JAVA 3

Classes e objetos

OS CONCEITOS BÁSICOS DE ORIENTAÇÃO A OBJETOS são apresentados neste módulo. Em seguida são demonstrados exemplos de sua implementação na linguagem Java, implementados em exemplos que mostram os conceitos de herança, polimorfismo, abstração e encapsulamento.

Tópicos abordados neste módulo

- Conceitos de programação orientada a objetos
- Classes e objetos em Java
- Abstração, herança, polimorfismo e encapsulamento
- Criação e destruição de objetos
- Pacotes
- Modificadores de acesso e qualidade

Índice

1.1.Programação orientada a objetos....2
Porque usar objetos?....3
Elementos do modelo orientado a objetos....3
O que é um objeto?....7
Métodos internos e variáveis públicas....10
O que são classes?....11
Herança....13
Exercícios....15
1.2.Classes e objetos em Java....16
Variáveis....18
Métodos....20
Criação de Objetos....22
Construtores....25
Exercícios....28
Variáveis e Métodos de Classe....29

Copyright © 1999, Helder L. S. da Rocha. Todos os direitos reservados. EP:SF-Rio/D99

Exercício 30
Referências e Apontadores 31
Destruição de Objetos 36
Exercícios 37
1.3.Herança, Classes Abstratas e Interfaces 37
Pacotes 38
Pacotes, diretórios e CLASSPATH 41
Modificadores de Acesso 43
Sobreposição de métodos ocultação de dados 45
Invocação de métodos e ligação dinâmica 46
Quando não estender uma classe? 47
Uma Aplicação Orientada a Objetos 50
Classes, métodos e variáveis finais 59
Classes e métodos abstratos 60
Interfaces 61
Conversão e Coerção 66

Objetivos

No final deste módulo você deverá ser capaz de:

- saber quais são as principais características de linguagens orientadas a objetos
- Saber como se cria uma classe em Java
- Saber como se cria objetos
- Saber criar métodos, variáveis, construtores
- Saber definir métodos abstratos, classes abstratas, interfaces, métodos finais e classes finais.
- Saber criar constantes
- Saber o que são e quando usar Interfaces.

1.1. Programação orientada a objetos

O termo "orientado a objetos" tem sido tão usado e abusado que o seu sentido concreto se perdeu. Tem caracterizado desde modelos de administração a métodos de auto-ajuda. Muitos produtos de software que alegam ser orientados a objetos não o são. Às vezes possuem alguns recursos baseados em objetos, outras vezes nem isso. Toda essa confusão de conceitos fez com que ninguém hoje soubesse mais o que era realmente ser "orientado a objetos".

Java é uma linguagem orientada a objetos e para entender Java é necessário compreender os conceitos da programação orientada a objetos ("POO"). O

Copyright © 1999, Helder L. S. da Rocha. Todos os direitos reservados. EP:SF-Rio/D99

objetivo desta seção é esclarecer o significado da POO, apresentando os principais conceitos por trás deste paradigma. Procuraremos usar uma linguagem clara e ilustrar os conceitos através de exemplos do mundo real e em seguida com exemplos de programação. A aplicação desses conceitos na programação em Java será apresentada posteriormente nas seções seguintes.

Apesar de ser um assunto muito badalado, o conceito de POO não é novo. Tem mais de trinta anos de idade: foi introduzido pela primeira vez na linguagem *Simula-67*. Desde então têm surgido várias outras linguagens com esses conceitos como *Smalltalk* e *Eiffel*. C++, a mais popular delas, é na verdade uma linguagem híbrida. É uma extensão do C com suporte a classes e objetos, mas permite que se desenvolva programas sem classes e objetos. *Perl 5* é outra linguagem híbrida. Java, por sua vez, é orientada a objetos desde o princípio. É *impossível* escrever um programa em Java sem usar objetos.

Porque usar objetos?

As vantagens de se usar objetos como blocos para a construção de aplicações são muitas. Podemos citar:

- *Simplicidade:* os objetos escondem a complexidade do código. Pode-se criar uma complexa aplicação gráfica usando botões, janelas, barras de rolamento, etc. sem conhecer complexidade do código utilizado para criá-los.
- *Reutilização de código:* Um objeto, depois de criado, pode ser reutilizado por outras aplicações, ter suas funções estendidas e ser usado como bloco fundamental em sistemas mais complexos.
- *Inclusão Dinâmica:* objetos podem ser incluídos dinamicamente no programa durante a execução. Isso permite que vários programas compartilhem os mesmos objetos e classes, reduzindo o seu tamanho final.

Elementos do modelo orientado a objetos

Os autores divergem quanto às características que fazem uma linguagem ser orientada a objetos, mas a maioria concorda que o paradigma se baseia em quatro princípios básicos: *abstração*, *encapsulamento*, *herança* e *polimorfismo*. Todas as características se relacionam entre si. Explicaremos cada uma delas a seguir.

Copyright © 1999, Helder L. S. da Rocha. Todos os direitos reservados. EP:SF-Rio/D99

Abstração

É o processo de extrair as características *essenciais* de um objeto real. A abstração é necessária para se ter um modelo fiel da realidade sobre o qual se possa operar. O conjunto de características resultante da abstração forma um *tipo de dados abstrato* com informações sobre seu estado e comportamento.

A abstração nem sempre produz os mesmos resultados. Depende do *contexto* onde é utilizada. Ou seja, a abstração de um objeto por uma pessoa pode ser diferente na visão de outra. Por exemplo, para um mecânico, um carro é caracterizado pela marca, modelo, potência, cilindradas, etc. Já um usuário comum pode caracterizá-lo pela cor, marca, modelo, consumo e preço. E finalmente, para o *Departamento de Trânsito*, um carro pode ser totalmente caracterizado pela placa, proprietário, marca, modelo, cor e número de chassi.

Vamos supor que você vai escrever um aplicativo para uma transportadora que trabalha exclusivamente para uma livraria e cobra por livro, apesar do custo ser baseado no peso. É possível ter uma boa noção do peso de um livro sabendo o número de páginas, o formato e o tipo de capa (dura ou mole). Portanto, você poderá criar um objeto `Livro` que tenha essas três características. Mas se você também for escrever um aplicativo semelhante para a livraria, não poderá aproveitar o mesmo objeto `Livro` que você criou pois a abstração será diferente. Para a livraria, interessam características como autor, título, assunto, editora, preço, etc. que não interessam à transportadora.

Encapsulamento

É o processo de combinar tipos de dados, dados e funções relacionadas em um único bloco de organização e só permitir o acesso a eles através de métodos determinados. O encapsulamento permite que o usuário do objeto o utilize através de uma *interface pública* bem definida onde todas as funções disponíveis são completamente especificadas.

Por exemplo, existe um número determinado de coisas que se pode fazer com um toca-fitas. Pode-se avançar, voltar, gravar, tocar, parar, interromper e ejetar a fita. Dentro do toca-fitas, porém, há várias outras funções sendo realizadas como acionar o motor, desligá-lo, acionar o cabeçote de gravação, liberar o cabeçote, e outras operações mais complexas. Essas funções são *escondidas* dentro do mecanismo do toca-fitas e não temos acesso a elas diretamente. Quando apertamos o "play" o motor é ligado e o cabeçote de

Copyright © 1999, Helder L. S. da Rocha. Todos os direitos reservados. EP:SF-Rio/D99

reprodução acionado, mas não precisamos saber como isto é feito para usar o toca-fitas. A interface pública do toca-fitas, constituída pelos botões, esconde a complexidade do aparelho, tornando o seu uso mais simples e mais seguro.

Uma das principais vantagens do encapsulamento é esconder a complexidade do código. Outra, é proteger os dados. Ao permitir o acesso a eles apenas através de métodos de uma interface pública evita que os dados sejam corrompidos por aplicações externas.

No exemplo do toca-fitas, quando usamos o "método" *gravar*, sabemos que o aparelho irá gravar informações na fita em um formato padrão que poderá ser lido por outros aparelhos similares. Se não existisse o método *gravar* e tivéssemos acesso direto à fita, cada pessoa poderia estabelecer uma maneira diferente de gravar informações (usando laser, imãs, etc.). Gravar uma fita passaria a ser uma tarefa complicada, pois o usuário teria que conhecer toda a complexidade do aparelho. Depois, se perderia a compatibilidade pois não haveria uma forma de proteger os dados para que fossem sempre gravados em um formato padrão. O encapsulamento, então, funciona tanto para proteger os dados como para simplificar o uso de um objeto.

Há vários exemplos na programação. Os aplicativos que conhecemos e que rodam no Windows tem uma grande semelhança porque usam os mesmos objetos. Cada objeto tem um número limitado de funções (métodos) que podem ser invocados sobre ele. Se você decide usar, em alguma parte do seu programa, dois botões: **OK** e **Cancela**, você não precisa entender como um botão foi escrito, nem saber quais as suas variáveis internas. Basta saber como construí-lo, mudar o texto que contém e depois incluí-lo em seu programa. Não há como corromper a estrutura interna do botão.

Herança

É o aproveitamento e extensão das características de uma classe existente.

A melhor analogia neste caso é a natureza. No reino animal, os mamíferos herdam a característica de terem uma espinha dorsal por serem uma subclasse dos vertebrados. Acrescentam a essa característica, várias outras, como ter sangue quente, amamentar, etc. Os roedores herdam todas as características dos vertebrados e mamíferos e acrescentam outras, e assim por diante.

Copyright © 1999, Helder L. S. da Rocha. Todos os direitos reservados. EP:SF-Rio/D99

Em programação, a herança ocorre quando um objeto aproveita a implementação (estruturas de dados e métodos) de um outro tipo de objeto para desenvolver uma especialização dele. *Tipo de objeto* é um termo que já usamos antes. Refere-se a uma característica comum de um grupo ou classe de objetos. Em POO, tipos de objetos são definidos por um protótipo chamado de "classe". A herança permite a formação de uma hierarquia de classes, onde cada nível é uma especialização do nível anterior, que é mais genérico.

É comum desenvolver estruturas genéricas para depois ir acrescentando detalhes em POO. Isto simplifica o código e garante uma organização maior a um projeto. Também favorece a reutilização de código. Se precisamos implementar um botão cor-de-rosa e já temos uma classe que define as características de um botão cinza, podemos fazer com que nossa implementação seja uma "subclasse" de `Botão_Cinza`, e depois estender suas características. Desta forma, nos concentramos apenas no necessário (a cor do botão) e evitamos ter que definir tudo a partir do zero.

Polimorfismo

É a propriedade de se utilizar um mesmo nome ou forma para fazer coisas diferentes.

Por exemplo, a ordem: "vire à esquerda!" pode produzir resultados diferentes dependendo da situação. Se quem recebe a ordem estiver dirigindo um automóvel, irá mover a direção para a esquerda; se estiver controlando o leme de uma lancha, moverá a cana do leme para a direita.

Em programação, uma mesma mensagem poderá provocar um resultado diferente, dependendo dos argumentos que foram passados e do objeto que o receberá. Por exemplo: o envio de uma instrução "`desenhe`" para uma *sub*classe de "`polígono`", poderá desenhar um triângulo, retângulo, pentágono, etc. dependendo do objeto que receber a instrução.

Isto é muito útil para escrever programas versáteis, que possam lidar com vários tipos diferentes de objetos. Por exemplo, pode-se projetar um `Posto` para abastecer qualquer tipo de transporte. Na classe `Transporte`, poderíamos definir um método "abastece" genérico. Suponha agora que `CarroElétrico`, `Jegue`, `Porsche` e `AirBus` sejam subclasses de `Transporte`. Cada um tem sua própria maneira de se abastecer (bateria, ração, gasolina ou querosene), mas a única coisa

Copyright © 1999, Helder L. S. da Rocha. Todos os direitos reservados. EP:SF-Rio/D99

que o `Posto` precisa fazer, é acionar o método "`abastece`" em cada um deles. Não precisa saber como é feito. Os detalhes de cada procedimento ficam com o objeto.

O tipo de polimorfismo do exemplo acima chama-se *sobreposição de métodos*, pois cada subclasse *sobrepôs* o método abastece definido na classe `Transporte`, e o redefiniu com uma nova implementação.

A vantagem desse tipo de polimorfismo em programação é a possibilidade de desenvolver programas genéricos que utilizam uma interface padrão. Por exemplo, criar um programa que opere sobre objetos do tipo `Transporte` que possuam um método abastece. Mesmo que um novo tipo de transporte seja inventado após a criação do programa, se este novo tipo estender a classe `Transporte` ele será compatível com o nosso programa. Em Java, inclusive, é possível implementar polimorfismo sem herança através de classes especiais chamadas de *interfaces*. Uma interface não tem métodos implementados, apenas suas assinaturas (tipo de retorno, nome e argumentos). Cada classe que a estende é obrigada a implementar cada método nela definido.

Outro tipo de polimorfismo é o que se chama de *sobrecarga de nomes*. Na sobrecarga usa-se o mesmo nome e mais alguma característica para se fazer a mesma coisa.

Suponha que existe um carro `Híbrido` que usa gasolina e eletricidade. Quando ele chega no posto, o seu método `abastece()` é chamado e ele enche as suas duas fontes de energia. Suponha que ele pode também ter esse método invocado pelo `Posto` com argumentos diferentes como `abastece (`*`qual_fonte`*`)`. Então se o posto aciona o método `abastece(gasolina)` em `Híbrido`, ele preenche somente o seu tanque de gasolina e não recarrega as baterias. Então, dependendo da chamada `abastece()` ter ou não argumentos, o abastecimento também poderá ocorrer de forma diferente.

O que é um objeto?

Objetos são a base da tecnologia orientada a objetos. Consistem de modelos (abstrações) de objetos reais[1]. Eles preservam as características essenciais de um objeto real: o seu *estado* e o seu *comportamento*.

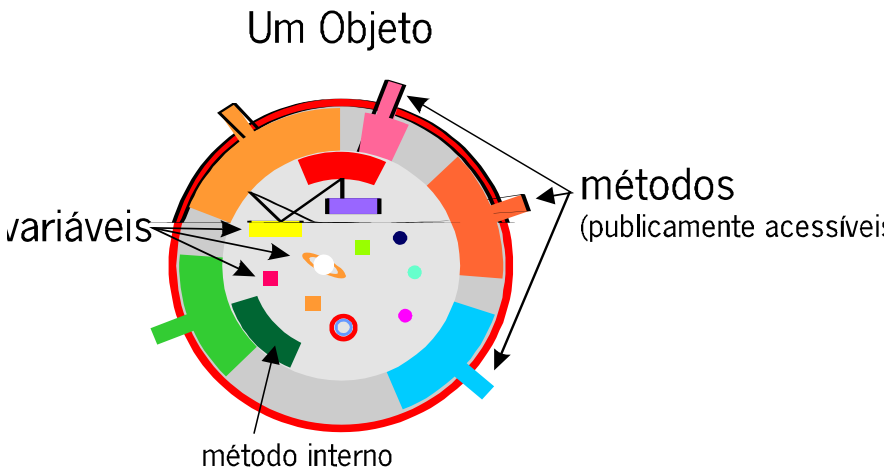
[1] Objetos também podem ser usados para modelar conceitos abstratos, como eventos. Um evento é um objeto usado em sistemas de janela para representar as ações do usuário (clíques do mouse , teclas digitadas).

Copyright © 1999, Helder L. S. da Rocha. Todos os direitos reservados. EP:SF-Rio/D99

Qualquer objeto real pode ser abstraído para filtrar seu estado e comportamento. Vejamos alguns exemplos:

OBJETO	ESTADO	COMPORTAMENTO
Carro	*velocidade atual, marcha, cor, modelo, marca*	*troca marchas, acelera, dá partida, freia*
Gato	*nome, raça, com fome, com preguiça*	*mia, dorme, se estica, brinca, caça, liga o "motor"*
Caneta	*cor, nível de tinta*	*escreve, coloca mais tinta*
Toca-fitas	*ligado, girando, sentido da fita, tocando, gravando*	*grava, toca, avança, volta, para, pausa, ejeta, liga*

Em programação orientada a objetos, o estado de um objeto é representado por *campos de dados* (variáveis), e seu comportamento é implementado com *métodos* (funções). A única forma de mudar o estado dos dados é através de métodos[2]. Por exemplo, para mudar o estado do toca-fitas para ligado, é necessário utilizar o método "liga". A figura abaixo representa um objeto de software:



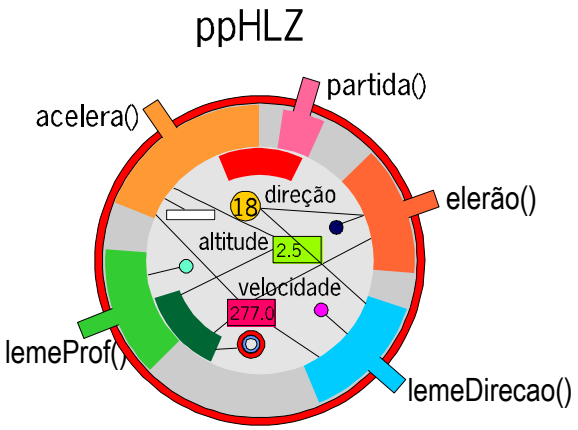
Tudo o que o objeto de software sabe (seu estado) e tudo o que ele pode fazer (o seu comportamento) é determinado pelos seus campos de dados e pelos seus métodos.

