM o converteu para “Object”. Apesar dele ainda ser o mesmo, seu tipo foi alterado. Logo, devemos forçar a conversão novamente para “pessoa” ao atribuí-lo à variável “y”:

```
y = (pessoa) tab.elementAt(0);
```

Apesar de ainda não havermos falado sobre Vector, este exemplo mostra bem o que o JVM faz automaticamente e o que devemos fazer manualmente.

Outro exemplo, do mesmo arquivo (ex01.java) é o seguinte:

```
1)        byte a = 5;
2)        byte b = 2;
3)        int c;
4)        byte d;

5)        c = a + b;
6)        d = a + b;
```

Na linha 5 temos a conversão automática dos operandos para “int” e depois sua atribuição à variável “c”, que é “int”. Isto o JVM faz automaticamente. Porém, na linha 6 dará o seguinte erro:

```
ex01.java:23: possible loss of precision
found   : int
required: byte
                d = a + b;
                      ^
```

O Java está reclamando da atribuição da soma de duas variáveis “byte” a uma terceira, também “byte”. Por que? Porque o Java converte automaticamente os operandos para “int”. Para dar certo teríamos que forçar ou transformar (Cast) o resultado:

```
          d = (byte) (a + b);
```

Estamos forçando a conversão para “byte” do resultado da expressão entre parêntesis (“a + b”).

Conversão de tipos primitivos

Os tipos primitivos, como vimos, são: “int”, “short”, “byte”, “long”, “char”, “float”, “double” e “boolean”. Os outros tipos são classes, derivados de “Object”.

A conversão é o processo de mudança de tipo automático, feito pelo Java em tempo de compilação. Ele ocorre nas situações:

- Atribuição
- Chamada de método
- Promoção aritmética

No caso da atribuição, a conversão ocorre quando tentamos atribuir o resultado de um tipo a outro. Por exemplo:

```
int num = 2;
double z;
z = num * 2;
```

Neste caso, na terceira linha, o resultado da expressão “x * 2” será convertido para “double” e atribuído automaticamente à variável “z”.

Porém, se tentarmos fazer o seguinte:

```
int num = 2;
short z;
z = num * 2;
```

Resultará no erro:

```
ex01.java:17: possible loss of precision
found   : int
required: short
                z = num * 2;
                      ^
```

O motivo é que tentamos atribuir o valor de um tipo maior que o da variável receptora. Isto se chama “Narrowing Conversion” ou conversão restringente. Se tentarmos atribuir uma expressão a uma variável de tipo maior que o dela, o Java aceitará, pois se trata de uma “Widening Conversion” ou conversão abrangente. Exemplos:

Tipo original	Novo tipo
byte	short, int, long, float ou double
short (16 bits)	int, long, float ou double
char (16 bits)	int, long, float ou double
int	long, float ou double
long	float ou double
float	double

Atribuições de literais a tipos primitivos menores

Se a seguinte questão caísse em um exame, qual seria a sua resposta?

```
Analise o código (arquivo “ex02.java”):

1)   float z = 1.44;
2)   byte n = 7;
3)   short x = 10;

   A) a linha 1 daria erro
   B) a linha 2 daria erro
   C) as linhas 1, 2 e 3 dariam erro
   D) nenhuma das anteriores
```

Seria razoável supor que TODAS as linhas apresentassem erro, correto? Por que? Porque LITERAIS INTEIROS em Java são do tipo “int” e LITERAIS REAIS em Java são do tipo “double”. Logo, ocorreriam conversões restringentes em todas as linhas.

O problema do Java é que ele só dará erro na linha 1. As outras compilam sem problemas. Isto é porque ele aceita conversões restringentes de LITERAIS INTEIROS para tipos inteiros pequenos, desde que o valor caiba. No arquivo “ex02.java” há uma linha:

```
byte y = 287;
```

Que apresenta erro de compilação, porque 287 (1 0001 1111) não cabe em apenas um byte.

Esta exceção apenas se aplica quando estamos atribuindo LITERAIS INTEIROS a um tipo pequeno inteiro (byte ou short). Se tentarmos atribuir uma variável, dará erro.

Chamada de métodos

De maneira semelhante, ao passarmos variáveis como argumentos para métodos, caso os tipos de dados sejam diferentes, ocorrerá conversão. Igualmente, só conversões abrangentes são permitidas. Por exemplo (ex02.java):

```
public void testa(int i) {
}
...
byte x = 3;
testa(x);
```

Esta seqüência não dará erro algum. Agora considere as seguintes linhas:

```
1)      float w = 1.44f;
2)      double dd = 7.2;
3)      byte n = 7;

4)      testa(n);
5)      testa(w);
6)      testa(dd);
```

As linhas 5 e 6 darão os seguintes erros:

```
ex02.java:16: testa(int) in ex02 cannot be applied to
(float)
                testa(w);
                ^
ex02.java:17: testa(int) in ex02 cannot be applied to
(double)
                testa(dd);
                ^
```

Repare que ele aceitou a versão “byte” (testa(n)) sem problemas.

Promoção aritmética

Quando temos uma expressão, o Java tenta compatibilizar os tipos de dados dos operandos. E, para isto, segue algumas regras específicas.

Temos dois tipos básicos de operadores em expressões aritméticas: unários e binários. Recordando, operadores unários são aplicados apenas a um só operando:

```
a = -x;
a = ++x;
```

Os operadores binários se aplicam a dois operandos:

```
a = (b – 2) * c;
```

Existem regras distintas para operadores unários e binários.

Promoção de operadores unários

Operando = byte/short/char e operador diferente de "- - " e "++"	Converter operando para int
Operando <> byte/shor/char	Não converter

Promoção de operadores binários (exclusivas)

Se um dos operandos for double	o outro será convertido para double
Se um dos operandos for float	o outro será convertido para float
Se um dos operandos for long	o outro será convertido para long
Senão	ambos serão convertidos para int

Por que a expressão seguinte dá erro:

```
byte x = 3;
byte a;
x = a + 1;
```

a) "x" é um nome inválido para o tipo "byte"
b) "byte" é um nome inválido para o tipo "x"
c) o literal "1" é INTEIRO
d) ambos os operandos foram convertidos para "int"

A resposta correta é a letra "d". Pelas regras de promoção aritmética de operadores binários, caso nenhum dos operandos seja "double"/"float"/"long", ambos serão convertidos para "int". Como a conversão de "int" para "byte" é restritiva, o Java reclamará.

Mais uma regra se aplica a conversão: booleans não podem ser convertidos para nada.

Transformação (Cast)

Quando você quer fazer uma conversão restritiva, deve informar ao Java que está ciente dos possíveis problemas de perda de dados resultantes da operação. Isto é feito através da transformação ou “Cast”.

O Cast existe em várias linguagens de programação e consiste em informar, entre parêntesis, antes da expressão, o tipo para o qual deseja converter o resultado. O Java não reclamará, mas poderá ocorrer perda de dados. Veja o exemplo “ex02.java”:

```
byte y = (byte) 287;
```

Quando executamos o programa obtemos “31” como sendo o valor de “y”. Por que? É o resultado da truncagem do valor 287 (1 0001 1111) para byte (0001 1111).

Ou seja, o Java deixará fazermos quase todas as besteiras possíveis, desde que utilizemos Cast.

A única exceção é que não podemos transformar anda em “boolean” ou “boolean” em outro tipo.

Conversão e transformação de referências a objetos

Temos tipos de dados não primitivos chamados Classes. Quando criamos uma variável pertencente a uma determinada Classe, estamos informando ao Java que aquela variável pode guardar uma referência a um Objeto daquela classe.

Objetos são criados em memória (com o comando “new”, por exemplo) e seu “endereço” é copiado para uma variável de referência.

Suponha o código (testaconv.java):

```
public static void main(String args[]) {
      pessoa xp;
      profissional pro2;
      Vector tab = new Vector();
      pessoa pe = new pessoa("Jose",10);
      profissional pro = new
            profissional("Paulo",11,"carpinteiro");
      tab.add(pro);
      xp = pro;
      pro2 = (profissional) xp;
      System.out.println(pro2.peganome());
}
```

Repare que há algumas conversões em andamento:

- de “profissional” para “Object” em “tab.add(pro)”
- de “profissional” para “pessoa” em “xp = pro”
- de “pessoa” para “profissional” em “pro2 = (profissional) xp”

Só que nas duas primeiras conversões não foi necessário fazer o “Cast”, mas na última sim.

Repare também que o Objeto “profissional” não perdeu sua identidade ao ser convertido para “pessoa”. Apesar de estar referenciado na variável “xp” (pessoa), ao ser convertido novamente para “profissional” manteve sua identidade. Ao imprimirmos o resultado de “pro2.peganome()” vemos que o método executado foi de uma classe “profissional”.

Uma breve conversa sobre Objetos e Interfaces

Uma classe possui métodos e variáveis isto compõe sua “interface” ou aquilo que é visível ao usuário. Como vimos antes, uma Classe pode ser declarada como “Abstract”, servindo apenas como gabarito para criação de sub-classes.

Uma interface é um conjunto de métodos que uma classe deve implementar. Parece com herança, só que a classe não descende da interface. É como se fosse possível a herança múltipla. Uma classe tem apenas um ancestral, mas pode implementar várias interfaces.

Uma classe pode derivar de outra Classe, herdando sua interface, ou mesmo implementar outras interfaces.

Veja o exemplo do arquivo “testaconv.java”:

```
interface converte {
         public String tudo();
}

class profissional extends pessoa implements converte {
           protected String titulo;
           public String tudo() {
               return nome + "," + titulo + "," + codigo;
           }
           public String peganome() {
               return nome + "(" + titulo + ")";
           }
           public profissional(String xnome, int xcodigo,
           String xtitulo) {
               super(xnome,xcodigo);
               this.titulo = xtitulo;

           }
}
```

Repare que a classe “profissional”, além de ser uma sub-classe de “pessoa”, também implementa a interface adicional “converte”.

Resumindo, a classe “profissional” implementa a interface de “pessoa” por herança, a interface “converte” e a sua própria (método “pegarnome” redefinido, construtor redefinido e variável “titulo”).

Podemos também definir referências a interfaces, ao invés de objetos. No próprio exemplo “testeconv.java” temos as linhas:

```
converte cvx;
...
cvx = pro2;
System.out.println(cvx.tudo());
```

Logo, haverá regras para conversão de interfaces também.

Conversão de referências

Quando temos algo semelhante a:

tipo_destino = tipo_original;

Haverá uma conversão automática feita pelo Java. Se for possível a conversão, a variável “tipo_destino” apontará para o Objeto apontado por “tipo_original”. Note que ambos podem ser ***Classes, Arrays ou Interfaces***!

As regras para conversão automática (feita pelo próprio Java sem reclamar) são:

	tipo_original é uma classe	tipo_original é uma interface	tipo_original é um Array
tipo_destino é uma classe	tipo_original deve descender de tipo_destino	tipo_destino deve ser do tipo Object	tipo_destino deve ser do tipo Object
tipo_destino é uma interface	tipo_original deve implementar tipo_destino	tipo_original deve ser uma sub-interface de tipo_destino	tipo_destino deve ser Clonable ou Serializable
tipo_destino é um Array	erro de compilação	erro de compilação	os objetos armazenados em tipo_original devem ser conversíveis nos tipos armazenados em tipo_destino

Aqui vai um resumo melhor:

- uma Interface só pode ser convertida em outra interface ou em Object. Se o novo tipo não for Object, deverá ser uma super-interface dela.
- uma Classe pode ser convertida em outra classe ou uma interface. Se for convertida em uma nova classe, esta deverá ser superclasse da original. Se for convertida em uma interface, esta deverá ser implementada pela classe original.
- um Array pode ser convertido para a Classe Object, para as interfaces Clonable ou Serializable, ou para outro Array. Só um Array de referências pode ser convertido para outro Array, desde que os tipos dos elementos sejam conversíveis.

Voltemos ao exemplo “testeconv.java”:

```
import java.util.Vector;

interface converte {
    public String tudo();
}

class pessoa {
    protected String nome;
    protected int codigo;
    public String peganome() {
          return nome;
    }
    public int pegacodigo() {
          return codigo;
    }
    public pessoa() {

    }
    public pessoa(String xnome, int xcodigo) {
          this.nome = xnome;
          this.codigo = xcodigo;
    }
}

class profissional extends pessoa implements converte {
    protected String titulo;
    public String tudo() {
          return nome + "," + titulo + "," + codigo;
    }
    public String peganome() {
          return nome + "(" + titulo + ")";
    }
    public profissional(String xnome, int xcodigo, String
    xtitulo) {
          super(xnome,xcodigo);
          this.titulo = xtitulo;
    }
```

```
}

public class testaconv {
    public static void main(String args[]) {
          pessoa xp;
          pessoa gente[] = new pessoa[10];
          profissional empregados[] = new profissional[10];
          converte testes[] = new converte[10];
          converte cvx;
          profissional pro2;
          Vector tab = new Vector();
          pessoa pe = new pessoa("Jose",10);
          profissional pro = new
               profissional("Paulo",11,"carpinteiro");
          tab.add(pro);
          xp = pro;
          pro2 = (profissional) xp;
          System.out.println(pro2.peganome());
          cvx = pro2;
          System.out.println(cvx.tudo());
          gente[0] = pro2;
          //empregados =  gente; //se descomentar dará erro
          testes[0] = pro2;
    }
}
```

Repare que as conversões possíveis são:

- Classe para Object em “tab.add(pro)”. Object é superclasse de “profissional”.
- Classe para Classe em “xp = pro”. “xp” é da classe “pessoa”, que é superclasse de “pro” (profissional).
- Classe para Interface em “cvx = pro2”. Como “pro2” é da classe “profissional”, ele implementa “converte”.

Porém, se tentarmos converter o Array de “pessoas” em um Array de “profissional” dará erro. O motivo é que a Classe “pessoa” é superclasse de “profissional”, logo, não pode ser convertida em uma subclasse.

Vamos complicar um pouco.

Transformações

É legal fazer Cast de um tipo para outro, desde que algumas regras sejam obedecidas. Quando estamos fazendo Cast, o verdadeiro tipo da referência só pode ser conhecido em tempo de execução. Logo, é feita uma verificação durante a compilação:

a) Se ambos os operandos são classes, um deve ser subclasse do outro (não importa qual).
b) Se ambos forem Arrays, devem ser de Referências (e não variáveis primitivas), além de dever ser legal converter de um para outro.
c) Sempre será aceita a conversão de uma interface para um Objeto (desde que não seja "final").

Durante a execução o verdadeiro tipo do objeto será avaliado de acordo com as regras:

a) Se o tipo_destino é uma classe, o resultado da expressão sendo atribuída a ele deve ser uma sub-classe sua.
b) Se o tipo_destino é uma interface, o resultado da expressão sendo atribuída a ele deve implementá-lo.

Com algumas modificações o exemplo "testeconv2.java" fica perfeito:

```
import java.util.Vector;

interface converte {
     public String tudo();
}

class pessoa {
     protected String nome;
     protected int codigo;
     public String peganome() {
          return nome;
     }
     public int pegacodigo() {
          return codigo;
     }
     public pessoa() {

     }
     public pessoa(String xnome, int xcodigo) {
          this.nome = xnome;
          this.codigo = xcodigo;
     }
}

class profissional extends pessoa implements converte {
     protected String titulo;
     public String tudo() {
```

```
          return nome + "," + titulo + "," + codigo;
     }
     public String peganome() {
          return nome + "(" + titulo + ")";
     }
     public profissional(String xnome, int xcodigo, String
     xtitulo) {
          super(xnome,xcodigo);
          this.titulo = xtitulo;
     }
}

class outracoisa extends pessoa {
}

public class testaconv2 {
     public static void main(String args[]) {
          pessoa xp;
          outracoisa oc = new outracoisa();
          profissional pc;
          pessoa gp = oc;
          pc = (profissional) gp;
          pessoa gente[] = new pessoa[10];
          profissional empregados[] = new profissional[10];
          converte testes[] = new converte[10];
          converte cvx;
          profissional pro2;
          Vector tab = new Vector();
          pessoa pe = new pessoa("Jose",10);
          profissional pro = new
               profissional("Paulo",11,"carpinteiro");
          tab.add(pro);
          xp = pro;
          pro2 = (profissional) xp;
          System.out.println(pro2.peganome());
          cvx = pro2;
          System.out.println(cvx.tudo());
          gente[0] = pro2;
          //empregados =  gente; //se descomentar dará erro
          testes[0] = pro2;
     }
}
```

O programa compilará OK, mas dará a seguinte Exception:

```
Exception in thread "main" java.lang.ClassCastException
        at testaconv2.main(testaconv2.java:48)
```

Justamente na linha marcada em negrito e itálico.

O motivo é que apesar do Cast ser legal, o conteúdo apontado por “gp” é da classe “outracoisa”, que não implementa “profissional”, resultando em uma Exception.

Moral da história: mesmo que o Cast seja legal, não significa que seja válido em tempo de execução.

Questões e Exercícios:

1) Analise o código Java e diga qual linhas dariam erro:

```
Arquivo: teste41.java
-------------------------
1> public class teste41 {
2>   public static void main(String args[]) {
3>        byte z = 126;
4>        byte a = 0;
5>        a += z;
6>        float x = 12.5;
7>        char v = z + a;
8>   }
9> }
```

R:_____

2) Analise o seguinte código Java:

```
Arquivo: teste42.java
-------------------------
1> class veiculo {
2>   public String placa;
3>   public String marca;
4>   public void ligar() {
5>        System.out.println("Ligando o veículo");
6>   }
7>
8> }
9>
10> class carro extends veiculo {
11>  public int lugares;
12>  public void ligar() {
13>       System.out.println("Ligando o carro");
14>  }
15> }
16>
17> class moto {
18>  public String placa;
19>  public String marca;
```

```
20>  public static int lugares = 2;
21>  public void ligar() {
22>       System.out.println("Ligando a moto");
23>  }
24>
25> }
26>
27> public class teste42 {
28>  public static void main(String args[]) {
29>       carro c = new carro();
30>       moto m = new moto();
31>       veiculo nv = new veiculo();
32>       veiculo v = c;
33>       carro nc = nv;
34>       c = m;
35>
36>  }
37> }
```

a) Dará erro na linha 32.
b) Dará erro nas linhas 32 até 34.
c) Dará erro na linha 33.
d) Dará erro na linha 34.

3) Digite e tente rodar o seguinte código Java:

```
Arquivo: teste43.java
-------------------------
1> interface veiculo {
2>   public void ligar();
3>
4> }
5>
6> interface biciclo {
7>   public void montar();
8> }
9>
10> class carro implements veiculo {
11>  public int lugares;
12>  public void ligar() {
13>       System.out.println("Ligando o carro");
14>  }
15> }
16>
17> class moto implements biciclo, veiculo {
18>  public String placa;
19>  public String marca;
20>  public static int lugares = 2;
21>  public void ligar() {
22>       System.out.println("Ligando a moto");
```

```
23>  }
24>  public void montar() {
25>       System.out.println("Montando na moto");
26>  }
27>
28> }
29>
30> public class teste43 {
31>  public static void main(String args[]) {
32>       carro c = new carro();
33>       moto m = new moto();
34>       veiculo v = c;
35>       veiculo w = m;
36>       w = c;
37>       c = (veiculo) m;
38>       m = c;
39>       biciclo g = m;
40>       m = (moto) w;
41>  }
42> }
```

Ele compilará? Não? Por que? _____

4) Supondo que haja erros de compilação no exercício 3, caso você venha a corrigí-los, ele rodara?

Respostas:

1) Dará erro nas linhas 6 e 7.

2) As respostas “C” e “D” estão corretas. A linha 32 não dá erro porque é uma “Widening” conversion.

3) Dará erro nas linhas 37 e 38. Na linha 37 você não pode fazer um Downcast de “veículo” para “carro”. Na linha 38, um carro não é sub-classe de moto, logo, são tipos diferentes.

4) Dará “ClassCastException” na linha 40 porque, apesar de sobreviver ao tempo de compilação, a transformação (Cast) irá dar problema na execução, porque o verdadeiro tipo de “w” é “carro”, conforme a linha 36.

Controle de Fluxo, Exceções e Assertivas de depuração

Exemplos em “exemplos/05”.

Em Java temos vários comandos que permitem alterar o fluxo da execução, tanto em situações normais quanto em situações de erro ou exceção.

Loops

Em Java não há a instrução “goto”, apesar dela ser palavra reservada. Se você quiser criar uma condição repetitiva ou “loop”, terá que utilizar alguma das instruções disponíveis: *while*, *do* e *for*.

while

o while permite executar um comando (ou bloco de comandos) se uma condição, representada por uma expressão booleana, possuir seu valor-verdade = VERDADEIRO.

Eis a sintaxe de um loop while:

```
while (<expressão booleana>)
     <comando>;

ou

while (<expressão booleana>) {
     <comando1>;
     <comando2>;
     ...
     <comandoN>;
}
```

Exemplos (arquivo “testeloops.java”):

```
public static void main(String args[]) {
     boolean x = false;
     int i = 0;
     String nome = args[0];
     String saida = "";
     i = nome.length() - 1;
     System.out.println(nome);
     while (x = saida.length() < nome.length())
          saida = saida + nome.charAt(i--);
     System.out.println(saida);
}
```

Este programa inverte as letras do primeiro argumento passado. Poderia ser escrito de várias formas, como veremos adiante. Repare que não colocamos o comando a ser repetido entre chaves ({}), embora esta seja uma boa prática de programação. Repare também o uso do operador unário (- -) sobre a variável “i”. Neste caso ele retorna o valor atual de “i” e, depois, o decrementa. Finalmente, repare que, mesmo desnecessariamente, estamos testando o resultado booleano da atribuição da condição à variável “x”.

Este loop poderia ser reescrito assim:

```
while (saida.lentgth() < nome.length()) {
     saida = saida + nome.charAt(i);
     i += 1;
}
```

Podemos ter vários whiles “aninhados”, como o exemplo “testeloops2.java”:

```
while(j < palavras.length) {
         String saida = "";
         i = palavras[j].length() - 1;
         System.out.println(palavras[j]);
         while(x = saida.length() < palavras[j].length())
               saida = saida + palavras[j].charAt(i--);
         System.out.println(saida);
         j++;
}
```

Observações sobre while:

- A expressão tem que ter um resultado “boolean”. Expressões que retornem Strings ou qualquer outro tipo de dado não são permitidas.
- Não podemos criar variáveis dentro do “while”, exemplo:
    - while(boolean y = a < b)

do

O “do” é um loop no qual os comandos são executados pelo menos uma vez, podendo ser repetidos dependendo da condição do “while”. Sua sintaxe básica é:

```
do {
    comando(s)
} while (expression);

ou

do
     comando;
     while (expressão);
```

Analise o seguinte exemplo:

```
Arquivo: dowhile.java
-------------------------
1> public class dowhile {
2>   public static void main(String args[]) {
3>        String nome = "Cleuton";
4>        StringBuffer reverso = new StringBuffer("");
5>        int i = nome.length() - 1;
6>        do
7>             reverso.append(nome.charAt(i--));
8>             while(i >= 0);
9>        System.out.println(reverso.toString());
10>  }
11> }
```

Este exemplo utiliza apenas um comando dentro do do/while, mas podemos ter um bloco de comandos também:

```
Arquivo: dowhile2.java
-------------------------
1> public class dowhile2 {
2>   public static void main(String args[]) {
3>        String nome = "Cleuton";
4>        StringBuffer reverso = new StringBuffer("");
5>        int i = nome.length() - 1;
6>        do {
7>             reverso.append(nome.charAt(i));
8>             i--;
9>        }
10>            while(i >= 0);
11>       System.out.println(reverso.toString());
12>  }
13> }
```

for

O comando “for” executa um bloco de comandos, ou um único comando, enumerando as interações através de uma variável de controle. A cada execução do bloco de comandos, a variável é incrementada (ou decrementada) e comparada com a condição de término. Sintaxe:

```
for(<inicialização>;<condição>;<incremento>) {
     commandos
}

ou

for(<inicialização>;<condição>;<incremento>)
     commando;
```

onde:

inicialização	Expressão utilizada para inicializar a(s) variável(eis) de controle. Podemos ter uma ou mais variáveis de controle.
Condição	Expressão booleana que serve para determinar se o loop será executado ou não.
Incremento	Expressão que altera o valor da variável de controle.

Exemplos:

```
for(x=0; x < nome.length(); x++) {
     char v = nome.charAt(x);
     System.out.println(v);
}

ou

for(x=0; x < nome.length(); x++)
     System.out.println(nome.charAt(x));
```

Podemos também decrementar a variável de controle em um Loop:

```
for(x=nome.length()-1; x >= 0; x--)
     System.out.println(nome.charAt(x));
```

Finalmente, podemos ter mais de uma variável de controle:

```
Arquivo: for1.java
--------------------------
1> public class for1 {
2>   public static void main(String args[]) {
3>        String nomes[] = {"joao", "pedro", "maria"};
4>        String saida[] = new String[3];
5>        int i = 0;
6>        int j = 0;
7>        for(i=0, j=2; i < 3; i++, j--) {
8>             saida[j] = nomes[i];
9>        }
10>       for(i=0; i < saida.length; i++)
11>            System.out.println(saida[i]);
12>  }
13> }
```

No primeiro “for” as duas variáveis, “i” e “j”, são controladas pelo loop. Note que devem ser separados por vírgulas.

break

O comando “break” termina um loop, saindo do seu bloco de comandos. Veja um exemplo:

```
for(x=0; x < nomes.length(); x++) {
     if(nomes.charAt(x) == “*”)
          break;
     System.out.println(nomes.charAt(x));
}
```

Este loop imprime os caracteres até encontrar um asterísco ou terminar o String, o que acontecer primeiro. Após o “break” a execução vai passar para depois do “}” final.

Veja este outro exemplo:

```
Arquivo: break1.java
-------------------------
1> public class break1 {
2>   public static void main(String args[]) {
3>        String clientes[] = {"ped*ro", "*jose", "maria"};
4>        for(int x=0; x<clientes.length; x++) {
5>             String nome = clientes[x];

6>             for(int y=nome.length()-1; y >= 0; y--) {
7>                  if(nome.charAt(y)=='*')
8>                       break;
9>                  System.out.print(nome.charAt(y));
10>            }
11>            System.out.println(" ");
12>       }
13>  }
14> }
```

Nestes dois loops, o comando “break” apenas termina o loop interno.

Qual seria o resultado deste programa?

(“or”,”esoj”,”airam”).

continue

Este comando funciona de maneira semelhante ao “break”. Ele provoca a saída imediata do loop, só que voltando à expressão de incremento. Em outras palavras, o “break” força a saída do loop de comandos e não volta ao comando de loop. O “continue” sai do bloco de comandos, executa a expressão de incremento, testa a expressão condicional e, se estiver dentro dos limites, volta a executar o loop.

