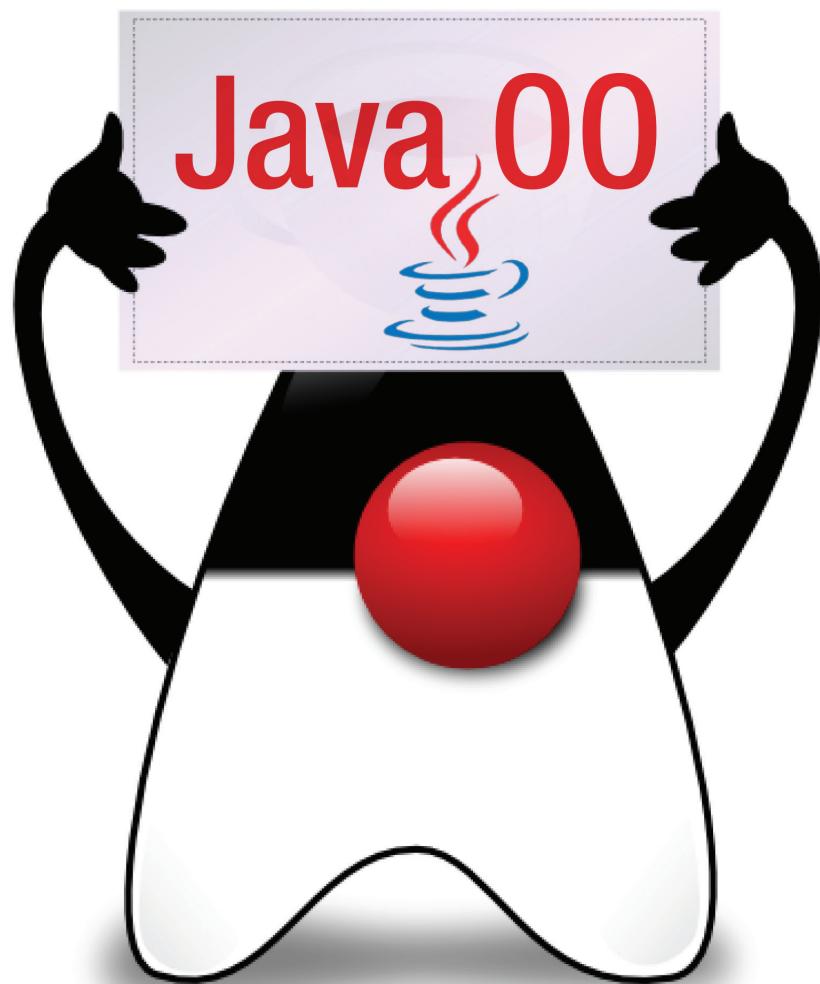




# Java e Orientação a Objetos



<b>Introdução à linguagem Java .....</b>	<b>6</b>
O que é Java? .....	7
Um pouco de História .....	8
Onde encontro Java? .....	9
Plataforma Java .....	10
Garbage Collection (Coletor de Lixo) .....	11
Divisão da Plataforma .....	12
Fases do Programa Java .....	14
Aplicações mais comuns em Java .....	16
O método main .....	18
<b>Identificadores, palavras-chave e tipo .....</b>	<b>19</b>
JavaDoc .....	20
Ponto-e-Vírgula, Blocos e Espaço .....	22
Identificadores e Palavras Reservadas .....	23
Variáveis, Declaração e Atribuição .....	25
Tipos de Dados .....	27
Casting de Tipos Primitivos .....	29
Classes Wrapper (Empacotadoras) .....	30
Construtores e método valueOf .....	31
Auto Boxing - Boxing and Unboxing .....	32
<b>Operadores .....</b>	<b>33</b>
Operadores Aritiméticos .....	34
Operadores Relacionais .....	35
Operadores Lógicos .....	36
&& ( e lógico) e & ( e binário) .....	37
( ou lógico) e   ( ou binário) .....	38
^ ( ou exclusivo binário) .....	39
! (Negação) .....	40
Operadores de Incremento e Decremento .....	41
Precedência de Operadores .....	42
Operador Condicional (?:) .....	44
<b>Estruturas de Controle .....</b>	<b>45</b>
Estrutura de decisão (if) .....	46
Estrutura de decisão (if-else) .....	47
Estrutura de decisão (if-else-if) .....	48
Estrutura de decisão (switch) .....	49
Estrutura de repetição (while) .....	50

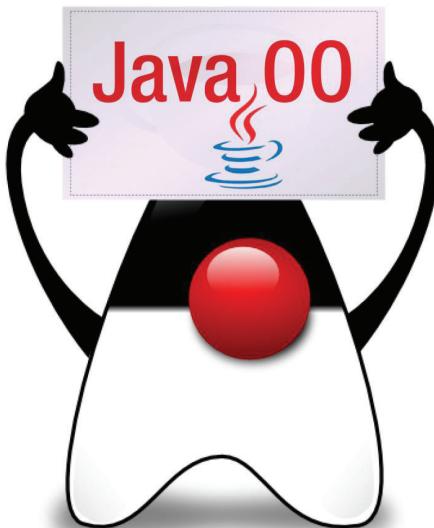
Estrutura de repetição (do-while) .....	51
Estrutura de repetição (for) .....	52
Declaração break .....	53
Declaração continue .....	54
<b>Array .....</b>	<b>55</b>
Array? .....	56
Declarando Array .....	57
Acessando um elemento do Array .....	59
Arrays Multidimensionais .....	60
Acessando um elemento de um Array Multidimensional .....	61
Percorrendo Arrays com Enhanced-for .....	62
Manipulando Arrays com java.util.Arrays .....	63
<b>Bases da programação Java 00 .....</b>	<b>64</b>
Pacotes .....	66
Modificadores de acesso .....	68
Classes .....	70
Definindo Classes .....	71
Métodos .....	72
Definindo Métodos .....	73
Objetos .....	74
Classe X Objeto .....	75
Notação UML .....	76
Notação UML   Diagrama de Classe .....	77
Notação UML   Relacionamentos .....	78
Herança .....	80
Agregação .....	81
<b>Métodos, Construtores e Membros Estáticos .....</b>	<b>82</b>
Declarando Membros: Variáveis e Métodos .....	83
Referência de Objetos .....	84
Invocação de Métodos .....	86
Passagem de Parâmetro por Valor .....	88
Passagem de Parâmetro por Referência .....	89
Sobrecarga de métodos (Overloading) .....	90
Construtores .....	91
Overloading de Construtores .....	93
Utilizando o Construtor this() .....	94
Instância de Classes .....	95

Membros estáticos .....	96
<b>Herança e Polimorfismo .....</b>	<b>98</b>
Herança .....	99
SuperClasse e SubClasse .....	101
Herança - classe Veiculo .....	102
Herança - classe Carro .....	103
Modificador de Classe final .....	104
Polimorfismo .....	105
Sobreposição de Métodos @Override .....	107
Referência polimórficas .....	109
Coleções Heterogêneas de Objetos .....	110
Determinando a Classe de um Objeto .....	111
Modificador de método final .....	113
Encapsulamento .....	114
Métodos de Configuração e Captura .....	115
<b>Classes Abstratas, Internas e Interfaces .....</b>	<b>116</b>
Classes Abstratas .....	117
Métodos Abstratos .....	118
Exemplos de implementação .....	119
Interfaces .....	121
Criando Interfaces .....	122
Implementando Interfaces .....	123
Herança entre interfaces .....	124
Interface vs. Classe .....	125
Classes Internas (Aninhadas) .....	126
Tipos Enumerados .....	128
<b>Exceções .....</b>	<b>130</b>
Categoria de Exceções .....	132
Manipulando Exceções .....	134
Throw e Throws .....	135
Exceções Verificadas e Não Verificadas .....	136
Criando Exceções .....	138
Sobrescrita de Métodos e Exceções .....	139
<b>Tipos Genéricos .....</b>	<b>140</b>
Declarando uma Classe utilizando Generics .....	143
Limitação “Primitiva” .....	145
Limitando Genéricos .....	146

Coringa <?> .....	148
<b>Java Orientado a Objetos Collections .....</b>	<b>149</b>
Java Collections .....	150
Java Collections - Hierarquia .....	151
Interfaces Set e List .....	153
Generics e Coleções Java .....	155
Interface Iterator .....	157
Percorrendo Collections .....	158
Classificando Coleções: Collections.sort .....	159
Interface Comparable .....	160
<b>Lendo e Escrevendo Arquivos .....</b>	<b>161</b>
Console I/O .....	162
Usando a classe Scanner .....	163
java.io.File .....	165
FileWriter e BufferedWriter .....	167
FileReader e BufferedReader .....	168
<b>Threads .....</b>	<b>169</b>
Ciclo de vida de uma Thread .....	172
Criando Threads .....	174
Iniciando Threads .....	176
Escalonamento da Thread .....	177
Prioridades de uma Thread .....	178
Sincronização .....	179
Bloqueando acesso Concorrente .....	180
<b>Construindo interfaces gráficas com Swing .....</b>	<b>182</b>
AWT versus Swing .....	183
Componentes AWT .....	185
Gerenciadores de layout .....	187
Componentes Swing .....	189
Containers JFrame .....	192
Manipulação de Eventos .....	193
Classes de Evento .....	195
Criação de aplicações gráficas com Eventos .....	197
Classes Adaptadoras .....	198

# Java e Orientação a Objetos

## Introdução à linguagem Java



## Java e Orientação a Objetos

### O objetivo do curso

Propiciar você a compreensão da sintaxe da linguagem Java, compilar e executar programas Java, criar programas com interface gráfica e entender os conceitos de orientação a objetos.

Para vocês que estão fazendo o curso Java e Orientação a Objetos, é recomendado estudar em casa aquilo que foi visto durante a aula, tentando resolver todos os exercícios.

### Dúvidas

Para tirar dúvidas dos exercícios, ou de Java em geral, recomendamos o fórum do GUJ (<http://www.guj.com.br>) e do Javafree (<http://javafree.uol.com.br/>), onde sua dúvida será respondida prontamente.

### Bibliografia extra

- > Java - Como programar, de Harvey M. Deitel;
- > Use a cabeça! - Java, de Bert Bates e Kathy Sierra;
- > Fundamentals of Object-Oriented Design in UML;





## Java e Orientação a Objetos

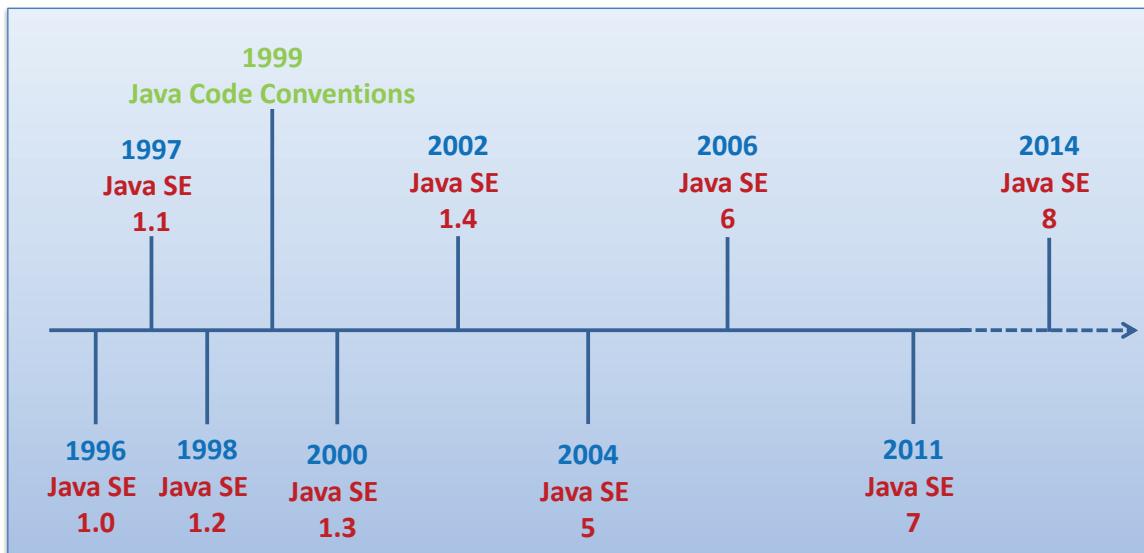
Java é uma linguagem de programação e uma plataforma de computação. É a tecnologia que capacita muitos programas da mais alta qualidade, como utilitários, jogos e aplicativos corporativos, entre muitos outros.

Java é uma linguagem simples, existem poucas regras, muito bem definidas, orientada a objetos, fácil de aprender, e que possui como maior vantagem ser portável entre as diversas plataformas existentes.

Por exemplo, se o programa Java for feito em uma plataforma Linux, ele pode ser usado na plataforma Mac ou Windows, o programa rodará sem problema nenhum.

De forma bastante simplificada podemos dizer que a tecnologia Java é composta por uma gama de produtos pensados para utilizar o melhor das redes de computadores e capaz de serem executados (rodar) em diferentes máquinas, sistemas operacionais e dispositivos. Ao longo deste curso você conseguira responder com suas próprias palavras o que Java.

# Um pouco da História



## Java e Orientação a Objetos

A tecnologia Java começou a ser criada em 1991 com o nome de Green Project. O projeto era esperado como a próxima geração de software embarcado. Nele trabalhavam James Gosling, Mike Sheridan e PatrikNaughton. Em 1992 surge à linguagem Oak, a primeira máquina virtual implementada. Várias tentativas de negócio foram feitas para vender o Oak, mas nenhuma com sucesso. Em 1994 surge à internet, a Sun vê uma nova possibilidade para o Green Project e cria uma linguagem para construir aplicativos Web baseada na Oak, o Java. Em 23 de maio de 1995 a linguagem Java é oficialmente lançada na SunWorld Expo 95 com a versão JDK 1.0 alpha. A Netscape apostava na idéia e inicia a implementação de interpretadores Java em seu navegador, possibilitando a criação de Java applets. A partir desta etapa o Java começa a crescer muito.

De 1998 até hoje a tecnologia evoluiu muito possuindo um dos maiores repositórios de projetos livres do mundo, o java.net. Em 1998 surgiu a plataforma para desenvolvimento e distribuição corporativa batizado de Java 1.2 Enterprise Edition (J2EE) e a plataforma Java 1.2 Mobile Edition (J2ME) para dispositivos móveis, celulares, PDAs e outros aparelhos limitados.

Atualmente Java é uma das linguagens mais usadas e serve para qualquer tipo de aplicação, entre elas: web, desktop, servidores, mainframes, jogos, aplicações móveis, chips de identificação, etc.

## Onde encontro Java?



### Java e Orientação a Objetos

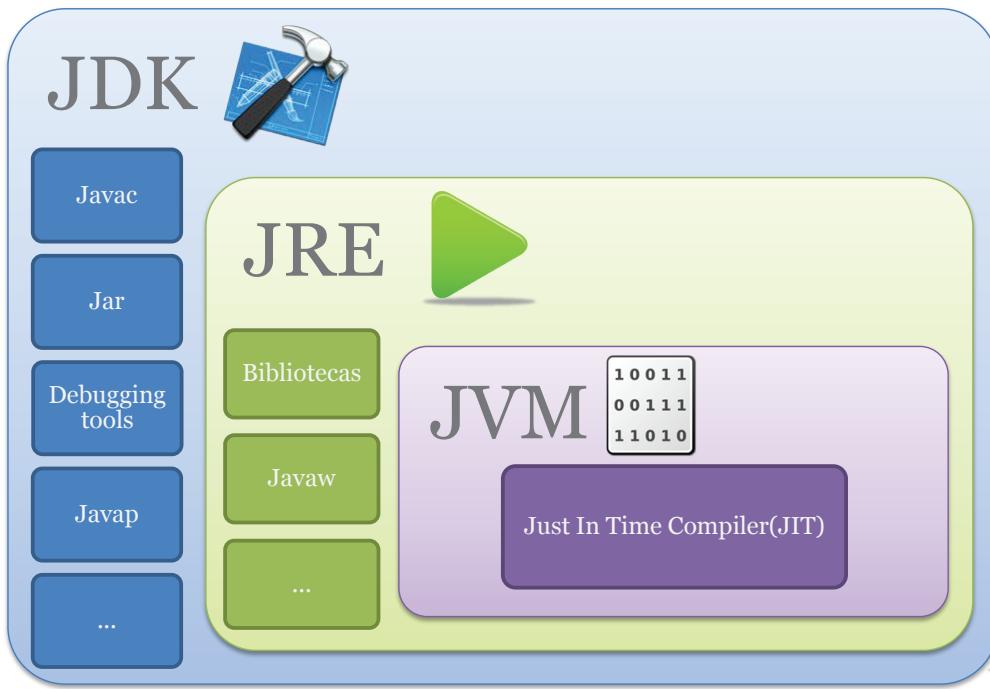
O Java é executado em mais de 850 milhões de computadores pessoais e em bilhões de dispositivos em todo o mundo. A tecnologia Java está em todo lugar! Ela pode ser encontrada inclusive em dispositivos de televisão, laptops, datacenters, consoles de jogo, super-computadores científicos, telefones celulares e até na Internet.

### O que você pode fazer em Java?

TUDO! Java é uma linguagem que não se prende a nenhuma arquitetura e a nenhuma empresa, é rápida e estável. Pode construir sistemas críticos, sistemas que precisam de velocidade e até sistemas que vão para fora do planeta, como a sonda Spirit enviada pela Nasa para Marte. Java tem um mar de projetos open source, que estão lá, esperando por usuários e desenvolvedores.

Há muitos aplicativos e sites que funcionam somente com o Java instalado, e muitos outros aplicativos e sites são desenvolvidos e disponibilizados com o suporte dessa tecnologia todos os dias. O Java é rápido, seguro e confiável.

# Plataforma Java



## Java e Orientação a Objetos

### JDK (Java Development Kit)

É o conjunto de ferramentas necessárias para realizar o desenvolvimento de aplicações Java e inclui a JRE e ferramentas de programação, como:

- > javac – Compilador
- > jar – Empacotador
- > javadoc – Ferramenta para geração de documentação

### JRE (Java Runtime Environment)

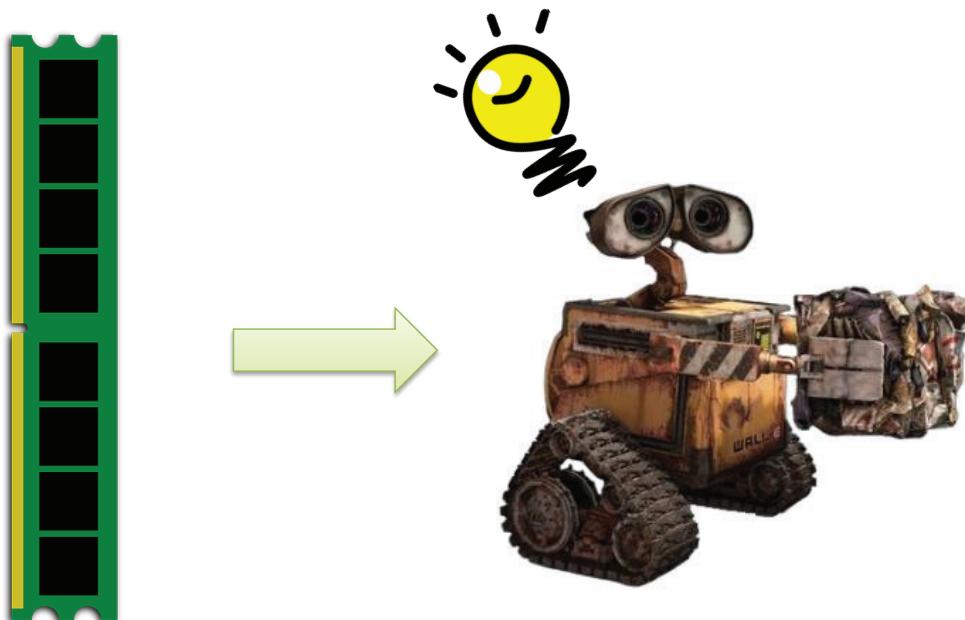
É composto pela JVM e pela biblioteca de classes Java utilizada para execução de aplicações Java, estas bibliotecas são chamadas de APIs Java. Portanto para rodar uma aplicação Java é necessário que você instale uma JRE no computador onde o software foi instalado.

### JVM (Java Virtual Machine)

A JVM é a máquina virtual responsável por interpretar e executar o código Java compilado (bytecode) e, portanto, são provedoras de formas e meios do aplicativo conversar com o sistema operacional.

Esta abstração viabiliza a implementações da JVM para diferentes plataformas de hardware e de sistemas operacionais, o que possibilita que aplicativos Java sejam multi-plataforma. Uma JVM pode ser desenvolvida por qualquer organização (comunidades / institutos / empresas), desde que sigam as especificações para a Java Virtual Machine.

# Garbage Collection (Coletor de lixo)



## Java e Orientação a Objetos

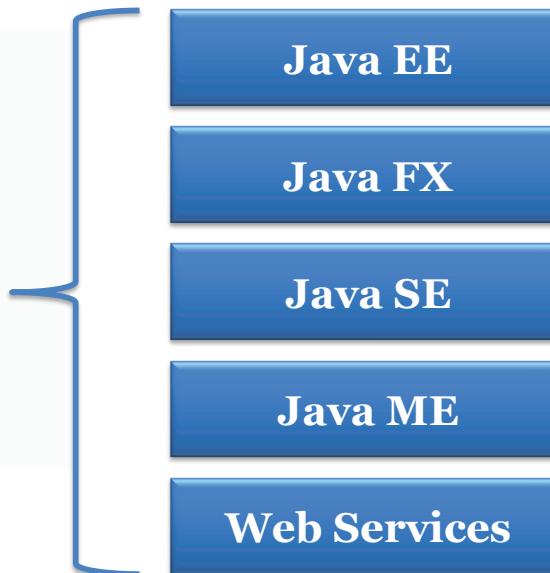
### Garbage Collector (Coletor de lixo)

O gerenciamento automático de memória pelo servidor é conhecido como garbage collector o serviço da JVM, que automaticamente libera blocos de memória que não serão mais usados em uma aplicação.

Os princípios básicos do coletor de lixo são encontrar resíduos de um programa que não serão mais acessados no futuro e deslocar os recursos utilizados, tornando a deslocação manual de memória desnecessária e geralmente proibindo tal prática. O coletor de lixo livra o programador de se preocupar com a liberação de recursos já não utilizados, o que pode consumir uma parte significativa do desenvolvimento do software. Também evita que o programador introduza erros no programa devido à má utilização de ponteiros.

Uma das vantagens desse sistema para o desenvolvedor é a não obrigação de manipular diretamente o acesso à memória do computador, pois periodicamente a memória utilizada por objetos não estão sendo referenciados há algum tempo é liberada. Porém o Garbage Collector funciona aleatoriamente sendo imprescindível que o desenvolvedor analise a utilização de memória do sistema.

# Divisão da Plataforma



## Java e Orientação a Objetos

### JSE (Java Standard Edition)

É o ambiente de desenvolvimento mais utilizado. Isso porque seu uso é voltado a PCs e servidores, onde há bem mais necessidade de aplicações. Além disso, pode-se dizer que essa é a plataforma principal, já que, de uma forma ou de outra, o JEE e o JME tem sua base aqui. Pode-se dizer também que esses ambientes de desenvolvimento são versões aprimoradas do JSE para as aplicações a que se propõem.

Por ser a plataforma mais abrangente do Java, o JSE é a mais indicada para quem quer aprender a linguagem.

### JEE (Java Enterprise Edition)

É a plataforma Java voltada para redes, internet, intranets e afins. Assim, ela contém bibliotecas especialmente desenvolvidas para o acesso a servidores, a sistemas de e-mail, a banco de dados, etc. Por essas características, o JEE foi desenvolvido para suportar uma grande quantidade de usuários simultâneos.

A plataforma JEE contém uma série de especificações, cada uma com funcionalidades distintas. Entre elas, tem-se:

- > **JDBC (Java Database Connectivity)** - utilizado no acesso a banco de dados;
- > **JSP (Java Server Pages)** - um tipo de servidor Web. Grosseiramente falando, servidores Web são as aplicações que permitem a você acessar um site na internet;
- > **Servlets** - para o desenvolvimento de aplicações Web, isto é, esse recurso “estende” o funcionamento dos servidores Web, permitindo a geração de conteúdo dinâmico nos sites.

## JME (Java Micro Edition)

É o ambiente de desenvolvimento para dispositivos móveis ou portáteis, como telefones celulares e palmtops. Como a linguagem Java já era conhecida e a adaptação ao JME não é complicada, logo surgiram diversos tipos de aplicativos para tais dispositivos, como jogos e agendas eletrônicas. As empresas saíram ganhando com isso porque, desde que seus dispositivos tenham uma JVM (Java Virtual Machine - Máquina Virtual Java), é possível, com poucas modificações, implementar os aplicativos em qualquer aparelho, sendo o único limite a capacidade do hardware.

A plataforma JME contém configurações e bibliotecas trabalhadas especialmente para a atuação em dispositivos portáteis. Assim, o desenvolvedor tem maior facilidade para lidar com as limitações de processamento e memória. Outro exemplo disso, é a configuração chamada CLDC (Connected Limited Device Configuration), destinada a dispositivos com recursos de hardware bastante limitados, como processadores de 16 bits e memórias com 512 KB de capacidade. Essa configuração contém uma JVM e um conjunto básico de bibliotecas que permite o funcionamento da aplicação Java em dispositivos com tais características.

## Java FX

É uma plataforma de software multimídia desenvolvida pela Oracle baseada em java para a criação e disponibilização de Aplicação Rica para Internet que pode ser executada em vários dispositivos diferentes.

A versão atual (JavaFX 2.1.0) permite a criação para desktop, browser e telefone celulares. TVs, videogames, Blu-rays players e outras plataformas estão sendo planejadas para serem adicionadas no futuro. JavaFX está totalmente integrado com o JRE - as aplicações JavaFX rodarão nos desktops e nos browsers que rodarem JRE e nos celulares que rodarem o JavaME.

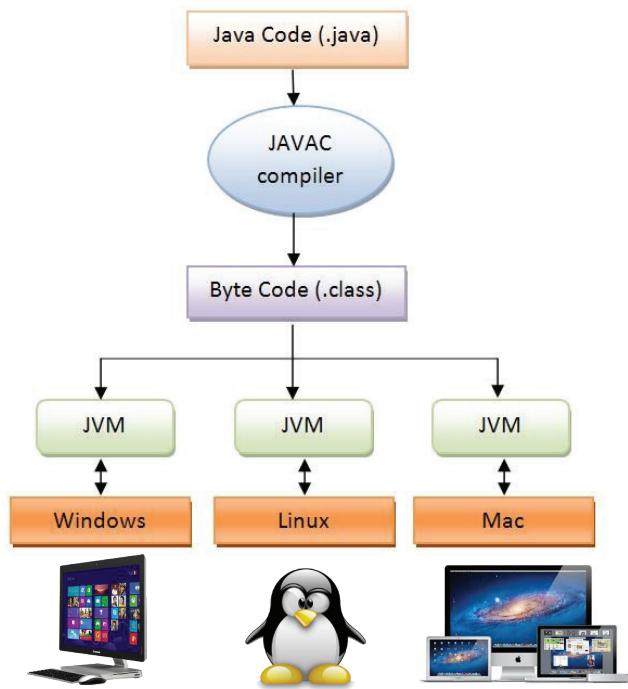
Para construir aplicações os desenvolvedores usam uma linguagem estática tipada e declarada chamada JavaFX Script. No desktop, por enquanto, existe somente para Windows XP, Windows Vista e Macintosh. A Oracle dedica-se para criar uma implementação no Linux também. Nos celulares, JavaFX é capaz de rodar em vários sistemas operacionais móveis como Android, Windows Mobile, e outros sistemas proprietários.

## WebServices

Webservices é uma solução utilizada na integração de sistemas e na comunicação entre aplicações diferentes. Com esta tecnologia é possível que novas aplicações possam interagir com aquelas que já existem e que sistemas desenvolvidos em plataformas diferentes sejam compatíveis. Os Webservices são componentes que permitem às aplicações enviar e receber dados em formato XML. Cada aplicação pode ter a sua própria “linguagem”, que é traduzida para uma linguagem universal, o formato XML.

Utilizando a tecnologia Web Service, uma aplicação pode invocar outra para efetuar tarefas simples ou complexas mesmo que as duas aplicações estejam em diferentes sistemas e escritas em linguagens diferentes. Por outras palavras, os WebServices fazem com que os seus recursos estejam disponíveis para que qualquer aplicação cliente possa operar e extrair os recursos fornecidos pelo Web Service.

# Fases do Programa Java



## Java e Orientação a Objetos

Quando escrevemos um código em alguma outra linguagem, como C, por exemplo, temos a seguinte situação:



Ou seja, o código fonte é compilado em código de máquina específico de uma determinada plataforma. Esse programa é compilado para conversar com um determinado sistema operacional, não conseguindo, na maioria das vezes, conversar com outros sistemas, tendo que ser recompilado.

Isso acontece porque, na maioria das vezes, a sua aplicação usa as bibliotecas do sistema operacional, e as bibliotecas são diferentes em diferentes sistemas operacionais (por exemplo, a biblioteca gráfica do Windows é diferente da biblioteca gráfica do Linux). Como você pode resolver esse problema?

O Java utiliza o conceito de Máquina Virtual, uma camada extra entre o sistema operacional e a aplicação, ganhando independência de sistema operacional. A VM gerencia memória, threads, pilha de execução, etc. Para cada SO há uma VM exclusiva, fazendo com que qualquer aplicativo Java rode em qualquer VM Java, independente do SO.

## Fases de um programa Java

Criação do código fonte: O código fonte criado para Java, em qualquer editor de texto, é salvo com a extensão “.java”, por exemplo, “Programa.java”.

Compilação do código fonte: Nessa fase, o compilador Java cria um novo arquivo chamado “Programa.class”, conhecido como arquivo de classe para “Programa.java”. Esse arquivo contém os bytecodes, que serão lidos pela JVM.

Conversão do bytecode em linguagem de máquina: Essa é a fase mais importante de um programa Java. Nos ambientes que oferecem o recurso de JIT (just-in-time), a máquina virtual responsável pela execução dos bytecodes resultantes da compilação do programa fonte realiza a tradução desse bytecode para código de máquina nativo enquanto o executa. No caso mais comum, cada trecho de código é traduzido no instante em que está para ser executado pela primeira vez, daí derivando o nome “just-in-time”.

# Aplicações mais comuns em Java



- Applets
- Servlets
- JSP
- JSF
- EJB
- JPA



## Java e Orientação a Objetos

### Applets

Uma applet é uma pequena aplicação executada em uma janela de uma aplicação (browser/appletviewer). Tem por finalidade estender as funcionalidades de browsers, adicionando som, animação, etc., provenientes de fontes (URLs) locais ou remotas, sendo que cada página web (arquivo.html) pode conter uma ou mais applets.

Applets sempre executam nos clientes web, nunca nos servidores. Por esta razão a carga das classes pode levar algum tempo. Toda applet é uma aplicação gráfica, não existindo portanto applets “modo texto”. A principal diferença entre uma “Java application” e uma “applet” é o fato de que a janela base da aplicação é derivada a partir da classe Applet (ou JApplet) e não a partir da classe Frame. Além disso, a parte da aplicação que instancia a classe Applet e relaciona-a com o browser é padrão e, portanto, não precisa ser descrita. Desta forma, applets não possuem a função “main( )”.

### Servlet

O nome “servlet” vem do inglês e dá uma ideia de servidor pequeno cujo objetivo basicamente é receber requisições HTTP, processá-las e responder ao cliente, essa resposta pode ser um HTML, uma imagem, etc.

O funcionamento se dá da seguinte forma:

- > Cliente (navegador) faz uma requisição HTTP ao servidor.
- > O servlet responsável trata a requisição e responde ao cliente de acordo.
- > O cliente recebe os dados e exibe.

## JSP

As páginas JSP, ou Java Server Pages, foram criadas para contornar algumas das limitações no desenvolvimento com Servlets: se em um Servlet a formatação da página HTML resultante do processamento de uma requisição se mistura com a lógica da aplicação em si, dificultando a alteração dessa formatação, em uma página JSP essa formatação se encontra separada da programação, podendo ser modificada sem afetar o restante da aplicação.

Assim, um JSP consiste de uma página HTML com alguns elementos especiais, que conferem o caráter dinâmico da página. Esses elementos podem tanto realizar um processamento por si, como podem recuperar o resultado do processamento realizado em um Servlet, por exemplo, e apresentar esse conteúdo dinâmico junto à página JSP.

## JSF

JavaServer Faces (JSF) é um framework MVC baseado em Java para a construção baseadas em componentes para aplicações web. Possui um modelo de programação dirigido a eventos, abstraindo os detalhes da manipulação dos eventos e organização dos componentes, permitindo que o programador se concentre na lógica da aplicação.

Foi formalizada como um padrão através do Java Community Process e faz parte da Java Platform, Enterprise Edition.

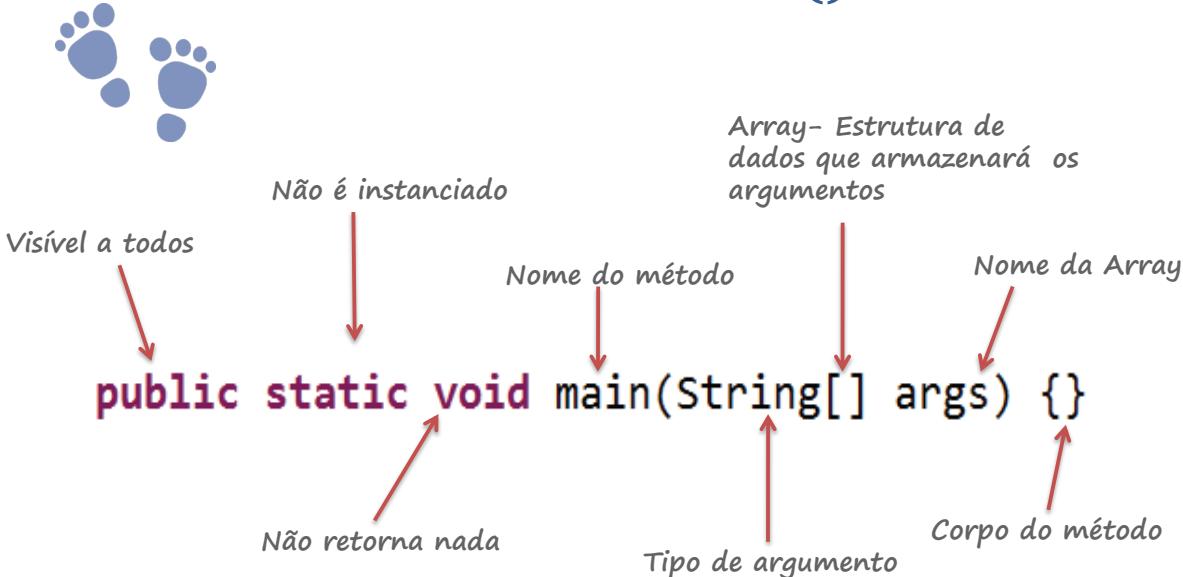
## EJB

Enterprise JavaBeans (EJB) é um componente da plataforma JEE que roda em um container de um servidor de aplicação. Seu principal objetivo consiste em fornecer um desenvolvimento rápido e simplificado de aplicações Java, com base em componentes distribuídos, transacionais, seguros e portáveis. Atualmente, na versão 3.1, o EJB tem seu futuro definido conjuntamente entre grandes empresas como IBM, Oracle e HP, como também por uma vasta comunidade de programadores numa rede mundial de colaboração sob o portal do JCP.

## JPA

Java Persistence API (ou simplesmente JPA) é uma API padrão do Java para persistência de dados que deve ser implementada por frameworks que desejem seguir tal padrão. A JPA define um meio de mapeamento objeto-relacional para objetos Java simples e comuns (POJOs), denominados beans de entidade. Diversos frameworks de mapeamento objeto/relacional como o Hibernate implementam a JPA. Também gerencia o desenvolvimento de entidades do Modelo Relacional usando a plataforma nativa Java SE e Java EE.

# O método main()



*Baby Steps*



Java e Orientação a Objetos

O método main é onde o programa inicia. Pode estar presente em qualquer classe. Os parâmetros de linha de comando são enviados para o array de Strings chamado args. Na execução de um programa Java, a JVM (Java Virtual Machine) tenta chamar o método main da classe que foi especificada. Quando declarado o método main como static permite que a JVM invoque o main sem criar uma instância da classe, ou seja, a classe é conhecida como classe principal ou classe testadora, que efetuará os testes e chamadas das classes para as execuções dos programas.

A JVM carrega a classe especificada pelo nome da classe que utiliza para invocar o método main (método principal/testador). Sempre que uma classe tiver esse método, é especificado que uma lista de Strings como argumentos de linha de comando, será passada para o aplicativo junto a JVM.

# Java e Orientação a Objetos

## Identificadores, palavras-chave e tipo

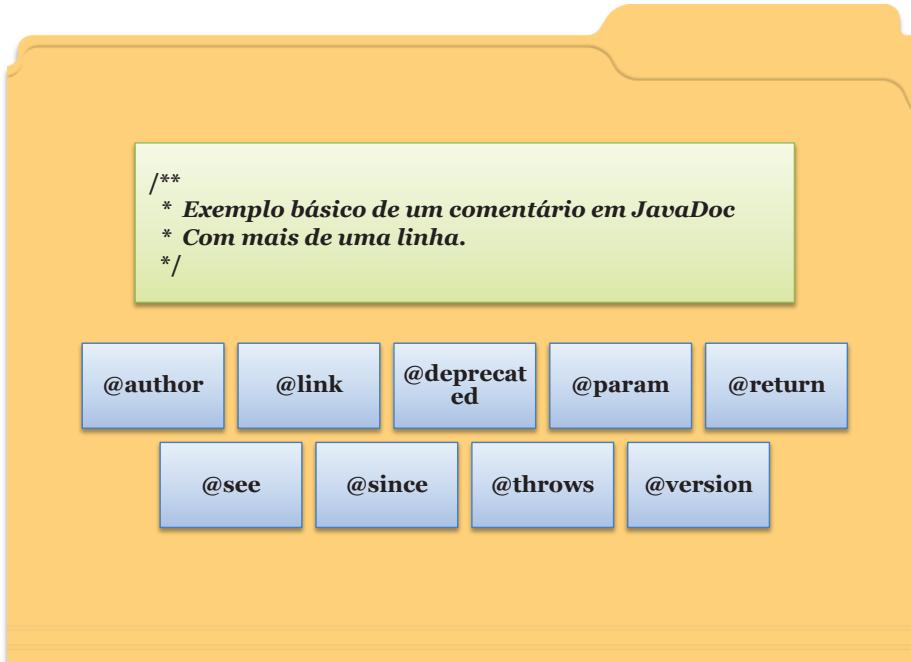


## Java e Orientação a Objetos

Assim como outras linguagens a linguagem de programação Java é definida por uma regra gramatical, que especifica como construções sintaticamente legais podem ser formadas usando os elementos da linguagem, e por uma definição semântica que especifica o significado das construções sintaticamente corretas.

Sintaxe e orientação a objetos são dois assuntos de grande importância na plataforma Java, veremos nesse módulo um pouco da sintaxe do Java.

# JavaDoc



## Java e Orientação a Objetos

É um gerador de documentação criado pela Sun Microsystems para documentar a API dos programas em Java, a partir do código-fonte. O resultado é expresso em HTML. É constituído, basicamente, por algumas marcações muitas simples inseridas nos comentários do programa.

Este sistema é o padrão de documentação de classes em Java, e muitas dos IDEs desta linguagem irão automaticamente gerar um Javadoc em HTML.

Um programa deve ser documentado com comentários e notas em lugares relevantes. Comentários tem como propósito a documentação, eles são ignorados pelo compilador.

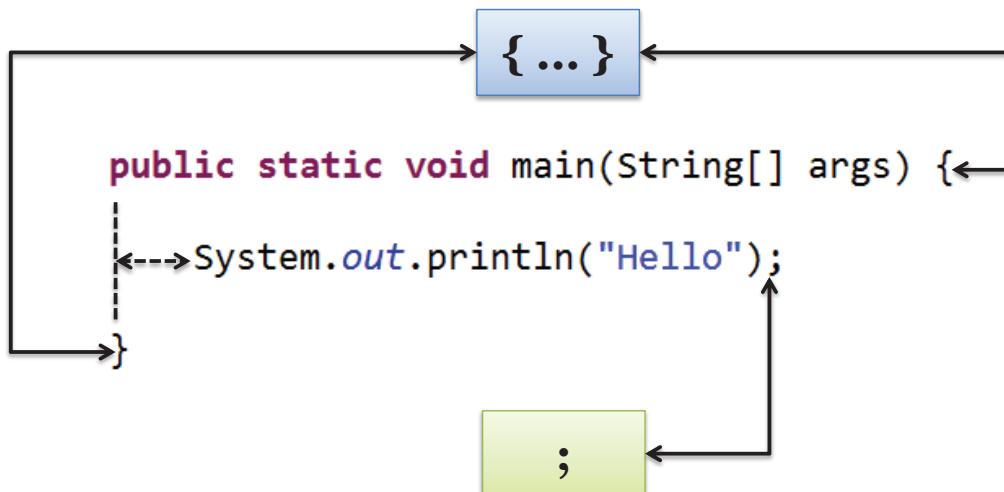
Para fazer um comentário em Java, você pode usar o // para comentar uma linha simples até o final, tudo após o // será ignorado pelo compilador, ou então usar o /\* \*/ para comentar múltiplas linhas, tudo o que estiver entre eles será ignorado pelo compilador.

Você pode criar comentários Javadoc começando-os com /\*\* e terminando-os com \*/, funciona como o exemplo anterior só que ao adicionarmos tags apropriadas (como @tag valor) podemos colocar mais informação em nossos comentários.

## Tags Java Doc

Tag	Descrição
<code>@author</code>	Nome do desenvolvedor
<code>@deprecated</code>	Marca o método como deprecated. Algumas IDEs exibirão um alerta de compilação se o método for chamado.
<code>@exception</code>	Documenta uma exceção lançada por um método — veja também <code>@throws</code> .
<code>@param</code>	Define um parâmetro do método. Requerido para cada parâmetro.
<code>@return</code>	Documenta o valor de retorno. Essa tag não deve ser usada para construtores ou métodos definidos com o tipo de retorno void.
<code>@see</code>	Documenta uma associação a outro método ou classe.
<code>@since</code>	Documenta quando o método foi adicionado à classe.
<code>@throws</code>	Documenta uma exceção lançada por um método.
<code>@version</code>	Exibe o número da versão de uma classe ou um método.

# Ponto-e-Vírgula, Blocos e Espaço



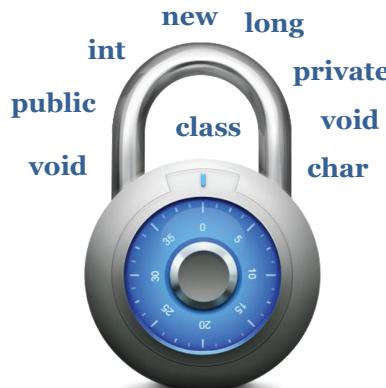
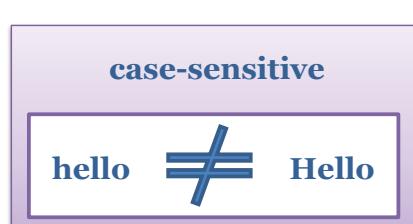
## Java e Orientação a Objetos

Uma sentença é constituída de uma ou mais linhas de código terminadas por ponto-e-vírgula (;).

Um bloco é uma ou mais sentenças cercadas por chaves ({}) de abertura e outra (}) de fechamento. Blocos de sentenças podem ser aninhados indefinidamente. Qualquer quantidade de espaços é permitida.

O uso do espaço em branco permite uma melhor visualização e legibilidade do código-fonte, facilitando uma possível manutenção.

# Identificadores e Palavras Reservadas



## Java e Orientação a Objetos

Identificadores são tokens (nomes) que utilizamos para representar as variáveis, classes, objetos, etc. Por exemplo, na Matemática utilizamos um nome para as incógnitas (x, y, z, etc.) que é o identificador daquela incógnita.

Utilizamos, em Java, as seguintes regras para criação do identificador:

- > Não pode ser uma palavra-reservada (palavra-chave);
- > Não pode ser true nem false - literais que representam os tipos lógicos (booleanos);
- > Não pode ser null - literal que representa o tipo nulo;
- > Não pode conter espaços em brancos ou outros caracteres de formatação;
- > Deve ser a combinação de uma ou mais letras e dígitos UNICODE-16. Por exemplo, no alfabeto latino, teríamos:

- > Letras de A a Z;
- > Letras de a a z;
- > Sublinha \_;
- > Cifrão \$;
- > Dígitos de 0 a 9;

**Observação 01:** caracteres compostos (acentuados) não são interpretados igualmente aos não compostos (não acentuados). Por exemplo, História e Historia não é o mesmo identificador.

