# COLECÇÕES DE OBJECTOS EM JAVA6

## ESTUDO DO
## JAVA COLLECTIONS FRAMEWORK 5.0

### PARTE I

**F. MÁRIO MARTINS**
**DI/UNIVERSIDADE DO MINHO**

**2007/2008**

# COMPOSIÇÃO/AGREGAÇÃO
# COLECÇÕES DE OBJECTOS

**QUESTÃO 1: COMO CRIAR EM JAVA VARIÁVEIS DE INSTÂNCIA QUE SÃO COLECÇÕES DE VALORES?**

**RESPOSTA 1: USANDO ARRAYS**

**EXEMPLO: COLECÇÃO DE TEMPERATURAS OU DE NOTAS (VALORES DE TIPO `double` E `int`).**

```
double[] temps = new double[MAXTEMP];
int[] notas = new int[100];


double somaTemp = 0.0;
for(int i = 0; i < MAXTEMP; i++) somaTemp += temps[i];  ou ainda
for(double temp : temps) somaTemp += temp; ☺   // construção foreach


int soma = 0.0;
for(int nota : notas) soma += nota; ☺   // construção foreach
```

## Questão2: Como criar em Java variáveis de instância que são colecções de objectos?

## Resposta 2.1: Usando arrays

## Exemplo: Colecções de objectos.

```
String[] nomes = new String[MAX_NOMES];
Balao[] baloes = new Balao[10];
Ponto2D[] linha = new Ponto2D[MAX_PONTOS];


int totalChars = 0;
for(String nm : nomes) totalChars += nm.length();  ☺   // construção foreach


int somaAlt = 0.0;
for(Balao b : baloes) somaAlt += b.getAltura();    ☺   // construção foreach

for(Balao b : baloes) out.println(b.toString());   ☺   // construção foreach


int simetricos = 0;
for(Ponto2D p : linha)
      if(p.simetrico()) simetricos++ ;             ☺   // construção foreach
```

# Resposta 2.2: Usando as Diversas Colecções de JCF (The "Java Collection Framework 5.0")

**Todas as Colecções de JCF São Estruturas de Objectos bem Nossas Conhecidas, cf. Listas, Conjuntos e Mappings (em várias implementações).**

**em JAVA5 passaram a Ser parametrizadas, ou seja, Devemos declarar os tipos dos Objectos que elas vão conter.**

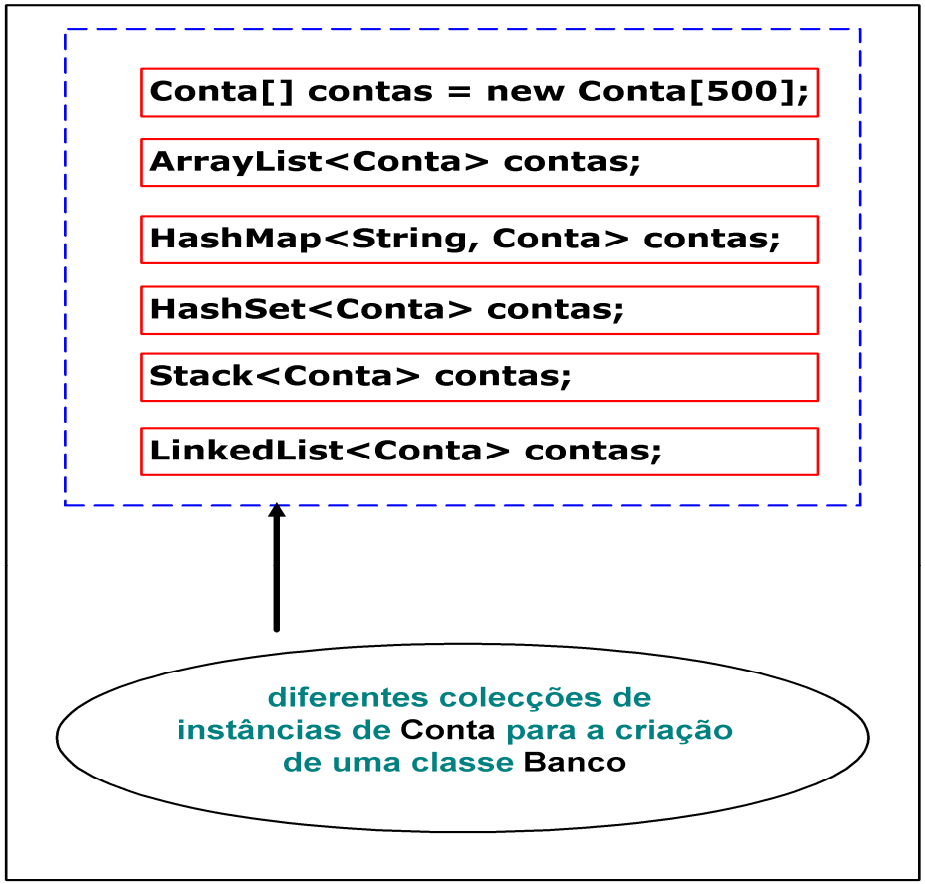
**A verificação de tipos passa a ser feita Pelo compilador (ou seja em tempo de Compilação), e não em tempo de execução Como em anteriores versões de Java**

