Capítulo 4

Classes e Objetos

Tente fazer uma lista de quatro objetos e de quatro não objetos. Você poderá notar que é muito mais fácil listar coisas que são objetos do que listar coisas que não são objetos. A tal ponto que falar a respeito de alguma coisa parece transformá-la em objeto!! Descartes (um famoso filósofo do século XVII) observou que os seres humanos vêem o mundo em termos de objetos. O cérebro humano "deseja" pensar em termos de objetos, e nossos pensamentos e memórias são organizados em termos de objetos e seu inter-relacionamento.

Uma das idéias da construção de software orientado a objetos é organizar o software de uma maneira a tornar mais fácil o trabalho da nossa mente orientada a objetos. Ao invés de instruções de máquina que mudam padrões de bits na memória principal, nós desejamos "coisas" que fazem "alguma coisa". É claro, que no nível de instruções de máquina nada mudou, mas pelo menos nós não temos que pensar a respeito disto. Programação orientada a objetos faz parte de um movimento mais amplo, que tenta transformar o computador num meio de expressão mais próximo ao nível humano. Este capítulo apresentará alguns dos conceitos básicos da programação orientada a objetos.

4.1 – Introdução aos Objetos

Todas as linguagens de programação fornecem mecanismos de abstrações. Pode-se argumentar que os problemas que nós podemos resolver estão diretamente relacionados com tipo e a qualidade das abstrações que nós somos capazes de construir.

A abordagem orientada a objetos fornece ferramentas ao programador para representar os elementos existentes no espaço do problema. Este tipo de representação é bastante geral de modo que o programador não fica restrito a nenhum tipo específico de problema. Os elementos no espaço do problema e sua contrapartida no espaço de programação são referidos como "*objetos"* (é claro que você também necessitará de outros objetos que não tem análogo no espaço do problema). Dessa forma o paradigma orientado a objetos (POO) permite que você descreva o problema em termos análogos àqueles que você utilizaria para descrever o problema em linguagem natural. É claro que ainda existe uma conexão com o computador, afinal a máquina ainda será responsável por resolver o problema em última instância.

Olhe novamente a sua lista de objetos. O que você acha que caracteriza uma entidade como sendo um objeto? Algumas respostas possíveis são:

- Um objeto é feito de material tangível (uma caneta é feita de plástico, metal, tinta, etc).
- Um objeto é um todo indivisível (uma caneta é um todo).
- Um objeto tem propriedades (a cor da caneta, seu peso, etc).
- Um objeto pode realizar coisas e ter coisas realizadas para ele.

A primeira resposta parece ser bastante restritiva, uma vez que a sua conta bancária é um objeto, mas este não é feito de material. Ou seja, embora você e seu banco possam utilizar algum tipo de material para manter o extrato de sua conta, sua conta é

independente deste material. Embora seja imaterial, a sua conta bancária possui propriedades (o saldo, o nome do proprietário, o valor aplicado, etc) e você pode realizar ações na sua conta (depositar dinheiro, sacar, encerrar a conta, etc).

Os três últimos itens da lista parecem ser adequados. De fato, cada um destes itens possui um nome:

- Um objeto tem uma identidade (que possibilita que o objeot atue como um todo).
- Um objeto tem um estado (formado pelos valores das propriedades do objeto) que pode mudar ao longo do tempo.
- Um objeto tem um comportamento (que possibilita ao objeto fazer coisas e ter coisas realizadas para ele).

A lista acima caracteriza o que é um objeto, mas o que caracteriza uma linguagem orientada a objetos? Alan Kay resumiu as cinco características básicas de Smalltalk, a primeira linguagem orientada a objetos bem sucedida. Estas características representam uma abordagem pura para a programação orientada a objetos:

1. **Tudo que existe são objetos.** Pense em um objeto como sendo uma espécie de variável refinada; este armazena dados, mas você pode também "fazer pedidos" a um objeto, de modo que este execute operações sobre si mesmo. Em teoria, você pode transformar qualquer componente conceitual do problema que você está tentando resolver (cachorros, prédios, serviços, etc.) e representá-los como um objeto em seu programa.

2. **Um programa é composto por um grupo de objetos dizendo um ao outro o que fazer por meio de mensagens enviadas.** Para fazer um pedido para um objeto você envia uma mensagem para aquele objeto. Mais concretamente, você pode pensar em uma mensagem como uma chamada de um subprograma que pertence a um objeto particular.

3. **Cada objeto tem sua própria memória feita de outros objetos.** Colocando de outra forma, você cria uma nova espécie de objeto empacotando objetos já existentes. Assim, você pode reduzir a complexidade de um programa enquanto, ao mesmo tempo, a oculta através da simplicidade dos objetos.

4. **Todo objeto tem um tipo.** Utilizando o vocabulário, cada objeto é uma *instância de uma classe*, em que "classe" é sinônimo de "tipo". Uma das característica mais importantes de uma classe é "Quais mensagens você pode mandar para ela?"

5. **Todos objetos de um tipo particular podem receber as mesmas mensagens.** Esta é realmente uma declaração forte, como você verá mais tarde. Porque um objeto do tipo círculo é também um objeto da classe forma, um círculo com certeza pode aceitar todas as mensagens aceitas por uma forma. Isto significa que você pode escrever código que se comunique com uma forma e automaticamente manipule tudo que se ajusta na descrição de uma forma. Esta capacidade de substituição é um dos conceitos mais poderosos da POO.

4.1.1– Um Objeto tem uma Interface

Aristóteles foi provavelmente o primeiro a começar um estudo cuidadoso do conceito de tipo; ele disse "a classe dos peixes e a classe dos pássaros". A idéia de que todo objeto, enquanto único, é também parte de uma classe de objetos que possui características e comportamentos em comum foi usada diretamente pela primeira linguagem orientada a objeto, Simula-67, por meio da palavra reservada **class**, a qual introduz um novo tipo dentro de um programa.

Simula, como seu nome indica, foi criada para o desenvolvimento de simulações, tal como o clássico problema do "caixa de banco". Objetos que são idênticos exceto por seu estado durante a execução do programa são agrupados em classes de objetos. A criação de tipos de dados abstratos (classes) é um conceito fundamental em programação orientada a objetos. Tipos de dados abstratos trabalham quase exatamente como tipos de dados pré-construídos: Você pode criar variáveis de um tipo (chamadas de objetos ou instâncias) e manipular estas variáveis. Os membros de cada classe compartilham alguns elementos comuns: toda a conta tem um saldo, todo caixa pode aceitar um depósito, etc. Ao mesmo tempo, cada membro tem seu próprio estado, cada conta tem seu próprio saldo, cada caixa tem seu próprio nome. Assim, os caixas, consumidores, contas, transações, etc. podem cada um deles ser representado por uma entidade única no programa de computador. Esta entidade é o objeto, e cada objeto pertence a uma classe particular que define suas características e seu comportamento.

Assim, embora o que nós realmente façamos em programação orientada a objetos seja criar novos tipos de dados, virtualmente todas as linguagens orientadas a objeto utilizam a palavra reservada "class". Quando você observar a palavra "tipo" pense em "classe" e vice-versa.

Desde que uma classe descreve um conjunto de objetos que possui características (dados) e comportamento (funcionalidade) similares, a classe é realmente um tipo de dado. A diferença é que o programador define uma classe para se ajustar ao problema ao invés de ser forçado a utilizar tipos de dados existentes que foram projetados para representar uma unidade de memória da máquina. Você estende a linguagem de programação por meio da adição de novos tipos de dados específicos para suas necessidades. O sistema de programação aceita a nova classe e lhe confere todo cuidado e checagem de tipos que dá aos tipos pré-definidos.

Uma vez que uma classe está definida, você pode fazer tantos objetos da classe quantos desejar, e então manipular estes objetos como se eles fossem os elementos que existem no problema que você está tentando resolver. Realmente, um dos maiores desafios da programação orientada a objetos é criar um mapeamento um para um entre os elementos do espaço do problema e os objetos no espaço de programação.

Mas como um objeto faz o trabalho que você quer que ele faça? Deve haver uma maneira de fazer um pedido ao objeto de modo que ele faça alguma operação, tal como completar uma transação, desenhar alguma coisa na tela, etc. Além disso, cada objeto só deve poder satisfazer alguns tipos de pedidos. Os pedidos que você pode fazer a um objeto são definidos na sua *interface*. Um exemplo simples pode ser dado através da representação de uma lâmpada (figura 4.1):

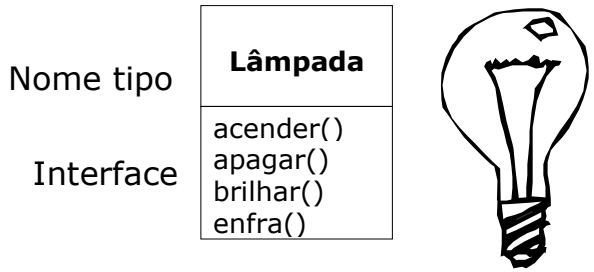

Figura 4. 1 - Interface da Classe Lâmpada

```
Lampada lt = new Lampada();
lt.acender();
```

A interface estabelece quais pedidos você pode fazer a um objeto particular. No entanto, deve existir um código em algum lugar que satisfaça àquele pedido. Este código, juntamente com a implementação das operações ocultas, formam a implementação. Do ponto de vista da programação imperativa, isto não é complicado. Um tipo sendo criado pode ter associado a ele uma função relacionada com cada pedido de operação possível, e quando você faz um pedido particular para um objeto, a função é chamada. Este processo é usualmente resumido dizendo-se que você "manda uma mensagem" para um objeto, e o objeto determina o que fazer com a mensagem (isto é, executa o código apropriado).

Aqui, o nome do tipo (classe) é `Lampada`, o nome do objeto particular é `lt`, e os pedidos que você pode fazer a um objeto da classe `Lampada` são acendê-lo, desligá-lo, torná-lo mais brilhante, ou fazê-lo mais fraco. Você cria um objeto Lampada definindo uma "referência" (`lt`) para o objeto e chamando new para criar um novo objeto daquele tipo. Para mandar uma mensagem para o objeto, você utiliza o nome do objeto e conecta a mensagem desejada por meio de um ponto.

4.1.2– Ocultando a Implementação

É útil dividir o universo de programadores em programadores de classes (aqueles que criam os novos tipos de dados) e programadores clientes (os consumidores que utilizam os novos tipos de dados em suas aplicações). A meta do programador cliente é colecionar um conjunto de classes que lhe permita desenvolver sua aplicação rapidamente. A meta do criador de classes é construir uma classe que exponha somente o que é necessário para o programador cliente e mantenha todo o resto oculto. Porquê? Porque se está oculto, o programador cliente não pode utilizá-lo, o que significa que o programador da classe pode mudar a porção oculta sem se preocupar com o impacto que esta mudança causará no programa cliente. A porção oculta normalmente representa a parte sensível do objeto que poderia ser facilmente corrompida por um programador cliente descuidado ou desinformado. Dessa forma, a implementação oculta reduz a possibilidade de ocorrência de erros no programa. Nunca é demais enfatizar a importância do ocultamento da implementação.

Em qualquer relacionamento é importante ter fronteiras que são respeitadas por todas as partes envolvidas. Quando você cria uma biblioteca, você estabelece um relacionamento com o programador cliente, que é também um programador, mas cujo objetivo é

implementar uma aplicação utilizando sua biblioteca, possivelmente para construir uma biblioteca maior.

Se todos os membros de uma classe estão disponíveis para todo mundo, então o programador cliente pode fazer qualquer coisa com aquela classe e não há maneira de forçar nenhuma regra. Mesmo que você queira que o programador cliente não manipule diretamente alguns dos membros de sua classe, sem controle de acesso não há maneira de prevenir isto. Tudo está a mostra para o mundo.

Assim, a primeira razão para controlar acesso é manter porções da classe fora do alcance do programador cliente (partes que são necessárias para o funcionamento interno do tipo de dado) e outras porções ao alcance dele (a interface que o usuário necessita a fim de poder resolver o seu problema particular). Isto é realmente um serviço que está sendo prestado ao usuário porque eles podem facilmente ver o que é importante para eles enquanto o que não lhes interessa é ignorado.

A segunda razão para controlar o acesso é permitir mudanças no funcionamento interno de uma classe sem se preocupar em como isto afetará o programa cliente. Por exemplo, você pode implementar uma classe particular de uma maneira simples de modo a facilitar o desenvolvimento, e então descobrir posteriormente que você necessita reescrevê-la de modo a torná-la mais eficiente. Se a interface e implementação estão claramente separados e protegidos, você pode realizar isto facilmente.

Java utiliza *especificadores de acesso* para determinar quem pode usar os membros das classes.

4.1.3– Reutilizando a Implementação

Uma vez que uma classe tenha sido criada e testada, esta deveria (idealmente) representar uma unidade de código útil. Reutilização de código é uma das grandes vantagens das linguagens de programação orientadas a objetos.

A maneira mais simples de reutilizar uma classe é apenas utilizando um objeto daquela classe, mas você pode também colocar esse objeto dentro de uma nova classe. Nós chamamos isto de "criação de um objeto membro". Uma nova classe pode ser feita com qualquer número e tipo de outros objetos, em qualquer combinação que você necessitar para alcançar a funcionalidade desejada em sua nova classe. Porque você estão compondo uma nova classe a partir de classes existentes, este conceito é chamado de *composição* (ou *agregação*). Composição é freqüentemente referida como um relacionamento "tem-um", tal como na frase "um carro tem um motor".

