6 - Classes Abstractas

6.1 Introdução às classes abstractas

Todas as classes até agora apresentadas definem completamente a estrutura e o comportamento das suas instâncias, ou seja, definem quer as variáveis de instância que cada instância sua possuirá como sendo o seu estado interno, quer o código dos métodos a executar em resposta a cada uma das mensagens a que as instâncias deverão ser capazes de responder.

Situações surgem no entanto na concepção de aplicações, em que se torna bastante difícil determinar que código colocar numa dada superclasse, mesmo quando se sabe já muito bem quais são algumas das subclasses que vamos querer implementar. Tal pode parecer estranho, mas são de facto situações muito frequentes, e até bastante fáceis de identificar em projectos.

Por exemplo, consideremos que, no desenvolvimento de uma dada aplicação, surge a necessidade de representar e manipular formas geométricas, ainda que numa fase inicial se pretenda ter apenas triângulos, quadrados e círculos. Qualquer que seja a representação definida para estas formas, fundamental é que todas elas implementem o mesmo conjunto de métodos, ou seja, "falem a mesma linguagem" ou API, e, em especial, tenham implementados os métodos `area()` e `perimetro()`.

Confrontados com a necessidade de criar estas novas classes, mas tendo em mente tudo o que anteriormente se viu relativamente às vantagens de se trabalhar sempre com uma superclasse que seja também o **supertipo** das várias subclasses, e da generalização e da facilidade de extensão que, pelos mecanismos intrínsecos da PPO, tal concepção introduz no código das aplicações, é portanto natural que se pense de imediato em criar uma classe que seja a *superclasse* destas três classes que nos são pedidas. Vamos dar-lhe um nome um pouco mais genérico que os das subclasses. Seja tal classe a classe **`Forma`**. Teremos então, conceptualmente, a hierarquia que se apresenta na Figura 6.1.

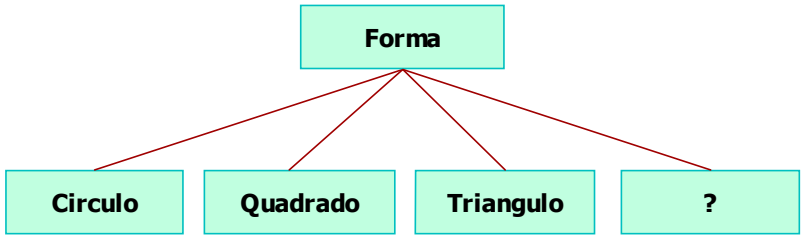

Figura 6.1 – Hierarquia com classe concreta

A ideia está perfeitamente correcta. A concepção baseada em herança e polimorfismo deve ser assim mesmo pensada, tanto mais que desse modo comporta futuras extensões sem qualquer tipo de necessidade de revisões do código. Por exemplo, se mais tarde outras figuras tiverem que ser acrescentadas (ver Figura 6.1), o código não terá que ser reescrito.

Vejamos agora como vamos implementar o que concebemos. Conforme os requisitos do projecto, as subclasses de `Forma` devem implementar todas elas os métodos pedidos. No entanto, todas elas herdam de `Forma`, pelo que temos uma certeza e uma dúvida.

A certeza é a de que, na classe `Forma`, vamos ter que escrever os métodos `area()` e `perimetro()`, desta forma **"obrigando" a que todas as subclasses**, por herança ou por redefinição, **respondam às respectivas mensagens**.

A dúvida é que código deve ser escrito em tais métodos de forma a que seja útil, e, assim, reutilizável por todas as actuais e futuras subclasses de `Forma`?

Sendo a resposta à dúvida "nenhum", porque **não há código comum possível** para tais métodos de tão diferentes formas, então, e já que somos obrigados a escrever "qualquer coisa" em `Forma`, o melhor é escrevermos algo tão disparatado *(dummy)* que a primeira coisa que o programador de uma subclasse de `Forma` vai fazer é reescrevê-lo.

Assim, poderíamos implementar em `Forma` métodos do tipo seguinte:

```
public double area() { return Double.MIN_VALUE; }
public double perimetro() { return Double.MAX_VALUE; }
```

que certamente obrigariam à sua imediata redefinição nas subclasses.

As duas frases anteriormente indicadas a negrito mostram as condições fundamentais que conduziram a este tipo de situação tão incómoda, em que se desenvolve uma classe para fins de generalidade de tipos e extensibilidade do código de um projecto, mas ela em si não tem intrinsecamente necessidade de ser codificada, pois não há factores comuns de estrutura nem de comportamento das suas subclasses que nela se possam reunir, ou seja, que a justifiquem enquanto **classe concreta**.

A maioria das linguagens de PPO (cf. Smalltalk, Eiffel, C++, JAVA, etc.) permite que a definição de uma classe possa ser **incompleta**, ou seja, que alguns métodos (ou mesmo todos) possam ser apresentados sintacticamente, isto é, lhes seja atribuído um nome, um tipo de resultado e uma assinatura, mas não possuam corpo ou código.

