Capítulo 5

Reuso de Classes

Uma das características mais importantes da orientação a objetos e, por consequência, da programação em Java é proporcionar facilidades para reutilização de código. Linguagens imperativas, tal como C, utilizam o mecanismo de cópia e modificação para que o código possa ser reusado. Contudo, este mecanismo não funciona muito bem. Além de ser um procedimento maçante para o programador, ele acaba introduzindo erros por conta de modificações feitas de maneira desatenta.

Como tudo em Java, a solução para o problema de reuso envolve o conceito de classes. Código é reusado através da criação de novas classes, mas ao invés da criação dessas novas classes a partir do zero, usa-se classes existentes que já foram criadas e depuradas, na maior parte das vezes, por outra pessoa.

A idéia é usar classes sem que se necessite alterar o código existente. Neste capítulo, veremos duas maneiras de alcançar este objetivo. O primeiro mecanismo é bastante direto e consiste simplesmente do uso de objetos das classes já existentes dentro da nova classe. Isto se chama *composição* (ou agregação) porque a nova classe é composta de objetos das classes existentes. Neste caso, nós reusamos a funcionalidade do código atuando como clientes das classes pré-existentes (elas são usadas como uma caixa fechada).

A outra forma de reuso é mais sutil. Ela consiste na criação de novas classes como sendo uma extensão de uma classe existente. Neste caso, a funcionalidade do código é reutilizada através da incorporação automática do código da classe pré-existente na nova classe. Para caracterizar a nova classe, apenas se adiciona o código que torna esta nova classe diferente da pré-existente, sem que seja necessário modificar a última. Quando utilizamos este tipo de reuso, estamos atuando como implementadores de classes, pois em boa parte das vezes é necessário conhecer a estrutura interna da classe pré-existente para adicionar propriamente o novo código. Este tipo de reuso é conhecido como *herança* (ou derivação). Herança é uma das características mais distintivas da programação orientada a objetos e tem implicações adicionais que serão estudadas no capítulo seguinte.

Boa parte da sintaxe e da semântica dos mecanismos de composição e herança são similares (o que faz sentido porque ambos são mecanismos para a confecção de novos tipos a partir de tipos pré-existentes). Neste capítulo, você aprenderá mais sobre estes mecanismos de reuso de código.

5.1– Mecanismo de Composição

Embora não tenhamos utilizado este nome, o mecanismo de composição já vem sendo usado neste texto de forma regular. Nós simplesmente colocamos referências a objetos dentro das novas classes. Por exemplo, suponha que quiséssemos criar um objeto que armazenasse diversos objetos da classe String, um par de variáveis primitivas e um objeto de outra classe. Enquanto nós teríamos de colocar referências para os objetos não primitivos dentro da nossa nova classe, as variáveis primitivas teriam de ser definidas dentro dela. O exemplo abaixo mostra um exemplo de composição:

```
class FonteDagua {
  private String s;
```

```
  FonteDagua () {
    System.out.println("FonteDagua ()");
    s = new String("Construida");
  }
  public String toString() { return s; }
}

public class Engarrafador {
  private String garrafa1, garrafa2, garrafa3,
garrafa4;
  FonteDagua fonte;
  int i;
  float f;
  void imprime() {
    System.out.println("garrafa1 = " + garrafa1);
    System.out.println("garrafa2 = " + garrafa2);
    System.out.println("garrafa3 = " + garrafa3);
    System.out.println("garrafa4 = " + garrafa4);
    System.out.println("i = " + i);
    System.out.println("f = " + f);
    System.out.println("fonte = " + fonte);
  }
  public static void main(String[] args) {
     Engarrafador x = new Engarrafador ();
       x.imprime ();
  }
}
```

Exemplo 5. 1 - Exemplo de Composição

Um dos métodos definidos em FonteDagua é especial: `toString( ).` Você aprenderá mais tarde que todo objeto não primitivo tem um método `toString( ),` e este é chamado em situações especiais quando o compilador quer uma String, mas obtém um destes objetos. Desta forma, na expressão

```
System.out.println("fonte = " + fonte);
```

o compilador assume que você está tentando adicionar um objeto `String` (`"fonte = "`) a uma `FonteDagua`. Isto não faz sentido, uma vez que somente se pode adicionar uma `String` a outra `String`. Dessa forma, o compilador transformará uma `FonteDagua` em uma `String` chamando o método `toString( )` de `FonteDagua`. Depois que isto tenha sido feito as duas strings podem ser combinadas e a string resultante pode ser passada para `System.out.println( )`. Sempre que você desejar este comportamento em uma classe, tudo o que você necessita fazer e criar um método um método `toString( ).`

A primeira vista, você poderia assumir que o compilador construiu objetos, automaticamente, para cada uma das referências no código acima, por exemplo, chamando o construtor default para `FonteDagua` para inicializar `fonte`. Contudo, a saída do método `imprime()` é:

```
garrafa1 = null
garrafa2 = null
garrafa3 = null
```

```
garrafa4 = null
i = 0
f = 0.0
fonte = null
```

No exemplo, os primitivos são automaticamente inicializados para zero, como já foi dito anteriormente. Mas as referências para os objetos são inicializadas para null. Se você tentar chamar um método para qualquer uma destas referências nulas será obtida uma exceção. O fato de você ainda podê-las imprimir sem que isto dispare uma exceção é realmente bastante útil.

A razão pela qual o compilador não cria um objeto default para toda referência é que isto provocaria uma sobrecarga desnecessária sobre a eficiência do código, em muitos casos. Se você realmente quer inicializar as referências, você deve fazer isto:

1. no ponto onde os objetos são definidos (isto significa que eles sempre serão inicializados antes que o construtor seja chamado);
2. no construtor daquela classe;
3. exatamente antes do ponto no qual você precisa do objeto. Isto pode reduzir a sobrecarga sobre a eficiência do código, se existirem trechos do programa onde o objeto não necessita ser criado.

