JAVA

by Jorge L. de Souza Leão

Atualizada para Java 8 até o capítulo 3 em 21 de março de 2015

Sumário

1. Introdução
2. O ambiente **jdk 8**
3. A linguagem básica
4. O ***package java.lang***
5. ***Enums***
6. O ***package java.io***
7. Programação multitarefa
8. Programação distribuída: ***package java.rmi***

1. Introdução

Histórico e Motivação

A linguagem Java é um produto de desenvolvimento da SUN Microsystems.

O trabalho original que levou à linguagem Java foi conduzido por James Gosling e uma equipe que desenvolvia software de aplicação e sistemas operacionais para produtos eletrônicos de consumo.

Eles utilizavam C++, até que a equipe de Gosling decidiu criar uma linguagem completamente nova.

1. Introdução

Traduzido de aathishankaran, "The creation of java", visto em www.java-samples.com/showtutorial.php?tutorialid=24 , em 08/mar/2012:

"Em 1991, James Gosling, Patrick Naughton, Chris Warth, Ed Frank, e Mike Sheridan conceberam a linguagem Java na Sun Microsystems, Inc..

O desenvolvimento da primeira versão operacional de Java levou 18 meses. Este primeiro protótipo foi inicialmente chamado "Oak", mas foi rebatizado "Java" em 1995.

1. Introdução

Entre a implementação inicial de Oak, no outono de 1992 e o anúncio público na primavera de 1995, muitas outras pessoas contribuiram para o projeto e a evolução da Linguagem. Bill Joy, Arthur van Hoff, Jonathan Payane, Frank Yellin e Tim Lindholm foram contribuidores chave para o amadurecimento do protótipo original."

Em 20 de abril de 2009, a ORACLE comprou a SUN e passou a ser a detentora dos direitos de Java.

1. Introdução

Pode-se supor que eles desejavam "consertar" algumas características de C++ e introduzir outras.

A esta altura, a equipe e o projeto ainda não estavam voltados para a WWW.

De qualquer modo, hoje é sabido que a equipe desejava atingir um bom compromisso de projeto que produzisse:

uma linguagem fácil de utilizar

uma linguagem poderosa

uma linguagem flexível

1. Introdução

A tônica da abordagem parece ter sido: “quando em dúvida, deixe fora da linguagem”, isto é, antes de tudo, produzir uma linguagem simples.

No final, pode-se dizer que eles chegaram a uma linguagem com as seguintes características:

simples, interpretada, porém com bom desempenho, portável, independente da arquitetura hospedeira, orientada a objetos, multitarefa, distribuída, robusta, segura, com ligação dinâmica, com coleta de lixo automática, fortemente tipada.

1. Introdução

Outras características importantes de Java são:

- possui um conjunto de tipos (básicos e classes) concebido para uma linguagem de alto nível (arrays, strings, booleanos),
- não possui ponteiros para uso pelo programador,
- possui uma interface com métodos nativos.

1. Introdução

A decisão por uma linguagem interpretada deveu-se a duas razões principais:

- diminuir o tempo de desenvolvimento de sistemas, acelerando o ciclo compilar-ligar-carregar-testar-falhar-depurar;

- obter portabilidade imediata.

1. Introdução

O Modelo de Execução

Java é uma linguagem interpretada. Isto significa que o código gerado pelo compilador java (*javac*) **não é** o código de máquina do processador da máquina hospedeira, mas sim um código intermediário, conhecido como ***byte code***.

1. Introdução

Este código vai ser interpretado por um programa que funciona como se fosse um processador cuja linguagem de máquina fosse o ***byte code***.

Este interpretador é conhecido como a “máquina virtual java” (JVM - Java Virtual Machine) e o programa chama-se *java* ou *java.exe*.

1. Introdução

Por isto, cada tipo de computador precisa ter um interpretador próprio. Entretanto, todos os interpretadores vão interpretar o mesmo código (***byte code***) gerado pelo compilador.

É o interpretador java que acessa o sistema operacional nativo para permitir que o programa que está sendo interpretado utilize a interface gráfica, o teclado, o mouse, os arquivos, os dispositivos de acesso à rede, etc.

O interpretador java deve fazer isto de uma maneira consistente em todas as plataformas.

2. O ambiente jdk-se

Vamos analisar o ambiente de execução da própria ORACLE conhecido como JDK (Java Development Kit).

A versão distribuída pela ORACLE quando estas notas foram escritas é a:

"Java SE Development Kit 8u40"

(jdk-8u40)

2. O ambiente jdk-se

Este ambiente pode ser obtido gratuitamente (ler o *copyright* !) de um *site* da ORACLE na Internet:

`www.oracle.com/technetwork/java/javase/downloads/index.html`

Este *site* possui várias versões para as plataformas mais populares, tais como Microsoft Windows (9X, NT, etc), Apple MacIntosh e vários tipos de Unix e Linux.

2. O ambiente jdk-se

IMPORTANTE!

Você só deve executar classes com uma JVM da mesma versão do compilador utilizado, isto é:

-Se o compilador usado (`javac –version`) foi da versão 8, a JVM só pode ser da versão 8.

- Tentar executar estas classes com uma JVM da versão 7 vai gerar um erro de execução!

2. O ambiente jdk-se

A instalação do JDK cria uma subárvore no diretório escolhido (.../java ou outro nome) com vários subdiretórios:

```
bin, demo, include, include-old, jre, lib, src
```

Em .../java/bin ficam os executáveis tais como javac, java, jar, javadoc, appletviewer e outros.

Normalmente, coloca-se este caminho na variável PATH (do shell que for usado).

2. O ambiente jdk-se

No caso do MS-Windows, poderíamos ter:

```
set PATH=c:\PROGRA~1\java\jdk1.8.0_40\bin;%PATH%
```

No caso do Unix/Linux com bash, poderíamos ter:

```
PATH=/usr/bin/java/jdk1.8.0_40/bin:$PATH
```

2. O ambiente jdk-se

Uma outra variável essencial é a variável CLASSPATH, que deve conter os caminhos de onde o interpretador Java vai carregar os arquivos das classes necessárias.

As classes padrão do JDK estão em `.../lib/rt.jar` e o compilador já as procura automaticamente.

Normalmente coloca-se o diretório atual (.) e alguma outra localização que possua classes úteis:

```
set CLASSPATH=c:\minhas_classes
```

2. O ambiente jdk-se

Um Exemplo

Os programas fonte são frequentemente editados em editores de texto comuns (ASCII puro) e devem possuir a terminação ***.java*** .

Vamos então supor, por exemplo, um programa fonte chamado:

`EleEscreve.java`

2. O ambiente jdk-se

O arquivo deve ter o mesmo nome da única classe pública declarada (adicionado da terminação ***.java***).

Os nomes de classes devem começar com uma letra maiúscula (embora isto não seja obrigatório) :

```
public class EleEscreve{
   public static void main(String[] args){
      System.out.println(”Ele escreve !”);
   }
}
```

2. O ambiente jdk-se

O compilador, chamado ***javac*** ou ***javac.exe***, invocado através de um interpretador de comandos (*shell*), vai gerar um arquivo compilado para cada classe existente no arquivo fonte.

Então, teríamos:

```
$ javac EleEscreve.java
```

2. O ambiente jdk-se

No nosso exemplo, ele geraria um arquivo de nome **EleEscreve.class**, com o código intermediário conhecido como *bytecode*.

É este código intermediário que vai ser interpretado.

2. O ambiente jdk-se

O interpretador, chamado ***java*** ou ***java.exe***, também é chamado através do interpretador de comandos.

Ele recebe como parâmetro de linha de comando o nome da classe (no nosso exemplo, o nome do arquivo com o código intermediário, mas agora sem a terminação ***.class*** !)

2. O ambiente jdk-se

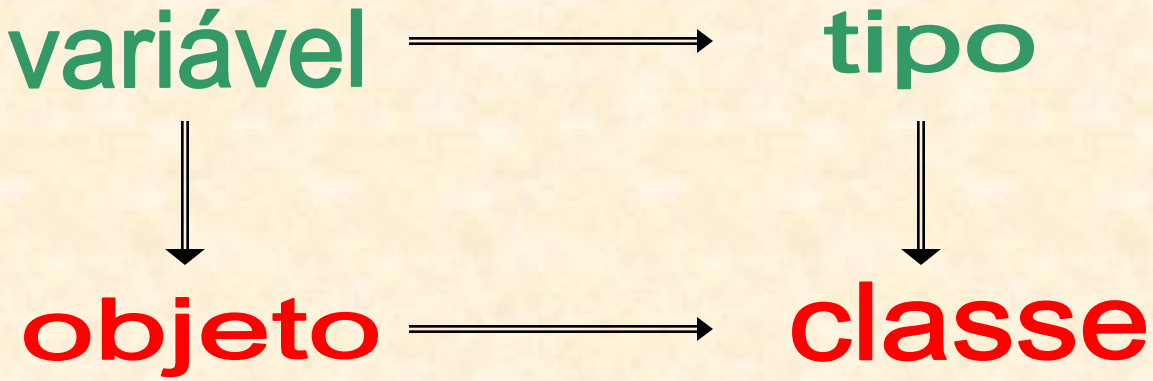
No nosso caso teríamos:

```
$ java EleEscreve
Ele escreve !
$
```

3. A linguagem básica

Java é uma linguagem orientada a objetos e, por isto, vamos começar com os conceitos básicos de orientação a objetos.

Os conceitos de **objetos** e **classes** podem ser vistos, respectivamente, como extensões dos conceitos de **variáveis** e **tipos**.



3. A linguagem básica

Uma **variável** é, antes de tudo, simplesmente um nome.

Associada a cada variável, ainda existem:

(i) um conjunto de **valores**, chamado a **sorte** da variável;

(ii) uma função chamada **estado** da variável (ou atribuição), que mapeia cada variável do programa em um valor da sua sorte;

(iii) um conjunto de funções, ou **operadores**, que mapeiam valores das variáveis, e outros valores, em um outro conjunto de valores.

3. A linguagem básica

Exemplo:

Seja uma **variável** de nome `contador`.

Suponhamos que a **sorte** desta variável seja o conjunto dos números inteiros de

-2.147.483.648 a +2.147.483.647 .

Suponhamos que os **operadores** associados a esta variável sejam:

a soma (+), a subtração (-), a multiplicação(*), a divisão inteira (/), o resto (%), a troca de sinal (-), etc.

Humm…, acho que já vi isto antes.

3. A linguagem básica

Imaginemos agora o conjunto de todas as variáveis, possíveis e imagináveis, com a mesma sorte de `contador` e com os mesmos operadores associados.

Este conjunto, **infinito**, é chamado de **tipo** das variáveis correspondentes.

Em C, C++ e **Java**, o tipo do exemplo é o nosso velho e conhecido tipo `int` (ou long).

3. A linguagem básica

Quando declaramos uma variável:

```
int contador;
```

estamos dizendo que escolhemos, ou **instanciamos**, uma **variável** com este nome do conjunto infinito de variáveis que é o **tipo** `int`.

Em geral, a maioria das linguagens não permite que, para um dado programa, se escolham variáveis com o mesmo nome de uma outra já escolhida, em situações aonde possa haver ambiguidade.

3. A linguagem básica

Um **objeto**, ao contrário de uma variável, não tem nome. **Os objetos são anônimos !**

Um **objeto** é um conjunto que possui basicamente as seguintes espécies de membros:

- variáveis (também chamadas de campos / *fields*);

- métodos, que são funções definidas pelo programador (correspondentes aos operadores das variáveis);

3. A linguagem básica

Assim como um tipo é um conjunto infinito de variáveis,

uma classe é um conjunto infinito de objetos com as mesmas variáveis e métodos.

A **declaração** de uma classe (não confundir com o conceito abstrato da classe propriamente dita) pode ser vista como um gabarito, modelo ou descrição dos objetos que a constituem.

No jargão do dia a dia, confundem-se os termos classe e declaração de classe...

3. A linguagem básica

Em Java, devido a flagrante semelhança entre o que acabamos de chamar tipos de variáveis e classes de objetos,

***as classes também são chamadas de tipos*.**

Temos então oito (8) **tipos primitivos** de variáveis:

`boolean byte short char int long float double`

e os **tipos criados pelo programador**:

classes, interfaces e arrays

3. A linguagem básica

Além disto, Java possui um outro recurso que poderíamos chamar de quase-tipo, os

ENUM

Um Enum pode ser chamado de quase-tipo porque é um conjunto **finito** de objetos.

Enums serão vistos no capítulo 5.

3. A linguagem básica

Além disto, existem:

- regras que vão controlar a visibilidade externa de uma classe (ou objeto) e seus componentes, ou regras de **encapsulamento**;
- regras que vão permitir definir outras classes a partir de uma classe, ou regras de **herança**;
- regras que vão permitir utilizar o mesmo nome para objetos diferentes sem que haja ambigüidade, ou para relaxar os requisitos de tipagem forte, as regras de **polimorfismo**.

3. A linguagem básica

Vejamos o seguinte exemplo:

```
class Point{
   double x,y,z;
}
```

`Point` agora é um novo tipo, criado pelo programador.

3. A linguagem básica

Como os objetos não têm um nome *a priori*, eles vão ser identificados e manipulados através o nono tipo de variável,

as **variáveis de referência**.

Por exemplo:

```
Point p1, p2;
```

3. A linguagem básica

A declaração de uma variável de referência, ou simplesmente **referência**, **não** instancia um objeto.

Um objeto tem que ser instanciado explicitamente, seja separadamente ou em conjunto com a declaração da referência.

3. A linguagem básica

Por exemplo, um objeto da classe `Point` poderia ser **instanciado** da forma abaixo:

```
p2 = new Point();
```

A declaração da referência e a instanciação do objeto também poderiam ser feitas em conjunto:

```
Point p3 = new Point();
```

3. A linguagem básica

Observação: Os exemplos mostrados até aqui foram simplesmente esquemáticos. Não foram utilizados modificadores como `public` **ou** `private`**, como seria mais realístico.**

3. A linguagem básica

Vistos os conceitos mais básicos, podemos ver alguns detalhes a mais.

Os detalhes que vamos ver envolvem os conceitos de:

- packages;
- modificadores de classes, variáveis e métodos;
- herança e interfaces;

3. A linguagem básica : *packages*

Pacotes, ou **packages**, são conjuntos de declarações de **classes** e **interfaces**.

Todo programa Java é organizado como um conjunto de pacotes.

A maior parte dos sistemas armazena os pacotes em sistemas de arquivos, mas também há previsão para que os pacotes sejam armazenados em outros tipos de bases de dados.

3. A linguagem básica : *packages*

Um pacote, por sua vez, é composto por *unidades de compilação*, que no nosso caso são simplesmente os arquivos com o código fonte java.

