5 - Hierarquia de Classes e Herança

5.1 A hierarquia de classes

A colocação das classes numa hierarquia de especialização (do mais genérico para o mais detalhado) para efeitos de reutilização de variáveis e métodos, bem como o conjunto de mecanismos de acesso e compatibilidade de tipos associados, são conceitos únicos e dos mais importantes das linguagens de PPO. Porém, são também dos mais complexos de compreender e utilizar plenamente, tanto mais que hierarquizar classes implica algo que mesmo no nosso dia-a-dia nos é muito difícil fazer: **classificar**.

Vamos, por isso, apresentar a noção de hierarquia de classes não através de uma simples definição, como é usual, mas antes partindo de uma situação em que tal possibilidade não exista, e analisando as circunstâncias em que a sua inclusão numa linguagem se justifica e em que, portanto, pode e deve ser aplicada com enormes vantagens para o programador.

Vimos anteriormente que o mecanismo de **composição** de classes nos surge como um primeiro mecanismo muito importante de reutilização de classes já definidas. Usando tal mecanismo, que é no dia-a-dia muito comum, como em linhas de produção industrial onde diferentes tipos de partes são juntas para formar o todo, vamos agora analisar se é ou não vantajoso que outros relacionamentos possam ser definidos entre classes.

De momento, mais nenhuma forma de relacionamento entre classes está definida em PPO. Então, as classes ao serem criadas podem usar outras classes por composição, mas, para além disso, são entidades isoladas umas das outras, todas posicionadas no mesmo espaço plano, conforme se procura ilustrar na figura seguinte, que é equivalente às usuais **bibliotecas** de muitas linguagens de programação.

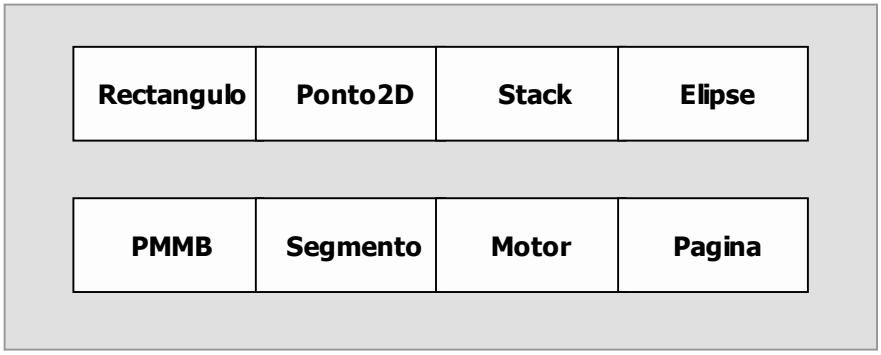

Figura 5.1 – Espaço plano de classes

Ao observarmos as classes apresentadas na Figura 5.1, é natural que, relativamente a algumas delas, não consigamos encontrar qualquer tipo de semelhança, afinidade ou qualquer outra particularidade comum. Tal é o caso, por exemplo, entre a classe `Motor` e a classe `Stack`, ou entre a classe `Ponto2D` e a classe `PMMB`.

Porém, tal exercício deve ser sim realizado perante o problema concreto de criarmos uma **nova classe** e verificarmos até que ponto alguma das classes existentes contém já grande parte da estrutura e do código que nos pode ajudar a definir a classe que pretendemos.

Vamos então considerar que nos era pedido para criarmos as quatro classes que se apresentam na Figura 5.2, sem quaisquer tipos de imposições quanto à sua implementação, tentando reutilizar algumas das classes apresentadas acima quer por composição quer por outro mecanismo qualquer (inclusive *copy&paste*, muitas vezes usado).

Figura 5.2 – Classes a definir

Em primeiro lugar, para realizar tal análise, temos que conhecer muito bem a estrutura das classes que existem, ou seja, como estão definidas as variáveis de instância em forma e tipo, que funcionalidade têm implementada, e saber o que precisamos de ter na classe que estamos a definir.

A classe **`Triangulo`** pode ser naturalmente definida de várias maneiras, e, olhando para as classes existentes e sabendo o que é um segmento (representado por dois pontos com coordenadas inteiras), poderíamos cair de imediato na tentação de dizer que:

1 triângulo ⇔ 3 x 1 segmento

ou seja, **usando composição**, a classe `Triangulo` poderia ser definida como:

```
public class Triangulo {
   // Variáveis de Instância
   private Segmento lado1;
   private Segmento lado2;
   private Segmento lado3;
}
```

No entanto, esta composição é delicada já que, em seguida, teríamos que garantir, através dos construtores da classe, que os pontos que constituem os segmentos satisfazem um conjunto de propriedades, de forma a que os segmentos correspondam de facto aos três lados de um triângulo. Tal só acontece para certas combinações dos seis pontos. A solução é portanto bastante simples estruturalmente mas complexa de validar.

Outra hipótese mais simples, também baseada em composição, seria considerarmos que um triângulo pode ser definido por um segmento e um ponto, como, por exemplo:

1 triângulo ⇔ 1 segmento + 1 ponto2D

definindo por **composição** a classe `Triangulo` como,

```
public class Triangulo {
   // Variáveis de Instância
   private Segmento lado;
   private Ponto2D vertice;
}
```

havendo agora apenas que garantir que `vertice` seja diferente dos pontos de `lado`.

Claro que, perante estas hipóteses, em que pouco há a ganhar com as reutilizações, em especial de `Segmento`, poderíamos pura e simplesmente tomar a decisão final de definir a classe `Triangulo` como sendo:

1 triângulo ⇔ 3 x 1 ponto2D

isto é,

```
public class Triangulo {
   // variáveis de instância
   private Ponto2D p1, p2, p3;
}
```

Vamos agora passar para a classe **`Ponto3D`**. O nome é sugestivo, tanto mais que já temos uma classe `Ponto2D` e não temos requisitos especiais. Assim sendo, vamos considerar que os nossos pontos 3D vão possuir coordenadas inteiras. Necessitaremos portanto de possuir três variáveis inteiras, uma para cada coordenada, e métodos para trabalhar com as três variáveis.

Ora a classe `Ponto2D` tem já implementado grande parte de tudo o que necessitamos para implementar `Ponto3D`, ou seja, tem já duas variáveis correspondentes a duas coordenadas (x e y) e tem já métodos para trabalhar com estas duas coordenadas. Precisaremos de mais uma variável para a terceira coordenada, seja z. Precisaremos também de métodos que usem esta terceira coordenada.

Portanto, quase que poderíamos dizer que, em termos quantitativos, cerca de 75% do nosso trabalho para criar a classe `Ponto3D` do zero está já codificado na classe `Ponto2D`, desde que encontremos uma forma de, a partir dela, criarmos a classe pretendida.

Em termos de equações com classes e instâncias, teríamos, em síntese:

$$\mathbf{Ponto3D = Ponto2D + \Delta_{prog}}$$

$$\mathbf{1\ ponto3D \Leftrightarrow 1\ ponto2D + \Delta_{var} + \Delta_{met}}$$

Ou seja, para termos uma classe `Ponto3D` completa precisamos de **possuir tudo o que já existe em Ponto2D e adicionar-lhe o que falta em termos de variáveis e métodos**, por forma a que uma instância de `Ponto3D` seja igual a uma instância de `Ponto2D` mas tendo mais essas variáveis e respondendo a mais essas mensagens. A classe `Ponto3D` aumenta, estende, detalha, refina, especializa a classe `Ponto2D`, podendo aproveitá-la por completo.

Precisamos portanto de um mecanismo de **inclusão total** de uma classe noutra, para que possamos reutilizar completamente uma classe já existente na definição de outra, tal como o exemplo aconselharia e mostra ser de grande utilidade.

Claro que, se tal mecanismo não existir, tal como num espaço plano de classes como o da Fig. 5.1, então a definição da nova classe `Ponto3D` só poderia ser feita de duas formas possíveis:

- Usando um mecanismo de "copy&paste&edit" a partir de `Ponto2D`;
- Usando em Ponto3D uma variável de instância que da classe `Ponto2D`.

Ainda que em ambas as soluções exista de facto reutilização, no primeiro caso via o mecanismo mais básico possível, e, no segundo, via composição, torna-se fácil de compreender que ambas as soluções são muito pobres em face daquela que acabámos de vislumbrar, pois ambas implicam duplicação de código.

Acresce ainda o facto de que, enquanto no exemplo foi de imediato afirmado que as classes possuem grandes afinidades, num sistema plano de classes o custo de se saber se existe alguma classe semelhante àquela que pretendemos criar é imenso, já que as classes não possuem qualquer mecanismo particular de classificação a não ser o seu próprio identificador. Tal constatação deveria ser feita por análise da sua API, ou, na pior das hipóteses, do seu código fonte.

Torna-se portanto muito importante criar um mecanismo que facilite a definição de classes à custa de classes existentes, não por composição, mas antes permitindo a inclusão de uma na outra, baseado nas noções de similaridade e especialização ou particularização, tornando-se assim um mecanismo adicional de relacionamento entre classes.

As linguagens de PPO introduzem, de facto, um espaço para a definição de classes, não plano mas **hierárquico**, cujo objectivo fundamental é que este relacionamento estabelecido entre as classes em tal hierarquia seja equivalente a um mecanismo automático de reutilização de código (Figura 5.3).

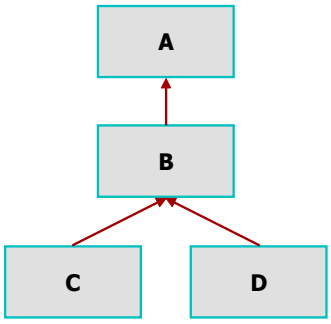

Figura 5.3 – Hierarquia de classes

Na Figura 5.3 apresenta-se uma parte de uma hierarquia de classes, em particular a sub-hierarquia estabelecida entre as classes **A**, **B**, **C** e **D**. A classe **B** é **subclasse** directa de **A**, por ocupar na hierarquia uma posição imediatamente inferior a **A**. A classe **A**, por seu lado, é **superclasse** de **B**. As classes **C** e **D** são subclasses (directas) de **B** e também subclasses (indirectas) de **A**. Assim, **B** é subclasse de **A** e superclasse de **C** e de **D**. As classes **C** e **D** não possuem subclasses.

Em certas linguagens (ex.: C++) a superclasse de uma dada classe é designada por **classe base** e a subclasse por **classe derivada**. Mas tudo isto é apenas sintaxe.

Em certas linguagens, tal como em JAVA, qualquer classe da hierarquia terá no máximo **uma superclasse**, enquanto que noutras, como em C++, uma classe pode ser subclasse de mais do que uma classe e, portanto, possuir mais de uma superclasse. Por tal razão a hierarquia de classes de JAVA diz-se uma **hierarquia simples** enquanto que a hierarquia de C++ se designa por **hierarquia múltipla**.

Uma hierarquia simples como a de JAVA deverá ter no topo da hierarquia uma classe que será a **superclasse de todas as classes** (cf. Figura 5.4).

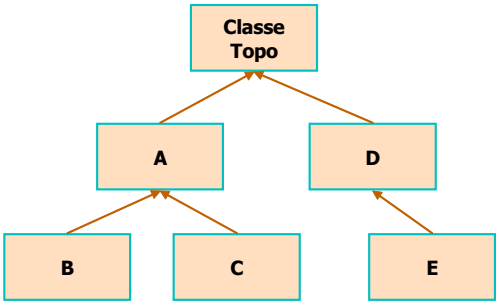

Figura 5.4 – Hierarquia típica em JAVA

Mas o que significa, do ponto de vista semântico e do relacionamento entre as classes, este posicionamento hierárquico das classes? Isto é, que significado atribuir ao facto de uma classe ser subclasse de outra, ou seja, posicionar-se abaixo daquela na hierarquia e que consequências advêm desse facto?

Na realidade, a hierarquia de classes em PPO é uma hierarquia de **especialização**, pelo que uma subclasse de uma dada classe é uma extensão, refinamento ou especialização desta, sendo, por isso, em geral mais detalhada ou refinada do que a sua superclasse, seja por possuir mais estrutura de dados seja por possuir mais comportamento, ou ambos.

Como sabemos do estudo de outras taxinomias, a classificação do conhecimento (e não só) é realizada do geral para o particular, seguindo uma semântica de especialização. A Figura 5.5 ilustra uma taxinomia deste tipo.

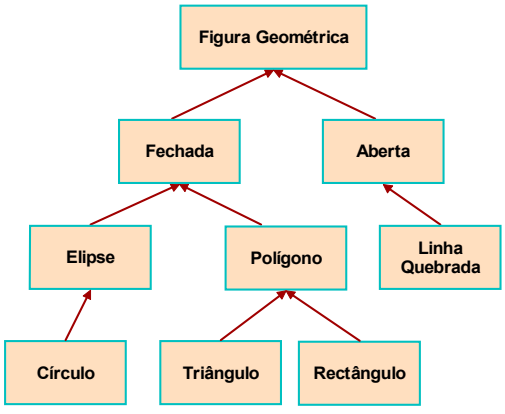

Figura 5.5 – Taxinomia por especialização

No entanto, há que tomar em atenção que muitas destas taxinomias com que por vezes somos confrontados, possuem uma característica muito particular que as torna bastante diferentes da hierarquia com que teremos que lidar em PPO e que consiste no facto de serem taxinomias fundamentalmente baseadas em atributos, ou estrutura, mas que não contemplam comportamento.

A hierarquia de classes em PPO é uma hierarquia baseada em especialização, mas na qual esta pode ser simultaneamente estrutural e comportamental, ou seja, em que a subclasse pode necessitar de mais estruturas de dados do que a sua superclasse para a sua representação e/ou pode necessitar de aumentar o conjunto de métodos que representam o comportamento da sua superclasse. Como se pode compreender facilmente, um aumento da estrutura de dados implica sempre que se devam acrescentar métodos, no mínimo os de acesso e os de modificação das novas variáveis criadas.

Ainda antes de respondermos a esta questão, o que sabemos é que, qualquer que seja a forma de reutilizarmos o código que pretendemos, as nossas classes `Ponto2D` e `Ponto3D` se vão relacionar hierarquicamente conforme se pode observar na Figura 5.6:

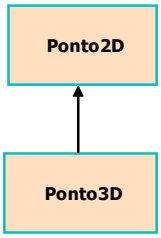

Figura 5.6 – Classe Ponto2D e subclasse Ponto3D

Em JAVA a declaração de que **uma classe B é subclasse de A** dá exactamente a ideia de que a subclasse aumenta ou acrescenta à superclasse, ao escrever-se no cabeçalho (no caso de se tratar de uma classe pública), o seguinte código:

```
public class B extends A { …
public class Ponto3D extends Ponto2D { …
```

Se uma classe B é, numa hierarquia, subclasse de A, então, B é uma especialização de A. Este relacionamento designa-se por **"é-um"** ou **"é-do-tipo"** (*is-a* em inglês) na área das metodologias por objectos, o que tem um significado muito importante. Ao dizer-mos, por exemplo, que "um triângulo **é-um** polígono" (ver Figura 5.5), tal significa que um triângulo possui os atributos característicos de um qualquer polígono aos quais acrescenta os seus próprios. Logo, **é um** polígono, e dizemos até que **é um tipo de polígono**.

Por outro lado, e pensando de uma forma menos atributiva e mais operacional, onde quer que seja solicitado um polígono, será que poderemos enviar em "representação" desse tal polígono um triângulo? Logicamente que sim porque tudo o que alguém pretender fazer com o polígono pode fazer com o triângulo pois não houve perda nem de atributos nem de comportamento com a troca (eventualmente o contrário).

Como se disse atrás, a hierarquia de classes de JAVA é uma hierarquia simples, pelo que cada classe tem uma e uma só superclasse, existindo uma classe única no topo da hierarquia. Sendo a hierarquia de especialização, esta classe de topo deverá ser a classe mais genérica de todas. Conforme veremos na secção seguinte, esta classe designa-se em JAVA por **`Object`** e possui, apesar de ser muito geral, características importantes para o funcionamento da linguagem.

Qualquer das classes que até agora definimos e compilámos com cabeçalhos sem qualquer cláusula **`extends`**, posiciona-se de imediato como subclasse directa de `Object`.

```
public class PMMB {      // é subclasse de Object
public class Segmento { // é subclasse de Object
```

Vendo-se a partir de agora uma subclasse como sendo uma extensão, um aumento, em estrutura e comportamento, da sua respectiva superclasse, mais pertinente se torna colocar a questão, dado não estar ainda resolvida, de que

forma a subclasse pode reutilizar a estrutura e o código da sua superclasse, que serão úteis para a sua definição, já que, cf. a palavra "especialização" indica, ambas as classes terão muito em comum.

A resposta a esta questão é dada em PPO pela introdução de um mecanismo que se designa por **herança,** e que é um mecanismo fundamental à reutilização, à partilha de código e ao estabelecimento de um estilo de programação incremental, ou seja, um estilo baseado numa programação, não do todo, mas apenas de extensões ao automaticamente herdado. Este mecanismo está automaticamente associado à hierarquia de classes.

5.2 O MECANISMO DE HERANÇA

Se uma classe **B é subclasse de A**, então:

1. **Se B pertence ao mesmo package de A**, B herda de A todas as variáveis e métodos de instância que não são `private`;
2. **Se B não pertence ao package de A**, B herda de A as variáveis e métodos de instância que não são `private` ou `package` (herda `public` e `protected`);
3. **B pode definir** novas variáveis e métodos de instância próprios;
4. **B pode redefinir** variáveis e métodos de instância herdados;
5. Variáveis e métodos de classe não são herdados mas podem ser redefinidos;
6. Construtores não sendo membros de classe não são herdados.