**(Isto é muito bom para a linguagem e Para a segurança dos programas !!)**

## 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.

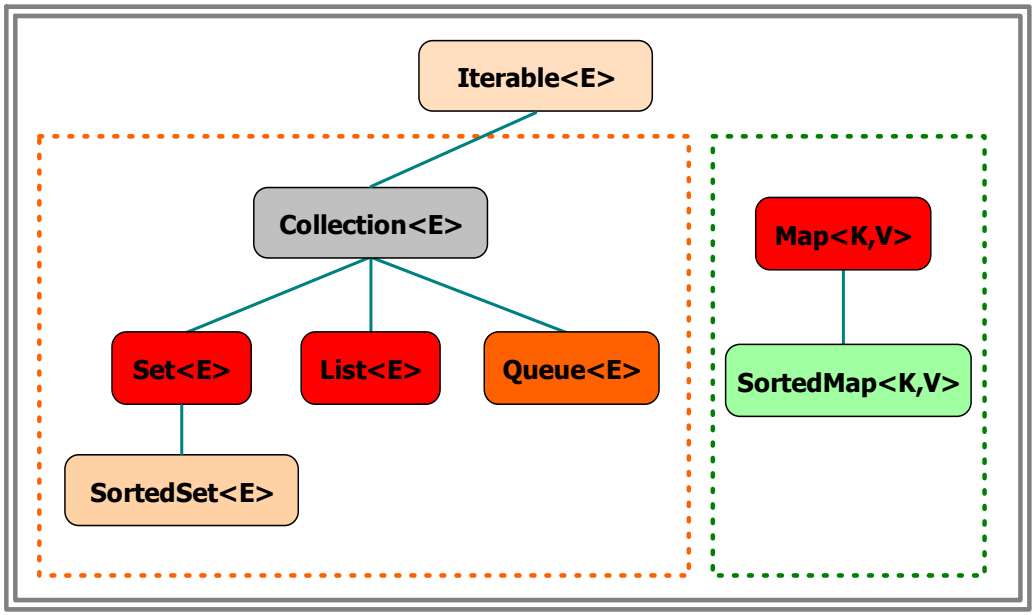
**Banco** (estruturação de Conta)



# COLECÇÕES DE JAVA6: "JAVA COLLECTIONS FRAMEWORK 5.0"

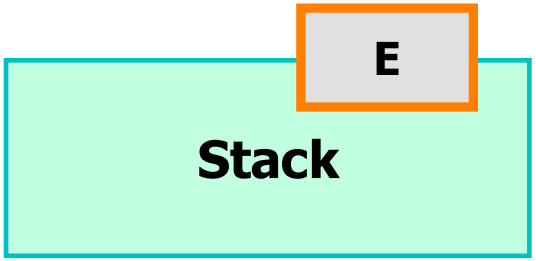
## SÃO VÁRIAS CLASSES GENÉRICAS (PARAMETRIZADAS), QUE CORRESPONDEM A IMPLEMENTAÇÕES DAS ESTRUTURAS FUNDAMENTAIS:

- LISTAS (ORDEM E POSSÍVEIS DUPLICADOS): SATISFAZEM A API `List<E>`
- CONJUNTOS (SEM ORDEM E SEM DUPLICADOS): SATISFAZEM A API `Set<E>`
- CORRESPONDÊNCIAS UNÍVOCAS: SATISFAZEM A API `Map<K,V>`




**JCF5.0: Tipos de colecções (APIs = INTERFACES)**

## CLASSES GENÉRICAS => TIPOS PARAMETRIZADOS




**Tipo genérico Stack<E>**

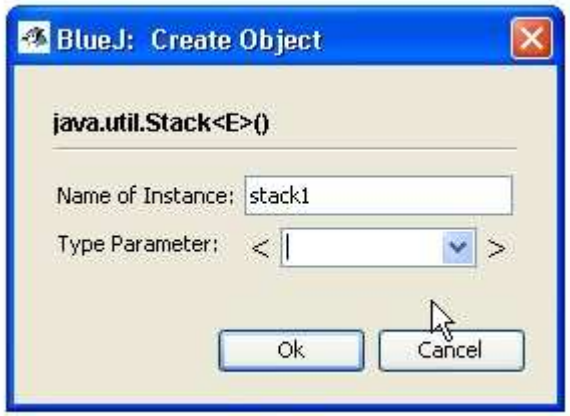
⇓ **para cada instanciação de E com uma classe, temos um tipo concreto**

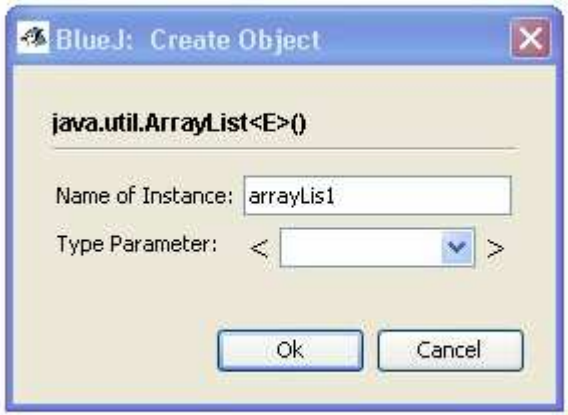
**Stack<String>**
**Stack<Ponto2D>**
**Stack<Livro>**
**Stack<Integer>**

