8 - Colecções e Tipos Parametrizados

8.1 Introdução

Estudámos até agora todas as construções que nos permitem desenvolver classes JAVA ou interfaces JAVA a serem implementadas por classes. Foi feita a distinção entre o papel das classes abstractas e das interfaces no desenho de aplicações, e ficámos a saber que as interfaces existem numa hierarquia distinta da hierarquia de classes, hierarquias que se relacionam semanticamente através de uma relação de implementação.

No entanto, e aparte as referências feitas aos *arrays* de JAVA, que não são objectos, não estudámos ainda mecanismos ou construções que nos permitam realizar agrupamentos de objectos em estruturas que os representem como uma unidade de informação, possuindo certas propriedades e podendo ser manipulada através de certas operações.

Ainda que o termo **estrutura de dados** seja tradicionalmente empregue para designar os agrupamentos de entidades que não são objectos mas valores, se alargarmos um pouco o seu sentido, poderemos dizer que nos falta agora estudar estruturações de objectos, que são, ao nível da PPO, os contentores da nossa informação.

Assim, vamos dedicar os próximos capítulos ao estudo de classes que representam em JAVA estruturas de dados predefinidas, que podemos reutilizar nos nossos programas na criação de classes mais complexas de que necessitamos na resolução de problemas de maior complexidade também.

Estas classes de JAVA designam-se por **colecções** (*collections*), representam as mais usuais estruturas de dados necessárias no desenvolvimento de aplicações, possuem características próprias que devemos conhecer para delas podermos tirar o melhor partido, são bastante genéricas e são muito eficientes. Com a possibilidade de agruparmos objectos em unidades de informação compactas e estruturadas que representam entidades do mundo real que têm características semelhantes, elevamos em muito o nível da nossa programação, pois temos à nossa disposição excelentes estruturas de representação de dados.

8.2 Colecções de JAVA5

Desde a versão original JDK1.0 de 1996, que JAVA possui no seu *package* `java.util` classes que implementam algumas das estruturas de dados, em conjunto com as respectivas operações (algoritmos), mais utilizadas no desenvolvimento de aplicações.

Porém, é com o aparecimento do JDK1.2, em 1999, que se dá uma verdadeira revolução no *package* `java.util`, com o aparecimento de uma estrutura de interfaces, classes abstractas e classes concretas, organizadas no sentido de oferecer ao programador uma grande diversidade de implementações eficientes das mais comuns estruturas de dados e entidades matemáticas, designada ***JAVA Collections FrameWork*** (JCF), muitas vezes até apenas ***Collections***.

Uma **colecção** (noutras linguagens designada *container*) é um objecto capaz de agrupar múltiplos elementos numa unidade única de representação e processamento devidamente organizada, possuindo propriedades de estrutura e funcionamento próprias. Por exemplo, como certamente sabemos, uma estrutura *stack* (ou pilha) é uma estrutura linear, ou seja existe uma ordem sequencial nos seus elementos, e funciona segundo uma lógica de que o último elemento inserido será o primeiro a ser removido. Estas são as propriedades de uma qualquer *stack*. A forma de implementarmos as propriedades são inúmeras, e estas propriedades são independentes dos conteúdos da estrutura de dados.

A ***Collections FrameWork*** é uma arquitectura unificada, constituída por interfaces, classes abstractas, implementações (classes concretas) e métodos (algoritmos), visando representar e manipular colecções, normalizando as API, oferecendo conversões entre estruturas, reduzindo o esforço de aprendizagem e de programação, e oferecendo grandes soluções para reutilização.

A importância deste *framework* para a criação de aplicações em JAVA é absolutamente crucial, dado que a maioria das aplicações que desenvolvemos usam muitas das classes implementadas no JCF. No entanto, e dada a proliferação destas e as diferentes especializações das suas implementações em termos de eficiência e utilização, a escolha da implementação adequada implica um conhecimento bastante aprofundado de todas as diferentes ***Colecções*** que o JCF coloca ao nosso dispor.

Vamos assim, e numa primeira abordagem, sintetizar a estrutura interna do JCF, do ponto de vista dos seus próprios autores, que a dividem nas seguintes infra-estruturas:

- **Utilitários para *arrays*:** Por exemplo a classe `Arrays` já antes referida, que presta serviços algorítmicos a *arrays*;
- **Algoritmos:** Métodos estáticos da classe **`Collections`** que prestam serviços às mais diversas colecções de objectos, como ordenação, procura, etc.;
- **Classes abstractas:** Implementações parciais de certas classes que satisfazem certas interfaces, para que sejam complementadas (implementadas) de diversas formas distintas pelas subclasses concretas;
- ***Wrapper*, *Concurrent*, *Special-purpose implementations***: Implementações que adicionam funcionalidade a outras, como métodos para conversão de e para tipos primitivos, sincronização ou gestão de concorrência, restrições, etc.;
- ***Legacy implementations***: Classes de anteriores implementações que, em vez de serem desactivadas (*deprecated*), foram reajustadas para satisfazerem as interfaces do JCF;
- ***General-purpose implementations*:** Classes do JCF que são as primeiras implementações das interfaces originais;
- ***Collection Interfaces*:** As API fundamentais que estruturam todas as classes do JCF, dividindo-as em quatro grandes famílias de estruturas: os conjuntos, as listas, as correspondências (*maps* ou *mappings*) e as filas. Estas quatro interfaces são a base de todo o JCF em termos de **tipos das colecções**.

A ***Collections Interface*** define quatro interfaces fundamentais, de tal forma que as classes que as implementam são estruturas com as seguintes propriedades:

1. **Set**: conjuntos de objectos no sentido matemático, logo, não existindo a noção de posição, conforme "primeiro", "último", etc., nem sendo permitidos elementos em duplicado;
2. **List:** sequências de objectos, logo, existindo a noção de ordem, cf. "primeiro", "último", "n-ésimo", etc., permitindo elementos duplicados;
3. **Map:** os *mappings*, que são correspondências unívocas um para um entre objectos (representando funções matemáticas finitas), em geral correspondências do tipo chave-valor, em que as chaves são o **domínio** e são um conjunto (logo, não havendo chaves em duplicado). A correspondência é apenas unívoca, porque a chaves diferentes pode ser feito corresponder o mesmo valor;
4. **Queue:** as filas são estruturas lineares do tipo FIFO (*first in first out*), com métodos próprios para inserção e remoção, destinadas em geral a conter elementos em estado de espera para serem processados.

Em finais de 2004, foi lançado o JDK1.5 Tiger (JAVA5), versão que veio introduzir na linguagem JAVA as maiores modificações sintácticas, e não só, desde o lançamento da mesma. Estas modificações praticamente não alteraram a JVM, à excepção do código gerado para as classes. No entanto, a nível sintáctico e ao nível dos tipos de dados muito mudou, passando a verificação de tipos a ser toda realizada em tempo de compilação.

São três as modificações introduzidas em JAVA5 que se relacionam directamente com as colecções:

- **Auto-Boxing** e **Auto-Unboxing**: mecanismo que converte automaticamente valores de tipos primitivos (cf. `int`, `double`, etc.) em objectos das classes correspondentes (*wrapper classes*) quando estes são inseridos

em colecções, e faz a conversão contrária quando instâncias de classes *wrapper* são lidas de colecções, evitando assim a necessidade de programar operações de *casting*.

- **Iterador *foreach***: sob a forma de um ciclo `for` aplicável a colecções que podem ser iteradas (`Iterable`), substituindo os iteradores explícitos em muitas circunstâncias;
- **Tipos genéricos:** que adicionam às colecções generalidade e segurança em tempo de compilação, e eliminam o intenso *casting* que era necessário.

A modificação de maior impacto foi a que implicou a alteração, ainda que não em termos de estrutura, de todo o JCF: a introdução dos **tipos genéricos** (***Generics*** no original).

A estrutura de interfaces, para além da sua parametrização e da introdução da interface `Queue<E>`, é a mesma que existia anteriormente e que se apresenta na Figura 8.1:

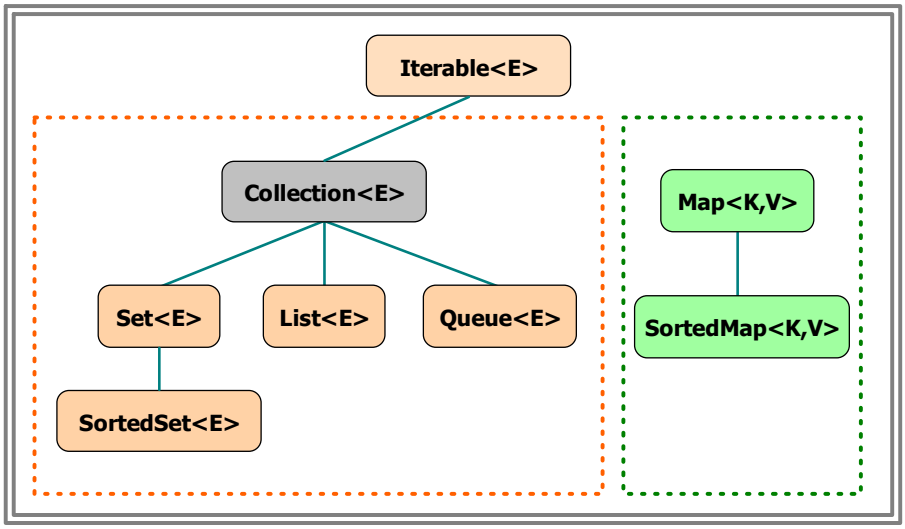


Figura 8.1 – Interfaces do JCF - Tipos de colecções

Em primeiro lugar, uma questão de notação e leitura: `Collection<E>` (ou, em geral, `I<E>`) deve ler-se "`Collection` **de elementos de tipo** `E`" ou apenas "uma `Collection` de `E`", em que `E` é o tipo dos elementos, por exemplo, `String`, `Integer`, `Ponto`, etc.

A primeira constatação a tirar da Figura 8.1 é que as correspondências (ou *maps*) não herdam da interface `Collection<E>` pelo que, em extremo rigor, não seriam colecções. Claro que os *maps* são colecções de objectos, são compatíveis com outras colecções, mas, como são parametrizados por dois tipos, não podem herdar de `Collection<E>` que apenas aceita um parâmetro. Ao longo do texto, sempre que falarmos de ***colecções*** de JAVA, os *maps* estarão incluídos em tal classificação mais lata.

Assim, em JAVA5, colecções monoparâmetro ou são **listas**, a hierarquia de `List<E>`, ou **conjuntos**, a hierarquia de `Set<E>`, ou **filas**, a hierarquia de `Queue<E>`. Todas as classes que são implementações de **`C<E>`** são `C`s implementadas de forma distinta, mas todas "falam" a mesma "linguagem" definida pela sua interface (API). Vantagem óbvia: depois de estudarmos uma implementação de **lista**, uma de *conjunto*, uma de **fila** e uma de ***map***, sabemos a API de praticamente todas, e apenas necessitaremos de saber quais as suas "melhores" implementações (especializações) para cada tipo de problema.

Neste capítulo, a nossa atenção estará centrada no conhecimento destas colecções e no seu estudo visando apenas a sua eficaz utilização na criação das nossas próprias classes e aplicações. Em capítulo posterior, veremos como podemos desenvolver as nossas próprias classes parametrizadas, sejam agrupamentos de objectos ou não.

Tal como vimos no capítulo das interfaces, sendo conveniente recordar aqui desde já, uma interface é um tipo, e uma qualquer classe que implemente tal tipo é compatível com esse tipo, pelo que as atribuições são válidas. Assim, é válida em termos genéricos a atribuição `List<E> = new implementação_de_List<E>()`, para qualquer `E` de um tipo referenciado.

Vamos estruturar o nosso estudo neste capítulo em três etapas:

1. Estudar a interface `List<E>` e uma das suas implementações;
2. Estudar a interface `Map<K,V>` e duas das suas implementações;
3. Estudar a interface `Set<E>` e suas implementações.

Em conjunto com a etapa 1, introduziremos o indispensável estudo dos tipos genéricos de JAVA e dos tipos parametrizados e sua utilização, que é um conhecimento fundamental para trabalharmos com toda a segurança, e de modo genérico e extensível com estas colecções de JAVA.

8.3 TIPOS GENÉRICOS

Um **tipo genérico** é um tipo referenciado (classe ou interface) que usa na sua definição um ou mais tipos de dados (não primitivos) como parâmetro, tipos parâmetro esses que serão mais tarde substituídos por tipos concretos quando o tipo genérico for *instanciado*.

A Figura 8.2 ilustra, através de um exemplo, as duas componentes de um **tipo genérico**.

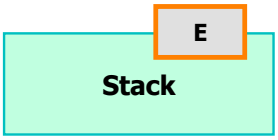


Figura 8.2 – Tipo genérico Stack<E>

Um **tipo genérico** tem por parâmetros uma ou mais variáveis "de tipo" (**parâmetros de tipo**), e é definido com base nestas variáveis de valor não especificado (formais). A Figura 8.2 ilustra o tipo `Stack<E>`, em que a variável parâmetro `E` representa uma infinidade de possíveis tipos concretos (excepto tipos primitivos) ou parâmetros actuais de `Stack`.

Um **tipo parametrizado** é o resultado de uma instanciação do tipo genérico para um valor concreto da variável de tipo `E`, por exemplo, `Stack<String>`, `Stack<Ponto>`, etc. O tipo parametrizado resultante da instanciação poderá então ser usado em declarações de variáveis, de parâmetros de métodos ou de tipos de resultado destes. É convenção usar-se uma única letra para os "parâmetros de tipo" dos tipos genéricos, em geral a letra `E` para parâmetros de tipo de colecções, e `T` e outras para os parâmetros de tipo de classes que não são colecções, por exemplo, `Par<T, U>`.

Os tipos genéricos existem há muitos anos em muitas linguagens de programação sob várias formas de implementação, quer em linguagens funcionais (como ML, 1976) quer em linguagens orientadas aos objectos (como *packages* em ADA, *templates* em C++ e "genéricos" em C# 2.0), podendo mesmo, numa solução pobre, ser emulados por simples substituições de texto pré-compilação. Pode fazer-se o exercício com o texto de definição de `Ponto<T>` a seguir, que, apenas com boas substituições textuais, daria para criar em JAVA diversas classes `Ponto` de diferentes tipos:

```
public class Ponto<T> { // tipo genérico Ponto<T>
   private T x;
   private T y;
   // Métodos
   public Ponto(T cx, T cy) { x = cx; y = cy; }
   public T daX() { return x; }
   public T daY() { return y; }
   public void setX(T cx) { x = cx }
   . . .
}
```

Como se pode verificar pelo exemplo, nas **classes genéricas**, os métodos são também definidos tendo os tipos parâmetro por base, tal como os parâmetros e os resultados.

A existência ou não de tipos genéricos não é, só por si, vantagem ou desvantagem, e este exemplo não só serve para dar uma ideia da estrutura interna do tipo de *template* que terá sempre um tipo genérico, como também para

mostrar que o problema não está na sintaxe, nem na forma, mas na semântica, em especial na teoria de tipos associada à verificação da sua segurança pelo compilador, e até na execução. Quanto a estes pontos, os tipos parametrizados de JAVA5 têm características próprias que referiremos no final do capítulo, depois de conhecermos melhor as formas como os podemos utilizar.

Os **tipos genéricos** podem, em JAVA, ser quer **classes** quer **interfaces**. Em JAVA5, todas as classes e interfaces de JCF são parametrizadas, salvo as excepções, os tipos enumerados e classes especiais ditas anónimas. No entanto, e por questões que têm a ver com a designada retrocompatibilidade, por cada classe `C<E>` ou interface `I<E>` genérica, foram mantidas as versões antigas sem parâmetros `C` e `I`, que não são *type-safe*, ou seja, seguras em termos de tipos, e que não estudaremos nem devem ser usadas pois irão desaparecer em futuras versões da linguagem. Estes tipos designam-se por ***raw types***. São exemplos os tipos (interfaces) `List` e `Set` e, entre outras, as implementações (classes) `ArrayList` e `HashSet`. Estas colecções, não tipadas, permitiam inserir um qualquer `Object`, sendo portanto heterogéneas, pois poderíamos ter na mesma colecção instâncias de múltiplas classes. Ao ler destas colecções seria sempre extraído um `Object`, devendo o programador realizar o *casting* para o tipo que esperava que o objecto tivesse de facto. Nada era pois verificado pelo compilador, a não ser que tudo era do tipo `Object`.

Porque ainda existem as duas versões das classes e interfaces, parametrizadas e não parametrizadas, ao longo do texto, haverá sempre o cuidado de escrever `C<E>` ou `C<>` para que não se confundam com o *raw type* `C`.

A variável de tipo `E,` em `Stack<E>` acima, pode ser também instanciada com um tipo parametrizado, por exemplo, `List<String>`, conduzindo a um tipo parametrizado de segundo nível, como `Stack<`**`List<String>`**`>`, uma *stack* de listas de `String`.

Quando todas as variáveis de tipo de um tipo genérico são instanciadas com tipos concretos (**argumentos de tipo**), obtém-se um **tipo parametrizado concreto**. Como veremos posteriormente, há certas instanciações de tipos genéricos que conduzirão não a um tipo parametrizado concreto, mas a famílias de possíveis concretizações.

8.4 LISTAS DE TIPO E: `List<E>`

Se uma interface define o conjunto de métodos (API) a implementar por uma classe que pretenda possuir tais características ou propriedades, então, a interface `Collection<E>` define o conjunto de métodos que qualquer grupo de objectos deverá implementar para que seja do tipo `Collection<E>`. Como podemos ver pela Figura 8.1, os *mappings*, apesar de serem colecções no sentido lato, não são do tipo `Collection<E>`, pois necessitam de duas variáveis de tipo, `K` e `V`, e os seus métodos possuem assinaturas em conformidade com as características deste tipo particular de estruturas de dados.

Vamos de momento centrar a nossa atenção na interface **`List<E>`**. Esta interface herda de `Iterable<E>` e de `Collection<E>`. Todas as classes que implementam `List<E>` respondem aos métodos do Quadro 8.1:

Categoria de Métodos	API de List<E>
Inserção de elementos	`add(E o); add(int index, E o)` `addAll(`*`Collection`*`); addAll(int i, `*`Collection`*`);`
Remoção de elementos	`remove(Object o); remove(int index);` `removeAll(`*`Collection`*`); retainAll(`*`Collection`*`)`

Consulta e comparação de conteúdos	`E get(int index); int indexOf(Object o);`
	`int lastIndexOf(Object o);`
	`boolean contains(Object o); boolean isEmpty();`
	`boolean containsAll(`*`Collection`*`); int size();`

Criação de Iteradores	`Iterator<E> iterator();`
	`ListIterator<E> listIterator();`
	`ListIterator<E> listIterator(int index);`

Modificação	`set(int index, E elem); clear();`
Subgrupo	`List<E> sublist(int de, int ate);`
Conversão	`Object[] toArray();`
Outros	`boolean equals(Object o);`

Quadro 8.1– API de `List<E>`

A implementação de cada um destes métodos deve satisfazer o facto de que um "grupo de objectos" de tipo `List<E>` possui as seguintes propriedades:

- Existe uma ordem nos seus elementos, ou seja, é uma sequência ou lista de objectos de tipo `E;`
- Cada elemento ocupa uma posição na lista referenciada por um índice inteiro, a partir de 0;
- Os elementos podem ser inseridos em qualquer posição da lista;
- A lista pode conter elementos em duplicado e elementos **`null`**.

Existem quatro classes (exceptuando as classes de gestão) que implementam de formas diferentes a interface `List<E>`, ou seja, são implementações parametrizadas de listas:

- `ArrayList<E>` (implementação em *array* dinâmico);
- `Vector<E>` (implementação de lista usando um *array* dinâmico);
- `Stack<E>` (estrutura com comportamento LIFO, *last-in-first-out*),
- `LinkedList<E>` (implementação em lista duplamente ligada);

Conforme se pode verificar observando a Figura 8.3, todas estas classes herdam da classe abstracta `AbstractList<E>` implementações parciais que completam por forma a serem classes concretas e a satisfazerem a interface `List<E>`. A classe `LinkedList<E>` é ainda subclasse de outra classe abstracta que lhe impõe acesso sequencial (cf. "anterior" e "seguinte"). `Vector<E>` e `ArrayList<E>` são implementações semelhantes de listas, apenas diferindo pelo facto de que a classe `Vector<E>` é uma implementação especial designada por `synchnonized`.

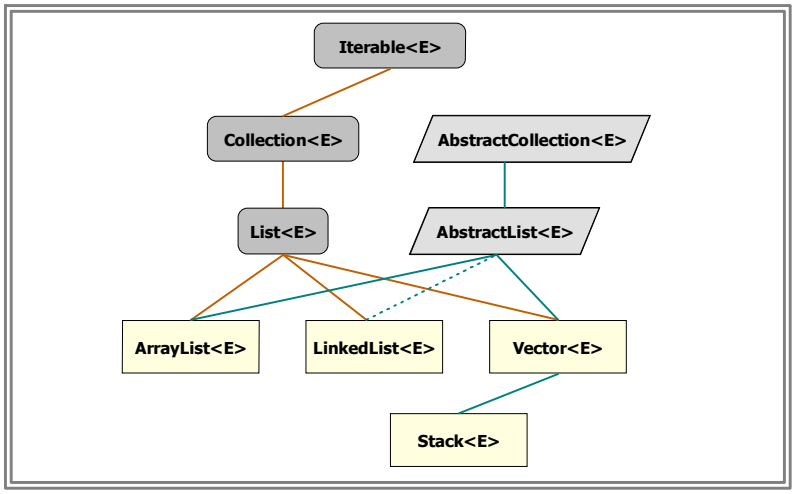


Figura 8.3 – Implementações de List<E>

Uma classe com implementação `synchronized` é uma classe cujo código está preparado para ser executado num ambiente onde podem existir diversas *threads* (processos *light* ou tarefas) ou mesmo processos concorrentes, nos quais os acessos a uma estrutura de dados deve ser convenientemente escalonado e esta mantida coerente para os processos que lhe vão aceder posteriormente. Naturalmente que se não estamos a programar aplicações com este tipo de exigências, como é o nosso caso, não precisaremos de pagar o ónus da sincronização, que tem algum peso na eficiência. Em todo o caso, qualquer classe não sincronizada de raiz, pode ser tornada *synchronized* pelo programador.

A classe `Stack<E>` é implementada à custa de `Vector<E>` sendo, assim, sua subclasse.

Estudaremos como exemplo de implementação de listas a classe `ArrayList<E>`, que é a mais utilizada, entre outras razões pela sua eficiente implementação, que é redimensionável de forma automática.

8.4.1 CLASSE ArrayList<E>

A classe `ArrayList<E>` corresponde a uma implementação de **listas** usando um *array* que é redimensionável, ou seja, pode crescer sempre que é necessário espaço para mais elementos, tal como pode diminuir à medida que elementos são removidos, naturalmente sem que o utilizador do *arraylist* tenha que se preocupar com tal aspecto (Figura 8.4).

Tal como um *array* normal, um `ArrayList<E>` é uma estrutura indexada, e, portanto, os seus elementos ocupam uma posição dada por um número inteiro a partir de 0.

A primeira coisa que há que decidir antes de se criar um *arraylist* usando os construtores disponíveis, é o tipo `E` dos seus elementos, sendo certo porém que nenhuma colecção de JAVA armazena valores de tipos primitivos. Assim, o tipo `E` será um tipo qualquer válido excepto um tipo primitivo, e, portanto, será o nome de uma classe, de uma classe abstracta ou de uma interface (Figura 8.4).

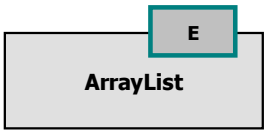


Figura 8.4 – ArrayList com tipo parâmetro E

Como primeiro exemplo simples, para melhor conhecermos as operações disponíveis para se manipularem objectos do tipo `ArrayList<E>`, vamos considerar que pretendemos de momento criar **listas de nomes**, sem grandes requisitos, mas com as usuais operações de procura, iteração, contagem, etc. Então, para este caso, `E = String`. Vamos, pois, criar objectos do tipo `ArrayList<String>`. A respectiva declaração de tipo, feita usando esta sintaxe, indica ao compilador que apenas objectos do tipo `E` poderão ser inseridos no *arraylist*. Um erro de compilação deverá ser produzido caso haja alguma instrução que tente inserir um objecto não `String`.

Vamos agora criar várias listas de nomes, conforme o tipo de nomes. As declarações para criarmos os nossos *arraylist* de *strings* são então as seguintes:

```
ArrayList<String> amigos = new ArrayList<String>();
ArrayList<String> musicos = new ArrayList<String>();
ArrayList<String> amigas = new ArrayList<String>(50);
```

Nos primeiros casos não nos preocupamos com a capacidade inicial (o mais usual), e, no último, indicamos uma capacidade inicial mínima de 50 elementos. Veremos adiante o terceiro construtor. Em todos os casos, podemos de imediato pretender calcular qual o número de elementos actual dos *arraylists*. O método **`size()`**, que é aplicável a todas as colecções, devolve tal valor inteiro, por exemplo, escrevendo:

```
int tam1 = amigos.size(); int tam2 = amigas.size();
```

o que, em ambos os casos, devolveria o valor 0 (pois capacidade ≠ nº de elementos).

Vamos agora inserir alguns nomes nas listas: o método mais simples para se realizar uma inserção numa lista é o método **`add(E elem)`** que junta o objecto parâmetro ao final da lista receptora e incrementa o número de elementos:

```
amigos.add(new Integer(127)); // para teste
amigos.add("Jorge"); amigos.add("Ivo"); ……
amigas.add("Vera"); amigas.add("Rute"); ……
```

Conforme se esperava, qualquer tentativa de inserir no *arraylist* de `String` um objecto não `String`, gera um erro de compilação que, como se pode verificar na Figura 8.5, é assinalado como método não encontrado por ter assinatura errada (ex.: `Integer`).

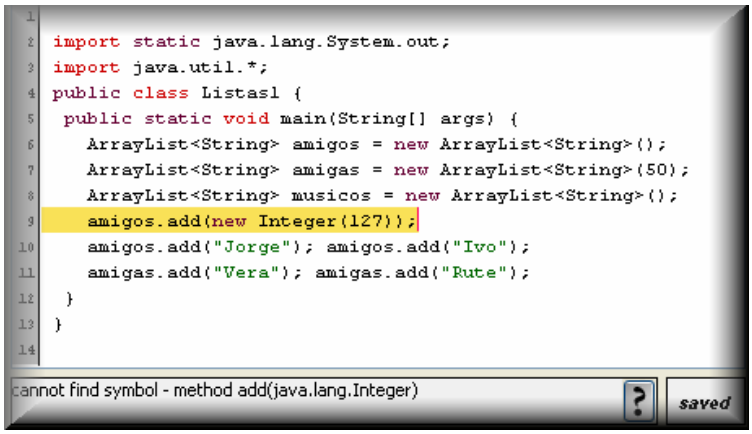


Figura 8.5 – Erro de tipos

Inseridos nomes de amigos e de amigas, vamos agora consultar um dos *arraylists*. Sendo um *arraylist* indexado, o método de consulta deve receber por parâmetro um índice actualmente válido (entre 0 e size()-1, para size() > 0) e devolver o elemento em tal posição do *arraylist*. O método que faz esta operação é o método **`E get(int index)`**:

```
String nome1 = amigos.get(0);
String nome2 = amigas.get(2);
```

Note-se que temos a certeza de que o tipo do objecto que nos é devolvido pelo método `get()` é sempre uma `String`, porque a verificação de tipos foi feita pelo compilador.

Em tempo de execução, temos, porém, que ter cuidado com o que nós próprios fazemos. Qualquer tentativa de aceder a um índice superior ao do último elemento conduz a uma excepção/erro de execução designada `IndexOutOfBoundsException` (Figura 8.6).

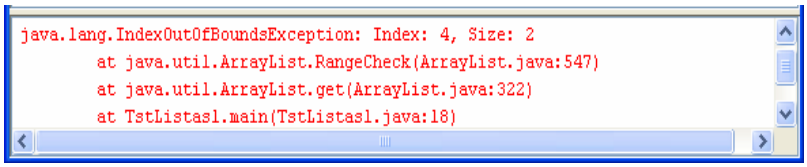


Figura 8.6 – Excepção/Erro na utilização de get()

O método **`set(int index, E elem)`** permite inserir o elemento parâmetro no índice `index` do *arraylist*. Se tal posição já for ocupada por um objecto, ele será perdido. Se tal índice for superior a size()-1 acontecerá a mesma situação de excepção de `get()`.

Estes erros, relativamente à utilização de `get()` e `set()` levantam o seguinte problema: que devemos fazer para podermos trabalhar com um `ArrayList<E>` que seja esparso, isto é, no qual os elementos não estão posicionados de forma contígua desde o índice 0 até à última posição ocupada, mas dispersos por várias posições?

Em tais situações, o que deve ser feito é inicializar um número razoável de posições a `null`, passando essa a ser a dimensão actual do *arraylist*. Haverá apenas que ter em atenção que o resultado de um `get()` pode ser `null`, o que não é um erro mas sim um valor não manipulável:

```
ArrayList<String> pals = new ArrayList<String>(20);
for(int i = 0; i < 20; i++) pals.add(null);
pals.set(15, "JAVA5"); pals.set(1, "Lista");
```

Com o código anterior, declarámos uma variável `pals` que é um `ArrayList<String>` com capacidade inicial para 20 *strings*. Com o ciclo `for`, inicializámos as 20 posições com o valor `null`. No final desta inicialização, o valor de `pals.size()` é 20. Agora, já podemos trabalhar livremente nos índices entre 0 e 19 deste *arraylist*, tal como se mostra nas duas últimas instruções.

Todas as colecções têm implementados os métodos **`clear()`**, que apaga os elementos do receptor, e **`isEmpty()`** que testa se a colecção receptora está vazia.

Para a realização de operações de pesquisa num *arraylist*, estão disponíveis os métodos **`contains(Object o)`**, **`indexOf(Object o)`** e **`lastIndexOf(Object o)`**. O primeiro apenas determina se um objecto dado como parâmetro existe ou não. O segundo devolve o índice da sua primeira ocorrência, caso ele exista, ou -1 no caso contrário. O terceiro devolve o índice da sua última ocorrência, caso ele exista, ou -1, no caso contrário.

O método **`remove(Object o)`** realiza também uma pesquisa no *arraylist* visando eliminar a primeira ocorrência do objecto parâmetro na lista, caso este exista. O método **`remove(int index)`** remove o elemento no índice dado, e **`removeRange(int from, int to)`** remove todos os elementos desde o índice `from` até ao índice `to-1`.

Cada remoção de um objecto de um *arraylist* implica o automático deslocamento de todos os elementos de índice superior para o índice imediatamente inferior, mantendo a estrutura compacta (operação designada por *shift up*).

O método **`subList(int from, int to)`** dá como resultado uma sublista contendo os elementos desde o índice `from` até ao índice `to-1` da lista receptora. Esta sublista não é uma cópia, apenas uma "janela" ou "visão" (através de partilha) sobre a original.

```
ArrayList<String> lstr = lista.subList(0, lista.size()/2);
```

Um exemplo típico de utilização é apagar uma porção da lista original, como em

```
lista.subList(i,f).clear(); // apaga de i a f-1
lista.subList(0, lista.indexOf(elem)).clear();
```

8.4.2 ITERADORES DE COLECÇÕES

Um dos tipos de operações mais utilizadas sobre colecções de objectos, é o que em geral se designa por operações de **iteração** ou **varrimento**. Estas operações deverão permitir ao utilizador de uma dada colecção percorrer todos os elementos desta, em geral sem uma ordem particular predefinida, para com tais elementos realizar um certo processamento, que se poderá reflectir ou não no estado interno da própria colecção.

No caso de um *arraylist*, ou de uma lista qualquer, salvo problemas de ineficiência que abordaremos mais tarde, qualquer lista poderá ser percorrida pelos seus índices, pelo que temos sempre uma forma "clássica" de aceder a todos os seus elementos usando um ciclo **`for`** tradicional usando os índices do *arraylist* (tal como o ciclo **`for`** sobre *arrays*).

```
// Iteração usando os índices
for(int i = 0; i < amigos.size(); i++) {
   out.println("Amigo : " + amigos.get(i));
}
```

Porém, nem todas as colecções são indexadas, mas é muito importante poder realizar sobre todas elas iterações. Em JAVA todas as colecções vão ser iteradas de forma semelhante, o que vai normalizar muito a linguagem e permitir ao programador abstrair-se muitas vezes do tipo de colecção com que está a lidar num dado momento. Vejamos como, usando como exemplo *arraylists*, mas chamando desde já a atenção de que tudo será muito semelhante para as outras colecções, mesmo não sendo listas.

