Capítulo 6

Polimorfismo

A palavra polimorfismo vem do grego *poli morfos* e significa muitas formas. Na orientação a objetos representa uma característica onde se admite tratamento idêntico para formas diferentes baseado em relações de semelhança, isto é, entidades diferentes podem ser tratadas de forma semelhante conferindo grande versatilidade aos programas e classes que se beneficiam destas características.

6.1 – Upcasting

No capítulo anterior, você viu que um objeto pode ser usado como um membro do seu próprio tipo ou como um membro de sua classe base (superclasse). Podemos, então, tratar uma referência a um objeto, como se fosse uma referência a um objeto de sua classe base. Este tipo de conversão é denominado **upcasting**.

```
// Musica.java
class Instrumento {
  public void tocar() {
    System.out.println("Instrumento.tocar()");
  }
}

// Instrumentos de sopro tambem sao instrumentos
class Sopro extends Instrumento {
// Redefinicao do metodo:
  public void tocar() {
    System.out.println("Sopro.tocar()");
  }
}

public class Musica {

  public static void melodia(Instrumento i) {
    i.tocar();
  }

  public static void main(String[] args) {
    Sopro flauta = new Sopro();
    melodia(flauta); // Upcasting
  }
}
```

Exemplo 6. 1 - Herança e Upcasting

O método `Musica.melodia()` aceita como parâmetro uma referência a um objeto da classe `Instrumento` ou objetos de outras classes que herdem de `Instrumento`. No método `main()`, você pode acompanhar o que acontece quando uma referência a um objeto da classe `Sopro` é passada como parâmetro na chamada do método

`melodia()`, sem que seja necessário explicitar a conversão através de **cast**. A conversão explícita não é necessária pois a classe `Sopro` possui uma interface que é comum com a interface da classe `Instrumento` (a classe `Sopro` herda de `Instrumento`). Ao convertermos (upcasting) uma referência a um objeto da classe `Sopro` em uma referência a um objeto da classe `Instrumento` estamos limitando a interface do objeto original e ao mesmo tempo, garantindo a interface completa para um objeto do tipo `Instrumento`.

6.2 – Esquecendo o Tipo de um Objeto

Esta idéia poderá parecer estranha a primeira vista. Para que alguém esqueceria intencionalmente o tipo de um objeto? Isto na verdade é o que acontece quando realizamos **upcast**. Talvez fosse mais natural se o método `melodia()` simplesmente recebesse como argumento uma referência a um objeto do tipo `Sopro`. Note que ao fazer isso, você precisará escrever um novo método `melodia()` para cada tipo de instrumento presente no seu sistema. Suponha que tenhamos agido desta forma e que precisamos acrescentar os instrumentos `Corda` e `Percussão`:

```
// Musica1.java
class Instrumento {
  public void tocar() {
    System.out.println("Instrumento.tocar()");
  }
}
class Sopro extends Instrumento {
  public void tocar() {
    System.out.println("Sopro.tocar()");
  }
}
class Corda extends Instrumento {
  public void tocar() {
    System.out.println("Corda.tocar()");
  }
}
class Percussao extends Instrumento {
  public void tocar() {
    System.out.println("Percussao.tocar()");
  }
}
public class Musica1 {
  public static void melodia(Sopro i) {
    i.tocar(); }
  public static void melodia(Corda i) {
    i.tocar(); }
  public static void melodia(Percussao i) {
    i.tocar(); }
  public static void main(String[] args) {
    Sopro saxofone = new Sopro();
    Corda guitarra = new Corda();
    Percussao bateria = new Percussao();
    melodia(saxofone); // Nao realiza upcast
```

```
    melodia(guitarra);
    melodia(bateria);
  }
}
```

Exemplo 6. 2 – Sobrecarga no lugar de Upcast

O exemplo acima também funciona, embora seja menos adequado e mais trabalhoso. Você deverá especificar um método específico para cada tipo de instrumento que acrescentar. Isso significa mais esforço de programação, e também significa que se você quiser acrescentar novos métodos como `melodia()` ou novos tipos de instrumento, você terá bastante trabalho! Acrescido do fato de o compilador não lhe retornar nenhuma mensagem de erro se você esquecer de sobrecarregar um destes métodos, tornando a manipulação destes tipos de objeto uma tarefa bastante árdua.

