
FICHA PRÁTICA 4

LABORATÓRIO DE COLECÇÕES I

ARRAYLIST<E> E HASHSET<E>

SÍNTESE TEÓRICA

Em JAVA, tal como em algumas outras linguagens de programação por objectos, certas estruturas de objectos (colecções) são *parametrizadas*, ou seja, aceitam um tipo parâmetro a ser posteriormente substituído por um tipo concreto (p.e., uma classe).

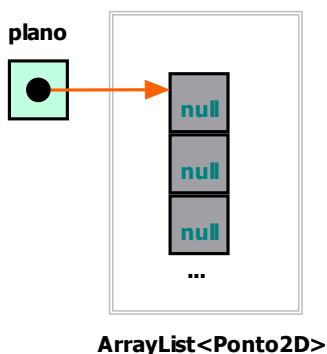
Uma colecção de JAVA de extraordinária utilidade na agregação de objectos e sua estruturação sob a forma de uma lista, é **ArrayList<E>**. Tendo por suporte um *array* de características dinâmicas, ou seja, capaz de aumentar ou diminuir de dimensão ao longo da execução de um programa, um **ArrayList<E>**, implementa listas de objectos, em que E representa a classe/tipo parâmetro de cada um dos objectos nele contidos. Um *arraylist* é pois, ao contrário de um *array*, um objecto e de dimensão dinâmica, possuindo um grande conjunto de métodos a si associados para realizar as operações mais usuais sobre listas, não podendo porém conter valores de tipos primitivos (apenas objectos).



A declaração de um *arraylist* concreto, consiste em definir-se qual o tipo concreto para o seu tipo parâmetro, numa declaração normal mas paramétrica, onde, em geral, usando um **construtor**, é criado imediatamente um *arraylist* inicialmente vazio (atenção ao **construtor** que também é parametrizado), com dimensão inicial ou não.

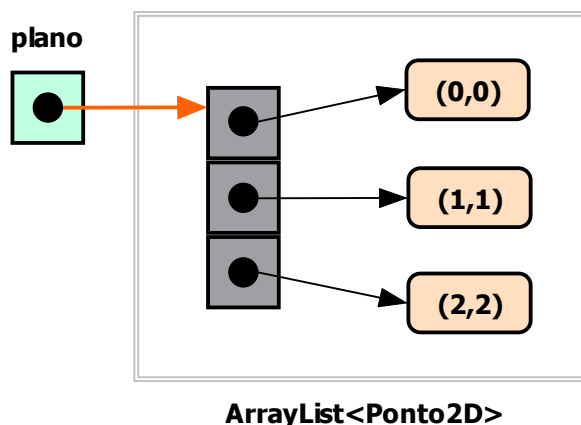
```
ArrayList<String> nomes = new ArrayList<String>(100);  
ArrayList<Ponto2D> plano = new ArrayList<Ponto2D>();
```

Em ambos os casos o estado inicial do *arraylist* deverá ser visto como sendo uma lista de apontadores a **null**, já que nenhum elemento foi ainda criado e inserido, cf.



Posteriormente, e à medida que cada um dos **Ponto2D** forem inseridos usando os diversos métodos que apresentaremos em seguida, cada posição do *arraylist* (iniciadas no índice 0),

referenciará uma instância de `Ponto2D` ou manter-se-á a `null`, tal como se ilustra na figura seguinte.



A API da classe `ArrayList<E>` é apresentada em seguida, encontrando-se as várias operações disponíveis agrupadas por funcionalidades para melhor compreensão. Todos os parâmetros representados como sendo do tipo `Collection` devem ser assumidos como podendo ser uma qualquer colecção de JAVA, ainda que neste momento estejamos limitados a trabalhar apenas com `ArrayList<E>`.

SINTAXE ESSENCIAL

Categoria de Métodos	API de <code>ArrayList<E></code>
Construtores	<code>new ArrayList<E>()</code> <code>new ArrayList<E>(int dim)</code> <code>new ArrayList<E>(Collection)</code>
Inserção de elementos	<code>add(E o); add(int index, E o);</code> <code>addAll(Collection); addAll(int i, Collection);</code>
Remoção de elementos	<code>remove(Object o); remove(int index);</code> <code>removeAll(Collection); retainAll(Collection)</code>
Consulta e comparação de conteúdos	<code>E get(int index); int indexOf(Object o);</code> <code>int lastIndexOf(Object o);</code> <code>boolean contains(Object o); boolean isEmpty();</code> <code>boolean containsAll(Collection); int size();</code>
Criação de Iteradores	<code>Iterator<E> iterator();</code> <code>ListIterator<E> listIterator();</code> <code>ListIterator<E> listIterator(int index);</code>
Modificação	<code>set(int index, E elem); clear();</code>
Subgrupo	<code>List<E> sublist(int de, int ate);</code>
Conversão	<code>Object[] toArray();</code>
Outros	<code>boolean equals(Object o); boolean isEmpty();</code>

Quadro – API de `ArrayList<E>`

NOTA:

Os parâmetros designados como sendo de tipo `Collection` serão explicados mais tarde. De momento apenas será necessário que se compreenda que são colecções de JAVA. De momento apenas podemos usar como parâmetros dos métodos colecções que sejam `ArrayList<T>` e em que `T` seja um tipo compatível com o tipo `E` (por exemplo, sendo mesmo `T = E`). Tal significa que se estivermos a trabalhar, por exemplo, com um `ArrayList<Ponto2D>` apenas parâmetros do tipo `ArrayList<Ponto2D>` devem ser considerados em métodos como `addAll()` e outros que aceitem `Collections` como parâmetros. Mais tarde veremos porquê.

1.- EXEMPLOS SIMPLES

Consideremos um exemplo simples cujo objectivo é a apresentação da semântica dos principais métodos da API de `ArrayList<E>`, através da sua utilização.

Vamos criar duas listas de nomes, uma contendo os nomes de amigos e outra contendo os nomes de pessoas que conhecemos e são músicos. Serão, naturalmente dois `ArrayList<String>`. Vamos em seguida realizar a sua declaração completa, limitando o número de músicos a uns 50.

DECLARAÇÕES

```
ArrayList<String> amigos = new ArrayList<String>(); // vazio
ArrayList<String> musicos = new ArrayList<String>(50); // capacidade = 50
```

INSERÇÕES NO FIM USANDO `add(E elem)`

```
amigos.add("Jorge"); amigos.add("Ivo"); amigos.add("Rino");
// ERRO DE COMPILAÇÃO => amigos.add(new Ponto2D(0.0, 2.0));
musicos.add("Rino"); musicos.add("Zeta");
```

NÚMERO ACTUAL DE ELEMENTOS DE UMA LISTA

```
int numAmigos = amigos.size();
```

CONSULTA DO ELEMENTO NO ÍNDICE `i <= amigos.size() - 1`

```
String amigo = amigos.get(10);
String nome = amigos.get(i);
```

ITERAÇÃO SOBRE TODOS OS ELEMENTOS DE UM `ArrayList<String>`

```
// Imprimir os nomes de todos os amigos - Solução A: Ciclo for
for(int i = 0; i <= amigos.size()-1; i++)
    out.printf("Nome: %s\n", amigos.get(i));
```

```
// Imprimir os nomes de todos os amigos - Solução B: Criando um Iterator
// Depois de criado, o Iterator<String> é percorrido até it.hasNext() ser falso,
// sendo cada elemento devolvido usando it.next()
for(Iterator<String> it = amigos.iterator(); it.hasNext();)
    out.printf("Nome: %s\n", it.next());
```

```
// Imprimir os nomes de todos os amigos - Solução C: Usando for(each)
for(String nome : amigos)
    out.printf("Nome: %s\n", nome);
```