O diagrama UML da figura 4.2 indica a operação de composição através da linha terminada com um losângulo preenchido. Composição é um mecanismo bastante flexível. Os objetos membros da nova classe são freqüentemente privados, fazendo-os inacessíveis para os programadores clientes que estão utilizando a classe. Isto permite que você mude estes membros sem perturbar o programa cliente. Você pode também mudar os objetos membros em tempo de execução, com o objetivo de mudar dinamicamente o comportamento do seu programa.

Figura 4. 2 - Composição

4.1.4– Criação de Objetos

Quando você cria uma referência, você quer conectá-la com um novo objeto. De uma forma geral você faz isto através do operador *new*. Intuitivamente new significa: "Faça um novo objeto". Assim, no exemplo seguinte, você pode dizer:

```
String s = new String("asdf");
```

Isto não significa apenas "Fazer um novo objeto string", mas também indica como fazê-la, fornecendo uma string inicial de caracteres.

É claro que a classe *String* não é o único tipo que existe. Java possui um elenco enorme de classes "pré-fabricados". Contudo, o mais importante é que você pode criar seus próprios tipos. De fato, esta é a atividade fundamental quando se está programando em Java.

4.1.4.1 – Vetores em Java

Virtualmente todas as linguagens de programação suportam vetores. O uso de vetores em C e C++ é uma tarefa complexa porque vetores são considerados como sendo simplesmente blocos de memória. Se um programa acessa uma posição fora do bloco ou utiliza a memória antes da inicialização dos vetores os resultados são imprevisíveis.

Uma das principais metas de Java é confiabilidade, de modo que muitos dos problemas com ponteiros que atormentavam os programadores em C e C++ foram sanados em Java. Garante-se em Java que um vetor é sempre inicializado e que ele não pode ser acessado fora do seu alcance. A checagem dos limites do vetor tem o ônus de demandar a utilização de um pequeno montante de memória adicional em cada vetor bem como o ônus de verificar em tempo de execução os limites do vetor, mas a suposição é que a confiabilidade e o aumento de produtividade compensam este custo adicional.

Quando você cria um vetor de objetos, você realmente está criando um vetor de referências, e cada uma dessas referências é automaticamente inicializada para um valor especial: ***null***. Quando Java encontra um valor *null*, este reconhece que a referência em questão não está apontando para um objeto. Você deve atribuir um objeto para cada referência antes que você possa utilizá-la, e se você tentar utilizar uma referência que é ainda *null***,** o problema será reportado em tempo de execução.

Você pode também criar um vetor de tipos primitivos. Novamente, o compilador garante a inicialização, atribuindo o valor default do tipo primitivo aos elementos do vetor.

4.1.5– Destruição de Objetos

Em muitas linguagens de programação, o conceito de tempo de vida de uma variável ocupa uma porção significativa do esforço de programação. Qual é o tempo de vida da variável x? Se você é responsável por destruí-la, quando deve fazer isto? Confusões a respeito do tempo de vida podem conduzir a muitos erros de programação. Esta seção mostra como Java simplifica bastante a vida do programador, uma vez que faz o trabalho de limpeza automaticamente.

4.1.5.1 – Escopo

Muitas linguagens imperativas tem o conceito de escopo. Este determina a visibilidade e o tempo de vida dos nomes definidos naquele escopo. Em C, C++ , e Java, o escopo é determinado pela colocação dos símbolos abre e fecha-chaves ({ }**)**. Por exemplo:

```
{
  int x = 12;
  /* somente x disponÌvel */
  {
    int q = 96;
    /* ambos x & q disponÌveis */
  }
  /* somente x disponÌvel */
  /* q "fora de escopo" */
}
```

Exemplo 4. 1 - Exemplo de Escopo

Uma variável de tipos primitivos definida dentro de um escopo está disponível somente até o fim do escopo.

Objetos em Java não tem o mesmo tempo de vida que as variáveis de tipos primitivos. Quando você cria um objeto em Java utilizando *new*, este objeto estende-se além do fim do seu escopo. Dessa forma se você utiliza:

```
{
  String s = new String("uma string");
} /* fim do escopo */
```

A referência **s** desaparece no fim do escopo. No entanto, o objeto *String* que foi apontado por s ainda existe. Neste segmento de código, não há outra maneira de acessar o objeto porque a única referência para ele está fora de seu escopo.

Em C++, o problema com este objeto pendente ganha um dimensão mais dramática porque você não tem nenhuma ajuda da linguagem. De modo que, em C++, você deve explicitamente destruir os objetos quando não necessita mais deles. Isto levanta uma questão interessante. Se Java permite a existência de objetos sem que exista nenhuma referência para eles, o que evita que estes objetos entulhem a memória do computador e conduzam o programa a uma parada por falta de memória, causando um problema de programação conhecido como vazamento de memória? Note que é exatamente este tipo de problema que ocorre em C++ quando os programadores se esquecem de destruir os objetos. Neste ponto é que se faz presente uma das características mais importantes de Java. Java possui um *coletor de lixo* (*garbage collector*), este coletor de lixo é responsável por manter uma lista de todos os objetos criados com new e determinar quando não há mais nenhuma referência para um objeto. Quando isto ocorre a memória para estes objetos é liberada, de modo que esta memória possa ser utilizada por novos objetos. Isto significa que você não necessita se preocupar com a destruição dos objetos e que o problema de vazamento de memória nunca ocorre.

4.1.6– Criação de Classes

Se tudo é um objeto, o que determina a estrutura e o comportamento de uma classe de objetos? Em outras palavras, o que define o tipo de um objeto? Seria razoável esperar que existisse uma palavra reservada chamada "*tipo*". Historicamente, no entanto, muitas linguagens orientadas a objetos utilizam a palavra reservada *class* para significar algo como: "Eu estou informando a você como o novo tipo deve se comportar". O nome do novo tipo deve vir logo depois da palavra reservada class. Por exemplo:

```
class UmNomedeTipo { /* o corpo da classe vai aqui
*/ }
```

Isto introduz um novo tipo, de modo que nós podemos agora criar um objeto deste tipo utilizando *new*:

```
UmNomedeTipo a = new UmNomedeTipo ();
```

4.1.6.1 – Atributos e Métodos

Quando você define uma classe (e tudo que você faz em Java é definir classes, criar objetos destas classes e mandar mensagens para estes objetos), você pode colocar dentro dela dois tipos de elementos: variáveis membros (algumas vezes chamadas de atributos), e funções membro (tipicamente chamadas de métodos). Uma variável membro é um objeto de qualquer tipo com o qual você pode se comunicar através de sua referência. Ela pode ser também um dos tipos primitivos (que não é uma referência). Se ela é uma referência para um objeto, você deve inicializar esta referência de modo a conectá-la a um objeto real (utilizando *new*) normalmente através de uma função especial chamada *construtor*. A variável membro também pode ser inicializada diretamente na definição da classe.

```
class SoDados {
  int i;
  float f;
  boolean b;
}
```

Exemplo 4. 2 - Exemplo de Atributos de Classe

Embora esta classe não faça nada, você pode criar objetos desta classe:

```
SoDados d = new SoDados();
```

Você pode atribuir valores às variáveis membros utilizando o operador de seleção (.), conforme mostra o exemplo a seguir:

```
d.i = 47;
d.f = 1.1;
d.b = false;
```

É também possível que um objeto contenha outros objetos que contenham dados que você gostaria de modificar. Para fazer isto você apenas necessita "conectar os operadores de seleção". Por exemplo:

```
estacao.reservatorio.capacidade = 100;
```

4.1.6.2 – Valores Default para Membros Primitivos

Quando um membro de uma classe de um tipo de dado primitivo não é inicializado durante a sua declaração, Java atribui automaticamente um valor default para este membro. A tabela abaixo apresenta os valores default dos tipos primitivos:

Tipo Primitivo	Default
`boolean`	`False`
`char`	`'\u0000' (null)`
`byte`	`(byte)0`
`short`	`(short)0`
`int`	`0`
`long`	`0L`
`float`	`0.0f`
`double`	`0.0d`

Tabela 4. 1 - Valores Default de Tipos Primitivos

Note cuidadosamente que os valores default são o que Java garante quando a variável é utilizada como membro de uma classe. Isto assegura que as variáveis membros de tipos primitivos serão sempre inicializadas (algo que C++ não faz). No entanto, este valor inicial pode não ser o valor correto ou mesmo o valor legal para o programa que você está escrevendo. Portanto, trata-se de melhor prática de programação inicializar sempre explicitamente suas variáveis membro. Além disso, esta garantia não se aplica a variáveis locais – aquelas que não são atributos. Assim, se dentro da definição de um método, nós temos:

```
int x;
```

Então, **x** obterá algum valor arbitrário (tal como em C e C++) não sendo automaticamente inicializada para zero. Você é responsável por atribuir um valor apropriado antes que **x** seja utilizada. Se você esquecer e tentar utilizar o valor desta variável, ocorrerá um erro em tempo de compilação dizendo que a variável pode não ter sido inicializada.

4.1.6.3 – Métodos, Parâmetros e Valores de Retorno

A classe `SoDados` do exemplo anterior não pode fazer muito mais do que reter dados, uma vez que não possui nenhuma função membro (método). Para entender como isto funciona, você deve entender antes o conceito de parâmetro e valor de retorno.

Métodos em Java determinam as mensagens que um objeto pode receber. Nesta seção você aprenderá como é simples definir um método.

A partes fundamentais de um método são o nome, os parâmetros, o tipo de retorno, e o corpo. Aqui esta a forma básica:

```
tipoRetorno metodoNome ( /* Lista de par,metros */ ) {
  /* Corpo do metodo */
```

```
}
```

O tipo de retorno é o tipo do valor produzido pelo método depois de sua chamada. A lista de parâmetros contém os tipos e os nomes para as informações que você passa para o método. O nome do método e a lista de parâmetros identificam o método de forma única.

Métodos em Java podem ser criados somente como parte de uma classe. Um método pode ser chamado somente por um objeto e aquele objeto deve ser capaz de executar o método. Se você tentar chamar um método errado para um objeto você obterá uma mensagem de erro em tempo de compilação. Você chama um método para um objeto nomeando o objeto e a seguir utilizando o operador ponto, seguido pelo nome do método e sua lista de parâmetros: `nomeObjeto.nomeMetodo (arg1, arg2, arg3)`. Por exemplo, suponha que você tenha um método `f( )` que retorna um valor do tipo *int*. Então, se você tem um objeto chamado **a** ao qual o método `f( )` pode ser aplicado, então é legítimo dizer:

```
int x = a.f();
```

O tipo do valor de retorno de `f()` deve ser compatível com o tipo de **x**.

Este ato de invocar um método é normalmente referido como *enviar uma mensagem para um objeto*. No exemplo acima, a mensagem é `f( )` e o objeto é `a`. A programação orientada a objetos é freqüentemente resumida como simplesmente "o envio de mensagens para objetos".

4.1.6.3.1 – Lista de Parâmetros

A lista de parâmetros especifica quais informações você passa para o método. Como você poderia esperar, esta informação – como todo o resto em Java – toma a forma de objetos ou tipos primitivos. Para definir a lista de parâmetros é necessário que você especifique os tipos e os nomes de cada um dos parâmetros.

A passagem de valores para os parâmetros funciona de modo equivalente à operação de atribuição. Quando o parâmetro é do tipo primitivo, a passagem equivale a uma atribuição por valor, isto é, faz-se uma cópia do valor do tipo primitivo para o parâmetro. Já se o parâmetro é do tipo objeto, atribui-se uma referência para o parâmetro. O tipo do argumento utilizado na chamada do método deve ser equivalente ao tipo declarado do parâmetro. Por exemplo, se o parâmetro é declarado como sendo uma *String*, o que você deve passar é uma *String*.

Veja abaixo um método que toma uma *String* como seu parâmetro. A sua definição deve ser colocada dentro de uma definição de classe para que possa ser compilada:

```
int memoriaArmazenada(String s) {
  return s.length() * 2;
}
```

Este método diz quantos bytes são requeridos para reter a informação em uma determinada *String* (cada *char* em uma *String* ocupa 16 bits, ou dois bytes, de modo a suportar os caracteres Unicode). O parâmetro é do tipo *String* e é chamado de **s**. Uma vez que **s** é passado para o método, você pode tratá-lo como se fosse um outro objeto

qualquer. Neste exemplo, o método `length( )`, um dos métodos de *String***,** é chamado. Ele retorna o número de caracteres em uma string.

Você pode observar também o uso da palavra chave *return*, a qual faz duas coisas. Primeiro, ela significa que o método deve ser interrompido. Segundo, se o método produz um valor, este valor é colocado à direita do *return*, logo depois da declaração de retorno. Neste caso, o valor de retorno é produzido pela avaliação da expressão `s.length( ) * 2`.

Você pode retornar qualquer tipo desejado, mas se você não quer retornar nada , você deve indicar isto explicitamente declarando que o método retorna *void*. Aqui estão alguns exemplos de métodos:

```
boolean flag() { return true; }
float naturalLogBase() { return 2.718f; }
void nada() { return; }
void nada2() {}
```

Quando o tipo do valor de retorno é *void*, a palavra chave *return* somente é utilizada para sair do método. Portanto, ela se torna desnecessária quando você alcança o fim do método. Você pode interromper um método usando *return* em qualquer ponto do mesmo, mas se você não fornecer um valor de retorno do tipo apropriado, o compilador lhe forçará a fazê-lo através da detecção de um erro de compilação.