Comecemos por deixar claro que, em bom rigor, **herdar** e **ter acesso a** não são em JAVA a mesma coisa. De facto, em geral, uma classe herda muito mais do que aquilo a que pode aceder. Porém, como, para quem programa, o que de facto tem valor é aquilo a que pode aceder, caso contrário tem um erro de compilação, torna-se assim mais cómodo dizer que **o que é herdado é apenas o que é tornado acessível**. Tal corresponde a uma perspectiva conceptual correcta e pragmaticamente mais simples, que aqui estamos a aplicar, pois facilita muito a compreensão do mecanismo. **Assim, e até afirmação em contrário, uma classe herda exactamente aquilo a que tem acesso**.

Temos, portanto, ao nosso dispor um mecanismo automático de reutilização de código, que vamos passar a analisar em detalhe, tomando especial atenção relativamente aos seguintes pontos:

- **Redefinição de métodos e variáveis;**
- **Procura de métodos;**
- **Criação das instâncias das subclasses;**

5.2.1 ALGORITMO DE PROCURA DE MÉTODOS

Se uma subclasse **B** de uma classe **A** herda automaticamente os métodos de **A**, tal significa que qualquer instância de **B** vai poder responder não só às mensagens que correspondem aos métodos definidos na sua classe **B**, mas também, de forma automática, a mensagens que dizem respeito à activação de métodos que estão definidos na sua superclasse **A.** Assim, uma instância de uma dada classe, por herança, passa a poder receber e a ser capaz de responder não só às mensagens correspondentes à sua própria API, mas também às mensagens que correspondem à activação dos métodos herdados e acessíveis da sua superclasse **A**.

Por outro lado, **a herança é transitiva**, isto é, a própria classe **A** herda da sua superclasse e esta da sua, e assim até ao topo da hierarquia. A única classe que não vai herdar de nenhuma outra será a classe de topo da hierarquia, a classe **`Object`**.

Deste modo, o conjunto real de mensagens a que uma instância de uma dada classe vai ser capaz de responder, é formado pelas mensagens que activam os seus métodos locais (definidos na sua classe) às quais se somam as mensagens que correspondem a todos os métodos herdados de todas as suas superclasses até ao topo da hierarquia.

Assim, quando uma instância recebe uma mensagem, o método é procurado de imediato na sua classe e, se for encontrado, é executado. Caso não seja encontrado, é procurado na superclasse desta, e assim recorrentemente até ser atingida a classe **`Object`**. Não sendo encontrado, finalmente, na classe de topo, um erro de execução será então gerado.

5.2.2 SOBREPOSIÇÃO DE MÉTODOS E VARIÁVEIS REFERÊNCIAS `this` E `super`

Sendo o mecanismo de herança automático, tal significa que uma dada classe **herda obrigatoriamente** da sua superclasse directa e das transitivas, um conjunto de métodos de instância cujo código, como vimos anteriormente, é acessível às suas instâncias em função dos respectivos modificadores de acesso.

Que fazer, no entanto, quando uma classe herda um método cuja definição não lhe serve? A solução é realizar localmente a **redefinição** de tal método, desta forma sobrepondo à definição herdada uma definição local, sendo a definição local prioritária, ou seja, a primeira a ser encontrada e executada quando a mensagem é enviada a uma instância da classe.

Em PPO, quando um método de instância de uma classe é redefinido numa sua subclasse, diz-se que é **reescrito** ou **sobreposto** (*overriden*). Quando o identificador de uma variável de instância é declarado numa sua subclasse diz-se que a variável é **escondida** (*hidden* ou *shadowed*)**.**

A questão seguinte que se pode colocar, é saber se, ao redefinir o método da superclasse, a subclasse perde em definitivo o acesso a esse método. As duas classes seguintes vão permitir responder à maioria das questões.

Antes porém, convém relembrar que, para o compilador de JAVA, todas as decisões sobre redefinição ou não de métodos, são realizadas tendo por base as assinaturas destes, e que a **assinatura** de um método consiste do seu nome, número e tipo dos seus parâmetros. Em certas circunstâncias, mesmo tendo a mesma assinatura, um método não poderá redefinir outro, devido aos modificadores de acesso ou por outras razões extra-assinatura.

Consideremos então a seguinte classe **`Super`**:

```
public class Super {
   protected int x = 10;
   String nome;
   // Métodos
   public int getX() { return x; }
   public String classe() { return "Super"; }
   public int teste() { return this.getX(); }
}
```

e uma sua subclasse **`SubA`** com o seguinte código:

```
public class SubA extends Super {
   private int x = 20; // "shadow"
   String nome;        // "shadow"
   // Métodos
   public int getX() { return x; }
   public String classe() { return "SubA"; }
   public String supClass() { return super.classe(); }
   public int soma() { return x + super.x; }
}
```

Note-se em primeiro lugar que as variáveis de instância de `Super` não são `private`, logo são herdadas, quer sejam `protected` quer sejam *default* (*package*). Note-se também que nenhuma das classes possui construtores. A subclasse redefine as variáveis `x` e `nome`, e faz sobreposição dos métodos `getX()` e `classe()`.

O método `supClass()` de `SubA` é um método que deverá dar como resultado o valor da variável `nome` da superclasse. Se a subclasse não tivesse redeclarado `nome`, bastaria escrever `return nome;`, porque a variável não sendo `public` é herdada. Se escrevessemos `this.classe()` num qualquer método de `SubA e SubA` não tivesse redefinido o método `classe()`, então o método `classe()` não seria encontrado em `SubA`, seria procurado em `Super` e daria como resultado a *string* `"Super"`. Mas havendo um método local a `SubA` com o mesmo nome, então, **`this.classe()`** invoca o método local a `SubA`, que dá como resultado `"SubA"`. O código **`super.classe()`** corresponde à invocação do método `classe()` da superclasse, deste modo não se perdendo o acesso a ambos os métodos a partir da classe `SubA` que redefiniu o método herdado.

A referência **`super`**, usada como **`super.m()`**, onde quer que seja encontrada, remete a procura desse método **`m()`** para a **superclasse do receptor da mensagem inicial**.

Deste modo, apesar da possibilidade de sobreposição, quer o método local quer o método herdado são acessíveis ao código dos métodos de instância da classe.

O BlueJ pode também ajudar-nos na continuação do exemplo. Na Figura 5.7 pode ver-se a hierarquia com as duas classes definidas.

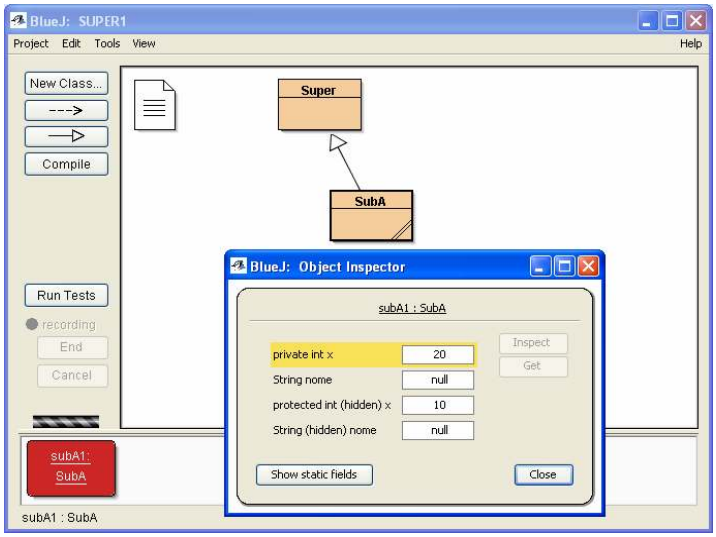

Figura 5.7 – Subclasse em BlueJ

A seta representativa de subclasse é colocada automaticamente logo que a classe **`SubA`** é compilada com a cláusula **`extends Super`** no seu cabeçalho. Na figura, foi já criada uma instância da classe `SubA`, de nome **`subA1`** da qual se inspecciona o estado interno, sendo de notar que o BlueJ indica que duas das variáveis herdadas estão *hidden*, ou seja, foram redefinidas pelas duas apresentadas acima. Note-se ainda que os campos `String` estão com o valor `null` pois JAVA usou os construtores por omissão, já que as classes não possuem construtores definidos.

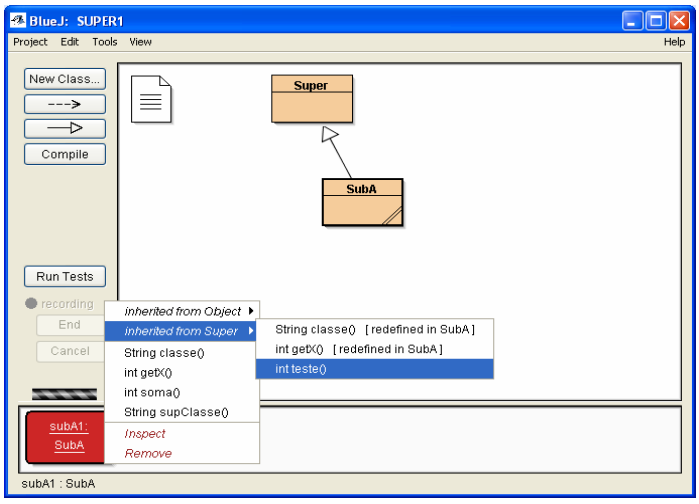

Figura 5.8 – Indicação de redefinição de métodos

Na Figura 5.8 verifica-se que é dada informação relativa aos métodos que foram herdados (*inherited*) e redefinidos na classe **`SubA`**.

Finalmente, o resultado de enviarmos a mensagem **`subA1.supClasse()`** é, conforme se pretendia, a que é apresentada na Figura 5.9.

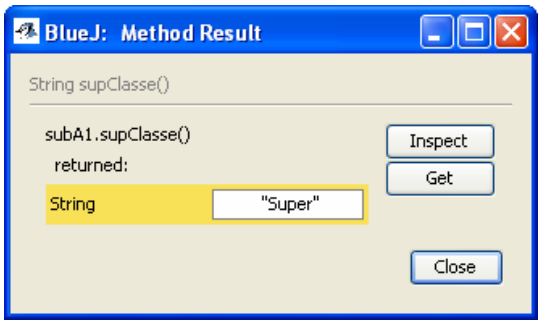

Figura 5.9 – Utilização de `supClasse()`

O mecanismo de herança, em particular através do respectivo algoritmo de procura de métodos, permite que a instância da classe `SubA` tenha acesso ao código do método do qual apenas existe uma cópia na classe `Super`, mas que, deste modo, é partilhável por todas as classes que o herdem. Há reutilização pelo facto de que tal código foi apenas definido uma vez, mas pode ser também usado por todas as classes que o herdam ou venham a herdar ao serem criadas como subclasses desta.

Note-se que, segundo a nossa metodologia de programação, as variáveis de instância são para nós um problema de herança a menos, já que todas elas são `private`. Assim, para aceder às mesmas, qualquer subclasse tem ao seu dispor os métodos públicos de consulta que são herdados.

A referência `super` não pode ser usada na mesma expressão mais do que uma vez, pelo que são incorrectas as expressões seguintes que pretenderiam aceder ao método `teste()` e à variável `z` duas classes acima da classe do receptor, como em:

```
super.super.teste();   // expressão incorrecta
super.super.z;         // expressão incorrecta
```

Vamos agora ver o que sucede quando enviamos a `subA1` a mensagem `teste()` que não faz parte das mensagens a que a classe `SubA` responde, pelo que o algoritmo de procura vai procurar o código do método na superclasse desta. Como se pode verificar, o código de `teste()` na superclasse é: `return this.getX();`. Ora, `getX()` em `Super` está codificado como `return x;` e `x` vale 10. Assim, o resultado esperado será 10. Vejamos o resultado da expressão **`subA1.teste();`** dado pelo BlueJ (ver Figura 5.10).

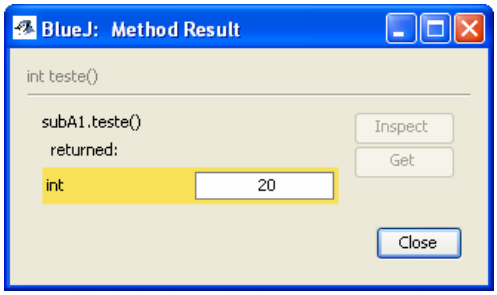

Figura 5.10 – Resultado de `subA1.teste()`

Este resultado, aparentemente surpreendente, deve-se à interpretação que deve ser feita de uma expressão do tipo **`this.m()`** ou do tipo **`super.m()`** quando encontrada no código de um método pertencente a uma qualquer classe da hierarquia. Como vimos atrás, quando as expressões são encontradas no código de um método da mesma classe do receptor da mensagem, então, elas referem o método **`m()`** local ou da superclasse. Porém, e se o objecto que recebeu a mensagem inicial não é uma instância da classe onde a expressão foi encontrada? Nesse caso, quem é `this`, sabendo-se que quando é assim usado refere sempre uma instância?

Vamos seguir o algoritmo de procura de métodos usando a Figura 5.11, que ilustra os vários passos de tal procura.

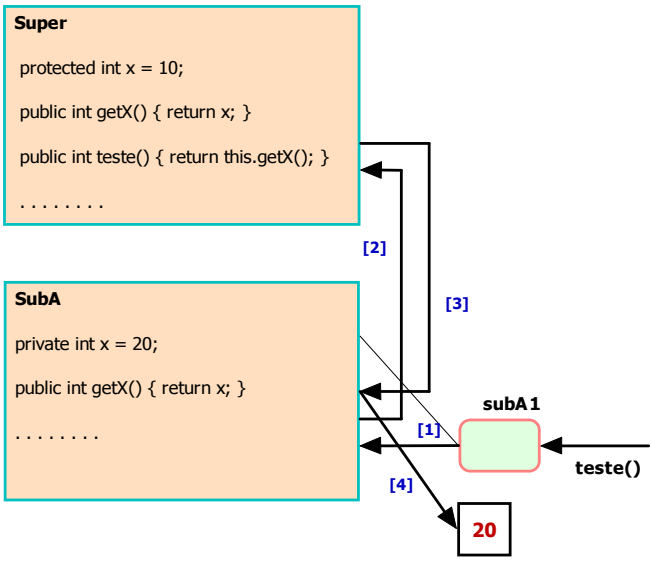

Figura 5.11 – Algoritmo de procura de métodos - exemplo

Quando a instância de `SubA` recebe a mensagem `teste()` o respectivo método é procurado na sua classe (cf. [1]) e, dado não ser encontrado, a procura continua na sua superclasse (cf. [2]). Em `Super`, o método é encontrado, e é executado `this.getX()`, que corresponde a enviar a mensagem `getX()` a `this`. Ora, este `this` não é uma instância da classe `Super`, mas a instância `subA1` de `SubA` que recebeu a mensagem `teste()`. Assim, o método `getX()` é o método da classe `SubA`, que é a **classe do receptor** [3], método que ao ser executado [4] devolve o valor `x` de `SubA` que é 20.

Assim, e de forma geral, a expressão **`this.m()`** onde quer que seja encontrada no código de um método de instância de uma dada classe da hierarquia, corresponderá sempre à execução do método **`m()`** da classe do receptor da mensagem que despoletou o algoritmo de procura.

Quando uma mensagem é enviada a um objecto, JAVA acrescenta aos parâmetros da mensagem um parâmetro adicional que é a referência do receptor (o equivalente a `this`), pelo que é directo determinar o receptor e a sua respectiva classe.

No exemplo das classes anteriores, obviamente que, se a mensagem fosse enviada a uma instância da classe `Super`, como `sup1.teste();` o resultado seria, tal como se pode verificar pela Figura 5.12, o valor 10, pois o método seria encontrado na própria classe do receptor e a execução de `getX()` daria, então, o resultado 10.

A mesma regra se aplica para expressões do tipo **`super.m()`**, em cujo caso a execução do método **`m()`** é remetida para a superclasse do receptor.

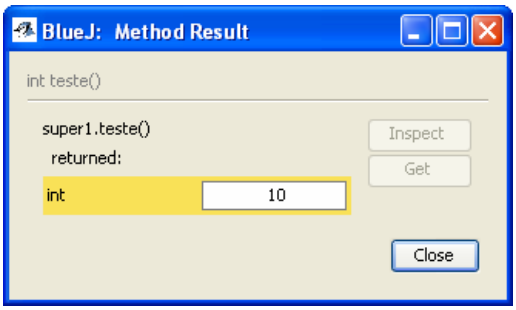

Figura 5.12 – Procura de métodos – exemplo

Considere-se, a título de exemplo, que criávamos agora uma classe `SubAA`, subclasse de `SubA` com o código que se apresenta a seguir, onde são feitas mais algumas redefinições de variáveis e métodos:

```
public class SubAA extends SubA {
   private int x = 30; // "shadow"
   String nome;        // "shadow"
   public int getX() { return x; }
   public String classe() { return "SubAA"; }
}
```

onde `getX()` e `classe()` são de novos redefinidos, e as variáveis `x` e `nome` continuam a ser usadas como, respectivamente, `private` e *package*. Criemos uma instância e enviemos-lhe a mensagem `teste()`, não existente na sua classe `SubAA` (Figura 5.13):

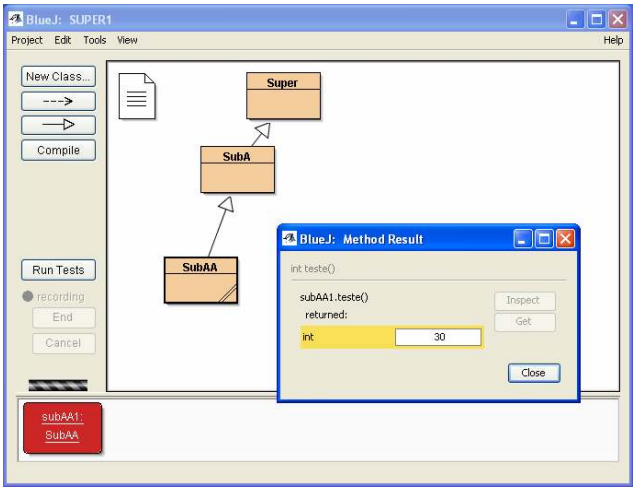

Figura 5.13 – Algoritmo de procura de métodos - Exemplo

Como se pode verificar pela Figura 5.13, o resultado de enviarmos à instância `subAA1` da nova classe `SubAA` a mensagem `teste()` é 30, ou seja, o valor da sua variável `x`, resultante da execução de `getX()` da classe `SubAA`, que não implementa `teste()`.