Não seria melhor se você pudesse escrever apenas um método que recebesse como argumento um objeto da classe base? Ou seja, não seria melhor que você esquecesse por um momento que existem classes derivadas, e escrever seu código para se comunicar apenas com a classe base? Isso é exatamente o que o polimorfismo lhe permite fazer.

6.3 – Amarração Tardia

Um melhor entendimento sobre como o polimorfismo se comporta pode ser observado ao executarmos o programa Musica.java, que terá como saída: `Sopro.tocar()`. Esta é claramente a saída esperada, embora o funcionamento do programa possa parecer não fazer sentido, pelo menos a primeira vista. Ao analisarmos o método `melodia()`:

```
public static void melodia(Instrumento i) {
 i.tocar();
}
```

Verificamos que ele recebe como argumento uma referência a um `Instrumento`. Então, como o compilador poderia saber que esta referência a um `Instrumento` aponta para um objeto da classe `Sopro` neste caso e não a objetos das classes `Corda` ou `Percussão`? Na verdade, não é bem o compilador que se encarregará de chamar o método apropriado.

As linguagens de programação que oferecem suporte ao polimorfismo quase sempre o fazem através de amarração tardia, também conhecida como amarração dinâmica. Este mecanismo permite que o tipo de um objeto seja determinado em tempo de execução e que seus métodos apropriados sejam chamados. Ou seja, o compilador continua não sabendo o tipo do objeto, mas o mecanismo de chamada de métodos descobre dinamicamente a classe do objeto e se encarrega de fazer a chamada aos métodos corretos.

Em Java, todos os métodos de instância utilizam o mecanismo de amarração tardia, a menos que o método tenha sido declarado como *final* (neste caso, o método não pode ser sobrescrito e, portanto, não há como haver amarração tardia). Isto significa que não há necessidade do programador ficar tomando decisões sobre quando a amarração tardia ocorrerá. Ela acontece automaticamente.

6.4 – Usando Amarração Tardia

Como toda amarração de métodos em Java ocorre polimorficamente via amarração tardia, pode-se escrever código para comunicar com a classe base e saber que todos os

objetos de classes derivadas funcionarão apropriadamente usando o mesmo código. Em outras palavras, pode-se enviar uma mensagem para um objeto e deixar que ele identifique qual a coisa certa a fazer.

O exemplo clássico em POO é o dos objetos `Forma`. Ele é frequentemente usado porque é fácil de entender e visualizar. Este exemplo tem `Forma` como classe base e vários outros tipos derivados, tais como: `Circulo, Quadrado, Triangulo, etc.` A razão pela qual o exemplo funciona tão bem é porque é fácil dizer que um `Circulo` é um tipo de `Forma` e ser entendido. A figura 6.1 mostra um diagrama de classes com as relações do exemplo:

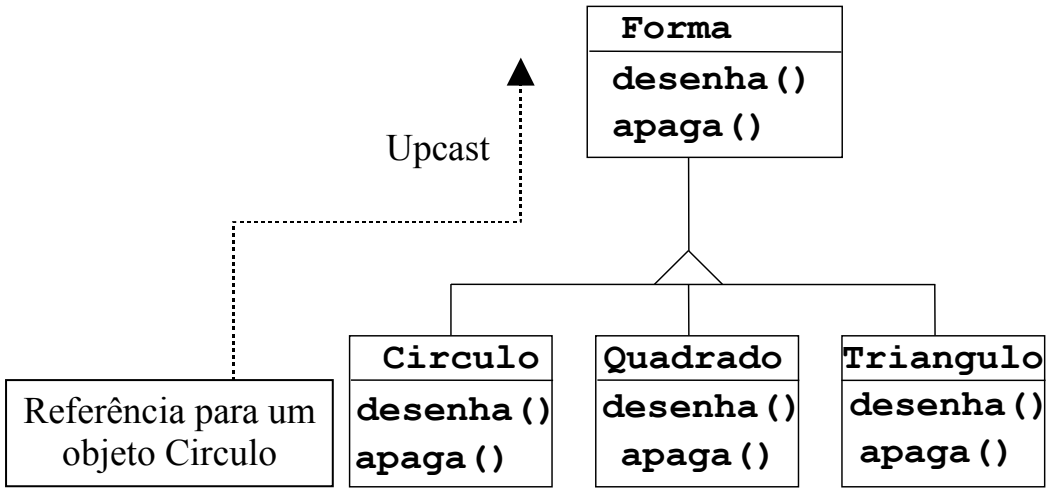


Figura 6. 1 - Hierarquia de Formas

O upcast poderia ocorrer em um comando tão simples quanto:

```
Forma s = new Circulo();
```

Aqui, um objeto `Circulo` é criado e a referência resultante é atribuída imediatamente para uma `Forma`, o que aparentemente poderia parecer um erro (de atribuição de um tipo a outro), mas que está correto porque um `Circulo` é uma `Forma` devido a herança. Quando um dos métodos da classe base é invocado (lembre que todos os métodos da classe base foram sobrescritos nas classes derivadas), tal como em

```
s.desenha();
```

você poderia esperar que o método `desenha( )` de `Forma` fosse chamado porque s foi declarado como uma referência para `Forma` e em tempo de compilação não há como se saber qual tipo ele assumirá durante a execução. No entanto, o mecanismo de amarração tardia torna esta chamada polimórfica, chamando apropriadamente o método `desenha()` de `Circulo`. O exemplo seguinte ilustra este comportamento.

```
// Formas.java
// Polimorfismo em Java

class Forma {
  void desenha() {}
  void apaga() {}
}
```

```
class Circulo extends Forma {
  void desenha() {
    System.out.println("Circulo.desenha()");
  }
  void apaga() {
    System.out.println("Circulo.apaga()");
  }
}

class Quadrado extends Forma {
  void desenha() {
    System.out.println("Quadrado.desenha()");
  }
  void apaga() {
    System.out.println("Quadrado.apaga()");
  }
}

class Triangulo extends Forma {
  void desenha() {
    System.out.println("Triangulo.desenha()");
  }
  void apaga() {
    System.out.println("Triangulo.apaga()");
  }
}

public class Formas {
  public static Forma randForma() {
    switch((int)(Math.random() * 3)) {
      default: // para que o compilador aceite
      case 0: return new Circulo();
      case 1: return new Quadrado();
      case 2: return new Triangulo();
    }
  }
  public static void main(String[] args) {
    Forma[] s = new Forma[9];
    // preenche o vetor com formas
    for(int i = 0; i < s.length; i++)
      s[i] = randForma();
    // chamadas polimorficas de metodos
    for(int i = 0; i < s.length; i++)
      s[i].desenha();
  }
} ///:~
```

Exemplo 6. 3 - Sobrescrita e Polimorfismo

A classe base `Forma` estabelece uma interface comum para qualquer uma de suas classes derivadas, ou seja, ela estabelece que todas as formas podem ser desenhadas e

apagadas. As classes derivadas sobrescrevem estas funções para fornecer o comportamento específico de cada tipo de forma.

A classe principal Formas contém um método estático `randForma ( )` que produz aleatoriamente uma referência a um objeto `Forma` a cada vez que é chamado. Note que o upcasting ocorre em cada comando *return,* o qual toma uma referência para um `Circulo`, `Quadrado` ou `Triangulo` e envia para fora do método uma referência para `Forma`. Assim, cada vez que este método é chamado não se pode saber qual o tipo que ele retornará de antemão.