ITERATOR<E>

A interface **`Iterable<E>`** é superinterface de `Collection<E>`, pelo que qualquer classe que implementar a interface `Collection<E>` tem que dar uma implementação para o método `iterator()`, método que cria um objecto **`Iterator<E>`** (um iterador automático) sobre a colecção de elementos de tipo `E`.

Um **`Iterator<E>`** é uma instância de uma estrutura computacional que implementa a respectiva interface e trabalha sobre os elementos de uma colecção usando três métodos:

- `hasNext()`: Método que determina se a colecção já foi esgotada;
- `next()`: Método que devolve o elemento seguinte da iteração;
- `remove()`: Método que elimina da colecção o último elemento iterado.

Uma das formas de trabalharmos com um **iterador** sobre uma colecção será seguindo o esquema segundo o qual em primeiro lugar criamos o iterador sobre a colecção, como:

```
Iterador<String> it = amigos.iterator();
```

e, em seguida, criamos um ciclo `while` com uma condição que testa a existência de mais elementos para serem iterados usando `hasNext()` (a colecção até pode estar vazia e não haver nenhuma iteração a fazer), e onde em cada iteração consumimos, usando o método `next()`, o elemento seguinte dado pelo iterador.

Teríamos, então, a seguinte estrutura de iteração completa:

```
// Iteração com Iterator<E> e while(it.hasNext())
Iterador<String> it = amigos.iterator();
while(it.hasNext()) {
   out.println("Amigo : " + it.next());
}
```

De notar que `it.next()` devolve garantidamente um elemento de tipo `E`, porque é este o tipo declarado dos elementos do *arraylist* e de `Iterator<E>`, no exemplo `String`.

A forma seguinte usa igualmente o método `iterator()` para criar um iterador sobre os elementos da colecção, mas torna tudo mais compacto ao usar um ciclo **`for`** tendo como condição de continuação a mesma do ciclo `while`, i.e. que `it.hasNext()` dê `true`, ou seja, que haja mais elementos a iterar. Teríamos, então, a forma seguinte:

```
// usando Iterator e for
for(Iterator<String> it = amigos.iterator(); it.hasNext();) {
     out.println("Amigo : " + it.next());
}
```

De notar que neste ciclo **`for` com iterador,** o iterador é criado no bloco de inicialização, e a expressão de incremento é inexistente já que o avanço na iteração é automaticamente realizado ao consumir-se o elemento seguinte da lista através do método `next()`. De notar igualmente que quando o iterador é criado sobre uma lista, o "próximo" elemento a ser acedido usando o método `next()` é o primeiro elemento da lista.

ITERADOR for(each)

O novo iterador ***foreach*** de JAVA5 torna as iterações sobre colecções ainda mais compactas, pois "esconde" a criação do `Iterator<E>`, o teste de fim de iteração, e o consumo do próximo elemento. A sua forma genérica é muito simples:

```
for(Tipo elem : col_iterável<Tipo>) instruções
```

lendo-se "com cada elemento `elem` de tipo `Tipo` obtido da **colecção iterável**, fazer..", pelo que, no nosso caso, se escreve simplesmente:

```
// Iteração com foreach
for(String nome : amigos)
   out.printf("Amigo ..%s%n", nome);
```

que é indiscutivelmente uma forma muito compacta de expressar a iteração.

Já havíamos usado anteriormente este novo iterador de JAVA5 aquando da apresentação dos *arrays*, aos quais também é aplicável pois estes são iteráveis.

Embora se escreva apenas **`for`**, vamos passar a designar este iterador por ***foreach*** de forma a distingui-lo da estrutura repetitiva `for`.

Vejamos mais alguns exemplos de utilização do iterador *foreach*, que será muitas vezes por nós aplicado nas mais diversas operações de varrimento usando colecções:

```
ArrayList<Ponto2D> pts = new ArrayList<Ponto2D>();
```

```
pts.add(new Ponto2D(0,0)); . . .
int maxY = Integer.MIN_VALUE;
for(Ponto2D pt : pts)
   if (p.getY() > maxY) maxY = p.getY();
out.printf("Maior Y = %3d%n", maxY);
```

Quando associada à iteração, temos uma ou mais condições particulares de terminação do ciclo, por exemplo, num algoritmo de procura, então, as iterações usando **`for`** e ***foreach*** devem ser abandonadas, devendo usar-se ciclos **`while`** pois tornam mais claros os objectivos do código. Vejamos um exemplo em que se pretende percorrer a lista de nomes até encontrar o primeiro nome de comprimento superior a cinco caracteres. Neste caso, como pode não existir nenhum, no limite teremos que iterar a lista toda, mas, logo que um nome com tais características é encontrado, queremos abandonar o ciclo. Assim sendo, devemos utilizar um `while`, conforme o seguinte código:

```
boolean encontrado = false;
String nome = "";
Iterador<String> it = amigos.iterator(); // iterador criado
while(it.hasNext() && !encontrado) {     // iteração
   nome = it.next();
   if (nome.length() > 5) encontrado = true;
}
out.println("Nome > 5 letras : " + nome);
```

Com os mesmos resultados mas de forma menos evidente teríamos, usando um `for`:

```
boolean encontrado = false;
String nome = "";
for(Iterator<String> it = amigos.iterator();
     it.hasNext() && !encontrado;) {
   nome = it.next();
   if (nome.length() > 5) encontrado = true;
}
out.println("Nome > 5 letras : " + nome);
```

LISTITERATOR<E>

As listas, e só estas, têm um método **`listIterator()`** (ou **`listIterator(int dim)`**) que devolve um iterador especial **`ListIterator<E>`** que, para além dos três métodos já referidos de um iterador usual, acrescenta métodos que possibilitam: navegar na lista em sentido inverso da ordem dos seus índices, como **`hasPrevious()`** e **`previous()`**; saber o índice do próximo elemento a ser iterado quer num sentido quer no outro, tal como os métodos **`nextIndex()`** e **`previousIndex()`**; fazer **`remove()`**, **`set(E e)`** e **`add(E e)`** em qualquer momento da iteração.

Este iterador é criado sobre uma dada lista e possui um cursor que está sempre posicionado antes do elemento da lista que será devolvido usando `next()`, e depois do elemento que será devolvido usando `previous()`. O método `listIterator()` coloca este cursor no início da lista, e o método `listIterator(int dim)` coloca o cursor depois do elemento de índice `dim-1` (o último da lista se `dim >= lista.size()`). Os métodos `remove()` e `set()` não funcionam com base na posição do cursor, mas operam sobre o último elemento iterado. O método `set(E e)` altera o seu valor para o valor dado, e o método `remove()` faz a sua remoção.

Um exemplo típico de utilização de um `ListIterator<E>` sobre listas, é o tradicional algoritmo de inversão de uma lista, que pode ser escrito de forma eficaz usando dois iteradores sobre listas. Vejamos como, num método que recebe uma lista de *strings*:

```
public ArrayList<String> inverte(ArrayList<String> lst) {
   ArrayList<String> aux = new ArrayList<String>(lst);
   ListIterator<String> normal = aux.listIterator();
   ListIterator<String> inv = aux.listIterator(aux.size());
   // 1° iterador no início; 2° iterador no fim
   int conta = 0; int meio = lista.size()/2;
   while(normal.next() && conta < meio ) {  // percorre até meio
      String temp = normal.next(); conta++;  // troca elementos
```

```
      normal.set(inv.previous()); // muda elemento dado em next()
      inv.set(temp);              // muda elem. dado em previous()
    }
    return aux;
}
```

Dada uma lista de *strings* como parâmetro, foi criada uma sua cópia de nome `aux`. Foram criados dois iteradores sobre `aux`. O segundo, `inv`, como foi criado usando o método `listIterator(aux.size())`, posiciona o cursor no fim. Encontrado o meio da lista, um iterador vai percorrê-la a partir do início enquanto o outro a percorre a partir do fim. O último elemento é trocado com o primeiro, o penúltimo com o segundo, etc. As trocas são feitas até ao meio, pois estamos a trabalhar com a mesma lista, apenas a percorrendo em sentidos diferentes. No final, a lista estará invertida e será dada como resultado do método.

Em síntese, temos, relativamente às listas, inúmeras maneiras de realizar a sua iteração, sendo no entanto de salientar o novo (de JAVA5) ciclo *foreach*, pois é aplicável a todas as colecções e também aos *arrays*, como havíamos já visto, e é muito intuitivo e de sintaxe simples, pois abstrai detalhes, de facto, desnecessários.

Alguns métodos de `ArrayList<E>` usam intrinsecamente estes iteradores como forma de implementarem certas operações que trabalham com quantidades massivas de dados (designadas *bulk operations*), tais como:

- **`addAll(`*`colecção`*`)`**: Junta ao fim da lista receptora, os elementos da colecção parâmetro, pela ordem dada pelo iterador desta; As colecções devem ser compatíveis no tipo dos elementos;
- **`addAll(int index, `*`colecção`*`)`**: O mesmo que o método anterior, tendo a inserção início no índice indicado;
- **`removeAll(`*`colecção`*`)`**: Remove do *arraylist* receptor os elementos existentes na colecção parâmetro;
- **`retainAll(`*`colecção`*`)`**: Remove do receptor os elementos que não pertencem à colecção parâmetro;
- **`containsAll(`*`colecção`*`)`**: Método que verifica se todos os elementos da colecção parâmetro são elementos da lista receptora, devolvendo verdadeiro ou falso.

Para além destes métodos de instância que recebem colecções como parâmetro, existe também um construtor que permite criar um *arraylist* a partir de uma colecção parâmetro. Por exemplo, tendo já as listas `amigos` e `amigas`, vamos usar tal construtor para criar uma nova lista contendo todos os nomes:

```
ArrayList<String> todos = new ArrayList<String>(amigos);
todos.addAll(amigas);  // junta a lista das amigas
for(String nome : todos) out.printf("Nome : %s%n", nome);
```

Para além destes métodos, qualquer colecção possui implementações para os métodos `clone()` (sempre cópias partilhadas, *shallow copies*), `toString()` e `equals()` que, como podemos compreender, não são muito usados nem fazem muito sentido, pois dependem sempre da existência dos métodos com o mesmo nome no tipo `E`, o que não é verificado por razões de eficiência.

Finalmente, quanto ao tipo parâmetro, não podendo ser um tipo primitivo, pode ser ele próprio uma colecção parametrizada. Assim, por exemplo, não há qualquer impedimento em que se defina um `ArrayList<`**`ArrayList<String>`**`>`, sendo certo que, a partir de tal declaração, o nosso tipo `E` passa a ser **`ArrayList<Sring>`**.

8.5 IMPLEMENTAÇÃO DE COLECÇÕES

A compreensão das várias operações sobre colecções passa por se ter uma ideia concreta sobre a forma como internamente estes objectos estão representados, dando-nos, deste modo, uma percepção correcta do funcionamento de certos métodos.

A maior parte das colecções são apenas estruturas de apontadores, cada um deles referenciando um dado elemento da colecção, tal como se mostra na Figura 8.7, onde se representa um `ArrayList<Ponto2D>` de nome `lst1` com três elementos.

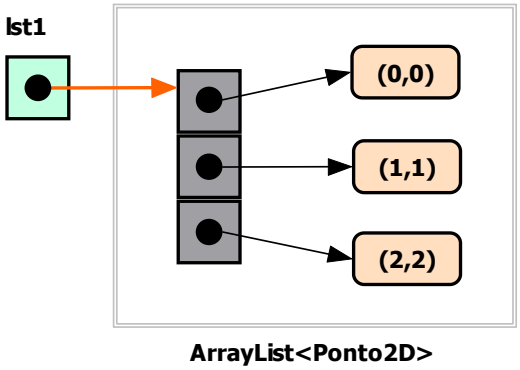


Figura 8.7 – Representação interna de um *arraylist*

Partindo deste `ArrayList<Ponto2D>`, vamos visualizar os resultados das operações mais comuns sobre colecções, podendo o que se vai apresentar sobre *arraylists* ser generalizado às outras colecções. Por exemplo, quando atribuímos um *arraylist* a outro de forma directa, ex.: `lst2 = lst1`, a única operação que de facto é feita é uma atribuição de endereços, ficando a variável `lst2` a conter o endereço contido em `lst1` (Figura 8.8).

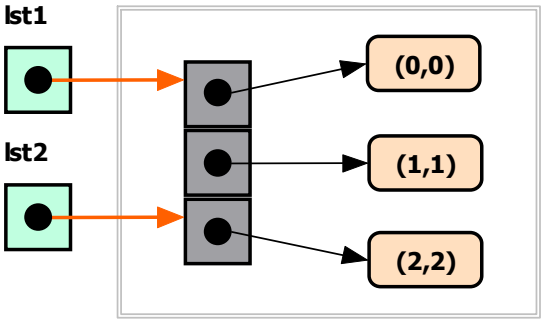


Figura 8.8 – Resultado de `lst2 = lst1`

A partir desta igualdade de endereços, a estrutura fica verdadeiramente partilhada pelas duas referências `lst1` e `lst2`. Qualquer operação de adicão, remoção ou de modificação de valor de elementos da colecção realizada usando a variável `lst1` reflecte-se na colecção, que, quando é acedida através da "vista" dada por `lst2`, já foi modificada, porque se trata da mesma e única colecção.

Se considerarmos que `lst1` é uma variável de instância de um qualquer objecto de uma classe nossa, podemos agora ter a ideia do que se passa quando a partilhamos com uma variável externa. Do exterior do nosso objecto alguém pode modificar o estado da colecção que internamente representamos por `lst1`, à qual não controlamos mais o acesso.

Assim, se escrevermos `lst2.add(new Ponto2D(9,9));`, passaremos a ter a colecção apresentada na Figura 8.9.

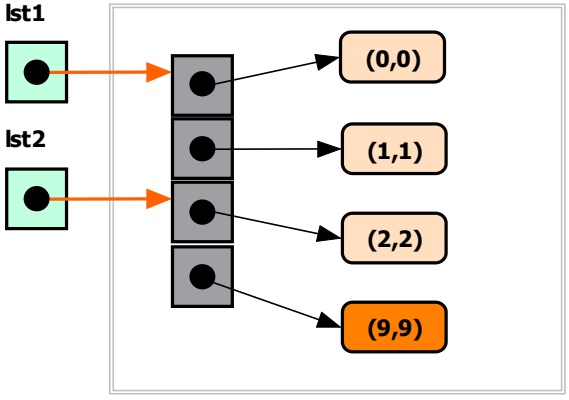


Figura 8.9 – Resultado de `lst1.add(new Ponto2D(9,9))`

Se, em seguida, através de `lst1` listássemos os elementos da colecção, escrevendo:

```
for(Ponto2D p : lst1) out.println(p);
```

seriam listados os quatro actuais pontos.

As operações de modificação da colecção são distintas das operações de modificação do estado dos elementos da colecção: as primeiras fazem-se através de métodos de *arraylist*, enquanto que as segundas se fazem através de métodos de `Ponto2D`. Assim, a instrução `lst1.set(1, new Ponto2D(10,10))` produz na estrutura o resultado que se apresenta na Figura 8.10.

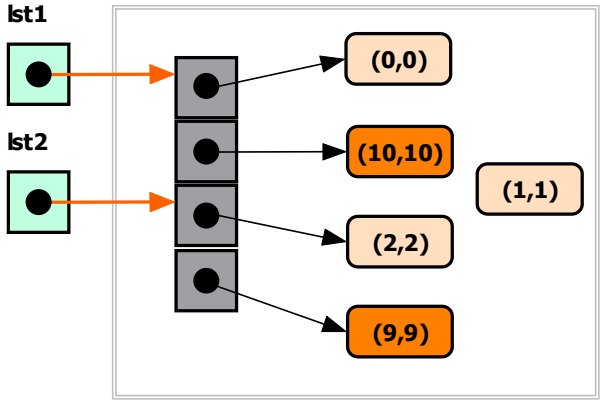


Figura 8.10 – Resultado de `lst1.set(1, new Ponto2D(10,10))`

O ponto (1,1) fica não referenciado e, assim, pronto a ser recolhido pelo *garbage collector* automático. A instrução `lst1.get(1).desloca(9,9);` produz um resultado diferente, como se pode verificar na Figura 8.11.

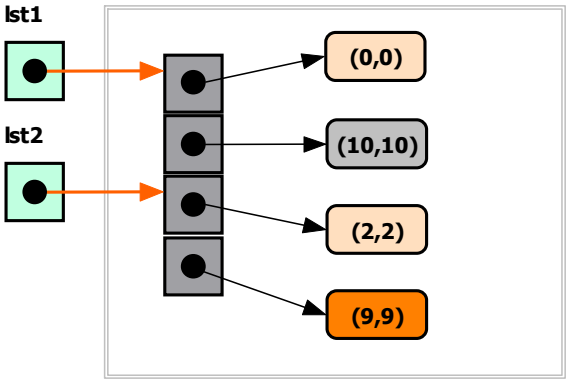


Figura 8.11 – Resultado de `lst1.get(1).desloca(9, 9)`

É de notar que, neste caso, nenhuma nova instância foi criada, apenas se modificou a instância existente, que foi deslocada de 9 unidades em `x` e em `y`.

Vamo agora analisar situações de **partilha parcial** de colecções que resultam de algumas das próprias operações das colecções. Consideremos os `ArrayList<Ponto2D>` que se apresentam na Figura 8.12, com os quais vamos realizar algumas operações.

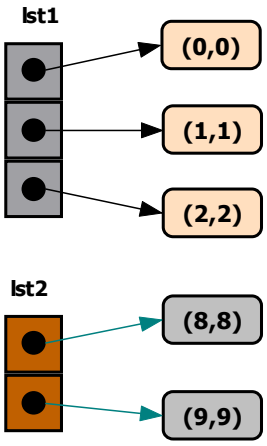


Figura 8.12 – `ArrayList<Ponto>`

Uma operação muito utilizada entre colecções, é a operação `addAll(c)` que junta à colecção receptora a colecção parâmetro, e que levanta sempre muitas dúvidas sobre se os elementos da colecção parâmetro são ou não copiados de forma *deep* para o receptor. Na Figura 8.13 apresenta-se o resultado efectivo de fazermos `lst1.addAll(lst2)` usando as listas apresentadas.

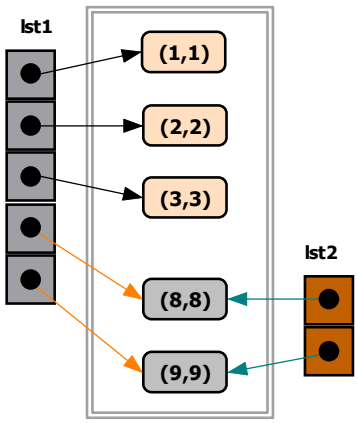


Figura 8.13 – Resultado de `lst1.addAll(lst2)`

Portanto, a operação realiza a inserção dos objectos existentes em `lst2` em `lst1`, objectos esses que passam a estar partilhados por ambas as listas, situação que é muito delicada e proibitiva até, se, por exemplo, a variável `lst1` for uma variável de instância de uma dada classe. Note-se que os objectos originais de `lst1` não são partilhados, mas os que foram agora adicionados são. Porém, quando trabalhamos com uma colecção tal é impossível de distinguir, pois não temos ao nosso dispor (a ideia não seria má) um *collection displayer* que nos apresente, após cada operação, estas configurações resultado.

Se fizermos `lst2.get(0).desloca(1,1); lst2.get(1).desloca(1,1);`, o resultado é o de se produzirem modificações em `lst1` por efeito lateral (Figura 8.14).

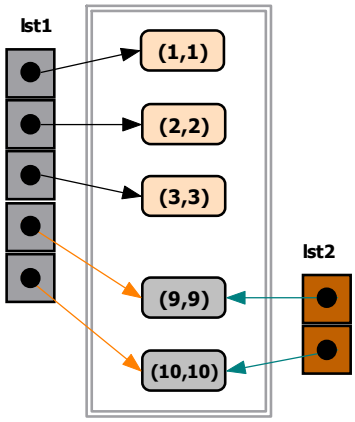


Figura 8.14 – Modificação por efeito lateral

A inserção de novos elementos e a remoção de elementos usando uma qualquer das variáveis não provocam problemas sobre a outra lista, porque cada estrutura possui os seus próprios apontadores. Por exemplo, `lst2.clear()` não afectaria `lst1`.

Assim, `lst1.remove(new Ponto2D(9,9));` e `lst2.add(new Ponto2D(5,5));` produziriam a configuração interna apresentada na Figura 8.15:

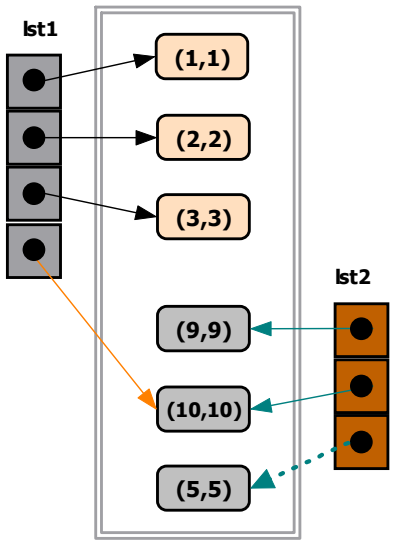


Figura 8.15 – Inserções e remoções com `add()` e `remove()`

O ciclo, `for(Ponto2D p : lst1) out.println(p);` dar-nos-ia como resultado os pontos `Pt(1,1)`, `Pt(2,2)`, `Pt(3,3)` e `Pt(10,10)`, usando `toString()` de `Ponto2D`.

Então, como garantir que, quando se adiciona uma colecção a outra, não ficam elementos partilhados entre elas? A resposta é clara agora: não usando `addAll()` mas copiando de forma *deep* cada um dos elementos da colecção parâmetro, realizando o `clone()` de cada elemento.

Para tal, bastaria escrever o código seguinte:

```
for(Ponto2D p : lst2) lst1.add(p.clone()); // clone de lst2
```

Como a Figura 8.16 mostra, usando esta operação as colecções são completamente independentes uma da outra, pelo que todos os problemas relacionados com alterações indevidas aos seus elementos deixam de existir.

É esta independência e protecção que pretendemos ter nas nossas variáveis de instância, garantindo que os seus valores apenas são modificados pelos métodos próprios.

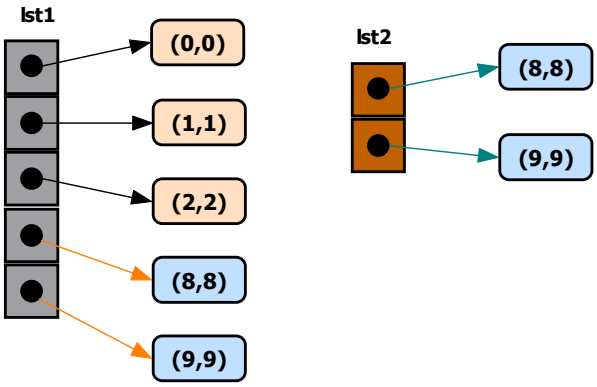


Figura 8.16 – Colecções independentes

Tudo o que acabámos de analisar usando como exemplo `ArrayList<Ponto2D>` é generalizável a todas as outras colecções, pois todas são, em termos da sua implementação concreta, *containers* de referências para objectos.

8.6 CLONE DE COLECÇÕES

Todas as colecções oferecem um método `clone()`. Este método é, como já vimos, um método de cópia *shallow*, pelo que o que o método faz é simplesmente copiar endereços. O resultado de `lst2 = lst1.clone()` será portanto (cf. Figura 8.17):

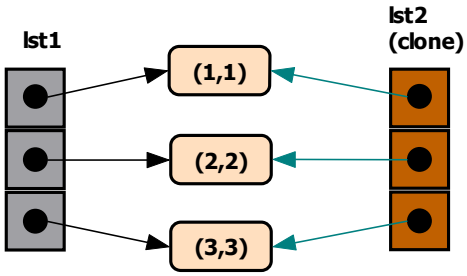


Figura 8.17 – Resultado de `lst2 = lst1.clone()`

Assim, quando numa classe nossa tivermos que devolver como resultado de um método uma variável de instância que seja uma colecção, não deveremos usar o método `clone()` dessa colecção, mas realizar a *deep copy* da colecção escrevendo código para tal, como, no caso de um *arraylist* do tipo `ArrayList<Ponto2D>`:

```
for(Ponto2D p : lst1) lstRes.add(p.clone());
return lstRes;  // devolve a cópia
```

8.7 EXEMPLO: LISTAS DE AMIGOS

Vamos agora escrever um pequeno programa no qual vamos realizar algumas operações com as listas de nomes apresentadas anteriormente:

```
import static java.lang.System.out;
import java.util.*;
public class ProgListas1 {
  public static void main(String[] args) {
    ArrayList<String> amigos = new ArrayList<String>();
    ArrayList<String> amigas = new ArrayList<String>(50);
    ArrayList<String> musicos = new ArrayList<String>();
     amigos.add("Jorge"); amigos.add("Ivo");
     amigos.add("Rino"); amigos.add("Armando");
     amigos.add("Pedro");
     amigas.add("Vera"); amigas.add("Rute");
```

```
amigas.add("Rita"); amigas.add("Zeta");
musicos.add("Rino"); musicos.add("Zeta");
out.println("--- AMIGOS ---");
for(String nome : amigos) out.printf(" %s", nome);
out.println("\n--- AMIGAS ---");
for(String nome : amigas) out.printf(" %s", nome);
out.println("\n--- MÚSICOS ---");
for(String nome : musicos) out.printf(" %s", nome);
out.println("\n--------------------");
// continuação do código a seguir
```

Após estas declarações, o estado actual das listas, com as quais vamos realizar as várias operações, é o apresentado na Figura 8.18.

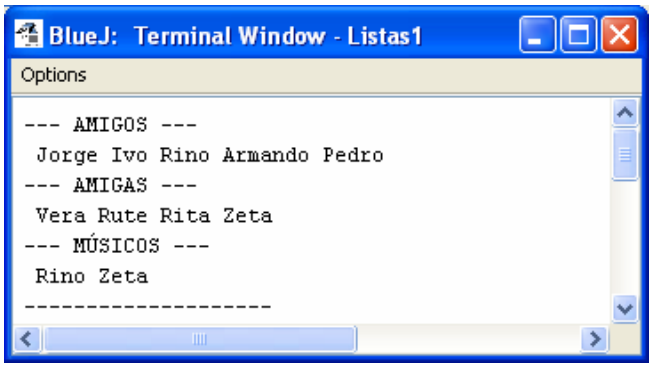


Figura 8.18 – Valores iniciais das listas

Comecemos pela procura de um nome, lido a partir do teclado, na lista de `amigos`. A operação será realizada até que o utilizador introduza uma *string* que, após ser convertida para maiúsculas, seja igual a "FIM".

```
// Procura de um nome lido na lista amigos
String nome = "";
out.print("Nome a procurar (ou FIM): ");
nome = input.next();
while(!nome.toUpperCase().equals("FIM")) {
  // iterador, while e pesquisa
  boolean vi = false;
  Iterator<String> it = amigos.iterator();
  while(it.hasNext() && !vi) {
     if(nome.equals(it.next())) vi = true;
  }
  out.println(nome + (vi ? " Existe." : " Nao existe."));
  out.print("Nome a procurar (ou FIM): ");
  nome = input.next();
}
```

Usámos um iterador e um *while* para realizar a procura (Figura 8.19), mas tal foi um mero exercício, pois neste caso bastar-nos-ia usar o método `indexOf()`.

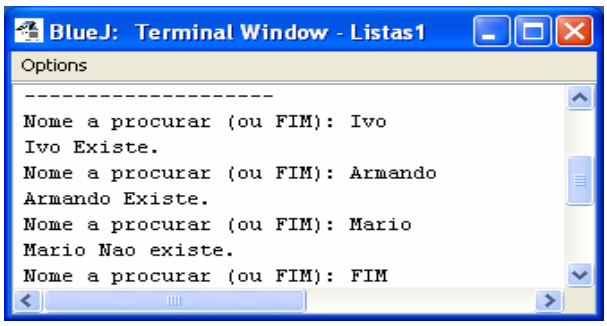


Figura 8.19 – Resultados da procura

Vamos agora criar a lista com os nomes dos amigos e das amigas que são músicos. Para tal vamos usar um `ArrayList<String>` auxiliar, para guardar os nomes comuns às listas de amigos, de amigas e de músicos:

```
ArrayList<String> temp = new ArrayList<String>(amigos);
temp.retainAll(musicos); // interseccao = amigos músicos
out.println("\n--- AMIGOS MÚSICOS ---");
for(String n : temp) out.printf(" %s");
temp = new ArrayList<String>(amigas);
temp.retainAll(musicos); // interseccao = amigas músicas
out.println("\n--- AMIGAS MÚSICAS ---");
for(String n : temp) out.printf(" %s");
```

A lista `temp` é declarada e de imediato inicializada com uma cópia dos elementos da lista `amigos`. Como estamos a trabalhar com *strings*, sabemos que estas são imutáveis pelo que, ainda que partilhadas, tal funciona como se fossem passadas por valor. Em seguida, usamos o método `retainAll(`*`colecção`*`)` para realizar a "intersecção" das duas listas. A lista auxiliar `temp` permite manter intocáveis as originais.

Para a lista de amigas copiámos usando `new ArrayList<E>(amigas)`, porque, se tivéssemos escrito `temp = amigas`, as listas seriam partilhadas, o que faria com que, ao escrever `temp.retainAll(devem)` a lista `amigas` ficasse, tal como `temp`, apenas com um único elemento, "Zeta" (tal como se apresenta na Figura 8.20).

Figura 8.20 – Intersecção de listas (comuns)

Vamos eliminar das listas de amigos e de amigas os músicos, usando `removeAll()` e observar as novas listas (Figura 8.21):

```
// Remover amigos e amigas que são músicos :(
amigas.removeAll(musicos); amigos.removeAll(musicos);
out.println("\n--- AMIGOS ---");
for(String n : amigos) out.printf(" %s", n);
out.println("\n--- AMIGAS ---");
for(String n : amigas) out.printf(" %s", n);
```

Figura 8.21 – Remoção de elementos

Este pequeno exemplo permitiu-nos já usar um razoável número de métodos da classe `ArrayList<E>`, designadamente, os construtores, o iterador `Iterator<E>` e o iterador `foreach`, os métodos de inserção `add()` e `addAll()`, de remoção `removeAll()`, de intersecção de listas `retainAll()`, bem como escrever alguns algoritmos de procura e de iteração sobre listas (`ArrayList<String>`, neste caso).

Note-se que `List<String> lista = new ArrayList<String>();` é uma atribuição correcta pois `List<E>` é o tipo de que `ArrayList<E>` é classe de implementação.

8.8 EXEMPLO: ZOO

Vamos em seguida criar uma classe que tenha uma variável de instância que seja do tipo `ArrayList<E>`, e escrever o código de métodos de instância que tenham como parâmetros `ArrayList<E>` e devolvam como resultados `ArrayList<E>`, por forma a estudarmos outros tipos de problemas relacionados com a utilização de colecções, e não apenas conhecermos os nomes dos métodos que lhe são aplicáveis.