**Observação 02:** Java é case-sensitive, ou seja, letras maiúsculas e minúsculas diferenciam os identificadores, ou seja, a é um identificador diferente de A, História é diferente de história, etc.

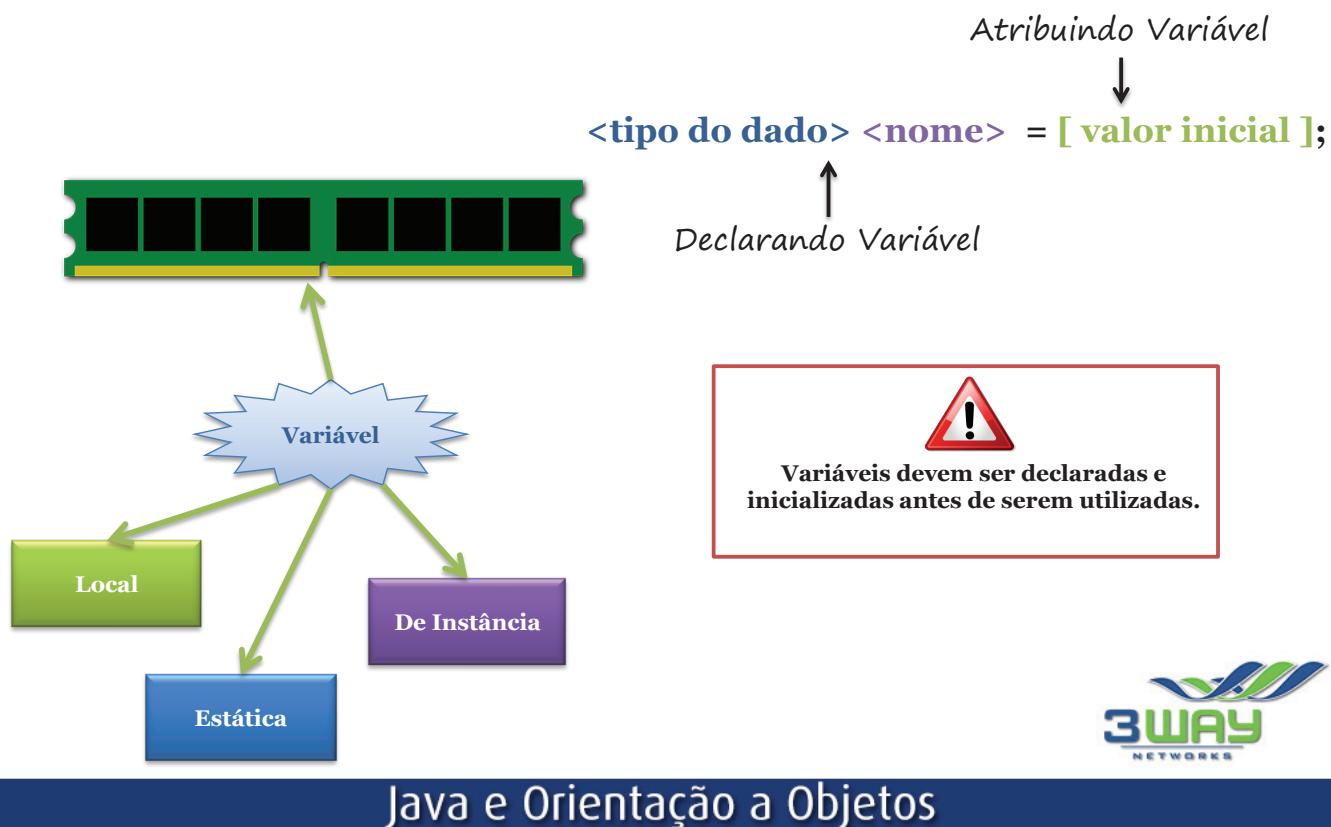
## Palavras Reservadas

Em programação, palavras-chave, ou palavras reservadas, são as palavras que não podem ser usadas como identificadores, ou seja, não podem ser usados como nome de variáveis, nome de classes, etc. Estas palavras são assim definidas ou porque já têm uso na sintaxe da linguagem ou porque serão usadas em alguns momentos, seja para manter compatibilidade com versões anteriores ou mesmo com outras linguagens. No caso do Java temos as seguintes palavras-chave.

**Palavras reservadas**

abstract	assert	boolean	break	byte
case	catch	char	class	continue
const	defaut	do	double	else
enum	extends	false	float	final
finally	for	goto	if	implements
import	instanceof	int	interface	long
native	new	null	package	private
protected	public	return	short	static
strictfp	super	switch	synchronized	this
throw	throws	transient	true	try
void	volatile	while		

# Variáveis, Declaração e Atribuição



## Java e Orientação a Objetos

Na programação, uma variável é um objeto (uma posição, frequentemente localizada na memória) capaz de reter e representar um valor ou expressão. Enquanto as variáveis só “existem” em tempo de execução, elas são associadas a “nomes”, chamados identificadores, durante o tempo de desenvolvimento.

Quando nos referimos à variável, do ponto de vista da programação de computadores, estamos tratando de uma “região de memória (do computador) previamente identificada cuja finalidade é armazenar os dados ou informações de um programa por um determinado espaço de tempo”. A memória do computador se organiza tal qual um armário com várias divisões. Sendo cada divisão identificada por um endereço diferente em uma linguagem que o computador entende.

O computador armazena os dados nessas divisões, sendo que em cada divisão só é possível armazenar um dado e toda vez que o computador armazenar um dado em uma dessas divisões, o dado que antes estava armazenado é eliminado. O conteúdo pode ser alterado, mas somente um dado por vez pode ser armazenado naquela divisão.

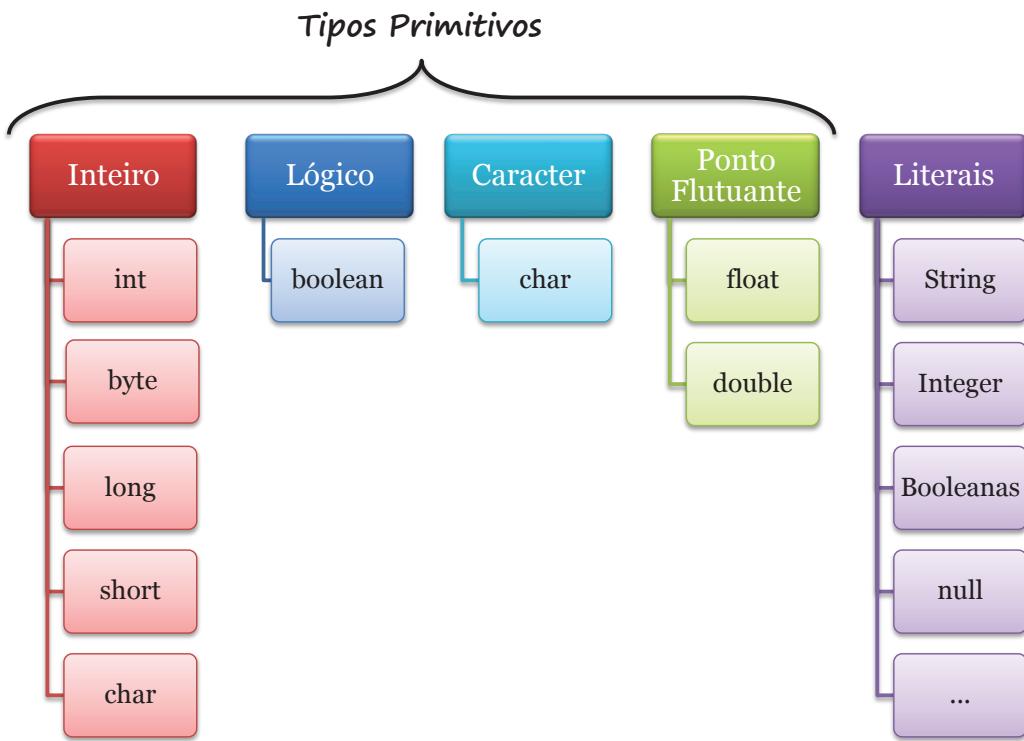
O computador identifica cada divisão por intermédio de um endereço no formato hexadecimal, e as linguagens de programação permitem nomear cada endereço ou posição de memória, facilitando a referência a um endereço de memória. Uma variável é composta por dois elementos básicos: o tipo, o identificador - um nome dado à variável para possibilitar sua utilização e o valor da variável.

Variáveis em Java podem ser de:

- **Instância:** é um espaço na memória usado para armazenar o estado de um objeto.
- **Estática ou de classe:** que pertencem à classe e não a uma instância de um objeto.
- **Local:** também chamadas de variáveis de método, que são declaradas em blocos ou métodos e são alocadas para cada invocação de método ou bloco.

JAVA é muito exigente com relação ao tipo de dado, é uma linguagem fortemente tipada. Por exemplo, o compilador não permitirá que você use o operador de atribuição (=) para inserir uma referência de objeto de uma classe **Integer** em uma variável referência do tipo da classe **String**, nem permitirá que se atribua um valor do tipo **float** a uma variável do tipo **int**.

# Tipos de Dados



## Java e Orientação a Objetos

### Tipos Primitivos | Tipo lógico

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` e 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.

### Tipo inteiro

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.

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`.

### Tipo caracter

O tipo de dado caracter representa um CARACTER UNICODE, conjunto de caracteres de 16 bits sem sinal, em substituição ao conjunto de caractere ASCII de 8 bits. Unicode permite o uso de símbolos e caracteres especiais das línguas.

Para usar um literal de caractere devemos cercar o caractere entre delimitadores de aspas simples. Por exemplo, a letra A, é representada como o 'A'. Ou podemos ainda usar o código hexadecimal do caractere UNICODE, por exemplo '\u0041', esta é a representação hexadecimal do caractere unicode'A'.

## Tipo 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. Ambos seguem a 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 portáveis. Esses dados serão aceitos por qualquer plataforma, independendo 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).

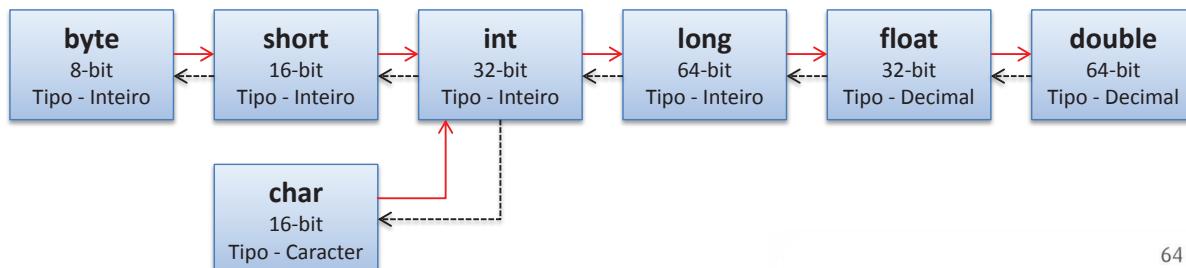
## Tipos Literais

Um literal é um valor constante, este valor pode ser numérico (inteiros e ponto-flutuantes), caracteres, booleanos ou cadeia de caracteres (String). Há ainda o literal null que representa uma referência nula (sem destino).

### Valores possíveis

Exemplo	Taman-ho	Valor pa-drão	Maior	Menor	Primitivo	Tipos
byte ex1 = (byte)1;	8 bits	0	127	-128	byte	Inteiro
short ex2 = (short)1;	16 bits	0	32767	-32768	short	
int ex3 = 1;	32 bits	0	2.147.483.647	-2.147.483.648	int	
long ex4 = 1l;	64 bits	0	9.223.327.036 .854.770.000	9.223.372.036 .854.770.000	long	
float ex5 = 5.5of;	32 bits	0	3.40282347E + 38	-1,4024E-37	float	
double ex6 = 10.2od; ou double ex6 = 10.20;	64 bits	0	1.79769313486 231570E + 308	-4,94E-307	double	Ponto Flutuante
char ex7 = 194; ou char ex8 = 'a';	16 bits	\0	65535	0	char	Caractere
boolean ex9 = true;	1 bits	false	true	false	boolean	Booleano

# Casting de Tipos Primitivos



→ Casting implícito (Automático)  
 ----> Casting explícito (Requer a utilização de cast)



## Java e Orientação a Objetos

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.

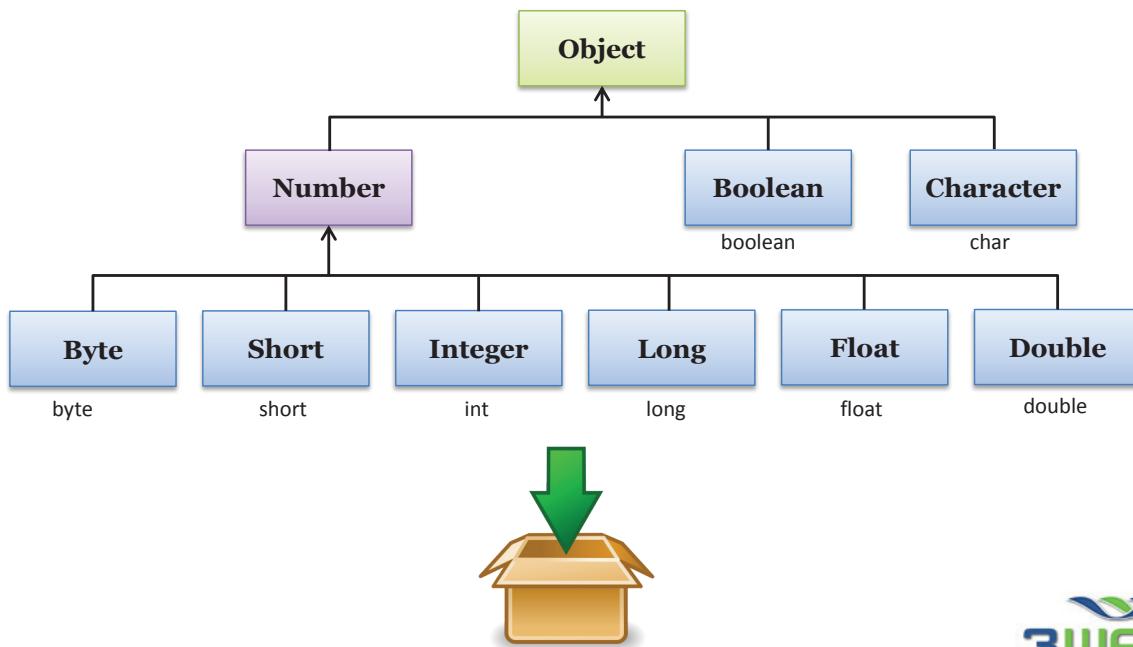
É possível fazer conversões de tipos de ponto flutuante para inteiros, e inclusive entre o tipo caractere, porém estas conversões podem ocasionar a perda de valores, quando se molda um tipo de maior tamanho, como um double dentro de um int.

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:

double	float	long	int	char	short	byte	DE\ PARA
Implícito	Implícito	Implícito	Implícito	char	Implícito		byte
Implícito	Implícito	Implícito	Implícito	char		byte	short
Implícito	Implícito	Implícito	Implícito		short	byte	char
Implícito	Implícito	Implícito		char	short	byte	int
Implícito	Implícito		int	char	short	byte	long
Implícito		long	int	char	short	byte	float
	float	long	int	char	short	byte	double

# Classes Wrapper (Empacotadoras)



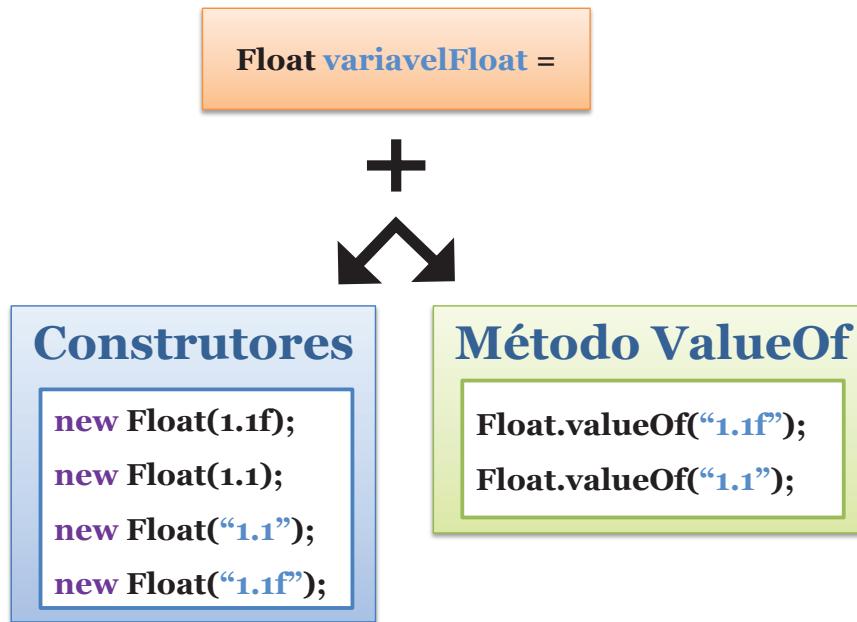
## Java e Orientação a Objetos

As classes wrapper em Java têm dois propósitos: prover o mecanismo de envolver valores primitivos em um objeto para que estes possam realizar atividades exclusivas de objetos (como ser adicionados em coleções ou ser retornado de um método que tenha um objeto como valor de retorno), e fornecer uma variedade de funções para os primitivos, como conversões para String ou para diferentes bases (binária, octal e hexadecimal).

Existe uma classe wrapper para cada tipo primitivo. Uma vez declarado o valor de um objeto wrapper, este nunca mais poderá ser mudado. Por exemplo, para o int existe a classe Integer, para o char existe a classe Character, para o long existe a classe Long, etc.

Classes derivadas da subclasse Number possuem vários métodos para devolverem um tipo primitivo, tais como: `byteValue()`, `shortValue()`, `intValue()`, `longValue()`, `doubleValue()`, `floatValue()`.

# Construtores e método valueOf



## Java e Orientação a Objetos

Todas as classes wrapper, exceto Character, fornecem dois construtores: um com um argumento do tipo primitivo, e outro com um argumento do tipo String.

A classe Character fornece apenas um construtor, que leva um char como argumento.

Os construtores da classe Boolean podem receber os valores true e false ou uma String correspondente a esses valores. Se a String é true (case-sensitive) o retorno será true, se for qualquer outro valor o retorno será false.

Quando as classes wrapper levam uma String como argumento e o valor não correspondem a uma literal válida, ou seja, não pode ser convertido para o primitivo apropriado, é lançado um erro.

### Métodos valueOf()

Os dois métodos static `valueOf`, fornecidos pela maioria das classes wrapper, trazem uma nova abordagem na criação de objetos wrapper.

Os dois métodos têm uma representação String do tipo primitivo como primeiro argumento, e o segundo método tem um argumento adicional int que indica em qual base (binária, octal ou hexadecimal) o primeiro argumento está representado.

# AutoBoxing – Boxing and Unboxing

```

int i = 10;
Integer iRef = new Integer(i); // Boxing Explicito
int j = iRef.intValue(); // Unboxing Explicito
iRef = i; // Boxing Automatico
j = iRef; // Unboxing Automatico
  
```



## Java e Orientação a Objetos

A criação das classes wrapper resolveram vários problemas, porém a conversão entre tipo primitivo para objeto (boxing) e objeto para primitivo (unboxing) se tornou uma tarefa trivial e tediosa.

A partir do JDK 1.5, a Sun resolveu este problema através do autoboxing, que consiste no boxing automático, ou seja, na conversão automática pelo compilador de tipo primitivo para objeto.

Além do autoboxing, existe o autounboxing, que é a conversão automática de objeto para tipo primitivo.

### BoxingConversion

O Boxing é a conversão de tipos primitivos em seu respectivo Wrapper

É óbvio que se você tentar realizar um BoxingConversion de um tipo primitivo para um Wrapper errado você terá um erro de compilação.

### UnboxingConversion

O UnboxingConversion é quando você deseja fazer o inverso do boxing, ou seja, deseja converter um objeto para um tipo primitivo.

# Java e Orientação a Objetos

## Operadores



## Java e Orientação a Objetos

Em Java temos diferentes tipos de operadores. Existem **operadores aritméticos**, **operadores relacionais**, **operadores lógicos** e **operadores condicionais**. Estes operadores obedecem a uma **ordem de precedência** para que o compilador saiba qual operação executar primeiro, no caso de uma sentença possuir grande variedade destes.

# Operadores Aritméticos



*	var1 * var2	Multiplicação
+	var1 + var2	Adição
-	var1 - var2	Subtração
%	var1 % var2	Resto da divisão
/	var1 / var2	Divisão



## Java e Orientação a Objetos

Os operadores aritméticos são operadores binários, ou seja, funcionam com dois operandos. Por exemplo, a expressão “a + 1” contém o operador binário “+” (mais) e os dois operandos “a” e “1”.

### Operadores Aritméticos

Operação	Operador	Expressão Java
Adição	+	A + 1
Subtração	-	B - 2
Multiplicação	*	a * b
Divisão	/	a / b
Resto	%	b % a

Observação importante: a divisão de inteiros produz um quociente do tipo inteiro, quando possuímos o número 1 maior que o número 2, por exemplo, a expressão “9/6” o resultado é interpretado como 1 e a expressão “23/8” é avaliada como 2, ou seja a parte fracionária em uma divisão de inteiros é descartada, não contendo nenhum arredondamento.

O módulo (%) fornece o resto da divisão, na expressão “x % y”, o resultado é o restante depois que x é dividido por y, sendo assim na expressão “7 % 4” o resultado é 3 e “17 % 5” o resultado produz 2. Esse operador é mais utilizado com operandos inteiros, mas também pode ser utilizado com outros tipos.

# Operadores Relacionais



<b>&gt;</b>	var1 > var2	Maior que
<b>&gt;=</b>	var1 >= var2	Maior ou igual
<b>&lt;</b>	var1 < var2	Menor que
<b>&lt;=</b>	var1 <= var2	Menor ou igual
<b>==</b>	var1 == var2	Igual
<b>!=</b>	var1 != var2	Diferente



## Java e Orientação a Objetos

Estabelecem uma relação entre dois elementos, retornando verdadeiro ou falso.

Vejamos uma tabela com os símbolos dos operadores relacionais e seus significados:

### Operadores Relacionais

Operador	Em Java	Setença	Ex. em algoritmos	Resultado
=	==	Igual a	4==4	Verdadeiro
>	>	Maior que	15>7	Verdadeiro
<	<	Menor que	80<72	Falso
>=	>=	Maior ou igual a	13>=13	Verdadeiro

Aqui podemos notar que só há uma diferença no operador de igualdade e no que calcula a diferença. No caso do operador de igualdade o motivo de ser utilizado dois iguais (**==**) ao invés de somente um é porque na maioria das linguagens de programação o sinal de igual é na verdade um sinal de atribuição.

O operador de diferença em português estruturado também difere da maioria das linguagens de programação, pois é composto de um sinal de menor e um sinal de maior (**≠**) em contraste com o sinal de exclamação seguido de um sinal de igual (**!=**) utilizado nas mesmas.

# Operadores Lógicos



**&&** var1 &&var2      'E' lógico (AND)

**&** var1 & var2      'E' binário

**||** var1 || var2      'OU' lógico (OR)

**|** var1 | var2      'OU' binário

**^** var1 ^ var2      'OU' exclusivo binário

**!** var1 ! var2      Negação (NOT)



## Java e Orientação a Objetos

Em algoritmos normalmente usamos os operadores lógicos E, OU e NAO. Mas na maioria das linguagens de programação temos outras formas de representá-los.

Operadores lógicos avaliam um ou mais operandos lógicos que geram um único valor final **true** ou **false** como resultado da expressão permitindo construir expressões lógicas.

São seis os operadores lógicos:

### Operadores Lógicos

Operador	Descrição	Exemplo
<b>&amp;&amp;</b>	Retorna true se ambos operandos forem true.	true && true
<b>&amp;</b>	Retorna true se ambos operandos forem true, avaliando as duas condições.	true & true
<b>  </b>	Retorna true se algum dos operandos for true.	false    true
<b> </b>	Retorna true se um dos operandos for true, avaliando ambos.	true   true

É importante dizer que quando utilizamos o operador de negação como exclamação (!) é preciso utilizar parênteses, caso contrário poderemos receber um erro do interpretador. Isso porque ele não sabe se você quer negar o número logo após o sinal (!) de negação ou a expressão inteira.

# && ( e lógico ) e & ( e binário )

Condição 1	Operador	Condição2	Resultado
true	&&	true	true
false	&&	true	false
true	&&	false	false
false	&&	false	false



A diferença básica do operador `&&` para `&` é que o `&&` suporta uma avaliação de curto-circuito (ou avaliação parcial), enquanto que o `&` não.



## Java e Orientação a Objetos

A diferença básica do operador `&&` para `&` é que o `&&` suporta uma avaliação de curto-circuito (ou avaliação parcial), enquanto que o `&` não.

O Operador `&` ULA (unidade lógica e aritmética) utiliza o processador para fazer operações bit a bit, sendo, portanto sua execução extremamente rápida.

O Operador `&&` é executado da esquerda para a direita utilizando saltos condicionais para obter o resultado. Só avalia os valores seguintes caso os anteriores não sejam suficientes para deduzir o resultado.

**Conclusão:** O operador `&&` irá avaliar a primeira expressão, se esta for false já será retornado imediatamente false, sem avaliar as expressões posteriores. Já o operador `&` sempre avalia as duas partes da expressão, mesmo que a primeira tenha o valor `false`.

# || ( ou lógico ) e | ( ou binário )



Condição 1	Operador	Condição2	Resultado
true		true	true
false		true	true
true		false	true
false		false	false



A diferença básica entre os operadores || e |, é que, semelhante ao operador &&, o || também suporta a avaliação parcial.



## Java e Orientação a Objetos

A diferença básica entre os operadores || e |, é que, semelhante ao operador &&, o || também suporta a avaliação parcial.

O Operador | ULA (unidade lógica e aritmética) utiliza o processador para fazer operações bit a bit, sendo, portanto sua execução extremamente rápida.

O Operador || é executado da esquerda para a direita utilizando saltos condicionais para obter o resultado. Só avalia os valores seguintes caso os anteriores não sejam suficientes para deduzir o resultado.

**Conclusão:** O operador || irá avaliar a primeira expressão, se esta for true já será retornado imediatamente true, sem avaliar as expressões posteriores. Já o operador | sempre avalia as duas partes da expressão, mesmo que a primeira tenha o valor true.

# $\wedge$ ( ou exclusivo binário )



Condição 1	Operador	Condição2	Resultado
true	$\wedge$	true	false
false	$\wedge$	true	true
true	$\wedge$	false	true
false	$\wedge$	false	false



## Java e Orientação a Objetos

O resultado de uma expressão usando o operador **ou exclusivo** binário terá um valor **true** somente se uma das expressões for verdadeira e a outra falsa. Note que ambos os operandos são necessariamente avaliados pelo operador  $\wedge$ .

O operador de bits  $\wedge$  (OU Exclusivo/XOR sobre bits) da linguagem Java é usado quando queremos comparar os bits individuais de dois valores inteiros (inteiros) e produzir um terceiro resultado. Os bits no resultado serão configurados como 1 SE SOMENTE UM dos bits nos dois outros valores for 1. Em caso contrário os bits são configurados como 0 (no caso em que ambos os bits correspondentes estão configurados como 1 ou 0).

# ! ( Negação )

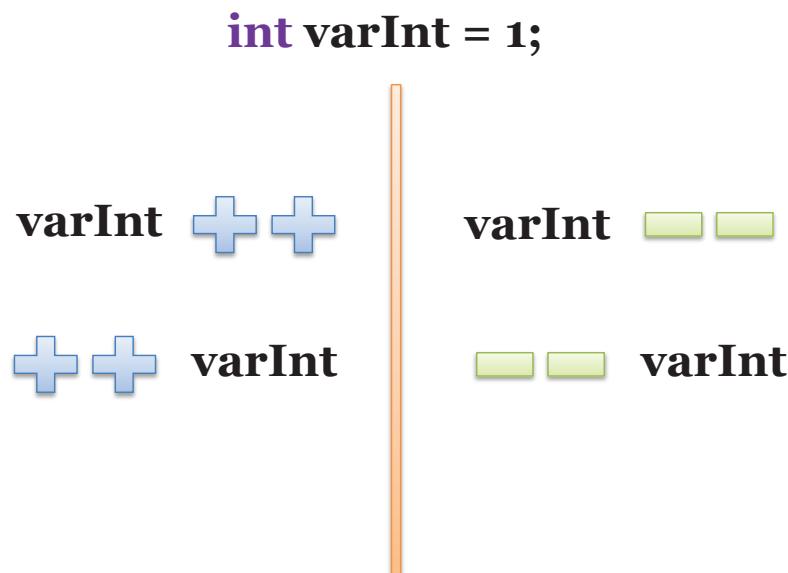
Condição	Operador	Resultado
true	!	false
false	!	true



## Java e Orientação a Objetos

O operador de negação **inverte o resultado lógico de uma expressão**, variável ou constante, ou seja, o que era verdadeiro será falso e vice-versa.

# Operadores de Incremento e Decremento



## Java e Orientação a Objetos

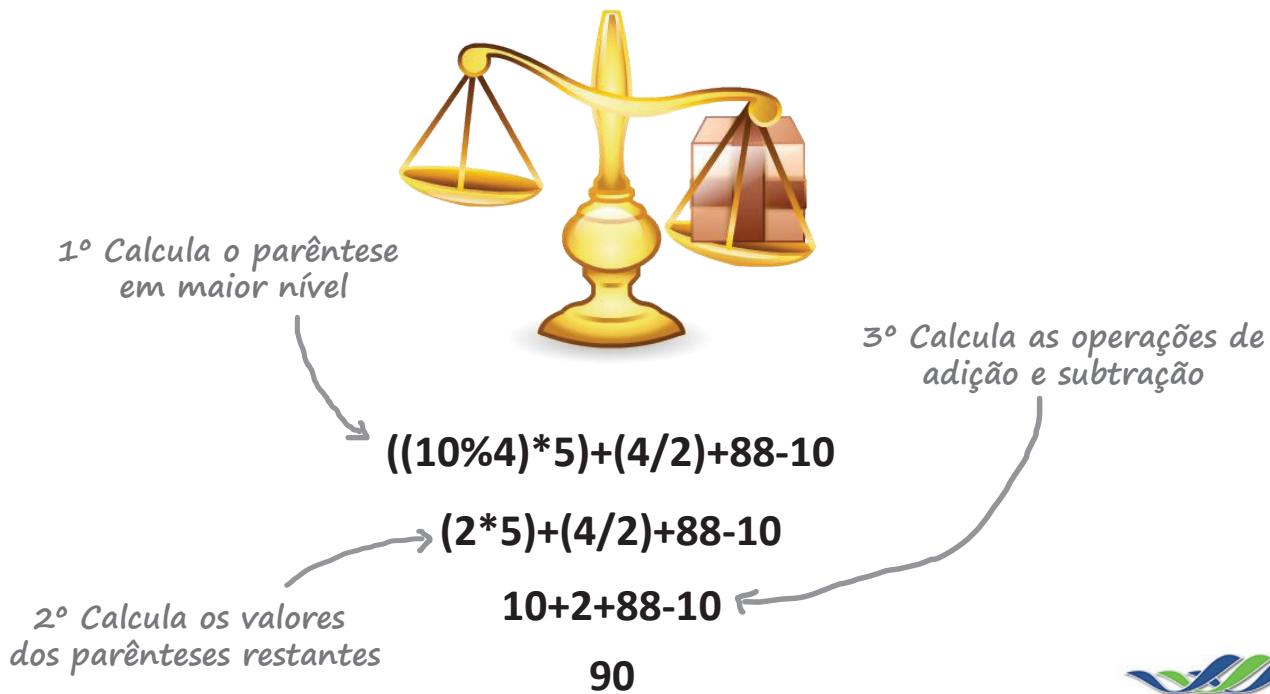
Além dos operadores aritméticos básicos, Java dá suporte ao operador unário de **incremento** (++) e ao operador unário de **decremento** (--). Operadores ++ ou --, respectivamente aumentam ou diminuem em 1 o valor da variável.

Operadores de Incremento e Decremento		
Operador	Uso	Descrição
++	op++	Incrementa op em 1; avalia a expressão antes do valor ser acrescido
++	++op	Incrementa op em 1; incrementa o valor antes da expressão ser avaliada
--	op--	Decrementa op em 1; Avalia a expressão antes do valor ser decrescido

Como visto na tabela acima, os operadores de **incremento** e **decremento** podem ser usados tanto **antes** como **após** o operando. E sua utilização dependerá disso. Quando usado antes do operando, provoca acréscimo ou decréscimo de seu valor antes da avaliação da expressão em que ele aparece.

Quando utilizado depois do operando, provoca, na variável, acréscimo ou decréscimo do seu valor após a avaliação da expressão na qual ele aparece.

# Precedência de Operadores



## Java e Orientação a Objetos

A precedência serve para indicar a ordem na qual o compilador interpretará os diferentes tipos de operadores, para que ele sempre tenha como saída um resultado coerente e não ambíguo. Na tabela os operadores na mesma linha possuem mesma precedência, a coluna ordem define a precedência do maior (1) para o menor (14).

Precedência de Operadores

Ordem	Operador	Descrição
1	<code>. [] () (tipo)</code>	Máxima precedência: separador, indexação, parâmetros, conversão de tipo.
2	<code>+ - ~ ! ++ --</code>	Operador unário: positivo, negativo, negação (inversão bit a bit), não (lógico), incremento, decremento.
3	<code>* / %</code>	Multiplicação, divisão e módulo (inteiros)
4	<code>+ -</code>	Adição, subtração
5	<code>&lt;&lt; &gt;&gt;</code> <code>&gt;&gt;&gt;</code>	Translação (bit a bit) à esquerda, direita sinalizada, e direita não sinalizada (o bit de sinal será 0)
6	<code>&lt; &lt;= &gt;= &lt;</code>	Operador relacional: menor, menor ou igual, maior ou igual, maior.
7	<code>== !=</code>	Igualdade: igual, diferente.

## Precedência de Operadores

Ordem	Operador	Descrição
8	&	Operador lógico <b>e</b> bit a bit
9	^	Ou exclusivo ( <b>xor</b> ) bit a bit
10		Operador lógico <b>ou</b> bit a bit
11	&&	Operador lógico <b>e</b> condicional
12		Operador lógico <b>ou</b> condicional
13	? :	Condisional: if-then-else compacto
14	= op =	Atribuição (Combinada)

**Considere a expressão:**

**1 | 2 ^ 3 \* 2 & 13 | 2**

A análise é feita da esquerda para a direita.

O primeiro cálculo a ser executado é **3 \* 2**, que resulta em **6**.

Em seguida, o resultado anterior é comparado com o **13**, uma operação **& (E): 6 & 13**, que resulta em **4**.

Agora é feita uma operação de **^ (Ou exclusivo)** com o **2** (mais a esquerda) e o **4** da operação anterior. O resultado é **6**.

É executada, então, a operação **| (Ou)** mais a esquerda: **1 | 6** que resulta em **7**.

Por último, é executada a operação de **| (Ou): 7 | 2** que também resulta em **7**.

A mesma expressão acima poderia ser escrita da seguinte forma:

**((1 | (2 ^ ((3 \* 2) & 13))) | 2)**

# Operador Condicional (?:)

Expressão Booleana  
retorna true ou false

**exp1** ? **exp2** : **exp3**

Se o valor de **exp1** for **verdadeiro**, então o resultado será **exp2**, caso contrário, **exp3**



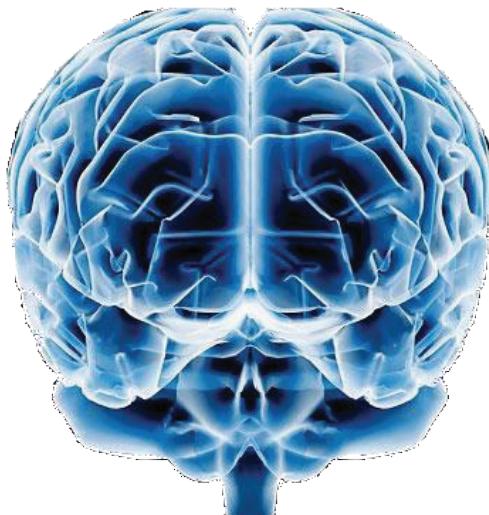
## Java e Orientação a Objetos

O operador condicional é também chamado de operador ternário. Isto significa que ele tem três argumentos que juntos formam uma única expressão condicional.

Retorna um entre dois valores de acordo com o resultado de uma expressão booleana. Se for true retorna o valor após o ?, caso contrário retorna o valor depois do : .

# Java e Orientação a Objetos

## Estruturas de Controle



### Java e Orientação a Objetos

Estruturas de controle de decisão são instruções em linguagem Java que permite que blocos específicos de código sejam escolhidos para serem executados, redirecionando determinadas partes do fluxo do programa.

Um programa de computador é uma sequência de instruções organizadas de forma a produzir a solução de um determinado problema; o que representa uma das habilidades básicas da programação. Naturalmente, as instruções de um programa são executadas em sequência, o que se denomina fluxo sequencial de execução. Mas, em inúmeras circunstâncias, é necessário executar as instruções de um programa em uma ordem diferente da estritamente sequencial. Tais situações são caracterizadas pela necessidade da repetição de instruções individuais ou conjuntos de instruções, e também pelo desvio do fluxo de execução, tarefas que podem ser realizadas por meio das “estruturas de controle” da linguagem.

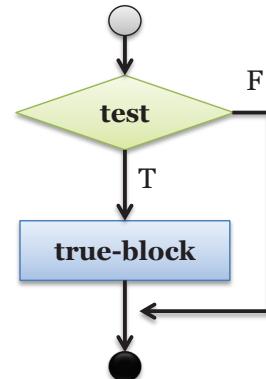
# Estrutura de decisão (if)

```

int nota = 68;
if (nota > 60) {
    System.out.println("Parabéns!");
    System.out.println("Você Passou!");
}
  
```

*Expressão Booleana  
test*

*Bloco de código da expressão true  
true-block*



## Java e Orientação a Objetos

A declaração **if** especifica que uma instrução ou bloco de instruções seja executado se, e somente se, uma expressão lógica for verdadeira.

A declaração **if** possui as seguintes formas:

### Declaração if

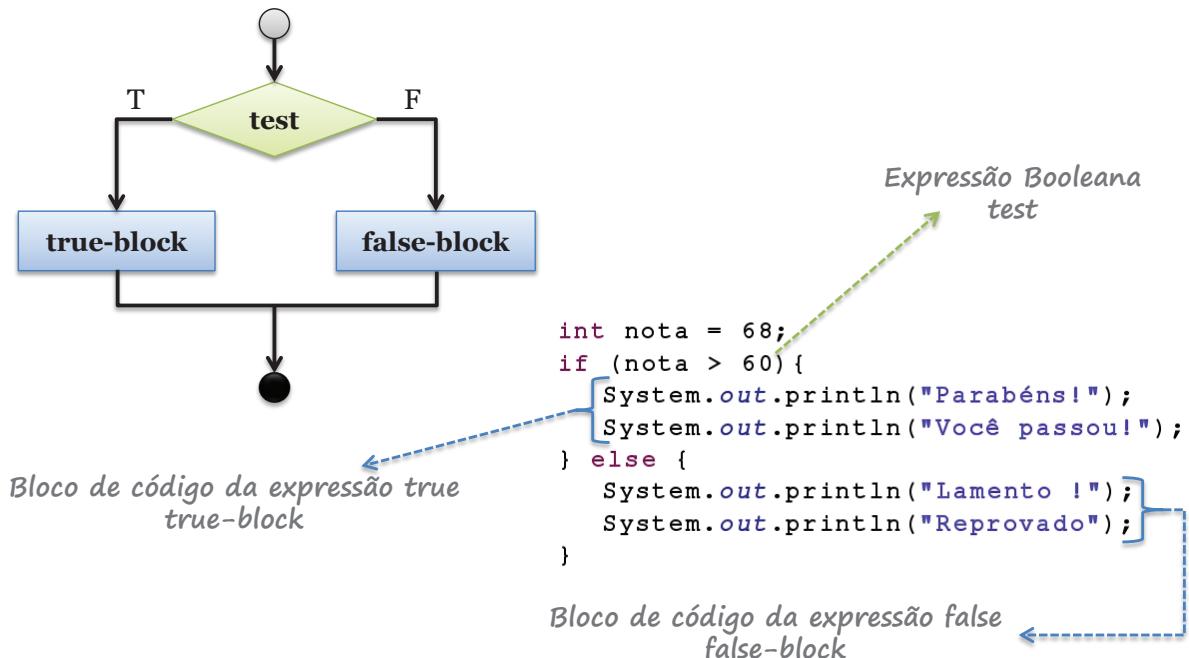
if (<expressão lógica>)  
<sentença 1>

```

if (<expressão lógica>) {
    <sentença 1>
    <sentença 2>
}
  
```

Onde, **expressão\_lógica** representa uma expressão ou variável lógica (que retorna **false** ou **true**).

# Estrutura de decisão (if-else)



## Java e Orientação a Objetos

A declaração **if-else** é usada quando queremos executar determinado conjunto de instruções se a condição for verdadeira e outro conjunto se a condição for falsa.

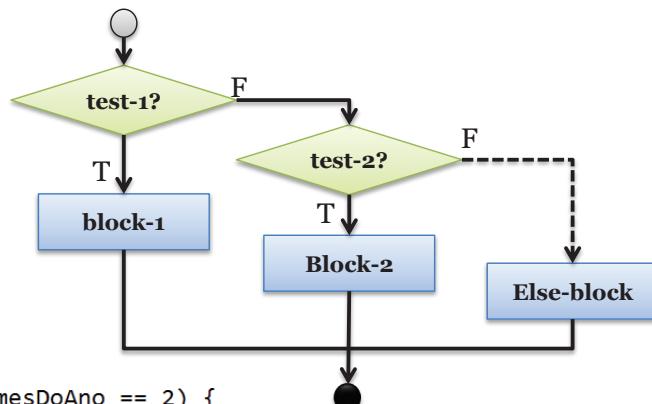
A declaração **if-else** possui as seguintes formas:

### Declaração if-else

```
if (<expressão lógica>)
<sentença 1>
else
<sentença 2>
```

```
if (<expressão lógica>) {
<sentença 1>
<sentença 2>
} else {
<sentença 3>
<sentença 4>
}
```

# Estrutura de decisão (if-else-if)



```

public static void main(String[] args) {
    int mesDoAno = 13;

    if (mesDoAno == 12 || mesDoAno == 1 || mesDoAno == 2) {
        System.out.println("Verão");
    } else if (mesDoAno == 3 || mesDoAno == 4 || mesDoAno == 5) {
        System.out.println("Outono");
    } else if (mesDoAno == 6 || mesDoAno == 7 || mesDoAno == 8) {
        System.out.println("Inverno");
    } else if (mesDoAno == 9 || mesDoAno == 10 || mesDoAno == 1) {
        System.out.println("Primavera");
    } else {
        System.out.println("Mês não é válido " + mesDoAno);
    }
}
  
```



## Java e Orientação a Objetos

A declaração **else** pode conter outra estrutura **if-else**. Este aninhamento de estruturas permite ter decisões lógicas muito mais complexas.

A declaração **if-else-if** possui a seguinte forma:

### Declaração if-else-if

```

if (<expressão_lógica>)
    <sentença 1>
else if (<expressão_lógica>)
    <sentença 2>
else
    <sentença 3>
  
```

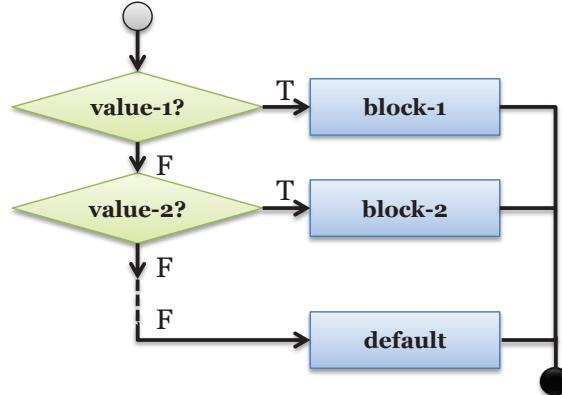
Podemos ter várias estruturas **else-if** depois de uma declaração **if**. A estrutura **else** é opcional e pode ser omitida. No exemplo mostrado acima, se a primeira condição é verdadeira, o programa executa a **<sentença 1>** e salta as outras instruções. Caso contrário, se a condição é falsa, o fluxo de controle segue para a análise da segunda condição. Se esta for verdadeira, o programa executa a **<sentença 2>** e salta a **<sentença 3>**. Caso contrário, se a segunda condição é falsa, então a **<sentença 3>** é executada.

