EXPLICADORES.NET

2022/2023

EAGS SIN / CAP

POO

Prof. Camila Dias

PROGRAMAÇÃO ORIENTADA A OBJETO

Bibliografia e conteúdo

SINTES, Anthony.Aprenda Programação Orientada a Objeto em 21 Dias.
São Paulo:
Makron Books, 2002.

PROGRAMAÇÃO ORIENTADA A OBJETOS

Introdução à programação orientada a objetos.
Encapsulamento.
Método.
Propriedades.
Construtores.
Herança.
Polimorfismo.
Introdução à UML.
Introdução à Análise Orientada a Objetos.
Introdução ao Projeto Orientado a Objetos.
Reutilizando projetos através de padrões de projeto.
Padrões avançados de projeto.
OO e programação de interface com o usuário.
Construindo software confiável através de testes.
Prática da orientação a objetos.

Linguagem de Programação

Linguagem de Programação é uma linguagem escrita e formal que especifica um conjunto de instruções e regras usadas para gerar programas (software). Um software pode ser desenvolvido para rodar em um computador, dispositivo móvel ou em qualquer equipamento que permita sua execução. Existem várias linguagens e elas servem para muitos propósitos.

É uma linguagem formal que, através de uma série de instruções, permite que um programador escreva um conjunto de ordens, ações consecutivas, dados e algoritmos para criar programas que controlam o comportamento físico e lógico de uma máquina.

Programador e máquina se comunicam por meio dessa linguagem, permitindo especificar, com precisão, aspectos como:
•quais dados um software deve operar;
•como esses dados devem ser armazenados ou transmitidos;
•quais ações o software deve executar, de acordo com cada circunstância variável.

A linguagem de programação é um sistema de comunicação estruturado, composto por conjuntos de símbolos, palavras-chave, regras semânticas e sintáticas que permitem o entendimento entre um programador e uma máquina.

Linguagem de Programação

Linguagem compilada x interpretada

1- Compilada

Em linguagens compiladas como em C++ o programa escrito é processado por um compilador, que gera código de máquina que depois é executado pelo usuário no computador. Por exemplo, quando o código de um programa em C++ é escrito pelo programador possui instruções como declarações de variáveis, loops e etc. Quando termina de escrever o código o programador utiliza um compilador para converter o código escrito em um arquivo, por exemplo um arquivo .EXE, que contém instruções de máquina e que pode ser executado no Windows.
Exemplos de linguagens compiladas:

- C++
- C#
- Java

2 - Interpretada

Em linguagens interpretadas como JavaScript o código escrito pelo programador não passa por nenhuma etapa de compilação ao fim do desenvolvimento. Quando o código é executado (por exemplo, quando o usuário clica em um botão em uma página Web) o interpretador entra em ação e converte aquela parte do código necessária em instrução de máquina que são processadas pelo computador do usuário.

- Javascript
- Python

Linguagem de Programação

Linguagem fortemente tipada x fracamente tipada

1- Fortemente Tipada

Em linguagens de tipagem estática ou fortemente tipada como em C++, C# ou Java os tipos das variáveis de um programa são explicitamente definidos no código e não podem ser modificados depois da sua declaração.
Exemplos de linguagens com tipagem estática:

- C++
- C#
- Java

2 – Fracamente Tipada

Em linguagens de tipagem dinâmica como Javascript é possível declarar uma variável de um tipo e depois modifica-lo.

Exemplos:

- JavaScript
- PHP

Linguagem de Programação

Paradigmas de programação

1 - Programação estruturada

Programação Estruturada (PE) é um padrão ou paradigma de programação da engenharia de softwares, com ênfase em sequência, decisão e, iteração (sub-rotinas, laços de repetição, condicionais e, estruturas em bloco), criado no final de 1950 junto às linguagens ALGOL 58 e ALGOL 60.
É formada por três estruturas:
•Sequência: a tarefa é executada logo após a outra;
•Decisão: a tarefa é executada após um teste lógico, e;
•Iteração: a partir do teste lógico, um trecho do código pode ser repetido finitas vezes.
Exemplos de linguagens estruturadas:
•Cobol
•Diversas linguagens relevantes hoje (e.g. Cobol, PHP, Perl e Go) ainda utilizam o paradigma estruturado, embora possuam suporte para a orientação ao objeto e para outros paradigmas de programação.

2 - Programação Orientada a objetos

Programação Orientada a Objetos (também conhecida pela sua sigla POO) é um modelo de análise, projeto e programação de software baseado na composição e interação entre diversas unidades chamadas de 'objetos'. Os programas são arquitetados através de objetos que interagem entre si. Dentre as várias abordagens da POO, as baseadas em classes são as mais comuns: objetos são instâncias de classes, o que em geral também define o tipo do objeto. Cada classe determina o comportamento (definido nos métodos) e estados possíveis (atributos) de seus objetos, assim como o relacionamento com outros objetos.
Exemplos de linguagens orientada a objetos:
•C++
•C#
•Java

Linguagem de Programação

Tipos de Linguagens de Programação

1- Linguagem de programação de baixo nível

São linguagens totalmente orientadas à máquina. Esse idioma serve como uma interface e cria um link inseparável entre hardware e software.
Além disso, exerce controle direto sobre o equipamento e sua estrutura física. Para aplicá-la adequadamente, é necessário que o programador conheça muito bem o hardware. Essa categoria pode ser subdividido em dois tipos:

Linguagem de máquina

É o mais primitivo dos idiomas e é uma coleção de dígitos ou *bits* binários (0 e 1) que o computador lê e interpreta e é o único idioma que os computadores entendem.

Exemplo: 10110000 01100001

Linguagem de Programação

Tipos de Linguagens de Programação

1- Linguagem de programação de baixo nível

Linguagem *Assembly*

A linguagem Assembly é a primeira tentativa de substituir a linguagem de máquina por uma mais próxima da usada por seres humanos.
Um programa escrito nessa linguagem é armazenado como texto (como nos programas de alto nível) e consiste em uma série de instruções que correspondem ao fluxo de pedidos executáveis por um microprocessador.
No entanto, essas máquinas não entendem a linguagem *Assembly*. Portanto, devem ser convertidas em linguagem de máquina por meio de um programa chamado *Assembler*.
Ele gera códigos compactos, rápidos e eficientes criados pelo programador que tem controle total da máquina.

Exemplo: **MOV AL, 61h** (atribui o valor hexadecimal 61 ao registro “AL”)

Linguagem de Programação

Tipos de Linguagens de Programação

2- Linguagem de programação de alto nível

Elas visam facilitar o trabalho do programador, pois usam instruções que são mais fáceis de serem entendidas.
Além disso, a linguagem de alto nível permite que você escreva códigos usando os idiomas que conhece (português, espanhol, inglês etc.) traduzindo-os em seguida para o idioma da máquina por tradutores ou compiladores.

Tradutor

Eles traduzem programas escritos em uma linguagem de programação para a linguagem de máquina do computador e são executados à medida que são traduzidos.

Compilador

Ele permite que você traduza um programa inteiro de uma só vez, tornando-o mais rápido e pode ser armazenado para uso posterior sem a necessidade de uma nova tradução.

Linguagem de Programação

Quais softwares de programação existem?

Por software de programação entendemos o conjunto de todas as ferramentas que permitem ao programador criar, escrever códigos, depurar, manter e empacotar projetos.

Conheça a seguir alguns dos diferentes programas pelos quais o projeto deve passar para ser administrado:

Editores de código ou texto: Ao escrever os códigos, eles se completam marcando os erros sintáticos e a refatoração.

Compiladores: Como mencionado acima, eles convertem o código digitado à linguagem de máquina, gerando um código binário executável.

Scrubbers: Eles servem para otimizar o tempo de desenvolvimento e ajudam a corrigir erros por meio do monitoramento da execução de um programa, dos valores de determinadas variáveis e da referência a objetos na memória.

Linguagem de Programação

Quais softwares de programação existem?

Linkers

Este programa pega objetos gerados nas primeiras etapas do processo de compilação e os recursos necessários da biblioteca, remove os processos e dados de que não precisa e vincula o código à referida biblioteca para aumentar seu tamanho e extensão.

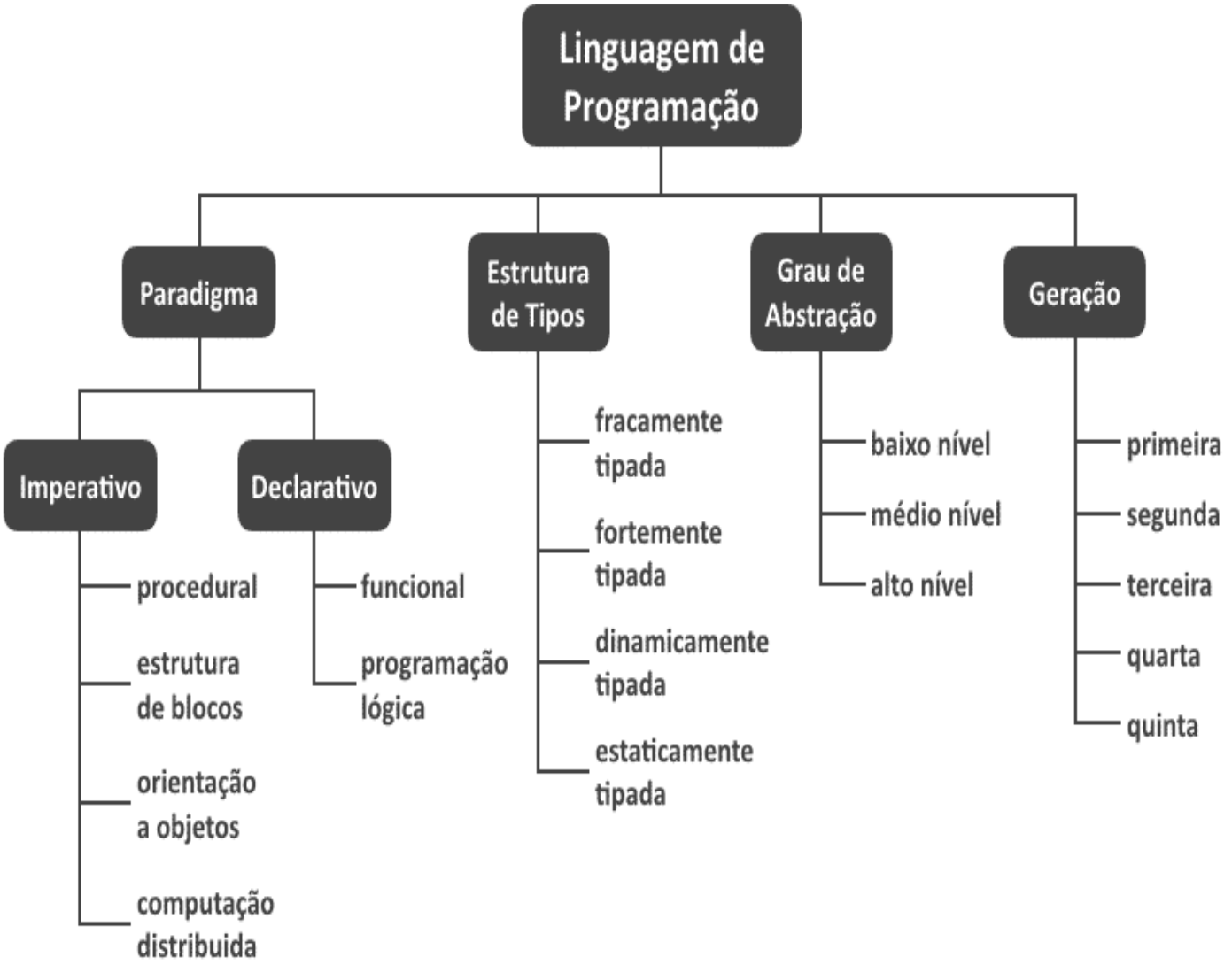
Intérpretes ou tradutores

Conforme você lê este artigo, o tradutor (ou intérprete) carrega o código digitado e converte as instruções para que o programa possa ser executado.

IDE

O IDE (*Integrated Development Environment*) ou Entorno de Desenvolvimento Integrado, é um aplicativo de computador que fornece uma série de serviços que facilitam a programação de software, como:

•funções de preenchimento automático;
•um editor de código fonte;
•gerenciamento de conexão com banco de dados;
•integração com sistemas de controle de versão;
•simuladores de dispositivos;
•um depurador para acelerar o processo de desenvolvimento de software, entre outros.


Linguagem de Programação
Paradigma
Imperativo
procedural
estrutura de blocos
orientação a objetos
computação distribuida
Declarativo
funcional
programação lógica
Estrutura de Tipos
fracamente tipada
fortemente tipada
dinamicamente tipada
estaticamente tipada
Grau de Abstração
baixo nível
médio nível
alto nível
Geração
primeira
segunda
terceira
quarta
quinta

PROGRAMAÇÃO ORIENTADA A OBJETOS

A POO dá o próximo passo lógico após a programação modular, adicionando herança e polimorfismo ao módulo.

A POO estrutura um programa, dividindo-o em vários objetos de alto nível.

Cada objeto modela algum aspecto do problema que você está tentando resolver. Escrever listas sequenciais de chamadas de procedimentos para dirigir o fluxo do programa não é mais o foco da programação sob a OO.

Em vez disso, os objetos interagem entre si, para orientar o fluxo global do programa. De certa forma, um programa OO se torna uma simulação viva do problema que está tentando resolver.

PROGRAMAÇÃO ORIENTADA A OBJETOS

Definir um programa em termos de objetos é uma maneira profunda de ver o software.

Os objetos o obrigam a ver tudo, em nível conceitual, do que o objeto faz: seus comportamentos.

Ver um objeto a partir do nível conceitual é um desvio da observação de como algo é feito: a implementação.

Essa mentalidade obriga a pensar em seus programas em termos naturais e reais. Em vez de modelar seu programa como um conjunto de procedimentos e dados separados (termos do mundo do computador), você modela seu programa em objetos. Os objetos permitem que você modele seus programas nos substantivos, verbos e adjetivos do domínio de seu problema.

A **implementação** define como algo é feito. Em termos de programação, implementação É O CÓDIGO.

Domínio é o espaço onde o problema reside. O domínio é o conjunto de conceitos que representam os aspectos importantes do problema que você está tentando resolver.

PROGRAMAÇÃO ORIENTADA A OBJETOS

Um objeto é uma construção de software que encapsula estado e comportamento. Os objetos permitem que você modele seu software em termos reais e abstrações.

Rigorosamente, um objeto é uma instância de uma classe.

Assim como o mundo real é constituído de objetos, da mesma forma é o software orientado a objetos. Em uma linguagem de programação OO pura, tudo é um objeto, desde os tipos mais básicos, como inteiros e lógicos, até as instâncias de classes mais complexas. Nem todas as linguagens Orientadas a Objetos chegam a esse ponto.

Em algumas linguagens, como Java, primitivas como int e float, não são tratadas como objetos.

PROGRAMAÇÃO ORIENTADA A OBJETO

Embora as linguagens orientadas a objetos já existam desde a década de 1960, nos últimos 10 anos temos visto um crescimento sem paralelo no uso e na aceitação de tecnologias de objeto, por todo setor de software.

Embora tenham começado como algo secundário, sucessos recentes, como Java, CORBA e C++, têm impulsionado as técnicas orientadas a objetos (OO) para novos níveis de aceitação.

PROGRAMAÇÃO ORIENTADA A OBJETO

Histórico

OO – Orientado a objetos. É um termo geral que inclui **qualquer** estilo de desenvolvimento que seja baseado no conceito de "objeto" –
uma entidade que exibe as práticas e comportamentos.

Pode-se aplicar uma estratégia orientada a objetos na programação, assim como na análise e no projeto.

A Orientação a objetos é um estado da mente, uma maneira de ver o mundo todo em termos de objetos.

Antigamente a programação era engenhosa, os programadores introduziam os programas diretamente na memória principal do computador, através de bancos de chaves (switches).

Os programadores escreviam seus programas em linguagens binárias. Tal programa em linguagem binária era extremamente propensa a erros e falta de estrutura tomou a manutenção do código praticamente impossível. Além disso a linguagem binária não era muito acessível.

PROGRAMAÇÃO ORIENTADA A OBJETO

Histórico

Quando os computadores se tornaram mais comuns, linguagens de nível mais alto e procedurais começaram a aparecer, a primeira foi a FORTRAN, ALGOL.

As linguagens procedurais permitem ao programador reduzir um programa em procedimentos refinados para processar dados.

Esses procedimentos refinados definem a estrutura global do programa.

Chamadas sequenciais a esses procedimentos geram a execução de um programa procedural.

O programa termina quando acaba de chamar sua lista de procedimentos

Um problema neste tipo de linguagem, é que **limita a reutilização de código,** e com muita frequência os programadores produzem códigos de espaguete.

PROGRAMAÇÃO ORIENTADA A OBJETO

Histórico

A **programação modular**, tenta melhorar algumas das deficiências encontradas na linguagem procedural, dividindo os programas em vários componentes ou módulos constituintes. Ao contrário da linguagem procedural, que separa dados e procedimentos, os módulos combinam os dois.

Um módulo consiste em dados e procedimentos para manipular esses dados. Quando outras partes do programa precisam usar um módulo, elas simplesmente exercitam a interface do módulo.

Como os módulos ocultam todos os dados internos do restante do programa, é fácil introduzir a ideia de estado: um módulo têm informações de estado que podem mudar a qualquer momento. Porém a programação modular sofre duas deficiências. Os módulos não são extensíveis, significando que você não pode fazer alterações incrementais em um módulo sem abrir o código a força e fazer as alterações diretamente. Você também não pode basear um módulo no outro, a não ser através de delegação. Embora um módulo possa definir um tipo, um módulo não pode compartilhar o tipo de outro módulo.

PROGRAMAÇÃO ORIENTADA A OBJETO

Histórico

Nas linguagens modulares e procedurais, os dados estruturados e não estruturados têm um ‘tipo’.

O tipo é mais facilmente pensado como o formato da memŕoria para os dados. As linguagens **fortemente tipadas** exigem que cada objeto tenha um tipo específico e definido. Entretanto, os tipos não podem ser estendidos para criar outro tipo, exceto através de um estilo chamado ‘agregação’.

Finalmente, a programação modular também é um híbrido procedural que ainda divide um programa em vários procedimentos. Agora, em vez de atuar em dados brutos, esses procedimentos manipulam módulos.

