Estruturas de Dados
2023/2024

Aula 01

- Plataforma de Colecções
- *Generics*
- *Wildcards*
- Explorar o `java.util.ArrayList`

Ricardo Santos
rjs@estg.ipp.pt

ESTG POLITÉCNICO DO PORTO

Introdução

- Vamos começar por estudar a interface **`Collection`** da *Java Collections Framework* JCF) que foi usada no ano passado
- Nesta aula serão analisadas questões relativas à herança, interfaces, classes abstractas e Java *generics*

História

- As linguagens de programação têm adicionado meios de organizar "coisas" ao longo dos anos
  - Primeiro apareceu o *array*, a desvantagem que apresentava era a obrigatoriedade das "coisas" serem todas semelhantes (do mesmo tipo)
  - Depois apareceu o registo, um registo permite aos programadores armazenar "coisas" de tipos diferentes

+ Mais recentemente, apareceu o conceito de objecto que permite armazenar dados de diferentes tipos e comportamentos

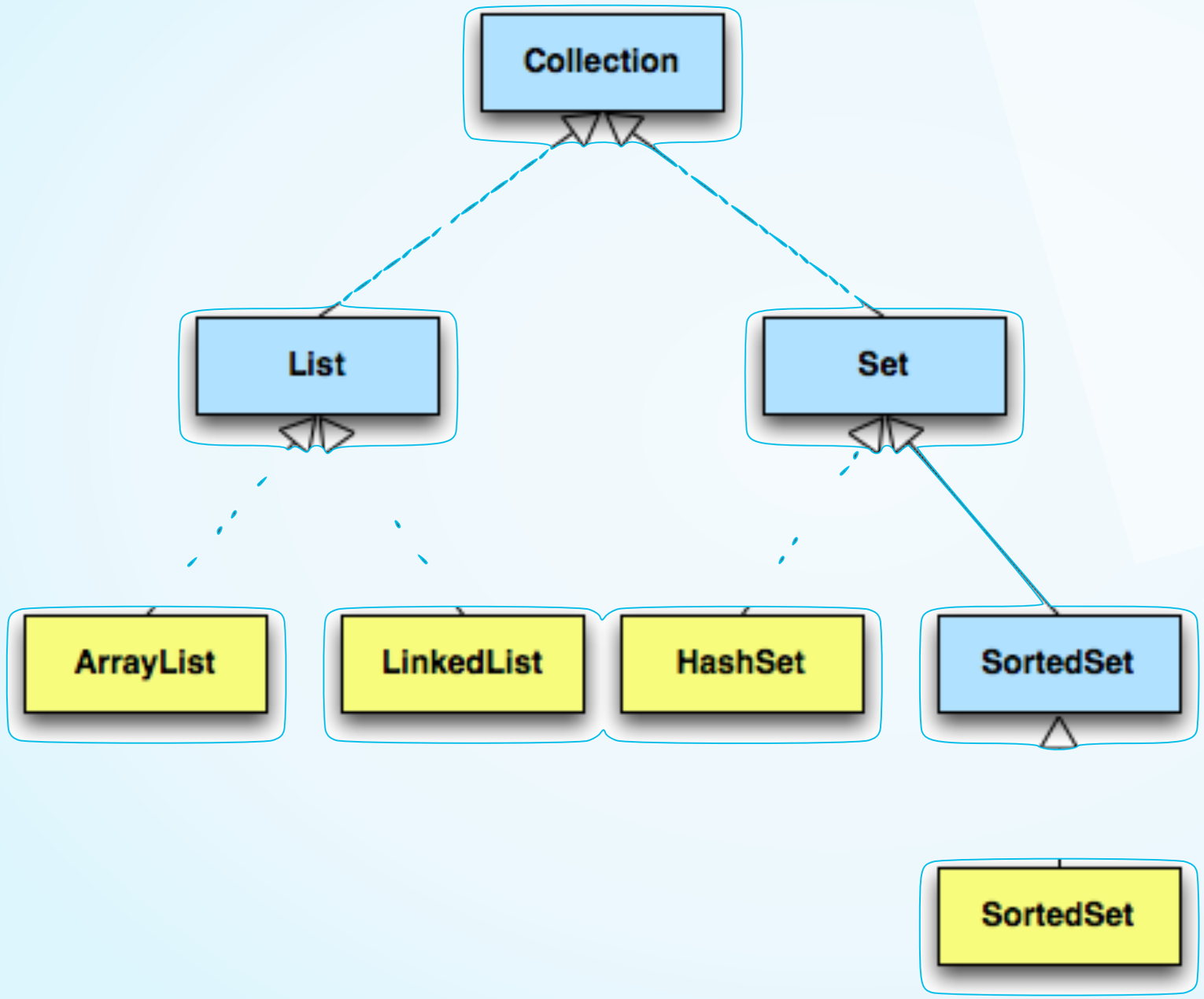
Origens: Java Collection Framework

- No Java 1.2, a linguagem passou a ter uma plataforma para gerir colecções
- As colecções são formas de organizar grupos de “coisas” e manipular esses grupos como entidades únicas

Java Collections Framework (JCF)

- A JCF é uma colecção de interfaces, classes concretas e abstractas que fornecem um conjunto *standard* de tipos de colecções (*containers*)
- A interface **`Collection`** é a raiz de grande parte da hierarquia de heranças de interfaces da JCF

Java Collections Framework



Termos

- **Interface**
  - As interfaces são declaradas com recurso à palavra reservada interface e podem conter apenas assinaturas de métodos e declarações de constantes
- **Classe abstracta**
  - É uma classe que possui pelo menos um método abstracto -> não pode ser instanciada
- **Classe concreta**
  - É uma classe que pode ter instâncias

Generics

- Praticamente todas as aplicações precisam armazenar colecções de objectos
  - Seria impraticável implementar uma nova classe colecção para armazenar cada tipo especifico de objecto
  - É impossível prever quais os tipos de objectos que um dia iremos querer armazenar no grande número de potenciais aplicações que poderão usar a nossa classe colecção
