



Universidade do Porto
Faculdade de Engenharia
FEUP

Programação em Java: linguagem, APIs, boas práticas e Eclipse

FEUP, Novembro 2005

Ademar Aguiar

ademar.aguiar @ fe.up.pt

<http://www.ademarguiar.org/>



Universidade do Porto
Faculdade de Engenharia
FEUP

Objectivos

- Aprender a desenvolver programas em linguagem Java:
 - utilizar os principais utilitários do kit de desenvolvimento para Java (JDK) versão 1.4.2.
 - desenvolver interfaces gráficas em linguagem Java recorrendo aos packages AWT e Swing.
 - conhecer e utilizar as principais funcionalidades disponíveis nos packages de colecções, entrada e saída de dados, acesso a dados de bases de dados e acesso a dados remotos.
- Motivar os participantes para a adopção de boas práticas de desenvolvimento de software:
 - testes unitários, refactoring, padrões de desenho, revisão de código, documentação,
- Utilizar o Eclipse como ambiente de desenvolvimento (IDE).
- Consolidação dos conhecimentos transmitidos através da sua aplicação na resolução de exercícios práticos.

Conteúdos

- Parte 1: Introdução ao Java e Classes fundamentais
- Parte 2: Collections e Entrada/saída de dados
- Parte 3: Interfaces gráficas com Swing
- Parte 4: Acesso a dados remotos por JDBC e HTTP

Bibliografia

- “Object-oriented Design With Applications”, Grady Booch,, The Benjamin/cummings Publishing Company Inc., 2nd Edition, Redwood City, California, 1995.
- “The Java Programming Language”, K. Arnold, J. Gosling, Adisson-Wesley, 2nd Edition, 1998, ISBN 0-201-31006-6.
- "Java in a Nutshell", Flanagan, David, O'Reilly & Associates, 2004.
- "Java Examples in a Nutshell", Flanagan, David, 3rd edition, O'Reilly & Associates, 2004.
- “Eclipse: Eclipse Rich Client Platform: Designing, Coding, and Packaging Java Applications”, Jeff McAffer, Jean-Michel Lemieux, Eclipse series, Addison-Wesley, 2005.



Universidade do Porto

Faculdade de Engenharia

FEUP

Parte 1



Universidade do Porto
Faculdade de Engenharia
FEUP

Introdução ao Java

Objetivos

- Identificar os elementos principais do Java
- Descrever a Java Virtual Machine (JVM)
- Comparar a utilização do Java para a construção de *applets* e de aplicações
- Identificar os componentes principais do Java Development Kit (JDK)
- Descrever as opções de instalação do Java (deployment)

O que é o Java?

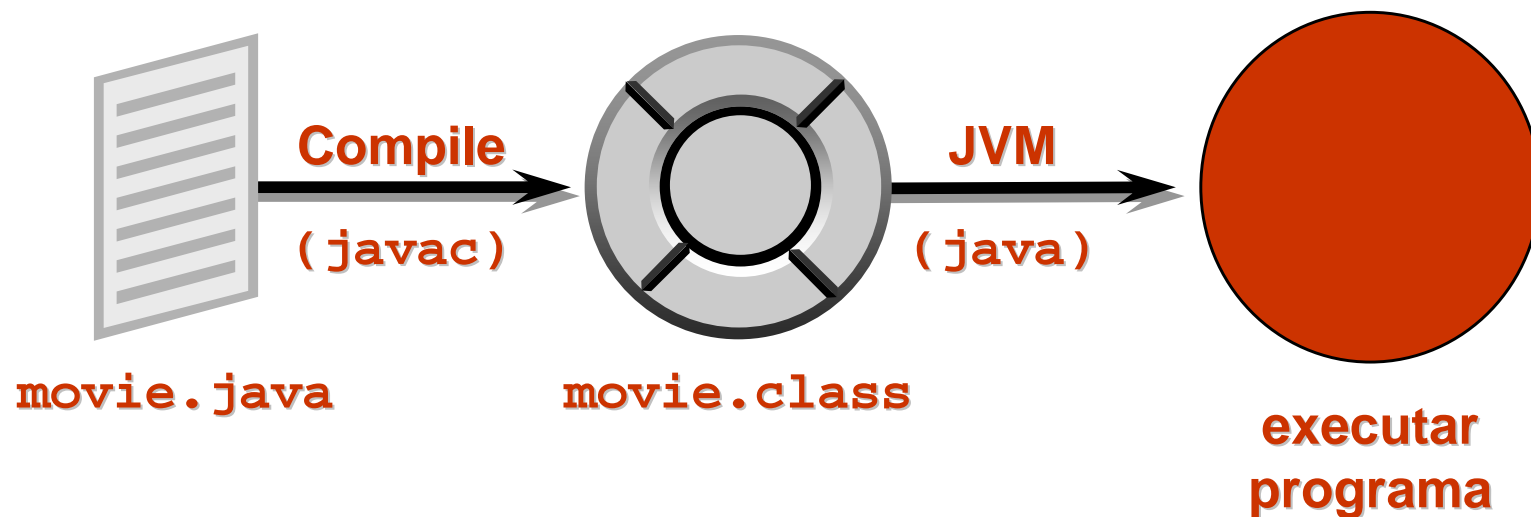
- Concebido pela Sun para a electrónica de consumo, mas rapidamente alcançou a WWW.
- Uma linguagem orientada por objectos e um conjunto de bibliotecas de classes (frameworks).
- Utiliza uma máquina virtual para a execução de programas.

Vantagens Principais do Java

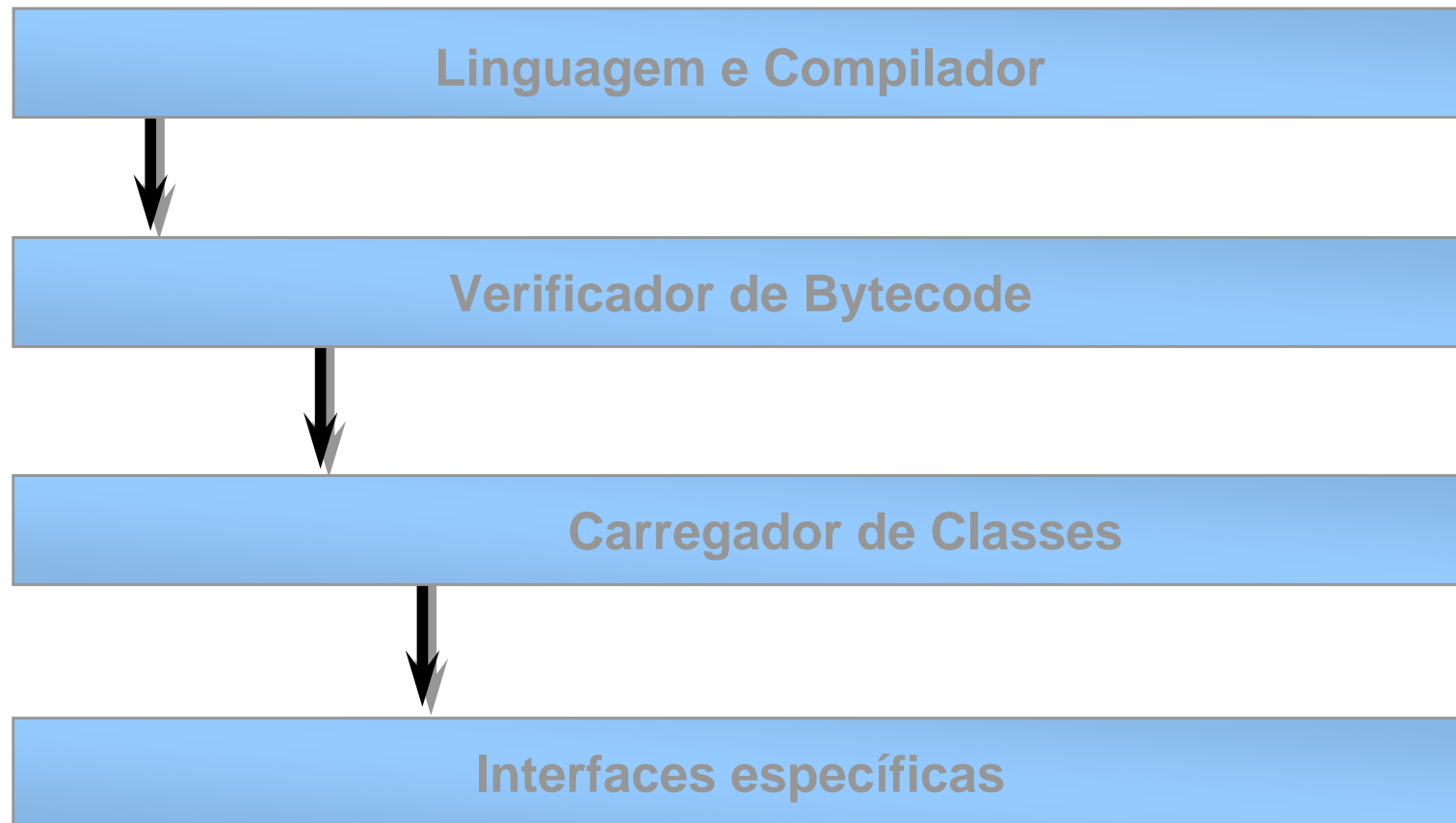
- Linguagem orientada por objectos
- Interpretado e independente da plataforma
- Dinâmico e distribuído
- “Multi-threaded”
- Robusto e seguro

Independente da Plataforma

- O código Java é armazenado num ficheiro `.java`
- Um programa `.java` é compilado para ficheiros `.class`
- Bytecodes são interpretados em tempo de execução



Ambiente de Segurança do Java



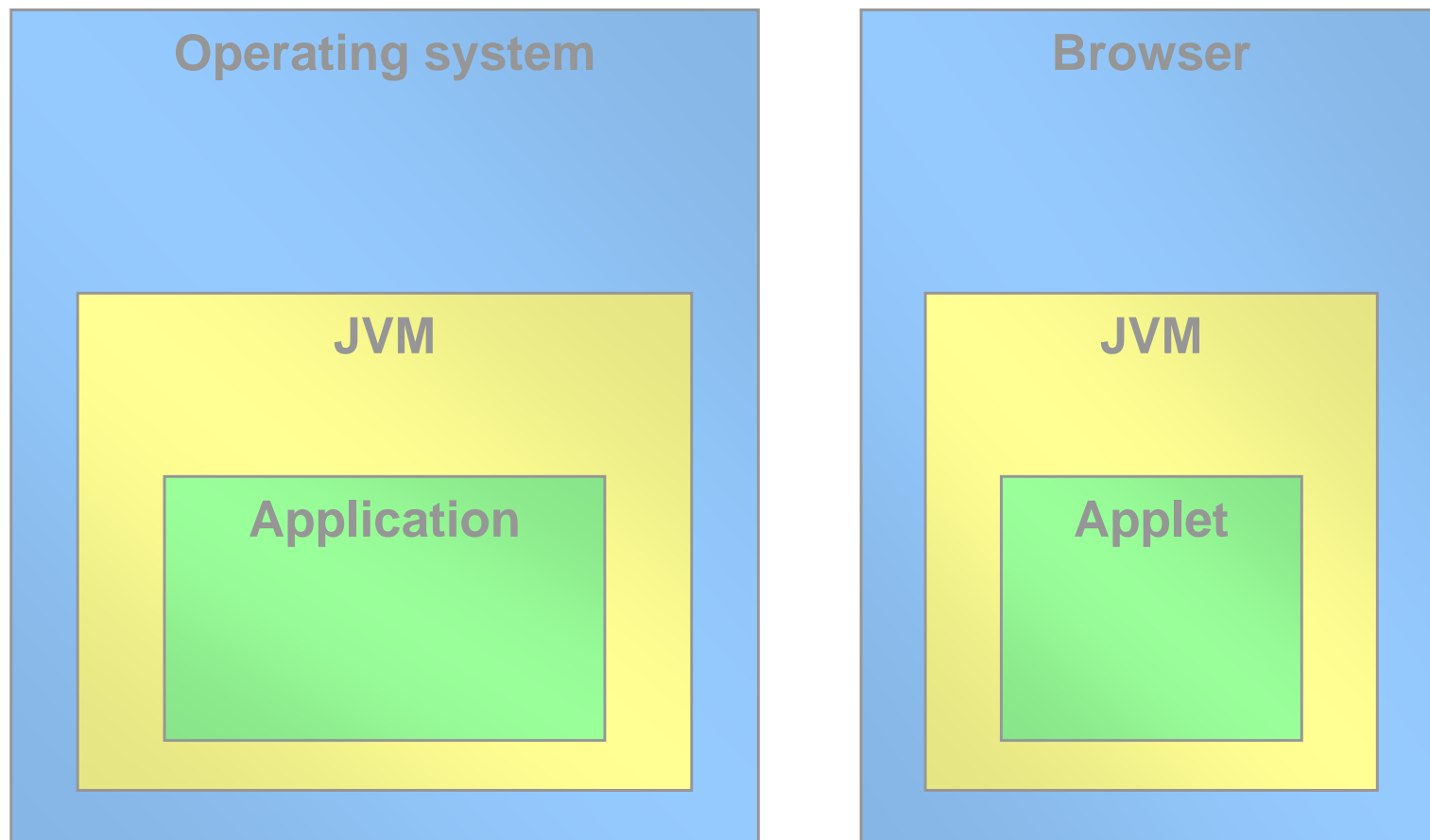
Applets Java

- A forma mais comum de utilização do Java, inicialmente
- Vocacionada para utilização em páginas HTML
- Pode incluir conteúdos activos (forms, áudio, imagens, vídeo)
- Aparece num *browser* e pode comunicar com o servidor

Aplicações Java

- Instalação no lado do cliente
 - JVM corre em aplicações autónomas
 - Não necessita de carregar classes pela rede
- Instalação do lado do servidor
 - Pode servir múltiplos clientes a partir de uma mesma origem
 - Encaixa bem com modelos multi-camada para computação na Internet

JVM - Java Virtual Machine



Como funciona a JVM

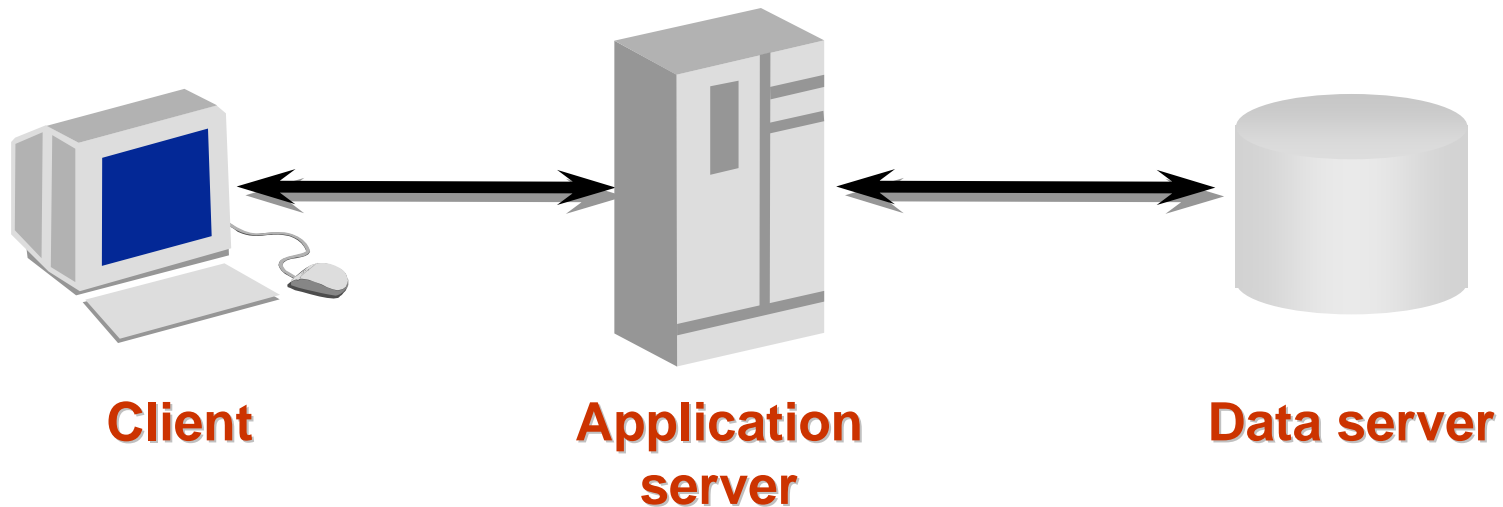
- O “JVM class loader” carrega todas as classes necessárias.
- O “JVM verifier” verifica os bytecodes ilegais.
- O gestor de memória da JVM liberta memória de volta ao sistema operativo.

Compiladores Just-in-Time (JIT)

- Melhoram a performance
- São úteis se os mesmos *bytecodes* forem executados repetidas vezes
- Traduz bytecodes para instruções nativas
- Optimizam código repetitivo, tais como ciclos

Java e Computação na Internet

- A computação na Internet opera-se em três camadas:



- Java pode ser usada em todas estas camadas.

Resumo

- O código Java é compilado para *bytecodes* independentes da plataforma.
- Os *bytecodes* são interpretados por uma JVM.
- As *applets* correm num browser no cliente.
- As aplicações Java são executadas de forma autónoma tanto no cliente como no servidor.



Universidade do Porto
Faculdade de Engenharia
FEUP

Conceitos Básicos do Java

Objectivos

- Conhecer os elementos principais do Java
- Conhecer a sintaxe básica do Java
- Descrever ficheiros `.java` e `.class`

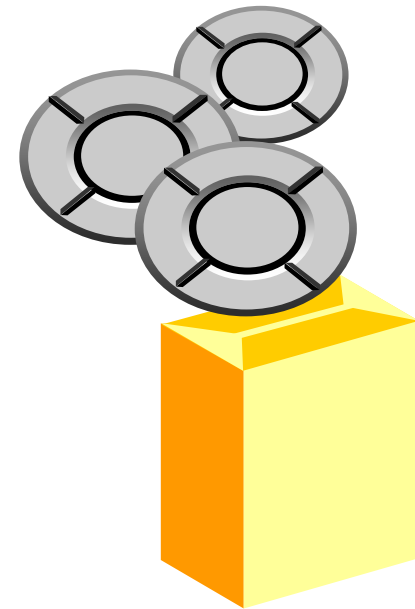
Tópicos

- Componentes Java
- Convenções
- Classes, objectos e métodos
- Utilização de Javadoc
- Compilar e executar programas Java

JDK - Java Development Kit

O JDK da Sun fornece:

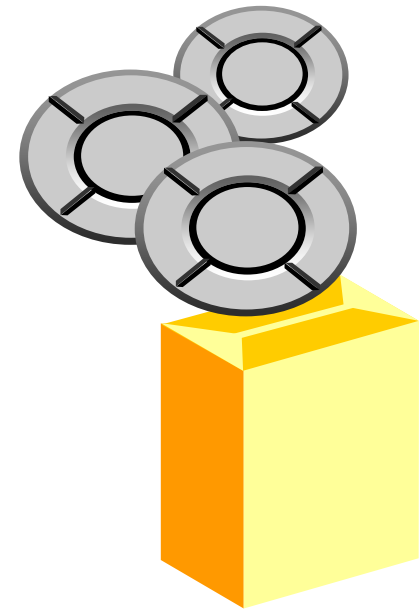
- Compilador (javac)
- Visualizador de *applets* (appletviewer)
- Interpretador de *bytecode* (java)
- Gerador de documentação (javadoc)



JDK - Java Development Kit

O JDK da Sun fornece pacotes standard para:

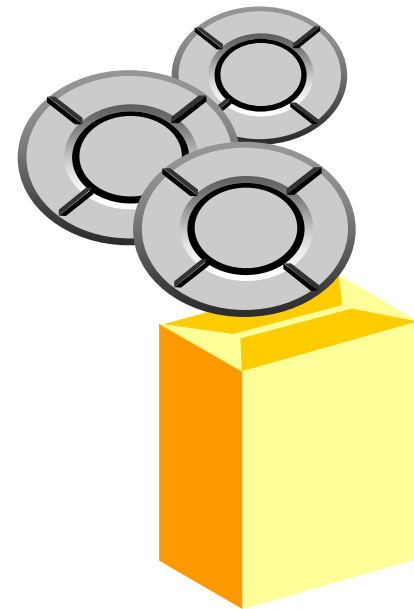
- linguagem
- sistema de janelas
- controlo de *applets*
- entrada/saída
- comunicação em rede



JDK - Java Development Kit

O JDK da Sun fornece suporte de documentação para:

- Comentários
 - Implementação
 - Documentação
- Gerador de Documentação (javadoc)



Convenções de Nomes

As convenções incluem:

- Nomes de ficheiros (Palavras capitalizadas)
 - Customer.java, RentalItem.java
- Nomes de Classes (Palavras capitalizadas)
 - Customer, RentalItem, InventoryItem
- Nomes de Métodos (verbo + palavras capitalizadas)
 - getCustomerName(), setRentalItemPrice()

Convenções de Nomes...

- Standard para variáveis
 - `customerName`, `customerCreditLimit`
- Standard para constantes
 - `MIN_WIDTH`, `MAX_NUMBER_OF_ITEMS`
- Utilização de caracteres maiúsculos e minúsculos
- Números e caracteres especiais

Definição de Classes

- A definição de classes normalmente inclui:
 - Modificador de acesso: *public*, *private*
 - A palavra-chave *class*
 - Campos das instâncias
 - Construtores
 - Métodos das instâncias
 - Campos da classe
 - Métodos da classe

Definição de Classes...

```
public class Customer {  
    // Instance variáveis  
    String customerName;  
    String customerPostalCode;  
    float customerAmountDue;  
    ...  
    // Instance métodos  
    float getAmountDue (String cust) {  
        ...  
    }  
    ...  
}
```

Declaração

**Variável
de
Instância**

**Método
da
Instância**

Definição de Métodos

- Sempre dentro de uma classe
- Especificam:
 - Modificador de acesso
 - Palavra-chave *static*
 - Argumentos
 - Tipo de retorno

```
[access-modifiers] [static] <return-type>  
<method-name> ([arguments]) <java code block>
```

Definição de Métodos

```
float getAmountDue (String cust) {  
    // método variáveis  
    int numberOfDays;  
    float due;  
    float lateCharge = 1.50;  
    String customerName;  
    // método body  
    numberOfDays = this.getOverDueDays();  
    due = numberOfDays * lateCharge;  
    customerName = getCustomerName(cust);  
    return due;  
}
```

Declaração

**Variáveis
de método**

**Instruções
de método**

Retorno

Variáveis e Constantes

- Devem ser declaradas antes de ser utilizadas
- Uma declaração por linha
- No início de um bloco de código
- O bloco de código define o âmbito
- A inicialização imediata é opcional

Variáveis e Constantes

```
float getAmountDue (String cust) {  
    float due = 0;  
    int numberOfDays = 0;  
    float lateFee = 1.50;  
    {int tempCount = 1; // new code block  
        due = numberOfDays * lateFee;  
        tempCount++;  
        ...  
    } // end code block  
    return due;  
}
```

**Variáveis
de método**

**Variáveis
temporárias**

Criação de blocos de código

- Agrupar todas as declarações de classe.
- Agrupar todas as declarações de métodos.
- Agrupar outros segmentos de código relacionado entre si.

```
public class SayHello {  
    public static void main(String[] args) {  
        System.out.println("Hello world");  
    }  
}
```

Criação de instruções

- As instruções terminam sempre com um ponto-e-vírgula (;)
- Instruções compostas são definidas dentro de chavetas { }.
- Utilizar chavetas para instruções de controlo.

Compilar e Executar Aplicações

- Para compilar um ficheiro .java:

```
$aaguiar> javac SayHello.java  
... compiler output ...
```

- Para executar um ficheiro .class:

```
$aaguiar> java SayHello  
Hello world  
Prompt>
```

- Atenção às maiúsculas e minúsculas!

Resumo

- O JDK fornece as ferramentas Java essenciais.
- O JDK fornece um conjunto valioso de classes e métodos pré-definidos.
- Os programas Java são constituídos por classes, objectos, e métodos.
- A adopção de normas de programação facilita a leitura e reutilização de código.

Exercícios

- Ex1. Instalar JDK 1.4.2_05
- Ex2. HelloWorld
- Ex3. Instalar Eclipse 3.1.1
- Ex4. FizzBuzz
- Ex5. Echo
- Ex6. Reverse Echo
- Ex7. Factorial



Universidade do Porto
Faculdade de Engenharia
FEUP

Tipos de Dados e Operadores

Objetivos

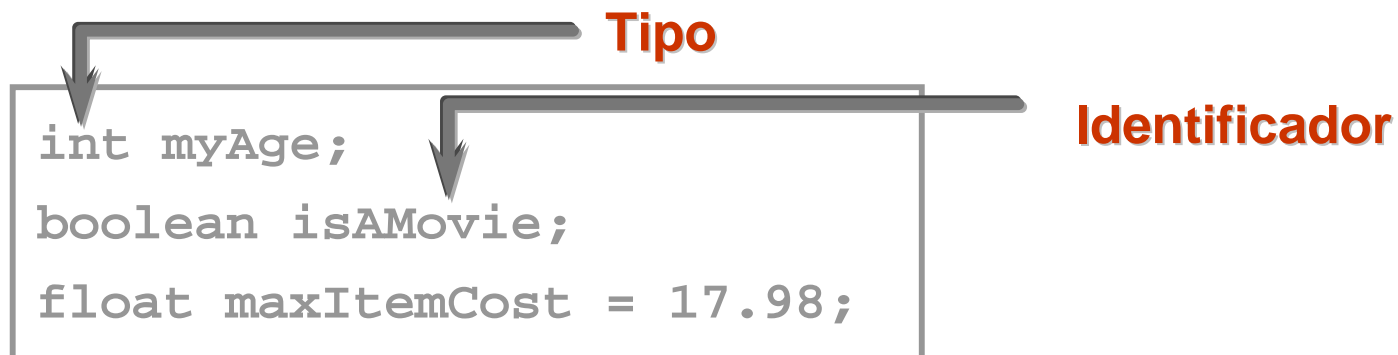
- Descrever os tipos de dados primitivos
- Declarar e inicializar variáveis primitivas
- Utilizar operadores para manipular o valor de uma variável primitiva

Tópicos

- O Java oferece primitivas para os tipos de dados básicos.
- As primitivas são a fundação para armazenar e utilizar informação.
- Declarar e inicializar primitivas é a base da construção de tipos definidos pelo utilizador.
- Os operadores manipulam dados e objectos.
- Aceitam um ou mais argumentos e produzem um valor.
- Java oferece 44 operadores diferentes.
- Alguns operadores alteram o valor do operando.

