Universidade Comunitária da Região de Chapecó

Ciência da Computação/Sistemas de informação

Algoritmos e Programação

OO, Abstração/Java.

Professor Valdemar LORENZON Junior

Revisão 2025-2

vljunior@unochapeco.edu.br

Este material foi construído pelo Professor Valdemar Lorenzon Junior, trazendo adaptações de material de própria autoria ao longo de anos de dedicação e trabalhos com pesquisa, desenvolvimento e docência com OO, recuperação de cursos certificados SUN e Oracle.

Este material é de propriedade intelectual do professor Valdemar Lorenzon Junior e pode ser utilizado se referenciado.

Material original criado em 2001 e reestruturado semestralmente com atualizações necessárias para atender a ementa, atualizações de conteúdo teórico e prático para adaptação ao momento de uso (semestre letivo e atualizações tecnológicas ou acadêmicas).

É um material teórico/prático enriquecido com atualizações de conteúdo e certificações de updates, podendo trazer alguma incompatibilidade em algum momento que pode ser corrigida quando observada.

Sumário

Parte I - Programação OO (OOP/POO)

A base da orientação a objetos é uma nova visão do problema, através da sua representação de conceitos do mundo real.

Segundo esta abordagem, o elemento básico de representação dos problemas e de construção de suas soluções é o objeto, conceito que permite combinar informações e o comportamento sobre estas numa única entidade: **a classe!**

Este é um dos diferenciais desta abordagem com relação a abordagens estruturadas, uma vez que, nelas, informações/atributos (variáveis) e comportamento/métodos (funções e procedimentos) são tratados de modo independente (fraco acoplamento).

Técnicas tradicionais versus orientação a objetos

Partindo do ponto de vista do programador de um paradigma anterior até o estruturado, algumas técnicas orientadas ao objeto parecem ser conceitos tradicionais com nomes diferentes. Alguns conceitos orientados a objeto são análogos à programação convencional.

Apresento aqui os contrastes entre termos e conceitos convencionais e orientado a objetos:

- As informações da classe ou dos métodos correspondem as declarações de variáveis na programação tradicional estendendo a capacidade de tipos para além de tipos primitivos com as classes;
- Os ponteiros serão implicitamente utilizados a partir das referências, as variáveis declaradas do tipo classes;
- Um método, comportamento, é como um procedimento ou função porque contêm instruções de processamento com retorno e assinatura. Porém, a função era isolada e independente e o método fará parte de um comportamento de uma estrutura denominada classe;
- A classe não é um tipo de dado primitivo. Uma classe é uma evolução de um tipo de dado. Um tipo de dado especial cujo seu "tipo" é representativo de um contexto criado pelo programador;
- A reusabilidade, reutilização, oriunda de classes e pacotes serão mais amplamente explorados na OO com o conceito de herança e polimorfismo do que na programação tradicional top down a partir de bibliotecas e funções;
- O "envio de mensagens" para objetos substitui a ideia de chamadas de função;

Apesar dos programadores tradicionais rapidamente perceberem similaridades com suas técnicas, será preciso que eles mudem suas imagens de programas sequenciais que contam com sequência de instruções para um sistema de objetos interativos (oriundo da orientação a objetos e orientação a eventos).

A tabela a seguir resume algumas diferenças entre as perspectivas convencionais e as OO.

Paradigma OO	Paradigmas pré OO
Comportamento, métodos.	Procedimentos, funções ou sub-rotinas.
Variáveis de instância, atributos.	Variáveis, informações, dados.
Classes.	Não há. Tipos de dados heterogêneo é próximo pela agregação de valores e notação ponto, mas não comporta acoplamento de comportamento.
Visibilidade.	Algo como escopo, mas não procede.
Hereditariedade.	Não tem.
Polimorfismo.	Não existe nada além do casting.
Mensagens.	Chamadas de funções.
Referências.	Ponteiros.
Objetos.	"Alocação na Heap", mas de valores isolados.

Abstração em tipos

Os fundamentos da programação OO residem na ideia da **abstração**. As linguagens de programação foram inventadas e são usadas para descrever um processo em nível mais alto que do código de máquina.

Tradicionalmente, a abstração oferecida pelas linguagens de programação pode ser dividida em duas categorias, relacionadas com a representação dos dados e com as estruturas de controle de fluxo.

A ideia fundamental da abstração na representação de dados é o conceito de tipo de dado. Historicamente, a primeira geração de linguagens de programação lançou o conceito de tipo, e a segunda geração incluiu a ideia de tipos de dados definidos pelo usuário. Desta forma, com o passar do tempo evoluímos de:

Tipos básicos, primitivos de dados

Os tipos básicos ou primitivos de dados são os blocos de construção mais simples e fundamentais da programação. Eles representam tipos de dados simples com operações predefinidas e ocupam um tamanho fixo de memória. Os tipos primitivos comuns incluem:

- Inteiro (int): Representa números inteiros, como 1, -3, 42;

- Ponto flutuante (float/double): Números reais (casas decimais), como 3.14, -0.001;
- Caractere (char): Representa um único caractere, como 'a', 'Z', '3';
- Booleano (bool): Representa verdadeiro ou falso.

Tipos avançados de dados

Os tipos avançados de dados são construídos a partir de tipos primitivos ou outros tipos avançados e oferecem mais funcionalidades. Eles podem armazenar coleções de dados ou estruturas mais complexas. Exemplos incluem:

- Strings: Sequências de caracteres, como "Olá, mundo!";
- Arrays (Vetores): Coleção de elementos do mesmo tipo, armazenados contiguamente na memória.

Tipos de dados definidos pelo usuário

Os tipos de dados definidos pelo usuário permitem que os programadores criem tipos de dados que se adaptam às necessidades específicas de seus programas, utilizando tipos primitivos e avançados como blocos de construção. Exemplos comuns incluem:

- Estruturas (structs): Coleção de variáveis (possivelmente de tipos diferentes) agrupadas sob um nome;
- Uniões (Unions): Semelhante às estruturas, mas a memória é compartilhada entre os campos;
- Enumerações (Enums): Define um tipo que pode ter um de vários valores nomeados predefinidos.

Tipos de dados abstratos

Os tipos de dados abstratos são modelos matemáticos para tipos de dados, onde um tipo é definido pelo seu comportamento (visto a partir das operações do usuário) ao invés de por sua implementação. São fundamentais na ciência da computação e incluem:

- Pilha (Stack): Uma coleção de elementos com duas operações principais: push (adiciona um item) e pop (remove o item mais recentemente adicionado);
- Fila (Queue): Uma coleção onde o primeiro elemento a ser adicionado é o primeiro a ser removido (FIFO: first-in, first-out);
- Árvores, Grafos, Tabelas de Hash, entre outros.

Estes conceitos são amplamente utilizados em programação e são cruciais para o design e a implementação de software eficiente e eficaz. Cada linguagem de programação tem sua própria maneira de definir e trabalhar com esses tipos de dados, mas os conceitos básicos permanecem consistentes entre elas.

Posteriormente, a ideia de tipo de dado evoluiu no conceito de tipo de dado abstrato, com um novo tipo que associa a representação de dados com a operação usada para manipulá-lo. Associando a este, pilares conceituais e de suporte na linguagem de programação para manipulação destes tipos.

Em um tipo de dado abstrato, a representação dos dados normalmente fica oculta aos usuários. Nessas linguagens, um novo tipo de dado é percebido como uma extensão do sistema de tipos fornecida pela linguagem.

As classes, conforme definido pelas linguagens de programação **OO**, podem ser consideradas uma extensão definitiva do conceito de tipos de dados abstratos.

Pelo lado das estruturas de controle, as primeiras linguagens de programação introduziram declarações para saltos, ampliações e loops. Então vem a ideia de sub-rotinas (procedimentos e funções). Com o advento das sub-rotinas, foram desenvolvidas muitas teorias sobre programação inclusive a decomposição funcional top-down.

Assim como o top-down veio substituir o código "spaghetti" gerado pelo "goto", a OO veio enriquecer o paradigma estruturado.

Atualmente ainda não há um novo paradigma, apenas extensões e tudo permeia a OO num paralelo com a programação funcional.

A abstração de sub-rotinas é ilustrada pelo fato de você poder chamá-las já sabendo o que elas fazem, mas não precisa saber como o fazem. De maneira similar a tipos de dados, as novas sub-rotinas podem ser consideradas extensões da linguagem de programação.

Uma classe é uma abstração de um conjunto de coisas do mundo real, de forma que:

- Todas as coisas do mundo real do conjunto tenham as mesmas características;
- Todas as instâncias estejam sujeitas as mesmas normas.