Neste ponto, podemos observar que um programa é apenas um grupo de objetos com métodos que tomam outros objetos como parâmetros e mandam mensagens para estes outros objetos. Isto é boa parte do trabalho que você tem de fazer ao escrever um programa orientado a objetos.

4.1.6.4 – Atributos e Métodos de Classe

Normalmente, quando uma classe é criada, você está descrevendo como objetos daquela classe são constituídos e como eles se comportam. Você realmente não fez nada até que se crie um objeto daquela classe com *new*, e é naquele ponto que a memória é reservada e os métodos se tornam disponíveis.

Há, contudo, duas situações em que esta abordagem não é adequada. Uma é se você quer ter um único atributo (variável) associado a uma classe , independente de quantos objetos são criados, ou mesmo se algum objeto é criado. A outra situação é se você necessita um método que possa ser chamado mesmo quando nenhum objeto tenha sido criado. Você pode alcançar os dois efeitos através da palavra reservada *static*. Quando você diz que alguma coisa é *static*, isto significa que a variável ou o método não é amarrado a qualquer objeto particular daquela classe.

Dessa forma, mesmo se nunca foi criado um objeto daquela classe, pode-se chamar um método *static* ou acessar uma variável *static*. Com variáveis ou métodos que não são *static*, você deve criar um objeto e utilizar aquele objeto para acessar a variável ou método . Uma vez que os métodos *static* não necessitam da existência de qualquer objeto, eles não podem acessar *diretamente* membros ou métodos não *static***.**

Algumas linguagens orientadas a objetos utilizam o termo *método de classe* e *variável de classe*, significando que estes atributos e métodos existem somente para a classe como um todo, e não para qualquer objeto particular da classe. Algumas vezes a literatura de Java também utiliza este termo.

Para fazer um método ou atributo *static*, você simplesmente coloca esta palavra antes da sua definição. Por exemplo, o seguinte segmento de código produz uma variável membro *static* e o inicializa:

```
class TesteVariavelDeClasse {
    static int i = 47;
    int j = 26;
}
```

Agora, mesmo se você fizer dois objetos `TesteVariavelDeClasse`, somente haverá uma única variável `TesteVariavelDeClasse.i`. Ambos objetos compartilharão o mesmo `i`. Por outro lado, haverá duas variáveis membro `j` (uma para cada objeto). Considere:

```
TesteVariavelDeClasse st1 = new TesteVariavelDeClasse ();
TesteVariavelDeClasse st2 = new TesteVariavelDeClasse ();
```

Neste ponto, `st1.i` e `st2.i` têm o mesmo valor 47, uma vez que ambos se referem a mesma variável.

Há duas maneiras para se referir a uma variável *static*. A primeira é tal como mostrado acima, onde você se refere a variável via um objeto da classe, como por exemplo, `st2.i`. A outra maneira é se referir diretamente a esta variável através de seu nome de classe, algo que não é possível de ser feito com um membro não static.

```
TesteVariavelDeClasse.i++;
```

Neste exemplo, o operador ++ incrementa a variável. Neste ponto, tanto `st1.i` quanto `st2.i` terão o valor 48. Por sua vez, se fizermos `st1.j++` , a variável `j` de `st1` será incrementada para 27, enquanto que a `j` de `st2` continuará com o valor 26. A figura 4.2 ilustra a diferença entre atributos de classe e atributos de objetos.

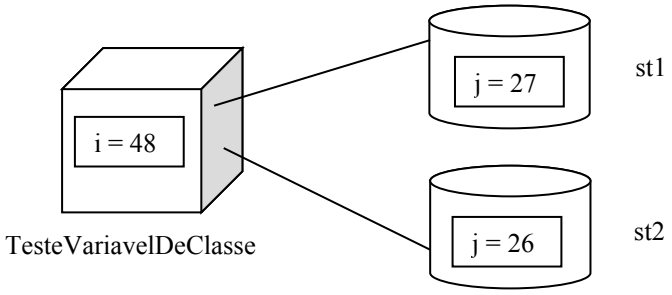

Figura 4. 3 - Atributos de Objetos e Atributos de Classe

Você define um método *static* de uma maneira similar a definição dos métodos normais:

```
class MetodoDeClasse {
  static void incr() { TesteVariavelDeClasse.i++; }
}
```

Você pode se referir a um método *static* através de um objeto, tal como você faz com qualquer método, ou através da sintaxe adicional especial `NomeDaClasse.metodo( )`. No exemplo acima, você pode ver que o método `incr( )` de `MetodoDeClasse` incrementa a variável *static* `i`. Você pode chamar `incr( )` da maneira típica, através de um objeto:

```
MetodoDeClasse sf = new MetodoDeClasse ();
sf.incr();
```

Ou chamar o método `incr( )` diretamente através da classe, uma vez que ele é um método static.

```
MetodoDeClasse.incr();
```

Quando *static* é aplicado a uma variável membro, ele muda definitivamente a maneira como o atributo é criado (um para cada classe, ao invés de um para cada objeto). Já quando é aplicado ao método, a mudança não é tão grande. Um uso importante de *static* aplicado a métodos é permitir que você chame o método sem ter de criar um objeto. Isto é essencial na definição do método `main` que é o ponto de entrada para a execução de sua aplicação.

4.2 – Inicialização e Remoção

Neste capítulo, já vimos como definir classes e como criar objetos destas classes através do operador *new*. Entretanto, muitas vezes, a criação de objetos não deve consistir apenas da alocação de espaço de memória para o objeto e da associação desta área de memória com a referência utilizada para acessá-la. Também é importante inicializar os valores dos atributos do objeto. Por outro lado, quando um objeto deixa de ser utilizado (isto é, não há mais nenhuma referência para ele), a área de memória utilizada deve ser disponibilizada para a criação de novos objetos. É preciso entender como isso é feito em Java.

Java utiliza o conceito de *construtor,* um método especial que é automaticamente chamado quando um objeto é criado, para permitir a inicialização de objetos. Adicionalmente, Java tem um coletor de lixo que automaticamente libera memória quando os objetos não são mais necessários.

4.2.1- Inicialização com o Construtor

Você pode criar um método chamado `inicializar( )` para toda classe que você escreve. Este método deveria ser chamado antes que o objeto fosse utilizado. Infelizmente, isto significa que o usuário deve se lembrar de chamar o método. Em Java, o projetista da classe pode garantir a inicialização de todo objeto fornecendo para isto um método especial chamado de construtor. Se uma classe tem um construtor, Java automaticamente chama este construtor quando um objeto é criado, antes que os usuários possam utilizá-los. Dessa forma, a inicialização é garantida.

O próximo desafio consiste em definir qual o nome deste método. Há duas questões: a primeira é que qualquer nome utilizado para o construtor poderia conflitar com um nome de um método membro da classe; a outra questão é que, uma vez que o compilador é responsável por chamar o construtor, este deve sempre saber qual método

chamar. A solução de aparentemente mais fácil e lógica é dar o nome da classe ao construtor. Mostra-se a seguir um exemplo de uma classe simples com um construtor:

```
// ConstrutorSimples.java
// Demonstracao de um construtor simples.
class Rocha {
  Rocha() { // Este e o construtor
    System.out.println("Criando uma Rocha");
  }
}

public class ConstrutorSimples {
  public static void main(String[] args) {
    for(int i = 0; i < 10; i++)
      new Rocha();
  }
}
```

Exemplo 4. 3 - Uso de Método Construtor

Agora, quando um objeto é criado através da sentença `new Rocha();` a memória é alocada para o objeto e o construtor é chamado. Isto garante que o objeto será apropriadamente inicializado antes que você possa utilizá-lo.

Como qualquer método, o construtor pode ter argumentos para permitir que se especifique como um objeto é criado. O exemplo acima pode ser facilmente mudado de modo que o construtor tenha argumentos:

```
// ConstrutorSimples2.java
// Construtor pode ter argumentos.
class Rocha2 {
  Rocha2(int i) {
    System.out.println(
      "Criando Rocha numero " + i);
  }
}
public class ConstrutorSimples2 {
  public static void main(String[] args) {
    for(int i = 0; i < 10; i++)
      new Rocha2(i);
  }
}
```

Exemplo 4. 4 - Uso de Construtor com Argumentos

Os argumentos no construtor permitem parametrizar a inicialização de um objeto. Por exemplo, se uma classe `Arvore` tem um construtor que toma como argumento um inteiro denotando a altura da árvore, você poderia criar um objeto árvore como este:

```
Arvore t = new Arvore(12);  // arvore de 12 metros
```

Se `Arvore(int)` é seu único construtor, então o compilador não permitirá que você crie um objeto Árvore sem que se especifique a sua altura.

Construtores eliminam um série de problemas e tornam o código mais fácil de ler. No segmento de código precedente, por exemplo, você não vê uma chamada explícita a

nenhum método `initializar( )` que é conceitualmente separado da definição. Em Java, definição e inicialização são conceitos unificados – você não pode ter um sem ter o outro. O construtor é um tipo de método incomum, uma vez que ele não possui nenhum valor de retorno.

4.2.2- Sobrecarga de Métodos

Uma característica importante em qualquer linguagem de programação é o uso de nomes. Quando você cria um objeto, você dá um nome para uma região de memória. Quando cria um método, você dá um nome para uma ação. Utilizando nomes para descrever seu sistema, você cria um programa que é mais fácil de ler, entender e modificar.

Um problema surge quando mapeamos os conceitos em linguagem humana para uma linguagem de programação. Freqüentemente, a mesma palavra expressa vários conceitos diferentes. Dizemos neste caso que esta palavra está **sobrecarregada**. Isto é útil, especialmente quando as diferenças são triviais. Você diz "lave a camisa", "lave o carro", e "lave o cachorro". Seria muito ruim ser forçado dizer, "laveCamisa a camisa", "laveCarro o carro", e "laveCachorro o cachorro" de modo que o ouvinte não necessite fazer qualquer distinção entre as ações a serem executadas.

Muitas linguagens de programação (em particular, C) demandam que você tenha um identificador único para cada função. Em Java (e C++), é possível utilizar um único nome para várias funções. De certa maneira, a sobrecarga de nomes de métodos é forçada pelo conceito de construtor. Uma vez que o nome do construtor é pré-determinado pelo nome da classe, somente pode haver um único nome de construtor. Contudo, pode-se querer criar um objeto de mais de uma maneira. Por exemplo, suponha que você construa uma classe que possa inicializar objetos de uma maneira padrão ou lendo informação de um arquivo. Você necessita de dois construtores, um que não tem argumentos (o construtor default) e um que toma como argumento uma `String`, que é o nome do arquivo a partir do qual as informações serão lidas. Ambos são construtores, portanto tem o mesmo nome – o nome da classe. Dessa forma, sobrecarga de métodos é essencial para permitir que se possa criar quantos construtores diferentes desejarmos. Uma vez que sobrecarga é necessária para os construtores, os projetistas de Java decidiram estender a sobrecarga de métodos para poder ser utilizada por qualquer método.

Aqui está um exemplo que mostra tanto a sobrecarga de construtores quanto de métodos ordinários:

```
// Sobrecarga.java
// Demonstracao de sobrecarga.
class Arvore {
  int altura;
  Arvore() {
    prt("Plantando uma semente ");
    altura = 0;
  }
  Arvore(int i) {
    prt("Criando nova arvore que tem "
        + i + " metros de altura ");
    altura = i;
  }
```

```
  void info() {
    prt("Arvore tem " + altura
        + " metros de altura ");
  }
  void info(String s) {
    prt(s + ": Arvore tem "
        + altura + " metros de altura ");
  }
  static void prt(String s) {
    System.out.println(s);
  }
}

public class Sobrecarga {
  public static void main(String[] args) {
    for(int i = 0; i < 5; i++) {
      Arvore t = new Arvore(i);
      t.info();
      t.info("metodo sobrecarregado ");
    }
    // Construtor sobrecarregado:
    new Arvore();
  }
}
```

Exemplo 4. 5 - Sobrecarga de Métodos

Um objeto `Arvore` pode ser criado como uma semente, com nenhum argumento, ou como uma planta crescida, com um determinada altura. Para suportar isto, existem dois construtores, um que não recebe nenhum argumento (nós chamamos este construtor de construtor default) e um que toma um inteiro representando a altura.

Você pode também desejar chamar o método info( ) de mais de uma maneira. Por exemplo, com uma *String* como argumento se você tem uma mensagem adicional para ser impressa, e sem a *String*, se você não tem mais nada a dizer.

4.2.2.1 - Distinção entre Métodos Sobrecarregados

Se os métodos tem o mesmo nome, como Java pode saber qual método invocar? Há uma regra simples: cada método sobrecarregado deve ter uma lista de argumentos diferente, isto é, o número ou os tipos dos argumentos da lista devem ser diferentes.

Se você pensar acerca deste assunto por um segundo, você chegará a conclusão que isto faz sentido: como poderia um programador dizer a diferença entre dois métodos que tem o mesmo nome, a não ser através da sua lista de argumentos?

Mesmo diferenças na ordem dos argumentos são suficientes para distinguir dois métodos:

```
// Sobrecarga baseada na ordem dos argumentos.
public class OrdemSobrecarregada {
  static void print(String s, int i) {
    System.out.println("String: " + s +
      ", int: " + i);
```

```
  }
  static void print(int i, String s) {
    System.out.println("int: " + i +
      ", String: " + s);
  }
  public static void main(String[] args) {
    print("primeira String ", 11);
    print(99, "primeiro Int ");
  }
}
```

Exemplo 4. 6 - Distinção de Métodos Sobrecarregados

Os dois métodos `print( )` têm argumentos idênticos, mas a ordem é diferente o que os torna distintos.