Classes Abstractas são exactamente todas as classes nas quais pelo menos um ou mesmo todos os métodos de instância não se encontram implementados, mas tão só declarados sintacticamente (daí serem designados como **métodos abstractos** ou **virtuais**).

No nosso exemplo, faria então sentido que a classe `Forma` definisse a forma sintáctica dos métodos que as subclasses devem implementar, mas deixando-os vazios, ou seja, sem qualquer código, o que corresponde explicitamente à ideia de que deles não há nada para ser herdado. São portanto **métodos abstractos**.

Em JAVA, classes e métodos são declarados como **abstractos** usando o modificador `abstract` nos cabeçalhos das suas definições, conforme o exemplo apresentado a seguir, que é ao mesmo tempo a solução correcta e elegante para o nosso problema.

```
public abstract class Forma {
//
   public abstract double area();
   public abstract double perimetro();
}
```

De notar que, para que uma classe deva ser considerada **abstracta**, é suficiente que não tenha implementado apenas um dos seus vários métodos. Torna-se igualmente evidente que, por tal motivo, **uma classe abstracta não pode criar instâncias**. De facto, estas instâncias, caso pudessem ser criadas, conduziriam a situações de erro, dado não serem capazes de responder às mensagens que correspondem aos métodos abstractos, pois estes não estão implementados.

Consideremos agora a nova hierarquia de classes de `Forma` apresentada na Figura 6.2.

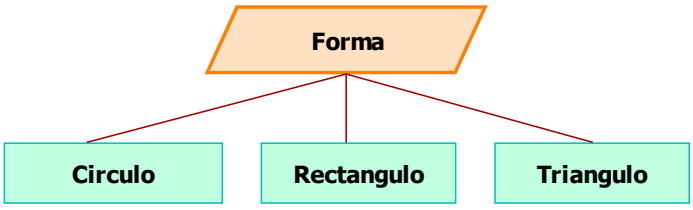

Figura 6.2 – Hierarquia com classe abstracta

Esta hierarquia vai-nos servir referencial para as considerações teóricas e práticas que iremos apresentar sobre **classes abstractas**. Nesta hierarquia, a classe `Forma` é uma classe abstracta. As classes concretas continuarão a ser representadas por rectângulos e as classes abstractas serão representadas por losangos.

A primeira questão que é usual colocar-se relativamente à existência de classes abstractas numa linguagem de PPO, é a da sua verdadeira **utilidade**, sabendo-se que, tal como vimos anteriormente, não podem sequer criar instâncias. O exemplo anterior permitiu-nos mostrar de uma forma pragmática diversas utilidades que agora vamos aprofundar.

Em primeiro lugar, deve-se chamar a atenção para o facto de que o mecanismo de herança se mantém em vigor mesmo que uma superclasse de um conjunto de subclasses seja uma classe abstracta. Logo, **uma classe abstracta é também um tipo**, e compatível com as instâncias das suas subclasses, tal como uma classe concreta, pelo que se mantêm válidas expressões de atribuição de instâncias de subclasses, tal como na expressão:

```
Forma f = new Triangulo();
```

Uma subclasse de uma classe abstracta herda automaticamente todos os métodos da classe abstracta, estejam estes implementados ou sejam abstractos. Ao herdar os métodos abstractos, a subclasse poderá implementá-los ou não. Se a subclasse implementar todos os métodos abstractos herdados, então passará a ser uma classe concreta. Se, porventura, deixar um qualquer destes métodos abstractos herdados por implementar, será igualmente uma classe abstracta tal como a sua superclasse, ou seja, não poderá igualmente criar instâncias.

Na hierarquia exemplo, a classe `Forma` é abstracta, como se apresentou, pelo que as suas subclasses `Circulo`, `Rectangulo` e `Triangulo` só não o serão também se fornecerem implementações para todos os métodos abstractos herdados.

Uma classe abstracta, ao não implementar certos (ou todos) métodos, **delega** nas suas subclasses (ou melhor **coage**) a implementação particular de tais métodos, facilitando no entanto o aparecimento de diferentes implementações dos mesmos métodos nas suas diferentes subclasses. Assim, enquanto na relação normal entre classes e subclasses a redefinição de métodos é opcional, na relação entre classes abstractas e suas subclasses, a (re)definição de métodos é uma obrigação, para que tenhamos implementações concretas da classe abstracta. No entanto, é exactamente neste ponto que reside uma das grandes vantagens de possuirmos classes abstractas.