A abstração é a capacidade da linguagem de programação possuir suporte à criação de classes que definem (tipificam) um objeto.

Desta forma, conceituamos o primeiro pilar da OO, a Abstração.

Linha do tempo OO

A linha do tempo da Programação Orientada a Objetos (OO) destaca marcos importantes na evolução desse paradigma de programação. Aqui está uma visão geral dos principais eventos:

1967: Linguagem **Simula 67**. Desenvolvida por Ole-Johan Dahl e Kristen Nygaard na Noruega, Simula 67 é considerada a primeira linguagem de programação orientada a objetos.

1972: Linguagem **Smalltalk**. Criada por Alan Kay, Dan Ingalls, e Adele Goldberg no Xerox PARC, Smalltalk é a primeira linguagem a implementar a programação orientada a objetos de forma abrangente.

1983: Linguagem C++. Bjarne Stroustrup desenvolve o C++, uma extensão do C que incorpora conceitos de orientação a objetos, como classes e herança.

1985: Publicação de "Object-Oriented Software Construction". Bertrand Meyer publica um dos primeiros livros a formalizar os conceitos de OO, ajudando a popularizar o paradigma.

1986: Fundamentos do padrão de projeto. Kent Beck e Ward Cunningham começam a trabalhar no conceito de padrões de projeto, que mais tarde serão popularizados pelo "Gang of Four".

1990: Publicação do padrão de design Patterns.O "Gang of Four" (Erich Gamma, Richard Helm, Ralph Johnson, e John Vlissides) publica "Design Patterns: Elements of Reusable Object-Oriented Software", formalizando 23 padrões de design OO.

1991: Linguagem **Python**. Guido van Rossum lança Python, que, apesar de não ser puramente OO, suporta totalmente o paradigma OO.

1995: Linguagem **Java**. A Sun Microsystems lança Java, projetada para ser portável e segura, com forte suporte à OO.

1995. **Delphi 1.0** foi lançado pela **Borland**, utilizando Object Pascal como linguagem principal. Combinava programação orientada a objetos, interface visual (RAD) e acesso rápido a banco de dados. Era OO: sim, e com uso de herança, encapsulamento e eventos orientados a objetos (componentes visuais). Viveu paralelo com Java e C# até 2000 onde iniciou seu declínio. Atualmente mantido pela **Embarcadero**, ainda "existe".

1995: Linguagem Ruby. Yukihiro "Matz" Matsumoto lança Ruby, uma linguagem OO pura que enfatiza a simplicidade e a produtividade.

2000: Plataforma .NET e C#. A Microsoft lança a plataforma .NET e a linguagem C#, ambas projetadas com suporte robusto para OO.

2006: Scala. Martin Odersky lança Scala, uma linguagem que combina OO e programação funcional, executada na JVM. A programação funcional se refere a um paradigma de programação que contrasta com o paradigma orientado a objetos (OO). A programação funcional é um estilo de desenvolvimento de software baseado em funções puras, imutabilidade, expressões e composição de funções. Ela vem da matemática, especialmente do cálculo lambda.

2010: Swift. A Apple lança Swift, uma linguagem para desenvolvimento iOS e macOS, com forte suporte à OO.

2014: Dart. Google lança Dart, uma linguagem orientada a objetos projetada para o desenvolvimento web e móvel.

2020: Modernização contínua. Linguagens como Kotlin, Scala, F# e Swift reforçam o suporte a funções puras, imutabilidade e programação funcional, mantendo OO. OO ainda é usado, mas a composição de objetos (em vez de herança pesada) e a imutabilidade por padrão ganham força. A arquitetura de sistemas evolui para usar objetos imutáveis e eventos reativos, com base no paradigma funcional.

2021: Java 17 introduz recursos como **records**, **sealed classes** e **pattern matching**, que facilitam modelagem de dados de forma imutável e orientada a tipo — um reflexo da influência funcional. Passa a usar menos herança e mais **composição + imutabilidade**.

2022-23: OO é reinterpretado à luz de Domain-Driven Design (DDD): foco em modelagem orientada ao domínio e encapsulamento significativo.

As classes passam a refletir melhor os conceitos do domínio do negócio, não apenas estruturas de dados isoladas com métodos. O reflexo do mundo real é mais amplo para uma característica de domínio com seus conceitos e regras.

Arquitetura limpa e hexagonal organizam o código OO em camadas, desacoplando lógica de infraestrutura. Podemos imaginar um "MVC" diferente. Regras de negócio, casos de uso e infraestrutura. O domínio deve ser independente da tecnologia externa e esta se conecta ao domínio por portas criando as implementações de adaptação.

Rust ganha popularidade, mesmo sendo multi-paradigma, por sua segurança e controle de memória — influenciando práticas OO.

Kotlin se consolida como linguagem OO com forte viés funcional e é cada vez mais usada em back-end (além de Android).

TypeScript continua sendo amplamente adotada com suporte OO + tipos + FP.

Após 2020, o paradigma orientado a objetos não foi abandonado, mas evoluiu, absorvendo boas práticas de outros paradigmas. Foi desacoplado da herança rígida e passou a favorecer composição, imutabilidade e tipagem mais expressiva. Tornou-se mais modular, funcional e testável, influenciado por contextos como nuvem, microsserviços e programação concorrente.

Tendências

- Refinamento da composição sobre herança: padrão de projeto mais funcional e desacoplado;
- Mais linguagens com suporte a tipos algébricos, como sealed class, enum, option, result, etc;
- Orientação a objetos contextual: foco em objetos ricos em comportamento, mas com menos ênfase em estruturas clássicas de herança;
- Interoperabilidade entre paradigmas: sistemas combinando OO, funcional e reativo em diferentes camadas;
- Abordagens orientadas a mensagem (Actor Model): OO repensada com foco em entidades autônomas que se comunicam por mensagens, como em Akka ou Elixir (inspirado em Erlang)

Parte II – O Pilar da Abstração

Identificando classes e objetos

A identificação de abstrair para criar classes e programar utilizando objetos é uma tarefa que pode parecer fácil, porém observação e construção de experiência sobre o novo paradigma. Não é simplesmente criar classes, mas saber como abstrair e distribuir o código nas abstrações. Por isso padrões devem ser estudados, por exemplo. Aproveitando-se a experiência com a adoção do que a comunidade oferece.

Começaremos por focalizar o problema em pauta nos perguntando:

"O que está envolvido no contexto de solução e abstração de um problema?"

A resposta provavelmente cairá classes em uma das cinco categorias seguintes de elementos:

- **Tangíveis**: Estes são os objetos mais fáceis de serem achados. Dado o problema apropriado, não poderíamos deixar escapar um objeto tais como: Avião, Pessoas, Empresas, etc ...;

- **Funções**: Esta categoria é melhor descrita através de exemplos: Médico, Paciente, Enfermeira, Corretor, Cliente, Empregado, Departamento, etc... Frequentemente, se tivermos um objeto-função, teremos outros também. Isto aparece quando os objetos estão sendo usados para fazer distinção entre funções diferentes desempenhadas pelos mesmos ou por diferentes indivíduos. Como exemplo: podemos esperar que para um hospital, poderíamos encontrar os objetos-funções paciente, enfermeira e médico. Este conjunto de objetos poderia descrever uma enfermeira que fosse atualmente paciente do hospital;

- **Incidentes**: Objetos Incidentes são usados para representar uma ocorrência ou um evento: algo que acontece num determinado período. Alguns objetos incidentes são: Vôo, Acidente, Apresentação, Evento, Falha de sistema, Chamada de serviços, etc...;

- **Interações**: Objetos interações geralmente possuem uma qualidade de "transação" ou de "contrato" e referem-se a dois ou mais outros objetos do modelo. Exemplos são: Compra (refere-se a um comprador, vendedor e objeto de compra e venda) e Casamento (refere-se a pessoas efetuando uma união);

- **Especificações**: Objetos específicos aparecem frequentemente quando tem-se um objeto condição e possivelmente outro objeto valores. Por exemplo: O que significa ser um modelo 172? Ser um modelo 172 significa ser um veículo com as características específicas de possuir acentos reclináveis, quatro portas, etc... A especificação pode ser uma classe de extensão de outra classe, por exemplo: Modelo de um carro e Especificação que pode ser aplicada a um modelo (separadamente).

Quando as características são simplesmente valores diferente entre objetos da mesma classe, não se configura um caso de especificação separada.

Objetos entidades x objetos valor

Identificando os objetos como algo do mundo real que pode ser representado computacionalmente, podemos subcategorizar algums objetos como:

Entidade: Uma entidade é um objeto que possui uma identidade única, independentemente de seus atributos mudarem ao longo do tempo.

