# Aula 1 – O que é Java?

Antes de mais nada, vamos ver um pouco do que é o Java, o qual te trouxe até aqui: há cerca de vinte anos, quando a linguagem Java nasceu, ela chamava a atenção por conta das seguintes características:

* Orientado a Objeto (O.O.)
* Muitas bibliotecas - api
* Parece com C++ (hoje em dia isso pode até ser uma desvantagem)
* Roda em vários sistemas operacionais

Você pode estar pensando "poxa, mas a linguagem que uso no dia a dia, atualmente, já possui estas características!". É verdade. É por isto que queremos focar na **plataforma Java**, e não especificamente na linguagem em si, algo que ficará mais claro no decorrer do curso, e até mesmo nesta aula!

A plataforma Java traz:

* Portabilidade
* Fácil acesso e desenvolvimento
* Segurança
* Onipresença

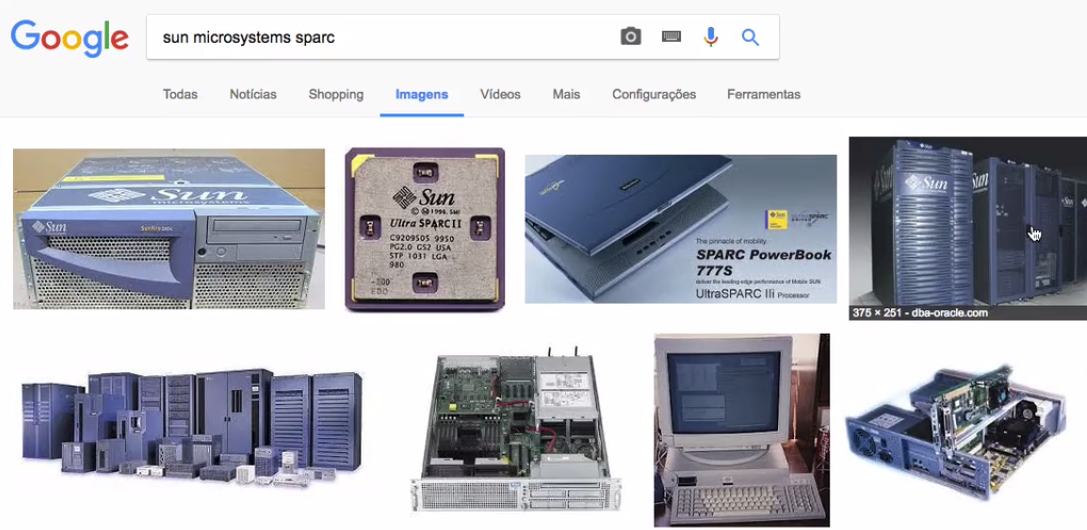
Você pode dar uma olhada no [site oficial](https://java.com/), porém ele ajuda mais o usuário do Java, do que aqueles que irão compilar e escrever programas.

Falando sobre a história da linguagem: James Gosling é considerado um dos "gênios da computação", sendo considerado o "pai do Java", apesar da linguagem ter sido criada por um grupo, normalmente considerado de quatro pessoas.

Em 1992, o James Gosling trabalhava em uma empresa atualmente inexistente chamada Sun Microsystems (sendo que Sun é acrônimo para *Stanford University Network*), uma dessas *startups* da década de 60, 70, para lidar mais com hardware, que é o que estava dando mais dinheiro.

Eles possuíam um microcomputador, o ***Sun Microsystems SPARC***, que hoje em dia já não aparecem em lugar algum, grandes servidores denominados "micro":

Eles possuíam um microcomputador, o ***Sun Microsystems SPARC***, que hoje em dia já não aparecem em lugar algum, grandes servidores denominados "micro":



Sendo a Sun uma empresa mais focada em hardware, naquela época, a IBM e a Microsoft começaram a crescer vendendo softwares. Os softwares que a Sun utilizava no sistema deles, o UNIX (o tal de Solaris), eram disponibilizados gratuitamente.

Um dia, esses executivos, dentre os quais o próprio James Gosling, se perguntaram como poderiam lucrar com softwares, já que eles o disponibilizavam de graça, e fizeram um retiro de um mês para tentarem chegar a uma conclusão.

A ideia que eles tiveram envolvia um problema de eletrônicos da década de 90: havia muitos deles sendo criados naquela época, como o VHS que, para quem não sabe, é o videocassete. Era a época de surgimento de TVs, videogames, liquidificadores e geladeira.

Cada um deles possui seu código fonte, necessitando de uma linguagem própria para funcionar, e escrever o código para cada um, reescrevendo-o quando tivessem que passar por uma troca de chip, por exemplo, não fazia muito sentido! A linguagem utilizada neles, ***Assembly***, que hoje em dia é raramente usada, precisava ser reescrito várias vezes, imagine o trabalho!

O James Gosling e sua equipe pensaram em escrever um único código que gerasse um "executável" - entre aspas porque após a compilação ele estará em um formato não exatamente compreensível pelo aparelho em si, mas por um intermediário, no caso, um processador ou uma placa de hardware, para que, aí sim, passe o código aos aparelhos.

Trata-se de algo que realmente simula um computador bem simples e traduz esta linguagem "executável" de acordo com o aparelho em questão. Isto é, esta "máquina de mentira" traduzirá tudo, como se fosse um sistema operacional.

É por isto que surgiu o nome **máquina virtual**, pois veio da ***virtual machine***!

A ideia deles foi, então, criar uma placa pequena, um hardware, que é uma máquina real e compõe todo liquidificador, computador, videocassete, e por aí vai. Desta forma, as pessoas poderão escrever em apenas uma linguagem, que na época se chamava ***Oak*** e depois se tornou Java.

Isso pareceu muito bom, mas acabou fracassando de maneira retumbante, pois era muito caro produzir chips distintos para cada aparelho, cada qual adaptado a uma determinada linguagem.

Então, em 1995, com o *boom* da Web e o surgimento de mais navegadores, como Mosaic, Netscape e posteriormente Internet Explorer, a ideia de máquina virtual foi visualizada como um problema interessante pelo Gosling.

Assim como na atualidade, existia uma variedade relevante de navegadores e sistemas operacionais. E, para escrever um código para Windows, utilizava-se a linguagem no Microsoft Visual Basic, que por sua vez era compilado por um executável (um EXE, no caso do Windows).

Isto é, ele só funciona neste sistema operacional, com determinadas DLLs na máquina, e assim por diante. O executável e o código fonte ficavam atrelados a uma plataforma específica, um conjunto de sistema operacional, hardware e outros detalhes.

Para tentar resolver este problema, que geraria um código e um executável diferentes para cada sistema operacional existente, o Gosling desengavetou a ideia da máquina de verdade, do chip, que eles haviam criado anteriormente.

Com um código fonte único, teríamos um intermediário que soubesse traduzir ou instruir o sistema operacional acerca dos comandos a serem enviados e recebidos. Este meio de campo seria realizado pela **Máquina Virtual Java (JVM)**, que não é meramente um interpretador por conta de alguns detalhes internos que vão além da interpretação.

O código, então, seria a linguagem Java, e o código "executável", quando compilado, não geraria um .exe (pois este seria lido apenas pelo Windows), e sim um formato chamado ***bytecode Java***, de extensão .class, lido pela Máquina Virtual Java, que passaria a informação aos sistemas operacionais.

Um exemplo deste formato entendido pela *virtual machine* (JVM), o *bytecode*, é o seguinte:

Compiled from "Onibus.java"

class Teste {

public static void main(java.lang.String);

Code:

0: new #2 // class Onibus

3: dup

4: invokespecial #3 Onibus."<init>":()V

7: astore\_1

8: aload\_1

9: ldc #4 // String Jabaquara...

11: putfield #5

// Field Onibus.linha:Ljava/lang/String;

14: return

}

Quem conhece a linguagem de ***Assembly*** talvez identifique a semelhança, mas este código não parece ser de fácil leitura e compreensão. Para meios de comparação, segue um exemplo de um arquivo .java, a ser compilado e traduzido para .class, o tal do bytecode:

public class Onibus {

String nome;

String linha;

}

class Teste {

public static void main(String args) {

Onibus o = new Onibus();

o.linha = "Jabaquara-Liberdade";

}

}

Então, em 1995 surgiu o Java, capaz de rodar em vários dispositivos e sistemas operacionais, com foco de criar *applets*, quando ainda tínhamos que instalar o Java para rodá-lo dentro do navegador.

O Java nasceu com um propósito, mas acabou se fortalecendo em ***server-side***, pois quando escrevemos uma aplicação, um site web ou sistema grande, não queremos ficar dependendo de diferentes sistemas operacionais, em implantações e *deploys*.

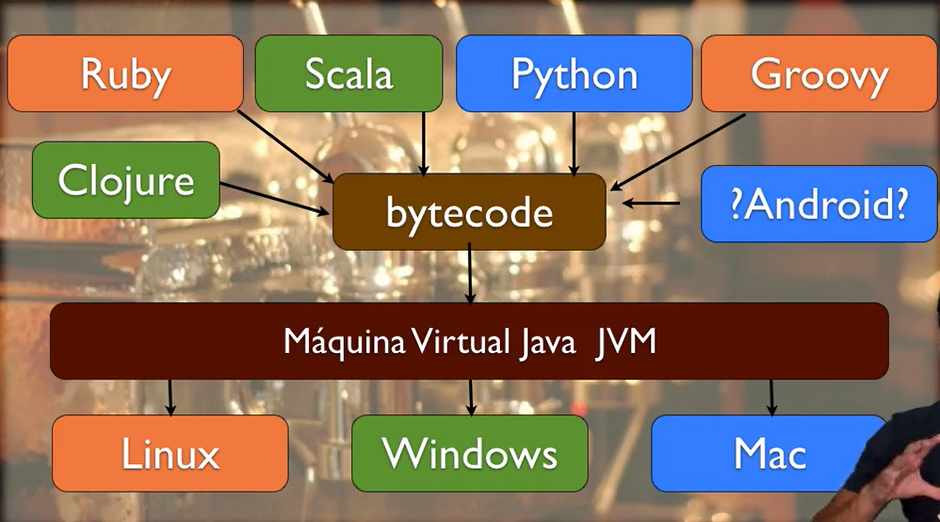
O Java traz liberdade, quebrando nossa dependência em relação às versões de sistema operacional e navegadores. Empresas grandes, como bancos e o governo, não querem ficar engessados - o que é conhecido por *Vendor lock-in*.