Vamos representar o modelo de um avião *Cessna 182*, através de um objeto de software. As variáveis indicam o seu estado atual: `velocidade = 277km/h`, `direção = 18`, `altitude = 2,6km`. `Partida()`, `acelera()`, `lemeDirecao(`*`direção`*`)`, `elerao(`*`direção`*`)`, `lemeProfund(`*`posição`*`)` e `altímetro()` são métodos que podem mudar o estado das variáveis fazendo o

[2] A maioria das linguagens (inclusive Java) permite a declaração de variáveis públicas, que não precisam respeitar ao encapsulamento.

Copyright © 1999, Helder L. S. da Rocha. Todos os direitos reservados. EP:SF-Rio/D99

avião ganhar altitude, velocidade ou mudar de direção ou fornecer informações importantes. O diagrama a seguir ilustra o objeto, que chamamos de `ppHLZ`:

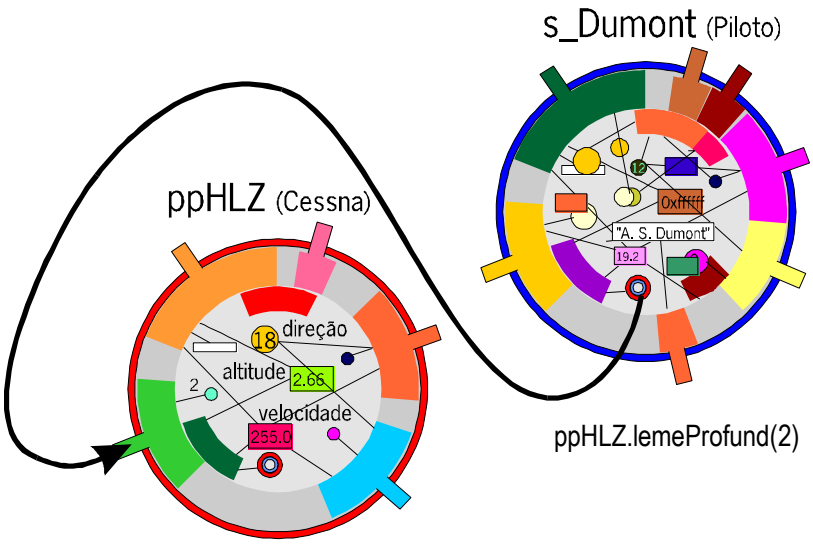


Um objeto sozinho não tem muita utilidade. Um avião sozinho no hangar e sem piloto só serve de enfeite. Para se conseguir alguma funcionalidade dos objetos, é necessário que haja uma interação com outros objetos. Só assim, se pode construir modelos de comportamento mais complexo.

Objetos de software se comunicam entre si através de mensagens. Como não temos acesso às variáveis a não ser através de métodos, a troca de mensagens consiste da invocação de métodos e o possível retorno de valores. No exemplo do `Cessna`, é necessário que haja um outro objeto, um `Piloto`, que envie mensagens para o avião para que ele possa sair do chão.

A figura abaixo ilustra esta situação. O avião voa para o Sul (180º) a 277km/h e 2,5km de altura. O objeto `s_Dumont` (um certo piloto) invoca o método `elevador()` sobre `ppHLZ` com o parâmetro `+2`. O método é então executado, provocando o movimento do elevador para cima duas posições, aumentando a altitude e reduzindo a velocidade (mudando o estado das variáveis de `ppHLZ`).

Copyright © 1999, Helder L. S. da Rocha. Todos os direitos reservados. EP:SF-Rio/D99



Para realizar a mudança acima em Java, dentro do código do objeto `s_Dumont` haveria uma linha do tipo:

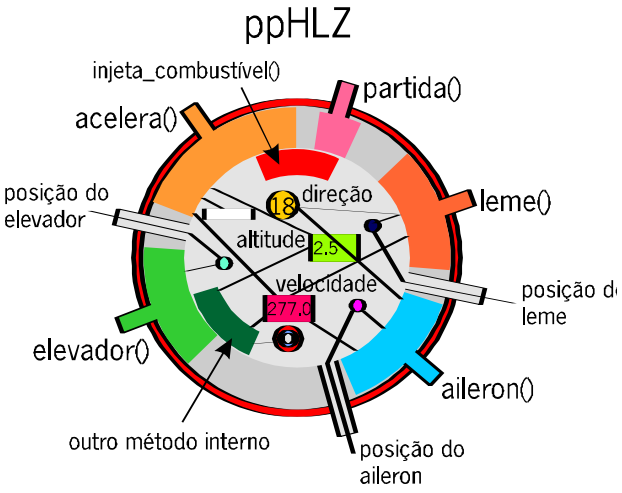
```
ppHLZ.lemeProfund(+2)
```

É assim que Java representa a chamada do método de um objeto. O primeiro nome (`ppHLZ`) é o objeto. Após o ponto, vem o nome de um método com seus atributos (se existirem). No exemplo acima, o método é `lemeProfund()`. Todos os métodos em Java consistem de um nome seguido de `()`, mesmo que não aceitem parâmetros.

Métodos internos e variáveis públicas

Observe que a única maneira de se mudar qualquer estado do avião é através dos seus *métodos públicos*. Os campos de dados são privativos e o seu acesso direto não é permitido ao mundo externo. Se os campos de dados fossem públicos, poderíamos alterá-los diretamente, comprometendo o funcionamento normal do objeto. A figura a seguir ilustra uma versão mais complexa e menos segura do `Cessna ppHLZ`, onde é possível alterar os valores das posições do leme de direção, elerões e leme de profundidade sem usar métodos.

Copyright © 1999, Helder L. S. da Rocha. Todos os direitos reservados. EP:SF-Rio/D99



Isto é como mover os lemes e elerões com a mão em vez de usar o manche e o pedal! Torna o avião mais difícil de usar e compromete a sua segurança. A maioria das linguagens de programação, inclusive Java, permitem que se use campos de dados públicos. Geralmente as desvantagens desse procedimento são maiores que as vantagens, portanto, deve-se usar o encapsulamento dos dados pelos métodos o quanto possível.

Um objeto complexo poderá ter centenas de métodos, mas talvez somente uns quatro ou cinco interessem a quem vai usá-lo. Quem dirige um carro não precisa saber de coisas como injetar combustível, provocar faísca, etc. Simplesmente ele liga a ignição. Neste caso, ligar a ignição é um método público, que interessa ao motorista. Já injetar combustível, provocar faísca, são métodos internos (ou privados) ao funcionamento do automóvel, que podem ser chamados por outros métodos sem que o motorista precise se preocupar.

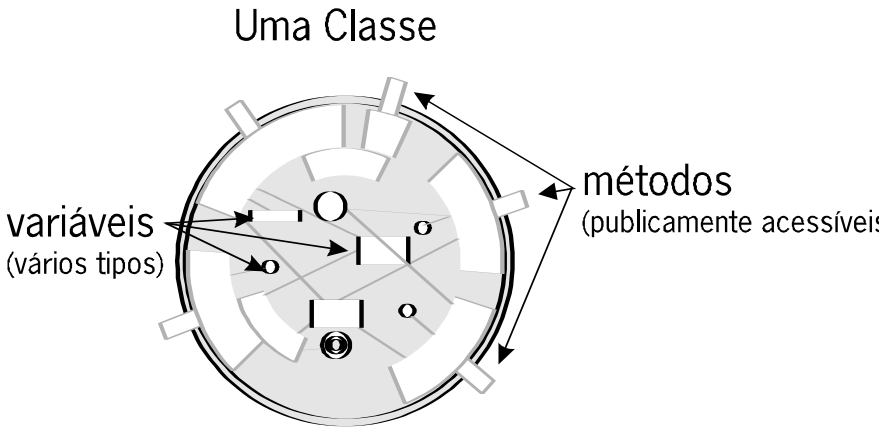
O que são classes?

Classes são protótipos utilizados para construir objetos. As características de estado e comportamento do objeto `ppHLZ`, ilustrado nos exemplos anteriores, são características de qualquer avião Cessna 182RG que sai da fábrica. Estas características são definidas nas classes, que podem ser vistas como a *planta* usada para construir o avião.

A figura abaixo representa uma classe. Observe a semelhança com o diagrama que utilizamos para representar os objetos. A classe não é um objeto pois não possui um estado, nem um comportamento específico. Ela não é algo

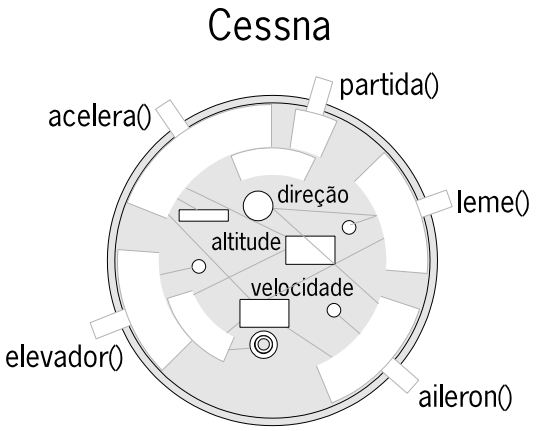
Copyright © 1999, Helder L. S. da Rocha. Todos os direitos reservados. EP:SF-Rio/D99

que existe e que possa ter estados. Por isso, aparece no diagrama como um fantasma, sem cores, pois representa apenas um gabarito com o qual podem ser construídos objetos.



Com as plantas que contém o projeto de um Cessna, pode-se construir vários aviões Cessna. O projeto inclui as especificações do motor, da fuselagem, da asa, vários desenhos de perspectiva, moldes e até uma maquete em miniatura. Descreve em detalhes tudo o que o Cessna pode fazer. Como vão funcionar os lemes, os elerões, o trem de pouso, etc. Contém tudo o que um Cessna tem mas não é um Cessna.

Para produzir os aviões `ppHLZ`, `ptANX` e `ptGRN`, todos do modelo Cessna 182, definimos uma classe chamada `Cessna`. Esta classe possui a declaração das variáveis que serão utilizadas para armazenar a direção, velocidade e altitude, além da implementação dos métodos que cada Cessna deve ter.



Copyright © 1999, Helder L. S. da Rocha. Todos os direitos reservados. EP:SF-Rio/D99

Os objetos são chamados de *instâncias* das classes. Quando criamos um objeto, dizemos que instanciamos uma classe. Cada instância é uma entidade individual e pode então receber valores para as suas variáveis e modificá-las através de métodos. Com uma planta, fazemos 100 aviões, que podem então ter velocidades, altitudes e direções diferentes.

Tendo criado a classe `Cessna`, podemos criar agora vários aviões com as mesmas características básicas. Por exemplo, se `ppHLZ`, `ptANX` e `ptGRN` são instâncias de `Cessna`, todos têm uma velocidade, uma altitude e uma direção distintas e podem dar partida, acelerar, subir, descer, etc. `ppHLZ` pode estar a 350km/h voando para o Norte, enquanto `ptANX` pode estar estacionado no hangar.

A grande vantagem de se usar classes é a reutilização do código. Toda a estrutura de um objeto é definida na classe, que depois é utilizada quantas vezes for necessário para criar vários objetos. O próprio avião pode ser construído a partir de objetos menores. Por exemplo, os objetos `tanque`, `hélice`, `motor`, `roda` podem ser peças que a fábrica de aviões já compra pronta de outra fábrica. A fábrica de aviões não precisa saber como faz uma hélice e nem precisa ter o molde ou planta que explica como fazê-la. Simplesmente usa o objeto. Já uma fábrica de hélices, possui uma classe `Hélice`, conhece os detalhes de uma hélice, faz centenas delas e as vende para fábricas de aviões.

Com orientação a objetos, o programador se preocupa com o objetivo principal do programa que quer desenvolver. Não fica preocupado com detalhes como programação de botões, molduras, interfaces gráficas, hélices, etc. Usa objetos pré-fabricados para montar a infraestrutura do seu projeto e se concentra na implementação específica do problema que pretende resolver.

Herança

Não bastasse a reutilização do código proporcionado pelas classes, na criação de objetos, a POO vai mais longe e define meios de permitir em uma classe, a reutilização do código de outras classes.

As características cor, velocidade, marcha, etc. podem estar definidas em uma classe chamada de `Avião`, que declara métodos `elevador()`, `partida()`, `leme()`, etc. Esses métodos e variáveis podem ser herdadas pelas classes `AHélice` e `AJato` que são tipos de aviões. A classe `Cessna` pode ser uma subclasse de

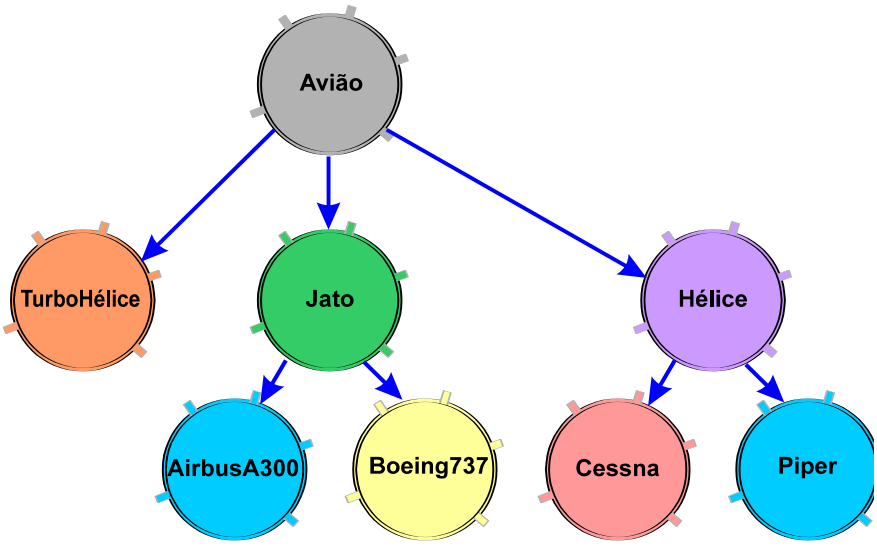
Copyright © 1999, Helder L. S. da Rocha. Todos os direitos reservados. EP:SF-Rio/D99

`AHélice` e a classe `AJato` pode ter outras subclasses como `F15`, `Boeing` e assim por diante.

Em cada classe, métodos novos podem ser criados, outros podem ser redefinidos e novos campos de dados podem ser declarados. Cada subclasse adiciona alguma nova funcionalidade à classe pai e continua sendo um objeto do tipo definido pela raiz das classes. Ou seja, tanto `Cessna`, `AirbusA300`, e `Boeing737` são aviões. `Cessna` e `Piper` são aviões e são movidos a hélice.

É importante observar esta relação de "ser" no projeto de sistemas orientados a objetos. Por exemplo, pode-se criar uma classe `Esfera` que estende `Círculo` e redefine alguns de seus métodos. `Esfera` e `Círculo` têm várias semelhanças, mas uma esfera *não é* um círculo e poderá haver problemas em uma fase posterior da implementação quando tentarmos passar uma esfera para um objeto que lida com formas bi-dimensionais.

A figura abaixo ilustra uma hipotética hierarquia de classes formada pelas subclasses da classe `Avião`.



Todos os métodos e dados públicos de uma superclasse são herdados pelas suas subclasses. Desta forma, se `Avião` já define o método `partida()`, não há necessidade de `Cessna` redefini-lo. Os objetos criados com a classe `Cessna` já nascem com um método `partida()`, mesmo que este não faça parte da planta do avião.

Mas, na maioria das vezes, a maneira como uma classe genérica define uma ação não serve para classes mais específicas. Provavelmente a partida na classe

Copyright © 1999, Helder L. S. da Rocha. Todos os direitos reservados. EP:SF-Rio/D99

`Avião` é manual. Neste caso, a subclasse pode sobrepor ou sobrecarregar os métodos herdados. O `Cessna`, por exemplo, pode ou definir um novo método `partida(tipo)` que suporte tanto partida manual como elétrica, mantendo o método `partida()` para acionamento manual (*sobrecarregando* o método), ou simplesmente rescrever `partida()` para que faça somente partida elétrica (*sobrepondo* o método).

`Cessna`, por exemplo, pode acrescentar novos métodos como `baixarTrem()` e `horizonte()` (para ler as indicações do horizonte artificial). `Jato` deve sobrepor o métodos `partida()`, `elevador()`, `leme()` com uma implementação específica para um jato genérico. Talvez acrescente `flaps()`, `pilotoAut()`. `Boeing737` também pode acrescentar novas variáveis `capacidadeDeCarga`, `númeroDePassageiros`, etc. para caracterizar melhor os objetos que serão produzidos.

Exercícios

Nas seções seguintes utilizaremos exaustivamente os conceitos aqui apresentados que são básicos para a linguagem Java, portanto a sua compreensão é indispensável. Os exercícios a seguir ajudarão a fixá-los.

1. Faça a representação (estado e comportamento) de um objeto do tipo FerrariF40. Liste alguns métodos e campos de dados e represente-o através de um diagrama.
2. Mostre como um objeto do tipo Motorista poderia invocar alguns dos métodos de FerrariF40 para alterar o seu estado.
3. Como poderíamos construir uma FerrariF40? Liste alguns objetos que poderiam ser utilizados como parte de uma Ferrari como rodas, bancos, motor, freios. Represente-os esquematicamente e liste os seus métodos. Mostre como a Ferrari poderia usar os seus componentes (como a Ferrari poderia invocar métodos de seus componentes. Ex: motorista aciona ferrari.partida() que aciona motor.partida())
4. Construa uma hierarquia de classes para Veículos Motorizados, que poderia ter como uma de suas subclasses, a classe Automóvel. Represente as outras classes e encaixe FerrariF40 em algum lugar da hierarquia. Quais métodos (e campos) poderiam ser definidos na classe raiz da hierarquia. Quais deveriam ser sobrepostos.

