/* EXEMPLO DE DEFINIÇÃO DE UMA CLASSE EM JAVA */

```
public class ContaVotos {

   // variáveis de instância

   private int part1;
   private int part2;
   private int part3;
```

part1	
part2	
part3	

```
   // construtores

   public ContaVotos() {
      part1 = 0; part2 = 0; part3 = 0;
   }

   public ContaVotos(int inicial) {
      part1 = inicial; part2 = inicial;
      part3 = inicial;
   }

   // métodos de instância

   public void conta1() {        // modificador
     part1 = part1 + 1;
   }

   public void conta2() {        // modificador
     part2++;
   }

   public void conta3() {        // modificador
     part3 += 1;
   }

   public void conta1(int xvotos) {// modificador
     part1 = part1 + xvotos;
   }

   // o mesmo para os outros dois contadores
```

```
    // algumas questões sobre o estado interno

    int maisVotos = 0;
    maisVotos = cv1.comMaisVotos();
    System.out.println("Mais votos: " + maisVotos);

/* ou, simplesmente */

    System.out.println("Mais votos: " +
                       cv1.comMaisVotos());


    // e assim sucessivamente sem interacção com o
    // utilizador para já !!


  }
}
```

/* EXEMPLO DE DEFINIÇÃO DE UMA CLASSE DE TESTE OU PROGRAMA PRINCIPAL PARA TESTE */

```
import java.lang.*; // redundante por implícita!
import java.util.*;
public class TesteContaVotos {

  public static void main(String[] args){
   // método especial onde se insere o código
   // do programa principal

   // Declaração e criação de dois Contadores de
      Votos

   ContaVotos cv1 = new ContaVotos();
   ContaVotos cv2 = new ContaVotos(10000);

   // Teste de valores iniciais

  System.out.println("P1/1 = " + cv1.getConta1());
  System.out.println("P1/2 = " + cv2.getConta1());
  System.out.println("P2/1 = " + cv1.getConta2());
  System.out.println("P2/2 = " + cv2.getConta2());

   // ou talvez melhor, usando toString()

   System.out.println("C1 = " + cv1.toString());
   System.out.println("C2 = " + cv2.toString());

   // algumas modificações

   cv1.conta1(100); cv1.conta2(200);
   cv1.conta3(300);

   // consulta do estado actual do 1° contador

   System.out.println("C1 = " + cv1.toString());

   // outras modificações

   cv2.conta1(1000); cv2.conta2(2000);
   cv2.conta3(3000);

   System.out.println("C2 = " + cv2.toString());
```

```
public int getConta1() {    // interrogador
  return part1;             // ou selector
}

public int getConta2() {    // interrogador
  return part2;
}
```

....

```
// outros métodos de instância interessantes

public int comMaisVotos() { // um tem mais !
  int mais;
  if( (part1 > part2) && (part1 > part3) )
       mais = 1;
  else // ou part2 ou part3 > part1
     if (part2 > part3)
          mais = 2;
     else
          mais = 3;
  return mais;
}

public int totaldeVotos() {
  return part1 + part2 + part3;
}

public String toString() {
  return new String("Part1 = " + part1 +", "+
                    "Part2 = " + part2 +", "+
                    "Part3 = " + part1 +", ");
}

}
```

EXEMPLO DE CODIFICAÇÃO PROIBIDA SEGUNDO O PRINCÍPIO DO ENCAPSULAMENTO

FUNDAMENTOS DE PROGRAMAÇÃO EM JAVA 2

```
public class Estudante
{
   //Atributos
   private String nome;
   private int [] notas;
   private float media;

   //Construtor da classe, promove a inicialização dos atributos
   public Estudante() {
      System.out.println();
      System.out.print("Nome do estudante: ");
      nome = Le.umaString();
      System.out.print("Quantas notas? ");
      int numNotas = Le.umInt();
      //Cria a tabela notas com o número de elementos necessário
      notas = new int[numNotas];
      for (int i=0;i<numNotas;i++) {
         System.out.print("Nota "+(i+1)+" deste aluno: ");
         notas[i] = Le.umInt();
      }
      media = calculaMedia();
   }

   //Escreve os dados de um estudante
   public void imprimeEstudante() {
      System.out.print("As notas de "+nome+" são: ");
      for (int i=0;i<notas.length;i++) {
         System.out.print(notas[i]+ " ");
      }
      System.out.println();
      System.out.println("A média é "+media);
   }

   //Método de acesso externo à média
   public float getMedia() {
      return media;
   }
```

input/output junto com camada computacional !!

NO EXEMPLO, ATÉ NOS CONSTRUTORES SÃO INTRODUZIDAS INSTRUÇÕES DE INPUT/OUTPUT !!

CONCLUSÃO: *ATENÇÃO AO QUE APARECE NOS LIVROS QUE NÃO TÊM PREOCUPAÇÕES QUANTO A METODOLOGIAS DE PROGRAMAÇÃO.*

MÉTODOS COMPLEMENTARES USUAIS

```
String toString();                 // Uma String representativa do objecto
boolean equals(Object obj);        // Igualdade de objectos do mesmo tipo
Object clone();                    // Cópia "deep" do objecto original
```

Object = classe genérica compatível com todas as outras classes

- **Estes métodos possuem definições mais ou menos "standard", apenas dependentes das características de estrutura da classe onde estão a ser definidos.**

EQUALS

```
boolean flag = t1.equals(t2);
```

```
public boolean equals(Object obj) {
    if ((obj != null) && (obj instanceof CLASSE))
        return < algoritmo de comparação >
    else
        return false;
}
```

Exemplo para classe Contador:

```
public boolean equals(Object obj) {
    if ((obj != null) && (obj instanceof Contador))
        return this.conta == obj.getConta();
    else
        return false;
}
```

toSTRING

```
String s = c1.toString();

public String toString() {
    StringBuffer s = new StringBuffer();
    s.append("------------------------------\n");
    s.append(var. de instância de tipo simples);
    ................
    s.append(var. instância de tipo ref.toString());
    .................
    return s.toString();
}

public String toString() {
    StringBuffer s = new StringBuffer();
    s.append("Contador: \n");
    s.append("Valor = ", conta);
    s.append("\nTotal = ", total);
    return s.toString();
}
```

CLONE (cópia do receptor)

```
public Object clone(){
    return new construtor da classe(......);
}

public Object clone(){
    return new Contador(conta, total);
}

public Object clone(){
    return new Circulo(centro, raio);
}
```

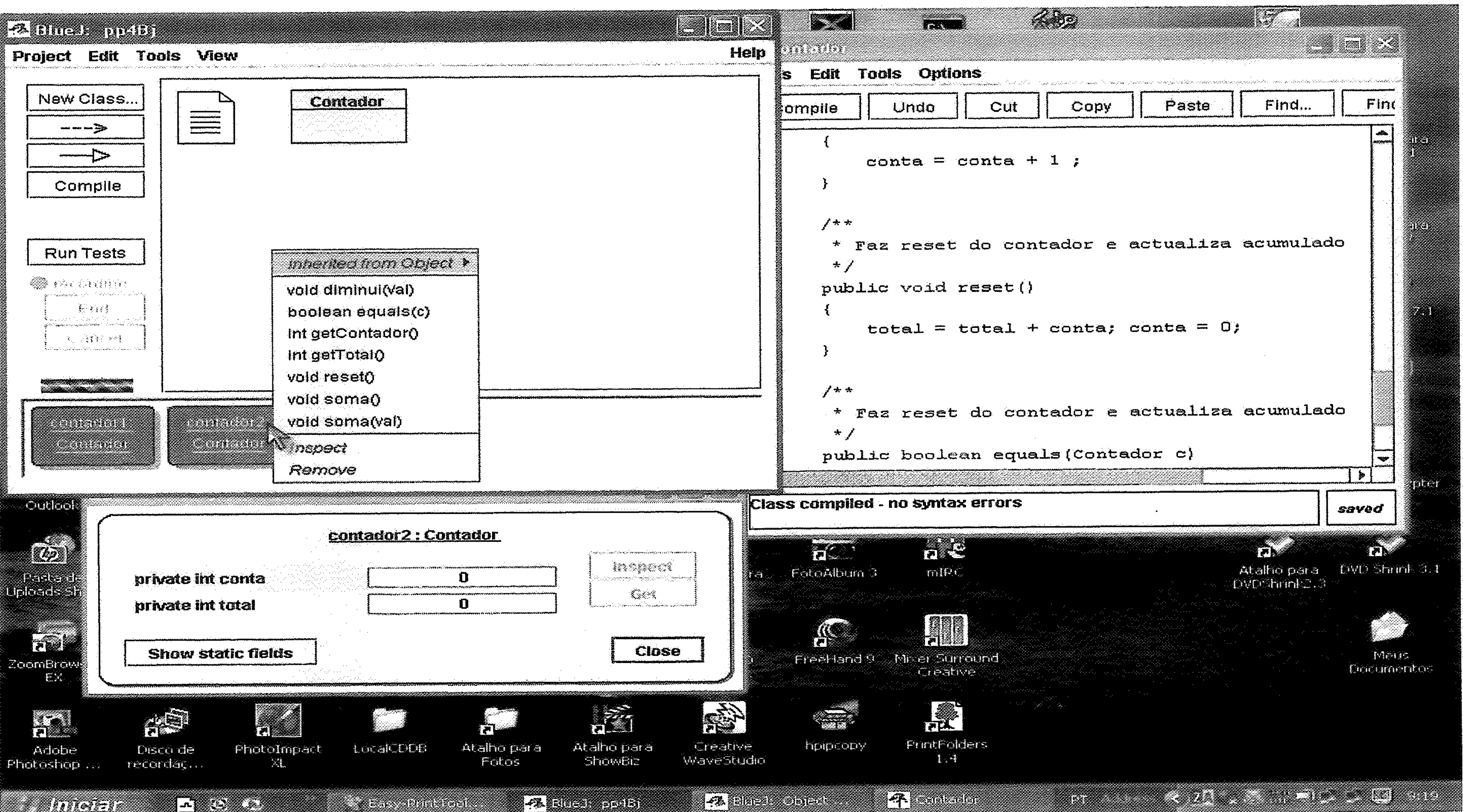
Utilização (sempre casting de Object -> para a classe) !!

```
Triangulo t1 = (Triangulo) t2.clone();
Circulo c2 = (Circulo) c1.clone()
Circulo c3 = ((Circulo) c2.clone())
double r = ((Circulo) c2.clone()).getRaio();
```

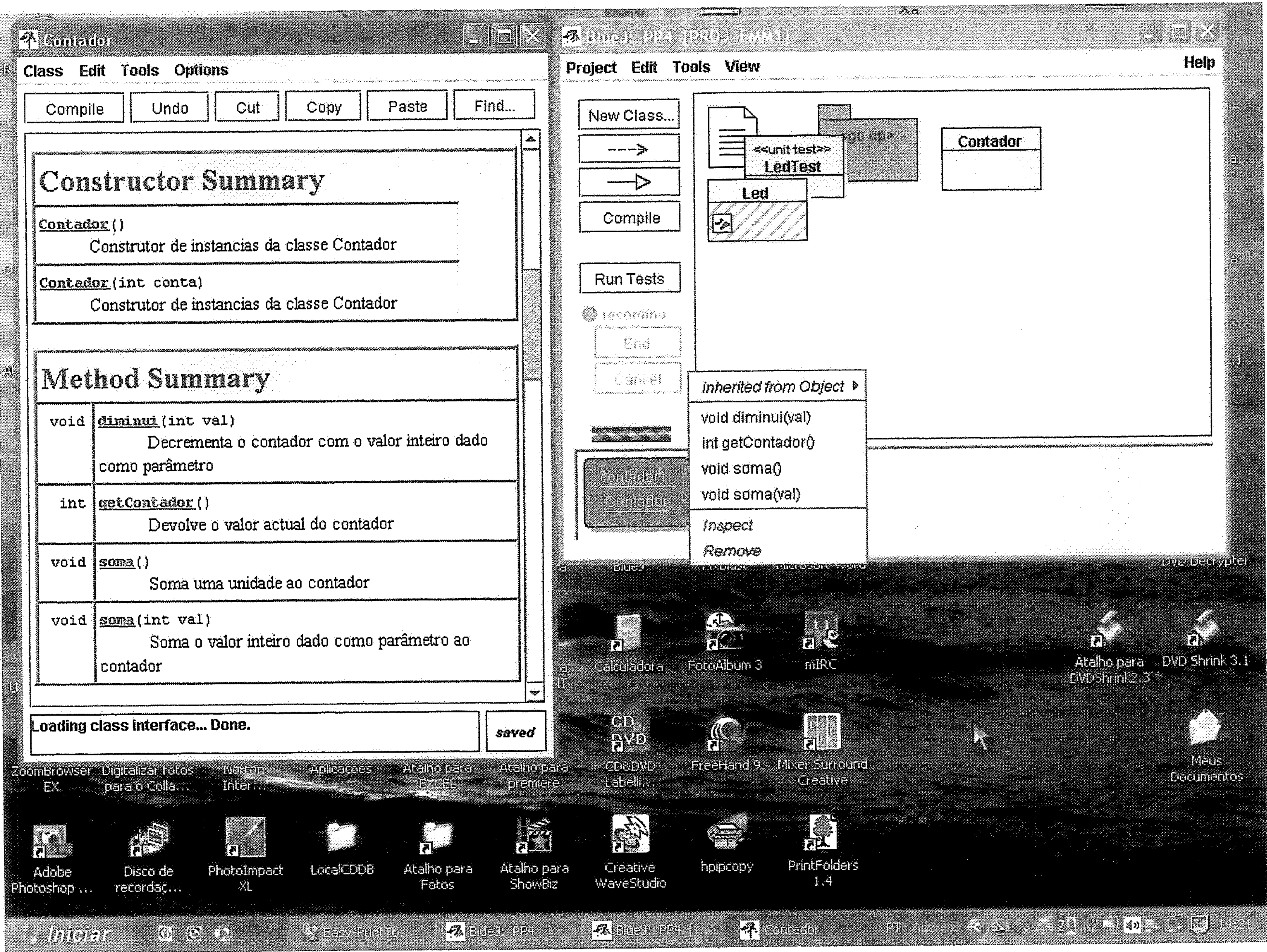
Erros comuns:

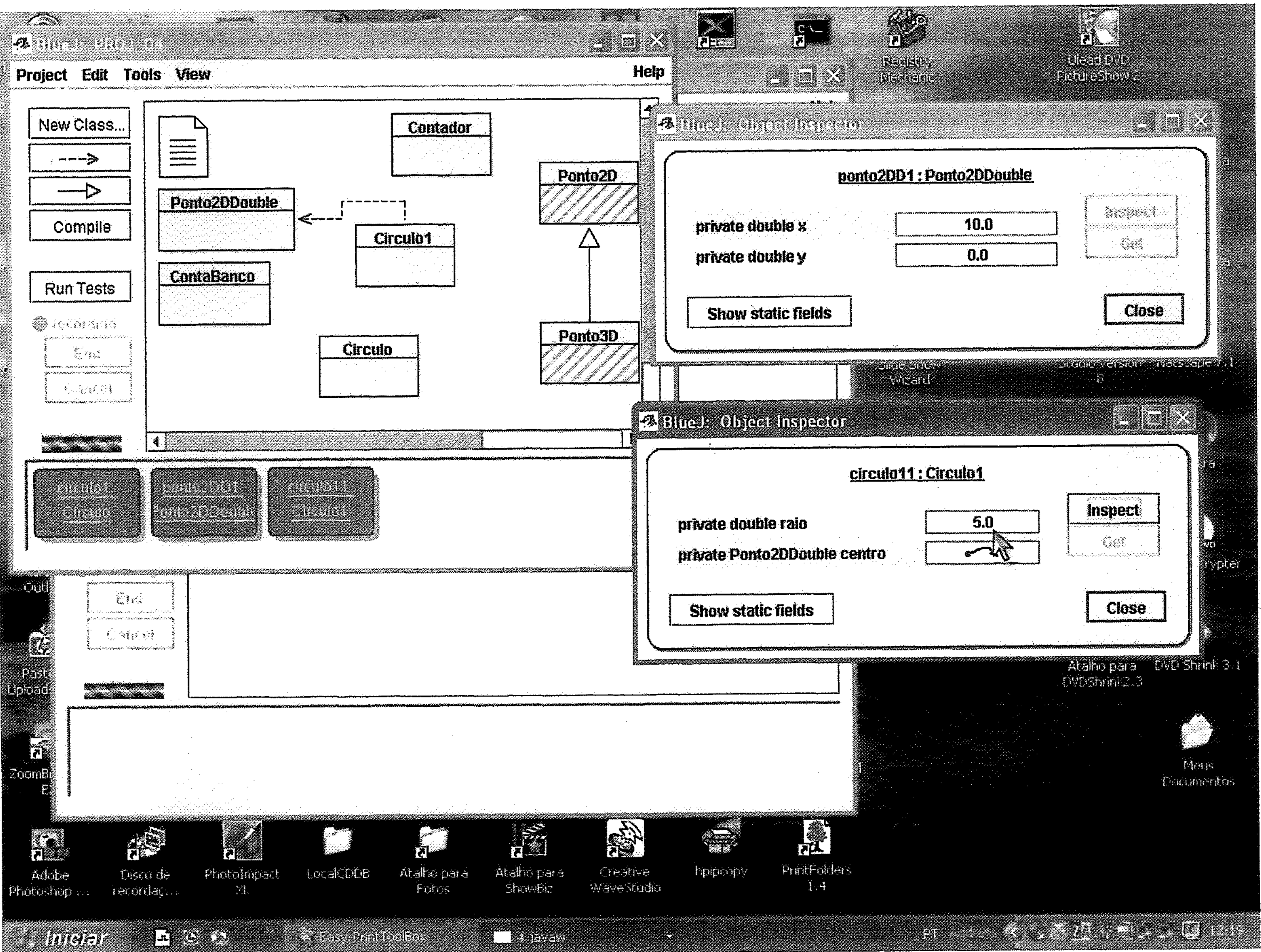
```
Circulo c2, c3;
c2 = new Circulo(10.0, 15.0, 5.0);
Circulo c3 = c2.clone(); // ERRO DE COMPILAÇÃO !!

double r = (c2.clone()).getRaio(); // ERRO

// c2.clone() deveolve uma instância de Object e não de
// Circulo. Object não responde à mensagem getRaio() !!
```

AMBIENTE BLUEJ



EDIÇÃO, COMPILAÇÃO, INSPECÇÃO E DEBUGGING DE JAVA


Contador
Class Edit Tools Options
Compile
Undo
Cut
Copy
Paste
Find...
Constructor Summary
Contador()
Construtor de instancias da classe Contador
Contador(int conta)
Construtor de instancias da classe Contador
Method Summary
void diminui(int val)
Decrementa o contador com o valor inteiro dado como parâmetro
int getContador()
Devolve o valor actual do contador
void soma()
Soma uma unidade ao contador
void soma(int val)
Soma o valor inteiro dado como parâmetro ao contador
Loading class interface... Done.
saved
Project Edit Tools View
Help
New Class...
Compile
Run Tests
<<unit test>>
LedTest
Led
Contador
inherited from Object
void diminui(val)
int getContador()
void soma()
void soma(val)
Inspect
Remove
Calculadora
FotoAlbum 3
mIRC
Atalho para DVDShrink2.3
DVD Shrink 3.1
CD&DVD Labell...
FreeHand 9
Mixer Surround Creative
Meus Documentos
ZoomBrowser EX
Digitalizar fotos para o Colla...
Aplicações
Atalho para EXCEL
Atalho para premiere
Adobe Photoshop ...
Disco de recordaç...
PhotoImpact XL
LocalCDDB
Atalho para Fotos
Atalho para ShowBiz
Creative WaveStudio
hpipcopy
PrintFolders 1.4
Iniciar
BlueJ: PP4
Contador


Project Edit Tools View
Help
New Class...
Compile
Run Tests
Contador
Ponto2DDouble
Circulo1
ContaBanco
Circulo
Ponto2D
Ponto3D
BlueJ: Object Inspector
ponto2DD1 : Ponto2DDouble
private double x
10.0
private double y
0.0
Inspect
Get
Show static fields
Close
BlueJ: Object Inspector
circulo11 : Circulo1
private double raio
5.0
private Ponto2DDouble centro
Inspect
Get
Show static fields
Close
Iniciar

BlueJ: PROJ_04

Project Edit Tools View Help

New Class...

Compile

Ponto2DDouble

Contador

Circulo1

Circulo

Ponto2D

Led

Run Tests

recording

End

Cancel

BlueJ: Object Inspector

circulo12 : Circulo1

private double raio 8.0

private Ponto2DDouble centro

Inspect

Get

Show static fields

Close

BlueJ: Object Inspector

circulo12.centro : Ponto2DDouble

private double x 92.0

private double y 94.0

Inspect

Get

Show static fields

Close

ponto2DD1 : Ponto2DDouble

circulo11 : Circulo1

circulo12 : Circulo1

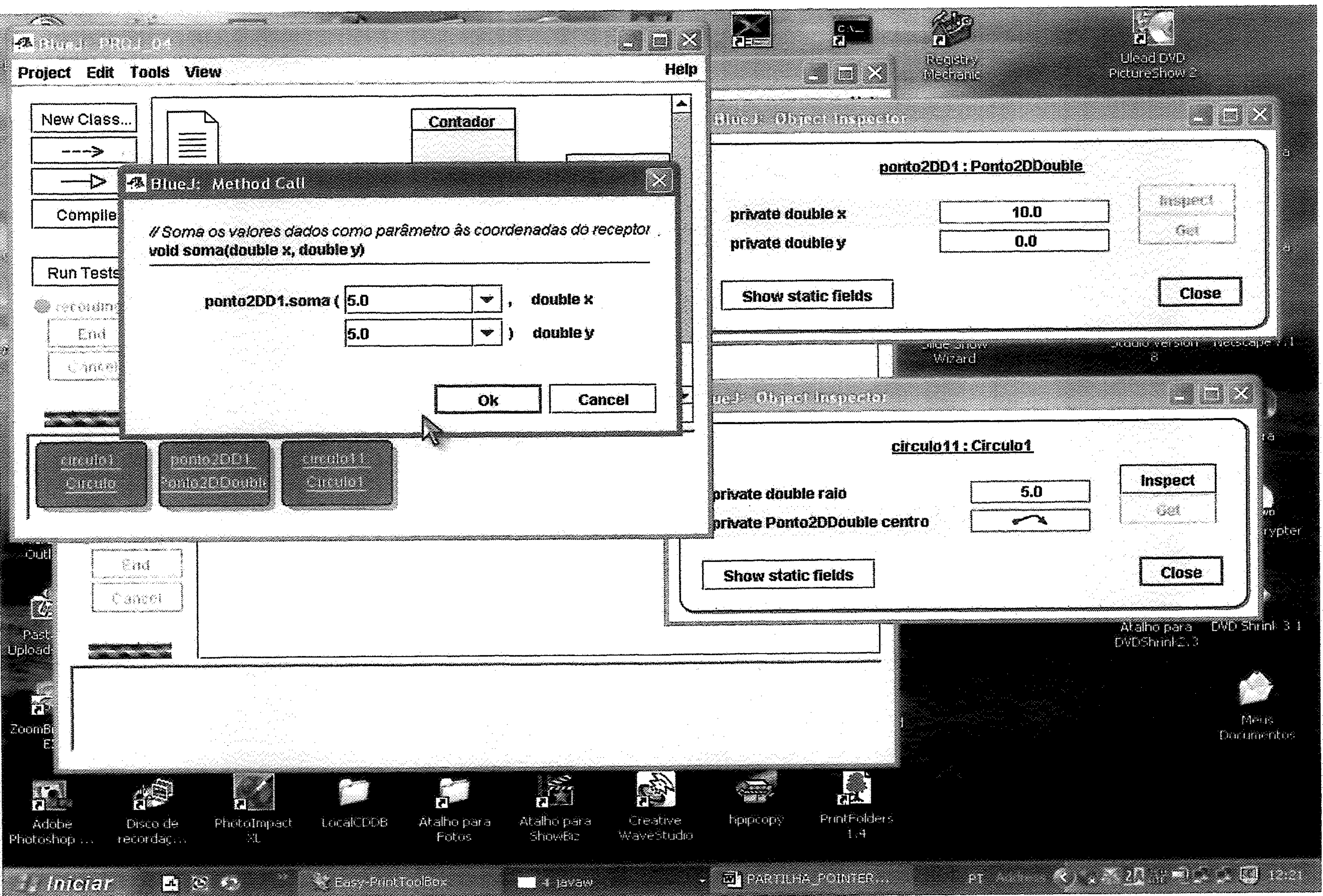
Iniciar

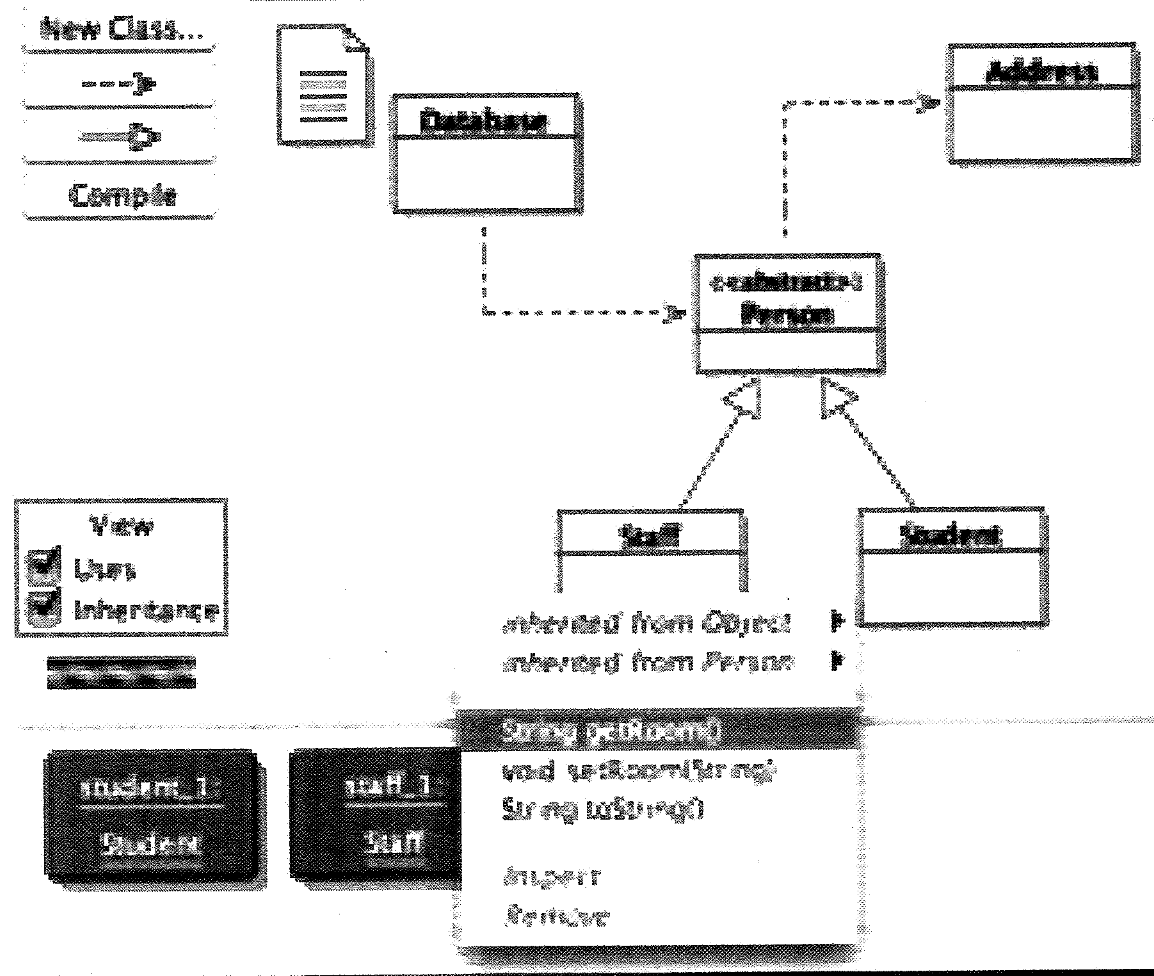
Easy-PrintToolBox

PARTILHA_POINTER...

PT

12:35


Project Edit Tools View
Help
New Class...
Compile
Run Tests
Contador
BlueJ: Method Call
// Soma os valores dados como parâmetro às coordenadas do receptor
void soma(double x, double y)
ponto2DD1.soma (5.0 , double x
5.0) double y
Ok
Cancel
BlueJ: Object Inspector
ponto2DD1 : Ponto2DDouble
private double x
10.0
private double y
0.0
Show static fields
Close
circulo11 : Circulo1
private double raio
5.0
private Ponto2DDouble centro
Inspect
Show static fields
Close
Iniciar



Class
Edit
Tools
Options
Compile
Undo
Cut
Copy
class Student extends Person
public Student()
Project
Edit
Tools
View
Help
New Class...
Compile
Database
Address
Person
Staff
Student
View
Uses
Inheritance
inherited from Object
inherited from Person
void setRoom(String)
String toString()

Contador

Class Edit Tools Options

Compile | Undo | Cut | Copy | Paste | Find... | Find Next | Close | Interface

Constructor Summary

Contador()   Construtor sem parametros (coloca tudo a zero).
Contador(int conta)   Construtor de instancia

Method Summary

void	**diminui**(int val) Decrementa o contador com o valor inteiro dado como parâmetro
int	**getContador**() Devolve o valor actual do contador
int	**getTotal**() Devolve o total acumulado.
boolean	**igual**(Contador c) Testa se os dois contadores são iguais
void	**reset**() Faz reset do contador e actualiza acumulado
void	**soma**() Soma uma unidade ao contador
void	**soma**(int val) Soma o valor inteiro dado como parâmetro ao contador
java.lang.String	**toString**() Converte estado interno para String

Loading class interface... saved

Iniciar Easy-PrintToolBox BlueJ: pp4Bi04 Contador PT 1:30

```
public class Circulo {

  /**
   * Constante acessível a todas as instancias - de CLASSE
   */

  private static final double PI = 3.1415926535897;

  /**
   * Variáveis de instância
   * sendo x e y as coordenadas do centro do círculo
   */

  private double x, y, raio;

  /**
   * Construtores de circulos
   */

  public Circulo() { x = 0.0; y = 0.0; raio = 1.0; }
  public Circulo(double raio) { this.raio = (raio <= 0.0 ? 1.0 : raio);
}
  public Circulo(double x, double y, double raio) {
      this.x = x; this.y = y; this.raio = raio;
  }

  /**
   * Métodos de instância
   */

  public double getX() {return x;}
  public double getY() {return y;}
  public double getRaio() {return raio;}
  public double area() { return PI*raio*raio; }
  public double perimetro() { return 2*PI*raio; }
  public void aumentaRaio(double rx) { raio = raio + rx;}

  public boolean igual(Circulo c) {
       return (x == c.getX() && y ==c.getY() && raio== c.getRaio());
  }

  public String toString() {
      StringBuffer s = new StringBuffer("Circulo com centro em ");
      s.append("Ponto( " + x + ", " + y + " )");
      s.append(" e raio = a " + raio);
      return s.toString();
  }
}
```

```
/**
 * Classe que define pontos bidimensionais (2D)
 * com abcissa e ordenada reais (double).
 *
 * @author Paradigmas de Programação IV
 * @version 1/03-2004
 */

public class Ponto2DDouble {

   // Variáveis de instancia
   private double x;
   private double y;

   /**
    * Construtores
    */

   public Ponto2DDouble() {
       // initializa as variáveis a 0.0
       x = 0.0;
       y = 0.0;
   }

   public Ponto2DDouble(double x, double y) {
       this.x = x; this.y = y;
   }

   // Dado que em Java construtores podem invocar construtores da
   // mesma classe, poderíamos definir Ponto2DDouble () como :
   //       public Ponto2DDouble() {
   //           this(0.0, 0.0) ;
   //       }
   // ou seja, usando o construtor Ponto2DDouble(x, y) com os
   // parametros a tribuir a x e y iguais a 0.0

   /**
    * Métodos de instância
    */

   /**
    * Dá como resultado a coordenada em x do ponto receptor
    */

   public double getX() {
      return x;
   }

   /**
    * Dá como resultado a coordenada em y do ponto receptor
    */
```

```
    public double getY() {
       return y;
    }

    /**
     * Incrementa a coordenada em x do receptor de deltax unidades
     */

    public void incX(double deltax) {
       x = x + deltax;
    }

    /**
     * Incrementa a coordenada em y do receptor de deltay unidades
     */

    public void incY(double deltay) {
       y = y + deltay;
    }

    /**
     * Testa se o ponto receptor e o ponto parametro sao iguais.
     * Dois pontos 2D serão iguais se tiverem a mesma abcissa e a
     * mesma ordenada.
     */

     public boolean igual(Ponto2DDouble ponto) {
       return (x == ponto.getX() && y == ponto.getY());
    }

     /**
     * Converte a representação interna do receptor numa string
     */

    public String toString() {
        StringBuffer s = new StringBuffer();
        s.append("Ponto2dDouble( ");
        s.append(x + ", "); s.append(y + " )"); s.append('\n');
        return s. toString();
    }

    /**
     * Soma as coordenadas do parametro às do receptor
     */

    public void soma(Ponto2DDouble ponto) {
        x = x + ponto.getX();
        y = y + ponto.getY();
    }

    /**
     * Soma os valores dados como parâmetro às coordenadas do receptor
     */
```

```
    public void soma(double x, double y) {
       this.x = this.x + x;
       this.y = this.y + y;
    }


   /**
    * Soma as coordenadas do ponto parâmetro às coordenadas do
    * receptor e devolve um novo ponto resultado desta soma, que
    * não modifica o estado do ponto receptor.
    */

    public Ponto2DDouble somaPontos(Ponto2DDouble p) {

        // usando variáveis auxiliares desnecessárias teríamos
        // double cx = x + p.getX();
        // double cy = y + p.getY();
        // Ponto2DDouble paux = new Ponto2DDouble(cx, cy) ;
        // return paux;

        // o que é equivalente a, simplesmente :
        return new Ponto2DDouble(x + p.getX(), y + p.getY());
    }
}
```

AGREGAÇÃO - COMPOSIÇÃO DE CLASSES

Mecanismo que permite que classes pré-definidas, sejam classes de SDK ou classes criadas pelo utilizador, possam ser usadas na criação de novas classes, em geral sob a forma de variáveis de instância que devem ser instâncias de tais classes e, naturalmente, satisfaçam (da forma mais adequada) todos os vários comportamentos requisitados.

CIRCULO

Ponto centro;

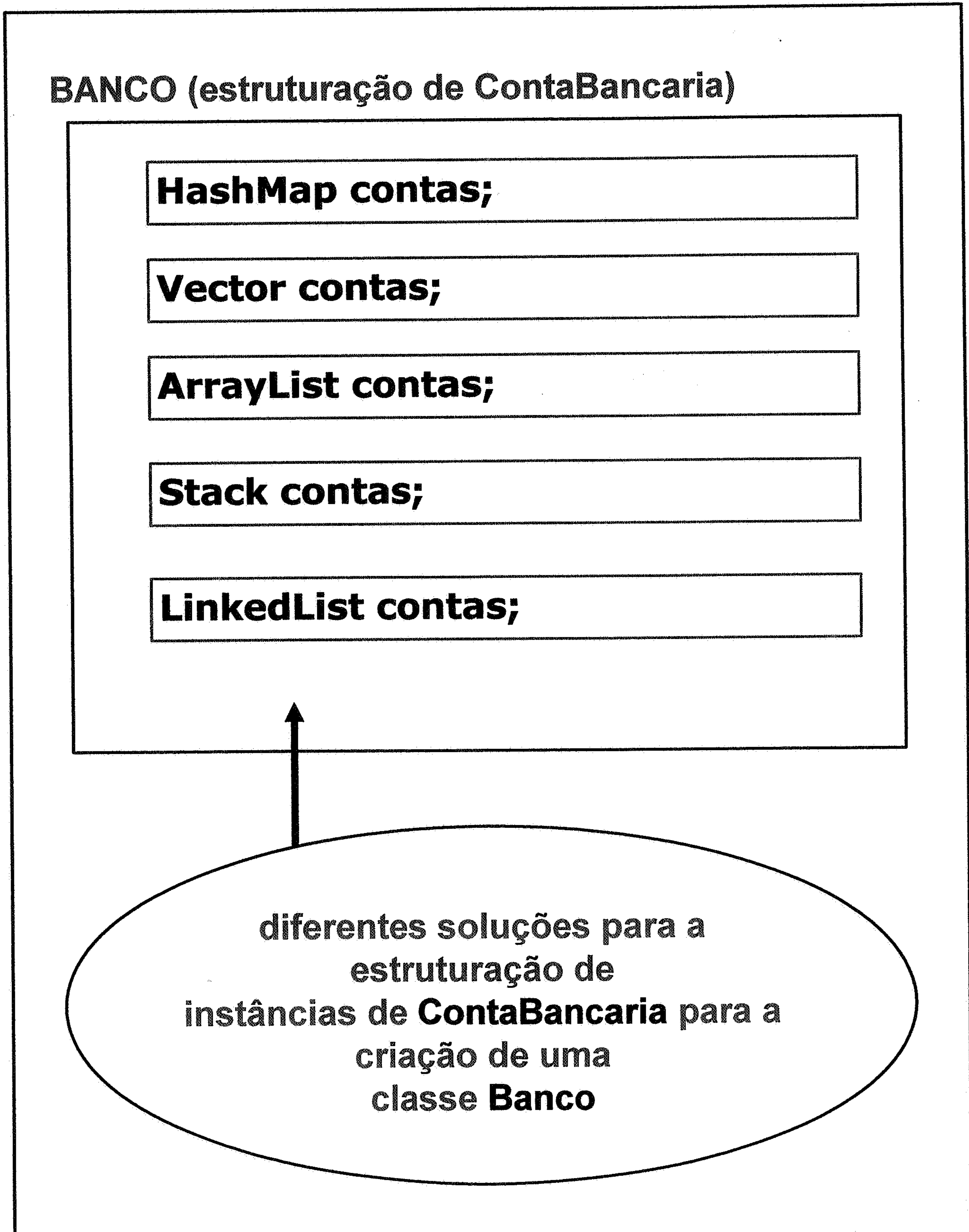
double raio;

```
public class Circulo1 {

   /**
    * Constante acessível a todas as instancias - de CLASSE
    */

   private static final double PI = 3.1415926535897;

   /**
    * Variáveis de instância
    * centro = ponto 2D que é o centro do circulo
    * raio = raio do círculo
    */

   private double raio;
   private Ponto2DDouble centro;

   /**
    * Construtores de circulos
    */

   public Circulo1() { centro = new Ponto2DDouble(); raio = 1.0; }
   public Circulo1(double raio) {
        this.raio = (raio <= 0.0 ? 1.0 : raio);
        centro = new Ponto2DDouble();
   }
   public Circulo1(double x, double y, double raio) {
       this.raio = raio;
       centro = new Ponto2DDouble(x, y);
   }

   public Circulo1(Ponto2DDouble centro, double raio) {
       this.raio = raio; this.centro = centro;
   }

   /**
    * Métodos de instância
    */

   public double getRaio() {return raio;}
   public Ponto2DDouble getCentro() {return centro;}
   public double area() { return PI*raio*raio; }
   public double perimetro() { return 2*PI*raio; }
   public void aumentaRaio(double rx) { raio = raio + rx;}

   public boolean igual(Circulo1 c) {
        return (raio == c.getRaio() && centro.igual(c.getCentro()));
   }

   public String toString() {
       StringBuffer s = new StringBuffer("Circulo com centro em ");
       s.append(centro.toString());
       s.append(" e raio = a "); s.append(raio);
```

```
        return s.toString();
    }
}
```

AGREGAÇÃO - COMPOSIÇÃO DE CLASSES

Mecanismo que permite que classes pré-definidas, sejam classes de SDK ou classes criadas pelo utilizador, possam ser usadas na criação de novas classes, em geral sob a forma de variáveis de instância que devem ser instâncias de tais classes e, naturalmente, satisfaçam (da forma mais adequada) todos os vários comportamentos requisitados.



PACKAGES E CLASSES IMPORTANTES DE JAVA2

PACKAGE java.util (via *import java.util.*;*)

- Contém um conjunto de classes que implementam, das mais diferentes formas, as mais importantes **estruturas de dados** de que necessitaremos ao longo do curso, sob a designação comum de **COLLECTIONS**.