PROCURA DE UM ELEMENTO => USAR `Iterator<E>` E `while`

```
String chave = Input.lerString();
encontrado = false; int index = 0;
Iterator<String> it = amigos.iterator(); // Iterator<String> criado
while(it.hasNext() && !encontrado) { // iteração
    encontrado = it.next().equals(chave); index++;
}
if (encontrado)
    out.printf("O nome %s está na posição %2d\n", chave, index-1);
else
    out.println("O nome " + chave + " não existe !\n");
```

PROCURA DE UM ELEMENTO (VERSÃO SIMPLES)

```
String chave = Input.lerString();
int index = amigos.indexOf(chave);
if (index == -1)
    out.println("Não existe tal nome !");
else
    out.println("Está no índice: " + index);
```

ARRAYLIST ESPARSO (NÃO CONTÍGUO): UTILIZAÇÃO CORRECTA

A) INICIALIZAR A null AS POSIÇÕES QUE SE PRETENDEM ACEDER;

B) USAR set(int index, E elem) SENDO index <= size()-1;

```
ArrayList<String> cantores = new ArrayList<String>(DIM); // capac = DIM
for(int i = 0; i <= DIM-1; i++) cantores.add(null);
// inserção de nomes em qualquer índice até DIM-1 !!
cantores.set(10, "Rui"); cantores.set(2, "Ana");
```

ELEMENTOS COMUNS A DOIS ARRAYS

```
// Determinar quais os amigos que também são músicos
//
// 1) Cria um arraylist e copia o arraylist amigos na criação
ArrayList<String> temp = new ArrayList<String>(amigos);
// ou, em alternativa equivalente usando temp1
ArrayList<String> temp1 = new ArrayList<String>();
for(String am : amigos) temp1.add(am);
// 2) Remove da lista temp todos os que não são musicos, ou seja,
// retém os que também estão na lista musicos.
temp.retainAll(musicos);
// imprime os nomes
out.println("\n--- AMIGOS MÚSICOS ---\n");
for(String name : temp) out.printf(" %s\n", name);
// o mesmo para a lista de amigas
temp = new ArrayList<String>(amigas); temp.retainAll(musicos);
out.println("\n--- AMIGAS MÚSICAS ---\n");
for(String n : temp) out.printf(" %s\n", n);

// Remover da lista de amigos e da lista de amigas os que são músicos;
//
// As listas amigos e amigas são modificadas caso alguns sejam músicos !!
// Outra hipótese seria usar, tal como acima, um arraylist temporário.
amigos.removeAll(musicos); amigos.removeAll(musicos);
// imprime os nomes de amigos e amigas restantes
out.println("\n--- AMIGOS NÃO MÚSICOS ---\n");
for(String n : amigos) out.printf(" %s\n", n);
out.println("\n--- AMIGAS NÃO MÚSICAS ---\n");
for(String n : amigas) out.printf(" %s\n", n);

// Criar um arraylist que seja a reunião das listas de amigos e amigas;
//
// Esta operação é semelhante a uma reunião matemática, mas não faz
// qualquer filtragem de elementos em duplicado (também não devem
// existir neste caso !!).
ArrayList<String> todos = new ArrayList<String>(amigos);
todos.addAll(amigas);
out.println("--- Todos : ---\n");
for(String nom : todos) out.printf("Nome: %s\n", nom);
```

SÍNTESE HASHSET<E> E TREESSET<E>

As coleções do tipo `Set<E>` são implementações de *conjuntos de objectos*, pelo que os seus elementos não são indexados ou acessíveis por índice, nem podem ocorrer em duplicado (cf. noção matemática de conjunto). Vamos de momento centrar a nossa atenção em `HashSet<E>` e em `TreeSet<E>`.

A coleção `HashSet<E>` é uma muito eficiente implementação de conjuntos. Conjuntos são coleções de objetos que não possuem ordem (não existe a noção de anterior e seguinte e não existe indexação) e que não admitem duplicados. Os métodos `add()` e `addAll()` de `HashSet<E>` garantem que nunca são inseridos elementos em duplicado.

A coleção `HashSet<E>` implementa conjuntos usando uma *tabela de hashing* e *algoritmos de hashing*, enquanto que `TreeSet<E>` é uma implementação de conjuntos baseada numa *árvore binária ordenada*, sendo em geral necessário fornecer o *algoritmo de comparação* de objetos (cf. maior, igual ou menor) aquando da criação do próprio `TreeSet<E>`. Porém, se o tipo parâmetro `E` do `TreeSet<E>` for do tipo `String` ou de uma das classes *wrapper* (cf. `Integer`, `Double`, `Float`, etc.) então esse algoritmo está previamente definido em JAVA (ordem sintática crescente para *strings* e a ordem natural para os tipos `int`, `double`, etc.) pelo que poderemos utilizar `TreeSet<E>` sem ter que definir qualquer algoritmo de comparação. Nestes casos, usar `TreeSet<E>` poderá ter vantagens relativamente a usar `HashSet<E>` caso pretendamos ter os elementos ordenados. Em geral, tal é útil em `TreeSet<String>`, como em:

```
// conjunto de nomes por ordem alfabética
TreeSet<String> nomes = new TreeSet<String>();
```

Qualquer coleção é, como vimos já, uma agregação de apontadores para objetos do tipo do parâmetro da coleção, que inicialmente tomam o valor `null`, já que nenhum elemento foi ainda inserido, cf.

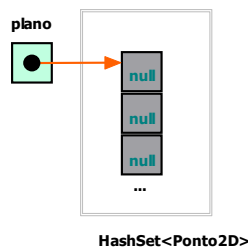


Fig. 4.4 – Estado inicial de `HashSet<Ponto2D>`

Posteriormente, e à medida que cada um dos `Ponto2D` for inserido, usando os diversos métodos que apresentaremos em seguida, cada posição referenciará uma instância de `Ponto2D`. Porém, nos conjuntos não há indexação, ou seja, não há acesso aos elementos por índice. Tal significa que a API de `Set<E>` é um subconjunto da API de `List<E>` sem as operações por índice.

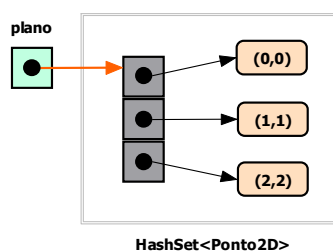


Fig. 4.5 – `HashSet<Ponto2D>` com 3 elementos `Ponto2D`

Assim a partir deste momento, sempre que num projeto pretendermos distinguir coleções que admitem duplicados de coleções que não admitem duplicados, ou entre coleções com ordem ou sem ordem, deveremos ter a preocupação de escolher entre `ArrayList<E>` e `HashSet<E>` ou `TreeSet<E>`.

API ESSENCIAL DE `HashSet<E>` E `TreeSet<E>`

Categoria de Métodos	API de <code>HashSet<E></code> e <code>TreeSet<E></code>
Construtores	<code>new HashSet<E>()</code> <code>new HashSet<E>(int dim)</code> <code>new HashSet<E>(Collection)</code>
Inserção de elementos	<code>add(E o);</code> <code>addAll(Collection);</code>
Remoção de elementos	<code>remove(Object o);</code> <code>removeAll(Collection);</code> <code>retainAll(Collection)</code>
Consulta e comparação de conteúdos	<code>boolean contains(Object o);</code> <code>boolean isEmpty();</code> <code>boolean containsAll(Collection);</code> <code>int size();</code>
Criação de Iteradores	<code>Iterator<E> iterator();</code>
Modificação	<code>clear();</code>
Conversão	<code>Object[] toArray();</code>
Outros	<code>boolean equals(Object o);</code>

Quadro 4.1 – API de `HashSet<E>` e `TreeSet<E>`

As API de `HashSet<E>` e de `TreeSet<E>` é um subconjunto da API de `ArrayList<E>`, tendo desaparecido os métodos que trabalham com índices e os iteradores específicos para listas (cf. `ListIterator<E>`).