- Identificador: normalmente representado por um atributo imutável (ex.: CPF, matrícula, código de produto).
- Persistência no tempo: a entidade continua sendo "a mesma" mesmo que suas características mudem.

Exemplo:

- Um aluno da universidade: se mudar de curso ou de endereço, continua sendo a mesma pessoa, identificada por sua matrícula.
- Um cliente em um sistema bancário: seu nome pode ser atualizado, mas o identificador da conta ou CPF mantém sua identidade.

Já, um **objeto-valor** é um objeto que não tem identidade própria. Ele existe para representar uma característica, atributo ou conceito que descreve uma entidade. Objetos-valor são geralmente imutáveis, para garantir consistência.

Dois objetos-valor são iguais se todos os seus atributos forem iguais. Não se trata de avaliação de referência. Diferem do objeto identidade que iguala objetos pelo seu identificador.

Exemplo:

- Um endereço (rua, número, cidade).
- Uma medida (altura, peso, temperatura).
- Uma data (dia, mês, ano).

Classes anêmicas ou amebas

O **modelo de domínio anêmico** ocorre quando uma classe possui apenas atributos e métodos de acesso, sem encapsular nenhuma lógica ou comportamento. Gera objetos "meros recipientes de dados", transferindo toda a lógica para serviços externos.

A consequência está na dificuldade de manutenção, aumento de acoplamento e redução de coesão.

Começamos aqui relacionar a criação de classes e objetos baseando-se em padrões que tem relações com boas práticas de programação. Eles serão citados e exemplificados ao longo do conteúdo.

Descrição e documentação de lasses e objetos

A descrição do objeto é uma breve declaração informativa que nos permite determinar se uma entidade do mundo real é ou não uma instância de uma classe específica. Essa descrição deve conter características essenciais que definem a classe e permitem a identificação clara de suas instâncias.

Imagine um sistema que gerencia diferentes tipos de veículos. Uma **Classe VeiculoTerrestre** representa veículos que se deslocam sobre a superfície terrestre. Temos as seguintes instâncias potenciais, objetos: carro, motocicleta e bicicleta. Ao descrevermos a classe, identificamos que um **VeiculoTerrestre** é um meio de transporte que possui rodas e é capaz de se mover sobre estradas ou trilhas. Além disso, geralmente é equipado com um sistema de direção e pode ser motorizado ou não.

Consideramos que um barco nitidamente não é uma instância da classe.

Documentar o código ou usar ferramentas de documentação das classes na codificação é essencial tal qual observado neste tópico. Não apenas a descrição da classe, mas de todo o seu conteúdo.

Conforme aprendermos a hierarquia de construção de interfaces, classes abstratas e classes concretas, devemos prestar atenção no contexto de suas nomenclaturas.

Nomenclatura de Objetos

Uma boa escolha dos nomes para os objetos contribuirá significativamente para a legibilidade e compreensibilidade do modelo de informação. Prefira nomes que sejam claros, diretos e verdadeiros embora isso nem sempre seja fácil. Considere a hierarquia das classes para nomear um objeto tanto em relação ao nível que ela se encontra como para os que possam surgir a partir dela.

Além desta representação é importante seguir uma notação padronizada para as declarações. Neste caso atual, sugere-se a adoção da **cammelCase** **UpperCase** para classes e **lowerCase** para instâncias e variáveis.

Classes

O primeiro elemento chave da programação OO é o conceito de classe.

Uma classe pode ser definida como a descrição abstrata de um grupo de objetos, cada um deles com um estado interno específico, mas todos capazes de executar as mesmas operações, pois são construídos através do mesmo "molde" (a classe).

É a partir de uma classe que criamos o objeto. Um objeto é uma instância de uma classe. É referenciado e criado, então construído a partir de uma nova instanciação de uma classe.

Uma classe é um conjunto de atributos e métodos e pode conter uma classe pai herdando dela ou servir de base para a derivação de uma nova classe. Para uma classe pai, usamos o termo **super** classe. Para o caso de derivação, teremos uma subclasse.

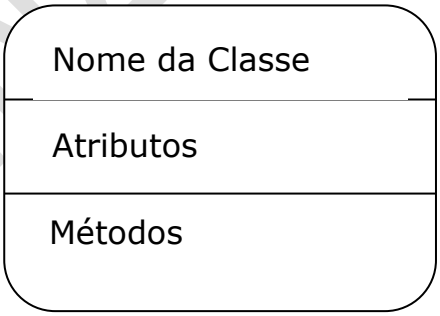
A distinção entre classes e objetos sempre causa dúvidas, mas na verdade é um conceito bastante simples, que aparece em muitas situações do mundo real. Considere, por exemplo, os personagens de animação Procurando Nemo da Pixar: Nemo e Dori. Não há dúvidas de que os indivíduos são diferentes. No entanto, eles possuem muitas características em comum, pois ambos são Peixes. **Peixe** é a classe e **nemo** e **dori** são objetos da classe.

Classes definem tipos de objetos, eles são a representação física das classes. Uma definição de classe é sempre um tipo, uma definição de objeto é sempre uma variável referência daquele tipo classe para com um objeto em memória.

Quando criamos um objeto, dizemos que instanciamos uma classe. Cada instância é uma entidade individual e pode então receber valores para as suas variáveis e modificá-las através de seus métodos.

A grande vantagem de se usar classes é a reutilização do código. Toda a estrutura de um objeto é definida na classe, que depois é utilizada quantas vezes for necessário para criar vários objetos.

Para o projetista de sistemas é útil possuir uma representação gráfica para os elementos que compõe a aplicação. A figura abaixo representa a figura proposta por Coad & Yourdon para representar uma classe. Segundo a notação, toda classe possui atributos e serviços. Cada atributo está associado a uma ou mais variáveis de instância. Um serviço representa alguma operação que pode ser solicitada ao objeto. Cada serviço corresponde a um método.



Simbologia de uma Classe

Abaixo descrevo um conceito de classes e objetos:

Classe: Estrutura de programação que descreve uma abstração através de suas propriedades (atributos) e métodos. Por essa definição vemos que uma classe é apenas um gabarito onde vão estar definidos todos os valores e procedimentos que um objeto dessa classe poderá ter. Em outras palavras, uma classe é uma abstração do objeto e não o próprio objeto, da mesma maneira que uma planta baixa é uma abstração de uma casa e não a própria casa.

Objeto: Elemento criado de acordo com as definições de uma classe. Enquanto que uma classe contém instruções genéricas para construção de objetos, um objeto é um elemento individualizado, um produto dessa classe. Todo o comportamento de um objeto é descrito através das rotinas (ou métodos) que manipulam seus dados (ou atributos) e o estado corrente do objeto está em seus próprios dados encontrados na memória.

Cada objeto, portanto, tem suas próprias características, de acordo com o tipo de operação que virá executar. Podemos definir objeto também como um empacotamento de dados e rotinas preparados para executar determinadas funções.

O objeto é uma instância da classe na memória do computador.

A seguir um exemplo sintático de linguagem que estudaremos criando um objeto a partir de uma classe.

```
NomeClasse nomeObjeto;
nomeObjeto = new NomeClasse ();
```

nomeObjeto	Nome do objeto, um "ponteiro" ou referência a ele.
=	Operador de atribuição.
new	Uma palavra reservada que invoca a construção de Objeto.
NomeClasse	O construtor de uma classe (padrão);
()	Uma indicação de assinatura (vazia) de método construtor;
;	Encerramento de uma instrução.

Mais especificadamente, outro exemplo:

```
class Exemplo { //A classe
}
//Na aplicação...
Exemplo objetoExemplo = null;      //Uma referência sem apontamento
objetoExemplo = new Exemplo();     //Instanciando e apontando para o
                                   //o objeto
```

Atributos

São as estruturas de dados que caracterizam um objeto. Os dados que compõem um objeto determinam o seu estado interno e seguem o padrão da linguagem utilizada, ou seja: Possuem um tipo (Numérico, alfabético, lógico, etc.) e também podem ser além de tipos primitivos, referências para classes (associações).

Operações ou métodos

As operações compõem a interface da classe. A terminologia usada para designá-las é bastante variada: funções membro, métodos, operações, etc. Quando uma função membro é chamada, se diz que o objeto está recebendo uma mensagem (para executar uma ação).

Para introduzir a teoria, vamos analisar um objeto do mundo real: um automóvel.