5.2.3 MODIFICADORES E REDEFINIÇÃO DE MÉTODOS

A possibilidade de redefinir um método está condicionada pelo tipo de modificadores de acesso do método da superclasse e do método redefinidor.

Assim, os métodos `public` apenas podem ser redefinidos por métodos `public`, os métodos `protected` por `public` e `protected`, e os métodos *package* (*default*) por um qualquer desde que não `private`. A regra geral é, como se pode verificar, que o método redefinidor nunca poderá diminuir ou restringir o nível de acessibilidade do método redefinido. Qualquer violação desta regra origina um erro de compilação.

Por outro lado, qualquer classe pode fazer garantir que os seus membros, sejam variáveis de instância ou de classe, sejam métodos de instância ou de classe, **não são redefinidos**, usando o modificador especial **`final`** na declaração destes.

Finalmente, os métodos `static` (de classe), ainda que não sejam herdados, podem ser redefinidos. Porém, um erro de compilação é gerado se um método de instância redefine (ou seja, tem a mesma assinatura de) um método de classe ou vice-versa.

Para todos estes métodos (e variáveis) que não são redefiníveis, o compilador de JAVA realiza optimizações de código, substituindo as suas invocações pelo seu código efectivo, operação que se designa por *inlining*, e que apenas se torna possível dado que para estes não há necessidade de activação do algoritmo de procura de métodos, pois, em tais casos, não existe qualquer possível indecisão quanto ao código a executar. A estes métodos não redefiníveis devem juntar-se os métodos não herdados, logo, por definição, não redefiníveis, ou seja, os métodos `private` e os métodos `static`.

5.2.4 CLASSES SEM SUBCLASSES

Em JAVA, é possível definir classes de tal forma que, garantidamente, estas nunca poderão ter subclasses. Para que tal propriedade particular seja associada a uma dada classe, basta que no seu cabeçalho esta seja declarada como **`final`**. Uma classe declarada como `final` corresponde a afirmar-se que tem uma definição completa, não permitindo posteriores especializações. Qualquer tentativa de se criar uma subclasse de uma classe `final` gera um erro de compilação. Assim, os métodos de uma classe final são automaticamente não redefiníveis.

Existem em JAVA classes que são `final`, como por exemplo a classe `String` (para desespero de alguns), o que é diferente de termos uma classe não `final` mas que tem métodos todos `final`. Esta poderá ter subclasses, mas tais métodos não podem ser reescritos.

5.2.5 SUBCLASSES E CONSTRUTORES UTILIZAÇÃO DE `this()` E `super()`

Podendo uma classe de JAVA possuir diversos construtores definidos pelo programador, para além do sempre existente construtor por omissão, será que é possível a um construtor de uma classe invocar um outro construtor da mesma classe? E fará tal invocação algum sentido?

A resposta a ambas as questões é sim. De facto, é possível em JAVA definir um dado construtor à custa de um outro construtor da mesma classe, usando no seu código a referência particular **`this(.,.)`** com os parâmetros adequados, em tipo e ordem, à invocação do outro construtor. Não confundir, no entanto, esta construção com a instrução `this.m()` em que `this` é uma referência especial para um objecto especial.

Retomando como exemplo a classe `Ponto` desenvolvida no capítulo 3, e cujos construtores foram definidos como:

```
public Ponto() {
   x = 0; y = 0;
}
public Ponto(int cx, int cy) {
   x = cx; y = cy;
}
```

faz todo o sentido, à luz do que aqui se pretende introduzir, que o primeiro construtor possa ser definido à custa do segundo, invocando-o usando a seguinte forma:

```
public Ponto() {
   this(0, 0); // invoca construtor Ponto(int, int)
}
```

A construção **`this(.,.)`** apenas pode ser utilizada em construtores, e possui uma restrição: **deverá ser obrigatoriamente a primeira instrução** do código do construtor.

Uma outra construção muito importante relacionada apenas com os construtores de JAVA é a construção **`super(.,.)`**. Quando criamos uma classe **B** que é subclasse de **A**, sabemos que **B** herda as variáveis de instância de **A** a que tem acesso**.** Assim, cada instância da classe **B** que seja criada possui uma estrutura interna que pode ser vista como sendo a soma das suas variáveis de instância (definidas na sua classe), com todas as outras variáveis de instância definidas nas suas superclasses.

Torna-se, assim, importante compreender de forma completa, não apenas como se podem manipular as variáveis locais, como também como podem ser usadas todas as variáveis a que, por herança, as instâncias têm acesso.

A Figura 5.14 reforça a ideia de que uma qualquer instância da classe **B** possui uma estrutura interna que é um somatório puro, caso não haja conflitos nos nomes das variáveis, das variáveis de instância herdadas e das definidas na sua classe. Assim sendo, e havendo a garantia de que a classe **B** possui construtores, pelo menos o seu construtor por omissão, temos a certeza de que, ao ser criada uma instância da classe **B**, as variáveis de instância definidas em **B** são inicializadas.

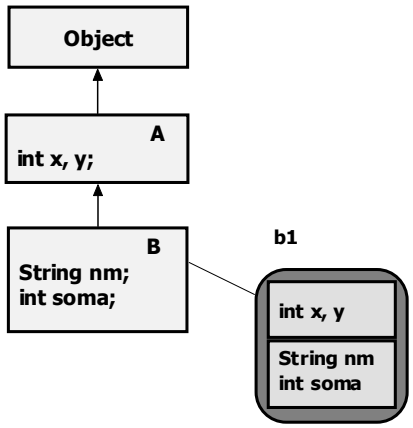

Figura 5.14 – Herança e estrutura das instâncias

Porém, coloca-se agora a questão de saber quem inicializa as variáveis que foram herdadas. A resposta é ao mesmo tempo simples e complexa. Tendo apenas em atenção o exemplo apresentado, torna-se óbvio que as variáveis herdadas de **A** apenas poderão ser inicializadas pelos construtores definidos em **A**, pelo que, assim sendo, qualquer construtor de instâncias da classe **B** deve obrigatoriamente invocar um qualquer dos construtores de instâncias de **A**, em última instância o construtor fornecido em JAVA por omissão de qualquer outro.

Sendo tal invocação obrigatória, sempre que esta regra não é respeitada no código dos construtores de uma dada classe, JAVA implicitamente insere como primeira instrução do construtor a instrução `super()`, que vai garantir a invocação do construtor por omissão da superclasse, dado que este existe sempre. Porém, caso o construtor da subclasse comece pela instrução `this()`, ou seja, invoque um outro construtor da mesma classe, JAVA vai verificar se este outro construtor invoca `super()` ou começa igualmente por `this()`. Dado que o próprio código de um construtor da subclasse pode invocar explicitamente um construtor da sua superclasse usando a construção `super(.,.)` com os devidos parâmetros, no final desta cadeia, quer implícita quer explicitamente, algum dos construtores da superclasse será sempre invocado.

Porém, caso `super(.,.)` seja de facto usado num construtor, então deverá ser a primeira instrução do código do construtor, mesmo quando `this(.,.)` for também usado.

A complexidade deste processo é que, sendo a herança transitiva, a criação de uma instância de uma dada classe obriga à invocação em cadeia dos construtores de todas as suas superclasses. Porém, como o algoritmo anterior é aplicado a todos os construtores de todas as classes, tal transitividade é assegurada. Isto é, quem tem que definir a classe **B** poderá ter algumas preocupações relativamente à invocação dos construtores mais adequados da superclasse **A**, mas tais preocupações não são extensíveis, já que quem definiu a classe **A** terá tido exactamente as mesmas, usando os construtores da superclasse de **A**, e assim sucessivamente.

A cadeia de construtores é assim implícita e, na pior das hipóteses, à falta de uma mais eficiente inicialização, usa os construtores que por omissão são definidos em JAVA. Esta é também uma das razões porque JAVA os disponibiliza por omissão.

Note-se, finalmente, que os construtores não são herdados pelas subclasses, ainda que, como acabámos de verificar, lhes sejam acessíveis desta forma.

Eis o momento ideal para que se compreenda, de forma pragmática e operacional, a questão de distinguir **herdar** de **ter acesso**. De facto, o que JAVA faz quando tem que criar uma instância de uma dada classe que tem várias superclasses é criar desde o topo da hierarquia instâncias de todas as superclasses desta, assim como uma instância desta contendo todas as variáveis e métodos, sejam ou não herdados. Isto é, a alocação de espaço é independente da possibilidade de tais conteúdos virem a ser usados ou não.

Tal é claro quando, por exemplo, inspeccionamos uma instância de `SubAA`, conforme se ilustra na Figura 5.15, onde se torna evidente que, ao nível da instância, o BlueJ apresenta quer as variáveis de instância locais, quer as herdadas das superclasses.

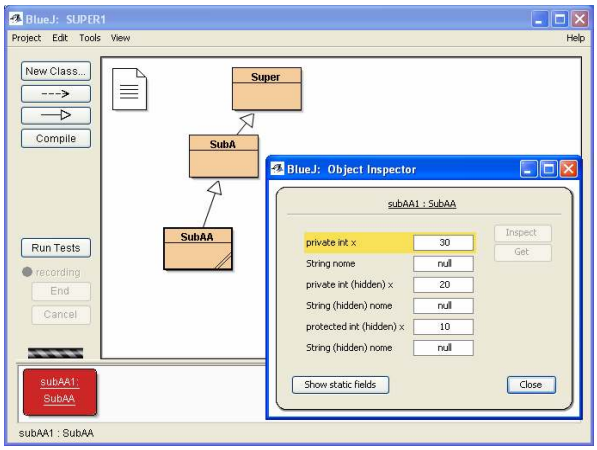

Figura 5.15 – Herança e estrutura das instâncias

Exemplos concretos de codificação de construtores usando estas construções são apresentados nos exemplos seguintes. Os construtores devem portanto, ser vistos a partir de agora como responsáveis por despoletarem os mecanismos necessários à alocação de espaço em memória para as instâncias, e por realizarem a atribuição de valores iniciais às respectivas variáveis, sejam valores por omissão sejam valores programados. Registe-se finalmente a importância crucial em todo este processo do construtor por omissão de JAVA a que nos referimos pela primeira vez no capítulo 3. Esta é a razão fundamental pela qual ele terá que existir, mesmo que não seja programado nas classes.

5.2.6 O TOPO DA HIERARQUIA: A CLASSE `Object`

Sabemos já que a classe que se posiciona no topo da hierarquia das classes de JAVA se designa por **`Object`**. Porquê este nome e o que deve conter esta classe?

A resposta a esta questão é muito fácil do ponto de vista filosófico ou conceptual, por um lado, mas muito complexa neste momento do ponto de vista de implementação, por outro. Ou seja, se pensarmos que a hierarquia de classes é uma hierarquia de especialização e que uma hierarquia de especialização é uma classificação "é-um" ou "é_do_tipo", então, no topo da hierarquia de JAVA tem que estar **algo** que todos sejam na hierarquia. Ora, provavelmente a única coisa que todos são em comum é **serem objectos**. Por isso, no topo da pirâmide está a classe **`Object`**, que representa o que todos têm em comum.

Complexo mesmo é encontrar atributos e comportamentos que devam ser definidos na classe `Object` para todas as subclasses herdarem. Sendo uma classe muito pouco específica, os métodos implementados são muito genéricos também. Sendo estes métodos "forçadamente" herdados pelas subclasses e não lhes sendo muito úteis (alguns), impõem às subclasses a sua redefinição. Caso estas não o façam, eles estão disponíveis (como se pode verificar, usando BlueJ, na Figura 5.16).

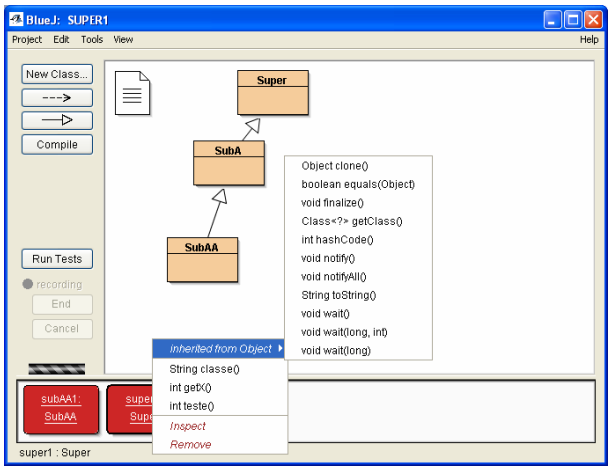

Figura 5.16 – Métodos herdados de `Object`

A classe `Object` define um pequeno conjunto de métodos que todas as subclasses vão herdar, dos quais se salientam os seguintes:

```
public Class<?> getClass()        // devolve a classe do objecto;
public boolean equals(Object obj) // igualdade de endereços;
protected Object clone()          // faz a clonagem do objecto;
public String toString()          // dá representação como string;
```

O método `getClass()`, que já usámos em conjunto com `getName()`, devolve a estrutura interna representativa de uma classe de JAVA (que tem o tipo `Classe<?>` que estudaremos mais tarde), estrutura que guarda toda a informação resultante da definição de uma classe e que é mantida em tempo de execução.

O método `toString()`, o equivalente ao `paraString()` que temos codificado até aqui, tendo que ter um código geral para ser herdado por todos os objectos, não faz mais do que passar para *string* o que todos os objectos possuem, quaisquer que sejam: o nome da sua classe e o seu identificador interno em hexadecimal (designado *object's hashcode*).

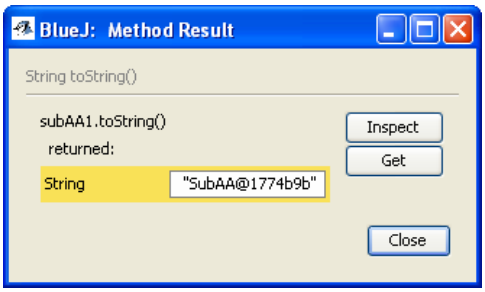

Figura 5.17 – Execução do método `toString()` de `Object`

Sendo o método `toString()` bastante simples quanto aos seus requisitos, pois apenas é especificado que deve devolver uma representação textual do objecto, passaremos a codificá-lo nas nossas classes tal como codificávamos o método `paraString()`, pelo que este ficará automaticamente redefinido.

Os métodos `equals(..)` e `clone()` da classe `Object` são métodos que, tal como os próprios especificadores da linguagem JAVA indicam, qualquer classe "bem comportada" deverá redefinir, mas que possuem requisitos de implementação desnecessariamente complexos em muitos casos.

O método `boolean equals(Object obj)` deverá implementar o teste de uma **relação de igualdade** entre o objecto passado como parâmetro e o objecto receptor da mensagem, cf. a expressão `obj1.equals(obj2);`. A classe `Object` implementa este método simplesmente verificando se `obj1 == obj2`, isto é, se possuem o mesmo endereço. Alguns autores designam este teste por *shallow comparison*. Dado que a classe `Object` não tem atributos, apenas pode assumir que endereços são atributos comparáveis entre objectos.

No entanto, as classes que queiram redefinir este método devem implementar uma comparação profunda, comparando os seus atributos com os do objecto parâmetro. A relação deve ser:

- Reflexiva: um objecto deve ser igual a si próprio, i.e. `o.equals(o) == true`;
- Simétrica: se `x.equals(y)` então `y.equals(x)`;
- Transitiva: se `x.equals(y) && y.equals(z)` => `x.equals(z)`.

A simetria é a propriedade de interpretação mais complexa. Caso seja interpretada de forma mais lata, então os objectos `x` e `y` podem ser de classes diferentes, sejam estas X e Y, e a satisfação da propriedade implicaria que os métodos `equals()` de ambas as classes fossem desenvolvidos em sintonia, por forma a que os algoritmos de comparação garantissem que, se um `y` comparado com um `x` em `X` dá `true`, então esse `x` comparado com o mesmo `y` em Y dará `true` também. O cumprimento da simetria para este caso quase implicaria refazer os usuais ciclos de desenvolvimento de *software*. Assim, vamos seguir a boa regra que diz que "alhos não se comparam com bugalhos", e vamos apenas considerar simetria dentro da mesma classe, ou seja, para instâncias da mesma classe.

Adicionalmente, a comparação deve ser consistente, ou seja, se `x.equals(y)` deu um dado resultado, não havendo modificações em x e y relevantes para a comparação, o resultado obtido deve manter-se em invocações posteriores. Finalmente, se um qualquer dos objectos for `null`, a comparação deve dar `false`. Omite-se a relação entre a igualdade de objectos e os valores dos seus *hashcode* (dados pelo método `hashCode()`).

Vejamos então a estrutura do método `equals()` tal como o implementaríamos na classe `Ponto2D`:

```
public boolean equals(Object obj) {
   if ( this == obj ) return true;   // é o próprio !!
   if ( obj == null ) return false;
   if ( this.getClass() != obj.getClass() ) return false;
   // Aqui temos a certeza que é um Ponto2D
   Ponto2D pp = (Ponto2D) obj;     // casting obrigatório
   return x == pp.getX() && y == pp.getY();
}
```

O método começa por testar se o receptor se está a comparar consigo próprio, ou seja, se o objecto parâmetro é ele próprio. Em seguida, é testada a possibilidade de o objecto parâmetro ser `null` sendo devolvido `false`. A linha seguinte testa, usando `getClass()`, se os objectos são instâncias da mesma classe, no exemplo `Ponto2D`. Superados estes testes, então o objecto parâmetro é convertido num `Ponto2D`, e é finalmente realizada a comparação de dados, no caso, as coordenadas do receptor com as do ponto parâmetro.