Variáveis

- A variável é a unidade básica de armazenamento.
- As variáveis devem ser declaradas explicitamente.
- Cada variável tem um tipo, um identificador, e um âmbito.
- As variáveis podem ser inicializadas.



Nomes de Variáveis

- Os nomes das variáveis devem começar por uma letra do alfabeto, um *underscore*, ou um \$.
- Os outros caracteres podem incluir dígitos.
- Deve-se utilizar nomes elucidativos para as variáveis; por exemplo,
customerFirstName, ageNextBirthday.



a	item_Cost
itemCost	_itemCost
item\$Cost	itemCost2



item#Cost	item-Cost
item*Cost	abstract
2itemCost	

Palavras Reservadas

`boolean`
`byte`
`char`
`double`
`float`
`int`
`long`
`short`
`void`

`false`
`null`
`true`

`abstract`
`final`
`native`
`private`
`protected`
`public`
`static`
`synchronized`
`transient`
`volatile`

`break`
`case`
`catch`
`continue`
`default`
`do`
`else`
`finally`
`for`
`if`
`return`
`switch`
`throw`
`try`
`while`

`class`
`extends`
`implements`
`interface`
`throws`

`import`
`package`

`instanceof`
`new`
`super`
`this`

Tipos de Variáveis

- Oito tipos de dados primitivos:
 - Seis tipos numéricos
 - Tipo *char*, para caracteres
 - Tipo Booleano, para valores verdadeiro ou falso
- Tipos definidos pelo utilizador
 - Classes
 - Interfaces
 - Arrays

Tipos de Dados Primitivos

Integer	Floating Point	Character	True False
byte short int long	float double	char	boolean
1, 2, 3, 42 07 0xff	3.0 .3337 4.022E23	'a' '\141' '\u0061' '\n'	true false

Declaração de Variáveis

- A forma básica de declaração de uma variável:

`tipo identifier [= valor]`

```
public static void main(String[] args) {  
    int itemsRented;  
    float itemCost;  
    int i, j, k;  
    double interestRate;  
}
```

- As variáveis podem ser inicializadas quando declaradas.

Declaração de Variáveis

- As variáveis locais estão contidas apenas num método ou bloco de código.
- As variáveis locais devem ser inicializadas antes de ser usadas.

```
class Rental {  
    private int instVar;        // instance variável  
    public void addItem() {  
        float itemCost = 3.50; // local variável  
        int numOfDays = 3;     // local variável  
    }  
}
```

Literais Númericos

Literais Inteiros

```
0 1 42 -23795 (decimal)
02 077 0123 (octal)
0x0 0x2a 0X1FF (hex)
365L 077L 0x1000L (long)
```

Literais Floating-point

```
1.0 4.2 .47
1.22e19 4.61E-9
6.2f 6.21F
```


Literais não-Númericos

Literais Booleanos

```
true false
```

Literais Character

```
'a' '\n' '\t' '\077'  
'\u006F'
```

Literais String

```
"Hello, world\n"
```

Exercício: Declaração de variáveis

- Encontrar os erros no código abaixo e corrigi-los.

```
1  byte sizeof = 200;  
2  short mom = 43;  
3  short hello mom;  
4  int big = sizeof * sizeof * sizeof;  
5  long bigger = big + big + big      // ouch  
6  double old = 78.0;  
7  double new = 0.1;  
8  boolean consequence = true;  
9  boolean max = big > bigger;  
10 char maine = "New England state";  
11 char ming = 'd';
```

Operadores

Cinco tipos de operadores:

- Atribuição
- Aritméticos
- Manipulação de bits
- Relacionais
- Booleanos

Operador de Atribuição

- A expressão da direita é atribuída à variável da esquerda:

```
int var1 = 0, var2 = 0;  
var1 = 50;           // var1 now equals 50  
var2 = var1 + 10;   // var2 now equals 60
```

- A expressão da direita é sempre avaliada antes da atribuição.
- As atribuições podem ser agrupadas:

```
var1 = var2 = var3 = 50;
```

Operadores Aritméticos

- Realizam operações aritméticas básicas
- Operam sobre variáveis e literais numéricos

```
int a, b, c, d;  
a = 2 + 2;    // addition  
b = a * 3;    // multiplication  
c = b - 2;    // subtraction  
d = b / 2;    // division  
e = b % 2;    // returns the remainder of division
```

Operadores Aritméticos...

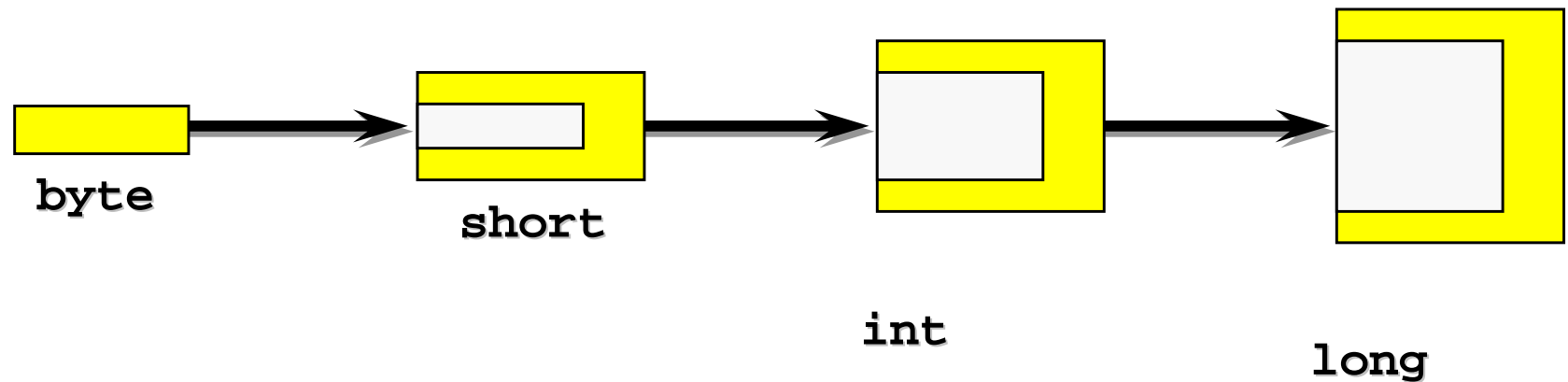
- A maioria das operações resultam num int ou long:

```
byte b1 = 1, b2 = 2, b3;  
b3 = b1 + b2;      // error: result is an int  
                  // b3 is byte
```

- Valores byte, char, e short são promovidos a int antes da operação.
- Se algum argumento for long, o outro é promovido a long, e o resultado é long.

Conversões e Casts

- O Java converte automaticamente valores de um tipo numérico para outro tipo maior.



- O Java não faz automaticamente o “downcast.”



Incrementar e Decrementar

- O operador ++ incrementa 1 unidade:

```
int var1 = 3;
var1++;           // var1 now equals 4
```

- O operador ++ pode ser usado de duas maneiras:

```
int var1 = 3, var2 = 0;
var2 = ++var1;    // Prefix: Increment var1 first,
                  // then assign to var2.
var2 = var1++;   // Postfix: Assign to var2 first,
                  // then increment var1.
```

- O operador -- decrementa 1 unidade.

Comparações

- Operadores relacionais e de igualdade:

>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to
==	equal to
!=	not equal to

```
int var1 = 7, var2 = 13;
boolean res = true;
res = (var1 == var2);      // res now equals false
res = (var2 > var1);      // res now equals true
```

Operadores Lógicos

- Os resultados de expressões Booleanas podem ser combinados usando operadores lógicos:

&&	&	e (with / without short-circuit evaluation)
		or (with / without short-circuit evaluation)
^		exclusive or
!		not

```
int var0 = 0, var1 = 1, var2 = 2;
boolean res = true;
res = (var2 > var1) & (var0 == 3);    // now false
res = !res;                            // now true
```

Atribuição Composta

- O operador de atribuição pode ser combinado com qualquer operador binário convencional:

```
double total=0, num = 1;
double percentage = .50;
...
total  = total + num;    // total is now 1
total += num;           // total is now 2
total -= num;           // total is now 1
total *= percentage;    // total is now .5
```

Precedência de Operadores

Order	Operators	Comments	Assoc.
1	<code>++ -- + - ~</code> <code>!(type)</code>	Unary operadores	R
2	<code>* / %</code>	Multiply, divide, remainder	L
3	<code>+ - +</code>	Add, subtract, add string	L
4	<code><< >> >>></code>	Shift (>>> is zero-fill shift)	L
5	<code>< > <= >=</code> <code>instanceof</code>	Relational, tipo compare	L
6	<code>== !=</code>	Equality	L
7	<code>&</code>	Bit/logical e	L
8	<code>^</code>	Bit/logical exclusive OR	L
9	<code> </code>	Bit/logical inclusive OR	L
10	<code>&&</code>	Logical e	L
11	<code> </code>	Logical OR	L
12	<code>? :</code>	Conditional operador	R
13	<code>= op=</code>	Assignment operadores	R

Precedências

- A precedência de um operador determina a ordem pela qual os operadores são executados:

```
int var1 = 0;  
var1 = 2 + 3 * 4;    // var1 now equals 14
```

- Operadores com a mesma precedência são executados da esquerda para a direita (ver nota):

```
int var1 = 0;  
var1 = 12 - 6 + 3;  // var1 now equals 9
```

- Os parêntesis permitem alterar a ordem definida.

Concatenação de String's

- O operador + cria e concatena strings:

```
String name = "Jane ";
String lastName = "Hathaway";
String fullName;
name = name + lastName;           // name is now
                                   // "Jane Hathaway"
                                   //      OR
name += lastName;                 // same result
fullName = name;
```

Resumo

- O Java tem oito tipos de dados primitivos.
- Uma variável deve ser declarada antes de ser usada.
- O Java dispõe de um bom conjunto de operadores.
- Casting explícitos podem ser necessários se utilizar tipos de dados menores do que int.
- Os operadores + e += podem ser usados para criar e concatenar strings.

Exercícios...



Universidade do Porto
Faculdade de Engenharia
FEUP

Instruções de Controlo de Fluxo

Objetivos

Ser capaz de:

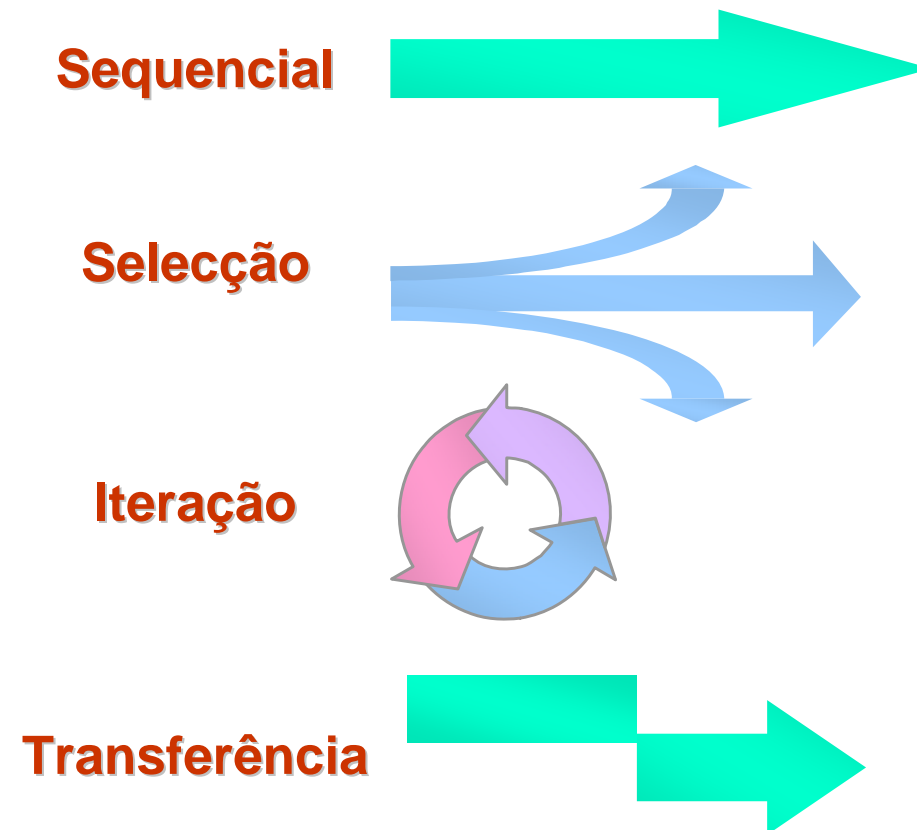
- Utilizar construções para tomar decisões
- Realizar ciclos de operações

Tópicos

- O código por defeito executa sequencialmente.
- Código mais complexo exige uma execução condicional.
- Existem instruções que necessitam de ser executadas repetidamente.
- O Java dispõe de mecanismos de controlo standard.

Tipos Básicos de Controle

- Controle de fluxo pode ser categorizado em quatro tipos:



Controlo de Fluxo em Java

- Agrupar instruções utilizando chavetas para formar uma instrução composta, i.e. um bloco.
- Cada bloco é executado como uma única instrução dentro da estrutura de controlo de fluxo.

```
{  
    boolean finished = true;  
    System.out.println("i = " + i);  
    i++;  
}
```

if ... else

Forma geral:

```
if ( boolean_expr )
    statement1;
[else]
    statement2;
```

Exemplos:

```
if (i % 2 == 0)
    System.out.println("Even");
else
    System.out.println("Odd");
...
```

```
if (i % 2 == 0) {
    System.out.println(i);
    System.out.println(" is even");
}
```

if...if...if...else if...else

```
if (speed >= 25)
    if (speed > 65)
        System.out.println("Speed over 65");
    else
        System.out.println("Speed over 25");
else
    System.out.println("Speed under 25");
```

```
if (speed > 65)
    System.out.println("Speed over 65");
else if (speed >= 25)
    System.out.println("Speed over 25");
else
    System.out.println("Speed under 25");
```

Operador Condicional (? :)

- (boolean_expr ? expr1 : expr2)

```
boolean_expr ? expr1 : expr2
```

- É uma alternativa útil ao if...else:
- Se boolean_expr=true, o resultado é expr1, senão o resultado é expr2:

```
int val1 = 120, val2 = 0;  
int highest;  
highest = (val1 > val2) ? 100 : 200;  
System.out.println("Highest value is " + highest);
```


Exercício: Descubra os Erros!

```
int x = 3, y = 5;
if (x >= 0)
    if (y < x)
        System.out.println("y is less than x");
else
    System.out.println("x is negative");
```

1

```
int x = 7;
if (x = 0)
    System.out.println("x is zero");
```

2

```
int x = 15, y = 24;
if ( x % 2 == 0 && y % 2 == 0 );
    System.out.println("x and y are even");
```

3

switch...case

- É útil para seleccionar um entre vários valores inteiros alternativos

```
switch ( integer_expr ) {  
  
    case constant_expr1:  
        statement1;  
        break;  
  
    case constant_expr2:  
        statement2;  
        break;  
  
    [default:  
        statement3;  
        break; ]  
  
}
```

switch...case

- As etiquetas de case devem ser constantes.
- Utilizar break para saltar fora do switch.
- Dar sempre uma alternativa default.

```
switch (choice) {  
    case 37:  
        System.out.println("Coffee?");  
        break;  
  
    case 45:  
        System.out.println("Tea?");  
        break;  
  
    default:  
        System.out.println("???");  
        break;  
}
```

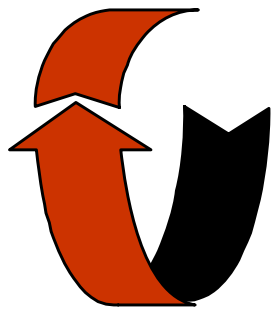
Ciclos

- Em Java existem três tipos de ciclos:
 - while
 - do...while
 - for
- Todos os ciclos têm quatro partes:
 - Inicialização
 - Iteração
 - Corpo
 - Terminação

while...

- O while é o mais simples de todos os ciclos:
- Exemplo:

```
while ( boolean_expr )  
    statement;
```

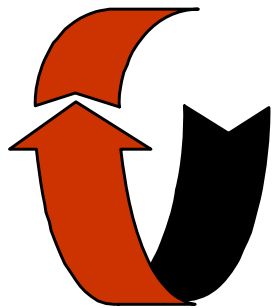


```
int i = 0;  
while (i < 10) {  
    System.out.println("i = " + i);  
    i++;  
}
```

do...while

- Os ciclos do...while têm o teste no fim do ciclo:
- Exemplo:

```
do
    statement;
while ( termination );
```



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

for...

- Os ciclos for são os mais comuns:

```
for ( initialization; termination; iteration )  
    statement;
```

- Exemplo:

```
for (i = 0; i < 10; i++)  
    System.out.println(i);
```

- Qual o ciclo *while* equivalente?

for...

- Podem ser declaradas variáveis na parte de inicialização do ciclo for:

```
for (int i = 0; i < 10; i++)  
    System.out.println("i = " + i);
```

- As partes de inicialização e iteração podem consistir de uma lista de expressões separadas por vírgulas:

```
for (int i = 0, j = 10; i < j; i++, j--) {  
    System.out.println("i = " + i);  
    System.out.println("j = " + j);  
}
```


Exercício: Descubra os Erros!

```
int x = 10;
while (x > 0);
    System.out.println(x--);
System.out.println("We have lift off!");
```

1

```
int x = 10;
while (x > 0)
    System.out.println("x is " + x);
x--;
```

2


```
int sum = 0;
for (; i < 10; sum += i++);
System.out.println("Sum is " + sum);
```

3

break

- Interrompe um ciclo ou uma instrução switch:
- Transfere o controlo para a primeira instrução depois do corpo do ciclo ou instrução switch
- Pode simplificar o código

```
...  
while (age <= 65) {  
    balance = (balance+payment) * (1 + interest);  
    if (balance >= 250000)  
        break;  
    age++;  
}  
...
```

A diagram consisting of a thick grey L-shaped line. It starts at the right side of the `break;` line, goes down to the level of the closing curly brace `}`, then turns left and goes to the left side of the first line after the loop, which is `...`. This indicates that the `break` statement exits the loop and jumps to the first statement following the loop's closing brace.

continue

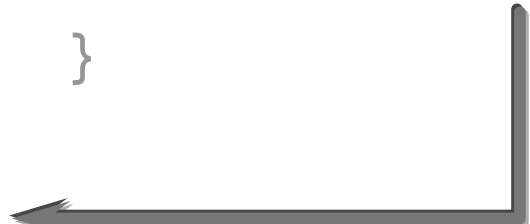
- Apenas pode ser usado em ciclos
- Abandona a iteração em curso e salta para a próxima iteração do ciclo

```
...  
for (int year = 2000; year < 2099; year++) {  
    if ((year % 100 == 0) && (year % 400 != 0))  
        continue;  
    if (year % 4 == 0)  
        System.out.println(year);  
}  
...
```

labeled break, continue

- Pode ser usado para saltar fora de ciclos encaixados, ou continuar um ciclo exterior ao ciclo corrente

```
outer_loop:  
for (int i = 0; i < 10; i++) {  
    for (int j = 0; j < 5; j++) {  
        System.out.println(i, j);  
        if (i + j > 7)  
            break outer_loop;  
    }  
}  
...  
...
```

A diagram consisting of a thick grey L-shaped arrow. The vertical part of the arrow starts at the line containing the `break outer_loop;` statement and points downwards to the closing curly brace of the inner loop. The horizontal part of the arrow then points leftwards to the closing curly brace of the outer loop, illustrating the jump from the inner loop to the outer loop.

Resumo

- A instrução if...else é a forma principal de implementar decisões.
- Java também dispõe de instrução switch.
- Java oferece três instruções de ciclos:
 - while
 - do...while
 - for
- A utilização de break e continue deve ser feita criteriosamente.

Exercícios...



Revisão de Conceitos de Orientação por Objectos

Tópicos

- Objectos: estado, comportamento
- Classes
- Encapsulamento
- Agregação: hierarquia de objectos
- Herança: hierarquia de classes
- Polimorfismo

Orientação por Objectos

- OO é um paradigma diferente para desenho e programação de software
- OO baseia-se na construção de modelos de objectos reais
- OO cria programas que são reutilizáveis e facilmente adaptáveis
- Os objects são autónomos e incluem informação e comportamento

O que é um Objecto?

- Definição filosófica: uma entidade que pode ser identificada
- Na terminologia
 - OO: uma abstracção de um objecto real
 - empresarial: uma entidade relevante para o domínio de aplicação
 - software: uma estrutura de dados e as funções associadas

Os Objectos executam Operações

- Um objecto existe para contribuir com funcionalidade (comportamento) a um sistema.
- Cada comportamento distinto é dado o nome de operação.



Objecto:
A minha caneta azul



Operação:
escrever



Objecto:
Caixa Multibanco



Operação:
levantamento

Os Objectos memorizam Valores

- Os objects têm conhecimento (informação) sobre o seu estado actual.
- Cada elemento de informação é dado o nome de atributo.



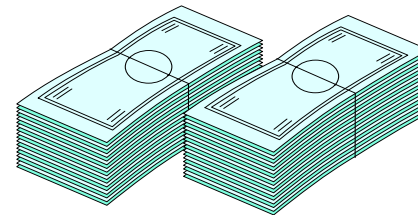
Objecto:
A minha caneta azul



Atributo:
Volume de tinta



Objecto:
Caixa Multibanco



Atributo:
Dinheiro levantado

Os Objectos são Abstracções

- No modelo de um objecto, apenas é necessário incluir operações e atributos que são importantes para o problema em questão.

Exemplo de uma operação que não interessa incluir:

- apontar-a

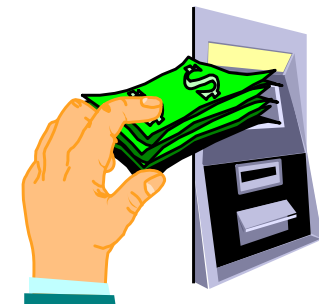


Exemplos de atributos que não interessam incluir:

- comprimento do bico
- fabricante da tinta
- idade

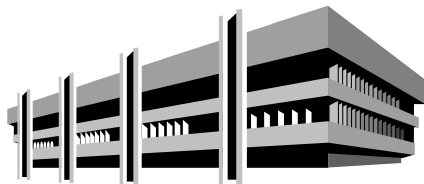
Encapsulamento

- O encapsulamento permite ocultar como as coisas funcionam e o que se sabe para além da interface—as operações de um objecto.
- Uma caixa Multibanco é um objecto que entrega dinheiro aos seus utilizadores:
 - A caixa MB encapsula isto para os seus utilizadores.
 - Violar o encapsulamento é considerado um roubo ao banco.
- Violar o encapsulamento em programação orientada por objectos é impossível.

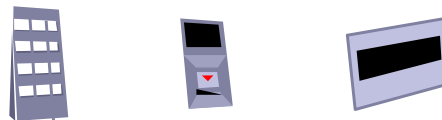


Hierarquias de objectos

- Os objectos são compostos por outros objectos.
- Os objectos podem fazer parte de outros objectos.
- Esta relação entre objectos é conhecida por **agregação**.



Um banco pode ser um objecto.



Uma caixa MB pode ter um teclado, leitor de cartões, dispensador de notas, todos podendo ser objectos.



Um banco pode ter uma caixa MB que também pode ser um objecto.

O que é uma Classe?

- Uma classe é uma especificação de objectos.
- Uma definição de uma classe especifica as operações e atributos para todas as instâncias de uma classe.



**Quando se cria a ‘minha caneta azul’, não é necessário especificar as suas operações e atributos.
Basta simplesmente indicar a classe a que pertence.**

Porque necessitamos de classes?

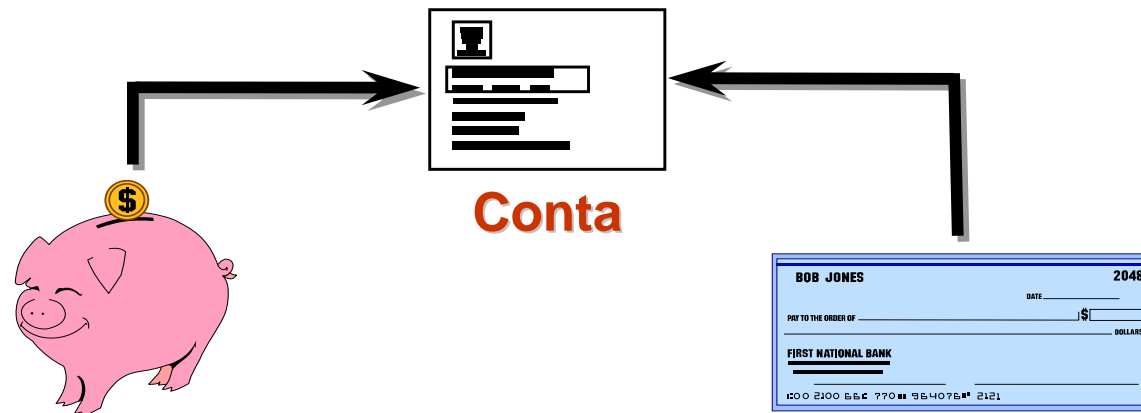
- Uma classe descreve o tipo de um objecto.
- Uma classe define o comportamento (operações) e estrutura (atributos) de um grupo de objectos:
 - Pode-se reduzir a complexidade utilizando classes.
 - No mundo existem imensos objectos, razão pela qual as pessoas os agrupam em tipos.
 - Se se compreender o tipo, pode-se aplicá-lo a vários objectos.

Classes versus Objects

- As classes são definições estáticas que nos permitem compreender todos os objectos de uma classe.
- Os objectos são as entidades dinâmicas que existem no mundo real e em suas simulações.
- Nota—em OO as pessoas frequentemente utilizam ambas as palavras classes e objectos de forma indiferente; é necessário utilizar o contexto para distinguir entre os dois significados possíveis.