Veja o mesmo exemplo anterior, agora com o “continue”:

```
Arquivo: continue1.java
-------------------------
1> public class continue1 {
2>   public static void main(String args[]) {
3>        String clientes[] = {"ped*ro", "*jose", "maria"};
4>        for(int x=0; x<clientes.length; x++) {
5>             String nome = clientes[x];
6>             for(int y=nome.length()-1; y >= 0; y--) {
7>                  if(nome.charAt(y)=='*')
8>                       continue;
9>                  System.out.print(nome.charAt(y));
10>            }
11>            System.out.println(" ");
12>       }
13>  }
14> }
```

O resultado é::

“ordep”
“esoj”
“airam”

Ao encontrar um “*” ele pula este caracter e continua a imprimir a partir do próximo.

Labels

Java não possui “goto” mas permite utilizarmos o “break” ou o “continue” com labels. Um label é um nome dado a uma linha. Exemplo:

```
Teste:
     If(x>=8) {
...
```

Veja um exemplo utilizando “break” e labels:

```
Arquivo: break2.java
-------------------------
1> public class break2 {
2>   public static void main(String args[]) {
3>        String clientes[] = {"pedro", "jo*se", "maria"};
4>        teste:
5>        for(int x=0; x<clientes.length; x++) {
6>             String nome = clientes[x];
7>             for(int y=nome.length()-1; y >= 0; y--) {
8>                  if(nome.charAt(y)=='*')
9>                       break teste;
10>                 System.out.print(nome.charAt(y));
11>            }
12>            System.out.println(" ");
13>       }
14>  }
15> }
```

O resultado é:

“ordep”
“es”

Porque quando ele encontra o primeiro asterisco ele faz um break para o label “teste”, que corresponde ao “for” externo. Logo, ele sai do “for” externo e termina tudo.

Agora, se substituirmos o “break teste” por “continue teste”, o resultado será:

“ordep”
“esairam”

Por que ao encontrar o “*” o loop interno provocará a repetição do loop externo, saindo do meio do cliente “jose” para o próximo cliente, “maria”.

return

O comando “return” termina o método em execução, retornando o controle para quem chamou.

Ele possui duas formas: com valor de retorno ou sem valor de retorno.

Exemplos:

```
return;
```

ou

```
return 56;
```

Se o “return” devolver algum valor, este deve ser do tipo compatível com o que foi declarado no método. Exemplo:

```
public int teste(int a) {
     return a * 2;
}
```

Ele não poderia retornar um String ou um “float” porque o valor declarado foi um “int”.

if

O “if” testa expressões. Na verdade nós já o vimos neste material. A sua sintaxe é:

```
if(<condição>) {
     comandos1;
}
else {
     comandos2;
}

ou

if(<condição>)
     comando1;
else
     comando2;
```

A “condição” pode ser simples ou composta. Uma condição simples seria:

```
if(x >= 3) {
```

Uma condição composta é interligada por operadores lógicos (veja “Comparadores Lógicos” e “Operadores Condicionais” neste material).

Existem operadores lógicos normais e “short-circuit”:

!	Negação (NOT)
&	Conjunção (E)
|	Disjunção (OU)
^	Disjunção exclusiva (XOR)
&&	Conjunção Short-Cirtuit
||	Disjunção Short-Cirtuit

A diferença entre os operadores “Short-Cirtuit” e os normais é que a segunda expressão pode não ser avaliada dependendo do resultado da primeira.

Exemplo:

```
Arquivo: shortcircuit.java
--------------------------
1> public class shortcircuit {
2>   public static void main(String args[]) {
3>        int x = 0;
4>        int y = 0;
5>        int z = 0;
6>        int t = 0;
7>
8>        if(x++ != 0 && y++ != 0) {
9>             System.out.println("Condição 1");
10>       }
11>
12>       System.out.println("X = " + x + ", Y = " + y);
13>
14>       if(z++ != 0 & t++ != 0 ) {
15>            System.out.println("Condição 2");
16>       }
17>
18>       System.out.println("Z = " + z + ", T = " + t);
19>
20>  }
21> }
```

O resultado é:

X = 1, Y = 0
Z = 1, T = 1

O motivo é que no primeiro “if” utilizamos o operador “short-circuit”, logo, como a primeira condição foi falsa, a segunda sequer foi avaliada. Logo, a variável “y” não foi incrementada.

Com o operador normal “&” ambas as condições foram avaliadas e a variável “t” foi incrementada.

switch

Este é um comando que permite a implementação da estrutura “case” do método Jackson. Ele é muito semelhante ao seu homônimo do C/C++ e ao comando “SELECT/CASE” do Visual Basic.

A sintaxe básica é:

```
switch (expressão inteira) {
    case expressão inteira:
         comando(s)
         break;
    ...
    default:
         comando(s)
         break;
}
```

A expressão avaliada pelo Switch tem que ter como resultado um valor inteiro. Caracteres ou strings não são permitidos. O “switch” apresenta o mesmo comportamento do seu homônimo em “C”: uma vez em um “case” ele desce até encontrar um “break”. Por exemplo:

```
a = 1;
switch(a) {
     case 1:
          System.out.println(“Valor = 1”);
     case 2:
          System.out.println(“Valor = 2”);
}
```

Neste exemplo anterior ambas as mensagens seriam impressas na console, mesmo sendo o valor de “a” = 1. Para evitar isto deve-s acrescentar um “break” ao final do código de cada “case”:

```
a = 1;
switch(a) {
     case 1:
          System.out.println(“Valor = 1”);
          break;
     case 2:
          System.out.println(“Valor = 2”);
          break;
     default:
          System.out.println(“Outro Valor”);
}
```

Neste caso a execução entrará pelo primeiro “case” e parará após a impressão da primeira mensagem, sendo desviada para após o fechamento do bloco do “switch”.

O tipo da expressão do “switch” deve ser: char, byte, short, ou int e o tipo de dados das expressões dos “cases” tem que ser compatíveis com o tipo do “switch”. Não pode haver mais de um “case” com o mesmo valor. E, finalmente, os valores dos “cases” tem que ser constantes, não podendo ser utilizadas variáveis.

Por exemplo, o seguinte código daria erro de compilação:

```
Arquivo: teste51.java
-------------------------
1> public class teste51 {
2>   public static void main(String args[]) {
3>        char a = 'A';
4>        switch (a) {
5>             case 'A': System.out.println("A");
6>                  break;
7>             case 65: System.out.println("65");
8>                  break;
9>        }
10>  }
11> }
```

Pois o valor ‘A’ e o valor inteiro 65 são iguais.

Existe o label “default”, para onde a execução é desviada caso nenhum “case” satisfaça a expressão do “switch”:

```
a = 1;
switch(a) {
     case 1:
          System.out.println(“Valor = 1”);
          break;
     case 2:
          System.out.println(“Valor = 2”);
          break;
     default:
          System.out.println(“Outro Valor”);
}
```

Outra coisa importante é que os comandos de um “case” não devem estar entre chaves, veja o exemplo:

```
Arquivo: teste51.java
-------------------------
1> public class teste51 {
2>   public static void main(String args[]) {
3>        char a = 'A';
4>        int b = 0;
5>        switch (a) {
6>             case 'A':
7>                  b = 1;
8>                  System.out.println("A " + b);
9>                  break;
10>            case 'B':
11>                 b = 2;
12>                 System.out.println("B " + b);
13>                 break;
14>       }
15>  }
16> }
```

Neste caso temos mais de um comando por “case” e, mesmo assim, não colocamos chaves para separá-los, como faríamos em um “if” ou “while”.

Excessões

As excessões ou “Exceptions” (daqui para a frente) são desvios do caminho da execução, normalmente causados por situações de erro. Uma situação de erro pode ser um arquivo não encontrado (“FileNotFoundException”) ou um Cast inválido (“ClassCastException”).

Quando ocorre uma exception, o fluxo do programa é desviado para um bloco de comandos que pode tratar o fato ocorrido. Este bloco é chamado de “Exception-Handling block” ou “bloco de manuseio de excessão”.

Se um bloco apropriado não puder ser encontrado no método em questão, então o JVM irá procurar no método que chamou. E assim por diante até “explodir” na cara do usuário. As vezes é isto mesmo que desejamos... É melhor um programa que “exploda” do que um que tente tratar uma exception e nada faça.

Podemos lançar exceptions quando uma situação não vai bem no nosso código. Para isto temos o comando “throw”. Também podemos tratar exceptions com os blocos de comando: “try/catch” e “try/finally”. Ou podemos especificar que nosso método lança exceptions com a declaração “throws”.

Exemplos:

```
public list () {
     Calendar cal = Calendar.getInstance();
     Date dtnow = cal.getTime();
     try {
          fw = new FileWriter("report.txt");
          bw = new BufferedWriter(fw);
          bw.write("report parser\r\nRun date=" + dtnow +
               "\r\n");
     }
     catch (IOException e) {
          System.out.println("**** IOEXCEPTION: " +
               e.getMessage());
          System.exit(999);
     }
}
```

O exemplo anterior lida com I/O, logo, pode gerar alguma exception. Isto deve ser tratado e está sendo feito pelo bloco “try/catch”. Caso algum problema aconteça nos comandos entre a abertura “{“ e o fechamento “}” do “try”, a execução será desviada para o bloco “catch”.

Se quiser um exemplo mais completo, abra o arquivo “exceptions.java”.

Checked e Unchecked exceptions

A classe “java.lang.Throwable” é a mãe de todas as exceptions. Ela possui duas sub-classes:

- java.lang.Error
- java.lang.Exception

A classe “Exception” tem uma sub-classe “java.lang.RuntimeException**”.**

Todo programa comercial deve se proteger contra os problemas que podem acontecer em um ambiente normal. Logo, deve reagir adequadamente a coisas como problemas de IO, por exemplo. O Java reforça este conceito em tempo de compilação, obrigando o programador a tratar estes erros. As Exceptions cujo tratamento é obrigatório são chamadas de “Checked Exceptions”.

Agora, problemas aleatórios, que não deveriam ocorrer em um ambiente normal, não podem ser previstos. Logo, o Java os diferencia através do conceito de “Unchecked Exceptions”. O programador não é obrigado a tratar estes problemas.

Quais são as “Checked Exceptions”?

É a classe “java.lang.Exception” e todas as suas descendentes, ***exceto a classe “java.lang.RuntimeException”***.

Exemplos de “Checked Exceptions:

- java.lang.Exception
- ClassNotFoundException
- InterruptedException
- IOException
- ParseException
- SQLException

Se você utilizar algum método de alguma classe que lance qualquer “Checked Exception”, você é obrigado a tratar a exception.

Quer um exemplo? Abra o exemplo “exceptions.java” e comente o “try {}“ e o “catch () {}”. Como abaixo:

```
public class exceptions {
     public void trata() throws MinhaException, Exception {
          throw new MinhaException("teste");
     }
     public static void main(String args[]) {
     //   try {
               new exceptions().trata();
     //   }
     //   catch (Exception ex) {
     //        System.out.println(ex.getMessage());
     //   }
     }
}
```

Você obterá um erro de compilação indicando que a exception “MinhaException” não está sendo tratada. Isto se deve ao fato de que o método “trata” declarou que lança a Exception “MinhaException” ou a Exception “Exception”.

Quais são as “Unchecked Exceptions”?

É a classe “java.lang.Error”, seus descententes, e a classe “java.lang.RuntimeException” e seus descendentes.

Blocos try/catch

Um bloco “try” é chamado de bloco “protegido” porque, caso ocorra algum problema com os comandos dentro do bloco, a execução desviará para os blocos “catch” correspondentes.

A sintaxe é:

```
try {
     comando(s);
}
catch (ClasseDeException VariavelException) {
     comando(s);
}
carch (OutraClasseDeException OutraVariavelException) {
     comando(s);
}
```

Veja um exemplo mais claro:

```
void saveNetDevices() {
      try {
            FileOutputStream fos = new
                  FileOutputStream("NetDevices.txt");
            ObjectOutputStream oos = new ObjectOutputStream(fos);
            oos.writeObject(mNetDevices);
            oos.close();
            fos.close();
      }
      catch (NullPointerException nexc) {
            JOptionPane.showMessageDialog(null,
             "NetDevices list is null - No devices saved",
            "Saving NetDevices",
             JOptionPane.INFORMATION_MESSAGE);
      }
      catch (InvalidClassException nexc) {
            JOptionPane.showMessageDialog(null,
             "NetDevices list is invalid - No devices saved",
            "Saving NetDevices", JOptionPane.INFORMATION_MESSAGE);
      }
      catch (NotSerializableException nexc) {
            JOptionPane.showMessageDialog(null,
             "NetDevices list is not serializable - No devices saved",
            "Saving NetDevices",
            JOptionPane.INFORMATION_MESSAGE);
      }
      catch (IOException nexc) {
            JOptionPane.showMessageDialog(null,
            "IO Exception saving NetDevices list - No devices saved",
            "Saving NetDevices", JOptionPane.INFORMATION_MESSAGE);
      }
}
```

Neste exemplo o bloco protegido pode gerar vários tipos de exceptions. Caso ocorra uma “InvalidClassException”, a execução será desviada para o segundo bloco “catch” e, após executar o seu código, sairá para o fim do método.

A ordem em que colocamos os blocos “catch” é muito importante: deve ser da mais específica para a mais genérica. Exemplo:

```
try {
     comando(s);
}
catch (Exception e) {
     comando(s);
}
catch (FileNotFoundException f) {
     comando(s);
}
```

Supondo que ocorreu uma “FileNotFoundException” dentro do bloco “try”, a execução vai procurar o primeiro bloco “catch” que seja compatível com o tipo de Exception ocorrida. O primeiro bloco será analisado e, como Exception é superclasse de

“FileNotFoundException”, a execução entrará no primeiro bloco “catch”. O segundo sequer será testado.

Para que tudo funcione temos que trocar a ordem dos blocos “catch”:

```
try {
     comando(s);
}
catch (FileNotFoundException f) {
     comando(s);
}
catch (Exception e) {
     comando(s);
}
```

Assim, caso ocorra uma “FileNotFoundException” o primeiro bloco será acionado. Se ocorrer uma outra classe de Exception, por exemplo: “InterrupedIOException”, ele entrará no segundo bloco “catch”.

Se você usa algum método que lança “Checked Exceptions” você deve, obrigatóriamente, tratar esta Exception. Ou seja, tem que haver um “catch” correspondente à Exception que você está lançando ou que seja superclasse desta. Veja um exemplo:

```
     try {
          FileReader fin = new FileReader("oids.txt");
          BufferedReader br = new BufferedReader(fin);
          while((mOID = br.readLine()) != null) {
               oidsList.add(new String(mOID));
          }
          br.close();
          fin.close();
     }
     catch (FileNotFoundException fnf) {
          JOptionPane.showMessageDialog(null,
          "Extra OIDS file not found",
          "Loading Extra OIDs",
          JOptionPane.INFORMATION_MESSAGE);
     }
```

O construtor da classe FileReader pode lançar “FileNotFoundException” caso o arquivo não exista. Logo, se utilizar esta classe, você tem que tratar esta exception. Por isto o comando está dentro de um bloco “try/catch”.

Bloco “finally”

As vezes é necessário executar um código mesmo que tenha havido uma Exception. É para isto que servem os blocos “finally”. Sua sintaxe é:

```
try {
     comando(s);
}
finally {
     comando(s);
}

ou

try {
     comando(s);
}
catch (ClasseDeException VarException) {
     comando(s);
}
finally {
     comando(s);
}
```

Os comandos dentro de um bloco “finally” serão executados de qualquer maneira, mesmo que a execução tenha passado por um bloco “catch”.

Ao contrário dos blocos “catch” você só pode ter um bloco “finally” por cada bloco “try”.

Cláusula throws

Se você escreveu um método que lança uma “Checked Exception” (seja por conta própria ou por utilizar um método que lança uma delas), você deve escrever um bloco “catch” correspondente. Caso contrário o Java apontará isto como erro de compilação.

O problema é que as vezes você não quer tratar a Exception. Não te interessa tratá-la ou você quer que ela seja tratada por quem chamou o seu método. Para isto existe a cláusula “throws”, que deve ser especificada na declaração do seu método:

```
public void meumetodo() throws IOException {
     comando(s);
}
```

Esta declaração informa que o método “meumetodo” poderá lançar uma exception da classe “IOException”. Repare que ele não tem “try/catch” para os comandos. Caso uma

Exception seja lançada, será responsabilidade de quem o invocou o seu correto tratamento.

A declaração “throws” diz que o método ***poderá*** lançar uma Exception, não que ele ***irá*** lançá-la.

Logo, ao utilizar um método que lança uma “Checked Exception” você pode optar por utilizar um “try/catch” ou um “throws”. Os dois trechos a seguir são equivalentes:

```
public void meumetodo() {
     try {
          FileReader fin = new FileReader("oids.txt");
          BufferedReader br = new BufferedReader(fin);
          while((mOID = br.readLine()) != null) {
               oidsList.add(new String(mOID));
          }
          br.close();
          fin.close();
     }
     catch (FileNotFoundException fnf) {
          JOptionPane.showMessageDialog(null,
          "Extra OIDS file not found",
          "Loading Extra OIDs",
          JOptionPane.INFORMATION_MESSAGE);
     }
}
```

ou

```
public void meumetodo() throws FileNotFoundException {
     FileReader fin = new FileReader("oids.txt");
     BufferedReader br = new BufferedReader(fin);
     while((mOID = br.readLine()) != null) {
          oidsList.add(new String(mOID));
     }
     br.close();
     fin.close();
}
```

E se o método lançar mais de uma “Checked Exception”?

Você pode especificar mais de uma “Checked Exception” dentro da cláusula “throws”:

```
public void meumetodo()
     throws FileNotFoundException, SocketException {
}
```

A ordem deve ser a mesma utilizada em blocos “catch”: do mais específico para o mais genérico.

Lançando Exceptions

Podemos lançar exceptions com o comando “throw”:

```
throw new Exception(“Erro”);
```

É importante que a construção e o lançamento da Exception sejam na mesma linha, de modo a não afetar o Stack Trace do Java. Veja o exemplo “exceptions.java”. Neste momento a execução desviará para um bloco “catch” ou para quem chamou o método. Se você lançar “Checked Exceptions” seu método deverá declarar isto com o “throws”.

Fluxo de execução de exceptions

- Se der uma exception em um bloco “try” a execução desviará para o bloco “catch” correspondente. A correspondência se dá pela Classe da Exception: ou igual ou superclasse. Se encontrar, a execução desviará para ele e depois irá para fora de todos os blocos “catch” subseqüentes.
- Se der uma exception em um bloco “catch” a execução sairá do método, indo para um bloco “catch” correspondente no método que o invocou.
- Dando ou não exception em um bloco “try/catch”, o bloco “finally” será sempre executado.
- Se der exception em um bloco “finally” a execução sairá do método sem ter concluído os seus comandos.
- Se a execução sair do método ela procurará um bloco “catch” com a exception correspondente no método chamador. Se não encontrar, sairá do método chamador também.

Assertivas de depuração

O Java 1.4 introduziu um novo comando chamado “assert”. Ele serve para verificarmos se determinada condição está correta.

Existem duas sintaxes:

- assert <expressão booleana>
- assert <expressão booleana>, <expressão detalhe>

A expressão booleana é a condição que deve ser analisada. Se for falsa, um Error “AssertionError” será lançado.

Se você utilizar a segunda forma, o valor da expressão detalhe será inserido no detalhe do AssertionError gerado.

Exemplos:

```
assert e > 0;
assert y <= 12, “erro em Y”;
assert validar(z,x,t), formatarmensagem(t);
```

Para você compilar uma classe que utilize o comando assert você tem que especificar isto no comando “javac”:

```
javac -source 1.4 minhaclasse.java
```

As assertions sempre serão desabilitadas em tempo de execução. Para habilitar assertions em tempo de execução especifique no comando “java”:

```
java –ea nomedaclasse
ou
java –enableassertions nomedaclasse
```

Você pode habilitar a execução de assertions apenas em determinados packages:

```
java –ea:com.teste.funcoes –da:com.teste.gerais minhaclasse
```

Neste exemplo as assertions foram habilitadas para a classe “funcoes”, do pacote “com.teste” mas foram desabilitadas para a classe “gerais”, do mesmo pacote.

As classes do sistema (aquelas que não são carregadas explicitamente) também podem ter suas assertions habilitadas ou não:

```
java –esa minhaclasse
java –enablesystemassertions minhaclasse
java –esa –ea:com.teste minhaclasse
```

Para desabilitar as assertions de classes do sistema utilizamos:

```
java –dsa minhaclasse
java –disablesystemassertions minhaclasse
```

Onde utilizar ou não assertions

Pelo fato de poderem ser desabilitadas externamente, as assertions não devem ser utilizadas indiscriminadamente. Existem casos específicos para os quais as assertions são recomendadas.

Internal Invariants

Condições que sempre serão verdadeiras. Normalmente colocamos comentários indicando isto no código. Exemplo:

```
if(tipo == 1) {
     comando(s);
}
else
     if(tipo == 2) {
          comando(s);
     }
assert tipo ==1 || tipo ==2;
```

Control-flow invariants

São pontos do programa em que nunca deveríamos entrar. Exemplo:

```
void procura(String x) {
     for(...) {
          if() {
               return;
          }
     }
     // a execução chegaria aqui
     assert false;
}
```

Preconditions

São pré-condições iniciais para a execução de determinados métodos. O padrão da programação-por-contrato é que se uma regra de interface foi quebrada, o método deveria lançar uma exception. ***Não use assertions para preconditions***. O normal é que isto aconteça:

```
void procura(String x) throws Exception
{
     if(x == null) {
          throw new Exception (“erro”);
     ...
```

Postconditions

Após a execução de um método e antes de retornar um resultado, você pode utilizar assertions para validar a execução:

```
int calcula(int x) {
     int z = 0;
     comando(s);
     // teste
     assert z < x;
     return z;
}
```

Class Invariants

É uma condição que deve ser verdadeira para todas as instâncias de uma determinada classe. Caso seja falsa, algo errado está acontecendo. Todos os métodos publicos (e o construtor) deveriam testar a condição. Veja um exemplo:

```
class Cliente {
     private boolean mCredito;
     public boolean temCredito() {
          return mCredito;
     }
     public Comprar {
          assert temCredito();
...
```

Para todas as situações, exceto Preconditions, as assertions podem ser utilizadas. Devemos lembrar que as assertions podem ser desabilitadas em tempo de execução, logo, em alguns casos, temos que rever a politica de utilização. Por exemplo, no caso da situação de crédito do cliente, a assertion não deveria ser o único método para evitar uma compra fraudulenta.

Questões e Exercícios:

1) Considere o seguinte código em Java:

```
Arquivo: exflux01.java
-------------------------
1> public class exflux01 {
2>    public static void main(String args[]) {
3>          int a = 0;
4>          while(a = obterValor()) {
5>          }
6>    }
7>    public static int obterValor() {
8>    }
9> }
```

a) Compilará e rodará sem problemas
b) Compilará mas dará exception
c) Dará erro de compilação na linha 7
d) Nenhuma das anteriores

2) Considere o seguinte código em java:

```
Arquivo: exflux02.java
-------------------------
1> public class exflux02 {
2>    public static void main(String args[]) {
3>          int a = 0;
4>          int b = 0;
5>          for(a=1,b=50;a < b; a++,b--) {
6>                System.out.println("a = " + a +
7>                      ", b = " + b);
8>          }
9>
10>   }
11> }
```

a) Dará uma exception na linha 5
b) Dará uma exception na linha 6
c) Dará erro de compilação na linha 5
d) Compilará e rodará sem problemas

3) Considere o seguinte arquivo em Java:

```
Arquivo: exflux03.java
-------------------------
1> public class exflux03 {
2>    public static void main(String args[]) {
3>          String nome = args[0];
4>          char Char[] = {'"','<','>'};
5>          String saida = "";
6>          primeiro:
7>          for(int i=nome.length()-1;i > -1; i--) {
8>                teste: for(int Instanceof=0;
9>                      Instanceof < Char.length;
10>                     Instanceof++) {
11>                     if(nome.charAt(i) == Char[Instanceof]) {
12>                           continue primeiro;
13>                     }
14>               }
15>               saida += nome.charAt(i);
16>         }
17>         System.out.println(saida);
18>   }
19> }
```

a) Dará erro de compilação nas linhas 6 e 12
b) Dará erro de compilação nas linhas 6, 8 e 12
c) Compilará mas dará exception na linha 11
d) Dará erro de compilação nas linhas 8, 9, 10 e 11
e) Nenhuma das anteriores

4) Exatamente o quê o programa do exemplo 4 faz?

5) Analise o seguinte código Java:

```
Arquivo: exflux04.java
-------------------------
1> public class exflux04 {
2>    public static void main(String args[]) {
3>          int a = 4;
4>          int b = 7;
5>          for(int c=10; c>0; c--) {
6>                if((a++ > 100) && (++b > 1)) {
7>                      System.out.println("OK");
8>                      break;
9>                }
10>               System.out.println(a +
11>                           "," + b);
12>         }
13>   }
14> }
```

a) Os valores finais de “a” e “b” serão respectivamente “14” e “17”
b) Dará exception na linha 6
c) Dará erro de compilação na linha 10
d) O valor final de “c” será “1” e o de “a” será “14”

6) Analise o seguinte código em Java:

```
Arquivo: exflux05.java
-------------------------
1> import java.io.*;
2>
3> class teste {
4>    public void metodo1(String arg1) throws IOException {
5>          System.out.println(arg1);
6>    }
7> }
8>
9> class teste2 extends teste {
10>   public void metodo1(String x) throws Exception {
11>         System.out.println(x);
12>   }
13>   public void metodo2(int x) throws FileNotFoundException {
14>   }
15> }
16>
17> class teste3 extends teste2 {
18>   public void metodo2(int y) {
19>   }
20> }
21>
22>
23> public class exflux05 {
24>   public static void main(String args[]) {
25>         teste t = new teste();
26>         teste2 t2 = new teste2();
27>         teste3 t3 = new teste3();
28>         t2.metodo2(5);
29>   }
30> }
```

a) Compilará e rodará sem problemas
b) Dará exception na linha 28
c) Apresentará erro de compilação nas linhas 10 e 28
d) Só dará erro na linha 10

Respostas

1) letra “d”. O “while só permite expressões com resultado boolean e a do exemplo é “int”.

2) letra “d”. O “for” permite incrementar mais de uma variável. Neste caso tanto “a” quanto “b” são variáveis de controle do “for”.

3) letra “e”. Não há erro algum. Note que “Instanceof” não é palavra reservada e que o programa está utilizando “labels” para os dois comandos “for”.

4) Pega um string informado como primeiro argumento (args[0]) e o imprime invertido, retirando os caracteres aspas-duplas, menor-que e maior-que.

5) letra “d”. Como as duas expressões no “if” são ligadas pelo operador AND short-circuit, a segunda expressão NUNCA será avaliada, logo, “b” nunca será incrementado.

6) letra “c”. Primeiramente o compilador dará erro na linha 10, pois o método original em “teste” levanta a exception “IOException” e o método sobrescrito em “teste2” está levantando “Exception”, que não é “filha” de “IOException. Depois, ao ser corrigido, dará erro na linha 28 porque o método2 em “teste2” levanta a exception “FileNotFound” que não está sendo levantada nem interceptada pelo método “main”.

Classes e Objetos

Exemplos em “exemplos/06”.

