

APOSTILA DE JAVA

CURSO BASICO

INDICE

APRESENTAÇÃO - ROAD MAP	I
JAVA	I
NÍVEL 1	I
Java - Fundamentos:	i
Conceitos de Orientação a Objetos:	i
Java - Manipulação de Erros:	ii
Java - Coleções:	ii
Java - Testes:	ii
Java - Pacotes:	ii
Estruturas de Dados:	ii
NÍVEL 2	III
JVM:	iii
Java - Gerenciamento da Memória:	iii
Spring Framework:	iii
Spring Boot:	iii
Build tools Java:	iv
Java - Persistência:	iv
NÍVEL 3	IV
Arquitetura de Microserviços:	iv
Java - Concorrência:	iv
Contêineres:	iv
Kafka:	v
Git e GitHub - Fundamentos:	v
HTTP - Fundamentos:	vi
JSON:	vi
Linha de comando - Fundamentos:	vi
Cloud - Fundamentos:	vi
SQL - Fundamentos:	vi
SOLID:	VII
CLEAN ARCHITECTURE:	VII
DESIGN PATTERNS:	VII
CLEAN CODE:	VII
CONCEITOS DE DESIGN ORIENTADO A DOMÍNIO (DOMAIN-DRIVEN DESIGN - DDD):	VII
HISTÓRIA DO JAVA	1
POR QUE JAVA TINHA COMO NOME "OAK"?	1
POR QUE A LINGUAGEM DE PROGRAMAÇÃO JAVA RECEBEU COMO NOME "JAVA"?	2
HISTÓRICO DE VERSÕES EM JAVA	2
RECURSOS DE JAVA	3
Simples	2
Orientada a objeto	2
Independência de plataforma	3
Segura	3
Robusto	4
Arquitetura neutra	4
Portável	4
Alta performance	5
Distribuída	5
Multithread	5
Dinâmico	5
C++ VERSUS JAVA	6
Nota	7
C++ EXAMPLE	7
JAVA EXAMPLE	7

PRIMEIRO PROGRAMA EM JAVA EXEMPLO DE HELLO WORLD	7
<i>Os requisitos para o exemplo do hello world em java</i>	8
<i>Criando o exemplo hello world</i>	8
<i>Parâmetros usados no Primeiro Programa Java</i>	8
COMO QUANTAS MANEIRAS PODEMOS ESCREVER UM PROGRAMA JAVA	10
ASSINATURAS VALIDAS PARA O MÉTODO MAIN DO JAVA	11
ASSINATURAS INVÁLIDAS PARA O MÉTODO MAIN DO JAVA	11
<i>Resolvendo o erro: "javac is not recognized as an internal or external command"?</i>	11
DETALHES INTERNOS DO PROGRAMA HELLO JAVA	11
O QUE ACONTECE EM TEMPO DE COMPILAÇÃO?	11
O QUE ACONTECE NO TEMPO DE EXECUÇÃO?	12
<i>Você consegue salvar arquivo fonte java por outro nome que não seja o nome da classe?</i>	12
<i>Você pode ter múltiplas classes em um arquivo fonte java?</i>	13
ARQUITETURA DA JVM (MÁQUINA VIRTUAL JAVA)	13
<i>O que é JVM</i>	13
<i>O que ela faz</i>	14
ARQUITETURA DA JVM	14
1) <i>Classloader</i>	14
2) <i>Área de Classe (Método)</i>	15
3) <i>Heap</i>	16
4) <i>Pilha</i>	16
5) <i>Registro do contador de programa</i>	16
6) <i>Pilha de método nativo</i>	16
7) <i>Mecanismo de Execução</i>	16
8) <i>Interface nativa Java</i>	16
CONFIGURANDO JAVA E MAVEN	17
<i>Tipo Boolean</i>	23
<i>Uma palavra sobre identificadores</i>	24
<i>Tipos de dados inteiros</i>	24
<i>Operações com inteiros</i>	26
<i>Tipo caracter</i>	27
<i>Tipos de ponto flutuante</i>	27
<i>Estruturas condicionais</i>	29
IF/ELSE	29
<i>Switch/case</i>	30
<i>Estruturas de repetição</i>	32
FOR	32
WHILE	34
O QUE É UM ARRAY EM JAVA?	35
QUAL A SINTAXE DO JAVA ARRAY?	35
QUAIS OS TIPOS DE ARRAYS EM JAVA?	36
<i>Arrays unidimensionais</i>	37
<i>Arrays multidimensionais</i>	37
QUAIS AS CARACTERÍSTICAS DOS ARRAYS EM JAVA?	37
<i>Elementos do array e tipos de dados</i>	38
<i>Índices do array</i>	38
<i>Dimensão do array</i>	38
COMO DECLARAR ARRAYS EM JAVA?	38
COMO DESCOBRIR O TAMANHO DE UM ARRAY?	39
COMO INICIALIZAR UM ARRAY?	41
COMO ALTERAR ELEMENTOS DE UM ARRAY?	42
<i>Confira 3 exemplos de aplicação para o Java array</i>	43
EM QUE USAMOS O ARRAY EM NOSSO DIA A DIA?	46

O QUE É A ARRAYLIST JAVA? _____	47
QUAIS AS PRINCIPAIS CARACTERÍSTICAS DA CLASSE ARRAYLIST JAVA? _____	48
QUAIS AS DIFERENÇAS ENTRE ARRAY E ARRAYLIST EM JAVA? _____	48
QUAL A HIERARQUIA DA CLASSE ARRAYLIST? _____	49
<i>Conjuntos</i> _____	50
<i>Listas</i> _____	50
<i>Filas</i> _____	51
<i>Mapas</i> _____	51
QUAIS OS PRINCIPAIS MÉTODOS USADOS COM A CLASSE ARRAYLIST? _____	52
<i>Confira 4 exemplos práticos de uso dos métodos!</i> _____	52
COMO ADOTAR, NA PRÁTICA? 3 EXEMPLOS DE USOS DA CLASSE ARRAYLIST _____	54
<i>Exemplo 1 — listando disciplinas oferecidas em uma escola</i> _____	54
<i>Exemplo 3 — Ordenando a ArrayList</i> _____	55
JAVA NÃO GENÉRICO VS COLEÇÃO GENÉRICA _____	56
O QUE É A INTERFACE JAVA MAP? _____	59
COMO FUNCIONA A HIERARQUIA DA INTERFACE JAVA MAP? _____	59
QUAL A SINTAXE DA INTERFACE JAVA MAP? _____	60
O QUE É HASHMAP()? _____	60
O QUE É HASHTABLE? _____	61
QUAIS OS MÉTODOS DA INTERFACE JAVA MAP? _____	62
<i>size()</i> _____	62
<i>isEmpty()</i> _____	63
<i>containsKey() e containsValue()</i> _____	63
<i>get()</i> _____	64
<i>put() e remove()</i> _____	65
OPERAÇÕES FUNCIONAIS NA INTERFACE JAVA MAP! _____	66
<i>Exemplo 1 — map.compute</i> _____	66
<i>Exemplo 2 — map.computePresent</i> _____	67
<i>Exemplo 3 — map.computeIfAbsent</i> _____	67
ARRAYS MULTIDIMENSIONAIS EM JAVA _____	68
MATRIZ BIDIMENSIONAL (MATRIZ 2D) _____	68
<i>Acessando elementos de matrizes bidimensionais</i> _____	70
MATRIZ TRIDIMENSIONAL (MATRIZ 3D) _____	71
<i>Acessando Elementos de Matrizes Tridimensionais</i> _____	73
CLASSIFIQUE UMA MATRIZ DE PARES USANDO JAVA PAIR AND COMPARATOR _____	77
<i>Operadores de atribuição em Java</i> _____	78
<i>Operadores aritméticos</i> _____	79
<i>Opções de operadores aritméticos</i> _____	79
<i>Operadores de incremento e decremento</i> _____	80
<i>Operadores de igualdade</i> _____	80
<i>Opções de operadores de igualdade</i> _____	81
<i>Operadores relacionais</i> _____	81
<i>Opções de operadores relacionais</i> _____	82
<i>Operadores lógicos</i> _____	82
<i>Opções de operadores de lógicos</i> _____	83
<i>Precedência de operadores</i> _____	83
<i>Exemplo prático</i> _____	84
INTRODUÇÃO A ORIENTAÇÃO AO OBJETO _____	85
O QUE É PROGRAMAÇÃO ORIENTADA A OBJETOS E QUAIS SÃO SEUS PILARES? _____	85
<i>Princípios de programação orientada a objetos</i> _____	85
O QUE SÃO OBJETOS? _____	85
O QUE SÃO CLASSES? _____	86
PRINCÍPIOS DA PROGRAMAÇÃO ORIENTADA A OBJETOS _____	88
<i>Encapsulamento</i> _____	89

Vantagens do encapsulamento em Java	94
<i>Herança</i>	94
Então o que exatamente é herdado?	94
Tipos de herança no Java	95
<i>Relacionamentos</i>	96
ABSTRAÇÃO	98
<i>Interface</i>	100
Vantagens das interfaces	103
Quando vamos querer mudar de uma classe para uma interface?	103
POLIMORFISMO	104
Sobrecarga de métodos	104
Regras a seguir para o polimorfismo	105
Sobrescrita de métodos	106
Tipos de objeto e tipos de referência	106
Casting de tipos de objeto	107
FLOAT:	109
DOUBLE:	109
BIGDECIMAL:	109
QUANDO DEVO USAR CADA UM DELES?	110
<i>Double vs BigDecimal</i>	111
JAVA STREAMS API: MANIPULANDO COLEÇÕES DE FORMA EFICIENTE	117
COMO CRIAR STREAMS	117
<i>Iterando sobre streams</i>	118
<i>Métodos das streams</i>	118
VISÃO GERAL	119
2. API DE FLUXO	119
2.1. <i>Criação de Stream</i>	119
2.2. <i>Multi-threading com fluxos</i>	120
3. OPERAÇÕES DE FLUXO	120
3.1. <i>Iteração</i>	120
3.2. <i>Filtragem</i>	121
3.3. <i>Mapeamento</i>	121
3.4. <i>Correspondência</i>	122
3.5. <i>Redução</i>	122
3.6. <i>Coleta</i>	122
O QUE É COLLECTIONS FRAMEWORK?	124
INTERFACES	125
IMPLEMENTAÇÕES	126
COMO APLICAR ADEQUADAMENTE A COLLECTIONS FRAMEWORK	128
<i>A interface List</i>	128
<i>Adicionando novo requisito – Ordem ascendente</i>	130
<i>Adicionando novo requisito – Novos dados</i>	131
<i>Classificação de objetos</i>	131
<i>A interface Iterator</i>	133
<i>A interface Set</i>	134
<i>A interface Map</i>	137
<i>Conclusões</i>	138
JAVA - CASTING DE TIPOS PRIMITIVOS	139
CASTING	139
COMO USAR FUNÇÕES LAMBDA EM JAVA	141
EXCEÇÕES E CONTROLE DE ERROS	149
MOTIVAÇÃO	149
EXCEÇÃO	150

EXERCÍCIO PARA COMEÇAR COM OS CONCEITOS	150
EXCEÇÕES DE RUNTIME MAIS COMUNS	155
ERROS	156
OUTRO TIPO DE EXCEÇÃO: CHECKED EXCEPTIONS	156
BOAS PRÁTICAS NO TRATAMENTO DE EXCEÇÕES	158
<i>Um pouco da grande família Throwable</i>	158
MAIS DE UM ERRO	159
LANÇANDO EXCEÇÕES	159
O QUE COLOCAR DENTRO DO TRY?	161
CRIANDO SEU PRÓPRIO TIPO DE EXCEÇÃO	162
PARA SABER MAIS: FINALLY	163
EXERCÍCIOS: EXCEÇÕES	163
CRIANDO ARQUIVO PARA LEITURA E ESCRITA	166
2.1 ManipuladorArquivo.java	166
2.1.1 Breve explicação	168
2.1.2 Método leitor	168
2.1.3 Método escritor	168
2.1.4 Importante	169
2.2 Principal.java	169
2.2.1 Breve explicação	170
3. Arquivo após utilização do Projeto	170
<i>Desenvolvendo a Aplicação Java</i>	171
INSTALANDO MYSQL E MYSQL WORKBENCH	174
INSTANDO O WORKBENCH	179
JDBC - JAVA DATABASE CONNECTIVITY	182
O QUE É JDBC?	182
O QUE É UM DRIVER?	182
A classe DriverManager	183
A interface Connection	184
Método close	185
Método isClosed	185
Método createStatement	185
Método prepareStatement	185
setAutoCommit	186
rollback	186
A interface Statement	187
Interface ResultSet	187
Conclusão	188
CONEXÃO COM BANCOS DE DADOS USANDO JDBC	189
<i>Consulta de dados</i>	192
<i>Manipulação de dados</i>	193
<i>Relação de algumas bases de dados</i>	195
<i>Exemplo de aplicação C.R.U.D. (Create, Read, Update, Delete)</i>	195
SQL – FUNDAMENTOS	204
O QUE É SQL E PARA QUE SERVE ?	204
CURIOSIDADES DO SQL	205
ANTES DO SQL	205
O SURGIMENTO DA SQL	205
POR QUE APRENDER SQL?	206
QUAIS SÃO AS VANTAGENS E DESVANTAGENS DA LINGUAGEM SQL?	206
Vantagens	206
Desvantagens	207

ONDE O SQL É USADO? _____	207
COMO O SQL FUNCIONA? _____	208
<i>Quais são os componentes de um sistema SQL?</i> _____	208
<i>O que são comandos SQL?</i> _____	209
<i>DDL (Data Definition Language)</i> _____	210
<i>DQL (Data Query Language)</i> _____	210
<i>DML (Data Manipulation Language)</i> _____	210
<i>DCL (Data Control Language)</i> _____	210
<i>TCL (Transaction Control Language)</i> _____	210
OS PRINCIPAIS COMANDOS SQL SÃO: _____	211
<i>O que são padrões SQL?</i> _____	211
<i>Sistemas de banco de dados que usam SQL</i> _____	213
<i>Qual o papel do DBA?</i> _____	213
<i>NoSQL versus SQL</i> _____	214
MODELAGEM DE DADOS _____	216
<i>Como está o mercado de trabalho para SQL?</i> _____	216
O SQL - COMANDOS BÁSICOS. _____	218
O QUE É SQL? _____	218
<i>SELECT</i> _____	219
DELETE _____	220
TRANSAÇÕES NO SQL: MANTENDO OS DADOS ÍNTEGROS E CONSISTENTES _____	222
PROBLEMAS QUE PODEMOS EVITAR COM O USO DE TRANSAÇÕES _____	222
TRANSAÇÕES IMPLÍCITAS _____	222
TRANSAÇÕES EXPLÍCITAS _____	222
SQL: CLÁUSULAS UPDATE E DELETE _____	225
INTRODUÇÃO: SQL — CLÁUSULAS UPDATE E DELETE _____	225
<i>Cláusula UPDATE</i> _____	225
<i>Cláusula DELETE</i> _____	226
UPDATE Com WHERE _____	227
<i>Chave primária</i> _____	231
CONCLUSÃO _____	231
SQL: CONSULTAS COM SELECT _____	232
SELECT SIMPLES _____	232
SELECT COM WHERE _____	233
<i>IN</i> _____	233
<i>BETWEEN</i> _____	234
<i>LIKE</i> _____	234
<i>ORDER BY</i> _____	234
<i>SELECT simples</i> _____	235
FUNÇÕES DE AGRUPAMENTO _____	239
AGRUPAMENTO _____	240

Apresentação - Road Map

Java

Nível 1

Java - Fundamentos:

- Java é uma linguagem de programação amplamente usada para codificar aplicações Web. Java é uma linguagem multiplataforma, orientada a objetos e centrada em rede que pode ser usada como uma plataforma em si. É uma linguagem de programação rápida, segura e confiável para codificar tudo, desde aplicações móveis e software empresarial até aplicações de big data e tecnologias do servidor.
- Conhecer os tipos primitivos
- Declarar variáveis, considerando os diferentes tipos
- Usar estruturas condicionais ('if', 'else')
- Conhecer os operadores de atribuição e comparação
- Usar estruturas de repetição e laços ('while', 'for')
- Usar funções, passando parâmetros e argumentos
- Manipular métodos
- Manipular arrays e listas
- Obter dados de uma API
- Criar construtores

Conceitos de Orientação a Objetos:

A Programação Orientada a Objetos é um paradigma de programação de software baseado na composição e interação entre diversas unidades chamadas de 'objetos' e as classes, que contêm uma identidade, propriedades e métodos). Ela é baseada em quatro componentes da programação: abstração digital, encapsulamento, herança e polimorfismo.

- Como funcionam objetos
- Criar e utilizar construtores
- O que são classes
- Criar e utilizar métodos
- Como funciona encapsulamento
- O que é herança
- O que é polimorfismo
- Como funcionam interfaces
- O que são abstrações

Java - Manipulação de Erros:

O tratamento de erros refere-se aos procedimentos de resposta e recuperação de condições de erro presentes em um aplicativo de software. Em outras palavras, é o processo composto de antecipação, detecção e resolução de erros de aplicação, de programação ou de comunicação.

- Tratar exceções pré-definidas
- Uso de 'try' e 'catch'
- Criar exceções específicas
- Fazer o processo de Debug

Java - Coleções:

Uma coleção representa um grupo de objetos, conhecidos como seus elementos. Eles são como recipientes que agrupam vários itens em uma única unidade. Algumas coleções permitem a duplicação de elementos e outras não. Algumas são ordenadas e outras não ordenadas.

- Aprender os usos e diferenças entre List, Set e Map
- Aprender os usos e diferenças entre Equals e hashCode
- Saiba trabalhar com ArrayList, LinkedList ou Vector
- Classes Wrappers

Java - Testes:

O teste de software é o processo de avaliação e verificação de que um software realmente faz o que deveria fazer. Os benefícios dos testes incluem a prevenção de bugs, a redução dos custos de desenvolvimento e a melhoria do desempenho.

- Usar testes unitários
- Usar testes de integração
- Usar testes de comportamento (behavior)
- Usar mocks

Java - Pacotes:

Um pacote (package) em Java é usado para agrupar classes relacionadas, de forma semelhante a uma pasta em um diretório de arquivos. Os pacotes são usados para evitar conflitos de nomes e para escrever um código de melhor manutenção.

Use imports e organize o seu código através de packages

- Conhecer a java.lang
- Entender a imutabilidade e a classe String
- Entender a classe java.lang.Object
- Conhecer a java.io

Estruturas de Dados:

No contexto dos computadores, uma estrutura de dados é uma forma específica de armazenar e organizar os dados na memória do computador para que esses dados possam ser facilmente recuperados e utilizados de forma eficiente quando necessário posteriormente.

- Conhecer as principais estruturas de dados
- Implementar as principais estruturas de dados

Nível 2

JVM:

Máquina virtual Java (em inglês, Java Virtual Machine, JVM) é um programa que carrega e executa os aplicativos Java, convertendo os bytecodes em código executável de máquina. A JVM é responsável pelo gerenciamento dos aplicativos, à medida que são executados. Graças à máquina virtual Java, os programas escritos em Java podem funcionar em qualquer plataforma de hardware e software que possua uma versão da JVM, tornando assim essas aplicações independentes da plataforma onde funcionam.

Entender como funciona a máquina virtual do Java

Java - Gerenciamento da Memória:

Em Java, o gerenciamento de memória é o processo de alocação e desalocação de objetos, chamado de gerenciamento de memória. Java faz o gerenciamento de memória automaticamente. Java usa um sistema de gerenciamento automático de memória chamado de Garbage Collector (coletor de lixo). Assim, não somos obrigados a implementar a lógica de gerenciamento de memória em nossa aplicação.

- Entender como funciona a memória e seu gerenciamento em Java
- Entender como funciona a memória o Garbage Collector

Spring Framework:

O Spring é um framework open source para a plataforma Java. Trata-se de um framework não intrusivo, baseado nos padrões de projeto (design patterns) de inversão de controle (IoC) e injeção de dependência. No Spring o contêiner se encarrega de "instanciar" classes de uma aplicação Java e definir as dependências entre elas através de um arquivo de configuração em formato XML, inferências do framework, o que é chamado de auto-wiring ou ainda anotações nas classes, métodos e propriedades. Dessa forma, o Spring permite o baixo acoplamento entre classes de uma aplicação orientada a objetos.

- Entender o conceito de Injeção de Dependências
- Entender o padrão MVC
- Usar o Spring Data para manipular dados

Spring Boot:

O Spring Boot é um framework de código aberto baseado em Java usado para criar microserviços com o Spring Framework. Ele é usado para construir aplicações Spring independentes e prontas para produção.

- Criar aplicações Spring standalone
- Usar os servidores HTTP embutidos

Build tools Java:

Uma build tool é um sistema que permite automatizar todas as tarefas rotineiras de um projeto de uma forma organizada e que evite que o desenvolvedor tenha que perder tempo. Em outras palavras, adicionar uma nova biblioteca, realização de testes, empacotamento e deploy, ou até mesmo, a compatibilidade entre as diversas IDEs são tarefas facilmente resolvidas com uma build tool.

Conheça as principais ferramentas de build do Java, como Maven, Jenkins, Apache Ant, Gradle, etc., e como usá-las

Java - Persistência:

O conceito de "persistência de dados" refere-se a garantir que as informações inseridas na aplicação serão armazenadas em um meio em que possam ser recuperadas de forma consistente. Ou seja, são registros permanentes e que não são perdidos quando há o encerramento da sessão.

- Entender sobre JDBC e JPA
- Usar frameworks como Spring Data e Hibernate
- Comunicar-se com um banco de dados relacional
- Entender a diferença entre relacionamentos EAGER e LAZY
- Planejar queries com join fetch
- Encapsular o acesso em um DAO
- Entender como a memória funciona nessa situação

Nível 3

Arquitetura de Microserviços:

Microserviços são uma abordagem de arquitetura na qual o software consiste de pequenos serviços independentes que se comunicam entre si e são organizados de acordo com seus domínios de negócio.

- Aprender o conceito de arquitetura planejada para microserviços
- Realizar a comunicação usando APIs
- Melhorar a escalabilidade de um sistema

Java - Concorrência:

Programação concorrente é um paradigma de programação para a construção de programas que fazem uso da execução simultânea de várias tarefas computacionais interativas, que podem ser implementadas como programas separados ou como um conjunto de threads criadas por um único programa.

- Executar tarefas simultaneamente
- Colocar tarefas para aguardar até que um determinado evento ocorra
- Entender como a memória funciona nessa situação

Contêineres:

Os contêineres são pacotes de software que contêm todos os elementos necessários para serem executados em qualquer ambiente. Gerenciamento de contêineres é uma área crucial na computação em nuvem e DevOps, que

envolve o uso de tecnologias para automatizar o processo de criação, implantação, escalonamento e monitoramento de contêineres. Contêineres são unidades de software padronizadas que permitem aos desenvolvedores empacotar todas as dependências de um aplicativo (código, bibliotecas, configurações, etc.) em um único pacote. Isso permite que o aplicativo seja executado de forma consistente em qualquer ambiente de infraestrutura.

A tecnologia de contêineres, como exemplificada pelo Docker, fornece um ambiente consistente e portátil para desenvolvimento, teste e implantação de aplicativos, o que é vital para o trabalho eficiente de engenharia de dados.

Além disso, o Kubernetes, um sistema de orquestração de contêineres, permite o gerenciamento, a automação e a escalabilidade de aplicações baseadas em contêineres em ambientes de produção. Dominar esses conceitos e tecnologias possibilita a engenheiros de dados construir e manter pipelines de dados eficientes e confiáveis.

O Kubernetes (também conhecido como k8s ou kube) é uma plataforma de orquestração de containers open source que automatiza grande parte dos processos manuais necessários para implantar, gerenciar e escalar aplicações em containers.

- Isolar seu software para funcionar independentemente
- Implantar software em clusters
- Modularizar seu sistema em pacotes menores
- Conhecer a plataforma Docker
- Conhecer Kubernetes

Kafka:

O Apache Kafka é uma plataforma distribuída de transmissão de dados que é capaz de publicar, subscrever, armazenar e processar fluxos de registro em tempo real. Essa plataforma foi desenvolvida para processar fluxos de dados provenientes de diversas fontes e entregá-los a vários clientes.

- Utilizar o Kafka para comunicação assíncrona
- Criar microsserviços com Kafka
- Criar produtores e consumidores
- Entender como usar o Kafka para paralelismo e execução serializada
- Obter garantias relativas ao envio ou entrega das mensagens

Habilidade Auxiliar: Infraestrutura

Git e GitHub - Fundamentos:

- Git é um sistema de controle de versão distribuído gratuito e de código aberto projetado para lidar com tudo, desde projetos pequenos a muito grandes com velocidade e eficiência.
- GitHub é um serviço de hospedagem para desenvolvimento de software e controle de versão usando Git.
- Criar um repositório
- Clonar um repositório
- Fazer commit, push e pull de e para o repositório

- Reverter um commit
- Criar branches e pull requests
- Lidar com merge e conflitos

HTTP - Fundamentos:

- HTTP significa Hyper Text Transfer Protocol. A comunicação entre computadores cliente e servidores web é feita enviando solicitações HTTP e recebendo respostas HTTP.
- Entender a diferença dos verbos HTTP
- Testar os requests e ver os status codes no navegador
- Saber fazer uma requisição HTTP na linha de comando com WGET
- Baixar uma imagem com WGET
- Fazer um post

JSON:

JSON significa JavaScript Object Notation (notação de objeto JavaScript). É um formato de texto para armazenar e transmitir dados.

- Criar um objeto
- Transformar um objeto em uma string
- Transformar uma string em objeto
- Manipular um objeto

Linha de comando - Fundamentos:

- - CLI é um programa de linha de comando que aceita entradas de texto para executar funções do sistema operacional.
- - Conhecer os principais comandos

Cloud - Fundamentos:

Cloud, ou computação em nuvem é a distribuição de serviços de computação pela Internet usando um modelo de preço pago conforme o uso. Uma nuvem é composta de vários recursos de computação, que abrangem desde os próprios computadores (ou instâncias, na terminologia de nuvem) até redes, armazenamento, bancos de dados e o que estiver em torno deles. Ou seja, tudo o que normalmente é necessário para montar o equivalente a uma sala de servidores, ou mesmo um data center completo, estará pronto para ser utilizado, configurado e executado.

- Conhecer a diferença entre IaaS, PaaS e SaaS
- Conhecer os maiores provedores de cloud
- Especializar-se em algum provedor

SQL - Fundamentos:

SQL (Structured Query Language, traduzindo, Linguagem de Consulta Estruturada) é uma linguagem de programação padronizada que é usada para

gerenciar bancos de dados relacionais e realizar várias operações sobre os dados neles contidos.

- Conhecer os comandos mais comuns do SQL
- Usar SELECT para consultar uma tabela
- Usar INSERT para inserir dados em uma tabela
- Usar UPDATE para atualizar uma tabela
- Usar DELETE para remover dados de uma tabela
- Usar JOIN para conectar os dados de múltiplas tabelas
- Conhecer as cláusulas (FROM, ORDER BY, etc)
- Habilidade Auxiliar: Boas práticas

SOLID:

O Solid possui cinco princípios considerados como boas práticas no desenvolvimento de software que ajudam os programadores a escrever os códigos mais limpos, separando as responsabilidades, diminuindo acoplamentos, facilitando na refatoração e estimulando o reaproveitamento do código.

Clean Architecture:

A Clean Architecture (Arquitetura Limpa) é uma forma de desenvolver software, de tal forma que apenas olhando para o código fonte de um programa, você deve ser capaz de dizer o que o programa faz.

Design Patterns:

Na engenharia de software, um "padrão de projeto" (Design Pattern em inglês) é uma solução geral e reutilizável para um problema que ocorre normalmente dentro de um determinado contexto de projeto de software.

- Conhecer e aplicar os principais Design Patterns

Clean Code:

- Aplicar técnicas simples que visam facilitar a escrita e leitura de um código
- Refatorar seu código para que fique mais claro

Conceitos de Design Orientado a Domínio (Domain-Driven Design - DDD):

O Design Orientado a Domínio (DDD) é uma abordagem ao projeto e desenvolvimento de software que é primeiramente informado pelos requisitos de negócios. Os componentes do programa (objetos, classes, matrizes, etc.) indicam a indústria, setor ou domínio empresarial em que o negócio opera.

- Modelar domínios de forma efetiva
- Basear projetos complexos em modelos do domínio
- Conhecer os blocos de construção de DDD

História do Java

Java foi originalmente designada para interagir com televisão, mas também foi avançada para a indústria de TV a cabo digital no momento. A história do java começa com o time Green. Os membros da equipe java (também conhecido como **Time Green**), iniciaram este projeto para desenvolver uma linguagem para dispositivos digitais tais como receptor de televisão, TVs, etc. Entretanto, foi adaptada para programação no âmbito da internet. Mais tarde, a tecnologia java foi incorporada pelo Netscape.

Os princípios para criação de programas java eram "simplicidade, robustez, portabilidade, Independência de plataforma, segurança, alta performance, multithread, arquitetura neutra, orientação a objeto, interpretada e dinâmica". **Java** foi desenvolvida por James Gosling, que é conhecido como o pai do java, em 1995. James Gosling e os membros do seu time começaram o projeto no início dos anos 90.

Atualmente, a programação java é usada na internet, dispositivos móveis, jogos, soluções de e-commerce, etc. Abaixo temos alguns tópicos importantes que descrevem a história do java.



1) **James Gosling**, Mike Sheridan, and Patrick Naughton iniciaram o projeto da linguagem java em junho de 1991. O pequeno time de engenheiros da sun chamado Time Green.

2) Inicialmente projetado para sistemas pequenos e **incorporados** em aparelhos eletrônicos, como receptores de televisão.

3) Antigamente, era chamado "**Greentalk**" por James Gosling, e o arquivo de extensão era .gt.

4) Depois que foi chamado **Oak** (carvalho) e foi desenvolvido como parte do projeto Green.

Por que java tinha como nome "Oak"?

5) **Por que Oak?** Oak é um símbolo de estratégia e escolhida como uma árvore nacional de muitos países como EUA, França, Alemanha, Romania, etc.

6) Em 1995, Oak teve seu nome modificado para "**Java**" porque já era uma marca comercial da Oak Tecnologias.

Por que a linguagem de programação java recebeu como nome "Java"?

7) **Por que eles escolheram o nome java para a linguagem java?** O time se reuniram para escolher um novo nome. As palavras sugestivas eram "dinâmica", "revolucionária", Silk, "alavancar", "DNA", etc. Eles procuraram algo que refletisse a essência da tecnologia: revolucionária, dinâmica, animado, legal, único, de fácil escrita e pronuncia divertida.

Segundo James Gosling, "Java foi das principais opções junto com **Silk**". Como java era tão único, muitos dos membros preferiram Java do que outros nomes.

8) Java é uma ilha da Indonésia onde o primeiro café foi produzido (conhecido como café java). É tipo um café expresso. O nome java foi escolhido por James Gosling enquanto estava tomando café perto do seu escritório.

9) Observe que java é apenas um nome, não um acrônimo.

10) Inicialmente desenvolvida por James Gosling na **Sun Microsystems** (que é agora uma subsidiária da Corporação Oracle) e release (lançamento) em 1995.

11) Em 1995, a revista Time chamou **Java um dos dez melhores produtos de 1995**.

12) JDK 1.0 release em (23 de Janeiro de 1996). Depois da primeira release do java, muitos recursos adicionais foram adicionado a linguagem. Agora java está sendo usado em aplicativos Windows, aplicações web, aplicações empresariais, aplicações móveis, cards (cartões), etc. A cada nova versão são adicionados novos recursos ao java.

Histórico de Versões em Java

Muitas versões do java já foram lançadas até agora. A atual release estável do java é a Java SE 21.

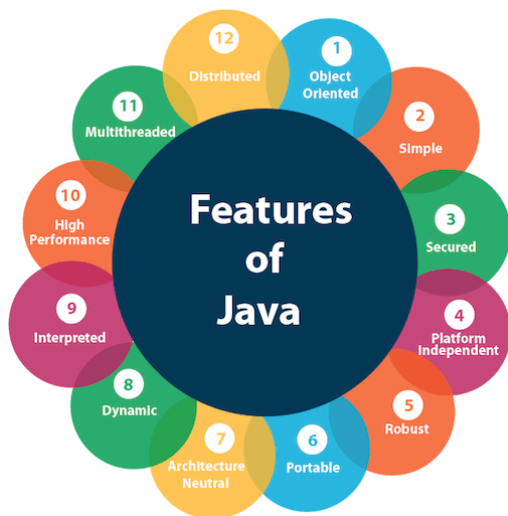
Oracle Java SE Support Roadmap				
Release	GA Date	Premier Support Until	Extended Support Until	Sustaining Support
8 (LTS)	Março 2014	Março 2022	Dezembro 2030	Indefinido
9 - 10 (non-LTS)	Setembro 2017 - Março 2018	Março 2018 - Setembro 2018	Não Disponível	Indefinido
11 (LTS)	Setembro 2018	Setembro 2023	Janeiro 2032	Indefinido
12 - 16 (non-LTS)	Março 2019 - Março 2021	Setembro 2019 - Setembro 2021	Não Disponível	Indefinido
17 (LTS)	Setembro 2021	Setembro 2026	Setembro 2029	Indefinido
18 (non-LTS)	Março 2022	Setembro 2022	Não Disponível	Indefinido
19 (non-LTS)	Setembro 2022	Março 2023	Não Disponível	Indefinido

Oracle Java SE Support Roadmap				
Release	GA Date	Premier Support Until	Extended Support Until	Sustaining Support
20 (non-LTS)	Março 2023	Setembro 2023	Não Disponível	Indefinido
21 (LTS)	Setembro 2023	Setembro 2028	Setembro 2031	Indefinido
22 (non-LTS)	Março 2024	Setembro 2024	Não Disponível	Indefinido
23 (non-LTS)	Setembro 2024	Março 2025	Não Disponível	Indefinido
24 (non-LTS)	Março 2025	Setembro 2025	Não Disponível	Indefinido
25 (LTS)	Setembro 2025	Setembro 2030	Setembro 2033	Indefinido

Recursos de Java

O objetivo primário da criação linguagem de programação java foi construir uma portátil, simples e segura linguagem de programação. Além disso, existem também alguns excelentes recursos que desempenham um papel importante na popularidade dessa linguagem. Os recursos de java são conhecidos como *java buzzwords*.

A lista dos mais importantes recursos da linguagem java são dados abaixo.



1. Simples
2. Orientada a Objeto
3. Portável
4. Independente de plataforma
5. Segura
6. Robusta

7. Arquitetura neutra
8. Interpretada
9. Alta performance
10. Multithread
11. Distribuída
12. Dinâmica

Simples

Java é bem tranquila para aprender, e sua sintaxe é simples, clara e fácil para entender. De acordo com a Sun, a linguagem java é uma linguagem de programação simples porque:

- A sintaxe do java é baseada em C++ (tão fácil para programadores aprender depois de C++).
- Java tem removido muitos recursos complicados e raramente usados, por exemplo, ponteiro explícito, operador de sobrecarga, etc.
- Não há necessidade de ficar removendo objetos não referenciados porque existe um coletor de lixo (Garbage Collection) automático em java.

