

Ainda sobre classes

- “*The concepts of a class and an object are tightly interwoven, for we cannot talk about an object without regard for its class. However, there are important differences between these two terms.* Whereas an object is a concrete entity that exists in time and space, a class represents only an abstraction, the “essence” of an object, as it were.
- a definição de classe que temos vindo a utilizar está incompleta
 - quer as instâncias, quer as classes são **objectos**

Variáveis e métodos de classe

- as classes não deixam de ser objectos
 - objectos que guardam o que é comum a todas as instâncias
 - **apenas um** objecto-classe por classe
 - se objectos possuem estado e comportamento, então podemos extrapolar e dizer que a classe também tem:
 - variáveis e métodos de classe

- os métodos de classe são activados a partir de mensagens que são enviadas para a classe
- exemplo: Ponto.metodoClasse()
- se uma classe possui variáveis de classe o acesso a essas variáveis deverá ser feito através dos métodos de classe
 - métodos de instância => var. instância
 - métodos de classe => var. classe

- o que é que se pode guardar como variável de classe?
- valores que sejam comuns a todas os objectos instância
- não faz sentido colocar estes valores em todos os objectos (repetição)

- Imagine-se que numa classe Factura se pretende:
 - a) saber quantas facturas (quantas instâncias) foram criadas (é possível saber quantas existem?)
 - b) definir uma taxa de imposto (ex: IVA) comum a todas as facturas
- como é que poderemos satisfazer os requisitos expressos em a) e b)?

- uma variável que guarde o número de facturas criadas não é certamente uma variável de instância
- actualização do contador em todas as instâncias?
- redundância?
- teriam de se ter implementados mecanismos (complexos) de comunicação entre todas as instâncias!!

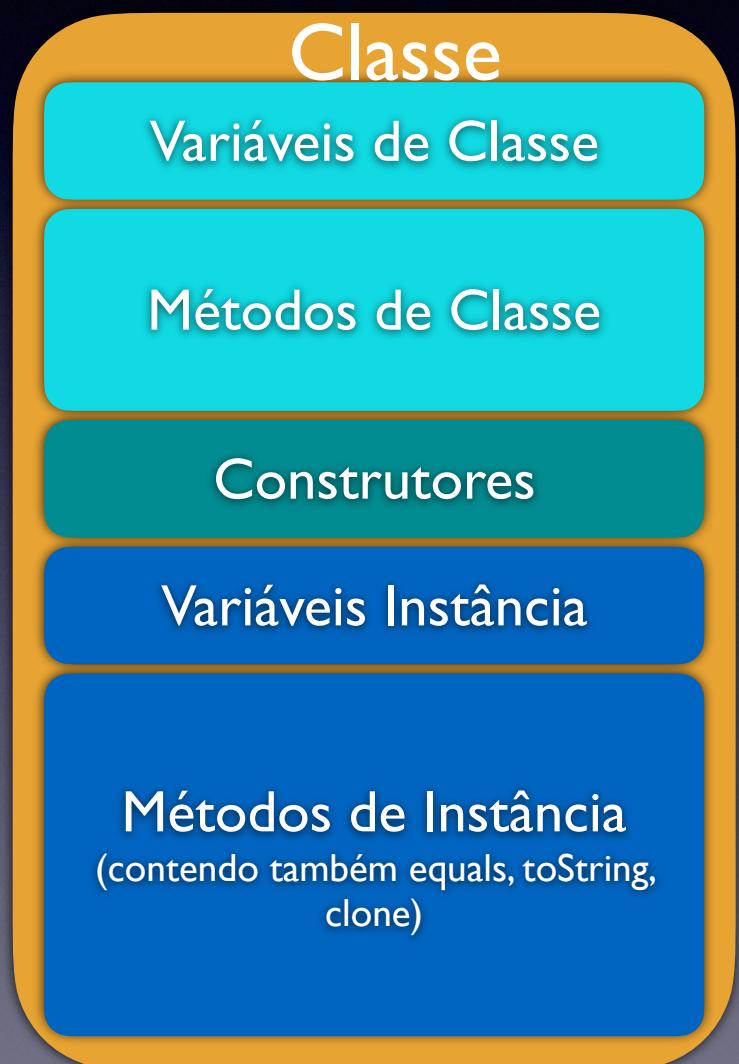
- as variáveis de classe servem para guardar *informação global* a todas as instâncias
- podem também ser utilizadas para guardar constantes que são utilizadas pelos diversos objectos instância
 - como já vimos, por em exemplo com Math.PI (e outros)

- os métodos de classe fazem o acesso às variáveis de classe
- aos métodos de classe aplicam-se as mesmas regras de visibilidade que se aplicam aos métodos de instância
- os métodos de classe são sempre acessíveis às instâncias, mas métodos de classe **não** tem acesso aos métodos de instância

- como se declaram métodos e variáveis de classe?
 - utilizando o prefixo **static**
- a definição de classe passa a ter:
 - declaração de variáveis de classe
 - declaração de métodos de classe
 - declaração de variáveis de instância
 - declaração de métodos de instância

Estrutura de uma classe

- estrutura tipo de uma classe
- para que possa ser utilizada pelos potenciais clientes
- esta definição irá ser refinada ao longo do semestre