Herança

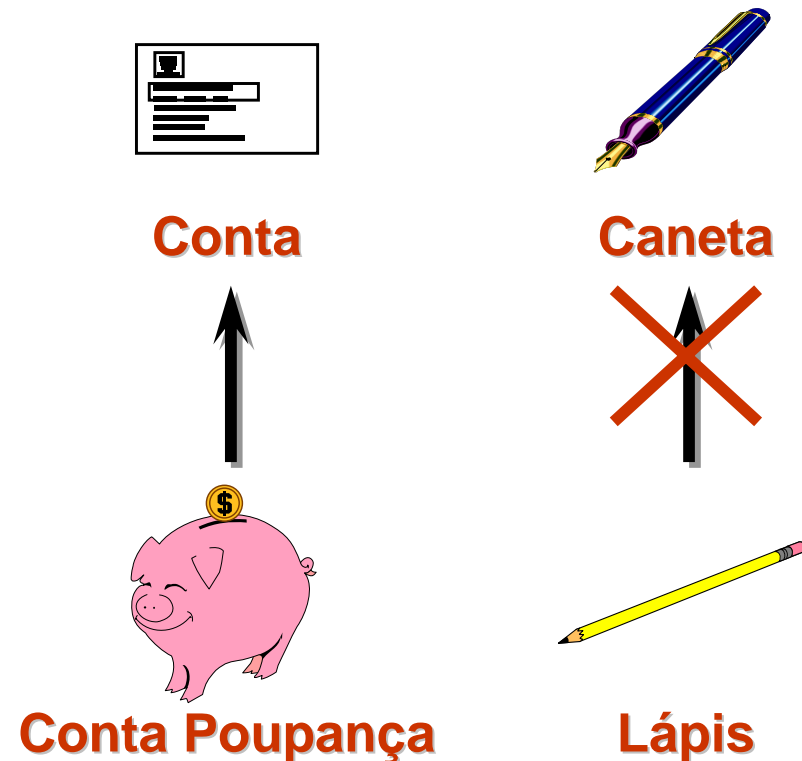
- Podem existir semelhanças entre classes distintas.
- Deve-se definir as propriedades comuns (atributos, operações) entre classes numa superclasse comum.



- **Conta Poupança** **Conta Depósitos à Ordem**
As subclasses utilizam herança para incluir as propriedades comuns entre elas.

Relação “Is-a-Kind-of”

- Um objecto de uma subclasse “é-um-tipo-de” objecto de uma superclasse.
- Uma subclasse deve ter todo o comportamento da superclasse.

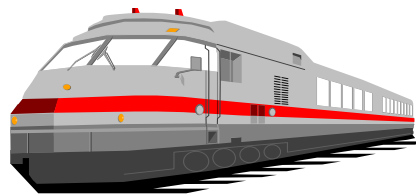
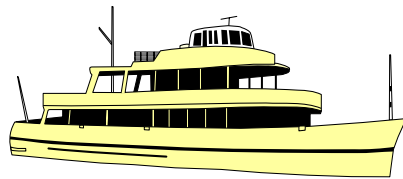


Polimorfismo

- O polimorfismo é a capacidade de um único nome poder referir objectos de classes diferentes, se relacionadas por uma superclasse comum
- O polimorfismo surge quando a linguagem de programação simultaneamente suporta herança e a associação dinâmica de tipos (*late binding*)

Polimorfismo...

- O polimorfismo permite que uma operação possa existir em diferentes classes.
- Cada operação tem um mesmo significado mas é executada de forma particular.



Transportar passageiros

Resumo

- Um objecto é uma abstracção de objecto real.
- Uma classe é um ‘molde’ ou ‘fôrma’ de objectos.
- As classes formam árvores de herança; as operações definidas numa classe são herdadas por todas as suas subclasses.
- O polimorfismo liberta quem invoca uma operação de conhecer a classe exacta do objecto que a irá receber.



Manipulação de Classes e Objectos

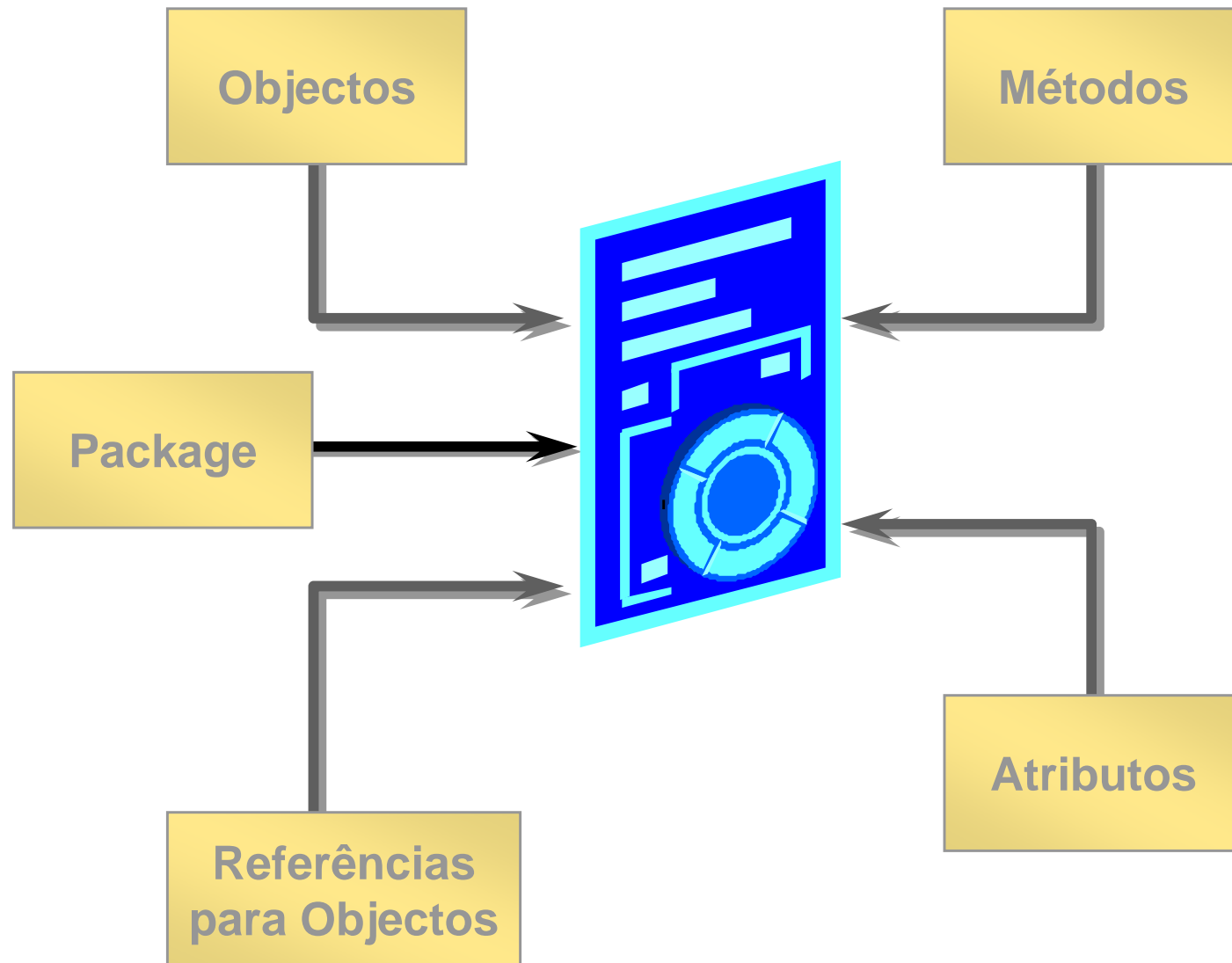
Objectivos

- Utilizar packages para agrupar classes relacionadas
- Definir variáveis e métodos de instâncias
- Criar objectos e invocar métodos
- Utilizar as palavras public, private e protected
- Redefinir métodos de uma classe (overloading)
- Escrever construtores
- Utilizar variáveis e métodos de classes

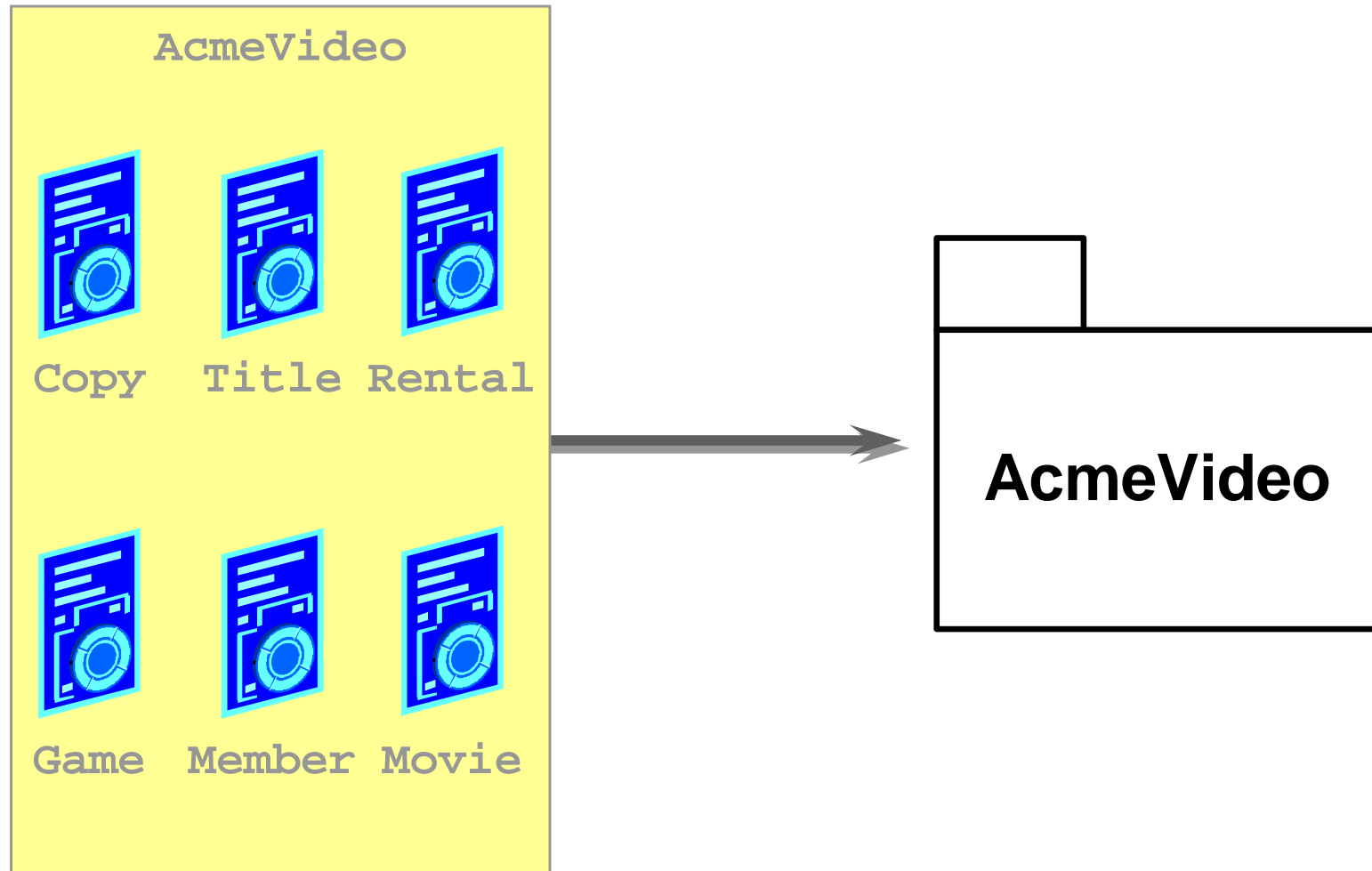
Tópicos

- As classes definem as características, atributos e comportamento dos objectos.
- Todo o código Java reside em classes.
- Toda a informação dos objectos é armazenada em variáveis.
- Os packages auxiliam a controlar o acesso a classes.
- O ‘overloading’ permite ter interfaces simples.
- Os construtores garantem consistência na criação de objectos.

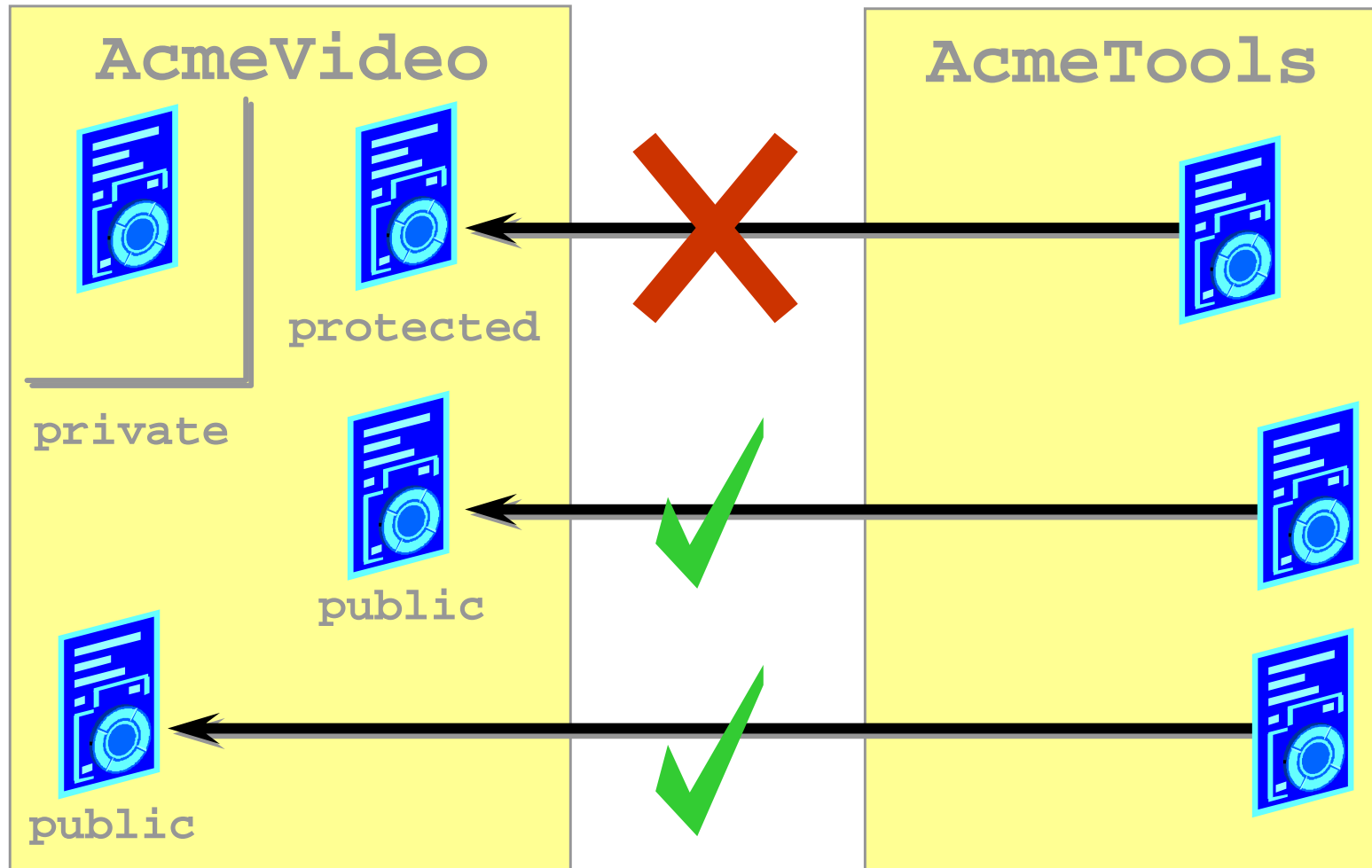
Classes Java



Packages

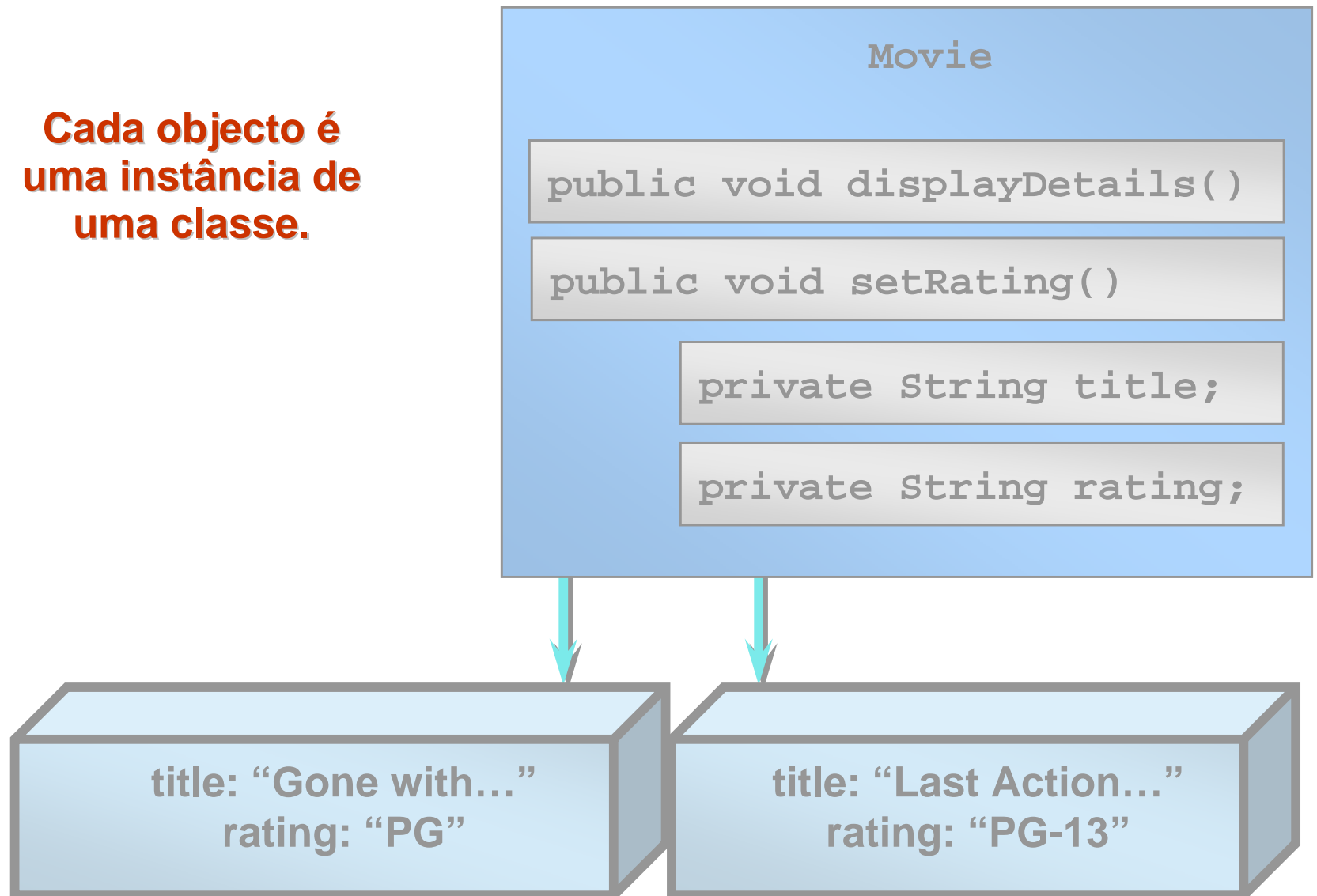


Controlo de Acesso



Classes e Objectos

**Cada objecto é
uma instância de
uma classe.**



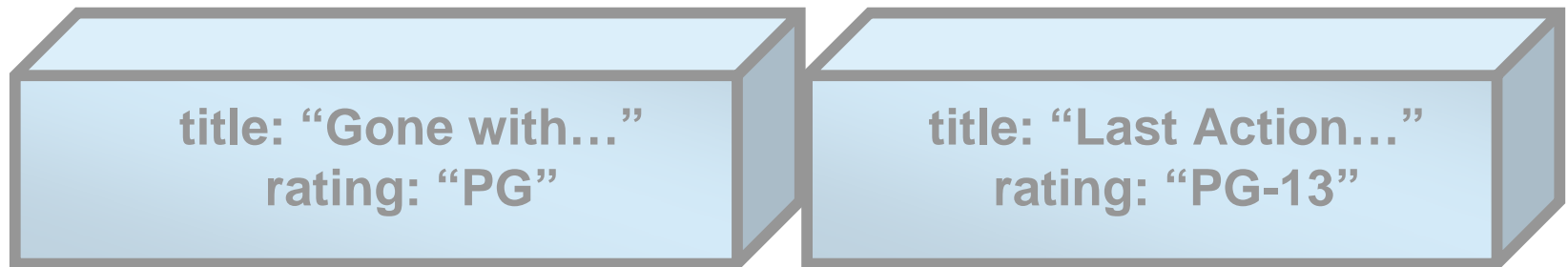
Criação de Objectos

- Os objectos são criados pelo operador new:

```
objectRef = new ClassName();
```

- Por exemplo, para criar dois objectos Movie:

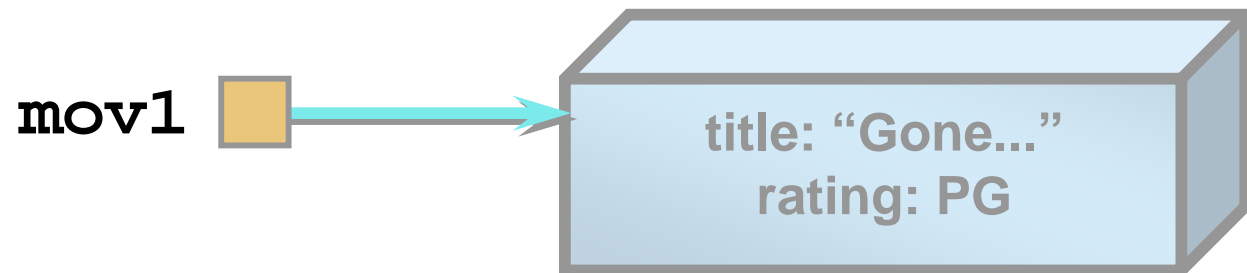
```
Movie mov1 = new Movie("Gone ...");  
Movie mov2 = new Movie("Last ...");
```



new

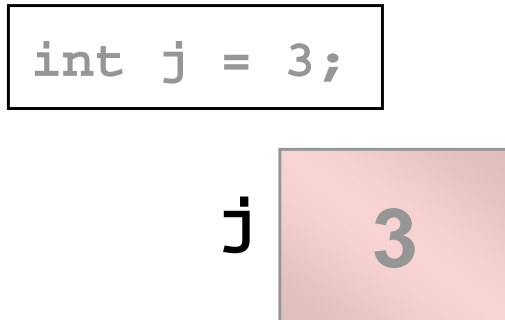
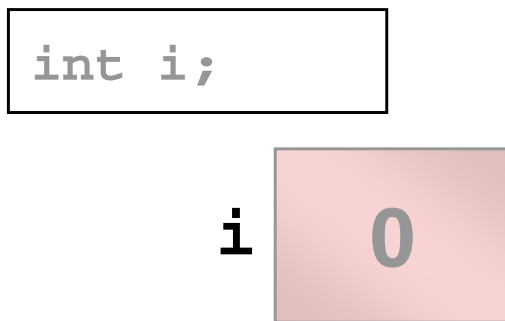
- O operador 'new' realiza o seguinte:
 - Aloca memória para o novo objecto
 - Invoca um método especial da classe para inicialização de objectos, um constructor
 - Retorna uma referência para o novo objecto

```
Movie mov1 = new Movie("Gone...");
```

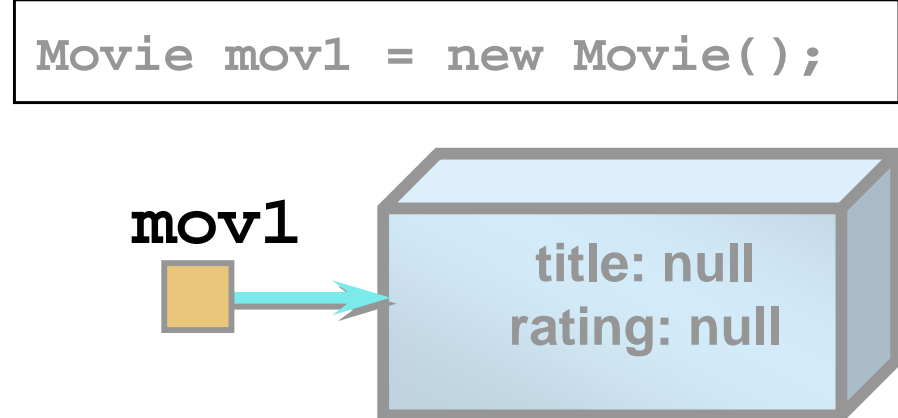
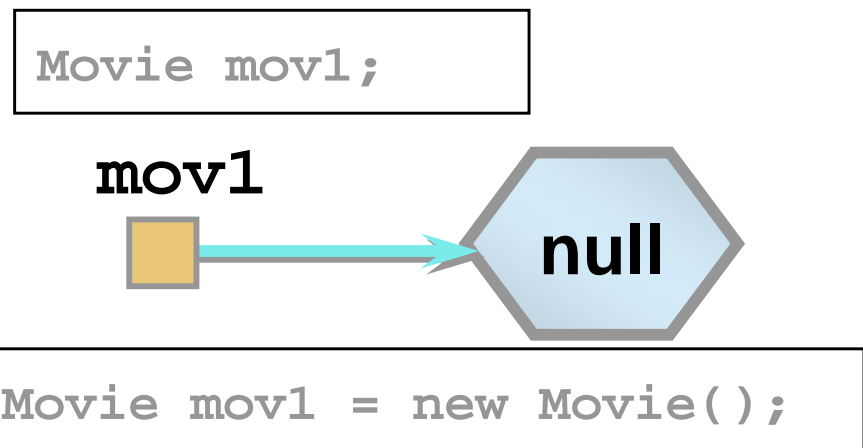


Objectos e valores primitivos

- As variáveis de tipos primitivos armazenam valores.



- As variáveis de tipos de classes armazenam referências para objectos.



