

Programação Orientada a Objetos com Java



Professora: Jéssica Félix
Abril, 2024

Módulo 1 - Conceitos Básicos de Programação Orientada a Objetos

Introdução

Antes de começar nosso curso de Programação Orientada a Objetos com Java, quero me apresentar e deixar alguns links e dicas que podem facilitar a sua jornada. Sou Jéssica, Engenheira de Software. Já ministrei aulas de carreira na Reprograma (escola de programação para pessoas que se identificam com o gênero feminino, de São Paulo) em 2019, aulas de Javascript no curso de Web Dev na Sirius Educação, em 2022 e, em 2023, publiquei o curso "Descomplicando a Sintaxe do Javascript", na Linux Tips.

Minha motivação é apoiar pessoas que, assim como eu, vem de transição de carreira, a construir bases mais sólidas desde o início do contato com programação, para crescer de forma sustentável na profissão e ter bastante propriedade sobre o próprio trabalho.

- Você pode conhecer mais sobre o conteúdo que eu crio para a comunidade:
 - [As palestras que já apresentei em eventos](#) diversos, nacionais e internacionais.
 - [Ouvir o podcast](#) que eu sou co-host, o Devs na Estrada
 - Email: jessilyneh@gmail.com

Neste curso você vai aprender a trabalhar com orientação a objetos com Java, então é necessário conhecimento prévio em lógica de programação. Vou te mostrar conceitos, exercícios práticos, fornecer materiais de apoio para que você possa ir além do conteúdo que estou apresentando, e que possa aplicar estes conhecimentos para outras linguagens de programação, como Javascript, C++ e Python.

- Principais referências do nosso curso:
 - Livros: [Desbravando Java e Orientação a Objetos](#) e [Orientação a Objetos e SOLID para Ninjas](#) e, embora não seja um livro focado em orientação a objetos, usaremos de referência os primeiros capítulos do livro [Arquitetura Limpaa](#).
 - [Tutoriais de Java do Beginners Book](#). Dica: Você pode traduzir páginas web para português em qualquer navegador. [Veja o tutorial do Canal Tech](#).

É importante lembrar que conseguimos construir conhecimentos em algum assunto com base em prática e repetição. Se não conseguir entender algo de primeira, tente assistir novamente os vídeos, reforçar os conteúdos e principalmente praticar. A prática ajuda a absorver melhor a teoria, mas não deixe de valorizar o conteúdo teórico também! Ele vai te dar repertório no dia a dia de trabalho para discussões e busca de soluções.

👉 Aula 1.1 Paradigmas de programação

Você já sabe a base de programação, já escreveu seus primeiros códigos e, agora, vamos aprender sobre como programas são pensados e construídos, que são os paradigmas de programação. Trazendo uma definição simples,

"Os diferentes modos ou "formas" de programar são conhecidos como paradigmas de programação. Eles determinam a estrutura e o estilo do código em desenvolvimento, relativamente não relacionadas à linguagem."

Imagine que você está aprendendo a cozinhar. Existem diferentes "paradigmas" ou estilos de cozinhar, como a culinária francesa, a italiana, a asiática, etc. Cada uma dessas abordagens tem suas próprias regras, técnicas e formas de organizar os ingredientes e os passos da receita. Da mesma forma, os paradigmas de programação são as diferentes "formas" de escrever e estruturar o código de um programa.

Vamos usar bastante a analogia com a cozinha para entender como funciona a orientação a objetos!

Alguns dos principais paradigmas de programação:

- **Programação Imperativa:** Neste paradigma, o código é escrito como uma série de instruções que alteram o estado do programa.

- **Programação Funcional:** Neste paradigma, o foco está em funções puras, que não alteram o estado do programa, mas sim transformam dados de entrada em dados de saída.
- **Programação Orientada a Objetos (POO):** Neste paradigma, o código é organizado em torno de objetos, que são entidades que possuem propriedades e comportamentos. Este vai ser o tema deste curso.
- **Programação Declarativa:** Neste paradigma, o foco está em descrever o que o programa deve fazer, em vez de especificar como ele deve fazer.

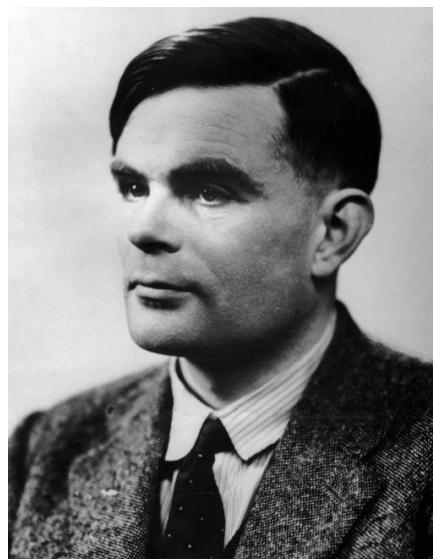
Mas, de onde tudo isso surgiu? E com qual motivação? Vamos começar entendendo como tudo começou:

Linha do tempo - paradigmas de programação

1936

Alan Turing estabelece as bases da programação

Alan Turing e a máquina de Turing



Não vou entrar em detalhes sobre a história completa da computação, mas vou trazer um recorte especial para entendermos a evolução dos paradigmas de programação, começando lá em 1936, quando **Alan Turing** publica o artigo "Sobre Números Computáveis", estabelecendo as bases da teoria da computação. Ele introduz a ideia de uma máquina universal que pode resolver qualquer problema computacional.

Alan Turing foi um renomado matemático e cientista pioneiro no campo da inteligência artificial e da ciência da computação, conhecido como um dos "pais da computação".

Neste artigo, Turing demonstrou que um problema é computável se puder ser resolvido por uma máquina de Turing, um dispositivo teórico que manipula símbolos em uma fita de acordo com regras predefinidas. Essa máquina foi fundamental para a compreensão da computação e estabeleceu as bases para o desenvolvimento da computação moderna.

O artigo de Turing é considerado uma base da computação porque ele mostrou que problemas matemáticos podem ser resolvidos de forma algorítmica, estabelecendo assim os fundamentos teóricos para a computação e a definição do que é computável.

Arquitetura Von Neumann.
Turing propõe os conceitos de laços e atribuições.

1945

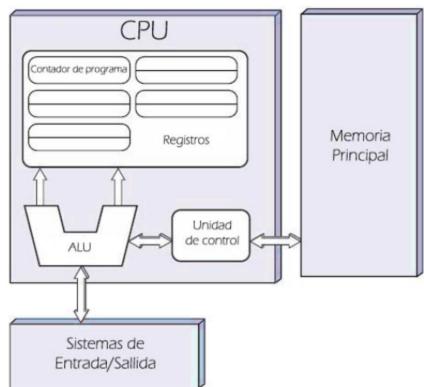
1936

Alan Turing estabelece as bases da programação

Em 1945 Turing propôs o conceito do Automatic Computing Engine (ACE), que incluía características como a capacidade de armazenar tanto dados quanto instruções na memória, e este é considerado um dos primeiros designs detalhados de um computador de programa armazenado.

O design do ACE incluía muitos conceitos que são fundamentais para a programação moderna, como laços e atribuições. No entanto, o ACE completo nunca foi construído durante a vida de Turing. Em vez disso, uma versão simplificada, conhecida como Pilot ACE, foi construída e funcionou pela primeira vez em 1950.

Arquitetura Von Neumann



Neste mesmo ano, John von Neumann propôs um design para computadores que se tornou a base para a maioria dos computadores modernos. Esta arquitetura, conhecida como arquitetura de von Neumann, tem quatro componentes principais: **a unidade de controle, a unidade aritmética e lógica (ALU), a memória e os dispositivos de entrada/saída**. A unidade de controle interpreta as instruções na memória e controla a execução dessas instruções. A ALU realiza operações matemáticas e lógicas. A memória armazena tanto os dados quanto as instruções. Os dispositivos de entrada/saída permitem que o computador se comunique com o mundo exterior.

Resumindo até aqui...

Começamos com as contribuições valiosas de Alan Turing com as bases do que conhecemos como programação, já que Turing foi o primeiro a conceber a ideia de que programas eram apenas dados. Quando Turing começou a escrever seus primeiros programas, é importante reforçar que a linguagem era binária. Já imaginou o trabalho que daria, escrever tudo com 1 e 0?

A criação da arquitetura de Von Neumann foi um outro marco bem importante, permitiu que os programas de computador fossem armazenados na memória, em vez de serem codificados diretamente no hardware. O que isso significa? Que seria mais fácil modificar e executar um programa. Perceba que a cada inovação, a ideia é conseguir facilitar mais e mais a vida de quem desenvolvia os programas para as máquinas.

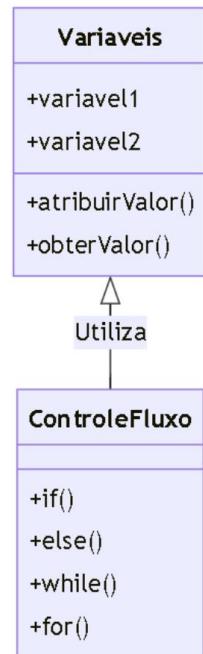
→ Aula 1.2 O paradigma imperativo

Nesta época, o paradigma de programação em uso era o **paradigma imperativo**, baseado no modelo teórico da Máquina de Turing, sendo a base para o desenvolvimento dos primeiros computadores e linguagens de programação.

John von Neumann aproximou o modelo teórico da Máquina de Turing à arquitetura de computadores, contribuindo para a implementação prática desse paradigma

O foco desse paradigma é especificar como um processamento deve ser feito no computador, por meio de uma sequência ordenada de comandos que alteram o estado da memória

Vamos conhecer um exemplo de código que usa o paradigma imperativo. Não se preocupe em entender todas as palavras chave, vamos nos concentrar apenas em compreender o conceito:



```

10 INPUT "Digite um número: ", num
20 IF num < 0 THEN
30   PRINT "O número é negativo"
40   GOTO 60
50 ELSE
60   PRINT "O número é positivo ou zero"
70 END IF
80 PRINT "Vamos contar até 5:"
90 FOR i = 1 TO 5
100 PRINT i
110 NEXT i
120 PRINT "Fim do programa."
130 END

```

Neste exemplo, podemos observar as principais características do paradigma imperativo:

- 1. Atribuição:** Com o uso da instrução INPUT para atribuir um valor digitado pelo usuário à variável num, na linha 10
- 2. Estruturas de controle:** As linhas 20-70 demonstram uma estrutura de controle IF-THEN-ELSE, que verifica se o número é negativo e imprime a mensagem correspondente.
- 3. Estruturas de repetição:** As linhas 90-110 demonstram uma estrutura de repetição FOR-NEXT, que imprime os números de 1 a 5.

O foco deste código está em especificar detalhadamente **os passos a serem executados pelo computador**, alterando o estado das variáveis e controlando o fluxo de execução. Essa abordagem de "como" fazer as coisas é a essência do paradigma imperativo.

Resumindo...

O paradigma imperativo é centrado em instruções claras sobre como realizar uma tarefa. O foco está em descrever explicitamente os passos necessários para atingir um resultado, envolvendo a definição de variáveis, estruturas de controle de fluxo (como loops e condicionais) e a manipulação direta do estado do programa.

Suas principais características são:

- Definir explicitamente as etapas a serem seguidas
- Computação descrita como ações, enunciados ou comandos que mudam o estado (variáveis) do programa
- Dependência da ordem de execução dos comandos

A linguagem assembly

Arquitetura Von Neumann.

Turing propõe os conceitos
de laços e atribuições.

1945

1936

Alan Turing estabelece as
bases da programação

1947

Linguagem Assembly

De volta a nossa linha do tempo, vamos falar agora sobre Assembly.

A linguagem Assembly é uma linguagem de programação de baixo nível, o que significa que ela está muito próxima da linguagem que o computador entende diretamente. Ela foi desenvolvida em 1949 por Maurice Wilkes para o computador EDSAC.

A pouco, aprendemos sobre o paradigma imperativo ou procedural, e existe um motivo bem especial para falarmos sobre assembly na sequência, porque ela segue justamente esse mesmo paradigma. Isso significa que o programa é escrito como uma série de instruções que o computador deve seguir passo a passo para realizar uma tarefa.

Em vez de usar comandos mais complexos e abstratos, como nas linguagens de alto nível, a linguagem Assembly utiliza instruções simples e diretas, como "ADD", "LOAD" e "STORE". Essas instruções correspondem diretamente às operações que o processador do computador pode realizar.

Veja como é escrito o código em assembly:

```
; Soma dois números e armazena o resultado
; Carregar o primeiro número na memória
LOAD 5 ; Carrega o valor 5 no acumulador
STORE NUM1 ; Armazena o valor 5 na variável NUM1

; Carregar o segundo número na memória
LOAD 3 ; Carrega o valor 3 no acumulador
STORE NUM2 ; Armazena o valor 3 na variável NUM2

; Realizar a soma
LOAD NUM1 ; Carrega o valor de NUM1 no acumulador
ADD NUM2 ; Soma o valor de NUM2 ao acumulador
STORE RESULT ; Armazena o resultado na variável RESULT

; Parar a execução do programa
HALT ; Encerra a execução do programa

; Definição das variáveis
NUM1 DATA 0 ; Reserva espaço na memória para a variável NUM1
NUM2 DATA 0 ; Reserva espaço na memória para a variável NUM2
RESULT DATA 0 ; Reserva espaço na memória para a variável RESULT
```

Neste exemplo, temos algumas características da linguagem Assembly:

- As instruções são simples e diretas, como "LOAD", "ADD" e "STORE".
- O controle de fluxo é feito usando rótulos (labels) como "NUM1", "NUM2" e "RESULT".
- Não há estruturas de controle complexas, como if-else ou loops.
- As variáveis são definidas usando a instrução "DATA" e armazenadas na memória.
- O programa é encerrado com a instrução "HALT".

Uma característica importante da linguagem Assembly é que o controle de fluxo do programa é feito usando "saltos" (jumps) e "rótulos" (labels), em vez de estruturas de controle mais organizadas, como if-else e loops. Isso pode tornar o código mais difícil de entender e manter.

Guarde a informação que ela será bastante importante para entendermos o surgimento do paradigma de orientação a objetos!

Por estar tão próxima do hardware do computador, a linguagem Assembly permite que os programadores tenham um controle muito preciso sobre o funcionamento do computador. Isso a torna útil em aplicações que exigem um desempenho muito eficiente, como sistemas operacionais e drivers de dispositivos.

Devido à sua proximidade com o hardware e à sua estrutura de baixo nível, a linguagem Assembly é considerada mais complexa de aprender do que as linguagens de alto nível, como Python, Java e tantas outras que você já ouviu falar.

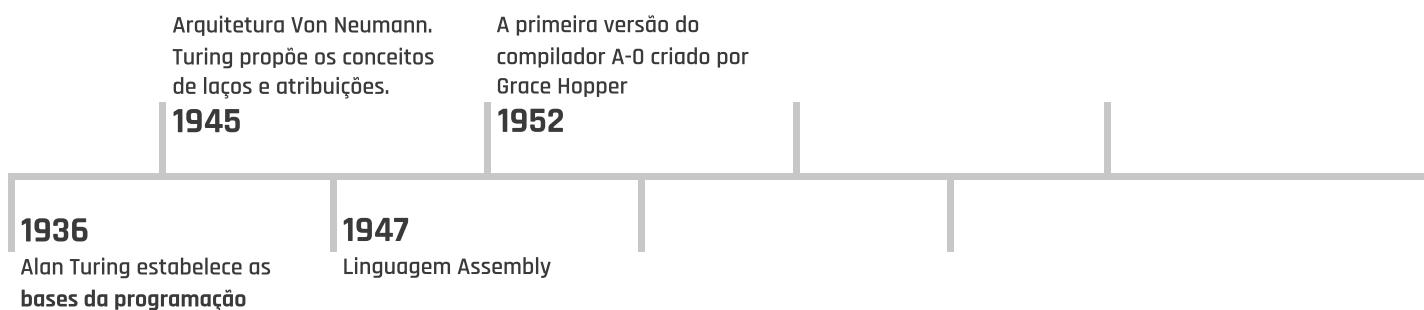
Resumindo...

Continuando neste período de 1940 em diante, começamos a passar por uma evolução nas linguagens, chegando ao Assembly ou assemblers, as chamadas "linguagens de montagem", que, mais uma vez, tinham o objetivo que poupar o trabalho de traduzir os programas para códigos binários.

Um ponto muito importante sobre assembly e arquitetura Von Neumann: as instruções **IF** e **GOTO** estavam incluídas nesta arquitetura, o que permitia que os programas pudessem incluir fluxos não lineares, conforme você acompanhou no trecho de código em C mostrando a aplicação do paradigma imperativo.

👉 Aula 1.3 - Compiladores, Linguagens Lisp e Lambda Calculus

O Compilador



Trazendo uma definição rápida sobre compiladores:

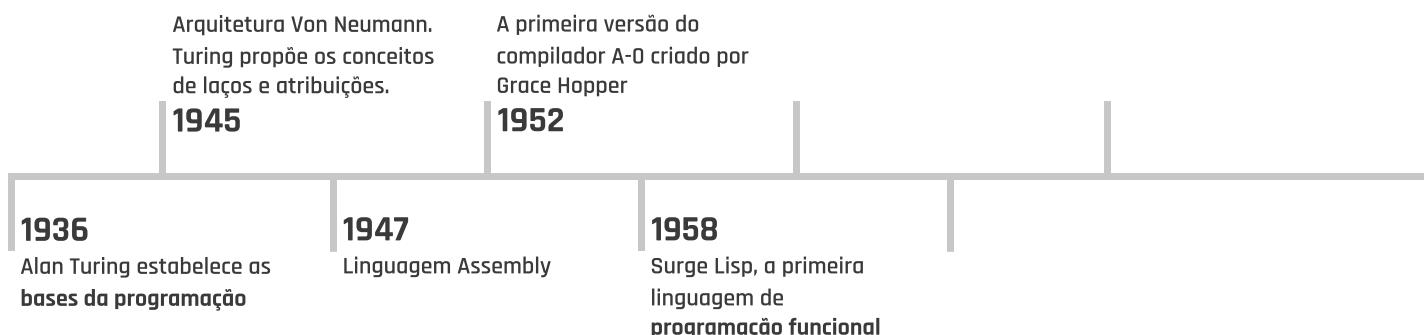
Os compiladores permitem que os programadores escrevam código em linguagens de programação de alto nível, compreensíveis por humanos, e o transformem em instruções de máquina que um computador pode executar.

Até agora, falamos sobre linguagens de máquina e binária, e como funcionava o paradigma imperativo. Com o desenvolvimento de compiladores, tornou-se possível "traduzir" instruções mais fáceis de pessoas como eu e você entendermos, em linguagem de baixo nível, como assembly.

o A-O era uma ferramenta precursora dos compiladores modernos, desenvolvida por Grace Murray Hopper para o UNIVAC I em 1951 e 1952. Foi essa ferramenta que deu origem ao conceito de compilador, que conhecemos hoje em dia.

Também foi graças a criação dos computadores que foi possível conceber a criação de novos paradigmas.

Linguagens LISP e o Lambda Calculus



Após o desenvolvimento do conceito de compilador, tivemos vários pontos importantes, como a invenção do FORTRAN entre 1953-1954, uma linguagem de programação de alto nível e, nos anos subsequentes, apareceram muitas outras linguagens de programação, como COBOL, PL/1, SNOBOL, C, Pascal e assim por diante, até chegarmos na criação da família de linguagens LISP.

John McCarthy desenvolveu Lisp, a primeira linguagem de alto nível de programação funcional. É importante pontuar que o paradigma funcional, apesar de ter sido adotado depois dos demais, foi criado bem antes da programação de computadores. Alonzo Church foi uma figura fundamental na história da programação funcional. Ele desenvolveu o Lambda Calculus em 1936, um sistema formal que serviu de base para a programação funcional. Suas contribuições influenciaram diretamente o surgimento de linguagens de programação funcionais, como o LISP.

De forma bem simples, o cálculo-1 trás o conceito de que os valores não mudam, a imutabilidade, que é uma característica muito marcante do paradigma funcional. Você pode imaginar que, por esse motivo, não existam declarações de atribuição, mas a maioria das linguagens funcionais tem meios para alterar valores de variáveis, mas apenas de forma muito restrita.

Uma pausa para uma reflexão..

Como diz Robert C. Martin em seu livro, Arquitetura Limpa, logo nos primeiros capítulos: **A programação funcional impõe disciplina sobre a atribuição.**

Essa frase é muito importante e vamos entender ao longo das explicações que cada paradigma impõe algum tipo de disciplina ou restrição com o seu uso, embora existam ganhos arquiteturais em usar cada tipo. Lembra do paradigma imperativo que vimos a pouco? Apesar de todos os benefícios apresentados para a época, se fizermos um paralelo com a frase de Martin, podemos afirmar que: **O paradigma imperativo impõe disciplina sobre a sequência de comandos e a manipulação direta do estado do programa**

Isso porque, no paradigma imperativo, a ordem em que os comandos são escritos e executados é crucial, e a manipulação direta do estado do programa através de atribuições e efeitos colaterais requer uma disciplina rigorosa para garantir a correta execução do código e evitar resultados inesperados.

Em programação, sempre precisamos ter em mente uma frase do cantor e compositor Chorão, do Charlie Brown Jr: "cada escolha, uma renúncia". Em desenvolvimento de software, temos uma palavra em inglês que quer dizer a mesma coisa em essência: trade-off.

Toda escolha de linguagem de programação, arquitetura ou qualquer outra escolha que você fizer, vai ter ganhos mas também vai ter algo que vai precisar abrir mão e chamamos esses pontos de tradeoff. Trouxe o exemplo da música apenas para facilitar o entendimento.

➊ Aula 1.4 - Paradigma Funcional

Voltando para o paradigma de programação funcional, vamos entender um pouco de como funciona. Uma das características é que as **funções podem ser atribuídas a variáveis**, passadas como argumentos para outras funções e retornadas como valores. Vamos ilustrar esse conceito com um trecho de código em LISP mas, novamente, não se preocupe em tentar entender as palavras reservadas:

```
; Definindo uma função que recebe outra função como argumento
(defun apply-twice (f x)
  (f (f x)))

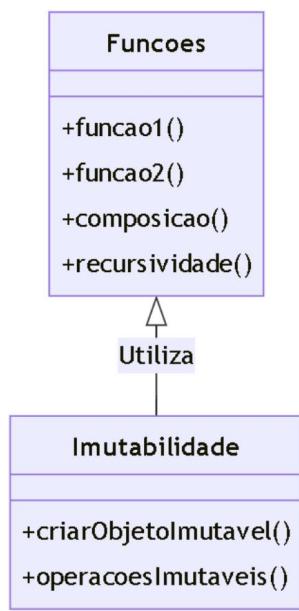
; Definindo uma função que retorna outra função
(defun make-adder (n)
  (lambda (x) (+ x n)))

; Atribuindo uma função a uma variável
(setq add5 (make-adder 5))

; Passando uma função como argumento
(apply-twice add5 3) ; Resultado: 13 (3 + 5 + 5)

; Retornando uma função
(funcall (make-adder 10) 5) ; Resultado: 15 (5 + 10)
```

Nesse exemplo, as funções são tratadas como valores de primeira classe, podendo ser atribuídas, passadas como argumentos e retornadas por outras funções. Explicando uma delas com mais detalhes, a última função, (`funcall (make-adder 10) 5`) a função retornada por `make-adder` é aplicada diretamente ao valor 5, adicionando 10 a ele.



Algumas outras características do paradigma funcional:

- **imutabilidade:** objeto criado não pode ser modificado;

Na programação funcional, quando você cria um objeto (como uma lista, por exemplo), esse objeto não pode ser modificado depois. Isso significa que, se você precisa fazer uma alteração nesse objeto, você não pode simplesmente mudar o que já existe.

Em vez disso, você precisa criar um novo objeto com as alterações desejadas.

Imagine que você tem uma lista de números e quer adicionar um novo número a ela. Em vez de modificar a lista original, você criaria uma nova lista com o novo número adicionado.

Essa nova lista seria um objeto diferente do original, mas com as mesmas características, exceto pela adição do novo número.

- **ausência de efeitos colaterais:** qualquer mudança no estado do sistema ou qualquer interação com o mundo exterior que ocorre dentro de uma função

Outra característica importante da programação funcional é a ausência de efeitos colaterais. Isso significa que, quando você chama uma função, ela não pode alterar nada fora dela mesma.

Por exemplo, imagine uma função que calcula a média de uma lista de números:

```
(defun calculate-average (numbers)
  "Calcula a média de uma lista de números."
  (/ (apply #'+ numbers) (length numbers)))
```

Essa função não deve fazer nada além de receber a lista, calcular a média e retornar o resultado. Ela não deve modificar a lista original, nem interagir com qualquer outra parte do sistema, como gravar algo em um arquivo ou enviar uma mensagem.

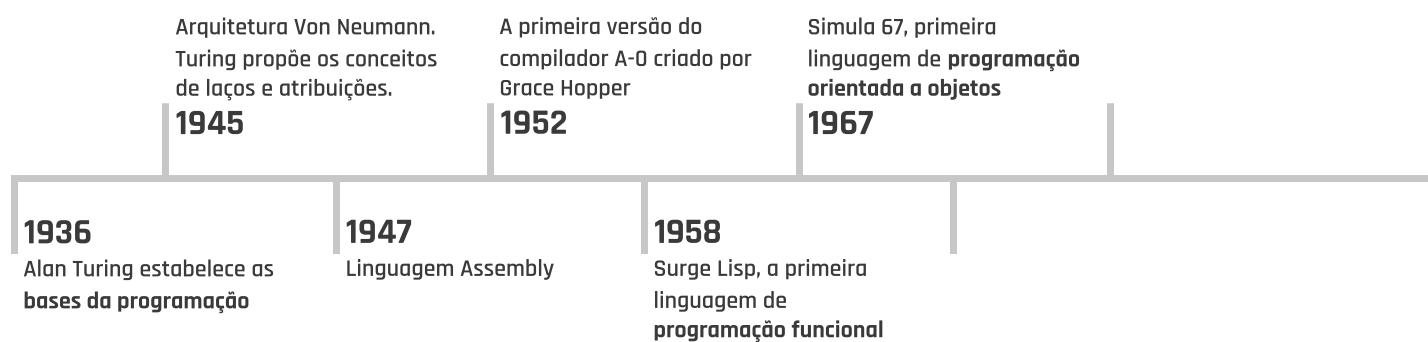
Dessa forma, a execução da função sempre produzirá o mesmo resultado, desde que os mesmos argumentos sejam fornecidos. Isso torna o código mais previsível e fácil de entender e testar.