As principais características do conceito de Máquina Virtual Java são:

* Multiplataforma
* Gerenciamento de memória
* Segurança
* Sandbox
* Otimizações
* JIT Compiler

Hoje, mais do que na linguagem Java em si, o enfoque está na plataforma, no **ecossistema Java**! A *virtual machine* é interessante para as empresas pois elas não dependem do que se encontra abaixo da sua *stack*, ou pilha de tecnologia, além do acesso a uma grande variedade de bibliotecas, e as linguagens Java que rodam nesta plataforma.

Não é à toa que há programas que lidam com linguagens Ruby, Clojure ou Scala, por exemplo, e geram o bytecode Java. Depois, basta a Máquina Virtual Java, JVM, trabalhar de acordo com o sistema operacional desejado.



Já falamos um pouco sobre o Bytecode que é um código de máquina parecido com o Assembly. Talvez você (como eu!) estranhou o nome Bytecode, no entanto, tem uma explicação bem simples para tal. Existe um conjunto de comandos que a máquina virtual Java entende. Esses comandos também são chamados de opcodes (operation code), e cada opcode possui o tamanho de exatamente 1 Byte! E aí temos um **opcode de 1 Byte** ou, mais simples, **Bytecode**. :)

# Aula 2 – Instalação e o primeiro programa

Você deve estar muito ansioso para instalar o Java e o ambiente de programação para compilar e executar seu primeiro programa!

Mas antes de todo este processo de instalação e configuração, gostaria de falar sobre **versões**, uma vez que é comum encontrarmos vários números e versões e ficarmos perdidos sem saber por onde iniciar no Java.

Apesar da última versão lançada ser o 9, lançada em 2017, a linguagem, surgida em 1995, teve mudanças consideráveis na versão 5, que saiu em 2004, e na 8, de 2014. Nelas, apareceram muitos recursos na linguagem, novos comandos, palavras-chave e conceitos.

Estes tais de **Streams**, de **Templates Generics**, serão vistos durante o curso - há até um [curso específico sobre estes novos recursos do Java 8](https://cursos.alura.com.br/course/java8-lambdas). Nas versões 9 e 7, houve mudanças pequenas e pontuais, além de bibliotecas.

Então, não se preocupe, você pode, sim, focar na versão 8, pois você verá que muitas empresas grandes inclusive ainda não alcançaram esta versão (o que é uma pena).

Aqui, usaremos a versão **Neon** do **Eclipse**, mas existe uma versão mais recente, **Oxygen**, que está sendo trabalhada para dar suporte ao Java 9. Até o momento, não há versão oficial do Eclipse que dê suporte para a última versão disponível do Java.

Todos os conceitos focados neste curso, que envolvem Orientação a Objeto, uso da herança, polimorfismo e as principais bibliotecas, são os mesmos para muitas versões da linguagem.

Ou seja, a dica é focar naquilo que é importante, que é o que passaremos aqui, e não nas versões mais recentes. A versão 10, provavelmente virá com muito menos novidades, já que as versões seguirão a tendência de serem lançadas mais rapidamente, não de 3 em 3 anos, e sim de 6 em 6 meses.

Minha recomendação é a de que você siga os passos feitos neste curso, respeitando a instalação do Java 8 e do Eclipse Neon. No entanto, se você realmente quiser utilizar a versão mais recente de cada um deles, por sua própria conta e risco, vá em frente. É bem provável que você não encontre problemas!

Porém, se você é iniciante em programação e nunca viu Java antes, indica-se a utilização das versões citadas neste curso.

O mundo Java é cheio de siglas com 3 ou 4 letras começando com J. Você já conhece duas famosas: o **JRE** e **JDK**. O primeiro é o ambiente de execução, o segundo são as ferramentas de desenvolvimento junto com o ambiente de execução. Simplificando podemos dizer:

JDK = JRE + ferramentas desenvolvimento

Existe uma terceira sigla, **JVM** (Java Virtual Machine), que também já usamos durante o curso. A responsabilidade da Java Virtual Machine é executar o Bytecode! Então qual é diferença entre JVM e JRE? Ambos executam o Bytecode, certo?

A resposta é simples: O JRE (o nosso ambiente de execução) contém a JVM, mas também possui um monte de bibliotecas embutidas. Ou seja, para rodar uma aplicação Java não basta ter apenas a JVM, também é preciso ter as bibliotecas.

Assim podemos simplificar e dizer:

JRE = JVM + bibliotecas

É importante entender que você não pode baixar a JVM apenas. Você sempre baixa o JRE que tem a JVM e as bibliotecas em conjunto

## Compile e rode seu primeiro programa

Como falamos no início, em um primeiro contato, o código em Java pode ser complicado de ser escrito e compreendido. Às vezes precisamos escrever um pouco mais do que gostaríamos para fazer algo.

Antes de usarmos um IDE para lidarmos com o código, é legal que você o faça em um sistema bem simples, como o bloco de notas - outras opções são o TextPad, Atom, Visual Studio Code, Sublime, ou qualquer outro.

Nosso primeiro código Java será feito no editor de texto mais simples possível, em plain text. Faremos o "Olá mundo" para testarmos e vermos como funciona a compilação e execução de programas Java.

O Java veio da linguagem C na década de 90, então, não é tão simples quanto digitarmos print("olá mundo"). A linha que faz um print na tela, por exemplo, é

System.out.println("olá mundo");

Nesta linguagem, toda instrução que damos sem as chaves necessita do ponto e vírgula (**;**). Todo código Java também precisa estar dentro de uma classe, que pode ser uma interface, um Enum. Neste caso, ele se insere na classe Programa.

Uma instrução como esta, com System.out.println(), precisa estar dentro de um método chamado main, que ainda não vimos, acompanhado de outros termos que também aprenderemos depois.

É muito comum o uso de public antes de class Programa, e embora isto não seja estritamente necessário no nosso caso, vamos colocá-lo para quando formos ler códigos de outros programadores e IDEs.

public class Programa {

public static void main(String[] args) {

System.out.println("olá mundo");

}

}

No momento, focaremos na linha System.out.println("olá mundo");, que poderá ser considerado um comando apesar de não ser um, e mostrará algo na saída padrão, no caso o prompt do MS-DOS.

O menor programa Java seria similar ao código acima. Vamos tentar ver como funciona sua compilação e execução? Antes disso, salvaremos o arquivo nomeando-o com "Programa.java", em uma nova pasta denominada "java-codigo".

O nome do arquivo é muito importante - entenderemos melhor o motivo mais adiante, mas ele precisa ser o mesmo da class inserida no código.

No Prompt de Comando, digitaremos cd .. duas vezes, seguidos de "Enter", e dir, para a listagem de todos os diretórios. Depois, usaremos cd java-codigo para acessar o diretório, e em seguida digitaremos dir novamente.

Dica: é possível usar a tecla TAB para autocompletar palavras!

Ali, é listado um arquivo "Programa.java"! No Windows, há um comando chamado type (equivalente ao cat do terminal do Linux), o qual permite a visualização do conteúdo do arquivo. Neste caso, usaríamos type Programa.java.

A extensão .java não é entendida pela virtual machine, que entende o formato "meio máquina" de Virtual Machine Java, o **bytecode**, um arquivo com extensão .class.

A seguir, usaremos o comando javac Programa.java, e daremos um "Enter", com o qual serão mostradas as mensagens de erro de compilação, fundamentais para o aprendizado.

Apesar de não entendermos o que é public class ou static void main ainda, sabemos que System.out.println() seguido de aspas e o conteúdo, irá mostrar uma mensagem.

Por meio de dir no prompt, você verá que há dois arquivos: "Programa.java" e "Programa.class", este último no formato binário, em bytecode. E para chamarmos a virtual machine, usaremos o comando java Programa, e veremos a impressão de "olá mundo". Trata-se da primeira execução do nosso programa Java!

Agora, veremos os principais erros e características deste código. O primeiro surge ao digitarmos java Programa.class, o que traz a seguinte mensagem de erro na execução do programa:

Erro: Não foi possível localizar nem carregar a classe principal Programa.class

Isto acontece porque o programa não se chama "Programa.class", e sim simplesmente "Programa", apesar de estar contido no arquivo "Programa.class".

Outros erros mais comuns são os de compilação, como quando esquecemos de colocar o ponto e vírgula no fim da linha. Além disso, o Java possui palavras chave (keywords, ou palavras reservadas), dentre os quais utilizamos "public", "class", "static" e "void", que devem estar em letra minúscula, uma vez que o Java é **case sensitive** (reconhece o uso de letras maiúsculas ou minúsculas).

Em um ambiente mais complexo, veremos que isto ficará mais claro e fácil de ser trabalhado. É importante **praticar e não ter medo das mensagens de erro de compilação**.

As chaves abrem e fecham os blocos de códigos, indicando por exemplo que tudo aquilo que se encontra em public static void main pertence ao public class Programa, da mesma forma que System.out.println() pertence ao public static void mainvisível também por meio das indentações.

O Java possui outras particularidades, como o "Enter" e a barra de espaço serem opcionais; são convenções do código. Agora, o importante é escrever, entendendo o que está por trás do código, errar e fazer vários testes!

## Mão na massa: instalando o JDK

Vamos instalar o JDK! Para isso, siga os passos abaixo de acordo com seu sistema operacional. E, após instalado, prossiga para as configurações finais.

## Instalando o JDK no Linux

No Ubuntu, podemos executar o seguinte comando no terminal:

sudo apt-get install oracle-java9-installer

## Instalando o JDK no Mac

No Mac você pode baixar a versão mais atual do JDK [aqui!](http://www.oracle.com/technetwork/java/javase/downloads/jdk9-downloads-3848520.html)

Após a instalação no seu Mac, teste seu Java com os seguintes comandos no terminal:

javac -version

java -version

## Instalando o JDK no Windows

Por fim, para instalar no Windows, acesse [esse site](http://www.oracle.com/technetwork/java/javase/downloads/jdk9-downloads-3848520.html).

Escolha então o JDK (Java Development Kit) e, por fim, a versão do seu sistema operacional. Após baixado, execute o instalador.