Figure 23-1: The collection classes of the java.util package

COLLECTIONS MAIS IMPORTANTES:

- Listas: **ArrayList, Vector, Stack;**
- Conjuntos: **HashSet, TreeSet**;
- Tabelas de Hashing: **Hashtable, HashMap, TreeMap;**

CLASSES DE java.util QUE NÃO SÃO COLLECTIONS

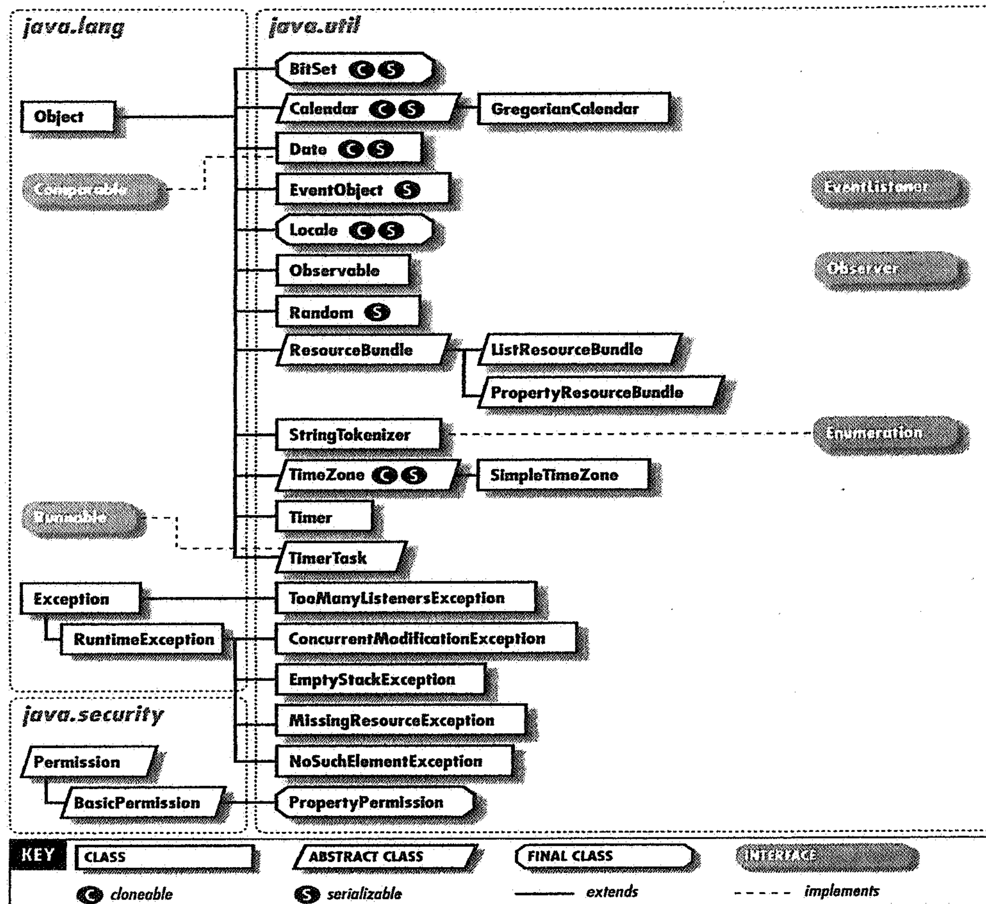


Figure 23-2: Other classes of the java.util package

CLASSES MUITO ÚTEIS:

- **Calendar, TimeZone, Date (deprecated);**
- **StringTokenizer;**
- **Random**

java.util

PACKAGE java.lang

- Contém um conjunto de classes que constituem elementos básicos da linguagem Java. Contém classes numéricas que compatibilizam os tipos simples com os referenciados (objectos) (cf. Integer versus int). Contém classes como Math que possuem uma enorme quantidade de métodos de classe disponíveis para os mais variados cálculos matemáticos. Possui a classe Class que representa a classe de implementação de qualquer classe de JAVA. Possui as classes para manipulação de "strings", em especial StringBuffer.
- **O package java.lang. não necessita de ser importado. Todas as suas classes estão disponíveis em qualquer contexto, dado ser um package básico da linguagem.**

Figure 12-1: The java.lang package

CLASSES IMPORTANTES de java.lang:

- Aritméticas: **Math, StrictMath, Integer, Float, Byte, etc;**
- Alfanuméricas: **Character, String, StringBuffer;**
- Outras: **System, Class, Thread.**

NOTA:

- O conhecimento destas classes, que são de grande utilidade permanente na construção de aplicações JAVA, passará por conhecermos exactamente o que cada uma oferece em termos de eficiência e gestão de espaço em memória (em especial para as diferentes colecções) e conhecer razoavelmente as suas APIs, que será a forma de as utilizar.

```
public class ArrayList extends AbstractList implements Cloneable, java.util.List, Serializable {
// Public Constructors
    public ArrayList();
    public ArrayList(int initialCapacity);
    public ArrayList(Collection c);
// Public Instance Methods
    public void ensureCapacity(int minCapacity);
    public void trimToSize();
// Methods Implementing List
    public boolean add(Object o);
    public void add(int index, Object element);
    public boolean addAll(Collection c);
    public boolean addAll(int index, Collection c);
    public void clear();
    public boolean contains(Object elem);
    public Object get(int index);
    public int indexOf(Object elem);
    public boolean isEmpty();                                              default:true
    public int lastIndexOf(Object elem);
    public Object remove(int index);
    public Object set(int index, Object element);
    public int size();
    public Object[ ] toArray();
    public Object[ ] toArray(Object[ ] a);
// Protected Methods Overriding AbstractList
    protected void removeRange(int fromIndex, int toIndex);
// Public Methods Overriding Object
    public Object clone();
}
```

Hierarchy: Object→ AbstractCollection(Collection)→ AbstractList(java.util.List(Collection))→ ArrayList(Cloneable, java.util.List(Collection), Serializable)

```java
import java.util.*;

/**
 * Stack implementada usando um ArrayList
 *
*/
public class StackAL {

    // Representação da stack de objectos

    private ArrayList stack;

    // Construtores

    public StackAL() {
        // initialise instance variables
        stack = new ArrayList();
    }

    public StackAL(Object o) {
        // initialise instance variables
        stack = new ArrayList(); stack.add(o);
    }

    public StackAL(Vector stkaux) {
        // initialise instance variables
        stack = new ArrayList(); stack.addAll(stkaux);
    }

    // Métodos de instância usuais para uma STACK

    /* Insere elemento no topo
     */

    public void push(Object o) {
        stack.add(o);
    }
```

Stack em ArrayList (1 de 2)

```java
// Determina se stack está vazia

public boolean empty() {
   return (stack.size() == 0);
}

// Determina comprimento da stack

public int size() {
   return stack.size();
}

/* Determina qual o topo da stack (se não estiver vazia !!)
   Se estiver vazia devolve null ⇨ não é muito boa programação
 */
public Object top() {
   return (stack.get(this.size()-1));
}

/* Remove o topo da stack (se não estiver vazia !!)
 */
public void pop() {
  stack.remove(this.size()-1);
}

/**
 * Inspect
 */
public ArrayList lookAtMe() {
  return (ArrayList) stack.clone();
}

}
```

Stack em ArrayList (2 de 2)

A HIERARQUIA DE CLASSES

Definida que está completamente a noção de classe e tendo sido já apresentadas duas formas de as classes se relacionarem entre si, cf. **usa** e **contém**, das quais a relação **contém,** ou composição, surge como um mecanismo muito importante de reutilização de classes já definidas, vamos agora analisar se é ou não vantajoso que outros relacionamentos possam ser definidos entre classes.

Vamos admitir que mais nenhuma forma de relacionamento entre classes é definida em PPO. Se tal acontecer, então as classes ao serem criadas podem usar outras classes por composição, mas, para além disso, são entidades isoladas umas das outras, todas posicionadas no mesmo espaço plano, conforme se procura ilustrar na figura seguinte:

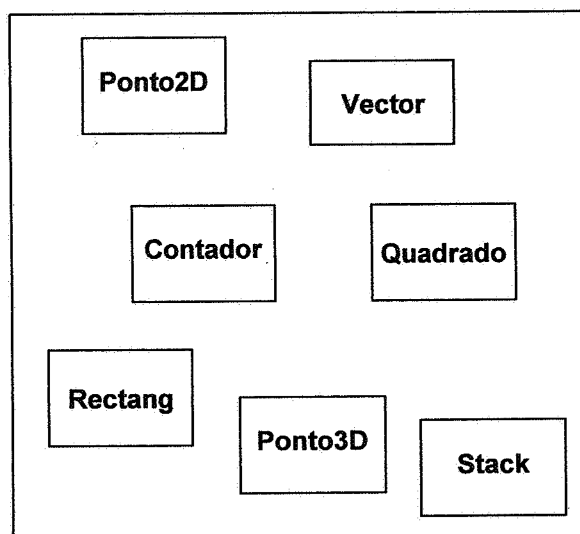


Fig. H.1 – Espaço plano de classes

Ao observarmos as classes apresentadas na figura, é natural que relativamente a algumas delas não consigamos encontrar qualquer tipo de semelhança, afinidade ou qualquer outra particularidade comum. Tal é o caso, por exemplo, entre a classe Contador e a classe Vector, ou entre a classe Ponto2D e a classe Stack. Porém, que dizer das afinidades que poderíamos encontrar entre a classe Quadrado e a classe Rectang(ulo) ou entre as classes Ponto2D e Ponto3D?

Relativamente ao primeiro par, poderíamos pensar que se tivéssemos disponível a classe Rectang e necessitassemos agora de criar a classe Quadrado, esta poderia usar praticamente toda a estrutura de representação e todo o comportamento que serviram para definir rectângulos, sendo os quadrados apenas um *caso especial* de rectângulo, já que possuem uma restrição relativa às dimensões dos seus lados. Ou seja, a classe pretendida, Quadrado, é uma *especialização* da classe Rectang já existente, *sendo as diferenças entre si mínimas.*

Sendo as diferenças entre duas classes muito menores que as suas semelhanças em estrutura e comportamento, em particular existindo a noção de que uma é apenas um caso particular ou especial da outra, faria todo o sentido que algum mecanismo de reutilização pudesse ser usado por forma a que a definição da nova classe fosse feita sem grande esforço a partir da classe semelhante já existente.

Claro que não existindo tal mecanismo implementado num espaço de classes que é plano, tal como o da figura acima, então a definição da nova classe Quadrado só poderia ser feita de duas formas possíveis:

1. *Usando um mecanismo de "copy&paste&edit" a partir de Rectang;*
2. *Usando em Quadrado uma variável de instância que é da classe Rectang, ou seja, agregando um rectângulo dentro de um quadrado.*

Ainda que em ambas as soluções exista de facto reutilização, no primeiro caso via o mecanismo mais básico possível e no segundo via composição, como é fácil de compreender ambas as soluções são muito pobres. Acresce ainda o facto de que, enquanto que no exemplo foi de imediato afirmado que as classes possuem grandes afinidades, o custo de saber num sistema plano de classes se existe alguma classe semelhante àquela que pretendemos criar é, como se compreende, imenso, já que as classes não possuem qualquer mecanismo particular de classificação a não ser o seu próprio identificador. Tal constatação deveria ser feita por análise da sua API ou, na pior das hipóteses, do seu código fonte.

Torna-se portanto muito importante criar um mecanismo que facilite a definição de classes à custa de classes existentes, que não por composição, mas antes baseado nas noções de similaridade e especialização ou particularização, tornando-se assim um mecanismo adicional de relacionamento entre classes para tais casos específicos que muitas vezes irão surgir.

Em PPO, a solução para este problema consistiu em introduzir um espaço para a definição de classes não plano mas **hierárquico**, ou seja, de tal forma que **todas** as classes existentes se encontram hierarquicamente relacionadas entre si.

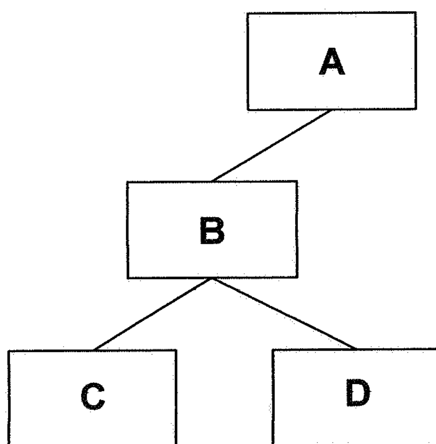


Fig. H.2. – Hierarquia de classes

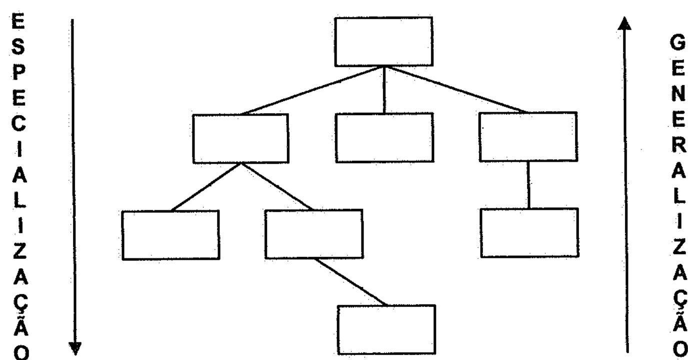
Na Fig. H.2. apresenta-se uma parte da hierarquia total de classes, em particular a sub-hierarquia estabelecida entre as classes **A**, **B**, **C** e **D**. A classe **B** é **subclasse** directa de **A**, por

ocupar na hierarquia uma posição imediatamente inferior a **A**. A classe **A**, por seu lado, é **superclasse** de **B.** As classes **C** e **D** são subclasses (directas) de **B** e também subclasses (indirectas) de **A**. Assim, **B** é subclasse de **A** e superclasse de **C** e **D**. As classes **C** e **D** não possuem subclasses.

Em certas linguagens, tal como em JAVA, qualquer classe terá no máximo uma superclasse, enquanto que noutras, como em C++, uma classe pode ser subclasse de mais do que uma classe e, portanto, possuir mais do que uma superclasse.

Mas o que significa do ponto de vista semântico este posicionamento hierárquico das classes? Isto é, que significado atribuir ao facto de uma classe ser superclasse de outra, dado posicionar-se mais acima na hierarquia?

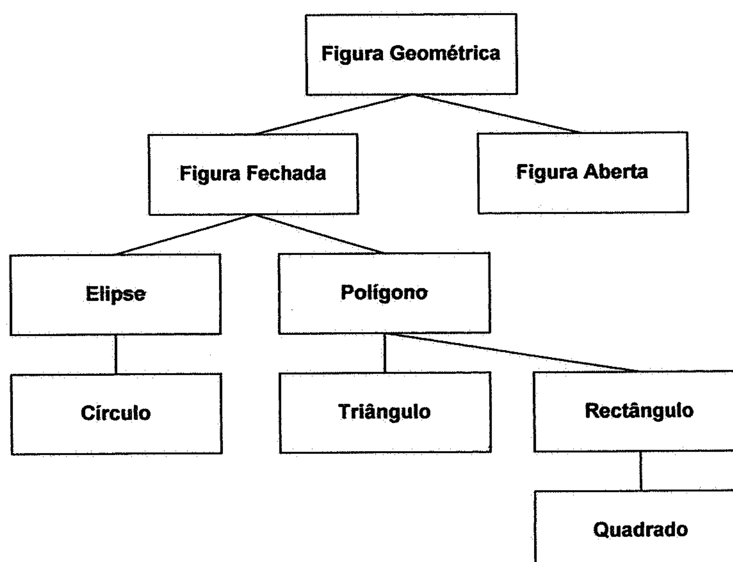
De facto, a hierarquia de classes em PPO é uma hierarquia de **especialização**, pelo que uma subclasse de uma dada classe é uma extensão ou refinamento desta, sendo por isso, em geral, mais detalhada ou refinada que a sua superclasse, seja por possuir mais estrutura de dados seja por possuir mais comportamento.



Semântica da hierarquia de classes

Assim, a classificação hierárquica correcta entre Rectang e Quadrado seria de facto tal que um Quadrado é subclasse de Rectang, porque tem comportamento semelhante, estrutura semelhante, mas é um caso particular, ou seja, uma **especialização**.

Como sabemos do estudo de outras taxonomias, a classificação do conhecimento é realizada do geral para o particular, seguindo uma semântica de **especialização**. A figura seguinte ilustra uma taxonomia deste tipo.



Taxonomia por especialização

No entanto, há que tomar em atenção que muitas destas taxonomias com que por vezes somos confrontados, possuem uma característica muito particular que as torna bastante diferentes da hierarquia com que teremos que lidar em PPO e que consiste no facto de serem taxonomias **fundamentalmente baseadas em atributos**, ou seja estrutura, mas que não contemplam comportamento.

A hierarquia de classes em PPO é uma hierarquia baseada em especialização, mas na qual a **especialização pode ser simultaneamente estrutural e comportamental**, ou seja, em que a subclasse pode necessitar de mais estruturas de dados que a sua superclasse para a sua representação e/ou pode necessitar de aumentar o conjunto de métodos que representam o comportamento da sua superclasse. Como se pode compreender facilmente, um aumento da estrutura de dados implica sempre que se devam acrescentar métodos (no mínimo os de acesso às variáveis criadas).

Podendo dentro desta perspectiva ver-se uma subclasse como sendo uma extensão, um aumento, em estrutura e comportamento, da sua respectiva superclasse, é ainda pertinente perguntar-se, por ser uma questão ainda não resolvida, de que forma a subclasse pode reutilizar a estrutura e o código da sua superclasse, que serão úteis para a sua definição, já que, por definição de especialização, ambas terão muito em comum.

A resposta a esta questão é dada em PPO pela introdução de um mecanismo que se designa por **herança** e que é um mecanismo em PPO fundamental à reutilização, à partilha de código e ao estabelecimento de um estilo de programação incremental, ou seja, baseado numa programação não do todo mas apenas de extensões ao que de imediato se tem como reutilizável.

O MECANISMO DE HERANÇA

Em PPO, entre as classes de uma hierarquia, em particular entre uma classe e a sua superclasse, é estabelecida uma relação de especialização que é automaticamente implementada através de um mecanismo de **herança**.

Este mecanismo automático de herança estabelece as seguintes propriedades entre uma **subclasse B** e a sua **superclasse A**:

1. ***B herda de A** todas as variáveis e métodos de instância que não sejam declarados como* `private`;
2. ***B pode definir** novas variáveis e novos métodos próprios;*
3. ***B pode redefinir** variáveis e métodos herdados;*
4. *A herança não se aplica a variáveis e métodos de classe.*

Subclasses são noutros contextos também designadas por **classes derivadas**.

A propriedade 1 especifica que a classe B **herda da sua superclasse tudo o que não seja definido como** `private`. Herdar significa em PPO, fundamentalmente, ter acesso a, poder invocar, em suma, poder usar o que foi definido na superclasse sem ter que definir de novo, ou seja, reutilizando a custo e esforço zero.

De tudo o que da superclasse é tornado acessível à subclasse por herança, devemos colocar de parte a questão das variáveis de instância, já que a estas, mesmo tendo acesso, nunca acederemos de forma directa, mas através dos métodos de acesso próprios. Assim, a compreensão e boa utilização do mecanismo de herança pode pois reduzir-se, numa primeira fase, à compreensão da herança comportamental, isto é, da *herança de métodos*.

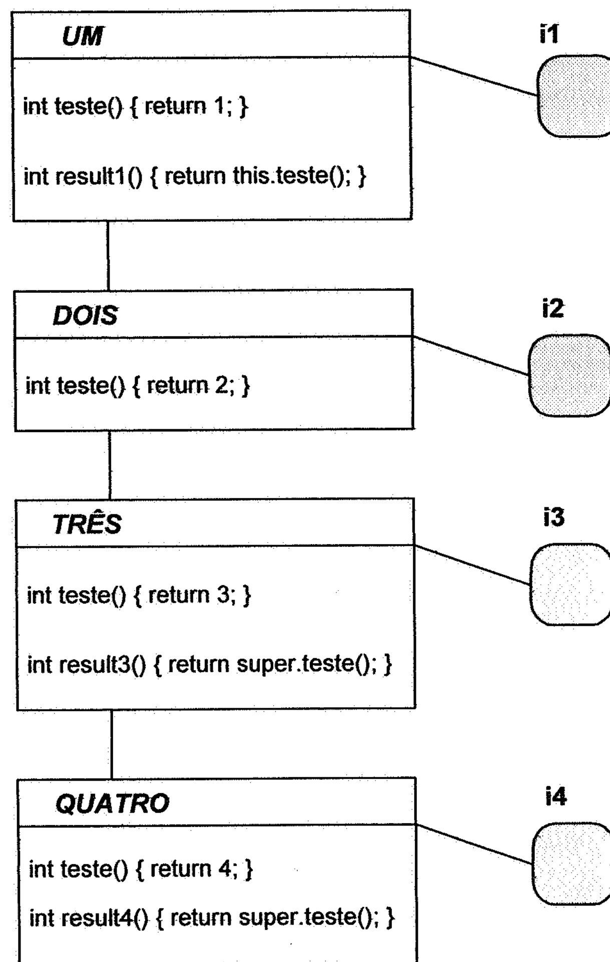


Fig. H.3 - Procura de métodos: hierarquia exemplo

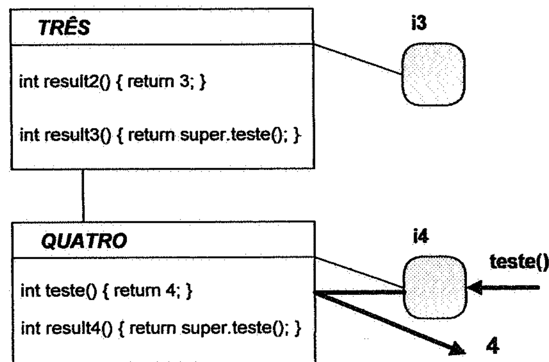
Vamos usar a hierarquia apresentada na figura para melhor compreendermos o mecanismo da herança.

Se uma subclasse **B** de uma classe **A** herda automaticamente os métodos de **A**, tal significa que qualquer instância de **B** vai poder responder não só às mensagens que correspondem aos métodos definidos na sua classe **B**, mas também, de forma automática, a mensagens que dizem respeito à activação de métodos que não estão definidos na sua classe **B**, mas na sua superclasse **A.** Assim, uma instância de uma dada classe, por herança, passa a poder receber e a ser capaz de responder não só às mensagens correspondentes à sua própria API, mas também às mensagens que correspondem à activação dos métodos herdados.

Por outro lado, **a herança é transitiva**, isto é, a própria classe **A** herda da sua superclasse e esta da sua. A única classe que não vai herdar de nenhuma outra será a classe que se irá posicionar no topo da hierarquia.

Deste modo, o conjunto real de mensagens a que uma instância de uma dada classe vai ser capaz de responder é formado pelo conjunto das mensagens que activam os seus métodos locais (definidos na sua classe), às quais se somam as mensagens que correspondem a todos os métodos herdados de todas as suas superclasses, até ao topo da hierarquia.

Por exemplo, na Figura H.3 a classe `Quatro` define os métodos de instância `int teste()` e `int result4`. Porém, a classe `Quatro` herda directamente da classe de nome `Três` os métodos `int result2()` e `int result3()`.

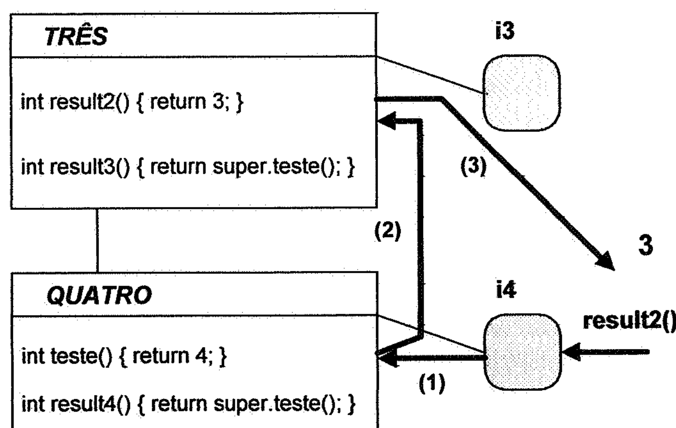


Tratamento da mensagem `teste()`

Assim sendo, à instância de `Quatro` identificada por **`i4`**, poderemos enviar, pelo menos, as mensagens: `teste()`, `result2()`, `result3()` e `result4()`. Vamos então ver o que acontece operacionalmente quando estas mensagens são enviadas à instância.

Caso a mensagem enviada seja `teste()`, o código do método `teste()`, dado ser local e não herdado, é de imediato encontrado na classe da instância receptora da mensagem, sendo executado e devolvendo o resultado 4, cf. figura.

Porém, se a mensagem enviada é `result2()`, a procura do código deste método na classe da instância receptora falha, dado este não ser um método definido em tal classe. Porém, o algoritmo de procura tem que respeitar o mecanismo de herança, pelo que vai agora procurar o método, sucessivamente, em todas as superclasses de `Quatro` até o encontrar, em cujo caso o executa, ou até não ter mais classes para procurar, neste caso sendo gerado um erro.



Tratamento da mensagem `result2()`

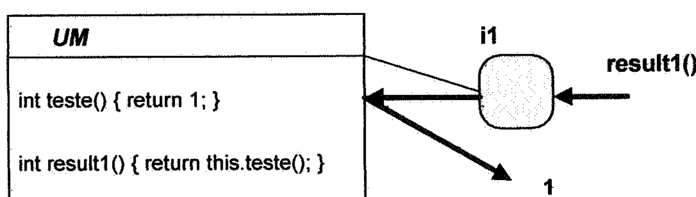
No exemplo, o método `result2()` é encontrado na classe `Três` e executado o seu código que dá como resultado 3. O exemplo permite-nos compreender também porque razão o mecanismo de herança é um mecanismo de partilha e reutilização de código. De facto, para

que a instância da classe `Quatro` possa responder à mensagem `result2()` que não está definida na sua classe, não foi necessário copiar o código de tal método para a classe `Quatro`.

O mecanismo de herança, em particular através do respectivo algoritmo de procura de métodos, permite que a instância da classe `Quatro` tenha acesso ao código do método, do qual apenas existe uma cópia na classe `Três`, mas que, deste modo, é partilhável por todas as classes que o herdem. Há reutilização pelo facto de que tal código foi apenas definido uma vez, mas pode ser também usado por todas as classes que o herdam ou venham a herdar ao serem criadas como subclasses desta.

No entanto, e consultando de novo a Figura H.3, verificamos que há mais métodos que foram herdados pela classe `Quatro`, e que, portanto, há mais mensagens a que as instâncias da classe devem poder responder. O método `result1()`, definido na classe `UM`, é também herdado pela classe `Quatro`.

Vamos começar por comparar o resultado do envio da mensagem `result1()` a **i1** e a **i4**.



Tratamento de `i1.result1()`

Quando a instância da classe `UM` identificada pela variável **i1** recebe a mensagem `result1()`, procura e encontra de imediato na sua classe o código do método correspondente à mensagem. O código do método, no entanto, possui a expressão `this.teste()`. A referência `this`, como vimos já, é uma forma anónima de se poder referir o receptor da mensagem, qualquer que ele seja, neste caso para que lhe seja enviada a mensagem `teste()`. Assim, tal código corresponde ao pedido de execução do método `teste()`, que deve ser procurado na classe do receptor original, **i1**, ou seja, na classe `UM`. Neste caso o método `teste()` é encontrado na classe `UM`, sendo de imediato executado e devolvendo o valor 1, que é o resultado da mensagem `result1()` enviada a **i1**. Claro que, apesar de termos definido o método `result1()` à custa de `this.teste()`, tal não significa que o método `teste()` deva obrigatoriamente ser codificado na classe `UM`. Se porventura este método não tivesse sido codificado em tal classe, após uma procura falhada na classe `UM`, o algoritmo de procura iria tentar encontrar tal método numa das superclasses de `UM`.

Bastante mais complexa vai ser a execução resultante de se enviar a mensagem `result1()` a **i4**, já que, como podemos verificar, a classe `QUATRO` não define esse método, tal como as classes `TRÊS` e `DOIS`. Vejamos o que se passa usando a Figura H.4.

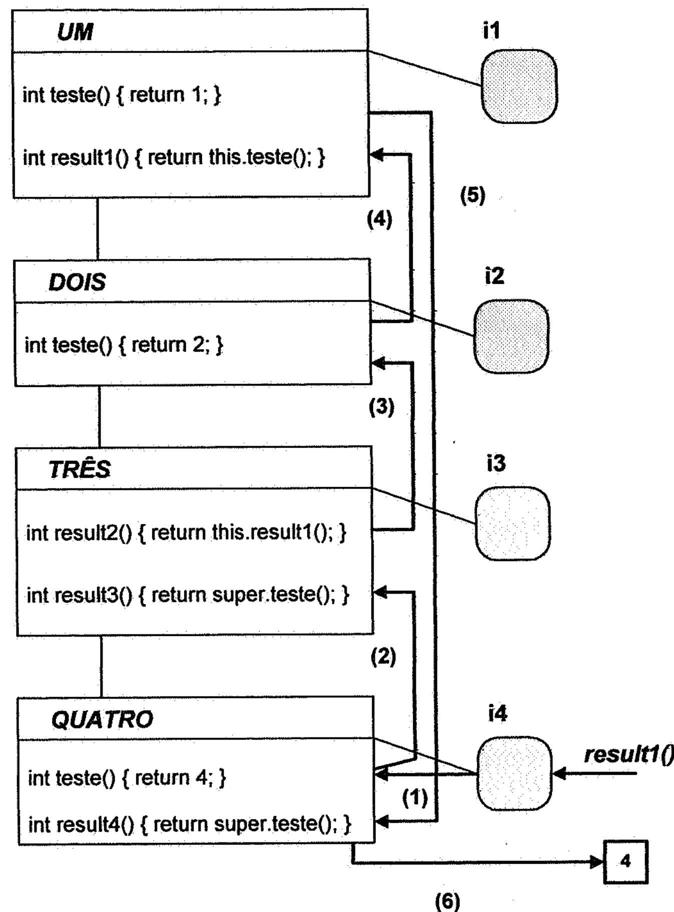


Fig. H.4. – Tratamento de `i4.result1()`

Como se pode verificar pela figura, os passos indicados por (1), (2) e (3) resultam em tentativas falhadas de encontrar o método respectivo na classe do receptor e nas duas superclasses imediatas. O método acaba por ser encontrado no passo (4) na classe `UM`, devendo ser executado o código `this.teste()`. Mas qual é agora o significado de `this` ? A resposta é que `this`, tal como no exemplo anterior, refere sempre a última instância que recebeu a mensagem, qualquer que esta seja, neste caso devendo ser enviada a **i4** a mensagem `teste()`. Ora no exemplo a mensagem `teste()` é encontrada na classe da instância receptora inicial, **i4**, cuja execução dá como resultado **4**.

SOBREPOSIÇÃO DE MÉTODOS (*"OVERRIDING"*). A REFERÊNCIA `super`

Sendo o mecanismo de herança automático, tal significa que uma dada classe herda obrigatoriamente das suas superclasses um conjunto de métodos de instância cujo código, como vimos anteriormente, é acessível às suas instâncias.

Que fazer, no entanto, quando uma classe herda um método cuja definição não lhe serve? A solução é realizar localmente a redefinição de tal método, desta forma sobrepondo à definição herdada uma definição local (*"overriding"*), sendo a definição local, como vimos atrás, a

primeira a ser encontrada e executada. Mas será que em definitivo é perdido o acesso ao método que foi redefinido?

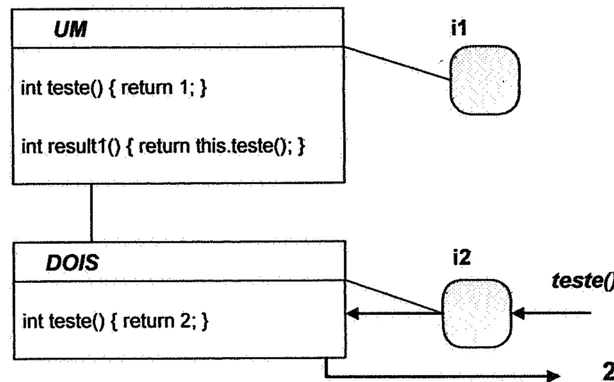


Fig. H.5 – Resultado de `i2.teste()`

Atentemos de novo na Figura H.3. O método `teste()`, inicialmente definido em `UM`, é redefinido em `DOIS`, é herdado sem redefinição por `TRÊS`, e de novo redefinido em `QUATRO`.

Considerando apenas as classes `UM` e `DOIS`, tal como na Figura H.5, o envio da mensagem `teste()` à instância **i2** produz o resultado determinado pela execução do método local à classe. Porém, como poderíamos na classe `DOIS` continuar a ter acesso ao método `teste()` da classe `UM`? A resposta é simples e passa pela utilização da referência `super`, indicando que o código do método deve ser directamente procurado na superclasse do receptor.

Assim, para que na classe `DOIS` pudéssemos executar directamente o código do método `teste()` da classe `UM`, sua directa superclasse, e não o código local de tal método, bastaria criar na classe `DOIS` um método, por exemplo, `result2()` cujo código fosse:

```
int result2() {
     return super.teste();
}
```

A mensagem `i2.result2()` resultaria de imediato na procura do método `teste()` na superclasse do receptor, método que, ao ser encontrado e executado, daria como resultado `1`.

Deste modo, quer o método local a uma classe que redefine um método herdado quer o método herdado, são ambos acessíveis às instâncias da classe que realizou o *overriding* do mesmo.

No entanto, a referência `super` não pode ser usada na mesma expressão mais do que uma vez, pelo que é incorrecta a expressão que pretenderia aceder ao método `teste()` duas classes acima da classe do receptor, como em:

```
super.super.teste();   // expressão incorrecta
```

Finalmente, convém não confundir **sobreposição** (*"overriding"*) de métodos com **sobrecarga** (*"overloading"*) de métodos. A **sobreposição** é uma redefinição de um método herdado, mantendo o seu nome, tipo e ordem dos parâmetros e resultado, mas modificando o seu

código. A **sobrecarga** consiste na criação ou definição de métodos *com o mesmo nome* dos já existentes na classe ou herdados, mas que têm parâmetros com tipos e/ou ordem diferentes e, eventualmente, diferentes tipos de resultado.

A herança pode implicar sobrecarga, já que métodos herdados podem possuir o mesmo nome de métodos locais. A sobrecarga pode mesmo existir dentro da classe, já que, como vimos, muitas vezes há vantagem em definir métodos com funcionalidades semelhantes usando o mesmo nome. Porém, a sobreposição nunca pode ser uma operação local, já que nenhuma classe pode ter dois métodos com o mesmo nome e a mesma assinatura. Assim, a sobreposição só pode acontecer em resultado de uma necessidade de se redefinir métodos herdados, não sendo nunca ambos os métodos, herdado e redefinidor, pertencentes à mesma classe.

Em JAVA, declarar que uma classe A é subclasse de uma classe **B** consiste em introduzir no cabeçalho da definição da subclasse:

```
public class A extends B {
```

Como cada classe possui uma e uma só superclasse (excepto a classe `Object`, por ser, em JAVA, a classe de topo da hierarquia) na cláusula `extends` apenas deve ser identificada a superclasse directa da classe. Por transitividade da herança, serão também automaticamente superclasses indirectas de A todas as superclasses de B.

No exemplo das classes Quadrado e Rectângulo teríamos, então, que declarar apenas:

```
public class Quadradro extends Rectang {
```

Quando a cláusula declarativa **`extends`** não é usada numa classe, tal significa que essa classe é uma subclasse imediata da classe de topo da hierarquia, que, como em seguida veremos, é a classe `Object`. Teremos nestes casos declarações tais como:

```
public class Point2D {
```

CLASSES SEM SUBCLASSES. MÉTODOS NÃO REDEFINÍVEIS

Em JAVA é possível definir classes de tal forma que, garantidamente, estas nunca poderão ter subclasses. Para que tal propriedade particular seja associada a uma dada classe, basta que no seu cabeçalho esta seja declarada como **`final`**. Uma classe declarada como `final` corresponde a dizer-se que tem uma definição completa, não necessitando de posteriores especializações. Qualquer tentativa de se criar uma subclasse de uma classe `final` gera um erro de compilação. Assim, os métodos de uma classe final são automaticamente não redefiníveis.

Igualmente, caso se pretenda que um dado método de instância de uma dada classe possa ser herdado por uma subclasse desta, mas não possa ser redefinido, então deveremos juntar o modificador **`final`** à declaração do método. O compilador de JAVA gera um erro de compilação relativamente a qualquer tentativa de redefinir um método declarado como `final`.

Finalmente, os métodos `static` (de classe), ainda que não sejam herdados, podem ser redefinidos. Porém, um erro de compilação é gerado se um método de instância redefine um método de classe ou vice-versa.

Para todos estes métodos que não são redefiníveis, o compilador de JAVA realiza optimizações de código, substituindo as suas invocações pelo seu código efectivo, operação que se designa por *"inlining"*, e que apenas se torna possível dada a não necessidade de activação do algoritmo de procura de métodos, dado que em tais casos não existe qualquer possível indecisão quanto ao código a executar.

Finalmente, e relativamente ao modificador de acesso do método redefinidor, métodos `public` apenas podem ser redefinidos por `public`, métodos `protected` por `public` ou `protected`, e métodos `package` por um qualquer desde que não `private`. A regra geral é, como se pode verificar, nunca diminuir ou restringir o grau de acessibilidade do método redefinido.

UTILIZAÇÃO DE `this()` E `super()` EM CONSTRUTORES

Podendo uma dada classe de JAVA possuir diversos construtores definidos pelo programador, para além do sempre existente construtor por omissão, será que é possível a um construtor de uma classe invocar um outro construtor da mesma classe? E faz tal invocação algum sentido?

A resposta a ambas as questões é sim. De facto, é possível em JAVA definir um dado construtor à custa de um outro construtor da mesma classe, usando no seu código a referência particular `this(.,.,.)` com os parâmetros adequados, em tipo e ordem, à invocação do outro construtor.

Tomando por exemplo a classe `Contador` do capítulo anterior, cujos construtores foram definidos como:

```
public Contador() {
      conta = 0;
}

public Contador(int val) {
      conta = val;
}
```

faria todo o sentido, à luz do que aqui se pretende introduzir, que o primeiro construtor pudesse ser definido à custa do segundo sob a forma:

```
public Contador() {
      this(0);            // invoca o construtor com parâmetro int
}
```

A utilização de `this(.,.,.)` tem uma restrição. Caso a construção seja usada num dado construtor, deverá ser obrigatoriamente a primeira instrução do código de tal construtor.

Uma outra construção muito importante relacionada com os construtores de JAVA é a construção `super(.,.,.)`. Quando criamos uma classe **B** que é subclasse de **A**, sabemos

que **B** herda as variáveis de instância de **A.** Assim, cada instância da classe B que seja criada possui uma estrutura interna que pode ser vista como sendo a soma das suas variáveis de instância definidas na classe com todas as outras variáveis de instância definidas nas suas superclasses.

Torna-se, assim, importante compreender de forma completa não apenas como se podem manipular as variáveis locais, como também como podem ser usadas todas as variáveis a que, por herança, as instâncias têm acesso.

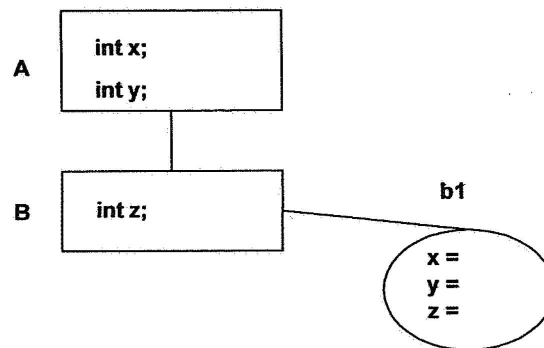


Fig. H.6 – Herança e estrutura das instâncias

A Fig. H.6 reforça a ideia de que uma qualquer instância da classe **B** possui uma estrutura interna que é um somatório puro, caso não haja conflitos nos nomes das variáveis, das variáveis de instância herdadas e das definidas na sua classe. Assim sendo, e havendo a garantia de que a classe **B** possui construtores, pelo menos o seu construtor por omissão, temos a certeza de que, ao ser criada uma instância da classe **B**, a variável de instância definida em **B** é inicializada.

Porém, coloca-se agora a questão de saber quem inicializa as variáveis que foram herdadas. A resposta é ao mesmo tempo simples e complexa. Tendo apenas em atenção o exemplo apresentado, torna-se óbvio que as variáveis herdadas de **A** apenas poderão ser inicializadas pelos construtores definidos em **A**, pelo que, assim sendo, qualquer construtor de instâncias da classe **B** deve obrigatoriamente invocar um qualquer dos construtores de instâncias de **A**, em última instância o construtor fornecido em JAVA por omissão de qualquer outro.

Sendo tal invocação obrigatória, sempre que esta regra não é respeitada no código dos construtores de uma dada classe, JAVA implicitamente insere como primeira instrução do construtor a instrução `super()`, que vai garantir a invocação do construtor por omissão da superclasse, dado que este existe sempre. Porém, caso o construtor da subclasse comece pela instrução `this()`, ou seja, invoque um outro construtor da mesma classe, JAVA vai verificar se este outro construtor invoca `super()` ou começa igualmente por `this()`. Dado que o próprio código de um construtor da subclasse pode invocar explicitamente um construtor da sua superclasse usando a construção `super(.,.,.)` com os devidos parâmetros, no final desta cadeia, quer implícita quer explicitamente, **algum dos construtores da superclasse será sempre invocado.**

Porém, caso `super(.,.,.)` seja de facto usado num construtor, então deverá ser a primeira instrução do código do construtor, mesmo quando `this(.,.,.)` for também usado.

A complexidade deste processo é que, sendo a herança transitiva, a criação de uma instância de uma dada classe obriga à invocação em cadeia dos construtores de todas as suas superclasses. Porém, como o algoritmo anterior é aplicado a todos os construtores de todas as classes, tal transitividade é assegurada. Isto é, quem tem que definir a classe **B** poderá ter algumas preocupações relativamente à invocação dos construtores mais adequados da superclasse **A**, mas tais preocupações não são extensíveis, já que quem definiu a classe **A** terá tido exactamente as mesmas para uma boa construção de instâncias de **A** usando os construtores da superclasse de **A**, e assim sucessivamente.

A cadeia de construtores é assim implícita e, na pior das hipóteses, à falta de uma mais eficiente inicialização, usa os construtores que por omissão são definidos em JAVA. Esta é também a razão porque JAVA os disponibiliza por omissão.

Note-se, finalmente, que os construtores não são herdados pelas subclasses, ainda que, como acabámos de verificar, lhes sejam acessíveis desta forma.

Exemplos concretos de codificação de construtores usando estas propriedades são apresentados mais adiante, restando apenas relembrar que os construtores, antes de terem como função a inicialização de variáveis, têm por função primária realizar a correcta alocação de espaço em memória para as instâncias que estão a criar e, adicionalmente, realizar a atribuição de valores iniciais.

HERANÇA E REDEFINIÇÃO DE VARIÁVEIS

Tudo o que vimos até aqui teve a ver com o conjunto de regras que necessitamos de conhecer para que de forma correcta possamos usar o mecanismo de herança quanto à herança comportamental ou de métodos.

Ainda que a herança de variáveis não seja muito relevante, dado que as regras da sua utilização nos obrigam a aceder às mesmas apenas através de métodos, por uma questão de completamento quanto a este assunto, vejamos as regras principais que se aplicam à herança de variáveis de instância, em particular tendo em atenção os modificadores `package`, `private`, `protected`, `public` e `final`.

Relativamente aos modificadores de acesso, à excepção de `private`, bastará dizer que os acessos que lhes são definidos são-no exactamente por herança, isto é, são acessíveis às subclasses, porque são por estas efectivamente herdadas. As variáveis e os métodos de instância declarados como `private` não são herdados. Os membros declarados como `final`, ainda que não redefiníveis, são herdados.

Quanto às `static`, ao contrário dos métodos, podem redefinir as não `static` que foram herdadas e vice-versa, pois têm formas de acesso e utilizações distintas.

O TOPO DA HIERARQUIA: A CLASSE OBJECT

Devemos agora pensar qual a classe que deverá ser posicionada no topo desta hierarquia de classe e porquê. A classe que for posicionada no topo da hierarquia de classes vai ver a sua estrutura e o seu comportamento herdado por todas as outras classes, dado que todas elas são suas subclasses. Assim, a responsabilidade de determinar qual a classe de topo é grande.

Por outro lado, sabemos que todas as instâncias de qualquer das classe existentes, e até as próprias classes, têm uma característica em comum: **são objectos**.

Assim, parece ser conveniente que, dado todas serem objectos, mas não tendo qualquer sentido procurar estrutura comum a todas elas, o seu comportamento comum enquanto objectos seja por todas herdado. Deste modo, e dado que tudo são objectos, a classe de topo da hierarquia será de facto a classe `Object`, que sendo bastante genérica, define o que é comportamentalmente comum a todas as suas subclasses. Assim, todas herdam de **`Object`**, pelo que assim todas as suas instâncias se comportam como **objectos**.

A classe `Object` de JAVA define um conjunto interessante de métodos que todas as subclasses vão herdar, dos quais se salientam os seguintes:

```
Class getClass()              // devolve a classe do objecto;
boolean equals(Object obj)    // igualdade de referências;
Object clone()                // clonagem;
String toString()             // representação como string;
```

Estes métodos são tão genéricos que em geral **devem ser redefinidos** pelas várias classes que são desenvolvidas. Caso não sejam redefinidos, e por herança, as instâncias das subclasses de `Object` responderão às mensagens respectivas executando o código destes métodos existente em `Object`.

CRIAÇÃO DE CLASSES VIA HERANÇA: EXEMPLOS

1º EXEMPLO: PARES E TRIPLOS

Vamos começar por um exemplo muito simples e genérico. Imaginemos que se pretende criar uma classe que defina ***pares de objectos***, pretendendo-se saber a cada momento o número total de pares criados, ter acesso ao primeiro e segundo componentes de um par, verificar a igualdade de pares e visualizar pares.

Não podendo herdar de nenhuma classe especial, a classe **`ParObj`** será uma subclasse directa de `Object` e terá a seguinte definição:

```
public class ParObj {

  // variáveis de classe

     static int tpares = 0;// total de pares criado

  // métodos de classe

     public static void incPares(); {   // aumenta total de pares
       tpares++;
     }

     public static int getTotPares(); {   // consulta total
       return tpares;
     }

     // construtores

     public ParObj(Object obj1, Object obj2) {
             campo1 = obj1; campo2 = obj2;
             this.incPares();
     }

     // variáveis de instância

     private Object campo1, campo2;

     // métodos de instância

     public Object daCampo1() {
           return campo1;
     }

     public Object daCampo2() {
           return campo2;
     }

     public boolean equals(ParObj par) {
           return campo1.equals(par.daCampo1()) &&
                  campo2.equals(par.daCampo2()) ;
     }

     public String toString() {
           return new String("Par(" + campo1.toString() + ", "
                                    + campo2.toString()) + ")";
           }

  }
```

A classe é genérica no sentido em que os campos que vão constituir o par foram definidos como sendo do tipo `Object`, o tipo mais genérico possível. As instâncias de `ParObj` podem ser heterogéneas, isto é, podem não ser constituídas por dois componentes necessariamente do mesmo tipo. Por exemplo, podemos ter como primeiro componente uma instância de `String` e como segundo uma instância de `Triângulo`. Usando *casting* poderemos sempre converter resultados `Object` em objectos da classe pretendida.