De facto, e tomando por exemplo uma classe 100% abstracta, o mecanismo de herança vai garantir que, dado que todas as suas subclasses vão herdar o mesmo **protocolo** ou API (ou seja, o conjunto de métodos abstractos definidos apenas sintacticamente), então, salvo extensões sempre possíveis, todas as subclasses da classe abstracta responderão ao mesmo *protocolo*, ou seja, garantidamente **falam a mesma linguagem** que foi herdada (ainda que cada uma o possa fazer à sua maneira, isto é, conforme a sua implementação particular!). Formam uma "tribo" sintáctica.

Temos, portanto, desde já, duas grandes vantagens na utilização de classes abstractas. Por um lado, se todas as suas subclasses implementarem os métodos abstractos, todas responderão às mensagens cujos nomes coincidem com tais identificadores de métodos. Deste modo, consegue-se uma importante garantia de normalização de vocabulário e, por isso, a sua redução. No exemplo em questão, e admitindo que as classes `Circulo`, `Rectangulo` e `Triangulo` são concretas, teremos de imediato a certeza de que as suas instâncias responderão às mensagens `area()` e `perimetro()`. E temos também uma **classificação**, pois todas são **formas** (ex.: `Triangulo` *is_a* `Forma`).

Por outro lado, quaisquer futuras subclasses de `Forma` (ex.: `Quadrado`, `Hexagono`, etc.) devem igualmente implementar os mesmos métodos, pelo que as suas instâncias responderão às mesmas mensagens.

Uma classe 100% abstracta pode ser vista como uma especificação meramente sintáctica (ou seja, uma **assinatura de um tipo de dados**), para a qual o mecanismo automático de herança impõe a construção de implementações obedecendo à linguagem definida na assinatura, ou seja, modelos ou implementações, eventualmente algumas delas enriquecidas, isto é, aumentadas em funcionalidade, dado ser sempre possível em JAVA acrescentar funcionalidade (métodos) ao comportamento herdado.

Em resumo, **classes abstractas** são um mecanismo muito importante em PPO, dado que permitem ao programador:

- Escrever especificações sintácticas para as quais múltiplas implementações são possíveis, de momento ou no futuro;

- Normalizar a "linguagem" (API) a partir de certos pontos da hierarquia;
- Introduzir flexibilidade e generalidade no código desenvolvido;
- Tirar todo o partido do polimorfismo (via princípio da substituição);
- Realizar classificações mais claras de subclasses.

6.2 UTILIZAÇÃO DE CLASSES ABSTRACTAS

Vejamos em seguida as definições detalhadas de cada uma das classes da hierarquia em questão.

```
  public abstract class Forma {
    // Classe abstracta
    public abstract double area();
    public abstract double perimetro();
  }
```

```
import static java.lang.Math.PI;
public class Circulo extends Forma {
  // variáveis de instância
  private double raio;
  // construtores
  public Circulo() { raio = 1.0; }
  public Circulo(double r) { raio = (r <= 0.0 ? 1.0 : r); }
  // métodos de instância
  public double area() { return PI*raio*raio; }
  public double perimetro() { return 2*PI*raio; }
  public double raio() { return raio; }
}
```

A classe `Circulo` é uma subclasse concreta de `Forma`, dado ter implementado os dois métodos abstractos herdados, tendo ainda definido uma variável de instância `raio` e um método adicional `raio()`. O construtor de `Circulo` que aceita um raio como parâmetro faz a validação desse valor para que não sejam criados círculos com raios negativos ou nulos. Caso o raio seja inválido, o círculo criado possuirá raio igual a 1.

As classes seguintes, `Rectangulo` e `Triangulo`, definem igualmente as suas variáveis de instância apropriadas, bem como métodos de instância adicionais.

```
public class Rectangulo extends Forma {
  // variáveis de instância
  private double comp, larg;
  // construtores
  public Rectangulo() { comp = 0.0; larg = 0.0; }
  public Rectangulo(double c, double l) { comp = c; larg = l; }
  // métodos de instância
  public double area() { return comp*larg; }
  public double perimetro() { return 2*(comp+larg); }
  public double largura() { return larg; }
  public double comp() { return comp; }
 }
```

```
import static java.lang.Math.*;
public class Triangulo extends Forma {
  /* altura tirada a meio da base */
  // variáveis de instância
  private double base, altura;
  // construtores
  public Triangulo() { base = 0.0; altura = 0.0; }
  public Triangulo(double b, double a) {
    base = b; altura = a;
  }
  // métodos de instância
  public double area() { return base*altura/2; }
```

```
  public double perimetro() {
    return base + (2*this.hipotenusa()); }
  public double base() { return base; }
  public double altura() { return altura; }
  public double hipotenusa() {
    return sqrt(pow(base/2, 2.0) + pow(altura, 2.0)) ;
  }
}
```