Cada arquivo que pertence ao mesmo pacote deve obrigatoriamente começar com a declaração *package*:

```
package meuExemplo;
...
```

3. A linguagem básica : *packages*

Vejamos o seguinte exemplo:

```
package meuExemplo;
public class MinhaClasse{
   public static int contador;
   public static void main
                     (String[] args){
   …
   }
}
```

3. A linguagem básica : *packages*

Isto significa que todos os nomes declarados em unidades deste pacote devem ser precedidos do nome do próprio pacote, separado por ponto.

Os nomes completos declarados neste arquivo são:

```
meuExemplo.MinhaClasse
meuExemplo.MinhaClasse.contador
meuExemplo.MinhaClasse.main()
```

3. A linguagem básica : *packages*

Se precisarmos referenciar a variável contador num método de uma outra classe, o correto seria escrever:

```
package outroPacote;
public class Outra{
   void metodoA(){
meuExemplo.MinhaClasse.contador++;
   }
}
```

3. A linguagem básica : *packages*

Mas, se não quisermos escrever o nome completo, podemos usar a declaração **import**:

```
package outroPacote;
import meuExemplo.*;
public class Outra{
  void metodoA(){
      MinhaClasse.contador++;
  }
}
```

3. A linguagem básica : *packages*

Supomos que a variável de ambiente CLASSPATH possua a localização do diretório onde estão as classes do pacote `meuExemplo`.

Por exemplo:

```
CLASSPATH=/home/leao/javaprogs/
```

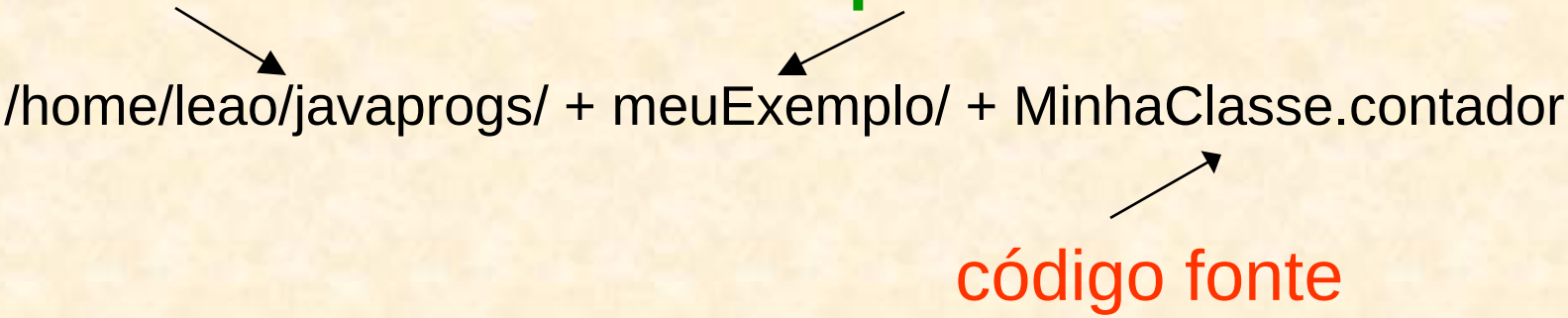
Neste diretório estariam subdiretórios com os vários pacotes existentes, inclusive o diretório do pacote `meuExemplo`.

3. A linguagem básica : *packages*

Quando o `metodoA` da classe `Outra` do pacote `outroPacote` começasse a execução e precisasse da classe `meuExemplo.MinhaClasse`, ele usaria a variável CLASSPATH para procurar no diretório `/home/leao/javaprogs/` , concatenado com o nome do diretório `meuExemplo` , um arquivo chamado `MinhaClasse.class`, que pertencesse ao pacote `meuExemplo`.

3. A linguagem básica : *packages*

CLASSPATH **import**



Nome completo:

meuExemplo.MinhaClasse.contador

Localização completa:

/home/leao/javaprogs/meuExemplo/MinhaClasse.class

3. A linguagem básica : *packages*

Podemos ver que a declaração import **não** importa nada, no sentido de trazer uma biblioteca, ligar ou carregar o que quer que seja !

A única coisa que o import faz é permitir que o programador não tenha que escrever o nome completo em todas as ocorrências do nome no texto do programa fonte.

Isto, desde que não haja ambigüidade nos nomes dos tipos, métodos e variáveis.

3. A linguagem básica : *packages*

Se um arquivo fonte (unidade de compilação) não possuir uma declaração de *package*, ele passa automaticamente a pertencer a um pacote *default* sem nome.

Os pacotes também podem possuir uma estrutura hierárquica como a dos diretórios.

Assim sendo, um pacote pode possuir subpacotes, que são mapeados em subdiretórios do pacote pai.

3. A linguagem básica : *packages*

A SUN chega mesmo a sugerir que as pessoas utilizem o sistema de nomes da Internet, na ordem invertida, para criar nomes inconfundíveis.

Por exemplo, o

```
package BR.UFRJ.GTA.simul;
```

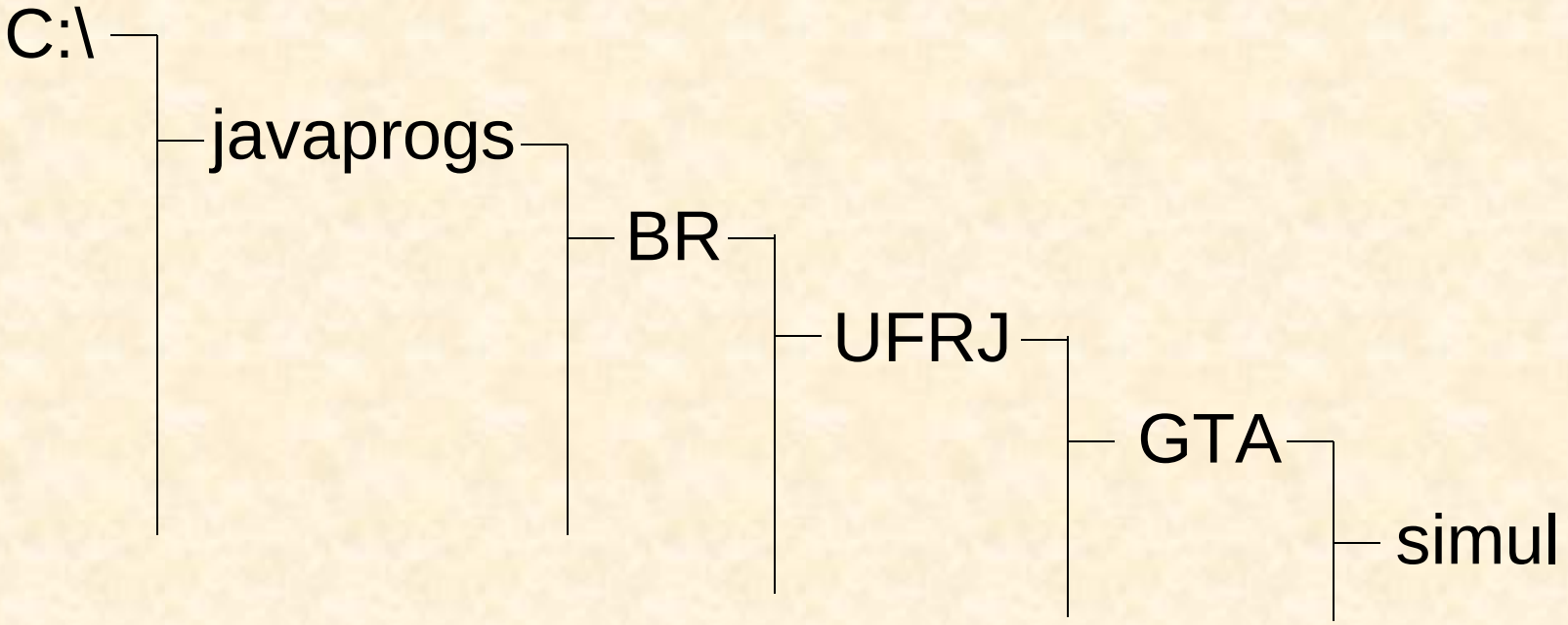
estaria mapeado em uma estrutura de subdiretórios abaixo de um diretório constante na variável CLASSPATH.

3. A linguagem básica : *packages*

Supondo a variável abaixo,

```
CLASSPATH=C:\javaprogs
```

teríamos:



3. A linguagem básica : *modifiers*

Os modificadores de classes e de interfaces permitem definir a visibilidade e as possibilidades de estender classes.

Se uma classe ou uma interface são declaradas **public**, elas são acessíveis por todo o código que pode acessar o pacote onde elas foram declaradas.

Se elas não foram declaradas **public**, elas só podem ser acessadas por código do próprio pacote onde elas foram declaradas.

3. A linguagem básica : *modifiers*

Além disto, só pode haver uma classe **public** em cada arquivo.

Uma classe declarada **final** não pode ser estendida, isto é, não é possível declara uma nova classe que estenda uma classe **final**.

Uma classe declarada **abstract**, por outro lado, nunca pode ter um objeto instanciado. Ela serve para ser a classe pai de outras que vão ser estendidas a partir dela.

3. A linguagem básica : *modifiers*

Variáveis declaradas **public** são visíveis por todo o código da aplicação (desde que se use o nome completo, claro !) e são sempre herdadas. Podem ser chamadas de variáveis **"globais"** e, por isto, devem ser usadas com consciência.

Variáveis declaradas **private** estão no lado oposto do espectro: só são visíveis pelo código da própria classe, não são visíveis nem pelo código de subclasses da sua classe e nem são herdadas ! Elas estão completamente encapsuladas, sob todos os aspectos.

3. A linguagem básica : *modifiers*

Se uma variável é declarada sem um modificador de visibilidade, ela é tomada como do tipo ***default*** (embora este modificador não exista de maneira explícita).

Variáveis ***default*** são visíveis pelas classes do próprio pacote e por subclasses que pertençam ao próprio pacote, mas elas **não** são visíveis **nem** são herdadas por subclasses que pertençam a outros pacotes.

3. A linguagem básica : *modifiers*

Variáveis declaradas **protected** são semelhantes às variáveis ***default***, mas ao contrário destas, elas são herdadas por subclasses que pertençam inclusive a outros pacotes.

Variáveis também podem ser declaradas **final**, o que significa que, uma vez iniciadas, não podem mais ser alteradas e, para todos os efeitos, são constantes.

3. A linguagem básica : *modifiers*

Podemos então resumir estas regras na tabela abaixo:

Modificadores	Acesso intrapacote	Herança interpacotes	Acesso interpacotes
public	**S**	**S**	**S**
protected	**S**	**S**	**N**
default	**S**	**N**	**N**
private	**N**	**N**	**N**

(desenho da tabela por Jorge L.S. Leão)

3. A linguagem básica : *modifiers*

Pergunta: Se uma classe é um conjunto de objetos com a mesma estrutura, isto é, variáveis e métodos, será que as variáveis são realmente as mesmas ou são clones independentes ?

Resposta: Existem os dois tipos, e isto pode ser escolhido pelo programador.

Se uma variável é declarada **static**, ela é uma só para todos os objetos da classe. Qualquer modificação feita por um código será visível por todo código que acessa-la após.

3. A linguagem básica : *modifiers*

Diz-se então que as variáveis **static** são **“variáveis de classe”**.

Se uma variável não é declarada static, então ela é dita uma **“variável de instância”**, e cada objeto possui na verdade uma instância independente desta variável.

3. A linguagem básica : *modifiers*

Métodos também podem ser declarados com o uso de modificadores.

No tocante à visibilidade, o significado é basicamente o mesmo.

Um método declarado **static**, por outro lado, só pode atuar sobre a classe e não pode ser usado para atuar sobre uma instância da classe.

3. A linguagem básica : *herança*

Pergunta: Porque a abordagem orientada a objetos é vantajosa em relação a abordagem procedural tradicional ?

Resposta: Provavelmente, a maior vantagem vem da possibilidade de se criar **hierarquias de classes** baseadas em relações escolhidas pelo programador.

3. A linguagem básica : *herança*

Exemplo:

Vamos supor que estamos projetando um sistema de simulação de movimento de trens.

No topo da hierarquia, podemos supor que existe uma classe de objetos genéricos, não específicos, chamada *Object*.

Abaixo de *Object*, vamos criar uma classe que é uma extensão desta, chamada *MaterialRodante*.

3. A linguagem básica : *herança*

Abaixo desta, vamos criar outras classes chamadas *TremDePassageiros*, *TremDeMinerio*, *TremDeCargaVariada*, *VagoneteDeServico*, etc

A classe *MaterialRodante* possui algumas propriedades (variáveis e métodos) que *Object* não possui. Mas *MaterialRodante* **é-um** (***is-a***) *Object* .

Da mesma forma, *TremDePassageiros*, *TremDeMinerio*, *TremDeCargaVariada* e *VagoneteDeServico* **são** (***is-a***) *MaterialRodante* .

3. A linguagem básica : *herança*

Além da relação **é-um** (***is-a***), também são muito comuns relações como **tem-um** (***has-a***), **faz-um** (***makes-a***), **usa-um** (***uses-a***) e outras.

Os mecanismos disponíveis em Java para criar estas hierarquias de classes são:

- **estender** uma classe e
- **implementar** uma ou mais **interfaces**.

Estes mecanismos também são conhecidos como mecanismos de herança.

3. A linguagem básica : *herança*

Como já foi mencionado, existe uma classe que está no topo de todas as hierarquias de classes: a classe **`java.lang.Object`** .

Esta classe possui vários métodos que são herdados por todas as classes que a estendem.

A maneira de definir novas classes é usando a palavra reservada `extends` .

3. A linguagem básica : *herança*

Exemplo:

```
class MaterialRodante extends Object{
. . .
}

class TremDeMinerio extends MaterialRodante{
. . .
}
```

3. A linguagem básica : *herança*

Antes de tudo, não é necessário escrever que uma classe estende *Object*.

Assim, seria mais normal e subentendido, escrever:

```
class MaterialRodante{
. . .
}
```

Um outro ponto importante é que uma classe só pode estender **uma única classe**, isto é, Java só possui **herança simples** (para classes).

3. A linguagem básica : *herança*

Quanto às variáveis e métodos da nova classe (subclasse), ela possui aqueles da superclasse que puderem ser herdados, dependendo dos modificadores public, private, protected e o default.

Na declaração da nova classe, o programador pode acrescentar novas variáveis e novos métodos,

3. A linguagem básica : *herança*

Ele pode inclusive utilizar os mesmos nomes, o que dá origem às situações de:

- sobrecarga / *overloading* (de métodos)
- sobreposição ou substituição / *overriding* (de métodos), e
- ocultação / *hiding* (de campos)

A palavra reservada *super*, é usada como uma referência para as implementações de **métodos não estáticos** de objetos da superclasse da classe atual.

3. A linguagem básica : *herança*

Uma classe que possui pelo menos um método declarado ***abstract***, tem que ser declarada ***abstract***.

Obviamente, uma classe ***abstract*** não pode ter objetos instanciados, mas é usada para declarar subclasses.

3. A linguagem básica : *herança*

O conjunto dos **campos** e **métodos** acessíveis **de fora** de uma classe, junto com a **descrição do comportamento esperado** destes, é conhecido como o **contrato da classe**.