Consideremos, pois, uma hierarquia de classes representativa da informação de teremos sobre mamíferos, hierarquia cuja superclasse é a classe abstracta `Mamifero` e que tem por subclasses concretas as classes `Cao`, `Gato` e `Coelho` (Figura 8.22), classes muito simples cujo código se apresenta a seguir:

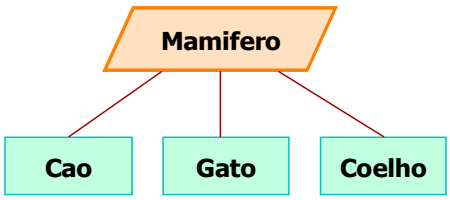


Figura 8.22 – Hierarquia de `Mamifero`

```
public abstract class Mamifero implements Serializable {
   // Construtores
   public Mamifero(String nom, String espec, int pesa) {
       nome = nom; especie = espec; peso = pesa; }
   // Variáveis de Instância
   private String nome;
   private String especie;
   private int peso;
   // Métodos de Instância
   public String daNome() { return nome; }
   public String daEspecie() { return especie; }
   public int daPeso() { return peso; }
   public void mudaPeso(int nvPeso) { peso = nvPeso; }
   public String toString() {
      return "Nome: " + nome + ...
   }
   // Métodos Abstractos
   public abstract Mamifero clone();
}
```

Apresentaremos apenas uma das classes concretas já que são todas muito semelhantes. Todas definem os seus construtores, redefinem o método `toString()`, por forma a que cada uma possa escrever o seu nome de classe, e definem o seu método `clone()`:

```
public class Gato extends Mamifero {
   public Gato(String nom, String espec, int pesa) {
      super(nom, espec, pesa);
   }
   // Redefinições
   public String toString() {
      StringBuilder s = new StringBuilder();
      s.append(super.toString() + "\n");
      s.append("Tipo" + this.getClass().getSimpleName());
   // Métodos de Instância não herdados
   public Gato clone() {
     return new Gato(this.daNome(), this.daEspecie(),
                     this.daPeso());
   }
```

```
}
```

Vamos agora definir uma classe designada **ZooMam** que deverá representar os animais existentes num dado zoológico, todos eles mamíferos e com as características indicadas.

A primeira ideia que poderia ocorrer para representar tal informação, já que acabámos de conhecer o tipo `ArrayList<E>`, seria definir o estado interno de tal classe à custa de três destes *arraylists*, como em:

```
ArrayList<Gato> gatos = new ArrayList<Gato>();
ArrayList<Cao> caes = ........
```

Claro que esta seria uma péssima solução sob todos os pontos de vista: em primeiro lugar porque não queremos colocar os mamíferos separados uns dos outros; e, em segundo lugar, porque, se mais tarde pretendêssemos que o zoo adquirisse elefantes ou baleias, a nossa representação teria que ser toda alterada e o código também.

Tal como já fizemos anteriormente com *arrays*, esta classe deverá conter uma variável de instância que será um *arraylist* capaz de guardar instâncias de qualquer das classes `Cao`, `Gato` e `Coelho`. A classe `Mamifero` é a superclasse abstracta destas classes, logo, o seu supertipo, pelo que a nossa variável de instância deverá ser, se possível, do tipo `ArrayList<Mamifero>` (com *arrays* escreveríamos também `Mamifero[]`).

Questão fundamental: de que tipo podem ser os objectos inseridos num `ArrayList<E>` usando o método `add(E elem)`? A resposta esperada e correcta é aquela que satisfaz o **princípio da substituição** e que mantém intactas todas as regras relativas a tipos estáticos e dinâmicos: **Num `ArrayList<E>` podem ser inseridos objectos do tipo `E` e de um qualquer subtipo de `E`.** Esta regra é aplicável a todas as colecções `Col<E>`.

Vamos assim definir na nossa classe **`ZooMam`** uma variável de instância **`zooList`**, que irá representar a lista dos mamíferos existentes, e que será declarada como:

```
ArrayList<Mamifero> zooList = new ArrayList<Mamifero>();
```

Nelal poderemos inserir, usando o método `add()`, instâncias de `Cao`, `Gato` e `Coelho`. Apenas não podemos inserir instâncias de `Mamifero` porque esta classe é abstracta e não cria sequer instâncias:

```
zooList.add(new Cao("XIMA", "BOX", 10));
zooList.add(new Gato("KIKA", "PER", 4));
zooList.add(new Coelho("C2", "SLV", 1));
```

A classe **`ZooMam`** vai implementar métodos para determinar o total de animais, inserir um novo mamífero, inserir uma lista de mamíferos, determinar a lista dos mamíferos com peso superior ao dado, a lista dos animais de dado tipo (ex.: `Cao`, `Gato`, etc.), pesquisar por nome, devolver a lista dos animais cujo nome se inicia por uma dada letra, número de animais de dado tipo e apresentação textual (`toString()`).

Dado que as classes e os programas tendem a ficar mais extensos, vamos apresentá-los e comentá-los simultaneamente, apresentando exemplos de execução sempre que tal puder ser um auxiliar adicional à apresentação do código.

```
import java.util.*;
import java.io.*;
public class ZooMam implements Serializable {
  // Variável de instância
  private ArrayList<Mamifero> zooList;
  // Construtores
  public ZooMam() { zooList = new ArrayList<Mamifero>(); }
  public ZooMam(ArrayList<Mamifero> animais) { . . . }
      ..............
```

Este segundo construtor deve receber um `ArrayList<Mamifero>` como parâmetro, e usá-lo para inicializar a variável `zooList` com uma cópia dos elementos deste. Tal pode ser feito de forma implícita, usando directamente o construtor de `ArrayList<E>`, ou de forma explícita codificando a cópia elemento a elemento. Vamos usar as duas formas.

A primeira solução usa o construtor de `ArrayList<E>` que permite realizar tal cópia:

```
zooList = new ArrayList<Mamifero>(animais);
```

Na segunda solução, realizamos nós tal cópia explicitamente, conforme o código:

```
// cópia elemento a elemento
zooList = new ArrayList<Mamifero>();
for(Mamifero mam : animais)
   zooList.add(mam);
```

Ambas as soluções usam métodos da classe `ArrayList<E>` para inserirem elementos no *arraylist* receptor, que especificam que os elementos dos parâmetros são copiados para o receptor, mas sem indicarem de forma precisa o tipo de cópia, se *shallow* ou *deep*, ou seja, se são copiadas as referências dos objectos ou os seus valores. Vamos testar.

A Figura 8.23 apresenta o contexto de tal teste. Foi criado um `ArrayList<Mamifero>` de nome `arrayLis2` e usando o método `add(E elem)` inseriram-se as instâncias de nome `gato2` e `cao2`. Criou-se em seguida o `arrayLis3` usando o construtor acima indicado, para o qual passamos como parâmetro `arrayLis2`. Portanto, a partir das duas instâncias de `Mamifero` criámos duas instâncias de `ArrayList<Mamifero>` usando duas formas diferentes de construção.

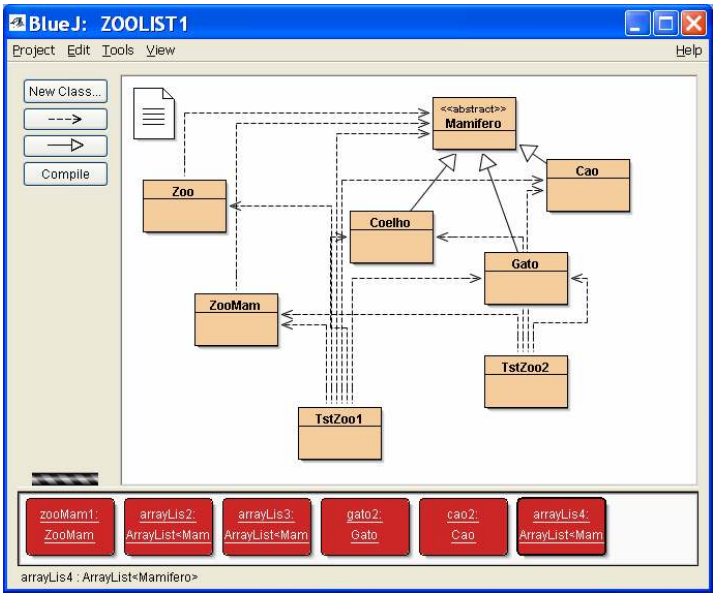


Figura 8.23 – Hierarquia de classes de `Zoo`

Em seguida, usando o método de `Mamifero mudaPeso()`, alterámos o peso de `gato2` e de `cao2`. Se estas instâncias não forem partilhadas por `arrayLis2` e `arrayLis3`, os seus valores nestes *arraylists* não terão sido afectados. Se forem partilhadas, o seu peso mudou para o novo valor.

Ao realizar a consulta do estado dos dois *arraylists*, verificou-se que os pesos dos objectos referidos **foram também modificados nos arraylists**. Assim, **são partilhados**. Podemos pois concluir que, quer o método `add(E elem)`, quer o construtor que cria um *arraylist* via cópia de um `ArrayList<E>` parâmetro, não realizam *deep copy* mas apenas partilha.

Este resultado é muito importante. Se não tivéssemos tido esta preocupação de testar a forma de funcionamento quer do método `add(E elem)`, quer do construtor por cópia, quer do método `addAll()`, pelas especificações ficaríamos com a ideia de que *copy* quer dizer *deep copy*, o que de facto não acontece, mas sim partilha de endereços.

Então, nenhuma das soluções anteriores é segura do ponto de vista do nosso código para a classe **`ZooMam`**, porque a nossa variável de instância `zooList` tem apenas referências para objectos exteriores. Não é isso que pretendemos. Assim, antes de atribuirmos tais objectos vindos do exterior via parâmetros, devemos fazer a sua **clonagem** usando os respectivos métodos `clone()`. Claro que não podemos pedir que `ArrayList<E>` possua um método `clone()` que faça isto para todos os possíveis `E`, por isso a cópia é *shallow*.

O método construtor correcto escrever-se-á então:

```
public ZooMam(ArrayList<Mamifero> animais) {
   zooList = new ArrayList<Mamifero>();
   for(Mamifero mam : animais) zooList.add(mam.clone());
}
```

e os métodos de instância devem escrever-se:

```
// Métodos de Instância
public void juntaMamif(Mamifero mam) {
   zooList.add(mam.clone());
}

// Junta uma arraylist de Mamifero ao zoo
public void juntaMamifs(ArrayList<Mamifero> mamifs) {
   for(Mamifero mam : mamifs) zooList.add(mam.clone());
}

// Lista com os nomes de todos os animais
public ArrayList<String> nomes() {
   ArrayList<String> nomes = new ArrayList<String>();
   for(Mamifero mamif : zooList)
      nomes.add(mamif.daNome());     // é String
   return nomes;
}

// Total de mamíferos
public int totalAnimais() { return zooList.size(); }
```

As *strings* são **objectos imutáveis** em JAVA pelo que não precisamos de fazer `clone()` dos nomes dos mamíferos ao "exportá-los" em *arraylist* para o exterior, pois não são modificáveis. Pelo contrário, o método seguinte, que "exporta" um *arraylist* com os dados de todos os animais, tem que fazer uma ***deep copy*** de `zooList`, daí o `clone()` de cada mamífero que é adicionado à lista resultado:

```
// Devolve uma lista com todos os animais
public ArrayList<Mamifero> getLista() {
   ArrayList<Mamifero> lstMam = new ArrayList<Mamifero>();
   for(Mamifero mam : zooList) lstMam.add(mam.clone());
   return lstMam;
}
```

Os métodos segintes correspondem às usuais operações de iteração para selecção por vários critérios. A iteração é sempre realizada usando um `foreach`, e apenas o critério de selecção (filtragem) varia. A adição do elemento seleccionado é sempre realizada usando `clone()`, pois este foi seleccionado a partir da variável de instância e nenhum iterador copia a colecção a iterar, apenas percorre os seus elementos:

```
// Devolve um ArrayList de Mamiferos do zoo com peso
// superior ao peso dado.
public ArrayList<Mamifero> pesoSupA(int peso) {
   ArrayList<Mamifero> mamRes = new ArrayList<Mamifero>();
   for(Mamifero mamif : zooList)
      if( mamif.daPeso() > peso ) mamRes.add(mamif.clone());
   return mamRes;
}

// Devolve um ArrayList de Mamiferos do zoo com nome
// iniciado pela letra dada como parâmetro.
public ArrayList<Mamifero> nomeIniciadoPor(char letra) {
   ArrayList<Mamifero> mamRes = new ArrayList<Mamifero>();
   for(Mamifero mamif: zooList)
      if( mamif.daNome().charAt(0) == letra ) mamRes.add(mamif.clone());
   return mamRes;
}
```

```
// Número de animais de dado tipo (CLASSE)
public int animaisPorTipo(String tipo) {
   int conta = 0;
   for(Mamifero mamif : zooList)
      if( (mamif.getClass().getSimpleName()).equals(tipo) )
         conta++;
   return conta;
}

// Lista dos animais de dado tipo
public ArrayList<Mamifero> lstMamTipo(String tipo) {
   ArrayList<Mamifero> lst = new ArrayList<Mamifero>();
   for(Mamifero mamif : zooList) {
      if(mamif.getClass().getSimpleName().equals(tipo))
         lst.add(mamif.clone());
   }
   return lst;
}

// Verifica se existe algum mamífero de nome dado
public boolean existeNome(String nome) {
   return (this.nomes().contains(nome));
}
// Devolve o 1º mamífero cujo nome é dado como parâmetro
public Mamifero animalDeNome(String nome) {
   Iterator<Mamifero> it = zooList.iterator();
   Mamifero mamif = null;
   boolean enc = false;
   while(it.hasNext() && !enc) {
      mamif = it.next();
      if( mamif.daNome().equals(nome) ) enc = true;
   }
   return mamif.clone();
}
```

Este último método realiza uma procura por nome sobre a `zooList`. Como o iterador *foreach* não permite condições de saída particulares e o ciclo `for` com iterador é pouco legível, devemos então usar um iterador e um ciclo `while`.

É importante notar que o iterador que deve ser criado é do tipo `Iterador<Mamifero>`, pois é obtido a partir de `zooList`, que é do tipo `ArrayList<Mamifero>`, ou seja, um *arraylist* de elementos do tipo `Mamifero`.

Assim, cada objecto obtido via `it.next()` a partir do iterador é, seguramente, do tipo `Mamifero`, o que facilita, do ponto de vista de tipos, a segurança da sua manipulação posterior, que é restrita a métodos definidos na classe `Mamifero`.

```
// Remove o primeiro mamífero de nome e tipo dados
public void remMamifero(String nome, String tipo) {
   Iterator<Mamifero> it = zooList.iterator();
   Mamifero mamif = null;
   boolean enc = false;
   while(it.hasNext() && !enc) {
      mamif = it.next();
      if( mamif.daNome().equals(nome) &&
          mamif.getClass().getSimpleName().equals(tipo) ) {
           enc = true; it.remove();  // remove em zoolist
      }
   }
}
```

```
  // toString()
  public String toString() { . . . }
} // fim da definição da classe ZooMam
```

Temos, assim, apresentado todo o código da classe **ZooMam**.

Vamos, em seguida, criar um pequeno programa para testar esta classe e os seus métodos, e, posteriormente, verificar em BlueJ os resultados da execução de alguns destes. O programa seguinte dá como resultado uma instância de `ZooMam`, para que seja de imediato usada no ambiente BlueJ:

```
import java.util.*;
public class TstZooMam {
  // Métodos auxiliares
  public static ArrayList<Cao> criaCaes() {
  // método estático para criar alguns cães
    ArrayList<Cao> caes = new ArrayList<Cao>();
    //
    caes.add(new Cao("BIS", "BASS", 10));
    caes.add(new Cao("LUKA", "RAF", 7));
    ........
    return caes;
  }
  //
  public static ArrayList<Gato> criaGatos() {
  // método estático para criar alguns gatos
    ArrayList<Gato> gatos = new ArrayList<Gato>();
    gatos.add(new Gato("PUF", "CHI", 2));
    ........
    return gatos;
  }
  //
  public static ArrayList<Coelho> criaCoelhos() {
      ........
  }
  //
  // Programa principal
    public static ZooMam main(String[] args) {
      // Variável do tipo ZooMam
      ZooMam zoo = new ZooMam();
      // arraylist auxiliar
      ArrayList<Mamifero> mamifs = new ArrayList<Mamifero>();
      // junta ArrayList<Cao>, <Gato>, <Coelho>
      mamifs.addAll(criaCaes()); // junta ArrayList<Cao>
      mamifs.addAll(criaGatos()); // junta ArrayList<Gato>
      mamifs.addAll(criaCoelhos()); // junta ArrayList<Coelhos>
      // junta 1 a 1
      zoo.juntaMamif(new Cao("TIL", "BOXER", 8));
      zoo.juntaMamif(new Gato("KUKA", "PER", 3));
      zoo.juntaMamif(new Coelho("ZAP", "CHI", 1));
      // junta ArrayList<Mamifero>
      zoo.juntaMamifs(mamifs); //
      //
      return zoo; // objecto devolvido (para uso em BlueJ)
    }
  }
```

A Figura 8.24 apresenta toda a estruturação das classes criadas, tal como são apresentadas no diagrama de classes do BlueJ. A figura apresenta também a instância criada em resultado da execução do programa `TstZooMam`, e os métodos disponíveis.

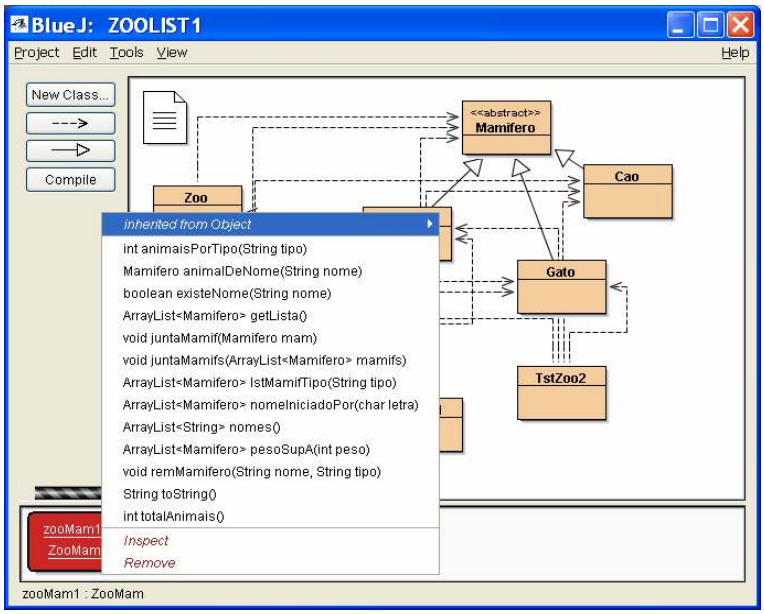


Figura 8.24 – Instância de `ZooMam` e métodos disponíveis

Temos, após execução do programa apresentado, um objecto `ZooMam` criado e todos os métodos definidos disponíveis (Conforme Figura 8.24).

Vamos, em seguida, executar alguns destes métodos, e observar os resultados obtidos usando o BlueJ, quer através da inspecção de alguns dos objectos resultantes da execução dos métodos dos quais fazemos **Get**, quer fazendo directamente **Inspect**.

Por exemplo, o método de procura de um dado animal por nome, quando tal nome existe, devolve um objecto de tipo `Mamifero` que corresponde à ficha do mamífero (Figura 8.25).

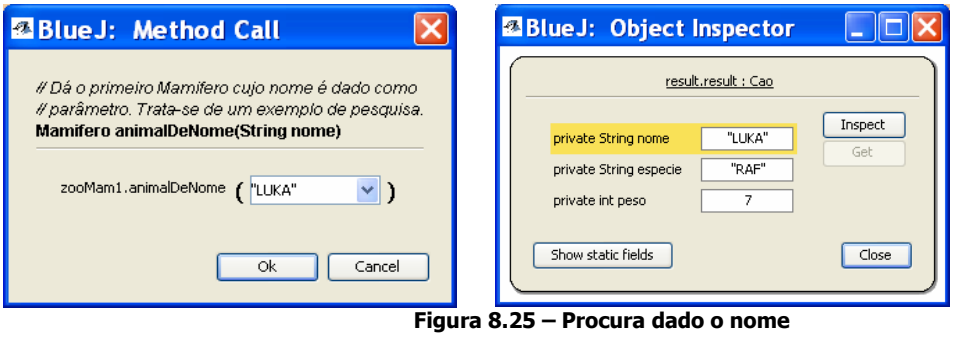
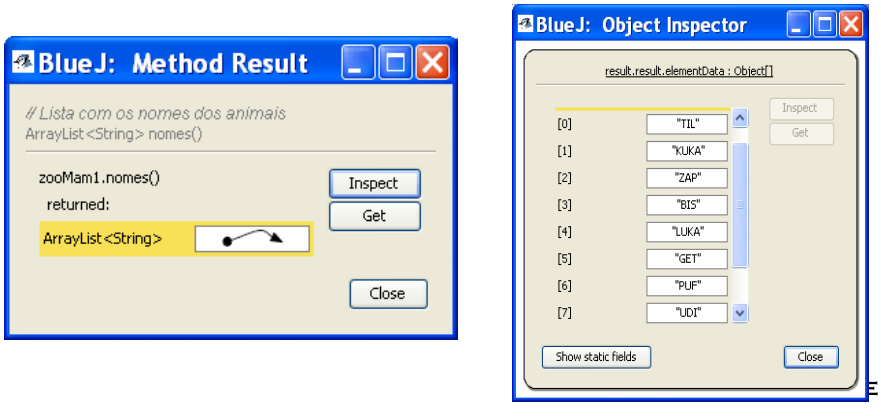


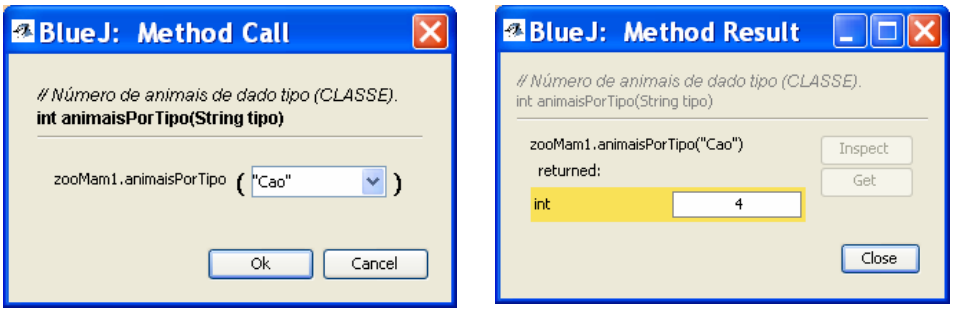
Figura 8.25 – Procura dado o nome

O método `nomes()` devolve uma lista de *strings* contendo os nomes de todos os animais registados (Figura 8.26).

Figura 8.26 – Lista de nomes



O método `animaisPorTipo(String tipo)` calcula o número total de animais do tipo dado, ou seja, da classe cujo nome corresponde à *string* indicada (Figura 8.27):



igura 8.27 – Total de animais de uma classe

A operação seguinte dá como resultado um `ArrayList<Mamifero>`, contendo todos os mamíferos do tipo dado como parâmetro (Figura 8.28).

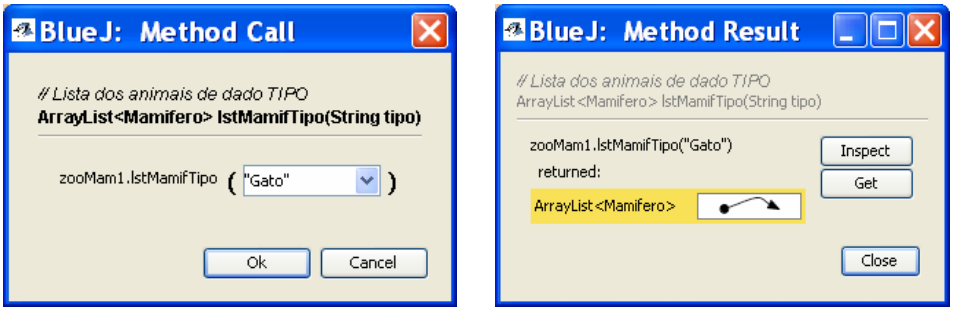


Figura 8.28 – `ArrayList<Mamifero>` resultado

Depois de obtido o resultado do método anterior, usámos a opção **Inspect** para podermos ter acesso ao conteúdo do *arraylist* e aceder aos dados de cada uma das instâncias de `Mamifero` (Figura 8.29).

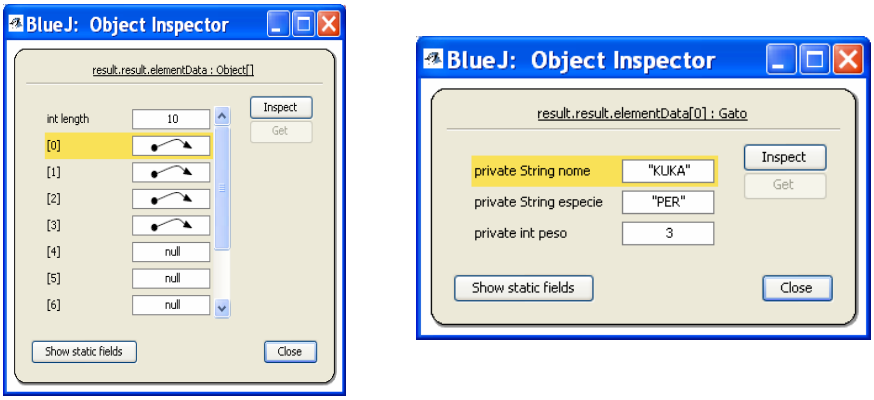


Figura 8.29 – Resultados 1

Apresentam-se os resultados de inspeccionarmos algumas das fichas de `Mamifero` que foram seleccionadas pelo método anterior (Figura 8.30).

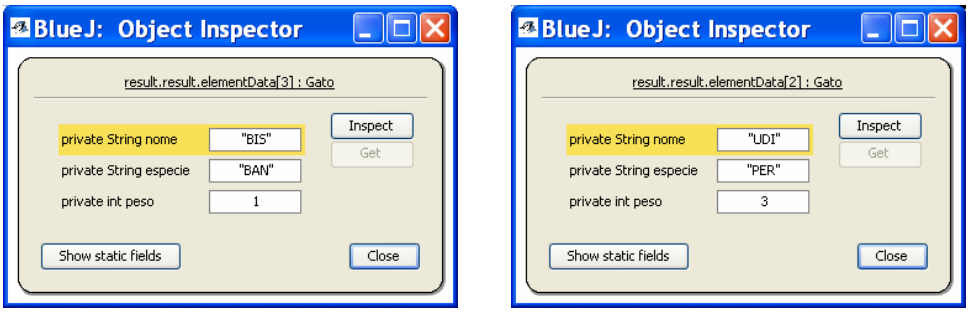


Figura 8.30 – Resultados 2

Vamos agora aumentar o número de animais no nosso zoológico. Para tal, começamos por criar um `ArrayList<Gato>` com mais alguns "gatos", e depois faremos o mesmo com os outros tipos de `Mamifero`. Vamos fazê-lo usando o programa **`TstZooMam`**, acrescentando mais algumas linhas, tal como se apresenta a seguir, sublinhando a negrito as novas linhas:

```
zoo.juntaMamifs(mamifs);
ArrayList<Gato> gatos = new ArrayList<Gato>();
// aqui com add(new Gato(...)) inserimos
// e agora vamos juntar os gatos ao zoo
zoo.juntaMamifs(gatos);
return zoo;
```

Recompilamos o nosso programa, e o resultado da compilação é um inesperado **erro de compilação**, que se apresenta na Figura 8.31.

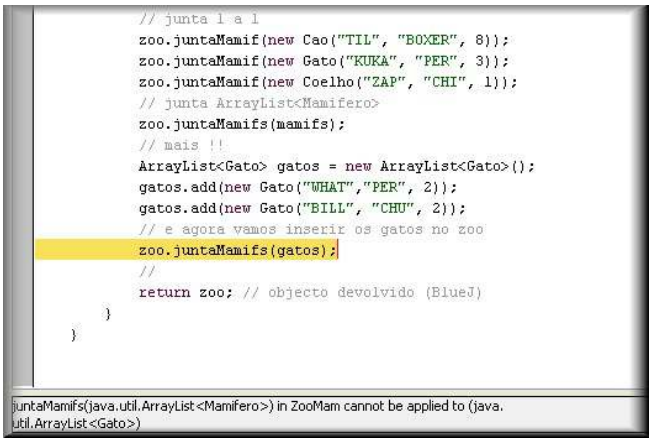


Figura 8.31 – `ArrayList<Mamifero>` e `ArrayList<Gato>` incompatíveis

À primeira vista, este erro de compilação não parece ter justificação. A mensagem de erro indica que o método `juntaMamifs()` tem um **`ArrayList<Mamifero>`** por parâmetro formal e portanto não aceita um parâmetro actual do tipo **`ArrayList<Gato>`**. Ou seja, o compilador está-nos a informar que `ArrayList<Gato>` e `ArrayList<Mamifero>` são tipos de listas incompatíveis, apesar de `Gato` ser subtipo de `Mamifero`.

O nosso método **`void juntaMamifs(ArrayList<Mamifero> mamifs)`** havia sido invocado anteriormente com um parâmetro que era um **`ArrayList<Mamifero>`**, no qual inserimos instâncias de subtipos de `Mamifero`. Porém, tal regra de subtipos não se aplica a `ArrayList<E>`, pelo que deste modo o `ArrayList<Mamifero>` é incompatível com o `ArrayList<Gato>`.

Intuitivamente, esperaríamos compatibilidade entre o tipo `ArrayList<Mamifero>` e o tipo `ArrayList<Gato>` dado que `Gato` é subtipo de `Mamifero`. Sabemos até que para os *arrays* esta compatibilidade é garantida, ou seja, se B é subtipo de A então é correcto que sendo `A[] arrayA` e `B[] arrayB` se escreva `arrayA = arrayB`.

Esta relação entre tipos designa-se por **covariância**, e *os **arrays*** **são covariantes**. *Arrays* de um supertipo são compatíveis com *arrays* de um subtipo.

Assim, um `Gato[]` é compatível com um `Mamifero[]` desde que `Gato` seja um subtipo de `Mamifero`, ou seja, **há covariância nos *arrays***, mas tal já não acontece com as instanciações dos tipos parametrizados. Este facto é muito importante porque, tal como se disse antes, parece contrariar a nossa intuição. Vejamos porquê, e como a falta de covariância entre instanciações dos tipos parametrizados foi resolvida em JAVA5.

8.9 *WILDCARDS:* PARTE I

Sendo esta incompatibilidade um facto, que, como veremos a seguir, tem todo o sentido, a pergunta que podemos colocar de imediato será a seguinte: se `ArrayList<Mamifero>` não é o supertipo de `ArrayList<Gato>`, `ArrayList<Cao>` e de `ArrayList<Coelho>` qual é esse supertipo? Ou seja, com que `ArrayList<>` são estes *arraylists* compatíveis?

A introdução dos **tipos genéricos** em JAVA5 teve por motivação principal tornar a linguagem mais segura em termos de tipos (*type safe* em inglês), em especial na fase de compilação, ou seja, na fase estática dos programas, na qual o compilador realiza a verificação da correcção das expressões quanto aos seus tipos (*static type checking*).

Antes da introdução dos tipos genéricos, por exemplo, em JAVA4, um *arraylist* era apenas `ArrayList` e não, como agora, um `ArrayList<E>` de objectos de um dado tipo `E`. No entanto, antes de JAVA5, era muito comum que, ao utilizar-se um `ArrayList`, todos os objectos inseridos fossem do mesmo tipo, por exemplo `String`, mas não havia forma de especificar tal propriedade desejada. Quando os objectos eram lidos do `ArrayList`, eram lidos como `Object` (tipo do resultado do `get()` de então), devendo ser o programador a codificar o *casting* para o

verdadeiro tipo do objecto, por exemplo, escrevendo `String nome = (String) nomes.get(i);`. Qualquer erro de tipo na inserção gerava um erro de execução ao fazer-se o *casting*.

O problema das colecções de objectos, em termos da sua hierarquia de tipos, ou seja, da forma como são compatíveis entre si, resulta do facto de que, para além de serem definidas de modo parametrizado, logo, terem parâmetros que são variáveis de tipo, tipos esses que também pertencem a uma hierarquia, toda a verificação de coerência tem que ser feita pelo compilador, portanto, em tempo de compilação.

Uma das operações neste sentido mais problemática com as colecções de objectos é a operação que permite **inserir** novos elementos na colecção (tipicamente `add()`), pois é esta que pode provocar a quebra de todas as regras de coerência da estrutura em termos do tipo de objectos que se espera que a mesma contenha.

Problemas deste tipo foram até JAVA5 deixados à preocupação do programador, não possuindo a linguagem um sistema forte de tipos para as colecções. Todas as colecções de JAVA eram programadas como sendo do tipo `Object`, ou seja, qualquer colecção poderia conter qualquer objecto de uma subclasse de `Object`, sendo heterogéneas (objectos de vários tipos possíveis).

