

# **Programação Orientada a Objectos**

## **Conceitos Gerais**

UA, DETI, Programação III  
José Luis Oliveira, Carlos Costa  
2017/18

1

## **Metodologias de Programação**

### **Programas Pequena Escala**

1. Procedimentos
2. Módulos
3. Tipos Abstractos de Dados
4. Objectos

Mudança do  
Paradigma

### **Cenários de Grande Complexidade**

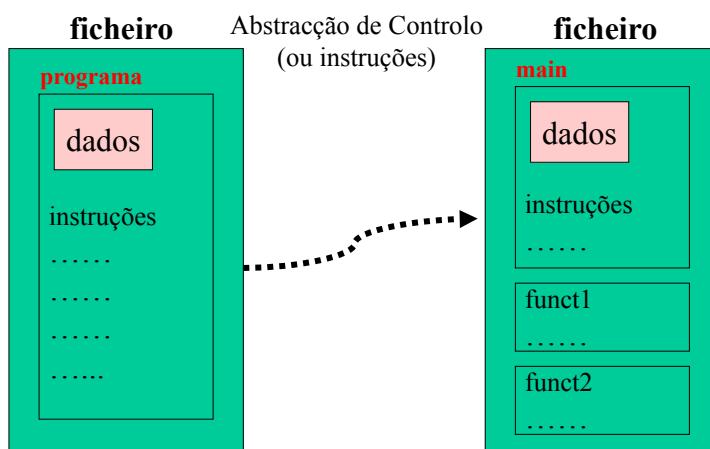
2

## 1. Procedimentos

- Programação tradicional.
- As funções/procedimentos assumem o papel estruturante dos programas.
  - Dados desempenham um papel secundário
- Polarizada para a eficiência de processamento.
- Baseia-se na filosofia:
  - Identificar as funções a utilizar e desenvolver para cada uma o melhor algoritmo
- Linguagens
  - Fortran, Algol, Pascal, C.

3

## 1. Procedimentos



### Problema:

o facto de serem unidades de código que são compilados conjuntamente com o programa principal limita a sua reutilização (só utilizando “copy & paste”).

4

## 2. Módulos

### Conceito:

- consiste em separar um programa em pequenos módulos independentes:
  - cada um deles “auto suficiente” e com relações bem definidas com outros módulos relevantes.
  - podem ser compilados separadamente;
- garantindo que:
  - diferentes pessoas podem trabalhar de forma independente em diferentes módulos;
  - a “fusão” dos vários módulos num único programa é consistente;
  - alterações num módulo terão um impacto mínimo ou nulo nos restantes;

5

## 2. Módulos

- Surgiu então um novo conceito:

Tipo de Dados

+

Conjunto de Operações Associadas

- O paradigma é:

- Decidir quais os módulos necessários.
  - Dividir o programa segundo esses módulos de forma a esconder os dados e a funções respectivas

- Linguagens:

- Modula-2, Ada.

6

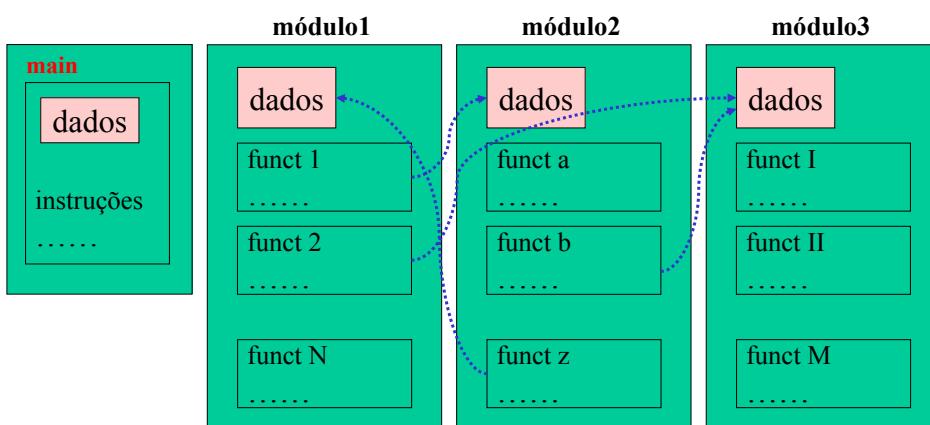
## 2. Módulos

### Vantagens

- Cada módulo pode ser desenvolvido, analisado e testado de forma independente;
- Reduz a complexidade do programa global através da implementação de mecanismos de abstracção para facilitar tarefas;
- Isolamento de domínio de responsabilidades:
  - cada módulo pode ser responsabilidade de entidades (pessoas/empresas) distintas;
- Reutilização de código:
  - ao desenvolvermos um módulo especializado numa tarefa/funcionalidade, podemos facilmente reutilizá-lo noutro programa com as mesmas necessidades;

7

## 2. Módulos



### Problema:

As funções de um módulo continuam a aceder a dados de outros módulos, comprometendo assim a desejada independência e reutilização em qualquer contexto.

8

### 3. Tipos abstractos de dados

- ADT (Abstract Data Types)
- Consiste em definir tipos que especificam propriedades e funcionalidades comuns a diversas entidades (objectos).
- Evolução da organização em Módulos
- O paradigma de programação é:
  - Decidir que tipos são necessários.
  - Fornecer um conjunto completo de operações para cada tipo
- Linguagens:
  - Ada, C++, Java.

9

### 3. Tipos abstractos de dados

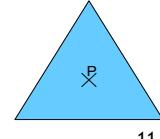
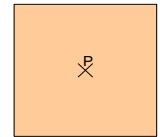
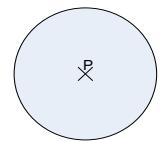
- Porquê "abstracto"?
  - Porque o Tipo de Dados fornece um conjunto de serviços (interface) bem definido (métodos, parâmetros, retorno) sem necessidade de conhecer a sua implementação
- Comparação Modelação Procedimental # ADTs
  - Na primeira dá-se relevo às estruturas de dados omitindo o seu comportamento
  - Cada Tipo Abstracto de Dados define um contracto semântico. Externamente não são conhecidos os dados nem os detalhes de construção



10

### 3. Implementação de tipos abstractos de dados - Problema

- Considerando três elementos gráficos distintos - círculos, triângulos, quadrados - é possível encontrar propriedades comuns que permitam a definição de um tipo genérico
  - O centro P e a cor C.
- Algumas funções tem implementações diferentes consoante o elemento gráfico
  - draw()
  - area()
- Problema!
  - Não distinção entre propriedades genéricas e propriedades específicas (sub-tipos de objectos).



11

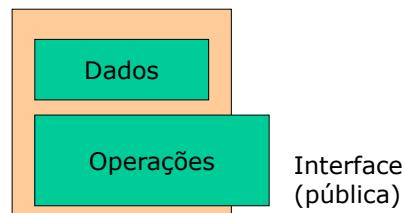
### 4. Objectos

- Ênfase nos objectos e no seu comportamento
  - “Ask not first what the system does; Ask what it does to!” (B. Meyer)
- Paradigma :
  - Decidir quais as classes necessárias
  - Fornecer um conjunto completo de operações para cada classe
  - Usar herança para expressar aspectos comuns
- Características principais da POO
  - Encapsulamento
  - Herança
  - Polimorfismo

12

## Encapsulamento

- Permite agrupar sob uma forma única de abstracção de dados um conjunto de elementos (ADT).
- Inclui os dados e métodos que manipulam esse dados (comportamento e estado).
- O utilizador de uma classe tem apenas acesso à sua interface (*information hiding*)



13

## Encapsulamento - exemplo

1

```
public class Figure {  
    private int px;  
    private int py;  
    private int cor;  
  
    public void draw();  
}
```

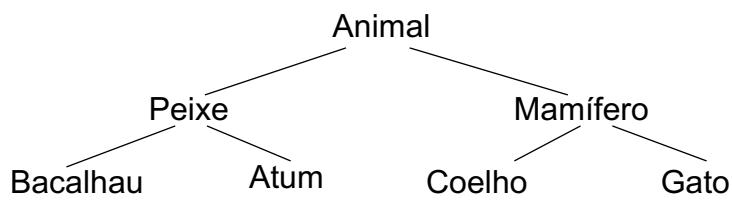
2

```
public class Point {  
    private int x;  
    private int y;  
    //...  
}  
  
public class Figure {  
    private Point p;  
    private int cor;  
  
    public void draw();  
}
```

Melhor

## Herança

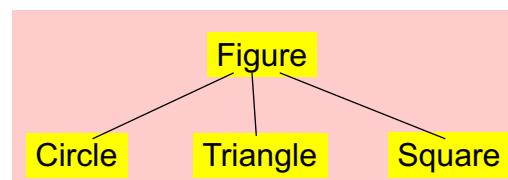
- Mecanismo através do qual várias classes podem ser relacionadas hierarquicamente.
- Explora similaridades (generalização e especialização)
- Promove reutilização e extensibilidade
- Explicita características e operações comuns



15

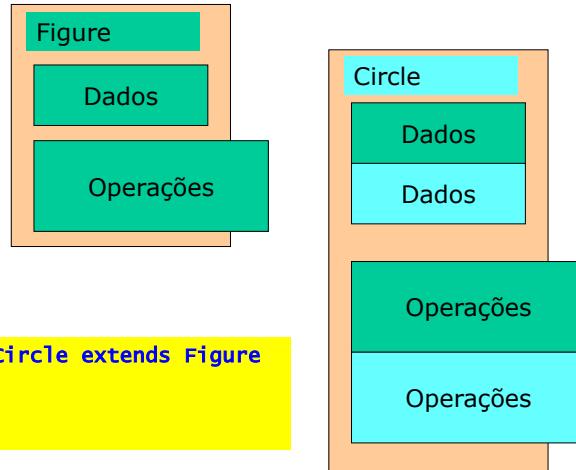
## Herança

```
public class Figure {  
    private Point p;  
    private int cor;  
  
    public void draw();  
}  
  
public class Circle extends Figure {  
    //...  
}
```



16

## Herança



17

## Polimorfismo

- Capacidade de um objecto tomar várias formas
  - Uma figura pode ser um círculo, um quadrado, um triângulo, ...
- Ligaçāo Estática vs Ligāção Dinâmica  
(*early binding vs late/dynamic binding*)
- Promove extensibilidade
- Depende da existēncia de herança

18

## ***Garbage Collection (conceito opcional)***

- Indispensável para alguns puristas da POO
  - Para outros .. nem tanto.
- Permite ao utilizador alguns mecanismos de controlo sobre a reserva e libertação de memória
- O utilizador pode implementar *garbage collection* automático através das classes (construtores e destrutores)
- Liberta o programador
  - ... mas à custa de perda de desempenho.
- Linguagens
  - Java, Smalltalk, EIFFEL.

19

## ***POO - Terminologia***

- **Classes**
  - abstracção que caracteriza múltiplas entidades com comportamentos semelhantes.
  - Outros termos: Tipo, tipo de objecto
- **Métodos**
  - funções dentro de uma classe que actuam sobre as suas instâncias.
  - Outros termos: Mensagens, Funções, Operações
- **Atributos**
  - características (dados) numa classe que diferem de objecto para objecto
  - Outros termos: Propriedades, Dados

20

## POO - Terminologia

- **Objectos**

- concretização de uma classe numa entidade particular;
- as instâncias de uma classe tem todas as mesmas operações, mesmas estruturas de dados, mas valores próprios.
- Outros termos: Variável, Instância (de uma classe)

- **Encapsulamento e visibilidade**

- Agregação de métodos e atributos numa entidade
- interface privada - constituída por dados e métodos que não são visíveis pelo exterior de cada objecto;
- interface pública - composta pelo conjunto de operações que o objecto disponibiliza.
- Outros termos: *information hiding*

21

# Java

## Encapsulamento, Classes e Objetos

UA, DETI, Programação III  
José Luís Oliveira, Carlos Costa  
2017/18

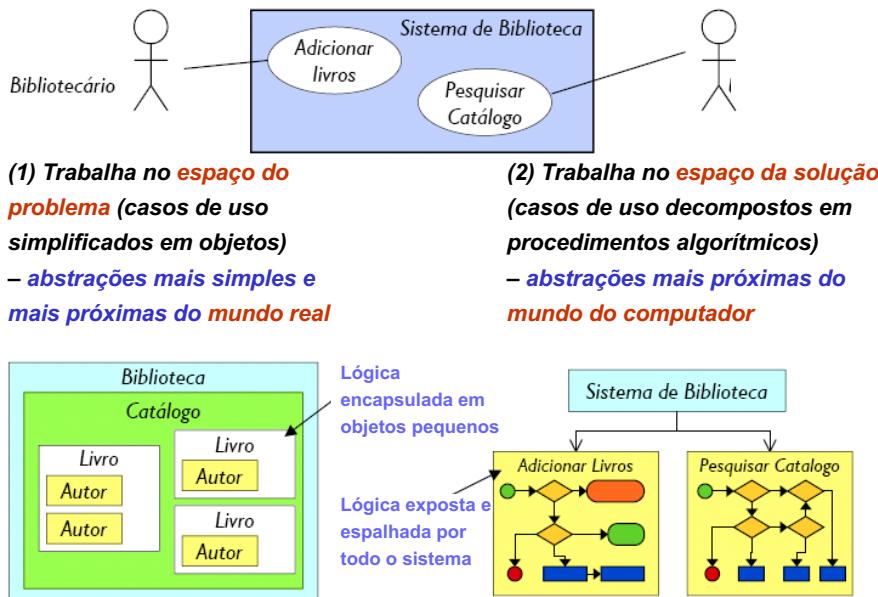
1

## O que é Orientação por Objetos?

- Paradigma do momento na engenharia de software
  - Afecta análise, projeto (design) e programação
- A **análise** orientada por objetos
  - Determina o que o sistema deve fazer: Quais os atores envolvidos? Quais as atividades a serem realizadas?
  - Decompõe o sistema em **objetos**: Quais são? Que tarefas cada objeto terá que fazer?
- O **design** orientado por objetos
  - Define como o sistema será implementado
  - Modela os relacionamentos entre os objetos e atores (pode-se usar uma linguagem específica como UML)
  - Utiliza e reutiliza abstrações como classes, objetos, funções, frameworks, APIs, padrões de projeto

2

## POO (1) versus Prog. Procedimental (2)

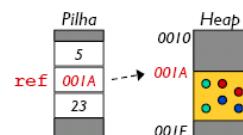


## Objetos

- Em Java os objetos são armazenados na memória heap e manipulados através de uma referência (variável), guardada na pilha.
  - Têm estado (atributos)
  - Têm comportamento (métodos)
  - Têm identidade (a referência)
- Todos os objetos são manipulados através de referências
 

```
Pessoa nome1, nome2;
nome1 = new Pessoa("Manuel");
nome2 = nome1;
```
- Todos os objetos devem ser explicitamente criados.
 

```
Circulo c1 = new Circulo(p1, 5);
String s = "Buga"; // Strings são exceção!
```
- Casos especiais: tipos primitivos
  - Não são objetos em Java



4

## Criação explícita - tipos primitivos

- Por questões de eficiência de programação e de código, podemos declarar variáveis automáticas associadas a cada um destes tipos base.
  - não são referências - são armazenadas na pilha
  - Exemplo:

```
char ch = 'B';
int x = 23;
```
- Os tipos de dados primitivos têm associadas classes "wrappers"
  - Neste caso são armazenados no heap (como todos os objetos).

```
Character ch1 = new Character(ch);
Integer idade = new Integer(23);

// Sintaxe melhor: Box/Unbox
Character ch1 = ch;
Integer idade = 23;
```

5

## Tipos primitivos em Java

- Têm apenas identidade (nome da variável) e estado (valor literal armazenado na variável)
  - - dinâmicos; + eficientes
- Classe 'Wrapper' faz transformação, se necessário

Tipo	Tamanho	Mínimo	Máximo	Default	'Wrapper'
boolean	—	—	—	false	Boolean
char	16-bit	Unicode 0	Unicode $2^{16} - 1$	\u0000	Character
byte	8-bit	-128	+127	0	Byte
short	16-bit	$-2^{15}$	$+2^{15} - 1$	0	Short
int	32-bit	$-2^{31}$	$+2^{31} - 1$	0	Integer
long	64-bit	$-2^{63}$	$+2^{63} - 1$	0	Long
float	32-bit	IEEE754	IEEE754	0.0	Float
double	64-bit	IEEE754	IEEE754	0.0	Double
void	—	—	—	—	Void

6

## Valores de omissão para tipos primitivos

- Se uma variável for utilizada como membro de uma classe o compilador encarrega-se de inicializá-la por omissão
- Isto não é garantido no caso de variáveis locais pelo que devemos sempre inicializar todas as variáveis

Tipo	Valor por omissão
boolean	false
char	'\u0000' (null)
byte	(byte)0
short	(short)0
int	0
long	0L
float	0.0f
double	0.0d

7

## Criação explícita - Vectores

- Um vector em Java representa um conjunto de referências
  - aplicam-se as regras anteriores nos valores por omissão

```
int[] x; // ou int x[] (C style); ref to int array

x = new int[10]; // reserva array

int[] a = new int[10]; // 10 int

int[] a1 = { 1, 2, 3, 4, 5 };

XptoCl[] xc = new XptoCl[10]; // 10 refs
```

8

## Vectores multidimensionais

- Vectores multidimensionais são vectores de vectores
  - É possível criar vectores rectangulares

```
int [][][] prisma = new int [3][2][2];
```

- ou criar apenas o primeiro nível e depois cada subsequente com dimensões diferentes

```
int [][][] prisma2 = new int [3][][];
prisma2[0] = new int[2][];
prisma2[1] = new int[3][2];
prisma2[2] = new int[4][4];
prisma2[0][0] = new int[5];
prisma2[0][1] = new int[3];
```

9

## Alcance/Scope

- Uma variável automática pode ser utilizada desde que é definida até ao final desse contexto
- Cada bloco pode ter os seus próprios objetos

```
{   int k;
    {   int i;
        } // 'i' não é visível aqui
    } // 'k' não é visível aqui
```

- Exemplo ilegal

```
{   int x = 12;
    {   int x = 96; /* erro! */
    }
}
```

10

## Tipos Abstractos de Dados

- Construção de modelos de objetos
  - (ADT) consiste em modular entidades de acordo com os seus serviços.
- O conceito de classe é uma forma de implementar tipos abstractos de dados em POO

```
Class UserDefinedType { /* ... */ }
```

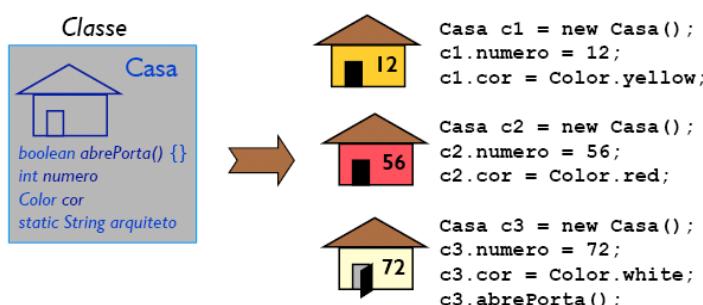
- Esta declaração introduz um novo tipo de dados

```
UserDefinedType myObj = new UserDefinedType();
```

11

## O que é uma classe?

- Classes são especificações para criar objetos
- Uma classe representa um tipo de dados complexo
- Classes descrevem
  - Tipos dos dados que compõem o objeto (o que podem armazenar)
  - Métodos que o objeto pode executar (o que podem fazer)



12

## Exemplo de classe

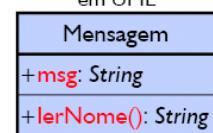
Definição da classe (tipo) Mensagem em Java

```
public class Mensagem {
    public String msg = "";

    public String lerNome() {
        String nomeEmMaiusculas =
            msg.toUpperCase();
        return nomeEmMaiusculas;
    }
}
```

Membros de classe

Representação em UML



Esta é a interface pública da classe. É só isto que interessa a quem vai usá-la. Os detalhes (código) estão encapsulados.

13

## Exemplo de classe

public class Casa {

```
    private Porta porta;
    private int numero;
    public java.awt.Color cor;
```

Atributos de instância: cada objeto poderá armazenar valores diferentes nessas variáveis.

```
    public Casa() {
        porta = new Porta();
        numero = ++contagem * 10;
    }
```

Procedimento de inicialização de objetos (Construtor): código é executado após a criação de cada novo objeto. Cada objeto terá um número diferente.

```
    public void abrePorta() {
        porta.abre();
    }
```

Método de instância: só é possível chamá-lo se for através de um objeto.

```
    public static String arquiteto = "Zé";
    private static int contagem = 0;
```

Atributos estáticos: não é preciso criar objetos para usá-los. Todos os objetos os compartilham.

```
    static {
        if ( condição ) {
            arquiteto = "Og";
        }
    }
```

Procedimento de inicialização estático: código é executado uma única vez, quando a classe é carregada. O arquitecto será um só para todas as casas: ou Zé ou Og.

## Métodos

Métodos, mensagens, funções, procedimentos

- A invocação é sempre efectuada através da notação de pontos.

```
myObj.add(25);
deti.abrePorta();
```

- O receptor da mensagem está sempre à esquerda.
- O receptor é sempre uma classe ou uma referência para um objeto.

15

## Elementos estáticos

- As variáveis estáticas, ou variáveis de classe, são comuns a todos os objetos dessa classe.
- A sua declaração é precedida por **static**.
- A invocação é feita através do identificador da classe

```
class S {
    public static int a=23;
    public static void class_function() { ... }
    // ...
}

S.class_function(); // invocada sobre a classe
S s1 = new S();
S s2 = new S();
System.out.println(S.a);
S.a++; // s1.a e s2.a será 24
```

16

## Espaço de Nomes - Package

- Em Java a gestão do espaço de nomes é efectuado através do conceito de **package**.
- Porque gestão de espaço de nomes?

→ Evita conflitos de nomes de classes

- Não temos geralmente problemas em distinguir os nomes das classes que construímos.
- Mas como garantimos que a nossa classe Book não colide com outra que eventualmente possa já existir?

17

## Package e import

- Utilização
  - As classes são referenciadas através dos seus nomes absolutos ou utilizando a primitiva **import**.  
`import java.util.ArrayList  
import java.util.*`
  - A cláusula *import* deve aparecer sempre na primeira linha de um programa.
- Quando escrevemos,  
`import java.util.*;`  
estamos a indicar um caminho para um pacote de classes permitindo usá-las através de nomes simples:  
`ArrayList al = new ArrayList();`
- De outra forma teríamos de escrever:  
`java.util.ArrayList al = new java.util.ArrayList();`

18

## Criar um package

- Na primeira linha de código:  
`package p3;`
  - garante que a classe pública dessa unidade de compilação (Sock por exemplo) fará parte do package p3.
- O espaço de nomes é baseado numa estrutura de sub-directórios
  - Este package vai corresponder a uma entrada de directório:  
`$CLASSPATH/p3`
  - Boa prática usar DNS invertido  
`pt.ua.deti.p3`
- A sua utilização será na forma:  
`p3.Sock sr = new p3.Sock();`
  - OU  
`import p3.*`  
`Sock sr = new Sock();`

19

## Características de uma Aplicação

- Qualquer aplicação é uma classe
- Tem um método estático designado por **main**

```
// BomDia.java
import java.util.Date;

public class BomDia {
    public static void main(String args[]) {
        System.out.print("Hello, current date is ");
        System.out.println(new Date());
    }
}
```
- Este ficheiro deve ser gravado com o nome da sua classe pública com a extensão .java (**BomDia.java**)
- A compilação vai gerar ficheiros .class - um por cada classe constante no programa.

20

## I/O - Leitura

- Classe Scanner

```
Scanner in = new Scanner(System.in);
String name = in.nextLine();
```

- System.in

```
BufferedReader in =
    new BufferedReader(
        new InputStreamReader(System.in));
String name = in.readLine();
```

21

## I/O - Escrita

```
...
System.out.println("Bom Dia!");
...
```

- **System** é o nome de uma classe
- **out** é um objeto estático do tipo PrintStream membro de System
  - System.out
  - System.err
  - System.in
- **println** é um método estático da classe PrintStream

22

## String

- **String** é uma classe que manipula cadeias de caracteres.
- Não é um array de char
 

```
String v = "Aveiro";
for (int i=0; i<v.length(); i++)
    System.out.print(v[i]); // ERRO ! usar v.charAt(i)
```
- É imutável
 

```
v[1] = 'b'; //errado!
```

  - Se precisarmos de strings mutáveis, devemos usar a classe **StringBuilder** (ou **StringBuffer**, thread-safe)
- Podemos criar String de várias formas

```
String s2 = "ABCD";
// ou
String s1 = new String("ABCD"); // Não é boa ideia! Porquê?
char[] caracteres = {'A','B','C','D'};
String s3 = new String(caracteres);
```

- → devemos evitar criar objetos duplicados

23

## String - operações

- Concatenação
 

```
String s1 = "Programação" + " 3";
```
- Conversão implícita
 

```
String s1 = "Programação" + 3;
System.out.println(s1+112);
System.out.println(112+s1);
```
- Comparação
 

```
if (s1.equals(s2))
    //... true or false

int greater = s1.compareTo("Programação 4");
// -1 (s1 menor), 0(iguais), 1 (s1 maior)

if (s1 == s2)
    //... s1 e s2 referenciam o mesmo objecto.
String s1 = "AAAA";
String s2 = "AAAA";
if (s1 == s2) System.out.println("Iguais!!"); //Porquê?
```

24

## toString

- Todos os objetos em Java entendem a mensagem `toString()`
- Exemplo

```
c1 = new Circulo(0, 0, 5);
System.out.println(c1); // c1.toString() Circle@lafa3
```

- Geralmente é necessário redefinir este método de modo a fornecer um resultado mais adequado.

```
@Override
public String toString() { // Circulo.toString()
    return "Centro: (" + x + "," + y + ") Raio: " + raio;
}
```

Centro: (0,0) Raio: 5

25

## Compilar e executar

- J2SE, J2EE

```
javac BomDia.java
java BomDia
```

- IDEs

- Eclipse
- NetBeans
- vim
- ... e muitos outros

26

## Comentários

- múltiplas linhas

```
/* Isto é um comentário em
múltiplas linhas
*/
```

- até ao fim de linha

```
// Isto é um comentário até ao final de linha
```

- javadoc

- Permite gerir comentários e documentação simultaneamente
- O compilador de javadoc gera documentos HTML

```
/** Todos os comandos javadoc são delimitados desta forma */
```

27

## Javadoc

- Existem 3 tipos de comentários de documentação

```
/** A class comment */
public class docTest {
    /** A variable comment */
    public int i;
    /** A method comment */
    public void f() {}
}
```

- É possível utilizar HTML dentro destes comentários
  - excepto headings que são inseridos pelo compilador javadoc.
- Outro modo é utilizar "doc tags"

28

## Javadoc - tags

- @see - referência a outras classes ("See Also")

```
@see classname
@see fully-qualified-classname
@see fully-qualified-classname#method-name
```

- Documentação de classes

```
@version
@author
@since
```

- Documentação de métodos

```
@param
@return
@throws
@deprecated
```

29

## JavaDoc - example

```
/**
 * Returns an Image object that can then be painted on the screen.
 * The url argument must specify an absolute {@link URL}. The name
 * argument is a specifier that is relative to the url argument.
 * <p>
 * This method always returns immediately, whether or not the
 * image exists. When this applet attempts to draw the image on
 * the screen, the data will be loaded. The graphics primitives
 * that draw the image will incrementally paint on the screen.
 *
 * @param url an absolute URL giving the base location of the image
 * @param name the location of the image, relative to the url argument
 * @return the image at the specified URL
 * @see Image
 */
public Image getImage(URL url, String name) {
    try {
        return getImage(new URL(url, name));
    } catch (MalformedURLException e) {
        return null;
    }
}
```

30

## JavaDoc - result

### **getImage**

```
public Image getImage(URL url,
                      String name)
```

Returns an `Image` object that can then be painted on the screen. The `url` argument must specify an absolute URL. The `name` argument is a specifier that is relative to the `url` argument.

This method always returns immediately, whether or not the image exists. When this applet attempts to draw the image on the screen, the data will be loaded. The graphics primitives that draw the image will incrementally paint on the screen.

**Parameters:**

`url` - an absolute URL giving the base location of the image.

`name` - the location of the image, relative to the `url` argument.

**Returns:**

the image at the specified URL.

**See Also:**

`Image`

31

## Resultado javadoc - métodos

### Constructor Summary

<a href="#">HelloApplet()</a>	
-------------------------------	--

### Method Summary

static void	<a href="#">main(java.lang.String[] args)</a>
-------------	---

	void <a href="#">paint(java.awt.Graphics g)</a>
--	---

### Methods inherited from class java.applet.Applet

destroy, getAppletContext, getAppletInfo, getAudioClip, getAudioClip, getCodeBase, getDocumentBase, getImage, getImage, getLocale, getParameter, getParameterInfo, init, isActive, newAudioClip, play, play, resize, resize, setStub, showStatus, start, stop
---

32

## Estilo de escrita de código

- Classes

```
String, Player, Student, AveiroStudent
```

- Métodos e variáveis

```
area, fillArea()
```

- Constantes

```
PI, SPEED_LIMIT
```

- Estrutura do código

```
class AllTheColorsOfTheRainbow {
    int anIntegerRepresentingColors;
    void changeTheHueOfTheColor(int newHue) {
        // ...
    }
    // ...
}
```

- Bons exemplos:

- <https://google-styleguide.googlecode.com/svn/trunk/javaguide.html>

33

## Sumário

- Elementos genéricos da linguagem
- Referências
- Tipos primitivos
- Classes
- Escrita de um programa
- Documentação

34

## Java Inicialização e Limpeza de Objetos

35

## Programação Insegura

- Muitos dos erros de programação resultam de:
  - dados não inicializados - alguns programas/bibliotecas precisam de inicializar componentes e fazem depender no programador essa tarefa.
  - gestão incorreta de memória dinâmica - "esquecimento" em libertar memória, reserva insuficiente,...
- Para resolver estes dois problemas a linguagem Java utiliza os conceitos de:
  - **construtor**
  - **garbage collector**

36

## Construtor

- A inicialização de um objeto pode implicar a inicialização simultânea de diversos tipos de dados.
- Uma função membro especial, **construtor**, é invocada sempre que um objeto é criado.
- A instanciação é feita através do operador **new**.

```
Carro c1 = new Carro();
```



- O construtor é identificado pelo mesmo nome da classe.
- Este método pode ser **overloaded** (sobreposto) de modo a permitir diferentes formas de inicialização.

```
Carro c2 = new Carro("Ferrari", "430");
```



37

## Construtor

- Não retorna qualquer valor
- Assume sempre o nome da classe
- Pode ter parâmetros de entrada
- É chamado apenas uma vez: na criação do objeto

```
public class Produto {
    public static int total = 0;
    public int serie = 0;
    public Produto() {
        serie = total + 1;
        total = serie;
    }
}

public class Livro {
    private String titulo;
    public Livro() {
        titulo = "Sem titulo";
    }
    public Livro(String umTitulo) {
        titulo = umTitulo;
    }
}
```

38

## Construtor - dummy example

```
// SimpleConstructor.java
// Demonstration of a simple constructor.

class Rock {
    Rock() { // This is the constructor
        System.out.println("Creating Rock");
    }
}

public class SimpleConstructor {
    public static void main(String[] args) {
        for(int i = 0; i < 10; i++)
            new Rock();
    }
}
```

Creating Rock  
 Creating Rock

39

## Construtor por omissão

- Um construtor sem parâmetros é designado por *default constructor* ou construtor por omissão.
- Este tipo de construtor é automaticamente criado pelo compilador caso não seja especificado nenhum construtor.

```
class Machine {
    int i;
}
Machine m = new Machine(); // ok
```

- No entanto, se houver pelo menos um construtor associado a uma dada classe, o compilador já não cria o de omissão.

```
class Machine {
    int i;
    Machine(int ai) { i= ai; }
}
Machine m = new Machine(); // erro!
```

40

## Questões?

- Qual o valor dos atributos de um objeto quando não foi definido nenhum construtor?

```
class Point
{
    public void display() {...}
    private double x, y;
};

Point p1 = new Point();
```

- Quais os valores de x e y ?
- É obrigatório iniciá-los no construtor?

41

## Sobreposição (Overloading)

- Podemos usar o mesmo nome em várias funções, desde que tenham argumentos distintos e que conceptualmente executem a mesma ação

```
void sort(int[] a);
void sort(Lista l);
void sort(Set s);
```

- A ligação estática verifica a assinatura da função (nome + argumentos)
- Não é possível distinguir funções pelo valor de retorno
  - porque é permitido invocar, p.e., void f() ou int f() na forma f() em que o valor de retorno não é usado

42

## Construtores sobrepostos

- Permitem diferentes formas de iniciar um objeto de uma dada classe.

```
public class Circulo {
    public Circulo(double x, double y, double r)  {...}
    public Circulo(double r) {...}
    public Circulo() {...}
}

Circulo c1 = new Circulo(12, 43, 2.5);
Circulo c2 = new Circulo(2.5);
Circulo c3 = new Circulo();
```

43

## Sobreposição com base em tipos primitivos

- Alguns tipos primitivos podem ser automaticamente promovidos para outros de maior dimensão
  - podemos encontrar alguns problemas com a sobreposição de métodos.
- Exemplo:

```
void f1(char x) {System.out.println("char");}
void f1(byte x) {System.out.println("byte");}
void f1(short x) {System.out.println("short");}
void f1(int x) {System.out.println("int");}
void f1(long x) {System.out.println("long");}
void f1(float x) {System.out.println("float");}
void f1(double x) {System.out.println("doub");}
```

- Se não existir a função  $f1(tipoN\ x)$  a ligação estática vai promover o  $tipoN$  ao seguinte maior que esteja definido

44

## A referência this

- A referência **this** pode ser utilizada internamente a cada objeto de forma a referenciar esse mesmo objeto

```
class Torneira {
    void fecha() { /* ... */ }
    void tranca() { fecha(); /* ou this.fecha() */ }
}

class Circulo {
    double x, y, raio;
    public Circulo(double x, double y, double r)
    {
        this.x = x; this.y = y; raio = r;
    }
}
```

45

## A referência this

- Outra utilização da referência **this** é para retornar, num dado método, a referência para esse objeto.
  - pode ser usado em cadeia (lvalue).

```
public class Contador {
    int i = 0;
    Contador increment() {
        i++; return this;
    }
    void print() {
        System.out.println("i = " + i);
    }
    public static void main(String[] args) {
        Contador x = new Contador();
        x.increment().increment().increment().print();
    }
}
```

46

## Invocar um construtor dentro de outro

- Quando escrevemos vários construtores podemos chamar um dentro de outro.
  - a referência `this` permite invocar sobre o mesmo objeto um outro construtor.

```
public Circulo(double x, double y, double r) {
    this.x = x; this.y = y; raio = r;
}
public Circulo(double r) { this(0.0, 0.0, r); }
public Circulo() { this(0.0, 0.0, 1.0); }
```

Duas formas:

  - `this.método()`
  - `this(..)`
- Esta forma só pode ser usada dentro de construtores;
  - neste caso `this` deve ser a primeira instrução a aparecer;
  - não é possível invocar mais do que um construtor `this`.

47

## O conceito *static*

- Os métodos estáticos não tem associado a referência `this`.
- Assim, não é possível invocar métodos não estáticos a partir de métodos estáticos.
- É possível invocar um método estático sem que existam objetos dessa classe.
- Os métodos **static** têm a semântica das funções globais (não estão associadas a objetos).

48

## Destruitor

- Mecanismo complementar ao construtor
  - Responsável por desfazer reservas de memória (por exemplo)
- Em algumas linguagens este é um método explícito semelhante ao construtor
  - Em C++ (por exemplo)
 

```
class String {
public:
    ~String();    // destrutor
    // ...
};

String::~String() { delete [] str; }
```
  - Nesta situação o destrutor é invocado sempre que um objeto é destruído (implícita ou explicitamente)
- Em Java a limpeza de objetos é realizada pelo *garbage collector*

49

## Finalização

- O *garbage collector* é responsável por limpar os objetos não referenciados.
  - Só liberta memória
- Em Java é possível utilizar (não invocar!) uma função, `finalize()`, que é chamada sempre que o g.c. inicia a destruição do objeto.
- Esta função não funciona 95% das vezes - não deve ser usada!
  - Uma vez que não temos controlo sobre a actuação do g.c. não há garantia que um dado objeto seja libertado antes de o programa terminar.
- Se for necessário finalização, o código deve ser colocado num bloco try {...} finally {...}
- Podemos ter necessidade de construir, numa dada classe, um método explícito de limpeza
  - `Connection.close()`

50

## Finalização

```
public class Example {  
    public synchronized void cleanup() {  
        // put your clean up code here, delete files, close connections..  
    }  
    public void finalize() {  
        cleanup();  
        super.finalize();  
    }  
}  
// ...  
Example example;  
try {  
    example = new Example();  
    // ... do whatever logic you need to  
}  
finally {  
    example.cleanup();  
}
```

51

## Questões?

- Quando é que um objeto pode ser eliminado pelo g.c.?

52

## Inicialização de membros

- A inicialização de variáveis (de tipos primitivos ou não primitivos) pode ser feita na sua declaração.

```
class Measurement {
    boolean b = true; // primitivos
    char c = 'x';
    byte B = 47;
    short s = 0xff;
    int i = 999;
    long l = 1;
    float f = 3.14f;
    double d = 3.14159;
    Superficie s = new Superficie(); // class
    int i = f(); // método
    ...
}
```

53

## Inicialização no Construtor

- A inicialização anterior é comum a todos os objetos da classe
- Outra solução (mais comum) é utilizar o construtor.

```
class Measurement {
    int i;
    char c;
    Measurement(int im, char ch) {
        i = im; c = ch;
    }
}
```

- Dentro de uma classe a ordem de inicialização é determinada pela ordem das definições das variáveis.
- Mesmo que as variáveis apareçam misturada com os métodos serão sempre inicializadas primeiro.

54

## Exemplo

```

class Tag { Tag(int m) { System.out.println(m); }

1   class Card {
4     Tag t1 = new Tag(1); // Before constructor
2     Tag t2 = new Tag(2); // After constructor
3     Tag t3 = new Tag(3); // At end
5   }

public class OrderOfInitialization {
  public static void main(String[] args) {
    0   Card t = new Card();
    5   t.f(); // Shows that construction is done
  }
}

```

1  
2  
3  
33  
f()

55

## Inicialização de membros estáticos

- Se existir inicialização de membros estáticos esta toma prioridade sobre todas as outras.
- Um membro estático só é inicializado quando a classe é carregada (e só nessa altura)
  - quando for criado o primeiro objeto dessa classe ou quando for usada pela primeira vez.
- Podemos usar um bloco especial - inicializador estático - para agrupar as inicializações de membros estáticos

```

class Circulo {
  static private double lista[ ] = new double[100];
  static { // inicializador estático
    // inicialização de lista[ ]
  }
}

```

56

## Sumário

- Inicialização e Limpeza
- Construtores
- Garbage collector
- Finalização
- Referência *this*
- Inicialização estática

57

## Java Encapsulamento

58

## Encapsulamento

- Ideias fundamentais da POO
  - Encapsulamento (*Information Hiding*)
  - Herança
  - Polimorfismo
- Encapsulamento
  - Separação entre aquilo que não pode mudar (interface) e o que pode mudar (implementação)
  - Controlo de visibilidade da interface (*public, default, protected, private*)

59

## Encapsulamento

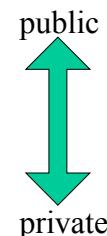
- Permite criar diferentes níveis de acesso aos dados e métodos de uma classe.
- Os níveis de controlo de acesso que podemos usar são, do maior para o menor acesso:
  - *public* - pode ser usado em qualquer classe
  - "omissão" - visível dentro do mesmo package
  - *protected* - visível dentro do mesmo package e classes derivadas
  - *private* - apenas visível dentro da classe

60

## Exemplo

```
class X {
    public void pub1() { /* . . . */ }
    public void pub2() { /* . . . */ }
    private void priv1() { /* . . . */ }
    private void priv2() { /* . . . */ }
    private int i;
    // ...
}

class XUser {
    private X myX = new X();
    public void teste() {
        myX.pub1(); // OK
        // myX.priv1(); Errado!
    }
}
```



- Um método de uma classe tem acesso a toda a informação e a todos os métodos dessa classe

61

## Modificadores>Selectores

- O encapsulamento permite esconder os dados internos de um objeto
  - Mas, por vezes é necessário aceder a estes dados diretamente (leitura e/ou escrita).
- **Regras importantes!**
  1. Todos os atributos deverão ser privados.
  2. O acesso à informação interna de um objeto (parte privada) deve ser efectuada sempre, através de funções da interface pública.

