Módulo 4 - Desenho de Classes

Objetivos

- Saber aplicar mecanismos para garantir a coerência do estado de um objeto.
- Saber organizar classes em pacotes.
- Entender as noções de classe aninhada, interna, e anónima.
- Saber definir classes anónimas através de expressões lambda.

Motivação: garantir a coerência de dados

Qualquer resolução de um problema em programação implica a representação dos dados do problema em variáveis. Tudo são números em computação (booleanos, caracteres, strings, cores, pontos, imagens, etc), e em teoria é possível resolver qualquer problema utilizando apenas números através de tipos primitivos (inteiros, reais, etc), e vetores para lidar com conjuntos de números. Porém, ao lidar com dados mais complexos, a distância conceptual entre aquilo que queremos representar e a codificação em números começa a aumentar, e a nossa (limitada) capacidade cognitiva para pensar apenas em termos de números irá ter implicações na produtividade e capacidade de abordar problemas. Ainda assim, há muitos tipos de dados que são facilmente representados com simples inteiros ou com vetores de inteiros. Por exemplo, uma cor codificada num inteiro, ou uma imagem representada numa matriz de inteiros (`int[][]`).

Em todas as linguagens de programação existe noção de variável composta (pe. *struct* em C), que consiste essencialmente numa variável que é constituída por outras variáveis (que podem também elas ser compostas). Em Java, as estruturas são representadas por objetos, cuja estrutura é definida por uma classe. Por exemplo, uma coordenada de ponto de uma imagem poderia ser um objeto da classe Point:

```
class Point {
      int x, y;
}
```

```
Point p = new Point();
p.x = ...
p.y = ...
```

Ainda que os tipos de dados acima (int[][] e Point) sejam adequados para resolver problemas com imagens, as variáveis não garantem que os valores contidos nas mesmas sejam válidos para esse contexto. Nem todas as matrizes de inteiros representam imagens válidas (os inteiros têm que obedecer a uma codificação de cores), e nem todas as instâncias de Point representam coordenadas válidas para uma imagem (as coordenadas terão que ter valores maiores ou iguais a zero). Assim sendo, será possível que um programa que manipule estas variáveis faça como que as mesmas fiquem com valores incoerentes face aquilo que as mesmas estariam a representar. Por exemplo, introduzindo valores inteiros numa matriz que não podem ser descodificados para uma cor, ou valores negativos num ponto (que não correspondem a uma coordenada de imagem).

Embora estes exemplos sejam simples (para facilitar a ilustração), em domínios com regras de negócio complexas e tipos de objetos a serem partilhados por diferentes projetos e equipas, a questão é mais séria. Uma equipa pode definir um tipo de dados e implementar funcionalidades assumindo que os dados estão coerentes, ao passo que a outra equipa desenvolve outras funcionalidades onde os dados são manipulados incorretamente, causando defeitos no código da primeira equipa ao integrar funcionalidades. Não havendo forma de garantir a coerência dos dados, a colaboração e integração será (mais) difícil.

Classes

As classes são um mecanismo, fulcral em programação orientada para objetos, que visa resolver o problema da incoerência de dados através da definição de tipos de dados que garantem que a sua a coerência seja mantida. Esta possibilidade é considerada uma mais-valia que ajuda a gerir a complexidade na implementação de sistemas e coordenação entre módulos.

No dicionário[1], *classe* pode ser definida como *"grupo de pessoas, animais ou coisas com atributos semelhantes"* ou *"categoria de funções da mesma natureza"*. Fazendo o paralelo para a programação, as *coisas* são os objetos e os *atributos semelhantes* são os dados neles contidos. A segunda definição é útil para enfatizar que as *coisas* podem ser simplesmente *funções/procedimentos* (e não conter dados necessariamente, veja-se os comparadores).

Ao definir uma classe de objetos, a mesma deverá ser desenhada por forma a impossibilitar que as suas instâncias tenham um estado incoerente face àquilo que o objeto é suposto representar. Se tal acontece, então o objeto "sai fora" da classe pois já não obedece às características esperadas, e logo, a classe não estará bem desenhada. As condições que definem o que é considerado um estado válido do objeto são denominadas de *invariantes*, pois deverão ser sempre verdade (daí o nome invariante: são sempre verdadeiras, nunca variam).