Todas as três abordagens são mostradas a seguir:

```
// Banho.java
// Inicializacao do construtor com composicao

class Sabao {
  private String s;
  Sabao() {
    System.out.println("Sabao()");
    // Inicializacao no construtor
    s = new String("Construido");
  }
  public String toString() { return s; }
}

public class Banho {
  private String
    // Inicializacao no ponto da definicao:
    s1 = new String("Feliz"),
    s2 = "Feliz",
    s3, s4;
  Sabao omo;
  int i;
  float brinquedo;
  Banho() {
    System.out.println("Dentro Banho()");
    s3 = new String("Maria");
    i = 47;
    brinquedo = 3.14f;
    omo = new Sabao();
  }
  void print() {
    // Inicializacao tardia:
```

```
    if(s4 == null) s4 = new String("Maria");
    System.out.println("s1 = " + s1);
    System.out.println("s2 = " + s2);
    System.out.println("s3 = " + s3);
    System.out.println("s4 = " + s4);
    System.out.println("i = " + i);
    System.out.println("brinquedo = " + brinquedo);
    System.out.println("omo = " + omo);
  }
  public static void main(String[] args) {
    Banho b = new Banho();
    b.print();
  }
}
```

Exemplo 5. 2- Exemplo de Composição

Note que no construtor da classe `Banho` uma diretiva é executada antes de serem feitas as inicializações de `s3` e `omo`. Quando você não inicializa o objeto no ponto de sua definição, não há nenhuma garantia que a inicialização do objeto será executada antes que uma mensagem seja enviada para a referência. Se for este o caso, ocorrerá uma exceção. Aqui esta a saída do programa:

```
Dentro Banho()
Sabao()
s1 = Feliz
s2 = Feliz
s3 = Maria
s4 = Maria
i = 47
brinquedo = 3.14
omo = Construido
```

Quando `imprime( )` é chamado, ele preenche `s4` de modo a fazer com que todos os campos estejam apropriadamente inicializados no momento em que serão utilizados.

5.2 – Herança

Herança é uma parte integrante de Java (e de qualquer linguagem orientada a objetos). De fato, em Java, quando você cria uma classe, você estará sempre fazendo herança porque todas as classes são subclasses de uma classe raiz padrão: a classe `Object`. Assim, ao se criar uma classe você está, implicitamente, herdando os atributos e métodos da classe `Object`.

Além de se criar classes herdeiras da classe `Object`, pode-se desejar criar classes que sejam herdeiras de outra classe qualquer (normalmente chamada de classe base). Neste caso, deve-se indicar explicitamente, na declaração da classe, de quem ela é herdeira imediata. Você faz isto no código usando, na declaração da classe, a palavra chave *extends* seguida do nome da classe base. Ao herdar de uma classe base, a nova classe adquire automaticamente todos os atributos e métodos da classe base.

```
// Detergente.java
// Propriedades da Heranca

class Limpador {
```

```
  private String s = new String("Limpador");
  public void anexar(String a) { s += a; }
  public void diluir() { anexar(" diluir()"); }
  public void aplicar() { anexar(" aplicar()"); }
  public void esfregar() { anexar(" esfregar()"); }
  public void imprimir() { System.out.println(s); }
  public static void main(String[] args) {
    Limpador x = new Limpador();
    x.diluir(); x.aplicar(); x.esfregar();
    x.imprimir();
  }
}

public class Detergente extends Limpador {
  // Muda um metodo:
  public void esfregar() {
    anexar(" Detergente.esfregar()");
    super.esfregar(); // Chama a versao da classe
base
  }
  // Adiciona metodos a interface:
  public void espumar() { anexar(" espumar()"); }
  // Testa a nova classe:
  public static void main(String[] args) {
    Detergente x = new Detergente();
    x.diluir();
    x.aplicar();
    x.esfregar();
    x.espumar();
    x.imprimir();
    System.out.println("Testando classe base:");
    Limpador.main(args);
  }
}
```

Exemplo 5. 3- Exemplo de Herança

O exemplo demonstra um certo número de características. Primeiro, no método `anexar()` da classe `Limpador`, uma string é concatenada a `s` utilizando o operador +=. Este é um dos operadores que os projetistas de Java sobrecarregaram para trabalhar com strings.

Segundo, tanto a classe `Limpador` quanto `Detergente` possuem um método `main( )`. Isto freqüentemente é recomendado porque o código de teste de sua classe se torna parte integrante dela. Mesmo se você tem muitas classes em um programa, somente o método `main( )` da classe invocada será chamado na linha de comando. Neste caso específico, quando você diz `java Detergente`, `Detergente.main( )` será chamado. Mas você pode também dizer `java Limpador` para invocar `Limpador.main( )`. Esta técnica de colocar um método `main( )` em cada classe permite a realização de testes específicos para cada classe. E você não necessita remover o `main( )` quando finaliza o teste; você pode deixá-lo lá para um teste posterior.

Aqui, você pode ver que `Detergente.main( )` chama `Limpador.main( )` explicitamente, passando para o último os mesmos argumentos da linha de comando (você também poderia passar um outro vetor de strings).

É importante notar que todos os métodos de `Limpador` são públicos. Lembre-se que se você não usa um especificador de acesso, o membro é considerado amigo ("friendly"), o que concede acesso livre apenas aos membros do mesmo pacote. Dessa forma, a classe `Detergente` não teria problema. No entanto, se uma classe de algum outro pacote fosse herdar de `Limpador`, ela poderia acessar somente os membros públicos. Como uma regra geral, faça todos os campos privados e todos os métodos públicos (membros protegidos também concedem acesso as classes derivadas; você aprenderá mais a respeito disto oportunamente). É claro que, em casos particulares, você deverá fazer ajustes, mas a regra acima é um guia para a maioria das situações.