  - Uma forma de generalizar a colecção é torná-la uma colecção do tipo **`Object`** (topo da hierarquia)

```
public class MyCollection {
  ...
  public void add(Object o) {...}
  public boolean remove(Object o) {...}
  public Object get(int position)
}
```

```
public class ShapeApp {
  ...
  MyCollection myc =
     new MyCollection();
  myc.add(new Rectangle(3,4));
  ...
  Rectangle rect =
     (Rectangle)myc.get(0);
}
```

- Todas as referências para cada objecto na classe (que representa a colecção) irão usar o tipo **`Object`**
  - Quando é obtido um objecto a partir da colecção, a aplicação pode-o converter de volta para o tipo original (através do *casting*)
  - Os comportamentos/operações pretendidos pela aplicação cliente que usar esta colecção terão de ser do tipo **`Object`**
    - (Exemplo) Se enviarmos uma Forma (*Shape*) para a colecção não podemos esperar que a colecção seja capaz de fornecer a operação que irá desenhar as formas
    - A operação "Desenhar" não está definida na classe **`Object`**

+ Não existe nenhuma forma da colecção ser capaz de informar se todos os objectos da colecção são do mesmo tipo

+ Uma classe genérica (*Generic Class*) apresenta uma solução para este problema pois pode ser vista como um *template* que pode ser concretizada para conter objectos de um tipo em particular uma vez instanciada

- A classe **`ArrayList`** é uma classe parametrizada
- Tem um parâmetro denominado por **`Base_Type`** que pode ser substituído por qualquer tipo de forma a obter uma classe **`ArrayList`** com o tipo especificado pelo programador
- O Java permite a definição de classes com tipos parametrizados desde a versão 5.0
  - Estas classes que têm tipos parametrizados são denominadas classes parametrizadas ou definições genéricas, ou simplesmente *generics*

- A definição da classe com o tipo parametrizado é armazenada em ficheiro e compilada tal como qualquer outra classe
- Depois da classe parametrizada ser compilada pode ser usada tal como outra classe
- No entanto o tipo parametrizado na classe terá de ser especificado antes de ser usada num programa
- Ao fazê-lo estamos a **instanciar** a classe genérica

```
Demo<String> object = new Demo<String>();
```