Copyright © 1999, Helder L. S. da Rocha. Todos os direitos reservados. EP:SF-Rio/D99

5. Represente uma classe HondaTitan125 na hierarquia de classes do exercício anterior. Quais os métodos que deveriam ser sobrepostos (ou definidos) para motos? Em que classes da hierarquia devem ser sobrepostos cada método?
6. Identifique os casos onde está havendo abuso da reutilização do código, nas subclasses que estendem as superclasses abaixo: 1) Esfera estende Circulo; 2) Círculo estende Elipse; 3) Globo estende Esfera; 4) TV estende RádioFM; 5) TVsemImagem estende TV; 6) Homem estende Macaco; 7) Homem estende Primata. (Dica: verifique se a subclasse "é" um objeto do tipo da superclasse ou se apenas "parte" dele está na superclasse).

1.2. Classes e objetos em Java

Como vimos na seção anterior, uma classe é um *gabarito* que define os dados e métodos comuns a todos os objetos do seu tipo. Desta forma, a classe caracteriza uma qualidade, ou *tipo* de um objeto.

A classe, em Java, é um *tipo de dados abstrato* que pode ser definido pelo usuário. É, na verdade, algo mais que isto. Uma classe consiste de declarações de *campos de dados* e *métodos*. Os campos de dados são variáveis, referências ou constantes e os métodos contém o código com as instruções que operam sobre os dados da classe ou dos objetos criados através dela. Esses dois ingredientes, campos de dados e métodos, permitem a criação de objetos com um certo *estado* (descrito nas variáveis) e *comportamento* (definido pelos métodos).

A forma mais simples de se criar uma classe em Java é usando a palavra-chave `class` seguida por um identificador que serve de nome para a classe. O corpo da classe é totalmente delimitado por chaves, e contém as declarações dos campos de dados e métodos. A ordem dos fatores aqui não é importante.

```
class Nome_da_classe {
 declaração/definição de campos (...)
 declaração/definição de métodos (...)
}
```

Por exemplo, a classe `Cessna`, que representamos esquematicamente no capítulo 2, poderia ser definida inicialmente da seguinte forma:

```
class Cessna {
 private int direção;
 private float altitude;
 private float velocidade;
```

Copyright © 1999, Helder L. S. da Rocha. Todos os direitos reservados. EP:SF-Rio/D99

```
    public void lemeProfund(int posição) {
        // instruções para mudar
        // posição do elevador
    }
    public void lemeDirecao(int posição) {
        // instruções para mudar
        // posição do leme
    }

    public boolean partida() {
        // instruções para dar partida
           return true; // partida foi dada
    }

    public float altimetro() {
           return altitude;
    }
}
```

Nesta definição há três variáveis de *instância* (variáveis declaradas no corpo da classe que não são `static` só podem ser usadas por objetos, e são chamadas *de instância*) e quatro métodos. As variáveis são privativas à classe, ou seja, não são acessíveis quando se utiliza um objeto do tipo `Cessna`. A única forma de se obter um registro da altitude do avião, por exemplo, é através da invocação do método `altímetro()`, que, como os outros métodos desta classe, é publicamente acessível.

Observe que, neste modelo, não é possível para um outro objeto (do tipo `Piloto`, por exemplo) saber a velocidade do avião criado com esta classe, pois não criamos um método `velocímetro()` que retornaria o valor da variável `velocidade`. Só são visíveis fora da classe os quatro métodos.

É ainda possível usar muitos outros métodos não definidos nesta classe como `equals()`, `clone()`, etc. De onde eles vêm? Da superclasse `Object`! Veremos mais sobre extensão de classes no capítulo seguinte, mas é importante lembrar (do capítulo 3) que toda classe em Java é a extensão de uma classe de nível mais alto na hierarquia. Se esta classe não é especificada (como nos nossos exemplos até agora), ela é subclasse da classe `Object`, que ocupa o nível mais alto da hierarquia e faz parte da biblioteca fundamental da linguagem Java (não precisa ser importada).

Copyright © 1999, Helder L. S. da Rocha. Todos os direitos reservados. EP:SF-Rio/D99

Variáveis

Toda declaração de variável obrigatoriamente deve informar o seu tipo. No exemplo acima, temos uma variável do tipo `int` e duas do tipo `float`. Os métodos, por sua vez, devem informar o *tipo que retornam* e caso não retornem valor algum devem ser declarados como `void`. É o caso do método `leme()`, no exemplo acima, que só trata de mudar a posição do leme e não retorna informação alguma.

A sintaxe geral para a declaração de variáveis é a seguinte:

```
[modificadores] tipo nome_da variável;
```

Os modificadores são opcionais. Podem modificar a visibilidade (acesso) de uma variável ou a sua qualidade. Os modificadores de visibilidade são: `public`, `private` e `protected`, ou a ausência de um deles. Esses modificadores definem que outras classes e objetos podem acessar a variável. As variáveis do nosso exemplo foram declaradas `private`, portanto não são visíveis de fora do objeto, a não ser que, como fizemos com a variável altitude, o seu valor seja repassado através de um método público.

Os modificadores de qualidade para variáveis são:

- **`final`**: define uma constante. Na declaração de uma constante, o seu valor já deve ser definido. Por exemplo, poderíamos ter em um programa que calcula órbitas de satélite, a gravidade da Terra ao nível do mar em m/s:

  ```
  public final g = 9.810;
  ```

- **`static`**: usado para declarar variáveis estáticas ou globais que são acessíveis somente através da classe e independente da criação de objetos. As variáveis estáticas são alocadas na hora em que a classe é carregada. Ocupa uma dimensão a parte da dimensão dos objetos. Se uma variável é declarada estática, só há uma cópia dela independente do número de objetos criados. Cada novo objeto criado tem acesso a esta variável e ela pode ser usada como forma de definir valores globais visíveis entre objetos.

Copyright © 1999, Helder L. S. da Rocha. Todos os direitos reservados. EP:SF-Rio/D99

- **`transient`:** usado em serialização de dados para informar que a variável não deve ser salva como parte integrante do objeto. Não vale para campos estáticos.
- **`volatile`:** usado para informar o compilador que ele não deve tentar fazer otimizações com esta variável. Vale para campos usados em linhas de execução (*threads*) sincronizadas.

Quanto aos modificadores de qualidade, no momento só nos interessam dois (os outros veremos mais na frente):

- **`public`:** torna a variável acessível de fora do objeto, ou seja, qualquer outro programa pode mudar o seu valor usando a notação:

  ```
  objeto.variável
  ```

- **`private`:** torna a variável inacessível de fora da classe ou do objeto.

Na grande maioria dos casos, um projeto não deve ter variáveis públicas. Sempre que possível, o estado de um objeto só deve ser modificável através de um método. Isto garante a integridade dos dados evita a corrupção de dados ou o mau uso de um objeto. Por exemplo, suponha que alguém criasse um círculo e alterasse seu raio:

```
Círculo c1 = new Círculo();
c1.raio = -5;
```

Não existem círculos com raios negativos. O programa que desenha o mesmo iria falhar por causa disto. Uma forma de evitar o problema é declarar raio como `private` e fornecer um método para alterar o seu valor e outro para permitir a leitura:

```
public double raio() {
   return raio;
}

public void mudaRaio(double novoRaio) {
   if (novoRaio != 0.0) {
       raio = Math.abs(novoRaio);
   } else {
       raio = 1.0;
   }
}
```

Esta mudança fará com que a única forma de mudar o raio seja através de um método:

Copyright © 1999, Helder L. S. da Rocha. Todos os direitos reservados. EP:SF-Rio/D99

```
c1.mudaRaio(-5);
```

O método fará com que e o sinal seja sempre ignorado e que raios nulos tenham pelo menos o valor 1.0, protegendo o objeto.

Métodos

A sintaxe geral para a declaração de métodos é a seguinte:

```
[modificadores] tipo_de_retorno nome_do_método ([tipo argumento,
tipo argumento, ...]) [throws Exceção] { ... }
```

O que está entre `[` e `]` é opcional. Os modificadores podem modificar a visibilidade (acesso) de um método ou a sua qualidade. Os modificadores de visibilidade novamente são: `public`, `private` e `protected`, que definem que outras classes e objetos podem acessar o método. Todos os métodos do nosso exemplo são públicos, por isso são acessíveis de fora.

Veremos os modificadores de visibilidade (ou acesso) em mais detalhes na próxima seção.

Os modificadores de qualidade para métodos são:

- **`final`**: faz com que a implementação do método seja uma versão final e não possa ser sobreposto em uma sub-classe, ou seja, se definíssemos `Cessna` como sub-classe de `Monomotor`, e tivesse um método `altímetro()` definido nesta classe como final, *não* poderíamos redefiní-lo em `Cessna`.

- **`abstract`**: usado para definir um método sem implementação, que necessariamente precisa ser *sobreposto* em uma sub-classe para que possa ser usado. Por exemplo, declarando o método `partida()` como `abstract` em `Avião`, tornará sua implementação obrigatória nas subclasses (todo avião deve ter um meio de dar partida) de uma forma particular a cada caso.

- **`native`**: declara um método que usa uma implementação escrita em outra linguagem (C ou C++, por exemplo).

- **`static`**: usado para declarar um método como pertencente à classe. Este método funciona mais como uma função pois não opera em cima de um objeto. Pode ser chamado através da própria classe (sem precisar

Copyright © 1999, Helder L. S. da Rocha. Todos os direitos reservados. EP:SF-Rio/D99

criar um objeto antes). Assim como as variáveis estáticas, os métodos estáticos estão em uma dimensão diferente daquela dos objetos. Eles agem como se estivessem completamente separados da classe (e podem ser separados sem prejudicar os objetos). Métodos estáticos não podem usar campos de dados ou métodos não-estáticos. O trecho seguinte *não* compila:

```
1. public class Thing {
2.
3.    private int size;
4.
5.    public static void main(String[] args) {
6.       size = 10;          // illegal
7.       System.out.println("Size is " + size);
8.    }
9. }
```

- **`synchronized`:** em programas que usam *multithreading*, este modificador garante que os dados manipulados pelo método não serão modificados por outro método enquanto este não terminar de usá-los.

A cláusula `throws` é usada quando um método causa uma exceção que não é capturada e tratada dentro do mesmo. É necessária esta declaração para que os programas que usem o método possam tentar capturar a exceção ou passá-la adiante.

O nome do método deve sempre ser seguido de parênteses, mesmo se não houver argumentos. Se houver, o tipo deles deve ser declarado. Os argumentos são separados por vírgulas. Toda a implementação do método deve vir entre as chaves { e } a menos que seja declarado `abstract` ou `native`. Neste caso, a declaração deve terminar com um ponto-e-vírgula:

```
public abstract boolean partida();

private native double integral(float expr, double sup, double inf);
```

Exercícios

1. Implemente um método `velocímetro()`, para `Cessna`, que leia a variável `velocidade` e um método `bússola()` que leia a variável `direção`.
2. Crie uma classe `Círculo` que tenha as variáveis `x`, `y` (coordenadas do centro) e `raio`. Todas devem ser públicas e de ponto-flutuante de simples precisão (`float`). Implemente 2 métodos para o `Círculo`: `area()` e

Copyright © 1999, Helder L. S. da Rocha. Todos os direitos reservados. EP:SF-Rio/D99

`circunferência()`. Ambos são públicos e devem retornar os valores como `float`. *Fórmulas do círculo*: Área = π * *raio*^{2}; Circunferência = 2 * π * *raio*.

3. Quais as vantagens e desvantagens de se ter variáveis públicas, como as do exercício 2?
4. Como você escreveria os seguintes tipo de dados abstratos em Java?

   a) Uma casa é uma construção que tem um número, um nome de rua e um proprietário.

   b) Um gato é um animal de estimação que tem um nome, um humano que é seu dono e uma idade.

   c) Uma família tem uma casa, um conjunto de humanos e um animal de estimação.

Criação de Objetos

Objetos são instâncias de uma classe. Não se pode fazer nada com uma classe pois ela só é uma representação genérica de um objeto. A classe é algo imaterial e eterna no "mundo dos objetos", que são "mortais". Os objetos são criados à imagem das classes, mas não podem vê-las. Vivem em outra dimensão. Para podermos usar uma classe como `Cessna` e `Círculo`, precisamos criar instâncias delas, definindo uma determinada aeronave Cessna ou um círculo específico.

Como definimos as classes `Cessna` e `Círculo` (exercício 2) elas agora são tipos em Java, podemos declarar outras variáveis com esse tipo:

```
Cessna ppHLX;
Cessna ptDAT;
Círculo c1;
```

Mas ainda as variáveis `ppHLX`, `ptDAC` e `c1` são apenas nomes que servem de *referência* a objetos `Cessna` ou `Círculo`. Não foi alocada ainda memória na criação de um objeto. Para efetivamente *criar* o objeto, precisamos usar a palavra-chave `new`, chamar uma função que construa o objeto (o construtor) e atribuir o resultado dessa operação à variável declarada como referência:

```
Cessna ppHLX;
Cessna ptDAT;
Círculo c1;
```

Copyright © 1999, Helder L. S. da Rocha. Todos os direitos reservados. EP:SF-Rio/D99

```
ppHLX = new Cessna();
ptDAT = new Cessna("Particular");
c1 = new Círculo();
```

Com esta operação, criamos um objeto `Círculo` e dois objetos `Cessna` e os atribuímos às variáveis `c`, que é do tipo `Círculo` e `ppHLX` e `ptDAT`, que são do tipo `Cessna`, respectivamente.

Observe que após o `new`, existe a chamada à um *método* (parece um método pois vem seguido de um par de parênteses), que tem o mesmo nome que a classe. Este é o *construtor* que define como é construído o objeto. A rigor, um construtor não é um método pois não pode ser chamado em outra ocasião a não ser aquela em que constrói um objeto. Depois, o único valor que retorna é uma referência à instância de um objeto. Construtores não são herdados e em alguns casos podem ser chamados mesmo não aparecem definidos na classe O construtor pode ser definido como uma função de inicialização de objetos.

O construtor sempre tem que ser invocado, mesmo que a classe não o defina. Ele é quem monta a infraestrutura básica do objeto. Como toda classe Java estende outra, o construtor sempre chama um outro construtor definido na sua superclasse. Por isso, quando não são usadas construções complexas, pode-se omitir o construtor já que ele sempre chama o da superclasse. Na nossa classe `Círculo` e na classe `Cessna`, o programa chama o construtor de `Object` na hora em que o seu construtor (invisível) é chamado.

O construtor que usamos é básico. Chama-se construtor *default* e não recebe parâmetros. Ele é fornecido *gratuitamente* pelo sistema. Algumas classes podem definir construtores que recebem parâmetros. Assim, a construção do objeto poderia definir previamente alguns valores (como mostra a criação do segundo `Cessna` acima). Neste caso, o construtor precisa estar definido na classe.

Agora que criamos os objetos, podemos acessar suas variáveis e métodos dentro do nosso programa (pode ser a mesma ou outra classe). Fazemos isto usando um ponto "`.`". Por exemplo, no programa do Piloto, ele pode ter uma instrução que invoca um método do avião:

```
ptDAT.lemeProfund(-2); // move o leme de profundidade para
                       // baixo em 2 posições
```

Suponha que também foi implementado no avião Cessna também um método:

Copyright © 1999, Helder L. S. da Rocha. Todos os direitos reservados. EP:SF-Rio/D99

```
public void acelera(int quantidade) {
    (...)
}
```

onde cada chamada a este método cause o aumento da aceleração do avião (e como conseqüência aumente a velocidade). Uma simples decolagem poderia ocorrer da seguinte forma:

```
class Piloto {
    public static void main(String[] args) {
        // criação do objeto
        Cessna meuAvião = new Cessna();

        // decolagem
        meuAvião.partida();
        meuAvião.acelera(100);
        if (meuAvião.velocímetro() < 70) {
            meuAvião.lemeProfund(+2);
        }
        if (meuAvião.altimetro() > 500) {
            meuAvião.acelera(-20);
        }
    }
}
```

No exemplo do círculo, podemos manipular com as variáveis pois elas são públicas. Sendo assim, podemos definir as coordenadas e o raio do círculo diretamente:

```
c1.x = 2.0f;  // centro em (2,2) e raio 1.
c.y = 2.0f;
c.raio = 1.0f;
```

Agora podemos fazer qualquer coisa que os métodos e variáveis dos objetos permitirem, como por exemplo, calcular a área do círculo:

```
Círculo c2;
c2 = new Círculo();
float a;
c1.raio = 5;
a = c1.area();     // área retorna um valor float
```

Observe que os métodos e variáveis operam em uma instância específica de `Círculo`. Não precisamos passar argumento para `area()` porque sabemos que o objeto sobre o qual operamos é `c1`. É implícito na linguagem que o método opera nas instâncias da classe na qual é definido.

Copyright © 1999, Helder L. S. da Rocha. Todos os direitos reservados. EP:SF-Rio/D99

Vejamos uma possível definição para `area()`:

```
public float area() {
    float a = 3.14159 * raio * raio; // a é variável
    return a;                        // local
}
```

Como é que um método que não recebe argumentos sabe em que dados operar? Na verdade, ele sabe. O argumento está implícito. É o objeto através do qual o método é invocado. Freqüentemente, esta relação é reforçada através da palavra-chave `this`. `this` se refere a "este" objeto, ou seja, a variável `r` utilizada pelo método `public` refere-se à variável `raio` do objeto tipo `Círculo` atual. Quando não aparece explicitamente, é porque geralmente não é necessária, como no exemplo abaixo, e é incluída apenas por clareza. O método `area()` pode ser rescrito da seguinte forma:

```
public float area() {
    return (3.14159 * this.r * this.r);
}
```

O uso de `this`, mesmo quando não obrigatório, ajuda a tornar o código mais claro.