**porquê?**

62

## Modificadores>Selectores

- Selector

- Devolve o valor actual de um atributo

```
public float getRadius() { // ou public float radius()
    return radius;
}
```

- Modificador

- Modifica o estado do objeto

```
public void setRadius(float newRadius) {
    // ou public void radius(float newRadius)
    this.radius = newRadius;
}
```

63

## Métodos privados

- Internamente uma classe pode dispor de diversos métodos privados que só são utilizados internamente por outros métodos da classe.

```
// exemplo de funções auxiliares numa classe
class Screen {
    private int row();
    private int col();
    private int remainingSpace();
    // ...
};
```

64

## O que pode conter uma classe

- A definição de uma classe pode incluir:
  - zero ou mais declarações de atributos de dados
  - zero ou mais definições de métodos
  - zero ou mais construtores
  - zero ou mais blocos de inicialização static
  - zero ou mais definições de classes ou interfaces internas
- Esses elementos só podem ocorrer dentro do bloco ‘class NomeDaClasse { ... }’
  - “tudo pertence” a alguma classe
  - apenas ‘import’ e ‘package’ podem ocorrer fora de uma declaração ‘class’ (ou ‘interface’)

65

## Métodos - resumo

```
class Point {
    public Point() {...}
    public Point(double x, double y) {...}

    public void set (double newX, double newY) {...}
    public void move (double deltaX, double deltaY) {...}

    public double getX() {...}
    public double getY() {...}
    public double DistanceTo(Point p) {...}
    public void Display() {...}

    private double x;
    private double y;
}
```

66

## Questões?

- No caso particular da classe Point será esta uma boa implementação?

```
class Point {
    public Point() {...}
    public Point(double x, double y) {...}

    public void set (double newX, double newY) {...}
    public void move (double deltaX, double deltaY) {...}

    public double getX() {...}
    public double getY() {...}
    public double distanceTo(Point p) {...}
    public void display() {...}

    private double x;
    private double y;
}
```

67

## Boas práticas

- A semântica de construção de um objeto deve fazer sentido
 

```
Pessoa p = new Pessoa();     ⊕
Pessoa p = new Pessoa("António Nunes"); ⊕
Pessoa p = new Pessoa("António Nunes", 12244, dataNasc); ⊕
```
- Devemos dar o mínimo de visibilidade pública no acesso a um objeto
  - Apenas a que for estritamente necessária
- Por vezes, faz mais sentido criar um novo objeto do que mudar os atributos existentes

```
Point p1 = new Point(2,3);
p1.set(4,5); ⊕
```

68

## Boas práticas

- Juntar membros do mesmo tipo
  - Não misturar métodos estáticos com métodos de instância
- Declarar as variáveis antes ou depois dos métodos
  - Não misturar métodos, construtores e variáveis
- Manter os construtores juntos, de preferência no início
- Se for necessário definir blocos static, definir apenas um no início ou no final da classe.
- A ordem dos membros não é importante, mas seguir convenções melhora a legibilidade do código

69

## Sumário

- Encapsulamento
- Níveis de visibilidade
- Métodos
  - Modificadores
  - Selectores
  - Privados

70

# Java Herança

UA, DETI, Programação III  
José Luis Oliveira, Carlos Costa  
2017/18

1

## Relações entre Classes

- Parte do processo de modelação em classes consiste em:
  - Identificar entidades candidatas a classes
  - Identificar relações entre estas entidades
- As relações entre classes identificam-se facilmente recorrendo a alguns modelos reais.
  - Por exemplo, um RelógioDigital e um RelógioAnalógico são ambos tipos de Relógio (**especialização ou herança**).
  - Um RelógioDigital, por seu lado, contém uma Pilha (**composição**).
- Relações:
  - IS-A
  - HAS-A

2

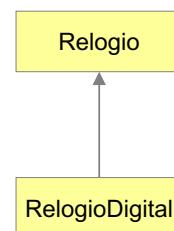
1

## Herança (IS-A)

- **IS-A** indica especialização (herança) ou seja, quando uma classe é um sub-tipo de outra classe.
- Por exemplo:

- Pinheiro é uma (IS-A) Árvore.
- Um RelógioDigital é um (IS-A) Relógio.

```
class Relogio {  
    /* ... */  
}  
  
class RelogioDigital extends Relogio {  
    /* ... */  
}
```



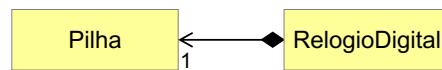
3

## Composição (HAS-A)

- **HAS-A** indica que uma classe é composta por objetos de outra classe.
- Por exemplo:

- Floresta contém (HAS-A) Árvores.
- Um RelógioDigital contém (HAS-A) Pilha.

```
class Pilha {  
    /* ... */  
}  
class RelogioDigital extends Relogio {  
    Pilha p;  
    /* ... */  
}
```



4

## Reutilização de classes

- Sempre que necessitamos de uma classe, podemos:
  - Recorrer a uma classe já existente que cumpre os requisitos
  - Escrever uma nova classe a partir "do zero"
  - Reutilizar uma classe existente usando composição
  - Reutilizar uma classe existente através de herança

5

## Identificação de Herança

- Sinais típicos de que duas classes têm um relacionamento de herança
  - Possuem aspectos comuns (dados, comportamento)
  - Possuem aspectos distintos
  - Uma é uma especialização da outra
- Exemplos:
  - Rato é um Mamífero
  - BTT é uma Bicicleta
  - Cerveja é uma Bebida

6

## Questões?

- Quais as relações entre:
  1. Trabalhador, Motorista, Vendedor, Administrativo e Contabilista
  2. Quadrado, Triângulo, Retângulo, e Losango
  3. Professor, Aluno e Funcionário
  4. Autocarro, Viatura, Roda, Motor, Pneu, Jante

7

## Questões?

- Modelar stock de uma livraria...
  - Livro
  - Artigo
  - Jornal
  - Publicação
  - Autor
  - Periódico
  - Editora
  - LivroEditado
  - Revista

8

## Questões?

- Modelar os *gadgets* de casa...
  - Telemóvel
  - Reprodutor de Áudio
  - Bateria
  - Carregador
  - MP3
  - Auscultador
  - Calculadora

9

## Herança - Conceitos

- A herança é uma das principais características da POO
- A classe CDeriv herda, ou é derivada, de CBase quando CDeriv representa um sub-conjunto de CBase
- A herança representa-se na forma:  
`class CDeriv extends CBase { /* ... */ }`
- CDeriv tem acesso aos dados e métodos de CBase
  - que não sejam privados em CBase
- Uma classe base pode ter múltiplas classes derivadas mas uma classe derivada não pode ter múltiplas classes base
  - Em Java não é possível a herança múltipla
- Terminologia
  - B é a classe Base / A é derivada de B
  - B é a classe Mãe (Parent) / A é a classe Filha (Child)
  - B é a classe Super / A é a classe Sub

10

## Herança - Exemplo

```
package heranca;
class Person {
    private String name;
    Person(String n) { name = n; }
    public String name() { return name; }
    public String toString() { return "PERSON"; }
}
class Student extends Person {
    private int nmecc;
    Student(String s, int n) { super(s); nmecc=n; }
    public int num() { return nmecc; }
    public String toString() { return "STUDENT"; }
}
public class Test {
    public static void main(String[] args) {
        Person p = new Person("Joaquim");
        Student stu = new Student("Andreia", 55678);
        System.out.println(p + " : " + p.name());
        System.out.println(stu + " : " + stu.name() + ", " + stu.num());
    }
}
```

11

Base

Derivada

PERSON : Joaquim  
STUDENT : Andreia, 55678

## Herança - Exemplo

```
class Art {
    Art() {
        System.out.println("Art constructor");
    }
}

class Drawing extends Art {
    Drawing() {
        System.out.println("Drawing constr.");
    }
}

public class Cartoon extends Drawing {
    Cartoon() {
        System.out.println("Cartoon constr.");
    }
}

public static void main(String[] args) {
    Cartoon x = new Cartoon();
}
```

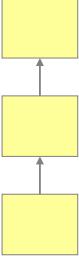
Art constructor  
Drawing constr.  
Cartoon constr.

A construção é feita a partir  
da classe base

12

## Construtores com parâmetros

- Em construtores com parâmetros o construtor da classe base é a primeira instrução a aparecer num construtor da classe derivada.



```
class Game {  
    Game(int i) { System.out.println("Game"); }  
}  
class BoardGame extends Game {  
    BoardGame(int i) { super(i);  
        System.out.println("BoardGame");  
    }  
}  
public class Chess extends BoardGame {  
    Chess() { super(11); System.out.println("Chess"); }  
    public static void main(String[] args) {  
        Chess x = new Chess();  
    }  
}
```



13

## Herança de Métodos

- Ao herdar métodos podemos:
  - mantê-los inalterados,
  - acrescentar-lhe funcionalidades novas ou
  - redefini-los

14

## Herança de Métodos - herdar

```
class Person {  
    private String name;  
    Person(String n) { name = n; }  
    public String name() { return name; }  
    public String toString() { return "PERSON"; }  
}  
class Student extends Person {  
    private int nmec;  
    Student(String s, int n) { super(s); nmec=n; }  
    public int num() { return nmec; }  
    public String toString() { return "STUDENT"; }  
}  
public class Test {  
    public static void main(String[] args) {  
        Student stu = new Student("Andreia", 55678);  
        System.out.println(stu + " : " +  
            stu.name() + ", " + stu.num());  
    }  
}
```

15

## Herança de Métodos - redefinir

```
class Person {  
    private String name;  
    Person(String n) { name = n; }  
    public String name() { return name; }  
    public String toString() { return "PERSON"; }  
}  
  
class Student extends Person {  
    private int nmec;  
    Student(String s, int n) { super(s); nmec=n; }  
    public int num() { return nmec; }  
    public String toString() { return "STUDENT"; }  
}
```

16

## Herança de Métodos - estender

```
class Person {  
    private String name;  
    Person(String n) { name = n; }  
    public String name() { return name; }  
    public String toString() { return "PERSON"; }  
}  
  
class Student extends Person {  
    private int nmec;  
    Student(String s, int n) { super(s); nmec=n; }  
    public int num() { return nmec; }  
    public String toString()  
        { return super.toString() + " STUDENT"; }  
}
```

17

## Herança e controlo de acesso

- Não podemos reduzir a visibilidade de métodos herdados numa classe derivada
  - Métodos declarados como *public* na classe base devem ser *public* nas subclasses
  - Métodos declarados como *protected* na classe base devem ser *protected* ou *public* nas subclasses. Não podem ser *private*
  - Métodos declarados sem controlo de acesso (default) não podem ser *private* em subclasses
  - Métodos declarados como *private* não são herdados

18

## Final

- O classificador final indica "não pode ser mudado"
- A sua utilização pode ser feita sobre:
  - Dados - constantes

```
final int i1 = 9;
```
  - Métodos - não redefiníveis

```
final int swap(int a, int b) { //:
```
  - Classes - não herdadas

```
final class Rato { //...
```
- "final" fixa como constantes atributos de tipos primitivos mas não fixa objetos nem arrays
  - nestes casos o que é constante é simplesmente a referência para o objeto

19

```
class Value { int i = 1; }
public class FinalData {
    // Can be compile-time constants
    private final int i1 = 9;
    private static final int VAL_TWO = 99;
    // Typical public constant:
    public static final int VAL_THREE = 39;

    public final int i4 = (int)(Math.random()*20);
    public static final int i5 = (int)(Math.random()*20);

    private Value v1 = new Value();
    private final Value v2 = new Value();
    private final int[] a = { 1, 2, 3, 4, 5, 6 }; // Arrays

    public static void main(String[] args) {
        FinalData fd1 = new FinalData();
        //! fd1.i1++; // Error: can't change value
        fd1.v2.i++; // Object isn't constant!
        fd1.v1 = new Value(); // OK -- not final
        for(int i = 0; i < fd1.a.length; i++)
            fd1.a[i]++; // Object isn't constant!
        //! fd1.v2 = new Value(); // Can't change ref
        //! fd1.a = new int[3];
    }
}
```

20

## Final - Dados

- Os dados final podem ser inicializados dentro do construtor

```
class Dummy { }

class BlankFinal {
    final int i = 0; // Initialized final
    final int j; // Blank final
    final Dummy p; // Blank final reference
    // Blank finals MUST be initialized in the
    constructor:

    BlankFinal() {
        j = 1; // Initialize blank final
        p = new Dummy();
    }

    BlankFinal(int x) {
        j = x; // Initialize blank final
        p = new Dummy();
    }
}
```

21

## Final - Argumentos

- A associação de "final" a argumentos de métodos garante que essas referências não serão alteradas dentro do método.

```
class Gizmo {
    public void spin() {}
}

public class FinalArguments {
    void with(final Gizmo g) {
        //! g = new Gizmo(); // Illegal -- g is final
        g.spin();
    }
    void without(Gizmo g) {
        g = new Gizmo(); // OK -- g not final
        g.spin();
    }
    // void f(final int i) { i++; } // Can't change
}
```

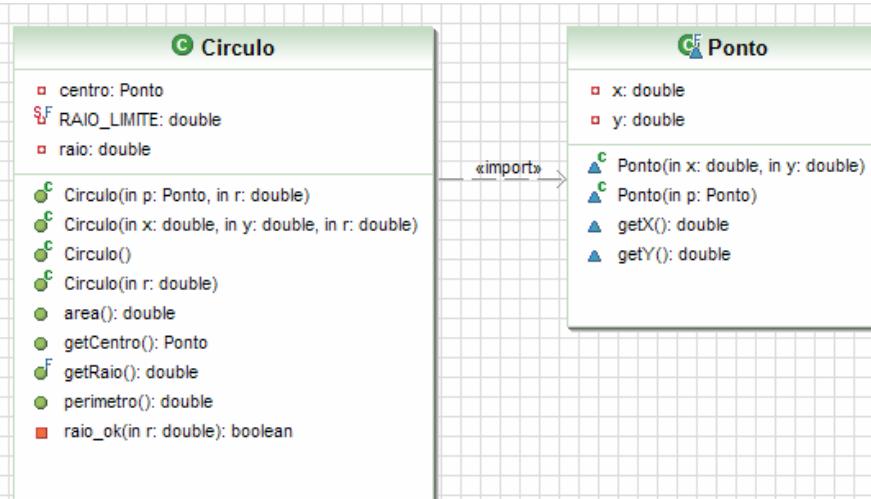
22

## Exemplo

```
public final class Ponto {  
    private double x;  
    private double y;  
  
    public Ponto(double x, double y) { this.x=x; this.y=y; }  
    public final double x() { return(x); }  
    public final double y() { return(y); }  
}  
  
public class Circulo {  
    private Ponto centro;  
    private double raio;  
  
    public static final double RAIO_LIMITE = 100.0;  
    private boolean raio_ok(double r) { return(r<=RAIO_LIMITE); }  
    public Circulo(Ponto p, double r) {  
        centro = p;  
        if (raio_ok(r)) raio = r; else raio = RAIO_LIMITE;  
    }  
    public Circulo(double x, double y, double r) { this(new Ponto(x, y), r); }  
    public double area() { return Math.PI*raio*raio; }  
    public double perimetro() { return 2*Math.PI*raio; }  
    public final double raio() { return raio; }  
    public final Ponto centro() { return centro; }  
}
```

23

## Representação UML



24

## Herança - Boas Práticas

- Programar para a interface e não para a implementação
- Procurar aspectos comuns a várias classes e promovê-los a uma classe base
- Minimizar os relacionamentos entre objetos e organizar as classes relacionadas dentro de um mesmo package
- Usar herança criteriosamente - sempre que possível favorecer a composição

25

## Métodos comuns a todos os objetos

- Todos as classe em Java derivam da super classe `java.lang.Object`
- Métodos
  - `toString()`
  - `equals()`,
  - `hashcode()`
  - `finalize()`
  - `clone()`
  - `getClass()`
  - `wait()`
  - `notify()`
  - `notifyAll()`

} final

26

## toString()

- Circulo c1 = new Circulo(1.5, 0, 0);
- System.out.println( c1 );
  - c1.toString() é invocado automaticamente
- O método `toString()` deve ser sempre redefinido para ter um comportamento de acordo com o objeto

```
public class Circulo {  
    // ....  
    @Override  
    public String toString() {  
        return "Centro : (" + centro.x() + ", " + centro.y() +  
               ") " + " Raio : " + raio;  
    }  
}
```

Centro : (1.5, 0) Raio : 0

27

## equals()

- A expressão `c1 == c2` verifica se as referências `c1` e `c2` apontam para o mesmo objeto
  - Caso `c1` e `c2` sejam variáveis automáticas a expressão anterior compara valores
- Os métodos `equals()` testam se dois objetos são iguais

```
String s1 = "Aveiro";  
String s2 = "Aveiro";  
System.out.println(s1 == s2);           // true (porquê?)  
System.out.println(s1.equals(s2));     // true  
Ponto p1 = new Ponto(1, 1);  
Ponto p2 = new Ponto(1, 1);  
System.out.println(p1 == p2);          // false  
System.out.println(p1.equals(p2));    // false (porquê?)
```
- `equals()` deve ser redefinido sempre que os objetos dessa classe puderem ser comparados
  - `Circulo`, `Ponto`, `Complexo` ...

28

## Problemas com equals()

- Propriedades da igualdade
  - reflexiva:  $x.equals(x) \rightarrow \text{true}$
  - simétrica:  $x.equals(y) \leftrightarrow y.equals(x)$
  - transitiva:  $x.equals(y) \text{ AND } y.equals(z) \rightarrow x.equals(z)$
- Devemos respeitar o contrato 'Object.equals(Object o)' !!!

```
public class Circulo {  
    ...  
    @Override  
    public boolean equals(Object obj) {  
        ...  
    }  
}
```
- Problemas
  - E se 'obj' for null?
  - E se referenciar um objeto diferente de Circulo?

29

## Circulo.equals()

```
@Override  
public boolean equals(Object obj) {  
    if (this == obj)  
        return true;  
    if (obj == null)  
        return false;  
    if (getClass() != obj.getClass())  
        return false;  
    Circulo other = (Circulo) obj;  
    // verify if the object's attributes are equals  
    if (centro == null) {  
        if (other.centro != null)  
            return false;  
    } else if (!centro.equals(other.centro))  
        return false;  
    if (raio != other.raio)  
        return false;  
    return true;  
}
```

30

## equals() em Herança

```
class BaseClass {
    public BaseClass( int i ) {
        x = i;
    }
    public boolean equals( Object rhs ) {
        if ( rhs == null ) return false;
        if ( getClass() != rhs.getClass() ) return false;
        if (rhs == this) return true;
        return x == ( (BaseClass) rhs ).x;
    }
    private int x;
}

class DerivedClass extends BaseClass {
    public DerivedClass( int i, int j ) {
        super( i );
        y = j;
    }
    public boolean equals( Object rhs ) {
        // Não é necessário testar a classe. Feito em base
        return super.equals( rhs ) && y == ( (DerivedClass) rhs ).y;
    }
    private int y;
}
```

31

## hashCode()

- Sempre que o método *equal()* for reescrito, *hashCode* também deve ser
  - Objetos iguais devem retornar códigos de hash iguais
- O objectivo do hash é ajudar a identificar qualquer objeto através de um número inteiro
  - Usado em HashTables

```
// Circulo.hashCode() - Exemplo muito simples !!!
public int hashCode() {
    return raio * centro.x() * centro.y();
}
//...
Circulo c1 = new Circulo(10,15,27);      4050
Circulo c2 = new Circulo(10,15,27);      4050
Circulo c3 = new Circulo(10,15,28);      4200
```

- A construção de uma boa função de hash não é trivial. Para a sua construção recomendam-se outras fontes

32

## Circulo.hashCode()

```
@Override  
public int hashCode() {  
    final int prime = 31;  
    int result = prime + ((centro == null) ? 0 : centro.hashCode());  
    long temp = Double.doubleToLongBits(raio);  
    result = prime * result + (int) (temp ^ (temp >>> 32));  
    // ^ Bitwise exclusive OR  
    // >>> Unsigned right shift  
    return result;  
}
```

33

## Sumário - Porquê herança?

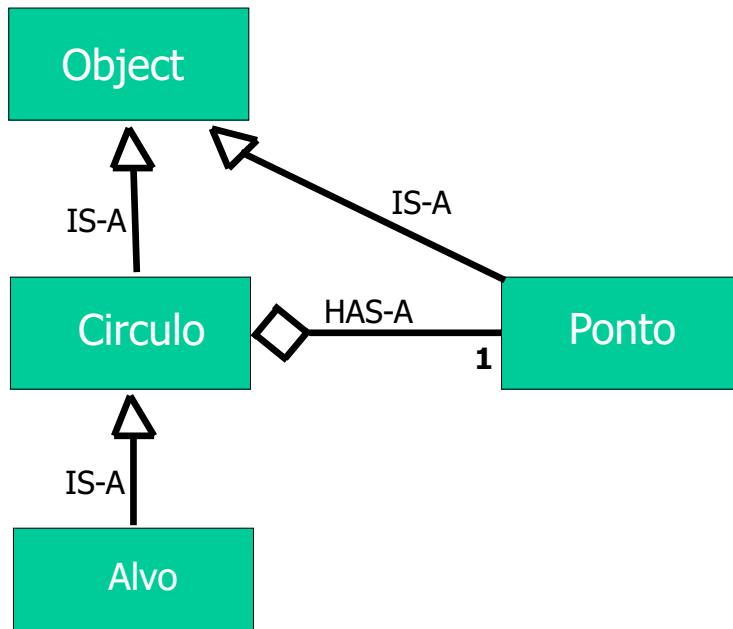
- Muitos objetos reais apresentam esta característica
- Permite criar classes mais simples com funcionalidades mais estanques e melhor definidas
  - Devemos evitar classes com interfaces muito "extensas"
- Permite reutilizar e estender interfaces e código
- Permite tirar partido do polimorfismo

34

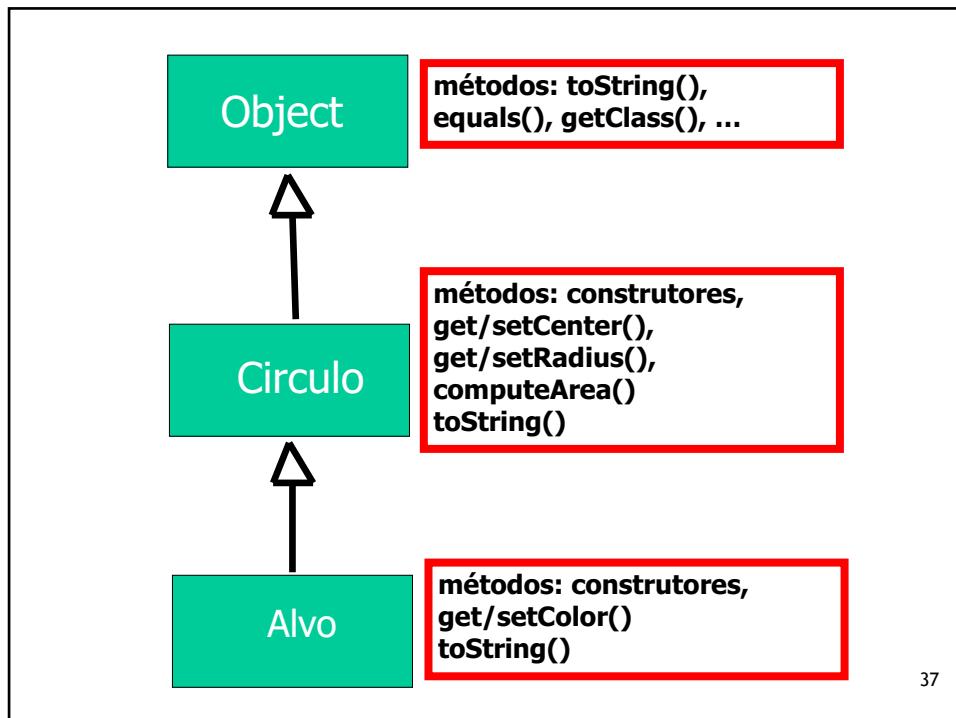
# Java Polimorfismo

UA, DETI, Programação III  
José Luis Oliveira, Carlos Costa  
2016/17

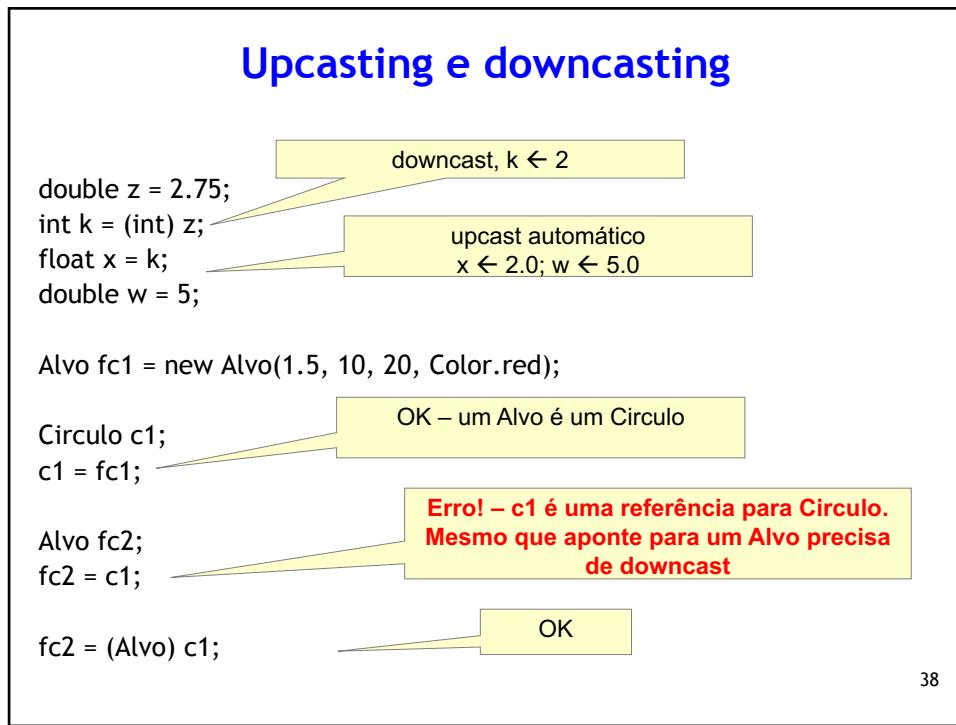
35



36



37



38

## Upcasting e downcasting

```
Circulo c2 = new Circulo(1.5f, 10, 20);
```

```
fc2 = (Alvo) c2;
```

run-time error:  
ClassCast exception

- O tipo do objeto pode ser testado com o operador instanceof

```
if (c3 instanceof Alvo)  
    fc2 = (Alvo) c3;
```

OK

39

## Polimorfismo

- Ideia base:

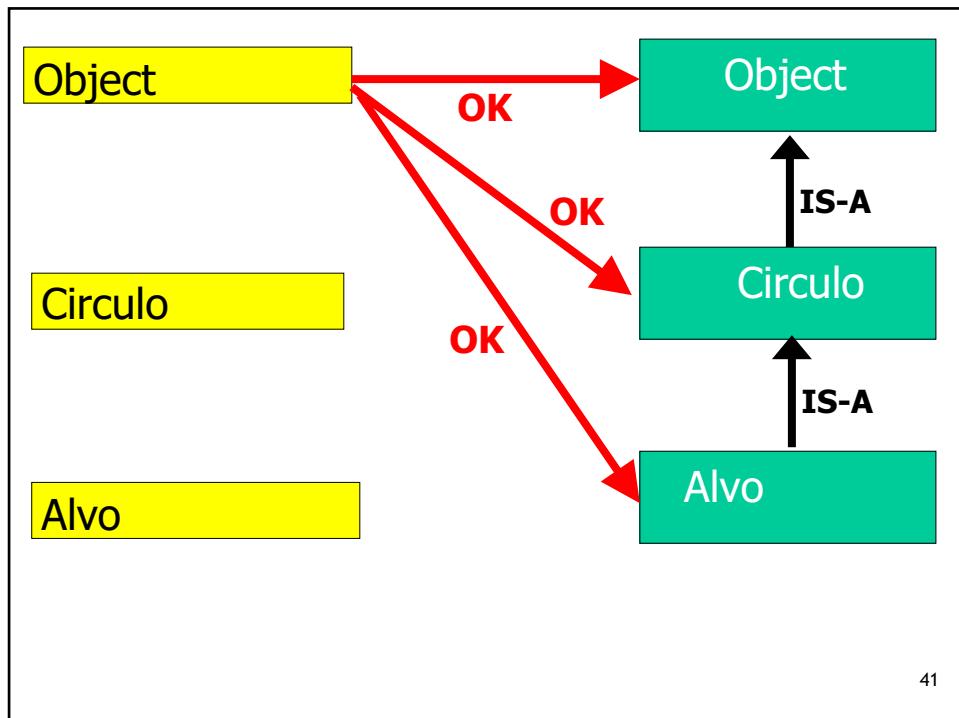
- o tipo declarado na referência não precisa de ser exatamente o mesmo tipo do objeto para o qual aponta - pode ser de qualquer tipo derivado

```
Circulo c1 = new Alvo(...);  
Object obj = new Circulo(...);
```

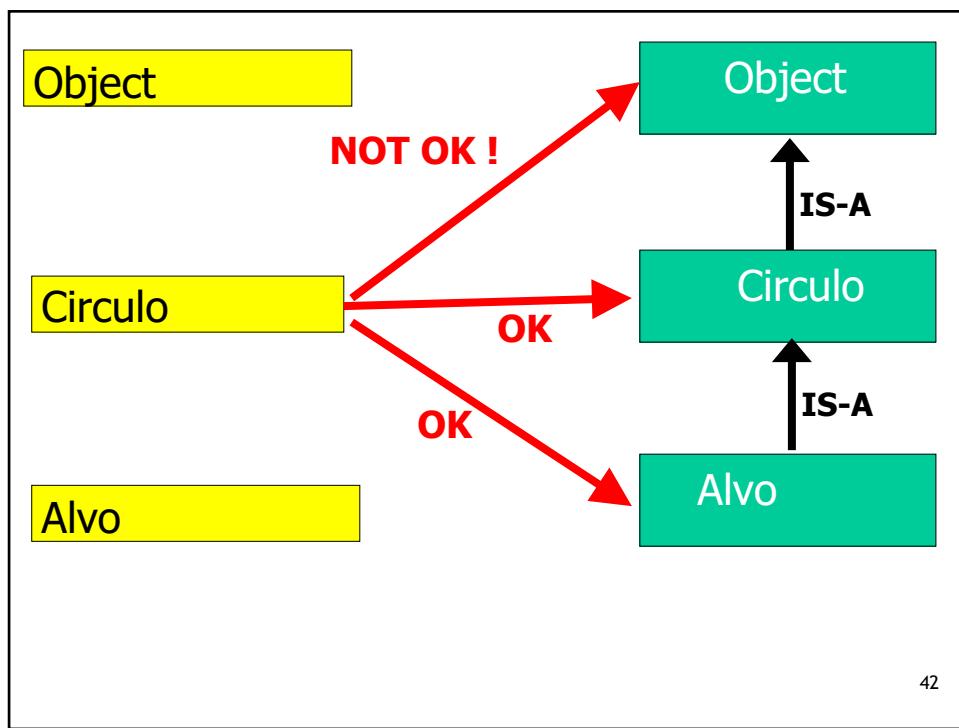
- Referência polimórfica

- T ref1 = new S();
- // OK desde que todo o S seja um T

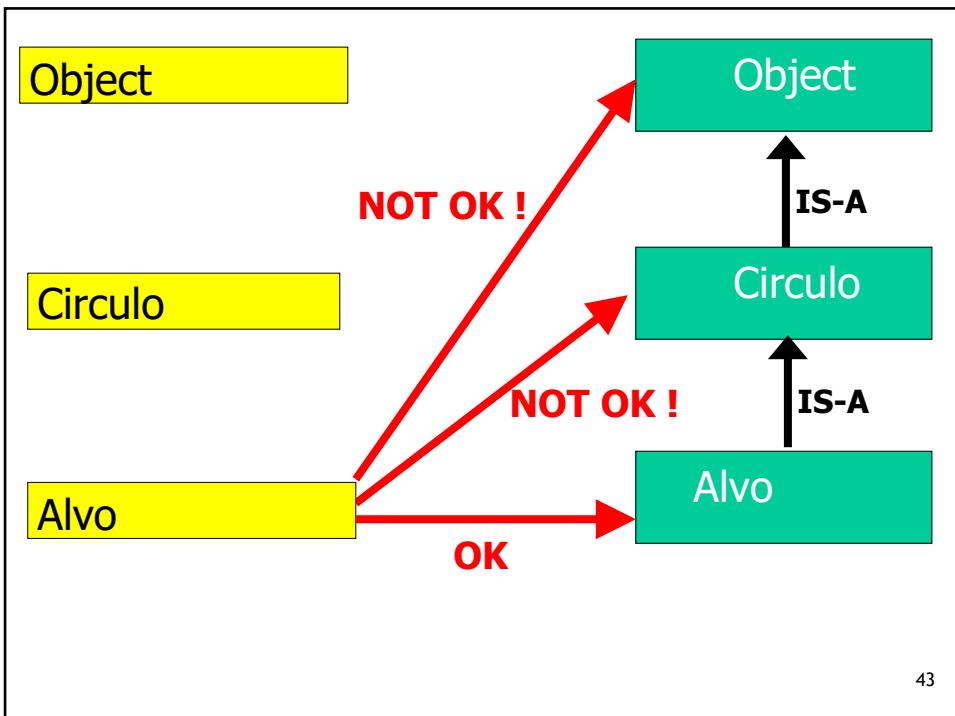
40



41



42



43

## Polimorfismo

- Polimorfismo é, conjuntamente com a Herança e o Encapsulamento, uma das características fundamentais da POO.
  - Formas diferentes com interfaces semelhantes.
- Outras designações:
  - Ligação dinâmica (Dynamic binding), late binding ou run-time binding
- Esta característica permite-nos tirar mais partido da herança.
  - Podemos, por exemplo, desenvolver um método X() com parâmetro CBase com a garantia que aceita qualquer argumento derivado de CBase.
  - O método X() só é resolvido em execução.
- Todos os métodos (à excepção dos final) são late binding.
  - O atributo final associado a uma função, impede que ela seja redefinida e simultaneamente dá uma indicação ao compilador para ligação estática (early binding) - que é o único modo de ligação em linguagens com o C.

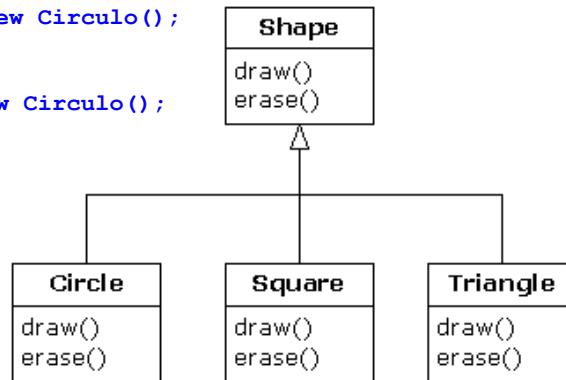
44

## Exemplo 1

```
Shape s = new Shape();
s.draw();

Circulo c = new Circulo();
c.draw();

Shape s2 = new Circulo();
s2.draw();
```



45

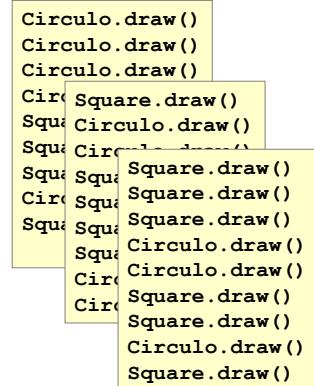
## Exemplo 2

```
class Shape { void draw() {} }

class Circle extends Shape {
    void draw() { System.out.println("Circle.draw()"); }
}

class Square extends Shape {
    void draw() { System.out.println("Square.draw()"); }
}

public class Shapes {
    public static Shape randShape() {
        switch((int)(Math.random() * 2)) {
            default:
            case 0: return new Circle();
            case 1: return new Square();
        }
    }
    public static void main(String[] args) {
        Shape[] s = new Shape[9];
        for(int i = 0; i < s.length; i++)
            s[i] = randShape(); // Fill up the array with shapes;
        for(int i = 0; i < s.length; i++)
            s[i].draw(); // Make polymorphic method calls:
    }
}
```



46

## Generalização

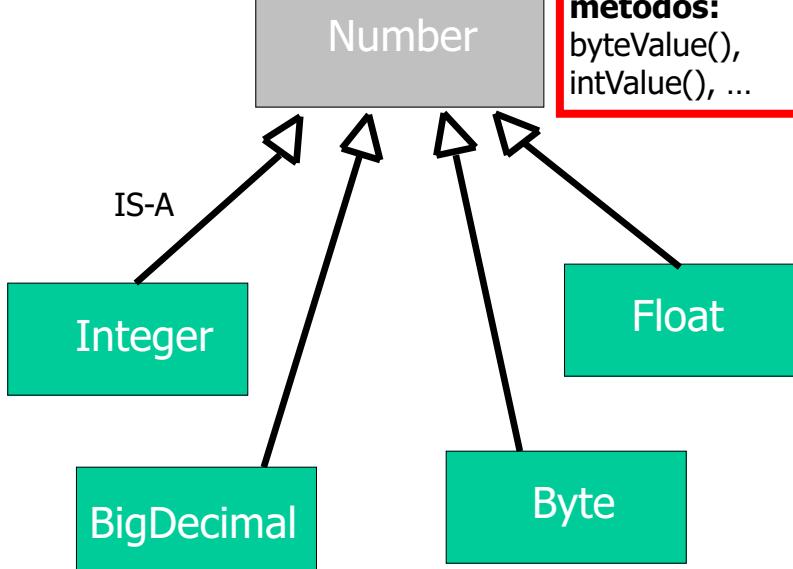
- A generalização consiste em melhorar as classes de um problema de modo a torná-las mais gerais.