# Estrutura de decisão (switch)

```

int mesDoAno = 13;

switch (mesDoAno) {
    case 12:
    case 1:
    case 2:
        System.out.println("Verão");
        break;
    case 3:
    case 4:
    case 5:
        System.out.println("Outono");
        break;
    case 6:
    case 7:
    case 8:
        System.out.println("Inverno");
        break;
    case 9:
    case 10:
    case 11:
        System.out.println("Primavera");
        break;
    default:
        System.out.println("Mês não é válido " + mesDoAno);
        break;
}
    
```



## Java e Orientação a Objetos

Outra maneira de indicar uma condição é através de uma declaração **switch**. A construção **switch** permite que uma única variável tenha múltiplas possibilidades de avaliação. O **switch** trabalha sobre os tipos primitivos **byte**, **short**, **char** e **int**, ele também pode operar sobre tipos **enumerados** (a serem vistos posteriormente) sobre classes empacotadoras **Character**, **Byte**, **Short** e **Integer**.

A declaração **switch** possui a seguinte forma:

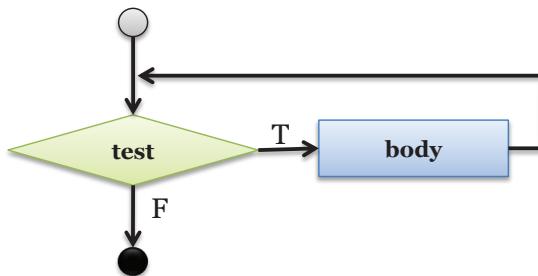
### Declaração switch

```

switch (<variável_do_tipo_permitido>) {
    case <valor 1>:
        [instrução1;]
        [...]
        [break;]
    case <valor n>:
        [instrução1;]
        [...]
        [break;]
    default:
        [instrução1;]
        [...]
        [break;]
}
    
```

O programa executa todas as instruções a partir do primeiro **case**, até encontrar uma instrução **break**, que interromperá a execução do **switch**, nada após o **break** será executado.

# Estrutura de repetição (while)



```

public static void main(String[] args) {
    int contador = 0;
    while (contador < 10) {
        System.out.println(contador);
        contador++;
    }
}
  
```

Expressão Booleana  
test

Corpo da estrutura de repetição  
body



## Java e Orientação a Objetos

A declaração **while** executa repetidas vezes um bloco de instruções enquanto a expressão lógica (condição) resultar em **true** (boolean).

O problema com estruturas de repetição, principalmente com **while**, é o que chamamos de looping infinito. Damos esse nome ao fato de que o programa fica repetindo a mesma sequência de códigos esperando por um resultado que nunca irá acontecer.

Portanto, é imprescindível que uma determinada variável seja modificada de acordo com cada loop.

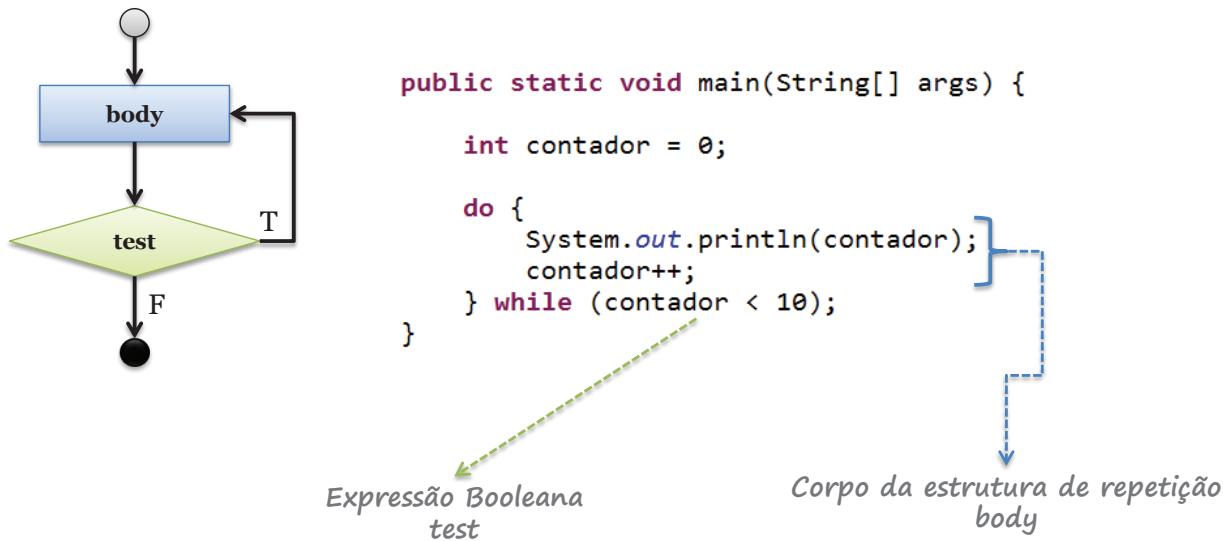
A declaração **while** possui a seguinte forma:

### Declaração while

```

while (expressão_lógica) {
    instrução1;
    instrução2;
}
  
```

# Estrutura de repetição (do-while)



## Java e Orientação a Objetos

A declaração **do-while** é similar ao **while**. As instruções dentro do laço **do-while** serão executadas **pelo menos uma vez**.

Neste caso, devemos ter as mesmas precauções quanto while, no que diz respeito a looping infinito. Mas não é necessário inicializar a variável antes do bloco de código como acontece com while, pois a comparação só será feita após todo o código ter sido executado.

A declaração **do-while** possui a seguinte forma:

### Declaração do-while

```

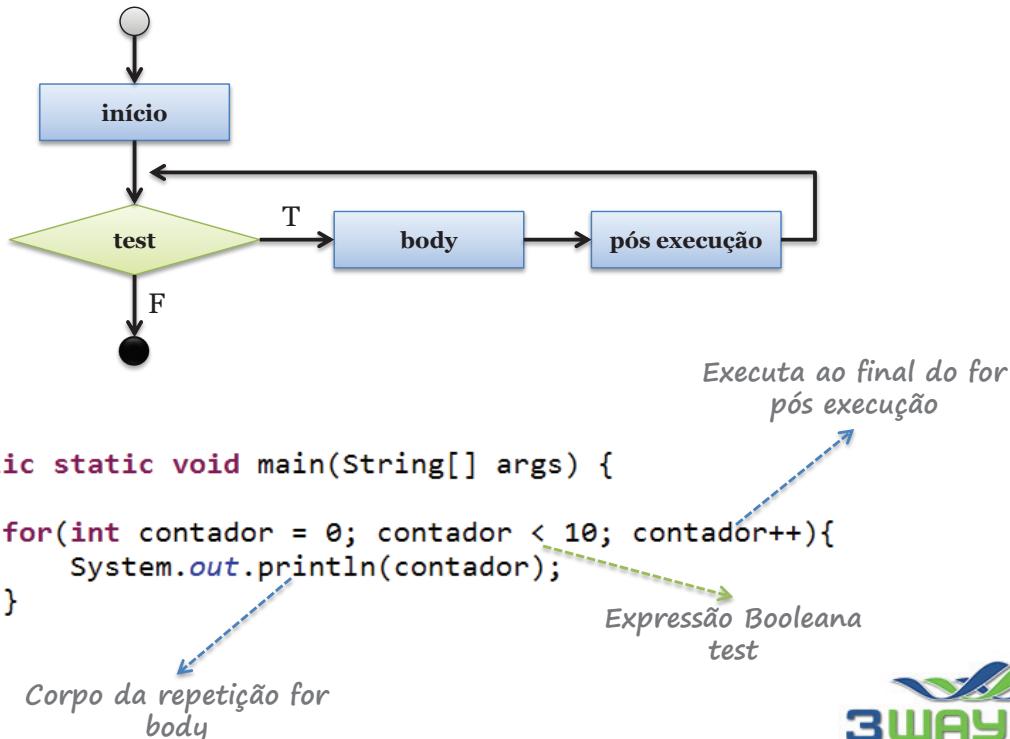
do {
    instrução1;
    instrução2; [...]
} while (expressão_lógica);

```

Inicialmente, as instruções dentro do laço **do-while** são executadas. Então, a condição na expressão\_lógica é avaliada. Se for verdadeira, as instruções dentro do laço **do-while** serão executadas novamente.

A diferença entre uma declaração **while** e **do-while** é que, no laço **while**, a avaliação da expressão lógica é feita antes de se executarem as instruções nele contidas enquanto que, no laço **do-while**, primeiro se executam as instruções e depois se realiza a avaliação da expressão lógica, ou seja, as instruções dentro em um laço **do-while** são executadas pelo menos uma vez.

# Estrutura de repetição (for)



## Java e Orientação a Objetos

A declaração **for**, como nas declarações anteriores, permite a execução do mesmo código uma quantidade determinada de vezes.

A declaração for possui a seguinte forma:

### Declaração for

```
for ([declaração]; [expressão]; [incremento]) {
    instrução1;
    instrução2;
    [...]
}
```

Onde:

1. **[declaração]** – seção de declaração e inicialização das **variáveis locais** para o laço, executada uma única vez na primeira iteração.
2. **[expressão]** – executada a cada passo do laço, é utilizada como condição de parada quando a expressão resultar em **false** as iterações terminam.
3. **[incremento]** – normalmente usada para incrementar um contador para o laço, mas qualquer declaração válida em Java poderá ser executada. É executado após a execução do bloco interno ao laço.

# Declaração break

Interrompe a repetição infinita



```
public static void main(String[] args) {
    int contador = 0;
    while(true){ //laço infinito
        if(contador == 10){
            System.out.println("break - (while-true)");
            break;
        }
        System.out.println(contador);
        contador++;
    }
}
```



## Java e Orientação a Objetos

É a declaração de desvio usada para sair de um laço antes do normal. O tipo determina para onde é transferido o controle. O break transfere o controle para o final de uma construção de laço (for, do, while ou switch). O laço vai encerrar independentemente de seu valor de comparação e a declaração após o laço será executada.

A declaração **break** possui duas formas: **unlabeled** (não identificada) e **labeled** (identificada).

### Declaração unlabeled break

A forma **unlabeled** de uma declaração **break** encerra a execução de um **switch** e o fluxo de controle é transferido imediatamente para o final deste. Podemos também utilizar a forma para terminar declarações **for**, **while** ou **do-while**.

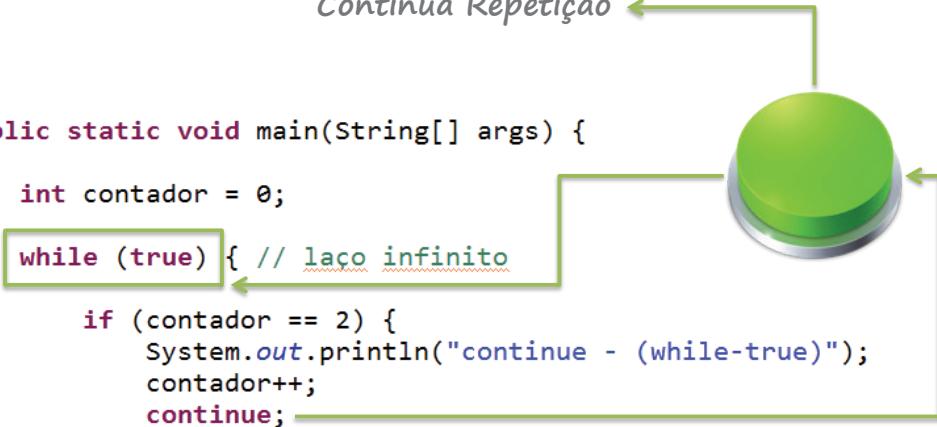
### Declaração labeled break

A forma **labeled** de uma declaração **break** encerra o processamento de um laço que é identificado por um **label** especificado na declaração **break**.

Um **label**, em linguagem Java, é definido colocando-se um nome seguido de dois-pontos.

# Declaração continue

*Continua Repetição*



```

public static void main(String[] args) {
    int contador = 0;
    while (true) { // laço infinito
        if (contador == 2) {
            System.out.println("continue - (while-true)");
            contador++;
            continue;
        }
        if (contador == 10) {
            System.out.println("break - (while-true)");
            break;
        }
        System.out.println(contador);
        contador++;
    }
}
  
```



## Java e Orientação a Objetos

A declaração continue faz com que a execução do programa volte imediatamente para o início do laço, porém para a próxima interação. O continue faz o interpretador pular para a próxima iteração e obriga-o a testar a condição.  
 A declaração continue tem duas formas: **unlabeled** e **labeled**. Utilizamos uma declaração continue para saltar a repetição atual de declarações **for**, **while** ou **do-while**.

### Declaração unlabeled continue

A forma **unlabeled** salta as instruções restantes de um laço e avalia novamente a expressão lógica que o controla.

### Declaração labeled continue

A forma **labeled** da declaração continue interrompe a repetição atual de um laço e salta para a repetição exterior marcada com o **label** indicado.

# Java e Orientação a Objetos

## Array

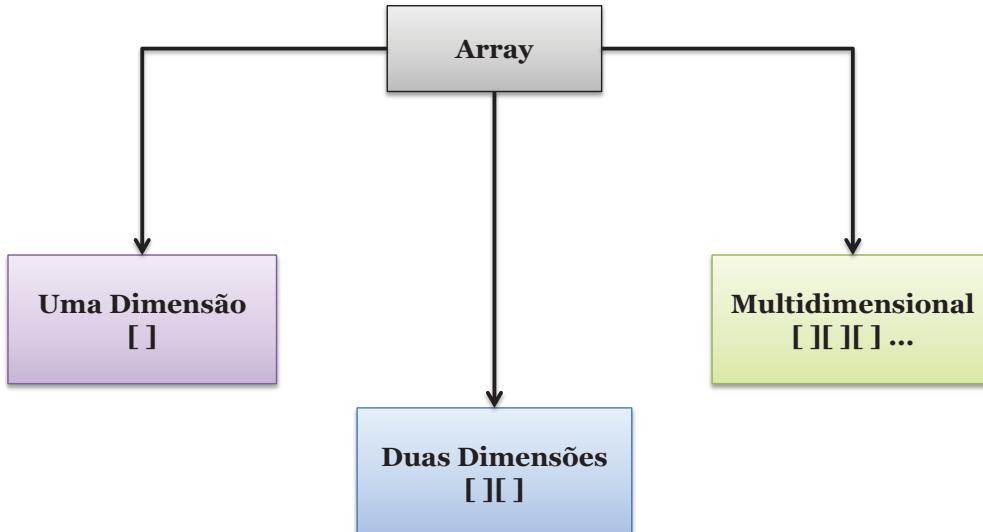


## Java e Orientação a Objetos

Arrays são estruturas muito utilizadas na programação por isso, é fundamental entender seu funcionamento e saber como utilizá-las. Os arrays ou matrizes, como são conhecidos em Java, fazem parte da API do Java. São objetos de recipientes que contém um número fixo de valores de um único tipo. O comprimento de um array é estabelecido quando criado, sendo que após a criação o seu comprimento fica fixo.

Cada item em um array é chamado de elemento, e cada elemento é acessado pelo número, o índice. Abaixo é mostrado se dá esse acesso aos seus elementos, lembrando que sempre sua numeração começa em 0.

# Array ?



## Java e Orientação a Objetos

Um **array** é uma estrutura de dados que define uma coleção ordenada de elementos homogêneos e de tamanho fixo. Um **array** armazena múltiplos itens de um mesmo tipo de dado em um bloco contínuo de memória. O tamanho de um **array** é fixo e **não pode ser alterado** depois de sua criação.

**Array, variável indexada** é também conhecido como vetor (para arrays unidimensionais) ou matriz (para arrays bidimensionais), é uma das mais simples estruturas de dados. Os arrays mantêm uma série de elementos de dados, geralmente do mesmo tamanho e tipo de dados. Elementos individuais são acessados por sua posição no array. A posição é dada por um índice, também chamado de subscrição. O índice geralmente utiliza uma sequência de números inteiros, mas o índice pode ter qualquer valor ordinal. Alguns arrays são multidimensionais, significando que eles são indexados por um número fixo de números inteiros, por exemplo, por uma sequência (ou sucessão) finita de quatro números inteiros.

Os arrays podem ser considerados como as estruturas de dados mais simples. Têm a vantagem de que os seus elementos são acessíveis de forma rápida, mas têm uma notável limitação: são de tamanho fixo.

Se imaginarmos que uma variável em Java é como uma xícara de café (onde só se toma café) um **array** seria uma bandeja contendo uma quantidade definida de xícaras.

# Declarando Array

**Inicialização**

```
int a[] = new int[12];
```

```
|| int []a = {1,2,3,4,5,6,7,8,9,10,11,12};
```

Tamanho (length) = 12

**Valores**

1	2	3	4	5	6	7	8	9	10	11	12
---	---	---	---	---	---	---	---	---	----	----	----

**Índices**

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	a[10]	a[11]
------	------	------	------	------	------	------	------	------	------	-------	-------



“Pode guardar somente papel”  
(Um único tipo de dado, anteriormente definido)



## Java e Orientação a Objetos

Independentemente do tipo de **array** com o qual você esteja trabalhando, o identificador do **array** é, na verdade, uma referência que foi criada na pilha (memória). Este é o objeto que guarda as referências para os outros objetos ou valores de tipos primitivos, pode ser criado tanto implicitamente como parte da sintaxe de inicialização do **array**, como explicitamente, com o operador **new**.

O atributo **length** (só está disponível para leitura), que informa a você quantos elementos tem armazenados naquele objeto **array**.

**Array** precisa ser declarado como qualquer variável. Ao declarar um **array**, defina o tipo de dados deste seguido por colchetes **[]** e pelo nome que o identifica.

Depois da declaração, precisamos inicializar o array e especificar seu tamanho. Uma vez que tenha sido inicializado, o tamanho de um array não pode ser modificado, pois é armazenado em um bloco contínuo de memória.

## Declaração e inicialização array

```
// declarar e inicializar
int idades[] = new int[100];
```

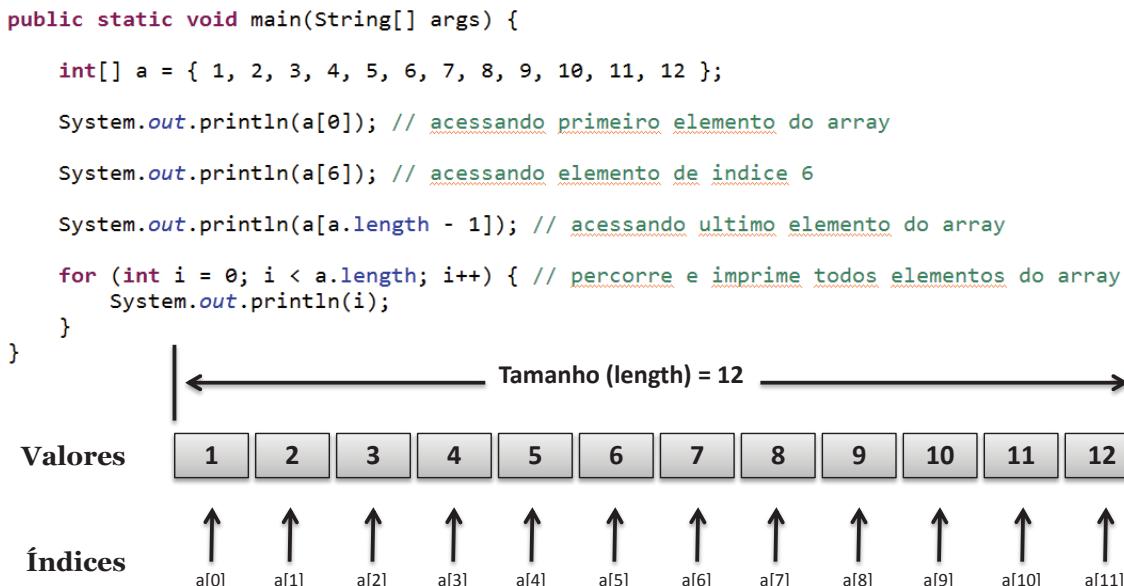
No exemplo, a declaração diz ao compilador Java que o identificador **idades** será usado como um nome de um **array** contendo valores do tipo **int**, e que o novo **array** contém 100 elementos.

## Exemplo

```
// criando um array de 4 variáveis double inicializados  
// com os valores {100, 90, 80, 75};  
double [] notas = {100, 90, 80, 75};
```

Ou invés de utilizar uma nova linha de instrução para construir um array, também é possível automaticamente declarar, construir e adicionar um valor uma única vez.

# Acessando um elemento do Array



## Java e Orientação a Objetos

Para acessar um elemento do **array**, ou parte de um **array**, utiliza-se um número inteiro chamado de **índice**. Os números dos índices são sempre inteiros. Eles começam com **zero** e **progredem sequencialmente** por todas as posições até o fim do **array**. Lembre-se que os elementos dentro do array possuem índice de 0 a **length - 1**.

Lembre-se que o **array**, uma vez declarado e construído, terá o valor de cada membro inicializado automaticamente, usando o valor **default** para cada tipo de dado primitivo.

Entretanto, tipos de dados por referência, como as variáveis do tipo **String**, não serão inicializados com caracteres em branco ou com uma **String** vazia “”, serão inicializados com o valor **null**. Deste modo, o ideal é preencher os elementos do **arrays** de forma explícita antes de utilizá-los. A manipulação de objetos com referência **null** causará a desagradável surpresa de uma exceção do tipo **NullPointerException**, por exemplo, ao tentar executar algum método da classe String

### Tamanho de Array

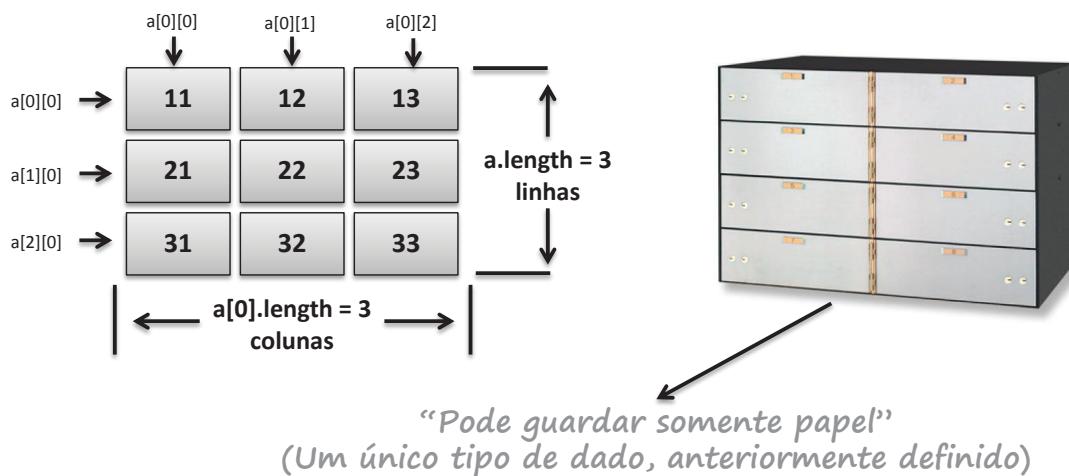
O atributo **length** de um **array** retorna seu tamanho, ou seja, a quantidade de elementos, ou seja, ele é uma constante já que o **array** depois de criado não sofrerá alteração de tamanho.

# Arrays Multidimensionais

Inicialização

```
int a[][] = new int[3][3];
```

```
int [][]a = {{11,12,13},{21,22,23},{31,32,33}};
```



## Java e Orientação a Objetos

**Arrays** multidimensionais são implementados como **arrays dentro de arrays**. São declarados ao atribuir um novo conjunto de colchetes depois do nome do array. Primeiramente, vamos definir o que é uma dimensão de um array. **A dimensão, ou quantidade de dimensões, é o conjunto de valores que precisamos definir para localizar uma informação.** Por exemplo, uma lista de alunos de 0 a 100 pode ser organizada em um array de uma dimensão, pois para localizar um aluno nessa lista basta indicar um valor da sequência.

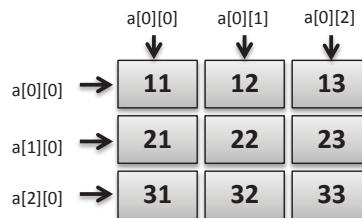
É possível definir n dimensões, porém, na prática, não é comum nem recomendável trabalhar com tantas dimensões. Na prática, é extremamente comum trabalharmos com apenas 1 (uma), e algumas vezes com 2 (duas), e, quase nunca com 3 (três) dimensões, pois há outros recursos na programação orientada a objetos que desencorajam e oferecem alternativas melhores a essa prática.

# Acessando um elemento de um Array Multidimensional

```

public static void main(String[] args) {
    int[][] matriz = { { 11, 12, 13 }, { 21, 22, 23 }, { 31, 32, 33 } }; // constroi matriz
    System.out.println(matriz.length); // imprime numero de linhas
    System.out.println(matriz[0].length); // imprime numero de colunas
    System.out.println(matriz[0][0]); // acessa elemento na linha [0] e coluna [0]
    System.out.println(matriz[2][2]); // acessa elemento na linha [2] e coluna [2]
    for (int linha = 0; linha < matriz.length; linha++) { // percorre todas linhas
        for (int coluna = 0; coluna < matriz[linha].length; coluna++) { // percorre todas colunas
            System.out.print(matriz[linha][coluna] + " ");
        }
        System.out.println("\n");
    }
}

```



## Java e Orientação a Objetos

Já para localizar um elemento em um array bidimensional precisamos de duas coordenadas, linha e coluna. Assim, para representarmos um tabuleiro de xadrez com arrays, são necessárias duas dimensões.

Acessar um elemento em um **array** multidimensional é semelhante a acessar elementos em um **array** de uma dimensão.

# Percorrendo Arrays com Enhanced-for

```

public static void main(String[] args) {

    int[] a = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 };

    for (int i = 0; i < a.length; i++) { // percorre array
        System.out.println(i); // imprime todos elementos do array
    }
}

public static void main(String[] args) {

    int[] a = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 };

    for (int i : a) { // utilizando Enhanced-for
        System.out.println(i); // imprime todos elementos do array
    }
}
  
```



## Java e Orientação a Objetos

Este laço é similar aos laços **for-each** de outras linguagens de programação (Perl, PHP, Python, etc.). O for aprimorado tem como objetivo de facilitar o loop em Array ou em conjuntos.

### Sintaxe do Enhanced-for

```

for (<declaração> : <Expressão>){
    <sentença>
}
  
```

**Declaração:** aqui declaramos o tipo da variável que vai receber os elementos de um Array ou Conjunto.

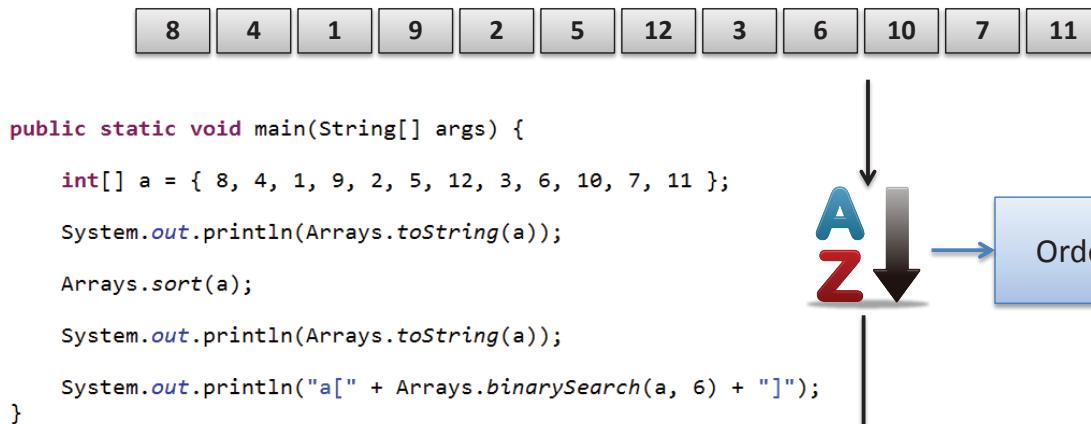
**Expressão:** aqui será o array ou o conjunto que deseja percorrer.

#### Ponto importante:

Deve ser usado Array e conjuntos do mesmo **tipo da declaração (int,double, Object)**, caso contrário, não compila.

O for aprimorado não tem como objetivo de substituir o for básico. Há situações que o for aprimorado não é mais adequado. ex.: *quando é necessário determinar que uma posição em um conjunto ou array.*

# Manipulando Arrays com java.util.Arrays



## Java e Orientação a Objetos

Dentro do pacote `java.util` encontramos uma classe chamada `Arrays`. Esta classe possui uma série de métodos estáticos que nos ajudam a trabalhar mais facilmente com vetores. Dentre seus principais métodos podemos evidenciar os seguintes:

### > `binarySearch`

Este método recebe sempre 2 parâmetros sendo um deles o vetor e outro o elemento que se deseja buscar dentro dele e utiliza o algoritmo da busca binária para localizar o elemento dentro do vetor.

### > `sort`

Realizar a ordenação de um vetor utilizando um algoritmo do tipo **Quick Sort**. Este tipo de algoritmo também será discutido mais a diante. Por este método receber o vetor por parâmetro (que lembrando, vetores são objetos) ele o ordena e não retorna valor algum.

### > `asList`

Converte o vetor em uma coleção do tipo lista.

### > `copyOf`

Cria uma cópia de um vetor. Pode-se copiar o vetor completamente ou apenas parte dele.

# Java e Orientação a Objetos

## Bases da programação Java OO



## Java e Orientação a Objetos

Durante anos, os programadores se dedicaram a construir aplicações muito parecidas que resolviam uma vez ou outra, os mesmos problemas. Para conseguir que os esforços dos programadores possam ser utilizados por outras pessoas foi criado a POO (Programação Orientada a Objetos). Esta é uma série de normas de realizar as coisas de maneira que você possa utilizá-las e adiantar seu trabalho, de maneira que consigamos que o código possa se reutilizar.

O termo Programação Orientada a Objetos foi criado por Alan Kay, autor da linguagem de programação Smalltalk. Mas mesmo antes da criação do Smalltalk, algumas das ideias da POO já eram aplicadas, sendo que a primeira linguagem a realmente utilizar estas ideias foi à linguagem Simula 67, criada por Ole Johan Dahl e Kristen Nygaard em 1967. Note que este paradigma de programação já é bastante antigo, mas só agora vem sendo aceito realmente nas grandes empresas de desenvolvimento de Software. Alguns exemplos de linguagens modernas utilizadas por grandes empresas em todo o mundo que adotaram essas ideias: Java, C#, C++, Object Pascal (Delphi), Ruby, Python, Lisp, entre outras.

A POO não é difícil, mas é uma forma especial de pensar, às vezes subjetiva de quem a programa, de forma que a maneira de fazer as coisas possa ser diferente segundo o programador. Embora possamos fazer os programas de formas distintas, nem todas elas são corretas, o difícil não é programar orientado a objetos e sim, programar bem. Programar bem é importante porque assim podemos aproveitar todas as vantagens da POO.

## Ideias básicas da POO

A POO foi criada para tentar aproximar o mundo real do mundo virtual: a ideia fundamental é tentar simular o mundo real dentro do computador. Para isso, nada mais natural do que utilizar Objetos, afinal, nosso mundo é composto de objetos, certo?

O mundo físico é constituído por objetos tais como carro, leão, pessoa, dentre outros. Estes objetos são caracterizados pelas suas propriedades (ou atributos) e seus comportamentos.

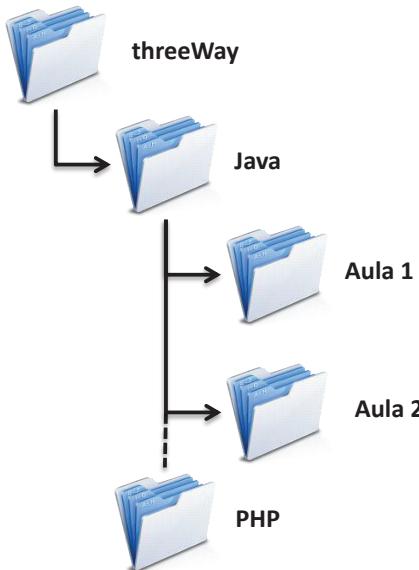
Na POO o programador é responsável por moldar o mundo dos objetos, e explicar para estes objetos como eles devem interagir entre si. Os objetos “conversam” uns com os outros através do envio de mensagens, e o papel principal do programador é especificar quais serão as mensagens que cada objeto pode receber, e também qual a ação que aquele objeto deve realizar ao receber aquela mensagem em específico.

## Como se pensa em objetos

Pensar em termos de objetos é muito parecido a como faríamos na vida real. Por exemplo, vamos pensar em um carro para dar um modelo em um esquema de POO. Diríamos que o carro é o elemento principal que tem uma série de características, como poderiam ser a cor, o modelo ou a marca. Ademais tem uma série de funcionalidades associadas, como podem ser andar, parar ou estacionar.

Então em um esquema POO o carro seria o objeto, as propriedades seriam as características como a cor ou o modelo e os métodos seriam as funcionalidades associadas como andar ou parar.

# Pacotes



```

package <nomePacote>;
import <nomePacote.elementoAcessado>;
<declaraçãoClasse>
  
```

```

package threeWay.Java.Aula;
import java.util.Arrays;
public class Teste { }
  
```



## Java e Orientação a Objetos

Quando um programador utiliza as classes feitas por outro, surge um problema clássico: como escrever duas classes com o mesmo nome?

Por exemplo: pode ser que a minha classe de Data funcione de certo jeito, e a classe Data de um colega, de outro jeito. Pode ser que a classe de Data de uma biblioteca funcione ainda de uma terceira maneira diferente.

Pensando um pouco mais, notamos a existência de outro problema e da própria solução: o sistema operacional não permite a existência de dois arquivos com o mesmo nome sob o mesmo diretório, portanto precisamos organizar nossas classes em diretórios diferentes.

Os diretórios estão diretamente relacionados aos chamados pacotes e costumam agrupar classes de funcionalidades similares ou relacionadas.

Por exemplo, no pacote `java.util` temos as classes `Date`, `SimpleDateFormat` e `GregorianCalendar`; todas elas trabalham com datas de formas diferentes.

Usamos pacotes para organizar as classes semelhantes. Pacotes, de um grosso modo, são apenas pastas ou diretórios do sistema operacional onde ficam armazenados os arquivos fonte de Java e são essenciais para o conceito de encapsulamento, no qual são dados níveis de acesso às classes.

## Criar Pacotes

Muitos compiladores criam automaticamente os pacotes como uma forma eficaz de organização, mas a criação de pacote pode ser feita diretamente no sistema operacional. Basta que criemos uma pasta e lhe demos um nome. Após isso, devemos gravar todos os arquivos fonte de Java dentro desta pasta.

## Definir Pacotes

Agora que já possuímos a pasta que será nosso pacote, devemos definir em nossa classe a qual pacote ela pertence. Isso é feito pela palavra reservada package.

Package deve ser a primeira linha de comando a ser compilada de nossa classe.

Para definir pacotes, declaramos como segue:

```
package <nomeDoPacote>;
```

Portanto, se tivéssemos criado uma pasta chamada meuPacote e fossemos criar uma classe nesta pasta (pacote), o começo de nosso código seria: package meuPacote;;

## Importar Pacotes

Para usar uma classe do mesmo pacote, basta fazer referência a ela como foi feito até agora simplesmente escrevendo o próprio nome da classe.

Quando precisamos fazer referência a uma classe que pertence a outro pacote, devemos usar seu nome completo ou podemos importar os pacotes dessas classes.

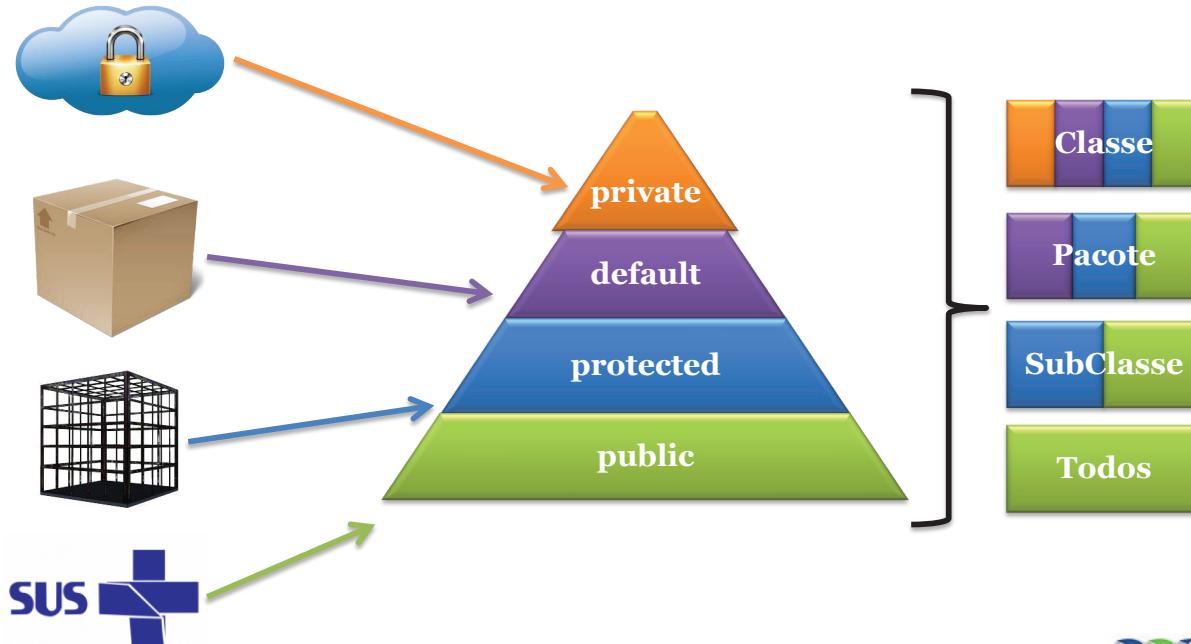
Ter que usar o nome completamente qualificado de uma classe não é muito agradável, além de tornar as linhas de nosso código-fonte bem grandes, tornando sua leitura mais difícil, o que não é nada bom para manutenção de um programa.

Para utilizar classes externas ao pacote, de uma forma mais simples, podemos importar os pacotes dessas classes. Por padrão, todos as suas classes Java importam o pacote java.lang implicitamente. É por isso que é possível utilizar classes como String e Integer dentro da sua classe, sem usar seu nome completo: java.lang.String, java.lang.Integer.

Para importar pacotes você usa a palavra-chave import seguido do nome completamente qualificado da classe.

Lembre-se, import deve ser colocado após a cláusula package e antes das definições de classe ou interface no arquivo de código fonte.

# Modificadores de acesso



## Java e Orientação a Objetos

Os modificadores de acesso são padrões de visibilidade de acessos às classes, atributos e métodos. Esses modificadores são palavras-chaves reservadas pelo Java, ou seja, palavras reservadas não podem ser usadas como nome de métodos, classes ou atributos.

### **public**

Uma declaração com o modificador **public** pode ser acessada de qualquer lugar e por qualquer entidade que possa visualizar a classe a que ela pertence.

### **private**

Os membros da classe definidos como **private** não podem ser acessados ou usados por nenhuma outra classe. Esse modificador não se aplica às classes, somente para seus métodos e atributos. Esses atributos e métodos também não podem ser visualizados pelas classes herdadas.

### **protected**

O modificador **protected** torna o membro acessível às classes do mesmo pacote ou através de herança, seus membros herdados não são acessíveis a outras classes fora do pacote em que foram declarados.

## default (padrão)

A classe e/ou seus membros são acessíveis somente por classes do mesmo pacote, na sua declaração não é definido nenhum tipo de modificador, sendo este identificado pelo compilador.

### Modificadores de acesso

	private	defalt	protect	public
mesma classe	sim	sim	sim	sim
mesmo pacote	não	sim	sim	sim
pacotes diferentes (subclasses)	não	não	sim	sim
pacotes diferentes (sem subclasses)	não	não	não	sim

## Outros modificadores de acesso

### final

Quando é aplicado na classe, não permite estendê-la, nos métodos impede que o mesmo seja sobreescrito (overriding) na subclasse, e nos valores de variáveis não pode ser alterado depois que já tenha sido atribuído um valor.

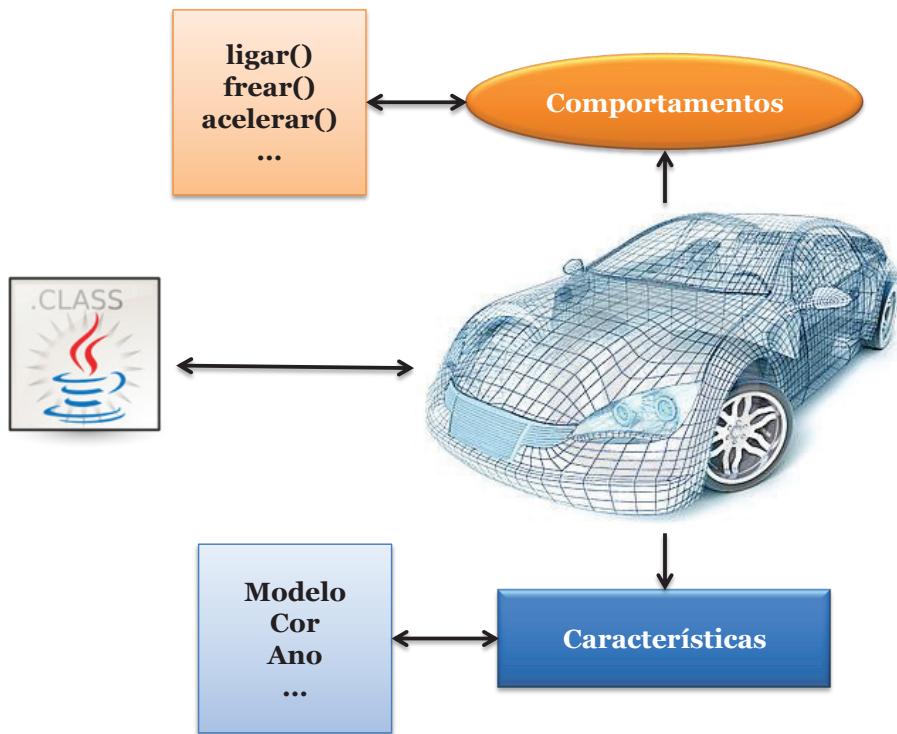
### abstract

Esse modificador não é aplicado nas variáveis, apenas nas classes. Uma classe abstrata não pode ser instanciada, ou seja, não pode ser chamada pelos seus construtores. Se houver alguma declaração de um método como abstract (abstrato), a classe também deve ser marcada como abstract.

### static

É usado para a criação de uma variável que poderá ser acessada por todas as instâncias de objetos desta classe como uma variável comum, ou seja, a variável criada será a mesma em todas as instâncias e quando seu conteúdo é modificado numa das instâncias, a modificação ocorre em todas as demais. E nas declarações de métodos ajudam no acesso direto à classe, portanto não é necessário instanciar um objeto para acessar o método.