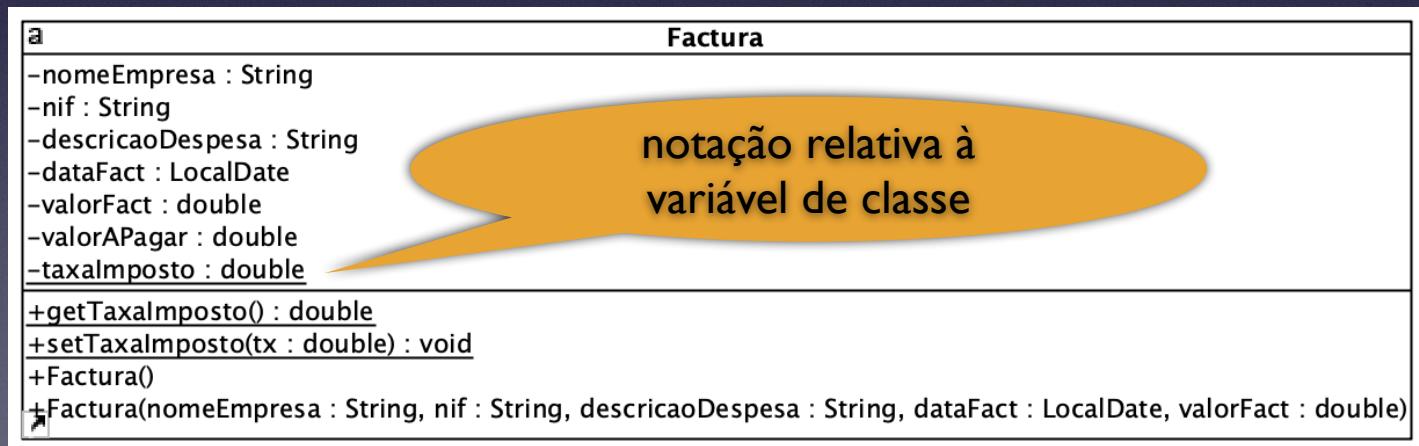


Classe Factura (exemplo)

- seja uma classe Factura, que modela de forma muito simples uma factura
- queremos representar a informação que permitirá ao criar uma factura ajustar o valor ao imposto que na altura está em vigor.
- terá de ser uma definição global a todas as instâncias

Classe Factura

- Representação em Diagrama de Classe



- uma variável de classe e respectivos métodos

```
public class Factura {  
    //variáveis de instância  
    private String nomeEmpresa;  
    private String nif;  
    private String descricaoDespesa;  
    private LocalDate dataFact;  
    private double valorFact;  
    private double valorAPagar; //pelo cliente, depois de aplicado o imposto  
  
    //a taxa de imposto é definida globalmente para todas  
    //as facturas e na altura da emissão deve utilizar-se  
    //a taxa de imposto actual.  
    private static double taxaImposto;  
  
    public static double getTaxaImposto() {  
        return taxaImposto;  
    }  
  
    public static void setTaxaImposto(double tx) {  
        taxaImposto = tx;  
    }  
}
```

- o valor a pagar é calculado aquando da invocação do construtor

```
public Factura(String nomeEmpresa, String nif, String descricaoDespesa,  
               LocalDate dataFact, double valorFact) {  
    this.nomeEmpresa = nomeEmpresa;  
    this.nif = nif;  
    this.descricaoDespesa = descricaoDespesa;  
    this.dataFact = dataFact;  
    this.valorFact = valorFact;  
    this.valorAPagar = this.valorFact * (1 + getTaxaImposto());  
}
```

- o método de classe utilizado no construtor poderia ter sido invocado,
- enviando o método ao objecto onde está definido, a classe Factura
- na forma:

```
this.valorAPagar = this.valorFact * (1 + Factura.getTaxaImposto());
```

- Nota: ajuda seguir a convenção que diz que as classes começam por letra maiúscula

Estruturas de Dados

- Como já vimos atrás, podemos criar conceitos mais complexos através do mecanismo de composição/agregação de objectos de classes mais simples:
 - a Turma a partir de Aluno
 - o Stand a partir de Veículo
 - etc.
- Mas para isso precisamos de ter colecções de objectos...

- Até ao momento apenas temos disponível a utilização de arrays

```
Aluno[] alunos = new Aluno[30];
Veiculo[] carros = new Veiculo[10];

for (int i=0; i<alunos.length && !encontrado; i++)
    if (alunos[i].getNota() == 20)
        encontrado = true;

for(Veiculo v: carros)
    System.out.println(v.toString());
```

- Esta é uma solução simples e testada, com o inconveniente de o tamanho da estrutura de dados ser estaticamente definido.
- Será que conseguimos ter uma estrutura de dados, baseada em arrays, que pudesse crescer de forma transparente para o utilizador?
 - (primeira solução) Sim, bastando para tal criarmos uma classe com esse comportamento

- Seja essa classe chamada de `GrowingArray` e, por comodidade, vamos utilizar instâncias de `Circulo`
- Que operações necessitamos:
 - adicionar um círculo (no fim e numa posição)
 - remover um círculo
 - ver se um círculo existe
 - dar a posição de um círculo na estrutura
 - dar número de elementos existentes

● Documentação com os métodos necessários:

Constructor Summary	
Constructors	
Constructor and Description	
<code>GrowingArray()</code>	
<code>GrowingArray(int capacidade)</code>	
Method Summary	
All Methods	Instance Methods
Concrete Methods	
Modifier and Type	
void	
<code>add(MyDynamicArray.Circulo c)</code>	
Adiciona o elemento passado como parâmetro ao fim do array	
void	
<code>add(int indice, MyDynamicArray.Circulo c)</code>	
Método que adiciona um elemento numa determinada posição, forçando a que os elementos à direita no array façam shift.	
boolean	
<code>contains(MyDynamicArray.Circulo c)</code>	
Método que determina se um elemento está no array.	
MyDynamicArray.Circulo	
<code>get(int indice)</code>	
Devolve o elemento que está na posição indicada.	
int	
<code>indexof(MyDynamicArray.Circulo c)</code>	
Método que determina o índice do array onde está localizada a primeira ocorrência de um objecto.	
boolean	
<code>isEmpty()</code>	
Método que determina se o array contém elementos, ou se está vazio.	
boolean	
<code>remove(MyDynamicArray.Circulo c)</code>	

Composição ou agregação?