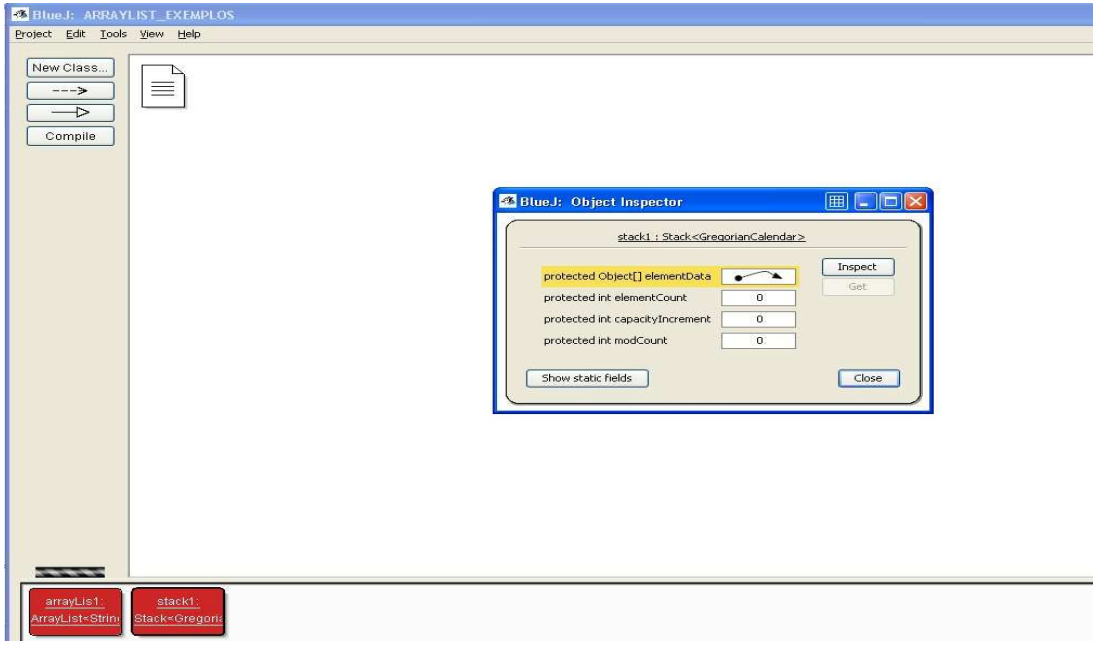
**todas satisfazem a API de Stack<E>**

**Stack<int> => restrição E ≠tipo primitivo**

## O IDE BLUEJ, SE INSTALADO SOBRE JAVA5-6, SABE QUE AS COLECÇÕES SÃO PARAMETRIZADAS, PELO QUE PEDE AO UTILIZADOR PARA INDICAR O TIPO DO PARÂMETRO.



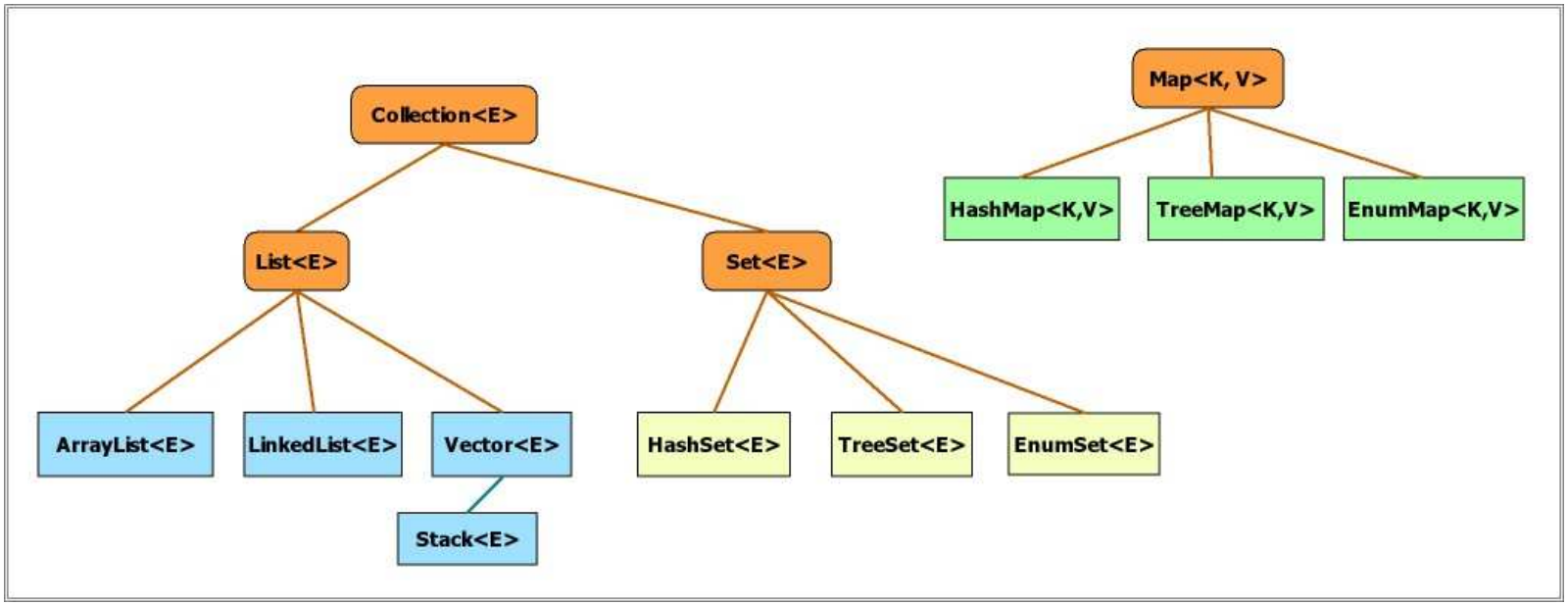



Tal como estudaremos mais tarde, uma Interface de JAVA, é apenas uma estrutu-ra sintáctica que define as assinaturas de um conjunto de métodos, ou seja, uma API abstracta (porque não define código).