Resumindo...

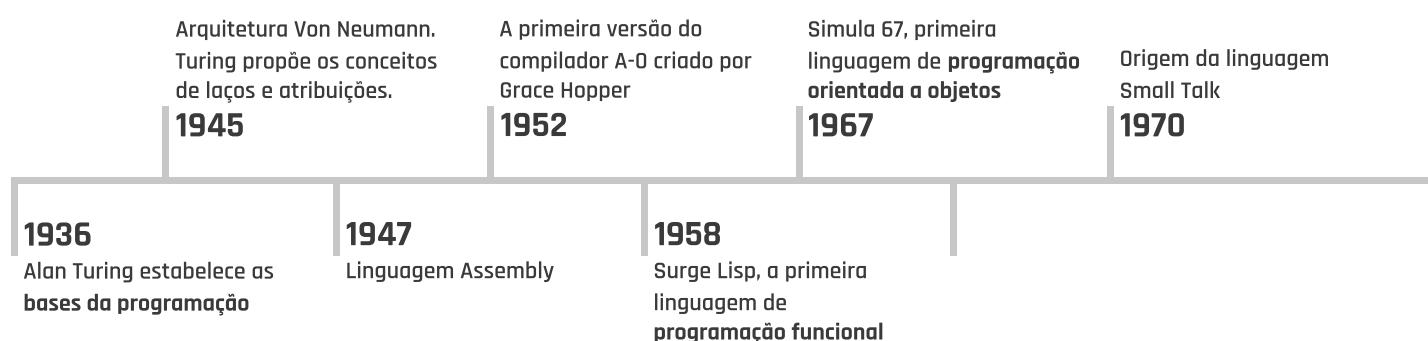
No paradigma funcional, funções podem ser atribuídas a variáveis, passadas como argumentos para outras funções e retornadas como valores.

- **imutabilidade:** objeto criado não pode ser modificado
- **ausência de efeitos colaterais:** qualquer mudança no estado do sistema ou qualquer interação com o mundo exterior que ocorre dentro de uma função

💡 Aula 1.5 - Paradigma de orientação a Objetos



Agora, de 1958, vamos pular para 1967, quando foi desenvolvida a primeira linguagem de programação orientada a objetos, a Simula 67. Ela introduziu conceitos fundamentais da programação orientada a objetos, como classes e herança, embora não com estes nomes, mas esses conceitos vamos ver bastante nestas aulas, pois são algumas das bases da programação orientada a objetos.



A Simula 67 foi a primeira linguagem de programação orientada a objetos, estabelecendo os princípios básicos desse paradigma, enquanto a Smalltalk foi uma linguagem subsequente, lançada nos anos também orientada a objetos, que incorporou e aprimorou os conceitos introduzidos pela Simula 67. Veja que aqui estamos pulando uma das datas, sobre a criação do paradigma estruturado, mas para ficar mais coeso, vou trazer essa linha do tempo primeiro.

Já sabemos que a Simula 67 foi a primeira linguagem de programação orientada a objetos e que a Smalltalk aprimorou os conceitos mas, quando exatamente o paradigma de orientação a objetos surgiu?

Existem muitas explicações sobre o que é Orientação a Objetos mas, para entendermos, precisamos compreender seus conceitos principais: **Herança, Polimorfismo e Encapsulamento**. Neste momento, estamos tendo nosso primeiro contato com os conceitos de paradigmas de programação, então, conceitos de forma mais simplificada e, ao longo das demais aulas, vamos entender na prática.

Conceitos principais do Paradigma de Orientação a Objetos:

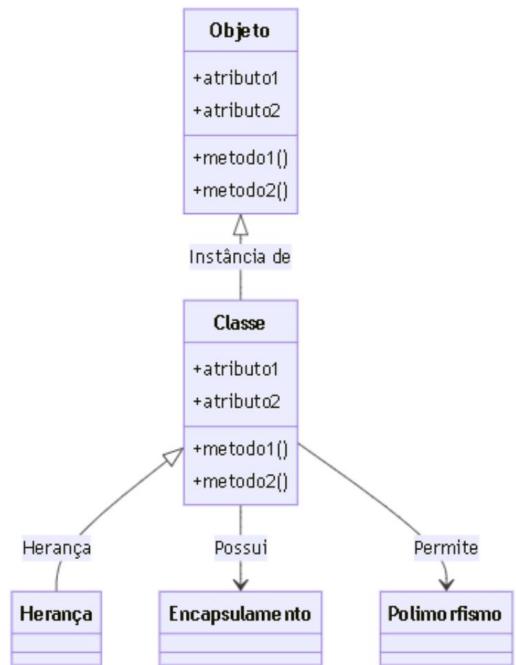
- **Encapsulamento:** Ideia de esconder os detalhes internos de um objeto, expondo apenas o que é necessário para o mundo exterior. Isso significa que os atributos de um objeto podem ser marcados como públicos, privados ou protegidos, controlando quem pode acessá-los
- **Herança:** Permite criar novas classes a partir de classes existentes. Isso significa que uma classe "filha" pode herdar atributos e métodos de uma classe "pai"
- **Polimorfismo:** Permite que objetos de diferentes classes possam ser tratados de maneira similar. Isso significa que um método em uma classe "filha" sobrescreva ou sobreponha um método na classe "pai", permitindo uma implementação diferente.

Um ponto importante para trazer aqui é que **esses 3 conceitos já existiam antes** de Orientação a Objetos ser inventada:

- Já existia encapsulamento em linguagem C, que não é uma linguagem Orientada a Objetos(OO)
- A herança é uma redeclaração de um grupo de variáveis e funções dentro de um escopo fechado, e já era possível aplicar este conceito antes de OO existir
- Polimorfismo é uma aplicação de ponteiros em funções, usada desde que as arquiteturas de Von Neumann foram implementadas.

Embora o polimorfismo já existia, as linguagens OO (Orientação a Objetos) tornaram seu uso muito mais seguro e prático. Usar ponteiros de forma intencional para criar comportamento polimórfico é perigoso porque você precisa lembrar de seguir uma série de convenções e, se esquecer de alguma, vai produzir bugs muito difíceis de encontrar e tratar. Com linguagens OO, o uso de polimorfismo fica mais fácil de aplicar e muito mais seguro.

Orientação a Objetos pode ser entendida como a habilidade de obter controle absoluto, através do uso do polimorfismo, sobre o funcionamento de cada parte do sistema de forma personalizada e eficaz.

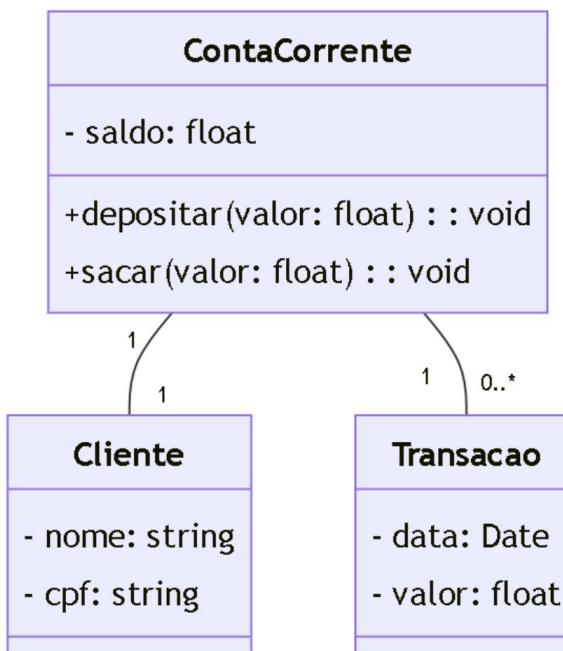


➊ Aula 1.6 - A organização por objetos

No Paradigma de Orientação a Objetos, o código é organizado em torno de objetos e, por isso, recebe este nome. Um objeto é uma "coisa" específica, como um carro, uma pessoa ou uma conta bancária. Trazendo um exemplo de um diagrama de como seria essa organização:

Para organizar o código por objetos no paradigma de orientação a objetos, é fundamental identificar as entidades relevantes e suas interações. No contexto de uma conta corrente, por exemplo, podemos ter objetos como **ContaCorrente**, **Cliente**, **Transacao**, entre outros.

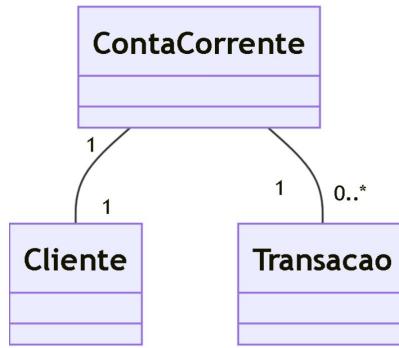
Cada tipo de objeto é definido por uma classe. A classe descreve as características (atributos) e as ações (métodos) que o objeto pode ter. Cada objeto tem seus próprios valores para os atributos, mas pode realizar as mesmas ações (métodos) definidos na classe.



Neste diagrama, temos as classes ContaCorrente, Cliente e Transacao. A classe ContaCorrente possui atributos como saldo e métodos para depositar e sacar. A classe Cliente tem atributos como nome e cpf, enquanto a classe Transacao possui data e valor. A relação entre as classes é representada pelas linhas que conectam os objetos, indicando a associação entre eles.

Essa relação entre classes e objetos é a base da Orientação a Objetos. Ela permite organizar o código de forma modular e reutilizável, facilitando a manutenção e o entendimento do programa.

No nosso diagrama, temos alguns números nas linhas das relações entre os objetos: 1:1 entre conta corrente e cliente e 1...* para conta corrente e transação.



O primeiro caso, ContaCorrente "1" -- "1" Cliente, indica que existe uma associação de **um para um** entre a classe ContaCorrente e a classe Cliente. Isso significa que cada instância de ContaCorrente está associada a exatamente uma instância de Cliente, e vice-versa. Nesse caso, cada conta corrente pertence a um único cliente e cada cliente possui uma única conta corrente.

No segundo caso, ContaCorrente "1" -- "0..*" Transacao, indica que existe uma associação de **um para muitos** entre a classe ContaCorrente e a classe Transacao. Isso significa que uma instância de ContaCorrente pode estar associada a zero ou mais instâncias de Transacao, enquanto cada instância de Transacao está associada a exatamente uma instância de ContaCorrente. Nesse caso, uma conta corrente pode ter várias transações associadas a ela.

Resumindo...

A programação orientada a objetos (OO) organiza o código em unidades independentes, os "objetos".

Cada objeto é um "exemplar" de uma "classe", que seria o "modelo", definindo as propriedades e métodos. Relembrando o que você já viu em outros módulos, objetos tem métodos e propriedades. Métodos são o que o objeto "faz". Propriedades são o que o objeto "é".

Principais conceitos:

- **Encapsulamento:** Esconde detalhes internos, expondo apenas o necessário.
- **Herança:** Permite criar novas classes a partir de classes existentes.
- **Polimorfismo:** Objetos de diferentes classes podem ser tratados de forma similar.

Esses conceitos já existiam antes da Orientação a Objetos, mas as linguagens OO tornaram o uso do polimorfismo mais seguro e prático.

💡 Aula 1.6 - A organização por objetos

Já aprendemos um pouco sobre como o paradigma de orientação a objetos funciona. Entendemos que, embora encapsulamento, herança e polimorfismo já existiam, o OOP tornou o uso do polimorfismo mais seguro e fácil. Citando Robert C. Martin novamente:

OOP (Orientação a Objeto) impõe disciplina na transferência indireta de controle

Esta afirmação, que pode parecer um pouco difícil de entender de primeira (eu demorei muitas semanas tentando entender, quando li este capítulo do livro "Arquitetura Limpa" pela primeira vez) está relacionada à forma como o polimorfismo é aplicado nesse paradigma.

No paradigma procedural ou imperativo, a transferência de controle entre diferentes partes do código é feita de forma direta, geralmente usando instruções como "goto" ou chamadas explícitas de funções. Isso pode levar a um código difícil de entender e manter, com fluxos de controle complexos e potencialmente com efeitos colaterais indesejados.

Vamos usar um exemplo, dessa vez vai ser em linguagem C, para entender como seria essa transferência de controle de forma direta, usando instruções goto (lemos "Go To", separado):

```
#include <stdio.h>

void funcaoB() {
    printf("Funcao B\n");
}

void funcaoA() {
    printf("Funcao A\n");
    goto fim;

inicio:
    printf("Inicio\n");
    funcaoB();
    goto fim;

fim:
    printf("Fim\n");
}
int main() {
    funcaoA();
    return 0;
}
```

A função funcaoA contém instruções "goto" para desviar o fluxo de execução para diferentes partes do código. Isso pode tornar o código confuso, porque a lógica de controle não é linear e depende de desvios explícitos.

Além disso, a chamada explícita da função funcaoB dentro de funcaoA e o uso de rótulos como inicio e fim sem uma estrutura clara de controle de fluxo podem dificultar a compreensão do que o código está fazendo e como ele está organizado.

Esse tipo de código, com desvios de fluxo não estruturados e chamadas explícitas de funções, pode ser difícil de entender e manter, especialmente à medida que o código cresce em complexidade. É um exemplo de como o uso excessivo de instruções como "goto" e chamadas explícitas de funções tornavam o código muito difícil de entender e fácil para gerar bugs.

Já na programação orientada a objetos, o polimorfismo permite que a transferência de controle entre diferentes partes do código seja feita de forma indireta, através da invocação de métodos em objetos. Isso impõe uma disciplina na forma como essa transferência de controle é realizada.

Agora, vamos ver nosso primeiro exemplo de código em Java! Não se preocupe se não entender algum trecho, vamos explicando o que cada parte do código faz. E como pessoa engenheira, treine seu olhar para procurar entender a lógica de um código, não crie apegos a linguagem em si, que ao longo da sua carreira, isso vai mudar.

```
public class ExemploOrientacaoObjetos {  
  
    public static void main(String[] args) {  
        FuncaoA funcaoA = new FuncaoA();  
        funcaoA.executar();  
    }  
}  
  
class FuncaoA {  
  
    void executar() {  
        System.out.println("Inicio");  
        FuncaoB funcaoB = new FuncaoB();  
        funcaoB.executar();  
        System.out.println("Fim");  
    }  
}  
  
class FuncaoB {  
  
    void executar() {  
        System.out.println("Funcao B");  
    }  
}
```

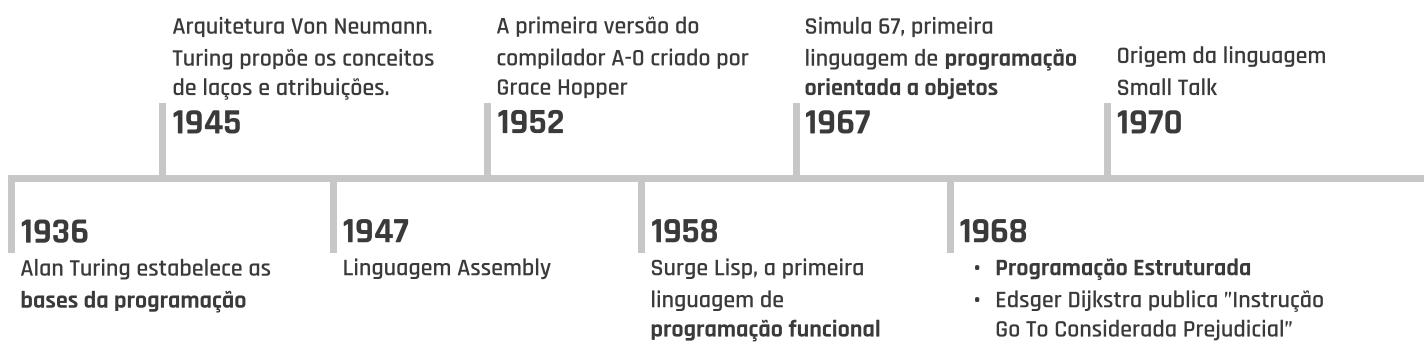
O que este código faz:

1. A função FuncaoA começa imprimindo "Inicio", depois chama a função FuncaoB, que imprime "Funcao B", e por fim imprime "Fim".
2. Em vez de usar "goto" para pular entre partes do código, usamos objetos e métodos para controlar o que acontece em cada etapa.
3. Cada classe tem sua própria função específica, e ao chamar essas funções em sequência, conseguimos controlar o fluxo do programa de forma mais organizada e fácil de entender.
4. Isso é um exemplo de polimorfismo, pois a classe FuncaoA não conhece a implementação específica da classe FuncaoB, mas ainda assim consegue chamar o método executar() da classe FuncaoB sem precisar saber que é uma classe diferente.

Quando você chama um método em um objeto, você não precisa se preocupar com a implementação específica desse método. O objeto se encarrega de executar a implementação correta, com base no seu tipo real. Isso torna o código mais modular, flexível e fácil de entender.

Essa abstração do uso de ponteiros e a aplicação do polimorfismo de forma segura e intuitiva é um dos grandes ganhos da programação orientada a objetos. Ela permite que o desenvolvedor se concentre na lógica do programa, sem se preocupar tanto com os detalhes de baixo nível da transferência de controle.

➊ Aula 1.8 - A crítica ao uso do Go To, Matemática, Ciência e a Programação estruturada



Agora, chegamos no último paradigma de programação que vamos conhecer aqui: a Programação Estruturada. Voltando para 1968, vamos falar da importância do artigo "Instrução Go To Considerada Prejudicial", escrito por Dijkstra.

Programar era um desafio muito grande, ainda mais para quem não tinha muita qualificação. Independente do nível de complexidade, qualquer programa de computador tem mais detalhes do que nosso cérebro pode processar sem ajuda. Principalmente porque as coisas parecem funcionar mas começam a falhar das formas mais inesperadas possíveis.

Dijkstra observou que os matemáticos constroem suas teorias a partir de uma hierarquia bem definida de postulados, teoremas, corolários e lemas. Essa estrutura organizada permite que eles provem a correção de suas afirmações de maneira rigorosa.

Inspirado por essa abordagem matemática, Dijkstra acreditava que os desenvolvedores de software também deveriam adotar uma hierarquia semelhante. Sua ideia era que os programadores usassem estruturas comprovadas, como essa hierarquia de conceitos matemáticos, para escrever códigos que pudessem então ser formalmente provados como corretos.

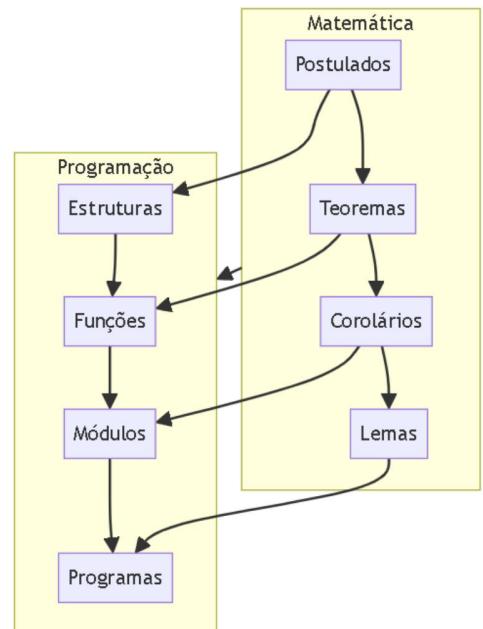
As Estruturas de programação (como controle de fluxo) seriam análogas aos Postulados matemáticos. As Funções corresponderiam aos Teoremas, os Módulos aos Corolários e os Programas aos Lemas. Essa hierarquia estruturada permitiria a prova formal da correção dos códigos.

Desta forma, Dijkstra viu um paralelo entre a maneira como os matemáticos constroem suas teorias e a forma como os programadores deveriam desenvolver seus programas. Isso o levou a questionar o uso indiscriminado do comando "Go To" e a defender uma abordagem mais estruturada e comprovada para a programação.

Relembrando o que aprendemos sobre paradigma imperativo, o comando "Go To" é uma instrução que permite que o fluxo de execução de um programa salte para uma parte específica do código, muitas vezes resultando em estruturas de controle complexas e difíceis de manter.

O impacto do artigo de Dijkstra foi muito grande, foi o que alguns autores chamaram de "revolução" sobre o tema. Apesar de críticas muito negativas à abordagem, conforme as linguagens de programação foram evoluindo, o uso do Go To foi caindo em desuso, até sumir.

Apesar de toda esta teoria parecer a solução dos problemas, na realidade, essa hierarquia nunca foi colocada em prática e este processo de comprovação formal era complexa de implementar. Ou seja, o artigo serviu apenas para levantar a discussão de abandonar o uso da instrução Go To, mas nada foi implementado. Ok, mas porque contamos toda essa história, se ela não foi pra frente?



Uma pausa para uma reflexão..

A matemática é a disciplina que comprova declarações como verdade. Por outro lado, a ciência consegue provar quais declarações são falsas.

Ou seja, um programa pode ser provado como incorreto por um teste, mas um teste não pode provar que um programa está correto.

Embora possa parecer que o software tem mais a ver com matemática, o software está mais para ciência.

Um programa que não é comprovável, como um projeto com uso abusivo da instrução Go To, não pode ser considerado como correto, mesmo que apliquemos muitos e muitos testes sobre ele e todos os testes passem. Provas de inexatidão podem ser aplicadas apenas em programas comprováveis, sendo que o programa comprovável é aquele que vai apresentar algum resultado.

Usando programação estruturada, nossa única alternativa é decompor um programa recursivamente em pequenas funções comprováveis (como falamos a pouco, que vão apresentar algum resultado). Assim, nossos testes seriam para verificar se o comportamento dessas funções estão como esperado. Se o teste "passar", não significa que o programa esteja totalmente certo e a prova de bugs, mas que as funções estão corretas para o nosso objetivo, ou seja, apresentam o resultado esperado.

Enquanto a matemática se concentra em provar a verdade, e a ciência em identificar falhas, o software encontra seu lugar mais próximo da ciência. Um programa, mesmo passando em testes, não pode ser considerado correto se não for comprovável. A programação estruturada decompõe nossos programas em funções comprováveis, permitindo testar e validar cada parte de forma precisa. Ao focar em funções com resultados mensuráveis, podemos identificar e corrigir falhas, garantindo a confiabilidade do software que desenvolvemos.

• Aula 1.9 - Paradigma de Programação estruturada e Decomposição Funcional

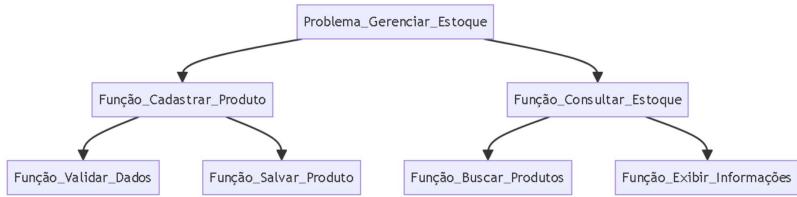
A programação estruturada permite pegar a declaração de um problema de larga escala e decompor este problema em funções de alto nível. E cada uma dessas funções pode ser decomposta em funções de níveis mais baixos e assim por diante. Cada uma dessas funções decompostas pode ser representada por estruturas de controle estritas.

Esse paradigma se baseia na divisão lógica e estruturada de um programa em blocos de código mais simples e fáceis de entender. Essa abordagem enfatiza a utilização de estruturas de controle como sequência, seleção (if/else) e repetição (loops), permitindo uma organização clara e eficiente do código.

- 1. Sequência:** A sequência é a estrutura de controle de fluxo mais simples. Ela implica que as instruções sejam executadas uma após a outra, na ordem em que estão escritas. Isso significa que o programa executa cada instrução sequencialmente, sem saltos ou loops.
- 2. Seleção:** A seleção é a estrutura de controle de fluxo que permite escolher entre diferentes caminhos de execução, baseado em uma condição específica. Isso é feito com a ajuda de estruturas como if/else, switch/case, etc. A seleção permite que o programa tome decisões dinamicamente, baseado em condições específicas.
- 3. Iteração:** A iteração é a estrutura de controle de fluxo que permite executar uma sequência de instruções repetidas vezes, até que uma condição específica seja satisfeita. Isso é feito com a ajuda de estruturas como loops for, while, do/while, etc. A iteração permite que o programa execute uma ação repetida, até que uma condição específica seja atendida.

Essas estruturas de controle de fluxo são essenciais para a decomposição funcional, pois permitem dividir um problema complexo em partes menores e mais gerenciáveis, cada uma com sua própria lógica de controle de fluxo. Mas, o que é decomposição funcional? Vamos entender agora!

A decomposição funcional é um método de análise e design onde processos complexos são divididos em partes menores e mais gerenciáveis. Essa abordagem envolve a decomposição de sistemas de software em módulos ou funções, permitindo uma melhor compreensão e organização do código. Ao dividir um programa em funções de alto nível, que por sua vez são compostas em funções de níveis mais baixos, a decomposição funcional ajuda a estruturar o código de forma clara e modular.



A decomposição funcional ainda é considerada uma das melhores práticas no desenvolvimento de software, especialmente para sistemas de maior complexidade. E a programação estruturada é uma metodologia eficaz que visa criar programas mais legíveis, modulares e robustos, contribuindo para o desenvolvimento de software de alta qualidade.

Algumas linguagens de programação que usam paradigma estruturado (apesar de terem suporte para outros paradigmas também) são: Linguagem C, Cobol, PHP, Perl e Go.