Esta parte do exame aborda:

- Overload (sobrecarga) de métodos
- Override (sobrescrita) de métodos
- Runtime Type (tipo de dados em tempo de execução
- Object Orientation (Orientação a Objetos)

Nota-se que muitos termos de Java (e de TI em geral) não possuem uma tradução coerente em português. Não se deve traduzir todos os termos indiscriminadamente. Por exemplo, a maioria dos programadores sabe o que significa “fazer um Override” de um método, mas poucos sabem o que significa “sobrescrever” um método.

Orientação a Objetos

Programação Orientada a Objetos é resolver problemas através do seu particionamento em Classes e Mensagens entre elas.

Não é a intenção deste trabalho ensinar os conceitos e fundamentos da OOP mas, como caem no exame da SUN, é bom clarificar algumas coisas.

Classe:

Tipo de dados abstrato que define métodos e propriedades que uma determinada classe de objetos deve possuir. Exemplos: “Veiculo” (assentos retirados propositadamente). Uma classe não possui memória alocada, sendo apenas um “gabarito” ou um tipo de dados.

Toda classe possui um conjunto de Métodos e Propriedades que formam a sua “interface” com o mundo externo. Quando desenvolvemos uma classe temos que manter o compromisso de não alterar a interface, caso contrário estaremos prejudicando os usuários de nossa classe. Se ela estiver bem “encapsulada” diminuímos este risco.

Objeto:

Um Objeto é uma classe instanciada. Ou seja, é uma instância real, com memória alocada, de uma classe pré-existente. Exemplo:

```
String xpto = new String(“teste”);
```

Este código definiu uma variável cuja classe é “java.lang.String” e a instanciou. Após a execução do comando passa a existir na memória uma variável “xpto” que aponta para um objeto String no “heap” (memória comum).

Propriedade:

Uma propriedade é uma variável com visibilidade externa a uma classe.

Método:
É uma função, pertencente à classe, que pode ser utilizada externamente. É um "comportamento" que a classe implementa.

Abra os exemplos "Cliente.java" e "usacliente.java" e examine a sua implementação.

Relacionamentos entre Classes

Existem dois relacionamentos básicos entre classes: "is a" (é um(a)) and "has a" (tem um(a)).

Eles podem ser denominados:

Gen-Spec (generalização-especialização) ou "is a":
Um Carro é um Veículo.

Todo-Parte (composição) ou "has a":
Um Carro tem um motor.

Suponha o seguinte exemplo:

"Um CLIENTE é uma PESSOA que possui uma CONTA"

Podemos implementar isso em java:

```
class Pessoa {
     // propriedades e métodos
}

class Conta {
     // propriedades e métodos
}

class Cliente extends Pessoa {
     public Conta contacorrente;
}
```

O relacionamento das classes "Cliente" e "Pessoa" é do tipo "is a" ou Gen-Spec. Ele pode ser expresso em Java de duas formas: herança ou interface. Utilizando a maneira mais simples, um Cliente é uma sub-classe de Pessoa ("class Cliente extends Pessoa").

O relacionamento das classes "Cliente" e "Conta" é do tipo "has a" ou Composição. Ele pode ser expresso em Java através de propriedades. Um Cliente possui uma instância da classe Conta ("public Conta contacorrente").

Princípios da Orientação a Objetos

Em OOP temos três conceitos básicos:

Herança	A capacidade de uma Classe herdar métodos e propriedades de uma classe ancestral. Ela pode acrescentar ou modificar o comportamento de sua ancestral.
Encapsulamento	A capacidade que uma Classe tem de ocultar a sua própria implementação, apresentando ao "cliente" que a utiliza apenas uma interface simplificada.
Polimorfismo	A habilidade de métodos com mesmo nome apresentarem comportamento diferente em classes diferentes (porém derivadas de um mesmo ancestral).

Para uma linguagem de programação ser considerada "Orientada a Objetos" é necessário implementar mecanismos que permitam utilizar estes três conceitos básicos.

Herança

Abra o exemplo "veiculos.java".

Nele é criado um modelo de objetos baseado na classe "Veiculo". Nota-se que as classes "Carro" e "Moto" são filhas da classe "Veiculo". Isto é denotado pelo uso da palavra-chave "extends". As classes "Carro" e "Moto" estendem (ou suplementam) a classe "Veiculo".

Em Java temos apenas a herança simples, ou seja, uma Classe só pode possuir um só ancestral. Porém podemos utilizar Interfaces para simular Herança Múltipla, o que veremos mais adiante.

Encapsulamento

Desde as técnicas de projeto estruturado de Meilir Page-Jones se fala em diminuir o acoplamento entre módulos.

Acoplamento é a ligação excessiva e indesejada entre dois módulos. Se for necessário alterar um deles o outro também o será. Isto gera multiplicidade de trabalho e aumento de pontos de risco em um sistema.

Uma maneira de diminuir o acoplamento é criar "Caixas Pretas" das quais só se conhece a interface.

Um exemplo de função "Caixa Preta" seria:

```
public boolean VerificarCPF(String CPF);
```

Não se sabe COMO a função verifica o CPF, ou seja, os detalhes de sua implementação estão “encapsulados” ou ocultos dentro da própria função e inacessíveis para o usuário externo.

Em Java podemos “encapsular” propriedades e métodos através dos Modificadores de acesso (Private, Protected etc), ocultando a implementação de determinada funcionalidade.

Abra os arquivos “veiculos.java” e “veiculos_encapsulados.java”. Note a diferença de implementação das propriedades “placa” e “marca”. Na classe original temos:

```
class veiculo {
     public String placa;
     public String marca;
     public void ligar() {
          System.out.println("\nLigando veículo");
     }
}
```

Note que as propriedades estão implementadas como variáveis públicas. Logo, o usuário da nossa classe “conhece” a implementação da mesma. Dizemos que a classe está mal encapsulada.

O que aconteceria se resolvêssemos mudar a implementação da propriedade “placa” de String para uma outra classe? Estaríamos “quebrando o contrato” com nosso cliente atrabés da alteração da interface. O que não acontece com a segunda versão da mesma classe:

```
class veiculo {
     protected String placa;
     protected String marca;
     public void ligar() {
          System.out.println("\nLigando veículo");
     }
     public String getPlaca() {
          return this.placa;
     }
     public void setPlaca(String mPlaca) {
          this.placa = mPlaca;
     }
     public String getMarca() {
          return this.marca;
     }
     public void setMarca(String mMarca) {
          this.marca = mMarca;
     }
}
```

Nesta segunda implementação nota-se que as variáveis “placa” e “marca” não podem mais ser acessadas do “mundo exterior”, pois seus modificadores de acesso agora são “protected” e não “public”. Como o usuário vai modificar as propriedades?

Foram criados os métodos “get” e “set” para cada propriedade. Assim temos “getPlaca” e “setPlaca” para a propriedade “placa” e “getMarca” e “setMarca” para a propriedade “marca”. Para usar a classe temos que fazer assim:

```
veiculo v = new veiculo();
v.setPlaca(“XXX-2222”);
v.setMarca(“Ford”);
System.out.println(v.getMarca() + “ “ + v.getPlaca());
```

Desta forma conseguimos “ocultar” ou “encapsular” a implementação das duas propriedades. Se quiséssemos alterar seu tipo de String para outra classe (podemos criar uma classe específica para Placas de Carro), não teríamos problemas com os nossos usuários. Além disto podemos implementar algumas validações no momento de setar ou informar a propriedade (nos métodos “setPlaca” e “setMarca”).

Polimorfismo

Polimorfismo é o comportamento diferenciado de dois objetos quando recebem a mesma mensagem. É comum nos referirmos a chamadas de métodos de um objeto como envio de mensagens a ele. Isto deriva da linguagem SmallTalk.

Considere o exemplo do arquivo “veiculos.java”. Nele temos as classes “veiculo”, “carro” e “moto”. Todos são auto-motores, logo, devem possuir um método que permita dar a partida no motor (“ligar”).

A classe “veiculo” é a ancestral comum de “carro” e “moto”. O papel de uma classe ancestral é definir o comportamento básico de todos os descendentes de “veiculo”. Porém, nem todos os “veiculos” são ligados da mesma maneira, logo, existe apenas uma implementação básica do método “ligar” na classe “veiculo”, deixando para as classes-filhas a sua própria implementação.

Note que as classes “carro” e “moto” possuem suas próprias versões do método “ligar”. Em tempo de execução é avaliado o Run-time type (ou tipo em tempo de execução) de cada objeto e o método “ligar” apropriado será invocado. Isto é polimorfismo. Apesar de serem todas filhas de “veiculo” implementam o método “ligar” de maneira diferente.

Para que isto seja possível é necessário que os métodos sejam Virtuais (possam ser sobrescritos – override em tempo de execução).

Herança x Interfaces

Herança é o fato de uma classe ser definida sobre outra, recebendo desta os métodos e propriedades já existentes. Em Java uma classe pode ser descendente de uma e somente uma classe. Em outras linguagens, como C++, temos o conceito de herança múltipla, onde uma classe pode ser descendente de mais de uma outra classe.

E se for necessário que uma classe tenha este comportamento? Em outras palavras, e se ela tiver mais de uma ancestral? Para resolver este problema o Java emprega o conceito de interface.

Segundo o tutorial da SUN, uma Interface é uma coleção definida (com nome) de declarações de métodos. Ela também pode declarar constantes (final).

Podemos pensar em uma interface como uma classe “oca”, ou seja, apenas uma casca vazia. Veja um breve exemplo:

```
interface automotor {
     void ligar();
     void desligar();
     int verCombustivel();
}
```

A interface “automotor” declara os métodos “ligar”, “desligar” e “verConbustivel”. Repare que ela não define os métodos, ou seja, eles não tem a sua implementação.

Depois que definimos uma interface podemos criar classes que a implementam. O relacionamento entre classe e interface força a primeira a implementar os métodos declarados na segunda. Em outras palavras, uma interface é um “contrato” listando os métodos que uma classe DEVE implementar.

Se uma classe implementa uma interface e não define TODOS os seus métodos, o compilador Java insistirá que ela deve ser declarada como ABSTRATA (*veja Abstract*).

Uma classe implementa uma interface (ou várias) através da palavra-chave “implements”:

```
class veiculo implements automotor {
     public String placa;
     public String marca;
     public void ligar() {
          System.out.println("\nLigando veículo");
     }
     public int verCombustivel() {
          return 999;
     }
     public void desligar() {
          System.out.println("\nDesligando veículo");
     }
}
```

Repare que a classe “veiculo” deste exemplo declara que implementa a interface “automotor” e, devido a isto, deve implementar TODOS os métodos declarados em “automotor”, à saber: “ligar”, “desligar” e “verCombustivel”.

Podemos declarar dentro de uma interface:

- **Constantes** (variáveis declaradas como “final” – *veja em Final*). Todas as constantes declaradas em uma interface são implicitamente “public”, “static” e “final”.
- **Métodos** podem ser declarados sem a implementação (“{“ e “}”), além disto não devem ser utilizados modificadores “transient”, “volatile”, “synchronized”, “private” e “protected”.

Veja o exemplo completo em “veiculos_interface.java”.

Uma classe pode implementar mais de uma interface, o que imita o conceito de herança múltipla. Veja um exemplo:

```
class teste implements interf1,interf2
```

Podemos comparar Interface com Herança:

Interface	**Herança**
Uma classe pode implementar mais de uma interface	Uma classe só pode ter uma ancestral.
Uma classe é obrigada a implementar TODOS os métodos da interface, caso contrário será considerada como ABSTRATA.	Uma classe já recebe a implementação de TODOS os métodos de sua ancestral. Ela pode, opcionalmente, fazer um Override dos métodos herdados.
Uma interface declara métodos e/ou constantes, sem implementação.	Uma classe ancestral pode ser totalmente funcional.
Métodos em uma interface não podem ser “private” ou “protected”. Além disto os modificadores: “transient”, “volatile” e “sinchronized” não podem ser utilizados.	Todos os modificadores podem ser utilizados em uma classe ancestral de outras.

Finalmente, uma interface pode ser utilizada como um tipo de dados. Podemos utilizar uma interface quase da mesma maneira que utilizamos uma classe ou um tipo primitivo de dados:

```
interface automotor {
     void ligar();
     void desligar();
     int verCombustivel();
}

public class veiculos_interface {
     public static void main(String args[]) {
          veiculo v = new veiculo();
```

```
          carro c = new carro();
          moto m = new moto();
          v.ligar();
          c.ligar();
          m.ligar();
          v.desligar();
          c.desligar();
          m.desligar();
          automotor am = m;
     }
}
```

Repare que estamos definindo a variável “am” como sendo do tipo “automotor”. Ela pode receber qualquer classe que implemente a mesma interface, no caso “m” é do tipo “moto”, que estende “veiculo” que, por sua vez, implementa “automotor”.

Só não podemos instanciar objetos do tipo interface com o comando “new”. Isto se deve ao fato de que, por não possuir implementação, uma interface é considerada como uma classe abstrata.

Overload (sobrecarga) e Override (sobrescrita) de Métodos

Existem duas modificações que uma classe pode fazer nos métodos que herdou de outra classe: sobrecarregar ou Overload, e sobrescrever ou Overrride. Note que isto só é possível no caso de herança, já que para interfaces a classe TEM QUE implementar TODOS os métodos declarados nas suas interfaces.

Override

Override é o ato de sobrescrever ou re-escrever um método originalmente declarado e definido em uma classe ancestral. No exemplo: “veiculos_interface.java” temos override do método “ligar”:

```
class veiculo implements automotor {
     public String placa;
     public String marca;
     public void ligar() {
          System.out.println("\nLigando veículo");
     }
     public int verCombustivel() {
          return 999;
     }
     public void desligar() {
          System.out.println("\nDesligando veículo");
     }

}
```

```
class carro extends veiculo implements automotor {
     public void ligar() {
          System.out.println("\nLigando carro");
     }
}
```

Repare que a classe “carro” redefine o método “ligar” (mas não o redeclara!). Ela substitui o CORPO do método definido na ancestral “veiculo” por outro.

Note que a ASSINATURA do método tem que ser igual. Assinatura é composta pelo VALOR DE RETORNO + NOME DO MÉTODO + TIPOS DOS ARGUMENTOS. Assinatura é a maneira que o Java identifica um método.

Overload

Overload é um “truque” com nomes de métodos. Como um método somente é identificado pela sua ASSINATURA, podemos variar a lista de argumentos, criando, assim, métodos diferentes com mesmo nome básico. Veja o exemplo “testeoverload.java”:

```
class revtexto {
      public String inverter(String mTexto) {
         String mRetorno = "";
         for(int i=mTexto.length()-1;i>=0;i--) {
                  mRetorno = mRetorno + mTexto.charAt(i);
         }
         return mRetorno;
      }
      public String inverter(String mTexto,int tamanho) {
         String mCopia = mTexto.substring(0,tamanho);
         mCopia = inverter(mCopia);
         mCopia = mCopia + mTexto.substring(tamanho);
         return mCopia;
      }
}

public class testeoverload {
      public static void main(String args[]) {
         revtexto rt = new revtexto();
         System.out.println("\n "+ rt.inverter("Aprendendo Java"));
         System.out.println("\n "+ rt.inverter("Aprendendo Java", 4));
      }
}
```

Temos dois métodos chamados “inverter”: um com um parâmetro String e outro com dois parâmetros (um String e um int). O Java irá procurar o método considerando o tipo de retorno, o nome do método e os tipos de dados dos argumentos (ou parâmetros) que estamos passando para ele. Quando chamamos:

```
System.out.println("\n "+ rt.inverter("Aprendendo Java", 4));
```

Ele irá procurar um método “inverter” que retorne um String e receba um String e um int.

Existem regras para Sobrescrita ou **Override**:

1. Uma classe não pode sobrescrever métodos marcados como “final” na classe ancestral.
2. Uma classe não pode sobrescrever métodos marcados como “static” na classe ancestral. Se você fizer isto estará apenas OCULTANDO o método da classe ancestral e não o sobrescrevendo.
3. Uma classe não pode sobrescrever métodos de sua ancestral tornando-os menos accessíveis. Ela pode sobrescrevê-los tornando-os mais accessíveis:
    a. Inválido:
        i. Original: public void calcula(x);
        ii. Override: private void calcula(x);
    b. Válido:
        i. Original: protected void calcula(x);
        ii. Override: public void calcula(x);
4. Uma classe não pode sobrescrever métodos de sua ancestral lançando excessões diferentes das do método original. Ela pode lançar excessões DESCENDENTES:

```
class primeira {
  public void teste (int x) throws IOException {

      }
}

class segunda extends primeira {
  public void teste (int x) throws FileNotFoundException {

      }
}
```

   Neste caso é permitido o Override porque “FileNotFoundException” é descendente de “IOException”. Se você inverter (colocar “FileNotFoundException” em primeira.teste e “IOException” em segunda.teste) o compilador dará erro. Isto é porque “IOException” é diferente de “FileNotFoundException” e não é descendente desta.

5. O tipo de retorno deve SER O MESMO do método original.
6. Você deve OBRIGATORIAMENTE sobrescrever TODOS os métodos de uma classe ancestral marcados como “abstract”. Se não fizer isto a sua classe deverá também ser “abstract”.

Como Overload é apenas um truque com nomes, podemos fazer sobrecarga ou Overload em qualquer caso. Note que os métodos SÃO DIFERENTES! Só o nome do método não é suficiente para identificar um método.

Construtores e Finalizadores

Construtores ou “constructors” são utilizados para inicializar o estado de uma instância de uma classe. Todas as classes têm pelo menos um construtor sem argumentos, também conhecido como “no-args” ou “default” constructor.

Se você criar uma classe assim:

```
public class teste {
     private String mVar;
     public mostra() {
     }
}
```

O compilador Java vai providenciar um construtor default para você. Este construtor não inicializará variável alguma.

Se quisermos providenciar inicialização de estado, ou seja, colocar nosso Objeto com um estado interno aceitável, temos que criar nossos próprios construtores. Veja um exemplo:

```
public class teste {
     private String mVar;
     public mostra() {
     }
     public teste() {
          this.mVar = “Valor Inicial”;
     }
}
```

Um construtor NÃO TEM valor de retorno. Nem mesmo **“void”**. Ele deve ter o mesmo NOME da classe, podendo ou não ter argumentos. Você pode utilizar modificadores no construtor:

- “public”: qualquer classe pode instanciar a sua classe.
- “private”: nenhuma classe pode instanciar a sua classe.
- “protected”: somente descendentes ou outras classes no mesmo “package”.
- Default (sem modificador): somente classes no mesmo “package” podem instanciar a sua classe.

Um aspecto muito importante sobre construtores é que eles não são “herdados” como os métodos comuns. Se você não criar um Construtor para a sua Classe, o Compilador criará um Construtor “no-args” padrão.

Se você criar um Construtor qualquer, mesmo que possua argumentos, o Compilador NÃO CRIARÁ O CONSTRUTOR “no-args”. Assim sendo, sua classe terá problemas em alguns processos que esperam o construtor “no-args”, como de-Serialização, por exemplo.

O que um Construtor deve fazer?

Sua função principal é invocar o construtor da classe ancestral imediata e inicializar variáveis desta classe. A ordem deve ser esta mesma. Veja um pedaço do arquivo de exemplo “veiculos_construtores.java”:

```
class veiculo {
     public String placa;
     public String marca;
     public void ligar() {
          System.out.println("\nLigando veículo");
     }
     public veiculo() {
          this.placa = "***";
          this.marca = "***";
          System.out.println("Veículo Inicializado");
     }
}

class carro extends veiculo {
     public String motor;
     public void ligar() {
          System.out.println("\nLigando carro");
     }
     public carro() {
          // O construtor de “veiculo” é chamado aqui
          this.motor = "1.0";
          System.out.println("Carro Inicializado");
     }
     public carro(String xmarca, String xplaca) {
          this.marca = xmarca;
          this.placa = xplaca;
     }
}
```

Veja o Construtor da classe “carro” ele chama o construtor da classe ancestral (como???) e inicializa a variável “motor”. Normalmente Construtores não devem imprimir mensagens, este o faz apenas para demonstração.

Um Construtor deve obrigatóriamente invocar o construtor da classe ancestral imediata, neste exemplo anterior: “veiculo”. Se você não colocar o comando para invocar o construtor do ancestral, o compilador inserirá um para você automaticamente.

Invocando outros construtores

Como já vimos, temos os prefixos “this” e “super” para especificar elementos DESTA (this) classe e da ANCESTRAL (super). Podemos utilizar isto para invocar o construtor da classe ancestral (super) ou outros construtores desta mesma classe (this).

Invocando o construtor da ancestral:

```
public class classe1 extends classe0 {
     public classe1() {
          super();
          System.out.println(“Inicializada”);
     }
}
```

Invocando outros construtores da mesma classe:

```
public class classe2 {
     public String teste;
     public String nome;
     public classe2() {
          teste = “*”;
     }
     public classe2(String xnome) {
          this();
          this.nome = xnome;
     }
}
```

Utilizando o prefixo “super”, sem especificar nome de método ou de propriedade, estamos invocando o construtor da classe ancestral. Se não informarmos argumentos, o construtor “no-args” será invocado.

Utilizando o prefixo “this”, igualmente sem nome de método ou de propriedade, estamos invocando OUTRO construtor da mesma classe. No arquivo-exemplo “veiculos_construtores.java” temos um exemplo disto na classe “moto”:

```
class veiculo {
     public String placa;
     public String marca;
     public void ligar() {
          System.out.println("\nLigando veículo");
     }
     public veiculo() {
          this.placa = "***";
          this.marca = "***";
          System.out.println("Veículo Inicializado");
     }
     public veiculo(String xplaca, String xmarca) {
          this.placa = xplaca;
          this.marca = xmarca;
     }
}
...
```

```
class moto extends veiculo {
     public String cilindrada;
     public void ligar() {
          System.out.println("\nLigando moto");
     }
     public moto() {
          this("****","****");
     }
     public moto(String xplaca, String xmarca) {
          super(xplaca, xmarca);
          this.cilindrada = "250";
     }
}
```

Neste exemplo o construtor “moto()” invoca o construtor “moto(String xplaca, String xmarca)” passando valores default.

Se você não informar “this” ou “super” na primeira linha do seu construtor, o Compilador inserirá um “super” automaticamente. Para ver isto, rode o exemplo “veiculos_construtores.java” e veja a chamada sendo feita ao construtor de “veiculo” quando criamos uma instância de “carro”.

Se você informar “this” ou “super” deve fazê-lo na primeira linha do seu construtor. Se tentar colocar qualquer um dos dois em outra linha, resultará em erro.

Finalizando uma classe

Em Java nós não gerenciamos a memória. Ou seja, criamos instâncias de classe mas não somos responsáveis por alocar memória ou liberar memória. Isto é feito pelo mecanismo de “Garbage Collector”. Ele libera a memória das variáveis.

O problema é que o GC (Garbage Collector) tem “personalidade própria”, ou seja, ele roda em baixa prioridade, liberando memória quando tem janela para isto. Não há como forçar o JVM (Java Virtual Machine) a liberar memória imediatamente.

Quando o GC está prestes a liberar a memória de uma determinada classe, ele tenta invocar o método “finalize()”, definido na classe “Object”, que é a ancestral de todas as classes em Java.

Neste momento podemos sobrescrever o método “finalize()” e proceder a alguma “limpeza geral”, liberando conexões e recursos. Mas não podemos esquecer de incluir um “super.finalize()” como último comando. Veja um exemplo em “veiculos_construtores.java”:

```
class moto extends veiculo {
     public String cilindrada;
     public void ligar() {
          System.out.println("\nLigando moto");
     }
     public moto() {
          this("****","****");
     }
     public moto(String xplaca, String xmarca) {
          super(xplaca, xmarca);
          this.cilindrada = "250";
     }
     protected void finalize() throws Throwable {
          System.out.println("Finalizando Moto");
          super.finalize();
     }
}

public class veiculos_construtores {
     public static void main(String args[]) {
          carro c = new carro("Gol","XXX-1111");
          System.out.println(c.placa + ", " + c.marca +
               ", " + c.motor);
          moto m = new moto("ZZZ-2222","Yamaha");
          m = null;
          System.gc();
     }
}
```

Repare que a assinatura do método “finalize()” em “Object” é: “protected void finalize()”, logo, temos que repetir a mesma assinatura se quisermos fazer uma sobrescrita (Override).

O finalizador somente roda após nularmos o objeto “moto” (“m”) e sugerirmos que o Garbage Collector rode (System.gc).

Métodos herdados de Object

A classe “Object” é a ancestral de todas as classes em Java. Se a sua classe não é “filha” de ninguém, ou seja, não possui a cláusula “extends”, ela é filha de “Object”.

A classe Object fornece algums métodos interessantes, os quais TODAS as classes herdam, eis os mais interessantes:

clone

Cria um novo objeto a partir de um já existente. Somente classes que implementem a interface “Clobable” podem ser “clonadas”. Veja um exemplo:

```
public class xpto implements Clonable {
       public Vector elementos;
       protected Object clone() {
            xpto novo = (xpto)super.clone();
            novo.elementos =
this.elementos.clone();
       }
}
```

Se uma classe invocar:

```
       xpto      a = new xpto();
       xpto      b = a.clone();
```

Vai criar um clone de XPTO com referências a OUTROS objetos.

equals and hashCode

Os métodos “equals” e “hashCode” devem ser sobrescritos juntos, ou seja, se você sobrescrever um deles, deve sobrescrever o outro. O método “equals” permite que você crie o algorítmo de comparação de duas instâncias da sua classe.

Como sabe, a implementação Original de “equals” em “Object” não verifica se dois Objetos são iguais, mas se as duas referências apontam para o mesmo Objeto.

```
public class teste1 {
       String nome1;
       String nome2;
       public boolean equals(Object o) {
       return this.nome1.equals(o.nome1)
            &&
            this.nome2.equals(o.nome2);
       }
}

...
```

```
teste1 a = new teste1();
...
teste1 b = new teste1();
...
if(a.equals(b)) {
}
```

Já o método “hashCode” serve para retornar um valor numérico correspondente a uma classe. O Java estabelece que:

1. Dois objetos iguais (de acordo com o método “equals”) devem retornar o mesmo valor de “hashCode”.
2. Não é requerido que dois objetos diferentes retornem “hashCode” diferentes, embora isto possa comprometer a performance de certas operações.

Eis um exemplo simples de “hashCode”:

```
public class teste1 {
       String nome1;
       String nome2;
       public int hashCode() {
            return nome1.hashCode * 2 +
                   nome2.hashCode * 3;
       }
```

finalize O método “finalize” é invocado pelo GC quando a memória ocupada pelo objeto está para ser liberada.

toString Este método retorna a representação na forma de um String do Objeto. Pode ser o nome da classe ou o seu valor em forma de String. Na implementação default de “Object” ele retornará:

```
getClass().getName() + '@' +
       Integer.toHexString(hashCode());
```

A classe System

Esta é uma classe que não pode ser instanciada e que possui alguns métodos e propriedades úteis. Eis algumas propriedades e métodos Estáticos que podemos utilizar:

static PrintStream err	O output stream de erro.
static InputStream in	O input stream padrão de entrada.
static PrintStream out	O output stream padrão de saída. Podemos utilizar normalmente: "System.out.println()" para imprimir na saída padrão.
static void exit(int status)	Termina a JVM fornecendo um código de erro.
static void gc()	Sugere que o GC deva coletar memória.
static void runFinalization()	Sugere que a JVM deva priorizar a chamada dos métodos "finalize" pendentes.

