Objetos e Classes

Introdução à Programação Orientada a Objetos

A POO é o paradigma de programação predominante dos dias atuais, tendo substituído as técnicas de programação baseadas em procedimentos, “estruturadas”, que foram desenvolvidas nos anos 70. A linguagem Java é totalmente orientada a objetos e é impossível programá-la no estilo de procedimentos com o qual talvez você esteja bem acostumado.

Vamos começar com uma pergunta que, aparentemente, parece não ter nada a ver com programação. Como empresas como Compaq, Dell, Gateway, Micron Technologies e outras principais fabricantes de computadores pessoais tornaram-se tão grandes tão rapidamente? A maior parte das pessoas haverá de dizer provavelmente que elas fizeram bons computadores de um modo geral e que os venderam a um bom preço numa era em que a demanda por computadores pessoais havia disparado. Mas, vamos aprofundarmo-nos mais um pouco e pensar, como elas foram capazes de fabricar tantos modelos tão rapidamente, respondendo às alterações que estavam acontecendo tão rápido?

Bem, uma grande parte da resposta está em que essas empresas redistribuíram um bocado de trabalho. Elas compravam componentes de fabricantes conhecidos e depois os montavam. Elas freqüentemente não investiam tempo nem dinheiro no projeto e na elaboração de fontes de alimentação, unidades de disco, placas mão e outros componentes. Isso tornou possível para essas empresas produzir um produto e fazer modificações rapidamente por menos custo do que se tivessem feito toda a engenharia por si mesmos.

O que os fabricantes de computadores pessoais estava comprando era “funcionalidade já pronta”. Por exemplo, quando eles compravam uma fonte de alimentação, eles estavam comprando algo com certas propriedades (tamanho, forma, etc.) e uma certa funcionalidade (saída controlada de alimentação, energia disponível etc.). A (antiga) Compac deixou de projetar engenharia de todas as peças de suas máquinas e passou a comprar muitas partes, ela teve um aumento significativo nos lucros.

A POO nasceu da mesma idéia. Um programa é feito de objetos com certas propriedades e operações que os objetos podem realizar. O estado atual pode mudar com o passar do tempo, mas você vai depender sempre de que os objetos não interajam entre si de formas não documentadas. Se você vai elaborar ou comprar um objeto vai depender de orçamento ou de prazo. Porém, basicamente, enquanto os objetos atenderem às especificações esperadas, não há por que se preocupar em saber como a funcionalidade foi implementada. Na POO, você só se importa com o que o objeto expõe. Assim da mesma forma os fabricantes de cópias não se importam com o que está dentro de uma fonte de alimentação desde que ela faça o que querem. A maioria dos programadores Java não quer saber como o componente clipe de áudio, por exemplo, foi implementado, desde que faça o que é desejado.

A programação tradicional estruturada consiste em projetar um conjunto de funções para resolver um problema.(Essas funções são geralmente chamadas de algoritmos). O passo seguinte convencional é encontrar a forma apropriada de armazenar os dados. É por isso que o criador do Pascal original, Niklaus Wirth, chamou seu famoso livro de Algoritmos + Estruturas de Dados = Programas (Prentice Hall, 1975). Observe que no título do livro de Wirth, os algoritmos vinham antes e as estruturadas de dados depois. Isso imita a forma com que os programadores trabalhavam naqueles tempos. Primeiro,

decidia-se como manipular os dados; depois, que estrutura impor aos dados a fim de tornar mais fácil o processamento. A POO inverte essa ordem e coloca as estruturas de dados em primeiro lugar, examinando depois os algoritmos que vão processar os dados.

A chave para ser mais produtivo em POO é tornar cada objeto responsável pela realização de um conjunto de tarefas relacionadas. Se um objeto depender de uma tarefa que não seja de sua responsabilidade, ele vai precisar ter acesso a um objeto cujas responsabilidades incluem essa tarefa. O primeiro objeto então pede ao segundo objeto para realizar a tarefa. Isso é feito com uma versão mais generalizada da chamada da função que você conhece na programação por procedimentos. (Lembre-se de que em Java essas chamadas de funções são geralmente chamadas de chamadas a métodos). No jargão da POO, há clientes que enviam mensagens a objetos servidores.

Em particular, um objeto nunca deveria manipular diretamente os dados internos de outro objeto. Toda a comunicação deve ser através de “mensagens”, ou seja, chamadas a métodos. Ao projetar seus objetos para lidar com todas as mensagens apropriadas e manipular seus dados internamente, maximiza-se a reutilização, reduz-se a dependência da dados e minimiza-se o tempo de depuração.

Evidentemente, assim como ocorre com os módulos de uma linguagem orientada a procedimentos, não se quer que um objeto individual faça coisas demais de uma só vez. Tanto o projeto quanto a depuração são simplificados quando se formam objetos pequenos que realizam algumas poucas tarefas em vez de objetos enormes com dados internos extremamente complexos, com centenas de funções para manipular os dados.

O vocabulário da POO

Agora, é necessário entender a terminologia da POO para poder prosseguir. O termo mais importante é *classe*. Uma classe é geralmente descrita como o modelo ou a forma a partir da qual um objeto é criado. Isso leva ao modo padrão de pensar sobre as classes: como sendo as formas com as quais se fazem biscoitos. Os objetos são os biscoitos. A “massa do biscoito”, na forma de memória, também terá de ser providenciada. A linguagem Java é ótima para esconder as etapas da “preparação da massa dos biscoitos”. É só usar a palavra-chave “new” para obter memória e o sistema coleta de lixo (garbage collector) interno irá comer os biscoitos que ninguém mais quer. (Bem, nenhuma analogia é perfeita...). Quando se cria um objeto a partir de uma classe, diz-se que foi criada uma instância da classe. Quando se tem uma instrução como:

```
Date data = new Date();
```

Internamente o operador será usado para criar uma nova instância da classe “Date”.