+ Definição de uma classe com um tipo parametrizado

```
public class Demo<T> {

   private T data;

   public void setData(T newData) {
      data = newData;
   }

   public T getData() {
      return data;
   }
}
```

+ **T é o tipo parametrizado**

- Uma classe que é definida à custa de um tipo parametrizado é denominada de classe genérica ou classe parametrizada
  - O tipo parametrizado é incluído entre "**<>**" depois do nome da classe no cabeçalho da definição da classe
  - Pode ser usado qualquer identificador desde que não seja uma palavra reservada para o tipo, mas por convenção, o parâmetro começa com uma letra maiúscula
  - O tipo parametrizado pode ser usado como outros tipos usados na definição da classe

+ Definição de uma classe Genérica

```
public class Pair<T> {

    private T first;
    private T second;

    public Pair() {
        first = null;
        second = null;
    }

    public Pair(T firstItem, T secondItem) {
        first = firstItem;
        second = secondItem;
    }

    public T getFirst() {
        return first;
    }

    public void setFirst(T first) {
        this.first = first;
    }
```

O cabeçalho do construtor não inclui o tipo parametrizado

```
public T getSecond() {
    return second;
}

public void setSecond(T second) {
    this.second = second;
}

public String toString() {
    return ("first: " + first.toString() + "\n"
      + "second: " + second.toString());
}

  public boolean equals(Object otherObject) {
      if (otherObject == null) {
          return false;
      } else if (getClass() != otherObject.getClass()) {
          return false;
      } else {
          Pair otherPair = (Pair) otherObject;
          return (first.equals(otherPair.first)
             && second.equals(otherPair.second));
      }
  }
}
```

+ Usar uma classe Genérica


```
public class GenericPairDemo {

    public static void main(String[] args) {

        Pair<String> secretPair =
              new Pair<String>("Happy", "Day");
        Scanner Keyboard = new Scanner(System.in);
        System.out.println("Enter two words:");
        String word1 = Keyboard.next();
        String word2 = Keyboard.next();
        Pair<String> inputPair =
              new Pair<String>(word1, word2);

        if (inputPair.equals(secretPair)) {
            System.out.println("You guesses the secret words");
            System.out.println("in the correct order!");
        } else {
            System.out.println("You guesses incorrectly.");
            System.out.println("You guessed");
            System.out.println(inputPair);
            System.out.println("The secret words are");
            System.out.println(secretPair);
        }
    }
}
```

+ Output

```
Enter two words:
two words
You guesses incorrectly.
You guessed
first: two
second: words
The secret words are
first: Happy
second: Day
```

+ Apesar do nome da classe ter um tipo parametrizado na definição este não é usado na definição do construtor:

```
public Pair<T>()
```

+ Mas pode usar o tipo definido na classe

```
public Pair(T first, T second)
```

+ No entanto quando a classe genérica é instanciada temos que usar os "<>"

```
Pair<String> pair = new Pair<STring>("Happy", "Day");
```

- O tipo usado como tipo parametrizado tem sempre de ser uma referência:
  - Não pode ser um tipo primitivo tais como o **`int`**, **`double`** ou **`char`**
  - No entanto, agora que o Java tem *automatic boxing*, esta não é uma grande restrição
  - Nota: Podem ser usados *arrays*

+ Usar uma classe Genérica `Pair` e *Automatic Boxing*

```
public class GenericPairDemo2 {
    public static void main(String[] args) {
        Pair<Integer> secretPair = new Pair<Integer>(42, 24);
        Scanner Keyboard = new Scanner(System.in);
        System.out.println("Enter two numbers:");
        int n1 = Keyboard.nextInt();
        int n2 = Keyboard.nextInt();
        Pair<Integer> inputPair = new Pair<Integer>(n1, n2);

        if (inputPair.equals(secretPair)) {
            System.out.println("You guesses the secret numbers");
            System.out.println("in the correct two order!");
        } else {
            System.out.println("You guesses incorrectly.");
            System.out.println("You guessed");
            System.out.println(inputPair);
            System.out.println("The secret numbers are");
            System.out.println(secretPair);
        }
    }
}
```