A implementação de classes caracteriza-se por agregar dados (atributos) e operações válidas para esses mesmos dados. No momento imediato após a sua criação, um objeto deverá estar num estado coerente. Por outro lado, as operações disponíveis nunca deverão permitir que o objeto fique com dados incoerentes. Desta forma, as operações podem assumir que os dados estão coerentes, e produzirão resultados válidos com base neste pressuposto.

Existem dois mecanismos para garantir a coerência do objeto, encapsulamento e lançamento de exceções. Encapsulamento é a forma que garante que os atributos de um objeto não são modificados externamente, mas sim apenas pelos métodos da classe, que detalharemos em seguida. O lançamento de exceções é o mecanismo para causar um erro devido a incorreta utilização de um objeto, à semelhança de uma exceção *Array Index Out Of Bounds* quando acedemos a um índice inválido para determinado vetor. Desta forma, tanto no momento da criação do objeto (construtor), como nos seus métodos, os valores dos parâmetros deverão ser validados por forma a não permitir que o objeto fique num estado incoerente. Ao não fazer este controlo, os benefícios oferecidos pela utilização de classes (em vez de meras estruturas) são postos em causa, e o paradigma de programação orientada para objetos não estará a ser aplicado de forma correta neste aspeto.

[1] Infopédia, www.infopedia.pt

Encapsulamento em classes

Para ilustrar os problemas que o encapsulamento resolve, considere o seguinte (mau) exemplo de classe para representar cores RGB.

```
public class Color {
  public int r, g, b;
  ...
}
```

Classe com atributos públicos mutáveis (mau desenho).

```
Color c = ...
c.r = 1000;
c.g = -1;
c.b = 255;
```

Modificação externa do valor dos atributos, tornando o objeto incoerente.

Esta solução permite que o objeto seja modificado para valores que não fazem sentido, como por exemplo no código exemplo à direita, pois os valores RGB teriam que estar no intervalo [0, 255]. O objeto, que se pretendia que representasse uma cor, ficou num estado que não faz sentido para esse propósito, o que vai contra a ideia de garantir a coerência.

Uma solução que impediria a modificação seria tornar os atributos definitivos, através do modificador `final`. Desta forma, validando os valores RGB iniciais no construtor da classe, não permitindo valores inválidos através do lançamento de exceção, garante que o objeto é criado com valores RGB válidos. Ao não ser possível alterar o valor dos atributos, o objeto estará sempre coerente, eliminando este problema.

```
public class Color {
  public final int r, g, b;

  public Color(int r, int g, int b)
    throws IllegalArgumentException {
      ...
}
```

Classe com atributos públicos, mas definitivos. Garante coerência, mas não facilita evolução.

```
Color c = ...
c.r = 1000;
c.g = -1;
c.b = 255;
```

A modificação de atributos definitivos gera erro de compilação.

Impedir a modificação, neste caso particular de objeto *imutável* (detalhes mais à frente), resolve o problema da coerência. Porém, permitir acesso aos atributos para leitura acarreta outra desvantagem. Esta desvantagem é a falta de flexibilidade, pois não será possível futuramente, alterar a lógica de funcionamento da classe sem quebrar o código que a utiliza. No caso das cores RGB, dado que a gama de valores é reduzida ([0, 255]), a classe Color poderia ser otimizada em termos de memória utilizando um único valor `int` (32 bits) "dividido" em pedaços de 8 bits, acomodando os três valores RGB (os 8 bits restantes utilizam-se normalmente para representar a transparência). Efetuar esta modificação iria potencialmente quebrar código cliente, como no seguinte exemplo.

```
public class Color {
  public final int value;

  public Color(int r, int g, int b)
    throws IllegalArgumentException {
      ...
}
```

Alteração na classe para otimização de memória.

```
static int luminance(Color c) {
  return (int) (
    c.r * 0.21 +
    c.g * 0.72 +
    c.b * 0.07
  );
}
```

Código correto para a versão anterior, mas quebrado na nova versão.

Imaginando um projeto grande, onde a classe Color é utilizada em centenas de partes do código, a alteração anterior iria provavelmente não ser levada a cabo, dados os custos de modificação do código cliente associados. A acontecer isto, o facto de manter os atributos públicos consistiu num impedimento à evolução (para melhor).