Um automóvel é um **objeto** e, como tal, possui características e ações que lhe são próprias. Entre as características, que chamaremos **propriedades**, podemos citar:

COR, ***TIPO DO COMBUSTÍVEL***, ***MARCA***, ***ANO DE FABRICAÇÃO***, etc...

Entre as ações peculiares de um automóvel, que chamaremos **métodos**, podemos destacar:

ACELERAR, ***FREAR***, ***LIGAR OS FARÓIS***, ***ACENDER AS LUZES DO FREIO***, etc..

Entre os métodos acima, alguns necessitam de um estímulo externo (uma intervenção do usuário) para serem executados, a esse estímulo chamaremos de **evento**. Vejamos alguns exemplos de eventos:

Pisar no acelerador, ***Girar*** a chave na ignição, ***Colocar*** a chave na ignição, ***Abrir*** a porta, etc...

Eventos: São todas as ações que o usuário pode realizar com o objeto (um clique do mouse, o pressionamento de uma tecla, etc). O conceito de evento surgiu com o advento das interfaces gráficas na qual o usuário tem à sua frente uma tela com vários objetos de interface como combo box, push bottons, etc..., podendo realizar qualquer ação (clique, duplo clique, drag-and-drop, etc.) sobre eles sem uma ordem específica. O objeto deve então executar a ação apropriada.

Não daremos tratativas a eventos. Os eventos serão mensagens enviadas aos objetos e não ações automáticas disparadas por gatilhos na aplicação.

Os métodos e eventos podem provocar alterações no **estado** interno do objeto. O estado do objeto é indicado pelo conjunto de seus atributos. Por exemplo, quando damos partida no automóvel mudamos o seu estado de desligado para ligado.

Classes em Java

Para criar uma classe em Java, usamos a palavra reservada **class** e dentro de um bloco declaramos os atributos e os métodos. Os atributos são como declarações de variáveis e os métodos como declarações de procedimentos e funções que possuem retorno e assinatura (lista de argumentos).

Uma referência é uma variável do tipo classe. Quando apenas referência, ela é o pode ter atribuído a ela a palavra reservada **null**.

Para que uma classe seja instanciada, usamos a palavra **new**, seguida do construtor da classe, o próprio nome da classe seguido de ();

Para acionar atributos e métodos de um objeto, utilizamos a notação ponto ".", nomeDoObjeto.atributo ou nomeDoObjeto.funcao();

Exemplo:

```
class Aluno {
          //atributos que definirão o estado interno do objeto
          String matricula = "";
          //método do tipo get
          String getMatricula (){
               return matricula;
          }
}
```

Lá na aplicação:

```
    //declarando variáveis, como referências/ponteiros null
    Aluno aluno1, aluno2;
    //instanciando a classe no objeto com construtor padrão
    aluno1 = new Aluno();
    //manipulando o objeto pelo estado interno diretamente
    aluno1.matricula = "123456"
    //mandando uma mensagem para o objeto
    System.out.println
          ("Matricula: %s", aluno1.getMatricula());// ou a1.matricula;
```

Uma classe e o Objeto

Antes de se definir um objeto, é necessário que seus componentes (atributos e métodos – variáveis membro e funções membro) sejam especificados.

Uma classe (simples) em Java é declarada utilizando a palavra reservada **class** da seguinte maneira numa notação cammelCase **UpperCase** para o nome da classe e um bloco da classe com seu conteúdo na seguinte ordem.

Lembre-se, para Java a classe pública de aplicação que contém o método main() deve ter o mesmo nome de Arquivo.Java da classe.

Várias classes podem ser desenhadas no mesmo arquivo, mas apenas uma poderá ser pública. Os arquivos .class serão criados separadamente para cada classe.

```
class <NomeDaClasse>  {

     <1-atributos com tipo e declaração com ou sem inicialização>;
     <2-comportamento ou funções membro>;
          <-Construtores não possuem retorno e possuem nome da classe e
            assinatura>;
          <-void são procedimentos, funções tem tipagem e return>;
}
```

Por hora, precisamos apenas entender que atributos são como declaração de variáveis num programa qualquer estruturado, como fizemos em função main.

Não precisaremos de construtores extras, pois Java implicitamente fornece um com o **nome da classe seguido de ();.**

Basta dar um "**new**" e atribuir o resultado a uma variável do tipo da classe. Exemplo:

```
     //Criando a referência para a classe, que tem valor null
     <NomeClasse> variavel; //é null

     //Instanciandoo objeto na referência com new e o construtor padrão
     variavel = new <NomeClasse>(); //apontará para um endereço de obj
```

Os métodos são criados como funções, tendo ou não um retorno, usando void para ser um procedimento ou um tipo qualquer de retorno e a palavra **return** para retornar o valor (exceto num método void).

A assinatura, passagem de parâmetros ou argumentos pode ser vazia ou declarativa como uma lista de declaração de variáveis, com seu tipo e identificador, separados por ",". Usando a notação já aderida até aqui: cammelCase.

Exemplo:

```
//A assinatura pode ser vazia ou com argumentos
<tipo-retorno ou void> nomeMetodo (<tipo> nomeArgumento, ...) {

     return <valor-tipo-retorno>
     // exceto se retorno void, não há retorno com return
}
```

Métodos devem representar comportamentos, e não apenas transporte de dados.

Quando enviamos uma mensagem a um objeto — isto é, quando chamamos um de seus métodos — essa mensagem precisa ser representativa de uma ação do próprio objeto. Em outras palavras, o método deve expressar algo que o objeto faz ou sabe fazer, e não simplesmente expor seus atributos.

Métodos que apenas fazem perguntas sobre o estado interno violam o princípio do Tell, Don't Ask (não pergunte, diga), pois transferem a lógica para fora do objeto. Uma exceção aceitável ocorre em métodos booleanos semânticos, como ehMaiorDeIdade() ou estaDisponivel(), que traduzem verificações próprias da abstração.

Os métodos devem estar sempre alinhados à abstração e responsabilidade da classe. Se um comportamento não pertence à natureza do objeto, ele não deveria estar ali. Caso contrário, cria-se acoplamento desnecessário, o que reduz coesão, diminui reuso e aumenta dependências.

Diálogo com padrões e princípios

Ao longo do tempo de aprendizado estaremos observando padrões e técnicas de boas práticas de programação, tais como:

- **Object Calisthenics**: "Exponha comportamento, não estado". Métodos bem projetados tornam os objetos vivos, não meros recipientes de dados;

- **DDD** (Domain-Driven Design): comportamentos de negócio devem estar próximos dos dados de negócio, respeitando o Bounded Context e evitando modelos anêmicos;

- **Clean Code**: métodos claros e orientados à ação tornam o código mais expressivo, legível e de fácil manutenção;

- **SOLID**: reforça e amarra todos esses pontos:

  - SRP: métodos devem refletir apenas responsabilidades próprias da classe;
  - OCP: abstrações bem definidas permitem extensão sem modificação;
  - LSP: respeitar a abstração garante substituição correta entre classes;
  - ISP: interfaces com métodos coerentes evitam dependências forçadas;
  - DIP: depender de abstrações permite que o comportamento seja flexível e testável.

Exemplo:

```
class Aluno { //Entidade
          //atributo
          String matricula = ""; // o que identifica o aluno
          String nome;


          //método com argumentos e retorno
          double calcularMediaAluno (double nota1, double nota2){
                double soma = nota1 + nota2;
                soma /=2;
                return soma;
          }
}
```

```
public class App {
    public static void main(String[] args) throws Exception {

        Aluno a1, a2;
        a1 = new Aluno();
        System.out.println (a1);
        a2 = new Aluno();
        System.out.println (a2);

        a1.matricula = "100";
        a2.matricula = "200";

        System.out.println (a1.matricula +  "\n" + a2.matricula);

        System.out.printf ("Média do aluno a1 %.2f\n\n",
                           a1.calcularMediaAluno (5.0, 6.5));

        a2 = a1; //aqui estamos apontando as duas referências ao mesmo obj
        System.out.println (a1);
        System.out.println (a2);

    }
}
```

Construtores

Os construtores são semelhantes a métodos, mas não podem ser chamados diretamente. São usados para inicializar campos dos objetos da classe. Podem ser sobrecarregados e não são considerados membros da classe. A sua declaração é semelhante a uma declaração de método com as restrições:

- **Nome igual** ao nome da classe;
- Nenhum tipo de retorno (nem mesmo void).

Um construtor padrão é sempre provido pelo compilador. Este não possui parâmetros e inicializa membros de acordo com o padrão estabelecido do Java. A partir da definição de um construtor pelo criador da classe, o construtor padrão se torna indisponível.