Consideremos agora um pequeno programa que servirá para testar estas classes e produzir um conjunto de resultados cuja análise permitirá compreender melhor a utilização efectiva de tais classes.

```
import static java.lang.System.out;
public class TstAbsForms {
//
  public static void main(String args[]) {

   Forma forma1 = new Triangulo(2.0, 6.0);
   Forma forma2 = new Rectangulo(2.0, 13.0);
   Forma forma3 = new Circulo(3.0);
  out.println("forma1 = " + forma1.getClass().getSimpleName());
   out.printf("area = %5.2f\n", forma1.area());
   out.printf("perímetro = %5.2f\n", forma1.perimetro());
  out.println("forma2 = " + forma2.getClass().getSimpleName());
   out.printf("area = %5.2f\n", forma2.area());
   out.printf("perímetro = %5.2f\n", forma2.perimetro());
  out.println("forma3 = " + forma3.getClass().getSimpleName());
   out.printf("area = %5.2f\n", forma3.area());
   out.printf("perímetro = %5.2f\n", forma3.perimetro());
  }
}
```

O programa cria três formas distintas, uma de cada tipo, que atribui a três variáveis do tipo `Forma`, que é a superclasse das formas. Em seguida, a cada uma destas variáveis são enviadas as mensagens que correspondem à invocação dos vários métodos definidos nas suas classes.

Vejamos os resultados produzidos pela execução deste programa que, em síntese, cria uma instância de cada uma das subclasses de `Forma`, usa os métodos `getClass()` e `getSimpleName()` para confirmação da classe a que pertencem tais instâncias e em seguida testa os métodos definidos em `Forma` como abstractos e que foram redefinidos em cada uma das subclasses.

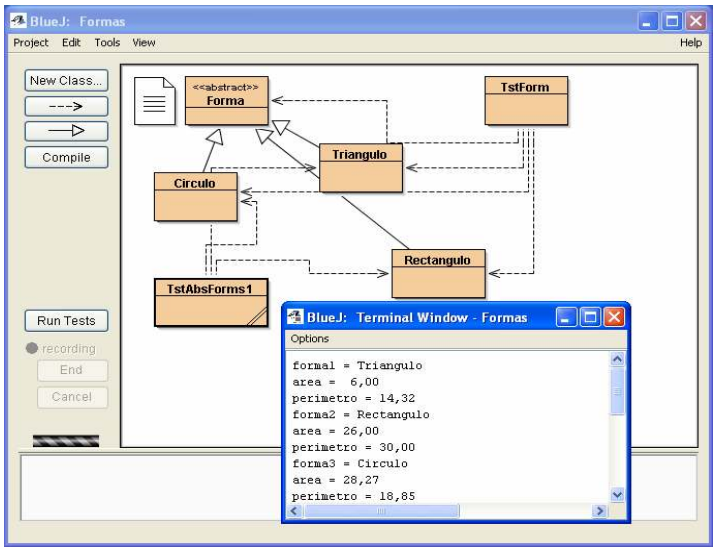

Figura 6.3 – Classe abstracta e polimorfismo

Como se pode verificar e era por nós esperado, a associação a variáveis da superclasse abstracta `Forma` de instâncias das subclasses concretas `Rectangulo`, `Triangulo` e `Circulo` não provocou qualquer erro na compilação, ou seja, do ponto de vista do compilador existe total compatibilidade sintáctica e não existe qualquer problema na execução, mesmo sendo `Forma` uma **superclasse abstracta**.

Pode portanto inferir-se que, em JAVA, uma qualquer variável declarada como sendo de uma dada classe (abstracta ou não) pode igualmente referenciar de forma correcta uma qualquer instância de uma qualquer subclasse sua, mantendo-se pois o polimorfismo e o algoritmo de *dynamic method lookup* do interpretador de JAVA.

Ou seja, o mecanismo de *dynamic binding* (isto é, associação em tempo de execução de métodos a instâncias de classes, com a correcta execução do respectivo código) funciona de forma eficaz. Assim, o interpretador é em tempo de execução capaz de distinguir qual o código a executar para o cálculo da área de um `Triangulo`, ou qual o código a executar para o cálculo da área de um `Circulo`, executando sempre o código que, de facto, corresponde ao método programado para a instância da classe em causa.

Assim, para todos os métodos abstractos declarados na classe abstracta e implementados nas suas subclasses concretas (cf. `area()` e `perimetro()` na classe `Forma`), não só o compilador reconhece a compatibilidade dos tipos, como o interpretador, através do seu mecanismo inteligente de *dynamic binding*, é capaz de seleccionar o código adequado à execução.