Tudo o que se escreve em Java está dentro de uma classe. A biblioteca Java padrão fornece centenas de classes para diversos propósitos tais como projeto de interface de usuário e programação em rede. Apesar de tudo, você ainda tem de criar suas próprias classes em Java para descrever os objetos dos domínios do problema de seus aplicativos e adaptar as classes que são fornecidas pela biblioteca padrão para atender a seus próprios objetivos.

Quando se começa a escrever classes em Java, outro princípio da POO facilita isso: as classes podem ser (e em Java sempre são) formadas a partir de outras classes. Nós dizemos que uma classe que é formada a partir de outra classe a estende. A linguagem Java, de fato, vem com uma espécie de “classe base”, ou seja, uma classe a partir da qual todas as outras classes são elaboradas. Em Java, todas as classes estendem essa classe base chamada “*Object*”.

Ao estender uma classe base, a nova classe inicialmente tem todas as propriedades

e métodos de seu progenitor. Pode-se escolher então entre modificar ou simplesmente manter qualquer método da progenitora e também adicionar novos métodos que aplicam-se somente à classe filha. O conceito genérico de estender uma classe base é chamado de herança.

Encapsulamento (algumas vezes chamada de ocultamento de dados) é outro conceito-chave ao se trabalhar com objetos. Formalmente, o encapsulamento são é nada mais que combinar dados e comportamento em um pacote e ocultar a implementação dos dados do usuário do objeto. Os dados em um objeto são geralmente chamados de variáveis de instância ou campos; e as funções e procedimentos de uma classe Java são chamados métodos. Um objeto específico que é uma instância de uma classe terão valores específicos em seus campos que definem seu estado atual.

Nunca será demais enfatizar que a chave para se fazer o encapsulamento funcionar é fazer programas que nunca tenham acesso direto às variáveis de instância (campos) de uma classe. Os programas precisam interagir com seus dados somente através dos métodos do objeto. O encapsulamento é a forma de se das aos objetos seu comportamento de "caixa preta", característica-chave para a reutilização e a confiabilidade. Isso significa que um objeto pode mudar totalmente como ele armazena seus dados, mas, enquanto continuar usando os mesmos método para processar os dados, nenhum outro objeto vai saber disso ou se importar com isso.

Objetos

Para trabalhar com POO, deve-se poder identificar três características-chave dos objetos. (para aqueles que ainda se lembram, é análogo à forma do "Quem, O quê e Onde" que os professores ensinavam para caracterizar um evento). As Três perguntas-chave aqui são:

- Qual é o comportamento do objeto?
- Qual é o estado do objeto?
- Qual é a identidade do objeto?

Todos os objetos que são instâncias de uma mesma classe compartilham uma semelhança de família apresentando um comportamento semelhante. O comportamento de um objeto é definido pelas mensagens que ele aceita.

Em seguida, cada objeto guarda informações sobre o que ele é no momento e como chegou a isso. Isto é o que geralmente é chamado de estado do objeto. O estado de um objeto precisa resultar de mensagens enviadas a este objeto (caso contrário o encapsulamento seria desrespeitado).

Contudo, o estado de um objeto não descreve completamente o objeto, pois cada objeto rem uma identidade distinta. Por exemplo, em um sistema de processamento de encomendas, duas encomendas são distintas mesmo se pedirem a mesma coisa. Observe que os objetos individuais que são instâncias de uma classe sempre tem identidades diferentes e, geralmente, têm estados diferentes.

Essas características-chave podem influenciar umas às outras. Por exemplo, o estado de um objeto pode influenciar seu comportamento. (Se uma encomenda for "enviada" ou "paga" ela pode rejeitar uma mensagem que solicita o acréscimo ou a remoção de itens da encomenda. Da mesma forma, se um pedido estiver "vazio", ou seja, ainda nenhum item tiver sido solicitado, ele não poderá ser enviado).

Recomendação: uma regra simples na identificação de classes é procurar por substantivos na analise de um problema. Já os métodos, por outro lado, correspondem aos verbos.

Por exemplo, em um sistema de processamento de pedidos, alguns desses nomes substantivos são:

- item
- pedido (ou encomenda)
- endereço de remessa
- pagamento
- conta

Esses nomes podem levar às classes: *Item, Pedido* etc.

Em seguida, procuram-se os verbos. Itens são adicionados aos pedidos. Pedidos são enviados ou cancelados. Pagamentos são aplicados aos Pedidos. Com cada verbo, como “adicionar”, “enviar”, “cancelar” e “aplicar”, é necessário identificar o objeto que tem a maior responsabilidade em efetuar essa ação. Por exemplo, quando um item novo é adicionado a um pedido, o objeto pedido deverá ser o encarregado, já que ele sabe como guardar e classificar os itens. Ou seja, *adicionar* deverá ser um método da classe *Pedido* que tem um objeto *Item* como parâmetro.

Evidentemente, a regra, “substantivo e verbo” é apenas uma diretriz e somente a experiência poderá ajudar você a decidir que substantivos e verbos são os mais importantes ao se elaborarem as classes.

Relação entre Classes

As relações mais comuns entre classes são:

- uso
- inclusão (“tem um”)
- herança (“é um”)

A relação uso é a mais óbvia e também a mais genérica. Por exemplo, a classe *Pedido* usa a classe *Conta*, já que os objetos *Pedido* precisam acessar os objetos *Conta* para verificar o crédito. Mas a classe *Item* não usa a classe *Conta* pois os objetos *Item* nunca precisam se preocupar com as contas dos clientes. Assim, uma classe usa outra classe se ela manipula objetos dessa classe.

De um modo geral, uma classe *A* usa uma classe *B* se:

- um método de *A* envia uma mensagem para um objeto da classe *B*, ou
- um método de *A* cria, recebe ou retorna objetos da classe *B*.

Recomendação: Tente minimizar o número de classes que usam umas às outras. O ponto é que, se uma classe *A* não sabe da existência de uma classe *B*, ela também fica indiferente ao que acontece com a classe *B* e isso significa que qualquer alteração na classe *B* não vai introduzir erros na classe [a]

A relação *inclusão* é fácil de entender porque ela é concreta; por exemplo, um objeto *Pedido* contém objetos *Item*. A inclusão significa que objetos da classe *A* contêm objetos da classe *B.* Evidentemente, a inclusão é um caso especial de uso; se um objeto *A* contém um objeto *B*, então pelo menos um método da classe *A* irá usar esse objeto da classe *B*.