NOTAS DO LIVRO

O estado de um objeto – é o significado combinado das variáveis internas do objeto

Uma variável interna – é um valor mantido dentro de um objeto

PROGRAMAÇÃO ORIENTADA A OBJETO

A POO dá o próximo passo lógico após a programação modular, adicionando herança e polimorfismo ao módulo.

A POO estrutura um programa, dividindo-o em vários objetos de alto nível.

Cada objeto modela algum aspecto do problema que você está tentando resolver. Escrever listas sequenciais de chamadas de procedimentos para dirigir o fluxo do programa não é mais o foco da programação sob a OO.

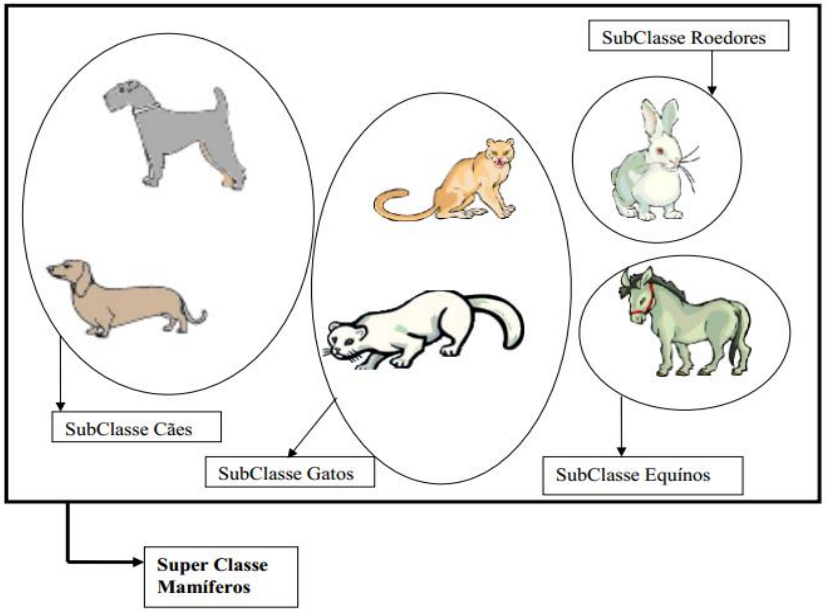
Em vez disso, os objetos interagem entre si, para orientar o fluxo global do programa. De certa forma, um programa OO se torna uma simulação viva do problema que está tentando resolver.

PROGRAMAÇÃO ORIENTADA A OBJETO

O que é uma CLASSE?

Assim como os objetos do mundo real, o mundo da POO agrupa os objetos pelos seus comportamentos e atributos comuns.

A biologia classifica todos os cães, gatos, elefantes e seres humanos como mamíferos. Características compartilhadas dão a essas criaturas separadas um senso de comunidade. No mundo do software, as classes agrupam objetos relacionados da mesma maneira.



NOVO TERMO Uma *classe* define os atributos e comportamentos comuns compartilhados por um tipo de objeto. Os objetos de certo tipo ou classificação compartilham os mesmos comportamentos e atributos. As classes atuam de forma muito parecida com um cortador de molde ou biscoito, no sentido de que você usa uma classe para criar ou *instanciar* objetos.

NOVO TERMO *Atributos* são as características de uma classe visíveis externamente. A cor dos olhos e a cor dos cabelos são exemplos de atributos.

Um objeto pode expor um atributo fornecendo um link direto a alguma variável interna ou retornando o valor através de um método.

NOVO TERMO *Comportamento* é uma ação executada por um objeto quando passada uma mensagem ou em resposta a uma mudança de estado: é algo que um objeto faz.

Um objeto pode exercer o *comportamento* de outro, executando uma operação sobre esse objeto. Você pode ver os termos *chamada de método*, *chamada de função* ou *passar uma mensagem*, usados em vez de executar uma *operação*. O que é importante é que cada uma dessas ações omite o comportamento de um objeto.

NOTA

Passagem de mensagem, operação, chamada de método e *chamada de função;* o quê você usa, freqüentemente depende de seus conceitos anteriores.

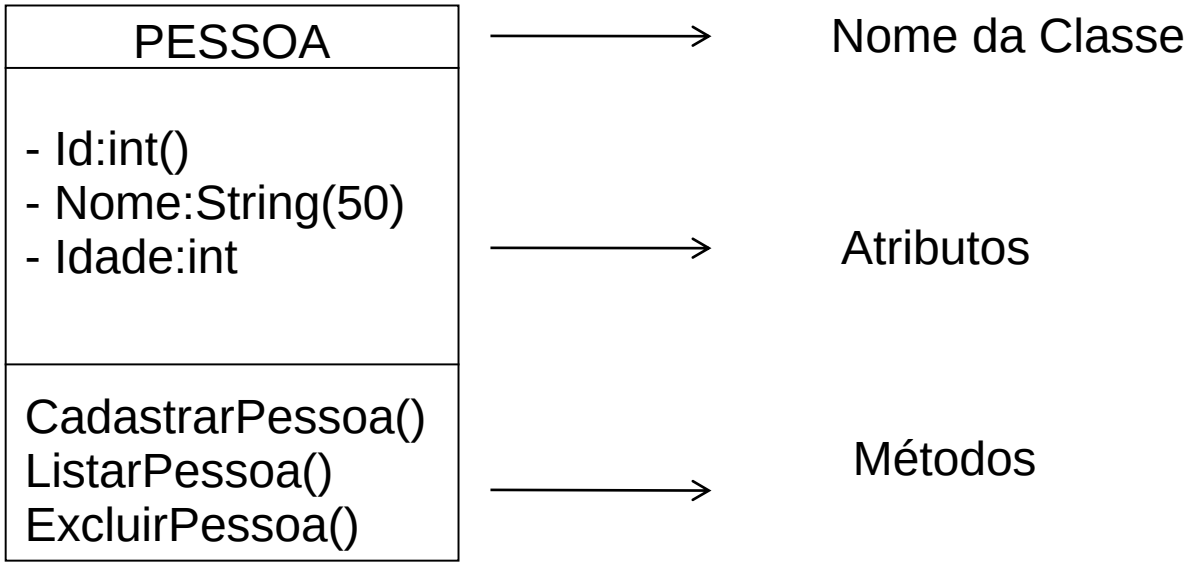
Pensar em termos de *passagem de mensagem* é uma maneira muito orientada a objetos de pensar. A *passagem de mensagem* é dinâmica. Conceitualmente, ela separa a *mensagem* do objeto. Tal mentalidade pode ajudar a pensar a respeito das interações entre objetos.

Linguagens como C++ e Java têm heranças procedurais, onde as chamadas de função são estáticas. Como resultado, essas linguagens freqüentemente se referem a um objeto realizando uma *chamada de método* a partir de outro objeto. Uma *chamada de método* está fortemente acoplada ao objeto.

Este livro normalmente usará *chamada de método*, devido à sua forte ligação com Java. Entretanto, podem existir ocasiões em que o termo *mensagem* é usado indistintamente.

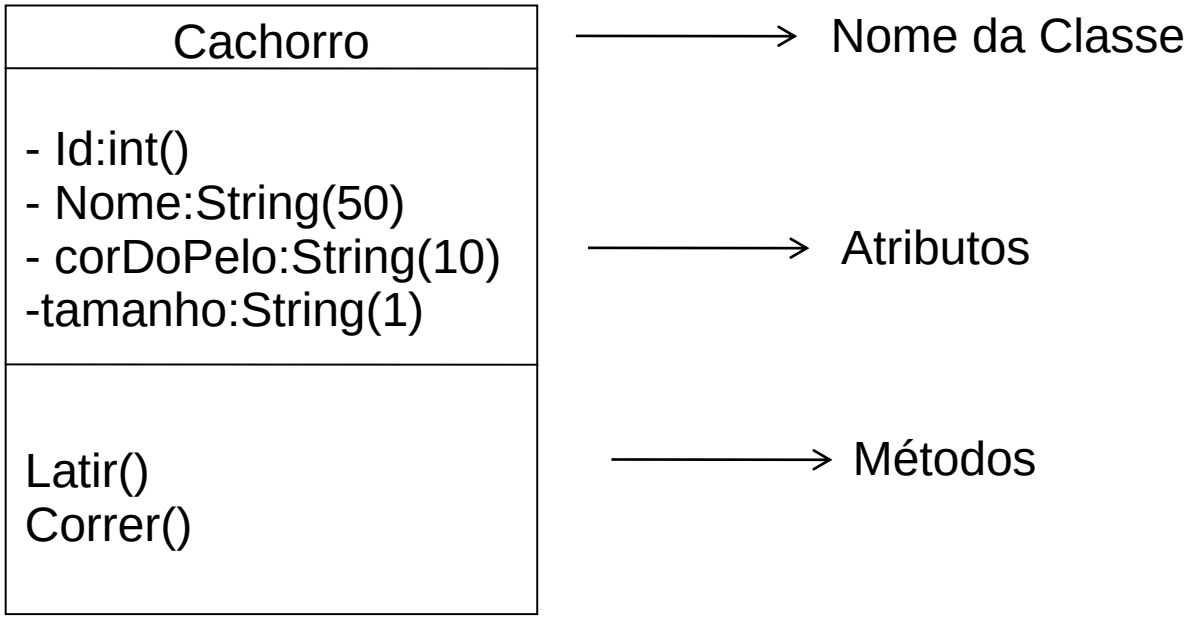
PROGRAMAÇÃO ORIENTADA A OBJETO

Representação de uma Classe

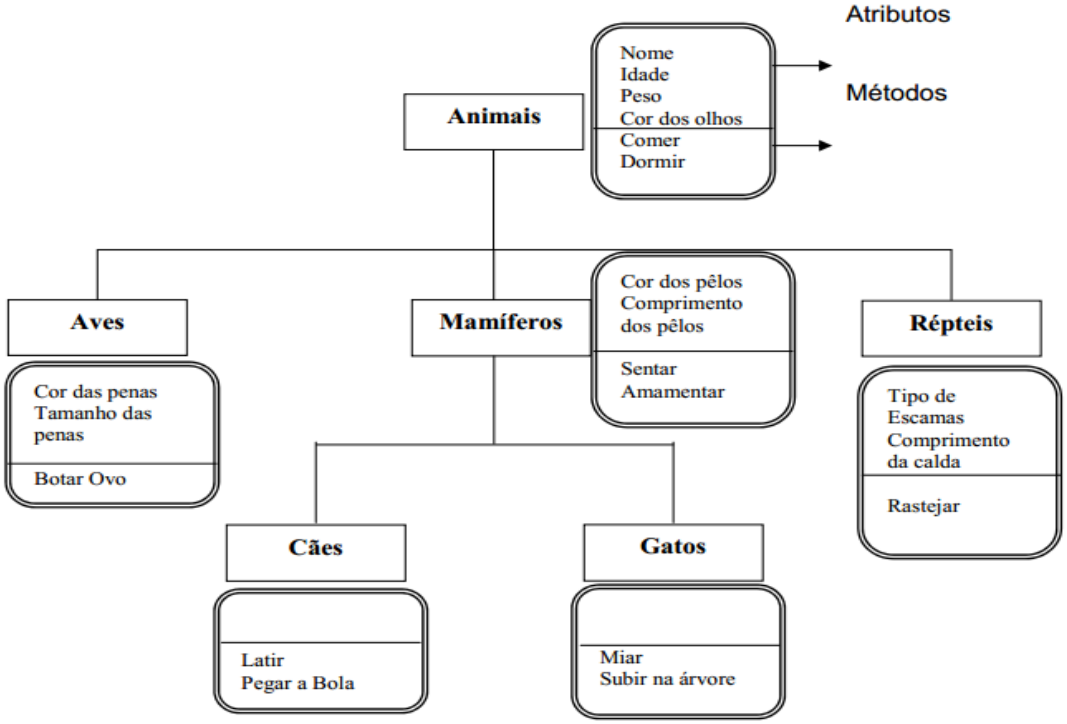


PROGRAMAÇÃO ORIENTADA A OBJETO

Representação de uma Classe



PROGRAMAÇÃO ORIENTADA A OBJETO



Uma classe define todas as características comuns a um tipo de objeto. Especificamente, a classe define todos os atributos e comportamentos expostos pelo objeto. A classe define a quais mensagens seus objetos respondem. Quando um objeto quer exercer o comportamento de outro objeto, ele não faz isso diretamente, mas pede ao outro objeto para que se mude, normalmente baseado em alguma informação adicional. Frequentemente, isso é referido como ‘envio de mensagem’.

PROGRAMAÇÃO ORIENTADA A OBJETO

Métodos como

```
public Item( String id, String description, int quantity, double price )
```

são chamados *construtores*. Os construtores inicializam um objeto durante sua criação.

NOVO TERMO *Construtores* são métodos usados para inicializar objetos durante sua instanciação. Você chama a criação de objetos de *instanciação* porque ela cria uma instância do objeto da classe.

NOTA

No construtor e por todo o exemplo `Item`, você pode notar o uso de `this`. `this` é uma referência que aponta para a instância do objeto. Cada objeto tem sua própria referência para si mesmo. A instância usa essa referência para acessar suas próprias variáveis e métodos.

Métodos como `setDiscount()`, `getDescription()` e `getAdjustedTotal()` são todos comportamentos da classe `Item` que retornam ou configuram atributos. Quando um caixa quer totalizar o carrinho, ele simplesmente pega cada item e envia ao objeto a mensagem `getAdjustedTotal()`.

PROGRAMAÇÃO ORIENTADA A OBJETO

NOVO TERMO Os *acessores* dão acesso aos dados internos de um objeto. Entretanto, os *acessores* ocultam o fato de os dados estarem em uma variável, em uma combinação de variáveis ou serem calculados. Os *acessores* permitem que você mude ou recupere o valor e têm 'efeitos colaterais' sobre o estado interno.

NOVO TERMO Os *mutantes* permitem que você altere o estado interno de um objeto.

Ao serem executados, seus programas usam classes como a `Item` para criar ou instanciar os objetos que compõem o aplicativo. Cada nova instância é uma duplicata da última. Entretanto, uma vez instanciada, a instância transporta comportamentos e controla seu estado. Então, o que inicia sua vida como clone poderia se comportar de maneira muito diferente durante sua existência.

Por exemplo, se você criar dois objetos item a partir da mesma classe `Item`, um objeto item poderá ter um desconto de 10%, enquanto o segundo pode não ter desconto. Alguns itens também são um pouco mais caros que outros. Um item poderia custar US$1.000, enquanto outro poderia custar apenas US$1,98. Assim, embora o estado de um item possa variar com o passar do tempo, a instância ainda é um objeto de `Item`. Considere o exemplo da biologia; um mamífero de cor cinza é tão mamífero quanto outro de cor marrom.

PROGRAMAÇÃO ORIENTADA A OBJETO

Relacionamentos de objeto

O modo como os objetos se relacionam é um componente muito importante da POO. Os objetos podem se relacionar de duas maneiras importantes:

1- Os objetos podem existir independentemente uns dos outros. Dois objetos de Item podem aparecer no carrinho de compras simultaneamente. Se esses dois objetos separados precisarem interagir, eles interagirão passando mensagens um para o outro.
Passar uma mensagem é o mesmo que chamar um método para mudar o estado do objeto ou para exercer um comportamento.

NOVO TERMO Os objetos se comunicam uns com os outros através de *mensagens*. As mensagens fazem com que um objeto realize algo.

2- Um objeto poderia conter outros objetos. Assim, como os objetos compõem um programa em POO, eles podem compor outros objetos através da agregação. Pode-se notar que um Item pode conter muitos objetos, uma id, uma descrição que podem conter objetos String. Cada um desses objetos tem uma interface que oferece métodos e atributos.

Nota: Em POO tudo é um objeto, mesmo as partes que compõem um objeto!

Vantagens e objetivos da OO

A programação orientada a objetos define 6 objetivos propostos para desenvolvimento de software. A Poo se esmera em produzir software que tenha as seguintes características:

1. Natural
2. Confiável
3. Reutilizável
4. Manutenível
5. Extensível
6. Oportunos

PROGRAMAÇÃO ORIENTADA A OBJETO

Vantagens e objetivos da OO

Natural

A POO produz software natural. Os programas naturais são mais inteligíveis. Em vez de programar em termos de regiões de memória, você pode programar usando a terminologia de seu problema em particular. Você não precisa se aprofundar nos detalhes do computador enquanto projeta seu programa. Em vez de ajustar seus programas para a linguagem do mundo dos computadores, a OO o libera para que expresse seu programa nos termos de seu problema.

A programação orientada a objetos permite que você modele um problema em um nível funcional e não em nível de implementação. Você não precisa saber como um software funciona, para usá-lo: você simplesmente se concentra no que ele faz.

Confiável

Para criar software útil, você precisa criar software que seja tão confiável quanto outros produtos, como geladeiras e televisões. Quando foi a última vez que seu microondas quebrou?

Programas orientados a objetos, bem projetados e cuidadosamente escritos são confiáveis. A natureza modular dos objetos permite que você faça alterações em uma parte de seu programa, sem afetar outras partes. Os objetos isolam o conhecimento e a responsabilidade de onde pertencem.

Uma maneira de aumentar a confiabilidade é através de testes completos. A OO aprimora os testes, permitindo que você isole conhecimento e responsabilidade em um único lugar. Tal isolamento permite que você teste e valide cada componente independentemente. Uma vez que tenha validado um componente, você pode reutilizá-lo com confiança.

PROGRAMAÇÃO ORIENTADA A OBJETO

Vantagens e objetivos da OO

Reutilizável

Um construtor inventa um novo tipo de tijolo cada vez que constrói uma casa? Um engenheiro eletricista inventa um novo tipo de resistor cada vez que projeta um circuito? Então, por que os programadores continuam 'reinventando a roda?' Uma vez que um problema esteja resolvido, você deve reutilizar a solução.

Você pode reutilizar prontamente classes orientadas a objetos bem feitas. Assim como os módulos, você pode reutilizar objetos em muitos programas diferentes. Ao contrário dos módulos, a POO introduz a herança para permitir que você estenda objetos existentes e o polimorfismo, para que você possa escrever código genérico.

A OO não garante código genérico. Criar classes bem feitas é uma tarefa difícil que exige concentração e atenção à abstração. Os programadores nem sempre acham isso fácil.

Através da POO, você pode modelar idéias gerais e usar essas idéias gerais para resolver problemas específicos. Embora você vá construir objetos para resolver um problema específico, freqüentemente construirá esses objetos específicos usando partes genéricas.