Encapsular o estado de um objeto consiste em "esconder" os seus atributos do exterior, utilizando o modificador `private`. Desta forma, apenas os métodos da classe podem aceder aos seus atributos, impossibilitando que código externo à classe aceda aos seus valores, quer para escrita (coerência) quer para leitura (flexibilidade de evolução). Para expor propriedades do objeto, sejam obtidas diretamente por um atributo ou não, definimos os chamados métodos *inspetores* (tipicamente em Java obedecem à nomenclatura "get*Propriedade()*").

```
public class Color {
  private int value;

  public Color(int r, int g, int b) {
      value = encode(r, g, b);
  }

  private int encode(int r, int g, int b) {...

  private Color(int value) {
    this.value = value;
  }

  public int getR() {
     return (value>>16) & 0xFF;
  }
  ...
  public Color copy() {
    return new Color(value);
  }
}
```

O encapsulamento também pode ser aplicado a métodos e construtores, quando a sua invocação externa não é desejável. As razões podem ser relativas à operação simplesmente não ser relevante para o exterior, ou mais grave, da sua execução poder comprometer a coerência do objeto.

O construtor Color(int) foi encapsulado, pois é um auxiliar para a função de cópia, e não é suposto utilizar externamente (o valor do parâmetro teria que estar coerentemente codificado). O método encode(...) foi também encapsulado, pois só é relevante internamente.

Pacotes (*packages*)

Em Java as classes e interfaces podem ser organizadas em pacotes, sendo que cada classe pode apenas pertencer a um pacote. Um pacote é identificado por uma sequência de identificadores separados por pontos, e para declarar que uma lhe classe pertence é necessário incluir uma declaração no início do ficheiro .java da classe.

```
package pt.iscte.poo;
import ...
public class Exemplo { ...
```

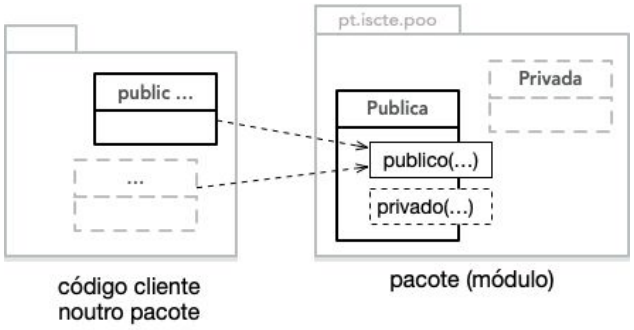
Ao incluir esta declaração seremos forçados a que o ficheiro da classe esteja localizado no diretório composto pela sequência de identificadores do pacote (pt/iscte/poo/Exemplo.java). As que classes que pertencem a determinado pacote serão à partida relacionadas em termos de cooperação para atingir determinada funcionalidade.

Encapsulamento em pacotes

A divisão de ficheiros em diretórios imposta pela definição de pacotes promove alguma organização, o que por só por si ajuda a lidar com sistemas compostos por um número elevado de classes. Porém, a divisão em pacotes oferece outras possibilidades relacionadas com a noção de módulo. Um conjunto de classes de um pacote que aborda determinada funcionalidade pode ser desenhado como sendo um módulo reutilizável por outras classes (por exemplo, as coleções do Java). As classes que não são relevantes (ou indesejáveis) para a utilização do módulo podem ser encapsuladas no pacote a que pertencem, no sentido em que não poderão ser acedidas por outras classes exteriores ao pacote. Isto pode ser feito omitindo o modificador *public* (ficando implicitamente *package-private*). Também é possível utilizar este tipo de permissão de acesso em construtores/métodos (e atributos, mas desaconselhado), forçando que estes apenas possam ser acedidos por classes do mesmo pacote.

```
package pt.iscte.poo;
class Privada { ...
```

```
package pt.iscte.poo;
public class Publica {
  public void publico(...) { ...
  void privado(...) { ...
```



Ter uma classe privada de um pacote que tenha sido desenhada de forma mais “descuidada” (sem encapsulamento, etc) é bastante menos grave que uma classe pública, pois o potencial “estrago” está confinado ao pacote, e logo, a um âmbito tipicamente reduzido de classes.

Exercício 4.1 - Classe para matrizes algébricas

a) Desenvolva uma classe Matrix para representar matrizes algébricas de inteiros, permitindo:

- Criar uma matriz (toda a zeros) dado o número de linhas e colunas.
- Criar uma matriz quadrada (toda a zeros) dado um único número para linhas/colunas.
- Saber o número de linhas e colunas da matriz.
- Aceder e alterar os seus elementos individualmente, dado o índice de linha e coluna.
- Alterar a matriz, multiplicando todos os seus elementos por inteiro (escalar).
- Saber se a matriz tem a mesma dimensão que outra.