O *Automatic Boxing* permite-nos usar um argumento `int` para um parâmetro `Integer`

+ *Output*

```
Enter two numbers:
42 24
You guesses the secret numbers
in the correct two order!
```

Múltiplos Tipos Parametrizados

- A definição de uma classe genérica pode ter qualquer número de tipos parametrizados
- Os múltiplos tipos parametrizados são listados dentro de "**<>**" separados por vírgulas

```
TwoTypePair<T1, T2>
```

+ Definição de uma classe Genérica

```
public class TwoTypePair <T1, T2>{

    private T1 first;
    private T2 second;

    public TwoTypePair() {
        first = null;
        second = null;
    }

    public TwoTypePair(T1 firstItem, T2 secondItem) {
        first = firstItem;
        second = secondItem;
    }

    public T1 getFirst() {
        return first;
    }

    public void setFirst(T1 first) {
        this.first = first;
    }
```

```
public T2 getSecond() {
      return second;
  }

  public void setSecond(T2 second) {
      this.second = second;
  }

  public String toString() {
      return ("first: " + first.toString() + "\n"
        + "second: " + second.toString());
  }

  public boolean equals(Object otherObject) {
      if (otherObject == null) {
          return false;
      } else if (getClass() != otherObject.getClass()) {
          return false;
      } else {
          TwoTypePair otherPair =
              (TwoTypePair) otherObject;
          return (first.equals(otherPair.first)
            && second.equals(otherPair.second));
      }
  }
}
```

O primeiro `equals` é o `equals` do tipo T1 e o segundo `equals` é o `equals` do tipo T2

Exercício 1: Testar a classe anterior `TwoTypePair` e observar o resultado

Limites para tipos parametrizados

+ Por vezes faz sentido restringir os tipos possíveis que podem ser usados para o tipo parametrizado `T`

+ Por exemplo, para garantir que apenas as classes que implementam a interface `Comparable` são passadas pelo tipo `T` devemos definir a classe da seguinte forma:

```
public class DemoBounds<T extends Comparable>
```

- "**`extends Comparable`**" é usado como limite para o tipo parametrizado `T`

- Qualquer tentativa de passar um tipo para `T` que não implemente a interface **`Comparable`** irá resultar numa mensagem de erro do compilador

- Um limite num tipo pode ser o nome de uma classe (em vez de uma interface)
- Apenas podem ser passadas pelo tipo **T** as subclasses da classe limite definido como tipo parametrizado:

```
public class ExClass<T extends
Class1>
```

- Um expressão de limite pode conter múltiplos interfaces e apenas uma classe
- Se existir mais de um parâmetro a sintaxe deverá ser da seguinte forma:

```
public class Two<T1 extends Class1, T2
   extends Class2 & Comparable>
```

+ Tipo Parametrizado com limite

```
public class Pair<T extends Comparable> {
    private T first;
    private T second;

    public T max() {
        if (first.compareTo(second)>=0)
            return first;
        else
            return second;
    }
}
```

+ **Exercício 2:** Testar a classe

Interfaces Genéricas

- As interfaces podem ter um ou mais tipos parametrizados

- Os detalhes e a notação são os mesmos que temos vindo a aprender para as classes com tipos parametrizados

Métodos Genéricos

+ Quando uma classe genérica é definida o tipo parametrizado pode ser usado nas definições de métodos para a classe genérica

+ Um método genérico pode ainda ser definido com os seus próprio tipos parametrizados sem que sejam os mesmo da classe

+ Um método genérico pode ser um membro de uma classe normal ou genérica que tenha algum tipo parametrizado

+ O tipo parametrizado de um método genérico é local a esse método e não à classe

- O tipo parametrizado tem de ser colocado (entre "**<>**") depois de todos os modificadores e antes do tipo de retorno:

```
public static <T> T genMethod(T[] a)
```

- Quando um deste métodos genéricos é invocado o nome do método é precedido pelo tipo pretendido entre "**<>**"

```
String s = NonG.<String>genMethod(c);
```