- Ao criar esta estrutura temos de tomar a decisão se a colecção de elementos assenta em composição ou agregação.
 - Quais são as implicações?
- Se for composição os métodos de inserção e get terão de prever a utilização de clone
 - é a solução adequada para todas as situações?

Composição ou agregação?

- Se a estratégia de construção for do tipo Composição, esta coleção terá sempre este comportamento.
- não poderá ser utilizada em estratégias de agregação.
- os métodos de get e set, adição e recuperação (e os construtores) farão sempre clone e gerarão novas referências.

- Declarações iniciais:

```
public class GrowingArray {  
  
    private Circulo[] elementos;  
    private int size;  
  
    /**  
     * variável que determina o tamanho inicial do array,  
     * se for utilizado o construtor vazio.  
     */  
    private static final int capacidade_inicial = 20;  
  
    public GrowingArray(int capacidade) {  
        this.elementos = new Circulo[capacidade];  
        this.size = 0;  
    }  
  
    public GrowingArray() {  
        this(capacidade_inicial);  
    }  
}
```



invocação do construtor anterior

- get de um elemento da estrutura de dados

```
/**  
 * Devolve o elemento que está na posição indicada.  
 *  
 * @param indice posição do elemento a devolver  
 * @return o objecto que está na posição indicada no parâmetro  
 * (deveremos ter atenção às situações em que a posição não existe)  
 */  
  
public Circulo get(int indice) {  
    if (indice <= this.size)  
        return this.elementos[indice];  
    else  
        return null; // ATENÇÃO!  
}
```

- set de uma posição da estrutura

```
/**  
 * Método que actualiza o valor de uma determinada posição do array.  
 *  
 * @param indice a posição que se pretende actualizar  
 * @param c o circulo que se pretende colocar na estrutura de dados  
 *  
 */  
public void set(int indice, Circulo c) {  
    if (indice <= this.size) //não se permitem "espaços vazios"  
        this.elementos[indice] = c;  
}
```

- se a estratégia de utilização for de composição, quem chama este método deve previamente fazer clone.

- adicionar um elemento à estrutura de dados

```
/**  
 * Adiciona o elemento passado como parâmetro ao fim do array  
 *  
 * @param c circulo que é adicionado ao array  
 *  
 */  
  
public void add(Circulo c) {  
    aumentaCapacidade(this.size + 1);  
    this.elementos[this.size++] = c;  
}
```

- método auxiliar que aumenta espaço

```
/**  
 * Método auxiliar que verifica se o array alocado tem capacidade  
 * para guardar mais elementos.  
 * Por cada nova inserção, verifica se estamos a mais de metade  
 * do espaço  
 * alocado e, caso se verifique, aloca mais 1.5 de capacidade.  
 */  
  
private void aumentaCapacidade(int capacidade) {  
    if (capacidade > 0.5 * this.elementos.length) {  
        int nova_capacidade = (int)(this.elementos.length * 1.5);  
        this.elementos = Arrays.copyOf(this.elementos, nova_capacidade);  
    }  
}
```

```
/**  
 * Método que adiciona um elemento numa determinada posição,  
 * forçando a  
 * que os elementos à direita no array façam shift.  
 * Tal como no método de set não são permitidos espaços.  
 *  
 * @param indice indice onde se insere o elemento  
 * @param c circulo que será inserido no array  
 *  
 */  
  
public void add(int indice, Circulo c) {  
    if (indice <= this.size) {  
        aumentaCapacidade(this.size+1);  
        System.arraycopy(this.elementos, indice, this.elementos,  
                         indice + 1, this.size - indice);  
        this.elementos[indice] = c;  
        this.size++;  
    }  
}
```

```
/**  
 * Remove do array o elemento que está na posição indicada no parâmetro.  
 * Todos os elementos à direita do índice sofrem um deslocamento  
 * para a esquerda.  
 * @param indice índice do elemento a ser removido  
 * @return o elemento que é removido do array. No caso do índice não  
 * existir devolver-se-á null.  
 */  
  
public Circulo remove(int indice) {  
    if (indice <= this.size) {  
        Circulo c = this.elementos[indice];  
  
        int deslocamento = this.size - indice - 1;  
        if (deslocamento > 0)  
            System.arraycopy(this.elementos, indice+1, this.elementos,  
                            indice, deslocamento);  
        this.elementos[--this.size] = null;  
        return c;  
    }  
    else  
        return null;  
}
```

```
* Remove a primeira ocorrência do elemento que é passado como parâmetro.  
* Devolve true caso o array contenha o elemento, falso caso contrário.  
*  
* @param c círculo a ser removido do array (caso exista)  
* @return true, caso o círculo exista no array  
*/  
public boolean remove(Circulo c) {  
    if (c != null) {  
        boolean encontrado = false;  
        for (int indice = 0; indice < this.size && !encontrado; indice++)  
            if (c.equals(this.elementos[indice])) {  
                encontrado = true;  
                int deslocamento = this.size - indice - 1;  
                if (deslocamento > 0)  
                    System.arraycopy(this.elementos, indice+1,  
                                     this.elementos, indice, deslocamento);  
                this.elementos[--this.size] = null;  
            }  
        return encontrado;  
    }  
    else  
        return false;  
}
```