# Classes



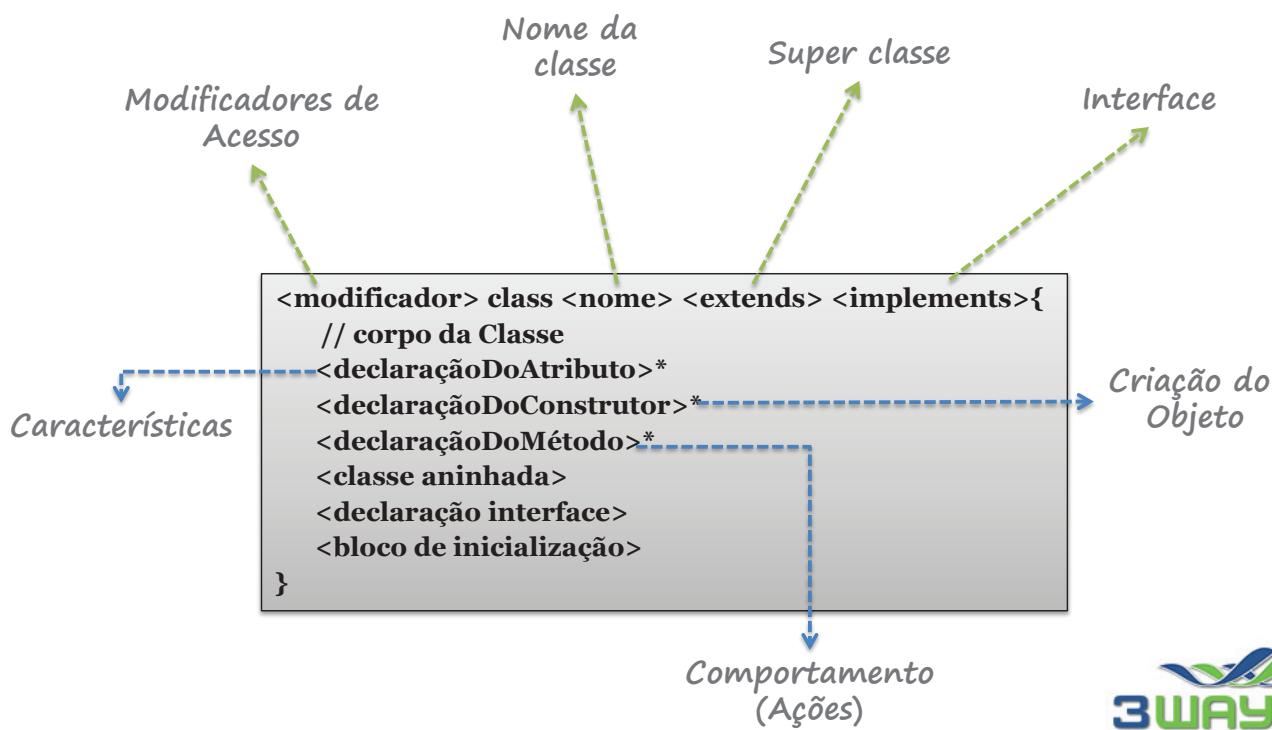
## Java e Orientação a Objetos

Abstrações é o fundamento utilizado por nós, seres humanos, a fim de lidar com a complexidade. Uma abstração demonstra o comportamento e propriedades essenciais de um objeto, que o diferencia de outro objeto qualquer. Por exemplo, ao utilizar um carro, você não pensa nele como um conjunto de dezenas de milhares de peças individuais. Pensamos em um carro como um único objeto sobre qual podemos realizar um determinado conjunto de operações. Essa abstração permite que as pessoas utilizem um carro sem nenhum conhecimento sobre mecânica de automóveis. Podemos ignorar os detalhes do motor como: sistema de injeção de combustíveis, transmissão, sistema de freios funcionam individualmente ou em conjunto. Ao invés disso, utilizamos o objeto automóvel como um elemento único.

A essência da Programação Orientada a Objetos (POO) consiste modelar as abstrações usando Classes e Objetos. A parte difícil deste trabalho é encontrar a abstração mais correta. Uma Classe modela uma abstração pela definição das propriedades e comportamentos dos objetos representados pela abstração. Uma Classe, então, especifica a categoria dos Objetos e funciona como uma “planta baixa” na criação desses objetos.

Propriedades de um Objeto da Classe são chamadas de atributos e são definidos usando variáveis em Java, também denominado Campos da Classe. Variáveis e métodos utilizados na definição da classe são, juntos, denominados de Membros da Classe.

# Definindo Classes



## Java e Orientação a Objetos

Em Java, classes são definidas através do uso da palavra-chave `class`.

A primeira linha é um comando que inicia a declaração da classe. Após a palavra-chave `class`, segue-se o nome da classe, que deve ser um identificador válido para a linguagem. O modificador é opcional; se presente, pode ser uma combinação de `public` e `abstract` ou `final`.

A definição da classe propriamente dita está entre as chaves `{` e `}`, que delimitam blocos na linguagem Java. Este corpo da classe usualmente obedece à seguinte sequência de definição:

As variáveis de classe ou de instância (Atributos).

Os construtores de objetos dessa classe.

Os métodos da classe, geralmente agrupados por funcionalidade.

Toda classe pode também ter um método `main` associado, que será utilizado pelo interpretador Java para dar início à execução de uma aplicação.

Java também oferece outra estrutura, denominada interface, com sintaxe similar à de classes, mas contendo apenas a especificação da funcionalidade que uma classe deve conter, sem determinar como essa funcionalidade deve ser implementada.

# Métodos

*Decomposição e a Solução  
para problemas*

*Separa o problema  
em partes menores e  
reaproveitáveis*

*Métodos resolve  
uma parte específica  
desses problemas*



*Na Orientação a Objetos  
os métodos referenciam comportamentos das classes.*



## Java e Orientação a Objetos

Métodos nada mais são que um bloco de códigos que podem ser acessados ou não (Depende do modificador de acesso) a qualquer momento e em qualquer lugar de nossos programas.

### As utilidades dos métodos Organização

Tudo que é possível fazer com os métodos, é possível fazer sem. Porém, os programas em Java ficariam enormes, bagunçados e pior: teríamos que repetir um mesmo trecho de código inúmeras vezes.

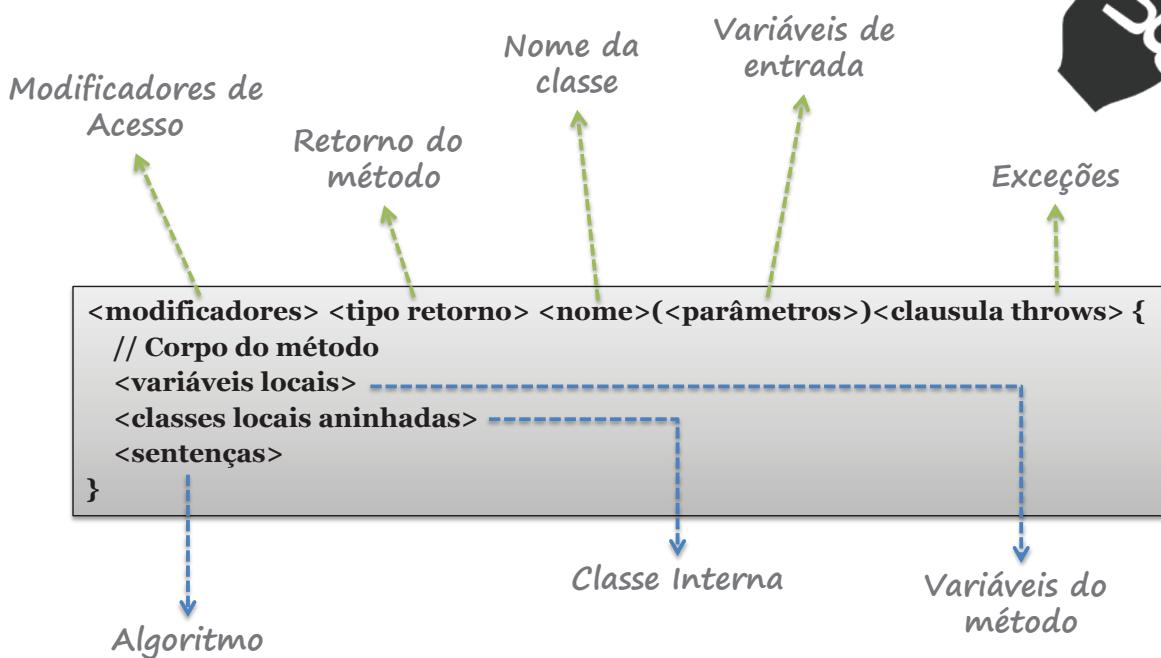
Uma das grandes vantagens, se não a maior, é utilizar e acessar várias vezes os métodos em Java que escrevemos.

### Reutilização

Quando você for iniciar um grande projeto, dificilmente você terá que fazer tudo do zero. Se fizer, muito provavelmente é porque você se organizou muito mal ou não fez o nosso Curso Java.

Crie métodos que façam coisas específicas e bem definidas, com nomes fáceis de identificar sua ação. Isso será útil para reutilização. Em vez de criar um método que constrói uma casa, crie um método que cria um quarto, outro método que cria a sala, outro que cria o banheiro, um método pra organizar a casa etc. Assim, quando tiver que criar um banheiro, já terá o método específico para aquilo. Veja os métodos como peças de um quebra-cabeça. Porém, são peças ‘coringa’, que se encaixam com muitas outras peças.

# Definindo Métodos



## Java e Orientação a Objetos

Qualificadores de método:

> **Visibilidade:**

- public - método acessível onde quer que a classe esteja.
- private - método acessível apenas na classe.
- protected - método acessível na classe, subclasses e classes no mesmo pacote.

> **abstract:** método sem corpo.

> **static:** método de classe.

> **final:** método que não pode ser redefinido nas subclasses

No caso de omissão de um qualificador de visibilidade, o método é acessível na classe e classes no mesmo pacote.

Com exceção dos qualificadores de visibilidade, um método pode ter mais do que um qualificador.

Contudo, um método não pode ser ao mesmo tempo abstract e final.

O tipo de retorno de um método é obrigatório.

Valor retornado pela instrução return.

Um método pode ter zero, um, ou mais parâmetros.

# Objetos



## Java e Orientação a Objetos

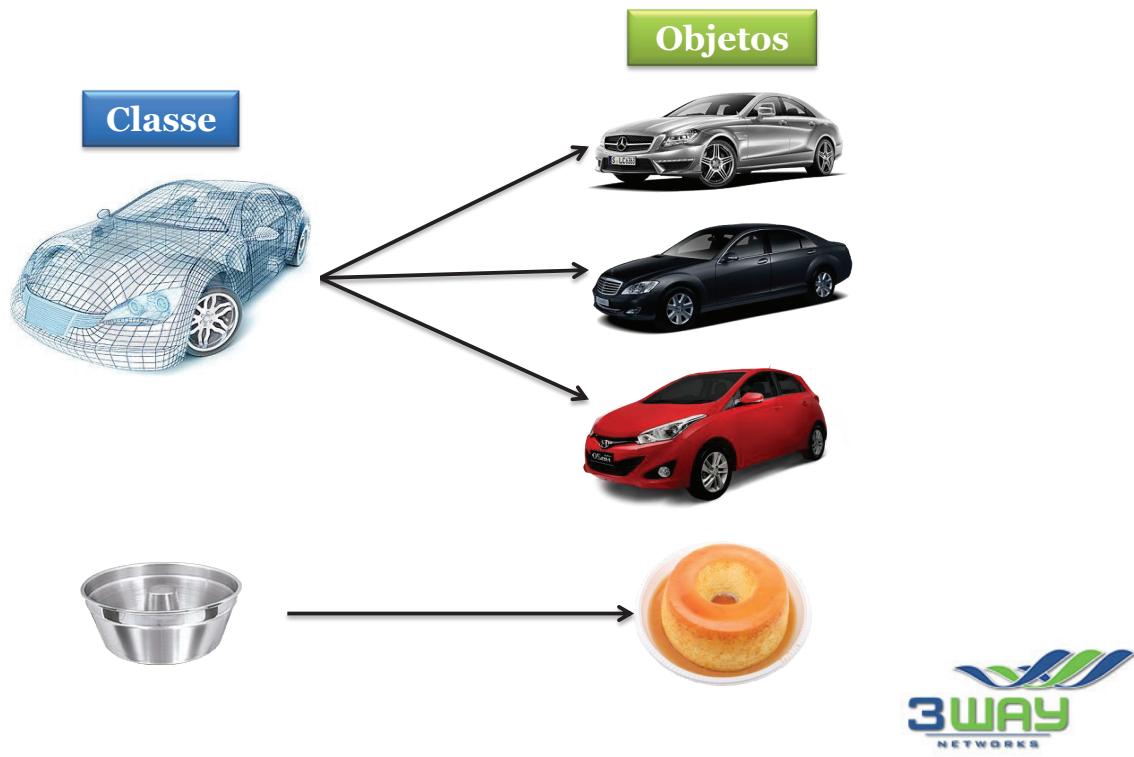
Um Objeto é qualquer coisa criada a partir dessa Classe, algo contendo as propriedades e comportamentos definidos na Classe. Voltando ao exemplo do carro, um grupo de engenheiros mecânicos cria o projeto de um carro (a Classe) e a partir deste projeto são reproduzidos milhares de automóveis (os Objetos), praticamente idênticos entre si.

O comportamento de um Objeto, também conhecido por operações, é definido usando-se métodos em Java.

O objeto é uma instância de uma classe. Ele é construído usando-se a classe, da mesma maneira que usamos uma “planta baixa” para construir uma casa, é uma entidade real da abstração que a classe representa; usando o exemplo do carro, uma instância representa um veículo criado a partir do projeto dos engenheiros mecânicos. Um objeto deve ser explicitamente criado a partir de uma classe antes que possa ser utilizado em um programa.

É através de objetos que (praticamente) todo o processamento ocorre em aplicações desenvolvidas com linguagens de programação orientadas a objetos. O uso racional de objetos, obedecendo aos bons princípios associados à sua definição conforme estabelecido no paradigma de desenvolvimento orientado a objetos, é chave para o desenvolvimento de bons sistemas de software.

# Classe x Objeto



## Java e Orientação a Objetos

Para diferenciar entre classes e objetos, vamos examinar um exemplo. O que temos aqui é uma classe Carro que pode ser usada pra definir diversos objetos do tipo carro. Na tabela mostrada abaixo, Carro A e Carro B são objetos da classe Carro. A classe tem os campos número da placa, cor, fabricante e velocidade que são preenchidos com os valores correspondentes do carro A e B. O carro também tem alguns métodos: acelerar, virar e frear.

### Exemplo

	Classe Carro	Objeto Carro A	Objeto Carro A
Atributos do Objeto	Número da Placa	ABC 1111	XYZ 9999
	Cor	Prata	Vermelho
	Modelo	Mercedes	Hyundai
	Ano	2013	2012
Métodos do Objeto	Método Frear		
	Método Ligar		
	Método Acelerar		

Quando construídos, cada objeto adquire um conjunto novo de estado. Entretanto, as implementações dos métodos são compartilhadas entre todos os objetos da mesma classe. As classes fornecem o benefício da Reutilização de Classes (ou seja, utilizar a mesma classe em vários projetos). Os programadores de software podem reutilizar as classes várias vezes para criar os objetos.

# Notação UML

Diagrama de Objetos

Diagrama de Objetos

Diagrama de Caso de Uso

Diagrama de Classe



Diagrama de Atividades

Diagrama de Componentes

Diagrama de Intereração

Diagrama de Implantação

Diagrama de Pacotes

Diagrama de Estrutura



## Java e Orientação a Objetos

A Unified Modeling Language (UML) é uma linguagem visual para especificação (modelagem) de sistemas orientados a objeto. A UML não é uma metodologia de desenvolvimento, o que significa que ela não diz para você o que fazer primeiro e em seguida ou como projetar seu sistema, mas ela lhe auxilia a visualizar seu desenho e a comunicação entre objetos.

A UML privilegia a descrição de um sistema seguindo três perspectivas:

- > Os diagramas de classes - (Dados estruturais);
- > Os diagramas de casos de uso (Operações funcionais);
- > Os diagramas de sequência, atividades e transição de Estados (Eventos temporais).

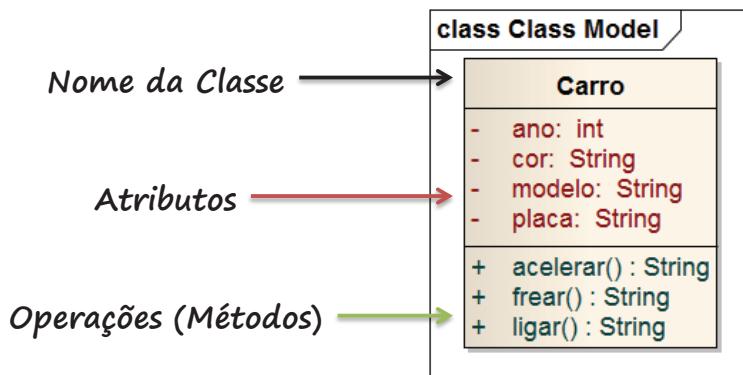
Basicamente, a UML permite que desenvolvedores visualizem os produtos de seus trabalhos em diagramas padronizados. Junto com uma notação gráfica, a UML também especifica significados, isto é, semântica. É uma notação independente de processos.

É importante distinguir entre um modelo UML e um diagrama(ou conjunto de diagramas) de UML. O último é uma representação gráfica da informação do primeiro, mas o primeiro pode existir independentemente. O XMI (XML Metadata Interchange) na sua versão corrente disponibiliza troca de modelos, mas não de diagramas.

Os objetivos da UML são: especificação, documentação, estruturação para abstração e maior visualização lógica do desenvolvimento completo de um sistema de informação.

# Notação UML

## Diagrama de Classe



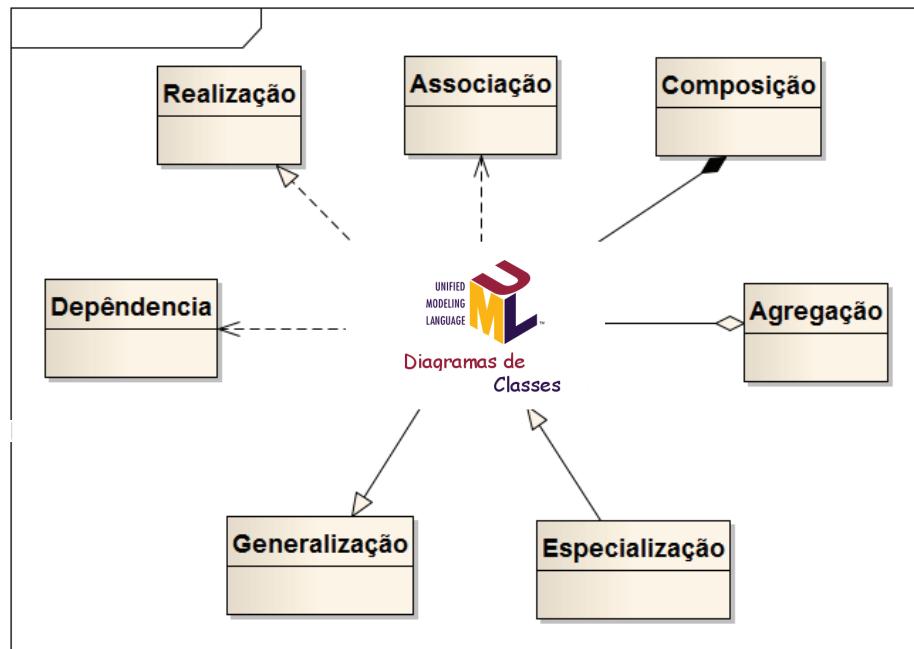
## Java e Orientação a Objetos

O diagrama de classes demonstra a estrutura estática das classes de um sistema onde estas representam as “coisas” que são gerenciadas pela aplicação modelada. Classes podem se relacionar com outras através de diversas maneiras: associação, dependência, especialização, ou em pacotes. Todos estes relacionamentos são mostrados no diagrama de classes juntamente com as suas estruturas internas, que são os atributos e operações. O diagrama de classes é considerado estático já que a estrutura descrita é sempre válida em qualquer ponto do ciclo de vida do sistema.

No modelo de classe trabalhamos com um único elemento um retângulo dividido em três partes, a primeira divisão é utilizada para o nome da classe, na segunda divisão colocamos as informações de atributos e a última divisão é utilizada para identificar os métodos.

É importante apresentar o conceito de instância, isto é, cada objeto é uma instância de sua classe. Cada instância tem seus próprios valores de atributos, compartilha os nomes dos atributos e os métodos com os outros objetos da mesma classe.

# Notação UML Relacionamentos



## Java e Orientação a Objetos

### Associação

São relacionamentos estruturais entre instâncias e especificam que objetos de uma classe estão ligados a objetos de outras classes. A associação pode existir entre classes ou entre objetos. Uma associação entre a classe Professor e a classe disciplina (um professor ministra uma disciplina) significa que uma instância de Professor (um professor específico) vai ter uma associação com uma instância de Disciplina. Esta relação significa que as instâncias das classes são conectadas, seja fisicamente ou conceitualmente.

### Dependência

São relacionamentos de utilização no qual uma mudança na especificação de um elemento pode alterar a especificação do elemento dependente. A dependência entre classes indica que os objetos de uma classe usam serviços dos objetos de outra classe.

### Generalização

Tipo de associação (é um). Generalização é a capacidade de identificar as similaridades entre várias classes, com isso criamos um supertipo que encapsula todas as funcionalidades comuns às demais classes filhas. Podemos usar a generalização para agrupar os nossos objetos em um tipo comum.

## Agregação

Tipo de associação (tem um, todo/parte) onde o objeto parte é um atributo do todo, onde o objeto parte somente é criado se o objeto todo ao qual está agregado seja criado. Carro é composto por motor, rodas etc.

## Composição

A composição é muito semelhante à agregação. O relacionamento entre um elemento (o todo) e outros elementos (as partes) onde as partes só podem pertencer ao todo e são criadas e destruídas com ele.

## Especialização

A especialização cria uma classe herdada da generalização onde refina o processo definido na classe pai. Como o próprio nome diz, especializa a classe pai para um tipo específico.

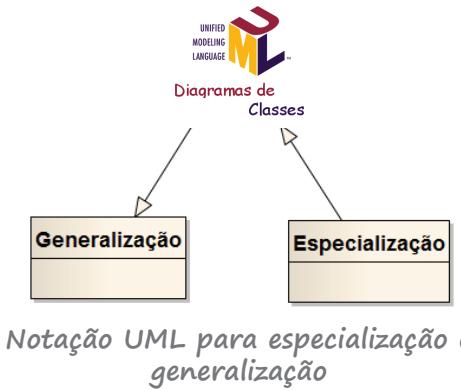
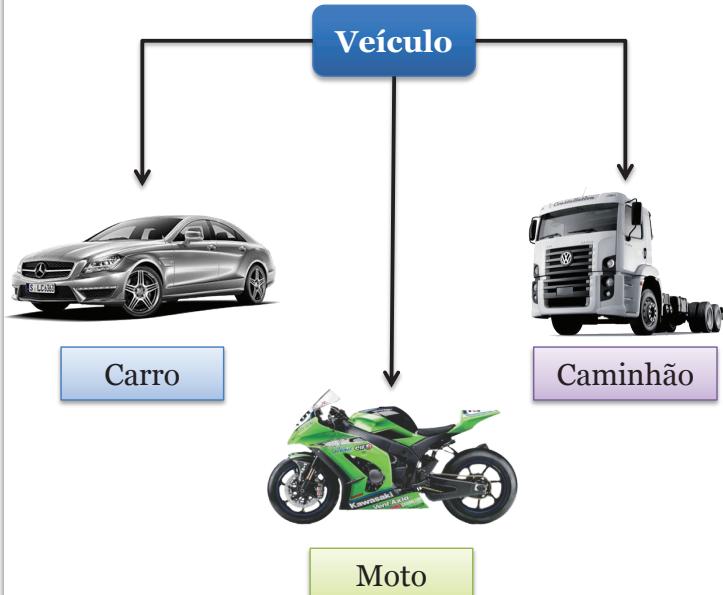
Usando a mesma imagem acima da generalização, podemos identificar quem é a generalização “subindo” e quem é a especialização “descendo”.

## Realização

A realização é um relacionamento entre os itens que implementa o comportamento especificado por outro. Um exemplo disso seria as classes abstratas e as interfaces que definem que o objeto “filho” deverá realizar algum método, propriedade no momento da herança.

# Herança

*Relacionamentos do tipo é um*



## Java e Orientação a Objetos

Generalização e herança são abstrações poderosas para compartilhar similaridades entre classes e ao mesmo tempo preservar suas diferenças.

Generalização é o relacionamento entre uma classe e um ou mais versões refinadas (especializadas) desta classe. A classe sendo refinada é chamada de superclasse ou classe base, enquanto que a versão refinada da classe é chamada uma subclasse ou classe derivada. Atributos e operações comuns a um grupo de classes derivadas são colocados como atributos e operações da classe base, sendo compartilhados por cada classe derivada. Diz-se que cada classe derivada herda as características de sua classe base. Algumas vezes, generalização é chamada de relacionamento (é-um), porque cada instância de uma classe derivada é também uma instância da classe base.

Uma classe filha (derivada) pode sobrepor uma característica ou comportamento de sua classe pai (base) definindo uma característica própria com o mesmo nome. A característica local (da classe filha) irá refinar e substituir a característica da classe base.

# Agregação

*Relacionamentos do tipo tem um*



## Java e Orientação a Objetos

Você pode construir novas classes de objetos por agregação de objetos já existentes, o objeto agregado é constituído de outros objetos.

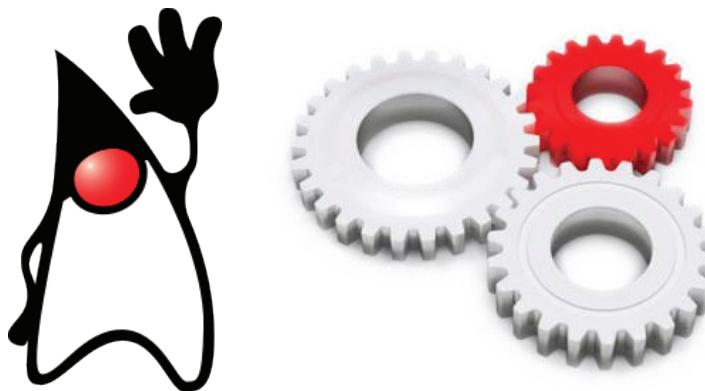
Java implementa o conceito de agregação de objetos pelo uso da referência. Objetos em Java não podem conter outros objetos explicitamente. Objetos somente poderão conter variáveis de tipos primitivos e referências a outros objetos.

Composição ou Agregação é um mecanismo de reaproveitamento (reutilização) de classes utilizado pela POO para aumentar a produtividade e a qualidade no desenvolvimento de software.

Reaproveitamento ou reutilização de classes significa que você pode usar uma ou várias classes para compor outra classe. Já o aumento de produtividade está relacionado com a possibilidade de não ser necessário reescrever código de determinadas classes, se alguma outra já existe com estado (atributos) e comportamento similar. Finalmente, com a composição, é possível também aumentar a qualidade dos sistemas gerados, porque há a possibilidade clara de reutilizar classes que já foram usadas em outros sistemas, e, portanto, já foram testadas e têm chances de conter menos erros.

# Java e Orientação a Objetos

## Métodos, Construtores e Membros Estáticos



## Java e Orientação a Objetos

Uma **classe** é um modelo usado para definir vários objetos com características semelhantes, nos próximos módulos iremos entender melhor o conceito de classe. Os elementos básicos de uma classe são chamados **membros** da classe e podem ser divididos em duas categorias:

- **As variáveis**, que especificam o estado da classe ou de um objeto instância desta classe.
- **Os métodos**, que especificam os mecanismos pelos quais a classe ou um objeto instância desta classe podem operar.

Quando usamos a palavra reservada ‘**new**’ para criar um objeto de uma determinada classe estamos alocando um espaço na memória. Esse ato de inicializar, ou construir, é feito pelos construtores, que são métodos. Iremos estudar mais a fundo posteriormente.

Os membros estáticos são campos, métodos, eventos ou propriedades que podem ser chamados em uma classe mesmo que não tenha sido criada uma instância desta classe.

# Declarando Membros: Variáveis e Métodos

```
public class Carro { // nome da classe

    // (1)Atributos - Variáveis
    private String modelo;
    private String cor;
    private int ano;
    private String placa;

    // (2)Construtor
    public Carro() {
        System.out.println("Criando objeto Carro");
    }

    // (3)Métodos
    public String acelerar() {
        return "Acelerando";
    }
    public String frear() {
        return "Freando";
    }
    public String ligar() {
        return "Ligando";
    }
}
```



## Java e Orientação a Objetos

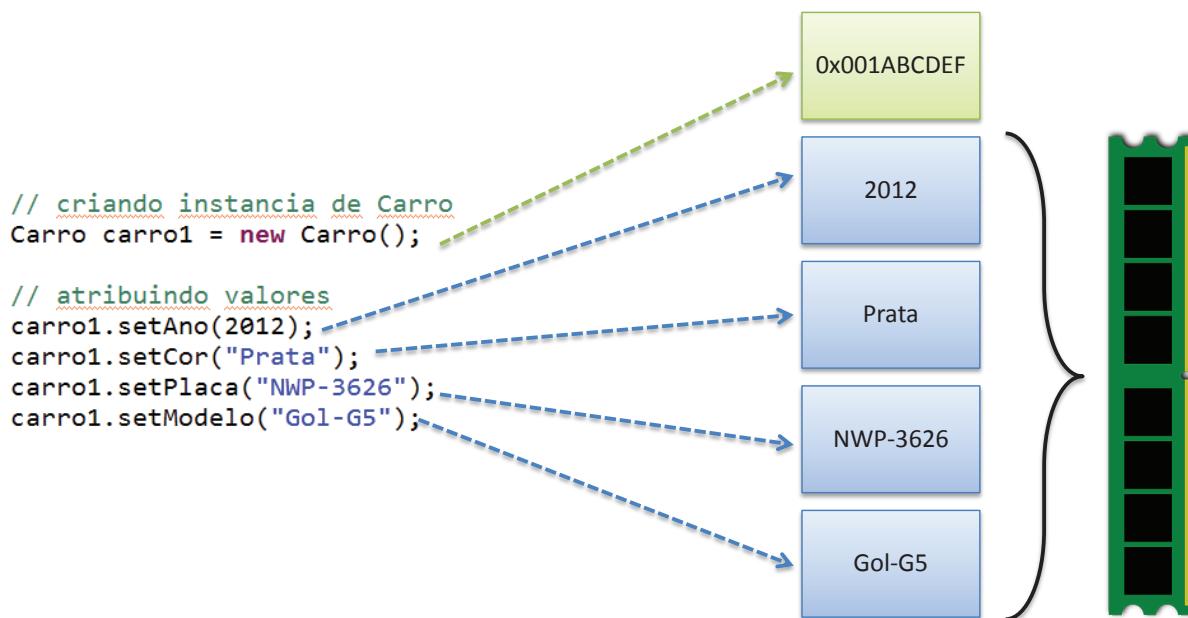
A classe acima mostra a implementação em Java da definição da classe mostrada. O exemplo, apesar de não se preocupar com a utilidade desta implementação, mostra as principais características da definição de uma classe em Java.

A classe possui quatro variáveis, toda variável em Java deve ter um tipo, possui três métodos representando as operações disponíveis aos objetos da classe e um método especial com o mesmo nome da classe, este método é denominado construtor e seu propósito é inicializar o objeto quando este é criado a partir da classe.

Cada objeto criado deverá ter sua própria instância de variáveis (atributos) definidas pela classe. Os métodos definem o comportamento de um objeto. Isto é importante, pois denota que um método pertence a cada objeto da classe. Porém, não devemos confundir isto com a implementação do método, que é compartilhada por todas as instâncias da classe.

String é uma classe em Java, ela armazena um conjunto de caracteres. Como estamos aprendendo o que é uma classe, detalharemos a classe String posteriormente.

# Referência de Objetos



## Java e Orientação a Objetos

Os programas utilizam as variáveis de tipos por referência para armazenar as localizações de objetos na memória do computador. Esses objetos que são referenciados podem conter várias variáveis de instância e métodos dentro do objeto apontado.

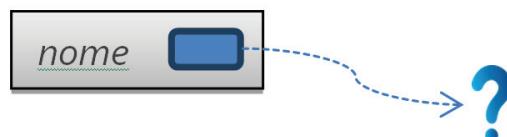
Para trazer em um objeto os seus métodos de instância, é preciso ter referência a algum objeto. As variáveis de referência são inicializadas com o valor “null” (nulo).

Por exemplo, ClasseCarro acao = new ClasseCarro (), cria um objeto de classe ClasseCarro e a variável acao contém uma referência a esse objeto ClasseCarro, onde poderá invocar todos os seus métodos e atributos da classe. A palavra chave new solicita a memória do sistema para armazenar um objeto e inicializa o objeto.

Quando se declara uma variável cujo tipo é o nome de uma classe, como em:

**String nome;**

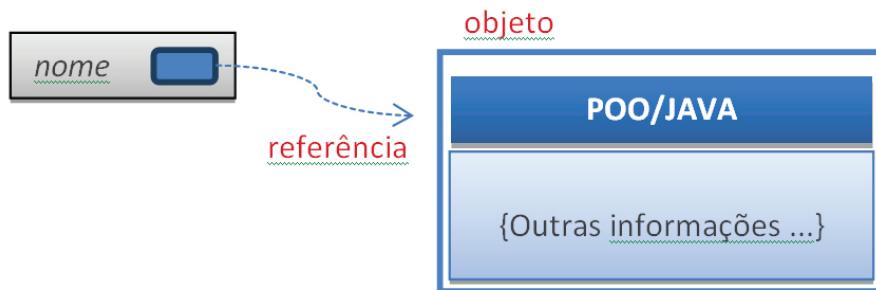
Não está se criando um objeto dessa classe, mas simplesmente uma referência para um objeto da classe String, a qual inicialmente não faz referência a nenhum objeto válido:



Quando um objeto dessa classe é criado, obtém-se uma referência válida, que é armazenada na variável cujo tipo é o nome da classe do objeto. Por exemplo, quando cria-se uma string como em:

**nome = new String("POO/Java");**

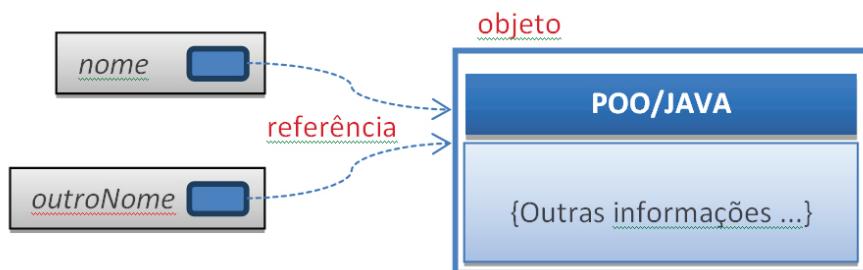
**nome** é uma variável que armazena uma referência para um objeto específico da classe **String** -- o objeto cujo conteúdo é “POO/Java”:



É importante ressaltar que a variável nome mantém apenas a referência para o objeto e não o objeto em si. Assim, uma atribuição como:

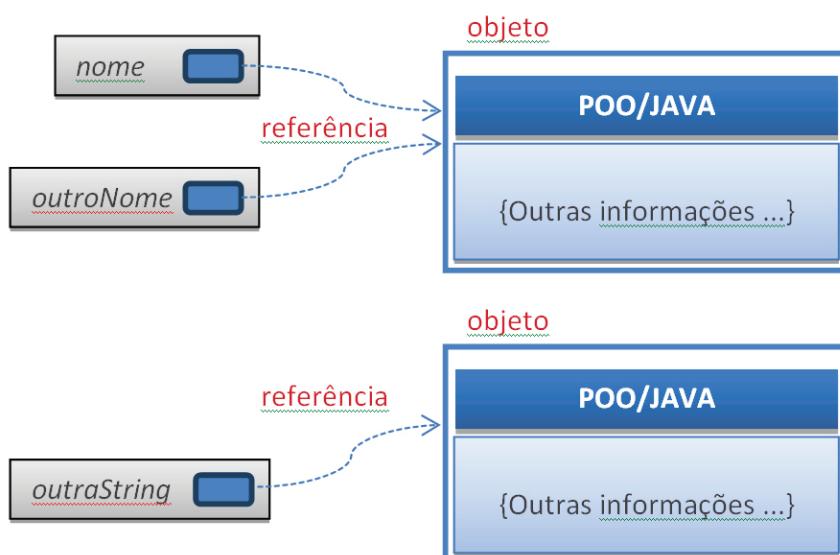
```
String outroNome = nome;
```

Não cria outro objeto, mas simplesmente outra referência para o mesmo objeto:



O único modo de aplicar os métodos a um objeto é através de uma referência ao objeto. Seguindo com o mesmo exemplo, para criar um novo objeto como mesmo conteúdo do objeto existente, o método `clone()` pode ser aplicado a este:

```
String outraString = nome.clone();
```



# Invocação de Métodos

```
nomeDoObjeto.nomeDoMétodo([argumentos separados por ',']);
```

```
// criando instancia de Carro
Carro carro1 = new Carro();

// invocando métodos
carro1.ligar();
carro1.acelerar();
carro1.freiar();
```



## Java e Orientação a Objetos

Objetos se comunicam pela troca de mensagens, isto significa que um objeto pode ter que mostrar um comportamento particular invocando uma operação apropriada que foi definida no objeto. Em Java, isto é feito pela chamada de um método de um objeto usando o operador binário “.”(ponto), devendo especificar a mensagem completa: o objeto que é o recebedor da mensagem, o método a ser invocado e os argumentos para o método (se houver). O método invocado no recebedor pode também enviar informações de volta ao objeto chamador através de um valor de retorno.

Para ilustrar como chamar os métodos, utilizaremos como exemplo a classe String. Pode-se usar a documentação da API Java para conhecer todos os atributos e métodos disponíveis na classe String.

Vamos pegar dois métodos encontrados na classe String como exemplo:

### Métodos encontrados na classe String

Declaração do método	Definição
public char charAt(int index)	Retorna o caractere especificado no índice.
public boolean equalsIgnoreCase (String anotherString)	Compara o conteúdo de duas Strings, ignorando maiúsculas e minúsculas.

## Usando os métodos

```
String str1 = "Hello";
char x = str1.charAt(0);
// retornará o caracter H e o armazenará no atributo x
```

```
String str2 = "hello";
// aqui será retornado o valor booleano
```

# Passagem de Parâmetro por Valor

```

public class PassagemPorValor {
    public static void main(String[] args) {
        int i = 10;
        // exibe o valor de i
        System.out.println(i);

        // chama o método teste
        // envia i para o método teste
        teste(i);-----+
        // exibe o valor de i não modificado |
+----->System.out.println(i); |
|   } |
| |
|     public static void teste(int j) { <-----+
|         // muda o valor do argumento |
+-----j = 33;
    }
}
  
```



## Java e Orientação a Objetos

Em exemplos anteriores, enviamos atributos para os métodos. Entretanto, não fizemos nenhuma distinção entre os diferentes tipos de atributos que podem ser enviados como argumento para os métodos.

Há duas formas para se enviar argumentos para um método, o primeiro é envio por valor e o segundo é envio por referência.

### Envio por valor

Quando ocorre um envio por valor o que significa: “passar por cópia dos bits de variável”, a chamada do método faz uma cópia do valor do atributo e o reenvia como argumento. O método chamado não modifica o valor original do argumento mesmo que estes valores sejam modificados durante operações de cálculo implementadas pelo método.

No exemplo dado, o método teste() foi chamado e o valor de i foi enviado como argumento. O valor de i é copiado para o atributo j do método. Já que j é o atributo modificado no método teste(), não afetará o valor do atributo i, o que significa uma cópia diferente do atributo.

Como padrão, todo tipo primitivo, quando enviado para um método, utiliza a forma de envio por valor.

# Passagem de Parâmetro por Referência

```

public class PassagemPorReferencia {
    public static void main(String[] args) {
        // criar um array de inteiros
        int[] idades = { 10, 11, 12 };
        // exibir os valores do array
        for (int i = 0; i < idades.length; i++) {
            System.out.println(idades[i]);
        }
        // chamar o método teste e enviar a
        // referência para o array
        teste(idades);
        // exibir os valores do array
        for (int i = 0; i < idades.length; i++) {
            System.out.println(idades[i]);-----+
        }
    }
}
----->public static void teste(int[] arr) {
    // mudar os valores do array
    for (int i = 0; i < arr.length; i++) {
        arr[i] = i + 50;-----+
    }
}
  
```



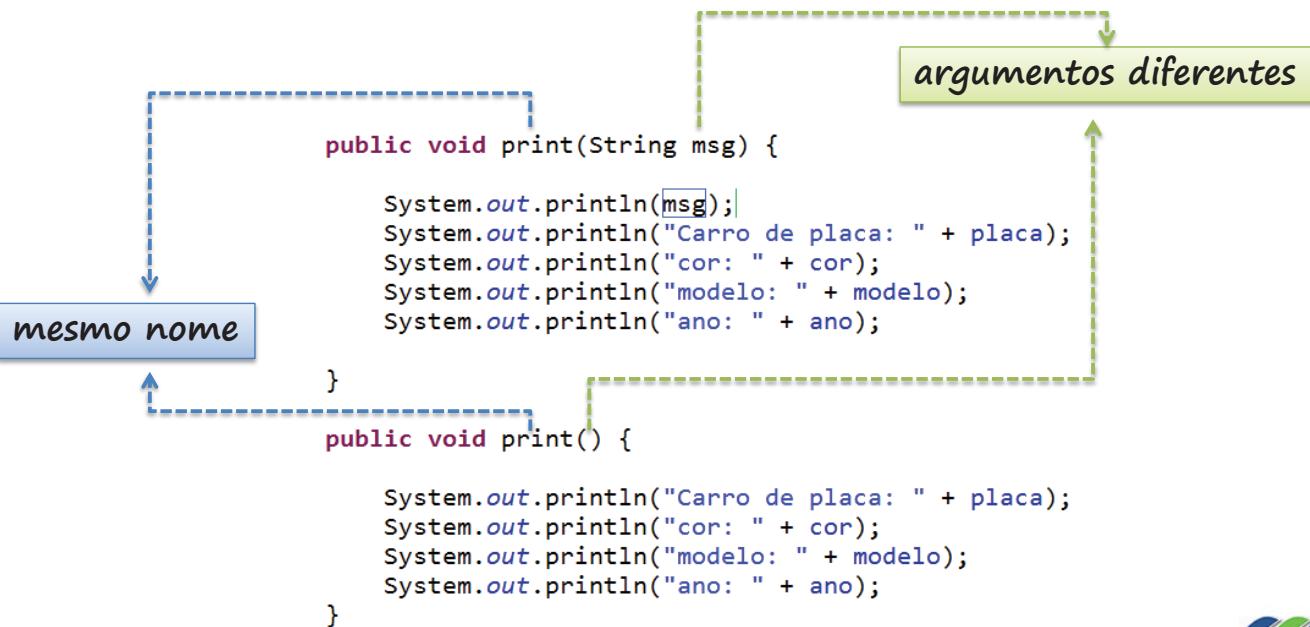
## Java e Orientação a Objetos

Quando ocorre um envio por referência, a referência de um objeto é enviada para o método chamado. Isto significa que o método faz uma cópia da referência do objeto enviado.

Entretanto, diferentemente do que ocorre no envio por valor, o método pode modificar o objeto para o qual a referência está apontando. Mesmo que diferentes referências sejam usadas nos métodos, a localização do dado para o qual ela aponta é a mesma.

Quando passamos um parâmetro por valor à função recebe uma cópia do argumento que está sendo enviado, enquanto que quando passamos o valor por referência, passamos na verdade a referência do argumento, ou seja, seu endereço na memória. Na prática, quando passamos o argumento por valor, mesmo que haja uma alteração do argumento dentro da função, essa alteração não reflete na variável externa, enquanto que na referência quando existe uma alteração dentro do argumento da função ela refletirá diretamente na variável externa, alterando seu valor.