Construtores

Quando criamos um objeto, chamamos uma função que possui o mesmo nome que a classe usada para criá-lo. Esta função especial, como vimos, é chamada de *construtor*.

Todo objeto têm um construtor e ele tem o mesmo nome que a sua classe. Pode ser definido explicitamente, mas se não for, o sistema fornece um construtor *default*. Listamos a seguir, uma implementação da classe `Círculo`, (proposta no exercício 2). Incluímos um construtor para que ele inicialize cada objeto criado já com um valor para o centro e raio:

```
public class Círculo {
 // variáveis de instância
    private float x, y, raio;

 // constante estática pi
    private static final float pi = 3.14159;

    // construtor
    public Círculo(float x, float y, float raio){
```

Copyright © 1999, Helder L. S. da Rocha. Todos os direitos reservados. EP:SF-Rio/D99

```
        this.x = x;
        this.y = y;
        this.raio = raio;
    }

    // métodos
    public float circunferência() {
        return (2 * pi * raio);
    }
    public float area() {
        return (pi * raio * raio);
    }
}
```

Neste caso, o uso de `this`, é obrigatório para evitar a ambigüidade entre as variáveis dos argumentos e as variáveis do objeto (seria dispensável se usássemos nomes distintos). Observe que não é declarado um *tipo de retorno* para o construtor. Ele implicitamente retorna uma instância da classe onde é definido.

Agora podemos criar vários círculos diferentes usando como gabarito a classe `Círculo` em uma única linha:

```
Círculo c1, c2, c3;

c1 = new Círculo(0.0f, 0.0f, 1.0f);
c2 = new Círculo(2.5f, 1.414f, 3.14159f);
c3 = new Círculo(2.0f, 2.0f, 2.0f);
```

Todos os objetos têm seus próprios métodos e variáveis. Uma operação sobre um dos objetos não afeta o outro:

```
float a1 = c1.area();  // a1 = 3.14159;
float a3 = c3.area();  // a2 = 12.56636
```

Múltiplos Construtores e Sobrecarga de Métodos

Podemos ter mais de um construtor para um objeto, assim como podemos ter formas diferentes de chamar um mesmo método. É possível *sobrecarregar* os nomes dos métodos e construtores para que, dependendo do *número* e *tipo* dos argumentos com que são chamados, eles realizem coisas (ou inicializações) diferentes. Isto é muito comum em linguagens orientadas a objetos e caracteriza um tipo de polimorfismo.

Quando criamos o novo construtor para `Círculo`, que necessita obrigatoriamente de três argumentos, *não podemos mais usar* aquele construtor

Copyright © 1999, Helder L. S. da Rocha. Todos os direitos reservados. EP:SF-Rio/D99

simples `Círculo()` a não ser que o redefinamos novamente. O sistema só dá construtores a quem não têm. Podemos fazer isto e ter dois construtores diferentes:

```
public class Círculo {
 // variáveis
    private float x, y, raio;

 // constante estática pi
    private static final float pi = 3.14159;

    // construtor 1
    public Círculo(float x, float y, float raio){
        this.x = x;
        this.y = y;
        this.raio = raio;
    }

    // construtor 2
    public Círculo(){
        this(0.0f, 0.0f, 1.0f);
    }

    // métodos
    public float circunferência() {
        return (2 * pi * raio);
    }

    public double area() {
        return (pi * raio * raio);
    }
}
```

`this(0.0f, 0.0f, 1.0f)` é uma chamada ao construtor *desta* classe que recebe três números de ponto flutuante como argumentos. É o mesmo que chamar `Círculo(0.0f, 0.0f, 1.0f)`. O segundo construtor então faz na verdade uma chamada ao primeiro construtor, com aqueles argumentos. Agora podemos voltar a criar círculos com o construtor simples:

```
Círculo c4 = new Círculo();  // Centro:(0,0); Raio 1
```

Construtor da superclasse

Antes falamos que o construtor chamava sempre o construtor da *superclasse*, mesmo que este não estivesse definido. O construtor chamado é sempre o construtor *default* (sem argumentos). Se estamos estendendo uma classe que tem

Copyright © 1999, Helder L. S. da Rocha. Todos os direitos reservados. EP:SF-Rio/D99

outros construtores, precisamos criar um construtor que realize esta chamada explicitamente. A chamada ao construtor da superclasse é feita através da palavra-chave super, por exemplo

```
public class CírculoOpaco extends Círculo {
 // variáveis
    private float x, y, raio;
    private Color cor;

    // construtor 1
    public CírculoOpaco(){
        super(0, 0, 1);
    }
}
```

O que o construtor *default* (aquele que ninguém vê) é na verdade o seguinte:

```
public SuaClasse(){
        super();
}
```

Se a classe que você estende não tem construtor *default*, você terá um erro de compilação que só será corrigido quando você definir explicitamente o construtor da sua classe.

Exercícios

5. Altere a classe `Círculo` fazendo com que suas variáveis `x`, `y` e `raio` sejam privativas à classe (use o modificador `private`). Agora a única forma de se definir um círculo é através de seu construtor. Crie três novos métodos então que permitam que se leia as coordenadas do centro e o raio.
6. Crie um método que aumente a dimensão do `raio` cada vez que é chamado. O tamanho do incremento deve ser passado como argumento do método.
7. Crie uma aplicação gráfica simples (pode ser baseada no `HelloWorld` Applet primeiro módulo) que desenhe um círculo na tela. O programa deve receber da linha de comando o raio do círculo, criar uma instância e usar os dados obtidos para desenhar o círculo na tela. Use o método `Graphics.drawLine` e a equação do círculo (no início deste capítulo). Você terá que definir coordenadas diferentes de 0,0 pois a origem da aplicação gráfica é seu ponto superior esquerdo.

Copyright © 1999, Helder L. S. da Rocha. Todos os direitos reservados. EP:SF-Rio/D99

Variáveis e Métodos de Classe

Toda variável em Java tem que estar contida dentro uma classe. Também não existem variáveis globais na linguagem. Como então armazenar dados que não sejam alteradas por cada instância de um objeto? Para contar objetos criados, por exemplo?

A solução é usar variáveis estáticas (ou variáveis de classe). Usando o modificador `static` na declaração de uma variável, esta será acessível através da própria classe, independente de objetos.

No exemplo abaixo, utilizamos uma variável de classe para contar o número de vezes que o construtor é acessado e poder estabelecer um número de série para cada Cessna criado. Este valor pode ser usado a qualquer momento para saber quantos aviões já foram criados. Uma variável desse tipo não poderia ser de instância pois seria inicializada a cada objeto criado.

```
public class Cessna {
 // variável de classe
    public static int numSérie = 0;

 // variável de instância
 public int serial;

 // Construtor
 Cessna() {
    (...)
    serial = numSérie++; // incrementa a cada
                         // chamada do construtor
 }

    (... restante da classe ...)
}
```

Podemos então, dentro de um programa de controle de estoque da fábrica de aviões por exemplo, saber quantos Cessnas já foram fabricados. A melhor forma de fazer isto é usando o nome da classe:

```
System.out.println("No de Cessnas criados: " +
       (Cessna.numSérie));
```

Você também pode usar os nomes dos objetos mas deve lembrar que se trata de uma variável estática. Um objeto pode alterar o valor que será lido por outro:

Copyright © 1999, Helder L. S. da Rocha. Todos os direitos reservados. EP:SF-Rio/D99

```
Cessna c1 = new Cessna();
Cessna c2 = new Cessna();
c1.numSérie = 100;
c2.numSérie = 120;
Cessna.numSérie = 200;
```

Todas as três últimas atribuições acima alteram a mesma variável. A última sintaxe é a mais indicada por ser menos confusa.

Um parêntese: veja que estrago pode ser causado por usar variáveis públicas! De repente o número de série deixou de servir para identificar aviões. Mantenha suas variáveis declaradas como `private`!

Os métodos que utilizamos em nossos exemplos até agora, com exceção do main(), foram métodos de instância, ou seja, atuavam sobre um objeto. Como fizemos com variáveis, podemos também definir métodos de classe, ou estáticos. Métodos estáticos se comportam como funções. Precisam de argumentos para operar (a não ser que operem somente localmente) pois não atuam sobre um objeto em particular. Os métodos de `java.lang.Math`, por exemplo, precisam ser invocados da forma:

```
Math.sqrt(2);
Math.abs(x + y);
```

Se definirmos um método tipo `static` em `Cessna`, para acessá-lo teríamos que fazer:

```
Cessna.métodoEstático();
```

Novamente, poderíamos usar uma instância, mas, além de ser algo desnecessário, torna a sintaxe confusa. Para usar métodos estáticos não precisamos criar objeto algum.

Exercício

8. Substitua o π usado no `Círculo` pelo valor mais preciso `Math.PI`, definido na biblioteca de classes (API) da linguagem Java. Para garantir a conversão de tipos (`double` para `float`, será necessário usar coerção (cast) explícita no retorno dos métodos. Ex:

```
return (float)(2 * Math.PI * raio);
```

9. Crie um método:

```
public float arco_x(float radianos) { ... }
```

Copyright © 1999, Helder L. S. da Rocha. Todos os direitos reservados. EP:SF-Rio/D99

que retorne a coordenada `x` do círculo correspondente ao arco do ângulo passado como argumento. Para calcular o seno do ângulo, use a função estática `Math.sin(float radianos)`. Use a equação: $x = raio * \text{sen}(\alpha)$;

10. Use a função `Math.random()` para gerar uma quantidade de 1 a 10 círculos de tamanhos aleatórios. Faça-os aparecer na sua aplicação gráfica. Registre a contagem dos círculos criados (use `Graphics.drawString()`).
11. Incremente o exercício anterior para que os círculos tenham posições diferentes também.

Referências e Apontadores

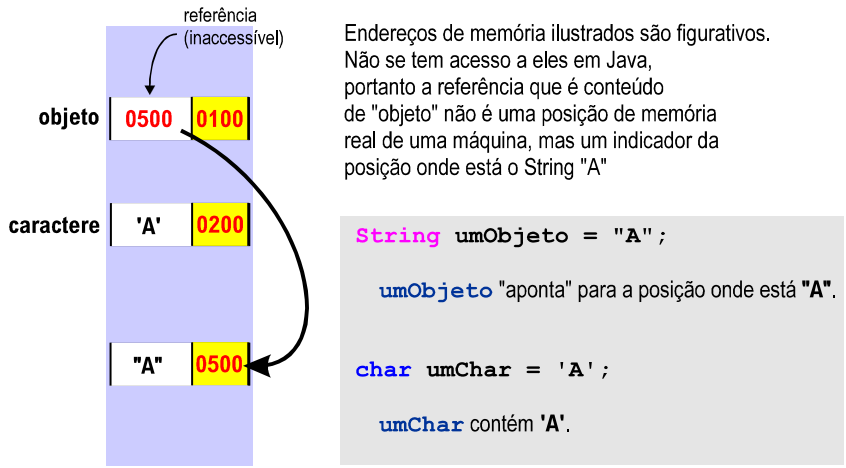
É amplamente divulgado que uma das principais características de Java que a difere de C com vantagens é a ausência de apontadores (ou ponteiros). Aí vem a pergunta: como, em uma linguagem sem apontadores, indexar vetores ou criar listas encadeadas?

Na verdade, Java tem sim apontadores, mas eles são internos à linguagem e não se pode manipulá-los da forma que se faz em C. Os apontadores de Java não têm um endereço que se possa ter acesso ou modificar. São implícitos e pode-se programar à vontade sem ter que se preocupar com a existência deles.

Os apontadores de Java são chamados de *referências*. Usamos várias delas neste capítulo. Sempre que se declara uma variável como sendo do tipo de uma classe (e não um tipo primitivo), cria-se uma referência a um objeto, por enquanto inexistente. Quando criamos o objeto (com new) e o atribuímos uma variável, ela passa a armazenar não o objeto em si, mas uma *referência* para ele.

Java então possui duas formas de armazenar informações em uma variável: diretamente ou indiretamente, através de referências. *Tipos primitivos* são passados diretamente, *por valor*. *Objetos* são *sempre passados por referência* e por isso são chamados de *tipos de referência*. A ilustração a seguir mostra este conceito.

Copyright © 1999, Helder L. S. da Rocha. Todos os direitos reservados. EP:SF-Rio/D99



Lembre-se que a cadeia de caracteres, ou *string*, é um objeto em Java. É construída a partir da classe `String`, que faz parte da biblioteca fundamental da linguagem. Já o tipo primitivo `char`, armazena caracteres individuais. A atribuição

```
String objeto = "A";
```

é na verdade um atalho para

```
String objeto = new String("A");
```

e só vale para a criação de objetos do tipo `String`. Outros objetos devem ser criados usando `new` e um construtor apropriado. Este atalho existe pelo fato da criação de cadeias de caracteres ser uma tarefa extremamente comum e pelo *status* especial dado aos literais do tipo `String`.

Como os tipos de referência não recebem o objeto em si, mas uma referência para ele, é possível haver mais de uma referência a um mesmo objeto e isto não for observado, poderá gerar erros. Veja o seguinte trecho de código:

```
Círculo cGrande, cPequeno; // declarações

// cria objeto
cPequeno = new Círculo(0.0, 0.0, 5.0);

// copia cPequeno em cGrande
cGrande = cPequeno;

// muda o raio de cPequeno
cPequeno.mudaRaio(2.0);

System.out.println("Raio de cGrande: " + cGrande.raio() +
                   "; Raio de cPequeno " + cPequeno.raio());
```

Copyright © 1999, Helder L. S. da Rocha. Todos os direitos reservados. EP:SF-Rio/D99

Se você incluir o trecho de código acima em um programa Java, irá esperar que ele imprima a linha a seguir, correto?

```
Raio de cGrande: 5.0; Raio de cPequeno: 2.0
```

Só que na realidade, ele vai imprimir outra coisa:

```
Raio de cGrande: 2.0; Raio de cPequeno: 2.0
```

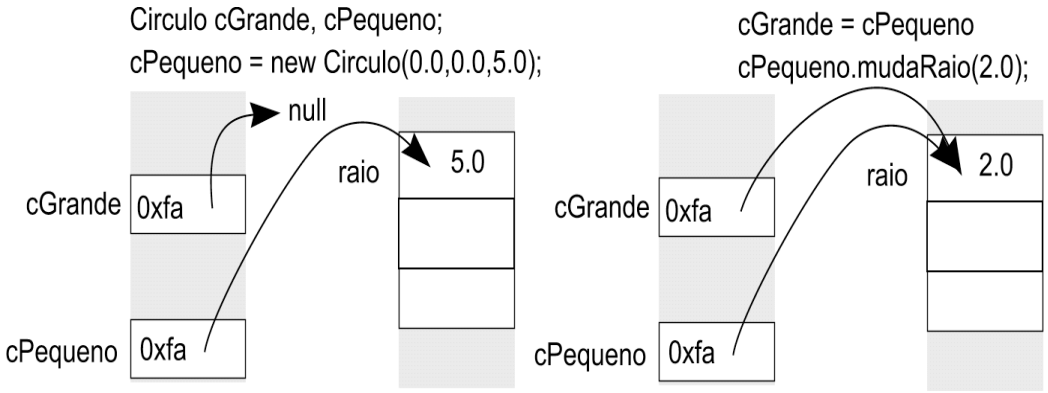
E agora? O que houve? Nós não havíamos guardado o valor anterior de `cPequeno` na variável `cGrande`?

A resposta é "Não!". Quando fizemos

```
cGrande = cPequeno;
```

nós guardamos sim o *conteúdo* de `cPequeno` em `cGrande` só que o conteúdo de `cPequeno` *não é* um círculo! O conteúdo de `cPequeno` é a *referência* para a posição onde está a um círculo de raio 5 e isto é o que foi copiado para a variável `cGrande`. Agora tanto `cPequeno`, como `cGrande`, *possuem a mesma referência* que aponta para o círculo de raio 2!

Alteramos o objeto apontado por `cPequeno`, mudando o seu raio. Como a referência de `cGrande` é a mesma de `cPequeno`, a mudança será vista em ambos ou seja, a informação *não foi* preservada. A figura a seguir ilustra todo o processo.