Tipos especiais de classes (Inner e anonymous)

Nem todas as classes precisam ser "top level". Podemos ter classes definidas dentro de outras classes. Elas são chamadas de "Nested classes". Também podemos ter classes definidas dentro de métodos e até mesmo anônimas (Anonymous classes).

Inner classes

São classes que são definidas dentro de outras classes. Veja um exemplo:

```
public ClasseDeFora {
     public ClasseDeDentro {
     }
}
```

Para instanciar a classe "ClasseDeFora" basta :

```
ClasseDeFora cd = new ClasseDeFora();
```

Para instanciar a classe "ClasseDeDentro" temos que ter uma instância da "ClasseDeFora":

```
ClasseDeFora.ClasseDeDentro cden = cd.new ClasseDeDentro();
```
Ou
```
ClasseDeFora.ClasseDeDentro cden = (new ClasseDeFora()).new
ClasseDeDentro();
```

Por que usaríamos Inner Classes?

Porque ambas são tão ligadas que uma não faz sentido sem a outra. O melhor exemplo que podemos ver é o do arquivo “demotexto.java”. Trata-se de um buffer para armazenar textos que tem a capacidade de permitir uma navegação palavra-por-palavra. Quem permite isto é a Inner-class “palavra”:

```
class texto  {
      private String conteudo;
      void setConteudo(String mtexto) {
            this.conteudo = mtexto;
      }
      class palavra {
            private int posicao;
            private int tamanho;
            private final char limites[] = {' ', ',', '.', '(', '{',
'[', ':', ';', '/', '<', '>', '+', '-', '*'};
            public int getPosicao() {
                  return posicao;
            }
            public String next() {
                  String s = "";
                  int i = posicao;
                  boolean delimiterFound = false;
                  if(posicao >= tamanho) {
                        return "";
                  }
                  for(;i<tamanho; i++) {
                        if(posicao==0) {
                              delimiterFound=true;
                        }
                        if(isDelimiter(conteudo.charAt(i))) {
                              if(delimiterFound) {
                                    posicao = i;
                                    return s;
                              }
                              delimiterFound=true;
                        }
                        else {
                              s += ("" + conteudo.charAt(i));
                        }
                  }
                  posicao = tamanho;
                  return s;

            }
            palavra() {
                  this.tamanho = conteudo.length();
            }
            boolean isDelimiter(char c) {
                  for(int i=0;i<limites.length;i++) {
                        if(c==limites[i]) {
                              return true;
                        }
                  }
                  return false;
            }
      }
}
```

Na classe principal instanciamos ambas as classes e utilizamos a Inner Class “palavra”:

```
public class demotexto {
     public static void main(String args[]) {
          texto t = new texto();
          t.setConteudo(args[0]);
          texto.palavra tp = t.new palavra();
          String p1 = tp.next();
          while(!p1.equals("")) {
               System.out.println("\n" + p1);
               p1 = tp.next();
          }
     }
}
```

Podemos instanciar várias classes “palavra” e teremos vários ponteiros para lista de palavras.

As Inner classes podem acessar TODOS os membros da classe à qual pertencem, independentemente do modificador de acesso (private, protected etc). Elas também podem ser: “abstract”, “final”, “private”, “protected” ou default.

Static nested classes (Classes estáticas aninhadas):

Uma Inner class pode ser declarada “static”, ao contrário de uma classe normal. Se uma Inner class é declarada como “static”, passa a ser denominada apenas “static nested class”. O efeito é praticamente o mesmo de qualquer membro estático (exemplo “testestatic.java”):

```
public class testestatic {
      static class teste1 {
            public static String x = "Teste1";
            public String teste() {
                  return "Este é um teste";
            }
      }
      public static void main(String args[]) {
            System.out.println(teste1.x);
            // System.out.println(teste1.teste());// dará erro
            teste1 t1 = new teste1();
            System.out.println(t1.teste());
      }
}
```

Repare que:

1. Somente uma Inner Class pode ser declarada como “static”.
2. Sendo “static” ela não pode acessar membros não-estáticos da sua classe “mãe”. Somente se tiver uma referência a uma instância da classe-mãe.
3. Não precisamos instanciar a static nested classe se formos utilizar apenas os seus membros “static”. Se formos utilizar outros membros (“public”, “protected” etc) temos que instanciá-la.

4. Podemos instanciar uma static nested class utilizando apenas o nome da classe-mãe como referência: [nome-da-classe-mãe].new classe-filha().

Nested class inside methods (Classes dentro de métodos):

Podemos declarar Nested Classes dentro de métodos (qualquer bloco de código). Classes declaradas assim somente possuem uma restrição curiosa: somente podem acessar variáveis locais do método onde se encontram, se estas forem declaradas como final. Veja o exemplo “testeinsider.java”:

```
public class testeinsider {
      private String xpto = "Teste";
      public String yz = "Teste2";
      public void metodo1(String arg1) {
            final String v2 = "teste3";
            String v3 = "teste v3";
            class testeinterno {
                  public void metodointerno(String x) {
                        System.out.println("\nMetodo Interno " + x);
                  }
                  public void metodoacesso(String y) {
                        System.out.println("\nVariável final: " + v2);
                        // System.out.println("\nVariável não final: "
                              + v3);
                        System.out.println("\nVariável Da classe mãe: "
                              + xpto);
                        System.out.println("\nArgumento não final: " +
                              y);
                  }
            }
            testeinterno ti = new testeinterno();
            ti.metodointerno(arg1);
            ti.metodointerno(v2);
            ti.metodointerno(xpto);
            ti.metodointerno(v3);
            ti.metodoacesso("teste");
      }
      public static void main(String args[]) {
            testeinsider t = new testeinsider();
            t.metodo1("\nEste é um teste");
      }
}
```

Se o comentário for removido da linha onde ele tenta imprimir a variável “v3”, resultará em um erro de compilação porque a Inner class está tentando acessar uma variável declarada em seu método-pai que não é constante (“final”). O resto vale...

Anonymous classes (Classes Anônimas):

Uma classe pode ser declarada e instanciada sem possuir nome algum. Neste caso, o compilador irá gerar um nome específico baseado no nome da classe-mãe.

Classes anônimas normalmente são utilizadas em propriedades de AWT ou Swing, como no exemplo “testeanonimous.java”:

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

public class testeanonimous extends JFrame {
      JButton btn1 = new JButton("Start");
      void initComponents() {
            btn1.addActionListener(new ActionListener() {
                  public void actionPerformed(ActionEvent e) {
                        JOptionPane.showMessageDialog(null,  "OK",
                              "Teste",
                              JOptionPane.INFORMATION_MESSAGE);
                  }

            });
            Container cont = getContentPane();
            cont.setLayout(new FlowLayout());
            cont.add(btn1);
      }

      public static void main(String args[]) {
            testeanonimous ta = new testeanonimous();
            ta.setSize(200,200);
            ta.setLocationRelativeTo(null);
            ta.initComponents();
            ta.setVisible(true);
      }
}
```

Neste exemplo foi criada uma classe anônima que implementa a interface "ActionListener", definida no pacote Swing. Ao compilarmos o programa veremos que são criados dois arquivos "class": "testeanonimous.class" e "testeanonimous$1.class". A primeira é a classe principal e a segunda a classe anônima. Isto acontece com QUALQUER inner class, o que você pode comprovar compilando os programas de exemplo. Se for distribuir o aplicativo deve colocar TODAS as classes componentes juntas, o que pode ser feito em um arquivo JAR:

```
jar –cvf testeanonimous.jar testeanonimous*.class
```

Para executar a classe dentro de um JAR:

```
java -cp testeanonimous.jar testeanonimous
```

Nota: classes anônimas não podem ter construtores. Para isto use um Initializer.

Initializers (Inicializadores)

O Java define dois tipos de inicializadores: os Static Initializers e os Instance Initializers.

Normalmente podemos inicializar membros de uma classe apenas com atribuição:

```
public class teste1 {
     public String mNome = “Teste1”;
     public static String mTipo = “1”;
}
```

Neste exemplo ambas as variáveis “mNome” e “mTipo” serão incializadas, porém em momentos diferentes. As variáveis membro de instância são inicializadas imediatamente antes da execução do Construtor, e as variáveis Estáticas imediatamente após a carga da classe no JVM.

Porém e se quisermos inicializar uma variável-membro que seja um Objeto? Normalmente podemos fazê-lo no Construtor mas, para isto, teremos que instanciar a classe.

Vamos começar com os Static Initializers, que são mais fáceis de assimilar. Suponha um pequeno programa que abre uma conexão com um Banco de Dados e a atribui como membro de uma classe Estática (padrão J2EE DAO). No exemplo “testestaticinit.java” criamos uma classe “bdados” para armazenar a conexão com o Banco MySQL (você provavelmente não vai conseguir rodar o programa):

```
class bdados {
      static Connection conec = null;
      static {
            try {
                  Class.forName("com.mysql.jdbc.Driver");
                  conec = DriverManager.getConnection(
                  "jdbc:mysql://localhost/test?user=root&password=");
            }
            catch (ClassNotFoundException nf) {
                  System.out.println("Driver: " + nf.getMessage());
            }
            catch (SQLException e) {
                  System.out.println("SQL Exc.: " + e.getMessage());
            }
      } // fim do static initializer
}
```

Este bloco “static” vai ser executado quando da carga da classe no JVM, tornando a variável estática “conec” disponível para uso pelas outras classes do programa, como na seqüência:

```
public class testestaticinit {
      public static void main(String args[]) {
            try {
                  Statement stmt = bdados.conec.createStatement();
                  ResultSet rs = stmt.executeQuery(
                        "SELECT * FROM Task");
                  while(rs.next()) {
                        System.out.println("\n"+
                              rs.getString("TaskNo")+
                              ", "+
                              rs.getString("TaskDesc"));
                  }
                  rs.close();
            }
            catch (SQLException e) {
                  System.out.println("SQL Exc.: " + e.getMessage());
            }
            catch (Exception e) {
                  System.out.println("Exception : " + e.getMessage());
            }
            finally {
                  try {
                        if(bdados.conec != null) {
                              bdados.conec.close();
                        }
                  }
                  catch (Exception e) {
                        System.out.println(e.getMessage());
                  }
            }
      }
}
```

Quando a classe testestaticinit for executada (Método “main”) a classe “bdados” já terá sido inicializada, tornando a variável estática “conec” disponível para uso.

Instance Initializers

Os Instance Initializers funcionam como os Static Initializers, só que sem o modificador “static”. Só há um único bom uso para eles: simular “construtores” para classes anônimas.

Todo o código colocado em um Instance Initializer (sem o modificador “static”) é rodado pelo Construtor da Classe imediatamente após a execução do construtor da classe ancestral, na ordem em que são encontrados. No momento da execução do Construtor, todos os Instance Initializers já foram executados.

Veja o exemplo “testeinit.java”:

```
public class testeinit {
      public String mVar;

      {
            mVar = "abc";
      }

      public testeinit() {
            System.out.println("\nmVar=" + mVar +
                  ", mVar2=" + mVar2);
      }
      private String mVar2;
      public int z;

      {
            mVar2 = "xpto";
      }

      public static void main(String args[]) {
            testeinit t = new testeinit();
      }
}
```

Repare que ao rodar este programa o construtor já exibirá as variáveis “mVar” e “mVar2” com o conteúdo fornecido pelos dois inicializadores (em negrito).

Certamente é melhor colocar esse código todo dentro do Construtor, pois fica muito mais documentacional. Mas um bom uso para Instance Initializers é como substituto do Construtor em classes anônimas:

```
btn1.addActionListener(new ActionListener() {
      {
            // Código de inicialização
      }
      public void actionPerformed(ActionEvent e) {
            JOptionPane.showMessageDialog(null,  "OK",
                  "Teste", JOptionPane.INFORMATION_MESSAGE);
      }

});
```

Questões e exercícios

1) Analise a classe abaixo:

```
class cliente extends pessoa {
     public String nomeComercial;
     cliente() {
     }
}
```

O que é verdade sobre ela?

a) Implementa uma interface chamada “pessoa”
b) Implementa encapsulamento porque utiliza herança
c) Pode ser instanciada por qualquer outra classe, já que possui um construtor

2) Considere o seguinte código em Java:

```
Arquivo: exobj01.java
-------------------------
1> class pessoa {
2>    protected void contatar(String nome) {
3>          System.out.println("pessoa");
4>    }
5> }
6> class cliente extends pessoa {
7>    public boolean contatar(String x) {
8>          System.out.println("cliente");
9>          return true;
10>   }
11> }
12> public class exobj01 {
13>   public static void main(String args[]) {
14>         cliente c = new cliente();
15>         c.contatar("José");
16>   }
17> }
```

a) Compilará e rodará sem problemas
b) Dará erro de compilação na linha 14
c) Dará exception na linha 15
d) Dará erro de compilação na linha 7
e) Nenhuma das anteriores

3) Considere o seguinte código-fonte em Java:

```
Arquivo: exobj02.java
-------------------------
1> interface pessoa {
2>    void contatar(String nome);
3>    int dependentes();
4> }
5> class cliente implements pessoa {
6>    public void contatar(String x) {
7>          System.out.println("cliente");
8>          return;
9>    }
10>   int dependentes(int x) {
11>   }
12> }
13> public class exobj02 {
14>   public static void main(String args[]) {
15>         cliente c = new cliente();
16>         c.contatar("José");
17>   }
18> }
```

a) Dará exception na linha 15
b) Dará erro de compilação na linha 5
c) Dará erro na linha 15
d) Compilará e Rodará sem problemas

4) Considere o seguinte programa em Java:

```
Arquivo: exobj03.java
-------------------------
1> class cliente  {
2>    cliente(int z) {
3>          System.out.println(z);
4>    }
5>    private cliente(String x) {
6>    }
7>    void contatar() {
8>    }
9>    void cliente() {
10>   }
11> }
12> public class exobj03 {
13>   public static void main(String args[]) {
14>         cliente c = new cliente();
15>   }
16> }
```

a) Compilará e rodará sem problemas
b) Dará erro de compilação na linha 14
c) Dará erro de compilação nas linhas 2, 5 e 9
d) Dará exception na linha 14
e) Nenhuma das anteriores

5) Considere o seguinte construtor:

```
cliente() {
     this.nome = “*”;
     super();
}
```

Está correto? Caso contrário, diga o motivo.

6) Quando o método “finalize” será chamado:

```
class tb {
     public int xpto() {
     }
     protected void finalize(int x) {
     }
}
```

a) No momento que usarmos o comando “delete” para remover a instância da classe
b) No momento em que rodarmos o programa GC
c) No momento em que a memória utilizada pela classe estiver para ser liberada
d) Nenhuma das respostas anteriores

7) Considere o seguinte trecho de código em Java:

```
class teste {
     public xpto() {
          String a = “C”;
          String c = “xpto”;
          for(int i=0; i< c.length(); i++) {
               if(c.charAt(i) == “t”) {
                    a = null;
               }
          }
     }
}
```

Quando “a” será disponibilizada para o GC ?

Respostas

1) nenhuma das afirmações é verdadeira.

- Ela não implementa interface alguma e “pessoa” é a ancestral dela.
- Encapsulamento é o fato de se ocultar a implementação de um método ou variável. Ela não está bem encapsulada porque a propriedade “nomeComercial” é apenas uma variável pública. Herança é outro conceito e nada tem a haver com a afirmação.
- Ela somente pode ser instanciada a partir de classes no mesmo package, já que o modificador de acesso do Construtor é default (sem especificar nada).

2) letra “d”. Dará erro de compilação na linha 7 porque estamos tentando sobrescrever (override) o método “contatar”, só que informando o tipo de retorno diferente. Para sobrescrever temos que utilizar o mesmo tipo de retorno e argumentos.

3) letra “b”. Dará erro de compilação na linha 5 porque a classe “cliente” implementa a interface “pessoa” e não definiu TODOS os seus métodos. O método original “dependentes” em “pessoa” não possui argumentos.

4) letra “b”. Dará erro de compilação na linha 14 porque, ao tentar instanciar “cliente” estamos chamando o construtor “no-args”, que a classe não possui. Veja que ela possui um método “void cliente()”. Construtores NÃO podem ter valor de retorno. Ela só possui um construtor e este recebe um “int” como argumento.

Quando criamos um construtor, o compilador NÃO cria o construtor “no-args”.

5) Não está correto. Se formos chamar o construtor da classe ancestral ou outro construtor da mesma classe, temos que fazê-lo na primeira linha do construtor.

6) letra “d”. Ele só será chamado se nós o invocarmos. A classe “Object” possui um método “finalize” cuja assinatura é “void finalize()”, logo, este método não o sobrescreve. Só para constar:

- Não existe o comando “delete”
- Nós não rodamos o GC. Podemos sugerir que ele recolha memória através de “System.gc()”
- Se a assinatura do “finalize” estivesse igual à da classe Object, a letra “c” seria a resposta correta

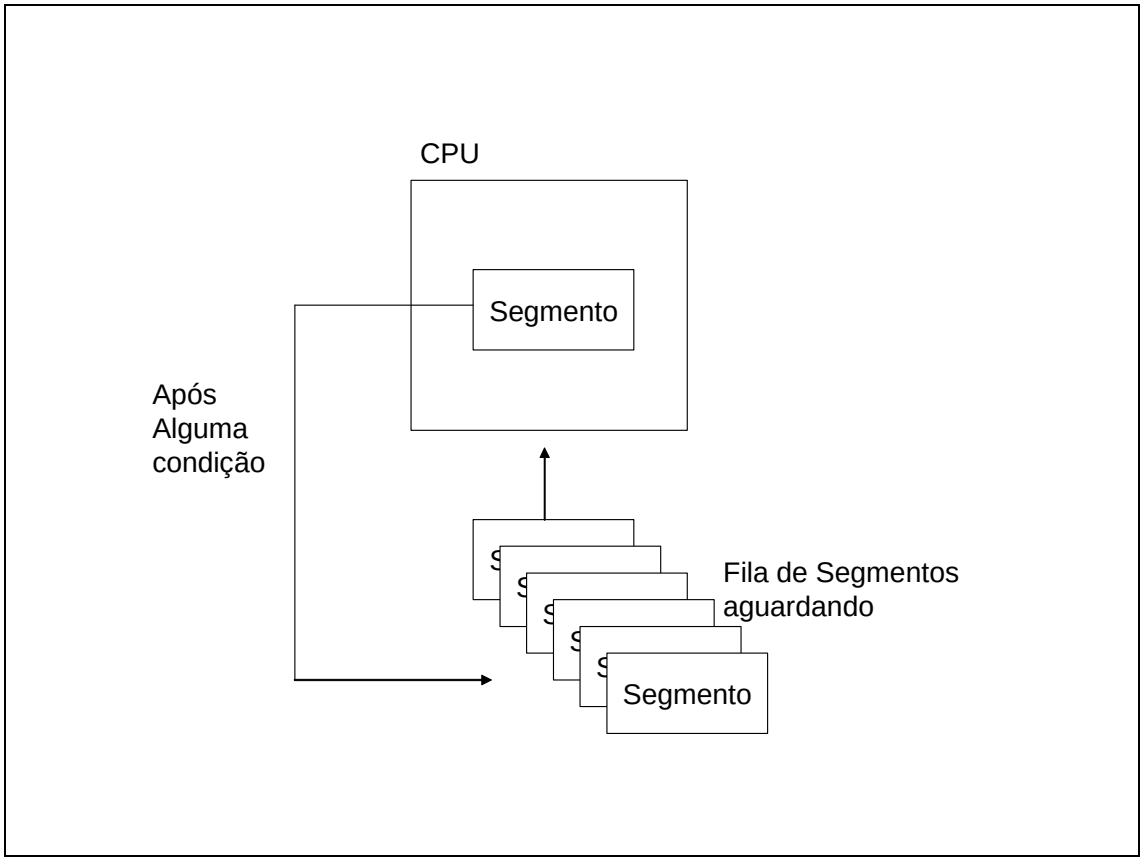
7) Após a execução de “a = null”. Os objetos nulos tornam-se disponíveis para o GC.

Segmentos (Threads)

Exemplos em “exemplos/07”.

Um segmento ou Thread é uma unidade de execução do Java. O que significa isto? Significa que todo programa Java tem pelo menos um segmento, o qual recebe tempo de execução do JVM.

Uma unidade de execução recebe o “direito” de executar determinado código e é posta em suspensão, dependendo de certas condições. Veja a figura seguinte:



Doravante vamos chamar “Segmento” de “Thread” pois é o nome mais comum. Utilizamos “Segmento” apenas para dar uma idéia melhor do que um Thread representa: um Segmento de um programa.

Na figura anterior temos o conceito básico de Thread explicado. Pode existir uma fila de Threads aguardando a oportunidade de entrar em execução na CPU, porém apenas um pode estar neste estado. Quando alguma condição o força a deixar a CPU, ele entra novamente na fila de Threads aguardando oportunidade.

Condições para um Thread deixar a CPU

Voluntariamente	Um Thread pode optar por deixar a CPU voluntariamente, dando assim oportunidade para outros Threads executarem. Isto pode ser feito com o método “yield” da classe “Thread”. Por que um Thread faria isto? Pode ser que ele esteja executando uma tarefa muito longa e deva, de tempos em tempos, oferecer oportunidade para outros Threads, caso contrário o programa “congela” e não deixa o usuário fazer mais nada.
Aguardando I/O	Um Thread necessita fazer alguma operação de I/O e, por isto, entra em estado de espera.
Aguardando Lock	Um Thread necessita obter acesso exclusivo a um Objeto e aguarda que o mesmo seja liberado.
Preempção	Dependendo da plataforma, o Sistema Operacional pode preemptar ou interromper a execução do Thread ao final do seu “time slice” (fatia de tempo destinada a cada Thread).
Prioridade	Um Thread de maior prioridade entra na fila da CPU. O de menor prioridade deve ser removido da execução e aguardar novamente na fila.
Término	O Thread terminou a sua tarefa.

Um programa multi-thread oferece como principal benefício o fato de que pode estar, aparentemente, executando mais de uma função simultaneamente. Imagine um programa que necessita formatar um relatório imenso e que demora muito tempo para ser preparado. Neste caso, usando um só Thread, o programa irá “congelar” até que a tarefa esteja pronta, sem dar qualquer informação ou oportunidade para o usuário.

Utilizando múltiplos Threads podemos distribuir as tarefas dedicando um Thread à Interface com o usuário, outro para formatar relatórios e ainda outro para receber comunicações via Sockets.

A classe Thread

Dentro do pacote java.lang temos a classe Thread que podemos utilizar para disparar segmentos de processamento separados dentro de um programa Java.

O arquivo “thread01.java” é um bom exemplo de múltiplos threads:

```
Arquivo: thread01.java
-------------------------
1> import java.io.*;
2>
3> public class thread01 extends Thread {
4>    thread01(String nome) {
5>          super(nome);
6>    }
7>    public void run()  {
8>          int tempo = (int) Math.round(Math.random() * 10);
9>          System.out.println("\nThread: " + this.getName() +
10>                   ", tempo: " + tempo);
11>         for(int i=0; i < 6; i++) {
12>               System.out.println("\nThread: " + this.getName());
13>               try {
14>                     sleep(tempo * 1000);
15>               }
16>               catch (InterruptedException ie) {
17>                     System.out.println("\nThread: " +
                            this.getName() +
18>                           ", Interrompido");
19>               }
20>         }
21>   }
22>   public static void main(String args[]) throws IOException {
23>         int nthreads = 0;
24>         BufferedReader console =
25>               new BufferedReader(new InputStreamReader(System.in));
26>         System.out.println("\nInforme o número de Threads: ");
27>         nthreads = Integer.parseInt(console.readLine());
28>         thread01 arrThreads[] = new thread01[nthreads];
29>         for(int i=0; i< arrThreads.length; i++) {
30>               arrThreads[i] = new thread01("T" + i);
31>               arrThreads[i].start();
32>         }
33>   }
34> }
```

Parece confuso, não? Mas não é.

Este programa cria uma classe descendente de Thread (thread01). Em seu método estático “main” ela cria um array com várias instâncias dela mesma e dispara a execução simultânea de cada uma.

O que cada Thread vai fazer é o que está no método “run”. Neste caso cada um vai imprimir uma mensagem (5 vezes), com um tempo aleatório entre elas.

Ao rodar o programa você verá que a execução é simultânea e que os Threads vão imprimindo conforme o seu próprio tempo aleatório.

O programa lê do console (System.in), através das classes “BufferedReader” e “InputStreamReader”, o número de Threads desejados e cria um array com este número de elementos. Logo a seguir ele instancia cada elemento e invoca o método “start” (cada elemento é descendente de Thread e possui um método “start”).

Para criarmos vários Threads podemos criar subclasses de Thread e sobrescrever o método “run” delas. Depois é só instanciar e invocar “start” (e não “run”) para que o Thread se torne elegível para execução. Veja um exemplo mais simples:

```
class meuThread extends Thread {
     public void run() {
          System.out.println(“OK”);
     }
}

public class xpto {
     public static void main(String args[]) {
          meuThread mt = new meuThread();
          mt.start();
     }
}
```

Podemos criar Threads com nome ou sem nome, bastando alterar o construtor.

Eis alguns métodos interessantes da classe Thread:

String getName()	Retorna o nome do Thread.
int getPriority() void setPriority(int newPriority)	Retorna a prioridade (veremos adiante) do Thread. Para alterar usamos o “setPriority”.
static void sleep(long millis)	Faz com que o Thread “durma” e deixe a CPU por pelo menos o número de milissegundos informados. Ao acordar ele entra na fila para conseguir a CPU novamente. Pode dar “InterruptedExeption” caso o Thread seja interrompido neste estado.
void start()	Torna o Thread “runnable” ou elegível para execução. Ele aguardará a oportunidade para voltar a ocupar a CPU.
static void yield()	Faz com que o Thread corrente (aquele que chamou o método) deixe a CPU momentaneamente. Ele volta à fila de espera pela CPU.

Implementando Runnable

Criar descendentes da classe Thread nos traz um problema relativo a modelagem de Objetos. Uma classe só pode ter uma ancestral, logo, será apenas filha de Thread. Se quisermos adicionar outros comportamentos não compatíveis com Thread, estamos ferindo o modelo de Objetos. A melhor regra é a seguinte: se a sua classe precisa ser filha de outra classe, então você não pode usar Thread como ancestral.

Como exemplo disto, se você quiser criar uma Applet, terá que fazer sua classe ser descendente da classe “Applet” ou “JApplet". Como poderia ser também filha de “Thread”?

Para evitar isto a classe Thread possui os seguintes construtores:

Thread(Runnable target)	Cria um novo Thread utilizando a classe intormada em “target”
Thread(Runnable target, String name)	Cria um novo Thread utilizando a classe intormada em “target” com o nome especificado
Thread(ThreadGroup group, Runnable target)	Cria um novo Thread utilizando a classe intormada em “target” dentro do Grupo de Threads especificado
Thread(ThreadGroup group, Runnable target, String name)	Cria um novo Thread utilizando a classe intormada em “target” dentro do Grupo de Threads especificado, com o nome especificado
Thread(ThreadGroup group, Runnable target, String name, long stackSize)	Cria um novo Thread utilizando a classe intormada em “target” dentro do Grupo de Threads especificado, com o nome especificado, informando também o tamanho da Pilha a ser utilizado

Logo podemos criar um Thread que usa uma classe que implementa a interface “Runnable” para executar seus comandos.

Com isto sua classe não precisa ser descendente de Thread, bastanto implementar a interface Runnable. Esta interface possui apenas o método: “void run()”. Logo, sua classe deve definir o método “run” com o que deseja que o novo Thread execute.

Veja o arquito “thread02.java” que é igual ao “thread01.java” só que utilizando a interface “Runnable”:

```
Arquivo: thread02.java
-------------------------
1> import java.io.*;
2>
3> class segmento implements Runnable {
4>    public void run()  {
5>          Thread t = Thread.currentThread();
6>          int tempo = (int) Math.round(Math.random() * 10);
7>          System.out.println("\nThread: " + t.getName() +
8>                       ", tempo: " + tempo);
9>          for(int i=0; i < 6; i++) {
10>               System.out.println("\nThread: " +
11>                         t.getName());
12>               try {
13>                         Thread.sleep(tempo * 1000);
14>               }
15>               catch (InterruptedException ie) {
16>                         System.out.println("\nThread: " +
17>                                   t.getName() +
18>                                   ", Interrompido");
19>               }
20>         }
21>   }
```

```
22> }
23>
24> public class thread02 {
25>   public static void main(String args[]) throws IOException {
26>         int nthreads = 0;
27>         BufferedReader console =
28>               new BufferedReader(new InputStreamReader(System.in));
29>         System.out.println("\nInforme o número de Threads: ");
30>         nthreads = Integer.parseInt(console.readLine());
31>         Thread arrThreads[] = new Thread[nthreads];
32>         segmento s1 = new segmento();
33>         for(int i=0; i< arrThreads.length; i++) {
34>               arrThreads[i] = new Thread(s1,"T" + i);
35>               arrThreads[i].start();
36>         }
37>   }
38> }
```

A classe “thread02” não é descendente de Thread. Note que na linha 32 ela cria uma instância da classe “segmento”, que por sua vez implementa “Runnable”. O array agora é todo de elementos “Thread”

Movemos o método “run” para a classe “segmento”. Como ela não é descendente de Thread, tivemos que obter uma referência para o Thread que está atualmente executando o método “run” e usá-la para obter o seu nome. Note que tivemos que colocar a referência à classe Thread no método “sleep”.

Eis um outro exemplo mais simples usando a interface “Runnable”:

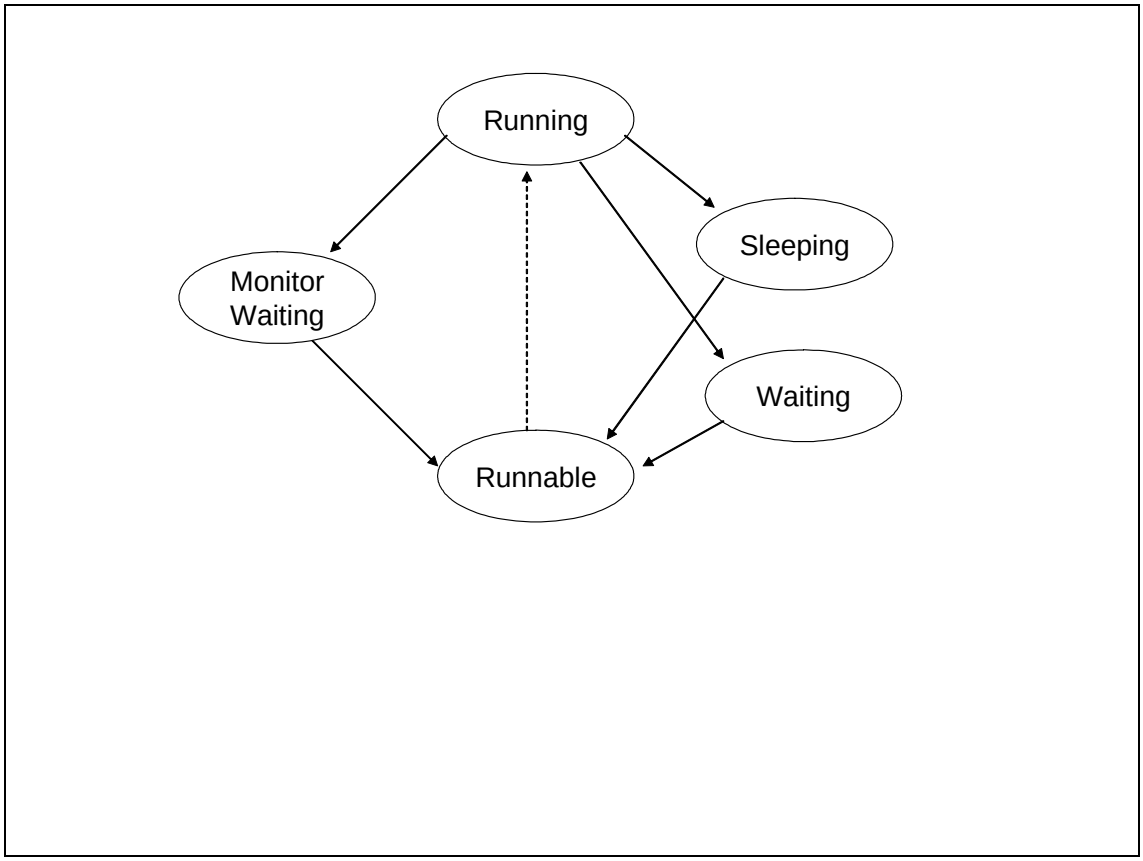
```
class meuThread implements Runnable {
     public void run() {
          System.out.println("OK");
     }
}

public class thread03 {
     public static void main(String args[]) {
          meuThread mt = new meuThread();
          new Thread(mt).start();
     }
}
```

Note que chamamos o método “start” diretamente depois do instanciamento do novo Objeto Thread. Em C++ isso daria um tremendo “memory leak”...

Estados de um Thread

Um Thread pode estar em vários estados, conforme a próxima figura.



Um Thread pode estar em qualquer um destes estados. O “monitor waiting” é uma situação que discutiremos mais adiante, mas os outros são fáceis de entender:

Running	O Thread está feliz! Está executando e tem o controle da CPU.
Sleeping	Você utilizou o método “sleep” para suspendê-lo.
Waiting	O Thread está esperando algum recurso.
Runnable	O Thread está pronto para ser executado novamente, aguardando apenas a vez na fila para a CPU.

Prioridade de Threads

O Java runtime suporta um mecanismo chamado de “Thread Scheduling” para determinar qual Thread, em estado “Runnable”, vai ocupar a CPU.

A classe Thread possui as constantes (static vars):

- MIN_PRIORITY
- MAX_PRIORITY
- NORM_PRIORITY

Para indicar a prioridade do Thread para efeitos de ocupação de CPU.

Ao criarmos um Thread ele irá herdar a prioridade do Thread que o criou. Se ele for o primeiro, irá utilizar a prioridade definida pela constante Thread.NORM_PRIORITY. Veja um exemplo:

```
class meuThread implements Runnable {
      public void run() {
            System.out.println("OK, prioridade: " +
                  Thread.currentThread().getPriority() +
                  ", NORM_PRIORITY: " + Thread.NORM_PRIORITY);
      }
}

public class thread03 {
      public static void main(String args[]) {
            meuThread mt = new meuThread();
            new Thread(mt).start();
      }
}
```

Ao rodar este programa podemos ver que o valor da prioridade do Thread é “5”, que corresponde a Thread.NORM_PRIORITY.

Um Thread de mais alta prioridade será sempre escolhido para ocupar a CPU em detrimento de Threads de mais baixa prioridade. Threads de mais baixa prioridade somente serão executados se:

- O Thread terminar
- O Thread executar “yield()”
- O Thread executar “sleep()”
- O Thread entrar em “wait” por alguma razão

Para evitar o fenômeno de “starvation” onde um Thread de mais alta prioridade ocupa a CPU por muito tempo, o Java runtime pode selecionar, ocasionalmente, um Thread de mais baixa prioridade para execução, preemptando o Thread de mais alta prioridade.

Podemos alterar a prioridade de um Thread com o método “setPriority”.

Espera pelo Monitor e sincronização (Monitor Waiting and Sinchronized)

Toda Classe e toda instância (Objeto) possui um mecanismo de “Lock”, que garante acesso a um e somente um Thread simultaneamente. Os Threads que desejam acessar o Objeto têm que primeiramente aguardar em uma fila para obter o acesso exclusivo. Todo Objeto que mantém uma fila de Threads é chamado de Monitor em Java.

As Classes também podem ser Monitores, desde que o método seja estático (veremos adiante).

Os recursos que um Objeto possui podem ser acessados por vários Threads simultaneamente. Todos os Objetos são assim e isto, geralmente, não representa problema algum, desde que o acesso seja apenas para leitura.

Se um Objeto possui uma propriedade (uma variável pública ou um método “set”) que pode ser alterada, o problema começa a aparecer. Imagine um Thread gravando na propriedade e outro Thread lendo a mesma propriedade! O resultado pode ser uma bagunça! Isto pode provocar o fenômeno de **Race Condition**.

Race Condition

Race condition é uma situação indesejável que ocorre quando um sistema tenta executar duas tarefas simultaneamente, para as quais a ordem de execução é crucial. Por exemplo, dois Threads tentam ler e gravar posições de memória simultaneamente. O resultado depende da ordem de execução das operações e pode resultar em uma exception ou em um dado corrompido.

Sincronização

Um Objeto pode utilizar um recurso do Java para evitar que mais de um Thread execute determinada operação ou acesse o conteúdo de uma propriedade (para alterar) simultaneamente.

Fazer um programa que deliberadamente provoque problemas de sincronização não é tarefa fácil... em princípio estes problemas somente podem ocorrer se um Thread for forçado a deixar a CPU antes de terminar de escrever ou se o computador é multi-processado (mais de uma CPU), neste caso mais de um Thread pode estar sendo executado.

Para simular os problemas em um ambiente multiprocessado, o programa “no_sincro.java” cria uma situação onde ele grava parte do dado, dorme por um período indeterminado, e depois volta a gravar a parte final. Assim você pode ver as conseqüências da falta de sincronização.

Veja um exemplo (arquivo “no_sincro.java”):

```
class altera implements Runnable {
      no_sincro master;
      public void run() {
            try {
                  while(true) {
                        Thread t = Thread.currentThread();
                        master.dado += "" +
            (Integer.parseInt(t.getName()) * 11111);
                        Thread.sleep((int) (Math.random() * 100));
                        master.dado += "" +
            (Integer.parseInt(t.getName()) * 11111);
                  }
            }
            catch (InterruptedException i) {
            }
      }
}
public class no_sincro {
      String dado;
      public static void main(String args[]) {
            no_sincro s = new no_sincro();
            Thread t[] = new Thread[5];
            altera a = new altera();
            a.master = s;
            for(int i=0; i<5; i++) {
                  t[i] = new Thread(a,"" + (i+1));
                  t[i].start();
            }
            try {
                  while(true) {
                        System.out.println(s.dado);
                        s.dado = "";
                        Thread.sleep(50);
                  }
            }
            catch (InterruptedException i) {
            }
      }
}
```

Como podemos ver, o método “run” da classe “altera” altera a propriedade “dado” da classe “no_sincro”, enquanto o seu método “main” está também lendo e alterando a mesma propriedade. Para simular um ambiente multiprocessado, colocamos o Thread que executa o “run” pra dormir entre uma alteração e outra.

Ao executarmos o programa, o resultado será uma verdadeira bagunça, com vários conteúdos misturados dentro da propriedade “dado”. Isto é o resultado de uma Race Condition.

O Java possui duas maneiras de impedir o acesso simultâneo a um recurso, uma delas é usar o modificador “synchronized” em um método, como no exemplo “sincro.java”:

```
class altera implements Runnable {
      sincro master;
      public void run() {
            try {
                  while(true) {
                        String conteudo = "";
                        Thread t = Thread.currentThread();
                        conteudo += "" + (Integer.parseInt(t.getName())
                              * 11111);
                        Thread.sleep((int) (Math.random() * 100));
                        conteudo += "" + (Integer.parseInt(t.getName())
                              * 11111);
                        master.alterarDado(conteudo);
                  }
            }
            catch (InterruptedException i) {
            }
      }
}
public class sincro {
      String dado;
      public synchronized void alterarDado(String valor) {
            this.dado = valor;
      }
      public static void main(String args[]) {
            sincro s = new sincro();
            Thread t[] = new Thread[5];
            altera a = new altera();
            a.master = s;
            for(int i=0; i<5; i++) {
                  t[i] = new Thread(a,"" + (i+1));
                  t[i].start();
            }
            try {
                  while(true) {
                        System.out.println(s.dado);
                        s.alterarDado("");
                        Thread.sleep(50);
                  }
            }
            catch (InterruptedException i) {
            }
      }
}
```

Agora existe um método especialmente criado para alterar a propriedade “dado” da classe “sincro”. Ele possui o modificador “synchronized”, o que significa que para poder executá-lo um Thread necessita adquirir o “lock” da instância de “sincro” que está rodando o método “run”.

Se um Thread não consegue obter o “lock” ele se junta à fila de Threads da instância de “sincro”, aguardando a sua vez de entrar no método “alterarDado”. Isto faz do Objeto que representa a instância de “sincro” um Monitor Java.

Quando um Thread, finalmente, consegue obter o “lock” ele volta ao estado “Runnable”, aguardando, agora, a sua vez de ocupar a CPU.

É importante notar que mesmo obtendo o “lock” do Monitor, o Thread não irá executar imediatamente! Ele tem que aguardar, em estado “Runnable”, por sua vez na CPU. Uma maneira de modificar isto é aumentar a prioridade do Thread, forçando qualquer Thread de menor prioridade a deixar a CPU.

Notificação de Threads

Quando um Thread entra na fila por recursos de um Monitor, ele aguarda a liberação do “lock” do Objeto.

Existem algumas situações onde podemos controlar QUANTO tempo o Thread deve aguardar antes de tentar obter o “lock” novamente. Igualmente, podemos avisar aos Threads que estão aguardando QUANDO deverão tentar obter o “lock” novamente.

Os seguintes métodos da classe “Object” implementam esse mecanismo de espera e notificação:

void notify()	“acorda” um Thread que esteja na fila do Monitor para que este procure obter o “lock” novamente. Se houver mais de um Thread na fila do Monitor, a escolha é arbitrária e dependente da implementação da JVM. Não podemos determinar QUAL Thread será “acordado”.
void notifyAll()	“acorda” TODOS os Threads que estejam na fila do Monitor. Eles procurarão obter o “lock” e apenas o que conseguir irá para o estado “Runnable”.
void wait()	Faz com que o Thread que invocou o “wait” aguarde na fila do Monitor até que o método “notify” (ou “notifyAll”) seja acionado.
void wait(long timeout) void wait(long timeout, int nanos)	Semelhante ao método “wait” só que ao expirar o tempo (timeout ou timeout e nanossegundos) ele tentará obter o “lock” novamente.

Onde e como estes métodos devem ser utilizados é uma questão mais complexa. Mas podemos pensar em um modelo “Producer-Consumer” ou “Produtor-Consumidor”, onde um Thread produz informação e outro(s) a consomem.

Neste caso o Produtor pode disponibilizar uma mensagem e esperar que um Consumidor a recolha.

Veja este primeiro exemplo do arquivo “monitor01.java”:

```
import java.util.*;
import java.text.DateFormat;

class msgserver implements Runnable {
      boolean temMensagem = false;
      String mensagem = "";
      public void run() {
            int tempo = 0;
            try {
                  while(true) {
                        // simula a recepção de uma msg
                        tempo = (int) Math.round(Math.random() * 10);
                        mensagem = getMensagem();
                        temMensagem = true;
                        Thread.sleep(tempo * 1000);
                  }
            }
            catch (InterruptedException ie) {
            }
      }
      private String getMensagem() {
            Date tNow = new Date();
            DateFormat tFormat =
DateFormat.getDateTimeInstance(DateFormat.LONG,
                              DateFormat.LONG,
                              Locale.getDefault());
            return tFormat.format(tNow);
      }
}

class msgclient implements Runnable {
      msgserver server;
      public void run() {
            try {
                  while(true) {
                        // a cada 10 segundos verifica mensagens
                        if(server.temMensagem) {
                              server.temMensagem = false;
                              System.out.println(server.mensagem);
                              server.mensagem = "";
                        }
                        Thread.sleep(10000);
                  }
            }
            catch (InterruptedException ie) {
            }
      }
}

public class monitor01 {
      public static void main(String args[]) {
            msgserver s1 = new msgserver();
            msgclient c = new msgclient();
            c.server = s1;
            new Thread(s1).start();
            new Thread(c).start();
      }
}
```

A classe “msgserver” simula um Servidor que recebe mensagens. Vamos supor que estas mensagens venham através de um Socket TCP aberto. Para simular a aleatoriedade da chegada das mensagens, ele aguarda um tempo (aleatório) antes de “receber” outra mensagem.

Quando chega uma mensagem ele liga um flag “temMensagem” e coloca o texto na propriedade “mensagem”.

A classe “msgclient” simula um Cliente que obtém as mensagens captadas pelo Servidor e as processa. Ele “dorme” por 10 segundos e então verifica o flag “temMensagem”, do Servidor. Se houver mensagem ele a imprime e limpa.

A primeira vista tudo parece legal (ou “bunitinho” - com “u” - como diria um amigo meu). Mas tem alguns “monstros” espreitando por baixo deste código. E, infelizmente, eles só surgirão quando o aplicativo estiver a pleno “vapor”.

Vamos analisar os problemas:

1. Não há proteção no acesso à variável pública “mensagem”. Logo, pode ocorrer uma situação onde os Threads “msgclient” estejam executando simultaneamente com o Thread “msgserver”.
2. Por não haver sincronismo entre o Servidor e o Cliente, pode ser que o Cliente “acorde” tarde demais e perca uma mensagem. Imagine se chegam duas mensagens com intervalos de milissegundos entre elas, como o Cliente só “acorda” a cada 10 segundos, ele, provavelmente, vai perder uma (ou várias) mensagem.
3. Não é o caso, mas se houver mais de um “msgclient” tentando ler mensagens teremos problemas como o do item “1”, só que entre os clientes.

Quando utilizamos o modificador “synchronized” estamos dizendo ao Thread que para executar o método é necessário adquirir o “lock” do Objeto onde ele está definido. Logo, podemos resolver alguns problemas criando métodos para armazenar novas mensagens e recuperar mensagens no próprio Servidor, utilizando o seu “lock” para controlar quando armazenar e quando recuperar.

O Servidor obtém o seu próprio “lock” para armazenar uma mensagem e o Cliente obtém o “lock” do Servidor para recuperar uma mensagem.

Existe também um truque importante sobre COMO determinar se há mensagem disponível. Mas vamos ver na nova versão do programa, arquivo “monitor02.java”:

Arquivo: monitor02.java

```
-------------------------
1> import java.util.*;
2> import java.text.DateFormat;
3>
4> class msgserver implements Runnable {
5>    boolean temMensagem = false;
6>    String mensagem = "";
7>    public void run() {
8>          int tempo = 0;
9>          try {
10>               while(true) {
11>                    // simula a recepção de uma msg
12>                    tempo = (int) Math.round(Math.random() * 10);
13>                    // sincroniza e armazena a mensagem recebida
14>                    armazena(getMensagem());
15>                    Thread.sleep(tempo * 1000);
16>               }
17>         }
18>         catch (InterruptedException ie) {
19>         }
20>   }
21>   private String getMensagem() {
22>         Date tNow = new Date();
23>         DateFormat tFormat =
DateFormat.getDateTimeInstance(DateFormat.LONG,
24>                                DateFormat.LONG,
25>                                Locale.getDefault());
26>         return tFormat.format(tNow);
27>   }
28>   public synchronized void armazena(String msg) {
29>         while(temMensagem == true) { // use "while" e não "if"!!!!!
30>                    try {
31>                    // Espera o cliente ler a mensagem que estava
lá
32>                        wait();
33>               } catch (InterruptedException e) {
34>                    }
35>         }
36>         // se saiu é porque não havia mensagem esperando
37>         mensagem = msg;
38>         temMensagem = true;
39>         // notifica os clientes que há mensagem para ser lida
40>         notifyAll();
41>   }
42>   public synchronized String recupera() {
43>         while (temMensagem == false) {// use "while" e não
"if"!!!!!
44>               try {
45>                    // Espera o Servidor disponibilizar mensagem
46>                    wait();
47>               }
48>               catch (InterruptedException e) {
49>               }
50>         }
51>         temMensagem = false;
52>         // Avisa ao Servidor que a mensagem foi lida
53>         notifyAll();
54>         return mensagem;
55>   }
56> }
```

```
57>
58> class msgclient implements Runnable {
59>   msgserver server;
60>   public void run() {
61>         while(true) {
62>               System.out.println(server.recupera());
63>         }
64>   }
65> }
66>
67> public class monitor02 {
68>   public static void main(String args[]) {
69>         msgserver s1 = new msgserver();
70>         msgclient c = new msgclient();
71>         c.server = s1;
72>         new Thread(s1).start();
73>         new Thread(c).start();
74>   }
75> }
```

Tanto o Cliente quanto o Servidor precisam sincronizar seus acessos à variável “mensagem”. Na linha “14” o Servidor invoca um método “synchronized” para armazenar a mensagem e ligar o flag “temMensagem”, logo, ele tentará obter o seu próprio “Lock” antes de fazer isto.

O Cliente invoca o método “recupera”, do Servidor na linha “62”. Como este método também é sincronizado, ele vai tentar obter o “lock” do Servidor. Isto impede que os Clientes e o Servidor entrem em Race Condition. Note que o Cliente não precisa mais esperar “10” segundos antes de verificar mensagens. Se não houver mensagens ele irá “dormir”.

Quando chega uma mensagem o Servidor tenta obter seu próprio “lock”. Conseguindo, ele irá verificar se o Cliente já leu a última mensagem. Isto é feito com o “while” da linha “29”. É importante usar “while” e não “if”. Se utilizássemos “if”, o bloco seria algo parecido com isto:

```
public synchronized void armazena(String msg) {
      if(temMensagem == true) { // use "while" e não "if"!!!!!
            try {
                  // Espera o cliente ler a mensagem que estava lá
                  wait();
            }
            catch (InterruptedException e) {
            }
      }
      // se saiu é porque não havia mensagem esperando
      mensagem = msg;
      temMensagem = true;
      // notifica os clientes que há mensagem para ser lida
      notifyAll();
}
```

Imagine que ao entrar no “if” o flag “temMensagem” esteja como “true”. Isto significa que o Cliente ainda não leu a última mensagem, logo, o Servidor não pode armazenar outra mensagem, por isto ele entra em “wait”. Ao sair do “wait” (através do “notifyAll”

que o Cliente faz no método “recupera”), ele irá armazenar a mensagem SEM TESTAR NOVAMENTE SE O CLIENTE JÁ LEU A QUE ESTAVA PENDENTE!

Por isto utilizamos o “while”, forçando o Servidor a verificar novamente se o Cliente já leu a última mensagem.

Ë claro que este código tem muitas fragilidades, como por exemplo o que fazer com a mensagem que chegou caso o Cliente ainda não tenha lido a anterior? Mas o objetivo não é implementar um serviço de mensagens, mas explicar os casos nos quais os métodos de “Object” (“wait”, “notify” e “notifyAll”) são úteis.

Pontos importantes:

- “wait”, “notify” e “notifyAll” são métodos da classe “Object” e não da classe “Thread”.
- Tanto em “notify” quanto em “notifyAll” você não pode determinar qual Thread vai ser “acordado”.

Sincronizando blocos de código

Assim como sincronizamos métodos podemos sincronizar apenas blocos de código, como no exemplo (“sincbloco.java”):

```
class altera implements Runnable {
      sincbloco  master;
      public void run() {
            try {
                  while(true) {
                        String conteudo = "";
                        Thread t = Thread.currentThread();
                        conteudo += "" + (Integer.parseInt(t.getName())
                              * 11111);
                        Thread.sleep((int) (Math.random() * 100));
                        conteudo += "" + (Integer.parseInt(t.getName())
                              * 11111);
                        synchronized(master) {
                              master.dado = conteudo;
                        }
                  }
            }
            catch (InterruptedException i) {
            }
      }
}
public class sincbloco {
      String dado;
      public static void main(String args[]) {
            sincbloco s = new sincbloco();
            Thread t[] = new Thread[5];
            altera a = new altera();
            a.master = s;
```

```
            for(int i=0; i<5; i++) {
                  t[i] = new Thread(a,"" + (i+1));
                  t[i].start();
            }
            try {
                  while(true) {
                        System.out.println(s.dado);
                        synchronized(s) {
                              s.dado = "";
                        }
                        Thread.sleep(50);
                  }
            }
            catch (InterruptedException i) {
            }
      }
}
```

Para sincronizar apenas um único bloco de código devemos informar qual é o Objeto que será o Monitor, para isto utilizamos a sintaxe:

```
synchronized(Objeto) {

}
```

O Thread que estiver executando este código vai ter que adquirir o "lock" do Objeto especificado. No caso do exemplo dado ("sincbloco.java") ambos tentam adquirir o "lock" da instância da classe "sincbloco". De nada adianta um bloco obter "lock" de um objeto diferente do que está sendo utilizado pelo outro bloco de código.

Grupos de Threads (ThreadGroup)

A classe ThreadGroup permite agrupar vários Threads em um único bloco, o que facilita a administração. Entre seus principais métodos estão:

int activeCount()	O número de Threads ativos neste grupo e em todos os grupos dos quais este é ancestral.
int getMaxPriority()	Obtém o valor da maior prioridade de Thread associado a este grupo.
void interrupt()	Interrompe TODOS os Threads deste grupo.

Podemos criar um ThreadGroup e depois, ao criarmos os Threads, informarmos a qual ThreadGroup eles pertencem:

```
Thread(ThreadGroup group, Runnable target)
```

Questões e Exercícios

1) Assinale as afirmativas verdadeiras:

a) Um Thread envia uma mensagem a um Objeto com o método “wait” e entra em sua fila de espera de “lock”.
b) Os estados de um Thread são: “Running”, “Waiting” e “Done”.
c) Os membros de um Objeto podem estar sendo acessados por Threads diferentes exatamente ao mesmo tempo.
d) O método “synchronize” da classe “Object” permite restringir acesso a recursos compartilhados.

2) Analise o seguinte código-fonte em Java:

```
Arquivo: exth01.java
-------------------------
1> public class exth01 extends Thread {
2>    public void run() {
3>          for(int i=0; i<101; i++) {
4>                System.out.println(i);
5>          }
6>    }
7>    public static void main(String args[]) {
8>          exth01 lista[] = new exth01[10];
9>          for(int i=0; i<lista.length; i++) {
10>               lista[i] = new exth01();
11>               lista[i].run();
12>         }
13>   }
14> }
```

Quantos Threads estarão em execução após a linha “12”?

3) Assinale a resposta correta:

a) O método “wait”, da classe Thread, o coloca em estado de espera na fila do Monitor.
b) O método “sleep” diz exatamente quanto tempo um Thread vai demorar para voltar a executar.
c) O método “notify” seleciona o Thread de maior prioridade para voltar a obter o “lock” do Monitor.
d) Podemos utilizar “sinchronized” em blocos de comando para utilizar “locks” de outros Objetos diferentes.

Respostas

1) A letra “c” é a única correta.

a) Incorreta. O método “wait” não envia mensagem alguma ao Objeto. Ele faz com que o Thread aguarde na fila até ser notificado pelo Objeto.
b) Incorreta. Os estados são: “Running”, “Sleeping”, “Waiting”, “Runnable” e “Monitor Waiting”.
c) Correta.
d) Incorreta. “synchronize” não é um método da classe Object. O modificador “synchronized” é que restringe acesso a recursos compartilhados.

2) Apenas um. Ao invocar o método “run” estamos apenas executando a sua função e não ativando os Threads. Eles não entrarão em execução simultânea. O correto seria chamar o método “start”.

3) A única resposta correta é a letra “d”.

a) Incorreta. O método “wait” é da classe “Object” e não da classe “Thread”.
b) Incorreta. O método “sleep” faz com que um Thread saia da CPU e “durma” por exatamente o tempo informado. Quando ele vai voltar a executar não pode ser determinado, pois ele voltará ao estado “Runnable”, aguardando sua vez na fila do processador.
c) Incorreta. O método “notify” não permite selecionar qual Thread será escolhido.

Pacotes Java.Lang e Java.Util

Exemplos em “exemplos/08”.

Os pacotes java.lang e java.util incluem diversas classes e interfaces úteis para a programação. No exame de Java Certified Programmer caem perguntas sobre:

- Classe Math
- Classes Wrapper (Integer, Float etc)
- Classes String e StringBuffer
- Collections API

Vamos começar pelo pacote java.lang, que é automaticamente importado em toda classe Java. *Todas as descrições de métodos foram traduzidas do **Java Tutorial**, da **SUN (www.java.sun.com)**.*

Classe Math

É uma classe “final”, significando que não pode ser estendida. Ela possui algumas constantes e vários métodos estáticos para que possamos utilizar sem criar instâncias. O seu construtor é “private”, o que significa que não é possível criar instâncias dela.

Propriedades

A classe Math possui apenas as constantes:

static double E	Valor aproximado da base do logarítmo neperiano (2,7183...).
static double PI	Valor aproximado de PI (3,14...).

Os seus principais métodos são (exemplo: “testemath.java”):

static double abs(double a)	Retorna o valor absoluto do argumento. Possui overloads para argumentos int, float e long.
static double ceil(double a)	Retorna o menor valor, maior que o argumento, porém igual a um inteiro. Por exemplo “Math.ceil(5.3)” retorna “6.0d”.
static double cos(double a)	Retorna o cosseno do ângulo, em radianos, informado como argumento.
static double floor(double a)	Retorna o maior valor, menor que o argumento, porém igual a um inteiro. Por

	exemplo “Math.floor(5.3)” retorna “5.0d”
static double max(double a, double b)	Retorna o maior valor dos dois argumentos. Existem overloads para int, float e long.
static double min(double a, double b)	Retorna o menor valor dos dois argumentos. Existem overloads para int, float e long.
static double pow(double a, double b)	Potenciação do primeiro argumento pelo Segundo.
static double random()	Retorna um número aleatório positivo entre 0.0 e 1.0.
static long round(double a)	Retorna o valor longo mais próximo do argumento. Exemplo: “Math.round(5.51)” retorna “6”. Possui um overload com argumento e retornos float.
static double sin(double a)	Retorna o seno do ângulo (em radianos) passado como argumento.
static double sqrt(double a)	Retorna a raiz quadrada do número passado como argumento.
static double tan(double a)	Retorna a tangente do ângulo (em radianos) passado como argumento.
static double toDegrees(double angrad)	Retorna o valor em graus do ângulo em radianos passado como argumento.
static double toRadians(double angdeg)	Retorna o valor em radianos do ângulo graus passado como argumento.

Classes Wrapper

As classes “wapper” ou “embrulho” servem para encapsular um valor primitivo (um int ou float, por exemplo) dentro de um Objeto imutável. Por exemplo:

```
Integer objInt = new Integer(5);
```

Além disto elas possuem vários métodos estáticos que podem ser utilizados para formatar ou converter elementos.

As classes Wrapper são:

Classe Wrapper	Primitivo que encapsula
Boolean	boolean
Byte	byte
Character	char
Double	double
Float	float
Integer	int
Long	long

Essas classes possuem dois construtores, sendo um com um argumento do tipo primitivo que elas encapsulam e outro contendo um String, que é a representação textual do primitivo. A única excessão é a classe Character, que só possui um construtor que receve um char.

Métodos para recuperar um valor (xxxValue)

Quase todas as classes Wapper possuem um método xxxValue para retornar o valor armazenado nelas como um primitivo. Temos os métodos:

- double doubleValue()
- float floatValue()
- int intValue()
- long longValue()

Quase todas possuem estes métodos, exceto a classe Character. Para obter um número a partir do conteúdo de uma instância de Character temos que usar o método: “char charValue()”.

Métodos parseXxx

Quase todas as classes Wrapper possuem métodos “parseXxx” que avaliam uma expressão String, dada como argumento, e retornam um primitivo correspondente ao seu tipo encapsulado. Exemplos (arquivo “wrapper.java”):

```
byte b = Byte.parseByte(“5”);
int x = Integer.parseInt(“10”,16);  // mudou para base 16
```

As excessões são as classes Boolean e Character, que não possuem método “parseXxx”.

Os métodos parseXxx normalmente levantam exceptions caso o String não represente um número válido. Por exemplo o método Integer.parseInt() levanta a exception NumberFormatException caso o string não represente um número válido.

Operações comuns

Exemplo: “wrapper.java”.

Obter um número à partir de um String:

```
converte(String numero) {
      try {
            int x = Integer.parseInt(numero);
            // ou
            Integer In = new Integer(numero);
            }
      catch (NumberFormatException nfe) {
            System.out.printl(“Numero invalido”);
      }
}
```

Convertendo números em String:

Podemos utilizar os métodos “toString” ou usar a classe “DecimalFormat”, do pacote java.text.*:

```
String is = i.toString();
String dbs = db.toString();
DecimalFormat df = new DecimalFormat("###,###.00");
String saida = df.format(db.doubleValue());
System.out.println("\ntoString em inteiro: " + i);
System.out.println("\ntoString em double: " + dbs);
System.out.println("\nFormat: " + saida);
```

Datas

Exemplo: “datas.java”.

Vamos ver rapidamente como lidar com datas, já que o exame nada menciona a respeito.

Para obter a data de hoje usamos as classes “Calendar” e “Date”, como no exemplo:

```
import java.util.Calendar;
import java.util.Date;
public class datas {
      public static void main(String args[]) {
            Calendar cal = Calendar.getInstance();
            Date datahoje = cal.getTime();
            int dia = cal.get(Calendar.DAY_OF_MONTH);
            int mes = cal.get(Calendar.MONTH);
            int ano = cal.get(Calendar.YEAR);

            System.out.println("\nData completa: " + datahoje);
            System.out.println("\nDia: " + dia + ", mes: " +
                        mes + ", ano: " + ano);
      }
}
```

A classe Vector

Esta é uma classe implementada na versão 1.2 do Java, que cria um array redimensionável de objetos. Ela foi refeita para implementar a interface List, da API Collections (veremos adiante), mas como é muito utilizada consideramos importante citá-la à parte.

Como ela implementa a interface List, oferece os métodos:

void	**add**(int index, Object element) Insere o argumento emelent na posição especificada pelo argumento index na lista.
boolean	**add**(Object o) Adiciona o argumento “o” ao final da lista.
boolean	**addAll**(Collection c) Adiciona todos os elementos da “Collection” “c” ao final da lista.
boolean	**addAll**(int index, Collection c) Insere todos os elementos da “Collection” “c” na posição especificada pelo argumento “index” no Vector atual.
void	**clear**() Remove todos os elementos.
boolean	**contains**(Object o) Returns true if this list contains the specified element.
boolean	**containsAll**(Collection c) Retorna “true” se o a lista contém todos os elementos da “Collection” “c”.
boolean	**equals**(Object o) Compara o Objeto “o” com esta lista para verificar igualdade.
Object	**get**(int index) Retorna o elemento cuja posição é indicada pelo argumento “index”.
int	**hashCode**() Retorna o hashCode da lista inteira.
int	**indexOf**(Object o) Retorna o índice da primeira ocorrência do objeto “o”. Caso não encontre, retorna -1.
boolean	**isEmpty**() Retorna “true” se a lista está vazia.
Iterator	**iterator**() Retorna um Objeto Iterator para podermos navegar na lista.
int	**lastIndexOf**(Object o) Retorna o índice da última ocorrência do objeto “o” nessa lista ou -1.

ListIterator	**listIterator**() Retorna um objeto listIterator (duplamente encadeada) para esta lista.
ListIterator	**listIterator**(int index) Retorna um objeto listIterator começando da posição informada em "index".
Object	**remove**(int index) Remove o elemento cujo índice foi informado em "index". Retora o objeto removido.
boolean	**remove**(Object o) Remove a primeira ocorrência do Objeto "o" da lista, retornando "true" em caso de sucesso.
boolean	**removeAll**(Collection c) Remove todos os elementos que também estão contidos na Collection "c".
boolean	**retainAll**(Collection c) Remove todos os elementos da lista que não estão contidos na Collection "c".
Object	**set**(int index, Object element) Subtitui o elemento na posição indicada por "index" pelo informado em "element". Retorna o elemento anterior.
int	**size**() Retorna o número de elementos na lista.
List	**subList**(int fromIndex, int toIndex) Retorna uma lista contendo dos elementos "fromIndex" (inclusive) até o "toIndex" (exclusive).
Object[]	**toArray**() Retorna um array de objetos de todos os elementos da lista.
Object[]	**toArray**(Object[] a) Faz o mesmo que o anterior mas com runtime type definido pelo array passado em "a".

Existem outros métodos da classe Vector que não interessam no momento.

Podemos criar um Vector com uma capacidade inicial (argumento "capacity" no construtor) e um fator de incremento (argumento "capacityincrement" no construtor). Quando a capacidade for atingida, serão alocados mais "capacityincrement" elementos.

Vamos ver o exemplo "mostravector.java":

```
import java.util.*;
public class mostravector {
      public static void main(String args[]) {
            Vector vetargs = new Vector(args.length);
            for(int i=0; i<args.length; i++) {
                  vetargs.add(args[i]);
            }
            System.out.println(vetargs.contains(args[0]));
            ListIterator li = vetargs.listIterator(vetargs.size());
            while(li.hasPrevious()) {
                  String s = (String) li.previous();
                  System.out.println("\n" + s);
            }
      }
}
```

Neste exemplo criamos um Vector contendo os elementos do array de argumentos informados (para rodar este exemplo você tem que informar os argumentos), depois verificamos se um deles está contido no Vector.

Ao final navegamos no Vector, em ordem inversa, utilizando o ListIterator (interface de java.util). Navegar com Iterators é um conceito introduzido pelo Standard Template Library em C++ que foi implementado em Java.

Para navegar com Iterator (interface de java.util) em Java basta utilizar os métodos “hasNext()” para testar se ainda existe um próximo elemento, e “next()” para obter o próximo elemento. A principal diferença entre Iterator e ListIterator é a capacidade de navegar ao inverso.

Eis a interface “Iterator”:

`boolean`	**`hasNext`**`()` Retorna “true” se existem mais elementos após o atual.
`Object`	**`next`**`()` Retorna o próximo elemento na lista.
`void`	**`remove`**`()` Remove o ultimo elemento retornado pelo Iterator (next).

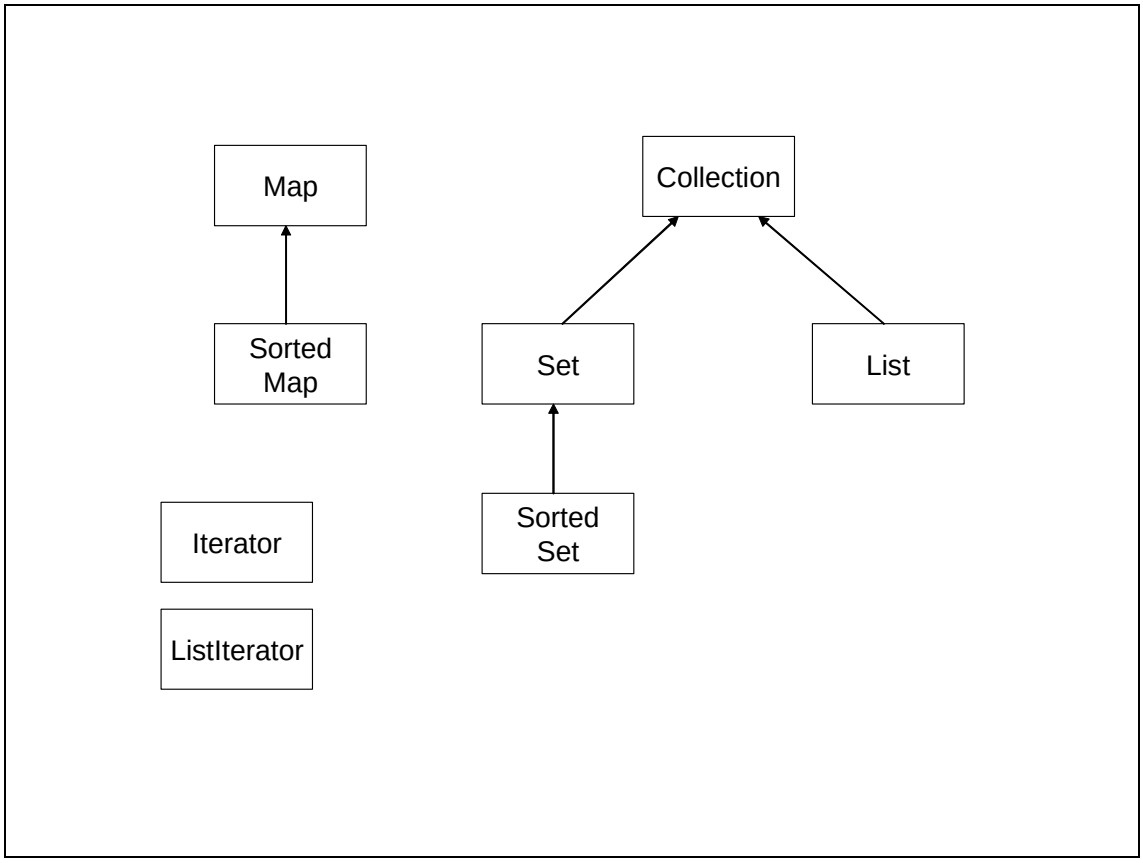
Eis a interface “ListIterator”:

`void`	**`add`**`(Object o)` Insere o objeto na lista à qual o ListIterator pertence.
`boolean`	**`hasNext`**`()` Retorna “true” se existem mais elementos além do atual, no sentido normal (do primeiro para o último).
`boolean`	**`hasPrevious`**`()` Retorna “true” se existirem elementos anteriores ao atual (no sentido reverse).
`Object`	**`next`**`()` Retorna o próximo elemento do ListIterator.
`int`	**`nextIndex`**`()` Retorna apenas o índice do próximo elemento a ser retornado pelo método “next()”.
`Object`	**`previous`**`()` Retorna o elemento anterior do ListIterator.
`int`	**`previousIndex`**`()` Retorna o índice do próximo elemento a ser retornado pelo método “previous()”.
`void`	**`remove`**`()` Remove da lista o elemento que foi retornado pelo “next()” ou “previous()”.
`void`	**`set`**`(Object o)` Substitui o ultimo elemento retornado por “next()” ou “previous()” pelo argumento “o”.

A API Collections

Dentro do pacote java.util temos as interfaces e classes que formam a API “Collections” do Java. Seu objetivo é lidar com coleções de objetos de maneira padronizada.

A estrutura da API (as vezes chamada de “framework”) é composta por várias interfaces, que podem ser visualizadas na figura seguinte:



Collection

Representa um grupo de elementos e descreve os membros comuns que toda coleção deve implementar. A classe AbstractCollection implementa a interface, porém é marcada como “abstract” para que você possa derivar outras classes dela. Eis os membros da Interface:

`boolean`	**`add`**`(Object o)` Adiciona o elemento representado pelo argumento “o” à coleção.
`boolean`	**`addAll`**`(Collection c)` Adiciona TODOS os elementos da coleção “c” a esta coleção.
`void`	**`clear`**`()` Remove todos os elementos desta coleção.
`boolean`	**`contains`**`(Object o)` Retorna “true” se esta coleção contém o elemento representado pelo argumento “o”.
`boolean`	**`containsAll`**`(Collection c)` Retorna “true” se esta coleção contém TODOS os elementos da coleção “c”.
`boolean`	**`equals`**`(Object o)` Compara a coleção representada pelo objeto “o” com esta coleção.
`int`	**`hashCode`**`()` Retorna o hash code desta coleção.
`boolean`	**`isEmpty`**`()`

	Retorna “true” se esta coleção está vazia.
`Iterator`	**`iterator`**`()` Retorna um objeto que implementa a interface “Iterator” para navegar nesta coleção.
`boolean`	**`remove`**`(Object o)` Remove o elemento representado pelo argumento “o” desta coleção.
`boolean`	**`removeAll`**`(Collection c)` Remove, desta coleção, TODOS os elementos da coleção “c”, que também estejam presentes nesta.
`boolean`	**`retainAll`**`(Collection c)` Elimina TODOS os elementos desta coleção, exceto aqueles que também estão presentes na coleção “c”.
`int`	**`size`**`()` Retorna o número de elementos desta coleção.
`Object[]`	**`toArray`**`()` Retorna um array contendo todos os elementos desta coleção.
`Object[]`	**`toArray`**`(Object[] a)` Retorna um array contendo todos os elementos desta coleção; o runtime type dos elementos do array será igual ao do array especificado.

Set

A interface Set estende a interface Collection, não declarando nenhum método a mais. A diferença é que um Set não pode conter elementos duplicados, é como se fosse um Conjunto matemático. Algumas implementações da interface Set não permitem o valor “null”, mas se ele só pode existir uma só vez no set inteiro.

Algumas implementações restringem o tipo de dados dos elementos e podem levantar as excessões: “ClassCastException” ou “NullPointerException”.

List

Uma List é uma seqüência ordenada de elementos, que podem estar duplicados. Como a ordem dos elementos é importante, existem vários métodos que utilizam o índice como argumento. A classe Vector é um exemplo de implementação da interface List (já descrita anteriormente).

Map

Um Map é uma lista de tuplas chave-valor (key-value) únicas, ou seja, não pode conter chaves duplicadas. Não há especificação de ordenação de chaves e os elementos de um mapa podem ser vistos como uma coleção de chaves, uma coleção de valores ou uma coleção de pares chave-valor. A interface Map contém uma sub-interface “Map.Entry” que permite lidar com cada par chave-valor.

Eis a interface Map:

void	**clear**() Remove todos os elementos (chave-valor) deste mapa.
boolean	**containsKey**(Object key) Retorna “true” se este mapa contém a chave especificada no argumento “key”.
boolean	**containsValue**(Object value) Retorna “true” se este mapa contém uma ou mais chaves que apontam para o valor especificado no argumento “value”.
Set	**entrySet**() Retorna um Set contendo elementos “map.entry” de cada elemento neste mapa.
boolean	**equals**(Object o) Compares the specified object with this map for equality.
Object	**get**(Object key) Retorna o valor apontado pelo argumento “key”.
int	**hashCode**() Retorna o hash code deste mapa inteiro.
boolean	**isEmpty**() Retorna “true” se este mapa está vazio.
Set	**keySet**() Retorna um Set contendo todas as Chaves neste mapa (só as chaves).
Object	**put**(Object key, Object value) Adiciona um par chave-valor a este mapa.
void	**putAll**(Map t) Copia todos os elementos (Chave-valor) do mapa especificado no argumento “t”.
Object	**remove**(Object key) Remove a dupla chave-valor correspondente a chave do argumento “key”.
int	**size**() Retorna o número de conjuntos chave-valor existentes neste mapa.
Collection	**values**() Retorna uma Collection contendo todos os valores (só os valores) deste mapa.

Sub-interface Map.Entry

boolean	**equals**(Object o) Compara este Map.Entry com o do argumento “o”.
Object	**getKey**() Retorna a chave deste Map.Entry.
Object	**getValue**() Retorna o valor deste Map.Entry.
int	**hashCode**() Retorna o hash code deste Map.Entry.
Object	**setValue**(Object value) Altera o valor deste Map.Entry.

equals, hashCode

Se criarmos classes que podem pertencer a coleções, devemos pensar seriamente em sobrescrever os métodos “equals” e “hashCode”, da classe “Object”. Classes especializadas como “Hashtable” necessitam que ambos estejam sobrescritos nos elementos que pretendemos armazenar.

Interface Comparable

Todos os objetos envolvidos em algum tipo de ordenação devem implementar a interface Comparable ou fornecer um Comparator. Esta interface determina a ordem natural dos elementos de uma classe. Se uma Coleção contém apenas Strings, esta ordem é alfabética, mas, se o elemento contém mais de um membro temos que dizer qual é a ordem natural de comparação entre eles.

Ela possui apenas o método: “int compareTo(Object o)”. Este método deve retornar o resultado da comparação do objeto atual com o especificado pelo argumento “o”. Este resultado é um valor “int” que deve ser:

- < 0 se o objeto é menor que o informado no argumento.
- > 0 se o objeto é maior que o informado no argumento.
- == 0 se o objeto é igual ao informado no argumento.

Interface Comparator

Em certas situações desejamos que o objeto de comparação seja externo à nossa classe. Neste caso podemos informar como argumento (para construtores de derivados de Collection) um objeto que implemente a interface Comparator. Note bem a diferença: Comparable deve ser implementada pela classe que forma os nossos elementos e Comparator é para uma classe diferente da dos nossos elementos.

Esta interface possui dois métodos:

int	**compare**(Object o1, Object o2) Compara dois objetos. (eles devem implementar a interface Comparable também. O retorno é o mesmo do "compareTo": <0 se o primeiro é menor que o segundo, >0 se for o contrário e ==0 se forem iguais.
boolean	**equals**(Object obj) Verifica se este comparator é igual a outro.

SortedSet

É uma interface que permite implementar um Set ordenado. Seus métodos são:

Comparator	**comparator**() Retorna o objeto comparator associado com este Set ou "null" se ele usa apenas a ordem natural (interface Comparable) dos elementos.
Object	**first**() Retorna o primeiro (Segundo a ordem de comparação) elemento do Set.
SortedSet	**headSet**(Object toElement) Retorna um SortedSet deste Set onde todos os elementos são menores que o argumento "toElement".
Object	**last**() Retorna o maior (último) elemento do Set.
SortedSet	**subSet**(Object fromElement, Object toElement) Retorna um SortedSet contendo dos elementos "fromElement" (inclusive) até "toElement" (exclusive).
SortedSet	**tailSet**(Object fromElement) Retorna um SortedSet contendo todos os elementos que são maiores ou iguais a "fromElement".

SortedMap

É uma interface que cria um Mapa classificado em ordem ascendente. Todos os elementos devem implementar a interface Comparable ou deve ser fornecido um Comparator na criação do SortedMap. Seus membros são:

Comparator	**comparator**() Retorna o Comparator utilizado na criação do mapa ou "null" se estiver utilizando a ordem natural das chaves do mapa.
Object	**firstKey**() Retorna a primeira (de acordo com a ordem) chave do mapa.
SortedMap	**headMap**(Object toKey) Retorna um SortedMal contendo todas os pares cuja chave seja menor que a informada no argumento "toKey".

Object	**lastKey**() Retorna a última chave (de acordo com a ordem) do mapa.
SortedMap	**subMap**(Object fromKey, Object toKey) Retorna um SortedMap contendo todas as chaves entre “fromKey” (inclusive) e “toKey” (exclusive).
SortedMap	**tailMap**(Object fromKey) Retorna um SortedMap contendo todas as chaves que são maiores ou iguais ao argumento “fromKey”.

Classes Abstratas que implementam as interfaces

- AbstractCollection. Para implementar uma classe funcional temos que sobrescrever os métodos “iterator”, “remove” e “size”.
- AbstractList. Para implementar uma classe funcional temos que sobrescrever os métodos “get”, “set”, “add” e “remove”.
- AbstractMap. Basta sobrescrever “entrySet”, “put” e o iterator deve implementar “remove”.
- AbstractSequentialList. Temos que sobrescrever “listIterator” e “size”. Além disto o list iterator deve ter seus métodos “hasNext”, “next”, “hasPrevious”, “previous” e “index” sobrescritos. Se desejarmos uma lista atualizável temos que sobrescrever os métodos do iterator: “set”, “add” e “remove”.
- AbstractSet. Apenas acrescenta os métodos “equals” e “hashCode” à classe AbstractCollection.

Classes concretas fornecidas na API Collections

ArrayList

Implementa uma lista de elementos através de um array redimensionável. Com esta estrutura, o método “add” pode demorar mais tempo se a capacidade for superada. Podemos redimensionar a lista ANTES de estourar a capacidade com o método “ensureCapacity”. Eis os Construtores:

ArrayList() Cria uma lista vazia com capacidade inicial de dez elementos.
ArrayList(Collection c) Cria uma lista contendo os elementos da coleção “c” na ordem dada pelo iterador desta última.
ArrayList(int initialCapacity) Constrói uma lista vazia com a capacidade inicial fornecida pelo argumento “initialCapacity”.

Eis os métodos (além dos de AbstractList):

Object	**clone**() Retorna uma cópia sem os elementos desta lista.
void	**ensureCapacity**(int minCapacity) Increases the capacity of this ArrayList instance, if necessary, to ensure that it can hold at least the number of elements specified by the minimum capacity argument.
void	**trimToSize**() Trunca a capacidade da lista para ficar igual ao tamanho atual da lista. (reduz consumo de memória).

Exemplo de ArrayList (ver arquivo “exArrayList.java”):

```
import java.util.*;

class cliente implements Comparable {
      private String nome;
      private String sobrenome;
      cliente(String sobrenome, String nome) {
            this.nome = nome;
            this.sobrenome = sobrenome;
      }
      String getNome() {
            return this.nome;
      }
      String getSobrenome() {
            return this.sobrenome;
      }
      public String toString() {
            return this.sobrenome + ", " + this.nome;
      }
      public int hashCode() {
            return this.sobrenome.hashCode() * 2 +
                  this.nome.hashCode() ;
      }
      public boolean equals(Object o) {
            cliente s = (cliente) o;
            return (this.nome.equals(s.getNome()) &&
                  this.sobrenome.equals( s.getSobrenome()));
      }
      public int compareTo(Object o) {
            cliente s = (cliente) o;
            return (2 * this.sobrenome.compareTo(s.getSobrenome()) +
                  this.nome.compareTo( s.getNome()));
      }