Definida esta classe `ParObj`, pretende-se agora criar uma outra classe que permita guardar **triplos de objectos**, ou seja, como se fossem registos ou estruturas com três campos também genéricos, e que conte igualmente os triplos criados, permita aceder aos valores de cada um dos campos, visualize os triplos e teste igualdade de triplos.

Sendo certo que poderíamos por composição criar esta nova classe `TriplObj`, conforme a definição parcial da classe:

```
public class TriplObj {

        ...........................

        // variáveis de instância

        private ParObj primeiros2;    // usa composição
        private Object campo3;
```

que usa um `ParObj` para os primeiros dois campos e um `Object` para o terceiro, a verdade é que, sendo esta classe subclasse de `Object`, não reflete segundo esta definição o facto evidente de que se trata de uma extensão, ou especialização, em estrutura e comportamento à classe `ParObj` já existente.

A forma de tornar clara tal relação de especialização, tirando de imediato partido do mecanismo de herança subjacente, é de facto declarar `TriplObj` como sendo subclasse de `ParObj`.

Vejamos então agora, como se torna simples, usando os nossos conhecimentos sobre a forma de funcionamento do mecanismo de herança e procura de métodos, definir a classe `TriplObj` à custa do que é herdado de `ParObj`.

```
public class TriplObj extends ParObj {

  // variáveis de classe

   static int ttriplos = 0;

 // métodos de classe

   public static void incTriplos(); {   // aumenta total
      ttriplos++;
   }

   public static int getTotTriplos(); {   // consulta total
     return ttriplos;
   }

   // construtores

   public TriplObj(Object obj1, Object obj2,
                   Object obj3) {
        super(Object obj1, Object obj2); // de ParObj
        comp3 = obj3;
        this.incTriplos();
   }

   // variáveis de instância

        private Object campo3; // os outros são herdados !!
```

```
// métodos de instância

public Object daCampo3() {
    return campo3;
}

private ParObj converte() {  // converte Triplo em Par
      return new ParObj(this.daCampo1(),
                        this.daCampo2());
}

public boolean equals(TriplObj triplo) {
   return this.converte().equals(triplo.converte()) //Ponto2D
       && campo3.equals(triplo.daCampo3()) ;
}

public String toString() {
   return new String("Triplo("
                  + this.daCampo1().toString()
                  +    ", " + this.daCampo2().toString())
                  +    ", " + this.campo3 + ")");
}
}
```

As notas mais importantes deste exemplo estão sublinhadas a negrito. Em primeiro lugar, a declaração de que `TriplObj` **extends** `ParObj` colocada no cabeçalho. Em seguida, o facto novo de que o construtor `TriplObj` pode invocar, usando a expressão **`super()`** com parâmetros adequados, um qualquer construtor definido na superclasse. Neste caso foi invocado o único existente. Caso existissem vários, a assinatura da invocação determinaria qual o que deveria ser executado. Depois, o facto de que apenas é necessário declarar a variável de instância **campo3** na nossa classe `TriplObj`, já que as outras duas necessárias são herdadas da superclasse. O mesmo se pode dizer relativamente aos métodos de acesso, sendo herdados exactamente aqueles que acedem às duas componentes herdadas. Assim, apenas mais um, específico da extensão, teve que ser criado `daCampo3()`. Finalmente, a criação de um método auxiliar, como tal `private`, que converte triplos em pares facilitando a reutilização dos métodos herdados que trabalham com pares. Deste modo, a igualdade de triplos acabou por ser definida à custa da igualdade de pares que havia sido herdada, mais a comparação do terceiro componente. Seria também perfeitamente válida a definição:

```
public boolean equals(TriplObj triplo) {
   return this.daCampo1().equals(triplo.daCampo1())
      &&  this.daCampo2().equals(triplo.daCampo2())
      &&  this.campo3.equals(triplo.daCampo3()) ;
}
```

2º Exemplo: Pontos2D e Pontos3D

Dada uma classe `Ponto2D` que implementa pontos no plano cartesiano, pretende- -se definir uma classe que implemente pontos tridimensionais. A classe `Ponto2D` é definida como:

```java
public class Ponto2D {

// construtores

        public Ponto2D(int x, int y) {
          this.x = x; this.y = y;
        }

        public Ponto2D() {
          this(0, 0);              // invoca o construtor anterior
        }                          // redefine construtor por omissão

        // variaveis de instancia

        private int x ;
        private int y ;

        // metodos de instancia

        public int cx() { return x; }

        public int cy() { return y; }

        public void incx() { x++ ; }

        public void incx(int dx) { x = x + dx ; }

        public void incy() { y++ ; }

        public void incy(int dy) { y = y + dy ; }

        public String toString() {
          return new String("Ponto2D(" + x + "," + y +")") ;
        }

        public void somaponto(Ponto2D p) {
           this.x = this.x + p.cx() ;
           this.y = this.y + p.cy() ;
        }

        public void somaponto(int x, int y) {
           this.x = this.x + x;
           this.y = this.y + y;
        }
}
```

A classe `Ponto3D` vai ser definida como sendo subclasse de `Ponto2D`, declarando apenas a variável de instância correspondente à coordenada em **z** e adequando o código dos métodos à sua representação herdada e local.

Nota: *Faça o exercício de definir Ponto3D usando apenas a API de Ponto2D, ou seja, como se nem conhecesse a estrutura interna de Ponto2D !!*

```
public class Ponto3D extends Ponto2D {

// construtores

        public Ponto3D(int x, int y, int z) {
          super(x, y) ;
          this.z = z ;
        }

        public Ponto3D() {
          super();
          z = 0;
        }

// variáveis de instância
        private int z ;

// métodos de instância

        public int cz() {
          return z;
        }

        public void somaponto(int x, int y, int z) {
              super.somaponto(x, y);
              this.z += z;
        }

        public String toString() {
           return new String("Ponto3D(" + this.cx() + ","
                   + this.cy() + "," + z + ")");
        }

        public Ponto3D pontosoma(Ponto3D p) {
          return new Ponto3D(this.cx() + p.cx(),
                            this.cy() + p.cy(),
                             z + p.cz());
        }
}
```

3º EXEMPLO: STACK COMO SUBCLASSE DE VECTOR

No capítulo anterior apresentámos a classe `Vector`, predefinida em JAVA, como sendo uma classe cujas instâncias são estruturas lineares indexadas e infinitas. Os métodos principais de `Vector` foram igualmente apresentados.
Vamos realizar o exercício de criar uma classe que implementa *stacks ilimitadas* de objectos. Sendo uma *stack* uma estrutura linear, ainda que não indexada, e que possui a particularidade de funcionar segundo um critério de *first in first out* (FIFO) relativamente à forma de inserção e remoção dos seus elementos, ou seja, funcionando sobre um único extremo de uma estrutura linear, então tanto podemos implementar *stacks* usando um array como variável de instância, isto é, por composição, como podemos definir uma classe `Stack` como sendo subclasse de `Vector`, herdando a estrutura interna de representação de vectores, bem como os respectivos métodos da mesma. Seja `Stack` subclasse de `Vector`.

```
import java.util.*;
public class Stack extends Vector {

// construtores

      public Stack() {        // usa construtor de Vector
            super();
      }

// métodos de instância

      public boolean empty() {          // pre-cond de pop()
            return super.isEmpty();          // e de top()
      }

      public void push(Object obj) {
            super.addElement(obj);
      }

      public Object top() { // tem pre-condição !!
            int index = super.size()-1;
            return super.elementAt(index);
      }

      public void pop() { // tem pre-condição !!
        return super.removeElementAt(super.size()-1);
      }
}
```

Note-se que criámos uma implementação de *stacks* genéricas **sem criar qualquer estrutura para a representação** de uma *stack* e até sem sabermos qual é, de facto, a estrutura usada em `Vector` – *abstracção completa* -, e sem criar qualquer código especial, mas apenas invocando os métodos disponíveis em `Vector`. Este exemplo já reflecte, de forma razoavelmente clara, a poupança de esforço que a herança pode proporcionar ao programador quando este a usa correctamente.

Note-se ainda, conforme se mostra no código assinalado a negrito, que sempre que foi preciso invocar um método da superclasse de `Stack`, optámos por usar a palavra reservada `super`, desta forma optimizando-se a procura dos métodos que serão de imediato pesquisados na classe `Vector`. Este tipo de programação não é no entanto, em geral, aconselhável já que, embora seja mais eficiente, impossibilita posteriores redefinições de tais métodos na subclasse. Por exemplo, se mais tarde pretendêssemos criar na classe `Stack` um método `size()`, pelo menos no método `pop()` este nunca seria invocado dada a utilização de `super`. No entanto, se tivéssemos escrito em tal método `this.size()` o código seria mais genérico e extensível, dado que, se na classe `Stack` existisse o método `size()` seria esse o invocado, caso contrário, por herança, seria procurado e invocado o da classe superclasse `Vector`. Esta solução é sempre, portanto, muito mais genérica.

Quanto à utilização do *construtor* `super()`, ela é óbvia, dado que mesmo que pretendêssemos usar um outro, não o saberíamos criar, pois não conhecemos qual a estrutura interna de `Vector`. Assim, usamos apenas o construtor usual da superclasse.

```
/**
 *
 * ESTA CLASSE E A CLASSE PONTO3DSM NAO POSSUEM, PROPOSITADAMENTE
 * MODIFICADORES DE ACESSO, PARA QUE SE POSSA ANALISAR TODAS AS
 * POSSIBILIDADES DE REDEFINIÇÃO DE VARIÁVEIS E MÉTODOS QUANDO
 * HÁ HERANÇA.
 *
 * AS CLASSES PONTO2DP e PONTO3DP SÃO AS IMPLEMENTAÇÕES CORRECTAS
 *
 * */

public class Ponto2D {

// constante de classe

static double PI = 3.14159365358;

// variáveis de classe

static int numPontos = 0;

// métodos de classe

static void maisUm() {
  numPontos = numPontos + 1 ;
}

static double daPi() {
  return PI;
}

static int daNumPontos() {
  return numPontos ;
}

// construtores

Ponto2D(int x, int y) {
  this.x = x; this.y = y;
  this.maisUm();
}

Ponto2D() {
  this(0, 0) ;
}

// variáveis de instância

int x ;
int y ;
```

```
// métodos de instância

int cx() {
  return x;
}

int cy() {
  return y;
}

void incx() {
  x = x + 1 ;
}

void incx(int dx) {   // polimorfismo
  x = x + dx ;
}

double meuPi() { // método de instância que acede a PI.
   return PI;
}

double vePi() { // usa método de classe
  return Ponto2D.daPi();
}

public String toString() {
     return (new String("Ponto2D(" + x + "," + y +")"));
}

void somaponto(Ponto2D p) {
   x = x + p.cx() ;
   y = y + p.cy() ;
}

void somaponto(int x, int y) {
   this.x = this.x + x;
   this.y = this.y + y;
}

Ponto2D pontosoma(Ponto2D p) {
  return new Ponto2D(x + p.cx(), y + p.cy());
}

}
```

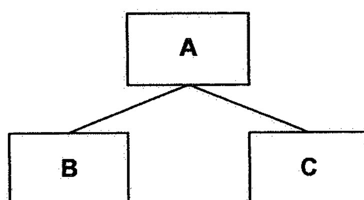
```
public class Ponto3D extends Ponto2D {

  // constante de classe - redefinição de PI herdado

  static double PI = 3.1416;

  // variáveis de instância

  int z ;  // x e y são herdadas !!

  // construtores

  Ponto3D(int x, int y, int z) {
    super(x, y) ;  // invoca construtor da superclasse
    this.z = z ;
  }

  Ponto3D() {
    super(); // invoca construtor da superclasse
    this.z = 0;
  }

  // métodos de instância

  int cz() {
    return z;
  }

  double meuPi() { // exemplo de redefinicao e visibilidade
    return PI;
  }

  double superPi() {
    return super.PI; // acesso à constante da superclasse
  }

  int cx() {  // exemplo simples de redefinição
     return x*10;
  }

  void somaponto(int x, int y, int z) {
     this.somaponto(x, y);
     this.z = this.z + z;
  }

  public String toString() {
     return new String("Ponto3D(" + this.cx() + ","
                                  + this.cy() + ","
                                  + z + ")");
  }
```

EXECUÇÃO DO TESTE - RESULTADOS

```
c:\jdk1.1>java Teste1

----Resultados-----
PI = 3.14159365358
Ponto 2D N° 1 =0, 0
Ponto 3D N° 1 =0, 0, 0
--------  PIs ----------------
Pis em Ponto2D =
Pi Local via meuPi() =3.14159365358
Pi Local via daPi() =3.14159365358
Pis em Ponto3D =
Pi Local via meuPi() =3.1416
Pi da Superclasse =3.14159365358
Pi da Superclasse =3.14159365358
Ponto 2D N° 1 =2, 0
Ponto 2D N° 2 (apos somaponto) =15, 35
Ponto 2D N° 1 (apos somaponto) =12, 5
Ponto Soma 2D 20, 50
X de paux2 22
X de paux1 5
Ponto 3D aux1 =45, 30, 45
Ponto 3D N° 2 (apos somaponto) =35, 25, 35
-----------------------

c:\jdk1.1>java Teste1
```

COMPATIBILIDADE ENTRE CLASSES E SUBCLASSES. O PRINCÍPIO DA SUBSTITUTIVIDADE.

Para além das vantagens claras que em PPO advêm de possuirmos um mecanismo de classificação hierárquico apoiado num mecanismo de herança, sempre que os mesmos são bem utilizados, vamos agora analisar o grau de compatibilidade entre instâncias das subclasses e a sua respectiva superclasse.
Consideremos a seguinte hierarquia exemplo:



Compatibilidade entre classes e subclasses: Exemplo

```
public class A {
      public A() { a = 1; }
      public A(int val) { a = val; }
      int a;
      public int daA() { return a; }
      public void metd() { a += 10 ; return a; }
}

public class B extends A {
      public B() { b = 2; }
      public B(int val) { b = val; }
      int b;
      public int daB() { return b; }
      public void metd() { b += 20 ; return b; }
}

public class C extends A {
      public C() { c = 3; }
      public C(int val) { c = val; }
      int c;
      public int daC() { return c; }
      public void metd() { c += 30 ; return c; }
}
```

O método **metd()** é sucessivamente redefinido. Consideremos a seguinte classe de teste e analisemos os resultados da execução do código respectivo.

```
public class TesteABC {
//
   public static void main(String args[]) {
       A a1, a2, a3;
       a1 = new A(); a2 = new B(); a3 = new C();
       a1.metd(); a2.metd(); a3.metd();
       System.out.println("a1.metd() = " + a1.daA());
       System.out.println("a2.metd() = " + a2.daB());
       System.out.println("a3.metd() = " + a3.daC());
}}
```

Em primeiro lugar, repare-se que declarámos 3 variáveis de tipo **A** e que a estas pudemos associar instâncias quer de **A**, quer de **B**, quer de **C**. O compilador aceita tais atribuições, dado estarem em conformidade com o denominado **princípio da substitutividade**.

Uma das mais interessantes características das linguagens de PPO é que o *valor* (instância) referenciado por uma variável pode não ser do *tipo* (classe) declarado para essa variável. Mas evidentemente também, não poderá ser um qualquer, sendo claro o que o **princípio da substitutividade** determina: *declarada uma variável como sendo de uma dada classe, é legal que lhe seja atribuído um valor da sua classe ou de qualquer subclasse desta.*

TIPOS ESTÁTICOS E DINÂMICOS.

Sendo a declaração de uma variável um procedimento estático e que tem a ver com a fase de compilação, enquanto que uma atribuição é um procedimento dinâmico, pois é realizado durante a execução do programa, torna-se importante então que se passe a fazer a distinção entre o **tipo estático** e o **tipo dinâmico** de uma variável, já que, como acabámos de ver pelo princípio enunciado, eles podem não coincidir.

Assim, o tipo estático será sempre o tipo da sua declaração tal como o compilador o aceitou. No exemplo em análise, em que se escreveu a declaração

```
A a1, a2, a3;
```

o **tipo estático** das variáveis é a classe **A**. O princípio da substitutividade garante, no entanto, a legalidade de se associarem valores (instâncias) a estas variáveis, que podem ser quer instâncias de **A** quer de uma qualquer subclasse de **A**. Assim, estão sintacticamente correctas as expressões no exemplo sublinhadas a negrito:

```
a1 = new A(); a2 = new B(); a3 = new C()
```

O facto de uma variável poder possuir um valor de tipo dinâmico diferente do seu tipo estático, ainda que dentro da regra de subclassificação enunciada, é uma fonte de poder expressivo, de generalidade e de flexibilidade que apenas as linguagens de PPO suportam de forma tão natural e que iremos procurar analisar e explorar mais adiante. Uma variável cujo tipo dinâmico pode ser diferente do tipo estático diz-se, naturalmente, que é **polimórfica**.

Informalmente, e de um ponto de vista meramente sintáctico, o princípio está de acordo com o facto de que subclasses são em PPO extensões, especializações, da superclasse, pelo que, herdando desta um conjunto de métodos e podendo a estes acrescentar os seus próprios, superclasses e subclasses são comportamentalmente compatíveis no comportamento herdado, isto é, o que foi definido na superclasse. Desde que o método correspondente à mensagem enviada a uma variável de dado tipo esteja definido na classe desta, o compilador tem a garantia de que, mesmo que mais tarde tal variável possua um valor não da sua classe mas de uma das suas subclasses, seja por herança ou porque a subclasse o redefiniu, tal método será sempre executável, ainda que seja necessário determinar em tempo de execução qual o método que efectivamente deverá ser executado.

Vamos, em seguida, analisar o que se passa em tempo de execução. A classe de teste que foi apresentada, vai apenas apresentar o resultado da execução das expressões:

```
a1.metd();
a2.metd();
a3.metd();
```

ou seja, de enviarmos às variáveis `a1, a2` e `a3`, da classe **A**, a mensagem `metd()`, cujo método correspondente está definido na classe **A**.

Se o algoritmo de procura e execução de métodos fosse baseado apenas nos *tipos estáticos* das variáveis, então, sendo as variáveis receptoras da mensagem da classe **A**, os resultados esperados seriam respectivamente, **`11, 21, 31`**, caso o método executado fosse 3 vezes o método `metd()` da classe **A**.

No entanto, o algoritmo de procura e execução está programado para explorar o princípio da substitutividade, pelo que se baseia não no tipo estático, que nunca mais irá mudar, mas sim no tipo dinâmico das variáveis, que pode variar ao longo da execução do programa (cf. por exemplo `a3 = new B()` mais tarde).

PROCURA DINÂMICA DE MÉTODOS *("DYNAMIC METHOD LOOKUP")*

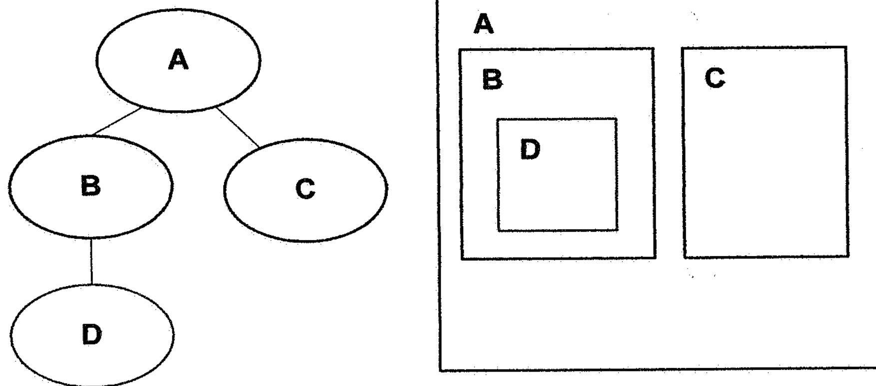
Ainda que ao compilador seja impossível saber qual o tipo dinâmico de uma certa variável, o código produzido por este vai permitir que o interpretador de JAVA use um algoritmo de **procura dinâmica de métodos**, em função da análise que faz em tempo de execução de uma expressão, tal como `a2.metd()`.

De facto, em vez de usar de imediato o método `metd()` associado ao tipo estático da variável a2 (classe **A**), o interpretador vai em primeiro lugar determinar o tipo dinâmico de a2 em tal expressão, a classe B no exemplo, e em tal classe procurar o código do método a executar. Caso exista, como no exemplo dado, o seu código é executado, neste caso, dando como resultado o valor **`21`**. Caso não exista em tal classe, a procura continua seguindo o algoritmo de procura nas superclasses desta.

Assim, no exemplo em análise e por todas as razões apresentadas, o resultado da execução da classe de teste seria de facto **`11, 22`** e **`33`**, o que demonstra que os métodos que foram realmente executados foram os que correspondem aos tipos dinâmicos dos receptores das mensagens e não ao seu tipo estático.

O princípio da substitutividade, o polimorfismo das variáveis e o algoritmo de procura dinâmica de métodos, associados à **herança**, conferem à PPO um poder expressivo, de generalização e de extensibilidade do código ímpar.

TEORIA DOS CONTENTORES
TIPOS ESTÁTICOS vs. TIPOS DINÂMICOS




```
A var_a1 = new A();
A var_a2 = new B();
A var_a3 = new C();
A var_a4 = new D();

B var_b1 = new B();
B var_b2 = new D();
```

A uma variável declarada como sendo de uma dada classe podem ser atribuídas instâncias dessa classe ou de qualquer das suas subclasses. O compilador aceita a regra e não dá qualquer erro de compilação. Depois compete ao interpretador, em tempo de execução, determinar de que classe é o objecto que a cada momento está contido em tal variável (usando `getClass()`). Determinada a classe do objecto contido na variável, o método correspondente a qualquer mensagem enviada será procurado na classe do objecto contido e não na classe da variável que o contém, ou seja, dinamicamente.

Se a mensagem enviada corresponder a um método que todas as subclasses implementam, então teremos flexibilidade máxima pois em tempo de execução será executado o código apropriado cf. a classe do objecto contido.

NOTA: Os maiores "contentores" que podemos usar (ou seja os mais genéricos) são os contentores declarados como sendo da classe `Object`, pois todas as classes de Java são subclasse de `Object`. Por isso, todo o código de todas as classes que são implementações de estruturas de dados (cf. Vector, ArrayList, HashMap, etc.) têm como parâmetros e como resultados `Object` !! Qualquer instância de qualquer classe pode ser guardada numa variável de tipo `Object`.

PROGRAMAÇÃO GENÉRICA VIA CLASSE OBJECT

Há muito que vinhamos afirmando que se pretendermos desenvolver classes com alguma generalidade deveríamos usar como classe de representação a classe que se encontra no topo da hierarquia de classes de JAVA, a classe `Object`.

Torna-se agora compreensível que, sendo a classe `Object` o topo da hierarquia de classes de JAVA, logo a superclasse de todas as classes, a uma variável de tipo estático `Object` é possível associar um valor que é uma instância de uma qualquer classe de JAVA. Assim, ao definirmos uma classe **`Stack`** de **`Object`**, estamos de facto a definir *stacks* não só genéricas, em que `Object` pode dinamicamente representar instâncias de qualquer subclasse, mas também heterogéneas, dado que os elementos da *stack* não necessitam de ser todos do mesmo tipo dinâmico.
No entanto, e dado que a classe `Object` possui uma API muito limitada, haverá que ter em atenção que, do ponto de vista estático, não serão muitas as mensagens que poderão ser enviadas a variáveis do tipo estático `Object`. Assim, ainda que em termos de tipos estáticos faça sentido usar `Object` nos parâmetros e resultados de métodos, cf. se viu na definição da classe `Stack`, em termos dinâmicos fazer *casting* para o tipo específico que se pretende garante a utilização correcta e total das instâncias de tal tipo, aumentando por exemplo o número total de mensagens que lhes podem ser enviadas, já que passando de `Object` para o seu tipo efectivo, a instância poderá responder às mensagens que correspondem a métodos herdados, bem como às que se referem aos seus métodos locais.

PROGRAMAÇÃO GENÉRICA VIA POLIMORFISMO

Continuando a utilizar as classes **A**, **B** e **C** anteriormente definidas, vamos agora apresentar mais um exemplo de como a programação se torna genérica e simples se usarmos os princípios e mecanismos estudados anteriormente.

Admitamos que se pretende criar um *array* que seja capaz de conter instâncias das classes **A**, **B** e **C**. Pensando de imediato em qual será a mais correcta declaração a fazer para tal *array*, a procura de generalidade determina que, estaticamente, este seja declarado como sendo do tipo mais genérico dos tipos em questão, ou seja, **A**.

Teremos assim a declaração **`A[] a;`** *array* no qual, em seguida, se poderiam introduzir diferentes instâncias das classes em questão, por uma ordem qualquer, tal como em:

```
a[x] = new A();
a[y] = new C();
a[z] = new B();
```

Ao pretendermos apresentar através de uma classe de teste os resultados do envio da mensagem `metd()` às diversas instâncias guardadas nas várias posições do *array*, bastaria escrever o código seguinte:

```
for(int i=0; i < a.length ; i++) {
     System.out.println(i + " -> " + a[i].metd());
}
```

O código de `metd()` a executar para cada `a[i]` será correctamente seleccionado pelo algoritmo de *dynamic method lookup*, conforme o tipo dinâmico da instância armazenada em tal índice do *array*, pelo que a escrita do código se torna não só simples, como genérica.

A não completa compreensão destes princípios e mecanismos, associada por vezes a muitos anos de programação imperativa, conduz por vezes à escrita de código não só redundante como não extensível, como o que se apresenta a seguir:

```
for(int i=0; i < a.length; i++;) {
      if (a[i] istanceof A)
        System.out.println(i + "->" + (A a[i]).metd());
      else
        if (a[i] istanceof B)
           System.out.println(i + "->" + (B a[i]).metd());
        else
             if (a[i] istanceof C)
              System.out.println(i + "->" + (C a[i]).metd());
}
```

O programador deste código, dado desconhecer os conceitos e mecanismos da PPO que foram anteriormente introduzidos, julga ser necessário determinar no seu código qual o tipo dinâmico de cada valor associado a dada posição do *array* e, em seguida, em função de tal tipo, realizar o *casting* correcto para só então enviar com toda a segurança a mensagem a tal objecto.

Desconhecendo que o algoritmo de procura dinâmica de métodos faz exactamente este trabalho em tempo de execução, este programador está a programar parte de tal algoritmo, o que é, portanto, um esforço redundante. Para além disso, o que é muito mais grave, provavelmente sem o saber, este seu código não é nem genérico nem extensível. Senão vejamos.

Vamos admitir que posteriormente é criada uma classe **D**, igualmente subclasse de **A**, que apresenta a mesma API, ou seja, responde às mesmas mensagens, mas para valores de **x** diferentes. É óbvio que caso instâncias desta nova classe passem a fazer parte do *array*, o ciclo `for` escrito no primeiro exemplo não necessita de qualquer modificação. Caso surja uma instância de **D**, ela será tratada de forma igual a todas as outras, sendo invocado o método `metd()` respectivo. Ou seja, esse código mantém-se inalterado, apesar de termos realizado uma extensão à hierarquia de classes. Esta é a propriedade da **extensibilidade**, ou seja, garantir continuidade do código, e não a sua caducidade, apesar das várias modificações que, posteriormente, podem ser introduzidas nos dados.

Porém, o código do programador que desconhece estes princípios não resiste a tais modificações, sendo necessário alterá-lo de cada vez que uma nova subclasse de **A** é criada. No exemplo, haveria agora a necessidade de alterar o código para testar se se trata de uma instância de **D** e realizar o respectivo tratamento, e o mesmo para cada nova extensão. Tal não faz sentido em PPO, dado que o que se pretende na utilização do paradigma é exactamente a flexibilidade oposta que, como se viu, é atingível sem esforço, mas com conhecimento do paradigma.

PROGRAMAÇÃO INCREMENTAL

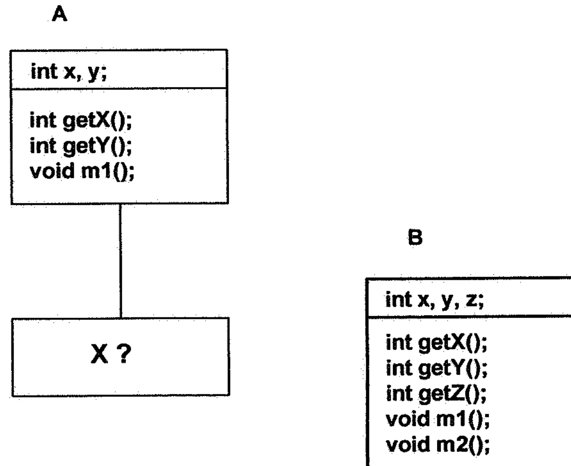
O mecanismo de herança introduz uma metodologia de programação que se pode designar por **programação incremental**. De facto, como pudemos anteriormente verificar através dos exemplos apresentados, a efectiva construção de uma nova classe consiste na resolução de uma simples equação, que se traduz literalmente na expressão:

HERDADO + PROGRAMADO = PRETENDIDO

De uma outra forma, se pretendemos criar uma nova classe **B** possuindo uma dada estrutura e um dado comportamento, e se já temos implementada uma classe **A** com dada estrutura e comportamento, então se a nova classe for subclasse desta, teremos apenas que programar a diferença entre o que é herdado de **A** e o que se pretende ter em **B**. Ou seja:

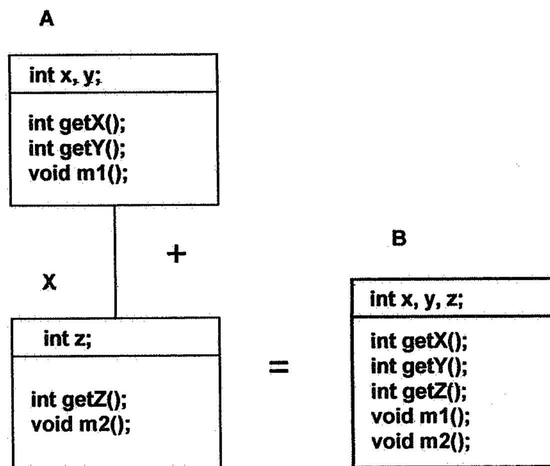
A PROGRAMAR = PRETENDIDO - HERDADO

Por exemplo se, tal como se apresenta na figura seguinte, tivermos uma classe **A** com as variáveis e métodos de instância indicados e pretendemos criar uma classe **B** que deve ter a estrutura e o comportamento apresentado, então a questão que se deve resolver é determinar qual a classe **X** a programar, tal que, sendo **X** subclasse de **A**, nos permita ver a estrutura e o comportamento total que se pretende para **B**.



Equação a resolver na programação incremental

A solução é, naturalmente, uma classe **X** definida como:



Classe X como solução

Torna-se agora compreensível que se afirme que, para a construção de uma classe nova, quanto mais pudermos herdar de classes existentes menos esforço teremos na programação da nova classe. Porém, tal como já foi afirmado antes, para que tal esforço possa ser de facto reduzido, é absolutamente necessário um conhecimento profundo das classes já existentes em JAVA. Só assim poderemos fazer um uso efectivo dos mecanismos de reutilização à nossa disposição em PPO.

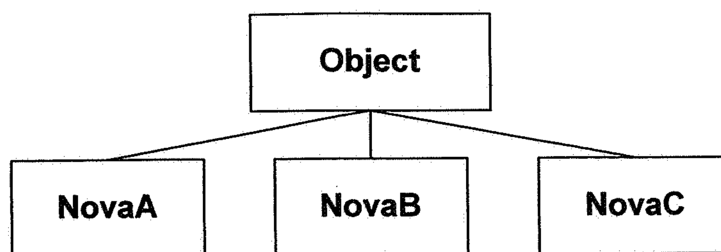
O PROBLEMA DA CLASSIFICAÇÃO

CLASSES NOVAS COMO SUBCLASSES DE OBJECT. ERRO DE CLASSIFICAÇÃO

A correcta classificação de uma nova classe é uma tarefa relativamente complexa. Aliás como sabemos do dia a dia, qualquer processo classificativo é complexo pois implica um conhecimento perfeito da entidade a classificar bem como das regras a empregar para realizar a sua correcta classificação.

Em PPO, o relativo desconhecimento das classes existentes e do que delas pode ser herdado e que em muito pode auxiliar à definição de novas classes, faz com que muitas vezes, por displicência, as novas classes sejam classificadas como subclasses directas da classe `Object`.

Reduzindo tal prática a uma situação absurda em que na hierarquia de classes todas as classes existentes, excepto `Object`, fossem subclasses de `Object`, a herança não seria necessária, dado que, praticamente, apenas existiria um nível de classes. A reutilização seria mínima e cada classe nova deveria ser programada de raiz.

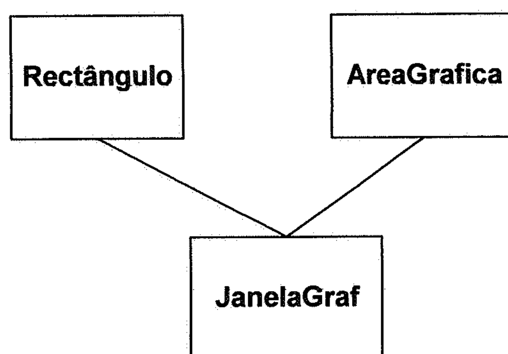


Subclassificação pobre: subclasses de Object

A um maior esforço de classificação das classes corresponderá sempre uma maior reutilização de código, por herança, e, em consequência, um menor esforço de programação.

HERANÇA SIMPLES E MÚLTIPLA

Em JAVA, qualquer classe, excepto a classe `Object`, tem sempre, no máximo, uma superclasse. Este tipo de herança designa-se por tal razão **herança simples**. No entanto, há linguagens de PPO como C++, em que uma classe pode ser subclasse de mais do que uma classe, ou seja, herdar em simultâneo de mais do que uma classe, sendo por tal motivo a herança designada por **herança múltipla**.



Exemplo de herança múltipla

Teoricamente, há muito que foi provada a equivalência expressiva das duas formas de herança, possuindo ambas vantagens e desvantagens entre si.

HERANÇA VERSUS COMPOSIÇÃO

Muitas das dificuldades que surjem no desenvolvimento de novas classes, resultam de alguma tendência para confundir herança com composição. Estas duas formas de relacionamento entre classes, extensível às suas instâncias, são, no entanto, muito distintas.

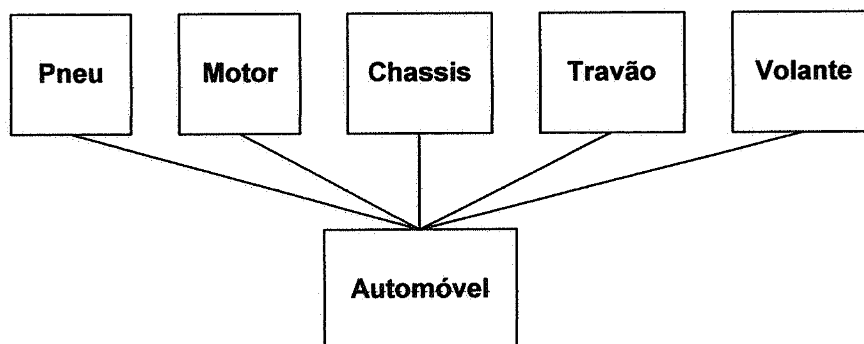
Quando uma classe é criada por **composição** de outras, tal consiste em definir que se entende que as classes agregadas fazem claramente *parte de*, ou são *partes*, da classe em definição. Assim, qualquer instância da nova classe vai ser constuída por partes que são instâncias das

diversas classes que foram agregadas. Estas partes, isto é, estas instâncias das classes agregadas que ajudaram a criar a instância da classe agregadora, têm uma ligação temporal, e até de tempo de vida, muito forte com a instância de que são parte. De facto, quando tal instância deixa de estar referenciada no ambiente de programação e é destruída pelo *garbage collector* de JAVA ou por um método destrutor explícito, tais partes são igualmente destruídas. O seu tempo de vida está completamente ligado ao tempo de vida da instância de que fazem parte.

Tem que ser claro que esta relação de composição nos permite, tal como no dia a dia, criar entidades complexas pela incorporação nestas de outras entidades com estruturas e comportamentos que podem ser também independentes, mas que, neste caso, vão servir como *partes* do todo que se pretende construir. Por exemplo, se se pretende especificar uma classe `Automóvel` e já temos definidas as classes `Pneu`, `Motor`, `Chassis`, `Travão`, etc., torna-se óbvio que a classe `Automóvel` deve ser criada usando composição, já que pretendemos que seja constituída por uma série de *partes* que, em conjunto, irão determinar a sua estrutura e comportamento.

Muitas vezes a existência de herança múltipla leva a que se tentem hierarquias como a apresentada na fugura abaixo, na qual uma interpretação errada do que se deve entender por herança múltipla, em particular se apenas se pretendia agregação, conduz a uma hierarquia segundo a qual um automóvel vai, supostamente, ter o seu comportamento e estrutura definidos herdando-os de 1 pneu, de 1 motor, de 1 chassis, de 1 travão e de 1 volante. Além do mais, apenas podemos ter 1 unidade de cada, o que não é manifestamente satisfatório.

Numa linguagem de herança simples esta situação não ocorreria certamente, dado que qualquer tentativa de classificar `Automóvel` como subclasse de `Pneu` ou de `Motor` pareceria de imediato tão ridícula que conduziria de imediato à rejeição de tal subclassificaçaão.



Interpretação errada da herança múltipla

Quando uma classe a ser criada, para além de quase inevitavelmente necessitar de possuir partes que são construídas por composição, for em tal estrutura e também no seu comportamento uma especialização ou refinamento de uma outra classe já definida, que certamente já agregou grande parte do que esta necessita de agregar, então estamos perante uma **especialização** ou **derivação**, e o mecanismo a usar será o de **subclassificação**, tendo por suporte o mecanismo de **herança**.

Dentro desta perspectiva, uma classe Automóvel nunca seria subclasse de `Pneu`, mas poderia ser uma subclasse de `Veículo_a_Motor`, ainda que tal dependesse da efectiva definição da classe `Veículo_a_Motor`, e seria ainda composta por um dado número de instâncias de `Pneu`, `Motor`, etc.

CRIAÇÃO DE CLASSES: REGRAS ANOTADAS

Apresentam-se em seguida algumas regras que sintetizam um conjunto extenso de directivas e princípios que foram sendo apresentados ao longo deste e de anteriores capítulos relativamente à criação de classes em JAVA (e na maioria das linguagens de PPO) e que são muito importantes para que, de forma metódica, sistemática e rigorosa, porque baseada em constatações, se possam atingir de facto os objectivos enunciados de garantia de concepção e desenvolvimento de aplicações possuindo propriedades fundamentais em Engenharia de Software, tais como modularidade, generalidade, clareza, extensibilidade e facilidade de manutenção.

Regra 1: ***Os dados devem ser sempre private.***

Caso se pretenda de forma rigorosa garantir a absoluta preservação das regras do encapsulamento, então todas as variáveis de instância devem ser declaradas como sendo `private`. Desta forma, e porque tais variáveis não são herdadas, a única forma de externamente ter acesso aos seus valores será através dos métodos de consulta e de alteração definidos na sua própria classe. Como vimos no capítulo 1, mantendo-se os dados privados, futuras mudanças na sua representação não geram quaisquer alterações nos utilizadores da classe e possibilitam detectar erros muito mais facilmente, já que os únicos métodos que as podem manipular são os métodos modificadores definidos na sua classe.

Porém, nunca poderemos esquecer que a completa obediência ao princípio do encapsulamento impõe a quem programa a responsabilidade de disponibiliar na API das classes os imprescindíveis métodos de consulta e modificação. Qualquer esquecimento desta responsabilidade conduz à criação de classes "surdas-mudas", ou seja, classes não utilizáveis do exterior, dado não existirem formas de acesso aos dados, nem directamente nem via métodos.

Regra 2: ***Inicializar sempre explicitamente os dados através de construtores.***

Apesar de sabermos as regras usadas pelos construtores na inicialização de dados, quer sejam de tipo simples quer sejam objectos, torna-se muito importante para a clareza e compreensão do código que todos os construtores de uma classe sejam programados, indicando de forma explícita quais os valores inicialmente atribuídos a todas as variáveis de instância. Desta forma, salvo em situações de herança, os construtores por omissão que JAVA associa implicitamente a cada nova classe, devem ser sempre redefinidos por um construtor programado que inicialize todas as variáveis necessárias de forma explícita, isto é, codificada.

Regra 3: ***Não usar demasiados tipos básicos numa classe. Quando tal acontecer procurar substituí-los, em grupo, por uma dada classe.***

Se numa classe são definidas diversas variáveis de instância independentemente, mas que possuem algum relacionamento entre si, tal significa que estas, no seu conjunto, podem ser estruturadas como sendo variáveis de instância de uma nova classe.

Um exemplo típico desta situação poderá ser a representação numa classe da ficha individual de um `Aluno` que foi definida como tendo, entre outras, as seguintes variáveis de instância:

```
private String rua;
private String numero;
private String localidade;
private String codigo_postal;
private String cidade;
```

Estas variáveis de instância isoladas, podem naturalmente ser em conjunto a base para a criação de uma classe `Morada`, que poderia ter a si associados métodos que permitissem alterar o código postal, editar o número da rua, enfim, alterar de uma forma geral a morada. Desta forma, para além desta funcionalidade ficar de forma clara associada a instâncias de `Morada` e não a instâncias de `Aluno`, na classe `Aluno` apenas teríamos que declarar:

```
private Morada mora;
```

Se porventura pretendessemos guardar a morada em tempo de aulas e a morada em tempo de férias, em vez de duplicarmos as 5 variáveis de instância inicialmente definidas na classe `Aluno`, teríamos apenas que declarar duas variáveis de instância do tipo `Morada` que, para além de permitirem representar tal informação, têm a si associados métodos específico para consulta e alteração.

```
private Morada mora_aulas;
private Morada mora_ferias;
```

Regra 4: ***Nem todas as variáveis de instância e classe necessitam de selectores e modificadores.***

Voltando ao exemplo da ficha de aluno, se tivermos uma variável de instância que representa a data da primeira inscrição do aluno na universidade, e/ou uma variável de instância que guarde a sua média de acesso, e/ou até uma instância de uma classe designada por `Filiação`, cujas instâncias contêm os nomes do pai e da mãe do aluno, e admitindo que garantidamente estes dados foram correctamente introduzidos aquando da criação da instância (como ?), então, dada a sua natureza, estes são claramente exemplos de dados e variáveis de instância imutáveis, ou seja, apenas consultáveis.

Noutras situações, as variáveis de instância não são sequer consultáveis. Todos os que jogam certos jogos de cartas em computador sabem que sendo o baralho do jogo gerado aleatoriamente, não é consultável. Todos os que jogam *Minesweeper* sabem que o campo de minas não é consultável, embora seja modificável.

Regra 5: ***Usar uma forma normalizada para as definições de classes.***

Qualquer que esta possa ser, desde que aceitável pelo compilador de JAVA, deve ser usada uma forma normalizada para a disposição das várias secções que formam uma declaração completa de uma classe de JAVA.
Na definição de tal forma normalizada, há que ter em atenção que os utilizadores de uma classe estão sempre mais interessados na interface pública da mesma, API, do que nos detalhes de implementação e, portanto, muito mais interessados em métodos do que em variáveis (às quais nem terão sequer acesso directo caso estas sejam `private`).

Nos exemplos apresentados ao longo deste livro, as declarações de classes foram divididas em secções apresentadas pela seguinte ordem:

Constantes de classe;
Variáveis de classe;
Métodos de classe;
Construtores;
Variáveis de instância;
Métodos de instância;
Métodos que implementam Interfaces.

