

## PG II

# Programação Orientada aos Objectos em Java

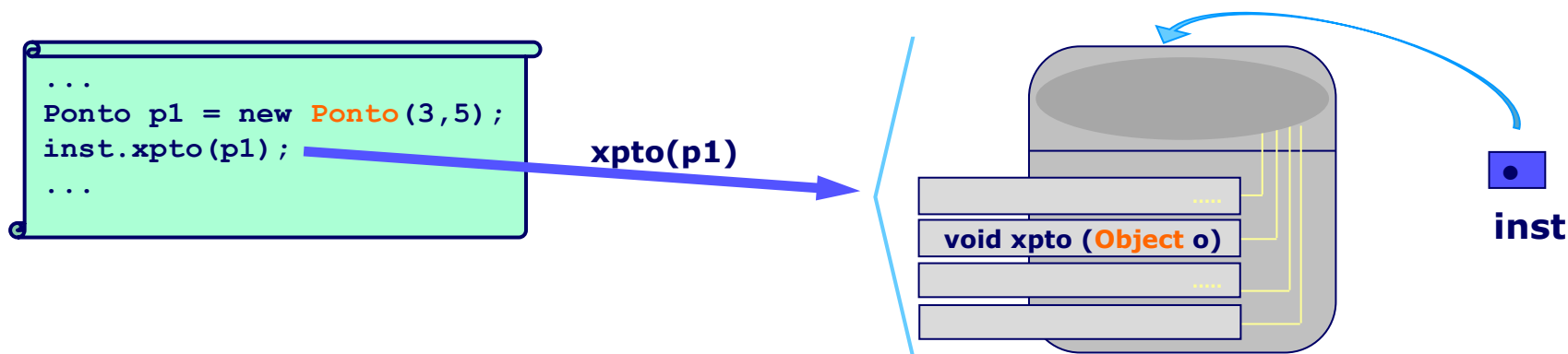
### **Polimorfismo:**

- O Topo da Hierarquia: Classe Object;
- Polimorfismo <> "Casting" entre Objectos;
- pg2.util.Collections