# Sobrecarga de métodos (Overloading)



## Java e Orientação a Objetos

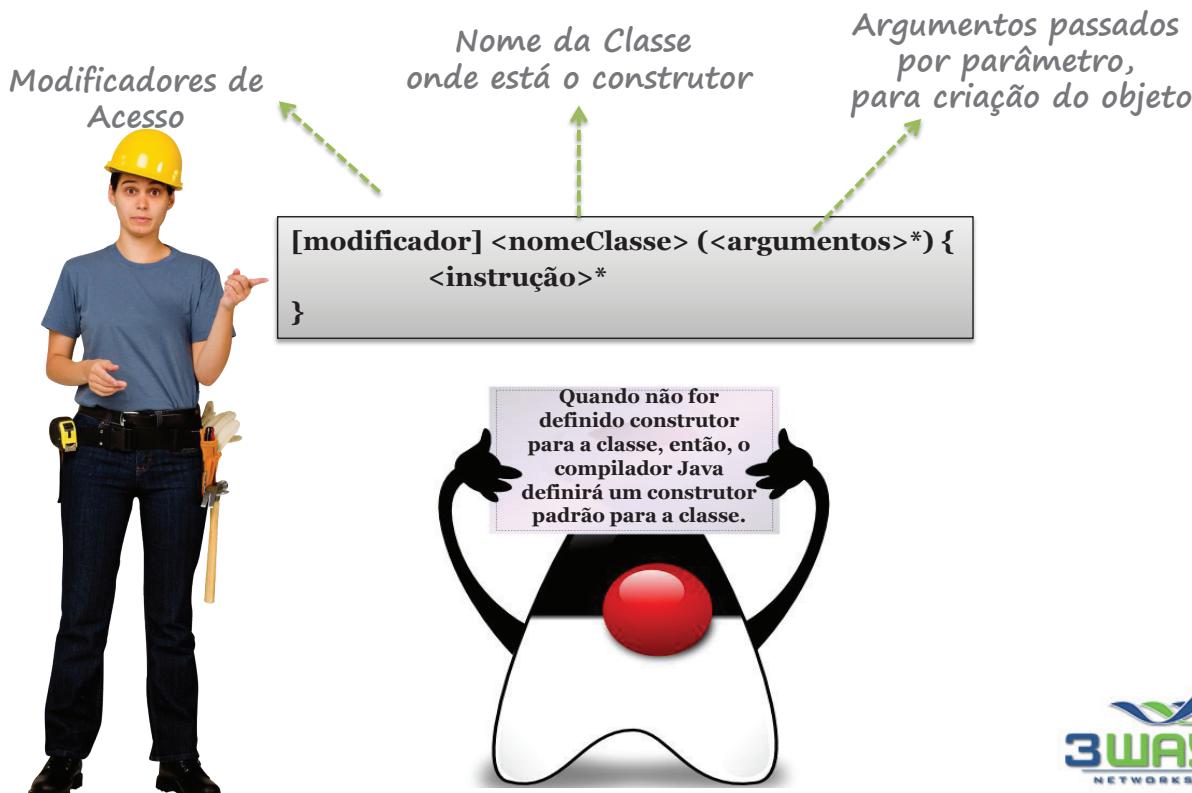
O termo sobrecarga vem do fato de declararmos vários métodos com o mesmo nome, estamos carregando o aplicativo com o ‘mesmo’ método. A única diferença entre esses métodos são seus parâmetros e/ou tipo de retorno.

Nas nossas classes, podemos necessitar de criar métodos que tenham os mesmos nomes, mas que funcionem de maneira diferente dependendo dos argumentos que informamos. Esta capacidade é chamada de overloading de métodos.

Embora os nomes sejam os mesmos, elas atuam de forma diferente e ocupam espaços diferentes em memória, pois estão lidando com tipos diferentes de variáveis. Quando invocamos o método com um inteiro como um argumento, o Java é inteligente o suficiente para invocar corretamente o método que foi declarado com inteiro como parâmetro.

Por exemplo: caso invoquemos o método usando um double como um argumento, o método a ser executado será aquele que foi declarado com o tipo double em seu parâmetro, mesmo que tenha outros métodos com o mesmo nome com outros parâmetros.

# Construtores



## Java e Orientação a Objetos

O (pseudo-)método construtor determina que ações devem ser executadas quando da criação de um objeto. Em Java, o construtor é definido como um método cujo nome deve ser o mesmo nome da classe e sem indicação do tipo de retorno, nem mesmo void. O construtor é unicamente invocado no momento da criação do objeto através do operador new. O retorno do operador new é uma referência para o objeto recém-criado. O construtor pode receber argumentos, como qualquer método, mas a assinatura de um construtor é diferente de um método, temos as propriedades de um construtor:

- > Devem ter o mesmo nome da classe
- > Construtores são como métodos, entretanto, somente as seguintes informações podem ser colocadas no cabeçalho do construtor:
  - > Escopo ou identificador de acessibilidade (como public, private)
  - > Nome do construtor
  - > Lista de parâmetros formais (opcional)
  - > Somente são invocados usando operador new durante a instanciação da classe ou dentro de outro construtor usando this() e/ou super()

No momento em que um construtor é invocado, a seguinte sequência de ações é executada para a criação de um objeto:

- > O espaço para o objeto é alocado e seu conteúdo é inicializado null.
- > O construtor da classe base é invocado. Se a classe não tem uma superclasse definida explicitamente, a classe Object é a classe base.

> Os membros da classe são inicializados para o objeto, seguindo a ordem em que foram declarados na classe.

> O restante do corpo do construtor é executado.

Seguir essa sequência é uma necessidade de forma a garantir que, quando o corpo de um construtor esteja sendo executado, o objeto já terá à disposição as funcionalidades mínimas necessárias, quais sejam aquelas definidas por seus ancestrais. O primeiro passo garante que nenhum campo do objeto terá um valor arbitrário, que possa tornar erros de não inicialização difíceis de detectar.

## Construtor Padrão (default)

Toda classe tem pelo menos um construtor sempre definido. Se nenhum construtor for explicitamente definido pelo programador da classe, um construtor padrão, que não recebe argumentos, é incluído para a classe pelo compilador Java. No entanto, se o programador da classe criar pelo menos um método construtor, o construtor padrão não será criado automaticamente, se o programador desejar mantê-lo, deverá criar um construtor sem argumentos explicitamente.

# Overloading de Construtores

```

public Carro() {
    //Qualquer código de inicialização aqui
}

public Carro( String placa ) {
    this.placa = placa;
}

public Carro( String modelo, String placa ) {
    this.modelo = modelo;
    this.placa = placa;
}

public Carro( String modelo, String cor, int ano, String placa )
{
    this.modelo = modelo;
    this.cor = cor;
    this.ano = ano;
    this.placa = placa;
}

```



Construtores com diferentes tipos de parâmetros  
(Overloading - Sobrecarga)



## Java e Orientação a Objetos

Construtores podem sofrer sobrecarga (overloading) como se fossem métodos, como no exemplo, temos quatro construtores.

A prática de sobrecarga vista anteriormente, também é aplicada a criação de construtores. Ou seja, uma mesma classe pode ter vários construtores, porém com parâmetros e inicializações de objeto diferentes.

Nas nossas classes, podemos necessitar de criar métodos construtores que tenham os mesmos nomes, mas que inicializam o objeto com diferentes argumentos que informamos.

# Utilizando o Construtor `this()`

```

public Carro() {
    System.out.println("Criando objeto Carro");
}

public Carro( String placa ) {
    this();
    this.placa = placa;
}

public Carro( String modelo, String placa ) {
    this();
    this.modelo = modelo;
    this.placa = placa;
}

public Carro( String modelo, String cor, int ano, String placa ) {
    this();
    this.modelo = modelo;
    this.cor = cor;
    this.ano = ano;
    this.placa = placa;
}
  
```



**As chamadas ao construtor DEVE SEMPRE OCORRER NA PRIMEIRA LINHA DE INSTRUÇÃO.**



## Java e Orientação a Objetos

This é usado para fazer autorreferência ao próprio contexto em que se encontra. Resumidamente, this sempre será a própria classe ou o objeto já instanciado.

Esse conceito de autorreferência é importante para que possamos criar métodos construtores sobrecarregados e métodos assessores mais facilmente.

Por base, se criarmos um método que receba um argumento chamado ligado que queremos atribuir para o atributo da classe, que também se chama ligado, devemos diferenciar ambos mostrando quem cada um pertence. Como this se refere ao contexto empregado, então o usamos para identificar que ligado será o atributo da classe e ligado sem o this se refere ao parâmetro do método.

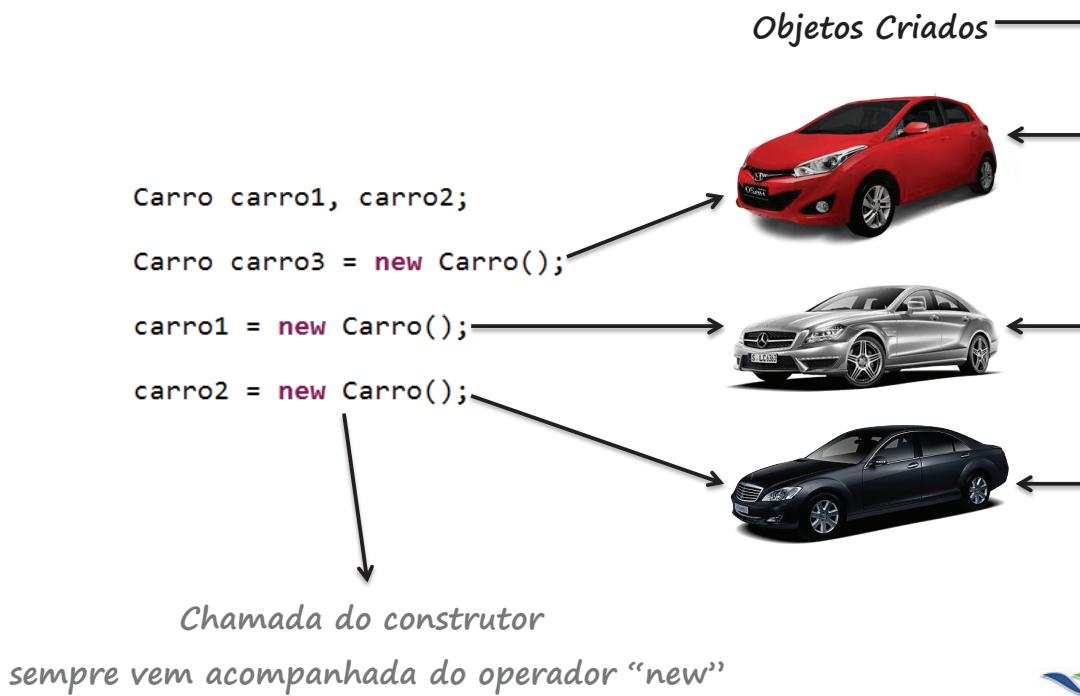
Chamadas a construtores podem ser encadeados, o que significa ser possível chamar um construtor a partir de outro construtor. Usamos uma invocação a `this()` para isso.

Há algumas regras para utilização da chamada ao construtor por `this()`:

- > A chamada ao construtor **deve sempre ocorrer na primeira linha de instrução;**
- > **Utilizado para a chamada de um construtor.** A chamada ao `this()` pode ser seguida por outras instruções.

Como boa prática de programação, é ideal nunca construir métodos que repitam as instruções. Buscamos a utilização de sobrecarga com o objetivo de evitarmos essa repetição.

# Instância de Classes



## Java e Orientação a Objetos

Em programação orientada a objetos, chama-se **instância** de uma classe, um objeto cujo comportamento e estado são definidos pela classe.

Instância é a concretização de uma classe. Em termos intuitivos, uma classe é como um “molde” que gera instâncias de certo tipo; um objeto é algo que existe fisicamente e que foi “moldado” na classe.

Assim, em programação orientada a objetos, a palavra “instanciar” significa criar. Quando falamos em “instanciar um objeto”, criamos fisicamente uma representação concreta da classe. Por exemplo: “animal” é uma classe ou um molde; “cachorro” é uma instância de “animal” e apesar de carregar todas as características do molde de “animal”, é completamente independente de outras instâncias de “animal”.

Em Java, objetos são manipulados através de **referências de objetos** (ou simplesmente **referência**). O processo de criação de um objeto normalmente envolve os seguintes passos:

**Declaração de uma variável de referência:** isto corresponde a declarar uma variável de uma classe apropriada para armazenar a referência ao objeto criado.

Para criar um objeto ou uma instância da classe, utilizamos o operador new. O operador new aloca a memória para o objeto e retorna uma referência para essa alocação. Ao criar um objeto, invoca-se, na realidade, o construtor da classe. O construtor é um método onde todas as inicializações do objeto são declaradas e possui o mesmo nome da classe.

# Membros estáticos

```

public class Carro { // nome da classe

    // (1)Atributos - Variáveis
    private String modelo;
    private String cor;
    private int ano;
    private String placa;
    // declaração de variável estática
    static int contador;

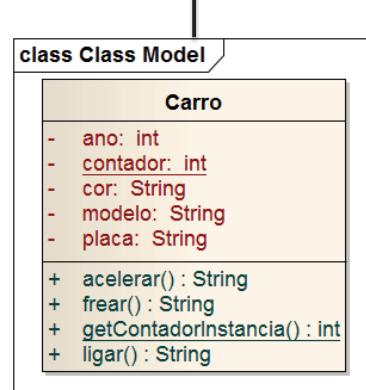
    // (2)Construtor
    // modificação para implementar contador de instâncias
    public Carro() {
        contador++;
        System.out.println("Criando objeto Carro");
    }

    // (3)Métodos
    public String acelerar() { return "Acelerando"; }
    public String frear() { return "Freando"; }
    public String ligar() { return "Ligando"; }

    // método estático
    public static int getContadorInstancia() {
        return contador;
    }
}

```

Representação UML  
de atributos e  
métodos estáticos



## Java e Orientação a Objetos

Só pelo nome, já dá pra desconfiar que seja algo relacionado com constante, algo ‘parado’ (estático).

Quando definimos uma classe e criamos vários objetos dela, já sabemos que cada objeto irá ser uma cópia fiel da classe, porém com suas próprias variáveis e métodos em lugares distintos da memória.

Ou seja, o objeto ‘fusca’ tem suas variáveis próprias, diferentes do objeto ‘Ferrari’, embora ambos tenham o mesmo ‘modelo’, que é a classe ‘Carro’.

Quando definimos variáveis com a palavra static em uma classe ela terá um comportamento especial: ela será a mesma para todos os objetos daquela classe.

Ou seja, não haverá um tipo dela em cada objeto. Todos os objetos, ao acessarem e modificarem essa variável, irão acessar a mesma variável, o mesmo espaço da memória, e a mudança poderá ser vista em todos os objetos.

# Membros estáticos

**NomeClasse.nomeMetodoEstatico(argumentos);**

Métodos que podem ser invocados sem que um objeto tenha sido instanciado pela classe (sem invocar a palavra chave new)

Pertencem à classe como um todo e não a uma instância (ou objeto) específico da classe

```
Carro carro1, carro2;
Carro carro3 = new Carro();
carro1 = new Carro();
carro2 = new Carro();
System.out.println(Carro.getContadorInstancia() + " instâncias criadas");
```



## Java e Orientação a Objetos

Há casos em que você precisará que alguns atributos sejam compartilhados por todos os objetos de uma classe em particular, ou seja, esses atributos devem pertencer somente à Classe e não aos objetos criados a partir dessa classe. Por exemplo, imagine que você queira manter um controle de quantos objetos de uma classe foram criados e permitir que todos os objetos consiga recuperar esta informação. Definir um contador como uma variável de instância na definição da classe não resolveria o problema, pois já sabemos que são criadas cópias desta variável para cada objeto criado a partir da definição de uma classe. Então como resolver o problema? A solução é usar variáveis estáticas - usar a palavra chave static antes da declaração da variável. Uma variável estática é inicializada quando a classe é carregada em tempo de execução.

### Quando usar variáveis static em Java

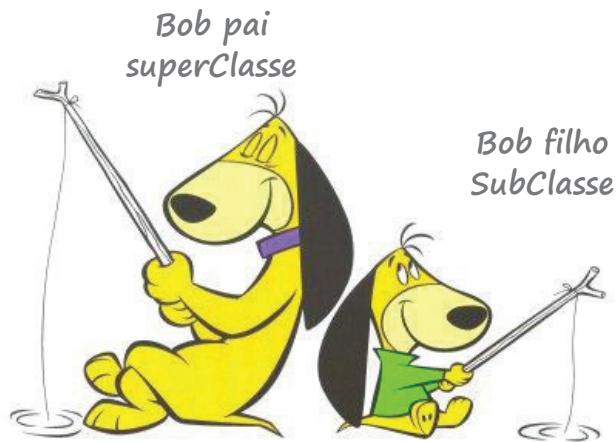
Principalmente quando você quiser ter um controle sobre os objetos ou quando todos os objetos devem partilhar uma informação (evitar ter que fazer Composição ou chamar métodos de outros objetos).

#### **Exemplo 1:** Para controle de número total de objetos

Imagine que você é dono de uma loja de venda de veículos. Cada venda, é um comprador diferente, dados diferentes etc. Portanto, cada carro será um objeto.

Você cria a variável estática ‘total’, e no construtor a incrementa (total++). Pronto, saberá quantos carros foram vendidos, automaticamente.

## Java e Orientação a Objetos Herança e Polimorfismo

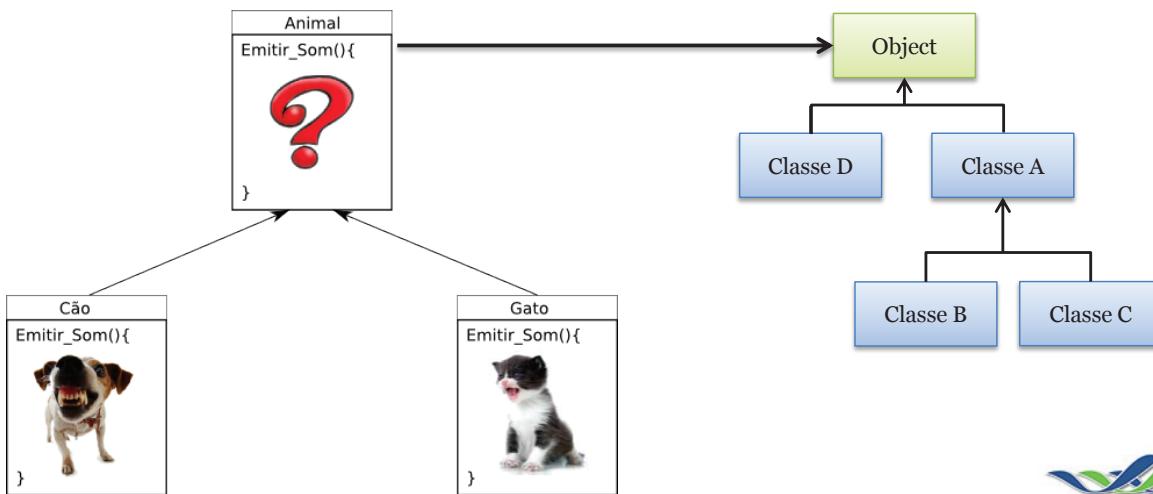


### Java e Orientação a Objetos

O paradigma da **Orientação a Objetos** traz um ganho significativo na qualidade da produção de software, contendo três pilares, Encapsulamento, herança e polimorfismo, além de agregar à programação o uso de boas práticas de programação e padrões de projeto, design patterns.

# Herança

*Em Java, todas as classes, incluindo as que formam a API Java, são **subclasses** da classe **Object***



## Java e Orientação a Objetos

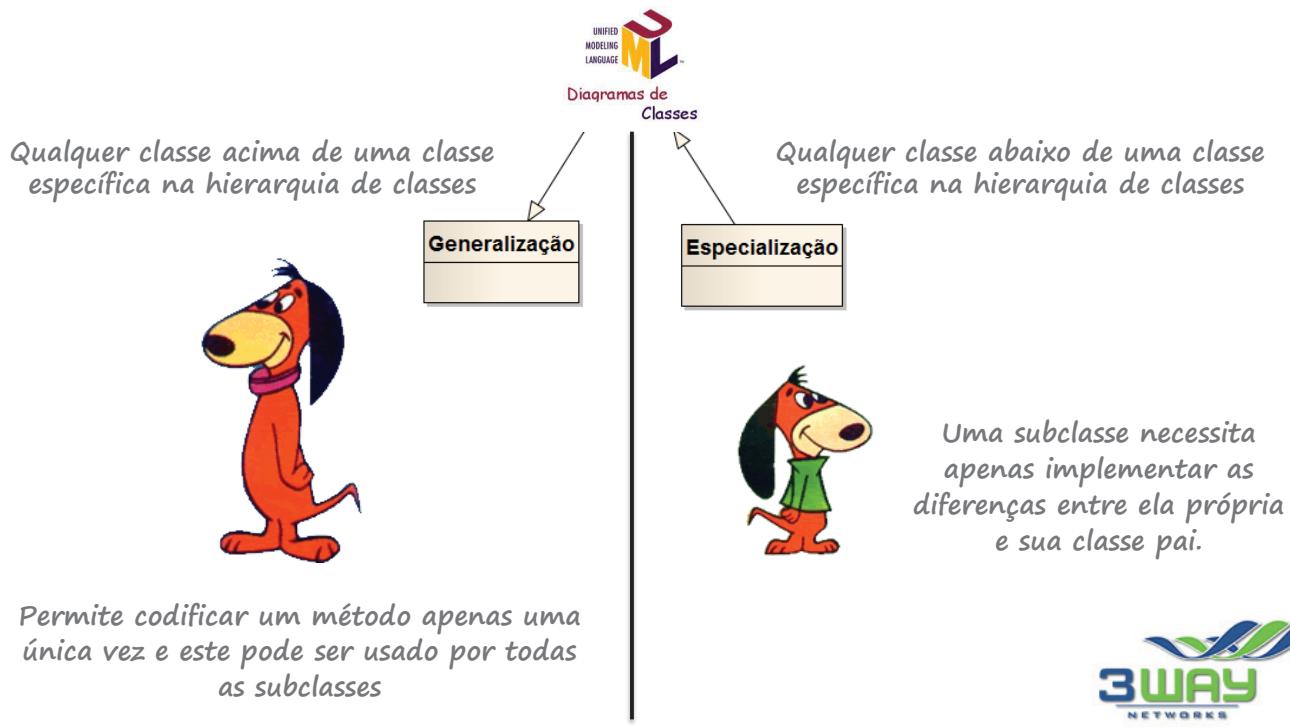
A herança é um mecanismo da Orientação a Objeto que permite criar novas classes a partir de classes já existentes, aproveitando-se das características existentes na classe a ser estendida. Este mecanismo é muito interessante, pois promove um grande reuso e reaproveitamento de código existente. Com a herança é possível criar classes derivadas, subclasses, a partir de classes bases, superclasses. As subclasses são mais especializadas do que as suas superclasses, mais genéricas. As subclasses herdam todas as características de suas superclasses, como suas variáveis e métodos. A linguagem Java permite o uso de herança simples, **mas não permite** a implementação de herança múltipla.

Por exemplo, Imagine que dentro de uma organização empresarial, o sistema de RH tenha que trabalhar com os diferentes níveis hierárquicos da empresa, desde o funcionário de baixo escalão até o seu presidente. Todos são funcionários da empresa, porém cada um com um cargo diferente. Mesmo a secretária, o pessoal da limpeza, o diretor e o presidente possuem um número de identificação, além de salário e outras características em comum. Essas características em comum podem ser reunidas em um tipo de classe em comum, e cada nível da hierarquia ser tratado como um novo tipo, mas aproveitando-se dos tipos já criados, através da herança. Os subtipos, além de herdarem todas as características de seus supertipos, também podem adicionar mais características, seja na forma de variáveis e/ou métodos adicionais, bem como reescrever métodos já existentes na superclasse, polimorfismo.

A herança permite vários níveis na hierarquia de classes, podendo criar tantos subtipos quanto necessário, até se chegar ao nível de especialização desejado. Podemos tratar subtipos como se fossem seus supertipos, por exemplo, o sistema de RH pode tratar uma instância de Presidente como se fosse um objeto do tipo Funcionário, em determinada funcionalidade. Porém não é possível tratar um supertipo como se fosse um subtipo, a não ser que o objeto em questão seja realmente do subtipo desejado e a linguagem suporte este tipo de tratamento, seja por meio de conversão de tipos ou outro mecanismo.

Outro exemplo: Pense no que um aluno, um professor e um funcionário possuem em comum? Todos eles são pessoas e, portanto, compartilham alguns dados comuns. Todos têm nome, idade, endereço, etc. E, o que diferencia um aluno de outra pessoa qualquer? Um aluno possui uma matrícula; Um funcionário possui um código de funcionário, data de admissão, salário, etc.; Um professor possui um código de professor e informações relacionadas à sua formação.

# SuperClasse e SubClasse



## Java e Orientação a Objetos

A classe **Object** é a classe ancestral de todas as outras. O compilador Java, automaticamente, insere a definição da herança para todas as definições de classes.

Na Orientação a Objetos as palavras classe base, supertipo, superclasse, classe pai e classe mãe são sinônimos, bem como as palavras classe derivada, subtipo, subclasse e classe filha também são sinônimos.

As subclasses são mais especializadas do que as suas superclasses, mais genéricas. As subclasses herdam todas as características de suas superclasses, como suas variáveis e métodos. A linguagem Java permite o uso de herança simples, mas não permite a implementação de herança múltipla.

Os atributos private não são visíveis na subclasse, os atributos protected e public mantêm a mesma visibilidade.

No caso de atributos e métodos sem modificador de visibilidade (visibilidade de pacote), a subclasse herda estes atributos e métodos, se estiver definida no mesmo pacote que a superclasse, e apenas neste caso.

# Herança – classe Veiculo

```

public class Veiculo { // nome da classe

    // (1)Atributos - Variáveis
    private String cor;
    private int ano;
    private String identificacao;

    // (2)Construtor
    public Veiculo( String cor, int ano, String identificacao ) {

        this.cor = cor;
        this.ano = ano;
        this.identificacao = identificacao;
        System.out.println("Criando objeto Veiculo");
    }

    // (3)Métodos
    public void mover() {
        System.out.println("Veiculo se movendo");
    }
}
  
```

Veiculo
- ano: int
- cor: String
- identificacao: String
+ mover(): void



## Java e Orientação a Objetos

No exemplo acima foi criado a classe veiculo que será nosso exemplo de superclasse, esta que contem os atributos e métodos que serão herdados pelas classes que estender essa classe pai, também foi criado um construtor para inicializar as variáveis da classe, assim criando o objeto veiculo.

E mostrado a representação dessa classe em UML.

# Herança – classe Carro

```

public class Carro extends Veiculo { // nome da classe

    // (1)Atributos - Variáveis
    private String modelo;

    // (2)Construtor
    public Carro( String cor, int ano, String placaIdentificacao, String modelo ) {

        super(cor, ano, placaIdentificacao);
        this.modelo = modelo;

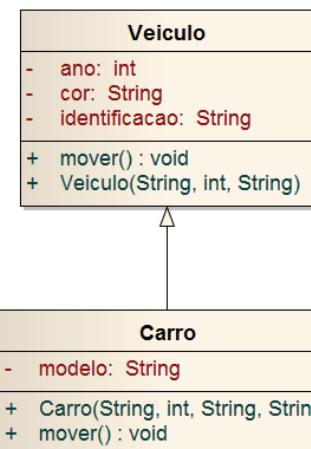
        System.out.println("Criando objeto Carro");
    }

    @Override
    public void mover() {
        System.out.println("Correr");
    }
}

```



*Uma chamada a um construtor super() no construtor de uma subclasse resultará na execução do construtor referente da superclasse, baseado nos argumentos passados.*



## Java e Orientação a Objetos

O exemplo acima foi criado a classe Carro que estende a superclasse Veiculo, assim herdando todos os métodos e variáveis da classe veiculo. Note que está utilizando a notação `@Override` para marcar a sobreposição do método `mover()` da classe veiculo. Também está sendo utilizado a palavra `super` para referenciar o construtor da superclasse Veiculo passando os argumentos necessários como parâmetro.

E mostrado a representação dessa herança em UML.

# Modificador de Classe *final*

*Classes que não podem ter subclasses*



`<modificador>* final class <nomeClasse> { ... }`



Muitas classes na API Java são declaradas **final** para certificar que seu comportamento não seja herdado e, possivelmente modificado.  
Exemplos, são as classes **Integer**, **Double**, **Math** e **String**



## Java e Orientação a Objetos

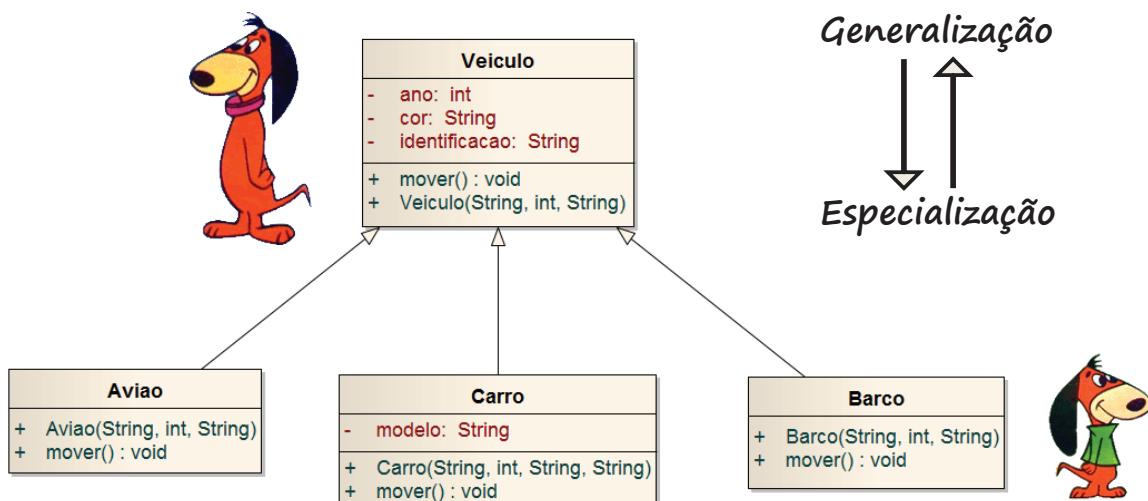
O modificador final pode ser aplicado a variáveis, métodos e classes. O comportamento descrito pelo operador final varia de elemento para elemento, mas tem um conceito em comum, o conceito de imutabilidade, isto é, elementos final não podem ser mudados.

Podemos declarar classes que não permitem a herança. Estas classes são chamadas classes finais. Para definir que uma classe seja **final**, adicionamos a palavra-chave **final** na declaração da classe (na posição do modificador).

Há pelo menos dois bons motivos para se declarar classe final, segurança e design. Em relação à segurança você pode querer declarar sua classe final de modo que uma subclasse não possa ser substituída dentro de um sistema, onde essa nova classe poderá ter códigos indesejáveis ao seu sistema. Um exemplo de uma classe final é a classe String, esta classe é vital para a normal operação da Máquina Virtual Java, já que todos os programas Java precisam dela, pois ela é declarada como parâmetro no método main (String args[]).

O outro motivo é simplesmente uma questão de design, por exemplo, você pode desejar declarar classes que programam seus algoritmos como classes finais.

# Polimorfismo



## Java e Orientação a Objetos

Polimorfismo é o princípio pelo qual duas ou mais classes derivadas de uma mesma superclasse podem invocar métodos que têm a mesma identificação, assinatura, mas comportamentos distintos, especializados para cada classe derivada.

O overloading não é um tipo de polimorfismo, pois com overloading a assinatura do método obrigatoriamente tem que ter argumentos diferentes, requisito que fere o conceito de Polimorfismo citado acima.

De forma genérica, polimorfismo significa várias formas. No caso da Orientação a Objetos, polimorfismo denota uma situação na qual um objeto pode se comportar de maneiras diferentes ao receber uma mensagem, dependendo do seu tipo de criação.

Por exemplo, a operação move quando aplicada a uma janela de um sistema de interfaces tem um comportamento distinto do que quando aplicada a uma peça de um jogo de xadrez. Um método é uma implementação específica de uma operação para certa classe. Desta forma, novas classes podem ser adicionadas sem necessidade de modificação de código já existente, pois cada classe apenas define os seus métodos e atributos. Em Java, o polimorfismo se manifesta apenas em chamadas de métodos.

A decisão sobre qual o método que deve ser selecionado, de acordo com o tipo da classe derivada, é tomada em tempo de execução, através do mecanismo de ligação tardia. A ligação tardia ocorre quando o método a ser invocado é definido durante a execução do programa. Através do mecanismo de sobrecarga, dois métodos de uma classe podem ter o mesmo nome, desde que suas assinaturas sejam diferentes, entretanto isso não é polimorfismo. Como dito anteriormente, tal situação não gera conflito, pois o compilador é capaz

de detectar qual método deve ser escolhido a partir da análise dos tipos dos argumentos do método. Nesse caso, diz-se que ocorre a ligação prematura para o método correto. Em Java, todas as determinações de métodos a executar ocorrem através de ligação tardia exceto em dois casos:

Métodos declarados como final não podem ser redefinidos e, portanto não são passíveis de invocação polimórfica da parte de seus descendentes;

Métodos declarados como private são implicitamente finais.

No caso de polimorfismo, é necessário que os métodos tenham exatamente a mesma identificação, sendo utilizado o mecanismo de redefinição de métodos, que é o mesmo que sob escrita de métodos em classes derivadas. A redefinição ocorre quando um método cuja assinatura já tenha sido especificada recebe uma nova definição, ou seja, um novo corpo, em uma classe derivada. É importante observar que, quando o polimorfismo está sendo utilizado, o comportamento que será adotado por um método só será definido durante a execução. Embora em geral esse seja um mecanismo que facilite o desenvolvimento e a compreensão do código orientado a objetos, há algumas situações onde o resultado da execução pode não ser intuitivo.

# Sobreposição de Métodos @Override



```
public class Veiculo {// nome da classe
    // ...
    public void mover() {
        System.out.println("Veiculo se movendo");
    }
    // ...
}
```

```
public class Barco extends Veiculo {
    // ...
    @Override
    public void mover() {
        System.out.println("Navegar");
    }
    // ...
}
```

```
public class Aviao extends Veiculo {
    // ...
    @Override
    public void mover() {
        System.out.println("Voar");
    }
    // ...
}
```

```
public class Carro extends Veiculo {
    // ...
    @Override
    public void mover() {
        System.out.println("Correr");
    }
    // ...
}
```

*O tipo de retorno do método na subclasse deve ser idêntico ao método sobreposto na superclasse.*



## Java e Orientação a Objetos

Substituir um método como o próprio nome já diz, significa cancelar um método de uma superclasse em uma subclasse dando lhe outro sentido, ou tecnicamente, outro código, portanto só poderá substituir um método se houver herança caso contrário não existe substituição de método.

Considere que a classe **Veículo** possui como descendentes as classes **Carro**, **Avião** e **Barco**, conforme ilustrado. Observe que as classes filhas sobreponem o método `mover()` da classe `Veiculo`, modificando seu comportamento.

As seguintes regras para sobreposição de métodos devem ser seguidas:

> O tipo de retorno do método na subclasse deve ser o mesmo ou um subtipo do tipo de retorno do método sobreposto na superclasse.

> O método que está sobrepondo não pode ser menos acessível que o seu método sobreposto.

> O método que está sobrepondo não pode lançar exceções de tipos diferentes das do método sobreposto. Veremos detalhes no estudo de exceções.

## Sobreposição do Método `toString()`

O **Java** oferece uma implementação padrão para o método `toString` através da classe `java.lang.Object` que é herdada por todas as classes Java. No entanto, o que ela retorna não é muito intuitivo. O seu retorno é composto pelo nome da classe seguido por um arroba (@) e pela representação do código de hash em hexadecimal sem sinal como, por exemplo, `Carro@163b91`, onde **carro** seria o nome da classe seguida pelo arroba e pelo código hash.

O método **toString** deve retornar uma representação concisa, mas informativa, que seja fácil de uma pessoa ler. Temos que concordar que *Carro@163b91 não é muito informativo, no entanto, se tivéssemos como retorno Carro[placa=NWP-1123]* seria muito mais informativo. Fornecer uma boa implementação de **toString** tornará a classe mais agradável de usar. O método **toString** sempre é chamado de forma automática quando um objeto é passado para `println`, `printf`, para o operador de concatenação de Strings ou para `assert`, ou exibido por um depurador.

# Referências polimórficas

→ Referência

```
public static void main(String[] args) {
    Veiculo veiculo = new Carro("Cinza", 2012, "NWP-2552", "Gol");
    veiculo.mover();                                     → Tem que correr
    veiculo = new Aviao("Prata", 2013, "NWP-2552");
    veiculo.mover();                                     → Tem que voar
    veiculo = new Barco("Branco", 2008, "NWP-2552");
    veiculo.mover();                                     → Tem que navegar
}
```

Instância



Uma variável de referência de uma classe mais genérica (superclasse) pode receber referência de objetos de classes mais especializadas (as subclasses).



## Java e Orientação a Objetos

Uma variável de referência de uma classe mais genérica (superclasse) pode receber referência de objetos de classes mais especializadas (as subclasses).

**Polimorfismo** permite que referências de tipos de classes mais **abstratas (genéricas)** representem o comportamento das classes concretas que referenciam.

O método **mover()**, definido como uma característica da classe **Veiculo**, foi sobreposto pelas versões do método nas subclasses **Carro**, **Avião** e **Barco**, embora possuam a mesma assinatura (mesmo nome, número de parâmetros e tipos dos parâmetros), eles apresentam resultados com formas diferentes.

Boa parte dos padrões de projeto de software baseia-se no uso de polimorfismo, por exemplo: Abstract Factory, Composite, Observer, Strategy, TemplateMethodetc;

O polimorfismo também é usado em uma série de refatorações, como substituir condicional por polimorfismo.

# Coleções Heterogêneas de Objetos

```

Carro [] carros = new Carro [3];
// coleção de carros
carros[0] = new Carro("Cinza", 2012, "NWP-2552", "Gol");
carros[1] = new Carro("Preto", 1995, "PAX-4589", "Pálio");
carros[2] = new Carro("Vermelho", 2000, "XPY-3895", "Celta");

Barco [] barcos = new Barco [2];
// coleção de Barco
barcos[0] = new Barco("Verde", 1999, "Naúfrago");
barcos[1] = new Barco("Preto", 1312, "Pérola Negra");

// criar coleção
Veiculo [] veiculo = new Veiculo [4];
// atribui referência a coleção
veiculo[0] = new Carro("Cinza", 2012, "NWP-2552", "Gol");
veiculo[1] = new Barco("Preto", 1312, "Pérola Negra");
veiculo[2] = new Carro("Preto", 1995, "PAX-4589", "Pálio");
veiculo[3] = new Aviao("Branco", 2010, "Boing 737");
  
```

O recurso do polimorfismo nos permite criar um único array para nossa coleção de animais.



## Java e Orientação a Objetos

Em nosso estudo sobre **arrays**, vimos que ele é uma estrutura de dados que armazena diversos elementos do mesmo tipo de dados, desde tipos primitivos bem como referências para outros objetos. Agora imagine formando um **petshop**, onde temos definidos, além das classes Cachorro e Gato, as classes Canarioe Coelho. Sem **polimorfismo**, essa tarefa nos levaria a definir uma estrutura de **array** para cada tipo de animal em nossa loja.

O recurso do polimorfismo nos permite criar um único **array** para nossa coleção de animais.

Uma vez que toda **classe** em Java **estende** a classe **Object**, é possível manipular coleções genéricas para todos os tipos de dados, inclusive de valores de tipos primitivos (neste caso devemos fazer uso das classes empacotadoras).

## Exemplo

```

Object[] primutivos = new Object[3];
primutivos[0] = new Integer(10);
primutivos[1] = new Double(2.2);
primutivos[2] = new Character("C");
  
```

# Determinando a Classe de um Objeto

```

// criar coleção
Veiculo [] veiculo = new Veiculo [4];
// atribui referência a coleção
veiculo[0] = new Carro("Cinza", 2012, "NWP-2552", "Gol");
veiculo[1] = new Barco("Preto", 1312, "Pérola Negra");
veiculo[2] = new Carro("Preto", 1995, "PAX-4589", "Pálio");
veiculo[3] = new Aviao("Branco", 2010, "Boing 737");

Carro      ↑-----System.out.println(veiculo[0].getClass().getSimpleName());
Barco      ↓-----System.out.println(veiculo[1].getClass().getSimpleName());

          System.out.println(veiculo[0] instanceof Carro); Retorna true se retornar
          System.out.println(veiculo[1] instanceof Barco); objeto do tipo Especificado
  
```



Um dos problemas que enfrentaremos ao lidar com referências genéricas para objetos de subclasses, é que não sabemos mais a qual classe pertence a referência armazenada.



## Java e Orientação a Objetos

Um dos problemas que enfrentaremos ao lidar com referências genéricas para objetos de subclasses, é que não sabemos mais a qual classe pertence a referência armazenada. Reveja nosso exemplo da coleção de veículos, a qual classe pertence a referência de objeto armazenada em **veiculo[2]**? Tudo bem você olhou o código-fonte e descobriu que é do tipo **Carro**, mas nem sempre isso é possível. Então como determinar a classe de um **objeto**?

Existem duas maneiras de se descobrir a qual classe determinado objeto pertence:

**Obtendo-se o nome da classe:** Utiliza-se o método **getClass()** que retorna a classe do objeto (onde Class é a classe em si) que possui o método chamado **getSimpleName()** que retorna o nome da classe.

**Testar se um objeto qualquer foi instanciado de uma determinada classe:** Utiliza-se a palavra-chave **instanceof**. Esta palavra-chave possui dois operadores: a referência para o objeto à esquerda e o nome da classe à direita. A expressão retorna um booleano dependendo de o objeto ser uma instância da classe declarada ou qualquer uma de suas subclasses.

## Casting de Referências de Objetos

A operação de casting é muito comum na programação orientada a objetos, para entender polimorfismo e retorno covariante é fundamental ter um bom conhecimento dessa operação. Existe o casting de tipos primitivos e casting de objetos, vamos focar em casting de objetos, com objetos temos dois tipos de casting, que são o up e o down.

O castup é uma operação que acontece sem a necessidade de forçar a mudança de tipo, por exemplo, uma instância de Carro pode ser vista como um Veiculo e até mesmo como Object. O castdown é um cast que deve ser explícito, isso acontece porque o compilador não pode garantir a integridade da instância.

Pode-se utilizar os recursos depolimorfismo para referenciar qualquer objeto como Object, isso permite criar coleções heterogêneas. Podemos até passar argumentos genéricos para métodos, um método que recebe uma referência a Object como argumento, isto é, qualquer objeto criado em Java. Todas as vezes que provocamos uma mudança no tipo da referência de **subclasse** para a **superclasse** dizemos que houve **up-casting**.

### Exemplo

```
//1  
Carro carro = new Carro();  
//2  
Veiculo veiculo = carro;  
//3  
Object objeto = veiculo;  
//4  
Carro carro2 = (Carro)veiculo;
```

**1)** Criação de um objeto do tipo Carro;

**2)** **up-casting**, não há necessidade de forçar a mudança de tipo, pois Carro é um Veiculo;

**3)** **up-casting**, Veiculo é um Object;

**4)** **down-casting**, necessário para garantir a integridade da instância, pois o compilador não consegue saber se realmente se trata do tipo especificado, porque a instância é criada em tempo de execução, e em caso dos tipos serem incompatíveis é lançada uma exceção(erro).

# Modificador de método *final*



*Impede o polimorfismo das subclasses por override*

```
<modificador> final <tipo retorno> <nome>(<parâmetros>)<clausula throws> { ... }
```



## Java e Orientação a Objetos

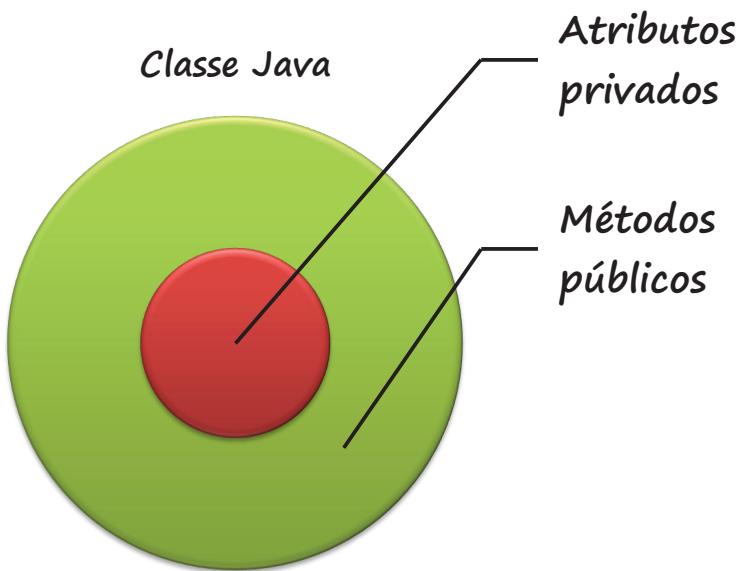
Métodos marcados como **final** são o que chamamos de métodos finais. Para declarar um método final, adicionamos a **palavra-chave final** na declaração do método (na posição do modificador).

Assim como uma classe final não pode ser herdada, gerando um erro de compilação caso o façamos. O mesmo acontecerá ao se tentar fazer um **override** de um **método final**.

Métodos declarados como final não podem ser redefinidos e, portanto não são passíveis de invocação polimórfica da parte de seus descendentes.

Métodos declarados como private são implicitamente finais.