O problema seguinte consiste em determinar se os **métodos específicos** de uma subclasse de uma classe abstracta são reconhecidos em tempo de compilação e/ou em tempo de execução, quando associamos a variáveis do tipo da classe abstracta instâncias das suas subclasses. Isto é, tomando os exemplos anteriores, se se criar uma variável do tipo `Forma`, e se a mesma for associada a uma instância da classe `Rectangulo`, será correcta e reconhecida pelo compilador a expressão correspondente ao envio a essa variável do tipo `Forma` da mensagem `comprimento()`, cujo método está apenas definido na classe `Rectangulo`?

```
Forma f = new Rectangulo(4.0, 6.0);
int b = f.comprimento();  // erro de compilação
```

A resposta é não, e não poderia ser outra. Quer `Forma` seja classe abstracta ou concreta, a expressão `f.comprimento()` não pode ser aceite pelo compilador, porque a variável `f` é do tipo estático `Forma`, e o método `comprimento()` não está definido em tal classe, nem sequer como método abstracto.

Torna-se portanto claro, do ponto de vista da criação de classes abstractas, que será sempre importante que estas tenham declarado o maior número de métodos abstractos possível, já que esta será a **linguagem comum** a todas as subclasses concretas que vierem a ser construídas. Este será o **comportamento comum** a todas estas subclasses e, mais até do que isso, este conjunto de métodos constitui o conjunto que garante total compatibilidade de tipos via polimorfismo.

Quando as subclasses adicionam comportamento específico, ou seja, métodos próprios, a compatibilidade com o tipo da superclasse termina, pelo que o polimorfismo termina também. No exemplo anterior, o método `comprimento()` apenas poderá ser enviado a uma instância da classe `Triangulo` e não a uma `Forma`. Então, se tivermos uma instância de `Rectangulo` referenciada por uma variável de tipo estático `Forma`, teremos que fazer uma operação de *casting* para o tipo estático correcto e só depois enviar a mensagem. A porção de código correspondente seria a seguinte:

```
if(f instanceof Rectangulo) // testa tipo dinâmico
   c = (Rectangulo f).comprimento();
else
   . . . . .
```

e, portanto, perdemos a generalidade e voltamos ao código com testes caso a caso. Assim, como se pode compreender, haverá sempre que evitar a definição de métodos específicos nas subclasses, muitas das vezes aumentando os métodos abstractos das superclasses.

Em função do que acabou de ser dito no ponto anterior, torna-se evidente que a classe que serviu de exemplo à introdução das classes abstractas, `Forma`, deveria ter declarados muitos outros métodos correspondentes à funcionalidade comum a qualquer forma concreta que possa vir a ser criada. O número e definição sintáctica dos

diversos métodos que deveriam ser declarados na classe `Forma` ou qualquer outra classe abstracta, dependerá sempre de uma correcta análise de requisitos do problema em questão.

Tal como no capítulo anterior, poderíamos concluir apresentando um excerto de código que demonstre a facilidade com que certas operações que noutros contextos poderiam ser complexas de programar, podem desta forma tornar-se muito simples.

Admitamos que pretendemos calcular o somatório das suas áreas e perímetros. O código seria:

```
import java.lang.System.out;
public class TesteFormas {

  public static void main(String[] args) {

     Forma[] formas = new Forma[100];
     int numFormas = 0; // contador
     //. . .
     formas[0] = new Rectangulo(10.2, 6.8);
     formas[1] = new Circulo(2.5);
     // . . . .
     /* calcula o somatório das áreas e dos perímetros
        das várias formas inseridas no array.
     */
     double areaTotal = 0.0;
     double perimTotal = 0.0;
     for(int i=0; i < numFormas; i++) {
          areatotal += formas[i].area();
          perimTotal += formas[i].perimetro();
     }
     out.printf("Cm2 das Formas = %5.2f\n", areaTotal);
     out.printf("Perim. das Formas = %5.2f\n", perimTotal);
  }
}
```

Tal como no exemplo final do capítulo anterior, a simplicidade do código é de facto surpreendente e o que ele encerra também. Podem ser milhares as classes concretas que podem estar representadas numa frase tão simples quanto `formas[i]`. Por outro lado, ninguém sabe, lendo este código, se `Forma` é uma classe abstracta ou não. E para quê?

6.3 CLASSES ABSTRACTAS E VARIÁVEIS DE INSTÂNCIA

Centrámos a nossa atenção até agora em classes abstractas das quais o comportamento, ou seja, os métodos, são o mais importante em termos de herança. Indiscutivelmente, como acentuaremos até na secção seguinte, este é o aspecto primordial da utilização de classes abstractas.

Porém, em muitas situações, as classes abstractas surgem não tanto pelo comportamento das entidades a representar nos projectos mas mais pelos seus atributos ou variáveis de instância. É comum que surjam situações de **modelação de dados** em que no projecto de *software* é fixado e definido um conceito ainda que relativamente vago, mas já com certos atributos, e depois outros conceitos com os mesmos atributos e outros atributos adicionais surjam, sendo claramente casos particulares do primeiro. Se a idênticos atributos se juntar métodos semelhantes, haverá vantagem em criar uma classe abstracta para a entidade mais genérica colocando na classe abstracta as variáveis de instância que são comuns a todas as subclasses desta, actuais e futuras (!).