Formas de generalização:

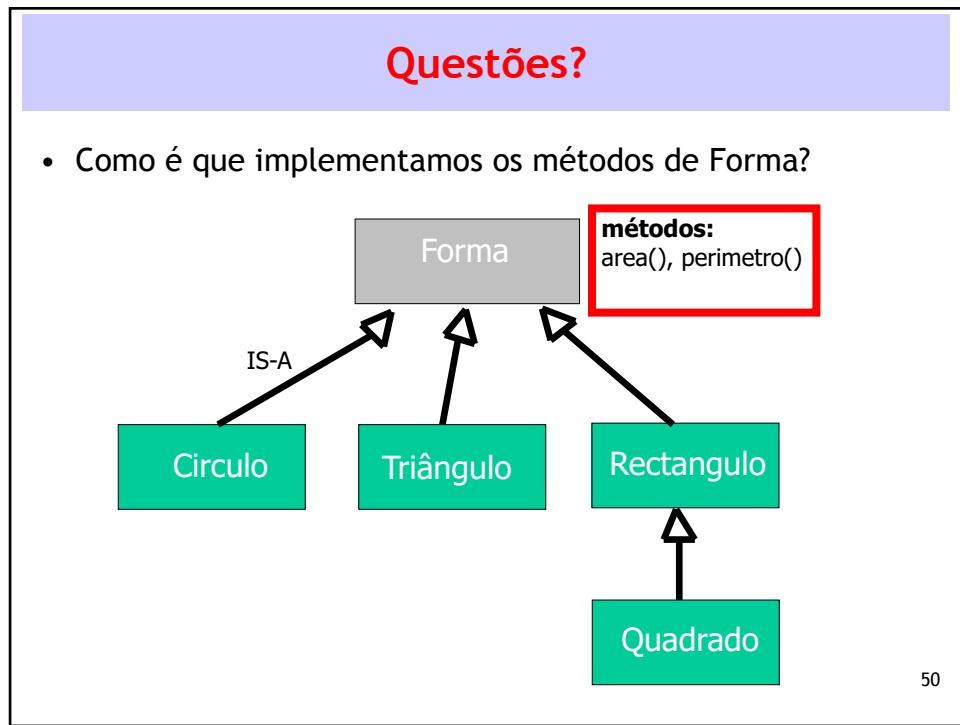
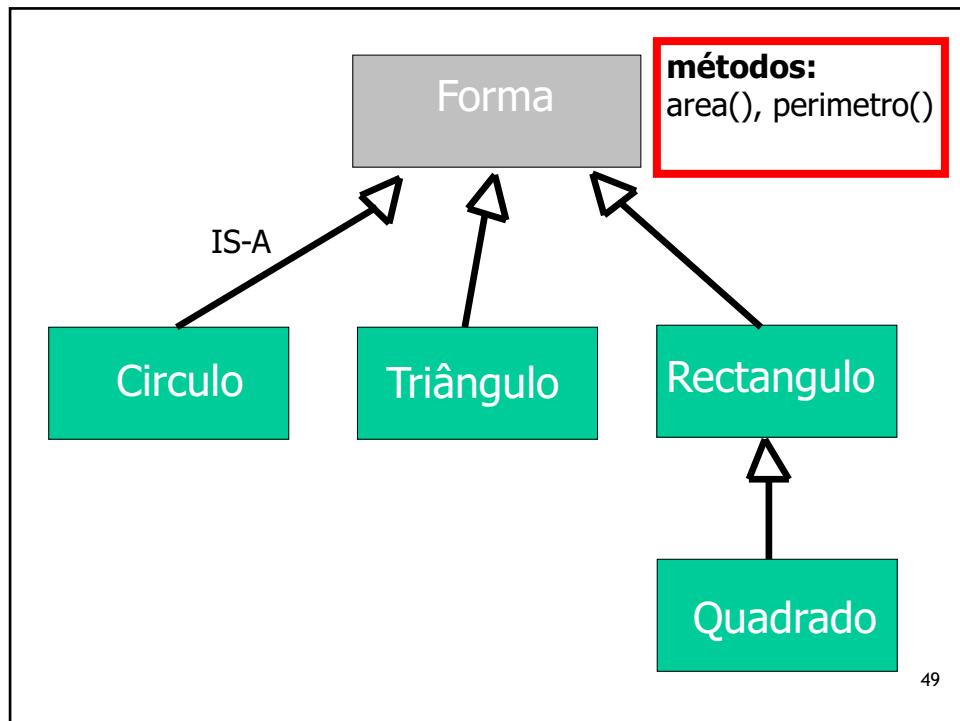
- Tornar a classe o mais abrangente possível de forma a cobrir o maior leque de entidades.  
`class ZooAnimal;`
- Abstrair implementações diferentes para operações semelhantes em classes abstractas num nível superior.  
`ZooAnimal.draw();`
- Reunir comportamentos e características e fazê-los subir o mais possível na hierarquia de classes.  
`ZooAnimal.peso;`

47

**métodos:**  
byteValue(),  
intValue(), ...



48



## Classes abstractas

- Uma classe é abstracta se contiver pelo menos um método abstracto.

- Um método abstracto é um método cujo corpo não é definido.

```
public abstract class Forma
{
    // pode definir constantes
    public static final double DOUBLE_PI = 2*Math.PI;

    // pode declarar métodos abstractos
    public abstract double area();
    public abstract double perimetro();

    // pode incluir métodos não abstractos
    public String aka() { return "euclidean"; }
}
```

- Uma classe abstracta não é instanciável.

```
Forma f;           // OK. Podemos criar uma referência para Forma
f = new Forma();   // Erro! Não podemos criar Formas
```

51

## Classes abstractas

- Num processo de herança a classe só deixa de ser abstracta quando implementar todos os métodos abstractos.

```
public class Circulo extends Forma {

    protected double r;

    public double area() {
        return Math.PI*r*r;
    }

    public double perimetro() {
        return DOUBLE_PI*r;
    }
}

Forma f;
f = new Circulo(); // OK! Podemos criar Circulos
```

52

## Classes abstractas e Polimorfismo

```
abstract class Figura {
    abstract void doWork();
    protected int cNum;
}

class Circulo extends Figura {
    Circulo(int i) { cNum = i; }
    void doWork() { System.out.println("Circulo"); }
}

class Alvo extends Circulo {
    Alvo(int i) { super(i); }
    void doWork() { System.out.println("Alvo"); }
}

class Quadrado extends Figura {
    void doWork() { System.out.println("Quadrado"); }
}

public class ArrayOfObjects {
    public static void main(String[] args) {
        Figura[] anArray = new Figura[10];
        for (int i = 0; i < anArray.length; i++) {
            switch ((int) (Math.random() * 3)) {
                case 0 : anArray[i] = new Circulo(i); break;
                case 1 : anArray[i] = new Alvo(i); break;
                case 2 : anArray[i] = new Quadrado(); break;
            }
        }
        // invoca o método doWork sobre todas as Figura da tabela
        // -- Polimorfismo
        for (int i = 0; i < anArray.length; i++) {
            System.out.print("Figura("+i+") --> ");
            anArray[i].doWork();
        }
    }
}
```

```
Figura(0) --> Quadrado
Figura(1) --> Circulo
Figura(2) --> Quadrado
Figura(3) --> Circulo
Figura(4) --> Quadrado
Figura(5) --> Alvo
Figura(6) --> Circulo
Figura(7) --> Quadrado
Figura(8) --> Circulo
Figura(9) --> Quadrado
```

53

# Java Excepções

Programação III  
José Luis Oliveira; Carlos Costa

# Problema!

- Nem todos os erros são detectados na compilação.
- Estes são, geralmente, os que são mais menosprezados pelos programadores:
  - Compila sem erros → Funciona!!!
- Tratamento Clássico:
  - Se sabemos à partida que pode surgir uma situação de erro em determinada passagem podemos tratá-la nesse contexto (if...)

```
if ((i = doTheJob()) != -1) {  
    /* tratamento de erro */  
}
```

# Excepção

- Uma excepção é gerada por algo imprevisto que não é possível controlar.
- Utilização de Excepções:

- Tratamento do erro no contexto local

```
try {  
    /* O que se pretende fazer */  
}  
catch (Errortype a) {  
}
```

- Delegação do erro - gerar um objecto excepção (throw) no qual se delega esse tratamento.

```
if (t == null)  
    throw new NullPointerException();  
// throw new NullPointerException("t null");
```

# Controlo de Excepção

- A manipulação de excepções é feita através de um bloco especial - try.

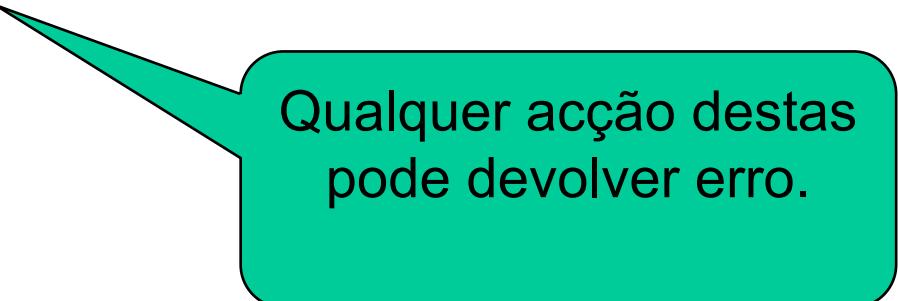
```
try {  
    // Code that might generate exceptions Type1,  
    // Type2 or Type3  
} catch(Type1 id1) {  
    // Handle exceptions of Type1  
} catch(Type2 id2) {  
    // Handle exceptions of Type2  
} catch(Type3 id3) {  
    // Handle exceptions of Type3  
} finally {  
    // Executada independentemente de haver ou não  
    // uma excepção  
}
```

# Vantagens das Excepções

- Separação clara entre o código regular e o código de tratamento de erros
- Propagação dos erros em chamadas sucessivas
- Agrupamento de erros por tipos

# Separação de código - exemplo (1)

```
readFile {  
    open the file;  
    determine its size;  
    allocate that much memory;  
    read the file into memory;  
    close the file;  
}
```



Qualquer acção destas  
pode devolver erro.

# Separação de código - exemplo (2)

```
errorCodeType readFile {
    initialize errorCode = 0;
    open the file;
    if (theFileIsOpen) {
        determine the length of the file;
        if (gotTheFileLength) {
            allocate that much memory;
            if (gotEnoughMemory) {
                read the file into memory;
                if (readFailed) { errorCode = -1; }
            } else { errorCode = -2; }
        } else { errorCode = -3; }
        close the file;
        if (theFileDidntClose && errorCode == 0) {
            errorCode = -4;
        } else { errorCode = errorCode and -4; }
    } else { errorCode = -5; }
    return errorCode;
}
```

Sem Excepções

# Separação de código - exemplo (3)

```
readFile {
    try {
        open the file;
        determine its size;
        allocate that much memory;
        read the file into memory;
        close the file;
    } catch (fileOpenFailed) {
        doSomething;
    } catch (sizeDeterminationFailed) {
        doSomething;
    } catch (memoryAllocationFailed) {
        doSomething;
    } catch (readFailed) {
        doSomething;
    } catch (fileCloseFailed) {
        doSomething;
    }
}
```

Com Excepções

# Propagação dos erros (1)

```
method1 {  
    call method2;  
}  
method2 {  
    call method3;  
}  
method3 {  
    call readFile;  
}
```

Solução sem  
Excepções

```
method1 {  
    errorCodeType error;  
    error = call method2;  
    if (error)  
        doErrorProcessing;  
    else  
        proceed;  
}  
  
errorCodeType method2 {  
    errorCodeType error;  
    error = call method3;  
    if (error)  
        return error;  
    else  
        proceed;  
}  
  
errorCodeType method3 {  
    errorCodeType error;  
    error = call readFile;  
    if (error)  
        return error;  
    else  
        proceed;  
}
```

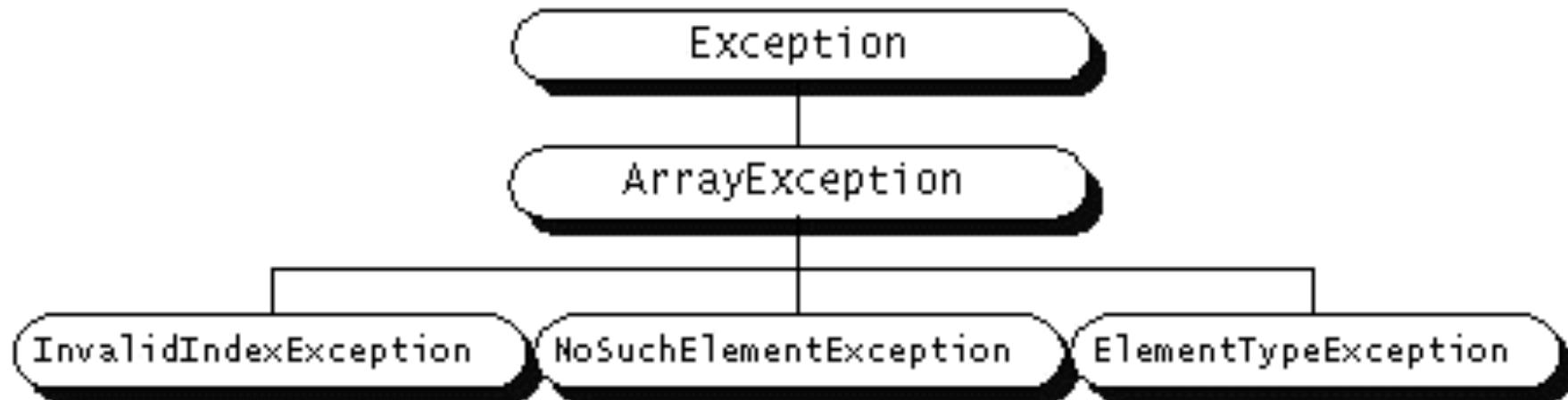
# Propagação dos erros (2)

```
method1 {  
    call method2;  
}  
method2 {  
    call method3;  
}  
method3 {  
    call readFile;  
}
```

Solução com  
Excepções

```
method1 {  
    try {  
        call method2;  
    } catch (exception) {  
        doErrorProcessing;  
    }  
}  
  
method2 throws exception {  
    call method3;  
}  
  
method3 throws exception {  
    call readFile;  
}
```

# Agrupamento de erros por tipos



```
catch (InvalidIndexException e) {
```

```
    . . .
```

```
}
```

Diferenciação

```
catch (ArrayException e) {
```

```
    . . .
```

```
}
```

Agrupamento

# Excepções - Hierarquia de Classes

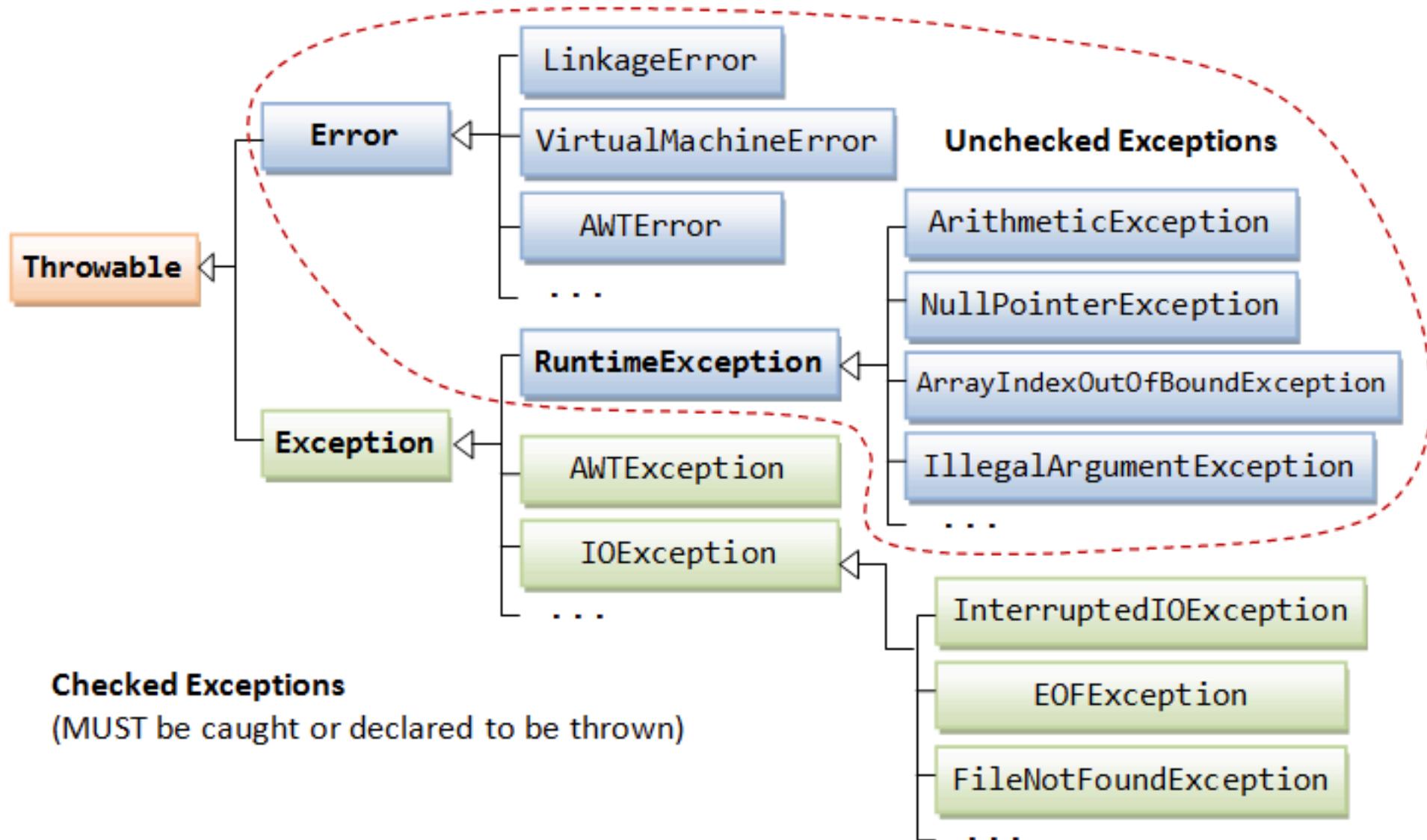
```
try {  
    ...  
}  
catch (NullPointerException e){  
    ...  
}  
catch (IndexOutOfBoundsException e){  
    ...  
}  
catch (ArithmetricException e){  
    ...  
}  
catch (Exception e){  
    ...  
}
```

A ordem dos "catch"  
é importante

```
try {  
    ...  
}  
catch (Exception e){  
    ...  
}  
catch (NullPointerException e){  
    ...  
}  
catch (IndexOutOfBoundsException e){  
    ...  
}  
catch (ArithmetricException e){  
    ...  
}
```

Má Solução

# Excepções - Hierarquia de Classes



# Classe Exception

- A classe Exception é derivada da classe Throwable
- Podemos usar a classe base java.lang.Exception para capturar qualquer exceção

```
catch(Exception e) {  
    System.out.println("caught an exception");  
}
```

- Podemos regenerar nova exceção de forma a ser tratada num nível superior

```
catch(Exception e) {  
    System.out.println("Exception was thrown");  
    throw e;  
}
```

# Especificação da Exceção

- Quando desenhamos métodos que possam gerar exceções devemos assinalá-las explicitamente

```
void f() throws TooBigException,  
          TooSmallException,  
          DivByZeroException {  
  
    // ...  
}
```

# Criar Novas Excepções

- Podemos usar o mecanismo de herança para personalizar algumas exceções

```
class MyException extends Exception {  
    // interface base  
    public MyException() {}  
    public MyException(String msg) {  
        super(msg);  
    }  
    // podemos acrescentar construtores e dados  
}
```

# Exemplo

```
public class Rethrowing {  
    public static void f() throws Exception {  
        System.out.println("exception in f()");  
        throw new Exception("thrown from f()");  
    }  
    public static void g() throws Exception {  
        try { f();  
        } catch(Exception e) {  
            System.out.println(" exception in g()");  
            throw e;  
        }  
    }  
    public static void main(String[] args) {  
        try { g();  
        } catch(Exception e) {  
            System.out.println("Caught in main");  
        }  
    }  
}
```

# Tipos de Excepções

- checked
  - Se invocarmos um método que gere uma *checked exception*, temos de indicar ao compilador como vamos resolvê-la:
    - 1) Resolver *try .. catch* ou
    - 2) Propagar *throw*
- unchecked
  - São erros de programação ou do sistema (podemos usar Aserções nestes casos)
  - são subclasses de `java.lang.RuntimeException` ou `java.lang.Error`

# Boas Práticas

- Usar excepções apenas para condições excepcionais
  - Uma API bem desenhada não deve forçar o cliente a usar excepções para controlo de fluxo
  - Uma excepção não deve ser usada para um simples teste

**X** `try {  
 s.pop();  
} catch(EmptyStackException es) {...}`

**V** `if (!s.empty()) s.pop(); // melhor!`

# Boas Práticas

- Usar preferencialmente excepções standards
  - `IllegalArgumentException`
  - - valor de parâmetros inapropriado
  - `IllegalStateException`
  - - Estado de objecto incorrecto
  - `NullPointerException`
  - `IndexOutOfBoundsException`
- Tratar sempre as excepções (ou delegá-las)

 `try {  
 // .. código que pode causar excepções  
} catch (Exception e) {}`

# Java Interfaces

UA, DETI, Programação III  
José Luis Oliveira, Carlos Costa  
2016/17

54

## Interfaces

- Uma interface é uma classe abstracta pura (só contém assinaturas\*).

```
public interface Desenhavel {  
    //...  
}
```
- Actua como um protocolo perante as classes que as implementam.

```
public class Grafico implements Desenhavel {  
    // ...  
}
```
- Uma classe pode herdar de uma só classe base e implementar uma ou mais interfaces.

\* Java 8: default and static methods

55

## Interfaces - Exemplo

```
interface Desenhavel {  
    public void cor(Color c);  
    public void corDeFundo(Color cf);  
    public void posicao(double x, double y);  
    public void desenha(DrawWindow dw);  
}  
  
class CirculoGrafico extends Circulo implements Desenhavel {  
    public void cor(Color c) {...}  
    public void corDeFundo(Color cf) {...}  
    public void posicao(double x, double y) {...}  
    public void desenha(DrawWindow dw) {...}  
}
```

56

## Características principais

- Todos os seus métodos são, implicitamente, abstractos.
  - Os únicos modificadores permitidos são `public` e `abstract`.
- Uma interface pode herdar (`extends`) mais do que uma interface.
- Não são permitidos construtores.
- As variáveis são implicitamente estáticas e constantes
  - `static final ..`
- Uma classe (não abstracta) que implemente uma interface deve implementar todos os seus métodos.
- Uma interface pode ser vazia
  - `Cloneable`, `Serializable`
- Não se pode criar uma instância da interface
- Pode criar-se uma referência para uma interface

57

## Interfaces em Java 8

- **Default Methods**

- Oferecem um implementação por defeito
- Podem ser reescritos nas classes que implementam a interface

```
public interface Interface1 {  
    default void defMeth(){//... do something }  
}  
public class MyClass implements Interface1 {  
    @Override  
    public void defMeth() { // ... do something }  
}
```

- **Static Methods**

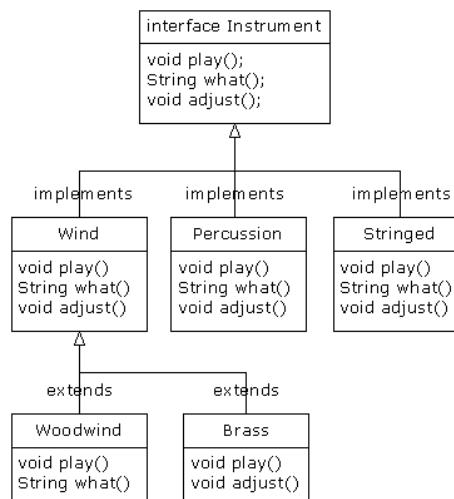
- Similares aos default methods
- Não podem ser reescritos nas classes que implementam a interface

```
public interface Interface2 {  
    static void stMeth(){//... do something }  
}  
public class MyClass implements Interface2 {  
    @Override  
    public void stMeth(){ // ... do something }  
}
```

58

## Interfaces - Exemplos

- Depois de implementada uma interface passam a actuar as regras sobre classes



59

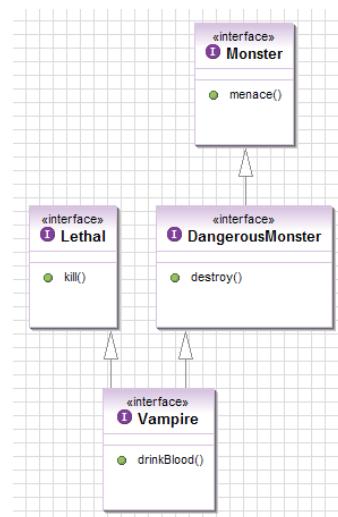
## Interfaces - Exemplos

```
interface Instrument {  
    // Compile-time constant:  
    int i = 5; // static & final  
    // Cannot have method definitions:  
    void play(); // Automatically public  
    String what();  
    void adjust();  
}  
  
class Wind implements Instrument {  
    public void play() {  
        System.out.println("Wind.play()");  
    }  
    public String what() { return "Wind"; }  
    public void adjust() {}  
}
```

60

## Herança em Interfaces

```
interface Monster {  
    void menace();  
}  
  
interface DangerousMonster  
    extends Monster {  
    void destroy();  
}  
  
interface Lethal {  
    void kill();  
}  
  
interface Vampire  
    extends DangerousMonster,  
            Lethal {  
    void drinkBlood();  
}
```



61

## Classes Abstractas versus Interfaces

### Classes Abstractas

- pode não ser 100% abstracta
- escrever software genérico, parametrizável e extensível
- relacionamento na hierarquia simples de classes

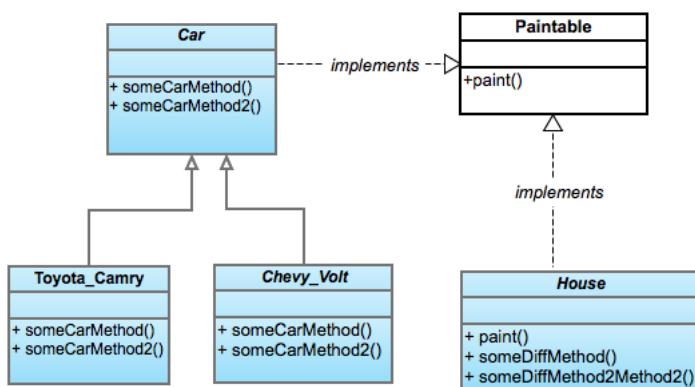
### Interfaces

- 100% abstractas
  - Java 8: default and static methods
- especificar um conjunto adicional de comportamentos / propriedades funcionais
- implementação horizontal na hierarquia

Não há regras ou metodologia: "... neste caso usa-se interfaces, no outro ..."

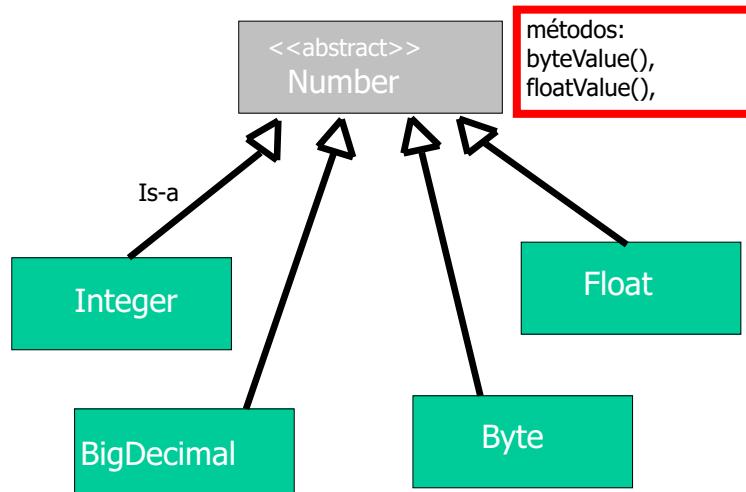
62

## Classes Abstractas versus Interfaces



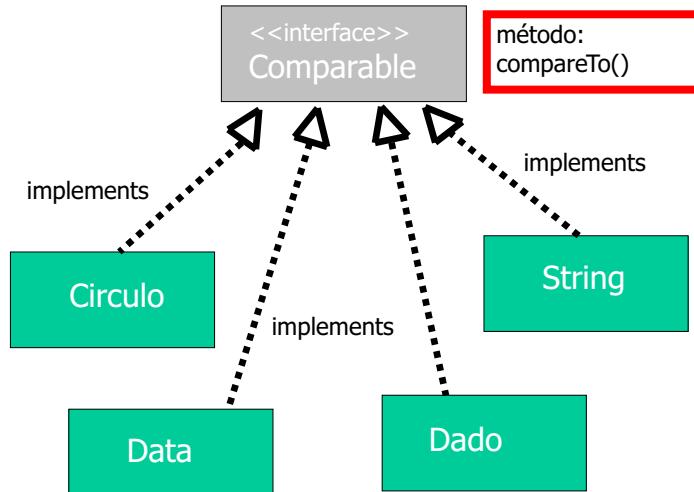
63

## Classes Abstractas versus Interfaces



64

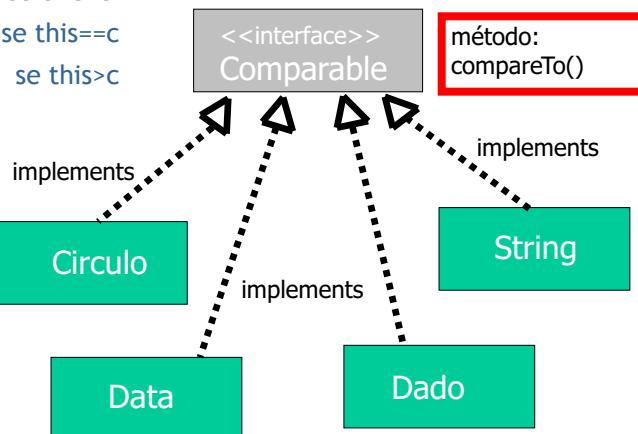
## Classes Abstractas versus Interfaces



65

## Questões?

- Qual o interesse de usar uma interface neste caso?
- Note que o método `int compareTo(Object c)` retorna:
  - <0 se `this < c`
  - 0 se `this == c`
  - >0 se `this > c`



66

## Interface Comparable

```
public interface Comparable<T> { // package java.lang;
    int compareTo(T other);
}

public abstract class Shape implements Comparable<Shape> {
    public abstract double area();
    public abstract double perimeter();

    public int compareTo( Shape irhs )  {
        if(irhs==null)
            throw new NullPointerException("....");

        return area() - rhs.area();
    }
}
```

67

```

class FindMaxDemo {

    public static <T> Comparable<T> findMax(Comparable<T>[] a)  {
        int maxIndex = 0;

        for ( int i = 1; i < a.length; i++ )
            if ( a[i] != null && a[i].compareTo((T) a[maxIndex]) > 0)
                maxIndex = i;

        return a[maxIndex];
    }

    public static void main( String[] args )  {
        Shape[] sh1 = { new Circle( 2.0 ),
                        new Square( 3.0 ),
                        new Rectangle( 3.0, 4.0 ) };

        String[] st1 = { "Joe", "Bob", "Bill", "Zeke" };

        System.out.println( findMax( sh1 ) );
        System.out.println( findMax( st1 ) );
    }
}

```

68

## instanceof

- Instrução que indica se uma referência é membro de uma classe ou interface
- Exemplo, considerando

```

class Dog extends Animal implements Pet {...}
Animal fido = new Dog();

```

- as instruções seguintes são true:

```

if (fido instanceof Dog) ...
if (fido instanceof Animal) ...
if (fido instanceof Pet) ...

```

69

## Copiar objetos (clone)

- `protected Object clone()`
  - Retorna um novo objeto cujo estado inicial é uma cópia do objeto sobre o qual o método foi invocado.
  - As alterações subsequentes na réplica não afetarão o original.
  - Este método realiza uma cópia simples de todos os campos. Nem sempre é adequado.
- Construtor de cópia
  - Construtor cujo argumento é um objeto da mesma classe

```
public Figura(Figura original) {  
    ...  
}
```

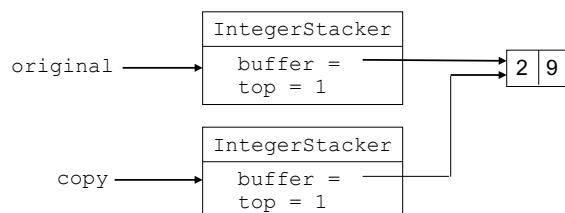
Não é comum em Java  
É preferível usar o método  
clone()

70

## Shallow cloning

- Cópia campo a campo.
  - This might be wrong if it duplicates a reference to an object that shouldn't be shared.

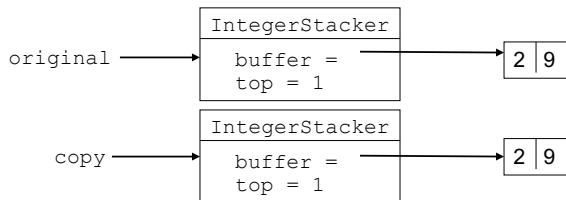
```
public class IntegerStack {  
    private int[] buffer; // a stack of integers  
    private int top;      // largest index in the stacker  
                          // (starting from 0)  
    ...  
}
```



71

## Deep cloning

- Cria uma réplica de todos os objectos que podem ser alcançados a partir do objeto que estamos a replicar



72

## Interface java.lang.Cloneable

- Se quisermos fazer uso de Object.clone() temos de implementar a interface Cloneable
  - não tem métodos nem constantes (vazia) e funciona como um marcador

```
public class Rectangle implements Cloneable{  
    ...  
}
```
  - Shallow copy

```
@Override protected Rectangle clone() throws  
CloneNotSupportedException {  
    return (Rectangle) super.clone();  
}
```
  - Deep copy - temos de ser nós a garantir a implementação local de clone()

```
@Override protected Rectangle clone() throws  
CloneNotSupportedException {  
    return new Rectangle(...);  
}
```

73

# Java Classes internas

UA, DETI, Programação III  
José Luis Oliveira, Carlos Costa  
2016/17

74

## Classes internas

- Classes podem ser membros de classes, de objetos ou locais a métodos. Podem até serem criadas sem nome, apenas com corpo no momento em que instanciam um objeto
  - Há poucas situações onde classes internas podem ou devem ser usadas. Devido à complexidade do código que as utiliza, deve evitar-se usos não convencionais
  - Usos típicos incluem tratamento de eventos em GUIs, criação de threads, manipulação de coleções e sockets
- Classes internas podem ser classificadas em quatro tipos
  - Classes estáticas - classes membros de classe (*nested classes*)
  - Classes de instância - classes membros de objetos
  - Classes locais - classes dentro de métodos
  - Classes anónimas - classes dentro de instruções

75

## Classes estáticas

- São declaradas como *static* dentro de uma classe
- A classe externa age como um pacote para uma ou mais classes internas estáticas
  - *Externa.Coisa, Externa.InternaDois, ..*
- O compilador gera arquivos tipo *Externa\$InternaUm.class*

```
class Externa {  
    private static class InternaUm {  
        public int campo;  
        public void metodoInterno() {...}  
    }  
    public static class InternaDois  
        extends InternaUm {  
        public int campo2;  
        public void metodoInterno() {...}  
    }  
    public static interface Coisa {  
        void existe();  
    }  
    public void metodoExterno() {...}  
}
```

76

## Classes de instância

- São membros do objeto, como métodos e atributos
- Requerem que objeto exista antes que possam ser usadas.
  - Externamente usa-se *referência.new* para criar objetos
- Deve usar-se *NomeDaClasse.this* para aceder a campos internos

```
class Externa {  
    public int campoUm;  
    public class Interna {  
        public int campoUm;  
        public int campoDois;  
        public void metodoInterno() {  
            this.campoUm = 10; // Externa.campoUm  
            Interna.this.campoUm = 15;  
        }  
    }  
    public static void main(String[] args){  
        Interna e = (new Externa()).new Interna();  
    }  
}
```

77

## Classes locais

- Servem para tarefas temporárias já que deixam de existir quando o método acaba
  - Têm o mesmo alcance de variáveis locais.

```
public Multiplicavel calcular(final int a, final int b) {  
    class Interna implements Multiplicavel {  
        public int produto() {  
            return a * b; // usa a e b, que são constantes  
        }  
        return new Interna();  
    }  
    public static void main(String[] args){  
        Multiplicavel mul = (new Externa()).calcular(3,4);  
        int prod = mul.produto();  
    }  
}
```

78

## Classes anónimas

- Servem para criar um único objeto
  - A classe abaixo estende ou implementa SuperClasse, que pode ser uma interface ou classe abstracta (o new, neste caso, indica a criação da classe entre chaves, não da SuperClasse)  
`Object i = new SuperClasse() { implementação };`
  - O compilador gera arquivo Externa\$1.class, Externa\$2.class,

```
public Multiplicavel calcular(final int a, final int b) {  
    return new Multiplicavel() {  
        public int produto() {  
            return a * b;  
        }  
    };  
}  
public static void main(String[] args){  
    Multiplicavel mul = (new Externa()).calcular(3,4);  
    int prod = mul.produto();  
}
```

A classe está dentro da instrução:  
preste atenção no ponto-e-vírgula!

Compare com parte em  
preto e vermelho do  
slide anterior!

79

## Classes internas

- São sempre classes dentro de classes. Exemplo:

```
class Externa {  
    private class Interna {  
        public int campo;  
        public void metodoInterno() {...}  
    }  
    public void metodoExterno() {...}  
}
```

- Podem ser *private*, *protected*, *public* ou *package-private*
  - Excepto as que aparecem dentro de métodos, que são locais
- Podem ser estáticas:
  - E chamadas usando a notação `Externa.Interna`
- Podem ser de instância e depender da existência de objetos:  
`Externa e = new Externa();  
Externa.Interna ei = e.new Externa.Interna();`
- Podem ser locais (dentro de métodos)
  - E nas suas instruções podem não ter nome (anónimas)

80

## Sumário

- Polimorfismo
- Generalização
- Classes abstractas
- Interfaces
- Classes internas

81

# Java Tipos Enumerados

UA, DETI, Programação III  
José Luis Oliveira, Carlos Costa  
2017/18

1

## Enumerados

- Um Enumerado é uma forma simples de associar constantes a um conjunto de valores.
- Antes do JAVA 5:

```
public static final int SPRING = 0;  
public static final int SUMMER = 1;  
public static final int FALL = 2;  
public static final int WINTER = 3;
```

Este formato tem um problema!!!

Qual?

Como se resolve?