Vamos supor agora que um programa tem uma senha que será comparada com a senha digitado por um usuário. O trecho de código a seguir ilustra esta situação:

```
String senha, tentativa;
senha = new String("Carpe Diem");

    (...)
```

Copyright © 1999, Helder L. S. da Rocha. Todos os direitos reservados. EP:SF-Rio/D99

```
tentativa = "Carpe Diem";

        // tentativa armazenaria a senha
        // digitada pelo usuário.

if (tentativa == senha) {
   System.out.println("Seja Bem Vindo!");
   entre();

} else {
   System.out.println("Vá embora!");
}
```

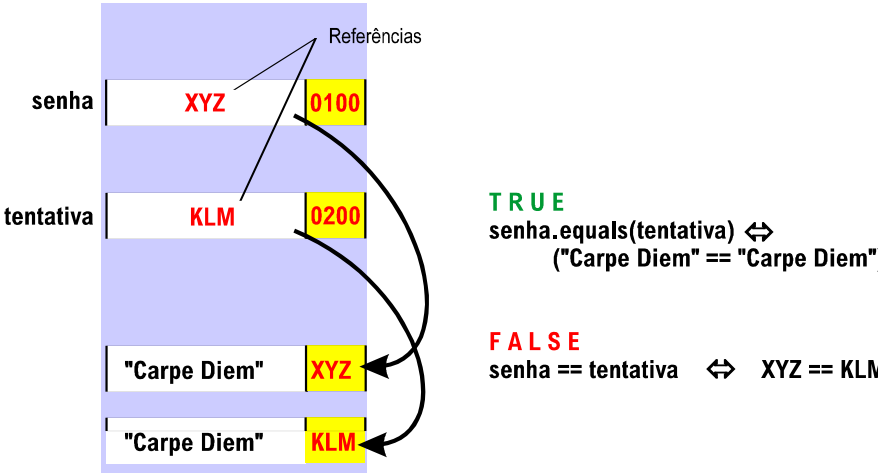
E então? Você roda o programa e...

```
Vá embora!
```

A situação é a mesma que foi descrita anteriormente, só que desta vez o erro aconteceu na comparação. A expressão:

```
tentativa == senha
```

é falsa porque as referências de `tentativa` e `senha` são diferentes, apesar de conterem *a mesma* cadeia de caracteres. As duas contém exatamente o mesmo texto, mas como estão em posições diferentes, e o que foi comparado aqui foram as referências, não o conteúdo dos objetos, a expressão resulta em `false`. A figura a seguir ilustra a situação.



A única forma de comparar objetos (ou fazer cópias deles) é através de métodos. A classe `String`, por exemplo, possui um método `equals()` somente para comparar duas cadeias de caracteres. Desta forma, poderíamos rescrever a expressão anterior da forma:

```
if (tentativa.equals(senha)) {
```

Copyright © 1999, Helder L. S. da Rocha. Todos os direitos reservados. EP:SF-Rio/D99

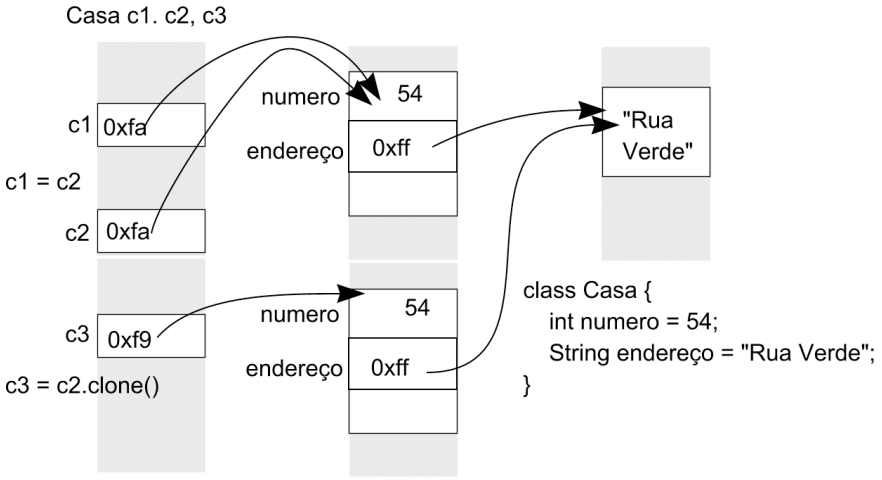
De forma semelhante, podemos usar o método `clone()` da classe `Círculo` para fazer cópias de objetos:

```
cGrande.clone(cPequeno);
```

resultando em duas variáveis com referências para objetos diferentes e com o mesmo conteúdo.

Os métodos `equals()` e `clone()` na verdade são definidos na classe `Object` e herdados por todas as classes. A versão de `String` é uma sobreposição do original para lidar com cadeias de caracteres. É importante sobrepor estes métodos em cada objeto que necessitar ser comparado ou duplicado.

O `clone` usado em círculo, por exemplo, só irá funcionar se `Círculo` redefinir o método `clone` (ou declarar `implements Cloneable`). Isto é porque `clone`, em `Object` é declarado `protected`, e como tal, só pode ser usado em superclasses. Ou Círculo redefine `clone` com modificador `public` ou implementa `Cloneable`, que é uma interface que já define `clone` como público. É sempre necessário redefinir `clone` quando um objeto possui campos de dados que são outros objetos. `clone`, como está definido em `Object`, apenas copia os valores dos campos do objeto. Se estes campos contiverem objetos, só as suas referências serão copiadas e não o seu conteúdo:



Como é que eu poderia comparar dois aviões Cessna? A implementação original (herdada de `Object`) provavelmente não servirá. O `equals()` para a classe `Cessna` deveria me dizer se um Cessna A e outro B correspondem ao mesmo avião. Isto só seria verdade se ambos tivessem o mesmo número de série. Poderíamos então implementar o método assim:

Copyright © 1999, Helder L. S. da Rocha. Todos os direitos reservados. EP:SF-Rio/D99

```
public boolean equals(Object outro) {
    if (this.serial == other.serial) return true;
    else return false;
}
```

Destruição de Objetos

Depois que vimos como criar objetos, resta agora saber como destruí-los e liberar espaço na memória, certo? Errado! Em Java, você não precisa se preocupar com a destruição de objetos! Uma das inovadoras características da linguagem é exatamente esta: a *coleta automática de lixo*, que libera automaticamente a memória que não é mais utilizada. Como é que o sistema de execução sabe que um objeto não vai ser mais utilizado? Simples: quando um objeto não têm mais referências apontando para ele, não pode ser mais usado então ele é descartado e a sua memória liberada. Por exemplo, suponha que eu crio um objeto qualquer:

```
StringBuffer sb;
sb = new StringBuffer("Odie");
```

A variável `sb` é uma referência para o objeto `StringBuffer` que contém o texto "Odie". Se agora fazemos:

```
sb = new StringBuffer("Garfield");
```

atribuímos a `sb` uma nova referência para outro objeto `StringBuffer`. O Odie ainda está lá, ocupando memória, mas logo que passar a "carrocinha" ele será recolhido, pois não tem dono.

O processo de coleta de lixo é realizado por uma linha de controle (*thread*) de baixa prioridade que fica monitorando o sistema. A maior parte do trabalho é realizado quando outras coisas não estão sendo feitas no programa, portanto, não há grande perda de desempenho.

Há um tipo de finalização porém que pode ainda ser necessária quando há arquivos ou *sockets*[3] abertos. Como esses recursos não são liberados pelo coletor de lixo, é necessário fazer a liberação "manualmente". Existe um método definido na classe `Object` (e portanto herdado por todas as outras classes) projetado para operações desse tipo: `finalize()`. Para usá-lo, é só sobrepor sua implementação original (vazia) com uma implementação específica. Ele é

[3] Conexões TCP/IP

Copyright © 1999, Helder L. S. da Rocha. Todos os direitos reservados. EP:SF-Rio/D99

chamado automaticamente pelo sistema de execução Java. Um exemplo para fechar um socket seria:

```
protected void finalize() throws IOException {
 if(socket != null) close();
}
```

Exercícios

12. Crie um método `equals()` para a classe `Círculo`. Qual o critério utilizado para comparar dois círculos?
13. O método `clone()` faz uma cópia de todas as referências que um objeto possui. Mas o `clone` herdado de `Object` pode não resolver todos os problemas. Se você tem um objeto que possui referências a outro objeto ou a um vetor, o clone não realizará uma cópia perfeita. Implemente uma versão de `clone()` pare resolver este problema.

1.3. Herança, Classes Abstratas e Interfaces

Todas as classes vistas nos exemplos apresentados até agora foram sempre extensões de uma classe já existente. Em alguns exemplos, isto esteve explícito através da cláusula "`extends`", seguida pelo nome da classe estendida, ou *superclasse*; em outros, a superclasse não fazia parte da declaração. Neste caso, ela *implicitamente* estendia a classe `Object` (`java.lang.Object`). Ou seja, dizer

```
class Cessna { ... }
```

é o mesmo que dizer

```
class Cessna extends Object { ... }
```

Para criar uma subclasse de qualquer classe já existente e acessível, basta declará-la como tal:

```
class Estrela extends Astro
```

A subclasse herda todas variáveis e métodos públicos e protegidos definidos na superclasse. Ela não herda os construtores mas ganha um construtor *default* e pode definir novos construtores e invocando os construtores da superclasse.

A extensão de classes caracteriza a *herança* e torna possível o *polimorfismo*, caracterizado principalmente pela *sobreposição de métodos*, analisada neste capítulo. Um objeto polimórfico pode assumir múltiplas formas, seja com as

Copyright © 1999, Helder L. S. da Rocha. Todos os direitos reservados. EP:SF-Rio/D99

características de sua própria classe, seja com características da classe que estende. A extensão de uma classe geralmente adiciona uma nova funcionalidade à classe estendida. A nova classe herda métodos e campos de dados públicos e protegidos, adiciona novos métodos e variáveis e possivelmente redefine parte das funções da superclasse.

Na última seção, nos concentramos na característica da *abstração*, definindo classes como tipos de dados abstratos, e no *encapsulamento*, protegendo os dados e publicando uma interface pública de funções para classes e objetos. Nesta seção, iremos estudar como Java implementa não só a herança e polimorfismo, mas também a *modularidade*, realizada através dos pacotes.

As classes abstratas e interfaces são peças fundamentais no desenvolvimento de sistemas com alto grau de reutilização de código. As mais complexas APIs da linguagem Java dependem de interfaces e classes abstratas para alcançarem um alto grau de versatilidade e facilidade de uso. Veremos neste capítulo como usar interfaces e classes abstratas em aplicações Java.

Pacotes

Além da hierarquia formada pelas classes e superclasses, Java oferece mais uma outra forma para organizar conjuntos de classes: os *pacotes*.

Você pode agrupar qualquer quantidade de classes relacionadas ou não em um pacote. As classes não precisam ter relação alguma entre si, embora isto seja comum quando se define qualquer hierarquia.

Todos os programas que fizemos até agora estavam dentro de pacotes! Não declaramos nada, então Java colocou nossas classes em um pacote *default*, restrito ao diretório onde compilamos e rodamos nossos arquivos. E pacotes são mais ou menos isto: *diretórios*. Quando definimos um pacote para agrupar classes em Java estas classes são armazenadas em diretórios específicos.

No *Java Development Kit (JDK)*, pacotes são utilizados para agrupar as classes da biblioteca fundamental da linguagem Java (API), por exemplo:

```
java.awt
java.lang
java.io
java.awt.event
```

Copyright © 1999, Helder L. S. da Rocha. Todos os direitos reservados. EP:SF-Rio/D99

são todos nomes de pacotes. Dentro deles estão as classes. Freqüentemente temos que usar uma ou outra classe de um desse pacotes e então importamos o mesmo, usando

```
import java.awt.*;
```

por exemplo, para ter acesso à classe `Frame`, `Button`, `Graphics`, etc.

Uma aplicação ou módulo de aplicação provavelmente usaria pacotes para organizar todas as suas classes. É um recurso a mais que o programador tem à disposição para realizar a organização de informação, e assim simplificar o seu uso.

Como criar um pacote

Para formar um pacote, basta declará-lo no início de cada classe que o compõe. O conjunto formado pelas *classes que declaram pertencer a um mesmo pacote* caracterizam o mesmo. Para isto é necessário que cada programa tenha como primeira instrução a declaração

```
package nome-do-pacote;
```

O nome de um pacote pode ser qualquer identificador válido. Na ausência dessa declaração, a classe é incluída em um pacote *default*, genérico, que na prática contém todas as classes que não tem package definido e que estão no mesmo subdiretório de trabalho.

Se estivéssemos escrevendo um simulador de vôo, poderíamos colocar a classe `Cessna` em um pacote `simulador`, para organizá-la junto com outras classes relacionadas com o programa de simulação de vôo, como `Aeroporto`, `Nuvens`, etc. e ter privilégios de *acesso especial* à métodos e variáveis de outras classes pertencentes ao mesmo pacote. No arquivo, colocaríamos:

```
package simulador;
public class Cessna { ... }
```

Agora a classe se chama `simulador.Cessna`.

Como usar um pacote

Para usar classes pertencentes a um pacote declarado, é necessário usar a combinação

```
nome_de_pacote.Nome_de_Classe
```

Copyright © 1999, Helder L. S. da Rocha. Todos os direitos reservados. EP:SF-Rio/D99

sempre que se precisar usar a classe, porque agora o nome da classe é a combinação do nome do pacote com o nome original declarado. Uma forma mais prática de usar uma classe, principalmente quando uma mesma classe vai ser usada várias vezes é importá-la através da declaração `import`:

```
import nome_de_pacote.Nome_de_Classe;  ou
import nome_de_pacote.*;
```

que permite que se use o nome *abreviado* da classe em vez do *nome longo*. `import` não é uma declaração necessária e pode ser dispensada sempre. Pode-se usar qualquer classe sem precisar de `import`, chamando-a pelo nome completo:

```
class AplicacaoGrafica extends java.awt.Frame {
    public static void main(java.lang.String args[]) {
        java.awt.Frame fr = new java.awt Frame("Frame");
        java.awt.Button b = new java.awt.Button("Botão");
        b.addActionListener(new java.awt.event.ActionListener() { ...
```

Usando dois *imports*, o codigo fica muito mais legível:

```
import java.awt.*;
import java.awt.event.ActionListener;

class AplicacaoGrafica extends Frame {
   public static void main(String args[]) {
        Frame fr = new Frame("Frame");
        Button b = new Button("Botão");
        b.addActionListener(new ActionListener() { ...
```

A classe `String` não precisa de `import` porque é parte do pacote `java.lang`, que o sistema já importa automaticamente.

O grande benefício aqui é que esta nova nomenclatura evita conflitos entre nomes de classes. Os fabricantes de APIs Java sempre distribuem todas suas classes dentro de pacotes e, se você encontra dois pacotes com uma classe de mesmo nome ainda assim pode usá-las no mesmo programa desde que use o nome totalmente qualificado para uma delas.

```
import com.microsoft.win32.Button;

(...)

Button bm = new Button("Botão Microsoft");
java.awt.Button bj = new java.awt.Button("Botão Java");
```

Copyright © 1999, Helder L. S. da Rocha. Todos os direitos reservados. EP:SF-Rio/D99

Pacotes, diretórios e CLASSPATH

A declaração `import` na verdade não importa uma classe, mas *importa* o caminho para que o interpretador a encontre e possa saber de quem se trata quando o programador não usar o nome completo da classe.

A declaração `package` é apenas parte da construção de um pacote. Depois da compilação, as classes que constam do pacote devem ser colocadas em um diretório que tenha o mesmo nome que o pacote. Quando o interpretador Java precisar importar uma classe pertencente a um pacote, ele irá procurá-la dentro desse diretório. O caminho até este diretório deve também ser conhecido pelo interpretador. Para isto, deve estar na lista de diretórios que ele usa para procurar classes, que é o `CLASSPATH`.

A `CLASSPATH` é uma variável de ambiente do sistema definida pela máquina virtual Java (JVM). Na hora em que você instala uma aplicação Java, ela poderá definir uma `CLASSPATH` antes que possa rodar.

O interpretador só procura classes dentro de pacotes que estejam acessíveis pelo `CLASSPATH`. Por exemplo, se você tem várias classes declaradas como parte de um pacote chamado `livro` e uma variável de ambiente (no *Windows*) definida no `AUTOEXEC.BAT` como :

```
CLASSPATH=C:\programas\
```

o interpretador procurará suas classes em:

```
.\livros\
C:\jdk1.2\jre\lib\rt.jar\livros\
C:\programas\livros\
```

O sistema já define previamente um `CLASSPATH` básico que inclui as classes fundamentais Java (`rt.jar`) e o diretório atual.

Dentro de pacotes pode haver classes e outros pacotes. Pode-se ter grandes hierarquias de pacotes. Na API Java, por exemplo, todos os pacotes `java.lang`, `java.awt`, `java.io`, etc estão dentro de um pacote chamado java. Há um pacote chamado `java.awt.event` dentro de `java.awt`. Se você abrir e visualizar o conteúdo do arquivos `rt.jar` (que contém as classes *run-time* Java no JDK), verá que cada subpacote representa um diretório. O pacote `java.awt.image` representa

```
%CLASSPATH%\java\awt\image
```

Copyright © 1999, Helder L. S. da Rocha. Todos os direitos reservados. EP:SF-Rio/D99

Se uma classe declara que está em um certo pacote, ela *deve estar* fisicamente dentro do diretório e subdiretórios correspondentes para permitir que o interpretador Java a encontre. Os pontos entre os pacotes são traduzidos em separadores de caminho de diretórios. Se você declara:

```
package BR.com.empresa.ferramentas.gui;
```

antes de uma definição de classe, você deve colocar suas classes em um subdiretório

```
%CLASSPATH%\BR\com\empresa\ferramentas\gui\
```

A convenção de usar o nome de domínio ao contrário é recomendado pela especificação da linguagem Java como um meio de garantir que nunca haja conflito entre nomes de classes entre fabricantes diferentes.

Se você usar o compilador `javac` com a opção `-d`, ele criará os diretórios necessários e moverá os arquivos compilados automaticamente para seus lugares:

```
javac –d C:\PROGRAMAS Cessna.java
```

cria automaticamente um diretório `simulador`, abaixo de `C:\PROGRAMAS` e copia as classes resultantes da compilação para lá. Você terá que incluir, depois, `C:\PROGRAMAS` no `CLASSPATH` para que o interpretador encontre os arquivos quando precisar. Você também pode fazer o mesmo usando o diretório atual:

```
javac –d . Cessna.java
```

que criará `simulador` abaixo do diretório atual.