Para fixarmos nosso conhecimento com código, vamos relembrar como seria um programa em C usando as instruções Go To:

```
#include <stdio.h>

int main() {
    int i = 0;

loop:
    printf("%d\n", i);
    i++;

    if (i < 5) {
        goto loop;
    }

    return 0;
}
```

Neste trecho, simplesmente usamos o comando Go To para criar um loop simples de impressão de números de 0 a 4. Mas lembre-se que num código de um projeto real, isso seria apenas uma pequena parte do todo! Aqui estamos vendo o mais simples, por enquanto, para entender conceitos.

Agora, vamos ver como seria o mesmo código, em paradigma estruturado:

```
#include <stdio.h>

void imprimirNumeros() {
    for (int i = 0; i < 5; i++) {
        printf("%d\n", i);
    }
}

int main() {
    imprimirNumeros();
    return 0;
}
```

No exemplo do código usando o paradigma estruturado, a lógica de imprimir os números de 0 a 4 está separada em uma função chamada `imprimirNumeros()`. Essa função contém um loop for que percorre os números e imprime cada um deles.

Ao colocar essa lógica em uma função separada, o código fica mais organizado e fácil de entender. Ao invés de ter todo o código de impressão dos números espalhado pelo programa, ele está concentrado em um único lugar, dentro da função `imprimirNumeros()`.

Além disso, o uso do loop for para iterar sobre os números é uma abordagem estruturada, pois segue uma estrutura de controle de fluxo bem definida. Isso torna o código mais claro e fácil de manter do que usar instruções goto para criar o mesmo loop.

Resumindo...

O Paradigma Estrutural surgiu como uma alternativa à programação imperativa tradicional, que fazia uso excessivo de instruções "goto" para controlar o fluxo do programa. Essa abordagem resultava em código difícil de entender e manter.

Em 1968, Dijkstra publicou o artigo "Instrução Go To Considerada Prejudicial", no qual defendia a abolição do comando "go to" nas linguagens de programação de alto nível. Dijkstra argumentava que o uso indiscriminado do "goto" contribuía para erros de programação e práticas inadequadas, tornando o código extremamente complexo.

A publicação desse artigo de Dijkstra foi um marco importante para o surgimento da Programação Estruturada. Esse paradigma se baseia em três estruturas de controle de fluxo:

- sequência,
- seleção (if/else)
- iteração (loops).

Ao organizar o código em torno dessas estruturas, a programação estruturada promove uma maior legibilidade, modularidade e facilidade de manutenção.

Outro conceito-chave da programação estruturada é a Decomposição Funcional. Essa abordagem envolve dividir um problema complexo em partes menores e mais gerenciáveis, representadas por funções ou procedimentos. Essa divisão hierárquica do código facilita a compreensão, a reutilização e a validação de cada módulo de forma independente.

1. Tire um tempo para refletir sobre tudo que vimos nesta aula que, apesar de teórica, é muito importante. Hoje em dia é muito fácil obter trechos de código usando ferramentas na internet mas, é preciso entender o porquê e se faz sentido. E como futura pessoa desenvolvedora, você é que vai fornecer essas respostas aos porquês.
2. Imagine que você está desenvolvendo um sistema de gerenciamento de uma biblioteca. Neste sistema, é necessário modelar livros, autores, usuários, empréstimos e devoluções. Cada livro possui informações como título, autor e gênero. Os autores têm nome, nacionalidade e obras publicadas. Os usuários da biblioteca têm nome, idade e histórico de empréstimos. Os empréstimos registram a data de retirada e devolução, além do livro e usuário envolvidos.

Com base no cenário descrito, analise qual paradigma de programação seria mais indicado para implementar o sistema de gerenciamento da biblioteca:

- a) programação estruturada
- b) programação imperativa
- c) programação orientada a objetos ou
- d) programação funcional

Justifique sua escolha considerando a estrutura e as interações entre os objetos do sistema.

Para facilitar, considere pontos como:

- Estrutura e interação das entidades do sistema (Livros, autores, usuários e demais)
- Reuso
- Qual paradigma facilitaria a evolução do sistema, no futuro
- Eficiência, dado o cenário apresentado



Dica: A combinação dos paradigmas de programação pode ser viável em certos contextos, mas geralmente não é recomendada devido a diferenças fundamentais entre os paradigmas

3. Você deve ter percebido que nossas aulas tem vários diagramas. Eles ajudam na correta compreensão dos conceitos e, quando projetamos sistemas reais, esses recursos nos ajudam a entender alguns detalhes de implementação e viabilidade do que estamos fazendo. Neste exercício, você vai criar seu primeiro diagrama!

Antes de começar a desenhar, tenha em mãos as seguintes respostas:

- Quais serão os elementos (classes) do diagrama?
- Com os conhecimentos que você já tem em programação, consegue pensar no tipo de cada propriedade? Por exemplo, no caso do livro, qual seria o tipo da propriedade "Título"? Seria uma string, um number, ou outro tipo?
- Qual seria o relacionamento entre os elementos? Pense em relacionamentos 1 para 1 e 1 para muitos

Regras:

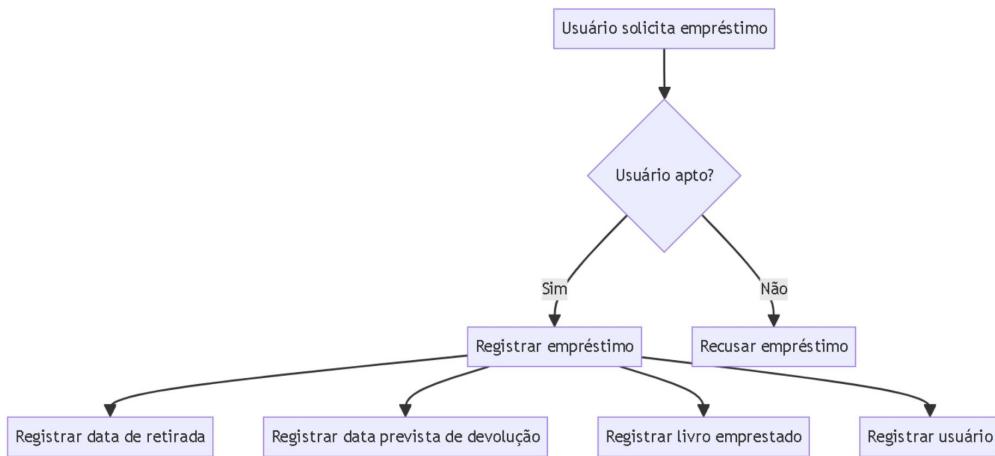
Seu diagrama deve estar de acordo com o que você definiu no exercício anterior, em Elementos de diagrama e Relações entre elementos

4. Agora, vamos acrescentar mais alguns detalhes ao seu diagrama! Já desenhamos o diagrama de classes, contendo propriedades e relacionamentos entre os objetos.

Agora, você vai receber os fluxos do nosso sistema de gerenciamento! Precisamos dessas informações para completar nosso exercício, que vai ser adicionar os métodos às nossas classes, de acordo com o que o sistema deve fazer. Para facilitar, vamos deixar os fluxos mais simples para o nosso projeto de aula.

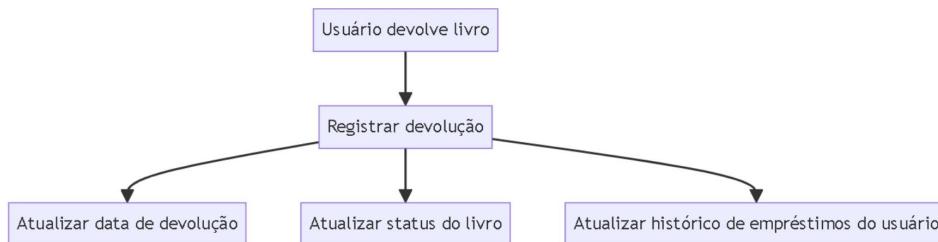
Fluxo de Empréstimo de Livros

1. O usuário solicita o empréstimo de um ou mais livros.
2. O sistema verifica se o usuário está apto a realizar o empréstimo:
 - **a)** Consulta o nome do usuário para identificá-lo
 - **b)** Verificar o histórico de empréstimos do usuário para ver se ele não possui livros vencidos.
 - **c)** Verificar a idade do usuário, caso o livro solicitado seja de um gênero impróprio para menores de 18 anos.
3. Se o usuário estiver apto, o sistema:
 - **a)** Registra a data de retirada do livro.
 - **b)** Registra a data prevista de devolução.
 - **c)** Associar o livro emprestado e o usuário que realizou o empréstimo.
4. Caso o usuário não esteja apto, o sistema recusa o empréstimo



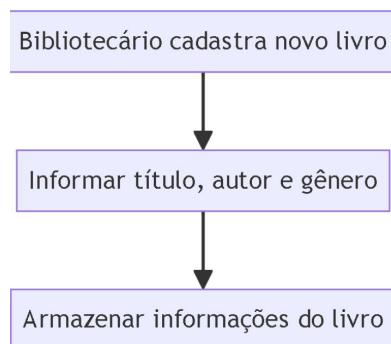
Fluxo de Devolução de Livros

1. O usuário devolve um ou mais livros.
2. O sistema registra a devolução, atualizando:
 - **a)** A data de devolução do livro.
 - **b)** O status do livro como disponível na biblioteca.
 - **c)** O histórico de empréstimos do usuário.



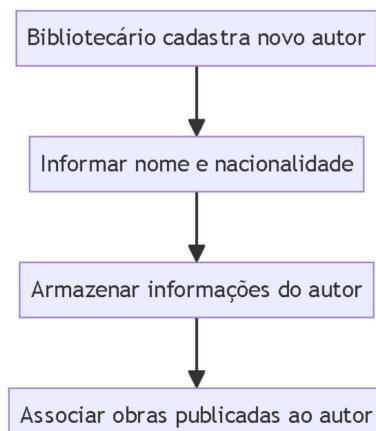
Fluxo de Cadastro de Livros

1. O bibliotecário cadastra um novo livro no sistema, informando:
 - **a)** O título do livro.
 - **b)** O autor do livro.
 - **c)** O gênero do livro.
2. O sistema armazena as informações do novo livro.



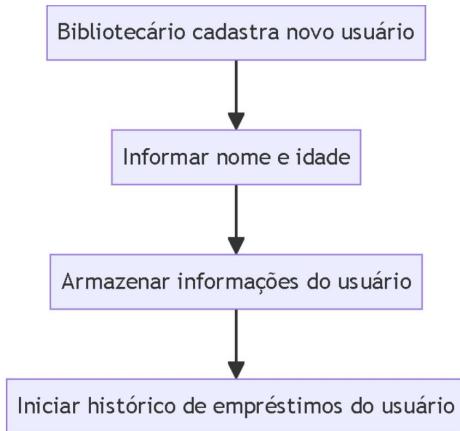
Fluxo de Cadastro de Autores

1. O bibliotecário cadastra um novo autor no sistema, informando:
 - **a)** O nome do autor.
 - **b)** A nacionalidade do autor.
2. O sistema armazena as informações do novo autor e associa as obras publicadas a ele.



Fluxo de Cadastro de Usuários

1. O bibliotecário cadastra um novo usuário no sistema, informando:
 - **a)** O nome do usuário.
 - **b)** A idade do usuário.
2. O sistema armazena as informações do novo usuário e inicia seu histórico de empréstimos.



Com base em todos os fluxos e regras de negócio, quais seriam os métodos que precisamos acrescentar em nossas classes?

Instruções para submissão:

No pasta **modulo1** do seu repositório no Github coloque os seguintes arquivos:

- a) `dissertativa.txt`: um arquivo de texto com a solução dissertativa.
- b) `diagrama-classes-1.png`: tente esboçar como você faria o desenho de diagrama de classes do nosso sistema!
- c) `diagrama-classes-2.png`: esse vai ser a evolução do nosso diagrama anterior, mas, dessa vez, com uma visão mais detalhada de cada classe.

Você pode usar ferramentas gratuitas como o <https://excalidraw.com/> ou outra que preferir. Desenhar à mão e tirar foto também vale! Não se preocupe com perfeição de desenho, o importante é ficar legível

Boa sorte!

Módulo 2 - Classes, Encapsulamento, Herança e Modificadores de Acesso

→ Aula 2.1 - Refletir, desenhar soluções e escrever código

Nos exercícios da aula passada, fizemos nosso primeiro diagrama de como vai ser o projeto final, escolhendo desde o paradigma de programação mais adequado, estudando os fluxos do que seria esperado que o sistema fizesse e utilizando esta informação para modelar nossas classes, definindo os métodos que cada uma deveria conter.

Ainda vamos modificar o desenho ao longo das demais aulas mas, para o que aprendemos até o momento, esse diagrama uniu todos os conceitos.

Ah, mas e o código? Quando começa a escrever código?

O código é muito importante, mas, é mais importante ainda, saber o que fazer com ele. Todas essas coisas podem parecer chatas mas, o que estou trazendo nessas primeiras aulas é trabalho de uma pessoa arquitetura de software

Quando eu atuei na área de arquitetura de um grande banco espanhol, era este o trabalho que eu fazia:

- Entendia o contexto de negócio
- Qual seria a arquitetura mais adequada para atender aquele problema
- Fazia toda a modelagem de alto nível, para garantir que as coisas fariam sentido e que a aplicação seria fácil de entender e de manter
- Depois, escrevia o código

Acho muito importante que você comece a sua carreira entendendo e aplicando conceitos de arquitetura de software e design de sistemas, que se acostume a trabalhar da forma que fazemos no mercado, analisando as regras de negócio e fluxos e refletindo no código, porque vai facilitar o seu processo de crescimento profissional nos próximos anos.

Começar com as bases bem sólidas, principalmente de conceitos que vamos usar a nossa vida toda trabalhando com desenvolvimento de software, vai facilitar seu desenvolvimento e evitar aquelas sensações de "vão descobrir que eu não sei nada e me demitir" porque você vai ter a segurança de saber aplicar o que aprendeu não apenas em java, mas na linguagem de orientação a objetos que precisar.

Linguagem de programação é detalhe de implementação

Isso quer dizer que usar Java, Python, C# ou C++ não vai mudar o desenho da solução de alto nível, que foi o que desenhamos até agora. A linguagem envolve detalhes de implementação, algo que fica inclusive muito mais fácil quando já temos as regras de negócio, requisitos de desenho da solução definida. Porém, começar pela linguagem de programação e depois ir vendo esses detalhes de implementação no caminho, não vai te ajudar a desenvolver raciocínio de engenharia e certamente, vai te fazer ter muito retrabalho e demorar.

Dê tanta atenção quanto possível para essa parte de entendimento de conceitos. Depois, o código vai ser a parte mais fácil.

→ Aula 2.2 - Classes

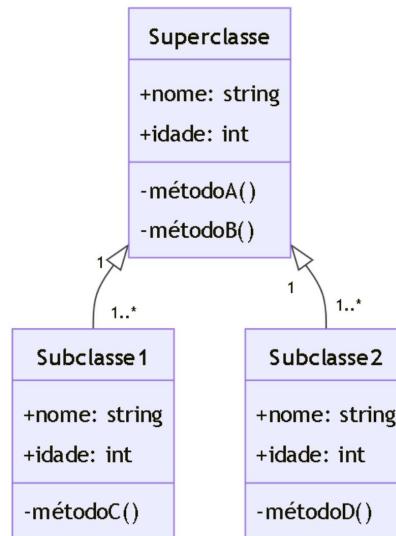
As classes em programação orientada a objetos são como "moldes" ou "modelos" a partir dos quais são criados objetos. Elas definem a estrutura e o comportamento desses objetos. É possível criar instâncias de uma classe (objetos), que, usando uma analogia bem simples, seria como usar um cortador de biscoitos em uma massa, onde nossa classe seria o cortador.

As classes contêm atributos (dados) e métodos (funções) que descrevem as características e comportamentos dos objetos. Quando criamos uma instância de classe, esse objeto herda os atributos e métodos da classe. Talvez você já tenha ouvido o termo "classe-pai" ou "classe-base" que quer dizer a mesma coisa e "classe-filha" porque, assim como na genética, que os filhos carregam características dos pais, nossos objetos também carregam as características (métodos e atributos) das suas superclasses.

Para não causar confusões, vamos usar superclasse para nos referirmos às classes-base ou classes-pai e sub-classes para as classes filhas.

A superclasse é essencial na programação orientada a objetos, já que estabelece a estrutura e o comportamento básico que será herdado pelas subclasses, promovendo a reutilização de código e a organização hierárquica das classes.

Este diagrama apresenta uma superclasse com atributos nome e idade, e métodos métodoA e métodoB. Duas subclasses, Subclasse1 e Subclasse2, herdam esses atributos e métodos da superclasse. Cada subclass adiciona seus próprios atributos e métodos (métodoC e métodoD, respectivamente).



Herança

A seta para cima, que representa a palavra reservada extends indica que Subclasse1 e Subclasse2 são subclasses da superclasse. Para essa ideia de extends fazer mais sentido, vamos dizer que “a Subclasse2 estende(extends) os métodos e atributos da Superclasse”, que representado em código, seria

```
class Subclasse1 extends Superclasse
```

O símbolo 1..* indica que uma superclasse pode ter várias subclasses.

Em programação orientada a objetos, uma classe não pode herdar de mais de uma classe diretamente. Isso é uma limitação importante a ser considerada ao projetar hierarquias de classes e garantir a coerência e clareza na estrutura do código.

Esse conceito é conhecido como **herança simples**, onde uma subclass herda propriedades e comportamentos de uma única superclasse. Em linguagens como Java, que suportam apenas herança simples, uma classe não pode herdar diretamente de múltiplas superclasses.

De curiosidade, você pode encontrar herança múltipla em linguagens de programação como C++ e Swift, porém, a herança múltipla pode gerar complexidades e ambiguidades no código, o que levou linguagens modernas, como o Java, a optarem por não suportar esse recurso.

Resumindo...

- **Classe:** Uma classe é um modelo ou estrutura que define a estrutura e comportamento de objetos. As classes são a base da programação orientada a objetos e são usadas para criar objetos que podem ter atributos e métodos.

- **Superclasse e Subclasse:** Uma superclasse é uma classe que fornece métodos e atributos para uma subclasse, ou seja, é a classe base para suas subclasses. Uma subclasse é uma classe que herda métodos e atributos de sua superclasse e pode acrescentar novos atributos e métodos ou sobrepor métodos herdados da superclasse. A relação entre uma superclasse e suas subclasses é de um para muitos, onde uma superclasse pode ter várias subclasses, mas cada subclasse está associada a apenas uma superclasse. A subclasse pode também adicionar novos métodos e atributos que não estão presentes na superclasse.
- **Herança:** Uma classe pode herdar atributos e métodos de outra classe. A herança simples, mais usada nas linguagens modernas como Java, permite que uma subclasse possa herdar apenas de uma superclasse. A herança múltipla de classes é suportada em linguagens como C++ e Swift, permitindo que uma classe herde de várias superclasses, mas esta abordagem pode causar complexidades no código e, como Java é uma linguagem pensada para ser mais simples, só tem suporte para herança simples.

→ Aula 2.3 - Construindo uma classe em Java

Ok, já entendemos do conceito de classes, já desenhamos classes em alto nível (que é nosso diagrama, porque ele abstrai detalhes de implementação) e pudemos nos concentrar em como todas as classes se relacionam e que informações deveriam carregar.

Agora, vamos entender como construir uma classe em Java. Não se preocupe neste momento de já começar a implementar algo, vamos primeiro entender o conceito e, na parte de exercício, vamos ter a oportunidade de trabalhar em código.

Partindo do princípio que você já aprendeu o que é uma classe, como vamos precisar usar várias em Java, temos uma convenção para chamar esses agrupamentos de várias classes que se relacionam, como o exemplo do nosso diagrama: pacote

Pacote

Um pacote em Java é como uma pasta que ajuda a organizar classes relacionadas. Os pacotes evitam que classes com o mesmo nome causem confusão, pois cada classe está em um pacote diferente. Isso quer dizer que eu posso criar duas classes chamadas "pessoa" em pacotes diferentes? Sim! Porque, como falamos, pacotes são pastas de classes relacionadas.

Além disso, os pacotes têm outra função muito importante: Ajuda a encontrar e gerenciar classes de forma mais fácil, especialmente em programas grandes. Você pode criar seus próprios pacotes para organizar suas classes, como organizar arquivos em pastas no computador.

Anatomia de uma classe

Superclasses e subclasses são classes. A diferença está apenas no nosso pensamento de como elas devem ser acessadas. Dito isso, vamos olhar agora o que é uma Superclasse e como ela deve ser:

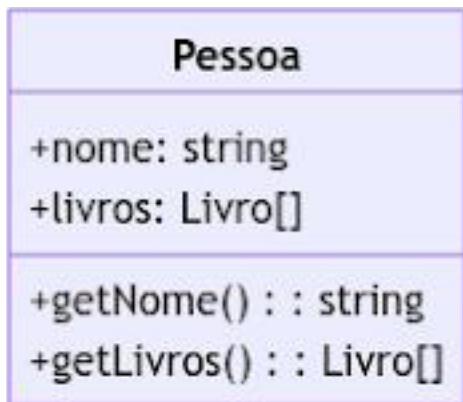
Superclasse:

- Definida usando a palavra-chave `class` seguida pelo nome da classe.
- Pode conter atributos e métodos que serão herdados pelas subclasses.
- Os atributos e métodos da superclasse podem ter diferentes modificadores de acesso, como `public`, `protected`, `private`, ou `default`.
- Para permitir a herança, os atributos e métodos que se deseja que sejam herdados devem ser declarados com os modificadores de acesso apropriados.
- A superclasse pode ter um construtor que pode ser invocado pelas subclasses usando a palavra-chave `super`. Vamos ver mais adiante sobre construtores.

Subclasse:

- Uma subclasse em Java é definida usando a palavra-chave `class` seguida pelo nome da classe, e a palavra-chave `extends` seguida do nome da superclasse da qual ela está herdando.
- A subclasse herda atributos e métodos da superclasse e pode adicionar novos atributos e métodos específicos.
- A subclasse pode sobreescrivar métodos da superclasse, fornecendo uma implementação específica para aquele método na subclasse.
- A subclasse pode invocar o construtor da superclasse usando a palavra-chave `super` para inicializar os atributos herdados da superclasse.

Vamos começar pela Superclasse, até porque, precisaremos aprender muitos conceitos novos para implementá-la. Lembra do nosso exercício de desenhar o diagrama de classes? Vamos rever o desenho de uma das classes definidas, a classe Pessoa e criar um código em Java para representá-la:



```
public class Pessoa {  
    private String nome;  
    private Livro[] livros;  
  
    public Pessoa(String nome, Livro[]  
    livros) {  
        this.nome = nome;  
        this.livros = livros;  
    }  
  
    public String getNome() {  
        return nome;  
    }  
  
    public Livro[] getLivros() {  
        return livros;  
    }  
}
```

Foi muito importante fazer o desenho da classe primeiro porque, na hora de criar o código, precisamos apenas trazer as mesmas informações. Por esse motivo, sempre prefira desenhar a parte do sistema que vai implementar antes de começar a escrever código! Guarde esse trecho de código da nossa classe que vamos usá-lo a seguir.

→ Aula 2.4 - Preparando o ambiente e rodando nosso primeiro código

As pessoas que estavam ansiosas para começar a criar código, este vai ser o momento! Para começarmos a tangibilizar o que já desenhamos até agora em código, vamos preparar nosso ambiente básico de desenvolvimento. Essas orientações são direcionadas para instalação em sistemas Linux.

Exercício guiado: rodando arquivos .java no terminal

Passo 1: Instalação do Java Development Kit (JDK)

1. Abra o Terminal.
2. Atualize os pacotes do sistema com o comando: `sudo apt-get update`.
3. Instale o JDK com o comando: `sudo apt-get install openjdk-13-jdk`. Vamos usar a versão Java 13 em nossas aulas.

Passo 2: Configuração do Ambiente

1. Verifique se o JDK foi instalado corretamente digitando: `java -version`.
2. Defina a variável de ambiente `JAVA_HOME` adicionando a seguinte linha ao arquivo `.bashrc` (localizado em sua pasta home): `export JAVA_HOME=/usr/lib/jvm/java-11-openjdk-amd64`.
3. Atualize as variáveis de ambiente com o comando: `source ~/.bashrc`.

Passo 3: Escrevendo o Código

1. Abra um editor de texto, como o Notepad++ ou outro editor de texto simples. Escreva o código da classe "Pessoa" com uma pequena alteração. Como ainda não temos a nossa classe Livro, vamos deixar com um array genérico e, posteriormente, deixamos o código conforme desenhemos. Sendo, vamos criar um erro na hora de compilar. Copie este código (vamos entender linha a linha depois, não se preocupe!):

```
import java.util.Arrays;

public class Pessoa {
    private String nome;
    private String[] livros;

    public Pessoa(String nome, String[] livros) {
        this.nome = nome;
        this.livros = livros;
    }

    public String getNome() {
        return nome;
    }

    public String[] getLivros() {
        return livros;
    }

    public static void main(String[] args) {
        String[] livros = {"Arquitetura Limpa", "Desbravando Java", "SOLID para
Ninjas"};
        Pessoa pessoa = new Pessoa("Jess", livros);
        System.out.println("Nome: " + pessoa.getNome());
        System.out.println("Livros: " + Arrays.toString(pessoa.getLivros()));
    }
}
```

2. Salve o arquivo com o nome "Pessoa.java".

Passo 4: Estrutura de Diretórios

1. No Terminal, navegue até o diretório onde você deseja criar o pacote.
2. Crie a pasta "biblioteca" com o comando: `mkdir biblioteca`.
3. Dentro da pasta "biblioteca", crie a subpasta "entidades" com o comando: `mkdir biblioteca/entidades`.
4. Mova o arquivo "Pessoa.java" para a pasta "entidades" com o comando: `mv Pessoa.java biblioteca/
entidades/`.