O método main ( ) contém um vetor de referências para Forma que é preenchido através de chamadas a randForma ( ). Neste ponto, você sabe que o vetor tem formas, mas não tem como saber qualquer coisa mais específica do que isto (e nem tampouco o compilador). Contudo, quando se percorre o vetor chamando o método desenha para cada um de seus elementos, o comportamento corresponde magicamente ao do tipo específico, tal como pode ser visto pela saída do exemplo:

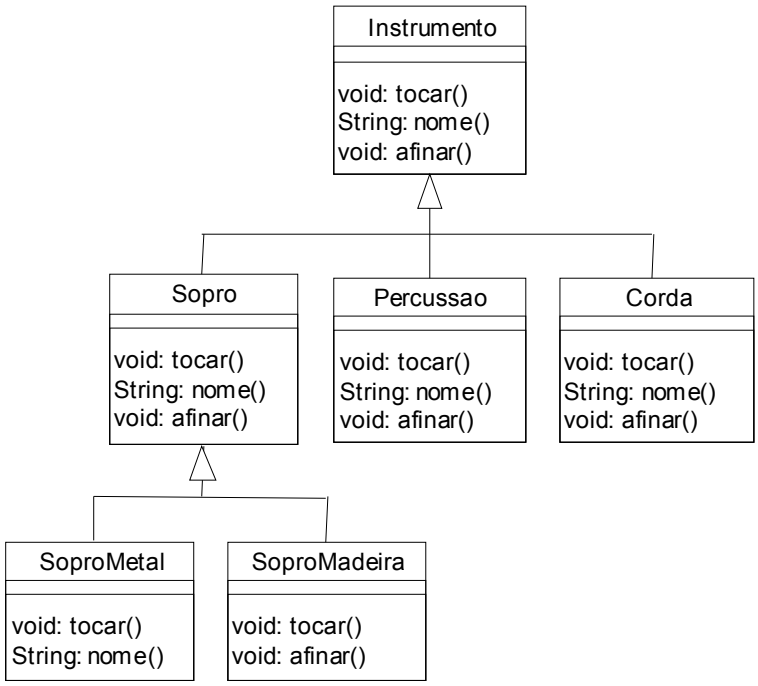
```
Circulo.desenha()
Triangulo.desenha()
Circulo.desenha()
Circulo.desenha()
Circulo.desenha()
Quadrado.desenha()
Triangulo.desenha()
Quadrado.desenha()
Quadrado.desenha()
```

Como as formas são geradas aleatoriamente a cada vez que o programa é executado, as diferentes execuções gerarão diferentes resultados. O papel da função aleatória do exemplo é mostrar que o compilador não tem como saber qual a função apropriada a ser chamada. Isto tem de ser feito através da amarração tardia.

6.5 – Extensibilidade

Agora vamos retornar ao exemplo dos instrumentos musicais. Graças ao polimorfismo, podemos adicionar ao sistema, tantos novos tipos de instrumento quanto quisermos sem precisar alterar o método `melodia()`. Em uma linguagem orientada a objetos bem projetada, a maioria dos métodos de um programa seguirá o modelo do método `melodia()`, ou seja, se comunicarão apenas pela interface comum da classe base. Isto torna qualquer hierarquia **extensível**, já que podemos adicionar novas funcionalidades ao programa apenas criando novos tipos de dados (classes) que herdem da classe base. Os métodos que manipulam a classe base terão sua interface mantida e apenas sobrescreverão algumas das funcionalidades da nova classe.