PROJETO EXEMPLO

INTRODUÇÃO

Os exercícios e respetivas resoluções que vão ser apresentados a partir deste capítulo assumem já as características não do simples desenvolvimento de uma classe mas do desenvolvimento de um pequeno projeto de *software* OO, possuindo as várias fases típicas de desenvolvimento desde a análise de requisitos até à efetiva implementação de uma aplicação segura, robusta, modificável e extensível.

A maioria destas propriedades resultam da obediência a princípios de engenharia de *software* que temos vindo gradualmente a introduzir, com especial atenção para o princípio do encapsulamento que é absolutamente crucial para a abstração e robustez das nossas aplicações.

A adoção de certas práticas e técnicas de programação garante-nos a satisfação de tais princípios como veremos. Assim sendo, e tendo em atenção a dimensão em termos de linhas de código que a maioria dos projetos apresentados a seguir possuem, não será razoável comentar as mesmas práticas em cada um dos projetos, dado serem no fundo as tais mesmas práticas que garantem a satisfação de tais princípios. Assim, visando combater tal redundância que seria até monótona, apresenta-se nesta seção o desenvolvimento passo a passo de um projeto exemplo desde a análise de requisitos até à sua eventual prototipagem em BlueJ (o BlueJ é neste contexto apenas um acessório, nunca o fundamental da questão).

Toda a análise e todas as decisões de implementação comentadas relativamente a este projeto serão igualmente válidas para todos os outros projetos apresentados.

REQUISITOS

Pretende-se desenvolver uma aplicação que permita realizar um conjunto de operações básicas sobre as informações registadas acerca dos alunos que frequentam uma dada **Turma**, ou seja, sobre todos os alunos inscritos em tal Turma.

Sobre cada aluno deve possuir-se a seguinte informação: número, nome, curso, ano do curso, média atual e lista com os nomes das disciplinas a que está inscrito.

Designando esta informação referente a cada aluno por **FichaAluno**, pretende-se que, para além dos construtores, das usuais operações de consulta e modificação, e ainda `equals()`, `toString()` e `clone()`, sejam disponibilizadas também as operações seguintes:

- *Operação de modificação do nome do aluno;*
- *Operação de alteração da média atual do aluno;*
- *Operação que devolve uma lista com os nomes das disciplinas a que o aluno se encontra inscrito;*
- *Operação que determina o número de disciplinas a que um aluno está inscrito;*
- *Operação que determina se o aluno está inscrito a uma disciplina cujo código é dado como parâmetro;*
- *Operação que inscreve o aluno a uma nova disciplina;*

Designando por **Turma** uma lista de fichas de alunos inscritos, à qual se associa também um código de disciplina e o seu nome, pretende-se agora desenvolver o seguinte conjunto de operações sobre tal coleção de fichas de aluno:

- *Inserir uma nova ficha de aluno na turma;*

- *Verificar se um aluno cujo número é dado está inscrito na turma;*
- *Operação que devolve a ficha completa de um aluno cujo número é dado;*
- *Remover a ficha do aluno cujo número é dado como parâmetro;*
- *Determinar o número atual de alunos inscritos;*
- *Operação que devolve a lista com os números dos alunos inscritos;*
- *Operação que devolve um conjunto com os números dos alunos com média superior à média dada como parâmetro;*
- *Operação que determina a maior média da turma;*

ANÁLISE DOS REQUISITOS

Da análise do problema torna-se evidente a necessidade de começarmos por definir a classe que irá permitir criar *fichas de aluno*. Designaremos tal classe por **FichaAluno**.

FichaAluno

Tal classe, muito bem caracterizada nos requisitos do problema não oferece grandes dúvidas quanto à sua estruturação. Assim, a classe **FichaAluno**, possuirá a seguinte estrutura:

```
// variáveis de instância
private String numero;    // numero do Aluno
private String nome;      // nome do Aluno
private String curso;     // código do Curso
private String anoCurso;
private double media;     // média actual
private ArrayList<String> discp; // nomes das disciplinas
```

Analiseemos agora os construtores que fará sentido sejam desenvolvidos. Sem dúvida que deveremos redefinir o construtor predefinido de JAVA `FichaAluno()`. Ainda que não sendo uma grande semântica, uma possibilidade seria:

```
public FichAluno() {
    numero = ""; nome = "";
    curso = ""; anoCurso = "";
    media = 0.0;
    discp = new ArrayList<String>();
}
```

O segundo construtor, em geral também óbvio e muito comum, é aquele que recebe todas as partes que irão constituir a instância e as atribui *corretamente* às respetivas variáveis de instância. Atribuir corretamente significa, sobretudo, garantir que não há entre *parâmetros* e *variáveis de instância* qualquer partilha de objectos, ou seja, que por erro de atribuição, nenhuma variável de instância fica a referenciar um objecto qualquer que já está a ser referenciado por uma outra variável, neste caso a que foi usada como parâmetro na invocação do construtor. Tais precauções não fazem sentido para tipos simples – que são valores –, devendo aplicar-se apenas a tipos referenciados, exceptuando as instâncias da classe `String` e das classes *wrapper* (cf. `Integer`, `Float`, etc.) que em JAVA são tratadas como valores, ou seja, são copiadas.

Assim, o segundo construtor terá o seguinte código:

```
public FichAluno(String num, String nom,
    double media, String curso,String anoC,
    ArrayList<String> nm_discp) {
    numero = num; nome = nom;
    this.media = media;
```



```

        this.curso = curso; anoCurso = anoC;
        // copiar o ArrayList parâmetro para discp
        discp = new ArrayList<String>();
        for(String nmD : nm_discp) { discp.add(nmD); }
    }

```

Note-se que foi realizada uma cópia explícita do *arraylist* parâmetro elemento a elemento, em vez de se fazer *clone* do mesmo, já que tal implicaria realizar *casting*, dado que o método `clone()` de `ArrayList<E>` (e de qualquer outra coleção) devolve um `Object` e não faz cópia segura. Assim, e por estas razões, nunca usaremos nos nossos projetos os métodos `clone()` de nenhuma coleção.

Analisemos agora o conjunto de métodos de instância que pretendemos implementar para instâncias de `FichaAluno`. Em primeiro lugar os usuais interrogadores ou modificadores básicos, que são:

```

    public String getNumero() { return numero; }
    public String getNome() { return nome; }
    public double getMedia() { return media; }
    public String getAnoCurso() { return anoCurso; }
    public String getCurso() { return curso; }
    public void setNome(String nom) { nome = nom; }
    public void setMedia(double nvMed) { media = nvMed; }

```

A operação que devolve um `ArrayList<String>` das disciplinas a que o aluno está inscrito tem a mesma particularidade da partilha de referências anteriormente referida. Programar apenas um `return discp;` em tal método, consistiria em devolver uma referência (apontador) para o *arraylist* de nomes das disciplinas do objecto ficha de aluno.