PROGRAMAÇÃO ORIENTADA A OBJETO

Vantagens e objetivos da OO

Manutenível

O ciclo de vida de um programa não termina quando você o distribui. Em vez disso, você deve manter sua base de código. Na verdade, entre 60% e 80% do tempo gasto trabalhando em um programa é para manutenção. O desenvolvimento representa apenas 20% da equação!

Um código orientado a objetos bem projetado é manutenível. Para corrigir um erro, você simplesmente corrige o problema em um lugar. Como uma mudança na implementação é transparente, todos os outros objetos se beneficiarão automaticamente do aprimoramento. A linguagem natural do código deve permitir que outros desenvolvedores também o entendam.

Extensível

Assim como você deve manter um programa, seus usuários exigem o acréscimo de nova funcionalidade em seu sistema. Quando você construir uma biblioteca de objetos, também desejará estender a funcionalidade de seus próprios objetos.

A POO trata dessas realidades. O software não é estático. Ele deve crescer e mudar com o passar do tempo, para permanecer útil. A POO apresenta ao programador vários recursos para estender código. Esses recursos incluem herança, polimorfismo, sobreposição, delegação e uma variedade de padrões de projeto.

Oportuno

O ciclo de vida do projeto de software moderno é freqüentemente medido em semanas. A POO ajuda nesses rápidos ciclos de desenvolvimento. A POO diminui o tempo do ciclo de desenvolvimento, fornecendo software confiável, reutilizável e facilmente extensível.

O software natural simplifica o projeto de sistemas complexos. Embora você não possa ignorar o projeto cuidadoso, o software natural pode otimizar os ciclos de projeto, pois você pode se concentrar no problema que está tentando resolver.

Quando você divide um programa em vários objetos, o desenvolvimento de cada parte pode ocorrer em paralelo. Vários desenvolvedores podem trabalhar nas classes independentemente. Tal desenvolvimento em paralelo leva a tempos de desenvolvimento menores.

PROGRAMAÇÃO ORIENTADA A OBJETO

Armadilhas:

1- Nunca pense que ao usar uma linguagem OO

2 – Ter medo da reutilização

3 – Pensar na OO como uma solução para tudo

4 – Programação egoísta

PROGRAMAÇÃO ORIENTADA A OBJETO

Armadilha 1: pensar na POO simplesmente como uma linguagem

Freqüentemente, as pessoas equiparam linguagens orientadas a objetos com a POO. O erro surge ao supor que você está programando de maneira orientada a objetos simplesmente porque usa uma linguagem orientada a objetos. Nada poderia estar mais distante da realidade.

A POO é muito mais do que simplesmente usar uma linguagem orientada a objetos ou conhecer certo conjunto de definições. Você pode escrever código horrivelmente não orientado a objetos em uma linguagem orientada a objetos. A verdadeira POO é um estado da mente que exige que você veja seus problemas como um grupo de objetos e use encapsulamento, herança e polimorfismo corretamente.

Infelizmente, muitas empresas e programadores supõem que, se simplesmente usarem uma linguagem orientada a objetos, se beneficiarão de todas as vantagens que a POO oferece. Quando falham, elas tentam culpar a tecnologia e não o fato de que não treinaram seus funcionários corretamente, ou que agarraram um conceito de programação popular sem entender realmente o que ele significava.

PROGRAMAÇÃO ORIENTADA A OBJETO

Armadilha 2: medo da reutilização

Você deve aprender a reutilizar código. Aprender a reutilizar sem culpa freqüentemente é uma das lições mais difíceis de aprender, quando você escolhe a POO pela primeira vez. Três problemas levam a essa dificuldade.

Primeiro, os programadores gostam de criar. Se você olhar a reutilização de modo errado, ela parecerá afastar algumas das alegrias da criação. Entretanto, você precisa lembrar que está reutilizando partes para criar algo maior. Pode não parecer interessante reutilizar um componente, mas isso permitirá que você construa algo ainda melhor.

Segundo, muitos programadores sofrem do sentimento de 'não escrito aqui'— significando que eles não confiam no software que não escreveram. Se um software é bem testado e atende sua necessidade, você deve reutilizá-lo. Não rejeite um componente porque você não o escreveu. Lembre-se de que reutilizar um componente o liberará para escrever outro software maravilhoso.

PROGRAMAÇÃO ORIENTADA A OBJETO

Armadilha 3: pensar na OO como uma solução para tudo

Embora a POO ofereça muitas vantagens, ela não é a solução para tudo no mundo da programação. Existem ocasiões em que você não deve usar OO. Você ainda precisa usar bom senso na escolha da ferramenta correta para o trabalho a ser feito. Mais importante, a POO não garante o sucesso de seu projeto. Seu projeto não terá sucesso automaticamente, apenas porque você usa uma linguagem OO. O sucesso aparece somente com planejamento, projeto e codificação cuidadosos.

PROGRAMAÇÃO ORIENTADA A OBJETO

Armadilha 4: programação egoísta

Não seja egoísta quando programar. Assim como você deve aprender a reutilizar, também deve aprender a compartilhar o código que cria. Compartilhar significa que você encorajará outros desenvolvedores a usarem suas classes. Entretanto, compartilhar também significa que você tornará fácil para outros reutilizarem essas classes.

Lembre-se dos outros desenvolvedores quando você programar. Faça interfaces limpas e inteligíveis. Mais importante, escreva a documentação. Documente suposições, parâmetros de métodos, documente o máximo que você puder. As pessoas não reutilizarão o que não podem encontrar ou entender.

PROGRAMAÇÃO ORIENTADA A OBJETO

Classe	• Representação de um conjunto de objetos com características afins. Definição do comportamento dos objetos (métodos) e seus atributos (atributos).
Objeto	• Uma instância de uma classe. • Armazenamento de estados através de seus atributos e reação a mensagens enviadas por outros objetos.
Herança	• Mecanismo pela qual uma classe (sub-classe) pode estender outra classe (super-classe), estendendo seus comportamentos e atributos.
Polimorfismo	• Princípio pelo qual as instâncias de duas classes ou mais classes derivadas de uma mesma super-classe podem invocar métodos com a mesma assinatura, mas com comportamentos distintos.
Encapsulamento	• Proibição do acesso direto ao estado de um objeto, disponibilizando apenas métodos que alterem esses estados na interface pública.

PROGRAMAÇÃO ORIENTADA A OBJETO

OS TRÊS PILARES DA PROGRAMAÇÃO ORIENTADA A OBJETOS:

1- Encapsulamento
2- Herança
3- Polimorfismo

PROGRAMAÇÃO ORIENTADA A OBJETO

Encapsulamento: o primeiro pilar

Em vez de ver um programa como uma única entidade grande e monolítica, o encapsulamento permite que você o divida em várias partes menores e independentes. Cada parte possui implementação e realiza seu trabalho independentemente das outras partes. O encapsulamento mantém essa independência, ocultando os detalhes internos ou seja, a implementação de cada parte, através de uma interface externa.

NOVO TERMO Encapsulamento é a característica da OO de ocultar partes independentes da implementação. O encapsulamento permite que você construa partes ocultas da implementação do software, que atinjam uma funcionalidade e ocultam os detalhes de implementação do mundo exterior.

Encapsulamento

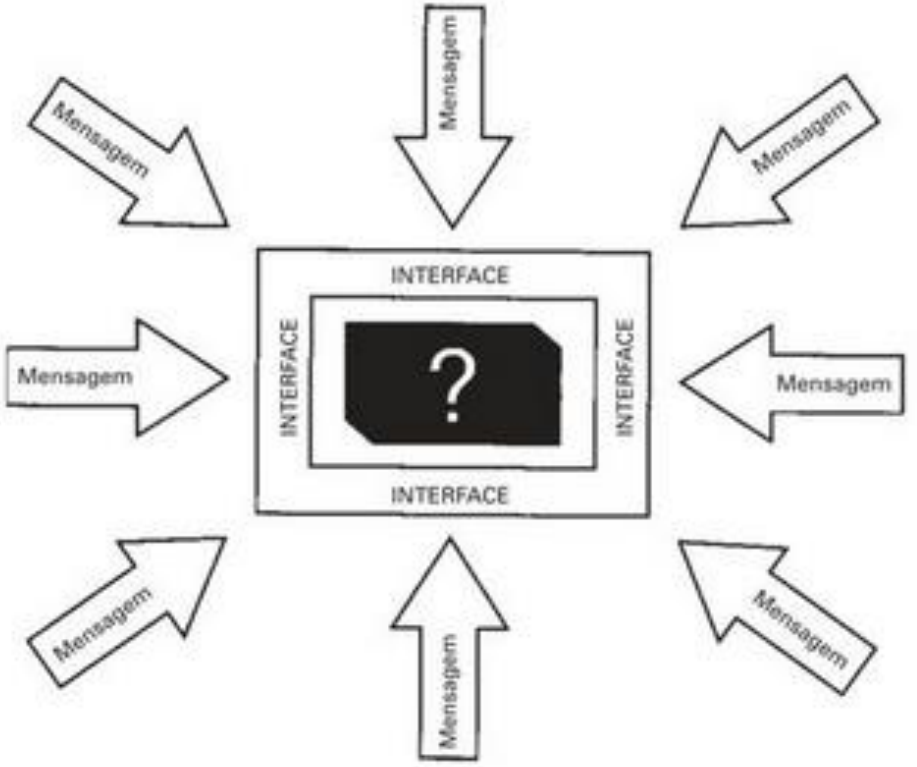
- Proibição do acesso direto ao estado de um objeto, disponibilizando apenas métodos que alterem esses estados na interface pública.

NOTA

Se você não estiver familiarizado com o termo *encapsulamento*, talvez reconheça os termos *módulo*, *componente* ou *bean*. Você pode usar esses termos em vez de 'software encapsulado'.

Uma vez encapsulado, você pode ver uma entidade de software como uma caixa preta. Você sabe o que a caixa preta faz, pois conhece sua interface externa. Conforme a Figura 2.1 ilustra, você simplesmente envia mensagens para a caixa preta. Você não se preocupa com o que acontece dentro da caixa; você só se preocupa com o fato de que isso aconteça.

FIGURA 2.1
Uma caixa preta.



PROGRAMAÇÃO ORIENTADA A OBJETO

NOVO TERMO Uma *interface* lista os serviços fornecidos por um componente. A interface é um contrato com o mundo exterior, que define exatamente o que uma entidade externa pode fazer com o objeto. Uma interface é o painel de controle do objeto.

NOTA

Uma interface é importante, pois ela diz o que você pode fazer com o componente. O mais interesse é o que uma interface *não* informa: como o componente fará seu trabalho. Em vez disso, a interface oculta a implementação do mundo exterior. Isso libera o componente para alterações na sua implementação a qualquer momento. As mudanças na implementação não mudam o código que usa a classe, desde que a interface permaneça inalterada. As alterações na interface necessitarão de mudanças no código que exerce essa interface.

NOTA

Talvez você esteja familiarizado com o termo de programação API (Interface de Programa Aplicativo). Uma *interface* é semelhante a API para um objeto. A interface lista todos os métodos e argumentos que o objeto entende.

NOVO TERMO A implementação define como um componente realmente fornece um serviço. A implementação define os detalhes internos do componente.

PROGRAMAÇÃO ORIENTADA A OBJETO

Público, privado e protegido

O que aparece e o que não aparece na interface pública é governado por diversas palavras chave. Cada linguagem OO define o seu próprio conjunto de palavras-chave, mas fundamentalmente essas palavras-chave acabam tendo efeitos semelhantes:

A maioria das linguagens OO suporta três níveis de acesso:

- Público — Garante o acesso a todos os objetos.
- Protegido — Garante o acesso à instância, ou seja, para aquele objeto, e para todas as subclasses (mais informações sobre subclasses no Dia 4, "Herança: obtendo algo para nada").
- Privado — Garante o acesso apenas para a instância, ou seja, para aquele objeto.

O nível de acesso que você escolhe é muito importante para seu projeto. Todo comportamento que você queira tornar visível para o mundo, precisa ter acesso público. Tudo que você quiser ocultar do mundo exterior precisa ter acesso protegido ou privado.

PROGRAMAÇÃO ORIENTADA A OBJETO

Por que devemos encapsular?

Quando usado cuidadosamente, o encapsulamento transforma seus objetos em componentes plugáveis. Para que outro objeto use seu componente, ele só precisa saber como usar a interface pública do componente. Tal independência têm três vantagens importantes:

Independência significa que você poderá reutilizar o objeto em qualquer parte. Quando vocÊ encapsula corretamente seus objetos, eles não estarão vinculados a nenhum programa em particular. Em vez disso, você poderá usá-los sempre que seu uso fizer sentido. Para usar o objeto em qualquer lugar, você simplesmente exerce sua interface.

O encapsulamento permite que você torne transparente as alterações em seu objeto.

Usar um objeto encapsulado não causará efeitos colaterais inesperados entre o objeto e o restante do programa.

PROGRAMAÇÃO ORIENTADA A OBJETO

Três características do encapsulamento eficaz:

- Abstração
- Ocultação da implementação
- Divisão de responsabilidades

PROGRAMAÇÃO ORIENTADA A OBJETO

Três características do encapsulamento eficaz:

- Abstração → É o processo de simplificar um problema difícil.

Considere dois exemplos.

Primeiro, imagine pessoas em fila em um banco, esperando por um caixa. Assim que um caixa se torna disponível, a primeira pessoa da fila avança para a janela aberta. As pessoas sempre deixam a fila na ordem de que o primeiro a entrar é o primeiro a sair (FIFO): essa ordem é sempre mantida.

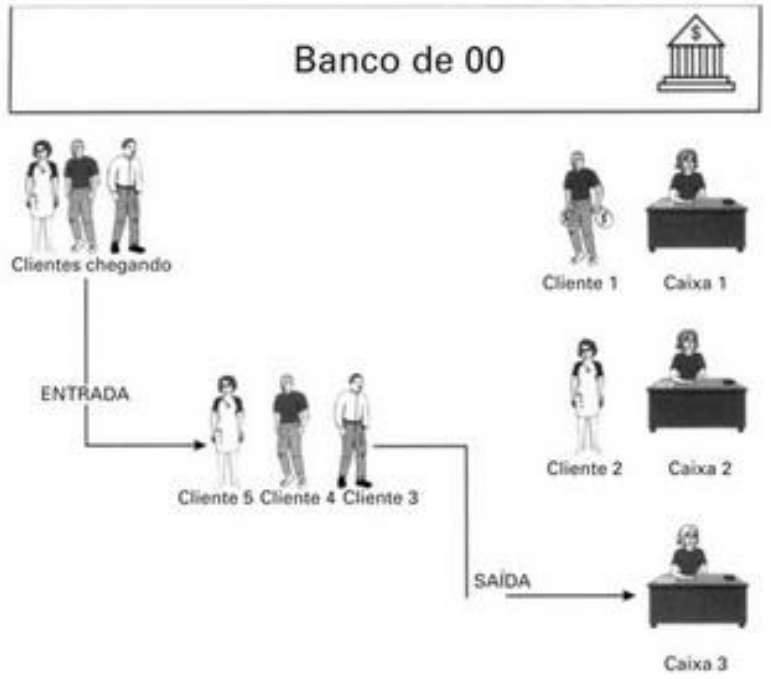


FIGURA 2.2
Uma fila em um banco.

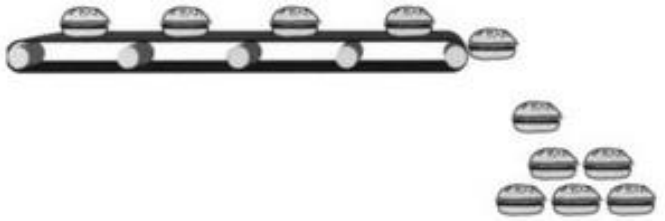
PROGRAMAÇÃO ORIENTADA A OBJETO

Três características do encapsulamento eficaz:

- Abstração → É o processo de simplificar um problema difícil.

Segundo, considere um estabelecimento de sanduíches do tipo *fast food*. Quando um novo sanduíche fica pronto, ele é colocado atrás do último sanduíche que está no escaninho; veja a Figura 2.3. Desse modo, o primeiro sanduíche retirado também é o mais antigo. FIFO é o esquema do restaurante.

FIGURA 2.3
Sanduíches ficando prontos.



Embora cada um desses exemplos seja específico, você pode encontrar uma descrição genérica que funcione em cada situação. Em outras palavras, você pode chegar a uma abstração.

Cada domínio é um exemplo de fila do tipo primeiro a entrar, primeiro a sair. Não importa quais tipos de elementos apareçam na fila. O que importa é que os elementos entram no final da fila e saem dela a partir da frente, conforme ilustrado na Figura 2.4.

FIGURA 2.4
Uma abstração de ambos os domínios.

Abstraindo os domínios, você pode criar uma fila uma vez e reutilizá-la em qualquer problema que modele um domínio onde exista uma ordenação FIFO de elementos.

PROGRAMAÇÃO ORIENTADA A OBJETO

Abstração eficaz

Neste ponto, você pode formular algumas regras para a abstração eficaz:

- Trate do caso geral e não do caso específico.
- Ao confrontar vários problemas diferentes, procure o que for comum a todos. Tente ver um conceito e não um caso específico.
- Não se esqueça de que você tem um problema a resolver. A abstração é valiosa, mas não descuide do problema na esperança de escrever código abstrato.
- A abstração pode não estar prontamente aparente. A abstração pode não saltar à sua frente na primeira, segunda ou terceira vez que você resolver um problema que está sujeito a ser abstraído.

- Prepare-se para a falha. É quase impossível escrever uma abstração que funcione em todas as situações. Você verá por que, posteriormente ainda hoje.

PROGRAMAÇÃO ORIENTADA A OBJETO

Não caia na paralisia da abstração. Resolva os problemas que você encontrar primeiro. Veja a abstração como um bônus e não como o objetivo final. Caso contrário, você vai se deparar com a possibilidade de prazos finais perdidos e abstração incorreta. Existem ocasiões para abstrair e ocasiões em que a abstração não é apropriada.

Uma boa regra geral é abstrair algo que você tiver implementado três vezes de maneira análoga. À medida que você ganhar experiência, aprenderá a escolher a abstração mais rapidamente.

Nem sempre você pode reconhecer oportunidades para uma abstração. Talvez você tenha de resolver um problema várias vezes, antes que uma abstração se torne aparente. Às vezes, diferentes situações ajudam a encontrar uma abstração eficaz e, mesmo então, a abstração pode precisar de alguma conversão. A abstração pode demorar a amadurecer.