4.2.2.2 - Sobrecarga e Tipos Primitivos

Vimos que um valor dos tipos primitivos num pode ser automaticamente promovido de um tipo menor para um maior. Isto pode ser ligeiramente confuso quando combinado com sobrecarga. O exemplo mostrado a seguir demonstra o que ocorre quando um tipo primitivo é manipulado por um método sobrecarregado:

```
// Promocao de tipos primitivos e sobrecarga
public class PrimitivoSobrecarregado {
  static void prt(String s) {
    System.out.println(s);
  }
  void f1(char x) { prt("f1(char): "+x); }
  void f1(byte x) { prt("f1(byte): "+x); }
  void f1(short x) { prt("f1(short): "+x); }
  void f1(int x) { prt("f1(int): "+x); }
  void f1(long x) { prt("f1(long): "+x); }
  void f1(float x) { prt("f1(float): "+x); }
  void f1(double x) { prt("f1(double): "+x); }
  void f2(short x) { prt("f2(short): "+x); }
  void f2(long x) { prt("f2(long): "+x); }
  void f2(double x) { prt("f2(double): "+x); }
  void f3(double x) { prt("f3(double): "+x); }

  void testeConstVal() {
    prt("Testando com 5");
      f1(5);f2(5);f3(5);
   }
  void testeChar() {
    char x = 'x';
    prt("argumento char:");
    f1(x);f2(x);f3(x);
  }
  void testeByte() {
    byte x = 0;
    prt("argumento byte:");
    f1(x);f2(x);f3(x);
```

```
  }
  void testeFloat() {
     float x = 0;
     prt("argumento float:");
     f1(x);f2(x);f3(x);
   }
 public static void main(String[] args) {
     PrimitivoSobrecarregado p =
       new PrimitivoSobrecarregado ();
     p.testeConstVal();
     p.testeChar();
     p.testeByte();
     p.testeFloat();
   }
 }
```

Exemplo 4. 7 - Promoção de Primitivos nos Argumentos

Se observarmos a saída deste programa, veremos que o valor constante 5 é tratado como um *int*, dessa forma se existe um método sobrecarregado que toma como argumento um *int* este é utilizado. Em todos os outros casos, se você tem um tipo de dado que é menor que o argumento no método, este tipo de dado é promovido. O argumento de tipo *char* produz um efeito ligeiramente diferente, uma vez que não se encontra um encaixe perfeito este é promovido para *int*. No nosso exemplo, o *char* é promovido para *long* porque não existe `f2(int)`.

O que ocorre se o argumento é maior que o argumento esperado pelo método sobrecarregado? Uma modificação do programa exemplo mostrará a resposta:

```
// Rebaixamento de primitivos e sobrecarga.
public class Rebaixamento {
  static void prt(String s) {
    System.out.println(s);
  }
  void f1(int x) { prt("f1(int)"); }
  void f1(double x) { prt("f1(double)"); }
  void f2(int x) { prt("f2(int)"); }
  void f2(float x) { prt("f2(float)"); }
  void f3(short x) { prt("f3(short)"); }
  void f3(long x) { prt("f3(long)"); }
  void f4(byte x) { prt("f4(byte)"); }
  void f4(int x) { prt("f4(int)"); }
  void f5(char x) { prt("f5(char)"); }
  void f5(short x) { prt("f5(short)"); }
  void f6(char x) { prt("f6(char)"); }
  void f6(byte x) { prt("f6(byte)"); }
  void f7(char x) { prt("f7(char)"); }
  void testeDouble() {
    double x = 0;
    prt("argumento double:");
    f1(x);f2((float)x);f3((long)x);f4((int)x);
    f5((short)x);f6((byte)x);f7((char)x);
  }
```

```
  public static void main(String[] args) {
    Rebaixamento p = new Rebaixamento ();
    p.testDouble();
  }
}
```

Exemplo 4. 8 - Rebaixamento de Primitivos nos Argumentos

Aqui, os métodos tomam valores primitivos mais estreitos. Se seu argumento é mais amplo então você deve fazer um cast para o tipo necessário utilizando o nome em parênteses. Se você não fizer isto, o compilador emitirá uma mensagem de erro.

Você deve estar consciente que esta é uma conversão perigosa, o que significa que você pode perder informação durante o cast. Esta é a razão porque o compilador força você a explicitamente indicar a conversão.

4.2.2.3 - Sobrecarregando o Valor de Retorno

É comum se perguntar porque a sobrecarga só diferencia nomes e lista de argumentos do método. Porque não se distinguir entre métodos baseado em seu valor de retorno? Por exemplo, embora os dois métodos seguintes tenham o mesmo nome e argumentos, eles são facilmente diferenciados um do outro.

```
void f() {}
int f() {}
```

Este tipo de sobrecarga funciona bem quando o compilador pode determinar sem margem de dúvida o significado do contexto, com em `int x = f( )`. No entanto, você pode chamar um método e ignorar o valor de retorno; isto é freqüentemente referenciado como *chamando um método devido o seu efeito colateral,* uma vez que você não se preocupa com o valor de retorno do método mas quer os outros efeitos da chamada do método. Dessa forma, se você chama `f();` Java não pode determinar qual método deve ser chamado. Além disso, a leitura e o entendimento do código passa a se tornar obscura. Devido a este tipo de problema, você não pode utilizar o tipo do valor de retorno para distinguir métodos sobrecarregados.

4.2.3- Construtores Default

Como foi mencionado anteriormente, um construtor default é um construtor que não possui nenhum argumento, e é utilizado para criar "objetos padrões". Se você cria uma classe que não tem nenhum construtor, o compilador automaticamente criará um construtor default para você. Por exemplo:

```
//ConstrutorDefault.java
class Passaro {
  int i;
}
public class ConstrutorDefault {
  public static void main(String[] args) {
    Passaro nc = new Passaro(); // default!
  }
}
```

Exemplo 4. 9 - Construtor Default

Este programa cria um novo objeto e chama o construtor default, mesmo quando este não foi explicitamente definido. Sem o construtor default não existiria nenhum método para construir nosso objeto. No entanto, se você define algum construtor (com ou sem argumento), o compilador *não* produzirá um para você:

```
class Arbusto {
  Arbusto(int i) {}
  Arbusto(double d) {}
}
```

Agora, se você escreve

```
new Arbusto();
```

o compilador reclamará que não pôde encontrar um construtor que encaixe com esta chamada. Portanto, a postura do compilador java é a seguinte: uma vez que algum construtor é sempre necessário, se você não escrever nenhum, ele faz um para você. Mas se você escreveu um construtor, o compilador considera que você sabe o que está fazendo. Assim, se você não escreve um construtor default o compilador considera que você não deseja um.

4.2.4- A Palavra Reservada *this*

Se você tem dois objetos do mesmo tipo chamados `a` e `b`, você pode se perguntar como é que você pode chamar um método `f( )` para ambos os objetos:

```
class Banana { void f(int i) { /* ... */ } }
Banana a = new Banana(), b = new Banana();
a.f(1);
b.f(2);
```

Se há apenas um método chamado `f( ),` como pode este método saber se ele está sendo chamado para o objeto `a` ou `b`? Para permitir que você escreva convenientemente o código em uma sintaxe orientada a objetos na qual você manda uma mensagem para um objeto, o compilador faz parte do trabalho para você. Existe um argumento secreto passado para o método f( ), e este argumento é a referência para o objeto que está sendo manipulado. Assim os dois métodos chamados tornam-se alguma coisa como:

```
Banana.f(a,1);
Banana.f(b,2);
```

Este procedimento é interno e você não pode escrever estas expressões, mas isto dá uma idéia do que está ocorrendo.

Suponha agora que você esteja dentro de um método e deseje obter uma referência para o objeto corrente. Uma vez que referências são passadas secretamente pelo compilador, não existe um identificador para ele. No entanto, para este propósito há uma palavra chave em Java chamada *this*. Esta palavra chave somente pode ser utilizada dentro de um método e produz uma referência para o objeto para o qual a mensagem está sendo enviada. Você pode tratar esta referência como se ela fosse uma referência para um objeto qualquer.

Se você está chamando um método de sua classe dentro de outro método da sua classe, você não necessita utilizar *this*; você simplesmente chama o método. A referência atual *this* é automaticamente utilizada pelo outro método. Considere por exemplo:

```
class Manga {
  void colher() { /* ... */ }
  void comer() { colher(); /* ... */ }
}
```

Dentro de `comer( )`, você poderia dizer `this.colher( )`, mas isto não é necessário. O compilador faz isto automaticamente para você. A palavra chave *this* é utilizada somente para casos especiais em que você necessita explicitamente referenciar o objeto corrente. Por exemplo, *this* é freqüentemente utilizado nas declarações *return* quando você quer retornar uma referência para o objeto corrente.

```
//Folha.java
// Uso simples da palavra chave this.
public class Folha {
  int i = 0;
  Folha incremento() {
    i++;
    return this;
  }
  void print() {
    System.out.println("i = " + i);
  }
  public static void main(String[] args) {
    Folha x = new Folha();
     x.incremento().incremento().incremento().print()
;
  }
}
```

Exemplo 4. 10 - Uso de this

Uma vez que `incremento( )` retorna uma referência para o objeto corrente via a palavra reservada *this*, operações múltiplas podem ser facilmente executadas sob o mesmo objeto.

4.2.5- Chamando Construtores dentro de outros Construtores

Quando você escreve vários construtores para uma classe, muitas vezes você gostaria de chamar um construtor de dentro de outro construtor de modo a evitar a duplicação de código. Você pode fazer isto utilizando a palavra chave *this*.

Normalmente, quando você usa *this*, isto significa “este objeto” ou “o objeto corrente”. Em um construtor, a palavra chave *this* adquire um significado diferente quando você a emprega juntamente com uma lista de argumentos. Nestes casos, *this* faz uma chamada explícita para o construtor que casa com a lista de argumentos. Dessa forma você tem uma maneira direta de chamar outros construtores:

```
// Fabrica.java
// Chamando um construtor com "this."
public class Fabrica {
```

```
  int contFunc = 0;
  String s = new String("nula");
  Fabrica(int funcionarios) {
    contFunc = funcionarios;
    System.out.println("Fabrica(int): " + contFunc);
  }
  Fabrica(String ss) {
    System.out.println("Fabrica(String): " + ss);
    s = ss;
  }
  Fabrica(String s, int funcionarios) {
    this(funcionarios);
//! this(s);      // não pode chamar dois!
    this.s = s;   // Outro uso de "this"
    System.out.println("String & int args");
  }
  Fabrica() {
    this("ola", 47);
    System.out.println("construtor default");
  }
  void print() {
//!   this(11);   //   Nao   pode   dentro   de   nao
construtores!
    System.out.println(contFunc + " - " + s);
  }
  public static void main(String[] args) {
    Fabrica x = new Fabrica();
    x.print();
  }
}
```

Exemplo 4. 11 - Uso de this como Construtor

O construtor `Fabrica(String s, int funcionarios)` mostra que você só pode usar *this* para chamar um construtor de dentro de outro construtor uma única vez. Isto faz sentido pois senão se poderia chamar dois construtores para criar um único objeto. Adicionalmente, o construtor chamado deve ser a primeira coisa a ser feita ou você obterá uma mensagem de erro.

4.2.5.1 - O Significado de *static*

Com o conceito de *this* em mente, você pode entender mais facilmente o que significa fazer um método estático. Isto significa que não existe nenhum *this* para aquele método particular. Você não pode chamar métodos não estáticos de dentro de métodos estáticos (embora o contrário seja possível), e você pode chamar um método *static* para a própria classe sem que exista qualquer objeto. De fato, este é o uso principal para um método *static*. Isto funciona como se você estivesse criando o equivalente de uma função global. Note que Java não permite funções globais. Contudo, ao colocar um método *static* dentro de uma classe você tem acesso aos outros métodos *static* e aos campos *static*.

4.2.6- Remoção

Se você toma como ponto de partida alguma linguagem de programação onde alocação de objetos no monte (heap) é dispendioso, você pode naturalmente assumir que o esquema de Java de alocar tudo no monte é dispendioso. No entanto, o coletor de lixo tem um impacto significativo na velocidade de criação de objetos. Isto pode parecer um pouco estranho a primeira vista : "a liberação de memória afeta o processo de alocação de memória". Mas é exatamente o que ocorre em muitas máquinas virtuais Java. Dessa forma criar um objeto no monte em Java é quase tão rápido quanto criar um objeto na pilha em outras linguagens de programação.

Por exemplo, você pode pensar no monte de C++ como um loteamento do qual cada objeto apodera-se de um lote. Este lote pode tornar-se abandonado no futuro e pode, dessa forma, ser reutilizado. Em muitas máquinas virtuais Java, o monte Java é bastante diferente; este se parece mais com uma correia transportadora que se move para frente toda vez você aloca um novo objeto. Isto significa que a alocação de memória é consideravelmente rápida. O "ponteiro para o monte" é simplesmente movido para frente dentro de um território virgem. Dessa forma, isto é efetivamente o mesmo que alocação na pilha em C++.

A chave deste processo é o coletor de lixo. Na medida que o coletor de lixo desaloca a memória desocupada, ele também compacta todos os objetos no monte de modo que o ponteiro do monte aproxima-se do começo da correia transportadora e, portanto, diminui a possibilidade de uma falha por estouro de memória.