2

1

## Exemplo: Solução 1

```
class Note {  
    private int value;  
    private Note(int val) { value = val; }  
    public static final Note  
        MIDDLE_C = new Note(0),  
        C_SHARP = new Note(1),  
        C_FLAT = new Note(2);  
}  
class Instrument {  
    public void play(Note n) {  
        System.out.println("Instrument.play()");  
    }  
}  
class Wind extends Instrument {  
    public void play(Note n) {  
        System.out.println("Wind.play()");  
    }  
}  
public class Music {  
    public static void tune(Instrument i) { // ...  
        i.play(Note.MIDDLE_C);  
    }  
    public static void main(String[] args) {  
        Wind flute = new Wind();  
        tune(flute); // Upcasting  
    }  
}
```

```
class Note {  
    public static final int  
        MIDDLE_C = 0,  
        C_SHARP = 1,  
        C_FLAT = 2;  
}
```

--> Play(int n)  
Não cria qq.  
ligação a Note

3

## Exemplo: Solução 2, usando enum

```
enum Note { MIDDLE_C, C_SHARP, C_FLAT }  
  
class Instrument {  
    public void play(Note n) {  
        System.out.println("Instrument.play()");  
    }  
}  
class Wind extends Instrument {  
    public void play(Note n) {  
        System.out.println("Wind.play()");  
    }  
}  
public class Music {  
    public static void tune(Instrument i) {  
        i.play(Note.MIDDLE_C);  
    }  
    public static void main(String[] args) {  
        Wind flute = new Wind();  
        tune(flute); // Upcasting  
    }  
}
```

```
enum Note {  
    MIDDLE_C(0),  
    C_SHARP(1),  
    C_FLAT(2);  
    private int value;  
    private Note(int val) {  
        value = val;  
    }  
}
```

4

## Tipos Enumerados

- Mais Valia Importante: “compile-time type safety”
- Forma mais Simples
  - Forma de referenciar  
`Color.WHITE, Color.RED, etc`
- Dentro de uma Classe
  - Forma de referenciar  
`Externa.Color.WHITE, Externa.Color.RED, etc`

5

## Tipos Enumerados em JAVA

- **enum** é uma *classe*, não um tipo primitivo.
  - São Objectos - podemos utilizar em Collections;
  - Pode implementar uma Interface.
  - Suportam comparação (== ou equals()).
- Tipos enumerados não são inteiros.
- Só têm construtores privados.
- Os valores enumerados são automaticamente public, static, final.

6

## Enum - uma Classe

- Podemos ter tipos Enumerados com dados e operações associadas:

```
public enum Color {  
    WHITE(21), BLACK(22), RED(23), YELLOW(24), BLUE(25);  
  
    // Dados  
    private int code;  
  
    // Construtor  
    private Color(int c) { code = c; }  
  
    // Método  
    public int getCode() { return code; }  
}
```

7

## Enum - implements Interface

- Os tipos enum podem implementar Interfaces

```
public enum Color implements Runnable {  
    WHITE, BLACK, RED, YELLOW, BLUE;  
  
    public void run() {  
        System.out.println("name()=" + name() + ",  
                           toString()=" + toString());  
    }  
}  
  
▪ Utilização:  
    for(Color c : Color.values()) { c.run();}  
▪ Ou  
    for(Runnable r : Color.values()) { r.run();}
```

8

## Enum - Métodos Disponíveis

- São Comparable (têm uma ordem).
- Fornecem alguns métodos úteis:
  - `toString()`
  - `valueOf(String val)` : converte a String (elemento do conjunto) para um valor
  - `ordinal()`: posição (int) do valor na lista de elementos
  - `values()`: devolve a lista de elementos

9

## Enum - `toString`

- Por omissão, a representação tipo String é o próprio nome de cada elemento. No entanto, podemos modificar redefinindo o método `toString()`.

```
public enum MyType {  
    ONE {  
        public String toString() {return "this is one";}  
    },  
  
    TWO {  
        public String toString() {return "this is two";}  
    }  
}
```

- Main:

```
public class EnumTest {  
    public static void main(String[] args){  
        System.out.println(MyType.ONE);  
        System.out.println(MyType.TWO);  
    }  
}
```

this is one  
this is two

10

## Enum - values()

- O método “values()” retorna um array com todos os elementos.

```
Name[] nameValues = Name.values();
```

- Exemplo de Utilização:

```
for (Name n : Name.values()) {  
    // ...;  
}
```

11

## Enum - Switch Statement

- A instrução switch funciona com enumerados

```
Color myColor = Color.BLACK;  
  
switch(myColor){  
    case WHITE: ...;  
    case BLACK: ...;  
    ...  
    case BLUE: ...;  
    default: ...;  
}
```

12

## Enum - ordinal()

- Definimos um Enum para os Meses do Ano

```
public enum Mes {  
    JANEIRO, FEVEREIRO, MARCO, ABRIL, MAIO,  
    JUNHO, JULHO, AGOSTO, SETEMBRO, OUTUBRO,  
    NOVEMBRO, DEZEMBRO ;  
}
```

- Se utilizarmos na classe Data

```
// Construtor de Data  
public Data (int iDia, Mes iMes, int iAno){  
    //...  
}  
// Main  
Data d1 = new Data(11, Mes.MAIO, 1900);
```

13

## Exemplo 1

```
enum Mes {  
    JANEIRO, FEVEREIRO, MARCO, ABRIL,  
    MAIO, JUNHO, JULHO, AGOSTO,  
    SETEMBRO, OUTUBRO, NOVEMBRO, DEZEMBRO ;  
}  
  
public class Enum1 {  
    public static void main(String[] args) {  
        for (Mes t: Mes.values())  
            System.out.println(t+" : "+t.name()+" : "+t.ordinal());  
    }  
}
```

JANEIRO, JANEIRO : 0  
FEVEREIRO, FEVEREIRO : 1  
MARCO, MARCO : 2  
ABRIL, ABRIL : 3  
...

14

```

enum Mes {
    JANEIRO(1), FEVEREIRO(2), MARCO(3), ABRIL(4),
    MAIO(5), JUNHO(6), JULHO(7), AGOSTO(8),
    SETEMBRO(9), OUTUBRO(10), NOVEMBRO(11), DEZEMBRO(12);

    private final int mes;
    private Mes(int m) {
        this.mes=m;
    }
    public int numMes() {
        return mes;
    }
    @Override public String toString() {
        return this.name().substring(0, 1)+ 
            (this.name().substring(1,this.name().length())).toLowerCase();
    }
}

public class Enum2 {
    public static void main(String[] args) {
        for (Mes t: Mes.values())
            System.out.println(t+" "+t.name()+" : "+
                t.ordinal() + ", "+t.numMes());
    }
}

```

Janeiro, JANEIRO : 0, 1  
 Fevereiro, FEVEREIRO : 1, 2  
 ...

### Exemplo 3

```

public enum Ensemble {
    SOLO(1), DUET(2), TRIO(3), QUARTET(4), QUINTET(5),
    SEXTET(6), SEPTET(7), OCTET(8), DOUBLE_QUARTET(8),
    NONET(9), DECTET(10), TRIPLE_QUARTET(12);

    private final int numberOfWorkers;
    Ensemble(int size) {
        numberOfWorkers = size;
    }

    public int numberOfWorkers() {
        return numberOfWorkers;
    }
}

```

```

public enum Planet {
    MERCURY (3.303e+23, 2.4397e6),
    VENUS   (4.869e+24, 6.0518e6),
    EARTH   (5.976e+24, 6.37814e6),
    MARS    (6.421e+23, 3.3972e6),
    JUPITER (1.9e+27,   7.1492e7),
    SATURN  (5.688e+26, 6.0268e7),
    URANUS  (8.686e+25, 2.5559e7),
    NEPTUNE (1.024e+26, 2.4746e7),
    PLUTO   (1.27e+22,  1.137e6);

    private final double mass; // in kilograms
    private final double radius; // in meters
    Planet(double mass, double radius) {
        this.mass = mass;      this.radius = radius;
    }
    public double mass() { return mass; }
    public double radius() { return radius; }

    // universal gravitational constant (m3 kg-1 s-2)
    public static final double G = 6.67300E-11;

    public double surfaceGravity() {return G*mass/(radius*radius); }
    public double surfaceWeight(double otherMass) {
        return otherMass * surfaceGravity();
    }
}

```

# Java Expressões Lambda (Java 8)

UA, DETI, Programação III  
José Luis Oliveira, Carlos Costa  
2018/19

1

## Cálculo lambda

- As linguagens de programação funcional são baseadas no cálculo lambda (cálculo- $\lambda$ ).
  - [Lisp](#), [Haskell](#), [Scheme](#)
- O cálculo lambda pode ser visto como uma linguagem de programação abstrata em que funções podem ser combinadas para formar outras funções.
- Ideia geral: formalismo matemático
  - $x \rightarrow f(x)$  i.e.  $x$  é *transformado em*  $f(x)$
- O cálculo lambda trata as funções como *elementos de primeira classe*
  - podem ser utilizadas como argumentos e retornadas como valores de outras funções.

## Sintaxe

- Uma expressão lambda descreve uma função anónima
- Representa-se na forma:
  - (argument) -> (body)  
`(int a, int b) -> { return a + b; }`
- Pode ter zero, um, ou mais argumentos
  - () -> { body }  
`() -> System.out.println("Hello World");`
  - (arg1, arg2...) -> { body }
- O tipo dos argumentos pode ser explicitamente declarado ou inferido
  - (type1 arg1, type2 arg2...) -> { body }  
`(int a, int b) -> { return a + b; }`  
`a -> return a*a // um argumento - podemos omitir (a)`
- O corpo (body) pode ter uma ou mais instruções

3

## Exemplos

lambda expression	equivalent method
<code>() -&gt; { System.gc(); }</code>	<code>void nn() { System.gc(); }</code>
<code>(int x) -&gt; { return x+1; }</code>	<code>int nn(int x) return x+1; }</code>
<code>(int x, int y) -&gt; { return x+y; }</code>	<code>int nn(int x, int y) { return x+y; }</code>
<code>(String... args) -&gt; { return args.length; }</code>	<code>int nn(String... args) { return args.length; }</code>
<code>(String[] args) -&gt; {     if (args != null)         return args.length;     else         return 0; }</code>	<code>int nn(String[] args) {     if (args != null)         return args.length;     else         return 0; }</code>

4

## Como usar?

- Uma expressão lambda não pode ser isoladamente

```
(n) -> (n % 2)==0 // Erro de compilação
```

- Precisamos de outro mecanismo adicional

- Interfaces funcionais
  - onde as expressões lambda passam a ser implementações de métodos abstratos.
  - O compilador Java converte uma expressão lambda num método privado da classe (isto é um processo interno).

5

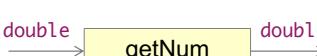
## Functional interfaces

- Uma interface funcional contém apenas um método/função abstrata

- Método abstrato numa interface? Não são todos?
  - A partir do JDK 8 passa a ser possível definir um comportamento por omissão nos métodos de uma interface (*default method*)

- Exemplo

```
@FunctionalInterface  
interface MyNum {  
    double getNum(double n);  
}  
  
public class Lambda1 {  
  
    public static void main(String[] args) {  
        MyNum n1 = (x) -> x+1;  
        // qualquer expressão que transforme double em double  
        System.out.println(n1.getNum(10));  
        n1 = (x) -> x*x;  
        System.out.println(n1.getNum(10));  
    }  
}
```



11.0  
100.0

6

## Exemplos

```
@FunctionalInterface  
interface Ecra {  
    void escreve(String s);  
}  
  
public class Lambda2 {  
  
    public static void main(String[] args) {  
  
        Ecra xd = (String s) -> {  
            if (s.length() > 2)  
                System.out.print(s);  
            else  
                System.out.print("-");  
        };  
        xd.escreve("Lamdba print");  
        xd.escreve("?");  
    }  
}
```

interface funcional

Lamdba print-

7

## Exemplos

```
// Another functional interface.  
interface NumericTest {  
    boolean test(int n);  
}  
  
class Lambda3 {  
    public static void main(String args[]) {  
        // A lambda expression that tests if a number is even.  
        NumericTest isEven = (n) -> (n % 2) == 0;  
        if (isEven.test(10)) System.out.println("10 is even");  
        if (!isEven.test(9)) System.out.println("9 is not even");  
        // Now, use a lambda expression that tests if a number is non-negative.  
        NumericTest isNonNeg = (n) -> n >= 0;  
        if (isNonNeg.test(1)) System.out.println("1 is non-negative");  
        if (!isNonNeg.test(-1)) System.out.println("-1 is negative");  
    }  
}
```

10 is even  
9 is not even  
1 is non-negative  
-1 is negative

8

## Exemplos

```
// Demonstrate a lambda expression that takes two parameters.  
interface NumericTest2 {  
    boolean test(int n, int d);  
}  
  
public class Lambda4 {  
    public static void main(String args[]) {  
        // This lambda expression determines if one number is  
        // a factor of another.  
        NumericTest2 isFactor = (n, d) -> (n % d) == 0;  
        if (isFactor.test(10, 2))  
            System.out.println("2 is a factor of 10");  
        if (!isFactor.test(10, 3))  
            System.out.println("3 is not a factor of 10");  
    }  
}
```

```
2 is a factor of 10  
3 is not a factor of 10
```

9

## Exemplos

```
// A block lambda that computes the factorial of an int value.  
interface NumericFunc {  
    int func(int n);  
}  
  
class Lambda5 {  
    public static void main(String args[]) {  
        // This block lambda computes the factorial of an int value.  
        NumericFunc factorial = (n) -> {  
            int result = 1;  
            for (int i = 1; i <= n; i++)  
                result = i * result;  
            return result;  
        };  
        System.out.println("The factorial of 3 is " + factorial.func(3));  
        System.out.println("The factorial of 5 is " + factorial.func(5));  
    }  
}
```

```
The factorial of 3 is 6  
The factorial of 5 is 120
```

10

## Exemplos

```
// A block lambda that reverses the characters in a string.  
interface StringFunc {  
    String func(String n);  
}  
class Lambda6 {  
    public static void main(String args[]) {  
        // This block lambda reverses the characters in a string.  
        StringFunc reverse = (str) -> {  
            String result = "";  
            int i;  
            for (i = str.length() - 1; i >= 0; i--)  
                result += str.charAt(i);  
            return result;  
        };  
        System.out.println("Lambda reversed is " + reverse.func("Lambda"));  
        System.out.println("Expression reversed is "  
            + reverse.func("Expression"));  
    }  
}
```

Lambda reversed is adbmaL  
Expression reversed is noisserpxE

11

## Interfaces funcionais genéricas

- Dos 2 exemplos anteriores

```
// A block lambda that computes the factorial of an int value.  
interface NumericFunc { int func(int n); }  
  
// A block lambda that reverses the characters in a string.  
interface StringFunc { String func(String n); }
```

- Repetição! Solução?

```
// A generic functional interface.  
interface SomeFunc<T> {  
    T func(T n); }  
    interface funcional genérica
```

- Utilização

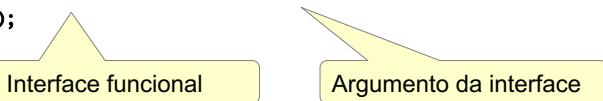
```
SomeFunc<String> reverse = ...  
SomeFunc<Integer> factorial = ...
```

12

## Expressões Lambda como argumento

- Exemplo

```
interface MyFunc<T> {
    T func(T n);
}
...
// Funções que aceita uma expressão lambda e o seu argumento (T n)
static String stringOp(MyFunc<String> sf, String s) {
    return sf.func(s);
}
...
// Outro exemplo
static Person PersonOp(MyFunc<Person> sf, Person s) {
    return sf.func(s);
}
```



13

## Expressões Lambda como argumento

- Utilização

```
String inStr = "Lambdas add power to Java";
String outStr = stringOp((str) -> str.toUpperCase(), inStr);
System.out.println("The string in uppercase: " + outStr);
// This passes a block lambda that removes spaces.
outStr = stringOp((str) -> {
    StringBuilder result = new StringBuilder();
    for(int i = 0; i < str.length(); i++)
        if(str.charAt(i) != ' ')
            result.append( str.charAt(i) );
    return result.toString();
}, inStr);
System.out.println("The string with spaces removed: " + outStr);
```

```
The string in uppercase: LAMBDAS ADD POWER TO JAVA
The string with spaces removed: LambdasaddpowertoJava
```

14

## Package java.util.function

- Várias interfaces funcionais são fornecidas pelo Java SE 8
  - Servem como ponto de partida para situações standard

Exemplos:

- **Predicate**: A property of the object passed as argument
  - boolean test(T t)
- **Consumer**: An action to be performed with the object passed as argument
  - void accept(T t)
- **Function**: Transform a T to a U
  - Rapply(T t)
- **Supplier**: Provide an instance of a T (such as a factory)
  - T get()

15

### java.util.function.Predicate<T>

```
@FunctionalInterface
public interface Predicate<T> {
    boolean test(T t);
    default Predicate<T> and(Predicate<? super T> other) {
        Objects.requireNonNull(other);
        return (t) -> test(t) && other.test(t);
    }
    default Predicate<T> negate() {
        return (t) -> !test(t);
    }
    default Predicate<T> or(Predicate<? super T> other) {
        Objects.requireNonNull(other);
        return (t) -> test(t) || other.test(t);
    }
    static <T> Predicate<T> isEqual(Object targetRef) {
        return (null == targetRef)
            ? Objects::isNull
            : object -> targetRef.equals(object);
    }
}
```

5 métodos? Default?? Static ???

16

## Métodos estáticos em Interfaces

```
interface X {  
    static void foo() {  
        System.out.println("foo");  
    }  
}  
  
class Y implements X {  
}  
  
public class Z {  
    public static void main(String[] args) {  
        X.foo();  
        // Y.foo(); // won't compile  
    }  
}
```

17

## java.util.function.Predicate

```
public static void printPersonsWithPredicate(  
    List<Person> roster, Predicate<Person> tester) {  
    for (Person p : roster) {  
        if (tester.test(p)) {  
            p.printPerson();  
        }  
    }  
}
```

Funções genérica para filtragem da lista

```
printPersonsWithPredicate(  
    roster,  
    p -> p.getGender() == Person.Sex.MALE  
        && p.getAge() >= 18  
        && p.getAge() <= 25  
);
```

Exemplo de filtro

18

## Versão genérica

```
public static <X, Y> void processElements(
    Iterable<X> source,
    Predicate<X> tester,
    Function <X, Y> mapper,
    Consumer<Y> block) {
    for (X p : source) {
        if (tester.test(p)) {
            Y data = mapper.apply(p);
            block.accept(data);
        }
    }
}

processElements(
    roster,
    p -> p.getGender() == Person.Sex.MALE
        && p.getAge() >= 18
        && p.getAge() <= 25,
    p -> p.getEmailAddress(),
    email -> System.out.println(email)
);
```

Mais sobre isto  
quando falarmos  
em *Collections*

19

## Utilização de expressões lambda

- Interfaces funcionais devem ter um único método abstrato (podem ter outros métodos *default*)
  - Classes anónimas com um único método
  - ActionListener, Runnable, ...
  - Maior versatilidade - podemos criar funções sem "poluir" a interface pública da classe.
- Java Collections

```
//Old way:
List<Integer> list = Arrays.asList(1, 2, 3, 4, 5, 6, 7);
for(Integer n: list) {
    System.out.println(n);
}
//New way:
List<Integer> list = Arrays.asList(1, 2, 3, 4, 5, 6, 7);
list.forEach(n -> System.out.println(n));
//or we can use :: double colon operator in Java 8 (method reference)
list.forEach(System.out::println);
```

20

## A history - Improving Processes (1)

```
// I have a List<Person> ...           // I want to do things with the list...

public class Person {                  // Step 1:
    public enum Sex {                 // Search for members (age condition)
        MALE, FEMALE
    }
    String name;
    LocalDate birthday;
    Sex gender;
    String emailAddress;
    public int getAge() {           // ...
        // ...
    }
    public void printPerson() {      // ...
    }
}
```

21

## A history - Improving Processes (2)

```
// Step 2:
// More generic approach -> specify
// the search criteria
// Usage - first approach
printPersons(roster,
new CheckPerson() {
    public boolean test(Person p) {
        return p.getGender() == Person.Sex.MALE
            && p.getAge() >= 18
            && p.getAge() <= 25;
    }
}
);
// Usage - second approach
// Lambda Expression
printPersons(roster,
    (Person p) -> p.getGender() ==
        Person.Sex.MALE
        && p.getAge() >= 18
        && p.getAge() <= 25
);

```

22

## A history - Improving Processes (3)

```
// Step 3:  
// Using a Standard Functional  
Interface  
  
interface Predicate<T> {  
    boolean test(T t);  
}  
  
interface CheckPerson {  
    boolean test(Person p);  
}  
  
public static void printPersons(  
    List<Person> roster,  
    Predicate<Person> tester) {  
    for (Person p : roster) {  
        if (tester.test(p)) {  
            p.printPerson();  
        }  
    }  
}
```

23

## A history - Improving Processes (4)

```
// Step 4:  
// Using more standard functional interface to create more generic app  
  
public static void processPersons (  
    Iterable<Person> roster,  
    Predicate<Person> tester,  
    Function<Person, String> mapper,  
    Consumer<String> block) {  
    for (Person p : roster) {  
        if (tester.test(p)) {  
            String data = mapper.apply(p);  
            block.accept(data);  
        }  
    }  
}  
  
processPersons( roster,  
    p -> p.getGender() == Person.Sex.MALE && p.getAge() >= 18  
        && p.getAge() <= 25,  
    p -> p.getEmailAddress(),  
    email -> System.out.println(email)  
);
```

24

## A history - Improving Processes (5)

```
// Step 5:  
// Java Streams... next lessons  
  
roster  
    .stream()  
    .filter(  
        p -> p.getGender() == Person.Sex.MALE  
            && p.getAge() >= 18  
            && p.getAge() <= 25)  
    .map(p -> p.getEmailAddress())  
    .forEach(email -> System.out.println(email));
```

Filter objects that match a Predicate object	Stream<T> <b>filter</b> (Predicate<? super T> predicate)
Map objects to another value as specified by a Function object	<R> Stream<R> <b>map</b> (Function<? super T,? extends R> mapper)
Perform an action as specified by a Consumer object	void <b>forEach</b> (Consumer<? super T> action)

25

# Java Sistema de Entrada e Saída (I/O)

Programação III  
José Luis Oliveira; Carlos Costa

1

1

## Operações de entrada/saída

### Entrada



*leitura*



### Saída



*escrita*

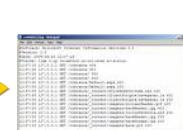
### ficheiro



*leitura*



### ficheiro



*escrita*

2

2

1

## Introdução

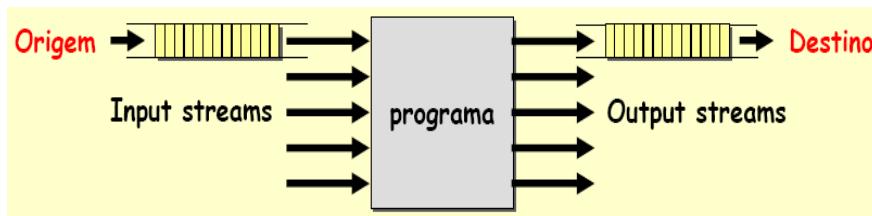
- Sem capacidade de interagir com o "resto do mundo", o nosso programa torna-se inútil
  - Esta interacção designa-se “input/output” (I/O)
- Problema → Complexidade
  - Diferentes e complexos dispositivos de I/O (ficheiros, consolas, canais de comunicação, ...)
  - Diferentes formatos de acesso (sequencial, aleatório, binário, caracteres, linha, palavras, ...)
- Necessidade → Abstracção
  - Libertar o programador da necessidade de lidar com as especificidade e complexidade de cada I/O
- Em Java, a abstracção I/O chama-se “Streams”

3

3

## I/O Streams

- O que são Streams?
  - um fluxo de dados que pode entrar ou sair de um programa.



4

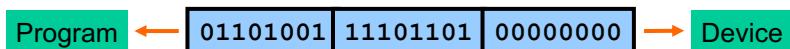
4

2

## Tipos de Streams

- Byte Streams

- binárias (machine-formatted)
- dados transferidos sem serem alterados de forma alguma
- não são interpretados
- não são feitos juízos sobre o seu valor



- Character Streams

- Os dados estão na forma de caracteres (human-readable data)
- interpretados e transformados de acordo com formatos de representação de texto



5

5

## Streams

- Os streams são objectos em Java. Temos 4 classes abstractas para lidar com I/O:

- **InputStream**: byte-input
- **OutputStream**: byte-output
- **Reader**: text-input
- **Writer**: text-output

Classes Abstractas

- Todas as classes de I/O são derivadas destas

- Entrada - **InputStream** (byte); **Reader** (char)
- Saída - **OutputStream** (byte); **Writer** (char)



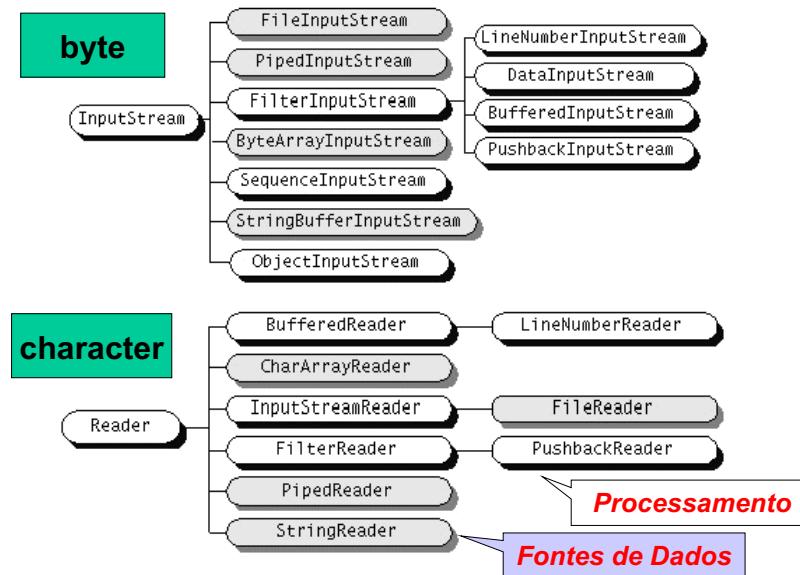
- Estas classes estão incluídas no package `java.io`

6

6

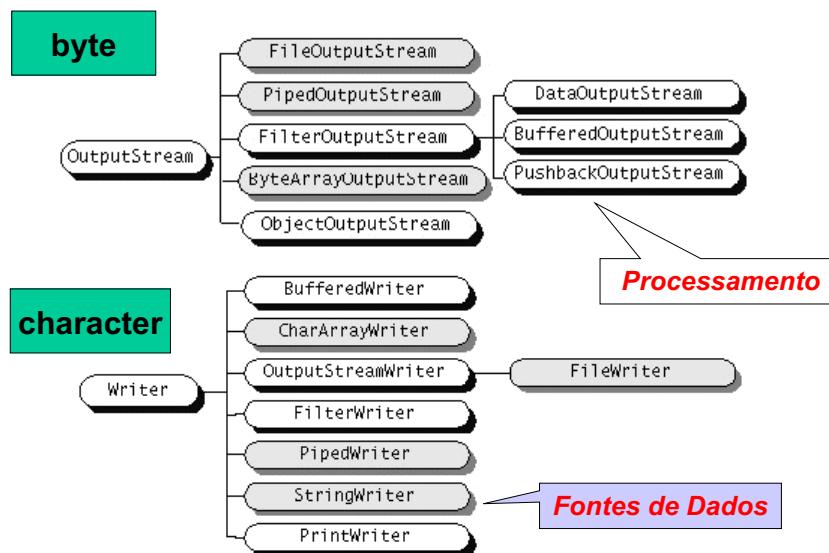
3

## Streams de Entrada



7

## Streams de Saída



8

8

4

## InputStream/Reader

- Reader e InputStream têm interfaces semelhantes mas tipos de dados diferentes
- Reader
  - int read()
  - int read(char cbuf[])
  - int read(char cbuf[], int offset, int length)
- InputStream
  - int read()
  - int read(byte cbuf[])
  - int read(byte cbuf[], int offset, int length)

9

9

## OutputStream/Writer

- Writer e OutputStream têm interfaces semelhantes mas tipos de dados diferentes
- Writer
  - int write()
  - int write(char cbuf[])
  - int write(char cbuf[], int offset, int length)
- OutputStream
  - int write()
  - int write(byte cbuf[])
  - int write(byte cbuf[], int offset, int length)

10

10

5

## Standard I/O

- **System.in** é do tipo `InputStream`

```
byte[] b = new byte[10];
InputStream stdin = System.in;
stdin.read(b);
```

- **System.out** é do tipo `PrintStream` (sub-tipo de `OutputStream`)

```
OutputStream stdout = System.out;
stdout.write(104); // ASCII 'h'
stdout.flush();
```

Field Summary	
java.lang.System	
static <code>PrintStream</code>	<code>err</code> The "standard" error output stream.
static <code>InputStream</code>	<code>in</code> The "standard" input stream.
static <code>PrintStream</code>	<code>out</code> The "standard" output stream.

11

11

## Utilização de Streams

Sink Type (Fontes de Dados)	Character Streams	Byte Streams
Memory	CharArrayReader, CharArrayWriter	ByteArrayInputStream, ByteArrayOutputStream
	StringReader, StringWriter	StringBufferInputStream
Pipe	PipedReader, PipedWriter	PipedInputStream, PipedOutputStream
File	FileReader, FileWriter	FileInputStream, FileOutputStream

12

12

## Classes de processamento

Process	CharacterStreams	Byte Streams
Buffering	BufferedReader, BufferedWriter	BufferedInputStream, BufferedOutputStream
Filtering	FilterReader, FilterWriter	FilterInputStream, FilterOutputStream
Converting between Bytes and Characters	InputStreamReader, OutputStreamWriter	
Concatenation		SequenceInputStream
Object Serialization		ObjectInputStream, ObjectOutputStream
Data Conversion		DataInputStream, DataOutputStream
Counting	LineNumberReader	LineNumberInputStream
Peeking Ahead	PushbackReader	PushbackInputStream
Printing	PrintWriter	PrintStream

13

## Ficheiros

- Classes principais:
- Java 7

[FileReader](#)

[FileWriter](#)

[FileInputStream](#)

[FileOutputStream](#)

[File](#)

[Path](#)

[Paths](#)

[Files](#)

[RandomAccessFile](#)

[SeekableByteChannel](#)

[Scanner \(java 5\)](#)

14

14

## Classes de processamento - wrappers

- Exemplos

```
DataInputStream src = new DataInputStream(System.in);  
BufferedReader in = new BufferedReader(src);  
...  
PrintWriter out = new PrintWriter(  
    new FileWriter("ARQUIVO.TXT"));
```

- A fonte é um objecto do tipo **DataInputStream** que por sua vez é aberto sobre uma outra fonte
  - **DataInputStream** decorado por **BufferedReader**
  - Desta forma pode usar-se o método **readLine** de **BufferedReader**
- Do mesmo modo **FileWriter** é adaptado (wrapped) num **PrintWriter** para que o programa possa usar o método **println**.
- Desta forma podemos combinar facilidades fornecidas por diferentes manipuladores de streams.

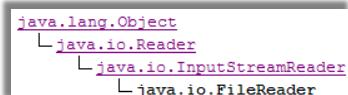
20

20

## BufferedReader

- Leitura de caracteres do System.in

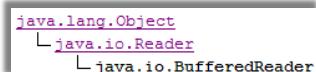
```
InputStreamReader isr = new InputStreamReader(System.in);  
char c;  
c = (char) isr.read();  
System.out.write(c);
```



- Esta leitura caracter-a-caracter não é eficiente!

- Podemos querer ler uma linha

```
BufferedReader stdin = new BufferedReader(  
    new InputStreamReader(System.in));  
  
System.out.print("Digite uma linha:");  
String linha = stdin.readLine();
```



21

21

## Leitura de dados do teclado

- Exemplo de leitura de um inteiro em Java 1.4

```
try {  
    BufferedReader r =  
        new BufferedReader (  
            new InputStreamReader(System.in));  
    String s = r.readLine();  
    int i = (new Integer(s)).intValue();  
    System.out.println(i);  
} catch(IOException e) { ... }
```

- ... e em Java 5.0

```
Scanner sc = new Scanner(System.in);  
int i = sc.nextInt();  
System.out.println(i);
```

- Compare a manipulação de Excepções!

22

22

## java.io.RandomAccessFile

- Vê uma file como uma sequência de bytes
- Possui um ponteiro (seek) para ler ou escrever em qualquer ponto do ficheiro.
- Genericamente inclui operações **seek**, **read**, **write**
- Podemos apenas ler ou escrever tipos primitivos
  - `writeByte()`, `writeln()`, `writeBoolean()`
  - `writeChars(String s)`, `writeUTF(String str)`, `String readLine()`

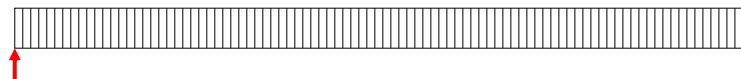
A partir de Java 7 existem outras classes / métodos

... `FileChannel`

23

23

## java.io.RandomAccessFile

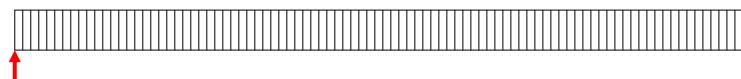


```
// In the file “./mydata”, copy bytes 10-19 to 0-9.  
RandomAccessFile file = new RandomAccessFile("mydata", "rw");  
byte[] buf = new byte[10];  
file.seek(10); file.read(buf);  
file.seek(0); file.write(buf);
```

24

24

## java.io.RandomAccessFile

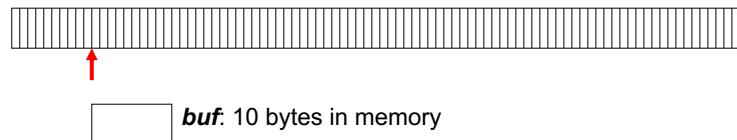


```
// In the file “./mydata”, copy bytes 10-19 to 0-9.  
RandomAccessFile file = new RandomAccessFile("mydata", "rw");  
byte[] buf = new byte[10];  
file.seek(10); file.read(buf);  
file.seek(0); file.write(buf);
```

25

25

## java.io.RandomAccessFile

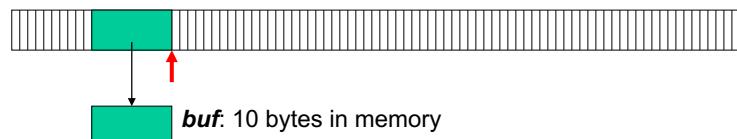


```
// In the file "./mydata", copy bytes 10-19 to 0-9.  
RandomAccessFile file = new RandomAccessFile("mydata", "rw");  
byte[] buf = new byte[10];  
file.seek(10); file.read(buf);  
file.seek(0); file.write(buf);
```

26

26

## java.io.RandomAccessFile

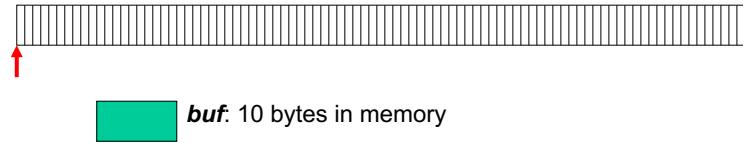


```
// In the file "./mydata", copy bytes 10-19 to 0-9.  
RandomAccessFile file = new RandomAccessFile("mydata", "rw");  
byte[] buf = new byte[10];  
file.seek(10); file.read(buf);  
file.seek(0); file.write(buf);
```

27

27

## java.io.RandomAccessFile

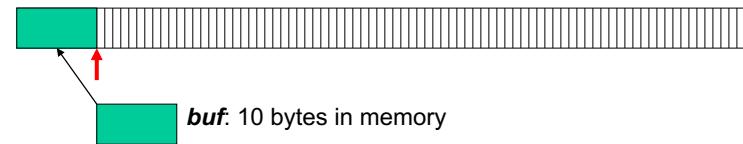


```
// In the file "./mydata", copy bytes 10-19 to 0-9.  
RandomAccessFile file = new RandomAccessFile("mydata", "rw");  
byte[] buf = new byte[10];  
file.seek(10); file.read(buf);  
file.seek(0); file.write(buf);
```

28

28

## java.io.RandomAccessFile



```
// In the file "./mydata", copy bytes 10-19 to 0-9.  
RandomAccessFile file = new RandomAccessFile("mydata", "rw");  
byte[] buf = new byte[10];  
file.seek(10); file.read(buf);  
file.seek(0); file.write(buf);
```

29

29

## java.io.RandomAccessFile

- Fazer *append* a um ficheiro que já existe.

```
File f = new File("um_ficheiro_qualquer");
RandomAccessFile raf = new RandomAccessFile(f, "rw");

// Seek to end of file
raf.seek(f.length());

// Append to the end
raf.writeChars("agora é que é o fim");
raf.close();
```

30

30

## Java NIO - What's New?

- New Abstractions
  - Buffers, Channels and Selectors
- New Capabilities
  - Non-Blocking\* Sockets
  - File Locking
  - Memory Mapping
  - Readiness Selection
  - Regular Expressions
  - Pluggable Charset Transcoders

>=Java 7

IO	NIO
Stream oriented	Buffer oriented
Blocking IO	Non blocking IO
	Selectors

\* more relevant in Prog 3 context

31

31

13

## Stream Oriented vs. Buffer Oriented

- Stream oriented (IO)
  - read one or more bytes at a time from a stream
  - cannot move forth and back in the stream data
- Buffer oriented (NIO)
  - data is read into a buffer from which it is later processed
  - can move forth and back in the buffer as needed
  - need to check if the buffer contains all the data you need in order to fully process it

32

32

## Blocking vs. Non-blocking IO

- Streams are blocking
  - `read()` or `write()`: thread is blocked until there is some data to read, or the data is fully written
  - thread can do nothing else in the meantime
- Non-blocking NIO
  - reading data from a channel - only get what is currently available (nothing if no data is currently available); thread can do something else
  - writing data to channel - thread can request data writing but not waits (blocks) for process conclusion

34

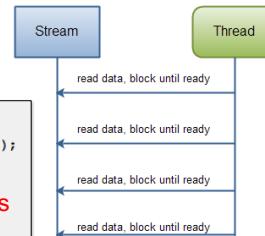
34

14

## Data Processing - Stream vs Buffer

- Text Stream

```
Name: Anna
Age: 25
Email: anna@mailserver.com
Phone: 1234567890
```

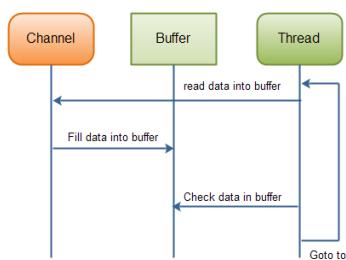


- Stream Oriented

```
InputStream input = ... ; // get the InputStream from the client socket
BufferedReader reader = new BufferedReader(new InputStreamReader(input));
String nameLine = reader.readLine(); // blocks
String ageLine = reader.readLine(); // blocks
String emailLine = reader.readLine(); // blocks
String phoneLine = reader.readLine(); // blocks
```

- Buffer (Channel) Oriented

```
ByteBuffer buffer = ByteBuffer.allocate(48);
int bytesRead = inChannel.read(buffer);
while(! bufferFull(bytesRead) ) {
    bytesRead = inChannel.read(buffer);
}
```



35

35

## Java NIO - Classes e Interfaces

- Mudanças significativas nas classes principais
- Classe `java.nio.file.Files`
  - Só métodos estáticos para manipular ficheiros, directórios,..
- Classe `java.nio.file.Paths`
  - Só métodos estáticos para retornar um Path através da conversão de uma string ou Uniform Resource Identifier (URI)
- Interface `java.nio.file.Path`
  - Utilizada para representar a localização de um ficheiro ou sistema de ficheiros, tipicamente system dependent.
- Utilização comum:
  - Usar Paths para obter um Path.
  - Usar Files para realizar operações.