Atenda aos princípios do encapsulamento. Utilize uma matriz de inteiros (`int[][]`) como representação interna.

b) A título de exemplo de código cliente da classe, desenvolva uma função numa classe à parte, que dadas duas matrizes, soma-as e devolve o resultado dessa soma escalado por um valor dado.

```
static Matrix sumAndScale(Matrix a, Matrix b, int scalar) { ...
```

c) Copie a classe desenvolvida na alínea (a) para outro ficheiro (para não a perder), e altere a classe inicial para representar internamente o conteúdo da matriz através de um vetor de inteiros e um inteiro para saber o número de colunas, que é uma representação alternativa à inicial, mais económica em termos de memória e que permite varrimentos por todos os elementos mais eficientes.

```
public class Matrix {
  private int[] data;
  private int columns;
  ...
}
```

Se tudo correr bem, poderá constatar que o código cliente desenvolvido na alínea (b) não terá sido afetado (e continuará a funcionar corretamente) após as alterações na classe.

Objetos imutáveis

Um objeto imutável, tal como o nome indica, é um objeto cujo estado nunca muda. Por outras palavras, o valor dos seus atributos é constante. A segunda versão da classe Color apresentada anteriormente é um exemplo disso. Outro exemplo de imutabilidade, são os objetos da classe String. Uma vantagem dos objetos imutáveis é que, garantida a sua correta criação, a sua coerência será sempre mantida (não é possível alterar o valor dos atributos).

Optar por uma classe de objetos imutáveis tem outra motivação que consiste no facto de pretendermos tratar estes objetos como valores (à semelhança os tipos primitivos), que são copiados na atribuição de variáveis. No caso de tipos de referência (pe. String), os objetos não são copiados mas sim a referência. Porém, como objeto não pode ser alterado, o efeito prático é o mesmo.

Considere o seguinte exemplo *hipotético* com Strings *mutáveis*.

```
Aluno aluno = new Aluno("Zé Povinho");
String nome = aluno.getNome();
nome.setCharAt(2, ‘,’);         // modificaria a string no indice 2
```

Para garantir a integridade do objeto Aluno, o método getNome() teria que devolver uma cópia defensiva do seu atributo para garantir que o mesmo não é alterado externamente de forma indevida (por exemplo, podia ser requisito ter um nome que apenas contém letras). Sendo as Strings na verdade imutáveis, a sua partilha nunca causará problemas, pois o objeto partilhado não pode ser alterado.

Igualdade em objetos imutáveis

Os objetos imutáveis muitas vezes representam valores, e dada a semelhança aos tipos primitivos, seremos tentados a efetuar comparações utilizando o operador ==. Porém, não esquecer que nos tipos de referência (todos os objetos e vetores), este operador verifica se as referências apontam para o mesmo objeto, e não se os objetos apontados são iguais em termos de conteúdo. Por norma, devemos utilizar sempre a operação `equals(...)` para comparar objetos (abordaremos os detalhes num módulo mais à frente).

Exercício 4.2 - Classe para tempos (duração)

Desenvolva uma classe Time de objetos imutáveis para representar tempos (*hh*:*mm*:*ss*). Deverá ser possível:

- Construir um tempo fornecendo (horas, minutos, seguntos) em inteiros separados, ou em formato textual (*“hh:mm:ss”*).
- Consultar o número de horas, minutos, e segundos isoladamente.
- Acrecentar isoladamente horas, minutos, ou segundos.
- Saber se outro tempo é menor.
- Obter um novo tempo resultante da soma com outro tempo.
- Obter um novo tempo resultante da subtração com outro tempo.

Sugestão: represente internamente o tempo num único atributo com o total de segundos.