Herança com classes genéricas

+ Uma classe genérica pode ser definida como subclasse de uma classe normal ou mesmo de uma outra classe genérica

+ Tal como nas classes normais um objecto do tipo de uma subclasse será também do tipo da superclasse

+ Dadas duas classes: **`A`** e **`B`**, e uma classe genérica: **`G`** não existe nenhum relacionamento entre **`G<A>`** e **`G<B>`**

+ É sempre verdade independente do relacionamento entre as classes **`A`** e **`B`**, por exemplo, se a classe **`B`** é uma subclasse da classe **`A`**

+ Exemplo

```
public class UnorderedPair<T> extends Pair<T> {

    public UnorderedPair() {
        setFirst(null);
        setSecond(null);
    }

    public UnorderedPair(T firstItem, T secondItem) {
        setFirst(firstItem);
        setSecond(secondItem);
    }

    public boolean equals(Object otherObject) {
        if (otherObject == null) {
            return false;
        } else if (getClass() != otherObject.getClass()) {
            return false;
        } else {
            UnorderedPair<T> otherPair =
                    (UnorderedPair<T>) otherObject;

            return (getFirst().equals(otherPair.getFirst())
              && getSecond().equals(otherPair.getSecond()))
              || (getFirst().equals(otherPair.getSecond())
              && getSecond().equals(otherPair.getFirst()));
        }
    }
}
```

+ Exemplo

```
public class UnorderedPairDemo {

    public static void main(String[] args) {
        UnorderedPair<String> p1 =
            new UnorderedPair<String>("peanuts", "beer");
        UnorderedPair<String> p2 =
            new UnorderedPair<String>("beer", "peanuts");
        if (p1.equals(p2)) {
            System.out.println(p1.getFirst() + " and "
                + p1.getSecond() + " is the same as");
            System.out.println(p2.getFirst() + " and "
                + p2.getSecond());
        }
    }
}
```

```
peanuts and beer is the same as
beer and peanuts
```

Wildcards

+ Queremos um método que imprima todos os elementos de uma colecção

```
void printCollection(Collection c){
   Iterator i = c.iterator();
   while (i.hasNext()) {
      System.out.println(i.next());
   }
}
```

Wildcards

+ 1ª Tentativa com o uso de *generics*

```
void printCollection(Collection<Object> c) {
   for (Object e: c){
      System.out.println(e);
   }
}
```

+ Se repararem bem acaba por ser pior que a primeira solução

```
printCollection(stones);
```

Não compila!

- Que tipo abrange todos os tipos de colecções?

```
Collection<?>
```

- "**Colecção de tipo desconhecido**" é uma colecção em que o tipo dos elementos pode ser qualquer um - tipo *wildcard*

```
void printCollection(Collection<?> c) {
   for (Object e: c){
      System.out.println(e);
   }
}
```

```
printCollection(stones);
```

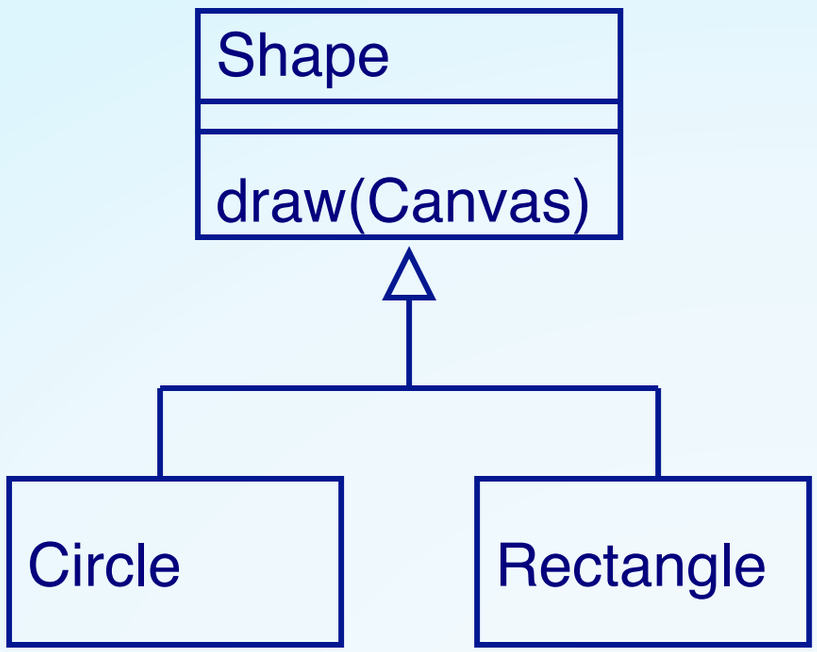
+ Atenção!


```
String myString;
Object myObject;
List<?> c = new ArrayList<String>();

// c.add("hello world");           // compile error
// c.add(new Object());            // compile error
((List<String>) c).add("hello world");
((List<Object>) c).add(new Object()); // no compile error!

// String myString = c.get(0);     // compile error
myString = (String) c.get(0);
myObject = c.get(0);
myString = (String) c.get(1);      // run-time error!
```