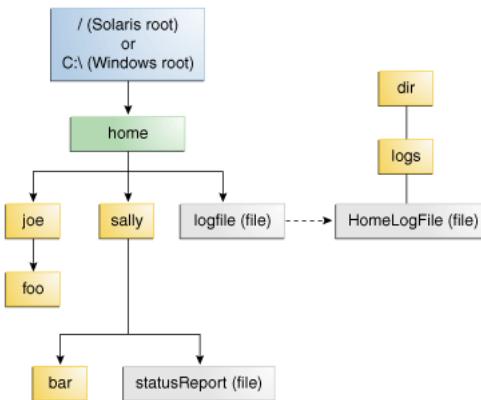


36

36

## java.nio.file.Path

- Notation dependent on the OS
  - /home/sally/statusReport
  - C:\home\sally\statusReport
- Relative or absolute
- Symbolic links
- java.nio.file.Path
  - Interface
  - Path might not exist



<http://docs.oracle.com/javase/tutorial/essential/io/pathOps.html>

37

## java.nio.file.Paths

- Classe auxiliar com 2 métodos estáticos
- Permite converter strings ou um URI num Path

**static Path get(String first, String... more)**

- Converts a path string, or a sequence of strings that when joined form a path string, to a Path.

**static Path get(URI uri)**

- Converts the given URI to a Path object.

38

38

16

## Path

- Criar

```
Path p1 = Paths.get("/tmp/foo");
Path p11 = FileSystems.getDefault().getPath("/tmp/foo"); // <=> p1
Path p2 = Paths.get(args[0]);
Path p3 = Paths.get(URI.create("file:///Users/joe/FileTest.java"));
```

- Criar no home directory logs/foo.log ou logs\foo.log

```
Path p5 = Paths.get(System.getProperty("user.home"), "logs", "foo.log");
```

39

39

## Path

- Alguns métodos:

```
// None of these methods requires that the file corresponding to the Path exists.
// Microsoft Windows syntax
Path path = Paths.get("C:\\home\\joe\\foo");
// Solaris syntax
Path path = Paths.get("/home/joe/foo");

System.out.format("toString: %s%n", path.toString());
System.out.format("getFileName: %s%n", path.getFileName());
System.out.format("getName(0): %s%n", path.getName(0));
System.out.format("getNameCount: %d%n", path.getNameCount());
System.out.format("subpath(0,2): %s%n", path.subpath(0,2));
System.out.format("getParent: %s%n", path.getParent());
System.out.format("getRoot: %s%n", path.getRoot());
```

40

40

## java.nio.file.Files

- Só métodos estáticos
  - copy, create, delete, ..
  - isDirectory, isReadable, isWritable, ..

- Exemplo de cópia de ficheiros

```
Path src = Paths.get("/home/fred/readme.txt");
Path dst = Paths.get("/home/fred/copy_readme.txt");

Files.copy(src, dst,
           StandardCopyOption.COPY_ATTRIBUTES,
           StandardCopyOption.REPLACE_EXISTING);
```

- Move

- Suporta atomic move

```
Path src = Paths.get("/home/fred/readme.txt");
Path dst = Paths.get("/home/fred/readme.1st");

Files.move(src, dst, StandardCopyOption.ATOMIC_MOVE);
```

41

41

## java.nio.file.Files

- delete(Path)

```
try {
    Files.delete(path);
} catch (NoSuchFileException x) {
    System.err.format("%s: no such" + " file or directory%n", path);
} catch (DirectoryNotEmptyException x) {
    System.err.format("%s not empty%n", path);
} catch (IOException x) {
    // File permission problems are caught here.
    System.err.println(x);
}
```

- deleteIfExists(Path)

- Sem exceções

42

42

## java.nio.file.Files

- Verificar se dois Paths indicam a mesma File
  - Num sistema de ficheiros com links simbólicos podemos ter dois caminhos distintos a representar o mesmo ficheiro
  - Usar isSameFile(Path, Path) para fazer a comparação

```
Path p1 = ...;
Path p2 = ...;

if (Files.isSameFile(p1, p2)) {
    // Logic when the paths locate the same file
}
```

43

43

## java.nio.file.DirectoryStream<T>

- Interface DirectoryStream actua como um iterador
  - Scales to large directories
  - Uses less resources
  - Smooth out response time for remote file systems
  - Implements Iterable and Closeable for productivity
- Filtering support
  - Build-in support for glob, regex and custom filters

```
Path srcPath = Paths.get("/home/fred/src");
try (DirectoryStream<Path> dir = Files.newDirectoryStream(srcPath, "*.java")) {
    for (Path file : dir)
        System.out.println(file);
}
```

44

44

## java.nio.file.DirectoryStream Exemplos de glob

- Glob é um padrão para filtragem de ficheiros. Exemplos de sintaxe:

\*.html - Matches all strings that end in .html  
??? - Matches all strings with exactly three letters or digits  
\*[0-9]\* - Matches all strings containing a numeric value  
.{{htm,html, pdf}} - Matches any string ending with .htm, .html or .pdf  
a?\*.java - Matches any string beginning with a, followed by at least one letter or digit, and ending with .java  
{foo\*,\*[0-9]\*} - Matches any string beginning with foo or any string containing a numeric value

- Exemplo de método que usa glob

```
static DirectoryStream<Path> newDirectoryStream(Path dir, String glob)
```

45

45

## Java NIO - Symbolic Links

- Path and Files are “link aware”

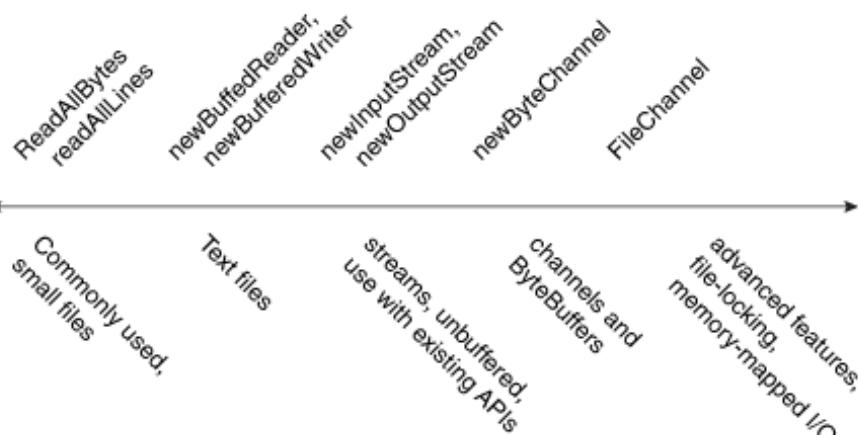
```
Path existingFile = Paths.get(...);  
Path newLink = Paths.get(...);  
  
try {  
    Files.createLink(newLink, existingFile);  
} catch (IOException x) {  
    System.err.println(x);  
} catch (UnsupportedOperationException x) {  
    //Some file systems or some configurations  
    //may not support links  
    System.err.println(x);  
}
```

46

46

20

## Reading, Writing, and Creating Files



<http://docs.oracle.com/javase/tutorial/essential/io/file.html>

47

## Ficheiros pequenos

- Utilizar métodos que lidam com o abrir e fechar do ficheiro
- Reading All Bytes or Lines from a File

```
Path file = ...;
byte[] byteArray;
fileArray = Files.readAllBytes(file);
// text file
List<String> lines = Files.readAllLines(path);
```

Ler/Escrever  
todas as linhas

- Writing All Bytes or Lines to a File

```
Path file = ...;
byte[] buf = ...;
Files.write(file, buf);

// text file
Files.write(path, lines); // lines is a List<String>
```

48

48

21

## newBufferedReader/Writer

- Leitura

```
Charset charset = Charset.forName("US-ASCII");
try (BufferedReader reader = Files.newBufferedReader(file, charset)) {
    String line = null;
    while ((line = reader.readLine()) != null) {
        System.out.println(line);
    }
} catch (IOException x) {
    System.err.format("IOException: %s%n", x);
}
```

Ler/Escrever  
linha-a-linha

- Escrita

```
Charset charset = Charset.forName("US-ASCII");
String s = ...;
try (BufferedWriter writer = Files.newBufferedWriter(file, charset)) {
    writer.write(s, 0, s.length());
} catch (IOException x) {
    System.err.format("IOException: %s%n", x);
}
```

49

49

## FileChannel

- Random access files - para aceder temos de abrir, usar seek e ler/escrever
- A interface **SeekableByteChannel** encapsula agora esta funcionalidade.

**position** – Returns the channel's current position  
**position(long)** – Sets the channel's position  
**read(ByteBuffer)** – Reads bytes into the buffer from the channel  
**write(ByteBuffer)** – Writes bytes from the buffer to the channel  
**truncate(long)** – Truncates the file (or other entity) connected to the channel

- A classe **FileChannel** implementa esta interface and many others...

50

50

## FileChannel

```
try (RandomAccessFile ra = new RandomAccessFile(filename, "rw");
    FileChannel fc = ra.getChannel()){

    System.out.println("FileChannel - Size: " + fc.size());
    System.out.println("FileChannel - Position: " + fc.position());
    fc.position(fc.position() + 30);

    ByteBuffer buf = ByteBuffer.allocate(8);
    int bytesRead = fc.read(buf);
    if (bytesRead > 0)
        System.out.println("FileChannel - Read 30-38: "
            + new String(buf.array()) + "\n");

} catch (IOException x) {
    System.err.format("IOException: %s%n", x);
}
```

51

51

## Iterar sobre a árvore de pasta e ficheiros

- A interface FileVisitor incorpora um conjunto de métodos que torna a navegação na árvore mais fácil.
- A classe SimpleFileVisitor implementa:
  - preVisitDirectory(T dir, BasicFileAttributes attrs);
  - visitFile(T dir, BasicFileAttributes attrs);
  - visitFileFailed(T dir, IOException exc);
  - postVisitDirectory(T dir, IOException exc);
- Geralmente esta classe deve ser extendida para incluir as funcionalidades desejadas.

52

52

## FileVisitor

```
Path inputPath = Paths.get("src");

FileVisitor<Path> pf = new SimpleFileVisitor<Path>() {
    @Override
    public FileVisitResult visitFile(Path file,
                                    BasicFileAttributes attrs) throws IOException {
        System.out.println(file.getFileName()+
                           ": "+files.size(file));
        return FileVisitResult.CONTINUE;
    }
};

Files.walkFileTree(inputPath, pf);
```

BST.java: 13440  
BSTNode.java: 490  
Col1.java: 1086  
Col2.java: 1075  
Queue.java: 533  
DC.java: 502

53

53

## Watching A Directory

- Criar um WatchService
- Registrar num directório
- “Watcher” can be polled or waited on for events
  - Events raised in the form of Keys
  - Retrieve the Key from the Watcher
  - Key has filename and events within it for create/delete/modify

```
import static java.nio.file.StandardWatchEventKinds.*;
Path dir = ...;
try {
    WatchKey key = dir.register(watcher,
                                ENTRY_CREATE, ENTRY_DELETE, ENTRY_MODIFY);
} catch (IOException x) {
    System.err.println(x);
}
```

54

54

## **Resumo de diferenças NIO**

- <http://docs.oracle.com/javase/tutorial/essential/io/legacy.html>
- <http://tutorials.jenkov.com/java-nio/nio-vs-io.html>

55

55

## **Serialização**

56

56

25

## Serialização

- E se quisermos **ler** ou **escrever Objectos** em Ficheiros?
  - **Serialização:** permite tornar persistentes os objectos
- O processo de Serialização é complicado em muitas linguagens
  - Podemos ter objectos contendo referências para outros objectos...
- Java permite implementar Serialização de forma simples
- **Definição:** Serialização é o processo de transformar um objecto numa sequência (stream) de bytes

57

57

## Serialização

- Para que uma classe seja serializável basta que implemente a interface **Serializable** (que é uma interface vazia!)
- ```
package java.io;  
public interface Serializable {  
    // there's nothing in here!  
};
```
- **Serializable** - Permite simplesmente indicar quais as classes serializáveis

58

58

## Condições de Serialização

- A classe deve ser declarada como `public`
- A classe deve implementar `Serializable`
- Todos os atributos (dados) devem ser serializáveis:
  - `Tipos primitivos (int, double, ...)`
  - `Objectos serializáveis`

59

59

## Serialização - Algumas Considerações

- Um atributo definido como `transient` não será “empacotado” no processo de serialização.
  - No processo de desserialização os atributos assumirão valores de defeito.
- Atributos do tipo `static` não são serializados.
- Se uma classe B serializable tem uma super-classe A que não é serializable, então objectos do tipo B podem ser serializados ... desde que a classe A tenha um construtor sem argumentos acessível.

60

60

## Serialização - serialVersionUID

- Atributo **Muito Importante**
- Deve ser sempre incluído na Classe:
  - `private static final long serialVersionUID = 75264722956L;`
- Não deve ser alterado em versões futuras das classes, excepto...
- ... ambas as versões gerarem objectos incompatíveis
  - A compatibilidade de novas versões com objectos antigos depende da natureza das alterações.

61

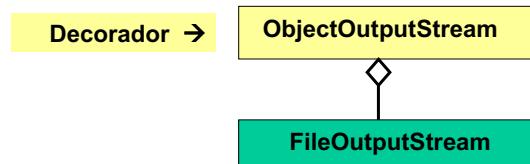
61

## Escrita de Objectos em Ficheiro

```
ObjectOutputStream objectOut =
    new ObjectOutputStream(
        new FileOutputStream(fileName));

objectOut.writeObject(serializableObject);

objectOut.close();
```

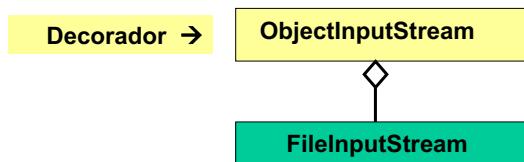


62

62

## Leitura de Objectos de Ficheiro

```
ObjectInputStream objectIn =  
    new ObjectInputStream(  
        new FileInputStream(fileName)));  
  
myObject = (ObjectType)objectIn.readObject();  
  
objectIn.close( );
```



63

63

## Exemplo - Serialização

- **ObjectOutputStream**

```
FileOutputStream out = new FileOutputStream("Time");  
ObjectOutputStream s = new ObjectOutputStream(out);  
s.writeObject("Today");  
s.writeObject(new Date());  
s.flush();
```

- **ObjectInputStream**

```
FileInputStream in = new FileInputStream("Time");  
ObjectInputStream s = new ObjectInputStream(in);  
String today = (String)s.readObject();  
Date date = (Date)s.readObject();
```

- A leitura faz-se pela mesma ordem da escrita

64

64

## Exemplo - Escrita Objectos

```
public static void main(String[] args) throws FileNotFoundException, IOException {
    Data d = new Data(11,2,2001);
    Pessoa p = new Pessoa("Carlos Costa", 234342124, new Data(22,11,1972));

    ObjectOutputStream objectOut = new ObjectOutputStream(
        new FileOutputStream("c:/out.bin"));

    objectOut.writeObject(d);
    objectOut.writeObject(p);
    objectOut.close();
}
```



C:\out.bin

```
ACED 0005 7372 0007 494F 2E44 6174 612D C51E 40EE 1B7D 8E02 0003 4900 0361
6E6F 4900 0364 6961 4900 036D 6573 7870 0000 07D1 0000 000B 0000 0002 7372
0009 494F 2E50 6573 736F 6171 314B 6257 84CE 2102 0003 4900 0262 694C 0008
6461 7461 4E61 7363 7400 094C 494F 2F44 6174 613B 4C00 046E 6F6D 6574 0012
4C6A 6176 612F 6C61 6E67 2F53 7472 696E 673B 7870 0DF7 C6EC 7371 007E 0000
0000 07B4 0000 0016 0000 000B 7400 0C43 6172 6C6F 7320 436F 7374 61
```

65

65

## Serialização - Utilização

- Persistência
  - Com FileOutputStream
  - Armazena as estruturas de dados em ficheiro para mais tarde recuperar
- Cópia
  - Com ByteArrayOutputStream
  - Armazena as estruturas de dados em memória (array) para poder criar duplicados
- Comunicações
  - Utilizando um stream associado a um Socket
  - Envia as estruturas de dados para outro computador

66

66

## Serialização - Deep Copy

```
// serialize object
ByteArrayOutputStream mOut = new ByteArrayOutputStream();
ObjectOutputStream serializer = new ObjectOutputStream(mOut);
serializer.writeObject(serializableObject);
serializer.flush();

// deserialize object
ByteArrayInputStream mIn = new
    ByteArrayInputStream(mOut.toByteArray());
ObjectInputStream deserializer = new ObjectInputStream(mIn);
Object deepCopyOfOriginalObject = deserializer.readObject();
```

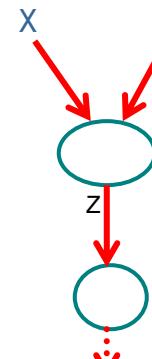
67

67

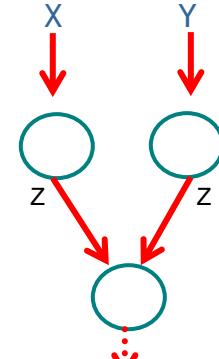
## Java Serializable Comparison

- Example (X.field = Z)

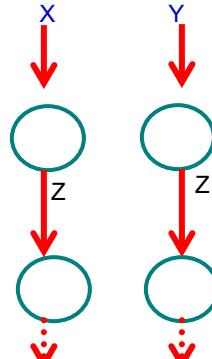
- Y = X



- Y = X.clone( )



- OS.writeObject(x)  
Y = readObject(IS)



69

69

## Jar Files

70

70

### O que são Jar files?

- O Java Archive (JAR) permite a inclusão de múltiplos ficheiros num único ficheiro arquivo.
- Tipicamente, o ficheiro JAR contém “.class files” e recursos auxiliares associados com applets ou aplicações.
- Os ficheiros JAR são compactados em formato ZIP
  - Podemos utilizar o “Winzip” para manipular JARs

71

71

32

## Vantagens

- **Compressão:** O arquivo JAR comprime os seus conteúdos.
  - Aumento da eficiência no transporte (- tempo download) e arquivo (- espaço disco)
- **Segurança:** Os ficheiros JAR podem ser assinados digitalmente.
  - autenticação da proveniência.
  - privilégios do software baseados na certificação da origem.

72

72

## Vantagens

- **Packaging for extensions:** é possível adicionar novas funcionalidades ao Java *core platform*, utilizando arquivos Jar.
- **Package Sealing:** forçar a consistência de versões.
  - Todas as classes definidas no package devem ser encontradas no mesmo arquivo Jar.
- **Package Versioning:** suporta informação relativa ao software: vendedor, versão, etc.
- **Portabilidade:** suporte de JARs é uma componente standard do Java *platform's core API*.

73

73

33

## Java Archive Tool - comando jar

### Operações

- create a JAR file
- view the contents of a JAR file
- extract the contents of a JAR file
- extract specific files from a JAR file
- run an application packaged as a JAR file (version 1.2 -- requires Main-Class manifest header)

### Comando

|                                  |
|----------------------------------|
| jar cf jar-file input-file(s)    |
| jar tf jar-file                  |
| jar xf jar-file                  |
| jar xf jar-file archived-file(s) |
| java -jar app.jar                |

74

74

## Jar - Manifest File

- Ficheiro especial que contém diversos tipos de ‘Meta’ informação relativas ao arquivo JAR:
  - electronic signing, version control, package sealing, entry-point, ...
- Na criação de um JAR é criada uma “default manifest file”

META-INF/MANIFEST.MF



Manifest-Version: 1.0  
Created-By: 1.6.0 (Sun Microsystems Inc.)

75

75

34

## Executable JAR archive

- Como tornar uma aplicação em Java num arquivo JAR executável?
  1. Colocar todas as classes num directório (estrutura árvore)
  2. Criar um arquivo JAR com esse directório
  3. Adicionar na Manifest File um *entry-point*  
**Main-Class: classname**
  4. A main-class deve ter o método  
**public static void main(String[] args)**

Manifest-Version: 1.0  
Class-Path: .  
Main-Class: aula1.Palindrome
  5. Para executar  
**\$ java -jar app.jar**

76

76

## Multi-Release JAR Files

>=Java 9

- Allows bundling code targeting multiple Java releases within the same Jar file.
- Setting **Multi-Release: true** in the MANIFEST.MF file

```
jar root
  - A.class
  - B.class
  - C.class
  - D.class
  - META-INF
    - versions
      - 9
        - A.class
        - B.class
      - 10
        - A.class
```

- JDKs < 9, only the classes in the root entry are visible to the Java runtime.
- JDK 9, the classes A and B will be loaded from the directory root/META-INF/versions/9, while C and D will be loaded from the base entry.
- JDK 10, class A would be loaded from the directory root/META-INF/versions/10.

77

77

## Sumário

- Sistema de Entrada e Saída (I/O)
- Streams de Dados (byte e char)
- Classes de Processamento
- Acesso Aleatório a Ficheiros
- Java NIO
- Serialização de Objetos
- Ficheiros JAR

78

78

# Java Swing, Programação por Eventos

Programação III  
José Luis Oliveira; Carlos Costa

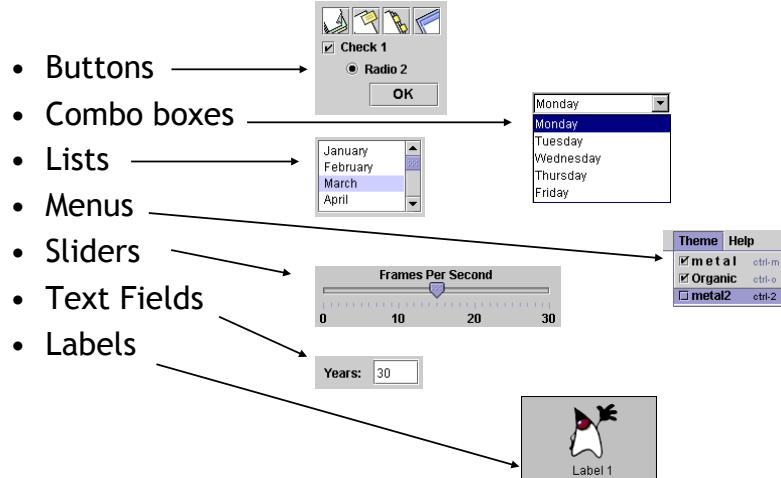
1

## Introdução

- A primeira interface gráfica de utilizador (GUI) incluída na versão 1.0 foi designada por AWT (Abstract Windows Toolkit).  
`java.awt.*`
- A ideia do AWT era criar interfaces que fossem boas em qualquer plataforma.
  - O resultado foi interfaces fracas em todos os sistemas :(
- A versão 2.0 do JDK passou a incluir uma nova API designada por Java Foundation Classes (JFC)
  - Swing  
`javax.swing.*`
  - AWT
  - Java2D
  - Drag-and-Drop

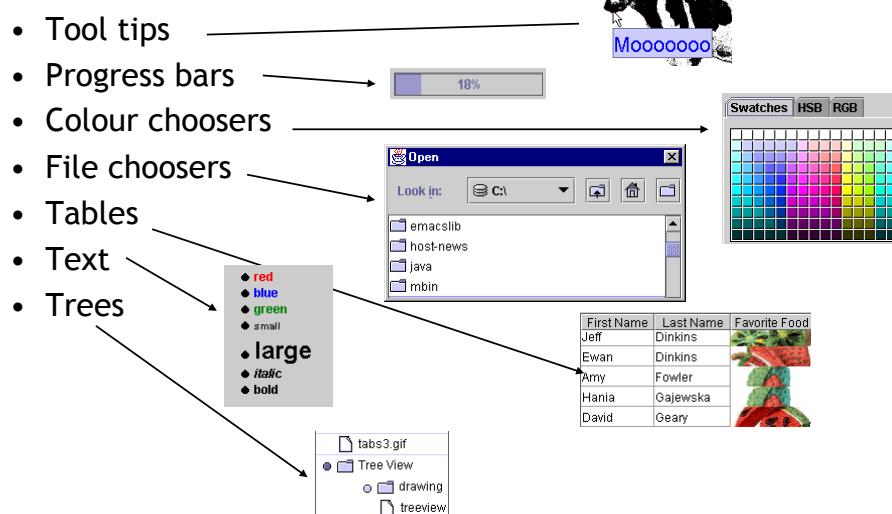
2

## Elementos de uma aplicação gráfica (1)



3

## Elementos de uma aplicação gráfica (2)



4

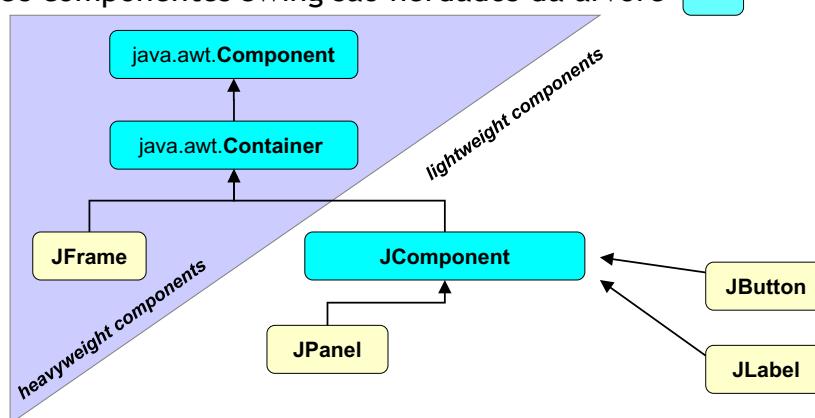
## Elementos de uma Aplicação Swing

- Componentes gráficos
  - Exemplos: windows, buttons, labels, text fields, menus, ...
  - Herdam de javax.swing.JComponent (a um nível mais elevado de java.awt.Component)
- Contentores (Containers)
  - Um Container é um componente onde podem ser colocados outros componentes.
  - Exemplos : JFrame (contentor principal), JPanel (contentor secundário)
- Eventos
  - Permite lidar com acções do utilizador sobre a interface gráfica
  - java.awt.event.\*

5

## Swing GUI

- Os componentes Swing são herdados da árvore



- As classes base definem muitas das funcionalidades encontradas nos diversos componentes Swing.

6

## AWT - Classes principais

- `java.awt.Component`
  - Classe abstracta → Todos os elementos gráficos são Component
  - alguns métodos:
 

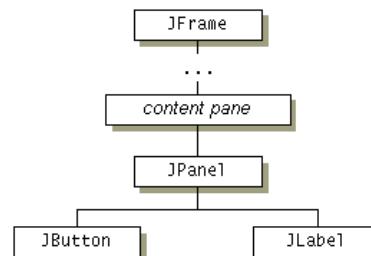
```
void setSize(int width, int height);
void setBackground(Color c);
void setFont(Font f);
void setVisible();
```
- `java.awt.Container`
  - Classe abstracta
  - alguns métodos:
 

```
void setLayout(LayoutManager lman);
void add(Component c);
```

7

## Containers

- Os Containers servem para organizar e gerir todos os componentes de uma aplicação Swing
- Top-Level
  - `JFrame`
  - `JDialog`
  - `JApplet`
  - `JWindow`
- Qualquer aplicação swing deve usar um objecto `JFrame` (ou derivado) como container de topo



8

## JFrame

```
import javax.swing.*;
public class Win1 {
    public static void main(String args[]) {
        JFrame frame = new JFrame("Teste Swing");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    }
}
```



```
import java.awt.*;
import javax.swing.*;

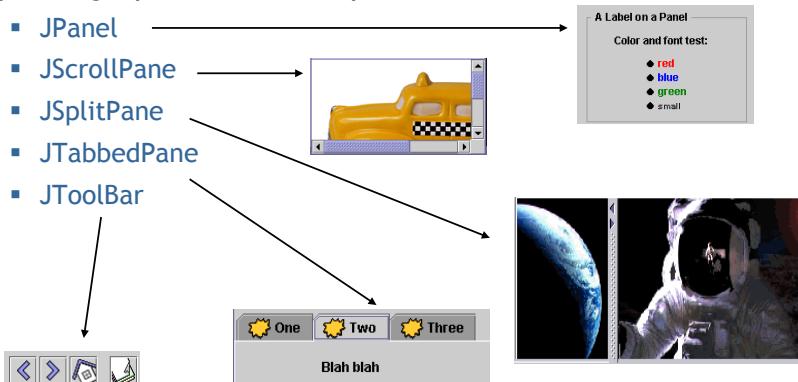
public class Win2 {
    public static void main(String args[]) {
        JFrame frame = new JFrame("Teste Swing 2");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(200,100);
        JLabel label = new JLabel("Hello World");
        label.setBackground(Color.CYAN);
        label.setOpaque(true);
        frame.getContentPane().add(label);
        frame.setVisible(true);
    }
}
```



9

## Painéis - Containers de agregação

- Os Painéis são usados dentro de um container de topo para agrupar outros componentes



10

## JPanel

- O JPanel é um componente:
  - onde se pode desenhar
  - que pode conter outros componentes
  - que permite criar sub-áreas dentro da janela principal

```
//Create a panel and add components to it.
JPanel painel = new JPanel();
painel.add(someComponent);
painel.add(anotherComponent);

//Make it the content pane.
painel.setOpaque(true);
topLevelContainer.setContentPane(painel);
```

11

## JPanel

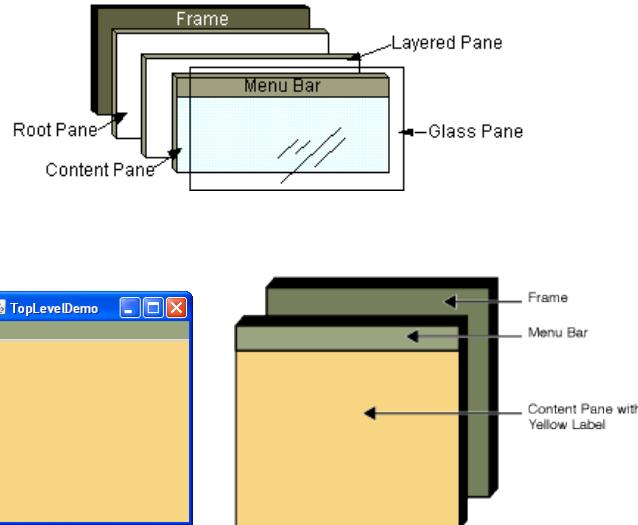
```
import java.awt.*;
import javax.swing.*;

public class Win3 {
    public static void main(String[] args) {
        JFrame f = new JFrame("Teste Swing 3");
        f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        f.setSize(400, 150);
        JPanel content = new JPanel();
        content.setBackground(Color.white);
        content.setLayout(new FlowLayout());
        JButton b1 = new JButton("Button 1");
        JButton b2 = new JButton("Button 2");
        content.add(b1);
        content.add(b2);
        f.setContentPane(content);
        f.setVisible(true);
    }
}
```



12

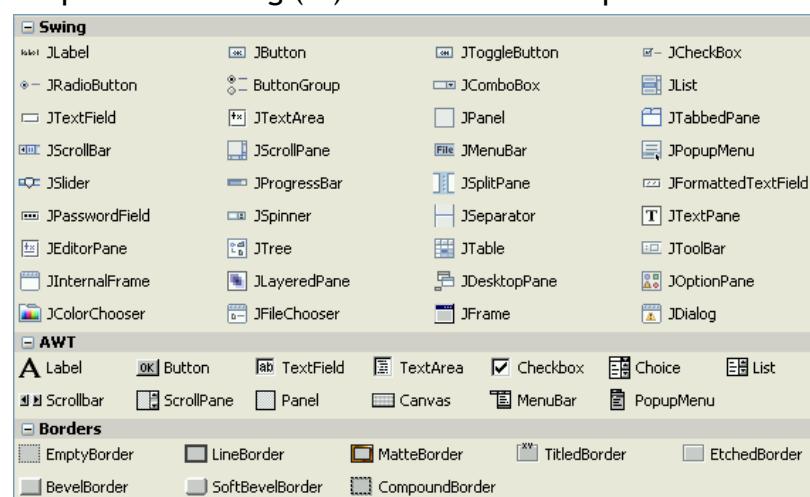
## JFrame e JPanel



13

## Componentes

- Com exceção dos componentes de topo, todos os outros componentes Swing (J\*) derivam de JComponent



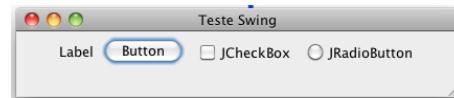
14

## Componentes - Exemplos

- **JLabel**
  - Usado para colocar texto num container. Serve essencialmente de rótulo a outros componentes
 

```
JLabel();
JLabel(String texto);
void setText(String novoTexto);
```
- **JButton**
  - Cria botões. Sempre que um botão é pressionado gera um evento.
 

```
JButton();
JButton(String texto);
void setText(String novoTexto);
void setMnemonic(char c);
```
- **JCheckBox, JRadioButton**
  - ```
JCheckBox();
JCheckBox(String texto);
JCheckBox(String texto, boolean state);
boolean isSelected();
void setSelected(boolean state);
```



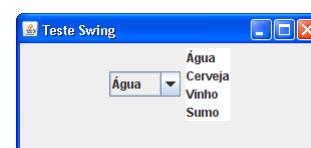
15

## Componentes - Exemplos

- **JComboBox**
  - usado para seleccionar uma opção de uma lista apresentada na forma de pop-up
 

```
JComboBox();
JComboBox(Object[] itens);
void addItem( Object item );
Object getSelectedItem();
int getSelectedIndex();
void setEditable( boolean edit );
void setSelectedIndex( int index );
```
- **JList**
  - usado para fazer selecções de uma lista de items.
  - O utilizador observa uma janela de opções e pode seleccionar vários itens.
 

```
JList();
JList( Object [] itens );
int setListData(Object [] itens );
void setSelectionMode( int mode );
Object getSelectedValue();
Object [] getSelectedValues();
```



16

## Componentes - Exemplos

- JTextField

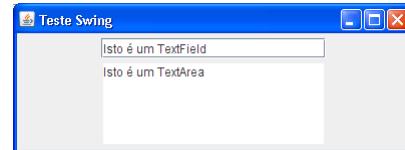
- Apresenta uma linha de texto
 

```
JTextField();
JTextField( int cols );
JTextField( String text, int cols );
String getText();
boolean isEditable();
void setEditable( boolean editable );
void setText( String text );
```

- JTextArea

- Apresenta uma área com múltiplas linhas de texto
- semelhante a JTextField

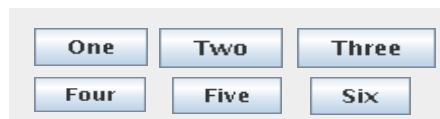
```
int getRows();
void setRows( int rows );
```



17

## Organização de componentes - Layout

- Como dispor os diversos componentes num painel?
  - Podemos definir as localizações manualmente
  - ou ... usar java.awt.LayoutManagers!
- Cada painel contém um objecto que implementa a interface LayoutManager
  - Permite organizar os componentes automaticamente
  - Por omissão é java.awt.FlowLayout, que arruma os componentes da esq. para a dir. e de cima para baixo
- Podemos passar um LayoutManager no construtor do painel ou invocar o método setLayout

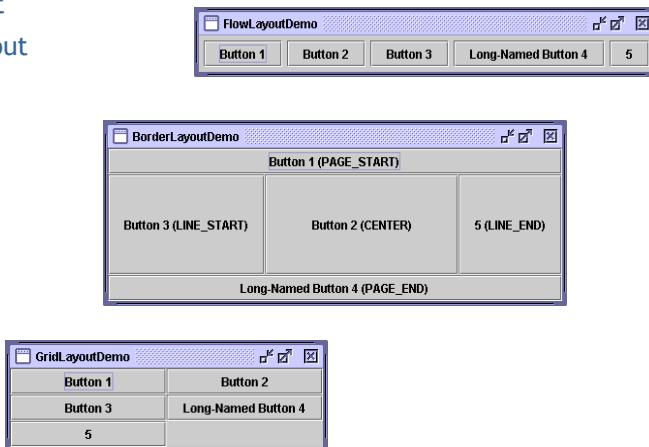


18

## Layout Managers

- Existem diversos tipos de layout. Exemplos:

- FlowLayout
- BorderLayout
- GridLayout



19

## Border layout

- O *BorderLayout* divide o painel em 5 regiões
  - NORTH, SOUTH, EAST, WEST, CENTER



- Na adição de um componente ao contentor é necessário indicar a região

```
panel.add(dp, BorderLayout.CENTER);
```

20

## Border Layout

- Não é necessário preencher todas as áreas
  - Cada região tem dimensão (0, 0) por omissão
  - O BorderLayout ajusta automaticamente os componentes
- Exemplos:

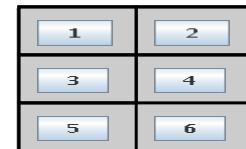


21

## Grid Layout

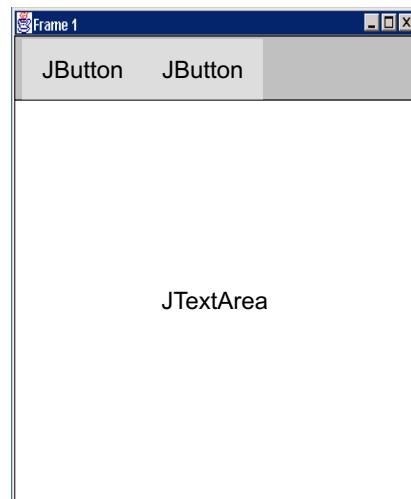
- Dispõe os componentes segundo uma grelha de linhas e colunas
- Redimensiona cada componente de forma a que todos tenham a mesma dimensão
  - igual ao elemento maior
- Os componentes são adicionados linha a linha da esquerda para a direita (e de cima para baixo)
- Exemplo:

```
JPanel gpanel = new JPanel();
gpanel.setLayout(new GridLayout(3, 2));
gpanel.add(new JButton("1"));
gpanel.add(new JButton("2"));
gpanel.add(new JButton("3"));
gpanel.add(new JButton("4"));
gpanel.add(new JButton("5"));
gpanel.add(new JButton("6"));
```



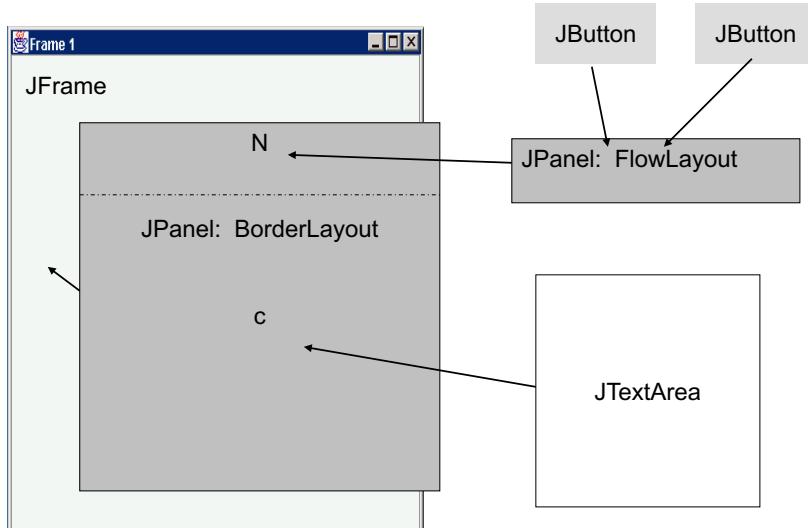
22

## Planificação da interface



23

## Planificação da interface



24

## Programação por Eventos

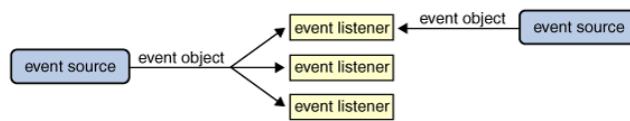
- Já vimos como construir janelas, painéis, componentes, ...
  - mas precisamos ainda de saber como lidar com eventos (selecção de um menu, premir de um botão, arrasto do rato, ...)