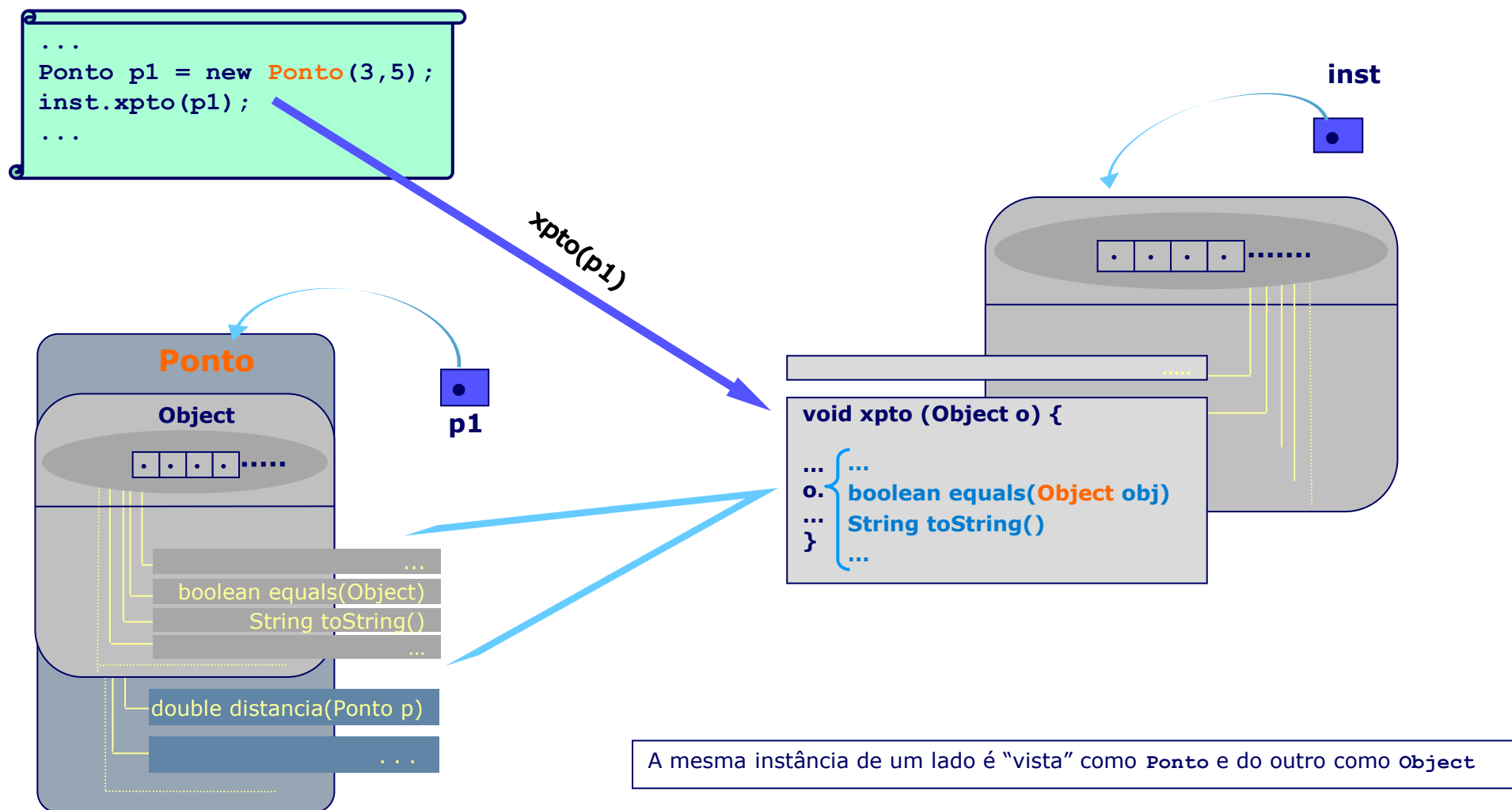
# O Topo da Hierarquia: Classe Object

O facto da **Classe Object** estar no topo da hierarquia garante que:

- Qualquer Classe em Java é, directa ou indirectamente, derivada da Classe Object (propriedade **transitiva**).
- Em consequência do ponto anterior, todas as **instâncias** respondem aos métodos: `boolean equals(Object obj)` e `String toString()`, que foram **herdados da Classe Object**.
- Uma determinada função ou método pode tratar um parâmetro como sendo **instância da classe Object** e receber uma **instância de outra Classe** qualquer, porque é garantido que essa instância tem o **mesmo comportamento** que a instância de Object.