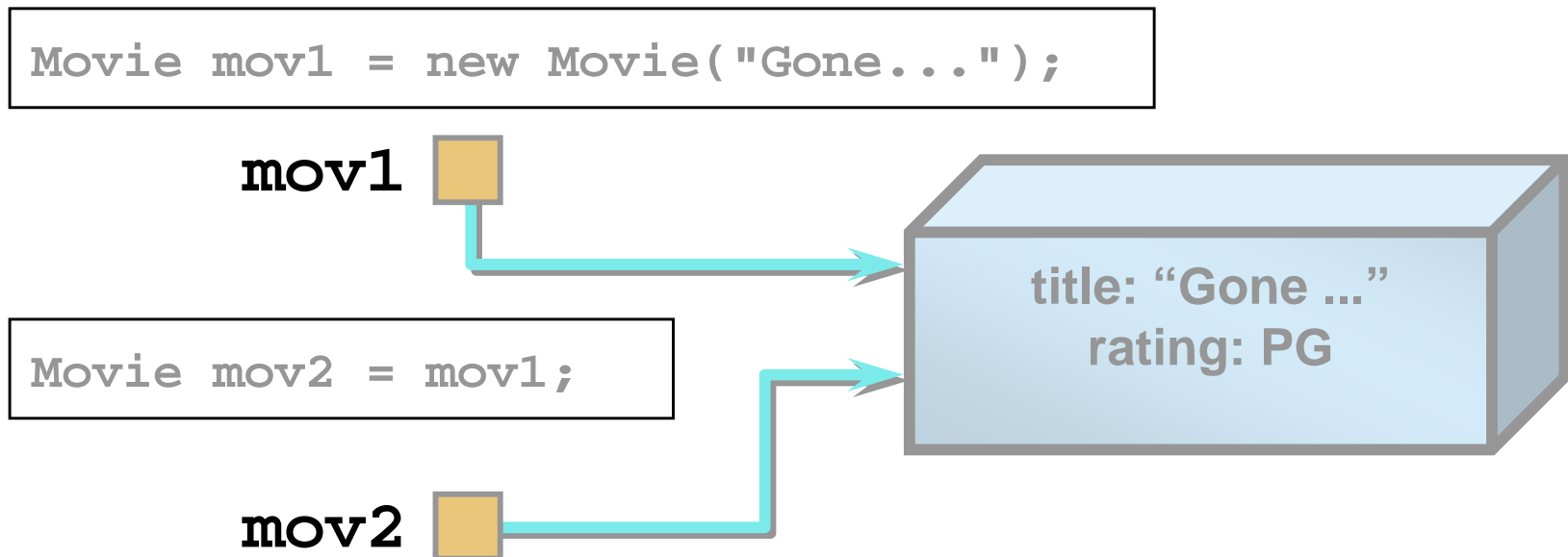
A referência null

- As referências para objectos têm o valor null até serem inicializadas.
- É possível comparar referências de objectos a null.
- Pode-se “eliminar” um objecto pela atribuição do valor null a uma referência.

```
Movie mov1 = null; //Declare object reference
...
if (mov1 == null) //Ref not initialized?
    mov1 = new Movie(); //Create a Movie object
...
mov1 = null; //Forget the Movie object
```

Atribuição de Referências

- A atribuição de uma referência a outra resulta em duas referências para o mesmo objecto:



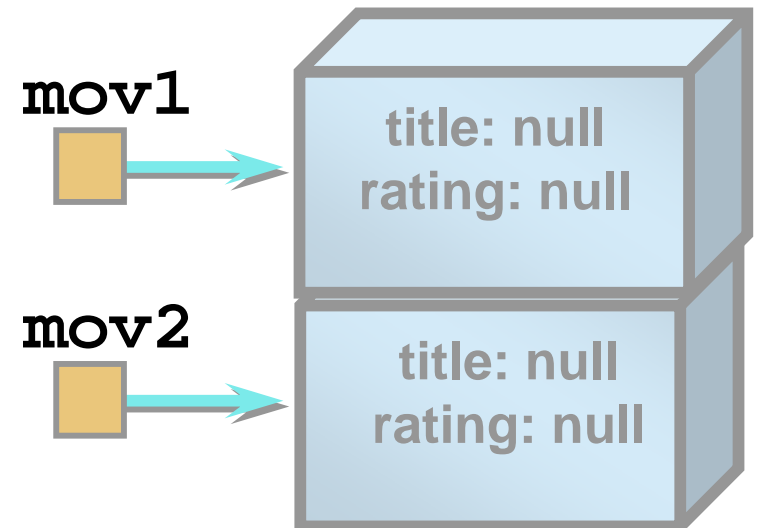
Variáveis de instância

- As variáveis de instância são declaradas na classe:

```
public class Movie {  
    public String title;  
    public String rating;  
    ...  
}
```

- Criação de vários 'movie':

```
Movie mov1 = new Movie();  
Movie mov2 = new Movie();
```



Acesso a variáveis de instância

- As variáveis públicas de instância podem ser acessadas através do operador '.' :

```
public class Movie {  
    public String title;  
    public String rating;  
    ...  
}
```

```
Movie mov1 = new Movie();  
mov1.title = "Gone ...";  
...  
if ( mov1.title.equals("Gone ... ") )  
    mov1.rating = "PG";
```

Será isto interessante? NÃO!

Criar e manipular objectos

```
public class Movie {  
    public String title;  
}
```

```
public class MovieTest {  
    public static void main(String[] args) {  
        Movie mov1, mov2;  
        ?  
        mov1.title = "Gone with the Wind";  
        mov2 = mov1;  
        mov2.title = "Last Action Hero";  
        System.out.println("Movie 1 is " + ? );  
        System.out.println("Movie 2 is " + ? );  
    }  
}
```

Métodos

- Um método é equivalente a uma função ou subrotina de outras linguagens:

```
modifier returnType methodName (argumentList) {  
    // method body  
    ...  
}
```

- Um método apenas pode ser definido na definição de uma classe.

Argumentos de Métodos

```
public void setRating(String newRating) {  
    rating = newRating;  
}
```

```
public void displayDetails() {  
    System.out.println("Title is " + title);  
    System.out.println("Rating is " + rating);  
}
```


Retorno de valores dum método

```
public class Movie {  
    private String rating;  
  
    ...  
    public String getRating () {  
        return rating;  
    }  
    public void setRating (String r) {  
        this.rating = r;  
    }  
}
```

Invocar métodos a uma instância

```
public class Movie {
    private String title, rating;
    public String getRating(){
        return rating;
    }
    public void setRating(String newRating){
        rating = newRating;
    }
}
```

```
Movie mov1 = new Movie();
```

```
...
```

```
if (mov1.getRating().equals("G"))
```

```
...
```

Operador '.' :

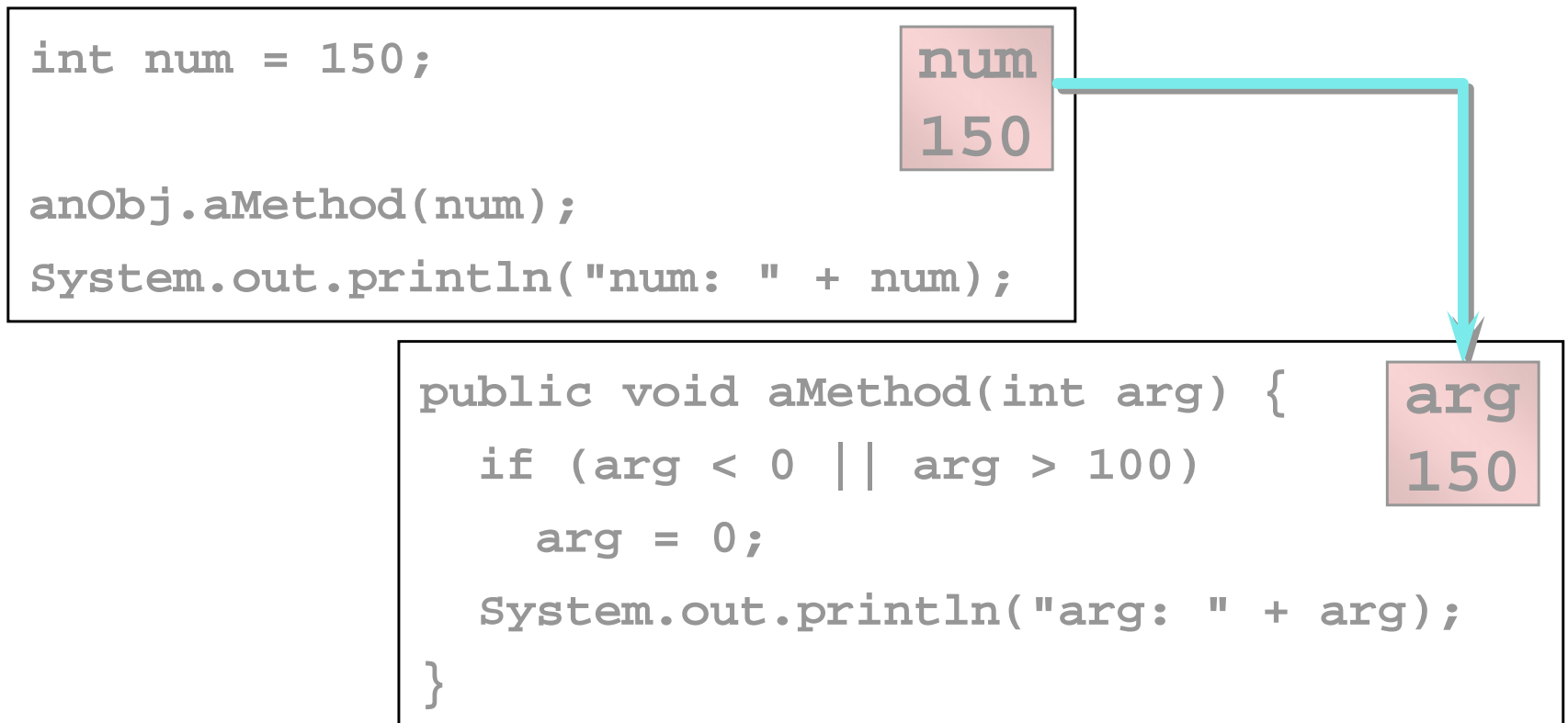
Encapsulamento

- As variáveis de instância devem ser declaradas `private`.
- Apenas métodos de instância podem ter acesso a variáveis de instância.
- O encapsulamento permite isolar a interface d uma classe da sua implementação interna.

```
Movie mov1 = new Movie();  
  
...  
if ( mov1.rating.equals("PG") )      // Error  
    mov1.setRating("PG");           // OK
```

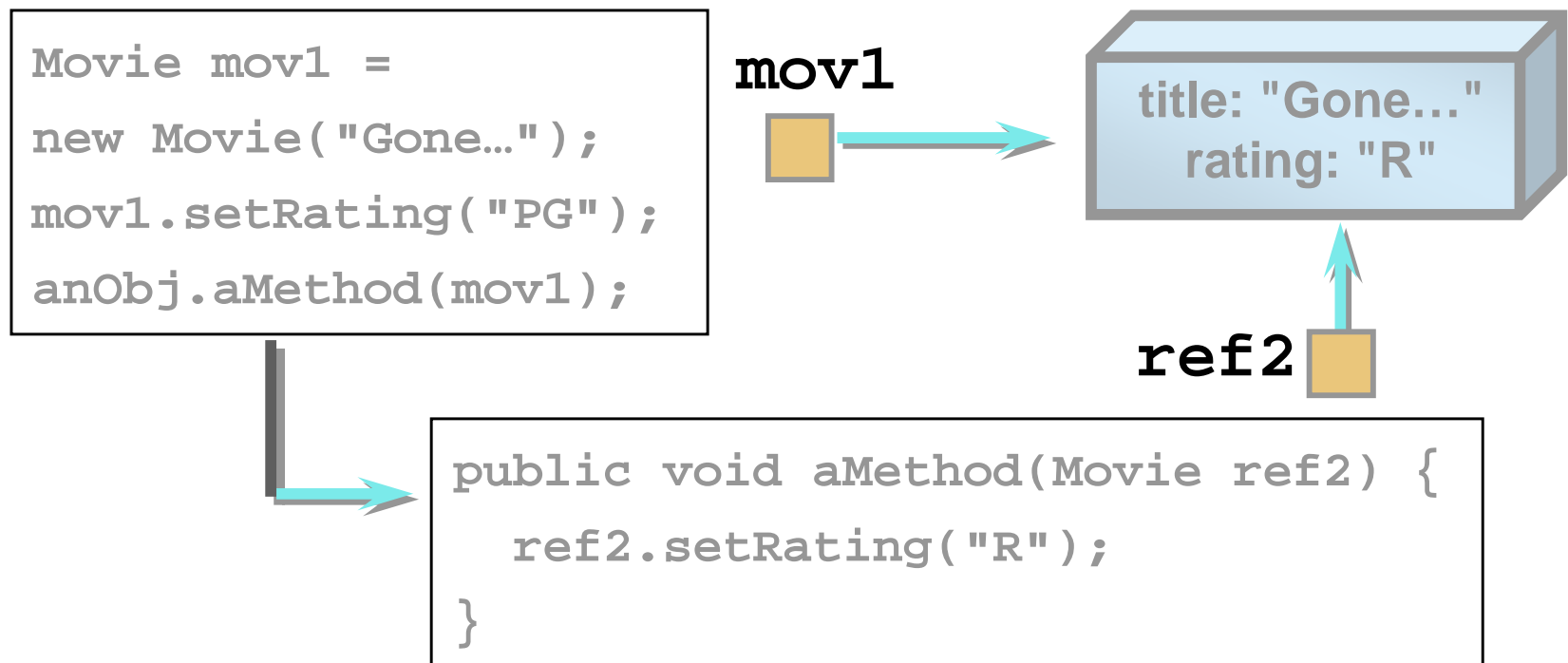
Passagem de valores a métodos

- Quando um valor primitivo é passado a um método, é gerada uma cópia do valor:



Passagem de objectos a métodos

- Quando um objecto é passado a um método, o argumento refere o objecto original:



‘Overloading’ de métodos

- Diversos métodos de uma classe podem ter o mesmo nome.
- Os métodos devem ter diferentes assinaturas.

```
public class Movie {  
    public void setPrice() {  
        price = 3.50;  
    }  
    public void setPrice(float newPrice) {  
        price = newPrice;  
    } ...  
}
```

```
Movie mov1 = new Movie();  
mov1.setPrice();  
mov1.setPrice(3.25);
```

Inicialização de atributos

- As variáveis de instância podem ser inicializadas na sua declaração.

```
public class Movie {  
    private String title;  
    private String rating = "G";  
    private int numOfOscars = 0;
```

- A inicialização é feita na criação do objecto.
- Inicializações mais complexas devem ser colocadas num método construtor.

Construtores

- Para uma inicialização adequada, a classe deve fornecer construtores.
- O construtor é invocado automaticamente quando o objecto é criado:
 - Normalmente declarado 'public'
 - Tem o mesmo nome da classe
 - Não especifica nenhum tipo de retorno
- O compilador automaticamente fornece um construtor por defeito sem argumentos.

Definição de Construtores

```
public class Movie {  
    private String title;  
    private String rating = "PG";  
  
    public Movie() {  
        title = "Last Action ...";  
    }  
    public Movie(String newTitle) {  
        title = newTitle;  
    }  
}
```



**A classe Movie
fornece dois
construtores**



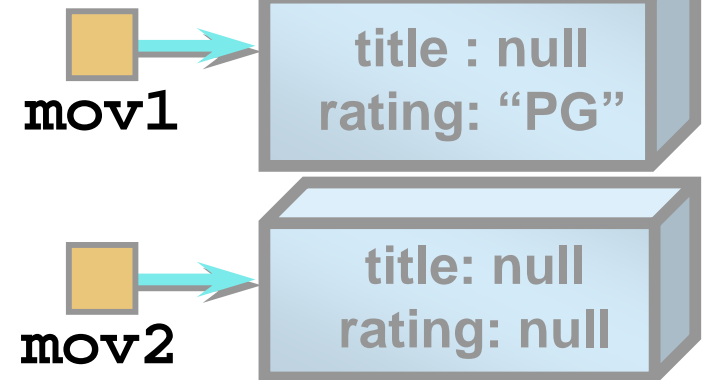
```
Movie mov1 = new Movie();  
Movie mov2 = new Movie("Gone ...");  
Movie mov3 = new Movie("The Good ...");
```

A referência 'this'

- Os métodos de instância recebem um argumento com o nome 'this', que refere para o objecto corrente.

```
public class Movie {  
    public void setRating(String newRating) {  
        this.rating = newRating; this    
    }  
}
```

```
void anyMethod() {  
    Movie mov1 = new Movie();  
    Movie mov2 = new Movie();  
    mov1.setRating("PG"); ...  
}
```



Partilha de código entre construtores

```
public class Movie {  
    private String title;  
    private String rating;  
  
    public Movie() {  
        this("G");  
    }  
    public Movie(String newRating) {  
        rating = newRating;  
    }  
}
```

Um construtor
pode invocar
outro através de
this()

```
Movie mov2 = new Movie();
```

Variáveis de Classe

- As variáveis de classe pertencem a uma classe e são comuns a todas as instâncias dessa classe.
- As variáveis de classe são declaradas como 'static' na definição da classe.

```
public class Movie {  
    private static double minPrice;    // class var  
    private String title, rating;     // inst vars
```



classe Movie



objectos Movie

Inicialização de variáveis de classe

- As variáveis de classe podem ser inicializadas na declaração.
- A inicialização é realizada quando a classe é carregada.

```
public class Movie {  
    private static double minPrice = 1.29;  
  
    private String title, rating;  
    private int length = 0;
```

Métodos de Classe

- Os métodos de classe são partilhados por todas as instâncias.
- São úteis para manipular variáveis de classe:

```
public static void increaseMinPrice(double inc) {  
    minPrice += inc;  
}
```

- Um método de classe pode ser invocado utilizando o nome da classe ou uma referência para um objecto.

```
Movie.increaseMinPrice(.50);  
mov1.increaseMinPrice(.50);
```

Métodos de classe ou de instância?

```
public class Movie {  
  
    private static float price = 3.50f;  
    private String rating;  
    ...  
    public static void setPrice(float newPrice) {  
        price = newPrice;  
    }  
    public float getPrice() {  
        return price;  
    }  
}
```

```
Movie.setPrice(3.98f);  
Movie mov1 = new Movie(...);  
mov1.setPrice(3.98f);  
float a = Movie.getPrice();  
float b = mov1.getPrice();
```

Exemplos de Java

- Exemplos de métodos e variáveis 'static':
 - main()
 - Math.sqrt()
 - System.out.println()

```
public class MyClass {  
  
    public static void main(String[] args) {  
        double num, root;  
        ...  
        root = Math.sqrt(num);  
        System.out.println("Root is " + root);  
    } ...  
}
```


Variáveis final

- Uma variável declarada 'final' é uma constante.
- Uma variável 'final' não pode ser modificada.
- Uma variável 'final' deve ser inicializada.
- Uma variável 'final' é normalmente pública para permitir acesso externo.

```
public final class Color {  
  
    public final static Color black=new Color(0,0,0);  
    ...  
}
```

Garbage Collection

- Quando todas as referências para um objecto são eliminadas, o objecto é marcado para ser destruído.
 - Garbage collection liberta a memória utilizada pelo objecto.
- Garbage collection é automática.
 - Não existe necessidade de intervenção do programador, mas não possui qualquer controlo sobre quando o objecto é realmente destruído



O método finalize()

- Se um objecto utilizar um outro recurso (p.e. Um ficheiro), o objecto deve libertá-lo.
- Pode ser fornecido um método finalize().
- O método finalize() é invocado antes do objecto ser destruído.

```
public class Movie {  
    ...  
    public void finalize() {  
        System.out.println("Goodbye");  
    }  
}
```

Resumo

- A definição de uma classe especifica as características comuns de um conjunto de objectos.
- Um objecto é uma instância de uma classe particular:
 - Criam-se objectos através do operador 'new'.
 - Manipula-se um objecto através da invocação de métodos públicos de instância.
- Os métodos de instância recebem a referência 'this'
- Os métodos podem ter diferentes implementações
- As classes fornecem um ou mais construtores para inicializar objectos.
- Podem ser definidos variáveis e métodos para implementar comportamentos globais à classe.

Exercícios...



Universidade do Porto

Faculdade de Engenharia

FEUP

Parte 2



Java Collections Framework (JCF)

Tipos abstractos de dados

- Estrutura de dados + Algoritmos
- Standard
- Fácil de compreender
- Eficiente
- Exemplos
 - Stack, queue, linked list

Desenho baseado em interfaces

- Separação entre interface e implementação
- Especificamente construído para a linguagem Java
- Polimorfismo
 - `List l = new LinkedList();`
 - `l.add()` invoca o método `add()` da classe `LinkedList`

Framework de Collections

- Interoperabilidade entre APIs diferentes
- Reduz o esforço de aprendizagem de APIs
- Reduz o esforço de desenho e implementação de APIs
- Promove reutilização de software

Objectivos da Collections Framework

- API pequena
 - Número de interfaces
 - Número de métodos por interface
 - Poucos conceitos
- Ter como base as colecções do Java (Vector, Hashtable)
- Permitir conversão para arrays Java

Disponibilidade da JCF

- Incluída desde o JDK 1.2
- <http://java.sun.com/docs/books/tutorial/collections/>

Desenho baseado em Interfaces

```
interface List {...}
```

```
class LinkedList implements List {...}
```

```
...
```

```
List l = new LinkedList();
```

```
l.add( new Date() );
```

```
Date d = (Date)l.get(0);
```

Interfaces principais

- Collection
- Set
- List
- Map
- SortedSet
- SortedMap

Interfaces e Classes Utilitárias

- Interfaces
 - Comparator
 - Iterator
- Classes
 - Collections
 - Arrays

Collection

- Um grupo de objectos
- Principais métodos:
 - `int size();`
 - `boolean isEmpty();`
 - `boolean contains(Object);`
 - `Iterator iterator();`
 - `Object[] toArray();`
 - `boolean add(Object);`
 - `boolean remove(Object);`
 - `void clear();`

Set

- Set herda de Collection
- Uma colecção de objectos não ordenados
- Não permite elementos duplicados
- Os mesmos métodos de Collection
 - A semântica é diferente; obriga a utilizar uma interface diferente.
- Implementada por AbstractSet, HashSet, TreeSet, ...

List

- List herda de Collection
- Uma coleção de objectos não ordenados
- Permite elementos duplicados
- Principais métodos:
 - `Object get(int);`
 - `Object set(int, Object);`
 - `int indexOf(Object);`
 - `int lastIndexOf(Object);`
 - `void add(int, Object);`
 - `Object remove(int);`
 - `List subList(int, int);`
 - `add()` inserts
 - `remove()` deletes
- Implementada por `AbstractList`, `ArrayList`, `LinkedList`, `Vector`

Map

- Map não herda de Collection
- Um objecto que mapeia chaves para valores
- Cada chave pode ter um valor associado
- Veio substituir a interface `java.util.Dictionary`
- Ordenação pode ser fornecida por classes de implementação
- Principais métodos:
 - `int size();`
 - `boolean isEmpty();`
 - `boolean containsKey(Object);`
 - `boolean containsValue(Object);`
 - `Object get(Object);`
 - `Object put(Object, Object);`
 - `Object remove(Object);`
 - `void putAll(Map);`
 - `void clear();`
- Implementada por `HashMap`, `Hashtable`, `Attributes`, `TreeMap`, ...

Acesso aos membros de um Map

- Métodos
 - `Set keySet();`
 - `Collection values();`
 - `Set entrySet();`
- Map.Entry
 - Objecto que contém o par chave-valor
 - `getKey()`, `getValue()`
- Thread safety
 - As colecções retornadas são criadas pelo map
 - Quando o map é alterado, a colecção também é alterada.

Iterator

- Representa um iterador para um ciclo
- Criado por `Collection.iterator()`
- Similar à Enumeration
 - Nomes melhorados
 - Permite a operação `remove()` no item corrente.
- Principais métodos:
 - `boolean hasNext()` devolve `true` se a iteração tem mais elementos
 - `Object next()` devolve o próximo elemento na iteração
 - `void remove()` remove o elemento corrente da coleção

ListIterator

- A interface ListIterator herda de Iterator
- Criado por List.listIterator()
- Adiciona métodos para
 - Visitar uma List em qualquer direcção
 - Modificar a List durante a iteração
- Métodos adicionados:
 - `hasPrevious()`, `previous()`
 - `nextIndex()`, `previousIndex()`
 - `set(Object)`, `add(Object)`

Set: Implementações

- HashSet
 - Um Set baseado numa hash table
- TreeSet
 - Uma implementação baseada numa árvore binária balanceada
 - Impõe ordenação dos elementos

List: Implementações

- ArrayList
 - Uma implementação unsynchronized, sem métodos legados, baseada num array de tamanho dinâmico, tal como Vector
- LinkedList
 - Uma implementação baseada numa lista duplamente ligada
 - Pode ter melhor performance do que a ArrayList se os elementos forem frequentemente inseridos/apagados no meio da lista
 - Útil para queues e double-ended queues (deques)
- Vector
 - Uma implementação synchronized baseada num array de tamanho dinâmico e com métodos legados adicionais.

Map: Implementações

- **HashMap**
 - Um Map implementado com base numa tabela de hash
 - Idêntico a Hashtable, mas suporta chaves e valores com null.
- **TreeMap**
 - Uma implementação baseada numa árvore binária balanceada
 - Impõe ordenação dos elementos
- **Hashtable**
 - Uma implementação synchronized baseada numa tabela de hash, com métodos "legados" adicionais.

Ordenação

- Disponibilizado por `Collections.sort()`
- `Arrays.sort(Object[])` permite ordenar Arrays
- `SortedSet`, `SortedMap` interfaces
 - Collections que permitem manter os seus elementos ordenados
 - Os iterators garantem uma visita ordenada
- Ordered Collection Implementations
 - `TreeSet`, `TreeMap`

Ordenação ...