25

## Eventos

- O que são?
  - Objectos gerados quando uma determinada acção ocorre (teclado, rato, etc.) e que descrevem o que sucedeu
  - Existem vários tipos de classes de eventos para tratar as diversas categorias de acções desencadeadas pelo utilizador
  - Subclasses de `java.util.EventObject`
- Quem gera o evento?
  - Componentes (**event source**).
  - Todo o evento tem associado um objecto fonte (quem gerou)
 

```
Object fonte = evento.getSource();
```

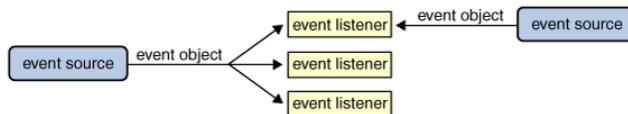


26

## Eventos

- Quem recebe e trata o evento?
  - Objectos receptores/ouvintes (Listeners ou Adapters)
  - Qualquer objecto pode ser notificado de um evento
- Os métodos dos ouvintes (listeners) que desejam tratar eventos, recebem eventos como argumento
 

```
public void eventoOcorreu(EventObject evento) {
    System.out.println(evento+ " em " +evento.getSource());
}
```
- Os ouvintes precisam ser registrados nas fontes
  - Quando ocorre um evento, um método de todos os ouvintes registrados é chamado e o evento é passado como argumento



27

```

import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

public class Win6 extends JFrame {
    public Win6() {
        super("Teste Swing");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setSize(400, 150);
        Container content = getContentPane();
        content.setLayout(new FlowLayout());
        JButton b1 = new JButton("Button 1");
        JButton b2 = new JButton("Button 2");
        content.add(b1);
        content.add(b2);
        b1.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                JOptionPane.showMessageDialog(content,
                    "BUTTON 1 PRESSED!");
            }
        });
        setVisible(true);
    }
    public static void main(String[] args) {
        new Win6();
    }
}

```

## Exemplo



28

```

import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

public class Win6 extends JFrame {
    public Win6() {
        super("Teste Swing");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setSize(400, 150);
        Container content = getContentPane();
        content.setLayout(new FlowLayout());
        JButton b1 = new JButton("Button 1");
        JButton b2 = new JButton("Button 2");
        content.add(b1);
        content.add(b2);
        b1.addActionListener(e ->
            JOptionPane.showMessageDialog(content,
                "BUTTON 1 PRESSED!")
        );
        setVisible(true);
    }
    public static void main(String[] args) {
        new Win6();
    }
}

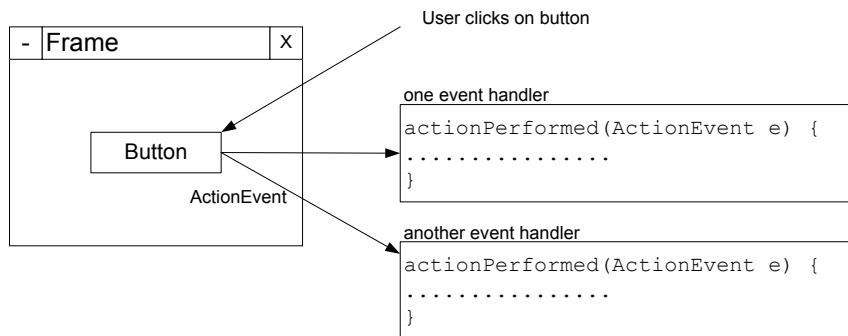
```

## Exemplo (c/ lambda exp.)

29

## Múltiplos eventos

- Um evento pode ser enviado para diversos event handlers



## Tipos de Eventos

- `java.awt.event.*`
  - ActionEvent
  - AdjustmentEvent
  - ComponentEvent
  - ContainerEvent
  - FocusEvent
  - InputEvent
  - InputMethodEvent
  - InvocationEvent
  - ItemEvent
  - KeyEvent
  - MouseEvent
  - MouseWheelEvent
  - PaintEvent
  - TextEvent

31

## Tipos de Receptores (EventListener)

- `java.awt.event.*`
  - ActionListener
  - ContainerListener
  - FocusListener
  - InputMethodListener
  - ItemListener
  - KeyListener
  - MouseListener
  - MouseMotionListener
  - MouseWheelListener
  - TextListener
  - WindowFocusListener
  - WindowListener

Geralmente uma interface `EventListener` (+ do que um método) tem associada uma classe `Adapter` que fornece uma implementação vazia ({} ) dos seus métodos

32

## Tipos de Receptores (EventListener)

Interface Listener	Classe Adapter	Métodos
ActionListener	-----	actionPerformed(ActionEvent)
AdjustmentListener	-----	adjustmentValueChanged(adjustmentEvent)
ComponentListener	ComponentAdapter	componentHidden(ComponentEvent) componentMoved(ComponentEvent) componentResized(ComponentEvent) componentShown(ComponentEvent)
ContainerListener	ContainerAdapter	componentAdded(ContainerEvent) componentRemoved(ContainerEvent)
FocusListener	FocusAdapter	focusGained(FocusEvent) focusLost(FocusEvent)
ItemListener	-----	itemStateChanged(ItemEvent)
KeyListener	KeyAdapter	keyPressed(KeyEvent) keyReleased(KeyEvent) keyTyped(KeyEvent)
MouseListener	MouseAdapter	mouseClicked(MouseEvent) mouseEntered(MouseEvent) mouseExited(MouseEvent) mousePressed(MouseEvent) mouseReleased(MouseEvent)

33

## Tipos de Receptores (EventListener)

Interface Listener	Classe Adapter	Métodos
MouseMotionListener	MouseMotionAdapter	mouseDragged (MouseEvent) mouseMoved(MouseEvent)
TextListener	-----	textValueChange(TextEvent)
WindowListener	WindowAdapter	windowActivated(WindowEvent) windowClosed(WindowEvent) windowClosing(WindowEvent) windowDeactivated(WindowEvent) windowDeiconified(WindowEvent) windowIconified(WindowEvent) windowOpened(WindowEvent)

34

## Formas alternativas de implementação

1. A **própria classe** do objecto que gera o evento implementa a interface *Listener* ou estende a classe *Adapter* do evento.

```
public class MinhaClasse implements WindowListener {...}
ou
public class MinhaClasse extends WindowAdapter {...}
```

2. Construir uma **classe externa** que implemente a interface ou estenda a classe *Adapter* do evento

```
public class MinhaClasse { ... }
public class EventoExt implements WindowListener {...}
ou
public class EventoExt extends WindowAdapter {...}
```

35

## Formas alternativas de implementação

3. Construir uma **classe membro interna** que implemente a interface ou estenda a classe *Adapter* respectiva

```
public class MinhaClasse {
    class EventoInt implements WindowListener {...}
    ou
    class EventoInt extends WindowAdapter {...}
}
```

4. Construir uma **classe anónima interna** que implemente a interface ou a classe *Adapter*

```
public class MinhaClasse {
    janela.addWindowListener( new WindowAdapter() {
        public void windowClosing(WindowEvent e) {
            System.exit(0);
        }
    });
}
```

36

## Como escrever um ActionListener

- São os eventos mais fáceis e os mais usados
- Implementa-se um ActionListener para responder a uma intervenção do utilizador
  - premir de um botão
  - duplo click numa lista
  - escolha de um menu
  - retorno num campo de texto
- O resultado é o envio de uma mensagem “actionPerformed” a todos os ActionListeners que estão registrados no componente fonte.

37

**ActionListener - Exemplo**

```
public class ActionListenerDemo extends JFrame {
    private JMenuBar jmbBarraMenu;
    private JMenu jmopcoes;
    private JMenuItem jmiSair;
    private JButton bArea, bPerimetro;
    private JPanel p1;
    private JTextField jtfBase, jtfAltura, jtfResultado;
    private JLabel jlBase, jlAltura, jlResultado;

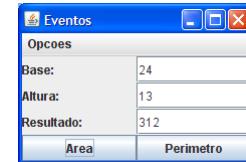
    public ActionListenerDemo() {
        jmbBarraMenu=new JMenuBar();
        jmopcoes=new JMenu("Opcoes");
        jmiSair=new JMenuItem("Sair");
        jmopcoes.add(jmiSair);
        jmbBarraMenu.add(jmopcoes);
        setJMenuBar(jmbBarraMenu);
        p1 = new JPanel(new GridLayout(4,2));
        jlBase = new JLabel("Base: ");
        jlAltura = new JLabel("Altura: ");
        jlResultado = new JLabel("Resultado: ");
        jtfBase = new JTextField(10);
        jtfAltura = new JTextField(10);
        jtfResultado = new JTextField(10);
        bArea = new JButton("Area");
        bPerimetro = new JButton("Perimetro");
        p1.add(jlBase); p1.add(jtfBase); p1.add(jlAltura); p1.add(jtfAltura);
        p1.add(jlResultado); p1.add(jtfResultado); p1.add(bArea); p1.add(bPerimetro);
        getContentPane().add(p1);
    }
}
```



38

## ActionListener - Exemplo

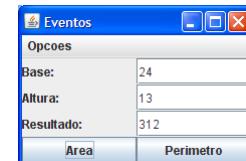
```
jtfBase.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        jtfBase.transferFocus();
    }
});
bArea.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        jtfResultado.setText(String.valueOf(
            Integer.parseInt(jtfBase.getText()) * Integer.parseInt(jtfAltura.getText())));
    }
});
bPerimetro.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        jtfResultado.setText(String.valueOf(2 * (Integer.parseInt(jtfBase.getText()) + 2
            * (Integer.parseInt(jtfAltura.getText())))));
    }
});
jmiSair.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        System.exit(0);
    }
});
})
}
```



39

## ActionListener - Exemplo

Com expressões lambda



```
jtfBase.addActionListener(e -> jtfBase.transferFocus());
bArea.addActionListener( e ->
    jtfResultado.setText(String.valueOf(
        Integer.parseInt(jtfBase.getText())
        * Integer.parseInt(jtfAltura.getText())))
);
bPerimetro.addActionListener(e -> {
    jtfResultado.setText(String.valueOf(
        2 * (Integer.parseInt(jtfBase.getText()) +
        2 * (Integer.parseInt(jtfAltura.getText())))));
}
);
jmiSair.addActionListener( e -> System.exit(0) );
```

40

## Eventos associado ao Rato

### A interface MouseListener

```
package java.awt.event;
public interface MouseListener {
    public void mouseClicked(MouseEvent event);
    public void mouseEntered(MouseEvent event);
    public void mouseExited(MouseEvent event);
    public void mousePressed(MouseEvent event);
    public void mouseReleased(MouseEvent event);
}
```



```
public class ML implements MouseListener {
    public void mouseClicked(MouseEvent e) {}
    public void mouseEntered(MouseEvent e) {}
    public void mouseExited(MouseEvent e) {}
    public void mousePressed(MouseEvent e) {
        System.out.println("Click!");
    }
    public void mouseReleased(MouseEvent e) {}
}
```

## Usando MouseListener

- Utilização

```
// Dado um panel qualquer
MyPanel panel = new MyPanel();
panel.addMouseListener(new MyMouseListener());
```

- Problemas:

- Temos que implementar toda a interface
- No entanto, podemos querer usar um único método, como no exemplo.

## MouseAdapter

- É uma classe abstrata com implementações vazias de todos os métodos da interface MouseListener
- Para usar, basta extender a classe MouseAdapter e reescrever os métodos que interessam.
- Evita a necessidade de implementar todos os métodos vazios que não nos interessam.
- Exemplo:

```
public class MyMouseAdapter extends MouseAdapter {  
    public void mousePressed(MouseEvent event) {  
        System.out.println("User pressed mouse button!");  
    }  
    // usando a classe que definimos (MyMouseAdapter)  
    MyPanel panel = new MyPanel();  
    panel.addMouseListener(new MyMouseAdapter());
```

## Objetos MouseEvent

- Todos os métodos de **MouseAdapter** recebem um parâmetro do tipo **MouseEvent**.
  - É uma classe pré-definida que contém informações sobre o evento que foi gerado.
- Constantes em **InputEvent** (classe base de MouseEvent)
  - public static int BUTTON1\_MASK,BUTTON2\_MASK,  
BUTTON3\_MASK,CTRL\_MASK, ALT\_MASK, SHIFT\_MASK

## Objectos MouseEvent

- Alguns métodos em MouseEvent

- public int getClickCount()
- public Point getPoint()
- public int getX(), getY()
- public Object getSource()
- public int getModifiers()

- Exemplo de uso:

```
public class MyMouseAdapter extends MouseAdapter {
    public void mousePressed(MouseEvent event) {
        Point p = event.getPoint();
        Object source = event.getSource();
        if (source == myPanel && p.getX() < 10)
            JOptionPane.showMessageDialog(null, "Lado esquerdo!");
    }
}
```

## Detecção de movimento do rato

### MouseMotionListener

```
package java.awt.event;
public interface MouseMotionListener {
    public void mouseDragged(MouseEvent event);
    public void mouseMoved(MouseEvent event);
}
```

- A classe abstrata MouseMotionAdapter fornece uma implementação vazia de ambos os métodos
  - Mesma ideia da classe MouseAdapter com a interface MouseListener

## Exemplo MouseMotionAdapter

```
public class MyAdapter extends MouseMotionAdapter {  
    public void mouseMoved(MouseEvent event) {  
        Point p = event.getPoint();  
        int x  = event.getX();  
        int y  = event.getY();  
        System.out.println("Mouse is at " + p);  
        System.out.println("x is " + x);  
        System.out.println("y is " + y);  
    }  
}  
// usando o método  
myPanel.addMouseMotionListener(new MyAdapter());
```

## MouseListener

- A interface **MouseListener** extende tanto a interface **MouseListener** quanto a interface **MouseMotionListener**
- Código:

```
package javax.swing.event;  
public interface MouseInputListener extends  
    MouseListener, MouseMotionListener {}
```
- Como nos casos anteriores, existe uma classe Adapter que implementa versões vazias de todos os métodos desta interface.
  - **MouseInputAdapter**

## Exemplo de MouseInputAdapter

```
public class MyMouseInputAdapter extends MouseInputAdapter {  
    public void mousePressed(MouseEvent event) {  
        System.out.println("Mouse was pressed");  
    }  
    public void mouseMoved(MouseEvent event) {  
        Point p = event.getPoint();  
        System.out.println("Mouse is at " + p);  
    }  
}  
// using the listener  
MyMouseInputAdapter adapter = new MyMouseInputAdapter();  
myPanel.addMouseListener(adapter);  
myPanel.addMouseMotionListener(adapter);
```

## Eventos de Teclado

- São usados para detectar a atividade no teclado dentro de um componente
  - geralmente um panel
- A partir deles podemos responder com ações apropriadas.



## A interface KeyListener

- A interface **KeyListener** deve ser implementada para detectarmos entradas do teclado.
- Código:
 

```
package java.awt.event;

public interface KeyListener {
    public void keyPressed(KeyEvent event);
    public void keyReleased(KeyEvent event);
    public void keyTyped(KeyEvent event);
}
```
- Como nos casos anteriores, existe uma classe Adapter que implementa versões vazias de todos os métodos desta interface.
  - **KeyAdapter**

## A Classe KeyEvent

- Objetos da classe **KeyEvent** são enviados para os receptores de eventos de teclado.
- **InputEvent**
  - `public static int CTRL_MASK, ALT_MASK, SHIFT_MASK`
- **KeyEvent** (descendente de **InputEvent**)
  - `public static int VK_A .. VK_Z, VK_0 .. VK_9, VK_F1 .. VK_F10, VK_UP, VK_LEFT, .., VK_TAB, VK_SPACE, VK_ENTER, ...` (um para cada tecla)
  - `public char getKeyChar()`
  - `public int getKeyCode()`
  - `public Object getSource()`
  - `public int getModifiers()` (máscaras definidas em **InputEvent**)

## Exemplo de KeyAdapter

```
class PacManKeyListener extends KeyAdapter {
    public void keyPressed(KeyEvent event) {
        char keyChar = event.getKeyChar();
        int keyCode = event.getKeyCode();

        if (keyCode == KeyEvent.VK_RIGHT) {
            pacman.setX(pacman.getX() + 1);
            pacpanel.repaint();
        } else if (keyChar == 'Q')
            System.exit(0);
    }
}

PacPanel panel = new PacPanel();
panel.addKeyListener(new PacKeyListener());
```

Quer dizer: se a tecla → foi pressionada...

## Eventos de Janelas

- Os eventos de janela são tratados por classes que implementem a interface **WindowListener**.
- Definição:
 

```
public interface WindowListener {
    public void windowClosing(WindowEvent e)
    public void windowClosed(WindowEvent e)
    public void windowOpened(WindowEvent e)
    public void windowIconified(WindowEvent e)
    public void windowDeIconified(WindowEvent e)
    public void windowActivated(WindowEvent e)
    public void windowDeactivated(WindowEvent e)
}
```
- Como seria de se esperar, existe uma classe chamada **WindowAdapter** que tem uma implementação vazia de cada um destes métodos.

## Resumo dos eventos em cada componente

Component	Event	Action	Adjustment	Component	Container	Focus	Item	Key	Mouse	Mouse Motion	Text	Window
Button	O		O		O		O	O	O		O	
Canvas		O		O			O	O	O		O	
Checkbox	O			O	O	O		O	O		O	
Choice	O				O		O	O	O		O	
Component	O			O			O	O	O		O	
Container	O	O	O				O	O	O		O	
Dialog	O	O	O				O	O	O		O	
Frame	O	O	O				O	O	O		O	
Label		O	O				O	O	O		O	
List	O		O	O	O	O	O	O	O		O	
MenuItem	O						O	O	O		O	
Panel		O	O	O			O	O	O		O	
Scrollbar	O	O	O				O	O	O		O	
TextArea		O		O			O	O	O		O	
TextField	O		O	O			O	O	O		O	
Window		O	O	O			O	O	O		O	

## Sumário

- Java Foundation Classes é um conjunto muito extenso de classes!
- Não é fácil dominar todas as funcionalidades
- Manter em mente a metáfora de construção de interfaces
  - Containers, componentes e gestão de eventos
- IDEs ajudam ..
- NetBeans
- Eclipse +
  - WindowBuilder
  - JFormDesigner - Swing GUI Designer 5.1
  - Jigloo SWT/Swing GUI Builder

# Java

## Introdução à Programação por Padrões

Programação III  
José Luis Oliveira; Carlos Costa

1

### O que é um padrão de software?

- Metodologia testada e documentada para alcançar um objectivo
  - Padrões são comuns em várias áreas da engenharia
- Design Patterns, ou Padrões de Software
  - Padrões para alcançar objectivos na engenharia de software usando classes e métodos em linguagens orientadas por objectos
  - "Design Patterns" de Erich Gamma, John Vlissides, Ralph Jonhson e Richard Helm, conhecidos como "The Gang of Four", ou GoF, descreve 23 padrões.

2

## Porquê padrões?

- Aprender com a **experiência** dos outros
  - **Identificar** problemas comuns em engenharia de software e utilizar **soluções testadas** e bem documentadas
  - Utilizar soluções que têm um **nome**: facilita a comunicação, compreensão e documentação
  - Conhecendo os padrões torna-se mais fácil a compreensão de sistemas existentes
- Desenvolver software de melhor **qualidade**
  - Os padrões utilizam eficientemente polimorfismo, herança, modularidade, composição, abstracção para construir código reutilizável, eficiente, de alta coesão e baixo acoplamento
- Usar **metáforas comuns** a diferentes linguagens e frameworks de desenvolvimento
  - Faz o sistema ficar menos complexo ao permitir que se fale num nível mais alto de abstracção

3

## Formas de classificação

- Há várias formas de classificar os padrões. GoF classifica-os de duas formas:
  - Por propósito: (1) **criação** de classes e objectos, (2) alteração da **estrutura** de um programa, (3) controlo do seu **comportamento**
  - Por alcance: **classe** ou **objecto**
- Metsker classifica-os em 5 grupos, por intenção (problema a ser solucionado):
  - (1) oferecer uma interface,
  - (2) atribuir uma responsabilidade,
  - (3) realizar a construção de classes ou objectos
  - (4) controlar formas de operação
  - (5) implementar uma extensão para a aplicação

4

## Classificação dos 23 padrões segundo GoF

Propósito			
	1. Criação	2. Estrutura	3. Comportamento
Classe	Factory Method	Class Adapter	Interpreter Template Method
Objeto	Abstract Factory Builder Prototype Singleton	Object Adapter Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

5

## Classificação dos padrões segundo Metsker

Intenção	Padrões
1. Interfaces	Adapter, Facade, Composite, Bridge
2. Responsabilidade	Singleton, Observer, Mediator, Proxy, Chain of Responsibility, Flyweight
3. Construção	Builder, Factory Method, Abstract Factory, Prototype, Memento
4. Operações	Template Method, State, Strategy, Command, Interpreter
5. Extensões	Decorator, Iterator, Visitor

6

## Alguns padrões ...

- Singleton
  - Garantir que uma classe só tenha uma única instância fornecendo um ponto de acesso global
- Decorator
  - Anexar responsabilidades adicionais a um objecto dinamicamente
- Iterator
  - Fornecer uma maneira de aceder sequencialmente a elementos de uma coleção sem expor a sua representação interna
- Factory
  - Definir uma interface para criar um objecto mas deixar que subclasses decidam que classe instanciar

7

## Singleton

- Problema
  - "Garantir que uma classe só tenha uma única instância, acedida a partir de um ponto de acesso global." [GoF]
- Aplicações
  - Driver de acesso a uma base de dados
  - Sistema de Ficheiros
  - Ficheiro de Log



- 1) Lock-up a class so that clients cannot create their own instances, but must use the single instance hosted by the class itself.

<http://www.vincehuston.org/dp/singleton.html>

8

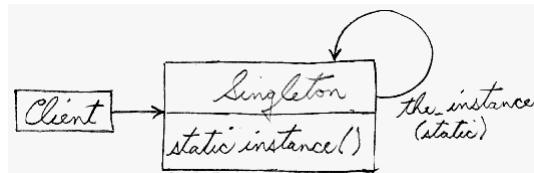
## Questões?

- Como garantir que só existe um objecto da classe?
- Como controlar ou impedir a construção normal?
- Como definir o acesso a uma única instância?
- Como garantir que o sistema funciona se a classe participar numa hierarquia de classes?

9

## Singleton Pattern

Singleton
-instance : Singleton
-Singleton() +getInstance() : Singleton



10

## Singleton em Java

```

class Singleton {
    private String name;

    static private Singleton instance =
        new Singleton("Ermita");
    private Singleton(String name) {
        this.name = name;
    }
    static public Singleton getInstance() {
        return instance;
    }
}
Singleton errado = new Xingleton("Coelho");
Singleton um = Singleton.getInstance();
Singleton dois = Singleton.getInstance();
Singleton tres = Singleton.getInstance();
System.out.printf("%s\n%s\n%s\n", um, dois, tres);
    
```

Singleton@1e0be38  
Singleton@1e0be38  
Singleton@1e0be38

11

## Singleton: Princípios

- Definir o construtor como privado (ou protected)  
`private Singleton(String name)`
- Definir uma referência estática privada para apontar para o único objecto da classe  
`static private Singleton instance`
- Definir um método de acesso a essa instância  
`static public Singleton getInstance()`
  - Os clientes poderão aceder ao objecto através deste (ou de outros) métodos

12

## Decorador (*Decorator*)

- Problema

- "Anexar responsabilidades adicionais a um objecto dinamicamente. Os Decoradores oferecem uma alternativa flexível ao uso de herança para estender uma funcionalidade." [GOF]

- Aplicações

- Sistema de I/O em Java
  - Substituição de herança (múltipla e complexa)

13

## Questões?

- Considere as seguintes entidades:

- Futebolista (joga, passa, remata),
  - Tenista (joga, serve),
  - Jogador (joga)
  - ... Quais as relações entre estas classes?

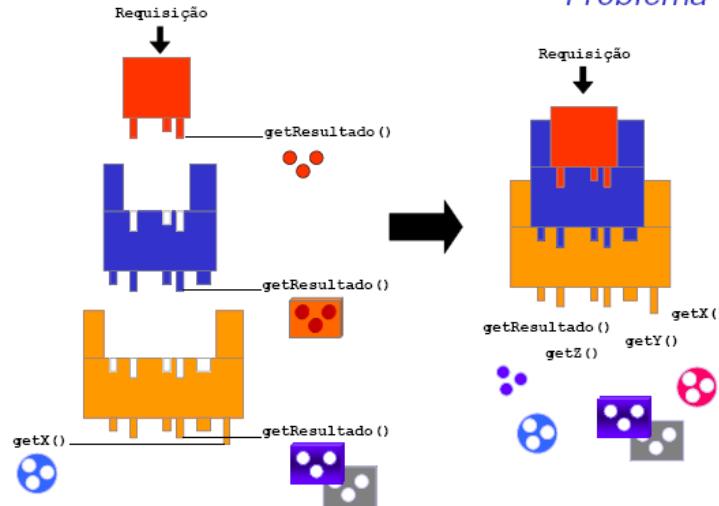
- Vamos complicar:

- O Rui joga Basquete e Futebol
  - A Ana joga Badminton e Basquete
  - O Paulo joga Xadrez, Futebol e Basquete
  - .... Solução?

14

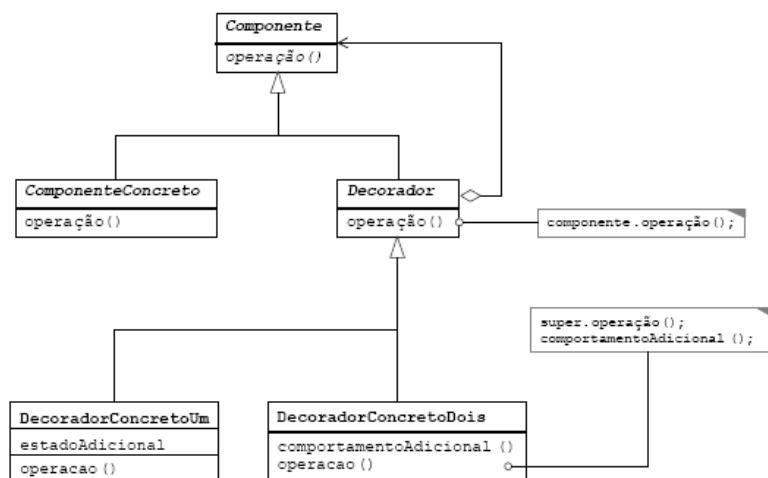
## Decorador

### Problema



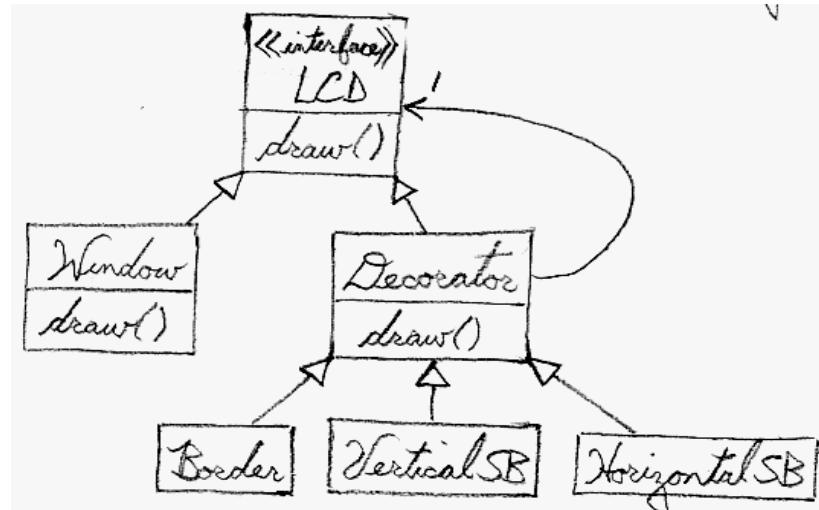
15

## Modelo de um Decorador



16

## Modelo de um Decorador - Exemplo



17

## Exemplo de Decorador

```

interface JogadorInterface {
    void joga();
}

class Jogador implements JogadorInterface {
    private String name;
    Jogador(String n) { name = n; }
    @Override public void joga()
        { System.out.print("\n"+name+" joga ");}
}

abstract class JogDecorator implements JogadorInterface {
    protected JogadorInterface j;
    JogDecorator(JogadorInterface j) { this.j = j; }
    public void joga() { j.joga(); }
}

```

18

## Exemplo de Decorador

```

class Futebolista extends JogDecorator {
    Futebolista(JogadorInterface j) { super(j); }
    @Override public void joga()
        { j.joga(); System.out.print("futebol "); }
    public void remata() { System.out.println("-- Remata!"); }
}

class Xadrezista extends JogDecorator {
    Xadrezista(JogadorInterface j) { super(j); }
    @Override public void joga() { j.joga();
        System.out.print("xadrez "); }
}

class Tenista extends JogDecorator {
    Tenista(JogadorInterface j) { super(j); }
    @Override public void joga()
        { j.joga(); System.out.print("tenis "); }
    public void serve() { System.out.println("-- Serve!"); }
}

```

19

## Exemplo de Decorador

```

public class Composition {
    public static void main(String args[])
        JogadorInterface j1 = new Jogador("Rui");
        Futebolista f1 = new Futebolista(new Jogador("Luis"));
        Xadrezista x1 = new Xadrezista (new Jogador("Ana"));
        Xadrezista x2 = new Xadrezista (j1);
        Xadrezista x3 = new Xadrezista (f1);
        Tenista t1 = new Tenista(j1);
        Tenista t2 =
            new Tenista(
                new Xadrezista (
                    new Futebolista(
                        new Jogador("Bruna"))));

```

Rui joga  
 Luis joga futebol  
 Ana joga xadrez  
 Rui joga xadrez  
 Luis joga futebol xadrez  
 Rui joga tenis  
 Bruna joga futebol xadrez tenis

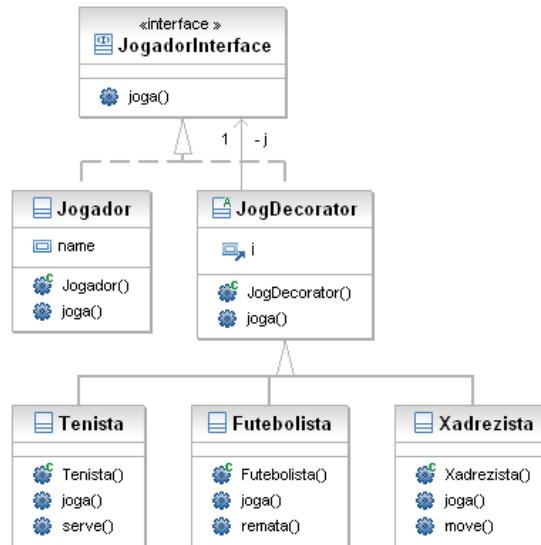
```

        JogadorInterface lista[] = { j1, f1, x1, x2, x3, t1, t2 };
        for (JogadorInterface ji: lista)
            ji.joga();
    }
}

```

20

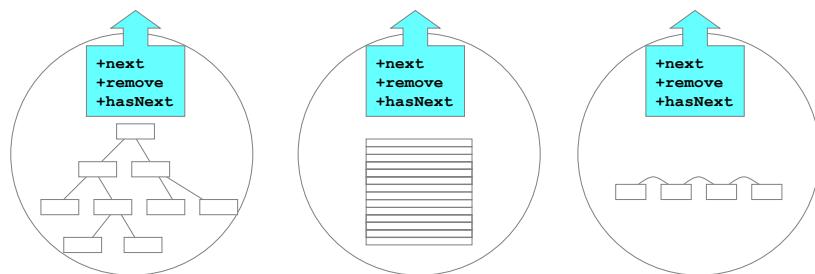
## Exemplo de Decorador



21

## Iterador

- Problema
  - “Fornecer uma forma comum de aceder aos elementos de um objecto agregado sequencialmente sem expôr a sua representação interna.” [GoF]
- Aplicações



22

## Exemplo - Listar colecções

```

// For a set or list
for (Iterator it=collection.iterator(); it.hasNext(); ) {
    Object element = it.next();
}

// For keys of a map
for (Iterator it=map.keySet().iterator(); it.hasNext(); ) {
    Object key = it.next();
}

// For values of a map
for (Iterator it=map.values().iterator(); it.hasNext(); ) {
    Object value = it.next();
}

// For both the keys and values of a map
for (Iterator it=map.entrySet().iterator(); it.hasNext(); ) {
    Map.Entry entry = (Map.Entry)it.next();
    Object key = entry.getKey();
    Object value = entry.getValue();
}

```

23

## Operações

### Colecção

- Construir um iterador - uma Fábrica devolve um iterador que aponta para o primeiro elemento do conjunto
  - `Collection.iterator()`

### Iterador

- Devolver a referência para o próximo elemento
  - `next();`
- Determinar se ainda existem mais elementos na sequência
  - `boolean hasNext();`
- Remover o elemento da colecção apontado pelo iterador
  - `remove();`

24

## Iteradores em Java

- Os iteradores são implementados nas colecções de Java
- Um iterador pode ser obtido através do método `Collection.iterator()` que devolve uma instância de `java.util.Iterator`.

- Interface `java.util.Iterator`:

```

package java.util;

public interface Iterator {
    boolean hasNext();
    Object next();
    void remove();
}

// Notas:
// Java5 já utiliza tipo de dados parametrizáveis - Iterator<E>
// Java8 tem uma implementação default do método remove()
// Java8 introduz um default method: forEachRemaining(...)
```

25

## Iteradores em Java

- Para implementar um iterador numa colecção, deve usar-se delegação:
  - Incluir um iterador na classe que gere o conjunto e um método `iterator()` ou similar
  - Implemente métodos `next()`, `hasNext()`, etc. extraíndo os dados na colecção.

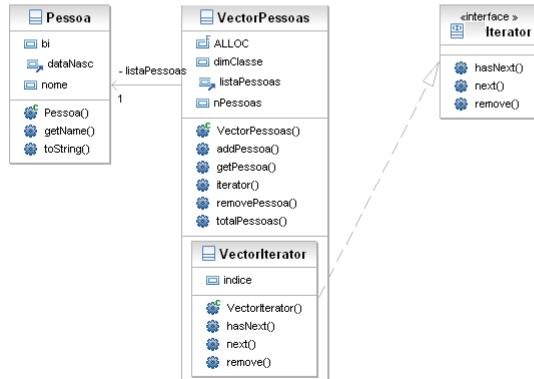
```

VectorPessoas vp = new VectorPessoas();
for (int i=0; i<10; i++)
    vp.addPessoa(new Pessoa("Bebé_"+i, 1000+i, Data.today()));
Iterator vec = vp.iterator();
while ( vec.hasNext() )
    System.out.println( vec.next() );
```

26

## Exemplo: Conjunto de Pessoas

```
public interface Iterator {
    boolean hasNext();
    Object next();
    void remove();
}
```



27

```
public class VectorPessoas {
    private Pessoa[] listaPessoas;
    private int nPessoas;
    private final int ALLOC = 50;
    private int dimClasse = ALLOC;

    public VectorPessoas() {
        listaPessoas = new Pessoa[ALLOC];
        nPessoas = 0;
    }

    public boolean addPessoa(Pessoa est) {
        if (est == null)
            return false;
        if (nPessoas >= dimClasse) {
            dimClasse += ALLOC;
            Pessoa[] newArray = new Pessoa[dimClasse];
            System.arraycopy(listaPessoas, 0, newArray, 0, nPessoas );
            listaPessoas = newArray;
        }
        listaPessoas[nPessoas++] = est;
        return true;
    }
}
```

28

```

public boolean removePessoa(Pessoa p) {
    for (int i = 0; i < nPessoas; i++) {
        if (listaPessoas[i] == p) {
            nPessoas--;
            for (int j = i; j < nPessoas; j++)
                listaPessoas[j] = listaPessoas[j + 1];
            return true;
        }
    }
    return false;
}

public int totalPessoas() {
    return nPessoas;
}

public Pessoa getPessoa(int i) {
    return listaPessoas[i];
}

Iterator iterator() {
    return (this).new VectorIterator();
}

```

29

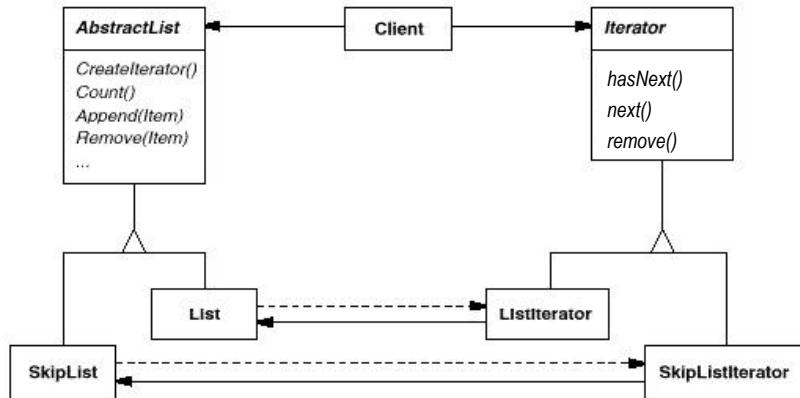
```

private class VectorIterator implements Iterator{
    private int indice;
    VectorIterator() {
        indice = 0;
    }
    public boolean hasNext() {
        return (indice < nPessoas);
    }
    public Object next() {
        if (hasNext()) {
            Object r = listaPessoas[indice];
            indice++;
            return r;
        }
        throw new IndexOutOfBoundsException("only "
            + nPessoas + " elements");
    }
    public void remove() {
        throw new UnsupportedOperationException(
            "Operação não suportada!");
    }
}

```

30

## Iterador genérico



31

## Fábrica (*Factory Method*)

- Problema
  - "Definir uma interface para criar um objeto mas deixar que subclasses decidam que classe instanciar. Factory Method permite que uma classe delegue a responsabilidade de instanciamento às subclasses." [GoF]
- Aplicações
  - Iteradores - Um iterador é um exemplo de Factory Method

32

## Como implementar?

- É possível criar um objecto sem ter conhecimento algum de sua classe concreta?
  - Esse conhecimento deve estar em alguma parte do sistema, mas não precisa estar no cliente
  - **Factory Method** define uma interface comum para criar objectos
  - O objecto específico é determinado nas diferentes implementações dessa interface
- Cria-se uma classe Factory com um método estático que, com base num parâmetro de entrada, decide qual o construtor (privado) a usar
- O objecto criador pode ser seleccionado com base noutras critérios que não requeiram parâmetros

33

## Exemplo

```

interface Arvore {
    void regar();
    void colherFruta();
}

class Figueira implements Arvore {
    protected Figueira() {System.out.println("Figueira plantada."); }
    public void regar() { System.out.println("Figueira: Regar muito pouco"); }
    public void colherFruta() { System.out.println("Hum.. figos!"); }
}

class Pessegueiro implements Arvore {
    protected Pessegueiro() {System.out.println("Pessegueiro plantado."); }
    public void regar() { System.out.println("Pessegueiro: Regar normal"); }
    public void colherFruta() { System.out.println("Boa.. pêssegos!"); }
}

class Nespereira implements Arvore {
    protected Nespereira() {System.out.println("Nespereira plantada."); }
    public void regar() { System.out.println("Nespereira: Regar pouco"); }
    public void colherFruta() { System.out.println("Ahh.. nêsperas!"); }
}

```

34

## Exemplo

```
class Viveiro {
    public static Arvore factory(String pedido) {
        if (pedido.equalsIgnoreCase("Figueira"))
            { return new Figueira(); }
        if (pedido.equalsIgnoreCase("Pessegueiro"))
            { return new Pessegueiro(); }
        if (pedido.equalsIgnoreCase("Nespereira"))
            { return new Nespereira(); }
        else
            throw new IllegalArgumentException("Árvore não existente!");
    }
}
```

35

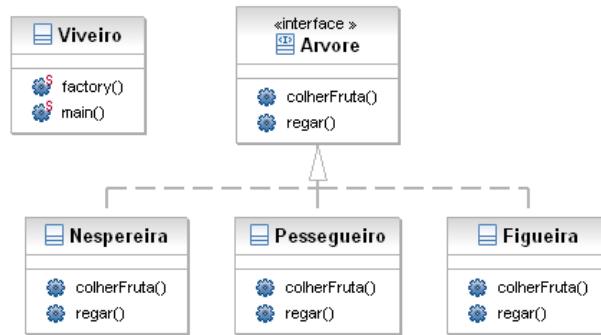
## Exemplo

```
public static void main(String[] args) {
    Arvore pomar[] = {
        Viveiro.factory("Figueira"),
        Viveiro.factory("Pessegueiro"),
        Viveiro.factory("Figueira"),
        Viveiro.factory("Nespereira")
    };
    for (Arvore a: pomar)
        a.regar();
    for (Arvore a: pomar)
        a.colherFruta();
}
```

Figueira plantada.  
 Pessegueiro plantado.  
 Figueira plantada.  
 Nespereira plantada.  
 Figueira: Regar muito pouco  
 Pessegueiro: Regar normal  
 Figueira: Regar muito pouco  
 Nespereira: Regar pouco  
 Hum.. figos!  
 Boa.. pêssegos!  
 Hum.. figos!  
 Ahh.. nêsperas!