Classes aninhadas (*nested classes*)

Existem classes cujos objetos são apenas utilizados no contexto de outra classe "mãe". Estes casos são comuns na situação onde uma classe precisa de uma estrutura auxiliar para realizar o seu propósito. Por exemplo, no Exercício 2.1, a classe sugerida *Inscrição* para gerir as inscrições e notas associadas só será utilizada pela classe *Disciplina*, Neste caso fará sentido definir esta classe como aninhada, isto é, definida dentro da classe onde é utilizada.

```
public class Disciplina {
  private static class Inscricao {
  ...
  }
}
```

A não inclusão do modificador `static` irá ter implicações (ver próxima secção). As classes aninhadas não são obrigatoriamente privadas, mas é comum é serem-no. Desta forma, só poderão ser utilizadas no código da classe que a define. Embora seja permitido, ter uma classe aninhada pública que é instanciada em várias classes não será uma boa forma de organização (i.e., sendo necessária em várias classes, porquê estar “arrumada” numa em particular?).

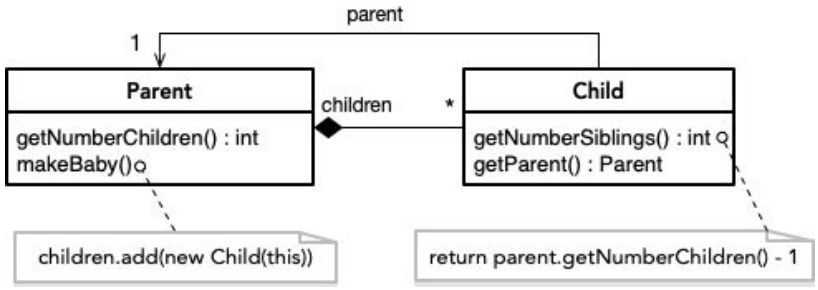
A utilização de uma classe aninhada não se traduz num efeito prático em termos de execução, é uma mera questão de organização do código. Porém, num projeto com centenas de classes, se pudermos "esconder" algumas estaremos a contribuir para ter menos ficheiros, e logo, para uma orientação no código mais fácil. Só precisaremos de tomar contacto com uma classe aninhada caso seja necessário entender a classe que a contém. Por outro lado, pode ser indesejável expor essa classe, e nesse caso devemos ter a classe aninhada privada (encapsulamento no pacote).

Classes internas (*inner classes*)

As classes aninhadas podem ser definidas como *classes internas*, por forma a que cada um dos seu objetos fique automaticamente associados ao objeto da *classe-mãe* que o criou. As classes internas são uma possibilidade sintática que economiza algum código, sendo úteis apenas quando os seus objetos necessitam de aceder ao objecto da classe-mãe que os criou.

Na seguinte ilustração (diagrama e código à esquerda) vemos uma classe Parent, que detém um conjunto de objetos Child, representando uma relação progenitor-descendentes. Note que o construtor de Child que recebe uma referência do objeto Parent que o cria. Sobre um objeto Child podemos saber quantos irmãos tem, e para isso, este acede ao objeto Parent (número de filhos deste menos um). Dado que a necessidade de associar objetos desta forma é comum, especialmente em objetos complexos (interfaces gráficas, *threads*, etc), as classes internas

oferecem a possibilidade de abreviar um pouco a sintaxe, como se ilustra na solução alternativa na parte direita. Note-se a ausência de `static` na declaração da classe interna. Isto fará com que "invisivelmente" exista um atributo (equivalente a `parent` na solução original) que é uma referência para o objeto que o criou (acedida através de `Parent.this`), copiada implicitamente no momento da criação do objeto. Os objetos da classe interna podem aceder aos membros da classe-mãe porque essas chamadas são delegadas nessa referência (p.e. `getNumberChildren()` é implicitamente `Parent.this.getNumberChildren()`).




```
class Parent {
  List<Child> children = new ...

  int getNumberChildren() {
    return children.size();
  }

  void makeBaby() {
       children.add(new Child(this));
   }

  static class Child {
    final Parent parent;

    Child(Parent parent) {
       this.parent = parent;
    }

    int getNumberSiblings() {
      return parent.getNumberChildren()-1;
    }

    Parent getParent() {
      return parent;
    }
  }
}
```

Solução com classe interna (inner class)

```
class Parent {
  List<Child> children = new ...

  int getNumberChildren() {
    return children.size();
  }

  void makeBaby() {
       children.add(new Child());
  }

  class Child {
    int getNumberSiblings() {
      return getNumberChildren()-1;
    }

    Parent getParent() {
      return Parent.this;
    }
  }
}
```

Ao passo que numa classe aninhada os objetos podem ser criados num contexto externo a um método de instância da classe-mãe, os objetos de uma classe interna necessitam sempre de um objeto da classe-mãe acessível (para que seja passado implicitamente na criação). Isto é tipicamente feito num método de instância da classe-mãe (objeto obtido por `this`). Se o objeto da classe aninhada não precisa de aceder ao objeto da classe-mãe, então a utilização de classe interna é desnecessária, pois esta terá mais um atributo (mais memória) com a referência para o objeto da classe mãe que o criou.