Note que `Limpador` tem um conjunto de métodos em sua interface: `anexar( ), diluir( ), aplicar( ), esfregar( ), e imprimir( )`. Como a classe `Detergente` é derivada da classe `Limpador` (pelo uso da palavra chave *extends*), ela obtém gratuitamente todos os métodos da interface da última. Assim, você pode pensar em herança como sendo o reuso da interface da classe base (embora a implementação também seja herdada, este não é o ponto principal).

Como pode ser visto no método `esfregar( )`, é possível tomar um método definido na classe base e modificá-lo. Este tipo de comportamento se chama **sobrescrever** o método na terminologia de orientação a objetos. Observe que o protótipo da função `esfregar ( )` é o mesmo para a classe base e para a classe derivada. Esta é uma característica da **sobrescrita** que a diferencia da sobrecarga.

Quando se faz sobrescrita, você pode desejar chamar o método da classe base dentro da nova versão. Mas dentro de `esfregar( )` você não pode simplesmente chamar `esfregar( )`, uma vez que isto produziria uma chamada recursiva, que não é o que você deseja. Para resolver este problema Java tem a palavra chave *super* a qual se refere a superclasse da classe corrente. Dessa forma a expressão `super.esfregar( )` chama a versão da classe base do método `esfregar( )`. É importante ressaltar que a palavra *super* também serve para se referir aos atributos da superclasse.

Ao herdar de uma classe, você não fica restrito a utilizar os métodos da classe base. Você também pode adicionar novos métodos às classes derivadas exatamente do mesmo modo que você faz para incluir qualquer novo método em uma classe, isto é, apenas o defina. O método `espumar( )` é um exemplo de inclusão de um novo método.

Em `Detergente.main( )` você pode ver que um objeto `Detergente` pode chamar todos os métodos que estão definidos na classe `Limpador`, bem como aqueles definidos na própria classe `Detergente` (por exemplo, `espumar( )`).

5.2.1 – Inicializando a Classe Base

Uma vez que na herança existem duas classes envolvidas - a classe base e a classe derivada - ao invés de apenas uma, pode ser um pouco confuso imaginar o objeto resultante produzido por uma classe derivada. De um ponto de vista externo parece que a nova classe tem a mesma interface que a classe base e talvez alguns atributos e métodos adicionais. Mas herança não é apenas a cópia da interface da classe base. Quando você cria um objeto da classe derivada, ele contém um objeto da classe base. Este objeto é idêntico a um objeto criado diretamente a partir da própria classe base.

Para um uso adequado do objeto da classe base, é essencial que ele seja corretamente inicializado. Existe somente uma maneira de garantir isto: executando a inicialização no seu construtor, através da chamada do construtor da classe base, o qual tem o conhecimento apropriado e privilégios para executar a inicialização dos objetos da classe base. Java insere implicitamente o construtor da classe base no construtor da classe derivada.

```
// Cartao.java
// Construtor chamado durante heranca

class Arte {
  Arte() {
    System.out.println("Construtor Arte");
  }
}

class Desenho extends Arte {
  Desenho() {
    System.out.println("Construtor Desenho");
  }
}

public class Cartao extends Desenho {
  Cartao() {
    System.out.println("Construtor Cartao");
  }
  public static void main(String[] args) {
    Cartao x = new Cartao();
  }
}
```

Exemplo 5. 4 - Inicialização com Herança

A saída deste programa mostra as chamadas automáticas:

```
Construtor Arte
Construtor Desenho
Construtor Cartao
```

Você pode ver que a construção ocorre a partir da classe base, de modo que os atributos da classe base são inicializados antes que os construtores da classe derivada possam acessar o objeto. Mesmo se você não cria um construtor para a classe `Cartao`, o compilador criará um construtor default para você, o qual chamará o construtor da classe base.

5.2.2 – Construtores com Argumentos

O exemplo acima utiliza construtores default, ou seja, eles não tem nenhum argumento. Desta forma, é fácil para o compilador chamá-los, uma vez que não há necessidade de passar argumentos para o construtor. Contudo, se sua classe não tem o construtor default, ou você quer chamar o construtor da classe que possui um argumento, você deve explicitamente escrever as chamadas aos construtores da classe base utilizando a palavra reservada *super* e a lista de argumentos apropriada:

```
// Xadrez.java
// Heranca, construtores e argumentos.
```

```
class Jogo {
  Jogo(int i) {
    System.out.println("Construtor Jogo");
  }
}

class JogoTabuleiro extends Jogo{
  JogoTabuleiro(int i) {
    super(i);
    System.out.println("Construtor JogoTabuleiro");
  }
}

public class Xadrez extends JogoTabuleiro {
  Xadrez() {
    super(11);
    System.out.println("Construtor Xadrez");
  }
  public static void main(String[] args) {
    Xadrez x = new Xadrez();
  }
}
```

Exemplo 5. 5 - Construtor com Argumentos

Se você não chamar o construtor da classe base `JogoTabuleiro` no construtor da classe `Xadrez`, o compilador reclamará que não pode encontrar um construtor da forma `JogoTabuleiro( )`. Adicionalmente, a chamada ao construtor da classe base deve ser a primeira coisa que você faz no construtor da classe derivada para que não haja erro de compilação.