36

## Exemplo - Estrutura



37

## Sumário

- Singleton, Iterador, Decorador, Fábrica .. mas há muitos outros padrões de programação
- Tal como os algoritmos ajudam a resolver problemas computacionais os padrões contribuem para desenvolver software mais bem estruturado

38

# Java Tipos Genéricos

Programação III  
José Luis Oliveira; Carlos Costa

1

## Motivações

- Quando os programas aumentam de dimensão é possível começarmos a ter métodos que executam operações similares com diferentes tipos de dados
- O que há de “errado” com o seguinte bloco de código?

```
String add (String a, String b) { return a + b; }
int    add (int a, int b)      { return a + b; }
double add (double a, double b) { return a + b; }

public static void main(String[] args){
    System.out.println(add(Gene, ricos));
    System.out.println(add(2, 5));
    System.out.println(add(3.2, 5.6));
}
```

- Nada?

O código está correcto mas definimos estaticamente operações repetidas.

2

1

## Motivações

```
class Node {  
    Object value;  
    Node next;  
  
    Node( Object o ) {  
        value = o;  
        next = null;  
    }  
}
```

```
Car c = new Car( ... );  
Node n = new Node( c );  
  
...  
  
Vehicle v = ( Vehicle ) n.value;
```

OK

```
Movie m = new UniversalMovie( "ET" );  
Node n = new Node( m );  
  
...  
  
Vehicle v = ( Vehicle ) n.value;
```

Down-Cast e Runtime Error

Run-Time Error

3

## O que são Genéricos?

- Uma forma de Polimorfismo Paramétrico
- Estruturas e Algoritmos são implementados uma única vez, mas utilizados com diferentes tipos de dados
- Dizemos que:
  - Os **Tipos de dados** também são um **Parâmetro**
- Genéricos aplicados a:
  - Métodos
  - Classes
  - Interfaces
- Introduzidos em JAVA na versão 5
- Em C++, designam-se por *templates*

4

## Classes Genéricas

### Exemplo: Conjunto Genérico

#### Declaração

```
class Conjunto<T> {  
    T[] c;  
    //...  
}
```

#### Nota Importante:

O tipo parametrizado (T), não pode ser instanciado com um tipo primitivo.

#### Utilização

```
Conjunto<Pessoa> c1 = new Conjunto<Pessoa>(..);  
Conjunto<Jogador> c2 = new Conjunto<Jogador>(..);  
Conjunto<Integer> c3 = new Conjunto<Integer>(..);
```

5

## Classes Genéricas

### Exemplo: Uma Pilha Genérica

#### Sem Genéricos

```
class Stack {  
    void push(Object o) { ... }  
    Object pop() { ... }  
    ...  
  
    String s = "Hello";  
    Stack st = new Stack();  
    ...  
    st.push(s);  
    ...  
    s = (String) st.pop();
```

↑  
E se estivermos errados?

#### Utilizando Genéricos

```
class Stack<T> {  
    void push(T a) { ... }  
    T pop() { ... }  
    ...  
  
    String s = "Hello";  
    Stack<String> st =  
        new Stack<String>();  
  
    st.push(s);  
    ...  
    s = st.pop();
```

OK

Runtime Error

6

## Genéricos

```
class Node< T > {  
    T value;  
    Node< T > next;  
  
    Node( T t ) {  
        value = t;  
        next = null;  
    }  
}
```

```
Car c = new Car( ... );  
Node<Car> node = new Node<Car>( c );  
  
...  
  
Car c2 = node.value;
```

OK

```
Movie m = new ActionMovie( ... );  
  
...  
  
Node<Car> node = new Node<Car>( m );
```

Compiler Error

Detecção de Erros em  
Compilação

7

## Genéricos - Processamento

- Etapas de processamento de Genéricos em JAVA
  - **Check:** Verificação da correcta utilização de tipos
  - **Erase:** Remove toda a informação “generic type”
  - **Compile:** Geração do byte-code
- Este processo denomina-se como:
  - **Type Erasure**

## Genéricos - Check

- Declaração

```
class Foo <T> {  
    void method(T arg);  
};
```

- Utilização

```
Foo<Bar> fb = new Foo<Bar>;  
fb.method(aBar);           // OK  
fb.method(not_a_Bar);     // Compile Error
```

- O Compilador garante que o *arg* é do mesmo tipo <Bar>

## Genéricos - Erase

- Cada parâmetro definido como genérico é substituído por um `java.lang.Object`
- Os *casts* “Object -> Tipo Concreto” são automaticamente introduzidos pelo compilador.

```
class choice <T>  
{ public T best ( T a , T b ) {..} }
```

É substituído por:

```
class choice  
{ public Object best ( Object a, Object b ) {..} }
```

10

## Genéricos em Classes

```
public class Stack_Generic<T> {
    private class Node<E> {
        E val;
        Node<E> next;
        Node(E v, Node<E> n) {
            val = v;
            next = n;
        }
    }

    private Node<T> top = null;

    public boolean empty() {
        return top == null;
    }

    public T pop() {
        T result = top.val;
        top = top.next;
        return result;
    }

    public void push(T v) {
        top = new Node<T>(v, top);
    }
}

public class TestStack {

    public static Figura randFig(int max) {
        Switch ((int)(Math.random()*(max))) {
            default:
            case 0:
                return new Circulo(1,3, 1.2);
            case 1:
                return new Quadrado(3,4, 2);
            case 2:
                return new Rectangulo(1,1, 5,6);
        }
    }

    public static void main(String[] args) {
        Stack_Generic<Figura> stk =
            new Stack_Generic<Figura>();

        for (int i=0; i<10; i++)
            stk.push(randFig(3));

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

## Genéricos em Métodos

- Declaração

```
public <T> T add (T a, T b) {
    return a + b;
}
```

Compilará?

- Utilização Pretendida

Podemos aplicar o operador + ao tipo Object?

```
int c = add(8, 3);
double d = add(4.5, 6.9);
String str = add("gene", "ricos");
```

Não.

Não.

## Genéricos - Métodos

```
public <T extends Number> T add (T a, T b) {  
    return a + b;  
}
```

- Impusemos um limite superior ao tipo passado para o método genérico
- Neste caso estamos a dizer que o tipo passado será um subtipo de java.lang.Number

• E Agora?

Podemos aplicar o  
operador + ?

E o tipo de Retorno.  
Correcto?

Não.

13

## Genéricos - Métodos

```
public static <T extends Number> Double add (T a, T b) {  
    return a.doubleValue() + b.doubleValue();  
}  
  
public static <T> String sumToString (List<T> aList){  
    StringBuilder aBuffer = new StringBuilder();  
    for (T x : aList)  
        aBuffer.append (x.toString());  
    return aBuffer.toString();  
}  
  
public static void main(String[] args) {  
    System.out.println("add: " + add (2.2, 3));  
    List<Integer> myList = new ArrayList<Integer>();  
    myList.add(22); myList.add(33); myList.add(44);  
    System.out.println("sumToString : " +  
                      sumToString(myList));  
}
```

add: 5.2  
sumToString: 223344

## Genéricos e Subtipos

- Já sabemos que:
  - Uma referência do tipo T pode apontar para uma instância da classe T ou para uma instância de um subtipo de T.
- Como se conjuga **Polimorfismo com Tipos Genéricos?**

```
public static void main(String[] args) {  
    LinkedList<Figura> list = new LinkedList<Figura>();  
    LinkedList<Quadrado> list2 = new LinkedList<Quadrado>();  
    Quadrado q = new Quadrado(3,4, 2);  
  
    list.add(q); // OK  
    list2.add(q);  
  
    LinkedList<Quadrado> list3 = list; // Compile-Time Error  
    LinkedList<Figura> list4 = list2;  
}
```

15

Porquê ?

## Genéricos e Subtipos

```
LinkedList<Quadrado> list = new LinkedList<Quadrado>();  
LinkedList<Figura> list2 = list; // Imaginando que é possível  
  
Figura f = new Figura(..); // Supondo que não é abstract  
  
list2.add(f);  
  
Quadrado q = list.get(0); // Runtime ERROR!!!!
```

Uma Figura não é um Quadrado

Se X é um subtipo de Y, e G um tipo genérico, não é verdade que G<X> é um subtipo de G<Y>

16

## Genéricos e Subtipos

```
public static void main( String[ ] args ) {  
    LinkedList<Quadrado> list = new LinkedList<Quadrado>();  
    Quadrado q = new Quadrado(3,4, 2);  
    list.add(q);  
    list.add(q);  
  
    print(list);  
}  
  
public static void print( List<Figura> listOfFig ) {  
    Iterator it = listOfFig.iterator();  
    while( it.hasNext() )  
        System.out.println( it.next() );  
}
```

Compile-Time Error

**Questão:** Como permitir que, tendo um argumento tipo `LinkedList` de `Figura`, se possa aceitar uma `LinkedList` de `Figura` mas também um dos seus subtipos?

17

## Genéricos - Wildcards

- **Resposta:** “Utilizando Wildcards”
- Bounded wildcards
  - `< ? extends class-T >`  
    subclass de `class-T`, incluindo `class-T`
  - `< ? extends class-T & interface-E >`  
    subclass de `class-T` e `int-E`, incl. `class-T` e `int-E`
  - `< ? super class-T >`  
    superclass de `class-T`, incluindo `class-T`
- Unbounded wildcards
  - `< ? extends Object >`  
    subclass de `Object`, i.e. qualquer tipo
  - `< ? >`  
    Semelhante a `< ? extends Object >`

18

## Genéricos e Subtipos

```
public static void main( String[ ] args ) {
    LinkedList<Quadrado> list = new LinkedList<Quadrado>();
    Quadrado q = new Quadrado(3,4, 2);
    list.add(q);
    list.add(q);

    print(list);
}

public static void print( LinkedList<? extends Figura> listOfFig ) {
    Iterator it = listOfFig.iterator();
    while( it.hasNext() )
        System.out.println( it.next() );
}
```

19

## Genéricos - Wildcards

```
public static void main( String[ ] args ) {
    LinkedList<Quadrado> list = new LinkedList<Quadrado>();
    Quadrado q = new Quadrado(1,2, 5.5);
    list.add(q);
    addMore(list);
```

OK

```
public static void addMore( LinkedList<? extends Figura> listIn ) {
    Rectangulo r = new Rectangulo(3,4, 2, 3);
    Quadrado q = new Quadrado(1,2, 5.5);

    listIn.add(r);
    listIn.add(q);
```

Compile-Time Error

Porquê ?

20

## Genéricos - Wildcards

- É possível ainda especificar um parâmetro tipo genérico que *extend/implement* uma *class/interface*
- Para ambos os casos é utilizada a keyword **extend**

```
public class Xpto <T extends Serializable &
                    Comparable<T>> {...}
```

- Tipo Genérico deve implementar Serializable e Comparable

21

## Genéricos

- **Excepções:** Também podemos passar um parâmetro genérico que é um subtipo de **Exception**

```
public class Employer <T extends Employee,
                     X extends Exception>{
    public double calculatePay (T employee) throws X {};
}
```

- **Arrays:** Não é possível criar um array de tipos genéricos

```
T[] array = new T[MAX];
```



```
Compile-Time Error
```

```
T[] newArray = (T[]) new Object[MAX];
```

22

## Exemplos

```
/**  
 * This version introduces a generic method.  
 */  
public class Box<T> {  
    private T t;  
    public void add(T t) {  
        this.t = t;  
    }  
    public T get() {  
        return t;  
    }  
    public <U> void inspect(U u){  
        System.out.println("T: " + t.getClass().getName());  
        System.out.println("U: " + u.getClass().getName());  
    }  
    public static void main(String[] args) {  
        Box<Integer> integerBox = new Box<Integer>();  
        integerBox.add(new Integer(10));  
        integerBox.inspect("some text");  
    }  
}
```

T: java.lang.Integer  
U: java.lang.String

23

## Exemplos

```
public class Box<T> {  
    ...  
    public static <U> void fillBoxes(U u, List<Box<U>> boxes) {  
        for (Box<U> box : boxes) {  
            box.add(u);  
        }  
    }  
    ...  
}  
  
Crayon red = ...;  
List<Box<Crayon>> crayonBoxes = ...;  
  
Box.fillBoxes(red, crayonBoxes);
```

24

## Java 7 - The Diamond

- In Java SE 7 and later, you can replace the type arguments required to invoke the constructor of a generic class with an empty set (<>) as long as the compiler can determine the type arguments from the context.
  - This pair of angle brackets, <>, is informally called the *diamond*.
- Exemplo:

```
Box<Integer> integerBox = new Box<>();
```
- For example, consider the following variable declaration:

```
Map<String, List<String>> myMap =
    new HashMap<String, List<String>>();
```
- In Java SE 7 and later, you can substitute the parameterized type of the constructor with an empty set of type parameters (<>):

```
Map<String, List<String>> myMap = new HashMap<>();
```

25

## Type Parameter Naming Conventions

- By convention, type parameter names are single, uppercase letters.
  - This stands in sharp contrast to the variable naming conventions that you already know about, and with good reason:
  - Without this convention, it would be difficult to tell the difference between a type variable and an ordinary class or interface name.
- The most commonly used type parameter names are:
  - E - Element (used extensively by the Java Collections Framework)
  - K - Key
  - N - Number
  - T - Type
  - V - Value

26

## Tipos genéricos em Java - Resumo

- Conseguimos:
  - eliminar necessidade de coerção explícita (cast)
  - aumentar robustez: verificação estática de tipo
  - aumentar legibilidade
- Não há múltiplas versões do código
  - declaração é compilada para todos os tipos
  - parâmetros formais possuem tipo genérico
  - na invocação, os tipos dos parâmetros actuais são substituídos pelos tipos dos formais

27

## Sumário

- Tipos Genéricos
- Motivações
- Processamento
- Wildcards e Sub-tipos
- Classe e Métodos Genéricos

28

## Colecções

Programação III  
José Luis Oliveira; Carlos Costa

1

## Colecções

- **Collection:** Interface de JAVA que determina o comportamento que uma coleção deve ter.
- Introduzidas no Java 1.2 com a denominação de “**JAVA Collections FrameWork (JCF)**”.
- São estruturas de dados com propriedades próprias que permitem agregar objectos de determinado tipo.
- Também são conhecidas como “containers”
- Não suporta tipos primitivos (*int*, *float*, *double*,...)
  - Utilizar Wrapper’s (*Integer*, *Float*, *Double*, ...)

2

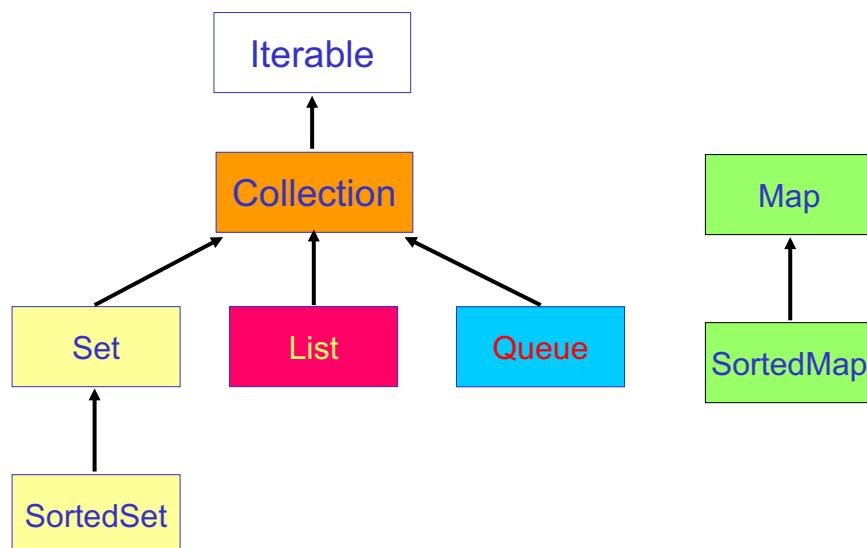
## Principais Interfaces

### Java Collections Framework (JCF):

- Conjunto de classes, interfaces e algoritmos que representam vários tipos de estruturas de armazenamento de dados.
- Conjunto de 4 Interfaces Principais:
  - **Conjuntos (Set)**: sem noção de posição (sem ordem), sem repetição
  - **Listas (List)**: sequências com noção de ordem, com repetição
  - **Filas (Queue)**: são as filas do tipo First in First Out
  - **Mapas (Map)**: estruturas associativas onde os objectos são representados por um par **chave-valor**. Pares chave-valor (com repetição - MultiMap)

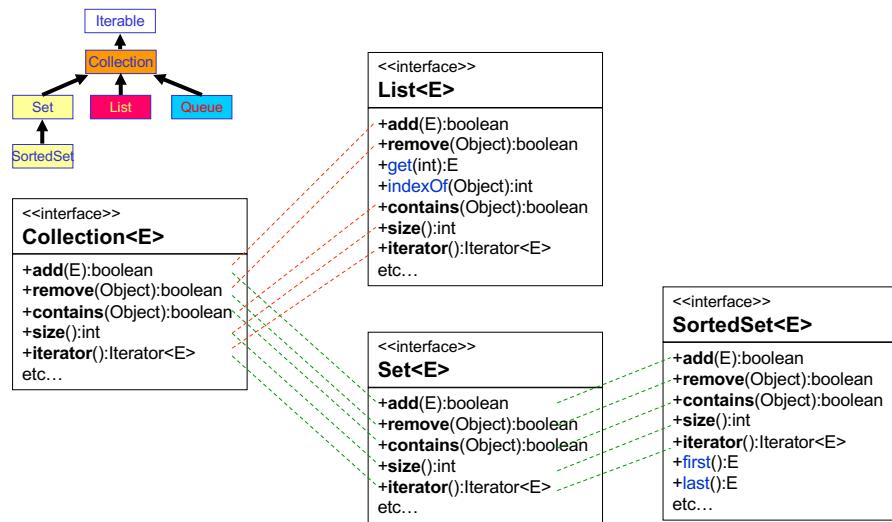
3

## Hierarquia de interfaces

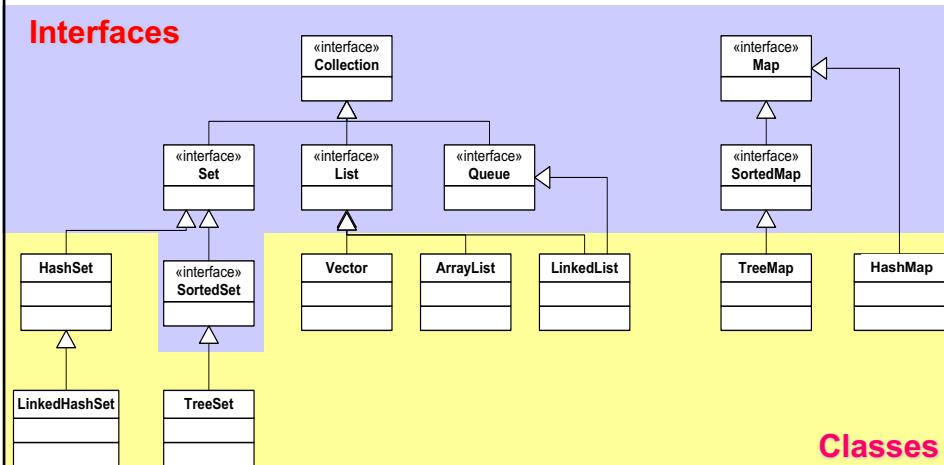


4

## Expansão de contratos



## Hierarquia de Classes



## Interfaces e Implementações

Collections					
Interfaces	Implementações				
	Hash table	Resizable array	Balanced Tree (sorted)	Linked list	Hash table + Linked list
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Queue		ArrayDeque		LinkedList	
Map	HashMap		TreeMap		LinkedHashMap

7

## Vantagens das Collections

- Vantagem de criar interfaces:
  - Separa-se a especificação da implementação
  - Pode-se substituir uma implementação por outra mais eficiente sem grandes impactos na estrutura existente.
- Exemplo:

```
Collection<String> c = new LinkedList<String>();
c.add("Aveiro");
c.add("Paris");
Iterator<String> i = c.iterator();
while (i.hasNext()) {
    System.out.println(i.next());
}
```

8

## Genéricos em Collections

Desde o JAVA 5 que as Collections são parametrizáveis

### Antes..

```
LinkedList lista =  
    new LinkedList();  
  
lista.add(new Data(...));  
lista.add(new Pessoa(...));  
  
Iterator i = lista.iterator();  
  
Data d = (Data)i.next();  
Pessoa p = (Pessoa)i.next();
```

### Agora..

```
LinkedList<Data> lista =  
    new LinkedList<Data>();  
  
lista.add(new Data(...));  
// lista.add(new Pessoa(...));  
  
Iterator<Data> i =  
    lista.iterator();  
  
Data d = i.next();  
//Pessoa p = (Pessoa)i.next();
```

Compile-Time Error

Compile-Time Error

9

## Interface Collection

```
public interface Collection<E> extends Iterable<E> {  
    // Basic operations  
    int size();  
    boolean isEmpty();  
    boolean contains(Object element);  
    boolean add(E element);  
    //optional  
    boolean remove(Object element);  
    //optional  
    Iterator<E> iterator();  
  
    // Bulk operations  
    boolean containsAll(Collection<?> c);  
    boolean addAll(Collection<? extends E> c);  
    //optional  
    boolean removeAll(Collection<?> c);  
    //optional  
    boolean retainAll(Collection<?> c);  
    //optional  
    void clear();  
    //optional  
  
    // Array operations  
    Object[] toArray();  
    <T> T[] toArray(T[] a);  
}
```

10

# Interface Iterable

```

public interface Iterable<T> {

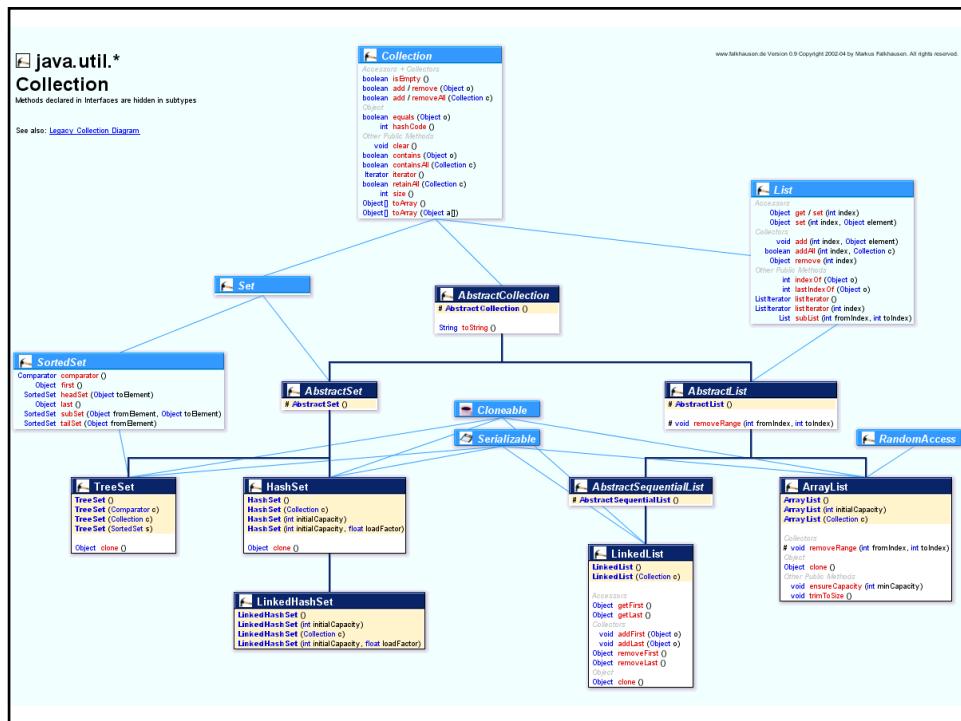
    default void forEach(Consumer<? super T> action)
        Performs the given action for each element of the Iterable until
        all elements have been processed or the action throws an
        exception.

    Iterator<T> iterator()
        Returns an iterator over elements of type T.

    default Spliterator<T> spliterator()
        Creates a Spliterator over the elements described by this
        Iterable.
}

```

11



## Set - Conjuntos

- Uma coleção que não pode conter elementos duplicados.
- Contém apenas os métodos definidos na interface Collection
  - Novos contratos nos métodos add, equals e hashCode
- Implementações:
  - HashSet
  - TreeSet
  - ..

13

## AbstractSet

```
public abstract class AbstractSet<E>
    extends AbstractCollection<E> implements Set<E>  {

    protected AbstractSet() {
    }

    public boolean equals(Object o) {
        if (!(o instanceof Set)) return false;
        return ((Set)o).size()==size() && containsAll((Set)o);
    }

    public int hashCode() {
        int h = 0;
        for( E el : this )
            if ( el != null ) h += el.hashCode();
        return h;
    }
}
```

14

## HashSet

- Usa uma [tabela de dispersão](#) (Hash Table) para armazenar os elementos.
  - [Uma instância de Hashmap](#)
- A inserção de um novo elemento não será efectuada se o [equals](#) do elemento a ser inserido com algum elemento do Set retornar true.
  - [A implementação da função equals é fundamental.](#)
- Desempenho constante,
  - O(-1) para add, remove, contains e size

```
java.lang.Object
  ↘ java.util.AbstractCollection<E>
    ↘ java.util.AbstractSet<E>
      ↘ java.util.HashSet<E>
```

15

## HashSet

```
import java.util.*;

public class TestHashSet {

    public static void main(String args[]) {
        String[] str = {"Rui", "Manuel", "Rui", "Jose",
                        "Pires", "Eduardo", "Santos"};

        Set<String> s = new HashSet<String>();
        for (String i: str ) {
            if (!s.add(i))
                System.out.println("Nome duplicado: " + i);
        }
        System.out.println(s.size() + " palavras distintas");

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

Nome duplicado: Rui  
6 palavras distintas

Manuel  
Rui  
Jose  
Eduardo  
Santos  
Pires

Ordem!

Porquê ?

Conclusão: sem noção de posição (sem ordem)

16

## TreeSet

- A implementação baseada numa estrutura em árvore balanceada.
- Desempenho  $\log(n)$  para add, remove e contains
- Permite a **Ordenação dos Elementos** pela:
  - sua “**ordem natural**”. Os objectos inseridos em TreeSet’s devem implementar a interface Comparable .
  - ou utilizando um objecto do tipo **Comparator** no construtor de TreeSet.

... Exemplo detalhado mais adiante

17

## TreeSet

```
public class TestTreeSet {  
  
    public static void main(String[] args) {  
        Collection<Quadrado> c = new TreeSet<Quadrado>();  
        c.add(new Quadrado(3, 4, 5.6)); c.add(new Quadrado(1, 5, 4));  
        c.add(new Quadrado(0, 0, 6)); c.add(new Quadrado(4, 6, 7.4));  
        System.out.println(c);  
  
        Quadrado q;  
        Iterator<Quadrado> itr = c.iterator();  
        while (itr.hasNext()) {  
            q = itr.next();  
            System.out.println(q);  
        }  
    }  
}  
[Quadrado de Centro (1.0,5.0) e de lado 4.0, Quadrado de Centro (3.0,4.0) e de lado 5.6,  
Quadrado de Centro (0.0,0.0) e de lado 6.0, Quadrado de Centro (4.0,6.0) e de lado 7.4]  
Quadrado de Centro (1.0,5.0) e de lado 4.0  
Quadrado de Centro (3.0,4.0) e de lado 5.6  
Quadrado de Centro (0.0,0.0) e de lado 6.0  
Quadrado de Centro (4.0,6.0) e de lado 7.4}
```

Ordem OK

```

import java.util.Comparator;
import java.util.TreeSet;

class MyComp implements Comparator<String> {
    public int compare(String a, String b) {
        return (a.length() > b.length() ? 1: -1);
    }
}

public class Teste {
    public static void main(String args[]) {
        TreeSet<String> ts = new TreeSet<String>(new MyComp());

        ts.add("jgdshj");
        ts.add("hj");
        ts.add("khsdfk jjskf");
        ts.add("f");
        ts.add("opeiwoj kn kndksjsa");
        ts.add("kndkd");

        for (String element : ts)
            System.out.println(element + " ");
    }
}

```

19

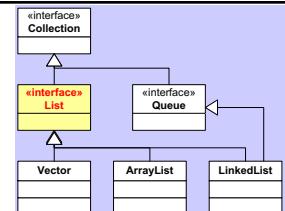
$$(a,b) \rightarrow a.length() > b.length() ? 1: -1$$

*Using FI*

f  
hj  
kndkd  
jgdshj  
khsdfk jjskf  
opeiwoj kn kndksjsa

## Listas

- Podem conter elementos duplicados.
- Para além das operações herdadas de Collection, a interface lista inclui ainda:
  - **Acesso Posicional** – manipulação de elementos baseada na sua posição (índice) na lista
  - **Pesquisa** – de determinado elemento na lista. Retorna a sua posição.
  - **ListIterator** – estende a semântica do Iterator tirando partido da natureza sequencial da lista.
  - **Range-View** – execução de operações sobre uma gama de elementos da lista. (`list.subList(fromIndex, toIndex).clear();`)



20

## List Interface

```
public interface List<E> extends Collection<E> {  
    // Positional Access  
    boolean add(E e)  
    void add(int index, E element);           // Optional  
    E get(int index);  
    E set(int index, E element);           // Optional  
    E remove(int index);                  // Optional  
    boolean addAll(Collection<? extends E> c); // Optional  
  
    // Search  
    int indexOf(Object o);  
    int lastIndexOf(Object o);  
  
    // Iteration  
    ListIterator<E> listIterator();          ■■■►  
    ListIterator<E> listIterator(int index);  
  
    // Range-view  
    List<E> subList(int from, int to);  
}
```

```
public interface ListIterator<E>  
    extends Iterator<E> {  
    boolean hasNext();  
    E next();  
    boolean hasPrevious();  
    E previous();  
    int nextIndex();  
    int previousIndex();  
    void remove(); //optional  
    void set(E e); //optional  
    void add(E e); //optional  
}
```

## Listas - Implementações

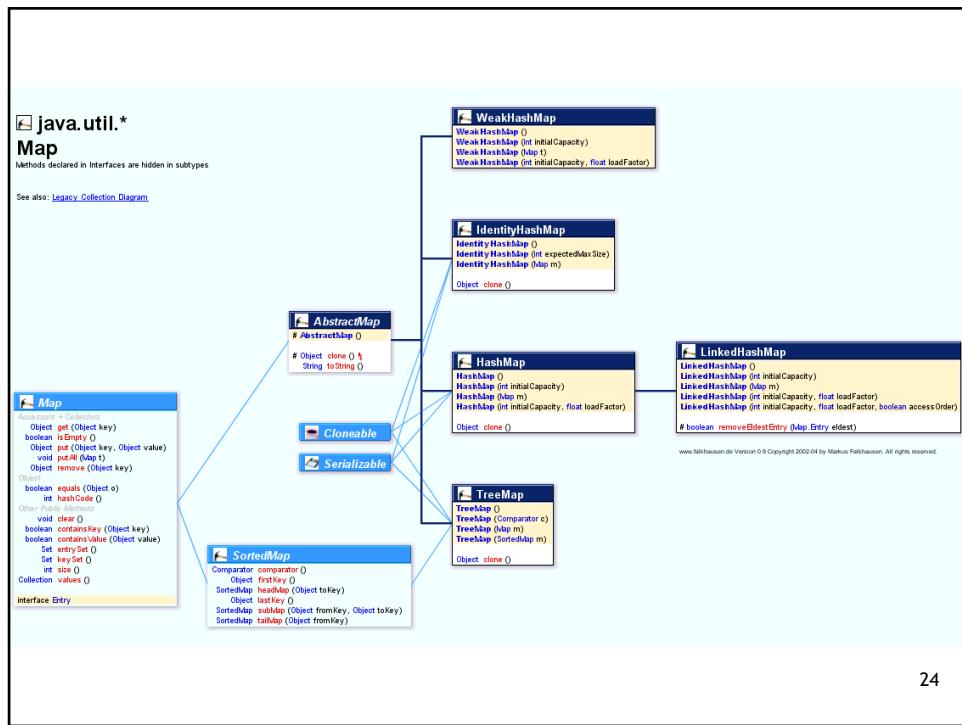
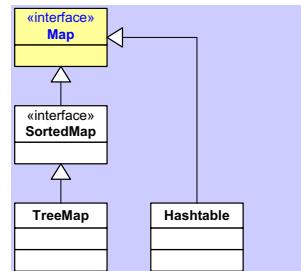
- **ArrayList** - Array redimensionável
  - **LinkedList** - Listas Ligadas
- }Diferença?

```
public static void main(String args[]) {  
    String[] str1 = {"Rui", "Manuel", "Jose", "Pires", "Eduardo", "Santos"};  
    String[] str2 = {"Rosa", "Pereira", "Rui", "Vidal", "Hugo", "Maria"};  
    List<String> larray = new ArrayList<String>();  
    List<String> llist = new LinkedList<String>();  
  
    for (String i: str1 )  larray.add(i);  
  
    for (String i: str2 )  llist.add(i);  
  
    llist.addAll(llist.size()/2, larray);  
    ListIterator itr = llist.listIterator();  
    while ( itr.hasNext() )  
        System.out.println( itr.next() );  
  
    System.out.println("Rui está na posição " +  
        llist.indexOf("Rui") + " e " + llist.lastIndexOf("Rui"));  
  
    llist.set(llist.lastIndexOf("Rui"), "Rui2");  
    System.out.println(llist.lastIndexOf("Rui"));  
}
```

Rosa  
Pereira  
Rui  
Rui  
Manuel  
Jose  
Pires  
Eduardo  
Santos  
Vidal  
Hugo  
Maria  
Rui está na posição 2 e 3  
2

## Mapas - Map

- A Interface Map não descende de Collections.
  - Interface Map<K,V>
- Um mapa é um objecto que associa uma chave (K) a um único valor (V)
  - Não contém keys duplicadas
- Também é denominado como dicionário ou memória associativa
- Métodos disponíveis:
  - adicionar: put(Object key, Object value)
  - remover : remove(Object key)
  - obter um objecto: get(Object key)



## Interface Map<K,V>

```
public interface Map<K,V> {  
    // Basic operations  
    V put(K key, V value);  
    V get(Object key);  
    V remove(Object key);  
    boolean containsKey(Object key);  
    boolean containsValue(Object value);  
    int size();  
    boolean isEmpty();  
  
    // Bulk operations  
    void putAll(Map<? extends K, ? extends V> m);  
    void clear();  
  
    // Collection Views  
    public Set<K> keySet();  
    public Collection<V> values();  
    public Set<Map.Entry<K,V>> entrySet();  
  
    // Interface for entrySet elements  
    public interface Entry {  
        K getKey();  
        V getValue();  
        V setValue(V value);  
    }  
}
```

25

Vistas

- Mapas não são Collections.
- No entanto, podemos obter vistas dos mapas.
- As vistas são do tipo Collection
- Há três vistas disponíveis:
  - conjunto (set) de chaves
  - coleção de valores
  - conjunto (set) de entradas do tipo par chave/valor

26

## Implementações de Map

- **HashMap**
  - Utiliza uma tabela de dispersão (Hash Table)
  - Não existe ordenação nos pares.
- **LinkedHashMap**
  - Semelhante ao HashMap, mas preserva a ordem de inserção
- **TreeMap**
  - Baseado numa árvore balanceada
  - Os pares são ordenados com base na chave - o acesso é  $O(\log N)$

27

## HashMap

```
public static void main(String[] args) {  
    Map<String, Double> mapa = new HashMap<>();  
    mapa.put("Rui", 32.4);  
    mapa.put("Manuel", 3.2);  
    mapa.put("Rita", 5.6);  
  
    System.out.println("O Mapa contém " + mapa.size() + " elementos");  
    System.out.println("O Rui está no Mapa? " + mapa.containsKey("Rui"));  
  
    System.out.println("O Rita tem " + mapa.get("Rita") + "€");  
    mapa.put("Rita", mapa.get("Rita") + 3.6);  
    System.out.println("O Rita tem " + mapa.get("Rita") + "€");  
  
    Set<Entry<String, Double>> set = mapa.entrySet();  
    Iterator<Entry<String, Double>> i = set.iterator();  
    while(i.hasNext()) {  
        Entry<String, Double> aux = i.next();  
        System.out.println("O " + aux.getKey() + " ganha " + aux.getValue() + "€");  
    }  
}
```

O Mapa contém 3 elementos  
O Rui está no Mapa? true  
O Rita tem 5.6€  
O Rita tem 9.2€  
O Manuel ganha 3.2€  
O Rui ganha 32.4€  
O Rita ganha 9.2€

Vista

28

## TreeMap

- Mesmas características das descritas para a TreeSet mas adaptadas a pares key/value.
  - No exemplo anterior, só necessitamos de substituir HashMap por TreeMap
- ```
public static void main(String[] args) {  
    Map<String, Double> mapa = new TreeMap<>();  
    mapa.put("Rui", 32.4);  
    ...  
}
```
- TreeMap oferece a possibilidade de ordenar objectos
    - utilizando a “Ordem Natural” (compareTo) ou um objecto do tipo Comparator

29

## Ordenação em Colecções

1. Implementações com ordenação (TreeSet, TreeMap).
2. Utilizando o método static Collections.sort()

Há duas formas de definir uma ordem (key) de objectos:

- **Ordem Natural**
  - Cada Classe ao implementar a interface Comparable.
  - Método: int compareTo(Object o)
- **Utilizando o Comparator**
  - Se um objecto não tem ordem natural e/ou pretendemos definir uma nova ordem arbitrária

```
interface Comparator<T> {  
    int compare(T o1, T o2)  
    boolean equals(Object obj)  
}
```

30

## TreeMap Ordenado

```
class StringLenComparator implements Comparator<String> {
    @Override
    public int compare(String s1, String s2) {
        if (s1 == null || s2 == null)
            throw new NullPointerException();
        return s1.length() - s2.length();
    }
}

public class TestTreeMap {
    public static void main(String[] args) {
        Map<String, Double> mapa =
            new TreeMap<>(new StringLenComparator());
        mapa.put("Rui", 32.4);
        ...
    }
}
```

O Mapa contém 3 elementos  
Rui está no Mapa? True  
O Rui ganha 32.4€  
O Rita ganha 9.2€  
O Manuel ganha 3.2€

**Ordenação**

31

## Collections sort()

- Para ordenar uma coleção não ordenada

```
public class TestArrayListSorted {
    public static void main(String args[]) {
        String[] str1 = {"Rui", "Manuel", "Jose",
                         "Pires", "Eduardo", "Santos"};
        List<String> list = new LinkedList<>();
        list.addAll(Arrays.asList(str1));

        Collections.sort(list, new StringLenComparator());

        ListIterator<String> itr = list.listIterator();
        while (itr.hasNext())
            System.out.println(itr.next());

    }
}
```

Rui  
Jose  
Pires  
Manuel  
Santos  
Eduardo

32

## Collections sort()

- Outra forma ... Classe Anónima

```
public class TestArrayListSorted {  
    public static void main(String args[]) {  
        String[] str1 = {"Rui", "Manuel", "Jose",  
                         "Pires", "Eduardo", "Santos"};  
        List<String> list = new LinkedList<>();  
        list.addAll(Arrays.asList(str1));  
        Collections.sort(list, new Comparator<String>()  
        {  
            @Override  
            public int compare(String s1, String s2) {  
                if (s1 == null || s2 == null)  
                    throw new NullPointerException();  
                return s1.length() - s2.length();  
            }  
        });  
        for (String s: list) // equivalente ao anterior  
            System.out.println(s);  
    }  
}
```

Rui  
Jose  
Pires  
Manuel  
Santos  
Eduardo

33

## Collections sort()

- Outra forma ainda ... Lambda expressions

```
public class TestArrayListSorted {  
    public static void main(String args[]) {  
        String[] str1 = {"Rui", "Manuel", "Jose",  
                         "Pires", "Eduardo", "Santos"};  
        List<String> list = new LinkedList<>();  
        list.addAll(Arrays.asList(str1));  
        Collections.sort(list, (s1,s2) -> {  
            if (s1 == null || s2 == null)  
                throw new NullPointerException();  
            return s1.length() - s2.length();}  
        );  
        for (String s: list) // equivalente ao anterior  
            System.out.println(s);  
    }  
}
```

Rui  
Jose  
Pires  
Manuel  
Santos  
Eduardo

34

## Collections sort()

- E ainda!... utilizando a Java Stream API\*

```
public class TestArrayListSorted {  
    public static void main(String args[]) {  
        String[] str1 = {"Rui", "Manuel", "Jose",  
                         "Pires", "Eduardo", "Santos"};  
        List<String> list = new LinkedList<>();  
        list.addAll(Arrays.asList(str1));  
  
        Collections.sort(list, Comparator.comparing(String::length));  
  
        for (String s: list) // equivalente ao anterior  
            System.out.println(s);  
    }  
}
```

Rui  
Jose  
Pires  
Manuel  
Santos  
Eduardo

Method Reference!