OBSERVAÇÃO IMPORTANTE: *Apesar da solução acima ser prática, ela pode trazer alguns problemas na hora em que você for compilar outras classes que usam programas de outro pacote. Depois que você compilar seus programas-fonte para um outro pacote,* remova-os *do seu diretório de trabalho. O compilador irá reclamar se ele achar os arquivos .java do pacote que você está usando no diretório atual. Para evitar esses problemas,* não use *a opção* –d *e crie seus diretórios manualmente, compilando os programas-fonte dentro deles. Isto não é problema se você usa um ambiente integrado de desenvolvimento diferente do JDK, como o JBuilder ou Visual Café Pro.*

Não é possível importar pacotes; apenas se importa classes. Portanto fazer:

```
import java.awt.*;
```

no início de um programa importa *todas as classes* do pacote `java.awt` como `java.awt.Frame`, `java.awt.Graphics`, mas *não* importa as classes que estão dentro de `java.awt.event`. Para importar essas classes também, é preciso fazer:

Copyright © 1999, Helder L. S. da Rocha. Todos os direitos reservados. EP:SF-Rio/D99

```
import java.awt.*;
import java.awt.event.*;
```

A declaração `package` tem que ser a primeira coisa que aparece em um arquivo .java, se estiver presente. Só pode haver comentários antes dela. Depois de `package` podem vir nenhuma, uma ou mais declarações `import`, e, finalmente, uma ou mais definições de classes ou interfaces. Se essas três construções de primeiro nível estiverem presentes em um arquivo .java, elas devem seguir precisamente esta ordem:

- declaração de pacote
- declaração(ões) import
- definição(ões) de classe(s) e/ou interface(s)

Modificadores de Acesso

Os modificadores de acesso definem se uma variável ou método pode ser acessada por outras classes, por subclasses, por nenhuma outra classe ou por classes do mesmo pacote. Valem somente para variáveis de classe ou de instância. Não podem ser usados por variáveis locais.

Os modificadores básicos são `public` e `private`, que vimos nos capítulos anteriores. Quando nenhum modificador de acesso aparece na declaração de um método ou variável de instância ou de classe, o acesso é permitido apenas *dentro do mesmo pacote*. Este modificador oculto é definido na especificação Java como "friendly" ou "package". Há ainda um modificador chamado `protected` que é usado para ampliar o acesso de métodos e variáveis às subclasses.

Um bom projeto de classe sugere que as variáveis e métodos tenham somente o nível de acesso necessário ao uso a que se destina. Isto torna a classe fácil de usar e garante maior segurança. Se uma classe só vai usar uma variável para operações locais, esta deve ser declarada como `private`. Se um método só deve ser usado pela própria classe, classes do mesmo pacote ou subclasses, deve ser declarado como `protected`.

Com esses modificadores, pode-se controlar o aspecto das interfaces de um objeto ou classe. A interface pública é o conjunto de métodos disponíveis a todos. A interface protegida é a funcionalidade proporcionada às subclasses. A interface *amigável*, expõe funções a classes "amigas", que compartilha o mesmo pacote.

Copyright © 1999, Helder L. S. da Rocha. Todos os direitos reservados. EP:SF-Rio/D99

Os modificadores de acesso disponíveis na linguagem Java estão listados abaixo com o nível de acesso em ordem decrescente:

- **`public`**: Torna o método ou variável universalmente acessível.

  ```
  public int x;
  ```

  Um bom projeto orientado a objetos raramente usa variáveis públicas (o acesso às variáveis sempre pode ser feito através de métodos). Use para métodos, variáveis globais ou constantes que devem ser visíveis em todo lugar.

- **`protected`**: O acesso é permitido somente às subclasses e classes do mesmo pacote.

  ```
  protected int y;
  ```

  Classes que não fazem parte de nenhuma das duas famílias não tem acesso. Use no desenvolvimento de classes incompletas, projetadas para serem estendidas; em métodos e variáveis que devem ser acessíveis somente nas subclasses e pacotes e não fora deles. Observe que se você não declara pacote, sua classe pertence ao pacote *default* e `protected` age como `public` para as classes do mesmo diretório.

- *package ou friendly* (*default*): Quando *não há* modificadores de acesso explícito, subentende-se que o acesso é restrito às classes do mesmo pacote.

  ```
  int z; // não há modificador explícito!
  ```

  Neste caso, apenas as classes que tem a mesma declaração `package` podem ter acesso. A proteção é maior que `protected`. Se uma subclasse não pertencer ao pacote, ela não tem acesso, mesmo que seja subclasse. Assim como ocorre com `protected`, caso a classe *não tenha* declaração `package`, o acesso será permitido *à todas* as classes do mesmo diretório que também não têm esta declaração. Use quando métodos ou variáveis só interessarem à própria aplicação ou módulo (pacote) dela.

- **`private`**: Torna o método ou variável privativo à classe (se for `static`) ou objeto.

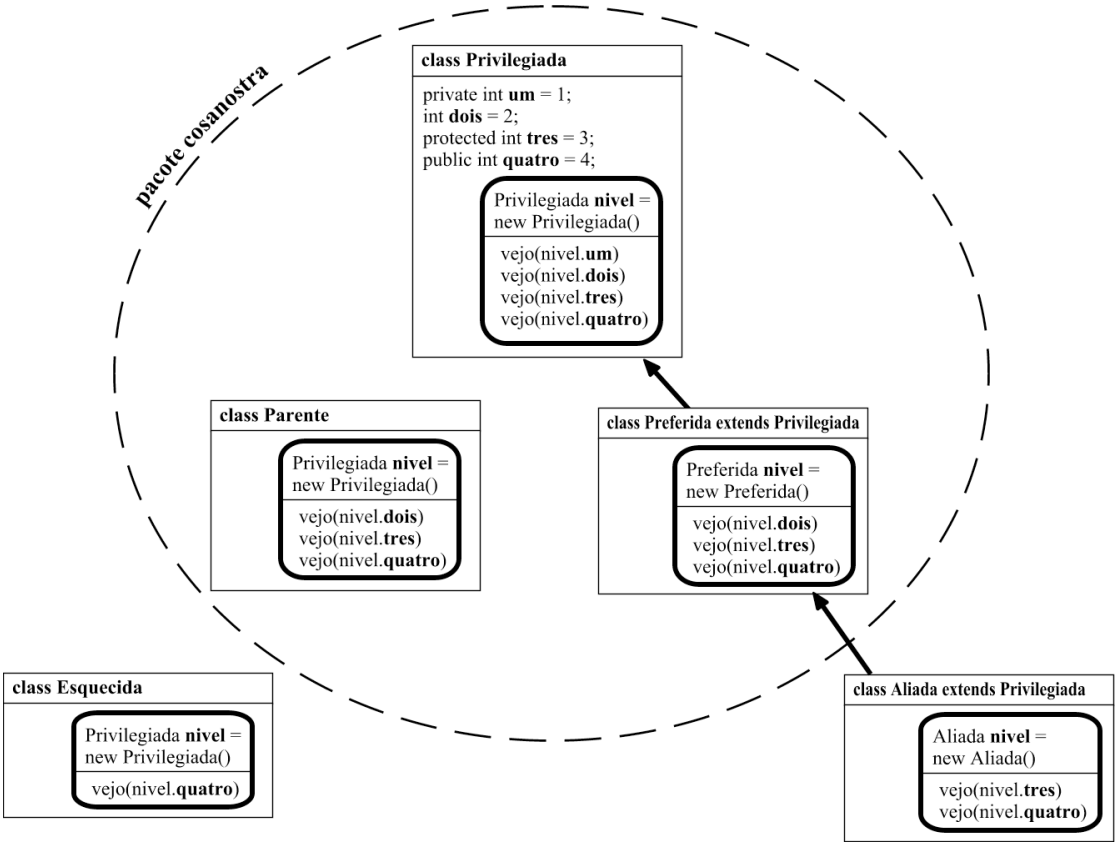
  ```
  private int k;
  ```

Copyright © 1999, Helder L. S. da Rocha. Todos os direitos reservados. EP:SF-Rio/D99

Os métodos e variáveis declarados com este modificador só podem ser usados internamente. Use para métodos e variáveis que só são usados dentro da classe, e não interessam à classes externas.

Agora que vimos as diferenças entre modificadores de acesso em *mil palavras*, vejamos tudo resumido na figura abaixo. As linhas pontilhadas separam pacotes diferentes. Os retângulos representam classes. Os retângulos de linha grossa e cantos arredondados representam objetos. As setas apontam para as superclasses.

Observe que o uso dos campos `protected` é feito pela própria subclasse, assumindo a sua posição de ser herdeira das variáveis das superclasses. As classes que não herdam os valores (`Esquecida` e `Parente`) são obrigadas a criar uma instância de uma classe que os possui para poder ter acesso.



Sobreposição de métodos ocultação de dados

Podemos ampliar ou alterar a funcionalidade de uma classe estendida através da *sobreposição* de seus métodos ou através da *sobrecarga* dos nomes dos métodos. A

Copyright © 1999, Helder L. S. da Rocha. Todos os direitos reservados. EP:SF-Rio/D99

sobrecarga já vimos no capítulo anterior com construtores. Funciona da mesma forma com métodos comuns. A diferença é que no caso dos métodos, a sobrecarga pode ocorrer por herança: se uma subclasse herda um método e define outro com mesmo nome e assinatura diferente, ela passa a ter dois métodos com o mesmo nome, o que caracteriza uma sobrecarga.

A sobreposição tem um efeito mais amplo sobre a funcionalidade do objeto. Consiste em *substituir* a implementação do método da superclasse com uma nova definição. As assinaturas, neste caso, devem ser *idênticas* ou serão interpretadas como um novo caso de sobrecarga.

Um método novo pode ter diferentes modificadores de acesso desde que estes garantam *mais* acesso que o método sobreposto. Um método declarado `protected` na superclasse pode ser declarado `protected` ou `public`, mas não `private` ou *friendly* (o modificador *default*) pois os últimos são mais restritivos. Os métodos também não podem declarar que provocam mais exceções que os métodos sobrepostos.

A sobreposição dos dados em variáveis e referências não os substitui. Eles continuam a existir mas a nova definição os torna ocultos. Na prática, isto pode ser visto como uma sobreposição. Eles ainda podem ser recuperados usando a palavra-chave super:

```
class Vinte {
   protected int num = 20;
}

class Duzentos extends Vinte {
   protected int num = 200;

   public static void main(String a[]) {
      System.out.println("Total: " + num + super.num);
   }
}
```

Invocação de métodos e ligação dinâmica

Quando um método é chamado sobre um objeto, o tipo daquele objeto em tempo de execução é quem diz qual implementação será usada. Você pode declarar um objeto como sendo do tipo de uma superclasse, por exemplo:

```
Object obj = new Cessna();
```

Isto é perfeitamente válido. Na hora em que o programa fizer

```
obj.toString();
```

Copyright © 1999, Helder L. S. da Rocha. Todos os direitos reservados. EP:SF-Rio/D99

não será o `toString()` definido em `Object` que irá ser chamado, mas o que está na memória naquele momento (objetos são alocados dinamicamente em tempo de execução), portanto, o `toString()` irá invocar o método em um objeto do tipo `Cessna`, e fazer o que foi definido na classe `Cessna`.

A única exceção a esta regra é quando em vez de se usar a referência a um objeto, se usa a palavra-chave `super`. A invocação `super.método()` *sempre* usa a implementação de um método de acordo com a superclasse e não de acordo com uma implementação sobreposta. O exemplo ("The Java Programming Language: p.61" - Gosling, Arnold - 1996) a seguir ilustra esta situação:

```
class That {
   /** retorna o nome da classe */
   protected String nm() {
      return "That";
   }
}

class More extends That {
   protected String nm() {
      return "More";
   }
   protected void printNM() {
      That sref = super;
      System.out.println("this.nm() = " + this.nm());
      System.out.println("sref.nm() = " + sref.nm());
      System.out.println("super.nm() = " + super.nm());
   }
}
```

A saída do programa acima é:

```
this.nm() = More
sref.nm() = More
super.nm() = That
```

Quando não estender uma classe?

Quando estender uma classe? Será que porque uma `Zebra` parece uma `Mula` poderíamos estender `Mula`, acrescentar umas listras e obter uma `Zebra`?

O processo de escolha de classes e objetos, a determinação das relações entre eles é um tópico complexo que têm resultado em milhares de publicações. A análise e projeto de sistemas orientados a objetos é um assunto importantíssimo. A sua aplicação no desenvolvimento de programas em Java ou em qualquer outra linguagem orientada a objetos é a melhor maneira de garantir

Copyright © 1999, Helder L. S. da Rocha. Todos os direitos reservados. EP:SF-Rio/D99

que um sistema vai atender às especificações da forma mais eficiente possível. No processo de desenvolvimento de um software, a implementação é a última etapa. A maior parte do esforço é despendido na análise do problema e posterior elaboração do modelo de objetos. Somente depois que todas as relações entre objetos são mapeadas é que tem início a implementação.

Não faz parte do escopo deste curso ensinar análise e *design* orientado a objetos, mas se você quiser levar à sério o desenvolvimento usando Java, não deve deixar de considerar a possibilidade de conhecer mais sobre esse assunto, essencial em cursos mais avançados sobre a linguagem Java e no desenvolvimento de software.

Para ilustrar um pouco a importância da análise cuidadosa do problema antes da sua implementação, vejamos um exemplo de implementação problemática.

Suponha que precisamos de uma classe que represente uma esfera. Poderíamos criar uma classe `Esfera` do zero. Mas, lembrando que um círculo parece-se com uma esfera e concluindo que muito do que já foi definido para `Círculo` poderia ser aproveitado em `Esfera` (duas coordenadas, alguns métodos, etc.), decidimos utilizar as propriedades das linguagens orientadas a objetos e fazer com que `Esfera` seja uma especialização de `Círculo`.

Fazemos então:

```
public Class Esfera extends Círculo { ... }
```

Como os métodos de `Círculo` são públicos, eles são todos herdados por `Esfera`. As variáveis não são vistas da nova classe porque são privativas. O ideal é que elas fossem `protected`, para que pudessem ser vistas pelas suas sub-classes. Vamos supor então, só para este exemplo, que as variáveis da classe Círculo são `protected`. Agora já podemos calcular a circunferência de um objeto `Esfera esf`:

```
float circ = esf.circunferência();
```

`Esfera` poderá definir seus próprios métodos e variáveis. Por exemplo, pode-se criar um método `volume()`:

```
public float volume() {
        return (3.14159 * raio * raio * raio);
}
```

Copyright © 1999, Helder L. S. da Rocha. Todos os direitos reservados. EP:SF-Rio/D99

Pode-se também sobrepor métodos da superclasse com uma nova implementação:

```
public double area() {
 return 4 * 3.14159 * r*r /3;
}
```

Também devemos ter um novo construtor para construir esferas, já que usamos uma dimensão a mais:

```
public Esfera(float x, float y, float z, float raio) {
        this.x = x;
        this.y = y;
        this.z = z;
        this.raio = raio;
}
```

No início dissemos que o exemplo acima era problemático, mas aparentemente ele funciona sem problemas. Qual é o problema em utilizá-lo, então? O problema surgirá quando esta classe (`Esfera`) for utilizada por outras classes e métodos que trabalham com `Círculos`. O compromisso assumido quando se cria uma classe é que a sua interface pública (tudo o que está acessível para o público) vai sempre funcionar da forma esperada e assim podemos usar o objeto sem problemas sem se preocupar com as subclasses novas que possam surgir. As subclasses devem ser sempre especializações, isto é, diferentes formas assumidas pela superclasse. Não deve haver problemas em receber objetos `Fusca`, `BMW` e `Vectra` em um objeto `Estacionamento` criado para acomodar objetos do tipo `Automovel`. Eles são automóveis e, portanto, devem estender `Automóvel`. O problema é que uma esfera *não é* um círculo, portanto não deveria ser definida como subclasse de um.

Que problemas isto poderia gerar? Suponha que temos um objeto que construa rodas a partir de círculos. Poderia ter um construtor, por exemplo:

```
class Roda { ...{
    Roda (Círculo c1) { ... }
...}
```

Qualquer subclasse de `Círculo` é considerada um círculo e portanto o construtor de `Roda` poderia receber como argumento: `CírculoAzul`, `CírculoComBorda`, etc. Como a implementação da classe `Esfera`, no nosso exemplo, diz que ela "é" um círculo, ela poderia ser usada também:

Copyright © 1999, Helder L. S. da Rocha. Todos os direitos reservados. EP:SF-Rio/D99

```
Esfera e1;
Roda rodaBicicleta = new Roda(e1);
```

E então, quando o programa for calcular a área, por exemplo, que deveria ser igual em qualquer círculo, o programa irá falhar, pois a área de `Esfera` não é área de círculo.

É possível evitar este tipo de erro no estágio de projeto do modelo orientado a objetos, observando se há uma relação *é um* entre classes e subclasses. Neste estágio, é possível descobrir que `Esfera` não *é um* `Círculo` e evitar complicações que irão aparecer em estágios posteriores do desenvolvimento. A sobreposição de métodos para fazer coisas totalmente diferentes também é uma dica para detectar erros de projeto.

Podemos evitar que as subclasses alterem métodos fundamentais de uma classe usando o modificador `final`. Se tivéssemos projetado a classe `Círculo` com a seguinte declaração no método área:

```
public final double area() { ... }
```

Não seria mais possível que a classe `Esfera` sobrepusesse este método.

