Parte 5 – Classes Abstratas e Interfaces

Fernando Barros, Karima Castro, Luís Cordeiro,
Marília Curado, Nuno Pimenta

Os conteúdos desta apresentação baseiam-se nos materiais produzidos por António José Mendes para a unidade curricular de Programação Orientada a Objetos.

Quaisquer erros introduzidos são da inteira responsabilidade dos autores.

Classes abstratas

- As classes até agora apresentadas definem completamente a estrutura e o comportamento das suas instâncias (objetos)
- Há situações em que não é fácil determinar qual o código a colocar numa superclasse, mesmo sabendo que existe um dado conjunto de subclasses
- Por exemplo, considere-se um programa que necessite de manipular formas geométricas, incluindo, triângulos, quadrados e círculos

Classes abstratas

- É importante que estas formas sejam compatíveis, ou seja implementem os mesmos métodos, nomeadamente (por exemplo) `area()` e `perimetro()`
- Se as três classes implementam estes métodos faz sentido criar uma superclasse, seja `Forma`, para estas três classes
  - Será possível posteriormente criar classes que representem outras figuras (rectangulo por exemplo), sem que isso implique alterações no código existente e garantindo-se que a nova classe é compatível com as anteriores (pois herda os mesmos métodos)

Classes abstratas

- Se `area()` e `perimetro()` estiverem definidos em `Forma` garante-se que todas as suas subclasses, actuais ou futuras, terão também estes métodos definidos
- No entanto, não é claro qual o código que estas classes devem conter, pois a forma de calcular a área e o perímetro varia entre as várias formas
  - Na verdade não há qualquer código comum possível na implementação destes métodos para as várias figuras
  - Portanto, apesar de ser correto incluir estes métodos em `Forma`, não há qualquer código que faça sentido incluir nesses métodos
  - É uma situação em que faz sentido ter uma classe, mas ela não tem necessidade de ser codificada

Classes abstratas

- As **classes abstratas** permitem a declaração de métodos abstratos (métodos sem implementação)
  - No exemplo, faz sentido colocar em `Forma` a declaração dos métodos `area()` e `perimetro()`, obrigando assim todas as suas subclasses a responderem a estes métodos, mas deixando para cada uma delas a obrigação de implementar a sua versão dos métodos
  - Não é possível criar instâncias de classes abstratas
  - Em Java as classes e métodos são declarados como abstractos através da utilização do modificador `abstract`

Classes abstratas

- A classe `Forma` pode ser definida por:

```
abstract class Forma {
  public abstract double area();
  public abstract double perimetro();
  public double semiPerimetro() {
       return 0.5 * perimetro();
  }
}
```

- Os métodos abstratos podem ser usados na definição de métodos concretos

Classes abstratas

- No exemplo, esta classe pode ser herdada pelas classes específicas de cada figura. Por exemplo:

```
class Retangulo extends Forma {
  private double comp, larg;
  public Retangulo(double comp, double larg) {
       this.comp = comp;
       this.larg = larg;
  }
  public double area() {
       return comp * larg;
  }
  public double perimetro() {
       return 2 * (comp + larg);
  }
}
```

Classes abstratas

- Uma subclasse de uma classe abstrata herda os seus métodos, podendo implementá-los ou não
  - Se implementar todos eles poderá ser uma classe concreta
  - Se deixar algum por implementar será também uma classe abstrata
- A relação de herança entre a classe abstrata e as suas subclasses é semelhante à relação entre qualquer duas classes, pelo que se pode fazer:

```
Forma f = new Triangulo(5, 2);
```

Classes abstratas

- Resulta que é conveniente declarar o máximo de métodos abstratos, já que esta será a linguagem comum a todas as classes que venham a ser criadas
  - Este conjunto de métodos é o que garante compatibilidade a nível de polimorfismo
- As classes abstratas podem também ter variáveis de instância que são herdadas pelas suas subclasses
  - Isto implica que, apesar de não poderem gerar instâncias, as classes abstratas podem ter construtores, cuja função é inicializar as variáveis e que são chamados pelos construtores das subclasses

Classes abstratas

- Uma classe abstrata, ao não implementar todos os seus métodos, delega (ou força) as suas subclasses concretas a fornecer a sua implementação particular desses métodos, facilitando o aparecimento de implementações diferentes nas várias subclasses
- Na herança entre classes concretas a redefinição de métodos é opcional
- Na herança a partir de classes abstratas a redefinição de métodos é obrigatória para se poder definir uma classe concreta

Classes abstratas

- As classes abstratas permitem:
  - Escrever especificações sintáticas para as quais são possíveis múltiplas implementações, de momento ou no futuro
  - Normalizar a linguagem a partir de certo ponto na hierarquia
  - Introduzir flexibilidade e generalidade no código
  - Tirar partido do polimorfismo
  - Realizar classificações mais claras de subclasses