Assim, por exemplo, quando se declarava um *arraylist*, não havia a necessidade de se indicar o tipo dos seus elementos. O tipo era, em geral, escrito em comentário, para que o programador não se esquecesse dele, tipo que acabava por ficar apenas "na sua cabeça" e não no código gerado.

Apenas a título de exemplo, se pretendêssemos criar um *arraylist* de *strings*, que, como vimos atrás, é agora muito seguro e muito simples, em versões anteriores de JAVA o código seria o seguinte:

```
ArrayList nomes = new ArrayList(); // de String
nomes.add("Rui"); nomes.add("Ana"); ....
```

Sempre que se pretendesse aceder a uma posição do *arraylist*, o método de acesso estava programado para devolver um `Object`, pelo que era obrigatório realizar *casting* para o tipo correcto do objecto que (supostamente) havíamos consumido da colecção. Por exemplo, uma consulta a uma qualquer posição teria o código:

```
nome1 = (String) nomes.get(i); // casting
```

Uma iteração teria a seguinte construção:

```
String nome = "";
for(Iterator it = nomes.iterator(); it.hasNext();) {
   nome = (String) it.next(); // casting
   // qualquer processamento com nome
}
```

Não havendo tipos definidos nas declarações (logo, estáticos), teremos que ter a preocupação de realizar sempre conversões do tipo genérico `Object` para o tipo efectivo, sendo pois deixada para o código do programa a verificação da correcção dos tipos.

Por exemplo, se tivéssemos escrito uma linha como:

```
nomes.add(new Ponto2D(2, 3)); // => ERRO DE EXECUÇÃO
```

sendo um ponto uma instância de uma subclasse de `Object`, o compilador teria aceite, mas, na iteração acima, seria gerado um erro de execução ao fazer o *casting* de `Ponto2D` para um objecto do tipo `String`.

JAVA5, com os tipos genéricos, veio tornar esta utilização muito segura *(type safe*), porque, sendo `ArrayList<E>` parametrizado por um tipo `E`, quando esse tipo é instanciado (ex.: `E = String`) apenas objectos desse tipo `E` (e subtipos) podem ser inseridos. Sendo essa correcção verificada pelo compilador, então não poderá haver erros de execução ao ler do *arraylist*, porque apenas objectos do tipo `E` foram inseridos, não sendo também necessário qualquer `casting`. Esta forma de utilização segura e fácil, já nós comprovámos nos exemplos anteriores, nos quais não usámos *casting* uma única vez.

O sistema forte de tipos de JAVA5 relativamente às colecções, que levou a reformular o JCF com a introdução dos tipos genérios (*generics*), veio praticamente eliminar a necessidade de *casting* e de preocupações com segurança de tipos. O objectivo geral do desenvolvimento dos tipos genéricos foi dado pelo *slogan* que apenas tem algum impacto em inglês "*Making JAVA easier to type and easier to type*". Porém, necessitamos agora de compreender as regras de tipos para colecções, tendo sempre como perspectiva o facto de que a maioria do trabalho de segurança de tipos é da responsabilidade do compilador.

Sendo o tipo lista genérico e devendo ser instanciado, há no entanto muitas situações em que pretendemos que uma lista, um *arraylist*, por exemplo, seja de facto apenas uma lista relativamente à qual não temos qualquer interesse no tipo dos seus elementos `E`. Por exemplo, métodos que apagam listas e métodos que calculam o seu comprimento não necessitam de saber o tipo dos elementos da lista. Se especificarmos um tipo, ou o tipo é absurdo, ou então o método já poderá não servir para outros tipos, quando poderia ser escrito para todas as listas, independentemente do tipo dos seus elementos, e ser assim **completamente genérico**.

Outro problema idêntico surge quando, em certas classes muito genéricas, necessitamos de variáveis de tipo *arraylist* (ou outras colecções) mas em que o tipo dos seus elementos não importa, como é o caso de classes como a `java.lang.Class<T>`.

A solução encontrada em JAVA5 para lidar com este tipo de situações em que se torna importante **abstrair completamente do tipo de elemento da estrutura**, pois o importante é a estrutura em si e o que se pretende realizar com ela, foi a introdução de um tipo especial de parâmetro para as colecções, representado por **`?`** e designado por ***wildcard***.

Este símbolo pode substituir um tipo concreto numa expressão de instanciação, significando que o tipo actual é indiferente, **qualquer** ou **desconhecido** (*unknown type*). Vejamos alguns exemplos da sua utilização:

```
public static void print(ArrayList<?> lista) { … }
public void juntaStack(Stack<?> stk) { … }
private ArrayList<?> list;
```

Tal como exemplificado, a utilização de tipos é sempre feita em dois contextos fundamentais: declaração de variáveis e declaração de métodos. Relativamente às variáveis, temos sempre que pensar que operações poderemos realizar com elas se forem de dado tipo, em especial inserções e leituras. Quanto aos métodos, se um parâmetro de entrada for de dado tipo parametrizado, que tipos actuais são compatíveis com esse, ou seja, quando invocamos o método, com que argumentos o podemos fazer. O mesmo se aplica aos resultados devolvidos pelo método quando se trata de um interrogador.

O tipo **`ArrayList<?>`** (leia-se **"arraylist de um tipo qualquer"** ou **"arraylist de um tipo desconhecido"**), **é o supertipo** de `ArrayList<E>`, o que significa que um *arraylist* de um qualquer tipo é compatível com o tipo `ArrayList<?>`.

Um método como `print(ArrayList<?> list)` aceita como parâmetros actuais quer `ArrayList<String>`, quer `ArrayList<Ponto>`, etc., portanto, sem limitações.

O tipo de parâmetro **?** designa-se por tal motivo um ***unbounded wildcard*** (ilimitado) e o tipo um **tipo parametrizado ilimitado**.

Temos assim **`ArrayList<?>`** como supertipo de qualquer *arraylist*, tal como se mostra na hierarquia da Figura 8.32.

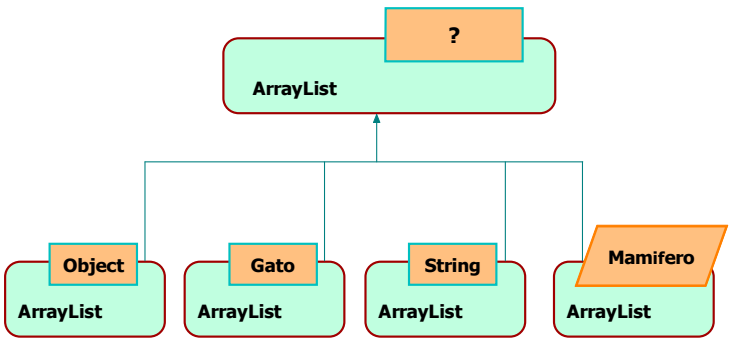


Figura 8.32 – Compatibilidade com `ArrayList<?>`

Esta hierarquia reflecte a relação entre tipos parametrizados que são *arraylists*, mas que é aplicável a qualquer outro tipo de colecções, excepto `Map<K, V>`.

Enquanto **tipo de uma variável**, o tipo parametrizado ilimitado `ArrayList<?>` não é muito útil pois quase não pode ser usado. Como o compilador não sabe o tipo de lista que a variável contém num dado momento, já que é até possível que tal tipo possa mudar ao longo do programa, por exemplo, como em:

```
ArrayList<?> lista = new ArrayList<String>();
. . . ; lista = listaGatos;
```

então, não é possível usar o método `add()`, ou seja, inserir elementos numa lista deste tipo, excepto **`null`**. No entanto, podem ser lidos elementos da lista usando `get()` ou iterando. Porém, como a lista pode ter elementos de um tipo resultante de uma atribuição qualquer, o compilador generaliza para `Object`, porque, não sabendo de que tipo são os elementos da lista, sabe que, de certeza, todos são compatíveis com `Object`.

Finalmente, a construção `new ArrayList<?>` (tal como com outra qualquer colecção) não é permitida directamente por razões óbvias (criar o quê?):

```
ArrayList<?> col1 = new ArrayList<?>(); // ERRO COMP.
Set<?> nms = new HashSet<?>();          // ERRO COMP.
```

Mesmo com estas limitações, podemos ainda escrever métodos do tipo:

```
public static void print(ArrayList<?> lista) {
   for(Object o : lista) out.println(o);
```

Um tipo parametrizado com *wildcard* **`?`** tem muitas semelhanças com uma interface, já que pode ser usado em declarações de parâmetros e variáveis, mas não é permitido que se criem objectos desse tipo.

O *wildcard* **`?`** não impõe ao tipo parâmetro concreto qualquer restrição, o que, em certos casos, faz bastante sentido como vimos. Noutros casos, no entanto, pretendemos expressar exactamente o facto contrário, ou seja, que apenas se aceitam tipos argumento que satisfaçam certas propriedades.

No nosso exemplo, pretendemos expressar que queremos aceitar *arraylists* não de um tipo qualquer, mas do tipo `Mamifero` ou de qualquer subtipo de `Mamifero`. Assim, no nosso caso, o *wildcard* **`?`** é demasiado lato, demasiado permissivo, pelo que precisamos de o restringir.

Esta restrição pode ser descrita através de outro *wildcard*, designado por ***upper bounded wildcard*** (*wildcard* com limite superior), e que se escreve **`? extends E`**. Este *wildcard* restringe os tipos argumento ao tipo `E` e a um qualquer subtipo de `E`.

O tipo parametrizado `ArrayList<? extends E>` deve ler-se **"arraylist de um tipo desconhecido** (ou qualquer) **que é um subtipo de `E`"**. Assim, o *wildcard* define um limite superior para o tipo argumento do tipo parametrizado, tal como se mostra na Figura 8.33. São exemplos válidos de utilização deste tipo de *wildcard* com limite, os seguintes:

```
ArrayList<? extends Object> l1 = new ArrayList<String>();
ArrayList<? extends Mamifero> lm = listaGatos;
```

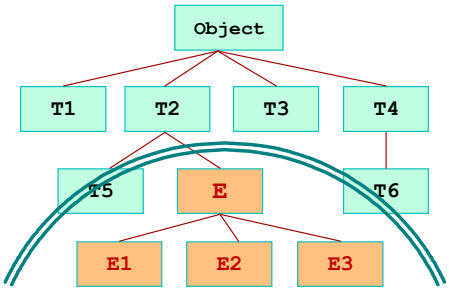


Figura 8.33 – Tipos que satisfazem `? extends E`

Claro que um `ArrayList<? extends E>` é **supertipo** de `ArrayList<E>` e, assim, é perfeitamente possível atribuir um `ArrayList<Gato>` ou um `ArrayList<Cao>` a um `ArrayList<? extends Mamifero>`. Um

`ArrayList<Mamifero>` e os *arraylists* dos seus subtipos não são sequer relacionáveis entre si, embora sejam todos `ArrayList<E>` ou seja, diferentes instanciações de `ArrayList<E>`.

Por outro lado, sendo **`? extends E`** uma restrição de **`?`**, o tipo `ArrayList<?>` é, naturalmente, supertipo de `ArrayList<? extends E>`.

Todas estas propriedades dos *wildcards* que estamos a analisar para o caso concreto dos *arraylists* são aplicáveis directamente aos `Set<E>`, bem como aos tipos individuais `K` e `V` dos `Map<K,V>`, o que verificaremos quando os estudarmos.

A Figura 8.34 ilustra a hierarquia de *arraylists* com *wildcards*, que representa também as compatibilidades relativas entre estes (**`E1 <: E`** significa que **`E1` é subtipo de `E`**):

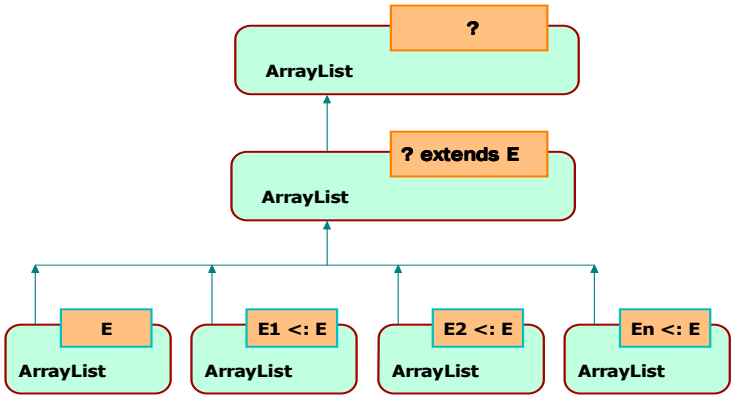


Figura 8.34 – Hierarquia de `<? extends E>`

Há, no entanto, restrições importantes a considerar quanto à utilização de tipos parametrizados como `ArrayList<? extends E>`. Mais uma vez, e pelas mesmas razões, não podem ser usados com `new`. Os seus elementos podem ser lidos da colecção mas não é possível adicionar (ex.: `add()`) elementos a este tipo de colecções. A razão é simples, e é a mesma que vimos antes para o caso do *unbounded wildcard* **`?`**, que aliás se mantém, ainda que limitado: o compilador desconhece o tipo, pelo que, naturalmente, a operação não seria segura.

Esta simples restrição irá certamente excluir das nossas mentes declarações inadequadas, tais como associar variáveis de instância a tipos com ***wildcards***, como, por exemplo:

```
zooList = ArrayList<? extends Mamifero>; // sem sentido
```

Tal tornaria a variável de instância *imutável*, pois não mais poderíamos adicionar-lhe quaisquer objectos, excepto atribuindo-lhe uma lista compatível (`zooList = lst;`).

Antes de concluirmos, por agora, os ***bounded wildcards***, vejamos, de um ponto de vista pragmático, a razão pela qual um `ArrayList<Mamifero>` não poderia, nem deveria, ser compatível com qualquer um dos *arraylists* de subtipos de `Mamifero`.

Vamos usar uma metáfora, não com animais mas com fruta. Imaginemos que num dado supermercado existem sacas próprias para cada tipo de fruta, cf. sacas para maças, sacas para figos, para bananas, etc., e que há uma saca especial onde se podem colocar frutas de um tipo qualquer, a "saca da fruta". Um dia esgotaram-se as sacas das laranjas e nós, que apenas queríamos comprar laranjas, pegámos numa "saca da fruta", metemos lá dentro umas laranjas e fomos para a caixa. Qual o nosso espanto quando a menina da caixa, em vez de apenas pesar a fruta (laranjas), abriu o saco para ver o que estava dentro. Quando lhe perguntámos porque estava a fazer aquilo, ela respondeu que, sendo aquela uma "saca da fruta", podia ter laranjas, bananas e outras frutas, e as frutas têm preços diferentes.

Se o compilador de JAVA5 não funcionasse por antecipação, ou seja, não deixando substituir uma saca de laranjas por uma de fruta, tudo seria muito perigoso em tempo de execução. Se se obtivesse uma referência do tipo `ArrayList<Fruta>` para referenciar uma instância de `ArrayList<Laranja>` (note-se que é o equivalente a termos uma referência do tipo `ArrayList<Mamifero>` à qual atribuíamos um `ArrayList<Gato>`), para além da

boa intenção de inserirmos o que devemos, a verdade é que também poderíamos inserir outros frutos por engano. Porém, eventualmente referenciada através de uma outra variável qualquer, a instância é, de facto, do tipo `ArrayList<Laranja>`, só que agora contém outros frutos que não laranjas. Vejamos o exemplo seguinte:

```
1. ArrayList<Fruta> lstF= new ArrayList<Fruta>();
2. ArrayList<Laranja> lstL = new ArrayList<Laranja>();
3. lstF = lstL; // partilha de refs entre lstF e lstL
4. lstF.add(new Laranja(20));
5. lstF.add(new Banana(33));
6. for(Laranja larj : lstL) out.println(larj); // BUM !!
```

Olhando para o código acima, apenas teremos que pensar onde, se fôssemos nós a tomar a decisão, gostaríamos mais que o erro fosse detectado: se na linha 3, isto é, na fase de compilação, indicando tipos incompatíveis e tudo estaria terminado, ou, então, deixando passar a atribuição da linha 3 sem erro de compilação, estando pois todo o programa correcto, mas podendo, um certo dia a dada hora, ao executar, dar um **erro de execução** na linha 6, porque um objecto "não-laranja" (isto é, não do tipo `Laranja` nem de um tipo compatível) foi extraído do *arraylist* onde, supostamente, apenas instâncias de `Laranja` haviam sido inseridas. Deixaríamos neste último caso de ter **segurança de tipos**.

Em JAVA5, como vimos, o compilador dará erro de compilação na linha 3, precavendo de imediato situações perigosas e incontroláveis como a que se mostra neste código. Ao realizar **verificação forte de tipos** em tempo de compilação, o compilador garante que não haverá "surpresas" de erros de tipos em tempo de execução.

Ainda que tal possa não ser tão intuitivo, esta regra garante o **princípio da substituição** para as colecções. Esta regra torna também imperiosa a existência de um tipo especial que compatibilize colecções de elementos de tipos que são hierarquicamente relacionados entre si, e este é o *wildcard* limitado **`? extends E`**. É este *wildcard* que reintroduz e garante a **covariância** das colecções.

Finalmente, é importante não confundir o **tipo desconhecido** `?` com o tipo `Object`, pois este é um tipo "bem conhecido", porque, para além do mais, é o topo da hierarquia de tipos, ou seja, o **supertipo de todos os tipos**, parametrizados ou não, colecções ou não.

Por exemplo, um `ArrayList<Object>` e, de forma geral, uma `Collection<Object>`, representa uma colecção **heterogénea** de elementos que podem ser de qualquer tipo referenciado (via `Object`). Uma `Collection<?>` é uma colecção de elementos que possuem um supertipo comum desconhecido. Se esse supertipo for uma classe `final`, a colecção é **homogénea**, ou seja, todos os elementos são do mesmo tipo. Adicionalmente, `Object` é um tipo parâmetro concreto, enquanto que `?` não.

Temos assim, retomando o nosso exemplo, que, ainda que `Mamifero` seja supertipo de `Cao`, `Gato`, etc., um `ArrayList<Mamifero>` pode conter objectos que são instâncias de subtipos de `Mamifero`, mas não é supertipo nem compatível com nenhum *arraylist* de subtipos de `Mamifero`. A hierarquia de `ArrayList<E>` não tem a mesma estruturação que a hierarquia de `E`, tal como se ilustra na Figura 8.35.

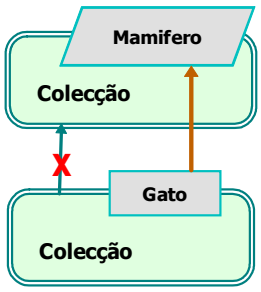


Figura 8.35 – Hierarquia de colecções vs. hierarquia dos seus tipos parâmetro

Assim, o supertipo de todos os *arraylists* de `Mamifero` e dos seus subtipos é o *arraylist* que se declara como **`ArrayList<? extends Mamifero>`** (Figura 8.36).

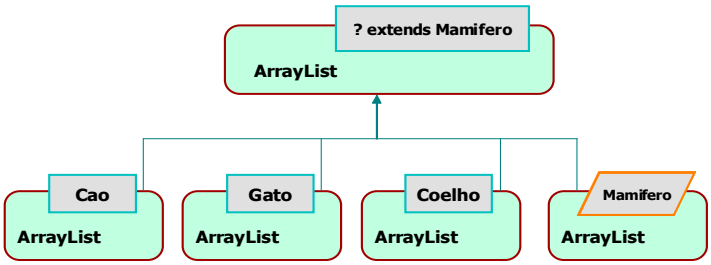


Figura 8.36 – Hierarquia de `ArrayList<? extends Mamifero>`

Agora que sabemos determinar o supertipo de um *arraylist*, podemos modificar o código do nosso método da classe **`ZooMam`**, para que passe a aceitar qualquer *arraylist* de animais da hierarquia cuja superclasse é `Mamifero`.

O código passaria então a ser o seguinte:

```
public void juntaMamifs(ArrayList<? extends Mamifero> mamifs) {
   for(Mamifero mam : mamifs) zooList.add(mam.clone());
}
```

tal como mudaria para o construtor que aceita um *arraylist*, que ficará mais genérico:

```
public ZooMamGen(ArrayList<? extends Mamifero> animais) {
   zooList = new ArrayList<Mamifero>();
   for(Mamifero mam : animais) zooList.add(mam.clone());
}
```

Seriam estas as únicas modificações que faríamos na classe original para que ficasse uma classe mais genérica, no sentido de aceitar vários tipos de *arraylist* parâmetro, que fariam todo o sentido em função do problema.

A classe é também compacta e segura, pois tivemos a preocupação de realizar `clone()` sempre que copiávamos objectos para a variável de instância da classe, e sempre que tivemos que devolver como resultado de métodos parte do estado interno da classe.

Curiosamente, parte da resposta à questão inicial estava no próprio código do programa **`TstZooMam`** que nos deu o erro de compilação. Naturalmente não reparámos, mas, umas linhas antes da instrução que gerou o erro de compilação, havíamos inserido um *arraylist* do tipo `ArrayList<Gato>` num `ArrayList<Mamifero>` de nome **`mamifs`**, que foi usado como variável auxiliar, escrevendo:

```
ArrayList<Mamifero> mamifs = new ArrayList<Mamifero>();
mamifs.addAll(criaCaes());  // junta ArrayList<Cao>
mamifs.addAll(criaGatos()); // junta ArrayList<Gato>
```

e tal instrução não gerou qualquer erro. É que o método `addAll()`, de `ArrayList<E>`, é um método que foi codificado tendo por parâmetro um `ArrayList<? extends E>`, logo, usando o supertipo de `ArrayList<E>`. Conhecido o significado dos *wildcards*, podemos agora apresentar as API das colecções (no Quadro 8.2 a de `ArrayList<E>`).

ArrayList<E>
`ArrayList()`
`ArrayList(Collection<? extends E> c )`
`boolean add(E o)`
`void add(int index, E o)`
`boolean addAll(Collection<? extends E> c)`
`boolean contains(Object o)`
`E get(int index)`
`int indexOf(Object elem)`
`int lastIndexOf(Object elem)`
`remove(int index)`
`boolean remove(Object o)`

`boolean removeAll(Collection<? extends E> c)`
`boolean retainAll(Collection<? extends E> c)`
`E set(int index, E element)`
`Iterator<E> iterator();` `ListIterator<E> listIterator();`
`clear(), clone(), isEmpty(), size()`

Quadro 8.2 – API de `ArrayList<E>`

8.10 AUTO-BOXING E AUTO-UNBOXING EM LISTAS

Suponhamos que, para uma dada aplicação, pretendemos armazenar números inteiros num *arraylist*. Como os valores de tipos primitivos não são objectos, e apenas os objectos podem ser guardados nas colecções, então, temos que usar a classe *wrapper* para valores do tipo `int`, a classe `Integer`, para converter valores inteiros em instâncias da classe `Integer`, e inseri-las num `ArrayList<Integer>`. Teríamos o seguinte código:

```
ArrayList<Integer> lstInt = new ArrayList<Integer>();
. . .
int i = lerInt();
Integer intg = new Integer(i);
lstInt.add(intg);
```

Este processo de conversão de valores de um tipo primitivo para uma instância da sua correspondente classe *wrapper* (de "embrulho") designa-se por ***boxing***. De notar que é verdadeira a expressão `i == intg.intValue()`.

Quando se pretende ler valores inteiros do *arraylist*, então, temos que realizar o processo inverso, ou seja, ler uma instância de `Integer`, e calcular o seu valor inteiro usando o método `intValue()`. Este processo designa-se por ***unboxing***.

JAVA5 introduziu um mecanismo automático de *boxing* e *unboxing* que, no fundo, faz "nos bastidores" as operações que foram atrás realizadas, poupando ao programador umas linhas de código, o que em todo o caso é sempre agradável. O mecanismo designa-se pois por ***Auto-Boxing/Unboxing*** e aplica-se a todas as colecções e *arrays*, sendo extensivo aos iteradores *foreach*.

O código anterior passaria a escrever-se simplesmente,

```
ArrayList<Integer> lstInt = new ArrayList<Integer>();
int i = 111;
lstInt.add(i); // auto-boxing
// etc…
int x = lstInt.get(0); // auto-unboxing
lstInt.add(222);       // auto-boxing
lstInt.add(333);       // auto-boxing
int soma = 0;
for(int ni : lstInt) soma += ni; // auto-unboxing
out.println("Soma = " + soma);
```

O mecanismo cria a ilusão de que as colecções aceitam tipos primitivos, o que pode ser em certos casos enganoso se tivermos em atenção que vamos poder escrever frases do tipo:

```
out.println(lstInt.get(1) + 124);
lstInt.add(lstInt.get(0)%2);
```

Introduzido o mecanismo, usá-lo-emos sempre que necessário como se fosse uma característica nativa da linguagem e, portanto, sem mais qualquer referência particular.

8.11 MAPS DE K PARA V

Nesta secção, vamos estudar as colecções que implementam as correspondências finitas entre **chaves** e **valores**, que são absolutamente fundamentais em Informática, já que toda a informação principal guardada em sistemas de dados tem a si associada uma **chave única** que é também, em geral, o seu identificador e veículo de acesso.

Os nossos números de bilhete de identidade, de contribuinte, de carta de condução, de aluno, de eleitor, as chapas de matrícula dos veículos, etc., são chaves únicas que estão associadas univocamente a grupos de dados particulares.

Um *mapping* é a representação de uma função matemática que estabelece uma qualquer correspondência entre um **conjunto de chaves** (keys) e um **conjunto de valores** (values), tal como representado na Figura 8.37.

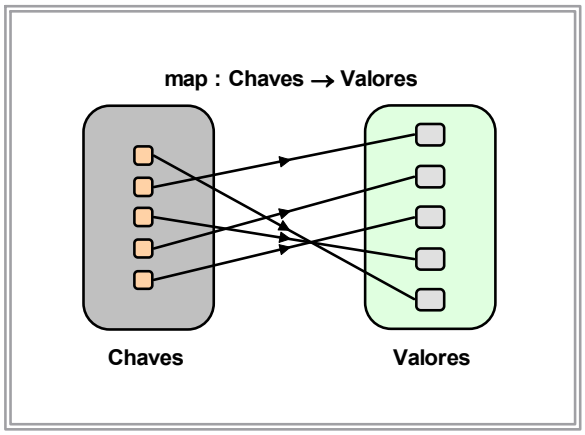


Figura 8.37 – Visualização de um *map*

A interface **`Map<K,V>`** representa a funcionalidade comum aos *maps*. A variável de tipo `K` representa o tipo das *keys* (chaves) e `V` o tipo dos *values* (valores).

Qualquer *map* tem as seguintes propriedades:

- São correspondência entre uma chave e um valor (1 para 1);
- As chaves são únicas e formam um conjunto; os valores podem ser em duplicado, isto é, duas chaves distintas podem ter o mesmo valor (ex.: duas cidades distintas com a mesma população) mas o valor não é partilhado, pelo que são duas correspondências (designadas pares) distintas;
- As remoções de um *map* removem sempre pares chave-valor e não apenas as chaves;
- Não há qualquer ordem definida sobre chaves ou pares.

Existem actualmente quatro implementações gerais de `Map<K,V>`, conforme se pode verificar pela Figura 8.37, designadamente:

- `HashMap<K,V>` (baseada em tabela de *hash*);
- `LinkedHashMap<K,V>` (tabela de *hash* e lista ligada);
- `TreeMap<K,V>` (árvore balanceada para ordenação das chaves);
- `EnumMap<K,V>` (representação bit-based para tipos enumerados)
- `Hashtable<K,V>` (baseada em tabela de *hash*, para concorrência).

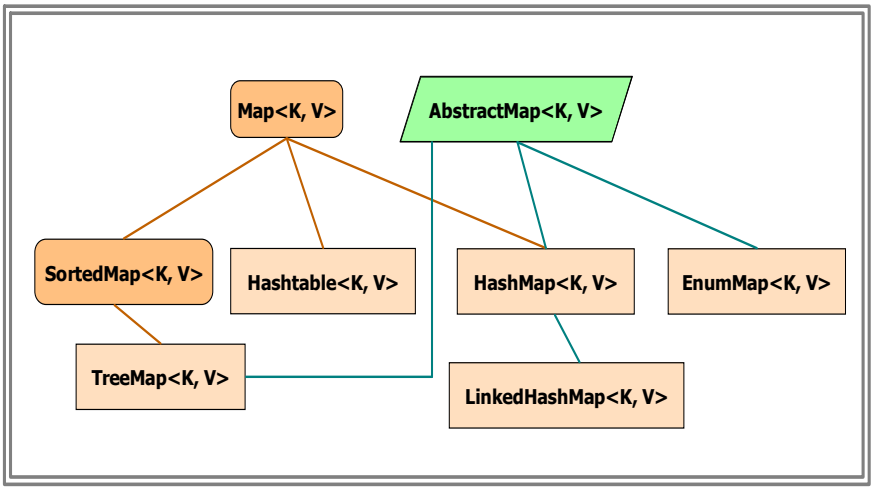


Figura 8.38 – Implementações de `Map<E>`

Apresentada a API de `Map<K,V>` no Quadro 8.3, vamos estudar em seguida a classe parametrizada `HashMap<K,V>`, a mais utilizada para as aplicações que não implicam processos, e abordaremos depois as outras.

Categoria de Métodos	API de Map<K,V>
Inserção de elementos	`put(K k, V v);` `putAll(Map<? extends K, ? extends V> m);`
Remoção de elementos	`remove(Object k);`

Consulta e comparação de conteúdos	`V get(Object k);`
	`boolean containsKey(Object k);`
	`boolean isEmpty();`
	`boolean containsValue(Object v); int size();`

Criação de Iteradores	`Set<K> keySet();`
	`Collection<V> values();`
	`Set<Map.Entry<K, V>> entrySet();`

Outros	`boolean equals(Object o); Object clone()`

Quadro 8.3 – API de `Map<K,V>`

8.12 HashMap<K,V>

A classe `HashMap<K,V>` corresponde a uma implementação de *maps* numa tabela de *hash*, com optimizações automáticas de espaço e carga, e algoritmos sofisticados para associar de forma eficaz chaves a valores.

Um `HashMap<K,V>` tem os construtores usuais para colecções, designadamente:

```
HashMap()
HashMap(Map<? extends K, ? extends V> m)
```

O segundo construtor recebe como parâmetro um *map* de qualquer tipo subtipo de `K` para qualquer tipo subtipo de `V`, e insere os elementos desse *map* no receptor.

O método que permite inserir pares chave-valor num *hashmap* é o método **`put(k,v)`**, que dada uma chave **`k`** de tipo `K` e um valor **`v`** de tipo `V`, insere o par no *map* receptor. O método **`V get(Object key)`** devolve o valor associado à chave dada como parâmetro, ou **`null`** se a chave não existir.

Os métodos **`clear()`**, **`isEmpty()`** e **`size()`** existem com os significados habituais. O método `clone()` realiza uma *shallow copy*.

O método **containsKey(Object k)** devolve `true` se o *map* receptor tem o objecto **k** como chave e se tal chave está associada a algum valor. **containsValue(Object v)** devolve o valor `true` se esse valor está associado a alguma chave.

Temos disponíveis os seguintes iteradores: **keySet()** que devolve uma "visão" sob a forma ou tipo de um `Set<K>` das chaves do *map* receptor, permitindo que, através de um *foreach* estas possam ser de imediato iteradas; o método **values()** devolve uma "visão" **Collection<V>** sobre os valores (pois estes podem estar em duplicado), que também pode ser iterada de imediato.

Quando dizemos "visão", tal significa que os métodos devolvem estruturas partilhadas com a colecção original, pelo que qualquer manipulação destas "visões" altera o original. Não são de facto cópias puras por razões de eficiência, por isso teremos que ter este facto em atenção ao utilizar estas "visões" fornecidas por certos métodos das colecções.

O método **remove(Object k)** remove a associação que tem `k` por chave deste *map*, devolvendo o valor associado a `k`, caso `k` exista, ou `null`, no caso contrário.

Finalmente, o método **putAll(Map<? extends K, ? extends V> m)** copia todos os pares do *map* `m` para o receptor, substituindo os pares do receptor pelos de `m`, caso as chaves sejam iguais (*sobreposição*). A cópia é, como sempre, de apontadores.