5.2.3 – Combinando Composição e Herança

É muito comum utilizar composição e herança juntos. O exemplo mostrado a seguir mostra como uma classe complexa pode ser criada utilizando-se herança e composição:

```
// PreparacaoMesa.java
// Combinando composicao e heranca.

class Prato
  Prato (int i) {
    System.out.println("Construtor Prato");
  }
}

class PratoJantar extends Prato {
  PratoJantar(int i) {
    super(i);
    System.out.println(
      "Construtor PratoJantar");
  }
}
```

```
class Utensilio {
  Utensilio(int i) {
    System.out.println("Construtor Utensilio");
  }
}

class Colher extends Utensilio {
  Colher(int i) {
    super(i);
    System.out.println("Construtor Colher");
  }
}

class Garfo extends Utensilio {
  Garfo(int i) {
    super(i);
    System.out.println("Construtor Garfo");
  }
}

class Faca extends Utensilio {
  Faca(int i) {
    super(i);
    System.out.println("Construtor Faca");
  }
}

// Uma maneira rotineira de fazer alguma coisa:
class Costume{
  Costume(int i) {
    System.out.println("Construtor Costume");
  }
}

public class PreparacaoMesa extends Costume {
  Sopa sp;
  Garfo frk;
  Faca kn;
  PratoJantar pl;
  PreparacaoMesa(int i) {
    super(i + 1);
    sp = new Colher(i + 2);
    frk = new Garfo(i + 3);
    kn = new Faca(i + 4);
    pl = new PratoJantar(i + 5);
    System.out.println(
      "Construtor PreparacaoMesa");
  }
  public static void main(String[] args) {
    PreparacaoMesa x = new PreparacaoMesa(9);
  }
}
```

Exemplo 5. 6 - Composição e Herança

Enquanto o compilador força a inicialização do objeto herdado da classe base, requerendo que você faca isto no começo do construtor, ele não controla a inicialização dos objetos membros. Deste modo, você deve atentar para a inicialização dos objetos membros porque esta tarefa é de sua inteira responsabilidade.

5.2.4 – Restauração

Java não tem o conceito de destrutor (um método que é chamado automaticamente quando um objeto é destruído). A razão é que a prática em Java é simplesmente esquecer o objeto, permitindo que o coletor de lixo requisite a memória quando for necessário.

Na maioria das vezes esta prática é adequada. Contudo, algumas vezes uma classe pode executar algumas atividades durante seu tempo de vida que demandam restauração ou limpeza. Como não se pode saber com certeza quando o coletor de lixo será chamado, você deve escrever explicitamente um método especial para realizar tal restauração. Adicionalmente, você deve garantir que o programador cliente fique sabendo que ele deve chamar explicitamente este método.

```
// SistemaCAD.java
// Assegurando limpeza apropriada.

class Forma {
  Forma(int i) {
    System.out.println("Construtor Forma");
  }
  void limpeza() {
    System.out.println("Limpeza Forma");
  }
}

class Circulo extends Forma {
  Circulo(int i) {
    super(i);
    System.out.println("Desenhando Circulo");
  }
  void limpeza() {
    System.out.println("Apagando um Circulo");
    super.limpeza();
  }
}

class Triangulo extends Forma {
  Triangulo(int i) {
    super(i);
    System.out.println("Desenhando um Triangulo");
  }
  void limpeza() {
    System.out.println("Apagando um Triangulo");
    super.limpeza();
  }
}
```

```
class Linha extends Forma {
  private int comeco, fim;
  Linha(int comeco, int fim) {
    super(comeco);
    this.comeco = comeco;
    this.fim = fim;
    System.out.println("Desenhando uma Linha: " +
           comeco + ", " + fim);
  }
  void limpeza() {
    System.out.println("Apagando uma Linha: " +
           comeco + ", " + fim);
    super.limpeza();
  }
}

public class SistemaCAD extends Forma {
  private Circulo c;
  private Triangulo t;
  private Linha[] linhas = new Linha[10];
  SistemaCAD (int i) {
    super(i + 1);
    for(int j = 0; j < 10; j++)
      linhas[j] = new Linha(j, j*j);
    c = new Circulo(1);
    t = new Triangulo(1);
    System.out.println("Construtor Combinado");
  }
  void limpeza() {
    System.out.println("SistemaCAD.limpeza()");
    // A ordem de remocao e o reverso
    // da ordem de inicializacao
    t.limpeza();
    c.limpeza();
    for(int i = linhas.length - 1; i >= 0; i--)
      linhas[i].limpeza();
    super.limpeza();
  }
  public static void main(String[] args) {
    SistemaCAD x = new SistemaCAD (47);
    // …
    x.limpeza();
  }
}
```

Exemplo 5. 7 - Restauração

Tudo neste sistema é alguma espécie de `Forma` (que por sua vez é uma espécie de `Object`, uma vez que herda implicitamente desta classe). Cada classe redefine o método `limpeza( )` da classe `Forma`, além de chamar a versão de `limpeza( )` da classe base utilizando a palavra chave *super*. As classes `Circulo`, `Triangulo` e `Linha` têm construtores que "desenham", embora qualquer método chamado durante o

tempo de vida do objeto pudesse ser responsável por fazer alguma coisa que necessitasse de restauração. Cada classe tem seu próprio método de limpeza para restaurar coisas não relacionadas com o armazenamento de memória para o estado em que se encontravam antes do objeto existir.

Note que, ao escrever o método de restauração de uma classe, você deve se preocupar com a ordem de chamada dos métodos de restauração dos objetos membros e do objeto herdado da classe base. De uma forma geral, você deve seguir a seguinte ordem: primeiro execute todo o trabalho de restauração específico para sua classe (o que pode requerer que os elementos da classe base ainda estejam disponíveis), na ordem inversa da criação. Então chame o método de restauração da classe base, como foi demonstrado no exemplo. Na grande maioria dos casos, a restauração não é um problema. Você deve apenas esperar pelo coletor de lixo. No entanto, atenção e cautela são recomendadas quando você deve fazê-la explicitamente.

Normalmente não se deve confiar na operação do coletor de lixo para se realizar qualquer operação, a exceção da liberação de memória. Isto decorre do fato de poder existir situações onde o coletor nunca será invocado. E mesmo nas situações nais quais o coletor de lixo é chamado, não existe garantia que ele removerá os objetos em uma ordem bem definida.