O teste `this.getClass() != obj.getClass()` é por vezes substituído pela forma:

```
!( this.getClass().getName().equals(obj.getClass().getName()) )
```

que compara as *strings* representativas dos nomes das classes dos objectos. Ambas são correctas, mas a primeira forma é, em geral, menos dependente do que a segunda.

Em alguns textos aparece como alternativa o teste

```
if ( !(obj instanceof Ponto2D) )
```

que supostamente deveria testar se o parâmetro é ou não da classe `Ponto2D`. No entanto este teste envolve alguns riscos pois falha (i.e. `instanceof` dá `true`) quando, por qualquer razão, se passa como parâmetro uma instância de uma subclasse de `Ponto2D`. Por exemplo, se como argumento do método `equals()` de `Ponto2D` fosse dada uma instância de `Ponto3D`, então, o teste usando `instanceof` daria `true` e o ponto seria mesmo considerado comparável. Uma instância de uma subclasse de `Ponto2D` é considerada `instanceof Ponto2D` também. Neste caso, o ponto iria ser mesmo comparado, e, se tivéssemos a comparação `Ponto2D(5, -2).equals(new Ponto3D(5, -2, 4))` esta daria mesmo `true`. Então, pela regra da simetria, a comparação `Ponto3D(5, -2, 4).equals(new Ponto2D(5, 2))` deveria dar `true` também, o que já não seria tão aceitável, tanto mais que, caso não fossemos nós próprios (ou quem implementou `equals()` de `Ponto2D`) a implementar também o método `equals()` de `Ponto3D`, dificilmente o implementaríamos com tal propriedade em mente. Em conclusão, o melhor será não utilizar `instanceof` e usar `getClass()`, evitando assim situações deste tipo, que, note-se, apenas resultariam de não se respeitar o facto de que o método espera uma instância de `Ponto2D`.

Os dois testes que darão `false` caso o objecto parâmetro não seja comparável, podem ser reunidos numa única expressão lógica, tal como em:

```
if ((obj == null) || (this.getClass() != obj.getClass()))
     return false;
```

Assim, podemos escrever agora o código típico genérico para o método `equals()` de **uma qualquer classe X**, e que a partir de agora será por nós utilizado nas nossas classes:

```
public boolean equals(Object obj) {
   if (this == obj) return true;
   if ((obj == null) || (this.getClass() != obj.getClass()))
     return false;
   // É de certeza um X
```

```
    X x = (X) obj;      // casting para tipo X
    return algoritmo de comparação
}
```

O algoritmo de comparação a implementar dará como resultado verdeiro ou falso, e, em geral, consiste na comparação dos objectos campo a campo, usando o operador de igualdade de valores == para campos de tipos primitivos, e o método `equals()` para os campos de tipos referenciados (objectos).

O método `clone()` de `Object` é um método ineficaz e, adicionalmente, apresenta alguma complexidade de utilização. Dado que a compreensão completa das questões relacionadas com o funcionamento deste método se prendem com a hierarquia de classes e com a herança, vamos, em primeiro lugar, estudar a criação de subclasses usando herança de variáveis e métodos, e como é realizada a redefinição destes, abordando depois, na secção correspondente à **clonagem de objectos**, as questões relacionadas com a utilização, ou não, do método `clone()` definido em `Object`.

5.3 CRIAÇÃO DE CLASSES VIA HERANÇA: EXEMPLOS

Os exemplos seguintes de utilização da herança são relativamente simples pois ainda não foram estudadas classes que nos permitam trabalhar exemplos com estruturas de dados. As classes apresentadas têm o seu código reduzido ao mínimo essencial para que possam sobressair os aspectos relacionados com a utilização dos mecanismos de herança, quer quanto à utilização dos construtores, quer quanto à herança e redefinição de métodos.

5.3.1 EXEMPLO1: SUBCLASSES DE Ponto2D

Consideremos o código correspondente à classe `Ponto2D` apresentado a seguir, escrito de forma compacta, sem comentários, código que irá servir de referência para a criação de duas subclasses da classe `Ponto2D`.

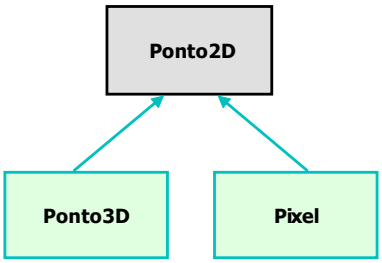

Figura 5.18 – Hierarquia a construir

```
public class Ponto2D {
   // Construtores
   public Ponto2D(int cx, int cy) { x = cx; y = cy; }
   public Ponto2D(){ this(0, 0); }
   // Variáveis de Instância
   private int x, y;
   // Métodos de Instância
   public int getX() { return x; }
   public int getY() { return y; }
   public void desloca(int dx, int dy) {
      x += dx; y += dy;
   }
   public void somaPonto(Ponto2D p) {
      x += p.getX(); y += p.getY();
   }
   public Ponto2D somaPonto(int dx, int dy) {
      return new Ponto2D(x += dx, y+= dy);
   }
   public String toString() {
      return new String("Pt= " +x + ", " + y);
   } }
```

A classe `Ponto2D` define dois construtores. O método `desloca()` permite incrementar as coordenadas de um ponto, o método `somaPonto(Ponto2D p)` soma ao ponto receptor o valor das coordenadas do ponto parâmetro, e `somaPonto(int dx, int dy)` devolve o ponto resultante de somar às coordenadas do receptor os valores dados como parâmetro. Os restantes métodos são óbvios.

PIXEL

Um *pixel* é um ponto do plano a duas dimensões inteiras que tem a si associada uma **cor**, sendo cada cor representada por um inteiro. Pretende-se criar uma classe que nos permita criar instâncias de *pixel* que possamos manipular como um normal ponto do plano, mas ao qual possamos atribuir e modificar a cor, e apresentar textualmente com as suas coordenadas e respectivo número de cor.

Assim, como seria de esperar, a classe `Pixel` é subclasse de `Ponto2D`, herda desta classe `Ponto2D` todos os atributos e métodos, e define um atributo **`cor`** (de 0 a 99), métodos de consulta e alteração de cor e redefine o método `toString()`.

```
public class Pixel extends Ponto2D {
  // Variáveis de Instância
  private int cor;
  // Construtores
  public Pixel() { super(0, 0); cor = 0; }
  public Pixel(int cor) { this.cor = cor%100; }
  public Pixel(int x; int y; int cor) {
    super(x, y); this.cor = cor%100;
  }
  // Métodos de Instância
  public int getCor(){ return cor; }
  public void mudaCor(int nvCor) { cor = nvCor%100; }
  public void somaPixel(int x, int y) { super.desloca(x,y); }
  public void somaPixel(Pixel pix) {
     super.desloca(pix.getX(),pix.getY());
     cor = (cor + pix.getCor())%100;
  }
  public String toString() {
    StringBuilder s = new StringBuilder();
    s.append(super.toString());
    s.append(", ("); s.append(cor); s.append(" )");
    return s.toString();
  }
}
```

O primeiro construtor, `Pixel()`, começa por invocar o construtor da superclasse para criar um ponto 2D com ambas as variáveis inicializadas a 0. Em seguida, inicializa `cor` a 0. O segundo construtor não usa `super(0, 0)` e apenas inicializa a variável de instância `cor`.

Vejamos em BlueJ o resultado de criarmos um *pixel* usando este segundo construtor e de imediato invocarmos sobre essa instância criada o método `toString()` (Figura 5.19).

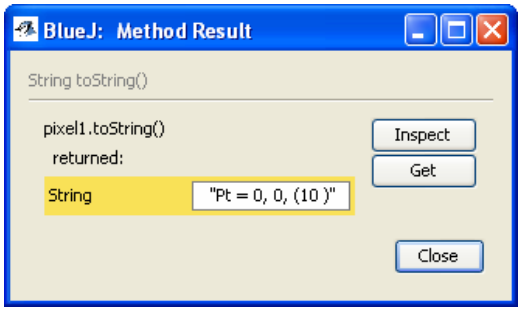

Figura 5.19 – Instância de `Pixel` via `toString()`

Como podemos verificar, as variáveis `x` e `y` são igualmente inicializadas a zero e a `cor` é 10 pois foi o valor de cor introduzido como parâmetro. Embora o segundo construtor não tenha invocado `super(0, 0)`, JAVA usou o construtor por omissão `super()`, que neste caso é exactamente equivalente a `super(0, 0)` por ser 0 o valor por omissão usado para inicialização de inteiros.

O terceiro construtor permite criar um *pixel* com as coordenadas dadas como parâmetro e com a cor indicada. Poderíamos ter criado também um construtor que, recebendo um *pixel* por parâmetro, realizasse a criação da instância por cópia do *pixel* parâmetro.

Vejamos de novo o resultado, agora usando *Inspect*, que nos permite observar todas as variáveis do objecto, quer locais quer da superclasse, tal como se apresenta na figura a seguir (Figura 5.20).

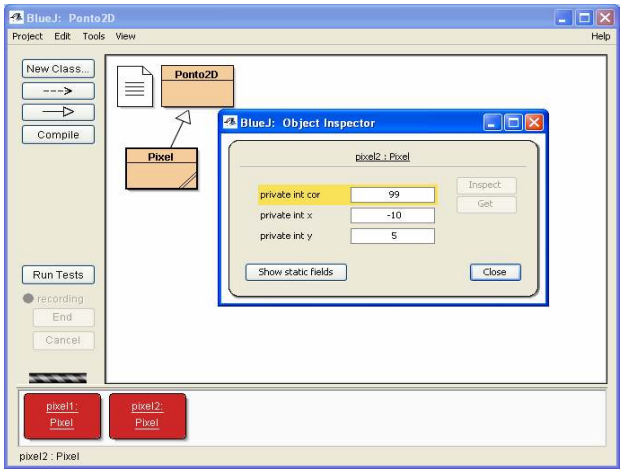

Figura 5.20 – Resultado do uso de `Pixel(int, int, int)`

O método `toString()` usa uma instância de `StringBuilder` (*java.lang*) na qual se faz sucessivamente o `append()` de *strings* e inteiros. Porém, como o método deve devolver uma `String` e temos uma instância de `StringBuilder`, na instrução `return` é feito o *casting* de um tipo para o outro usando o método `toString()` de `StringBuilder`, que devolve uma `String`, tal como pretendemos.

O método `somaPixel()` recebe um *pixel* como parâmetro, e vai ter que somar as coordenadas deste *pixel* às do receptor. Para tal, extrai as coordenadas do parâmetro usando os métodos `getX()` e `getY()`, e depois invoca o método `desloca(int, int)` que soma estes valores às suas coordenadas `x` e `y`, a queas instâncias de `Pixel` não têm acesso directo, tendo pois que usar métodos da classe `Ponto2D`. O método `desloca()` permite realizar as alterações necessárias em tais coordenadas (numa delas ou em ambas).

Repare-se que se enviarmos a uma instância de `Pixel` a mensagem `getX()`, será devolvido como resultado a sua coordenada em `x`, mas quem implementa a resposta a tal mensagem é a classe `Ponto2D`.

PONTO3D

A classe `Ponto3D` vai ser definida como sendo subclasse de `Ponto2D`, declarando apenas a variável de instância correspondente à coordenada em **z** e adequando o código dos métodos à sua representação herdada e local:

```
public class Ponto3D extends Ponto2D {
  // Construtores
  public Ponto3D(int x, int y, int z) {
    super(x, y) ; this.z = z ; }
  public Ponto3D() { super(); z = 0; } // ou this(0, 0, 0)
  // Variáveis de Instância
  private int z ;
  // Métodos de Instância
  public int getZ() { return z; }
  public void desloca(int dx, int dy, int dz) {
```

```
        super.desloca(dx, dy); z += dz;
    }
    public void somaponto(int x, int y, int z) {
        super.somaponto(x, y); this.z += z;
    }
    public void somaPonto(Ponto3D p) {
        super.somaPonto(p); z += p.getZ();
    }
    public Ponto3D somaPonto(int dx, int dy, int dz) {
        return new Ponto3D(this.getX() += dx,
                           this.getY() += dy, z += dz);
    }
    public String toString() {
      StringBuilder s = new StringBuilder();
      s.append(super.toString());
      s.append(", "); s.append(z);
      return s.toString();
    }
}
```

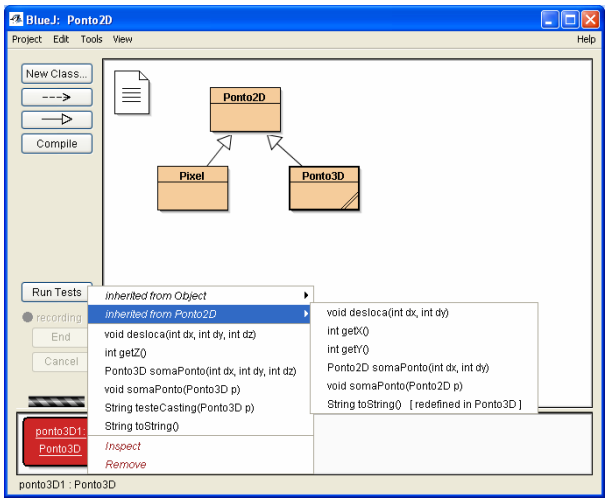

Figura 5.21– API total de `Ponto3D`

O método `void somaPonto(Ponto3D p)` começa por chamar o método da superclasse que soma pontos 2D, usando a expressão **`super.somaPonto(p)`** onde **`p`** é argumento do tipo `Ponto3D`. Ora, o método `somaPonto()` da superclasse, espera um `Ponto2D` como parâmetro, mas é-lhe passado um `Ponto3D`, que sendo uma subclasse de `Ponto2D`, é, portanto, um subtipo compatível, pelo que tal está correcto. Pela primeira vez usámos a possibilidade de atribuir a uma variável de uma superclasse uma instância de uma sua subclasse e testar a compatibilidade.

As instruções sublinhadas a negrito reflectem a quantidade de métodos da superclasse que foram reutilizados pela subclasse e, também, a forma simples como tal reutilização pode ser feita.

Note-se como padrão comum à maioria dos métodos, a reutilização de um método da superclasse para trabalhar com as coordenadas x e y, e em seguida a operação necessária sobre a variável de instância `z`, que é local.

Neste exemplo, ao contrário até de alguns anteriores, apenas o método `toString()` foi redefinido. Todos os outros métodos são herdados e estão directamente acessíveis.

5.3.2 EXEMPLO2: COMPLEXOS (RECT ▶ POLAR)

Consideremos uma classe que implementa números complexos sob a forma rectangular (ex.: **z = a + bi**), usando duas variáveis de instância `double` e implementando alguns métodos comuns, tais como a soma, o produto e o cálculo do conjugado.

```
public class ComplexoR {
   // Variáveis de Instância
   private double a;  //  z = a + bi
   private double b;
   // Construtores
   public ComplexoR() { a = 0.0; b = 0.0; }
   public ComplexoR(double r, double i) { a = r; b = i; }
   // Métodos de Instância
   public double getR() { return a; } // parte real
   public double getI() { return b; } // p. imaginária
   // z + w = (a + bi) + (c + di) = (a+c) + (b+d)i
   public ComplexoR soma(ComplexoR cr) {
    return new ComplexoR(a + cr.getR(), b + cr.getI());
   }
   // z*w = (a + bi) * (c + di) = (ac – bd) + (bc + ad) i
   public ComplexoR produto(ComplexoR cr) {
     double a1 = a*cr.getR() – b*cr.getI();
     double b1 = b*cr.getR() + a*getI();
     return new ComplexoR(a1, b1);
   }
   // z* = a – bi (conjugado de z)
   public ComplexoR cojugado() {
      return new ComplexoR(a, -b);
   }

   // toString()
   public String toString() {
     StringBuilder s = new StringBuilder();
     s.append("Cplx = "); s.append(a);
     s.append(b>=0 ? "+" : ""); s.append(b +"i");
     return s.toString();
   }
}
```

Mostra-se em seguida a implementação da subclasse `ComplexoP`, que, baseando-se na sua superclasse, em especial na estrutura de variáveis, constrói uma representação polar para números complexos, sendo **z = r (cos θ + sen θ i)**, sem que para tal necessite de declarar quaisquer variáveis de instância, pois apenas usa as regras de conversão:

$a = r.\cos\Theta;\ b = r.\text{sen}\,\Theta\ ;\ r = \text{sqrt}(a^2 + b^2)\ ;\ \Theta = \arctan\ b/a$

```
import static java.lang.Math.*;
public class ComplexoP extends ComplexoR {
   // Construtores
   public ComplexoP() { super();}
   public ComplexoP(double raio, double teta) {
      super(raio*cos(teta), raio*sin(teta));
   }
   // Métodos de Instância
   // Conversão r = sqrt(a*a + b*b)
   public double getRaio() {
     double temp = pow(super.getR(),2) + pow(super.getI(),2);
     return sqrt(temp);
   }
   // Conversão θ = arctan b/a
   public double getTeta() {
     return atan(super.getI()/super.getR());
   }
   // Conversão de ComplexoP em ComplexoR
   public ComplexoR converte(ComplexoP cp) {
     return new ComplexoR(cp.getRaio()*cos(cp.getTeta()),
                          cp.getRaio()*sin(cp.getTeta()));
   }
```

```
    // Soma dois ComplexoP
    public ComplexoP soma(ComplexoP cp) {
      double ac = this.getRaio()*cos(this.getTeta()) +
                  cp.getRaio()*cos(cp.getTeta());
      double bd = this.getRaio()*sin(this.getTeta()) +
                  cp.getRaio()*sin(cp.getTeta());
      return new ComplexoP(sqrt(pow(ac,2) + pow(bd,2)),
                           atan(bd/ac));
    }

    // z1*z2 = r1*r2 (cos (θ1+θ2) + sen(θ1+θ2)i)
    public ComplexoP multiplica(ComplexoP cp) {
      double nvRaio = this.getRaio() * cp.getRaio();
      double nvTeta = this.getTeta() + cp.getTeta();
      return new ComplexoP(nvRaio, nvTeta);
    }
}
```

Esta classe mostra-nos um exemplo muito especial de herança em que a superclasse usa completamente os métodos da superclasse para criar uma nova representação de números complexos, neste caso em coordenadas polares. O facto interessante do exemplo, é que tal representação não existe, a classe não possui variáveis de instância. Guarda o seu estado nas variáveis da superclasse e dá a ilusão de que possui um `raio` e um `teta`. No entanto a classe apenas faz conversões de um formato para o outro via superclasse.