8.12.1 HASHCODE E equals()

As chaves de um `HashMap<K,V>` ou de um `HashSet<E>` são muito importantes na eficiência do mecanismo de *hashing*, que, nestas colecções, associa a cada chave um endereço. Este mecanismo usa por omissão o método `hashCode()` geral herdado de `Object`, sendo no entanto recomendável, excepto para as classes predefinidas e simples de JAVA, que, em todas as classes de objectos que vão ser chave ou elementos destas estruturas, seja definido um método `int hashCode()` próprio (que redefine o herdado) que devolve um *hashcode* para cada objecto da classe.

Este método `int hashCode()` deverá estar em coerência com o método `equals()` de tal classe, de tal forma que se verifique a seguinte propriedade:

```
x.equals(y) == true ⇒ x.hashCode() == y.hashCode()
```

O método `equals()` é muito importante neste tipo de estruturas, pois será com base nos seus resultados que será determinado se uma chave já existe num *hashmap*, ou se um dado elemento está ou não contido num *hashset*.

Não sendo fácil criar "boas" soluções de `hashCode()` para os nossos objectos, existem no entanto algumas regras simples que conduzem, em geral, a bons resultados.

- Na criação do *hashcode* usar valores das variáveis de instância que são relevantes no objecto, isto é, mais identificativas e diferenciadoras;
- Para variáveis de instância de tipos como `String`, `Integer`, etc., o próprio método `hashCode()` destas classes pode ser usado;
- Nos cálculos que conduzem à criação do *hashcode* único, usar pesos que sejam números primos.

Vejamos alguns exemplos usando os campos das classes que se identificam:

```
public int hashCode() {     // Ponto2D
   return x*7 + y*11;
}
public int hashCode() {     // Pessoa
   return 7*nome.hashCode() + 11*apelido.hashCode() + idade*13;
}
```

Para variáveis de tipo referenciado, se o seu valor for `null` o seu `hashValue` deverá ser 0, senão, deverá invocar-se o respectivo método `hashCode()`.

A garantia de coerência com o método `equals()` é fácil de conseguir, sendo de grande importância para a coerência das próprias estruturas de objectos que baseiam os seus algoritmos de comparação em tais códigos e no método `equals()`.

8.12.2 EXEMPLO: BANCO

Vamos criar em seguida uma classe `Banco` que nos vai permitir usar os métodos de `HashMap<K,V>` que foram apresentados. A classe `Banco` vai implementar as várias operações comuns sobre contas bancárias. A classe `Conta` representará a informação a ter sobre cada conta, designadamente: número da conta (único), titular, saldo, data de abertura e data do último movimento:

```
import java.util.GregorianCalendar;
import java.io.*;
public class Conta implements Serializable {
  private String numConta;
  private String titular;
  private double saldo;
  private GregorianCalendar dataInicio;
  private GregorianCalendar dataUltMov;
  // Construtores
  public Conta(String num, String tit, double saldo,
               GregorianCalendar dataInic,
               GregorianCalendar dataUMov) { . . }
  // Métodos de Instância
  public String getNumConta() { return numConta; }
  public double getSaldo() { return saldo; }
  public String getTitular() { return titular; }
  public GregorianCalendar getInicio() { return dataInicio; }
  public GregorianCalendar getUltMov() { return dataUltMov; }
  public void levantamento(double valor) { saldo -= valor; }
  public void deposito(double valor) { saldo += valor; }
  public void mudaUltMov(GregorianCalendar dataMov) {
     dataUltMov = dataMov; }
  public String toString() {
    StringBuilder s = new StringBuilder();
    s.append("-----------------------------\n");
    s.append("Conta N°: " + numConta + "\t\t\t");
    s.append("Titular: " + titular + "\n");
    s.append("Saldo: " + saldo + "\n");
    int ano = dataInicio.get(GregorianCalendar.YEAR);
    int mes = dataInicio.get(GregorianCalendar.MONTH);
    int dia = dataInicio.get(GregorianCalendar.DAY_OF_MONTH);
    s.append("Inicio: " + dia +"/"+ mes +"/"+ ano + "\t\t");
    ano = dataUltMov.get(GregorianCalendar.YEAR);
    mes = dataUltMov.get(GregorianCalendar.MONTH);
    dia = dataUltMov.get(GregorianCalendar.DAY_OF_MONTH);
    s.append("Últ. Mov: " + dia +"/"+ mes +"/"+ ano + "\n");
    return s.toString();
  }
  public boolean equals(Object ct) { .. }
  public Conta clone() { . . . }
}
```

A classe `Banco` vai conter o nome do banco como `String`, e uma variável de instância que vai associar a cada número de conta a informação da conta bancária respectiva, sendo portanto idealmente um *hashmap* do tipo **`HashMap<String, Conta>`**:

```
import java.util.*;
import java.io.*;
public class Banco implements Serializable {
```

```
    // Variáveis de Instância
    private HashMap<String, Conta> contas;
    private String nomeBanco;
    . . .
}
```

Comecemos por programar um construtor sem parâmetros que apenas vai inicializar o *hashmap* das contas bancárias:

```
public Banco(String nome) {
   contas = new HashMap<String, Conta>(); nomeBanco = nome;
}
```

e um construtor que copia um *hashmap* parâmetro para a variável de instância:

```
public Banco(HashMap<String, Conta> cts, String nome) {
   nomeBanco = nome;
   contas = new HashMap<String, Conta>();
   for(Conta cb : cts.values())
      contas.put(cb.getNumConta(), cb.clone());
}
```

Neste construtor, `cts.values()` devolve um `Iterator<Conta>` que iteramos usando um `foreach`. Cada objecto iterado é uma `Conta` que inserimos no *hashmap* `contas`, associando o seu número `cb.getNumConta()` à sua cópia `cb.clone()`, usando o metodo `put(k,v)`.

Note-se que, se, porventura, houvesse a possibilidade (bem real) de o banco vir a ter que gerir vários tipos de contas, como por exemplo, Conta Normal, Conta a Prazo, etc., então, toda a concepção desta aplicação deveria mudar: `Conta` deveria passar a ser uma classe abstracta e `ContaNormal`, `ContaPrazo`, etc., seriam subclasses desta. Assim, para que o anterior construtor pudesse ser genérico, deveríamos modificar o seu parâmetro para:

```
public Banco(HashMap<String, ? extends Conta> cts, … ) {
```

preparando-o para a eventualidade de receber como parâmetro um *hashmap* de `String` para `ContaPrazo` (ou seja, apenas com contas a prazo), ou outros semelhantes.

Depois de um conjunto de operações simples cuja semântica se indica em comentário,

```
// Número total de contas
public int numContas() { return contas.size(); }
// Nome actual do Banco
public String getNomeBanco() { return nomeBanco; }
public void mudaNmBanco(String nvNm) { nomeBanco = nvNm; }
```

a operação seguinte a programar para a classe `Banco` será a operação que dá como resultado a informação sobre a conta bancária de número indicado. Esta operação pressupõe que o número da conta existe no conjunto das chaves do *hashmap* `contas`.

```
public Conta daConta(String numConta) {
     return contas.get(numConta).clone(); // devolve cópia
}
```

Caso o número de conta não exista, a operação dá um erro de execução porque o resultado do método `get()` é o valor `null`, e o resultado de se tentar fazer o `clone()` de `null` é um erro de execução. Assim, antes da invocação deste método, deve ser sempre executado o método seguinte, que corresponde à pré-condição da operação:

```
public boolean existeConta(String numConta) {
    return contas.containsKey(numConta);
}
```

O método usa `containsKey()` para testar se uma dada *string* (que é o número da conta) pertence ao conjunto das chaves do *hashmap* de contas.

O método seguinte insere uma nova conta no *hashmap* de contas bancárias caso o número de conta ainda não exista:

```
public void criaConta(Conta conta) {
   contas.put(conta.getNumConta(), conta.clone());
}
```

O uso de `clone()` garante que o que se insere no *hashmap* é uma cópia do parâmetro.

A Figura 8.39 mostra o resultado de, num programa `main()` onde criámos um banco com algumas contas (criadas com valores gerados automaticamente), as termos listado usando (implicitamente em `println()`) o método `toString()` definido em `Conta`. Cada conta foi obtida iterando sobre as chaves da instância de `Banco` e usando o método que devolve uma `Conta` a partir da sua chave (o seu número):

```
ArrayList<String> codigos = new ArrayList<String>();
codigos = meuBanco.listaNumeros();
out.println("------- Contas --------");
for(String num : codigos)
   out.println(meuBanco.daConta(num));
```

A iteração usando os valores do *hashmap* (método `values()`) para *output* da informação de cada `Conta` torna o código ainda mais simples, conforme:

```
for(Conta ct : meuBanco.values())
   out.println(ct); // usa automaticamente toString() de Conta
```

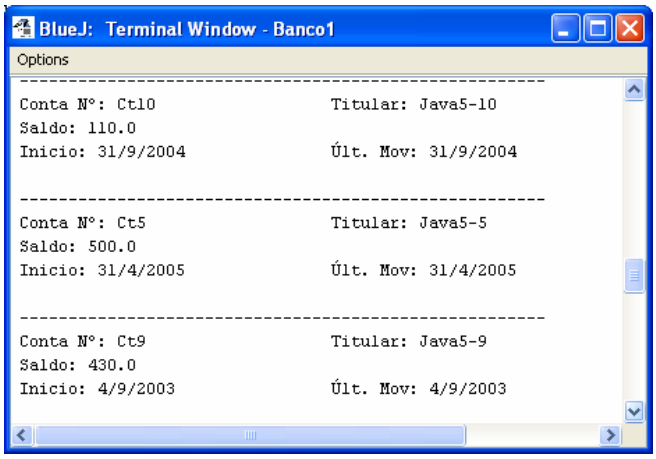


Figura 8.39 – Listagem de contas

Vamos em seguida realizar algumas operações de filtragem, operações que tipicamente se baseiam na utilização de iteradores.

```
public ArrayList<String> listaNumeros() {
   ArrayList<String> nums = new ArrayList<String>();
   for(String numc : contas.keySet())
      nums.add(numc); // clone() não necessário
   return nums;
}
```

O método `listaNumeros()` cria um *arraylist* contendo as *strings* que são os números de todas as contas existentes. Para tal, é criado um iterador *foreach* sobre o conjunto `Set<String>` devolvido pelo método `keySet()`, sendo cada um dos números de conta inserido no *arraylist* auxiliar `nums`. O método dá como resultado este *arraylist*, mas, de facto, deveria dar como resultado um conjunto, pois não existem duplicados.

Repare-se qual seria a diferença no código do método caso tivéssemos que criar um conjunto de números de conta, por exemplo, usando um `HashSet<String>`:

```
public HashSet<String> conjNumeros() {
   HashSet<String> cnums = new HashSet<String>();
   for(String num : contas.keySet())
      cnums.add(num);
   return cnums;
}
```

As duas únicas diferenças no código estão sublinhadas a negrito. A razão é simples: todas as colecções implementam `Collection<E>` e, portanto, a maioria dos métodos de listas e de conjuntos possui o mesmo nome e assinatura (claro que as suas implementações são distintas). Um deles é o método `add()`.

O método seguinte devolve uma lista contendo os números das contas com saldo superior a um valor dado como parâmetro.

```
public HashSet<String> codCtSldMaiorQue(double valor) {
   HashSet<String> cods = new HashSet<String>();
   for(Conta conta : contas.values())
      if( conta.getSaldo() > valor )
           cods.add(conta.getNumConta());
   return cods;
}
```

Dado que não existirão números duplicados no resultado, este deverá ser um conjunto, pelo que usámos um `HashSet<String>` para guardar os números que satisfazem a condição de selecção. A iteração foi feita usando `values()`, o que nos dá acesso imediato a todas as contas existentes. Note-se que usar `values()` é equivalente a iterar pelo conjunto das chaves, depois de usar `contas.keySet()`, e, para cada chave, usar o método `get(chave)` para obter o respectivo valor associado. Mas, para que se possa obter o número da conta a partir da ficha da conta, houve a necessidade de ser redundante e colocar tal número quer como chave quer como campo da ficha. Tal redundância de dados é, em geral, compensadora quando se usam *hashmaps*.

Um outro método muito semelhante poderia mesmo dar como resultado um conjunto com a informação de todas as contas com saldo superior a um dado valor (Figura 8.40):

```
public HashSet<Conta> ctsSldMaiorQue(double valor) {
   HashSet<Conta> cts = new HashSet<Conta>();
   for(Conta conta : contas.values())
      if( conta.getSaldo() > valor )
           cts.add(conta.clone());
   return cts;
}
```

A utilização de `clone()` é fundamental para se garantir que não há partilha.

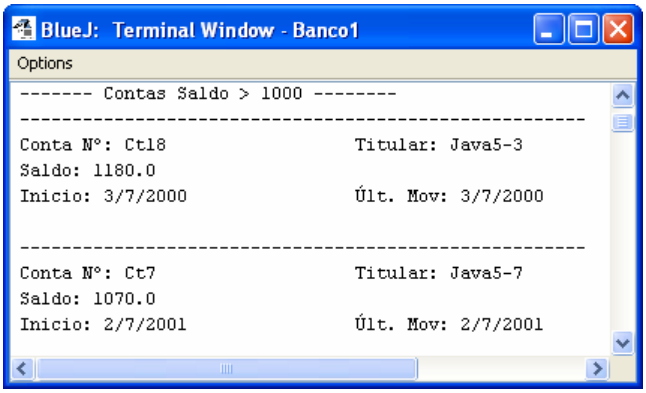


Figura 8.40 – Listagem de contas por dado critério de selecção

O método seguinte regista o levantamento de uma quantia numa conta cujo número é dado. O método assume que o número de conta é válido, e que o saldo da conta torna tal levantamento possível (cf. pré-condição):

```
public void levantamento(String numConta, double valor) {
   Conta conta = contas.get(numConta);
   conta.levantamento(valor);
   conta.mudaUltMov(new GregorianCalendar());
   contas.remove(numConta);
   contas.put(numConta, conta);
}
```

Este método realiza uma alteração na ficha de uma conta modificando o seu saldo. Na sua codificação, depois de se ter obtido a ficha relativa a tal conta, alterou-se o saldo atribuindo-lhe um novo valor. Assim, a ficha contida na variável **`conta`** foi modificada e tem o novo valor do saldo. Como estará neste momento a ficha original que está no *hashmap*? Igual ao que estava ou terá sido igualmente alterada? Enfim, depende de como funcionar o método `get()`.

Ora, o método `get()` devolve uma referência para a ficha que está no *hashmap*, pelo que, quando fazemos `conta.levantamento(valor)`, estamos a alterar a ficha que está no *hashmap*. Assim, após esta instrução o saldo já foi modificada na ficha do *hashmap*, tal como queríamos.

As operações a negrito são portanto dispensáveis nos *hashmaps* de JAVA, porque `get()` faz partilha da referência do objecto obtido. No entanto, se `get()` nos devolvesse uma cópia da ficha, então, trabalharíamos sobre a cópia realizando as modificações necessárias, e, em seguida, removíamos do *hashmap* o par existente (chave/valor antigo), e inseríamos o novo par (chave/valor modificado). Dado que `get()` faz partilha, tal não é necessário.

O código do método, ainda que possa parecer estranho por não parecer codificar um verdadeiro *update*, mas que realmente realiza, será portanto:

```
public void levantamento(String numConta, double valor) {
   Conta conta = contas.get(numConta);
   conta.levantamento(valor);
   conta.mudaUltMov(new GregorianCalendar());
}
```

A variável que a cada momento contém a data do último movimento realizado é actualizada por este método, sendo-lhe atribuída a data actual, `new GregorianCalendar()`, que é dada como parâmetro do respectivo método modificador.

A Figura 8.41 apresenta o resultado de se ter realizado um levantamento numa das contas, sendo de salientar a actualização da data do último movimento, que ficou a ter a data do sistema quando este levantamento foi efectuado sobre a conta.

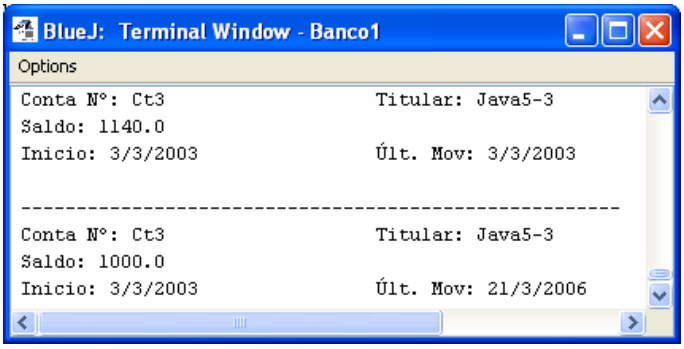


Figura 8.39 – Estado da conta antes e após um levantamento

O método que realiza um depósito na conta cujo número é dado é, naturalmente, um método sem restrições, e com código semelhante ao anterior:

```
public void deposito(String numConta, double valor) {
   contas.get(numConta).deposito(valor);
   contas.get(numConta).mudaUltMov(new GregorianCalendar());
}
```

A remoção de uma conta existente consiste na invocação do método `remove()` contendo a chave (número de conta) que permite remover a associação chave-valor correspondente:

```
public void removeConta(String numConta) {
   contas.remove(numConta);
}
```

O método auxiliar `numDias()` determina o número de dias decorridos entre o dia actual e a data dada como parâmetro. Este método será posteriormente usado noutros métodos que necessitem de trabalhar com as datas de movimentação ou de criação das contas:

```
protected long numDias(GregorianCalendar data) {
 GregorianCalendar hoje = new GregorianCalendar();
 long hojeMils = hoje.getTimeInMillis();
 long dataMils = data.getTimeInMillis();
 long milis = hojeMils - dataMils;
 return milis/(24*60*60*1000);
}
```

A classe `GregorianCalendar` é um complexo calendário que, quando é instanciado, se inicia na era, século, ano, dia, data, hora, etc., actuais, e que possui métodos para consulta de cada um destes parâmetros de uma data. As diferenças entre duas datas, quando se pretende precisão, são medidas em milissegundos, determinando para cada uma delas o seu valor em milissegundos e calculando a diferença. A partir da sua diferença em milissegundos, as sucessivas divisões darão os segundos (por 1000), os minutos (por 60), as horas (por 60) e os dias (por 24).

O método é agora usado para determinar os números das contas que não são movimentadas há mais de um dado número de dias (Figura 8.42):

```
public HashSet<String> contasParadas(int numDias) {
   HashSet<String> nums = new HashSet<String>();
   GregorianCalendar hoje = new GregorianCalendar();
   for(Conta c : contas.values())
     if( this.numDias(c.getUltMov()) > numDias )
        nums.add(c.getNumConta());
   return nums;
}
```

Figura 8.42 – Listagem de contas não movimentadas

O resultado apresentado na figura anterior resulta da execução do seguinte código escrito no método `main()`, em que a variável `meuBanco` é do tipo `Banco`:

```
HashSet<String> nums = new HashSet<String>();
nums = meuBanco.contasParadas(365);
out.println("------- Contas Paradas --------");
for(String num : nums)
   out.println(meuBanco.daConta(num).toString());
```

O método seguinte vai criar um *hashmap* a partir do *hashmap* `contas`. Pretende-se criar um *hashmap* que associe a cada nome de titular o somatório dos saldos das suas contas, pelo que se pretende criar um *hashmap* do tipo `HashMap<String, Double>`. Antes porém, vamos criar um método mais simples, que nos irá auxiliar na construção do seguinte. Este método, `totalTitular(String nome)`, recebe o nome de um titular de uma ou de mais contas, e determina o valor total destas:

```
public double totalTitular(String nome) {
   double total = 0.0;
   for(Conta cb : contas.values())
```

```
      if(cb.getTitular().equals(nome)) total += cb.getSaldo();
    return total;
  }
```

O método que pretendemos desenvolver para criar o *hashmap* que a cada titular vai associar o valor total das suas contas, vai agora criar o conjunto dos titulares e, iterando sobre esse conjunto, para cada titular invocar o método anterior para calcular o seu saldo total. Os resultados do método são apresentados na Figura 8.43.

```
public HashMap<String, Double> totalTitular() {
  HashMap<String, Double> valTit =
            new HashMap<String, Double>();
  HashSet<String> tits = new HashSet<String>();
  for(Conta cb : contas.values())
    tits.add(cb.getTitular());
  for(String tit : tits)
    valTit.put(tit, new Double(this.totalTitular(tit)));
  return valTit;
}
```

A frase `new Double(this.totalTit(tit))` apenas é aqui escrita porque ainda não falámos nos mecanismos de *boxing* e *unboxing* automáticos usando *maps*, o que faremos na secção seguinte. No entanto, tal como para as outras colecções, para os *maps* também existem, pelo que podíamos escrever apenas `put(tit, this.totalTitular(tit))` correspondente à assinatura `put(String,double)`.

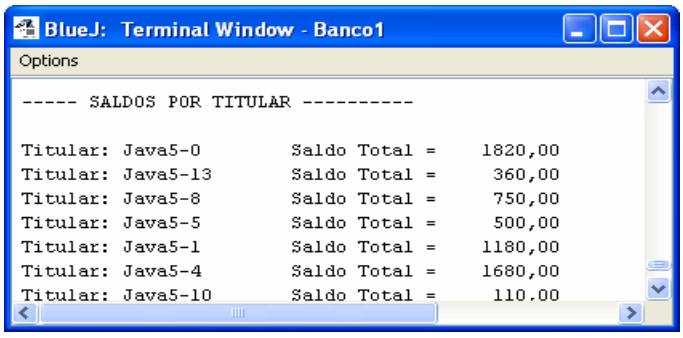


Figura 8.43 – Resultado da operação `totalTitular()`

O código escrito em `main()` para a obtenção deste resultado seria simplesmente:

```
HashMap<String, Double> tt = new HashMap<String, Double>();
tt = meuBanco.totalTitular();
for(String nm : tt.keySet()) {
   out.print("Titular: " + nm + "\tSaldo Total = ");
   out.printf("%10.2f%n", tt.get(nm));
}
```

Finalmente, um método que remove do *hashmap* de contas todas as contas não movimentadas há mais de um número de anos dado como parâmetro. O método não só remove as contas em tais condições como devolve um conjunto contendo os respectivos números de conta. O resultado é do tipo `Set<String>`.

```
public Set<String> remParadasXAnos(int numAnos) {
  HashSet<String> nums = new HashSet<String>();
  for(Conta c : contas.values())
    if( numDias(c.getUltMov()) >= 365*numAnos )
      nums.add(c.getNumConta());
  for(String nc : nums) contas.remove(nc);
  return nums;
}
```

Este primeiro exemplo de utilização de *hashmaps* permitiu-nos não só utilizar praticamente todos os métodos desta classe, como verificar que, com grande facilidade, podemos mesmo utilizar algumas outras colecções sem grandes problemas, dada a semelhança das suas API, por terem `Collection<E>` como interface/tipo comum.

8.12.3 MODIFICAÇÃO DAS CHAVES DE UM MAP

Quando se insere um par chave-valor num *map*, o algoritmo que vai associar um endereço físico a essa chave baseia-se naturalmente no valor da mesma. Daí que tudo o que tenha a ver com igualdade de chaves, valores `null`, e `hashCode()` dos objectos, sejam assuntos bastante delicados, quando quer as chaves quer as aplicações são complexas.

No entanto, antes das questões de optimização de utilização, há questões de robustez que são de uma importância crucial e que devem ser abordadas. Uma destas questões tem a ver com a necessidade, nem sempre evidente, de se realizar uma modificação correcta do valor de uma chave existente num `Map<K,V>`.

Seja, por exemplo, `planoEsp` uma variável do tipo `HashMap<Ponto,Integer>`, que a cada ponto (`Ponto2D`) de coordenadas inteiras associa um `Integer` que é o peso desse ponto no plano.

Admitamos que no conjunto das chaves deste plano especial existe uma chave que é a instância `Ponto(-2,3)`, que, por exemplo via um iterador, passámos a referenciar pela variável `p`, ou seja, `p => Ponto(-2,3)`.

A variável `p` partilha com o `HashMap<Ponto,Integer>` o endereço onde se localiza a chave `Ponto(-2,3)`, endereço esse que lhe foi atribuído porque a chave valia exactamente `Ponto(-2,3)` no momento da inserção.

Se agora escrevermos `p.desloca(2,2)`, a chave passa a "valer" `Ponto(0,5)`, está associada ao mesmo valor `Integer`, ocupa a mesma localização no *hashmap*, mas, muito provavelmente, deveria estar agora noutro endereço. Por tal motivo, quando escrevermos `planoEsp.contains(Ponto(0,5));` não há qualquer garantia de que o resultado seja `true`. O `HashMap<Ponto,Integer>` deixa portanto de ser consistente tornando-se imprevisível e corrompido.

Não se devem alterar valores de chaves mantendo-as na sua localização original, pois esta localização pode não corresponder à localização que lhes seria atribuída pela função de *hash*. A solução correcta será, nestes casos, obter a chave, fazer o seu `clone()`, fazer `get()` do respectivo valor, alterar à vontade a cópia da chave, remover a chave antiga e inserir a nova associação, tal como se mostra no código seguinte:

```
Ponto ponto = pontoChave.clone();     // copia a chave
Integer val = planoEsp.get(ponto);    // busca valor
ponto.desloca(x, y);                  // altera chave
planoEsp.remove(pontoChave);          // remove antiga
planoEsp.put(ponto, val);             // insere novo par
```

Vamos em seguida estudar o mecanismo de *auto-boxing* e *auto-unboxing* tal como pode ser aplicado a *hashmaps*.

8.12.4 AUTO-BOXING E AUTO-UNBOXING EM MAPS

Tal como as outras colecções, os *maps* apenas permitem a inserção de objectos, mas, tal como as outras, têm a si associados mecanismos de *auto-boxing* e *auto-unboxing* que os compatibilizam com tipos primitivos.

Consideremos um pequeno exemplo usando uma classe **`Populacao`**, que vai conter um *hashmap* que associa a cada nome de cidade a sua população sob a forma de um valor inteiro. O *hashmap* que constituirá a variável de instância da classe `Populacao` será do seguinte tipo:

```
HashMap<String, Integer> populacao;
```

Dado que os valores são `Integer`, usaremos ***auto-boxing*** para converter para objectos os parâmetros `int` dos métodos, e vice-versa.

```
public class Populacao {
  // Variável de Instância
  private HashMap<String, Integer> populacao;
  // Construtores
  public Populacao() {
```

```
    populacao = new HashMap<String, Integer>();
  }
  public Populacao(Map<String, Integer> pop) {
    populacao = new HashMap<String, Integer>();
    populacao.putAll(pop);
  }
```

Recebido o *map* parâmetro, foi usado o método `putAll()` para inserir todos os pares no *map* que é a nossa variável de instância. Não foram realizadas operações de `clone()` pois quer as `String` quer os `Integer` são instâncias imutáveis:

```
// Métodos de Instância
public void insereCidade(String cidade, int pop) {
  populacao.put(cidade, pop); // boxing
}
```

Dado um par cidade-população, e admitindo que foi previamente testado que a cidade não existe no conjunto das chaves do *hashmap*, este método usa o método `put()` para inserir o par. A variável `pop`, sendo de tipo `int`, é convertida no `Integer` equivalente:

```
// Devolve a população de uma cidade existente
public int populacao(String cidade) {
   return populacao.get(cidade); // unboxing
}

// Devolve o nome da cidade mais populosa
public String maisPopulosa() {
  int maxPop = 0; String nomeCidade = "";
  int popCidade = 0;
  Set<String> cidades = populacao.keySet();
  for(String cidade : cidades) {
    popCidade = populacao.get(cidade); // unboxing
    if( popCidade > maxPop ) {
      nomeCidade = cidade;
      maxPop = popCidade;
    }
  }
  return nomeCidade;
}

// Devolve a média de população por cidade
public double mediaDasPopulacoes() {
  int total = 0;
  for(int pop : populacao.values()) // unboxing
        total += pop;
  return total/populacao.size();
}

// Devolve um array de inteiros com as populações de
// todas as cidades (uso de UNBOXING)
public int[] daPopulacoes() {
  int[] pops = new int[populacao.size()];
  int index = 0;
  for(Integer pop : populaces.values()) {
        pops[index] = pop; // unboxing aqui
        index++;
  }
  return pops;
}

// Devolve um conjunto com os nomes das cidades
public HashSet<String> cidades() {
  HashSet<String> nomCidades = new HashSet<String>();
  for(String cidade : populacao.keySet()) nomCidades.add(cidade);
```

```
    return nomCidades;
  }

  // toString()
  public String toString() {
    StringBuffer s = new StringBuffer();
    s.append("--------   POPULAÇÕES ----------------\n");
    for(String cidade : populacao.keySet()) {
        s.append("Cidade : "); s.append(cidade);
        s.append("\t\t População : ");
        s.append(populacao.get(cidade) + "\n"); // unboxing
    }
    s.append("-----------------------------------\n\n");
    return s.toString();
  }
}
```

Como se pode verificar, *auto-boxing* e *auto-unboxing* tornam a vida do programador mais simples, evitando as preocupações de conversão. No caso dos *hashmaps*, esta vantagem é, aliás, acrescida quando temos *maps* numéricos do tipo `HashMap<Integer,Double>` (e são muitos, como estatísticas, mapas de ocorrências, etc.). Consideremos, por exemplo, uma tabela de nascimentos por ano e código que usa *auto-boxing* e *auto-unboxing*:

```
HashMap<Integer, Integer> tabNatal =
           new HashMap<Integer, Integer>();
tabNatal.put(1950, 570.345);
tabNatal.put(1951, 575.876);
. . . .
int totNasc = 0;
// total de nascimentos
for(int nasceram : tabNatal.values())
   totNasc += nasceram;
. . .
out.printf("Em %d nasceram %8.3d crianças.%n", ano, nasc);
. . .
int totAnos = tabNatal.size();
out.println(totAnos + " nasceram " + totNasc + "bebés.";
```

Apenas nestes casos é natural que as chaves numéricas sejam associadas a `Integer` e, portanto, possamos usufruir de *boxing* e *unboxing* automáticos, já que, nas aplicações informáticas, as chaves de acesso numéricas, tais como número de aluno, número do bilhete de identidade, etc., são implementadas usando o tipo `String`.

8.13 *WILDCARDS:* PARTE II

Wildcards são usados como especificadores dos tipos parâmetro aceitáveis por um tipo parametrizado, mas um tipo parametrizado com *wildcards* não pode ser usado para criar objectos ou *arrays* com `new` (salvo, no caso dos *arrays*, uma raríssima excepção sem interesse prático).

Vimos atrás o significado dos *wildcards* `?` e `? extends E` na especificação dos tipos parâmetro das colecções. O *wildcard* `? extends E` é particularmente importante pois restabelece a **covariância** nos tipos parametrizados que são resultantes do mesmo tipo genérico e para um dado tipo `E`, como `ArrayList<? extends E>` e `ArrayList<E>`, e, de modo semelhante, para todos os outros tipos.

Foram também definidas algumas das principais regras de compatibilidade entre colecções do mesmo tipo parametrizado em função do *wildcard* especificador. Para que todas as regras de compatibilidade de tipos possam ser apresentadas, temos primeiro que apresentar todas as formas possíveis de especificação de parâmetros, em especial a última forma de *wildcard*.

Alguns métodos exigem que os seus tipos parâmetro sejam, no mínimo, de um dado tipo, ou seja, especificam **um limite inferior de tipo** (*lower bound type*). A especificação de que o tipo parâmetro é indiferente e/ou

desconhecido mas deve satisfazer tal regra, faz-se usando um ***lower bounded wildcard*** da forma `? super E`, que se lê: **"um tipo qualquer que seja `E` ou supertipo de `E`"**.

Certas colecções que vamos estudar, tais como `TreeMap<E,V>` e `TreeSet<E>`, ordenam os seus elementos por uma ordem que pode ser fornecida aquando da sua criação através de uma classe que implemente a interface `Comparator<? super E>`. Tal significa que, se pretendermos comparar objectos do tipo `String`, podemos passar como parâmetro `Comparator<String>`, mas, se passarmos `Comparator<Object>` também estará correcto pois `String`s são `Object`s, e portanto `Object` :> `String`.

Temos em síntese, conforme se apresenta no Quadro 8.4, três formas de utilização dos *wildcards* na especificação de tipos argumento de tipos parametrizados. Outras formas com *wildcards* e sem *wildcards* que não referiremos aqui podem ser usadas, mas apenas na especificação de tipos argumento de classes e métodos genéricos.