Para entender como isto funciona, você necessita ter uma melhor idéia de como os vários coletores de lixo trabalham. Uma técnica simples, mas lenta, para um coletor de lixo é a contagem de referências. Isto significa que cada objeto contem um contador de referências, e toda vez que uma referência é associada a um objeto, o contador aumenta. Toda vez que uma referência sai do escopo ou é ajustada para null, o contador de referência é decrementado. Assim, a administração do contador de referência é uma sobrecarga pequena mas constante que ocorre durante todo o tempo de vida do programa. O coletor de lixo percorre a lista de objetos e quando este encontra um contador de referência cujo valor é zero, ele libera a memória associada ao objeto em questão. Um problema com esta abordagem é o problema da referência circular entre objetos. Nesta situação, os objetos podem ter contadores de referência diferentes de zero e, no entanto, representarem lixo. Localizar tais grupos de objetos com referência mútua requer uma dose considerável de trabalho extra para o coletor de lixo. O mecanismo de contagem de referência é normalmente utilizado para explicar o mecanismo de coleta de lixo, contudo este não é utilizado em nenhuma máquina virtual Java.

Existem esquemas de coleção de lixo mais eficientes que não se baseiam em um contador de referência. Ao invés disto, estes esquemas baseiam-se na idéia de que um objeto vivo possui um caminho que conduz, em última instância, a uma referência que reside na pilha ou na área de armazenagem estática. A cadeia pode seguir através de várias camadas de objetos. Dessa forma, se você começa com a pilha e a área de armazenagem estática e caminha através de todas as referências você encontrará todos os objetos vivos. Para cada referência encontrada, deve-se seguir todas as referências existentes no objeto referenciado, e assim por diante. Cada objeto que atingido ainda está vivo. Note que o problema de grupos com referência mútua não ocorre pois estes grupos simplesmente não são encontrados, e são portanto coletados automaticamente como lixo.

Existem raras situações onde você quer ter a garantia que o coletor de lixo não entrará em funcionamento. Tais situações ocorrem quando você precisa do máximo de eficiência no tempo de resposta em um trecho crítico de seu programa. A forma como você pode minimizar esta possibilidade é chamando explicitamente o coletor de lixo antes de entrar neste trecho crítico. Dessa maneira, o coletor de lixo só atuará se for estritamente necessário desalocar espaço de memória durante a execução de seu trecho de código. Você pode chamar explicitamente o coletor de lixo através do método `gc()` da classe `System`.

4.2.7- Inicialização de Atributos

Como já vimos, Java inicializa automaticamente as variáveis membro dos tipos primitivos com valores default, caso você não as inicialize por conta própria. No caso de variáveis que são definidas localmente a um método, Java força a inicialização através da detecção de um erro em tempo de compilação quando elas não são inicializadas.

O mesmo ocorre quando as variáveis são do tipo objeto. Quando você define um atributo de uma classe como sendo uma referência para um objeto sem inicializá-lo para um novo objeto, esta referência é inicializada automaticamente para o valor especial *null* (que é uma palavra chave em Java). Este procedimento não é realizado se a variável é local aos métodos, mas se você tenta utilizar esta referência, antes dela ser associada a um objeto, também ocorrerá um erro de compilação.

O que ocorre se você quer dar um valor inicial a uma variável membro objeto? Uma maneira simples é atribuir o valor no ponto onde você define a variável na classe. Por exemplo, se `Largura` é uma classe, você pode inserir uma variável e inicializá-la da seguinte forma:

```
class Medida {
  Largura x = new Largura();
  boolean b = true;
  public Medida f ( ) { return new Medida( ) }
  public Medida g (Medida m) { return m; }
}
```

Se você não dá um valor inicial ao objeto x e tenta utilizá-lo (por exemplo, chamando um método de `Largura)`, será obtido um erro em tempo de execução e será disparada uma exceção.

Outro modo de inicializar variáveis membro é através da chamada de um método que retorne um objeto do mesmo tipo da variável:

```
class MedidaI {
  Medida i = new Medida( );
  Medida j = i.f();
}
```

O método pode ter argumentos, mas estes argumentos não podem ser outros membros da classe que ainda não tenham sido inicializados. Dessa forma, você pode fazer:

```
class MedidaII {
  Medida i = new Medida( );
  Medida j = i.f();
  Medida k = i.g(j);
```

```
}
```

Contudo, não é possível fazer:

```
class MedidaIII {
  Medida i = new Medida( );
  Medida k = i.g(j);
  Medida j = i.f();
}
```

Nesta última situação, o compilador reclama apropriadamente acerca de uma referência futura, uma vez que o problema está relacionado com a ordem de inicialização.

O construtor pode ser utilizado para executar inicialização, e isto dá a você uma maior flexibilidade, uma vez que você pode criar métodos e executar ações em tempo de execução para determinar os valores iniciais. No entanto, há uma coisa que você deve estar atento, você não está abrindo mão da inicialização automática, uma vez que esta inicialização ocorre antes que o construtor seja invocado. Por exemplo, se você diz:

```
class Contador {
  int i;
  Contador() { i = 7; }
  // ...
```

Então, i será primeiro inicializado para 0, e depois para 7. Isto é verdade com todos os tipos primitivos e com referências para objetos, incluindo as que são explicitamente inicializadas no ponto de definição. Por esta razão, o compilador não tenta forçar você a inicializar os elementos em qualquer lugar particular do construtor, uma vez que a inicialização já está garantida.

4.2.7.1 - Ordem de Inicialização

Dentro de um classe, a ordem de inicialização é determinada pela ordem de definição das variáveis dentro da classe. As definições podem estar dispersas entre as definições dos métodos da classe, mas as variáveis são inicializadas antes que qualquer método possa ser chamado, mesmo o construtor. Por exemplo:

```
//: OrdemDeInicializacao.java
// Demonstra a ordem de inicializacao.
class Rotulo {
  Rotulo(int marca) {
    System.out.println("Rotulo(" + marca + ")");
  }
}
class Cartao {
  Rotulo t1 = new rotulo(1);     //     Antes     do
construtor
  Cartao() {
    System.out.println("Cartao()");
    t3 = new Rotulo(33); // Reinicializa t3
  }
  Rotulo t2 = new Rotulo(2);     //     Depois     do
construtor
  void f() {
```

```
    System.out.println("f()");
  }
  Rotulo t3 = new Rotulo(3);     // no fim
}
public class OrdemDeInicializacao {
  public static void main(String[] args) {
    Cartao t = new Cartao();
    t.f(); // Mostra que construcao e feita  }
}
```

Exemplo 4. 12 - Ordem de Inicialização de Variáveis Membro

Na classe `Cartao`, as definições dos objetos `Rotulo` estão intencionalmente espalhadas de modo a provar que elas serão todas inicializadas antes que o construtor venha a ser invocado ou alguma outra coisa venha a ocorrer. Adicionalmente, `t3` é reinicializado dentro do construtor. A saída deste programa é:

```
Rotulo(1)
Rotulo(2)
Rotulo(3)
Cartao()
Rotulo(33)
f()
```

Dessa forma, a referência `t3` é inicializada duas vezes, uma vez antes e uma vez durante a chamada do construtor (o primeiro objeto é abandonado, sendo coletado posteriormente pelo coletor de lixo).

4.2.7.2 - Inicialização de Atributos Estáticos

Quando a variável é estática a mesma coisa ocorre; se ela é de um tipo primitivo e você não a tenha inicializado, ela obtêm o valor inicial default. Se ela é uma referência para um objeto, o valor inicializado é null, a não ser que você crie um novo objeto e associe sua referência a ele.

Se você quer colocar inicialização no ponto de definição, isto é feito da mesma forma que para variáveis não estáticas. Há, contudo, a questão de quando a memória estática é inicializada. O exemplo abaixo esclarece esta questão:

```
// InicializacaoEstatica.java
// Especificando valores iniciais em uma definicao
// de classe.
class Tigela {
  Tigela (int marca) {
    System.out.println("Tigela(" + marca + ")");
  }
  void f(int marca) {
    System.out.println("f(" + marca + ")");
  }
}
class Mesa {
  static Tigela b1 = new Tigela(1);
  Mesa() {
    System.out.println("Mesa()");
```

```
    b2.f(1);
  }
  void f2(int marca) {
    System.out.println("f2(" + marca + ")");
  }
  static Tigela b2 = new Tigela(2);
}
class Armario {
  Tigela b3 = new Tigela(3);
  static Tigela b4 = new Tigela(4);
  Armario() {
    System.out.println("Armario()");
    b4.f(2);
  }
  void f3(int marca) {
    System.out.println("f3(" + marca + ")");
  }
  static Tigela b5 = new Tigela(5);
}

public class InicializacaoEstatica {
  public static void main(String[] args) {
    System.out.println("Criando novo Armario()");
    new Armario();
    System.out.println("Criando novo Armario()");
    new Armario();
    t2.f2(1);
    t3.f3(1);
  }
  static Mesa t2 = new Mesa();
  static Armario t3 = new Armario();
}
```

Exemplo 4. 13 - Inicialização de Atributos de Classe

A saída mostra o que ocorre:

```
Tigela(1)
Tigela(2)
Mesa()
f(1)
Tigela(4)
Tigela(5)
Tigela(3)
Armario()
f(2)
Criando novo Armario ()
Tigela(3)
Armario ()
f(2)
Criando novo Armario ()
Tigela(3)
```

```
Armario ()
f(2)
f2(1)
f3(1)
```

A inicialização estática ocorre somente quando é necessária. Se você não cria um objeto Mesa e você nunca se refere a Mesa.b1 ou Mesa.b2, os estáticos Tigela b1 e b2 jamais serão criados. Além disso, eles somente são inicializados quando o primeiro objeto Mesa é criado (ou o primeiro acesso estático ocorre). Depois disso os objetos estáticos não são mais reinicializados.

A ordem de inicialização é estáticos primeiro, se eles não foram inicializados pela criação de um objeto anterior, e então objetos não estáticos.

4.2.7.3 - Bloco de Inicialização de Atributos Estáticos

Java permite que você agrupe outras inicializações estáticas dentro de uma cláusula de construção estática (algumas vezes chamado de bloco estático) em uma classe. Esta clausula tem a seguinte forma:

```
class Colher {
  static int i;
  static {
    i = 47;
  }
  // . . .
```

Isto parece ser um método, mas é apenas a palavra chave **static** seguida pelo corpo de um método. Este código, como outras inicializações estáticas, é executado apenas uma única vez, a primeira vez que você faz um objeto daquela classe ou na primeira vez que você acessa um membro estático daquela classe (mesmo se você nunca faz um objeto daquela classe). Por exemplo:

```
//EstaticoExplicito.java
// Inicializacao estatico explicito
class Copo {
  Copo(int marca) {
    System.out.println("Copo(" + marca + ")");
  }
  void f(int marca) {
    System.out.println("f(" + marca + ")");
  }
}
class Copos {
  static Copo c1;
  static Copo c2;
  static {
    c1 = new Copo(1);
    c2 = new Copo(2);
  }
  Copos() {
    System.out.println("Copos()");
  }
}
```

```
public class EstaticoExplicito {
  public static void main(String[] args) {
    System.out.println("Dentro do main()");
    Copos.c1.f(99);  // (1)
  }
  // static Copos x = new Copos();  // (2)
  // static Copos y = new Copos();  // (2)
}
```

Exemplo 4. 14 - Bloco de Inicialização Estático

O bloco inicializador *static* para `Copos` executa quando um acesso a um objeto estático `c1` ocorre na linha marcada (1), ou se a linha (1) é comentada e as linhas marcadas (2) não são comentários. Se ambas as linhas (1) e (2) são comentários, a inicialização estática para Copos nunca ocorre. Note, também, que se ambas as linhas ou uma das linhas marcadas com (2) deixam de ser comentários; a inicialização estática ocorre somente uma vez.

4.2.7.4 - Bloco de Inicialização de Atributos Não Estáticos

Java fornece uma sintaxe similar para a inicialização de variáveis membro não estáticas para cada objeto. Aqui está um exemplo:

```
//Carros.java
// Java "Inicializacao Instancia."
class Carro {
Carro(int marca) {
    System.out.println("Carro(" + marca + ")");
  }
  void f(int marca) {
    System.out.println("f(" + marca + ")");
  }
}
public class Carros {
  Carro c1;
  Carro c2;
  {
    c1 = new Carro(1);
    c2 = new Carro(2);
    System.out.println("c1 & c2 inicializados");
  }
  Carros() {
    System.out.println("Carros()");
  }
  public static void main(String[] args) {
    System.out.println("Dentro de main()");
    Carros x = new Carros();
    Carros y = new Carros();
  }
}
```

Exemplo 4. 15 - Bloco de Inicialização de Atributos Não Estáticos

A clausula inicialização de atributos não estáticos parece-se exatamente com a cláusula de inicialização exceto pela ausência de palavra chave *static*. Esta sintaxe é necessária para suportar a inicialização de classes internas anônimas.