# Encapsulamento



*Para implementar o encapsulamento, temos os modificadores de acesso*



## Java e Orientação a Objetos

Encapsulamento vem de encapsular, que em programação orientada a objetos significa separar o programa em partes, o mais isolado possível. A ideia é tornar o software mais flexível, fácil de modificar e de criar novas implementações. O Encapsulamento serve para controlar o acesso aos atributos e métodos de uma classe. É uma forma eficiente de proteger os dados manipulados dentro da classe, além de determinar onde esta classe poderá ser manipulada. Usamos o nível de acesso mais restritivo, private, que faça sentido para um membro particular. Sempre usamos private, a menos que tenhamos um bom motivo para deixá-lo com outro nível de acesso. Não devemos permitir o acesso público aos membros, exceto em caso de ser constantes. Isso porque membros públicos tendem a nos ligar a uma implementação em particular e limita a nossa flexibilidade em mudar o código. O encapsulamento que é dividido em dois níveis:

> **Nível de classe:** Quando determinamos o acesso de uma classe inteira que pode ser public ou Package-Private (padrão);

> **Nível de membro:** Quando determinamos o acesso de atributos ou métodos de uma classe que podem ser public, private, protected ou Package-Private (padrão).

Então para ter um método encapsulado utilizamos um modificador de acesso que geralmente é public, além do tipo de retorno dele. Para se ter acesso a algum atributo ou método que esteja encapsulado utiliza-se o conceito de get e set. Por definição, com SET é feita uma atribuição a algum atributo, ou seja, define, diz o valor que algum atributo deve ter. E com GET é possível recuperar esse valor.

# Métodos de Configuração e Captura

```
private String cor;

public void setCor(String cor) {
    this.cor = cor;
}
```

possibilita alteração dos valores (variáveis) por outros objetos.

`set<NomeAtributo>(<tipo dado> <parâmetro>)`



```
public String getCor() {
    return cor;
}
```

usados para ler valores de atributos.  
`get<NomeDoAtributo>`.



## Java e Orientação a Objetos

Até agora víhamos cometendo um grande erro na implementação de nossas classes. Nosso erro decorre do fato de deixarmos os dados de nossas classes exposto ao acesso externo. Isso significa que podemos alterar o valor das variáveis de instância através do operador ponto.

Para que se possa implementar o princípio do **encapsulamento**, isto é, não permitir que qualquer objeto acesse os dados de qualquer modo. Para isto declaramos os campos ou atributos, da nossa classe com modificador de acesso **private**. Entretanto, há momentos em que queremos que outros objetos acessem alguns destes atributos, tanto para escrita quanto para leitura. Para que possamos fazer isso, criamos métodos de configuração chamados de **getter e setter**, com **visibilidade pública**.

### Métodos Getter

O método de captura recebe o nome de `get<NomeDoAtributo>()`. Ele **retorna** um objeto do mesmo **tipo do atributo** e deve retornar o **valor do atributo** desejado.

### Métodos Setter

Para que outros objetos possam modificar os atributos da classe, disponibilizamos métodos que possam gravar ou modificar os valores das variáveis de instância. Este método é escrito como `set<NomeDoAtributoDeObjeto>(<tipo do atributo parametrizado>)`.

# Java e Orientação a Objetos

## Classes Abstratas, Internas e Interfaces



## Java e Orientação a Objetos

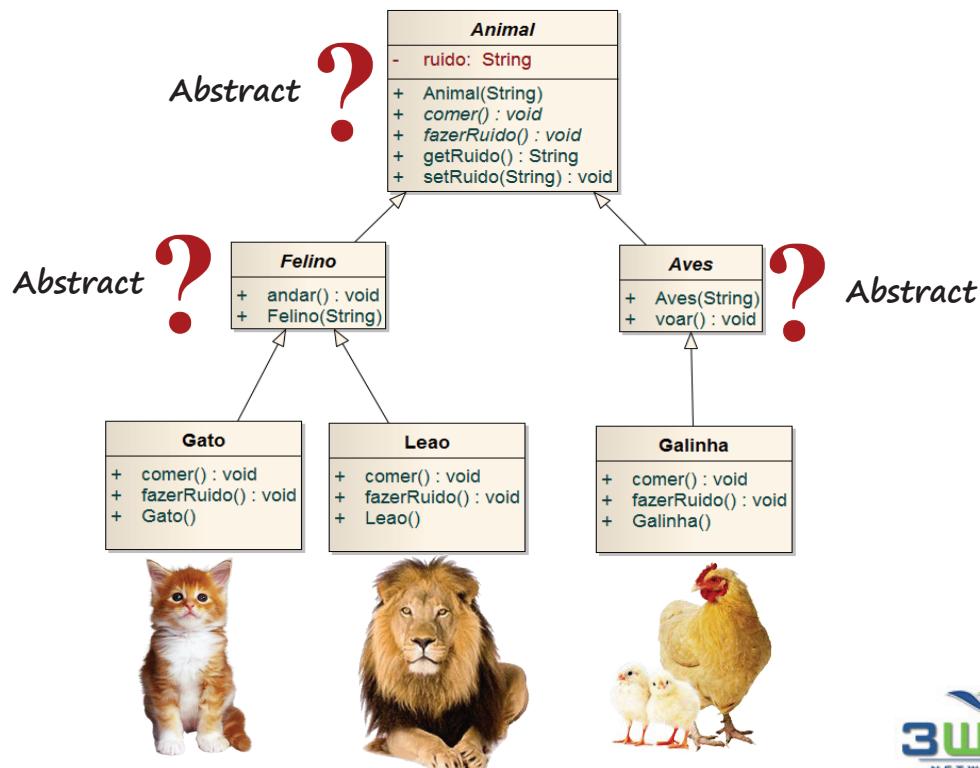
Ao pé da letra, a definição de classe abstrata é: classes que não serão instanciadas. Ou seja, são classes em que não criaremos nenhum objeto dela, diretamente. Para evitar confusão, é mais correto se referir a esse tipo de classes como superclasses abstratas. Usamos classes abstratas para representar grupos que tem características comum, mas que, em alguns detalhes específicos, agem de maneira diferente assim evitando a repetição de código.

Classes internas como o próprio nome diz, são classes que são definidas dentro de outra classe. Sendo que elas tem um relacionamento especial com sua classe externa (classe onde ela está definida), em relação as outras classes. Pelo fato de que elas podem acessar os membros privados da classe externa.

Interface é um recurso da orientação a objeto utilizado em Java que define ações que devem ser obrigatoriamente executadas, mas que cada classe pode executar de forma diferente.

Uma interface é criada da mesma forma que uma classe, mas utilizando a palavra-chave interface no lugar de class. interface nomeDaInterface { métodoAbstrato (argumentos); }

# Classes Abstratas



## Java e Orientação a Objetos

Classes abstratas são declaradas com o modificador `abstract` antes de class.

Classes abstratas tem uma função importante na orientação a objeto em Java, de forma objetiva, uma classe abstrata serve apenas como modelo para uma classe concreta.

Classes abstratas são modelos de classes, então, não podem ser instanciadas diretamente com o `new`, elas sempre devem ser herdadas por classes concretas. São apenas modelos, abstrações que agrupam e definem um conjunto de objetos com características semelhantes.

Classes abstratas são superclasses que servem como modelo. Esses modelos possuem, no entanto, somente as características gerais.

Outro fato importante de classes abstratas é que elas podem conter ou não métodos abstratos, que tem a mesma definição da assinatura de método encontrada em interfaces, ou seja, uma classe abstrata pode implementar ou não um método.

# Métodos Abstratos



São métodos criados nas classes **abstratas** sem implementação.

```
<modificador>* abstract <tipoRetorno><nomeMetodo>(<argumento>*);
```



Toda classe que contém um método *abstract* deve ser declarada *abstract*.

Classe abstrata não precisa ter método abstrato



## Java e Orientação a Objetos

Para criar métodos em classes devemos, necessariamente, saber qual o seu comportamento. Entretanto, em muitos casos não sabemos como estes métodos se comportarão na classe que estamos criando, e, por mera questão de padronização, desejamos que as classes que herdem desta classe possuam, obrigatoriamente, estes métodos.

Os métodos nas classes abstratas que não têm implementação são chamados de métodos abstratos. Para criar um método abstrato, apenas escreva a assinatura do método sem o corpo e use a palavra-chave *abstract*.

Métodos abstratos só podem ser declarados em classes abstratas.

# Exemplos de implementação

```
public abstract class Animal {  
    private String ruido; // atributo da classe abstrata  
  
    public Animal( String ruido ) { // construtor  
        this.ruido = ruido;  
    }  
    public abstract void fazerRuido(); // métodos abstratos  
    public abstract void comer();  
  
    //get e set  
    public String getRuido() { return ruido; }  
    public void setRuido(String ruido) { this.ruido = ruido; }  
}  
  
  
public abstract class Felino extends Animal {  
    public Felino( String ruido ) {  
        super(ruido);  
    }  
  
    public void andar(){ System.out.println("Anda com 4 patas"); }  
}
```

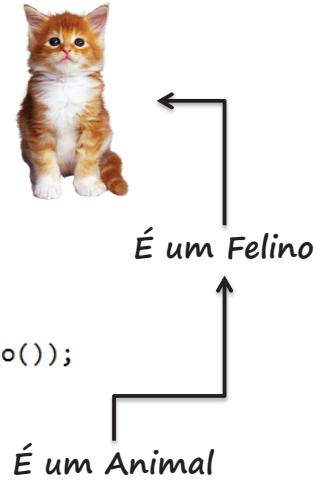


## Java e Orientação a Objetos

No exemplo acima foi criado a classe abstrata Animal com dois métodos abstratos, pois essa classe “não sabe a implementação” que suas subclasses irão desenvolver, porem serão obrigadas as implementar esse método. A classe Felino é outra classe abstrata que herda todas as características e ações que a classe Animal tem, adicionando o método andar();. Lembrando que não é possível instanciar/criar objetos a partir de classes abstratas.

# Exemplos de implementação

```
public class Gato extends Felino {  
  
    public Gato() {  
        super("Miauuuu, miauuu");  
    }  
  
    @Override  
    public void fazerRuido() {  
        System.out.println("Miar= " + this.getRuido());  
    }  
  
    @Override  
    public void comer() {  
        System.out.println("Come rato");  
    }  
  
}
```



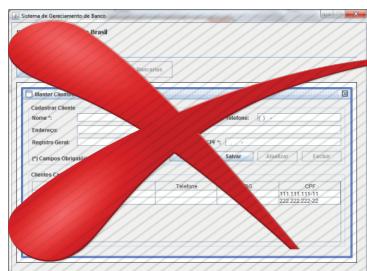
## Java e Orientação a Objetos

No exemplo acima foi criado a subclasse Gato que estende da classe Felino herdando todos os aspectos de Felino e Animal, assim tendo que implementar os métodos abstratos presentes na superclasse abstrata Animal, note o uso da anotação @Override para declarar uma sobreposição de métodos.

# Interfaces



*Não é o que você  
está pensando!*

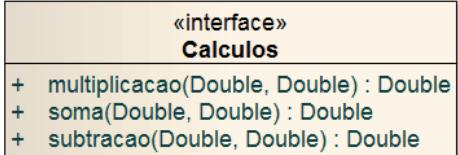


*É um tipo especial de classe contendo  
métodos abstratos e atributos finais*

*Interfaces por natureza são abstratas*

*Define um meio público e padrão de  
especificar o comportamento das classes*

## Notação UML



## Java e Orientação a Objetos

Para superar a limitação de herança múltipla que o Java não permite, este faz uso de interfaces, o qual pode ser visto como uma “promessa” que certos métodos com características previamente estabelecidas serão implementados, usando inclusive a palavra reservada `implements` para garantir esta implementação. As interfaces possuem sintaxe similar as classes, no entanto apresentam apenas a especificação das funcionalidades que uma classe deve conter, sem determinar como essa funcionalidade deve ser implementada.

Uma **interface** é um **tipo de classe abstrata** que só pode conter **métodos abstratos** ou **atributos** marcados com o **modificador final**. Interfaces, por natureza, são abstratas, não necessitando do modificador **abstract**.

Utilizamos interfaces quando queremos uma **relação entre classes não relacionadas** (semelhante a uma herança) que implementem métodos similares. Através de **interfaces**, podemos **compartilhar esse comportamento** entre as classes, mas **sem forçar um relacionamento** entre elas.

Existem outros motivos para você querer utilizar interfaces em seus programas:

Usar referências de objetos sem saber qual a classe que a implementa. Comoveremos mais adiante, podemos utilizar uma interface como tipo de dados.

Utilizar interfaces como mecanismo **alternativo para herança múltipla**, que permite às classes estender mais de uma superclasse.

# Criando Interfaces

```
[public] [abstract] interface <NomeDaInterface> {
    [public] [final] <tipoAtributo> <atributo> = <valorInicial>;
    [public] [abstract] <retorno> <nomeMetodo>(<parametro>*);
}
```



```
public interface Calculos {
    public Double soma(Number x, Number Y);
    public Double subtracao(Number x, Number Y);
    public Double multiplicacao(Number x, Number Y);
}
```

Criar



## Java e Orientação a Objetos

Para criarmos uma interface, utilizamos a sintaxe acima. Para **implementar** esta **interface** você deve usar a **palavra chave implements** na definição da classe.

Quando a classe implementa uma interface ela aceita o contrato da interface, então ela tem que implementar todos os métodos definidos na interface, caso contrário o compilador Java emitirá um erro de compilação.

Há alguns pontos importantes a considerar neste exemplo:

- > Uma interface declara-se usando a palavra reservada **interface** no lugar de **class**.
- > Os métodos são declarados e não implementados, ou seja, terminam por ponto e vírgula.
- > O nome dos métodos na classe que implementa a interface deve ser igual ao nome usado na interface, bem como a sua assinatura( valor de retorno e número de argumentos).
- > O nome dos argumentos não precisam ser iguais aos que serão usados na classe.

# Implementando Interfaces

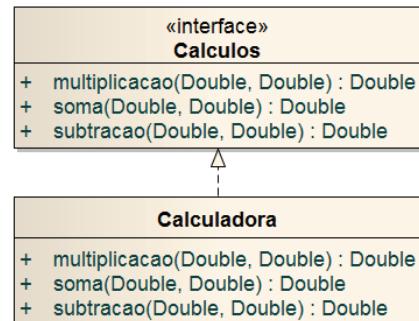
```
public class Calculadora implements Calculos {
    @Override
    public Double soma(Double x, Double y) {
        return x + y;
    }

    @Override
    public Double subtracao(Double x, Double y) {
        return x - y;
    }

    @Override
    public Double multiplicacao(Double x, Double y) {
        return x * y;
    }
}
```

Palavra reservada **implements** e usada para implementar uma interface

## Notação UML



## Java e Orientação a Objetos

Como vimos anteriormente, uma classe pode estender suas funcionalidades obtendo as características de outra classe num processo que chamamos de herança. Uma interface não é herdada, mas sim, implementada. Porque todo o código dos métodos da interface deve ser escrito dentro da classe que o chama. Dessa forma, obtemos as assinaturas dos métodos da interface em uma classe usando a palavra-chave **implements**.

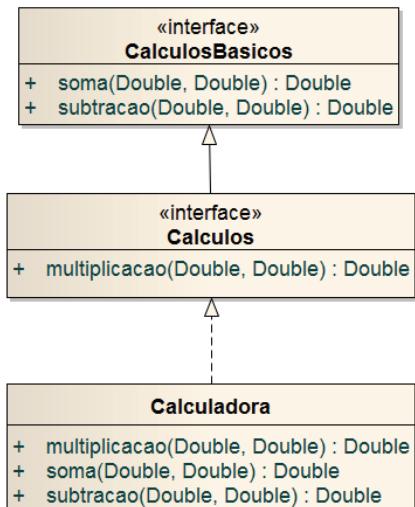
A vantagem principal das interfaces é que não há limites de quantas interfaces uma classe pode implementar, o que ajuda no caso de heranças múltiplas que não é possível ser feito em Java, pois uma classe apenas pode herdar as características de uma outra classe, ou seja, a classe pode apenas **herdar** uma **única superclasse**, mas pode também **implementar diversas interfaces**.

Uma interface não faz parte de uma hierarquia de classes. Classes não relacionadas podem implementar a mesma interface.

# Herança entre interfaces

Interfaces não são partes da hierarquia de classe. Entretanto, interfaces podem ter relacionamentos de herança entre elas próprias

## Notação UML



```

public interface CalculosBasicos {
    public Double soma(Double x, Double y);
    public Double subtracao(Double x, Double y);
}

public interface Calculos extends CalculosBasicos{
    public Double multiplicacao(Double x, Double y);
}
  
```



## Java e Orientação a Objetos

Interfaces não são partes de uma hierarquia de classes. Entretanto, **interfaces podem ter relacionamentos entre si**. Por exemplo, suponha que tenhamos duas interfaces, IEstudante e IPessoa. Se IEstudante estende IPessoa, esta **herda todos os métodos declarados** em IPessoa.

Além da própria definição e uso de interfaces, Java também permite realizar herança entre interfaces.

O conceito de herdar uma interface pode parecer estranho à primeira vista, já que as interfaces não possuem implementação.

Entretanto, justamente por isso, é um artifício de simples compreensão.

Uma interface, ao herdar de outra, automaticamente assume os métodos desta de forma implícita.

# Interface vs. Classe

Interfaces e classes são tipos

Uma interface pode ser usada em lugares onde pode se usar uma classe

```
Calculos calcula = new Calculadora();  
  
Calculadora calculadora = new Calculadora();  
  
//Calculos calculos = new Calculos(); // Erro
```

*Não é permitido criar instância de uma interface*



## Java e Orientação a Objetos

Uma **característica** comum entre **uma interface e uma classe** é que ambas **são tipos**. Isto significa que uma interface pode ser usada no lugar onde uma classe é esperada. Entretanto, **não se pode criar uma instância de uma interface**, assim como não podemos instanciar classes abstratas.

Outra característica comum é que ambas, interfaces e classes, podem definir métodos, embora uma interface não possa tê-los implementados.

### Interface vs. Classe Abstrata.

A principal diferença entre uma interface e uma classe abstrata é que a classe abstrata pode possuir métodos implementados (concretos) ou não implementados (abstratos). Na interface, todos os métodos são definidos com **abstratos** e **públicos**, sendo que a palavra-chave **abstract** e **public** sendo opcional na declaração.

Se uma classe só tem métodos abstratos, é melhor declará-la como interface.

# Classes Internas (Aninhadas)



Classe interna (**nested class**) é um recurso que permite definir uma classe dentro de outra

```
public class ClasseExterna {

    public class ClasseInterna {
        public String toString() {
            return "Classe Interna";
        }
    } // ClasseInterna

    public String toString() {
        ClasseInterna ci = new ClasseInterna();
        return "Classe Externa com " + ci;
    } // toString

    public static void main(String[] args) {
        ClasseExterna ce = new ClasseExterna();
        System.out.println(ce);
    } // main

} // ClasseExterna
```



## Java e Orientação a Objetos

Classe aninhada (nestedclass) é um recurso que permite definir uma classe dentro de outra e que surgiu a partir da versão 1.1 do Java. Assim como métodos e propriedades, uma classe aninhada é considerada um membro da classe.

Uma classe aninhada é utilizada para reforçar sua dependência com a sua classe externa, ou seja, ela depende dos outros membros da classe externa para funcionar.

Alguns consideram que esse novo recurso afetou a legibilidade do código, tornando-o mais complexo, inclusive aninhando uma à outra de forma recursiva.

Apesar do recurso de classes aninhadas dificultarem a legibilidade (se for mal aplicado), a maioria reconhece que classes internas é um recurso que permite organizar melhor o seu conjunto de classes. Especialmente as classes internas são muito úteis para:

Tratamento de eventos gráficos: clique de mouse, pressionamento de botão, etc. Esse recurso é bastante utilizado quando se estuda a parte de tratamento de eventos gráficos AWT.

Classe aninhadas de teste: em vez de criar uma classe externa a uma classe, para testar as suas funcionalidades, é possível e recomendável criar uma classe aninhada que poderá ter vários métodos de teste. Iremos ver sobre isso mais à frente.

Veja o exemplo acima, ele define a classe **Classe Externa** e uma **Classe Interna**, note que é possível ter classes dentro de outras, onde a classe **Classe Externa**, definiu uma classe internamente denominada **Classe Interna**. Ambas sobrepõe o método **toString()**, que é executado automaticamente toda vez que é invocado (chamado) o método **System.out.println()**.

Note que quando se compila **Classe Externa.java**, surgem dois arquivos **.class**: **Classe Externa.class** - este é o resultado da compilação de **Classe Externa**. **Classe Externa\$Classe Interna.class** - este é o resultado da compilação de **Classe Interna**, porém como ela é interna, a sua classe externa também é listada, separada pelo cifrão (\$).

## Classes internas anônimas

**Classes internas anônimas (anonymous inner classes)** em geral aumentam bastante à complexidade de entendimento de seu código, portanto deve ser utilizado de forma comedida ou em situações em que seu uso já seja esperado.

A linguagem Java permite que você declare classes praticamente em qualquer lugar, inclusive no meio de um método, se necessário, e mesmo sem atribuir um nome para ela. Basicamente, isso é um truque do compilador, mas há momentos em que ter classes internas anônimas é extremamente prático.

# Tipos Enumerados

O tipo **enum** estende implicitamente a classe **java.lang.Enum**. As enumerações podem ter construtores, métodos, variáveis.

```
public enum EnumEstacoes {
    PRIMAVERA(EnumMes.SETEMBRO, EnumMes.NOVEMBRO),
    VERAO(EnumMes.DEZEMBRO, EnumMes.FEVEREIRO),
    OUTONO(EnumMes.MARCO, EnumMes.MAIO),
    INVERNO(EnumMes.JUNHO, EnumMes.AGOSTO);

    private EnumMes inicio, fim;

    private EnumEstacoes( EnumMes inicio, EnumMes fim ) {
        this.inicio = inicio;
        this.fim = fim;
    }

    // somente métodos get são necessários
    public EnumMes getInicio() { return inicio; }
    public EnumMes getFim() { return fim; }

    enum EnumMes {
        JANEIRO, FEVEREIRO, MARCO, ABRIL, MAIO, JUNHO, JULHO,
        AGOSTO, SETEMBRO, OUTUBRO, NOVEMBRO, DEZEMBRO;
    }
}
```

➤ Como as classes, as enumerações podem ser declaradas dentro de classes, como estáticas e locais.



## Java e Orientação a Objetos

Quando você precisar criar uma constante, você utilizará definição de variáveis **estática** e **final**. Toda vez que usamos a palavra-chave **final** estamos informando que algo **não pode mudar**. Com classes, final significa que não podemos mais fazer herança, com métodos não podemos mais sobrepor-lo e com variáveis significa que após a inicialização o valor da variável não pode mais ser alterado, exemplo:

```
static final int JANEIRO=1;
```

Algumas vezes necessitamos de conjuntos de valores constantes - por exemplo, conjunto de estação do ano. Uma abordagem padrão para definição de um conjunto com esta natureza seria definir uma classe ou interface contendo as constantes como abaixo:

```
public class Estacoes {
    public static final int PRIMAVERA = 1;
    public static final int VERAO = 2;
    public static final int OUTONO = 3;
    public static final int INVERNO = 4;
}
```

A abordagem utilizada apresenta algumas desvantagens:

- > Não é segura quanto ao tipo – qualquer inteiro pode ser passado ao método setEstacao().
- > Devemos utilizar o nome da classe (ou interface) para referenciar a constante.
- > O valor não representa a informação textual do nome da constante.
- > As classes que utilizam estas constantes devem ser recompiladas caso seus valores sofram mudança.

**Java** implementa este conceito de conjunto de constantes com o novo tipo **enum**. O tipo **enum** representa os tipos enumerados ou conjuntos comumente vistos em outras linguagens de programação, ele define um conjunto de nomes e valores constantes. Veja o exemplo acima usando **enum**:

O tipo **enum** estende implicitamente a classe **java.lang.Enum**. Nossas enumerações podem ter construtores, métodos, variáveis. Como as classes, as enumerações podem ser declaradas dentro de classes, como enumerações estáticas e locais.

O tipo enumerado acrescenta dois métodos implicitamente:

## Métodos

<code>static&lt;tipo da sua enum&gt;[] values()</code>	Retorna um array contendo as constantes desta enum, na ordem em que são declaradas.
<code>static&lt;tipo da sua enum&gt; valueOf(String nome)</code>	Retorna uma constante da enum com o argumento nome passado.

O comando **switch**, também pode ser usado com enumerados e não somente tipos inteiros, o rótulo `case` pode ser acompanhado da constante.

# Java e Orientação a Objetos

## Exceções

Foi detectado um problema e o windows foi desligado para evitar danos ao computador

PAGE\_FAULT\_IN\_NONPAGED\_AREA

Se esta for a primeira vez que você vê esta tela de erro de parada, reinicie o computador, se a tela foi exibida novamente, siga estas etapas:

Certifique-se de que existe espaço suficiente em disco. Se um driver for identificado na mensagem de parada, desative o drivers ou solicite atualizações do driver ao fabricante, experimente trocar os adaptadores de vídeo

Consulte o fornecedor do hardware para obter atualizações de BIOS. Desative opções de memória BIOS, como cache ou sombreamento. Se precisar usar o modo de segurança para remover ou desativar componentes, reinicie o computador, pressione F8 para selecionar as opções avançadas de inicialização selecione o modo de Segurança.

Informações técnicas:

```
*** STOP: 0x0000008E (0xC0000005, 0xBFABFF1B, 0xB8F61B14, 0x00000000)
*** nv4_disp.dll - Address BHABBF1B base at BF9D4000, Datestamp 4410c8d4
```

Iniciando despejo de memória física.  
Despejo de memória física concluída.  
Entre em contato com o administrador.  
do sistema ou grupo de suporte técnico para obter a informação.



## Java e Orientação a Objetos

Quando se desenvolve software em Java, há possibilidade de ocorrer erros imprevisíveis durante sua execução, esses erros são conhecidos como exceções em **Java**, e podem ser provenientes de erros de lógica ou acesso a dispositivos ou arquivos externos.

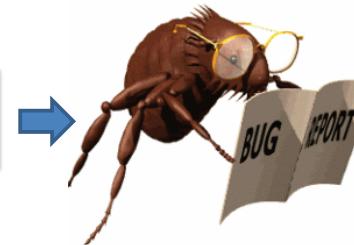
Uma exceção representa uma situação que normalmente não ocorre e representa algo de estranho ou inesperado no sistema ou seja, ocorrem quando algo imprevisto acontece, elas podem ser provenientes de erros de lógica ou acesso a recursos que talvez não estejam disponíveis.

Alguns possíveis motivos externos para ocorrer uma exceção são:

- > Tentar abrir um arquivo que não existe.
- > Tentar fazer consulta a um banco de dados que não está disponível.
- > Tentar escrever algo em um arquivo sobre o qual não se tem permissão de escrita.
- > Tentar conectar em servidor inexistente.

# Exceções

Erros que ocorrem **durante a execução** do programa



É um **evento** que **interrompe** o fluxo normal de **processamento** de uma classe



## Java e Orientação a Objetos

Uma exceção é um **evento que interrompe o fluxo** normal de processamento de uma classe. Este evento é um erro de algum tipo. Isto causa o término inesperado da execução de seu programa.

Estes são alguns dos exemplos de exceções que podem ter ocorridos em exercícios anteriores:

> **Array Index OutOfBounds Exception:** ocorre ao acessar um elemento inexistente de um **array**.

> **Number Format Exception:** ocorre ao enviar um parâmetro não numérico para o método **Integer.parseInt()**.

Podemos dizer que as exceções são instâncias de classes. E como qualquer classe ou objeto, podemos facilmente manipular.

Existem métodos comuns entre todas as classes de Exceções, dentre as quais podemos citar:

> **toString()**: Converte os dados da exceção para String para visualização.

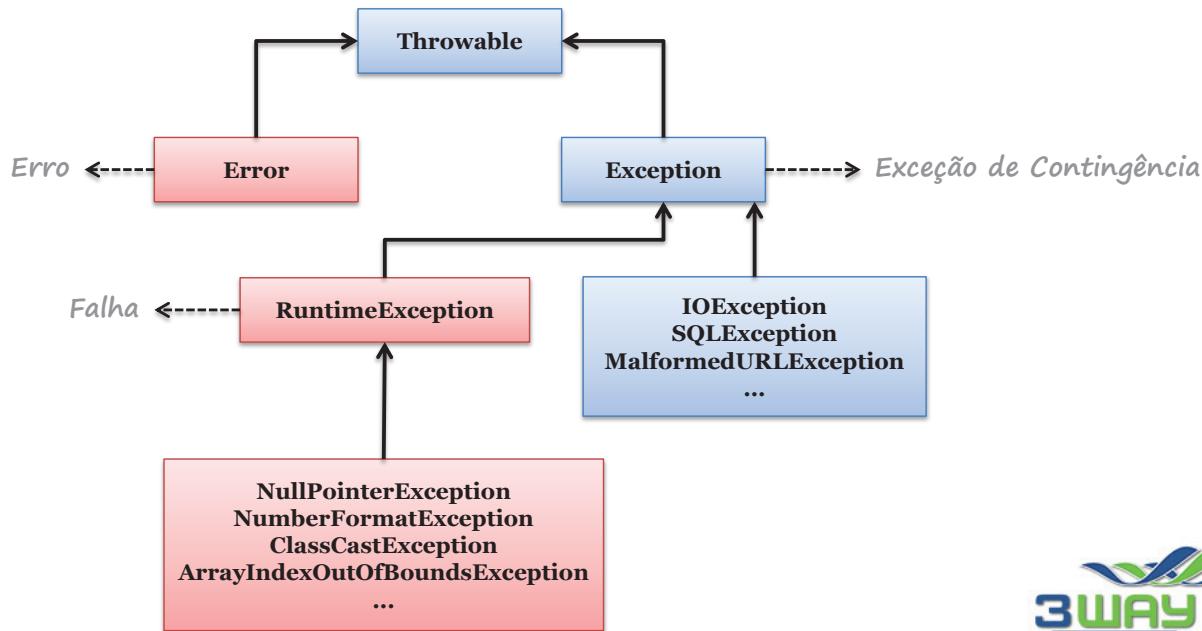
> **printStackTrace()**: Imprime na saída de erro padrão (geralmente console) todos os frames de onde foram detectados erros. Útil para depuração no desenvolvimento, pois mostra todo o histórico do erro, além das linhas onde foram ocasionados.

> **getCause()**: Retorna a causa da Exceção, ou null se a causa for desconhecida ou não existir.

> **getMessage()**: Retorna uma String com o erro. É uma forma simples e elegante de mostrar a exceção causada, geralmente, utilizada como forma de apresentação ao usuário.

# Categoria de Exceções

Todas as exceções são **subclasses**, direta ou indiretamente, da classe **java.lang.Throwable**



## Java e Orientação a Objetos

Objetos que podem ser lançados para indicar que algo anormal aconteceu, e capturados para lidar com esta situação, devem estender a classe Throwable do pacote java.lang. Os métodos mais úteis desta classe são:

- > **public String getMessage()**// retorna uma mensagem de erro
- > **public void printStackTrace()**// imprime uma descrição da pilha no instante em que o problema ocorreu

Como já mencionado, a superclasse de todas as exceções é a classe Exception do pacote java.lang. Conforme indicado acima, esta classe é uma subclass da classe Throwable. Existe um grande número de subclasses de Exception espalhadas pelos vários pacotes de biblioteca. O programador pode estender a classe Exception ou uma das suas subclasses para construir as suas próprias exceções.

Nem todos os problemas que podem ocorrer são considerados exceções. Problemas mais sérios, que em geral não podem ser tratados pelo programador, são chamados erros e são objetos da classe Error do pacote **java.lang** (ou das subclasses desta classe, que também estão espalhadas pelos vários pacotes).

## A hierarquia de exceções

A classe Throwable tem duas subclasses:

**java.lang.Exception**

É a raiz das classes derivadas de Throwable que indica situações que a aplicação pode querer capturar e realizar um tratamento que permita prosseguir com o processamento.

## java.lang.Error

É a raiz das classes derivadas de `Throwable` que indica situações que a aplicação não deve tentar tratar. Usualmente indica situações anormais, que não deveriam ocorrer.

**Exemplos de exceções já definidas em Java incluem:**

### java.lang.ArithmeticsException

Indica situações de erros em processamento aritmético, tal como uma divisão inteira por 0.

### java.lang.NumberFormatException

Indica que se tentou a conversão de uma `String` para um formato numérico, mas seu conteúdo não representava adequadamente um número para aquele formato. É uma subclasse de `java.lang.IllegalArgumentException`.

### java.lang.ArrayIndexOutOfBoundsException

Indica a tentativa de acesso a um elemento de um arranjo fora de seus limites, ou o índice era negativo ou era maior ou igual ao tamanho do arranjo. É uma subclasse de `java.lang.IndexOutOfBoundsException`, assim como a classe `java.lang.StringIndexOutOfBoundsException`.

### java.lang.NullPointerException

Indica que a aplicação tentou usar `null` onde uma referência a um objeto era necessária, invocando um método ou acessando um atributo, por exemplo.

### java.lang.ClassNotFoundException

Indica que a aplicação tentou carregar uma classe, mas não foi possível encontrá-la.

### java.io.IOException

Indica a ocorrência de algum tipo de erro em operações de entrada e saída. É a raiz das classes `java.io.EOFException` (fim de arquivo), `java.io.FileNotFoundException` (arquivo especificado não foi encontrado) e `java.io.InterruptedIOException` (operação de entrada ou saída foi interrompida), entre outras.

Entre os erros definidos em Java, subclasses de `java.lang.Error`, estão `java.lang.StackOverflowError` e `java.lang.OutOfMemoryError`. São situações onde não é possível uma correção a partir de um tratamento realizado pelo próprio programa que está executando.

# Manipulando Exceções

Utilizando a declaração **try-catch-finally**

```
public static void main(String[] args) {
    int array[] = { 1, 2, 3 };

    try {
        System.out.println(array[4]);
    } catch (ArrayIndexOutOfBoundsException exception) {
        System.out.println("Array acessado não existe!");
    } finally {
        System.out.println("Obrigado por tratar esse erro!");
    }
}
```

O bloco **catch** recebe um argumento do tipo de exceção que será tratado.

Tratamento da exceção

Sempre será executado



Para cada bloco **try**, pode haver um ou mais blocos **catch**, mas somente um bloco **finally**



## Java e Orientação a Objetos

Umas das utilidades proporcionadas pela orientação a objetos de Java é a facilidade em tratar possíveis erros de execução chamados de exceções. Sempre que um método de alguma classe é passível de causar algum erro, então, podemos usar o método de tentativa - o try.

Tudo que estiver dentro do bloco try será executado até que alguma exceção seja lançada, ou seja, até que algo dê errado. Quando uma exceção é lançada, ela sempre deve ser capturada. O trabalho de captura da exceção é executado pelo bloco catch.

Um bloco try pode possuir vários blocos de catch, dependendo do número de exceções que podem ser lançadas por uma classe ou método. O bloco catch obtém o erro criando uma instância da exceção. A sintaxe do bloco **try catch** pode ser vista no exemplo acima.

### Finally

Finally é o trecho de código final. A função básica de finally é sempre executar seu bloco de dados mesmo que uma exceção seja lançada.

É muito útil para liberar recursos do sistema quando utilizamos, por exemplo, conexões de banco de dados e abertura de buffer para leitura ou escrita de arquivos.

Finally virá após os blocos de catch.

# Throw e Throws



Se um método causar uma exceção mas não capturá-la, então deve-se utilizar a palavra-chave **throws**

```
public class Calculadora {
    public static void main(String[] args) {
        Double nota1 = 5.0;
        Double nota2 = 3.0;

        try {
            System.out.println(Calculadora.calculaMedia(nota1, nota2));
        } catch (Exception e) {
            System.out.print("Tratamento de erro: ");
            System.out.println(e.getMessage());
        }
    }

    public static Double calculaMedia(Double x, Double y) throws Exception {
        Double media = (x + y) / 2;
        if (media < 6) {
            throw new Exception("Criando exceção com throws");
        }
        return media;
    }
}
```



## Java e Orientação a Objetos

As cláusulas `throw` e `throws` podem ser entendidas como ações que propagam exceções, ou seja, em alguns momentos existem exceções que não podem ser tratadas no mesmo método que gerou a exceção. Nesses casos, é necessário propagar a exceção para um nível acima na pilha.

### A palavra-chave `throw`

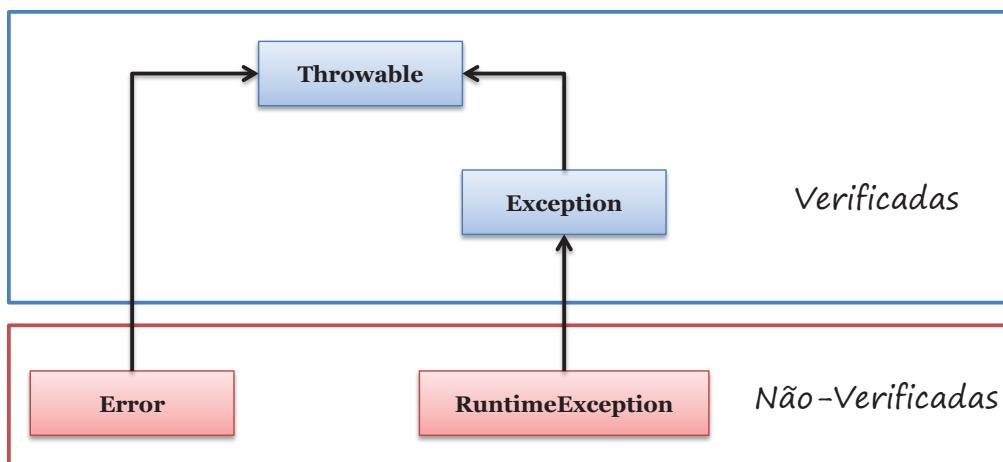
Além de capturar exceções, Java também permite que os métodos lancem exceções. A sintaxe para o lançamento de exceções é:

`throw<objetoExceção>;`

### Desviando Exceções com `Throws`

No caso de um método causar uma exceção, mas não capturá-la, deve-se utilizar a palavra-chave `throws` para repassar esta exceção para quem o método que o chamou. Esta regra se aplica apenas para exceções verificadas. Veremos mais sobre exceções verificadas e não verificadas mais a frente.

# Exceções Verificadas e Não-Verificadas



## Java e Orientação a Objetos

Qualquer exceção ocorrida em uma linha dentro do `try` transferirá o controle para o bloco `catch`. Como só existe um `printStackTrace()` solitário dentro do `catch`, o código prosseguirá naturalmente nas linhas seguintes, como se nada tivesse acontecido. Mal tivemos a chance de saber qual era o tipo do erro que ocorreu. Poderia ser, por exemplo, um mero erro de programação ou algo mais grave como uma chamada inválida a uma API ou uma falha no servidor de banco de dados.

O programador Java precisa saber com qual categoria de erro ele está lidando: As exceções podem ser divididas em três categorias:

> **Exceções geradas por erros de programas:** Em geral são *bugs* de sistema, como por exemplo, acessar um objeto nulo, o que ocasionará uma **`NullPointerException`** ou tentar acessar uma posição inválida de um *array*, que devolvem outra exceção famosa: **`ArrayIndexOutOfBoundsException`**. Em geral, não há nada o que fazer quando esses erros acontecerem, senão, deixar que o erro simplesmente ocorra.

> **Exceções geradas por violações de contratos entre APIs:** O programa tenta fazer algo não permitido pela API, a qual gera um erro específico para a ocasião: Um exemplo é tentar processar um arquivo XML corrompido. Nestes casos, é possível contornar o problema informando ao usuário sobre o problema. Outro é tentar passar uma URL inválida para o construtor da classe URL. O endereço `httpk://www.pucpr.br` vai gerar uma **`MalformedURLException`**. Claro, não existe o protocolo `httpk`.

> **Exceções geradas por falhas em recursos externos:** São geradas quando algum recurso externo falhar. Um exemplo é tentar conectar um servidor de FTP e o mesmo estando fora do ar. Neste caso, o programa poderá contornar o problema tentando novamente segundos depois por n vezes ou pedindo outro servidor. Outra abordagem é deixar que o erro fosse tratado pela própria VM. A forma de tratar esses erros vai variar muito de acordo com o contexto em que o programa está inserido.

## Mas como o Java trata essas exceções?

Em Java temos dois tipos de exceção: **checked (verificadas)** e **unchecked (Não verificadas)**. Preferi manter os termos em inglês, uma vez que são muito usados em artigos e documentações.

Entende-se por **exceções verificadas**, aquelas exceções que devem ser tratadas pelo programa cliente usando **try..catch** ou delegadas através da clausula **throws**. Uma exceção **verificada** é a forma que o método chamado tem para avisá-lo: “Ei, você precisa fazer alguma coisa!”. Todas as classes que são filhas de **Exception** exigem tratamento, pois são consideradas **exceções verificadas**.

Já as **exceções não verificadas** não exigem nenhum tratamento por parte do programador (apesar de deixar tratá-las). A abordagem destes tipos de exceção é: “Deixe que o erro aborte todo o processo, não há nada o que fazer.”. Todas as classes que são filhas de **RuntimeException** não precisam ser tratadas, assim como aquelas que são filhas da classe **Error**.

# Criando Exceções

```

public class MediaInsuficienteException extends Exception {
    public MediaInsuficienteException() {
        super("Exception criada para média menor que 6.0");
    }
}

public static void main(String[] args) {
    Double nota1 = 5.0;
    Double nota2 = 3.0;
    try {
        System.out.println(Calculadora.calculaMedia(nota1, nota2));
    } catch (MediaInsuficienteException e) {
        System.out.print("Tratamento de erro: ");
        System.out.println(e.getMessage());
    }
}

public static Double calculaMedia(Double x, Double y) throws MediaInsuficienteException {
    Double media = (x + y) / 2;
    if (media < 6) {
        throw new MediaInsuficienteException();
    }
    return media;
}
  
```



*Atributos de objetos e construtores podem ser adicionados à classe*

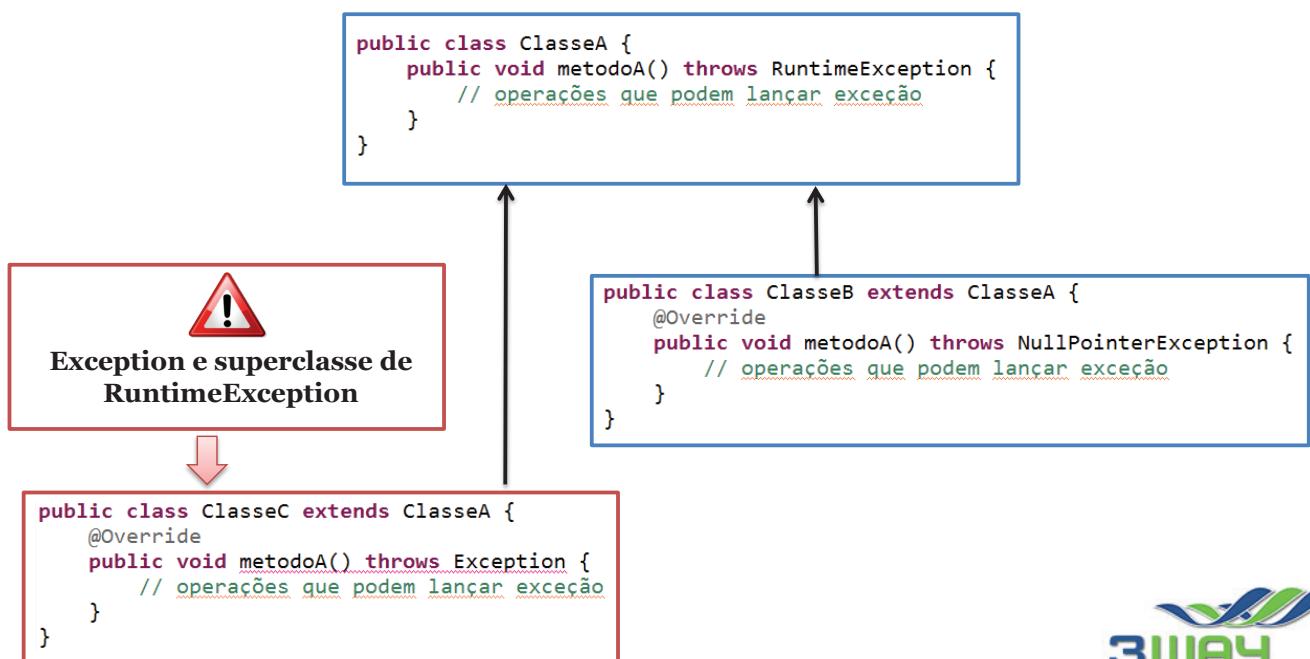


## Java e Orientação a Objetos

Apesar de muitas classes de exceção já existirem no pacote **java.lang**, essas classes de exceção não são suficientes para cobrir todas possibilidades de exceções que podem ocorrer na implementação de suas próprias classes. Daí a razão de querermos criar nossas próprias classes de exceções.