Assim, temos que codificar tal método de modo a que não haja partilha de endereços, o que faremos através da criação de uma cópia que será dada como resultado.

```

    /** Devolve uma cópia dos códigos das disciplinas a que
        o aluno está inscrito */
    public ArrayList<String> getDiscp(){
        ArrayList<String> dsp = new ArrayList<String>();
        for(String nome : discp) { dsp.add(nome); }
        return dsp;
    }

```

Esta operação poderia perfeitamente ter sido codificada usando `TreeSet<String>`, em cujo caso os nomes das disciplinas seriam guardados por ordem alfabética crescente. Note que as modificações no código são mínimas.

```

    /** Devolve o conjunto ordenado das disciplinas
        a que o aluno está inscrito */
    public TreeSet<String> getDiscp(){
        TreeSet<String> dsp = new TreeSet<String>();
        for(String nome : discp) { dsp.add(nome); }
        return dsp;
    }

```

A operação que verifica se o aluno a que ficha diz respeito está inscrito numa dada disciplina, consiste em utilizar-se o método **`contains()`** da API de `ArrayList<E>` sobre a variável `discp`, devolvendo tal método um valor lógico como resultado.

```

    /** Aluno inscrito a esta disciplina? */
    public boolean inscritoA(String nomeDiscp) {
        return discp.contains(nomeDiscp);
    }

```

Devolver o número de disciplinas a que o aluno está inscrito é calcular o tamanho actual do *arraylist* de códigos de disciplinas a que o aluno está inscrito segundo a sua ficha.

```

    /** Total de disciplinas a que o aluno está inscrito */    public    int
    numInscricoes() { return discp.size(); }

```

O método void `inscreveA(String novaDiscp)` será o método a desenvolver para se inscrever um aluno a mais uma *nova* disciplina. Assim, é importante que, antes que tal método seja invocado, se possa ter a certeza de que o aluno não está inscrito a tal disciplina. O método ou programa invocador do método `inscreveA(String novaD)`, deverá, antes de realizar a invocação, usar o método `inscritoA(String novaD)` para verificar se tal disciplina já pertence ou não ao conjunto das disciplinas a que o aluno está inscrito. Se já existir, então não será necessário realizar a invocação do método pois tal nome *está errado*. Trata-se de uma verificação prévia que deteta o erro e evita a invocação. O método, que deste modo será sempre invocado em situações corretas, limita-se a usar o método `add(E elem)` de `ArrayList<E>`, inserindo o nome no fim do *arraylist*.

```

    /**Junta um novo nome de disciplina às inscrições */
    public void inscreveA(String novaDiscp) {
        discp.add(novaDiscp);
    }

```

Operação que permite obter uma representação completa sob a forma de texto de um qualquer objecto do tipo `FichaAluno`, para que possa ser visualizado em ecrã ou até ser escrito num ficheiro de texto.

```

    public String toString() {
        StringBuilder s = new StringBuilder();
        s.append(" --- FICHA DO ALUNO Nº: ");
        s.append(numero); s.append("\n");
        s.append("NOME : "); s.append(nome); s.append("\n");
        s.append("MEDIA : ");
        s.append(media); s.append("\n");
        s.append("CURSO : ");
        s.append(curso); s.append("\n");
        s.append("ANO : ");
        s.append(anoCurso); s.append("\n");
        s.append("----- INSCRITO A -----\n");
        for(String cod : discp) {
            s.append(cod) ; s.append("\n");
        }
        return s.toString();
    }

```

A operação que realiza a criação de uma cópia de uma `FichaAluno` será realizada, tal como já fizemos em projetos anteriores e continuaremos a fazer em projetos futuros, criando uma nova instância da classe `FichaAluno` usando o *construtor de cópia*, codificação que para nós será sempre padrão.

```

    /** Clonagem - criação de instância que é uma cópia */
    public FichAluno clone() { return new FichAluno(this); }

```

Vamos em seguida analisar a estrutura e o comportamento da classe que irá agregar estas fichas, classe que designaremos por **TurmaList**.

TurmaList

Nesta primeira versão deste projeto, e de forma propositada, vamos considerar que a classe **TurmaList** estrutura as fichas de aluno sob a forma de uma *lista de fichas* sem ordem especial a não ser a sua ordem de entrada, tendo-se assim decidido utilizar a classe `ArrayList<E>`, que implementa uma estrutura sequencial indexada de 0 até n – uma lista -, em *arrays* dinâmicos em tamanho e virtualmente infinitos de elementos de tipo E, possuindo um enorme conjunto de métodos para operar sobre tal estrutura.

Assim, para além do seu código e do seu nome, a classe turma possuirá um *arraylist* de *FichaAluno*, cf. as variáveis de instância a seguir apresentadas.

```
// variáveis de instância
private String codigo; // código interno da disciplina
private String nomeDiscp;
private ArrayList<FichaAluno> turma;
```

Vamos criar os construtores usuais, um que redefina o construtor por omissão, outro que corresponde ao construtor completo ou das partes e o construtor de cópia.

```
public TurmaList() {
    codigo = ""; nome = "";
    turma = new ArrayList<FichaAluno>();
}

public TurmaList(TurmaList t) {
    codigo = t.getCodigo();
    nome = t.getNome();
    turma = new ArrayList<FichaAluno>();
    for(FichaAluno fa : t.daFichas())
        turma.add(fa.clone());
}

public TurmaList(String cod, String nm) {
    codigo = cod; nome = nm;
    turma = new ArrayList<FichaAluno>();
}

/** Construtor completo
    Insere um ArrayList de Fichas válidas em turma
 */
public TurmaList(String cod, String nm,
    ArrayList<FichaAluno> colFichas) {
    codigo = cod; nome = nm;
    turma = new ArrayList<FichaAluno>();
    turma.addAll(colFichas); // código incorreto !
}
```

O construtor completo assume que o *arraylist* de fichas de aluno que lhe é passado como parâmetro foi anteriormente validado, ou seja, o construtor não assume qualquer responsabilidade por introduzir fichas que possam conter incoerências, isto é, não cumpram as propriedades (invariantes, requisitos, regras, etc.).

Este construtor, aparentemente correto, não funciona como se pretendia dado que o método `addAll()` de algumas coleções de JAVA não faz, nem poderia fazer, uma cópia do objecto a adicionar à coleção recetora, pois apenas lhe passa o endereço do objecto. Ou seja, este objecto passa a estar partilhado pelas duas coleções: a recetora e a colecção parâmetro. Tal, como sabemos, não pode acontecer. O código está errado.

O código seguinte, que se baseia na utilização do construtor de `ArrayList<E>` que aceita uma coleção como parâmetro e inicializa o *arraylist* que está a criar com os elementos de tal coleção, também está apenas sintaticamente correto. De facto, o resultado desta sintaxe é exatamente igual ao uso de `addAll()` e portanto, os endereços das *FichaAluno* serão mais uma vez partilhados

```
turma = new ArrayList<FichaAluno>(colFichas);
```

Para que o código seja de fato correto, é necessário que cada *FichaAluno* oriunda do parâmetro de entrada seja por nós explicitamente copiada para a coleção recetora, usando o método `clone()`

definido na classe `FichaAluno`. Teremos finalmente uma *deep copy* e endereços não compartilhados, ou seja, objetos distintos (ainda que com iguais valores).

```
/** Construtor completo
    Insere um ArrayList de Fichas válidas em turma
*/
public TurmaList(String cod, String nm,
                  ArrayList<FichaAluno> colFichas) {
    codigo = cod; nome = nm;
    turma = new ArrayList<FichaAluno>();
    for(FichaAluno ficha : colFichas)
        turma.add(ficha.clone()); // cópia antes de inserir
}
```

Vamos agora analisar os métodos de instância de **TurmaList** começando pelos usuais métodos de consulta e de modificação.

```
// Métodos de Instância
/** Devolve o nome da Disciplina/Turma */
public String getNome() { return nomeDiscp; }

/** Muda o nome da Disciplina/Turma */
public void mudaNomeDiscip(String novoNome) {
    nomeDiscp = novoNome;
}

/** Devolve o código da Disciplina/Turma */
public String getCodigo() { return codigo; }

/** Determina o número de alunos da Turma */
public int numAlunos() { return turma.size(); }

/** Devolve uma cópia do ArrayList de FichaAluno */
public ArrayList<FichaAluno> daFichas() {
    ArrayList<FichaAluno> fichas =
        new ArrayList<FichaAluno>();
    for(FichaAluno fa : turma) fichas.add(fa.clone());
    return fichas;
}
```