Considere os impactos sofridos pelo sistema ao adicionarmos mais métodos nas classes base e aumentarmos o número de subclasses. Aqui está o diagrama de classes:



Esta é a implementação das classes do diagrama anterior:

```
// Musica3.java
class Instrumento {
  public void tocar() {
    System.out.println("Instrumento.tocar()");
  }
  public String nome() {
    return "Instrumento";
  }
  public void afinar() {}
}

class Sopro extends Instrumento {
  public void tocar() {
     System.out.println("Sopro.tocar()");
  }
  public String nome() { return "Sopro"; }
  public void afinar() {}
}

class Percussao extends Instrumento {
  public void tocar() {
    System.out.println("Percussao.tocar()");
  }
  public String nome() { return "Percussao"; }
  public void afinar() {}
}
class Corda extends Instrumento {
  public void tocar() {
```

```
    System.out.println("Corda.tocar()");
  }
  public String nome() { return "Corda"; }
  public void afinar() {}
}

class SoproMetal extends Sopro {
  public void tocar() {
    System.out.println("SoproMetal.tocar()");
  }
  public void afinar() {
    System.out.println("SoproMetal.afinar()");
  }
}

class SoproMadeira extends Sopro {
  public void tocar() {
    System.out.println("SoproMadeira.tocar()");
  }
  public String nome() { return "SoproMadeira"; }
}

public class Musica3 {
// Tambem funcionara com os novos tipos adicionados!
  static void melodia(Instrumento i) {
    // ...
    i.tocar();
  }

  static void sinfonia(Instrumento[] e) {
    for(int i = 0; i < e.length; i++) melodia(e[i]);
  }

  public static void main(String[] args) {
    Instrumento[] orquestra = new Instrumento[5];
    int i = 0;
    // Upcasting:
    orquestra[i++] = new Sopro();
    orquestra[i++] = new Percussao();
    orquestra[i++] = new Corda();
    orquestra[i++] = new SoproMetal();
    orquestra[i++] = new SoproMadeira();
    sinfonia(orquestra);
  }
}
```

Exemplo 6. 4 - Um Programa Estendido

Os novos métodos adicionados foram `nome()`, que retorna uma `String` responsável pela descrição do instrumento, e `afinar()`, que provê alguma maneira de afinar o instrumento. No método `main()`, ao adicionarmos objetos ao vetor de instrumentos, realizamos upcast para `Instrumento`.

Como você pode ver, o método `melodia()` não foi afetado pelas alterações de código que ocorreram no programa e ainda funciona corretamente, inclusive com as novas classes adicionadas à hierarquia. E isto é exatamente o que o polimorfismo nos oferece.

6.6 – Sobrescrita x Sobrecarga

Vamos encarar de forma um pouco diferente o primeiro exemplo deste capítulo. No exemplo seguinte, a interface do método `tocar()` foi alterada no processo de sua redefinição, isso significa que o método não foi sobrescrito, e sim sobrecarregado. Como a sobrecarga de métodos é permitida em Java, o programa compilará normalmente, embora o resultado obtido com a sua execução pode não ser o esperado. Vejamos o exemplo seguinte:

```
// SoproErro.java
class Nota {
   public static final int
    DO = 0, RE = 1, MI = 2;
}

class Instrumento {
   public void tocar(int Nota) {
     System.out.println("Instrumento.tocar()");
   }
}

class Sopro extends Instrumento {
 // Mudanca na interface do metodo:
  public void tocar(Nota n) {
    System.out.println("Sopro.tocar(Nota n)");
  }
}

public class SoproErro {
  public static void melodia(Instrumento i) {
    // ...
    i.tocar(Nota.DO);
  }
  public static void main(String[] args) {
    Sopro flauta = new Sopro();
    melodia(flauta); // Nao eh como o esperado!
  }
}
```

Exemplo 6. 5 - Mudança Acidental na Interface

Na classe `Instrumento`, o método `tocar()` recebe um int como argumento, mas na classe `Sopro`, o método `tocar()` recebe como argumento uma referência a `Nota`. Aparentemente, o programador tinha como objetivo redefinir o método `tocar()`, da classe base `Instrumento`, mas acidentalmente, acabou sobrecarregando o método `tocar( )`, que passará a existir na classe `Sopro` com duas assinaturas. O compilador, entretanto, assume que o código fonte está correto e não emite nenhuma mensagem de

erro. Portanto, o método que será chamado no método `melodia( )` será o método `tocar( )` que recebe um inteiro como parâmetro, herdado da superclasse, e que não foi redefinido pela classe `Sopro`. A saída do programa é:

```
Instrumento.tocar()
```