- Uma variável declarada como Object (do tipo `Object`), poderá ser iniciada com uma instancia de qualquer outra Classe.
- Via **polimorfismo**, podemos enviar a um método, que recebe como parâmetro uma instância da Classe Object, uma instância de outra Classe qualquer.

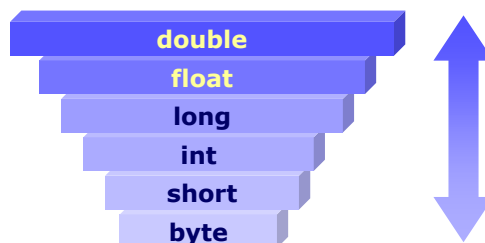


## ... O Topo da Hierarquia: Classe Object



# Polimorfismo <> “Casting” entre Objectos

No mecanismo de **casting entre tipos primitivos**, existe uma redefinição dos **recursos de memória** de acordo com a transformação realizada.



No “casting” entre objectos, **não existe uma transformação da instância** mas apenas na forma como é apresentada. Exemplo:

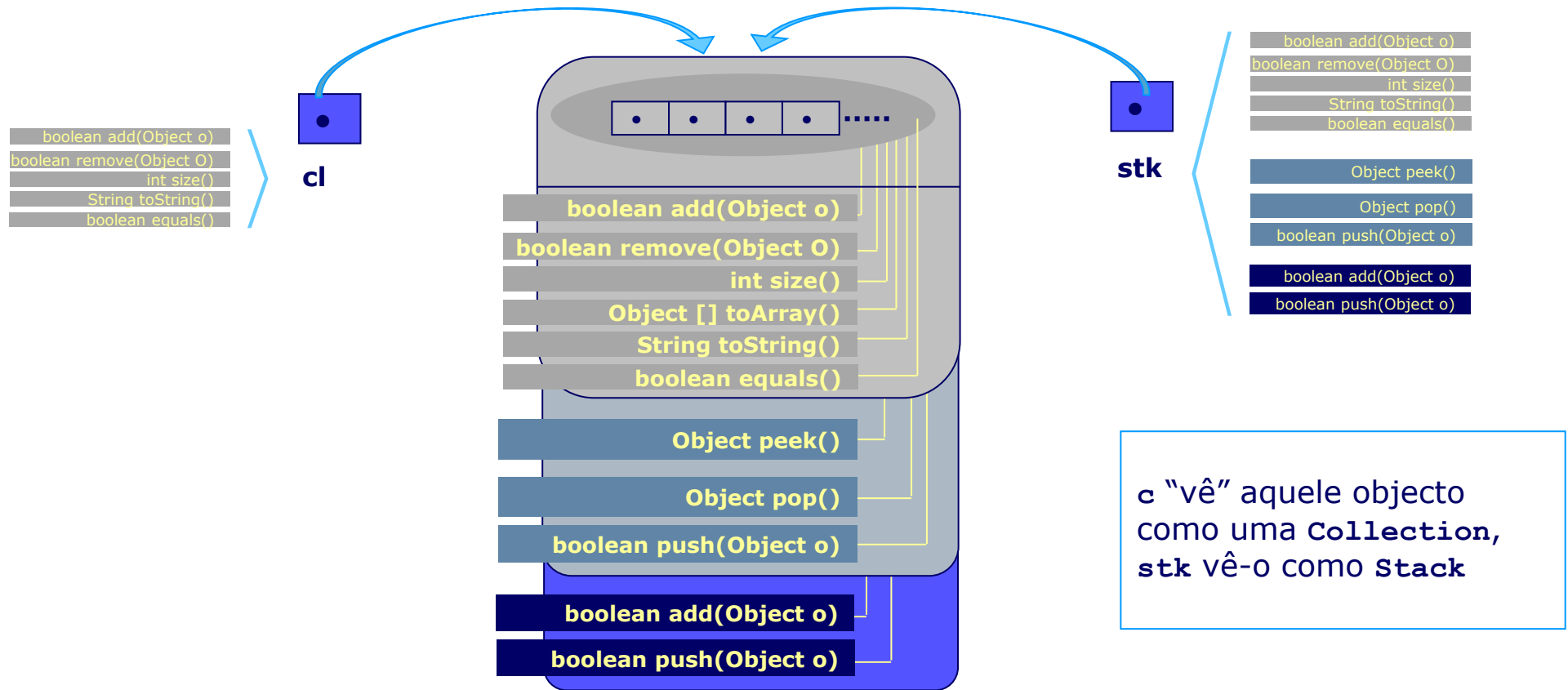
```
...
StackInt stk = new StackInt();
...
Collection c = stk;
...
```