Serão as classes que as irão implementar em concreto, ou seja, que irão apresentar diferentes soluções de implementação para as mesmas Interfaces ou APIs.

Para cada API (interface) existem várias classes de implementação:




Estudaremos Fundamentalmente:

`ArrayList<E>; HashSet<E> e TreeSet<E>; HashMap<K,V> e TreeMap<K,V>;`

## CLASSES GENÉRICAS => TIPOS PARAMETRIZADOS

**EXEMPLOS:**

```
ArrayList<String> cidades;
ArrayList<Ponto2D> linha;
LinkedList<Circulo> bolas;
ArrayList<Aluno> turma;
```

listas de objectos

```
HashSet<Ponto2D> conjPontos;
HashSet<Aluno> bolseiros;
TreeSet<Ponto2D> plano;
```

conjuntos de objectos

```
HashMap<String, Aluno> inscritos;
HashMap<Ponto2D, Propriedade> plano2D;
TreeMap<GPS, InfoCidade> mapaGPS;
```

correspondências 1:1

**NOTA: ANTES DE JAVA5, AS COLECÇÕES NÃO ERAM PARAMETRIZADAS E ERAM TODAS COLECÇÕES DE TIPO `Object`. EM CONSEQUÊNCIA, QUALQUER INSTÂNCIA DE QUALQUER CLASSE PODERIA SER ADICIONADA A UMA COLECÇÃO (ERAM HETEROGÉNEAS).**

```
ArrayList cidades = new ArrayList();     // não há parâmetros (é Java2 !!)
cidades.add(new Ponto2D()); cidades.add(new Circulo());
```

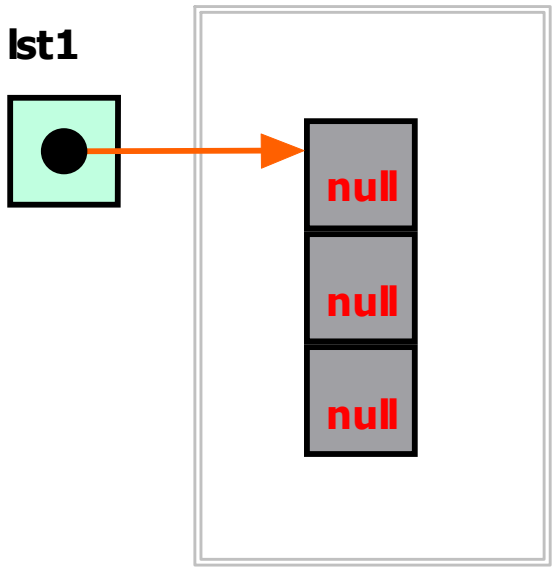
## TIPO GENÉRICO `ArrayList<E>`

**PERMITE CRIAR LISTAS DE OBJECTOS DE UM QUALQUER TIPO E, SENDO ARRAYS DINÂMICOS E DE TAMANHO ILIMITADO, E SENDO OS SEUS ELEMENTOS TODOS DO MESMO TIPO DEFINIDO, E INDEXADOS A PARTIR DO ÍNDICE 0;**

### DECLARAÇÃO E CRIAÇÃO

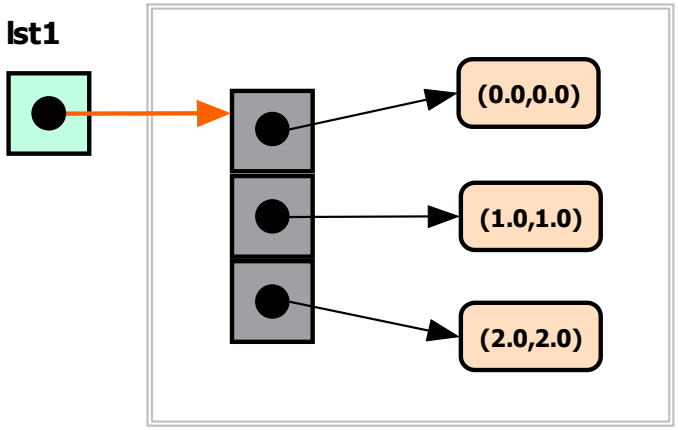
```
ArrayList<Ponto2D> lst1 = new ArrayList<Ponto2D>();   // construtor
```

**ArrayList<Ponto2D> inicialmente vazio. Espaço alocado mas sem elementos inseridos. Assim, todas as posições alocadas têm inicialmente o valor null**