### Configurações do PATH no Windows

1) Vá em painel de controle e procure por Sistema, após isso, clique em Configurações avançadas do sistema

2) Na aba "Avançado", clique em variáveis de ambiente e na janela "Variáveis do sistema" selecione o "Path" e clique em editar.

3) Nessa nova janela, clique no botão novo e nessa linha que foi selecionada (última), coloque o caminho para seu diretório bin dentro da pasta jdk, que por sua vez, está dentro da pasta java.

4) Salve a mudança clicando em OK e feche todas as janelas.

5) Após feito isso, feche seu prompt de comando e abra novamente.

Teste os seguintes comandos:

java -version

javac -version

## O que aprendemos?

Nessa aula você escreveu o seu primeiro código Java e aprendemos:

* qual é a diferença entre JRE e JDK
* como compilar um código fonte Java na linha de comando (javac)
* como executar o Bytecode na linha de comando (java)
* um programa Java deve estar escrito dentro de uma classe (class)
* toda instrução Java deve ser finalizada com ;
* para abrir e fechar um bloco usaremos as chaves {}
* um programa Java possui uma entrada que é uma função (método) main
* para imprimir algo no console usamos a instrução System.out.println()

# Aula 03 – Começando com Eclipse

Este capítulo apresentou:

* O papel de um IDE e sua diferença de um editor
* Como fazer o download do Eclipse IDE
* Para que serve um workspace
* O conceito de perspectiva
* Como criar um projeto Java, inclusive classes e como executá-lo.
* Como exibir diferentes views

# Aula 04 – Tipos e variáveis

[...]

No código, usamos algumas convenções: ao criarmos a classe TestaVariaveis, cuja funcionalidade ainda desconhecemos, usamos a primeira letra em maiúscula e, ao acrescentarmos a segunda palavra, não utilizamos underscore ou algo do tipo, e sim a primeira letra em caixa alta de novo. Isto se chama **Camel Case**, e aparece com frequência no Java e em muitas outras linguagens - é uma **convenção de código**, e seu uso não é obrigatório.

Da mesma forma, a variável iniciando-se com "i" minúsculo é o padrão, bem como não há o costume de se abreviar palavras. No Java, vocês verão nomes gigantescos de variáveis! É legal nos atentarmos a estas práticas para começarmos a nos acostumar com estes hábitos essenciais para quando formos trabalhar com grandes equipes.

[...]

No Java, entretanto, há uma regra: quando há uma divisão entre dois números inteiros, ele "forçará" um número inteiro como resultado. Se printarmos esta divisão, obteremos como resultado o valor 2. Estranho, não?

E se quiséssemos que o resultado fosse 2.5 de fato, poderíamos tentar double novaTentativa = 5 / 2;, e pediríamos sua impressão, que traria 2.0. Piorou! O Java irá ler apenas o lado que vem antes da atribuição na linha de código relativa ao double. Ou seja, primeiro, ele irá executar a divisão 5 / 2, e depois ele tentará colocar o resultado em um double.

Na verdade, o que gostaríamos é que a conta tivesse sido feita partindo-se do double e, neste caso, bastaria que um dos valores da divisão fosse deste tipo, como em 5.0 / 2. Desta forma, como trata-se de um double dividido por um int, a conta é feita levando-se em consideração o ponto flutuante.

Esta divisão, sim, trará 2.5 como resultado. Parece pegadinha, mas são características de linguagem que vão te deixar mais a par de como o Java funciona.

Para tentarmos fazer com que a parte do código comentada acima funcione, poderemos forçar a conversão, moldando um double para que ele se encaixe em um int.

É claro que não haverá encaixe perfeito, resultando em arestas que provavelmente serão perdidas. Faremos isso utilizando uma sintaxe comum a outras linguagens, o casting, para que o double seja transformado em um int.

public class TestaConversao {

public static void main(String[] args) {

double salario = 1270.50;

int valor = (int) salario;

System.out.println(valor);

}

}

Se printarmos valor, será mostrada apenas a parte inteira daquele número: 1270. É isso que chamamos de casting que, nestas variáveis que guardam números, não é algo muito complexo.

Mais adiante, veremos o casting de variáveis que são referência, e têm a ver com orientação a objetos, se são compilados ou não, se darão exceptions; é um mundo à parte.

# [Aula 05 - Trabalhando com caracteres](https://cursos.alura.com.br/course/java-primeiros-passos/section/5306/tasks)

# AULA06 -Praticando condicionais

Uma variável, a partir de sua declaração, passa a valer entre as chaves correspondentes, o que se denomina **escopo**.

**public** **boolean** transfere(**double** valor, Conta destino) {

**if** (**this**.saldo >= valor) {

**this**.saldo -= valor;

destino.deposita(valor);

System.***out***.println("Transferencia concluida para a conta: " + destino.numero);

**return** **true**;

} **else**

**return** **false**;

}

# Associação

**public** **class** Conta {

**double** saldo;

**int** agencia;

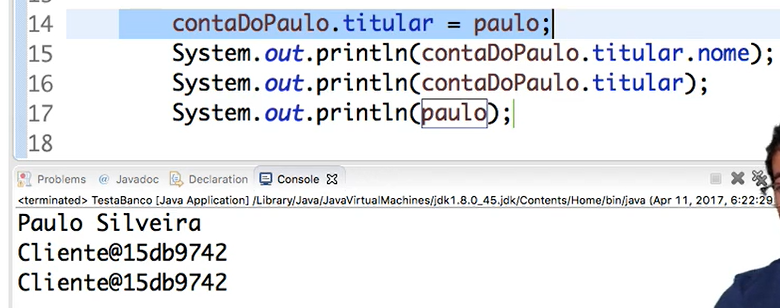
**int** numero;

Cliente titular;

contaDoPaulo.titular = paulo;

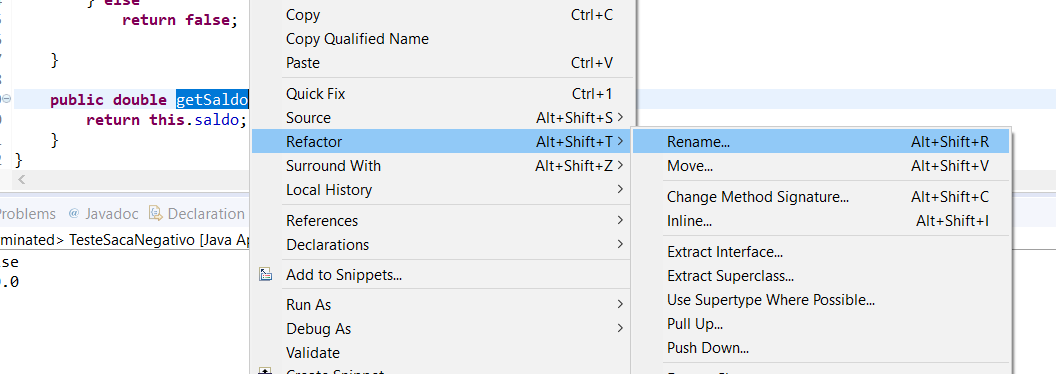
System.***out***.println(contaDoPaulo.titular.nome);

Aqui podemos ver a Referenciação. ContaDoPaulo.titular tem a mesma referencia que o objeto Paulo:



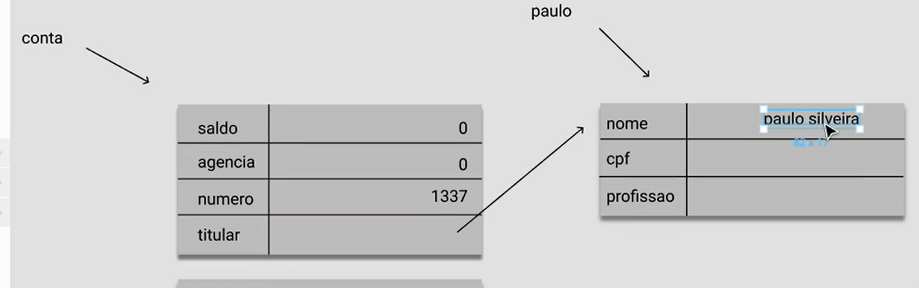
Quando o objeto dessa associação não está instanciado, não é possível atribuir valores e o valor retornado é *null*.

# Refatorar



Por convenção se escreve os parâmetros dos setters com o mesmo nome.

# Referenciação somada a getters e setters



System.***out***.println(conta.getTitular().getNome());

------------------------------------

Setters e Getters devem ser usados com cautela e nem todos os atributos privados precisam ter expostos esses dois métodos com riscos de cairmos em um modelo anêmico que tem os seus comportamentos controlados por outras classes.

# Criando um contador de objetos instanciados

**private** **static** **int** *totalContas*;

O *static* faz com que esse atributo da classe, seja de uso de todas os objetos instanciados. Assim todo objeto usa um só atributo.

Conta.*totalContas*++;

Tanto que ao fazer um incremento nesse atributo, a referencia usada é a classe, no caso a Classe conta, e não o habitual *this*.

public class Carro{

private int ano;

private String modelo;

private double preco;

public Carro(int ano, String modelo, double preco){

if(ano >= 1891){

this.ano = ano;

}else{

System.out.println("O ano informado está inválido. Por isso usaremos 2017!");

this.ano = 2017;

}

if( modelo != null){

this.modelo = modelo;

}else{

System.out.println("O modelo não foi informado. Por isso usaremos Gol!");

this.modelo = "Gol";

}

if(preco > 0){

this.preco = preco;

}else{

System.out.println("O preço não é válido. Por isso usaremos 40000.0!");

this.preco = 40000.0;

}

}

//Novo construtor AQUI!

public Carro(String modelo, double preco){

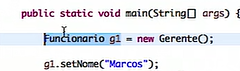
this(2017, modelo, preco);

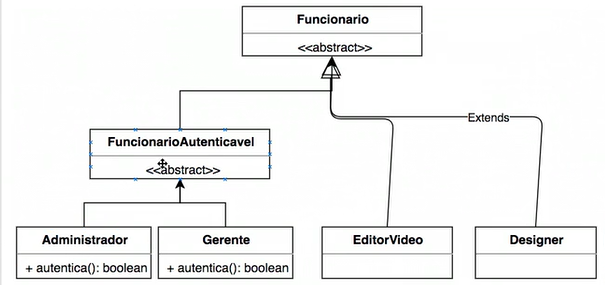
}

//getters e setters omitidos

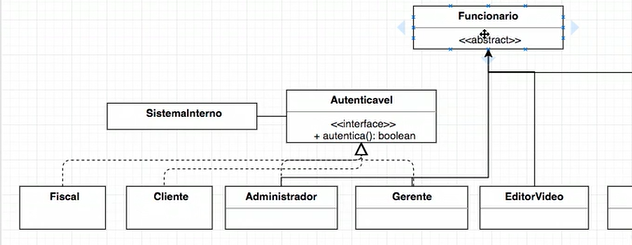
}

É possível criar objetos referenciando à classe mais genérica, mas esse objeto só acessa os objetos do tipo genérico.