Passo 5: Compilação do Código

1. No Terminal, navegue até o diretório onde você salvou o arquivo "Pessoa.java".
2. Compile o código com o comando: `javac Pessoa.java`

Passo 6: Execução do Código

Após a compilação bem-sucedida, execute o código com o comando: `java Pessoa`. Você deve receber a saída no seu terminal:

```
Nome: Jess
Livros: [Arquitetura Limpa, Desbravando Java, SOLID para Ninjas]
```

Muito bem, código executado! Temos nossa primeira classe implementada e funcionando.

Ok, agora eu devo várias explicações do que esse código fez.

Vamos entender parte por parte nestas aulas. Preste bastante atenção na explicação porque faremos as demais classes que desenhamos no nosso plano, na parte de exercícios.

Primeira coisa que você deve estar se perguntando é: porque é necessário compilar os arquivos .java para serem executados?

O compilador

A função do compilador em Java é traduzir o código fonte escrito em linguagem Java para um código de nível intermediário chamado bytecode.

Esse processo é essencial para que o código Java seja executado pela Java Virtual Machine (JVM), que é responsável por interpretar o bytecode e executar o programa em diferentes sistemas operacionais e hardware, seguindo o princípio "Escreva uma vez, rode em qualquer lugar" do Java.

A JVM é um componente responsável por interpretar e executar programas Java.

O compilador Java, conhecido como javac, converte o código fonte Java em bytecode, que é uma linguagem de nível intermediário compreendida pela JVM.

Isso permite que os programas Java sejam portáveis e executados em diferentes plataformas sem a necessidade de recompilação, garantindo a interoperabilidade e a independência de plataforma do Java.

Agora que você já sabe porque precisa compilar as classes, vamos entender linha a linha sobre a superclasse Pessoa.

→ Aula 2.5 - O Método Main

Vamos começar falando deste trecho de código, que é super importante!

```
public static void main(String[] args) {  
    String[] livros = {"Arquitetura Limpa", "Desbravando Java", "SOLID para  
Ninjas"};  
    Pessoa pessoa = new Pessoa("Jess", livros);  
    System.out.println("Nome: " + pessoa.getNome());  
    System.out.println("Livros: " + Arrays.toString(pessoa.getLivros()));  
}  
public static void main(String[] args) {
```

Essa é a parte mais importante do programa. É aqui que tudo começa a acontecer. Essa linha diz que existe um método chamado Main() que é público (pode ser acessado de qualquer lugar) e estático (pode ser chamado sem precisar criar um objeto da classe). Dentro desse método, o programa vai executar todas as suas instruções.

```
String[] livros = {"Arquitetura Limpa", "Desbravando Java", "SOLID para  
Ninjas"};
```

Essa linha cria um array de String chamado livros e o inicializa com três títulos de livros. Lembra da definição que fizemos na nossa classe Pessoa? Ela recebe um nome e um array de livros. Então, assim como para fazer uma massa de biscoitos, precisamos ter todos os ingredientes separados e prontos para uso, para montar nosso objeto Pessoa, precisamos passar para a classe Pessoa um nome e um array de livros. Aqui, já criei o meu array.

```
Pessoa pessoa = new Pessoa("Jess", livros);
```

Agora nós criamos um objeto da classe "Pessoa". Esse objeto tem um nome ("Jess") e a lista de livros que criamos anteriormente. Preste atenção que, para criar um novo objeto Pessoa, precisamos passar um nome e uma lista de livros para o construtor. Podemos passar qualquer nome, mas precisamos obrigatoriamente passar um nome e uma lista.

Experimente trocar o nome Jess (meu nome) pelo seu nome. Só não esqueça que, depois de fazer a modificação no arquivo, você deve salvá-lo, compilar novamente. A sequência seria:

```
javac Pessoa.java  
java Pessoa.java
```

Se você não seguir essa sequência, seu programa continuará aparecendo na versão anterior, mesmo se você salvar o arquivo corretamente.

```
System.out.println("Nome: " + pessoa.getNome());
```

Essa linha imprime na tela o nome da pessoa. Ela usa o método `getNome()` da classe "Pessoa" para obter o nome e mostrá-lo.

`System.out.println()` é um método que pertence à classe `System` e é usado para imprimir algo no console. Vamos usar ele bastante para ter certeza que nosso código está funcionando corretamente.

Em "Nome: " + `pessoa.getNome()` concatenou a string "Nome: " com o resultado do método `getNome()` chamado no objeto `pessoa`. Isso significa que estamos juntando a palavra "Nome: " com o nome da pessoa.

Agora, vamos falar especificamente de `getNome`. Em Java, é uma boa prática encapsular os atributos de uma classe, tornando-os privados e fornecendo métodos públicos para acessá-los (métodos getters).

No caso da classe `Pessoa`, o atributo `nome` é privado, o que significa que ele não pode ser acessado diretamente de fora da classe.

Se o atributo `nome` fosse declarado como público (`public String nome;`), você poderia acessá-lo diretamente sem usar o método `getNome()`.

Porém, ao manter o atributo `nome` privado e fornecer o método `getNome()`, você segue o princípio de encapsulamento, que ajuda a controlar o acesso aos dados da classe e a manter a integridade do objeto.

```
System.out.println("Livros: " + Arrays.toString(pessoa.getLivros()));
```

Essa linha imprime na tela a lista de livros da pessoa, usando o método `getLivros()` e a classe `Arrays` para converter o array em uma string.

O método `Arrays.toString()` é um método estático da classe `Arrays` que converte um array em uma representação de string. Ele retorna uma string que representa o conteúdo do array passado como argumento.

Um método estático é um método que pertence à classe, e não a um objeto específico dessa classe. Isso significa que você pode chamar esse método diretamente pelo nome da classe, sem precisar criar um objeto.

O método `getLivros()` é um método da classe `Pessoa` que retorna a lista de livros associada a uma instância específica da classe `Pessoa`.

Bastante coisa, né? Mas não se assuste! E nem se preocupe que vai precisar escrever "muito código" porque, em aplicações modernas usando frameworks, essa complexidade é bastante abstraída. Mas como estamos aprendendo, é muito importante entender todos os porquês e escrever todos os arquivos.

Para que serve a Main?

Como foi dito no começo da explicação do código, a Main é como a porta de entrada do seu programa. É o lugar onde o programa começa a ser executado. Quando você roda um programa Java, o sistema procura pela classe Main e começa a executar a partir dela.

Aqui estão algumas coisas importantes sobre o método main:

- A classe Main possui o método main um método especial e obrigatório em qualquer programa Java.
- Ele é o ponto de partida do seu programa, onde a execução começa.
- Sem o método main, o programa não saberia por onde começar.
- O método main deve ter exatamente a assinatura `public static void main(String[] args)`.
- Os args dentro dos parênteses são os argumentos que você pode passar para o programa quando o executa. No nosso caso, o argumento de execução foi a criação do array, do objeto Pessoa e imprimir os resultados na tela

Por último, vamos falar rapidamente sobre o construtor, um elemento bem importante nas nossas classes:

O que é um construtor?

Um construtor em Java é um tipo especial de método que é chamado automaticamente quando uma nova instância de uma classe é criada. Ele tem o mesmo nome da classe e é responsável por inicializar o objeto recém-criado. Vamos relembrar o nosso construtor feito no exemplo da classe Pessoa:

```
public Pessoa(String nome, String[] livros) {
    this.nome = nome;
    this.livros = livros;
}
```

Diferente de métodos e funções que tem retornos, construtores não têm um tipo de retorno, nem mesmo void, e são invocados usando o operador new. Eles podem receber argumentos como qualquer outro método, e múltiplos construtores podem ser definidos para uma classe usando sobrecarga de métodos.

Quando um construtor é invocado, ele aloca memória para o objeto, inicializa seu conteúdo, invoca o construtor da classe base (se presente), inicializa os membros da classe e executa o restante do corpo do construtor. Essa sequência de ações garante que o objeto tenha as funcionalidades necessárias definidas, garantindo que nenhum campo do objeto tenha um valor arbitrário que possa levar a erros de inicialização.

É possível criar uma classe sem um construtor?

Sim, é possível criar uma classe em Java apenas com métodos getters e setters sem a necessidade de um construtor explícito.

Quando nenhum construtor é definido em uma classe, o compilador Java automaticamente cria um construtor padrão (construtor sem parâmetros) para a classe.

Esse construtor padrão é utilizado para inicializar objetos da classe quando nenhum outro construtor é especificado.

Portanto, se você criar uma classe com métodos getters e setters, mas sem definir nenhum construtor, o compilador Java irá adicionar um construtor padrão implicitamente.

Isso significa que você pode acessar e modificar os atributos da classe por meio dos métodos getters e setters sem a necessidade de um construtor explícito.

Importante lembrar que, se você precisar de um construtor com parâmetros específicos para inicializar os atributos da classe de forma personalizada, será necessário definir esse construtor explicitamente na classe.

Caso contrário, o construtor padrão gerado pelo compilador será suficiente para a inicialização básica dos objetos da classe.

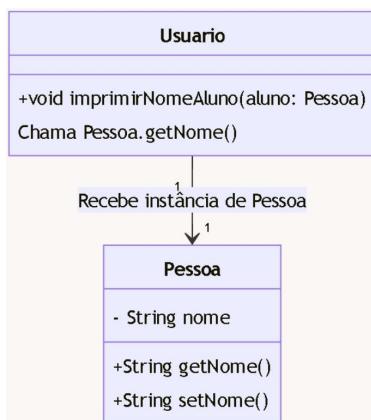
Modificadores de acesso

Você deve ter percebido o uso das palavras `public` tanto no método `main` como na construção da nossa classe. Ele é um modificador de acesso! É uma palavra-chave usada para especificar o nível de visibilidade e acessibilidade de uma classe, método, atributo ou outro membro de um programa. Os modificadores de acesso em Java são: `public`, `private`, `protected` e `default`.

→ Aula 2.6 - Modificador de acesso `public`

Quando você declara algo como "`public`", significa que qualquer parte do seu programa pode acessar e usar esse elemento. É como se fosse uma porta aberta para todo mundo.

Vamos ver um exemplo de aplicação do modificador de acesso `public`. Do lado, deixo o código da classe `Pessoa`, para explicar alguns pontos:



```
public class Pessoa {  
    private String nome;  
  
    public String getNome() {  
        return this.nome;  
    }  
  
    public void setNome(String nome) {  
        this.nome = nome;  
    }  
}
```

Neste diagrama, temos duas classes: `Pessoa` e `Usuario`.

A classe `Pessoa` tem um atributo chamado `nome`, que é privado. Isso significa que apenas a própria classe `Pessoa` pode acessar e modificar esse atributo diretamente. Em diagramas de classes, representamos atributos ou métodos privados colocando o sinal de `-` e os métodos e atributos públicos, usamos o `+`.

No entanto, a classe `Pessoa` também tem dois métodos públicos: `getNome()` e `setNome(String)`. Esses métodos públicos permitem que outras classes, como a `Usuario`, possam obter e definir o valor do atributo `nome` da `Pessoa`.

Mais adiante vamos explicar sobre getters e setters, por enquanto, se concentre nos modificadores de acesso.

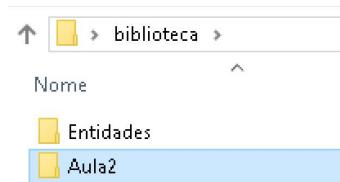
A classe `Usuario` tem um método chamado `imprimirNomeAluno(aluno: Pessoa)`. Esse método recebe uma instância da classe `Pessoa` como parâmetro. Dentro desse método, a classe `Usuario` pode chamar o método `getNome()` da instância de `Pessoa` recebida, para obter o nome do aluno.

Então, mesmo que o atributo `nome` da classe `Pessoa` seja privado, a classe `Usuario` pode acessá-lo indiretamente, usando o método público `Pessoa.getNome()`, já que a classe `Pessoa` é pública (então, podemos acessá-la) e ela tem um método público.

Para ver esse código funcionando na prática, vamos fazer uma demonstração.

Exercício guiado - Criando uma superclasse e uma subclasse

1 - Lembra do exercício de código que fizemos anteriormente? Acesse a sua pasta Biblioteca novamente. Agora, crie uma nova pasta, que esteja no mesmo nível da pasta "Entidades". Vou chamar a minha de Aula2;



Se preferir, você pode acessar pela linha de comando e criar essa pasta com o comando `mkdir`.

2- Agora, você precisar criar um novo arquivo, `Main.java`

Você pode fazer isso por linha de comando com o comando `touch Main.java`

3- Na aula 1, criamos nossa primeira classe em Java, a `Pessoa.java` e, dentro do arquivo, já tínhamos nosso método `Main`. Porém, agora precisaremos criar duas classes e, tem duas formas de fazer isso: criar arquivos separados para cada classe ou deixar tudo num único arquivo.

Criando classes em arquivos separados

Ah, mas não faria mais sentido começar pela forma de um arquivo só e, depois, testar a forma de mais arquivos?

Em tese sim, mas na prática, precisamos aprender um outro modificador de acesso antes, então, vamos começar com a estratégia dos arquivos das classes separados.

4 - Agora, vamos criar mais dois arquivos, dentro do diretório da aula 2:

`Pessoa.java`

`Usuário.java`

5 - Abra o arquivo `Pessoa.java` e copie o seguinte código:

```
public class Pessoa {  
    private String nome;  
  
    public String getNome() {  
        return this.nome;  
    }  
  
    public void setNome(String nome) {  
        this.nome = nome;  
    }  
}
```

Repare que, diferente do primeiro código que criamos, este é bem mais simples e não tem construtor. Inclusive, o construtor é tão importante que, para garantir que você nunca vai esquecer de usa-lo, vamos ver o que acontece se escrever o código sem ele e com ele.

Analizando nosso código linha a linha:

```
public class Pessoa {
```

Aqui estamos criando a classe `Pessoa`, que é como um modelo que define como uma pessoa deve ser representada no programa.

```
private String nome;
```

O atributo nome é declarado como private, o que significa que ele só pode ser acessado dentro da própria classe Pessoa. Outras classes não podem acessar diretamente esse atributo.

```
public String getNome() {
```

Este é um método público chamado getNome(), que retorna o valor do atributo nome. Como é público, outras classes podem chamar este método para obter o nome de uma pessoa.

```
    return this.nome;
```

Aqui estamos retornando o valor do atributo nome quando o método getNome() é chamado. O this é usado para referenciar o atributo nome da própria instância da classe.

```
public void setNome(String nome) {
```

Este é um método público chamado setNome(String nome), que permite definir o valor do atributo nome. Ele recebe um novo nome como parâmetro.

```
this.nome = nome;
```

Aqui estamos atribuindo o novo valor recebido como parâmetro ao atributo nome. O this é usado para referenciar o atributo nome da própria instância da classe.

6 - Agora, Abra o arquivo Usuario.java e escreva este código:

```
public class Usuario {  
    public void imprimirNomeAluno(Pessoa usuario) {  
        System.out.println("Nome do aluno: " + usuario.getNome());  
    }  
}
```

Vamos entender cada linha aqui:

```
public void imprimirNomeAluno(Pessoa aluno) {
```

Método público que imprime o nome do aluno.

```
System.out.println("Nome do aluno: " + aluno.getNome());
```

Acessa o nome do aluno através do método getNome() da classe Pessoa.

Repare que a classe Usuario não acessa diretamente o atributo nome da classe Pessoa, mas utiliza o método público getNome() para obter o nome do aluno de forma controlada e segura. Vamos falar mais disso na próxima aula!

7 - No arquivo Main.java, escreva este código:

```
public class Main {  
    public static void main(String[] args) {  
        Pessoa pessoa = new Pessoa();  
        pessoa.setNome("Jess");  
  
        Usuario usuario = new Usuario();  
        usuario.imprimirNomeAluno(pessoa);  
    }  
}  
  
public class Main {
```

Aqui estamos criando a classe Main, que é a classe principal do programa.

```
public static void main(String[] args) {
```

Este é o método main(), que é o ponto de entrada do programa. Ele é declarado como public e static, o que significa que pode ser acessado e executado diretamente.

```
Pessoa pessoa = new Pessoa();
```

Aqui estamos criando uma nova instância da classe Pessoa. A classe Pessoa não é declarada como public neste arquivo, mas pode ser acessada porque está no mesmo pacote.

```
pessoa.setNome("Jess");
```

Estamos usando o método público setNome() da classe Pessoa para definir o nome da pessoa como "Jess". Mesmo o atributo nome sendo private na classe Pessoa, podemos acessá-lo indiretamente através deste método público.

```
Usuario usuario = new Usuario();
```

Aqui estamos criando uma nova instância da classe Usuario. A classe Usuario também não é declarada como public neste arquivo, mas pode ser acessada porque está no mesmo pacote.

```
usuario.imprimirNomeAluno(pessoa);
```

Estamos chamando o método público imprimirNomeAluno() da classe Usuario, passando a instância de Pessoa que criamos anteriormente como parâmetro. Dentro deste método, a classe Usuario pode acessar o nome do aluno através do método público getName() da classe Pessoa.

Neste código, tanto a classe Pessoa quanto a classe Usuario são declaradas como public, o que significa que elas podem ser acessadas e utilizadas livremente por qualquer outra classe, como a classe Main. Além disso, os métodos setNome() e getName() da classe Pessoa também são públicos, permitindo que a classe Main interaja com o atributo nome.

8 - Agora, compile as classes criadas:

```
javac Main.java Usuario.java Pessoa.java
```

9 - Por fim, execute o arquivo Main:

```
java Main
```

10 - Você deve visualizar a seguinte saída:

```
Nome do aluno: Jess
```

A abordagem de separar cada classe em um arquivo diferente promove a organização, reutilização, colaboração, encapsulamento e facilita a manutenção do código, tornando-o mais legível, modular e escalável. É uma prática recomendada em programação orientada a objetos para criar sistemas mais robustos e fáceis de manter.

→ Aula 2.7 - Modificador de acesso Public, parte 2

É possível definir múltiplas classes em um único arquivo Java, desde que apenas uma delas seja pública e o nome do arquivo corresponda ao nome da classe pública. Para funcionar, a classe pública deve ser a primeira no arquivo e o nome do arquivo deve ser Main.java para corresponder à classe pública.

1-Abra o arquivo Main.java e substitua o conteúdo atual por este código:

```
public class Main {  
    public static void main(String[] args) {  
        Pessoa pessoa = new Pessoa();  
        pessoa.setNome("Alice");  
  
        Usuario usuario = new Usuario();  
        usuario.imprimirNomeAluno(pessoa);  
    }  
  
    static class Pessoa {  
        private String nome;  
  
        public String getNome() {  
            return this.nome;  
        }  
  
        public void setNome(String nome) {  
            this.nome = nome;  
        }  
    }  
  
    static class Usuario {  
        public void imprimirNomeAluno(Pessoa aluno) {  
            System.out.println("Nome do aluno: " + aluno.getNome());  
        }  
    }  
}
```

2 - Compile novamente o arquivo Main. A cada vez que você fizer uma alteração em algum arquivo, não se esqueça de salvar e compilar novamente!

```
javac Main.java
```

3 - Agora, só executar a classe Main:

```
java Main
```

4 - Você deve visualizar a seguinte saída:

```
Nome do aluno: Jess
```

Exatamente a mesma saída do exemplo anterior! Funcionou, não está errado e usou apenas um arquivo. Mas, imagine que você precise crescer este sistema e, em dado momento, você vai ter 30 classes.

Seria muito difícil realizar manutenção, concorda? E se você precisar trabalhar no código com outros colegas, essa operação vai ficar mais difícil ainda.

As classes estarem no mesmo arquivo não é a única diferença em relação a abordagem dos arquivos separados. No nosso exemplo, as classes Pessoa e Usuario estão declaradas como static dentro da classe Main. Isso significa que elas são "especiais" e podem ser acessadas diretamente pelo método main(), sem precisar criar objetos delas.

Então, no método main(), podemos chamar os métodos setNome(), getName() e imprimirNomeAluno() diretamente, sem precisar criar instâncias das classes Pessoa e Usuario. Isso torna o código um pouco mais simples e direto, neste nosso caso.

Mas é importante lembrar que, normalmente, é melhor separar as classes em arquivos individuais, como fizemos antes. Isso torna o código mais organizado e fácil de manter, especialmente em projetos maiores.

Modificador Static

Quando uma classe ou membro (método ou atributo) é declarado como static, isso significa que ele pertence à classe em si, e não a instâncias individuais dessa classe. Aqui estão alguns pontos-chave sobre o static:

- 1. Membros Estáticos:** Quando um membro é declarado como static, ele é compartilhado por todas as instâncias da classe. Isso significa que todas as instâncias da classe terão o mesmo valor para esse membro. Um exemplo comum de membro estático é o método main(), que é estático e serve como ponto de entrada de um programa Java. Ele pode ser chamado diretamente, sem a necessidade de criar uma instância da classe.
- 2. Acesso Direto:** Membros estáticos podem ser acessados diretamente através do nome da classe, sem a necessidade de criar uma instância da classe. Por exemplo, se um método é estático, você pode chamá-lo diretamente usando NomeDaClasse.metodoStatic().
- 3. Variáveis Estáticas:** Variáveis estáticas mantêm seu valor mesmo entre diferentes instâncias da classe. Elas são úteis para armazenar informações que são comuns a todas as instâncias da classe.
- 4. Métodos Estáticos:** Métodos estáticos podem ser chamados sem criar uma instância da classe. Eles são úteis para operações que não dependem do estado de um objeto específico, mas sim da classe em geral.
- 5. Constantes:** Em Java, é comum declarar constantes como static final, o que significa que elas são estáticas e não podem ser alteradas após a inicialização.

Constantes são valores que não mudam durante a execução de um programa e frequentemente declaradas como static final, o que as torna estáticas e imutáveis.

O uso de constantes ajuda a tornar o código mais legível, evita a repetição de valores fixos e facilita a manutenção do código, pois valores constantes podem ser alterados em um único local.

Então, essa abordagem de ter as classes Pessoa e Usuario dentro da classe Main e marcá-las como static é uma opção mais simples, mas nem sempre a melhor. Ela pode ser útil em casos muito simples, mas a separação em arquivos individuais é geralmente a prática mais recomendada.

Resumindo...

Modificador de Acesso: É uma ferramenta em programação que controla quem pode acessar e modificar certos elementos de uma classe, como atributos e métodos.

Modificador de Acesso "public": Permite que atributos e métodos sejam acessados de qualquer lugar no código. No nosso exemplo, permitiu que a classe fosse acessada de qualquer lugar do código.

Separação de Classes em Arquivos: Cada classe em seu próprio arquivo, facilitando a organização, colaboração e manutenção do código, além de promover a modularidade e reutilização de classes.

Classe Main e Método Main: O método main() é o ponto de entrada do programa, permitindo criar instâncias de classes e chamar seus métodos, seguindo os princípios de organização e modularidade.

Modificador Static: Membros estáticos pertencem à classe, não a instâncias, podendo ser acessados diretamente pela classe, sendo úteis para definir constantes e métodos utilitários.

➊ Aula 2.8 - Modificador de acesso private

Quando você declara algo como "private", significa que apenas a própria classe onde esse elemento foi definido pode acessá-lo. É como se fosse uma porta fechada com chave, só a própria classe tem a chave.

Lembram dos exemplo que vimos na aula anterior, da classe Pessoa? Vamos prestar atenção no método private, desta vez:

Agora, se por acaso tentemos acessá-lo diretamente, como em

```
pessoa.name = "Jess"
```

Que eu estou tentando modificar o valor diretamente. Neste caso, vamos receber o seguinte erro:

```
error: nome has private access in Pessoa  
pessoa.nome = "Jess";
```

Esse erro está dizendo exatamente o que vimos na explicação a pouco: apenas a própria classe Pessoa pode acessar e modificar esse atributo diretamente.

Essa é uma forma comum de implementar o encapsulamento em programação orientada a objetos, onde os atributos são mantidos privados, mas métodos públicos são fornecidos para acessá-los e modificá-los de forma controlada.

Já vimos exemplos de herança e encapsulamento! Logo mais, vamos ver exemplos de polimorfismo também e praticar exercícios para identifica-los no código.

Porquê usar encapsulamento?

A prática de encapsular atributos, como o atributo nome da classe Pessoa, mesmo que possa parecer mais fácil acessá-lo diretamente, traz diversos benefícios e é uma boa prática de programação orientada a objetos.

Aqui estão algumas razões pelas quais o encapsulamento é importante:

- 1. Controle de Acesso:** Ao encapsular um atributo como private, você controla quem pode acessá-lo e quem não pode. Isso ajuda a evitar que o atributo seja modificado de forma inesperada ou indevida por outras partes do código.
- 2. Manutenção e Evolução:** Se no futuro você precisar adicionar validações, lógica específica ou alterar a forma como o atributo é armazenado, ter métodos get e set permite fazer essas mudanças sem afetar o restante do código que utiliza a classe.
- 3. Segurança:** O encapsulamento ajuda a garantir a integridade dos dados, evitando que valores inválidos sejam atribuídos ao atributo diretamente.
- 4. Encapsulamento de Comportamento:** Os métodos get e set podem incluir lógica adicional, como validações, cálculos ou notificações, proporcionando um comportamento consistente e controlado ao acessar e modificar o atributo.
- 5. Abstração:** Ao acessar o atributo por métodos get e set, você está abstraindo a implementação interna da classe. Isso permite que a classe mantenha sua lógica interna privada e forneça uma interface pública clara e consistente para interagir com ela.

➡ Aula 2.9 - Getters, Setters

Acabamos de ver alguns conceitos novos: getters e setters.

Getters e Setters

Getters são métodos que permitem acessar (obter) o valor de um atributo privado de uma classe.

Setters são métodos que permitem modificar (definir) o valor de um atributo privado de uma classe.

Getters e setters, juntamente com os modificadores de acesso, são ferramentas importantes para implementar o encapsulamento em programação orientada a objetos. Eles permitem controlar o acesso e a modificação dos dados de uma classe, garantindo a integridade e a flexibilidade do código.

Ao tornar os atributos de uma classe private, você os protege de acesso e modificação direta. Os getters e setters permitem o acesso controlado a esses atributos. Lembra da classe Pessoa?

```
public class Pessoa {  
    private String nome;  
  
    public String getNome() {  
        return this.nome;  
    }  
  
    public void setNome(String nome) {  
        this.nome = nome;  
    }  
}
```

Aqui, temos um exemplo bem fácil para entender a aplicação de getters e setters, afinal, esses métodos que permitem o acesso controlado a propriedade privada nome.