}
public class exArrayList {
      public static void main(String args[]) {
            cliente a = new cliente("silva", "maria");
            cliente b = new cliente("silva", "jose");
            cliente c = new cliente("pereira", "francisco");
            System.out.println("\nHashcodes a,b,c: " +
                  a.hashCode() + ", " +
                  b.hashCode() + ", " +
                  c.hashCode());
            System.out.println("\nEquals a(b): " +
                  a.equals(b));
            System.out.println("\ncompareTo a(b): " +
```

```
                  a.compareTo(b));
            System.out.println("\ncompareTo b(c): " +
                  b.compareTo(c));
            ArrayList al = new ArrayList(3);
            al.add(a);
            al.add(b);
            al.add(c);
            ListIterator li = al.listIterator();
            while(li.hasNext()) {
                  System.out.println("\nElemento: " +
                        (cliente)(li.next()));
            }

      }
}
```

A classe “cliente”, do exemplo anterior, implementa os métodos “hashCode”, “equals” e “compareTo”, de modo a ser utilizada em todos os exemplos seguintes.

Note que a ordem dos elementos será a ordem em que foram adicionados à ArrayList.

Classe LinkedList

Descende de AbstractSequentialList e implementa uma lista duplamente encadeada, que permite seu uso como pilha ou fila, por exemplo. Possui os métodos adicionais:

`void`	**`addFirst`**`(Object o)` Insere o elemento no começo da lista.
`void`	**`addLast`**`(Object o)` Adiciona o elemento no fim da lista.
`Object`	**`getFirst`**`()` Retorna o primeiro elemento na lista.
`Object`	**`getLast`**`()` Retorna o último elemento da lista.
`Object`	**`removeFirst`**`()` Remove o primeiro elemento da lista, retornando-o.
`Object`	**`removeLast`**`()` Remove o último elemento da lista, retornando-o.

Classe TreeMap

Implementa uma árvore sem duplicidade de folhas (interface SortedMap), em ordem natural (“compareTo”) ou através de um Comparator. É uma boa escolha para listas que tem que ser ordenadas.

Eis o mesmo exemplo anterior usando TreeMap:

```
import java.util.*;

class cliente implements Comparable {
     private String nome;
     private String sobrenome;
     cliente(String sobrenome, String nome) {
          this.nome = nome;
          this.sobrenome = sobrenome;
     }
     String getNome() {
          return this.nome;
     }
     String getSobrenome() {
          return this.sobrenome;
     }
     public String toString() {
          return this.sobrenome + ", " + this.nome;
     }
     public int hashCode() {
          return this.sobrenome.hashCode() * 2 +
               this.nome.hashCode() ;
     }
     public boolean equals(Object o) {
          cliente s = (cliente) o;
          return (this.nome.equals(s.getNome()) &&
               this.sobrenome.equals( s.getSobrenome()));
     }
     public int compareTo(Object o) {
          cliente s = (cliente) o;
          return (2 *
this.sobrenome.compareTo(s.getSobrenome()) +
               this.nome.compareTo( s.getNome()));
     }

}
public class exTreeMap {
     public static void main(String args[]) {
          cliente a = new cliente("silva", "maria");
          cliente b = new cliente("silva", "jose");
          cliente c = new cliente("pereira", "francisco");
          System.out.println("\nHashcodes a,b,c: " +
               a.hashCode() + ", " +
               b.hashCode() + ", " +
               c.hashCode());
          System.out.println("\nEquals a(b): " +
               a.equals(b));
          System.out.println("\ncompareTo a(b): " +
               a.compareTo(b));
          System.out.println("\ncompareTo b(c): " +
```

```
                b.compareTo(c));
            TreeMap al = new TreeMap();
            al.put(a.getNome(),a);
            al.put(b.getNome(),b);
            al.put(c.getNome(),c);
            Iterator li = al.values().iterator();
            while(li.hasNext()) {
                System.out.println("\nElemento: " +
                    (cliente)(li.next()));
            }

        }
}
```

A maior diferença é que não temos um método “listIterator”, mas podemos obter uma Collection através do método “values”, logo, podemos navegar utilizando o seu “iterator”. Os elementos virão na ordem da chave, dada pelo método “compareTo”.

Outras classes:

HashMap extends Dictionary implements Map, Cloneable, Serializable	Uma implementação de Map utilizando uma tabela baseada em hashCodes. Permite valures e chaves “null”. Não garante a ordenação dos valores.
HashSet extends AbstractSet implements Set, Cloneable, Serializable	Uma implementação da interface Set utilizando uma tabela baseada em hashCodes. Também não garante a ordem dos elementos.
Hashtable extends Dictionary implements Map, Cloneable, Serializable	Esta classe implementa uma tabela de hashCodes. As chaves devem ser diferentes de “null”. Os elementos utilizados como chave devem implementar os métodos “hashCode” e “equals”.
IdentityHashMap extends AbstractMap implements Map, Serializable, Cloneable	Implementa a interface Map em uma hash table utilizando a igualdade referencial (“==”) na comparação de chaves.
LinkedHashMap extends HashMap	Implementação de um Map utilizando hash table e linked list. A ordem é a de inserção na lista.
LinkedHashSet extends HashSet implements Set, Cloneable, Serializable	Implementação da interface Set utilizando uma tabela de hash codes e uma lista encadeada. A ordem utilizada é a de inserção.
TreeSet extends AbstractSet implements SortedSet, Cloneable, Serializable	Implementa a interface Set utilizando uma classe TreeMap. A ordem sera a ascendente, de acordo com a classificação dos elementos (Comparable ou Comparator).
WeakHashMap extends AbstractMap	Uma implementação da interface Map que

implements Map	permite chaves fracas. Uma entrada em um WeakHashMap será automaticamente removida quando não estiver mais em uso. Aceita valores e chaves nulos.

Questões e Exercícios

1) Case a primeira coluna com a segunda, se houver correspondência:

1) Abs(double a)	( ) Retorna o valor do ângulo em radianos
2) Retorna o menor valor, maior que o argumento	( ) cos(double a)
3) Retorna o valor do cosseno do ângulo em graus	( ) Retorna o valor absoluto do argumento
4) pow	( ) Eleva um número à potência de outro
5) toRadians	( ) Instancia a classe Math
6) Math m = new Math()	( ) ceil
	( ) floor

2) Analise o seguinte trecho de código em Java:

a) Int i = new Int(2);
b) Char c = i.intValue();
c) Double d = new Double(Double.parseDouble(i.toString()));
d) Float f = d.floatValue();

Existem linhas com erro de compilação? Caso positivo identifique-as.

3) Analise o seguinte trecho de código em java:

```
Arquivo: excol.java
-------------------------
1> import java.util.*;
2> public class excol {
3>    public static void main(String args[]) {
4>          Vector v = new Vector();
5>          for(int i=0; i<args.length; i++) {
6>                v.add(args[i]);
7>          }
8>          TreeSet ts=new TreeSet(v);
9>          Iterator it = ts.iterator();
10>         while(it.hasNext()) {
11>               System.out.println(it.next());
12>         }
13>   }
14> }
```

a) Há um erro de compilação
b) A linha 10 dará exception
c) Rodará e listara a mesma lista de argumentos fornecidos na mesma ordem
d) Se colocarmos um cast “(Collection)” na lina 8, compilará sem problemas
e) Nenhuma das anteriores

4) Analise o seguinte código em Java:

```
Arquivo: exset.java
-------------------------
1> import java.util.*;
2> public class exset {
3>    public static void main(String args[]) {
4>          TreeSet ts=new TreeSet();
5>          for(int i=0; i<args.length; i++) {
6>                ts.add(args[i]);
7>          }
8>          Iterator it = ts.iterator();
9>          while(it.hasNext()) {
10>               System.out.println(it.next());
11>         }
12>   }
13> }
```

a) Dará erro de compilação na lina 6
b) Compilará mas dará exception na linha 9
c) O resultado é exatamente igual ao do exercício anterior
d) Se trocarmos por HashSet o resultado será o mesmo
e) N.R.A.

Respostas

1)

1) Abs(double a)	(5) Retorna o valor do ângulo em radianos
2) Retorna o menor valor, maior que o argumento	( ) cos(double a)
3) Retorna o valor do cosseno do ângulo em graus	( ) Retorna o valor absoluto do argumento
4) pow	(4) Eleva um número à potência de outro
5) toRadians	( ) Instancia a classe Math
6) Math m = new Math()	(2) ceil
	( ) floor

Porque:

- A função que retorna o valor absoluto é “abs” e não “Abs”.
- A função “cos” retorna o valor do cosseno do ângulo fornecido em graus.
- A classe Math não pode ser instanciada porque o construtor é privado.
- Nada foi mencionado sobre “floor”.

2) Somente as linhas abaixo estão incorretas:

a) Int i = new Int(2);
b) Char c = i.intValue();

As classes Wrapper são, respectivamente, Integer e Character.

3) A resposta correta é “E”, nenhuma das anteriores, pois o programa listará os argumentos em ordem ascendente.

4) A resposta correta é “c”, pois terá o mesmo efeito da classe do exercício 3.

Entrada e Saída de dados

Exemplos em “exemplos/09”.

Em Java temos comandos para entrada e saída de dados como em qualquer linguagem. Podemos ler e gravar dados binários ou em formatos de alto nível utilizando os Streams. Também podemos acessar arquivos Randômicos como fazemos em outras linguagens.

Não podemos confundir E/S de dados com acesso a Bancos de Dados, o que em Java é feito pelo JDBC (java.sql).

Impressão direta

O pacote java.awt.print possui algumas classes úteis para impressão direta, com criação de PrintJobs etc.

Este assunto não faz parte do exame Java Programmer e nem é considerado parte do conhecimento básico de Java, mas, mesmo assim, resolvemos incluir alguma coisa sobre impressão direta para facilitar o entendimento.

Interfaces

- Pageable
  - É um conjunto de páginas a ser impresso.
- Printable
  - Implementa o método “print” que realmente imprime a página.
- PrinterGraphics
  - Implementada pelos objetos “graphics” que são passados para os objetos que implementam a classe “Printable”.

Classes

- Book
  - Representa um documento que pode ter formatos diferentes de páginas.
- PageFormat
  - Descreve uma página a ser impressa, no que tange a tamanho e orientação.
- Paper
  - Descreve as características físicas de um tipo de papel.
- PrinterJob
  - É a classe que controla todo o processo de impressão.

Exemplo de impressão

O arquivo “testeprinter.java” imprime uma página diretamente na impressora, invocando a caixa de diálogo padrão do Windows:

```
// Exemplo de impressão em Java - Cleuton
import java.awt.*;
import java.awt.event.*;
import java.awt.print.*;
import javax.swing.*;

public class testeprinter extends JFrame implements Printable {
      JButton btn1 = new JButton("Start");
      void initComponents() {
            btn1.addActionListener(new ActionListener() {
                  public void actionPerformed(ActionEvent e) {
                        imprimir();
                  }

            });
            Container cont = getContentPane();
            cont.setLayout(new FlowLayout());
            cont.add(btn1);
      }

      public static void main(String args[]) {
            testeprinter ta = new testeprinter();
            ta.setSize(200,200);
            ta.setLocationRelativeTo(null);
            ta.initComponents();
            ta.setVisible(true);
      }
      public void imprimir() {
            PrinterJob job = PrinterJob.getPrinterJob();
            PageFormat portrait = job.defaultPage();
            portrait.setOrientation(PageFormat.PORTRAIT);

            // Vamos usar um BOOK
            Book b = new Book();
            b.append(this,job.defaultPage());
            job.setPageable(b);
            if(job.printDialog()) {
                  try {
                        job.print();
                  }
                  catch (Exception e) {
                  }
            }
      }

      public int print(Graphics g, PageFormat pf, int pageIndex)
            throws PrinterException {
            Font fnt = new Font("Helvetica-Bold", Font.PLAIN, 48);
            g.setFont(fnt);
            g.setColor(Color.black);
            g.drawString("Teste de Impressão", 100, 200);
            return Printable.PAGE_EXISTS;
      }
}
```

O pacote java.io

Este pacote contém as principais classes e interfaces utilizadas para acessar dados em Java. Logo, se vamos utilizar estas características, temos que importá-lo.

A classe File

Esta classe representa os arquivos ou diretórios físicos. Ao contrário do que parece, ela não permite ler e gravar arquivos, mas lidar com eles.

Ela serve para obtermos informações sobre arquivos e diretórios em disco. Eis os métodos da classe File: (obtidos do tutorial Java – www.java.sun.com)

Construtores
`File``(File parent, String child)` Cria uma nova instância de File a partir de um diretório "pai" – parent – e um diretório "filho" – child.
`File``(String pathname)` Cria uma nova instância de File a partir do pathname informado.
`File``(String parent, String child)` Cria uma nova instância de File a partir de um diretório "pai" – parent (String) – e um diretório "filho" – child.
`File``(URI uri)` Cria uma nova instância convertendo a URI informada em um pathname.

Métodos	
`boolean`	**`canRead`**`()` Verifica se a aplicação pode ler o arquivo representado pela instância de File.
`boolean`	**`canWrite`**`()` Verifica se a aplicação pode gravar o arquivo representado pela instância de File.
`int`	**`compareTo`**`(File pathname)` Compara dois pathnames lexicograficamente.
`int`	**`compareTo`**`(Object o)` Compara este File com outro.
`boolean`	**`createNewFile`**`()` Cria um arquivo vazio apenas se não existir outro com o mesmo nome no disco.

static File	**createTempFile**(String prefix, String suffix) Cria um arquivo no diretório temporário padrão, com o prefixo e sufixo desejados para criar o nome.
static File	**createTempFile**(String prefix, String suffix, File directory) Cria um novo arquivo no diretório especificado usando o prefixo e o sufixo desejados.
boolean	**delete**() Deleta o arquivo físico (ou diretório) representado por esta instância de File.
void	**deleteOnExit**() Estabelece que o arquivo ou diretório representado pela instância de File seja deletado quando o programa terminar.
boolean	**equals**(Object obj) Verifica se esta instância é igual a outra.
boolean	**exists**() Verifica se o arquivo / diretório representado pela instância de File existe.
File	**getAbsoluteFile**() Retorna o caminho absoluto do arquivo.
String	**getAbsolutePath**() Idem só que na forma String.
File	**getCanonicalFile**() Retorna a forma canônica deste arquivo (resolve os “.” E “..”).
String	**getCanonicalPath**() Retorna a forma canônica como um String.
String	**getName**() Retorna o nome do arquivo ou diretório.
String	**getParent**() Retorna o pathname do diretório “pai” do File ou “null”.
File	**getParentFile**() Idem só que retorna uma instância de File.
String	**getPath**() Retorna o pathname.
int	**hashCode**() Retorna o hash code deste.
boolean	**isAbsolute**() Verifica se é um pathname absolute ou não.
boolean	**isDirectory**() Verifica se é um diretório.
boolean	**isFile**() Verifica se é um arquivo.
boolean	**isHidden**() Verifica se é oculto pelo sistema operacional.
long	**lastModified**()

	Retorna a data/hora em que este arquivo foi modificado.
long	**length**() Retorna o tamanho do arquivo físico.
String[]	**list**() Retorna um array de arquivos e diretórios que este diretório contém. Supondo que seja um diretório.
String[]	**list**(FilenameFilter filter) Retorna a lista de arquivos e diretórios neste diretório, que satisfaçam o filtro especificado.
File[]	**listFiles**() Retorna um array de Files dos arquivos e diretórios contidos neste diretório.
File[]	**listFiles**(FileFilter filter) Retorna um array de Files dos arquivos e diretórios contidos neste diretório, desde que satisfaçam o filtro informado. Usa como argumento a interface FileFilter.
File[]	**listFiles**(FilenameFilter filter) Retorna um array de Files dos arquivos e diretórios contidos neste diretório, desde que satisfaçam o filtro informado. Usa como argumento a interface FileNameFilter.
static File[]	**listRoots**() Lista todos os filesystems Roots.
boolean	**mkdir**() Cria o diretório encapsulado por esta instância de File.
boolean	**mkdirs**() Cria o diretório e todos os seus “pais” inexistentes.
boolean	**renameTo**(File dest) Renomeia o arquivo apontado por este File.
boolean	**setLastModified**(long time) Altera a data/hora da última modificação neste pathname. Semelhante ao “touch” do Unix.
boolean	**setReadOnly**() Marca este arquivo ou diretório como “read only”.
String	**toString**() Retorna o pathname deste File.
URI	**toURI**() Retorna uma URI que representa este pathname.
URL	**toURL**() Converte este pathname em uma URL.

Aqui está um pequeno exemplo de uso da classe File (arquivo “readfile.java”):

```
import java.io.*;
import java.util.Date;
public class readfile {
      public static void main(String args[]) {
            File f = new File("readfile.java");
            System.out.println("\nArquivo: " + f.getName() + "\n");
            System.out.println("\nTamanho: " + f.length() + "\n");
            Date dt = new Date(f.lastModified());
            System.out.println("\nModificado em: " + dt + "\n");
            try {
                  FileInputStream fi = new FileInputStream(f);
                  byte lido = 0;
                  while((lido = (byte) fi.read()) >= 0) {
                        System.out.print((char)lido);
                  }
                  fi.close();
            }
            catch (FileNotFoundException fnf) {
            }
            catch (IOException ioe) {
            }
      }
}
```

A versatilidade da classe File se dá quando desejamos navegar por um path, como no exemplo recursivo “readpath.java”:

```
import java.io.*;
import java.util.Date;
public class readpath {
      public static void main(String args[]) {
            File f = new File(args[0]);
            if(!f.isDirectory()) {
                  System.out.println("\nNao eh um diretorio");
            }
            else {
                  readpath rp = new readpath();
                  rp.processa("",f);
            }
      }
      public void processa(String desloc, File arq) {
            System.out.println("\n" + desloc + arq.getName());
            File arqs[] = arq.listFiles();
            for(int i=0;i<arqs.length;i++) {
                  String desloc2 = desloc + "    ";
                  System.out.println("\n" + desloc2 +
                        arqs[i].getName());
                  if(arqs[i].isDirectory()) {
                        processa(desloc2 + "    ",
                              arqs[i]);
                  }
            }
      }
}
```

A classe RandomAccessFile

Esta classe permite ler e gravar arquivos de acesso aleatório ou randômico. Podemos utilizar RandomAccessFile com praticamente qualquer arquivo, embora sejam mais úteis com arquivos que possuam conjuntos repetitivos (registros) de estruturas semelhantes. (Obtido do tutorial Java – www.java.sun.com).

Construtores
RandomAccessFile(File file, String mode) Cria um RandomAccessFile a partir do File informado e com o modo de acesso especificado em "mode" ("r" – read; "rw" – read e write; "rws" – read e write com gravação imediata – incluindo metadados; "rwd" – read e write com gravação imediata).
RandomAccessFile(String name, String mode) Cria um RandomAccessFile a partir do pathname informado.

Métodos	
void	**close**() Fecha o arquivo.
FileChannel	**getChannel**() Retorna o FileChannel associado a este arquivo.
FileDescriptor	**getFD**() Retorna o FileDescriptor associado com este arquivo.
long	**getFilePointer**() Retorna a posição dentro do arquivo (a partir do início).
long	**length**() Retorna o tamanho deste arquivo.
int	**read**() Lê um byte deste arquivo.
int	**read**(byte[] b) Lê tantos bytes quantos caibam no array informado.
int	**read**(byte[] b, int off, int len) Lê tantos bytes quantos o argumento "len" especificar, a partir do deslocamento informado em "off".
boolean	**readBoolean**() Lê um boolean.
byte	**readByte**() Lê um byte sinalizado do arquivo.
char	**readChar**() Lê um character Unicode do arquivo.
double	**readDouble**() Lê um valor double.

float	**readFloat**() Lê um float.
void	**readFully**(byte[] b) Lê tantos bytes quantos couberem no array começando da posição atual.
void	**readFully**(byte[] b, int off, int len) lê exatamente “len” bytes, a partir do deslocamento.
int	**readInt**() Lê um inteiro 32 bits.
String	**readLine**() Lê uma linha inteira (até o \r\n exclusive).
long	**readLong**() Lê um inteiro de 64 bits.
short	**readShort**() Lê um inteiro de 16 bits.
int	**readUnsignedByte**() Lê um byte não sinalizado.
int	**readUnsignedShort**() Lê um inteiro de 16 bits não sinalizado.
String	**readUTF**() Lê um String. (os primeiros dois bytes dão o comprimento da codificação do String).
void	**seek**(long pos) Posiciona o arquivo para futures leituras e gravações.
void	**setLength**(long newLength) Modifica o tamanho do arquivo.
int	**skipBytes**(int n) Pula “n” bytes descartando-os.
void	**write**(byte[] b) Grava os bytes do array..
void	**write**(byte[] b, int off, int len) Grava “len” bytes do array, começando no deslocamento “off” deste arquivo.
void	**write**(int b) Grava o **byte** especificado pelo argumento no arquivo.
void	**writeBoolean**(boolean v) Grava um boolean como um só byte no arquivo.
void	**writeByte**(int v) Grava o valor “v” como um só byte.
void	**writeBytes**(String s) Grava o string “s” como uma seqüência de bytes.
void	**writeChar**(int v) Grava o char representado por “v” no arquivo. O byte de primeira ordem virá primeiro.
void	**writeChars**(String s)

	Grava o string como se fossem várias chamadas de "writeChar".
void	**writeDouble**(double v) Converte o double em long (Double.doubleToLongBits) e grava como 8 bytes.
void	**writeFloat**(float v) Converte o float "v" em int, utilizando o método "Float.floatToIntBits" e grava como quarto bytes.
void	**writeInt**(int v) Grava "v" como um inteiro, quarto bytes.
void	**writeLong**(long v) Grava um longo como oito bytes.
void	**writeShort**(int v) Grava um short (16 bits).
void	**writeUTF**(String str) Grava um string como um UTF-8 de maneira independente de plataforma.

Para demonstrar o uso de RandomAccessFile temos os exemplos "writerandom.java" e "readrandom.java":

"writerandom.java":

```
import java.io.*;
public class writerandom {
      public static void main(String args[]) {
            try {
                  RandomAccessFile rf = new
RandomAccessFile("saida.dat","rw");
                  rf.writeUTF(args[0]);
                  rf.writeInt(args[0].length());
                  rf.close();
            }
            catch (FileNotFoundException fnf) {
            }
            catch (IOException io) {
            }
      }
}
```

"readrandom.java"

```
import java.io.*;
public class readrandom {
      public static void main(String args[]) {
            try {
                  RandomAccessFile rf = new
RandomAccessFile("saida.dat","r");
                  String lido = rf.readUTF();
                  System.out.println("\nString lido: " + lido);
                  int tamanho = rf.readInt();
                  System.out.println("\nint lido: " + tamanho);
                  rf.close();
```