invocação do
método equals de Circulo

faz shift left

```
/**  
 * Método que determina o índice do array onde está localizada a  
 * primeira ocorrência de um objecto.  
 *  
 * @param c círculo de que se pretende determinar a posição  
 * @return a posição onde o círculo se encontra. -1 caso não esteja no  
 * array ou o círculo passado como parâmetro seja null.  
 */  
  
public int index0f(Circulo c) {  
    int posicao = -1;  
    if (c != null) {  
        boolean encontrado = false;  
        for (int i = 0; i < this.size && !encontrado; i++)  
            if (c.equals(this.elementos[i])) {  
                encontrado = true;  
                posicao = i;  
            }  
    }  
    return posicao;  
}
```

invocação do
método equals de Circulo

```
/**  
 * Método que determina se um elemento está no array.  
 *  
 * @param c círculo a determinar se está no array  
 * @return true se o objecto estiver inserido na estrutura de dados,  
 * false caso contrário.  
 */  
public boolean contains(Circulo c) {  
    return indexOf(c) >= 0;  
}  
  
/**  
 * Método que determina se o array contém elementos, ou se está vazio.  
 *  
 * @return true se o array estiver vazio, false caso contrário.  
 */  
public boolean isEmpty() {  
    return this.size == 0;  
}
```

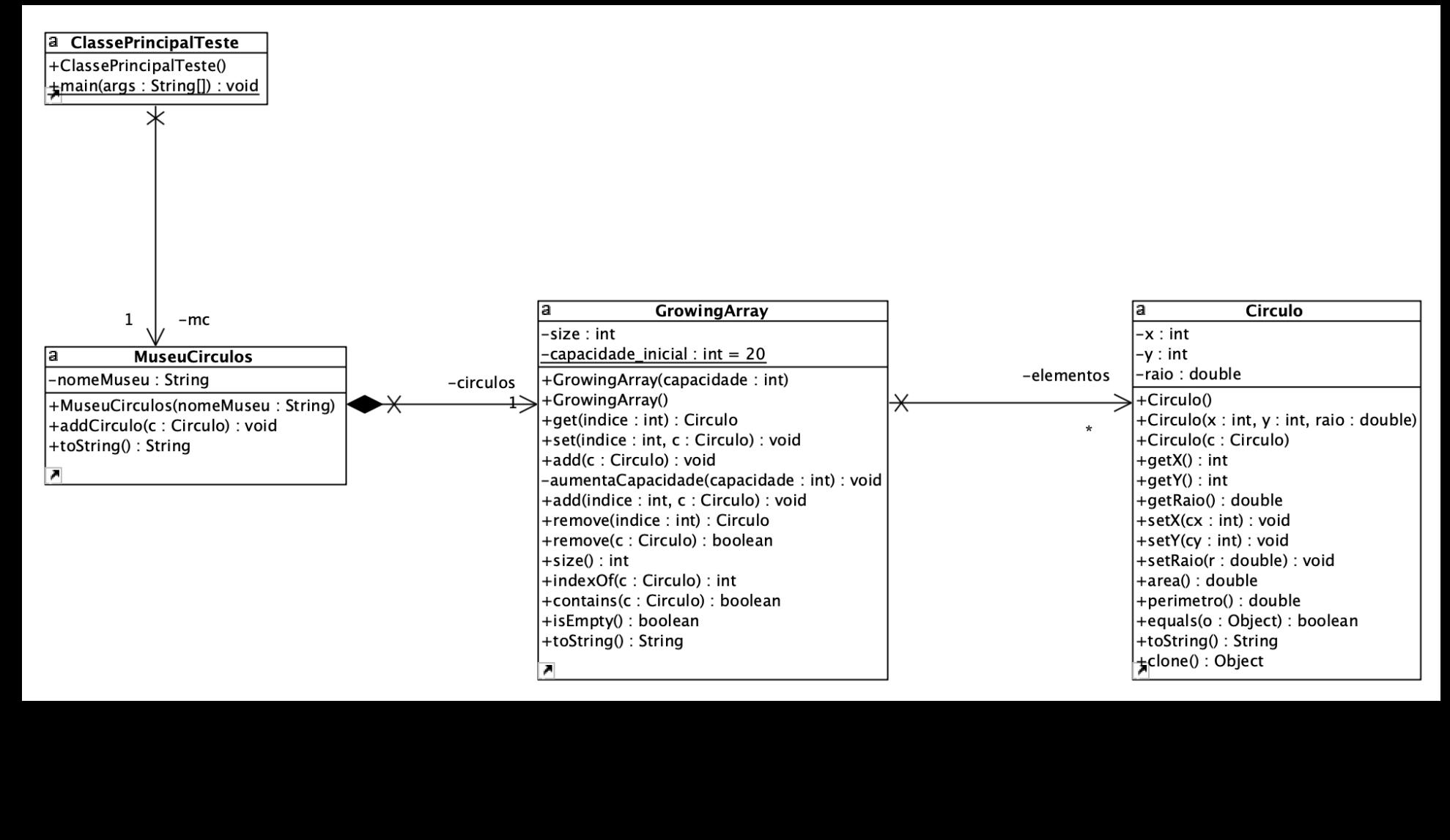
```
public class TesteGA {  
    public static void main(String[] args) {  
  
        Circulo c1 = new Circulo(2,4,4.5);  
        Circulo c2 = new Circulo(1,4,1.5);  
        Circulo c3 = new Circulo(2,7,2.0);  
        Circulo c4 = new Circulo(3,3,2.0);  
        Circulo c5 = new Circulo(2,6,7.5);  
  
        GrowingArray ga = new GrowingArray(10);  
        ga.add(c1.clone());  
        ga.add(c2.clone());  
        ga.add(c3.clone());  
  
        System.out.println("Num elementos = " + ga.size());  
        System.out.println("Posição do c2 = " + ga.indexOf(c2));  
  
    }  
}
```

estratégia de
composição

A utilização de GrowingArray

- Imagine-se que queremos criar uma aplicação em que temos uma objecto que tem uma coleção de círculos:
 - A classe GrowingArray é a estrutura de dados
 - Temos de ter uma classe que implemente a lógica de negócio e essa classe é que tem a v.i. do tipo GrowingArray
 - Precisamos de ter uma classe de teste com o método main