A invocação do construtor dá-se automaticamente durante a criação, construção, do objeto a partir de uma classe, uma referência para a mesma e o uso de new.

A seguir uma série de exemplos de classe e construtores:

```
//Não redefinindo construtor, o padrão é implícito
class ClasseExemplo1 {
        int i;
        void metodoUnico()            {
        };
}
```

Na App:

```
ClasseExemplo1 referencia1 = new ClasseExemplo1();
```

```
//Redefinindo construtor o padrão
class ClasseExemplo2 {
        int i;
        ClasseExemplo2(){
             i = 0; //inicializando os valores internos
        }
        void metodoUnico() {

        };
}
```

Na App:

```
ClasseExemplo2 referencia2 = new ClasseExemplo2(); //padrão?
```

```
//Redefinindo construtor o padrão e ainda criando um novo
class ClasseExemplo3 {
        int i;
        ClasseExemplo3(){
                i = 0; //inicializando os valores internos
        }
        ClasseExemplo3(int i){
                this();
                this.i = i; //inicializando via argumentos
        }
        void metodoUnico()              {
        };
}
```

Na App:

```
ClasseExemplo3 referencia3 = new ClasseExemplo3(10);
```

A referência this pode ser utilizado para acessar um atributo de uma classe que tenha sido ocultado dentro de uma função-membro.

Explorando a referência this

Cada objeto de uma classe tem sua própria cópia de cada atributo não estático que compõe a classe. Os métodos, por outro lado, são compartilhados por todos os objetos da classe. Isto quer dizer, por exemplo, que dada a classe a seguir:

```
class A {
  int x; //atributo de escopo da classe/objeto
  int mX (void) {
        return x;
  }
}
```

Se a1 e a2 forem objetos desta classe, a1.x refere-se ao membro x do objeto a1 e a2.x refere-se ao membro x de a2.

Por outro lado, a1.mX() e a2.mX() referem-se ao mesmo método mX(), mas este método, por sua vez, acessa o membro x de cada objeto que chama o método. O estado interno de cada objeto.

Então, a pergunta que se faz é: Que mecanismo é utilizado para fazer com que o método saiba a que objeto o membro x pertence? A resposta para esta questão está no fato de o compilador acrescentar a cada método (não estático) um argumento adicional que é uma referência para um objeto da classe em questão. Então, quando um método é chamado (utilizando um objeto), a referência do objeto é utilizada para acessar os campos do referido objeto.

Algumas vezes, é necessário fazer referência ao objeto que chama um determinado método. Neste caso, utiliza-se a palavra reservada **this** que representa uma referência para o objeto utilizado para chamar o método.

Podemos dizer que, o construtor se não fosse automatizado seria um método que retornaria o tipo classe e seu return seria o objeto: this. Num exemplo hipotético* deste pensamento temos uma imaginação sobre a classe Aluno apresentada:

```
Aluno Aluno(){ //class NomeClass, que é o mesmo
     //instancia a classe e...
     return this;
}
```

**Cuidado, este código é explicativo e não representativo.*

A referência this pode ser acessada apenas no corpo de métodos não-estáticos de uma classe.

Como vimos, a referência this pode ser utilizada para acessar um atributo de uma classe que tenha sido ocultado dentro de uma função-membro. Por exemplo, no segmento a seguir:

```
class C {
   int x;
   void metodo();
   {
        int x; // Esta definição oculta o atributo x acima
        ...
        x = 5; // Refere-se à variável local x
        this.x = 10; // Refere-se ao membro, atributo x
        ...
   }    ...}
```

O uso de this na definição de metodo() serve para especificar o acesso ao membro x que foi ocultado pela variável local x. Em situações normais, qualificar um membro com this é redundante.

Outro exemplo pode ser visto a seguir:

```
class Exemplo {
        long x, y;
        //o construtor que recebe inicializações
        Exemplo (int x, int y);    {
             this.x = x;
             this.y = y;
        }
}
```

A referência **this** pode também ser usado para invocar um construtor de dentro de outro construtor da mesma classe. Neste caso, esta chamada deve ser a primeira instrução do construtor que faz a chamada do outro construtor. Exemplo:

```
class UmaClasse {
   public UmaClasse() {...}     // Construtor padrão feito a mão

   public UmaClasse(int x, double y)  // Outro construtor
   {
      this();  // Chama o construtor padrão da classe
      ...
   }
}
```

Acesso a Membros de um objeto

O acesso aos membros de dados é feito utilizando a notação ponto, mesma ideia utilizada por linguagens para acesso a membros de um tipo struct. Somente o que for público ou que pertencer ao mesmo pacote ou padrão Java pode ser acessado via objeto. A visibilidade será tratada no encapsulamento. Funções-membro podem também ser acessadas (chamadas) desta maneira, dizendo-se enviando uma mensagem para o objeto.

Exemplo:

```
// Define "classeObjeto" como uma referência de "Classe"
// para um objeto da classe Classe
Classe classeObjeto = new Classe ();

// Acessa (chama) o método m1() usando classeObjeto
classeObjeto.m1();
```

Destruição e Finalização de Objetos (Destrutores)

A liberação de memória ocupada por objetos é feita automaticamente pelo mecanismo de garbage collector de Java. O mecanismo é automático e libera espaço quando acha necessário. Pode-se **sugerir** que o mecanismo atue usando System.gc().

O mecanismo sempre chama o método finalize() da classe a que o objeto pertence antes de liberar a memória ocupada pelo objeto. Normalmente, o método finalize() é declarado como protected da seguinte maneira:

```
protected void finalize() {
    //O que você quer que aconteça antes de um objeto
    //deixar de existir - for coletado
    }
```

Este método está obsoleto desde Java 9.

Você pode tornar o objeto elegível para remoção, atribuindo a ele null. Em Java moderno, nos preocupamos mais em **liberar referências** do que em destruir objetos. O GC faz isso por nós.

Classes sem Instância

Às vezes não faz sentido permitir que uma classe tenha instâncias. Por exemplo, não faz sentido a existência de instâncias da classe **Math** pois os métodos nesta são do tipo static. Pode-se impedir que uma classe tenha qualquer instância declarando um construtor default como sendo private.

Apresentação de abstrações da linguagem Java

O pacote java.lang

O pacote lang é um pacote auto importado para o seu programa, o único que possui esta característica, não tendo necessidade de import. É implícito.

Na documentação da linguagem é possível e importante conhecer o seu conteúdo:

https://docs.oracle.com/javase%2F8%2Fdocs%2Fapi%2F%2F/java/lang/package-summary.html

A classe String

As cadeias de caracteres em Java são criadas a partir de instanciação implícita da classe String. Strings são constantes, por isso a cada manipulação uma nova referência é criada e devolvida.

A classe Integer

Apesar dos inteiros serem tipos primitivos a partir de int, a classe Integer representa os tipos inteiros e sua manipulação em Java. A representação similar para todos os tipos primitivos.

O pacote utils e a classe Array

O pacote utils precisa ser "importado" para ser usado. Ele é um pacote com classes utilitárias, com exemplo Arrays.

Os arranjos em Java são implicitamente criados particularmente em Java. Apesar de Arrays representa-los, não é possível instanciar a classe, apenas utilizar a mesma para manipulação de objetos do tipo arranjos como Object em java.

Parte III - UML

Para ilustrar essas diferenças de forma clara, vamos criar um diagrama UML que inclui um exemplo de cada um desses tipos de relação.

UML, ou Linguagem de Modelagem Unificada (do inglês, Unified Modeling Language), é uma linguagem padrão de modelagem de sistemas que proporciona uma maneira sistemática de visualizar a estrutura, o design e o comportamento de um sistema. Ela é amplamente utilizada em engenharia de software para documentar, explorar, e comunicar o design de sistemas de software, bem como para a modelagem de processos de negócios e outras aplicações não software.

A UML oferece um conjunto de diagramas que abrangem diferentes aspectos de um sistema:

Diagramas estruturais, como diagramas de classes, de objetos, de componentes, e de implantação, que focam na estrutura do sistema e suas partes.

Diagramas comportamentais, como diagramas de casos de uso, de atividades, de estados, e de sequência, que descrevem o comportamento e a interação entre os componentes do sistema.

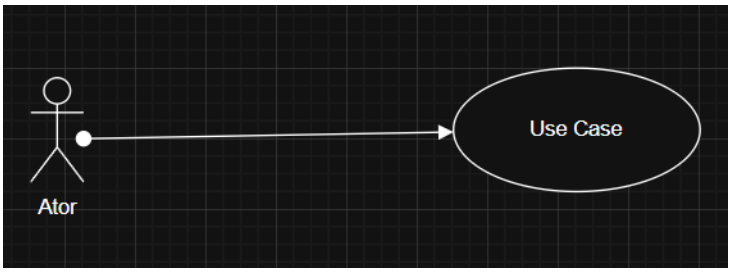
A UML permite que desenvolvedores visualizem complexidades e planejem de forma eficaz, promovendo uma compreensão clara do sistema tanto para os stakeholders técnicos quanto para os não técnicos.