Quando se estende uma superclasse, é considerada uma boa prática **não alterar o contrato** ou **só alterar a implementação** do contrato da superclasse, na subclasse que se está declarando.

3. A linguagem básica : *herança*

Quando se tem uma classe na qual todos os métodos são ***abstract*** temos o que se chama a **expressão pura de um projeto**.

Quer dizer que não se fornece nenhuma implementação para o contrato da classe.

Uma forma de declarar tal tipo de “classe” é como uma ***interface***.

Embora Java só permita herança simples para classes, **para interfaces**, Java permite **herança múltipla**.

3. A linguagem básica : *herança*

Como as interfaces são na verdade um tipo de classe, elas também podem ser estendidas.

Pelo mesmo motivo, também é possível definir uma referência para uma interface.

3. A linguagem básica : *herança*

Exemplos:

```
interface A { void metodo1( ); }
interface B extends A { void metodo2( ); }
interface EstoqueIF { int nro; }
```

```
EstoqueIF v;
```

“ herança múltipla ” :

```
class D implements B,C { . . . }
```

3. A linguagem básica

Na verdade, a própria declaração da classe poderia ser feita em conjunto com a declaração da referência e a instanciação do objeto, dispensando então o uso de um nome para a classe (**classe anônima**):

```
Point p4 = new Point(){
        boolean fixado = true;
  }
```

3. A linguagem básica

O objeto referenciado por `p4` pertence a uma classe anônima que descende da, ou **estende a classe** `Point`, acrescentando à primeira a variável `fixado`.

O acesso a variáveis e métodos de uma classe ou objeto é feito através a **notação pontuada** tradicicional:

```
p4.x = 1.0;
p4.y = 1.2;
p4.z = 1.4;
```

3. A linguagem básica : *procedural*

A parte léxica de Java:

A maior parte das linguagens de programação utiliza como conjunto de caracteres o ASCII e suas extensões, como o Latin-1.

Java utiliza o conjunto Unicode, um conjunto de caracteres de 16 bits que suporta línguas e alfabetos, como o cirílico, o árabe, o chinês e outras.

Em particular, o conjunto de carcteres ASCII é um subconjunto do Unicode.

3. A linguagem básica : *procedural*

Além dos lexemas, o compilador Java reconhece alguns caracteres e conjuntos de carcteres que têm função de espaçadores (white space) ou de comentários.

Os espaçadores são: space, newline, tab, ff, …

Segundo alguns autores (!), os comentários são a parte mais importante de uma linguagem de programação.

3. A linguagem básica : *procedural*

Quem tem dúvida disto, deve tentar entender um código em uma linguagem que não se usa mais, escrito a 30 anos atrás, antes que o “bug do milênio” faça a empresa ir à falência…

Em Java, há três formas de comentários:

1) A forma que começa com “//” em qualquer lugar de uma linha e acaba no final da linha.

Exemplo;

```
int nform;  // contador de formularios recebidos
// atualizado a cada recepcao de um formulario
```

3. A linguagem básica : *procedural*

2) A forma que começa com "`/*`" em qualquer lugar de uma linha e termina com "`*/`" em qualquer lugar da mesma ou de outra linha.

Exemplo:

```
/* Nome do arquivo: FormAval.java
   Funcao: applet da IFHM do form de avaliacao
   Autor: JLSL                                  */
public class FormAval { . . .
```

3. A linguagem básica : *procedural*

3) A forma chamada de “comentários doc”, usada por um programa utilitário (javadoc) para gerar uma documentação padronizada para as classes, interfaces, campos e métodos, extraída de comentários semelhantes à segunda forma, mas que começam com “`/**`”.

Estes comentários devem preceder as declarações de classes, interfaces, campos e métodos.

3. A linguagem básica : *procedural*

Exemplo:

```
/** Classe principal da aplicacao de
*   formularios de avaliacao
*/
public class FormAval {

/** Metodo principal da aplicacao */
public static void main(String[] args){ . . .
```

Observação: Os três tipos de comentários não podem ser ainhados ou sobrepostos.

3. A linguagem básica : *procedural*

Os lexemas (*tokens*) de Java podem ser classificados nos seguintes conjuntos:

- palavras reservadas
- separadores (operadores, pontuação)
- literais
- identificadores

3. A linguagem básica : *procedural*

O analisador léxico de Java é do tipo “guloso”, isto é, havendo mais de uma posssibilidade, ele sempre vai procurar identificar o maior lexema.

Exemplo: A expressão `A +++++ B;`

vai ser interpretada como

`A ++ ++ + B;` que é inválida,

ao invés de

`A ++ + ++ B;` que seria válida !

3. A linguagem básica : *procedural*

Identificadores

Um identificador começa com uma letra, um cifrão ($) ou uma sublinha (_ , *underscore*). Os caracteres seguintes do identificador podem ser letras, dígitos (0..9), sublinhas ou cifrões.

Um identificador deve estar separado de outros lexemas por espaçadores (que **não** são lexemas) ou por separadores (que **são** lexemas),

É bom observar que, como Java usa Unicode, o conjunto das letras é bem maior do que simplemente a..z e A..Z.

3. A linguagem básica : *procedural*

Os identificadores são definidos pelo programador e usados para dar nomes às classes, interfaces, variáveis e métodos.

Exemplos:

v1 , v2 , _x1 , thisCase , ifOpened , π

3. A linguagem básica : *procedural*

Palavras reservadas

Alguns lexemas que se encaixariam na definição de identificadores são na verdade palavras reservadas para dar nome a construções da linguagem e **não** podem ser usados pelo programador como identificadores.

3. A linguagem básica : *procedural*

São elas (50):

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

3. A linguagem básica : *procedural*

Separadores

A terceira categoria de lexemas de Java é constituída de caracteres e conjuntos de caracteres que não precisam obrigatoriamente estar separados dos outros lexemas por espaçadores.

São sinais de pontuação e os operadores.

3. A linguagem básica : *procedural*

Os **sinais de pontuação** são:

( ) usados para delimitar argumentos de métodos, o tipo em expressões de conversão de tipo e dar uma precedência em expressões;

[ ] usados para delimitar o índice de um array;

{ } usados para delimitar blocos de instruções;

. usado para separar componentes dos nomes de campos, métodos, classes, interfaces (também usado, mas não como separador em literais de float e double);

, usado para separar os argumentos de um método;

; usado para terminar uma instrução.

3. A linguagem básica : *procedural*

Os **operadores**, que representam funções usadas para construir expressões lógicas e aritméticas, são

Operador	Função	Operador	Função	Operador	Função
=	atribuição	/	divisão	^	ou exclusivo
==	igual	/=	divisão e atrib.	^=	ou ex. e atrib.
+	soma	~	inversão binária	->	expressões lambda
+=	incremento e atrib.	&&	e lógico	--	decremento de um
>	maior	&	e binário	%	resto
>=	maior ou igual	&=	e lógico e atrib.	%=	resto e atrib.
-	subtração	?	if ternário	<<	desloc. à esquerda
-=	decremento e atrib.	||	ou lógico	<<=	desloc. à esq. e atrib.
<	menor	|	ou binário	>>	desloc. à direita
<=	menor ou igual	|=	ou lógico e atrib.	>>=	desloc. à dir. e atrib
*	multiplicação	:	if ternário ou domínio for	>>>	desloc. à direita com zeros
*=	multip. e atrib.	::	referência a método	>>>=	desloc. à dir. com zeros e atrib
!	negação	++	incremento de um	instanceof	compar. de tipo
!=	diferente				

3. A linguagem básica : *procedural*

Literais

Literais são os lexemas que representam os valores que as variáveis de um certo tipo pode assumir, isto é, representam os elementos de uma sorte.

Os literais do tipo `boolean` são: `true` e `false`.

Os literais dos tipos `byte`, `short`, `int` e `long` são numerais que podem ser expressos em octal, decimal ou hexadecimal.

3. A linguagem básica : *procedural*

Caso um literal numérico comece com `0`, ele está representado em octal, caso ele comece com `0x`, ele está representado em hexadecimal e caso ele comece com qualquer outro dígito ele está representado em decimal.

Na falta de outra informação, Java assume por *default* que um literal numérico é do tipo `int`.

Se um literal numérico terminar com a letra `L` (ou `l`), ele será tratado como um `long`.

3. A linguagem básica : *procedural*

Se, por outro lado, o literal estiver dentro dos limites de `short` e for atribuído a uma variável (ou argumento) `short`, então ele será tratado como `short`. A mesma regra se aplica ao tipo `byte`.

Em todos os outros casos, o programador deve usar explicitamente a conversão de tipo (*type casting*) para fazer uma atribuição de um literal `int` a um `short` ou a um `byte`.

3. A linguagem básica : *procedural*

Exemplos:

```
int n = 0xFFFFFFFF;

short m = -1;

if (m == (short)(-1)) ...
```

3. A linguagem básica : *procedural*

Os literais do tipo `float` e `double` são expressos como números decimais com um ponto decimal (opcional) e um expoente (também opcional).

Deve haver pelo menos um dígito em um literal (!).

Exemplos:

```
1               .3E-2
-1              1.0E+3D
.05             1F    (que não se confunde
                      com (0x1F !)
```

3. A linguagem básica : *procedural*

Os literais do tipo char podem ser expressos por um caractere entre apóstrofos: `'A'`

No caso de caracteres do conjunto Unicode que não possam ser usados no editor disponível e no caso de alguns caracteres especiais, usa-se uma sequência de escape (começada por '\').

Algumas das sequências são:

\udddd caractere Unicode, onde dddd é a representação hexadecimal (sem 0x) do código correspondente;

3. A linguagem básica : *procedural*

\ddd caractere ASCII (ddd é a representação octal, sem o 0, menor ou igual a 377, ou seja, 0x00FF);

\b backspace (\u0008)

\t tab (\u0009)

\n newline (LF ou linefeed, \u000A)

\f formfeed (\u000C)

\r return (CR, carriage return, \u000D)

\” double quote (aspas, \u0022)

\’ single quote (apóstrofo, \u0027)

\\ backslah (o próprio, \u005C)

3. A linguagem básica : *procedural*

Detalhando um pouco mais os tipos básicos,

Tipo	Sorte	Valor mínimo	Valor máximo	Valor inicial default
boolean	Valores lógicos	true	false	false
byte	Inteiros de 8 bits	-128	+127	0
short	Inteiros de 16 bits	-32.768	+32.767	0
char	Caracteres codificados em Unicode de 16 bits	\u0000	\uFFFF	\u0000
int	Inteiros de 32 bits	-2.147.483.648	+2.147.483.647	0
long	Inteiros de 64 bits	-9.223.372.036.854.775.808	+9.223.372.036.854.775.807	0
float	Pto. Flut. de 32 bits	NEGATIVE_INFINITY	POSITIVE_INFINITY	0
double	Pto. Flut. de 64 bits	NEGATIVE_INFINITY	POSITIVE_INFINITY	0

3. A linguagem básica : *procedural*

As declarações

A forma mais simples de declaração de variáveis é muito semelhante à da linguagem C e é dada pelo seguinte esquema:

<tipo da variável> <nome da variável> ;

A variável também pode ser iniciada, com um valor diferente do *default*, na sua declaração:

<tipo da variável> <nome da variável> = <valor>;

3. A linguagem básica : *procedural*

Exemplos:

```
int n;
float x,y;
double raio = 10.0;
char letra = ‘A’;
```

3. A linguagem básica : *procedural*

Um ***array***, tipo indexado, é na verdade um tipo especial de objeto, cuja sintaxe se assemelha à de arrays em outras linguagens.

A declaração de uma referência para um array pode ser feita de duas maneiras:

<tipo da variável> [ ] <nome da variável> ;

ou

<tipo da variável> <nome da variável> [ ] ;

Além disto, pode-se declarar um ou mais índices.

3. A linguagem básica : *procedural*

Exemplo:

```
float[][] matriz;
```
(forma preferida)

ou

```
float matriz [][];
```

Observe que o tamanho do array não é declarado. Este só vai ser definido quando da instanciação do array propriamente dito:

```
matriz = new float[20][20];
```

3. A linguagem básica : *procedural*

As instruções

Uma instrução simples em Java é **terminada** por um “ ; ” (ponto-e-vírgula). Outras linguagens, como Pascal, usam o “ ; ” para **separar**, e não para terminar as instruções.

Por outro lado, as chaves ({ }) formam uma instrução composta, ou bloco, e portanto não são terminadas por um “ ; ” .

Podemos dizer que existem 8 instruções em Java, sujeitas a variações.

3. A linguagem básica : *procedural*

As instruções são:

- atribuição
- if-else
- switch
- for
- while
- label-break-continue
- return
- try-catch-finally

3. A linguagem básica : *procedural*

atribuição:

Tem a forma do esquema abaixo

```
<variável> <operador> <expressão> ;
```

ou

```
<variável> <operador> ;
```

ou

```
<operador> <variável> ;
```

3. A linguagem básica : *procedural*

Exemplos:

```
n = n+1;
perímetro = 2.0 * PI * raio;
total += deposito;
saldo -= saque;
contagem++;
```

3. A linguagem básica : *procedural*

if-else:

A instrução if segue o seguinte esquema:

```
if ( <expressão lógica> ) <instrução>
```

ou

```
if ( <expressão lógica> ) <instrução>
   else <instrução>
```

3. A linguagem básica : *procedural*

Exemplo:

Pode-se aninhar várias construções if, como no exemplo abaixo:

```
if (cond1) {
   . . .
} else if (cond2) {
   . . .
} else if (cond3) {
   . . .
} else { . . . }
```

3. A linguagem básica : *procedural*

switch:

Segue o seguinte esquema:

```
switch ( <seletor> ) {
   case <valor> : <instrução>
   . . .
   case <valor> : <instrução>
   default : <instrução>
}
```

3. A linguagem básica : *procedural*

Exemplo:

```
switch ( digitoHex ) {
   case ’1’: case ’2’: case ’3’: case ’4’:
   case ’5’: case ’6’: case ’7’: case ’8’:
   case ’9’: case ’0’: {
      n = (int)(digitoHex - ’0’); break; }
   case ’A’: case ’B’: case ’C’: case ’D’:
   case ’E’: case ’F’:
      n = (int)(digitoHex - ’A’);
}
```

3. A linguagem básica : *procedural*

while:

Segue o seguinte esquema:

```
while ( <expressão booleana> )
     <instrução>
ou
do <instrução> while
     ( <expressão booleana> )
```

3. A linguagem básica : *procedural*

Exemplos:

```
while ( n < 10){
   n++;
   . . .
}

do {
   . . .
   n++;
} while ( n < 9 );
```

3. A linguagem básica : *procedural*

for:

Segue o seguinte esquema:

```
for (<iniciação> ; <cond_iteração> ;
     <atualização>)    <instrução>
```

Exemplo:

```
for (int i=1; i<=10 ; i++) {. . .}
```

3. A linguagem básica : *procedural*

label-break-continue:

Java possui um conjunto de recursos para controlar a saída de blocos de instruções ou da iteração em laços de repetição de uma maneira estruturada.