Diagrama de classe



```
public class MuseuCirculos {  
    private String nomeMuseu;  
    private GrowingArray circulos;  
  
    public MuseuCirculos(String nomeMuseu) {  
        this.nomeMuseu = nomeMuseu;  
        this.circulos = new GrowingArray();  
    }  
  
    public void addCirculo(Circulo c) {  
        this.circulos.add(c.clone());  
    }  
  
    public String toString() {  
        StringBuilder sb = new StringBuilder();  
        sb.append("Museu de Circulos --> ");  
        sb.append("Nome : ");  
        sb.append(this.nomeMuseu + "\n");  
        sb.append(this.circulos.toString());  
  
        return sb.toString();  
    }  
}
```

```
public class ClassePrincipalTeste {  
  
    public static void main(String[] args) {  
        MuseuCirculos mc = new MuseuCirculos("Circulos de P00");  
  
        Circulo c1 = new Circulo(2, 4, 4.5);  
        Circulo c2 = new Circulo(1, 4, 1.5);  
        Circulo c3 = new Circulo(2, 7, 2.0);  
        Circulo c4 = new Circulo(3, 3, 2.0);  
        Circulo c5 = new Circulo(2, 6, 7.5);  
  
        mc.addCirculo(c1); // internamente mc vai fazer cópia  
        mc.addCirculo(c2);  
        mc.addCirculo(c3);  
        mc.addCirculo(c4);  
        mc.addCirculo(c5);  
  
        System.out.println("== toString do museu ==");  
        System.out.println(mc.toString());  
    }  
}
```

Colecções Java

- O Java oferece um conjunto de classes que implementam as estruturas de dados mais utilizadas
 - oferecem uma API consistente entre si
 - permitem que sejam utilizadas com qualquer tipo de objecto - são parametrizadas por tipo

- Poderemos representar:
 - `ArrayList<Aluno> alunos`
 - `HashSet<Aluno> alunos;`
 - `HashMap<String, Aluno> turmaAlunos;`
 - `TreeMap<String, Docente> docentes;`
 - `Stack<Pedido> pedidosTransferência;`
 - ...

- Ao fazer-se `ArrayList<Aluno>` passa a ser o compilador a testar, e validar, que só são utilizados objectos do tipo `Aluno` no `ArrayList`.
- isto dá uma segurança adicional aos programas, pois em tempo de execução não teremos erros de compatibilidade de tipos
- os tipos de dados são verificados em tempo de compilação

- As colecções em Java beneficiam de:
 - auto-boxing e auto-unboxing, ie, a capacidade de converter automaticamente tipos primitivos para instâncias de classes wrapper.
 - int para Integer, double para Double, etc.
 - o programador não tem de codificar a transformação

- tipos genéricos
 - as colecções passam a ser definidas em função de um tipo de dados que é associado aquando da criação
 - a partir daí o compilador passa a garantir que os conteúdos da colecção são do tipo esperado

Collections Framework

- O Java Collections Framework agrupa as várias classes genéricas que correspondem às implementações de referência de:
 - Listas: API de List<E>
 - Conjuntos: API de Set<E>
 - Correspondências unívocas: API de Map<K,V>



Abstração de implementação!

A definição Collection

- Da documentação: “*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.*”
- “*All general-purpose Collection implementation classes (...) should provide two "standard" constructors: a void (no arguments) constructor, which creates an empty collection, and a constructor with a single argument of type Collection, which creates a new collection with the same elements as its argument.*”

- E ainda:
 - “*Many methods in Collections Framework interfaces are defined in terms of the equals method. For example, the specification for the contains(Object o) method says: "returns true if and only if this collection contains at least one element e such that (o==null ? e==null : o.equals(e))."*

AAPI de Collection

boolean	<code>add(E e)</code>	Ensures that this collection contains the specified element (optional operation).
boolean	<code>addAll(Collection<? extends E> c)</code>	Adds all of the elements in the specified collection to this collection (optional operation).
void	<code>clear()</code>	Removes all of the elements from this collection (optional operation).
boolean	<code>contains(Object o)</code>	Returns <code>true</code> if this collection contains the specified element.
boolean	<code>containsAll(Collection<?> c)</code>	Returns <code>true</code> if this collection contains all of the elements in the specified collection.
boolean	<code>equals(Object o)</code>	Compares the specified object with this collection for equality.
int	<code>hashCode()</code>	Returns the hash code value for this collection.
boolean	<code>isEmpty()</code>	Returns <code>true</code> if this collection contains no elements.
<code>Iterator<E></code>	<code>iterator()</code>	Returns an iterator over the elements in this collection.
default <code>Stream<E></code>	<code>parallelStream()</code>	Returns a possibly parallel <code>Stream</code> with this collection as its source.
boolean	<code>remove(Object o)</code>	Removes a single instance of the specified element from this collection, if it is present (optional operation).
boolean	<code>removeAll(Collection<?> c)</code>	Removes all of this collection's elements that are also contained in the specified collection (optional operation).
default boolean	<code>removeIf(Predicate<? super E> filter)</code>	Removes all of the elements of this collection that satisfy the given predicate.
boolean	<code>retainAll(Collection<?> c)</code>	Retains only the elements in this collection that are contained in the specified collection (optional operation).
int	<code>size()</code>	Returns the number of elements in this collection.
default <code>Spliterator<E></code>	<code>spliterator()</code>	Creates a <code>Spliterator</code> over the elements in this collection.
default <code>Stream<E></code>	<code>stream()</code>	Returns a sequential <code>Stream</code> with this collection as its source.
<code>Object[]</code>	<code>toArray()</code>	Returns an array containing all of the elements in this collection.