Uma Aplicação Orientada a Objetos

Nada melhor que um exemplo, para fixar os conceitos apresentados. Vamos mostrar na prática o uso de classes, superclasses e objetos, aplicando o conceito de polimorfismo (um diretório *é um* arquivo) e de composição (uma lista de arquivos *é parte de* um diretório). As classes do nosso exemplo fazem parte do pacote `sa` e são as seguintes:

- **Arquivo**: uma classe que representa um arquivo qualquer em um sistema hipotético.
- **Diretório**: um tipo especial de arquivo capaz de conter listas de arquivos.

As listagens, com comentários, seguem. Logo após há um breve explicação das principais partes de cada uma. Inicialmente, a classe `Arquivo`:

```
package sa;

/* Esta classe representa um arquivo
 * em um sistema de arquivos hipotético, semelhante ao
 * Windows 95 ou Unix */
public class Arquivo implements Cloneable {
```

Copyright © 1999, Helder L. S. da Rocha. Todos os direitos reservados. EP:SF-Rio/D99

```
///// Variáveis de classe /////
private static int inode; // guarda o número de arquivos criados
private static int cluster = 512; // tamanho minimo de arquivo

///// Variáveis de instância /////
private String nome; // nome do arquivo
private String extensão;  // extensão do arquivo
private int tamanho; // tamanho (bytes) do arquivo
private int id;      // identificação do arquivo
private Diretorio pai; // diretório pai

///// Construtores /////
/** Construtor completo */
public Arquivo(String nome,   String extensão,
               Diretorio pai, int tamanho) {
   this.nome = nome;
   this.pai = pai;
   if (pai != null) {
      pai.adiciona(this);
   }
   this.extensão = extensão;
   this.tamanho = ((int)(tamanho/cluster) + 1) * cluster;
   id = inode++;
}

/** Construtor para arquivos sem extensão */
public Arquivo(String nome, Diretorio pai, int tamanho) {
   this(nome, "", pai, tamanho);
}

/** Construtor para arquivos vazios somente para subclasses
    e classes deste pacote */
protected Arquivo(String nome, Diretorio pai) {
   this(nome, "", pai, cluster);
}

///// Métodos de instância /////
/**
 * Retorna o nome do arquivo
 */
 public String getNome() {
   return nome;
 }
 /** Retorna o pai
 */
 public Diretorio getPai() {
   return pai;
 }

/**
 * Retorna o tamanho do arquivo
 */
 public int getTam() {
```

Copyright © 1999, Helder L. S. da Rocha. Todos os direitos reservados. EP:SF-Rio/D99

```
    return tamanho;
  }

 /**
  * Retorna a extensão do arquivo
  */
  public String getExt() {
    return extensão;
  }

  /**
  * Muda o nome do arquivo
  */
  public void setNome(String novoNome) {
    nome = novoNome;
  }

 /**
  * Muda a extensão do arquivo
  */
  public void setExt(String novaExt) {
    extensão = novaExt;
  }

 /** Muda tamanho do arquivo (somente para classes
  *  amigas ou subclasses
  */
  protected void setTam(int novoTam) {
    tamanho = novoTam;
  }

  /**
  * Retorna caminho de subdiretórios
  */
  public String caminho() {
     String ext = "." + extensão;
     if (this instanceof Diretorio) {
          ext = "/";
     }
     if (pai != null) {
       return pai.caminho() + nome + ext;
     } else return "/";

  }

  ///// Métodos sobrepostos /////
 /**
  * Sobrepõe Object.toString()
  */
  public String toString() {
    String ext = "." + extensão;
    if (this instanceof Diretorio) {
       ext = "/ ";
```

Copyright © 1999, Helder L. S. da Rocha. Todos os direitos reservados. EP:SF-Rio/D99

```
        }
        return "[" + id + "] " + nome + ext + "\t(" + tamanho + ")";
      }

     /**
      * Sobrepõe Object.equals()
      */
      public boolean equals (Object obj) {
        if ( obj instanceof Arquivo ) {
           Arquivo arq = (Arquivo) obj;

           if ( arq.id == this.id ) {
              return true;
           }
        }
        return false;
      }
     /**
      * Sobrepõe Object.clone()
      */
      public Object clone() {
        try {
           return super.clone(); // retorna cópia do arquivo
        } catch (CloneNotSupportedException e) {
           throw new InternalError(e + "");
        }
      }
  }
```

A classe `Arquivo` define o estado e comportamento de um arquivo. Um arquivo é um objeto que pode ser reproduzido (copiado), por isto ele implementa `Cloneable`. Um arquivo tem uma identidade, um nome, uma extensão, um diretório pai e um tamanho. Um arquivo possui métodos que permitem ler e alterar algumas de suas variáveis, imprimir suas características e compará-lo com outro arquivo.

A forma de comparar um arquivo com outro é diferente da forma de comparar um objeto genérico, portanto o método `equals` foi redefinido. A forma de imprimir as informações de um arquivo também é diferente da forma de imprimir as informações de um objeto genérico, então `toString` foi redefinido. Isto permite que o arquivo seja usado em qualquer lugar onde um `Object` for aceito e poderá invocar os métodos `toString` e `equals`. O método `clone` é sobreposto para implementar a interface `Cloneable` e garantir que o arquivo pode ser copiado.

Copyright © 1999, Helder L. S. da Rocha. Todos os direitos reservados. EP:SF-Rio/D99

No construtor é verificado se o diretório pai é `null`. Todo arquivo têm um diretório pai a não ser que este arquivo seja a própria raiz do sistema. Se o arquivo em questão não for a própria raiz, ele pode ser adicionado a um diretório pai. Para isto, é invocado um método de `Diretorio` que adiciona o próprio objeto atual (`this`), que é um arquivo, ao seu diretório pai.

O método `toString` verifica se o objeto atual é uma instância de `Diretorio`, usando o operador `instanceof`. Como todo diretorio é um arquivo, ele também pode invocar `toString` e a verificação garante que o que vai ser impresso é diferente se o objeto for um `Diretorio`.

A listagem a seguir contém a classe `Diretório`:

```
package sa;

import java.util.Vector;  // array dinamico

/* Esta classe representa um diretório
 * em um sistema de arquivos*/

public class Diretorio extends Arquivo {

   private Vector lista;

   ///// Construtores /////

   /** Constroi um diretório */
   public Diretorio(String nome, Diretorio pai) {

      super(nome, pai); // esta linha é necessária!
      lista = new Vector(2);
   }

   ///// Métodos de instância /////

   /**
    * Adiciona novo arquivo na lista
    */
    public void adiciona(Arquivo arquivo) {
      lista.addElement(arquivo);
      setTam(this.getTam() + arquivo.getTam());
    }

   /**
    * Tira arquivo da lista
    */
    public void remove(Arquivo arquivo) {
      lista.removeElement(arquivo);
      setTam(this.getTam() - arquivo.getTam());
```

Copyright © 1999, Helder L. S. da Rocha. Todos os direitos reservados. EP:SF-Rio/D99

```
  }

 /**
  * Retorna vetor com conteudo do diretorio
  */
  public Arquivo[] dir() {
    Arquivo[] dirList = new Arquivo[lista.size()];
    lista.copyInto(dirList);  // copia Vector em vetor dirList
    return (Arquivo[])dirList;
  }

  /**
  * Lista conteúdo com filtro de extensão
  */
  public Arquivo[] dir(String ext) {
    Vector listaFilter = new Vector(lista.size());
    for (int i = 0; i < lista.size(); i++) {
       Arquivo arq = (Arquivo)lista.elementAt(i);
       if (arq.getExt().equals(ext)) {
          listaFilter.addElement(arq);
       }
    }
    Arquivo[] dirList = new Arquivo[listaFilter.size()];
    listaFilter.copyInto(dirList);  // copia Vector em dirList
    return (Arquivo[])dirList;
  }


  ///// Métodos sobrepostos /////

  /**
  * Sobrepõe Arquivo.toString()
  */
  public String toString() {
    return super.toString() + " " + lista.size() + " arquivos";
  }



  /**
   * Sobrepõe Arquivo.clone()
   * É necessário pois a cópia de um diretório não copia
   * a lista de arquivos que ele contém (passa a ser
   * simplesmente um atalho para o mesmo lugar)
   */
  public Object clone() {
    Diretorio d = (Diretorio)super.clone(); // copia diretorio
    d.lista = new Vector(lista.size()); // cria novo Vector vazio

    // copia arquivos um por um para nova localidade
    for (int idx = 0; idx < this.lista.size(); idx++) {
       Arquivo arq = (Arquivo)lista.elementAt(idx); // le da lista
       Arquivo dolly = (Arquivo)arq.clone(); // copia arquivo
```

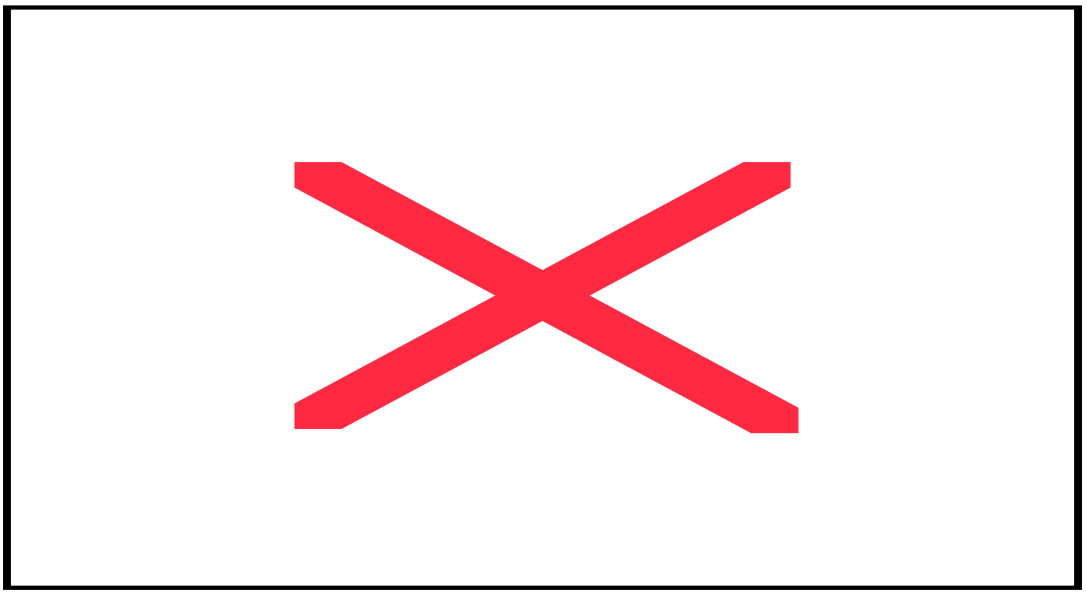
Copyright © 1999, Helder L. S. da Rocha. Todos os direitos reservados. EP:SF-Rio/D99

```
            d.lista.addElement(dolly); // adiciona no Vector
        }
        return d;  // devolve novo diretório
    }
}
```

Um `Diretório` é um `Arquivo`. Ele tem uma lista de arquivos, que aqui é representada através de um objeto do tipo `Vector`. Vector é um vetor de dimensões que podem variar dinamicamente. `Vector` armazena a lista de arquivos contida no diretório. Um diretório possui métodos para adicionar, remover e listar os arquivos que contém. Uma sobrecarga do método `dir` (lista), permite ainda que a listagem seja feita filtrando a extensão do arquivo.

`Diretorio` sobrepõe `toString` porque um diretório possui informações diferentes para exibir como a quantidade de arquivos que possui. Mas, o resto das informações é semelhante, então ele invoca `super.toString` para concluir o trabalho.

`Diretório` sobrepõe `clone` porque não basta copiar o objeto `Diretorio` para outro lugar. É necessário também copiar o objeto Vector. Se isto não for feito, o novo diretório vira um *mero atalho* do diretório anterior, pois a variável lista é uma referência e precisa ter seus dados copiados também.



Para testar todas as classes acima, usamos um programa que cria alguns arquivos, alguns diretórios e pede aos objetos que informem o seu caminho e o seu conteúdo, se forem diretórios. Esta classe tem apenas um método `main`.

Copyright © 1999, Helder L. S. da Rocha. Todos os direitos reservados. EP:SF-Rio/D99

```
/**
 * Esta classe é um aplicativo que usa os objetos
 * Diretorio e Arquivo para simular um sistema de
 * Arquivos.
 */

import sa.*; // é necessário importar porque esta classe
             // não pertence ao pacote sa.

public class SimSA {

   public static void main (String[] args) {

      ///// Objetos da simulação /////
      Diretorio root, java, awt, lang, event;
      Arquivo frameJ, frameC, buttonJ, stringJ,
              threadJ, actionT, mouseJ;

      /* criando diretórios e arquivos */
      root = new Diretorio ("/", null);   // root não tem pai
      java = new Diretorio ("java", root);
      awt = new Diretorio ("awt", java);
      lang = new Diretorio ("lang", java);
      event = new Diretorio ("event", awt);

      frameJ = new Arquivo("Frame", "java", awt, 1200);
      frameC = new Arquivo("Frame", "class", awt, 4700);
      buttonJ = new Arquivo("Button", "java", awt, 970);
      stringJ = new Arquivo("String", "java", lang, 11902);
      threadJ = new Arquivo("Thread", "java", lang, 7760);
      actionT = new Arquivo("ActionEvent", "txt", event, 3140);
      mouseJ = new Arquivo("MouseAdapter", "java", event, 5790);

      System.out.println("\nDIR /:\n");
      Arquivo[] list3 = root.dir();
      for (int i = 0; i < list3.length; i++) {
         System.out.println(list3[i]);
      }

      System.out.println("\nDIR /java:\n");
      Arquivo[] list4 = java.dir();
      for (int i = 0; i < list4.length; i++) {
         System.out.println(list4[i]);
      }
      System.out.println("\nDIR /java/awt:\n");
      Arquivo[] list1 = awt.dir();
      for (int i = 0; i < list1.length; i++) {
         System.out.println(list1[i]);
      }

      System.out.println("\nDIR /java/awt/*.java:\n");
```

Copyright © 1999, Helder L. S. da Rocha. Todos os direitos reservados. EP:SF-Rio/D99

```
        Arquivo[] list2 = awt.dir("java");
        for (int i = 0; i < list2.length; i++) {
           System.out.println(list2[i]);
        }

        System.out.println("\nCAMINHOS RECURSIVOS:");
        System.out.println("actionT.caminho(): " + actionT.caminho());
        System.out.println("threadJ.caminho(): " + threadJ.caminho());
        System.out.println("event.caminho(): " + event.caminho());

     }
  }
```

Depois de compiladas todas as classes, podemos testar o sistema rodando o programa acima. Obteremos o seguinte resultado:

```
E:\Programas\> java SimSA

DIR /:

[1] java/       (1024) 2 arquivos

DIR /java:

[2] awt/        (1024) 4 arquivos
[3] lang/       (1024) 2 arquivos

DIR /java/awt:

[4] event/      (1024) 2 arquivos
[5] Frame.java  (1536)
[6] Frame.class (5120)
[7] Button.java (1024)

DIR /java/awt/*.java:

[5] Frame.java  (1536)
[7] Button.java (1024)

CAMINHOS RECURSIVOS:
actionT.caminho(): /java/awt/event/ActionEvent.txt
threadJ.caminho(): /java/lang/Thread.java
event.caminho(): /java/awt/event/

E:\Programas\>
```

Para criar estas classes foi preciso saber como era a estrutura dos métodos `clone`, `equals` e `toString`, que sobrepomos de `java.lang.Object`. Estas informações estão disponíveis na documentação da API. Aponte seu browser para a URL local `file:///c|jdk1.2/docs/`. Uma outra fonte de informações são os códigos-fonte, que estão disponíveis para todas as classes. Veja em `c:\jdk1.2\src\`.

Copyright © 1999, Helder L. S. da Rocha. Todos os direitos reservados. EP:SF-Rio/D99

Exercícios

1. Construa uma classe `Astro`, que representa um corpo celeste. Todo astro tem um nome e um outro astro ao redor do qual orbita. Crie também uma classe `Estrela`, que é um astro, e tem uma magnitude e uma cor. Sobreponha o método `Object.toString()` em `Astro` que imprima seu nome e a mesma informação sobre o astro que orbita. Sobreponha este método em `Estrela` para que imprima também a magnitude e a cor.
2. Teste o programa acima em uma classe `SistemaSolar`. Você deve criar objetos `sol`, `mercurio`, `venus`, `terra`, `marte`, `lua`, `phobos` e `deimos` (luas de Marte). Use `toString` para imprimir os resultados.
3. Repita o exercício anterior usando `SistemaAlphaCentauri`. Invente uns 5 planetas extras e acrescente mais duas estrelas, que devem orbitar a maior (Alpha Centauri é um sistema triplo). Pode haver mais planetas orbitando cada estrela, luas orbitando luas, etc.
4. Sobreponha `Object.clone()` em `Astro`. Garanta que quando um astro for clonado, ele continue orbitando o mesmo astro.

Classes, métodos e variáveis finais

Algumas classes não podem ter subclasses. São as classes declaradas com o modificador `final`. Se você tentar estender uma delas, o compilador acusará um erro. Declare suas classes como finais se você não deseja que seu comportamento seja alterado por subclasses. Classes consideradas "prontas", que ninguém deve estender, por decisão de projeto, devem ser declaradas `final`.

```
public final class Terminada { ... }
```

Você pode desejar apenas que um certo método não seja sobreposto pelas subclasses. Por exemplo, em um classe que faz verificação de senhas há um método chamado `valida()` que retorna `true`, se a senha for correta e false, se ela for false. Uma outra classe poderia estender esta classe e corrompê-la totalmente, sobrepondo `valida()` e fazendo-a sempre retornar `true`! Nesses casos, pode-se definir método como final, da forma:

```
public final valida(double senha) { ... }
```

Se um método for definido como `final` ou `static`, ele não pode sobreposto em uma subclasse. Em classes finais, todos os métodos são implicitamente finais.