- Comparable interface
 - Deve ser implementado por todos os elementos de SortedSet
 - Deve ser implementado por todas as chaves SortedMap
 - `int compareTo(Object o)`
 - Define uma ordem natural para os objectos de uma classe
- Comparator interface
 - Define uma função que compara dois objectos
 - Permite esquemas de ordenação diversos
 - `int compare(Object o1, Object o2)`

Operações não suportadas

- Uma classe de implementação pode decidir não suportar um determinado método da interface reportando então uma exceção em tempo de execução do tipo `UnsupportedOperationException`.

Modificação de Colecções

■ Modifiable/Unmodifiable

- Modifiable: colecções que suportam operações de modificação: `add()`, `remove()`, `clear()`
- Unmodifiable: colecções que não suportam operações de modificação

■ Mutable/Immutable

- Immutable: colecções que garantem que nenhuma modificação poderá ser efectuada por operações de interrogação: `iterator()`, `size()`, `contains()`
- Mutable: colecções que não garantem serem imutáveis.

Thread safety

- As Collections, por omissão, NÃO são thread-safe, por motivos de performance e simplicidade.
- Soluções:
 - Encapsulated Collections
 - Synchronized Collections
 - `List list = Collections.synchronizedList(new ArrayList(...));`
 - Unmodifiable Collections
 - `List list = Collections.unmodifiableList(new ArrayList(...));`
 - Fail-fast iterators

Classes Utilitárias

- Collections

- `sort(List)`
- `binarySearch(List, Object)`
- `reverse(List)`
- `shuffle(List)`
- `fill(List, Object)`
- `copy(List dest, List src)`
- `min(Collection)`
- `max(Collection)`
- `synchronizedX`, `unmodifiableX` factory methods

Classes Utilitárias ...

- Arrays (métodos aplicáveis a arrays)
 - `sort`
 - `binarySearch`
 - `equals`
 - `fill`
 - `asList` - retorna um `ArrayList` com os conteúdos do array

Exercícios...



Universidade do Porto
Faculdade de Engenharia
FEUP

Tratamento de Exceções

Excepções

- Permitem tratar (“catch”) de forma agradável os erros que podem acontecer durante a execução de um programa.
- Permitem especificam diferentes formas de tratamento de excepções distintas.
- Oferecem uma forma standard de gerar (“throw”) erros.
- Permitem representar como objectos os erros de um programa que podem ser recuperados (“exceptions”).
- Permitem criar uma hierarquia extensível de classes de excepções para um tratamento de erros preciso.

Tipos de Exceções/Erros

- Exceções (podem ser resolvidos pelo programa):
 - Erros de I/O (teclado/ floppy / disco / entrada de dados)
 - Erros de rede (internet, LAN)
 - Casting ilegal, desreferenciação de objectos (null), matemática
 - Índice de array / colecção fora dos limites
- Erros (não podem ser resolvidos de forma conveniente):
 - Falta de memória
 - Erro / bug / crash na Java Virtual Machine
 - Classes Java requeridas pelo programa corruptas

Hierarquia de herança de Exception

java.lang.Throwable

Error

ThreadDeath

OutOfMemoryError

VirtualMachineError

Exception

AWTException

IOException

FileNotFoundException

MalformedURLException

RemoteException

SocketException

RuntimeException

ArithmeticException

ClassCastException

IllegalArgumentException

IndexOutOfBoundsException

NullPointerException

UnsupportedOperationException

Exceções Checked vs. Runtime

- checked:
 - Podem ter sido causadas por algo fora do controlo do nosso programa; DEVEM ser tratadas pelo nosso código, senão o programa não compila.
- unchecked (runtime):
 - São culpa nossa!
 - (provavelmente) poderiam ter sido evitadas se fossem devidamente analisadas e tratadas no código (verificar situações de erro).
 - Não precisam de ser tratadas, mas farão com que o programa crash no caso de ocorrerem em runtime.

Gerar Exceções em Runtime

- Podem ser geradas em qualquer parte do código pelo programador.
- Não precisam de ser tratadas (handled) pelas chamadas que as apanham.

```
public Object get(int index) {  
    // verificar se argumento é válido  
    if (index < 0)  
        throw new IndexOutOfBoundsException("indice < 0!");  
  
    return dados[index];  
}
```

Gerar “Checked Exceptions”

- No cabeçalho do método devem ser especificados os tipos de exceções que ele pode originar (“throws”).
- Quem invocar um método que possa originar exceções deve tratá-las ou então passar essas exceções a quem o invocou.

```
public void readFile(String fileName) throws IOException {  
    if (!canRead(fileName))  
        throw new IOException("Can't read file!");  
    else  
        doSomething();  
}
```


Síntaxe

```
try
{
    codeThatMightCrash();
}
catch (KindOfException exceptionVarName)
{
    // code to deal with index exception

}
// optional!
finally {
    // code to execute after the "try" code,
    // or exception's "catch" code has finished running
}
```

Tratamento de múltiplas exceções

```
try {
    codeThatMightCrash();
    moreBadCode();
} catch (IndexOutOfBoundsException ioobe) {
    // code to deal with index exception
} catch (IOException ioe) {
    // optional; code to deal with i/o exception
} catch (Exception e) {
    // optional; code to deal with any other exception
} finally {
    // optional; code to execute after the "try" code,
    // or exception's "catch" code has finished running
}
```

Try/Catch: Exemplo 1

```
try {
    readFile("hardcode.txt");
} catch (IOException ioe) {
    // code could throw compile-time IOException; must catch.
    // I'll handle it by printing an error message to the user
    System.out.println("Unable to read file!");
} finally {
    // whether it succeeded or not, I want to close the file
    closeFile();
}
```

Try/Catch: Exemplo 2

```
while (true) {  
    int index = kb.readInt();  
    try {  
        Object element = myCollection.get(index);  
        break; // if I get here, it must have worked!  
    } catch (IndexOutOfBoundsException ioobe) {  
        // wouldn't have to catch this runtime exception...  
        System.out.print("Bad index; try again.");  
    }  
}
```

Exceções: “Boas maneiras”

- Imprimir uma mensagem de erro (`System.out.println`)
- Abrir uma caixa de erro (em GUIs)
- Voltar a perguntar ao utilizador (em erros de teclado)
- Tentar a operação novamente (para problemas de I/O)
- Corrigir o erro (nem sempre possível)
- Re-lançar a exceção (provavelmente alguém acima conseguirá tratar a exceção).

“Más maneiras” de tratar exceções

- Try-block muito pequenos.
- Try-block muito grandes.
- Bloco de instruções do catch muito genérico (Exception).
- Bloco de instruções do catch ineficaz ({} ?).
- Tratar uma exceção runtime quando podia ter sido evitada por verificação prévia (null, index).

A nossa classe Exception

```
public class DeckException
    extends RuntimeException {
    public DeckException(String message) {
        super(message);
    }

    public String toString() {
        return "Error in deck card: " + getMessage();
    }
}
```

Algumas observações...

- Não se conseguem tratar alguns erros: porquê?
- As exceções podem ocorrer em diversas áreas:
 - I/O
 - rede / internet
 - Invocação remota de código
 - Código com reflexão (“reflection”)

Prevenção / Debugging

- Obrigar os métodos a lançar “checked exceptions” obrigando-os a serem usados de forma mais cuidadosa.
- Técnica de debug interessante:
`new RuntimeException().printStackTrace();`
- Assertões (J2SE v1.4): testes booleanos

```
public void deposit(double amount) {  
    assert amount >= 0.0;
```
- debugger: Eclipse / JDB (JDK)
- logging, profiling...

Referências

- The Java Tutorial: Exception-Handling.
 - <http://java.sun.com/docs/books/tutorial/java/nutsandbolts/exception.html>

- The Java Tutorial: Handling Errors with Exceptions.
 - <http://java.sun.com/docs/books/tutorial/essential/exceptions/index.html>



Universidade do Porto

Faculdade de Engenharia

FEUP

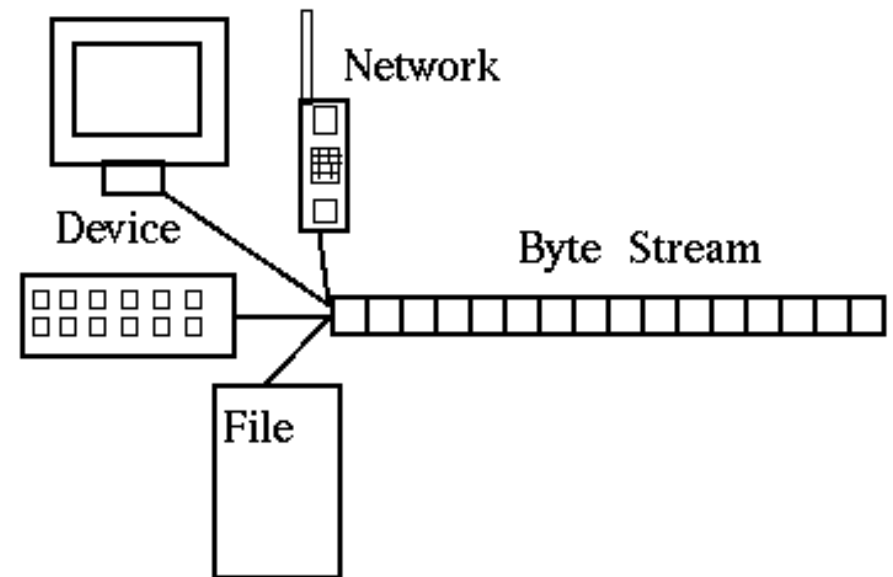
Entrada/Saída de dados

Objectivos

- Aprender a ler dados de ficheiros.
- Compreender o modelo unificado do Java para I/O.
- Aprender a serializar objectos.
- Compreender a base de acesso a recursos em rede, web, wireless, etc.

Streams

- stream: uma abstracção para uma “origem” ou “destino” de dados
- Os bytes “fluem” “de” (input) “para” (output) streams
- Podem representar múltiplas fontes de dados:
 - Ficheiros em disco
 - Outros computadores em rede
 - Páginas web
 - Dispositivos de entrada (teclado, rato, etc.)



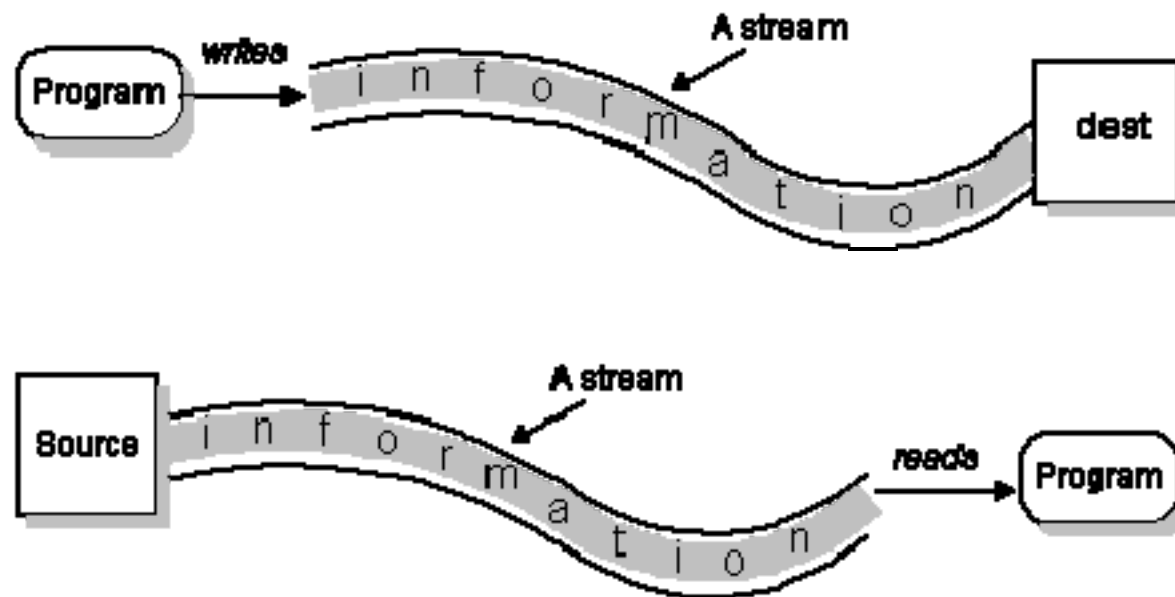
Hierarquia de Streams

- `java.io.InputStream`

- `AudioInputStream`
- `FileInputStream`
- `ObjectInputStream`

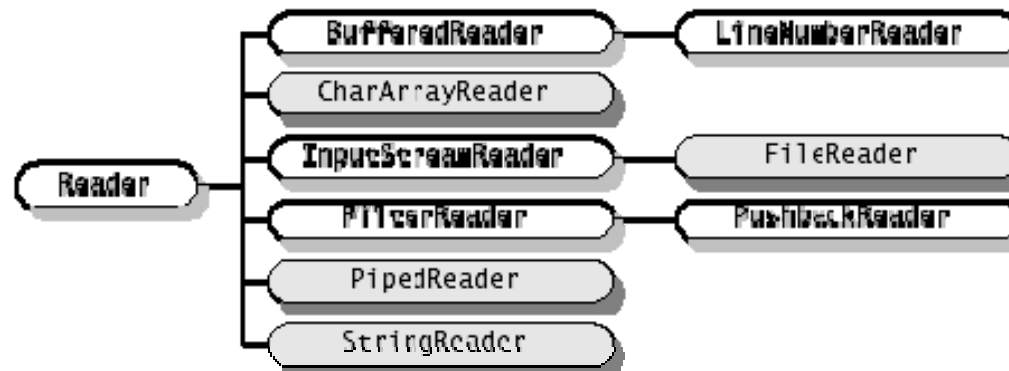
- `java.io.OutputStream`

- `ByteArrayOutputStream`
- `FileOutputStream`
- `ObjectOutputStream`



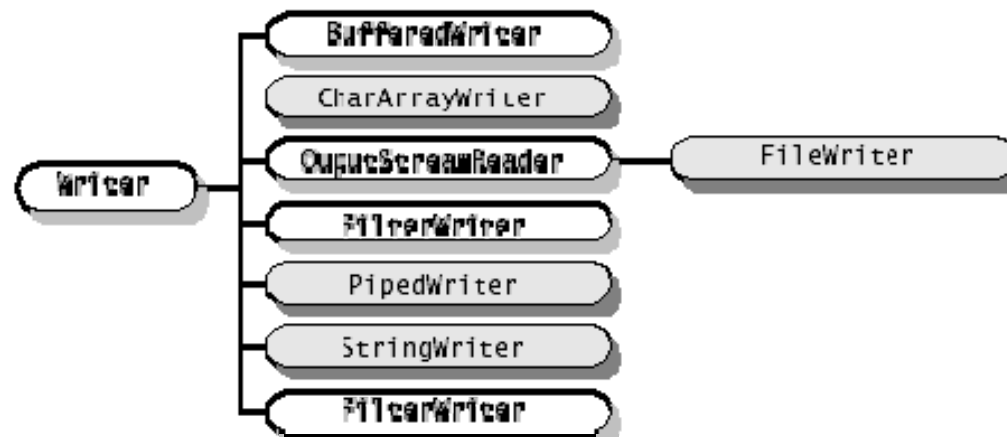
Input Streams

- Métodos comuns a todas as “input streams”:
 - `int read()` throws `IOException`
lê um byte (caracter) de dados
 - `void reset()` throws `IOException`
inicia a stream para que os seus bytes possam ser lidos novamente
 - `void close()` throws `IOException`
notifica a stream de que ela vai deixar de ser usada



Output Streams

- Métodos comuns a todas as “input streams”:
 - `void write(int n) throws IOException`
escreve um byte (character) de dados
 - `void flush() throws IOException`
escreve os bytes que estavam à espera para ser escritos
 - `void close() throws IOException`
notifica a stream de que de que ela vai deixar de ser usada



Exemplo (MAU, aborrecido...)

```
try {
    InputStream in = new FileInputStream("file.txt");
    char oneLetter = (char)in.read();

    String str = "a long string";
    OutputStream out = new FileOutputStream("file.txt");

    // write each letter of string to file
    for (int ii = 0; ii < str.length(); ii++)
        out.write(str.charAt(ii));

    out.close();
} catch (IOException ioe) {
    ...
}
```

Filtered Streams

- Uma stream que obtém os seus dados de outra stream.
- Pode-se criar cadeias de streams para combinar as suas capacidades/características.
- A stream exterior pode adicionar funcionalidade à stream interior, ou melhorar a sua funcionalidade
- Exemplo do “Decorator pattern”

```
InputStream in = new FileInputStream("file.txt");  
DataInputStream dis = new DataInputStream(in);  
double number = dis.readDouble();
```

Readers e Writers

- Classes utilitárias para ajudar a utilizar streams
- Colmatam falhas de métodos em streams e tornam-nas mais robustas.
- Outro exemplo do “Decorator pattern”.
- Os “readers” são mais comuns do que os “writers”.

Alguns “Readers” de interesse

- `java.io.Reader`
 - `public int read()` throws `IOException`
 - `public boolean ready()`
- `java.io.InputStreamReader`
 - `public InputStreamReader(InputStream in)`
- `java.io.FileReader`
 - `public FileReader(String fileName)`
- `java.io.BufferedReader`
 - `public BufferedReader(Reader r)`
 - `public String readLine()` throws `IOException`



Mais funcionalidade

```
InputStream in = new FileInputStream("hardcode.txt");  
InputStreamReader isr = new InputStreamReader(in);  
BufferedReader br = new BufferedReader(isr);  
String wholeLine = br.readLine();
```

```
// or, shorter syntax for reading files...  
BufferedReader br2 = new BufferedReader(  
    new FileReader("hardcode.txt"));  
String anotherLine = br2.readLine();
```

- Mais uma vez, o “Decorator pattern”.

java.nio: a nova API para I/O

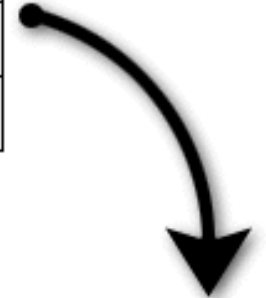
- Novas funcionalidades e performance melhorada em termos de gestão de buffering, dados remotos e I/O de ficheiros, suporte para character-sets, filtragem com expressões regulares.
- A nova API “java.nio” adiciona funcionalidade à API “java.io”.
- As APIs NIO incluem o seguinte:
 - Buffers para dados de tipos primitivos
 - Codificadores e decodificadores dependentes do “character-set”
 - Pattern-matching através de expressões regulares tipo Perl.
 - Channels, uma nova abstracção de I/O
 - Uma interface para ficheiro que suporta “locks” e mapeamento de memória.
 - Uma facilidade de “non-blocking I/O” para escrita de servidores escaláveis.
- As novas APIs são sofisticadas, úteis, mas para situações complexas.

Serialização

- Ler e escrever objectos e o seu estado exacto usando streams.
- Permitem aos próprios objectos se escreverem em ficheiros, através da rede, web, etc.
- Evita a conversão do estado do objecto para um formato textual arbitrário.

Object with data

a: 100	myarray: (0): 24232 (1): 9823.23 (2): 12.782
pi: 3.141592	
msg: Hello, World!	



Serialized data

```
<<100><3.141592><Hello, World!><<24232><9823.23><12.782>>>
```

Classes usadas para serialização

- `java.io.ObjectInputStream`
 - `public ObjectInputStream(InputStream in)`
 - `public Object readObject() throws ...`

- `java.io.ObjectOutputStream`
 - `public ObjectOutputStream(OutputStream out)`
 - `public void writeObject(Object o)
throws IOException`

Exemplo de serialização: escrita

```
try {  
    OutputStream os = new FileOutputStream("file.txt");  
    ObjectOutputStream oos = new ObjectOutputStream(os);  
    oos.writeObject(someList);  
    oos.flush();  
    os.close();  
} catch (IOException ioe) { ... }
```

Exemplo de serialização: leitura

```
try {  
    InputStream is = new FileInputStream("file.txt");  
    ObjectInputStream ois = new ObjectInputStream(is);  
    ArrayList someList = (ArrayList)ois.readObject();  
    is.close();  
} catch (Exception e) { ... }
```

Tornar uma classe “Serializable”

- Basta implementar a interface `java.io.Serializable` para que uma classe seja compatível com streams I/O de objectos.

```
public class BankAccount implements Serializable
{
    ...
}
```

- Garantir que todas as variáveis de instância da classe são igualmente serializáveis ou temporárias (“transient”).

Possíveis problemas com a serialização

- Grafos de objectos
 - Quando um objecto tem variáveis de instância que são referências para outros objectos, esses objectos também deve ser guardados (“object graph”).
- Variáveis temporárias
 - As variáveis de instância que não precisam de ser guardadas (ou não podem ser serializadas) devem ser declaradas com o modificador “transient”

```
private transient TextReader in;
```
- Uma instância do “Memento pattern”.

Serialização para XML

- `java.beans.XMLDecoder` e `java.beans.XMLEncoder`
 - A serialização para XML pode ser facilmente obtida através da API para XML Object Serializers.

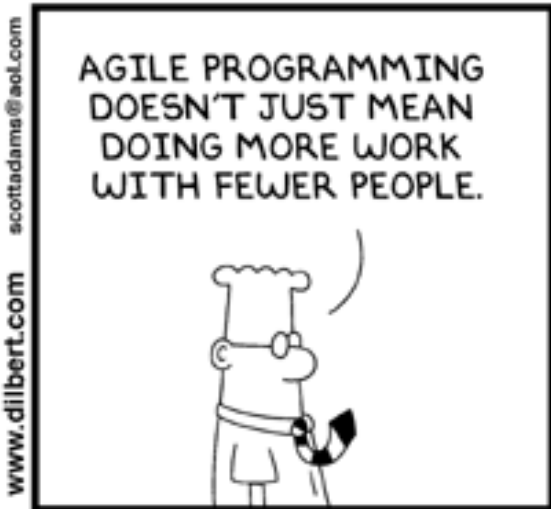
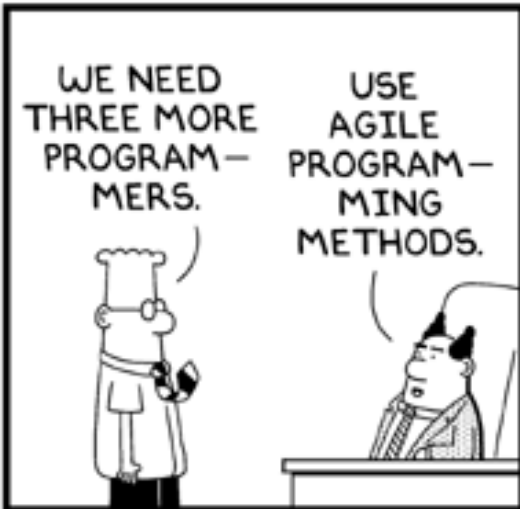
```
/** Save the data to disk. */
public void write(Object theGraph) throws IOException {
    XMLEncoder os = new XMLEncoder(new BufferedOutputStream(
        new FileOutputStream(FILENAME)));
    os.writeObject(theGraph);
    os.close();
}

/** Dump the data */
public void dump() throws IOException {
    XMLDecoder is = new XMLDecoder(new BufferedInputStream(
        new FileInputStream(FILENAME)));
    System.out.println(is.readObject());
    is.close();
}
```

Referência

- The Java Tutorial: I/O.
<http://java.sun.com/docs/books/tutorial/essential/io/index.html>

Exercícios...



© Scott Adams, Inc./Dist. by UFS, Inc.



Parte 3



Universidade do Porto
Faculdade de Engenharia
FEUP

Multithreading

Objetivos

- Explicar os conceitos básicos de ‘multithreading’
- Criar threads múltiplos
- Aplicar a palavra reservada ‘synchronized’
- Descrever o ciclo de vida de um thread
- Usar wait() e notifyAll()



Threads múltiplos



Trabalhador



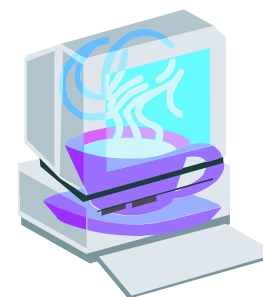
Prioridade de espera e de regresso



Trabalhador



Afectação



Escalonamento pela JVM

O que é um Thread?

- Um *thread* é uma execução sequencial de um programa.
- Cada programa tem pelo menos um *thread*.
- Cada *thread* tem a sua própria pilha, prioridade e conjunto de registos virtuais.
- Os *threads* subdividem o comportamento de um programa em subtarefas independentes.

Onde é que se usam Threads?

- São usados virtualmente em todos os computadores:
 - Em muitas aplicações (imprimir)
 - Em programas como browsers Internet
 - Em bases de dados
 - No sistema operativo
- Os *Threads* são normalmente usados sem serem percebidos pelo utilizador.

Porque se devem usar Threads?

- Para melhor aproveitar as capacidades do computador (utilizar o CPU enquanto se faz entrada/saída de dados)
- Maior produtividade para o utilizador final (uma interface mais interactiva)
- Vantagens para o programador (simplificar a lógica aplicacional)

Os Threads são algo de novo?

- Não!
- Evolução:
 - Utilizador único e sistemas em batch
 - Sistemas multi-processo
 - Sistemas multi-tarefa
 - Sistemas multi-thread
 - Sistemas multi-processador

A Classe Thread

- Mantém o estado de um thread
- Fornece diversos construtores
- Fornece diversos métodos
 - `Thread.currentThread()`
 - `Thread.sleep()`
 - `Thread.setName()`
 - `Thread.isAlive()`
- Escalonados pela JVM
- Utiliza o sistema operativo ou um package de threads

Exemplo de utilização de Thread

- Cada programa corre num thread.
- Adormecer um thread é uma técnica que permite que outros threads executem.

```
public static void main (args[] s) {  
    System.out.println("I am thread " +  
        Thread.currentThread().getName());  
    try {Thread.sleep(5000)}  
    catch (InterruptedException e){}  
    ...  
}
```

Criação de um novo Thread

- 1. Criar a nova classe.
 - a. Definir uma subclasse de Thread.
 - b. Redefinir o seu método run().
- 2. Instanciar e executar o thread.
 - a. Criar uma instância da classe.
 - b. Invocar o método start().
- 3. O escalonador invoca o método run().