A relação de *herança* denota especialização. Por exemplo, uma classe *PedidoExpresso* será herdeira de uma classe *Pedido*. A classe especializada *PedidoExpresso* tem métodos especiais para lidar com prioridades e um método diferente para calcular as taxas de remessa, enquanto que os outros métodos, como adição de

itens e cobrança serão simplesmente herdados da classe *Encomenda*. Em geral, se a classe *A* estender a classe *B*, a classe *A* vai herdar métodos da classe *B* e ter recursos a mais.

Recomendações para o Projeto de Classes

1. Sempre mantenha os dados privados

   . Esta regra vem antes de tudo: qualquer coisa diferente vai violar o encapsulamento. Você pode ter de escrever um método “acessador” (get) ou “modificador” (set) ocasionalmente, mas será ainda melhor manter privados os campos de instância.

2. Sempre inicialize os dados.

   . A linguagem Java não inicializa as variáveis locais automaticamente, mas inicializa as variáveis de instância de objetos. Não confie nos valores a priori, mas inicialize as variáveis explicitamente, seja fornecendo um valor inicialmente ou especificando os padrões em todos os construtores.

3. Não use tipos básicos em demasia numa classe.

   . A idéia é substituir os usos relacionados múltiplos de tipos básicos por outras classes. Isso mantém as classes mais fáceis de entender e alterar. Por exemplo, substitua os campos de instância a seguir em uma classe *Cliente*

```
private String rua;
private String cidade;
private String estado;
private int cep;
```

   . por uma nova chamada *Endereco*. Dessa forma, você poderá lidar facilmente com alterações de endereço, bem como com a necessidade de lidar com endereços internacionais.

4. Nem todos os campos precisam de “acessadores” (get) e “modificadores” (set) de campos individuais.

   . Pode ser necessário obter e especificar o salário de alguém, o que é normal. Porém, você certamente não vai querer alterar sua data de contratação, uma vez o objeto construído. E com bastante freqüência, os objetos têm variáveis de instância que não se quer que outros leiam ou modifiquem.

5. Use uma forma padrão de definição das classes.

   . Nós sempre listamos o conteúdo das classes na seguinte ordem:

      1. recursos públicos
      2. recursos de escopo de pacote
      3. recursos privados
      4. constantes
      5. construtores
      6. métodos
      7. métodos estáticos

   . Use você essa ordem ou não, a coisa mais importante é ser consistente.

6. Divida classes com tarefas demais.

. Esta recomendação é, evidentemente, um tanto quento vaga, pois "demais" é algo relativo. Contudo, se houver uma forma óbvia de tornar uma classe complicada em duas conceitualmente mais simples, aproveite a oportunidade. (Por outro lado, não exagere; 10 classes, cada uma com somente um método, é geralmente tiro de canhão em mosquito).

7. Dê nomes às classes e métodos que representem suas tarefas.

. Assim como as variáveis devem ter nomes representativos, as classes também devem seguir essa metodologia.

. Uma boa convenção é que um nome de classe deva ser um substantivo (Pedido) ou um substantivo junto com um adjetivo (ex.: *PedidoExpresso* ou *EnderecoCobranca*). Como nos métodos, siga a convenção padrão dos métodos "acessadores" que começam com *get* em minúsculas (*getDia*) e "modificadores" que começam com *set* também em minúsculas (*setSalario*).

Primeiros Passos com a Herança

Vamos supor que você trabalhe para uma empresa cujos diretores são tratados de maneira bem diferente que os outros empregados (classe *Empregado*). Seus aumentos são calculados diferentemente; eles têm direito a uma secretária e coisas assim. Esse é o tipo de situação que, em POO, pede herança. Por que? Bem, é necessário definir uma nova classe, (*Gerente*) e adicionar funcionalidade. Mas, pode-se reter algumas das coisas que já foram programadas na classe *Empregado* e todos os campos de instância da classe original podem ser preservados. De forma mais abstrata, há uma evidente relação entre *Empregado* e *Gerente*, pois cada gerente é um empregado: essa relação é a marca registrada da herança.

Eis o código da classe *Empregado*.

```
import java.util.*;

public class Empregado {
  private String nome;
  private double salario;
  private Date dataContratacao;

  public Empregado(String nome, double sal, Date data) {
    this.nome = nome;
    this.salario = sal;
    this.dataContratacao = data;
  }

  public void aumentaSalario(double porPercentual) {
      salario *= 1 + porPercentual / 100;
    }

  public int anoContratacao() {
    Calendar dt = Calendar.getInstance();
    dt.setTime(dataContratacao);

    return dt.get(Calendar.YEAR);
  }

  public String toString() {
      return String.format(new Locale("pt", "BR"), "%s %,.2f %d", nome, salario,
        anoContratacao());
    }
}
```

A palavra-chave *extends* na primeira linha da classe Gerente indica que está sendo criada uma nova classe que deriva de uma classe existente. A classe existente é chamada de superclasse, classe base ou classe progenitora. A nova classe é chamada de subclasse, classe derivada, ou classe filha. Os termos superclasse e subclasse são os mais usados habitualmente por programadores Java.

A classe *Empregado é uma superclasse,* mas não porque ela é superior a sua *subclasse* ou contenha mais funcionalidade. Na verdade, o que acontece é o oposto: as subclasses têm mais funcionalidades que suas superclasses. Por exemplo, como veremos ao analisar o restante do código da classe *Gerente*, essa classe encapsula mais dados e tem maior funcionalidade que sua superclasse *Empregado*.