4.2.8- Inicialização de Vetores

Um vetor é simplesmente uma seqüência de objetos ou valores primitivos, todos do mesmo tipo e empacotados juntos sob o mesmo nome de identificador. Vetores são definidos e utilizados com o operador de indexação `[ ]`. Para definir um vetor você simplesmente escreve o nome do tipo seguido de um abre e fecha colchete e do identificador do vetor.

```
int[] a1;
```

Você também pode colocar os colchetes depois do identificador pois isto produz exatamente o mesmo resultado:

```
int a1[];
```

O compilador não permite que você lhe diga qual é o tamanho do vetor. Tudo o que você tem neste ponto é uma referência para um vetor, e nenhum espaço de memória foi alocado para o vetor. Para criar espaço de memória para o vetor você deve escrever um expressão de inicialização. Tal como em outras variáveis, a inicialização de vetores pode aparecer em qualquer lugar do seu código. Além disso, pode-se utilizar também um tipo especial de inicialização no ponto onde o vetor é criado. Esta inicialização especial é um conjunto de valores colocados entre um abre ({) e fecha chaves (}). A alocação de memória será feita como se o operador *new* tivesse sido utilizado. Por exemplo:

```
int[] a1 = { 1, 2, 3, 4, 5 };
```

Uma pergunta que você pode estar se fazendo é porque definir uma referência para um vetor sem que seja acompanhada de um vetor? A resposta é porque pode-se atribuir um vetor a outro vetor em Java, dessa forma você pode dizer:

```
int[] a2;
a2 = a1;
```

Na realidade, o que você realmente fez foi copiar uma referência, como demonstrado abaixo:

```
// Vetores.java
// Vetores de primitivos.
public class Vetores {
  public static void main(String[] args) {
    int[] a1 = { 1, 2, 3, 4, 5 };
    int[] a2;
    a2 = a1;
    for(int i = 0; i < a2.length; i++)  a2[i]++;
    for(int i = 0; i < a1.length; i++)
          System.out.println("a1[" + i + "] = " +
a1[i]);
  }
}
```

Exemplo 4. 16 - Inicialização de Vetores Primtivos

Você pode notar que um valor inicial é dado ao vetor `a1`, enquanto que `a2` não recebe um valor inicial; `a1` é atribuído a `a2` posteriormente. Observe que há algo novo aqui: todos os vetores tem um membro intrínseco (se eles são um vetor de objetos ou um vetor de primitivos) que você pode consultar, mas não mudar, que lhe permite saber quantos elementos existem no vetor. Este membro é chamado `length`. Uma vez que vetores em Java, como em C e C++, começam a contagem a partir de zero, o último elemento tem índice `length - 1`. Java protege você contra o acesso fora do alcance do vetor, neste caso ocorre um erro em tempo de execução (uma exceção). É claro que, ao checar todo acesso ao vetor custa tempo de execução. Como não há maneira de contornar estes custos, os acessos a vetores podem ser uma fonte de perda de eficiência em seu programa. Contudo, na imensa maioria das aplicações, o ganho de segurança e produtividade vale o preço pago pela perda de eficiência.

O que fazer se você não sabe quantos elementos o vetor conterá quando está escrevendo o programa? Em Java, você simplesmente utiliza o operador *new* para criar os elementos no vetor.

```
//VetorNovo.java
// Criando vetores com new
import java.util.*;
public class VetorNovo {
  static Random rand = new Random();
  static int pRand(int mod) {
    return Math.abs(rand.nextInt()) % mod + 1;
  }
  public static void main(String[] args) {
    int[] a;
    a = new int[pRand(20)];
         System.out.println("tamanho de a = " +
a.length);
    for(int i = 0; i < a.length; i++)
      System.out.println("a[" + i + "] = " + a[i]);
  }
}
```

Exemplo 4. 17 - Definição Dinâmica de Tamanho do Vetor

Uma vez que o tamanho do vetor é escolhido aleatoriamente (utilizando o método `pRand()`), fica claro que a criação do vetor está ocorrendo em tempo de execução. Adicionalmente, você poderá observar na saída deste programa que os elementos de tipos primitivos do vetor são automaticamente inicializados para o valor default do tipo.

É claro que, o vetor poderia também ter sido definido e inicializado na mesma declaração:

```
int[] a = new int[pRand(20)];
```

Se você está manipulando um vetor de objetos, você terá sempre de utilizar *new* para criar os objetos do vetor. Novamente a questão das referências vêm a tona, porque o que você cria é um vetor de referências. Considere o tipo envoltório `Integer`, que é uma classe e não um tipo primitivo:

```
// VetorClasseObj.java
// Criando um vetor de objetos nao primitivos.
```

```
import java.util.*;
public class VetorClasseObj {
  static Random rand = new Random();
  static int pRand(int mod) {
    return Math.abs(rand.nextInt()) % mod + 1;
  }
  public static void main(String[] args) {
    Integer[] a = new Integer[pRand(20)];
         System.out.println("tamanho de a = " +
a.length);
    for(int i = 0; i < a.length; i++) {
      a[i] = new Integer(pRand(500));
      System.out.println("a[" + i + "] = " + a[i]);
    }
  }
}
```

Exemplo 4. 18 - Vetores de Objetos

Aqui, depois que *new* é chamado para criar o vetor, o que se tem é apenas um vetor de referências, e será assim até o momento em que a própria referência é inicializada através da criação de um novo objeto `Integer`. Se você esquecer de criar o objeto, quando tentar usar a localização vazia do vetor você obterá uma exceção em tempo de execução.

Considere a formação de objetos `String` dentro da declaração de impressão. Você pode ver que a referência para o objeto `Integer` é automaticamente convertida para produzir uma `String` representando o valor dentro do objeto.

Também é possível inicializar vetores de objetos utilizando a notação de abre e fecha chaves.

```
// InicVetor.java
public class InicVetor {
  public static void main(String[] args) {
    Integer[] a = {              // Java 1.0
      new Integer(1),
      new Integer(2),
      new Integer(3),
    };
    Integer[] b = new Integer[] {     // Java 1.1
      new Integer(1),
      new Integer(2),
      new Integer(3),
    };
  }
}
```

Exemplo 4. 19 - Inicialização de Vetores de Objetos

Isto é útil algumas vezes, contudo é mais limitado, uma vez que o tamanho do vetor é determinado em tempo de compilação. A vírgula na lista de inicializadores é opcional.

Uma outra forma de inicialização de vetores fornece uma sintaxe conveniente para criar e chamar métodos que podem produzir o mesmo efeito de uma lista de argumentos variável em C. Estas listas podem incluir uma quantidade desconhecida de argumentos bem como de tipos desconhecidos. Uma vez que todas as classes são herdeiras de uma classe raiz comum (este assunto será abordado de forma mais minuciosa posteriormente), você pode criar um método que toma um vetor de objetos quaisquer:

```
// VarArgs.java
class A { int i; }
public class VarArgs {
  static void f(Object[] x) {
    for(int i = 0; i < x.length; i++)
      System.out.println(x[i]);
  }
  public static void main(String[] args) {
    f(new Object[] {
        new Integer(47), new VarArgs(),
        new Float(3.14), new Double(11.11) });
    f(new Object[] {"um", "dois", "tres" });
    f(new Object[] {new A(), new A(), new A()});
  }
}
```

Exemplo 4. 20 - Uso de Vetor de Objetos como Parâmetro

Neste ponto, não há muito que você possa fazer com estes objetos desconhecidos, e este programa utiliza a conversão automática de `String` para fazer alguma coisa útil com cada `Object`.

4.2.9Vetores multidimensionais

Java permite que se criem vetores multidimensionais. De fato, vetores multidimensionais em java podem ser entendidos conceitualmente como sendo vetores de vetores, isto é, vetores cujos elementos são também vetores, e assim por diante.

```
// VetorMultiDim.java
// Criando vetores multidimensionais.
import java.util.*;
public class VetorMultiDim {
  static Random rand = new Random();
  static int pRand(int mod) {
    return Math.abs(rand.nextInt()) % mod + 1;
  }
  static void prt(String s) {
    System.out.println(s);
  }
  public static void main(String[] args) {
    int[][] a1 = {
      { 1, 2, 3, },
      { 4, 5, 6, },
    };
    for(int i = 0; i < a1.length; i++)
      for(int j = 0; j < a1[i].length; j++)
```

```
        prt("a1[" + i + "][" + j +
            "] = " + a1[i][j]);
    // Vetor 3-D com tamanho fixo:
    int[][][] a2 = new int[2][2][4];
    for(int i = 0; i < a2.length; i++)
      for(int j = 0; j < a2[i].length; j++)
        for(int k = 0; k < a2[i][j].length; k++)
          prt("a2[" + i + "][" + j + "][" + k +
              "] = " + a2[i][j][k]);
    // Vetor 3-D com vetores de tamanho variável:
    int[][][] a3 = new int[pRand(7)][][];
    for(int i = 0; i < a3.length; i++) {
      a3[i] = new int[pRand(5)][];
      for(int j = 0; j < a3[i].length; j++)
        a3[i][j] = new int[pRand(5)];
    }
    for(int i = 0; i < a3.length; i++)
      for(int j = 0; j < a3[i].length; j++)
        for(int k = 0; k < a3[i][j].length; k++)
          prt("a3[" + i + "][" + j + "][" + k +
              "] = " + a3[i][j][k]);
  }
}
```

Exemplo 4. 21 - Vetores Multidimensionais

O código de impressão do exemplo utiliza o atributo `length` para que possa ser independente do tamanho do vetor. O primeiro exemplo mostra um vetor multidimensional de primitivos cujo tamanho e elementos é conhecido em tempo de compilação. Usa-se a identação de abre e fecha chaves para indicar como os elementos são distribuídos no vetor multidimensional. Cada conjunto de colchetes move você dentro do próximo nível do vetor.

O segundo exemplo mostra um vetor tridimensional de tamanho fixo alocado com *new*. Neste caso o vetor inteiro é alocado contigüamente de uma única vez:

O terceiro exemplo mostra que cada vetor constituinte da matriz pode ser de qualquer tamanho. O primeiro *new* cria um vetor com um primeiro elemento de tamanho aleatório e o resto indeterminado. O segundo *new* dentro do laço *for* preenche os elementos mas deixa o terceiro índice indeterminado até que o terceiro *new* seja executado.

Você verá que a saída dos valores do vetor são automaticamente inicializados para zero qunado você não os inicializa explicitamente com um valor. Pode-se lidar com vetores de objetos não primitivos de uma forma similar.

4.3 – Encapsulamento e Ocultamento de Informação

No mundo real, um objeto pode interagir com outro sem conhecer seu funcionamento interno. Uma pessoa, por exemplo, geralmente utiliza uma televisão sem saber efetivamente qual a sua estrutura interna ou como seus mecanismos internos são ativados. Para utilizá-la, basta saber realizar algumas operações básicas, tais como ligar/desligar a TV, mudar de um canal para outro, regular volume, cor, etc. Como estas

operações produzem seus resultados, mostrando um programa na tela, não interessa ao telespectador.

O encapsulamento consiste na separação dos aspectos externos de um objeto, acessíveis por outros objetos, de seus detalhes internos de implementação, que ficam ocultos para os demais objetos. A interface de comunicação de um objeto deve ser definida de forma a revelar o menos possível sobre o seu funcionamento interno.

Por exemplo, para usar um carro, uma pessoa não precisa conhecer sua estrutura interna (motor, caixa de marcha, etc...), nem tão pouco como se dá a implementação de seus métodos. Sabe-se que é necessário ligar o carro, mas não é preciso saber como esta operação é implementada. Assim, sobre carros, um motorista precisa conhecer apenas as operações que permite utilizá-lo, o que chamamos de *interface do objet*o, o que inclui a ativação de operações, tais como ligar, mudar as marchas, acelerar, frear, etc..., e não como essas operações são de fato implementadas.

A principal motivação para o encapsulamento é facilitar a reutilização de objetos e facilitar a manutenção dos sistemas. Um encapsulamento bem feito pode servir de base para restringir as partes do projeto que necessitam ser alteradas quando alguma modificação é necessária. Por exemplo, uma operação pode ter sido implementada de maneira ineficiente e, portanto, pode ser necessário escolher um novo algoritmo. Se a operação está encapsulada, apenas a classe que a define necessita ser modificada, garantindo assim que os códigos dos clientes da classe não precisam ser alterados.

4.3.1– Pacotes

Um pacote *(packag*e) em Java nada mais é do que um conjunto de classes relacionadas construídas com um certo propósito. O *package* é o mecanismo fornecido por Java para implementar a idéia das bibliotecas (ferramenta muito comum em outras linguagens de programação).

Por exemplo, a própria linguagem Java é distribuída juntamente com vários pacotes de sua biblioteca padrão, dentre eles `java.lang`, `java.util`, `java.net` e vários outros. Os pacotes padrões de Java são exemplos de pacotes hierárquicos. Assim como você possui subdiretórios no seu disco rígido, você também pode organizar seus pacotes em níveis hierárquicos. Por exemplo, todos os pacotes padrões da linguagem Java citados anteriormente fazem parte do pacote `java`.

Uma biblioteca em Java é uma coleção de arquivos **.class**. Em geral, cada arquivo **.java** conterá uma classe pública (você não é obrigado a ter uma classe pública por arquivo, mas esta é a situação mais comum). Se quisermos garantir que estes arquivos façam parte de um mesmo grupo, devemos inseri-los em um mesmo pacote.