Classes anónimas

Um exemplo típico de classe aninhada consiste na definição de pequenas classes auxiliares para uma tarefa, como o por exemplo um comparador, numa classe que implementa a interface java.util.Comparator. Por vezes, este tipo de classes é necessário apenas para uma única situação no código, e ainda que a classe seja aninhada, é necessário escrever bastante e será instanciada apenas uma vez. Nestas situações podemos recorrer à utilização de classes *anónimas*, que são classes internas que implementam uma interface (ou derivam de outra classe, a ver mais tarde) e cuja definição é "misturada" com a criação de um objeto compatível com essa interface, normalmente apenas junto do local onde é necessária. O exemplo seguinte consiste num exemplo de classe anónima para um comparador de inteiros. Como se pode ver, a classe que está a ser definida não tem nome (anónima).

```
List<Integer> list = ...
Comparator<Integer> comp = new Comparator<Integer>() {
  public int compare(Integer a, Integer b) {
    return a - b;
  }
};
list.sort(comp);
```

Uma forma comum de utilizar classes anónimas no código será sem sequer definir uma variável, mas sim como no seguinte exemplo.

```
list.sort(new Comparator<Integer>() {
  public int compare(Integer a, Integer b) {
    return b - a;
  }
});
```

Mesmo que as classes anónimas economizem alguma escrita de código, ainda sofrem de alguma verbosidade. Note-se nos dois últimos exemplos que, à parte de um utilizar variável e outro não, as definições das classes anónimas têm muitos caracteres em comum, só diferem no corpo da função. A próxima secção apresenta um mecanismo para evitar esta questão.

Expressões lambda

Muitas das interfaces utilizadas em programação declaram apenas uma operação, tal como no caso de java.util.Comparator. No Java, este tipo de interfaces são designadas de interfaces *funcionais*. Ao escrever uma classe que implementa estas interfaces, estamos no fundo apenas a definir uma única função (ou procedimento). A anotação `@FunctionalInterface` pode ser utilizada na interface para garantir esta característica em tempo de compilação.

A definição de classes internas anónimas para interfaces funcionais pode ser sintaticamente muito abreviada com a utilização de *expressões lambda*. No exemplo seguinte podemos ver uma expressão lambda que define um comparador equivalente ao primeiro exemplo da secção anterior (ascendente). Os parâmetros da função são referidos por `(a, b)` e a expressão que define o resultado da função é escrita à direita de `->`. Podem ser utilizados quaisquer identificadores para os parâmetros.

```
Comparator<Integer> comp = (a, b) -> a - b;
```

Quando a função tem apenas um parâmetro é possível omitir os parênteses. Quando a parte direita consiste apenas numa expressão, esta corresponde implicitamente ao retorno da função. Como se pode verificar na seguinte comparação, a utilização de expressões lambda tem potencial para economizar bastante a escrita de código.

```
list.sort(new Comparator<Integer>() {
  public int compare(Integer a, Integer b) {
    return b - a;
  }
});
```

```
list.sort((a, b) -> b - a);
```