Esta certamente não é uma chamada a um método polimórfico. E uma vez descoberto o ocorrido, o problema pode ser facilmente resolvido. Mas imagine o quão difícil pode ser encontrar este tipo de erro perdido em um programa.

6.7 – Classes e Métodos Abstratos

Em algumas circunstâncias desejamos orientar como uma classe deve ser implementada, ou melhor, como deve ser a interface de uma certa classe. Em outros casos, o modelo representado é tão amplo que certas classes tornam-se por demais gerais, não sendo possível ou razoável que possuam instâncias. Para estes casos dispomos das classes abstratas.

Tais classes são assim denominadas por não permitirem a instanciação, isto é, por não permitirem a criação de objetos do seu tipo. Sendo assim, seu uso é dirigido para a construção de classes modelo, ou seja, de especificações básicas de classes através do mecanismo de herança.

Uma classe abstrata deve ser estendida, ou seja, deve ser a classe base de outra, mais específica, que contenha os detalhes que não puderam ser incluídos na superclasse (abstrata). Outra possível aplicação das classes abstratas é a criação de um ancestral comum para um conjunto de classes que, se originados desta classe abstrata, poderão ser tratados genericamente através do polimorfismo.

Um classe abstrata, como qualquer outra, pode conter métodos, mas também pode adicionalmente conter métodos abstratos, isto é, métodos que não são implementados na classe abstrata, mas deverão ser implementados em suas subclasses não abstratas. Para fazer isso, Java permite que se use a palavra reservada *abstract* precedendo a definição do método abstrato, tal como:

```
abstract void X();
```

Uma classe que contenha pelo menos um método abstrato é também abstrata e, por conseguinte, deve ser declarada como abstrata através do uso da palavra *abstract* precedendo a sua definição (senão, o compilador não permitirá e emitirá uma mensagem de erro). Se alguém tentar crair um objeto de uma classe abstrata, o compilador também impedirá emitindo mensagem de erro. Deste modo, o compilador assegura a pureza da classe abstrata e você não necessita se preocupar sobre o mal uso dela.

Se você herdar de uma classe abstrata e quiser criar objetos desta nova classe, você deve implementar todos os métodos abstratos definidos na classe base. Se você não o fizer (e pode ser que seja isso que você deseje), então a classe derivada também será abstrata e o compilador o forçará a qualificá-la como abstrata.

É possível declarar uma classe como *abstract* sem incluir nenhum método abstrato. Isto é útil quando você cria uma classe onde não faz sentido ter métodos abstratos, mas ainda assim você quer prevenir qualquer objeto de ser criado a partir daquela classe. Segue-se a reimplementação do exemplo da orquestra, agora usando métodos e classes abstratas.

```
// Musica4.java
```

```
abstract class Instrumento {
  int i;
  abstract public void tocar();
  public String nome() {
    return "Instrumento";
  }
  abstract public void afinar();
}

class Sopro extends Instrumento {
  public void tocar() {
     System.out.println("Sopro.tocar()");
  }
  public String nome() { return "Sopro"; }
  public void afinar() {}
}

class Percussao extends Instrumento {
  public void tocar() {
    System.out.println("Percussao.tocar()");
  }
  public String nome() { return "Percussao"; }
  public void afinar() {}
}
class Corda extends Instrumento {
  public void tocar() {
    System.out.println("Corda.tocar()");
  }
  public String nome() { return "Corda"; }
  public void afinar() {}
}

class SoproMetal extends Sopro {
  public void tocar() {
    System.out.println("SoproMetal.tocar()");
  }
  public void afinar() {
    System.out.println("SoproMetal.afinar()");
  }
}

class SoproMadeira extends Sopro {
  public void tocar() {
    System.out.println("SoproMadeira.tocar()");
  }
  public String nome() { return "SoproMadeira"; }
}

public class Musica4 {
// Tambem funcionara com os novos tipos adicionados!
  static void melodia(Instrumento i) {
```

```
    // ...
    i.tocar();
  }

  static void sinfonia(Instrumento[] e) {
    for(int i = 0; i < e.length; i++) melodia(e[i]);
  }

  public static void main(String[] args) {
    Instrumento[] orquestra = new Instrumento[5];
    int i = 0;
    // Upcasting:
    orquestra[i++] = new Sopro();
    orquestra[i++] = new Percussao();
    orquestra[i++] = new Corda();
    orquestra[i++] = new SoproMetal();
    orquestra[i++] = new SoproMadeira();
    sinfonia(orquestra);
  }
}
```