Criar a Classe

```
public class SleepingThread extends Thread {
    public void run () {
        Date startTime = new Date();
        try {Thread.currentThread().sleep
            ((int) (1000 * Math.random()));}
        catch (Exception es) {}
        long elapsedTime =
            new Date().getTime() - startTime.getTime();
        System.out.println(
            Thread.currentThread().getName() +
            ": I slept for " + elapsedTime +
            "milliseconds"); }}
```

Instanciar e Executar

```
public static void main(String[] args) {  
    new SleepingThread().start();  
    new SleepingThread().start();  
    System.out.println("Started two threads...");  
}
```

```
Started two threads..  
Thread-1: I slept for 78 milliseconds  
Thread-2: I slept for 428 milliseconds
```

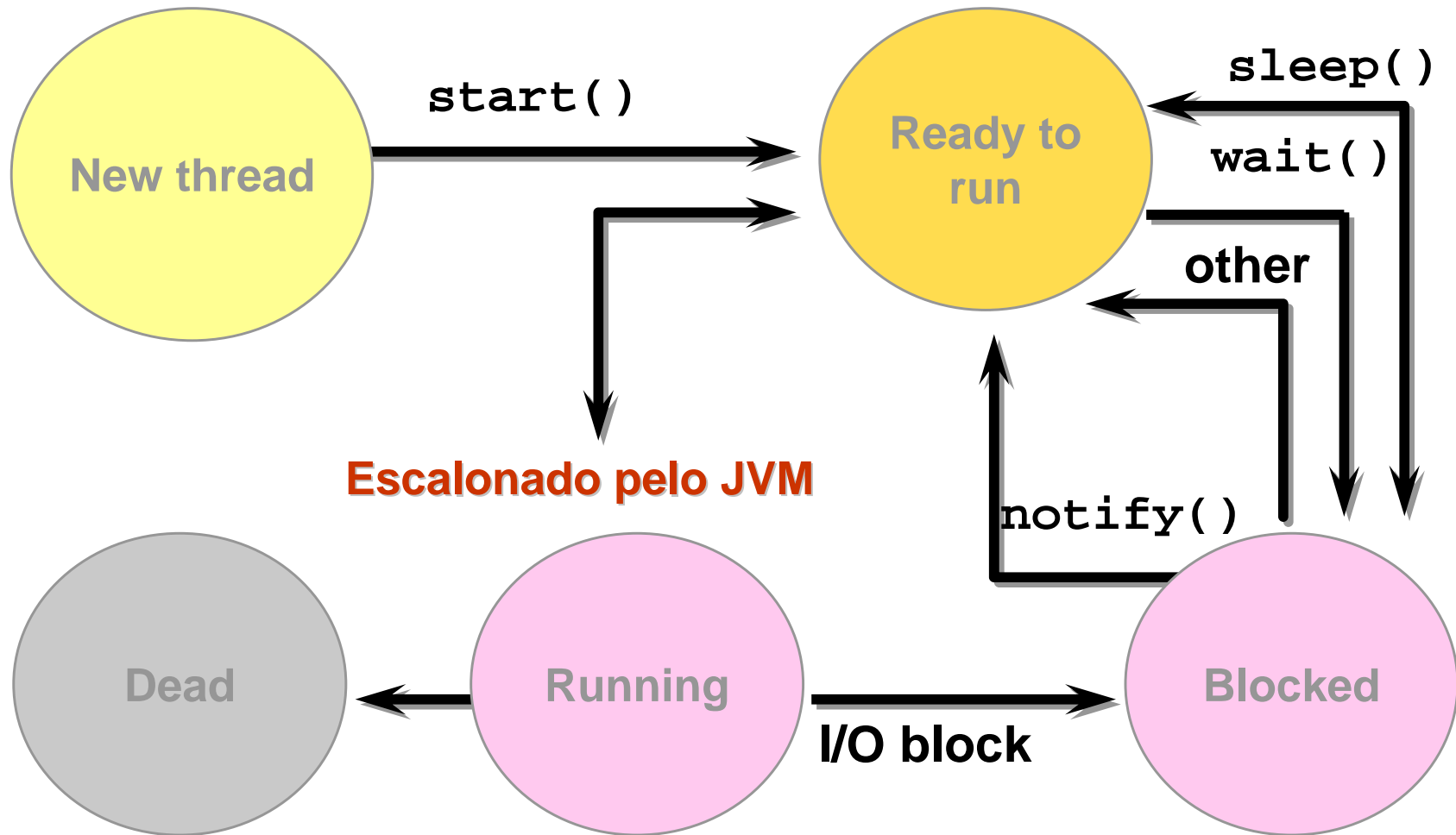
Acesso a Recursos partilhados

- Os dados podem ficar corrompidos se acedidos por vários threads:

```
public class BankAccount {  
    private double balance;  
    public void withdraw(double amt) {  
        balance -= amt;  
    }  
}
```

- Utilizar a palavra *synchronized* para evitar conflitos de recursos partilhados.

Ciclo de Vida de um Thread



Bloquear um Thread

- Utilizar `wait()` para bloquear o thread actual.
- O thread deve conter o monitor do objecto, ie, ser sincronizado (`synchronized`).
- O monitor será desbloqueado quando o `wait()` for invocado.
- O thread pode esperar indefinidamente ou por um período fixo de tempo.
- `notifyAll()` acorda todos os threads bloqueados.

Métodos `synchronized`

- Se um thread invoca um método `synchronized`, nenhum outro thread pode executar um método sincronizado no mesmo objecto até o primeiro thread completar a sua tarefa.

```
public class CheckOutFrame extends JFrame {  
    public synchronized void updateElapsedTime() {  
        ...  
    }  
    public synchronized void  
        computeAverageTime(Date old) {  
        ...  
    }  
}
```

Métodos synchronized

```
public class BankAccount {
    private double balance;
    public synchronized void withdraw(double amt) {
        balance -= amt;
    }
    public synchronized void deposit(double amt) {
        balance += amt;
    } ...
}
```

Cuidado com synchronized!

- Cuidado para evitar deadlocks, evitando que todos os métodos sejam synchronized.

```
void FinishButton(ActionEvent e) {  
    ...  
    finished = true;  
    while(elapsedTime == 0) {}  
    jText.setText("...");  
}
```

Outra forma de criar Threads

- Implementar Runnable.
- Implementar o método run().
- Criar uma instância da classe (objecto alvo).
- Criar uma instância do Thread, passando o objecto alvo como um parâmetro.
- Invocar start() no objecto Thread.
- O escalonador invoca run() sobre o objecto alvo.

Exemplo com Runnable

```
public class MyApplet extends Applet
    implements Runnable{
    private Thread t;
    public void startApplet(){        // called by
browser
        t = new Thread(this); // creates a new
                                // runnable Thread
        t.start();              // starts the new Thread
    }
    public void run(){           // The new runnable Thread
    ...                          // calls run() and the
                                // method runs in a
    } ...                        // separate thread
```

Escalonamento e Prioridades

- Cada thread tem uma prioridade (1 to 10).
- O escalonamento é dependente do sistema operativo.
 - Um thread de alta prioridade pode interromper um de baixa-prioridade.
 - Threads de alta prioridade podem dominar o processador.
 - Threads de prioridade idêntica podem ser escalonados de forma circular.

Exercícios...



Universidade do Porto
Faculdade de Engenharia
FEUP

Introdução ao Swing



Universidade do Porto
Faculdade de Engenharia
FEUP

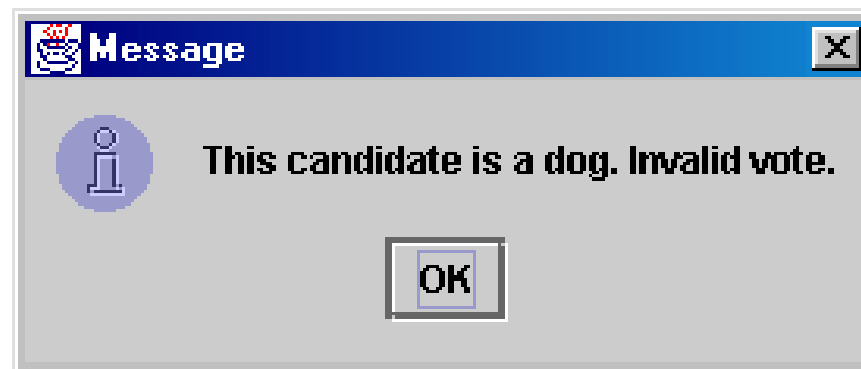
Componentes Principais e Layout

Objetivos

- Aprender a criar interfaces gráficas com o utilizador usando Java e Swing
- Conhecer a hierarquia de componentes gráficos do Java
- Aprender o modelo de eventos do Java

A GUI mais simples!

- `javax.swing.JOptionPane`
 - Não muito boa
 - Não muito poderosa (apenas simples caixas de diálogo)

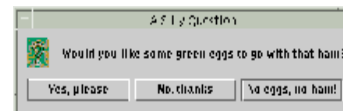


JOptionPane

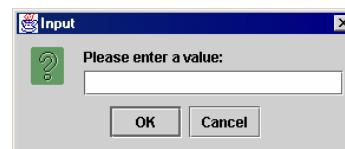
- Um conjunto de caixas de diálogo para simples entrada/saída de dados
 - `public static void showMessageDialog(Component parent, Object message)`
 - Mostra uma mensagem numa caixa de diálogo com um botão “OK”.



- `public static void showConfirmDialog(Component parent, Object message)`
- Mostra uma mensagem e uma lista de opções “Yes” “No” “Cancel”

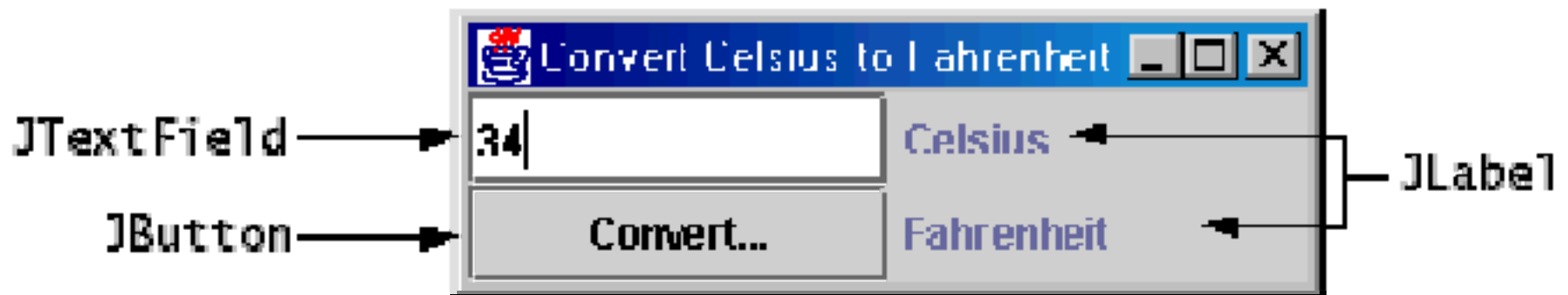


- `public static String showInputDialog(Component parent, Object message)`
- Mostra uma mensagem e um campo de texto, e retorna o valor introduzido como uma String.



Elementos Principais

- Components: coisas desenhadas no écran.
 - Exemplos: botão, caixa de texto, etiqueta.
- Containers: grupos lógicos de components.
 - Exemplo: painel.
- Top-level containers: elementos principais do desktop.
 - Exemplos: frame, caixas de diálogo.



Java GUI: AWT e Swing

- Ideia inicial da Sun (JDK 1.0, 1.1)
 - Criar um conjunto de classes/métodos para desenvolver GUI's multi-plataforma (Abstract Windowing Toolkit, or AWT).
 - Problema AWT: não suficientemente poderosa; limitada.
- Segunda edição (JDK v1.2): Swing
 - Uma nova biblioteca escrita bottom-up que permita o desenvolvimento de gráficos e GUI's mais poderosas.
- Swing e AWT
 - Ambas existem no Java actual
 - São fáceis de misturar; por vezes ainda se têm que usar ambas.

Swing: hierarquia de componentes

java.lang.Object

+--java.awt.Component

+--java.awt.Container

+--javax.swing.JComponent

| +--javax.swing.JButton

| +--javax.swing.JLabel

| +--javax.swing.JMenuBar

| +--javax.swing.JOptionPane

| +--javax.swing.JPanel

| +--javax.swing.JTextField

|

+--java.awt.Window

+--java.awt.Frame

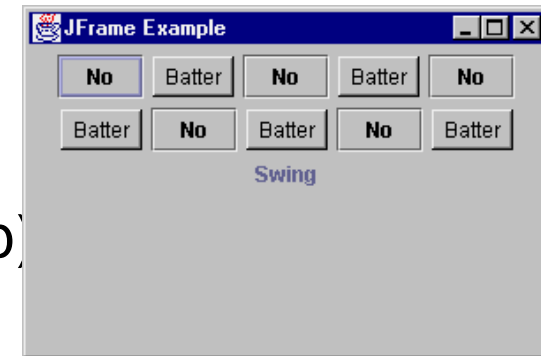
+--javax.swing.JFrame

Componentes Swing: métodos comuns

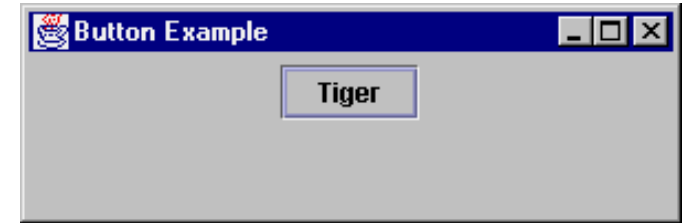
- `get/setPreferredSize`
- `get/setSize`
- `get/setLocation`
- `getLength/Width`
- `repaint`
- `setBackground(Color)`
- `setFont(Font)`
- `isEnabled / setEnabled(boolean)`
- `isVisible / setVisible(boolean)`

JFrame

- Uma frame é uma janela gráfica que pode ser usada para conter outros componentes
- `public void setTitle(String title)`
 - define o título da barra da janela.
- `public void setDefaultCloseOperation(int op)`
 - Define a acção a executar quando fechar.
 - Valor comum: `JFrame.EXIT_ON_CLOSE`
- `public Container getContentPane()`
 - Retorna a área central da janela onde os componentes podem ser adicionados.
- `public void pack()`
 - Redimensiona a frame para que os componentes caibam.
- `public void show()`
 - Mostra a frame no écran.



JButton



- O componente mais comum
 - Uma região de écran que o utilizador pressiona para invocar a execução de um comando.
- `public JButton(String text)`
 - Cria um novo botão com o texto dado.
- `public String getText()`
 - Retorna o texto mostrado no botão.
- `public void setText(String text)`
 - Define o texto do botão.

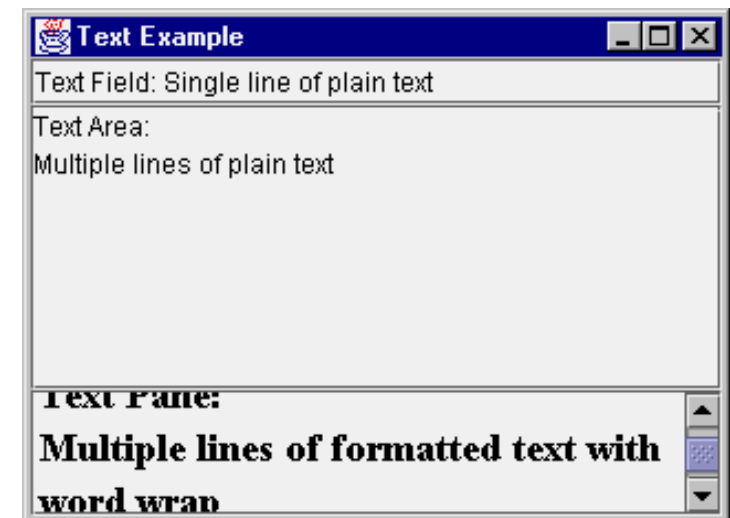
JLabel



- Uma etiqueta de texto é uma simples string de texto visualizada no écran de forma gráfica para fornecer informação sobre outros componentes.
- `public JLabel(String text)`
 - Cria uma nova etiqueta com o texto dado.
- `public String getText()`
 - Retorna o texto actual da etiqueta.
- `public void setText(String text)`
 - Define o texto da etiqueta.

JTextField

- Um campo de texto é como uma etiqueta, mas o texto pode ser editado pelo utilizador.
- `public JTextField(int columns)`
 - Cria um novo campo de texto com um dado número de colunas.
- `public String getText()`
 - Retorna o texto actualmente armazenado no campo.
- `public void setText(String text)`
 - Define o texto do campo.



Como posicionar e dimensionar?

- Como é que o programador pode especificar onde cada componente deve ser posicionado, e qual o tamanho que deve assumir quando a janela é movida e o seu tamanho é alterado (minimizar, maximizar, etc)?
- Posicionamento absoluto (C++, C#, etc) especificam as coordenadas exactas para cada componente.
- Layout managers (Java) são objectos especiais que decidem onde posicionar os componentes com base em critérios bem definidos.

Layout Management!

The image displays five Java Swing windows, each illustrating a different layout manager. Each window has a blue title bar with the Java logo and the name of the layout manager, and standard window control buttons (minimize, maximize, close).

- BorderLayout:** A window with five buttons. 'Button 1' is at the top center. Below it, 'Button 3', '2', and 'Button 5' are arranged horizontally. At the bottom, 'Long-Named Button 4' spans the width.
- GridLayout:** A window with five buttons in a grid. 'Button 1' and '2' are in the top row. 'Button 3' and 'Long-Named Button 4' are in the middle row. 'Button 5' is in the bottom row.
- FlowLayout:** A window with five buttons arranged horizontally from left to right: 'Button 1', '2', 'Button 3', 'Long-Named Button 4', and 'Button 5'.
- BoxLayout:** A window with five buttons arranged vertically from top to bottom: 'Button 1', '2', 'Button 3', 'Long-Named Button 4', and 'Button 5'.
- GridBagLayout:** A window with five buttons. 'Button 1', '2', and 'Button 3' are in a top row. 'Long-Named Button 4' is a large button in the middle. 'Button 5' is a smaller button at the bottom right.

Container

- Um contentor é um objecto que agrega outros componentes; faz a gestão do seu posicionamento, tamanhos e critérios de redimensionamento.
- `public void add(Component comp)`
- `public void add(Component comp, Object info)`
 - Adiciona um componente ao contentor, eventualmente dando informação sobre onde o posicionar.
- `public void remove(Component comp)`
- `public void setLayout(LayoutManager mgr)`
 - Define o layout manager a utilizar para posicionar os componentes no contentor.
- `public void validate()`
 - Informa o layout manager que deve re-posicionar os objectos no contentor.

JPanel

- Um painel é o contentor mais usado.
- `public JPanel()`
 - Constrói um novo JPanel com um flow layout manager por omissão.
- `public JPanel(LayoutManager mgr)`
 - Constrói um novo JPanel com um dado layout manager.
- `public void paintComponent(Graphics g)`
 - Desenha o painel no écran.
- `public void revalidate()`
 - Permite invocar o reposicionamento dos componentes no painel.

BorderLayout

- Divide o contentor em cinco regiões: NORTH, SOUTH, WEST, EAST, CENTER.
- NORTH e SOUTH expandem para preencher a região na horizontal e utilizam a dimensão vertical preferida.
- WEST e EAST expandem para preencher a região na vertical e utilizam a dimensão horizontal preferida.
- CENTER utiliza todo o espaço não ocupado pelos restantes componentes.



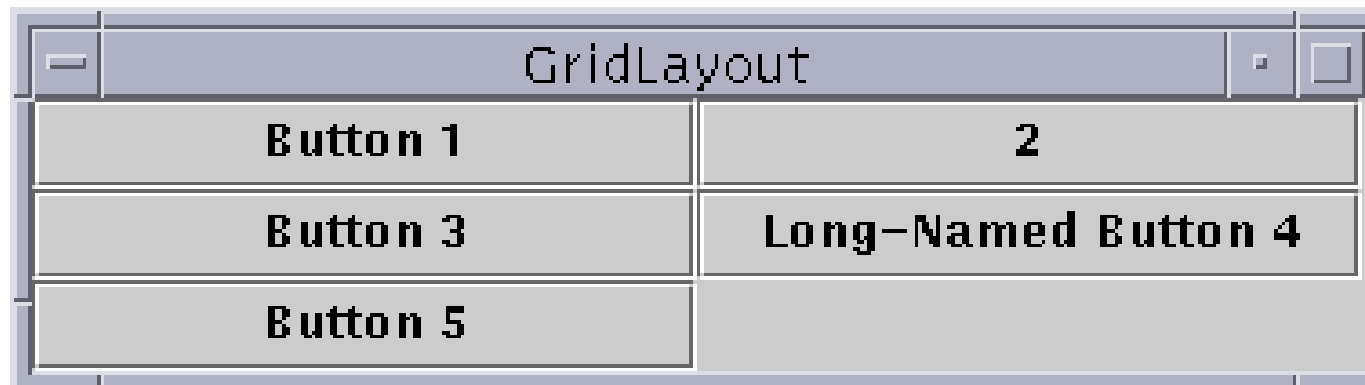
FlowLayout

- Trata o contentor como uma página ou parágrafo preenchido da esquerda para a direita ou de cima para baixo.
- Os componentes assumem as dimensões verticais e horizontais preferidas.
- Os componentes são posicionados pela ordem que são adicionados.
- Quando necessário, os componentes passam para a linha seguinte.



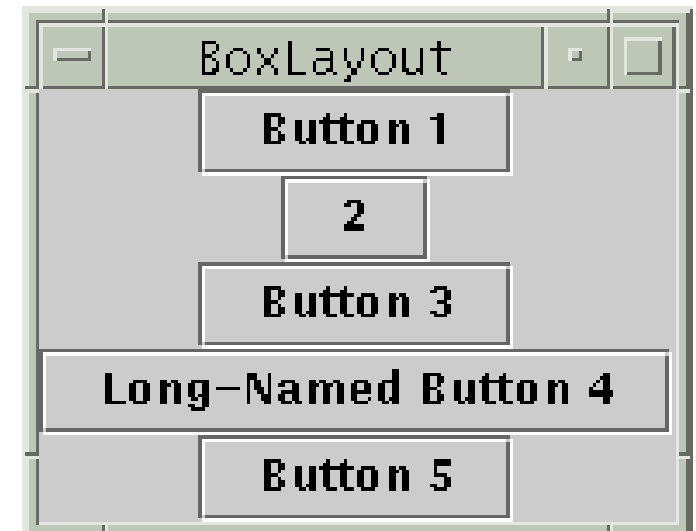
GridLayout

- Trata o contentor como uma grelha de linhas e colunas de tamanhos iguais.
- Os componentes são dimensionados igualmente (horizontal/verticalmente), independentemente das suas dimensões preferidas.
- Pode-se especificar 0 linhas/colunas para indicar a expansão numa direcção desejada.



BoxLayout

- Alinha os componentes no contentor numa única linha ou coluna.
- Os componentes usam as suas dimensões preferidas e são alinhados de acordo com o seu alinhamento preferido.
- A forma preferida de construir um contentor com um box layout é:
 - `Box.createHorizontalBox()`
 - ou `Box.createVerticalBox()`.



Outros Layouts

- CardLayout: camadas de “cartas” empilhadas; apenas uma é visível em cada instante.



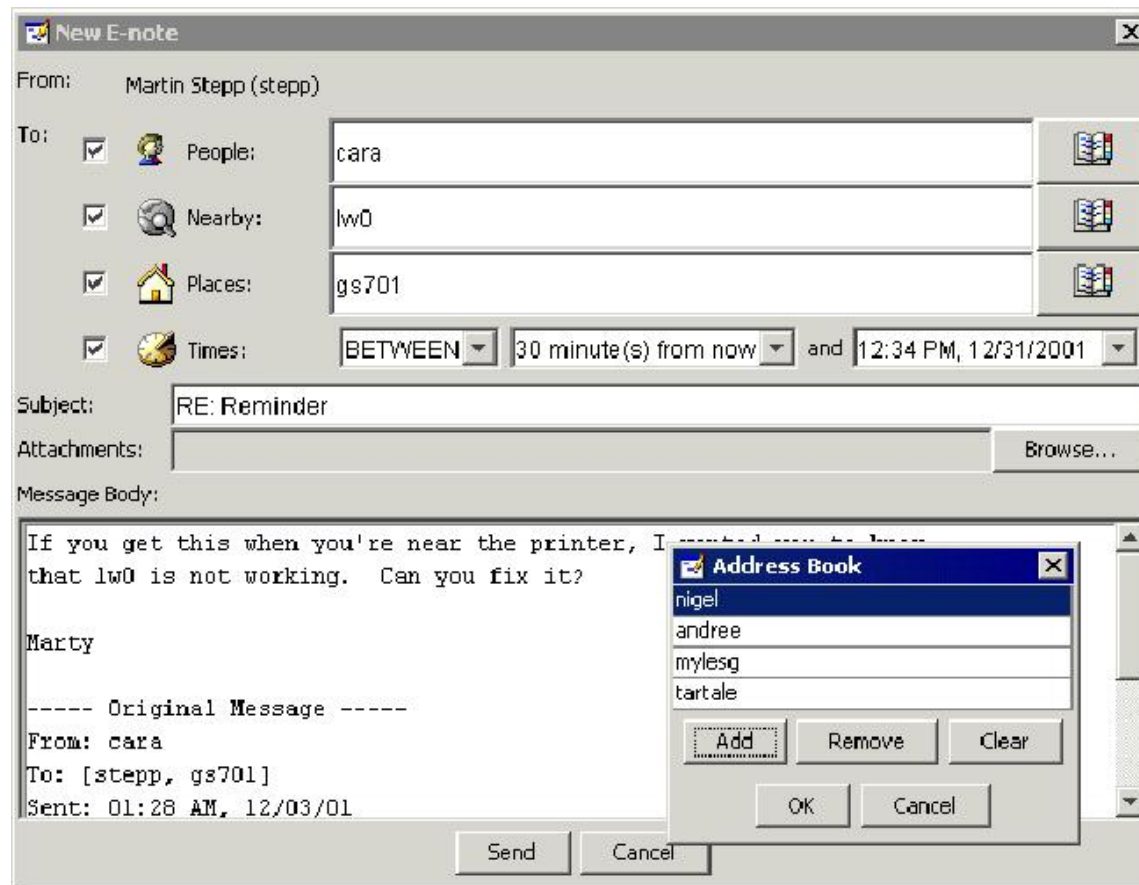
- GridBagLayout: algo complicado, mas bastante poderoso se usado convenientemente.