Considere-se que pretendemos criar uma base de dados contendo todos os dados sobre os VHS, CD, Hi8 e DVD que temos em casa. Todos têm, no mínimo, um título, uma data e um comentário. Cada um poderá ter atributos próprios, mas deverá ser possível calcular, para cada um, o seu tempo de duração efectivo (que não é a sua capacidade).

Fará então sentido numa situação destas que em vez de termos quatro classes desligadas entre si, tenhamos uma superclasse `ItemMulti`, de item multimédia (nome genérico) que agrega a informação comum a todas as subclasses de momento conhecidas. Claro que, ao tomar esta decisão, ela pesa sobre todas as classes futuras que vão garantidamente herdar tais atributos. Admitindo que cada item é que saberá calcular o seu tempo de duração em função da sua informação interna, o método pedido terá que ser abstracto, pelo que a classe `ItemMulti` será uma classe abstracta mas **contendo variáveis de instância**.

Na Figura 6.4 apresenta-se a hierarquia correspondente.

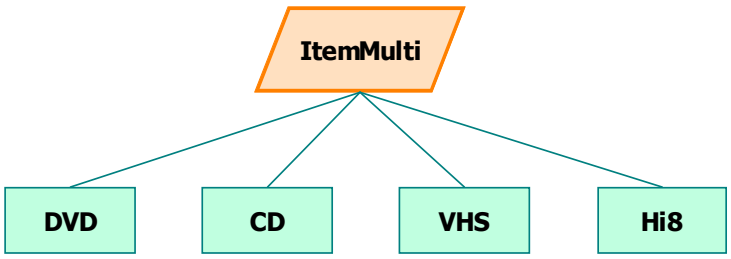

Figura 6.4 – Hierarquia de items multimédia - exemplo

Vejamos agora a primeira tentativa de declaração da classe `ItemMulti`:

```
public abstract class ItemMulti {
   // variáveis
   private String titulo;
   private String data;
   private String obs;
   // métodos para redefinição
   public abstract double duracao();
}
```

Criada a superclasse abstracta, vamos em seguida criar a classe `Hi8`, que tem como atributos adicionais a capacidade (em minutos), a percentagem de ocupação e o número de gravações. Teremos então o seguinte código:

```
public class Hi8 {
   // Variáveis de Instância
   private int minutos;
   private double ocupacao;
   private int gravacoes;
   // Construtores
   public Hi8(String titulo, String data, String obs,
              int min, double ocup, int gravc) {
      super(titulo, data, obs); // super ??
      minutos = min; ocupacao = ocup; gravacoes = gravc
   //
   . . . . . .
}
```

Claro que agora temos um problema que não havia surgido enquanto não tínhamos variáveis herdadas de classes abstractas: as subclasses necessitam de inicializar essas variáveis através dos construtores da superclasse. Mas, se a superclasse é abstracta, para que necessita de construtores dado que não cria instâncias? A resposta está dada através deste exemplo.

Uma classe abstracta poderá definir construtores, não para criar instâncias, mas para que estes sejam invocados pelas suas subclasses quando estas necessitam de criar as suas próprias instâncias.

Tal como para as classes concretas, para as classes abstractas que definam variáveis, as subclasses, ao definirem os seus construtores, terão que invocar os construtores da superclasse para que os valores das variáveis da superclasse (neste caso, abstracta) sejam correctamente inicializados. Estes construtores `super()` devem mesmo ser as

primeiras instruções do código dos construtores das subclasses. Por este motivo, caso não sejam definidos na classe abstracta as instâncias das subclasses, estarão mal formadas.

Então, o código final e correcto da classe `ItemMulti` será:

```
public abstract class ItemMulti {
   // variáveis
   private String titulo;
   private String data;
   private String obs;
   // construtores
   public ItemMulti() {
     titulo = ""; data = ""; obs = "";
   }
   public ItemMulti(String tit, String dat, String com) {
     titulo = tit; data = dat; obs = com;  }
   // método abstracto definido
   public abstract double duracao();
}
```

Apresenta-se na Figura 6.5 o resultado da criação de uma instância de Hi8, onde se podem ver os campos que são da responsabilidade do construtor da classe `ItemMulti` devidamente inicializados com os valores dados como parâmetro em `Hi8(..)`. A não definição em `Hi8` do método `duracao()` conduziria a um erro de compilação.