Agora observe o construtor da classe *Gerente*:

```
public Gerente(String nome, double sal, Date data) {
  super(nome, sal, data);
  nomeSecretaria = "";
}
```

A palavra-chave *super* sempre se refere a uma superclasse (neste case, *Empregado*). De modo que a linha

```
super(nome, sal, data);
```

é um modo mais sucinto de chamar o construtor da classe *Empregado* usando *nome*, *sal*, e *dia* como parâmetros. O motivo dessa linha de código é que todo o construtor de uma subclasse necessita chamar um construtor para os campos de dados da superclasse. Se isto não ocorrer, o compilador Java irá procurar pelo construtor padrão da superclasse, que na sua ausência gera-rá um erro de compilação.

Se você comparar a classe *Gerente* com a classe *Empregado*, vai notar que muitos dos métodos da classe *Empregado* não são repetidos na classe *Gerente*. Isto porque, a menos que especificado, uma subclasse sempre usa os métodos da superclasse. Em particular, quando se elabora a subclasse por herança a partir da superclasse, basta indicas as diferenças entre a subclasse e a superclasse. A capacidade de reutilizar métodos da superclasse é automática.

Os Objetos sabem como fazer seu trabalho: Polimorfismo

É importante entender o que acontece quando a chamada de um método é aplicada a objetos de vários tipos em uma hierarquia de heranças. Lembre-se de que na POO são enviadas mensagens para objetos, pedindo-lhes que realizem certas ações. Ao enviar uma mensagem que pede para uma subclasse aplicar um método usando certos parâmetros, eis o que acontece:

- A subclasse verifica se ela tem ou não um método com esse nome e com exatamente os mesmos parâmetros. Se tiver, usa-o.

Caso não,

- a classe progenitor torna-se responsável pelo processamento da mensagem e procura por um método com esse nome e esses parâmetros. Se encontrar, chama esse método.

Como o processamento da mensagem pode continuar subindo pela seqüência de heranças, as classes progenitoras são verificadas até que a cadeia pára ou até que um método coincidente seja encontrado. (Se não houver nenhum método coincidente em

nenhum lugar da seqüência de heranças, vai surgir um erro em tempo de compilação). Observe que podem existir métodos com mesmo nome na seqüência. Isso nos leva a uma das regras fundamentais de herança:

- Um método definido em uma subclasse com o mesmo nome e mesma lista de parâmetros que um método em uma de suas classes antecessoras oculta o método da classe ancestral a partir da subclasse.

A capacidade de um objeto em decidir que método aplicar a si mesmo, dependendo de onde ele está na hierarquia de heranças, é geralmente chamada de polimorfismo. A idéia por trás do polimorfismo é que, embora a mensagem possa ser a mesma, os objetos podem responder diferentemente. O polimorfismo pode ser aplicado a qualquer método que seja herdado de uma superclasse.

A chave para fazer o polimorfismo funcionar é chamada de ligação tardia (late binding). Isso significa que o compilador não gera código para chamar um método em tempo de compilação. Em vez disso, cada vez que se aplica um método a um objeto, o compilador gera código para calcular que método deve ser chamado, usando informações de tipo do objeto. (Esse processo é também chamado de ligação dinâmica ou despacho dinâmico). O mecanismo de chamada de método tradicional é chamado de ligação estática (static binding ou early binding), pois a operação a ser executada é totalmente determinada em tempo de compilação. A ligação estática depende apenas do tipo de variável objeto; já a ligação dinâmica depende do tipo do objeto real em tempo de execução.

Para resumir, herança e polimorfismo permitem ao aplicativo determinar a maneira geral de proceder. As classes individuais na hierarquia de heranças são responsáveis pelos detalhes – usando polimorfismo para determinar que métodos chamar. O polimorfismo em uma hierarquia de heranças é algumas vezes chamado de polimorfismo verdadeiro. A idéia é distingui-lo do tipo mais limitado de sobrecarga de nome (overloading) que não é resolvido dinamicamente, mas estaticamente em tempo de compilação.

Como evitar Herança: Classes e Métodos Finais (Final)

Ocasionalmente, pode-se querer impedir que uma classe seja derivada a partir de outras. As Classes que não podem ser classes-mãe são chamadas de classes finais e usa-se o modificador *final* na definição da classe para indicar isso.

Pode-se também fazer *final* um método específico de uma classe. Se isso for feito, então nenhuma subclasse poderá sobrepor ou substituir esse método. (Todos os métodos de uma classe final são automaticamente “finais”). Dois motivos para fazer uma classe ou um método serem finais:

1. Eficiência

   . A ligação dinâmica é mais trabalhosa, em termos de processamento, que a ligação estática. Ou seja, os métodos virtuais executam mais devagar. O mecanismo de ligação dinâmica é um pouco menos eficiente que a chamada direta a um procedimento. Mais importante, o compilador não tem como substituir um método trivial por um código “in-line” porque é possível que uma classe derivada sobreponha esse código trivial. O compilador pode colocar métodos finais in-line. Por exemplo, se “e.getNome()” for final, o compilador poderá substituí-lo por “e.nome”. (Assim, pode-se obter todos os benefícios do acesso direto a campos de instância ser violar o encapsulamento).

2. Segurança

. A flexibilidade do mecanismo de despacho dinâmico significa que não há controle do que acontece quando se chama um método. Ao enviar uma mensagem, como “e.getNome()”, é possível que “e” seja um objeto de uma classe derivada que redefiniu o método “getNome” para retornar um string totalmente diferente. Ao fazer o método final, essa ambigüidade é evitada.

Conversão de Tipo Explícita (cast)

Para efetuar a conversão de um tipo primitivo para outro utilizamos em Java uma notação especial chamada de “cast”. Por exemplo,

```
double x = 3.405;
int nx = (int)x;
```

converte a expressão “x” em um inteiro, descartando a parte fracionária.