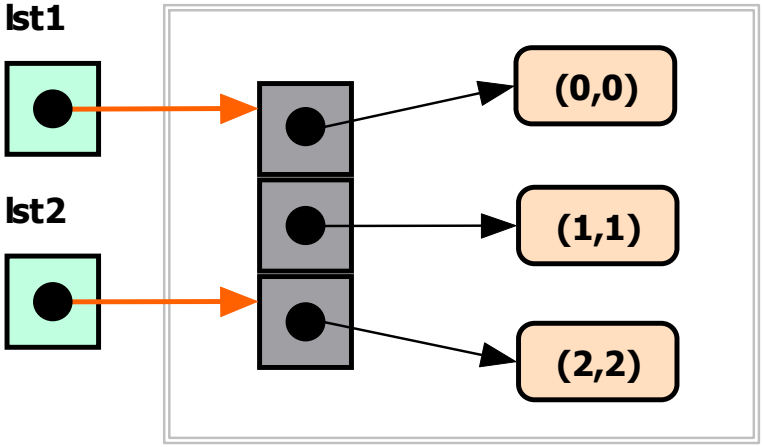



**ArrayList<Ponto2D>**

## Representação Interna de `ArrayList<Ponto2D>`




**ArrayList<Ponto2D>**




**Resultado de `lst2 = lst1 (partilha)`**

## API DE public class java.util.ArrayList<E>

```
/* CONSTRUTORES */
new ArrayList<E>()                  // capacidade inicial = 10; tamanho = 0
new ArrayList<E>(int capacidade)    // capacidade inicial dada
new ArrayList<E>(Collection c)      // valores copiados (partilha) de uma colecção dada

/* MÉTODOS DE INSTÂNCIA */
boolean add(E elem);                    boolean add(int index, E elem);
boolean addAll(Collection c);           boolean addAll(int index, Collection c);

E get(int index);                       E set(int index, E elem); // index <= this.size()

boolean contains(Object o);             boolean containsAll(Collection c);
int indexOf(Object o);                  int lastIndexOf(Object o);

boolean remove(int index);              boolean remove(Object o);
boolean removeAll(Collection c);        boolean removeRange(int from, int to);
boolean retainAll(Collection c);

int size(); void clear();               boolean isEmpty(); Object clone(); // por partilha

List<E> subList(int from, int to);

Iterator<E> iterator();
ListIterator<E> listIterator();

Object[] toArray();  // conversão da colecção num array de Object
```

**Nota:** ***Collection*** **significa uma qq. colecção compatível (a ver mais tarde)**

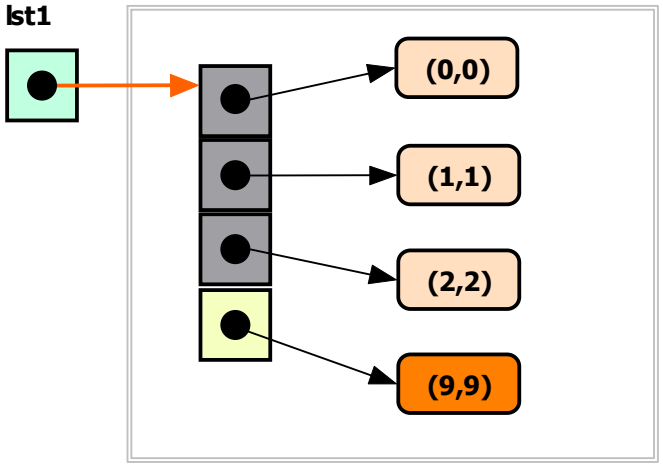
## CONSTRUTORES

```
ArrayList<Ponto2D> lst1 = new ArrayList<Ponto2D>();
ArrayList<Ponto2D> lst2 = new ArrayList<Ponto2D>(1000);
ArrayList<Ponto2D> lst3 = new ArrayList<Ponto2D>(lst1);
```

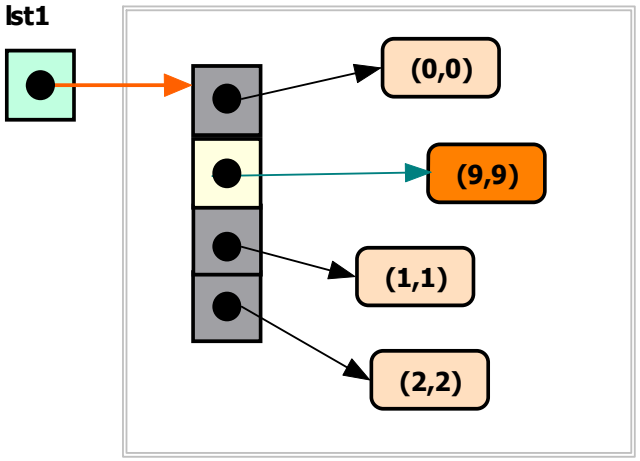
## SEMÂNTICA DOS MÉTODOS

```
boolean add(E elem);              // adiciona um elemento no fim do arraylist

boolean add(int index, E elem);   // adiciona no índice, reposicionando os outros
```