Casos de uso

Um caso de uso descreve uma ação ou funcionalidade do sistema, do ponto de vista do usuário. Ele responde: "O que o sistema deve fazer?". Ajuda a entender quem usa o sistema e para quê.

Seus elementos são:

- **Atores** (Usuários ou sistemas externos) – figuras palito;
- **Casos de uso** – elipses com o nome da ação ou macro ações (conjunto de ações);
- **Sistema** – uma caixa que envolve os casos de uso.

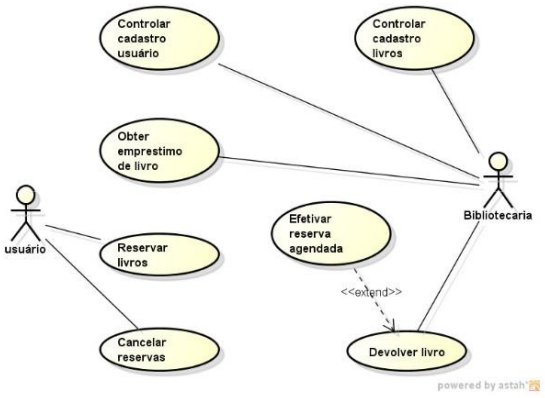


A sua construção baseia-se em resenhar os atores com nomes reais (aluno, professor, usuário, etc, administrador) e descrever as ações como verbos de ação (Enviar pedido, Emitir relatório). Usam-se setas de um ponto de partida apontando para o caso de uso, demonstrando quem executa a ação.

As setas podem conter rótulos de relacionamento que são divididos em:

- **<include>**: Quando o caso de uso sempre chama outro caso de uso porque é necessário para funcionar;
- **<extends>**: Quando o caso de uso pode ser estendido para outro caso de uso, em situações específicas ou pontuais/opcionais.

Exemplo de um caso de uso.



Quando você migra do **diagrama de casos de uso** para o **diagrama de classes**, essa elipse normalmente não vira uma **classe** por si só.

O que acontece é que:

- O **caso de uso** serve como **entrada** para identificar quais **responsabilidades** e **operações** precisarão existir nas classes;
- Em termos práticos, o nome ou a ação descrita na elipse geralmente se torna:
    1. **Um ou mais métodos** (operações) dentro de classes, responsáveis por implementar essa funcionalidade;
    2. Eventualmente, um conjunto de interações entre várias classes (diagramas de sequência ou comunicação), não apenas um método isolado – *não abordados nesta disciplina.*
- Raramente um caso de uso vira diretamente uma classe, a menos que se descubra que essa funcionalidade é um **objeto conceitual autônomo**.

O caso de uso (elipse) inspira e define **métodos** ou **conjuntos de métodos** nas classes que irão implementar a funcionalidade descrita.

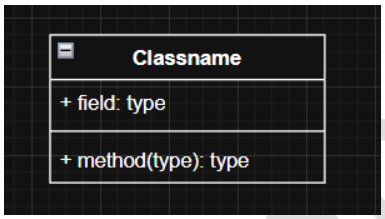
Para exercitar, vamos construir um caso de uso para um sistema. Utilizaremos o https://app.diagrams.net/.

Diagrama de Classes

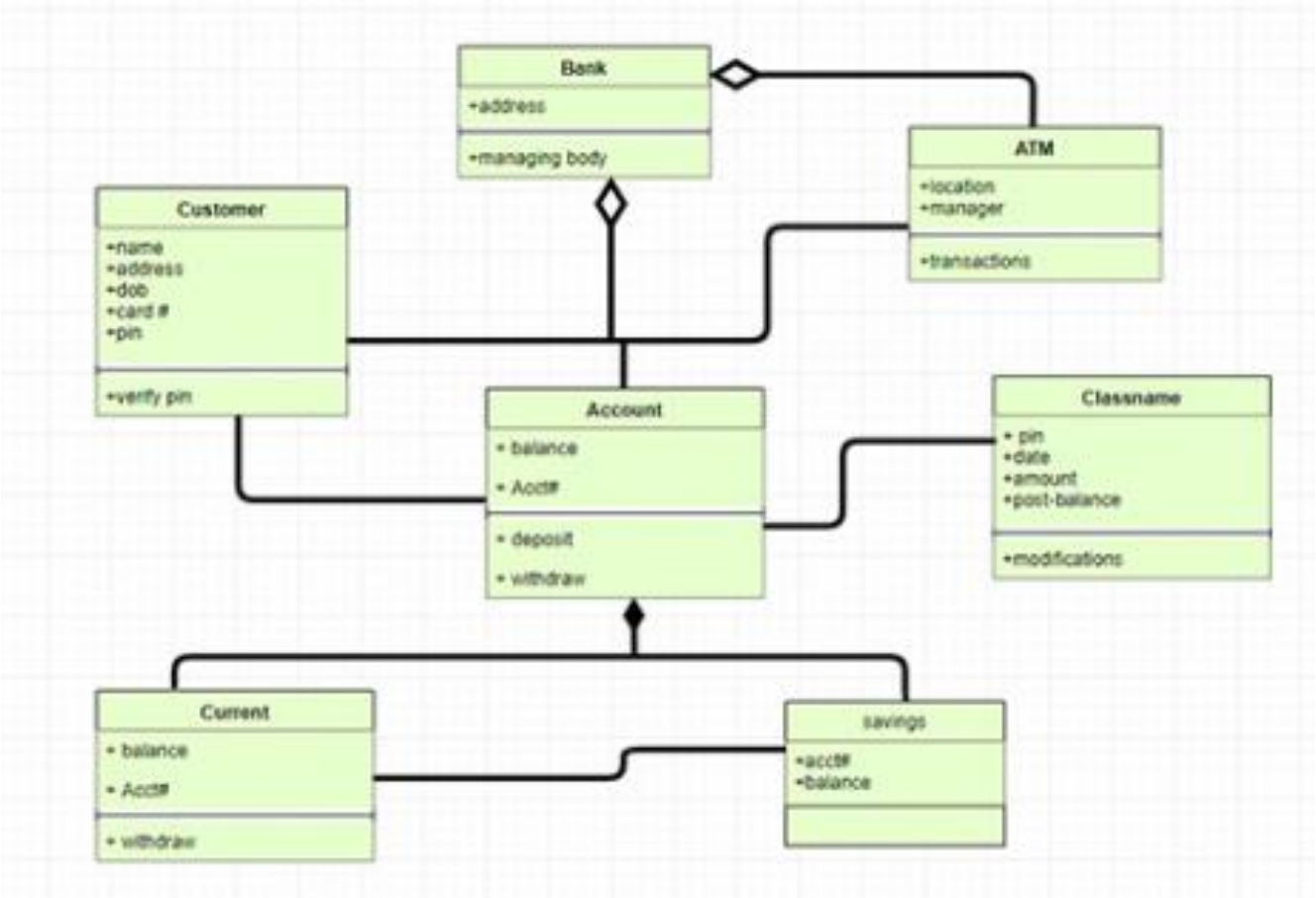
O diagrama de classes é o modelo que representa a estrutura das classes do sistema, seus atributos, métodos e relações. É a ponte entre o mundo real (modelagem) e o código (implementação).

Neste diagrama, a classe é um retângulo dividido em 3 partes. Seu Nome, atributos e comportamento.

Os atributos trazem informações de seus tipos. Seu comportamento traz as assinaturas e retornos. A visibilidade pode ser incluída para atributos e comportamento (privado (-) ou público (+)).



O relacionamento é destacado, onde classes se relacionam umas com as outras. Um exemplo de um diagrama de classes é apresentado a seguir:



Relações entre classes

As classes se relacionam. Uma ou mais instâncias de uma classe estão como atributos da classe. Desta forma, o relacionamento entre as classes pode ser assim definido:

Associação (PODE TER)

Associação é o relacionamento mais básico entre objetos. Ela estabelece uma conexão entre duas classes independentes onde os objetos de uma classe podem conhecer os objetos da outra. A associação pode ser unidirecional, onde uma classe conhece a outra. A possibilidade de associação bidirecional é possível, ambas as classes conhecem uma à outra. Ex: Professor e Aluno. O aluno tem seu Professor. Desta forma temos uma associação unidirecional. Se pensarmos então que, o Professor também terá seus Alunos, estamos criando uma associação bidirecional.