As variáveis `c` e `stk` referenciam o mesmo objecto, mas enquanto `c` “vê” aquele objecto como uma `Collection`, `stk` vê-o como `Stack`. Ambas as perspectivas são correctas na medida em que sendo `Stack` uma derivação de `Collection`, então `stk` pode ser entendida também como instância de `Collection`, ou seja, ambas as expressões são `true`:

```
stk instanceof Stack e
stk instanceof Collection
```

# ... Polimorfismo <> "Casting" entre Objectos

```
...
StackInt stk = new StackInt();
...
Collection c = stk;
...
```



## ... Polimorfismo <> "Casting" entre Objectos

O mesmo processo em sentido contrário pode já não fazer sentido.

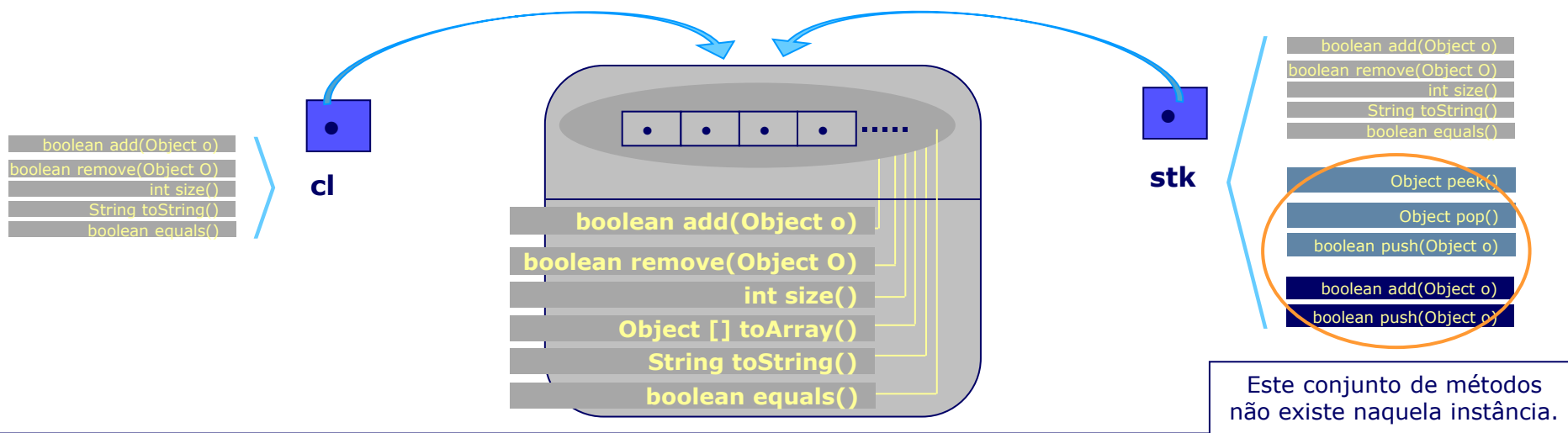
E por isso, um "down cast" tem que ser sempre declarado explicitamente ao compilador.

```
...
Collection cl = new Collection();
StackInt stk = (StackInt) cl;
...
```

Descer ao longo da hierarquia é entendida como uma "especialização", pelo que tem que ser declarada explicitamente.

No entanto isto não significa que em execução não seja dada uma excepção. Neste caso em *runtime*, seria dada a seguinte excepção:

```
java.lang.ClassCastException: pg2.aula05.Collection
```



Seja considerada uma classe `pg2.util.Collections` com um conjunto de métodos estáticos, que operam sobre os contentores do *package* `pg2.aula05`.