Inicialmente, qualquer instrução pode receber um rótulo (label). Por exemplo:

```
coluna:for (int col = 1; col<=10; col++)
 linha:for (int lin = 1; lin<=10; lin++)
          a[lin,col] /= a[col,col];
```

3. A linguagem básica : *procedural*

O comando `break` permite saltar as instruções até o final do bloco. Se tratar-se de um laço, a sua execução é terminada e, de qualquer modo, a execução sai do bloco.

```
coluna:for (int col = 1; col<=10; col++)
 linha:for (int lin = 1; lin<=10; lin++)
     if(a[col,col]==0)
          break linha;
     else
          a[lin,col] /= a[col,col];
```

3. A linguagem básica : *procedural*

O comando `continue` permite saltar as instruções até o final do laço e sua execução continua, dependendo do próprio laço.

```
coluna:for (int col = 1; col<=10; col++)
 linha:for (int lin = 1; lin<=10; lin++)
     if(a[col,col]==0)
          continue coluna;
     else
          a[lin,col] /= a[col,col];
```

3. A linguagem básica : *procedural*

return:

O comando `return` termina a execução de um método, retornando-a ao método chamador. Eventualmente, a instrução pode retornar o valor calculado, do tipo apropriado.

```
public boolean eh_par(int n){
  if((n % 2) == 0) return true;
  else return false;
}
```

3. A linguagem básica : *procedural*

goto:

Embora Java possua a palavra reservada `goto`, não existe uma instrução `goto`!

3. A linguagem básica : *procedural*

try-catch-finally:

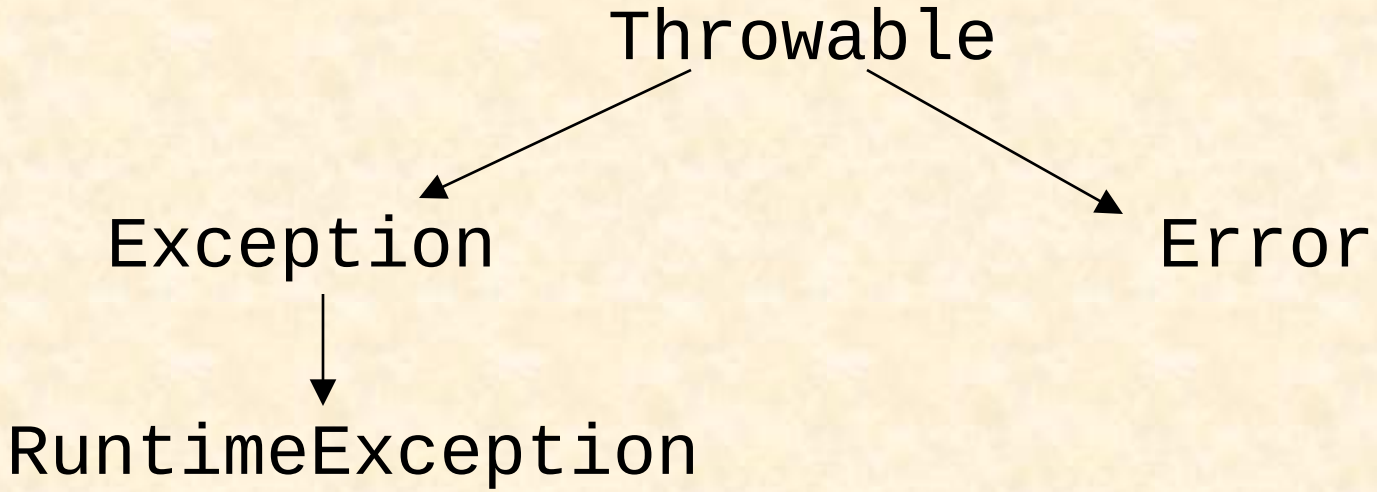
O comando *try-catch-finally* permite dar um tratamento estruturado para condições excepcionais, detetadas pela própria máquina ou pelo código do programador.

O primeiro ponto importante é a existência da classe `java.lang.Throwable`, que possui duas extensões:

`java.lang.Exception` e `java.lang.Error`

3. A linguagem básica : *procedural*

Além disto, existe ainda a classe `java.lang.RuntimeException` que estende `java.lang.Exception`



3. A linguagem básica : *procedural*

A ocorrência de uma condição excepcional vai ser sinalizada pela instanciação e lançamento de um objeto de uma classe que, possivelmente, estende `Throwable` ou, mais frequentemente, `Exception`, `RuntimeException` ou `Error`.

O compilador Java verifica se o código declara e trata a maioria das excessões. São as excessões verificadas (*checked exceptions*).

3. A linguagem básica : *procedural*

Por outro lado, as excessões padrão geradas pelo sistema de execução (*standard runtime exceptions*), que estendem `RuntimeException` ou `Error`, podem acontecer em muito mais pontos do programa e, por isto, não são verificadas pelo compilador (*unchecked exceptions*).

O lançamento de uma excessão é feito pela instrução `throw`.

3. A linguagem básica : *procedural*

Exemplo:

```
class MinhaExcessao extends
        Exception{. . .}
. . .
MinhaExcessao  e;
e = new MinhaExcessao();
throw e;
```

ou, de uma forma mais simples:

```
throw new MinhaExcessao();
```

3. A linguagem básica : *procedural*

Nos dois casos, um objeto, sem nome, é lançado para ser capturado no escopo do próprio `throw` ou relançado novamente para o escopo de código que chamou o método do primeiro escopo.

A captura é feita na estrutura `try-catch-finally`, enquanto o relançamento é declarado pela cláusula `throws`.

3. A linguagem básica : *procedural*

Exemplo:

```
try{
   . . .
   throw new MinhaExcessao();
   . . .
}catch(MinhaExcessao me){
   . . .
}
```

3. A linguagem básica : *procedural*

Se a excessão for efetivamente lançada, a execução é desviada da instrução `throw` para o bloco da cláusula `catch`.

A instância da excessão é passada para este bloco como se fosse um argumento de um método, recebendo o nome formal (neste exemplo, `me`).

Após a execução do bloco `catch`, o programa continua com a próxima instrução (não retorna para o bloco *try* !).

3. A linguagem básica : *procedural*

Opcionalmente, pode haver a cláusula `finally`:

```
try{
   throw new MinhaExcessao;
}catch(MinhaExcessao me){
   . . .
}finally{
   . . .
}
```

A cláusula `finally`, se existir, **sempre** é executada, quer haja a excessão ou não.

3. A linguagem básica : *procedural*

Por outro lado, se o programador não colocar um bloco `try-catch-finally`, ele deve declarar a excessão no método em cujo contexto a excessão pode ser lançada:

```
void metodoA()throws MinhaExcessao{
   . . .
   throw new MinhaExcessao();
   . . .
}
```

3. A linguagem básica : *procedural*

Para todos os trechos em que podem ser lançadas as *checked exceptions*, o compilador Java verifica se o código possui um bloco *try-catch* ou, caso não haja este bloco, ele verifica se o método declara a excessão com a cláusula *throws*.

Neste caso, a excessão vai ser relançada, ou propagada, para o código chamador.

No caso das *unchecked exceptions*, o compilador não obriga o tratamento, mas o programador tem a opção de faze-lo.

3. A linguagem básica : *procedural*

Exemplo:

```
//=========================================
public class TstEx{
//-----------------------------------------
   public static void main(String[] argumento)
                    throws MinhaExcessao{
      throw new MinhaExcessao();
   }
//-----------------------------------------
}
```

(continua)

3. A linguagem básica : *procedural*

(continuação)

```
//=========================================
class MinhaExcessao extends Exception{
//-----------------------------------------
      public MinhaExcessao(){
         super();
      }
//-----------------------------------------
}
//=========================================
```

4. O package java.lang

O pacote ***java.lang*** contem as classes e interfaces que são mais fundamentais para a linguagem Java.

Certamente, as classes mais importantes, que pertencem a este pacote, são:

- a classe **Object**, raiz da hierarquia de classes;

- a classe **Class**, da qual uma instância é criada cada vez que uma classe é carregada pela JVM durante a execução.

4. O package java.lang

É da maior importância compreender a relação que existe entre estas duas classes e o próprio conceito de interpretação de um programa pela JVM.

Esta relação aparece, por exemplo, nos métodos destas duas classes, que se complementam e se ajudam.

4. O package java.lang

O pacote java.lang possui:

- 3 interfaces;
- 30 classes;
- 24 excessões (extensões de Exception e Throwable)
- 21 erros (extensões de Error)

Além disto, ele possui dois subpacotes:

- java.lang.ref
- java.lang.reflect

4. O package java.lang

Vamos começar vendo as seguintes classes:

– **Object** e **Class**

– ***Runnable*** (interface) e **Thread**

– **System** , **Runtime** e **Process**

– **String** e **StringBuffer**

–As *"wrapper classes"* : **Boolean**, **Character**, **Number** e suas extensões (**Integer**, **Long**, **Float** e **Double**)

– **Math**

4. O package java.lang: Object

A classe **Object**:

É a *mãe* de todas as classes. Possui um único construtor e os seguintes métodos (escolhidos):

– getClass

– toString

– hashCode

– equals

– clone

– wait, notify e notifyAll (a serem vistos com a interface Runnable e a classe Thread)

4. O package java.lang : Object

```
public final Class getClass()
```

Como este método não é ***static***, ele só pode ser aplicado a referências. Ele devolve a classe atual da referência.

Ele na verdade devolve uma referência para uma instância da classe ***Class*** da referência em questão. Esta instância é criada quando cada classe é carregada pela JVM.

Para complementar, veja a classe ***Class*** do próprio pacote ***java.lang***.

4. O package java.lang : Object

Resumindo: Quando a JVM carrega uma classe (por exemplo, **MinhaClasse**), ela instancia um objeto da classe Class (por exemplo, **obMinhaClasse**) que vai, daí em diante, representar a classe que foi carregada.

Suponhamos agora uma referência para um objeto da classe **MinhaClasse** que foi carregada: **ob**.

A expressão **ob.getClass()** devolve a referência para a instância **obMinhaClasse**.

4. O package java.lang : Object

Exemplo:

```
public class TstObject{
   static Object ob;
public static void main(String[] argumentos){
      ob = new TstObject();
      System.out.println("A classe eh: ” +
                                 ob.getClass());
   }
}
```

Saída: `A classe eh: class TstObject`

4. O package java.lang : Object

`public String toString()`

Também é um método de instância que devolve um texto que deve representar o objeto de uma forma legível.

No exemplo anterior, o texto fornecido, pré-existente, foi: `class TstObject .`

No caso de um objeto qualquer, o texto pré-definido é: `getClass().getName() + '@' + Integer.toHexString(hashCode())`

Recomenda-se que todas as classes substituam este método.

4. O package java.lang : Object

Exemplo:

```
public class TstObject{
   static Object ob;
   static int serialno;
   int ser;
   public TstObject(){
      ser = serialno++;
   }
   public String toString(){
      return "Serialno do objeto: ” +
                        new Integer(ser).toString();
   }
```

(continua)

4. O package java.lang : Object

(continuação)

```
 public static void main(String[] argumentos){
      System.out.println(new
                         TstObject().toString());
      System.out.println(new
                         TstObject().toString());
 }
}
```

Saída:

```
   Serialno do objeto: 0
   Serialno do objeto: 1
```

4. O package java.lang : Object

```
public int hashCode()
```

É um método de instância que devolve um código *hash* correspondente ao objeto.

Como é normal para os códigos *hash*:

- duas referências para o **mesmo** objeto vão produzir o **mesmo** *hashCode*;

- dois objetos diferentes **podem** possuir o mesmo *hashCode*;

- se duas referências produzem hashCodes diferentes, os objetos **são** diferentes.

4. O package java.lang : Object

`public boolean equals(Object obj)`

É um método de instância que devolve *true* sse o objeto do argumento e o da referência usada com o **`equals()`** (alvo) são "consistentemente equivalentes".

A implementação de **`equals()`** de **Object** devolve true se os "ponteiros" forem os mesmos.

4. O package java.lang : Object

Entretanto, o programador pode (ou mesmo deve) substituir (*override*) este método para atender suas necessidades.

Isto deve ser feito de forma consistente com a substituição de **`clone()`**.

4. O package java.lang : Object

`protected Object clone()`

Para uma classe implementar o método ***clone***, ela também deve implementar a interface ***Cloneable***, caso contrário, a execução de ***clone*** lançará a excessão ***CloneNotSupportedException***

A classe Object **não** implementa o método ***clone*** nem a interface ***Cloneable***.

4. O package java.lang : Object

Exemplo:

```
public class TstClone implements Cloneable{
   static TstClone ob1,ob2;
 public Object clone(){
   return (Object)(new TstClone());
 }
```

(continua)

4. O package java.lang : Object

(continuação)

```
 public static void main(String[] args){
   ob1 = new TstClone();
   ob2 = (TstClone)ob1.clone();
   System.out.println(
    "   Hash ob1: " + ob1.hashCode() + "\n" +
    "   Hash ob2: " + ob2.hashCode() + "\n" +
    "   ob1==ob2: " + ob1.equals(ob2)
   );
 }
}
```

(continua)

4. O package java.lang : Object

(continuação)

A saída do programa é:

```
Hash ob1: 7474923
Hash ob2: 3242435
ob1==ob2: false
```

4. O package java.lang : Class

A classe **Class**:

A classe **Class** **não** possui construtores públicos ! Isto se deve ao fato que somente a JVM pode criar instâncias da classe **Class** a medida que ela vai carregando novas classes e interfaces.

O usuário pode obter uma referência para uma instância de "uma **Class**" com o método ***getClass()*** de **Object**, e depois usar os métodos públicos de **Class**.

4. O package java.lang : Class

Os conceitos envolvidos na classe **Class** são os conceitos mais simples de **reflexão**.

Os principais métodos são:

– toString
– getName
– isInterface
– getSuperclass
– getInterfaces
– newInstance

4. O package java.lang : Class

`public String toString()`

Este método sobrepõe o método ***toString()*** de **Object** para devolver o seguinte:

`(isInterface()?“interface”:“class”)+getName()`

`public String getName()`

Devolve o nome completo da classe, exceto para *arrays*. Neste caso, ele devolve uma codificação que dá o número de dimensões e o tipo básico do *array*.

4. O package java.lang : Class

`public boolean isInterface()`

Devolve *true* se a classe for uma interface e *false* em caso contrário.

`public Class getSuperclass()`

Se a classe da referência em questão não é Object, nem é um *array*, devolve a superclasse desta.

Se a classe da referência for Object ou null, devolve null.

Se a referência for de um *array*, devolve Object.

4. O package java.lang : Class

`public Class[] getInterfaces()`

Se a referência for de uma classe, o array representa as interfaces implementadas pela classe.