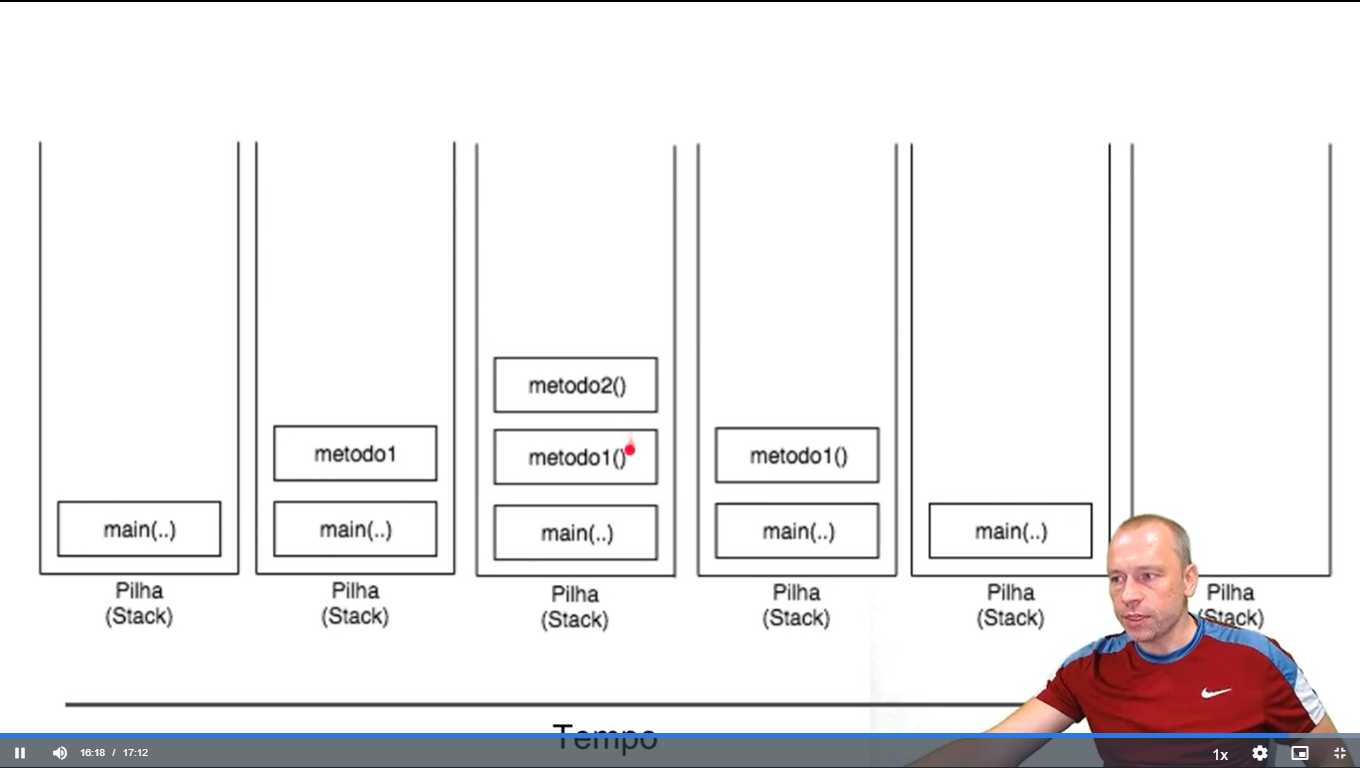


# Interfaces



# Java parte 4 – Entendendo exceções

## Pilha de execução

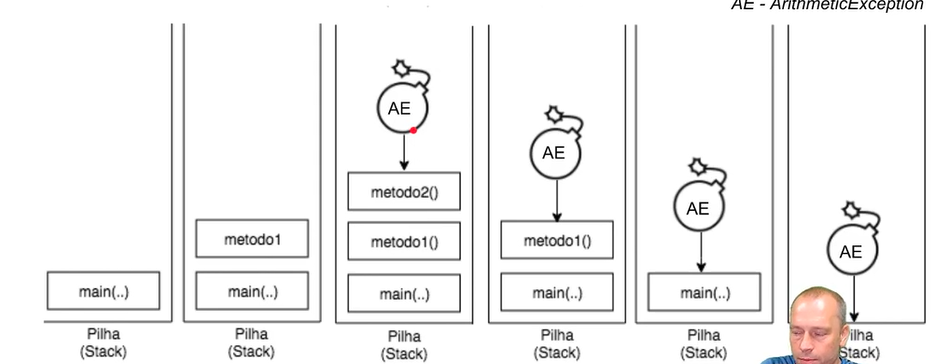


## Introdução a exceções

Exceções ocorrem quando não há erro sintático no código e sim quando coisas que não deveriam acontecer, acontecem.

Exceções mudam o fluxo. Fazem parte do controle de fluxo assim como os laços.

Quando não há código que resolva uma determinada situação.



Como nenhum dos métodos conseguiu resolver o problema da divisão por 0, todos os métodos foram descartados, e com isso encerrada a execução.

Não é possível ter uma instrução catch, sem uma instrução try.

Exato. A lógica de tratamento de erro no bloco catch só é disparada quando uma exceção é lançada dentro de um bloco try.

**try** {

*metodo2*();

} **catch** (ArithmeticException ex) { // nome da exceção

System.***out***.println("ArithmeticException");

}

System.***out***.println("Fim do metodo1");

ArithmeticException é uma classe do tipo ex. Podemos criar exceções através de classes.

String msg = ex.getMessage();

ex.printStackTrace();

Lançando Exceptions

Para lançar uma exceção, além de instanciá-la, é necessário lançá-la através do **throw**.

não é necessário guardar a exceção em uma referência, podendo lançá-la diretamente em uma linha só, por exemplo:

throw new ArithmeticException();

**public** **class** MinhaExcecao **extends** RuntimeException{

**public** MinhaExcecao(String msg) {

**super**(msg);

}

}

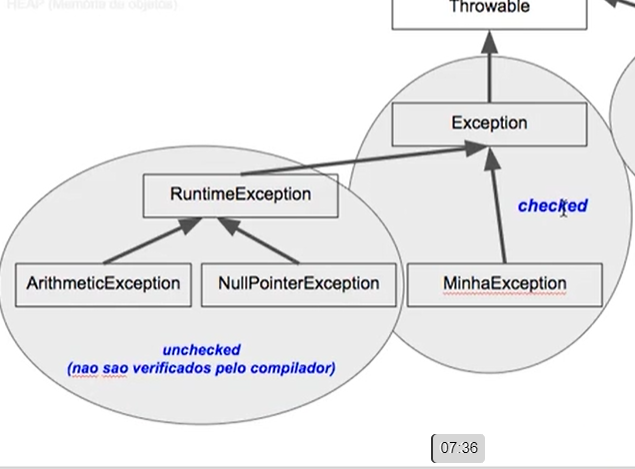
Entendendo Erros

# Checked e unchecked

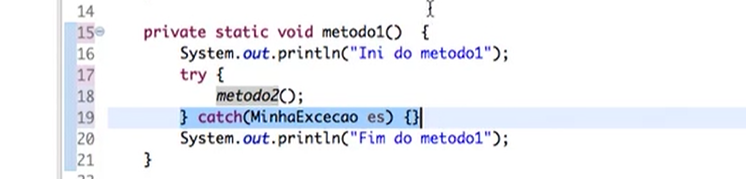
Quando se referencia uma Exceção própria que está herdando diretamente da classe Exception, precisa se deixar explícito na assinatura do método que ele joga uma exceção:

**public** **class** MinhaExcecao **extends** Exception{

**private** **static** **void** metodo2() **throws** MinhaExcecao {



Ou então se usa o try...catch:



Capturando qualquer exceção

Quando se sabe que o método pode ter erros, usa-se o checked.Quando erros não estão previstos, usa-se o unchecked e deixa livre para quem for usar tratar quando e como for necessário.

Capturando qualquer exceção:

} **catch** (Exception ex) {

Usando a classe mais genérica *Exception*, consegue-se pegar todas as exceções previstas por ela.

Normalmente se faz o tratamento da exceção antes.

# Finally e try with resources

IllegalStateException();// classe padrão do java que verifica estados inconsistentes. um objeto possui um estado inválido.

# Finally

**try**{

} **catch** (IllegalStateException ex) {;

}**finally** {

con.fecha();

}

O Finally é sempre executado, independente de haver erro ou não.