Assim como você, ocasionalmente, precisa converter um número de ponto flutuante em inteiro, também pode precisar converter a referência de um objeto de uma classe para outra. Assim como na conversão dos tipos básicos, esse processo é chamado de “cast”. Para realmente fazer uma conversão explícita, usa-se uma sintaxe semelhante à que foi usada para converter dados de variáveis de tipos básicos. Cerque o tipo do alvo com parênteses e coloque-o antes da referência de objeto que se quer converter. Por exemplo,

```
Gerente chefe = (Gerente)grupo[0];
```

Só há uma razão para que se queira fazer uma conversão de tipo explícita – usar um objeto em sua plena capacidade após seu tipo verdadeiro ter sido subestimado. Por exemplo, na classe “Gerente”, o array “grupo” teria de ser um arrai de objetos “Empregado” já que alguns de seus itens era empregados normais. E teríamos de converte os elementos do array, de volta para “Gerente”, a fim de acessar qualquer um de seus novos campos.

Como sabemos, em Java, toda variável objeto tem um tipo. O tipo descreve o gênero de objeto ao qual a variável faz referência e o que ela pode fazer. Por exemplo, “grupo[i]” refere-se a um objeto “Empregado” (e, portanto, também pode fazer referência a um objeto “Gerente”).

O código é elaborado com base nessas descrições e o compilador verifica a correção dessas descrições de variáveis. Se um objeto de subclasse for atribuído a uma variável de superclasse, é como se você estivesse prometendo menos coisas e o compilador vai simplesmente permitir que isso seja feito. Mas, se você atribuir um objeto de superclasse a uma variável de subclasse, é como prometer mais do que havia sido especificado e o compilador exige confirmação disso através do uto de “cast” para o tipo da superclasse.

O que aconteceria se você tentasse casar para baixo na cadeia de herança e estivesse “mentindo” a respeito do que um objeto contém? Durante a execução do programa o sistema lançaria uma exceção e o programa iria, normalmente, ser interrompido. É uma boa prática de programação descobrir se um objeto é ou não uma instância de outra classe antes de fazer uma conversão de tipo explícita “cast”. Isso é realizado com o operador “instandeof”.

```
if(grupo[1] instanceof Gerente)
  chefe = (Gerente)grupo[1];
```

Finalmente, o compilador não vai permitir que seja feito um “cast” se não houver possibilidades de que a conversão seja bem sucedida.

```
Window w = (Window)grupo[1];
```

gera um erro em tempo de compilação pois “Window” não é uma subclasse de “Empregado”.

Resumindo:

- Pode-se fazer uma conversão de tipo explícita “cast” somente dentro de uma hierarquia de heranças.
- Usa-se “instandeof” para verificar uma hierarquia antes de fazer a conversão de tipo explícita de um objeto de superclasse para uma subclasse.

Classes Abstratas

Ao subir na hierarquia de heranças, as classes tornam-se mais genéricas e, provavelmente, mais abstratas. Em algum ponto, a classe ancestral torna-se tão geral que acaba sendo vista mais como um modelo pra outras classes do que uma classe com instâncias específicas que são usadas. Considere, por exemplo, um sistema de mensagens eletrônicas que integre e-mail, fax e correio de voz. Ele terá de ser capaz de lidar com mensagens de texto e correio de voz.

Seguido os princípios da POO, o programa vai precisar de classes como “TextMessage” e “VoiceMessage”. Evidentemente, uma caixa postal terá de armazenar uma mistura deses tipos de mensagens, de modo que possa acessá-las através de referências à classe mão “Message” comum a todas as demais. A hierarquia das heranças é mostrada abaixo:

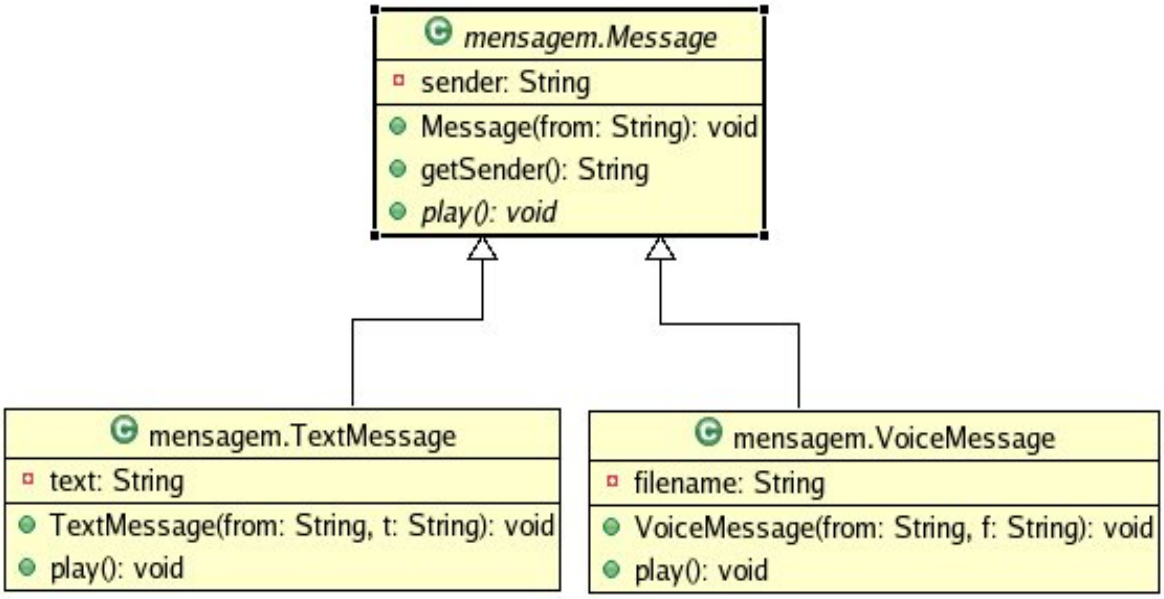

Diagrama de herança das classes mensagens

Porque se preocupar com um nível tão alto de abstração? A resposta é que isso torna o projeto das classes mais claro. No final das contas, uma das chaves da POO é entender como combinar e reunir operações comuns num nível mais alto na hierarquia de heranças. Em nosso caso, por exemplo, todas as mensagens têm um método comum denominado “play()”. É fácil perceber como reproduzir uma mensagem de voz enviando-a ao alto falante, ou uma mensagem de texto exibindo-a em uma janela de texto, ou uma mensagem de fax imprimindo-a ou mostrando-a em uma janela gráfica, mas e como fazer para implementar o método “play()” na classe progenitora “Message”?