Se a referência for uma interface, o array representa as interfaces que estendem diretamente esta interface.

4. O package java.lang : Class

```
public Object newInstance() throws
                    InstantiationException,
                    IllegalAccessException
```

Cria e devolve uma nova instância da classe **Class** representada pela referência.

4. O package java.lang : Thread

A interface **Runnable** e a classe **Thread**:

Java implementa o que é conhecido como programação multitarefa ou concorrente (*multithreaded*).

A criação de novas tarefas, ou *threads*, é feita por métodos nativos na JVM e pode usar recursos da máquina, do sistema operacional ou ser implementada pela própria JVM.

4. O package java.lang : Thread

Qualquer *thread* deve implementar ***Runnable***:

```
public interface Runnable{
   public abstract void run();
}
```

A classe **Thread** é uma classe básica que implementa ***Runnable***. Ela possui vários métodos, dos quais veremos alguns a seguir enquanto outros serão estudados mais adiante.

4. O package java.lang : Thread

A classe **Thread** possui um método que é o responsável por causar o início da execução de um novo *thread* :

```
void start()
```

A execução deste método faz o seguinte:

1. Cria um novo *thread*, que inicia uma execução independente pelo método `void run()` do objeto alvo;
2. Retorna a execução para o *thread* original.

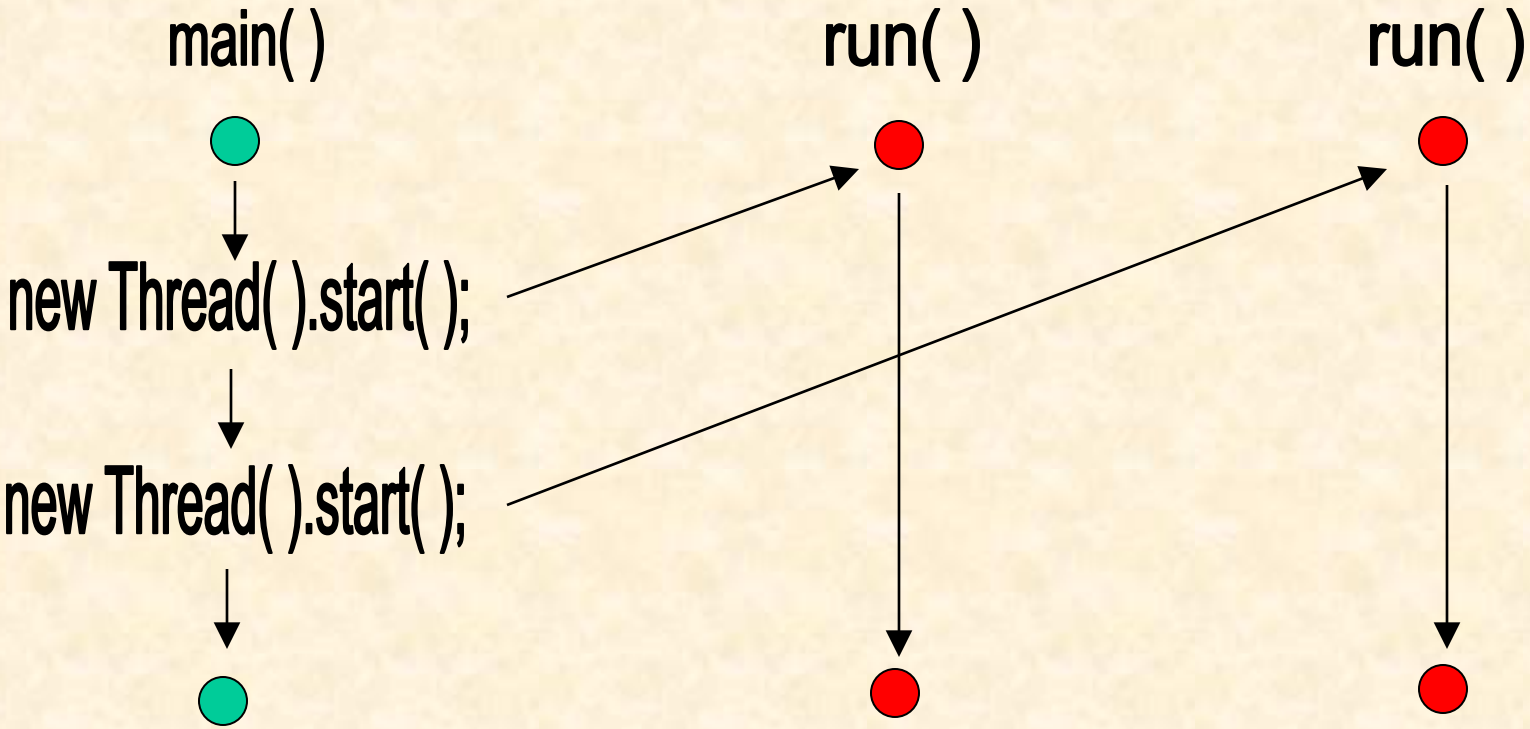
4. O package java.lang : Thread

Exemplo:

```
public class TstThread implements Runnable{
   public static void main(String[] args){
      new Thread(new TstThread()).start();
      new Thread(new TstThread()).start();
   }
   public void run(){
      System.out.println("Thread ” +
           Thread.currentThread().getName() +
                                  " rodando");
   }
}
```

4. O package java.lang : Thread

A execução deste programa pode ser representada pelo seguinte diagrama:



4. O package java.lang : Thread

Alguns outros métodos importantes da classe **Thread** são:

public static Thread currentThread()

public final String getName()

public final boolean isAlive()

public final void join() throws InterruptedException

public static void sleep() throws InterruptedException

public string toString()

public static void yield()

4. O package java.lang : System

A classe **System**:

Esta classe representa, ao menos em parte, a própria execução da JVM.

Ela possui 3 campos:

– in, out e err

Os principais métodos (todos *static*) são:

– currentTimeMillis

– exit

– gc

– arraycopy

– load, loadLibrary

4. O package java.lang : System

`public static InputStream in`

Este campo é uma referência para o **Stream** correspondente à entrada padrão, ou **stdin**, no estilo de sistemas como o Unix.

Isto significa que ele é um **Stream** de bytes (baixo nível ou não formatado), já aberto e que mais freqüentemente é o teclado.

4. O package java.lang : System

`public static PrintStream out`

Este campo é uma referência para o **Stream** correspondente à saída padrão, ou **stdout**, no estilo de sistemas como o Unix.

Isto significa que ele é um **Stream** de bytes (baixo nível ou não formatado), já aberto e que mais freqüentemente é a tela ou uma janela no monitor (texto).

4. O package java.lang : System

`public static PrintStream err`

Este campo é uma referência para o **Stream** correspondente à saída de erros, ou **stderr**, no estilo de sistemas como o Unix.

Isto significa que ele é um **Stream** de bytes (baixo nível ou não formatado), já aberto e que mais freqüentemente é a tela ou uma janela no monitor (texto).

4. O package java.lang : System

```
public static Long
            currentTimeMillis()
```

Devolve o tempo medido em milisegundos desde “the epoch”, 00:00:00 de 01 de janeiro de 1970.

4. O package java.lang : System

```
public static void exit(int status)
        throws SecurityException
```

Este método causa o término da execução da JVM (e conseqüentemente do próprio programa Java), devolvendo o valor de **`status`** ao ambiente (*shell*) que iniciou sua execução.

Um método equivalente existe na classe **Runtime**.

4. O package java.lang : System

```
public static void gc()
```

A chamada deste método sugere à JVM que ela faça a coleta de lixo (*garbage collect*).

Não é assegurado que a JVM vá realmente realiza-la imediatamente.

Um método equivalente existe na classe **Runtime**.

4. O package java.lang : System

```
public static void arraycopy(
    Object src, int srcOffset,
    Object dst, int dstOffset,
    int length) throws
            NullPointerException,
            ArrayStoreException,
            IndexOutOfBoundsException
```

Este é um método utilitário que realiza a cópia dos elementos de um ***array*** para outro.

4. O package java.lang : System

`load e loadLibrary`

Estes métodos servem basicamente para carregar a implementação de ***métodos nativos***, e por isto mesmo, são altamente dependentes da plataforma.

4. O package java.lang : Runtime

A classe **Runtime**:

Esta classe representa, de certa maneira, o sistema hospedeiro.

Os principais métodos (só um *static* !) desta classe são:

- getRunTime
- exit
- exec
- totalMemory
- freeMemory

4. O package java.lang : Runtime

```
public static Runtime getRuntime()
```

Este é o único método *static*.

Ele precisa ser chamado uma única vez para que os outros métodos possam ser aplicados à referência devolvida por este.

4. O package java.lang : Runtime

```
public void exit(int status)throws
                              SecurityException
```

Este método é semelhante ao método de mesmo nome da classe **System**.

4. O package java.lang : Runtime

```
public Process exec(...)throws
            IOException,
            SecurityException,
            IndexOutOfBoundsException
```

Existem 4 métodos com este nome que diferem somente nos argumentos.

Eles permitem que se execute um comando para o computador hospedeiro como um novo processo.

4. O package java.lang : Runtime

As variações permitem que se especifique o comando como um **String** que vai ser analisado ou na forma de um *array*, com os argumentos de linha já separados. Além disto, opcionalmente, pode-se especificar um ambiente (*array* de variáveis de ambiente).

4. O package java.lang : Runtime

```
public long totalMemory()
```

Devolve a quantidade de memória total (utilizada e disponível) para esta JVM.

4. O package java.lang : Runtime

`public long freeMemory()`

Devolve a quantidade de memória disponível para uso futuro desta JVM.

4. O package java.lang : Process

A classe **Process**:

Esta classe representa, de uma certa meneira, processos da máquina hospedeira.

Os métodos **exec( )** de **Runtime** devolvem uma referência para **Process**, à qual se aplicam os principais métodos desta classe, que são:

- getOutputStream, getInputStream, getErrorStream
- waitFor
- exitValue
- destroy

4. O package java.lang : Process

```
public abstract OutputStream
                    getOutputStream()
```

Devolve um **Stream** de saída ! A escrita neste **Stream** vai ser direcionada para o stdin do processo representado pela referência a **Process**.

4. O package java.lang : Process

```
public abstract InputStream
                    getInputStream()
```

Devolve um **Stream** de entrada ! A escrita no *stdout* do processo representado pela referência a **Process** vai ser direcionada para este **Stream**, do qual pode-se ler os dados de saída do outro processo.

4. O package java.lang : Process

```
public abstract InputStream
                    getErrorStream()
```

Devolve um **Stream** de entrada ! A escrita no *stderr* do processo representado pela referência a **Process** vai ser direcionada para este **Stream**, do qual pode-se ler os dados de erros do outro processo.

4. O package java.lang : Process

```
public abstract int waitFor()throws
                    InterruptedException
```

A chamada a este método faz com que este *thread* do programa Java se suspenda até que o processo correspondente à referência de **Process** termine, devolvendo um valor de *status* pelo método **`exit()`**, valor este devolvido pelo método.

4. O package java.lang : Process

```
public abstract int
exitValue()throws
        IllegalThreadStateException
```

Se o processo correspondente à referência de **Process** já terminou, o método devolve o valor de status enviado por exit(). Se o processo não terminou, a excessão declarada é lançada.

Este método faz então uma consulta **assíncrona** à terminação de um outro processo, enquanto `waitFor()` faz uma consulta **síncrona**.

4. O package java.lang : Process

```
public abstract void destroy()
```

Este método causa a terminação forçada do processo correspondente.

4. O package java.lang : String

As classes **String** e **StringBuffer**:

Um objeto da classe **String** é intrinsicamente *final*, isto é, uma vez instanciado, ele possui um valor (a cadeia de caracteres) que **não** pode mais ser alterado.

Quer dizer que uma instância de **String** simplesmente encapsula um literal *final* **String**.

Esta classe possui **7** (sete !) **construtores e 40** (quarenta !) **métodos**. Entretanto, os contratos destes métodos são simples e intuitivos.

4. O package java.lang : String

Por outro lado, a classe **StringBuffer** representa cadeias que podem ser modificadas.

Esta classe possui 3 construtores e 28 métodos.

Os métodos desta classe complementam, de certa forma e juntamente com a classe **String**, a funcionalidade que se espera de um tipo de cadeias de caracteres numa linguagem de alto nível.

4. O package java.lang : *wrapper*

As *"wrapper classes"* : Void, Boolean, Character, Number e suas extensões (Byte, Short, Integer, Long, Float e Double)

De uma maneira algo análoga às classes **String** e **StringBuffer**, que encapsulam um valor da sorte de cadeias de caracteres, o pacote **java.lang** também possui classes que encapsulam os tipos básicos (que são simplesmente variáveis, não objetos !).

4. O package java.lang : *wrapper*

Estas classes têm várias utilidades:

- Como a passagem de argumentos em Java só é feita ***por valor***, a passagem de um tipo básico encapsulado numa classe funciona como uma passagem ***por referência***;

- Estas classes são o lugar apropriado para os métodos (*static*) e campos utilitários relacionados com os tipos básicos;

- Elas são úteis para o uso de técnicas de reflexão.

4. O package java.lang : *wrapper*

A classe **Void** é diferente das outras porque ela não encapsula nenhum valor (o tipo básico void tem o conjunto vazio como sorte).

As classes correspondentes aos tipos básicos numéricos possuem uma superclasse, a classe **Number**.

4. O package java.lang : *wrapper*

Todas as *wrapper classes* (exceto **Void**), possuem:

- Um construtor com um argumento que é um valor do próprio tipo básico, que cria o objeto que o encapsula;

- Um construtor com um argumento **String** que converte esta cadeia para o valor correspondente a ser encapsulado (exceto **Character** e **Void**);

(continua)

4. O package java.lang : *wrapper*

(continuação)

- Um método **`valueOf()`** com um argumento **String** que converte esta cadeia e atualiza o valor encapsulado;

- Um método **`toString()`** que faz o trabalho inverso;

- Um método ***`tipo`*`Value()`** que produz um valor do tipo primitivo igual ao valor encapsulado (P.Ex.: **`ob.booleanValue()`**);

(continua)

4. O package java.lang : *wrapper*

(continuação)

- Um método **`equals()`** apropriado a comparar o valor encapsulado;
- Um método **`hashCode()`**;
- Um campo *static* **`TYPE;`**
- Campos e métodos utilitários específicos.

4. O package java.lang : *wrapper*

Exemplo:

A classe Integer possui um método **`parseInt()`** que permite converter a representação de um inteiro dada por um **String** em um valor **`int`** :

```
int i = Integer.parseInt(“1234”);
```

4. O package java.lang : Math

A classe **Math**:

Esta classe possui as constantes (duas: **PI** e **E**) e os métodos básicos (todos *static* !)para os cálculos matemáticos, tais como seno, cosseno, tangente, arcoseno, arcococoseno, arcotangente, exponencial, potência, logarítimos, raiz quadrada, geração de números aleatórios (*random*) e outras funções.