*Try’s* não compilam se estiverem sozinhos. É necessário pelo menos um *catch* ou um *finally.*

Esse trecho de try:

**try** (Conexao conexao = **new** Conexao()){

conexao.leDados();

} **catch** (IllegalStateException ex){

System.***out***.println("Erro na conexão");

}

Resume/Simplifica todo esse com try ... catch:

Conexao con = **null**;

**try**{

con = **new** Conexao();

con.leDados();

} **catch** (IllegalStateException ex) {

System.***out***.println("Erro na conexão");

}**finally** {

con.fecha();

}

IllegalArgumentException

# Java parte 5 - Pacotes e java.lang

Num projeto com Java, normalmente são criadas centenas de classes,os *packages*(pacotes) tem o objetivo de organizá-las. Os *packages* não são nada mais que as pastas dentro do projeto. Toda classe dentro de um *package* deve ter especificado na primeira linha a qual *package* ela pertence:

**package** modelo;

**public** **class** CalculadorDeImposto {

Ao se usar as classes desses *packages*, precisa-se referenciar/qualificar a que modelo ela pertence:

**package** teste;

**public** **class** TesteConta {

**public** **static** **void** main(String[] args) **throws** modelo.SaldoInsuficienteException{

modelo.ContaCorrente cc = **new** modelo.ContaCorrente(111, 111);

cc.deposita(100);

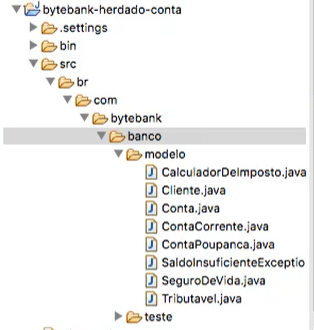
modelo.ContaPoupanca cp = **new** modelo.ContaPoupanca(222,222);

cp.deposita(200);

O nome que se dá à essas referencias completas é – Full Qualified Name (FQN)

O padrão de FQN definido pela comunidade, simula uma URL de site mas invertida:

NOME\_DO\_SITE\_AO\_CONTRARIO.NOME\_DO\_PROJETO



Para simplificar os FQN, existe as importações, com o *import* declarado na segunda linha da classe, podemos continuar usando as classes sem necessitar qualificar todas elas:

**import** br.com.bytebank.banco.modelo.\*;

Devido a quantidade de classes num projeto, acaba sendo melhor importar apenas classes especificas e não a pasta inteira.

## Distribuição do seu código

### JavaDoc

Comentários de documentação, feitos em cima da classe, do construtor e em cima dos metodos:

/\*\*

\* Classe representa a forma de uma conta.

\* **@author** brusa

\*

\*/

Já vimos nessa aula algumas tags (ou anotações) do *javadoc* como @version ou @author. Segue a lista completa:

* @author (usado na classe ou interface)
* @version (usado na classe ou interface)
* @param (usado no método e construtor)
* @return (usado apenas no método)
* @exception ou @throws (no método ou construtor)
* @see
* @since
* @serial
* @deprecated

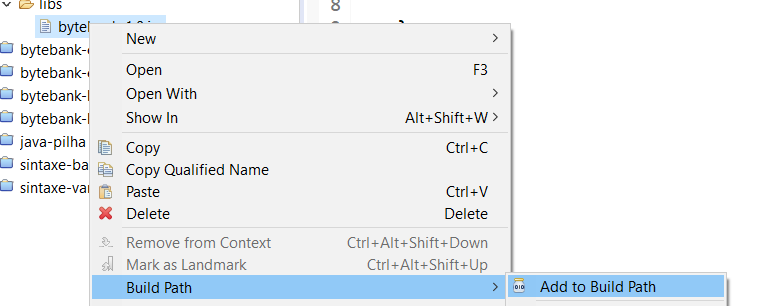
Os comentários feitos no projeto podem ser exportados para uma pagina html:

Project -> Generate JavaDoc

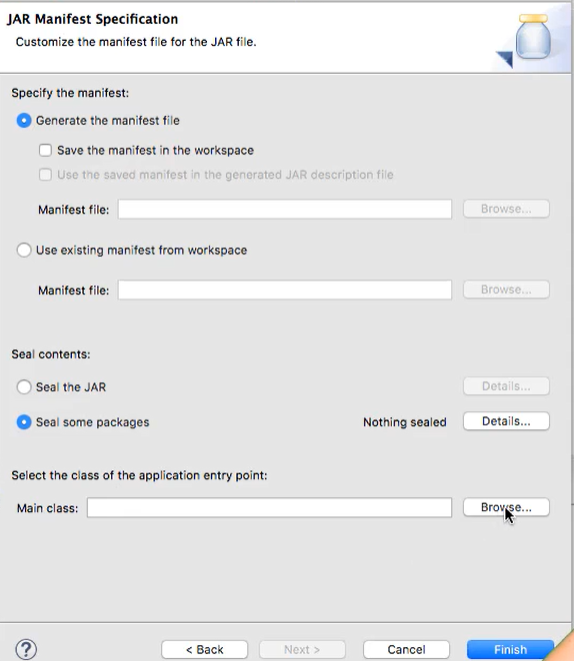
## Criando uma biblioteca com JAR

Para isso se exporta todas as classes compiladas ou não através do *Export.*

Esses arquivos são usados em outros projetos como *lib’s*.

Alem de adicionar o .jar no projeto, é necessário adicioná-los através do menu: 

## JAR Executável



# Para saber mais: Maven

[**PRÓXIMA ATIVIDADE**](https://cursos.alura.com.br/course/java-pacotes-e-java-lang/task/38490/next)

Java é uma plataforma de desenvolvimento completa que se destaca com sua grande quantidade de projeto open source. Para a maioria dos problemas no dia a dia do desenvolvedor já existem bibliotecas para resolver. Ou seja, se você gostaria de se conectar com um banco dados, ou trabalhar no desenvolvimento web, na área de data science, criação de serviços ou Android, já existem bibliotecas para tal, muitas vezes mais do que uma.

Aí existe a necessidade de organizar, centralizar e versionar os JARs dessa biblioteca e gerenciar as dependências entre elas. Para resolver isso, foram criadas ferramentas especificas e no mundo Java se destacou o Maven. O Maven organiza os JARs (código compilado, código fonte e documentação) em um repositório central que é publico e pode ser pesquisado:

<https://mvnrepository.com/>

Lá você pode ver e até baixar os JARs, mas o melhor é que a ferramenta Maven pode fazer isso para você. Se ficou interessado em aprender o Maven que ainda tem outros recursos bem legais, dá uma olhada no nosso curso especifico:

[Maven: Build do zero a web](https://cursos.alura.com.br/course/maven-build-do-zero-a-web)

Obs: Se você é usuário Linux, o Maven é bem parecido com os gerenciadores apt ou rpm. No MacOS existe o brew com o mesmo propósito. No mundo .Net temos o nuget e a plataforma node.js usa npm. Gerenciar dependências é um problema do coitadinho do desenvolvedor, e cada sistema ou plataforma possui a sua solução.

## O Pacote java.lang

Objetos do tipo String são imutáveis. Para fazer essa alteração é necessário criar outro objeto.

Exemplo:

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

String nome = "Alura";

String outra = nome.replace("A", "a");

//nome.toLowerCase();

System.***out***.println(outra);

}

Strings são apresentadas com aspas duplas. Os char’s com aspas simples.

# A interface CharSequence

[**PRÓXIMA ATIVIDADE**](https://cursos.alura.com.br/course/java-pacotes-e-java-lang/task/38496/next)

Nos vídeos talvez você tenha percebido que alguns métodos da classe String recebem uma variável do tipo CharSequence. O tipo CharSequence é uma interface que a própria classe String implementa (pois uma String é uma sequência de caracteres!):

public class String implements CharSequence {

Quando usamos a classe String até poderìamos declarar a variável com o tipo da interface, mas isso é raro de se ver:

CharSequence seq = "é uma sequencia de caracteres";

O interessante é que existem outras classes que também implementam a interface CharSequence. Em outras palavras, existem outras classes que são sequências de caracteres além da classe String. Por quê?

# A classe StringBuilder

Vimos que a classe String é especial pois gera objetos imutáveis. Isso é considerado benéfico pensando no design mas é ruim pensando no desempenho (e por isso devemos usar aspas duplas na criação, pois a JVM quer contornar os problemas no desempenho com otimizações).

Agora vem um problema: imagina que você precisa criar um texto enorme e precisa concatenar muitas String, por exemplo:

String texto = "Socorram";

texto = texto.concat("-");

texto = texto.concat("me");

texto = texto.concat(", ");

texto = texto.concat("subi ");

texto = texto.concat("no ");

texto = texto.concat("ônibus ");

texto = texto.concat("em ");

texto = texto.concat("Marrocos");

System.out.println(texto);

Nesse pequeno exemplo já criamos vários objetos, só porque estamos concatenando algumas Strings. Isso é nada bom pensando no desempenho e para resolver isso existe a classe StringBuilder que ajuda na concatenação de Strings de forma mais eficiente.

Veja o mesmo código usando o StringBuilder:

StringBuilder builder = new StringBuilder("Socorram");

builder.append("-");

builder.append("me");

builder.append(", ");

builder.append("subi ");

builder.append("no ");

builder.append("ônibus ");

builder.append("em ");

builder.append("Marrocos");

String texto = builder.toString();

System.out.println(texto);

O StringBuilder é uma classe comum. Repare que usamos o new para a criação do objeto. Além disso, como o objeto é mutável, utilizamos a mesma referência, sem novas atribuições.

# A interface CharSequence

Agora o legal é que a classe StringBuilder também implementa a interface CharSequence:

public class StringBuilder implements CharSequence {

CharSequence cs = new StringBuilder("também é uma sequencia de caracteres");

Isso faz que alguns métodos da classe String saibam trabalhar com o StringBuilder, por exemplo:

String nome = "ALURA";

CharSequence cs = new StringBuilder("al");

nome = nome.replace("AL", cs);

System.out.println(nome);

Vice-versa a classe StringBuilder tem métodos que recebem o tipo CharSequence. Dessa forma podemos trabalhar de maneira compatível com as duas classes, baseado numa interface comum.

Sobre o System.out.println

Public class System{

Public static ????? out; // atributo

}

System

- classe, java.lang, acesso publico

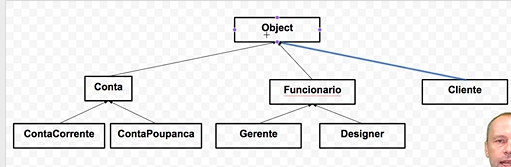
Out

- atributo, publico, referencia, static

Println

- método, publico, no static, sobrecarga , não joga exceções do tipo checked.

## Usando a classe object



Todas as classes herdam da classe OBJECT

# Java parte 6 – Conhecendo o java.util

## Começando Com Arrays

**int** [] idades = **new** **int** [5]; // **int** idades[] = **new** **int** [5];

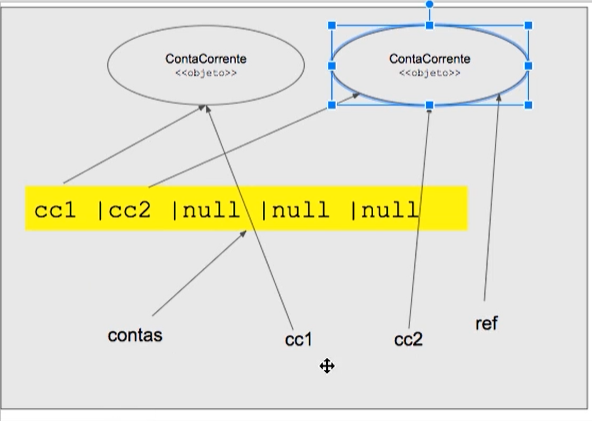
Os colchetes podem ficar tanto após o tipo, quanto após o nome do *Array*.