Quanto à acessibilidade dos métodos, deve seguir-se um critério de importância do ponto de vista do seu exterior, sendo apresentados primeiro os métodos `public`, depois os métodos *package*, em seguida os `protected` e, por fim, os `private`.

Regra 6: ***Não criar classes com estrutura e APIs demasiado extensas; Caso tal aconteça, procurar fazer a sua divisão em classes mais pequenas.***

É uma regra de bom senso, dado não ser possível quantificar rigorosamente o que se deve entender por classes *demasiado extensas*. Em todo o caso, classes extensas denotam em geral demasiada concentração de responsabilidades numa única classe, pelo que pelo menos alguma preocupação com a sua extensão e clareza deverá ser tida em conta e, sempre que possível, realizar a divisão da classe em duas classes que serão certamente conceptualmente mais simples.

Regra 7: ***Ser criterioso nos identificadores de classes, variáveis e métodos. Que estes possuam a maior carga semântica possível.***

Ainda que em JAVA muitos maus exemplos de aplicação desta regra se possam encontrar, os nomes das variáveis, classes e métodos devem reflectir a sua semântica. As classes devem ser em geral identificadas usando substantivos como, por exemplo, `Contador`, `String`, `Factura`, `Encomenda`, ainda que por vezes a tal substantivo se possa associar um adjectivo, ou outra palavra qualquer, que indique de forma mais clara algum grau de especialização ou caso particular, tal como, por exemplo, `FacturaUrgente`, `StackLimitada`. Se os nomes das classes forem escritos em inglês, tais palavras surgem como prefixos e não sufixos como, por exemplo, `MainAddress`, `RadioButton` ou `HashTable`.

Os identificadores dos métodos devem denotar a acção associada, tendo já sido anteriormente referida a regra dos nomes `getX()` e `setX()` para os métodos que consultam e modificam variáveis de instância. Todos os outros são em geral *verbos* que representam as acções executadas.

Regra 8: ***Não confundir composição com herança.***

Sendo mecanismos de relacionamento entre classes muito distintos, a confusão de um com o outro conduz a classes conceptualmente muito distorcidas.

Regra 9: ***Ser estudioso, ou seja, ser um conhecedor profundo de tudo o que já existe implementado no ambiente de desenvolvimento, seja JDK ou um outro qualquer.***

Esta regra é de facto trabalhosa e a sua aplicaação dispendiosa em tempo, mas é de facto um investimento fundamental. Caso seja aplicada por fases, duas fases podem ser consideradas. Numa primeira fase, dever-se-á procurar ganhar conhecimento de classes existentes e apenas das suas APIs, ou seja, o que estas nos disponibilizam. Esta fase é, só por si, muito relevante para que se possa empregar uma estratégia de concepção e programação de aplicações com base em reutilização.

Numa segunda fase, a leitura e análise de código fonte de JAVA poderá ajudar-nos mais tarde a compreender melhor o próprio funcionamento da JVM e das coisas mais complexas do ambiente JAVA.

Regra 10: ***Ser preguiçoso, isto é, após cumprir a Regra 9, tentar ser apenas um reutilizador.***

EXERCÍCIOS PROPOSTOS

Para todos os exercícios propostos desenvolva em JAVA as classes especificadas, realize a compilação do código e, em seguida, crie e compile os programas de teste das mesmas.

1.- Crie uma classe `ParString` capaz de implementar uma estrutura formada por um par de strings, classe que deve possuir métodos para consultar cada uma das duas strings, alterar cada uma delas e, ainda, implementar os usuais métodos `toString`, `equals` e `clone`.

2.- Tendo por base a sua definição anterior da classe `ParString`, defina, em seguida, a classe `TriploString`, capaz de representar triplos que são strings, consultar cada uma destas, modificar cada uma destas e implementar também os métodos `toString`, `equals` e `clone`.

3.- Desenvolva uma classe `Aluno` *representativa da informação fundamental a registar sobre um aluno, designadamente: número, nome, curso, ano, código e nome das disciplinas a que está inscrito, morada e telefone.*
Desenvolva métodos para consulta das variáveis de instância, para modificação do curso e para modificação do ano (devendo ambos tornar vazia a lista das disciplinas), para alteração da morada ou telefone, para inserção e remoção de uma disciplina e para consulta do total de disciplinas a que o aluno está inscrito.

Sugestão: Crie uma classe `Disciplina`*, cujas instâncias vão conter o nome e o código de cada disciplina, como sendo subclasse da classe* `ParString`*.*

Que alterações deveria introduzir na estrutura desenvolvida para a classe `Aluno` *se pretendessemos associar a cada aluno a lista das disciplinas já realizadas e as classificações a estas obtidas?*

4.- Defina uma classe `ListOrdPares`*, cujas instâncias devem ser sequências ordenadas de pares, e em que os primeiros componentes dos pares são todos instâncias de uma classe indicada aquando da criação da instância (p.ex. pares cujos primeiros componentes são todos String ou Ponto2D).*

Todas as instâncias de `ListOrdPares` *devem ser capazes de responder às seguintes mensagens:*

- *daPrimeiroPar (devolve primeiro par da lista);*
- *daUltimoPar (devolve último par da lista);*
- *inserePar (insere na lista, ordenadamente, um novo par);*
- *existePar (verifica se o par parâmetro existe na lista);*
- *removePar (remove o par parâmetro, caso este exista);*
- *enumeraLista (cria uma Enumeration da lista de pares);*
- *juntaListaPares (junta à lista receptora a lista parâmetro).*

Sugestão: Analise a hierarquia das classe JAVA designada por `Collections`*, e veja qual a classe que mais funcionalidade para reutilização lhe oferece. Faça, em seguida, as devidas adaptações.*

5.- Construa uma classe `Empregado` *capaz de representar toda a informação que numa dada empresa é informatizada sobre cada um dos seus empregados, isto é, o seu número, nome, ano de nascimento, categoria, anos na empresa e salário mensal. Construa métodos de instância de acesso e modificação (excepto para número), e ainda os seguintes métodos particulares:*

- *Aumenta salário de um dado valor real;*
- *Aumenta salário de um valor real X por ano de trabalho na empresa.*

6.- Defina uma classe `Empresa`, *cujas instâncias correspondem a representações da estrutura de informação e da funcionalidade de uma empresa. Cada empresa possui uma tabela com os códigos e nomes dos seus diversos departamentos e uma tabela que, por categoria de empregado, define o valor do salário.*
Cada empresa possui também uma lista contendo todas as informações sobre cada um dos seus empregados.
A informação sobre cada empregado deverá contemplar: o seu número, nome, morada, telefone, nº de anos na empresa, categoria e nº de filhos.
Cada instância desta classe `Empresa` *deve ser capaz de responder, no mínimo, às seguintes mensagens:*

- *Determinação do número total de funcionários de dada categoria;*
- *Determinação do número total de diferentes categorias na empresa;*
- *Qual o salário actual de uma categoria;*
- *Número total de empregados de uma dada categoria;*
- *Criação e remoção de empregados;*
- *Transferência de um empregado de um departamento para outro;*
- *Cálculo do total de vencimentos a pagar no final do mês, sabendo-se que por cada filho um empregado recebe um valor estipulado pela empresa, o mesmo se passando relativamente a cada ano de serviço;*
- *Aumento de vencimentos: dada uma tabela com o valor do aumento por categoria, o método deve actualizar os vencimentos;*
- *Criação de uma tabela com o total de vencimentos por departamento.*

7.- Desenvolva um Sistema de Gestão de Contas Bancárias capaz de realizar as diversas operações bancárias sobre contas, mas tendo em atenção que existem de momento definidos três diferentes tipos de conta: Normal, Vencimento e a Prazo.
Todas as contas têm um número, um titular, uma data de criação e um saldo. As contas normais não permitem que o saldo seja alguma vez inferior a 0. As contas vencimento permitem que o saldo seja negativo até um valor predefinido, mas que pode ser mudado. A conta a prazo rende juros, dependendo do tempo que cada depósito permaneça na conta, juros esses definidos numa tabela própria.

Sobre todas as contas pretende-se ter disponíveis as seguintes operações:

- *Criação de conta (com validações);*
- *Levantamento de uma quantia;*
- *Depósito de uma quantia;*
- *Consulta do saldo actual;*
- *Consulta dos últimos 5 movimentos (depósitos e levantamentos).*

CLASSES ABSTRACTAS

Introdução às Classes Abstractas.

❒ Todas as classes até agora apresentadas definem completamente a estrutura e o comportamento das suas instâncias, ou seja, definem as variáveis de instância que cada instância possuirá como sendo o seu estado interno, e definem todos os métodos correspondentes às mensagens a que as instâncias são capazes de responder.

➨ Porém, a maioria das linguagens de PPO (cf. Smalltalk, C++, Eiffel, Objective-C, Java, etc.) permite que a definição de uma classe possa ser incompleta, ou seja, que, por exemplo, alguns métodos (ou mesmo todos) possam ser apresentados sintacticamente mas não sejam implementados, isto é, não possuam corpo ou código.

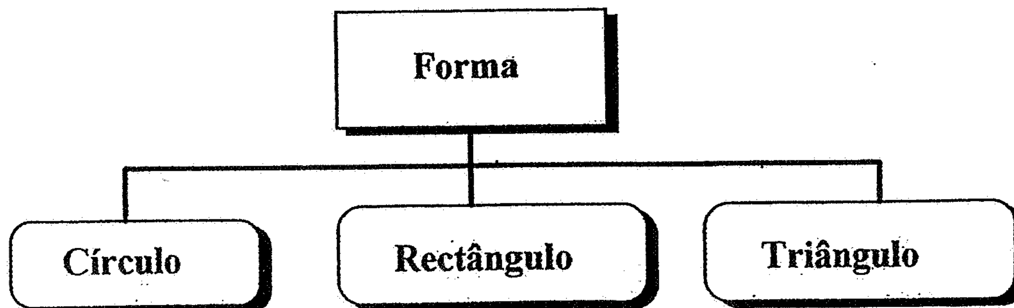
➨ **Classes Abstractas** são exactamente todas as classes nas quais *alguns* ou *todos* os métodos de instância não se encontram implementados (sendo portanto *métodos abstractos* ou *virtuais*) mas apenas declarados sintacticamente

➨ Em Java classes abstractas e métodos abstractos são definidos utilizando o qualificador `abstract` antes das suas definições, conforme o exemplo apresentado a seguir.

```
public abstract class Forma {
//
   public abstract double area() ;
   public abstract double perimetro() ;
}
```

❒ Sendo de notar que para que uma classe deva ser considerada *abstracta* é suficiente que não tenha implementado apenas 1 dos seus diversos métodos, torna-se igualmente evidente que, por tal motivo, **uma classe abstracta não pode criar instâncias**. De facto, estas intâncias, caso pudessem ser criadas, conduziriam a situações de erro dado não serem capazes de responder às mensagens correspondentes aos métodos de instância não implementados.

Consideremos a título de exemplo a seguinte hierarquia de classes que irá servir de referencial para as considerações teóricas e práticas que iremos apresentar sobre a importância de se possuir *classes abstractas* em PPO, hierarquia na qual a classe `Forma` é uma classe abstracta.



❒ A primeira questão que é usual levantar-se relativamente à existência de classes abstractas é a sua utilidade, sabendo-se que não podem criar instâncias sequer. No entanto, as classes abstractas são um mecanismo muito importante em PPO, por várias razões que passaremos a expor de seguida.

❒ Em primeiro lugar deve chamar-se a atenção para o facto de que *o mecanismo de herança se mantém em vigor mesmo que a classe seja abstracta*. Deste simples facto começam a surgir as mais diversas implicações de podermos ter classes abstractas na hierarquia.
Assim, qualquer subclasse da classe abstracta herda os métodos abstractos não implementados. Ao herdar os métodos abstractos, a subclasse poderá implementá-los ou não. Se a subclasse implementar *todos* os métodos abstractos herdados, então passará a ser uma classe concreta, ou seja, não-abstracta, enquanto que se deixar um qualquer destes métodos herdados por implementar será igualmente uma classe abstracta tal como a sua superclasse.

❒ Na hierarquia exemplo, a classe `Forma` é abstracta, como se apresentou, pelo que as suas subclasses `Círculo`, `Rectângulo` e `Triângulo` só não o serão também se fornecerem implementações para os dois métodos abstractos herdados.

❒ Uma classe abstracta, ao não implementar certos (ou todos) métodos, *delega* nas suas subclasses (ou melhor *coage*) a implementação particular de tais métodos, facilitando no entanto o aparecimento de diferentes implementações dos mesmos métodos.
Assim, enquanto que na relação normal entre classes e subclasses a redefinição de métodos é opcional, na relação entre classes abstractas e suas subclasses a (re)definição de métodos é uma obrigação para que tenhamos implementações concretas da classe abstracta. No entanto, é exactamente neste ponto que reside uma das grandes vantagens de possuirmos classes abstractas.

De facto, e tomando por exemplo uma classe 100% abstracta, o mecanismo de herança vai garantir que, dado que todas as suas subclasses vão herdar o mesmo *protocolo* ou *API* (ou seja, o conjunto de métodos abstractos definidos apenas sintacticamente), então, salvo extensões sempre possíveis, todas as subclasses da classe abstracta responderão ao mesmo *protocolo*, ou seja, "falam" a mesma linguagem que foi herdada (ainda que cada uma o possa fazer à sua maneira, isto é, conforme a sua implementação particular !!).

Portanto, e tomando por exemplo a hierarquia apresentada, as instâncias das classes Círculo, Rectângulo e Triângulo, caso estas subclasses de Forma não sejam elas próprias abstractas, deverão todas responder às mensagens `area()` e `perimetro()`. Por outro lado, quaisquer futuras subclasses destas (exº TriânguloRectângulo, Quadrado, etc.), continuarão a ter que responder a tais mensagens, segundo métodos por si especificamente implementados.

❒ A introdução de uma *classe abstracta* na hierarquia de classes tem por primeiro possível objectivo, do ponto de vista do programador, garantir que, conforme se procurou explicar anteriormente, todas as subclasses da classes abstracta introduzida respeitem e respondam a uma mesma *linguagem comum*, ou seja, implementem os mesmos métodos e, portanto, possam responder às mesmas mensagens (o polimorfismo da linguagem garantirá a correcta execução).
Uma classe 100% abstracta pode ser vista como uma especificação meramente sintáctica (ou seja, uma *assinatura* Σ), para a qual o mecanismo automático de herança impõe a construção de implementações obedecendo à linguagem definida na *assinatura* Σ, ou seja, Σ-modelos ou

Σ-implementações, eventualmente algumas delas enriquecidas, isto é, aumentadas em funcionalidade.

❑ Em resumo, *classes abstractas* são um mecanismo muito importante em PPO dado que permitem ao programador :

- *Escrever especificações para as quais múltiplas implementações são possíveis;*
- *Normalizar a "linguagem" (API) a partir de certos pontos da hierarquia;*
- *Introduzir flexibilidade e generalidade nas classes criadas;*
- *Tirar todo o partido do polimorfismo.*

Utilização de Classes Abstractas.

Consideremos a hierarquia de classes anteriormente apresentada, e vejamos de seguida as definições detalhadas de cada uma destas classes, dado que nos irão servir de exemplo nas considerações seguintes sobre classes abstractas.

```
public abstract class Forma {
//
     public abstract double area();
     public abstract double perimetro();
}
```

`Forma` é uma classe abstracta que impõe às suas subclasses a implementação dos métodos abstractos `area()` e `perimetro()`, que retornam um valor do tipo `double`.

```
public class Circulo extends Forma {
// Constante PI para cálculos dentro da classe
     protected static final double PI=3.1415926535897;
// Variável de instância = Raio do círculo
     protected double r;
// Construtores
     Circulo() { r=1.0; }
     Circulo(double r) { this.r = (r<= 0) ? 1.0 ; r); }
// metodos de instancia
     public double area() { return PI*r*r; }
     public double perimetro() { return 2*PI*r; }
     public double raio() { return r; }
}
```

A classe `Circulo` é uma subclasse concreta de `Forma`, dado ter implementado os dois métodos abstractos herdados, tendo ainda definido uma constante de classe, uma variável de instância `r` e um método adicional, `raio()`. As instâncias da classe Circulo possuem construtores próprios, como é normal, mas o construtor que aceita um raio como parâmetro faz a validação de tal valor por forma a que não sejam criados círculos com raios negativos ou nulos.

```
public class Rectangulo extends Forma {
      protected double comp, larg;
// Construtores
      Rectangulo() { comp=0.0; larg=0.0; }
      Rectangulo(double c, double l) { comp=c; larg=l; }
// metodos de instancia
      public double area() { return comp*larg; }
      public double perimetro() { return 2*(comp+larg); }
      public double largura() { return larg; }
      public double comp() { return comp; }
}
```

As classes seguintes, `Rectangulo` e `Triangulo`, definem igualmente as suas variáveis de instância apropriadas, bem como métodos de instância adicionais às "obrigatórias" implementações dos métodos abstractos herdados.

```
public class Triangulo extends Forma {
//
      protected double base, altura;
// Construtores
   Triangulo() { base=0.0; altura=0.0; }
   Triangulo(double b, double a) { base=b; altura=a; }
// metodos de instancia
      public double area() { return base*altura/2; }
      public double perimetro() {
            return base+altura+this.hipotenusa(); }
      public double base() { return base; }
      public double altura() { return altura; }
      public double hipotenusa() {
         return Math.sqrt(Math.pow(base, 2.0) +
                              Math.pow(altura, 2.0)) ; }
}
```

Consideremos agora um programa que testa estas classes e produz um conjunto de resultados que procuram mostrar como tais classes podem ser utilizadas:

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

Triangulo forma1 = new Triangulo(2.0, 6.0);
Rectangulo forma2 = new Rectangulo(2.0, 13.0);
Circulo forma3 = new Circulo(3.0);

System.out.println("forma1 = " + forma1.getClass());
System.out.println("forma2 = " + forma2.getClass());
System.out.println("forma3 = " + forma3.getClass());

System.out.println("---------- TRIANGULO ----------");
System.out.println(forma1.base());
System.out.println(forma1.altura());
System.out.println(forma1.area());
System.out.println(forma1.perimetro());
```

```
System.out.println("---------- RECTANGULO ----------");
System.out.println(forma2.comp());
System.out.println(forma2.largura());
System.out.println(forma2.area());
System.out.println(forma2.perimetro());

System.out.println("---------- CIRCULO ----------");
System.out.println(forma3.raio());
System.out.println(forma3.area());
System.out.println(forma3.perimetro());
}
}
```

Vejamos os resultados produzidos pela execução deste programa:

```
c:\jdk1.1> java TstAbsForms
forma1 = class Triangulo
forma2 = class Rectangulo
forma3 = class Circulo
---------- TRIANGULO ----------
2
6
6
14.3246
---------- RECTANGULO ----------
2
13
26
30
---------- CIRCULO ----------
3
28.2743
18.8496

c:\jdk1.1>
```

Os resultados do programa são obviamente os esperados dado que este apenas criou instâncias das classes concretas `Rectangulo`, `Triangulo` e `Circulo`, às quais enviou as mensagens a que estas, conforme os métodos implementados, eram capazes de responder.

Vamos nos programas de teste seguintes analisar a compatibilidade entre a classe abstracta e as suas subclasses concretas (ie., implementações).

Comecemos por construir um programa que declara variáveis da classe abstracta `Forma`, para as quais vamos tentar associar instâncias das subclasses de `Forma`, e verificar qual o comportamento de tais instâncias.

```
public class TstAbsForms1 {
//
public static void main(String args[]) {

Forma forma1 = new Triangulo(2.0, 6.0);
Forma forma2 = new Rectangulo(2.0, 13.0);
Forma forma3 = new Circulo(3.0);

System.out.println("forma1 = " + forma1.getClass());
System.out.println("forma2 = " + forma2.getClass());
System.out.println("forma3 = " + forma3.getClass());

System.out.println("---------- TRIANGULO ----------");
System.out.println(forma1.area());
System.out.println(forma1.perimetro());

System.out.println("---------- RECTANGULO ----------");
System.out.println(forma2.area());
System.out.println(forma2.perimetro());

System.out.println("---------- CIRCULO ----------");
System.out.println(forma3.area());
System.out.println(forma3.perimetro());
}
}
```

Vejamos os resultados produzidos por este programa para que possamos realizar a análise:

```
c:\jdk1.1> java TstAbsForms1
forma1 = class Triangulo
forma2 = class Rectangulo
forma3 = class Circulo
---------- TRIANGULO ----------
6
14.3246
---------- RECTANGULO ----------
26
30
---------- CIRCULO ----------
28.2743
18.8496

c:\jdk1.1>
```

❒ Como se pode deduzir dos resultados apresentados, a associação a variáveis da superclasse abstracta `Forma` de instâncias das subclasses concretas `Rectangulo`, `Triangulo` e `Circulo`, não provocou qualquer incompatibilidade, mesmo sendo esta uma superclasse *abstracta*. Pode assim inferir-se que qualquer referência a uma superclasse (abstracta ou não) pode igualmente referenciar uma qualquer subclasse (ou seja, as respectivas instâncias) da mesma.

❐ Uma das mais interessantes características da maioria das linguagens de PPO é o facto de o *valor* referenciado por uma variável poder não corresponder exactamente ao tipo da sua declaração, conforme o exemplo :

```
Forma forma1 = new Triangulo(2.0, 6.0);
```

⇒ Esta possibilidade resulta em particular das características de *polimorfismo* das linguagens de PPO, e ainda das noções de *tipo estático* (ou seja, o tipo associado a uma variável através de uma declaração e que é verificado pelo compilador) e tipo dinâminco (ou seja, o tipo do valor que em tempo de execução é associado a uma variável, e que é verificado pelo interpretador mas não pode ser verificado pelo compilador).
O facto de uma variável poder conter um valor de um *tipo dinâmico* que é diferente do seu *tipo estático* declarado (ainda que dentro de certas restrições, tais como as anteriormente referidas), é uma fonte de flexibilidade nas linguagens de PPO que lhes confere um incomparável poder expressivo e extensibilidade.
Procuraremos na secção designada por *Sublasses e Subtipos em Java* apresentar de um modo mais rigoroso a relação entre classes e tipos na linguagem Java.

❐ Como se pode deduzir igualmente pelos resultados apresentados, em *tempo de execução* o interpretador de Java é capaz de distinguir correctamente a que subclasse de `Forma` pertence efectivamente cada instância criada. A correcção dos resultados obtidos em resultado do envio das mensagens `area()` e `perimetro()` às diversas instâncias são disso exemplo. Ou seja, o mecanismo de *dynamic binding* (ie., associação em tempo de execução de métodos a instâncias, com a correcta execução do respectivo código) funciona de forma eficiente.
Portanto, em tempo de execução o interpretador é capaz de distinguir que o código a executar para o cálculo da área de um `Triangulo` é diferente do código a executar para o cálculo da área de um `Circulo`, executando assim o código correspondente ao método de instância programado para a instância em causa.

⇒ Assim, *para todos os métodos abstractos declarados na classe abstracta e implementados nas suas subclasses concretas* (cf. `area()` e `perimetro()` na classe `Form`), não só o compilador de Java é capaz de reconhecer a compatibilidade dos tipos, como o interpretador, através do mecanismo de *dynamic binding*, é capaz de seleccionar o código adequado à execução.

⇒ O problema seguinte em Java é determinar se os *métodos específicos* de uma subclasse de uma superclasse abstracta são reconhecidos, em tempo de compilação e/ou em tempo de execução, quando associamos a variáveis do tipo da superclasse instâncias das suas subclasses. Isto é, tomando os exemplos anteriores, se se criar uma variável do tipo `Forma` e se a mesma for associada a uma instância da classe `Rectangulo`, será correcto, e reconhecido pelo compilador, o envio da mensagem `largura()` definida na classe `Rectangulo` ?

O programa seguinte esclarece este tipo de dúvida de forma clara, ao criar um Array de formas e ao atribuir a certas posições do Array diferentes instâncias de subclasses de `Forma`. O programa procura realizar de seguida diferentes manipulações de tais instâncias, em particular enviando a tais instâncias mensagens, a que, como vimos anteriormente, estas são capazes de responder.

```
public class TstForm {

//
static Forma[] formas;

public static void main(String args[]) {
//
formas = new Forma[20];
formas[1] = new Triangulo(2.0, 6.0);
formas[2] = new Rectangulo(2.0, 13.0);
formas[3] = new Circulo(3.0);

System.out.println(formas[1].getClass());
System.out.println(formas[2].getClass());
System.out.println(formas[3].getClass());

System.out.println("---------- TRIANGULO ----------");
System.out.println(formas[1].base());
System.out.println(formas[1].altura());
System.out.println(formas[1].area());
System.out.println(formas[1].perimetro());

System.out.println("---------- RECTANGULO ----------");
System.out.println(formas[2].comp());
System.out.println(formas[2].largura());
System.out.println(formas[2].area());
System.out.println(formas[2].perimetro());

System.out.println("---------- CIRCULO ----------");
System.out.println(formas[3].raio());
System.out.println(formas[3].area());
System.out.println(formas[3].perimetro());
}
}
```

O programa produz no entanto os seguintes erros de compilação:

```
c:\jdk1.1>javac TstForm
  compiling: TstForm.java
TstForm.java(19) : Method base() not found in class Forma.
System.out.println(formas[1].base());
                                 ^

TstForm.java(20) : Method altura() not found in class Forma.
System.out.println(formas[1].altura());
                                 ^

TstForm.java(25) : Method comp() not found in class Forma.
System.out.println(formas[2].comp());
                                 ^

TstForm.java(26) : Method largura() not found in class Forma.
System.out.println(formas[2].largura());
                                 ^

                    ..........................................
```

⇒ Podemos portanto concluir que o compilador de Java reconhece as referências designadas por formas[1], formas[2], etc., como sendo do tipo `Forma`, independentemente de terem sido associadas a instâncias das subclasses de `Forma`, pelo que, naturalmente, não reconhece os métodos específicos destas subclasses nas instâncias de `Forma`.
Recorde-se aqui de novo a diferença entre *tipos estáticos* e *dinâmicos* atrás apresentada, e que explica tais erros de compilação.

⇒ Podemos igualmente concluir que todas as instâncias de subclasses de `Forma` podem ser atribuídas a variáveis da superclasse `Forma`, sendo ainda reconhecidas as mensagens que se referem ao protocolo comum definido na superclasse abstracta (e por todas as outras herdado e, no exemplo, implementado.

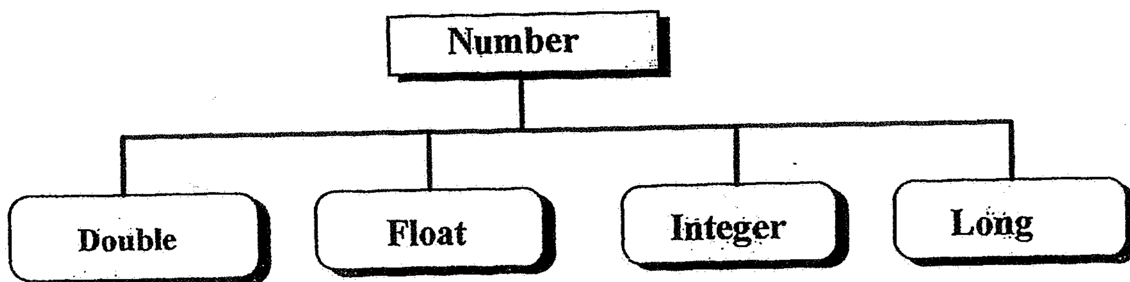
⇒ Torna-se portanto claro que, do ponto de vista da criação de classes abstractas será sempre importante que estas tenham declarado o maior número de métodos abstractos possível, já que esta será a linguagem comum as todas as subclasses concretas que vierem a ser construídas. Este será o *comportamento* comum a todas estas subclasses. Para o determinarmos basta consultar a API da classe abstracta.
As classes abstractas funcionam portanto também como classes de normalização linguística na hierarquia de classes.

⇒ Em função do que acabou de ser dito no ponto anterior, torna-se evidente que a classe que serviu de exemplo à introdução das classes abstractas, `Forma`, deveria ter declarados muitos outros métodos correspondentes à funcionalidade comum a qualquer forma concreta que possa vir a ser criada.

Considerações Finais e Exemplos Java

❒ Como mecanismo de abstracção, as classes abstractas permitem que a concepção e o desenvolvimento de software possam ser realizados segundo uma metodologia de *refinamento progressivo por especialização*, ou seja, construindo gradualmente classes mais concretas até se possuirem implementações completas, ao contrário das metodologias tradicionais que promoviam o *refinamento progressivo por decomposição* ou subdivisão dos problemas.

❒ As classes abstractas tendem naturalmente a ocupar os níveis mais elevados das hierarquias de classes, dado serem muito pouco físicas e muito mais conceptuais (especificações). A figura seguinte mostra uma pequena hierarquia de classes Java tendo no topo uma classes abstracta.



❒ As classes abstractas são por vezes também utilizadas não tanto com a finalidade de "normalizar" protocolos, mas como máximos denominadores comuns de classes para as quais não é facilmente verificável uma relação de inclusão dado, por exemplo, consistirem em duas implementações distintas (particularmente em estrutura) de um mesmo problema.

A título de exemplo, vamos considerar que pretendemos construir duas implementações distintas de *Correspondências Chave-Valor* (também designadas por *Funções Finitas*). Procurando que estas sejam o mais genéricas possível decidimo-nos por duas implementações de correspondências de Objecto para Objecto.

Porém, a primeira implementação, designada por *SmallIndex*, assume que tais correspondências nunca atingirão uma grande dimensão (apenas algumas centenas de pares) e, portanto, decide por uma representação baseada num *Array de Chaves* e num *Array de Valores*. Quanto ao protocolo definido, o mesmo corresponde ao que é usual admitir-se como indispensável para a manipulação deste tipo de estruturas.
Apresenta-se de seguida o código completo da classe Java desenvolvida:

```
public class SmallIndex {
//
// variaveis de instancia

protected Object[] chaves ;       // array de chaves
protected Object[] valores;       // array de valores
protected int numElems;           // contador de pares
protected int MAX_ELEMS;          // limite máximo

// Construtores

SmallIndex() {
   MAX_ELEMS = 100;
   chaves = new Object[MAX_ELEMS];
   valores = new Object[MAX_ELEMS];
   numElems = 0;
}

SmallIndex(int max) {
   MAX_ELEMS = (max > 0 ? max : 100);
   chaves = new Object[MAX_ELEMS];
   valores = new Object[MAX_ELEMS];
   numElems = 0;
}

// metodos de instancia

int length() {
   return numElems;
}

boolean isEmpty() {
  return (numElems == 0);
}

boolean isFull() {
    return (numElems == MAX_ELEMS - 1) ;
}
```

```
Object elementAt(Object chave) throws IndexErrorException {
    if ( (! this.includesKey(chave)) || this.isEmpty() )
         throw new IndexErrorException();
    else
        { int indice = searchKey(chave); return valores[indice]; }
}

boolean includesKey(Object chave) {
  boolean encontrada = false;
  int ind = 0;
  //
  while(!encontrada && ind < numElems) {
     if (chaves[ind].equals(chave))
        encontrada = true;
     else
        ind +=1;
  }
  return encontrada;
 }

 private int searchKey(Object chave) {
   boolean encontrada = false;
   int ind = 0;
   //
   while(!encontrada && ind < numElems) {
      if (chaves[ind].equals(chave))
         encontrada = true;
      else
         ind +=1;
   }
   if (encontrada) return ind; else return 0 ;
 }

 void insKeyValue(Object chave, Object valor) throws
                                               IndexErrorException {
    if (this.includesKey(chave) || this.isFull())
       throw new IndexErrorException();
    else
       { chaves[numElems] = chave;
         valores[numElems] = valor;
         numElems += 1;
       }
 }

 Enumeration keys() {
    Vector dom = new Vector(this.length());
    int i;
    //
    for (i=0; i < this.length()-1; i++) {
        dom.addElement(chaves[i]);
    }
    return dom.elements();
 }

 Enumeration values() {
    Vector cdom = new Vector(this.length());
    int i;
    //
    for (i=0; i < this.length()-1; i++) {
        cdom.addElement(valores[i]);
    }
    return cdom.elements();
 }
}
```

A segunda implementação, designada por *BigFastIndex*, assume que tais correspondências atingirão grandes dimensões e decide por uma representação mais eficiente baseada na utilização de uma *Hashtable*, utilizando no entanto o protocolo definido como "standard" para a manipulação destas correspondências. Esta implementação baseada em *Hashtable* poderia ter usado herança, ou seja, tornar-se uma subclasse da classe *Hashtable*, tendo porém decidido usar o mecanismo de agregação (ou seja, usar uma variável própria que é instância de Hashtable), e, sem compromissos de hierarquia e herança, poder tornar-se subclasse de qualquer outra classe.

Vejamos o respectivo código, notando desde logo como a possibilidade de reutilização conduz a certas poupanças definicionais:

```
public class BigFastIndex {
//
// variaveis de instancia

protected Hashtable index;
protected int MAX_ELEMS;

// Construtores

BigFastIndex() {
   index = new Hashtable();
   MAX_ELEMS = 101;    // conforme o construtor de Hashtable
}

BigFastIndex(int max) {
   index = new Hashtable(max);
   MAX_ELEMS = max;
}

// metodos de instancia

int length() {
   return index.size();
}

boolean isEmpty() {
  return index.isEmpty();
}

boolean isFull() {
    return  (index.size() == MAX_ELEMS);
}

Object elementAt(Object chave) throws NullPointerException {
          Object r = index.get(chave);
          if ( r == null)
             throw new NullPointerException();
          else return r;
}

boolean includesKey(Object chave) {
  return index.containsKey(chave);
}
```

```
void insKeyValue(Object chave, Object valor) throws
IndexErrorException {
   if (index.containsKey(chave) || this.isFull())
      throw new IndexErrorException();
   else
      index.put(chave, valor);
}

Enumeration keys() {
  return index.keys();
}

Enumeration values() {
  return index.elements();
}
}
```

Vejamos os resultados obtidos pela execução do seguinte programa de teste:

```
import java.util.*;

public class TstIndex {
  //
  public static void main(String args[]) {

  BigFastIndex index1 = new BigFastIndex();
  SmallIndex index2 = new SmallIndex();

  System.out.println("index1 vazio ? " + index1.isEmpty());
  System.out.println("index2 vazio ? " + index2.isEmpty());

  System.out.println("index1 tamanho = " + index1.length());
  System.out.println("index2 tamanho = " + index2.length());

   try { index1.elementAt("LESI"); }
   catch(NullPointerException e)
      { System.out.println("ERRO DE NULL POINTER !"); }

   try
     { index1.insKeyValue(new String("LESI"), new Integer(800));
       index2.insKeyValue(new String("LESI"), new Integer(800));

       index1.insKeyValue(new String("LMCC"), new Integer(400));
       index2.insKeyValue(new String("LMCC"), new Integer(400));

       index1.insKeyValue(new String("LEPS"), new Integer(300));
       index2.insKeyValue(new String("LEPS"), new Integer(300));

       index1.insKeyValue(new String("LME"), new Integer(100));
       index2.insKeyValue(new String("LME"), new Integer(100));

       // duas insercoes erradas !!
       index1.insKeyValue(new String("LME"), new Integer(150));
       index2.insKeyValue(new String("LME"), new Integer(150));

     }
   catch(IndexErrorException e)
      { System.out.println("ERRO DE INSERCAO EM INDEX !"); }

   System.out.println("Index1 = " + index1.length() + " elementos");
   System.out.println("Index2 = " + index2.length() + " elementos");
```

```
try
  { System.out.println("LESI : " + index1.elementAt("LESI"));
    System.out.println("LESI : " + index2.elementAt("LMCC"));   }
catch(IndexErrorException e)
   { System.out.println("ERRO DE PROCURA EM INDEX !"); }

System.out.println(index1.includesKey("LMCC"));
System.out.println(index2.includesKey("MCC"));

System.out.println(index1.keys());     // nao , visualizavel !!
System.out.printIn(index1.values());

// Codigo para visualização de Chaves e Valores

Enumeration chaves1 = index1.keys();
Enumeration valores1 = index1.values();

while(chaves1.hasMoreElements()) {
    System.out.println(chaves1.nextElement());
}

while(valores1.hasMoreElements()) {
    System.out.println(valores1.nextElement());
}

}
}
```

Os resultados produzidos pelo programa são os seguintes:

```
c:\jdk1.1> java TstIndex
index1 vazio ? true
index2 vazio ? true
index1 tamanho = 0
index2 tamanho = 0
ERRO DE NULL POINTER !
ERRO DE INSERCAO EM INDEX !
Index1 = 4 elementos
Index2 = 4 elementos
LESI : 800
LMCC : 400
true
false
java.util.HastableEnumerator@1653c10
java.util.HastableEnumerator@1653c98
LEPS
LESI
LME
LMCC
300
800
100
400

c:\jdk1.1
```

❑ Note-se desde já, tal é o objectivo do exemplo, que ambas as implementações produzidas possuem os mesmos métodos de criação, consulta e manipulação, designadamente, possuindo a API seguinte:

- *SmallIndex()*
- *SmallIndex(int)*
- int length()
- boolean isEmpty()
- boolean isFull()
- Object elementAt(Object)
- boolean includesKey(Object)
- void insKeyValue(Object, Object)
- Enumeration keys()
- Enumeration values()

- *BigFastIndex()*
- *BigFastIndex(int)*

Esta será igualmente a API de base para a construção da classe abstracta de que as duas implementações serão subclasses, ou seja, a classe:

```
import java.util.*;

public abstract class Index {
//
  abstract int length();
  abstract boolean isEmpty();
  abstract boolean isFull();
  abstract Object elementAt(Object chave)
                        throws NullPointerException,
                               IndexErrorException;
  abstract boolean includesKey(Object chave);
  abstract void insKeyValue(Object chave, Object valor)
                        throws IndexErrorException;
  abstract Enumeration keys();
  abstract Enumeration values();
}
```

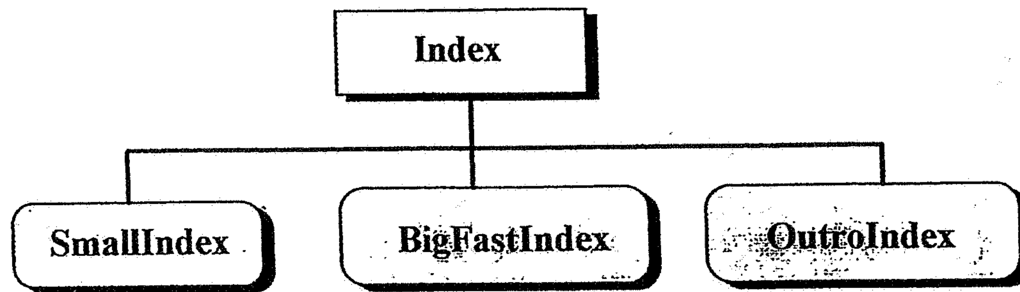
Note-se a obrigatoriedade de declarar as excepções que podem ser lançadas por alguns métodos. Por exemplo, o método `elementAt()` declara poder lançar duas excepções distintas ao ser implementado nas subclasses. De facto em `SmallIndex` este método pode lançar a excepção `IndexErrorException` enquanto que em `BigFastIndex` ele pode lançar a excepção `NullPointerException`.

⇒ Portanto, o *somatório das excepções* que podem ser lançadas pelas implementações de um dado método abstracto devem ser declaradas na classe abstracta.

As classes SmallIndex e BigFastIndex deverão agora ser colocadas como subclasses desta através da alteração das suas cláusulas de classe que passarão a ser:

```
public class BigFastIndex extends Index {

public class SmallIndex extends Index {
```

Passaremos a ter a hierarquia,



❐ Importante salientar que o código do programa de teste e os seus resultados permaneceram inalterados apesar das alterações realizadas nas classes exemplo.

❐ A criação de outras implementações de `Index` tem cùsto zero, ou seja, segue os mesmos passos que realizamos para as impelmentações de `SmallIndex` e `BigFastIndex`.

❐ Ainda que neste exemplo a classe abstracta Index seja 100% abstracta, ou seja, nada possua implementado que possa ser herdado por todas as suas implementações, convém referir que em certas situações faz todo o sentido, e é perfeitamente possível, colocar na classe abstracta partes de código (a situação mais comum) ou mesmo constantes ou variáveis que, desta forma, são comuns a todas as suas possíveis implementações. A classe constinuará a ser abstracta desde que algum método de instância ainda o seja.

❐ Quanto mais funcionalidade puder ser colocada numa superclasse (quer esta seja ou não abstracta) melhor optimização do espaço funcional estamos a fazer, dado que não será necessário replicar tal código em cada uma das implementações (ou subclasses) da superclasse. O mecanismo de herança garante a acessibilidade de qualquer instância a tal código partilhado.
O mesmo se poderá dizer relativamente a variáveis de instância, ainda que, como sabemos, a partilha de estrutura (ie. dados) seja bastante mais complexa e de difícil alteração.

❐ A linguagem Smalltalk, dado não possuir um sistema forte de tipos e utilizar extensamente o mecanismo de *"dynamic binding"*, recorre de uma forma mais intensa às classes abstractas e à colocação nestas de porções de código comum a todas as futuras possíveis implementações.

⇒ Em Java a filosofia é um pouco diferente dado que a linguagem possui mecanismos alternativos de construir implementações distintas para a mesma especificação abstracta de um tipo de dados. De facto, poucas são as classes Java 100% abstractas e muito pouco é o código comum nestas colocado. Porém, Java possui um muito interessante mecanismo adicional de associar implementações realizadas em classes concretas a especificações 100% abstractas de tipos de dados. Este mecanismo, que será estudado de seguida, é o mecanismo naturalmente mais próximo em Java das noções formais de assinatura algébrica (Σ) de um TAD, tendo como pares noutras linguagens os `DEFINITION MODULES` de MODULA-2, as `signatures` de ML, etc., e designa-se em Java o mecanismo das *INTERFACES*.

POLIMORFISMO vs. CASES

Exemplo:

Pretende-se definir uma Casa como sendo uma agregação de um número desconhecido de divisões, sendo certo que cada instância particular de cada divisão, qualquer que seja o seu tipo, responderá à mensagem da_dim(), que dará como resultado uma String indicativa da sua dimensão - "pequena", "média" ou "grande".

Cada instância da classes Casa deve ter implementado um método mostraTamanhos que apresenta as dimensões das diferentes divisões.

SOLUÇÃO 1:

Criar as classes correspondentes às mais diversas divisões, sendo certo que para este construtor WC, Cozinha, Sala, etc., terão sempre dimensões fixas para cada casa que ele construa, podendo no entanto variar em número.

```
class WC {
     String da_Dim() { return "pequena"; }
}

class Cozinha {
     String da_Dim() { return "média"; }
}

class Sala {
     String da_Dim() { return "grande"; }
}
=================================

class Casa {
     private Object[] quartos = new Object[3];

// Construtor

      Casa() {
      quartos[0] = new WC();
      quartos[1] = new Cozinha();
      quartos[2] = new Sala();
}
```

```
// métodos de instância
public void mostraTamanhos() {
   for (int i = 0; i < quartos.length; ++i) {
     String tam;
     if (quartos[i] instanceOf Sala)
          tam = ((Sala) quartos[i]).da_Dim();
     else if (quartos[i] instanceOf Cozinha)
               tam = ((Cozinha) quartos[i]).da_Dim();
          else if (quartos[i] instanceOf WC)
                    tam = ((WC) quartos[i]).da_Dim();
          return tam;
   }}}
```

Problema:

Quem escreve código Java deste tipo não percebe o que é o polimorfismo, e, portanto, não o sabe utilizar em seu próprio benefício - em termos de esforço de escrita de programas -, nem em benefício destes, pois não os consegue tornar extensíveis, genéricos e simples.

Apesar de o codificador poder dizer que tanto entendeu polimorfismo que usou uma variável de uma superclasse - *Object* - para conter instâncias das suas subclasses, a verdade é que, ao usar **instanceOf**, ele comprometeu o que disse ter entendido. Quem entende o que é polimorfismo não usa **instanceOf**, excepto em situações muito especiais.

Quem entende o que é polimorfismo teria criado uma classe que pudesse ser a superclasse de todas as classes representativas das divisões da casa - mas não Object -, talvez *Quarto*, cf.