Wildcards com limite

- Considere uma simples aplicação de desenho que pretende desenhar formas (*Shapes*): *Circles*, *rectangles*, ...

- Como implementar o método para desenhar todas as formas de uma colecção: **`drawAll`**



Canvas

draw(Shape)
drawAll(List**<Shape>**)

Limitado a List<Shape>

Qual a solução para que **`drawAll`** possa desenhar todas as formas?

+ Um método que aceita uma **`List`** de qualquer tipo de **`Shape`**:

```
public void drawAll(List<? extends Shape>) {…}
```

+ **`Shape`** é o limite superior do *wildcard*

+ Outros exemplos:

```
public void pushAll(Collection<? extends E> collection) {
    for (E element : collection) {
        this.push(element);
    }
}

public List<E> sort(Comparator<? super E> comp) {
    List<E> list = this.asList();
    Collections.sort(list, comp);
    return list;
}
```

+ **`? extends E`** qualquer coisa que seja subclasse de **`E`** (incluindo **`E`**)

+ **`? super E`** qualquer coisa que seja superclasse de **`E`** (incluindo **`E`**)

Usar o java.util.ArrayList

+ O **`ArrayList`** é um *array* genérico que pode ser redimensionado automaticamente

```
List<String> al = new ArrayList<String>(5);
for(int i = 0; i < 5; i++) {
  al.add(new String(i))
}
//automatic resizing when one more item is added
al.add(new String(5));
```

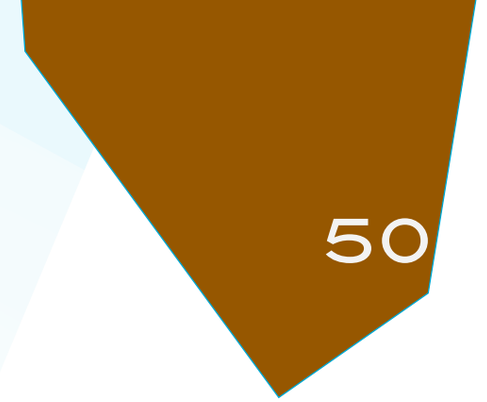
+ Quando se pretende adicionar o sexto objecto o **`ArrayList`** verifica que está cheio

+ São realizadas as seguintes acções:

- É alocado um novo *array* com capacidade de 10 (o dobro do anterior com capacidade de 5)
- Todos os itens do *array* antigo são copiados para o novo
- O novo item a ser adicionado é acrescentado ao novo *array*
- A referência interna para a colecção é alterado do antigo *array* para o novo *array* - *garbage collector* elimina o anterior

- Este processo pode voltar a repetir-se caso o **`ArrayList`** volte a encher a capacidade (preencher todas as 10 localizações), aquando da introdução de um novo elemento
  - Novo *array* de 20 posições

- Ver a classe **`java.util.ArrayList.java`** (código fonte)



- Como pudemos verificar o **`ArrayList`** usar um *array* como estrutura de dados!
- De que outras formas podemos armazenar os nossos dados?
- Que outras estruturas de dados existe?
- Que diferenças vantagens/desvantagens têm entre elas?