5. Enums

Java possui, desde a JDK 5, um recurso na linguagem (*facility*) para representar tipos enumerados com segurança.

Este recurso é um novo quasi-tipo chamado **`enum`**.

Um exemplo simples é mostrado a seguir:

5. Enums

```
enum EstacoesDoAno {
   PRIMAVERA, VERAO, OUTONO, INVERNO
}
```

Esta construção funciona como uma “definição de constantes”.

Entretanto, esta simplicidade é só aparente.

Na verdade, esta declaração é uma abreviação compreendida pelo compilador do que seria, aproximadamente (!), a declaração:

5. Enums

```
final Class EstacoesDoAno extends java.lang.Enum {
   private final EstacoesDoAno e;
   public EstacoesDoAno(EstacoesDoAno e){
      this.e = e;
   }
   static final EstacoesDoAno PRIMAVERA = new
      EstacoesDoAno(EstacoesDoAno.PRIMAVERA);
   static final EstacoesDoAno VERAO = new
      EstacoesDoAno(EstacoesDoAno.VERÃO);
   static final EstacoesDoAno OUTONO = new
      EstacoesDoAno(EstacoesDoAno.OUTONO);
   static final EstacoesDoAno INVERNO = new
      EstacoesDoAno(EstacoesDoAno.INVERNO);
}
```

(observe a circularidade)

5. Enums

Na verdade, o *enum* é mais complexo que isto:

- A declaração da classe **`java.lang.Enum`**, na verdade é:

```
public abstract class Enum<E extends Enum<E>>
                      extends     Object
                      implements  Comparable<E>,
                                  Serializable
```

Como pode ser visto, a declaração de Enum é abstrata e circular!

5. Enums

Enum é abstrata porque ela é a classe base de todo *enum*.

O construtor de Enum é protected.

A classe definida (ou melhor dizendo, o quasi-tipo `enum EstacoesDoAno`) é final.

5. Enums

- O programador pode acrescentar outros construtores, campos e métodos ao *enum* !

- Cada “constante” é na verdade uma referência (*static* e *final*) para uma instância da classe que estende Enum.

- Tomamos a liberdade de chamar de quasi-tipo um conjunto finito de instâncias.

5. Enums

Podemos então ver a declaração:

```
enum EstacoesDoAno {
   PRIMAVERA, VERAO, OUTONO, INVERNO
}
```

como sendo:

```
static{
 EstacoesDoAno.PRIMAVERA = new EstacoesDoAno(EstacoesDoAno.PRIMAVERA);
 EstacoesDoAno.VERAO = new EstacoesDoAno(EstacoesDoAno. VERAO);
 EstacoesDoAno.OUTONO = new EstacoesDoAno(EstacoesDoAno. OUTONO);
 EstacoesDoAno.INVERNO = new EstacoesDoAno(EstacoesDoAno. INVERNO);
}
```

5. Enums

Podemos ainda, por exemplo, adicionar os seguintes campos, métodos e construtor:

```
public enum EstacoesDoAno {
    PRIMAVERA(new GregorianCalendar(2009,02,10)),
     VERAO(new GregorianCalendar(2009,05,10)),
     OUTONO(new GregorianCalendar(2009,8,10)),
     INVERNO(new GregorianCalendar(2009,11,10));
    private GregorianCalendar data;
    EstacoesDoAno(GregorianCalendar data){
       this.data = data;
    }
    public GregorianCalendar data(){return data;}
}
```

5. Enums

O programa que usa este exemplo é:

```
public class Main {
    static EstacoesDoAno estacao;
    public static void main(String[] args) {
        for(EstacoesDoAno e : estacao.values()){
            System.out.println(
          e.name() +
          " começa em " +
          e.data().get(GregorianCalendar.DATE) +
          "/" +
           (e.data().get(GregorianCalendar.MONTH)+1) +
          ”.”);
        }
    }
}
```

5. Enums

A saída deste é:

```
PRIMAVERA começa em 10/3.
VERAO começa em 10/6.
OUTONO começa em 10/9.
INVERNO começa em 10/12.
```

6. O package java.io

A entrada e saída em Java é baseada no conceito de ***stream***.

Streams **não** são complicados.

O conceito de ***stream*** permite-nos abstrair-nos dos detalhes da estrutura de dados subjacente e utilizar um conjunto de métodos comum a vários tipos de ***streams***.

6. O package java.io

Um ***stream*** encapsula uma seqüência de dados (como veremos, bytes, caracteres, int, etc).

Um ***stream*** ainda possui uma ***fonte*** e um ***destino***.

Se um trecho de código ***escreve*** em um ***stream***, ele é a ***fonte*** deste.

Se um trecho de código ***lê*** de um ***stream***, ele é o ***destino*** deste.

6. O package java.io

Os ***streams*** em Java são unidirecionais e os dois mais simples são:

```
public abstract class InputStream
public abstract class OutputStream
```

Estes são ***streams de bytes***, que não podem ser instanciados, mas que podem ser estendidos.

Entretanto, os ***streams*** da classe **java.lang.System**, in e out, não são instanciados pelo programador mas já estão abertos quando um programa começa a sua execução pela JVM.

6. O package java.io

Outros dois ***streams*** fundamentais de Java são:

```
public abstract class Reader
public abstract class Writer
```

Estes são ***streams de caracteres*** (`char` - 16 bits), que não podem ser instanciados, mas que podem ser estendidos.

Extensões destes podem ser usados para converter ***streams*** de ***bytes*** em ***streams*** de ***char***.

6. O package java.io

Além destas quatro classes básicas, abstratas e genéricas de ***streams***, ainda existem as seguintes classes mais específicas, mas que também estão no topo da hierarquia de ***streams***:

File
FileDescriptor
ObjectStreamClass
SerializablePermission
RandomAccessFile
FilePermission
ObjectStreamField
StreamTokenizer

6. O package java.io

O pacote java.io também possui 10 interfaces:

DataInput
DataOutput
FileFilter
FilenameFilter
ObjectInput
ObjectOutput
ObjectInputValidation
ObjectStreamConstants
Serializable
Externalizable

6. O package java.io

As 4 classes básicas ainda podem são estendidas das seguintes maneiras:

- array streams
- piped streams
- filter streams
  - buffered streams
  - pushback streams

6. O package java.io

Um exemplo simples de uso da classe **InputStream** é mostrado a seguir:

(Observe o uso de *typecasting*)

6. O package java.io

```
import java.io.*;
public class TstIO{
 static byte b;
 static InputStream in = System.in;
 static StringBuffer linha = new StringBuffer();
 public static void main(String[] args) throws
                                        IOException{
   while((b=(byte)in.read())!=13){
      linha = linha.append((char)b);
   }
   System.out.println(linha);
 }
}
```

6. O package java.io

Na verdade, como as 4 classes básicas de ***streams*** são abstratas, é necessário usar classes que são extensões destas para ligar os ***streams*** a dados reais, arquivos ou sockets.

A excessão acontece com objetos que são fornecidos pela JVM e portanto não precisam ser inastanciados, como **System.in** , **System.out** e **System.err** .

6. O package java.io

Estas classes são chamadas de fontes e destinos reais (*sources* e *sinks*):

ByteArrayInputStream	ByteArrayOutputStream
FIleInputStream	FileOutputStream
PipedInputStream	PipedOutputStream
SequenceInputStream	StringBufferInputStream
ObjectInputStream	ObjectOutputStream

6. O package java.io

Um exemplo das classes ByteArrayInputStream e ByteArrayOutputStream é:

```
import java.io.*;
public class TstIO3{
  static StringBuffer linha = new StringBuffer("Teste de arrays...");
  static char c;
  public static void main(String[] args) throws IOException{
     ByteArrayOutputStream out = new ByteArrayOutputStream();
     for(int i=0;i<linha.length();i++)
                    out.write(linha.charAt(i));
     out.flush();
     System.out.println("Size: "+out.size());
     System.out.println(linha);
```

(continua)

6. O package java.io

(continuação)

```
    ByteArrayInputStream in = new
                                ByteArrayInputStream(out.toByteArray());
    linha = new StringBuffer("");
    int k=in.available();
    System.out.println("Available: "+k);
    for(int i=0;i<k;i++){
      c=(char)in.read();
      linha=linha.append(c);
    }
    System.out.println(linha);
  }
}
```

6. O package java.io

As classes **FileInputStream** e **FileOutputStream** permitem ler e escrever bytes em um arquivo.

No exemplo a seguir podemos ver a semelhança com o exemplo anterior, o que mostra o poder do conceito de ***streams***.

6. O package java.io

```
import java.io.*;
public class TstArquivo{
  static char c;
  public static void main(String[] args) throws IOException{
     StringBuffer linha = new StringBuffer("Teste de arquivos...");
     FileOutputStream out = new FileOutputStream("tst-io.txt");
     System.out.println("Vai escrever: "+linha);
     for(int i = 0 ; i<linha.length() ; i++)
                              out.write(linha.charAt(i));
      out.close();
```

(continua)

6. O package java.io

(continuação)

```
    FileInputStream in = new FileInputStream("tst-io.txt");
    linha = new StringBuffer("");
    int k = in.available();
    for(int i=0;i<k;i++){
        c = (char)in.read();
        linha = linha.append(c);
    }
    in.close();
    System.out.println(linha);
  }
}
```

6. O package java.io

Um exemplo das classes **Piped…Stream** é:

```
import java.io.*;
public class TstPipe{
   static PipedOutputStream out = new PipedOutputStream( );
   static PipedInputStream   in = new PipedInputStream( );
   public static void main(String[ ] args){
            Thread th1 = new Thread(new Producer( ));
            th1.start( );
            Thread th2 = new Thread(new Consumer( ));
            th2.start( );
   }
}
```

(contina)

6. O package java.io

(continuação)

```
class Producer implements Runnable{
  String s = new String(
              "Mensagem do produtor para o consumidor...");
   public void run( ){
       try{  TstPipe.out.connect(TstPipe.in);
              for(int i = 0 ; i<s.length() ; i++)
                    TstPipe.out.write(s.charAt(i));
     }catch(IOException e){
          System.out.println("Excessao de IO...");}
   }
}
```

(continua)

6. O package java.io

(continuação)

```
class Consumer implements Runnable{
  public void run( ){
   StringBuffer s = new StringBuffer("");      char c;
   try{  int k = TstPipe.in.available( );
            for(int i = 0 ; i<k ; i++){
                     c = (char)TstPipe.in.read( );
                     s = s.append(c);      }
   }catch(IOException e){System.out.println("Excessao de IO...");}
   System.out.println(s);
 }
}
```

7. Programação multitarefa

Execução concorrente e o modelo do tempo

O objetivo das técnicas de programação concorrente é produzir programas corretos, isto é, que não apresentem erros típicos tais como o bloqueio fatal (*deadlock*), falta de justiça (*starvation*) e outros.

Para compreender estas técnicas é importante entender a **hipótese do intercalamento**, que é a ferramenta básica de raciocínio sobre programas concorrentes.

7. Programação multitarefa

Inicialmente, precisamos de um modelo para o tempo.

A execução de um programa será analisada com este modelo do tempo.

Supomos que o eixo do tempo é dividido em intervalos alternados abertos e fechados, de maneira a cobrir toda a semi-reta:

7. Programação multitarefa

Os **intervalos fechados** representam a **execução** das instruções. O comprimento destes intervalos é o tempo que o processador leva para executar a respectiva instrução. Ele pode ter uma duração finita (dependendo do processador) ou ser zero (!) no caso de um processador ideal.

Durante os **intervalos abertos** o processador **não está executando** qualquer instrução. Eles representam um tempo de folga entre a execução destas.

7. Programação multitarefa

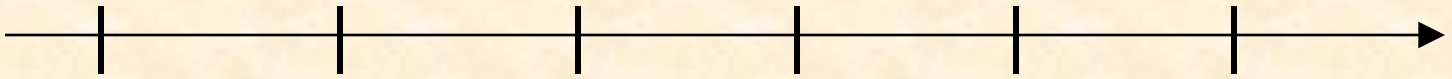
Durante a execução das instruções, **o estado** das posições de memória e registradores do processador **não é definido** (ele pode estar sendo alterado !).

Por outro lado, **durante os intervalos de folga**, **o estado** das posições de memória e os registradores **é bem definido**.

Em particular, o estado do registrador conhecido por **PC** (*program counter*), que indica a próxima instrução a ser executada, tem seu valor válido

7. Programação multitarefa

Se supusermos o caso ideal, em que a execução de cada instrução é instantânea, temos um gráfico como o abaixo:



O trecho de programa a seguir ajuda a ilustrar o uso do modelo:

7. Programação multitarefa

PC = 10

```
10   :  a = 0 ;
20   :  for ( ; ; ) {
21   :     a ++ ;
220  :     if ( a >= 10 )
221  :           break;
           }
```

7. Programação multitarefa

Podemos definir um ***thread*** como sendo um trecho de código em execução.

Para cada ***thread***, vamos ter um **PC** (*program counter*) correspondente.

Em JAVA podemos ter um grande número de threads sendo executados “ao mesmo tempo” em um mesmo programa.

7. Programação multitarefa

Para raciocinar sobre tais programas, vamos assumir a **hipótese do intercalamento**, que se compõe de duas partes:

1) A qualquer instante, só teremos a execução de **uma única instrução**.

2) Entre a execução de duas instruções seqüenciais de um ***thread***, suporemos o **intercalamento arbitrário** da execução de um número finito **de instruções** seqüenciais de algum ou **de** todos os **outros *threads***.

7. Programação multitarefa

A adoção da **hipótese do intercalamento** nos leva **não a uma** seqüência de execução de instruções, **mas a um grande número** (infinito) de **seqüências possíveis**.

Nosso objetivo é construir programas **corretos**, isto é, que cumpram as suas especificações,

para todos os intercalamentos possíveis.

7. Programação multitarefa

Os estados de um thread

As técnicas para resolver os novos problemas baseiam-se em um novo conceito introduzido nos programas concorrentes (*multithreaded*), os estados de um ***thread***.

Um ***thread*** pode estar **Ativo** ou **Suspenso**.

7. Programação multitarefa

Se um ***thread*** estiver **Ativo**, ele certamente (supondo justiça) **será** executado em algum tempo futuro, após a execução de outros ***threads*** mais prioritários e daqueles que estejam à sua frente na fila.

Se um ***thread*** estiver **Suspenso**, ele **não será** executado, a menos que algum outro ***thread***, ou uma temporização façam-no passar para o estado de **Ativo**. Um ***thread*** **Suspenso** não pode passar para o estado **Ativo** por iniciativa própria.

7. Programação multitarefa

O código *synchronized*

Java possui a construção ***synchronized***, que segue o seguinte padrão:

```
synchronized (Object obj) { … }
```

Por exemplo:

```
synchronized(this){ temp=a; a=b; b=temp;}
```

7. Programação multitarefa

Esta construção define um bloco (ou uma única instrução) que é executada com ***exclusão mútua***.