A resposta é que não se pode. Em Java, usa-se a palavra-chave “abstract” para

indicar que um método não pode ser especificado nossa classe. Para aumentar a clareza, uma classe com um ou mais métodos abstratos precisa, ela mesma, ser declarada abstrata.

```
public abstract class Message {
  ...
  public abstract void play();
}
```

Além dos métodos abstratos, as classes abstratas podem ter dados e métodos concretos. Por exemplo, a classe “Message” pode armazenar o remetente da mensagem postal e ter um método concreto que retorne o nome do remetente “sender”.

```
public abstract class Message {
  private String sender;

  public Message(String from) {
    sender = from;
  }

  public String getSender() {
    return sender;
  }

  public abstract void play();
}
```

Um método abstrato promete que todos os descendentes não abstratos dessa classe abstrata irão implementar esse método abstrato. Os métodos abstratos funcionam como uma espécie de guardador de lugar para métodos que serão posteriormente implementados nas subclasses.

Uma classe pode ser declarada como abstrata mesmo sem ter métodos abstratos.

Não se podem criar objetos a partir de classes abstratas. Ou seja, se uma classe é declarada abstrata, então nenhum objeto dessa classe pode ser variado. Será necessário ampliar essa classe a fim de criar uma instância da classe. Observe ainda assim podem-se criar variáveis objeto de uma classe abstrata, embora essas variáveis tenham de fazer referência a um objeto de uma subclasse não abstrata.

```
Message msg = new VoiceMessage("bemvindo.wav");
```

Aqui, “msg” é uma variável do tipo abstrato “Message” que faz referência a uma instância da subclasse não abstrata “VoiceMessage”.

Para vermos uma realização concreta dessa classe abstrata e também do método “play”, vamos exemplificar o código para a classe “TextMessage”:

```
import javax.swing.JOptionPane;

public class TextMessage extends Message {
  private String text;

  public TextMessage(String from, String t) {
    super(from);
    text = t;
```

```
  }

  public void play() {
    JOptionPane.showMessageDialog(null, text, "M e n s a g e m :  "
      + getSender(), JOptionPane.INFORMATION_MESSAGE);
  }
}
```

Observe que nós precisamos fornecer somente uma definição concreta do método abstrato “play” na classe “TextMessage”.

Abaixo segue o código completo do programa exemplo de criação e leitura de mensagens.

Classe Message

```
public abstract class Message {
  private String sender;

  public Message(String from) {
    sender = from;
  }

  public String getSender() {
    return sender;
  }

  public abstract void play();
}
```

Classe TextMessage

```
import javax.swing.JOptionPane;

public class TextMessage extends Message {
  private String text;

  public TextMessage(String from, String t) {
    super(from);
    text = t;
  }

  public void play() {
    JOptionPane.showMessageDialog(null, text, "M e n s a g e m :  "
      + getSender(), JOptionPane.INFORMATION_MESSAGE);
  }
}
```

Classe VoiceMessage

```
import java.net.*;
import java.applet.*;
import javax.swing.JOptionPane;

public class VoiceMessage extends Message {
  private String filename;

  public VoiceMessage(String from, String f) {
    super(from);
    filename = f;
  }
```

```
  @Override
  public void play() {
    try {
      URL u = new URL("file", "localhost", filename);
      AudioClip clip = Applet.newAudioClip(u);
      clip.play();
    } catch(Exception ex) {
      JOptionPane.showMessageDialog(null, "NÃ£o posso abrir o arquivo " +
filename,
        "E R R O", JOptionPane.ERROR_MESSAGE);
    }

  }
}
```

Classe Mailbox

```
public class Mailbox {
  private static final int MAX_MSG = 10;
  private int in = 0;
  private int out = 0;
  private int nmsg = 0;
  private Message[] messages = new Message[MAX_MSG];

  public Message remove() {
    Message r = null;

    if(nmsg > 0) {
      r = messages[out];
      nmsg--;
      out = (out + 1) % MAX_MSG;
    }

    return r;
  }

  public void insert(Message m) {
    if(nmsg < MAX_MSG) {
      messages[in] = m;
      nmsg++;
      in = (in + 1) % MAX_MSG;
    }
  }

  public String status() {
    String msg = "";

    if(nmsg == 0)
      msg = "Mailbox vazio";
    else if(nmsg ==1)
      msg = "1 mensagem";
    else if(nmsg < MAX_MSG)
      msg = nmsg + " mensagens";
    else
      msg = "Mailbox cheio";

    return msg;
  }
}
```

Classe MailboxTest

```
import javax.swing.JOptionPane;

public class MailboxTest {
  public static void main(String[] args) {
    Mailbox mbox = new Mailbox();

    while(true) {
      String from = "";
      String msg = "";

      int cmd = JOptionPane.showOptionDialog(null, "Selecione a aÃ§Ã£o",
        "Mailbox " + mbox.status(), JOptionPane.DEFAULT_OPTION,
        JOptionPane.QUESTION_MESSAGE, null, new String[] {
          "LÃª mensagem", "Cria mensagem Texto", "Insere mensage de Voz", "Sair"
        }, "LÃª mensagem");

      switch(cmd) {
        case 0:
          Message m = mbox.remove();
          if(m != null) {
            JOptionPane.showMessageDialog(null,
              "Selecione OK para obter a mensagem", "Mensagem de: "
                + m.getSender(), JOptionPane.INFORMATION_MESSAGE);
            m.play();
          } else {
            JOptionPane.showMessageDialog(null, "NÃ£o existem mensagens",
              "A t e n Ã§ Ã£ o", JOptionPane.INFORMATION_MESSAGE);
          }
          break;
        case 1:
          from = JOptionPane.showInputDialog(null, "Informe seu nome",
            "Mensagem Texto", JOptionPane.QUESTION_MESSAGE);
          msg = JOptionPane.showInputDialog(null, "Entre com a mensagem",
            "Mensagem Texto", JOptionPane.QUESTION_MESSAGE);
          mbox.insert(new TextMessage(from, msg));
          break;
        case 2:
          from = JOptionPane.showInputDialog(null, "Informe seu nome",
            "Mensagem de Voz", JOptionPane.QUESTION_MESSAGE);
          msg = JOptionPane.showInputDialog(null,
            "Informe o nome do arquivo de Ã¡udio", "Mensagem de Voz",
            JOptionPane.QUESTION_MESSAGE);
          mbox.insert(new VoiceMessage(from, msg));
          break;
        default:
          System.exit(0);
      }
    }
  }
}
```