**int** [] idades = **new** **int** [5];

idades[0] = 29;

System.***out***.println(idades[0]);

# Guardando objetos num *Array*



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

//int [] idades = new int [5];

ContaCorrente [] contas = **new** ContaCorrente[5];

ContaCorrente cc1 = **new** ContaCorrente(22, 11);

ContaCorrente cc2 = **new** ContaCorrente(22,12);

contas[0] = cc1;

contas[1] = cc2;

System.***out***.println(cc2.getNumero());

System.***out***.println(contas[1].getNumero());

ContaCorrente ref = contas[1];

System.***out***.println(ref.getNumero());

}

Até agora vimos a forma "classica" de criar um objeto array usando a palavra chave new, por exemplo:

int[] numeros = new int[6];

numeros[0] = 1;

numeros[1] = 2;

numeros[2] = 3;

numeros[3] = 4;

numeros[4] = 5;

No entanto também há uma forma literal. Literal, nesse contexto, significa usar valores diretamente, menos burocrático, mais direito. Veja a diferença:

int[] refs = {1,2,3,4,5};

Usamos as chaves {} para indicar que se trata de um array e os valores já ficam declarados dentro das chaves.

# *Arrays* do tipo Object

A referencia também pode ser feita através de um tipo mais genérico da classe:

Conta[] contas = **new** Conta[5];

ContaCorrente cc1 = **new** ContaCorrente(22, 11);

ContaPoupanca cc2 = **new** ContaPoupanca(22,12);

contas[0] = cc1;

contas[1] = cc2;

System.***out***.println(cc2.getNumero());

System.***out***.println(contas[1].getNumero());

Conta ref = contas[1];

System.***out***.println(ref.getNumero());

E também através do *TYPE CAST :*

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

// int [] idades = new int [5];

Conta[] contas = **new** Conta[5];

ContaCorrente cc1 = **new** ContaCorrente(22, 11);

ContaPoupanca cc2 = **new** ContaPoupanca(22, 12);

contas[0] = cc1;

contas[1] = cc2;

System.***out***.println(cc2.getNumero());

System.***out***.println(contas[1].getNumero());

ContaPoupanca ref = (ContaPoupanca) contas[1]; // type cast

System.***out***.println(ref.getNumero());

}

# Cast implícito e explícito de primitivos

Para ser correto, já vimos o cast acontecendo antes mesmo de defini-lo. Temos dois exemplos, o primeiro do mundo de primitivos:

int numero = 3;

double valor = numero; //cast implícito

Repare que colocamos um valor da variável numero (tipo int) na variável valor (tipo double), sem usar um cast explícito. Isso funciona? A resposta é sim, pois qualquer inteiro cabe dentro de um double. Por isso o compilador fica quieto e não exige um cast explicito, mas nada impede de escrever:

int numero = 3;

double valor = (double) numero; //cast explícito

Agora, o contrário não funciona sem cast, uma vez que um double não cabe em um int:

double valor = 3.56;

int numero = (int) valor; //cast explicito é exigido pelo compilador

Nesse caso o compilador joga todo valor fracional fora e guarda apenas o valor inteiro.

# Cast implícito e explícito de referências

Nas referências, o mesmo princípio se aplica. Se o cast sempre funciona não é necessário deixá-lo explícito, por exemplo:

ContaCorrente cc1 = new ContaCorrente(22, 33);

Conta conta = cc1; //cast implicito

Aqui também poderia ser explícito, mas novamente, o compilador não exige pois qualquer ContaCorrente é uma Conta:

ContaCorrente cc1 = new ContaCorrente(22, 33);

Conta conta = (Conta) cc1; //cast explícito mas desnecessário

# Type cast explícito sempre funciona?

A resposta é não. O cast explicito só funciona se ele for possível, mas há casos em que o compilador sabe que um cast é impossível e aí nem compila, nem com type cast. Por exemplo:

Cliente cliente = new Cliente();

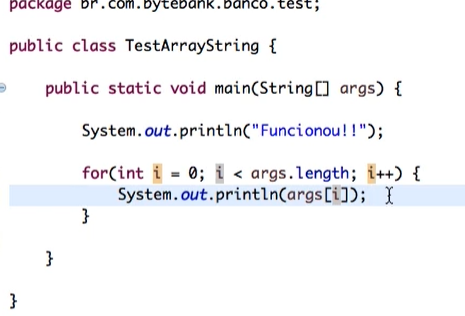
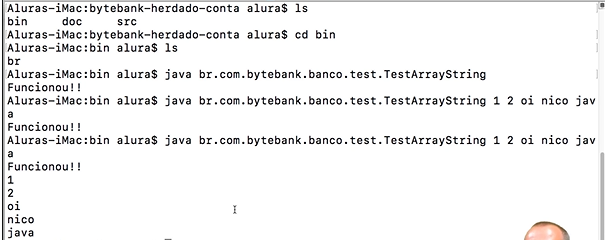
Conta conta = (Conta) cliente; //impossível, não compila

Como o cliente não estende a classe Conta ou implementa uma interface do tipo Conta, é impossível esse cast funcionar, pois uma referência do tipo Conta jamais pode apontar para um objeto do tipo Cliente.

A certificação Java tem muitas dessas perguntas sobre cast possível, impossível, explícito e implícito. Se você pretende tirar essa certificação, vale a pena estudar esse assunto com muita calma.

# *O que é o “....*main(String[] args)...”

O método *main* é capaz de receber parâmetros, que são guardados no *array* args[] que é do tipo String.



# Desvantagens de se usar um Array

O atibuto .lenght em si já é um problema, porque ele é público. Além disso, ele não é muito usual já que dificilmente vamos querer saber o tamanho do *array*, é muito mais provável que queiramos saber quantas posições do nosso *array* estão ocupadas. Somado à esses problemas, temos o fato de que um *array* não é expansível/dinâmico.

Para driblar o problema dos métodos públicos do *array,* poderia-se criar uma classe que tivesse como variável, um array que fosse *private*.

**public** **class** TesteGuardador {

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

GuardadorDeContas guardador = **new** GuardadorDeContas();

Conta cc = **new** ContaCorrente(22, 11);

guardador.adiciona(cc);

Conta cc2 = **new** ContaCorrente(22, 22);

guardador.adiciona(cc2);

**int** tamanho = guardador.getQuantidadeDeElementos();

System.***out***.println(tamanho);

Conta ref = guardador.getReferencia(0);

System.***out***.println(ref.getNumero());

}

}

Mesmo assim, não seria possível driblar a falta de expansividade do array.

# ArrayList

Métodos Padrões de um ArrayList:

lista.add(cc2);

lista.size();

lista.get(0); - Como os ArrayList funcionam guardando outros “Arrays”/Objetos dentro dele, o retorno disso é um Object.

ARRAYLISTS só guardam referencias à objetos, não guardam variáveis.

sobre a ArrayList:

* a) Guarda referências.
* b) É do pacote java.util
* c) Usa internamente um array

O que acontece se você adiciona uma referência na lista sem definir a posição? Método .add();

O elemento é adicionado no fim da lista

Quantos elementos um objeto do tipo java.util.ArrayList pode guardar?

O limite é a memória da JVM.

Num ArrayList só pode se guardar um tipo de dado. Por isso, a partir do EcmaScript 5, foi introduzida a especificação do tipo aceito por aquele ArrayList:

ArrayList<Conta> lista = **new** ArrayList<Conta>();

<> são conhecidos como Generics

Os generics entraram na versão 1.5 na plataforma Java e foram levemente melhorados no Java 1.7. Quais são os principais benefícios?

O código mais legível, já que fica explícito o tipo dos elementos.

Correto, na criação da lista fica claro qual a intenção. Por exemplo, na declaração abaixo está bem claro que a lista guarda Strings:

ArrayList<String> nomes = new ArrayList<String>();

Evitar *casts* excessivos

Correto, uma vez parametrizada a lista, não precisamos mais o cast, por exemplo:

Cliente ref = (Cliente) lista.get(0); // desnecessário com generics

Antecipar problemas de *casts* no momento de compilação.

Correto, o compilador avisa se erramos no tipo, por exemplo:

ArrayList<String> lista = new ArrayList<String>();

lista.add("Nico");

Conta c = lista.get(0); //nao compila

## Lista com capacidade predefinida

Falamos que o ArrayList é um array dinâmico, ou seja, por baixo dos panos é usado um array, mas sem se preocupar com os detalhes e limitações.

Agora pense que você precisa criar uma lista representando todos os 26 estados do Brasil. Você gostaria de usar um ArrayList para "fugir" do array, mas sabe que o ArrayList cria um array automaticamente, do tamanho que a classe acha conveniente.

Será que não há uma forma de criar essa lista já definindo o tamanho do array? Claro que tem e é muito simples. O construtor da classe ArrayList é sobrecarregado e possui um parâmetro que recebe a capacidade:

ArrayList lista = new ArrayList(26); //capacidade inicial

## Lista a partir de outra

Outra forma de inicializar uma lista é baseado na outra que é muito comum no dia a dia. Para tal a ArrayList possui mais um construtor que recebe a lista base:

ArrayList lista = new ArrayList(26); //capacidade inicial

lista.add("RJ");

lista.add("SP");

//outros estados

ArrayList nova = new ArrayList(lista); //criando baseado na primeira lista

Quanto mais sabemos sobre as classes Java padrão mais fácil fica o nosso código.

Método Equals

O método *contains()* verifica apenas a igualdade entre referencias dentro de um ArrayList, porque usa dentro de si, o método *equals()* que verifica a igualdade das referencias.

É possível modificar essa verificação através da sobrescrita desse método equals().

* Devemos sobrescrever para definir o critério de igualdade.

Correto, devemos sobrescrever o método equals para definir a igualdade do objeto.

Em geral, os métodos equals, toString e hashCode existem para sobrescrita.