Copyright © 1999, Helder L. S. da Rocha. Todos os direitos reservados. EP:SF-Rio/D99

Variáveis definidas como `final`, são consideradas constantes e não podem ter seus valores redefinidos, se forem tipos primitivos, ou não podem referir-se a outros objetos, se forem referências. Seus valores devem ser atribuídos no momento da criação e declaração:

```
pubic static final SQRT2 = 1.41421356
```

Classes e métodos abstratos

O que é um veículo? Alguém já viu um veículo na rua? Descreva um veículo: qual a sua cor, como se locomove?

Não é possível dar respostas claras e únicas a estas perguntas porque um veículo é um termo abstrato. Você até pode definir alguma coisa sobre um veículo. Ele deve ter uma capacidade de carga, deve ter uma velocidade. Se você implementar um modelo em Java, seu código até pode definir métodos que retornam estes valores, mas não pode definir com precisão outros que realizam coisas mais específicas como dar partida, virar à esquerda, etc. Veleiros, por exemplo, são veículos que viram à esquerda empurrando o leme para a direita!

Java permite que sejam criadas classes não totalmente especificadas, e você não precisa preencher o corpo dos métodos que fazem coisas abstratas. Basta declará-los como `abstract`, que eles não precisam ter corpo algum:

```
abstract void sigaParaEsquerda(int quantidade);
```

Métodos declarados como `abstract` não têm corpo. Eles devem terminar em ponto-e-vírgula. A conseqüência é que não se pode mais criar objetos com esta classe, já que parte dela não está especificada. Sendo assim, é preciso declarar toda a classe também `abstract`, sinalizando que ela não está completamente especificada.

```
public abstract class Veiculo { ... }
```

Na seção anterior vimos classes que não podiam ser estendidas. Com as classes abstratas ocorre o contrário. Elas *precisam* ser estendidas. E a classe que as estende precisa implementar (sobrepor) *todos* os métodos não implementados. Se isto não for feito, a subclasse também terá que ser declarada `abstract`.

Para que servem então as classes abstratas? Observe que as classes abstratas são classes e assim definem um *tipo de dados*. Elas só não podem ser construídas como objetos, mas seus métodos estáticos podem ser invocados e elas podem ser usadas para declarar tipos abstratos. Isto é útil, por exemplo, quando se constrói

Copyright © 1999, Helder L. S. da Rocha. Todos os direitos reservados. EP:SF-Rio/D99

um método que recebe um argumento e não se sabe exatamente o que ele vai receber em tempo de execução.

Suponha que tenhamos uma classe `Estacionamento`. Esta classe tem um método `admite()` que recebe um carro como argumento. Não seria muito interessante ter um método para cada tipo de carro existente, mesmo que fossem poucos, pois no dia seguinte em que o programa fosse rodar em um estacionamento, já poderia haver um novo modelo não suportado. Para resolver isto, usamos a classe abstrata `Carro`:

```
public abstract class Carro {
    public void estaciona();
}
```

Na nossa classe estacionamento, o método `admite` pode receber qualquer objeto do tipo Carro:

```
public boolean admite(Carro c) {
 c.estaciona();
}
```

Então, durante a operação do estacionamento, aparece um `Fusca fuscaBUM3456`, que é uma extensão de `Carro`. Ele é aceito sem problemas:

```
myPark.admite(fuscaBUM3456); // myPark é instância de Estacionamento
```

Depois, uma `FerrariF40 f40BZX2934` e uma `Charrete ch09`, ambos extensões de Carro:

```
myPark.admite(f40BZX2934);
myPark.admite(ch09);
```

Em cada classe `Fusca`, `Charrete` e `FerrariF40` o método `estaciona` deve estar implementado. Cada qual com suas particularidades, de forma que seja possível receber qualquer um desses veículos no estacionamento.

Interfaces

Herança múltipla pode ser uma característica desejável. Se você tem duas classes: `Gambá` e `Pato`, e quer fazer um `Ornitorrinco`, poderia ganhar tempo aproveitando algumas características dos dois bichos, não é? Mas Java não suporta a herança de múltiplas classes por ser uma fonte de muito código obscuro que pode levar a erros difíceis de detectar. A linguagem preservou,

Copyright © 1999, Helder L. S. da Rocha. Todos os direitos reservados. EP:SF-Rio/D99

porém, uma solução que mantém as boas qualidades da herança múltipla e evita suas desvantagens. São as *interfaces*.

Interfaces são uma espécie de classes abstratas onde *todos* os métodos são abstratos. Interfaces também não têm métodos estáticos nem variáveis. Além de declarações vazias de métodos pode haver apenas valores constantes.

A principal diferença de sintaxe entre interfaces e classes abstratas está na declaração. Uma interface deve ser declarada `interface`, e não `abstract class`. Uma interface também pode estender outras interfaces, mas não pode estender classe alguma. As declarações a seguir são válidas para interfaces.

```
interface Cockpit { ...}
public interface Painel { ... }
interface PaineldeCessna extends Painel { ... }
```

Os métodos abstratos declarados nas interfaces não precisam ter o modificador `abstract`.

Interfaces também definem *tipos de dados* mas não podem ser usadas para criar objetos. Para usar uma interface, uma outra classe deve *implementar* todos os seus métodos. Isto não é feito através da palavra-chave `extends`, mas usando `implements`.

```
public class Cessna implements Cockpit { ... }
```

A classe que implementa uma interface passa a ter uma relação "*é um*" com aquela interface, ou seja, o tipo da interface pode ser usado para representar objetos criados com aquela classe. Além disso, diferentemente da cláusula `extends`, `implements` pode receber mais de uma interface como argumento. As interfaces devem ser separadas por vírgulas. Uma classe pode ao mesmo tempo estender outra classe e implementar múltiplas interfaces.

```
public class Cessna extends AviãoHelice
             implements PaineldeCessna, Cockpit { ... }
```

Neste exemplo, *todos* os métodos existentes nas interfaces `PaineldeCessna` e `Cockpit` devem ser implementados ou o compilador irá reclamar dizendo que `Cessna` deve ser declarada `abstract`, por não definir todos os seus métodos (qualquer classe parcialmente implementada deve ser declarada `abstract`).

O uso de interfaces não caracteriza herança, mas composição. Por exemplo, podemos ter vários métodos para acionar controles de vôo (elevador, leme,

Copyright © 1999, Helder L. S. da Rocha. Todos os direitos reservados. EP:SF-Rio/D99

ailerão) declaradas na interface `Cockpit` e vários outros métodos para interpretar informações de um avião (altímetro, horizonte artificial, bússola) na interface `PaineldeCessna`. A interface sozinha não tem implementação. A classe `Cessna`, ao implementar as duas, herda os *protótipos* dos métodos. Qualquer lugar que recebe um `Cockpit`, recebe um Cessna, pois agora a referência para o `Cessna` é um `Cockpit` e um `Painel`. Qualquer `Piloto` que souber manusear um `Cockpit` e um `PaineldeCessna` conseguirá acionar os métodos de `Cessna` que controlam o avião. A implementação de todos os métodos da interface deve estar nas classes que a implementam. Se um `SimuladordeCessna` implementar as classes `Cockpit` e `PaineldeCessna`, o `Piloto` poderá interagir tão bem com ele como já fazia com o `Cessna`, pois o nome e formato dos métodos é o mesmo, mesmo que as implementações sejam diferentes.

Exemplos! Suponha que a classe `PaineldeCessna` fosse o seguinte:

```
interface PaineldeCessna {
 int altimetro();
 int horizonte();
 int bússola();
 int tacometro();

}
```

e que `Cockpit` fosse:

```
interface Cockpit {
 void partida();
 void leme(int posicao);
 void alieron(int posicao);
 void elevador(int posicao);
}
```

A declaração da classe `Cessna`, implementando as duas interfaces seria:

```
public class Cessna extends AviãoHelice
             implements PaineldeCessna, Cockpit {
...
   public int altimetro() { return altitude; }
   public int horizonte() { return horizonte; }
   public int bussola() { return direcao; }
      (...)
   public void aileron(int posicao) {
 // implementacao
   }
   public void elevador(int posicao) {
 // implementacao
```

Copyright © 1999, Helder L. S. da Rocha. Todos os direitos reservados. EP:SF-Rio/D99

```
    }
}
```

Se amanhã surgir um novo modelo de `Cessna`, o Piloto vai conseguir voá-lo tão bem quanto o antigo, desde que ele implemente as interfaces que já são familiares ao piloto. A interface é como o painel de um automóvel. Quem dirige um carro, dirige todos, já que todos têm direção, velocímetro, freio, etc. mesmo que a implementação de tais "métodos" seja diferente nos vários modelos.

As interfaces são, portanto, formas excelentes de *reutilizar* código e separar o *design* da implementação. Com ela, pode-se desenvolver programas extensíveis que permitem ter suas implementações alteradas no futuro sem precisar alterar dos componentes que já os utilizam.

Interfaces também são essenciais quando se precisa de herdar protótipos de mais de uma classe. São essenciais na criação de applets que precisam de funcionalidade adicional, como *multithreading*. Todo applet precisa estender a classe `java.lang.Applet` e para suportar *multithreading*, precisaria ser um `Thread` também. Como não há herança múltipla em Java, a única forma de implementar um objeto que cria *threads* a partir de si mesmo é através de interfaces.

O exemplo abaixo mostra como é possível utilizar uma interface para tornar um programa independente de forma de armazenamento. Inicialmente, temos uma interface chamada de `Armazenavel`:

```
public interface Armazenavel  {

    /** Escreve em qualquer lugar */
    public boolean save(String txt);

}
```

Esta interface é usada por um programa que pretende salvar um objeto do tipo `String`, em lugar ainda não definido. Pode ser em disco, em um banco de dados, em uma máquina remota, não interessa. Para o programa a seguir, ele está salvando em um objeto `Armazenavel`.

```
public class SalvaTextos implements Serializable {

    private transient Armazenavel peer;

    /** salva em disco */
    public boolean salvaString() {
```

Copyright © 1999, Helder L. S. da Rocha. Todos os direitos reservados. EP:SF-Rio/D99

```
        try {
            Armazenavel np = getStorage();
            if (np.save(String t)) {
                return true;
            } else return false;
        } catch (Exception e) {
            return false;
        }
    }

    /** retorna uma fonte de armazenamento */
    private Armazenavel getStorage() {
        // obtem uma instancia de uma fonte para armazenamento
        return peer;
    }
}
```

O método `getStorage` precisa obter um objeto que implemente `Armazenavel`. Ele pode conseguir isto de várias formas. Pode ser por escolha do usuário do programa ao escolher uma opção na interface do programa, por exemplo.

Para salvar em disco, é necessário ter uma implementação que lide com as peculiaridades da escrita em disco:

```
import java.io.*;
public class Disco implements Armazenavel, Serializable {
    (... variaveis ...)
    public boolean save(String t) {
    try {
            fos = new FileOutputStream("arquivo.txt");
            os = new ObjectOutputStream(fos);
            os.writeObject(t);
            return true;
        } catch (Exception e) {
            return false;
        }
    }
}
```

Se o `getStorage` receber um objeto `Disco`, ele salva em `Armazenável` e o texto é salvo em disco. Para salvar em um banco de dados, o `getStorage` precisaria retornar um objeto `BancoDados`, que lida com a escrita em um banco de dados:

```
import java.sql.*;
public class BancoDados implements Armazenavel {
```

Copyright © 1999, Helder L. S. da Rocha. Todos os direitos reservados. EP:SF-Rio/D99

```
(... variáveis e construtor ...)

    public boolean save(String t) {

        String sql = "INSERT INTO (...)";
        try {
            stmt.executeUpdate(sql);
        } catch (SQLException e) {
            System.err.println(e.getMessage());
        }
        return true;
    }

}
```

Conversão e Coerção

Na primeira seção deste módulo vimos que podíamos fazer conversões entre tipos primitivos através de atribuições, chamadas de métodos, promoção aritmética e coerção (*casting*). Tipos de referência também podem ser convertidos mas as regras são diferentes. Não é possível converter, através de uma simples atribuição, *cast* ou operação aritmética, um tipo de referência em um tipo de dados primitivo ou vice-versa. A única exceção ocorre com a classe `String`. Mas um tipo de referência pode ser convertido em outro tipo de referência, desde que haja alguma relação (extensão ou implementação) entre eles.

As regras de conversão entre tipos de referências são complexas porque há dois níveis a considerar. Como uma referência só vai encontrar realmente seu objeto em tempo de execução, é preciso considerar as conversões feitas nas duas situações. Um tipo sempre pode ser convertido em outro implicitamente (sem a necessidade de *cast*), se esta conversão for na direção da raiz da árvore de herança. Ou seja, uma subclasse sempre pode ser passada no lugar de uma superclasse. Por exemplo, qualquer classe pode ser usada onde se aceita `Object`. O contrário não é permitido sem uma conversão explícita. Exemplificando:

```
// AirCarrier extends Ship
AirCarrier air = new AirCarrier();
Ship s = new AirCarrier();
s = air; // OK!
air = s; // Illegal!
```

Uma referência do tipo de uma superclasse pode estar sendo usada em tempo de execução para apontar para um objeto do tipo de uma determinada subclasse. Neste caso, em tempo de execução, seria válido passar a referência

Copyright © 1999, Helder L. S. da Rocha. Todos os direitos reservados. EP:SF-Rio/D99

deste objeto (superclasse) para a referência da subclasse. Isto porque o objeto na memória *é* do tipo da subclasse. Mas o compilador não sabe disto e ele só permite a conversão se o programador se assumir o risco usando uma conversão explícita (*cast*). Mais exemplos:

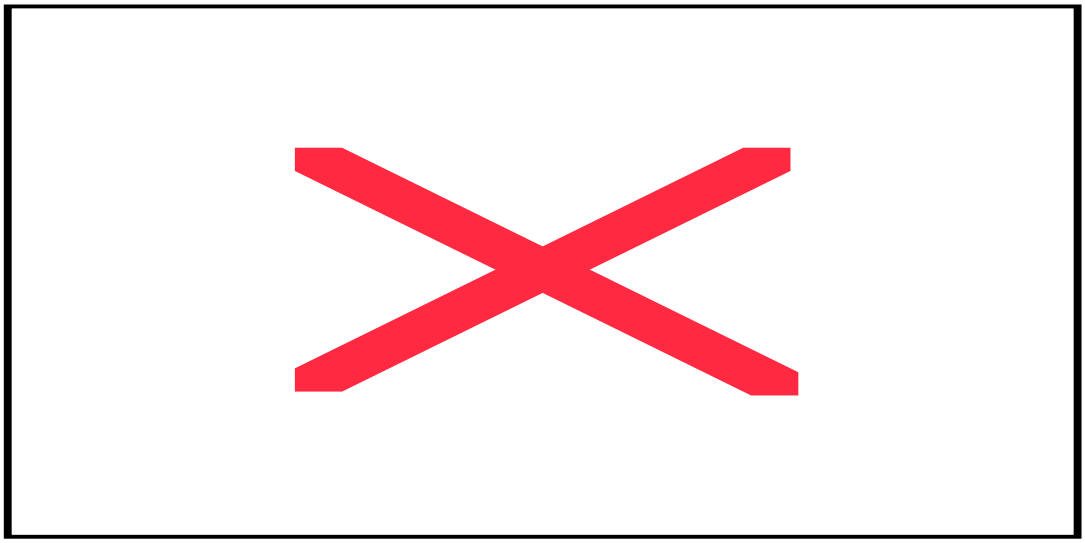
```
s = air; // Sempre OK!
air = (AirCarrier)s; // OK para o compilador!
```

O compilador sempre aceitará *casts* entre classes relacionadas. Mas pode ser que, na hora da execução, a classe não seja aquela prometida pelo programador. Neste caso, o programa lançará uma exceção.

```
Ship s = new Cruiser();
air = (AirCarrier)s; // OK para o compilador! Erro na execução!
```

No exemplo acima, como `s` não é um `AirCarrier` como esperado, haverá um erro em tempo de execução. O compilador acredita no programador e não acusa erro algum.

Toda a conversão sempre ocorre com referências. O objeto na memória nunca é afetado. É possível sempre ter várias referências para um mesmo objeto:



Às vezes o *cast* é usado simplesmente para permitir que o compilador saiba da existência de um campo ou método de um objeto. Se a sua classe `Texto` têm um método `imprime()` e um objeto do tipo `Texto` é passado para um método que aceita `Object` como argumento, a referência do tipo `Object` não vai

Copyright © 1999, Helder L. S. da Rocha. Todos os direitos reservados. EP:SF-Rio/D99

conseguir enxergar seu método. Para que isto seja possível, é necessário informar ao compilador que aquele `Object` é na verdade um `Texto`:

```
public void processa(Object o) {
    o.imprime(); // Erro de compilação! Compilador não conhece
                 // método imprime em Object!

    ((Texto)o).imprime(); // OK! Compilador agora sabe que
                          // objeto é um Texto!
```

`Object` pode ser usado para receber qualquer objeto, inclusive vetores. Vetores só podem ser convertidos em outros vetores ou em `Object` através de *cast* (desde que em tempo de execução, a referência de `Object` se refira a um vetor). Qualquer interface pode ser convertida em qualquer classe e vice-versa desde que usando *casts*.

Copyright © 1999, Helder L. S. da Rocha. Todos os direitos reservados. EP:SF-Rio/D99