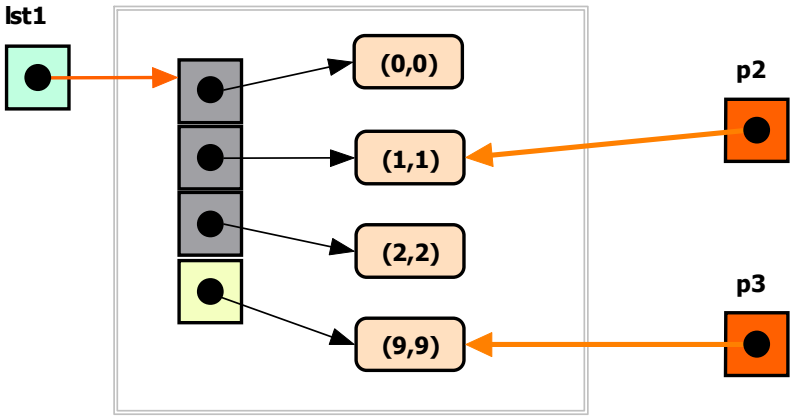



```
lst1.add(new Ponto2D(9.0,9.0));
```



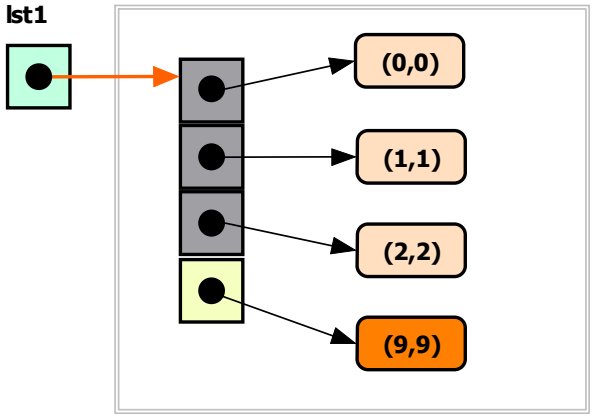

```
lst1.add(1, new Ponto2D(9.0,9.0))
```

```
E get(int index);  // devolve o elemento no índice dado [0 .. size()-1]
                   // de facto, como sabemos já, apenas é devolvido o endereço !!
```



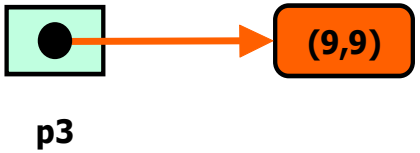

```
p2 = lst1.get(1)
p3 = lst1.get(3)
```

**SE NÃO PRETENDERMOS PARTILHA DEVEMOS FAZER UMA "CÓPIA" !!**



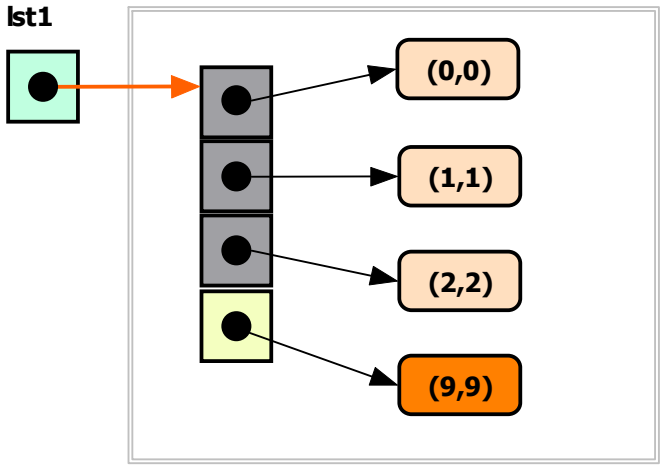

```
p2 = lst1.get(1).clone();
```

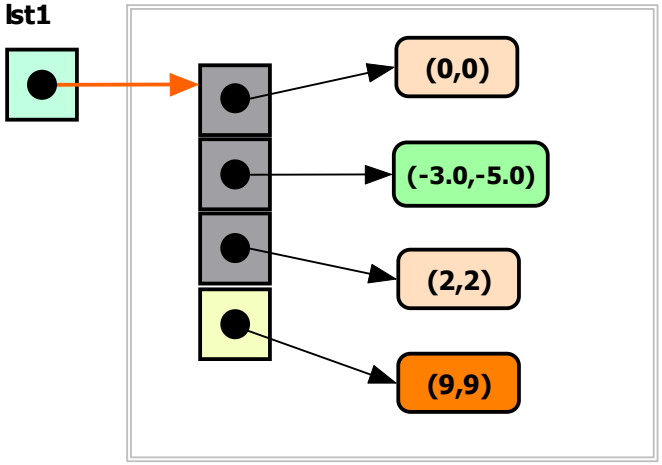

```
p3 = lst1.get(3).clone();
```




**NOTA: MANTÊM-SE AS REGRAS FUNDAMENTAIS DO ENCAPSULAMENTO CASO A COLECÇÃO SEJA UMA VARIÁVEL DE INSTÂNCIA. ASSIM, CLONE() ANTES DE INSERIR E CLONE() ANTES DE DEVOLVER.**

```
E set(int index, E elem);     // se índice válido, coloca o parâmetro nesse índice
                              // e dá como resultado o elemento removido ☹.
```




```
lst1.set(1, new Ponto2D(-3.0, -5.0));
```

**OU, CASO SEJA VIA VARIÁVEL,**

```
Ponto2D p = new Ponto2D(-3.0, -5.0);
……..……..
a)    lst1.set(1, p.clone());       // insere uma cópia (não partilha)
b)    p2 = lst1.set(1, p.clone()); // insere novo e guarda antigo
```


p2

## TESTES, PROCURAS E REMOÇÕES

### TESTES

```
int size();           // número de elementos
boolean isEmpty();    // vazio ?

boolean contains(Object o);           // verifica se elemento existe
boolean containsAll(Collection c);    // verifica se todos os elementos existem
```