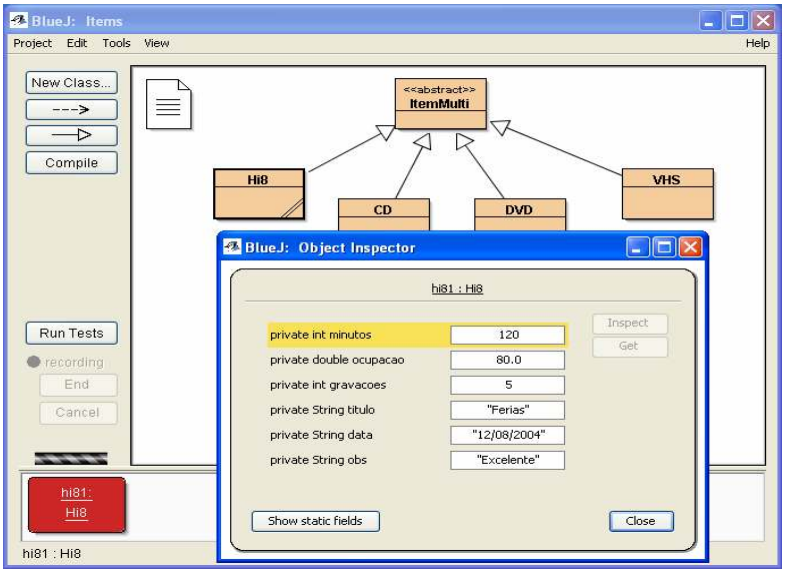

Figura 6.5 – Criação de instâncias de subclasses de `ItemMulti`

6.4 CONSIDERAÇÕES FINAIS E EXEMPLOS

Como mecanismo de abstracção, as classes abstractas permitem que a concepção e o desenvolvimento de *software* possam ser realizados segundo uma metodologia baseada em **refinamento progressivo por especialização**, ou seja, construindo gradualmente classes mais concretas até se possuirem implementações completas, ao contrário das metodologias mais tradicionais que aconselhavam um refinamento progressivo por decomposição ou subdivisão dos problemas.

As classes abstractas tendem naturalmente a ocupar os níveis mais elevados das hierarquias de classes, dado serem muito pouco físicas e muito mais conceptuais (especificações).

A Figura 6.6 mostra uma pequena hierarquia de classes JAVA tendo no topo uma classe abstracta:

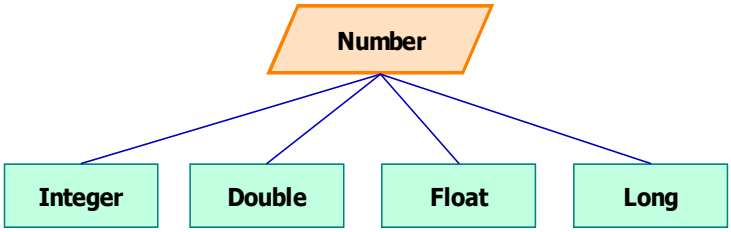

Figura 6.6 – Classe abstracta Number de JAVA

As classes abstractas são por vezes também utilizadas, não tanto com a finalidade de "normalizar" protocolos, mas como máximos denominadores comuns de classes para as quais não é facilmente verificável uma relação de inclusão, isto é, de subclasse, dado que, por exemplo, estas são apenas implementações distintas, particularmente em estrutura, ou seja, em atributos, de um mesmo problema.

Ainda que, na maior parte das vezes, a classe abstracta seja 100% abstracta, ou seja, nada possua implementado que possa ser herdado pelas suas implementações, em certas situações faz todo o sentido, e é perfeitamente possível, colocar na classe abstracta partes de código (a situação mais comum) ou mesmo constantes e/ou variáveis que, desta forma, são comuns a todas as suas possíveis implementações nas subclasses. A classe continuará a ser abstracta desde que algum método de instância ainda o seja.

No entanto, deve notar-se que colocar atributos ou código concreto numa classe abstracta implica que todas as futuras subclasses desta irão herdar tal código e/ou tais variáveis. Será que, de facto, todas as possíveis actuais e futuras subclasses concretas da classe abstracta têm "mesmo" em comum tais variáveis e tais métodos? Depende dos projectos.

Porém, quando tal fizer sentido, quanto mais funcionalidade puder ser colocada numa superclasse (quer esta seja ou não abstracta), melhor optimização do espaço funcional estamos a fazer, dado que não será necessário replicar tal código em cada uma das implementações (ou subclasses) da superclasse. O mecanismo de herança garante a acessibilidade de qualquer instância a tal código partilhado. O mesmo se poderá dizer relativamente a variáveis de instância, ainda que, como sabemos agora, a partilha de estrutura (os dados) seja bastante mais complexa e de difícil alteração.

As classes abstractas de JAVA são, de uma forma geral, 100% abstractas, ou seja, nenhum código ou estrutura de dados comum é nestas colocado. Desde a versão 1.2 de JAVA (JAVA2), com a introdução de classes que representam diversos tipos de colecções de objectos (JCF − *Java Collections Framework*), o papel das classes abstractas, enquanto classes que representam especificações sintácticas a obedecer pelas subclasses que vão realizar as implementações, passou a ser mais claro e importante de ser estudado.