5.2.5 – Composição x Herança

Tanto composição como herança permitem que você coloque objetos dentro de uma nova classe. Você pode se questionar a respeito da diferença entre os dois e sobre quando usar um ou outro.

Composição é geralmente utilizada quando você quer as características de uma classe existente dentro de uma nova classe, mas não sua interface. Isto é, você insere um objeto de modo a utilizá-lo para implementar a funcionalidade de sua nova classe, mas o usuário acessa sua nova classe somente através da interface que você definiu, não usando a interface do objeto inserido. Para alcançar este efeito você inclui os objetos de classes existentes como privados em sua nova classe.

Algumas vezes faz sentido permitir que o usuário da classe acesse diretamente a composição existente em sua nova classe. Para isto, você torna públicos os membros objeto.

```
// Automovel.java
// Composicao com objetos publicos.

class Motor {
  public void ligar() {}
  public void parar() {}
}

class Pneu {
  public void encher (int psi) {}
}

class Janela {
  public void fechar() {}
  public void abrir() {}
}
```

```
class Porta {
  public Janela janela = new Janela();
  public void abrir() {}
  public void fechar() {}
}

public class Automovel {
  public Motor motor = new Motor();
  public Pneu[] pneus = new Pneu[4];
  public Porta esquerda = new Porta(),
       direita = new Porta();          // 2-portas
  public Automovel() {
    for(int i = 0; i < 4; i++)
      pneus[i] = new Pneu();
  }
  public static void main(String[] args) {
    Automovel carro = new Automovel();
    carro.esquerda.janela.fechar();
    carro.pneus[0].encher(72);
  }
}
```

Exemplo 5. 8 - Atributos Públicos

Ao tornar públicos os atributos membros das classes, você está reduzindo a complexidade do trabalho do implementador da classe. Isso pode ser útil em algumas situações. Contudo, com esta postura, você muitas vezes sacrifica o reuso do código usuário quando ocorrem modificações na classe fornecedora. Assim, você deve ter em mente que a declaração de atributos como públicos é um caso especial e que em geral você deve fazer os campos *private*.

Quando você herda, você parte de uma classe existente e cria uma nova versão desta classe. Em geral, isto significa que você está tomando uma classe de propósito mais geral e especializando-a para uma necessidade particular. Observe que não faria sentido compor um automovel a partir de um objeto veículo pois um automovel não contém um veiculo. De fato, um automovel é um veículo. Como regra geral, você pode usar a seguinte regra: um relacionamento entre objetos do tipo **é-um** é expresso como herança e um relacionamento do tipo **tem-um** é expresso como composição.

5.2.6 – Membros Protegidos

Agora que você foi apresentado ao conceito de herança, a palavra chave *protected* finalmente ganha significado. Em projetos reais, algumas vezes, você quer deixar algo oculto dos clientes da classe, mas ainda conceder acesso livre aos implementadores de classes derivadas. A palavra chave *protected* faz exatamente isto. Ela diz "Isto é privado, contudo está disponível para alguém que herde desta classe ou pertença ao mesmo pacote". Assim, além de liberar acesso para subclasses, a palavra *protected* se comporta automaticamente como amiga ("friendly").

A melhor estratégia é deixar os atributos membros *private* de modo a sempre preservar seu direito de mudar a implementação da classe sem implicar na alteração do código usuário. Você pode, então, permitir acesso controlado as herdeiros de sua classe por meio de métodos protegidos:

```
// Maluca.java
```

```
// A palavra chave protected
import java.util.*;

class Aloprada {
  private int i;
  protected int ler() { return i; }
  protected void ajustar (int ii) { i = ii; }
  public Aloprada (int ii) { i = ii; }
  public int valor (int m) { return m*i; }
}

public class Maluca extends Aloprada {
  private int j;
  public Maluca (int jj) { super(jj); j = jj; }
  public void mudar(int x) { ajustar(x); }
}
```

Exemplo 5. 9 - Especificador de Acesso Protegido

Você pode ver que `mudar( )` de `Maluca` tem acesso a `ajustar( )` de `Aloprada` porque este método é protegido.

5.2.7 – Vantagens da Herança

Uma das vantagens da herança é suportar o desenvolvimento incremental, permitindo que você crie um novo código sem introduzir erros no código pré-existente. Herdando de uma classe já existente que esteja funcionando e adicionando atributos e métodos membros (e, eventualmente, também redefinindo os métodos existentes), você deixa o código existente já depurado intocado, de modo que um outro usuário ainda possa utilizá-lo sem maiores problemas. Se um erro ocorre, você sabe que este ocorreu no novo código.

Outra vantagem da herança é decorrente do relacionamento **é-um** expresso entre a nova classe e a classe base. Isto significa que um objeto da classe derivada pode ser usado onde se espera um objeto da classe base. Esta característica possibilita que se crie código cliente que se aplica tanto para objetos da classe base quanto da classe derivada. Como exemplo, considere uma classe chamada `Instrumento` representando instrumentos musicais, e uma classe derivada chamada `Sopro`. Uma vez que herança significa que todos os métodos na classe base estão também disponíveis na classe derivada. Se a classe `Instrumento` tem um método `tocar( )`, a classe `Sopro` também terá. Isto significa que nós podemos dizer que um objeto da classe `Sopro` é também um tipo de `Instrumento`.

```
// Sopro.java
// Relacionamento eUm

class Instrumento{
  public void tocar() {}
  static void afinar(Instrument i) {
    // ...
    i.tocar();
  }
}
```

```
// Objetos Sopro sao instrumentos
// pois tem a mesma interface
public class Sopro extends Instrumento{
  public static void main(String[] args) {
    Sopro flauta = new Sopro();
    Instrumento.afinar(flauta); // Upcasting
  }
}
```