Um detalhe importante é que, nos métodos setters, você pode adicionar lógica para validar os valores que serão atribuídos aos atributos, garantindo a integridade dos dados. Vamos ver isso na prática:

Exercício Guiado: Getters e Setters

1 - Abra sua classe Pessoa e escreva este código:

```
public class Pessoa {  
    private int idade;  
  
    public int getIdade() {  
        return this.idade;  
    }  
  
    public void setIdade(int idade) {  
        if (idade >= 0 && idade <= 120) {  
            this.idade = idade;  
        } else {  
            System.out.println("Idade invalida. A idade deve estar entre 0 e 120  
anos.");  
        }  
    }  
}
```

2 - Agora, altere a classe Main com o seguinte código:

```
public class Main {  
    public static void main(String[] args) {  
        Pessoa pessoa = new Pessoa();  
        pessoa.setIdade(150); // Tentando definir uma idade inválida  
  
        System.out.println("Idade da pessoa: " + pessoa.getIdade()); // A idade  
        não foi alterada devido à validação  
    }  
}
```

3 - Compile ambas as classes e execute o arquivo Main, como já fizemos nos exercícios anteriores.

4 - A saída desse código será:

```
Idade inválida. A idade deve estar entre 0 e 120 anos.  
Idade da pessoa: 0
```

Neste exemplo simplificado, a classe Pessoa possui um atributo idade e um método setIdade(int idade) que valida se a idade está dentro do intervalo de 0 a 120 anos.

Se a idade fornecida estiver fora desse intervalo, uma mensagem de erro é exibida e a idade não é alterada. Isso garante que apenas valores válidos sejam atribuídos ao atributo idade.

Os modificadores de acesso, como private, public, protected, etc., controlam a visibilidade e o acesso aos elementos de uma classe (atributos e métodos).

Ao declarar um atributo como private, você o protege de acesso direto fora da classe. Isso torna os getters e setters essenciais para interagir com esse atributo de forma controlada.

Usar private para atributos e public para métodos getters e setters é uma prática comum que segue o princípio de encapsulamento.

Interface

Em programação, uma interface é como um contrato que define quais ações uma classe pode realizar. É como um conjunto de regras que uma classe deve seguir se quiser ser considerada de um determinado tipo.

Imagine que você tem um controle remoto para a sua TV. Esse controle remoto tem botões como "ligar", "desligar", "aumentar volume" e "diminuir volume".

Esses botões representam a interface do controle remoto. Você não precisa saber como cada botão funciona por dentro, apenas precisa saber o que cada botão faz.

Da mesma forma, em programação, uma interface define quais métodos uma classe deve implementar, mas não especifica como esses métodos devem ser implementados.

Isso permite que diferentes classes possam seguir a mesma interface, garantindo que tenham comportamentos semelhantes, mesmo que a implementação interna seja diferente.

Ou seja, qualquer classe que usar esse método get ou set, vai conseguir fazer do mesmo jeito.

Por exemplo, se tivermos uma interface Animal com um método fazerSom(), qualquer classe que implemente essa interface deve ter um método fazerSom(), mas cada classe (como Cachorro, Gato, Pato) pode implementar esse método de forma diferente para representar o som que cada animal faz.

Vamos voltar novamente neste assunto de interfaces, numa aula só deste assunto, mais a diante. Aqui, a ideia foi apenas apresentar o conceito.

Resumindo...

Métodos Getter e Setter: São métodos especiais usados para acessar (getter) e modificar (setter) os valores dos atributos de uma classe de forma controlada, mantendo a integridade dos dados e seguindo o princípio de encapsulamento.

Interface: É como um contrato que define quais ações uma classe deve realizar, sem especificar como essas ações são implementadas.

Permite que diferentes classes sigam a mesma interface, garantindo comportamentos semelhantes, mas com implementações diferentes.

Encapsulamento: Manter os detalhes internos de uma classe privados e fornecer uma interface pública para interagir com a classe, promovendo a modularidade, reutilização e evolução do código.

→ Aula 2.10 - Modificador de Acesso Protected e Extends

Protected

Quando você declara algo como "protected", significa que a própria classe e as subclasses (que herdam dessa classe) podem acessar esse elemento. É como se fosse uma porta com uma fechadura especial, apenas a classe e suas "filhas" têm a chave.

No contexto da nossa classe de exemplo, podemos usar o modificador protected para garantir que apenas classes que herdam a classe Pessoa accessem a propriedade Nome.

Herdar propriedades de uma superclasse, como já vimos outros exemplos, é chamado de Herança. Porém, neste caso, como vamos usar uma propriedade protected, vamos precisar usar também a palavra-chave extends, senão, não vamos conseguir acessar.

Extends

O extends é usado para criar uma relação de herança entre classes. Quando uma classe B estende (herda de) uma classe A, isso significa que a classe B (subclasse) recebe todos os atributos e métodos da classe A (superclasse).

A principal diferença entre usar a palavra-chave extends e não usá-la está na relação de herança entre as classes.

Quando uma classe estende outra usando extends, a subclasse herda todos os membros não-privados da superclasse, sejam eles públicos, protegidos ou padrão (package-private).

Isso significa que a subclasse pode acessar e utilizar esses membros herdados, desde que tenham um modificador de acesso que permita essa interação.

Vamos ver na prática essa dupla trabalhando junto:

Exercício Guiado: Atributo protected

1 - Abra a classe Pessoa e altere a propriedade nome para protected e acrescente a propriedade idade:

```

public class Pessoa {
    protected String nome;
    protected int idade;

    public String getNome() {
        return this.nome;
    }

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

    public int getIdade() {
        return this.idade;
    }

    public void setIdade(int idade) {
        this.idade = idade;
    }
}

```

2 - Na classe Usuario, faça as seguintes adições:

```

public class Usuario {
    private String matricula;

    public String getMatricula() {
        return this.matricula;
    }

    public void setMatricula(String matricula) {
        this.matricula = matricula;
    }

    public void imprimirInformacoes() {
        System.out.println("Nome: " + getNome());
        System.out.println("Idade: " + getIdade());
        System.out.println("Matricula: " + getMatricula());
    }
}

```

3 - Por fim, faça a seguinte alteração na classe Main:

```

public class Main {
    public static void main(String[] args) {
        Usuario usuario = new Usuario();
        usuario.setNome("Jess");
        usuario.setIdade(21);
        usuario.setMatricula("123456");
        usuario.imprimirInformacoes();
    }
}

```

4 - Não se esquema de compilar os arquivos e, a seguir, executar a classe Main:

```

javac Main.java Usuario.java Pessoa.java
java Main

```

5 - Se você escreveu tudo certinho, vai visualizar o seguinte erro:

```
Main.java:4: error: cannot find symbol
    usuario.setNome("Jess");
          ^
  symbol:   method setNome(String)
  location: variable usuario of type Usuario
Main.java:5: error: cannot find symbol
    usuario.setIdade(21);
          ^
  symbol:   method setIdade(int)
  location: variable usuario of type Usuario
Usuario.java:13: error: cannot find symbol
    System.out.println("Nome: " + getNome());
          ^
  symbol:   method getNome()
  location: class Usuario
Usuario.java:14: error: cannot find symbol
    System.out.println("Idade: " + getIdade());
          ^
  symbol:   method getIdade()
  location: class Usuario
4 errors
```

Mas ei, não se assuste! Vamos analizar o erro.

O compilador está procurando por um método `setNome(String)` na classe `Usuario`, mas não o encontra, pois ele não está definido lá.

Isso é o que o erro "cannot find symbol" significa: o compilador não pode encontrar o símbolo (ou seja, o método) que você está tentando usar.

E de fato, os métodos `getNome()` e `setNome()` não estão na classe `Usuario`, mas sim, na classe `Pessoa`. Mesmo usando getter e setter, como as propriedades `nome` e `idade` são `protected`.

A classe `Usuario` não pode acessá-las diretamente, pois não há uma relação de herança entre as duas classes.

Para resolver esse problema, a classe `Usuario` precisa estender a classe `Pessoa` usando a palavra-chave `extends`.

Dessa forma, a classe `Usuario` herdará todos os membros não-privados (públicos e protegidos) da classe `Pessoa`, incluindo os métodos `getNome()` e `setNome()`.

Assim, a classe `Usuario` poderá acessar e utilizar esses métodos herdados diretamente.

6 - Agora, faça uma única modificação na classe `usuario`, na primeira linha e deixe o restante do código como já estava antes:

```
public class Usuario extends Pessoa {
```

Compile os arquivos modificados e execute novamente o nosso arquivo `main`. Você vai ver essa saída no seu terminal:

Nome: Jess

Idade: 21

Matrícula: 123456

Adicionando "`extends Pessoa`" a classe `Usuario` estende a classe `Pessoa`, herdando seus métodos `getNome()`, `setNome()`, `getIdade()` e `setIdade()`.

Dessa forma, a classe `Usuario` pode acessar e utilizar esses métodos herdados diretamente, resolvendo o erro anterior.

Resumindo...

Métodos Getter e Setter: Os métodos getter e setter são frequentemente utilizados em conjunto com o modificador protected para permitir o acesso controlado aos atributos de uma classe por suas subclasses, garantindo a integridade dos dados.

Modificador Protected: O modificador protected é essencial para permitir que os membros de uma classe sejam acessíveis pelas subclasses, facilitando a herança e a extensão de funcionalidades sem expor detalhes internos sensíveis.

Herança (extends): A palavra-chave extends estabelece a relação de herança entre classes, permitindo que uma subclass herde os membros não-privados da superclasse, como os atributos protegidos, e amplie ou modifique seu comportamento.

Encapsulamento: O encapsulamento, aliado ao modificador protected, ajuda a manter a integridade dos dados ao restringir o acesso direto aos atributos de uma classe, incentivando o uso de métodos getter e setter para interações controladas.

Reuso de Código: A herança, facilitada pelo extends, promove o reuso de código ao permitir que as subclasses herdem e reutilizem os membros não-privados da superclasse, evitando a duplicação de implementações.

→ Aula 2.11 - Modificador padrão

Chegamos no nosso ultimo modificador de acesso!

Quando você não coloca nenhum modificador de acesso, o elemento fica com acesso "padrão" (também conhecido como pacote-private), que significa que apenas as classes dentro do mesmo "pacote" (grupo de classes relacionadas) podem acessá-lo.

É como se fosse uma porta com uma fechadura que só as classes do mesmo "prédio" têm a chave, ou seja, esse nível de acesso permite que os membros (atributos, métodos, classes) sejam acessíveis apenas dentro do mesmo pacote.

Vamos adicionar o modificador de acesso padrão em um exemplo com as classes Pessoa e Usuario que criamos anteriormente e entender na prática:

Exercício Guiado: Aplicando modificadores padrão

1 - Vamos alterar nossa classe Pessoa. Repare que tiramos os getters e setters de nome e idade:

```
class Pessoa {  
    protected String nome;  
    protected int idade;  
  
    String getPais() {  
        return "Brasil";  
    }  
}
```

2 - Agora, vamos fazer mais uma alteração na classe Usuário:

```
class Usuario extends Pessoa {  
    private String matricula;  
  
    void imprimirInformacoes() {  
        System.out.println("Nome: " + nome);  
        System.out.println("Idade: " + idade);  
        System.out.println("Matrícula: " + matricula);  
        System.out.println("País: " + getPais());  
    }  
}
```

3 - Não faremos modificações na Main, o código se encontra da seguinte forma:

```
public class Main {  
    public static void main(String[] args) {  
        Usuario usuario = new Usuario();  
        usuario.nome = "Jess";  
        usuario.idade = 21;  
        usuario.matricula = "123456";  
        usuario.imprimirInformacoes();  
    }  
}
```

4 - Depois de compilar os arquivos modificados e executar a Main, vamos ter essa saída no terminal:

```
Nome: Jess  
Idade: 21  
Matricula: 123456  
Pais: Brasil
```

Neste exemplo, removemos explicitamente o modificador de acesso (public, protected, private) dos membros das classes Pessoa e Usuario, tornando-os de acesso padrão. Isso significa que esses membros só podem ser acessados por outras classes no mesmo pacote.

O método getPaís() na classe Pessoa foi definido com o modificador de acesso padrão, tornando-o acessível apenas dentro do mesmo pacote.

Na classe Usuario, o método imprimirInformações() acessa o método getPaís() da classe Pessoa, demonstrando o acesso a membros de acesso padrão entre classes no mesmo pacote.

Observe que a classe Main consegue acessar os atributos nome e idade da classe Pessoa, bem como o método getPaís(), pois esses membros têm acesso padrão e estão no mesmo pacote.

No entanto, se tentarmos mover a classe Main para um pacote diferente, ela não conseguirá acessar os membros de acesso padrão das classes Pessoa e Usuario, pois elas estarão em um pacote diferente.

5 - Crie um novo diretório dentro de Biblioteca, com o nome que preferir. Vamos remove-lo depois. Mova nosso arquivo Main para dentro dessa pasta e tente executá-la. Veja seu terminal:

```
Main.java:3: error: cannot find symbol  
    Usuario usuario = new Usuario();  
           ^  
      symbol:   class Usuario  
      location: class Main  
Main.java:3: error: cannot find symbol  
    Usuario usuario = new Usuario();  
           ^  
      symbol:   class Usuario  
      location: class Main  
2 errors
```

E você vai encontrar esses dois erros porque a classe Usuario não está visível para a classe Main nesse novo local.

Quando você move a classe Main para um diretório diferente, ela passa a estar em um pacote diferente do pacote onde estão as classes Pessoa e Usuario.

Isso significa que a classe Main não consegue acessar as classes Pessoa e Usuario, pois elas têm acesso padrão (package-private) e, portanto, só podem ser acessadas por classes no mesmo pacote.

Resumindo...

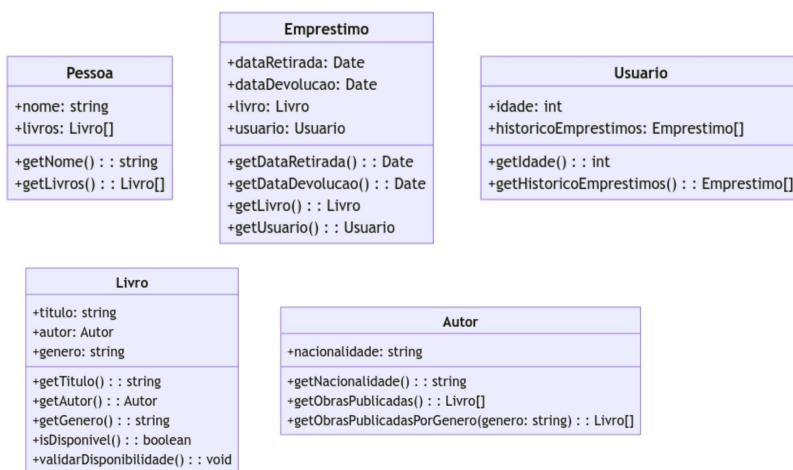
Modificador de Acesso Padrão: O modificador de acesso padrão (ou pacote-private) restringe o acesso aos membros de uma classe apenas ao mesmo pacote em que estão localizados. Esses membros não são visíveis fora do pacote, limitando sua acessibilidade a classes externas.

Importância do Encapsulamento: O uso do modificador de acesso padrão contribui para o encapsulamento, mantendo a coesão e a organização do código ao restringir o acesso aos membros da classe, promovendo a modularidade e a segurança do sistema.

Limitações e Boas Práticas: O modificador de acesso padrão é uma ferramenta importante para controlar a visibilidade dos membros de uma classe, evitando o acesso não autorizado e garantindo a integridade do código.

Modulo 2 - Exercícios

1 - Com base no desenho que realizamos no exercício do módulo 1, agora que você já praticou bastante a criação de classes, modificadores de acesso e métodos, crie as classes restantes para o nosso projeto, sendo elas:



Relações entre as classes:

1 Autor é 1 pessoa

1 usuário é 1 pessoa

1 Livro possui 1 Autor principal.

Cada Autor pode escrever N Livros.

Cada Livro pode ser emprestado N vezes mas não pode estar em mais de 1 empréstimo por vez

Cada Usuário pode realizar N Empréstimos por vez.

Cada Empréstimo envolve pelo menos 1 Livro e apenas 1 Usuário.

Lembre-se de usar os modificadores de acesso, respeitar a hierarquia de superclasses e subclasses, getters, setters e usar tudo que aprendemos nesta aula para começarmos a dar vida para o nosso projeto!

Ao executar a sua Main, ela deve trazer o seguinte resultado:

```
O livro não está disponível
Livro: Java for Beginners
Autor: Jessica Felix
Genero: Tecnologia
Usuario: Lucas Rafael
Idade: 25
Data de Retirada: Wed May 08 23:37:21 BRT 2024
Data de Devolucao: Wed May 08 23:37:21 BRT 2024
```

Voce pode alterar o nome do livro, Autor, tipo do gênero, usuário, idade e as datas. Mas o importante é que a sua Main imprima os resultados.

Instruções para submissão:

Na pasta modulo2 do seu repositório no Github coloque os seguintes arquivos:

1. Pessoa.java
2. Usuario.java
3. Autor.java
4. Livro.java
5. Emprestimo.java
6. Main.java

Boa sorte!

Módulo 3 - Testes Unitários em Java com JUnit e Construtores

→ Aula 3.1 - Testes unitários

Aprendemos a escrever nossas classes e fizemos um exercício mais complexo no módulo passado! Lembram de uma sessão que tivemos no primeiro módulo, sobre ciência e matemática?

Ao focar em funções com resultados mensuráveis, podemos identificar e corrigir falhas, garantindo a confiabilidade do software que desenvolvemos.

Precisamos sempre garantir que nosso código é confiável e a melhor forma de fazer isso é incluindo testes a eles. Os testes ajudam a garantir que tudo continua funcionando após uma alteração, além de acrescentar mais confiabilidade à nossa entrega, afinal, podemos mostrar que nosso código não está falhando nos pontos de comportamento esperados que mapeamos.

Mas lembre-se: não é possível provar que um sistema está certo, conseguimos provar que ele não está falhando nos pontos que identificamos.

Os testes unitários são uma prática fundamental na programação para garantir que partes individuais do código funcionem conforme o esperado.

Em Java, o framework mais popular para testes unitários é o JUnit. Também existem outros tipos de testes, como testes de integração, testes de carga, testes de usabilidade, mas, para começar, vamos escrever testes unitários, o tipo mais simples de teste.

Ao escrever testes unitários, criamos métodos que verificam o comportamento de pequenas unidades de código, como métodos ou classes, isoladamente. Isso ajuda a identificar erros e garantir que o código se comporte como esperado desde o início do desenvolvimento.

Os testes unitários são essenciais para facilitar a identificação de bugs, permitir a refatoração do código com segurança e melhorar a qualidade do código.

Mesmo que inicialmente possa parecer demorado, escrever testes unitários pode economizar tempo no desenvolvimento ao evitar problemas futuros e garantir a manutenção do código.

Até agora, pudemos executar nossos arquivos java direto do terminal. Porém, como vamos precisar de mais funcionalidades para escrever nossos testes, vamos precisar de uma ferramenta chamada IDE(Integrated Development Environment ou Ambiente de Desenvolvimento Integrado).

Embora seja possível escrever testes em um editor de texto simples e executá-los via terminal usando o comando javac para compilar e java para executar, uma IDE oferece recursos específicos para facilitar a criação, execução e gerenciamento de testes unitários.

As IDEs geralmente têm integração com frameworks de teste como o JUnit, fornecendo funcionalidades como geração automática de testes, execução de testes com um clique, visualização de resultados e relatórios detalhados. Isso torna o processo de teste mais eficiente e produtivo, especialmente para quem está aprendendo a escrever testes pela primeira vez.

→ Aula 3.2 - Preparando o Ambiente para testes

Usando uma IDE

Você pode escolher entre vários ambientes de desenvolvimento para Java, como IntelliJ IDEA, Eclipse, NetBeans ou VS Code. A escolha vai da sua preferência. No material complementar você vai encontrar instruções para baixar as IDEs. Escolha a que você se adaptar melhor.

Caso tenha escolhido usar o VS Code, na parte de Extensions, instale o "Extension Pack for Java + Spring" da Loiane Groner, que tem algumas extensões muito úteis para trabalhar com Java. Instale também o Test Runner for Java da Microsoft, que vai permitir que possamos rodar e debuggar nossos testes.

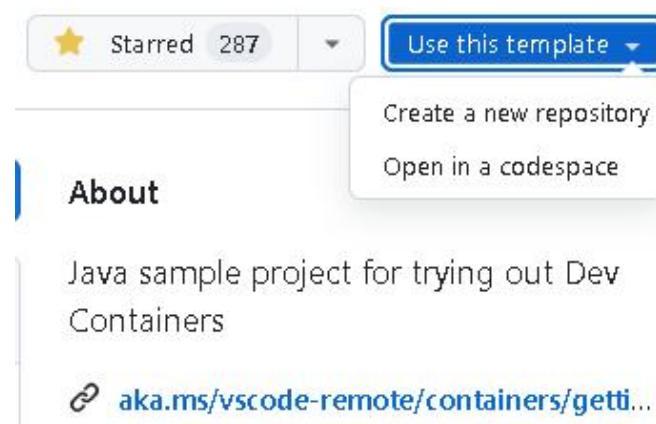
É mais comum usar IntelliJ para desenvolvimento Java, mas nos exemplos, usaremos o Code Spaces do Github, pois é um desenvolvimento de baixa complexidade e não vamos precisar fazer muitas configurações de ambiente, pois já está quase tudo pronto.

Você também pode usar o Code Spaces do Github, basta criar um repositório e importar nossos arquivos vistos em aula. Para preparar o seu ambiente no Codespaces de uma forma mais rápida, você pode usar um template pronto.

Usando o Code Spaces

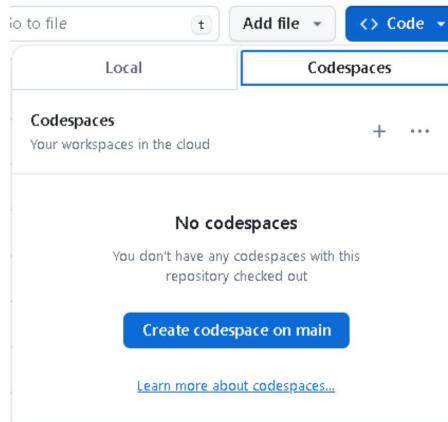
Nessa sessão, é importante que você já tenha conhecimentos mínimos sobre Git e Github, e já tenha uma conta no Github criada. Caso não, você pode ver a documentação do próprio Github.

Caso já esteja com familiaridade, basta acessar o template [vscode-remote-try-java](#)



Basta clicar no botão Use this template e depois, Create a new repository.

Você vai criar um repositório normalmente. Após essa criação, selecione a opção code spaces e crie um novo code space:



Depois de abrir o code spaces, você vai encontrar um projeto Java já completo, pronto para usarmos.

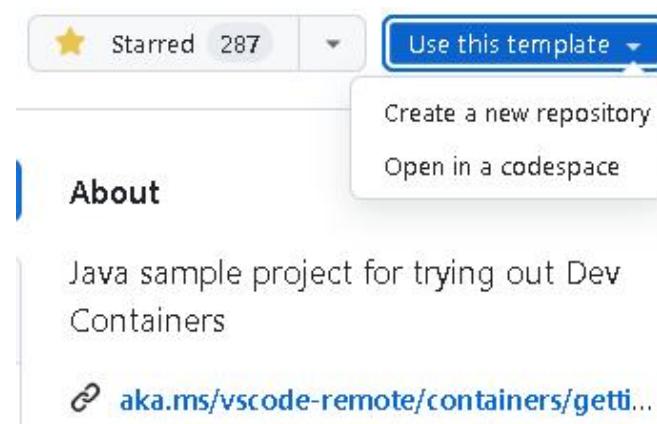
É mais comum usar IntelliJ para desenvolvimento Java, mas nos exemplos, usaremos o Code Spaces do Github, pois é um desenvolvimento de baixa complexidade e não vamos precisar fazer muitas configurações de ambiente, pois já está quase tudo pronto.

Você também pode usar o Code Spaces do Github, basta criar um repositório e importar nossos arquivos vistos em aula. Para preparar o seu ambiente no Codespaces de uma forma mais rápida, você pode usar um template pronto.

Usando o Code Spaces

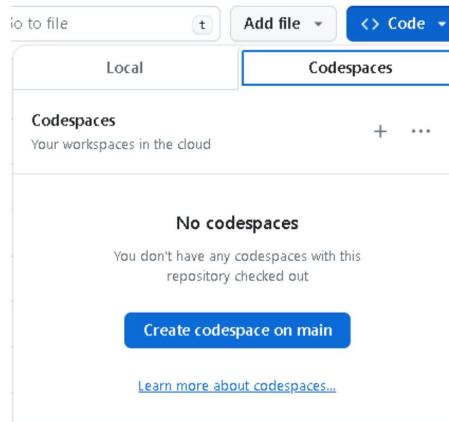
Nessa sessão, é importante que você já tenha conhecimentos mínimos sobre Git e Github, e já tenha uma conta no Github criada. Caso não, você pode ver a documentação do próprio Github.

Caso já esteja com familiaridade, basta acessar o template [vscode-remote-try-java](#)

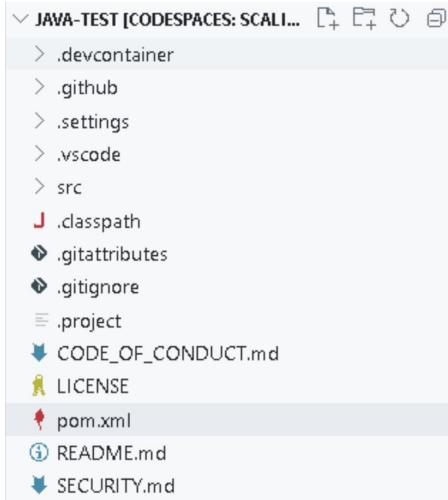


Basta clicar no botão Use this template e depois, Create a new repository.