Orientada a objeto

Java é uma linguagem de programação orientada a objeto. Tudo em java é um objeto. Orientação a objetos significa que organizamos nosso software com uma combinação de diferentes tipos de objetos que incorporando dado e comportamento.

Programação Orientada a Objetos (POO) é uma metodologia que simplifica o desenvolvimento e manutenção de software por providenciar algumas regras.

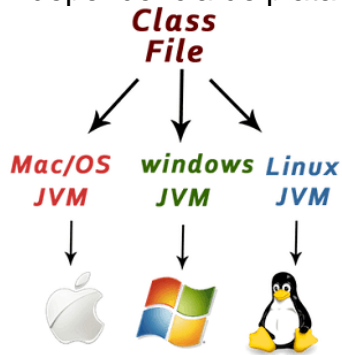
Conceitos básicos de POO:

1. Objeto
2. Classe
3. Herança
4. Polimorfismo
5. Abstração
6. Encapsulamento

.java, .class, bytecodes



Independência de plataforma



Java é independente de plataforma porque é diferente das outras linguagens como C, C++, etc. Que são compiladas em máquinas de plataforma específica enquanto Java é uma linguagem de escrita única, executada em qualquer lugar. A plataforma é o hardware ou ambiente de software em que um programa é executado.

Existem dois tipos de plataformas: baseada em software e baseada em hardware. Java prove uma plataforma baseada em software.

A plataforma Java difere de muitas outras plataformas no sentido que ela é uma plataforma baseada em software que roda em cima de outras plataformas baseadas em hardware. Possuindo dois componentes:

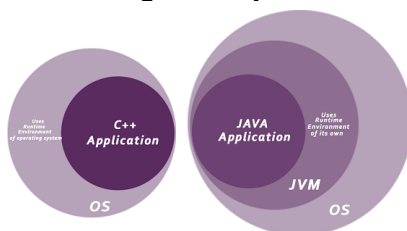
1. Ambiente de tempo de execução
2. API (Interface de Programação de Aplicativos)

O código Java pode ser rodado em múltiplas plataformas, por exemplo, Windows, Linux, Sun Solaris, Mac/OS, etc. É compilado por um compilador e convertido para em bytecode. Este bytecode é um código independente de plataforma porque pode ser executado em múltiplas plataformas, ou seja, escreva uma vez e execute em qualquer lugar (Write Once and Run Anywhere - WORA).

Segura

Java é melhor conhecida por sua segurança. Com java, nós podemos desenvolver sistemas livre de vírus. Java é segura porque:

- Nenhum ponteiro explícito
- Programas java são executados em uma máquina virtual



- Carregador de classe: Classloader em java é uma parte do Java Runtime Environment (JRE) que é usada para carregar classes java dentro da Java Virtual Machine (JVM) dinamicamente. Adiciona segurança, separando o pacote para as classes do sistema de arquivos local daquelas que são importadas de recursos de rede.
- Verificador de bytecode: Verifica o fragmentos de código em busca de código ilegal que possa violar o acesso a objetos.
- Gerenciador de Segurança: Determina qual recurso uma classe pode acessar tais como leitura e escrita no disco local.

A linguagem java fornece essas seguranças por padrão. Alguma segurança pode ser providenciada por um desenvolvedor de aplicações de maneira explicita por meio de SSL, JAAS, criptografia, etc.

Robusto

Robusto simplesmente significa forte. Java é robusto porque:

- Usa gerenciador de memória potente.
- Falta de ponteiros evitando assim alguns problemas de segurança.
- Há um garbage collection automático em java que roda na JVM para liberação de objetos que não são mais usados por uma aplicação java qualquer.
- Há um tratamento de exceção e um mecanismo de checagem de tipos em java. Todos esses pontos tornam o java robusto.

Arquitetura neutra

Java possui uma arquitetura neutra porque não há recursos dependentes da implementação, por exemplo, o tamanho dos tipos primitivos é fixo.

Em linguagem C, o tipo de dado int ocupa dois bytes de memória para uma arquitetura 32 bits e quatro bytes para uma arquitetura 64 bits. Entretanto, no java é utilizado quatro bytes de memória para as arquiteturas 32 e 64 bits,

Portável

Java é portátil porque facilita o transporte do bytecode do java para qualquer plataforma. Não requer nenhuma implementação.

Alta performance

Java é mais veloz do que outras tradicionais linguagens de programação interpretadas porque o bytecode é "próximo" do código nativo. Ainda é um pouco mais lento que uma linguagem compilada (por exemplo C++). Java é uma linguagem interpretada e é por isso que é mais lenta que as linguagens compiladas por exemplo, C, C++, etc.

Distribuída

Java é distribuída porque facilita usuários a criar aplicações distribuídas em java. RMI e EJB são usados para criar aplicações distribuídas. Este recurso do java nos permite acessar arquivos chamando os métodos de qualquer máquina na internet.

Multithread

Uma thread é como um programa separado. Executando simultaneamente. Podemos escrever programas java que lidam com muitas tarefas ao mesmo tempo, definindo vários encadeamentos. A principal vantagem do multithread é que ele não ocupa memória para cada thread. Ele compartilha uma área de memória comum. Threads são importantes para multimídia, aplicativos da web, etc.

Dinâmico

Java é uma linguagem dinâmica. Suporta carregamento dinâmico de classes. Isso significa que as classes são carregadas sob demanda. Também suporta funções vindas de suas linguagens nativas, ou seja, C e C++.

Java suporta compilação dinâmica e gerenciamento de memória automático (garbage collection / coleta de lixo).

C++ versus Java

Existem algumas diferenças e similaridades entre a linguagem de programação C++ e Java. Uma lista das principais diferenças entre C++ e java são dadas abaixo:

Índice de comparação	C++	Java
Independência de plataforma	Depende da plataforma	Não depende de plataforma
Principalmente usada para	Programação de sistemas	Programação de aplicações. Amplamente usada no Windows, web, enterprise e aplicativos mobile.
Objetivo do design	Foi projetado para programação de sistemas e aplicativos. Foi uma extensão da linguagem de programação C.	Foi projetado e criado como um interpretador para sistemas de impressão, mas depois foi estendida como um suporte a computação em rede. Foi projetada com um objetivo de ser fácil para usar e acessível para um mais amplo público.
Goto	Suporta a declaração goto	Não suporta a declaração goto
Herança múltipla	Suporta herança múltipla	Java não suporta herança múltipla através de class. Pode ser alcançada usando interfaces em java.
Sobrecarga de operadores	Suporta sobrecarga de operadores	Não suporta sobrecarga de operadores
Ponteiros	Suporta ponteiros. Você consegue utilizar ponteiros em programas C++	Suporta ponteiro internamente. Entretanto, você não pode trabalhar com ponteiros em programas java.
Compilador e interpretador	Usa somente compilador. C++ é compilado e executado usando o compilador que converte o código fonte em código de máquina, então, C++ é dependente de plataforma.	Usa ambos. O código fonte java é convertido em bytecode em tempo de compilação. O interpretador executa este bytecode em tempo de execução e produz uma saída. Java é interpretado e é por isso que é independente de plataforma.
Chamada por valor e chamada por referência	C++ suporta ambos.	Suporta somente chamada por valor. Não existe chamada por referência em java.
Structure e Union	C++ suporta structures e unions	Java não suporta structures e unions
Suporte a thread	C++ não tem suporte embutido para threads. Depende de bibliotecas de terceiros para suportar thread.	Java tem suporte a thread embutido
Documentação e Comentário	Não suporta comentário	Suporta comentário (<code>/** ... */</code>) para criar comentário para código fonte java.

Palavra reservada Virtual	Suporta virtual mas nós que decidimos se sobrescreve ou não uma função	Não tem a palavra reservada virtual. Nós podemos sobrescrever todos os métodos não estáticos por default. Em outras palavras, métodos não estáticos são virtual por padrão.
Shift para a direita sem sinal >>>	Não suporta operador >>>	Suporta operador >>> que preenche de zeros a esquerda para números negativos. Para números positivos, funciona da mesma forma que o operador >>
Árvore de Herança	Cria sempre uma nova árvore de herança	Java usa sempre uma única árvore de herança porque todas as classes são filhas da classe object em java. A classe objeto é a raiz da árvore de herança em java.
Hardware	É mais perto do hardware.	Não é tão interativo com o hardware
Orientação a objetos	É uma linguagem orientada a objetos. Entretanto, na linguagem C, hierarquia de única raiz não é possível.	É também uma linguagem orientada a objetos. Entretanto, tudo (com exceção dos tipos fundamentais) é um objeto em java. É uma hierarquia de raiz única, pois tudo é derivado de java.lang.Object.

Nota

- Java não suporta argumentos padrão como C++.
- Java não suporta cabeçalhos de arquivos como em C++. Java usa a palavra reservada import para incluir diferentes classes e métodos.

C++ Example

Arquivo: main.cpp

```
#include <iostream>

using namespace std;

int main() {
    cout << "Hello C++ Programming";

    return 0;
}
```

Java Example

Arquivo: Simple.java

```
class Simple {
    public static void main(String args[]) {
        System.out.println("Hello Java");
    }
}
```

Primeiro Programa em Java | Exemplo de Hello World

1. Requisitos de Software
2. Criando exemplo hello em java
3. Resolvendo o problema do javac não está sendo reconhecido

Como escrever um simples programa em java. Podemos escrever um simples programa java hello facilmente depois de instalar o JDK.

Para criar um simples programa em java., você precisa criar uma classe que contenha o método main. Vamos entender primeiro os pré-requisitos.

Os requisitos para o exemplo do hello world em java

Para executar algum programa java, você precisa:

- Instalação do JDK se você não tem instalado, baixe o JDK e instale.
- Definir o caminho do diretório do jdk/bin
- Cria o programa java
- Compila e executa o programa java

Criando o exemplo hello world

Vamos criar o programa hello java:

```
class Simple {  
    public static void main(String args[]) {  
        System.out.println("Hello Java");  
    }  
}
```

Salve este arquivo como Simple.java

Compilar: javac Simple.java

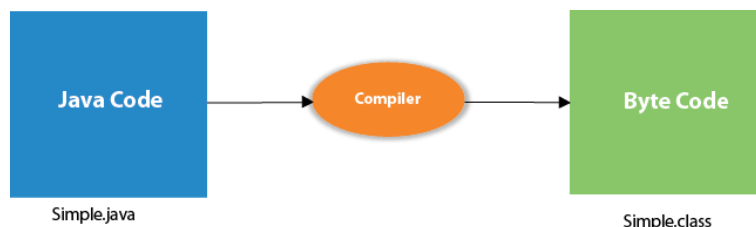
Executar: java Simple

Saída:

```
Hello Java
```

Fluxo de compilação:

Quando nós compilamos um programa java usando a ferramenta javac, o compilador java converte o código fonte em bytecode.



Parâmetros usados no Primeiro Programa Java

Vamos ver qual é o significado de class, public, static, void, main, String[], System.out.println().

- **class:** Palavra reservada que é usada para declarar um classe em java.
- **public:** Palavra reservada que é um modificador de acesso que representa visibilidade. Significa que é visível para todos.
- **static:** É uma palavra reservada. Se nós declararmos algum método como static, é conhecido como método estático. A principal vantagem do método estático é que não é necessário criar um objeto para invocar o método estático. O método principal é executado pela JVM, mas não requer a criação de um objeto para invocar o método principal. Por isso, economiza memória.
- **void:** É um tipo de retorno de método. Significa que não retorna nenhum valor.
- **main:** Representa o ponto de início do programa.
- **String args[]:** É usado para argumento de linha de comando. Nós aprenderemos isso depois.
- **out.println():** É usada imprimir declarações. Aqui, System é uma class, out é o objeto da classe PrintStream, print() é o método da classe PrintStream. Nós aprenderemos sobre o trabalho interno da declaração System.out.println() depois.

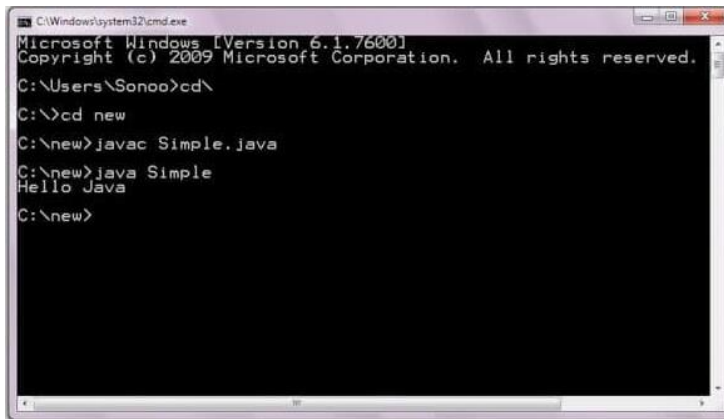
Para escrever o programa simple, você precisa abrir o notepad por **menu de início** → **Todos os programas** → **Acessórios** → **notepad** e escrever um programa simples que é apresentado abaixo:

```

class Simple{
public static void main(String args[]){
System.out.println("Hello Java");
}
}

```

Como exibido na imagem acima, escreva o programa simples de java no bloco de notas e salve-o como Simple.java. Para compilar e executar este programa, você precisa abrir o prompt de comando no **menu Iniciar** → **Todos os Programas** → **Acessórios** → **prompt de comando**.



```
C:\Windows\system32\cmd.exe
Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\Sonoo>cd\
C:\>cd new
C:\new>javac Simple.java
C:\new>java Simple
Hello Java
C:\new>
```

Para compilar e executar o programa acima, ir para seu diretório atual primeiro, meu diretório atual é c:\new. Escreva aqui.

Compilar: javac Simple.java

Executar: java Simple

Como quantas maneiras podemos escrever um programa java

Existe algumas maneiras de escrever um programa java. As modificações que podem se feitas em um programa java são apresentadas abaixo:

1) Alterando a sequência dos modificadores, o protótipo do método não é alterado em java.

Vamos ver um simples código do método main.

1. **static public void** main(String args[])

2) A notação subscrita no array java pode ser usada depois do tipo, antes da variável ou depois da variável.

Vamos ver a diferentes códigos para escrever o método main.

1. **public static void** main(String[] args)
2. **public static void** main(String []args)
3. **public static void** main(String args[])

3) Você pode fornecer suporte a var args para o método main passando três elipses (pontos)

Vamos ver um simples código usando varargs no método main. Nós aprenderemos sobre varargs depois no capítulo Java Novos Recursos.

1. **public static void** main(String? args)

4) Ter uma ponto e virgula no final da classe é opcional em java

Vamos ver um simples código.

```
1. class A{  
2.     static public void main(String... args){  
3.         System.out.println("hello java");  
4.     }  
5. };
```

Assinaturas validas para o método main do java

1. **public static void** main(String[] args)
2. **public static void** main(String []args)
3. **public static void** main(String args[])
4. **public static void** main(String... args)
5. **static public void** main(String[] args)
6. **public static final void** main(String[] args)
7. **final public static void** main(String[] args)
8. **final strictfp public static void** main(String[] args)

Assinaturas inválidas para o método main do java

1. **public void** main(String[] args)
2. **static void** main(String[] args)
3. **public void static** main(String[] args)
4. **abstract public static void** main(String[] args)

Resolvendo o erro: "javac is not recognized as an internal or external command"?

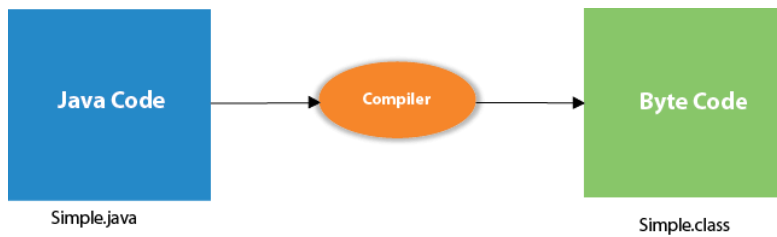
Se ocorrer um problema como o mostrado na imagem abaixo, você precisa definir o path. Desde o DOS não conhecer o javac ou java, nós precisamos definir o path. O path não é obrigatório neste caso, se você salvar seu programa dentro do diretório JDK/bin.

Detalhes internos do programa hello java

Na página anterior, nós temos aprendemos sobre o primeiro programa, como compilar e executar o primeiro programa java. Aqui, nós vamos aprender, o que acontece quando o programa java está compilando e executando. Mais afrente, nós iremos ver alguns questões baseadas no primeiro programa.

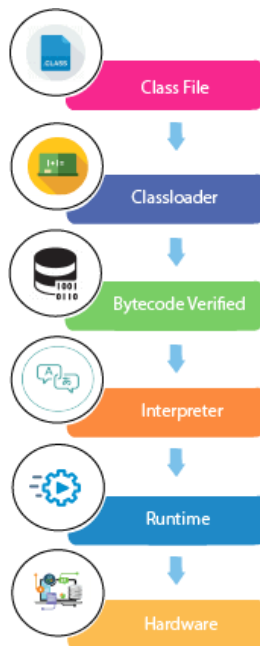
O que acontece em tempo de compilação?

Em tempo de compilação, o arquivo java é compilado pelo compilador java (ele não interage com o sistema operacional) e converte o código java em bytecode.



O que acontece no tempo de execução?

Em tempo de execução, as seguintes etapas são executadas:



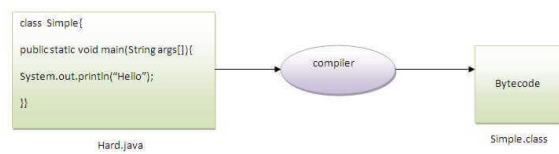
Carregador de classe: no subsistema da JVM que é usada para carregar arquivos de classe.

Verificador de bytecode:

- **checa os fragmentos do código procurando código ilegal que pode violar o direito de acesso aos objetos.**
- **Interpretador: ler o stream do bytecode então executa as instruções**

Você consegue salvar arquivo fonte java por outro nome que não seja o nome da classe?

Sim, se a classe não é pública. É explicado na figura abaixo:

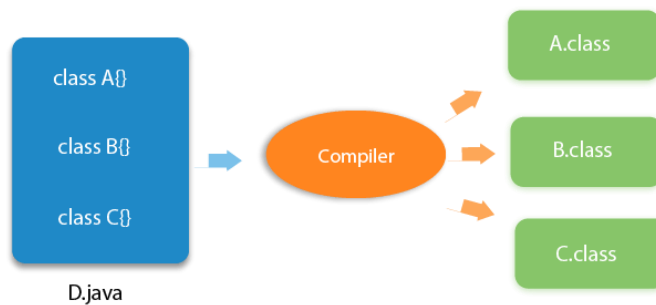


Compilar: javac Hard.java

Executar: java Simple

Você pode ter múltiplas classes em um arquivo fonte java?

Sim, como mostra a figura abaixo:



Arquitetura da JVM (Máquina Virtual Java)

JVM (Java Virtual Machine) é uma máquina abstrata. É uma especificação que fornece um ambiente de tempo de execução no qual o bytecode do java pode ser executado.

As JVMs estão disponíveis para muitas plataformas de hardware e software (ou seja, a JVM depende da plataforma).

O que é JVM

Isto é:

1. **Uma especificação** em que o trabalho da Java Virtual Machine é especificado. Mas o provedor de implementação é independente para escolher o algoritmo. Sua implementação foi fornecida pela Oracle e outras empresas.

2. **Uma implementação** Sua implementação é conhecida como JRE (Java Runtime Environment).
3. **Instância de tempo de execução** Sempre que você grava o comando java no prompt de comando para executar a classe java, uma instância da JVM é criada.

O que ela faz

A JVM executa a seguinte operação:

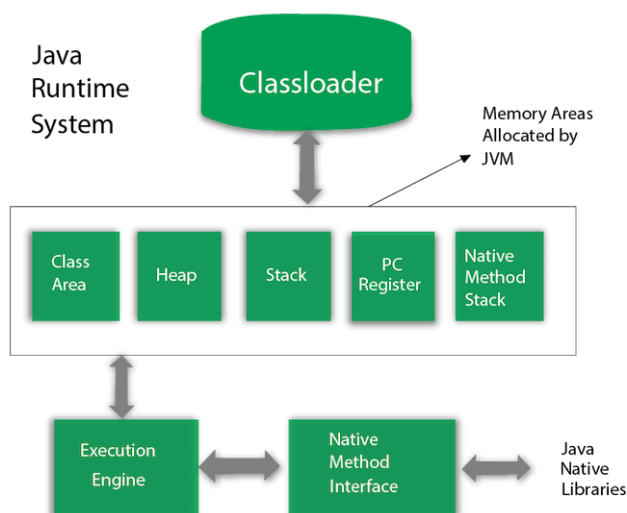
- Carrega código
- Verifica código
- Executa código
- Fornece ambiente de tempo de execução

A JVM fornece definições para:

- Área de memória
- Formato de arquivo de classe
- Conjunto de registros
- Pilha de garbage collector
- Relatório de erros fatais etc.

Arquitetura da JVM

Vamos entender a arquitetura interna da JVM. Contém carregador de classe, área de memória, mecanismo de execução etc.



1) Classloader

O Classloader é um subsistema da JVM que é usado para carregar arquivos de classe. Sempre que executamos o programa java, ele é carregado primeiro pelo carregador de classe. Existem três carregadores de classes integrados em Java.

1. **Bootstrap carregador de classe:** Este é o primeiro carregador de classes, que é a superclasse do carregador de classes Extension. Carrega o arquivo *jar*, que contém todos os arquivos de classe do Java Standard Edition, como as classes de pacote *java.lang*, classes de pacote *java.net*, classes de pacote *java.util*, classes de pacote *java.io*, classes de pacote *java.sql* etc.
2. **Extension carregador de classe:** Este é o carregador de classes filho do Bootstrap e o carregador de classes pai do System classloader. Ele carrega os arquivos jar localizados no diretório \$ JAVA_HOME/jre/lib/ext.
3. **Sistema / Aplicativo carregador de classe:** Este é o carregador de classes filho do carregador de classes Extension. Carrega os arquivos de classe do classpath. Por padrão, o caminho da classe é definido como o diretório atual. Você pode alterar o caminho de classe usando a opção "-cp" ou "-classpath". Também é conhecido como carregador de classes de aplicativos.

1. //Vamos ver um exemplo para imprimir o nome do carregador de classe

```
public class ClassLoaderExample
{
    public static void main(String[] args)
    {
        // Vamos imprimir o nome do carregador de classes da classe atual.
        // O carregador de classe Aplicativo / Sistema carregará esta classe
        Class c=ClassLoaderExample.class;
        System.out.println(c.getClassLoader());
        /// Se imprimirmos o nome do carregador de classes de String, ele
        imprimirá nulo porque é uma classe incorporada que é encontrada em rt.jar,
        portanto é carregada pelo carregador de classe Bootstrap
        System.out.println(String.class.getClassLoader());
    }
}
```

Saída:

```
sun.misc.Launcher$AppClassLoader@4e0e2f2a
null
```

Estes são os carregadores de classes internos fornecidos pelo Java. Se você deseja criar seu próprio carregador de classe, é necessário estender a classe *ClassLoader*.

2) Área de Classe (Método)

Classe (método) A área armazena estruturas por classe, como o pool constante de tempo de execução, dados de campo e método, o código para métodos.

3) Heap

É a área de dados de tempo de execução na qual os objetos são alocados.

4) Pilha

Java Stack armazena quadros. Ele contém variáveis locais e resultados parciais e desempenha um papel na invocação e no retorno do método.

Cada encadeamento possui uma pilha JVM privada, criada ao mesmo tempo que encadeamento.

Um novo quadro é criado toda vez que um método é chamado. Um quadro é destruído quando sua chamada de método é concluída.

5) Registro do contador de programa

O registro do PC (contador de programa) contém o endereço da instrução da máquina virtual Java atualmente em execução.

6) Pilha de método nativo

Ele contém todos os métodos nativos usados no aplicativo.

7) Mecanismo de Execução

Contém:

1. **Um processador virtual**
2. **Interpretador:** Lê o fluxo do bytecode e execute as instruções.
3. **Compilador Just-In-Time (JIT):** É usado para melhorar o desempenho. O JIT compila partes do código de bytes que possuem funcionalidade semelhante ao mesmo tempo e, portanto, reduz a quantidade de tempo necessária para a compilação. Aqui, o termo "compilador" refere-se a um tradutor do conjunto de instruções de uma Java Virtual Machine (JVM) para o conjunto de instruções de uma CPU específica.

8) Interface nativa Java

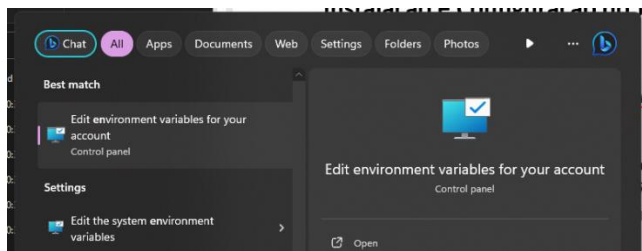
Java Interface Nativa (JNI) é uma estrutura que fornece uma interface para se comunicar com outro aplicativo escrito em outra linguagem como C, C ++, Assembly etc. Java usa a estrutura JNI para enviar saída ao console ou interagir com as bibliotecas do sistema operacional.

Instalação e Configuração do Java

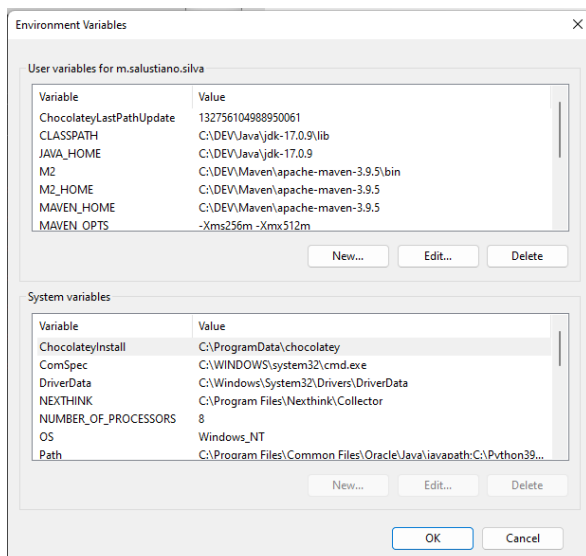
Configurando Java e Maven

A variável JAVA_HOME é basicamente o “lar do java”, é onde o JDK encontra-se instalado, neste caso fica em “C:\DEV\Java\jdk-17.0.9”, podendo variar o nome de acordo com a versão instalada. Para configurar deve-se seguir os passos abaixo:

Na barra “Search” colocar Environment e abrir



```
CLASSPATH = C:\DEV\Java\jdk-17.0.9\lib
JAVA_HOME = C:\DEV\Java\jdk-17.0.9
M2 = C:\Apache\apache-maven-3.8.5\bin
M2_HOME = C:\Apache\apache-maven-3.8.5
MAVEN_HOME = C:\Apache\apache-maven-3.8.5
MAVEN_OPTS = -Xms512m -Xmx1024m
```

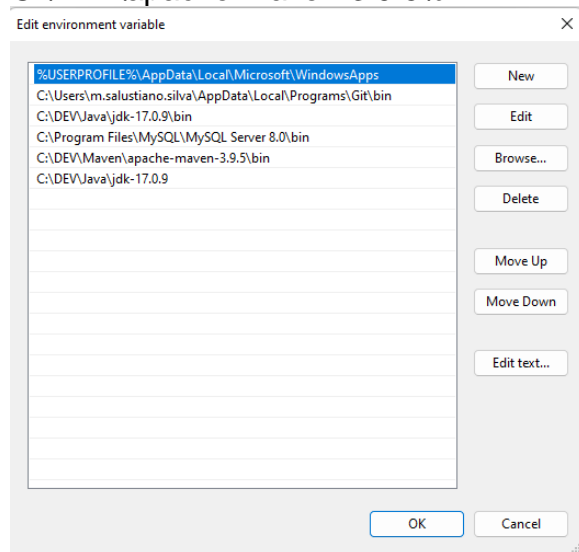


Na variável PATH, colocar o caminho do Java e Maven

C:\DEV\Java\jdk-17.0.9\bin

C:\DEV\Java\jdk-17.0.9

C:\DEV\apache-maven-3.9.5\bin



Verificando a instalação

Abrir o Prompt e rodar os seguintes comandos:

java -version

```
C:\Users\m.salustiano.silva>java -version
java version "17.0.7" 2023-04-18 LTS
Java(TM) SE Runtime Environment (build 17.0.7+8-LTS-224)
Java HotSpot(TM) 64-Bit Server VM (build 17.0.7+8-LTS-224, mixed mode, sharing)
C:\Users\m.salustiano.silva>
```

javac -version

```
C:\Users\m.salustiano.silva>javac -version
javac 17.0.7
C:\Users\m.salustiano.silva>
```

mvn -version

```
C:\Users\m.salustiano.silva>mvn -version
Apache Maven 3.9.5 (57804ffe001d7215b5e7bcb531cf83df38f93546)
Maven home: C:\DEV\Maven\apache-maven-3.9.5
Java version: 17.0.9, vendor: Oracle Corporation, runtime: C:\DEV\Java\jdk-17.0.9
Default locale: en_US, platform encoding: Cp1252
OS name: "windows 11", version: "10.0", arch: "amd64", family: "windows"
C:\Users\m.salustiano.silva>
```

Configurando o Eclipse e Lombok.

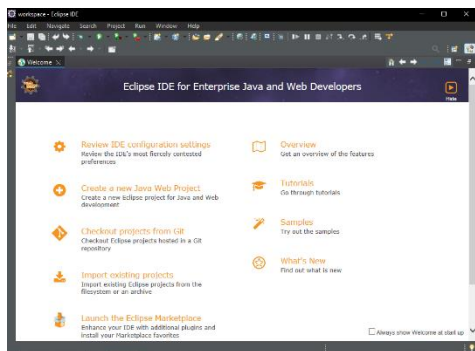
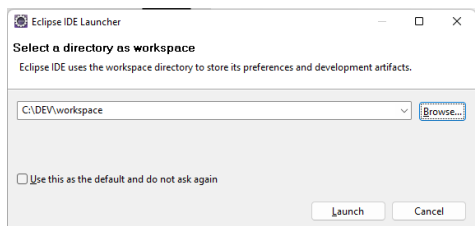
Extrair o zip do Eclipse para a uma pasta dentro de C:\DEV

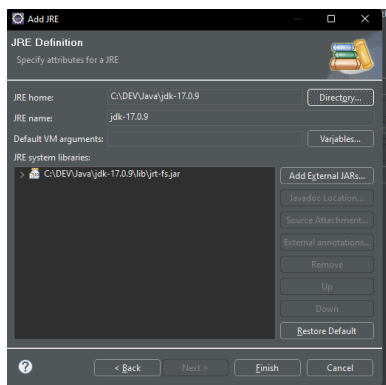
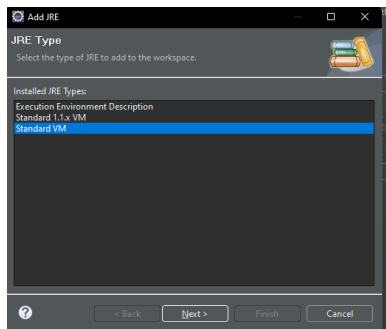
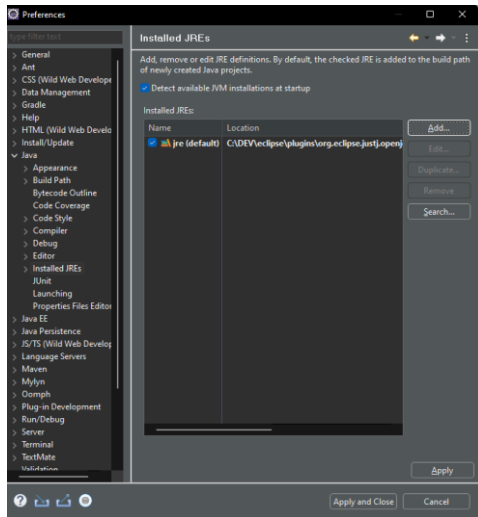
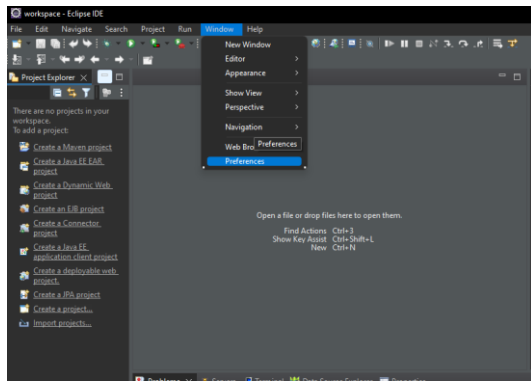
Copiar o arquivo lombok.jar para dentro do eclipse e executar

```
C:\Users\m.salustiano.silva>cd \
C:\>cd DEV\eclipse
C:\DEV\eclipse>java -jar lombok.jar
```



Criar uma pasta workspace dentro de C:\DEV





C:\DEV\Java\jdk-17.0.9

Java - Fundamentos

- Java é uma linguagem de programação amplamente usada para codificar aplicações Web. Java é uma linguagem multiplataforma, orientada a objetos e centrada em rede que pode ser usada como uma plataforma em si. É uma linguagem de programação rápida, segura e confiável para codificar tudo, desde aplicações móveis e software empresarial até aplicações de big data e tecnologias do servidor.
- Conhecer os tipos primitivos
- Declarar variáveis, considerando os diferentes tipos
- Usar estruturas condicionais ('if', 'else')
- Conhecer os operadores de atribuição e comparação
- Usar estruturas de repetição e laços ('while', 'for')
- Usar funções, passando parâmetros e argumentos
- Manipular métodos
- Manipular arrays e listas
- Obter dados de uma API
- Criar construtores

Tipos de dados primitivos

O trabalho com computadores, desde os mais simples como escrever mensagens na tela, até os mais complexos como resolver equações ou desenhar imagens tridimensionais em animação, consiste essencialmente em manipulação de dados. Os dados representados em um computador podem ser números, caracteres ou simples valores.

A linguagem Java oferece diversos tipos de dados com os quais podemos trabalhar. Este capítulo cobrirá os tipos de dados mais importantes. Na verdade há basicamente duas categorias em que se encaixam os tipos de dados: *tipos primitivos* e *tipos de referências*. Os tipos primitivos correspondem a dados mais simples ou escalares e serão abordados em detalhe no que segue, enquanto os tipos de referências consistem em arrays, classes e interfaces. Estes serão vistos em capítulos subsequêntes.