Seja considerado no âmbito desta classe o seguinte método estático:

```
package aula05;

public class Collections {

    /**
     * Devolve uma nova coleção apenas com os elementos distintos do contentor
     * <code>c</code>. Ou seja, quaisquer dois elementos, <code>e1</code> e <code>
     * e1</code>, da nova coleção verificam a condição <code>e1.equals(e2) == false
     * </code>. A capacidade da nova coleção será igual ao número de elementos alojado,
     * ou seja, <code>elements.length</code> é igual a <code>c.size()</code>.
     */
    public static Collection distinctObjects(Collection c) {
        Object [] a = c.toArray();           // Passagem para array dos elementos de c.
        int size = 1;                         // Contabiliza o nº de elementos distintos.
        for(int i=1; i<a.length; i++){       // Avaliação de cada um dos elementos do array.
            int j = 0;
            for (; j < i; j++)                // Testa a igualdade entre o elemento i e cada
                if(a[j] != null)              // um dos elementos anteriores, j.
                    if (a[j].equals(a[i]))    // Se encontrar dois elementos iguais interrompe
                        break;                 // este ciclo.
            if(j==i) size++;                  // Se o ciclo não foi interrompido então incrementa size;
            else a[i] = null;                 // se não, limpa do array o elemento repetido.
        }
        Collection novaCol = new Collection(size); // Cria uma nova coleção com capacidade
        for(int i=a.length-1; i>=0; i--)         // para o nº de elementos distintos.
            if (a[i] != null) novaCol.add(a[i]);  // Passagem dos elementos do array para a
        return novaCol;                          // nova coleção que é retornada no final.
    }
}
```

## Method Detail

### distinctObjects

```
public static aula05.Collection distinctObj
```

Devolve uma nova coleção apenas com os elementos distintos do contentor `c`. Ou seja, quaisquer dois elementos, `e1` e `e1`, da nova coleção verificam a condição `e1.equals(e2) == false`. A capacidade da nova coleção será igual ao número de elementos alojado, ou seja, `elements.length` é igual a `c.size()`.

## ... pg2.util.Collections

À descrição da classe Collections podia ainda ser acrescentado: "Disponibiliza algoritmos polimórficos que operam sobre colecções...".

Porquê **polmórficos**?

Porque podem ser passados como parâmetro à função `distinctObjects(Collection c)`, tanto objectos de `aula05.Collection`, como de `aula05.Stack` como de `aula05.StackInt`. Todos eles são, ou derivam de `aula05.Collection`, como tal qualquer um destes objectos pode ser considerado `Collection`.

```
...
public static void main(String [] args){
    Collection col = new Collection(args.length);
    Stack stk = new Stack(args.length);
    for (int i = args.length-1; i >= 0; i--) {
        col.add(args[i]);
        stk.add(args[i]);
    }
    IO.cout.writeln("\nContentor original = " + col.toString());
    IO.cout.writeln("Elementos distintos = " + Collections.distinctObjects(col).toString() + "\n");
    IO.cout.writeln("\nStack original = " + stk.toString());
    IO.cout.writeln("Elementos distintos = " + Collections.distinctObjects(stk).toString() + "\n");
}
```

```
D:\work>java aula05.Collections 34 76 42 89 34 89 76 42 34 87 90 89
Contentor original = [(34)<(76)<(42)<(89)<(34)<(89)<(76)<(42)<(34)<(87)<(90)<(89)]
Elementos distintos = [(34)<(76)<(42)<(89)<(87)<(90)]

Stack original = [(34)<(76)<(42)<(89)<(34)<(89)<(76)<(42)<(34)<(87)<(90)<(89)]
Elementos distintos = [(34)<(76)<(42)<(89)<(87)<(90)]

D:\work>
```



Ainda no âmbito dos algoritmos polimórficos, seja agora considerada na classe `pg2.util.Arrays` um novo método estático, com a funcionalidade semelhante ao de `Collections`:

## Conclusões:

- Se quisermos implementar uma função/método suficientemente genérico, que receba e trate qualquer **tipo** de parâmetro (**objecto**) que lhe seja passado, então declaramos parâmetros do tipo **Object**.
- Contudo, estamos limitados ao **comportamento** (interface) disponibilizado na classe Object, que se resume a: `getClass()`, `boolean equals (Object obj)`, `Object clone()` e `String toString()`