Wildcard	Tipos representados
`?`	Qualquer tipo
`? extends E`	`E` e qualquer subtipo de `E`
`? super E`	`E` e qualquer supertipo de `E`

Quadro 8.4 – *Wildcards* como especificadores de argumentos

Interessa-nos agora analisar que hierarquia ou hierarquias de tipos são estabelecidas como resultado da utilização dos *wildcards*, para termos a certeza de todas as compatibilidades possíveis entre tipos parametrizados com e sem *wildcards*.

Relembrando essa hierarquia genérica de tipos, agora de forma mais geral, ou seja, para famílias de colecções de tipo monoparâmetro, sabemos já que se verifica (Figura 8.44):

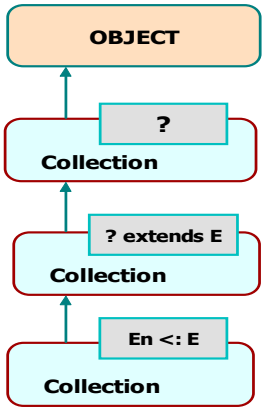


Figura 8.44 – Compatibilidade com *wildcards*

A classe `Object` é o supertipo de todos os tipos, e cada `Collection<>` com *wildcard* uma família de colecções relacionadas com o tipo `E` a instanciar.

No entanto, e tal como dissemos atrás, quando falamos de **hierarquia** neste contexto, devemos clarificar bem o conceito. Em primeiro lugar, a única verdadeira hierarquia que existe é a hierarquia que contém as classes e interfaces genéricas e não genéricas, tais como: `Object`, `Serializable`, `Collection<E>`, `List<E>`, `ArrayList<E>`, etc. Porém, as instanciações dos tipos genéricos criam tipos parametrizados que vão ter entre si certas relações de **supertipo-subtipo** (e de compatibilidade de tipos), que são muito importantes de se compreender para que se invoquem bem métodos, se realizem bem atribuições e se façam *castings* correctos.

Estas relações de supertipo-subtipo em conjunto com a hierarquia de classes fazem parte do sistema de tipos da linguagem e **é este sistema de tipos que determina que tipos são compatíveis com outros**.

Assim, as relações supertipo-subtipo determinadas pela instanciação de tipos genéricos e as relações supertipo-subtipo determinadas pela hierarquia de classes vistas como tipos são dois aspectos distintos que irão contribuir

para a hierarquia final de classificação e de compatibilidades. É, no entanto, de salientar que são aspectos bem distintos.

Dada a complexidade de representação de tal hierarquia, opta-se por uma tabela de síntese que se apresenta na Figura 8.45.

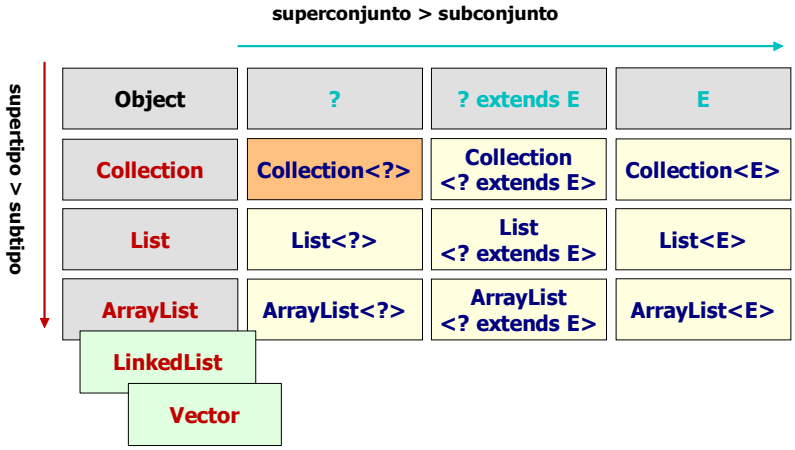


Figura 8.45 – Tabela de compatibilidades de tipos

No eixo vertical, esta tabela apresenta a hierarquia dos tipos parametrizados em função das suas relações de herança, ou seja, posicionamento na hierarquia de `java.util` (no exemplo, apenas para o ramo `List<E>`). No eixo horizontal apresentam-se famílias de tipos parametrizados de acordo com as suas relações de inclusão de tipos.

São compatíveis com um tipo de uma dada célula, todos os tipos da subtabela abaixo e à direita da célula desse tipo. No exemplo, são compatíveis com `Collection<?>` todos os tipos parametrizados das células a cinza claro.

Uma `Collection<Object>` não é supertipo de `Collection<String>` nem de outra colecção qualquer, e todas estas têm como supertipo o tipo **`Collection<?>`**, ainda que, em função do tipo de `E`, possam ter superclasses como `Collection<? extends E>`.

Algumas dúvidas poderão no entanto surgir quanto às diferenças entre os seguintes tipos de *colecções*:

- `Collection<Object>;`
- `Collection<? extends Object>;`
- `Collection<?>;`
- `Collection` (não usar).

Em primeiro lugar, `Collection<Object>` e `Collection` são tipos concretos enquanto que as outras duas formas não, e representam famílias de tipos que são instanciações possíveis de `Collection<E>`. `Collection` é não parametrizado, e portanto não tipado, sendo todos os seus elementos do tipo `Object`. É um ***raw type*** e é heterogéneo.

Qualquer *raw type* tem por supertipo o tipo parametrizado sem limite do mesmo nome. Assim, `Collection` é subclasse de `Collection<?>`, `List` é subclasse de `List<?>`, etc., sendo tipos muito diferentes na forma de utilização.

Coloca-se também a questão de saber quais os tipos dos elementos que devemos considerar que nelas estão contidos quando, por exemplo, fazemos uma operação de `get()` ou, o que de certo modo é equivalente, uma iteração com *foreach*. Que tipo de elemento se deve especificar em cada caso como sendo extraído destas colecções?

Consideremos o caso de um método auxiliar `static`, a escrever num programa principal, que deve imprimir em ecrã os elementos de uma colecção. A sua forma geral será:

```
public static void printColl(Collection_tipo c) {
  for(tipoElem elem : Collection_tipo)
```

```
        out.println(elem);
    }
}
```

Temos que resolver para cada caso as incógnitas (***Collection_tipo,tipoElem***), ou seja, para cada tipo de parâmetro determinar qual o tipo do elemento a usar. O algoritmo é simples, tendo que encontrar o tipo mais geral, mais abrangente possível para cada caso. A resposta para todas é `Object`, pois, para todas, a única certeza que o compilador tem é que as instâncias contidas em tais colecções serão compatíveis com `Object`.

Concluímos assim o estudo dos *wildcards* como mecanismo sintáctico que nos permite especificar as possíveis instanciações dos tipos argumento de tipos parametrizados. A sua compreensão é de momento fundamental para uma boa reutilização das colecções existentes em JAVA5 para a criação das nossas próprias classes.

8.14 TreeMap<K,V>

A classe `TreeMap<K,V>` é usada exactamente da mesma forma que um *hashmap* mas possui a vantagem de manter os pares chave-valor guardados pela *ordem crescente* das suas chaves, ordem essa predefinida ou não em função do tipo das chaves.

As chaves de tipos mais comuns, como `String` e `Integer`, são ordenadas segundo a sua "ordem natural". Para outros tipos mais complexos, em especial classes definidas pelo utilizador, este deverá fornecer o método de comparação de chaves como parâmetro do construtor `TreeMap<K,V>(Comparator<? super K> c)`.

A classe `TreeMap<K,V>` possui um construtor que aceita por parâmetro um `Map<K,V>`, bem como o método `putAll()`. Tal significa que um qualquer *hashmap* pode ser convertido de forma *shallow* num *treemap* correspondente, usando ou o construtor ou o método de instância:

```
TreeMap<String, Conta> ctsOrd =
     new TreeMap<String, Conta>(contas);  // ordena por chave
TreeMap<String, Conta> ctsOrd1 =
     new TreeMap<String, Conta>();
ctsOrd1.putAll(contas);                   // ficam ordenadas
```

A vantagem óbvia é que, de imediato, as chaves são ordenadas por ordem crescente dos valores das *strings* dos números de conta. Por outro lado, conforme podemos verificar pela API de `TreeMap<K, V>`, passamos a ter disponíveis métodos que devolvem *submaps* ordenados pelas chaves (do tipo `SortedMap<K,V>`) que podem ser iterados. O método `headMap(k)` dá-nos o *submap* inicial até à chave dada, `tailMap(k)` o *submap* final e `subMap(ki,kf)` o *submap* entre as chaves dadas. Em todos estes *submaps*, os limites são exclusive (abertos), excepto o limite inferior do método `subMap()`. A forma especial `k+"\0"` permite que a chave `k` faça também parte dos limites (limite fechado).

TreeMap<K, V> métodos adicionais
`TreeMap<K, V>()`
`TreeMap<K, V>(Comparator<? super K> c)`
`TreeMap<K, V>(Map<? extends K, ? extends V> m)`
`K firstKey()`
`SortedMap<K, V> headMap(K toKey)`
`K lastKey()`
`SortedMap<K, V> subMap(K fromKey, K toKey)`
`SortedMap<K, V> tailMap(K fromKey)`

Quadro 8.5 – API parcial de `TreeMap<K,V>`

Antes de apresentarmos um exemplo completo usando `TreeMap<K,V>` vejamos alguns exemplos simples de utilização dos construtores e métodos particulares desta classe.

Comecemos por uma simples **agenda de telemóvel** representada por uma simples variável do tipo *hashmap*, conforme a declaração seguinte, na qual inserimos pares nome-número de telemóvel, e que no final convertemos em `TreeMap<String,String>`:

```
HashMap<String>, <String> agendaH =
          new TreeMap<String>, <String>();
agendaH.put("Carlos Efe","990120879");
. . .
TreeMap<String, String> agendaT =
            new TreeMap<String, String>(agendaH);
for(String nm : agendaT.keySet())
  out.println("Nome: " + nm + " -- " + agendaT.get(nm));
```

Executando em seguida a iteração sobre o `keySet()` de cada agenda, verifica-se que as chaves no caso do *hashmap* surgem por uma ordem qualquer, enquanto que no *treemap* nos são apresentadas pela sua ordem natural, que é a ordem crescente do tipo `String`.

A Figura 8.46 apresenta os resultados destas duas iterações simples sobre as chaves, indicando quais são as chaves de cada um dos *maps*.

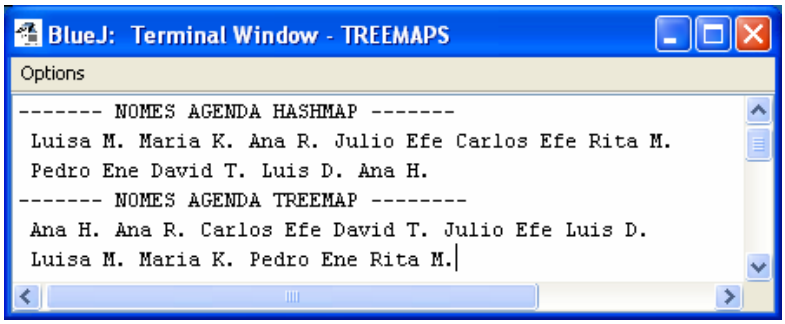


Figura 8.46 – `keySet()` de `TreeMap<String, String>`

Vamos usar os métodos que devolvem *submaps*, para, sobre a variável `agendaT`, realizarmos algumas selecções de chaves e valores.

Por exemplo, pretendemos apenas o conjunto de nomes entre a letra C e a letra M, inclusive, pelo que temos que usar o método `subMap()`, que devolve um `SortedMap<,>`:

```
SortedMap<String, String> nms = agendaT.subMap("C","M"+"\0");
for(String nm : nms) out.printf(nm + " ");
```

ou seleccionar apenas os pares nome-número de telemóvel dos nomes começados por L, começando por apagar os resultados guardados no *sortedmap* resultante da selecção anterior, e realizando a nova selecção:

```
nms.clear(); nms = agendaT.subMap("L","M");
for(String nm : nms) out.printf(nm + " ");
```

Os resultados são apresentados na Figura 8.47, mas necessitam de uma análise parti-cular já que o resultado da segunda selecção foi um *map* vazio!

Figura 8.47 – Resultados de `subMap()`

Como se pode verificar pela própria janela de resultados, existem dois nomes iniciados pela letra L, pelo que não há razão aparente para a segunda invocação de `subMap()` dar como resultado um *map* vazio.

Cometemos porém um erro muito comum e que é em geral cometido com todas as colecções, e com todos os métodos destas que nos permitem ter acesso a "visões", isto é, partes das colecções. É que estas **visões** são apenas uma espécie de "lupa" sobre o original, mas o que estamos a manipular é mesmo o original através da lupa, porque não há cópia dos elementos da colecção. Assim, alterações na **visão** alteram o original.

No código anterior, ao fazermos `nms.clear()` apagámos do *map* original os pares correspondentes aos elementos que havíamos filtrado na selecção anterior, como se pode comprovar pelos resultados da Figura 8.48 após nova iteração sobre as chaves restantes.

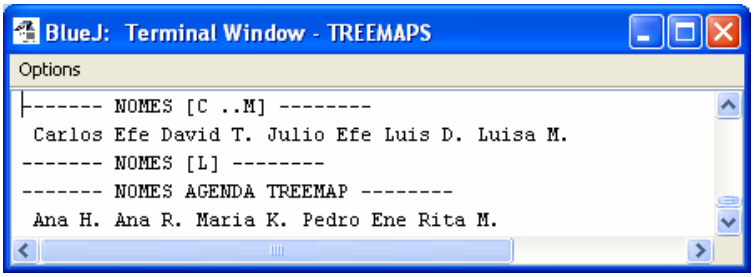


Figura 8.48 – Resultados de `subMap().clear()`

Porém, e tal como a legenda da figura indica, vejamos os resultados obtidos pela positiva, e pensemos que "descobrimos" uma excelente forma de **remover *submaps*** de um dado *map*. Usamos `subMap()` ou outro, para seleccionar a porção do *map* que se pretende, e em seguida usamos `clear()`:

```
agendaT.tailMap("X").clear(); // remove assoc de chave>="X"
agendaT.subMap(agendaT.firstKey(),"I").clear(); // [1ª.."I"[
agendaT.headMap("I").clear(); // [1ª .. "I"[
```

Falta-nos apenas ver como podemos criar *treemaps* com um comparador de chaves por nós definido, o qual deve ser passado como parâmetro para o construtor de *treemap* usado na declaração e criação da instância do *treemap*.

Pretendemos que a variável `planoEsp` seja agora um `TreeMap<Ponto,Integer>`. A classe `Ponto` não tem sobre si definida qualquer ordem natural, pelo que necessitamos de definir um método de comparação de pontos (nota: `Ponto` ≡ `Ponto2D`).

O construtor `TreeMap<K,V>(Comparator<? super K> c)` espera que lhe seja passado como parâmetro um *comparator* de tipo `K` ou de um supertipo de `K`. No nosso caso, em que pretendemos criar um `TreeMap<Ponto,Integer>`, temos pois que criar uma classe que implemente a interface `Comparator<Ponto>`:

```
import java.util.Comparator;
import java.io.Serializable;
public class PontoComparator
       implements Comparator<Ponto>, Serializable {
  public int compare(Ponto p1, Ponto p2) {
   if( p1.getX() < p2.getX() ) return -1;
   if( p1.getX() > p2.getX() ) return 1;
   if( p1.getY() < p2.getY() ) return -1;
     else if(p1.getY() > p2.getY()) return 1; else return 0; }
}
```

A classe `PontoComparator` implementa a interface `Comparator<Ponto>` e ainda a interface `Serializable`, o que é de grande importância também, pois quando uma classe que necessita de um comparador para organizar as suas chaves ou elementos é serializada, o método de comparação deve poder sê-lo também, pois vai ser necessário na desserialização e, portanto, faz parte integrante da classe que dele depende:

```
TreeMap<Ponto, Integer> planoEsp =
  new TreeMap<Ponto, Integer>(new PontoComparator());
```

Apresentam-se a seguir alguns métodos da classe `PlanoPesado` que usa um *treemap* do tipo `<Ponto,Integer>`.

```
public class PlanoPesado {
  private TreeMap<Ponto, Integer> plano;
  // Construtores
  public PlanoPesado() {
```

```
    plano = new TreeMap<Ponto, Integer>(new PontoComparator());
  }
  public PlanoPesado(Comparator<Ponto> cp) {
    plano = new TreeMap<Ponto, Integer>(cp);
  }
  // Métodos de Instância
  public void juntaPonto(Ponto p, int peso) {
    plano.put(p, peso);
  }
  public void remPonto(Ponto p) {
    plano.remove(p);
  }
  public boolean existePonto(Ponto p) {
    return plano.containsKey(p);
  }
  //
  public TreeSet<Ponto> dom() { // cópia ordenada das chaves
    TreeSet<Ponto> paux =
       new TreeSet<Ponto>(new PontoComparator());
    for(Ponto p : plano.keySet()) paux.add(p.clone());
    return paux;
  }
  //
  public HashSet<Ponto> chaves() {
    HashSet<Ponto> paux = new HashSet<Ponto>();
    for(Ponto p : plano.keySet()) paux.add(p.clone());
    return paux;
  }
}
```

A Figura 8.49 apresenta os resultados obtidos depois de se realizar uma iteração sobre as chaves de `TreeMap<Ponto,Integer>`, mostrando que **`PontoComparator`**, o comparador de pontos, realizou a comparação correcta dos pontos inseridos no *treemap*.

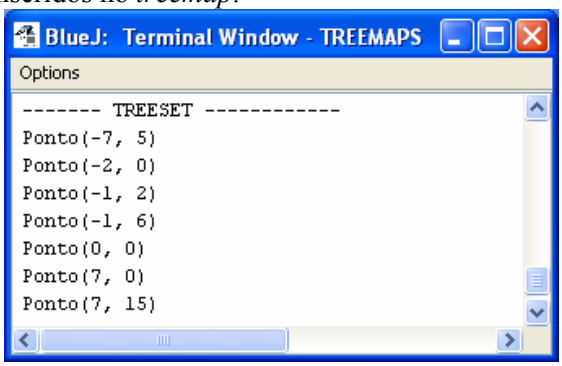


Figura 8.49 – Iteração sobre `TreeSet<Ponto>`

É de notar que no método `dom()`, que deve devolver o conjunto das chaves do *treemap*, tivemos o cuidado de não programar um simples `return plano.keySet();` mas realizar a cópia das chaves (desnecessária) para uma estrutura que nos garantisse que estas pudessem manter a sua ordem tal como existente no *treemap*.

Para tal, criámos um `TreeSet<Ponto>`, apenas sabendo que há um construtor que aceita um comparador (que é o mesmo) e que o método `add()` faz parte da API de todas as colecções monoparâmetro. Esta é uma das grandes vantagens da normalização e unificação de API que a existência das interfaces proporcionam no JCF.

8.14.1 EXEMPLO: INFOMAPA DE PAÍSES

Vamos agora usar os principais métodos de `TreeMap<K,V>` num exemplo concreto um pouco mais complexo, no qual vamos implementar uma classe que nos vai permitir associar o nome de um dado país a valores mais estruturados, designadamente, à sua ficha de informação que irá conter a sua capital, o continente a que pertence e a

sua população, e ainda um *map* que, a cada entrada de tipo `String` contendo uma designação de uma área de interesse, como Turismo, Economia, Desporto, etc., associa uma lista de páginas de texto sobre tal tópico.

Este *map* vai ser representado como um `TreeMap<String,ArrayList<Pagina>>`, em que a classe `Pagina`, que não se apresenta, implementaria uma página de texto editável. A classe que vai conter a informação de cada país vai designar-se por **`InfoPais`**, e terá a seguinte estrutura e métodos:

```
import java.io.Serializable;
import java.util.*;
public class InfoPais implements Serializable, Cloneable {
  // Variáveis de instância
  private String pais;
  private String capital;
  private String continente;
  private double popula; // população em milhões
  private TreeMap<String, ArrayList<Pagina>> infoTemas;
  // Construtores
  public InfoPais(String pais, String cap,
                  String cont, double pop,
                 TreeMap<String, ArrayList<Pagina>> temas) {
    this.pais = pais; capital = cap;
    continente = cont; popula = pop;
    infoTemas = this.copiaTemas(temas);  // método auxiliar
  }
  // Métodos de Instância
  private TreeMap<String, ArrayList<Pagina>>
       copiaTemas(TreeMap<String, ArrayList<Pagina>> temas) { .. }
  public String getPais() { return pais; }
  public String getCapital() { return capital; }
  public String getCont() { return continente; }
  public double getPop() { return popula; }
  public TreeMap<String, ArrayList<Pagina>> getTemas() {
    return this.copiaTemas(infoTemas); }
  public void mudaPop(double nvPop) { popula = nvPop; }
  public String toString() { . . . .}
  public InfoPais clone() {
    return new InfoPais(pais, capital, continente,
                popula, this.copiaTemas(infoTemas));
  }
}
```

O nome do país vai ser a chave de acesso à sua informação, mas vamos também colocá-lo nos valores (nestas fichas) pois é útil nas iterações pelas fichas usando `values()`.

Para realizar uma cópia de um `TreeMap<String,ArrayList<Pagina>>`, foi programado um método auxiliar `copiaTemas()` que é usado no construtor e em `clone()`. O método deve ser bem analisado pois a variável `pagsAux` é delicada:

```
  private TreeMap<String, ArrayList<Pagina>>
      copiaTemas(TreeMap<String, ArrayList<Pagina>> temas) {
    TreeMap<String, ArrayList<Pagina>> temasAux =
       new TreeMap<String, ArrayList<Pagina>>();
    ArrayList<Pagina> pagsAux = null;
    //
    for(String tema : temas.keySet()) {  // para cada tema
      pagsAux = new ArrayList<Pagina>(); // faz clear()
      for(Pagina pag : temas.get(tema))  // copia pags.
       pagsAux.add(pag.clone());
      temasAux.put(tema, pagsAux); // largar este endereço !
    ..// pagsAux.clear() não pode ser feito aqui pois
      // destruiria o ArrayList<Pagina> colocado no treemap.
    }
    return temasAux;
  }
```

O método cria um `TreeMap<>` do tipo do resultado, inicialmente vazio, para onde vai copiar o `TreeMap<>` parâmetro do mesmo tipo. Para tal, é criado um ciclo *foreach* sobre as chaves do *map* parâmetro, e para cada tema, é feito o `get()` do `ArrayList<Pagina>` associado a esse tema. As várias páginas são copiadas usando `clone()` para um *arraylist* de páginas auxiliar. O tema e este *arraylist* são depois inseridos no *map* resultado. Na próxima iteração, o *arraylist* de páginas deve estar de novo vazio, mas não pode ser apagado com `clear()`, pois tal apagaria a lista que acabou de ser introduzida no *treemap*. Assim, a cada nova iteração, tem que se fazer `new ArrayList<Pagina>()`, "largando" o endereço antigo, que está partilhado com o *treemap*, e dando-lhe um novo. Depois de iterados todos os temas, este *map* é devolvido como resultado do método.

A classe principal, **`PaisesMap`** vai ter um *treemap* que vai associar cada nome de país, uma `String`, a uma instância desta classe `InfoPais`. Tal variável `paises` será do tipo:

```
  países = TreeMap<String, InfoPais>;
```

```
import java.util.*;
import java.io.Serializable;
public class PaisesMap implements Serializable {
  // Variáveis de instância
  private TreeMap<String, InfoPais> paises;
  // Construtores
  public PaisesMap() {
    paises = new TreeMap<String, InfoPais>();
  }
  public PaisesMap(TreeMap<String, ? extends InfoPais> infos) {
    paises = new TreeMap<String, InfoPais>();
    for(InfoPais info : infos.values())
      paises.put(info.getPais(), info.clone());
  }
```

O segundo construtor tem por parâmetro `TreeMap<String,? extends InfoPais>` por forma a garantir generalidade e extensibilidade caso mais tarde venham a ser criadas subclasses de `InfoPais`, eventualmente com mais informação sobre cada país. O *map* parâmetro é copiado, par a par, para a variável de instância usando a instrução seguinte:

```
  paises.put(info.getPais(), info.clone());
```

Veja-se de novo a vantagem de termos na ficha de informação de cada país o nome do país (chave). Ao iterarmos pelos valores, temos a chave, **`info.getPais()`**, e temos o valor, **`info`**, directamente acessíveis.

```
  // Métodos de Instância
  // Total de Países
  public int numPaises() { return paises.size(); }
  // Devolve a informação de um país de nome dado
  public InfoPais daInfop(String pais) {
      return paises.get(pais).clone();
  }
```

Neste método, dado o nome do país, que deverá existir, consulta-se o *treemap* para obter a respectiva informação `InfoPais`, cuja cópia é dada como resultado.

```
  // Insere um novo País e seus dados
  public void inserePais(InfoPais infop) {
    paises.put(infop.getPais(), infop.clone());
  }
  // Verifica se o nome de um país existe como chave
  public boolean existePais(String pais) {
    return paises.containsKey(pais);
  }
  // Remove um país e sua informação
  public void removeInfop(String nomeP) {
    paises.remove(nomeP);
  }
  // Altera a população de um país de nome dado
```

```
public void altPopPais(String pais, double nvPop) {
 InfoPais infop = paises.get(pais);
 infop.mudaPop(nvPop);
}

public List<String> listaNomes() {
 ArrayList<String> noms = new ArrayList<String>();
 for(String nome : paises.keySet()) noms.add(nome);
   return noms;
}
```

Este método `listaNomes()` foi codificado como tendo `List<String>` como tipo de resultado. Qual a diferença entre este tipo e `ArrayList<String>` como resultado? Há diferenças para quem programa e há diferenças para quem usa o método.

Quem tem que programar o código do método, sabe que tem que devolver um objecto compatível com o tipo `List<String>`, mas como existem várias implementações de `List<String>`, tal significa que terá várias possibilidades e liberdade de escolha. Neste caso, escolheu-se `ArrayList<String>` mas poderia ter-se escolhido `Vector<String>` ou outra qualquer implementação. O que importa de facto é que são todas compatíveis com `List<String>` pois esta interface é implementada por todas estas classes.

Quem utiliza o método, recebe o resultado e trabalha com ele independentemente da sua real representação que não lhe interessa, a menos que tivesse pedido explicitamente que pretendia ter como resultado uma `LinkedList<String>`, por exemplo. Não o tendo pedido expressamente, poderemos então generalizar e devolver um tipo manipulável via métodos comuns a todas as listas.

Veja-se a seguinte transcrição de código, onde a classe e o método que estamos a apresentar devolvem tal lista, e como podemos trabalhar com tal tipo de objecto de forma absolutamente normal, obviamente respeitando a sua API:

```
PaisesMap pmap = new PaisesMap();
pmap.inserePais(infoP);
  . . . . .;
List<String> nomes = pmap.listaNomes();
for(String p : nomes) out.println(p);
nomes.add("Russia");
nomes.add("Holanda");
for(String p : nomes) out.println(p);
```

Claro que, como fomos nós que programámos, sabemos que o tipo é `List<String>` e que o suporte é um `ArrayList<String>`, mas quando somos nós que recebemos de alguém um objecto especificado como sendo do tipo `List<E>`, `Set<E>` ou mesmo `Collection<E>`, é exactamente nestes termos que devemos pensar e compreender este relacionamento entre **interface** (como tipo e API) e **implementação** (classe de suporte).

Por isso, o método seguinte tem um desafio maior. O resultado pretendido não é sequer uma lista. Pretende-se um método que devolva os nomes dos países com população maior do que o valor dado. Ora este resultado não é, em rigor, uma lista porque não terá nunca nomes em duplicado. Assim, deverá ser rigorosamente um conjunto de nomes de países. Se é um conjunto, é uma implementação de `Set<E>` para *strings*. Se consultássemos neste momento a hierarquia de `Set<E>`, veríamos que poderíamos usar `TreeSet<String>` ou `HashSet<String>`. E não necessitamos de consultar sequer a API dos conjuntos para usar estas classes, porque sabemos que inserir em qualquer colecção é `add()` e iterar é com *foreach*, etc. Assim, criamos um objecto `HashSet<String>` usando o construtor, e fazemos `add()` dos nomes dos países a guardar e depois devolvemos esse objecto:

```
public Set<String> paisesPopMaiorQue(double valor) {
   HashSet<String> noms = new HashSet<String>();
   for(InfoPais infop : paises.values())
     if( infop.getPop() > valor ) noms.add(infop.getPais()) ;
   return noms;
}
```

O código de um programa que invoca o método ficaria agora da seguinte forma:

```
Set<String> paises = pmap.paisesPopMaiorQue(popul);
for(String p : paises) out.println(p);
```

```
paises.remove("xxx"); // etc. etc.
```

Vamos escrever mais alguns métodos usando esta filosofia de generalização dos tipos dos resultados dos métodos e aplicando alguns métodos específicos de `TreeMap<K,V>` :

```
// Devolve uma cópia do hashmap
public Map<String, InfoPais> copia() {
   HashMap<String, InfoPais> mapAux =
                new HashMap<String, InfoPais>();
   for(InfoPais infop : paises.values())
      mapAux.put(infop.getPais(), infop.clone());
   return mapAux;
}
// Conjunto de país de dado continente
public Set<String> paisesCont(String cont) {
   HashSet<String> noms = new HashSet<String>();
   for(InfoPais infop : paises.values()) {
      if( infop.getCont().equals(cont))
          noms.add(infop.getPais()) ;
   }
   return noms;
}
// Infomapa de um país de nome dado
public TreeMap<String, ArrayList<Pagina>>
     infoMapaPais(String pais) {
   return paises.get(pais).getTemas(); }

// Para um dado pais existente, dá as páginas
// correspondentes a um dado tema existente.
public ArrayList<Pagina> pagsTemaPais(String pais,
                                      String tema) {
   return paises.get(pais).getTemas().get(tema);
}
// Para todos os países existentes, devolve um map
// onde a cada país se associa uma lista com as
// páginas existentes sobre um tema dado como parâmetro.
// Se o tema não existir a lista de páginas é vazia.
public Map<String, ArrayList<Pagina>> pagsTema(String tema) {
   TreeMap<String, ArrayList<Pagina>> infoTema =
     new TreeMap<String, ArrayList<Pagina>>();
   TreeMap<String, ArrayList<Pagina>> temas = null;
   ArrayList<Paginas> pags = null;
   for(String pais : paises) {
     pags = new ArrayList<Pagina>();
     temas = paises.get(pais).getTemas();
     if(temas.containsKey(tema))
      pags = temas.get(tema);
     infoTema.put(pais, pags);
   }
   return infoTema;
}
// Devolve um objecto PaisesMap apenas com os países
// de nome alfabeticamente igual ou superior ao nome dado.
public PaisesMap subPaisesMap(String nomInf) {
   TreeMap<String, InfoPais> mapRes =
          new TreeMap<String, InfoPais>();
   SortedMap<String, InfoPais> mp = paises.tailMap(nomInf);
   for(String pais: mp.keySet())
     mapRes.put(pais, mp.get(pais).clone());
   return new PaisesMap(mapRes);
}

public String toString() {
```

```
     StringBuilder s = new StringBuilder();
     s.append("\n---------- PAISES --------------------\n");
     for(InfoPais infop : paises.values()) {
        s.append(infop.toString());
     }
     return s.toString();
   }
} // fim da classe PaisesMap
```

Quando generalizamos os tipos dos resultados dos métodos para as interfaces `List<E>`, `Set<E>` e `Map<K,V>`, não estamos, como vimos, a perder quaisquer propriedades, antes estamos a ganhar em abstracção. Tal também é importante porque nos aproximamos mais da própria forma como as API das classes de JAVA são apresentadas, e as próprias classes codificadas, o que nos leva ao patamar seguinte de abstracção.

Muitos dos métodos das API de JAVA são apresentados como devolvendo como resultado uma `Collection<E>` ou uma `Collection<? extends E>`. Pois bem, em relação ao que vimos atrás, isto apenas quer dizer que quem programa pode devolver ou uma lista ou um conjunto, e quem recebe o resultado da invocação do método não sabe se é lista ou conjunto, e só poderá usar os métodos de `Collection<E>` que, no fundamental, lhe irão permitir manipular suficientemente a colecção devolvida. No limite, poderíamos devolver um `Iterable<E>`, em cujo caso a única operação que poderia ser realizada seria a iteração usando *foreach* que, com `println()`, daria sempre para fazer umas úteis listagens. O resto seria completo segredo (ou abstracção).