```
abstract class Quarto {
    abstract String da_dim();
}
```

e teria definido as diferentes possíveis divisões como sendo subclasses de Quarto, cf.

```
class Cozinha extends Quarto {
    String da_Dim() { return "média"; }
}

class Sala extends Quarto {
    String da_Dim() { return "grande"; }
}
```

O array de Objects passaria agora a ser um array de Quartos, e o ciclo para determinação das dimensões de cada divisão escrito simplesmente como:

```
String s = "";
for (int i = 0; i < quartos.length; ++i) {
  s = s +"Quarto "+ (i+1) + quartos[i].da_dim();
}
return s;
```

O método invocado da_dim será sempre o adequado à classe da instância de Quarto guardada em quartos[i], sem casting nem instanceOf, mas via "dynamic lookup".

```
// I/O básico, casting e introdução à utilização de Excepções;
//
// Criação de uma classe CONSOLA que lê da StandardInput e
// escreve na StandardOutput (pré-definidas cf. System.out e
// System.in
//
// I/O mais sofisticado será estudado nas Streams e Applets;


import java.lang.*;

public class Consola {

// métodos de classe

// escreve o Prompt

public static void escreveTxt(String txt) {
  System.out.print(txt + ":");
  System.out.flush();
}


// lê uma String, seja com texto de output ou não

public static String leStr() {
  int ch;
  String r = "";
  boolean fim = false;
  while (!fim) {
    try
       { ch = System.in.read();
         if (ch < 0 || (char) ch == '\n')
            fim = true;
         else
            r = r + (char) ch;
       }
       catch(java.io.IOException e) { fim = true;}
  }
  return r;
}


public static String leStr(String txt) {
   escreveTxt(txt);
   return leStr();
}
```

```java
public static int leInt(String txt) {
    while (true)
     { escreveTxt(txt);
       try
          { return Integer.valueOf(leStr().trim()).intValue();}
       catch(NumberFormatException e)
          { System.out.println("Nao e um numero inteiro !!"); }
     }
}


// le um Double

public static double leDouble(String txt) {
    while (true)
     { escreveTxt(txt);
       try
        { return Double.valueOf(leStr().trim()).doubleValue();}
       catch(NumberFormatException e)
            {System.out.println("Nao e um numero em VF !!"); }
     }
}


// le um Float

public static float leFloat(String txt) {
    while (true)
     { escreveTxt(txt);
       try
         { return Float.valueOf(leStr().trim()).floatValue(); }
       catch(NumberFormatException e)
         {System.out.println("Nao e um numero em VF !!"); }
     }
}


// le um booleano

public static boolean leBool(String txt) {
    while (true)
     { escreveTxt(txt);
       try
          { String s = leStr().trim().toLowerCase();
            if ( s.equals("true") || s.equals("false"))
              return Boolean.valueOf(s).booleanValue();
            else
                throw new IllegalArgumentException(""); }
       catch(IllegalArgumentException e)
           {System.out.println("Nao e um Booleano "); }
     }
}}
```

```
import java.lang.*;

public class Tconsola1 {

// teste

public static void main(String args[]) {
   int i = 0;
   double db = 0.0;
   float f ;
   String s = new String("");
   boolean b;

   s= Consola.leStr("Introduza o texto ");
   System.out.println("Texto = "+ s);

   i = Consola.leInt("Int ");
   System.out.println("Inteiro = "+ i);

   db = Consola.leDouble("Double ");
   System.out.println("Duplo = "+ db);

   f = Consola.leFloat("Float ");
   System.out.println("Float = "+ f);

   b = Consola.leBool("Booleano ");
   System.out.println("Booleano = "+ b);


  }
  }
```

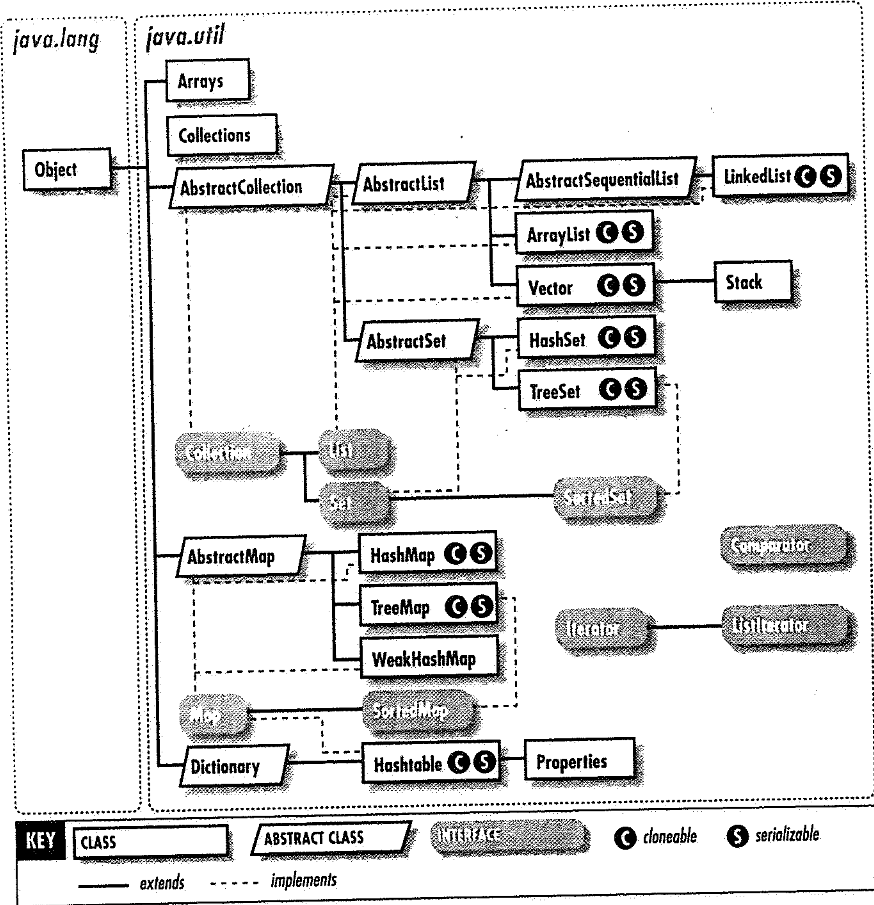


Figure 23-1: The collection classes of the java.util package

Note that if you subclass AbstractCollection directly, you are implementing a *bag*—an unordered collection that allows duplicate elements. If your add() method rejects duplicate elements, you should subclass AbstractSet instead. See also AbstractList.

```
public abstract class AbstractCollection implements Collection {
// Protected Constructors
   protected AbstractCollection();
// Methods Implementing Collection
   public boolean add(Object o);
   public boolean addAll(Collection c);
   public void clear();
   public boolean contains(Object o);
   public boolean containsAll(Collection c);
   public boolean isEmpty();
   public abstract Iterator iterator();
   public boolean remove(Object o);
   public boolean removeAll(Collection c);
   public boolean retainAll(Collection c);
   public abstract int size();
```

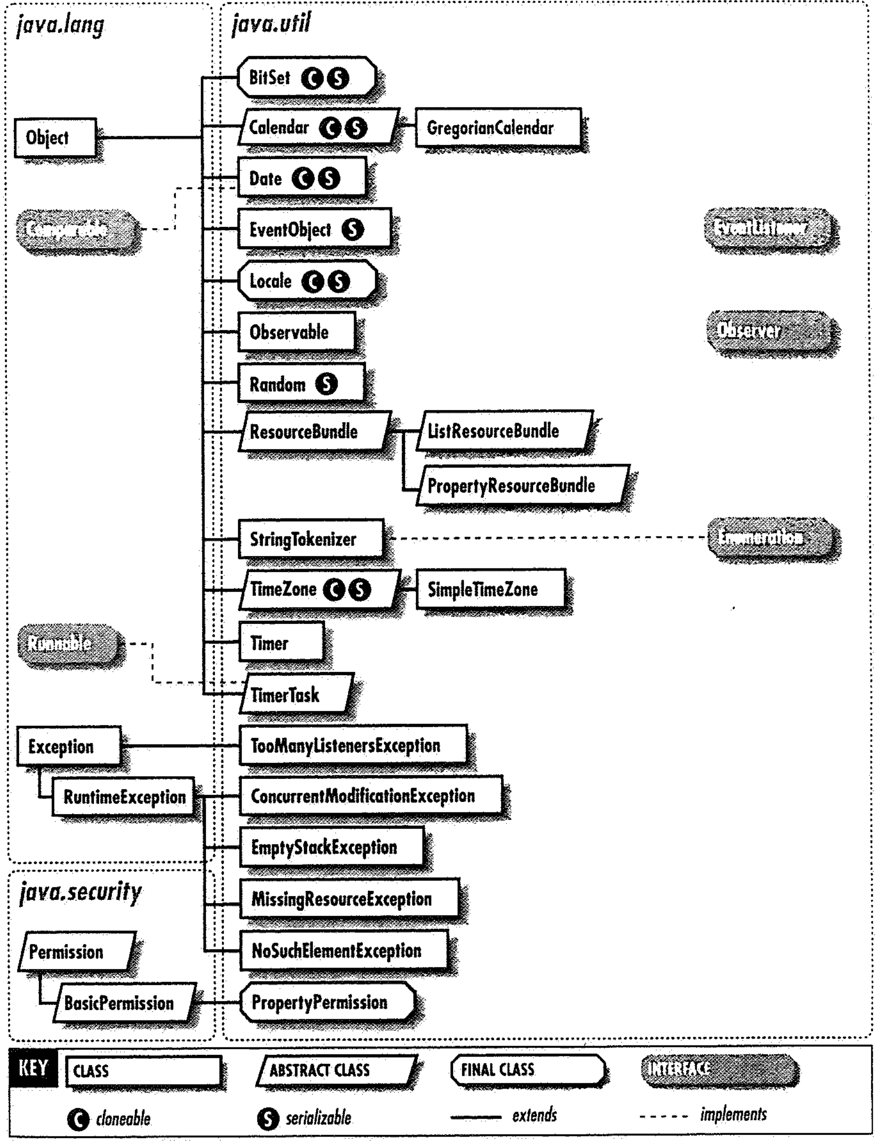



Figure 23-2: Other classes of the java.util package

```
   public Object[ ] toArray();
   public Object[ ] toArray(Object[ ] a);
// Public Methods Overriding Object
   public String toString();
}
```

EXCEPÇÕES EM JAVA: DECLARAÇÕES e TRATAMENTO.

Introdução

❒ A tão proclamada **robustez** do código dos programas Java está relacionada com 3 mecanismos importantes da linguagem:

- *o seu sistema forte de tipos;*
- *o seu modelo de memória sem apontadores e com um sistema de "garbage collection"*;
- *a captura e o tratamento de "excepções".*

❒ Embora tal não seja, só por si, suficiente para que se garantam programas completamente fiáveis, independentes do contexto e de grande qualidade, são inegavelmente mecanismos que, se bem compreendidos e explorados, muito podem ajudar na procura da obtenção desse tão importante tipo de programas.

Procura-se nesta secção introduzir e exemplificar, usando programas concretos, as diversas formas de utilização do *mecanismo de excepções* (**exceptions**) da linguagem Java, dos pontos de vista sintáctico, semântico e pragmático - ou seja, quanto ao seu correcto emprego -, visando a construção de programas não só de qualidade relativamente aos princípios gerais da PPO, mas também robustos, ou seja, com capacidade de se manterem em execução mesmo em situa-ções em que erros inesperados surgem em resultado da sua própria execução. Para que tal seja possível, é necessário que seja o próprio código do programa a ser capaz de capturar tais situa-ções de erro e realizar o seu adequado tratamento (correcção e/ou recuperação).

Para tal, temos que dominar o mecanismo de excepções de Java colocado ao nosso dispor, me-canismo capaz de permitir ao programador:

- *definir quais as excepções que devem ser detectadas* num dado contexto do programa;
- *definir quais as acções que pretende ver executadas* em tais situações;
- *definir as excepções "lançadas" pelo próprio programa em certas condições.*

O que é uma Excepção ?

/ Uma *Excepção* (**Exception**) é um sinal gerado pela máquina virtual de Java, portanto em tempo de execução do programa, que é comunicado ao programa indicando a ocorrência de um **erro recuperável**.
São excepções comuns em tempo de execução, tentativas de divisão por zero, indexação para além dos limites de um Array, tentativas de leitura para além do fim de um ficheiro, etc. Trata-se sempre de tentativas de violação das restrições semânticas da linguagem de programação.

⇒ Um *Erro* (**Error**) em Java corresponde a uma situação para a qual nenhuma recuperação é já possível[1], limitando-se o interpretador de Java a enviar uma mensagem de erro e terminar a execução do programa.

❐ Quer os erros sejam ou não recuperáveis, a maioria das linguagens de programação tem problemas em realizar de forma elegante e eficiente a detecção e o tratamento de erros. É comum depararmos com porções de código do tipo,

```
int resultado = invocaRotinaQualquer( );

if ( resultado == VALOR_DE_ERRO ) {
    switch(var_global_de_erro) {
      case 1 : ................ ; break;
      case 2 : ................ ; break;
        ..........
    };
else {
            // o código normal
            // se não surgir o erro
      }
```

que, para além da sua fraca legibilidade, mostram a dificuldade em lidar com situações detalhadas de erro. Por vezes, os valores de erro podem ser confundidos com valores normais de retorno (cf. `NULL` em apontadores, `-1` em funções que retornam inteiros, etc.). Pior ainda será quando múltiplos erros puderem ser gerados pela mesma função.

O mecanismo de excepções de Java é, como veremos, inovador e claro, em muito ajudando à clareza dos programas, mesmo em tais circunstâncias excepcionais do fluxo de execução, apesar da inerente complexidade sempre introduzida.

⇒ Uma excepção é sinalizada ou "lançada" (*thrown*) a partir do ponto do código Java em que ocorreu, e diz-se "apanhada" (*caught*) no ponto do código para o qual o controlo do programa foi transferido.

⇒ As excepções podem ser *implícitas* (ou *assíncronas*), ou seja, automaticamente lançadas pela máquina virtual em função do erro detectado, ou até *explícitas* (ou *programadas*) quando é o próprio programador a codificar (declarar) o seu lançamento através de uma instrução própria (*throws*).
Em qualquer dos casos, Excepções e Erros de Java, são duas subclasses distintas da classe Java `java.lang.Throwable`, subclasses que estão no topo das respectivas hierarquias das classes de Erro e das classes de Excepção. Assim, *todas as excepções são instâncias da respectiva classe* de Excepção.

[1] Exceptua-se o erro *ThreadDeath* que determina o fim da execução da *thread* respectiva, mas que não implica o envio de mensagem nem altera a execução de outras possíveis *threads*.

Por exemplo, uma excepção devida a uma tentativa de acesso a um índice inexistente de um Array, será uma instância da classe `ArrayIndexOutOfBoundsException`, que tem o seguinte posicionamento na hierarquia:

```
java.lang.Object
   < Throwable < Exception < RunTimeException <
   < IndexOutOfBoundsException < ArrayIndexOutOfBoundsException
```

Veremos posteriormente qual a hierarquia completa das Excepções, que nos permitirá ter uma ideia de todas as classes de Excepções existentes em Java, e ainda como se pode manipular a informação contida em cada instância-excepção gerada.

Vamos agora estudar em detalhe o mecanismo de captura e tratamento de excepções, de momento centrando a atenção nas excepções *implícitas*, ou seja, as excepções lançadas pelo interpretador.

Tratamento de Excepções: cláusulas try - catch.

⇨ A linguagem Java permite a descrição de situações de excepção de uma forma normalizada atraves da utilização de 5 palavras chave correspondentes a cláusulas especiais, a saber: `try`, `catch`, `throw`, `throws` e `finally`.

⇨ A estrutura genérica para a captura e tratamento das excepções, implícitas ou explícitas, tem a seguinte forma:

```
try {
        // O código do programa tal como seria escrito mesmo que garantidamente não pudesse
        // gerar qualquer erro, é colocado neste bloco. Porém, ao colocá-lo num bloco try
        // passaremos a ter a possibilidade de detectar a ocorrência de alguns possíveis erros.
    }
catch(Identificador_da_excepção var_exc1)
    {
        // É aqui escrito o código de tratamento da excepção identificada na cláusula catch,
        // sendo var_exc1 a instância da excepção que foi gerada (e que pode ser usada).
    }
catch(Identificador_da_excepção var_exc2)
    {
        // Podemos ter inúmeras cláusulas catch para o mesmo bloco try, cada uma correspondendo
        // a uma classe de excepção diferente.
    }
finally
    {
        // O código aqui colocado será sempre executado caso surja ou não uma excepção em try.
        // Este código pode ser muito importante, por exemplo, para fazer o fecho de ficheiros,
        // libertar recursos alocados, ou manipular variáveis.
        // Se foi detectada excepção em try e existe uma cláusula catch local ao método, então o
        // bloco catch é executado, e só depois o bloco finally.
        // Se não existe catch local para a excepção ocorrida, é executado o bloco finally e, caso exista,
        // a cláusula catch externa ao método que primeiro for encontrada (por exemplo no
        // método invocador).
        // Se a forma de saída do bloco try, for return, continue ou break, quer exista ou não catch, o
        // bloco finally é de imediato executado.
    }
```

❐ A cláusula catch representa o que normalmente se designa por um *"exception handler"*, ou seja, um bloco de tratamento da excepção lançada. A ocorrência de excepções pela execução do código de um dado método (correspondente a uma mensagem enviada a uma instância ou classe), só é possível se esse método for capaz de as lançar (via instrução *throw*), e não fizer localmente o `catch`, isto é, deixar que a excepção seja detectada e tratada externamente, ou seja, no método invocador ou noutro ainda mais externo.
Vejamos agora alguns exemplos concretos que nos irão auxiliar a explicar em detalhe a semântica operacional deste mecanismo de tratamento de excepções.

Exemplo E.1: Leitura de uma sequência de Palavras e armazenamento de cada uma na posição de um Array cujo índice é introduzido pelo utilizador (sem validação prévia).

```
import java.lang.*;

public class TstExcEx1 {
//
   private static String leStr() {
   int ch ;          // código ASCII do caracter lido
   String r = "" ;   // representação do inteiro
   boolean fim = false;

     while(!fim) {
       try
        { ch = System.in.read();    // leitura de um caracter
          if (ch < 0 || (char) ch == '\n')   // EOF ? CR-LF ?
             fim = true ;
          else
             r = r + (char) ch ;     // concatenação
        }
       catch(java.io.IOException e)   // este catch é obrigatório
          { fim = true ; }
     }
     return r ;
   }

  public static void main(String args[]) {
  //
   boolean fim = false ;
   int index;
   String palavra = "" ;
   String[] tabPal = new String[10];    // Array de Palavras

         System.out.print("Introduza uma Palavra :");
         System.out.flush();
         palavra = leStr();
         while (!palavra. equals("")) {      // "" => fim do ciclo
             System.out.print("Introduza um Indice :");
             System.out.flush();
             index = Integer.valueOf(leStr().trim()).intValue();
             tabPal[index]= palavra;
         // nova leitura de uma palavra
             System.out.print("Introduza uma Palavra :");
             System.out.flush();
             palavra = leStr();
         }
  }
}
```

Análise :

O método `main` implementa um ciclo para leitura de Palavras e para o seu armazenamento nas posições do Array de Strings, posições essas igualmente indicadas pelo utilizador. O ciclo termina com a leitura de uma String vazia.

O método para leitura de Strings caracter a caracter, o método privado `private leStr()`, possui um par `try-catch` que não seria obrigatório. Porém, ao colocar uma instrução `System.in.read()` num bloco `try`, somos imediatamente obrigados (pelo compilador) a fornecer a respectiva cláusula `catch`, dado que o método `read()` pode lançar uma excepção de I/O, tal como se mostra de seguida ao apresentar o código fonte de tal método.

```
/*
 * @(#)InputStream.java 1.20 97/01/25
 * CopyrightVersion 1.1_beta
    package java.io;
    public abstract class InputStream {
    /**
    * Reads the next byte of data from this input stream. This method blocks until input data
    * is available, the end of the stream is detected, or an exception is thrown.
    * <p>
    * A subclass must provide an implementation of this method.
    *
     * @exception  IOException  if an I/O error occurs.
    */
    public abstract int read() throws IOException;
```

Assim, o código apresentado é robusto relativamente à ocorrência de erros de I/O resultantes da leitura da `InputStream`, tendo, em particular, sido programada uma cláusula `catch` muito simples que apenas termina o ciclo de leitura de caracteres e permite ao método retornar a "string" `r` até esse momento construída.

No entanto, tal robustez do código não se estende às outras instruções do método `main`. Este método lê, dentro de um ciclo `while`, as palavras introduzidas e os índices do Array onde as mesmas devem ser inseridas e, sem qualquer teste particular, procura inseri-las em tais posições do Array.
Naturalmente que não sendo validado o valor do índice do Array no qual a palavra lida deve ser inserida, poderá acontecer que o utilizador introduza um valor de índice errado e, na tentativa de inserção no Array, um erro de indexação possa surgir. No entanto, o código apresentado não apresenta qualquer par `try-catch` de segurança relativamente a esta condição de excepção potencial. Ou seja, *o código apresentado não é, neste caso robusto*.

A prova desta falta de robustez potencial é exemplificada na seguinte sessão de trabalho, na qual o utilizador tenta atribuir à posição 14 do Array de Palavras a Palavra anteriormente lida, gerando a consequente finalização por erro do programa e a apresentação da respectiva "stack" de execução para "debbuging":

```
c:\jdk1.1> java TstExcEx1
Introduza uma Palavra : abc
Introduza um Indice : 1
Introduza uma Palavra : xyz
Introduza um Indice : 14
java.lang.ArrayIndexOutOfBoundsException: 14
   at TstExcEx1.main(TstExcEx1.java:39)
```

Vamos então agora alterar o código introduzindo uma cláusula `try` que procura capturar tal situação de excepção potencial. Teríamos a seguinte alteração no código original :

```
System.out.print("Introduza uma Palavra :");
System.out.flush();
palavra = leStr();
while (!palavra. equals("")) {      // "" => fim do ciclo
    System.out.print("Introduza um Indice :");
    System.out.flush();
    index = Integer.valueOf(leStr().trim()).intValue();
    try
       { tabPal[index]= palavra; }
// nova leitura de uma palavra
    System.out.print("Introduza uma Palavra :");
    System.out.flush();
    palavra = leStr();
}
```

Esta alteração produziria no entanto um erro de compilação dado que introduzimos uma cláusula `try` sem a correspondente cláusula `catch`, e nos encontrarmos no contexto de um método `main()`, ou seja, sem hipóteses de que tal excepção possa ser capturada por um método invocador daquele em que a excepção foi produzida, dado este ser o mais externo possível.

Assim, a alteração final correcta a introduzir poderia ser,

```
System.out.print("Introduza uma Palavra :");
System.out.flush();
palavra = leStr();
while (!palavra. equals("")) {      // "" => fim do ciclo
    System.out.print("Introduza um Indice :");
    System.out.flush();
    index = Integer.valueOf(leStr().trim()).intValue();
    try
       { tabPal[index]= palavra; }
    catch(ArrayIndexOutOfBoundsException e)
       { System.out.printl(e.getMessage()); }
// nova leitura de uma palavra
    System.out.print("Introduza uma Palavra :");
    System.out.flush();
    palavra = leStr();
}
```

A sessão de trabalho passaria a produzir os seguintes resultados :

```
c:\jdk1.1> java TstExcEx2
Introduza uma Palavra : abc
Introduza um Indice : 1
Introduza uma Palavra : xyz
Introduza um Indice : 14
14
Introduza uma Palavra :

c:\jdk1.1>
```

O resultado do "catch" da excepção, tal como programado na cláusula, faz apenas o "output" do resultado de enviarmos à variável de instância a mensagem `getMessage()`. Tal mensagem é, neste caso, muito pobre, limitando-se a produzir o valor do índice de erro.
Vamos por isso substituí-la por uma mensagem própria màis elucidativa para o utilizador, cf.,

```
try
   { tabPal[index]= palavra; }
catch(ArrayIndexOutOfBoundsException e)
   { System.out.printl("Indice fora dos limites !"); }
```

sendo agora produzidas as seguintes saídas,

```
c:\jdk1.1> java TstExcEx2
Introduza uma Palavra : abc
Introduza um Indice : 1
Introduza uma Palavra : xyz
Introduza um Indice : 14
Indice fora dos limites !
Introduza uma Palavra :

c:\jdk1.1>
```

Aparentemente temos já um programa robusto. Porém tal não é, de facto, verdade, conforme se poderá deduzir da seguinte sessão:

```
c:\jdk1.1> java TstExcEx2
Introduza uma Palavra : abc
Introduza um Indice : 1
Introduza uma Palavra : xyz
Introduza um Indice : 2a
java.lang.NumberFormatException: 1a
        at java.lang.Integer.parseInt(Integer.java: 147)
        at java.lang. Integer. valueOf(Integer.java: 193)
        at TstExcEx2.main(TstExcEx2.java: 36)

c:\jdk1.1>
```

Como se pode verificar pela "stack de execução" apresentada pelo interpretador de Java, a excepção ocorrida tem a ver com o formato errado do número introduzido (`2a` não é um inteiro válido), tendo sido lançada pelo método `parseInt()` (linha 147), que foi invocado pelo método `valueOf()` (linha 193) que havia sido invocado na linha 36 do método `main()` da classe `TstExcEx2`.

Assim, para que o programa seja de facto robusto teremos que introduzir uma outra cláusula de `catch` para este tipo particular de excepção e, ainda, colocar no bloco `try` a declaração na qual a excepção pode surgir .

Passaremos então a ter o seguinte corpo no método em questão,

```
System.out.print("Introduza uma Palavra :");
System.out.flush();
palavra = leStr();
while (!palavra. equals("")) {        // "" => fim do ciclo
    System.out.print("Introduza um Indice :");
    System.out.flush();
    try
      { index = Integer.valueOf(leStr().trim()).intValue();
        tabPal[index]= palavra; }
    catch(ArrayIndexOutOfBoundsException e)
      { System.out.printl("Indice fora dos limites !"); }
    catch(NumberFormatException e)
      { System.out.printl("Inteiro invalido !"); }
 // nova leitura de uma palavra
    System.out.print("Introduza uma Palavra :");
    System.out.flush();
    palavra = leStr();
 }
```

e a seguinte sessão interactiva, como exemplo,

```
c:\jdk1.1> java TstExcEx3
Introduza uma Palavra : abc
Introduza um Indice : 1a
Inteiro Invalido !
Introduza uma Palavra : abcd
Introduza um Indice : 14
Indice fora dos limites !
Introduza uma Palavra : abcd
Introduza um Indice : 1
Introduza uma Palavra : xyz
Introduza um Indice : 10
Indice fora dos limites !
Introduza uma Palavra : aaa
Introduza um Indice : 9
Introduza uma Palavra :

c:\jdk1.1>
```

O programa apresentado é neste momento robusto, ou seja, é capaz de realizar o tratamento lógico e manter a sequência lógica de execução mesmo que surjam as condições de excepção detectadas, não sendo de prever outras em função do código apresentado.

Torna-se no entanto importante determinar agora, em função das experiências anteriores, se uma excepção detectada num método que não o método `main()`, mas por este invocado, e não localmente tratada, gera um erro do compilador. Para tal vamos considerar que a leitura da posição do Array onde a palavra deve ser inserida é realizada num método auxiliar, no qual é declarado o `try` mas não definido o `catch`, conforme o exemplo :

```
private static int leIndex() {
//
int index;

try
   { index = Integer.valueOf(leStr().trim()).intValue(); }
return index;
}

public static void main(String args[]) {
//
boolean fim = false ;
int ind;
String palavra = "" ;
String[] tabPal = new String[10];   // Array de Palavras


System.out.print("Introduza uma Palavra :");
System.out.flush();
palavra = leStr();
while (!palavra. equals("")) {     // "" => fim do ciclo
      System.out.print("Introduza um Indice :");
      System.out.flush();
      try
         { ind = leIndex();
           tabPal[ind]= palavra; }
      catch(ArrayIndexOutOfBoundsException e)
         { System.out.printl("Indice fora dos limites !"); }
      catch(NumberFormatException e)
            { System.out.printl("Inteiro invalido !"); }
      // nova leitura de uma palavra
      System.out.print("Introduza uma Palavra :");
      System.out.flush();
      palavra = leStr() ;
   }
}
```

A compilação deste código produz de imediato a resposta desejada:

```
c:\jdk1.1> javac TstExcEx4.java
   compiling: TstExcEx4.java
TstExcEx4: java(28): 'try' without 'catch' or 'finally'.
         return index
         ^
1 error
c:\jdk1.1>
```

⇒ Isto é, mesmo em métodos mais interiores que `main()`, para cada cláusula `try` o compilador de Java obriga à introdução da respectiva cláusula `catch`.

⇒ Se, porventura, a cláusula catch local não contempla a excepção efectivamente lançada, então é procurada uma cláusula catch *mais exterior* correspondente a tal excepção. Caso esta não seja encontrada, então o programa termina a sua execução e a "stack de execução" é apresentada (cf. exemplo seguinte).

```
private static int leIndex() {
//
int index;

try
   { index = Integer.valueOf(leStr().trim()).intValue();
     return index;
   }
catch(StringIndexOutOfBoundsException e)
   { System.out.println("Erro na String de entrada !"); }
}

public static void main(String args[]) {
//
boolean fim - false ;
int ind;
String palavra = "" ;
String[] tabPal = new String[10];    // Array de Palavras

System.out.print("Introduza uma Palavra :");
System.out.flush();
palavra = leStr();
while (!palavra. equals("")) {      // "" => fim do ciclo
       System.out.print("Introduza um Indice :");
       System.out.flush();
       try
          { ind = leIndex();
            tabPal[ind]= palavra; }
       catch(ArrayIndexOutOfBoundsException e)
          { System.out.printl("Indice fora dos limites !"); }
       // nova leitura de uma palavra
       System.out.print("Introduza uma Palavra :");
       System.out.flush();
       palavra = leStr() ;
   }
}
```

conforme a sessão exemplo,

```
c:\jdk1.1> javac TstExcEx5.java
    compiling: TstExcEx5.java
c:\jdk1.1> java TstExcEx5
Introduza uma Palavra : aabb
Introduza um Indice : 1a
java.lang.NumberFormatException: 1a
        at java.lang.Integer.parseInt(Integer.java: 147)
        at java.lang. Integer. valueOf(Integer.java: 193)
        at TstExcEx5.leIndex(TstExcEx5.java: 28)
        at TstExcEx5.main(TstExcEx5.java: 49)
```

Porém, caso o código tivesse sido escrito conforme se apresenta a seguir, a excepção gerada no método `leIndex()`, ainda que não capturada e tratada localmente, sê-lo-ia no método `main()`, evitando assim a terminação da execução do programa.

```
private static int leIndex() {
//
int index;

try
   { index = Integer.valueOf(leStr().trim()).intValue();
     return index;
   }
catch(StringIndexOutOfBoundsException e)
   { System.out.println("Erro na String de entrada !"); }
}

public static void main(String args[]) {
//
boolean fim = false ;
int ind;
String palavra = "" ;
String[] tabPal = new String[10];    // Array de Palavras


System.out.print("Introduza uma Palavra :");
System.out.flush();
palavra = leStr();
while (!palavra. equals("")) {       // "" => fim do ciclo
       System.out.print("Introduza um Indice :");
       System.out.flush();
       try
          { ind = leIndex();
            tabPal[ind]= palavra; }
       catch(ArrayIndexOutOfBoundsException e)
          { System.out.printl("Indice fora dos limites !"); }
       catch(NumberFormatException e)
             { System.out.printl("Inteiro Invalido !"); }
       // nova leitura de uma palavra
       System.out.print("Introduza uma Palavra :");
       System.out.flush();
       palavra = leStr() ;
   }
}
```

Um exemplo de sessão interactiva com este programa seria:

```
c:\jdk1.1> java TstExcEx6
Introduza uma Palavra : abcd
Introduza um Indice : 2b
Inteiro Invalido !
Introduza uma Palavra : xyz
Introduza um Indice : 19
Indice fora dos limites !
Introduza uma Palavra : abcd
Introduza um Indice : 9
Introduza uma Palavra :
c:\jdk1.1>
```

⇒ Assim, quando **cláusulas** `try` **são declaradas seguindo uma estrutura aninhada**, ou seja, quando uma declaração `try` é colocada à volta da execução de um dado método, e o próprio método possui declarações `try` relativas a dado código, a regra de procura das cláusulas `catch` é muito simples e intuitiva: a cláusula `catch` é procurada do interior do método que originou a excepção para o exterior, sendo o limite para o tratamento da excepção o código do método `main()`.

⇒ Metodologicamente parece preferível que as excepções resultantes da execução do código de um dado método sejam capturadas e tratadas localmente. A esta regra podem no entanto sobrepor-se razões lógicas particulares de escrita de código genérico em certas aplicações.

Lançamento Explícito de Excepções: `throw` e `throws`.

⇒ Torna-se por vezes necessário no código de um dado método, "lançar" explicitamente uma excepção, ou seja, alertar explicitamente para uma situação de erro entretanto ocorrida na execução de tal código. Qualquer método tem a possibilidade de o fazer usando a cláusula `throw` e criando uma instância de uma dada excepção usando `new`.

⇒ A linguagem Java requer que qualquer método que possa provocar a ocorrência de uma excepção normal, ou seja, que possua uma declaração `throw`, ou faça localmente o tratamento de tal excepção numa cláusula `catch`, ou declare explicitamente que pode lançar tal excepção embora não a trate localmente. Neste último caso, no cabeçalho do método devem ser explicitamente declaradas as excepções lançadas através de uma cláusula `throws`.
No exemplo seguinte mostra-se o código de um método `pop()` da classe `StackLim` que pode gerar uma situação de excepção mas que não a trata localmente.

```
void pop() throws EmptyStackException {
      if (this.empty())
            throw new EmptyStackException();
      else
            numElem -= 1;
}
```

⇒ Dado que este método não faz o tratamento local da excepção por ele próprio gerada, deixando tal tratamento para o código mais exterior, então é obrigatória a declaração `throws` que é apresentada no seu cabeçalho.

⇒ Se mais do que uma excepção não localmente tratada puder ser lançada por um método, então todas devem ser declaradas, separadas por uma vírgula, na declaração `throws`, cf.

```
void push() throws FullStackException,
                   IllegalArgumentError,
                   ArrayIndexOutOfBoundsException {
```

objectos inseridos são de classes distintas, podendo ser instâncias de qualquer classe dado que StackLim foi definida como contendo elementos da classe `Object`.

Testa-se de seguida o método de visualização da "stack" como String, testa-se o método que consulta o elemento no topo da "stack", e de seguida, invoca-se 5 vezes o método `pop()`. Por esta altura a "stack" deverá estar já vazia, o que é testado a seguir. De seguida é enviada a mensagem correspondente à execução do método de consulta do elemento que se encontra no topo da "stack".

Vejamos o resultado da execução deste programa de teste:

```
c:\jdk1.1> java TstStackLim
Dimensao Maxima da Stack1 : 15
Dimensao Max. da Stack1 = 15
true
true
A Maria tem 20 anos
A
true
java.util.EmptyStackException
        at StackLim.top(StackLim.java:57)
        at TstStackLim.main(TstStackLim.java:25)

c:\jdk1.1>
```

Este resultado não nos pode surpreender porque, como podemos verificar pelas mensagens da "stack de execução", o erro surgiu na execução do método `top()` que gera a excepção registada sempre que a "stack" se encontra vazia, tal como declarado no seu cabeçalho.
Tal erro resulta de não termos codificado o tratamento desta excepção no código exterior, neste caso o método `main()`.

Esta excepção e todas as outras que podem ocorrer durante a manipulação das instâncias da classe `StackLim`, devem portanto ser tratadas, neste exemplo, no método `main()`. Vamos então introduzir estas alterações, e observar a robustez do programa exemplo após a introdução de tais modificações.

```
dimensao = Consola.leInt("Dimensao Maxima da Stack1 ");
StackLim stack1 = new StackLim(dimensao);
try
  { stack1.push("anos"); stack1.push(new Integer(20));
    stack1.push("tem"); stack1.push("Maria");stack1.push("A");
    System.out.println(stack1.output());
    System.out.println(stack1.top());
    stack1.pop();stack1.pop();stack1.pop();stack1.pop();
    stack1.pop();
    System.out.println(stack1.empty());
    stack1.top();
  }
catch(EmptyStackException e)
  { System.out.println("STACK VAZIA !!"); }
```

Teríamos agora a seguinte sessão interactiva:

```
c:\jdk1.1> java TstStackLim
Dimensao Maxima da Stack1 : 15
Dimensao Max. da Stack1 = 15
true
true
A Maria tem 20 anos
A
true
STACK VAZIA !!

c:\jdk1.1>
```

A excepção `EmptyStackException` está já tratada no mètodo `main()`, ainda que de forma simples. Porém, e conforme as cláusulas `throws` declaradas nos diversos métodos da classse `StackLim`, a excepção pré-definida de Java `IndexOutOfBoundsException`, usada para detectar uma tentativa de acesso a uma posição inexistente da "stack", em função das suas dimensões, não foi ainda capturada e tratada no código de `main()`. Assim, um erro potencial é ainda negligenciado no código de `main()`.

Em resultado de tal negligência, o seguinte código de `main()`,

```
dimensao = Consola.leInt("Dimensao Maxima da Stack1 ");
StackLim stack1 = new StackLim(dimensao);
try
  { stack1.push("anos"); stack1.push(new Integer(20));
    stack1.push("tem"); stack1.push("Maria");stack1.push("A");
    System.out.println(stack1.output());
    System.out.println(stack1.top());
    stack1.pop();stack1.pop();stack1.pop();stack1.pop();
    stack1.pop();
    System.out.println(stack1.empty());
    stack1.top();
  }
 catch(EmptyStackException e)
  { System.out.println("STACK VAZIA !!"); }
```

em função dos valores introduzidos pelo utilizador, poderia produzir a seguinte sessão interactiva:

```
c:\jdk1.1> java TstStackLim1
Dimensao Maxima da Stack1 : 3
Dimensao Max. da Stack1 = 3
true
true
java.lang.IndexOutOfBoundsException: Stack Cheia !!
        at StackLim.push(StackLim.java:41)
        at TstStackLim1.main(TstStackLim1.java:19)

c:\jdk1.1>
```

Há, portanto, a necessidade de no código de `main()`, dado ser o código imediatamente exterior à ocorrência das excepções declaradas, codificar via `catch` as regras de tratamento das excepções lançadas pelos métodos de `StackLim`.

O código final de `main()`, 100% robusto, para a manipulação da classe `StackLim` ficaria pois da forma seguinte:

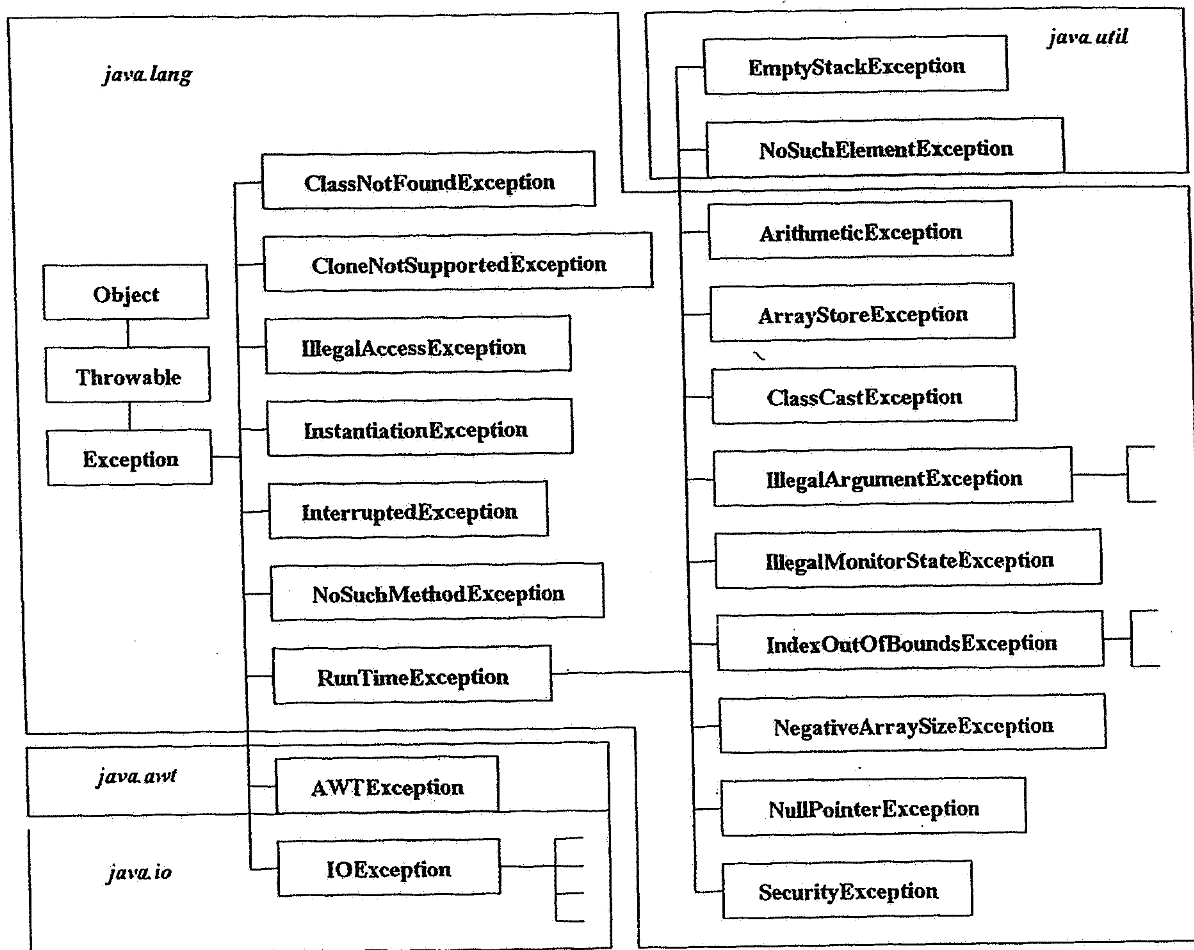
```
import java.util.*;

public class TstStackLim3 {
//
public static void main(String args[]) {
//
int dimensao;
dimensao = Consola.leInt("Dimensao Maxima da Stack1 ");
StackLim stack1 = new StackLim(dimensao);
System.out.println("Dimensao Max. da Stack1 = " + stack1.dim());
StackLim stack2 = new StackLim();
//
System.out.println(stack1.empty());
System.out.println(stack2.empty());
try
  { stack1.push("anos");
    stack1.push(new Integer(20));
    stack1.push("tem");
    stack1.push("Maria");
    stack1.push("A");
    System.out.println(stack1.output());
    System.out.println(stack1.top());
    stack1.pop(); stack1.pop(); stack1.pop(); stack1.pop();
    stack1.pop();
    System.out.println(stack1.empty());
    stack1.top();
  }
 catch(EmptyStackException e)
   { System.out.println("STACK VAZIA !!"); }
 catch(IndexOutOfBoundsException e)
   { System.out.println("STACK CHEIA !!"); }
 }
}
```

tendo como sessão interactiva exemplo a seguinte:

```
c:\jdk1.1> java TstStackLim1
Dimensao Maxima da Stack1 : 3
Dimensao Max. da Stack1 = 3
true
true
STACK CHEIA !!