Exemplo 5. 10 - Relacionamento é-um na Herança

O interessante neste exemplo é o fato do método `afinar( )`, que aceita uma referência para um `Instrumento`, estar sendo chamado com uma referência para um objeto `Sopro`. Uma vez que Java dá atenção especial a checagem de tipo parece estranho que um método possa aceitar um objeto de outro tipo. Contudo, um objeto da classe `Sopro` é também um objeto `Instrumento`, e não existe nenhum método `afinar` que possa ser chamado para um `Instrumento` que não possa também ser chamado para um objeto da classe `Sopro`. Dentro de `afinar( )`, o código é implementado para trabalhar com `Instrumento` ou qualquer coisa derivada de `Instrumento` (visto tudo que é de `Instrumento` também existirá na classe derivada). No exemplo, o código de `afinar( )` atua convertendo uma referência para `Sopro` em uma referência para `Instrumento`. Esta conversão é chamada de **"upcasting"**.

5.2.8 – Dados, Métodos e Classes Finais

A palavra reservada *final* tem significados ligeiramente diferentes, dependendo do contexto em que são usadas. Mas, em geral, ela diz "Isto não pode ser mudado". Você pode desejar prevenir mudanças por duas razões principais: por eficiência ou para conseguir certas propriedades de projeto. As seções seguintes discutem os três lugares onde **final** pode ser utilizado: para dados (atributos ou locais), métodos e classes.

5.2.8.1 – Dados Finais

Muitas linguagens de programação possuem uma maneira de dizer ao compilador que um pedaço de dado é constante. Uma constante é útil em dois contextos:

1. Quando um valor conhecido em tempo de compilação nunca será mudado.
2. Quando um valor inicializado em tempo de execução nunca será mudado.

No caso de uma constante definida em tempo de compilação, o compilador pode substituir o valor constante em qualquer cálculo em que a constante é utilizada. Eventualmente, certos cálculos podem ser feitos em tempo de compilação mesmo, poupando assim tempo de execução. Em Java, este tipo de constante deve ser de tipo primitivo e deve ser expresso utilizando a palavra reservada *final*. Um valor deve ser fornecido para a constante no momento de sua definição. Um atributo estático e final ocupa um único espaço de memória que não pode ser mudado.

Quando a palavra *final* é utilizada sobre referências a objetos, ao invés de tipos primitivos, o significado é um pouco diferente. Uma vez que uma referência é inicializada para um objeto, esta referência não pode ser mudada de modo a apontar para um outro objeto. No entanto, o próprio objeto pode ser modificado; Java não fornece nenhuma forma de fazer um objeto específico constante.

```
// DadoFinal.java
```

```
// Efeito de final

class Valor {
  int i = 1;
}

public class DadoFinal {
  // Podem ser constantes em tempo de compilacao
  final int i1 = 9;
  static final int CONST_2 = 99;
  // Constante publica comum:
  public static final int CONST_3 = 39;
  // Nao podem ser constantes em tempo de compilacao:
  final int i4 = (int)(Math.random()*20);
  static final int i5 = (int)(Math.random()*20);

  Valor v1 = new Valor();
  final Valor v2 = new Valor();
  static final Valor v3 = new Valor();
  // Arrays:
  final int[] a = { 1, 2, 3, 4, 5, 6 };

  public void print(String id) {
    System.out.println(
      id + ": " + "i4 = " + i4 +
      ", i5 = " + i5);
  }
  public static void main(String[] args) {
    DadoFinal fd1 = new DadoFinal();
    //! fd1.i1++; // Erro: nao pode mudar o valor
    fd1.v2.i++; // Objeto nao eh constante!
    fd1.v1 = new Valor(); // OK -- nao final
    for(int i = 0; i < fd1.a.length; i++)
      fd1.a[i]++; // Objeto nao e constante!
    //! fd1.v2 = new Valor(); // Erro: Nao
    //! fd1.v3 = new Valor(); // pode mudar
referencia
    //! fd1.a = new int[3];

    fd1.print("fd1");
    System.out.println("Criando novo DadoFinal");
    DadoFinal fd2 = new DadoFinal();
    fd1.print("fd1");
    fd2.print("fd2");
  }
}
```

Exemplo 5. 11 - Dados Finais

Este exemplo também demonstra a diferença entre fazer um valor *final static* ou não. A diferença é mostrada na saída do programa:

```
fd1: i4 = 15, i5 = 9
Criando novo DadoFinal
```

```
fd1: i4 = 15, i5 = 9
fd2: i4 = 10, i5 = 9
```

Note que os valores de `i4` para `fd1` e `fd2` são únicos, mas o valor de `i5` não é mudado pela criação do segundo objeto `DadoFinal`. Uma vez que é *static*, `i5` é inicializado uma única vez e não toda vez que um objeto desta classe é criado.

5.2.8.1.1 – Dados Finais não Inicializados na Definição

Java permite a criação de variáveis locais e atributos declarados como *final*, mas que não recebem um valor na sua definição. Neste caso, o compilador exige que seja atribuído um valor à variável ou atributo não inicializado antes que ele seja utilizado. Este mecanismo torna o uso da palavra *final* muito mais flexível, uma vez que permite, por exemplo, a um atributo *final* ser diferente para cada objeto de uma classe e ainda assim imutável.

```
// DadoFinalLivre.java
// Dados finais nao inicializados

class Viagem { }

class DadoFinalLivre {
  final int i = 0; // Final inicializado
  final int j; // Final nao inicializado
  final Viagem p; // Referencia final nao
inicializada
  // Finais nao inicializados DEVEM ser inicializados
  // em todos os construtores e somente neles
  DadoFinalLivre () {
    j = 1; // Inicializa Final
    p = new Viagem();
  }
  DadoFinalLivre (int x) {
    j = x; // Inicializa Final
    p = new Viagem();
  }
  public static void main(String[] args) {
    DadoFinalLivre bf = new DadoFinalLivre();
  }
}
```

Exemplo 5. 12 - Dado Final Não Inicializado

Você é forçado a inicializar o atributo ou variável local *final* por meio de uma expressão no ponto de definição do *final* ou em todo construtor da classe. Desta forma, se garante que o *final* é sempre inicializado antes que seja utilizado.