5.4 COMPATIBILIDADE ENTRE CLASSES E SUBCLASSES

As vantagens de possuirmos uma hierarquia baseada num mecanismo de herança são agora bastante mais claras. Podemos criar novas classes com menor esforço, evitando duplicação de código através da reutilização de código já existente.

Mas, sendo classes também **tipos**, aos quais variáveis são associadas estaticamente através de declarações, como `Ponto2D p1;`, haverá agora que analisar até que ponto é que classe e suas subclasses são compatíveis, isto é, o **grau de compatibilidade entre tipos e subtipos** definidos pela hierarquia de classes. Pretendemos assim responder à pergunta: **É uma classe compatível com as suas subclasses?**

5.4.1 O PRINCÍPIO DA SUBSTITUIÇÃO

Uma das mais interessantes e importantes características das linguagens de PPO é que o "**valor"** (instância) referenciado por uma variável pode não ser do **tipo** (classe) declarado para essa variável. Mas, evidentemente, também não poderá ser um tipo qualquer, sendo claro o que o **princípio da substituição** determina: **"Se uma variável é declarada como sendo de uma dada classe (tipo), é legal que lhe seja atribuído um valor (instância) dessa classe ou de uma qualquer das suas subclasses"**. Este princípio estabelece uma das regras mais importantes de toda a programação por objectos, como veremos.

Ou seja, a compatibilidade, tal como já anteriormente havíamos referido intuitivamente, é no sentido ascendente da hierarquia (anteriormente Triângulo e Polígono). Uma instância de uma subclasse poderá ser atribuída a uma variável do tipo da superclasse.

Trata-se de um princípio perfeitamente lógico. Se Triângulo, Quadrado e Círculo são tipos especiais de algo mais genérico (superclasse) designado por Forma, então, se nos pedirem uma Forma podemos apresentar um Quadrado ou um Triângulo, etc., ou seja, uma entidade que possa correctamente representar uma Forma.

São várias as implicações a ter em conta em JAVA pelo facto de, sendo **B subclasse de A** podermos escrever, segundo o princípio da substituição, expressões como:

```
A a, a1;
a = new A(); a1 = new B();
```

Em primeiro lugar, a distinção do ponto de vista de tipos entre **declaração de variável** e **atribuição de valor**. Sabemos que, do ponto de vista do compilador, ou seja, do ponto de vista da **verificação estática** de tipos de dados,

ambas são importantes, e ambas têm que ser verificadas no momento da compilação, tendo em conta o princípio da substituição.

Assim, as expressões acima estão ambas correctas, pois `B` é uma subclasse de `A`. Veremos depois como é que o **interpretador** vai realizar a efectiva determinação da classe do "conteúdo concreto" de `a` e `a1`, quando, por exemplo, tiver que executar uma expressão do tipo `a.m()`, já que tal conteúdo pode ser de vários tipos: uma instância de `A` ou uma instância de uma **qualquer subclasse de `A`** – o princípio assim o determina.

O **compilador** tem que verificar se o método `m()` existe em `A` ou numa superclasse de `A`. Se existir numa destas classes é como se existisse na subclasse `B`, desde que seja **acessível** a `B` claro. A expressão será então considerada sintacticamente correcta. Porém, será o interpretador que, em tempo de execução, deverá determinar qual o método `m()` a executar de facto, tal dependendo do "valor" (*instância*) nesse momento (execução) efectivamente contido (referenciado por) em `a` (daí o nome *dynamic binding*). Este, pode ser uma instância de `A`, de `B` ou de qualquer outra subclasse de `A`. Se todas implementarem o método `m()`, o interpretador deve executar o método da correspondente classe.

Vamos ver um exemplo concreto. Considere-se a seguinte hierarquia de classes:

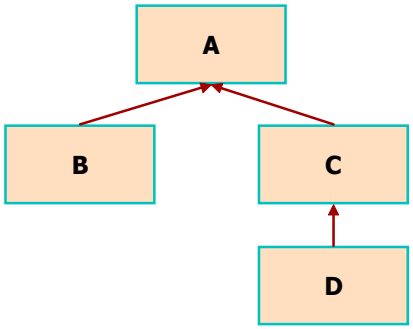

Figura 5.22 – Hierarquia exemplo: compatibilidade classe-subclasses

Cada uma das classes **`A`**, **`B`**, **`C`** e **`D`** é definida como se apresenta a seguir, contendo um construtor e dois métodos, `daVal()` e `metd()`, que cada uma delas redefine a seu modo, mas que realizam em todas as classes a mesma função. O método `daVal()` devolve o valor da variável de instância definida na classe, e o método `metd()` altera o valor dessa variável depois de, sobre ela, realizar uma determinada operação. O método é redefinido em cada uma das subclasses de **`A`**, tal como se apresenta a seguir:

```
public class A {
   public A() { a = 1; }
   public int daVal() { return a; }
   public void metd() { a += 10; }
}
public class B extends A {
   public B() { b = 2; }
   private int b;
   public int daVal() { return b; }
   public void metd() { b += 20 ; }
}
public class C extends A {
   public C() { c = 3; }
   private int c;
   public int daVal() { return c; }
   public void metd() { c += 30 ; }
}
public class D extends C {
   public D() { d = 33; }
   private int d;
   public int daVal() { return d; }
   public void metd() { d = d * 10 + 3 ; } }
```

Consideremos o seguinte programa de teste e tentemos analisar quais os resultados esperados da execução do código respectivo:

```
import static java.lang.System.out;
public class TesteABCD {
   public static void main(String args[]) {
     A a1, a2, a3, a4;
     a1 = new A(); a2 = new B(); a3 = new C(); a4 = new D();
     a1.metd(); a2.metd(); a3.metd(); a4.metd();
     out.println("a1.metd() = " + a1.daVal());
     out.println("a2.metd() = " + a2.daVal());
     out.println("a3.metd() = " + a3.daVal());
     out.println("a4.metd() = " + a4.daVal());
   }
}
```

Em primeiro lugar, repare-se que declarámos quatro variáveis de tipo **A** e que a estas pudemos associar instâncias quer de **A**, quer de **B**, quer de **C**, quer de **D**. O compilador aceita tais atribuições, dado estarem em conformidade com o **princípio da substituição** e a relação de inclusão transitiva que está implícita no mesmo. A Figura 5.23 procura representar de forma visual a relação de inclusão existente entre as quatro classes.

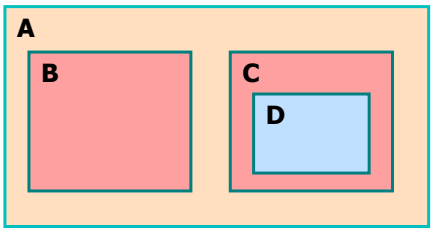

Figura 5.23 – Relação de inclusão de tipos

Sendo a declaração de uma variável um procedimento estático e que tem a ver com a fase de compilação, enquanto que uma atribuição é um procedimento dinâmico, pois é realizado durante a execução do programa, torna-se importante, então, que se passe a fazer a distinção entre o **tipo estático** e o **tipo dinâmico** de uma variável, já que, como acabámos de ver pelo princípio enunciado, eles podem não coincidir.

Assim, o **tipo estático** de uma variável será sempre o tipo da sua declaração tal como o compilador o aceitou. No exemplo em análise, em que se escreveu a declaração `A a1, a2, a3, a4;` o **tipo estático** das variáveis é a classe (tipo) **A**. O princípio da substituição garante, no entanto, a legalidade de se associarem a estas variáveis valores (**instâncias**), que podem ser, quer instâncias de **A** quer de uma qualquer subclasse de **A**. Então, estão sintacticamente correctas as expressões **`a1 = new A(); a2 = new B();`** etc.

O facto de uma variável poder possuir um valor de tipo dinâmico diferente do seu tipo estático, ainda que dentro da regra de subclassificação enunciada, é uma fonte de poder expressivo, de generalidade e de flexibilidade que apenas as linguagens de PPO suportam de forma tão natural. Pelo facto de uma variável de dado tipo estático poder possuir diferentes tipos dinâmicos (ou **formas**) em tempo de execução, ela diz-se **polimórfica**.

Informalmente, e de um ponto de vista meramente sintáctico, o princípio está de acordo com o facto de que as subclasses são em PPO extensões, especializações, da superclasse, pelo que, herdando desta um conjunto de métodos e podendo a estes acrescentar os seus próprios, superclasses e subclasses são comportamentalmente compatíveis no comportamento comum (o herdado, isto é, o que foi definido na superclasse). Desde que o método correspondente à mensagem enviada a uma variável de dado tipo esteja definido na classe desta, o compilador tem a garantia de que, mesmo que mais tarde tal variável possua um valor não da sua classe mas de uma das suas subclasses, seja por herança ou porque a subclasse o redefiniu, tal método será sempre executável, **ainda que seja necessário determinar, em tempo de execução, qual o código do método que deve ser efectivamente executado**.

Este mecanismo, usado pelo interpretador (logo, em tempo de execução), que selecciona automaticamente o método de instância adequado, em função do tipo dinâmico do receptor de uma mensagem, designa-se por ***dynamic binding*** (associação dinâmica).

5.4.2 PROCURA DINÂMICA DE MÉTODOS
DYNAMIC METHOD LOOKUP

Ainda que ao compilador seja impossível saber qual o tipo dinâmico de uma certa variável, o código produzido por este vai permitir que o interpretador de JAVA use um algoritmo de **procura dinâmica de métodos**, em função da análise que faz, em tempo de execução, de uma expressão tal como `a2.metd()` escrita na classe `TesteABC`.

De facto, em vez de usar de imediato o método `metd()` associado ao tipo estático da variável `a2` (classe **`A`**), o interpretador vai em primeiro lugar determinar o tipo dinâmico de `a2` em tal expressão, no exemplo a classe `B`, e em tal classe procurar o código do método a executar. Caso exista, como no exemplo dado, o seu código é executado, neste caso colocando a variável privada `b` com o valor `22`. Caso o método não existisse em tal classe, a procura continuaria, seguindo o algoritmo de procura, nas superclasses desta.

Assim, no exemplo em análise, o resultado da execução das instruções

```
a1.metd(); a2.metd(); a3.metd(); a4.metd();
```

seria a activação dos métodos `metd()` das classes `A`, `B`, `C` e `D`, que, sendo métodos modificadores, colocariam as variáveis privadas das instâncias `a1`, `a2`, `a3` e `a4` com os valores, respectivamente, `a = 11`, `b = 22`, `c = 33` e `d = 333`.

Se o algoritmo de procura e execução de métodos fosse baseado apenas nos tipos estáticos das variáveis, então, sendo de tipo `A` todas as variáveis receptoras da mensagem, tal resultaria em quatro invocações do método `metd()` da classe **`A`**, cada uma incrementando a variável `a` da respectiva instância em dez unidades. Não é, no entanto, este o caso.

Ora, a invocação do método `metd()` apenas modificou o estado interno das instâncias. Agora, para cada uma delas pretendemos consultar o valor da sua variável de instância. O método programado para tal efeito em cada classe, desde `A` até `D`, é o método `daVal()`. É necessário, mais uma vez, que o interpretador seleccione o `daVal()` de `A` para `a1`, o `daVal()` de `B` para `a2`, etc. para que os resultados finais sejam os correctos. Escolher o método correcto para aplicar a cada uma, quer para o caso de `metd()` quer para o caso de `daVal()`, só poderá ser realizado pelo interpretador através da **procura dinâmica de métodos**, visando a associação correcta do método ao objecto, o **dynamic binding**.

A Figura 5.24, apresentada a seguir, é reveladora. Temos a hierarquia, temos a classe de teste e temos a janela com os resultados da execução do programa. Como podemos ver, sem nos preocuparmos com a forma como tal foi conseguido, o interpretador de JAVA determinou correctamente qual o método `daVal()`a aplicar a cada uma das instâncias, daí que os resultados apresentados sejam os que havíamos determinado como correctos. Há que notar como tudo é complexo e parece simples, e como nos é dada a possibilidade de, usando um **supertipo**, programarmos um só comportamento que é adequado a todos os seus **subtipos**.

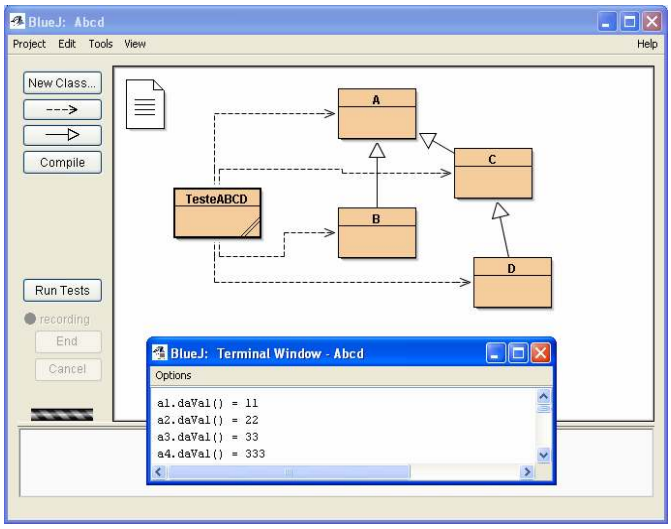

Figura 5.24 – Procura dinâmica de métodos e *dynamic binding*

Temos **a mesma mensagem** a funcionar de forma diferente em quatro classes. Não somos nós, com estruturas `switch` por exemplo, que estamos a distinguir qual deve ser o método a aplicar a cada instância, nem nunca precisaremos de fazer tal. Apenas usámos os intrínsecos **polimorfismo** (cf. o princípio da substituição) e **dynamic binding**.

Dada a importância da compreensão destes mecanismos, vejamos um outro exemplo no qual uma classe de nome **`Msg`** cria instâncias que simplesmente produzem uma mensagem no monitor. A mensagem tem uma parte fixa inicial, que será sempre igual, à qual se segue um texto contido no método **`String msg()`** que a classe implementa. Cada uma das suas subclasses deverá redefinir este método por forma a compor a mensagem final de texto que pretender:

```
public class Msg {
   public static String inicio() { return "Ola, "; }
   public String msg() { return " eu sou <a preencher>."; }
}
```

Sendo a parte inicial do texto fixa e sempre igual, trata-se de um valor comum a todas as instâncias e, portanto, deveria ser codificado como sendo uma variável de classe. Neste caso decidiu-se codificá-lo como sendo um método de classe que devolve uma *string*. Foram definidas duas subclasses da classe `Msg`, cada uma redefinindo a seu modo o método `msg()`, tal como se pode ver no código seguinte:

```
public class MsgPedro extends Msg {
   public String msg() { return " eu sou o Pedro."; }
}

public class MsgRita extends Msg {
   public String msg() { return " eu sou a Rita."; }
}
```

O programa principal cria três instâncias, uma da superclasse e uma de cada subclasse. A cada uma envia a mensagem `inicio()`. A *string* resultante é concatenada com o resultado do envio da mensagem `msg()`, sendo produzida a *string* final. Vejamos:

```
public class TesteMsg {
   public static void main(String args[]) {
     //
      Msg mens, mens1, mens2;
      mens = new Msg();
      mens1 = new MsgRita(); mens2 = new MsgPedro();
      out.println(mens.inicio() + mens.msg());
      out.println(mens1.inicio() + mens1.msg());
      out.println(mens2.inicio() + mens2.msg());
```

```
    }
}
```

Este programa principal não gera erros de compilação. Mas será que não deveria gerar? Será correcto enviar a mensagem `inicio()` à instância `mens1` da classe `MsgRita`? O método `inicio()` não é sequer um método de instância. Não se deveria ter antes escrito `Msg.inicio()`?

De facto, foi a designada sorte de principiante. A forma correcta, já que sabíamos ser um método de classe, e sabíamos também que não havia sido redefinido em nenhuma subclasse, seria escrever `Msg.inicio()`. Como dissemos atrás, os métodos de classe são resolvidos em tempo de compilação. Não há sobreposição, e por serem de classe não há que os procurar em função do tipo dinâmico (daí também a designação *static*).

A sorte, referida há pouco, é que o compilador ao procurar o tipo estático das variáveis `mens1` e `mens2` encontrou o tipo **`Msg`** e aí encontrou o método, gerando de imediato o seu código "inline", o que faz para métodos `static`, `final` e `private`.

Assim, o resultado final do programa é o apresentado na janela do BlueJ da Figura 5.25:

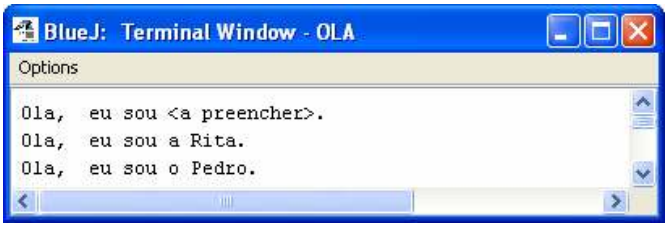

Figura 5.25 – Procura dinâmica de métodos – exemplo 2

Como se pode verificar, o texto inicial é de facto igual para todas as situações, e o texto resposta a `msg()` de cada subclasse foi correctamente encontrado e inserido.