c:\jdk1.1>
```

Classes de Excepção em Java

➯ A hierarquia de classes de Excepção de Java é a seguinte:



Note-se o posicionamento das classes de Excepção relativamente aos "packages" a que pertencem.

Criação de novas Excepções

❒ Sendo excepções instâncias das classes de Excepção existentes em Java, a criação de novos tipos de excepções consiste na simples criação de classes particulares de excepção definidas pelo utilizador, e no seu posicionamento na hierarquia anteriormente apresentada.

➯ O código Java típico para a criação de um novo tipo de excepção é simplesmente, dado herdarmos a implementação definida em Exception:

```
class NovaExcepcao extends Exception {
    NovaExcepcao() { super(); }
    NovaExcepcao(String s) { super(s); }
}
```

INTERFACES JAVA

Subclasses versus Subtipos.

❐ Numa linguagem de PPO as classes definem como sabemos a estrutura e o comportamento comum das suas instâncias. Por outro lado, as classes fazem parte de uma hierarquia particular que as relaciona, e cujo mecanismo de herança faz com que seja uma hierarquia de especialização. Assim, uma classe que declara que `extends` uma outra torna-se *subclasse* da primeira, e a primeira é, em Java, a única possível *superclasse* da segunda, já que a herança é, como vimos já, de tipo simples. A subclasse é uma possível *especialização* da primeira pois herda toda a estrutura e comportamento nesta definidos podendo ainda acrescentar-lhe mais estrutura e comportamento. No sentido descendente da hierarquia temos classes cada vez mais específicas.

❐ O conceito de *subclasse* (associado ao mecanismo de *herança*) está assim muito ligado a uma filosofia muito interessante de construção de componentes de software que se baseia na procura de *reutilização* de componentes já existentes, e que fomenta um estilo de programação que, como já vimos atrás, é *programação incremental*.
Numa linguagem pura de PPO - onde tudo são objectos, como por exemplo em Smalltalk - o *tipo* de uma dada instância associa-se de forma natural à classe que a definiu e criou, sendo as subclasses associadas a *subtipos*. No entanto esta associação é abusiva conforme veremos.

❐ A noção de *subtipo* tem, em geral, uma conotação mais abstracta que a de *subclasse*. Um *subtipo* é definido sempre em função apenas do comportamento e não da estrutura. Assim, **B** pode ser considerado um subtipo de **A**, se qualquer entidade do tipo **A** puder ser *substituída* por uma do tipo **B** sem que se observe qualquer alteração no comportamento (princípio designado por *princípio da substitutividade*).

❐ Em termos abstractos, portanto, *subclasses* e *subtipos* são conceitos que não têm necessariamente que estar relacionados entre si.

❐ De facto, em linguagens puras e com *"dynamic binding & checking"* (associação e verificação de correcção em tempo de execução), como o Smalltalk, tais conceitos não estão directamente relacionados. Por exemplo, duas classes podem ter comportamento exactamente igual (ou seja as suas instâncias respondem exactamente ao mesmo conjunto de mensagens), mas não terem a mesma implementação nem sequer uma superclasse comum (para além da classe raiz). Poderíamos portanto afirmar que as suas instâncias são *substituíveis*, ou seja, cada uma delas poderia funcionar como *subtipo* da outra, embora nem sequer exista entre as respectivas classes uma relação de *subclasse*.

❐ Porém, as linguagens "strongly typed" (fortemente tipadas), ou seja, com um rigoroso mecanismo de verificação de tipos em tempo de compilação, tais como o Object-Pascal e o C++, tendem a diminuir a distinção entre *subclasses* e *subtipos* de duas formas:

- *permitem que variáveis detenham valores de tipo diferente se o tipo dinâmico do valor for subclasse do tipo estático da variável;*

- *assumem que subclasses são subtipos, o que pode não ser verdade já que as subclasses podem reescrever (isto é, redefinir) métodos.*

⇒ De todas as actuais linguagens de PPO, Java é a linguagem que melhor procura clarificar os conceitos de *subclasse* e de *subtipo*, para tal adoptando todos os usuais mecanismos de herança e subclassificação, mas introduzindo um novo mecanismo visando a *especificação* e *implementação* de Tipos Abstractos de Dados (TADs), ou seja, permitindo a introdução da noção de *tipos* e *subtipos* definidos pelo utilizador: as *Interfaces*.

⇒ Adicionalmente, como veremos, o mecanismo de *interfaces* de Java permitirá resolver de forma elegante e eficiente, o que noutras linguagens de PPO é solucionado recorrendo ao mecanismo de herança múltipla, ou seja, a necessidade que existe por vezes de que uma classe exiba um comportamento que é a soma dos comportamentos de mais do que uma outra classe existente (ainda que, mesmo com herança múltipla, estas devam ser suas superclasses).

⇒ Finalmente, como veremos também, o mecanismo de *interfaces* de Java permitirá igualmente definir mecanismos simples de *comunicação* e *sincronização* de estados entre diferentes classes, em particular entre classes computacionais e classes de interface com o utilizador, dado que as últimas são extraordinariamente dependentes do estado das aplicações, representados naturalmente nos estados internos das primeiras.
Por exemplo, as simples opções que devem ser disponibilizadas num menu de operações, são dependentes do estado interno (ou seja do contexto) actual da aplicação. Como se sabe, uma operação só deverá estar acessível se, num dado contexto, a sua pré-condição for calculada como sendo satisfeita.

❒ Depois de se apresentar a forma correcta da sua declaração, estudaremos o mecanismo de *interfaces* de Java seguindo exactamente a sequência dos três problemas que o mesmo visa melhorar relativamente a outras linguagens de PPO, a saber:

- *garantia de implementação de comportamento herdado de várias fontes;*
- *especificação de novos tipos de dados e flexibilidade na sua implementação;*
- *implementação de mecanismos de comunicação e sincronização entre classes.*

Interfaces Java: Declaração.

⇒ A declaração de uma *interface* Java é muito simples dado que uma interface poderá apenas conter um *identificador*, um *opcional conjunto de constantes* (ou seja identificadores declarados como `static` e `final`, e um *conjunto de assinaturas (declarações) de métodos que são implicitamente abstractos* (sendo pois opcional usar o qualificador `abstract`).

⇒ Comparadas com as *classes abstractas* as *interfaces* Java são ainda mais restritivas. Numa classe abstracta é possível introduzir métodos concretos. *Numa interface todos os métodos são abstractos por definição*. Numa classe abstracta é possível introduzir variáveis de instância que todas as subclasses herdam e podem redefinir. *Numa interface os únicos identificadores de valores são constantes* e `final` (embora possam ser redefinidas).
Assim, só obedecendo a estas restrições uma classe 100% abstracta poderia representar uma *interface*. Porém, dadas as outras propriedades das *interfaces*, nunca o conseguiria fazer.

⇒ Consideremos alguns exemplos de declarações de *interfaces*:

```
public interface Enumeravel {
  public abstract boolean vazia();
  public abstract Object seguinte();
}

public interface Colorivel {
  public abstract void defineCor(int cor);
}

public interface Ordem {
  public abstract boolean igual(Ordem elem);
  public abstract boolean maior(Ordem elem);
  public abstract boolean menor(Ordem elem);
}

public interface Desenhavel extends Colorivel {
  public abstract void posicao(double x , double y);
  public abstract void desenha();
}

public interface Amovivel { // que pode mover-se !!
  public void movimento(double x, double y);
}

public interface ComMotor extends Amovivel {
  public static final int limiteVel = 120;
  public abstract String motor();
}

public interface Transformavel extends Escalavel,
                                       Rodavel,
                                       Desenhavel {

     ..........................
}
```

❒ Estas declarações de interfaces são, como se pode observar, perfeitamente abstractas, ou seja, não impõem qualquer tipo de restrição sobre as implementações. No entanto, como se sabe, apenas classes poderão fornecer implementações para estas *especificações*. Assim, o mecanismo de *interfaces* permite *especificar* um conjunto de operações que, a partir de agora, uma dada classe pode declarar *implementar* em obediência à *interface declarada*, e independentemente do seu posicionamento na hierarquia de classes. Deste modo, *uma classe pode implementar um tipo de dados especificado numa dada interface* independentemente de ser também, em simultâneo, *subclasse* de uma classe qualquer.

⇒ Como se pode agora melhor compreender, numa dada classe Java convergem as noções de *subclasse* e *subtipo*, ainda que por mecanismos diferentes e devidamente identificados.
De facto, *se uma classe B é subclasse de A*, declaramos,

```
public class B extends A {
```

mas se em simultâneo a classe B *implementa uma interface I*, ie., *o tipo especificado em I*, deveremos então declarar,

```
public class B extends A implements I {
```

➡ Podemos agora igualmente analisar com mais detalhe as declarações de interfaces Java apresentadas anteriormente, tendo em atenção que as mesmas prescrevem *obrigatoriedade de implementação de todos os métodos abstractos* por parte das classes que declararem que são suas implementações, conforme a declaração *implements*.

Assim, a primeira declaração de interface,

```
public interface Enumeravel {
  public abstract inicio_enum();
  public abstract boolean vazia();
  public abstract Object seguinte();
}
```

obriga a que qualquer classe implementadora apresente uma implementação concreta para os métodos `inicio_enum()`, `vazia()` e `seguinte()`. Deste modo, qualquer classe que declare implementar a *interface* `Enumeravel`, corresponde à *implementação de um tipo de dados* que obedece à especificação da interface, ou seja, que possui *a funcionalidade necessária para que seja de facto enumerável*. Ou seja, *interfaces* definem *propriedades ou requisitos operacionais*.

Consideremos a título de exemplo a criação de uma *StackEnumeravel*, ou seja, uma subclasse da classe java.util.Stack e que implementa a interface *Enumeravel*. Tal classe deveria possuir a seguinte definição, cf. a definição da *interface*:

```
import java.util.*;

public class StackEnumeravel extends Stack
                                        implements Enumeravel {
// sendo todo o resto herdado, eis as variaveis e os
// metodos que implementam a interface Enumeravel !

    private Stack stackCopy;

    public boolean vazia() {
         return stackCopy.empty();
    }

    public inicio_enum() {
       // faz agora a cópia actualizada da stack de trabalho
       stackCopy = (Stack) this.clone() ;
    }

    public Object seguinte() {
        Object o = stackCopy.peek();
        stackCopy.pop();
        return o;
    }
}
```

A classe de teste que se apresenta a seguir,

```
public class TstStackEnum {
  //
  public static void main(String args[]) {
  //
  StackEnumeravel stack1 = new StackEnumeravel();

  System.out.println("Dimensao Max. da Stack1 = " + stack1.size());

  stack1.push("anos");
  stack1.push(new Integer(20));
  stack1.push("tem");
  stack1.push("Maria");
  stack1.push("A");

  try {
        System.out.println("topo = " + stack1.peek());
     }
  catch(EmptyStackException e)
          { System.out.println("STACK VAZIA !"); }

  stack1.pop(); stack1.pop();

  System.out.println("Dimensao = " + stack1.size());

  try {
        System.out.println("topo = " + stack1.peek());
     }
  catch(EmptyStackException e)
        { System.out.println("STACK VAZIA !"); }

  // enumeração

  stack1.inicio_enum();    // fixa stack a ser enumerada

  while (!stack1.vazia()) {
          System.out.println(stack1.seguinte());
  }
 }
}
```

com os seguintes resultados,

```
c:\jdk1.1> java TstStackEnum
0
A
3
tem
tem
20 anos

c:\jdk1.1
```

❑ Se eventualmente pretendessemos uma *Stack limitada* com as *propriedades operacionais* especificadas pela *interface*, bastaria passarmos a ter a declaração

```
public class StackLimEnumeravel extends StackLim
                                implements Enumeravel {
```

⇒ Adicionalmente, tal como se pode depreender da declaração seguinte,

```
public interface Desenhavel extends Colorivel {
  public abstract void posicao(double x , double y);
  public abstract void desenha();
}
```

as *interfaces* Java *possuem a sua própria hierarquia*, podendo-se assim falar de *superinterfaces* e de *subinterfaces*.
No exemplo anterior, a interface `Desenhavel` declara que *extends* `Colorivel`, ou seja, passa a ser uma *subinterface* de `Colorivel`, a qual aumenta declarando dois métodos.

⇒ A declaração seguinte,

```
public interface Transformavel extends Escalavel,
                                       Rodavel,
                                       Desenhavel {

    ...........................
}
```

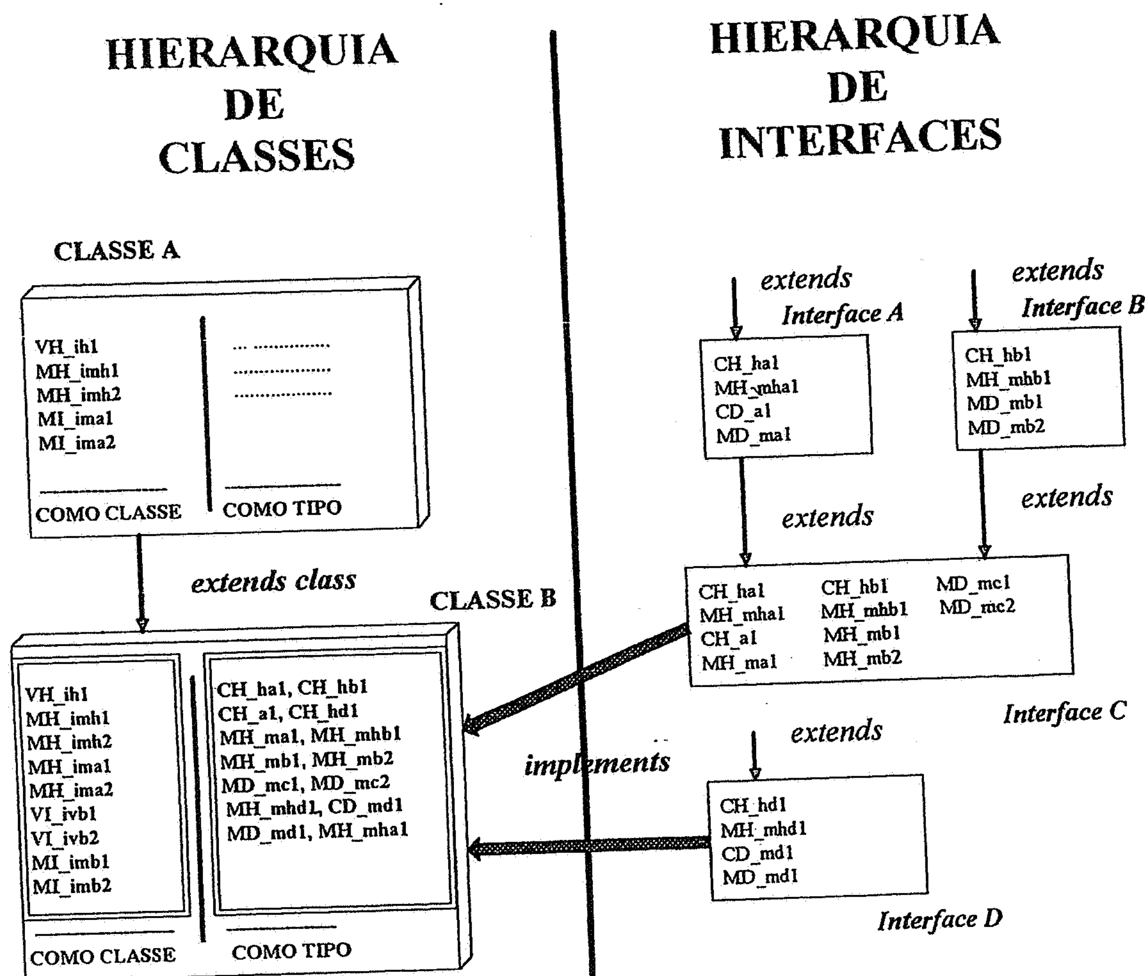
revela agora qual o tipo de *hierarquia lógica* (ou seja sintáctica e apenas de comportamento) foi implementada em Java para as *interfaces*. De facto, ao declarar que `extends` as interfaces declaradas, a interface `Transformavel "herda"` os métodos abstractos declarados em todas as interfaces indicadas, tratando-se portanto de um mecanismo de *herança múltipla* (ainda que esta herança não possa ser comparada, em termos de complexidade, com a herança usual entre classes, dado tratar-se de um mecanismo de puro *enriquecimento sintáctico*).
Uma *interface* pode ter portanto várias *superinterfaces*

⇒ Conceptualmente, as interfaces Java estão ligadas a certas *propriedades operacionais* que se pretendem ver realizadas nas diversas implementações do respectivo tipos de dados, tais como, o tipo ser enumerável, o tipo ser clonável (e talvez colunável também), ser visualizável, ser colorível, ser ordenável, etc., o que é garantido dado que *qualquer classe que implemeta uma interface tem obrigatoriamente que fornecer uma implementação concreta para cada um dos métodos abstractos declarados na interface, bem como para todos os por esta herdados.*

⇒ Havendo uma clara separação entre subtipos e subclasses, é curioso verificar-se que serão em qualquer caso as *classes* Java a fornecerem as implementações. Porém, a partir de agora, uma classe passará a ter duas *"views"* (ou interpretações) possíveis:

- É **subclasse** por se encontrar inserida na hieraquia normal de classes, que possui um mecanismo de herança simples de estrutura e comportamento com redefinições;
- É **subtipo** por se encontrar "inserida", ainda que apenas como mecanismo de implementação, numa hierarquia lógica, com herança múltipla de comportamento abstracto, ou seja, de especificações sintácticas.

A figura seguinte procura apresentar esta nova mas mais completa perspectiva das classes de Java.



➯ As *interfaces* Java parecem ser excelentes substitutas para as *classes abstractas*. Porém, esta ideia aparentemente correcta dado existir alguma possível semelhança entre uma classe 100% abstracta e uma interface, não tem em consideração um grande número de diferenças (para além das conceptuais que acabámos de apresentar) entre as duas entidades da linguagem Java, entre outras:

- *Uma classe abstracta pode não ser 100% abstracta; Uma interface é sempre 100% abstracta;*
- *Uma classe abstracta não impõe às suas subclasses a implementação obrigatória dos métodos abstractos; Uma interface impõe, a quem a implementar, uma implementação completa;*
- *Uma classe abstracta pode ser usada para se escrever software genérico, parametrizável e extensível; Uma interface não tem, em princípio, tais objectivos.*
- *Classes e Interfaces "vivem" em Hierarquias com propriedades distintas, resultando o poder da linguagem Java na simbiose das duas, o que requer, naturalmente, muita prática em desenvolvimento de grandes aplicações.*

⇒ Todas as interfaces Java são implicitamente abstractas pelo que a utilização do modificador `abstract` na sua declaração é redundante, sendo mesmo desaconselhada. Por outro lado, as interfaces Java devem ser todas declaradas como `public`.

⇒ As constantes declaradas numa interface são, por defeito, `public`, `static` e `final`, pelo que não é necessário declarar mais do que o seu *tipo*.

Compatibilidade entre Interfaces e Classes.

⇒ Sendo, como se disse, as *interfaces* Java especificações de tipos de dados a serem implementadas em classes, então os *identificadores de interfaces* são, naturalmente, *identificadores de tipos*, podendo pois ser usados na *declaração de variáveis*, conforme, em

```
Enumeravel enum, enum1;
Desenhavel f1, f2;
Colorivel fc;
```

⇒ Estas variáveis, cf. `enum` e `enum1`, por exemplo, podem agora ser associadas a quaisquer instâncias de classes que *implementem o seu respectivo tipo*, isto é, sejam com o mesmo compatíveis. Por exemplo, dado que a classe `StackEnumeravel` é uma implementação da interface `Enumeravel`, a seguinte associação é válida,

```
Enumeravel enum;
StackEnumeravel stack2 = new StackEnumeravel();
..................................
enum = stack2;

enum.inicio_enum();
while(!enum.vazia()) {
      System.out.println(enum.seguinte());
}
```

⇒ A compatibilidade é *absolutamente restrita ao conjunto de métodos de implementação* da interface `Enumeravel`, gerando um erro de compilação qualquer tentativa de utilização da variável do tipo `Enumeravel` usando métodos não pertencentes ao tipo (*interface*), tais como:

```
enum = stack2;          // Enumeravel <= StackEnum; OK !!
enum.push("ABC");       // erro de compilação.
enum.empty();           // erro de compilação
```

⇒ Se, porventura, uma dada classe implementa mais do que um tipo de dados (*interface*), e admitindo que tais interfaces não têm métodos nem constantes em comum (são *disjuntas*), as considerações que podem ser colocadas continuam a obedecer ao princípio anterior, ou seja, a associação entre *variáveis de dado tipo* e *instâncias de dada subclasse de implementação* de tal tipo é compatível. Vejamos um exemplo, sendo as interfaces em questão

```
public interface Colorivel {
   public abstract void defineCor(int cor);
 }
```

```
public interface Desenhavel extends Colorivel {
  public abstract void posicao(double x , double y);
  public abstract void desenha();
}
```

e a classe de implementação,

```
public class QuadradoDC
                    extends Quadrado
                    implements Desenhavel, Colorivel {
```

são perfeitamente correctas as associações resultantes do seguinte código,

```
Colorivel var_col;
Desenhavel var_des;
QuadradoDC qdc = new QuadradoDC();

var_col = qdc;   // responde à parte Colorivel de QuadradoDC
var_des = qdc;   // responde à parte Desenhavel de QuadradoDC
```

dentro das limitações de compatibilidade de comportamentos atrás referidas.

Sobreposicão ("Overloading") e Visibilidade na Hierarquia de Interfaces.

❐ Existindo uma hierarquia de interfaces, de herança múltipla até, torna-se importante analisar quais as regras da linguagem Java relativamente à sobreposição de constantes e métodos nas seguintes possíveis circunstâncias relacionadas apenas com a *implementação de interfaces* por uma dada classe,

1. *Uma subinterface* ***redefine uma constante herdada de uma superinterface;***
2. *Uma subinterface* ***herda constantes com o mesmo nome das suas superinterfaces;***
3. *Uma subinterface* ***redefine uma constante herdada de várias superinterfaces;***
4. *Uma subinterface* ***redefine um método herdado de uma superinterface;***
5. *Uma subinterface* ***herda métodos com o mesmo nome das suas superinterfaces;***
6. *Uma subinterface* ***redefine um método herdado de várias superinterfaces;***

e ainda, analisar qual o grau de compatibilidade das redefinições e herança em *interfaces* com idênticas redefinições e heranças pelo facto da classe de implementação ser também uma *subclasse* normal na hierarquia de classes.

❐ Analisemos em primeiro lugar cada uma das situações relacionadas com herança em interfaces, através de exemplos simples que permitem determinar com clareza quais as regras Java a aplicar em tais situações.

⇒ Vejamos em primeiro lugar o que acontece em **herança simples de constantes** entre interfaces, através de um exemplo. Consideremos as interfaces seguintes:

```
public interface Interf1 {
// constantes
    int x = 1;    // implicitamente são static final
    int y = 2;
    int z = 3;
// métodos
    public abstract int soma(); // são sempre abstract
}

public interface Interf2 extends Interf1 {
// constantes
    int x = 10;
    int y = 20;

}
```

em que `Interf2 extends Interf1` redefinindo directamente as constantes `x` e `y`, porém não redefinindo a constante `z`. Qualquer classe que implemente a `Interf1` ou a `Interf2` deve ainda implementar o método `soma()` de modo compatível com a sua definição.

Consideremos agora as duas classes seguintes. A classe `Class1` implementa `Interf1`, e a classe `Class2` implementa a `Interf2` que herda de `Interf1` de que é superinterface.

```
public class Class1 implements Interf1 {
    public int soma() {
        return x+y+z;
    }
}

public class Class2 implements Interf2 {
    public int soma() {
        return x+y+z;
    }
}
```

Vamos agora criar um programa de teste para verificarmos os resultados obtidos através da invocação dos respectivos métodos de instância. Seja o programa de teste o seguinte,

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

        Class1 c1 = new Class1();
        Class2 c2 = new Class2();

        //

        System.out.println(c1.soma());
        System.out.println(c2.soma());

    }
}
```

Após a compilação com sucesso das interfaces e das classes, a execução do programa de teste produziria os seguintes resultados:

```
c:\jdk1.1> java TstInt
6
33

c:\jdk1.1>
```

Análise:

1) O primeiro resultado representa a execução normal do método `soma()`, que devolve a soma dos valores constantes definidos em `Interf1`, tal como implementado em `Classe1` que é uma implementação correcta de `Interf1`.

2) O segundo resultado representa a execução do método `soma()`, que devolve a soma dos valores constantes definidos em `Interf2`, tal como implementado em `Classe2` que é uma implementação correcta de `Interf2`, verificando-se no entanto, pelo valor do resultado devolvido, que os valores das constantes `x` e `y` foram tomados como os valores redefinidos pela interface `Interf2`, enquanto que o valor da constante `z`, dado não ter sido redefinido, foi herdado e encontra-se acessível no contexto de uma classe de implementação de `Interf2`.

Questão: *Será possível numa classe de implementação de uma dada interface referir constantes das suas superinterfaces bem como as constantes de igual nome que possam ter sido localmente redefinidas na sua interface ?*

A classe seguinte vai procurar responder a esta questão.

```
public class Class3 implements Interf2 {
   public int soma() {
       return Interf1.x+y+z;
   }
}
```

Assim, se no programa de teste acrescentarmos ordenadamente as linhas,

```
Class3 c3 = new Class3();

System.out.println(c3.soma());
```

passaríamos a ter os seguintes resultados,

```
c:\jdk1.1> java TstInt
6
33
24

c:\jdk1.1>
```

Conclusões 1 - Herança simples de Constantes :

1) O modificador `final` em constantes definidas em interfaces (tal como nas definidas em classes) ***não impõe que as mesmas não possam ser redefinidas em subinterfaces***, ou seja, lhes possam ser associados novos valores;

2) ***Uma constante herdada de uma superinterface pode ser redefinida por uma subinterface***, podendo ainda qualquer classe de implementação da subinterface referir quer o valor redefinido quer os valores herdados das superinterfaces. Neste último caso, e por questões de ambiguidade, o identificador da constante deve ser prefixado com o identificador da interface de referência (cf. `Interf1.x`).

3) ***Uma constante herdada de uma superinterface que não seja redefinida por uma subinterface***, pode ser referida directamente em qualquer classe de implementação da subinterface sem necessidade de prefixo, e mantendo naturalmente o valor definido e herdado.

➾ Vejamos agora o que se passa em situações de **herança múltipla de constantes**. Consideremos as seguintes interfaces exemplo:

```
public interface Interf3 extends Interf1, Interf2 {
// constantes
     int x = 100;
     int y = 200;
     int z = 300;
// métodos
     public abstract int soma();
}

public interface Interf4 extends Interf1, Interf2 {
}
```

A interface `Interf3` herda das superinterfaces `Interf1` e `Interf2` constantes e métodos com o mesmo nome, mas redefine todas as constantes e impõe uma nova implementação do método.
A interface `Interf4` herda das superinterfaces `Interf1` e `Interf2` constantes e métodos com o mesmo nome, não redefinindo as constantes nem impondo uma nova implementação do método.
Ambas as interfaces compilam sem erros.

Consideremos agoras as seguintes possíveis classes de implementação,

```
public class Class3 implements Interf3 {
   public int soma() {
       return x+y+z;
   }
}

public class Class4 implements Interf4 {
   public int soma() {
       return x+y+z;
   }
}
```

Note-se, em primeiro lugar, que ambas as interfaces compilam sem qualquer tipo de erros, ainda que se possa observar que `Interf4` herda duas vezes os mesmos identificadores de constantes (ainda que com valores distintos) e o mesmo identificador de método.

Porém, ao compilar as classes de implementação, a situação é distinta. A classe `Class3` compila sem erros, enquanto que a compilação de `Class4` gera os erros

```
c:\jdk1.1> javac Class4.java
  compiling: Class4.java
Class4.java(3): Reference to x is ambiguous. It is defined in interface
Interf2 and interface Interf1.
        return x+y+z;
               ^
Class4.java(3): Reference to y is ambiguous. It is defined in interface
Interf2 and interface Interf1.
        return x+y+z;
                 ^
2 errors

c:\jdk1.1>
```

A implementação de `Interf4` tal como definida em `Class4` é, naturalmente, ambígua dado que não define explicitamente a qual dos dois valores herdados se refere (dado que não criou novas definições locais, ao contrário de `Class3` que implementa `Interf3`, que embora herde igualmente de duas superclasses redefiniu as constantes).

Conclusões 2 - Herança múltipla de Constantes :

1) ***Uma constante herdada de mais do que uma superinterface pode ser redefinida por uma subinterface***, podendo ainda qualquer classe de implementação da subinterface referir quer o valor redefinido quer os valores herdados das suas superinterfaces. Neste último caso, e por questões de ambiguidade, os identificadores das constantes devem ser prefixados com os identificadores das interfaces de referência (cf. regra anterior para herança simples).

2) ***Uma constante herdada de mais do que uma superinterface que não seja redefinida por uma subinterface***, não pode ser referida directamente em qualquer classe de implementação da subinterface sem necessidade de prefixo, gerando um erro de ambiguidade na classe de implementação.

⇒ Assim, e a título de exemplo, dentro destas regras a classe

```
public class Class6 implements Interf4 {
   public int soma() {
       return Interf1.x+Interf2.y+z;
   }
}
```

compilaria sem problemas, e qualquer instância desta classe criada através de uma declaração do tipo,

```
Class6 c6 = new Class6();
```

responderia à mensagem apresentada com um valor que, apresentado via método `println()`, conforme a declaração,

```
System.out.println(c6.soma());
```

seria (cf. `Interf1.x (1) + Interf2.y (20) + z (3)`) igual a `24`.

❐ Vamos agora analisar o que se passa na **herança simples de métodos**, quer estes sejam ou não redefinidos nas subinterfaces. Consideremos as duas interfaces seguintes:

```
public interface Inter1 {
// constantes
    int a = 1;
    int b = 2;

// métodos
    public abstract int soma();
    public abstract int prod() ;
}

public interface Inter2 extends Inter1 {
// constantes
    int b = 20;

    public abstract int soma();
    public abstract int soma(int x) ;
    public abstract int prod(int x) ;

}
```

Consideremos agora as duas seguintes classes que procuram implementar as interfaces acima apresentadas,

```
public class Class1 implements Inter1 {
//
    public int soma() { return a+b; }
    public int prod() { return a*b; }
}


public class Class2 implements Inter2 {

    public int soma() { return a+b; }
    public int soma(int x) { return x+a+b; }
    public int prod() { return a*b; }
    public int prod(int k) { return a*k*b; }   // k <=> x OK !!
}
```

A classe `Class2` *é obrigada* a implementar os métodos definidos em `Inter2` e ainda o método `prod()` herdado de `Inter1`. Note-se ainda que a interface `Inter2` introduz dois novos métodos, `soma(int x)` e `prod(int x)`, que embora tenham os mesmos nomes de métodos herdados da superinterface *possuem uma assinatura diferente ainda que com o mesmo tipo de resultado*.

Vamos então executar o seguinte programa de teste,

```
public class TstInt1 {
   public static void main (String args[]) {

        Class1 c1 = new Class1();
        Class2 c2 = new Class2();
        //

        System.out.println(c1.soma());
        System.out.println(c1.prod());
       //
        System.out.println(c2.soma());
        System.out.println(c2.soma(10));
        System.out.println(c2.prod());
        System.out.println(c2.prod(5));
    }
 }
```

que produz os seguintes resultados,

```
c:\jdk1.1> java TstInt1
3
2
21
31
20
100

c:\jdk1.1>
```

Análise :

Estes resultados, que não merecem comentários particulares pois são de explicação evidente, mostram apenas que *os métodos herdados podem ser redefinidos naturalmente desde que mantenham a assinatura do método herdado*, que *novos métodos podem ser declarados mesmo que em sobreposição com métodos existentes, mesmo que com parâmetros diferentes mas com o mesmo tipo de resultado*.

Vejamos o que acontece se tentarmos redefinir um método herdado de uma superinterface ou criar um novo método de nome igual ao herdado mas com *tipo de resultado e parâmetros diferentes*. Seja para tal considerada a seguinte interface:

```
public interface Inter3 extends Inter1 {
// constantes
     int b = 20;

     public abstract float soma() ; // redefine int soma()
     public abstract boolean prod(int x, int y) ; // novo
 }
```

A compilação desta interface gera o erro seguinte, pelas razões apontadas,

```
c:\jdk1.1> javac Inter3.java
  compiling: Inter3.java
Inter3.java(4): Methods can't be redefined with a different return
type: float soma() was int soma().

1 error

c:\jdk1.1>
```

Note-se que o novo método `boolean prod(int x, int y)` não produziu erro de compilação, ou seja, não é incompatível com o método herdado `int prod()` dado terem parâmetros de entrada diferentes.

Corrigida a interface `Inter3` para,

```
public interface Inter3 extends Inter1 {
// constantes
     int b = 20;

     public abstract int soma() ; // redefine int soma()
     public abstract boolean prod(int x, int y) ; // novo
 }
```

uma classe de implementação tal como a apresentada a seguir poderia ser construída sem que quaisquer tipo de erros, quer de compilação quer de execução, surgissem agora.

```
public class Class3 implements Inter3 {

   public int soma() { return a+b; } ;
   public int prod() { return a*b; } ;
   public int soma(int x) { return a+x; } ;
   public boolean prod(int x, int y) { return x == y ;}
}
```

Analisadas que foram todas as possíveis situações de herança de métodos em interfaces, com ou sem redefinição, e ainda todas as situações de declaração de novos métodos, com ou sem colisão de identificadores e assinatura com métodos herdados, estamos pois em condições de apresentar um conjunto de conclusões relativamente à herança e redefinição de métodos em interfaces em situações de herança simples.

Antes porém de se apresentarem as conclusões, é importante analisar uma situação de referência idêntica à surgida relativamente às constantes, ou seja,

Questão: *Será possível numa classe de implementação de uma dada interface referir métodos das suas superinterfaces ?*

A classe seguinte vai procurar responder a esta questão.

```
public class Class3 implements Inter2 {

  public int soma() { return a+b; }
  public int soma(int x) { return x + Inter1.soma(); }
  public int prod() { return a*b; }
  public int prod(int k) { return k* Inter1.prod(); }
}
```

A compilação desta classe fornece a resposta desejada,

```
c:\jdk1.1> javac Class3.java
  compiling: Class3.java
Class3.java(7): Can't make static reference to method int soma()
in Inter1.
                    return x + Inter1.soma();
                                 ^
Class3.java(7): Can't make static reference to method int prod()
in Inter1.
                    return k * Inter1.prod();
                                 ^

2 errors

c:\jdk1.1>
```

Conclusões 1 - Herança Simples de Métodos :

1) Toda a análise relativa a métodos herdados e redefinidos e a novos métodos é feita tendo por base a *assinatura* dos mesmos;

2) Métodos herdados e não redefinidos não apresentam qualquer tipo de problemas;

3) Os métodos herdados só podem ser correctamente redefinidos se para o mesmo identificador e para o mesmo conjunto de tipos de parâmetros de entrada, for devolvido o mesmo tipo de resultado. Se para o mesmo conjunto de tipos de parâmetros de entrada, e para o mesmo identificador de método, for devolvido um tipo de resultado diferente, tal redefinição provoca um erro de compilação da subinterface (cf. `Inter3` na primeira versão).

4) Os métodos herdados podem ver os seus nomes "sobrecarregados" (*overloaded*) por métodos novos definidos nas subinterfaces, desde que, qualquer que seja o tipo de retorno destes, os tipos dos parâmetros sejam diferentes (cf. `boolean prod(int x, int y)` em relação a `int prod()` herdado na definição da interface `Inter3`).

5) Não podem ser feitas referências prefixadas a métodos das superclasses;

6) Ainda que as interfaces possam compilar sem erros, é necessário sempre analisar se as várias declarações não provocam colisões que geram erros na compilação das classes de implementação, tanto mais que as classes de implementação de interfaces herdam igualmente das suas superclasses na hierarquia de classes, e podem, portanto, acontecer colisões de nomes e funcionalidades em métodos.

❒ Vamos agora analisar o que se passa na **herança múltipla de métodos**, quer estes sejam ou não redefinidos na subinterface.

Vamos considerar um exemplo no qual se combinam todas as situação possíveis de conflitos em herança múltipla de métodos, designadamente:

- *métodos com o mesmo nome que têm o mesmo tipo de resultado e parâmetros;*
- *métodos com o mesmo nome que têm o mesmo tipo de resultado mas diferentes tipos nos parâmetros;*
- *métodos com o mesmo nome mas diferente tipo de resultado;*

Consideremos uma interface `Int11` definida como,

```
public interface Int11 {
// constantes
      int a = 1;
      int b = 2;

// métodos
      public abstract int soma();
      public abstract int prod() ;
      public abstract float div(int x, int z);
}
```

e consideremos também as duas interfaces seguintes, a última das quais irá herdar das duas anteriores, redefinindo métodos e herdando simplesmente outros,

```
public interface Int22 {
// constantes
      int b = 20;
//
      public abstract int soma();
      public abstract int soma(int x) ;
      public abstract float prod();
      public abstract boolean prod(int x, int y) ;
}

public interface Int33 extends Int11, Int22 {
//
      public abstract int soma(int x) ;
      public abstract float prod(int x, int y) ; // redefine
}
```

✏ No entanto, a interface `Int33` gera de imediato um erro de compilação, dado que estamos a tentar *redefinir* o método herdado `boolean prod(int, int)` com um tipo de resultado diferente.

✏ Ou seja, ainda que numa interface métodos possam ser declarados com o mesmo nome mas com assinaturas diferentes em redefinições de métodos herdados, alterações no *tipo do resultado* para os mesmos tipos de parâmetros não podem acontecer.

A interface passa a compilar sem erros se modificada para,

```
public interface Int33 extends Int11, Int22 {
//
      public abstract int soma(int x) ;
      public abstract boolean prod(int k) ;
 }
```

✏ Curiosamente, em termos de compilação, foi aceite que `Int33` possa herdar dois métodos de igual nome, iguais tipos de parâmetros de entrada e diferente resultado, como é o caso de `int prod()` herdado de `Int11` e `float prod()` herdado de `Int22`. Veremos posteriormente as consequências desta facilidade quando uma classe tiver que implementar `Int33`.

A `Int33` pode ser vista, sem regras de optimização, como declarando a seguinte lista de constantes e métodos (locais ou herdados) :

```
int a = 1;    // de Inter11
int b = 2;    // de Inter11
int b = 20;   // redefine Inter11.b

public abstract int soma(); // de Inter11
public abstract int soma(); // de Inter22

public abstract int prod();   // de Inter11
public abstract float prod(); // de Inter22

public abstract float div(int x, int z); // de Inter11

public abstract int soma(int x); // de Inter22
public abstract int soma(int x); // redefinição local

public abstract boolean prod(int x, int y);//de Inter22
public abstract boolean prod(int k); // local
```

Esta interface, assim declarada, já não produz qualquer erro de compilação, pelo que podemos desde já inferir quase todas as regras para a herança múltipla de métodos.

Conclusões 2 - Herança Múltipla de Métodos :

1) A herança múltipla de métodos e a sua correcta sobreposição, caso exista, é determinada com base nas suas assinaturas;

2) Uma interface pode herdar mais do que um método com a mesma assinatura, o que não provoca qualquer erro de compilação;

3) Uma interface pode herdar métodos com o mesmo nome e diferentes assinaturas (cf. no exemplo `int prod()` e `float prod()`) desde que não declare nenhum outro com o mesmo nome, iguais tipos de entrada e diferente tipo de resultado. Daí ser correcta a declaração de `boolean prod(int k)` na interface `Inter33`. Resta porém saber, quando tal interface tiver que ser implementada numa dada classe, qual dos métodos de igual tipo de parâmetros e tipo de resultado diferente deverá ser implementado.

4) ***Enfim, erros de compilação em interfaces só serão gerados se métodos herdados forem localmente redefinidos com o mesmo nome, iguais tipos de parâmetros de entrada, e diferentes tipos de resultado.***

❒ Vamos então verificar o grau de compatibilidade existente numa classe de implementação de `Inter33` entre os métodos que asseguram que tal classe implementa o tipo `Inter33`, e os métodos de instância da classe enquanto subclasse na hierarquia de classes. Vamos também verificar se surgem problemas ao nível da implementação pelò facto de serem herdados métodos de interface com igual aridade de entrada e diferente tipo de resultado.

➼ O exemplo seguinte demonstra o grau de compatibilidade. Para tal prova foi criada uma classe `Classe33` que implementa a interface `Inter33` e que, para além de ser subclasse da classe `Class1`, possui adicionalmente os seus próprios métodos de instância, alguns dos quais, propositadamente, colidem em nome com os métodos para implementação da interface declarada. Vejamos pois a declaração de tal classe, admitindo que a classe `Class1` para este exemplo tem a definição seguinte,

```
public class Class1 {
// variaveis de instancia
   int x = 1;
   Int y = 2;
//
   public int soma() { return x+y; }
   public int prod() { return x*y; }
}
```

```
public class Class33 extends Class1 implements Inter33 {

// implementacao do tipo Inter33

   public int soma() { return b+b; }
   public int soma(int x) { return x+b; }
   public int prod() { return b*b; }
   public float prod() { return (float) b; }
   public boolean prod(int x, int y) { return x==y; }
   public boolean prod(int x) { return x>0; }
   public float div(int x, int y) { return (float) x/y ; }

// metodos de instancia

   public float soma(float x, float y) { return x+y; }
   public float prod(float x, float y) { return x*y; }

}
```

➼ Porém, e apesar da interface `Int33` não ter apresentado qualquer erro de compilação, ao ser compilada a classe de implementação `Class33` o seguinte erro de compilação da classe é apresentado:

```
c:\jdk1.1> javac Class33.java
  compiling: Class33.java
Class33.java(8): Methods can't be redefined with a different return
type: float prod() was int prod().
       public float prod() { return (float) b; }
              ^

1 error

c:\jdk1.1>
```

ou seja, a herança múltipla dos métodos `prod()` com iguais parâmetros de entrada e diferente tipo de resultado *passou como compatível ao nível das interfaces*, mas não passa ao nível das implementações.