Acesso Protegido

Como sabemos, os campos de instância numa classe são melhor declarados como “private” e os métodos normalmente declarador como “public”. Quaisquer recursos declarados “private” não serão visíveis para as outras classes. Isto é verdade também para subclasses: uma subclasse não pode acessar os membros de dados privados de sua superclasse.

Há porém, casos em que se quer que uma subclasse tenha acesso a um método ou dado de uma classe progenitora. Nesse case, declara-se esse recurso como “protected” (protegido). Por exemplo, se a classe Empregado declarar o objeto “diaContratação” como “protected”, em vez de “private”, então os métodos de “Gerente” poderão acessá-lo diretamente.

Na prática, deve-se usar o atributo “protected com cautela. Suponha que sua classe seja usada por outros programadores e que você a tenha projetado com dados protegidos. Sem que você saiba, outros programadores poderão derivar classe a partir de sua classe e então começar a acessar seus campos de instância protegidos. Neste caso, você não poderá mais alterar a implementação de sua classe sem atrapalhar os outros programadores. Isso, evidentemente, é contrário ao espírito da POO, que enfatiza o encapsulamento dos dados.

Já os métodos protegidos fazem mais sentido. Uma classe pode declarar um método como “protected” se sua utilização for específica. Isso indica que as subclasses (as quais, presumivelmente, conhecem bem seus ancestrais) podem ser confiáveis na utilização do método corretamente, mas outras classes não.

Eis um resumo dos quatro modificadores de acesso em Java que controlam a visibilidade:

1. Visível somente para a classe – private
2. Visível para o mundo public
3. Visível para o pacote e todas as subclasses – protected
4. Visível para o pacote - “sem declaração de encapsulamento”

Interfaces

Uso de uma Superclasse Abstrata

Suponha que queiramos escrever uma rotina de ordenamento genérica que trabalhe com vários tipos diferentes de objetos Java. Agora você já sabe como organizar isso em um modelo orientado a objetos. Você começa com uma classe abstrata “Sortable” contendo um método “compareTo” que determina se ou não um objeto ordenável é menor que, igual ou maior que outro.

Primeiro, deve-se implementar um algoritmo de ordenamento genérico. Eis a seguir uma implementação de uma variante do algoritmo “Shell”, para ordenar um array de objetos “Sortable”. Não se preocupe em saber exatamente como esse algoritmo funciona. Nós simplesmente o escolhemos porque é simples de implementar. Observe sim que o algoritmo analisa cada elemento do array e os compara usando o método “compareTo”, ordenando-os em seguida.

```
public abstract class Sortable {
  public abstract int compareTo(Sortable b);
}


public class ArrayAlg {

  public static void shellSort(Sortable[] a) {
    int n = a.length;
    int incr = n / 2;

    while(incr >= 1) {
```

```
      for(int i = incr;i < n;i++) {
        Sortable temp = a[i];
        int j = i;

        while(j >= incr && temp.compareTo(a[j - incr]) < 0) {
          a[j] = a[j - incr];
          j -= incr;
        }
        a[j] = temp;
      }
      incr /= 2;
    }
  }
}
```

Essa rotina de ordenamento pode ser usada em todas as subclasses da classe abstrata Sortable (sobrepondo o método “compareTo” nas subclasses).

Por exemplo, para ordenar um array de empregados (ordenando-os por qualquer coisa, como seus salários por exemplo), é necessário:

1. Derivar “Empregado” a partir de “Sortable”.
2. Implementar o método “compareTo” para os empregados.
3. Chamar “ArrayAlg.shellSort” no array de empregados.

Eis a seguir um exemplo do código extra necessário para fazer isso em nossa classe “Empregado”:

```
public class Empregado extends Sortable {
  ...

  @Override
  public int compareTo(Sortable b) {
    int resultado = 0;
    Empregado ep = (Empregado)b;

    if(salario < ep.salario)
      resultado = -1;
    else if(salario > ep.salario)
      resultado = 1;

    return resultado;
  }
}
```

Abaixo segue o código completo do algoritmo de ordenamento genérico e a classificação do array de empregados.

Classe EmpregadoSortTest

```
import java.util.*;

public class EmpregadoSortTest {
  public static void main(String[] args) {
    Empregado[] grupo = new Empregado[3];
    Calendar dt = Calendar.getInstance();
    dt.set(1989, 10, 1);
    grupo[0] = new Empregado("Chico da Silva", 1500, dt.getTime());
    (dt = Calendar.getInstance()).set(1987, 12, 15);
    grupo[1] = new Gerente("JosÃ© de Oliveira", 2500, dt.getTime());
    (dt = Calendar.getInstance()).set(1990, 3, 18);
    grupo[2] = new Empregado("Ana Ribeiro", 1700, dt.getTime());

    ArrayAlg.shellSort(grupo);
```

```
    for(int i = 0;i < grupo.length;i++)
      System.out.println(grupo[i]);
  }
}
```

Classe ArrayAlg

```
public class ArrayAlg {
  public static void shellSort(Sortable[] a) {
    int n = a.length;
    int incr = n / 2;

    while(incr >= 1) {
      for(int i = incr;i < n;i++) {
        Sortable temp = a[i];
        int j = i;

        while(j >= incr && temp.compareTo(a[j - incr]) < 0) {
          a[j] = a[j - incr];
          j -= incr;
        }
        a[j] = temp;
      }
      incr /= 2;
    }
  }
}
```