A abstração pode tornar um componente encapsulado mais reutilizável, pois ele está personalizado para um domínio de problemas e não para um uso específico. Entretanto, há mais coisas importantes quanto ao encapsulamento do que a simples reutilização de componentes. O encapsulamento também é importante por ocultar os detalhes internos. O tipo abstrato de dados é um bom lugar para ver em seguida, na busca do encapsulamento eficaz.

PROGRAMAÇÃO ORIENTADA A OBJETO

Guardando seus segredos através da ocultação da implementação

A abstração é apenas uma característica do encapsulamento eficaz. Você pode escrever código abstrato que não é encapsulado. Em vez disso, você também precisa ocultar as implementações internas de seus objetos.

A ocultação da implementação tem duas vantagens:

- Ela protege seu objeto de seus usuários.
- Ela protege os usuários de seu objeto do próprio objeto.

Vamos explorar a primeira vantagem — proteção do objeto.

PROGRAMAÇÃO ORIENTADA A OBJETO

Protegendo seu objeto através do TAD (Abstract Data Type – Tipo Abstrato de Dados)

O Tipo Abstrato de Dados (TAD) não é um conceito novo. Os TADs, junto com a própria OO, cresceu a partir da linguagem de programação Simula, introduzida em 1966. Na verdade, os TADs são decididamente não OO; em vez disso, eles são um subconjunto da OO. Entretanto, os TADs apresentam duas características interessantes: *abstração* e *tipo*. É essa idéia de tipo que é importante, pois sem ela, você não pode ter um verdadeiro encapsulamento.

> O *verdadeiro encapsulamento* é imposto em nível de linguagem, através de construções internas da linguagem. Qualquer outra forma de encapsulamento é simplesmente um acordo de cavalheiros, que é facilmente malogrado. Os programadores o contornarão porque podem fazer isso!

NOVO TERMO Um *TAD* é um conjunto de dados e um conjunto de operações sobre esses dados. Os TADs permitem que você defina novos tipos na linguagem, ocultando dados internos e o estado, atrás de uma interface bem definida. Essa interface apresenta o TAD como uma única unidade atômica.

Os TADs são uma maneira excelente de introduzir encapsulamento, pois eles o liberam de considerar o encapsulamento sem a bagagem extra da herança e do polimorfismo: você pode se concentrar no encapsulamento. Os TADs também permitem que você explore a noção de tipo. Uma vez que o tipo seja entendido, é fácil ver que a OO oferece uma maneira natural de estender uma linguagem, definindo tipos personalizados do usuário.

PROGRAMAÇÃO ORIENTADA A OBJETO

O que é um tipo?

Quando programar, você criará diversas variáveis e atribuirá valores para elas. Os tipos definem as diferentes espécies de valores que estão disponíveis para seus programas. Você usa tipos para construir seu programa. Exemplos de alguns tipos comuns são integers (inteiros), longs (inteiros longos) e floats (reais). Essas definições de tipo informam exatamente quais espécies de tipos estão disponíveis, o que os tipos fazem e o que você pode fazer com eles.

Usaremos a seguinte definição de tipo:

NOVO TERMO Os *tipos* definem as diferentes espécies de valores que você pode usar em seus programas. Um tipo define o domínio a partir do qual seus valores válidos podem ser extraídos. Para inteiros positivos, são os números sem partes fracionárias e que são maiores ou iguais a 0. Para tipos estruturados, a definição é mais complexa. Além do domínio, a definição de tipo inclui quais operações são válidas no tipo e quais são seus resultados.

NOTA O tratamento formal de *tipo* está fora dos objetivos de um livro sobre POO para iniciantes.

Os tipos são unidades atômicas da computação. Isso significa que um tipo é uma unidade independente. Pegue o inteiro, por exemplo. Quando soma dois inteiros, você não pensa sobre a adição de bits individuais; você pensa apenas a respeito da adição de dois números. Mesmo que os bits representem o inteiro, a linguagem de programação apresenta o inteiro apenas como um número para o programador.

PROGRAMAÇÃO ORIENTADA A OBJETO

Os TADs são ferramentas de encapsulamento valiosas, pois eles permitem que você defina novos tipos da linguagem que são seguros de usar. Assim como novas palavras são acrescentadas no idioma inglês a cada ano, um TAD permite que você crie novas palavras de programação, onde você precisa expressar uma nova idéia.

Uma vez definido, você pode usar um novo tipo como qualquer outro. Assim como você pode passar um inteiro para um método, também pode passar um TAD para um método. Isso é conhecido como sendo um objeto de primeira classe. Você pode passar objetos de primeira classe como parâmetros.

NOVO TERMO Um *objeto de primeira classe* é aquele que pode ser usado exatamente da mesma maneira que um tipo interno.

NOVO TERMO Um *objeto de segunda classe* é um tipo de objeto que você pode definir, mas não necessariamente usar, como faria com um tipo interno.

PROGRAMAÇÃO ORIENTADA A OBJETO

A ocultação da implementação leva a um projeto mais flexível, pois ela impede que os usuários de seus objetos se tornem fortemente acoplados à implementação subjacente dos objetos. Então, não apenas a ocultação da implementação protege seus objetos, como também protege aqueles que usam seus objetos, estimulando um código fracamente acoplado.

NOVO TERMO O *código fracamente acoplado* é independente da implementação de outros componentes.

NOVO TERMO O *código fortemente acoplado* é fortemente vinculado à implementação de outros componentes.

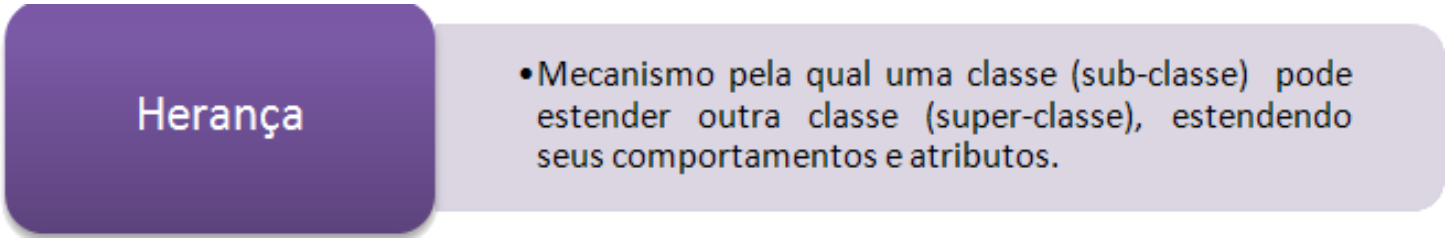
Você poderia estar se perguntando, "para que serve código fracamente acoplado?"

Quando um recurso aparece na interface pública de um objeto, todo mundo que usa o recurso se torna dependente do fato de ele existir. Se o recurso desaparecer repentinamente, você precisará alterar o código que tiver se desenvolvido de forma dependente a esse comportamento ou atributo.

NOVO TERMO *Código dependente* é dependente da existência de determinado tipo. O código dependente é inevitável. Entretanto, existem graus para a dependência aceitável e para a superdependência.

PROGRAMAÇÃO ORIENTADA A OBJETO

HERANÇA



A herança permite À classe que está herdando redefinir qualquer comportamento do que não goste. Tal recurso permite que você adapte seu software, quando seus requisitos mudarem.

PROGRAMAÇÃO ORIENTADA A OBJETO

"É um" *versus* "tem um": aprendendo quando usar herança

Para apresentar os mecanismos de herança, a primeira seção abordou o que é conhecido como *herança de implementação*. Conforme você viu, a herança de implementação permite que suas classes herdem a implementação de outras classes. Entretanto, somente porque uma classe pode herdar de outra não significa que isso deve ser feito!

Então, como você sabe quando deve usar herança? Felizmente, existe uma regra geral a ser seguida, para evitar uma herança incorreta.

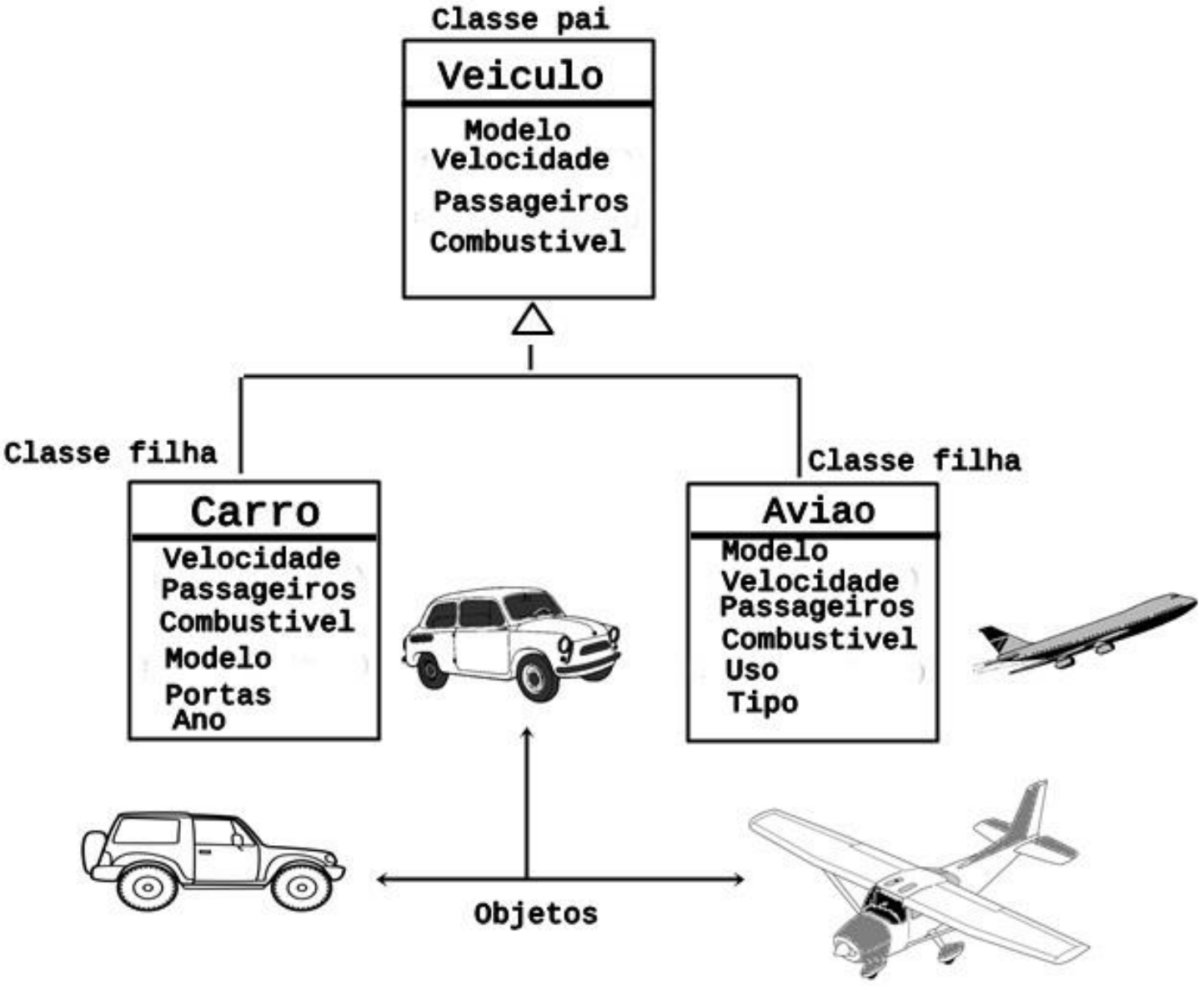
Quando você está considerando a herança para reutilização ou por qualquer outro motivo, precisa primeiro perguntar-se se a classe que está herdando é do mesmo tipo que a classe que está sendo herdada. O fato de pensar em termos de tipo enquanto se herda é freqüentemente referido como teste 'é um'.

NOVO TERMO *É um* descreve o relacionamento em que uma classe é considerada do mesmo tipo de outra.

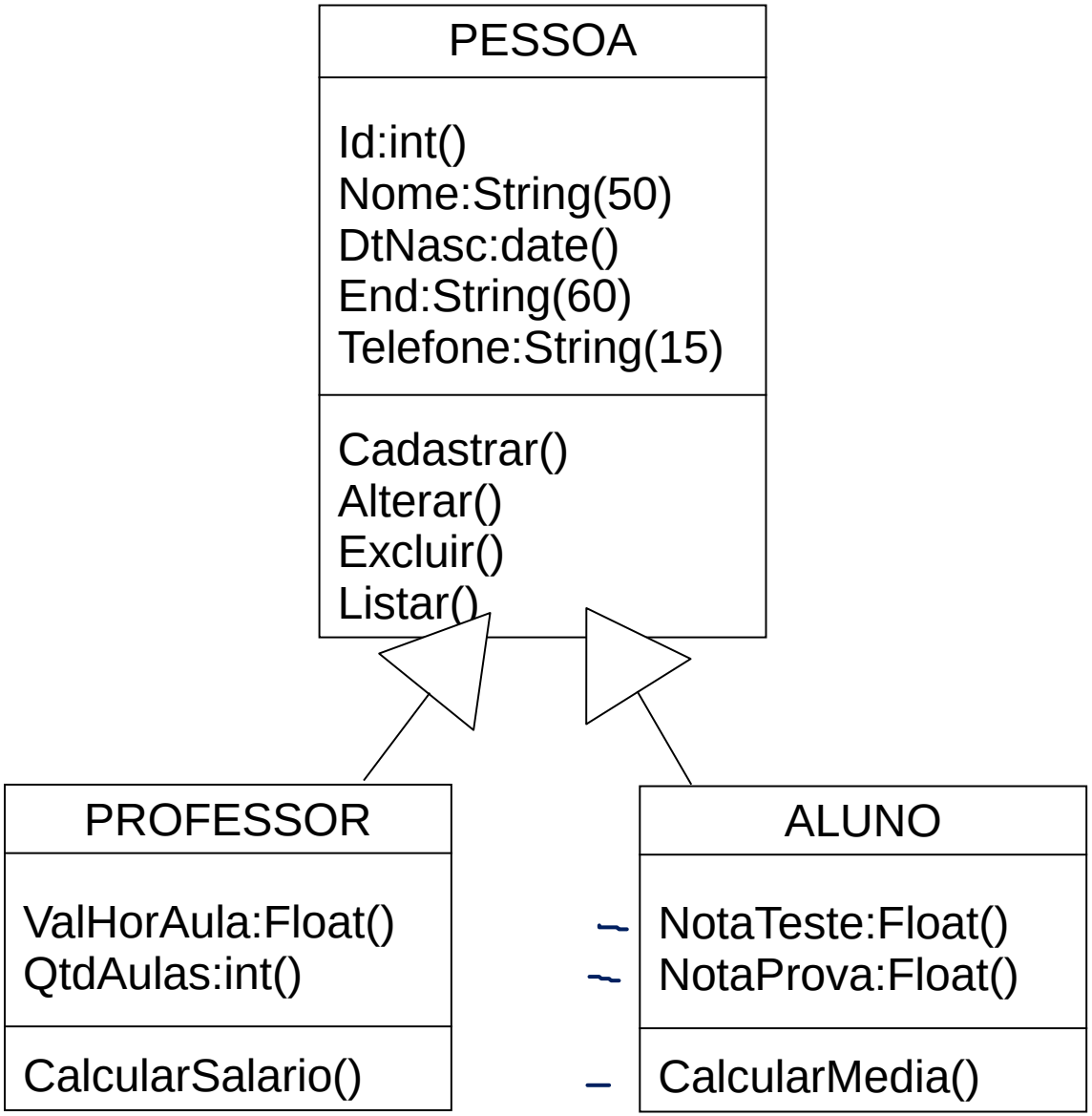
Para usar 'é um', você diz a si mesmo, "um objeto `CommissionedEmployee` 'é um' `Employee`". Essa declaração é verdadeira e você saberia imediatamente que a herança é válida nessa situação. Agora, pare e considere a interface `Iterator` Java:

```
public interface Iterator {
      public boolean hasNext();
      public Object next();
      public void remove();
}
```

PROGRAMAÇÃO ORIENTADA A OBJETO



PROGRAMAÇÃO ORIENTADA A OBJETO



Existirão muitas situações onde o teste 'é um' falhará, quando você quiser reutilizar alguma implementação. Felizmente, existem outras maneiras de reutilizar implementação. Você sempre pode usar composição e delegação (veja o quadro a seguir). O teste 'tem um' salva o dia.

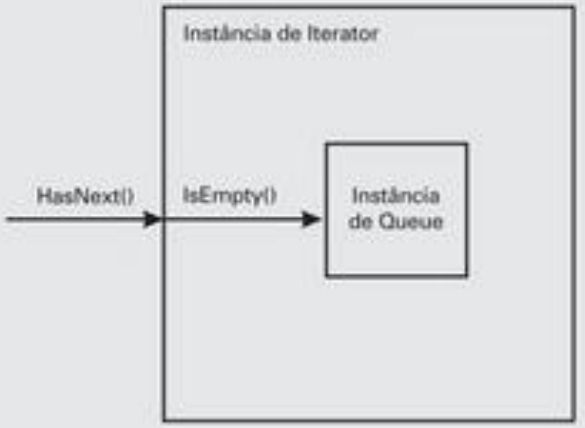
NOVO TERMO *Tem um* descreve o relacionamento em que uma classe contém uma instância de outra classe.

NOVO TERMO *Composição* significa que uma classe é implementada usando-se variáveis internas (chamadas de variáveis membro), que contêm instâncias de outras classes.

Composição é uma forma de reutilização que você já viu. Se você não puder herdar, nada o impede de usar instâncias da outra classe dentro da nova classe. Quando você quiser usar os recursos de outra classe, use simplesmente uma instância dessa classe como uma de suas partes constituintes. É claro que você sofre as limitações apresentadas anteriormente.

Considere novamente o exemplo `Queue/Iterator`. Em vez de herdar de `Queue`, a interface `Iterator` pode simplesmente criar uma instância de `Queue` e armazená-la em uma variável de instância. Quando a interface `Iterator` precisa recuperar um elemento ou verificar se está vazia, ela pode simplesmente delegar o trabalho para a instância de `Queue`, como demonstrado na Figura 4.2.

FIGURA 4.2
Uma interface `Iterator` *delegando chamadas de método para* `Queue`.



Quando usa composição, você escolhe cuidadosamente o que vai usar. Através da delegação, você pode expor alguns ou todos os recursos de seus objetos constituintes. A Figura 4.2 ilustra como a interface `Iterator` direciona o método `hasNext()` para o método `isEmpty()` de `Queue`.