Você vai criar um repositório normalmente. Após essa criação, selecione a opção code spaces e crie um novo code space:



Depois de abrir o code spaces, você vai encontrar um projeto Java já completo, pronto para usarmos.



Se abrirmos o arquivo pom.xml, vamos localizar por volta da linha 9 ou 10, este trecho de código:

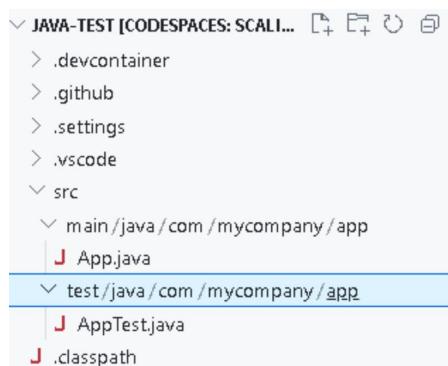
```
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.13.2</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```

Ele mostra que o JUnit já está instalado e que está na versão 4.13.2

Agora, vamos precisar apenas importar nossos arquivos e começar a trabalhar.

Importando nossos arquivos para o repositório do Github Spaces

No menu a esquerda, procure pela pasta "src", que contém um único arquivo em Java. Você vai localizar também o arquivo de testes, logo a baixo:



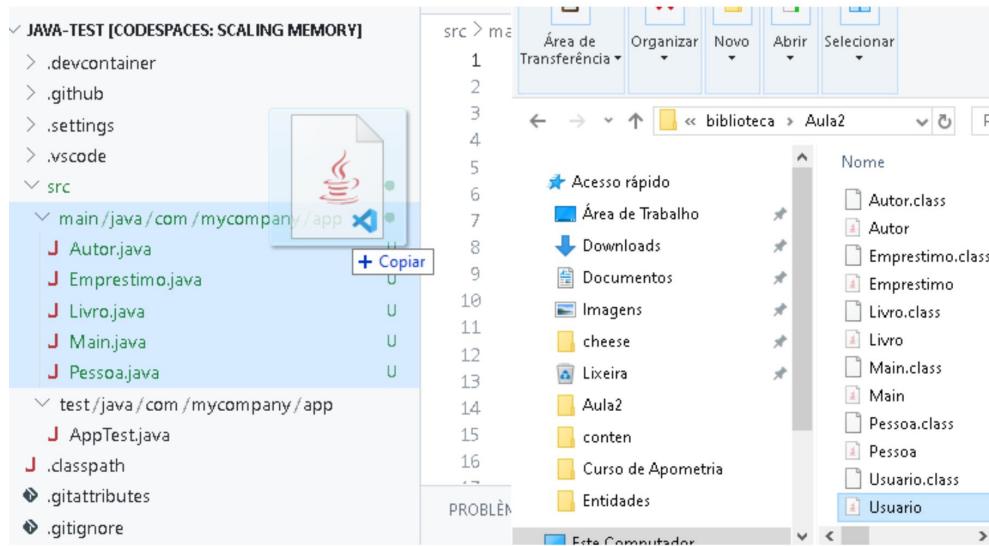
Dentro do diretório app, abra o arquivo App.java. Voce vai encontrar o seguinte código:

```
package com.mycompany.app;

public class App {
    public static void main(String[] args) {
        System.out.println("Hello Remote World!");
    }
}
```

Repare na primeira linha, que tem o nome do package. Vimos nos módulos anteriores como os pacotes funcionam, então, em todos os nossos arquivos que vamos importar, das classes Pessoa, Usuario e todas as demais, vamos sempre colocar o package.

Você pode simplesmente arrastar seus arquivos para dentro da pasta App, no Code Spaces:



Não esqueça de incluir o package com.mycompany.app; em todos os arquivos que foram importados. Ele deve ficar na primeira linha da classe, conforme o exemplo:

```
package com.mycompany.app;

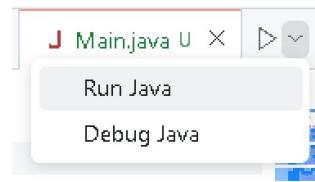
import java.util.Date;

public class Main {
    public static void main(String[] args) {
```

Agora, basta compilar todos os arquivos:

```
javac Autor.java Emprestimo.java Livro.java Main.java Usuario.java Pessoa.java
```

E claro, vamos executar nosso projeto para ter certeza que tudo está funcionando normalmente. No topo da sua tela, no canto direito, você vai encontrar uma seta apontando para a direita. Clique nela e depois, na opção Run Java:



Agora, no terminal, você deve visualizar a seguinte saída:

```
O livro não está disponível.
Livro: Java for Beginners
Autor: Jessica Felix
Gênero: Tecnologia
Usuário: Lucas Rafael
Idade: 25
Data de Retirada: Thu May 09 23:48:45 UTC 2024
Data de Devolução: Thu May 09 23:48:45 UTC 2024
```

Nosso projeto está funcionando normalmente e podemos começar a escrever nossos testes!

→ Aula 3.3 - Criando o primeiro teste unitário

Lembra do diretório Test, logo a baixo de Main? Nele, já temos um arquivo modelo, o AppTest.java:

```
package com.mycompany.app;
import org.junit.Test;
import static org.junit.Assert.assertTrue;
public class AppTest {
    public AppTest() {
    }

    @Test
    public void testApp() {
        assertTrue(true);
    }

    @Test
    public void testMore() {
        assertTrue(true);
    }
}
```

Vamos entender linha a linha o que está acontecendo neste arquivo:

```
package com.mycompany.app;
```

Em Java, um pacote é uma forma de organizar classes relacionadas em um mesmo espaço de nomes. Isso ajuda a evitar conflitos de nomes entre classes com o mesmo nome, mas em pacotes diferentes.

Neste caso, a classe AppTest está definida dentro do pacote "com.mycompany.app".

```
import org.junit.Test;
```

Em Java, o import é usado para trazer classes, interfaces ou métodos de outros pacotes para dentro do seu código. Neste caso, estamos importando a anotação @Test da biblioteca JUnit.

As anotações em Java são usadas para adicionar metadados a elementos do código, como classes, métodos ou variáveis.

```
import static org.junit.Assert.assertTrue;
```

Aqui estamos importando estaticamente o método assertTrue da classe Assert do JUnit.

Isso significa que podemos usar o método assertTrue diretamente, sem precisar referenciar a classe Assert.

```
public class AppTest {
```

Em Java, uma classe é um modelo ou blueprint que define as propriedades e comportamentos de um objeto. Neste caso, AppTest é uma classe pública que contém os métodos de teste.

```
    public AppTest() { }
```

Este é o construtor da classe AppTest. Um construtor é um método especial usado para inicializar objetos de uma classe quando eles são criados.

Neste caso, o construtor está vazio, pois não há necessidade de inicializar nada específico para os testes.

@Test:

Esta é a anotação que marca um método como um teste unitário. Quando o framework JUnit executa os testes, ele procura por métodos marcados com @Test e os executa.

```
    public void testApp() {
```

Este é um método público chamado testApp que não retorna nada (void).

Lembre-se que em Java, os métodos são blocos de código nomeados que podem ser chamados para executar uma tarefa específica.

```
assertTrue(true);
```

O método `assertTrue` é fornecido pela classe `Assert` do JUnit. Ele verifica se a expressão passada como argumento é verdadeira. Neste caso, estamos verificando se `true` é verdadeiro, o que sempre será verdade.

```
public void testMore() {
```

Este é outro método público chamado `testMore` que também não retorna nada (`void`). Ele também é marcado como um teste unitário pelo JUnit.

```
assertTrue(true);
```

Novamente, estamos usando o método `assertTrue` do JUnit para verificar se `true` é verdadeiro.

Vamos começar o primeiro teste, para a classe `Pessoa`.

1- Dentro do diretório `/workspaces/java-test/src/test/java/com/mycompany/app` crie um arquivo chamado `PessoaTest.java`, que vai estar gerado desta forma, quando você abrir:

```
package com.mycompany.app;  
  
public class PessoaTest {  
  
}
```

Para criar testes unitários para a classe `Pessoa` em JUnit, a classe de testes que acabamos de criar, a `PessoaTest`, deve conter métodos que testem as diferentes funcionalidades da classe `Pessoa`

Vamos começar pensando em quais os métodos que temos na nossa classe. São eles:

- 1. `getNome()`:** Este é um método público que retorna o nome da pessoa. Ele não recebe nenhum parâmetro e retorna uma `String`.
- 2. `setNome(String nome)`:** Este é um método público que permite definir o nome da pessoa. Ele recebe uma `String` como parâmetro, que será o novo nome da pessoa.
- 3. `getLivros()`:** Este é um método público que retorna a lista de livros da pessoa. Ele não recebe nenhum parâmetro e retorna um array de objetos da classe `Livro`.
- 4. `setLivros(Livro[] listaLivros)`:** Este é um método público que permite definir a lista de livros da pessoa. Ele recebe um array de objetos da classe `Livro` como parâmetro, que será a nova lista de livros da pessoa.

Agora, vamos escrever um teste para cada método, sempre pensando em como podemos chamar o método e confirmar que ele está funcionando.

Vamos começar escrevendo nosso método de testar o método `getNome!` Vamos pensar etapa a etapa o que precisaríamos fazer antes de começar a escrever. Para testar o método `getName`, vamos precisar:

Criar uma instância da classe `Pessoa`

Nesta etapa, você cria um objeto da classe `Pessoa` usando o construtor padrão ou qualquer outro construtor disponível. Isso é feito para ter um objeto no qual você possa testar o comportamento do método `getNome`.

Vamos escrevendo nosso código aos poucos agora, pensando nesta primeira parte:

```

package com.mycompany.app;

import org.junit.Test;
import static org.junit.Assert.*;

public class PessoaTest {
    @Test
    public void testGetNome() {
        // Cria uma instância da classe Pessoa
        Pessoa pessoa = new Pessoa();
    }
}

```

Atribuir um nome para a pessoa

Após criar a instância da classe Pessoa, você chama o método setNome para atribuir um nome específico à pessoa. Isso simula o cenário em que você define um nome para a pessoa.

```

package com.mycompany.app;

import org.junit.Test;
import static org.junit.Assert.*;

public class PessoaTest {
    @Test
    public void testGetNome() {
        // Cria uma instância da classe Pessoa
        Pessoa pessoa = new Pessoa();

        // Atribuir um nome para a pessoa
        pessoa.setNome("Jess");
    }
}

```

➊ Aula 3.4 - Verificar se o nome retornado por `getNome()` é o mesmo que foi atribuído

Nesta etapa, você chama o método getNome para obter o nome da pessoa e armazena esse valor em uma variável. Em seguida, você usa um método de assert (como assertEquals) para comparar o nome retornado pelo método getNome com o nome que você atribuiu anteriormente.

Se os nomes forem iguais, o teste passa, indicando que o método getNome está retornando o nome corretamente. Caso contrário, o teste falha, o que significa que há um problema com a implementação do método getNome.

```

package com.mycompany.app;

import org.junit.Test;
import static org.junit.Assert.*;

public class PessoaTest {

    @Test
    public void testGetNome() {
        // Cria uma instância da classe Pessoa
        Pessoa pessoa = new Pessoa();

        // Define um nome para a pessoa
        pessoa.setNome("Jess");

        // Verifica se o nome retornado é o mesmo que foi setado
        assertEquals("Jess", pessoa.getNome());
    }
}

```

Eu imagino que você deve estar se perguntando: Como vou saber qual método do JUnit vou precisar chamar para os testes que eu quero fazer? Onde vou achar essa informação?

Você pode consultar a documentação do JUnit, mas, também pode seguir este resumo de métodos mais usados, para começar a se acostumar com os cenários de teste.

Lista de métodos mais comuns e mais usados no JUnit para testes unitários

assertEquals: Use este método quando quiser verificar se um valor ou objeto retornado por um método é igual ao valor esperado. É o método mais comumente usado.

assertTrue e assertFalse: Use esses métodos quando quiser verificar se uma condição booleana é verdadeira ou falsa, respectivamente. Por exemplo, testar se um método retorna true ou false corretamente.

assertNull e assertNotNull: Use esses métodos quando quiser verificar se um objeto é nulo ou não nulo, respectivamente. Útil para testar se um método retorna o objeto correto ou se lança uma exceção quando deveria retornar nulo.

assertArrayEquals: Use este método quando quiser verificar se um array retornado por um método é igual ao array esperado. Útil para testar métodos que retornam arrays.

assertThrows: Use este método quando quiser verificar se um método lança uma exceção específica. Útil para testar se um método lança a exceção correta em determinadas circunstâncias.

Alguns outros métodos úteis são:

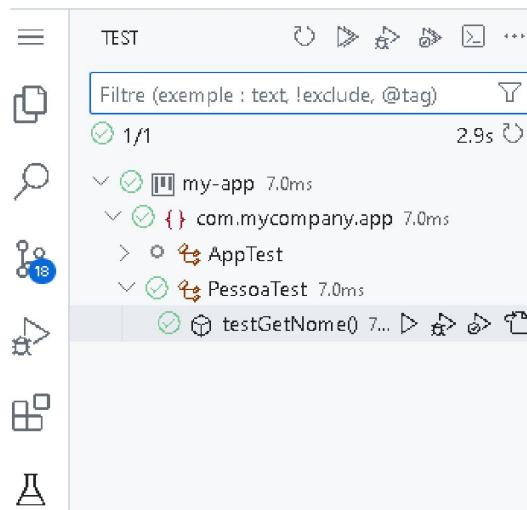
assertNotEquals: Verifica se dois valores ou objetos são diferentes.

assertSame e assertNotSame: Verifica se dois objetos são a mesma instância.

Comece usando os métodos **assertEquals**, **assertTrue**, **assertFalse**, **assertNull** e **assertNotNull**, pois eles cobrem a maioria dos cenários de teste comuns. À medida que você escrever mais testes, ficará mais familiarizado com os outros métodos e saberá quando usá-los.

➊ Aula 3.5 - Rodando o teste unitário

A execução do teste no Code Spaces é bem simples! Basta clicar com o botão direito do mouse sobre o nosso arquivo de testes e selecionar a opção "Executar os testes do arquivo atual". Você também pode buscar pelo símbolo de um becker, que tem toda a informação do teste:



Se você clicar na seta apontando para a direita, você vai executar apenas o teste selecionado. Se você for na opção de cima, PessoaTest, também vai aparecer a mesma opção. A diferença é que você vai executar o teste do arquivo todo.

Agora, vai abrir um terminal com a seguinte mensagem:

```
%TESTC 1 v2
%TSTTREE1,testGetNome(com.mycompany.app.PessoaTest),false,1,false,-1,testGetNome
(com.mycompany.app.PessoaTest),,
%TESTS 1,testGetNome(com.mycompany.app.PessoaTest)

%TESTE 1,testGetNome(com.mycompany.app.PessoaTest)

%RUNTIME22
```

Vamos analisar cada parte:

1. %TESTC 1 v2

Esta linha indica o início de um novo conjunto de testes. O número "1" representa o número do conjunto de testes e "v2" indica a versão do formato do relatório.

2. %TSTTREE1,testGetNome(com.mycompany.app.PessoaTest),false,1,false,-1,testGetNome
(com.mycompany.app.PessoaTest),

Esta linha descreve a estrutura da árvore de testes. Ela inclui informações como o nome do teste ("testGetNome"), o nome completo da classe de teste ("com.mycompany.app.PessoaTest"), se é um teste de caso de teste (false), o número de testes (1), se é um teste de suíte (false), o ID do pai (-1), o nome do teste novamente e alguns campos em branco.

3. %TESTS 1,testGetNome(com.mycompany.app.PessoaTest) :

Esta linha indica que o conjunto de testes número 1 inclui o teste "testGetNome" da classe "PessoaTest" no pacote "com.mycompany.app".

4. %TESTE 1,testGetNome(com.mycompany.app.PessoaTest)

Esta linha indica o término do teste "testGetNome" da classe "PessoaTest" no pacote "com.mycompany.app".

5. %RUNTIME22 :

Esta linha indica informações sobre o tempo de execução dos testes, como a duração total da execução dos testes foi de 22 milissegundos.

Nosso teste passou! Agora, vamos ver o comportamento de testes que não passam. Faça a seguinte modificação no seu código, salve e execute o teste novamente:

```
pessoa.setNome("Jessica");
```

Você vai se deparar com esta saída no terminal:

```
%TESTC 1 v2
%TSTTREE1,testGetNome(com.mycompany.app.PessoaTest),false,1,false,-1,testGetNome
(com.mycompany.app.PessoaTest),,
%TESTS 1,testGetNome(com.mycompany.app.PessoaTest)

%FAILED 1,testGetNome(com.mycompany.app.PessoaTest)
%EXPECTS
Jess
%EXPECTE
%ACTUALS
```

E mais uma barra vermelha no seu código, apontando exatamente para a linha com o erro. Seu Code Spaces vai estar com a seguinte aparência:

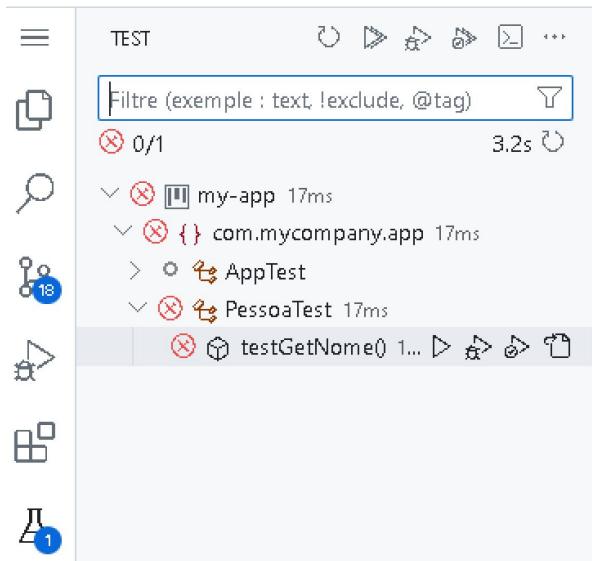
```

J PessoaTest.java ×
src > test > java > com > mycompany > app > J PessoaTest.java > P PessoaTest > testGetNome()
6  public class PessoaTest {
8    public void testGetNome() {
13      pessoa.setName("Jessica");
14
15      // Verifica se o nome retornado é o mesmo que foi setado
16      assertEquals(expected:"Jess", pessoa.getName()); Expected [Jess] but was [Jessica]

```

Expected [Jess] but was [Jessica] testGetNome()

Attendu
-Jess | Réel
+Jessica



É bastante fácil entender onde um teste falha, usando uma IDE.

Em ambos, tanto na IDE em si como no terminal, a mensagem é: é esperado Jess mas recebeu Jessica. Graças aos testes unitários, caso alguém faça alguma modificação no seu código que não era para ser feita, você vai descobrir assim que rodar seus testes, evitando colocar bugs em produção, garantindo um código mais seguro.

Vamos escrever testes para os outros métodos agora:

Teste do método SetNome()

```

@Test
public void testSetNome() {
    // Cria uma instância da classe Pessoa
    Pessoa pessoa = new Pessoa();

    // Define um nome para a pessoa
    pessoa.setName("Jess");

    // Verifica se o nome retornado é o mesmo que foi setado
    assertEquals("Jess", pessoa.getName());
}

```

Como pode notar, apesar do primeiro teste ter sido um pouco mais trabalhoso, o segundo foi super rápido, porque reutilizamos quase tudo do primeiro teste, só precisamos mudar a chamada do método.

Veja como está o nosso arquivo de testes com esse novo teste do metodo testeSetNome():

```

package com.mycompany.app;

import org.junit.Test;
import static org.junit.Assert.*;

public class PessoaTest {
    @Test
    public void testGetNome() {
        // Cria uma instância da classe Pessoa
        Pessoa pessoa = new Pessoa();

        // Atribui um nome para a pessoa
        pessoa.setNome("Jess");

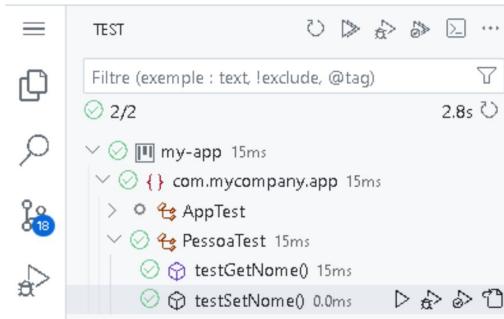
        // Verifica se o nome retornado é o mesmo que foi setado
        assertEquals("Jess", pessoa.getNome());
    }
    @Test
    public void testSetNome() {
        // Cria uma instância da classe Pessoa
        Pessoa pessoa = new Pessoa();

        // Define um nome para a pessoa
        pessoa.setNome("Jess");

        // Verifica se o nome retornado é o mesmo que foi setado
        assertEquals("Jess", pessoa.getNome());
    }
}

```

E nossa IDE está mostrando que os testes passaram:



Repare que tem um círculo verinho, mostrando que nossos testes passaram. Se você olhar o seu arquivo de testes, repare que do lado do nome de cada método, também temos o círculo verde indicando que o teste passou, nas linhas 6, 8 e 19.

```

6  public class PessoaTest {
7      @Test
8      public void testGetNome() {
9          // Cria uma instância da classe Pessoa
10         Pessoa pessoa = new Pessoa();
11
12         // Atribui um nome para a pessoa
13         pessoa.setNome("Jess");
14
15         // Verifica se o nome retornado é o mesmo que foi setado
16         assertEquals(expected:"Jess", pessoa.getName());
17     }
18     @Test
19     public void testSetName() {

```

Vamos concluir os testes dos demais métodos!

Teste do método `testGetLivros()`

Agora, no caso do método `getLivros()`, vamos criar uma lista de livros para a pessoa, então, vamos criar um array e vamos precisar comparar a lista. Neste caso, vamos usar o `assertArrayEquals` afinal, vamos comparar arrays. Vamos acabar usando os métodos mais comuns do JUnit, então não se preocupe em conhecer todos os métodos:

```
@Test
public void testGetLivros() {
    // Cria uma instância da classe Pessoa
    Pessoa pessoa = new Pessoa();

    // Define uma lista de livros para a pessoa
    Livro[] livros = new Livro[];
    livros = new Livro("Livro 1");
    livros = new Livro("Livro 2");
    pessoa.setLivros(livros);

    // Verifica se a lista de livros retornado é a mesma que foi definida
    assertEquals(livros, pessoa.getLivros());
}
```

Agora vem um ponto muito importante pelo qual devemos sempre escrever testes para melhorar nosso código ou detectar inconsistências. Assim que você rodar esse teste, você vai identificar um erro no seu terminal:

Type mismatch: cannot convert from Livro to Livro[]

Este erro está acontecendo porque estamos tentando criar um objeto `Livro` passando uma `String` como argumento para o construtor, mas a classe `Livro` não possui um construtor!

Nosso código funcionou corretamente até aqui, quando executamos no terminal. Porém, como comentamos, o código rodar e exibir as saídas que desejamos não quer dizer que ele não tem erros nos comportamentos esperados e, por isso, os testes são tão importantes.

→ Aula 3.6 - Criando construtores nas classes

Antes de começarmos, vamos voltar para a tão prometida explicação, entendendo o que são construtores e que vantagens teremos com seu uso.

Construtores em Java

Um construtor é um método especial em uma classe Java que é usado para inicializar os objetos dessa classe. Quando você cria uma nova instância de uma classe usando o operador `new`, o construtor é automaticamente chamado para inicializar o objeto recém-criado.

Algumas regrinhas importantes sobre os construtores em Java:

- 1. Nome do construtor:** O nome do construtor deve ser exatamente o mesmo nome da classe.
- 2. Não há tipo de retorno:** Ao contrário dos métodos normais, os construtores não têm um tipo de retorno, nem mesmo `void`.
- 3. Pode haver sobrecarga:** Assim como os métodos, os construtores podem ser sobrecarregados, o que significa que você pode ter vários construtores com diferentes listas de parâmetros na mesma classe.
- 4. Chamada implícita:** Quando você cria um novo objeto usando `new`, o construtor é chamado implicitamente.
- 5. Inicialização de propriedades:** O principal propósito de um construtor é inicializar as propriedades da classe com valores apropriados.
- 6. Pode chamar outros construtores:** Um construtor pode chamar outro construtor da mesma classe usando a palavra-chave `this`.

Lembra-se da nossa classe Pessoa? Ela não possui nenhum construtor. Temos apenas as propriedades e os métodos getters e setters:

```
public class Pessoa {  
    private String nome;  
  
    public String getNome() {  
        return nome;  
    }  
    public void setNome(String nome) {  
        this.nome = nome;  
    }  
  
    private Livro[] listaLivros;  
  
    public Livro[] getLivros() {  
        return listaLivros;  
    }  
  
    public void setLivros(Livro[] listaLivros) {  
        this.listaLivros = listaLivros;  
    }  
}
```

E, para criarmos uma pessoa, precisamos chamar o método setNome, mas, e se esquecermos de chamar algum método para definir propriedades ou comportamentos? Isso poderia induzir ao erro!

Com o construtor, teremos uma forma de garantir que todas as instâncias de classe serão criadas com as mesmas características.