Eis uma visão geral dos tipos que serão abordados neste capítulo:

Tipo	Descrição
<i>boolean</i>	Pode assumir o valor true ou o valor false
<i>char</i>	Caractere em notação Unicode de 16 bits. Serve para a armazenagem de dados alfanuméricos. Também pode ser usado como um dado inteiro com valores na faixa entre 0 e 65535.
<i>byte</i>	Inteiro de 8 bits em notação de complemento de dois. Pode assumir valores entre $-2^7=-128$ e $2^7-1=127$.
<i>short</i>	Inteiro de 16 bits em notação de complemento de dois. Os valores possíveis cobrem a faixa de $-2^{15}=-32.768$ a $2^{15}-1=32.767$

int	Inteiro de 32 bits em notação de complemento de dois. Pode assumir valores entre $-2^{31}=2.147.483.648$ e $2^{31}-1=2.147.483.647$.
long	Inteiro de 64 bits em notação de complemento de dois. Pode assumir valores entre -2^{63} e $2^{63}-1$.
float	Representa números em notação de ponto flutuante normalizada em precisão simples de 32 bits em conformidade com a norma IEEE 754-1985. O menor valor positivo representável por esse tipo é $1.40239846e-46$ e o maior é $3.40282347e+38$.
double	Representa números em notação de ponto flutuante normalizada em precisão dupla de 64 bits em conformidade com a norma IEEE 754-1985. O menor valor positivo representável é $4.94065645841246544e-324$ e o maior é $1.7976931348623157e+308$.

Ao contrário do que acontece com outras linguagens de programação, as características dos tipos de dados listados acima dependem da plataforma em que o programa deverá ser executado. Dessa forma, os tipos de dados primitivos são realmente únicos e garantem a capacidade de intercâmbio de informações entre diversos tipos de computadores, aliviando o programador da preocupação e da árdua tarefa de converter dados em formatos apropriados para a portagem.

Tipo Boolean

Este é o tipo de dado mais simples encontrado em Java. Uma variável *booleana* pode assumir apenas um entre dois valores: **true** ou **false**. Algumas operações possíveis em Java como **a<=b**, **x>y**, etc têm como resultado um valor booleano, que pode ser armazenado para uso futuro em variáveis booleanas. Estas operações são chamadas *operações lógicas*. As variáveis booleanas são tipicamente empregadas para sinalizar alguma condição ou a ocorrência de algum evento em um programa Java. Por exemplo:

```
boolean fim_do_arquivo = false;
```

é a declaração de uma variável do tipo **boolean**, cujo nome é **fim_do_arquivo**. O valor **false** à direita do sinal "=" indica que a variável recebe esse valor como valor inicial. Sem essa especificação o valor de uma variável é imprevisível, podendo ser qualquer um dos valores possíveis para seu tipo (neste caso **true** ou **false**).

Aproveitando o ensejo, há nessa linha a essência da declaração de qualquer variável em Java:

1. Informar o tipo de dado que deseja declarar (**boolean**)
2. Informar o nome que será usado para batizar a variável (**fim_do_arquivo**)
3. Atribuir à variável um valor inicial (**= false**)
4. Terminar a declaração com um ponto-e-vírgula ";".

Uma palavra sobre identificadores

Na declaração acima usamos o nome **fim_do_arquivo** para designar a variável. Um nome de variável, assim como nome de um método, classe, rótulo e dezenas de outros itens lexicográficos, constitui o que é chamado um *identificador*. Uma vez criado, um identificador representa sempre o mesmo objeto a ele associado, em qualquer contexto em que seja empregado.

As seguintes regras regem a criação de identificadores:

- O primeiro caráter de um identificador deve ser uma letra. Os demais caracteres podem ser quaisquer sequências de numerais e letras
- Não apenas os numerais e letras latinas podem ser empregadas, como também letras de quaisquer outro alfabeto
- Devido a razões históricas, o underscore "_" e o sinal de dólar "\$" são considerados letras e podem ser usados nos identificadores
- Assim como em outras linguagens, como C e C++, os identificadores distinguem o tipo de caixa das letras, isto é, as maiúsculas são consideradas distintas das minúsculas. Isso significa que **fim_de_arquivo** é um identificador diferente de **Fim_De_Arquivo**
- Os identificadores não podem ser palavras chave, como: **class**, **for**, **while**, **public**, etc

Tipos de dados inteiros

Os tipos de dados primitivos **byte**, **int**, **char**, **short** e **long** constituem tipos de dados inteiros. Isso porque variáveis desses tipos podem conter um valor numérico inteiro dentro da faixa estabelecida para cada tipo individual. Por exemplo, um **byte** pode conter um inteiro entre -128 e 127, enquanto um **short** pode conter um valor entre -32.768 e 32.767. Já o tipo **long** é suficiente para contar todos os mosquitos do Pantanal Matogrossense.

Há diversas razões para se utilizar um ou outro dos tipos inteiros em uma aplicação. Em geral, não é sensato declarar todas as variáveis inteiras do programa como **long**. Raramente os programas necessitam trabalhar com dados inteiros que permitam fazer uso da máxima capacidade de armazenagem de um **long**. Além disso, variáveis grandes consomem mais memória do que variáveis menores, como **short**.

Obs: Se alguma operação aritmética cria um resultado que excede um dos limites estabelecidos para o tipo inteiro empregado, não há qualquer indicação de erro para avisar sobre essa ocorrência. Ao invés disso, um complemento de dois do valor obtido será o resultado. Por exemplo, se a variável for do tipo **byte**, ocorrem os seguintes resultados: **127+1 = -128**, **127+9=-120** e **127+127=-2**.

Entretanto, uma exceção do tipo **ArithmeticException** é levantada caso ocorra uma divisão por zero. Vejamos o seguinte código:

```
public class Arith
{
    public static void main(String args[])
    {
        byte a = 127;
        short b = 32767;
        int c = 2147483647;
        long d = 9223372036854775807L;
        int e = 0;
        a += 1;
        b += 1;
        c += 1;
        d += 1;
        System.out.println("Valor de a = " + a);
        System.out.println("Valor de b = " + b);
        System.out.println("Valor de c = " + c);
        System.out.println("Valor de d = " + d);
        d /= e;    // Vai dar erro porque e = 0
    }
}
```

com seu respectivo resultado de execução:

```
java Arith
Valor de a = -128
Valor de b = -32768
Valor de c = -2147483648
Valor de d = -9223372036854775808
java.lang.ArithmeticException: / by zero
    at Arith.main(Arith.java:18)
```

Seguem abaixo alguns exemplos de declarações de variáveis de tipo inteiro:

```
byte Contador = 1;
int AnguloEmGraus = -45;
char Indice = 6;
```

A diferença entre essas declarações e a declaração de dados booleanos vista acima está no tipo de dado especificado e no valor atribuído a cada variável.

Operações com inteiros

Podemos realizar uma série de operações com os dados do tipo inteiro. A tabela seguinte mostra uma lista completa.

Operação	Descrição
=, +=, -=, *=, /=, %=	Operadores de atribuição
==, !=	Operadores de igualdade e diferença
<, <=, >, >=	Operadores de desigualdade
+, -	Operadores unários
+, -, *, /, %	Adição, subtração, multiplicação, divisão e módulo
+=, -=, *=, /=, %=	Operadores de atribuição com adição, subtração, multiplicação, divisão e módulo
++, --	Incremento e decremento
<<, >>, >>>	Operadores de deslocamento de bits
<<=, >>=, >>>=	Operadores de atribuição com deslocamento de bits
~	Operador lógico de negação
&, , ^	Operadores lógicos E, OU e OU-exclusivo
&=, =, ^=	Operadores de atribuição com operação lógica E, OU e OU-exclusivo

Muitos das operações que aparecem na lista acima são familiares e praticamente não requerem explicação. Há outros, porém, que pode ser um tanto quanto ambíguos. É o caso dos operadores de atribuição aritméticos. Estes consistem de atalhos para atribuir um novo valor a uma variável onde esse novo valor depende do valor anterior lá armazenado. Por exemplo: += adiciona um valor ao valor antigo de uma variável e a soma passa a ser o novo valor. Esse padrão também é obedecido para as operações -=, *=, /= e %=. Temos assim as seguintes correspondências:

x += 5	x = x + 5
x -= y	x = x - y
x *= 2	x = x * 2
z /= 4	z = z / 4

Os operadores de incremento e decremento referem-se a apenas uma variável (logo são chamados de unários). Por exemplo, o operador de incremento soma um ao operando conforme o exemplo:

```
x++;
```

É uma maneira muito mais concisa de se escrever **x = x + 1**. Mas não só, esses operadores se comportam de modo diferente quando seguem ou precedem o nome de uma variável. Se o operador precede o nome da variável, então o incremento (ou decremento) ocorre antes que o valor da variável seja tomado para a expressão aritmética. No seguinte exemplo, o valor das variáveis x e y será 6:

```
int x = 5;  
int y = ++x;
```

Porém, no próximo exemplo o valor de x será 6 enquanto que o valor de y será 5:

```
int x = 5;  
int y = x++;
```

Vejamos alguns exemplos de declarações e de utilizações de operações envolvendo tipos inteiros:

```
byte j = 60;  
short k = 24;  
int l = 30;  
long m = 12L;  
long resultado = 0L;  
  
resultado += j;           // resultado = 60 (0 mais 60)  
resultado += k;           // resultado = 84 (60 mais 24)  
resultado /= m;           // resultado = 7 (84 dividido por 12)  
resultado -= l;           // resultado = -23 (7 menos 30)  
resultado = -resultado;   // resultado = 23 ( -(-23) )  
resultado %= m;           // resultado = 11 (resto de 23 div. 12)
```

Tipo caracter

Uma variável do tipo **char** armazena um caractere Unicode. Um caractere Unicode é um caractere de 16 bits, sendo que de 0 a 255 correspondem aos caracteres do código ASCII (a tabela ASCII é uma tabela padronizada internacionalmente de associações entre caractere e a sua representação numérica no computador). Uma constante do tipo caractere é representada colocando-se entre apóstrofes, ou pelo valor numérico correspondente na tabela Unicode (ou ASCII), ou ainda, pela sequência '\x' onde x especifica o caractere a ser referido. Esta especificação de sequência de escape obedece às mesmas convenções do C/C++. Por exemplo: 'a', 'f', '\n', etc, são constantes do tipo **char**.

Tipos de ponto flutuante

Em Java, existem duas categorias de variáveis de *ponto flutuante*: **float** armazena valores numéricos em ponto flutuante de precisão simples e **double** de precisão dupla. Ambas seguem norma: *IEEE Standard for Binary Floating Point Arithmetic*, ANSI/IEEE Std. 754-1985 (IEEE, New York). O fato de obedecer a essa norma é que torna os tipos de dados aceitos pela linguagem Java tão portáteis. Esses dados serão aceitos por qualquer plataforma, independentemente do tipo de sistema operacional e do fabricante do computador. A representação dos valores em ponto flutuante pode ser feita usando a notação decimal (exemplo: -24.321) ou a notação científica (exemplo: 2.52E-31).

Além dos possíveis valores numéricos que uma variável de ponto flutuante pode assumir há também os seguintes:

- menos infinito
- mais infinito

- zero
- NAN - not a number

Estes são requeridos pela norma. O mais infinito é usado, por exemplo, ao somarmos 1 ao maior valor possivelmente representável por esse sistema.

Muitas das operações realizáveis com inteiros (porém não todas) têm análogas para números de ponto flutuante. Eis um resumo:

<i>Operação</i>	<i>Descrição</i>
=, +=, -=, *=, /=	Operadores de atribuição
==, !=	Operadores de igualdade e diferença
<, <=, >, >=	Operadores de desigualdade
+, -	Sinais unários
+, -, *, /	Adição, subtração, multiplicação e divisão
+=, -=, *=, /=	Operadores de atribuição com adição, subtração, multiplicação e divisão
++, --	Operadores unários de incremento e decremento

Example

#

Um número hexadecimal é um valor na base 16. Existem 16 dígitos, 0-9 e as letras AF (maiúsculas e minúsculas não importa). AF representa 10-16.

Um número octal é um valor na base 8 e usa os dígitos de 0 a 7.

Um número binário é um valor na base 2 e usa os dígitos 0 e 1.

Todos esses números resultam no mesmo valor, 110:

```
int dec = 110;           // no prefix --> decimal literal
int bin = 0b1101110;     // '0b' prefix --> binary literal
int oct = 0156;          // '0' prefix --> octal literal
int hex = 0x6E;          // '0x' prefix --> hexadecimal literal
```

O literal octal pode facilmente ser uma armadilha para erros semânticos. Se você definir um '0' inicial para seus literais decimais, obterá o valor errado:

```
int a = 0100;            // Instead of 100, a == 64
```

Estruturas condicionais

As estruturas condicionais possibilitam ao programa tomar decisões e alterar o seu fluxo de execução. Isso possibilita ao desenvolvedor o poder de controlar quais são as tarefas e trechos de código executados de acordo com diferentes situações, como os valores de variáveis.

As estruturas condicionais geralmente analisam expressões booleanas e, caso estas expressões sejam verdadeiras, um trecho do código é executado. No caso contrário, outro trecho do código é executado.

If/else

O if/else é uma estrutura de condição em que uma expressão booleana é analisada. Quando a condição que estiver dentro do if for verdadeira, ela é executada. Já o else é utilizado para definir o que é executado quando a condição analisada pelo if for falsa. Caso o if seja verdadeiro e, consequentemente executado, o else não é executado.

O if pode ser utilizado em conjunto com o else ou até mesmo sozinho, caso necessário.

Abaixo, temos um exemplo onde o if é utilizado em conjunto com o else.

```
package com.accenture;

public class Exemplo {

    public static void main(String[] args) {
        int resposta = 10;
        if (resposta == 10) {
            // Se a variável for igual a 10, a frase abaixo será escrita
            System.out.println("Você acertou!");
        } else {
            // Caso contrário, a frase abaixo será escrita
            System.out.println("Você errou!");
        }
    }
}
```

Também podemos utilizar somente o if, não definindo um fluxo alternativo.

```
package com.accenture;
```

```

public class Exemplo {

    public static void main(String[] args) {
        int resposta = 10;
        if (resposta == 10) {
            // Se a variável for igual a 10, a frase abaixo será escrita
            System.out.println("Você acertou!");
        }
        // Se a variável não for igual a 10, nenhuma frase será exibida
    }

}

```

Ainda é possível encadear múltiplas estruturas if/else caso necessário.

```

package com.accenture;

public class Exemplo {

    public static void main(String[] args) {
        int resposta = 10;
        if (resposta == 10) {
            System.out.println("A resposta é exatamente 10!");
        } else if (resposta > 10) {
            System.out.println("A resposta é maior que 10!");
        } else {
            System.out.println("A resposta é menor que 10!");
        }
    }

}

```

Switch/case

A estrutura condicional switch/case vem como alternativa em momentos em que temos que utilizar múltiplos ifs no código. Múltiplos if/else encadeados tendem a tornar o código muito extenso, pouco legível e com baixo índice de manutenção.

O switch/case testa o valor contido em uma variável, realizando uma comparação com cada uma das opções. Cada uma dessas possíveis opções é delimitada pela instrução case.

Podemos ter quantos casos de análise forem necessários e, quando um dos valores corresponder ao da variável, o código do case correspondente será executado. Caso a variável não corresponda a nenhum dos casos testados, o último bloco será executado, chamado de default (padrão).

A análise de cada caso também precisa ter seu final delimitado. Essa delimitação é feita através da palavra break.

```
package com.accenture;

public class Exemplo {

    public static void main(String[] args) {
        int mes = 2;
        switch (mes) {
            case 1:
                System.out.println("O mês é janeiro");
                break;
            case 2:
                System.out.println("O mês é fevereiro");
                break;
            case 3:
                System.out.println("O mês é março");
                break;
            case 4:
                System.out.println("O mês é abril");
                break;
            case 5:
                System.out.println("O mês é maio");
                break;
            case 6:
                System.out.println("O mês é junho");
                break;
            case 7:
                System.out.println("O mês é julho");
                break;
            case 8:
                System.out.println("O mês é agosto");
                break;
            case 9:
                System.out.println("O mês é setembro");
                break;
            case 10:
                System.out.println("O mês é outubro");
                break;
            case 11:
                System.out.println("O mês é novembro");
                break;
            case 12:
                System.out.println("O mês é dezembro");
                break;
            default:
                System.out.println("Mês inválido");
                break;
        }
    }
}
```

Estruturas de repetição

Estruturas de repetição, também conhecidas como loops (laços), são utilizadas para executar repetidamente uma instrução ou bloco de instrução enquanto determinada condição estiver sendo satisfeita.

As principais estruturas de repetição na maioria das linguagens são o for e o while.

For

O for é uma estrutura de repetição na qual seu ciclo será executado por um tempo ou condição pré-determinados e em uma quantidade de vezes que determinamos.

O for possui a seguinte estrutura:

```
for (<variável de controle>, <análise da variável de controle>, <incremento da variável de controle>) {  
    // Código a ser executado  
}
```

Quando utilizamos o for, precisamos de uma variável para auxiliar a controlar a quantidade de repetições a serem executadas. Essa variável é chamada de variável de controle e é declarada no primeiro argumento do for.

O segundo argumento do for é utilizado para definir até quando o for será executado. Geralmente, trata-se de uma condição booleana em cima da variável de controle.

O terceiro argumento indica o quanto a variável de controle será modificada no final de cada execução dentro do for.

Veja o exemplo abaixo:

```
package com.accenture;

public class Exemplo {

    public static void main(String[] args) {
        for (int i = 0; i <= 10; i++) {
            System.out.println("A variável i agora vale " + i);
        }
    }
}
```

A execução desse código causaria a seguinte saída:

A variável i agora vale 0 A variável i agora vale 1 A variável i agora vale 2 A variável i agora vale 3 A variável i agora vale 4 A variável i agora vale 5 A variável i agora vale 6 A variável i agora vale 7 A variável i agora vale 8 A variável i agora vale 9 A variável i agora vale 10

Isso acontece porque:

- A variável de controle, que chamamos de “i”, tem seu valor inicial como 0;
- No segundo bloco, onde escrevemos “i <= 10”, estamos dizendo que o conteúdo do for será executado enquanto o valor de i for menor ou igual a 10;
- Com o terceiro bloco definido como “i++”, estamos dizendo que, no fim de cada execução do for, o conteúdo de “i” será incrementado em 1 unidade. Isso quer dizer que, no fim da primeira execução, i irá de 0 para 1; na segunda execução, irá de 1 para 2, e assim por diante;
- Com isso, o conteúdo do for será executado por 11 vezes, já que o i é iniciado em 0. A saída do código acima mostra que a mensagem foi escrita por 11 vezes, onde o “i” variou de 0 até 10.

Um ponto importante sobre o for é que, por causa da utilização da variável de controle, geralmente ele é utilizado quando sabemos exatamente quantas vezes queremos repetir a execução do trecho de código.

While

O while também é uma estrutura de repetição, assim como o for. A diferença entre ambas é que, enquanto usamos o for quando geralmente conhecemos a quantidade de vezes que o trecho de código deverá ser repetido, nós utilizamos o while quando não sabemos exatamente quantas vezes o código será repetido.

O while possui a seguinte estrutura:

```
while (<condição>) {  
    // Trecho de código a ser repetido  
}
```

Perceba que a condição para interrupção da repetição do trecho dentro do while se dá através de uma condição booleana.

Abaixo, temos um exemplo do while. Neste exemplo, é pedido ao usuário que tente adivinhar o número. Enquanto o usuário não acerta o número, é pedido para que o usuário digite o valor que ele acha que é o correto. Perceba que nós temos um trecho de código que é sempre repetido, que é o pedido do número para o usuário. Mas, não sabemos exatamente quando o usuário vai acertar este número, ou seja, não sabemos exatamente quantas vezes o trecho de código será repetido. Nessa situação, o while é a estrutura de repetição mais adequada.

```
package com.accenture;  
  
import java.util.Scanner;  
  
public class Exemplo {  
  
    public static void main(String[] args) {  
        Scanner in = new Scanner(System.in);  
        int numero = -1;  
        while (numero != 10) {  
// enquanto a variável não for 10, o trecho de código será repetido  
            System.out.println("Digite um número: ");  
            numero = in.nextInt();  
            if (numero == 10) {  
                System.out.println("Você acertou!");  
            } else {  
                System.out.println("Você errou :(");  
            }  
        }  
    }  
}
```

```
}  
}  
}
```

O que é um array em Java?

Quando precisamos armazenar mais de um valor em uma variável, fazemos isso utilizando um array. Essa palavra pode ser traduzida de forma livre como um “conjunto de variedades” — como qualquer conjunto, implica na existência de alguns elementos agrupados, portanto.

Nos arrays, **cada elemento desse grupo tem a sua posição**. Dessa forma, para armazenar esse conjunto de valores em uma única variável, sem precisar declarar cada uma para um valor, usamos um array.

Qual a sintaxe do Java array?

Em Java, quando utilizamos um array, escrevemos sua representação com o uso de chaves — “[elemento1, elemento2 ...]”. Podemos definir uma array sem número especificado, apenas indicando que aquela variável é um array. A sintaxe é a seguinte:

tipo do objeto, como String, Integer, etc. [] nome de nosso array

Para facilitar o entendimento, veja um exemplo aplicado:

```
String[] nomes;
```

No exemplo acima, apenas definimos um array — sem determinar elementos ou número de posições para serem ocupadas. Poderemos preencher esses espaços no array apenas os designando durante o desenvolvimento do código. Assim:

```
String[] nomes = {"Darci", "Cris", "Alex", "Juraci"}
```

Quando precisarmos acessar alguns desses elementos pertencentes às mesmas variáveis, buscamos diretamente a variável do array “nome”. Vamos a um exemplo prático, bem simples, para ajudar no entendimento:

```
public class Main {  
    public static void main(String[] args) {  
        String[] carros = {"Fusca", "Brasilia", "Gurgel"};  
        System.out.println(carros[0]);  
    }  
}
```

O primeiro elemento de um array começa sempre a ser contado em 0. Nesse caso, quando pedimos a posição zero (“[0]”) do array “carros”, a saída no console desse código será:

```
Fusca
```

Quais os tipos de arrays em Java?

Durante o desenvolvimento de aplicações, muitas vezes, não temos a possibilidade de alocar todos os elementos de que precisamos para o funcionamento do sistema durante a programação. Nesses casos, podemos fazer um array vazio, mas com “espaços” para alocar cada elemento.

Sendo um objeto, um array em Java, **não pode apenas ser declarado** — para definirmos o tamanho de nosso array sem a necessidade de criar os elementos, precisamos definir o número de posições, dessa forma:

```
String[] nomes = [4]
```

Já para criar um novo array, por sua vez, usamos a expressão “new”, um comum operador do Java:

```
int[] novo_array = new int[100];
```

Aqui, temos um array simples, o qual tem 100 posições possíveis de serem ocupadas. Isso é chamado “Index” — o índice do array existe, basicamente, para que seja possível enumerar e classificar cada um dos valores contidos nele.

Mas antes de criarmos nosso array, para que possamos manipulá-lo, precisamos saber alguns detalhes importantes. Por exemplo, **um array pode conter apenas uma linha de elemento, enquanto outros serão preenchidos da mesma forma que matrizes**. Essa é a diferença entre arrays unidimensionais e multidimensionais. Confira, a seguir, alguns detalhes.

Arrays unidimensionais

Arrays de uma única dimensão **são vetores que têm um único índice como identificador para cada elemento contido em um array**. No caso do exemplo de que falamos logo acima, a forma que usamos para declarar um array sem determinar seus elementos, mas apenas as posições que serão ocupadas, será da seguinte maneira:

```
public class MyClass {  
    public static void main(String args[]) {  
        int[] array_de_uma_dimensão = new int[5];  
    }  
}
```

Repare que, nesse exemplo, criamos um array com o nome de “array_de_uma_dimensão”, e percebemos que ele terá 5 posições possíveis de serem ocupadas. **Mas será que podemos criar objetos como tabelas, com mais de uma linha de índices?** É sobre isso que falaremos, a seguir.

Arrays multidimensionais

Saiba que também podemos operar com arrays de mais de uma dimensão. **É possível compará-los a tabelas**. Isso porque você pode enumerar múltiplos índices em um mesmo array, formando, assim, diversas linhas e colunas, com cada espaço sendo pertencente a um elemento. Para declararmos um array multidimensional, fazemos dessa forma:

```
public class MyClass {  
    public static void main(String args[]) {  
        int[][] array_multidimensão = new int[5][5];  
    }  
}
```

Aqui, temos a aplicação de um novo array, mas com duas dimensões — ou dois índices. Eles acabam formando uma tabela 5 X 5, com 25 posições possíveis, nesse caso.

Quais as características dos arrays em Java?

Para um bom funcionamento e maior dinâmica durante as operações realizadas por meio de nossos sistemas em Java, temos quase uma **inumerável série de características que permeiam os recursos para esse devido aproveitamento**. Para facilitar o seu entendimento dessas questões, trazemos, em nosso guia, algumas das principais delas, as quais mais interferem ao desenvolvermos serviços com essas estruturas de dados. Confira!

Elementos do array e tipos de dados

É cada unidade do tipo de dado inserido, podendo ser números (integer), caracteres (string), decimais (double) ou verdadeiro/falso (boolean). **Cada elemento ocupa uma posição determinada no índice** e pode ser acessado a partir da manipulação do array.

Índices do array

O índice do array pode ser explicado como uma lista, ordenada de alguma maneira ou não, dos elementos contidos em um dado array. **Os arrays podem ter apenas um ou vários índices.**

Dimensão do array

A dimensão de um array é relacionada à sua quantidade de índices contidos. Se um array tem um único índice, ele é chamado unidimensional, e caso tenha mais de um, multidimensional.

Como declarar arrays em Java?

De maneira geral, definir arrays em Java é a mesma coisa que realizar a declaração de variáveis. Podemos declarar um **array de uma única dimensão** — ou vetor, como também é chamado — das seguintes formas:

```
tipo do dado nomeDoArray [opcional: quantidade de elementos];  
tipo do dado nomeDoArray {elementos separados por vírgula};  
tipo do dado [opcional: quantidade de elementos] nomeDoArray;
```

Veja um exemplo que envolve a aplicação de várias formas de um array — repare nos comentários feitos ao longo do código:

```
public class Main {
    public static void main(String[] args) {
        int[] a = new int[4];
        int[] b;
        b = new int[10];

        // Declarando mais de um array por vez:
        int[] r = new int[44], k = new int[23];

        //Inicializando valores do array:
        int[] inicia_Valores = {2,52,504,7,88,9,6,14,666};

        //Declarando array tipo interger (números inteiros:)
        int[] array_exemplo_N_inteiro;

        //Definindo 9 posições para nossos números e inicializando cada elemento:
        array_exemplo_N_inteiro = new int[9];
        array_exemplo_N_inteiro [0] = 100;
        array_exemplo_N_inteiro [1] = 85;
        array_exemplo_N_inteiro [2] = 88;
        array_exemplo_N_inteiro [3] = 93;
        array_exemplo_N_inteiro [4] = 123;
        array_exemplo_N_inteiro [5] = 952;
        array_exemplo_N_inteiro [6] = 344;
        array_exemplo_N_inteiro [7] = 233;
        array_exemplo_N_inteiro [8] = 622;

        //imprimir no console as posições 8 e 4:
        System.out.println(array_exemplo_N_inteiro[8]);
        System.out.println(array_exemplo_N_inteiro[4]);
    }
}
```

Como descobrir o tamanho de um array?

Como já era de se esperar, há diversas maneiras de se contabilizar os elementos contidos em um array. A mais óbvia, para aquelas pessoas já familiarizadas ao mundo da programação, é a **criação de uma variável, e usar um laço de repetição para modificá-la a cada elemento**. Ficaria dessa maneira:

```
public class Main {
    public static void main(String[] args) {
        int contador = 0;
        String[] carros = {"Fusca", "Kombi", "Gol", "Ferrari"};
        for (String i : carros) {
            contador = ++contador;
        }

        System.out.println(contador);
    }
}
```

Veja que, nesse exemplo, temos como resultado a impressão do número 4 no console, indicando a presença de 4 elementos. Perceba que a contagem se deu a partir do laço for utilizado na variável “contador”.

Outra maneira possível e bastante optada pelas pessoas desenvolvedoras é usar o método `length()`. Com ele, acessamos diretamente a informação referente à quantidade de posições ocupadas em um array. Trazemos um exemplo para facilitar seu entendimento:

```
public class Main {
    public static void main(String[] args) {
        String[] carros = {"Fusca", "Kombi", "Gol", "Ferrari"};
        System.out.println(carros.length);
    }
}
```

A saída será a mesma da anterior:

```
4
```

Uma última maneira é o método `size()`. Com ele, acessamos diretamente o tamanho do número de posições atualmente ocupadas em um array. Trazemos um exemplo prático para que você possa entender melhor:

```
import java.util.*;

public class Main {
    public static void main(String[] arg)
    {
        List<Integer> lista = new array<Integer>();

        lista.add(1);
        lista.add(2);
        lista.add(3);
        lista.add(4);
        lista.add(5);

        System.out.println("Todos elementos " + lista);

        int tamanho = lista.size();

        System.out.println("Tamanho total dos elementos: = " + tamanho);
    }
}
```

O console exibirá a seguinte saída para o código:

```
Todos elementos: [1, 2, 3, 4, 5]
Tamanho total dos elementos: = 5
```


O método `size()` foi utilizado na declaração da variável tamanho — dessa forma, é possível **percebermos que o método já tem a operação de contagem desses elementos em seus processos internos**, não sendo necessário contarmos cada um deles.

Como inicializar um array?

A melhor forma de inicializar um array, de acordo com as melhores práticas, é utilizar o **operador new**, como já apresentamos:

```
int[] novo_array = new int[100];
```

Precisamos detalhar, primeiramente, o tipo de dado (inteiros, caracteres, booleanos ou decimais), seguido dos colchetes (“[]”). Após isso, damos um nome para nosso array e usamos o sinal de igual para **associá-lo a um novo conjunto de elementos**, de determinado tipo de dado.

Precisamos dar a quantidade de posições disponíveis nesse conjunto entre os colchetes, que, no caso do exemplo acima, seria o valor de 100. Há outras formas de inicializar o array, e já descrevemos as principais delas, na seção **“Qual a sintaxe do Java array”** — se você perdeu essa parte, vale a pena voltar e conferir!

Como percorrer os elementos de um Array?

Na maioria das vezes, usamos o laço `for` aprimorado — “enhanced-for loop”, ou também conhecido por `forEach` Loop — para **percorrer cada elemento de nosso array**. No caso que apresentaremos, percorremos cada elemento de um array e, como já fizemos anteriormente, usaremos uma variável para armazenar o valor que precisamos.

Confira o exemplo, na sequência:

```
public class Somando_os_elementos {
    public static void main(String[] args) {
        int[] array = {44, 33, 55, 66, 77, 88, 99, 0, 1};
        int total = 0;
        //Percorrendo cada um dos elementos com forEach e armazenando e somando-os na variável
        "total":
        for(int i : array)
            total += i;
        System.out.printf("Total: %d\n", total);
    }
}
```

Como alterar elementos de um array?

Podemos realizar a alteração de nossos elementos em um array de diversas formas. Uma delas é **a partir do índice, buscando alterar o valor de dada posição ocupada em nosso array.**

A partir do uso de um laço for, percorrendo cada posição em nosso array, usamos uma variável para armazenar temporariamente os elementos para que possamos manipulá-los, seja para modificá-los no valor, seja na posição. Na seção de exemplos que trazemos, o primeiro deles é a aplicação desse método. Você pode ir até lá para conferir e depois voltar aqui.

Outra maneira, amplamente utilizada, é a implementação do método `skip()` — a partir do Java na versão 8. O método `skip` descarta elementos e retorna aqueles restantes:

```
import java.util.*;
public class Main {
    public static void main(String[] args) {
        int [] array = { 0, 1, 2, 3, 4, 5, 6, 7 ,8 };
        Arrays.stream(array).skip(1).forEach(System.out::println);
    }
}
```

No exemplo acima, além de usarmos um **método `forEach`** para iterar cada elemento, adotamos o **método `skip()`** para retirar o elemento que ocupa a posição 1, no caso, o número 1. O console nos mostrará a seguinte saída:

```
2
3
4
5
6
7
8
```

Podemos, também, mudar nossos arrays bem facilmente com o uso da função `Collections.rotate()`. **Ela realizará a rotação da ordem de um array** — definiremos quantas posições serão puladas, veja o exemplo:

```
import java.util.*;
public class Main {
    public static void main(String args[]){
        String [] strArray = {"Primeiro","Segundo","Terceiro","Último"};
        System.out.println("Lista normal : " + Arrays.toString(strArray));

        List<String> list = Arrays.asList(strArray);
```

```
Collections.rotate(list, 3);

System.out.println("Lista manipulada " + list);

}
}
```

A saída no console será a seguinte:

```
Lista normal: [Primeiro, Segundo, Terceiro, Último]
Lista manipulada: [Segundo, Terceiro, Último, Primeiro]
```

Saiba que existe outra maneira, e essa é bem mais simples. Basta definirmos um novo valor para determinada posição, assim:

```
String[] nomes = {"Juci", "Darci", "Alex", "Frye"};
nomes[0] = "Benedita";
System.out.println(nomes[0]);
```

Saída no console:

```
Benedita
```

Aqui, a saída não será a posição zero, que no caso, seria “Juci”, mas sim, “Benedita” — a posição zero, representada por “[0]” foi, simplesmente, substituída.

Confira 3 exemplos de aplicação para o Java array

Vamos à prática para que você entenda bem do que estamos falando.