5.2.8.1.2 – Argumentos Finais

Java permite que você use a palavra *final* junto com os argumentos na lista de argumentos de um método. No caso de argumentos primitivos, isto significa que o método não pode mudar o valor do argumento. No caso de objetos, significa que a referência apontada pelo argumento não pode ser mudada de modo a apontar para outro objeto:

```
// Utilizando Final com argumentos.
```

```
class Argumento {
  public void spin() {}
}

public class ArgumentosFinais {
  void com(final Argumento g) {
    //! g = new Argumento(); // Ilegal  -- g e final
  }
  void sem(Argumento g) {
    g = new Argumento(); // OK -- g não final
    g.spin();
  }
  // void f(final int i) { i++; } // Não pode mudar
  // VocÍ pode somente ler dos dados primitivos:
  int g(final int i) { return i + 1; }
  public static void main(String[] args) {
    ArgumentosFinais bf = new ArgumentosFinais();
    bf.sem(null);
    bf.com(null);
  }
}
```

Exemplo 5. 13 - Argumentos Finais

Note que você pode ainda atribuir uma referência null a um argumento que é final. Os métodos f( ) e g( ) mostram o que ocorre quando argumentos primitivos são final: você pode ler os argumentos, mas você não pode altera-los.

5.2.8.2 – Métodos Finais

Existem duas razoes para se criar métodos finais. A primeira é colocar uma tranca no método, de modo a prevenir que qualquer classe herdeira modifique o significado do método. Neste caso, você quer garantir que o comportamento do método não é modificado durante a herança.

A segunda razão para criar métodos finais é eficiência. Se você torna um método *final*, você está permitindo que o compilador utilize chamadas "inline". Quando o compilador vai tratar uma chamada de método *final*, ele pode (a seu próprio critério) não utilizar o mecanismo comum de chamada de métodos (criando um registro de ativação do método na pilha). Ao invés disso, ele pode substituir a chamada do método por uma cópia do código descrito no corpo do método. Isto elimina o trabalho adicional envolvido na chamada do método.

Qualquer método privado em uma classe é implicitamente final. Uma vez que você não pode acessar um método privado, você também não pode sobrescrevê-lo. Note que o compilador não dá uma mensagem de erro, se você tentar sobrescrevê-lo. Na realidade, você não sobrescreveu o método, apenas criou um novo método. Você pode adicionar o especificador *final* a um método privado, contudo isto não dá ao método qualquer significado extra. Esta questão pode causar alguma confusão, uma vez que se você tentar sobrescrever um método privado (o qual é implicitamente *final*), isto parece funcionar:

```
// IlusaoFinal.java
// Aparentemente sobrescreve  método privado
```

```
class ComFinais {
  private final void f() {
    System.out.println("ComFinais.f()");
  }
  // Mesma coisa
  private void g() {
    System.out.println("ComFinais.g()");
  }
}

class SobrePrivado1 extends ComFinais {
  private final void f() {
    System.out.println("SobrePrivado1.f()");
  }
  private void g() {
    System.out.println("SobrePrivado1.g()");
  }
}

class SobrePrivado2 extends SobrePrivado1 {
  public final void f() {
    System.out.println("SobrePrivado2.f()");
  }
  public void g() {
    System.out.println("OverridingPrivate2.g()");
  }
}

public class IlusaoFinal {
  public static void main(String[] args) {
    SobrePrivado2 op2 = new SobrePrivado2();
    op2.f();
    op2.g();
    // Voce pode fazer "upcast"
    SobrePrivado1 op1 = op2;
    // Mas voce nao pode chamar os metodos
    //! op1.f();
    //! op1.g();
    // O mesmo aqui
    ComFinais wf = op2;
    //! wf.f();
    //! wf.g();
  }
}
```

Exemplo 5. 14 - Sobrescrevendo Métodos Privados

A sobrescrita pode ocorrer somente se alguma coisa é parte da interface da classe base. Isto é, você deve ser capaz de fazer "upcast" de um objeto para sua classe base e ainda assim poder chamar o método específico da classe original do objeto. Se um método é privado, ele não faz parte da interface da classe base. Este método é apenas algum código que está oculto dentro da classe. Se você cria um método com mesmo nome na classe derivada (seja ele público, privado, protegido ou amigo na classe derivada, não

existe nenhuma conexão com o método privado homônimo da classe base. Uma vez que um método privado está fora de alcance das outras classes e efetivamente invisível, este afeta somente a organização da classe na qual este foi definido.

5.2.8.3 – Classes Finais

Quando você diz que uma classe é final (precedendo sua definição com a palavra *final*), você não permite que alguém herde desta classe. Em outras palavras, por alguma razão de projeto, você não deseja que existam subclasses desta classe.

```
// Jurassico.java
// Classe final

class CerebroPequeno {}

final class Dinossauro {
  int i = 7;
  int j = 1;
  CerebroPequeno x = new CerebroPequeno();
  void f() {}
}

//! class Tiranossauro extends Dinossauro {}
// Nao pode estender a classe final Dinossauro

public class Jurassico {
  public static void main(String[] args) {
    Dinossauro n = new Dinossauro();
    n.f();
    n.i = 40;
    n.j++;
  }
}
```

Exemplo 5. 15 - Classes Finais

Note que os atributos membros podem ser finais ou não, como você escolher. A mesma regra aplica-se aos membros quer a classe seja final ou não. Definindo a classe como final simplesmente previne herança e nada mais. No entanto, uma vez que previne herança todos os métodos na classe são implicitamente finais, e desta forma não existe forma de sobrescrevê-los.