```
public class Pessoa {  
    private String nome;  
  
    public Pessoa(String nome) {  
        this.nome = nome;  
    }  
  
    // Getters e Setters  
}
```

Fariamos a chamada desse construtor como:

```
Pessoa pessoa = new Pessoa("Jess");
```

No caso da classe livro, isso fica ainda mais evidente, já que precisamos de várias propriedades para criar um livro e não podemos esquecer de nenhuma delas. Precisamos de um título, um objeto autor, o gênero do livro e se ele está disponível.

Nosso construtor vai ficar da seguinte forma:

```
public Livro(String titulo, Autor autor, String genero, boolean disponivel) {  
    this.titulo = titulo;  
    this.autor = autor;  
    this.genero = genero;  
    this.disponivel = disponivel;  
}
```

Agora, para criar uma instância da Classe Livro, vamos precisar passar os seguintes argumentos:

```
Livro livro = new Livro("Java For Begginers", autor, "tecnologia", true);
```

Caso algum dos argumentos não seja passado, a nossa instância de classe não será criada. Bem melhor, não é? Então, vamos acrescentar esse construtor no nosso arquivo Livro:

```
package com.mycompany.app;

public class Livro {
    private String titulo;
    private Autor autor;
    private String genero;
    private boolean disponivel;

    // Construtor
    public Livro(String titulo, Autor autor, String genero, boolean disponivel)
    {
        this.titulo = titulo;
        this.autor = autor;
        this.genero = genero;
        this.disponivel = disponivel;
    }

    // Getters e Setters

    public String getTitulo() {
        return titulo;
    }

    public void setTitulo(String titulo) {
        this.titulo = titulo;
    }

    public Autor getAutor() {
        return autor;
    }

    public void setAutor(Autor autor) {
        this.autor = autor;
    }

    public String getGenero() {
        return genero;
    }

    public void setGenero(String genero) {
        this.genero = genero;
    }

    public boolean isDisponivel() {
        return disponivel;
    }

    public void setDisponivel(boolean disponivel) {
        this.disponivel = disponivel;
    }

    // Método validarDisponibilidade
    public void validarDisponibilidade() {
        if (disponivel) {
            System.out.println("O livro está disponível.");
        } else {
            System.out.println("O livro não está disponível.");
        }
    }
}
```

Também vamos precisar fazer uma alteração na chamada do nossa classe Main:

```
Livro livro = new Livro("Java For Begginers", autor, "tecnologia", true);
livro.validarDisponibilidade();
```

Ficou bem mais elegante agora, além de mais curto de ler e mais coeso.

Também vamos modificar nossos testes, na classe PessoaTest. No array de livros que criamos, vamos usar o novo construtor da classe Livro:

```
Livro[] livros = new Livro[2];
livros[0] = new Livro("Java Basico", "Jessica Felix", "tecnologia", true);
livros[1] = new Livro("Java Avançado", "Jessica Félix", "tecnologia", true);
pessoa.setLivros(livros);
```

Tente rodar seus testes novamente e...

Agora teremos mais um erro!

```
The constructor Livro(String, String, String, boolean) is undefined
```

→ Aula 3.7 - Corrigindo erros durante os testes

Mas não acabamos de definir qual seria o construtor? Porque ele está undefined? Porque nosso construtor espera receber um objeto Autor, não uma string.

Então, vamos fazer mais um ajuste no nosso código da classe PessoaTest:

```
@Test
public void testGetLivros() {
    // Cria uma instância da classe Pessoa e da classe Autor
    Pessoa pessoa = new Pessoa();
    Autor autor = new Autor();

    // Define uma lista de livros para a pessoa
    Livro[] livros = new Livro[2];
    livros[0] = new Livro("Java Basico", autor, "tecnologia", true);
    livros[1] = new Livro("Java Avançado", autor, "tecnologia", true);
    pessoa.setLivros(livros);

    // Verifica se a lista de livros retornado é a mesma que foi definida
    assertEquals(livros, pessoa.getLivros());
}
```

Ok, agora nosso código passou! Mas, quando precisarmos buscar por um livro, repare que o autor não está sendo passado com nenhuma informação! Então, vamos criar também um construtor para a classe Autor:

```
package com.mycompany.app;

public class Autor extends Pessoa {
    private String nacionalidade;
    private Livro[] obrasPublicadas;

    // Construtor com nome e nacionalidade
    public Autor(String nome, String nacionalidade) {
        super(nome); // Chama o construtor da classe Pessoa para inicializar o
        nome
        this.nacionalidade = nacionalidade;
    }
    // Getters e Setters
```

Agora, vamos atualizar nosso teste:

```
public void testGetLivros() {
    // Cria uma instância da classe Pessoa
    Pessoa pessoa = new Pessoa("Jessica");
    Autor autor = new Autor("Jess", "Brasileira");

    // Define uma lista de livros para a pessoa
    Livro[] livros = new Livro[2];
    livros[0] = new Livro("Java Basico", autor, "tecnologia", true);
    livros[1] = new Livro("Java Avançado", autor, "tecnologia", true);
    pessoa.setLivros(livros);

    // Verifica se a lista de livros retornado é a mesma que foi definida
    assertEquals(livros, pessoa.getLivros());
}
```

Nosso último ajuste vai ser na classe Usuario que, embora não vai ser usada no teste, como acrescentamos um construtor à classe Pessoa, Usuario herda Pessoa e, se não adequarmos, vai causar um erro quando executarmos nosso código novamente:

```
package com.mycompany.app;

public class Usuario extends Pessoa {
    private int idade;
    private Emprestimo[] historicoEmprestimos;

    // Construtor com nome, idade e histórico de empréstimos
    public Usuario(String nome, int idade) {
        super(nome); // Chama o construtor da classe Pessoa para inicializar o nome
        this.idade = idade;
    }
    // Getters e Setters
```

Agora, vamos ajustar nossa classe Main, que faz todas as chamadas para nos mostrar aquele resultado de cadastros. Como temos nossos construtores, não precisamos mais chamar os Setters:

```
package com.mycompany.app;

import java.util.Date;

public class Main {
    public static void main(String[] args) {
        // Criando um Autor
        Autor autor = new Autor("Jess", "Inglesa");

        // Criando um Livro
        Livro livro = new Livro("Java For Begginers", autor, "tecnologia", true);
        livro.validarDisponibilidade();

        // Criando um Usuario
        Usuario usuario = new Usuario("Jess", 21);

        // Criando um Emprestimo
        Emprestimo emprestimo = new Emprestimo();
        emprestimo.setDataRetirada(new Date());
        emprestimo.setDataDevolucao(new Date());
        emprestimo.setLivro(livro);
        emprestimo.setUsuario(usuario);
```

```

// Exibindo informações
    System.out.println("Livro: " + livro.getTitulo());
    System.out.println("Autor: " + livro.getAutor().getNome());
    System.out.println("Gênero: " + livro.getGenero());
    System.out.println("Usuário: " + usuario.getNome());
    System.out.println("Idade: " + usuario.getIdade());
    System.out.println("Data de Retirada: " + emprestimo.getDataRetirada());
    System.out.println("Data de Devolução: " + emprestimo.getDataDevolucao());
}
}

```

Por fim, também vamos criar um construtor para empréstimo, para remover o uso dos Setters e deixar nossas classes todas padronizadas:

```

public Emprestimo(Date dataRetirada, Date dataDevolucao, Livro livro,
Usuario usuario) {
    this.dataRetirada = dataRetirada;
    this.dataDevolucao = dataDevolucao;
    this.livro = livro;
    this.usuario = usuario;
}

```

Como vamos instanciar a classe Emprestimo, na Main:

```

Emprestimo emprestimo = new Emprestimo((new Date()), (new Date()),
livro, usuario);

```

Agora, depois de todas estas mudanças, vamos ajustar nosso arquivo de PessoaTest e adequar as instâncias que criamos, de acordo com os novos construtores. Nossa class vai ficar assim:

```

package com.mycompany.app;

import org.junit.Test;
import static org.junit.Assert.*;

public class PessoaTest {
    @Test
    public void testGetNome() {
        Pessoa pessoa = new Pessoa("Jessica Felix");
        pessoa.setNome("Jess");
        assertEquals("Jess", pessoa.getNome());
    }

    @Test
    public void testSetNome() {
        Pessoa pessoa = new Pessoa("Jess");
        pessoa.setNome("Jessica");
        assertEquals("Jessica", pessoa.getNome());
    }

    @Test
    public void testGetLivros() {
        // Cria uma instância da classe Pessoa
        Pessoa pessoa = new Pessoa("Jessica");
        Autor autor = new Autor("Jess", "Brasileira");

        // Define uma lista de livros para a pessoa
        Livro[] livros = new Livro[2];
        livros[0] = new Livro("Java Basico", autor, "tecnologia", true);
        livros[1] = new Livro("Java Avançado", autor, "tecnologia", true);
        pessoa.setLivros(livros);
    }
}

```

```

// Verifica se a lista de livros retornado é a mesma que foi definida
    assertEquals(livros, pessoa.getLivros());
}

@Test
public void testSetLivros() {
    // Cria uma instância da classe Pessoa
    Pessoa pessoa = new Pessoa("Jessica");

    // Setta uma lista de livros para a pessoa
    Livro[] livros = new Livro[2];
    livros[0] = new Livro(null, null, null, false);
    livros[1] = new Livro(null, null, null, false);
    pessoa.setLivros(livros);

    // Verifica se a lista de livros retornado é a mesma que foi setada
    assertEquals(livros, pessoa.getLivros());
}
}

```

Com nossos testes passando, vamos rever tudo que aprendemos nesta aula.

Resumindo...

Testes Unitários: São como testes de qualidade para o código, garantindo que cada parte funcione corretamente, como verificar se um livro pode ser emprestado sem erros.

JUnit: É uma ferramenta que ajuda a fazer esses testes de forma organizada e automática, como ter um assistente para verificar se os livros estão sendo emprestados corretamente.

Construtores no Java: São como receitas para criar objetos, dizendo como preparar um objeto com os ingredientes certos, como seguir uma receita para fazer um bolo.

new Date(); É uma instrução que invoca o construtor padrão da classe Date em Java, criando uma instância que representa um ponto específico no tempo, como a data e hora exatas em que um empréstimo foi realizado. De forma mais simples, é como pedir um relógio novo em Java, que permite marcar momentos específicos no tempo, como registrar a hora exata de quando um livro foi emprestado.

Pacotes no Java: São namespaces utilizados para agrupar classes relacionadas logicamente, visando melhorar a organização e gerenciamento de projetos Java, facilitando a modularidade e reutilização do código.

Modulo 3 - Exercícios

Você gostou de criar testes para a classe Pessoa? Eles ajudam a ter uma melhor compreensão do código e a fazer melhorias para entregar uma aplicação de mais qualidade.

Agora, como atividade, faça a cobertura de testes da classe das demais classes, exceto a Main.

Regras:

1- Você deve criar testes para todos os métodos das classes Livro, Autor, Usuario e Empréstimo.
 Para você ganhar tempo e se focar apenas nos testes, já deixamos prontas as instâncias de classes para você usar e, também para facilitar a correção:

Para você ganhar tempo e se focar apenas nos testes, já deixamos prontas as instâncias de classes para você usar e, também para facilitar a correção:

```
EmprestimoTest.java           deve           usar:  
    Date dataRetirada = new Date();  
    Date dataDevolucao = new Date();  
    Livro livro = new Livro("Java Basics", new Autor("Alan Turing",  
"Inglês"), "Tecnologia", true);  
    Usuario usuario = new Usuario("Gabriel", 21);  
  
LivroTest.java                deve           usar:  
    Autor autor = new Autor("Jess", "Brasileira");  
    Livro livro1 = new Livro("Java Basico", autor, "tecnologia", true);  
    Livro livro2 = new Livro("Java Avançado", autor, "tecnologia", false);
```

Instruções para submissão: Na pasta **modulo3** do seu repositório no Github coloque os seguintes arquivos:

7. LivroTest.java
8. AutorTest.java
9. UsuarioTest.java
10. EmprestimoTest.java

Boa sorte!

Modulo 4.0 - Conceitos Práticos da Linguagem Java

Agora, para avançarmos em conceitos mais avançados de Orientação a Objetos, vamos precisar ter familiaridade com alguns conceitos da Linguagem Java. Mas não se preocupe que este curso é focado em orientação a objetos, vamos aprender o que for necessário para avançarmos em tratamento de exceções e S.O.L.I.D.

Antes de começar a entrar em conceitos como Interfaces e outros importantes sobre Java, vamos precisar fazer alguns ajustes na organização de diretórios do nosso projeto.

→ Aula 4.1 - Reorganizando a estrutura de pastas do projeto

Exercício Guiado: Criando a pasta Model e Interfaces

Vamos criar, dentro de src/main/java/com/mycompany/app dois novos diretórios: Interfaces e model e fazer mais algumas mudanças, para nosso projeto crescer mais fácil de dar manutenção e mais coeso.

Vamos guardar nossas classes feitas na aula passada dentro do diretório model nossas interfaces vão ficar no diretório com o mesmo nome.

A pasta "model" é comumente usada em projetos de software para armazenar as classes que representam o modelo de dados da aplicação. Essas classes são como representações das entidades do mundo real com as quais o sistema interage. Elas contêm informações sobre essas entidades, como atributos e métodos que descrevem seu comportamento e características.

1 - Dentro do pacote com.mycompany.app, crie uma subpasta chamada interfaces e outra chamada model. O diretório deve estar da seguinte forma:

```
src/
  main/
    java/
      com/
        mycompany/
          app/
            interfaces/
            model/
```

2 - Agora, move nossas classes para dentro do diretório model, mas mantenha a classe main onde ela estava. Sabe as nossas classes com .class, tipo Pessoa.class? Pode apagar todos os arquivos com .class. A estrutura das suas pastas deve estar da seguinte forma:

```
src/
  main/
    java/
      com/
        mycompany/
          app/
            main.java
            interfaces/
            model/
            Pessoa.java
            Usuario.java
            Autor.java
            Livro.java
            Emprestimo.java
```

A pasta "model" é responsável por encapsular a lógica de negócios e a representação dos dados dentro de um sistema. Isso significa que as classes presentes nessa pasta lidam com a manipulação e a validação dos dados, bem como com a execução das regras de negócio da aplicação.

Elas representam as entidades do sistema e definem como essas entidades interagem entre si e com o mundo exterior.

A classe Main é comumente utilizada como ponto de entrada da aplicação, onde o programa é iniciado e a lógica principal é executada. Ela não faz parte do modelo de dados da aplicação, mas sim coordena a interação entre as diferentes partes do sistema.

É uma prática comum manter a classe Main em um diretório separado, como src/main/java/com/mycompany/app, fora do diretório model. Isso ajuda a manter a organização do código, separando claramente a lógica de inicialização e execução do programa da definição das entidades e regras de negócio do modelo de dados.

3 - Nosso próximo passo é organizar onde os arquivos .class ficarão. Você deve ter percebido que, quando compilamos nossas classes, para cada arquivo .java foi gerado um arquivo .class, certo? Porém, além de poluir a visão da pasta (imagine ter 30 classes? Teríamos 60 arquivos no total) não faz sentido deixá-los na pasta model, já que os arquivos .class são o binário da classe, não a classe em si.

Crie um diretório chamado bin na raiz do seu projeto. Ele deve ficar na raiz porque vai precisar ter acesso ao arquivo pom.xml. Voce deve visualizar essa estrutura de pastas:

```
projeto/
  bin/
  src/
    main/
      java/
        com/
          mycompany/
            app/
```

4 - Agora, vamos adicionar uma configuração no arquivo pom.xml. Este arquivo é onde as configurações gerais do projeto estão. Acrescente esse trecho, dentro da tag de build. :

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-compiler-plugin</artifactId>
  <version>3.8.1</version>
  <configuration>
    <source>11</source>
    <target>11</target>
    <outputDirectory>${project.basedir}/bin</outputDirectory>
  </configuration>
</plugin>
```

Seu arquivo pom.xml , dentro da tag de build, deve estar dessa forma:

```
<build>
  <pluginManagement>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-surefire-plugin</artifactId>
        <version>3.0.0-M6</version>
      </plugin>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.8.1</version>
        <configuration>
          <source>11</source>
          <target>11</target>
          <outputDirectory>${project.basedir}/bin</outputDirectory>
        </configuration>
      </plugin>
    </plugins>
  </pluginManagement>
</build>
</project>
```

Agora, o que acabamos de fazer?

Gerenciamento de projeto com Maven

A pasta bin é comumente usada para armazenar arquivos compilados, como os arquivos .class, em projetos Java. Ela é tradicionalmente usada para separar os arquivos compilados dos arquivos de código-fonte, mantendo a estrutura do projeto organizada e limpa.

A pasta bin geralmente é criada na raiz do projeto, ao lado das pastas de código-fonte, como src. Dentro da pasta bin, os arquivos compilados são organizados de forma semelhante à estrutura de pacotes dos arquivos de código-fonte.

Para gerar os arquivos .class compilados dentro da pasta bin, você precisa configurar seu ambiente de desenvolvimento ou a ferramenta de build que está usando para compilar o projeto.

Aqui, estamos usando o [Maven](#), uma ferramenta popular de gerenciamento de projetos e build para Java.

O trecho que acabamos de colar no nosso arquivo pom.xml é uma configuração que diz ao Maven para usar o plugin maven-compiler-plugin para compilar o código-fonte Java. O parâmetro outputDirectory especifica o diretório onde os arquivos .class compilados devem ser gerados, neste caso, a pasta bin na raiz do projeto.

Atualizando os imports do projeto

Agora que já realizamos toda reorganização de pasta, também precisamos realizar a organização dos imports nos arquivos, caso contrário, quando você tentar executar o projeto, vai receber um erro de build.

1- Nas nossas classes que estão na pasta Model, complete o valor do package:

```
package com.mycompany.app.model;
```

2 - Nos arquivos de teste e na Main, você precisa apontar corretamente onde as classes que os testes criam instancias:

```
import com.mycompany.app.model.Autor;
import com.mycompany.app.model.Emprestimo;
import com.mycompany.app.model.Livro;
import com.mycompany.app.model.Usuario;
import com.mycompany.app.model.Pessoa;
```

Agora, faça o build do projeto com o comando:
mvn package

O que são os arquivos .class?

Os arquivos .class são gerados durante o processo de compilação de um projeto Java pelo Maven. O Maven utiliza o conceito de POM (Project Object Model) e um conjunto de plugins para compilar o código-fonte Java.

Ao definir a configuração correta no arquivo pom.xml, o Maven compila o código-fonte Java e gera os arquivos .class compilados. Esses arquivos contêm o bytecode executável do código-fonte Java e são essenciais para a execução do programa Java.

→ Aula 4.2 - Interfaces em Java

Você já viu uma explicação sobre interface nas primeiras aulas mas, agora, vamos criar interfaces.

Em termos simples, uma interface em Java é um conjunto de métodos que uma classe deve implementar. É como uma lista de tarefas que uma classe precisa realizar, mas não diz como cada tarefa deve ser feita.

Na orientação a objetos, as interfaces são usadas para definir um comportamento comum que várias classes podem compartilhar.

Por exemplo, se temos várias classes que podem ser comparadas entre si, podemos criar uma interface chamada Comparable que define um método compareTo().

Todas as classes que implementam essa interface concordam em ter um método compareTo() que as torna comparáveis.

Trazendo um exemplo possível de interface aplicado ao nosso projeto de biblioteca, Vamos criar uma interface Pessoa que define comportamentos comuns para as classes Autor e Usuario.

Essa interface pode ter métodos como getNome() e setNome() para que as classes que a implementam possam lidar com o nome da pessoa.

```
public interface Pessoa {
    String getNome();
    void setNome(String nome);
}
```

Agora, as classes Autor e Usuario podem implementar essa interface, fornecendo suas próprias implementações para os métodos getNome() e setNome(). Vamos ver como esse uso de Interface aconteceria na classe Autor. Para facilitar o foco na interface, vamos mostrar apenas a parte do código que faremos alterações:

```

public class Autor extends Pessoa {
    // Restante do código da classe Autor

    // Implementação dos métodos da interface Pessoa

    @Override
    public String getNome() {
        return super.getNome();
    }

    @Override
    public void setNome(String nome) {
        super.setNome(nome);
    }
}

```

com a implementação da interface Pessoa na classe Autor, a classe Autor agora é capaz de implementar os métodos `getNome()` e `setNome(String nome)` que são definidos na interface Pessoa.

Isso significa que a classe Autor agora tem a capacidade de fornecer e alterar o nome de uma pessoa, o que é um comportamento comum para uma classe que representa uma pessoa.

Por exemplo, você pode criar um objeto Autor e chamar o método `getNome()` para obter o nome do autor:

```

Autor autor = new Autor("Alan Turing", "Inglês");
String nome = autor.getNome();

```

Ou você pode chamar o método `setNome(String nome)` para alterar o nome do autor:

```
autor.setNome("Alan Turing");
```

Mas, eu expliquei tudo isso apenas pra você entender o conceito.

Na prática, esse exemplo é muito inutil porque eu posso conseguir o mesmo comportamento usando um construtor, já que as classes Usuário e Autor não tiram proveito nenhum a interface, já que não modificam a implementação dos métodos dela.

Inclusive, se mantivermos essa interface, só vamos acrescentar complexidade desnecessária para o nosso código. Descarte as alterações feitas em Pessoa, Descarte a PessoalInterface.java e descarte as alterações em Autor.java.

Esse episodio de precisar apagar código ja escrito é bastante comum quando começamos a codar sem entender bem os fundamentos do que estamos fazendo e sem ter um diagrama apoiando nossa decisão

Também acontece quando tentamos encaixar uma implementação sem que haja uma real necessidade para ela e, no nosso cenário atual, não existe necessidade nenhuma de ter flexibilidade de implementação de um método.

→ Aula 4.3 - Lidando com o crescimento de uma aplicação Orientada a Objetos

Mas, como estamos num curso e eu quero te ensinar a trabalhar com interfaces, classes abstratas e classes concretas, e além disso, a biblioteca é minha, vou fazer uma adição de regra de negócio:

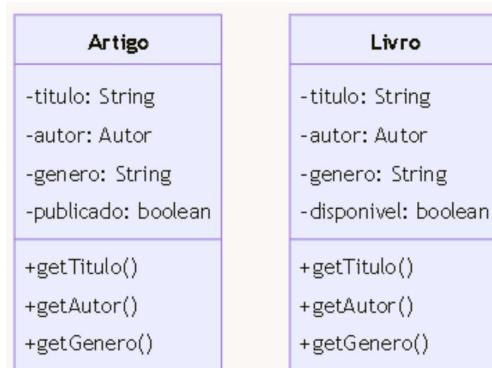
Agora, a Biblioteca foi para o mundo digital e teremos um site, onde os autores podem publicar suas obras em formato de e-book e nossos usuários podem publicar artigos.

Dessa forma, aumentamos o tráfego no site por causa dos artigos, que vai ajudar a dar mais visibilidade para as obras novas dos autores.

A regra, porém, é que autores não podem publicar artigos e usuários não podem publicar livros.

Ok, agora sim temos um problema que faz sentido pensar em implementações diferentes para a mesma coisa! O autor precisa publicar um livro, já o nosso usuário, precisa publicar um artigo. Ambos precisam realizar a mesma ação, publicar, mas a implementação vai ser diferente.

Vamos entender como seria essa dinâmica nas classes. Ao lado do diagrama, eu deixo o código que usei para cria-lo.



```

classDiagram
    class Publicável {
        <<interface>>
        +publicar()
    }

    class Usuário {
        -nome: String
        -idade: int
        +publicar()
    }

    class Autor {
        -nome: String
        -nacionalidade: String
        +publicar()
    }

    class Livro {
        -titulo: String
        -autor: Autor
        -genero: String
        -disponivel: boolean
        +getTitulo()
        +getAutor()
        +getGenero()
        +isDisponivel()
    }

    class Artigo {
        -titulo: String
        -autor: Autor
        -genero: String
        -publicado: boolean
        +getTitulo()
        +getAutor()
        +getGenero()
        +isPublicado()
    }

    Usuário ..|> Publicável
    Autor ..|> Publicável
    Livro o-- Autor
    Artigo o-- Usuário
  
```

É escrito em Mermaid, uma linguagem de marcação de texto que permite criar diagramas. Embora esse não seja o assunto das nossas aulas, fica a sugestão de conhecer e tentar criar alguns diagramas com essa ferramenta.

Por ser uma linguagem de marcação, é possível renderizar os diagramas em arquivos HTML ou Markdown, muito útil para manter a documentação já dentro do seu repositório de Github e versioná-lo como fazemos com código. Você também pode ver a renderização desse diagrama [num editor online](#) de Mermaid. Basta copiar corretamente o código do diagrama.

Agora, vamos voltar para o nosso diagrama.

→ Aula 4.4 - Aprendendo decisões arquiteturais: o Usuário que virou autor

Analizando o diagrama, entendemos algumas relações das classes:

- A interface Publicável define o método publicar().
- A classe Usuário e Autor implementam a interface Publicável, implementando o método publicar() de maneira específica para cada classe.
- A classe Livro têm uma associação com a classe Autor, indicando que cada Livro tem um Autor.
- A classe Artigo têm uma associação com a classe Usuário, indicando que cada Artigo tem um Usuário.

Agora, temos mais um problema para pensar: Se um Usuário publica um artigo, ele vira um Autor, pela lógica. Como vamos tratar isso na nossa estrutura de classes?

Esse é o dia a dia de uma pessoa que cuida de arquitetura de sistemas, seja uma pessoa arquiteta de software ou uma pessoa engenheira de software Sênior. E essas decisões nunca são fáceis.

Quero lembrar que esse assunto está totalmente dentro da nossa pauta de orientação a objetos, afinal, é um problema que precisamos lidar levando em consideração a implementação do paradigma de programação escolhido.

Vamos começar analisando algumas possíveis abordagens:

Herança ou Composição:

- Uma abordagem seria fazer com que a classe Usuário também implemente a interface Autor, além da interface Publicável. Isso permitiria que um Usuário que publica artigos seja tratado como um Autor quando necessário.
- Outra abordagem seria ter uma classe Autor separada, mas permitir que um Usuário se torne um Autor dinamicamente, por exemplo, ao publicar um certo número de artigos ou ao atingir um certo status no sistema.

Gerenciamento de papéis:

- Você pode introduzir um mecanismo de gerenciamento de papéis, onde um Usuário pode ter diferentes papéis, como Usuário Comum, Autor, Editor, etc. Dependendo do papel, o Usuário teria diferentes funcionalidades e permissões.

Refatoração da Lógica de Negócios:

- Revisar a lógica de negócios e a modelagem de classes para garantir que a relação entre Usuário e Autor seja tratada de forma consistente e lógica. Isso pode envolver ajustes na estrutura das classes e nas relações entre elas.