Exemplo 1: Mudando a ordem das estruturas

Aqui, usamos o método `ShiftToRight()` para trocar os elementos de lugar em relação à posição 4 do array, em que a quarta posição acaba sendo a primeira—lembrando, mais uma vez, **que iniciamos a contagem dos elementos em um array a partir de 0**:

```
import java.util.Arrays;

public class Main {
    public static void main(String args[]){
        int [] array_a = {1,2,3,4,5};
        int n = 4;
        System.out.println("Nosso array é: "+Arrays.toString(array_a));
        ShiftToRight(array_a,n);
    }
    public static void ShiftToRight(int a[],int n){
        int temp = a[n];

        for (int i = n; i > 0; i--) {
            a[i] = a[i-1];
        }

        a[0] = temp;

        System.out.println("Mas, usando o ShiftToRitgh, fica: "+Arrays.toString(a));
    }
}
```

O resultado será assim:

```
Nosso array é: [1, 2, 3, 4, 5]
Mas, usando o ShiftToRitgh, fica: [5, 1, 2, 3, 4]
```

Exemplo 2 — usando arrays multidimensionais

A seguir, um exemplo simples do uso de arrays multidimensionais e como podemos manipulá-los:

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

        int[][] arrayn1 = { { 9,8,7 }, { 6, 5, 4 } };
        int[][] arrayn2 = { { 0, 1 }, { 5 }, { 3, 4, 9 } };

        //imprime o array 2 em cada linha
        System.out.println("Array 1:");
        outputArray( arrayn1 );

        //imprime o array 2 em cada linha
        System.out.println("Array 2:");
        outputArray( arrayn2 );
    }
    public static void outputArray(int[][] array)
    {
        for(int linha = 0; linha < array.length; linha++)
        {
            for( int coluna = 0; coluna < array[linha].length; coluna++)
                System.out.printf("%d ", array[linha][coluna]);
            System.out.println();
        }
    }
}
```

No exemplo, fazemos a iteração pelos elementos do array para **cada linha e cada coluna que é processada**. Como podemos perceber, o laço for será realizado tanto na linha quanto na coluna. Teremos a seguinte saída no console:

```
Array 1
9 8 7
6 5 4
Array 2:
0 1
5
3 4 9
```

Exemplo 3 — qual o tamanho do meu array?

Aqui, veremos como contar cada elemento de nosso array. Podemos fazer a maior parte de operações matemáticas para contabilizá-los ou compará-los no tamanho. A seguir, um exemplo bem simples do que é possível ser feito com esses métodos que apresentamos:

```
public class Main {  
    public static void main(String[] args) {  
        int[] array_1 = { 20, 25, 66, 33, 3, 6, 0, 1, 2, 3, 4 };  
        int[] array_2 = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };  
  
        if(array_1.length > 2){  
            System.out.println("Nosso array 1 tem mais de uma posição!");  
            System.out.println("\nNosso array 2 tem o seguinte número de posições:  
"+array_1.length);  
        }else{  
            System.out.println("Tamanho do ArrayDois - Menor que 1!");  
        }  
    }  
}
```

A saída que veremos no console será essa:

```
Nosso array 1 tem mais de uma posição!  
Nosso array 2 tem o seguinte número de posições: 10
```

Em que usamos o array em nosso dia a dia?

É bastante complicado explorarmos o uso de array no dia a dia das pessoas programadoras. Isso se deve ao fato de a aplicação desse recurso estar intimamente ligada a **uma imensa variedade de recursos utilizados por diversas das funções, métodos e classes** disponíveis nos inúmeros pacotes que atentem à linguagem Java.

A cada atualização da linguagem, temos vários desse elementos como novidade. É verdade que veremos muito sua aplicação quando trabalhamos no desenvolvimento de jogos e coisas mais simples — o princípio didático ainda se torna bastante expoente para o array.

Como já temos estruturas e frameworks que lidam mais facilmente com os arrays, optamos por eles na maioria das vezes. Isso porque trarão **maior dinamicidade, confiança e agilidade** para o andamento de nossa aplicação que está sendo desenvolvida.

No decorrer de nosso guia, vimos a importância do Java array no desenvolvimento de aplicações e que as possibilidades de suas funções são quase infinitas. Abordamos aspectos teóricos e trazemos exemplos práticos de uso de arrays para explicá-los. Conferimos suas principais características de funcionamento e implementação, assim como exploramos seus detalhes.

Portanto, por meio deste guia, possibilitamos às pessoas iniciantes da linguagem Java se inteirar de mais um recurso importantíssimo da linguagem, assim como alguns pontos que merecem atenção das boas práticas durante as explicações. Esperamos que, com isso, haja uma maior evolução daqueles que se aplicam em aprender sempre mais sobre tecnologia e sobre o mundo digital.

Por outro lado, também descobrimos que o Java array em seu estado puro não é tão utilizado em aplicações e ferramentas — na maior parte das vezes em que usamos a linguagem nos ambientes profissionais da programação. Mesmo assim, é muito importante entender sobre a base em que as muitas funções, métodos e classes opera — utilizada amplamente em ambientes mais complexos, com imensa importância para as melhores pessoas desenvolvedoras.

Arraylist Java: usando listas em Java!

Muitas vezes, durante o desenvolvimento de aplicações, usamos o **ArrayList Java para agrupar todos os dados que queremos em estruturas** para posterior utilização em nosso programa. O ArrayList é uma das **principais maneiras que manipulamos listas**, dicionários, filas, pilhas ou qualquer outro tipo de coleção.

Às pessoas iniciantes na linguagem — ou na programação — pode ser um tanto confuso, ainda mais com outras nomenclaturas parecidas ou funções ainda desconhecidas que permeiam a utilização do ArrayList. Na realidade, poucos são os que iniciam no Java e que sabem realmente a importância que essa estrutura tem dentro dos códigos modernos.

Sabendo disso, trazemos, neste guia, um apanhado de explicações e abordagens práticas diferentes sobre o uso do ArrayList para a linguagem Java, para **ajudar quem está iniciando a entender mais sobre essa classe e sua utilização**. Além disso, claro, fazer com que os mais experientes relembrem algumas questões importantes sobre as melhores práticas estabelecidas na linguagem. Abordaremos os seguintes conteúdos:

O que é a Arraylist Java?

O Java ArrayList, é, basicamente, **um array dinâmico que encontramos no java.util** — um pacote que contém uma grande variedade de recursos, como

estruturas de coleções, modelos de data e hora, recursos para internacionalização, entre muitas outras facilidades para o desenvolvimento de aplicações em Java. Esses arrays redimensionáveis **são muito úteis quando utilizados para implementações em que precisamos manipular listas.**

A dinamicidade do recurso possibilita à pessoa desenvolvedora a criação de coleções — arrays, classes e objetos — **sem precisar se preocupar com o redimensionamento dos vetores.** Caso algum haja necessidade de uma posição adicional em um array, o ArrayList realiza a operação de maneira autônoma.

Quais as principais características da classe ArrayList Java?

Entre as interfaces e classes das estruturas disponibilizados para uso durante o desenvolvimento de aplicações em Java, certamente, o ArrayList é uma das principais delas. Sua característica de **construção dinâmica de arrays possibilita manipulá-las com o uso de métodos para adicionar ou retirar objetos.** Sua implementação é realizada usando de um array subjacente a ele.

Em outras palavras, quando adicionamos um elemento em um ArrayList, o que acontece é que a classe aloca um elemento adicional em outro vetor, maior, para posteriormente copiá-lo em um novo. Por ser custoso mover arrays manualmente, **fica muito mais fácil criarmos um ArrayList com um tamanho predefinido, igual ao número que o vetor terá ao estar cheio.**

Normalmente, quem trabalha com desenvolvimento não especifica um tamanho inicial para um ArrayList, apenas em casos específicos — nem sempre saberemos qual será o tamanho máximo de posições em vetor, não é mesmo?

Quais as diferenças entre array e arraylist em java?

Quando utilizamos o array em Java, estamos trabalhando com tamanhos fixos de coleções. Para inserirmos algum elemento naquele grupo de objetos, precisamos criar outro que contemple aquele novo tamanho, com determinado número de posições a mais — isso não acontece com o ArrayList.

É muito comum a confusão de que arrays e ArrayList são a mesma coisa, mas saiba que eles são muito diferentes. O ArrayList não é um array padrão, ele apenas utiliza essa funcionalidade para realizar o armazenamento dos objetos contidos nas listas manipuladas por ele. Nem ao menos os atributos desse array, que o ArrayList utiliza, é possível ser acessado durante o processamento.

Como a **classe ArrayList** é um **agrupamento dinâmico de objetos**, isso quer dizer que podemos adicionar ou retirar elementos de, por exemplo, uma lista, sem que seja necessário criar uma nova, **mantendo a original com um número diferente dos objetos ali contidos**. Desde o momento em que criamos um array, seu tamanho não pode ser mudado:

```
String [] meuStringArray = new String[3];
```

Repare que, nesse caso, independentemente de termos objetos ou não dentro de nosso novo array, sempre teremos para ele três posições disponíveis, nunca mais do que isso. O que é bem diferente para o ArrayList.

Aqui, trazemos um exemplo prático da aplicação:

```
list lista = new ArrayList();  
lista.add("Pessoa 1");  
lista.add("Pessoa 2");  
lista.add("Pessoa 3");
```

Isso porque a **classe ArrayList** é **independente do número de objetos contidos nela**. Podemos aumentar ou diminuir seu tamanho, apenas inserindo ou retirando mais objetos.

Qual a hierarquia da classe Arraylist?

Por serem membros de uma mesma classe, a java.util, as interfaces nas estruturas das coleções Java obedecem a uma hierarquia. Entre essas estruturas presentes nesses pacotes, algumas aparecem como as principais para o conhecimento das pessoas iniciantes nessa linguagem.

Uma coleção pode agregar vários tipos de interface. Podemos destacar conjuntos, listas (list), filas (queue) e mapas (maps). **Cada uma delas pode fazer parte de uma mesma coleção, independentemente da estrutura utilizada.**

A hierarquia serve para representar que cada uma dessas estruturas são de um tipo único. Explicando de outra maneira: **não é possível termos uma lista e, ao mesmo tempo, termos um conjunto na mesma coleção**. Por exemplo, para declararmos uma coleção, usamos a seguinte sintaxe:

```
Collection<String> coleção = new ArrayList();
```

Para adicionarmos valores nessas coleção:

```
import java.util.Collection;
import java.util.ArrayList;

public class MetodoAdd {
    public static void main(String[] args) {
        Collection<String> nomes = new ArrayList();
        nomes.add("João");
        nomes.add("Maria");
        nomes.add("Eduardo");
        nomes.add("Silvana");
        nomes.add("Mário");

        System.out.println("Lista de nomes: "+nomes);
    }
}
```

Explicaremos mais detalhes sobre as coleções e as interfaces de estrutura na sequência, acompanhe.

Conjuntos

Também chamada de Set Interface, **essa interface não armazena dados sobre a contagem dos elementos presentes ou a ordem em que se encontram**. É um arranjo de objetos que pode ser comparado às estruturas de conjunto que aprendemos nas aulas de matemática, no ensino fundamental.

Usamos o método set() para adicionar um elemento a um conjunto. Também é possível usar o método sortedset() para ordená-lo e não termos objetos repetidos nesse modelo de estrutura.

Por exemplo, aqui, usamos uma aplicação da interface Set feita de maneira bem simples para fins didáticos. Suponha que você tenha uma Coleção, que chamaremos de “col”, e queira criar outra, contendo os mesmos elementos, mas com todas as duplicatas eliminadas. O código em uma linha ficará dessa maneira:

```
Collection<Type> semDuplicados = new HashSet<Type>(col);
```

Listas

A interface List é implementada pela classe ArrayList, **fornecendo métodos para que seja possível manipular os objetos baseados na posição em que estão**. Essa sequência de elementos, como também pode ser chamada, tem os índices iniciados em 0 — ou seja, o sequenciamento acontece da mesma maneira dos vetores, e começa sua contagem do 0, em vez do 1.

Nesse exemplo, vemos como podemos trocar dois valores de posição usando a interface list:

```
public static void swap(List a, int i, int j) {
    tmp = a.get(i);
    a.set(i, a.get(j));
    a.set(j, tmp);
}
```

Filas

Na interface de estrutura de dados que chamamos fila (ou queue), **a ordem de chegada de cada elemento é o que mais importa para o processamento da operação**. Conforme os elementos são adicionados, são colocados no fim da fila e aguardam até que sejam acessados.

Um exemplo possível para essa interface seria a criação de um temporizador — usando a função da fila para armazenar valores e imprimi-los na saída a cada segundo, podemos criar um pequeno programa de timer de um minuto:

```
int relógio = Integer.parseInt(args[0]);

Queue<Integer> queue = new LinkedList<Integer>();{
    for (int i = time; i >= 0; i--)
        queue.add(i);
    while (!queue.isEmpty()) {
        System.out.println(queue.remove());
        Thread.sleep(60);
    }
}
```

Mapas

Map é um **objeto que opera armazenando chaves e valores**. As chaves nunca poderão ser repetidas e sempre devem ter algum valor. Seu funcionamento pode ser comparado à abstração das funções em matemática. Essa interface utiliza métodos comuns para operar, como set(), add(), remove() etc.

Um bom exemplo de uso para a interface map seria a ordenação de cada pessoa em determinado grupo em relação à cidade em que mora:

```
Map<String, List<Pessoa>> pessoasPorCidade
    = pessoaColecao.collect(Collectors.groupingBy(Pessoa::getCidade));
```

Quais os principais métodos usados com a classe ArrayList?

Seja para adicionar algum objeto, seja para remover ou apagar todo um vetor usando a classe ArrayList, precisamos usar alguns métodos. Para facilitar o seu entendimento, listamos alguns deles com uma breve explicação de cada um. Acompanhe-os para compreender os exemplos com maior profundidade:

- **new ArrayList():** cria um novo ArrayList. Por padrão, essa classe tem a capacidade inicial de 10 elementos;
- **add(item):** é o método utilizado para adicionar novos elementos ao ArrayList. Os elementos são colocados no fim da lista, por padrão;
- **remove(posição):** remove um item de determinada posição na lista criada;
- **set(item, lista):** usamos para definir um elemento em determinado index;
- **get(item):** retorna o objeto ligado a determinado índice;
- **iterator():** responsável por iterar um elemento na sequência adequada do vetor;
- **clear():** limpa todos os elementos contidos na lista.

Confira 4 exemplos práticos de uso dos métodos!

Para entender mais sobre o assunto, vale a pena ver como funciona, na prática.

1. Criando novo ArrayList com o método new ArrayList()

```
import java.util.ArrayList;

public class Main {
    public static void main(String[] args) {
        ArrayList<String> carros = new ArrayList<String>();
        System.out.println(carros);
    }
}
```

2. Adicionando elementos como método add()

```
import java.util.ArrayList;

public class Main {
    public static void main(String[] args) {
        ArrayList<String> carros = new ArrayList<String>();
        carros.add("Fusca");
        carros.add("Brasília");
        carros.add("Chevette");
        carros.add("Monza");
        carros.add("Monza");
        System.out.println(carros);
    }
}
```

3. Apagando todo o ArrayList com o método clear()

```
import java.util.ArrayList;

public class Main {
    public static void main(String[] args) {
        ArrayList<String> carros = new ArrayList<String>();
        carros.add("Fusca");
        carros.add("Kombi");
        carros.add("Brasília");
        carros.add("Ferrari");
        carros.clear();
        System.out.println(carros);
    }
}
```

Não teremos impressão de informação de nenhum carro na tela, repare que a lista é limpa com o método clear(). Teremos a impressão de algo como:

```
[ ]
```

4. Acessando um elemento da lista com o método get()

```
import java.util.ArrayList;

public class Main {
    public static void main(String[] args) {
        ArrayList<String> cars = new ArrayList<String>();
        carros.add("Fusca");
        carros.add("Kombi");
        carros.add("Brasília");
        carros.add("Ferrari");
        System.out.println(carros.get(0));
    }
}
```

Aqui, teremos a impressão do nome do carro que ocupa a posição 0 de nosso ArrayList:

```
Fusca
```

Como adotar, na prática? 3 exemplos de usos da classe ArrayList

A seguir, você acompanha alguns exemplos básicos que trazemos para ilustrar tudo o que exploramos no decorrer de nosso guia. Você pode tentar replicá-los em seu computador para fixar melhor o conteúdo e praticar um pouco a implementação de um ArrayList. Vamos lá?

Exemplo 1 — listando disciplinas oferecidas em uma escola

Aqui, é possível observarmos o uso de um ArrayList com inserção de objetos (nesse caso, disciplinas escolares) com o método add(). Primeiro, criamos uma lista primária com algumas das disciplinas usando o ArrayList.

Em seguida, criamos uma ArrayList e incluímos todas as disciplinas da lista anterior, adicionando duas novas na composição:

```
import java.util.ArrayList;
import java.util.Arrays;

public class TesteA {
    public static void main(String[] args) {
        String[] disciplinas = {"matemática", "filosofia", "história", "física"};
        ArrayList<String> novaLista = new ArrayList<String>(Arrays.asList(disciplinas));
        novaLista.add("geografia");
        novaLista.add("língua inglesa");

        for (String str: novaLista)
        {
            System.out.println(str);
        }
    }
}
```

A saída desse código no terminal deverá ser:

```
matemática
filosofia
história
física
geografia
língua inglesa
```

Exemplo 2 — Declarando uma agenda de contatos e realizando seu instanciamento

```
import java.util.*;

public class Exemplo2 {

    public static void main(String[] args) {
        Scanner ler = new Scanner(System.in);
        ArrayList<String> agenda = new ArrayList();

        agenda.add("Bezerra da Silva;11 1111-1111");
        agenda.add("Paulo Ricardo;22 2222-2222");
        agenda.add("Roberto Carlos;33 3333-3333");
        agenda.add("Moraes Moreira;44 4444-4444");

        // mostrando os contatos a agenda:
        System.out.printf("\nPercorrendo o ArrayList\n");
        i = 0;
        for (String contato: agenda) {
            System.out.printf("Posição %d- %s\n", i, contato);
            i++;
        }

        System.out.printf("\n Iterando o ArrayList\n");
        i = 0;
        Iterator<String> iterator = agenda.iterator();
        while (iterator.hasNext()) {
            System.out.printf("Posição %d- %s\n", i, iterator.next());
            i++;
        }
    }
}
```

Nesse exemplo, realizamos a iteração dos elementos contidos em nosso ArrayList mostrando todos os contatos da agenda. Repare que o método `size()` é utilizado para darmos um tamanho para nossa lista, definindo a quantidade de posições que ela terá.

Exemplo 3 — Ordenando a ArrayList

```
import java.util.*;

public class Main {
    public static void main(String[] args) {
        ArrayList<Integer> numeros = new ArrayList<Integer>();
        numeros.add(903);
        numeros.add(105);
        numeros.add(290);
        numeros.add(904);
        numeros.add(890);
        numeros.add(1902);

        Collections.sort(numeros);

        for (int i : numeros) {
            System.out.println(i);
        }
    }
}
```

Nesse exemplo, o que fazemos é ordenar os elementos de nossa ArrayList com a utilização do método `sort()`. A saída desse código será a seguinte:

```
105
290
890
903
904
1902
```

Java não genérico vs coleção genérica

Os tipos genéricos foram **introduzidos a partir da versão 5 do Java**, permitindo que um método seja capaz de operar sobre vários objetos de diferentes tipos. Em outras palavras, usando uma coleção genérica em Java, **escreveremos uma classe, um método ou uma interface somente uma vez** — e poderemos usá-los em qualquer tipo de dado.

Já se escrevermos de maneira não genérica, precisaríamos reescrever cada uma das vezes que a estrutura é utilizada. Um bom exemplo para explicar essa diferença seria supormos um ArrayList responsável pelo armazenamento de nomes de alunos de uma escola.

Se, por algum engano, o programador utiliza objetos de valores inteiros, em vez de strings, o compilador não vai proibir. Veja esse exemplo:

```
import java.util.*;

class Alunos{
    public static void main(String[] args)
    {
        //Criando um array – sem a especificação de seu tipo:
        ArrayList al = new ArrayList();
        al.add("Bob");
        al.add("Carlo");
        al.add(10);

        String s1 = (String)al.get(0);
        String s2 = (String)al.get(1);

        try {
            //Acusa Runtime Exception:
            String s3 = (String)al.get(2);
        }
        catch (Exception e) {
            System.out.println("Exception: " + e);
        }
    }
}
```


Nesse caso, é um exemplo bem simples que mostra que o não uso de coleções genéricas pode causar “run time exceptions” durante sua execução — uma saída que acusa que o compilador demorou muito tempo para o processamento, de forma resumida. A saída desse código será a seguinte:

```
java.lang.ClassCastException:  
java.lang.Integer não pode ser convertido em java.lang.String
```

Concluindo, a conversão de tipo individual não é necessária com o uso de coleções genéricas. Evitamos **problemas futuros** durante a implementação se o compilador já entender previamente que a nossa lista apenas pode usar dados do tipo string, por exemplo:

```
import java.util.*;  
  
class Test {  
    public static void main(String[] args)  
    {  
        //Especificamos o ArrayList como “string” desta vez:  
        ArrayList<String> al = new ArrayList<String>();  
  
        al.add("Bob");  
        al.add("Carlo");  
  
        //Agora o compilador não deixa que isso ocorra:  
        al.add(10);  
  
        String s1 = al.get(0);  
        String s2 = al.get(1);  
        String s3 = al.get(2);  
    }  
}
```

E teremos o seguinte erro na linha 15:

15: erro: nenhum método adequado encontrado para add (int)

```
al.add (10);
```

Vimos, no decorrer deste guia, as principais aplicações para a classe Java ArrayList e suas características mais importantes quando aplicadas durante o desenvolvimento de sistemas em Java. Você pode entender as diferenças para o Array e a utilização de cada um deles para casos específicos.

Conhecemos a hierarquia estabelecida dentro do **pacote java.util** e as principais **interfaces de estruturas de dados** ali contidas. Exploramos os métodos mais relevantes que podem ser utilizados com **ArrayList** e exemplificamos com aplicações práticas desses métodos.

Deu para perceber a importância da utilização de coleções genéricas e suas aplicações mais recorrentes, certo? Concluindo, passamos pelas principais características e aplicações possíveis para o **Java ArrayList**, com exemplos práticos que contemplem a teoria da maneira mais simples e acessível que encontramos.

Java map: o que é e quais os métodos dessa interface?

Os conjuntos de dados do tipo map são também **conhecidos por “arrays associativos”**. São muito úteis durante o desenvolvimento de aplicações em Java quando precisamos **pesquisar e atualizar elementos correspondentes com base em uma chave específica para determinado valor**. Assim, o Java map “mapeia” os valores contidos no conjunto de dados tomando as chaves por referência, resumidamente.

Para muitas das pessoas iniciantes no mundo do desenvolvimento, pode parecer uma tarefa difícil **estabelecer alguma relação entre esse conceito e o seu uso na linguagem Java**. Se você é uma dessas pessoas, você está no lugar certo. Neste guia, exploraremos os principais detalhes dessa interface e como aplicá-la, usando vários exemplos práticos durante nossas explicações.

Entender bem sobre as principais interfaces da linguagem Java é de extrema importância para **quem quer desenvolver boas aplicações**, obedecendo às boas práticas e evitando muitos problemas desnecessários por falta do conhecimento de alguns detalhes. O Java map está entre essas interfaces, ocupando um espaço que merece a atenção de quem aspira à carreira do desenvolvimento.

Por esse motivo, trazemos este guia até você. Nele, serão tratados diversos assuntos relevantes sobre o tema:

Tudo será explicado de maneira descomplicada, com uma teoria dinâmica percorrendo exemplos práticos e uso dos recursos apresentados. Acompanhe-nos em mais uma jornada pela linguagem Java!

O que é a interface Java map?

A Java map é **uma das interfaces facilitadoras** que estão disponíveis para serem usadas durante o desenvolvimento de aplicações em Java, **por meio do pacote java.util**. Sua função é facilitar a manipulação de uma estrutura de correspondência — ou seja, os conjuntos maps representam, sempre, **uma chave e um valor correspondentes entre si**:

fruta : cor
pera : amarela
maçã : vermelha
melancia : verde

O comportamento dessa interface **difere das outras interfaces do framework Collections** — o tipo map **não possibilita o armazenamento de duas chaves idênticas**. Apesar disso, é possível usar valores iguais entre as chaves, sem nenhum problema.

Diferentemente do que muitas pessoas iniciantes em Java podem pensar, **a interface Map não é uma das subinterfaces das coleções**, como um List ou ArrayList. Na sequência, abordaremos mais sobre a hierarquia dessa estrutura.

Como funciona a hierarquia da interface Java map?

É possível implementarmos os conjuntos de dados de tipo map por meio de duas interfaces: **Map e sortedMap** (em que existe a ordenação dos elementos contidos nas chaves), usando suas três classes para ser implementada — **HashMap(), LinkedHashMap() e TreeMap()**. Cada uma delas carrega um detalhe específico e ocupa um modelo de implementação de um map de modos diferentes:

- **HashMap()**: implementação de Map — é importante levar em consideração que o uso dessa classe não realiza a ordenação dos elementos;
- **LinkedHashMap()**: a mesma coisa que a classe anterior, mas com ordenamento dos elementos contidos no map em ordem de inserção. Suporta o uso de valor e chaves nulos ("null").
- **TreeMap()**: o **treeMap** é implementação das interfaces Map e sortedMap — sempre usa a ordenação ascendente (do menor para o maior). É bastante usado na manipulação de dados hierárquicos e não tem suporte para uso de valores nulos ("null").

Qual a sintaxe da interface Java map?

Como apenas pode haver uma chave específica para cada elemento em um map, é importante obedecermos a essa característica na hora que criarmos uma nova estrutura. Portanto, fazemos da seguinte forma:

```
Map<objeto_chave, objeto_valor> nome_do_mapa = new tipo_de_dado();
```

Repare que precisamos **declarar o tipo de dado de cada objeto**, seja no valor, seja na chave. Pense em uma situação em que temos nomes que **precisamos relacionar** com alguns números, como em uma agenda. Nesse caso, usamos as chaves para armazenar os nomes, e ela precisa ser única para cada posição.

Como são caracteres, definimos como String. Já no **valor**, como são números, **precisamos declará-los com Inteiros** (ou Integer). Confira o exemplo, com atenção especial a como realizamos a declaração do tipo de dados que queremos, pois será importante para a próxima parte do guia:

```
import java.util.HashMap;
import java.util.Map;

public class Main {

    public static void main(String[] args)
    {

        Map<String, Integer> agenda = new HashMap<String, Integer>();
        agenda.put("Luma", 11222);
        agenda.put("Alex", 22333);
        agenda.put("Andrea", 3344);

        System.out.println(agenda);

    }
}
```

Pelo motivo de a **interface map não ser instanciável**, usamos a classe **HashMap()** para manipular o conjunto. A seguir, exploraremos mais dessa classe, com exemplos e detalhes sobre a implementação do Java map.

O que é HashMap()?

Falando de uma maneira bem resumida, o **HashMap é a classe responsável por implementar uma estrutura map** para a interface Map, armazenar chaves únicas (hashes) e as atribuir a valores. Embora possa parecer complicada de início, a sintaxe da interface e das classes usadas nos códigos são bem simples.

Para declararmos **um novo map**, podemos utilizar o **HashMap**. Para essa implementação, usaremos o “Map” + “nome que queremos para ele” e o igualaremos ao operador “new”, seguido do método `HashMap()`. Dessa maneira:

```
Map nomeDoMap = new HashMap();
```

Veja, a seguir, um exemplo prático em um código utilizando essa implementação:

```
import java.util.*;

public class Main {
    public static void main(String args[])
    {
        Map<String, String> meu_Map
            = new HashMap<String, String>();

        meu_Map.put("T", new String("melhor"));
        meu_Map.put("r", new String("escola"));
        meu_Map.put("y", new String("de"));
        meu_Map.put("b", new String("programação"));
        meu_Map.put("e", new String("e tecnologia"));

        for (Map.Entry<String, String> be :
            meu_Map.entrySet()) {

            System.out.print(be.getKey() + ":");
            System.out.println(be.getValue());
        }
    }
}
```

Caso ainda não tenha familiaridade com os métodos `get()`, `set()` e `put()`, não tem nenhum problema. Na sequência, abordaremos os principais detalhes desses recursos.

O que é Hashtable?

São também comumente conhecidas como “**tabelas de dispersão**”. É uma estrutura em forma de tabela, a qual é responsável por implementar um array abstrato e associá-lo a determinado tipo de estrutura de dado — realizando a **correspondência das chaves com os valores contidos em um map**, resumidamente.

A utilização mais comum das hashtables é a realização de buscas em estruturas map de maneira agilizada. É implementada por meio da **função `HashTable()`**, fazendo o uso de diferentes métodos para manipular seus elementos. A sintaxe utilizada na implementação é a mesma para o hashmap:

```
import java.util.*;

public class Hashtable_trybe{
```

```

public static void main(String args[]){
    Hashtable<Integer,String> ht=new Hashtable<Integer,String>();

    ht.put(100,"AAA");
    ht.put(102,"BBB");
    ht.put(101,"CCC");
    ht.put(103,"DDD");

    for(Map.Entry i:ht.entrySet()){
        System.out.println(i.getKey()+" "+i.getValue());
    }
}
}

```

Quais os métodos da interface Java map?

Assim como prometido, trazemos uma lista com os principais métodos utilizados na manipulação de um Map. Confira!

size()

Para usarmos o método size(), seguimos a seguinte sintaxe:

Hash_Map.size().

Ele não recebe parâmetros — o método retorna o tamanho do map, ou seja, o **número dos pares de chaves e valores contidos nele**.

Exemplo de uso:

```

import java.util.*;

public class Main {
    public static void main(String[] args)
    {
        HashMap<Integer, String> H_map = new HashMap<Integer, String>();

        // Mapping string values to int keys
        H_map.put(0, "Venha");
        H_map.put(1, "com");
        H_map.put(2, "a");
        H_map.put(3, "Trybe");

        // Displaying the size of the map
        System.out.println("The size of the map is " + H_map.size());
    }
}

```

isEmpty()

O método boolean isEmpty() realiza uma verificação nas chaves e valores contidos no map para saber se está ou não vazio. Não usa argumentos, e retorna verdadeiro em caso de algum par chave-valor esteja vazio:

nome_map.isEmpty()

Exemplo de uso:

```
import java.util.*;
public class Main {

    public static void main(String[] args)
    {
        Map<String, String> novo_map = new HashMap<>();
        System.out.println(novo_map);
        System.out.println(novo_map.isEmpty());
    }
}
```

containsKey() e containsValue()

Os métodos containsKey() e containsValue() funcionam da mesma forma, mas um apenas para chaves e o outro somente para valores, respectivamente. A sintaxe é:

Hash_Map.contains(Key ou Val)(elemento).

Os parâmetros necessários são a especificação do elemento a ser verificado, chaves ou valores. Retorna um dado booleano: verdadeiro, para os casos em que o elemento verificado esteja presente, e falso, para caso negativo.

Exemplo de uso:

```
import java.util.*;

public class Main {
    public static void main(String[] args)
    {
        HashMap<Integer, String> hash_map = new HashMap<Integer, String>();
        hash_map.put(0, "Estude");
        hash_map.put(1, "na");
        hash_map.put(2, "Trybe!");
        hash_map.put(3, "Be");
        hash_map.put(5, "Trybe");

        System.out.println("Elementos internos do map: " + hash_map);

        System.out.println("Verificar se a chave 0 existe " +
            hash_map.containsKey(0));

        System.out.println("Verificar se a chave 10 existe " +
            hash_map.containsKey(10));

        System.out.println("Verificar se a chave 2 existe: " +
            hash_map.containsValue(Be));
    }
}
```

A saída será:

```
Elementos internos do map: {0=Estude, 1=na, 2=Trybe!, 3=Be, 5=Trybe}
Verificar se a chave 0 existe true
Verificar se a chave 10 existe false
Verificar se a chave 2 existe: true
```

get()

O método `get()` é usado para recuperar o elemento especificado no parâmetro que damos para ele. A sintaxe do `get()` é:

`thisMap.get(objeto, chave_do_elemento)`

O método retorna o valor contido na chave especificada.

Exemplo de uso:

```
import java.util.*;

public class Main {
    public static void main(String[] args)
    {
        Map<Integer, String> map = new HashMap<Integer, String>();

        map.put(0, "Aprenda");
        map.put(1, "com");
        map.put(2, "a");
        map.put(3, "Trybe");

        System.out.println("Elementos internos do map: " + map);

        System.out.println("Verificar a chave 0: " + map.get(0));

        System.out.println("Verificar a chave 5: " + map.get(5));
    }
}
```

put() e remove()

Put() e remove() são métodos responsáveis por colocar ou retirar elementos em uma estrutura map. Sua sintaxe é:

Hash_Map.put_ou_remove(key, val)

Não retorna valores, já que é responsável por inserir elementos.

Exemplo de uso:

```
import java.util.*;

public class HashMap{
    public static void main(String[] args)
    {
        HashMap<Integer, String> hmap = new HashMap<Integer, String>();

        hmap.put(0, "Estude");
        hmap.put(1, "com");
        hmap.put(2, "a");
        hmap.put(3, "Trybe!");

        System.out.println("O Map completo é: " + hmap);
        hmap.remove(2, "a");
        hmap.remove(3, "Trybe!");

        System.out.println("O Map depois do remove é: " + hmap);
        System.out.println("Agora, o map será: " + hmap);
    }
}
```

Fora esses, saiba que ainda existem **alguns outros métodos que podem ser utilizados** para implementar estruturas em map, como:

- **o `forEach()`**, usado para iterar os valores e chaves da interface;
- **o `replace()`**, utilizado para substituir elementos;
- **o `merge()`**, operação funcional usada para mesclar diferentes maps, entre outros métodos.

O importante, agora, é entendermos o funcionamento e as possibilidades dessa estrutura para o desenvolvimento de aplicações em Java.

Operações funcionais na interface Java map!

Quando falamos sobre operações funcionais em Java map, estamos tratando daquelas **funções em que podemos manipular estruturas de tipo map de maneira mais funcional**. Nessas operações são usadas as expressões lambda como parâmetro.

Expressões lambda no Java também podem ser entendidas como se fossem um **objeto, podendo ser executadas sob demanda**. Foi implementada a partir do Java 8.

As principais operações funcionais que podem ser implementadas com a interface map são: **`compute()`, `computeIfAbsent()` e `computeIfPresent()`** — elas usam uma chave e uma expressão lambda como parâmetros. De modo geral, **o método `compute()` funciona no cálculo do mapeamento de uma chave especificada e seu valor**, se já estiver atualmente mapeado (usará o null, caso não exista).

A seguir, algumas aplicações do método `compute()` sendo utilizado em estruturas map como exemplos.

Exemplo 1 — `map.compute`

Mapeia a chave e seu valor atual — um exemplo de `compute()` incrementando um valor inteiro no map:

```
map.compute(key, (k, v) -> (v == null) ? 1 : v+1)
```

Exemplo 2 — map.computePresent

Mapeia chaves e valores, atualizando o valor especificado para a chave designada.