Classe Sortable

```
public abstract class Sortable {
  public abstract int compareTo(Sortable b);
}
```

Classe Empregado (Modificada)

```
import java.util.*;

public class Empregado extends Sortable {
  private String nome;
  private double salario;
  private Date dataContratacao;

  public Empregado(String nome, double sal, Date data) {
    this.nome = nome;
    this.salario = sal;
    this.dataContratacao = data;
  }

  public void aumentaSalario(double porPercentual) {
      salario *= 1 + porPercentual / 100;
    }

  public int anoContratacao() {
    Calendar dt = Calendar.getInstance();
    dt.setTime(dataContratacao);

    return dt.get(Calendar.YEAR);
  }
```

```
  public String toString() {
      return String.format(new Locale("pt", "BR"), "%s %,.2f %d", nome, salario,
        anoContratacao());
    }

  @Override
  public int compareTo(Sortable b) {
    int resultado = 0;
    Empregado ep = (Empregado)b;

    if(salario < ep.salario)
      resultado = -1;
    else if(salario > ep.salario)
      resultado = 1;

    return resultado;
  }
}
```

Class Gerente

```
import java.util.*;

public class Gerente extends Empregado {
    private String nomeSecretaria;

    public Gerente(String nome, double sal, Date data) {
      super(nome, sal, data);
      nomeSecretaria = "";
    }
    public String getNomeSecretaria() {
      return nomeSecretaria;
    }
    public void setNomeSecretaria(String nome) {
      nomeSecretaria = nome;
    }
}
```

Uso de Interfaces

Infelizmente, há um problema importante ao se usar uma classe abstrata para expressar uma propriedade genérica. Eis a seguir um exemplo onde esse problema surge: considere uma classe “Tile” que modela janelas lado a lado (tiled) sobre a área de trabalho (tela) do computador. As janelas dado a lado são na verdade retângulos com uma ordem z (z-order). Ora, as janelas com ordem z maior são exibidas na frente daquelas com ordem z menor. Para reutilizar código, nós derivamos “Tile” de “Rectangle”, uma classe que já está definida no pacote “java.awt”.

```
import java.awt.Rectangle;

public class Tile extends Rectangle {
  private int z;

  public Tile(int x, int y, int width, int height, int z) {
    super(x, y, width, height);
    this.z = z;
  }
}
```

Agora nós gostaríamos de ordenar um array de janelas lado a lado comparando suas ordens z. Se tentarmos aplicar o procedimento para tornar essas janelas ordenáveis, ficaríamos presos no primeiro passo. Nós não podemos derivar “Tile” de “Sortable”, pois esta já deriva de “Rectangle”!

O problema é que, em Java, uma classe só pode ter uma superclasse. Outras linguagens de programação, em particular C++, permitem que uma classe tenha mais de uma superclasse. Esse recurso é chamado de *herança múltipla*.

Em vez disso, a linguagem Java introduz a noção de interfaces para recuperar grande parte da funcionalidade que a herança múltipla oferece. Os projetistas Java optaram por esse caminho porque as heranças múltiplas tornam os compiladores muito complexos (como em C++) ou muito ineficientes (como em Eiffel). As interfaces também são o método preferido de se implementar funções de “callback” (chamada de retorno) em Java.

Bom, e o que é uma interface afinal? Essencialmente, é uma promessa de que uma classe vai implementar certos métodos com certas características. Usa-se inclusive a palavra-chave “*implements*” para indicar que a classe vai manter essa promessa. A maneira com que esses métodos são implementados depende da classe, evidentemente. O que é importante, até onde interessa ao compilador, é que os métodos tenham as características corretas.

Por exemplo, a biblioteca padrão define uma interface chamada “Comparable” que poderia ser usada por qualquer classe cujos elementos possam ser comparados. O código da interface “Comparable” se parece com este:

```
public interface Comparable {
  public int comparaTo(Object b);
}
```

Esse código promete que qualquer classe que implemente a interface “Comparable” terá um método “compareTo”. Evidentemente, a maneira pela qual o método “compareTo” funciona (ou mesmo se funciona ou não conforme esperado) em uma classe específica vai depender da classe que está implementando a interface “Comparable”. O ponto a ser observado aqui é que qualquer classe pode prometer implementar “Comparable” - não importa se sua superclasse promete ou não o mesmo. Todos os descendentes de uma classe dessa haveriam de implementar “Comparable” automaticamente, pois todos eles teriam acesso a um método 'compareTo” com as características corretas.

Para informar à linguagem Java que uma classe implementa a interface “Comparable”, ela deve ser definida da seguinte forma:

```
public class Tile extends Rectangle implements Comparable
```

Depois, basta implementar um método “compareTo” dentro da classe:

```
public class Tile extends Rectangle implements Comparable<Tile> {
  ...
  public int compareTo(Tile o) {
    return z - o.z;
  }
}
```

Convenientemente, a classe “Arrays” do Java fornece um algoritmo de ordenamento

para um array de objetos “Comparable”. Nós podemos usar isso para ordenar um array de janelas lado a lado:

```
    Tile[] a = new Tile[20];
    ...
    Arrays.sort(a);
```

Abaixo segue o fonte do exemplo de janelas lado a lado.

Classe Tile

```
import java.awt.Rectangle;

public class Tile extends Rectangle implements Comparable<Tile> {
  private int z;

  public Tile(int x, int y, int width, int height, int z) {
    super(x, y, width, height);
    this.z = z;
  }

  @Override
  public String toString() {
    return super.toString() + "z=" + z + "]";
  }

  public int compareTo(Tile o) {
    return z - o.z;
  }
}
```

Classe TileTest

```
import java.util.*;

public class TileTest {
  public static void main(String[] args) {
    Tile[] a = new Tile[20];

    for(int i = 0;i < a.length;i++)
      a[i] = new Tile(i, i, 10, 20, (int)(100 * Math.random()));

    Arrays.sort(a);

    for(int i = 0;i < a.length;i++)
      System.out.println(a[i]);
  }
}
```