* A implementação padrão compara as referências.

Correto, a implementação do método equals da classe Object compara apenas as referências.

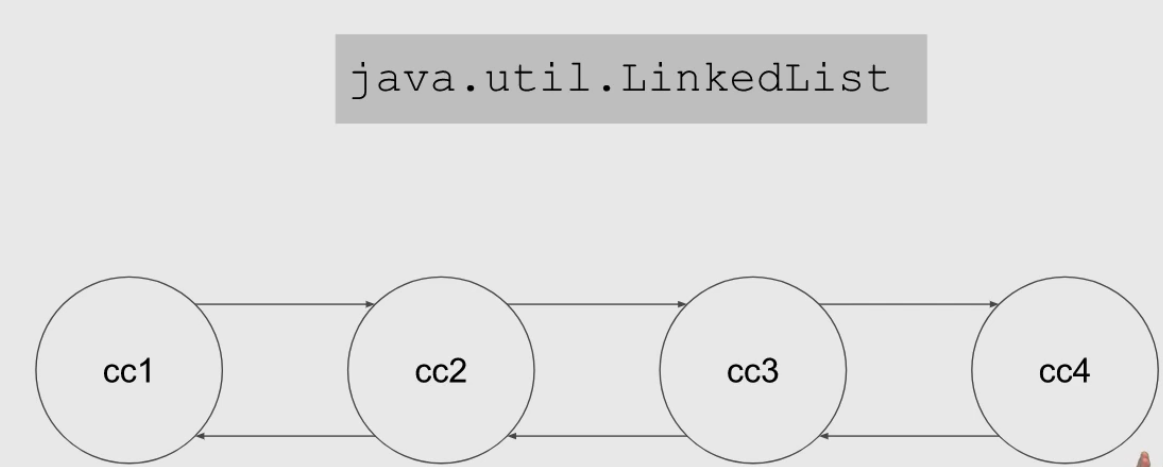
# Vantagens ArrayList

Acesso aleatório fácil

Adicionar novos elementos é fácil, pois internamente ele sabe a última posição vazia.

Desvantagens  
Quando um elemento for apagado, o Array teria que reorganizar as posições dos elementos. Se o primeiro elemento for apagado, os demais teriam que ser “empurrados para trás”, o que é bem trabalhoso no ArrayList.

# LinkedList



A ordem de inserção é importantíssima.

Trabalha com índice: métodos get, etc.

Mas ele não usa um Array “por debaixo dos panos”.

Lista duplamente encadeada. Os elementos tem a informação de quem é seu antecessor e seu sucessor.

Sabe-se que se está no final da lista porque esse elemento não tem um anterior.

Apagar um elemento não é tão custoso porque em tese está apenas sendo tirada referência de qual é o antecessor do elemento seguinte ao que foi apagado.

Desvantagem

Para fazer iterações ou fazer acessos diretos a um elemento, é necessário sempre começar à partir do primeiro elemento.

As LinkedList têm os mesmos métodos da ArrayList.

A classe ArrayList e LinkedList são especificações/tipos da interface List.

Todas as listas garantem a ordem de inserção.

Correto, as listas garantem a ordem de inserção. Isso significa que ao iterar recebemos os elementos na mesma ordem que eles foram inseridos.

A partir de agora vamos usar as listas para fugir das desvantagens do array. No entanto, se lembra do nosso array String[] do método main? Com certeza, e não podemos mudar a assinatura do método main pois a JVM não aceita isso. Bom, já que não podemos alterar a assinatura será que não tem uma forma de transformar uma array em uma lista? Claro que existe, e para tal, existe já uma classe que ajuda nessa tarefa: java.util.Arrays

A classe java.util.Arrays possui vários métodos estáticos auxiliares para trabalhar com arrays. Veja como fica simples de transformar um array para uma lista:

public class Teste {

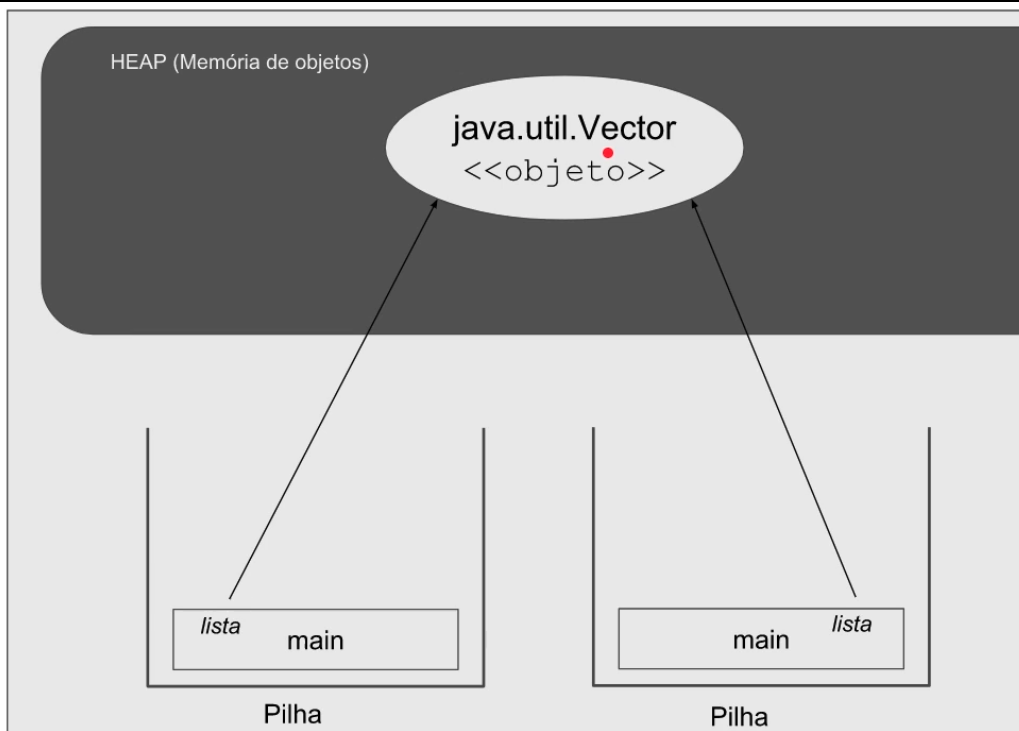
public static void main(String[] args) {

List<String> argumentos = Arrays.asList(args);

}

}

Vetores:

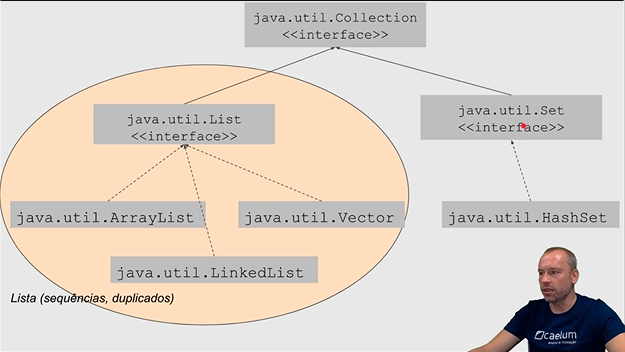


Quando se cria um “outro método main()”, ele atua em paralelo com o main() principal, gerando uma outra pilha. Quando se quer que essas duas pilhas trabalhem em cima da mesma lista, isso só será possível através do vetor(que é thread safe).

Vetor também é uma implementação da interface *List.*

*UMA LISTA ACEITA ELEMENTOS DUPLICADOS, REFERÊNCIAS DUPLICADAS*

# A interface Collection

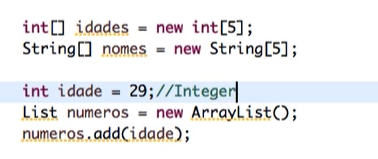


# As Classes Wrappers

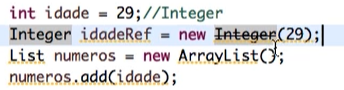
Tipos de estruturas de dados (e dados):

* Variáveis (dados)
* Arrays
* ArrayLists
* LinkedLists
* Vector

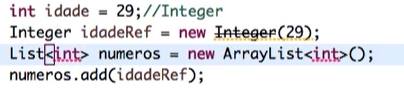
As estruturas de dados, por não guardarem variáveis e sim objetos, o que está sendo feito no código abaixo é “impossível”.



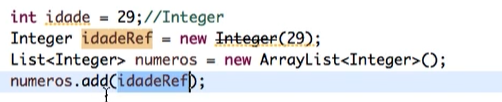
O que acontece por de baixo dos panos, é que a partir desse variável que está sendo guardada nesse ArrayList(que é um estrutura de dados complexa) está sendo criado um objeto da *class* INTEGER.



Da mesma forma que essa variável não é guardada no ArrayList por ser de um tipo primitivo, fazer o uso dos <Generics> para tentar corrigir esse problema, também não é válido:



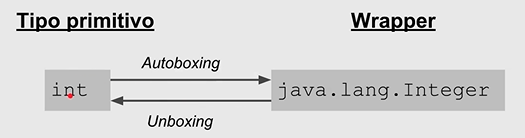
Para resolver isso, o <Generics> tem que ser feito usando a classe que foi criada para representar os tipos primitivos, que nesse caso é o *int*, cuja classe é o INTEGER.