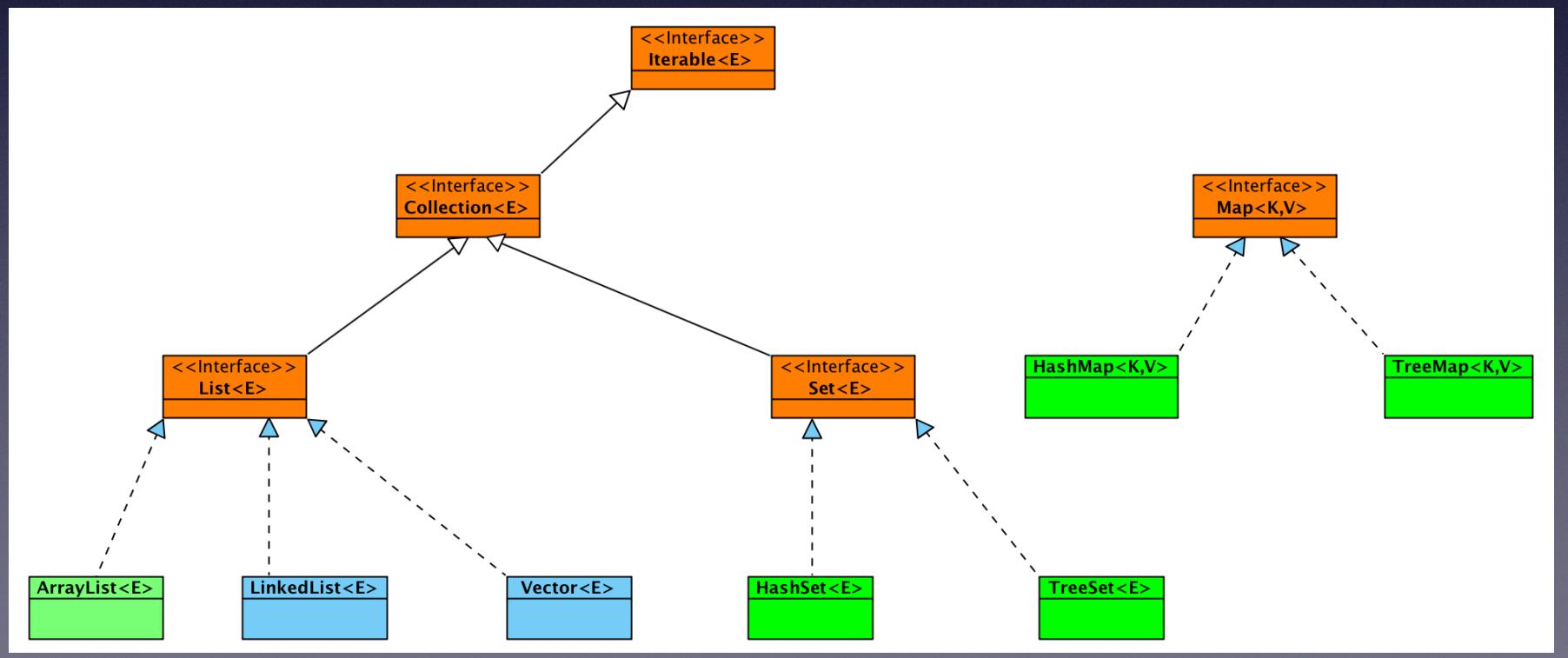
Estrutura da JCF

- Existe uma arrumação por API (interfaces)