Exemplo 6. 6 - Métodos e Classes Abstratas

Observe que realmente não há qualquer mudança no exemplo, a exceção da classe base.

Classes e métodos abstratos são úteis porque eles tornam explícita a natureza abstrata da classe, comunicando tanto ao usuário quanto ao compilador como você as projetou para serem usadas.

Como comentários finais sobre as classes abstratas, existem algumas restrições na declaração de seus métodos: (i) os métodos estáticos *(static*) não podem ser abstratos, (ii) construtores não podem ser abstratos e (iii) os métodos abstratos não podem ser privados.

6.8 – Inicialização com Amarração Tardia

A hierarquia de chamada de construtores juntamente com o mecanismo de amarração tardia traz um dilema interessante. O que acontece ao se chamar, de dentro de um construtor, um método amarrado dinamicamente do objeto sendo construído? No caso de um método que não seja o construtor é fácil saber a resposta. Isto tem de ser resolvido em tempo de execução porque o compilador não tem como saber se o objeto pertence a classe do método em execução ou se pertence a uma de suas classes derivadas. A mesma reposta é válida para o caso de métodos construtores. Contudo, a combinação de construtores com métodos amarrados dinamicamente pode gerar efeitos inesperados, que podem provocar erros difíceis de serem encontrados.

Conceitualmente, o papel do construtor é dar existência ao objeto. Ao criar um método construtor, contudo, o que podemos saber é que o objeto pode estar apenas parcialmente formado (você sabe apenas que os objetos da classe base foram inicializados, mas não pode saber quais classes são herdeiras de sua classe). Como métodos que usam amarração tardia podem chamar métodos de uma classe derivada, se você fizer isto dentro de um construtor, você pode acabar manipulando atributos membros que ainda não foram inicializados. Isto é uma receita certa para um disastre!!! Você pode ver este problema no exemplo seguinte:

```
// PoliConstrutor.java
// Construtores e polimorfismo
abstract class Simbolo {
  abstract void desenha();
  Simbolo() {
    System.out.println("Simbolo() antes");
    desenha();
    System.out.println("Simbolo() depois");
  }
}

class SimboloGrego extends Simbolo {
  int tamanho = 1;
 SimboloGrego (int r) {
    tamanho = r;
    System.out.println(
      " SimboloGrego.SimboloGrego(), tamanho = "
      + tamanho);
  }
  void desenha() {
    System.out.println(
      " SimboloGrego.desenha (),tamanho = " +
tamanho);
  }
}

public class PoliConstrutor {
  public static void main(String[] args) {
    new SimboloGrego(5);
  }
} ///:~
```