É importante indicar que a delegação difere da herança de duas maneiras importantes:

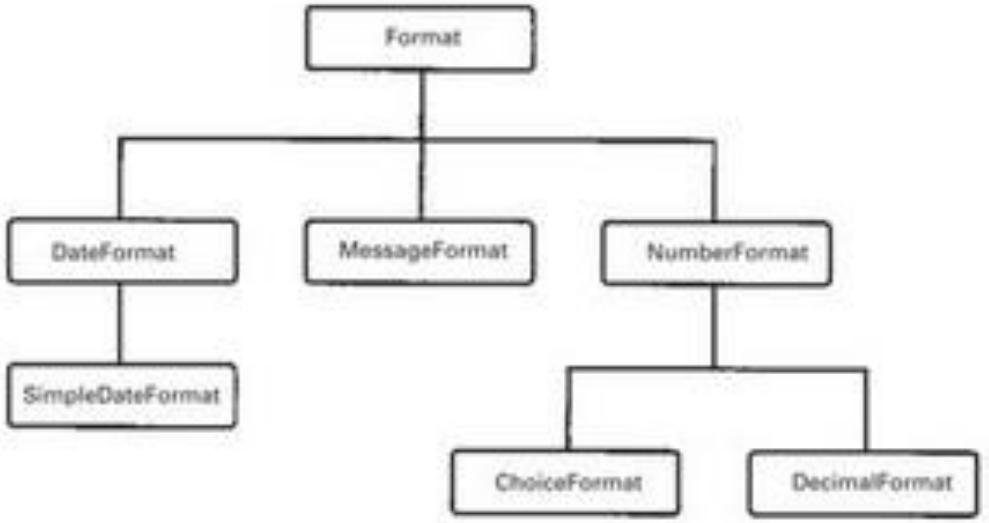
1. Com a herança, você tem apenas uma instância do objeto. Existe apenas um objeto indivisível, pois o que é herdado se torna uma parte intrínseca da nova classe.
2. A delegação geralmente fornece ao usuário apenas o que está na interface pública. A herança normal dá mais acesso aos detalhes internos da classe herdada. Vamos falar a respeito de tal acesso em detalhes, no final da lição de hoje.

Aprendendo a navegar na teia emaranhada da herança

Os conceitos de 'é um' e composição mudam a natureza da discussão sobre herança da ambiciosa reutilização da implementação para inter-relacionamentos de classe. Uma classe que herda de outra deve se relacionar com essa classe de alguma maneira, para que os relacionamentos ou hierarquias de herança resultantes façam sentido.

NOVO TERMO Uma *hierarquia de herança* é um mapeamento do tipo árvore de relacionamentos que se formam entre classes como resultado da herança. A Figura 4.3 ilustra uma hierarquia real extraída da linguagem Java.

FIGURA 4.3
Um exemplo de hierarquia de `java.text`.



A herança define a nova classe, a *filha*, em termos de uma classe antiga, a *progenitora* ou *mãe*. Esse relacionamento filha-progenitora ou filha-mãe é o relacionamento de herança mais simples. Na verdade, todas as hierarquias de herança começam com uma progenitora e uma filha.

NOVO TERMO A *classe filha* é a classe que está herdando; também conhecida como subclasse.

NOVO TERMO A *classe progenitora* ou *mãe* é a classe da qual a filha herda diretamente; ela também é conhecida como superclasse.

A Figura 4.4 ilustra um relacionamento progenitora/filha. `NumberFormat` é a progenitora das duas filhas `ChoiceFormat` e `DecimalFormat`.

Sobrepor um método também é conhecido como *redefinir* um método. Redefinindo um método, a filha fornece sua própria implementação personalizada do método. Essa nova implementação fornecerá um comportamento novo para o método. Aqui, `ThreeDimensionalPoint` redefine o comportamento do método `toString()`, para que ele seja corretamente transformado em um objeto String.

NOVO TERMO *Sobrepor* é o processo de uma filha pegar um método que aparece na progenitora e reescrevê-lo para mudar o comportamento do método. A sobreposição de um método também é conhecida como *redefinição* de um método.

PROGRAMAÇÃO ORIENTADA A OBJETO

Níveis de acesso

- Privado: um nível de acesso que restringe o acesso apenas à classe.
- Protegido: um nível de acesso que restringe o acesso à classe e às filhas.
- Público: um nível de acesso que permite o acesso a todos e a qualquer um.

Os métodos e atributos protegidos são aqueles aos quais você deseja que apenas as subclasses tenham acesso. Não deixe tais métodos públicos. Apenas aqueles com amplo conhecimento da classe devem usar métodos e atributos protegidos.

PROGRAMAÇÃO ORIENTADA A OBJETO

Tipos de Herança:

1 - Para reutilização de implementação
2 - Para diferença
3 - Para substituição de tipo

A Herança para implementação

Em vez de recortar e colar código ou instanciar e usar um componente através de composição, a herança torna o código automaticamente disponível, como parte da nova classe.

A Herança para diferença

A programação pela diferença permite que você programe especificando apenas como uma classe filha difere de sua classe progenitora.
Significa herdar uma classe e adicionar apenas o código que torne a nova classe diferente da classe herdada. Possibilidade de programar através de incrementos.

Para substituição de tipo

Definido como especialização.

PROGRAMAÇÃO ORIENTADA A OBJETO

ESPECIALIZAÇÃO

Especialização é o processo de uma classe filha ser projetada em termos de como ela é diferente de sua progenitora. Quando tudo estiver dito e feito, a definição de classe da filha incluirá apenas os elementos que a tornam diferente de sua progenitora.

Uma classe filha se especializa em relação à sua progenitora, adicionando novos atributos e métodos em sua interface, assim como redefinindo atributos e métodos previamente existentes. A adição de novos métodos ou a redefinição de métodos já existentes permite que a filha expresse comportamentos que são diferentes de sua progenitora.

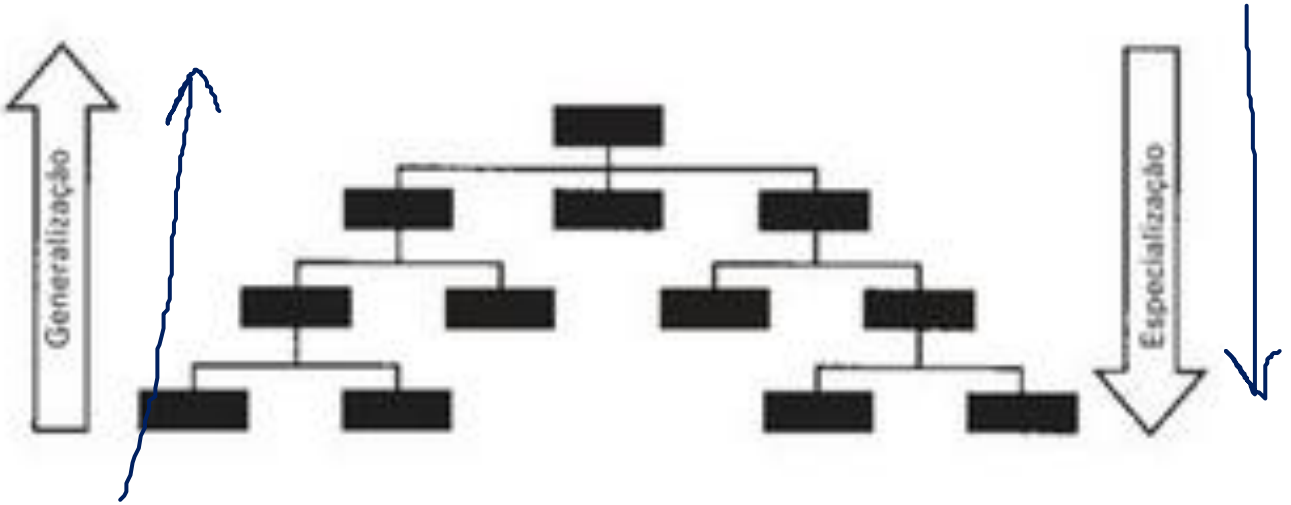
PROGRAMAÇÃO ORIENTADA A OBJETO

ESPECIALIZAÇÃO

Não se confunda com o termo especialização. A especialização permite apenas que você adicione ou redefina os comportamentos e atributos que a filha herda de sua progenitora. A especialização, ao contrário do que o nome possa sugerir, não permite que você remova da filha comportamentos e atributos herdados**. Uma classe não obtém herança seletiva.**

Quando você percorre uma hierarquia para baixo você especializa. Quando você percorre para cima, você generaliza.

FIGURA 4.7
Quando percorre uma hierarquia para cima, você generaliza. Quando percorre uma hierarquia para baixo, você especializa.



NOVO TERMO Dada alguma filha, uma *ancestral* é uma classe que aparece na hierarquia de classes antes da progenitora. Conforme a Figura 4.8 ilustra, `Format` é uma ancestral de `DecimalFormat`.

NOVO TERMO Dada uma classe, toda classe que aparece depois dela na hierarquia de classes é uma *descendente* da classe dada. Conforme a Figura 4.8 ilustra, `DecimalFormat` é uma descendente de `Format`.

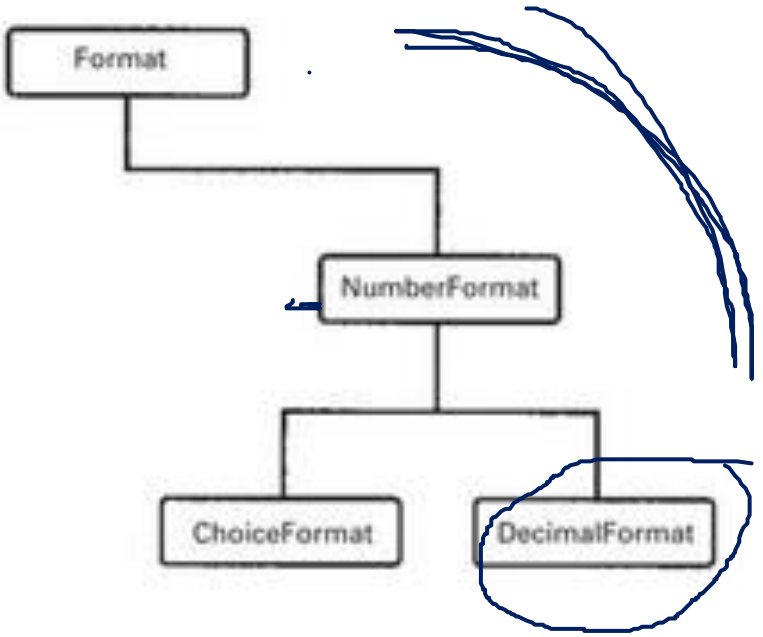


FIGURA 4.8
`DecimalFormat` é descendente de `Format`.

Digamos que tivéssemos a hierarquia de herança de classes mostrada na Figura 4.9. Dizemos que `OneDimensionalPoint` é a progenitora de `TwoDimensionalPoint` e ancestral de `ThreeDimensionalPoint` e de `FourDimensionalPoint`. Também podemos dizer que `TwoDimensionalPoint`, `ThreeDimensionalPoint` e `FourDimensionalPoint` são todas descendentes de `OneDimensionalPoint`. Todos os descendentes compartilham os métodos e atributos de seus ancestrais.

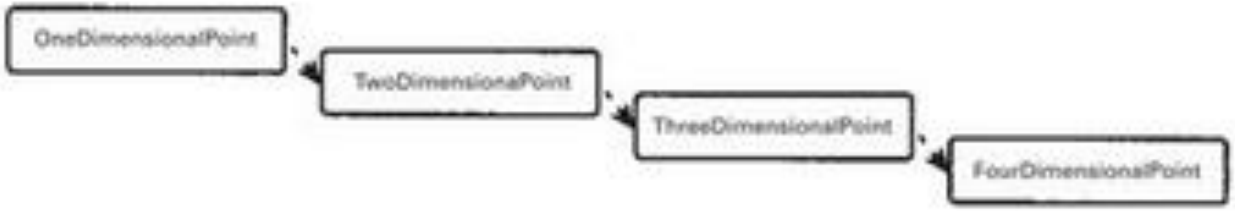


FIGURA 4.9
A hierarquia de ponto.

Podemos fazer mais algumas declarações interessantes sobre a hierarquia de classes. `OneDimensionalPoint` é a raiz e `FourDimensionalPoint` é uma folha.

NOVO TERMO A *classe raiz* (também referida comumente como classe base) é a classe superior da hierarquia de herança. A Figura 4.9 mostra que `OneDimensionalPoint` é uma classe raiz.

NOVO TERMO Uma *classe folha* é uma classe sem filhas. Na Figura 4.8, `DecimalFormat` é uma classe folha.

É importante notar que as descendentes refletirão as alterações feitas nas ancestrais. Digamos que você encontre um erro em `TwoDimensionalPoint`. Se você corrigir `TwoDimensionalPoint`, todas as classes de `ThreeDimensionalPoint` até `FourDimensionalPoint` tirarão proveito da alteração. Assim, se você corrigir um erro ou tornar uma implementação mais eficiente, todas as classes descendentes da hierarquia tirarão proveito disso.

Herança múltipla

Em todos os exemplos você viu herança simples. Algumas implementações de herança permitem que um objeto herde diretamente de mais de uma classe. Tal implementação de herança é conhecida como *herança múltipla*. A herança múltipla é um aspecto controverso da POO. Alguns dizem que ela só torna o software mais difícil de entender, projetar e manter. Outros têm grande confiança nela e dizem que uma linguagem não está completa sem ela.

De qualquer modo, a herança múltipla pode ser valiosa, se usada cuidadosa e corretamente. Existem vários problemas introduzidos pela herança múltipla. Entretanto, uma discussão completa do que a herança múltipla pode e não pode fazer está fora dos objetivos deste dia.

Resumo Herança

SuperClasse Classe Pai Classe Base Generalização	[e]	SubClasse Classe Filho Classe Derivada Especialização

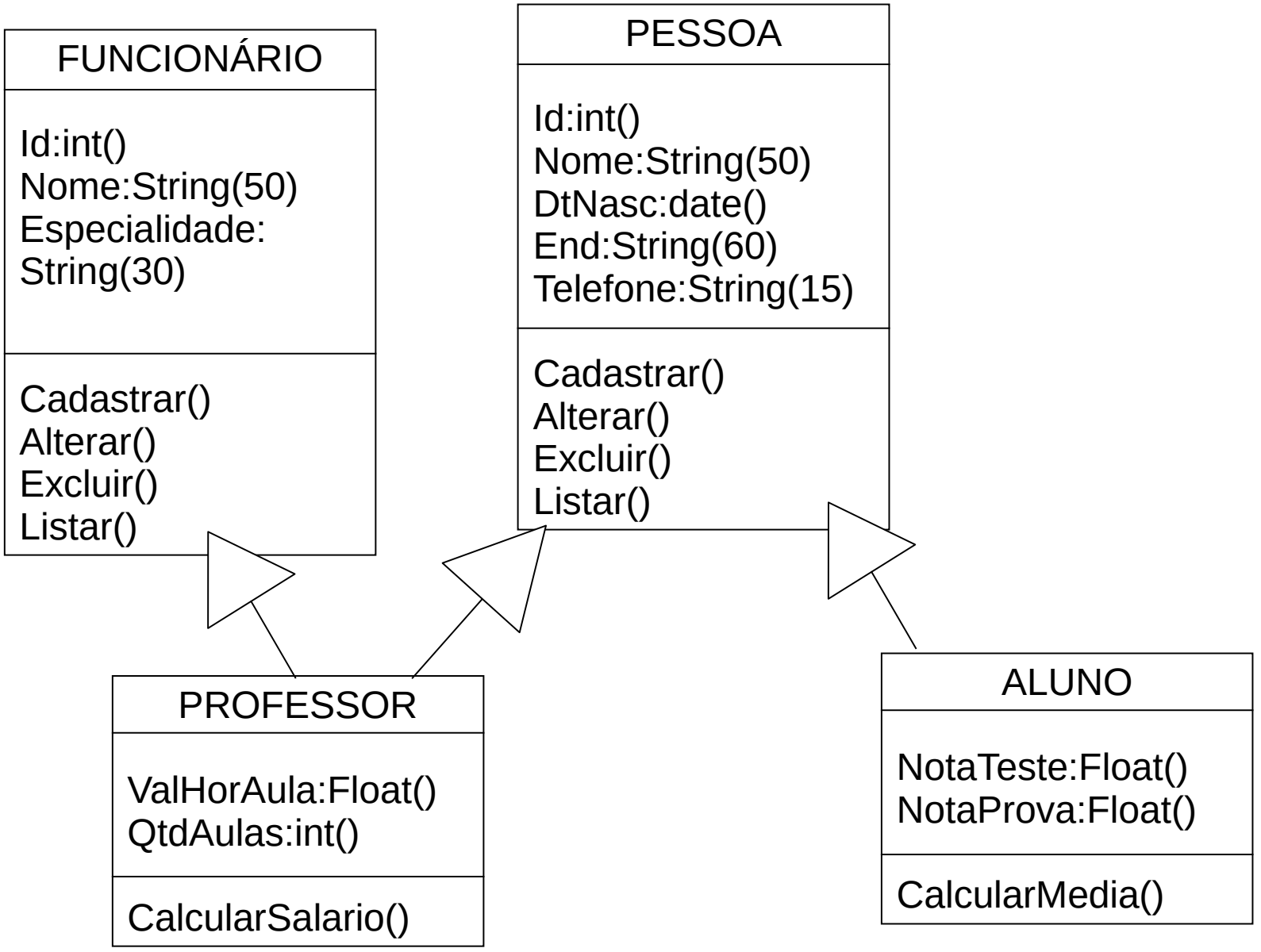
Reutilização de Código

- **Classes abstratas**

É um tipo de classe especial que não pode ser instanciada, apenas herdada. Sendo assim, uma classe abstrata não pode ter um objeto criado a partir de sua instanciação. Essas classes são muito importantes quando não queremos criar um objeto a partir de uma classe "geral", apenas de suas "subclasses".

Uma classe abstrata é tipicamente usada como uma classe base e não pode ser instanciada. Ela pode conter métodos abstratos e não-abstratos, e pode ser uma subclasse de uma classe abstrata ou uma não-abstrata. Todos os métodos abstratos precisam ser defindos pela classe que herda (extende) ao menos que a subclasse seja também abstrata.