Vamos finalmente apresentar um último exemplo que, embora não venha acrescentar nada aos anteriores, é um pouco mais real, e exemplifica um processo de concepção baseado nestas propriedades estudadas, que devemos passar a usar de forma intuitiva e imediata, porque nos oferece uma flexibilidade ímpar.

Considere-se que temos uma classe **`Empregado`** que representa a informação básica sobre os empregados de uma empresa, designadamente o seu nome e o número de dias de trabalho de dado mês. A classe guarda igualmente o valor do salário base por dia que é praticado, que é igual para qualquer empregado, tratando-se, por isso, de uma variável de classe. A classe define métodos de consulta para a informação essencial e define um método que, com base no salário dia e nos dias de trabalho efectivos, determina o vencimento mensal do empregado.

Pretende-se agora definir duas classes especiais de `Empregado`. O **`Gestor`** que tem definido um bónus salarial que entra na fórmula do salário mensal sob a forma de percentagem (ex.: salário x 1.35), e o **`Motorista`**, que, para além de um bónus, possui ainda um acréscimo salarial dependente do número de quilómetros realizado, multiplicado pelo valor definido na sua classe para cada quilómetro. Todas as classes devem ter definido o método `double salario()` que calcula o salário para cada instância sua.

```
public class Empregado {
   // Variáveis de Classe
   public static double salDia = 50.00;
   public static double getSalDia() { return salDia; }
   // Variáveis de Instância
   private String nome;
   private int dias;          // dias de trabalho no mês
   // Construtores
   public Empregado(String nome, int dias)
      { this.dias = dias; this.nome = nome; }
   // Métodos de Instância
   public int getDias() { return dias; }
```

```
    public double salario() { return dias * getSalDia(); }
    public String getNome() { return nome; }
  }

  public class Gestor extends Empregado {
    // Variáveis de Instância e Construtores
    private double bonus;
    public Gestor(String nm, int dias, double bon) {
      super(nm, dias); bonus = bon; }
    // Métodos de Instância
    public double getBonus() { return bonus; }

    public double salario()
       { return getSalDia()*this.getDias()*bonus;
     }
  }

  public class Motorista extends Empregado {
    // Variáveis de Classe
    public static double valorKm = 0.98;
    public static double mudaValKm(double nvKm) {
        valorKm = nvKm; }
    // Variáveis de Instância e Construtores
    private double bonus;
    private int km;
    public Motorista(String nm, int dias, double bon, int km) {
      super(nm, dias); bonus = bon; this.km = km; }
    // Métodos de Instância
    public double getBonus() { return bonus; }
    public double salario() {
     return getSalDia()*this.getDias()*bonus + (valorKm*km); }
  }
```

Vamos criar um programa principal que nos permita criar instâncias destas classes e testar os vários métodos programados. Sendo `Empregado` superclasse das classes `Gestor` e `Motorista` vamos usá-la para conter as várias instâncias criadas, independentemente dos seus tipos, para depois usarmos o polimorfismo e o *dynamic binding*:

```
import static java.lang.System.out;
public class TesteEmp {
  public static void main(String args[]) {
    //
    Empregado emp1, emp2, emp3;
    emp1 = new Empregado("Joao", 20);
    emp2 = new Gestor("Rui", 20, 2.2);
    emp3 = new Motorista("Ze", 20, 1.2, 200);
    out.println(emp1.getClass().getSimpleName() + " " +
        emp1.getNome() + " : " + emp1.salario());
    out.println(emp2.getClass().getSimpleName() + " " +
        emp2.getNome() + " : " + emp2.salario());
    out.println(emp3.getClass().getSimpleName() + " " +
        emp3.getNome() + " : " + emp3.salario());
  }
}
```

A instrução `emp1.getClass().getSimpleName()` deve ser lida da esquerda para a direita, e corresponde ao envio à instância `emp1` da mensagem `getClass()` que devolve a estrutura representativa da classe do receptor. Se a essa estrutura (objecto) enviarmos a mensagem `getSimpleName()`, é devolvida uma `String` que é o nome simples (sem prefixo) da classe dessa instância. Assim, confirmaremos para cada uma delas qual a sua verdadeira classe em tempo de execução (**tipo dinâmico**, portanto). O mesmo se verificará com o método `salario()` que deverá calcular o salário com a fórmula especificada em cada classe. Para os valores introduzidos em `main()` ao serem criadas as instâncias, os resultados apresentados pelo programa principal são os observados na Figura 5.31.

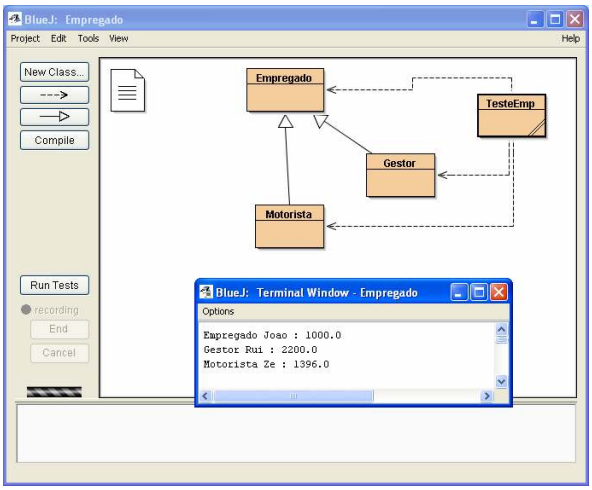

Figura 5.26 – Procura dinâmica de métodos – exemplo 3

As várias instâncias criadas foram guardadas em variáveis da classe `Empregado`. Numa delas temos uma instância de `Empregado`, noutra uma de `Gestor` e na terceira temos uma instância de `Motorista`. A execução do programa mostra que o interpretador sabe distinguir o tipo dinâmico de cada uma delas, sendo o seu nome apresentado pelo programa (cf. `Empregado`, etc.) Determina-o usando `getClass().getSimpleName()`, exactamente da mesma forma como o fizemos no nosso programa.

A qualquer variável de tipo `Empregado` podemos enviar as mensagens `getNome()` e `salario()` pois estão definidas na classe `Empregado`. O envio destas mensagens às três variáveis vai produzir resultados distintos em função dos seus tipos dinâmicos, sendo executados os métodos com o mesmo nome das classes correspondentes aos tipos dinâmicos das instâncias, produzindo os resultados que pretendíamos. Não temos assim qualquer preocupação de controlar que método é aplicado. O interpretador realiza essa selecção exactamente da forma como pretendemos.

Note-se no entanto que se as subclasses definirem os seus próprios métodos que aumentam a funcionalidade herdada da superclasse, relativamente a estes métodos específicos das subclasses não haverá compatibilidade com a superclasse.

Por exemplo, as instruções:

```
out.println("Bonus: " + emp2.getBonus());
out.println("Bonus: " + emp3.getBonus());
```

resultariam em erros de compilação pelas razões apresentadas. Assim, e caso pretendêssemos ter acesso a estes métodos, teríamos que realizar o *casting* para os tipos que os implementam, tal como se mostra nas instruções seguintes:

```
out.println("Bonus: " + ((Gestor)emp2).getBonus());
out.println("Bonus: " + ((Motorista)emp3).getBonus());
```

Esta necessidade de realizar *casting* apenas acontece quando nas subclasses introduzimos métodos específicos que nos obrigam a perder generalidade para os podermos usar. Esta perda de generalidade será sempre de evitar, e pode sempre ser evitada. Uma das formas seria codificarmos "forçadamente"o método `getBonus()` na classe `Empregado`, dando como resultado por exemplo 0, o que passaria a compatibilizar (do ponto de vista do compilador) todos os outros métodos `getBonus()` que o redefinem nas subclasses. No capítulo seguinte veremos uma forma mais elegante de o fazer.

O **princípio da substituição,** o **polimorfismo das variáveis** e o **algoritmo de procura dinâmica de métodos**, associados à **herança**, conferem à PPO um poder expressivo, de generalização e de extensibilidade do código ímpares.

5.4.3 TIPOS COVARIANTES E REDEFINIÇÃO DE MÉTODOS

A partir de JAVA5, e apenas para tipos referenciados, o tipo de resultado de um método que redefine um método de uma superclasse pode não ser exactamente o tipo do método redefinido mas um subtipo deste (**tipo covariante**). Esta regra generaliza o princípio da substituição aos tipos de resultado dos métodos que redefinem métodos das superclasses.

Assim, por exemplo, se na classe `Ponto2D` tivermos definido um método que nos permita realizar a cópia do estado interno de um ponto 2D, cf. o código

```
public Ponto2D copia() {
   return new Ponto2D(x, y);
}
```

os métodos seguintes são redefinições do método anterior nas classes `Ponto3D` e `Pixel` pois os seus tipos de resultados são subtipos de `Ponto2D`.

```
public Ponto3D copia() {
   return new Ponto3D(this.getX(), this.getY(), z);
}

public Pixel copia() {
   return new Pixel(this.getX(), this.getY(), cor);
}
```

Dada a importância de possuirmos nas nossas classes um método que seja capaz de criar uma cópia de uma qualquer instância de uma dada classe, por exemplo, para que a mesma seja enviada como resultado de um método sem partilhar endereço com a instância, vamos na secção seguinte abordar a questão do ***clone*** de objectos.

5.4.4 CLONE DE OBJECTOS

A classe `Object` implementa, como vimos, um método `clone()` que é `protected` e devolve como resultado uma instância de `Object`. Sendo `protected`, este método apenas pode ser invocado no código das subclasses de `Object`, não podendo, por exemplo, ser aplicado no contexto de um programa principal para copiar uma instância.

Genericamente, este método faz uma cópia "*bit* a *bit*" do receptor da mensagem `clone()` para uma outra zona da memória de trabalho (*heap*). Porém, como o método é nativo e apenas trabalha com *bits*, não tem acesso aos tipos dos objectos que está a copiar, limitando-se a copiar blocos de memória. Como não sabe o que copiou, devolve o tipo mais genérico possível, `Object`, e passa ao programador a responsabilidade de fazer o *casting* para o tipo do objecto que pediu que fosse copiado, tal como se mostra a seguir.

```
Ponto2D p1 = (Ponto2D) p.clone();  // clone seguido de cast
```

Porém, mesmo assim, para que uma subclasse de `Object` como `Ponto2D` possa usar o método `clone()` outros requisitos mais complexos deverão ter que ser satisfeitos.

Sendo indiscutível a importância de possuírmos nas nossas classes, pelo menos para uso da própria classe e das suas subclasses, um método disponível que permita realizar uma **cópia correcta** de uma qualquer instância dessa classe, vamos analisar os vários tipos de cópia de objectos (*cloning* na terminologia usual).

Sendo, em JAVA, todos os objectos referenciados acedidos e passados por endereço, a simples atribuição de identificadores não é mais do que uma **cópia de endereços** e não dos seus conteúdos. Quando um método de instância recebe um identificador de um objecto como parâmetro, o que é recebido, de facto, é o endereço desse objecto. Se esse objecto for atribuído a uma variável de instância, é o seu endereço que é atribuído, e a variável de instância passa a **partilhar** o objecto com o identificador exterior. Qualquer alteração realizada no objecto através do seu identificador exterior, modifica também o valor da variável de instância. Não existe, portanto, qualquer encapsulamento.

Por outro lado, quando um método de instância tem de dar como resultado uma parte do estado interno de uma instância, se o fizer directamente devolvendo o nome da variável de instância de um tipo referenciado, o que está a

passar para o exterior é o endereço desta, pelo que, a partir de então, quem o receber pode modificar directamente o valor da variável de instância. Foi de novo quebrado o encapsulamento e a protecção de dados.

Claro que o que acabámos de dizer não se aplica a variáveis de tipos primitivos pois estas contêm valores, e valores são copiados. O que dissemos aplica-se a tipos referenciados ou objectos. Há, no entanto, desde já que distinguir entre objectos que são **mutáveis** e objectos que são **imutáveis**.

Um **objecto imutável** é um objecto que a partir do momento em que é instanciado não mais vai modificar o seu estado interno, porque na sua classe, seja por herança seja por definição local, não há métodos para tal. Se uma classe cria instâncias imutáveis e as suas subclasses também, então não precisamos de clonagem.

A classe `String` e as classes `Integer`, `Double`, etc., são exemplos de classes que geram **objectos imutáveis**. No entanto, uma variável de instância pode ser do tipo `String` e ser mutável, para tal bastando que exista codificado um método modificador que lhe possa atribuir uma nova *string*. São, portanto, duas questões distintas.

Vejamos então como copiar objectos. Consideremos uma classe `Data` definida como:

```
public class Data {
   private int dia;
   private int mes;
   private int ano;
   public Data(int d, int m, int a) {
      dia = d; mes = m; ano = a;
   }
   ...
}
```

e seja `Data dia1 = new Data(10, 9, 2005)`. Depois de `Data dia2 = dia1`, as duas variáveis `dia1` e `dia2` partilham a mesma instância de `Data` (Figura 5.27).

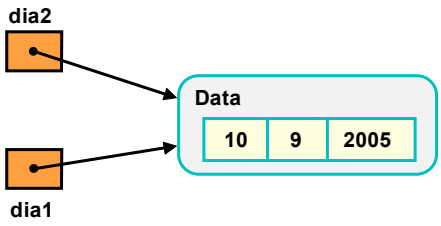

Figura 5.27 – Objecto partilhado após `Data dia2 = dia1`

No entanto, o que pretendemos é um método que nos garanta que quando copiamos `dia1` para `dia2`, a configuração de variáveis seja a que se apresenta a seguir (Figura 5.28).

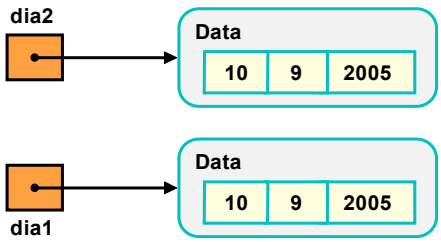

Figura 5.28 – Objecto clonado (Data dia2 = *clone* de dia1)

Em tal configuração, as duas variáveis possuem o mesmo valor, mas não há qualquer partilha pois são duas instâncias distintas.

Porém, o exemplo anterior não reflecte a dificuldade real de se realizar uma clonagem correcta, dado que o objecto a clonar, `dia1`, possui apenas atributos de tipo primitivo, logo, imediatamente copiáveis por valor.

Por isso, consideremos agora uma situação em que o objecto a clonar possui variáveis de instância de tipo referenciado. Sejam `c1` e `c2` objectos da classe `Circulo`, classe que tem por variáveis de instância `raio`, de tipo `int`, e `centro` de tipo `Ponto`, e tomemos como exemplo o código que se apresenta em seguida:

```
Circulo c1, c2; c1 = new Circulo(3, new Ponto(-2, 6));
```

Nele se declaram duas variáveis de tipo `Circulo` e se atribui a `c1` uma instância de círculo. A Figura 5.29 representa o resultado da atribuição.

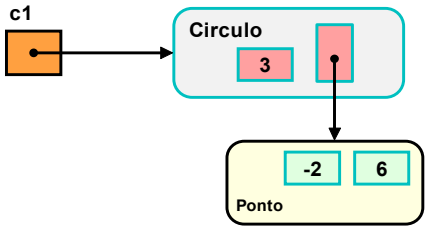

Figura 5.29 – Objecto da classe `Circulo`

Como vimos antes, escrever `c2 = c1` conduz à partilha dos objectos, e é, portanto, uma cópia incorrecta.

Outra possibilidade de cópia incorrecta, seria copiarmos bem a parte que diz respeito à instância de círculo (ou seja sem a partilhar e dando-lhe memória própria) mas não copiarmos bem, ou seja, partilharmos, a instância correspondente ao ponto que é o centro do círculo (Figura 5.30).

Este tipo de cópia, é a que resulta de se realizarem cópias de *bits*. Os *bits* da instância são copiados para outra zona de memória e o endereço é atribuído à variável do lado esquerdo da atribuição (ex.: `c2 = copy(c1);`). Todos os campos que são valores, ficam com valores iguais aos da instância copiada, e todos os campos que são referências para objectos têm tais referências copiadas, ou seja, são copiados os endereços. Assim, a nova instância referencia os mesmos objectos que a original (Figura 5.30).

Esta cópia, que partilha variáveis de instância, é designada por *shallow copy* (desde os operadores *copy* e *deepCopy* de Smalltalk).

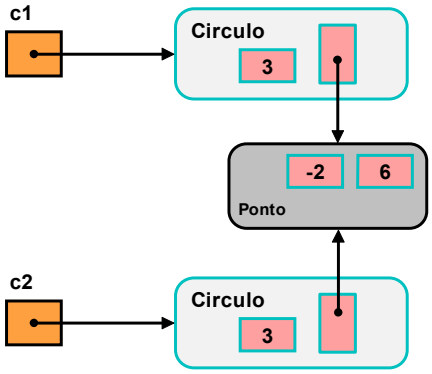

Figura 5.30 – *Shallow Copy* de c1

A cópia das variáveis de tipos referenciados, como não se distinguem das outras, é feita copiando os *bits* que representam os seus endereços, mas não os seus valores (objectos). Assim, no final, todas serão partilhadas, pois todos os seus endereços são copiados para o objecto resultado. No exemplo, ainda que seja criada uma nova instância de `Circulo`, o objecto `Ponto(-2, 6)` passa a ser partilhado pelas variáveis que nas instâncias de círculo `c1` e `c2` representam o ponto que é o centro do círculo, podendo qualquer destas modificar os valores do ponto de forma independente e livre. Quando mudarmos a posição do centro do círculo `c2` mudaremos também a posição do centro do círculo `c1`!