Exemplo 6. 7 - Construtor com Polimorfismo

Em `Simbolo`, o método `desenha( )` é abstrato, ou seja, foi projetado para ser sobrescrito. De fato, você é forçado o sobrescrevê-lo em `SimboloGrego`. Como o construtor `Simbolo( )` chama este método, o método `desenha( )` de `SimboloGrego` acaba sendo chamado, o que parecia ser o intento. Mas, veja qual é a saída do programa:

```
Simbolo() antes
SimboloGrego.desenha(), tamanho = 0
Simbolo() depois
SimboloGrego.SimboloGrego(), tamanho = 5
```

Quando o construtor de Simbolo chama desenha ( ), ao invés de um, o valor de tamanho contém o valor zero. Isto provavelmente resultaria em um ponto ou em nada sendo desenhado na tela, e você ficaria se interrogando por muito tempo porque isto estaria acontecendo.

Por conta disso, um bom conselho é: "Faça o mínimo possível para colocar o objeto em um bom estado e, se você pode evitar, não chame métodos de dentro dos construtores". Os únicos métodos seguros para serem chamados de dentro do construtor são aqueles

que são finais na classe base (isto também se aplica aos métodos privados, que são automaticamente finais). Estes métodos não podem ser sobrescritos e portanto não produzem este tipo de surpresa.

6.9 – A Classe Object

Agora já podemos observar melhor algumas características fornecidas pela classe `Object` (a mãe de todas as classes de Java). Se uma classe é declarada sem nenhuma cláusula *extends*, o compilador adiciona implicitamente um `extends Object` à declaração. Assim, a declaração

```
public class Empregado { … }
```

equivale a

```
public class Empregado extends Object { … }
```

Através desta característica, você pode herdar vários métodos da classe `Object`. Vamos discutir agora um importante método herdado de `Object`, chamado `equals`. O exemplo seguinte nos mostra como ele funciona:

```
// Equivalencia.java
class Valor {
  int i;
}
public class Equivalencia {
    public static void main(String[ ] args) {
      Integer n1 = new Integer(47);
      Integer n2 = new Integer(47);
      System.out.println(n1 == n2);
      System.out.println(n1 != n2);
      System.out.println(n1.equals (n2));
      Valor v1 = new Valor();
      Valor v2 = new Valor();
      v1.i = v2.i = 100;
      System.out.println(v1.equals(v2));
   }
}
```

Exemplo 6. 8 - Método equals

A saída desse programa será

```
false
true
true
false
```

O primeiro resultado é falso porque o operador == compara as referências dos objetos. Por razão equivalente, o segundo resultado é verdadeiro. O terceiro resultado é verdadeiro porque a classe `Integer` sobrescreve o método `equals` herdado de `Object` comparando assim o seu conteúdo. A razão pela qual o quarto resultado dá falso é porque a classe `Valor` não sobrescreve o método `equals`. Assim, ela herda o método de Object que implementa equals utilizando o operador ==.

6.10 – Exercícios

1. Crie uma hierarquia de classes a partir da classe MembroExercito, contendo subclasses para representar as diversas patentes dos membros do Exército (por exemplo, General, Coronel, Major, Tenente, Soldado) . Na classe base forneça métodos comuns a todos os membros do Exército e sobrescreva-os nas classes derivadas para realizar diferentes comportamentos dependendo do tipo específico de membro do Exército. Crie um vetor de membros do Exército, preencha-o com elementos das subclasses de Exército e chame os métodos da classe base para cada um dos elementos do vetor. Veja o que acontece.
2. Corrija o problema no programa do arquivo SoproErro.java.
3. Crie uma hierarquia de classes a partir da classe Móvel, contendo subclasses para representar diversos tipos de móveis (por exemplo, Cama, Mesa, Cadeira, Sofá, etc.) . Na classe base forneça métodos comuns a todos os móveis e sobrescreva-os nas classes derivadas para realizar diferentes comportamentos dependendo do tipo específico de móvel. Crie uma lista de móveis, preencha-a com elementos das subclasses de Móvel e chame os métodos da classe base para cada um dos elementos do vetor. Inclua um novo tipo de móvel no programa e verifique quais as alterações são necessárias.