Tem mais casos possíveis mas, pra não aumentarmos muito a complexidade da discussão, vamos manter apenas 4 abordagens.

Dessas, como é uma aula sobre interfaces, eu vou começar apostando na criação de uma interface Autor, ao invés da classe Autor, ai eu teria apenas Pessoa e Usuário e eu definiria uma interface autor que seria implementada em caso de autores, seja de artigo ou de livro.

Vamos ver um exemplo de código aplicando esta abordagem. Quando estamos fazendo escolhas arquiteturais, pode fazer sentido criar o que chamamos de Prova de Conceito, ou simplesmente POCs.

```

public interface Autor {
    void publicar();
}

public class Usuário implements Autor {
    // Implementação do método publicar() para Usuário
    @Override
    public void publicar() {
        System.out.println("Usuário publicando...");
    }
}

public class Livro implements Autor {
    // Implementação do método publicar() para Livro
    @Override
    public void publicar() {
        System.out.println("Livro publicando...");
    }
}

public class Artigo implements Autor {
    // Implementação do método publicar() para Artigo
    @Override
    public void publicar() {
        System.out.println("Artigo publicando...");
    }
}

```

Neste exemplo, a interface Autor define o método publicar(), que é implementado de forma específica por Usuário, Livro e Artigo. Isso permite que diferentes entidades que desempenham o papel de autor possam publicar de acordo com suas próprias regras e comportamentos específicos, mantendo a consistência e a flexibilidade no sistema.

Parece uma boa solução, não é? Mas lembre, aqui trabalhamos com ciência! Não podemos afirmar que algo está certo, apenas porque parece certo.

Vamos analisar as possíveis desvantagens dessa abordagem, levando em consideração o que essa mudança de remover a classe Autor e colocar uma interface no lugar, dado que temos os seguintes construtores que usam essa classe:

```

// Livro, que recebe um titulo, um autor, um genero e se está disponivel para emprestimo

    Livro livro = new Livro("Java For Begginers", autor, "tecnologia", true);

    // Emprestimo, que recebe data de retirada, data de devolução, um objeto livro e um objeto usuário
    Emprestimo emprestimo = new Emprestimo(new Date(), (new Date()), livro,
    usuario);

    // Artigo, que recebe um titulo, um autor, um genero e se está publicado
    Artigo artigo = new Artigo("Entendendo Compiladores", autor,
    "tecnologia", true);

```

Problemas da abordagem de interface Autor:

1. Complexidade adicional: Ao introduzir uma interface

- Ao introduzir uma interface Autor, você adiciona um nível extra de abstração ao seu sistema, o que pode aumentar a complexidade geral.
- Você precisa gerenciar a implementação da interface Autor em diferentes classes, como Usuário, Livro e Artigo.

2. Impacto nos construtores: Com a introdução da interface

- a. Com a introdução da interface Autor, você precisará ajustar os construtores das classes Livro e Artigo para aceitar uma instância de Autor em vez de uma instância de Autor específica.
- b. Por exemplo, o construtor de Livro ficaria assim: Livro(String titulo, Autor autor, String genero, boolean disponivel).

3. Possível aumento da complexidade em alguns casos

- a. Em alguns casos, usar uma interface Autor pode adicionar complexidade desnecessária, especialmente se você tiver apenas uma implementação de Autor (por exemplo, se apenas Livro implementar Autor).

Benefícios da abordagem da interface Autor:

1. Flexibilidade:

- a. A abordagem da interface Autor permite que diferentes entidades, como Usuário, Livro e Artigo, desempenhem o papel de autor de forma independente, sem a necessidade de herança de classes.

2. Extensibilidade:

- a. Novos tipos de autores podem ser facilmente adicionados ao sistema, bastando implementar a interface Autor, sem a necessidade de alterar a hierarquia de classes existente.

3. Separação de Responsabilidades:

- a. A interface Autor ajuda a separar a definição do papel de autor das implementações específicas de cada entidade, promovendo um design mais coeso e modular.

Agora, que temos essas informações, precisamos pensar qual opção traria menos trade-offs para o nosso sistema de biblioteca?

Considerando que o sistema não vai crescer mais porque estamos chegando mais próximo do fim do curso e visando minimizar o impacto na implementação dos construtores, uma opção com trade-offs menos impactantes seria manter uma classe Autor e utilizar um campo adicional para indicar se o autor é um Usuário ou um Autor tradicional.

Isso permitiria diferenciar entre autores que são usuários e autores que são autores de livros e artigos, mantendo a simplicidade na implementação.

Essa abordagem permitiria que o sistema continuasse a funcionar sem grandes alterações na estrutura existente, mantendo a clareza e a simplicidade na implementação.

Você tomaria outra decisão neste caso? Fica de exercício fazer uma reflexão e entender qual seria a decisão que você tomaria e porquê, pensando em tudo que já aprendemos até aqui.

Antes de confirmarmos que realmente tomaremos essa decisão, vamos entender sobre alguns conceitos que faltaram abordar.

• Aula 4.5 - Implements, Classes Abstratas e Classes Concretas

Vamos começar com a conceituação de cada classe!

Classes Abstratas:

Uma classe abstrata é uma classe que não pode ser instanciada diretamente, ou seja, não é possível criar objetos a partir dela. Ela pode conter métodos abstratos (métodos sem implementação) e métodos concretos (com implementação).

Em uma hierarquia de classes, quando você cria uma nova classe que é uma versão mais específica de outra classe mais geral, a nova classe precisa chamar o "construtor" da classe mais geral.

Mesmo que você não possa criar diretamente objetos a partir da classe mais geral (chamada de classe abstrata), as subclasses precisam usar o construtor dela.

Isso acontece porque as subclasses herdam características da classe abstrata, então é importante que a classe abstrata tenha um construtor para garantir que tudo funcione corretamente.

Características:

- Pode conter métodos abstratos e concretos.
- Pode conter variáveis de instância.
- Pode ter construtores.
- Pode ser estendida por outras classes.
- É útil para fornecer uma implementação padrão e compartilhar código entre classes relacionadas.

Classes Concretas:

Uma classe concreta é uma classe que pode ser instanciada diretamente, ou seja, é possível criar objetos a partir dela. Ela fornece implementações para todos os métodos abstratos herdados de suas superclasses.

Características:

- Pode ser instanciada diretamente.
- Deve fornecer implementações para todos os métodos abstratos herdados.
- Pode conter métodos e variáveis de instância.
- Pode ter construtores.
- Pode implementar interfaces.

No contexto da biblioteca, a classe Publicação, pensando que vamos publicar artigos e livros, poderia ser uma classe abstrata, pois pode conter métodos comuns a todas as publicações, como getTítulo() e getAutor(), e pode ser estendida por classes concretas como Livro e Artigo, que fornecem implementações específicas para esses métodos:

```
// Classe abstrata Publicação
public abstract class Publicação {
    private String título;
    private Autor autor;

    public Publicação(String título, Autor autor) {
        this.título = título;
        this.autor = autor;
    }

    public String getTítulo() {
        return título;
    }

    public Autor getAutor() {
        return autor;
    }
}
```

```

// Método abstrato para ser implementado pelas subclasses
    public abstract void validarPublicação();
}

// Classe Livro que herda de Publicação
public class Livro extends Publicacao {
    private String genero;
    private boolean disponivel;

    public Livro(String titulo, Autor autor, String genero, boolean disponivel)
    {
        super(título, autor);
        this.genero = genero;
        this.disponivel = disponivel;
    }

    public String getGenero() {
        return genero;
    }

    public boolean isDisponivel() {
        return disponivel;
    }

    @Override
    public void validarPublicação() {
        // Lógica específica de validação para Livro
        System.out.println("Validando publicação de Livro...");
    }
}

// Classe Artigo que herda de Publicação
public class Artigo extends Publicação {
    private String genero;
    private boolean publicado;

    public Artigo(String título, Autor autor, String genero, boolean publicado)
    {
        super(título, autor);
        this.genero = genero;
        this.publicado = publicado;
    }

    public String getGenero() {
        return genero;
    }

    public boolean isPublicado() {
        return publicado;
    }

    @Override
    public void validarPublicação() {
        // Lógica específica de validação para Artigo
        System.out.println("Validando publicação de Artigo...");
    }
}

```

Por outro lado, as classes Livro e Artigo seriam classes concretas, pois podem ser instanciadas diretamente e fornecem implementações para os métodos herdados da classe abstrata Publicação.

```
// Classe Livro que herda de Publicação
public class Livro extends Publicação {
    private String genero;
    private boolean disponivel;

    public Livro(String título, Autor autor, String genero, boolean disponivel) {
        super(título, autor);
        this.genero = genero;
        this.disponivel = disponivel;
    }

    public String getGenero() {
        return genero;
    }

    public boolean isDisponivel() {
        return disponivel;
    }

    @Override
    public void validarPublicação() {
        // Lógica específica de validação para Livro
        System.out.println("Validando publicação de Livro...");
    }
}

// Classe Artigo que herda de Publicação
public class Artigo extends Publicação {
    private String genero;
    private boolean publicado;

    public Artigo(String título, Autor autor, String genero, boolean publicado) {
        super(título, autor);
        this.genero = genero;
        this.publicado = publicado;
    }

    public String getGenero() {
        return genero;
    }

    public boolean isPublicado() {
        return publicado;
    }

    @Override
    public void validarPublicação() {
        // Lógica específica de validação para Artigo
        System.out.println("Validando publicação de Artigo...");
    }
}
```

Ok, então, se a classe abstrata não permite criar objetos a partir dela, as interfaces também são assim. Então, classe abstrata e interface tem alguma relação?

Classe abstrata e interface não são a mesma coisa. No contexto do nosso projeto de biblioteca, a diferença entre uma classe abstrata e uma interface estaria na natureza de sua implementação e no propósito que cada uma serve.

Classe Abstrata:

- É útil quando você deseja fornecer uma implementação padrão para métodos e compartilhar código entre classes relacionadas.
- Uma classe abstrata como Publicação poderia conter métodos comuns a todas as publicações, como getTítulo() e getAutor(), permitindo que as classes concretas como Livro e Artigo herdem e implementem esses métodos de maneira específica.

Interface:

- Uma interface é um contrato que define métodos que uma classe deve implementar, mas não fornece implementações.
- Não pode conter variáveis de instância, construtores ou implementações de métodos.
- Pode ser implementada por várias classes independentes.
- É útil quando você deseja definir um contrato comum para classes não relacionadas.
- No contexto da biblioteca, uma interface como Publicável poderia definir o método publicar(), que classes como Usuário e Autor implementariam de maneira específica.

Agora, vamos entender melhor a diferença entre Interface e classe abstrada, para não restar dúvidas:

- Uma classe abstrata pode conter implementações de métodos, variáveis de instância e construtores, enquanto uma interface não pode.
- Uma classe abstrata é usada para fornecer uma implementação padrão e compartilhar código entre classes relacionadas, enquanto uma interface é usada para definir um contrato comum para classes não relacionadas.

No contexto do nosso projeto de biblioteca, usariamos uma classe abstrata quando quisermos fornecer uma implementação padrão para métodos e compartilhar código entre classes relacionadas, como Livro e Artigo.

Por outro lado, usariamos uma interface quando desejarmos definir um contrato comum para classes não relacionadas, como Usuário e Autor, que podem ter comportamentos diferentes, mas precisam implementar um método comum, como publicar().

Resumindo...

Classes abstratas fornecem uma estrutura comum e compartilham código entre classes relacionadas, enquanto classes concretas são instanciáveis diretamente e fornecem implementações específicas para métodos herdados.

Enquanto uma classe abstrata é usada para compartilhar código e fornecer implementações padrão entre classes relacionadas, uma interface é usada para definir um contrato comum para classes não relacionadas, permitindo uma maior flexibilidade e reutilização de código em contextos diversos.

Exercícios

Com base em tudo que vimos nessa aula e nosso novo desenho de arquitetura, vamos aos nossos desafios da aula:

4.1 - Adicionar um campo na Classe Autor para indicar se o autor é um Usuário ou um Autor tradicional. Atualizar o código da Main, para garantir que vamos chamar nossos construtores corretamente

4.2 - Criar um teste unitário para esta alteração

4.3 - Adicionar um objeto Artigo na main, que recebe um título, um autor, um gênero e se está publicado. O construtor dessa classe deve ter essa estrutura, que montamos anteriormente:

Artigo artigo = new Artigo("Entendendo Compiladores", autor, "tecnologia", true);

4.4 - Criar um arquivo de testes para a classe Artigo

Módulo 5 - Design de código

• Aula 5.1 - S.O.L.I.D.: Boas práticas de Design de Código para Orientação a Objetos

Agora que já aprendemos sobre o conceito e aplicação do paradigma de orientação a objetos, vamos passar por um assunto um pouquinho mais avançado mas muito importante, o S.O.L.I.D.

S.O.L.I.D. é um acrônimo que representa cinco princípios de design na programação orientada a objetos, destinados a produzir arquiteturas de software mais compreensíveis, flexíveis e fáceis de manter.

Esses princípios foram introduzidos por Robert C. Martin, o autor do livro Arquitetura Limpa, que já citamos aqui várias vezes.

Os princípios SOLID têm uma forte relação com o paradigma de Orientação a Objetos. Eles foram criados para ajudar a aplicar os conceitos fundamentais da OO de forma eficiente e promover um design de código de alta qualidade.

Cada letra representa um princípio de design e, traduzindo seus nomes para o português, ficariam dessa forma:

1. Responsabilidade Única: Uma classe deve ter apenas uma única responsabilidade ou motivo para mudar. Isso significa que uma classe deve fazer apenas uma coisa e fazer bem feito.

Exemplo no código: Cada classe deve ter apenas uma razão para mudar. No contexto do projeto, isso significa que as classes Autor, Usuario, Livro, Artigo, Publicacao e Emprestimo devem ter responsabilidades bem definidas e específicas.

2. Aberto/Fechado: O código deve ser aberto para extensão, mas fechado para modificação. Isso significa que você deve poder adicionar novas funcionalidades sem precisar alterar o código existente.

Exemplo no código: As classes devem estar abertas para extensão, mas fechadas para modificação. Para aplicar esse princípio, podemos utilizar interfaces e classes abstratas para definir comportamentos comuns e permitir que novas funcionalidades sejam adicionadas sem alterar o código existente.

3. Substituição de Liskov: Se você tem uma classe pai e uma classe filha, você deve poder usar a classe filha no lugar da classe pai sem quebrar nada. A subclasse não deve fazer coisas que a superclasse não faz.

Exemplo no código: Objetos de uma subclasse devem poder ser substituídos por objetos da superclasse sem afetar o comportamento do programa. Isso implica garantir que as classes Autor e Usuario, que herdam de Pessoa, respeitem o contrato da classe base.

4. Segregação de Interfaces: Prefira ter várias interfaces menores e específicas em vez de uma grande interface geral. Isso evita que as classes dependam de métodos que elas não precisam.

Contexto do projeto: Interfaces específicas devem ser preferidas a interfaces gerais. No contexto do projeto, isso significa que as interfaces devem ser coesas e conter apenas os métodos necessários para cada tipo de objeto

5. Inversão de Dependências: Dependa de abstrações, não de implementações concretas. Isso significa que você deve depender de interfaces ou classes abstratas em vez de classes concretas. Isso torna o código mais flexível e fácil de testar.

Contexto no código: Dependa de abstrações, não de implementações concretas. Para aplicar esse princípio, devemos utilizar injeção de dependência e interfaces para reduzir o acoplamento entre as classes.

Exemplo de como podemos aplicar:

1 - Interface para validação de publicação

```
interface ValidacaoPublicacao {  
    boolean validarPublicacao();  
}
```

2 - Classe abstrata Publicação com métodos comuns

```
abstract class Publicacao {  
    protected String titulo;  
    protected Autor autor;  
    protected String genero;  
  
    public Publicacao(String titulo, Autor autor, String genero) {  
        this.titulo = titulo;  
        this.autor = autor;  
        this.genero = genero;  
    }  
  
    public abstract String getTitulo();  
    public abstract Autor getAutor();  
    public abstract ValidaçãoPublicacao validarPublicacao();  
}
```

3 - Classe Livro que estende Publicacao

```
class Livro extends Publicacao {  
    private boolean disponívelParaEmprestimo;  
  
    public Livro(String titulo, Autor autor, String genero, boolean disponívelParaEmprestimo) {  
        super(titulo, autor, genero);  
        this.disponívelParaEmprestimo = disponívelParaEmprestimo;  
    }  
  
    @Override  
    public ValidaçãoPublicacao validarPublicacao() {  
        return new ValidaçãoLivro();  
    }  
}
```

4 - Classe Artigo que estende Publicação

```
class Artigo extends Publicacao {  
    private boolean publicado;  
  
    public Artigo(String titulo, Autor autor, String genero, boolean publicado) {  
        super(titulo, autor, genero);  
        this.publicado = publicado;  
    }  
  
    @Override  
    public ValidaçãoPublicacao validarPublicacao() {  
        return new ValidaçãoArtigo();  
    }  
}
```

5 - Interface para validação de Livro

```
class ValidaçãoLivro implements ValidaçãoPublicacao {  
    public boolean validarPublicacao() {  
        // Lógica de validação para Livro  
        return true;  
    }  
}
```

6 - Interface para validação de Artigo

```
class ValidacaoArtigo implements ValidacaoPublicacao {  
    public boolean validarPublicacao() {  
        // Lógica de validação para Artigo  
        return true;  
    }  
}
```

A aplicação dos princípios SOLID é de extrema importância para garantir um design de software de alta qualidade, mesmo que o projeto não seja necessariamente grande.

Os princípios SOLID são diretrizes fundamentais que promovem a coesão, baixo acoplamento e flexibilidade no código, independentemente do tamanho do projeto.

Mas isso deve ser só para projetos grandes, não é mesmo? Não! Devemos seguir esses princípios de design independente do tamanho ou complexidade do nosso projeto, porém, os princípios de S.O.L.I.D. não são regras matemáticas estritas.

Em algumas vezes ao longo da nossa carreira escrevendo software, vamos precisar abrir mão de um benefício em detrimento de outro, exatamente como fizemos no módulo anterior. Portanto, nenhuma boa prática de design de software deve ser seguida a todo custo.

Algumas vantagens de aplicar S.O.L.I.D. nos nossos projetos:

Manutenção Simplificada: Ao seguir os princípios SOLID, você torna o código mais modular e coeso, facilitando a identificação e correção de erros, bem como a implementação de novas funcionalidades de forma mais eficiente.

Escalabilidade e Flexibilidade: Os princípios SOLID ajudam a tornar o código mais escalável, permitindo que ele se adapte facilmente a mudanças e evoluções no projeto, garantindo que novos requisitos possam ser implementados sem grandes impactos.

Facilidade de Teste e Entendimento: A aplicação dos princípios SOLID torna o código mais testável e compreensível, facilitando a escrita de testes de unidade e a manutenção do sistema ao longo do tempo.

Reaproveitamento e Durabilidade: Ao seguir os princípios SOLID, você promove o reaproveitamento de código e a durabilidade do sistema, garantindo que ele permaneça em utilização por um longo período.

→ Aula 5.2 - Injeção de Dependência

A injeção de dependência é um padrão de design utilizado para desacoplar componentes de um sistema, permitindo que as dependências necessárias sejam injetadas em um objeto em tempo de execução, em vez de serem criadas dentro do próprio objeto. Isso promove a modularidade, flexibilidade e facilita a manutenção e testes do sistema.

No contexto do sistema de biblioteca que você está criando em Java, a injeção de dependência pode ser aplicada para gerenciar as dependências entre as classes de forma mais eficiente e desacoplada.

Um exemplo de injeção de dependência no contexto do seu sistema de biblioteca poderia ser a injeção da dependência do objeto Autor em um Livro ou Artigo.

Em vez de criar uma instância de Autor dentro da classe Livro ou Artigo, a dependência do Autor seria injetada no construtor da classe, permitindo que diferentes autores sejam facilmente associados aos livros e artigos.

Isso torna o código mais flexível, facilita a substituição de implementações e melhora a testabilidade do sistema.

Aqui está um exemplo simplificado de como a injeção de dependência poderia ser aplicada no contexto do nosso sistema de biblioteca em Java:

```

// Classe Autor
public class Autor {
    private String nome;

    public Autor(String nome) {
        this.nome = nome;
    }

    public String getNome() {
        return nome;
    }
}

// Classe Livro com injeção de dependência do Autor
public class Livro {
    private String titulo;
    private Autor autor;
    private String genero;
    private boolean disponivelParaEmprestimo;

    public Livro(String titulo, Autor autor, String genero, boolean disponivelParaEmprestimo) {
        this.titulo = titulo;
        this.autor = autor;
        this.genero = genero;
        this.disponivelParaEmprestimo = disponivelParaEmprestimo;
    }

    // Métodos da classe Livro
}

```

Neste exemplo, a classe Livro recebe um objeto Autor no construtor, em vez de criar uma instância de Autor internamente.

➊ Aula 5.3 - Padrões de Projeto mais comuns em Orientação a Objetos

Design patterns são soluções reutilizáveis para problemas comuns de design de software.

Eles são como modelos que você pode personalizar para resolver um problema específico de design em seu código.

Esses padrões ajudam a comunicar de forma eficiente e padronizada entre os membros da equipe de desenvolvimento, acelerando o processo de desenvolvimento e melhorando a qualidade do código.

Os design patterns são como receitas que fornecem soluções testadas e comprovadas para problemas recorrentes no desenvolvimento de software.

Eles não são designs acabados que podem ser transformados diretamente em código, mas sim descrições ou modelos de como resolver um problema que podem ser aplicados em diferentes situações.

Esses padrões ajudam a prevenir problemas sutis que podem causar grandes problemas e melhoraram a legibilidade do código.

Se você quer aprender mais sobre padrões de projeto, veja os artigos do site [Refactoring Guru](#), que é uma das melhores referências sobre o assunto.

Vamos ver alguns padrões de projeto mais comuns para Orientação a Objetos e que funcionam no contexto do nosso projeto:

Padrão Decorator (Decorator Pattern):

O padrão Decorator permite adicionar funcionalidades a objetos existentes sem alterar sua estrutura. É como adicionar camadas de decoração a um objeto. Vamos ver um exemplo de aplicação:

```
interface Publicavel {
    void publicar();
}

class Autor implements Publicavel {
    public void publicar() {
        System.out.println("Publicando como autor...");
    }
}

class UsuarioDecorator implements Publicavel {
    private Publicavel usuario;

    public UsuarioDecorator(Publicavel usuario) {
        this.usuario = usuario;
    }

    public void publicar() {
        usuario.publicar();
        System.out.println("Publicando como usuário...");
    }
}

// Uso do padrão Decorator
Publicavel autor = new Autor();
Publicavel usuarioDecorado = new UsuarioDecorator(autor);
usuarioDecorado.publicar();
```

Padrão Estratégia (Strategy Pattern):

O padrão Estratégia permite definir uma família de algoritmos, encapsula-los e torná-los intercambiáveis. É como ter diferentes estratégias para realizar uma mesma tarefa.

```
interface EstrategiaPublicacao {
    void publicar();
}

class PublicacaoBlog implements EstrategiaPublicacao {
    public void publicar() {
        System.out.println("Publicando no blog...");
    }
}

class PublicacaoArtigo implements EstrategiaPublicacao {
    public void publicar() {
        System.out.println("Publicando artigo...");
    }
}

// Uso do padrão Estratégia
EstrategiaPublicacao estrategia = new PublicacaoBlog();
estrategia.publicar();
```

Padrão Fábrica (Factory Pattern):

O padrão Fábrica permite criar objetos sem especificar a classe exata do objeto que será criado. É como ter uma fábrica que produz diferentes tipos de objetos

```
interface Pessoa {  
    void saudacao();  
}  
  
class Autor implements Pessoa {  
    public void saudacao() {  
        System.out.println("Olá, sou um autor!");  
    }  
}  
  
class Usuario implements Pessoa {  
    public void saudacao() {  
        System.out.println("Olá, sou um usuário!");  
    }  
}  
  
class PessoaFactory {  
    public Pessoa criarPessoa(String tipo) {  
        if (tipo.equals("autor")) {  
            return new Autor();  
        } else if (tipo.equals("usuario")) {  
            return new Usuario();  
        }  
        return null;  
    }  
}  
  
// Uso do padrão Fábrica  
PessoaFactory factory = new PessoaFactory();  
Pessoa pessoa = factory.criarPessoa("autor");  
pessoa.saudacao();
```

Exercícios:

Já implementamos o padrão decorator no nosso projeto. Agora, que tal tentar implementar o padrão strategy no nosso projeto?

Este padrão permite que o comportamento de uma classe seja alterado em tempo de execução, escolhendo entre diferentes estratégias (algoritmos) que podem ser intercambiáveis dentro do contexto da aplicação. Isso é particularmente útil para lidar com diferentes tipos de usuários (por exemplo, usuários que são autores de artigos versus usuários que apenas fazem empréstimos de livros) e suas ações específicas.

É importante dizer que essa aplicação tem fins apenas didáticos, NUNCA devemos escolher um padrão para implementar no nosso projeto mas sim, escolher o padrão que resolva um problema nosso.

Voltando a atividade, para facilitar, vamos listar como faremos essa implementação:

1.Reaproveitar a interface PublicavelInterface que tem o método publicar()

2.Criar Classes Concretas de Estratégia: Implementam diferentes estratégias de publicação.

- EstrategiaPublicacaoLivro: Específica para publicação de livros.
- EstrategiaPublicacaoArtigo: Específica para publicação de artigos.

3. Modificação na classe UsuarioDecorator (ou Autor): Incorpora uma referência a EstrategiaPublicacao para determinar como a publicação deve ser realizada.
 - Atributo: EstrategiaPublicacao estrategiaPublicacao;
 - Método: setEstrategiaPublicacao(EstrategiaPublicacao estrategia), publicar()
4. Crie uma classe de testes para validar a lógica.

Módulo 6.0 - Encerramento e Próximos passos

Para continuar evoluindo com Java, Sprint e outras tecnologias backend, acesse o site <https://roadmap.sh/>

Muito Obrigado!