Criar uma lista com os números dos alunos da turma, sendo os números dos alunos da turma do tipo `String`, pode ser resolvido usando um `ArrayList<String>` que se inicializa a vazio e para o qual se copiam, um a um, todos os números dos alunos obtidos de cada uma das fichas encontradas no *arraylist* `turma`. Para percorrer o *arraylist* vamos mais uma vez usar o iterador sobre coleções `for(each)` (ler “*para cada ...obtida de ...*”).

```
/** Devolve uma lista com os números dos alunos */
public ArrayList<String> codigos() {
    ArrayList<String> cods = new ArrayList<String>();
    for(FichaAluno ficha : turma)
        cods.add(ficha.getNumero());
    return cods;
}
```

Estando tal solução perfeitamente correta algoritmicamente, tanto mais que não existem dois alunos com o mesmo número e, portanto, tal lista, ainda que sendo uma lista, é de fato um conjunto pois não contém duplicados, a verdade é que, em informática, existe uma certa tendência para nos requisitos dos projetos nos pedirem *listas de coisas* que, de fato, não são listas mas sim conjuntos. No exemplo anterior, e embora nos tenha sido pedida uma *lista dos números dos alunos* e tenha sido isso que tenha sido documentado e programado, mais correto seria termos programado e documentado de forma

explícita que o resultado do método é um *conjunto de números de alunos* pois é uma coleção onde não devem existir duplicados.

Procurando realizar tal correção, que é bastante simples, teríamos apenas que procurar saber quais as classes de JAVA que implementam conjuntos matemáticos. Teríamos duas possíveis implementações genéricas: `TreeSet<E>` e `HashSet<E>`. A classe `TreeSet<E>` não só implementa conjuntos mas também permite definir uma ordenação dos seus elementos e vamos usá-la neste caso.

```
/** Cria um conjunto com os números dos alunos */
public TreeSet<String> codigos() {
    TreeSet<String> cods = new TreeSet<String>();
    for(FichaAluno ficha : turma)
        cods.add(ficha.getNumero());
    return cods;
}
```

Verificar se um dado aluno cujo código é dado está inscrito na turma, consiste em realizar uma operação de pesquisa sequencial sobre as fichas dos alunos da turma, quer até encontrar tal número em cujo caso o aluno está inscrito, quer até esgotar o conjunto das fichas, em cujo caso o aluno não está inscrito. Dado que não se trata de um algoritmo que necessite garantidamente de percorrer todo o espaço de procura (todas as fichas), pois pode terminar a qualquer momento, ou seja, não é exaustivo ou de “varrimento”, deve ser usado um `Iterator<E>` porque o ciclo `for(...)` sobre coleções não permite incluir condições de paragem da iteração.

```
/** Verifica se um aluno de número dado existe */
public boolean existeAluno(String numAluno) {
    Iterator<FichaAluno> it = turma.iterator();
    FichaAluno ficha = null;
    String numero;
    boolean existe = false;
    while(it.hasNext() && !existe) {
        ficha = it.next();
        numero = ficha.getNumero();
        if(numero.equals(numAluno)) existe = true;
    }
    return existe;
}
```

Uma codificação alternativa para este método consistiria em criar o conjunto de todos os números dos alunos a partir das suas fichas e determinar se esse conjunto contém o número dado como parâmetro, o que se traduziria no seguinte código bem mais simples:

```
/** Verifica se um aluno de número dado existe */
public boolean existeNumero(String numAluno) {
    HashSet<String> cods = new HashSet<String>();
    for(FichaAluno fa : turma)
        cods.add(fa.getNumero());
    return cods.contains(numAluno);
}
```

Inserir um novo aluno na turma depois de se ter a garantia de que não está ainda inscrito (usando o método anterior no programa principal) consiste simplesmente em juntar a sua ficha à turma.

```
/** Insere um novo aluno na turma */
public void insereAluno(FichaAluno ficha) {
    turma.add(ficha.clone());
}
```

Determinar a maior média da turma trata-se mais uma vez de codificar um algoritmo que cai na classe dos algoritmos que têm por essência a iteração completa da coleção de fichas representadas neste caso no *arraylist*. O que pretendemos neste caso de cada uma delas? Obter a média do aluno. Para

quê? Para a comparar com a maior média até então encontrada e guardar a deste aluno se for superior. Finalmente, com que média deve ser comparada a média do primeiro aluno? Com uma média suficientemente baixa para que a dele seja superior de certeza (algoritmo de máximo).

```
/** Determina a melhor média */
public double maiorMediaTurma() {
    double maiorMedia = Double.MIN_VALUE;
    double mediaAluno;
    for(FichaAluno ficha : turma) {
        mediaAluno = ficha.getMedia();
        if(mediaAluno > maiorMedia)
            maiorMedia = mediaAluno;
    }
    return maiorMedia;
}
```

Criar a lista (ou o conjunto) dos números dos alunos com média superior à média dada como parâmetro continua a ser um problema que necessita de uma iteração completa (varrimento) sobre a coleção de fichas (resolvida com o ciclo `for(each)`), onde para cada ficha vamos consultar a média que vamos comparar com a média dada como parâmetro. Caso a média do aluno consultado seja maior guardamos o seu número num `HashSet<String>` que no final da iteração devolvemos como resultado do método.

```
/** Cria a lista com os códigos dos alunos com média
    superior à dada como parâmetro (exº média > 12).
*/
public HashSet<String> codAlsMedSup(double notaRef){
    HashSet<String> cods = new HashSet<String>();
    for(FichaAluno ficha : turma)
        if(ficha.getMedia() > notaRef)
            cods.add(ficha.getNumero());
    return cods;
}
```

A operação de remover a ficha de um aluno garantidamente existente, tal como a operação de inserir um novo aluno, pressupõe uma validação prévia da existência de tal aluno na turma. Sendo certo portanto que tal aluno existe, a operação vai consistir, antes de mais, na procura da posição ocupada pela ficha de tal aluno no *arraylist*, ou seja o valor do índice (entre 0 e o nº de fichas – 1) do *arraylist* em que se encontra guardada. Encontrado tal valor, e caso se tratasse de um *array* normal (cf. [] de C ou [] JAVA), em seguida deveríamos implementar algoritmos para eliminar tal ficha procurando reaproveitar o seu espaço da melhor maneira. A classe `ArrayList<E>` possui um método `remove(int i)` que, felizmente, faz tudo isto automaticamente, ou seja, remove o elemento que se encontra no índice dado como parâmetro e automaticamente faz o *shift-down* de todos os elementos em índices superiores ao removido, passando-os para o índice imediatamente inferior.

```
/** Remove a ficha de um aluno garantidamente existente */
public void removeAluno(String numAluno) {
    Iterator<FichaAluno> it = turma.iterator();
    FichaAluno ficha;
    String numero; int index = 0;
    boolean encontrado = false;
    while(it.hasNext() && !encontrado) {
        ficha = it.next();
        numero = ficha.getNumero();
        if(numero.equals(numAluno)) encontrado = true;
        else index++;
    }
    turma.remove(index);
}
```

O método que vai devolver a ficha do aluno cujo número é dado, começa por criar um iterador sobre as fichas da turma e enquanto existirem fichas e não tiver encontrado a ficha do aluno com tal número vai percorrendo a coleção. A variável *ficha*, que vai guardar cada uma das fichas percorrida, é inicializada a *null* pelo que, se não for encontrada nenhuma ficha com tal número o método devolverá o valor *null*.