PROGRAMAÇÃO ORIENTADA A OBJETO



PROGRAMAÇÃO ORIENTADA A OBJETO

Na prática

Representação de uma Classe

PESSOA
+ Id:int() + Nome:String(50) + End:String(50)
+CadastrarPessoa() +ListarPessoa() +ExcluirPessoa()

No código:

JAVA:

```
public class Pessoa {
 public String Id;
 public String nome;
 public String end;

}
```

Na prática

PROGRAMAÇÃO ORIENTADA A OBJETO

ENCAPSULAMENTO

No código: O uso do private Determina o conceito de Encapsulamento

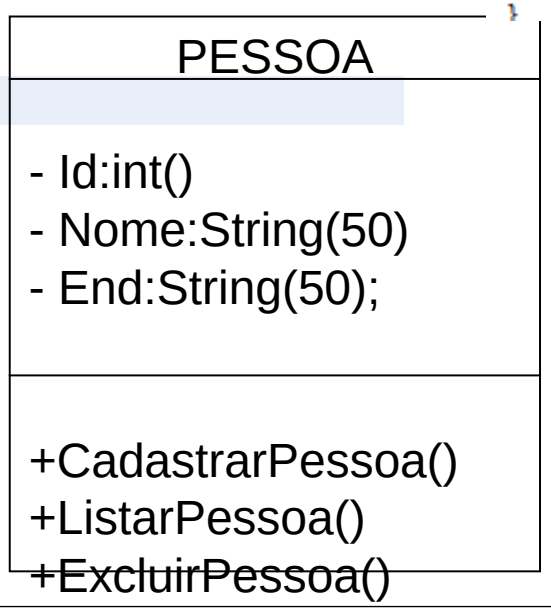
JAVA:

Pessoa.Java

```
public class Pessoa {
 private String Id;
 private String nome;
 private String end;

}
```

TesteAppJava.iava

```
public class TesteAppJava {

    public static void main(String [] args){

        Pessoa p=new Pessoa();

    }
}
```



PROGRAMAÇÃO ORIENTADA A OBJETO

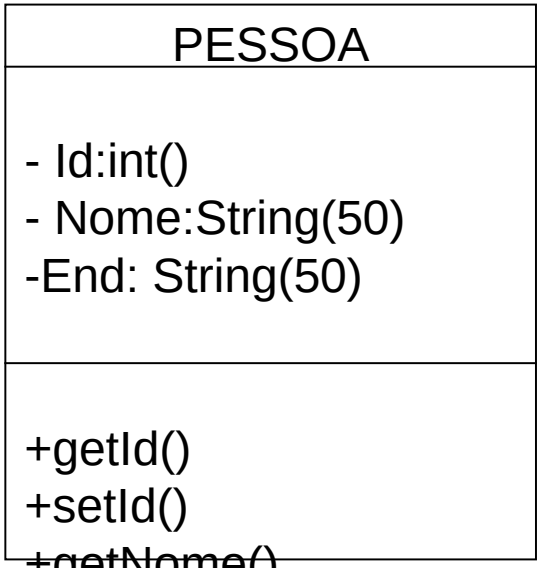
Na prática

Pessoa.Java

```
public class Pessoa {
 private String Id;
 private String nome;
 private String end;
    public String getId() {
        return Id;
    }

    public void setId(String Id) {
        this.Id = Id;
    }

    public String getNome() {
        return nome;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }

    public String getEnd() {
        return end;
    }

    public void setEnd(String end) {
        this.end = end;
    }
```

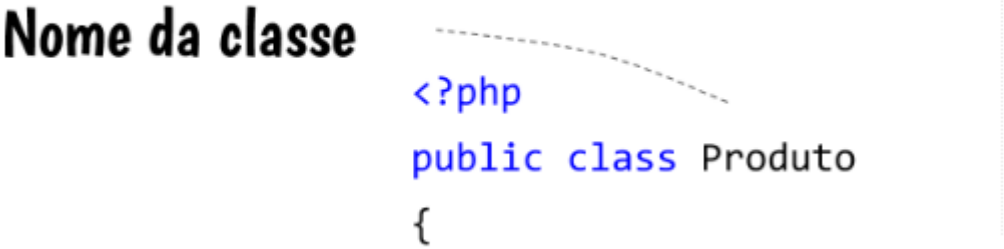
TesteAppJava.java

```
public class TesteAppJava {

    public static void main(String [] args){

        Pessoa p=new Pessoa();
        p.setNome("Ana");

        System.out.println("Seja bem vinda" + p.getNome());
    }
}
```



PROGRAMAÇÃO ORIENTADA A OBJETO

Uma classe em PHP



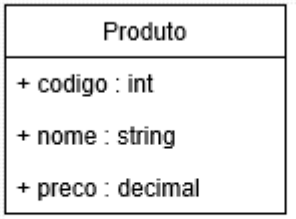


Figura 1. Diagrama da classe Produto

PROGRAMAÇÃO ORIENTADA A OBJETO

```php
<?php

class Produto {

    private $codigo;
    private $nome;
    private $preco;

    public function getCodigo()
    {
        return $this->codigo;
    }
    public function setCodigo($codigo)
    {
        $this->codigo = $codigo;
    }
    public function getNome()
    {
        return $this->nome;
    }
    public function setNome($nome)
    {
        $this->nome = $nome;
    }
    public function getPreco()
    {
        return $this->preco;
    }
    public function setPreco($preco)
    {
        $this->preco = $preco;
    }
}
```

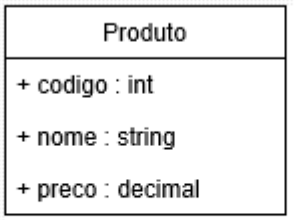


Figura 1. Diagrama da classe Produto

PROGRAMAÇÃO ORIENTADA A OBJETO

PESSOA
- Id:int() - Nome:String(50) - End:String(50)
CadastrarPessoa() ListarPessoa() ExcluirPessoa()

```php
<?php

class Pessoa {

    private $id;
    private $nome;
    private $end;

    public function getId()
    {
        return $this->Id;
    }
    public function setId($id)
    {
        $this->id= $id;
    }
    public function getNome()
    {
        return $this->nome;
    }
    public function setNome($nome)
    {
        $this->nome = $nome;
    }
    public function getEnd()
    {
        return $this->end;
    }
    public function setEnd($end)
    {
        $this->end = $end;
    }
}
```

PROGRAMAÇÃO ORIENTADA A OBJETO

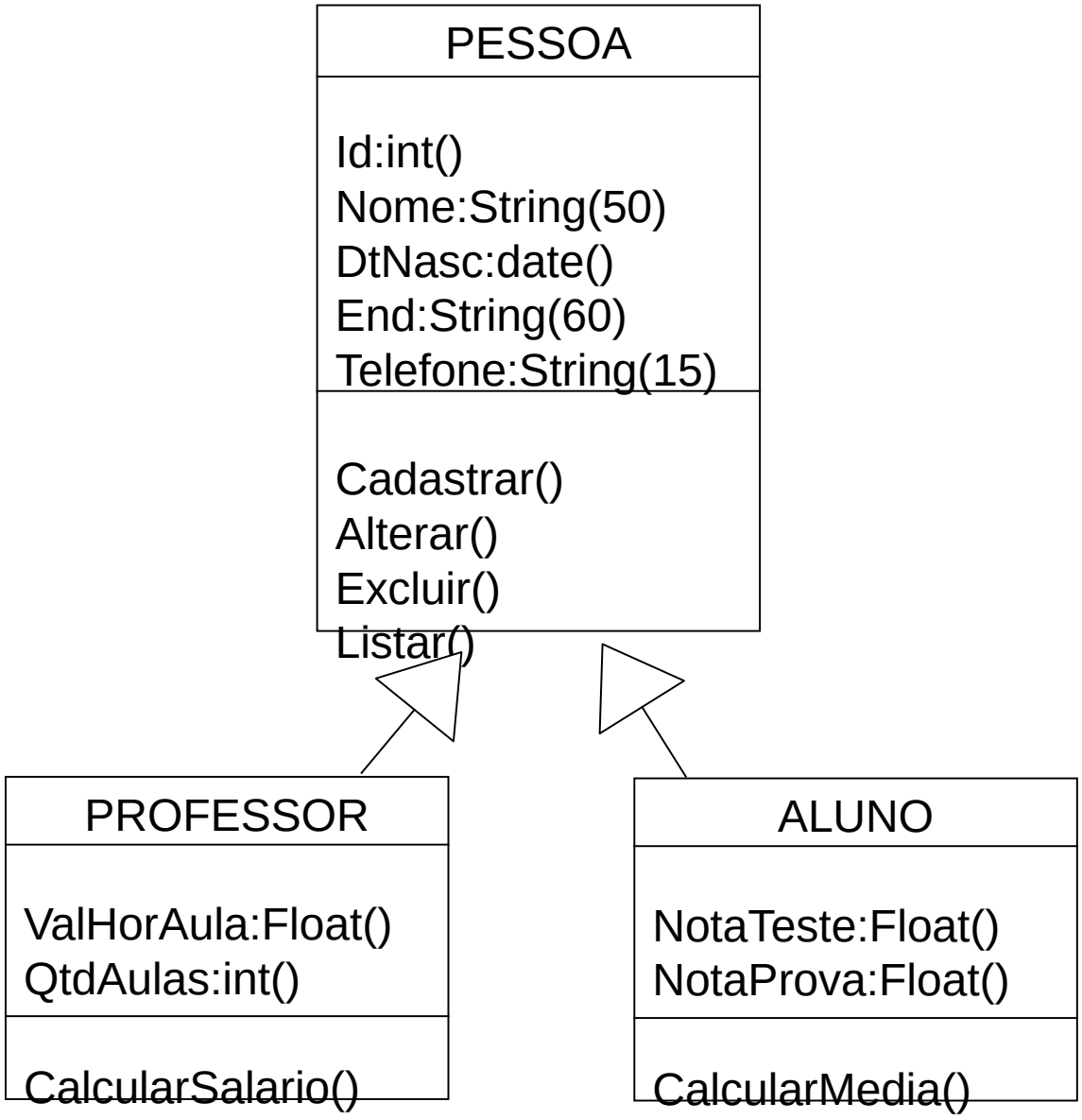
PESSOA
- Id:int() - Nome:String(50) - End: String(50)
CadastrarPessoa() ListarPessoa() ~~ExcluirPessoa()~~

```php
<?php

class Pessoa {

    private $id;
    private $nome;
    private $end;

    public function getId()
    {
        return $this->Id;
    }
    public function setId($id)
    {
        $this->id= $id;
    }
    public function getNome()
    {
        return $this->nome;
    }
    public function setNome($nome)
    {
        $this->nome = $nome;
    }
    public function getEnd()
    {
        return $this->end;
    }
    public function setEnd($end)
    {
        $this->end = $end;
    }
}
```

PROGRAMAÇÃO ORIENTADA A OBJETO

Na prática

HERANÇA



PROGRAMAÇÃO ORIENTADA A OBJETO

Código em JAVA:

Pessoa.Java

```java
public class Pessoa {
 private String Id;
 private String nome;
 private String end;
    public String getId() {
        return Id;
    }

    public void setId(String Id) {
        this.Id = Id;
    }

    public String getNome() {
        return nome;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }

    public String getEnd() {
        return end;
    }

    public void setEnd(String end) {
        this.end = end;
    }
```

Professor.Java

```java
public class Professor extends Pessoa{
    private double valorHoraAula;
    private int qtdAulas;


    public double getValorHoraAula() {
        return valorHoraAula;
    }

    public void setValorHora(double valorHoraAula) {
        this.valorHoraAula = valorHoraAula;
    }

    public int getQtdAula() {
        return qtdAulas;
    }

    public void setQtdAula(int qtdAulas) {
        this.qtdAulas = qtdAulas;
    }


}
```

PROGRAMAÇÃO ORIENTADA A OBJETO

Código em PHP:

Pessoa.php

```php
<?php

class Pessoa {

    private $id;
    private $nome;
    private $end;

    public function getId()
    {
        return $this->Id;
    }
    public function setId($id)
    {
        $this->id= $id;
    }
    public function getNome()
    {
        return $this->nome;
    }
    public function setNome($nome)
    {
        $this->nome = $nome;
    }
    public function getEnd()
    {
        return $this->end;
    }
    public function setEnd($end)
    {
        $this->end = $end;
    }
}
```

Professor.php

```php
<?php

  class Professor extends Pessoa {

      private $valorHoraAula;
      private $qtdAulas;

      public function getValorHoraAula()
      {
          return $this->valorHoraAula;
      }
      public function setValorHoraAula($valorHoraAula)
      {
          $this->valorHoraAula = $valorHoraAula;
      }
     public function getQtdAulas()
      {
          return $this->qtdAulas;
      }
      public function setQtdAulas($valorqtdAulas)
      {
          $this->qtdAulas = $qtdAulas;
      }
```

PROGRAMAÇÃO ORIENTADA A OBJETO

POLIMORFISMO

Polimorfismo

- Princípio pelo qual as instâncias de duas classes ou mais classes derivadas de uma mesma super-classe podem invocar métodos com a mesma assinatura, mas com comportamentos distintos.

Se o encapsulamento e a herança são os socos um e dois da POO, o polimorfismo é o soco para nocaute seguinte. Sem os dois pilares, você não poderia ter o polimorfismo, e sem o polimorfismo, a POO não seria eficaz. O polimorfismo é onde o paradigma da programação orientada a objetos realmente brilha e seu domínio é absolutamente necessário para a POO eficaz.

Polimorfismo significa muitas formas. Em termos de programação, o polimorfismo permite que um único nome de classe ou nome de método represente um código diferente, selecionado por algum mecanismo automático. Assim, um nome pode assumir muitas formas e como pode representar código diferente o mesmo nome pode representar muitos comportamentos diferentes.

PROGRAMAÇÃO ORIENTADA A OBJETO

Conceitos sobre Polimorfismo:

1- De inclusão
2- Paramétrico
3- Sobreposição *(Sobrescrita , overwrite)
4- Sobrecarga *(Overhead)

O Polimorfismo de inclusão, às vezes chamado de polimorfismo puro, permite que você trate objetos relacionados genericamente.

Também associado a Sobreposição ou sobrescrita

O Polimorfismo paramétrico permite que você crie métodos e tipos genéricos. Assim como o polimorfismo de inclusão, os métodos e tipos genéricos permitem que você codifique algo uma vez e faça isso trabalhar com muitos tipos diferentes de argumentos.
Também associado a sobrecarga.

PROGRAMAÇÃO ORIENTADA A OBJETO

Na prática: **SOBRESCRITA**

JAVA:

```java
public class Professor extends Pessoa{
    private double valorHoraAula;
    private int qtdAulas;

    public String getNome() {
        System.out.println("Olá professor: "+nome);
        return nome;
    }

    public double getValorHora() {
        return valorHoraAula;
    }

    public void setValorHora(double valorHoraAula) {
        this.valorHoraAula = valorHoraAula;
    }

    public int getQtdAula() {
        return qtdAulas;
    }

    public void setQtdAula(int qtdAulas) {
        this.qtdAulas = qtdAulas;
    }
```

```java
public class Aluno extends Pessoa{
    private double nota1;
        private double nota2;


        public String getNome() {
        System.out.println("Olá aluno: "+nome);
        return nome;
    }
    public double getNota1() {
        return nota1;
    }

    public void setNota1(double nota1) {
        this.nota1 = nota1;
    }

    public double getNota2() {
        return nota2;
    }

    public void setNota2(double nota2) {
        this.nota2 = nota2;
    }
```

PROGRAMAÇÃO ORIENTADA A OBJETO

Na prática

Pessoa.Java

```
public class Pessoa {
 private String Id;
 private String nome;
 private String end;
    public String getId() {
        return Id;
    }

    public void setId(String Id) {
        this.Id = Id;
    }

    public String getNome() {
        return nome;
    }

    public void setNome(String nome) {
        this.nome = nome;
    }

    public String getEnd() {
        return end;
    }

    public void setEnd(String end) {
        this.end = end;
    }
```

TesteAppJava.java

```
public class TesteAppJava {

    public static void main(String [] args){

        Pessoa p=new Pessoa();
        p.setNome("Ana");

        System.out.println("Seja bem vinda" + p.getNome());
    }
}
```

PROGRAMAÇÃO ORIENTADA A OBJETO

Na prática: **SOBRECARGA**

JAVA:

```
public class Aluno extends Pessoa{
    private double nota1;
        private double nota2;


        public String getNome() {...4 linhas }
    public double getNota1() {...3 linhas }

    public void setNota1(double nota1) {...3 linhas }

    public double getNota2() {...3 linhas }

    public void setNota2(double nota2) {...3 linhas }


    public void imprimir(){
        System.out.println("Nome: "+ nome);
        System.out.println("Endereço: "+ end);
        System.out.println("Idade: "+ idade);
        System.out.println("Nota1: "+ nota1);
        System.out.println("Nota2: "+ nota2);

    }

    public void imprimir(double nota1, double nota2){
        double md= (nota1+nota2)/2;
        System.out.println("A média é:"+ md);
    }
```

PROGRAMAÇÃO ORIENTADA A OBJETO

Na prática: **SOBRECARGA**

JAVA:

```
10
11      public static void main(String[] args) {
12
13
14          Aluno a = new Aluno();
15
16
17          a.setNome("Matheus");
18          a.setEnd("Rua suvaco da minhoca");
19          a.setIdade(19);
20          a.setNota1(7.9);
21          a.setNota2(10.0);
22
23          a.imprimir();
24
25
26      }
27
```

Sem nenhum parâmetro!!!