### PROCURAS

```
int indexOf(Object o);                // índice da primeira ocorrência do elemento
int lastIndexOf(Object o);            // índice da última ocorrência do elemento
```

### REMOÇÕES

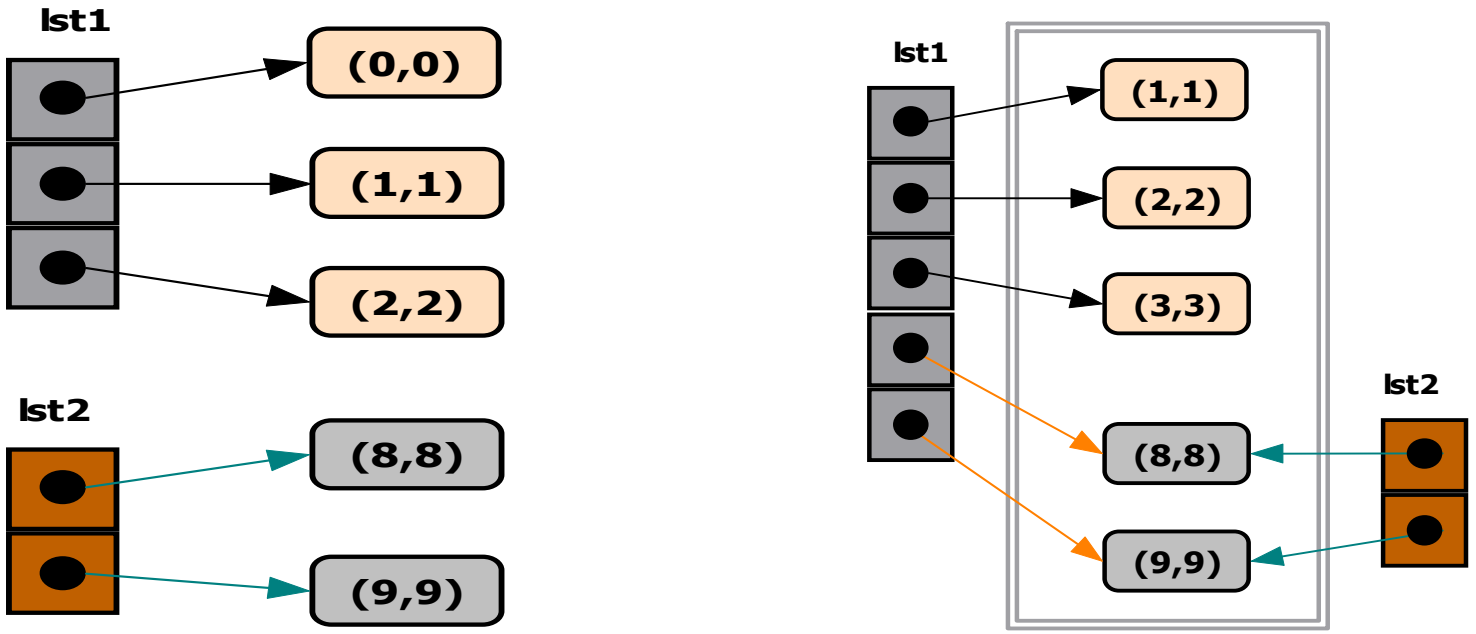
```
boolean remove(int index);            // remove o elemento no índice dado e reposiciona
boolean remove(Object o);             // remove o elemento dado, se encontrado
boolean removeAll(Collection c);      // remove todos os que pertencem à colecção dada

boolean removeRange(int from, int to); // remove todos entre dois índices
```

### SELECÇÃO

```
List<E> subList(int from, int to);    // devolve a sublista entre índices (partilhada!!)
```

```
boolean addAll(Collection c);              // adiciona os elementos da colecção parâmetro
boolean addAll(int index, Collection c);   // neste caso, a partir do índice dado
```




```
lst1.addAll(lst2);
```

**NOTA: OS ELEMENTOS ADICIONADOS NÃO SÃO COPIADOS, PELO QUE PASSAM A ESTAR PARTILHADOS PELAS DUAS COLECÇÕES. COMO SABEMOS, ISTO É MUITO PERIGOSO E, PORTANTO, `addAll()` NÃO DEVE SER USADO COM VARIÁVEIS DE INSTÂNCIA. SERÁ NECESSÁRIO FAZER `clone()` DOS ELEMENTOS A INSERIR.**

## ITERAÇÃO DE COLECÇÕES (VARRIMENTO) COM ITERADOR `for` (foreach de JAVA5)

```
for(E elem : coleccao<E>) {
  ...// fazer qq. coisa com o objecto em elem
}
```