5.3 – Inicialização e Carga de uma Classe

Em linguagens mais tradicionais, programas são carregados completamente como parte do processo de carga. A seguir, ocorre a inicialização, e então o programa começa. Java adota uma abordagem de carregamento diferente. Uma vez que tudo em Java é um objeto, muitas atividades tornam-se mais fáceis e esta é uma delas. Em geral, você pode dizer que o código para uma classe é carregado no ponto em que a classe é utilizada pela primeira vez. Freqüentemente isto significa que o código é carregado quando o primeiro objeto da classe é construído, mas também pode ocorrer quando um atributo ou método estático é acessado. A inicialização também acontece no ponto de primeiro uso. Todos os objetos estáticos e o bloco de código estático serão inicializados na ordem textual, isto é, na ordem que você os escreveu na definição da classe.

5.3.1 – Inicialização com Herança

É útil olhar o processo global de inicialização, incluindo herança, de modo a obter uma visão geral do que ocorre. Considere o seguinte código:

```
// O processo completo de inicializacao.

class Inseto {
  int i = 9;
  int j;
  Inseto() {
    prt("i = " + i + ", j = " + j);
    j = 39;
  }
  static int x1 = prt("static Inseto.x1
inicializado");
  static int prt(String s) {
    System.out.println(s);
    return 47;
  }
}

public class Besouro extends Inseto {
  int k = prt("Besouro.k inicializado");
  Besouro() {
    prt("k = " + k);
    prt("j = " + j);
  }
  static int x2 = prt("static Besouro.x2
inicializado");
  public static void main(String[] args) {
    prt("Besouro construtor");
    Besouro b = new Besouro();
  }
}
```

A saida deste programa e :

```
static Inseto.x1 inicializado
static  Besouro.x2 inicializado
Besouro construtor
i = 9, j = 0
Besouro.k inicializado
k = 47
j = 39
```

A primeira coisa que ocorre quando você executa `java Besouro` é tentar acessar `Besouro.main( )` (um método estático). Desta forma, o carregador entra em ação e encontra o código compilado para a classe `Besouro` (código este que se encontra em um arquivo chamado `Besouro.class`). No processo de carregamento, o carregador nota que esta classe tem uma classe base (isto é o que a palavra *extends* diz). Assim, esta classe base é carregada. Isto ocorrerá independentemente de você criar ou não um objeto da classe base.

Se a classe base herda de uma outra classe, a segunda classe base será então carregada e assim sucessivamente. A seguir, a inicialização *static* na classe base raiz (neste caso, `Inseto`) é executada, e então a próxima classe derivada, e assim por diante. Isto é importante porque a inicialização *static* da classe derivada pode depender da inicialização apropriada de membros estáticos da classe base da classe.

Neste ponto, todas as classes necessárias foram carregadas de modo que os objetos podem ser criados. Primeiro, todos os primitivos neste objeto são ajustados para seus valores defaults e as referências aos objetos são ajustadas para *null*. Então, o construtor da classe base é chamado. Neste caso, a chamada e automática, mas você pode também especificar a chamada ao construtor da classe base (como primeira operação no construtor `Besouro`) utilizando a palavra reservada *super*. A construção da classe base segue o mesmo processo e na mesma ordem que o construtor da classe derivada. Depois que o construtor da classe base completou o seu trabalho, as variáveis são inicializadas na ordem textual. Por último, o resto do corpo do construtor é executado.

5.4 – Comentário sobre Herança

Em programação orientada a objetos, a forma mais provável de criação e uso de código é através do simples empacotamento de atributos e métodos em uma classe e da utilização de objetos desta classe. Você também utilizará classes existentes para construir novas classe por composição. Menos freqüentemente, você utilizará herança. Desta forma, embora herança obtenha muita ênfase durante a aprendizagem de POO, isto não significa que você deva utilizá-la em todos os lugares. Você deve utilizar o conceito de herança somente nos lugares onde este conceito pode ser verdadeiramente útil.

5.5 – Exercícios

1. Crie duas classes, A and B, com construtores default que se auto anunciam. Herde de A uma nova classe chamada de C e crie um membro B dentro de C. Não crie um construtor para C. Crie um objeto da classe C e observe os resultados.
2. Modifique o exercício 1 de modo que A e B tenham construtores com argumentos ao invés de construtores default. Escreva um construtor para C e realize toda a inicialização de dentro do construtor de C.
3. Comente o construtor da classe Cartao no arquivo Cartao.java, compile, execute e explique o que ocorre.
4. Comente o construtor da classe Xadrez no arquivo Xadrez.java, compile, execute e explique o que ocorre.
5. Crie uma subclasse Empregado da classe Pessoa, implementada nos exercícios do capítulo anterior, para representar um empregado de uma empresa. Além do nome e data de aniversário, um empregado possui uma matrícula, uma data de entrada na empresa e um salário. Implemente métodos para a leitura e escrita dos dados de um empregado e para a atualização dos mesmos.
6. Crie uma subclasse Formado a partir da classe Pessoa. Além dos dados de Pessoa, uma pessoa formada possui como atributos adicionais o ano de formatura e o curso no qual se formou. Implemente métodos para a leitura e escrita dos dados de um formado e para a atualização dos mesmos.

5.6 – Trabalho

Use a classe Empregado do exercício 5 para fazer um programa que mantenha o cadastro dos empregados de uma empresa. Além das operações comuns sobre cadastro

(leitura, inclusão, alteração, exclusão e salvamento), o programa deve permitir o cálculo do custo total mensal da empresa com salários, a identificação do funcionário mais antigo, a impressão da lista de funcionários ordenada por nome, a impressão da lista de funcionários ordenada por matrícula e a impressão da lista de funcionários ordenada por salário.