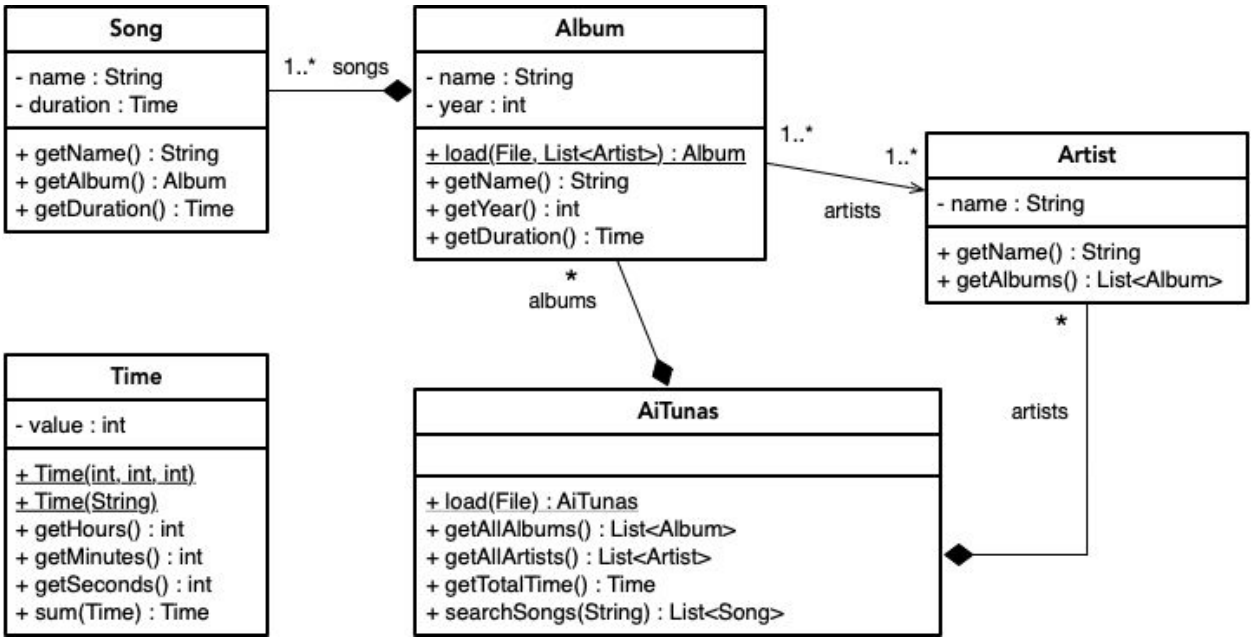
Equivalente ao código à esquerda utilizando expressão lambda

Caso seja necessário mais do que uma instrução, a parte direita pode ser definida com um bloco como se de uma função normal se tratasse, e nesse caso a instrução de retorno terá que ser explícita.

```
list.sort((a, b) -> {
  int c = b - a;
  return c;
});
```

Exercício 4.3 - AiTunas

Suponha um programa para gerir músicas a que chamaremos AiTunas. O programa gere um conjunto de músicas, organizadas em álbuns. Considere que cada música pertence a exatamente um álbum, e é definida por um nome e uma duração. Cada álbum tem um nome e um ano de lançamento, e está associado a um ou mais artistas. Um artista pode estar associado vários álbuns. O seguinte diagrama ilustra a estrutura de classes e operações pretendidas para o AiTunas.



Desenvolva um programa onde um álbum pode ser carregado de um ficheiro. Por sua vez, os álbuns do AiTunas são carregados a partir de um diretório, carregando um álbum por cada ficheiro encontrado. Teste as operações da classe AiTunas, considerando que as listagens de álbuns são por ordem de ano de lançamento, as de artistas por ordem alfabética do seu nome, e as de músicas agrupadas por duração (decrescente). Utilize a classe Time desenvolvida no Exercício 4.2 para representar as durações. Sugestão de faseamento:

1. Desenvolver Album e Song (podendo esta ser classe interna), ignorando Artist e AiTunas; desenvolver carregamento de Album de ficheiro (descartando a informação dos artistas).
2. Desenvolver AiTunas e carregamento de conjunto de albums de ficheiro, continuando a ignorar Artist;
3. Desenvolver Artist, e integrar com Album e AiTunas, acrescentando um atributo a Album e a AiTunas, e passando a considerar os artistas no carregamento de ficheiros. Dado que os artistas são partilhados entre álbuns, será conveniente fazer o carregamento dos vários álbuns passando uma lista partilhada de artistas.

Formato do ficheiro

`Juntos (Live)`	*título do álbum*
`2015`	*ano de lançamento*
`Sérgio Godinho`	*artista*
`Jorge Palma`	*artista*
	(linha em branco para assinalar fim da lista de artistas)
`5:09 Mudemos de Assunto`	*duração (espaço) nome de música*
`3:44 Dá-me Lume`	*duração (espaço) nome de música*
`5:11 O Lado Errado da Noite`	*duração (espaço) nome de música*
`...`	*...*

Exercício 4.4 - AiTunas + Playlists

Estenda o exercício anterior de forma a que seja possível definir *playlists* que agregam músicas que podem ser provenientes de vários álbums. As *playlists* podem ser definidas manualmente, i.e. selecionando música a música, ou podem ser *smart playlists*, onde dado um artista contêm automaticamente todas as músicas do mesmo.