```
map.computePresent("100", (key, val) -> val == null ? null :  
value.toString().toUpperCase());
```

Exemplo 3 — map.computeIfAbsent

Calcula o valor de determinada chave usando dada função de mapeamento. Se a chave ainda não estiver associada a um valor (ou estiver mapeada para null), insere esse valor calculado no próprio Hashmap — ou o atualiza para “null”, caso não exista:

```
map.computeIfAbsent("123", (key) -> "abc");
```

Em todas essas situações, o valor retornado é inserido na chave da estrutura map. A chave é apagada em caso de retorno “null” da função.

Abordamos, durante o nosso guia, uma série de questões bastante relevantes para desenvolvimento de aplicações com o Java map, tanto no entendimento que a compreende como **Interface** quanto como **estrutura de elementos de um conjunto**. Com a leitura, você pode estabelecer sua definição de um ponto de vista bastante prático, explorando elementos de sua hierarquia e sintaxe.

Você viu alguns **métodos utilizados para implementação**, juntamente com alguns **exemplos práticos** sobre eles — com uma construção bastante didática. Com isso, obtive um aprofundamento sobre programação funcional com a utilização de alguns elementos presentes no Java map.

ARRAYS MULTIDIMENSIONAIS EM JAVA

Multidimensional Arrays podem ser definidos em palavras simples como array de arrays. Os dados em matrizes multidimensionais são armazenados em forma tabular (na ordem principal da linha).

Sintaxe:

data_type [1ª dimensão] [2ª dimensão] [] .. [enésima dimensão] **array_name** = **novo data_type** [size1] [size2].... [sizeN];

Onde:

- **data_type** : tipo de dados a serem armazenados na matriz. Por exemplo: int, char, etc.
- **dimensão** : a dimensão da matriz criada. Por exemplo: 1D, 2D, etc.
- **array_name** : Nome do array
- **size1, size2,..., sizeN** : Tamanhos das dimensões respectivamente.

Exemplos:

```
Two dimensional array:
int[][] twoD_arr = new int[10][20];

Three dimensional array:
int[][][] threeD_arr = new int[10][20][30];
```

Tamanho de matrizes multidimensionais : O número total de elementos que podem ser armazenados em uma matriz multidimensional pode ser calculado multiplicando o tamanho de todas as dimensões.

Por exemplo:
O array `int [][] x = new int [10] [20]` pode armazenar um total de $(10 * 20) = 200$ elementos.

Da mesma forma, array `int [] [] [] x = new int [5] [10] [20]` pode armazenar um total de $(5 * 10 * 20) = 1000$ elementos.

Matriz bidimensional (matriz 2D)

A matriz bidimensional é a forma mais simples de uma matriz multidimensional. Um array bidimensional pode ser visto como um array de um array unidimensional para facilitar o entendimento.

Método indireto de declaração:

Declaração - Sintaxe:

```
data_type [] [] array_name = novo data_type [x] [y];
Por exemplo: int [] [] arr = new int [10] [20];
```

Inicialização - Sintaxe:

```
array_name [row_index] [column_index] = value;  
    Por exemplo: arr [0] [0] = 1;
```

Exemplo:

```
class GFG {  
    public static void main(String[] args)  
    {  
  
        int[][] arr = new int[10][20];  
        arr[0][0] = 1;  
  
        System.out.println("arr[0][0] = " + arr[0][0]);  
    }  
}
```

Saída:

```
arr [0] [0] = 1
```

Método direto de declaração:

Sintaxe:

```
data_type[][] array_name = {  
    {valueR1C1, valueR1C2, ....},  
    {valueR2C1, valueR2C2, ....}  
};  
  
For example: int[][] arr = {{1, 2}, {3, 4}};
```

Exemplo:

```
class GFG {  
    public static void main(String[] args)  
    {  
  
        int[][] arr = { { 1, 2 }, { 3, 4 } };  
  
        for (int i = 0; i < 2; i++)  
            for (int j = 0; j < 2; j++)  
                System.out.println("arr[" + i + "][" + j + "] = "  
                                    + arr[i][j]);  
    }  
}
```

Saída:

```
arr [0] [0] = 1  
cheg [0] [1] = 2  
cheg [1] [0] = 3  
chega [1] [1] = 4
```

Acessando elementos de matrizes bidimensionais

Os elementos em matrizes bidimensionais são comumente referidos por **x [i][j]**, onde 'i' é o número da linha e 'j' é o número da coluna.

Sintaxe:

```
x[row_index][column_index]
```

Por exemplo:

```
int[][] arr = new int[10][20];  
arr[0][0] = 1;
```

O exemplo acima representa o elemento presente na primeira linha e na primeira coluna.

Nota : Em matrizes, se o tamanho da matriz for N. Seu índice será de 0 a N-1. Portanto, para row_index 2, o número da linha real é $2 + 1 = 3$.

Exemplo:

```
class GFG {  
    public static void main(String[] args)  
    {  
  
        int[][] arr = { { 1, 2 }, { 3, 4 } };  
  
        System.out.println("arr[0][0] = " + arr[0][0]);  
    }  
}
```

Saída:

```
arr [0] [0] = 1
```

Representação de matriz 2D em formato tabular: Uma matriz bidimensional pode ser vista como uma tabela com 'x' linhas e 'y' colunas onde o número da linha varia de 0 a (x-1) e o número da coluna varia de 0 a (y-1). Uma matriz bidimensional 'x' com 3 linhas e 3 colunas é mostrada abaixo:

	Column 0	Column 1	Column 2
Row 0	x[0][0]	x[0][1]	x[0][2]
Row 1	x[1][0]	x[1][1]	x[1][2]
Row 2	x[2][0]	x[2][1]	x[2][2]

Imprimir matriz 2D em formato tabular:

Para produzir todos os elementos de um array bidimensional, use loops for aninhados. Para isso, são necessários dois loops for, um para percorrer as linhas e outro para percorrer as colunas.

Exemplo:

```
class GFG {  
    public static void main(String[] args)  
    {  
  
        int[][] arr = { { 1, 2 }, { 3, 4 } };  
  
        for (int i = 0; i < 2; i++) {  
            for (int j = 0; j < 2; j++) {  
                System.out.print(arr[i][j] + " ");  
            }  
  
            System.out.println();  
        }  
    }  
}
```

Saída:

```
1 2  
3 4
```

Matriz tridimensional (matriz 3D)

A matriz tridimensional é uma forma complexa de uma matriz multidimensional. Uma matriz tridimensional pode ser vista como uma matriz de matriz bidimensional para facilitar o entendimento.

Método indireto de declaração:

- **Declaração - Sintaxe:**

```
data_type [] [] [] array_name = novo data_type [x] [y] [z];  
Por exemplo: int [] [] [] arr = new int [10] [20] [30];
```

- **Inicialização - Sintaxe:**

```
array_name [array_index] [row_index] [column_index] = value;  
Por exemplo: arr [0] [0] [0] = 1;
```

Exemplo:

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

        int[][][] arr = new int[10][20][30];
        arr[0][0][0] = 1;

        System.out.println("arr[0][0][0] = " + arr[0][0][0]);
    }
}
```

Saída:

```
arr [0] [0] [0] = 1
```

Método direto de declaração:

Sintaxe:

```
data_type[][][] array_name = {
    {
        {valueA1R1C1, valueA1R1C2, ....},
        {valueA1R2C1, valueA1R2C2, ....}
    },
    {
        {valueA2R1C1, valueA2R1C2, ....},
        {valueA2R2C1, valueA2R2C2, ....}
    }
};

For example: int[][][] arr = { {{1, 2}, {3, 4}}, {{5, 6}, {7, 8}} };
```

Exemplo:

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

        int[][][] arr = { { { 1, 2 }, { 3, 4 } }, { { 5, 6 }, { 7, 8 } } };

        for (int i = 0; i < 2; i++)
            for (int j = 0; j < 2; j++)
                for (int z = 0; z < 2; z++)
                    System.out.println("arr[" + i
                                            + "]["
                                            + j + "]["
                                            + z + "] = "
                                            + arr[i][j][z]);
    }
}
```

Saída:

```
arr [0] [0] [0] = 1
arr [0] [0] [1] = 2
arr [0] [1] [0] = 3
chegada [0] [1] [1] = 4
arr [1] [0] [0] = 5
cheg [1] [0] [1] = 6
cheg [1] [1] [0] = 7
chegada [1] [1] [1] = 8
```


Acessando Elementos de Matrizes Tridimensionais

Os elementos em matrizes tridimensionais são comumente referidos por **x [i] [j] [k]** onde 'i' é o número da matriz, 'j' é o número da linha e 'k' é o número da coluna.

Sintaxe:

```
x[array_index][row_index][column_index]
```

Por exemplo:

```
int[][][] arr = new int[10][20][30];  
arr[0][0][0] = 1;
```

O exemplo acima representa o elemento presente na primeira linha e na primeira coluna da primeira matriz na matriz 3D declarada.

Nota : Em matrizes, se o tamanho da matriz for N. Seu índice será de 0 a N-1. Portanto, para row_index 2, o número da linha real é $2 + 1 = 3$.

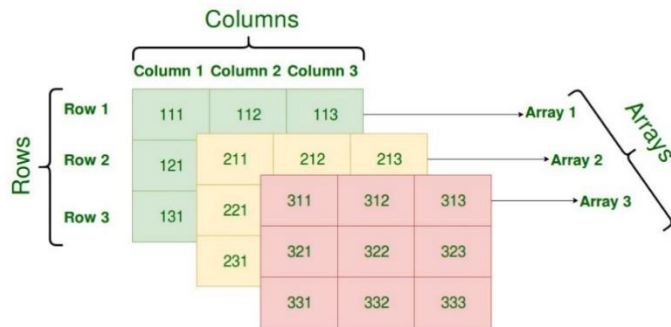
Exemplo:

```
class GFG {  
    public static void main(String[] args)  
    {  
  
        int[][][] arr = { { { 1, 2 }, { 3, 4 } }, { { 5, 6 }, { 7, 8 } } };  
  
        System.out.println("arr[0][0][0] = " + arr[0][0][0]);  
    }  
}
```

Saída:

```
arr [0] [0] [0] = 1
```

Representação de matriz 3D em formato tabular: Uma matriz tridimensional pode ser vista como uma tabela de matrizes com 'x' linhas e 'y' colunas onde o número da linha varia de 0 a (x-1) e o número da coluna varia de 0 para (y-1). Uma matriz tridimensional com 3 matrizes contendo 3 linhas e 3 colunas é mostrada abaixo:



Imprimir matriz 3D em formato tabular:

Para produzir todos os elementos de um array tridimensional, use loops for aninhados. Para isso, três loops for são necessários, um para percorrer as matrizes, o segundo para percorrer as linhas e outro para percorrer as colunas.

Exemplo:

```
class GFG {
    public static void main(String[] args)
    {
        int[][][] arr = { { { 1, 2 }, { 3, 4 } },
                          { { 5, 6 }, { 7, 8 } } };

        for (int i = 0; i < 2; i++) {
            for (int j = 0; j < 2; j++) {
                for (int k = 0; k < 2; k++) {
                    System.out.print(arr[i][j][k] + " ");
                }
                System.out.println();
            }
            System.out.println();
        }
    }
}
```

Saída:

```
1 2
3 4
5 6
7 8
```

Inserindo um array multidimensional durante o tempo de execução:

Este tópico é forçado a aceitar a entrada definida pelo usuário em um array multidimensional durante o tempo de execução. É focado no usuário, primeiro

fornecendo todas as entradas para o programa durante o tempo de execução e, depois de todas as entradas inseridas, o programa dará saída com relação a cada entrada de acordo. É útil quando o usuário deseja fazer a entrada para vários casos de teste com vários valores diferentes primeiro e depois de todas essas coisas feitas, o programa começará a fornecer a saída.

Como exemplo, vamos encontrar o número total de números pares e ímpares em uma matriz de entrada. Aqui, usaremos o conceito de array bidimensional. Aqui estão alguns pontos que explicam o uso dos vários elementos no próximo código:

- O número inteiro da linha é considerado como o número de casos de teste e os valores da coluna são considerados como valores em cada caso de teste.
- Um loop for() é usado para atualizar o número do caso de teste e outro loop for() é usado para obter os respectivos valores do array.
- Como todas as entradas de entrada são feitas, novamente dois loops for() são usados da mesma maneira para executar o programa de acordo com a condição especificada.
- A primeira linha de entrada é o número total de TestCases.
- A segunda linha mostra o número total dos primeiros valores da matriz.
- A terceira linha fornece valores de array e assim por diante.

Implementação:

```
import java.util.Scanner;

public class GFGTestCase {
    public static void main(
        String[] args)
    {
        // Scanner class to take
        // values from console
        Scanner scanner = new Scanner(System.in);

        // totalTestCases = total
        // number of TestCases
        // eachTestCaseValues =
        // values in each TestCase as
        // an Array values
        int totalTestCases, eachTestCaseValues;

        // takes total number of
        // TestCases as integer number
        totalTestCases = scanner.nextInt();

        // An array is formed as row
        // values for total testCases
        int[][] arrayMain = new int[totalTestCases][];

        // for loop to take input of
        // values in each TestCase
        for (int i = 0; i < arrayMain.length; i++) {
            eachTestCaseValues = scanner.nextInt();
            arrayMain[i] = new int[eachTestCaseValues];
            for (int j = 0; j < arrayMain[i].length; j++) {
                arrayMain[i][j] = scanner.nextInt();
            }
        }
    }
}
```

```

    } // All input entry is done.

    // Start executing output
    // according to condition provided
    for (int i = 0; i < arrayMain.length; i++) {

        // Initialize total number of
        // even & odd numbers to zero
        int nEvenNumbers = 0, nOddNumbers = 0;

        // prints TestCase number with
        // total number of its arguments
        System.out.println(
            "TestCase " + i + " with "
            + arrayMain[i].length + " values:");
        for (int j = 0; j < arrayMain[i].length; j++) {
            System.out.print(arrayMain[i][j] + " ");

            // even & odd counter updated as
            // eligible number is found
            if (arrayMain[i][j] % 2 == 0) {
                nEvenNumbers++;
            }
            else {
                nOddNumbers++;
            }
        }
        System.out.println();

        // Prints total numbers of
        // even & odd
        System.out.println(
            "Total Even numbers: " + nEvenNumbers
            + ", Total Odd numbers: " + nOddNumbers);
    }
}

```

```

Input:
2
2
1 2
3
1 2 3

Output:
TestCase 0 with 2 values:
1 2
Total Even numbers: 1, Total Odd numbers: 1
TestCase 1 with 3 values:
1 2 3
Total Even numbers: 1, Total Odd numbers: 2

Input:
3
8
1 2 3 4 5 11 55 66
5
100 101 55 35 108
6
3 80 11 2 1 5

```

```

Output:
TestCase 0 with 8 values:
1 2 3 4 5 11 55 66
Total Even numbers: 3, Total Odd numbers: 5
TestCase 1 with 5 values:
100 101 55 35 108
Total Even numbers: 2, Total Odd numbers: 3
TestCase 2 with 6 values:
3 80 11 2 1 5
Total Even numbers: 2, Total Odd numbers: 4

```

CLASSIFIQUE UMA MATRIZ DE PARES USANDO JAVA PAIR AND COMPARATOR

Dado uma matriz de pares de inteiros. A tarefa é classificar a matriz em relação ao segundo elemento do par.

Exemplos:

```

Input: [(1, 2), (3, 5), (2, 6), (1, 7)]
Output: [(1, 2), (3, 5), (2, 6), (1, 7)]

Input: [(10, 20), (20, 30), (5, 6), (2, 5)]
Output: [(2, 5), (5, 6), (10, 20), (20, 30)]

```

Abordagem:

- Armazene os pares em uma matriz usando uma *classe Pair* definida pelo usuário .
- Substitua o método comparador para classificar a matriz de acordo com o primeiro elemento.
- Classifique a matriz de acordo com o primeiro elemento.

Abaixo está a implementação da abordagem acima:

```

// Java code to sort the array
// according to second element
import java.io.*;
import java.util.*;

// User defined Pair class
class Pair {
    int x;
    int y;

    // Constructor
    public Pair(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
}

// class to define user defined comparator
class Compare {

```

```

static void compare(Pair arr[], int n)
{
    // Comparator to sort the pair according to second element
    Arrays.sort(arr, new Comparator<Pair>() {
        @Override public int compare(Pair p1, Pair p2)
        {
            return p1.y - p2.y;
        }
    });

    for (int i = 0; i < n; i++) {
        System.out.print(arr[i].x + " " + arr[i].y + " ");
    }
    System.out.println();
}

// Driver class
class GFG {
    // Driver code
    public static void main(String[] args)
    {
        Scanner sc = new Scanner(System.in);

        // length of array
        int n = 5;

        // Array of Pair
        Pair arr[] = new Pair[n];

        arr[0] = new Pair(10, 20);
        arr[1] = new Pair(1, 2);
        arr[2] = new Pair(3, 1);
        arr[3] = new Pair(10, 8);
        arr[4] = new Pair(4, 3);

        Compare obj = new Compare();

        obj.compare(arr, n);
    }
}

```

Saída:

```
3 1 1 2 4 3 10 8 10 20
```

Operadores de atribuição em Java

O operador de atribuição é utilizado para definir o valor inicial ou sobrescrever o valor de uma variável. Em seu uso, o operando à esquerda representa a variável para a qual desejamos atribuir o valor informado à direita.

Exemplo de uso:

```
int lado = 2;
float pi = 3.1426F;
String texto = "DevMedia";
lado = 3;
```

Nesse exemplo iniciamos as variáveis `lado`, `pi` e `texto`, sobrescrevendo a variável `lado` em seguida.

Operadores aritméticos

Os operadores aritméticos realizam as operações fundamentais da matemática entre duas variáveis e retornam o resultado. Caso seja necessário escrever operações maiores ou mais complexas, podemos combinar esses operadores e criar expressões, o que nos permite executar todo tipo de cálculo de forma programática.

Exemplo de uso:

```
int area = 2 * 2;
```

Esse código demonstra como calcular a área de um quadrado de lado igual a 2.

Também podemos utilizar os operadores aritméticos em conjunto com o operador de atribuição, realizando, em uma mesma instrução, as ações de calcular o valor e atribuí-lo à variável.

Exemplo de uso:

```
int area = 2;
area *= 2;
```

Nota: A segunda linha desse código é equivalente a `area = area * 2`.

Opções de operadores aritméticos

A tabela abaixo apresenta os **operadores aritméticos** da linguagem **Java**:

+	operador de adição
-	operador subtração
*	operador de multiplicação
/	operador de divisão
%	operador de módulo (ou resto da divisão)

Operadores de incremento e decremento

Os **operadores de incremento** e decremento também são bastante utilizados. Basicamente temos dois deles: `++` e `--`, os quais podem ser declarados antes ou depois da variável e incrementam ou decrementam em 1 o valor da variável.

Exemplo de uso:

```
int numero = 5;
numero++;
numero--;
//numero continuará sendo 5.
```

Quando declaramos esse operador antes da variável, o incremento é realizado antes do valor da variável ser lido para o processamento ao qual a instrução pertence. Quando declarado depois, ocorre o contrário: lê-se o valor da variável para processamento e só então o valor da variável é incrementado. Com base nisso, suponha que temos o código abaixo:

Exemplo de uso:

```
int desafioUm = 5;
System.out.println(desafioUm += ++desafioUm );

int desafioDois = 5;
System.out.println(desafioDois += desafioDois++);
```

Quais valores serão impressos no console? 10 e 10, 10 e 11, 11 e 10 ou 11 e 11? A resposta é 11 e 10.

No primeiro `println()`, `desafioUm` é incrementado antes de seu valor ser lido para compor a instrução de soma. Sendo assim, temos `desafioUm = 5 + 6`. Já no segundo `println()`, primeiro o valor é lido, resultando em `desafioDois = 5 + 5`. Somente após a leitura `desafioDois` é incrementado, e depois, recebe o valor da soma, tendo seu valor sobrescrito com o número 10.

Operadores de igualdade

Os operadores de igualdade verificam se o valor ou o resultado da expressão lógica à esquerda é igual ("`==`") ou diferente ("`!=`") ao da direita, retornando um valor booleano.

Exemplo de uso:

```
int valorA = 1;
int valorB = 2;

if(valorA == valorB){
    System.out.println("Valores iguais");
} else {
    System.out.println("Valores diferentes");
}
```

Esse código verifica se duas variáveis contêm o mesmo valor e imprime o resultado. Uma vez que as variáveis `valorA` e `valorB` possuem valores diferentes, o trecho de código presente no `else` será executado. Caso ainda não conheça as estruturas de condição, acesse: [Documentação Java: if/else e o operador ternário](#).

Opções de operadores de igualdade

A tabela abaixo apresenta os **operadores de igualdade** do **Java**:

<code>==</code>	Utilizado quando desejamos verificar se uma variável é igual a outra.
<code>!=</code>	Utilizado quando desejamos verificar se uma variável é diferente de outra.

Nota: Os operadores de igualdade normalmente são utilizados para comparar tipos primitivos (byte, short, int, long, float, double, boolean e char). No entanto, também podemos utilizá-los para saber se duas instâncias estão apontando para o mesmo objeto.

Operadores relacionais

Os **operadores relacionais**, assim como os de igualdade, avaliam dois operandos. Neste caso, mais precisamente, definem se o operando à esquerda é menor, menor ou igual, maior ou maior ou igual ao da direita, retornando um valor booleano.

Exemplo de uso:

```
int valorA = 1;
int valorB = 2;

    if (valorA > valorB) {
        System.out.println("maior");
    }

    if (valorA >= valorB) {
        System.out.println("maior ou igual");
    }

    if (valorA < valorB) {
        System.out.println("menor");
    }

    if (valorA <= valorB) {
        System.out.println("menor ou igual");
    }
```

Esse código realiza uma série de comparações entre duas variáveis para determinar o que será impresso no console. Uma vez que o valor da variável `valorA` é menor que `valorB` serão impressas as mensagens “menor” e “menor ou igual”.

Opções de operadores relacionais

A tabela abaixo apresenta os operadores relacionais do Java:

>	Utilizado quando desejamos verificar se uma variável é maior que outra.
>=	Utilizado quando desejamos verificar se uma variável é maior ou igual a outra
<	Utilizado quando desejamos verificar se uma variável é menor que outra.
<=	Utilizado quando desejamos verificar se uma variável é menor ou igual a outra.

Operadores lógicos

Os **operadores lógicos** representam o recurso que nos permite criar expressões lógicas maiores a partir da junção de duas ou mais expressões. Para

isso, aplicamos as operações lógicas E (representado por “&&”) e OU (representado por “||”).

Exemplo de uso:

```
if((1 == (2 - 1)) && (2 == (1 + 1))) {  
    System.out.println("Ambas as expressões são verdadeiras");  
}
```

Uma vez que utilizamos o operador lógico &&, o `System.out.println` somente será executado se as duas condições declaradas no `if` forem verdadeiras.

Opções de operadores de lógicos

A tabela abaixo apresenta os operadores lógicos do Java:

&&	Utilizado quando desejamos que as duas expressões sejam verdadeiras.
	Utilizado quando precisamos que pelo menos um das expressões seja verdadeira.

Precedência de operadores

Uma vez que os operadores aritméticos buscam reproduzir as operações matemáticas fundamentais, é natural que eles mantenham as suas regras de precedência, que podem ser manipuladas pelo programador com o uso de parênteses.

Por exemplo, a expressão $1 + 1 * 2$, quando analisada pelo compilador, vai retornar o valor 3, porque a multiplicação será resolvida antes da adição. Usando parênteses, a expressão $(1 + 1) * 2$ retornará o valor 4, pois a adição, por estar dentro dos parênteses, será resolvida primeiro.

Exemplo de uso:

```
if ((1 != (2 - 1)) || (2 == (1+1))) {  
    System.out.println("iguais");  
}
```

Nota: Para facilitar a leitura das expressões e evitar erros de lógica, é recomendado o uso dos parênteses para separar e agrupar as condições.

Exemplo prático

Suponha que você precisa programar um código simples para definir o salário dos funcionários de uma empresa considerando o tempo que cada um tem nessa empresa e o número de horas trabalhadas. Para tanto, podemos utilizar alguns dos operadores apresentados nessa documentação.

Exemplo de uso:

```
int quantidadeAnos = 5;
int horasTrabalhadas = 40;
int valorHora = 50;
int salario = 0;

if (quantidadeAnos <= 1) {
    salario = 1500 + (valorHora * horasTrabalhadas);
} else if ((quantidadeAnos > 1) && (quantidadeAnos < 3)) {
    salario = 2000 + (valorHora * horasTrabalhadas);
} else {
    salario = 3000 + (valorHora * horasTrabalhadas);
}
```

Introdução a Orientação ao Objeto

O QUE É PROGRAMAÇÃO ORIENTADA A OBJETOS E QUAIS SÃO SEUS PILARES?

A programação orientada à objetos oferece uma forma sustentável de escrever código espaguete. Ela permite que você escreva programas como se fosse uma colcha de retalhos.

— Paul Graham

Princípios de programação orientada a objetos

A programação orientada a objetos é um paradigma de programação onde tudo é representado como um objeto.

Objetos passam mensagens uns para os outros. Cada objeto decide o que fazer com uma mensagem recebida. A POO (em inglês, *OOP - Object Oriented Programming*) foca nos estados e comportamentos de cada objeto.

O que são objetos?

Um objeto é uma entidade que possui estados e comportamentos.

Por exemplo, cão, gato e veículo. Para ilustrar, um cão possui estados como idade, cor, nome e comportamentos como comer, dormir e correr.

Os estados nos dizem como o objeto se parece ou quais propriedades ele possui.

O comportamento nos diz o que o objeto faz.

Podemos representar um cão do mundo real em um programa como um objeto de software definindo seus estados e comportamentos.

Objetos de software são uma representação de um objeto do mundo real. É alocado espaço em memória sempre que um objeto lógico é criado.

Um objeto pode também ser referenciado como uma instância de uma classe. Instanciar uma classe significa a mesma coisa que criar um objeto.

Algo importante de se lembrar quando estamos criando um objeto é: o tipo de referência deve ser **do mesmo tipo** ou um **supertipo** do tipo do objeto. Veremos o que é um tipo de referência mais a frente neste artigo.

O que são classes?

Uma classe é um *template* - algo como a "planta" de uma construção - a partir do qual os objetos são criados.

Imagine uma classe como um cortador de biscoitos e os objetos como os próprios biscoitos.



Figura 1 - ilustra a relação entre classe e objeto por meio do cortador de biscoitos e dos biscoitos em si.

Classes definem estados como variáveis de instância e comportamentos como métodos de instância.

Variáveis de instância também são conhecidas como "variáveis membro".

Classes não consomem espaço em memória.

Para dar uma ideia sobre classes e objetos, vamos criar uma classe Cat (Gato) que representa estados e comportamentos de um gato no mundo real.

```
public class Cat {  
    /*  
    Variáveis de instancia: Estados do gato  
    */  
    String ;  
    int ;  
    String ;  
    String ;  
}
```

```

/*
Métodos de instancia: Comportamento do gato
*/
void sleep(){
    System.    .println("Dormindo");
}
void play(){
    System.    .println("Brincando");
}
void feed(){
    System.    .println("Comendo");
}
}

```

Agora que definimos um *template* de gatos, vamos dizer que temos dois gatos, chamados Thor e Rambo.



Figura 2 - Thor está dormindo.



Figura 3 - Rambo está brincando

Como podemos defini-los em nossos programas?

Primeiro, precisamos criar dois objetos da classe Cat.

```
public class Main {  
    public static void main(String[] args) {  
        Cat thor = new Cat();  
        Cat rambo = new Cat();  
    }  
}
```

Agora, vamos definir seus estados e comportamentos.

```
public class Main {  
  
    public static void main(String[] args) {  
        /*  
        Criando os objetos  
        */  
        Cat thor = new Cat();  
        Cat rambo = new Cat();  
  
        /*  
        Definindo o gato Thor  
        */  
        thor.name = "Thor";  
        thor.age = 3;  
        thor.breed = "Azul russo";  
        thor.color = "Marrom";  
  
        thor.sleep();  
  
        /*  
        Definindo o gato Rambo  
        */  
        rambo.name = "Rambo";  
        rambo.age = 4;  
        rambo.breed = "Maine Coon";  
        rambo.color = "Marrom";  
  
        rambo.play();  
    }  
}
```

Como nos exemplos de código acima, podemos definir nossas classes, instanciá-las (criar objetos) e especificar os estados e comportamentos para esses objetos.

Até aqui já cobrimos o básico de orientação a objetos. Vamos avançar para os princípios da orientação a objetos.

Princípios da programação orientada a objetos

Esses são os quatro princípios fundamentais do paradigma de programação orientada a objetos. Entendê-los é essencial para se tornar um programador de sucesso.

1. Encapsulamento
2. Herança
3. Abstração

4. Polimorfismo

Agora, vamos dar uma olhada neles com mais detalhes.

Encapsulamento

Encapsulamento é um processo de envolver dados e código em uma única unidade.

É como uma capsula que possui uma mistura de diversos medicamentos, é uma técnica que ajuda a manter as variáveis de instância protegidas.

Essa proteção pode ser conquistada utilizando o modificador de acesso `private`, que indica que a variável ou dado não pode ser acessado de fora da classe. Para acessar estados privados de modo seguro, temos que providenciar métodos *getters* e *setters* públicos (em Java, esses métodos devem seguir os padrões de nomenclatura "JavaBeans").

Digamos que existe uma loja de discos que vende álbuns de músicas de diferentes artistas e um estoque para o gerenciamento.

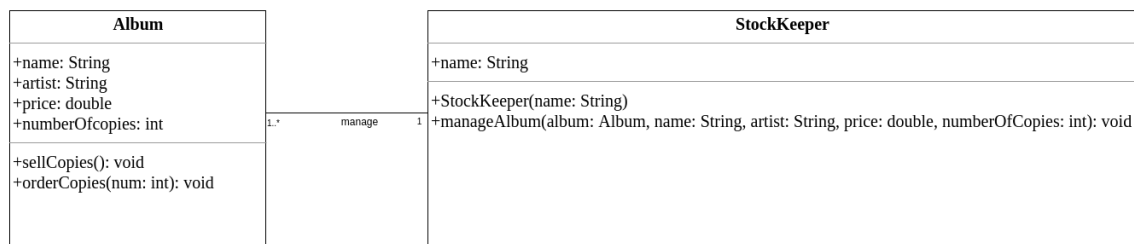


Figura 4 - diagrama de classes sem encapsulamento

Se você observar a figura 4, a classe `StockKeeper` pode acessar os estados da classe `Album` diretamente, já que os atributos da classe `Album` estão definidos como `public` (sinal de `+`).

Se o gerenciador de estoque criar um álbum e definir seus estados como negativos, ele pode acabar fazendo isso mesmo que sem intenção.

Para ilustrar, vamos ver um exemplo de um programa em Java que explica o diagrama e a afirmação acima.

Classe Album:

```
public class Album {
    public String name;
    public String artist;
    public double price;
    public int numberOfCopies;
    public void sellCopies(){
        if(numberOfCopies > 0){
            numberOfCopies--;
            System.out.println("Um album foi vendido!");
        }
        else{
            System.out.println("Não há albuns disponíveis!");
        }
    }
    public void orderCopies(int num){
        numberOfCopies += num;
    }
}
```

Classe StockKeeper:

```
public class StockKeeper {
    public String name;
    public StockKeeper(String name){
        this.name = name;
    }
    public void manageAlbum(Album album, String name, String artist, double price, int
numberOfCopies){
        /*
        Definindo os estados e comportamentos para album
        */
        album.name = name;
        album.artist = artist;
        album.price = price;
        album.numberOfCopies = numberOfCopies;

        /*
        Imprimindo os detalhes do album
        */
        System.out.println("Album gerenciado por :"+ this.name);
        System.out.println("Detalhes do album::::::::::::");
        System.out.println("Nome do album: " + album.name);
        System.out.println("Artista do Album : " + album.artist);
        System.out.println("Preço do Album : " + album.price);
        System.out.println("Número de cópias do album : " + album.numberOfCopies);
    }
}
```

Classe Main:

```
public class Main {
    public static void main(String[] args) {
        StockKeeper johnDoe = new StockKeeper("John Doe");
        /*
        O gerenciador de estoque cria um album e atribui valores negativos para o
preço e o numero de cópias disponíveis
        */
    }
}
```

```
johnDoe.manageAlbum(new Album(), "Slippery When Wet", "Bon Jovi", -
1000.00, -50);
}
}
```

Saída:

```
Album gerenciado por :John Doe
Detalhes do album:::::::::::
Nome do album : Slippery When Wet
Artista do Album : Bon Jovi
Preço do Album : -1000.0
Número de cópias do album : -50
```

O preço do álbum e número de cópias não podem ser valores negativos. Como podemos evitar essa situação? Aqui é onde usamos o encapsulamento.

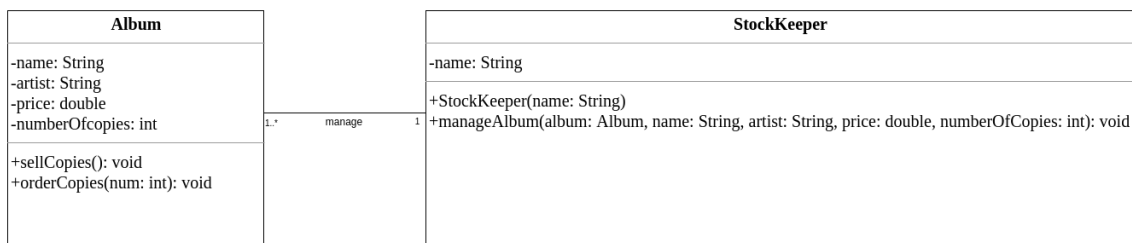


Figura 5 - diagrama de classes com encapsulamento

Neste cenário, podemos impedir o gerenciador de estoque de atribuir valores negativos. Se ele tentar atribuir valores negativos para o preço e o número de cópias do álbum, definiremos 0.0 como o valor e 0 como o número de cópias.

Classe Album:

```
public class Album {
    private String name;
    private String artist;
    private double price;
    private int numberOfCopies;
    public void sellCopies(){
        if(numberOfCopies > 0){
            numberOfCopies--;
            System.out.println("Um álbum foi vendido!");
        }
        else{
            System.out.println("Nenhum álbum disponível!");
        }
    }
    public void orderCopies(int num){
        numberOfCopies += num;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getArtist() {
        return artist;
    }
    public void setArtist(String artist) {
        this.artist = artist;
    }
    public double getPrice() {
        return price;
    }
    public void setPrice(double price) {
        if(price > 0) {
            this.price = price;
        }
        else {
            this.price = 0.0;
        }
    }
    public int getNumberOfCopies() {
        return numberOfCopies;
    }
    public void setNumberOfCopies(int numberOfCopies) {
        if(numberOfCopies > 0) {
            this.numberOfCopies = numberOfCopies;
        }
        else {
            this.numberOfCopies = 0;
        }
    }
}
```

Classe StockKeeper:

```
public class StockKeeper {
    private String name;
    StockKeeper(String name){
        setName(name);
    }
    public void manageAlbum(Album album, String name, String artist, double price, int
numberOfCopies){
        /*
         Definindo estados e comportamentos para o album
        */
        album.setName(name);
        album.setArtist(artist);
        album.setPrice(price);
        album.setNumberOfCopies(numberOfCopies);
        /*
         Imprimindo os detalhes do album
        */
        System.out.println("Album gerenciado por :"+ getName());
        System.out.println("Detalhes do album::::::::::::");
        System.out.println("Nome do album : " + album.getName());
        System.out.println("Artista do Album : " + album.getArtist());
        System.out.println("Preço do Album : " + album.getPrice());
        System.out.println("Número de cópias do album : " + album.getNumberOfCopies());
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
}
```

Classe Main:

```
public class Main {
    public static void main(String[] args) {
        StockKeeper johnDoe = new StockKeeper("John Doe");
        /*
         Gerenciador de estoque cria um album e atribui valores negativos para o preço e
o número de cópias disponíveis do album
        */
        johnDoe.manageAlbum(new Album(), "Slippery When Wet", "Bon Jovi", -1000.00, -
50);
    }
}
```

Saída:

```
Album gerenciado por :John Doe
Detalhes do album::::::::::::
Nome do album : Slippery When Wet
Artista do Album : Bon Jovi
Preço do Album : 0.0
Número de cópias do album : 0
```

Com o encapsulamento, impedimos nosso gerenciador de estoque de definir valores negativos, o que significa que temos o controle sobre as informações.

Vantagens do encapsulamento em Java

1. Podemos fazer uma classe **somente leitura** ou **somente escrita**. Para uma classe somente leitura, temos que informar apenas os métodos *getters*. Para uma classe somente escrita, devemos informar apenas os métodos *setters*.
2. Controle sobre os dados: podemos controlar os dados adicionando lógica nos métodos *setters*, assim como fizemos para evitar que o gerenciador de estoques definisse valores negativos nos exemplos acima.
3. Proteção dos dados: outras classes não podem acessar membros privados de uma classe diretamente.

Herança

Consideremos que a loja de discos que falamos anteriormente também venda filmes em Blu-ray.

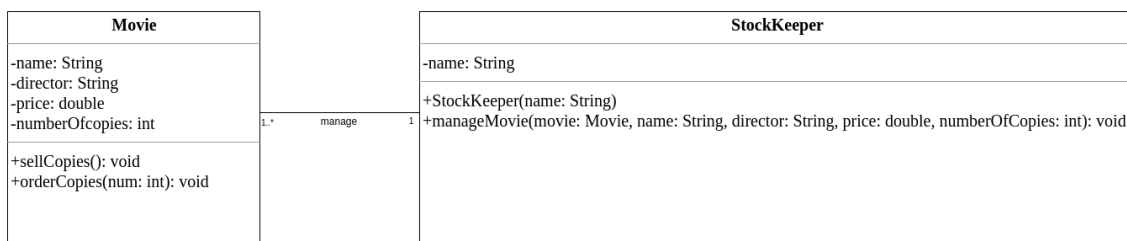


Figura 6 - diagrama de classes para Movie (filme) e StockKeeper (gerenciador de estoque)

Como você pode ver no diagrama acima, existem muitos estados e comportamentos em comum (código duplicado) entre Album e Movie.

Quando for transformar esse diagrama de classes em código, você vai reescrever/copiar todo o código novamente para Movie? Caso você faça isso, você estará se repetindo. Como você pode evitar a duplicação de código? Aqui é onde usamos a herança.

Herança é um mecanismo onde um objeto recebe todos os comportamentos e estados de um objeto pai.

A herança utiliza um relacionamento de pais e filhos (relacionamento "É um").

Então o que exatamente é herdado?

Visibilidade/modificadores de acesso impacta o que pode ser herdado de uma classe para a outra.

Em Java, como **regra fundamental**, tornamos as variáveis de instância *private* e os métodos de instância *public*.

Neste caso, certamente podemos dizer que o seguinte será herdado:

1. métodos públicos de instância.

2. variáveis de instância privadas (que podem ser acessadas apenas por meio de métodos *getters* e *setters* públicos) .

Tipos de herança no Java

Existem cinco tipos diferentes de herança no Java. Elas são: simples, multinível, hierárquica, múltipla e híbrida.

Classes permitem heranças simples, multinível e hierárquicas. Interfaces permitem heranças múltiplas e híbridas.

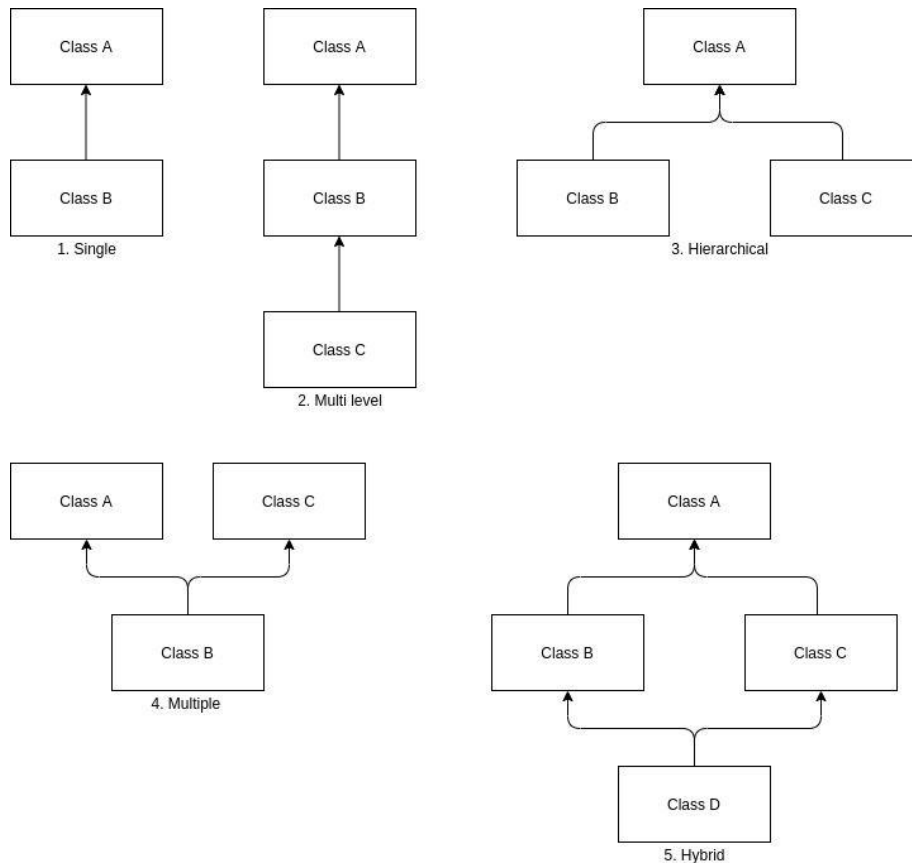


Figura 7 - tipos de herança no Java

Uma classe pode estender apenas uma classe. Entretanto, não há limite para a implementação de interfaces. Uma interface pode estender mais de uma interface.

Parent	Child	Keyword
Class	Class	extends
Interface	Interface	extends
Interface	Class	implements

Figura 8 - palavras-chaves para herança no Java

Relacionamentos

I. Relacionamento É UM

Um relacionamento É UM refere-se à herança ou implementação.

a. Generalização

Generalização usa um relacionamento É UM de uma classe especializada para uma classe generalizada.

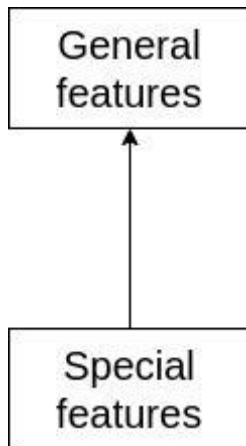


Figura 9 - diagrama de generalização (recursos especiais para recursos gerais)

II. Relacionamento TEM UM

Uma instância de uma classe TEM UMA referência para uma instância de outra classe.

a. Agregação

Neste relacionamento, a existência de uma classe A e B não são dependentes umas das outras.

Para essa parte de agregação, vamos ver um exemplo da classe Student e da classe ContactInfo.

```
class ContactInfo {
    private String homeAddress;
    private String emailAddress;
    private int telephoneNumber; //12025550156
}
public class Student {
    private String name;
    private int age;
    private int grade;
    private ContactInfo contactInfo;//Student TEM UM ContactInfo
    public void study() {
        System.out.println("Study");
    }
}
```

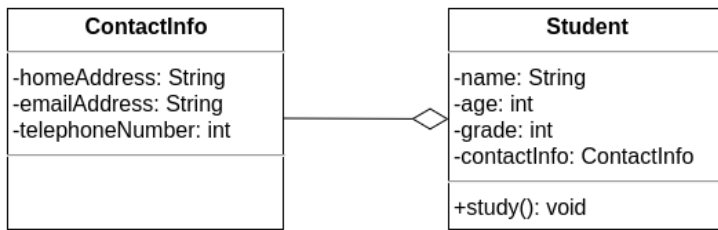



Figura 10 - : o diagrama de classes mostra um relacionamento de generalização

Student (aluno) TEM UMA ContactInfo (informação de contato).

ContactInfo pode ser usado em outros lugares – por exemplo, uma classe Employee (funcionário) de uma companhia também poderia utilizar a classe ContactInfo. Assim, Student pode existir sem ContactInfo e ContactInfo pode existir sem Student. Este tipo de relacionamento é conhecido como agregação.

b. Composição

Neste relacionamento, a classe B não pode existir sem uma classe A – mas a classe A **pode** existir sem a classe B.

Para dar uma ideia sobre composição, vamos ver esse exemplo da classe Student e a classe StudentId.

```

class StudentId {
    private String idNumber;//A-123456789
    private String bloodGroup;
    private String accountNumber;
}
public class Student {
    private String name;
    private int age;
    private int grade;
    private StudentId studentId;//Student TEM UM StudentId
    public void study() {
        System.out.println("Study");
    }
}
  
```

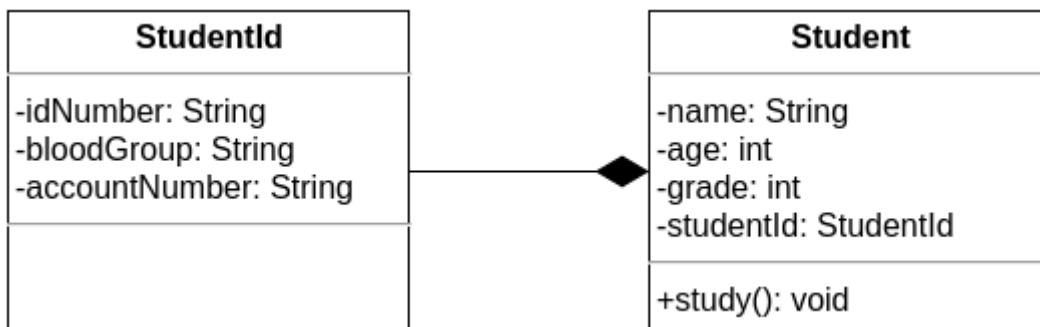


Figura 11 - diagrama de classes mostra o relacionamento de composição

Student TEM UM StudentId. Student pode existir sem StudentId, mas StudentId não pode existir sem Student. Esse tipo de relacionamento é conhecido como composição.

Agora, vamos voltar para nosso exemplo da loja de discos que discutimos acima.

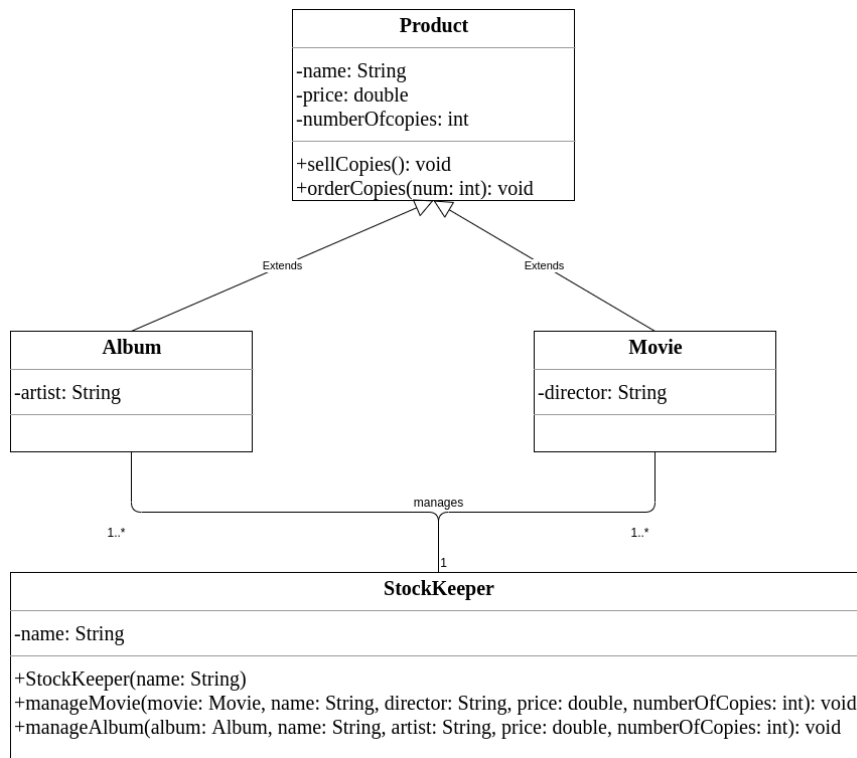


Figura 12 - diagrama de classes com herança

Podemos implementar esse diagrama em Java para evitar duplicação de código.

Vantagens de se usar herança

1. Reaproveitamento de código: as classes filhas herdam todos os membros de instância da classe pai.
2. Você tem mais flexibilidade de mudar o código: mudar o código em um lugar é o suficiente.
3. Você pode usar polimorfismo: a sobrescrita de métodos requer um relacionamento É UM.

Abstração

Abstração é o processo de esconder os detalhes de implementação e exibir apenas as funcionalidades para o usuário.

Um exemplo comum de abstração é o acelerador do carro: pisando mais forte, você aumenta a velocidade. Os motoristas, no entanto, não sabem como essa ação altera a velocidade – eles não precisam saber.

Tecnicamente, abstrato significa algo incompleto ou a ser finalizado no futuro.

Em Java, podemos obter abstração de duas maneiras: classes abstratas (0% a 100%) e interfaces (100%).

A palavra-chave `abstract` pode ser aplicada à classes e métodos. `abstract` e `final` ou `static` nunca podem estar juntas.

I. Classes abstratas

Uma classe é abstrata quando ela contém a palavra reservada `abstract`. Classes abstratas não podem ser instanciadas (não é possível criar objetos de classes abstratas). Elas podem ter construtores, métodos estáticos e métodos finais.

II. Métodos abstratos

Um método é abstrato quando ele contém a palavra chave `abstract`. Um método abstrato não possui implementação (não possui um corpo e termina com ponto e virgula). Métodos abstratos não devem ser marcados como `private`.

III. Classes abstratas e métodos abstratos

- Se pelo menos um método for abstrato dentro de uma classe, então toda a classe deve ser abstrata.
- É possível ter uma classe abstrata sem nenhum método abstrato.
- Podemos ter qualquer quantidade de métodos abstratos e não abstratos ao mesmo tempo na mesma classe.
- A primeira classe concreta que herde de uma classe abstrata deve prover implementação para **todos** os métodos abstratos.
- Caso a subclasse não implemente os métodos abstratos da superclasse, ela deve também ser marcada como abstrata.

Em um cenário real, a implementação vai ser feita por alguém desconhecido ao usuário final. Usuários não conhecem a classe de implementação nem os detalhes da implementação.

Vamos considerar um exemplo de uso do conceito de classes abstratas.

```
abstract class Shape {
    public abstract void draw();
}
class Circle extends Shape{
    public void draw() {
        System.out.println("Círculo!");
    }
}
public class Test {
    public static void main(String[] args) {
        Shape circle = new Circle();
        circle.draw();
    }
}
```

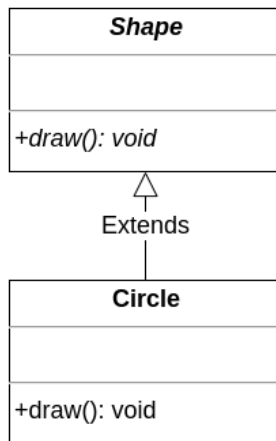


Figura 13 - diagrama de classes que mostra o relacionamento entre uma classe abstrata (Shape - ou forma em português) e uma classe concreta (Circle - ou círculo, em português)

Quando vamos querer marcar uma classe como abstrata?

1. Para forçar subclasses a implementar métodos abstratos.
2. Para impedir que existam objetos daquela classe.
3. Para manter a referência à uma classe.
4. Para manter código comum.

Interface

Uma interface é um *template* (ou uma "planta" de construção) de uma classe.

Uma interface é 100% abstrata. Construtores não são permitidos aqui. A interface representa o relacionamento "É UM".

Observação: interfaces apenas definem quais os métodos necessários. Não podemos manter código comum.

Uma interface possui apenas métodos abstratos, não possui métodos concretos. Os métodos da interface são, por padrão, `public` e `abstract`. Então, dentro da interface, não precisamos especificar as palavras-chaves `public` e `abstract`.

Então, quando uma classe implementa um método da interface sem especificar os modificadores de acesso daquele método, o compilador vai lançar uma exceção dizendo "Cannot reduce the visibility of the inherited method from interface" (Não é possível reduzir a visibilidade de um método herdado de uma interface). Sendo assim, o modificador de acesso de um método implementado de uma interface deve ser `public`.

Por padrão, as variáveis de interface são `public`, `static` e `final`.

Por exemplo:

```

interface Runnable {
    int a = 10; //equivale a: public static final int a = 10;
}
  
```

```

    void run(); //equivale a: public abstract void run();
}
public class InterfaceChecker implements Runnable{
    public static void main(String[] args) {
        Runnable.a = 5; //o valor do campo final Runnable.a não pode ser
        reatribuido.
    }
}

```

Vamos ver um exemplo que explica o conceito de interface:

```

interface Drawable {
    void draw();
}
class Circle implements Drawable{
    public void draw() {
        System.out.println("Círculo!");
    }
}
public class InterfaceChecker {
    public static void main(String[] args) {
        Drawable circle = new Circle();
        circle.draw();
    }
}

```

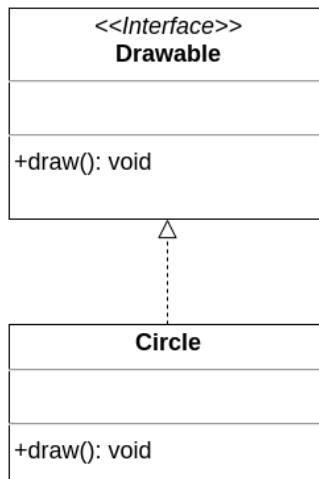


Figura 14 - diagrama de classes que mostra o relacionamento entre uma interface e uma classe concreta (Drawable - ou desenhável, em português e Circle - ou círculo, em português)

Métodos default e métodos estáticos nas Interfaces

Normalmente, implementamos os métodos de uma interface em classes separadas. Digamos que seja necessário adicionar um novo método à interface. Então, deveremos implementar esse método em todas as outras classes que implementam essa interface também.

Para evitar esse tipo de problema, a partir do Java, 8 foi introduzida a possibilidade de implementar métodos default e estáticos dentro de uma interface, além dos métodos abstratos.

- **Método default**

```

public interface DefaultInterface {
    void sleep();
    default void run() {
        System.out.println("Estou correndo!");
    }
}

public class InterfaceCheckers implements DefaultInterface{
    public void sleep() {
        System.out.println("Dormindo...");
    }
    public static void main(String[] args) {
        InterfaceCheckers checker = new InterfaceCheckers();
        checker.run();
        checker.sleep();
    }
}
/*
Saída:
Estou correndo!
Dormindo...
*/
  
```

- **Métodos estáticos**

Assim como métodos estáticos de classes, nós podemos chamá-los usando o nome da interface.

```
public interface DefaultInterface {
    void sleep();
    static void run() {
        System.out.println("Estou correndo!");
    }
}

public class InterfaceCheckers implements DefaultInterface{
    public void sleep() {
        System.out.println("Dormindo...");
    }
    public static void main(String[] args) {
        InterfaceCheckers checker = new InterfaceCheckers();
        DefaultInterface.run();
        checker.sleep();
    }
}

/*
Saída:
Estou correndo!
Dormindo...
*/
```

- **Interface de marcação**

São interfaces vazias. Por exemplo as interfaces *Serializable*, *Cloneable* e *Remote*.

```
public interface Serializable
{
    //Sem campos ou métodos
}
```

Vantagens das interfaces

- Nos ajudam a utilizar herança múltipla no Java.
- Elas fornecem abstração.
- Elas fornecem baixo acoplamento: os objetos são independentes uns dos outros.

Quando vamos querer mudar de uma classe para uma interface?

1. Para forçar subclasses a implementar métodos abstratos.
2. Para evitar a criação de objetos dessa classe.
3. Para manter a referência à uma classe.

Observação: lembre-se de que não podemos manter código comum dentro de uma interface.

Caso você queira definir métodos que podem ser necessários e código comum, use uma **classe abstrata**.

Caso queira apenas definir métodos necessários, use uma **interface**.

Polimorfismo

Polimorfismo é a habilidade de um objeto de assumir diversas formas.

Polimorfismo em POO acontece quando uma superclasse faz referência a um objeto de uma subclasse.

Todos os objetos do Java são considerados polimórficos já que eles compartilham mais de um relacionamento. É UM (pelo menos, todos os objetos vão passar no teste É UM para seu próprio tipo e para a classe *Object*).

Podemos acessar um objeto através de uma variável de referência. Uma variável de referência pode ser apenas de um tipo. Após declarada, o tipo da variável de referência não pode ser alterado.

Uma variável de referência pode ter uma classe ou uma interface como tipo.

Um único objeto pode ser referenciado por variáveis de referência de diversos tipos (desde que esses tipos estejam na mesma hierarquia), sendo o objeto do mesmo tipo da variável de referência ou de uma superclasse.

Sobrecarga de métodos

Se uma classe possui vários métodos, que possuem o mesmo nome mas parâmetros diferentes, isso é conhecido como sobrecarga de métodos (*method overload*).

Regras da sobrecarga de métodos:

1. Precisa ter uma lista de parâmetros diferente.
2. Pode possuir tipos de retorno diferentes.
3. Pode possuir modificadores de acesso diferentes.
4. Pode lançar exceções diferentes.

```
class JavaProgrammer{
    public void code() {
        System.out.println("Programando em C++");
    }
    public void code(String language) {
        System.out.println("Programando em " + language);
    }
}
public class MethodOverloader {
    public static void main(String[] args) {
        JavaProgrammer gosling = new JavaProgrammer();
        gosling.code();
        gosling.code("Java");
    }
}
```



```

}
/*
Saída:
Programando em C++
Programando em Java
*/

```

Observação: métodos estáticos também podem sofrer sobrecarga.

```

class Addition {
    public static int add(int a,int b) {
        return a+b;
    }
    public static int add(int a,int b,int c) {
        return a+b+c;
    }
}

public class PolyTest {
    public static void main(String[] args) {
        System.out.println(Addition.add(5, 5));
        System.out.println(Addition.add(2, 4, 6));
    }
}

```

Observação: podemos sobrecarregar o método *main()*, mas a JVM (*Java Virtual Machine*) vai chamar o método *main()* que recebe um *array* de *strings* como parâmetro.

```

public class PolyTest {
    public static void main() {
        System.out.println("main()");
    }
    public static void main(String args) {
        System.out.println("String args");
    }
    public static void main(String[] args) {
        System.out.println("String[] args");
    }
}
//Saída: String[] args

```

Regras a seguir para o polimorfismo

Regras de tempo de compilação

1. O compilador conhece apenas tipos de referência.
2. Ele só pode procurar métodos em tipos de referência.
3. Gera uma assinatura de método.

Regras de tempo de execução

1. Em tempo de execução, a JVM segue o tipo de tempo de execução exato (**tipo de objeto**) para localizar o método.
2. Deve corresponder a assinatura do método de tempo de compilação ao método na classe real do objeto.

Sobrescrita de métodos

Se uma subclasse tem o mesmo método que foi declarado na superclasse, isso é conhecido como **sobrescrita de métodos (method override)**.

Regras da sobrescrita de métodos:

1. Deve possuir a mesma lista de parâmetros.
2. Deve possuir o mesmo tipo de retorno: embora um retorno covariante nos permita alterar o tipo de retorno do método sobrescrito.
3. Não pode possuir um modificador de acesso mais restritivo: deve possuir um modificador de acesso menos restritivo.
4. Não deve lançar uma exceção verificada (*checked exception*) nova ou mais ampla: pode lançar exceções verificadas mais restritas e pode lançar qualquer exceção não verificada.
5. Apenas métodos herdados podem ser sobrescritos (é necessário um relacionamento É UM).

Exemplo de sobrescrita de método:

```
public class Programmer {
    public void code() {
        System.out.println("Programando em C++");
    }
}
public class JavaProgrammer extends Programmer{
    public void code() {
        System.out.println("Programando em Java");
    }
}
public class MethodOverridder {
    public static void main(String[] args) {
        Programmer ben = new JavaProgrammer();
        ben.code();
    }
}
/*
Saída:
Programando em Java
*/
```

Observação: métodos estáticos não podem ser sobrescritos porque métodos são sobrescritos em tempo de execução (*runtime*). Métodos estáticos são associados com classes, enquanto métodos de instância são associados com objetos. Então, em Java, o método `main()` não pode ser sobrescrito.

Observação: construtores podem ser sobrecarregados mas não podem ser sobrescritos.

Tipos de objeto e tipos de referência

```

class Person{
    void eat() {
        System.out.println("A pessoa está comendo");
    }
}
class Student extends Person{
    void study() {
        System.out.println("O estudante está estudando");
    }
}
public class InheritanceChecker {
    public static void main(String[] args) {
        Person alex = new Person();//New Person "É UMA" Person
        alex.eat();
        Student jane = new Student();//New Student "É UM" Student
        jane.eat();
        jane.study();
        Person mary = new Student();//New Student "É UM" Person
        mary.eat();
        //Student chris = new Person(); //New Person não É UM Student.
    }
}

```

Em `Person mary = new Student();` esse modo de criar o objeto está correto. `mary` é uma variável de referência do tipo `Person` e `new Student()` criará um novo objeto do tipo `Student`.

`mary` não pode acessar `study()` em tempo de compilação porque o compilador conhece apenas o tipo de referência. Já que `study()` não existe na classe do tipo de referência, não é possível acessá-lo. Em tempo de execução, no entanto, `mary` será do tipo `Student` (tipo de tempo de execução/tipo de objeto). Neste caso, podemos convencer o compilador dizendo "em tempo de execução, `mary` será do tipo `Student`, então permita-me chamar o método `study()`". Como podemos convencer o compilador a fazer isso? Aqui é onde usaremos o *casting*.

Podemos converter `mary` ao tipo `Student` em tempo de compilação e chamar `study()` se utilizarmos o *casting*.

```

((Student)mary).study();

```

Aprenderemos sobre *casting* a seguir.

Casting de tipos de objeto

O *casting* no Java é classificado em dois tipos:

1. *Casting* de ampliação (*widening casting*, implícito): conversão automática de tipos.
2. *Casting* de restrição (*narrowing casting*, explícito): precisa de uma conversão explícita.

Com primitivos, long é um tipo mais abrangente que int. Assim como ocorre com os objetos, a classe mãe é mais abrangente que a classe filha.

A variável de referência apenas faz referência a um objeto. Utilizar *casting* em uma variável de referência não muda o tipo do objeto em memória, mas rotula o mesmo objeto de uma outra maneira para se ter acesso aos membros de instância.

I. Widening casting (*casting* de ampliação)

`Superclass superRef = new Subclass();`

II. Narrowing casting (*casting* de restrição)

`Subclass ref = (Subclass) superRef;`

Temos que tomar cuidado quando utilizamos um *casting* restritivo. Quando utilizamos o *casting* restritivo, convencemos o compilador a compilar sem erros. Caso estejamos errados, receberemos um erro em tempo de execução (geralmente, `ClassCastException`).

Para executar um *casting* restritivo corretamente, usamos o operador `instanceof`. Com ele, podemos verificar se o relacionamento é do tipo É UM.

```
class A {
    public void display(){
        System.out.println("Class A");
    }
}

class B extends A{
    public void display(){
        System.out.println("Class B");
    }
}

public class Test {
    public static void main(String[] args) {
        A objA = new B();
        if(objA instanceof B){
            ((B)objA).display();
        }
    }
}

/**
 * Saída: Class B
 */
```

Devemos sempre nos lembrar de uma coisa muito importante quando criamos um objeto utilizando a palavra-chave `new`: o tipo de referência deve ser **do mesmo tipo** ou de um **supertipo** do objeto sendo criado.

Diferenciais entre um double, float e um BigDecimal, e quando seria apropriado usar cada opção.

Vale acrescentar que, todos esses tipos de dados são usados para representar um número decimal. Vamos começar!

Float:

Também representa números de ponto flutuante, porém não tem precisão dupla. O float é representado por 32 bits. Ele tem 24 dígitos de precisão.

Double:

Representa números de ponto flutuante de precisão dupla (por isso o nome). O double é representado por 64 bits, ou seja, ele tem o dobro de precisão que um float, e possui 53 dígitos de precisão.


BigDecimal:

O BigDecimal é uma classe do Java, criada para nos auxiliar quando precisarmos realizar cálculos exatos, ou seja, quando a precisão é muito importante. Eles são números decimais imutáveis, e sua classe possui operações como manipulação de escala, arredondamento, comparação e conversões de formato.

O BigDecimal tem uma grande desvantagem: ele é mais lento comparado ao double e float, isso porque a classe armazena o valor decimal como uma String, e usa aritmética decimal de precisão arbitrária, o que acaba demandando mais memória e tempo, além disso, você só pode usar os próprios métodos da classe para fazer suas operações matemáticas.

Quando devo usar cada um deles?

Double: Você pode usar o double quando a precisão for importante, mas não crucial.

BigDecimal: Você pode usar o tipo quando estiver trabalhando com um atributo que representa dinheiro , ou que a precisão é MUITO importante.

Float: Sugiro usar quando a precisão não for importante, quando a memória é limitada ou quando a velocidade é importante, um caso de uso poderia ser no desenvolvimento de jogos.

Trabalhar com sistema que manipulam valores monetários é uma tarefa crítica e que deve exigir o máximo de atenção do desenvolvedor. Como exemplo, temos sistemas de Caixa de Lojas, Supermercados, Sistemas Bancários e etc.

Um cálculo errado em um **Sistema Bancário** pode ocasionar grandes transtornos para o Banco, assim como em qualquer outra ocasião onde se faz necessária a manipulação de dinheiro. Sabendo disto, explicaremos neste artigo porque utilizar a classe `BigDecimal` em vez do `Double` para realizar operações monetárias, assim como as funções mais comuns e utilizadas do `BigDecimal` e como utilizá-las.

Double vs BigDecimal

Ao trabalhar com dinheiro precisamos de valores exatos, nem 1 centavo a menos nem 1 centavo a mais, afinal, há muita coisa envolvida neste tipo de processo. Quando trabalhamos com `Double` (seja o `Double` wrapper ou mesmo o `double` tipo primitivo) não há uma precisão de casas decimais que precisamos, podemos perceber isso em somas simples onde o resultado deveria ser simples, mas acaba sendo estranho e confuso. Veja o exemplo na **Listagem 1**.

```
public class MyAppBigDecimal {  
    /**  
     * @param args  
     */  
    public static void main(String[] args) {  
        double d1 = 0.1;  
        double d2 = 0.2;  
        System.out.println(d1 + d2);  
    }  
}
```

Listagem 1 - Double gerando valores estranhos

A saída do código acima será “0.30000000000000004”, mas nós esperávamos apenas um “0.3”, não? O problema todo está na forma de representar o tipo `double` na `JVM`, pois ela adota uma representação binária que segue o padrão IEEE 754.

Pelo fato da JVM trabalhar com representação binária para o tipo `double`, o valor “0.1” é transformado para binário e vira uma dízima periódica, ou seja, um valor infinito, algo como “0.110011001100..”.

Agora vamos realizar o exemplo da **Listagem 1**, mas com o $d1 = 0.25$, que é representado por “0.1” em binário, ou seja, é um valor finito e exato. Observe a **Listagem 2**.

```
public class MyAppBigDecimal {
    /**
     * @param args
     */
    public static void main(String[] args) {
        double d1 = 0.25;
        double d2 = 0.2;
        System.out.println(d1 + d2);
    }
}
```

Listagem 2 - Double gerando valor correto com 0.25

A saída do código acima será exatamente o que você espera: “0.45”, isso porque como dissemos anteriormente, o 0.25 é um número representado de forma finita pela JVM.

Se você ainda não está convencido que o tipo double pode lhe causar problemas, sérios problemas quando você precisa de exatidão nos valores, vamos ver na **Listagem 3** vários valores sendo calculados de forma imprecisa pelo double.

```
public class MyAppBigDecimal {
    /**
     * @param args
     */
    public static void main(String[] args) {
        System.out.println(2.00 - 1.1);
        System.out.println(2.00 - 1.2);
        System.out.println(2.00 - 1.3);
        System.out.println(2.00 - 1.4);
        System.out.println(2.00 - 1.5);
        System.out.println(2.00 - 1.6);
        System.out.println(2.00 - 1.7);
        System.out.println(2.00 - 1.8);
        System.out.println(2.00 - 1.9);
        System.out.println(2.00 - 2);
    }
}
```

Saída:

```
0.8999999999999999
0.8
0.7
0.6000000000000001
0.5
0.3999999999999999
0.30000000000000004
0.19999999999999996
0.10000000000000009
0.0
```

Listagem 3 - Calculando valores com double

Dada as situações apresentadas acima, é perceptível que não podemos contar com o double para realizar cálculos que envolvemos precisão. E não só para dinheiro, mas para qualquer outra regra de negócio que exija precisão em todas

as casas decimais como por exemplo, projetos de construção civil. Mas então qual a saída para toda essa precisão? Utilizar a classe **BigDecimal**.

A classe `BigDecimal` trabalha com pontos flutuantes de precisão arbitrária, ou seja, você consegue estipular qual o nível de precisão você precisa trabalhar. Diferentemente do `double` que trabalha com representação binária, o `BigDecimal` guardará seu valor usando a base decimal. Por exemplo: enquanto que o `double` tenta representar o valor “0.1” em binário e encontra uma díxima, o `BigDecimal` representa o mesmo valor através da base decimal “ 1×10^{-1} ”.

Utilizando o mesmo exemplo da **Listagem 1**, somaremos o valor “0.1” + “0.2”, mas agora com `BigDecimal`. Veremos que a saída será outra, conforme a **Listagem 4**.

```
import java.math.BigDecimal;

public class MyAppBigDecimal {

    /**
     * @param args
     */
    public static void main(String[] args) {
        BigDecimal big1 = new BigDecimal("0.1");
        BigDecimal big2 = new BigDecimal("0.2");
        BigDecimal bigResult = big1.add(big2);
        System.out.println(bigResult.toString());
    }
}
```

Listagem 4 - Somando valores com BigDecimal

A saída do código acima será exatamente “0.3”. Mas cuidado, o construtor do `BigDecimal` pode trazer surpresas inesperadas e acabar causando grandes dores de cabeça.

Perceba que no código da **Listagem 4** usamos o construtor do `BigDecimal` passando como parâmetro Strings e não valores “doubles”, isso porque o próprio `javadoc` deste construtor nos aconselha a usar construtor do `BigDecimal` com Strings. Veja o que ocorre no exemplo da **Listagem 5**.