- Custom / null layout: permite definir p
utilizando setSize e setLocation.

Problemas com Layout Managers

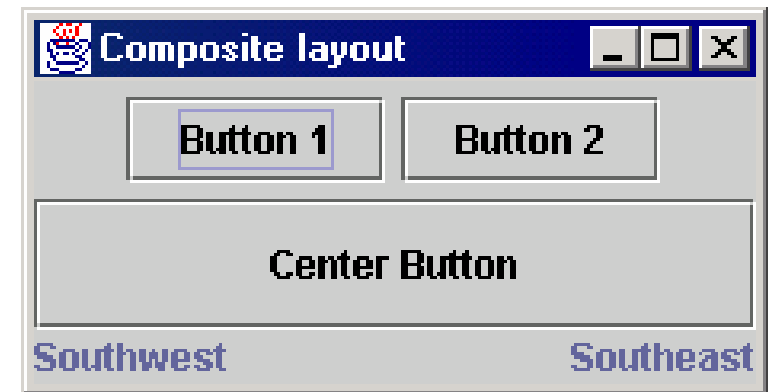
- Como criar uma janela complexa como esta utilizando os layout managers apresentados?



Solução: Composite Layout

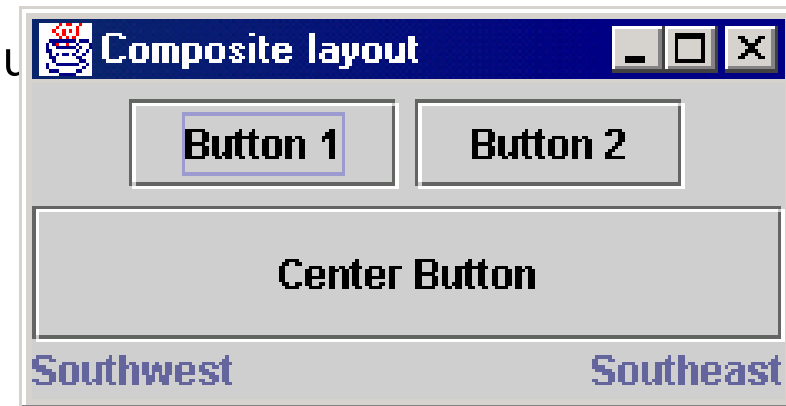
- Criar painéis dentro de painéis.
- Cada painel pode usar um layout diferente e ao combinar com outros layouts, consegue-se obter layouts mais complexos e poderosos.
- Exemplo
 - Quantos painéis?
 - Qual o layout usado em cada um deles?

- Pattern: “Composite” pattern



Composite Layout Code Example

```
Container north = new JPanel(new FlowLayout());  
north.add(new JButton("Button 1"));  
north.add(new JButton("Button 2"));
```



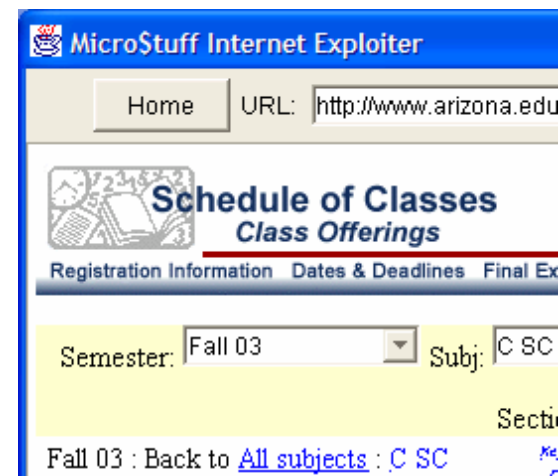
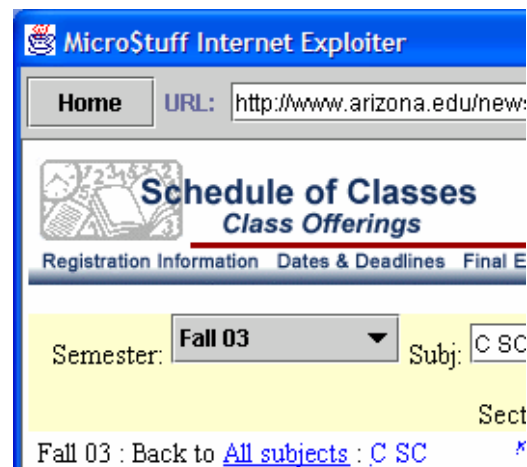
```
Container south = new JPanel(new BorderLayout());  
south.add(new JLabel("Southwest"), BorderLayout.WEST);  
south.add(new JLabel("Southeast"), BorderLayout.EAST);
```

```
Container cp = getContentPane();  
cp.add(north, BorderLayout.NORTH);  
cp.add(new JButton("Center Button"), BorderLayout.CENTER);  
cp.add(south, BorderLayout.SOUTH);
```


Look and Feel

- Sendo o Java Swing uma biblioteca multi-plataforma para GUI's, ele pode assumir diferentes aspectos (look & feel).
- Look & feel por omissão: “Metal”

```
try {  
    UIManager.setLookAndFeel(  
        UIManager.getSystemLookAndFeelClassName());  
} catch (Exception e) { }
```



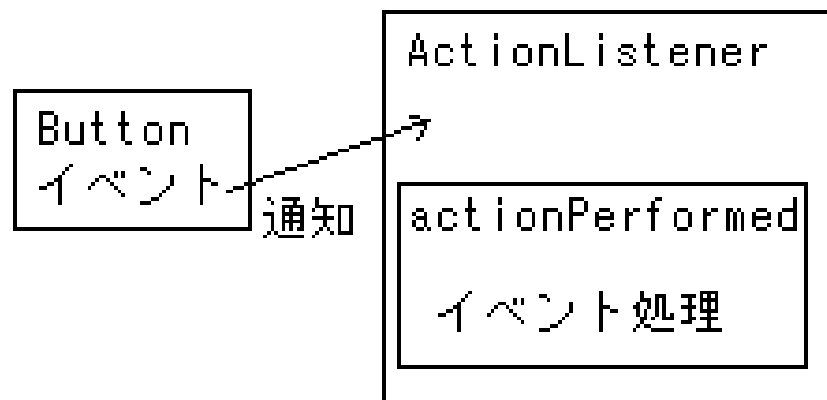
Referências

- The Java Tutorial: Visual Index to the Swing Components.
 - <http://java.sun.com/docs/books/tutorial/uiswing/components/components.html>

- The Java Tutorial: Laying Out Components Within a Container.
 - <http://java.sun.com/docs/books/tutorial/uiswing/layout/index.html>



Mecanismo de eventos



Programação baseada em eventos

- A execução do programa é indeterminada.
- Os componentes gráficos originam eventos através de acções do utilizador sempre que são premidos ou usados.
- Os eventos são recebidos e tratados por programas com base em critérios pré-definidos de encaminhamento dos eventos.
- O programa responde aos eventos (programa “event-driven”).

Java Event Hierarchy

java.lang.Object

+--java.util.EventObject

+--java.awt.AWTEvent

+--java.awt.event.ActionEvent

+--java.awt.event.TextEvent

+--java.awt.event.ComponentEvent

+--java.awt.event.FocusEvent

+--java.awt.event.WindowEvent

+--java.awt.event.InputEvent

+--java.awt.event.KeyEvent

+--java.awt.event.MouseEvent

Acções de eventos (ActionEvent)

- Tipo de eventos mais comum no Swing.
- Representam uma acção que ocorreu num componente GUI.
- São criadas por:
 - Cliques em botões
 - Activar e desactivar Check box's
 - Cliques em menus
 - Ao pressionar [Enter] em campos de texto
 - etc.

“Auscultar Eventos” (event listeners)

- Adicionar um listener aos componentes.
- O método apropriado do listener será invocado quando o evento ocorre (p.e. quando o botão é premido).
- Para eventos de acção, utiliza-se ActionListener's.



Exemplo de um ActionListener

```
// part of Java; you don't write this
public interface ActionListener {
    void actionPerformed(ActionEvent event);
}

// Prints a message when the button is clicked.
public class MyActionListener implements ActionListener
{
    public void actionPerformed(ActionEvent event) {
        System.out.println("Event occurred!");
    }
}
```


Adicionar um ActionListener

```
 JButton button = new JButton("button 1");  
 MyActionListener listener = new MyActionListener();  
 button.addActionListener(listener);  
  
 // now button will print "Event occurred!" when clicked  
 // addActionListener method is in many components
```

Propriedades objectos `ActionEvent`

- `public Object getSource()`
 - Retorna o objecto que originou o evento.
- `public String getActionCommand()`
 - Retorna uma string que representa este evento.
- Questão: onde colocar a classe listener?

ActionListener como “Inner Class”

```
public class Outer {  
    private class Inner implements ActionListener {  
        public void actionPerformed(ActionEvent event) {  
            ...  
        }  
    }  
}  
  
public Outer() {  
    JButton myButton = new JButton();  
    myButton.addActionListener(new Inner());  
}  
}
```

ActionListener como “Anonymous Inner”

```
public class Outer {  
    public Outer() {  
        JButton myButton = new JButton();  
        myButton.addActionListener(  
            new ActionListener() {  
                public void actionPerformed(ActionEvent e) {  
                    ...  
                }  
            }  
        );  
    }  
}
```

// Anonymous inner classes are good for something!

ActionListener numa JFrame

```
public class Outer extends JFrame
    implements ActionListener
{
    public Outer() {
        JButton myButton = new JButton();
        myButton.addActionListener(this);
    }

    public void actionPerformed(ActionEvent event) {
        ...
    }
}
```

Exercício: uma GUI simples

- Problema → Solução
- Uma frame com dois botões
- Cada botão, uma acção → Criar um ActionListener para cada acção.
- Vários ActionListeners → Encapsular as acções em classes (Command pattern) para conseguir usar um ActionListener genérico.
- Como parametrizar o ActionListener genérico? → Criar um ficheiro de configuração com os nomes das acções e dos Command's respectivos que por Reflection permite carregar dinamicamente as classes respectivas.

Referências

- The Java Tutorial: How to Write Action Listeners.
 - <http://java.sun.com/docs/books/tutorial/uiswing/events/actionlistener.html>



Universidade do Porto
Faculdade de Engenharia
FEUP

Applets

O que é uma Applet?

- É um programa Java que pode ser inserido numa página web e é executado ao carregar essa página num browser
 - Permite dar mais vida a páginas web, adicionando conteúdos interactivos, multimédia, jogos, etc.
 - As applets foram a feature do Java responsável pela sua popularidade inicial

- Implementação: um contentor principal, como JFrame

Interação com o Browser

- As applets correm numa Java Virtual Machine dentro do browser.
- Problema: muitos browsers web (MS IE, Netscape 4, etc.) apenas disponibilizam uma JVM v1.1.8
- Solução 1: usar apenas classes/features do Java que já existem desde v1.1.8.
- Solução 2: utilizar o Java Plug-in para dar ao browser a capacidade de utilizar as novas features do Java.

Applet Inheritance Hierarchy

java.lang.Object

java.awt.Component

java.awt.Container

java.awt.Panel

java.awt.Applet

javax.swing.JApplet

javax.swing.JApplet

- Diferenças entre JApplet and JFrame:
 - Não é necessário o método main
 - Não é necessário invocar show() (é automático!)
 - Não é necessário definir a setDefaultCloseOperation(...)
 - Não é necessário setSize(...) / pack(); o tamanho é determinado pela página web
 - Não é necessário setTitle(String); o título é dado pela página web

JApplet: restrições

- Não se consegue abrir ficheiros do disco do utilizador.
- Não se consegue estabelecer ligações em rede com outros computadores para além do servidor da página da applet.
- Não é possível executar programas.
- Não se consegue saber muita informação sobre o sistema cliente.
- Qualquer janela aberta por uma applet terá um aviso no fundo.



JApplet: métodos de controlo

- `public void init()`
 - Invocado pelo browser quando a applet é descarregada a primeira vez.
- `public void start()`
 - Invocado pelo browser sempre que o utilizador visita a página web
- `public void stop()`
 - Invocado pelo browser sempre que o utilizador abandona a página web da applet.
- `public void destroy()`
 - Invocado pelo browser quando este termina.

Página web com uma Applet JDK 1.1

- Exemplo de página web:

```
<HTML>
<HEAD><TITLE>Marty's Applet Page</TITLE></HEAD>

<BODY>
<APPLET code="mypackage/MyApplet.class"
  width=400 height=300> </APPLET>
</BODY>
</HTML>
```

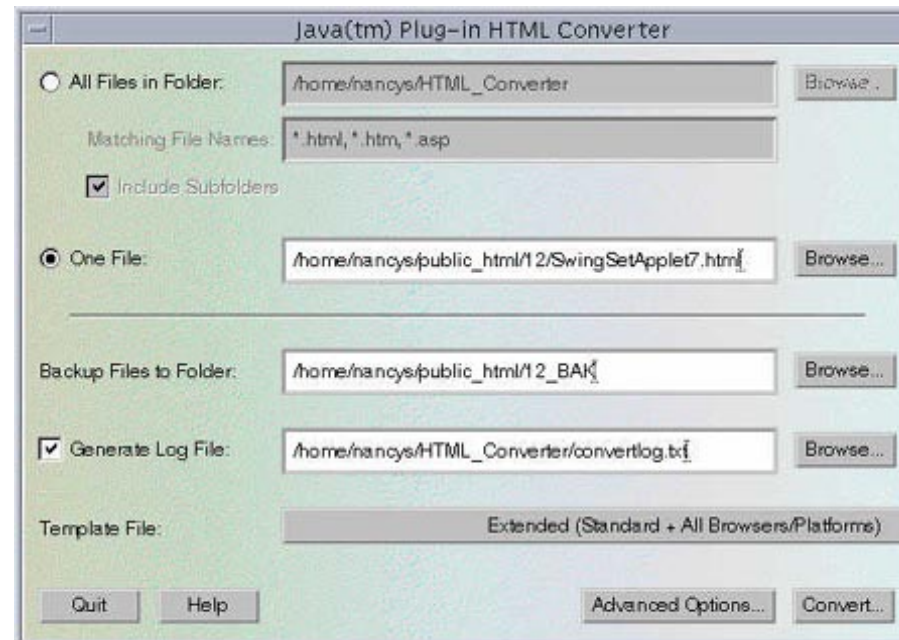
- Problema: não usa Java Plug-in; limitado ao JDK 1.1

Página web com uma JApplet JDK 1.3

```
<HTML>
<HEAD><TITLE>Exemplo</TITLE></HEAD>
<BODY>
<OBJECT classid="clsid:8AD9C840-044E-11D1-B3E9-00805F499D93" width=400 height=300
  codebase="http://java.sun.com/products/plugin/1.3.0_02/jinstall-130_02-
    in32.cab#Version=1,3,0,2">
  <PARAM name="code" value="mypackage/MyApplet.class">
  <PARAM name="type" value="application/x-java-applet;version=1.3">
  <COMMENT>
    <EMBED type="application/x-java-applet;version=1.3" code=" mypackage/MyApplet.class"
      width="400" height="400" scriptable=false pluginspage="http://java.sun.com/products/plugin/
1.3.0_02/plugin-install.html">
    <NOEMBED> </NOEMBED>
  </EMBED>
  </COMMENT>
</OBJECT>
</BODY></HTML>
```


Java HTML Converter

- <http://java.sun.com/products/plugin/1.3/converter.html>
- Converte uma página HTML para usar as novas tags que provocam a invocação Java Plug-in, possibilitando as novas features v1.3+, tais como o Swing.



JApplet Workarounds

- Não se pode usar: Toolkit's `getImage(String)`
- Usar antes: JApplet's `getImage(URL)` ou `getImage(URL, String)` métodos (depois de colocar imagem na web)

- Não se pode usar: `FileInputStream` para ler um ficheiro
- Usar antes: `new URL(file).openStream();` (depois de colocar ficheiro na web)

Referências

- The Java Tutorial: Writing Applets.
 - <http://java.sun.com/docs/books/tutorial/applet/>
- Java HTML Converter download.
 - <http://java.sun.com/products/plugin/1.3/converter.html>

Exercícios...



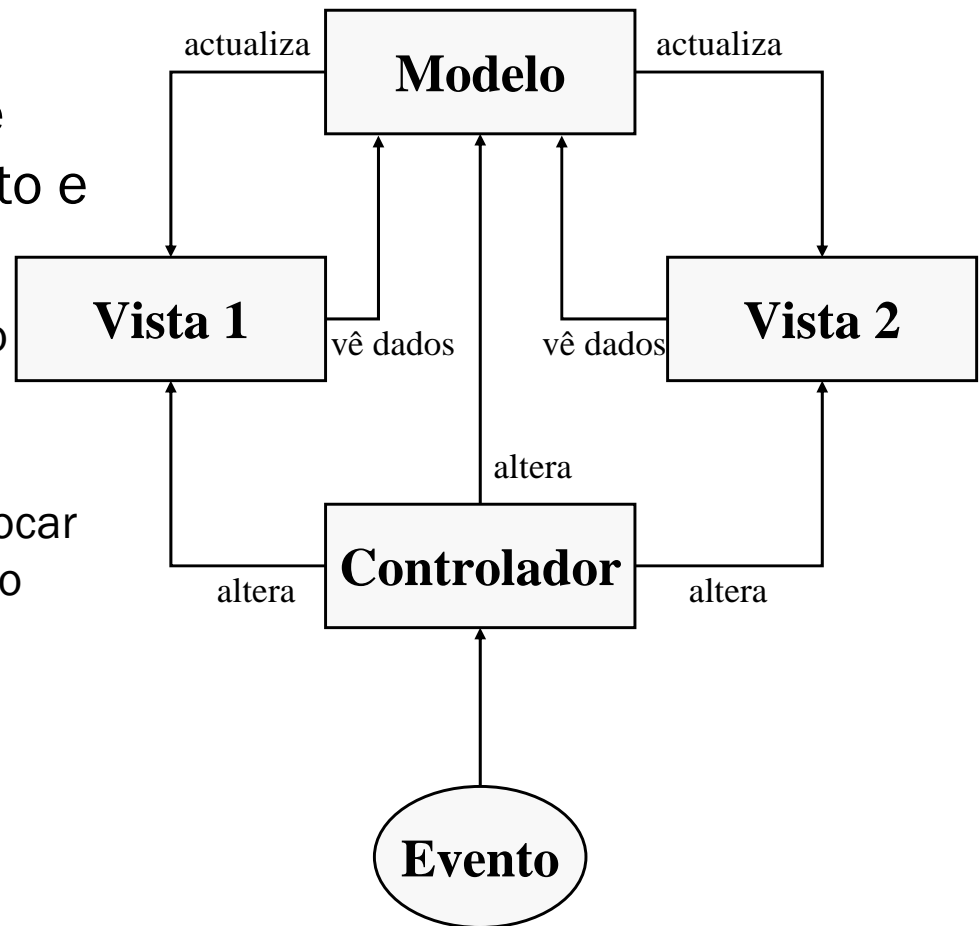
Model-View-Controller

Arquitectura MVC

- Arquitectura para construção de aplicações OO em que se separam três dimensões
 - Modelo: mantém dados usando os algoritmos apropriados e fornece métodos de acesso
 - Vista: constroi uma representação visual de parte ou todos os dados do modelo
 - Controlador: trata os eventos
- Quando o modelo altera os seus dados, gera eventos que fazem com que a vista actualize a sua representação, segundo as ordens do controlador
- Podem existir várias vistas e controladores para o mesmo modelo, o qual pode permancer inalterado quando este evolui.

Comunicação MVC

- Uma alteração no modelo provoca um evento de alteração que é difundido para todos os objectos que estão à escuta desse evento e desencadeia as alterações
 - Facilita manter o sincronismo entre vistas diferentes de um mesmo modelo
 - Actuar numa vista pode provocar alterações no modelo que são reflectidas nas outras vistas

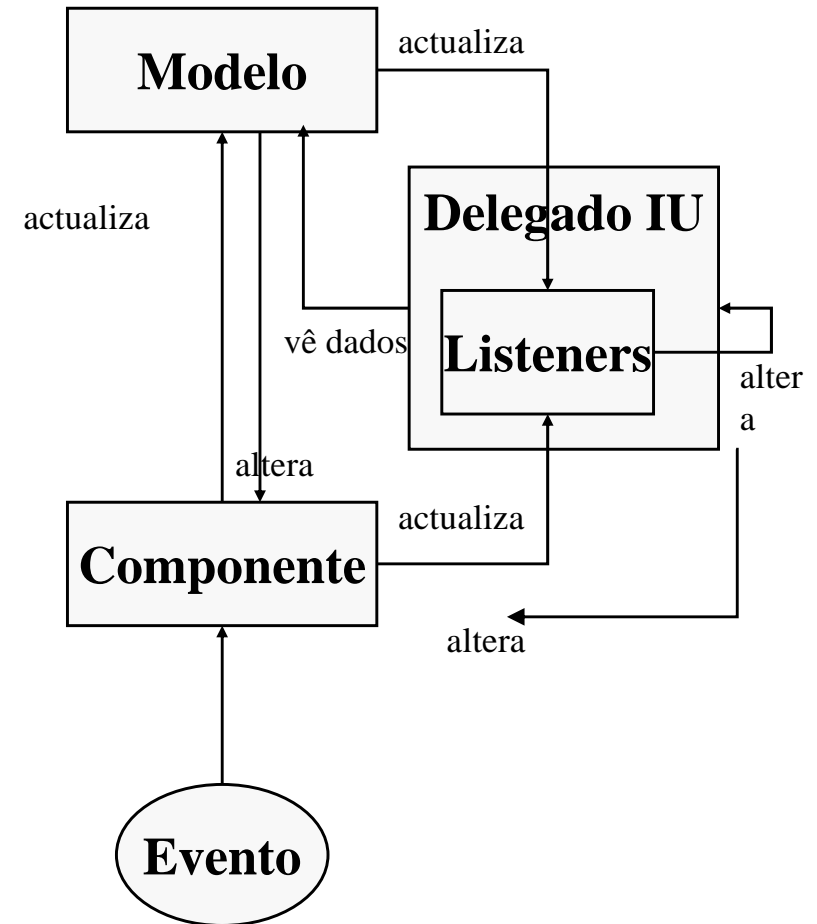


Arquitetura MVC em Swing

- Um componente Swing leve inclui os seguintes objectos:
 - Um modelo que mantém os dados (→ modelo da MVC básica)
 - fornece métodos de acesso
 - notifica os listeners quando é alterado
 - Um delegado da IU que é uma vista (→ vista) com listeners (→ controladores)
 - combina as duas funções colocando os listeners junto dos objectos controlados
 - listeners são habitualmente implementados por classes internas
 - Um componente que estende JComponent
 - um componente fornece uma API para o programador
 - transfere a construção de interfaces para os delegados; passa-lhes os eventos
 - torna o modelo transparente para o programador; atravessado pelos métodos
- Suporta a troca do look & feel: Macintosh, Windows, Motif.

Comunicação MVC em Swing

- **Componente**
 - Faz alterações ao modelo e faz seguir para o modelo alterações que venham da interface
 - Escutam o modelo para passarem os eventos para os seus listeners
- **Listeners do delegado IU**
 - Tanto escutam o modelo como o componente
 - Pedem informação ao modelo
 - Alteram o próprio delegado



Exercícios...

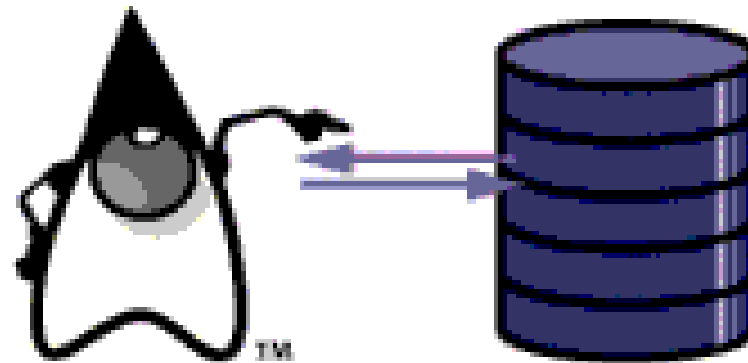


Universidade do Porto
Faculdade de Engenharia
FEUP

Parte 4



Acesso a Bases de Dados por JDBC `java.sql.*`



Tópicos

- Introdução ao JDBC e JDBC Drivers
- Seis Passos para usar JDBC
- Exemplo 1: Definir tabelas por JDBC
- Exemplo 2: Inserir dados por JDBC
- Exemplo 3: Interrogações por JDBC
- Tratamento de Excepções em JDBC

Introdução ao JDBC

- JDBC é uma API simples para acesso a múltiplas bases de dados a partir de aplicações Java.
- Permite uma integração simples entre o mundo das bases de dados e o mundo das aplicações Java.
- A ideia é a de uma API para acesso universal a bases de dados, inspirada no Open Database Connectivity (ODBC) desenvolvido para criar um standard para o acesso a bases de dados em Windows.
- A JDBC API (`java.sql.*` e `javax.sql.*`) pretende ser o mais simples possível e simultaneamente oferecer a máxima flexibilidade aos programadores.

JDBC Drivers

- Para ligar a uma base de dados é necessário primeiro ter um driver JDBC.
- JDBC Driver: conjunto de classes que fazem interface com um motor específico de bases de dados.
- Existem drivers JDBC para a maioria das bases de dados: Oracle, SQL Server, Sybase, and MySQL.