Para criar nossa própria exceção, teremos que criar uma classe que estenda a classe **RuntimeException** ou **Exception**, ou qualquer de suas subclasses. Deste modo, devemos personalizar a classe de acordo com o problema a ser resolvido. Atributos de objeto e construtores podem ser adicionados na sua classe de exceção. Veja o exemplo no slide acima.

# Sobrescrita de Métodos e Exceções



## Java e Orientação a Objetos

Quando sobreponemos um método que contenham a cláusula throws na sua definição, temos que tomar o cuidado de sobrescrevermos este método com um subconjunto apropriado das exceções já definidas no método da superclasse:

> As exceções declaradas devem ser da mesma classe ou subclasse da exceções marcadas na cláusula throws. Por exemplo, se o método da superclasse lança um IOException então o método sobreposto pode lançar um IOException ou FileNotFoundException (subclasse de IOException), mas não um Exception.

> Podemos usar um subconjunto das exceções declaradas na cláusula throws do método na superclasse

> Não podemos adicionar novos tipos de classes de exceções - além das que já estão definidas na cláusula throws no método da superclasse ao método sobreposto na subclasse.

# Java e Orientação a Objetos

## Tipos Genéricos



### Java Orientado a Objeto

Neste módulo veremos uma introdução à Generics ou programação genérica, na qual será muito importante quando formos tratar de classes e interfaces de coleção como, por exemplo, a classe ArrayList.

Generics, ou programação genérica, serve para determinar para o compilador, qual tipo de classe deve ser interpretada.

Para que essa classe possa aceitar qualquer tipo de classe, devemos fazer uso de generics. Generics é indicado como um identificador entre os sinais de maior e menor (<>).

# Java Orientado a Objetos

## Tipos Genéricos



É um perigo em potencial para uma <b>ClassCastException</b>	Permite que uma única classe trabalhe com uma grande <b><u>variedade de tipos</u></b>
Torna nossos códigos mais poluídos e menos legíveis	É uma forma natural de eliminar a necessidade de se fazer <b>cast</b>
Destroi benefícios de uma linguagem com <b>tipos fortemente definidos</b>	Preserva benefícios da checagem de tipos



## Java Orientado a Objeto

Uma grande inconveniência da linguagem de programação Java era a necessidade constante de fazer **typecast** ou **casting** (conversões de tipo explícitas) em expressões que necessitassem de referências de objetos mais específicas. Por exemplo, um objeto **ArrayList** permite que você adicione objetos de qualquer tipo à coleção, mas quando você quiser recuperar estes elementos, terá de fazer um **typecast** nos objetos para que o tipo de referência seja mais específico à sua necessidade, ou seja, uma referência para uma subclasse de **Object**. O **casting** é um perigo em potencial, uma vez que não é possível, em tempo de compilação, garantir qual o tipo do objeto poderá ser referenciado – podendo gerar uma exceção do tipo **ClassCastException**, em tempo de execução. Adiciona-se a isto o fato de seu código ficar mais poluído e, portanto, menos legível e destrói os benefícios de uma linguagem com os tipos fortemente definidos, já que esse procedimento anula a segurança trazida pela checagem de tipos durante a compilação.

O principal objetivo da adição de **Generics** ao Java é solucionar problemas como os apresentados no parágrafo anterior. A adição de tipos **Generics**, em Java 5.0, permite definir o uso de um **tipo específico** para uma classe, como **ArrayList**, que trabalha com uma grande variedade de tipos.

Considere um objeto **ArrayList** para ver como **Generics** ajuda a melhorar nosso código. Um objeto **ArrayList** possui a capacidade de armazenar referências de objetos, elementos de qualquer tipo de referência em uma coleção de objetos. Uma instância de **ArrayList**, entretanto, sempre força a realizar um **casting** nos objetos que recuperamos a partir da coleção.

Para ajudar o desenvolvedor na adoção destas práticas existe esse recurso para fazer a verificação, em tempo de compilação, de tipos manipulados por uma classe. Um dos motivos para a inclusão de Generics é a garantia de que pelo menos um tipo de objeto será relacionado a estas classes, deixando claro para o profissional com quais elementos ele está trabalhando em uma coleção. Por exemplo, podemos ter um objeto ArrayList gerenciando outros Double e outra coleção responsável por objetos Calendar. Ao programar desta maneira, evitamos o uso de uma única lista para ambos, tornando o código mais claro para o desenvolvedor.

Outro objetivo da inclusão de Generics é a redução de bugs em tempo de execução. Com ele, é possível a captura de erros na compilação, alertando o desenvolvedor sobre a ocorrência de exceções comuns, como uma ClassCastException. Consequentemente, fornece um meio para a escrita de um código mais seguro.

A reusabilidade também faz parte do recurso. Utilizando novamente uma classe da API de Collections como exemplo, podemos criar listas de String, outras de Integer e outras para qualquer classe. Desta forma, evitamos a escrita de código repetido e desnecessário.

# Declarando uma Classe utilizando Generics

Classe não trabalha com nenhuma referência a um tipo específico

```

public class ManipulaArray<T> { // Contém o parâmetro <?>
    private T[] array; // atributo de tipo genérico
    public ManipulaArray( T[] array ) { // Construtor
        this.array = array;
    }

    public boolean existeElemento(T elementoABuscar) {
        for (T elemento : array) {
            if (elemento.equals(elementoABuscar)) {
                return true;
            }
        }
        return false;
    }

    // get e set de atributo genérico
    public T[] getArray() { return array; }
    public void setArray(T[] array) {this.array = array;}
}

```

Indica que a classe declarada é uma classe Generics

Declarando Métodos Genéricos



## Java Orientado a Objeto

Você também pode criar suas próprias classes utilizando **Generics**. Vamos explicar a sintaxe de **Generics**, utilizada na declaração da classe **ManipulaArray**.

### 1) Declarando a classe:

**public class ManipulaArray<T>{...}**

O nome da classe é seguido por um par de sinais **< (menor que) e > (maior que)** envolvendo o literal **<T>**. Isto é chamado de parâmetro de tipo, qualquer identificador válido em Java pode ser usado - por convenção utiliza-se a letra maiúscula **T** de **Type** (ou Tipo). Os usos dos sinais **<...>** indicam que a classe declarada é uma não trabalha com um tipo específico. Por isso, observe que o atributo da classe foi declarado para ser do tipo **<T>**:

### 2) Declarando a atributo:

**private T[] array;**

Essa declaração especifica que a variável de instância **array** é de um tipo **Genérico**, e depende do tipo de dado que será usado na declaração da referência de objeto da classe **ManipulaArray<T>**. Quando declarar uma instância da classe, você deverá especificar o tipo da classe com qual você deseja trabalhar. Por exemplo:

**ManipulaArray<Integer>**

**arrayInteiros = new ManipulaArray<Integer>()**

A sintaxe <Integer> após a declaração de **ManipulaArray** especifica que essa instância da classe irá trabalhar com variáveis do tipo Integer. Podemos trabalhar com variáveis do tipo **Double** ou qualquer outro tipo de referência.

Observe a declaração para o método **get()**:

```
public T[] getArray(){
    return this.array;
}
```

O método **get()** retorna um array do tipo **T**, que é um tipo **Genérico**. Isso significa que o método terá um **tipo de dado em tempo de execução**, ou mesmo em tempo de compilação. Depois de declarar um objeto do tipo Integer, <T> será ligado a um tipo de dado específico. Essa instância atuará como se tivesse sido declarada para ter esse tipo de dado específico, e apenas esse, desde o início.

Note que a criação de uma instância de uma classe que utiliza **Generics** é bem similar à criação de uma classe normal, exceto pelo tipo de dado específico entre os sinais <> logo após o nome do construtor. Essa informação adicional indica o tipo de dado com que você. Depois de criada a instância, torna-se possível acessar os elementos da classe sem a necessidade do **typecast** no valor do retorno do método **get()**, uma vez que já foi decidido que ele irá trabalhar com um tipo de dado de referência específico.

## Declarando Métodos Genéricos

Além de declarar classe **Generics**, também podemos declarar métodos **Generics**. Parametrizar métodos é útil quando queremos realizar tarefas onde as dependências de tipo entre os argumentos e o valor de retorno são **Generics**, mas não depende de nenhuma informação do tipo da classe, e mudará a cada chamada ao método.

Se as operações realizadas por vários métodos sobrecarregados forem idênticas para cada tipo de argumento, os métodos sobrecarregados podem ser codificados mais compacta e convenientemente com um método genérico.

Você pode escrever uma única declaração de método genérico que pode ser chamada com argumentos de tipos diferentes.

Com base nos tipos dos argumentos passados para o método genérico, o compilador trata cada chamada de método apropriadamente.

```
public static <T> void printArray (T[] arrayEntrada){
    // exibe elementos do array
    for(T elemento : arrayEntrada){
        System.out.print(elemento + " - ");
    }
}
```

# Limitação “Primitiva”

Tipos **Generics** do Java são restritos a tipos de referência (**objetos**) e não funcionarão com tipos primitivos



Solução  
→

**Utilizar Classes Wrapper(Empacotadoras)**



## Java Orientado a Objeto

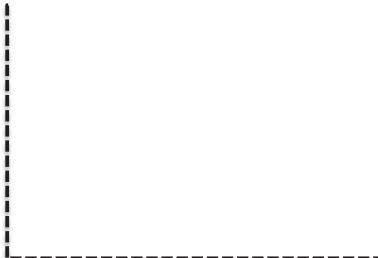
Uma limitação de **Generics** em Java, é que eles são de uso restrito a tipos referência de Objetos, e não funcionarão com tipos primitivos. Por exemplo, o comando a seguir seria ilegal, uma vez que **int** é um tipo de dado primitivo.

```
ArrayList <int> listaInt = new ArrayList <int>();
```

Você terá que empacotar os tipos primitivos com um a classe **Wrapper**, antes de usá-los como argumentos **Generics**.

# Limitando Genéricos

```
public class ColecaoBichoFelino<T extends Felino> {
    T[] animais;
}
```



## Java Orientado a Objeto

`ArrayList <Integer> listaInteger = new ArrayList <Integer>();`

No exemplo mostrado anteriormente, os parâmetros de tipo da classe **ManipulaArray** podem ser referencias para qualquer tipo de classe. Há casos, entretanto, onde você quer restringir os tipos que poderão ser usados em instanciações de uma classe do tipo **Generics**. Java permite limitar o conjunto de possíveis tipos que podem ser utilizados.

Por exemplo, a classe **ColecaoBichoFelino**, que utiliza **Generics**, serviria como um contêiner para um felino qualquer. Em tempo de execução, T será uma instância de uma subclasse de **Felino**. É utilizado **extends** para fazer a limitação de classes que possam fazer uso da classe genérica criada.

```
public class TesteLimitacaoGenerics{
    public static void main(String args[]) {
        ColecaoBichoFelino <Gato> obj1 = new ColecaoBichoFelino <Gato>();
        //O comando seguinte é ilegal
        ColecaoBichoFelino <Galinha> obj2 = new ColecaoBichoFelino <Galinha>();
    }
}
```

A instanciação de **obj1** é válida, uma vez que Gato é uma subclasse de Felino. Porém a criação de **obj2** causará um erro, uma vez que Galinha não é uma subclasse de **Felino**.

Usar **Generics** limitados nos dá um benefício adicional, que é a checagem estática de tipos. Como resultado, nós temos a garantia de que toda instanciação de um tipo **Generics** respeita os limites (ou restrições) que atribuímos a ele. Uma vez que asseguramos que toda instância do tipo é uma subclasse do limite atribuído, e então podemos chamar, de forma segura, qualquer método do objeto.

# Coringa <?>

&lt;? extends T&gt;

Aceita T e todos os seus descendentes

&lt;? super T&gt;

Aceita T e todos os seus ascendentes

```

public class ColecaoBichoFelino {

    public void addAnimal(List<? extends Felino> animais) {
        //animais.add(new Leao()); //não pode adicionar quando é utilizado
        //<? extends Felino>
        for (Felino bicho : animais) {
            bicho.fazerRuido();
        }
    }

    public static void main(String[] args) {
        List<Leao> animais = new ArrayList<Leao>();
        animais.add(new Leao());
        ColecaoBichoFelino colecao = new ColecaoBichoFelino();
        colecao.addAnimal(animais);
    }
}

```

Aceita somente Felino  
Neste caso [Leão]



## Java Orientado a Objeto

Existem 3 tipos de Wildcards(Coringas) em Generics:

**List<?>**, mas conhecido como Unknown Wildcard, ou seja, Wildcard desconhecido.

**List<? extends A>**

**List<? super A>**

O uso de Generics faz-se necessário para evitar casts excessivos e erros que podem ser encontrados em tempo de compilação, antes mesmo de ir para a produção. Todo profissional da área deve ter o conhecimento de como utilizar este recurso tão poderoso, pois em muito se aumenta a produtividade utilizando-o.

## Unknown Wildcard

Como você não sabe o tipo do objeto, você deve tratá-lo da forma mais genérica possível.

## Extends Wildcard

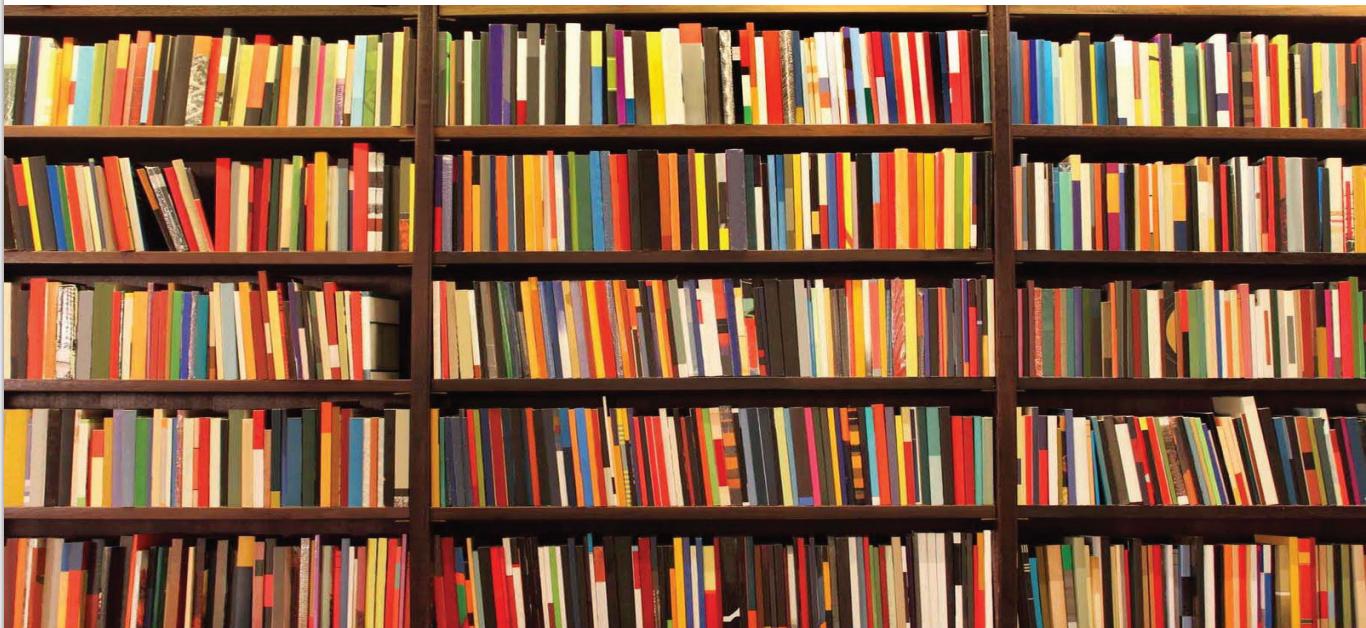
Podemos utilizar este tipo de Wildcard para possibilitar o uso de vários tipos que se relacionam entre si, ou seja, podemos dizer que o nosso método **addAnimal** aceita uma lista de animal que estende a classe **Felino**.

## Super wildcard

Ao contrário do **extends**, o wildcard super, permite somente que elementos Object sejam utilizados, isso significa que somente a classe Object que é a única classe pai de **Felino** poderá utilizar o método.

# Java e Orientação a Objetos

## Collections



## Java e Orientação a Objetos

Como vimos no modulo de arrays, manipular array é bastante trabalhoso. Essa dificuldade aparece em diversos momentos:

- > Não podemos redimensionar um array em Java;
- > É impossível buscar diretamente por um determinado elemento cujo índice não se sabe;
- > Não conseguimos saber quantas posições do array já foram populadas.

A API do Collections é robusta e possui diversas classes que representam estruturas de dados avançadas que substitui na maioria dos casos a utilização de array(matrizes).

# Java Collections



 É um objeto onde podemos agrupar vários elementos (outros objetos)

Métodos implementados que realizam operações (sort, reverse, isEmpty, size ...) sobre as coleções



## Java e Orientação a Objetos

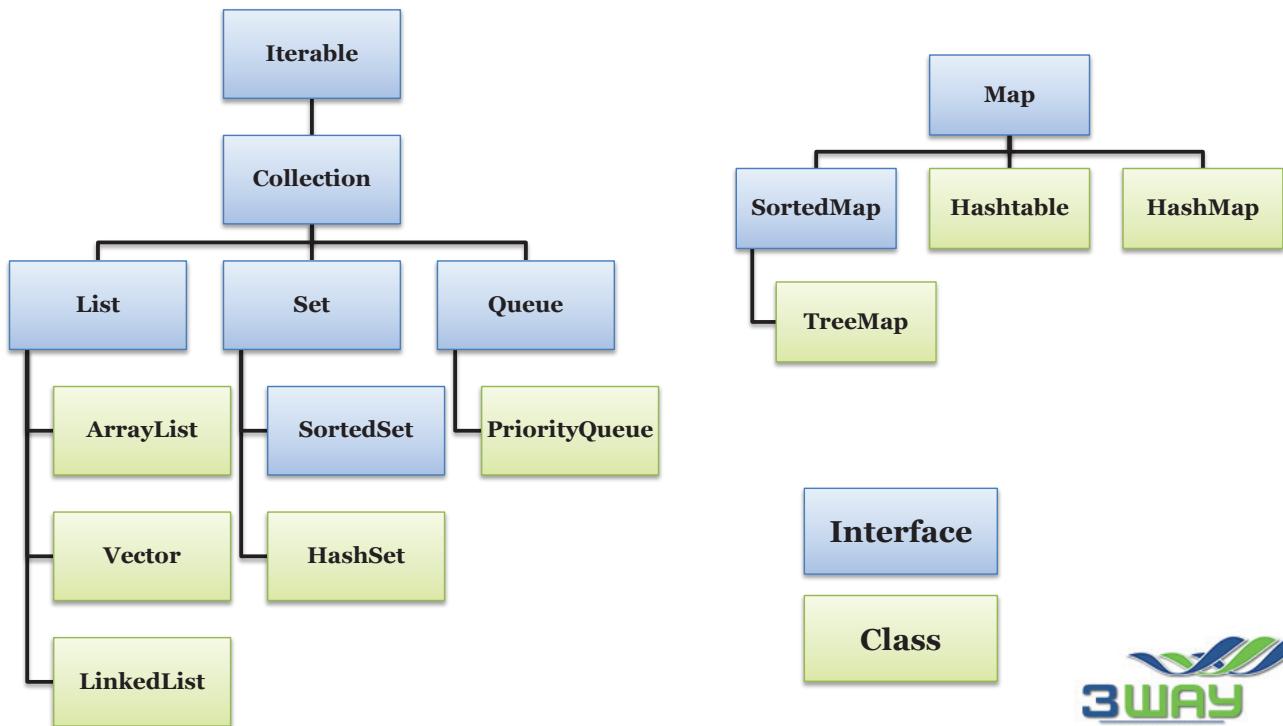
As estruturas de dados são muito utilizadas na programação em geral. Quando desenvolvemos algum programa, frequentemente utilizamos **arrays**. Em algoritmos complexos, listas (ligadas ou não), mapas, pilhas e filas são essenciais.

A infraestrutura Collections ou coleção do Java disponibiliza os recursos de estrutura de dados, ou seja, contém as principais funcionalidades para se trabalhar com conjunto de elementos (coleções).

Por serem muito utilizadas, várias estruturas já estão disponíveis no pacote **java.util** são chamados de **Java Collection Framework**, ou simplesmente **Collection**. Nestas classes podemos contar com **várias operações muito comuns**, como método de ordenação busca **interseção, união, diferença** e vários outros.

O Java possui uma interface raiz denominada Collections, com disponibilidade de outras estruturas para ser utilizada, em geral as Collections são estruturas dinâmicas, ou seja, podem crescer conforme a necessidade de expansão, diferente do array por exemplo.

# Java Collections - Hierarquia



## Java e Orientação a Objetos

Vamos entender que **Collection** é o topo da API Collections, disponível no pacote **java.util**. **Collections** é a API que oferece diversas coleções (interfaces e classes concretas).

O conjunto de classes Java para coleções (**JavaCollection Framework**) é composto de várias interfaces e classes concretas. Existem três interfaces principais: **Collection**, **Map** e **Queue**, que definem os métodos comuns de estruturas de dados do tipo conjuntos, mapas e filas respectivamente.

Caso você precise de alguma estrutura de dados, você deve utilizar uma das classes concretas, que de uma forma ou outra herdam de uma das três interfaces citadas.

As Interfaces e classes concretas que implementam a interface **Collection**, terão obrigatoriamente que fazer a implementação de diversos métodos que manipulam coleções de dados, tais como adicionar e remover elementos.

Algumas Collections são organizadas, ou seja, garante que as coleções serão percorridas na mesma ordem em que os elementos foram inseridos, já quando a Collection é ordenada, esta possui métodos, regras para ordenação dos elementos.

**Organizada:** `LinkedHashSet`, `ArrayList`, `Vector`, `LinkedList`, `LinkedHashMap`

**Nãoorganizada:** `HashSet`, `TreeSet`, `PriorityQueue`, `HashMap`, `HashTable`, `TreeMap`

**Ordenada:** `TreeSet`, `PriorityQueue`, `TreeMap`

**Nãoordenada:** `HashSet`, `LinkedHashSet`, `ArrayList`, `Vector`, `LinkedList`, `HashMap`, `HashTable`, `LinkedHashMap`.

Bem cada tipo apresentado, tem uma forma de trabalhar, vamos entender cada uma delas:

> **Set:** Não aceita itens duplicados, aceita nulos, não possui índice, rápido para inserir e pesquisar elementos;

> **HashSet:** esta é uma implementação concreta de Set não organizada, ou seja, os elementos são percorridos aleatoriamente, e também não é ordenada, não há regras de ordenação. Além disso, assim como Set, não aceita itens duplicados.

> **TreeSet:** implementação concreta de Set organizada, mas também não aceita itens duplicados.

> **TreeSet:** Também é uma implementação concreta, não aceitando itens duplicados, mas é ordenada.

> **List:** Aceita itens duplicados, organizada e todas as implementações seguem este padrão, elementos percorridos por ordem ser inserção, não ordenada, aceita nulo, trabalha com índices da mesma forma que os arrays, é rápido para inserir elementos, mas um pouco mais lento que o Set, já as pesquisas são mais lentas que o Set.

> **ArrayList:** implementação concreta de List, aceita itens duplicados e seu funcionamento é semelhante a um array convencional, com a principal diferença de ser dinâmica, ou seja, pode crescer conforme a necessidade.

> **Vector:** Outra implementação de List, pode ser visto como uma ArrayList, mas os métodos são sincronizados, ou seja, o acesso simultâneo por diversos processos será coordenado, é também mais lento que o ArrayList quando não há acesso simultâneo.

> **LinkedList:** outra implementação de List, mas também implementa Queue, aceitando itens duplicados e sendo organizada. É similar ao ArrayList e ao Vector, mas fornece alguns métodos adicionais para a inserção, remoção e acesso aos elementos no inicio e no final da lista. Possui melhor performance do que o ArrayList e o Vector quando se trata de inserir, remover e acessar elementos no inicio ou no final da lista, mas se for precisar acessar algum elemento pelo índice, a performance é muito inferior, sendo lento para pesquisas.

> **Queue:** é a fila, semelhante à lista, tendo como padrão o aceite duplicado de elementos, normalmente organizado, normalmente utilizado em itens que a ordem é importante, como uma fila de banco.

> **PriorityQueue:** implementação concreta de Queue e não aceita nulos

> **Map:** Um Map é um tipo de coleção que identifica os elementos por chaves, desta forma aceita itens duplicados com chaves diferentes.

> **HashMap:** implementação concreta de Map, aceita itens duplicados com chaves diferentes, não organizado, ou seja, os elementos são percorridos aleatoriamente, não ordenada e aceita nulos.

> **HashTable:** outra implementação de Map, aceita itens duplicados com chaves diferentes, semelhante ao HashMap mas os métodos são sincronizados.

> **TreeMap:** é uma outra implementação concreta de Map, também suporta itens duplicados com índices diferentes, é ordenado.

# Interfaces Set e List

## Interface Set



Coleções não ordenadas que não contém duplicidades

## Interface List



Coleções de classes ordenadas onde as duplicidades são permitidas.



## Java e Orientação a Objetos

Um List é uma Collection ordenada, também chamada de sequencia. O List pode conter elementos duplicados.

Existem diversos algoritmos à disposição que tornam a implementação mais simples quando o objetivo é manipular listas. Aqui está um resumo desses algoritmos:

- > **sort** – Ordena uma lista usando o algoritmo de ordenação Merge.
- > **shuffle** – permuta os elementos.
- > **reverse** – inverte a ordem dos elementos.
- > **rotate** – roda todos os elementos - lista circular.
- > **swap** – troca os elementos nas posições específicas.
- > **replaceAll** – substitui todas as ocorrências de um valor especificado com outro.
- > **fill** – substitui cada elemento em uma lista com o valor especificado.
- > **copy** – copia a lista de origem para lista de destino.

> **binarySearch** – Procura por um elemento em uma lista ordenada usando o algoritmo de busca binária.

> **indexOfSubList** – retorna o índice da primeira sublistas de uma lista que é igual a outra.

> **lastIndexOfSubList** – retorna o índice da última sublistas de uma lista que é igual a outra.

## Interface Set

Um **Set é uma Collection** que não pode conter elementos duplicados. Modela a abstração matemática dos conjuntos. A interface Set contém apenas métodos herdados da Collection e adiciona a restrição de que *elementos duplicados são proibidos*. Para por em prática esta característica, faz uso dos métodos *equals* e *hashCode*, garantindo a igualdade entre os objetos.

## Interface Map

Mapas (também chamados de arrays associativos) de objetos são mais uma das formas de organizarmos coleções de objetos, apesar de se parecerem com arrays de objetos, seus índices não precisam necessariamente ser valores inteiros positivos sequenciais, por isso podem ser instâncias de uma classe qualquer, simplificando: Mapas são conjuntos de pares de objetos, sendo um chamado chave e o outro, valor.

Mapas permitem valores iguais, porém, não permitem chaves repetidas, é importante lembrar que chaves diferentes podem ser ou estarem associadas a valores iguais.

No Java as interfaces e classes que implementam Mapas não herdam da interface Collection, mas mesmo assim é possível de forma separada acessar e manipular chaves e valores de mapas como se estes fossem coleções.

Os métodos que podem ser aplicados a um mapa são definidos pela interface Map, esses métodos são implementados por duas classes HashMap e TreeMap.

# Generics e Coleções Java

Métodos de **java.util.Collection<E>**:

```

boolean add(E e)
boolean addAll(Collection<? extends E> c)
boolean contains(Object o)
boolean containsAll(Collection<?> c)
boolean remove(Object o)
boolean removeAll(Collection<?> c)
boolean retainAll(Collection<?> c)
void clear()
int size()
boolean isEmpty()
Object[] toArray()
<T> T[] toArray(T[] a)

```

**Generics**

**<?> coringa**

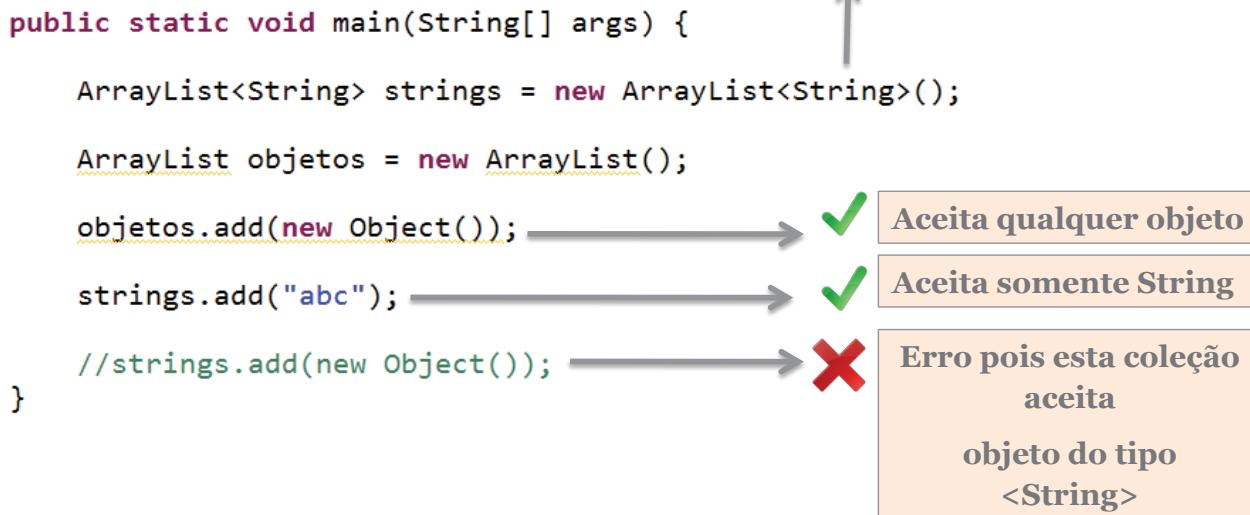


## Java e Orientação a Objetos

Os Principais métodos da Classe Collection

- > **add(Object objeto)** - Adiciona a coleção a um determinado objeto
- > **addAll(Collection outraCollection)** – Adiciona todos elementos de outra coleção.
- > **clear()** - Limpa todos os elementos de uma coleção.
- > **contains(Object objeto)** – Retorna true se o objeto já fizer parte da coleção.
- > **containsAll(Collection outraCollection)** – Retorna true caso todos os elementos de outra coleção estiverem presentes em determinada coleção.
- > **hashcode()** - Retorna o hashcode do objeto.
- > **iterator()** - Retorna o objeto de iteração com os elementos desta coleção.
- > **remove(Object objeto)** – Remove o objeto da coleção
- > **removeAll(Collection outraCollection)** – Remove todos os elementos que pertençam à coleção corrente e à outra coleção determinada.
- > **retainAll(Collection outraCollection)** – Remove todos os elementos que não fazem parte da coleção corrente e da outra coleção.
- > **Size()** - Retorna a quantidade elementos existentes na coleção.
- > **toArray()** - Retorna uma matriz de objetos(Object[]) dos elementos que estão contidos na coleção.
- > **toArray([]matriz)** – Retorna uma matriz do fornecido e, se a matriz contiver a quantidade de elementos suficiente, passa a ser utilizada para armazenamento.

# Generics e Coleções Java



## Java e Orientação a Objetos

O padrão das **Collections** é aceitar Generics, ou seja, aceitar qualquer tipo de elemento, inclusive elementos diferentes: Double, String, Integer, ou outro objeto que desejar.

Em qualquer lista, é possível colocar qualquer Object. Com isso, é possível misturar objetos.

Mas e depois, na hora de recuperar esses objetos? Como o método **get** devolve um Object, precisamos fazer o **cast**. Mas com uma lista com vários objetos de tipos diferentes, isso pode não ser tão simples.

Geralmente, não nos interessa uma lista com vários tipos de objetos misturados; no dia-a-dia, usamos listas como aquela de contas correntes. Podemos usar o recurso de Generics para restringir as listas a um determinado tipo de objetos(e não qualquer Object).

O uso de Generics também elimina a necessidade de casting.

A partir do Java 7, se você instancia um tipo genérico na mesma linha de sua declaração, não é necessário passar os tipos novamente, basta usar new **ArrayList<>()**. É conhecido como **operador diamante**.

# Interface Iterator

Uma referência **Iterator** é obtido na própria coleção, que **implementa Iterator**, através do método **iterator()**

```
public interface Iterator<E> {  
    //retorna true se houver mais elementos a iterar  
    boolean hasNext();  
    //retorna o proximo elemento na proxima iteracao  
    E next();  
    //remove o ultimo elemento retornado pela iteracao  
    void remove();  
}
```



## Java e Orientação a Objetos

O Iterator é uma interface disponível no pacote `java.util` que permite percorrer coleções da API Collection, desde que tenham implementado a Collection, fornecendo métodos como o `next()`, `hasnext()` e `remove()`.

A interface Iterator define métodos para percorrer Collections:

> **iterator()**: As classes que implementam a interface Collection oferecem este método que deve retornar um objeto Iterator para a coleção.

> **hasNext()**: devolve o valor true se ainda existir objeto a ser percorrido na coleção.

> **next()**: devolve o próximo objeto da coleção.

> **remove()**: remove o objeto da coleção.

# Percorrendo Collections

```
ArrayList<String> strings = new ArrayList<String>();
strings.add("String1");
strings.add("String2");
strings.add("String3");
```

Enhanced-for	Iterator
<pre>for (String str : strings) {     System.out.println(str); }</pre>	<pre>Iterator&lt;String&gt; iterator = strings.iterator(); while (iterator.hasNext()) {     System.out.println(iterator.next()); }</pre>



## Java e Orientação a Objetos

Há duas maneiras de percorrer coleções:

- > Usando o laço **for-enhanced** (ou **for-each**)
- > **Iteradores** – Visto anteriormente

O For-each é um ciclo for, mas que é adaptado para utilização em Collections e outras listas. Ele serve para percorrer todos os elementos de qualquer Collection contida na API Collections.

# Classificando Coleções: *Collections.sort*

A classe **Collections** nos permite ordenar coleções, através do um método estático **sort**, que recebe um **List** como argumento.

```
import java.util.*;

public class OrdenandoCollection {
    public static void main(String[] args) {
        List<String> lista = new ArrayList<String>();

        lista.add("Goiânia");
        lista.add("São Paulo");
        lista.add("Aracajú");
        // lista sem ordenação
        System.out.println(lista);
        Collections.sort(lista); → A Z ↓ → Ordenação
        // lista ordenada
        System.out.println(lista);

    }
}
```



## Java e Orientação a Objetos

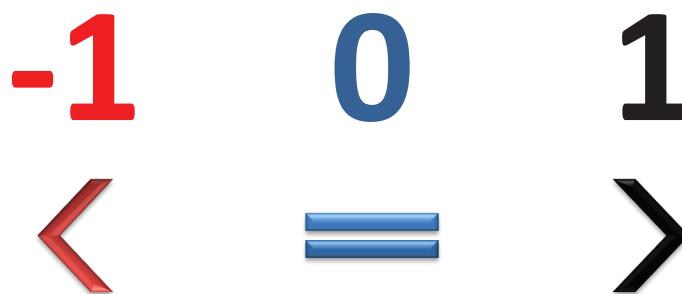
Vimos anteriormente que as listas são percorridas de maneira pré-determinada de acordo com a inclusão dos itens. Mas muitas vezes queremos **percorrer** a nossa lista de maneira **ordenada**, classificando os elementos de acordo com nossa necessidade.

A classe **Collections** (com s no final) nos permite **ordenar** coleções, através do um método estáticos **sort**, que recebe um **List** como argumento e o ordena em sua ordem natural (números = crescente, String = alfabética).

O método **sort()** utiliza a implementação do método **compareTo()**, da interface **Comparable**, implementado pela classe **String**, para classificar um **List** ou **arrays** de objetos **String**. Se quisermos fazer com que os elementos da nossa coleção sejam **ordenados** devemos **implementar** a interface **java.lang.Comparable**, e fornecer uma implementação para o método **int compareTo(Object)**, do objeto que será colocado na coleção.

# Interface Comparable

Para ordenar objetos criados devemos implementar a interface **java.lang.Comparable**, implementando o método **int compareTo(Object)** do objeto criado.



## Java e Orientação a Objetos

Essa interface fornece o método, `compareTo(Objeto)` , aonde você deverá sobre-screvê-lo para que, defina a ordem de comparação.

O método `compareTo()` deverá ser implementado. Ele retorna:

- > Um inteiro **menor que zero se objeto atual for “menor” que o recebido como parâmetro**
- > Um inteiro **maior que zero se objeto atual for “maior” que o recebido como parâmetro**
- > **Zero se objetos forem iguais**

# Java e Orientação a Objetos

## Lendo e Escrevendo Arquivos



## Java e Orientação a Objetos

Praticamente todos que trabalham com desenvolvimento, de uma forma ou de outra, acabam tendo que manipular arquivos, sejam eles de texto, planilhas ou gerar relatórios. Neste modulo será explicado como manipular arquivos com Java, bem como escrever e ler arquivos no formato de texto (txt).

A manipulação de arquivos em Java acontece de forma simples e rápida, pois a linguagem dispõe de classes que executam praticamente todas as operações necessárias para tanto.

# Console I/O

```
***** CUNIT CONSOLE - MAIN MENU *****
<Run all, <Select suite, <List suites. Show <Failures, <Quit
Enter Command : r

Running Suite Suite_success
  Running test : successful.test_1
  Running test : successful.test_2
  Running test : successful.test_3
WARNING Suite cleanup failed for Suite_init_failure.
Running Suite Suite_clean_failure
  Running test : successful.test_4
  Running test : successful.test_5
  Running test : successful.test_1
WARNING Suite cleanup failed for Suite_clean_failure.
Running Suite Suite_all
  Running test : successful.test_2
  Running test : failed.test_4
  Running test : failed.test_5
  Running test : successful.test_4

--Run Summary: Type Total Ran Passed Failed
  suites   4      3    n/a     2
  tests    15     10    7     3
  asserts  10     10    7     3
***** CUNIT CONSOLE - MAIN MENU *****
<Run all, <Select suite, <List suites. Show <Failures, <Quit
Enter Command :
```

```
import java.io.Console;

public class Teste {
    Console con = System.console();
}
```



System.in



System.out



## Java e Orientação a Objetos

### Java.io

Assim como todo o resto das bibliotecas em Java, a parte de controle de entrada e saída de dados (conhecido como **io**) é orientada a objetos e usa os principais conceitos mostrados até agora: interfaces, classes abstratas e polimorfismo.

A ideia atrás do polimorfismo no pacote `java.io` é de utilizar fluxos de entrada (`InputStream`) e de saída (`OutputStream`) para toda e qualquer operação, seja ela relativa a um **arquivo**, ou até mesmo às **entrada e saída padrão** de um programa (normalmente o teclado e o console).

As classes abstratas `InputStream` e `OutputStream` definem, respectivamente, o comportamento padrão dos fluxos em Java: em um fluxo de entrada, é possível ler bytes e, no fluxo de saída, escrever bytes.

### Console I/O

A classe `Console`, que é definida no pacote `java.io` como public e final, fornece métodos para acessar o dispositivo de console baseado em caracteres associado com a máquina virtual Java (JVM) sendo executada no momento. Um objeto desta classe é obtido por meio de uma chamada ao método `console()` da classe `System`.

Se a JVM atual tiver um console, então este é representado por uma instância única da classe `Console`, que pode ser obtida por meio de uma chamada ao método `console()` da classe `System`. Se nenhum dispositivo de console estiver disponível, uma chamada a este método retornará o valor `null`.

# Usando a classe Scanner



**Scanner** engloba diversos métodos para facilitar a entrada de dados

```
public static void main(String[] args) {
    Scanner scan = new Scanner(System.in);
    String teste = scan.nextLine();
    System.out.println("palavra digitada: " + teste);
}
```



## Java e Orientação a Objetos

Quem programa em linguagens estruturadas, como C e Pascal, e está aprendendo Java, depara-se com a seguinte situação: como atribuir valores para uma variável usando o teclado?

No **Java**, a partir do Java 1.5 ou J2SE 5, está disponível a classe **Scanner** do pacote **java.util**. Essa classe implementa as operações de entrada de dados pelo teclado no console.

A classe Scanner é do pacote **java.util**. Ela possui métodos muito úteis para trabalhar com Strings, em especial, diversos métodos já preparados para pegar números e palavras já formatadas através de expressões regulares.

Para utilizar a classe Scanner em uma aplicação Java deve-se proceder da seguinte maneira:

### Utilizar classe Scanner em uma aplicação Java

Passos	Descrição
[ 1 ]	importar o pacote <b>java.util</b> : <b>import java.util.Scanner;</b>
[ 2 ]	Instanciar e criar um objeto Scanner: <b>Scanner ler = new Scanner(System.in);</b>
[ 3 ]	Lendo valores através do teclado:

## Utilizar classe Scanner em uma aplicação Java

Passos	Descrição
[ 3.1 ]	Lendo um valor inteiro: <pre>int n; System.out.printf("Informe um número para a tabuada: "); n = ler.nextInt();</pre>
[ 3.2 ]	Lendo um valor real: <pre>float preco; System.out.printf("Informe o preço da mercadoria = R\$ "); preco = ler.nextFloat();</pre>
[ 3.3 ]	Lendo um valor real: <pre>double salario; System.out.printf("Informe o salário do Funcionário = R\$ "); salario = ler.nextDouble();</pre>
[ 3.4 ]	Lendo uma String: <pre>String s; System.out.printf("Informe uma cadeia de caracteres:\n"); s = ler.nextLine();</pre>

# java.io.File



**java.io.File**



Representa um arquivo ou diretório no sistema operacional.  
apenas **REPRESENTA**, não significa que o arquivo ou  
diretório realmente exista.



## Java e Orientação a Objetos

A classe File representa um arquivo ou diretório no sistema operacional. Importante saber que apenas REPRESENTA, não significa que o arquivo ou diretório realmente exista.

A classe se encontra no pacote java.io. Com dessa classe pode-se fazer algumas operações em um determinado path (caminho), sendo para um arquivo ou mesmo um diretório.

Intuitivamente pode-se notar que em se tratando de arquivos, conseguiremos descobrir seu tamanho, ultima modificação e também verificar algumas permissões, como por exemplo, de leitura e escrita.

### Alguns métodos da classe File

- > **boolean canRead()** retorna true se for possível ler o arquivo, falso o contrário
- > **boolean canWrite()** retorna true se for possível escrever no arquivo, falso o contrário
- > **boolean exists()** retorna true se o diretório ou arquivo se o objeto File existe, falso o contrário

> **boolean isFile()** retorna true se o argumento passado ao construtor da File é um arquivo, falso o contrário

> **boolean isDirectory()** retorna true se o argumento passado ao construtor da File é um diretório, falso o contrário

> **boolean isAbsolute()** retorna true para caso o argumento seja de um caminho absoluto, falso o contrário

- > **String getAbsolutePath()** retorna uma String com o caminho absoluto do diretório ou arquivo
- > **String getName()** retorna uma String com o nome do arquivo ou do diretório
- > **String getPath()** retorna uma String com o caminho do arquivo ou diretório
- > **String getParent()** retorna uma String com o caminho do diretório pai (acima | anterior) ao do arquivo ou diretório atual
- > **long length()** retorna um tamanho, em bytes, do arquivo ou inexistente, caso seja diretório
- > **long lastModified()** retorna o tempo em que o arquivo ou diretório foi modificado pela última vez; varia de acordo com o sistema
- > **String[] list()** retorna um array de Strings com o conteúdo do diretório, ou null se for arquivo

# FileWriter e BufferedWriter

```

public static void main(String[] args) {

    try {

        File arquivo = new File("C:\\\\teste.txt");

        FileWriter fw = new FileWriter(arquivo);

        BufferedWriter bw = new BufferedWriter(fw);

        bw.write("Texto a ser escrito no txt");
        bw.newLine();
        bw.write("Quebra de linha");

        bw.close();

        fw.close();

    } catch (IOException e) {
        System.out.println("Arquivo não existe!");
    }

}

```



## Java e Orientação a Objetos

As classes `FileWriter` e `BufferedWriter` servem para escrever em arquivos de texto.

A classe `FileWriter` serve para escrever diretamente no arquivo, enquanto a classe `BufferedWriter`, além de ter um desempenho melhor, possui alguns métodos que são independentes de sistema operacional, como quebra de linhas.

É possível escrever conteúdo no arquivo através do método `write()`.

Após escrever tudo que queria, é necessário fechar os buffers e informar ao sistema que o arquivo não está mais sendo utilizado através do método `close()`.