Isto quer dizer que, quando um ***thread*** estiver executando as instruções do bloco, outros ***threads*** que o tentarem passarão do estado de **Ativo** para o estado de **Suspenso**.

7. Programação multitarefa

Além disto, a referência passada no início do bloco (*obj*) é chamada de ***objeto do monitor*** e serve para identificar um **monitor**, isto é, um objeto que pode ser compartilhado através exclusão mútua por vários ***threads***.

7. Programação multitarefa

Blocos ***synchronized*** definidos pelas mesmas referências são relacionados e implicam na restrição de exclusão mútua para todos os blocos com a mesma referência. Estes blocos pertencem ao mesmo **monitor**.

Blocos ***synchronized*** definidos por referências distintas não são relacionados e não implicam na restrição de exclusão mútua entre eles. Eles referem-se a **monitores** distintos.

7. Programação multitarefa

Um ***thread*** que consegue entrar no bloco ***synchronized*** ganha a posse da sincronização sobre o objeto correspondente.

É oportuno lembrar que o objeto referência da sincronização não precisa ter nenhum papel especial no problema a ser resolvido. Ele pode ser simplesmente uma referência para definir o **monitor**.

7. Programação multitarefa

Os ***threads*** que estiverem suspensos por tentar executar um código correspondente ao mesmo **monitor** formam um conjunto de ***threads*** suspensos correspondente àquele **monitor**.

Quando o ***thread*** que está de posse do objeto de sincronização de um **monitor** deixa o bloco sincronizado, um ***thread*** será escolhido de maneira **não determinística** entre aqueles do conjunto de suspensos correspondente.

7. Programação multitarefa

É importante observar que os ***threads*** suspensos em um **monitor** só esperam pela oportunidade de executar o código protegido, isto é, esperam pela saída do ***thread*** que possui a sincronização.

Eles não necessitam que nenhum outro ***thread*** lhes passe algum tipo de sinal explícito.

7. Programação multitarefa

Wait e Notify

A classe **Object** possui dois tipos de métodos que podem fazer a troca de estados de um ***thread***:

```
public final void wait()
                  throws InterruptedException
public final void wait(long timeout)
                  throws InterruptedException

public final void notify()
public final void notifyAll()
```

7. Programação multitarefa

O método **`wait( )`** faz com que o ***thread* atual** passe do estado de **Ativo** para o estado de **Suspenso**, até que outro ***thread*** use o método **`notify( )`** ou **`notifyAll( )`**.

O método **`wait(long timeout)`** faz com que o ***thread* atual** passe do estado de **Ativo** para o estado de **Suspenso**, até que outro ***thread*** use o método **`notify( )`** ou **`notifyAll( )`** ou até que se esgote o tempo especificado em milisegundos pelo argumento **`timeout`**, o que ocorrer primeiro.

7. Programação multitarefa

O método **`notify( )`** acorda, de uma maneira não determinística, um (***só um !***) ***thread*** entre os que tenham executado ***wait*** no **monitor** correspondente.

O método **`notifyAll( )`** acorda todos os ***threads*** que tenham executado ***wait*** no **monitor** correspondente. Estes ***threads*** competirão da maneira usual para adquirir a sincronização deste monitor.

7. Programação multitarefa

Os métodos ***wait***, ***notify*** e ***notifyAll*** aplicam-se ao objeto de referência de um monitor.

O ***thread*** que chama um método ***wait*** (temporizado ou não), ***notify*** ou ***notifyAll*** deve estar de posse do objeto correspondente àquele **monitor**, caso contrário é lançada uma excessão **`IllegalMonitorStateException`**.

7. Programação multitarefa

Os métodos ***wait*** podem lançar uma excessão **`InterruptedException`**. Este tipo de interrupção ainda não está completamente implementada e ainda é simplesmente uma previsão para uma facilidade futura, mas já precisa ser declarada.

7. Programação multitarefa

Exemplo (simples, um thread):

```
synchronized (obj) {
   if (Arbitro.deveSuspender())
      try{
           obj.wait();
      }catch(InterruptedException e){
           System.out.println(“Erro: “+e);
      }
}
```

7. Programação multitarefa

Exemplo (mais real, vários threads, notifyAll):

```
synchronized (obj) {
   while(Arbitro.deveSuspender())
      try{
           obj.wait();
      }catch(InterruptedException e){
           System.out.println("Erro: "+e);
      }
}
```

7. Programação multitarefa

Regiões críticas

Os problemas mais simples de programação concorrente só necessitam que se garanta a exclusão mútua entre a execução de alguns trechos de código que possivelmente acessam dados que precisam manter algum tipo de consistência.

Cada um destes trechos de código passa então a ser chamado de uma **região crítica** e a solução para este problema é simplesmente protégé-las com um bloco **synchronized**.

7. Programação multitarefa

Existe uma forma sintática adaptada do bloco ***synchronized*** que se aplica quando:

- o bloco é todo um método;

- a referência do monitor é o próprio objeto deste método.

Neste caso teríamos por exemplo:

```
public synchronized void metodoA( ) { … }
```

7. Programação multitarefa

Mutex e filas fifo

Um outro exemplo exige que o acesso à região crítica seja feito pela ordem de chegada.

Neste caso, costuma-se chamar este padrão de um **Mutex FIFO**, mostrado no exemplo a seguir:

```
public class MutexFIFO{
    Object obj;
    Vector threadFIFO = new Vector();
    boolean saiuUm = false;
```

(continua)

7. Programação multitarefa

```
public synchronized void mutexLock(){
      threadFIFO.add(Thread.currentThread());
      while (!threadFIFO.firstElement( ).equals
                              (Thread.currentThread( )) || saiuUm)
                        try{this.wait();}catch(Exception e){}
     try{     Object o = threadFIFO.firstElement();
              threadFIFO.remove(o);
      }catch(NoSuchElementException ex){}
      saiuUm = true;
}
```

(continua)

7. Programação multitarefa

```
    public synchronized void mutexUnlock(){
        saiuUm = false;
        this.notifyAll();
    }
}
```

7. Programação multitarefa

Uma aplicação típica poderia ser assim:

```
public class Main{
  public static Random r;       public static MutexFIFO m;
  public static int nro;
  static final int MAXTHREADS = 10;
        public static void main(String[ ] args){
                r = new Random( );
                m = new MutexFIFO( );
                for(int i=0 ; i<MAXTHREADS ; i++)
                        new Thread(new Tarefa( )).start( );
        }
}
```

(continua)

7. Programação multitarefa

(continuação)

```
class Tarefa implements Runnable{
      static int serial;
      int id;
      public Tarefa( ){
            id = serial++;
      }
```

(continua)

7. Programação multitarefa

(continuação)

```
  public void run( ){
      System.out.println("Ini. tar.” + id + "." );
      while(true){
          try{Thread.sleep(500));}catch(InterruptedException e){};
          Main.m.mutexLock( );
          try{Thread.sleep(500);}catch(InterruptedException e){ };
          Main.m.mutexUnlock( );
      }
  }
}
```

7. Programação multitarefa

Cada região crítica independente usaria um objeto diferente. Regiões críticas relacionadas usariam o mesmo objeto mutex (no exemplo, rc).

7. Programação multitarefa

Semáforos

Outro tipo de problemas é melhor solucionado com um dispositivo chamado de semáforo.

Um semáforo é uma classe que encapsula uma variável inteira não negativa (e.g. s), iniciada pelo construtor, e que possui dois métodos:

- semWait( )
- semSignal( )

7. Programação multitarefa

- semWait( ) :

se s>0 então s = s - 1 ,

caso contrário, o thread se suspende em uma fila FIFO;

- semSignal( ) :

se existirem threads suspensos, acorda o primeiro da fila,

caso contrário, s = s + 1 .

7. Programação multitarefa

Exemplo:

```
public class Semaforo{
   Object  obj;
   Vector  threadFIFO = new Vector();
   int  s;
//-------------------------------------
   public Semaforo(int s0){
      s = s0;
   }
//-------------------------------------
```

7. Programação multitarefa

(continuação)

```
public synchronized void semWait(){
   threadFIFO.add(Thread.currentThread());
   while(s==0 || !threadFIFO.firstElement()
            .equals(Thread.currentThread( )))
      try{this.wait();}catch(Exception e){}
   try{ threadFIFO.remove(
                  threadFIFO.firstElement());
   }catch(NoSuchElementException ex){}
   s--;
   this.notifyAll();
}
```

(continua)

7. Programação multitarefa

(continuação)

```
    public synchronized void semSignal(){
        this.notifyAll();
        s++;
    }
}
```

7. Programação multitarefa

Um exemplo típico de semáforos seria o produtor-consumidor.

O código típico de um produtor seria:

```
public void run(){
   for(;;){
      item = produzir();
      podeColocar.semWait();
      Buffer.colocar(item);
      podeRetirar.semSignal();
   }
}
```

7. Programação multitarefa

O código típico de um consumidor seria:

```
public void run(){
   for(;;){
      podeRetirar.semWait();
      item = Buffer.retirar();
      podeColocar.semSignal();
      consumir(item);
   }
}
```

7. Programação multitarefa

Monitor de Hoare

Existe um padrão para a construção de monitores de sincronização conhecido como Monitor de Hoare.

Um monitor de Hoare em Java também é implementado como um objeto com ***um conjunto de métodos sincronizados***.

Além disto, ele encapsula objetos de uma classe chamada **Condition**.

7. Programação multitarefa

Um objeto da classe **Condition** encapsula uma fila FIFO de *threads* suspensos e possui dois métodos públicos:

condWait - que suspende incodicionalmente o *thread* atual na fila FIFO;

condSignal - que causa a retomada de execução do primeiro *thread* da fila FIFO, se houver um.

7. Programação multitarefa

Um semáforo pode ser implementado como um monitor de Hoare.

O método ***wait*** de um ***semáforo*** seria:

```
public synchronized void semWait(){
        if(s==0)
           c.condWait();
        else
           s--;
}
```

7. Programação multitarefa

O método ***signal*** de um ***semáforo*** seria:

```
public synchronized void semSignal(){
        if(c.notEmpty())
           c.condSignal();
        else
           s++;
}
```

7. Programação multitarefa

Leitores e escritores

O problema dos **leitores e escritores** é importante porque é um problema de exclusão mútua generalizado.

Sua solução clásica com um **monitor de Hoare** indica um método genérico para problemas deste tipo.

7. Programação multitarefa

A especificação do problema dos **leitores e escritores** pode ser resumida como:

1. Um arquivo é compartilhado por leitores e escritores.

2. Vários leitores podem ler ao mesmo tempo.

3. Um escritor só pode escrever no arquivo sob exclusão mútua, seja em relação a leitores ou a outros escritores.

7. Programação multitarefa

4. Quando leitores estão lendo e escritores esperando para escrever, outros leitores que quiserem ler serão suspensos.

5. Quando um escritor terminar de escrever e houver leitores e escritores querendo acessar o arquivo, a prioridade será dada aos leitores.

6. Quando o último dos leitores terminar de ler e houver leitores e escritores querendo acessar o arquivo, a prioridade será dada ao primeiro escritor.

7. Programação multitarefa

7. Programação multitarefa

O código típico de um **leitor** é o seguinte:

```
class Leitor{
   public void run(){
      while(true){
            le.queroLer();
            <ler o arquivo>
            le.acabeiLer();
            <processar o que leu>
      }
   }
}
```

7. Programação multitarefa

O código típico de um **escritor** é o seguinte:

```
class Escritor{
   public void run(){
      while(true){
            <produzir para escrever>
            le.queroEscrever();
            <escrever no arquivo>
            le.acabeiEscrever();
      }
   }
}
```

7. Programação multitarefa

O objeto **le** que aparece nos leitores e escritores é uma instância do monitor:

```
class LeitEsc{
   public synchronized void queroLer(){
   . . . }
   public synchronized void acabeiLer(){
   . . . }
   public synchronized void queroEscrever(){
   . . . }
   public synchronized void acabeiEscrever(){
   . . . }
}
```

7. Programação multitarefa

O monitor **LeitEsc** encapsula duas instâncias da **Condition** de Hoare:

podeLer e **podeEscrever**

e duas variáveis:

int leitores;

boolean escrevendo;

7. Programação multitarefa

O código do monitor é:

```
public synchronized void queroLer( ){
   if(escrevendo || podeEscrever.notEmpty( )
      podeLer.condWait( );
   leitores++;
   if(podeLer.notEmpty( ))
      podeLer.condSignal( );
}
```

7. Programação multitarefa

```
public synchronized void acabeiLer(){
   leitores--;
   if((leitores==0) &&
                     podeEscrever.notEmpty()){
            podeEscrever.condSignal();
   }
}
```

7. Programação multitarefa

```
public synchronized void queroEscrever(){
   if(escrevendo || (leitores>0)){
      podeEscrever.condWait();
   }
   escritores++;
}
```

7. Programação multitarefa

```
public synchronized void acabeiEscrever(){
   escritores--;
   if(podeLer.notEmpty()){
      podeLer.condSignal();
   }else if(podeEscrever.notEmpty()){
      podeEscrever.condSignal();
   }
}
```

7. Programação multitarefa

O paradigma cliente-servidor

Suponhamos dois objetos que estendem **Thread** e que seus respectivos métodos ***run( )*** tenham sido iniciados como novos ***threads***.

Estes objetos são chamados de ***objetos ativos***. Por outro lado, objetos que não possuam seus próprios ***threads*** são chamados de ***objetos passivos***.

7. Programação multitarefa

O paradigma **cliente-servidor** corresponde à situação em que um objeto ativo, o **cliente**, chama um método de um outro objeto ativo, o **servidor**.

Como em algum momento poderíamos ter dois ***threads***, do próprio servidor e do cliente, executando código do servidor, precisamos garantir a consistência do estado do servidor.

7. Programação multitarefa

O padrão mostrado a seguir faz isto controlando a chamada do cliente, só permitindo a sua execução durante a chamada do método ***acceptMetodoA( )***.

7. Programação multitarefa

```
Class Servidor{
   public void run(){
      while(true){
            acceptMetodoA( );
            processarDados();
      }
   }
   synchronized void acceptMetodoA(){. . .}
   public synchronized void metodoA( ){. . .}
}
```

7. Programação multitarefa

Se o servidor chegar ao ***acceptMetodoA( )***, antes do cliente fazer a chamada ao ***metodoA( )***, **o servidor vai esperar** suspenso até que o cliente faça a chamada.

Se o cliente fizer a chamada ao ***metodoA( )*** antes do servidor executar ***acceptMetodoA( )***, **o cliente vai esperar** suspenso até que o servidor execute o ***acceptMetodoA( )***.

7. Programação multitarefa

Quando tanto o cliente tiver chamado o ***metodoA( )*** como o servidor tiver chamado ***acceptMetodoA( )*** , diz-se que a transação foi iniciada.

Iniciada a transação, **o servidor espera que o cliente termine a execução** do ***metodoA( )*** para continuar sua execução.