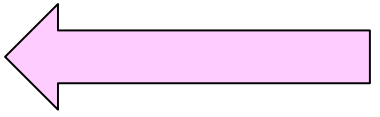
```
Saída -
Nome: Matheus
Endereço: Rua suvaco da minhoca
Idade: 19
Nota1: 7.9
Nota2: 10.0
CONSTRUÍDO COM SUCESSO (tempo total: 0 segundos)
```

PROGRAMAÇÃO ORIENTADA A OBJETO

Na prática: **SOBRECARGA**

JAVA:

```
public class Aluno extends Pessoa{
    private double nota1;
        private double nota2;


        public String getNome() {...4 linhas }
    public double getNota1() {...3 linhas }

    public void setNota1(double nota1) {...3 linhas }

    public double getNota2() {...3 linhas }

    public void setNota2(double nota2) {...3 linhas }


   public void imprimir(){
        System.out.println("Nome: "+ nome);
        System.out.println("Endereço: "+ end);
        System.out.println("Idade: "+ idade);
        System.out.println("Nota1: "+ nota1);
        System.out.println("Nota2: "+ nota2);

   }

   public void imprimir(double nota1, double nota2){
        double md= (nota1+nota2)/2;
        System.out.println("A média é:"+ md);
   }
```

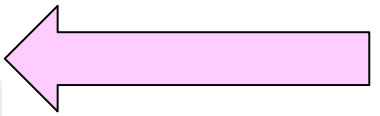


PROGRAMAÇÃO ORIENTADA A OBJETO

Na prática: **SOBRECARGA**

JAVA:

```
11      public static void main(String[] args) {
12
13
14          Aluno a = new Aluno();
15
16
17          a.setNome("Matheus");
18          a.setEnd("Rua suvaco da minhoca");
19          a.setIdade(19);
20          a.setNota1(7.9);
21          a.setNota2(10.0);
22
23          a.imprimir(a.getNota1(), a.getNota2());
24
25
26      }
27
```



Com 2 parâmetros!!!

```
Saída -
run:
A média é:8.95
CONSTRUÍDO COM SUCESSO (tempo total: 0 segundos)
```

PROGRAMAÇÃO ORIENTADA A OBJETO

Para o uso de INTERFACES em PHP usamos a palavra reservada: **IMPLEMENTS**

```php
<?php

  class Assinatura extends Produto implements Expiravel {

     private $dataExpiracao;

     public function getDataExpiracao()
     {
         return $this->dataExpiracao;
     }
     public function setDataExpiracao($dataExpiracao)
     {
         $this->dataExpiracao = new \DateTime($dataExpiracao);
     }
     public function getTempoRestante()
     {
         return $this->dataExpiracao->diff(new \DateTime());
     }
  }
```

PROGRAMAÇÃO ORIENTADA A OBJETO

- **Interfaces**

As interfaces oferecem um conjunto de métodos públicos que não possuem corpo. Uma classe que implementa uma interface deve prover implementações concretas de todos os métodos definidos pela interface, ou ela deve ser declarada como abstrata.
Uma interface é declarada usando-se a palavra-chave interface, seguida do nome da interface e de um conjunto de declarações de métodos.

```
public interface Cantor  {
    void cantar();
}
```

Uma classe que implementa uma interface utiliza a palavra-chave implements na definição da classe.

```
class RobertoCarlos  implements Cantor {
  public void cantar() {
    System.out.println("Você meu amigo de fé...");
  }
}
```

As classes podem implementar múltiplas interfaces, e as interfaces podem estender múltiplas interfaces.

PROGRAMAÇÃO ORIENTADA A OBJETO

- **Interfaces**
- **As interfaces são padrões definidos através de contratos ou especificações. Um contrato define um determinado conjunto de métodos que serão implementados nas classes que assinarem esse contrato. Uma interface é 100% abstrata, ou seja, os seus métodos são definidos como abstract, e as variáveis por padrão são sempre constantes (static final).**
- **Uma interface é definida através da palavra reservada "interface". Para uma classe implementar uma interface é usada a palavra "implements", descrita na Listagem 8.**
- **Como a linguagem Java não tem herança múltipla, as interfaces ajudam nessa questão, pois bem se sabe que uma classe pode ser herdada apenas uma vez, mas pode implementar inúmeras interfaces. As classes que forem implementar uma interface terão de adicionar todos os métodos da interface ou se transformar em uma classe abstrata, veja nos exemplos abaixo.**
- **Na Listagem 7 a interface Conta tem seus métodos sem corpo, apenas com os parâmetros e o tipo de retorno.**

```
1  interface Conta{
2      void depositar(double valor);
3      void sacar(double valor);
4      double getSaldo();
5  }
```

Listagem 7. Declaração de uma interface

PROGRAMAÇÃO ORIENTADA A OBJETO

Para o uso de INTERFACES em PHP usamos a palavra reservada: **IMPLEMENTS**

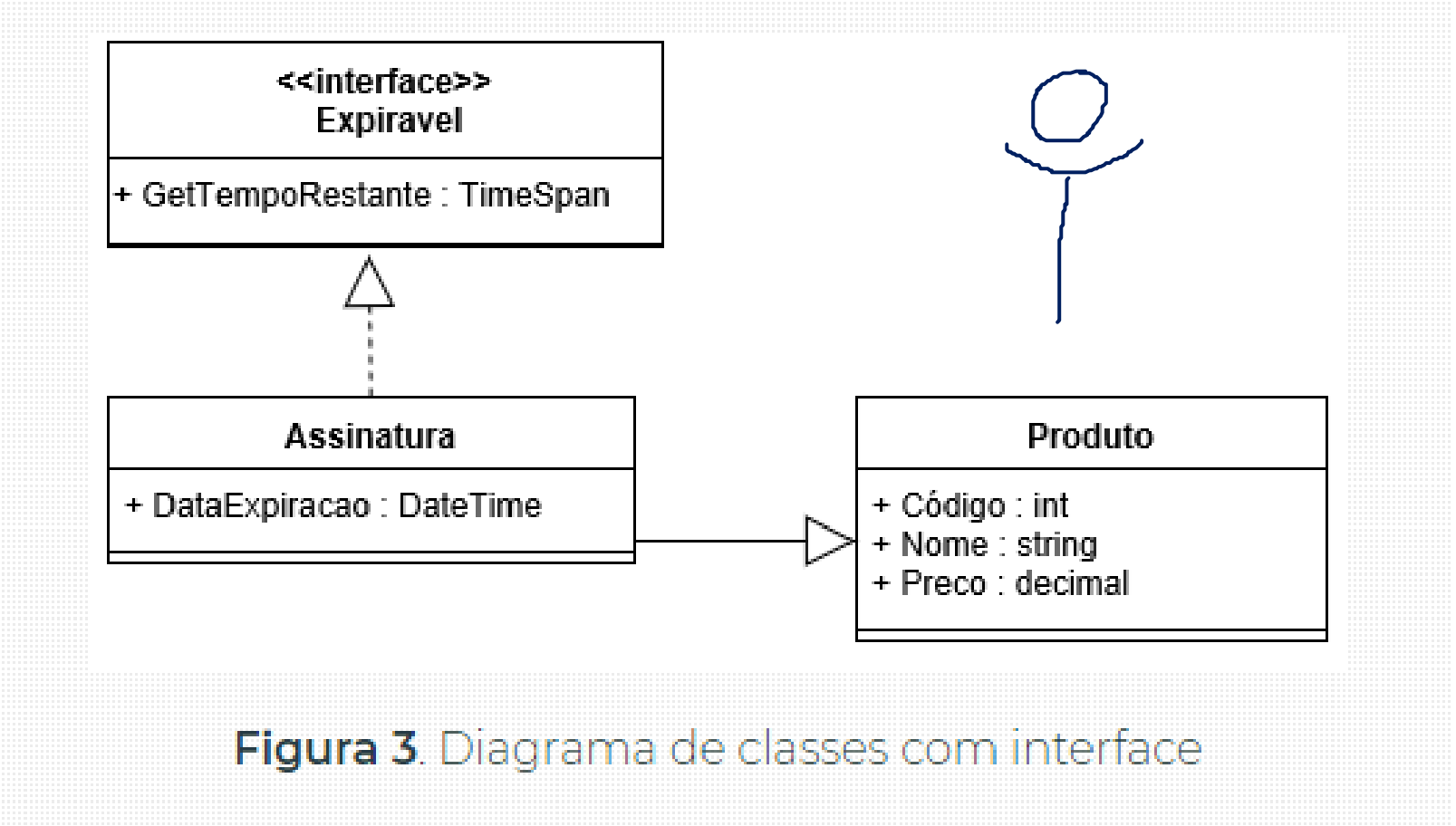


Figura 3. Diagrama de classes com interface

PROGRAMAÇÃO ORIENTADA A OBJETO

Mais exemplos práticos sobre POO:

https://www.devmedia.com.br/sobrecarga-e-sobreposicao-de-metodos-em-orientacao-a-objetos/33066

PROGRAMAÇÃO ORIENTADA A OBJETO

UML

Basicamente, UML (Unified Modeling Language) é uma **linguagem de notação** (um jeito de escrever, ilustrar, comunicar) para uso em projetos de sistemas.

Esta linguagem é expressa através de diagramas. Cada **diagrama** é composto por **elementos** (formas gráficas usadas para os desenhos) que **possuem relação** entre si.

Os diagramas da UML se dividem em dois grandes grupos: diagramas **estruturais** e diagramas **comportamentais**.

Diagramas estruturais devem ser utilizados para especificar **detalhes da estrutura do sistema** (parte estática), por exemplo: classes, métodos, interfaces, namespaces, serviços, como componentes devem ser instalados, como deve ser a arquitetura do sistema etc.

Diagramas comportamentais devem ser utilizados para especificar **detalhes do comportamento do sistema** (parte dinâmica), por exemplo: como as funcionalidades devem funcionar, como um processo de negócio deve ser tratado pelo sistema, como componentes estruturais trocam mensagens e como respondem às chamadas etc.

PROGRAMAÇÃO ORIENTADA A OBJETO

UML

UML deixa as coisas claras

UML ajuda muito a deixar o escopo claro, pois centraliza numa única visão (o diagrama) um determinado conceito, utilizando uma linguagem que todos os envolvidos no projeto podem facilmente entender.

Mas ajuda desde que **utilizada na medida certa**, ou seja, apenas **quando realmente é necessário**.

O maior problema na produção de software, a maior dor, em qualquer país do mundo, chama-se **comunicação ruim**.

Vejamos um rápido exemplo **didático** de como se dá a comunicação em equipes de produção de software:

> *\/* João quer A, explica à equipe algo "parecido" com B. Marcos entende que João quer C, e explica para Claudia que é para fazer D. Claudia faz um "D que mais se parece um E", e entrega um "meio E" para João. E João queria um A... */*

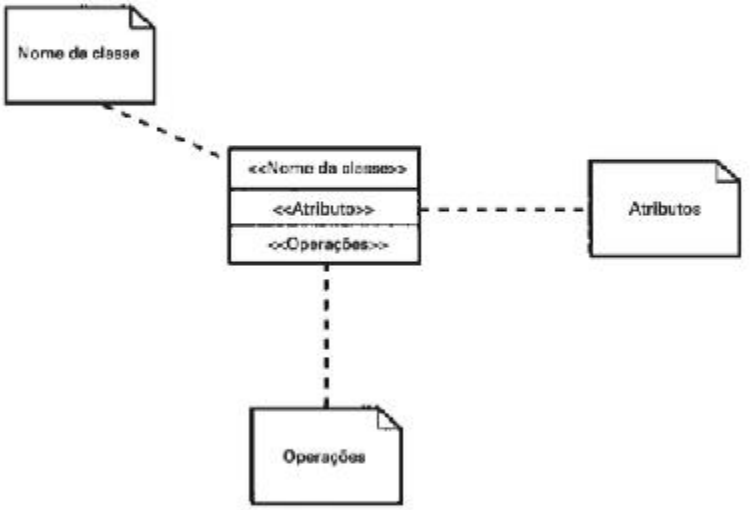
PROGRAMAÇÃO ORIENTADA A OBJETO

UML

Introdução à Unified Modeling Language

Quando um construtor constrói uma casa, ele não faz isso aleatoriamente. Em vez disso, o construtor constrói a casa de acordo com um conjunto de cópias heliográficas detalhadas. Essas cópias heliográficas dispõem o projeto da casa explicitamente. Nada é deixado ao acaso.

FIGURA 8.1
A notação de classe da UML.

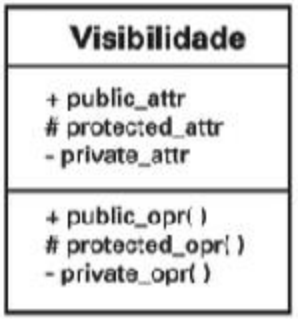


Dentro do modelo, você pode usar os caracteres -, # e +. Esses caracteres transmitem a *visibilidade* de um atributo ou de uma operação. O hifen (-) significa privado, o jogo da velha (#) significa protegido e o sinal de adição (+) significa público (veja a Figura 8.2).

PROGRAMAÇÃO ORIENTADA A OBJETO

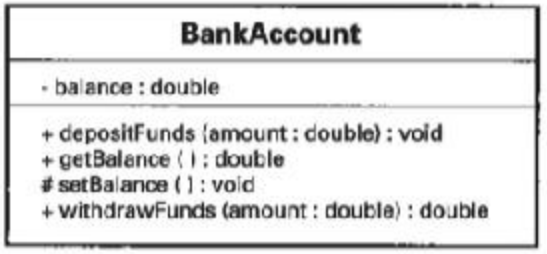
UML

FIGURA 8.2
A notação da UML para especificar visibilidade.



A Figura 8.3 ilustra a completa classe `BankAccount` dos dias 5 e 7.

FIGURA 8.3
Uma classe totalmente descrita.



PROGRAMAÇÃO ORIENTADA A OBJETO

UML

Às vezes, uma nota ajudará a transmitir um significado que, de outro modo, ficaria perdido ou seria ignorado, como a nota da Figura 8.4.

FIGURA 8.4
Um exemplo detalhado de nota.

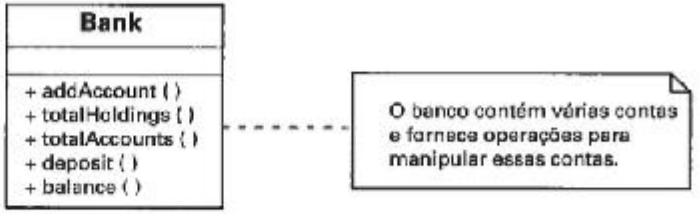
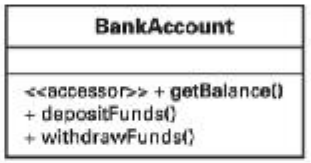
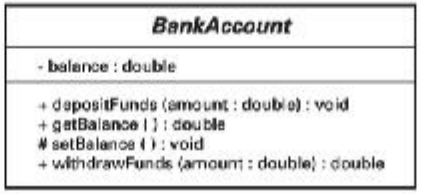


FIGURA 8.5
Um estereótipo que qualifica a operação.



Finalmente, você pode se lembrar que a classe `BankAccount` foi originalmente definida como uma classe concreta. Entretanto, o Dia 7 redefiniu a classe `BankAccount` como uma classe abstrata. A UML fornece uma notação para transmitir que uma classe é abstrata: o nome da classe abstrata é escrito em itálico. No caso de `BankAccount`, o nome deve ser escrito em itálico, conforme ilustrado na Figura 8.6.

FIGURA 8.6
O objeto `BankAccount` *abstrato.*



PROGRAMAÇÃO ORIENTADA A OBJETO

UML

Modelando um relacionamento de classe

As classes não existem no vácuo. Em vez disso, elas têm relacionamentos complexos entre si. Esses relacionamentos descrevem como as classes interagem umas com as outras.

NOVO TERMO Um *relacionamento* descreve como as classes interagem entre si. Na UML, um relacionamento é uma conexão entre dois ou mais elementos da notação.

A UML reconhece três tipos de alto nível de relacionamentos de objeto:

- Dependência
- Associação
- Generalização

Embora a UML possa fornecer notação para cada um desses relacionamentos, os relacionamentos não são específicos da UML. Em vez disso, a UML simplesmente fornece um mecanismo e vocabulário comum para descrever os relacionamentos. Entender os relacionamentos, independentemente da UML, é importante em seu estudo de OO. Na verdade, se você simplesmente ignorar a notação e entender os relacionamentos, estará bem adiantado nos estudos.

PROGRAMAÇÃO ORIENTADA A OBJETO

UML

Dependência

Dependência é o relacionamento mais simples entre objetos. A dependência indica que um objeto depende da especificação de outro objeto.

NOTA

Especificação é uma maneira diferente de dizer interface ou comportamento.

NOVO TERMO

Em um *relacionamento de dependência*, um objeto é dependente da especificação de outro objeto. Se a especificação mudar, você precisará atualizar o objeto dependente.

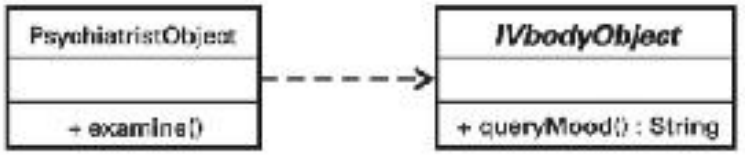


FIGURA 8.8
Um relacionamento de dependência simples.

PROGRAMAÇÃO ORIENTADA A OBJETO

UML

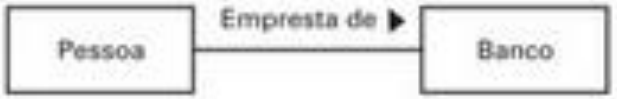
Associação

Os relacionamentos de associação vão um pouco mais fundo do que os relacionamentos de dependência. As associações são relacionamentos estruturais. Uma associação indica que um objeto contém — ou que está conectado a — outro objeto.

NOVO TERMO Uma *associação* indica que um objeto contém outro objeto. Nos termos da UML, quando se está em um relacionamento de associação, um objeto está conectado a outro.

Como os objetos estão conectados, você pode passar de um objeto para outro. Considere a associação entre uma pessoa e um banco, como ilustrado na Figura 8.9.

FIGURA 8.9
Uma associação entre uma pessoa e um banco.



A Figura 8.9 mostra que uma pessoa *empresta de* um banco. Na notação UML, toda associação tem um nome. Neste caso, a associação é chamada de *empresta de*. A seta indica a direção da associação.

PROGRAMAÇÃO ORIENTADA A OBJETO

UML

NOVO TERMO O nome da associação é um nome que descreve o relacionamento.

Cada objeto em uma associação também tem um papel, conforme indicado na Figura 8.10.

FIGURA 8.10
Os papéis na associação.

NOVO TERMO Na associação, o papel de `Pessoa` é devedor e o papel de `Banco` é credor.

NOVO TERMO O papel da associação é a parte que um objeto desempenha em um relacionamento.

Finalmente, a multiplicidade indica quantos objetos podem tomar parte em uma associação.

NOVO TERMO A *multiplicidade* indica quantos objetos podem tomar parte na instância de uma associação.

A Figura 8.11 ilustra a multiplicidade da associação entre `Pessoa` e `Banco`.

FIGURA 8.11
Multiplicidade.

PROGRAMAÇÃO ORIENTADA A OBJETO

UML

Essa notação nos informa que um banco pode ter um ou mais devedores e que uma pessoa pode utilizar 0 ou mais bancos.

Você especifica suas multiplicidades através de um único número, uma lista ou com um asterisco (*).

Um único número significa que determinado número de objetos — não mais e não menos — podem participar da associação. Assim, por exemplo, um 6 significa que seis objetos e somente seis objetos podem participar da associação.

* significa que qualquer número de objetos pode participar da associação.