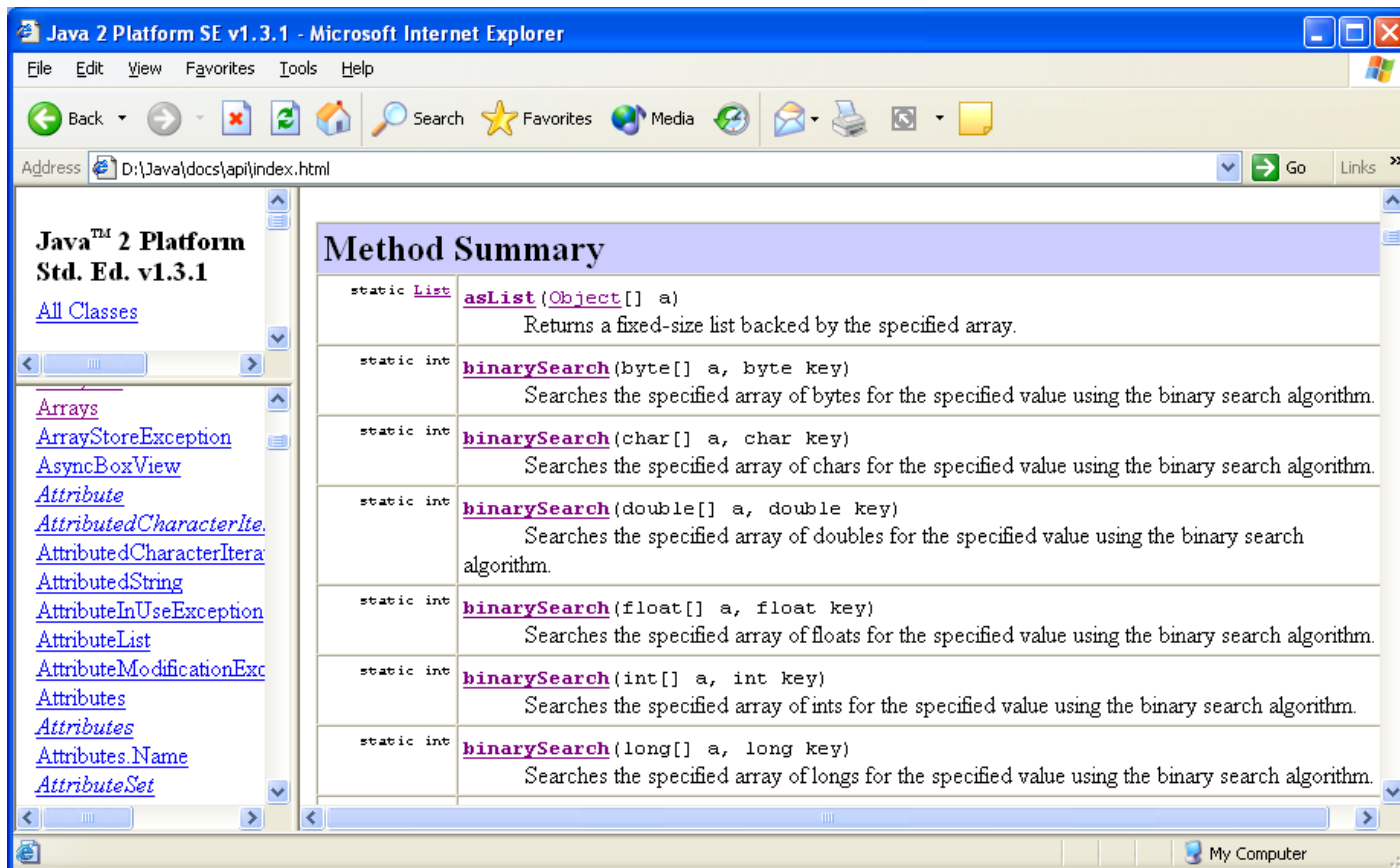
Ou seja, para um determinado parâmetro "o" de um método xpto, apenas podemos fazer uso dos métodos: `getClass()`, `boolean equals (Object obj)`, `Object clone()` e `String toString()`

```
public void xpto (Object o) {  
    ...  
    String str = o.toString();  
    Object obj = o.clone();  
    ...  
}
```

E se quisermos implementar um método suficientemente genérico para receber como parâmetro uma **instância de Object**, mas que tenha outros métodos disponíveis **além dos já existentes na interface da classe Object**?

# Exemplo da classe Arrays

A classe Arrays, disponibiliza um conjunto de métodos diversos, nomeadamente para ordenação e procura de elementos:



**Method Summary**

static List	<b>asList</b> (Object[] a)	Returns a fixed-size list backed by the specified array.
static int	<b>binarySearch</b> (byte[] a, byte key)	Searches the specified array of bytes for the specified value using the binary search algorithm.
static int	<b>binarySearch</b> (char[] a, char key)	Searches the specified array of chars for the specified value using the binary search algorithm.
static int	<b>binarySearch</b> (double[] a, double key)	Searches the specified array of doubles for the specified value using the binary search algorithm.
static int	<b>binarySearch</b> (float[] a, float key)	Searches the specified array of floats for the specified value using the binary search algorithm.
static int	<b>binarySearch</b> (int[] a, int key)	Searches the specified array of ints for the specified value using the binary search algorithm.
static int	<b>binarySearch</b> (long[] a, long key)	Searches the specified array of longs for the specified value using the binary search algorithm.

Os algoritmos de pesquisa dicotómica pressupõem que o array recebido como parâmetro já foi ordenado.