Na **UML**, uma linha simples conecta as classes envolvidas na associação. Setas podem ser usadas para indicar a direção da relação (unidirecional ou bidirecional).

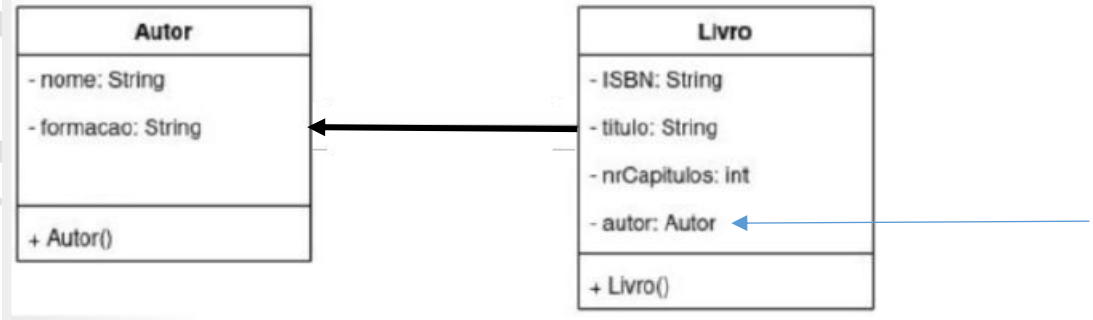
Símbolos de Cardinalidade: Podem ser adicionados perto das classes para indicar o número de instâncias que podem participar da associação.

A associação é implementada na forma de que uma classe se torna um atributo de outra classe.

```java
public class Autor {
    String nome;
    String formacao;

}
```

```java
public class Livro{
    String ISBN;
    String titulo;
    int nrCapitulos;
    Autor autor;
}
```



Agregação (PODE TER UM ou MAIS)

Agregação é um tipo especial de associação que representa um relacionamento "tem-um" ou "tem-muitos", onde o objeto contêiner (ou todo) pode conter outros objetos (partes), mas as partes não dependem do contêiner para existir. Ou seja, a existência das partes não é estritamente dependente do todo.

Exemplo: Em um sistema universitário, uma classe Departamento pode ter vários objetos da classe Professor. Os professores podem existir sem o departamento, o que significa que se o departamento for destruído, os professores não serão necessariamente destruídos.

Relação Fraca: Em termos de ciclo de vida, a agregação representa uma relação fraca entre o agregador e o agregado; a destruição do objeto agregador não implica a destruição dos objetos agregados.

Independência: Os objetos agregados mantêm sua independência e podem existir sem o objeto agregador.

Cardinalidade: A agregação pode ter cardinalidade um para muitos ou muitos para muitos, refletindo quantos objetos podem ser associados nesse tipo de relação.

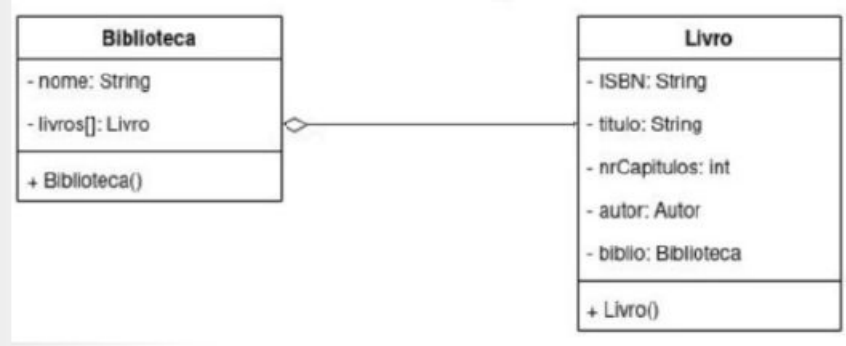
Nas linguagens de programação, não há distinção de código para sua representação. Tanto a associação como agregação são criadas da mesma forma. Na UML, há distinção a ser observada nos seus diagramas específicos.

Similar à associação, mas com um pequeno diamante vazio na extremidade da linha junto à classe "todo" que contém a classe "parte".

Símbolos de Cardinalidade: Também podem ser utilizados para mostrar quantas partes podem ser associadas ao todo.

```java
public class Biblioteca{
    String nome;
    Livro livros[];
}
```

```java
public class Livro{
    String ISBN;
    String titulo;
    int nrCapitulos;
    Autor autor;
    Biblioteca biblioteca;
}
```



Composição (TEM)

A composição é uma forma mais forte de agregação onde as partes não podem existir sem o todo. Se o objeto contêiner (todo) for destruído, então todas as suas partes também serão. Isso estabelece uma relação de dependência mais forte entre o todo e suas partes.

Exemplo: Considere um sistema de gestão de projetos onde uma classe Projeto contém objetos da classe Tarefa. Se o projeto for encerrado (destruído), todas as suas tarefas associadas também devem ser encerradas (destruídas), pois não faz sentido que as tarefas existam sem o projeto.

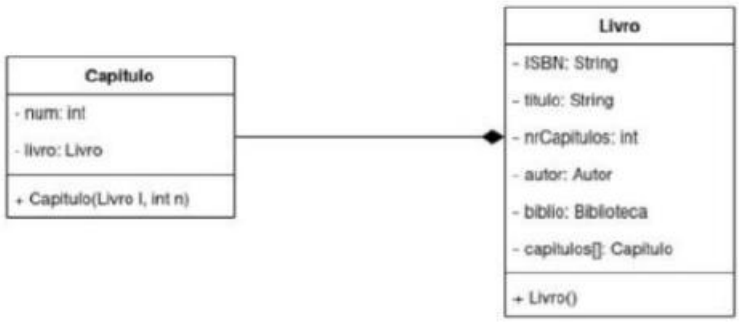
Utiliza uma linha com um diamante preenchido na extremidade junto à classe "todo", indicando uma relação mais forte que a agregação.

Símbolos de Cardinalidade: São usados para indicar o número de partes que o todo pode conter.

```java
public class Capitulo{
    int num;
    Livro livro;

    Capitulo (Livro livro, int numero){
        this.num = num;
        this.livro = livro;
    }
}
```

```java
public class Livro {
    String ISBN;
    String titulo;
    int nrCapitulos;
    Autor autor;
    Biblioteca biblioteca;
    Capitulo capitulos[];
}
```



A composição é utilizada como método de injeção de objetos frente ao pilar de herança.

Diferenças Chave

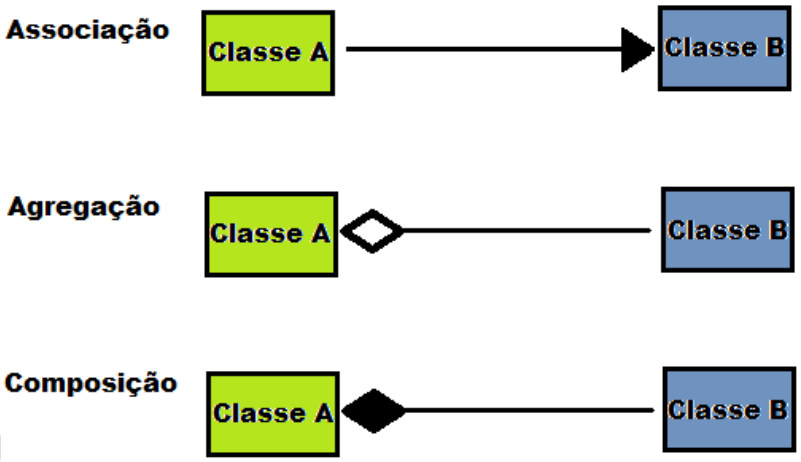
Vida Útil: Em uma composição, as partes não sobrevivem ao todo. Na agregação, as partes podem sobreviver ao todo.

Tipo de Relacionamento: A agregação é um relacionamento "tem-um" ou "tem-muitos" menos estrito, enquanto a composição é um relacionamento "tem-um" ou "tem-muitos" mais estrito e dependente.

Direção: A associação pode ser unidirecional ou bidirecional. Tanto a agregação quanto a composição são tipos específicos de associação, mas com regras mais rígidas sobre a dependência e a vida útil dos objetos envolvidos.