8.14.2 MAPS COMPLEXOS

Os *maps* são muitas vezes usados para representar estruturas indexadas multinível, onde, em cada nível, temos apenas um conjunto de chaves de acesso ao nível seguinte, até atingirmos o nível das folhas onde se encontram as fichas de informação ou valores.

Considere-se, por exemplo, um catálogo de automóveis que, no nível 1 contém a marca, no nível 2 contém o modelo, no nível 3 a referência e no nível 4 contém as características e o preço de um veículo. Considere-se também um livro que se encontra dividido em capítulos, subcapítulos e secções, ou uma estrutura de ficheiros onde recorrentemente, a partir da raiz, temos um número ilimitado de níveis, onde em cada nível temos ficheiros e outras ramificações que são directórios.

Todas estas estruturas multinível são bem representadas pelo tipo `Map<K,V>` onde o tipo `V` é sucessivamente substituído por outro *map*, gerando **expressões de tipo** com a forma (usando chaves `String`):

```
Map<String, Map<String, Map<String, Ficha>>>
```

que, por exemplo, serviria para representar o tipo do catálogo de automóveis referido anteriormente. Porém, estamos a explicitar no tipo a estrutura dos objectos, o que só é possível quando a estrutura for finita e conhecida à partida. Tal estrutura é, igualmente, uma estrutura rígida, e não mais pode ser alterada, não permitindo, por exemplo, que sejam inseridos mais níveis durante o tempo de vida da estrutura de objectos.

No caso do catálogo de automóveis, teríamos a declaração:

```
TreeMap<String, TreeMap<String, TreeMap<String, Ficha>>> cat =
  new TreeMap<String, TreeMap<String, TreeMap<String, Ficha>>>();
```

Curiosamente, a declaração é de legibilidade muito mais complexa do que as operações que se podem realizar com esta estrutura. Por exemplo, a determinação do preço de um dado automóvel poderia ser escrita de forma simples como,

```
cat.get(marca).get(modelo).get(ref).getPreco();
```

onde cada `get(String s)` corresponde a descermos um nível na hierarquia do catálogo, até chegarmos à `Ficha` do veículo, da qual extraímos o preço com `getPreco()`. Este código pressupõe que as *strings* `marca`, `modelo` e `ref` são chaves válidas, no conjunto das chaves de cada *map* seleccionado através de cada `get()`.

A modificação do preço de um dado carro seria feita escrevendo uma expressão semelhante:

```
cat.get(marca).get(modelo).put(ref, novopreco);
```

Se pretendermos, de facto, ter uma estrutura mais flexível, com um número de níveis não determinado à partida, e suportada por um conjunto de métodos que nos permitam realizar a sua gestão interna, então teríamos que utilizar uma definição recursiva usando *maps*.

A classe `Catalogo`, que nos permitiria criar, gerir e consultar um catálogo automóvel multinível, deveria então ser definida como:

```
public class Catalogo implements Serializable {
   // Variável de Instância
   private TreeMap<String, Catalogo> cat; // recursividade
   //
```

ou, usando herança a partir de `TreeMap<String, Catalogo>` (recursividade):

```
public class Catalogo extends TreeMap<String, Catalogo>
                      implements Serializable {
   // Construtores
   public Catalogo() {
      super();
   }
   public Catalogo(TreeMap<String, Catalogo> c) {
      super();
      for(String s : c.keySet())
         this.put(s, c.get(s).clone());
   }
```

As classes deverão conter pelo menos um método que determine quando se atinge o nível do último *map*, pois este é de tipo diferente de todos os outros, devendo ser do tipo `TreeMap<String,Ficha>`. Uma variável de instância deste tipo, a `null` ou com um *treemap* não `null`, poderia solucionar o problema, servindo de *flag* indicativa de que foi atingido o penúltimo nível. Outras soluções baseadas na definição de que um `Catalogo` é ou um `TreeMap<String,Catalogo>` ou um `TreeMap<String,Ficha>` poderiam ser encontradas, observando que o tipo `V` do `TreeMap<>` ou é do tipo `Catalogo` ou é do tipo `Ficha`.

Estas soluções são gerais e podem ser adaptadas a qualquer outro tipo parametrizado que se pretenda definir de forma recursiva.

8.15 CONJUNTOS DE TIPO E: Set<E>

Os conjuntos de tipo `E` são colecções que satisfazem a interface **`Set<E>`** e que implementam também a classe abstracta `AbstractSet<E>` (Figura 8.50). Não existindo ordem, os conjntos não possuem, ao contrário das listas, métodos de acesso por índice, mantendo-se disponíveis todos os outros métodos usuais de colecções.

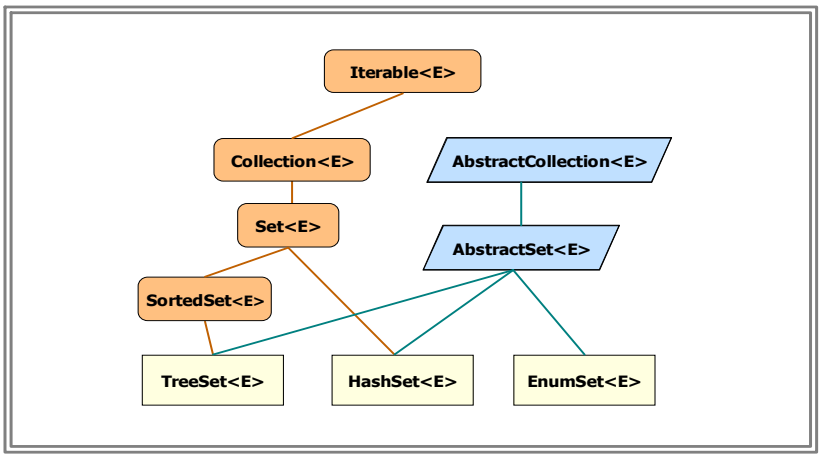


Figura 8.50 – Hierarquia de `Set<E>`

As três implementações de conjuntos existentes são bastante específicas e orientadas a certo tipo de propósitos: um **`HashSet<E>`** é uma implementação geral e eficiente baseada num tabela de hashing; um **`TreeSet<E>`** é uma implementação ordenada usando uma árvore balanceada; e um **`EnumSet<E>`** é uma implementação especial para utilização com tipos enumerados.

Estão disponíveis sobre os conjuntos todas as operações apresentadas sobre listas excepto as que usam índices. Algumas destas operações foram anteriormente utilizadas.

Assim, apresentaremos apenas os métodos que nos permitem implementar as usuais operações da teoria de conjuntos. Sendo `conjA` e `conjB` conjuntos do mesmo tipo `E`, as operações são expressas da seguinte forma:

```
conjA.retainAll(conjB);     // conjA = conjA ∩ conjB
conjA.addAll(conjB);  // conjA = conjA ∪ conjB
conjA.containsAll(B); // conjA ⊇ conjB
conjB.clear();        // ∅
```

Todos os exemplos apresentados utilizando `ArrayList<E>` podem ser convertidos em exemplos com *sets*, substituindo os construtores e eliminando os `get()` e `set()`. Os elementos de um *set* qualquer são lidos usando iteradores (Quadro 8.6):

Categoria de Métodos	API de Set<E>
Inserção de elementos	`add(E o);` `addAll(Collection<? extends E);`

Remoção de elementos	`boolean remove(Object o);`
	`boolean removeAll(Collection<?> c);`
	`boolean retainAll(Collection<?> c)`

Consulta e comparação	`boolean contains(Object o); boolean isEmpty();` `boolean containsAll(Collection<?> c); int size();`
Iteradores	`Iterator<E> iterator();`
Modificação	`void clear();`
Conversão	`Object[] toArray();`
Outros	`boolean equals(Object o);`

Quadro 8.6 – API de `Set<E>`

8.15.1 TreeSet<E>

Um `TreeSet<E>` é uma implementação de conjuntos de elementos de tipo `E` usando uma instância de `TreeMap<E>`. Possui portanto a maior parte das características indicadas anteriormente para a classe `TreeMap<E>`. Os elementos são mantidos segundo a sua ordem natural (o tipo `E` deve implementar a interface `Comparable`) ou segundo um `Comparator<? super E>` dado como parâmetro no respectivo construtor.

A ausência de um algoritmo de comparação para o tipo `E` de um `TreeSet<E>`, gera erro de execução ao invocar-se o método `add(E elem)`.

A sua utilização é, após a sua criação, absolutamente idêntica a todas as que fizemos em exemplos anteriores usando `HashSet<E>`. Os subconjuntos que são os resultados das visões `headSet()`, `tailSet()` e `subSet()`, implementam a interface `SortedSet<E>` e aplicam-se-lhes todas as operações apresentadas no Quadro 8.7, e que foram apresentadas aquando do estudo dos `TreeMap<K, V>`.

TreeSet<E> métodos adicionais
`TreeSet<E>()`
`TreeSet<E>(Comparator<? super E> c)`
`TreeSet<E>(Collection<? extends E c)`
`TreeSet<E>(SortedSet<E> s)`
`E first()`
`SortedSet<E> headSet(E toElement)`
`E last()`
`SortedSet<E> subSet(E fromElem, E toKey)`
`SortedSet<E> tailSet(E fromElem)`

Quadro 8.7 – API parcial de `TreeSet<E>`

A criação de um `TreeSet<E>` apenas tem um caso especial quando pretendemos passar como parâmetro do construtor um `Comparator<? extends E>`. Tal como vimos antes, tal passará por criarmos uma classe que implemente o comparador adequado e passar uma instância dessa classe como parâmetro, tal como em:

```
TreeSet<Ponto> paux =
     new TreeSet<Ponto>(new PontoComparator());
```

Tal como para outras colecções, o método `equals()` deve ser coerente com qualquer método de comparação definido.

8.16 FILAS DE TIPO E: Queue<E>

A interface `Queue<E>`, nova em JAVA5, define um pequeno protocolo para colecções que guardam objectos sequencialmente, para serem, tipicamente, processados por processos concorrentes. Sendo filas (estruturas FIFO, *first-in-first-out*), podem organizar os seus elementos de formas diferentes, podendo mesmo ser *stacks* (LIFO, *last-in-first-out*).

O protocolo é formado por cinco métodos: `element()`, que remove o primeiro da fila, `peek()`, que consulta o primeiro da fila, `offer(E o)`, que tenta inserir o objecto na fila, e `poll()`e `remove()` que, respectivamente, devolvem e retiram o primeiro da fila.

Para além da classe `LinkedList<E>`, as classes que implementam o tipo `Queue<E>` são `PriorityQueue<E>`, `BlockingQueue<E>` e outras subclasses destas, que são usadas em programação concorrente.

8.17 TIPOS ENUMERADOS

Um **tipo enumerado** é um tipo de dados cujos valores são um *conjunto* pequeno e fixo de constantes definidas pelo programador, e que são representadas de forma literal.

São exemplos típicos de **enumerações** os pontos cardeais {Norte, Sul, Este, Oeste}, os dias da semana {Dom, Seg, Ter, …}, os estados de um semáforo {Verde, Amarelo, Vermelho, AmInterm}, o tipo de café que pedimos {Curto, Normal, Cheio}, as cores dos tinteiros de uma impressora {Black, Cyan, Magenta, Yellow}, os naipes de um baralho de cartas {Paus, Espadas, Ouros, Copas}, o estado de uma lâmpada {On, Off}, etc.

Algumas linguagens fornecem construções que permitem declarar tipos que representam pequenos conjuntos de valores identificados, em geral chamados **tipos enumerados**. Um tipo enumerado é um conjunto definido em extensão (daí ser pequeno em geral), exactamente pela apresentação (enumeração) dos seus valores válidos. Uma variável de um tipo enumerado contém em cada momento um e um só destes valores.

A primeira linguagem a introduzir tais tipos foi o Pascal, à qual se seguiram praticamente todas as outras (Modula-2, Ada, C, C++, C#), à excepção de JAVA, até aqui. Em JAVA5 foi introduzida uma classe/tipo **`enum`** própria para representar enumerações seguras e simples, fornecendo um conjunto interessante de operações sobre os valores das mesmas.

Como até JAVA5 as enumerações não eram directamente suportadas pela linguagem, várias técnicas usando outras construções eram usadas pelos programadores procurando implementar de forma eficiente e o mais segura possível as enumerações. Apresentam-se em seguida duas dessas técnicas, naturalmente não para que sejam usadas, mas porque a análise das mesmas em muito facilitará a compreensão das novas classes **`enum`**.

Uma das técnicas usadas para implementar enumerações consiste em criar uma **interface**, nesta definir um conjunto de identificadores de constantes (que são `public static final` por omissão) e atribuir a cada um destes identificadores um valor de dado tipo.

Vejamos um exemplo desta técnica e a sua utilização num programa:

```
public interface Direccao {
   // são public static final
   int Norte = 1;
   int Sul = 2;
   int Este = 3;
   int Oeste = 4;
}
```

O programa seguinte vai testar esta enumeração. Define um método auxiliar `static` para mostrar a direcção actual, método que tem como parâmetro a variável `paraOndeVamos` que tem que ser do tipo `int`:

```
import static java.lang.System.out;
public class TesteEnum1 {
   //
   public static void mostraDireccao(int paraOndeVamos) {
      out.println("Vamos para " + paraOndeVamos);
   }
   //
   public static void main(String[] args) {
      int vamosPara = Direccao.Norte;
      int nordeste = 6;
      mostraDireccao(vamosPara);
      mostraDireccao(nordeste);
   }
}
```

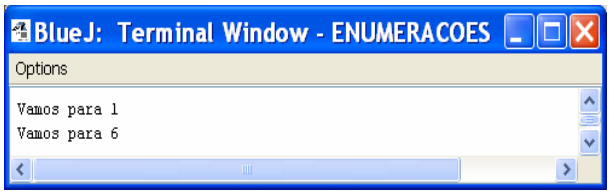


Figura 8.51 – Resultados do programa `TesteEnum1`

Para se poder codificar o método foi necessário "saber" que cada "direcção" é representada por um número inteiro, isto é, ao contrário de todos os princípios, tivemos que ter acesso à representação e isso é muito mau sinal quando se pretende abstracção.

No método `main()`, invocámos este método com o valor de `nordeste` (cf. 6), fora da gama de direcções válidas portanto, e, naturalmente, o valor é aceite pelo compilador. Não existe qualquer segurança de tipos de dados (não é *type safe*).

Para o compilador, um valor do tipo `Direccao` e um valor do tipo `int` são compatíveis. Podemos pois passar um `int` onde for pedida uma `Direccao` que o compilador não tem objecções. Quando muito, poderemos escrever métodos que, usando um `switch`, façam a validação dos valores inteiros mas só em tempo de execução. Até fará algum sentido usar a estrutura `switch`, dado que indicar a direcção através de um número inteiro não tem qualquer utilidade, justificando-se a conversão para uma *string*. Mas, como se torna óbvio, esta técnica é muito pouco segura e muito pobre em termos de solução:

```
  public static void mostraDireccao(int paraOndeVamos) {
    switch(paraOndeVamos) {
      case 1 : out.println("Vamos para Norte."); break;
      case 2 : out.println("Vamos para Sul."); break;
      case 3 : out.println("Vamos para Este."); break;
      case 4 : out.println("Vamos para Oeste."); break;
      default : out.println("Direccao Errada !");
    }
  }
```

Uma técnica segura em termos de tipos e de nomes de valores em espaços de variáveis é a que se baseia na criação de uma classe em que **cada um dos valores a enumerar é uma instância dessa mesma classe** que é guardada numa variável de classe `final`, tal como se exemplifica no código seguinte:

```
public class Direccao {
  // Variável de Instância
  private String nomeDir; // guarda a string
  // Construtor
  private Direccao(String dir) { nomeDir = dir; }
  // Método de Instância
  public String mostraDir() { return nomeDir; }
  // Constantes de Classe
  public static final Direccao Norte = new Direccao("Norte");
  public static final Direccao Sul = new Direccao("Sul");
  public static final Direccao Este = new Direccao("Este");
  public static final Direccao Oeste = new Direccao("Oeste");
}
```

A classe tem uma variável de instância onde é guardada uma *string* que irá conter o nome da direcção em texto. Note-se que o construtor da classe `Direccao` é `private`, logo não pode ser usado fora da classe, pelo que a classe não vai poder criar instâncias. Assim, as suas únicas instâncias apenas existem dentro da própria classe, mas podem (e essa é a ideia!) ser referenciadas externamente pelos seus ***nomes***, ex.: `Direccao.Norte`, etc. Finalmente, sendo instâncias de `Direccao`, respondem à mensagem `mostraDir()`, bem como a todas as outras que tivéssemos programado. Vejamos um programa que usa esta classe, e vejamos porque é esta técnica segura:

```
import static java.lang.System.out;
public class TesteEnum2 {
  public static void mostraDireccao(Direccao paraOndeVamos) {
    out.println("Vamos para " + paraOndeVamos.mostraDir());
  }
  public static void main(String[] args) {
    Direccao vamosPara = Direccao.Norte;
    // Direccao nordeste = "Oriente"; ERROS DE COMPILAÇÃO
    // Direccao nordeste = new Direccao("Oriente");
    mostraDireccao(vamosPara);
    vamosPara = Direccao.Sul;
    out.println("e agora para .. " + vamosPara.mostraDir());
  }
}
```

Note-se que agora o compilador conhece "bem" o tipo `Direccao` e não permite qualquer má utilização do mesmo, quer por erro de tipo quer por acesso a construtor privado. O método `mostraDireccao()` manteve-se apenas para comparação pois não seria necessário. Porém, o seu parâmetro é agora "fechado", do tipo `Direccao`. O método de instância `mostraDir()` é um conversor para *string* do identificador da constante, operação que é aliás usual definir em associação com enumerações. Os resultados apresentam-se na Figura 8.52:

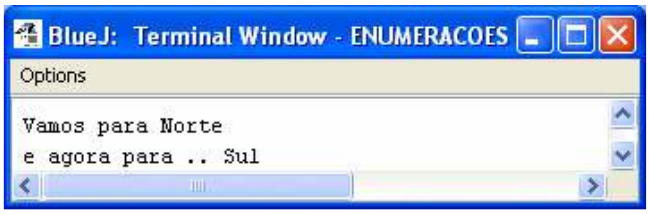


Figura 8.52 – Resultados do programa `TesteEnum2`

No entanto, muitos outros métodos teriam que ser programados para que esta solução pudesse ser completamente versátil. Teríamos que implementar formas de iterar as constantes, conversões automáticas de identificadores para *string* e vice-versa, comparadores das constantes, etc.

Os novos tipos enumerados de JAVA5, que vamos apresentar a seguir, trazem consigo quer a segurança de tipos quer um interessante conjunto de métodos predefinidos para a sua manipulação, bem como classes de apoio que foram desenvolvidas especificamente para funcionarem com tipos enumerados (como `EnumSet<E>` e `EnumMap<E>`).

8.17.1 TIPOS ENUMERADOS EM JAVA5

Em JAVA5, um **tipo enumerado** é definido através de uma **classe particular** identificada pela palavra reservada **`enum`**, no corpo da qual, separados por vírgulas, se declaram os identificadores das designadas **constantes da enumeração**. Na sua forma mais simples, a declaração é semelhante às declarações de tipos enumerados de outras linguagens, sendo no entanto desde já de reforçar a ideia de que, em JAVA5, **uma enumeração é uma classe**, que passaremos a designar apenas por `enum`.

Na sua forma mais simples, a declaração de um `enum` tem em JAVA a seguinte sintaxe:

```
public enum Direccao { Norte, Sul, Este, Oeste }
```

ou, usando um padrão mais semelhante com o de uma classe:

```
public enum Direccao {
   Norte, Sul, Este, Oeste
}
```

Um tipo enumerado não cria instâncias, o que não significa que não possa ter construtores (tal como as classes abstractas), e apenas possui as suas constantes, que são implementadas como se fossem constantes de classe (`public static final`), e que, como tal, podem ser usadas através de expressões do tipo **`<IdClasse_Enum>.<Id_Constante>`**, tal como, por exemplo, `Direccao.Norte`.

Sendo uma construção nova de JAVA5, uma questão relacionada com os identificadores das constantes se coloca: deverá escrever-se `Direccao.NORTE`, `Direccao.Norte` ou `Direccao.norte`, isto é, qual a convenção para os identificadores de constantes de tipos enumerados? A expressão `Direccao.NORTE` passaria a ser de semântica dúbia. `NORTE` pode ser uma constante de classe ou uma constante do enumerado. As outras duas não oferecem este problema, pelo que usaremos a regra de iniciar o identificador de constante por letra maiúscula, seguida de letras minúsculas ou dígitos, como `Direccao.Norte`. Assim, todos os exemplos apresentados atrás obedecem a esta nossa convenção para tais identificadores (alguns autores preferem usar apenas letras minúsculas).

Para tais exemplos, teríamos então definições de enumerados e declarações, tais como:

```
public enum Luz { On, Off }
public enum Cafe { Curto, Normal, Cheio }
public enum Dia { Dom, Seg, Ter, Qua, Qui, Sex, Sab }
// utilização
Luz sala = Luz.On; Cafe meu = Cafe.Normal;
Dia hoje = Dia.Sex;
```

Obviamente que os valores errados são de imediato detectados pelo compilador.

8.17.2 MÉTODOS PREDEFINIDOS

Qualquer `enum` definido pelo utilizador é subclasse de `java.lang.Enum`, pelo que basta criar o tipo enumerado, tal como fizemos acima, para que imediatamente, sobre os seus valores, esteja predefinido um conjunto de métodos. Vamos através de um programa estudar e usar tais métodos que se aplicam a qualquer `enum`.

```
import static java.lang.System.out;
public class TstDireccao1 {
  public static void main(String[] args) {
     Direccao vamosPara = Direccao.Norte;
     out.println("Ordem = " + vamosPara.ordinal());
     String str = "Virar para " + vamosPara.name();
     out.println(str);
     out.println("Direccao = " + vamosPara);
     out.println("Sul: " + Direccao.valueOf("Sul").ordinal());
     // Todas as direcções (como strings !)
     for(Direccao dir : Direccao.values())
       out.println("Podemos ir para : " + dir);
  } }
```

Os valores de um `enum` possuem uma ordem natural implícita que é a própria ordem da sua apresentação na definição do tipo, sendo ao primeiro associado o ordinal 0 e assim sucessivamente. O método **`ordinal()`** devolve o inteiro correspondente à ordem de uma constante. Assim, no programa exemplo, definimos a variável `vamosPara` como sendo do tipo `Direccao`, demos-lhe um valor (`Norte`), e, em seguida, realizámos um conjunto de operações com a mesma sem termos programado nenhuma. Enviámos a mensagem `ordinal()` que deu como resultado 0. A variável contém a primeira constante da enumeração. Em seguida, enviámos a mensagem `name()` e concatenámos o resultado desta mensagem com a *string* à esquerda. O método **`name()`** aplicado a uma constante de um `enum` converte-a na *string* equivalente à sua forma literal. O método **`toString()`** está também definido e é equivalente, estando o método `toString()` implícito ao ser invocado na linha onde escrevemos `println( .. + vamosPara)` (Figura 8.53).

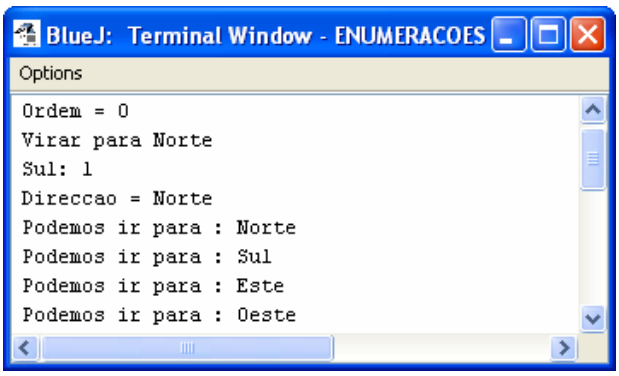


Figura 8.53 – Teste de métodos sobre enumerações

O método **`valueOf()`** é de certa forma o inverso, mas é um **método de classe**. Dada uma *string* que deverá ser a representação válida de uma das constantes do tipo, o método devolve essa *constante*. No exemplo, "Sul" é convertida em `Sul`, cujo `ordinal()` dá como resultado 1. Este método `valueOf()` poderá ser útil na leitura de valores do tipo a partir de *strings*, fazendo-se de imediato a respectiva validação.

Em seguida, mostra-se a aplicação conjunta do iterador **`for`** com o método de classe **`values()`**. O método `values()` quando aplicado a uma classe `enum`, devolve um *array* com as constantes dessa classe, que podem portanto ser iteradas com um iterador *foreach*, tal como no exemplo: `for(Direccao dir : Direccao.values()) { … }`.

Outras formas de iterar os valores de `Direccao` (e de qualquer outro `enum`) seriam:

```
  Direccao[] dirs = Direccao.values();
  for(int dir = Direccao.Norte.ordinal();
     dir<=Direccao.Oeste.ordinal(); dir++)
          out.println("Dir: " + dirs[dir]);
  // ou
```

```
  for(Direccao dir : dirs)
   out.printf("%s %d letras%n", dir.name(), dir.name().length());
```

A existência de um ordem natural e de um método `ordinal()` seria suficiente para que pudéssemos comparar os valores de uma enumeração. Existe, no entanto, definido para qualquer `enum` um método de instância que faz tal comparação directamente, que é o método `compareTo()`. A expressão `ve1.compareTo(ve2)` dá como resultado -1 se o valor de enumeração `ve1` for menor do que `ve2`, 1 se for maior e 0 se forem iguais. O método `equals()`, bem como o operador de igualdade ==, podem ser usados para verificar se dois valores de uma enumeração são iguais.

Vamos agora mudar de exemplo de enumeração considerando a seguinte:

```
  public enum Cafe { Curto, Normal, Cheio }
```

e apresentar um programa que realize a leitura validada de valores da enumeração e faça a sua atribuição correcta, tal como é usual fazer-se nas aplicações:

```
import java.util.Scanner;
import static java.lang.System.out;
public class TstCafe1 {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    int opcao = 0;
    Cafe meuCafe = null;
    out.println("-----  ESCOLHAS ----");
    Cafe[] cafes = Cafe.values();
    for(Cafe c : cafes)
     out.printf("%d - %s%n", c.ordinal()+1, c.name());
    do {
       out.print("Como quer o seu Cafe ? ");
       opcao = input.nextInt();
    }
    while(opcao<1 || opcao>cafes.length);
    meuCafe = cafes[opcao-1];
    out.println("\nOh Ze, tira um cafe " + meuCafe);
  }
}
```

Assim, o programa começa por inicializar as variáveis necessárias, cria uma variável do tipo `Cafe` inicializada a `null`, pois não há construtores para estas variáveis de `enum`, e determina o *array* de valores (cf. `Cafe[]`).

Em seguida, é apresentado o pequeno menu ao utilizador no qual a cada nome de constante é associado o seu ordinal + 1 como opção (como 1, 2 e 3). O ciclo *do-while* espera um inteiro entre 1 e 3 (o programa obriga-o mesmo a tomar café!). Depois de ser lida uma opção válida, seleccionamos no *array* de valores a constante que lhe corresponde, que atribuímos depois à variável `meuCafe`.

Os resultados da execução deste programa são apresentados na Figura 8.54, sendo de salientar o facto de que praticamente toda a componente de interacção com o utilizador, ainda que simples, foi automaticamente gerada usando alguns dos métodos predefinidos para os tipos enumerados.

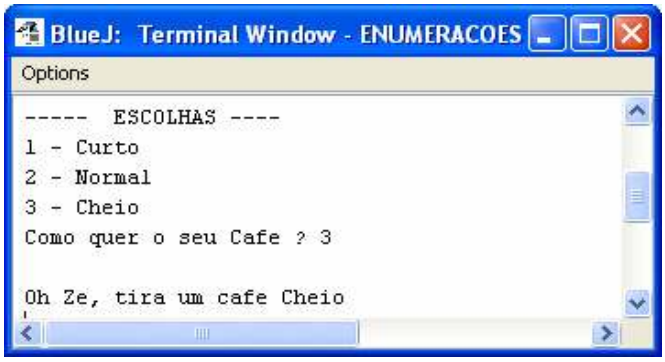


Figura 8.54 – Resultados de `TstCafe1`

Poderíamos agora pensar em alternativas. Uma delas seria deixar o utilizador introduzir a *string* que corresponde ao nome da constante da enumeração. O utilizador introduzia, por exemplo, a *string* "Médio", que o programa guardava na variável `tipoCaf`, e agora bastaria escrever no programa `meuCafe = Cafe.valueOf(tipoCaf);`. No entanto, o problema está na validação da *string* dada pelo utilizador. "Medio" estaria correcta, mas "médio", "medio", "MEdio", etc. não, o que obrigaria a validações complexas.

Assim, e dado que os tipos `enum` não terão nunca grande número de constantes, a melhor solução para o utilizador atribuir um valor de um `enum` a uma variável será através da selecção de uma opção de fácil leitura e validação, a partir da qual se determina a constante. No exemplo anterior, poderíamos também ter usado um `switch(opcao)` que, para cada caso, gerasse a *string* do nome da constante a usar no `valueOf()`.

Porém, como já dissemos várias vezes, usar `switch` em PPO é mau sinal. Um `switch` é uma estrutura estática. Quando é compilada para três casos diferentes apenas funciona para esses casos. No nosso exemplo, o programa não usa `switch` para saber os valores válidos de `Cafe` mas sim `values()`, que é um método dinâmico, de tempo de execução.

Assim, e propositadamente, acaba-se de modificar a enumeração `Cafe` para:

```
public enum Cafe { Vazio, Curto, Normal, Cheio, XCheio }
```

tendo inserido um elemento antes de todos e um depois de todos os que existiam, recompilando-se o programa sem alterar o código.

Imagine-se o que teria acontecido ao programa que usa `switch` para as opções 1, 2 e 3. Pois bem, não se modificou uma vírgula no programa anterior e vai ser agora executado. O resultado é o que se apresenta na Figura 8.55.

Repare-se que a ordem do último elemento da primeira enumeração `Cafe` poderia ter sido escrita `Cafe.Cheio.ordinal()` que dava 2, mas não era genérica. Para uma qualquer enumeração, o seu último valor deverá ser sempre o que tem por ordinal o *length-1* do *array* dos seus valores, tal como devolvido pelo método `values()`.

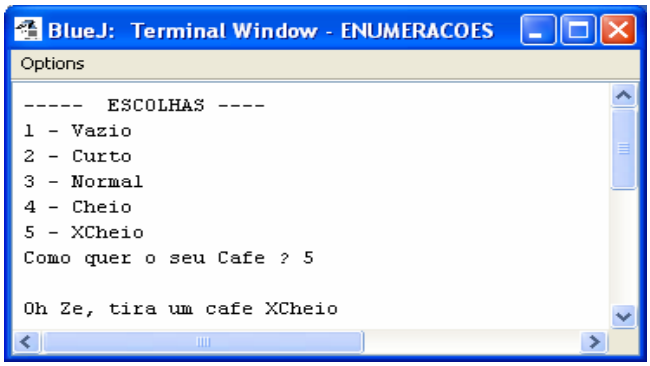


Figura 8.55 – Teste de métodos sobre enumerações

Finalmente, a estrutura `switch` passou em JAVA5 a admitir como expressão de decisão constantes de tipos `enum`, e aqui sim, se, de momento, quiséssemos saber o valor a pagar pelo café, caso fossem diferentes para cada caso, poderíamos usar um `switch`, ou, então, juntar à representação um *array* de preços `double[]`, com índices "sincronizados" com o *array* das constantes. Mais adiante, veremos um melhor solução usando `EnumMap<E>`.

8.17.3 EnumSet<E>

A classe `EnumSet<E>` do *package* `java.util` foi desenvolvida em JAVA5 para criar implementações especiais de conjuntos (`Set<E>`) que trabalham com elementos de tipos enumerados, com a única restrição de que tais elementos devem ser todos provenientes do mesmo tipo `enum` quando o `EnumSet<E>` é criado. Um `EnumSet<E>` é, portanto, um conjunto de elementos de um **mesmo tipo enumerado**.