```
import java.math.BigDecimal;

public class MyAppBigDecimal {

    /**
     * @param args
     */
    public static void main(String[] args) {
        BigDecimal big1 = new BigDecimal(0.1);
        BigDecimal big2 = new BigDecimal(0.2);
        BigDecimal bigResult = big1.add(big2);
        System.out.println(bigResult.toString());

    }

}
```

Saída:

0.30000000000000000166533453693773481063544750213623046875

Listagem 5 - Usando construtor double do `BigDecimal`

Acontece que os números 0.1, e 0.2 já são do tipo double e ao passar para o construtor do `BigDecimal` ocorre uma transformação com imprecisão, que é exatamente o que não queremos. Então fica o conselho: utilize sempre Strings no construtor do `BigDecimal`.

Sempre que trabalhar com `BigDecimal` você precisa chamar seus métodos para realizar operações aritméticas, assim teremos garantia da precisão que tanto precisamos. Sendo assim, você jamais poderá fazer “`bigDecimal1 + bigDecimal2`” ou mesmo “`bigDecimal1 – bigDecimal2`”. Para isso existem métodos específicos, que veremos na **Listagem 6**.

```
import java.math.BigDecimal;
public class MyAppBigDecimal {
    /**
     * @param args
     */
    public static void main(String[] args) {
        System.out.println("Subtrai");
        System.out.println(new BigDecimal("2.00").subtract(new BigDecimal("1.1")));

        System.out.println("");
        System.out.println("Soma");
        System.out.println(new BigDecimal("2.00").add(new BigDecimal("1.2")));

        System.out.println("");
        System.out.println("Compara");
        System.out.println(new BigDecimal("2.00").compareTo(new BigDecimal("1.3")));

        System.out.println("");
        System.out.println("Divide");
        System.out.println(new BigDecimal("2.00").divide(new BigDecimal("2.00")));

        System.out.println("");
        System.out.println("Máximo");
    }
}
```

```

        System.out.println(new BigDecimal("2.00").max(new BigDecimal("1.5")));

        System.out.println("");
        System.out.println("Mínimo");
        System.out.println(new BigDecimal("2.00").min(new BigDecimal("1.6")));

        System.out.println("");
        System.out.println("Potência");
        System.out.println(new BigDecimal("2.00").pow(2));

        System.out.println("");
        System.out.println("Multiplica");
        System.out.println(new BigDecimal("2.00").multiply(new BigDecimal("1.8")));
    }
}

```

Listagem 6 - Realizando operações aritméticas com BigDecimal

Temos ainda um caso interessante no `BigDecimal`: este não faz nenhum tipo de arredondamento por si próprio, ou seja, você precisa especificar como deseja arredondar determinado valor se for necessário, caso contrário, ele lançará uma exceção `java.lang.ArithmeticException`. Quando desejamos dividir $1/3$, por exemplo, o resultado seria uma dízima periódica, e como o `BigDecimal` não arredondará o valor para você, ele retornará uma exceção apontando este problema. Sendo assim, precisamos dizer explicitamente como desejamos que o arredondamento ocorra caso haja uma dízima. Vejamos a **Listagem 7**.

```

//Código errado retornando o Exception
import java.math.BigDecimal;
public class MyAppBigDecimal {
    /**
     * @param args
     */
    public static void main(String[] args) {
        System.out.println("Divide");
        System.out.println(new BigDecimal("1.00")
            .divide(new BigDecimal("1.3")));
    }
}
Saída:
Divide
Exception in thread "main" java.lang.ArithmeticException:
Non-terminating decimal expansion; no exact representable decimal result.
    at java.math.BigDecimal.divide(BigDecimal.java:1603)
    at MyAppBigDecimal.main(MyAppBigDecimal.java:11)

```

```
//Código Correto retornando valor arredondado
import java.math.BigDecimal;
import java.math.RoundingMode;

public class MyAppBigDecimal {
    /**
     * @param args
     */
    public static void main(String[] args) {
        System.out.println("Divide");
        System.out.println(new BigDecimal("1.00").divide
            (new BigDecimal("1.3"),3,RoundingMode.UP));
    }
}
Saída:
Divide
0.770
```

Listagem 7 - Arredondando para evitar ArithmeticException

Perceba que no código acima optamos por arredondar o valor para cima e deixar três casas decimais após a virgula.

Java Streams API: manipulando coleções de forma eficiente

Motivação

Entre as diversas funcionalidades adicionadas à **linguagem Java em sua versão 8** está a Streams API, recurso que oferece ao desenvolvedor a possibilidade de trabalhar com conjuntos de elementos de forma mais simples e com um número menor de linhas de código. Isso se tornou possível graças à incorporação do **paradigma funcional**, combinado com as **expressões lambda**, o que facilita a manutenção do código e aumenta a eficiência no processamento devido ao uso de paralelismo.

A **proposta em torno da Streams API** é reduzir a preocupação do desenvolvedor com a forma de implementar controle de fluxo ao lidar com coleções, deixando isso a cargo da API. A ideia é iterar sobre essas coleções de objetos e, a cada elemento, realizar alguma ação, seja ela de filtragem, mapeamento, transformação, etc. Caberá ao desenvolvedor apenas definir qual ação será realizada sobre o objeto.

Como criar streams

O primeiro passo para trabalhar com streams é saber como criá-las, e a forma mais comum para isso é através de uma **coleção de dados**. A **Listagem 1** mostra como criar uma stream ao invocar o método `stream()` a partir da interface `java.util.Collection`.

Nesse trecho de código, primeiramente uma lista de strings é definida e três objetos são adicionados a ela. Em seguida, uma stream de strings é obtida ao chamar o método `items.stream()`, na linha 7. Outra forma de criar streams é invocando o método `parallelStream()`, que possibilitará paralelizar o seu processamento, oferecendo maior eficiência ao processamento.

```
01 List<String> items = new ArrayList<String>();
02
03 items.add("um");
04 items.add("dois");
05 items.add("três");
06
07 Stream<String> stream = items.stream();
```

Listagem 8 - Criação de uma stream a partir de um List.

O método `stream()` também foi adicionado a outras interfaces, como a `java.util.map`, a partir da qual também podemos obter streams da mesma forma que vimos anteriormente.

Além disso, uma stream pode ser gerada a partir de I/O, arrays e valores. Para isso, basta chamar os métodos estáticos `Files.lines()`, `Stream.of()`, `Arrays.stream()`, respectivamente, como mostra o código a seguir:

```
Stream<String> lines= Files.lines(Paths.get("myFile.txt"),
Charset.defaultCharset());
Stream<Integer> numbersFromValues = Stream.of(1, 2, 3, 4, 5);
IntStream numbersFromArray = Arrays.stream(new int[] {1, 2, 3, 4, 5});
```

Iterando sobre streams

Para iterar sobre uma **coleção de dados** usando streams, ou seja, percorrer os elementos em um loop, a API oferece o método `forEach`, que deve ser invocado a partir da stream e recebe como parâmetro a ação que será realizada em cada iteração.

No código abaixo, temos um exemplo de uso dessa funcionalidade, no qual criamos uma stream a partir de uma lista de objetos do tipo `Pessoa` e iteramos sobre ela, exibindo o nome de cada pessoa presente na lista:

```
List<Pessoa> pessoas = populaPessoas(); //Cria uma lista de Pessoa
pessoas.stream().forEach(pessoa -> System.out.println(pessoa.getNome()));
```

Perceba que o argumento do método `forEach` foi passado utilizando o recurso de **expressões lambda**, no qual o operador `pessoa` faz referência a cada item da lista durante as iterações.

Seguindo essa ideia, outras operações mais complexas podem ser realizadas com cada item da lista, bastando indicar o método responsável por esse processamento como argumento de `forEach`.

Métodos das streams

Normalmente, para realizar ações em uma lista, como filtros e operações matemáticas, é necessário efetuar um loop sobre seus itens. Com a Streams API esse tipo de tarefa também foi simplificado, bastando agora fazer chamadas a métodos específicos que, em conjunto com as expressões lambda recebidas como parâmetro, se responsabilizam por percorrer a coleção e retornar apenas o resultado esperado.

Um desses métodos é o `filter()`, que é utilizado para filtrar elementos de uma stream de acordo com uma condição (predicado passado por parâmetro). O trecho de código abaixo mostra um exemplo de uso dessa operação. Primeiramente, é criada uma lista com alguns objetos do tipo `Pessoa`. Em seguida, é criada a stream e logo após o método `filter()` recebe como parâmetro uma condição, representada por uma expressão lambda, que tem por objetivo buscar todas as pessoas cuja nacionalidade é a brasileira.

```
List<Pessoa> pessoas = populaPessoas(); //Cria uma lista de Pessoa
Stream<Pessoa> stream = pessoas.stream().filter(pessoa ->
pessoa.getNacionalidade().equals("Brasil"));
```

Outro método bastante útil nesse contexto é o `average()`, que permite calcular a média dos valores dos elementos. A **Listagem 9** mostra um exemplo de uso dessa operação, no qual é calculada a média de idade de todas as pessoas que nasceram no Brasil.

```
01 List<Pessoa> pessoas = new Pessoa().populaPessoas();
02 double media = pessoas.stream()
03 filter(pessoa -> pessoa.getNacionalidade().equals("Brasil"))
04 mapToInt(pessoa -> pessoa.getIdade())
05 average()
06.getAsDouble();
```

Listagem 9 - Exemplo de uso do método `average()`;

De forma semelhante, também é possível realizar outras operações, como obter os valores máximo, mínimo e a soma dos elementos, sempre utilizando métodos simples e sem a necessidade de aplicar explicitamente laços de repetição.

Visão geral

Neste artigo, vamos dar uma olhada rápida em uma das principais partes da nova funcionalidade que o Java 8 adicionou – Streams.

Explicaremos o que são fluxos e mostraremos a criação e as operações básicas de fluxo com exemplos simples.

2. API de fluxo

Um dos principais novos recursos do Java 8 é a introdução da funcionalidade de fluxo – `java.util.stream` – que contém classes para processamento de seqüências de elementos.

A classe de API central é `Stream<T>`. A seção a seguir demonstrará como os fluxos podem ser criados usando as fontes de provedor de dados existentes.

2.1. Criação de Stream

Os fluxos podem ser criados a partir de diferentes fontes de elementos, por exemplo, coleções ou matrizes com a ajuda dos métodos `stream()` e `of()`:

```
String[] arr = new String[]{"a", "b", "c"};
Stream<String> stream = Arrays.stream(arr);
stream = Stream.of("a", "b", "c");
```

Um método padrão `stream()` é adicionado à interface `Collection` e permite criar um `Stream<T>` usando qualquer coleção como uma fonte de elemento:

```
Stream<String> stream = list.stream();
```

2.2. Multi-threading com fluxos

A API de fluxo também simplifica o multithreading fornecendo o método `parallelStream()` que executa operações sobre os elementos do fluxo no modo paralelo.

O código abaixo nos permite executar o método `doWork()` em paralelo para cada elemento do fluxo:

```
list.parallelStream().forEach(element -> doWork(element));
```

Na seção a seguir, apresentaremos algumas das operações básicas da API de fluxo.

3. Operações de fluxo

Há muitas operações úteis que podem ser executadas em um fluxo.

Eles são divididos em **operações intermediárias** (return `Stream<T>`) e **operações terminais** (return a result of definite type). Operações intermediárias permitem o encadeamento.

Também vale a pena notar que as operações em fluxos não alteram a origem.

Aqui está um exemplo rápido:

```
long count = list.stream().distinct().count();
```

Assim, o método `distinct()` representa uma operação intermediária, que cria um novo fluxo de elementos exclusivos do fluxo anterior. E o método `count()` é uma operação de terminal, que retorna o tamanho do fluxo.

3.1. Iteração

A API de fluxo ajuda a substituir loops *por*, *para cada* e *enquanto*. Ele permite concentrar-se na lógica da operação, mas não na iteração sobre a sequência de elementos. Por exemplo:

```
for (String string : list) {  
    if (string.contains("a")) {  
        return true;  
    }  
}
```

Este código pode ser alterado apenas com uma linha de código Java 8:

```
boolean isExist = list.stream().anyMatch(element -> element.contains("a"));
```


3.2. Filtragem

O método *filter()* nos permite escolher um fluxo de elementos que satisfazem um predicado.

Por exemplo, considere a seguinte lista:

```
ArrayList<String> list = new ArrayList<>();  
list.add("One");  
list.add("OneAndOnly");  
list.add("Derek");  
list.add("Change");  
list.add("factory");  
list.add("justBefore");  
list.add("Italy");  
list.add("Italy");  
list.add("Thursday");  
list.add("");  
list.add("");
```

O código a seguir cria um *Stream<String>* da *List<String>*, localiza todos os elementos desse fluxo que contêm o caractere "d" e cria um novo fluxo contendo apenas os elementos filtrados:

```
Stream<String> stream = list.stream().filter(element -> element.contains("d"));
```

3.3. Mapeamento

Para converter elementos de um *Stream* aplicando uma função especial a eles e coletar esses novos elementos em um *Stream*, podemos usar o método *map()*:

```
List<String> uris = new ArrayList<>();  
uris.add("C:\\My.txt");  
Stream<Path> stream = uris.stream().map(uri -> Paths.get(uri));
```

Assim, o código acima converte *Stream<String>* para o *Stream<Path>* aplicando uma expressão lambda específica a cada elemento do *Stream* inicial.

Se você tiver um fluxo onde cada elemento contém sua própria sequência de elementos e você deseja criar um fluxo desses elementos internos, você deve usar o método *flatMap()*:

```
List<Detail> details = new ArrayList<>();  
details.add(new Detail());  
Stream<String> stream  
    = details.stream().flatMap(detail -> detail.getParts().stream());
```

Neste exemplo, temos uma lista de elementos do tipo *Detail*. A classe *Detail* contém um campo *PARTS*, que é um *List<String>*. Com a ajuda do método *flatMap()*, cada elemento do campo *PARTS* será extraído e adicionado ao novo fluxo resultante. Depois disso, o *Stream<Detail>* inicial será perdido.

3.4. Correspondência

A API de fluxo fornece um conjunto útil de instrumentos para validar elementos de uma sequência de acordo com algum predicado. Para fazer isso, um dos seguintes métodos pode ser usado: *anyMatch()*, *allMatch()*, *noneMatch()*. Seus nomes são autoexplicativos. São operações terminais que retornam um *booleano*:

```
boolean isValid = list.stream().anyMatch(element -> element.contains("h")); // true
boolean isValidOne = list.stream().allMatch(element -> element.contains("h")); // false
boolean isValidTwo = list.stream().noneMatch(element -> element.contains("h")); // false
```

Para fluxos vazios, o método *allMatch()* com qualquer predicado determinado retornará *true*:

```
Stream.empty().allMatch(Objects::nonNull); // true
```

Este é um padrão sensato, pois não podemos encontrar nenhum elemento que não satisfaça o predicado.

Da mesma forma, o método *anyMatch()* sempre retorna *false* para fluxos vazios:

```
Stream.empty().anyMatch(Objects::nonNull); // false
```

Novamente, isso é razoável, pois não podemos encontrar um elemento que satisfaça essa condição.

3.5. Redução

A API de fluxo permite reduzir uma sequência de elementos a algum valor de acordo com uma função especificada com a ajuda do método *reduce()* do tipo *Stream*. Este método usa dois parâmetros: primeiro – valor inicial, segundo – uma função acumuladora.

Imagine que você tem um *Inteiro<Lista>* e deseja ter uma soma de todos esses elementos e algum *Inteiro* inicial (neste exemplo 23). Assim, você pode executar o seguinte código e o resultado será 26 (23 + 1 + 1 + 1).

```
List<Integer> integers = Arrays.asList(1, 1, 1);
Integer reduced = integers.stream().reduce(23, (a, b) -> a + b);
```

3.6. Coleta

A redução também pode ser fornecida pelo método *collect()* do tipo *Stream*. Essa operação é muito útil no caso de converter um fluxo em uma *Coleção* ou um *Mapa* e representar um fluxo na forma de uma única cadeia de caracteres. Existe uma classe de utilitários *Coletores* que fornecem uma solução para quase todas as operações típicas de coleta. Para algumas tarefas, não triviais, um *coletor* personalizado pode ser criado.

```
List<String> resultList  
    = list.stream().map(element ->  
    element.toUpperCase()).collect(Collectors.toList());
```

Esse código usa a operação *collect()* do terminal para reduzir um `Stream<String>` para `List<String>`.

Desde as primeiras versões, Java dispõe das estruturas de arrays e as classes `Vector` e `Hashtable`. No entanto, além da dificuldade em implementar estruturas de dados utilizando arrays, os desenvolvedores sentiam falta de classes que implementassem estruturas como listas ligadas e tabelas de espalhamento (hash). Para atender a essas necessidades, a partir de Java 1.2, foi criado um conjunto de interfaces e classes denominado Collections Framework, que faz parte do pacote `java.util`.

O que é Collections Framework?

Collections Framework é um conjunto bem definido de interfaces e classes para representar e tratar grupos de dados como uma única unidade, que pode ser chamada coleção, ou collection. A Collections Framework contém os seguintes elementos:

- **Interfaces:** tipos abstratos que representam as coleções. Permitem que coleções sejam manipuladas tendo como base o conceito “Programar para interfaces e não para implementações”, desde que o acesso aos objetos se restrinja apenas ao uso de métodos definidos nas interfaces;
- **Implementações:** são as implementações concretas das interfaces;
- **Algoritmos:** são os métodos que realizam as operações sobre os objetos das coleções, tais como busca e ordenação.

A **Figura 15** mostra a árvore da hierarquia de interfaces e classes da Java Collections Framework que são derivadas da interface `Collection`. O diagrama usa a notação da UML, onde as linhas cheias representam `extends` e as linhas pontilhadas representam `implements`.

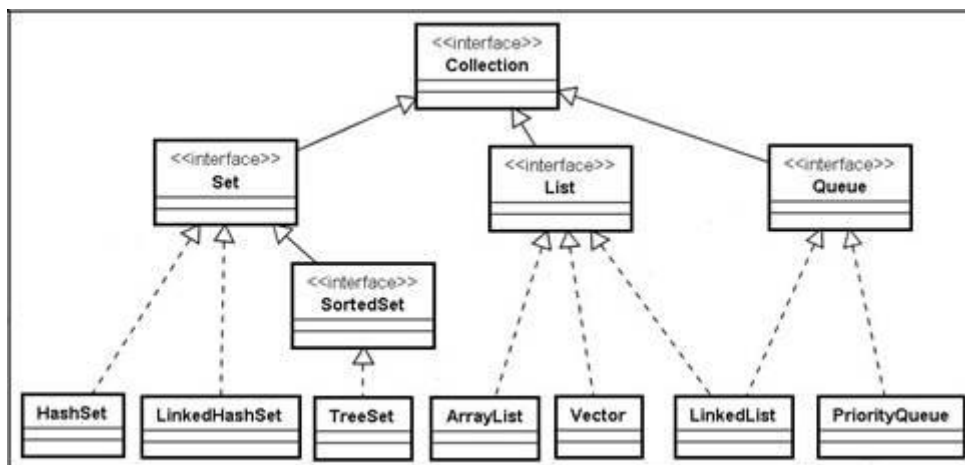


Figura 15 - A hierarquia de interfaces e classes

A hierarquia da Collections Framework tem uma segunda árvore. São as classes e interfaces relacionadas a mapas, que não são derivadas de `Collection`, como mostra a **Figura 16**. Essas interfaces, mesmo não sendo consideradas coleções, podem ser manipuladas como tal.

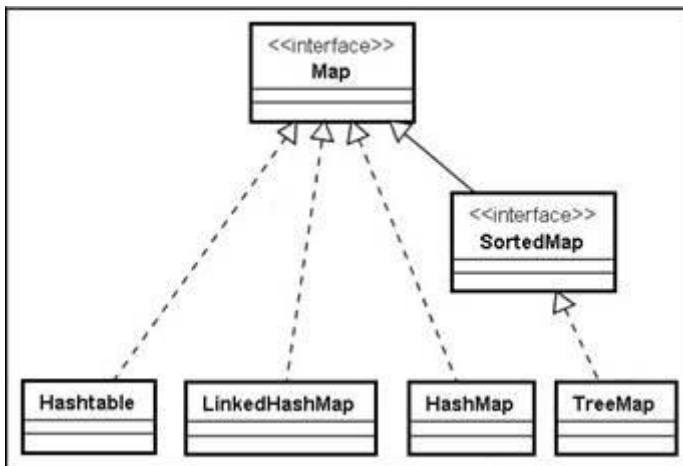


Figura 16 - Hierarquia de mapas

Interfaces

Neste momento vamos apresentar uma breve descrição de cada uma das interfaces da hierarquia:

- **Collection** – está no topo da hierarquia. Não existe implementação direta dessa interface, mas ela define as operações básicas para as coleções, como adicionar, remover, esvaziar, etc.;
- **Set** – interface que define uma coleção que não permite elementos duplicados. A interface **SortedSet**, que estende **Set**, possibilita a classificação natural dos elementos, tal como a ordem alfabética;
- **List** – define uma coleção ordenada, podendo conter elementos duplicados. Em geral, o usuário tem controle total sobre a posição onde cada elemento é inserido e pode recuperá-los através de seus índices. Prefira esta interface quando precisar de acesso aleatório, através do índice do elemento;
- **Queue** – um tipo de coleção para manter uma lista de prioridades, onde a ordem dos seus elementos, definida pela implementação de **Comparable** ou **Comparator**, determina essa prioridade. Com a interface fila pode-se criar filas e pilhas;
- **Map** – mapeia chaves para valores. Cada elemento tem na verdade dois objetos: uma chave e um valor. Valores podem ser duplicados, mas chaves não. **SortedMap** é uma interface que estende **Map**, e permite classificação ascendente das chaves. Uma aplicação dessa interface é a classe **Properties**, que é usada para persistir propriedades/configurações de um sistema, por exemplo.

Nota: Observe que usamos acima os termos **ordenação** e **classificação**. Dizemos que uma estrutura está ordenada se ela pode ser percorrida (iterada) em uma certa ordem, tal como os itens de um **ArrayList** podem ser percorridos através de seus índices. Por sua vez, a classificação diz respeito à ordenação na essência dos dados, tal como a classificação em ordem alfabética de **Strings** ou ordem numérica das classes **wrapper**, como **Integer** e **Double**, por exemplo. Podemos afirmar que uma estrutura classificada é uma estrutura ordenada, mas o inverso não é verdadeiro.

A API oferece também interfaces que permitem percorrer uma coleção derivada de Collection. Neste artigo falaremos de:

- **Iterator** – possibilita percorrer uma coleção e remover seus elementos;
- **ListIterator** – estende **Iterator** e suporta acesso bidirecional em uma lista, modificando e/ou removendo elementos.

Implementações

As interfaces apresentadas anteriormente possuem diversas implementações que são utilizadas para armazenar as coleções. Na **Tabela 1** estão resumidas as implementações mais comuns.

Tabela 1 -Implementações de uso geral

Implementações					
Interfaces	Tabela de Espalhamento	Array Redimensionável	Árvore	Lista Ligada	Tabela de Espalhamento + Lista Ligada
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Queue					
Map	HashMap		TreeMap		LinkedHashMap

A seguir apresentamos algumas características das implementações que podem ajudar a decidir qual delas utilizar em uma aplicação:

- **ArrayList** – é como um array cujo tamanho pode crescer. A busca de um elemento é rápida, mas inserções e exclusões não são. Podemos afirmar que as inserções e exclusões são lineares, o tempo cresce com o aumento do tamanho da estrutura. Esta implementação é preferível quando se deseja acesso mais rápido aos elementos. Por exemplo, se você quiser criar um catálogo com os livros de sua biblioteca pessoal e cada obra inserida receber um número sequencial (que será usado para acesso) a partir de zero;
- **LinkedList** – implementa uma lista ligada, ou seja, cada nó contém o dado e uma referência para o próximo nó. Ao contrário do **ArrayList**, a busca é linear e inserções e exclusões são rápidas. Portanto, prefira **LinkedList** quando a aplicação exigir grande quantidade de inserções e exclusões. Um pequeno sistema para gerenciar suas compras mensais de supermercado pode ser uma boa aplicação, dada a necessidade de constantes inclusões e exclusões de produtos;
- **HashSet** – o acesso aos dados é mais rápido que em um **TreeSet**, mas nada garante que os dados estarão ordenados. Escolha este conjunto quando a solução exigir elementos únicos e a ordem não for importante. Poderíamos usar esta implementação para criar um catálogo pessoal das canções da nossa discografia;
- **TreeSet** – os dados são classificados, mas o acesso é mais lento que em um **HashSet**. Se a necessidade for um conjunto com elementos não duplicados e acesso em ordem natural, prefira o **TreeSet**. É recomendado

utilizar esta coleção para as mesmas aplicações de `HashSet`, com a vantagem dos objetos estarem em ordem natural;

- `LinkedHashSet` – é derivada de `HashSet`, mas mantém uma lista duplamente ligada através de seus itens. Seus elementos são iterados na ordem em que foram inseridos. Opcionalmente é possível criar um `LinkedHashSet` que seja percorrido na ordem em que os elementos foram acessados na última iteração. Poderíamos usar esta implementação para registrar a chegada dos corredores de uma maratona;
- `HashMap` – baseada em tabela de espalhamento, permite chaves e valores `null`. Não existe garantia que os dados ficarão ordenados. Escolha esta implementação se a ordenação não for importante e desejar uma estrutura onde seja necessário um ID (identificador). Um exemplo de aplicação é o catálogo da biblioteca pessoal, onde a chave poderia ser o ISBN (International Standard Book Number);
- `TreeMap` – implementa a interface `SortedMap`. Há garantia que o mapa estará classificado em ordem ascendente das chaves. Mas existe a opção de especificar uma ordem diferente. Use esta implementação para um mapa ordenado. Aplicação semelhante a `HashMap`, com a diferença que `TreeMap` perde no quesito desempenho;
- `LinkedHashMap` – mantém uma lista duplamente ligada através de seus itens. A ordem de iteração é a ordem em que as chaves são inseridas no mapa. Se for necessário um mapa onde os elementos são iterados na ordem em que foram inseridos, use esta implementação. O registro dos corredores de uma maratona, onde a chave seria o número que cada um recebe no ato da inscrição, é um exemplo de aplicação desta coleção.

Cada uma das implementações tem todos os métodos definidos em suas interfaces. Em qualquer uma delas é possível inserir elementos `null`. Em mapas, tanto chaves quanto valores podem ser `null`. Diferente de `Vector` e `Hashtable`, não são seguras para serem usadas com threads (não são Thread-safe). Ou seja, o acesso concorrente a esses objetos pode produzir resultados imprevisíveis. Além disso, são serializáveis – isto é, seus estados podem ser salvos – e suportam o método `clone()`, que cria uma cópia de um objeto.

Nota: Thread-safe é o termo designado a objetos seguros para serem usados com threads.

Seguindo as boas práticas de orientação a objetos, você deve programar para interfaces e não para implementações. A recomendação é escolher uma implementação para instanciar o objeto e atribuir a nova coleção ao tipo de interface correspondente. Ou ainda, passar o objeto coleção para um método que espera um argumento do tipo interface. Seguindo essas práticas você conseguirá o que chamamos de baixo acoplamento, ou seja, poderá mudar facilmente de implementação sem que isso acarrete alteração no código da aplicação. Desta forma você fica livre para mudar a implementação sempre que questões relacionadas a desempenho ou detalhes de comportamento exigirem a mudança.

Nota: Não confunda a interface `Collection` com a classe `Collections`. Essa classe oferece métodos estáticos utilitários que podem manipular coleções. Outra classe utilitária é `Arrays`, cujos métodos estáticos são aplicados a arrays.

Após uma visão geral da Collections Framework, vamos por as mãos na massa e desenvolver uma aplicação exemplo a partir de situações do dia-a-dia, nas quais são analisadas as necessidades apresentadas pelo problema para decidir a interface a ser utilizada.

Como aplicar adequadamente a Collections Framework

Vimos então que temos sete interfaces: `Collection`, `List`, `Set`, `SortedSet`, `Map`, `SortedMap` e `Queue`. A pergunta que geralmente se faz é: Qual delas usar? Para selecionar adequadamente uma interface devemos analisar o problema e verificar como ele se enquadra nas características de cada interface. Somente após isso devemos decidir.

Iniciaremos com um problema simples. Queremos manter uma lista de nomes de alunos de uma escola que oferece cursos de Informática básica. Essa lista será percorrida na ordem em que os elementos são inseridos. Além disso, queremos poder acessar um nome de aluno aleatoriamente.

Analisando os requisitos do problema (lista na ordem de inserção e recuperação aleatória) e as características das interfaces disponíveis, optamos por utilizar `List`. Elementos não duplicados é um requisito que pode ser inferido na descrição do problema e isso poderia nos levar a escolher `Set`, mas vamos manter nossa decisão inicial por questões didáticas.

A interface List

`List` tem duas implementações – `ArrayList` e `LinkedList`. `ArrayList` oferece acesso aleatório rápido através do índice. Já em `LinkedList` o acesso aleatório é lento e necessita de um objeto nó para cada elemento, que é composto pelo dado propriamente dito e uma referência para o próximo nó, ou seja, consome mais memória. Além dessas considerações, se for necessário inserir elementos no início e deletar elementos no interior da lista, a melhor opção poderia ser `LinkedList`. Para apoiar a decisão de usar uma ou outra implementação é melhor fazer testes de desempenho. Um teste simples é mostrado na **Listagem 10**. Execute o programa e anote o tempo. Substitua `ArrayList` por `LinkedList` e repita o teste. Ao final escolha a implementação mais eficiente.

Optamos então por usar `ArrayList`. Uma implementação básica pode ser vista na **Listagem 11**. Esta aplicação instancia um `ArrayList` e o atribui a uma referência do tipo `List`. Insere alguns nomes de alunos com o método `add()` e finalmente imprime a lista – as implementações de coleções sobrescrevem o método `toString()`, por isso podemos imprimir a lista passando apenas a referência para o método `println()`.


```
import java.util.*;
public class Teste {

    public static void main(String[] args) {
        final int MAX = 20000;
        long tInicio = System.currentTimeMillis();
        List<Integer> lista = new ArrayList<Integer>();
        for (int i = 0; i < MAX; i++) {
            lista.add(i);
        }

        for (int i = 0; i < MAX; i++) {
            lista.contains(i);
        }
        long tFim = System.currentTimeMillis();
        System.out.println("Tempo total: " + (tFim - tInicio));
    }
}
```

```
import java.util.*;
public class ListaAluno {

    public static void main(String[] args) {
        List<String> lista = new ArrayList<String>();
        lista.add("João da Silva");
        lista.add("Antonio Sousa");
        lista.add("Lúcia Ferreira");
        System.out.println(lista);
    }
}
```

Neste exemplo, a primeira consideração a fazer é que, tendo em mente a programação para interface, na declaração de `lista` foi usado o tipo `List`. Portanto, se decidirmos mudar a implementação para `LinkedList`, é necessário apenas substituir o tipo `ArrayList`.

A segunda consideração refere-se ao tipo de dado que uma lista pode adicionar. Normalmente é possível inserir qualquer `Object` em uma lista, ou seja, assim como poderíamos inserir uma `String`, poderíamos inserir `Aluno`, `Integer`, etc. Se a lista permite inserir `Object`, na hora de recuperar esses dados, é necessário fazer `cast` para o tipo desejado. Além disso, não se teria certeza do tipo de dado que foi inserido, e o `cast` poderia causar uma exceção. A partir de Java 5 foi introduzido o conceito de Generics, que nos permite escrever código reusável para qualquer tipo de objeto. Sob a ótica da utilização deste conceito em coleções, para definirmos o tipo que lista poderá adicionar, incluímos o parâmetro em sua declaração. Dessa forma o compilador gerará um erro caso se tente adicionar um objeto que não seja `String`. E não será necessário usar `cast` durante a iteração.

Adicionando novo requisito – Ordem ascendente

Vamos supor agora que desejamos que a lista seja classificada em ordem ascendente. Observando a documentação da implementação `ArrayList`, verificamos que não existe um método de ordenação. Para solucionar este requisito, uma opção seria mudar nossa aplicação para utilizar a interface `Set`, onde os elementos estariam classificados pela ordem natural, no entanto a inserção de novos elementos seria mais lenta. Sendo assim, vamos utilizar a classe utilitária `Collections`. Esta classe dispõe do método `sort()`, que pode classificar uma interface `List` em ordem natural ou classificar de acordo com a implementação da interface `Comparator`, que logo estudaremos ainda neste artigo. A aplicação deste método pode ser constatada na **Listagem 12**, onde podemos observar que o método `sort()` altera a lista original.

Listagem 12 - Utilização do método `sort()` da classe `Collections`

```
import java.util.*;

public class ListaAluno {

    public static void main(String[] args) {
        List<String> lista = new ArrayList<String>();
        lista.add("João da Silva");
        lista.add("Antonio Sousa");
        lista.add("Lúcia Ferreira");
        System.out.println(lista);
        Collections.sort(lista);
        System.out.println(lista);
    }
}
```

Adicionando novo requisito – Novos dados

Considere agora que as necessidades da nossa aplicação foram modificadas e que, precisamos, além do nome do aluno, o nome do curso que ele está fazendo e a sua nota. Definimos então a classe `Aluno` de acordo com a **Listagem 13**, e modificamos a classe `ListaAluno` conforme a **Listagem 14**, de maneira que a lista possa adicionar objetos `Aluno` ao invés de `String`.

Listagem 13 - Classe Aluno

```
public class Aluno {
    private String nome;
    private String curso;
    double nota;

    Aluno(String nome, String curso, double nota) {
        this.nome = nome;
        this.curso = curso;
        this.nota = nota;
    }

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

    // Métodos getters e setters
}
```

Listagem 14 - Classe ListaAluno modificada

```
import java.util.*;

public class ListaAluno {

    public static void main(String[] args) {
        List<Aluno> lista = new ArrayList<Aluno>();

        Aluno a = new Aluno("João da Silva", "Linux básico", 0);
        Aluno b = new Aluno("Antonio Sousa", "OpenOffice", 0);
        Aluno c = new Aluno("Lúcia Ferreira", "Internet", 0);
        lista.add(a);
        lista.add(b);
        lista.add(c);
        System.out.println(lista);
    }
}
```

Classificação de objetos

Se incluirmos na **Listagem 14** a chamada ao método `sort()` veremos que o código não compila. O compilador retornará um erro informando que não encontrou o método `sort()`. Visto que apenas trocamos a classe `String` pela classe `Aluno`, parece razoável supor que o problema está na classe `Aluno`, e está correto.

A documentação da classe `Collections` nos informa que o método `sort()` aceita apenas listas cujos elementos sejam de tipos que implementem a interface `Comparable`, e `Aluno` não implementa `Comparable`.

Esta interface tem apenas um método a ser implementado, `compareTo()`. Sua implementação deve ser feita de forma a retornar um inteiro negativo, zero ou um inteiro positivo caso o objeto que execute o método seja menor, igual ou maior que o objeto passado como parâmetro. Cabe ao desenvolvedor decidir o critério que será adotado para comparar dois objetos.

Na classe `Aluno` consideraremos que a comparação entre dois objetos será determinada pela comparação entre seus nomes, que são do tipo `String`. Dessa forma a classe `Aluno` deve ser alterada para que fique de acordo com a **Listagem 15**.

Listagem 15 - A classe `Aluno` com implementação da interface `Comparable`

```
public class Aluno implements Comparable<Aluno>{
    private String nome;
    private String curso;
    double nota;

    Aluno(String nome, String curso, double nota) {
        this.nome = nome;
        this.curso = curso;
        this.nota = nota;
    }

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

    public int compareTo(Aluno aluno) {
        return this.nome.compareTo(aluno.getNome());
    }

    // Métodos getters e setters

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

Note que no método `compareTo()` fizemos simplesmente uma chamada ao mesmo método, só que para o atributo `nome`, que é do tipo `String`. `String` é uma classe comparável, isto é, já implementa `Comparable`.