Se a ficha de tal aluno for encontrada, então o ciclo *while* será abandonado e na variável *ficha* temos a ficha do aluno cujo número foi dado como parâmetro. Porém coloca-se a questão. Temos uma *cópia* dessa ficha ou temos uma *referência* (apontador) para ela? A resposta é que temos uma referência porque um iterador não copia a coleção que itera apenas referenciando – contendo os endereços - os respetivos elementos. Assim, se escrevêssemos *return ficha;* estaríamos a enviar para o exterior de *TurmaList* o endereço de memória da ficha deste aluno e quem o recebesse poderia modificar tal ficha de aluno e, portanto, o estado interno da turma também.

Por tal motivo, na linha de *return* e caso a variável *ficha* não seja *null*, ou seja, contenha uma qualquer ficha, realizamos o seu *clone()* para que não haja endereços partilhados, eliminando a possibilidade de acessos incontroláveis.

```
/** Devolve a Ficha completa do aluno de número dado */
public FichaAluno procuraAluno(String numAluno) {
    Iterator<FichaAluno> it = turma.iterator();
    FichaAluno ficha = null; String numero;
    boolean existe = false;
    while(it.hasNext() && !existe) {
        ficha = it.next();
        numero = ficha.getNumero();
        if(ficha.getNumero().equals(numAluno))
            existe = true;
    }
    return existe ? ficha.clone() : null;
}

/** Representação textual da Turma */
public String toString() {
    StringBuilder s = new StringBuilder();
    s.append("----- TURMA -----\\n");
    s.append("DISCIPLINA : "); s.append(nomeDiscp\\n);
    s.append("CODIGO : "); s.append(codigo\\n);
    s.append("----- ALUNOS -----\\n");
    for(FichaAluno ficha : turma) {
        s.append(ficha.toString());
    }
    s.append("-----\\n");
    return s.toString();
}

/** Cópia segura do recetor via construtor de cópia */
public TurmaLista clone() { new TurmaList(this); }
```

Tendo codificado todos os construtores e métodos de instância das duas classes que constituem as classes principais do projeto, resta apenas ter em consideração que tais classes, quer venham a ser completadas em ambiente JDK puro ou em ambiente de suporte BlueJ, deverão estar definidas num ficheiro de texto com exactamente o mesmo nome da classe e extensão *.java*.

PROTOTIPAGEM EM BLUEJ

Não sendo obrigatório nem necessário para a realização dos projetos que se têm apresentado e que se vão apresentar em seguida usar o IDE BlueJ, existem no entanto algumas vantagens que, principalmente numa primeira fase de desenvolvimento de projetos, são de salientar.

A possibilidade de desenvolvermos o projeto de *forma iterativa* (ou seja, passo a passo) é talvez a mais interessante, pois permite que a cada momento tenhamos a garantia de que o código de cada método está corretamente desenvolvido. Quando dizemos de forma iterativa, tal significa exatamente podermos criar instâncias de uma dada classe imediatamente após a definição dos seus construtores e, em seguida, método a método, desenvolver o código do método e testá-lo para verificação da sua correção. Os IDEs mais sofisticados e mais profissionais não permitem a criação rápida de instâncias pelo que não facilitam este desenvolvimento iterativo. Porém, destinam-se a equipas de profissionais e para estes são de extrema utilidade.

O BlueJ permite ainda a realização de I/O simples através de uma janela designada Terminal Window.

Apresentaremos para o projeto exemplo, através de algumas figuras comentadas, a forma de realizar a sua execução no ambiente BlueJ.

CLASSE DE TESTE

A criação de uma classe de teste será algo que faremos em todos os projectos apresentados, não apenas porque usando o método `toString()` em certas circunstâncias nos permitirá ter uma visualização do estado dos objetos de interesse, como nos permitirá no caso de objetos mais complexos criar uma instância que é devolvida de imediato ao ambiente BlueJ para ser em seguida testada.

A classe de teste deverá ser desenvolvida mesmo por aqueles que não vierem a usar o ambiente BlueJ, em cujo caso apenas não necessitarão de fazer `return` de tal instância, tal como veremos a seguir.

```
/** Classe de teste do projeto TurmaList */
import java.util.ArrayList;
public class TestTurmaList {

    public static TurmaList main() {

        FichaAluno ficha1, ficha2, ficha3, ficha4;
        ArrayList<String> discp = new ArrayList<String>();
        discp.add("LPII"); discp.add("MPII");
        discp.add("CGI"); discp.add("SC");
        ficha1 = new FichaAluno("11", "Rita", 15, discp);

        discp.clear(); // limpa o arraylist para inserir novas
        discp.add("LPII"); discp.add("MPII");
        discp.add("SOII"); discp.add("AQSI");
        ficha2 = new FichaAluno("22", "Pedro", 16, discp);

        discp.clear();
        discp.add("AMI"); discp.add("AMII");
        discp.add("CL"); discp.add("ALG");
        ficha3 = new FichaAluno("33", "Rui", 12, discp);

        discp.clear();
        discp.add("LPIV"); discp.add("ALG");
        discp.add("SOII"); discp.add("AQSI");
        ficha4 = new FichaAluno("44", "Paula", 12, discp);

        TurmaList turma1 =
            new TurmaList("501345", "LAB-POO-JAVA7");

        turma1.insereAluno(ficha1); turma1.insereAluno(ficha2);
        turma1.insereAluno(ficha3); turma1.insereAluno(ficha4);

        return turma1; // entrega a instância ao BlueJ
    }
}
```


EXECUÇÃO E TESTE EM BLUEJ

Criado o projeto **Turma_JAVA** e, através do editor do BlueJ, criadas as classes `FichaAluno`, `TurmaList` e `TestTurmaList`, o **Diagrama de Classes** do projeto, gerado automaticamente pelo BlueJ, é o que se apresenta na figura seguinte.