Ao se criar um pacote, é necessário especificar o nome do pacote nos arquivos fonte onde são descritas as classes do pacote. Esta especificação com o nome do pacote deve ser a primeira linha não comentada do arquivo fonte que define a classe. O exemplo a seguir mostra como definir uma classe chamada `MinhaClasse` no arquivo `MinhaClasse.Java` pertencente ao pacote `meupacote`.

```
package meupacote;
public class MinhaClasse {
. . .
}
```

Caso se deseje usar `MinhaClasse` ou qualquer outra classe pública do pacote `meupacote` no desenvolvimento de qualquer outra classe externa ao pacote em questão, deve-se usar o comando `import` seguido do nome do pacote ou usar sempre o nome do pacote seguido do nome da classe durante o corpo do programa, como nos exemplos seguintes:

```
// usando import
import meupacote.*;
MinhaClasse m = new MinhaClasse();

// mesma coisa sem import
meupacote.MinhaClasse m = new meupacote.MinhaClasse();
```

Note que o uso do comando `import` torna o código mais limpo. O comando *import meupacote.** faz com que todas as classes do pacote `meupacote` possam ser usadas naquele arquivo. Caso se deseje usar apenas a classe `MinhaClasse`, deste mesmo pacote, podemos também especificar o nome completo `meupacote.MinhaClasse` juntamente com o comando `import` como no exemplo seguinte:

```
import meupacote.MinhaClasse;
```

Neste caso, somente a classe `MinhaClasse` poderá ser usada sem nenhuma restrição durante o programa. As demais classes do pacote não serão acessíveis no arquivo que inclui o *import*.

Você pode ainda estar se perguntando quais as razões de se incluir o mecanismo de pacote em Java. Os principais motivos para isto são:

- Você e outros programadores podem facilmente determinar qual conjunto de classes são relacionadas.
- Você e outros programadores podem facilmente determinar onde estão localizadas as classes relacionadas a um determinado assunto.
- Os nomes das suas classes não conflitarão com os nomes de classes em outros pacotes porque cada pacote estabele um novo espaço de nomes para as classes definidas lá. Como vimos nesta seção, o nome do pacote se torna um prefixo para o nomes das classes definidas dentro dele.
- Pode-se estabelecer políticas de acesso sobre as classes do pacote (isso será visto em detalhe em uma seção posterior ainda neste capítulo).

4.3.1.1 – Nomes de Pacotes

Com programadores de todo o mundo criando bibliotecas em Java, é bastante razoável acreditar que dois programadores usarão o mesmo nome para duas classes diferentes. De fato, isso é algo que acontece. Como Java usa o nome do pacote como prefixo dos nomes das classes, a maioria das colisões podem ser tratadas através do uso do nome completo (incluindo o nome do pacote) da classe.

Isso funciona bem a menos que dois programadores usem o mesmo nome para seus pacotes. O que se pode fazer nessas situações? Usar uma **convenção**. Os projetistas de Java sugerem como convenção que os programadores utilizem o nome invertido de seu domínio na Internet (por exemplo: `br.ufes.inf`) como a parte inicial dos nomes de seus pacotes. Como os domínios da Internet são garantidamente únicos, não haveria a

possibilidade de colisão entre pacotes de domínios distintos. Colisões de nomes que possam ocorrer dentro de um domínio devem ser resolvidos através da adoção de convenções particulares pela comunidade que compartilha o domínio.

4.3.1.2 – Localização dos Pacotes

A especificação da linguagem Java não criou um padrão que defina como os compiladores ou os ambientes de desenvolvimento devem organizar o código fonte de seus usuários. Assim, esta definição fica a cargo dos projetistas do produto. O compilador javac do JDK (distribuído pela Sun) e a maioria dos outros compiladores Java utilizam o sistema hierárquico de arquivos para armazenar os arquivos **.java** e **.class** dos pacotes. Ao invés de ter de se criar uma estrutura de diretórios para armazenar os arquivos de cada pacote, pode-se alternativamente, utilizar arquivos **.zip** ou **.jar**. Obviamente, a estrutura de diretórios deve estar presente dentro destes arquivos. Os arquivos fonte devem ser colocados nos diretórios cujos nomes correspondem ao nome do pacote onde a classe foi definida. Por exemplo, todos os arquivos do pacote br.ufes.inf.util devem estar localizados em um subdiretório denominado br\ufes\inf\util (br/ufes/inf/util no Unix).

Ao se criar um pacote, é responsabilidade do programador criar a estrutura de diretórios correspondentes ao pacote e colocar lá os arquivos compilados (.class). Por exemplo, ao compilar um arquivo-fonte que inicia com a linha

```
package acm.util;
```

O programador deve colocar o arquivo resultante da compilação (.class) no subdiretório acm\util. O compilador não realizará esta tarefa.

Estes subdiretórios não precisam partir diretamente do diretório raiz; eles podem partir de qualquer diretório presente na trilha de classes ou mesmo do diretório em que a aplicação que usa estes pacotes está localizada.

Você pode estar se perguntando o porquê dessa preocupação toda com diretórios. A razão para isto é que as ferramentas do JDK precisam saber como encontrar todas as classes usadas por seu programa. Quando o compilador encontra uma nova classe usada por seu programa, ele necessita achar o arquivo **.class** (ou **.java**) desta classe de modo a poder verificar se existem conflitos de nomes, fazer checagem de tipos, etc. Similarmente, quando o interpretador encontra uma nova classe, enquanto executa seu programa, ele necessita achar o arquivo **.class** da classe para executar o código correspondente. Tanto o compilador quanto o interpretador procuram pelas classes em cada diretorio ou arquivo **.zip** ou **.jar** listados na sua trilha de classes. Normalmente, a trilha de classes é definida pela variável do sistema CLASSPATH.

Cada diretório listado na trilha de classes é um ponto de partida para a busca dos arquivos de um pacote. A partir deste diretório, a ferramenta do JDK pode construir o resto da trilha baseando-se no nome do pacote e da classe. Por default, a ferramenta do JDK busca no diretório corrente e no arquivo que contém os arquivos das classes do JDK. Em outras palavras, o diretório corrente e os arquivos de classes do JDK estão automaticamente em sua trilha de classes. A maior parte das classes podem ser achadas nestas duas localizações. Contudo, em alguns casos, você necessitará definir sua trilha de classes. Portanto, se for necessário, você pode mudar sua trilha de classes. Existem dois modos de fazer isto:

1. Redefinir a variável de sistema CLASSPATH (não recomendado).

2. Usar a opção -classpath quando invocar a ferramenta do JDK.

Não se recomenda a alteração da variável CLASSPATH porque a alteração pode perdurar. É fácil esquecer disto e, um dia, seu programa pode não funcionar porque a ferramenta do JDK carrega uma classe antiga ao invés da que você deseja. A segunda opção é preferida porque ela não modifica definitivament a variável do sistema. Veja como ela pode ser usada.

UNIX:

```
javac -classpath .:~/classes Trilha.java
```

Windows 95/NT:

```
C:> javac -classpath .;C:\classes Trilha.java
```

Quando você especifica a trilha de classes desta maneira, você deve incluir explicitamente o diretório corrente. Além disso, a ordem de entradas na trilha de classes é importante. Quando a ferramenta JDK está procurando por uma classe, ela procura as entradas na sua trilha de classes na ordem onde são encontradas até encontrar a classe com o nome correto. Ao encontrar o primeiro nome de classe que casa, a procura é interrompida. Isto pode causar problemas se tivermos duas classes com o mesmo nome em diferentes trilhas.

4.3.1.3 – Compilação Automática

A primeira vez que um objeto de uma classe importada é criado (ou se acessa um membro estático de uma classe), o compilador busca o arquivo **.class** de mesmo nome (se o objeto é da classe **X**, ele procurará o arquivo **X.class**) no diretório apropriado. Se neste diretório, ele encontra apenas o arquivo **X.class**, ele será usado. Por outro lado, se ele encontra apenas o arquivo **X.java** ele o compilará também. Contudo, se ambos arquivos **X.class** e **X.java** se encontram no diretório, ele primeiro compara a data e horário da última atualização dos dois arquivos. Se o arquivo **.java** é mais recente que o **.class**, ele irá automaticamente recompilar o arquivo **.java** para gerar um **.class** atualizado. Se a classe procurada não está em um arquivo **.java** do mesmo nome da classe, este comportamento não ocorrerá.

4.3.1.4 – O Pacote Padrão

Você ficaria surpreso ao descobrir que o código abaixo é compilável, mesmo isto parecendo não ser possível a primeira vista:

```
// Bolo.java
class Bolo {
 public static void main(String[] args) {
 Torta x = new Torta();
 x.f();
 }
}
// Torta.java
class Torta {
 void f() { System.out.println("Torta.f()"); }
}
```

Exemplo 4. 22 - Pacote Padrão

Inicialmente estes dois arquivos podem parecer completamente independentes, mas o Bolo é capaz de criar um objeto Torta e chamar o método f()! Você provavelmente acharia que Torta e f() não seriam acessíveis pela classe Bolo. A razão pela qual eles são acessíveis por Bolo é que ambos estão localizados no mesmo diretório e não possuem um nome de pacote explícito. Nesta situação, a linguagem Java trata estes arquivos como parte implícita do "pacote padrão" do diretório em que eles se localizam, o que permite acessar os membros não privados de todos os arquivos contidos neste diretório que não tiverem um nome de pacote associado.

4.3.2– Especificadores de Acesso

Em Java, a determinação da acessibilidade dos membros de uma classe é feita pelos especificadores de acesso, ou seja, por palavras reservadas da linguagem que definem o grau de encapsulamento exibido por uma classe e seus elementos. Estas palavras são especificações usadas para se restringir o acesso as declarações pertencentes a uma classe. Isto pode ser feito através do uso das palavras reservadas *public* (público), *private* (privado), e *protected* (protegido). Em Java, existe uma forma adicional de especificação de acesso que não é associada ao uso de uma palavra reservada, isto é, quando não se usa nenhum especificador de acesso na definição do membro, assume-se a especificação de acesso *friendly* (amigo).

Quando usados, os especificadores de acesso *public, protected* e *private* devem ser inseridos na frente das definições de cada membro de suas classes, sejam estes atributos ou métodos. Cada especificador de acesso controla o acesso de cada uma destas definições em particular. Note que há uma diferença entre os especificadores de acesso de Java e de C++ (onde cada especificador de acesso controla todas as definições dos membros subseqüentes da classe até que um outro especificador de acesso apareça).

4.3.2.1 – Membros Públicos

Os membros de classe declarados como *public* podem ser acessados por qualquer classe. O exemplo abaixo ilustra o uso do especificador de acesso *public*:

```
// Biscoito.java
// criação da biblioteca
package sobremesa;
public class Biscoito {
 public Biscoito() {
  System.out.println("construtor do biscoito");
 }
 public void calorias() {
  System.out.println("MUITAS!");
 }
 void mordida() {
  System.out.println("mordida!");
 }
}
// Jantar.java
// uso da biblioteca sobremesa
import sobremesa.*;
public class Jantar {
 public Jantar() {
  System.out.println("Construtor do Jantar");
```

```
 }
 public static void main(String[] args) {
  Biscoito x = new Biscoito();
  x.calorias();
  // x.mordida(); // não pode ser acessado!
 }
}
```

Exemplo 4. 23 - Membros Públicos

No exemplo acima, é possível criarmos um objeto `Biscoito`, visto que o método construtor e a classe são públicos (o conceito de classe pública será melhor abordado posteriormente). Pela mesma razão, é possível utilizar o método `calorias()`. Entretanto, o método `mordida()` não pode ser acessado pela classe `Jantar` visto que `mordida()` foi especificada como amigo, sendo assim acessada apenas pelas classes do pacote sobremesa.

4.3.2.2 - Membros Privados

Já os membros declarados como *private* só podem ser acessados pelas classes que os definem. O exemplo seguinte ilustra o uso do especificador de acesso *private*:

```
// Sundae.java
class Sundae {
 private Sundae() { }
 public static Sundae fazerUmSundae() {
  return new Sundae();
 }
}
// Sorvete.java
public class Sorvete {
 public static void main(String[] args) {
  //! Sundae x = new Sundae();
  Sundae x = Sundae.fazerUmSundae();
 }
}
```

Exemplo 4. 24 - Membros Privados

O código acima mostra como o especificador *private* pode ser útil: você pode querer controlar como um objeto é criado e impedir que os construtores de uma classe sejam manipulados diretamente. No exemplo acima, não é possível a criação de um objeto `Sundae` através do uso de seus construtores (declarados como *private*). Em vez disso deve-se fazer uma chamada ao método `fazerUmSundae()` que criará o objeto.

4.3.2.3 – Membros Amigos

E se não for informado nenhum especificador de acesso na declaração de algum membro de uma classe? O especificador de acesso padrão de Java não possui nenhuma palavra-chave associada, mas é comumente referenciado como "*friendly*" ou "*package*". Isso significa que todas as outras classes do pacote em questão têm acesso aos membros

amigáveis desta classe, mas para todas as outras classes que não pertençam a este pacote os membros da classe se comportarão como *private*.

O especificador "friendly" permite que classes relacionadas sejam agrupadas em pacotes, facilitando assim a interação entre elas. Portanto, ao armazenarmos classes num mesmo pacote, garantimos o acesso mútuo a seus membros "*friendly*" o que acarreta em uma pequena quebra do encapsulamento. Isto faz sentido se considerarmos que os pacotes geralmente são escritos pelo mesmo programador ou pela mesma equipe de programadores. Note que no final das contas o encapsulamento do pacote é mantido e as aplicações que usarão este pacote só terão acesso aos seus membros públicos. O exemplo seguinte mostra o uso do especificador de acesso amigo:

```
// Chocolate.java
package sobremesa;
public class Chocolate {
 public Chocolate() { }
 void esquentarChocolate() {
  System.out.println("Esquentando chocolate...");
 }
}
// Brigadeiro.java
package sobremesa;
public class Brigadeiro {
 public Brigadeiro() { }
 void fazerBrigadeiro() {
  Chocolate choc = new Chocolate();
  choc.esquentarChocolate(); // amigo
  System.out.println("Fazendo Brigadeiro...");
 }
}
```

Exemplo 4. 25 - Membros Amigos

No exemplo acima, como Chocolate e Brigadeiro pertencem ao mesmo pacote sobremesa, é permitido que ambas tenham acesso aos membros declarados como amigos de todas as classes contidas no pacote sobremesa. Portanto, Brigadeiro pode criar objetos de classe Chocolate (que foi declarada como pública) e também poderá fazer uso do método esquentarChocolate() (especificado como amigo).