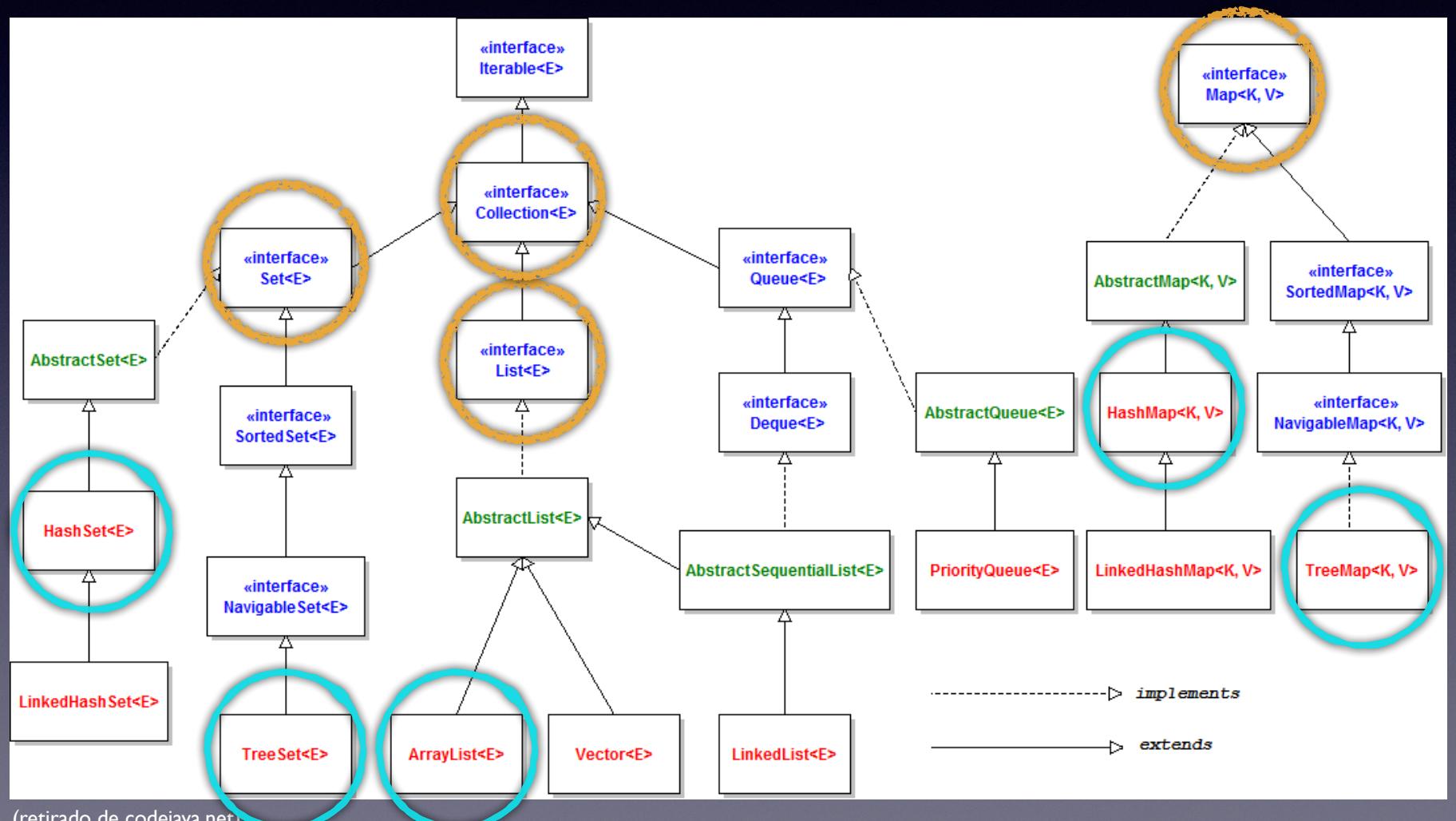


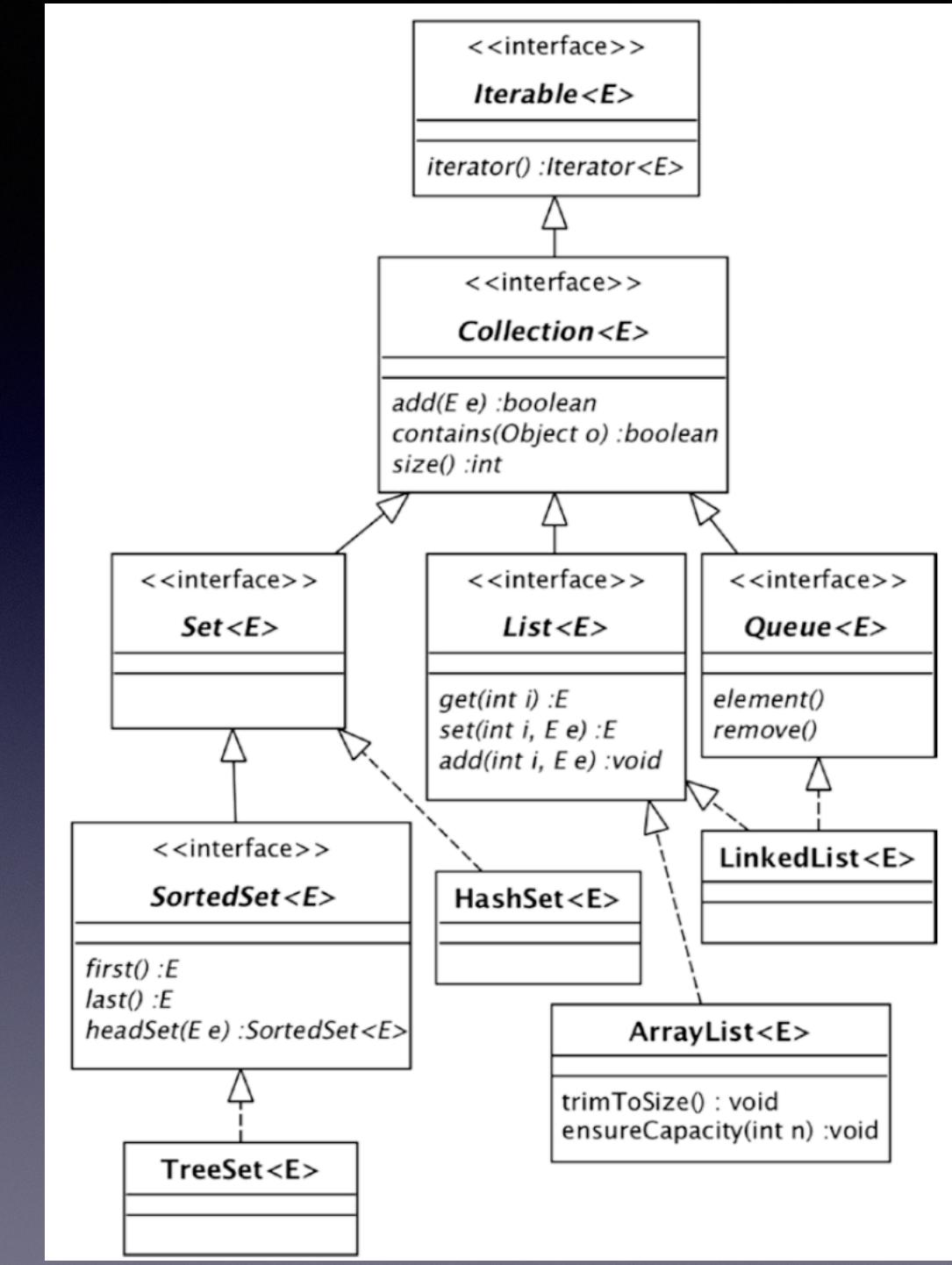
- Todas as colecções, `Collection<E>`, são iteráveis externamente através de `Iterable<E>`