Interfaces

- O termo **interface** é usado para representar o conjunto de serviços fornecidos por um objeto
  - **Nota**: não confundir interface com interface gráfico (GUI)
- O conjunto de métodos que um objeto oferece define a forma como o resto do sistema interage com ele
- A linguagem Java tem um meio de formalizar este conceito
- Uma interface é uma coleção de métodos abstratos, ou seja é um conjunto de especificações sintáticas de métodos que representam um dado comportamento que qualquer classe pode implementar

Interfaces

- O conceito de interface permite definir um tipo de dados abstrato e também garantir que classes sem relações de herança apresentem comportamentos comuns
- As interfaces especificam um tipo de dados e qualquer classe que os implemente passa a ser compatível com esse tipo de dados
- Uma classe (não abstrata) que implemente uma dada interface terá que fornecer uma implementação para todos os métodos abstratos definidos nessa interface

Interfaces

- Esta relação (realização) é especificada na definição da classe:

```
class aClasse implements anInterface {...}
```

- Por exemplo, definindo a seguinte interface:

```
interface IForma {
    double area();
    double perimetro();
}
```

Interfaces

- Podemos agora criar uma classe que implementa a interface `IForma`:

```
class Circulo implements IForma {
  private double raio;
  public Circulo(double raio) {
       this.raio = raio;
  }
  public double area() {
       return Math.PI * raio * raio;
  }
  public double perimetro() {
       return 2 * Math.PI * raio;
  }
}
```

Interfaces

- Uma variável `IForma` pode referenciar qualquer instância de uma classe que implemente a interface `IForma`

```
class Retangulo implements IForma {…}
IForma a = new Circulo(3.0);
IForma b = new Retangulo(…);
```

- Uma classe pode implementar várias interfaces
- A possibilidade de implementar várias interfaces fornece muitas das características de herança múltipla (a capacidade de uma classe herdar várias classes)
  - A herança múltipla não é suportada em Java

Interfaces

- Uma interface pode ser implementada por muitas classes, mesmo que não sejam relacionadas por herança
- Uma interface não é uma classe e não pode ser utilizada para criar objetos
- Uma interface pode ser derivada de outra utilizando a palavra reservada `extends`
- A subinterface herda todas as definições da super interface

Exemplo

- Pretende-se um programa que leia os dados de um conjunto de estudantes (nome e um conjunto de notas), calcule a sua média e ordene os estudantes por ordem decrescente das médias
  - Classe `Estudante` para representar um estudante
  - Classe para representar uma turma de estudantes (`Turma`)
    - Vai guardar um conjunto de estudantes num `ArrayList`
    - Tem como comportamentos:
      - a sua inicialização
      - a adição de um estudante ao `ArrayList<Estudante>`
      - a escrita dos dados de todos os estudantes
      - a sua ordenação por ordem decrescente de médias
  - Classe para fazer a gestão do sistema (`GereTurma`)

Exemplo – interface `Comparable<E>`

- O odenamento de uma lista pode ser obtido de uma forma simples pela utlização do método
  - `Collections.sort(List<E> lista);`
- A classe `E` deverá implementar o interface `Comparable<E>` e definir o método `int compareTo(E)` que será usado para definir o ordenamento da lista
- Por exemplo: para ordenar estudantes por ordem descrescente da média poderemos definir a classe

```
class Estudante implements Comparable<Estudante> {
    ...
    public float getMedia() {...}
    public int compareTo(Estudante e) {
        return (e.getMedia() - getMedia() > 0)? 1: -1;
    }
}
```

Exemplo

```
class Turma {
   private List<Estudante> lista;
   //Construtor - cria ArrayList para guardar dados
   public Turma() {
      estudantes = new ArrayList<Estudante>();
   }
   //Ordena estudantes por ordem decresc. da média
   public void ordenaTurma() {
      Collections.sort(estudantes);
   }
}
```

Exemplo

```
//Adiciona um novo estudante
public void juntaEstudante() {
   Estudante e = new Estudante();
   //Lê informação relativa ao estudante
   estudantes.add(e);
}
//Imprime os dados de todos os estudantes
public void imprimeTurma() {
   for (Estudante e: estudantes)
       System.out.println(e);
}
```

Exemplo

```
class GereTurma {
    public static void main(String[] args) {
       Turma t = new Turma(); // Cria uma turma
       int escolha;
       Scanner stdin = new Scanner(System.in);
       do {// Menu
        System.out.println("1 - Adicionar estudante");
        System.out.println("2 - Ordenar estudantes");
        System.out.println("3 - Lista de estudantes");
        System.out.println("0 - Sair");
        escolha = stdin.nextInt();
        switch (escolha) {
            case 1: t.juntaEstudante(); break;
            case 2: t.ordenaTurma(); break;
            case 3: t.imprimeTurma(); break;
            case 0: System.exit(0);
        }
       } while (escolha != 0);
       stdin.close();
    }
}
```