Por exemplo, no package `java.util` foram definidas algumas classes abstractas que especificam o comportamento de uma qualquer lista, através de `AbstractList`, de um qualquer conjunto, cf. `AbstractSet`, ou correspondência, cf. `AbstractMap`.

A Figura 6.7 mostra a mini-hierarquia dos conjuntos. As subclasses `HashSet`, `TreeSet` e `EnumSet` são três implementações distintas da classe abstracta. Têm em comum a sua API, ou seja, todas implementam as mesmas operações definidas na sua superclasse abstracta `AbstractSet`.

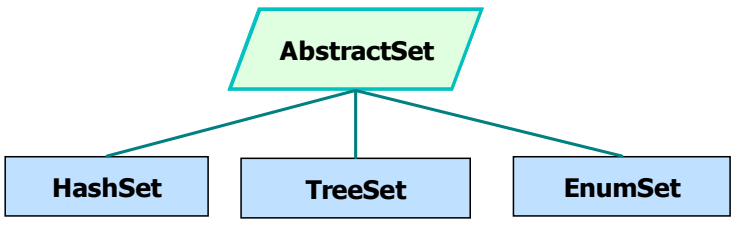

Figura 6.7 – Classe abstracta de JAVA e suas implementações

Outro aspecto curioso desta hierarquia é que a classe abstracta `AbstractSet` é uma subclasse da classe abstracta `AbstractCollection`. Poderia parecer, à partida, não fazer sentido colocar uma classe abstracta como subclasse de outra abstracta. Porém, e tomando por base a hierarquia exemplo, torna-se fácil compreender que o objectivo fundamental é que qualquer classe concreta que seja subclasse da classe `AbstractSet`, dado `AbstractSet` ser subclasse de `AbstractCollection`, se veja igualmente obrigada a implementar métodos abstractos definidos em `AbstractCollection`. Deste modo, as instâncias da classe `TreeSet`, por exemplo, não só respondem aos métodos que têm a ver com o facto de serem **colecções de objectos**, como também aos métodos que dizem respeito ao facto de serem **conjuntos**, ou seja, não terem ordem explícita nos seus elementos e não admitirem duplicados.

Veremos em capítulo apropriado que estas classes de JAVA implementam **colecções de objectos** são desde JAVA5 classes genéricas, ou parametrizadas, ou seja, cada uma delas representa um conjunto de tipos. Por exemplo, a classe que representa todos os possíveis treesets é representada por `TreeSet<E>` em que `E` é uma variável que será pelo programador associada a um dado tipo simples ou não. Assim, vamos poder ter, a partir da mesma classe e conforme a instanciação de `E`, os tipos concretos `TreeSet<String>`, `TreeSet<Ponto>`, `TreeSet<Circulo>`, etc. Todos são no entanto **sets** e como tal se irão comportar.

As classes abstractas têm por vezes também a missão de implementar parcialmente métodos genéricos aplicáveis a todas as suas subclasses, ou realizar inicializações que são também úteis. As suas subclasses, em tais casos, podem limitar-se a complementar ou refinar tais métodos, tendo assim o seu esforço de codificação parcialmente reduzido.

6.5 SÍNTESE DO CAPÍTULO

Como se pode agora compreender, as classes abstractas são um mecanismo muito importante em PPO, não apenas pela normalização sintáctica que impõem, mas também pelo facto de que representam especificações de comportamento para as quais várias implementações podem ser, a qualquer momento, desenvolvidas. Estas características particulares, associadas ao princípio da substituição e inerente polimorfismo, garantem um grau de extensibilidade ímpar, dado que a criação de novas classes concretas, que correspondem a novas implementações das classes abstractas, pode ser facilmente realizada pois não tem implicações sobre o código até aí desenvolvido.

Pragmaticamente, a utilização de uma classe abstracta tem sentido sempre que tal comportamento é específico de um conjunto de classes que são hierarquicamente relacionadas, ou seja, que o devem herdar, mas não quando esse comportamento é de tal forma genérico que pretenderíamos que o mesmo pudesse ser implementado livremente por qualquer classe em qualquer ponto da hierarquia.

JAVA possui ainda um outro muito interessante mecanismo que permite associar implementações realizadas em classes concretas a especificações abstractas de tipos de dados. Este mecanismo, que será estudado em seguida, é o mecanismo em JAVA mais próximo da noção formal de **assinatura** de um TAD (tipo abstracto de dados), ou seja, a definição sintáctica do conjunto de operações que podem ser realizadas sobre os valores de tal tipo (que não classe!), designando-se em JAVA por **interfaces**.

As **interfaces** são especificações sintácticas (abstractas) idênticas às especificações das classes abstractas, mas que podem ser implementadas **por qualquer classe em qualquer ponto da hierarquia**, sendo também verdade que, ao contrário das classes abstractas, as **interfaces** não fazem parte da hierarquia de classes.