```
            }
            catch (FileNotFoundException fnf) {
            }
            catch (IOException io) {
            }
      }
}
```

Streams

Um Stream é uma abstração que permite transportar dados de um dispositivo de/para um programa ou outro Stream.

Podemos também enxergar um Stream como um "canal" por onde passam os dados, cujas pontas podem ser: um dispositivo, um programa, a memória ou outro Stream.

No pacote java.io temos os seguintes elementos (listamos apenas os relevantes): (obtidos da documentação do Java – API – www.java.sun.com).

Interfaces

- DataInput
  - Permite ler bytes de um Stream e reconstruir os tipos primitivos em Java.
- DataOutput
  - Converte qualquer primitivo Java em uma série de bytes a serem gravados em um Stream binário.
- ObjectInput
  - Estende a interface DataInput para incluir a leitura de Objetos Java, arrays e Strings.
- ObjectOutput
  - Estente a interface DataOutput para incluir a gravação de Objetos, arrays e Strings.
- Serializable
  - Indica que a classe que a implementa pode ser serializada, ou gravada em disco. Não possui métodos.

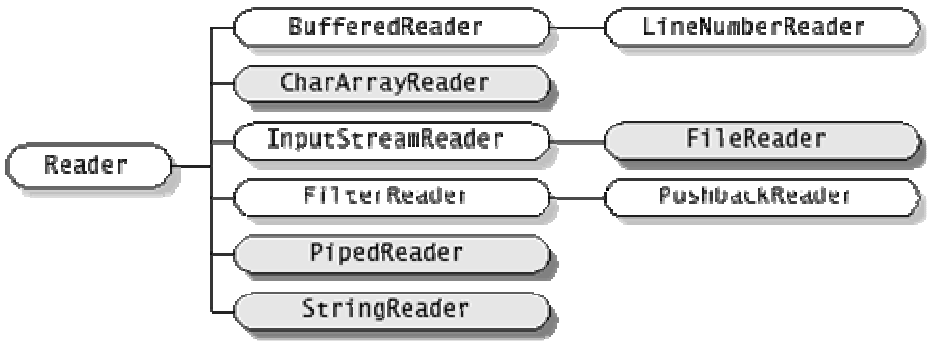
Classes

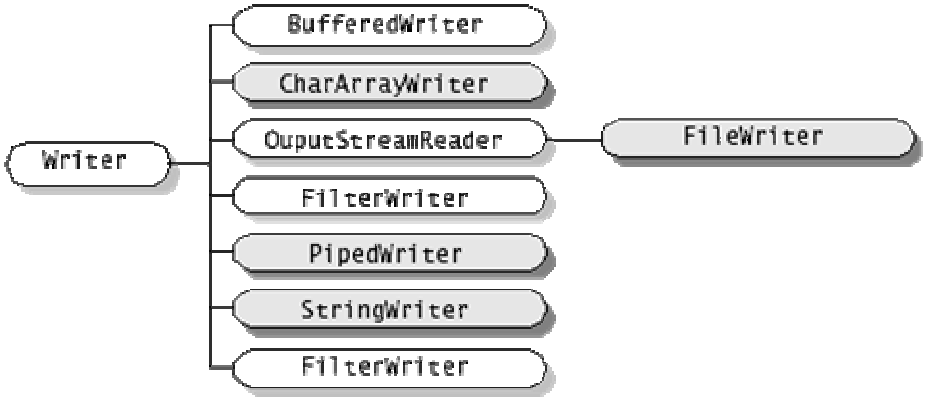
- BufferedInputStream
  - Adiciona funcionalidade a outro Input Stream para incluir armazenamento (bufferização) de dados. Torna a leitura mais eficiente.
- BufferedOutputStream
  - Implementa uma saída bufferizada, com isto evita-se a chamada a rotina de IO para cada byte gravado.
- BufferedReader
  - Lê texto de um Stream de entrada em caracteres, bufferizando-os para tornar a leitura de caracteres, arrays e linhas mais eficiente.
- BufferedWriter
  - Grava texto em um Stream de saída (caracteres), utilizando um buffer de saída para tornar o processo mais eficiente.
- ByteArrayInputStream

    - Contém um buffer interno que armazena os bytes a serem fornecidos. É um Stream para um array de bytes.
- ByteArrayOutputStream
    - Esta classe implementa um Stream de saída no qual os dados são gravados em um array de bytes.
- CharArrayReader
    - Implementa um buffer de caracteres que pode ser utilizado como Stream de entrada de dados.
- CharArrayWriter
    - Implementa um buffer de caracteres que pode ser utilizado como Stream de saída.
- DataInputStream
    - Permite que uma aplicação leia tipos de dados primitivos Java a partir de um Stream de Entrada binário.
- DataOutputStream
    - Permite que uma aplicação grave tipos de dados primitivos Java em um Stream de saída binário.
- FileInputStream
    - Obtém bytes a partir de um arquivo físico.
- FileOutputStream
    - Grava bytes em um arquivo físico (File).
- FileReader
    - Lê caracteres de arquivos em formato textual.
- FileWriter
    - Grava caracteres em arquivos em formato textual.
- InputStream
    - Esta classe abstrata é a superclasse de todos os Streams de Entrada em bytes.
- InputStreamReader
    - Funciona convertendo bytes em caracteres. Lê bytes e os decodifica em caracteres.
- LineNumberReader
    - Um Stream de entrada de caracteres bufferizado que conta os números de linhas físicas lidas.
- ObjectInputStream
    - Permite des-serializar objetos previamente gravados utilizando ObjectOutputStream.
- ObjectOutputStream
    - Permite serializar objetos tornando-os persistentes.
- OutputStream
    - É a superclasse (abstrata) de todas as classes que gravam Bytes.
- OutputStreamWriter
    - Transforma caracteres em bytes para saída.
- PipedInputStream
    - Se conecta a um PipedOutputStream para obter dados deste. Funcionam como mecanismo de comunicação entre Threads.
- PipedOutputStream
    - Pode ser conectado a um PipedInputStream para formar um canal de comunicação entre dois Objetos ou Threads diferentes.

- PipedReader
  - A mesma coisa que um PipedInputStream só que lida com caracteres.
- PipedWriter
  - Idem para PipedOutputStream com caracteres.
- PrintStream
  - Adiciona funcionalidade a outro Stream de saída para poder graver representações dos tipos de dados Java (incluindo String).
- PrintWriter
  - Imprime os dados formatados em um Stream de saída de caracteres. Implementa todos os métodos de PrintStream.
- PushbackInputStream
  - Adiciona funcionalidade de "push back" a outro Stream de Entrada. Com isto podemos retroalimentar cada byte lido.
- PushbackReader
  - Um Stream de entrada de caracteres que permite realimentá-los.
- Reader
  - Classe abstrata para ler Streams de Caracteres.
- StringReader
  - Um Stream de Entrada de caracteres cuja fonte é um String.
- StringWriter
  - Um Stream de saída de caracteres que gera um StringBuffer.
- Writer
  - Classe abstrata para gravar em Streams de saída de caracteres.

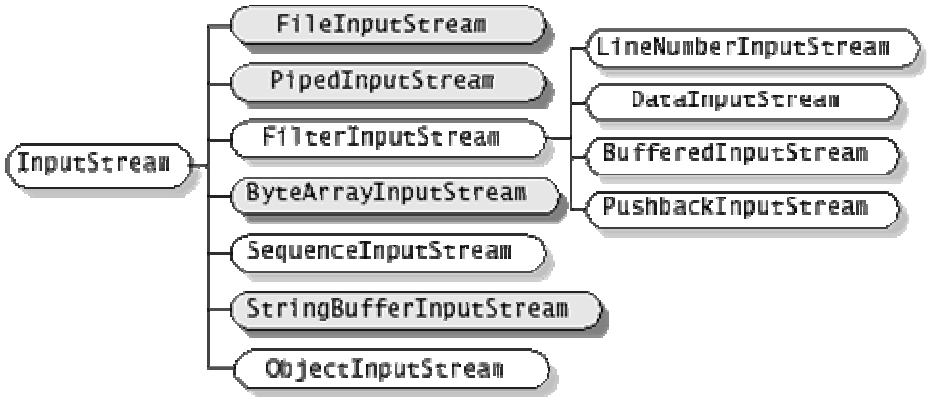
Dentro do tutorial java (www.java.sun.com) temos um gráfico que mostra o relacionamento entre os vários Streams, tanto binários (Byte Streams) quanto caracteres (Character Streams):

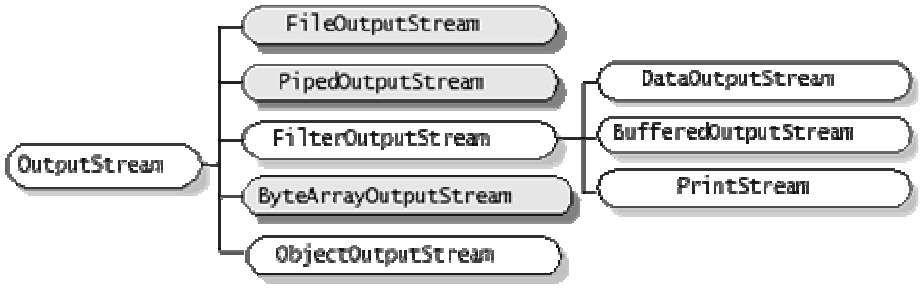




Os marcados em cinza lêem ou gravam diretamente de fontes de dados. Os outros lêem ou gravam de/para outros Streams.

Para byte streams:





Alguns Streams lidam diretamente com a fonte de dados. Vejamos o exemplo “directstreams.java”:

```
import java.io.*;
public class directstreams {
      public static void main(String args[])
                  throws IOException, FileNotFoundException {
            File f = new File("directstreams.java");
            FileInputStream fi = new FileInputStream(f);
            String s = "";
            int a = 0;
            while((a = fi.read()) >= 0) {
                  s += (char) a;
            }
            fi.close();
            StringReader sr = new StringReader(s);
            String w = "";
            int b = 0;
            while((b = sr.read()) >= 0) {
                  w += (char) b;
            }
            sr.close();
            System.out.println(w);
      }
}
```

Buffered Streams

Para leitura e gravação de arquivos normalmente “conectamos” um Stream a outro, de modo a permitir a leitura bufferizada de tipos de dados Java. Podemos conectar um BufferedInputStream ou um BufferedReader a qualquer outro InputStream, o que tornará a leitura ou gravação mais eficiente. Além disto podemos conectar um Stream de Leitura de alto nível, que permite ler tipos Java, ao nosso buffer, permitindo maior flexibilidade na programação.

Eis um exemplo completo de leitura bufferizada (“bufferedstreams.java”):

```
import java.io.*;
public class bufferedstreams {
      public static void main(String args[])
                  throws IOException, FileNotFoundException {
            File f = new File("bufferedstreams.java");
            FileInputStream fi = new FileInputStream(f);
            BufferedInputStream bi = new BufferedInputStream(fi);
            DataInputStream di = new DataInputStream(bi);
            String s = "";
            int a = 0;
            try {
                  while((a = di.readUnsignedByte()) >= 0) {
                        s += (char) a;
                  }
            }
            catch (EOFException eof) {
                  System.out.println(s);
                  di.close();
                  bi.close();
                  fi.close();
            }
      }
}
```

Neste caso o FileInputReader vai ler o arquivo sob controle do BufferedInputStream. O DataInputStream vai ler tipos primitivos do Buffer do arquivo. O método "readUnsignedByte" levanta a exception "EOFException" quando atinge o fim do arquivo.

É bom lembrarmos quais Streams lidam diretamente com fontes de dados e quals necessitam de outro Stream para fornecer entrada ou receber saída. Isto pode ser visto nas figuras anteriores.

Eis um exemplo de grava e lê dados (arquivo "readwrite.java"):

```
import java.io.*;
public class readwrite {
      public static void main(String args[])
                  throws IOException, FileNotFoundException {
            File f = new File("readwrite.dat");
            FileOutputStream fo = new FileOutputStream(f);
            BufferedOutputStream bo = new BufferedOutputStream(fo);
            DataOutputStream dos = new DataOutputStream(bo);
            dos.writeUTF(args[0]);
            dos.writeInt(5015);
            dos.writeDouble(101.34);
            dos.close();
            bo.close();
            fo.close();

            FileInputStream fi = new FileInputStream(f);
            BufferedInputStream bi = new BufferedInputStream(fi);
            DataInputStream di = new DataInputStream(bi);
            try {
                  String lido = di.readUTF();
                  int numero = di.readInt();
                  double valor = di.readDouble();
                  System.out.println(lido + ", " +
                        numero + ", " + valor);
            }
            catch (EOFException eof) {
                  System.out.println("\nErro");
            }
            di.close();
            bi.close();
            fi.close();
      }
}
```

UTF é a representação de caracteres utilizando o idioma atual (Locale). Ele utiliza o número de bytes necessários para representar cada caracter.

Questões e Exercícios

1) Marque os Streams que podem ser utilizados para ler diretamente do File:
File f = new File(“teste.dat”);

a) BufferedInputStream
b) FileReader
c) DataInputStream
d) FileInputStream

2) Como podemos detetar o fim de arquivo quando lendo um File?

a) Com o método “isEof()” da classe File.
b) Com a propriedade “eof”, comparando-a com “File.EOF”.
c) Interceptando a exception “EOFException”.
d) Testando se o resultado do método “read()” é positivo.
e) Nenhuma das respostas anteriores.

3) Analise o seguinte código-fonte Java:

```
Arquivo: readfile.java
-------------------------
1> import java.io.*;
2> import java.util.Date;
3> public class readfile {
4>    public static void main(String args[]) {
5>          try {
6>                FileReader f = new FileReader("readfile.java");
7>                byte lido = 0;
8>                while((lido = (byte) f.read()) >= 0) {
9>                      System.out.print((char)lido);
10>               }
11>               f.close();
12>         }
13>         catch (FileNotFoundException fnf) {
14>         }
15>         catch (IOException ioe) {
16>         }
17>   }
18> }
```

a) Dará erro de compilação somente na linha 8.
b) Dará erro de compilação nas linhas 8 e 9.
c) Dará exception.
d) Se trocarmos por FileInputStream passará sem problemas.
e) Não imprimirá o tamanho do arquivo.

Respostas

1) As letras “b” e “d” lêem diretamente de um File. As outras respostas representam Streams de mais alto nível.

2) A letra “e” é a única resposta possível. A classe File não lê arquivos, apenas os representa.

3) A única resposta que não está errada é a letra “e”.

Anexo – Licença GNU de documentação livre

Esta é uma tradução não oficial da Licença de Documentação Livre GNU em Português Brasileiro. Ela não é publicada pela Free Software Foundation, e não se aplica legalmente a distribuição de textos que usem a GFDL - apenas o texto original em Inglês da GNU FDL faz isso. Entretanto, nós esperamos que esta tradução ajude falantes de português a entenderem melhor a GFDL.

Licença de Documentação Livre GNU

Versão 1.1, Março de 2000

0. INTRODUÇÃO

O propósito desta Licença é deixar um manual, livro-texto ou outro documento escrito ``livre" no sentido de liberdade: assegurar a qualquer um a efetiva liberdade de copiar ou redistribuí-lo, com ou sem modificações, comercialmente ou não. Secundariamente, esta Licença mantém para o autor e editor uma forma de ter crédito por seu trabalho, sem ser considerado responsável pelas modificações feitas por terceiros.

Esta licença é um tipo de ``copyleft" (``direitos revertidos"), o que significa que derivações do documento precisam ser livres no mesmo sentido. Ela complementa a GNU Licença Pública Geral (GNU GPL), que é um copyleft para software livre.

Nós fizemos esta Licença para que seja usada em manuais de software livre, porque software livre precisa de documentação livre: um programa livre deve ser acompanhado de manuais que forneçam as mesmas liberdades que o software possui. Mas esta Licença não está restrita a manuais de software; ela pode ser usada para qualquer trabalho em texto, independentemente do assunto ou se ele é publicado como um livro impresso. Nós recomendamos esta Licença principalmente para trabalhos cujo propósito seja de intrução ou referência.

1. APLICABILIDADE E DEFINIÇÕES

Esta Licença se aplica a qualquer manual ou outro texto que contenha uma nota colocada pelo detentor dos direitos autorias dizendo que ele pode ser distribuído sob os termos desta Licença. O ``Documento", abaixo, se refere a qualquer tal manual ou texto. Qualquer pessoa do público é um licensiado e é referida como ``você".

Uma ``Versão Modificada" do Documento se refere a qualquer trabalho contendo o documento ou uma parte dele, quer copiada exatamente, quer com modificações e/ou traduzida em outra lingua.

Uma ``Seção Secundária" é um apêndice ou uma seção inicial do Documento que trata exclusivamente da relação dos editores ou dos autores do Documento com o assunto geral do Documento (ou assuntos relacionados) e não contém nada que poderia ser incluido diretamente nesse assunto geral. (Por exemplo, se o Documento é em parte um livro texto de matemática, a Seção Secundária pode não explicar nada de matemática). Essa relação poderia ser uma questão de ligação histórica com o assunto, ou matérias relacionadas, ou de posições legais, comerciais, filosóficas, éticas ou políticas relacionadas ao mesmo.

As ``Seções Invariantes" são certas Seções Secundárias cujos títulos são designados, como sendo de Seções Invariantes, na nota que diz que o Documento é publicado sob esta Licença.

Os ``Textos de Capa" são certos trechos curtos de texto que são listados, como Textos de Capa Frontal ou Textos da Quarta Capa, na nota que diz que o texto é publicado sob esta Licença.

Uma cópia ``Transparente" do Documento significa uma cópia que pode ser lida automaticamente, representada num formato cuja especificação esteja disponível ao público geral, cujos conteúdos possam ser vistos e editados diretamente e sem mecanismos especiais com editores de texto genéricos ou (para imagens compostas de pixels) programas de pintura genéricos ou (para desenhos) por algum editor de desenhos grandemente difundido, e que seja passível de servir como entrada a formatadores de texto ou para tradução automática para uma variedade de formatos que sirvam de entrada para formartadores de texto. Uma cópia feita em um formato de arquivo outrossim Transparente cuja constituição tenha sido projetada para atrapalhar ou desencorajar modificações subsequentes pelos leitores não é Transparente. Uma cópia que não é ``Transparente" é chamada de ``Opaca".

Exemplos de formatos que podem ser usados para cópias Transparentes incluem ASCII simples sem marcações, formato de entrada do Texinfo, formato de entrada do LaTeX, SGML ou XML usando uma DTD disponibilizada publicamente, e HTML simples, compatível com os padrões, e projetado para ser modificado por pessoas. Formatos opacos incluem PostScript, PDF, formatos proprietários que podem ser lidos e editados apenas com processadores de texto proprietários, SGML ou XML para os quais a DTD e/ou ferramentas de processamento e edição não estejam disponíveis para o público, e HTML gerado automaticamente por alguns editores de texto com finalidade apenas de saída.

A ``Página do Título" significa, para um livro impresso, a página do título propriamente dita, mais quaisquer páginas subseqüentes quantas forem necessárias para conter, de forma legível, o material que esta Licença requer que apareça na página do título. Para trabalhos que não tenham uma tal página do título, ``Página do Título" significa o texto próximo da aparição mais proeminente do título do trabalho, precedendo o início do corpo do texto.

2. FAZENDO CÓPIAS EXATAS

Você pode copiar e distribuir o Documento em qualquer meio, de forma comercial ou não comercial, desde que esta Licença, as notas de copyright, e a nota de licença

dizendo que esta Licença se aplica ao documento estejam reproduzidas em todas as cópias, e que você não acrescente nenhuma outra condição quaisquer que sejam às desta Licença.

Você não pode usar medidas técnicas para obstruir ou controlar a leitura ou confecção de cópias subsequentes das cópias que você fizer ou distribuir. Entretanto, você pode aceitar compensação em troca de cópias. Se você distribuir uma quantidade grande o suficiente de cópias, você também precisa respeitar as condições da seção 3.

Você também pode emprestar cópias, sob as mesmas condições colocadas acima, e você também pode exibir coópias publicamente.

3. FAZENDO CÓPIAS EM QUANTIDADE

Se você públicar cópias do Documento em número maior que 100, e a nota de licença do Documento obrigar Textos de Capa, você precisa incluir as cópias em capas que tragam, clara e legivelmente, todos esses Textos de Capa: Textos de Capa da Frente na capa da frente, e Textos da Quarta Capa na capa de trás. Ambas as capas também precisam identificar clara e legivelmente você como o editor dessas cópias. A capa da frente precisa apresentar o título completo com todas as palavras do título igualmente proeminentes e visíveis. Você pode adicionar outros materiais às capas. Fazer cópias com modificações limitadas às capas, tanto quanto estas preservem o título do documento e satisfaçam essas condições, pode tratado como cópia exata em outros aspectos.

Se os textos requeridos em qualquer das capas for muito volumoso para caber de forma legível, você deve colocar os primeiros (tantos quantos couberem de forma razoável) na capa verdadeira, e continuar os outros nas páginas adjacentes.

Se você publicar ou distribuir cópias Opacas do Documento em número maior que 100, você precisa ou incluir uma cópia Transparente que possa ser lida automaticamente com cada cópia Opaca, ou informar em ou com cada cópia Opaca a localização de uma cópia Transparente completa do Documento acessível publicamente em uma rede de computadores, à qual o público usuário de redes tenha acesso a download gratuito e anônimo utilizando padrões públicos de protocolos de rede. Se você utilizar o segundo método, você precisa tomar cuidados razoavelmente prudentes, quando iniciar a distribuição de cópias Opacas em quantidade, para assegurar que esta cópia Transparente vai permanecer acessível desta forma na localização especificada por pelo menos um ano depois da última vez em que você distribuir uma cópia Opaca (diretamente ou através de seus agentes ou distribuidores) daquela edição para o público.

É pedido, mas não é obrigatório, que você contate os autores do Documento bem antes de redistribuir qualquer grande número de cópias, para lhes dar uma oportunidade de prover você com uma versão atualizada do Documento.

4. MODIFICAÇÕES

Você pode copiar e distribuir uma Versão Modificada do Documento sob as condições das seções 2 e 3 acima, desde que você publique a Versão Modificada estritamente sob

esta Licença, com a Versão Modificada tomando o papel do Documento, de forma a licenciar a distribuição e modificação da Versão Modificada para quem quer que possua uma cópia da mesma. Além disso, você precisa fazer o seguinte na versão modificada:

- **A**. Usar na Página de Título (e nas capas, se alguma) um título distinto daquele do Documento, e daqueles de versões anteriores (que deveriam, se houvesse algum, estarem listados na seção Histórico do Documento). Você pode usar o mesmo título de uma versão anterior se o editor original daquela versão lhe der permissão.
- **B**. Listar na Página de Título, como autores, uma ou mais das pessoas ou entidades responsáveis pela autoria das modificações na Versão Modificada, conjuntamente com pelo menos cinco dos autores principais do Documento (todos os seus autores principais, se ele tiver menos que cinco).
- **C**. Colocar na Página de Título o nome do editor da Versão Modificada, como o editor.
- **D**. Preservar todas as notas de copyright do Documento.
- **E**. Adicionar uma nota de copyright apropriada para suas próprias modificações adjacente às outras notas de copyright.
- **F**. Incluir, imediatamente depois das notas de copyright, uma nota de licença dando ao público o direito de usar a Versão Modificada sob os termos desta Licença, na forma mostrada no Adendo abaixo.
- **G**. Preservar nessa nota de licença as listas completas das Seções Invariantes e os Textos de Capa requeridos dados na nota de licença do Documento.
- **H**. Incluir uma cópia inalterada desta Licença.
- **I**. Preservar a seção entitulada ``Histórico", e seu título, e adicionar à mesma um item dizendo pelo menos o título, ano, novos autores e editor da Versão Modificada como dados na Página de Título. Se não houver uma sessão denominada ``Histórico"; no Documento, criar uma dizendo o título, ano, autores, e editor do Documento como dados em sua Página de Título, então adicionar um item descrevendo a Versão Modificada, tal como descrito na sentença anterior.
- **J**. Preservar o endereço de rede, se algum, dado no Documento para acesso público a uma cópia Transparente do Documento, e da mesma forma, as localizações de rede dadas no Documento para as versões anteriores em que ele foi baseado. Elas podem ser colocadas na seção ``Histórico". Você pode omitir uma localização na rede para um trabalho que tenha sido publicado pelo menos quatro anos antes do Documento, ou se o editor original da versão a que ela ser refira der sua permissão.
- **K**. Em qualquer seção entitulada ``Agradecimentos"; ou ``Dedicatórias";, preservar o título da seçãom e preservar a seção em toda substância e tim de cada um dos agradecimentos de contribuidores e/ou dedicatórias dados.
- **L**. Preservar todas as Seções Invariantes do Documento, inalteradas em seus textos ou em seus títulos. Números de seção ou equivalentes não são considerados parte dos títulos da seção.
- **M**. Apagar qualquer seção entitulada ``Endossos";. Tal sessão não pode ser incluída na Versão Modificada.
- **N**. Não re-entitular qualquer seção existente com o título ``Endossos"; ou com qualquer outro título dado a uma Seção Invariante.

Se a Versão Modificada incluir novas seções iniciais ou apêndices que se qualifiquem como Seções Secundárias e não contenham nenhum material copiado do Documento, você pode optar por designar alguma ou todas aquelas seções como invariantes. Para fazer isso, adicione seus títulos à lista de Seções Invariantes na nota de licença da Versão Modificada. Esses títulos precisam ser diferentes de qualquer outro título de seção.

Você pode adicionar uma seção entitulada ``Endossos";, desde que ela não contenha qualquer coisa além de endossos da sua Versão Modificada por várias pessoas ou entidades - por exemplo, declarações de revisores ou de que o texto foi aprovado por uma organização como a definição oficial de um padrão.

Você pode adicionar uma passagem de até cinco palavras como um Texto de Capa da Frente , e uma passagem de até 25 palavras como um Texto de Quarta Capa, ao final da lista de Textos de Capa na Versão Modificada. Somente uma passagem de Texto da Capa da Frente e uma de Texto da Quarta Capa podem ser adicionados por (ou por acordos feitos por) qualquer entidade. Se o Documento já incluir um texto de capa para a mesma capa, adicionado previamente por você ou por acordo feito com alguma entidade para a qual você esteja agindo, você não pode adicionar um outro; mas você pode trocar o antigo, com permissão explícita do editor anterior que adicionou a passagem antiga.

O(s) autor(es) e editor(es) do Documento não dão permissão por esta Licença para que seus nomes sejam usados para publicidade ou para assegurar ou implicar endossamento de qualquer Versão Modificada.

5. COMBINANDO DOCUMENTOS

Você pode combinar o Documento com outros documentos publicados sob esta Licença, sob os termos definidos na seção 4 acima para versões modificadas, desde que você inclua na combinação todas as Seções Invariantes de todos os documentos originais, sem modificações, e liste todas elas como Seções Invariantes de seu trabalho combinado em sua nota de licença.

O trabalho combinado precisa conter apenas uma cópia desta Licença, e Seções Invariantes Idênticas com múltiplas ocorrências podem ser substituídas por apenas uma cópia.Se houver múltiplas Seções Invariantes com o mesmo nome mas com conteúdos distintos, faça o título de cada seção único adicionando ao final do mesmo, em parênteses, o nome do autor ou editor origianl daquela seção, se for conhecido, ou um número que seja único. Faça o mesmo ajuste nos títulos de seção na lista de Seções Invariantes nota de licença do trabalho combinado.

Na combinação, você precisa combinar quaisquer seções intituladas ``Histórico"; dos diversos documentos originais, formando uma seção entitulada ``Histórico"; da mesma forma combine quaisquer seções entituladas ``Agradecimentos", ou ``Dedicatórias". Você precisa apagar todas as seções intituladas como ``Endosso".

6. COLETÂNEAS DE DOCUMENTOS

Você pode fazer uma coletânea consitindo do Documento e outros documentos publicados sob esta Licença, e substituir as cópias individuais desta Licença nos vários documentos com uma única cópia incluída na coletânea, desde que você siga as regras desta Licença para cópia exata de cada um dos Documentos em todos os outros aspectos.

Você pode extrair um único documento de tal coletânea, e distribuí-lo individualmente sob esta Licença, desde que você insira uma cópia desta Licença no documento extraído, e siga esta Licença em todos os outros aspectos relacionados à cópia exata daquele documento.

7. AGREGAÇÃO COM TRABALHOS INDEPENDENTES

Uma compilação do Documento ou derivados dele com outros trabalhos ou documentos separados e independentes, em um volume ou mídia de distribuição, não conta como uma Versão Modificada do Documento, desde que não seja reclamado nenhum copyright de compilação seja reclamado pela compilação. Tal compilação é chamada um ``agregado", e esta Licença não se aplica aos outros trabalhos auto-contidos compilados junto com o Documento, só por conta de terem sido assim compilados, e eles não são trabalhos derivados do Documento.

Se o requerido para o Texto de Capa na seção 3 for aplicável a essas cópias do Documento, então, se o Documento constituir menos de um quarto de todo o agregado, os Textos de Capa do Documento podem ser colocados em capas adjacentes ao Documento dentro do agregado. Senão eles precisam aparecer nas capas de todo o agregado.

8. TRADUÇÃO

A tradução é considerada como um tipo de modificação, então você pode distribuir traduções do Documento sob os termos da seção 4. A substituição de Seções Invariantes por traduções requer uma permissão especial dos detentores do copyright das mesmas, mas você pode incluir traduções de algumas ou de todas as Seções Invariantes em adição as versões orginais dessas Seções Invariantes. Você pode incluir uma tradução desta Licença desde que você também inclua a versão original em Inglês desta Licença. No caso de discordância entre a tradução e a versão original em Inglês desta Licença, a versão original em Inglês prevalecerá.

9. TÉRMINO

Você não pode copiar, modificar, sublicenciar, ou distribuir o Documento exceto como expressamente especificado sob esta Licença. Qualquer outra tentativa de copiar, modificar, sublicenciar, ou distribuir o Documento é nula, e resultará automaticamente no término de seus direitos sob esta Licença. Entretanto, terceiros que tenham recebido cópias, ou direitos, de você sob esta Licença não terão suas licenças terminadas tanto quanto esses terceiros permaneçam em total acordo com esta Licença.

10. REVISÕES FUTURAS DESTA LICENÇA

A Free Software Foundation pode publicar novas versões revisadas da Licença de Documentação Livre GNU de tempos em tempos. Tais novas versões serão similares em espírito à versão presente, mas podem diferir em detalhes ao abordarem novos porblemas e preocupações. Veja http://www.gnu.org/copyleft/.

A cada versão da Licença é dado um número de versão distinto. Se o Documento especificar que uma versão particular desta Licença ``ou qualquer versão posterior'' se aplica ao mesmo, você tem a opção de seguir os termos e condições daquela versão específica, ou de qualquer versão posterior que tenha sido publicada (não como rascunho) pela Free Software Foundation. Se o Documento não especificar um número de Versão desta Licença, você pode escolher qualquer versão já publicada (não como rascunho) pela Free Software Foundation.

Índice Remissivo