É este o tipo de **clonagem** que o método `Object.clone()` realiza, e que é portanto uma ***shallow copy***. Naturalmente que é uma cópia na maioria dos casos insuficiente, à qual se adicionam um enorme conjunto de dificuldades na utilização correcta do método. O que pretendemos ter, de facto, é a situação apresentada na Figura 5.31:

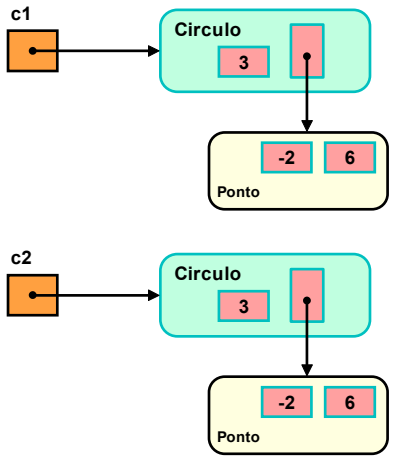

Figura 5.31 – *Deep Copy* ou *clone* de c1

Finalmente, teremos a cópia correcta, ou o verdadeiro *clone*, onde não há qualquer forma de partilha, e em que a aplicação da clonagem deve ser realizada estruturalmente, pois as próprias variáveis de instância podem ser compostas, etc. Esta cópia estrutural, completa e sem partilhas, designa-se, por tal razão, ***deep copy***. Será daqui em diante a perspectiva que usaremos quanto ao que deve entender-se por **realizar o clone** de um objecto.

A criação de *clones* de objectos é muito importante numa abordagem ao desenvolvimento de aplicações baseada em princípios da Engenharia de *Software* como o encapsulamento, por várias razões. Em primeiro lugar, porque sendo JAVA uma linguagem baseada numa semântica de apontadores, o mecanismo de passagem de parâmetros é o de passagem por referência. Logo, ou copiamos o objecto que nos é passado como parâmetro, ou onde quer que o guardemos ele está a ser partilhado. Assim, "alguém" o pode alterar. Igualmente por tal motivo, quando, em todo ou em parte, devolvemos como resultado de um método objectos que fazem parte da estrutura interna das instâncias, se não realizarmos uma cópia estamos a enviar para o exterior "canais de acesso" a tais dados que são incontroláveis.

Assim, teremos que garantir nas nossas classes a implementação de um método que garanta a possibilidade de realizarmos ***deep copy*** das instâncias ou, pelo menos, as designadas ***safe copy***, que são aquelas onde se relaxa a cópia efectiva dos objectos imutáveis, tais como `String`, `Integer`, etc., sempre que tal implicar esforço relevante.

Pelas razões atrás apontadas, não utilizaremos o método `clone()` da classe `Object` e vamos definir nas nossas classes um método próprio. Se em versões anteriores de JAVA alguma complicação poderia existir com o nome de tal método, em JAVA5, com a possibilidade de o tipo de resultado de um método redefinidor ser subtipo do redefinido, então os nossos método de cópia podem chamar-se `clone()` e devolver um objecto do tipo da classe onde forem implementados, como se mostra nos exemplos a seguir:

```
public Ponto2D clone() {
public Pixel clone() {
public Ponto3D clone() {
```

Todos estes métodos são portanto redefinições correctas do método `clone()` de `Object`, pois têm o mesmo nome, modificador de acesso mais lato e tipo de resultado que é um subtipo de `Object`. Quando usados, estes métodos devolverão de imediato um objecto da mesma classe do objecto "clonado", não sendo necesário realizar qualquer *casting*:

```
Ponto2D p2 = p1.clone();
Pixel pix2 = pix1.clone();
```

A implementação destes métodos de cópia vai ser realizada tendo por base os construtores de cada classe. Torna-se por isso importante que cada classe possua um construtor que receba todos os campos de um objecto dessa classe, e os copie para criar uma nova instância. Este construtor designa-se por **construtor completo**.

Por exemplo, um construtor completo para a classe `Circulo` seria:

```
public Circulo(int raio, Ponto2D centro) {
   new Circulo(raio, centro.clone());
}
```

Dado que o código dos três métodos `clone()` anteriores foi apresentado em exemplos anteriores sob o nome `copia()`, e não apresenta dificuldade, pois, em tais casos apenas houve a necessidade de copiar variáveis de tipos primitivos, vejamos agora um exemplo onde é necessário copiar objectos, usando a classe `Circulo` definida como:

```
public class Circulo {
   private double raio;
   private Ponto2D centro;
   // Construtores
   public Circulo(double r, Ponto2D p) {
      raio = r; centro = p.clone();
   }
   public Circulo(Circulo c) {
      raio = c.getRaio();
      centro = c.getCentro().clone();
   }
   public double getRaio() { return raio; }
   public Ponto2D getCentro() { return centro.clone(); }
   public void mudaRaio(double r) { raio = r; }
   public void mudaCentro(Ponto2D p) { centro = p.clone(); }                public
Circulo clone() { ... }
}
```

Haverá a partir de agora uma preocupação nossa, metodológica, de implementar em cada classe desenvolvida o método `clone()`, para que possa ser invocado nas classes que as usem.

No código do construtor da classe `Circulo`, recebemos uma instância de `Ponto2D` como parâmetro, e como vamos atribuir esse objecto à variável de instância `centro`, então temos que fazer o `clone()` desse `Ponto2D`. Tal só é possível se tivermos implementado `clone()` em `Ponto2D`, o que é fundamental para que se possam "clonar" pontos.

Copiar um círculo, será criar uma nova instância de `Circulo` usando o construtor completo, e copiar os atributos do objecto receptor da mensagem, ou seja, copiar o valor `double` do raio, e realizar o `clone()` do ponto que é o centro da instância. Teremos então o código seguinte:

```
public Circulo clone() {
   return new Circulo(raio, centro.clone());
}
```

Neste último caso, como o próprio construtor `Circulo(int r, Ponto2D p)` faz o `clone()` do parâmetro `Ponto2D`, poderíamos escrever apenas:

```
      return new Circulo(raio, centro);
```

No entanto, fazer o `clone()` do parâmetro `Ponto2D` no construtor é fundamental, pois este é usado em muitas mais situações do que apenas as que se prendem com a criação de *clones*, e a variável de instância `centro` não deve nunca partilhar endereços com pontos exteriores. Note-se que apenas não haveria partilha de endereço com pontos exteriores, se a invocação do construtor num qualquer método fosse realizada usando a seguinte expressão (em que o ponto tem endereço mas não referência):

```
Circulo c = new Circulo(4, new Ponto2D(-1, 0));
```

Um outro construtor que é muito útil, e que pode oferecer uma outra forma de definição do método `clone()`, é o designado **construtor de cópia** (*copy constructor*), que recebe um `Circulo` como parâmetro (e não as partes) e cria uma nova instância que é uma fiel cópia do argumento:

```
// Construtor completo
public Circulo(Circulo circ) {
   raio = circ.getRaio(); centro = circ.getCentro().clone();
}
```

A existência deste construtor na classe `Circulo`, permitirá que o método `clone()` possa ser definido simplesmente como:

```
public Circulo clone() {
   return new Circulo(this);  // this => o receptor de clone()
}
```

A utilização do método `clone()` nas subclasses destas classes será feita em função da necessidade de criar cópias das suas variáveis de instância, não havendo, em geral, a necessidade de usar `super.clone()` directamente. Por exemplo, se tivermos uma classe `Elipse` definida como subclasse de `Circulo`, teríamos o código:

```
public class Elipse extends Circulo {
  private Ponto2D ponto2;
  public Elipse (Elipse e) {
    super(((Circulo) e).clone());
    ponto2 = e.extremo2().clone(); }
  public Elipse(double alt, Ponto2D p1, Ponto2D p2) {
    super(alt, p1.clone()); ponto2 = p2.clone(); }
  public double getAlt() { return super.getRaio(); }
  public void mudaAlt(double alt) { super.mudaRaio(alt); }
  public void mudaPonto1(Ponto2D p) {
    super.mudaCentro(p.clone()); }
  public void mudaPonto2(Ponto2D p) { ponto2 = p.clone(); }
  public Ponto2D extremo1() { return super.getCentro(); }
  public Ponto2D extremo2() { return ponto2; }
  public Elipse clone() { return new Elipse(this); }
}
```

Note-se que a única utilização do método `clone()` da superclasse se fez apenas porque convertemos uma `Elipse` num `Circulo`, aliás sem necessidade e apenas para exemplo.

5.5 A CLASSE `Object`

Há muito que vínhamos afirmando que se pretendermos desenvolver classes com alguma generalidade deveríamos usar como classe de representação a classe que se encontra no topo da hierarquia de classes de JAVA, a classe `Object`.

Torna-se agora compreensível que, sendo a classe `Object` o topo da hierarquia de classes de JAVA, logo a superclasse de todas as classes, a uma variável de tipo estático `Object` é possível associar um valor que é uma instância de **uma qualquer classe** de JAVA.

Assim, ao definirmos, por exemplo, uma classe **`Stack`** de **`Object`**, estamos de facto a definir *stacks* não só genéricas, em que `Object` pode dinamicamente representar instâncias de qualquer subclasse, mas, também, heterogéneas ou polimórficas, dado que os elementos da *stack* não necessitam de ser todos do mesmo tipo dinâmico.

No entanto, e dado que a classe `Object` possui uma API muito limitada, haverá que ter em atenção que, do ponto de vista estático, não serão muitas as mensagens que poderão ser enviadas a variáveis do tipo estático `Object`. Assim, ainda que em termos de tipos estáticos faça sentido usar `Object` nos parâmetros e resultados de métodos, em termos dinâmicos deveremos fazer *casting* de `Object` para o tipo específico que se pretende. Só assim se garante a utilização correcta e total das instâncias de tal tipo, aumentando por exemplo o número total de

mensagens que lhes podem ser enviadas, já que passando de `Object` para o seu tipo efectivo, a instância poderá responder às mensagens que correspondem aos seus métodos, bem como aos herdados.

A generalidade das colecções de JAVA antes de JAVA5 (e não só, como veremos), tinha por base a utilização de instâncias de `Object` na representação dos seus elementos.

5.6 PROGRAMAÇÃO GENÉRICA VIA POLIMORFISMO

Continuando a utilizar as classes `Empregado`, `Motorista` e `Gestor` anteriormente definidas, vamos agora apresentar um exemplo de como a programação se torna genérica e simples se usarmos os princípios e mecanismos estudados anteriormente, em especial na construção de colecções de objectos.

Como neste momento ainda não estudamos as classes de JAVA que implementam as colecções ou estruturas de dados mais comuns (cf. listas, conjuntos, *hashtables*, etc.), o exemplo terá que ser dado, sem prejuízo algum do mesmo, usando uma estrutura *array*.

Os *arrays* têm a propriedade particular de serem **estruturas covariantes**, ou seja, respeitam a hierarquia de tipos dos seus elementos. Assim um *array* `Ponto2D`[] é compatível com um *array* `Ponto3D`[] e com um `Pixel`[] mesmo para atribuições directas. No entanto, e para o exemplo seguinte, o importante é notar-se que, se pretendemos guardar num *array* instâncias de uma classe e de várias suas subclasses, a decisão correcta é, de imediato, definir o *array* como sendo do tipo da superclasse, tornando-o o mais genérico e compatível possível. Os mecanismos de *dynamic binding* e polimorfismo ajudarão no resto da concepção.

Admitamos, então, que se pretende criar um *array* que seja capaz de conter instâncias de **`Empregado`** e suas subclasses. Pensando de imediato qual será a mais correcta declaração a fazer para tal *array*, a procura de generalidade determina que, estaticamente, este seja declarado como sendo do tipo mais genérico dos tipos em questão, ou seja, **`Empregado`**, pois este é compatível com todas as suas subclasses.

Teremos assim a seguinte declaração (sendo `MAX_EMP` uma constante):

```
Empregado[] empresa = new Empregado[MAX_EMP];
```

array no qual, em seguida, vamos introduzir diferentes instâncias dos vários tipos de empregados da empresa, por uma ordem qualquer, usando os respectivos construtores (claro que esta não é a forma usual de preencher as posições de um *array*):

```
empresa[0] = new Empregado("Joao", 20);
empresa[1] = new Gestor("Rui", 20, 2.2);
empresa[2] = new Motorista("Ze", 20, 1.2, 200);
 . . .
empresa[i] = new Motorista("Beto", 15, 1.4, 500);
```

Se agora pretendêssemos num programa principal calcular o somatório dos vencimentos a pagar a todos os empregados da empresa, tal como registados no *array*, onde temos Motoristas, Empregados e Gestores, não teremos que nos preocupar com os seus tipos, já que o respectivo método `salario()` será correctamente invocado em função do tipo deste. Assim, o algoritmo torna-se extremamente simples, tal como se apresenta a seguir:

```
// Cálculo do somatório dos vencimentos de todos os
// empregados, com determinação automática do salário
// em função da sua classe (categoria).
double totVenc = 0.0;
for(int i=0; i < numEmp ; i++) {
   totVenc += empresa[i].salario();
}
out.println("Total de Vencimentos = " + totVenc);
```

O código de `salario()` a executar para cada empregado representado por `empresa[i]`, será correctamente seleccionado pelo algoritmo de ***dynamic method lookup***, conforme o tipo dinâmico da instância armazenada em tal índice do *array*, pelo que a escrita do código se torna não só simples como **genérica**, para a hierarquia que tem por

topo a classe `Empregado`. Código mais genérico apenas usando `Object`. Não usamos porque, naturalmente, apenas pretendemos inserir no *array*, instâncias de `Empregado` e das suas subclasses. **`Empregado`** é um limite superior de tipos.

Se posteriormente adicionarmos mais tipos de `Empregado`, ou seja, mais subclasses, este algoritmo continua a funcionar correctamente, quer cada uma delas implemente a sua fórmula de cálculo de `salario()` quer mesmo se não a implementar, em cujo caso será usada a da superclasse. Temos pois **extensibilidade** garantida.

A não completa compreensão destes princípios e mecanismos, associada em certos casos a muitos anos de programação imperativa, conduz, por vezes, à escrita de código não só redundante como não extensível, como o que se apresenta a seguir:

```
for(int i=0; i < a.numEmp; i++;) { // exemplo de má codificação
  if (empresa[i] instanceof Empregado)
       totVenc += ((Empregado) empresa[i]).salario());
  else
       if (empresa[i] instanceof Motorista)
         totVenc += ((Motorista) empresa[i]).salario());
       else
          ... // fim de exemplo de má codificação
```

onde é usado o operador boleano `instanceof` para testar de que tipo dinâmico é a instância, para, em seguida, fazer o *casting* para a sua classe (como por exemplo na expressão **`((Empregado) empresa[i])`**), para que o método `salario()` invocado seja garantidamente o da classe da instância.

Desconhecendo que o algoritmo de procura dinâmica de métodos faz exactamente este trabalho em tempo de execução, este programador está a programar por si própio parte de tal algoritmo, o que é, portanto, um esforço redundante. Para além disso, o que é muito mais grave, provavelmente sem o saber, o seu código não é nem genérico nem extensível. Senão, vejamos porquê.

Vamos admitir que posteriormente é criada uma classe **Secretaria**, igualmente subclasse de **Empregado**, que apresenta a mesma API, ou seja, responde às mesmas mensagens. É óbvio que caso instâncias desta nova classe passem a fazer parte do *array*, o ciclo `for` escrito no primeiro exemplo não necessita de qualquer modificação. Caso surja uma instância de **Secretaria**, ela será tratada de forma igual a todas as outras, sendo invocado o método `salario()` respectivo.

Ou seja, esse código mantém-se inalterado, apesar de termos realizado uma extensão à hierarquia de classes. Esta é a propriedade da **extensibilidade**, ou seja, da garantia da continuidade do código, e não a sua caducidade, apesar das várias modificações que posteriormente podem ser introduzidas na estrutura da informação do problema.

Porém, o código do programador que desconhece estes princípios não resiste a tais modificações, sendo necessário alterá-lo de cada vez que uma nova subclasse de **`Empregado`** é criada. No exemplo, haveria agora a necessidade de alterar o código para testar se se trata de uma instância de **`Secretaria`** e realizar o respectivo tratamento. E, de forma igual, a cada nova extensão.

Tal não faz sentido em PPO, dado que o que se pretende na utilização do paradigma é exactamente a flexibilidade oposta que, como se viu, é atingível sem esforço mas com conhecimento dos mecanismos do mesmo.

5.7 PROGRAMAÇÃO INCREMENTAL

O mecanismo de herança introduz uma metodologia de programação que se pode designar por **programação incremental**. De facto, como pudemos anteriormente verificar através dos exemplos apresentados, a efectiva construção de uma nova classe consiste na resolução de "equações", que se traduzem literalmente em expressões do tipo:

Classe Pretendida = SuperClasseExistente + Δprog

SubInstância ⇔ SuperInstância + Δvar + Δmet

De uma outra forma, se pretendemos criar uma nova classe **B** possuindo uma dada estrutura e um dado comportamento, e se já temos implementada uma classe **A** com dada estrutura e comportamento, então, se a nova classe for subclasse desta, teremos apenas que programar a diferença entre o que é herdado de **A** e o que se pretende ter em **B**. **A solução é, naturalmente, uma classe X definida como contendo a diferença**.

Torna-se agora compreensível que se afirme que, para a construção de uma classe nova, quanto mais pudermos herdar de classes existentes, menos esforço teremos na programação da nova classe. Porém, tal como já foi afirmado antes, para que tal esforço possa ser de facto reduzido, é absolutamente necessário um conhecimento profundo das classes já existentes em JAVA. Só assim poderemos fazer um uso efectivo dos mecanismos de reutilização à nossa disposição em PPO.

5.8 O PROBLEMA DA CLASSIFICAÇÃO CLASSES NOVAS COMO SUBCLASSES DE `Object`

A correcta classificação de uma nova classe é uma tarefa relativamente complexa. Aliás como sabemos do dia-a-dia, qualquer processo classificativo é complexo pois implica um conhecimento perfeito da entidade a classificar, bem como das regras a empregar para realizar a sua correcta classificação.

Existem mesmo múltiplas razões que podem ser encontradas para criar classes como subclases de classes já existentes, algumas até que podem parecer muito pouco evidentes para quem se inicia no processo.