## binarySearch

```
public static int binarySearch(int[] a,  
                                int key)
```

Searches the specified array of ints for the specified value using the binary search algorithm. The array **must** be sorted (as by the `sort` method, above) prior to making this call. If it is not sorted, the results are undefined. If the array contains multiple elements with the specified value, there is no guarantee which one will be found.

### Parameters:

a - the array to be searched.

key - the value to be searched for.

### Returns:

index of the search key, if it is contained in the list; otherwise,  $-(\text{insertion point}) - 1$ . The *insertion point* is defined as the point at which the key would be inserted into the list: the index of the first element greater than the key, or `list.size()`, if all elements in the list are less than the specified key. Note that this guarantees that the return value will be  $\geq 0$  if and only if the key is found.

### See Also:

[`sort\(int\[\]\)`](#)

## ... binarySearch

```

Arrays Auxiliar
import pg2.io.IO;
import pg2.trabl.Auxiliar;
/**
 * Title: Implementação de um algoritmos de ordenação e procura
 * @author MCarvalho
 * @version 1.1
 */

public class Arrays {

    public static int binarySearch(int[] a, int key, int low, int high) {
        if (low <= high) {
            int mid =(low + high)/2;
            long midVal = a[mid];
            if (midVal < key)
                low = mid + 1;
            else
                if (midVal > key)
                    high = mid - 1;
                else
                    return mid; // key found
            mid = Arrays.binarySearch(a, key, low, high);
            return mid;
        }
        return -(low + 1); // key not found.
    }
}

```

Algoritmo de  
pesquisa dicotómica  
recursivo

## ... binarySearch

```

Arrays Auxiliar

public static void main (String args []){
    int [] aux = new int[args.length];
    for (int i=0; i<aux.length;i++)
        aux[i] = Integer.parseInt(args[i]);

    Auxiliar.sort(aux, 0, aux.length-1);
    IO.cout.write("\n");
    for (int i=0; i<aux.length;i++)
        IO.cout.write("(" + aux[i] + ")");
    IO.cout.writeln("");

    IO.cout.writeln("\nInsira o numero a pesquisar:");
    int val = IO.cin.readInt();

    IO.cout.writeln("\nResultado da pesquisa:\n" +
        Arrays.binarySearch(aux,val,0,aux.length-1));
}

```

```

C:\ Command Prompt

D:\Java\Projects>java pg2.aula05.Arrays 23 15 31 5 45 9
[<5><9><15><23><31><45>]
Insira o numero a pesquisar:
31
Resultado da pesquisa:
4

```

```

C:\ Command Prompt

D:\Java\Projects>java pg2.aula05.Arrays 23 15 31 5 45 9
[<5><9><15><23><31><45>]
Insira o numero a pesquisar:
17
Resultado da pesquisa:
-4
D:\Java\Projects>

```

## ... binarySearch para arrays de Objectos genéricos

Então, e se quisermos implementar uma pesquisa para fracções?

**Resposta:** Tenho que implementar um novo método:

```
public static int binarySearch( Fraccao[] a, int key, int low, int high) {  
    . . .  
}
```

E se agora quisermos implementar uma pesquisa para pontos?

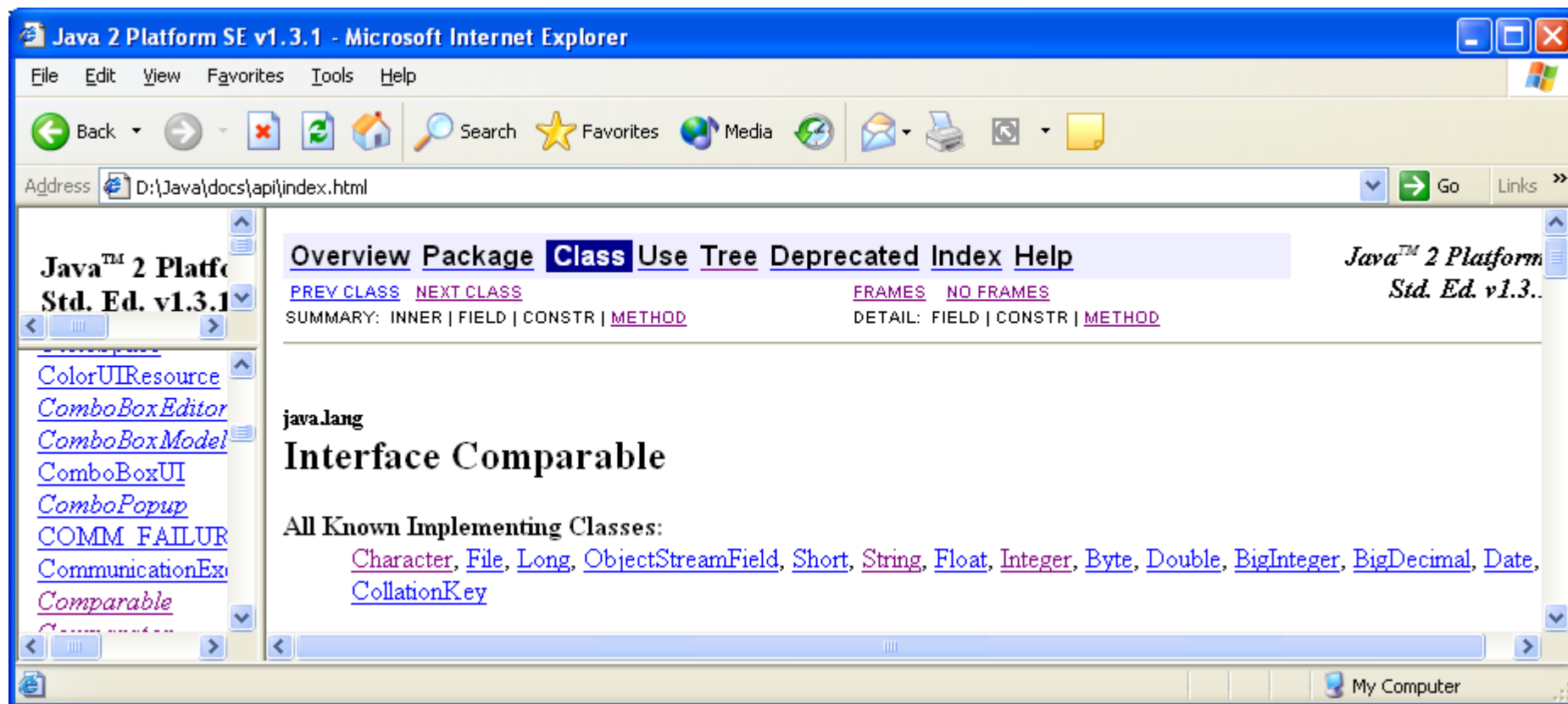
**Resposta:** Tenho que implementar mais um método?