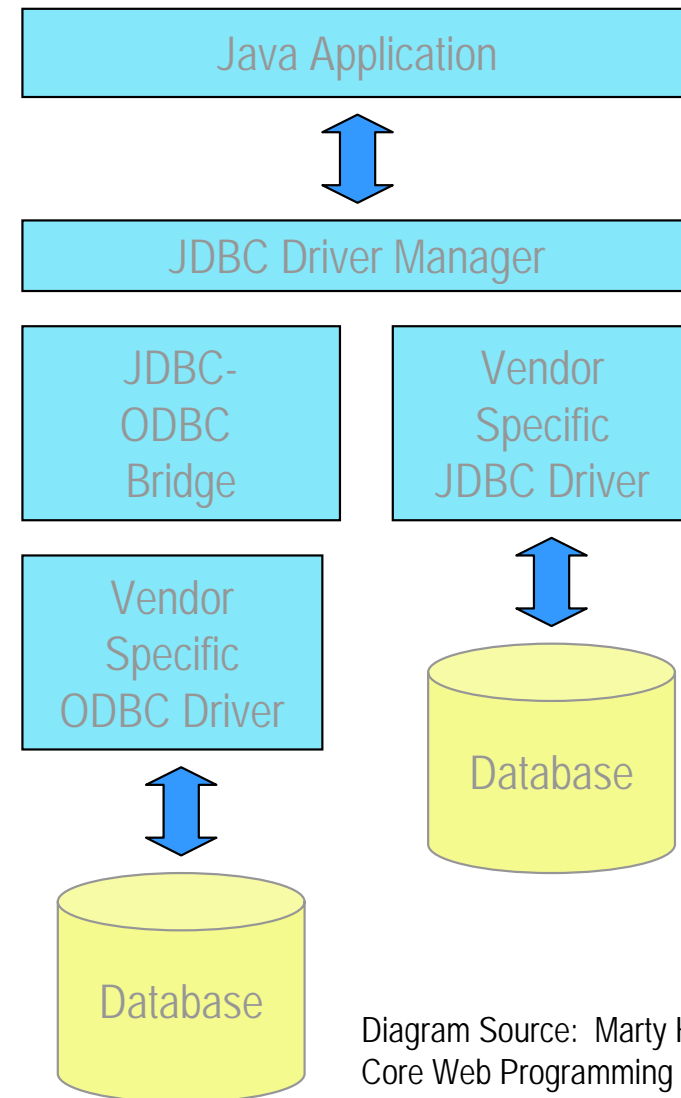


Diagram Source: Marty Hall,
Core Web Programming
(Prentice Hall.)

Instalar o driver MySQL

- Para MySQL, existe por exemplo o MySQL Connector/J, open source.
 - <http://www.mysql.com/downloads/api-jdbc.html>.
- Para usar o driver MySQL Connector/J, é necessário descarregar a distribuição completa.
- Adicionar o respectivo JAR à CLASSPATH:
 - `mysql-connector-java-3.0.11-stable-bin.jar`



Seis Passos para usar JDBC

Seis Passos para Usar JDBC

1. Carregar o driver JDBC
2. Estabelecer a ligação à base de dados (Connection)
3. Criar um objecto Statement
4. Executar uma Query
5. Processar os Resultados
6. Fechar a ligação

1. Carregar o Driver JDBC

- Para usar um driver JDBC, é necessário carregar o driver através do método `Class.forName()` (reflection).

- Em geral, o código é algo como:

```
Class.forName( "jdbc.DriverXYZ" );
```

Em que `jdbc.DriverXYZ` é o driver JDBC que se pretende carregar.

- Se usarmos um driver JDBC-ODBC, o código será algo como:

```
Class.forName( "sun.jdbc.odbc.JdbcOdbcDriver" );
```

Carregar o Driver JDBC...

- Se usarmos o MM MySQL Driver, o código será algo como:

```
try {  
    Class.forName("com.mysql.jdbc.Driver");  
} catch(java.lang.ClassNotFoundException e) {  
    System.err.print("ClassNotFoundException: ");  
    System.err.println(e.getMessage());  
}
```

- `Class.forName()` origina uma `ClassNotFoundException` se a `CLASSPATH` não estiver correctamente definida.
- Deve-se portanto colocar um `try/catch` em volta de `forName()`.

2. Estabelecer a Ligação (Connection)

- Depois de carregado o driver JDBC, pode-se estabelecer a ligação à base de dados.

```
Connection con = DriverManager.getConnection(url,  
"myLogin", "myPassword");
```

- A única dificuldade é especificar o URL correcto.
- O URL tem normalmente o seguinte formato:
jdbc:subprotocol:subname.
 - *jdbc* indica que é uma JDBC Connection
 - O *subprotocol* identifica o driver que se pretende usar.
 - O *subname* identifica o nome da base de dados e sua localização.

URL: Exemplo para ODBC

- As linhas seguintes usam uma “bridge” JDBC-ODBC para ligar à base de dados FEUP local:

```
String url = "jdbc:odbc:FEUP";
```

```
Connection con = DriverManager.getConnection(url,  
    "aaguiar", "password");
```

URL: Exemplo para MySQL

- Para ligar a MySQL:

```
String url = "jdbc:mysql://localhost/feup";  
Connection con =  
    DriverManager.getConnection(url);
```

- Neste caso, está-se a usar um driver MySQL JDBC Driver para ligar à base de dados “feup”, na máquina *localhost*.
- Se este código executar correctamente teremos um objecto `Connection` para comunicar directamente com a base de dados.

3. Criar um objecto Statement

- O objecto JDBC `Statement` envia comandos SQL para a base de dados.
- Os objectos `Statement` são criados a partir de objectos `Connection` activos.
- Por exemplo:

```
Statement stmt = con.createStatement();
```
- Com um objecto `Statement`, pode-se enviar chamadas SQL directamente à base de dados.

4. Executar uma Query

- `executeQuery()`
 - Executa uma query SQL e retorna os dados numa tabela (`ResultSet`)
 - A tabela resultante pode estar vazia mas nunca null.

```
ResultSet results = stmt.executeQuery("SELECT a, b FROM  
table");
```

- `executeUpdate()`
 - Utilizado para executar instruções SQL INSERT, UPDATE, ou DELETE.
 - O resultado é o número de linhas que foram afectadas na base de dados.
 - Suporta instruções Data Definition Language (DDL) tipo:
 - CREATE TABLE
 - DROP TABLE
 - ALTER TABLE

Statement: Métodos Úteis

- `getMaxRows/setMaxRows`
 - Determina o número de linhas que um `ResultSet` pode conter
 - Por omissão, o número de linhas é ilimitado (return 0)
- `getQueryTimeout/setQueryTimeout`
 - Especifica o tempo que um driver deve esperar pela execução de um `STATEMENT` antes de lançar a exceção `SQLException`

5. Processar os Resultados

- A `ResultSet` contém os resultados da query SQL.
- Métodos úteis
 - Todos os métodos podem lançar uma `SQLException`
 - `close`
 - Liberta os recursos alocados pelo JDBC
 - O conjunto de resultados é **automaticamente fechado** sempre que o `Statement` associado **executa uma nova query**.
 - `getMetaDataObject`
 - Retorna um objecto `ResultSetMetaData` que contém informação sobre as colunas do `ResultSet`
 - `next`
 - Tenta mover para próxima linha do `ResultSet`
 - Se bem sucedido é devolvido `true`; senão, `false`
 - A primeira invocação de `next` posiciona o cursor na primeira linha

ResultSet: mais métodos

- Métodos úteis (cont.)
 - findColumn
 - Retorna o valor inteiro correspondente à coluna especificada por nome
 - Os números das colunas nos resultados não mapeiam necessariamente para as mesmas colunas na base de dados.
 - getXxx
 - Retorna o valor da coluna especificada pelo nome ou índice como um tipo Xxx do Java
 - Retorna 0 ou null, se o valor SQL for NULL
 - Tipos usados em getXxx são:

double

byte

int

Date

String

float

short

long

Time

Object

6. Fechar a ligação

- Para fechar a ligação:

```
stmt.close();  
connection.close();
```

- Nota: alguns servidores aplicativos mantêm um conjunto de ligações à base de dados.
 - Isto é muito mais eficiente, uma vez que as aplicações não têm o custo associado a constantemente abrir e fechar ligações com a base de dados.



Exemplo 1: Criar Tabelas por JDBC

As tabelas do exemplo

- Instruções SQL:
CREATE TABLE COFFEES
(COF_NAME VARCHAR(32),
SUP_ID INTEGER,
PRICE FLOAT,
SALES INTEGER,
TOTAL INTEGER);

A tabela Coffee

- Pode-se criar uma tabela directamente na base de dados (MySQL), mas pode-se também criar por JDBC.
- Observações sobre a tabela:
 - A coluna SUP_ID contém um valor inteiro para indicar o ID do fornecedor (Supplier ID).
 - Os fornecedores serão guardados numa tabela separada.
 - SUP_ID é uma *foreign key*.
 - A coluna SALES armazena valores do tipo SQL INTEGER e indica o total em euros de café vendido durante a semana corrente.
 - A coluna TOTAL contém um SQL INTEGER que indica o total em euros de café vendido desde sempre.


```
import java.sql.*;

public class CreateCoffees {
    public static void main(String args[]) {
        String url = "jdbc:mysql://localhost/feup";
        Connection con;
        String createString;
        createString = "create table COFFEES " +
            "(COF_NAME VARCHAR(32), " +
            "SUP_ID INTEGER, " +
            "PRICE FLOAT, " +
            "SALES INTEGER, " +
            "TOTAL INTEGER)";
        Statement stmt;
```

```
try {  
    Class.forName("com.mysql.jdbc.Driver");  
} catch(java.lang.ClassNotFoundException e) {  
    System.err.print("ClassNotFoundException: ");  
    System.err.println(e.getMessage());  
}
```

1

```
try {  
    con = DriverManager.getConnection(url);  
    stmt = con.createStatement();  
    stmt.executeUpdate(createString);  
    stmt.close();  
    con.close();  
} catch(SQLException ex) {  
    System.err.println("SQLException: " + ex.getMessage());  
}  
}  
}
```

2

3

4

6



Exemplo 2: Inserir dados por JDBC

```
import java.sql.*;

public class InsertCoffees {

    public static void main(String args[]) throws SQLException {
        System.out.println ("Adding Coffee Data");
        ResultSet rs = null;
        PreparedStatement ps = null;
        String url = "jdbc:mysql://localhost/feup";
        Connection con;
        Statement stmt;
        try {
            Class.forName("org.gjt.mm.mysql.Driver"); 1
        } catch(java.lang.ClassNotFoundException e) {
            System.err.print("ClassNotFoundException: ");
            System.err.println(e.getMessage());
        }
    }
}
```

```
try {
```

```
    con = DriverManager.getConnection(url);
```

2

3

```
    stmt = con.createStatement();
```

```
    stmt.executeUpdate ("INSERT INTO COFFEES " +  
        "VALUES('São Jorge', 49, 9.99, 0, 0)");
```

4

```
    stmt.executeUpdate ("INSERT INTO COFFEES " +  
        "VALUES('Arábicas', 49, 9.99, 0, 0)");
```

```
    stmt.executeUpdate ("INSERT INTO COFFEES " +  
        "VALUES('SICAL', 49, 10.99, 0, 0)");
```

```
    stmt.executeUpdate ("INSERT INTO COFFEES " +  
        "VALUES('SICAL decaffé', 49, 10.99, 0, 0)");
```

6

```
    stmt.close();
```

```
    con.close();
```

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

```
    } catch(SQLException ex) {
```

```
        System.err.println("-----SQLException-----");
```

```
        System.err.println("SQLState: " + ex.getSQLState());
```

```
        System.err.println("Message: " + ex.getMessage());
```

```
        System.err.println("Vendor: " + ex.getErrorCode());
```

```
    }
```

```
    }
```

```
}
```



Exemplo 3: Interrogações por JDBC

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

```
public class SelectCoffees {
```

```
    public static void main(String args[]) throws SQLException {
```

```
        ResultSet rs = null;
```

```
        PreparedStatement ps = null;
```

```
        String url = "jdbc:mysql://localhost/feup";
```

```
        Connection con;
```

```
        Statement stmt;
```

```
        try {
```

```
            Class.forName("org.gjt.mm.mysql.Driver"); 1
```

```
        } catch (java.lang.ClassNotFoundException e) {
```

```
            System.err.print("ClassNotFoundException: ");
```

```
            System.err.println(e.getMessage());
```

```
        }
```

```
        try {
```

```
            con = DriverManager.getConnection(url); 2
```

```
            3 stmt = con.createStatement();
```

4 ResultSet uprs = stmt.executeQuery("SELECT * FROM COFFEES");
System.out.println("Table COFFEES:");

5 while (uprs.next()) {
 String name = uprs.getString("COF_NAME");
 int id = uprs.getInt("SUP_ID");
 float price = uprs.getFloat("PRICE");
 int sales = uprs.getInt("SALES");
 int total = uprs.getInt("TOTAL");
 System.out.print(name + " " + id + " " + price);
 System.out.println(" " + sales + " " + total);

6 }
uprs.close();
stmt.close();
con.close();

} catch(SQLException ex) {
 System.err.println("-----SQLException-----");
 System.err.println("SQLState: " + ex.getSQLState());
 System.err.println("Message: " + ex.getMessage());
 System.err.println("Vendor: " + ex.getErrorCode());

}

}

}



Tratamento de Exceções JDBC

Exceções SQL

- Quase todos os métodos JDBC podem originar uma `SQLException` em resposta a um erro de acesso a dados
- Se mais do que um erro ocorrer, eles são encadeados.
- As exceções SQL contêm:
 - Descrição do erro, `getMessage`
 - O `SQLState` (Open Group SQL specification) identificado a exceção, `getSQLState`
 - Um código de erro específico do vendedor da base de dados, `getErrorCode`
 - Uma referência para a próxima `SQLException`, `getNextException`

SQLException: Exemplo

```
try {  
    ... // JDBC statement.  
} catch (SQLException sqle) {  
    while (sqle != null) {  
        System.out.println("Message: " + sqle.getMessage());  
        System.out.println("SQLState: " + sqle.getSQLState());  
        System.out.println("Vendor Error: " +  
            sqle.getErrorCode());  
        sqle.printStackTrace(System.out);  
        sqle = sqle.getNextException();  
    }  
}
```

Resumo

- Um Driver JDBC liga uma aplicação Java a uma base de dados específica.
- Seis passos para usar JDBC:
 - Carregar o Driver
 - Estabelecer uma “Connection”
 - Criar um objecto “Statement”
 - Executar uma “Query”
 - Processar o “Result Set”
 - Fechar a “Connection”
- Garantir que as chamadas JDBC são guardadas por blocos try/catch.

Referências

- Sun JDBC Tutorial
 - <http://java.sun.com/docs/books/tutorial/jdbc/TOC.html>
- George Reese, “Database Programming with JDBC and Java” (O’Reilly & Associates.)

Exercícios...



Universidade do Porto
Faculdade de Engenharia
FEUP

Java Networking, java.net.*

Package java.net

- Java dispõe de diversas classes para manipular e processar informação em rede
- São suportados dois mecanismos básicos:
 - Sockets - troca de pacotes de informação
 - URL - mecanismo alto-nível para troca de informação
- Estas classes possibilitam comunicações baseadas em sockets:
 - permitem manipular I/O de rede como I/O de ficheiros
 - os sockets são tratados como *streams* alto-nível o que possibilita a leitura/escrita de/para sockets como se fosse para ficheiros

Package java.net

- O package contém as seguintes classes:
 - URL – encapsula um endereço WWW
 - URLconnection – uma ligação WWW
 - InetAddress – um endereço IP com nome de host
 - Socket – lado do cliente, liga a um porto, utiliza TCP
 - ServerSocket – ausculta um determinado porto por ligações de clientes (a ligação implica TCP)
 - DatagramSocket – um socket UDP, para clientes e servidores
 - DatagramPacket – empacota informação num pacote UDP com informação de encaminhamento IP

Sockets

- Um socket é um mecanismo que permite que programas troquem pacotes de bytes entre si.
- A implementação Java é baseada na da BSD Unix.
- Quando um socket envia um pacote, este é acompanhado por duas componentes de informação:
 - Um endereço de rede que especifica o destinatário do pacote
 - Um número de porto que indica ao destinatário qual o socket usar para enviar informação
- Os sockets normalmente funcionam em pares: um cliente e um servidor

Sockets e Protocolos

- Protocolos *Connection-Oriented*
 - O socket cliente estabelece uma ligação para o socket servidor, assim que é criado
 - Os pacotes são trocados de forma fiável
- Protocolos *Connectionless*
 - Melhor performance, mas menos fiabilidade
 - Exemplos de utilização: envio de um pacote, audio em tempo-real
- Comparação
 - TCP/IP utiliza sete pacotes para enviar apenas um (1/7).
 - UDP utiliza apenas um pacote (1/1).

Sockets/Protocolos em Java

	Cliente	Servidor
Connection-oriented Protocol	Socket	ServerSocket
Connectionless Protocol	DatagramSocket	DatagramSocket

Sockets em Protocolos *Connection-Oriented*

- Pseudo-código típico para um servidor:
 - Criar um objecto `ServerSocket` para aceitar ligações
 - Quando um `ServerSocket` aceita uma ligação, cria um objecto `Socket` que encapsula a ligação
 - O `Socket` deve criar objectos `InputStream` e `OutputStream` para ler e escrever bytes para e da ligação
 - O `ServerSocket` pode opcionalmente criar um novo *thread* para cada ligação, por forma a que o servidor possa aceitar novas ligações enquanto comunica com os clientes
- Pseudo-código típico para um cliente
 - Criar um objecto `Socket` que abre a ligação com o servidor, e utiliza-o para comunicar com o servidor

Exemplo: Servidor de Ficheiros

```
public class FileServer extends Thread {
    public static void main(String[] argv) {
        ServerSocket s;
        try {
            s = new ServerSocket(1234, 10);
        } catch (IOException e) {
            System.err.println("Unable to create socket");
            e.printStackTrace();
            return;
        }
        try {
            while (true) {
                new FileServer(s.accept());
            }
        } catch (IOException e) {
        }
    }
    private Socket socket;
    FileServer(Socket s) {
        socket = s;
        start();
    }
}
```

Exemplo: Servidor de Ficheiros...

```
public void run() {
    InputStream in;
    String fileName = "";
    PrintStream out = null;
    FileInputStream f;
    try {
        in = socket.getInputStream();
        out = new PrintStream(socket.getOutputStream());
        fileName = new DataInputStream(in).readLine();
        f = new FileInputStream(fileName);
    } catch (IOException e) {
        if (out != null)
            out.print("Bad: "+fileName+"\n");
        out.close();
        try {
            socket.close();
        } catch (IOException ie) {
        }
        return;
    }
    out.print("Good:\n");
    // send contents of file to client.
}
```

Exemplo: Servidor de Ficheiros...

```
public class FileClient {
    private static boolean usageOk(String[] argv) {
        if (argv.length != 2) {
            String msg = "usage is: " +
                "FileClient server-name file-name";
            System.out.println(msg);
            return false;
        }
        return true;
    }
    public static void main(String[] argv) {
        int exitCode = 0;
        if (!usageOk(argv))
            return;
        Socket s = null;
        try {
            s = new Socket(argv[0], 1234);
        } catch (IOException e) {
            //...
        }
        InputStream in = null;
        // ...
    }
}
```


Sockets em Protocolos Connectionless

- Pseudo-código típico para um servidor:
 - Criar um objecto DatagramSocket associado a um determinado porto
 - Criar um objecto DatagramPacket e pedir ao DatagramSocket para colocar o próximo bloco de dados que recebe no DatagramPacket
- Pseudo-código típico para um cliente
 - Criar um objecto DatagramPacket associado a um bloco de dados, um endereço de destino, e um porto
 - Pedir a um DatagramSocket para enviar o bloco de dados associado ao DatagramPacket para o destino associado ao DatagramSocket
- Exemplo: TimeServer

Objectos URL

- A classe URL fornece um acesso a dados a um mais alto-nível do que os sockets
- Um objecto URL encapsula um Uniform Resource Locator (URL) que uma vez criado pode ser usado para aceder a dados de um endereço especificado pelo URL
- O acesso aos dados não necessita de se preocupar com o protocolo utilizado
- Para alguns tipos de dados, um objecto URL sabe devolver os conteúdos. Por exemplo, dados JPEG num objecto ImageProducer, ou texto numa String

Criação de objectos URL

- URL's absolutos

```
try {
    URL js = new URL("http://www.javasoft.com/index.html");
} catch (MalformedURLException e) {
    return;
}
```

- URL's relativos

```
try {
    URL jdk = new
    URL(js, "java.sun.com/products/JDK/index.html");
} catch (MalformedURLException e) {
    return;
}
```

- Métodos de acesso

- `getProtocol()`, `getHost()`, `getFile()`, `getPort()`, `getRef()`, `sameFile(URL)`, `getContent()`, `openStream()`

Exercícios...



Universidade do Porto
Faculdade de Engenharia
FEUP

Reflection

Analisar Objectos com “Reflection”

- “Reflection”: mecanismo para descoberta de informação sobre, ou manipulação, de objectos e classes em tempo de execução.
- Utilizações possíveis de “reflection”:
 - Visualizar informação sobre um objecto.
 - Criar uma instância de uma classe cujo nome apenas é conhecido em tempo de execução.
 - Invocar um método arbitrário através do seu nome.

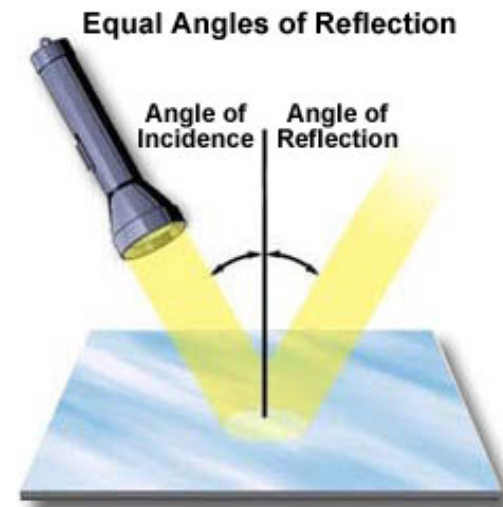


Figure 1

Reflection em Java: a classe Class

- Um objecto da classe `java.lang.Class` é uma representação de um tipo Java.
- Com um objecto `Class`, pode-se:
 - Saber informação sobre a classe
 - Conhecer os campos e métodos dessa classe
 - Criar instâncias (objectos) dessa classe
 - Descobrir as superclasses, subclasses, interfaces que implementa, etc, relativas a essa classe.

Métodos da classe Class

- `public static Class.forName(String className)`
 - Retorna um objecto Class object que representa a classe com o nome dado.
- `public String getName()`
 - Retorna o nome completo da classe deste objecto, p.e. “java.awt.Rectangle”.
- `public int getModifiers()`
 - Retorna um conjunto de flags com informação sobre a classe, como por exemplo, se é abstract, se é uma interface, etc.
- `public Object newInstance()`
 - Retorna uma nova instância do tipo representado pelo objecto Class. Assume um constructor sem argumentos.

Mais métodos da classe Class

- `public Class[] getClasses()`
- `public Constructor getConstructor(Class[] params)`
- `public Constructor[] getConstructors()`
- `public Field getField(String name)`
- `public Field[] getFields()`
- `public Method getMethod(String name, Class[] params)`
- `public Method[] getMethods()`
- `public Package getPackage()`
- `public Class getSuperClass()`

Programação com Class

- Num método toString:

```
public String toString() {  
    return "My type is " + getClass().getName();  
}
```

- Para imprimir os nomes de todos os métodos de uma classe:

```
public void printMethods() {  
    Class claz = getClass();  
    Method[] methods = claz.getMethods();  
    for (int ii = 0; ii < methods.length; ii++)  
        System.out.println(methods[ii]);  
}
```

Outras Classes de “Reflection”

Package `java.lang.reflect`

- **Field**

- `public Object get(Object obj)`
- `public <type> get<type>(Object obj)`
- `public void set(Object obj, Object value)`
- `public void set<type>(Object obj, <type> value)`

- **Constructor**

- `public Object newInstance(Object[] args)`

- **Method**

- `public Object invoke(Object obj, Object[] args)`

Obter a Class Pretendida

- Todas as classes têm um objecto Class acessível por:
 - Nome da classe seguido por `.class` (e.g. `Vector.class`)
 - Invocando `.getClass()` numa instância dum tipo (e.g. `new Vector().getClass()`)
 - Invocando `Class.forName(className)` com o nome do tipo como String (e.g. `Class.forName("java.util.Vector")`)
 - Carregar uma classe a partir de um ficheiro `.class` usando um objecto `ClassLoader`:
 - `ClassLoader loader = ClassLoader.getSystemClassLoader();`

Exemplo

```
Class cl = Class.forName("java.awt.Rectangle");
Class[] paramTypes = new Class[] {
    Integer.TYPE, Integer.TYPE};
Constructor ctor = cl.getConstructor(paramTypes);

Object[] ctArgs = new Object[] {
    new Integer(20), new Integer(40)};
Object rect = ctor.newInstance(ctArgs);

Method meth = cl.getMethod("getWidth", null);
Object width = meth.invoke(rect, null);

System.out.println("Object is " + rect);
System.out.println("Width is " + width);
```

Resultado:

```
Object is java.awt.Rectangle[x=0,y=0,width=20,height=40]
Width is 20.0
```

Reflection e Factory

```
public static Shape getFactoryShape (String s)
{
    Shape temp = null;
    if (s.equals ("Circle"))
        temp = new Circle ();
    else
        if (s.equals ("Square"))
            temp = new Square ();
        else
            if (s.equals ("Triangle"))
                temp = new Triangle ();
            else
                // ...
                // continues for each kind of shape
    return temp;
}
```



```
public static Shape getFactoryShape (String s)
{
    Shape temp = null;
    try
    {
        temp = (Shape) Class.forName (s).newInstance ();
    }
}
```

Reflection e ActionListener

```
public void actionPerformed(ActionEvent e) {  
    String command = event.getActionCommand();  
    if (command.equals("play"))  
        play();  
    else if (command.equals("save"))  
        save();  
    else ...  
}
```



```
public void actionPerformed(ActionEvent e) {  
    String command = event.getActionCommand();  
    Method meth = getClass().getMethod(command, null);  
    meth.invoke(this, null);  
}
```

Outras utilizações de “Reflection”

- JavaBeans (componentes GUI dinâmicos)
- Bases de dados JDBC
- JavaMail
- Jini
- Carregar / usar classes enviadas pela rede

Referências

- *Java Tutorial: Reflection API.*
 - <http://java.sun.com/docs/books/tutorial/reflect/>

- *O'Reilly Java Reflection Tutorial.*
 - <http://www.oreilly.com/>

Exercícios...