Por exemplo, um caso particular de especialização de uma classe consiste em impor-lhe restrições que normalmente não tinha. Uma classe `Quadrado`, por exemplo, pode ser criada a partir da classe `Rectangulo` impondo-lhe certas restrições, quer via construtores quer via modificadores de estado. Uma classe `ParqueLimitado` poderia ser um outro exemplo de uma subclasse de uma classe `Parque`. Estes são exemplos de subclassificação para **limitação**. O exemplo do `Ponto2D` e `Ponto3D` é considerado um caso típico de subclassificação para **extensão**, porque o comportamento herdado não foi modificado e novo comportamento foi adicionado. Outras poderiam ser referidas.

Em PPO, um bom conhecimento das classes existentes e do que delas pode ser herdado é que, mais do que catalogar subclassificações, em muito pode auxiliar à definição de novas classes. O seu desconhecimento faz com que muitas vezes, por displicência, as novas classes sejam classificadas como subclasses directas da classe `Object`.

Reduzindo tal prática a uma situação absurda em que na hierarquia de classes todas as classes existentes, excepto `Object`, fossem subclasses de `Object`, a herança não seria necessária, dado que, praticamente, apenas existiria um nível de classes. A reutilização seria mínima e cada classe nova deveria ser programada de raiz (Figura 5.32).

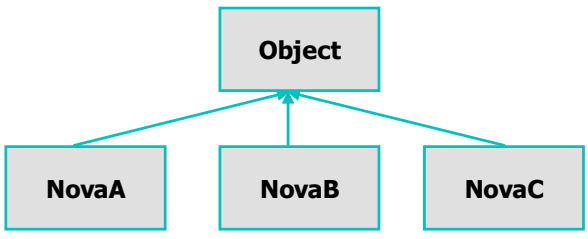

Figura 5.32 – Subclassificação pobre - subclasses de Object

A um maior esforço de classificação das classes corresponderá sempre uma maior reutilização de código, por herança, e, em consequência, menor esforço de programação.

5.9 HERANÇA *VERSUS* COMPOSIÇÃO

Muitas das dificuldades que surjem no desenvolvimento de novas classes, resultam de alguma tendência para confundir herança com composição. Estas duas formas de relacionamento entre classes, extensível às suas instâncias, são, no entanto, muito distintas.

Quando uma classe é criada por **composição** de outras, tal consiste em definir que se entende que as classes agregadas fazem claramente **parte de**, ou são **partes**, da classe em definição. Assim, qualquer instância da nova classe vai ser constituída por partes que são instâncias das diversas classes que foram agregadas. Estas partes, isto é, estas instâncias das classes agregadas que ajudaram a criar a instância da classe agregadora, têm uma ligação temporal, e até de tempo de vida, muito forte com a instância de que são parte. De facto, quando tal instância deixa de estar referenciada no ambiente de programação e é destruída pelo *garbage collector* de JAVA ou por um método destrutor explícito, tais partes são igualmente destruídas. O seu tempo de vida está completamente ligado ao tempo de vida da instância de que fazem parte.

Tem que ser claro que esta relação de composição nos permite, tal como no dia-a-dia, criar entidades complexas pela incorporação nestas de outras entidades com estruturas e comportamentos que podem ser também independentes, mas que, neste caso, vão servir como **partes** do todo que se pretende construir. Por exemplo, se se pretende especificar uma classe `Automovel` e já temos definidas as classes `Pneu`, `Motor`, `Chassis`, `Travão`, etc., torna-se óbvio que a classe `Automovel` deve ser criada usando **composição**, já que pretendemos que seja constituída por uma série de **partes** que, em conjunto, irão determinar a sua estrutura e comportamento.

Muitas vezes a existência de herança múltipla leva a que se tentem hierarquias como a apresentada na Figura 5.33, na qual uma interpretação errada do que se deve entender por herança múltipla, em particular se apenas se pretendia agregação, conduz a uma hierarquia segundo a qual um avião vai, supostamente, ter o seu comportamento e estrutura definidos herdando-os de uma asa, de uma turbina, de um trem e de um travão. Além do mais, no máximo, podemos ter uma unidade de cada, o que não é manifestamente satisfatório em geral, e pior ainda no caso do exemplo.

Numa linguagem de herança simples esta situação não ocorreria certamente, dado que qualquer tentativa de classificar `Avião` como subclasse de `Asa` ou de `Turbina` pareceria de imediato tão ridícula que conduziria à rejeição de tal subclassificação, pois o relacionamento **Asa is-a Avião** não teria sentido, mas sim **Asa part-of Avião**.

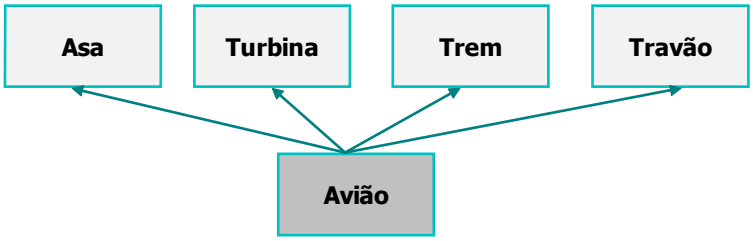

Figura 5.33 – Interpretação errada da herança múltipla

Quando uma classe a ser criada, para além de quase inevitavelmente necessitar de possuir partes que são elas próprias construídas por composição, for na sua estrutura e também no seu comportamento uma especialização ou refinamento de uma outra classe já definida, que certamente já agregou grande parte do que esta necessita de agregar, então, estamos perante uma **especialização** ou **derivação**, e o mecanismo a usar é de **subclassificação**, tendo por suporte o mecanismo de **herança**.

Dentro desta perspectiva, uma classe `Aviao` nunca seria subclasse de `Asa`, mas poderia ser uma subclasse de `Veículo_a_Motor`, ainda que tal dependesse da efectiva definição da classe `Veículo_a_Motor`, e seria ainda composta por um dado número de instâncias de `Turbina`, de `Asa`, de `Trem`, etc.

5.10 CRIAÇÃO DE CLASSES: REGRAS ANOTADAS

Apresentam-se em seguida algumas regras que sintetizam um conjunto extenso de directivas e princípios que foram sendo apresentados ao longo deste e de anteriores capítulos relativamente à criação de classes em JAVA (e na maioria das linguagens de PPO). Estas regras que são muito importantes para que, de forma metódica, sistemática e rigorosa, porque baseada em constatações, se possam atingir de facto os objectivos enunciados de garantia de concepção e desenvolvimento de aplicações possuindo propriedades fundamentais em Engenharia de *Software*, tais como modularidade, generalidade, clareza, extensibilidade, segurança e facilidade de manutenção.

Regra 1: Os dados devem ser sempre `private`.

Caso se pretenda de forma rigorosa garantir a absoluta preservação das regras do encapsulamento, então, todas as variáveis de instância devem ser declaradas como sendo `private`. Desta forma, e porque tais variáveis não são herdadas, a única forma de externamente ter acesso aos seus valores será através dos métodos de consulta e de alteração definidos na sua própria classe. Como vimos no capítulo 1, mantendo-se os dados privados, futuras mudanças na sua representação interna não geram quaisquer alterações nos utilizadores da classe e possibilitam detectar erros muito mais facilmente, já que os únicos métodos que as podem manipular são os métodos modificadores definidos na sua classe.

Porém, nunca poderemos esquecer que a completa obediência ao princípio do encapsulamento impõe a quem programa a responsabilidade de disponibiliar na API das classes os imprescindíveis métodos de consulta e modificação. Qualquer esquecimento desta responsabilidade conduz à criação de classes "surdas-mudas", ou seja, classes não utilizáveis do exterior, dado não existirem formas de acesso aos dados, nem directamente nem via métodos.

Regra 2: Inicializar sempre explicitamente os dados através de construtores.

Apesar de sabermos as regras usadas pelos construtores na inicialização de dados, quer sejam de tipo simples quer sejam objectos, torna-se muito importante para a clareza e compreensão do código que todos os construtores de uma classe sejam programados, indicando de forma explícita quais os valores inicialmente atribuídos a todas as variáveis de instância. Desta forma, salvo em situações de herança, os construtores por omissão que JAVA associa implicitamente a cada nova classe devem ser sempre redefinidos por um construtor programado, que inicialize todas as variáveis necessárias de forma explícita, isto é, codificada.

Regra 3: Usar *deep copy* em parâmetros de entrada de construtores e métodos, e em resultados de métodos que possam dar acesso às variáveis de instância.

A atribuição de valores às variáveis de instância de tipos referenciados e a devolução de resultados de métodos de consulta, são "cavalos de Tróia" ideais para acessos exteriores ao interior dos objectos que, via encapsulamento, pretendemos absolutamente privados. Assim, com bom senso, o uso de *deep copy* nestas duas circunstâncias permite garantir privacidade e segurança, propriedades muito importantes.

Regra 4: Não usar demasiados tipos básicos numa classe. Quando tal acontecer procurar substituí-los, em grupo, por uma dada classe.

Se numa classe são definidas diversas variáveis de instância independentemente, mas que possuem algum relacionamento entre si, tal significa que estas, no seu conjunto, podem ser estruturadas como sendo variáveis de instância de uma nova classe.

Por exemplo, numa classe `FichaAluno` definiram-se, entre outras, as seguintes variáveis de instância:

```
private String rua;
private String numero;
private String localidade;
private String codigo_postal;
private String cidade;
```

Estas variáveis de instância isoladas, podem, se tomadas em conjunto, ser a base para a criação de uma classe `Morada`, que poderia ter a si associados métodos que permitissem alterar o código postal, editar o número da rua, enfim, alterar de uma forma geral a morada. Desta forma, para além desta funcionalidade ficar de forma mais clara associada a instâncias de `Morada` e não a instâncias de `FichaAluno`, nesta classe teríamos apenas que declarar:

```
private Morada mora;
```

Se porventura pretendêssemos guardar a morada em tempo de aulas e a morada em tempo de férias, em vez de duplicarmos as cinco variáveis de instância inicialmente definidas, teríamos apenas que declarar duas variáveis de instância do tipo `Morada` que, para além de permitirem representar tal informação, têm a si associados métodos específicos para consulta e alteração:

```
private Morada mora_aulas;
private Morada mora_ferias;
```

Regra 5: Nem todas as variáveis de instância e classe necessitam de selectores e modificadores.

Voltando ao exemplo da ficha de aluno, se tivermos uma variável de instância que representa a data da primeira inscrição do aluno na universidade, e/ou uma variável de instância que guarde a sua média de acesso, e/ou até uma instância de uma classe designada por `Filiacao`, cujas instâncias contêm os nomes do pai e da mãe do aluno, e admitindo que garantidamente estes dados foram correctamente introduzidos aquando da criação da instância, então, dada a sua natureza, estes são claramente exemplos de dados e variáveis de instância imutáveis, ou seja, apenas consultáveis.

Noutras situações, as variáveis de instância não são sequer consultáveis. Todos os que jogam certos jogos de cartas em computador sabem que sendo o baralho do jogo gerado aleatoriamente, não é consultável. Todos os que jogam *Minesweeper* sabem que o campo de minas não é consultável, embora seja modificável.

Regra 6: Usar uma forma normalizada para as definições de classes.

Qualquer que seja a forma normalizada escolhida, desde que seja aceite pelo compilador de JAVA, deve ser normalizada a disposição das várias secções que formam uma declaração completa de uma classe.

Na definição de tal forma normalizada, há que ter em atenção que os utilizadores de uma classe estão sempre mais interessados na interface pública da mesma, API, do que nos detalhes de implementação e, portanto, muito mais interessados em métodos do que em variáveis (às quais nem terão sequer acesso directo caso estas sejam `private`).

Nos exemplos apresentados ao longo deste livro, e salvo razões de apresentação, as declarações de classes são divididas em secções apresentadas pela seguinte ordem:

- Constantes de classe;
- Variáveis de classe;
- Métodos de classe;
- Variáveis de instância;
- Construtores;
- Métodos de instância `public`;
- Métodos que implementam Interfaces (ver capítulo sobre Interfaces);
- Métodos `protected` e `private`;
- Outros membros das classes (ver capítulo sobre Classes especiais de JAVA).

Quanto à acessibilidade dos métodos, deve seguir-se um critério de importância do ponto de vista do seu exterior, sendo apresentados em primeiro lugar os métodos `public`, em seguida os `protected`, depois os métodos *package*, e, por fim, os `private`.

Regra 7: Não criar classes com estrutura e API demasiado extensas; caso tal aconteça, procurar fazer a sua divisão em classes mais pequenas.

É uma regra de bom senso, dado não ser possível quantificar rigorosamente o que se deve entender por *classes demasiado extensas*. Em todo o caso, classes extensas denotam, em geral, demasiada concentração de responsabilidades numa única classe, pelo que, pelo menos, alguma preocupação com a sua extensão e clareza deverá ser tida em conta, e assim, sempre que possível, realizar a divisão da classe em classes que sejam conceptualmente mais simples.

Regra 8: Ser criterioso nos identificadores de classes, variáveis e métodos, de modo a que estes possuam a maior carga semântica possível.

Ainda que no código fonte de JAVA muitos maus exemplos de aplicação desta regra se possam encontrar (mas perdoa-se), os nomes das variáveis, classes e métodos devem reflectir a sua semântica. As classes devem ser em geral identificadas usando substantivos como, por exemplo, `Contador`, `String`, `Factura` e `Encomenda`, ainda

que por vezes a tal substantivo se possa associar um adjectivo, ou outra palavra qualquer, que indique de forma mais clara algum grau de especialização ou caso particular, tal como, por exemplo, `FacturaUrgente`, `StackLimitada`. Se os nomes das classes forem escritos em inglês, tais qualificativos surgem como prefixos e não como sufixos, como por exemplo, em `MainAddress`, `RadioButton` ou `HashMap`.

Os identificadores dos métodos devem denotar a acção associada, tendo já sido anteriormente referida a regra dos nomes `getX()` e `setX()` para os métodos que consultam e modificam variáveis de instância. Todos os outros são em geral verbos que representam as acções executadas, como por exemplo `junta(E e)` e `elimina(E e)`. Quando os nomes dos métodos visarem dar mais informação, cada uma das palavras que o formam, excepto a primeira, devem ser iniciadas por letra maiúscula tendo a missão de separadora, como por exemplo em `totalDePaises()`, `insereCodigo(Codigo c)`, etc.

Regra 9: Não confundir composição com herança.

Sendo mecanismos de relacionamento entre classes muito distintos, a confusão de um com o outro conduz a classes conceptualmente muito distorcidas. Composição tem a ver com B **ser parte de** A, enquanto que herança ter a ver com especialização e B **ser do tipo de** A.

Regra 10: Ser estudioso, ou seja, tentar ser um conhecedor profundo de tudo o que já existe implementado no ambiente de desenvolvimento, seja JDK ou outro qualquer.

Esta regra é de facto trabalhosa e a sua aplicação dispendiosa em tempo, mas é de facto um investimento fundamental. Caso seja aplicada por fases, duas fases podem ser consideradas: numa primeira fase, dever-se-á procurar ganhar conhecimento de classes existentes e apenas das suas API, ou seja, o que estas nos disponibilizam. Esta fase é, só por si, muito relevante para que se possa empregar uma estratégia de concepção e programação de aplicações com base em reutilização; numa segunda fase, a leitura e análise do próprio código fonte de JAVA poderá ajudar-nos a compreender melhor como certas classes são implementadas internamente e, eventualmente, daí extrair-se uma maior eficácia de utilização, extrapolando para outras situações.

Regra 11: Ser preguiçoso, isto é, após cumprir a Regra 10, tentar ser sempre apenas um reutilizador.

Esta regra só deve ser aplicada por aqueles que cumpriram eficazmente a regra anterior. Para esses, a regra ao ser exercitada é, simultaneamente, um prémio merecido, e um sinal de que um patamar elevado de conhecimentos foi atingido em PPO, pois a ideia fulcral da PPO é, exactamente, a reutilização de *software*.

5.11 SÍNTESE DO CAPÍTULO

Neste capítulo foram apresentados alguns conceitos que são de importância fundamental para a compreensão e utilização do paradigma da programação por objectos. A hierarquia simples de classes de JAVA, baseada no mecanismo de herança, possibilita que classes sejam criadas reutilizando classes já existentes. A hierarquia de classes introduz as noções de superclasse e subclasse, compatíveis entre si segundo o princípio da substituição.

O princípio da substituição, que garante a compatibilidade entre classes e as suas subclasses, vem introduzir o polimorfismo nas variáveis e a distinção importante entre o tipo estático e o tipo dinâmico de uma variável.

O algoritmo de procura dinâmica de métodos, o polimorfismo e o *dynamic binding*, vêm introduzir a possibilidade de desenvolvermos código baseado em supertipos, que, ao serem compatíveis com qualquer dos seus subtipos, possibilitam a criação de soluções que têm a propriedade de ser genéricas e facilmente extensíveis, como vimos.

Do ponto de vista pragmático, neste capítulo foram introduzidas as regras da herança, de métodos e de variáveis, e as regras de sobreposição de métodos. Foram introduzidos vários exemplos de correcta criação de subclasses, a utilização de `this` e `super`, e as formas correctas de reescrita de métodos.

A covariância nos tipos de resultado dos métodos, uma nova propriedade dos métodos de JAVA5, foi também analisada, em especial relativamente à liberdade adquirida para uma fácil redefinição do método `clone()` da classe `Object`.

As questões relativas à cópia correcta de objectos foi também analisada de forma muito profunda, dadas as suas implicações nas questões de privacidade e encapsulamento dos dados das classes. Uma alternativa à complexa e ineficiente forma de realizar a cópia de objectos usando o método `clone()` da classe `Object` foi apresentada, tendo por base uma utilização correcta dos construtores das classes. Regras de boa prática foram apresentadas.