Então era preferível implementar um só método mais genérico:

```
public static int binarySearch( Object[] a, int key, int low, int high) {  
    . . .  
}
```

**Questão:** Mas a classe Object não implementa o método ***compareTo()***. Então como solucionamos esta questão?

# ... binarySearch para arrays de Objectos genéricos



Java 2 Platform SE v1.3.1 - Microsoft Internet Explorer

File Edit View Favorites Tools Help

Address <D:\Java\docs\api\index.html> Go Links

Java™ 2 Platform Std. Ed. v1.3.1

Overview Package **Class** Use Tree Deprecated Index Help

[PREV CLASS](#) [NEXT CLASS](#) [FRAMES](#) [NO FRAMES](#)

SUMMARY: INNER | FIELD | CONSTR | [METHOD](#) DETAIL: FIELD | CONSTR | [METHOD](#)

java.lang

## Interface Comparable

All Known Implementing Classes:

[Character](#), [File](#), [Long](#), [ObjectStreamField](#), [Short](#), [String](#), [Float](#), [Integer](#), [Byte](#), [Double](#), [BigInteger](#), [BigDecimal](#), [Date](#), [CollationKey](#)

## Method Summary

int [compareTo](#)(Object o)  
Compares this object with the specified object for order.



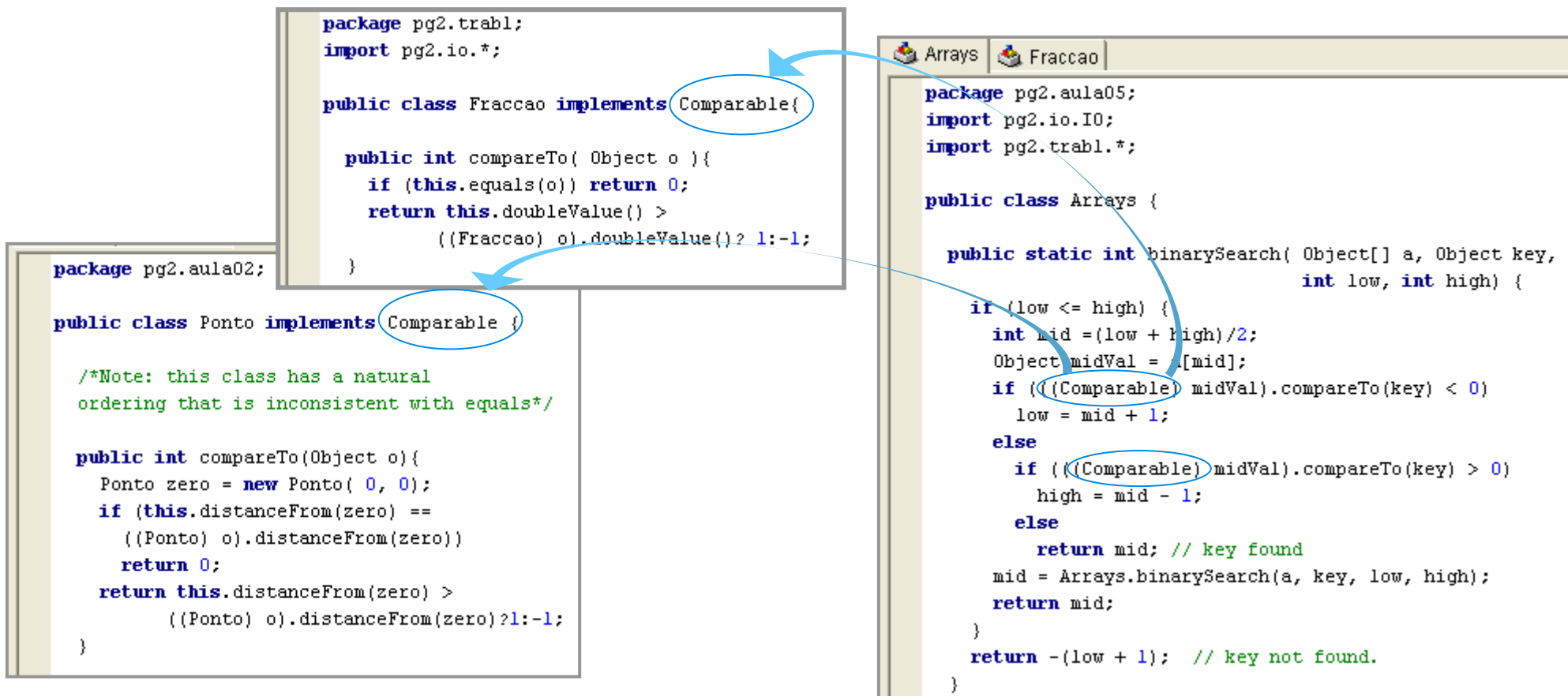
## ... binarySearch para arrays de Objectos genéricos

```
public static int binarySearch(int[] a, int key,
                              int low, int high) {
    if (low <= high) {
        int mid = (low + high) / 2;
        long midVal = a[mid];
        if (midVal < key)
            low = mid + 1;
        else
            if (midVal > key)
                high = mid - 1;
            else
                return mid; // key found
        mid = Arrays.binarySearch(a, key, low, high);
        return mid;
    }
    return -(low + 1); // key not found.
}
```

```
public static int binarySearch( Object[] a, Object key,
                              int low, int high) {
    if (low <= high) {
        int mid = (low + high) / 2;
        Object midVal = a[mid];
        if (((Comparable) midVal).compareTo(key) < 0)
            low = mid + 1;
        else
            if (((Comparable) midVal).compareTo(key) > 0)
                high = mid - 1;
            else
                return mid; // key found
        mid = Arrays.binarySearch(a, key, low, high);
        return mid;
    }
    return -(low + 1); // key not found.
}
```

## ... binarySearch para arrays de Objectos genéricos

Neste exemplo de implementação de um "binarySearch para arrays de Objectos genéricos", a interface **Comparable** funciona como a ponte entre a Classe Object parâmetro, da função binarySearch, e os objectos que queremos passar à função, seja um array de Fracções, Pontos, etc



# Declaração de interfaces

```
public interface Comparable {  
    public int compareTo(Object o);  
}
```

Uma *interface* Java pode conter:

- Um conjunto opcional de constantes, ou seja, identificadores declarados como **static** e **final**;
- Um conjunto de assinaturas (declarações) de métodos que são implicitamente abstractos (sendo opcional usar o qualificador **abstract**).

As *interfaces* Java possibilitam assim:

- A implementação de métodos genéricos;
- Normalização de API's entre diversas Classes;
- Tirar partido do mecanismo de polimorfismo.

e ainda:

- Normalizar as API's a partir de certos pontos da hierarquia.

## ... Declaração de interfaces

... Normalizar as API's a partir de certos pontos da hierarquia. **Exemplo: java.util.Collection**

```
public interface Collection
```

The root interface in the *collection hierarchy*. A collection represents a group of objects, known as its *elements*. Some collections allow duplicate elements and others do not. Some are ordered and others unordered. The SDK does not provide any *direct* implementations of this interface: it provides implementations of more specific subinterfaces like *Set* and *List*. This interface is typically used to pass collections around and manipulate them where maximum generality is desired.

```
package java.util;
public interface Collection {
    boolean add(Object o);
    boolean addAll(Collection c);
    void clear();
    boolean contains(Object o);
    boolean containsAll(Collection c);
    boolean equals(Object o);
    int hashCode();
    boolean isEmpty();
    Iterator iterator();
    boolean remove(Object o);
    boolean removeAll(Collection c);
    boolean retainAll(Collection c);
    int size();
    Object[] toArray();
    Object[] toArray(Object[] a);
}
```

# Classes Abstractas

Se numa interface existirem, além dos métodos abstractos:

- Métodos implementados (concretos);
- Variáveis de instância,

então teremos uma **Classe Abstracta**.

Exemplo:

```
public abstract class AbstractCollection implements Collection
```

A classe **AbstractCollection**, implementa a interface **Collection**, mas mantém alguns métodos abstractos tais como `iterator()` e `size()`, daí ser uma **Classe Abstracta**.