**LER: COM CADA OBJECTO DA COLECÇÃO DE OBJECTOS DE TIPO E QUE PODE SER ATRIBUÍDO À VARIÁVEL elem, FAZER ...**

**EXEMPLOS:**

- `ArrayList<String> nomes = new ArrayList<String>();`
- `ArrayList<Ponto2D> linha = new ArrayList<Ponto2D>();`

```
// conta o número de nomes começados pela letra A
int conta = 0;
for(String nome : nomes)
    if(nome.charAt(0) == "A") conta++;

// coloca nomes linha a linha
StringBuilder linhas = new StringBuilder();
for(String nome : nomes) linhas.append(nome + "\n");
```

```java
// determina o comprimento médio dos nomes
int compTotal = 0; double compMedio = 0.0;
for(String nome : nomes) compTotal += nome.length();
compMedio = compTotal/(nomes.size());


// deslocar todos os pontos da linha de dx e dy
for(Ponto2D p : linha) p.incCoord(dx, dy);


// determinar o ponto de maior X
double max = Double.MIN_VALUE; Ponto2D ptMaiorX = null;
for(Ponto2D p : linha)
  if(p.getX() > max) { max = p.getX(); ptMaiorX = p.clone(); }


// criar um arraylist com todos os pontos com X maior que Y
ArrayList<Ponto2D> pontosXMY = new ArrayList<Ponto2D>();
for(Ponto2D p : linha)
  if(p.getX() > p.getY()) pontosXMY.add(p.clone());


// criar uma cópia do arraylist linha = clone()
ArrayList<Ponto2D> copiaLinha = new ArrayList<Ponto2D>();
for(Ponto2D p : linha) copiaLinha.add(p.clone());
```

## OUTROS ITERADORES DE COLECÇÕES

```
Iterator<E> iterator();  // método que cria um Iterador sobre a colecção
                         // que recebeu a mensagem. Se a colecção tem
                         // objectos de tipo E o Iterador é de tipo E
```

- UM `Iterator<E>` É UMA ESTRUTURA COMPUTACIONAL QUE IMPLEMENTA UM ITERADOR SOBRE TODOS OS ELEMENTOS DA COLECÇÃO ORIGEM (SEM ORDEM PREDEFINIDA), USANDO OS MÉTODOS `hasNext()`, `next()` E `remove()`.

## EXEMPLO 1: USO DE `iterator()` COM `while() { ... }`

```
Iterator<E> it = colecção.iterator();
E elem; // tipo dos objectos que estão na colecção
while(it.hasNext()) {
   elem = it.next();
     // fazer qq. coisa com elem
}
```

## EQUIVALENTE A :

```
for(E elem : colecção) { … }
```

**NOTA: Mas *foreach* não permite parar as *procuras* !!**

## Assim, em algoritmos típicos de procura tem que se usar a construção baseada em `iterator()`, cf.

```
Iterator<E> it = colecção.iterator();   // criar o Iterator<E>
E elem ;  // variável do tipo dos objectos da colecção
boolean encontrado = false;
while(it.hasNext() && !encontrado) {
    elem = it.next();
    if(elem possui certa propriedade) encontrado = true;
}
```

## Exemplo:

```
// encontrar o 1º ponto com coordenada X igual a Y
Ponto2D pontoCopia = null;
Iterator<Ponto2D> it = linha.iterator();   // criar o Iterator<E>
Ponto2D ponto ;  // variável do tipo dos objectos da colecção
boolean encontrado = false;
while(it.hasNext() && !encontrado) {
    ponto = it.next();
    if(ponto.getX() == ponto.getY()) encontrado = true;
}
if(encontrado) pontoCopia = ponto.clone();
```

## EXEMPLO 2: USO DE iterator() COM CICLO for() (desaconselhada)

```
  E elem;  // tipo  dos objectos guardados no ArrayList<E>
  for(Iterator<E> it = colecção.iterator(); it.hasNext();) {
      elem = it.next();
        // usar elem para qq. coisa
   }
```

```
ArrayList<String> nomes = new ArrayList<String>(); …

String nome; int conta = 0;
for(Iterator<String> it = nomes.iterator(); it.hasNext();) {
    nome = it.next();
    if(nome.length() > 10) conta++;
}
```

QUE É EQUIVALENTE À FORMA MAIS SIMPLES,

```
int conta = 0;
for(String nome : nomes) if( nome.length()>10 ) conta++;
```

## IMPORTANTE:

**A API de `Set<E>`, ou seja, dos conjuntos, é um SUBCONJUNTO da API de `List<E>` porque os conjuntos não possuem operações usando índices dado não serem indexados. No restante, todos os métodos possuem os mesmos nomes e idêntica semântica (claro que conjuntos não admitem duplicados).**

**O que se pode fazer com um conjunto de objectos do tipo `HashSet<E>` ou `TreeSet<E>`?**

**public class java.util. HashSet**

**/* Construtores */**

**HashSet<E>()**
**HashSet<E>(*Collection* c)**
**HashSet<E>(int dim)**

**/* Métodos de Instância */**

**public boolean add(E elem)**
**public boolean addAll(*Collection* c)**
**public void clear()**
**public boolean contains(Object o)**
**public boolean containsAll(*Collection* c)**
**public boolean equals(Object o)**
**public boolean isEmpty()**
**public Iterator<E> iterator()**
**public boolean remove(Object o)**
**public boolean removeAll(*Collection* c)**
**public boolean retainAll(*Collection* c)**
**public int size()**
**public Object[ ] to Array()**

Ou seja, com a excepção dos métodos que usam índices, temos exactamente os mesmos métodos que vimos já sobre os `ArrayList<E>`

Assim, é só declarar e usar, cf.

```
HashSet<Ponto2D> linha = new HashSet<Ponto2D>();
TreeSet<String> nome = new TreeSet<String>();

linha.add( new Ponto2D(1.0, 2.0) );

nomes.add("Pedro"); nomes.add("Rita");

for(String nm : nomes) out.println(nm);
```