Uma lista define um intervalo de objetos que podem participar da associação. Por exemplo, 1..4 indica que de 1 a 4 objetos podem participar da associação. 3..* indica que três ou mais objetos podem participar.

Quando você deve modelar associações?

Você deve modelar associações quando um objeto contiver outro objeto — o relacionamento *tem um*. Você também pode modelar uma associação quando um objeto usa outro. Uma associação permite que você modele quem faz o que em um relacionamento.

PROGRAMAÇÃO ORIENTADA A OBJETO

UML

A UML também define dois tipos de associação: *agregação* e *composição*. Esses dois subtipos de associação o ajudam a refinar mais seus modelos.

Agregação

Uma agregação é um tipo especial de associação. Uma agregação modela um relacionamento *tem um* (ou *parte de*, no jargão da UML) entre pares. Esse relacionamento significa que um objeto contém outro. *Pares* significa que um objeto não é mais importante do que o outro.

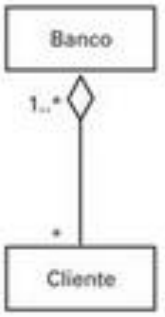
NOVO TERMO Um *relacionamento todo/parte* descreve o relacionamento entre objetos onde um objeto contém outro.

NOVO TERMO Uma *agregação* é um tipo especial de associação que modela o relacionamento 'tem um' de relacionamentos todo/parte entre pares.

Importância, no contexto de uma agregação, significa que os objetos podem existir independentemente uns dos outros. Nenhum objeto é mais importante do que o outro no relacionamento.

Considere a agregação ilustrada pela Figura 8.12.

FIGURA 8.12
Agregação entre um banco e seus clientes.



PROGRAMAÇÃO ORIENTADA A OBJETO

UML

Composição

A composição é um pouco mais rigorosa do que a agregação. A composição não é um relacionamento entre pares. Os objetos não são independentes uns dos outros.

A Figura 8.13 ilustra um relacionamento de composição.

FIGURA 8.13
Composição entre um banco e suas filiais.

Aqui, você vê que `Banco` pode conter muitos objetos `Filial`. O losango fechado diz que esse é um relacionamento de composição. O losango também diz quem 'tem um'. Neste caso, `Banco` 'tem um', ou contém, objetos `Filial`.

NOTA

Um losango fechado simboliza a composição. O losango toca o objeto que é considerado o todo do relacionamento. O todo é constituído de partes. No exemplo anterior, `Banco` é o todo e os objetos `Filial` são as partes.

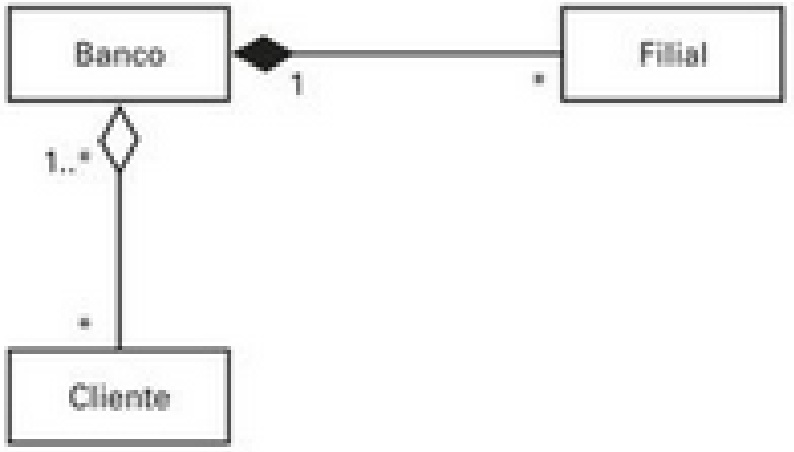
Como esse é um relacionamento de composição, os objetos `Filial` não podem existir independentemente do objeto `Banco`. A composição diz que, se o banco encerrar suas atividades, as filiais também fecharão. Entretanto, o inverso não é necessariamente verdade. Se uma filial fechar, o banco poderá permanecer funcionando.

Um objeto pode participar de uma agregação e de um relacionamento de composição ao mesmo tempo. A Figura 8.14 modela tal relacionamento.

PROGRAMAÇÃO ORIENTADA A OBJETO

UML

FIGURA 8.14
Banco em um relacionamento de agregação e de composição, simultaneamente.



PROGRAMAÇÃO ORIENTADA A OBJETO

UML

Generalização

Um relacionamento de generalização é um relacionamento entre o geral e o específico. É a herança.

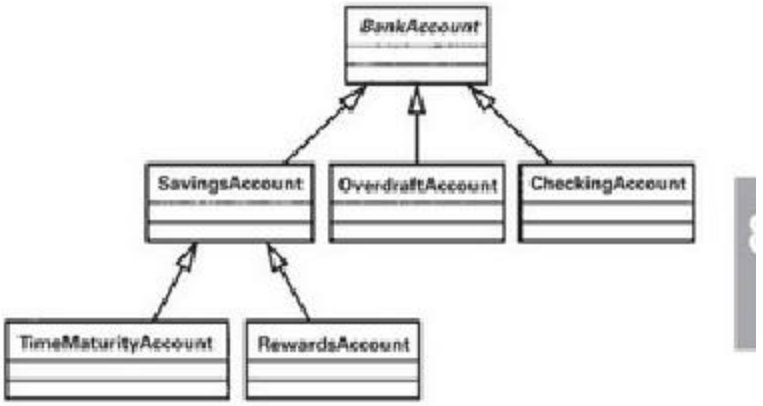
NOVO TERMO Um *relacionamento de generalização* indica um relacionamento entre o geral e o específico. Se você tem um relacionamento de generalização, então sabe que pode substituir uma classe filha pela classe progenitora.

A generalização incorpora o relacionamento *'é um'* sobre o qual você aprendeu no Dia 4. Conforme foi aprendido no Dia 4, os relacionamentos 'é um' permitem que você defina relacionamentos com capacidade de substituição.

Através de relacionamentos com capacidade de substituição, você pode usar descendentes em vez de seus ancestrais, ou filhos em vez de seus progenitores.

A UML fornece uma notação para modelar generalização. A Figura 8.15 ilustra como você modelaria a hierarquia de herança `BankAccount`.

FIGURA 8.15
A hierarquia de herança `BankAccount`.

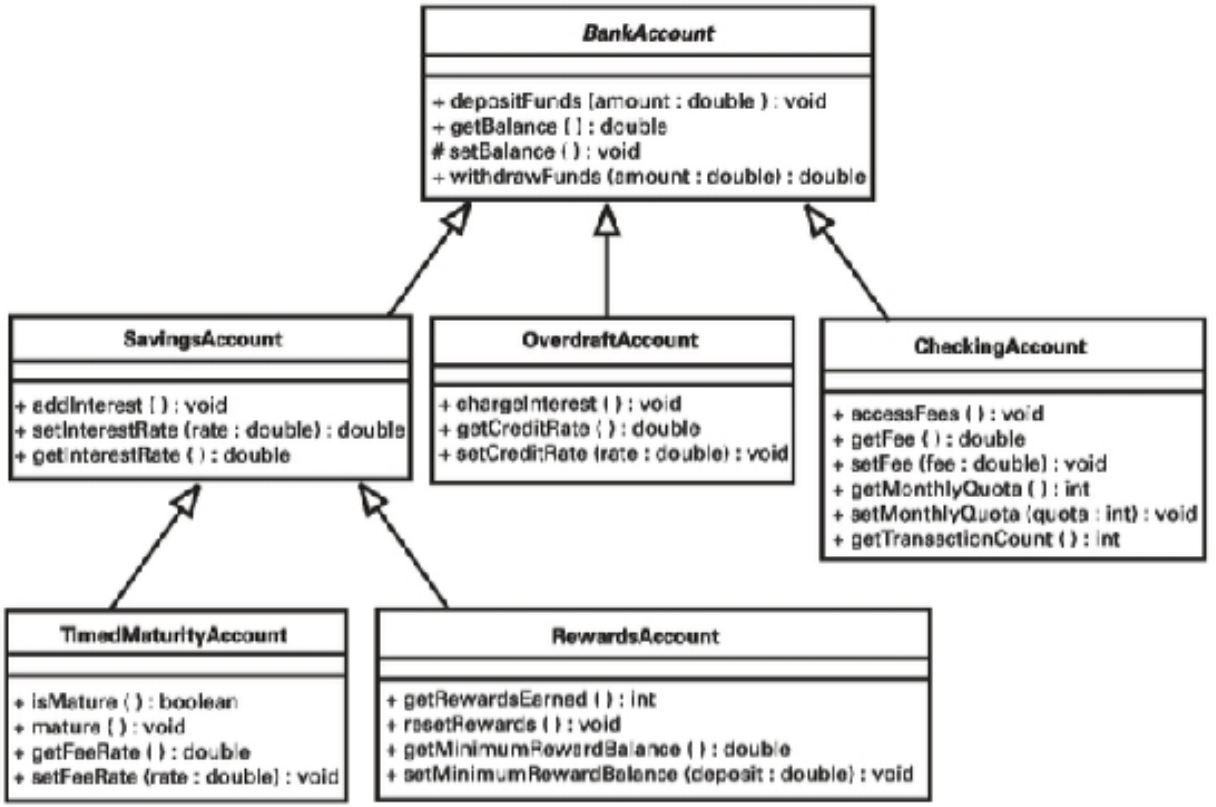


Uma linha cheia com uma seta fechada e vazada indica um relacionamento de generalização.

PROGRAMAÇÃO ORIENTADA A OBJETO

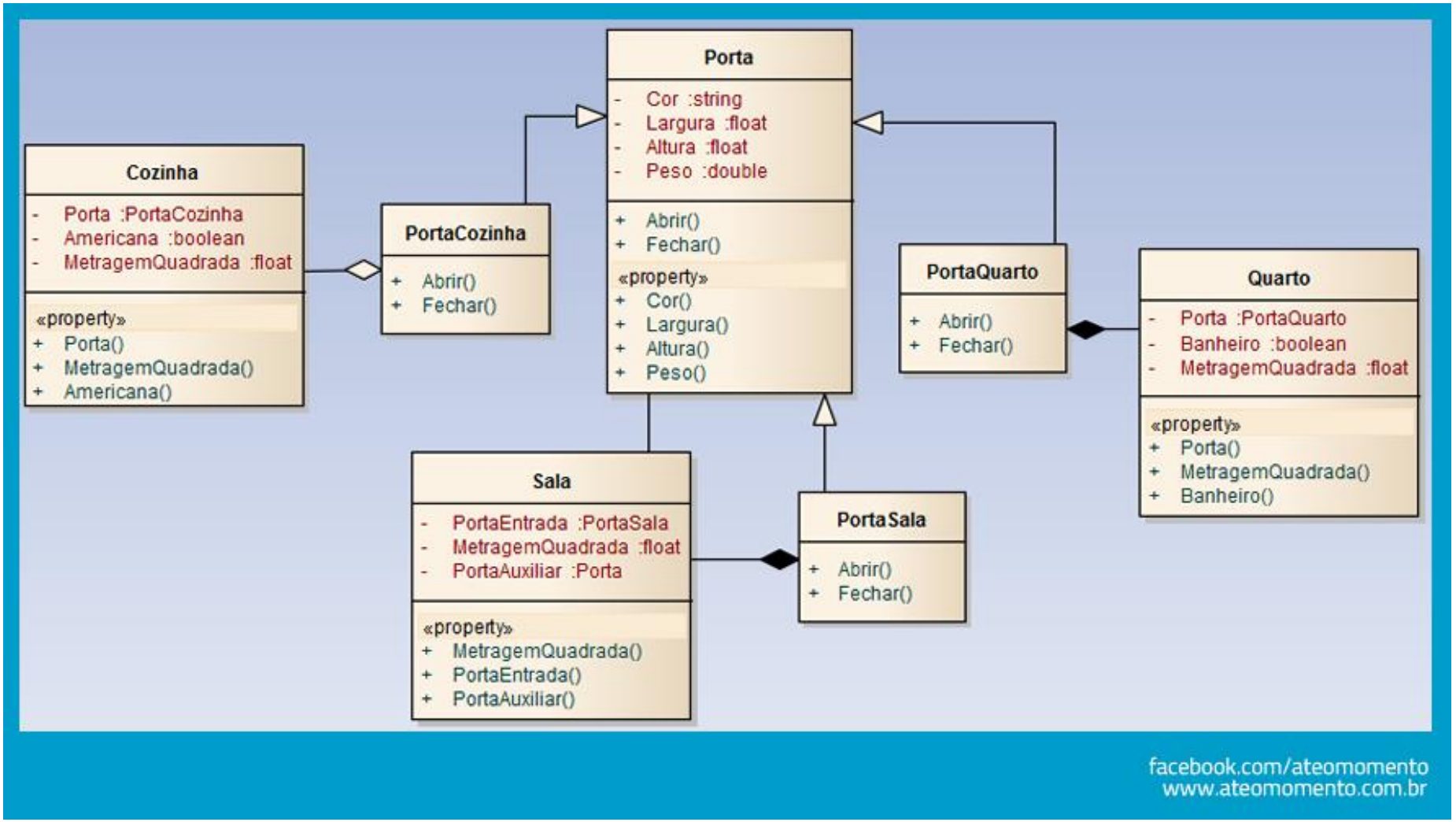
UML

FIGURA 8.17
Uma hierarquia de herança BankAccount *mais detalhada.*



PROGRAMAÇÃO ORIENTADA A OBJETO

UML - Diagrama de Classes Completo



PROGRAMAÇÃO ORIENTADA A OBJETO

UML - Diagrama de Classes Completo

PROGRAMAÇÃO ORIENTADA A OBJETO

Exercícios

1) A programação orientada a objetos define 6 objetivos propostos para o desenvolvimento de software. A POO se esmera em produzir software com características marcantes. Dentre elas marque a alternativa que define que o software não é estático e que ele deve crescer e mudar com o passar do tempo, assim o mantendo útil:

A) Oportuno

B) Extensível

C) Natural

D) Reutilizável

PROGRAMAÇÃO ORIENTADA A OBJETO

Exercícios

1) A programação orientada a objetos define 6 objetivos propostos para o desenvolvimento de software. A POO se esmera em produzir software com características marcantes. Dentre elas marque a alternativa que define que o software não é estático e que ele deve crescer e mudar com o passar do tempo, assim o mantendo útil:

A) Oportuno

B) Extensível

C) Natural

D) Reutilizável

PROGRAMAÇÃO ORIENTADA A OBJETO

Exercícios

2) Marque Verdadeiro ou falso e assinale a alternativa correspondente:

( ) A POO estrutura um programa, dividindo-o em vários objetos de alto nível, assim cada objeto modela algum aspecto do problema que você está tentando resolver.
( ) As linguagens fortemente tipadas exigem que cada objeto tenha um tipo específico e definido.
( ) O estado de um objeto é o significado combinado das variáveis externas do objeto
( ) A implantação define como algo é feito. Em termos de programação, Implantação É O CÓDIGO.

A) V – V – V – F
B) F – F – F – V
C) V – V – F – F
D) V – F – V – F

PROGRAMAÇÃO ORIENTADA A OBJETO

Exercícios

2) Marque Verdadeiro ou falso e assinale a alternativa correspondente:

( ) A POO estrutura um programa, dividindo-o em vários objetos de alto nível, assim cada objeto modela algum aspecto do problema que você está tentando resolver.
( ) As linguagens fortemente tipadas exigem que cada objeto tenha um tipo específico e definido.
( ) O estado de um objeto é o significado combinado das variáveis externas do objeto
( ) A implantação define como algo é feito. Em termos de programação, Implantação É O CÓDIGO.

A) V – V – V – F
B) F – F – F – V
C) V – V – F – F
D) V – F – V – F

PROGRAMAÇÃO ORIENTADA A OBJETO

Exercícios

3) _____define como algo é feito, em termos de programação, é o código. Enquanto _____ é o espaço onde o problema reside.

a) Encapsulamento e Algorítmo
b) Implementação e Domínio
c) Programação Orientada a Objetos e IDE
d) UML e Programação Orientada a Objetos

PROGRAMAÇÃO ORIENTADA A OBJETO

Exercícios

3) _____define como algo é feito, em termos de programação, é o código. Enquanto _____ é o espaço onde o problema reside.

a) Encapsulamento e Algorítmo
b) Implementação e Domínio
c) Programação Orientada a Objetos e IDE
d) UML e Programação Orientada a Objetos

PROGRAMAÇÃO ORIENTADA A OBJETO

Exercícios

4) Sobre o conceito de classes e seus itens básicos na POO, complete as lacunas em branco e marque a opção correta:

Uma classe define todas as características comuns a um tipo de _____.

Especificamente, a classe define todos os _____ e _____ expostos pelo _____. Quando um objeto quer exercer o comportamento de outro objeto, ele não faz isso diretamente, mas pede ao outro objeto para que se mude, normalmente baseado em alguma informação adicional. Frequentemente, isso é referido como _____.

a) comportamento, atributos, objetos, comportamento e envio de mensagem

b) objeto, atributos, comportamentos, objeto e envio de mensagem

c) atributos, objetos, métodos, atributos e envio de mensagem

d) objeto, atributos, métodos, domínio e envio de mensagem

PROGRAMAÇÃO ORIENTADA A OBJETO

Exercícios

4) Sobre o conceito de classes e seus itens básicos na POO, complete as lacunas em branco e marque a opção correta:

Uma classe define todas as características comuns a um tipo de _____.

Especificamente, a classe define todos os _____ e _____ expostos pelo _____. Quando um objeto quer exercer o comportamento de outro objeto, ele não faz isso diretamente, mas pede ao outro objeto para que se mude, normalmente baseado em alguma informação adicional. Frequentemente, isso é referido como _____.

a) comportamento, atributos, objetos, comportamento e envio de mensagem

b) objeto, atributos, comportamentos, objeto e envio de mensagem

c) atributos, objetos, métodos, atributos e envio de mensagem

d) objeto, atributos, métodos, domínio e envio de mensagem

PROGRAMAÇÃO ORIENTADA A OBJETO

Exercícios

Assinale a alternativa que apresenta, corretamente, o conceito de programação orientada a objetos que promove a reutilização de software.

A) Abstração de dados.

B) Herança.

C) Polimorfismo.

D) Sobrecarga de métodos.

PROGRAMAÇÃO ORIENTADA A OBJETO

Exercícios

Assinale a alternativa que apresenta, corretamente, o conceito de programação orientada a objetos que promove a reutilização de software.

A) Abstração de dados.

B) Herança.

C) Polimorfismo.

D) Sobrecarga de métodos.