➼ Qualquer tentativa para solucionar o problema ao nível da implementação, ou seja, ao nível da classe `Class33`, tal como, não definir nenhum dos dois métodos ou definir apenas um deles, gera o mesmo erro de compilação da classe. A solução única, portanto, é mesmo alterar uma das interfaces importadas por `Inter33`, por forma a que não haja tal tipo de herança múltipla. Vamos portanto admitir que Inter22 define também o método `int prod()`.
A classe Class33 passará a ser,

```
public class Class33 extends Class1 implements Inter33 {

// implementacao do tipo Inter33

  public int soma() { return b+b; }
  public int soma(int x) { return x+b; }
  public int prod() { return a*b; }
  public boolean prod(int x, int y) { return x==y; }
  public boolean prod(int x) { return x>0; }
  public float div(int x, int y) { return (float) x/y ; }

// metodos de instancia

   public float soma(float x, float y) { return x+y; }
   public float prod(float x, float y) { return x*y; }
 }
```

Porém, a compilação desta implementação apresenta ainda erros resultantes da ambiguidade na utilização de constantes herdadas multiplamente, designadamente relacionadas com `b`, cf.

```
     Class33.java(5): Reference to b is ambiguous. It is defined in interface
Int11 and interface Int22.
     ............
     4 errors
```

Portanto, na classe de implementação, para tais casos, somos obrigados a definir se nos estamos a referir a uma constante ou a outra usando prefixos que são os identificadores das respectivas interfaces. A implementação passaria a ser, por exemplo,

```
public class Class33 extends Class1 implements Inter33 {

// implementacao do tipo Inter33

  public int soma() { return Int22.b + Int11.b; }
  public int soma(int x) { return x + Int22.b; }
  public int prod() { return a* Int11.b; }
  public boolean prod(int x, int y) { return x==y; }
  public boolean prod(int x) { return x>0; }
  public float div(int x, int y) { return (float) x/y ; }

// metodos de instancia

   public float soma(float x, float y) { return x+y; }
   public float prod(float x, float y) { return x*y; }

}
```

A execução para teste desta definição de classe, contida no código da classe de teste que se apresenta a seguir,

```
public class TstClass33 {
  public static void main (String args[]) {
  Class33 c33 = new Class33();
  //
  System.out.println("c33.soma() = " + c33.soma());
  System.out.println("c33.soma(12) = " + c33.soma(12));
  System.out.println("c33.prod() = " + c33.prod());
  System.out.println("c33.div(10, 12) = " + c33.div(10, 12));
  System.out.println("c33.prod(10, 11) = " + c33.prod(10,12));
  System.out.println("c33.prod(5) = " + c33.prod(5));
  System.out.println("c33.soma(0.12f, 2.25f) = " +
                       c33.soma(0.12f, 2.25f));
  System.out.println("c33.prod(0.12f, 2.25f) = " +
                       c33.prod(0.12f, 2.25f));
  }
}
```

produziria os seguintes resultados,

```
c:\jdk1.1> java TstClass33
c33.soma() = 22
c33.soma(12) = 32
c33.prod() = 2
c33.div(10, 12) = 0.833333
c33.prod(10, 11) = false
c33.prod(5) = true
c33.soma(0.12f, 2.25f) = 2.37
c33.prod(0.12f, 2.25f) = 0.27
```

Conclusões 3 - Herança Múltipla de Métodos :

- *Ainda que tal não gere qualquer tipo de erro na compilação das interfaces, não faz sentido que uma interface possa herdar de outras métodos com a mesma assinatura de parâmetros mas diferente tipo de resultado, porque o erro irá surgir na compilação da respectiva classe de implementação. Assim, tal tipo de métodos é impossível de implementar, pelo que não devem ser definidas interfaces com tal tipo de incompatibilidade relativamente às suas superinterfaces.*

❒ Falta-nos apenas observar a compatibilidade entre variáveis de instância de uma dada classe e os identificadores de constantes implicitamente herdados pelo facto da classe implementar uma dada interface. Para tal vamos reescrever a classe `Class33` como sendo,

```
public class Class33 extends Class1 implements Inter33 {

// variaveis de instancia
int a = 1000;
int b = 2000;

// implementacao do tipo Inter33

  public int soma() { return Int22.b + Int11.b; }
  public int soma(int x) { return x + Int22.b; }
  public int prod() { return a* Int11.b; }
  public boolean prod(int x, int y) { return x==y; }
  public boolean prod(int x) { return x>0; }
  public float div(int x, int y) { return (float) x/y ; }

// metodos de instancia

   public float soma(float x, float y) { return x+y; }
   public float prod(float x, float y) { return x*y; }

}
```

Agora, o mesmo programa de teste produziria uma simples, mas significativa, alteração nos resultados, que seria,

```
..........
c33.prod() = 2000
............
```

dado que, na omissão de identificador de constante herdada via interface, o método `prod()` assume `a` como referência à variável de instância, sendo `b` prefixado, logo não ambíguo.

➯ Estão assim caracterizadas *quase todas* as regras de compatibilidade na hierarquia de interfaces, bem como as regras de compatibilidade entre esta hierarquia e a hierarquia de classes, dado serem as classses o mecanismo existente em Java para implementação das interfaces.

➯ Falta apenas analisar situações em que métodos lançam excepções e em que métodos necessitam de sincronização, situações que serão analisadas em contexto próprio.

8. I/O EM JAVA. AS STREAMS

Objectivos

- Introdução ao I/O básico de JAVA;
- Introdução ao conceito de *stream*;
- *Streams* de caracteres versus *streams* de *bytes*;
- Implementações de Writer e Reader;
- Implementações de Input e OutputStreams;
- Criação e manipulação de *streams* de objectos *serializable*;
- *ObjectStreams* e persistência de objectos e dados;

8.1 INTRODUÇÃO

Este capítulo tem por objectivo introduzir as técnicas e os mecanismos de JAVA que nos irão possibilitar programar correctamente a leitura de dados a partir de um qualquer dispositivo de entrada, e a escrita de dados para um qualquer dispositivo de saída. O objectivo principal é estudarmos um conjunto de classes que nos irão permitir ter acesso para leitura a dados de que necessitamos para a execução dos nossos programas, e, também, para que possamos armazenar de forma permanente dados criados ou manipulados pelos programas JAVA que criamos.

Não se trata, no entanto, de um capítulo sobre o desenvolvimento de *interfaces com o utilizador*, ou seja, de *software* especial de interacção, nem sobre *applets*, assuntos que, dada a sua especificidade e extensão, têm sido abordados em obras que lhes são inteiramente dedicadas.

Como sabemos, porventura de experiências com outras linguagens, as operações de *input/output*, a partir de agora designadas por I/O, lidam com dispositivos físicos muito diversos, em geral a muito baixo nível, sendo também operações muito susceptíveis de gerarem situações de erro. Por isso, o mecanismo de excepções de JAVA é um mecanismo muito importante de ser bem conhecido antes de se poder abordar o I/O em JAVA.

Por outro lado, o que sempre se espera de uma linguagem de programação, é que consiga "proteger" os que a utilizam de todos os inúmeros e enfadonhos detalhes de um I/O de baixo nível. Para tal, espera-se sempre que a linguagem ofereça um nível de abstracção grande, de modo a que a programação de operações de I/O se realize a um nível de grande simplicidade. Ainda que todos saibamos que a baixo nível tudo se realiza sob a forma de leitura ou escrita de *bytes*, não é a esse nível que se pretende realizar a codificação.

Adicionalmente, as fontes de leitura e os destinos de escrita de dados são, hoje em dia, muito diversificados, tando podendo tratar-se de dispositivos locais, como de fontes ou destinos acessíveis por uma ligação a uma rede, etc. No entanto, e caso seja feita a muito alto nível, a manipulação de tais dados de origens diversas é, essencialmente, a mesma ou muito semelhante. Há, assim, alguma possibilidade de generalização e de abstracção.

Em JAVA, todas as considerações (ou quase todas) que se relacionam com as mais diferentes formas de se realizar a *leitura* e a *escrita de dados* a partir das mais diversas fontes e para os mais diferentes destinos, são reunidas, e abstraídas, no conceito de *stream*.

Uma *stream* é uma abstracção que representa uma *fonte genérica de entrada de dados ou um destino genérico para escrita de dados, de acesso sequencial e independente de dispositivos físicos concretos, formatos ou até de mecanismos de optimização de leitura e escrita*. É, portanto, uma abstracção e, como tal, terá que ser sempre refinada e concretizada, e, em particular, ser associada a uma entidade física de suporte de dados, seja um ficheiro em disco ou em CDROM, um *website*, um *array* de *bytes*, uma *string*, um DVD, etc.

Em JAVA, existirão duas grandes classes de *streams*, designadamente:

- *streams de caracteres, ou seja, streams de texto;*
- *streams de bytes, ou seja, streams binárias.*

Salvo raras excepções, um objecto a partir do qual seja possível ler sequências de *bytes* ou caracteres será uma *input stream*. Igualmente, um qualquer objecto no qual possam ser escritas sequências de *bytes* ou de caracteres será uma *output stream*.

Todas estas classes, que irão implementar formas particulares de I/O em *streams*, serão subclasses de quatro classes abstractas do *package* `java.io`, que se designam por `InputStream` e `OutputStream` para as *streams de bytes*, e por `Reader` e `Writer` para as *streams* de caracteres. Estas classes, sendo abstractas, definem o conjunto de métodos que uma qualquer implementação de tais *streams* deverá concretizar.

O *package* `java.io` contém praticamente todas as classes e algumas interfaces que, em conjunto, representam todas as diferentes formas de se realizar I/O, ou seja, que implementam as abstracções que são as *streams*. São estas classes e tais interfaces que estudaremos a seguir de forma estruturada. A Figura 8.1 deverá ser utilizada como guião, dado o elevado número de classes de I/O existentes.

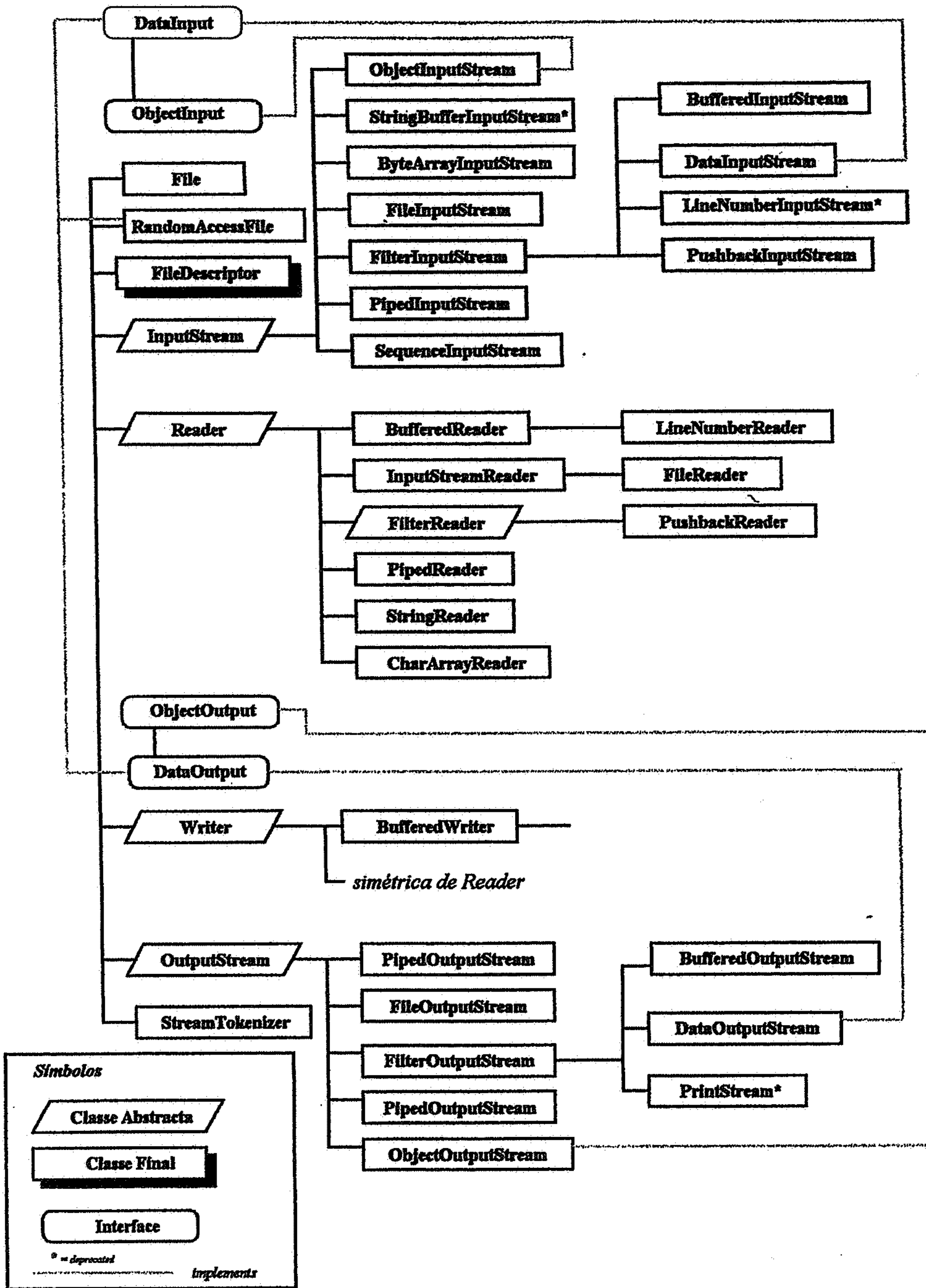


Fig. 8.1 – Hierarquia de classes e interfaces de `java.io`

8.2 I/O BÁSICO

A classe abstracta `InputStream` define um conjunto básico de métodos que deverão ser implementados por qualquer *input stream* de JAVA que deva ler *bytes* de uma dada fonte. Destes, o método fundamental é o método especificado como:

```
public abstract int read() throws IOException
```

Este método básico, sendo abstracto, especifica a leitura de um *byte* de uma *input stream*, e a devolução do valor inteiro desse *byte*, ou do valor –1 caso se atinja o fim da *stream* fonte. Por exemplo, o implementador de uma `FileInputStream`, *stream* de *input* que deve ler *bytes* a partir de um ficheiro, deve garantir que este método lê um *byte* de um ficheiro. No entanto, o implementador da classe que se designa `ByteArrayInputStream`, igualmente subclasse de `InputStream`, terá que garantir que a leitura de tal *byte* é feita a partir de um *array* de *bytes*.

Tal como vimos anteriormente, a classe `java.lang.System`, oferece alguns serviços básicos de I/O através de três variáveis de classe, `in`, `out` e `err`, que são de facto instâncias de *streams* que estão associadas, respectivamente, a operações de leitura a partir do teclado e de escrita no monitor, e que cumprem a missão de implementar a *standard input*, *output* e *error*. Daí termos já, anteriormente, usado construções de escrita tais como:

```
System.out.print("abc");
System.out.println("Rita e Pedro");
```

As leituras *byte* a *byte* são trabalhosas quando o que, de facto, se pretende ler são dados que, mesmo sendo de tipos simples, possuem uma representação que deve obedecer a uma gramática. A sua leitura passa a ser também uma validação através de código de reconhecimento (ou *parsing*), suficientemente robusto para que seja capaz de tratar as situações de leituras de valores inválidos.

A classe `Consola` cujo código se apresenta a seguir, vem servir, neste momento, dois propósitos distintos. Por um lado, serve para demonstrar quão trabalhoso é o código de I/O quando deve ser desenvolvido a baixo nível, ou seja, caso não pudéssemos dispor de implementações de alto nível. Por outro, porque se trata de uma classe relativamente útil de ser reutilizada em programas que se pretendam um

pouco mais interactivos, já que implementa a leitura, validada, de alguns valores de tipos simples.

```
import java.lang.*;
import java.io.*;
public class Consola {

// métodos de classe

   // escreve uma mensagem no ecrã sem fazer <new line>.
   // flush() é importante para esvaziar o buffer.

   public static void escreveTxt(String txt) {
     System.out.print(txt + ":");
     System.out.flush();
   }


   // lê uma string caracter a caracter, terminada por
   // <ENTER> ou <ESC>

   public static String leStr() {
     int ch;
     String r = "";        // string resultado
     boolean fim = false;
     while (!fim) {        // ciclo de leitura de caracteres
       try
           { ch = System.in.read();
             if (ch < 0 || ch == 27 || ch == 13) //ESC ou CR
               { fim = true;
                 // se for ESC (27) tem que consumir CR/LF
                 if (ch==27) { ch = System.in.read();
                               ch = System.in.read();}
                  // se for CR (13) tem que consumir o LF
                 if (ch==13) { ch = System.in.read();}
               }
             else
                r = r + (char) ch;  // casting int -> char
           }
         catch(IOException e)
           { fim = true;}
     }
     return r;
   }
```

```
// lê uma string após escrever mensagem

public static String leStr(String txt) {
   escreveTxt(txt);
   return leStr();       // devolve a string lida.
}

// lê um inteiro válido após escrita da mensagem.

public static int leInt(String txt) {
  while (true) {
    escreveTxt(txt);
    try
      { return
          Integer.valueOf(leStr().trim()).intValue(); }
    catch(NumberFormatException e)
    { System.out.println("Não é um inteiro válido !!"); }
  }
}

// A leitura é realizada caracter a caracter, usando o
// método leStr(). Da string lida, são eliminados os
// espaços, servindo a string resultante de parâmetro
// do método valueOf(), da classe Integer, que tentará
// converter a string num objecto Integer.
// Se a string for válida, a conversão é feita, e a
// mensagem intValue() enviada à instância de Integer
// vai produzir o valor do tipo int que é desejado como
// sendo o resultado deste método.
//
// Todos os métodos seguintes usam métodos semelhantes
// das respectivas classes (cf. Float, Double, etc.).

// lê um Double

public static double leDouble(String txt) {
  while (true) {
    escreveTxt(txt);
    try
      { return
          Double.valueOf(leStr().trim()).doubleValue(); }
    catch(NumberFormatException e)
     { System.out.println("Não é um double válido !"); }
  }
}
```

```
  // lê um Float

  public static float leFloat(String txt) {
    while (true) {
      escreveTxt(txt);
      try
        { return
          Float.valueOf(leStr().trim()).floatValue(); }
      catch(NumberFormatException e)
        { System.out.println("Não é um float válido !"); }
    }
  }

  // lê um boolean

  public static boolean leBool(String txt) {
    while (true) {
      escreveTxt(txt);
      try
        { String s = leStr().trim().toLowerCase();
            if ( s.equals("true") || s.equals("false"))
              return Boolean.valueOf(s).booleanValue();
            else
               throw new IllegalArgumentException(""); }
       catch(IllegalArgumentException e)
        { System.out.println("Não é um booleano !"); }
    }
  }
}
```

A classe `System` possui duas variáveis de classe, `in` e `out`, que são, de facto, instâncias de duas *streams* de JAVA, designadamente, de uma `InputStream` que é associada ao teclado, e através da qual se lêem *bytes* usando o método `read()`, e de uma `PrintStream`, uma *stream* de escrita que recebe caracteres e que está associada ao monitor, na qual se escreve usando `print()` e `println()`.

Naturalmente, como se verifica, este I/O é muito básico, sendo, de facto, muito raro que os programadores de JAVA tenham que utilizar métodos tão primitivos quando os dados que pretendem ler ou escrever são números, textos e objectos.

Vamos, portanto, estudar em seguida algumas das classes que representam as mais importantes implementações quer de `OutputStream` quer de `Writer`, para as operações de escrita, quer de `InputStream` quer de `Reader`, para as operações de leitura, e que colocam a muito mais alto nível todas as questões relacionadas com operações de I/O em JAVA.

8.3 PACKAGE JAVA.IO

A Figura 8.1 apresenta, de forma detalhada, esta hierarquia de classes, que deverá ser seguida durante a apresentação das várias implementações de *streams* que iremos estudar neste capítulo.

Como é possível verificar, analisando tal hierarquia, existem no *package* `java.io` algumas classes particulares que não são *streams*, ou seja, que não são subclasses de `InputStream` nem de `OutputStream`, nem de `Reader` nem de `Writer`, tal como, por exemplo, as classes `File` ou `RandomAccessFile`, mas que, ainda assim, representam componentes típicos de escrita e leitura de dados, tais como ficheiros organizados em sistemas de directorias e ficheiros de texto em disco.

Comecemos por estudar algumas destas classes de I/O que não são *streams*.

CLASSES NÃO *STREAM* DE JAVA.IO

CLASSE FILE

A primeira classe *não-stream* representativa de dispositivos comuns de I/O é a classe `File`, que representa os usuais *ficheiros e directorias de um sistemas de ficheiros*, ainda que de forma independente da plataforma (sistema operativo). Esta classe define métodos para a realização das usuais operações sobre ficheiros e directorias, tais como, realizar a listagem de directorias, apagar um ou mais ficheiros, determinar atributos de um ficheiro, alterar o seu nome, etc.

Dado não ter qualquer tipo de relação com as classes abstractas que especificam as *streams*, a classe `File` é uma subclasse de `Object`.

Vejamos, apenas a título de exemplo, os passos principais para se estabelecer uma associação entre uma instância de `File` e um ficheiro em disco, e algumas das operações simples que podem ser realizadas:

```
File f1 = new File("c:\jdk\TstFile1.java");
// f1 é o nome lógico do ficheiro físico indicado.
String nome = f1.getName());
String path = f1.getPath());
boolean teste1 = f1.isFile();
boolean teste2 =  f1.isDirectory();
int comp = f1.length());
f1.delete();
```

Caso a variável `f1` tivesse (cf. resultado do interrogador `isDirectory()`) sido associada a uma *directoria*, então as seguintes mensagens passariam igualmente a ter significado,

```
String[] files = new String[200];
files = f1.list(); // lista ficheiros da directoria
boolean res = f1.renameTo(new File("c:\jdk\Testes"));
```

para além de todas as anteriormente apresentadas, que se aplicam indistintamente a ficheiros e a directorias.

INTERFACE FILENAMEFILTER

Esta interface define um método, `accept(File dir, String nomefich)`, que permite que o método `list(FilenameFilter filtro)` seja invocado sobre uma directoria, mas faça apenas a selecção dos nomes dos ficheiros que satisfazem o filtro especificado como parâmetro.

A construção de um filtro do tipo `FilenameFilter` passa, em geral, por se criar uma classe de implementação com um construtor que aceita um padrão de texto a ser comparado com a *string* que representa o nome do ficheiro, e cuja comparação é implementada no método `accept()`. Naturalmente que quanto mais genérico

for o código do método de filtragem `accept()` mais diferentes filtragens teremos num só método.

Vamos, por exemplo, criar um filtro que fará o "matching" de um prefixo com o início do nome de um ficheiro e ainda com a respectiva extensão. Qualquer nome de ficheiro obedecendo a tal padrão é seleccionável via `accept()` pois o método devolverá `true` na comparação.

```
public class Filtro1 implements FilenameFilter {

  private String prefixo;
  private String sufixo;

  public Filtro1(String pref, String extensao) {
     this.prefixo = pref;
     this.sufixo = "." + extensao;
  }

  public boolean accept(File dir, String filename) {
     return (filename.startsWith(prefix) &&
             filename.endsWith(sufixo));
  }
}
```

A criação efectiva de um filtro consiste em associar a uma variável do tipo `FilenameFilter` uma instância da classe `Filtro1`, que são compatíveis pois a classe implementa a interface, e passar tal filtro como parâmetro da mensagem `list()`. Teríamos então:

```
FilenameFilter f = new Filtro1("Tst", "java");
File dir = new File("c:\jdk\minhas");
String[] files = dir.list(f);
```

A utilização de `FilenameFilter` é, por exemplo, importante para a criação de classes que vão ser depois usadas por instâncias da classe `FileDialog`, definida no *package* `java.awt` (onde estão implementadas as classes para interacção com o utilizador), já que esta classe apenas manipula implementações concretas de `FilenameFilter`, ou seja, listagens que obedeçam aos filtros especificados.

CLASSE RANDOMACCESSFILE

Esta classe, que é subclassse de `Object`, implementa métodos de leitura e escrita, não necessariamente sequencial, em ficheiros, ou seja, permitindo que a próxima escrita ou leitura de dados em ficheiro possa ser feita numa posição arbitrária deste, posição que é especificada pelo programador através da mensagem `seek(long pos)`, a que se seguirá a desejada operação de `read` ou `write`.

Temos, portanto, implementados nesta classe os métodos que possibilitam um *acesso aleatório* a ficheiro. Os dados a serem escritos e lidos podem ser *bytes*, texto e tipos de dados primitivos de JAVA, mas não objectos. De facto, *ficheiros em disco* podem ser de acesso aleatório mas *streams de dados* não o podem ser.

No entanto, e dado que a indicação da posição onde vai ser realizada a próxima leitura ou a próxima escrita deve ser feita a muito baixo nível, *em particular em termos do número de bytes a partir do início do ficheiro (offset)*, estes ficheiros são de pouca utilidade prática efectiva, podendo, quando muito, servir de base para implementações de mais alto nível, ou para situações em que os registos a gravar (e ler) de ficheiro têm comprimento fixo, situação em que se torna fácil calcular as suas posições relativamente ao início do ficheiro em termos de *número de bytes*.

Por exemplo, se, sendo os dados agrupados em registos, todos os registos tiverem comprimento igual a 125 bytes, o 1° registo poderá ser lido no *offset 0* (cf. `seek(0)`), o 2° registo no *offset 125* (cf. `seek(125)`) e o enésimo, termo geral do cálculo, usando `seek((n-1)*125)`.

A posição actual no ficheiro onde se vai escrever, ou de onde se vai ler, é sempre determinada por um *file pointer* que é um registo que referencia tal posição, e que existe implicitamente associado a este tipo de ficheiros. Todas as mensagens que não sejam de interrogação actualizam o *file pointer*.

Assim, para além dos usuais readXXX() e writeXXX(), em que XXX representa o identificador de um dado tipo primitivo, a classe fornece ainda métodos tais como,

```
getFilePointer()      // devolve posição actual
read()                // lê um byte
read(byte[])          // lê um array de bytes
seek(long pos)        // posiciona-se no offset dado
```

```
skipBytes(int)          // avança dado número de bytes
length()                // total de bytes do ficheiro
```

Devido ao baixo nível dos seus serviços, a classe `RandomAccessFile` é em geral pouco utilizada, sendo mesmo por vezes preterida em relação a ficheiros de acesso sequencial puro através das *streams*.

Esta classe particular de ficheiros oferece uma linguagem comum à de algumas *streams*, dado implementarem as interfaces `DataInput` e `DataOutput`.

A única possível vantagem na utilização deste tipo de ficheiros acontece quando, de facto, os registos são de comprimento fixo, logo de fácil posicionamento como vimos, propriedade a que se junta a vantagem destes ficheiros permitirem, em simultâneo, o acesso para leitura e escrita.

8.4 AS STREAMS DE JAVA

A Figura 8.1 representa toda a hierarquia de classes existente no *package* `java.io`, e as relações destas com algumas interfaces.

Das classes ainda não apresentadas, comecemos por analisar as classes `Writer` e `Reader`. Estas classes abstractas definem o comportamento de um tipo especial de *streams*, que não são *streams de bytes* mas sim *streams de caracteres*. A primeira grande diferença entre umas e outras, é que uma representação de inteiros, *strings* ou outros dados sob a forma de caracteres pode, caso seja feita por exemplo sobre ficheiros, ser visualizável de forma compreensível, ou seja, como texto.

Vamos construir agora uma classe que nos servirá de exemplo para a apresentação das várias *streams* que iremos estudar neste capítulo. Deverá ser, naturalmente, uma classe estruturalmente interessante, por forma a que nos permita escrever, e ler, quer tipos de dados simples quer objectos.

Para nos auxiliar no estudo que vamos realizar sobre as diversas *streams* de JAVA, vamos então criar uma pequena classe, designada `FichaAluno`, cujas instâncias irão representar a ficha individual de um aluno, contendo o seu nome, idade, curso, ano de curso e lista de disciplinas a que está inscrito. Temos, portanto, variáveis para escrever e ler que vão desde os tipos simples aos objectos.

A ficha de aluno não inclui o seu número dado que se pretende reutilizar mais tarde esta classe para criar uma classe `Turma`, na qual a cada número de aluno se irá associar, de forma única, uma ficha destas, o que tornaria redundante a inclusão de tal número na ficha.

A definição da classe `FichaAluno` é, portanto, a seguinte:

```
import java.util.*;
public class FichaAluno {
//
   public FichaAluno (String nome, int idade,
                      int ano, String curso,
                      Vector discp) {
      this.nome = nome; this.idade = idade;
      this.ano = ano; this.curso = curso;
      this.inscricao = discp;
   }

   // variáveis de instância

   private String nome;
   private int idade;
   private String curso;
   private int ano;
   private Vector inscricao;

// métodos de instância

   public String daNome() { return this.nome; }
   public int daIdade() { return this.idade; }
   public String daCurso() { return this.curso; }
   public int daAno() { return this.ano; }
   public Vector daInsc() {
      return (Vector) this.inscricao.clone();
   }

   public Object clone() {
     return new FichaAluno(nome, idade, ano, curso,
                           this.daInsc());
   }
```

```
public String toString() {
  StringBuffer ficha =
      new StringBuffer("--------------------");
  ficha.append('\n');
  ficha.append("Nome: "); ficha.append('\t');
  ficha.append(this.nome); ficha.append('\n');
  ficha.append("Idade: "); ficha.append('\t');
  ficha.append(this.idade); ficha.append('\n');
  ficha.append("Curso: "); ficha.append('\t');
  ficha.append(this.curso); ficha.append('\n');
  ficha.append("Ano: "); ficha.append('\t');
  ficha.append(this.ano); ficha.append('\n');
  int numDiscp = this.inscricao.size();
  ficha.append("Disciplinas: "); ficha.append('\t');
  ficha.append(numDiscp); ficha.append('\n');
  for(int i = 0; i < numDiscp; i++) {
    ficha.append(inscricao.elementAt(i));
    ficha.append('\n');
  }
  return ficha.toString();
}

public String asString() {
  String ficha = "--------------------" + "\n";
  ficha = ficha + "Nome: " + "\t" + this.nome + "\n";
  ficha = ficha + "Idade: " + "\t"+ this.idade + "\n";
  ficha = ficha + "Curso: " + "\t"+ this.curso + "\n";
  ficha = ficha + "Ano: " + "\t"+ this.ano + "\n";
  int numDiscp = this.inscricao.size();
  ficha = ficha + "Disciplinas: " + "\t"+
                                  numDiscp + "\n";
  for(int i = 0; i < numDiscp; i++) {
      ficha = ficha + inscricao.elementAt(i) + "\n";
  }
  return ficha;
 }
}
```

Tendo por base esta classe `FichaAluno`, vamos estudar as várias possibilidades existentes em JAVA para se realizar I/O, procurando analisar quer funcionalidade quer eficiência.

Por funcionalidade disponibilizada por uma dada classe de I/O, deve entender-se a funcionalidade do conjunto dos métodos nesta implementados e, em particular, os meios e os formatos em que os nossos dados podem ser lidos ou escritos.

A eficiência que procuraremos medir relativamente às várias soluções de escrita e de leitura, terá apenas a ver com a eficiência das operações de I/O. No entanto, e com objectivos puramente didácticos, vamos propositadamente apresentar alguns exemplos que visam demonstrar que as análises de eficiência, quando são feitas de forma pouco rigorosa, podem conduzir a resultados bastante enganadores.

Comecemos o nosso estudo sobre as *streams* de JAVA pelas classes que permitem realizar a escrita de dados, em particular por aquelas que permitem escrever dados sob a forma de caracteres, e que são as subclasses da classe `Writer`.

CLASSE WRITER

A classe abstracta `Writer` é a superclasse de todas as *streams de caracteres*, e especifica um conjunto básico de métodos para escrita de caracteres que qualquer implementação de uma *stream de caracteres* deverá oferecer, designadamente, os métodos `write(String s)`, `write(int c)`, `write(char[] b)`, `flush()` e `close()`.

A classe `Writer` é para as *streams de caracteres* equivalente à `OutputStream` para as *streams de bytes*, que estudaremos posteriormente.

As diversas subclasses de `Writer`, `Reader`, `InputStream` e `OutputStream`, possuem designações iniciadas por prefixos tais como, `Print`, `Buffered`, `File`, `Piped`, `CharArray`, `String` e outros, que visam dar uma ideia mais clara do tipo de implementação que foi realizada em tal classe, ou, em certos casos, sobre o tipo de *stream* que deve ser considerada como fonte, ou destino, de certo tipo de operações de I/O.

Há, assim, uma certa distinção entre a *stream lógica* que nos interessa considerar ao nível da programação, ou seja, a classe que possui os métodos que pretendemos utilizar, e a *stream física* que, ao nível do sistema operativo, está de facto a ser por nós lida ou escrita, seja através de caracteres ou de *bytes*.

Por exemplo, em certas circunstâncias, a *stream de caracteres* que deve servir de fonte, ou destino, das nossas operações de leitura, ou escrita, que são realizadas usando os métodos de uma dada *stream*, pode não ser um ficheiro em disco, mas, tão simplesmente, uma dada *string*. Se tal for o caso, então teremos que criar uma instância da *stream lógica* que pretendemos usar, mas associando-a fisicamente à *stream* de JAVA que implementa a manipulação de uma fonte, ou de um destino físico, que é uma *string*, neste caso, uma `StringReader` ou `StringWriter`.

ANINHAMENTO DE STREAMS

Por esta razão, a maioria dos construtores de *streams* apresentam uma declaração "aninhada" que poderia parecer estranha à primeira vista, tal como em,

```
public BufferedWriter(Writer out);
```

que permite criar uma *stream* de escrita de caracteres, com *buffer*, e que tem como parâmetro uma qualquer subclasse de `Writer`, ou seja, pode ser usada sobre um qualquer meio representado por uma subclasse de `Writer`.

Analisando uma vez mais a Figura 8.1, verificamos que, deste modo, poderemos ter operações de *output* com *buffer* sobre ficheiros em disco, por exemplo, desde que criemos instâncias da classe `BufferedWriter` sobre uma qualquer instância de `FileWriter`, declarando, por exemplo:

```
BufferedWriter bout =
   new BufferedWriter(new FileWriter("fich1.dat"));
```

O que se consegue com este tipo de aninhamento é, antes de mais, a possibilidade de, ao nível da programação, usarmos um protocolo ou API de alto nível, que nos permite abstrair de um grande número de detalhes concretos de implementação.

No exemplo apresentado, a utilização de uma *stream* intermediária que usa um mecanismo de *buffering* para a escrita de caracteres num dado destino, torna-se, para quem usa os métodos, um processo quase completamente abstracto, dado não ser necessário conhecer quaisquer detalhes, e apenas possuir alguma compreensão do mecanismo, em particular sabendo que tal mecanismo torna as escritas numa qualquer fonte mais eficiente.

Por outro lado, a comunicação entre a classe que implementa a escrita com ***buffer*** e a classe que implementa a escrita física no objecto destino, ou seja, o processo de tradução das mensagens que a classe de alto nível `BufferedWriter` recebe para as mensagens correspondentes que deve enviar à instância efectivamente criada (no exemplo apresentado uma `FileWriter`, mas poderia ser uma qualquer subclasse de `Writer`), é, mais uma vez, um processo completamente abstracto para quem programa, dado ser de tratado de modo automático pelas classes implementadas.

A figura seguinte procura ilustrar, em concreto, esta situação de aninhamento, ou associação, entre *streams* lógicas e físicas:

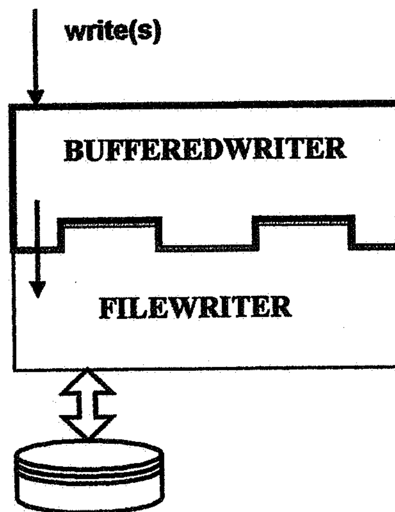


Fig. 8.2 – Associação entre *streams* compatíveis

O programador criou uma instância de `BufferedWriter` que foi aberta sobre uma instância de `FileWriter`. O programador envia as mensagens à instância de `BufferedWriter`, e esta serve de interface automática para a manipulação da `FileWriter` que lhe foi associada.

Precisamos agora de conhecer, ainda que de forma sintética, as propriedades características de cada uma destas subclasses, por forma a que, em cada situação, possamos saber escolher a que mais nos convém.

Note-se, desde já, que esta escolha terá sempre a ver não só com o tipo de dados que pretendemos ler ou escrever, como também com o formato em que os mesmos devem ser lidos ou escritos, em particular texto ou *bytes*, e ainda com a possível necessidade de utilização de certos mecanismos de optimização do I/O, tal como, no exemplo anterior, ser importante usar um *buffer* para optimização, ou, noutras circunstâncias, pretendermos usar *pipes* para comunicação entre processos, ou até mecanismos de leitura que facilitem a implementação de *parsers*.

Por outro lado, as operações de escrita estarão sempre muito relacionadas com as operações de leitura, pelo que o binómio leitura/escrita deverá estar sempre presente quando escolhemos um dado tipo e formato de escrita. Não interessa, em geral, escolher uma forma de *output* que se torne muito simples mas que introduza grande complexidade na leitura dos mesmos dados.

Em JAVA, as *streams* de *bytes* e de caracteres são, praticamente, simétricas, pelo que se torna fácil, apesar do seu número, e conhecidas as principais *streams*, compreender e usar tal simetria.

Por exemplo, se anteriormente se utilizou uma `BufferedWriter` para escrever de forma optimizada sequências de caracteres numa `FileWriter`, torna-se evidente que, ao realizarmos a leitura dos dados a partir do mesmo ficheiro, se deva usar uma instância de `BufferedReader` sobre a `FileReader` que acede ao ficheiro.

Vejamos agora as principais *streams* de caracteres de JAVA e suas características:

Classe BufferedWriter

Mecanismo de comunicação : *buffer de caracteres com 8Kbytes*
Destino dos dados : *uma stream de caracteres subclasse de Writer*
Estruturação das escritas: *inteiros, strings e arrays de caracteres*
Utilização : A classe `BufferedWriter` realiza escritas optimizadas sobre *streams de caracteres*, através da implementação de um mecanismo de *buffering*, ou seja, realizando as pequenas operações de escrita sobre um *buffer*, e apenas realizando a escrita efectiva no destino onde se pretende escrever quando este *buffer* atinge o máximo da sua capacidade, que é definível pelo programador.

Deste modo, são optimizados os acessos desnecessários e demorados à *stream* destino, que pode ser, por exemplo, um ficheiro em disco, uma *string* ou um *array* de caracteres. Para além de implementar os métodos abstractos de `Writer`, esta classe implementa também o método `newLine()`, mas de forma independente do sistema operativo.

Uma instância de `BufferedWriter` é criada, tal como está definido no seu construtor, sobre uma qualquer *stream* de caracteres, ou seja, sobre uma qualquer subclasse da classe `Writer`, tal como definido em:

```
public BufferedWriter(Writer out);
```

Assim, sempre que se pretenda optimizar alguma subclasse de `Writer` que possua operações de escrita pouco eficientes, esta associação deve ser criada. São casos muito comuns a utilização de instâncias de `BufferedWriter` com as, em geral, pouco eficientes `FileWriter`, por exemplo, realizando a seguinte associação:

```
BufferedWriter inp1 =
    new BufferedWriter(new FileWriter("fich1.dat"));
```

Procurando, de forma prática, demonstrar o sentido de tais considerações, vamos agora usar a tal classe exemplo `FichaAluno`, construindo um programa que crie e grave, usando uma instância de `BufferedWriter` sobre uma `FileWriter`, um significativo número de instâncias de tal classe.

Dado que uma `BufferedWriter` não oferece métodos específicos para gravar dados de tipos simples nem objectos, mas apenas *strings* e caracteres, somos assim obrigados a usar intensivamente o método `asString()` de `FichaAluno`, que gera uma *string* já com os usuais separadores de campos (cf. *tab* e *newline*) com toda a representação da ficha, *string* esta que é depois gravada usando o método `write(String s)` de `BufferedWriter`.

Para que este exemplo e todos os exemplos seguintes sejam significativos, serão sempre criadas pelos diferentes programas em análise 300 mil instâncias diferentes de fichas de alunos, ainda que sejam todas iguais em valor, o que é, para a análise em questão, irrelevante, procurando-se assim realizar algumas comparações de "performance" relativas aos vários tipos de *streams*. Trata-se, assim, de um exemplo razoável, já que escreve, apesar de tudo, cerca de 30Mbytes de dados.

Vamos, então, apresentar o código que vai criar em memória e gravar, sob a forma de caracteres, num ficheiro em disco, as 300 mil instâncias da classe `FichaAluno`, usando uma `BufferedWriter` que comunica com uma `FileWriter`.

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

 public class TstFichaAluno1 {
  //
  public static void main(String args[]) {

    // constante

    final int MAXFICHAS = 300000;   // fichas a gravar

    // Disciplinas

    Vector disc = new Vector();
    disc.insertElementAt(new String("PPIV"), 0);
    disc.insertElementAt(new String("MPII"), 1);
    disc.insertElementAt(new String("PPIII"), 2);
    disc.insertElementAt(new String("ACI"), 3);
    disc.insertElementAt(new String("MPI"), 4);

    // Cria um array com MAXFICHAS fichas de igual valor,
    // mas que são diferentes objectos (cf. clone()).