4.3.2.4 – Membros Protegidos

O especificador de acesso *protected* está associado à idéia de herança, que será melhor abordada no próximo capítulo. Por enquanto, imagine herança como sendo a capacidade de criar uma nova classe a partir da inclusão de novos atributos ou métodos a uma classe existente. De fato, vocês verão que herança é um pouco mais que isso, visto que o comportamento dos métodos da classe usada como base também podem ser alterados na nova classe. Para fazer uma classe herdar de uma classe base, usamos a palavra reservada *extends*:

```
class Chefe extends Empregado {
...
```

Caso um novo pacote seja criado e você tente fazer uma classe deste novo pacote herdar de uma classe localizada em um outro pacote, os únicos membros herdados aos quais

você terá acesso serão os membros especificados como *public* no pacote original. Claro que se as duas classes estiverem localizadas no mesmo pacote, o acesso aos membros friendly é mantido.

Em algumas situações como a mencionada no parágrafo anterior pode ser interessante que uma classe tenha acesso aos atributos ou métodos herdados de uma outra classe sem que os mesmos precisem ser públicos (acessíveis a todos). Isto é o que o especificador de acesso *protected* faz. O exemplo seguinte ilustra o uso do especificador de acesso *protected*:

```
// Biscoito.java
package sobremesa;
public class Biscoito {
 public Biscoito() {
  System.out.println("construtor do biscoito");
 }
 protected void mordida() {
  System.out.println("mordida!");
 }
}
// BiscoitoMaizena.java
package doce;
import sobremesa.*;
public class BiscoitoMaizena extends Biscoito {
 public BiscoitoMaizena () {
  System.out.println("construtor de maizena");
 }
 public static void main(String[] args) {
  BiscoitoMaizena x = new BiscoitoMaizena();
  x.mordida(); // pode ser acessado!
 }
}
```

Exemplo 4. 26 - Membros Protegidos

Uma das características mais interessantes da herança é que o método `mordida()` existe tanto na classe `Biscoito` como também nas classes que herdam de `Biscoito`. Mas, se `mordida()` for especificado como privado da classe `Biscoito` ou como amigo do pacote `sobremesa`, ele não será acessível pela classe `BiscoitoMaizena`. Claro que se ele fosse declarado como público, poderia ser acessado não somente por `BiscoitoMaizena`, mas também por todas as outras classes. Contudo, isso pode não ser desejado.

No exemplo, o método `mordida()` foi declarado como *protected.* Assim, ele continua se comportando como amigo dentro do pacote `sobremesa`, mas também torna-se acessível a qualquer classe que herde da classe `Biscoito`. Note que `mordida()` não é visível para as classes externas ao pacote `sobremesa`, com exceção das que forem subclasse de `Biscoito`.

4.3.2.5 – Resumo de Especificadores de Acesso

A tabela 4.3 mostra como funcionam os especificadores de acesso de membros de classes em Java.

Acesso	Público *public*	Protegido *protected*	Amigo -	Privado *private*
Mesma Classe	Sim	Sim	Sim	Sim
Classes no Mesmo Pacote	Sim	Sim	Sim	Não
Subclasses em Pacotes Diferentes	Sim	Sim	Não	Não
Não Subclasses em Pacotes Diferentes	Sim	Não	Não	Não

Tabela 4. 2 - Especificadores de Acesso

4.3.3– Interface e Implementação

Como vimos até agora, as únicas formas de se garantir o acesso aos membros de uma classe são:

- Tornar o membro público, o que o torna acessível por qualquer classe.
- Tornar o membro "amigo" (não informando o especificador de acesso) e inserir a outra classe no mesmo pacote. Note que as demais classes do pacote também terão acesso a este membro.
- Uma classe herdada pode acessar, além dos membros públicos, os membros definidos como protegidos. O mesmo não é válido para os membros privados. Os membros amigos só poderão ser acessados caso as duas classes pertençam ao mesmo pacote.
- Uma classe pode ainda prover métodos públicos para fornecer acesso a seus membros inacessíveis externamente (também conhecidos como métodos "get/set"). Estes métodos definem a interface da classe. Caso seja necessário alterar a classe internamente, a preservação desta interface garante a compatibilidade desta classe com as outras classes que com ela se relacionam.

Especificadores de acesso podem estabelecer o que os programas clientes podem e o que eles não podem acessar em uma classe. Com isso, podemos mudar a estrutura interna de uma classe sem perder a compatibilidade com os programas clientes. Claro

que isto só poderá ser feito se a classe em questão tiver suas estruturas internas protegidas e a mesma possuir uma interface consistente de comunicação com outras classes.

Este encapsulamento é freqüentemente conseguido através do **ocultamento de informação**, isto é, escondendo detalhes que não contribuem para suas características essenciais. Tipicamente, em um sistema orientado a objetos, a estrutura de um objeto, e a implementação de seus métodos, são encapsuladas. Observe no exemplo seguinte como a estrutura interna da classe `Pilha` fica protegida dos programas clientes:

```
import java.util.*;
public class Pilha {
 private Vector estrutura;
 public Pilha() {
  estrutura = new Vector(10,10);
 }
 public void empilha(Object obj) {
  estrutura.add(obj);
 }
 public Object desempilha() {
  Object obj = estrutura.get(estrutura.size()-1);
  estrutura.remove(estrutura.size()-1);
  return obj;
 }
}
```

Exemplo 4. 27 - Ocultamento de Informação

Observe que a classe `Pilha` fornece para os programas usuários a função construtora e os métodos `empilha()` e `desempilha()`. A estrutura interna de armazenamento, neste caso um vetor (`Vector`), foi protegida e não pode ser acessada diretamente pelos programas usuários, que passam a ser obrigados a usar os métodos `empilha()` e `desempilha()` para armazenar ou remover objetos da pilha.

Caso a estrutura interna da classe `Pilha` precisasse ser modificada para atender a novos requisitos do problema (como desempenho ou flexibilidade, por exemplo), poderíamos fazer essas alterações sem que a classe `Pilha` perdesse a compatibilidade com os programas clientes que fazem uso de seus métodos públicos. A seguir temos um exemplo da classe `Pilha` modificada, usando agora usa uma lista encadeada (`LinkedList`) como estrutura de armazenamento dos objetos:

```
import java.util.*;
public class Pilha {
 private LinkedList estrutura;
 public Pilha() {
   estrutura = new LinkedList();
 }
 public void empilha(Object obj) {
   estrutura.addFirst(obj);
 }
 public Object desempilha() {
   return estrutura.removeFirst();
```

```
  }
}
```

Exemplo 4. 28 - Alteração da Implementação

Note que, apesar de terem ocorrido mudanças na estrutura interna da classe `Pilha`, a interface de comunicação com os programas usuários foi mantida, portanto a classe continua sendo compatível com os mesmos. Note também que a classe `Pilha` é usada pelos programas usuários sem que os mesmos tenham conhecimento sobre a estrutura interna desta classe.

4.3.4– Controlando o acesso às Classes

Em Java, especificadores de acesso também podem ser usados para determinar que classes de um pacote poderão ser usadas por classes externas ao pacote. Se você quiser que uma classe seja acessível aos programas clientes, então a palavra `public` deve preceder a palavra `class` na definição desta classe, como no código abaixo:

```
package sobremesa;
public class Biscoito {
```

Como `Biscoito` é uma classe do pacote `sobremesa`, qualquer programa cliente terá acesso à classe `Biscoito` ao usar o comando `import`, como mostrado abaixo:

```
import sobremesa.Biscoito;
```

ou

```
import sobremesa.*;
```

Entretanto, algumas considerações devem ser analisadas:

- Pode existir no máximo uma classe pública por arquivo fonte. A idéia é que cada arquivo tenha uma única interface representada pela classe pública. O arquivo pode ter várias classes amigas, mas no caso de ocorrer mais de uma classe pública, o compilador exibirá uma mensagem de erro.
- O nome da classe pública do arquivo deve ser ter o mesmo nome que o arquivo, incluindo aí letras maiúsculas e minúsculas. Portanto para a classe Biscoito, o nome do arquivo deve ser Biscoito.java e não biscoito.java. Novamente se o nome do arquivo não for o mesmo que o da classe pública o compilador exibirá uma mensagem de erro.
- É possível, porém não muito usual, termos um arquivo sem nenhuma classe pública. Neste caso o arquivo poderá ter qualquer nome.

4.4 - Exercícios

1. Crie uma classe com um construtor default que imprime uma mensagem. Crie um objeto desta classe.
2. Adicione um construtor sobrecarregado ao exercício 1 que toma uma string como argumento e imprime uma mensagem com esta string. Crie um objeto desta classe.

3. Crie um vetor de objetos da classe criada no exercício 2, mas não crie realmente os objetos para atribuir ao vetor. Verifique se as mensagens de inicialização das chamadas dos construtores são impressas quando você executa o programa
4. Complete o exercício 3 criando os objetos e os atribuindo ao vetor.
5. Crie uma classe com atributos e métodos públicos, privados, protegidos e amigos. Crie um objeto dessa classe e veja quais os tipos de mensagem o compilador apresenta quando você tentar acessar todos os membros da classe.
6. Crie uma classe Data com os seguintes construtores:
   6.1. Default (inicializa a data com 01/01/0001) .
   6.2. Recebe três inteiros correspondendo a dia, mês e ano.
   6.3. Recebe uma string no formato "dd/mm/aaaa".
   6.4. Recebe um inteiro correspondendo ao total de dias da data desde a data inicial 01/01/0001.
7. Faça um programa que use a classe Data para:
   7.1. Ler uma data qualquer e dizer se ela é válida.
   7.2. Ler uma data qualquer e dizer se é seu aniversário.
   7.3. Ler uma data qualquer e indicar se o ano é bissexto.
   7.4. Ler uma data qualquer e indicar o número total de dias que ela corresponde.
   7.5. Ler uma data qualquer e indicar o número de semanas que ela corresponde.
   7.6. Ler uma data qualquer e indicar o número de meses que ela corresponde.
   7.7. Ler uma data qualquer e indicar o número de anos que ela corresponde.
8. Faça um programa que leia a data de hoje e a data de seu nascimento e diga quantos anos, meses e dias de vida você tem.
9. Faça um programa que leia uma data no formato dd/mm/aaaa e a imprima no formato mm/dd/aa.
10. Crie uma classe Pessoa que contenha o nome e a data de nascimento de uma pessoa.
11. Crie uma classe Aniversariantes e faça um programa que use um vetor e a classe Pessoa para armazenar os nomes e datas de nascimento de cada pessoa de nossa sala de aula. Em seguida, o programa deve ler a data do dia e responder se esta data corresponde ao aniversário de alguém. Se for este o caso, o programa deve dizer quem são os aniversariantes.
12. Coloque as classes Data e Pessoa em um pacote chamado br.inf.ufes.util. Lembre-se de incluir a instrução package nos arquivos Data.java e Pessoa.java, de recompilá-los, de criar a estrutura de diretórios a partir de seu diretório corrente, e mover para lá os novos arquivos Data.class e Pessoa.class. Altere o programa que verifica os aniversariantes no seu diretório corrente para agora importar o pacote br.ufes.inf.util. Recompile-o. Qual o problema que ocorre? Observe que são criados novos arquivos Data.class e Pessoa.class no diretório corrente. Remova-os. Mude os arquivos Data.java e Pessoa.java para um diretório diferente do corrente (crie um diretório auxiliar dentro do corrente e os coloque lá). Recompile. O que acontece?
13. Agora, mova os arquivos Data.java e Pessoa.java do diretório auxiliar para o diretório do pacote, onde estão os arquivos Data.class e Pessoa.class. Abra o arquivo Pessoa.java e inclua uma nova classe não pública chamada Xpto que contém apenas um campo inteiro. Volte ao diretório corrente e recompile o arquivo Arnivesariantes.java. Volte ao diretório do pacote e liste os arquivos existentes. Observe que foi criado um novo arquivo chamado Xpto.class. Porque isso ocorreu?

4.5 - Trabalho

Crie uma classe chamada ListaPessoas para armazenar uma coleção de Pessoas. Além dos métodos construtores, esta classe deve possuir os métodos comuns associados a listas (tais como, inclusão, obtenção de elemento, remoção, verifica se lista é vazia, etc.). Use um vetor para armazenar internamente os objetos Pessoa. Coloque essa classe no pacote util. Refaça o programa dos aniversariantes para utilizar agora a classe ListaVetorPessoas ao invés do vetor.

Reimplemente agora a classe ListaPessoas usando encadeamento dinâmico. É importante manter a mesma interface da classe ListaPessoas. O que é necessário fazer para que o programa dos aniversariantes utilize agora a nova implementação? Execute o programa aniversariantes com a nova implementação de ListaPessoas.