- Para cada API (interface) existem diversas implementações (a escolher consoante critérios do programador)
- *Vamos estudar as que estão a verde*



Interfaces e Implementações do JFC





- todas as coleções permitem ser iteradas
- implementações de `List<E>` adicionam métodos com indicação de posição

(retirado de Java Program Design, Edward Sciore)

ArrayList<E>

- As classes da Java Collections Framework são exemplos muito interessantes de codificação
- Como o código destas classes está escrito em Java é possível ao programador observar como é que foram implementadas

ArrayList<E>: v.i. e construtores

```
public class ArrayList<E> extends AbstractList<E>
    implements List<E>, RandomAccess, Cloneable, java.io.Serializable
{
    private static final long serialVersionUID = 8683452581122892189L;

    /**
     * The array buffer into which the elements of the ArrayList are stored.
     * The capacity of the ArrayList is the length of this array buffer.
     */
    private transient Object[] elementData;

    /**
     * The size of the ArrayList (the number of elements it contains).
     *
     * @serial
     */
    private int size;

    /**
     * Constructs an empty list with the specified initial capacity.
     *
     * @param initialCapacity the initial capacity of the list
     * @throws IllegalArgumentException if the specified initial capacity
     *         is negative
     */
    public ArrayList(int initialCapacity) {
        ...
        this.elementData = new Object[initialCapacity];
    }

    /**
     * Constructs an empty list with an initial capacity of ten.
     */
    public ArrayList() {
        this(10);
    }
}
```

ArrayList<E>: existe?

```
public boolean contains(Object o) {  
    return indexOf(o) >= 0;  
}  
  
public int indexOf(Object o) {  
    if (o == null) {  
        for (int i = 0; i < size; i++)  
            if (elementData[i]==null)  
                return i;  
    } else {  
        for (int i = 0; i < size; i++)  
            if (o.equals(elementData[i]))  
                return i;  
    }  
    return -1;  
}
```

Espera-se que
o equals esteja
(bem) definido!

ArrayList<E>: inserir

```
public boolean add(E e) {
    ensureCapacityInternal(size + 1); // Increments modCount!!
    elementData[size++] = e;
    return true;
}

public void add(int index, E element) {
    rangeCheckForAdd(index);

    ensureCapacityInternal(size + 1); // Increments modCount!!
    System.arraycopy(elementData, index, elementData, index + 1,
                     size - index);
    elementData[index] = element;
    size++;
}
```

ArrayList<E>: get e set

```
public E get(int index) {  
    rangeCheck(index);  
  
    return elementData(index);  
}  
  
public E set(int index, E element) {  
    rangeCheck(index);  
  
    E oldValue = elementData(index);  
    elementData[index] = element;  
    return oldValue;  
}
```

Colecções Java

- Tipos de colecções disponíveis:
 - listas (definição em List<E>)
 - conjuntos (definição em Set<E>)
 - queues (definição em Queue<E>)
- noção de família (muito evidente) nas APIs de cada um destes tipos de colecções.

List<E>

- Utilizar sempre que precise de manter ordem
- O método add não testa se o objecto existe na coleção (admite repetições)
- O método contains verifica sempre o resultado de equals
- Implementação utilizada: **ArrayList<E>**

ArrayList<E>

Construtores	<code>ArrayList<E>(); ArrayList<E>(int initialCapacity)</code>
Adicionar elementos	<code>boolean add(E e); void add(int index, E element); boolean addAll(Collection c); boolean addAll(int index, Collection c)</code>
Remover elementos	<code>boolean remove(Object o); E remove(int index); boolean removeAll(Collection c) boolean retainAll(Collection c) boolean removeIf(Predicate p)</code>
Consultar	<code>E get(int index); int indexOf(Object o); int lastIndexOf(Object o); boolean contains(Object o) boolean containsAll(Collection c) boolean isEmpty(); int size()</code>
Alterar elementos	<code>E set(int index, E element); void clear();</code>
Iteradores externos	<code>Iterator<E> iterator()</code>
Iteradores internos	<code>Stream<E> stream(); void forEach(Consumer c)</code>
Outros	<code>boolean equals(Object o); T[] toArray(T[] a)</code>

```
import java.util.ArrayList;
public class TesteArrayList {
    public static void main(String[] args) {

        Circulo c1 = new Circulo(2,4,4.5);
        Circulo c2 = new Circulo(1,4,1.5);
        Circulo c3 = new Circulo(2,7,2.0);
        Circulo c4 = new Circulo(3,3,2.0);
        Circulo c5 = new Circulo(2,6,7.5);

        ArrayList<Circulo> circs = new ArrayList<Circulo>();
        circs.add(c1.clone());
        circs.add(c2.clone());
        circs.add(c3.clone());

        System.out.println("Num elementos = " + circs.size());
        System.out.println("Posição do c2 = " + circs.indexOf(c2));

        for(Circulo c: circs)
            System.out.println(c.toString());
    }
}
```

estratégia de
composição