A classe `EnumSet<E>` não tem construtores públicos, mas oferece um grande número de métodos de classe que nos permitem criar instâncias. Vamos ver os métodos mais úteis, de classe e de instância, continuando a tomar por exemplo o tipo enumerado `Cafe`.

A criação em extensão, é realizada enumerando os elementos do tipo `enum`, como em:

```
EnumSet<Cafe> cafes1 = EnumSet.of(Cafe.Cheio);
EnumSet<Cafe> cafes2 = EnumSet.of(Cafe.Normal, Cafe.Vazio);
EnumSet<Cafe> cafes3 = EnumSet.of(<1º, ...resto>);
```

A criação pode ser feita também usando todos os elementos de um dado enumerado cuja classe é indicada, ou a partir de uma instância de um conjunto enumerado:

```
EnumSet<Cafe> todos = EnumSet.allOf(Cafe.class);
EnumSet<Cafe> cafes5 = EnumSet.allOf(cafes3);
```

A criação usando apenas uma subgama dos elementos de um conjunto, faz-se escrevendo:

```
EnumSet<Cafe> cafes5 = EnumSet.range(Cafe.Vazio, Cafe.Normal);
```

Depois de termos instâncias de `EnumSet<E>`, é então possível realizar várias operações com as mesmas, operações típicas da teoria de conjuntos:

```
// Diferença de conjuntos  todos – cafes2
EnumSet<Cafe> dif = todos.complementoOf(cafes2);
for(Cafe c : dif) out.printf("Café %s ", c);
// Reunião
EnumSet<Cafe> temp = EnumSet.noneOf(Cafe.class); // [] vazio
temp.addAll(cafes1); temp.addAll(cafes2);
// Intersecção
temp.retainAll(cafes3);
for(Cafe c : temp) out.printf("Café %s ", c);
```

O programa seguinte usa todas estas operações num pequeno exemplo concreto de dois estabelecimentos que vendem café, e que, através das várias operações com enumerados e conjuntos de enumerados procuram estabelecer as suas afinidades e diferenças:

```
import java.util.EnumSet;
import static java.lang.System.out;
// Cafe = { Vazio, Curto, Normal, Cheio, XCheio )
public class TstEnumSet1 {
 //
 public static void main(String[] args) {
  EnumSet<Cafe> todos = EnumSet.allOf(Cafe.class);
  EnumSet<Cafe> meuCafe = EnumSet.range(Cafe.Curto, Cafe.Cheio);
  EnumSet<Cafe> teuCafe = EnumSet.range(Cafe.Vazio, Cafe.Normal);
  EnumSet<Cafe> dif = todos.complementOf(meuCafe);
  // Operações de comparação
  out.println("---- EU SIRVO ----");
  for(Cafe c : meuCafe) out.printf("Café %s ", c);
  out.println("\n---- MAS NAO SIRVO ----");
  for(Cafe c : dif) out.printf("Café %s ", c);
  out.println("\n------------------");
  out.printf("Tu serves %s ? %s%n", Cafe.Curto.name(),
      teuCafe.contains(Cafe.Curto) ? "SIM!":"NÃO!");
  out.printf("Serves %s ? %s%n", Cafe.XCheio.name(),
      teuCafe.contains(Cafe.XCheio) ? "SIM!":"NÃO!");   // ? :
  out.println("Então serves : ");
  for(Cafe c : teuCafe) out.printf(" %s ", c);
  dif.clear(); // limpa conjunto
  // Diferença (meus – teus)
  dif = EnumSet.copyOf(meuCafe); dif.removeAll(teuCafe);
  out.println("\n---- EU SIRVO ----");
  for(Cafe c : dif) out.printf("%s ", c);
```

```
    out.println("\n---- E TU NÃO !! ----\n");
    // Reunião
    EnumSet<Cafe> temp = EnumSet.noneOf(Cafe.class); // []

    temp.addAll(meuCafe);
    temp.addAll(teuCafe);
    out.println("-- JUNTOS ---");
    for(Cafe c : temp)
      out.printf("Café %s%n", c);
    // Intersecção
    temp.clear();
    temp = EnumSet.copyOf(meuCafe);
    temp.retainAll(teuCafe);
    out.println("-- IGUAIS ---");
    for(Cafe c : temp)
      out.printf("Café %s%n", c);
  }
}
```

O ecrã que se apresenta lateralmente (Figura 8.56) apresenta os resultados da execução do programa anterior. De salientar que alguns dos métodos usados no programa são métodos de `Set<E>`, e, por isso, são métodos de instância. A maioria dos métodos de `EnumSet<E>` são, no entanto, métodos de classe.

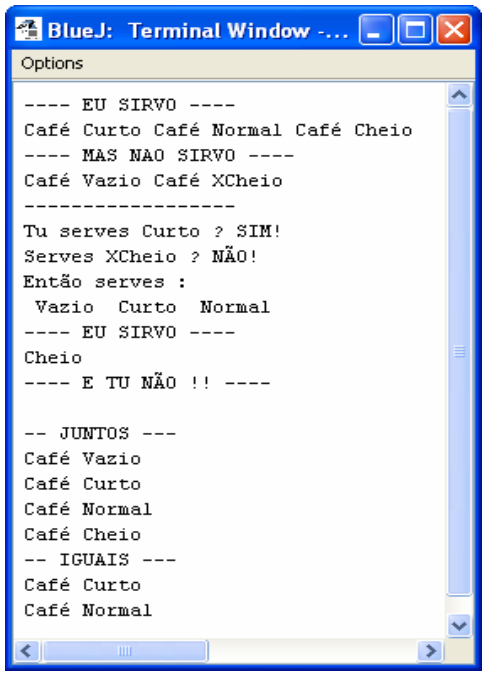


Figura 8.56 - Resultados

Agora que já temos um enorme conjunto de operações que podem ser realizadas com os identificadores de um tipo enumerado, sabendo até, mais ou menos, como é que estes tipos são internamente implementados (vimos uma técnica *type safe* que até é extensível), apetecia perguntar se, em certas circunstâncias, não faria sentido associar às constantes da enumeração informação adicional. Por exemplo, e até para terminar os exemplos com os cafés, fazia sentido (que não na vida real) associar um preço a cada "tipo" de café, ou uma distância a cada direcção, ou um número de dias a cada mês, etc. Isto é, parece de facto ser útil e interessante ter a possibilidade de associar um ou mais valores a cada um dos identificadores (ou constantes) de uma enumeração e, em consequência, possuir os meios necessários para os manipular (no mínimo, consultar).

Os `EnumSet<E>` em que `E` é um tipo enumerado, são também de grande utilidade na validação de dados introduzidos pelo utilizador e de parâmetros de métodos, dado que cada constante possui a sua *string* equivalente, e porque, a partir de uma *string*, podemos obter a constante da enumeração.

Esta compatibilidade entre os nomes das constantes e as *strings* pode ser útil em diversas ocasiões, especialmente tendo em atenção que o valor de uma variável de um tipo enumerado acabará sempre por resultar da conversão de um valor dado pelo utilizador.

Vamos em seguida ver mais algumas propriedades interessantes dos tipos enumerados de JAVA5.

8.17.4 ATRIBUTOS E MÉTODOS EM ENUMERADOS

A forma de associarmos um ou mais valores (um **mini-estado**) a cada identificador de uma enumeração é em tudo idêntica à forma como definimos o estado de um objecto, ainda que com sintaxes diferentes. Vejamos um exemplo:

```
public enum TShirt {
  S(15.0),
  M(17.5),
  L(20.0),
  XL(22.0),
  XXL(25.0);  // ; de separação
```

```
  // Variável interna
  private double preco;
  // Construtor e método
  private TShirt(double preco)
    { this.preco = preco; }
  public double getPreco() { return preco; };
}
```

Temos, então, que cada constante da enumeração `TShirt` tem a si associado um `double` que, tal como podemos ver pelo construtor `private`, e pela variável `private`, são o seu preço. A diferença entre este código e o código apresentado como técnica segura, é que, neste caso, cada constante é declarada com o valor que é o seu respectivo parâmetro no construtor `TShirt()` (ex.: `XXL = new Tshirt(25.0)`). É acrescentado o método `getPreco()`, que permite consultar o preço de qualquer *t-shirt* dos tamanhos existentes. A variável de instância e o método definidos são estado e método de **cada constante**.

Um *foreach* que permitiria ver todos os tamanhos e preços respectivos do tipo enumerado `TShirt` seria escrito num programa ou método, simplesmente como:

```
for(TShirt t : TShirt)
  out.printf("Tamanho %s Preço %4.1f%n,t, t.getPreco());
```

Poderíamos também acrescentar um método capaz de alterar o preço de um elemento:

```
public void mudaPreco(double nvPreco) {
  preço = nvPreco; }
```

a ser usado da forma `TShirt.XL.mudaPreco(23.5);`. Naturalmente que alterações que sejam feitas aos valores associados aos identificadores através deste método são voláteis, ou seja, só têm validade no contexto do programa em que foram realizadas, não implicando mudança no tipo `enum`.

Vamos agora criar uma encomenda de *t-shirts* de vários tamanhos e calcular qual o valor total da mesma. Uma encomenda poderá ser representada por um `HashMap<K,V>`, em que as chaves são do tipo `Tshirt` e os valores do tipo `Integer`.

Teríamos, então:

```
HashMap<TShirt, Integer> encom =
          new HashMap<TShirt, Integer>();
encom.put(TShirt.S, 100);         // criação da encomenda
encom.put(TShirt.M, 150);
encom.put(TShirt.L, 50);
```

O cálculo do valor da encomenda processa-se agora iterando as chaves do *hashmap*, e, para cada tamanho de *t-shirt* (chave), multiplicando a quantidade encomendada (valor) desse tamanho pelo respectivo preço.

```
// Valor da encomenda
double valor = 0.0;
for(TShirt ts : encomenda.keySet())
  valor += encomenda.get(ts)*ts.getPreco();
out.printf("Valor total = %7.2f%n", valor);
```

Consideremos que temos agora um enumerado `Cliente` especificando vários tipos de cliente, e contendo, associados a cada constante, o desconto, percentagem máxima de valor de crédito e código, tal como se apresenta no código seguinte:

```
public enum Cliente {
// enumeração
  Normal(0.0 0.0 "N1"),
  Empresa (0.1 0.2 "E1"),
  Export(0.2 0.25 "X1"),
  VIP(0.3 0.5 "V1"),
  Amigo(0.5 0.8 "A1"), ;
```

```
  // construtores privados da classe
  private Cliente() {
    desconto = 0.0; maxCredito = 0.0; codigo = "N1";
  }
  private Cliente(double td, double tc, String cd) {
    desconto = td; maxCredito = tc; codigo = cd;
  }
  // variáveis de estado
  private double desconto;          // em %
  private double maxCredito;        // em %
  private String codigo;            // código interno
  // consulta do estado de cada identificador
  public double getDesconto() { return desconto; }
  public double getmaxCredito() { return maxCredito; }
  public String getCodigo() { return codigo; }
  // métodos para mudança de estado
  public void xDesconto(double nvDesconto) {
       desconto = nvDesconto; }
  public void xCredito(double nvCredito) {
       maxCredito = nvCredito; }
}
```

Tendo esta informação completa de tipo de cliente e uma encomenda, podemos agora determinar o valor total da encomenda num método como o que se apresenta a seguir, que recebe um cliente e uma encomenda e calcula o valor total da factura em função do preço associado a cada tamanho de *t-shirt*.

```
  public double calcFactura(Cliente c,
                            HashMap<TShirt, Integer> encom) {
  // Valor da encomenda
    double valor = 0.0;
    for(TShirt ts : encomenda.keySet())
      valor += encomenda.get(ts)*ts.getPreco();
    return valor*(1.0 - c.getDesconto());
  }
```

Para além de podermos associar valores a cada constante de um tipo enumerado e definir métodos `public` associados ao tipo, é ainda possível associar a cada constante um **corpo de classe**, ou seja, uma classe anónima onde são definidos métodos de instância `public` que vão ficar associados a essa constante. Esses métodos redefinem os equivalentes **métodos abstractos** declarados no corpo da definição do tipo enumerado:

```
public enum TShirtA {
  S(15.0) {
            public double prcAmigo() { return getPreco()*0.8; }
            public double getPreco() { return 14.85; }
          },
  M(17.5) . . ., // aqui código de prcAmigo()
  L(20.0) . . ., // para cada constante
  XL(22.0) . . .,
  XXL(25.0) . . .;
  //
  private double preco;
  //
  private TShirtA(double preco)
    { this.preco = preco; }
  public double getPreco() { return preco; };
  abstract double prcAmigo();
}
```

Como o método `prcAmigo()` é `abstract` e o tipo enumerado cria as suas próprias instâncias que são as constantes da enumeração, então, cada uma delas deverá fornecer uma implementação para o método, caso contrário, a classe dá erro de compilação. De notar que imediatamente a seguir ao identificador da constante e ao parâmetro do construtor, começa um bloco **{ .. }** que apenas termina no separador de constantes (`,`). Neste bloco podemos escrever o código de outros métodos. No exemplo, escrevemos o código de um método que redefine o

método `getPreco()` de `TShirt` para o tamanho `S`. O método usa `getPreco()` para saber o preço actual e desconta 20%.

Para sabermos quais os preços para amigos praticados sobre as *t-shirts*, teremos que as enumerar e usar o método implementado por cada uma, conforme o seguinte código:

```
for(TShirtA t : TShirtA.values())
  out.printf("Tamanho %s Preço %4.1f%n",t, t.prcAmigo());
```

O exemplo clássico desta última facilidade dos tipos enumerados, mas sem grande utilidade prática, é o exemplo da "Calculadora Enumerada", cujo código é o seguinte:

```
public enum Operacao {
  MAIS  { double calc(double x, double y) { return x+y; } },
  MENOS { double calc(double x, double y) { return x-y; } },
  VEZES { double calc(double x, double y) { return x*y; } },
  DIV   { double calc(double x, double y) { return x/y; } };
  //
  abstract calc(double x, double y);
}
```

que pode ser testada usando o seguinte programa simples:

```
int v1, v2;
out.print("1º valor: "); v1 = input.nextInt();
out.print("2º valor: "); v2 = input.nextInt();
for(Operacao op : Operacao.values())
 out.printf("%f %s %f = %f%n", v1, op, v2, op.calc(v1,v2));
```

8.17.5 EnumMap<K,V>

Estas classes são uma implementação especial de `Map<K,V>` para funcionarem com tipos enumerados, já que todas as chaves de um `EnumMap<K,V>` devem pertencem a **um único tipo enumerado** especificado na sua criação. Não são permitidas chaves de valor `null`.

Um `EnumMap<K,V>` mantém uma ordem nas suas associações que é a ordem natural das suas chaves, ou seja, a ordem natural das chaves tal como apresentadas no tipo enumerado. Esta ordem é reflectida nas iterações usando `keySet()` e `values()`.

O tipo da nossa anterior "encomenda", poderia ser vantajosamente definido como sendo:

```
EnumMap<TShirt, Integer> encom =
         new EnumMap<TShirt, Integer>(TShirt.class);
```

e a tabela de preços contendo o preço para cada tamanho de *t-shirt* definida como:

```
EnumMap<TShirt, Double> tprecos =
        new EnumMap<TShirt, Double>(TShirt.class);
```

Note-se que o construtor de `EnumMap<K, V>` recebe a classe do enumerado que lhe vai fornecer as chaves como parâmetro (ver as razões para tal no capítulo 11).

Depois de introduzirmos nos *maps* preços e quantidades encomendadas para cada tamanho válido de *t-shir*t, como se mostra no código seguinte:

```
tprecos.put(TShirt.S, 6.0);
tprecos.put(TShirt.M, 7.5);
  . . .
encom.put(TShirt.S, 150);
encom.put(TShirt.M, 200);
```

o cálculo do valor total da encomenda consistirá em realizar a iteração sobre as chaves da encomenda, para cada chave aceder à quantidade encomendada, e na tabela de preços ao respectivo preço. O código da operação seria então:

```
// Valor total da Encomenda
double total = 0.0;
for(TShirt t : encom.keySet())
   total += encom.get(t) * tprecos.get(t);
out.println("Total da encomenda: " + total + " euros.");
```

É de salientar que a implementação de `EnumMap<K,V>` é baseada em *arrays* e é muito eficiente. Para conjuntos de chaves pequenos e relativamente estáveis, como são os tipos enumerados, e sempre que temos que associar uma chave a um valor, esta implementação é preferível à utilização de um campo valor no próprio tipo enumerado.

8.17.6 PROPRIEDADES ADICIONAIS

Os tipos enumerados são particularmente úteis sempre que necessitarmos de representar um conjunto fixo de constantes, ou seja, conjuntos de valores bem conhecidos em tempo de compilação. São exemplos de conjuntos deste tipo as opções de um menu, os dias da semana, os meses do ano, os estados de um processo, etc.

Para além de ser possível juntar a cada constante um pequeno "estado" formado por valores manipuláveis, para além dos métodos predefinidos herdados e dos métodos definidos pelo utilizador, os tipos enumerados são `Serializable` e são `Comparable`, mas não `Cloneable`, porque as suas constantes não podem ser clonadas.

Os tipos enumerados não criam instâncias por razões óbvias, por isso a construção `new` não lhes é aplicável. Os tipos enumerados podem ser definidos como uma classe de topo, ou seja, num ficheiro próprio com o mesmo nome, ou como membros `static` de uma outra classe, não sendo, neste caso, serializados.

Um tipo enumerado é implicitamente `static` e `final`, e pode implementar interfaces, podendo ser usados como implementações *type safe* para o problema da representação de constantes. Adicionalmente, as constantes de um tipo enumerado não são compiladas nas "máquinas cliente", pelo que alterações na sua ordem, etc., não afectam os clientes das aplicações que os usam.

As implementações das classes `EnumSet<E>` e `EnumMap<E>` são muito eficientes pois foram pensadas para pequenos volumes de dados. Quando temos que representar informações muito tipificadas, estáveis e de pequena dimensão, estas classes são muito adequadas e têm uma representação muito eficiente (são classes novas em JAVA5).

8.18 COLECÇÕES: *PERFORMANCE*

O JCF fornece uma arquitectura de classes e interfaces (tipos) para a estruturação dos nossos objectos, que tem uma organização, eficiência, facilidade e utilidade ímpares (desde JAVA2), a que junta agora a muito importante segurança de tipos, a sua parametrização, o ciclo *foreach* e *auto-boxing* e *auto-unboxing* (JAVA5).

Tudo isto, como vimos, foi conseguido à custa de apenas dois conceitos, `Collections` e `Maps`, na prática traduzidos em três tipos `List<E>`, `Set<E>` e `Map<K,V>`, sendo o tipo `Collection<E>` o supertipo dos dois primeiros.

A garantia de que todos os tipos são iteráveis, alguns até de mais do que uma forma, permite ao programador a utilização frequente e sistemática de *foreach* para aceder aos elementos das diferentes colecções, qualquer que seja a sua efectiva implementação.

Os construtores foram criados de tal forma que se torna fácil converter *collections* noutras *collections* e *maps* noutros *maps*, bem como através da utilização de métodos designados por *bulk*, tais como `addAll()` e `putAll()`.

Naturalmente que, por questões de eficiência, todas as operações mais "pesadas" existentes sobre colecções, tais como `addAll()`, `putAll()`, `clone()`, etc., bem como todas as operações que dão como resultado certas *views* das colecções, tais como `subList()`, `subMap()`, etc., usam **partilha de referências** com os objectos das colecções originais ou parâmetro, e não cópia, devendo este facto estar sempre presente quando usamos tais métodos, por questões de segurança.

Compreendida a arquitectura do JCF e a sua funcionalidade, resta ao programador, em função dos requisitos do problema, definir o tipo das suas colecções dentre os três grandes tipos existentes, e escolher uma das classes de implementação de tal tipo.

As nossas considerações finais sobre colecções vão para um aspecto em geral descurado, que é a sua eficiência de desempenho em situações de carga, ou seja, quando são sujeitas a armazenamento, pesquisa e iteração com grandes quantidades de objectos.

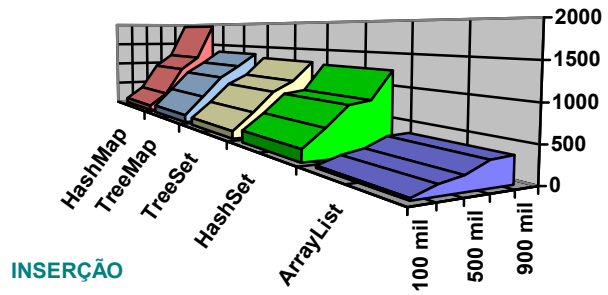
O programa de teste utilizado é algoritmicamente muito simples para não interferir nos próprios objectivos do teste. O programa foi executado numa máquina de configuração comum, em que a invocação de `Runtime.getRuntime().maxMemory()` revela uma *heap* disponível de cerca de 64MB, a *heap* disponibilizada pelo ambiente BlueJ.

Estruturas como `LinkedList<E>`, naturalmente pouco eficientes para operarem com grandes massas de dados, porque não foi esse o objectivo com que foram criadas, ou eficientes mas para dimensões menores, tais como `EnumSet<E>` e `EnumMap<K, V>` não foram incluídas. Estruturas como `Stack<E>`, que baseiam a sua implementação numa das estruturas testadas ou numa muito semelhante (ex.: `Vector<E>`), não foram também incluídas. Os *arrays*, sendo as estruturas de implementação base de todas as outras (através de `Object[]`), foram também excluídos nos testes comparativos.

Assim, foram testadas as colecções usualmente mais utilizadas, designadamente, uma implementação de `List<E>`, a colecção `ArrayList<Integer>` (a implementação sincronizada `Vector<Integer>` apresenta desempenho muito semelhante), duas implementações de `Set<E>`, `HashSet<Integer>` e `TreeSet<Integer>`, e, como implementações de `Map<K,V>`, `HashMap<Integer,Integer>` e `TreeMap<Integer,Integer>`.

O programa **insere** em cada uma destas colecções entre 100 mil e 900 mil instâncias de `Integer`, usando `add(new Integer(2*i))` onde `i` representa a variável de controlo do ciclo `for` onde tal é realizado. No caso dos *maps*, é inserido um par chave-valor dado pela instrução `put(new Integer(2*i), new Integer(2*i))`.

O gráfico e a tabela da Figura 8.57, mostram a média dos tempos, em milissegundos, calculados através dos métodos da classe `GregorianCalendar` antes apresentados.



	100 mil	300 mil	500 mil	700 mil	900 mil
ArrayList	30	140	200	340	360
HashSet	190	461	770	820	1250
TreeSet	130	430	770	950	1280
TreeMap	140	500	880	1100	1350
HashMap	100	370	930	1200	1950

Figura 8.57 – Teste comparativo - Inserção

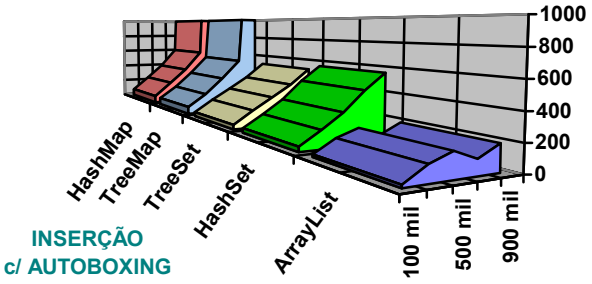
Dos resultados obtidos, são de salientar as características quase lineares de comportamen-to do `ArrayList<>`, o que não será de admirar dado que a inserção é realizada de forma directa no fim da lista actual, e em posições de memória alocadas sequencialmente. Na inserção, o `TreeSet<>` tem pior desempenho que o `HashSet<>` pois tem que garantir que os valores se mantêm ordenados usando comparações com os que já foram inseridos. O `HashSet<>` apenas tem que calcular um endereço e, eventualmente, resolver algumas colisões. Os dois *maps*

apresentam valores semelhantes, sendo mesmo de salientar que o `TreeMap<>`, que usa uma árvore balanceada para manter as chaves na sua ordem natural, tem melhor desempenho que o `HashMap<>`.

Os testes de inserção foram refeitos inserindo objectos do tipo `Ponto2D`, com coordenadas geradas da mesma forma, ou seja, escrevendo `new Ponto2D(i,2*i)`, para comparar com os tempos de inserção dos `Integer`. Para as classes que necessitam de ordem, `TreeSet<Ponto2D>` e `TreeMap<Ponto2D>`, passámos como parâmetro, na sua criação, o necessário `Comparator<Ponto2D>`.

Os resultados obtidos não são apresentados pois são em tudo semelhantes aos medidos usando instâncias de `Integer`.

O teste seguinte consistiu em realizar a mesma operação de inserção mas utilizando o mecanismo de *auto-boxing*, ou seja, em vez de escrevermos `add(new Integer(2*i))`, criando a instância de `Integer` explicitamente, deixou-se o mecanismo de *auto-boxing* actuar, escrevendo apenas `add(2*i)` e `put(2*i, 2*i)`.



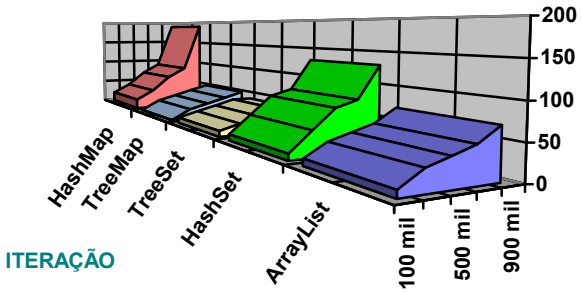
	100 mil	300 mil	500 mil	700 mil	900 mil
ArrayList	20	120	180	111	190
HashSet	40	150	270	430	560
TreeSet	40	160	270	380	520
TreeMap	70	200	360	550	3000
HashMap	75	230	391	601	4900

Figura 8.58 – Teste comparativo - Inserção com *auto-boxing*

Os resultados são muito interessantes, em particular se repararmos que o eixo dos tempos passou a ser graduado só até aos 1000 ms. Os tempos de inserção baixaram drasticamente em todas as colecções. Os *maps*, a partir das 750.000 chaves começaram neste caso a revelar algum comportamento de ruptura, devido a valores do seu factor de carga possivelmente desajustados e/ou necessidade de paginação.

Assim, podemos concluir sem qualquer tipo de dúvida que o mecanismo de *auto-boxing* não só auxilia em termos sintácticos, como está eficientemente implementado, substituindo com vantagem a utilização explícita das classes *wrapper*.

A **iteração** foi programada usando um simples ciclo *foreach*, tendo-se percorrido não as chaves mas os valores no caso dos *maps*. Nenhuma acção foi executada sobre o objecto `Integer` extraído em cada iteração (Figura 8.59).



	100 mil	300 mil	500 mil	700 mil	900 mil
ArrayList	10	30	40	50	70
HashSet	10	40	70	80	130
TreeSet	10	20	30	40	40
TreeMap	0	20	30	40	50
HashMap	20	40	70	90	190

Figura 8.59 – Teste comparativo - Iteração completa

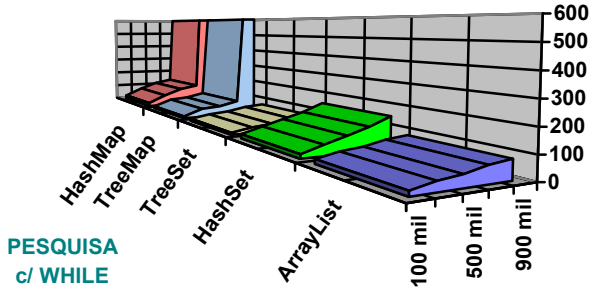
Todas as colecções apresentam tempos bastante impressionantes para a sua completa iteração. `ArrayList<>` e `TreeSet<>` são, dentro das `Collection<>`, de grande eficiência, especialmente o `TreeSet<>` que parece não acusar a carga. Tal pode dever-se a uma alocação de memória mais adequada. Os *maps* são eficientes também ao serem iterados pelos seus valores, operação que é muito comum apesar de existir acesso por chave, sendo de novo o `TreeMap<>` de uma *performance* surpreendente, que resulta certamente da sua excelente implementação.

O último teste consistiu de uma usual operação de **procura** de um objecto na colecção, objecto garantidamente inexistente. No caso dos *maps*, o objecto não foi procurado no conjunto das chaves, mas sim na sua lista de valores.

Numa primeira abordagem foram usados os métodos `contains()` e `containsValue()` para os *maps*. Porém, os resultados obtidos não são interessantes, pois, na pior das hipóteses, os tempos coincidem com os tempos de uma iteração completa.

Assim, as operações de pesquisa foram implementadas usando iteradores e uma estrutura `while`, tal como se mostra a seguir, apresentando-se os resultados na Figura 8.60.

```
while( valores.hasNext() && (!found) ) {
     found = valores.next().equals(new Integer(CHAVE));
}
```



	100 mil	300 mil	500 mil	700 mil	900 mil
ArrayList	20	40	50	70	80
HashSet	20	50	80	110	150
TreeSet	10	30	40	60	70
TreeMap	10	50	60	85	1600
HashMap	20	70	110	130	2500

Figura 8.60 – Teste comparativo - Pesquisa sem sucesso

As classes `ArrayList<>`, `TreeSet<>` e `TreeMap<>` são, como se pode verificar, muito eficientes na procura e principalmente muito lineares. Os *maps* voltam no entanto a acusar sobrecarga a partir de um dado volume de objectos na estrutura, mas há variáveis de configuração dos *maps* que podem ser ajustadas (factor de carga e outras).

Em conclusão, o `ArrayList<>` é de facto uma lista de objectos muito eficiente em todas as operações. Os `TreeSet<>` e `TreeMap<>` são mais equilibrados entre as várias operações do que os `HashSet<>` e `HashMap<>` respectivamente.

Em aplicações onde não existam muitas operações de inserção, ou em que a maioria das operações sejam de consulta, de iteração ordenada ou de remoção, as estruturas que realizam a ordenação através de uma árvore revelaram-se muito eficientes.

No caso do `TreeSet<>`, comparativamente com o `HashSet<>`, a utilização deste último só tem justificação pelo facto de não obrigar à criação de um `Comparator<E>`.

No caso dos *maps*, as diferenças entre ambos justificam, em quase todos os casos, a utilização de `TreeMap<>`, porque são muito semelhantes no esforço de inserção, sendo o `TreeMap<>` claramente mais estável e eficiente nas iterações e pesquisas, para além de ter ordem implícita ou explícita nas suas chaves.

Depois destas últimas considerações sobre a *performance* das várias colecções ao nosso dispor no JCF, mantém-se, no entanto, como fundamental o princípio de base de qualquer engenharia, devidamente adaptado: "primeiro pôr os programas a funcionar bem; depois pô-los rápidos, se possível!".

8.19 SÍNTESE DO CAPÍTULO

O JCF revela-se uma arquitectura de classes genéricas, interfaces, métodos e classes de serviços, de uma organização e concepção excelentes, e de enormíssima utilidade para o programador. Os tipos parametrizados, o novo ciclo *foreach* e os mecanismos de *auto-boxing* e *auto-unboxing*, introduzidos em JAVA5, tornam a linguagem muito mais agradável de utilizar, muito menos confusa e obscura do que quando, quase linha a linha, era necessária uma instrução de *casting*.

O novo ciclo *foreach* é de uma utilidade e simplicidade surpreendentes, tanto mais que se aplica aos *arrays* e é compatível com *unboxing*. Eventualmente, apenas o seu nome poderia ter sido outro (de facto, `foreach`), mas, não sendo *foreach* uma palavra reservada de JAVA (*keyword*), problemas de compatibilidade poderiam surgir em aplicações antigas.

Comparativamente com soluções já encontradas em C++ e C#, o método de cópia de objectos, `Object.clone()` continua a ser uma péssima solução de compromisso, e seria preferível que fosse tornado *deprecated*. A inexistência de uma declaração `const` para objectos imutáveis, e a passagem, por omissão, de cópias dos objectos parâmetro no caso dos construtores, são lacunas que deverão vir a ser colmatadas no futuro.

Os *wildcards*, que permitem especificar parâmetros de tipo aceitáveis como tipos actuais de tipos paramatrizados, repõem a covariância estrutural, que de outra forma seria perdida nos tipos parametrizados.

Outras questões relacionadas com a parametrização e a generalização das classes serão estudadas na abordagem às classes genéricas que faremos no respectivo capítulo.