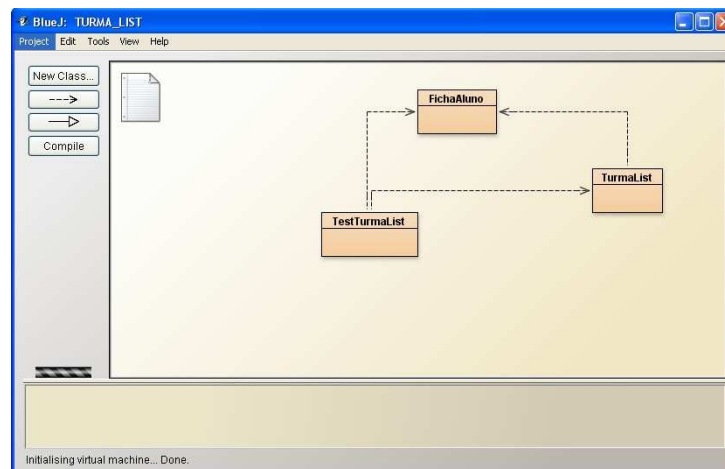

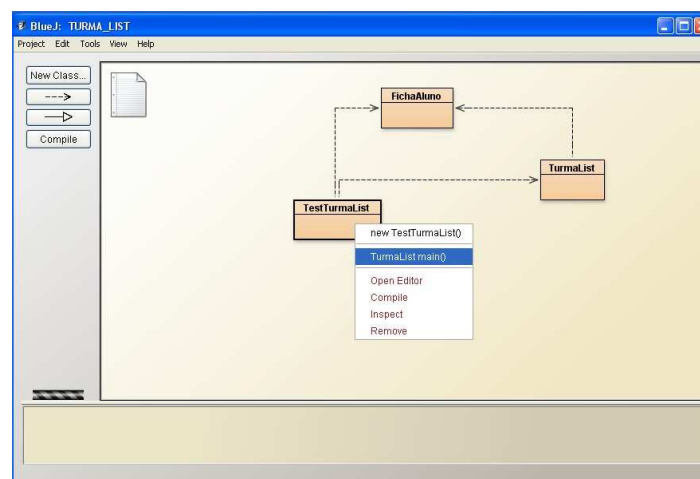



Diagrama de Classes do projeto TurmaList

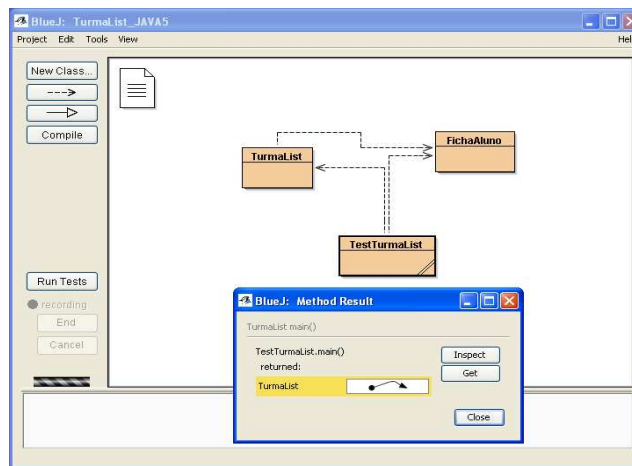
Podemos a partir deste momento criar instâncias de qualquer das classes `FichaAluno` e `TurmaList` usando os seus construtores. Quer num caso quer noutro, a criação de tais instâncias a partir do zero, e sendo tais classes compostas por variáveis de instância que são estruturadas, implica algum trabalho paralelo. A criação de classes auxiliares que fazem tal trabalho revela-se nestes projetos compensador, tal como no exemplo a classe `TestTurmaList`.

Seleccionada a classe `TestTurmaList` e accionado  (click no botão direito) surge o menu contendo todas as operações realizáveis sobre esta classe. Dentre estas, estamos particularmente interessados em seleccionar a execução do método `main()` que, como a sua assinatura indica, devolverá como resultado uma instância de `TurmaList`.



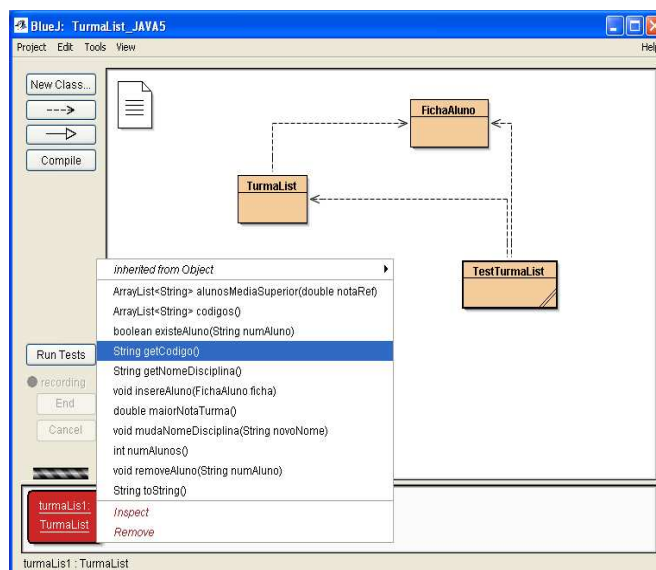
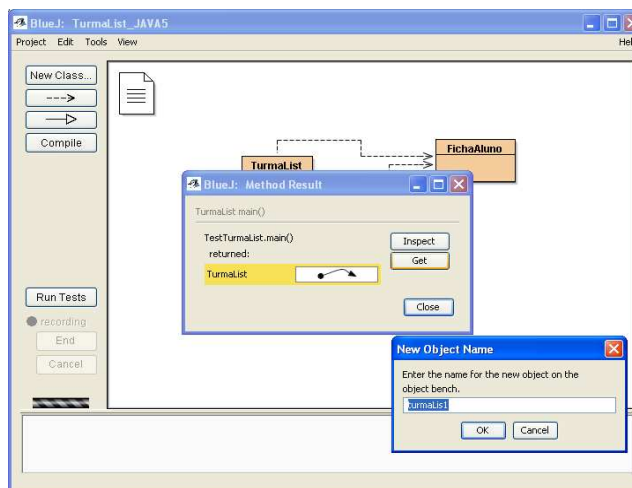
Invocação de main()

Para tal, depois de seleccionada a operação basta fazermos  para que a mesma seja executada e neste caso surja no ecrã uma janela com o resultado da execução do método, ou seja, um objeto do tipo `TurmaList` ao qual deveremos atribuir um nome.



Criação de instância de TurmaList

Vamos fazer o **Get** do objecto e atribuir-lhe um nome ou aceitar o nome proposto pelo BlueJ.



API de TurmaList

Criada desta forma simples a instância de TurmaList – note-se como seria muito mais complexo criar 6 fichas de aluno, colocá-las no espaço de instâncias uma a uma, criar uma turma vazia e inserir

as fichas uma a uma, etc.-, e colocada no espaço de instâncias, podemos agora, tal como poderíamos em qualquer momento do projeto desde que já tivéssemos pelo menos os construtores, testar cada um dos métodos de instância da classe `TurmaList` (admite-se que já o havíamos feito com `FichaAluno`).

Na imagem acima é também visível o conjunto de mensagens a que a instância de `TurmaList` é capaz de responder, ou seja, o conjunto de métodos implementados.

Vamos testar agora as principais operações do projeto. Para cada operação iremos mostrar a sua invocação ou já a fase de introdução de parâmetros se tal for o caso, e o resultado da mesma. Caso o resultado seja mais estruturado utilizaremos ***Inspect*** para analisarmos os conteúdos dos objectos resultantes.

EXERCÍCIOS:

ExA: Compreensão do funcionamento de `ArrayList<E>`

Crie no BlueJ, uma pasta de projecto de nome `TesteArrayList` e importe para tal pasta a classe `Ponto2D` anteriormente desenvolvida. Em seguida, usando `Tools/Use Library Class...`, crie uma instância vazia de um `ArrayList<String>` e outra de um `ArrayList<Ponto2D>`. De seguida, com ambas as instâncias de `arraylist` criadas, invoque os métodos da API de `ArrayList<E>`, cf. `add()`, `get()`, `addAll()`, `remove()`, `removeAll()`, `size()`, `indexOf()`, `contains()`, `retainAll()`, etc., e conclua sobre a semântica de cada método usando `INSPECT` após a sua invocação.

ExB: Compreensão do funcionamento de `HashSet<E>` e de `TreeSet<E>`

Crie no BlueJ, uma pasta de projeto de nome `TesteHashSet` e importe para tal pasta a classe `Ponto2D` anteriormente desenvolvida. Em seguida, usando `Tools/Use Library Class...`, crie uma instância vazia de um `HashSet<String>` e outra de um `HashSet<Ponto2D>`. Em seguida, com ambas as instâncias de `hashset` criadas, invoque os métodos da API de `HashSet<E>`, cf. `add()`, `remove()`, `size()`, `addAll()`, `removeAll()`, `contains()`, `containsAll()`, `retainAll()`, etc., e conclua sobre a semântica e efetivo funcionamento de cada método usando `Inspect` após a sua invocação. Tenha em especial atenção os métodos `add()` e `addAll()` já que num conjunto não são admitidos elementos em duplicado.

Posteriormente crie uma `TreeSet<String>` e uma `TreeSet<Integer>` e verifique, usando `Inspect` ou criando um programa de teste, a ordem dos respectivos elementos.

Ex 1: Desenvolva uma classe `Plano` que represente um conjunto de pontos 2D de um plano cartesiano num `ArrayList<Ponto2D>` (a representação correcta seria um conjunto mas como ainda não estudamos como representar conjuntos em JAVA usaremos um `arraylist`).