\*described in the next slides...

35

## Algoritmos

- A JCF fornece ainda um conjunto de algoritmos que podem ser usados em colecções
  - Métodos estáticos de utilização global
- Exemplos:
  - sort, binarySearch, copy, shuffle, reverse, max, min, etc.
- java.util.Collections
- java.util.Arrays

36

## java.util.Collections

|                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Collections                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| +binarySearch(list: List, key: Object) : int<br>+binarySearch(list: List, key: Object, c: Comparator) : int<br>+copy(src: List, des: List) : void<br>+enumeration(c: final Collection) : Enumeration<br>+fill(list: List, o: Object) : void<br>+max(c: Collection) : Object<br>+max(c: Collection, c: Comparator) : Object<br>+min(c: Collection) : Object<br>+min(c: Collection, c: Comparator) : Object<br>+nCopies(n: int, o: Object) : List<br>+reverse(list: List) : void<br>+reverseOrder() : Comparator<br>+shuffle(list: List) : void<br>+shuffle(list: List, rd: Random) : void<br>+singleton(o: Object) : Set<br>+singletonList(o: Object) : List<br>+singletonMap(key: Object, value: Object) : Map<br>+sort(list: List) : void<br>+sort(list: List, c: Comparator) : void<br>+synchronizedCollection(c: Collection) : Collection<br>+synchronizedList(list: List) : List<br>+synchronizedMap(m: Map) : Map<br>+synchronizedSet(s: Set) : Set<br>+synchronizedSortedMap(s: SortedMap) : SortedMap<br>+synchronizedSortedSet(s: SortedSet) : SortedSet<br>+unmodifiedCollection(c: Collection) : Collection<br>+unmodifiedList(list: List) : List<br>+unmodifiedMap(m: Map) : Map<br>+unmodifiedSet(s: Set) : Set<br>+unmodifiedSortedMap(s: SortedMap) : SortedMap<br>+unmodifiedSortedSet(s: SortedSet) : SortedSet |

## java.util.Arrays

|                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                 |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Arrays                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |
| +asList(a: Object[]) : List<br>+binarySearch(a: byte[], key: byte) : int<br>+binarySearch(a: char[], key: char) : int<br>+binarySearch(a: double[], key: double) : int<br>+binarySearch(a: float[], key: float) : int<br>+binarySearch(a: int[], key: int) : int<br>+binarySearch(a: long[], key: long) : int<br>+binarySearch(a: Object[], key: Object) : int<br>+binarySearch(a: Object[], key: Object, c: Comparator) : int<br>+binarySearch(a: short[], key: short) : int<br>+equals(a: boolean[], a2: boolean[]) : boolean<br>+equals(a: byte[], a2: byte[]) : boolean<br>+equals(a: char[], a2: char[]) : boolean<br>+equals(a: double[], a2: double[]) : boolean<br>+equals(a: float[], a2: float[]) : boolean<br>+equals(a: int[], a2: int[]) : boolean<br>+equals(a: long[], a2: long[]) : boolean<br>+equals(a: Object[], a2: Object[]) : boolean<br>+equals(a: short[], a2: short[]) : boolean<br>+fill(a: boolean[], val: boolean) : void<br>+fill(a: boolean[], fromIndex: int, toIndex: int, val: boolean) : void |

Overloaded fill method for char, byte, short, int, long, float, double, and Object.

|                                                                                  |
|----------------------------------------------------------------------------------|
| +sort(a: byte[]) : void<br>+sort(a: byte[], fromIndex: int, toIndex: int) : void |
|----------------------------------------------------------------------------------|

Overloaded sort method for char, short, int, long, float, double, and Object.

## Exemplo

```
String[] str1 = {"Rui", "Manuel", "Jose",
                 "Pires", "Eduardo", "Santos"};
List<String> list = new ArrayList<>();
list.addAll(Arrays.asList(str1));
Collections.sort(list, new Comparator<String>() {
    @Override
    public int compare(String s1, String s2) {
        if (s1 == null || s2 == null)
            throw new NullPointerException();
        return s1.compareTo(s2);
    }
});
for (String s: list)
    System.out.println(s);

for (int i=0; i<str1.length; i++)
    System.out.println(Collections.binarySearch(list,str1[i]));

```

Eduardo  
Jose  
Manuel  
Pires  
Rui  
Santos  
4  
2  
1  
3  
0  
5

39

## Collections in Java 8

Stream API

40

## Method References

- Treating an existing method as an instance of a Functional Interface

- Examples

```
class Person {  
    private String name;  
    public String getName() { return name; }  
}  
  
Person[] people = ...;  
Comparator<Person> byName =  
    Comparator.comparing(Person::getName);  
Arrays.sort(people, byName);
```

- More Examples

```
Consumer<Integer> b1 = System::exit;  
Consumer<String[]> b2 = Arrays::sort;  
Consumer<String> b3 = MyProgram::main;  
Runnable r = MyProgram::main;
```

## Method References

- A **static** method (ClassName::methName)
- An **instance** method of a particular static object (instanceRef::methName)
- A **super** method of a particular object (super::methName)
- An instance method of an arbitrary object of a particular type (ClassName::methName)
- A **class constructor** reference (ClassName::new)
- An **array constructor** reference (TypeName[]::new)- "Instance method of an arbitrary object" adds an argument of that type which becomes the receiver of the invocation

## Traversing Collections

There are three ways to traverse collections:

1. Iterator

```
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove(); //optional
}
```

2. "for-each" e forEach (java 8)

```
for (Object o : collection) // for each
    System.out.println(o);

List<String> l = Arrays.asList("Ana", "Ze", "Rui");
l.forEach(s -> System.out.println(s)); // forEach
// l.forEach(System.out::println); // forEach
```

3. Aggregate operations (java 8)

## Aggregate Operations - Java 8 Streams API

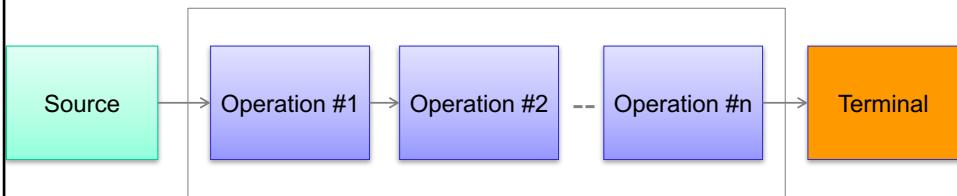
- The preferred method of iterating over a collection is to obtain a stream and perform aggregate operations on it.
- Aggregate operations are often used in conjunction with lambda expressions to make programming more expressive, using less lines of code.
- **Package java.util.stream**
  - The key abstraction introduced in this package is stream.
  - The classes **Stream**, **IntStream**, **LongStream**, and **DoubleStream** are streams over objects and the primitive int, long and double types.

## java.util.stream

Streams differ from collections in several ways:

- No storage
  - A stream is not a data structure that stores elements; instead, it conveys elements through a pipeline of computational operations.
- Functional in nature
  - An operation on a stream produces a result, but does not modify its source.
- Laziness-seeking
  - Many stream operations, such as filtering, mapping, or duplicate removal, can be implemented lazily, exposing opportunities for optimization.  
Intermediate operations are always lazy.
- Possibly unbounded
  - While collections have a finite size, streams need not.
- Consumable
  - The elements of a stream are only visited once during the life of a stream.  
Like an Iterator, a new stream must be generated to revisit the same elements of the source.

## Stream Pipeline



- (1) Obtain a stream from a **source**
- (2) Perform one or more intermediate **operations**
- (3) Perform one **terminal** operation

46

## java.util.stream - sources

- Streams can be obtained in a number of ways. Streams **sources** include:
  - From a **Collection** via the stream() and parallelStream() methods;
  - From an **Array** via Arrays.stream(Object[]);
  - From static factory methods on the stream classes, such as Stream.of(Object[]), IntStream.range(int, int) or Stream.iterate(Object, UnaryOperator);
  - The lines of a file can be obtained from BufferedReader.lines();
  - Streams of file paths can be obtained from methods in Files;
  - Streams of random numbers can be obtained from Random.ints();
  - Numerous other stream-bearing methods in the JDK, including BitSet.stream(), Pattern.splitAsStream(java.lang.CharSequence), and JarFile.stream().

## java.util.stream - Intermediate operations

- **.filter** - excludes all elements that don't match a Predicate
- **.map** - perform transformation of elements using a Function
- **.flatMap** - transform each element into zero or more elements by way of another Stream
- **.peek** - performs some action on each element
- **.distinct** - excludes all duplicate elements (equals())
- **.sorted** - ordered elements (Comparator)
- **.limit** - maximum number of elements
- **.substream** - range (by index) of elements
- (*and many more -> see java.util.stream.Stream<T>*)

```
List<Person> persons = ...;
Stream<Person> tenPersonsOver18 = persons.stream()
    .filter(p -> p.getAge() > 18)
    .limit(10);
```

## java.util.stream - Terminating operations

- Reducers
  - `reduce()`, `count()`, `findAny()`, `findFirst()`
- Collectors
  - `collect()`
- `forEach`
- iterators

```
List<Person> persons = ...;
List<Student> students = persons.stream()
    .filter(p -> p.getAge() > 18)
    .map(Student::new)
    .collect(Collectors.toList());
```

## Stream.Filter

- Filtering a stream of data is the first natural operation that we would need.
- Stream interface exposes a filter method that takes in a Predicate that allows us to use lambda expression to define the filtering criteria:

```
List<Person> persons = ...
Stream<Person> personsOver18 =
    persons.stream().filter(p -> p.getAge() > 18);

// other Filter example with Predicate && Consumer

List<String> l = Arrays.asList("Ana", "Ze", "Rui");
l.stream().filter(n -> n.length()>3)
    .forEach(System.out::println);
```

## Stream.Map

- The map operations allows us to apply a function that takes in a parameter of one type, and returns something else.

```
Stream<Student> map = persons.stream()
    .filter(p -> p.getAge() > 18)
    .map(person -> new Student(person));

// other example with Map && Consumer

List<String> l = Arrays.asList("Ana", "Ze", "Rui");
l.stream().map(n -> "Nome Pessoa:" + n)
    .forEach(System.out::println);
```

## Stream.Reduce

- A *reduction* operation takes a sequence of input elements and combines them into a single summary result by repeated application of a combining operation
- For instance, finding the sum or maximum of a set of numbers, or accumulating elements into a list.

```
// example with Map & Reduce
List<Integer> costBeforeTax = Arrays.asList(100, 200, 300,
    400, 500);

double bill = costBeforeTax.stream()
    .map(cost -> (cost*1.23))
    .reduce(0.0,(sum, cost) -> sum + cost));

System.out.println("Total : " + bill);
```

## Stream.Collect

- While stream abstraction is continuous by its nature, we can describe the operations on streams but to acquire the final results we have to collect the data somehow.
- The Stream API provides a number of “terminal” operations. The collect() method is one of those terminals that allows us to collect the results of the operations:

```
List<Student> students = persons.stream()
    .filter(p -> p.getAge() > 18)
    .map(Student::new)
    .collect(Collectors.toList());

// other example with Map && Collect
List<String> l = Arrays.asList("Ana", "Ze", "Rui");
List<String> res = l.stream()
    .map(n -> "Nome: " + n)
    .collect(Collectors.toList());
res.forEach(System.out::println);
```

## Stream.Parallel and Sequential

- One interesting feature of the new Stream API is that it doesn't require the operations to be either parallel or sequential from beginning till the end.
- It is possible to start consuming the data concurrently, then switch to sequential processing and back at any point in the flow:

```
List<Student> students = persons.stream()
    .parallel()
    .filter(p -> p.getAge() > 18)
    // filtering will be performed concurrently
    .sequential()
    .map(Student::new)
    .collect(Collectors.toCollection(ArrayList::new));
```

## Aggregate Operations - examples

- The following code sequentially iterates through a collection of shapes and prints out the red objects:

```
myShapesCollection.stream()
    .filter(e -> e.getColor() == Color.RED)
    .forEach(e -> System.out.println(e.getName()));
```
- There are many different ways to collect data with this API. For example, you might want to convert the elements of a Collection to String objects, then join them, separated by commas:

```
String joined = elements.stream()
    .map(Object::toString)
    .collect(Collectors.joining(", "));
```
- Or perhaps sum the salaries of all employees:

```
int total = employees.stream()
    .collect(Collectors.summingInt(Employee::getSalary));
```

## bulk operations

- The Collections framework has always provided a number of so-called "bulk operations" as part of its API.
- These include methods that operate on entire collections, such as containsAll, addAll, removeAll, etc.
- Do not confuse those methods with the aggregate operations that were introduced in JDK 8.
- The key difference between the new aggregate operations and the existing bulk operations (containsAll, addAll, etc.) is that the old versions are all mutative, meaning that they all modify the underlying collection.
- In contrast, the new aggregate operations do not modify the underlying collection. When using the new aggregate operations and lambda expressions, you must take care to avoid mutation so as not to introduce problems in the future, should your code be run later from a parallel stream.

## Sumário

- JAVA Collections FrameWork (JCF)
  - Organização e Principais Interfaces
  - Conjuntos (HashSet e TreeSet)
  - Listas (ArrayList e LinkedList)
  - Mapas (HashMap e TreeMap)
  - Operações sobre Colecções
- JAVA Stream API

57

# Java Reflection

Programação III  
José Luis Oliveira; Carlos Costa

1

## What is reflection?

- When you look in a mirror:
  - You can see your reflection
  - You can act on what you see, for example, straighten your tie
- In computer programming:
  - Reflection is infrastructure enabling a program can see and manipulate itself
  - It consists of metadata plus operations to manipulate the metadata
- Meta means self-referential
  - So metadata is data (information) about oneself

*Introduction to Java Reflection, Ciaran McHale.*

## What is reflection?

### Reflection

- “Reflection is the ability of a program to manipulate as data something representing the state of the program during its own execution.” [Demers and Malenfant]

### Java Tutorials

- “Reflection is commonly used by programs which require the ability to examine or modify the runtime behavior of applications running in the Java virtual machine.”
- “... advanced feature ... a powerful technique ... can enable applications to perform operations which would otherwise be impossible.”

3

## Java looking at Java

- Reflection permite que um programa se examine a si mesmo.
- Podemos:
- Determinar a classe de um objecto
  - Descobrir toda a informação associada a determinada classe:
    - access modifiers, superclass, fields, constructors, and methods
  - Obter informação relativa ao conteúdo de uma interface.
  - Mesmo sem saber o nome (classes, métodos,...) podemos:
    - Criar uma instância de uma classe
    - ler/modificar variáveis
    - Invocar métodos
    - Criar e manipular vectores de objectos

## Utilização de Java Reflection

- JavaBeans (component architectures)
- Database applications
- Serialization
- Scripting applications
- Runtime Debugging/Inspection Tools
- etc

5

## Acesso a metadados

- Java armazena metadados em classes
  - Metadata for a class: `java.lang.Class`
  - Metadata for a constructor: `java.lang.reflect.Constructor`
  - Metadata for a field: `java.lang.reflect.Field`
  - Metadata for a method: `java.lang.reflect.Method`
- Podemos aceder à Class de um objecto de duas formas:

```
Class<?> cl1 = Class.forName("java.util.Properties");  
ou  
Object obj = ... // e.g. new StringBuffer("Teste");  
Class<?> cl2 = obj.getClass();
```
- As classes do package Reflection são inter-dependentes
  - Exemplos a seguir...

## Metadata de tipos primitivos e vectores

- Java associa uma instância de Class a cada tipo primitivo:

```
Class<?> c1 = int.class;  
Class<?> c2 = boolean.class;  
Class<?> c3 = void.class;
```

- Podemos usar Class.forName() para aceder à classe de um vector

```
Class<?> c4 = byte.class; // byte  
Class<?> c5 = Class.forName("[B"); // byte[]  
Class<?> c6 = Class.forName("[[B"); // byte[][]
```

- Encoding scheme utilizado por Class.forName()

B → byte; C → char; D → double; F → float; I → int; J → long;  
Lclass-name → class-name[]; S → short; Z → boolean  
Use as many "["s as there are dimensions in the array

## Reflection API - Class

```
public final class Class<T>  
extends Object  
implements Serializable, GenericDeclaration,  
Type, AnnotatedElement  
  
    static Class<?> forName(String className);  
    T newInstance();  
    Field[] getFields();  
    Method[] getMethods();  
    boolean isInstance(Object obj);  
    String getName();  
  
    getInterfaces(), getSuperclass(),  
    getModifiers(), getField(), getMethod(), ...
```

```
void printClassName(Object obj) {  
    System.out.println("The class of " + obj +  
        " is " + obj.getClass().getName());  
}
```

8

## Reflection API - Field

```
public final class Field  
extends AccessibleObject  
implements Member  
  
    Object get(Object obj);  
    void set(Object obj, Object val);  
  
    getType(), getDeclaringClass(),  
    setDouble(...), setInt(...), .....
```

```
Field[] flds = someObject.getClass().getFields();  
for (Field f: flds)  
    System.out.println(f.getName());
```

9

## Reflection API - Method

```
public final class Method  
extends AccessibleObject  
implements GenericDeclaration, Member  
  
    Object invoke(Object obj, Object... args);  
    Class<?> getReturnType();  
    Class<?>[] getParameterTypes(),  
  
    getExceptionTypes(), getDeclaringClass(), ..
```

```
Method methods[] = someClass.getMethods();  
for (Method m: methods)  
    System.out.println(m);
```

10

## Reflection API - others...

```
class Constructor<T>
class AccessibleObject
    ← Constructor,
    ← Field,
    ← Method
class Array;
class Proxy;
Class Modifier
```

### java.lang.reflect

#### Interfaces

AnnotatedElement  
GenericArrayType  
GenericDeclaration  
InvocationHandler  
Member  
ParameterizedType  
Type  
TypeVariable  
WildcardType

#### Classes

AccessibleObject  
Array  
Constructor  
Field  
Method  
Modifier  
Proxy  
ReflectPermission

#### Exceptions

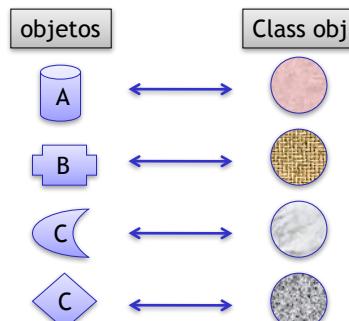
InvocationTargetException  
MalformedParameterizedTypeException  
UndeclaredThrowableException

#### Errors

GenericSignatureFormatError

## A Classe Class

- Para cada objecto carregado pela JVM, existe um objecto do tipo *Class associado*.
  - Os tipos primitivos também são representados por objectos Class.*
- As instâncias do tipo *Class* armazenam informações sobre a classe:
  - Nome da classe
  - Herança
  - Interfaces Implementadas
  - Métodos
  - Atributos
- Permite invocar métodos e referenciar atributos*



12

## Métodos de java.lang.Class - 1

- **public static Class<?> forName(String className)**
  - returns a Class object that represents the class with the given name
- **public String getName()**
  - returns the full name of the Class object, such as “java.lang.String”.
- **public int getModifiers()**
  - returns an integer that describes the class modifier: public, final or abstract
- **public T newInstance()**
  - creates an instance of this class at runtime

13

## Exemplo - newInstance

```
public class ReflectionNew {  
    public static void main(String[] args) throws Exception {  
        Class<?> sc = Class.forName("aula5_1.Circulo");  
  
        System.out.println("Name = " + sc.getName());  
        System.out.println("SimpleName = " + sc.getSimpleName());  
        System.out.println("CanonicalName = " + sc.getCanonicalName());  
  
        Class<?>[] paramTypes = { Double.TYPE, Double.TYPE, Double.TYPE };  
        Constructor<?> cons = sc.getConstructor(paramTypes);  
        Object ar[] = { 2, 4, 10 };  
        Object theObject = cons.newInstance(ar);  
        System.out.println("New object: " + theObject);  
  
        Constructor<?> cs = sc.getConstructor(new Class<?>[]{Double.TYPE});  
        System.out.println("New object: " + cs.newInstance(new Object[]{20}));  
    }  
}
```

Name = aula5\_1.Circulo  
SimpleName = Circulo  
CanonicalName = aula5\_1.Circulo  
New object: Circulo de Centro (2.0,4.0) e de raio 10.0  
New object: Circulo de Centro (0.0,0.0) e de raio 20.0

14

## Exemplo - Modifiers

```
public class SampleModifier {  
    public static void main(String[] args) {  
        printModifiers(new String());  
        printModifiers(new SampleModifier());  
    }  
    public static void printModifiers(Object o) {  
        Class<?> c = o.getClass(); // returns the Class object of o  
        System.out.print("***** Class " + c.getName() + " : ");  
  
        int m = c.getModifiers(); // return the class modifiers  
        if (Modifier.isPublic(m)) // checks if is public  
            System.out.print("public ");  
        if (Modifier.isAbstract(m)) // checks if it is abstract  
            System.out.print("abstract ");  
        if (Modifier.isFinal(m)) // checks if it is final  
            System.out.print("final "); System.out.println();  
    }  
}  
***** Class java.lang.String : public final  
***** Class reflection.SampleModifier : public
```

15

## Métodos de java.lang.Class - 2

- **public Class[] getClasses()**
  - returns an array of all inner classes of this class
- **public Constructor getConstructor(Class[] params)**
  - returns all public constructors of this class whose formal parameter types match those specified by params
- **public Constructor[] getConstructors()**
  - returns all public constructors of this class

16

## Exemplo - Construtores

```
public class Reflection2 {  
    public static void main(String[] args) throws InstantiationException,  
        IllegalAccessException {  
        String s="Mar";  
        Class<?> sc = s.getClass();  
        System.out.println("\n***** Construtores *****\n");  
        Constructor<?> contrs[] = sc.getConstructors();  
        for (Constructor<?> c: contrs)  
            System.out.println(c);  
    }  
}***** Construtores *****  
public java.lang.String()  
public java.lang.String(java.lang.String)  
public java.lang.String(char[])  
public java.lang.String(char[],int,int)  
public java.lang.String(int[],int,int)  
public java.lang.String(byte[],int,int,int)  
public java.lang.String(byte[],int)  
public java.lang.String(byte[],int,int,int,int,java.lang.String) throws java.io.UnsupportedEncodingException  
public java.lang.String(byte[],int,int,java.nio.charset.Charset)  
public java.lang.String(byte[],java.lang.String) throws java.io.UnsupportedEncodingException  
public java.lang.String(byte[],java.nio.charset.Charset)  
public java.lang.String(byte[],int)  
public java.lang.String(byte[])  
public java.lang.String(java.lang.StringBuffer)  
public java.lang.String(java.lang.StringBuilder)
```

17

## Métodos de java.lang.Class - 3

- **public Field getField(String name)**
  - returns an object of the class Field that corresponds to the instance variable of the class that is called name
- **public Field[] getFields()**
  - returns all accessible public instance variables of the class
- **public Field[] getDeclaredFields()**
  - returns all declared fields (instance variables) of the class

18

## Exemplo - Fields

```
public class Reflection2 {  
    public static void main(String[] args) throws InstantiationException,  
        IllegalAccessException {  
        String s="Mar";  
        Class<?> sc = s.getClass();  
        System.out.println("\n***** Fields *****\n");  
        Field fields[] = sc.getFields();  
        for (Field f: fields)  
            System.out.println(f);  
    }  
  
    public static final java.util.Comparator java.lang.String.CASE_INSENSITIVE_ORDER
```

19

## Exemplo - Fields

```
public static void main(String[] args) throws Exception {  
    Class<?> sc = Class.forName("aula5_1.Circulo");  
    System.out.println("\n***** Fields *****\n");  
    Field fields[] = sc.getFields();  
    for (Field f: fields)  
        System.out.println(f);  
    System.out.println("\n***** Declared Fields *****\n");  
    Field dfields[] = sc.getDeclaredFields();  
    for (Field f: dfields)  
        System.out.println(f);  
    System.out.println("\n***** raio Field *****\n");  
    Field field = sc.getField("raio"); // deve usar-se getDeclaredField  
    System.out.println(field);  
}  
***** Fields *****  
***** Declared Fields *****  
private double aula5_1.Circulo.raio  
***** raio Field *****  
Exception in thread "main" java.lang.NoSuchFieldException: aula5_1.Circulo.raio  
at java.lang.Class.getField(Class.java:1520)  
at reflection.Reflection2.main(Reflection2.java:39)
```

20

## Ler atributos

```
class SampleGet {  
    public static void main(String[] args) {  
        Rectangle r = new Rectangle(100, 325);  
        printHeight(r);  
    }  
    static void printHeight(Rectangle r) {  
        Field heightField; //declares a field  
        Integer heightValue;  
        Class<?> c = r.getClass(); //get the Class object  
        try {  
            heightField = c.getField("height"); //get the field object  
            heightValue = (Integer)heightField.get(r); //get the value of the  
field  
            System.out.println("Height: " + heightValue.toString());  
        } catch (Exception e) {  
            e.printStackTrace();  
        }  
    }  
}
```

Height: 325

21

## Modificar atributos

```
class SampleSet {  
    public static void main(String[] args) {  
        Rectangle r = new Rectangle(100, 20);  
        System.out.println("original: " + r.toString());  
        modifyWidth(r, new Integer(300));  
        System.out.println("modified: " + r.toString());  
    }  
    static void modifyWidth(Rectangle r, Integer widthParam ) {  
        Field widthField; //declare a field  
        Integer widthValue;  
        Class<?> c = r.getClass(); //get the Class object  
        try {  
            widthField = c.getField("width"); //get the field object  
            widthField.set(r, widthParam); //set the field to widthParam =300  
        } catch (Exception e ) {  
            // . . .  
        }  
    }  
}
```

original: java.awt.Rectangle[x=0,y=0,width=100,height=20]  
modified: java.awt.Rectangle[x=0,y=0,width=300,height=20]

22

## Métodos de java.lang.Class - 4

- `public Method getMethod(String name, Class[] params)`
  - returns an object Method that corresponds to the method called name with a set of parameters params
- `public Method[] getMethods()`
  - returns all accessible public methods of the class
- `public Method[] getDeclaredMethods()`
  - returns all declared methods of the class.
- `public Package getPackage()`
  - returns the package that contains the class
- `public Class getSuperClass()`
  - returns the superclass of the class

23

## Exemplo - Métodos

```
public class Reflection2 {  
    public static void main(String[] args) throws InstantiationException,  
        IllegalAccessException {  
        String s="Mar";  
        Class<?> sc = s.getClass();  
        System.out.println("\n***** Métodos *****\n");  
        Method methods[] = sc.getMethods();  
        for (Method m: methods)  
            System.out.println(m);  
    }  
}  
  
***** Métodos *****  
public boolean java.lang.String.equals(java.lang.Object)  
public java.lang.String java.lang.String.toString()  
public int java.lang.String.hashCode()  
.....  
public final native void java.lang.Object.notify()  
public final native void java.lang.Object.notifyAll()
```

24

## Manipulação de vectores

```
public static void main(final String[] args) {
    try {
        String[] z = new String[] { "Jim", "John", "Joe" };
        final Class<?> type = z.getClass();
        if (!type.isArray()) {
            throw new IllegalArgumentException();
        } else {
            System.out.println("Name = " + type.getName() +
                "\nType = " + type.getComponentType());
        }
    } catch (final Exception ex) {
        ex.printStackTrace();
    }
}
```

Name = [Ljava.lang.String;  
Type = class java.lang.String

25

## Manipulação de vectores

```
public class ArrayNew {
    public static void main(String[] args) throws ClassNotFoundException {
        System.out.println(createNativeArray("int", 12).getClass());
        System.out.println(createNativeArray("boolean", 10, 10).getClass());
        System.out.println(createNativeArray("double", 5, 5, 5).getClass());
    }
    public static Object createNativeArray(String typeName, int... dim)
        throws ClassNotFoundException {
        Class<?> clazz = null;
        if ("int".equals(typeName)) {
            clazz = Integer.TYPE;
        } else if ("boolean".equals(typeName)) {
            clazz = Boolean.TYPE;
        } else if ("double".equals(typeName)) {
            clazz = Double.class;
            // All other native types: short, long, float .....
        } else {
            throw new ClassNotFoundException(typeName);
        }
        return Array.newInstance(clazz, dim);
    }
}
```

class [I  
class [[Z  
class [[[Ljava.lang.Double;

26

## Utilização de Plugins

```
public interface IPlugin {  
    public void metodo();  
}  
  
public class Plugin1 implements IPlugin {  
    public void metodo() {  
        System.out.println("Plugin1: metodo invocado");  
    }  
}  
  
public class Plugin2 implements IPlugin {  
    public void metodo() {  
        System.out.println("Plugin2: metodo invocado");  
    }  
}  
  
public class Plugin3 implements IPlugin {  
    public void metodo() {  
        System.out.println("Plugin3: metodo invocado");  
    }  
}
```

27

IPlugin.java

Plugin1.java

Plugin2.java

Plugin3.java

## Utilização de Plugins

```
package reflection;  
import java.io.File;  
  
abstract class PluginManager {  
    public static IPlugin load(String name) throws Exception {  
        Class<?> c = Class.forName(name);  
        return (IPlugin) c.newInstance();  
    }  
}  
  
public class Plugin {  
    public static void main(String[] args) throws Exception {  
  
        File proxyList = new File("reflection/plugins");  
        for (String f: proxyList.list()) {  
            try {  
                IPlugin obj =  
                    PluginManager.load("reflection."+f.substring(0,f.lastIndexOf('.')));  
                obj.metodo();  
            }  
            catch (Exception e) {  
                e.printStackTrace();  
            }  
        }  
    }  
}
```

Plugin.java

Plugin1: metodo invocado  
Plugin2: metodo invocado  
Plugin3: metodo invocado

28

## Padrões: Fábrica sem reflection

```
class Viveiro {  
    public static Arvore factory(String pedido) {  
        if (pedido.equalsIgnoreCase("Figueira"))  
            { return new Figueira(); }  
        if (pedido.equalsIgnoreCase("Pessegoiro"))  
            { return new Pessegoiro(); }  
        if (pedido.equalsIgnoreCase("Nespereira"))  
            { return new Nespereira(); }  
        else  
            throw new IllegalArgumentException("Árvore não  
existente!");  
    }  
}
```

29

## Padrões: Fábrica com reflection

```
class Viveiro {  
    public static Arvore factory(String pedido){  
        Arvore arv = null;  
        try {  
            arv =  
                (Arvore) Class.forName("patterns."+pedido).newInstance();  
        }  
        catch(Exception e) {  
            throw new IllegalArgumentException("Árvore não  
existente!");  
        }  
        return arv;  
    }  
}
```

30

## java.lang.reflect.Proxy

- Proxy provides static methods for creating dynamic proxy classes and instances, and it is also the superclass of all dynamic proxy classes created by those methods.
- To create a proxy for some interface Foo:

```
InvocationHandler handler = new MyInvocationHandler(...);
Class proxyClass = Proxy.getProxyClass(
    Foo.class.getClassLoader(), new Class[] { Foo.class });
Foo f = (Foo) proxyClass.
    getConstructor(new Class[] { InvocationHandler.class }).
    newInstance(new Object[] { handler });
```

- or more simply:

```
Foo f = (Foo) Proxy.newProxyInstance(Foo.class.getClassLoader(),
    new Class[] { Foo.class },
    handler);
```

31

## Criação de proxies

- Podemos criar dinamicamente um proxy usando o método `Proxy.newProxyInstance()`. Este método aceita 3 parâmetros:
  1. O `ClassLoader` que “carrega” dinamicamente a class proxy
  2. Um vector das interfaces implementadas
  3. Um `InvocationHandler` para reencaminhar as chamadas aos métodos
- Exemplo:

```
InvocationHandler handler = new MyInvocationHandler();
MyInterface proxy = (MyInterface) Proxy.newProxyInstance(
    MyInterface.class.getClassLoader(),
    new Class[] { MyInterface.class },
    handler);
```

  - A variável proxy passa a referenciar uma implementação dinâmica da interface `MyInterface`.
  - Todas as invocações ao proxy serão passadas à implementação do handler (do tipo `InvocationHandler`)

32

## Proxy - utilização

- Database Connection and Transaction Management
- Dynamic Mock Objects for Unit Testing
- Adaptation of DI Container to Custom Factory Interfaces
- AOP-like Method Interception
- ..

33

## Dynamic Proxy Classes

```
package reflection;

import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;

interface MyInterface {
    void method();
}

class MyInterfaceImpl implements MyInterface {
    public void method() {
        System.out.println("method");
    }
}

class MyInterfaceImpl2 implements MyInterface {
    public void method() {
        System.out.println("outro método");
    }
}
```

34

## Dynamic Proxy Classes

```
class ProxyClass implements InvocationHandler {  
    Object obj;  
  
    public ProxyClass(Object o) {  
        obj = o;  
    }  
  
    public Object invoke(Object proxy, Method m, Object[] args) throws Throwable  
    {  
        Object result = null;  
        try {  
            System.out.println("before the method is called ");  
            result = m.invoke(obj, args);  
        } catch (Exception eBj) {  
        } finally {  
            System.out.println("after the method is called");  
        }  
        return result;  
    }  
}
```

35

## Dynamic Proxy Classes

```
public class ProxySample {  
    public static void main(String[] argv) throws Exception {  
  
        MyInterface myintf =  
            (MyInterface) Proxy.newProxyInstance(MyInterface.class.getClassLoader(),  
            new Class[] { MyInterface.class },  
            new ProxyClass(new MyInterfaceImpl()));  
        myintf.method();  
  
        myintf =  
            (MyInterface) Proxy.newProxyInstance(MyInterface.class.getClassLoader(),  
            new Class[] { MyInterface.class },  
            new ProxyClass(new MyInterfaceImpl2()));  
        myintf.method();  
    }  
}
```

before the method is called  
method  
after the method is called  
before the method is called  
outro método  
after the method is called

36

# Java Streams

## Demonstrative Examples

### map

```
public static List<String> transform7(List<String> collection) {  
    List<String> coll = new ArrayList<>();  
    for (String element : collection) {  
        coll.add(element.toUpperCase());  
    }  
    return coll;  
}  
  
public static List<String> transform8(List<String> collection) {  
    return collection.stream() // Convert collection to Stream  
        .map(String::toUpperCase) // Convert each element to upper case  
        .collect(Collectors.toList()); // Collect results to a new list  
}
```

## filter

```

public static List<String> transform7(List<String> collection) {
    List<String> newCollection = new ArrayList<>();
    for (String element : collection) {
        if (element.length() < 4) {
            newCollection.add(element);
        }
    }
}

public static List<String> transform(List<String> collection) {
    return collection.stream() // Convert collection to Stream
        .filter(value -> value.length() < 4) // Filter elements with length smaller than 4 characters
        .collect(Collectors.toList()); // Collect results to a new list
}

```

## flatMap

```

public static List<String> transform7(List<List<String>> collection) {
    List<String> newCollection = new ArrayList<>();
    for (List<String> subCollection : collection) {
        for (String value : subCollection) {
            newCollection.add(value);
        }
    }
}

public static List<String> transform(List<List<String>> collection) {
    return collection.stream() // Convert collection to Stream
        .flatMap(value -> value.stream()) // Replace list with stream
        .collect(Collectors.toList()); // Collect results to a new list
}

```

## max

```
public static Person getOldestPerson7(List<Person> people) {
    Person oldestPerson = new Person("", 0);
    for (Person person : people) {
        if (person.getAge() > oldestPerson.getAge()) {
            oldestPerson = person;
        }
    }
}

public static Person getOldestPerson(List<Person> people) {
    return people.stream() // Convert collection to Stream
        .max(Comparator.comparing(Person::getAge)) // Compares people ages
        .get(); // Gets stream result
}
```

## sum and reduce

```
public static int calculate7(List<Integer> numbers) {
    int total = 0;
    for (int number : numbers) {
        total += number;
    }
    return total;
}

public static int calculate(List<Integer> people) {
    return people.stream() // Convert collection to Stream
        .reduce(0, (total, number) -> total + number); // Sum elements with 0 as starting value
}
```

## filter and map

```
public static Set<String> getKidNames7(List<Person> people) {  
    Set<String> kids = new HashSet<>();  
    for (Person person : people) {  
        if (person.getAge() < 18) {  
            kids.add(person.getName());  
        }  
    }  
  
    public static Set<String> getKidNames(List<Person> people) {  
        return people.stream()  
            .filter(person -> person.getAge() < 18) // Filter kids (under age of 18)  
            .map(Person::getName) // Map Person elements to names  
            .collect(Collectors.toSet()); // Collect values to a Set  
    }  
}
```