Agora podemos incluir uma chamada ao método `sort()` na classe `ListaAluno`. A ordenação implementada por `Comparable` é chamada ordenação natural. Por exemplo, em uma `String` a ordenação natural é a ordem alfabética, em uma classe `wrapper` – `Integer`, `Float`, etc. – a ordenação natural é a ordem numérica.

Em certas situações precisamos de uma ordenação diferente da natural ou temos uma coleção de objetos de uma classe de terceiros que não é comparável, ou seja, não implementa `Comparable`. Nesses casos usamos a interface `Comparator`. Para implementar esta ordenação é necessário escrever uma classe que implementa essa interface, definindo como os objetos da lista

serão comparados. A interface possui apenas um método, `compare()`. Ele recebe dois objetos que são comparados e retorna um inteiro negativo, zero ou um inteiro positivo se o primeiro objeto é menor, igual ou maior que o segundo. Na **Listagem 16** temos um exemplo de implementação de `Comparator`.

Listagem 16 - Implementação da interface `Comparator`

```
import java.util.Comparator;

public class ComparaAluno implements Comparator<Aluno> {
    public int compare(Aluno a, Aluno b) {
        return a.getNome().compareTo(b.getNome());
    }
}
```

Para usar esta implementação chamamos o método sobrecarregado `sort()` da classe `Collections`. Ele recebe como argumentos a lista a ser ordenada e uma instância da implementação de `Comparator`, conforme a **Listagem 17**.

Listagem 17 - Uso de `Comparator` para ordenar a lista

```
import java.util.*;

public class ListaAluno {

    public static void main(String[] args) {
        List<Aluno> lista = new ArrayList<Aluno>();
        ComparaAluno ca = new ComparaAluno();

        Aluno a = new Aluno("João da Silva", "Linux básico", 0);
        Aluno b = new Aluno("Antonio Sousa", "OpenOffice", 0);
        Aluno c = new Aluno("Lúcia Ferreira", "Internet", 0);
        lista.add(a);
        lista.add(b);
        lista.add(c);
        System.out.println(lista);
        Collections.sort(lista, ca);
        System.out.println(lista);
    }
}
```

Nota: A classe `Arrays`, cujos métodos estáticos se aplicam a arrays, também tem os métodos `sort()` que precisam das implementações de `Comparable` e `Comparator`, semelhante ao que foi estudado anteriormente para a classe **`Collections`**

A interface `Iterator`

As interfaces que estendem `Collection` herdam o método `iterator()`. Quando este método é chamado por um `collection` ele retorna uma interface `Iterator`. Após essa chamada, usamos os métodos de `Iterator` para percorrer um `collection` do início ao fim e até remover seus elementos. Na **Listagem 18** é mostrada uma aplicação desta interface em `ListaAluno`.

Listagem 18. Utilização da interface Iterator para percorrer uma lista

```
import java.util.*;

public class ListaAluno {

    public static void main(String[] args) {
        List<Aluno> lista = new ArrayList<Aluno>();

        Aluno a = new Aluno("João da Silva", "Linux básico", 0);
        Aluno b = new Aluno("Antonio Sousa", "OpenOffice", 0);
        Aluno c = new Aluno("Lúcia Ferreira", "Internet", 0);
        Aluno d = new Aluno("Antonio Sousa", "OpenOffice", 0);
        lista.add(a);
        lista.add(b);
        lista.add(c);
        lista.add(d);
        System.out.println(lista);
        Aluno aluno;
        Iterator<Aluno> itr = lista.iterator();
        while (itr.hasNext()) {
            aluno = itr.next();
            System.out.println(aluno.getNome());
        }
    }
}
```

Observe que é necessário informar o tipo que será retornado pelo `Iterator`. O método `hasNext()` retorna `true` se houver elemento a ser lido, e o método `next()` retorna o objeto, de acordo com o tipo informado na declaração da interface. A partir de Java 5 foi introduzido o `enhanced-for`, que facilita muito a iteração sobre collections e arrays. Mostraremos uma aplicação desse comando quando falarmos de Map.

Nota: A interface `Iterator` pode ser usada também para percorrer um `Set`. O método `listIterator()` retorna uma interface `ListIterator`. Esta interface, além dos métodos `hasNext()` e `next()`, oferece o método `hasPrevious()`, que retorna `true` se existir um elemento anterior, e o método `previous()` que retorna o elemento anterior. Além desses, ela tem métodos para remover e adicionar novos elementos.

A interface Set

Uma das características de `List` é que ela permite elementos duplicados, o que não é desejável em nossa lista de alunos. Analisando as interfaces, concluímos que `Set` é o que realmente precisamos, pois não permite elementos duplicados. Como `HashSet` tem desempenho superior a `TreeSet`, optamos por esta implementação.

Dessa forma, pode-se observar na **Listagem 19** a classe `ListaAluno` modificada para usar `Set`. Note que forçamos a inserção de um objeto duplicado, mas quando executamos a aplicação constatamos que o objeto foi inserido. Se um

Set não permite elementos duplicados, onde está o erro? Como `HashSet` determina que dois objetos estão duplicados?

Listagem 19. ListaAluno usando a interface Set

```
import java.util.*;

public class ListaAluno {

    public static void main(String[] args) {
        Set<Aluno> conjunto = new HashSet<Aluno>();

        Aluno a = new Aluno("João da Silva", "Linux básico", 0);
        Aluno b = new Aluno("Antonio Sousa", "OpenOffice", 0);
        Aluno c = new Aluno("Lúcia Ferreira", "Internet", 0);
        Aluno d = new Aluno("Antonio Sousa", "OpenOffice", 0);
        conjunto.add(a);
        conjunto.add(b);
        conjunto.add(c);
        conjunto.add(d);
        System.out.println(conjunto);
    }
}
```

`HashSet` usa o código hash do objeto – dado pelo método `hashCode()` – para saber onde deve por e onde buscar o mesmo no conjunto (Set). Antes ele verifica se não existe outro objeto no Set com o mesmo código hash. Se não há código hash igual, então ele sabe que o objeto a ser inserido não está duplicado. Dessa forma, classes cujas instâncias são elementos de `HashSet` devem implementar o método `hashCode()`. Como consequência disso, a classe `Aluno`, no nosso exemplo, deve sobrescrever o método `hashCode()`.

Conforme o contrato geral de `hashCode()`, que consta na especificação da classe `Object`, se dois objetos são diferentes de acordo com `equals()` então não é obrigatório que seus códigos `hash` sejam diferentes.

Portanto, objetos que retornam o mesmo código hash não são necessariamente iguais. Assim, quando encontra no conjunto um objeto com o mesmo código hash do objeto a ser inserido, `HashSet` faz uma chamada ao método `equals()` para verificar se os dois objetos são iguais. Dessa forma, a classe `Aluno` deve sobrescrever o método `equals()` também. Veja a classe `Aluno` com esses métodos implementados na **Listagem 20**.

Criar código para `equals()` e `hashCode()` não é trivial, pois existem contratos definidos pela API de Java que devem ser rigorosamente seguidos. Por exemplo: se dois objetos são iguais, eles devem permanecer iguais durante toda a aplicação e devem resultar no mesmo `hashCode()`. Para facilitar essa tarefa, Eclipse e NetBeans têm opções para gerar esses métodos para as classes.

Listagem 20. Implementação de equals() e hashCode() na classe Aluno

```
public class Aluno implements Comparable<Aluno>{
    private String nome;
    private String curso;
    double nota;

    Aluno(String nome, String curso, double nota) {
        this.nome = nome;
        this.curso = curso;
        this.nota = nota;
    }

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

    public int compareTo(Aluno aluno) {
        return this.nome.compareTo(aluno.getNome());
    }

    public boolean equals(Object o) {
        Aluno a = (Aluno) o;
        return this.nome.equals(a.getNome());
    }

    public int hashCode() {
        return this.nome.hashCode();
    }

    // Métodos getters e setters
    public String getNome() {
        return this.nome;
    }
}
```

A implementação de `hashCode()` e `equals()` foi simplificada devido a questões didáticas. Definimos que um aluno terá o código hash igual ao hash do seu nome – que é `String`. Sendo assim, precisamos apenas retornar o código hash do nome do aluno no método `hashCode()`. Ficou definido também que dois alunos são iguais quando têm nomes iguais, por isso no método `equals()` é retornada a comparação entre os nomes de dois alunos.

Agora podemos executar o programa da Listagem 10 e verificar que o objeto duplicado não foi inserido, tal como desejávamos. O método `add()` de `Set` retorna `true` ou `false` para indicar se o objeto foi inserido ou não no `collection`. Se for necessário, verifique o retorno do método para ter garantia da inclusão do objeto.

Se você programar pensando na interface e precisar de um conjunto (`Set`) classificado, use `TreeSet` em vez de `HashSet` sem necessidade de alterar o restante do código, pois tanto `TreeSet` como `HashSet` implementam exatamente os mesmos métodos de `Set`. No entanto, vale ressaltar que a classe dos elementos que são adicionados ao `TreeSet` deve implementar `Comparable`. Como `Aluno` já implementa esta interface não precisamos nos preocupar com isso.

Nota: Todas as classes em Java são derivadas de `Object`, herdando assim métodos que, por padrão, devem ser sobrescritos, tais

como: `clone()`, `equals()`, `hashCode()`, `toString()`, entre outros. Por padrão, o método `equals()` usa `==` para verificar se duas referências são iguais, enquanto `hashCode()` retorna um inteiro calculado a partir do endereço do objeto. Classes Java como `String` e `Date` já sobreescrevem tais métodos.

A interface Map

Vamos supor que agora queremos uma estrutura onde possamos recuperar os dados de um aluno passando apenas o seu nome como argumento de um método. Ou seja, informamos o nome do aluno e o objeto correspondente a esse nome é devolvido. Para isso vamos usar a interface `Map`, que não estende `Collection`. Isso causa uma mudança profunda na aplicação, visto que os métodos usados anteriormente não poderão ser usados. `Map` tem seus próprios métodos para inserir/buscar/remover elementos na estrutura.

Esta interface mapeia chaves para valores. Considerando a nova proposta do problema, a chave será o nome do aluno e o valor será o objeto aluno.

Para usar uma classe que implementa `Map`, quaisquer classes que forem utilizadas como chave devem sobreescrever os métodos `hashCode()` e `equals()`. Isso é necessário porque em um `Map` as chaves não podem ser duplicadas, apesar dos valores poderem ser. Para a implementação mostrada na **Listagem 21**, utilizamos um `TreeMap`, que garante que as chaves estarão em ordem ascendente.

Listagem 21. Implementação de estrutura usando Map

```
import java.util.*;

public class MapaAluno {

    public static void main(String[] args) {
        Map<String, Aluno> mapa = new TreeMap<String, Aluno>();

        Aluno a = new Aluno("João da Silva", "Linux básico", 0);
        Aluno b = new Aluno("Antonio Sousa", "OpenOffice", 0);
        Aluno c = new Aluno("Lúcia Ferreira", "Internet", 0);
        Aluno d = new Aluno("Benedito Silva", "OpenOffice", 0);
        mapa.put("João da Silva", a);
        mapa.put("Antonio Sousa", b);
        mapa.put("Lúcia Ferreira", c);
        mapa.put("Benedito Silva", d);
        System.out.println(mapa);
        System.out.println(mapa.get("Lúcia Ferreira"));

        Collection<Aluno> alunos = mapa.values();
        for (Aluno e : alunos) {
            System.out.println(e);
        }
    }
}
```

Note que na declaração do collection informamos dois tipos: String e Aluno. O primeiro refere-se à chave e o segundo ao valor. O método para inserir na estrutura é `put()`, que recebe dois objetos (chave e valor). Para recuperar um objeto específico utilizamos o método `get()` passando a chave como parâmetro.

Como Map não estende Collection, não tem os métodos `iterator()` e `listIterator()`. Entretanto, existe o método `keySet()` que retorna um Set com as chaves do mapa, e o método `values()` que retorna um Collection com os valores associados às chaves. Assim, podemos percorrer o mapa partindo desses métodos e usando enhanced-for. A aplicação deste comando (for (Objeto obj: colecao) { ... }) para percorrer o mapa também é mostrada na **Listagem 21**.

Com tudo o que foi apresentado, podemos constatar que não existe a melhor implementação que resolve todos os problemas de estruturas de dados. Cada tipo de problema requer uma implementação diferente dependendo das características do mesmo. Escolher a implementação certa envolve saber o que sua interface oferece, quais as suas características e como ela será usada.

Conclusões

Java Collections Framework tem muito mais recursos do que aqueles que apresentamos neste artigo. É fundamental estudarmos a documentação da API para nos familiarizarmos com as opções que esta estrutura oferece. Falamos no texto que as interfaces não são thread-safe, no entanto a classe Collections possui um método `synchronized` para cada collection. Este método retorna objetos thread-safe, para o caso de você necessitar de acesso concorrente. O conhecimento dos contratos de `equals()` e `hashCode()` é muito importante para a utilização adequada das interfaces aqui estudadas. A implementação errada desses métodos pode produzir resultados inesperados e errôneos.

Além das interfaces apresentadas, existem outras, tais como `NavigableSet`, `BlockingQueue`, `Deque`, `BlockingDeque`, `NavigableMap`, etc. É muito importante consultar sempre a documentação de Java SE para usar com eficiência a API.

Entender as características de cada interface e implementação fornece a base para a decisão de qual delas utilizar, visando solucionar, da melhor maneira possível, os problemas apresentados durante o desenvolvimento de aplicações.

Java - Casting de tipos primitivos

Casting

Na linguagem Java, é possível se atribuir o valor de um tipo de variável a outro tipo de variável, porém para tal é necessário que esta operação seja apontada ao compilador. A este apontamento damos o nome de *casting*.

É possível fazer conversões de tipos de ponto flutuante para inteiros, e inclusive entre o tipo caractere, porém estas conversões podem ocasionar a perda de valores, quando se molda um tipo de maior tamanho, como um double dentro de um int.

O tipo de dado boolean é o único tipo primitivo que não suporta casting.

Segue abaixo uma tabela com todos os tipos de casting possíveis:

DE \ PARA	byte	short	char	int	long	float	double
byte		Implícito	char	Implícito	Implícito	Implícito	Implícito
short	byte		char	Implícito	Implícito	Implícito	Implícito
char	byte	short		Implícito	Implícito	Implícito	Implícito
int	byte	short	char		Implícito	Implícito	Implícito
long	byte	short	char	int		Implícito	Implícito
float	byte	short	char	int	long		Implícito
double	byte	short	char	int	long	float	

Para fazer um casting, basta sinalizar o tipo para o qual se deseja converter entre parênteses, da seguinte forma:

```
//Conversão do double 5.0 para float.
float a = (float) 5.0;

//Conversão de double para int.
int b = (int) 5.1987;

//Conversão de int para float é implícito, não precisa de casting.
float c = 100;

//Conversão de char para int é implícito, não precisa de casting.
int d = 'd';
```

O casting ocorre implicitamente quando adiciona uma variável de um tipo menor que o tipo que receberá esse valor.

Exemplo:

```
/**
 * Exemplo de conversão de tipos primitivos utilizando casting.
 */
public class ExemploCasting {
    public static void main(String[] args) {
        /* Casting feito implicitamente, pois o valor possui um
           tamanho menor que o tipo da variavel que irá recebe-lo. */
        char a = 'a';
        int b = 'b';
        float c = 100;

        System.out.println(a); //Imprime a
        System.out.println(b); //Imprime 98
        System.out.println(c); //Imprime 100.0

        /* Casting feito explicitamente, pois o valor possui um tamanho
           maior que o tipo da variavel que irá recebe-lo. */
        int d = (int) 5.1987;
        float e = (float) 5.0;
        int f = (char) (a + 5);
        char g = (char) 110.5;

        System.out.println(d); //Imprime 5
        System.out.println(e); //Imprime 5.0
        System.out.println(f); //Imprime 102
        System.out.println(g); //Imprime n
    }
}
```

Quando executamos a classe **ExemploCasting** temos a seguinte saída no console:

```
C:\>javac ExemploCasting.java
C:\>java ExemploCasting
a
98
100.0
5
5.0
102
n
```

Como usar funções lambda em Java

Aprenda nesse artigo a utilizar as funções lambda em Java, conceito esse adicionado ao Java 8, e que tem como principal objetivo adicionar ao Java técnicas de linguagens funcionais, como Scala e LISP. A grande vantagem de funções lambda é diminuir a quantidade de código necessária para a escrita de algumas funções, como por exemplo, as classes internas que são necessárias em diversas partes da linguagem Java, como Listeners e Threads.

Simplificando um pouco a definição, uma função lambda é uma função sem declaração, isto é, não é necessário colocar um nome, um tipo de retorno e o modificador de acesso. A ideia é que o método seja declarado no mesmo lugar em que será usado. As funções lambda em Java tem a sintaxe definida como (argumento) -> (corpo), como mostram alguns exemplos da **Listagem 22**.

```
(int a, int b) -> { return a + b; }  
() -> System.out.println("Hello World");  
(String s) -> { System.out.println(s); }  
() -> 42  
() -> { return 3.1415 };  
a -> a > 10
```

Listagem 22. Exemplos de funções lambda em Java

Uma função lambda pode ter nenhum ou vários parâmetros e seus tipos podem ser colocados ou podem ser omitidos, dessa forma, eles são inferidos pelo Java (veremos alguns exemplos disso mais para frente). A função lambda pode ter nenhum ou vários comandos: se a mesma tiver apenas um comando as chaves, não são obrigatórias e a função retorna o valor calculado na expressão; se a função tiver vários comandos, é necessário colocar as chaves e também o comando return - se nada for retornado, a função tem um retorno void.

Para demonstrar como utilizar as funções lambda em Java, vamos analisar alguns casos. O primeiro exemplo será a utilização com threads, onde elas são muito utilizadas para simplificar o código. O segundo exemplo será a utilização com as classes de coleção do Java, para aumentar a flexibilidade de diversas funções, como ordenar e filtrar listas. O terceiro exemplo será a utilização com classes do tipo Listeners, onde as funções lambda são utilizadas para simplificar o código. E finalmente, o quarto exemplo será para a criação de funções genéricas, que aceitam expressões lambda como parâmetros, e podem ser utilizadas em diversos problemas.

Exemplo 1 – Funções Lambda com Threads

Para exemplificar a utilização de expressões lambda com threads será analisado um programa que cria uma thread com uma função interna e que vai apenas mostrar a mensagem “Thread com classe interna!”. A **Listagem 23** mostra o código dessa implementação.

```
Runnable r = new Runnable() {  
    public void run() {  
        System.out.println("Thread com classe interna!");  
    }  
};  
new Thread(r).start();
```

Listagem 23. Thread criado com uma classe interna

Primeiro é criada uma implementação do método run da interface Runnable, e em seguida é criada a Thread com essa implementação. É possível verificar a grande quantidade de código necessário para um exemplo bastante simples.

Já com a utilização de expressões lambda, o código necessário para a implementação dessa mesma funcionalidade é bastante simples e bem menor que o anterior. A **Listagem 24** mostra um exemplo do código da expressão lambda com threads.

```
Runnable r = () -> System.out.println("Thread com função lambda!");  
new Thread(r).start();
```

Listagem 24. Thread com funções lambda

Essa expressão não passa nenhum parâmetro, pois ela será passada para a função run, definida na interface Runnable, que não tem nenhum parâmetro, então ela também não tem nenhum retorno.

Um código ainda mais simples é a passagem da função diretamente como parâmetro para o construtor da classe Thread. A **Listagem 25** mostra um exemplo desse código, mostrando que as funções lambda podem ser definidas e passadas como parâmetros diretamente para outros métodos, e isso pode ser bastante útil, como veremos nos próximos exemplos.

```
new Thread(  
    () -> System.out.println("hello world")  
).start();
```

Listagem 25. Função lambda passada como parâmetro para um método

Exemplo 2 – Funções Lambda com as classes de Collections

As funções lambdas podem ser bastante utilizadas com as classes de coleções do Java, pois nessas fazemos diversos tipos de funções que consistem

basicamente em percorrer a coleção e fazer uma determinada ação, como por exemplo, imprimir todos os elementos da coleção, filtrar elementos da lista e buscar um determinado valor na lista.

A **Listagem 26** mostra um exemplo de como normalmente é feito o código para percorrer uma lista e imprimir os valores dentro dela.

```
System.out.println("Imprime todos os elementos da lista!");  
List<Integer> list = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9);  
for(Integer n: list) {  
    System.out.println(n);  
}
```

Listagem 26. Imprimindo os elementos de uma lista

Com as funções lambda é possível implementar a mesma funcionalidade com muito menos código, bastando chamar o método `forEach` de uma lista, que é um método que espera uma função lambda como parâmetro. Esse método executará, a cada iteração na lista, a função passada como parâmetro. A **Listagem 27** mostra o exemplo de imprimir todos os elementos de uma lista com expressão lambda.

```
System.out.println("Imprime todos os elementos da lista!");  
List<Integer> list = Arrays.asList(1, 2, 3, 4, 5, 6, 7);  
list.forEach(n -> System.out.println(n));
```

Listagem 27. Imprimindo os elementos de uma lista com expressões lambda

Dentro do código de uma função lambda é possível executar diversos comandos, como por exemplo, na **Listagem 28**, que antes de imprimir o número, verifica se ele é par ou ímpar: se for par o número é impresso, caso contrário, nada é realizado. Nesse exemplo é possível verificar que dentro de uma expressão lambda pode ser realizado qualquer tipo de operação.

```
System.out.println("Imprime todos os elementos pares da lista!");  
List<Integer> list = Arrays.asList(1, 2, 3, 4, 5, 6, 7);  
list.forEach(n -> {  
    if (n % 2 == 0) {  
        System.out.println(n);  
    }  
});
```

Listagem 28. Imprimindo apenas os elementos pares de uma lista

Mais um exemplo do que pode ser feito com funções lambda, são expressões matemáticas. Na **Listagem 29** é exibido o código para mostrar o quadrado de todos os elementos de uma lista de números inteiros.

```
System.out.println("Imprime o quadrado de todos os elementos da lista!");
List<Integer> list = Arrays.asList(1, 2, 3, 4, 5, 6, 7);
list.forEach(n -> System.out.println(n * n));
```

Listagem 29. Imprimindo o quadrado dos elementos da lista

Funções lambda podem ser utilizadas também para a ordenação de listas com a interface `Comparator`. Por exemplo, caso exista uma classe `Pessoa` com os atributos `nome` e `idade` e é necessário ordenar uma lista em ordem alfabética pelo nome, ou em ordem das idades, é necessário implementar dois comparators, um para cada tipo de parâmetro, e chamá-lo no método `sort` da lista que será ordenada. A **Listagem 30** mostra o código do exemplo descrito utilizando a interface `Comparator` tradicional. O código da classe `Pessoa` foi omitido, mas ela é uma classe bastante simples, apenas com os atributos `nome` e `idade`, o construtor e os métodos `get` e `set`.

```
System.out.println("Ordenando pessoas pelo nome:");
List<Pessoa> listPessoas = Arrays.asList(new Pessoa("Eduardo", 29),
new Pessoa("Luiz", 32), new Pessoa("Bruna", 26));
Collections.sort(listPessoas, new Comparator<Pessoa>() {
    @Override
    public int compare(Pessoa pessoa1, Pessoa pessoa2){
        return pessoa1.getNome().compareTo(pessoa2.getNome());
    }
});
listPessoas.forEach(p -> System.out.println(p.getNome()));

System.out.println("Ordenando pessoas pela idade:");
Collections.sort(listPessoas, new Comparator<Pessoa>() {
    @Override
    public int compare(Pessoa pessoa1, Pessoa pessoa2){
        return pessoa1.getIdade().compareTo(pessoa2.getIdade());
    }
});
listPessoas.forEach(p -> System.out.println(p.getNome()));
```

Listagem 30. Ordenando objetos de uma classe pessoa utilizando comparators

É fácil observar que o código, apesar de bastante simples, ficou muito grande, apenas para ordenar duas vezes a lista com dois parâmetros diferentes.

É possível reimplementar esse exemplo utilizando funções lambda e deixando o código muito mais conciso. A **Listagem 31** mostra a reimplementação da **Listagem 29**, mas agora utilizando expressões lambda.

```
List<Pessoa> listPessoas = Arrays.asList(new Pessoa("Eduardo", 29),
new Pessoa("Luiz", 32), new Pessoa("Bruna", 26));
System.out.println("Ordenando pessoas pelo nome:");
Collections.sort(listPessoas, (Pessoa pessoa1, Pessoa pessoa2)
-> pessoa1.getNome().compareTo(pessoa2.getNome()));
listPessoas.forEach(p -> System.out.println(p.getNome()));
System.out.println("Ordenando pessoas pela idade:");
Collections.sort(listPessoas, (Pessoa pessoa1, Pessoa pessoa2)
-> pessoa1.getIdade().compareTo(pessoa2.getIdade()));
listPessoas.forEach(p -> System.out.println(p.getNome()));
```

Listagem 31. Comparator com expressões lambda

Vejam que os dois exemplos implementam exatamente a mesma funcionalidade, mas o código utilizando expressões lambda tem apenas cinco linhas, enquanto que o código que não utiliza expressões lambda tem 15 linhas.

Com expressões lambda também é possível filtrar elementos de uma coleção de objetos criando para isso um stream de dados (também um novo conceito do Java 8). É chamado o método `filter` do stream e como parâmetro para esse método é passado uma função lambda que faz o filtro dos elementos desejados. A **Listagem 32** mostra dois exemplos de filtros sendo realizados também na listagem de pessoas utilizadas nos exemplos anteriores. O primeiro filtro é feito apenas para pessoas com mais de 30 anos e o segundo para apenas pessoas que tem o nome iniciado com a letra “E”.

```
System.out.println("Pessoas com mais de 30 anos:");
List<Pessoa> maioresTrinta = listPessoas.stream().filter(p
-> p.getIdade() > 30).collect(Collectors.toList());
maioresTrinta.forEach(p -> System.out.println(p.getNome()));

System.out.println("Pessoas que o nome iniciam com E:");
List<Pessoa> nomesIniciadosE = listPessoas.stream().filter(p
-> p.getNome().startsWith("E")).collect(Collectors.toList());
nomesIniciadosE.forEach(p -> System.out.println(p.getNome()));
```

Listagem 32. Filtros com Funções Lambda

Exemplo 3 – Funções Lambda utilizadas com Listeners

Listeners são classes que implementam o padrão de projeto Observer, que representa objetos que ficam esperando ações realizadas em outros objetos e, a partir dessa ação, executam algum código. Um exemplo bem comum são os Listeners de botões da API de interfaces gráficas Swing. Por exemplo, quando é necessário implementar um código para realizar alguma ação quando um usuário clica em um JButton, passamos um objeto do tipo ActionListener para o método `addActionListener` do botão e isso, normalmente, é implementado com classes internas. A **Listagem 33** exibe o código para a criação desse listener.

```
button.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        System.out.println("O botão foi pressionado!");
        //Realiza alguma ação quando o botão for pressionado
    }
});
```

Listagem 33. Listener sem utilizar funções lambda

Apesar de funcionar, o código exibido é muito grande, mas utilizando funções lambda é possível implementar a mesma funcionalidade com um código muito mais enxuto e simples de entender. A **Listagem 34** mostra como esse mesmo código seria implementado utilizando funções lambda.

```
button.addActionListener( (e) -> {
    System.out.println("O botão foi pressionado, e o código
    executado utiliza uma função lambda!");
    //Realiza alguma ação quando o botão for pressionado
});
```

Listagem 34. Listener utilizando funções lambda

Praticamente qualquer Listener pode ser escrito utilizando expressões lambda, como os para escrever arquivos em logs e para verificar se um atributo foi adicionado em uma sessão de um usuário em aplicação web.

Exemplos 4 – métodos que aceitam funções lambda como parâmetros

Além de escrever funções lambda, também é possível criar métodos que as recebam como parâmetro, o que é bastante útil e pode tornar um método bastante flexível. Por exemplo, podemos criar um método genérico para imprimir elementos de uma lista, mas passamos como parâmetro a função para a filtragem dos elementos dessa lista, assim, com apenas um método, e passando a função como parâmetro, é possível fazer a filtragem da lista de várias maneiras diferentes. A **Listagem 35** mostra um exemplo de como poderia ser feito isso.

```

package teste.devmedia;

import java.util.Arrays;
import java.util.List;
import java.util.function.IntFunction;
import java.util.function.Predicate;

public class Main {

    public static void main(String [] a) {

        System.out.println("Cria a lista com os elementos que serão realizadas operações");
        List<Integer> list = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12);

        System.out.println("Imprime todos os números:");
        avaliaExpressao(list, (n)->true);

        System.out.println("Não imprime nenhum número:");
        avaliaExpressao(list, (n)->false);

        System.out.println("Imprime apenas número pares:");
        avaliaExpressao(list, (n)-> n%2 == 0 );

        System.out.println("Imprime apenas números ímpares:");
        avaliaExpressao(list, (n)-> n%2 == 1 );

        System.out.println("Imprime apenas números maiores que 5:");
        avaliaExpressao(list, (n)-> n > 5 );

        System.out.println("Imprime apenas números maiores que 5 e menores que 10:");
        avaliaExpressao(list, (n)-> n > 5 && n < 10);

    }

    public static void avaliaExpressao(List<Integer> list, Predicate<Integer>
predicate) {
        list.forEach(n -> {
            if(predicate.test(n)) {
                System.out.println(n + " ");
            }
        });
    }
}

```

Listagem 35. Método para a filtragem de elementos de uma lista

O método `avaliaExpressao` recebe como parâmetro uma lista e um objeto do tipo `Predicate`, que é uma interface, que espera uma função lógica, isto é, que avalia uma expressão booleana, e retorna `true` ou `false`. Essa função é executada chamando o método `test`, que executará a função passada como parâmetro. Ao ser avaliada, se essa função retornar verdadeiro, imprime o valor da lista, caso contrário, não imprime nada.

A **Listagem 36** exibe um código parecido, mas ao invés de um `Predicate`, a função passada como parâmetro para o método é um `IntFunction`, que espera funções que são realizadas sobre números inteiros, como soma e multiplicação. No exemplo, é possível observar que dentro do método `realizaOperacao`, o objeto `function` chama o método `apply`, que executa a função lambda passada

como parâmetro e, assim como o método anterior, permite muita flexibilidade. Podemos fazer qualquer tipo de operação sobre uma lista de números inteiros.

```
package teste.devmedia;

import java.util.Arrays;
import java.util.List;
import java.util.function.IntFunction;
import java.util.function.Predicate;

public class Main {

    public static void main(String [] a) {

        System.out.println("Cria a lista com os elementos que serão realizadas operações");
        List<Integer> list = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12);
        System.out.println("Multiplica todos os elementos da lista por 5:");
        realizaOperacao(list, (n)-> n * 5);

        System.out.println("Calcula o quadrado de todos os elementos da lista:");
        realizaOperacao(list, (n)-> n * n);

        System.out.println("Soma 3 em todos os elementos da lista:");
        realizaOperacao(list, (n)-> n + 3);

        System.out.println("Coloca 0 em todos os elementos da lista:");
        realizaOperacao(list, (n)-> 0);

    }

    public static void realizaOperacao(List<Integer> list, IntFunction<Integer> function) {
        list.forEach(n -> {
            n = function.apply(n);
            System.out.println(n + " ");
        });
    }
}
```

Listagem 36. Método para realizar operações matemáticas em uma lista

Assim como o Predicate e o IntFunction, existem diversas outras interfaces que podem ser utilizadas para utilizar funções lambda. Todas elas têm o mesmo objetivo, que é receber uma função como parâmetro, mas a diferença entre essas interfaces são o tipo e o número de parâmetros de entrada esperados e o tipo de retorno da função.

As funções lambda são mecanismos bastante poderosos, que facilitam muito a escrita de código conciso e evitam que o programador seja obrigado a escrever um monte de código “inútil”, principalmente em operações simples, além de flexibilizar o mesmo.

Apesar de serem bastante úteis, as funções lambda nem sempre são a melhor opção, caso seja necessário reutilizar diversas vezes uma função, talvez seja melhor criar uma classe ou interface com apenas um método e não uma expressão lambda.

EXCEÇÕES E CONTROLE DE ERROS

"Quem pensa pouco erra muito."--Leonardo da Vinci

Ao final deste capítulo, você será capaz de:

- Controlar erros e tomar decisões com base neles;
- Criar novos tipos de erros para melhorar o tratamento deles em sua aplicação ou biblioteca;
- Assegurar que um método funcionou como diz em seu contrato.

Motivação

Em Java, os métodos dizem qual o **contrato** que eles devem seguir. Se, ao tentar sacar, ele não consegue fazer o que deveria, ele precisa, ao menos, avisar ao usuário que o saque não foi feito.

Veja no exemplo abaixo, estamos forçando uma `Conta` a ter um valor negativo, isto é, a estar em um estado inconsistente de acordo com a nossa modelagem:

```
Conta minhaConta = new Conta();
minhaConta.deposita(100);
minhaConta.setLimite(100);
minhaConta.saca(1000);
// o saldo é -900? É 100? É 0? A chamada ao método saca funcionou?
```

Em sistemas de verdade, é muito comum que quem saiba tratar o erro seja aquele que chamou o método, e não a própria classe! Portanto, nada mais natural do que a classe sinalizar que um erro ocorreu.

A solução mais simples utilizada antigamente é a de marcar o retorno de um método como `boolean` e retornar `true` se tudo ocorreu da maneira planejada, ou `false`, caso contrário:

```
boolean saca(double quantidade) {
    // posso sacar até saldo+limite
    if (quantidade > this.saldo + this.limite) {
        System.out.println("Não posso sacar fora do limite!");
        return false;
    } else {
        this.saldo = this.saldo - quantidade;
        return true;
    }
}
```

Um novo exemplo de chamada do método acima:

```
Conta minhaConta = new Conta();
minhaConta.deposita(100);
minhaConta.setLimite(100);
if (!minhaConta.saca(1000)) {
    System.out.println("Não saquei");
}
```