O nome dado ao ato de tentar adicionar um tipo primitivo a uma estrutura de dados, como foi feito aqui, é AUTOBOXING :

List <Integer> numeros = **new** ArrayList<Integer>();

numeros.add(29);//Autoboxing



Metodos da classe integer

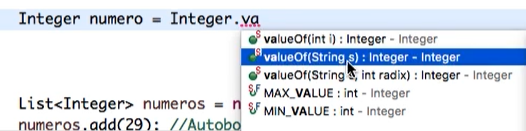
Para criar um objeto da Classe Integer usa-se:

Integer idadeRef = Integer.*valueOf*(29);

**int** valor = idadeRef.intValue();

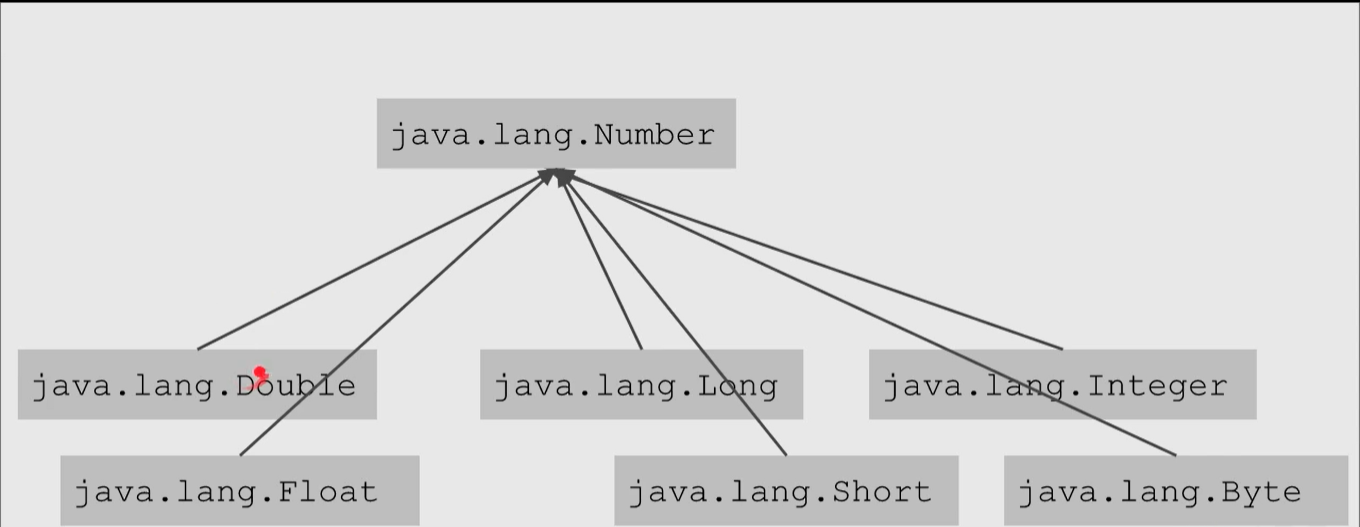
O código acima é um exemplo de *Autoboxing* e *Unboxing.*

Existe também um outro método .valueOf() que lê números guardados como Strings e os retorna como int.



É muito comum pegar um numero de um campo de texto como String e transforma-lo para int, double, etc.

Constantes no Java, são escritas com letras maiúsculas: ***MAX\_VALUE.***



# Ordenação de Listas

Para ordenar uma lista, é usado o método .sort(), esse método recebe um parâmetro que é uma interface chamada *Comparator*. Essa interface possui os métodos de comparação, que podem ser implementados/sobrescritos na classe em que ele será usado. Sem definirmos o nosso método/critério de comparação através da implementação de um *Comparator* especifico, o método .sort() não é tão efetivo quanto deveria.

**class** NumeroDaContaComparator **implements** Comparator<Conta> {

@Override

O <Generics> de <Conta> ali, está para especificar que essa classe, e consequentemente o método dela, que vai fazer a comparação, só vão aceitar objetos do tipo Conta.

A interface *Comparator*, dá retornos de números negativos, positivos, e zero, no caso da comparação ser falsa, verdadeira, ou deles serem iguais.

(Creio que esse retorno é usado em alguma outra parte da interface)

Outras formas de criar a logica para comparação:

**return** c1.getNumero() - c2.getNumero();

Para usar esse novo comparador criado, é necessário instanciar ela:

NumeroDaContaComparator comparador = **new** NumeroDaContaComparator();

Forma mais enxuta de realizar o instanciamento da classe:

lista.sort(**new** TitularDaContaComparator());

E então coloca-la como parâmetro do método .sort():

lista.sort(comparador);

# Comparando Strings

Para ordenar as Strings. O principio é muito parecido com o usado nas contas. É ainda mais simples por a Classe String já possui o método .compare.To(), com ele não é preciso montar uma logica de como ordenar os caracteres.

**class** TitularDaContaComparator **implements** Comparator <Conta>{

@Override

**public** **int** compare(Conta c1, Conta c2) {

String nomeC1 = c1.getTitular().getNome();

String nomeC2 = c2.getTitular().getNome();

**return** nomeC1.compareTo(nomeC2);

}

}

# A ordem Natural

Classe Collections

Qualquer objeto incluído na lista pode definir a ordem natural. É aquilo que define a ordem, se não foi definido uma ordem.

**public** **abstract** **class** Conta **extends** Object **implements** Comparable<Conta>{

Com isso, se implementa um método na classe, o .compareTo() na classe “raiz”, funcionando como um construtor,

No contexto da aula, o que significa ordem natural?

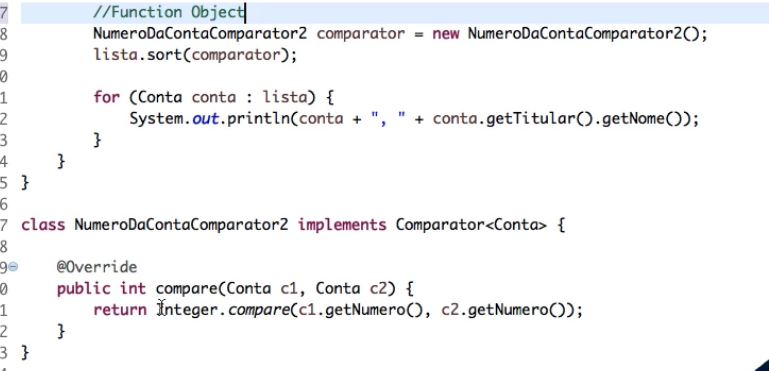
É a ordem definida pelo próprio elemento da lista.

Correto, no nosso exemplo a classe Conta define o seu próprio critério de ordenação implementando a interface java.lang.Comparable.

Nesse caso não é utilizado nenhum Comparator.

# Classes anônimas e Lambdas

Sendo o Java uma linguagem O.O., ela preza por suas boas práticas. A O.O. define que uma classe, deveria conter seus próprios atributos e métodos. Porém a solução dada para ordenar as listas, foi: criar uma classe, que pegasse os atributos de outras Classes para comparar, ou seja, essa classe foi criada apenas para encapsular um método, o que foge das orientações a objetos. A esses objetos instanciados, que tem função de métodos apenas, é dado o nome de FUNCTION OBJECT.



Criar uma nova classe do 0 apenas para encapsular um método, que ainda vai ser usado só em um momento especifico no código, é um motivo fraco, por isso existem as classes anônimas.

## Classes Anônimas

A “Function Object” ou a Classe que continha apenas um método:

**class** NumeroDaContaComparator2 **implements** Comparator<Conta> {

@Override

**public** **int** compare(Conta c1, Conta c2) {

**return** c1.getNumero() - c2.getNumero();

}

}

Que ainda precisava ser instanciada para funcionar:

NumeroDaContaComparator comparator = **new** NumeroDaContaComparator();

Pode ser substituída por uma Classe Anônima:

lista.sort(**new** Comparator<Conta>() {

@Override

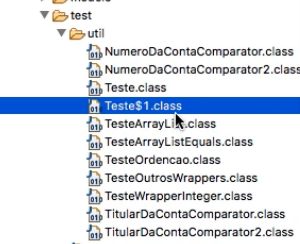
**public** **int** compare(Conta c1, Conta c2) {

**return** c1.getNumero() - c2.getNumero();

}

});

Com isso, essa classe é até criada na pasta do projeto:



Ainda pode se criar uma classe Anônima, passando ela para uma variável, que nesse caso, poderia ser passada para o método .sort() para fazer a ordenação:

Comparator<Conta> comp = **new** Comparator<Conta>() {

@Override

**public** **int** compare(Conta c1, Conta c2) {

String nomeC1 = c1.getTitular().getNome();

String nomeC2 = c2.getTitular().getNome();

**return** nomeC1.compareTo(nomeC2);

}

};

Com a criação das Classes Anônimas, acaba-se economizando na criação de outra classe, porém isso acaba dificultando a legibilidade do código.

## Finalmente Lambdas

Lambdas são basicamente Arrow Functions ‘-‘. Sintaxe:

lista.sort((Conta c1, Conta c2) -> {

**return** c1.getNumero() - c2.getNumero();

}

);

Outro Exemplo:

Comparator<Conta> comp = (Conta c1, Conta c2) -> {

String nomeC1 = c1.getTitular().getNome();

String nomeC2 = c2.getTitular().getNome();

**return** nomeC1.compareTo(nomeC2);

};

Simplificado ainda mais com a retirada das chaves e do return:

lista.sort((Conta c1, Conta c2) ->

Integer.*compare*(c1.getNumero(), c2.getNumero())

);

# O padrão Iterator

Você já sabe agora que existem muitas coleções. Só nesse treinamento vimos ArrayList, LinkedList e Vector. Se você assistir ainda o curso dedicado as coleções você aprenderá as interfaces para fila (Queue), conjunto (Set) e mapa (Map) cada uma com várias implementações.

Aí vem uma pergunta: Como posso acessar (iterar) todas essas implementações de maneira uniforme sem saber os detalhes de cada implementação? A resposta está na "caixa de padrões de projeto" e se chama Iterator.

Uma Iterator é um objeto que possui no mínimo dois métodos: hasNext()e next(). Ou seja, você pode usá-lo para perguntar se existe um próximo elemento e pedir o próximo elemento. A notícia boa é que isso funciona com TODAS as implementações e aí a grande vantagem.

Veja o código para usar o Iterator de uma lista:

List<String> nomes = new ArrayList<>();

nomes.add("Super Mario");

nomes.add("Yoshi");

nomes.add("Donkey Kong");

Iterator<String> it = nomes.iterator();

while(it.hasNext()) {

System.out.println(it.next());

}

Se você entendeu esse código, você já aprendeu como iterar com filas, conjuntos ou mapas. Veja o uso do Iterator através de um conjunto:

Set<String> nomes = new HashSet<>();

nomes.add("Super Mario");

nomes.add("Yoshi");

nomes.add("Donkey Kong");

Iterator<String> it = nomes.iterator();

while(it.hasNext()) {

System.out.println(it.next());

}

Se ficou ainda com dúvida sobre o Iterator, sem problemas (!) pois o curso [Dominando as Collections](https://cursos.alura.com.br/course/java-collections) possui um capítulo dedicado.

# Pacote Java.io



## Estabelecendo uma Entrada

**import** java.io.IOException;

**import** java.io.FileNotFoundException;