7. Programação multitarefa

Servidores sequencial e concorrente

Diz-se que um **servidor é seqüencial** se, a cada pedido, ele presta o serviço completamente antes de aceitar outro pedido.

Diz-se que um **servidor é concorrente** se ele começa a atender um novo pedido antes de terminar o atendimento do pedido anterior.

7. Programação multitarefa

Um servidor concorrente pode ser implementado com um único ***thread*** - **servidor concorrente com implementação seqüencial** - ou ele pode ser implementado com um ***thread*** para fazer a aceitação dos pedidos (***front-end***) e com a criação de novos ***threads*** para prestar o serviço correspondente a cada pedido (***back-end***) - **servidor concorrente com implementação concorrente**.

7. Programação multitarefa

O código destes métodos poderia ser:

```
synchronized void acceptMetodoA(){
    vezAccept = false;
    cp.condSignal();
    cf.condWait();
}
```

(continua)

7. Programação multitarefa

(continuação)

```
public synchronized Pedido metodoA(Pedido p){
   Thread t;
   Filho filho;
   if(!cf.notEmpty() || vezAccept) cp.condWait();
   vezAccept = true;
   cf.condSignal();
   filho = new Filho(p);
   t = new Thread(filho);
   t.start();
   try{t.join();}catch(Exception e){}
   return p;
}
```

7. Programação multitarefa

Hoje em dia, este padrão de programação está se tornando um dos principais para linguagens OO, porque:

(i) ele não se afasta do paradigma de OO e

(ii) porque hoje existem os recursos de CORBA e do RMI em Java que permitem estender o padrão para objetos distribuídos.

8. Programação distribuída: *java.rmi*

Java possui um mecanismo para construir sistemas distribuídos que incorpora os conceitos mais modernos de programação OO, além de possuir um bom desempenho.

Trata-se do ***rmi - remote method invocation***, ou ***chamada remota de procedimento***.

8. Programação distribuída: *java.rmi*

Usando-se **rmi**, um objeto (de uma máquina que chamaremos de ***máquina local***) pode chamar métodos de outros objetos que estejam em outras máquinas (***máquinas remotas***) como se eles estivessem na mesma máquina.

Como é usual, o objeto que faz a chamada é o **objeto cliente**, enquanto o objeto cujo método é invocado é o **objeto servidor**.

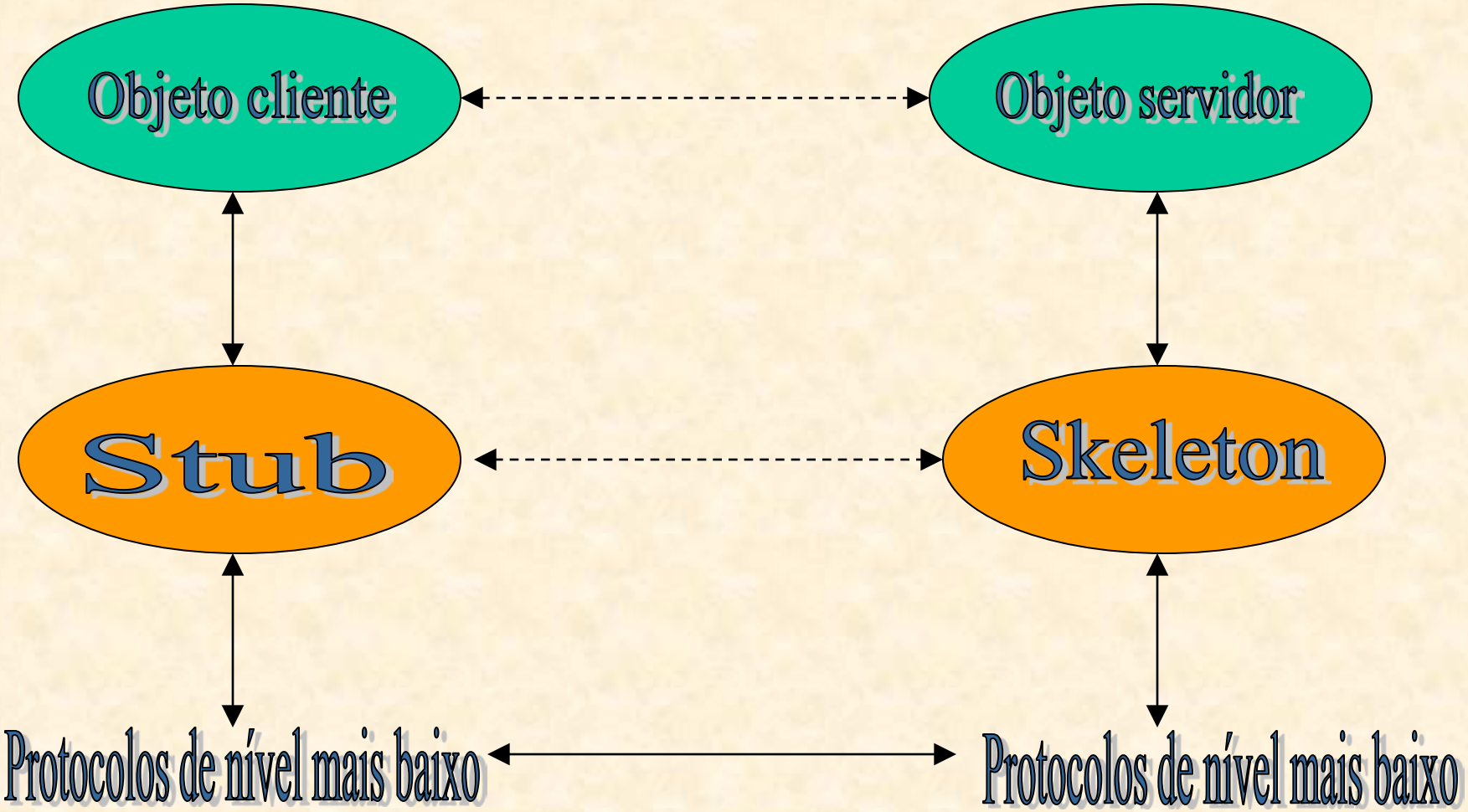
É importante observar que **não** é necessário que qualquer dos dois objetos seja um objeto ativo.

8. Programação distribuída: *java.rmi*

Como o ponto de vista preferido costuma ser o do cliente, o **objeto cliente** é frequentemente chamado de **objeto local**, enquanto o **objeto servidor** é o **objeto remoto**.

A implementação do ***rmi*** é feita através uma ***arquitetura em camadas***, que na sua forma mais simples pode ser vista como na figura a seguir.

8. Programação distribuída: *java.rmi*



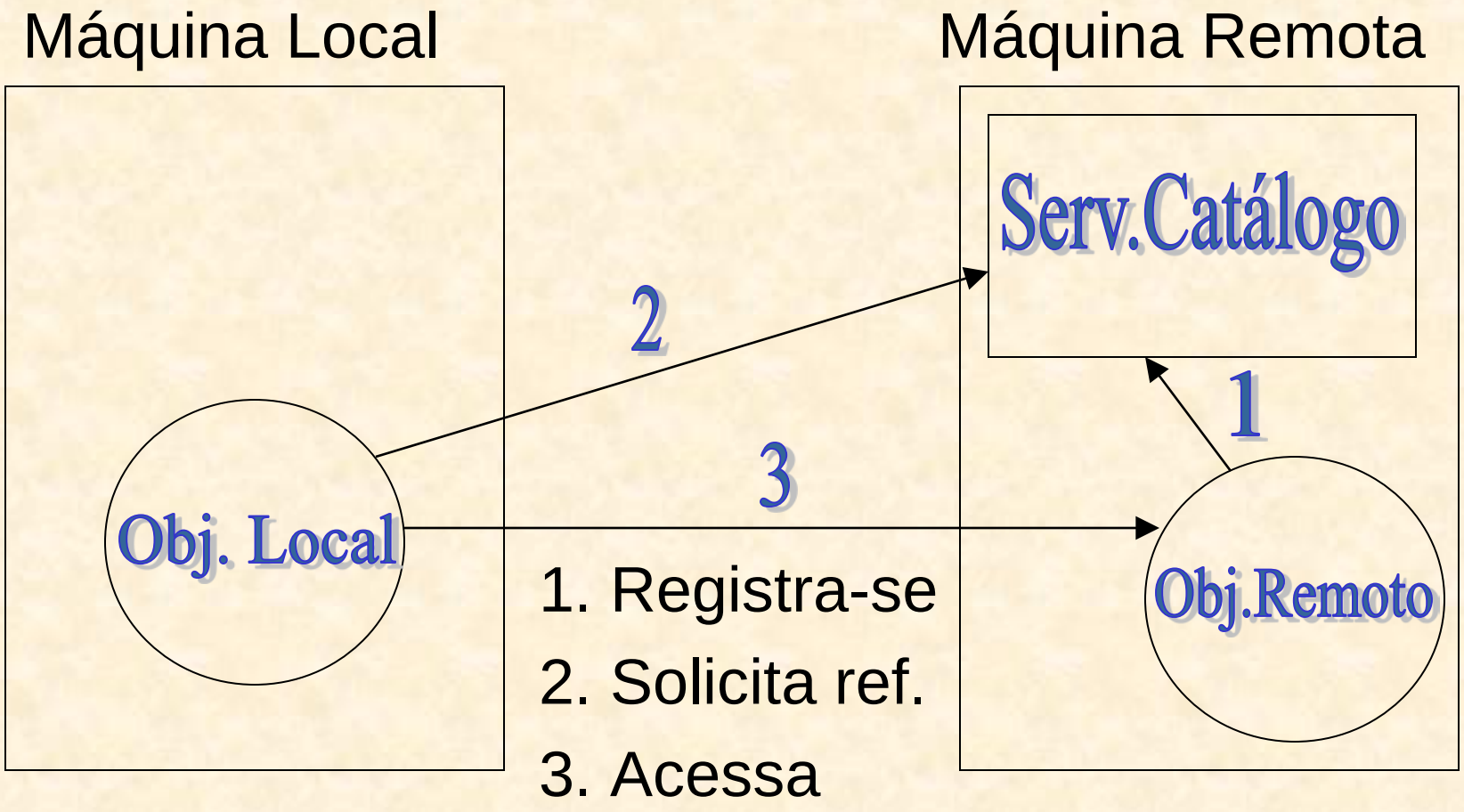
8. Programação distribuída: *java.rmi*

Para que um objeto remoto exporte sua interface para outros objetos, ele precisa registrar-se num servidor chamado ***remote registry server*** ***(servidor de catálogo)***.

Este servidor reside ***na mesma máquina*** do objeto remoto.

Assim, o objeto cliente (ou local) consulta antes o ***servidor de catálogo*** para obter uma referência do objeto servidor (ou remoto).

8. Programação distribuída: *java.rmi*



8. Programação distribuída: *java.rmi*

Um objeto remoto precisa implementar a interface ***Remote*** e estender a classe ***rmi.server.RemoteServer*** ou uma sua descendente, como a ***UnicastRemote Object***.

Todos os métodos desta interface, e que portanto podem ser invocados remotamente, devem declarar a excessão ***RemoteException***.

8. Programação distribuída: *java.rmi*

Vamos exemplificar através um servidor de mensagens simpliifcado, que possui duas classes servidoras (objetos remotos).

As classes são a de ***Usuario*** e a de ***Mensagem***.

Primeiramente declaramos as interfaces que vão ser oferecidas remotamente.

8. Programação distribuída: *java.rmi*

```
package mensagem;
import java.util.*;
import java.rmi.*;

public interface IFUsuario extends Remote{
public static synchronized Usuario
getUserByName(String n);
public synchronized String getName();
public synchronized void openInbox();
public synchronized Mensagem nextInbox();
public synchronized void delInboxMsg();
}
```

8. Programação distribuída: *java.rmi*

```
package mensagem;
import java.util.*;
import java.rmi.*;

public interface IFMensagem extends Remote{
public synchronized void setText(String t);
public synchronized String getText();
public synchronized Usuario getSender();
public synchronized Usuario getDest();
public synchronized void setDest(Usuario
                                 dest);
public synchronized void send(Usuario dest);
}
```

8. Programação distribuída: *java.rmi*

As classes dos servidores vão estender **UnicastRemoteObject** e implementar a respectiva interface:

```
package mensagem;
import java.util.*;
import java.rmi.*;
import java.rmi.server.*;

public class Usuario extends
   UnicastRemoteObject implements IFUsuario{
```

(continua)

8. Programação distribuída: *java.rmi*

(continuação)

```
    . . .
        String RMIServerHostIP     = args[0];
        String port                = args[1];
        String remoteObjectName = "/Worker";
        String url = "rmi://" + RMIServerHostIP
              + ":" + port + remoteObjectName;
        Naming.rebind(url, workerRef);

}
```

8. Programação distribuída: *java.rmi*

```
package mensagem;
import java.util.*;
import java.rmi.*;
import java.rmi.server.*;

public class Mensagem extends
    UnicastRemoteObject implements IFMensagem{
    . . .
}
```

8. Programação distribuída: *java.rmi*

A classe do cliente precisa fazer uma consulta ao servidor de catálogo (***Remote Registry Server***) antes de poder fazer a invocação do método remoto.

Isto é feito, por exemplo, através o método estático **`lookup`** da classe **`rmi.Naming`**.

8. Programação distribuída: *java.rmi*

```
package mensagem;
import java.util.*;
import java.rmi.*;
public class Cliente{
   static Mensagem m;
   public static void main(String[] args){
     String url = new String( "rmi://" +
                  RMIServerHostIP + port +
                                  "/Usuario" );
     IFUsuario userRef =
             (IFUsuario)Naming.lookup(url);
```

(continua)

8. Programação distribuída: *java.rmi*

```
while(true){
      try{Thread.sleep(1000);
      }catch(Exception e){}
```

(continua)

8. Programação distribuída: *java.rmi*

(continuação)

```
            m = new Mensagem(eu);
            m.setText("Bla, bla, bla, ...");
            m.send(
             Usuario.getUserByName("MARIA"));
            eu.openInbox();
            m = eu.nextInbox();
            System.out.println(m.getText());
            eu.delInboxMsg();
        }
    }
}
```

8. Programação distribuída: *java.rmi*

O próximo passo é compilar o cliente e o servidor com o compilador javac, da maneira usual:

```
javac  *.java
```

Depois, deve-se usar o compilador rmi, o rmic, para gerar o ***stub*** e o ***skeleton*** a partir dos arquivos **`*.class`** dos servidores:

```
rmic  mensagem.Usuario
rmic  mensagem.Mensagem
```

8. Programação distribuída: *java.rmi*

O compilador rmic gera os seguintes arquivos:

`Usuario_Skel.class,`

`Usuario_Stub.class,`

`Mensagem_Skel.class e`

`Mensagem_Stub.class`

8. Programação distribuída: *java.rmi*

O próximo passo é iniciar o servidor de catálogo (Remote Registry Server) na máquina onde o servidor vai ser executado:

```
start rmiregistry   (windows)
```

No Unix, basta:

```
rmiregistry &
```