Desenvolva, para além dos construtores e métodos usuais, métodos que implementem as seguintes funcionalidades:

- Determinar o número total de pontos de um plano;
- Adicionar um *novo* ponto ao plano e remover um ponto caso exista;
- Dado um `ArrayList` de `Pontos2D`, juntar tais pontos ao plano receptor;
- Determinar quantos pontos estão mais à direita ou mais acima (ou ambos) relativamente ao `Ponto2D` dado como parâmetro;
- Deslocar todos os pontos com coordenada em `XX` igual a `cx`, dada como parâmetro, de `dx` unidades, igualmente dadas como parâmetro (alterar os pontos portanto);
- Dado um plano como parâmetro, determinar quantos pontos são comuns aos dois planos;
- Criar a lista contendo os pontos comuns ao plano receptor e ao plano parâmetro;
- Criar uma lista contendo todos os pontos do plano com coordenada em `XX` inferior a um valor dado como parâmetro (atenção, pretende-se obter cópias dos originais);
- Criar um novo plano que contenha os pontos comuns entre o plano receptor e um plano dado como parâmetro;
- Não esquecer os métodos `equals()`, `toString()` e `clone()`.

Ex 2: Uma ficha de informação de um país, `FichaPais`, contém 3 atributos: nome do país, continente e população (real, em milhões). Crie uma classe `ListaPaíses` que permita criar listas de `FichaPais`, por uma ordem qualquer, e implemente os seguintes métodos:

- Determinar o número total de países;
- Determinar o número de países de um continente dado;
- Dado o nome de um país, devolver a sua ficha completa, caso exista;
- Criar uma lista com os nomes dos países com uma população superior a um valor dado;

- Determinar a lista com os nomes dos continentes dos países com população superior a dado valor;
- Determinar o somatório das populações de dado continente;
- Dada uma lista de FichaDePaís, para cada país que exista na lista de países alterar a sua população com o valor na ficha; caso não exista inserir a ficha na lista;
- Dada uma lista de nomes de países, remover as suas fichas;

Ex 3: Uma **Stack (ou pilha) é uma estrutura linear do tipo LIFO (“last in first out”), ou seja, o último elemento a ser inserido é o primeiro a ser removido. Uma stack possui assim apenas um extremo para inserção e para remoção. Implemente uma Stack de nomes, com as usuais operações sobre stacks:**

- String top(): que determina o elemento no topo da stack;
- void push(String s): insere no topo;
- void pop(): remove o elemento do topo da stack, se esta não estiver vazia;
- boolean empty(): determina se a stack está vazia;
- int length(): determina o comprimento da stack;

Ex 4: Cada e-mail recebido numa dada conta de mail é guardado contendo o endereço de quem o enviou, a data de envio, a data de recepção, o assunto e o texto do mail (não se consideram anexos, etc.). Crie a classe **Mail que represente cada um dos mails recebidos.**

Em seguida crie uma classe designada **MailList que permita guardar todos os actuais e-mails existentes numa dada conta, e implemente as seguintes operações sobre os mesmos:**

- Determinar o total de mails guardados;
- Guardar um novo mail recebido;
- Determinar quantos mails têm por origem um dado endereço;
- Criar uma lista contendo os índices dos mails que no assunto contêm uma palavra dada como parâmetro (qualquer que seja a posição desta);
- O mesmo que a questão anterior, mas criando uma lista contendo tais mails;
- Eliminar todos os e-mails recebidos antes de uma data que é dada como parâmetro;
- Criar uma lista dos mails do dia;
- Dada uma lista de palavras, eliminar todos os mails que no seu assunto contenham uma qualquer destas (anti-spam);
- Eliminar todos os mails anteriores a uma data dada;

Ex 5: Desenvolva uma classe CdRecPlay que represente o comportamento de um leitor e gravador de faixas musicais armazenadas num CDRW. A classe CdRecPlay deve reconhecer o CDRW e ser capaz de aceder às várias faixas musicais do CDRW.

A classe Faixa possui uma informação textual sobre a música e ainda o tempo de reprodução. Cada CDRW será representado pela classe CdRW na qual cada CDRW é representado por um título e pela lista de faixas que contém. As instâncias de CdRW devem possuir o seguinte comportamento:

- Devolver uma faixa do CD dado o seu número;
- Devolver o número total de faixas do CD;
- Permitir adicionar uma nova faixa ao CD no fim deste;
- Adicionar uma lista de novas faixas ao CD no fim deste;
- Apagar do CD a faixa de número dado reposicionando todas as outras;
- Determinar o tempo total das músicas do CD;

Cada instância de CdRecPlay possui um título do que CD que está inserido, a lista de faixas que pode reproduzir e o número da faixa atual (próxima a reproduzir - *play*).

Desenvolva, para além dos construtores e métodos usuais, métodos que implementem as seguintes funcionalidades para a classe `CdRecPlay`:

- Simular a inserção de um CD;
- Simular a ejeção um CD, limpando toda a informação atual;
- Inserir uma nova faixa no final da lista de faixas;
- Inserir uma lista de novas faixas no final da lista atual de faixas;
- Devolver toda a informação respeitante à faixa atual;
- Devolver a informação da faixa atual como uma `String`;
- Apagar a faixa cujo número é dado como parâmetro;
- Modificar a faixa atual para a faixa de número dado;
- Determinar o tempo total de música disponível;
- Simular a reprodução de uma música no dispositivo.

Ex 6: Uma empresa transportadora tem permanentemente atualizada uma lista de todos os transportes de carga realizados pelos seus camiões, lista essa cuja ordem corresponde à conclusão de tal transporte. Assim, o último transporte concluído é o último elemento da lista. A informação sobre cada transporte realizado por cada camião consiste do seu código de transporte, da matrícula do camião, da data (dia, mês e ano) do transporte, do total de quilómetros realizados e do conjunto das mais diversas cargas.

Cada carga é representada por um código único, pelo seu custo individual de transporte por Km, pelo número de Kms de transporte até ao seu destino, por um código do respetivo cliente e por um nome identificativo da carga.

Desenvolva as classes `Carga` e `Transporte`, implementando para esta última as seguintes funcionalidades:

- Juntar mais uma carga a um transporte;
- Remover uma carga de um transporte;
- Carregar um transporte com um conjunto de cargas;
- Determinar a faturação total de um transporte;
- Alterar o número de quilómetros efectuados;

Implemente em seguida a classe `ListaTransportes` que regista, em sequência, cada um dos transportes realizados, e que deverá disponibilizar as seguintes operações:

- Realizar o registo de mais um transporte;
- Adicionar à lista um conjunto de transportes;
- Listar todos os transportes realizados num dado mês de dado ano;
- Determinar o total de quilómetros realizados pelo camião de matrícula dada;
- Determinar uma lista de pares Código de Cliente – Total Pago, que indica por cliente quanto este pagou à empresa num dado intervalo de tempo;
- Listar as matrículas de todos os camiões envolvidos;
- Listar por matrícula o total de quilómetros de cada camião.

Ex 7: A informação sobre cada empregado de uma empresa consiste de: código, nome, secção, salário por dia e dias de trabalho no último mês.

Crie uma classe `EmpresaList` que guarde os registos dos seus empregados por uma ordem qualquer e sobre tal informação realize as seguintes operações:

- Inserir a ficha de um novo empregado;
- Remover um empregado dado o seu código;
- Consultar a ficha de um empregado dado o seu código;

- Para todos os empregados cujos códigos são dados num conjunto passado como parâmetro, determinar os seus salários actuais;
 - Listar os códigos de todos os empregados de uma dada secção;
 - Calcular o total de vencimentos a pagar no mês actual;
 - Admitindo meses de 20 dias de trabalho, calcular por secção o total de faltas dadas no mês actual;
 - Fazer reset dos dias de trabalho actuais de todos os empregados;
 - Determinar o total de empregados com vencimento superior a um valor dado;
 - Guardar num ficheiro de texto indicado a lista de pagamentos do mês dado, com número, nome e salário do mês de todos os funcionários.
-