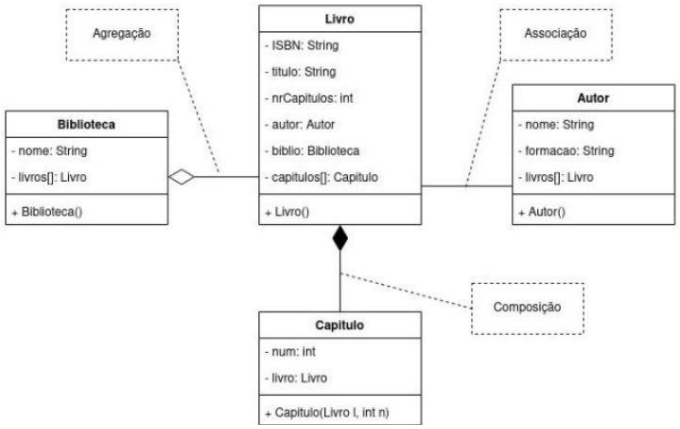
Estes conceitos são cruciais para a modelagem de sistemas complexos em programação orientada a objetos, permitindo uma representação mais precisa e organizada das relações entre diferentes partes de um sistema.

Em resumo:



Na prática:

Os códigos usados para representar a associação, agregação e composição são basicamente os mesmos. O objeto relacionado vira atributo na classe. Em geral, estes relacionamentos são representados da mesma forma em linguagens de programação orientada a Objetos.



Estudo de caso prático 1

Vamos criar um modelo simplificado em Java para representar pessoa jurídica associada a pessoa física, além de uma classe associada para o endereço com ambas as classes citadas. As associações referem-se a um objeto de pessoa física (CEO) com um objeto de pessoa jurídica (empresa) e tanto pessoa física quanto jurídica possuem um endereço.

1. Classe **Endereço**: Responsável por armazenar informações de localização e imprimir o endereço formatado como uma "etiqueta" em monitor;

2. Classe **PessoaFisica**: Representa uma pessoa física com atributos básicos como nome e CPF. Conterá também um endereço;

3. Classe **PessoaJuridica**: Representa uma pessoa jurídica com atributos como nome da empresa e CNPJ. Terá uma associação com PessoaFisica representando o CEO da empresa. Conterá também um endereço;

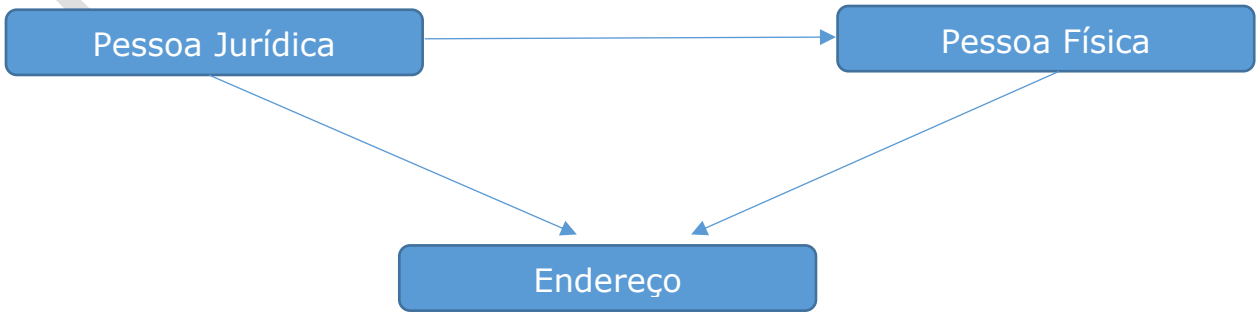
4. Programa principal (onde está o método main) : Criará objetos para alguma grande empresa de software, com seu endereço e seu CEO, também com seu respectivo endereço.

Para o aluno que já está habituado com o encapsulamento em OO, ignorar este pilar de conceito neste momento, não usar modificadores de visibilidade e nem construtores definidos.

Usaremos os padrões Java para a visibilidade e construtor sem especificadores de encapsulamento ou criação de novos construtores.

Não há necessidade de usar dados de entrada pelo usuário na aplicação **AppTester**, mas caso se faça necessário, a classe Teclado será utilizada.

Esboço das associações:



Estudo de caso prático 2

Desenvolver um sistema simplificado de banco em Java para gerenciar contas correntes.

O sistema deverá permitir a criação de duas contas corrente, bem como operações de depósito e saque.

1. **Abertura de Conta**: A conta corrente deverá ter um número. O saldo inicial da conta deverá ser 0. Considerar um limite de crédito para a conta de 30% do salário de seu titular;

2. O titular da conta é uma PessoaFisica, classe desenhada no estudo de caso anterior. Neste caso então titular é uma associação com a conta corrente.

    a. Alguns atributos essenciais para o titular devem ser previstos, ostente modificar a classe original de pessoa física para atender este novo contexto.

3. Operações Financeiras:

    a. **Depósito**: Adicionar um valor ao saldo da conta;

    b. **Saque**: Retirar um valor do saldo da conta após a validação da senha do titular. Não é possível efetuar saques sem saldo ou saque maior que o saldo da conta e o seu limite.

Observações para construção do caso:

- Não adicionar construtores nas classes;
- As operações sobre a conta devem ser realizadas por métodos da classe;
- Não é necessário implementar funcionalidade para emissão de extratos bancários, pois as operações realizadas não são gravadas, não há persistência;
- A passagem de parâmetros em Java é por cópia para tipos primitivos e por referência para objetos;
- No método main() da aplicação, instancie objetos das classes criadas e manipule os objetos para testar sua abstração;
    - Crie um menu de opções, por exemplo;
    - Exiba o saldo atual da conta corrente após a execução de qualquer operação.

Estudo de caso prático 3

Implementar uma classe **VeiculoAutomotor** em Java que modele de maneira simplificada um veículo, controlando seu estado interno como velocidade e quantidade de combustível, e permitindo operações como acelerar, desacelerar e abastecer.

Atributos:

- velocidadeAtual: velocidade atual do veículo em km/h;
- quantidadeCombustivel: quantidade atual de combustível no tanque;
- capacidadeTanque: capacidade total do tanque de combustível em litros;

Ao criar uma instância da classe, use um método inicializador da capacidadeTanque, defina quantidadeCombustivel e velocidadeAtual como zero. Imprima uma mensagem de inicialização – Não use construtor!

Comportamento da abstração:

- acelerar(): aumenta a velocidade em 20 km/h, não ultrapassando o limite de 120 km/h;
- desacelerar(): diminui a velocidade em 20 km/h, garantindo que não fique negativa;
- abastecer(int litros): adiciona combustível ao tanque sem exceder a capacidade total – abastecer é sempre deixar o tanque cheio;
- obterVelocidadeAtual(): retorna a velocidade atual do veículo;
- obterQuantidadeCombustivel(): retorna a quantidade de combustível no tanque;
- darPartida(): permite iniciar o movimento do veículo se houver combustível suficiente;
- desligar(): interrompe o movimento do veículo;
- andar(): simula o veículo andando por uma hora, reduzindo a quantidade de combustível conforme o consumo (suponha um consumo de 16 km/l).

Implemente um menu de operações na sua aplicação que permita ao usuário interagir com o veículo.

Um exercício como exemplo guiado: O jogo de dado de 2 jogadores

Tendo como objetivo criar uma aplicação que crie um jogo de dados para ser jogado por dois jogadores. Os jogadores lançam uma única vez para ver quem ganha ou se empate, até que algum vença, continuam lançando os dados.

Como abstrair?

- Ao invés de começar pensando no fluxo de execução, a ideia do paradigma é traçar as abstrações e a relação entre elas primeiramente, sendo o que sobra por último a aplicação;
    - Horizontalizamos o código vertical do algoritmo em classes;
    - A quebra de código em funções dá lugar ao seu agrupamento em quebra de código por classes entidades ou valor que possuem funções como comportamento;
- O que se pensa em termos de quais classes se fazem necessárias para esta aplicação do jogo e jogadores de dados?
- A lógica de fluxo do jogo está na aplicação ou na abstração do jogo?
- Há relacionamentos entre as abstrações criadas como solução?
- Quando se dará a instanciação de cada abstração? Há necessidade de validação?

1. Identifique as classes de execução da solução;
2. Esboce os atributos e métodos das classes percebendo possíveis "associações";
    a. Pense em métodos que sejam chamadas de funções que digam o que fazer e não que perguntem algo (Tell, dont ask!);
3. Crie a aplicação para fomentar a lógica conforme o enunciado para as abstrações criadas;
    a. A sua aplicação deve ter a compatibilidade do **N**ome da classe x **A**rquivo.java;
    b. Suas classes devem estar separadas, cada qual em arquivos distintos, não havendo necessidade de agrupamento em pacote;
    c. Cada objeto deve ser criado, mantendo coesão em seu estado interno, valores internos válidos.