    FichaAluno a1 =
          new FichaAluno("LUIS VAZ", 21, 2, "LESI", disc);
    FichaAluno[] turma = new FichaAluno[MAXFICHAS];

    for(int i=0; i < MAXFICHAS-1; i++) {
       turma[i] = (FichaAluno) a1.clone();
    }

    // Grava as MAXFICHAS fichas numa FileWriter via uma
    // BufferedWriter

    String ficha;
    GregorianCalendar c = new GregorianCalendar();
    Date d = c.getTime();
    System.out.println(d); // inicio da gravação
```

```
    try {
      BufferedWriter bout =
        new BufferedWriter(new FileWriter("fich1.dat"));
      for(int i=0; i < MAXFICHAS-1; i++) {
        ficha = turma[i].asString();
        bout.write(ficha);   // escreve a string
      }
      bout.flush(); bout.close();
    }
    catch(IOException e)
       { System.out.println(e.getMessage()); }

    c = new GregorianCalendar();
    d = c.getTime();
    System.out.println(d);   // fim da gravação
  }
}
```

O ficheiro resultante da execução deste código é um ficheiro de caracteres com cerca de 30Mbytes, que pode ser editado e consultado, e cujo conteúdo, tal como programado no método `asString()` que converte cada ficha de aluno numa *string* que é posteriormente gravada, tem o seguinte aspecto (tendo em atenção que todas as fichas são iguais, por razões de simplificação do código, e das quais apenas se apresentam algumas das milhares gravadas em tal ficheiro):

```
---------------------
Nome: LUIS VAZ
Idade: 21
Curso: LESI
Ano: 2
Disciplinas:  5
PPIV
MPII
PPIII
ACI
MPI
---------------------
Nome: LUIS VAZ
Idade: 21
Curso: LESI
Ano: 2
Disciplinas:  5
PPIV
............
```

O envio da mensagem `getTime()` às instâncias da classe `GregorianCalendar`, programados no início e no fim da gravação, permitirão ter uma ideia do intervalo de tempo necessário à gravação das 300 mil fichas.

No exemplo em análise, teríamos obtido um tempo de cerca de 130s.

Experimentemos, agora, não realizar a ***clonagem*** das fichas mas criar uma cópia que é armazenada de imediato no array `turma`. Para tal, foi escrito na classe de teste o seguinte código, em substituição do anterior:

```
for(int i=0; i < MAXFICHAS-1; i++) {
   turma[i] = new FichaAluno("LUIS VAZ", i, 2, "LESI", disc);
}
```

que, sem dúvida, cria 300000 diferentes fichas de aluno, indexando, ficticiamente claro, a idade de cada um à sua posição no *array* `turma`.

Esta solução apresenta um tempo de gravação de 138 segundos, muito semelhante, portanto, ao primeiro tempo que foi conseguido, em condições de réplica pura de `a1` usando `clone()`, o que não pode surpreender tendo em atenção o idêntico número de linhas e de caracteres que foram gravados.

Ainda que os métodos `asString()` e `toString()` da classe `FichaAluno` sejam em tudo semelhantes, pois criam uma *string* contendo toda a ficha com os campos separados por *tab* e *newline*, vamos agora usar, em vez de `asString()` o método `toString()`.

O código seria o mesmo, introduzindo-se apenas a transformação sublinhada:

```
ficha = turma[i].toString();
bout.write(ficha);   // escreve a string
```

A gravação das 300 mil fichas passou a ser realizada no, comparativamente, surpreendente tempo médio de cerca de **42 segundos**.

Como se pode verificar agora, as primeira soluções têm a si associadas um factor de ineficiência de cerca de 3 vezes, que não pode ser imputável ao I/O, mas antes ao método `asString()`.

De facto, o código do método `asString()` usa o operador de concatenação para criar a *string* resultado que vai ser escrita em ficheiro. As *strings* resultantes de um ou de outro método são exactamente iguais, possuem a mesma dimensão, têm os mesmos separadores e, como poderemos verificar pelos ficheiros finais, produzem ficheiros com a mesma dimensão e estrutura.

No entanto, o método `asString()` trabalha directamente com *strings*, fazendo a sua concatenação. As *strings* de JAVA são, por definição, *imutáveis*, ou seja, não são alteráveis no seu conteúdo original. Assim, quando escrevemos uma operação de concatenação, tal como no código do método `asString()`, sob a forma,

```
ficha = ficha + "Nome: " + "\t" + this.daNome() + "\n"
```

este código é muito ineficiente porque de cada subconcatenação resultará uma nova *string* a ser temporariamente criada e, no final, uma nova *string* instância deverá igualmente ser atribuída à variável `ficha`.

Como deveremos então fazer sempre que pretendermos ter *strings* mutáveis, ou seja, armazenar caracteres, modificá-los, retirar *substrings*, etc. A solução consiste em criar uma instância da classe `StringBuffer`, classe que oferece um conjunto de métodos próprios para realizar tais operações, por exemplo, `append()` que permite juntar ao buffer de carateres qualquer valor de qualquer tipo. Para que no final possamos de novo ter um objecto que é uma *string*, basta enviar à instância de `StringBuffer` a mensagem `toString()`, tal como exemplificado no método `toString()` de `FichaAluno`.

Num pequeno programa de tratamento de algumas dezenas de fichas, esta evidente ineficiência talvez nunca fosse detectada. No nosso caso porém, em que temos que realizar tal operação sobre 300 mil fichas, conclui-se que cerca de 66% do tempo total gasto no total de I/O, era consumido por razões de ineficiência que nada têm a ver com o tipo de *streams* que estamos a experimentar. Naturalmente que o método `asString()` da classe `FichaAluno` apenas serviu para que se pudesse apresentar este exemplo e, portanto, não vai ser mais usado.

Tal como havíamos afirmado anteriormente, é necessário ter em atenção quando se realizam testes de eficiência, que devemos sempre avaliar até que ponto factores externos ao parâmetro em observação podem afectar os resultados obtidos. De uma forma geral, tal passa por se considerarem e testarem diferentes alternativas.

Temos assim analisada a primeira associação entre *streams* de caracteres, na qual uma `BufferedWriter` foi usada como controladora de uma `FileWriter`.

A subclasse `FileWriter`, sobre a qual se utilizou uma `BufferedWriter`, é apresentada a seguir.

Classe FileWriter

Mecanismo de comunicação : *standard*
Destino dos dados : *ficheiro de caracteres*
Estruturação das escritas : herda de `OutputStreamWriter` métodos para escrever *caracteres, strings e arrays de caracteres num ficheiro*
Utilização : Deve ser associada a uma `BufferedWriter` para maior eficiência.

Se no programa anterior alterarmos o código de I/O para,

```
for(int i=0; i < MAXFICHAS-1; i++) {
   turma[i] = (FichaAluno) a1.clone();
}
try {
    FileWriter fout = new FileWriter("fich2.dat");
    for(int i=0; i < MAXFICHAS-1; i++) {
      ficha = turma[i].toString();
      fout.write(ficha);  // escreve a string
    }
    fout.flush(); fout.close();
}
catch(IOException e)
  { System.out.println(e.getMessage()); }
```

ou seja, recorrendo directamente a uma `FileWriter` aberta sobre um ficheiro, e, portanto, não usando o *buffering* oferecido pela classe `BufferedWriter`, pode notar-se, de facto, uma ligeira degradação nos tempos de execução, que se tornarão próximos dos 60 segundos.

As classes de escrita até agora utilizadas, dado não possuirem funcionalidade para escrever tipos mais estruturados que caracteres e *strings*, obrigaram-nos a usar o

método `toString()` para converter cada instância de `FichaAluno` numa *string*, para que esta fosse gravada usando o método `write()`.

Vamos, em seguida, analisar a classe `PrintWriter` que já implementa métodos para a criação de representações textuais quer de tipos primitivos quer de objectos, sendo por isso uma classe muito utilizada para escrita em *streams* de caracteres.

As instâncias de `PrintWriter` podem ser também criadas sobre uma qualquer `OutputStream`, em cujo caso os caracteres são convertidos para *bytes* e, assim, armazenados. A classe `PrintWriter` é a substituição da classe `PrintStream`, existente em versões anteriores de JAVA e, entretanto, desactivada (*deprecated*).

Classe PrintWriter

Mecanismo de comunicação : *standard*
Destino dos dados : *stream de bytes ou stream de caracteres (cf. construtor)*
Estruturação das escritas : *tipos simples e objectos escritos como caracteres*
Utilização : Sobre uma subclasse de `Writer` ou sobre uma subclasse de `OutputStream`. Substitui a classe `PrintStream` que na versão 1.2 se tornou caduca (*deprecated* na terminologia JAVA). A classe oferece para além dos usuais métodos `write()`, `flush()` e `close()`, métodos `print()` e `println()`, que recebem como parâmetro um valor de qualquer tipo simples ou até uma instância de uma dada classe. Para que tal instância possa ser transformada em caracteres, é necessário que a sua classe implemente o método `toString()`, método invocado por `print()` ou `println()` para realizar tal transformação.

Como se verifica, a existência de um método `toString()` definido na classe dos objectos que se pretendem armazenar textualmente, é fundamental para que estes possam ser gravados usando uma `PrintWriter`. Foi também esta a abordagem por nós seguida anteriormente para podermos gravar fichas de alunos sob a forma de caracteres, usando, até aqui, classes que não o fazem directamente.

Vamos reescrever parte do código do programa anterior para que seja agora usada uma instância de `PrintWriter` para a escrita das fichas dos alunos numa `FileWriter`, continuando a usar o método `toString()` para obter a *string* que

se pretende gravar e, assim, invocando apenas o método `print(String s)` da classe `PrintWriter`, para além dos métodos `flush()` e `close()` que temos sempre vindo a usar e que são muito importantes para uma correcta execução destes programas de escrita. Teríamos então o código seguinte,

```
try {
    PrintWriter fout =
      new PrintWriter(new FileWriter("fich3.dat"));
    for(int i=0; i < MAXFICHAS-1; i++) {
     ficha = turma[i].toString();
     fout.print(ficha);
    }
    fout.flush(); fout.close();
   }
 catch(IOException e)
   { System.out.println(e.getMessage()); }
```

O ficheiro criado, *fich3.dat*, possui a mesma estrutura que todos os anteriores, e pode ser consultado usando um qualquer editor de texto. Saliente-se ainda o facto de que, apesar de se ter usado uma `FileWriter` para a escrita dos caracteres, o tempo de execução da gravação dos 300000 registos, que correspondem à criação de um ficheiro de 30 Mbytes, manteve-se na casa dos 60 segundos, tal como para a anterior implementação sem buffer.

Note-se, no entanto, que a maioria do trabalho está a ser realizado pelo método `toString()` de `FichaAluno`, que prepara uma *string* já com *new lines* para ser passada como parâmetro aos métodos `write()`, antes, e `print()` agora.

Torna-se, portanto, importante verificar se obteríamos uma maior eficiência caso tivéssemos optado por usar o método `print()` de `PrintWriter` passando-lhe não a *string* completa mas cada um dos campos da ficha de aluno.

O bloco fundamental de código da classe de teste passaria a ser:

```
try {
    Vector discp;
    PrintWriter fout =
      new PrintWriter(new FileWriter("fich31.dat"));
    for(int i=0; i < NUMFICHAS-1; i++) {
      ficha = turma[i];
```

```
        fout.println("--------------------");
        fout.print("Nome: ");
        fout.println(ficha.daNome());
        fout.print("Idade: ");
        fout.println(ficha.daIdade());
        fout.print("Curso: ");
        fout.println(ficha.daCurso());
        fout.print("Ano: ");
        fout.println(ficha.daAno());
        discp = ficha.daInsc();
        fout.print("Disciplinas:");
        fout.println(discp.size());
        for(int dc=0; dc < discp.size()-1; dc++) {
           fout.println(discp.elementAt(dc));
        }
     }
     fout.flush(); fout.close();
  }
catch(IOException e) {System.out.println(e.getMessage());}
```

Este código, aparentemente menos eficiente que o do método `toString()`, dado ser mais detalhado porque invoca cada um dos selectores de `FichaAluno` e só depois os imprime na `FileWriter`, revelou-se, conforme esperado, bastante menos eficiente que as restantes soluções, executando em cerca de 80 segundos.

Assim, efectivamente, a utilização de uma `PrintWriter` associada a um método de instância que converta as instâncias em *strings* para que depois estas possam ser escritas em *streams* de caracteres, parece ser um processo bastante mais eficiente que o simples acesso aos valores de tais campos e à sua imediata escrita, neste caso sobre uma `FileWriter`.

No entanto, a formatação realizada foi equivalente a todas as anteriores, pelo que o ficheiro final de dados surge igualmente estruturado e legível através de um simples editor de texto, e com o formato anteriormente apresentado.

Os implementadores de JAVA aconselham que as associações entre instâncias de `PrintWriter` e `FileWriter` sejam realizadas usando, uma `BufferedWriter` como classe intermediária para optimização, dada a pouca eficiência da sua combinação directa.

Esta sugestão conduz a um aninhamento de *streams* tal como o representado na Figura 8.3, e que iremos testar com dois exemplos.

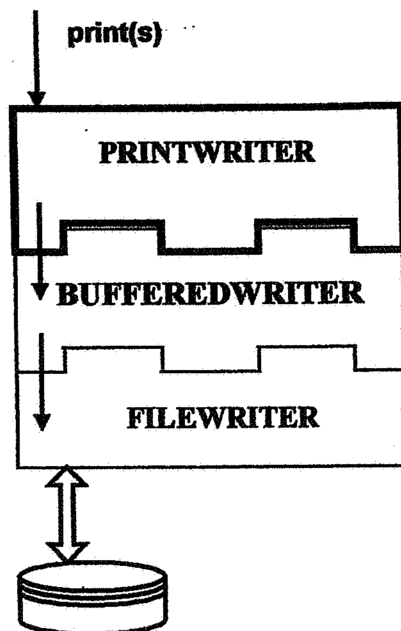


Fig. 8.3 – `BufferedWriter` como classe intermédia de optimização

Neste primeiro exemplo continuaremos a usar o método `toString()` para obter a *string* a gravar. O correspondente novo código de I/O será:

```
try {
      PrintWriter fout =
         new PrintWriter(new BufferedWriter (
                              new FileWriter("fich32.dat")));
      for(int i=0; i < MAXFICHAS-1; i++) {
        ficha = turma[i].toString();
        fout.print(ficha);
      }
      fout.flush(); fout.close();
     }
catch(IOException e) {System.out.println(e.getMessage());}
```

A média dos tempos das execuções deste código resultou num tempo de gravação de cerca de 42 segundos, ou seja, um dos melhores tempos de todas as soluções até agora testadas.

Foi, em seguida, testada a última hipótese possível, ou seja, a gravação por campos da ficha de aluno, usando a mesma associação, cujo código é, portanto:

```
try {
    Vector discp;
     PrintWriter fout =
        new PrintWriter(new BufferedWriter (
                    new FileWriter("fich33.dat")));
    for(int i=0; i < NUMFICHAS-1; i++) {
      ficha = turma[i];
      fout.println("--------------------");
      fout.print("Nome: ");
      fout.println(ficha.daNome());
      fout.print("Idade: ");
      fout.println(ficha.daIdade());
      fout.print("Curso: ");
      fout.println(ficha.daCurso());
      fout.print("Ano: ");
      fout.println(ficha.daAno());
      discp = ficha.daInsc();
      fout.print("Disciplinas:");
      fout.println(discp.size());
      for(int dc=0; dc < discp.size()-1; dc++) {
         fout.println(discp.elementAt(dc));
      }
   }
   fout.flush(); fout.close();
  }
  catch(IOException e)
   { System.out.println(e.getMessage()); }
```

Esta solução permite gravar as 300000 fichas de alunos num ficheiro de caracteres, que é, portanto, consultável e editável, em 29 segundos. Esta é, portanto, a melhor solução encontrada até ao momento. A eficiência desta solução é surpreendente e inesperada, não só porque usámos 3 diferentes tipos de *streams*, mas também pelo facto de usarmos, individualmente, os campos da ficha de aluno e não a *string* que resulta do envio da mensagem `toString()`.

Caso tivéssemos evitado gravar caracteres de auxílio à formatação dos dados para visualização, ou seja, se tivéssemos codificado o ciclo de gravação apenas como:

```
for(int i=0; i < NUMFICHAS-1; i++) {
  ficha = turma[i];
  fout.println(ficha.daNome());
  fout.println(ficha.daIdade());
  fout.println(ficha.daCurso());
  fout.println(ficha.daAno());
  discp = ficha.daInsc();
  fout.println(discp.size());
  for(int dc=0; dc < discp.size()-1; dc++) {
     fout.println(discp.elementAt(dc));
  }
}
```

o tempo de gravação das 300 mil fichas baixaria para cerca de 20 segundos.

O que, de facto, se torna indiscutível, a partir dos exemplos apresentados, é que há diferenças de eficiência consideráveis no que diz respeito à escrita de dados e, em circunstâncias idênticas, entre a utilização do método `toString()` para gravação de uma simples *string* e gravações mais atomizadas baseadas em escritas campo a campo.

Vamos agora analisar o que se passa caso seja criada uma `PrintWriter` sobre uma `FileOutputStream`, situação em que caracteres devem ser convertidos em *bytes*, ainda que o ficheiro resultante seja igualmente consultável através de um editor de texto. O código de associação entre as duas *streams* seria neste caso:

```
try {
    PrintWriter fout = new PrintWriter(
                new FileOutputStream("fich4.dat"));
    for(int i=0; i < MAXFICHAS-1; i++) {
     ficha = turma[i].toString();
     fout.print(ficha);
    }
    fout.flush(); fout.close();
  }
catch(IOException e)
  { System.out.println(e.getMessage()); }
```

O tempo de execução deste código mantém-se, na máquina de referência, à volta dos 42 segundos, e o ficheiro criado mantém a estrutura de todos os anteriores.

Vamos, em seguida, atomizar mais uma vez as escritas, ou seja, não usar o método `toString()` sobre cada ficha de aluno, mas aceder e escrever individualmente cada um dos campos. O código final seria, portanto:

```
try {
    Vector discp;
     PrintWriter fout =
      new PrintWriter(new
                    FileOutputStream("fich41.dat"));
      for(int i=0; i < MAXFICHAS-1; i++) {
        ficha = turma[i];
        fout.println("--------------------");
        fout.print("Nome: ");
        fout.println(ficha.daNome());
        fout.print("Idade: ");
        fout.println(ficha.daIdade());
        fout.print("Curso: ");
        fout.println(ficha.daCurso());
        fout.print("Ano: ");
        fout.println(ficha.daAno());
        discp = ficha.daInsc();
        fout.print("Disciplinas:");
        fout.println(discp.size());
        for(int dc=0; dc < discp.size()-1; dc++) {
           fout.println(discp.elementAt(dc));
        }
      }
      fout.flush(); fout.close();
    }
    catch(IOException e) {
 System.out.println(e.getMessage()); }
```

código que conduz a um resultado de cerca de 29 segundos de tempo de gravação para o mesmo número de fichas.

Retirando, uma vez mais, as linhas de formatação e deixando apenas o código de escrita efectivo, o tempo obtido seria reduzido para 21 segundos. O ficheiro criado é igualmente consultável através de um editor de texto, muito embora cada um dos campos deixe de ser etiquetado.

Como veremos posteriormente, as leituras de ficheiros de caracteres que são muito formatados em prol de uma melhor visualização dos dados, são, temporalmente, bastante menos eficientes que as leituras realizadas sobre ficheiros nos quais apenas foram gravados os dados, dado haver a necessidade de ler um muito maior número de caracteres.

Este é um dos compromissos a atender relativamente a *streams* de caracteres. De facto, melhor visualização implica mais caracteres escritos, e, consequentemente, menor "performance" das operações de leitura.

A tabela seguinte sintetiza os vários resultados obtidos relativamente às soluções apresentadas para o exemplo em questão, no qual 300000 fichas de aluno, com cerca de 100bytes cada uma, devem ser armazenadas em ficheiro, num forma consultável, ou seja, como caracteres, tendo todas as soluções gerado ficheiros de igual dimensão e igual estrutura:

***Stream* usada**	***Stream* de Interface**	***Tempo médio de gravação***
BufferedWriter	FileWriter (c/ toString())	42 s
FileWriter	------------------	60 s
PrintWriter	FileWriter (c/ toString())	60 s
PrintWriter	FileWriter (por campos)	80 s
PrintWriter	BufferedWriter FileWriter (c/ toString())	42 s
PrintWriter	BufferedWriter FileWriter (por campos)	29 s
PrintWriter	FileOutputStream (c/ toString())	42 s
PrintWriter	FileOutputStream (por campos)	29 s

A primeira grande evidência é que as soluções sobre `FileWriter` apenas são boas se for usada uma `BufferedWriter` como classe associada, seja de forma directa, seja como *stream* de interface. A utilização de `PrintWriter` é cómoda, pois implementa métodos para gravar tipos primitivos, e é, em geral, eficiente. No entanto, uma `PrintWriter` torna-se mais eficiente sempre que a escrita é mais

atomizada, e não quando a escrita se resume a escrever uma *string* resultante de `toString()`. Sobre uma `FileOutputStream` uma `PrintWriter` é sempre muito eficiente.

Apesar de se ter procurado que as variáveis de instância de `FichaAluno` fossem de vários tipos, designadamente, simples, *strings* e objectos, a tabela anterior deve sempre ser encarada com a subjectividade relativa ao caso de teste. Ainda assim, foi possível confirmar a correcção de alguns conselhos dos implementadores de JAVA. Num projecto concreto, nada como realizar alguns testes tais como os que neste capítulo foram apresentados para as operações de escrita.

Vejamos, em síntese, as restantes classes que implementam `Writer`, mas que são demasiado específicas para que delas, de momento, possamos apresentar alguns exemplos de real interesse:

Classe PipedWriter

Mecanismo de comunicação : *permite implementar pipes*
Destino dos dados : *streams de caracteres*
Estruturação das escritas : *caracteres, array de caracteres*
Utilização : Em conjunto com `PipedReader` e, idealmente, usando *threads*, permite implementar comunicação via *pipes*.

Classe StringWriter

Mecanismo de comunicação : *usa uma instância de* `StringBuffer`
Destino dos dados : *uma string interna*
Estruturação das escritas : *caracteres, array de caracteres*
Utilização : Realizar operações típicas sobre *streams* tendo como destino dos caracteres uma `StringBuffer` interna à instância de `StringWriter`. O estado da *stream* pode ser inspeccionado a qualquer momento, recorrendo ao método `toString()` que esta classe implementa.

Classe `CharArrayWriter`

Mecanismo de comunicação : *standard*
Destino dos dados : *um array de caracteres interno*
Estruturação das escritas : *caracteres, array de caracteres*
Utilização : Realizar operações típicas sobre *streams* tendo como destino dos caracteres um *array* interno de caracteres. O estado desta *stream* pode ser a qualquer momento inspeccionado recorrendo aos métodos `toString()` e `toCharArray()` que esta classe implementa.

Classe `OutputStreamWriter`

Mecanismo de comunicação : *standard*
Destino dos dados : *embora sejam caracteres destinam-se a uma stream de bytes*
Estruturação das escritas : *caracteres traduzidos para bytes*
Utilização : Para realizar a adaptação entre *streams* de caracteres e *streams* de bytes. Deve ser associada a uma `BufferedWriter` para maior eficiência. Note-se que a classe `FileWriter`, já utilizada, é uma subclasse desta.

Vamos, em seguida, analisar as classe simétricas de `Writer`, ou seja, as subclasses da classe abstracta `Reader`, que especifica o comportamento para leitura de dados armazenados em *streams* de caracteres.

CLASSE READER

Trata-se de uma classe abstracta que define um conjunto de métodos para leitura de *streams de caracteres*, designadamente, `int read()` e `int read(char[] cbuf)`, que fazem a leitura de um caracter ou de um *array* de caracteres, e que devolvem o valor –1 ao ser atingido o fim da *stream* origem. A classe define ainda os métodos `close()`, `mark()` e `ready()` para controlo das leituras.

Esta classe é simétrica da classe `InputStream` para *streams* de *bytes*, e as suas subclasses são as simétricas das subclasses de `Writer` anteriormente apresentadas, possuindo até as mesmas propriedades. São, portanto, subclasses de `Reader` as

classes seguintes: `BufferedReader`, `CharArrayReader`, `StringReader`, `PipedReader`, `FileReader` e `InputStreamReader`.

Vamos agora procurar realizar a leitura de cada uma das *streams* até aqui criadas para armazenamento das fichas de aluno, usando as classes de *input* disponíveis, e que são as várias subclasses de `Reader` apresentadas.

Note-se que, neste contexto, por *leitura de dados* se deve entender não apenas a leitura dos valores para uma simples visualização, mas a efectiva reconstituição das instâncias que foram gravadas, o que é um problema bastante diferente, já que, em geral, em ficheiros de caracteres há a tendência para adicionar aos dados efectivos caracteres adicionais de formatação que devem ser posteriormente desprezados.

Comecemos por ler o ficheiro *fich1.dat* onde, usando uma `BufferedWriter` sobre uma `FileWriter` gravámos as fichas de alunos, escritas sob a forma de diferentes linhas de texto e com os campos separados por *tabs*.

Vamos fazer a leitura usando as classes simétricas, ou seja, criando agora uma `BufferedReader` sobre uma `FileReader`. A classe `FileReader` permite que sejam lidas, usando o método `readLine()`, sucessivas linhas de texto do ficheiro original. Porém, no nosso caso, gravámos estas linhas não apenas com dados mas também com texto adicional de formatação, designadamente, separadores e texto que identifica cada um dos valores. Ainda que tal formatação auxilie à visualização do ficheiro de texto, agora, que apenas pretendíamos ler os dados efectivos, temos que ter em atenção o texto adicional que deve ser lido mas desprezado.

A classe de JAVA designada `StringTokenizer`, pode, neste caso, dar-nos uma ajuda preciosa, já que possui métodos adequados à extracção de *tokens* (palavras) a partir de uma *string*, palavras (*tokens*) essas reconhecidas desde que estejam na *string* separadas entre si por espaços, *tabs* ou *newlines*. Estes são os delimitadores por omissão. A classe possui ainda construtores que permitem especificar outros caracteres como delimitadores. No nosso caso, pretendemos apenas encontrar os *tabs* já que foi assim que separámos os campos.

Quando se armazena uma *string* numa instância de `StringTokenizer`, a *string* parâmetro é vista como constituída por uma sequência de *tokens* (palavras), desde que entre estes existam os referidos separadores. Então, o método `nextToken()`

permite extrair, a cada momento, o *token* seguinte. Outros métodos existem que implementam operações adicionais, tais como os métodos, `hasMoreTokens()`, `countTokens()`, etc.

No exemplo, necessitamos apenas de ler uma linha de cada vez a partir do ficheiro usando o método `readLine()`, criar uma instância de `StringTokenizer` sobre essa linha, e, usando `nextToken()`, extrair os dados relevantes, desprezando as palavras que apenas serviram de auxiliares de formatação.

Adicionalmente, caso os dados devam ser convertidos de *string* para, por exemplo, valores inteiros, é usado o código de conversão que foi já anteriormente analisado (ver classe `Consola`).

Apresenta-se a porção de código que diz respeito à leitura e reconstrução do *array* de instâncias de `FichaAluno`.

```
FichaAluno[] turma = new FichaAluno[MAXFICHAS];
FichaAluno ficha;

//  Lê as MAXFICHAS de uma FileReader via uma
//  BufferedReader

try
 {
  String linha;
  StringTokenizer st;
  String txt, nome, curso;
  int ano, idade, discp;
  BufferedReader bin =
    new BufferedReader(new FileReader("fich1.dat"));
    for(int i=0; i < MAXFICHAS-1; i++) {
     // le uma ficha de aluno
     linha = bin.readLine(); //despreza separador
     // NOME: .................
     st = new StringTokenizer(bin.readLine(), "\t");
     txt = st.nextToken();
     nome = st.nextToken();
     // Idade: ...............
     st = new StringTokenizer(bin.readLine(), "\t");
     txt = st.nextToken();
     idade = Integer.valueOf(st.nextToken()).intValue();
     // Curso: ...............
```

```
      st = new StringTokenizer(bin.readLine(), "\t");
      txt = st.nextToken();
      curso = st.nextToken();
      // Ano: ..................
      st = new StringTokenizer(bin.readLine(), "\t");
      txt = st.nextToken();
      ano = Integer.valueOf(st.nextToken()).intValue();
      // Disciplinas: ..........
      st = new StringTokenizer(bin.readLine(), "\t");
      txt = st.nextToken();
      discp = Integer.valueOf(st.nextToken()).intValue();
      // ciclo para as disciplinas
      Vector disciplinas = new Vector();
      for(int ds=0; ds < discp; ds++) {
        st = new StringTokenizer(bin.readLine());
        txt = st.nextToken();
        disciplinas.insertElementAt(txt, ds);
      }
      ficha = new FichaAluno(nome, idade, ano, curso,
                             disciplinas);
      turma[i] = ficha;
    }
    bin.close();
  }
  catch(IOException e) {System.out.println(e.getMessage();}
```

Poderíamos, tal como fizemos para criar o ficheiro *fich2.dat*, usar directamente uma `FileReader` para ler tais dados, no entanto o código seria exactamente igual dado que deveríamos usar o método `readLine()` para ler as linhas do ficheiro, e uma `StringTokenizer` para delas extrair os *tokens*.

Dado não existir uma classe `PrintReader`, simétrica de `PrintWriter`, não faz sentido criar a leitura inversa à escrita.

No exemplo seguinte, apresenta-se um extracto de um programa que permite ler linhas de texto de um ficheiro de caracteres anteriormente criado. Dado não ser conhecido o número exacto de linhas do ficheiro, o ciclo principal do programa é construído testando, sucessivamente, se o ficheiro de onde estamos a ler as linhas já se esgotou, usando a construção `bin.ready()`, comum às classes `Reader`.

O código principal é o seguinte:

```
//   Lê linhas de texto de uma FileReader via uma
//   BufferedReader e conta-as.

     int nlinhas = 0;
     try {
         String linha;
         BufferedReader bin =
                 new BufferedReader(new
                              FileReader("fich2.dat"));
         while(bin.ready()) {
            linha = bin.readLine();
            nlinhas++
         }
    }
    catch(IOException e)
    { System.out.println(e.getMessage()); }
```

Este código lê, e faz a contagem, das 3.299.989 linhas do ficheiro `fich2.dat` anteriormente criado, em cerca de 44 segundos.

São igualmente subclasses de `Reader` as seguintes classes:

Classe InputStreamReader

Mecanismo de comunicação : *standard*
Origem dos dados : *streams de bytes*
Estruturação das leituras : *bytes* traduzidos para *caracteres.*
Utilização : Serve para realizar a adaptação entre *streams de bytes* e *streams de caracteres.* Deve ser associada a uma `BufferedReader` para maior eficiência. A classe `FileReader`, usada no exemplo anterior, é uma subclasse desta.

Classe FileReader

Mecanismo de comunicação : *standard*
Origem dos dados : *ficheiros de bytes*
Estruturação das leituras : *bytes* traduzidos para *caracteres.*
Utilização : Deve ser associada a uma `BufferedReader` para maior eficiência.

Classe PipedReader

Mecanismo de comunicação : *permite implementar pipes*
Origem dos dados : *streams de caracteres*
Estruturação das leituras : *caracteres, array de caracteres*
Utilização : Em conjunto com `PipedWriter`, e idealmente usando *threads*, permite implementar comunicação via pipes.

Classe StringReader

Mecanismo de comunicação : *standard*
Origem dos dados : *uma string*
Estruturação das leituras : *caracteres, array de caracteres*
Utilização : Realizar operações típicas sobre *streams* tendo como fonte dos caracteres uma *string*.

Classe CharArrayReader

Mecanismo de comunicação : *standard*
Origem dos dados : *um array de caracteres*
Estruturação das leituras : *caracteres, array de caracteres*
Utilização : Realizar operações típicas sobre *streams* tendo como fonte dos caracteres um *array* de caracteres.

Analisada a questão de armazenar em *streams de caracteres* quer tipos de dados simples quer objectos, estes após serem convertidos em *strings* usando o método `toString()`, é chegada a altura de analisar se a utilização de *streams de bytes* é mais eficiente que a utilização de *streams de caracteres*.

Vamos pois estudar as `Output` e `Input` *streams de bytes*, e verificar até que ponto, ainda que perdendo, possivelmente, a possibilidade de uma visualização dos dados que foram gravados, estas podem ser mais eficientes e mais fáceis de usar que as *streams* que usam caracteres.

CLASSES OUTPUTSTREAM E INPUTSTREAM

Qualquer implementação da classe abstracta `OutputStream` é uma *stream de bytes*. Todas as *streams de bytes* subclasses de `OutputStream`, têm a sua análoga *stream de caracteres* que é subclasse de `Writer`.

Os métodos definidos pela classe abstracta `OutputStream` são idênticos aos da classe `Writer`, com a simples diferença de que, em vez de caracteres, aceitam como parâmetros de `write()` *bytes*, simples ou em *array*.

Como vimos anteriormente a gravação de um número significativo de dados numa `FileOutputStream` é tão eficiente como a mesma gravação feita sobre uma `FileWriter`. No entanto, e em ambos os casos, não gravámos nem lêmos tipos primitivos directamente, sendo sempre necessário realizar as suas conversões para *string* antes de os escrever, e realizar o procedimento inverso na leitura.

As classes que são subclasses de `OutputStream`, tal como as que são subclasses de `Writer`, possuem métodos bastante primitivos para escrita de *bytes*. Porém, há duas subclasses de `OutputStream` que merecem uma atenção especial dadas as características particulares dos seus métodos.

A classe `DataOutputStream` implementa métodos específicos para a gravação em *streams de bytes* de valores dos diversos tipos primitivos de JAVA, tais como, os métodos `writeChar(c)`, `writeByte(b)`, `writeInt(i)`, etc. Assim sendo, torna-se muito conveniente usar uma instância desta classe para escrever dados em qualquer *stream* de saída. Vamos analisar em seguida a utilização de uma *stream* deste tipo sobre uma `FileOutputStream`, ou seja, uma *stream de bytes* que usa um ficheiro em disco.

```
try {
    DataOutputStream dout = new DataOutputStream(
                            new FileOutputStream("fich51.dat"));
       for(int i=0; i < MAXFICHAS-1; i++) {
         ficha = turma[i];
         dout.writeUTF("--------------------");
         dout.writeUTF(ficha.daNome());
         dout.writeInt(ficha.daIdade());
         dout.writeUTF(ficha.daCurso());
         dout.writeInt(ficha.daAno());
```

```
            insc = ficha.daInsc();
            discp = insc.size();
            dout.writeInt(discp);
            for(int ds=0; ds < discp; ds ++) {
              dout.writeUTF((String) insc.elementAt(ds));
            }
         }
         dout.flush(); dout.close();
         }
    catch(IOException e) { System.out.println(e.getMessage()); }
```

Esta solução possui no entanto um custo em termos de eficiência que deve ser tomado em consideração. Enquanto que todas as soluções anteriores, ainda que aparentemente mais exigentes em termos de código a desenvolver, realizaram a gravação das 300000 fichas, no máximo, em cerca de 80 s, serão agora necessários cerca de 270 segundos para criar o ficheiro desejado. Adicionalmente, do ficheiro criado, com cerca de 23Mbytes, note-se alguma redução de espaço, apenas são reconhecíveis através de um editor de texto os campos que foram gravados como caracteres (usando `writeUTF()` que usa o *Unicode Text Format* de JAVA).

Sabemos também, de uma experiência anterior, que o problema não tem a ver com a utilização de uma `FileOutputStream`, que usámos já em associação com uma `PrintWriter`, mas, possivelmente, à pouca eficiência da `DataOutputStream` usada.

Por outro lado, e como se pode verificar pelo conjunto de métodos disponíveis na classe `DataInputStream`, o método `readLine()` é, na actual versão de JAVA, definido como *deprecated* por razões de eficiência, sendo aconselhada a utilização do método `readUTF()` para os campos escritos como caracteres.

O código de leitura seria então:

```
try {
      DataInputStream din =
         new DataInputStream(new
                              FileInputStream("fich51.dat"));
      for(int i=0; i < MAXFICHAS-1; i++) {
         txt = din.readUTF(); // despreza 1ª linha
         nome = din.readUTF();
         idade = din.readInt();
         curso = din.readUTF();
```

```
            ano = din.readInt();
            numdiscp = din.readInt();
            inscricao = new Vector();
            for(int ds=0; ds < numdiscp; ds++) {
                discipl = din.readUTF();
                inscricao.insertElementAt(discipl, ds);
            }
            ficha = new FichaAluno(nome, idade, ano, curso,
                                              inscricao);
            turma[i] = ficha;
        }
        din.close();
    }
    catch(IOException e)
    { System.out.println(e.getMessage()); }
```

A solução de invocar o método `toString()` e gravar a ficha completa usando o método `writeUTF(String s)` seria muito mais ineficiente.

Como solução alternativa, poderíamos criar uma `BufferedReader` sobre a `FileInputReader`. Esta é uma solução já anteriormente apresentada e analisada, pelo que passaremos a analisar uma solução alternativa concedida pelas *streams* de JAVA, e que consiste na gravação directa, em *stream*, de instâncias completas de certas classes.

OUTPUT E INPUT OBJECT STREAMS

Uma `ObjectOutputStream` é uma *stream* especial de JAVA que é capaz de armazenar objectos, *arrays* e valores de tipos simples, usando um método de nome `writeObject()` que implementa um algoritmo particular de *serialização*, isto é, um algoritmo que garante que todas as referências cruzadas existentes entre instâncias de diferentes classes são devidamente repostas aquando do processo de leitura de tais instâncias.

Para que tal seja possível, é apenas necessário que as classes de tais instâncias que se pretendem gravar numa `ObjectOutputStream`, implementem a interface `Serializable` ou a interface `Externalizable`.

Tal como foi dito no capítulo dedicado às *interfaces* de JAVA, uma classe pode ser considerada como `Serializable` se, por homomorfismo, todas as variáveis de instância por esta definidas pertencerem também a classes `Serializable`. Os tipo simples são, por definição, `Serializable`. As instâncias da classe `String` são-no também, bem como os *arrays*. No entanto, as variáveis declaradas com os atributos `static` (de classe) ou `transient` (cujos valores resultam de cálculos intermédios que podem sempre ser refeitos), não são serializadas.

A interface `Serializable` é, tal como outras, uma *marker interface*, ou seja, uma interface que não impõe uma particular implementação, mas apenas a satisfação de uma dada propriedade.

Assim, e tal como especificado, para que possamos gravar instâncias da classe que temos vindo a usar, `FichaAluno`, numa `ObjectOutputStream`, deveremos, de imediato se possível, tornar tal classe `Serializable`.

A modificação da declaração de tal classe para:

```
public class FichaAluno implements Serializable {
```

não tem quaisquer implicações adicionais na definição da classe e na sua anterior utilização, já que, de facto, todas as suas variáveis de instância, incluindo instâncias de `Vector`, são *serializable*.

Garantidas estas condições, a gravação de instâncias da classe `FichaAluno` numa `ObjectOutputStream`, depois de criar as diferentes instâncias, passaria pela escrita do simples código seguinte:

```
// Grava as MAXFICHAS instâncias numa ObjectStream

try {
    ObjectOutputStream oout =
       new ObjectOutputStream(new
                  FileOutputStream("fich7.dat"));
    for(int i=0; i < MAXFICHAS-1; i++) {
       oout.writeObject(turma[i]);
    }
    oout.flush(); oout.close();
   }
```

```
catch(IOException e)
  { System.out.println(e.getMessage()); }

// lê de novo as MAXFICHAS fichas !

try {
      ObjectInputStream oin =
       new ObjectInputStream(new
                    FileInputStream("fich7.dat"));
      for(int i=0; i < MAXFICHAS-1; i++) {
        novaturma[i] = (FichaAluno) oin.readObject();
      }
      oin.close();
    }
    catch(IOException e)
      { System.out.println(e.getMessage()); }
    catch(ClassNotFoundException e)
      { System.out.println(e.getMessage()); }
```

Este programa, que realiza a gravação das instâncias seguida da leitura para um outro *array*, executará a gravação em cerca de 17 segundos e a leitura em cerca de 95s, valores que reflectem a complexidade maior da operação de leitura e reconstrução dos objectos gravados que, para além das leituras dos dados, deve realizar a sua alocação em memória central e resolver as referências entre objectos (ou seja, realizar a *desserialização*), mas que são valores que reflectem uma grande eficiência que se associa à facilidade de utilização.

Note-se que a leitura é feita usando uma `ObjectInputStream`, e, em especial, usando o método `readObject()` que dá como resultado um `Object`, pelo que é sempre necessário realizar o *casting* para a classe específica a que diz respeito a leitura, no nosso exemplo, `FichaAluno`.

Apesar de os ficheiros criados não serem consultáveis através de um editor, a comodidade que as *object streams* oferecem na sua utilização, havendo apenas que ter em atenção o facto de que só podem ser serializados objectos de classes que declarem implementar a interface `Serializable`, torna-as ideais como suporte à implementação de mecanismos de *persistência* nos objectos criados em memória central pelos programas.

A serialização e desserialização de objectos é um processo relativamente complexo porque os relacionamentos entre objectos podem ser facilmente deturpados caso a desserialização não seja feita correctamente.

Considere-se o exemplo da Figura 8.4, segundo a qual duas das posições de um dado vector ou *array* contêm referências para a *mesma* instância de `FichaAluno`.

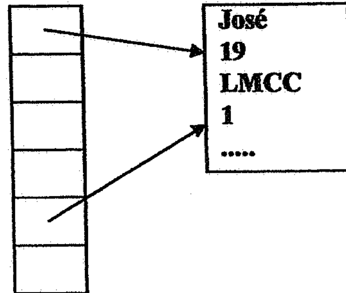


Fig. 8.4 – Estrutura a *serializar*

Após a leitura da `ObjectInputStream` pretendemos que tal relacionamento seja completamente respeitado, sendo de notar que, por exemplo, o relacionamento que se apresenta na figura seguinte, aparentemente equivalente, é, de facto, resultado de uma desserialização errada.

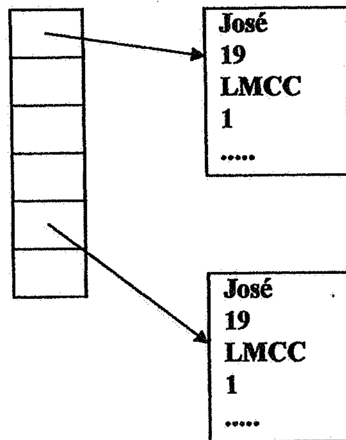


Fig. 8.5 – Exemplo de desserialização errada

Repare-se que as duas posições do vector ou *array*, antes da serialização, tinham uma referência para a mesma instância de `FichaAluno`, enquanto que agora têm referências para instâncias (objectos) distintas, ainda que de igual valor.

Felizmente que na versão actual de JAVA já não necessitamos de programar os algoritmos de serialização e de desserialização, pois estão já implementados nos métodos `writeObject()` e `readObject()` das *object streams*.

Usando *object streams*, as instâncias de classes *serializable* podem facilmente ser gravadas em memória secundária, e lidas para memória central sempre que seja importante incorporá-las no ambiente de *run-time*. Resta finalmente referir que as variáveis declaradas com atributos `static` ou `transient` não são serializadas.

8.5 OUTRAS CONSIDERAÇÕES

Todas as operações de escrita e leitura de dados realizadas até ao momento, foram programadas exteriormente aos objectos a gravar ou ler, em particular, em código apresentado no método `main()` das diferentes classes de teste.

Esta foi uma opção tomada que teve a ver, fundamentalmente, com a necessidade de alguma proximidade do código facilitando, assim, a exposição. Torna-se, por tal motivo, importante salientar agora que poderíamos ter optado por estruturar esse código noutros métodos `static` da classe de teste, que poderiam ser invocados a partir do código escrito no método `main()`. De tal opção teria resultado uma codificação mais estruturada dos exemplos, que se aconselha, mas mais complexa e extensa de visualizar e analisar.

Por outro lado, nada impede que certas classes definam como métodos de instância certos métodos específicos para a sua gravação em ficheiro ou noutros meios. Faz até algum sentido que, se no momento da definição da classe `FichaAluno` fosse um dos requisitos que todas as instâncias desta classe deveriam implementar um método que realizasse a sua gravação numa *stream*, usando uma `PrintWriter`, tivéssemos definido, por exemplo, um método do tipo:

```
public void printFichaAluno(PrintWriter opw) {
```

O que já não é muito comum, é que uma dada classe defina métodos de instância para leitura. De facto, não é muito clara a semântica de enviar a uma dada instância uma mensagem para ler os seus conteúdos de uma dada *stream*.

ESCRITAS E LEITURAS POLIMÓRFICAS

Em PPO é muito comum termos estruturas em que os objectos referenciados não são todos da mesma classe. Por exemplo, a classe `Stack` de JAVA, tal como muitas outras, é programada considerando que os seus elementos são instâncias de `Object`. Como sabemos, tal significa, na prática, que poderão ser instâncias de uma qualquer subclasse de `Object`, sendo portanto tais estruturas genéricas.

Sem a existência de *streams de objectos*, a gravação de uma instância de *stack* numa qualquer *stream*, implicaria um conjunto fundamental de passos, tais como, para cada elemento (seja `elem`) da *stack* fazer:

- obter o nome da sua classe usando `elem.getClass().getName();`
- escrever na *stream* este nome, usando formato UTF com `writeUTF()`, tal como em `writeUTF(elem.getClass().getName());`
- escrever na *stream* o resultado de `elem.writeAs(??)`, método que seria sempre muito dependente da *stream* parâmetro (subclasse de `Writer` ou de `OutputStream` ?), que grava o receptor.

A reconstituição da *stack* original, dependeria muito da capacidade de reconhecer de que classe cada objecto seria instância, e da invocação do respectivo método de leitura função da *stream* a ler, sendo necessário implementar os seguintes passos.

- ler o nome da classe com `String className = input.readUTF();`
- obter a classe com `Classe c = Class.forName(className);`
- criar uma instância vazia da classe que deverá ser a classe mais genérica do problema, em geral uma classe abstracta, mas que, neste caso, poderá ser a classe `Object`. O código neste exemplo seria:

  ```
  Object obj = c.newInstance();
  ```

  mas poderia ser, noutro contexto:

  ```
  Documento obj = c.newInstance();
  ```

- Em ambos os casos, e na condição da respectiva classe ter implementado um método `lerDados(input)` que leia de uma *input stream* os dados da instância, a expressão `obj.lerDados(input)` resultaria sempre numa reconstituição do objecto desejado, a armazenar, em memória central, em qualquer estrutura definida.

Ainda que a utilização das `ObjectStreams` nos permita, embora sob um certo número de restrições, usufruir automaticamente destes benefícios, é importante que se compreendam os mecanismos que estão na base de algumas de tais facilidades.

Em conclusão, a linguagem JAVA oferece-nos um conjunto bastante completo de classes de I/O, quase todas elas baseadas na noção de *stream*. Algumas delas são bastante específicas, como, por exemplo, as classes `PushbackInputStream` ou `CharArrayReader`. No entanto, as classes de maior interesse geral, ou seja, para armazenamento e leitura de dados usando ficheiros, acabam por ser todas aquelas que neste capítulo foram estudadas e experimentadas. Assim sendo, este capítulo é, simultaneamente, uma síntese e uma análise das mais úteis classes de I/O de JAVA.

8.6 EXERCÍCIOS PROPOSTOS

A proposta de exercícios é bastante genérica, dado não fazer qualquer sentido que, relativamente a questões de I/O, em particular quando se trata de reutilizar código já existente, se possam colocar questões muito específicas ou detalhadas.

Assim sendo, propõem-se como exercícios associados a este capítulo, a redefinição ou especialização de classes anteriormente apresentadas, ou até, propostas como exercício, por forma a que as suas instâncias sejam, parcial ou totalmente, capazes de implementar os seguintes comportamentos:

- *Serem representáveis num ficheiro de texto, consultável por um qualquer editor de texto;*
- *Serem representáveis num ficheiro de objectos, e facilmente recuperáveis em memória central.*

As soluções desenvolvidas devem tratar um número significativo de instâncias de cada uma das classes consideradas.