### Classes

Classe de <code>java.io</code>	Estende de	Argumento dos construtores	Principais métodos
<code>FileWriter</code>	<code>Writer</code>	<code>File</code> <code>String</code>	<code>flush()</code> <code>write()</code> <code>close()</code>
<code>BufferedWriter</code>	<code>Writer</code>	<code>Writer</code>	<code>newLine()</code> <code>write()</code> <code>close()</code> <code>flush()</code>

# FileReader e BufferedReader

```

public static void main(String[] args) {
    try {
        File arquivo = new File("C:\\\\teste.txt");
        FileReader fr = new FileReader(arquivo);
        BufferedReader br = new BufferedReader(fr);

        while (br.ready()) {
            String linha = br.readLine();
            System.out.println(linha);
        }

        br.close();
        fr.close();
    } catch (FileNotFoundException e) {
        System.out.println("Arquivo não existe!");
    } catch (IOException e) {
        System.out.println("Erro ao ler arquivo!");
    }
}
  
```



## Java e Orientação a Objetos

As funções de FileReader e BufferedReader, servem para ler o conteúdo de arquivos texto, porém uma apenas abre o arquivo para leitura (FileReader), enquanto a outra, percorre o arquivo, armazenando o valor, semelhante a ponteiros (BufferedReader), lembrando que o Buffer só chega ao fim do arquivo se for ‘null’.

Para ler o arquivo, basta utilizar o método ready(), que retorna se o arquivo tem mais linhas a ser lido, e o método readLine(), que retorna a linha atual e passa o buffer para a próxima linha.

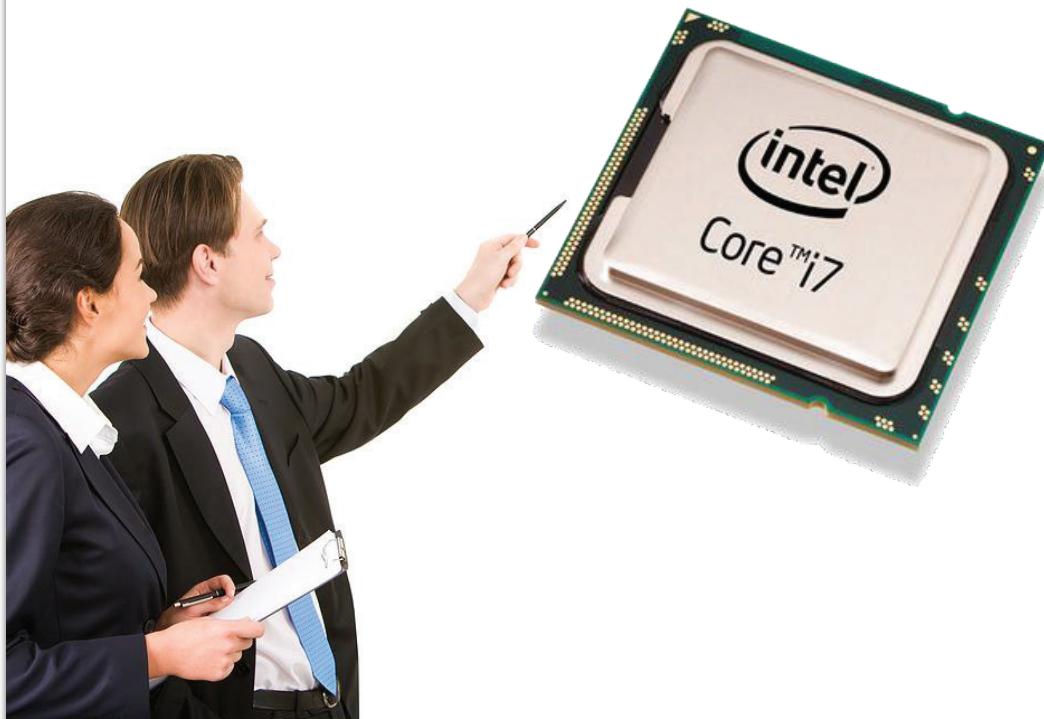
Da mesma forma que a escrita, a leitura deve fechar os recursos através do método close().

### Classes

Classe de java.io	Estende de	Argumento dos construtores	Principais métodos
FileReader	Reader	String File	read() close()
BufferedReader	Reader	Reader	read() readLine()

# Java e Orientação a Objetos

## Threads



## Java e Orientação a Objetos

Para entender o funcionamento de uma thread é necessário analisar, inicialmente, um processo. A maioria dos sistemas de hoje são baseados em computadores com um processador que executa várias tarefas simultâneas. Ou seja, vários processos que compartilham do uso da CPU tomando certas fatias de tempo para execução. A esta capacidade é denominado o termo multiprocessamento.

Uma solução encontrada foi o uso de threads, também conhecidas por linhas de execução. A thread pode ser vista como um subprocesso de um processo, que permite compartilhar a sua área de dados com o programa ou outras threads. O início de execução de uma thread é muito mais rápido do que um processo, e o acesso a sua área de dados funciona como um único programa.

# Threads

Pense em **threads** como processos do sistema operacional



## Java e Orientação a Objetos

### Definição de Processo

“Um processo é basicamente um programa em execução, sendo constituído do código executável, dos dados referentes ao código, da pilha de execução, do valor do contador de programa (registrador PC), do valor do apontador do apontador de pilha (registrador SP), dos valores dos demais registradores do hardware, além de um conjunto de outras informações necessárias à execução dos programas.”

Podemos resumir a definição de processo dizendo que o mesmo é formado pelo seu espaço de endereçamento lógico e pela sua entrada na tabela de processos do Sistema Operacional. Assim um processo pode ser visto como uma unidade de processamento passível de ser executado em um computador e essas informações sobre processos são necessárias para permitir que vários processos possam compartilhar o mesmo processador com o objetivo de simular paralelismo na execução de tais processos através da técnica de escalonamento, ou seja, os processos se revezam (o sistema operacional é responsável por esse revezamento) no uso do processador e para permitir esse revezamento será necessário salvar o contexto do processo que vai ser retirado do processador para que futuramente o mesmo possa continuar sua execução.

## Definição de Thread

Em várias situações, precisamos “rodar duas coisas ao mesmo tempo”, quando usamos o computador também fazemos várias coisas simultaneamente: queremos navegar na internet e *ao mesmo tempo* ouvir música.

A necessidade de se fazer várias coisas simultaneamente, ao mesmo tempo, **paralelamente**, aparece frequentemente na computação. Para vários programas distintos, normalmente o próprio sistema operacional gerencia isso através de vários processos em paralelo.

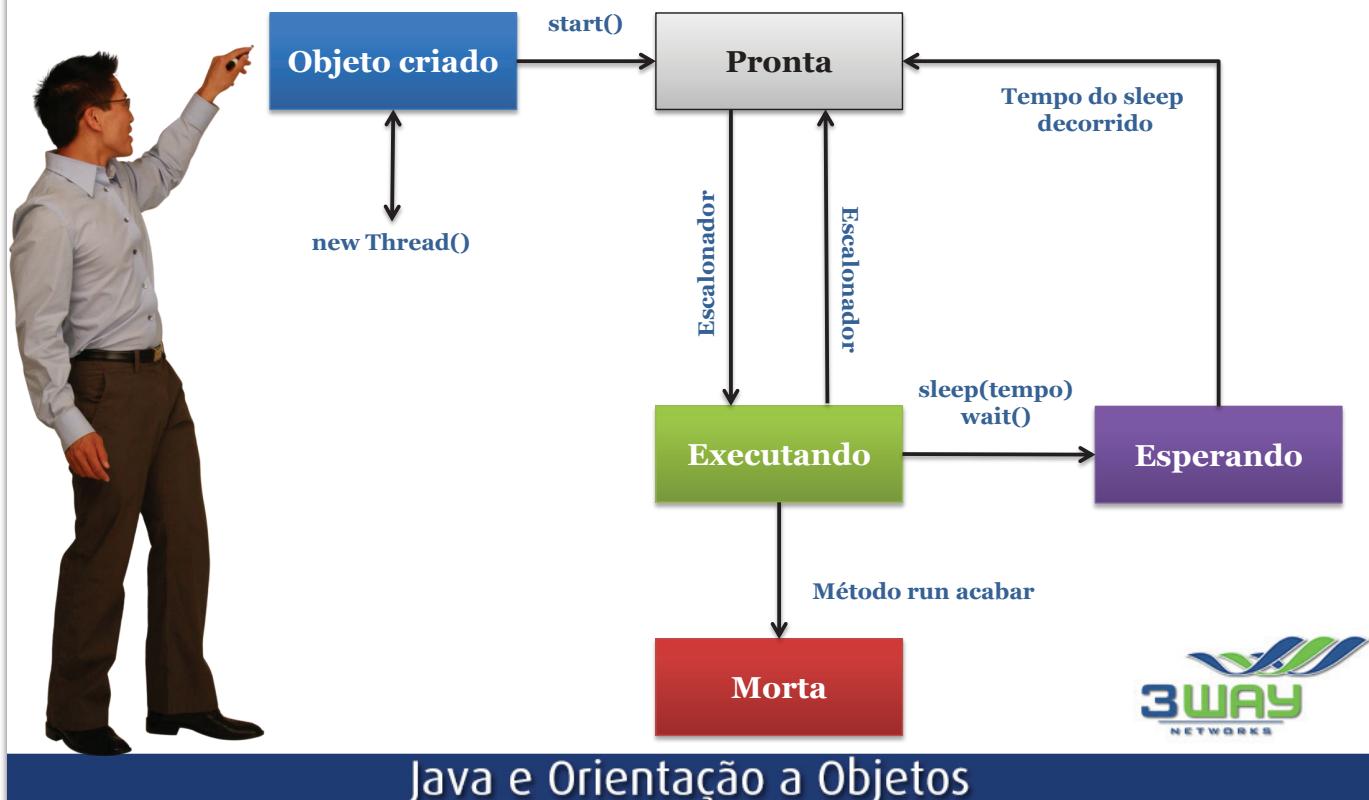
Em um programa só (um processo só), se queremos executar coisas em paralelo, normalmente falamos de **Threads**.

Um sistema de computação normalmente tem muitos processos e threads. Um processo tem um ambiente de execução próprio com um conjunto completo de recursos para a sua execução; em particular, um processo tem seu próprio espaço de memória.

Threads existem dentro de um processo; todo processo tem pelo menos uma thread. Threads compartilham os recursos do processo, incluindo memória e arquivos abertos. Isto torna eficiente, mas potencialmente problemática, a comunicação.

A execução multithread é uma característica da plataforma Java. Toda aplicação tem pelo menos uma thread – ou várias. Do ponto de vista do programador, você inicia com uma thread, chamada main thread - pense a main thread como o aplicativo Java com o método main(). Esta thread tem a capacidade de criar threads adicionais.

# Ciclo de vida de uma Thread



A melhor forma de analisar o ciclo de vida de uma thread é através das operações que podem ser feitas sobre as mesmas, tais como : criar, iniciar, esperar, parar e encerrar. O diagrama ilustra os estados que uma thread pode assumir durante o seu ciclo de vida e quais métodos ou situações levam a estes estados.

## Criando Threads

A criação de uma thread é feita através da chamada ao seu construtor colocando a thread no estado **Nova**, o qual representa uma thread vazia, ou seja, nenhum recurso do sistema foi alocado para ela ainda. Quando uma thread está nesse estado a única operação que pode ser realizada é a inicialização dessa thread através do método `start()`, se qualquer outro método for chamado com a thread no estado **Nova** irá acontecer uma exceção (`IllegalThreadStateException`), assim como, quando qualquer método for chamado e sua ação não for condizente com o estado atual da thread.

## Iniciando Threads

A inicialização de uma thread é feita através do método `start()` e nesse momento os recursos necessários para execução da mesma são alocados, tais como recursos para execução, escalonamento e chamada do método `run` da thread. Após a chamada ao método

start a thread está pronta para ser executada e será assim que for possível, até lá ficará no estado **Pronta**. Essa mudança de estado (Pronta/Executando) será feito pelo escalonador do Sistema de Execução Java. O importante é saber que a thread está pronta para executar e a mesma será executada, mais cedo ou mais tarde de acordo com os critérios, algoritmo, de escalonamento do Sistema de Execução Java.

## Fazendo Thread Esperar

- > Uma thread irá para o estado **Esperando** quando:
- > O método sleep (faz a thread esperar por um determinado tempo) for chamado;
- > O método wait (faz a thread esperar por uma determinada condição) for chamado;
- > Quando realizar solicitação de I/O.

Quando um thread for para estado Esperando ela retornará ao estado **Pronta** quando a condição que a levou ao estado **Esperando** for atendida, ou seja :

- > Se a thread solicitou dormir por determinado intervalo de tempo (sleep), assim que este intervalo de tempo for decorrido;
- > Se a thread solicitou esperar por determinado evento (wait), assim que esse evento ocorrer (outra thread chamar notify ou notifyAll);
- > Se a thread realizou solicitação de I/O, assim que essa solicitação for atendida.

## Finalizando Threads

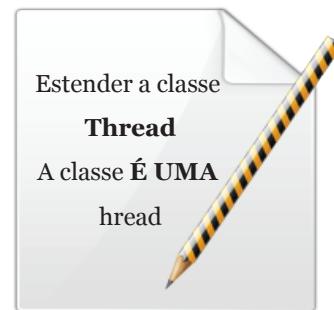
Uma Thread é finalizada quando acabar a execução do seu método run, e então ela vai para o estado Morta, onde o Sistema de Execução Java poderá liberar seus recursos e eliminá-la .

## Verificando se Threads estão Executando/Pronta/Esperando ou Novas/Mortas

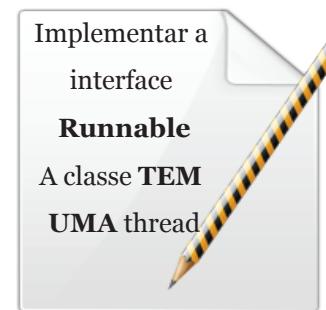
A classe Thread possui o método isAlive, o qual permite verificar se uma thread está nos estados **Executando/Pronta/Esperando** ou nos estados **Nova/Morta**. Quando o retorno do método for true a thread esta participando do processo de escalonamento e o retorno for false a thread está fora do processo de escalonamento. Não é possível diferenciar entre **Executando**, **Pronta** ou **Esperando**, assim também como não é possível diferenciar entre **Nova** ou **Morta**.

# Criando Threads

```
public class HerancaThread extends Thread {
    @Override
    public void run() {
        // conteúdo da thread
        System.out.println("extends Thread");
        for (int i = 0; i < 10; i++) {
            System.out.println(i);
        }
    }
}
```



```
public class RunnableThread implements Runnable {
    @Override
    public void run() {
        // conteúdo da thread
        System.out.println("implements Runnable");
        for (int i = 0; i < 10; i++) {
            System.out.println(i);
        }
    }
}
```



## Java e Orientação a Objetos

A linguagem Java possui apenas alguns mecanismos e classes desenhadas com a finalidade de permitir a programação Multi-Thread, o que torna extremamente fácil implementar aplicações Multi-Threads, sendo esses :

- > A classe `java.lang.Thread` utilizada para criar, iniciar e controlar Threads;
- > As palavras reservadas `synchronized` e `volatile` usadas para controlar a execução de código em objetos compartilhados por múltiplas threads, permitindo exclusão mútua entre estas;
- > Os métodos `wait`, `notify` e `notifyAll` definidos em `java.lang.Object` usados para coordenar as atividades das threads, permitindo comunicação entre estas.

## Criando Threads em Java

A criação de threads em Java é uma atividade extremamente simples e existem duas formas distintas de fazê-lo: uma através da herança da classe `Thread`, outra através da implementação da interface `Runnable`, e em ambos os casos a funcionalidade (programação) das threads é feita na implementação do método `run`.

## Implementando o Comportamento de uma Thread

O método `run` contém o que a thread faz, ele representa o comportamento (implementação) da thread, é como um método qualquer podendo fazer qualquer coisa que a lin-

guagem Java permita. A classe Thread implementa uma thread genérica que por padrão não faz nada, contendo um método run vazio, definindo assim uma API que permite a um objeto runnable prover uma implementação para o método run de uma thread.

## Criando uma subclasse de Thread

A maneira mais simples de criar uma thread em Java é criando uma subclasse de Thread e implementando o que a thread vai fazer sobrecarregando o método run.

## Implementando a Interface Runnable

A outra forma de criar threads em Java é implementando a interface Runnable e implementando o método run definido nessa interface. Sendo que a classe que implementa a thread deve declarar um objeto do tipo Thread e para instanciá-lo deve chamar o construtor da classe Thread passando como parâmetro a instância da própria classe que implementa Runnable e o nome da thread. Como o objeto do tipo Thread é local a classe que vai implementar thread, e normalmente privado, há a necessidade de criação de um método start para permitir a execução do objeto thread local. O mesmo deve ser feito para qualquer outro método da classe Thread que dever ser visto ou usado fora da classe que implementa Runnable.

## Escolhendo entre os dois métodos de criação de threads

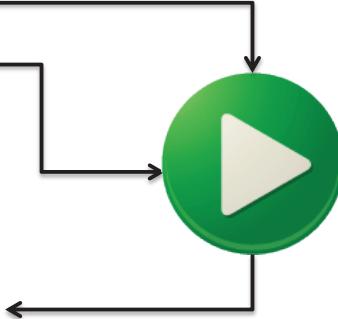
O principal fator a ser levado em consideração na escolha entre um dos dois métodos de criação de thread é que a linguagem Java não permite herança múltipla, assim quando uma classe que já for subclasse de outra precisar implementar thread é obrigatório que isso seja feito através da implementação da interface Runnable, caso contrário pode-se utilizar subclasses da classe Thread.

# Iniciando Threads



```
public static void main(String[] args) {  
    HerancaThread threadSimples = new HerancaThread();  
  
    Thread threadRunnable = new Thread(new RunnableThread());  
  
    threadSimples.start();  
    threadRunnable.start();  
}
```

Inicia execução do método run()



presente nas Threads



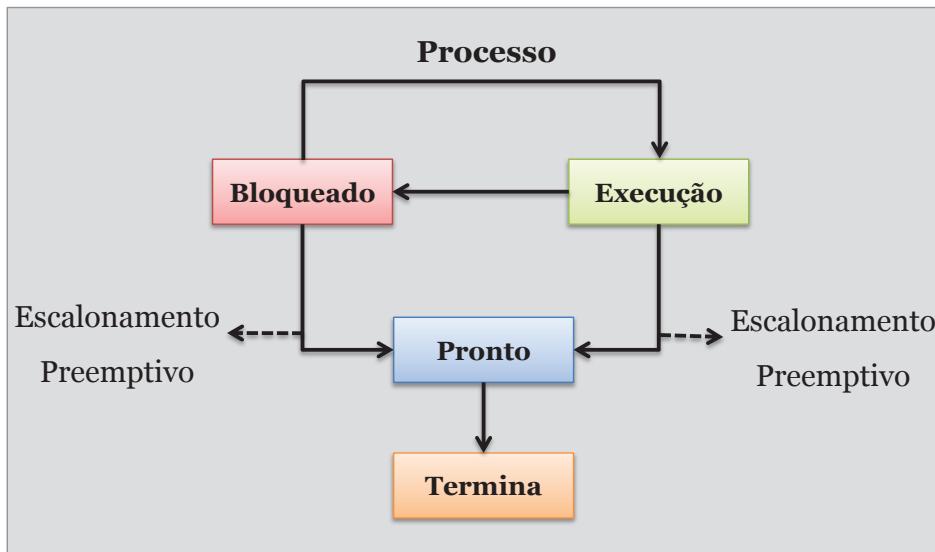
## Java e Orientação a Objetos

A inicialização de uma thread é feita através do método start() e nesse momento os recursos necessários para execução da mesma são alocados, tais como recursos para execução, escalonamento e chamada do método run da thread. Após a chamada ao método start a thread está pronta para ser executada e será assim que for possível, até lá ficará no estado **Pronta**. Essa mudança de estado (Pronta/Executando) será feito pelo escalonador do Sistema de Execução Java. O importante é saber que a thread está pronta para executar e a mesma será executada, mais cedo ou mais tarde de acordo com os critérios, algoritmo, de escalonamento do Sistema de Execução Java.

# Escalonamento da Thread



O Escalonador seleciona um entre os processos em memória prontos para executar e aloca a CPU para ele



## Java e Orientação a Objetos

O escalonamento é fundamental quando é possível a execução paralela de threads, pois, certamente existirão mais threads a serem executadas que processadores, assim a execução paralela de threads é simulada através de mecanismos do escalonamento dessas threads, onde os processadores disponíveis são alternados pelas diversas threads em execução. O mecanismo de escalonamento utilizado pelo Sistema de Execução Java é bastante simples e determinístico, e utiliza um algoritmo conhecido como **Escalonamento com Prioridades Fixas**, o qual escalona threads baseado na sua prioridade.

As threads escalonáveis são aquelas que estão nos estados **Executando** ou **Pron-ta**, para isso toda thread possui uma prioridade, a qual pode ser um valor inteiro no intervalo **[MIN\_PRIORITY ... MAX\_PRIORITY]**, (estas são constantes definidas na classe Thread), e quanto maior o valor do inteiro maior a prioridade da thread.

# Prioridades de uma Thread



```
public static void main(String[] args) {
    HerancaThread threadSimples = new HerancaThread();
    Thread threadRunnable = new Thread(new RunnableThread());
    threadSimples.setPriority(Thread.MAX_PRIORITY);
    threadSimples.start();
    threadRunnable.setPriority(Thread.MIN_PRIORITY);
    threadRunnable.start();
}
```

Maior a chance de ser executado antes



## Java e Orientação a Objetos

Cada thread Nova recebe a mesma prioridade da thread que a criou e a prioridade de uma thread pode ser alterada através do método setPriority(int priority).

O algoritmo de escalonamento com Prioridades Fixas utilizadas pelo Sistema de Execução Java funcionada da seguinte forma:

> Quando várias threads estiverem Prontas, aquela que tiver a maior prioridade será executada.

> Quando existir várias threads com prioridades iguais, as mesmas serão escalonadas segundo o algoritmo Round-Robin de escalonamento.

> Uma thread será executada até que : uma outra thread de maior prioridade fique Pronta; acontecer um dos eventos que a faça ir para o estado Esperando; o método run acabar; ou em sistema que possuam fatias de tempo a sua fatia de tempo se esgotar.

> Threads com prioridades mais baixas terão direito garantido de serem executadas para que situações de starvation não ocorram.

# Sincronização

Métodos **synchronized** têm um lock para acesso ao objeto.

métodos sem o modificador **synchronized** não exigem o lock do objeto para acesso.



Objeto

Cada objeto tem um lock de acesso.



## Java e Orientação a Objetos

Até o momento temos abordado threads como unidades de processamento independentes, onde cada uma realiza a sua tarefa sem se preocupar com o que as outras threads estão fazendo (paralelismos), ou seja, abordamos threads trabalhando de forma assíncrona e embora a utilização de threads dessa maneira já seja de grande utilidade, muitas vezes precisamos que um grupo de threads trabalhem em conjunto e agindo sobre objetos compartilhados, onde várias threads desse conjunto podem acessar e realizar operações distintas sobre esses objetos, sendo que, muitas vezes a tarefa de uma thread vai depender da tarefa de outra thread, dessa forma as threads terão que trabalhar de forma coordenada e simbiótica, ou seja, deverá haver uma sincronização entre as threads para que problemas potenciais advindos do acesso simultâneo a esses objetos compartilhado não ocorram.

Esses problemas são conhecidos como *condições de corrida* (Race Conditions) e os trechos de códigos, das threads, que podem gerar condições de corrida são chamados de *regiões críticas* (critical sections), assim as condições de corridas podem ser evitadas através da exclusão mútua das regiões críticas das threads e sincronização entre as threads envolvidas.

O mecanismo para prover exclusão mútua entre threads em Java é bastante simples, necessitando apenas da declaração dos métodos que contém regiões críticas como `synchronized`. Isso garante que quando uma das threads estiver executando uma região crítica em um objeto compartilhado nenhuma outra poderá fazê-lo. Ou seja, quando uma classe possuir métodos sincronizados, apenas um deles poderá estar sendo executado por uma thread.

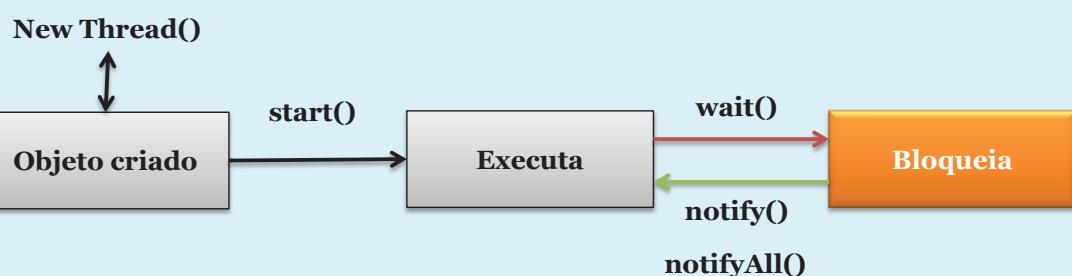
# Bloqueando acesso Concorrente

A coordenação é feita com métodos da classe Object:

**wait()**

**notify()**

**notifyAll()**



## Java e Orientação a Objetos

Outro aspecto muito importante na implementação de threads que trabalham concorrentemente sobre um objeto compartilhado é que muitas vezes a atividade de uma thread depende da atividade de outra thread, assim é necessário que as threads troquem mensagem a fim de avisarem umas as outras quando suas tarefas forem concluídas ou quando precisam esperar por tarefas de outras. Em Java isso é feito através dos métodos wait, notify, notifyAll, onde :

### Métodos

Nome	Descrição
wait	Faz a thread esperar (coloca no estado Esperando) em um objeto compartilhado, até que outra thread a notifique ou determinado intervalo de tempo seja decorrido.
notify	Notifica (retira do estado Esperando) uma thread que esta esperando em um objeto compartilhado.
notifyAll	Notifica todas as threads que estão esperando em um objeto compartilhado, onde uma delas será escalonada para usar o objeto e as outras voltarão ao estado Esperando.

**Observação:** Esses métodos devem ser usados na implementação das operações dos objetos sincronizados, ou seja, nos métodos synchronized dos objetos sincronizados. Pois só faz sentido uma thread notificar ou esperar por outra em objetos sincronizados e o

Sistema de Execução Java tem mecanismos para saber qual thread irá esperar ou receber a notificação, pois somente uma thread por vez pode estar executando um método sincronizado de um objeto compartilhado.

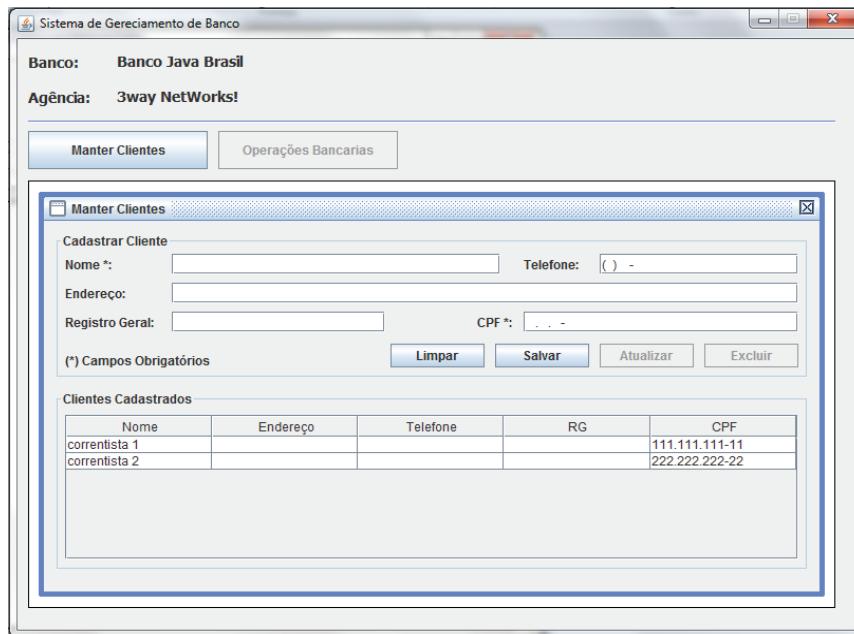
## Evitando Starvation e Deadlock

Com a programação concorrente de diversas threads que trabalham em conjunto e acessam objetos compartilhados surge o risco potencial de Starvation e Deadlock, onde Starvation acontece quando uma thread fica esperando por um objeto que nunca lhe será concedido e o Deadlock é quando duas ou mais threads esperam, de forma circular, por objetos compartilhados e nenhuma delas será atendida pelo fato de esperar umas pelas outras.

Na programação Java Starvation e Deadlock devem ser evitados pelo controle lógico de acesso aos objetos compartilhados, fazendo com que o programador deva identificar situação que possam gerar Starvation e Deadlock e evitá-las de alguma forma e com certeza essa é a tarefa mais desafiadora e trabalhosa na programação concorrente.

# Java e Orientação a Objetos

## Construindo interfaces gráficas com Swing



## Java e Orientação a Objetos

Mesmo sem conhecer interface gráfica com o usuário ou GUI (Graphical User Interface) é possível criar uma grande variedade de diferentes projetos. Entretanto, estas aplicações não teriam grandes chances de serem agradáveis aos usuários, tão acostumados aos ambientes gráficos como Windows, Solaris e Linux.

Ter um projeto desenvolvido em GUI afeta o uso de sua aplicação pois resulta em facilidade de uso e melhor experiência para os usuários de seus aplicativos. Java fornece ferramentas como Abstract Window Toolkit (AWT) e Swing para desenvolver aplicações GUI interativas.

# AWT versus Swing

AWT

Código nativo

São dependente de plataforma

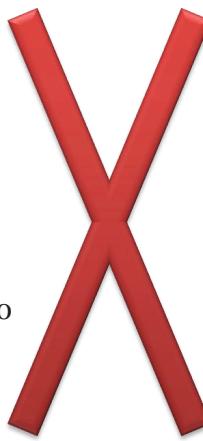
Assegura que a aparência de uma aplicação executada em diferentes máquinas seja comparável, mutável baseado no SO

Swing

Escrito em Java

Independente de plataforma

Mesma aparência em plataformas diferentes



## Java e Orientação a Objetos

A **JFC (Java Foundation Classes)** é uma importante parte do **Java SDK**. Refere-se a uma coleção de APIs que simplificam o desenvolvimento de aplicações Java GUI (**Graphical User Interface**). Consistem basicamente em cinco APIs incluindo **AWT** e **Swing**.

Nas primeiras versões do Java a única forma de fazer programas gráficos era através da AWT, uma biblioteca de baixo-nível que dependia de código nativo da plataforma onde rodava. Ela traz alguns problemas de compatibilidade entre as plataformas, fazendo que nem sempre o programa tinha aparência desejada em todos os sistemas operacionais, sendo também mais difícil de ser usada. Para suprir as necessidades cada vez mais frequentes de uma API mais estável e fácil de usar, o Swing foi criado como uma extensão do Java a partir da versão 1.2. Swing fornece componentes de mais alto nível, possibilitando assim uma melhor compatibilidade entre os vários sistemas onde Java roda. Ao contrário da AWT, Swing não contém uma única linha de código nativo, e permite que as aplicações tenham diferentes tipos de visuais ( skins ), os chamados “Look and Feel”. Já com AWT isso não é possível, tendo todos os programas a aparência da plataforma onde estão rodando. Apesar da AWT ainda estar disponível no Java, é altamente recomendável que seja usado Swing, pelas razões descritas aqui e por várias outras. Componentes Swing contêm um “J” na frente, como em JButton por exemplo. Componentes AWT não contêm inicial alguma ( “Button” no caso ).

## AWT

AWT (Abstract Window Toolkit) é um conjunto de recursos gráficos comumente conhecidos pelos sistemas de interface usando janelas. Podemos criar janelas, botões, janela de diálogos, etc.

Programação orientada a eventos é uma técnica de programação na qual, construímos o programa através de regras (métodos) a ser executados cada vez que o evento diferente ocorrer. Por exemplo, pressionar teclado, mover mouse, clicar no botão, receber dados pela rede, todos são eventos. Cada vez que um desses eventos ocorrer, o método correspondente é ativado.

Os elementos de AWT são na maioria, componentes, isto quer dizer que eles podem ser adicionados numa janela (frame) ou painel (panel). Para começar, precisamos de uma janela ou uma área em branco para poder colocar os botões e menus.

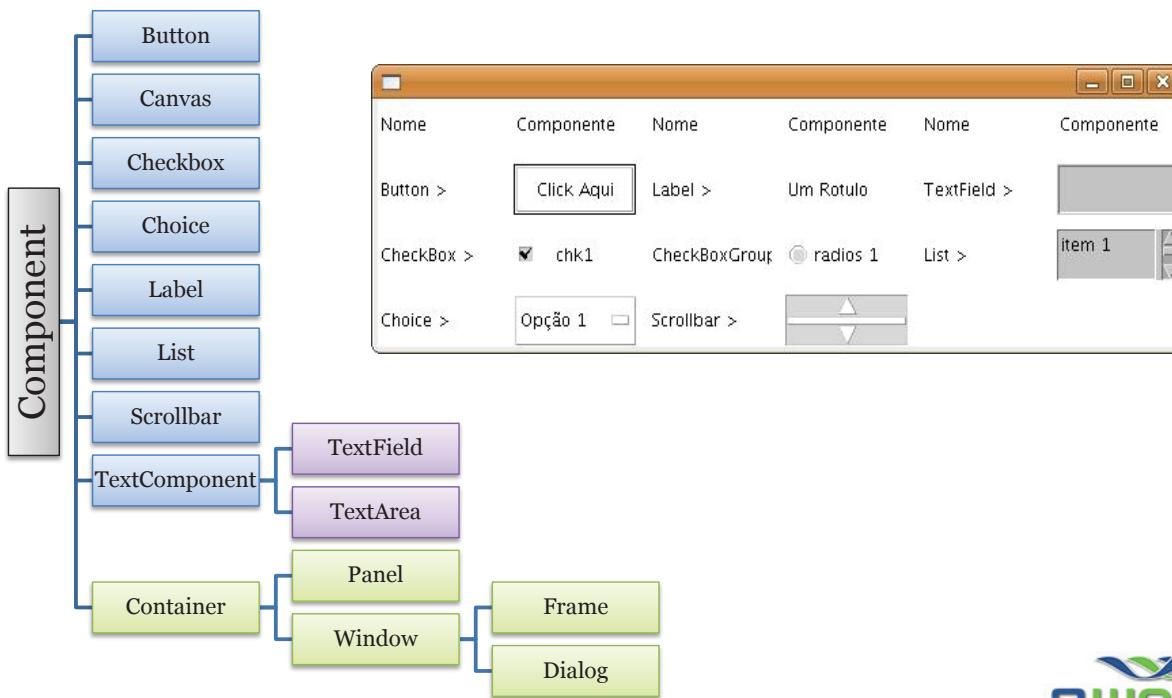
## Swing

É um widget toolkit para uso com o Java. Ele é compatível com o Abstract Window Toolkit (AWT), mas trabalha de uma maneira totalmente diferente. A API Swing procura renderizar/desenhar por conta própria todos os componentes, ao invés de delegar essa tarefa ao sistema operacional, como a maioria das outras APIs de interface gráfica trabalham.

Por ser uma API de mais alto nível, ou seja, mais abstração, menor aproximação das APIs do sistema operacional, ela tem bem menos performance que outras APIs gráficas e consome mais memória RAM em geral. Porém, ela é bem mais completa, e os programas que usam Swing têm uma aparência muito parecida, independente do Sistema Operacional utilizado.

Ambas **AWT** e **Swing** dispõem de componentes GUI que podem ser usadas na criação de aplicações **Java** e **Applets**. Diferentemente de alguns componentes **AWT** que usam código nativo, a API gráfica de seu sistema operacional. **Swing** fornece uma implementação independente de plataforma que assegura que aplicações desenvolvidas em diferentes plataformas tenham a mesma aparência. **AWT**, entretanto, assegura que o **look and feel** (a aparência) de uma aplicação executada em duas máquinas diferentes sejam compatíveis. A API **Swing** é construída sobre um número de **APIs** que implementa várias partes da **AWT**. Como resultado, componentes **AWT** ainda podem ser usados com componentes **Swing**, mas não é recomendado.

# Componentes AWT



## Java e Orientação a Objetos

Apesar de não ser muito utilizada a **AWT**, ainda é utilizada indiretamente por **Swing**, como **gerenciadores de layouts** e classes para **controle de eventos**. Assim apresentamos alguns componentes para que você se familiarize com o processo de desenvolvimento e vá gradualmente aprendendo a utilizar a **JFC**.

## Fundamental Window Classes

No desenvolvimento de aplicações GUI, os componentes como os botões ou campos de texto são localizados em **containers**. Essa é uma lista de importantes classes **containers** fornecida pela AWT.

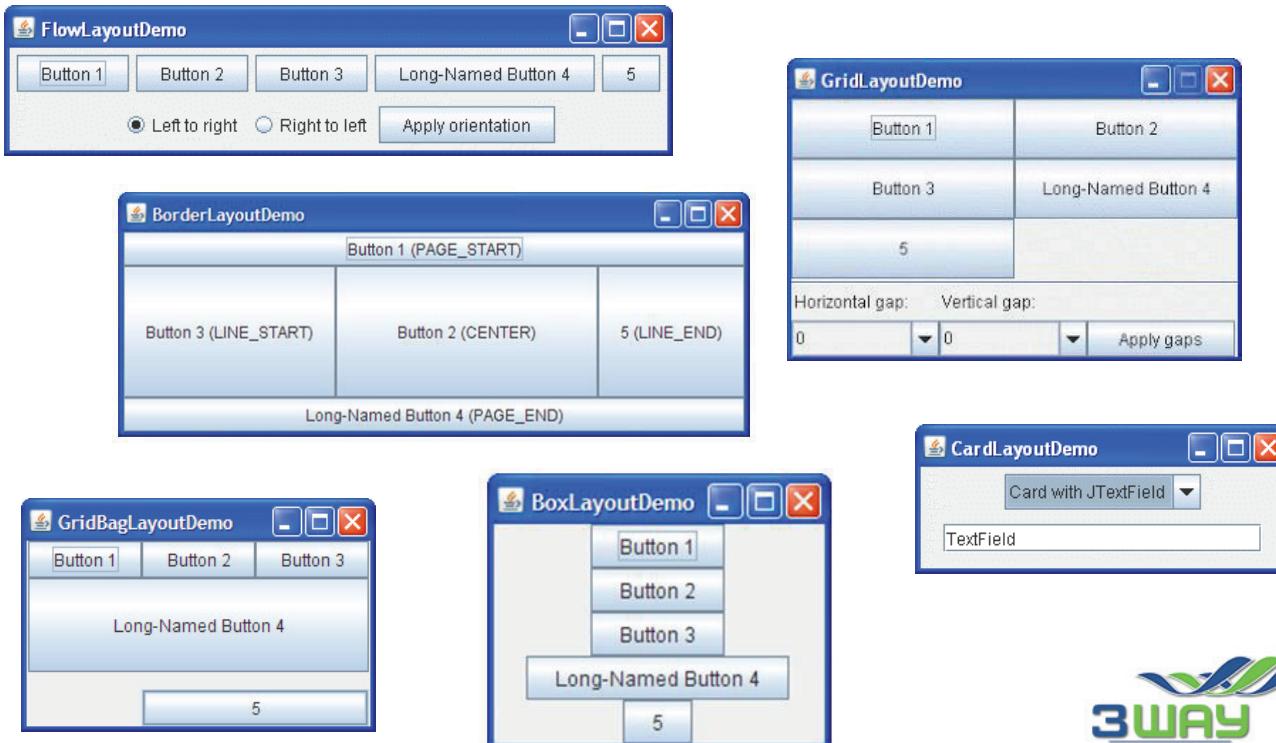
### Classes containers AWT

Classe AWT	Descrição
Component	Uma classe abstrata para objetos que podem ser exibidos no console interagir com o usuário. A raiz de todas as outras classes AWT.
Container	Uma subclasse abstrata da classe Component. Um componente que pode conter outros componentes.

## Classes containers AWT

Classe AWT	Descrição
Panel	Herda a classe Container. Uma área que pode ser colocada em um Frame, Dialog ou Window. Superclasse da classe Applet.
Window	Também herda a classe Container. Uma janela top-level, que significa que ela não pode ser contida em nenhum outro objeto. Não tem bordas ou barra de menu.
Dialog	Uma janela contendo a barra de título e o botão de fechar, utilizada para criar janelas para comunicação com o usuário.
Frame	Uma janela completa com um título, barra de menu, borda, e cantos redimensionáveis. Possui quatro construtores, dois deles possuem as seguintes assinaturas: Frame() Frame(String title)

# Gerenciadores de layout



## Java e Orientação a Objetos

Gerenciadores de layout são objetos que auxiliam os **containers** no posicionamento relativo de seus componentes. Isso inclui a adição de componentes no **container**, o redimensionamento, o alinhamento e a ancoragem dos componentes, assim como o comportamento de autorredimensionamento dos mesmos.

Isso quer dizer que ao redimensionar uma janela de um aplicativo, a reorganização dos componentes depende do gerenciador de **layout** utilizado. O Swing também permite que não utilize qualquer gerenciador de layout, ficando a cargo do programador posicionar cada componente em coordenadas que indicam a posição absoluta dentro do **container**. Nesse caso, ao redimensionar a janela, você não perceberá mudança no tamanho dos componentes.

O Swing provê vários gerenciadores de layout para uso geral. Entre eles estão os que seguem na tabela:

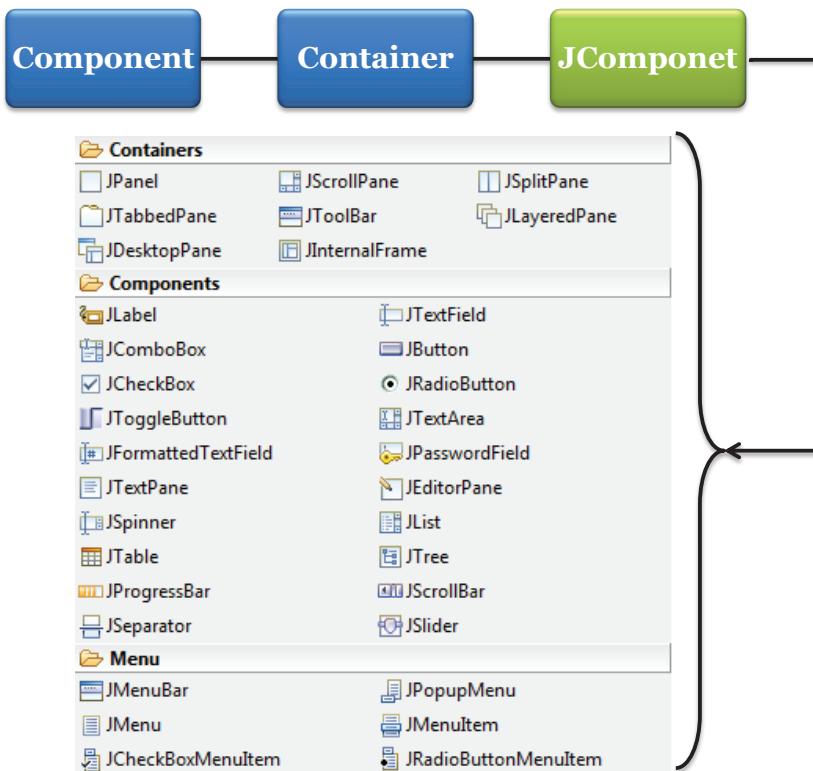
## Gerenciadores de layout

Nome	Descrição
Border Layout	O BorderLayout é o gerenciador de layout padrão em todo o ‘content pane’. ‘content pane’ é o container principal de todos os ‘frames’. Ele posiciona componentes em até cinco áreas: o topo, a base, a esquerda, a direita e o centro. Todo o espaço extra é posicionado na área central.

## Gerenciadores de layout

Nome	Descrição
BoxLayout	O BoxLayout posiciona os componentes em uma simples linha ou coluna. Ele respeita o tamanho máximo requerido por cada componente e permite que você alinhe os componentes.
CardLayout	O CardLayout permite que você implemente uma área que contenha diferentes componentes em diferentes momentos. Ele é geralmente controlado por um comboBox, com o estado do comboBox determinando que grupo de componentes o CardLayout irá mostrar.
FlowLayout	O FlowLayout é o gerenciador padrão para todo JPanel. Ele simplesmente organiza os componentes em uma linha, começando uma nova linha caso o container não seja largo o suficiente.
GridBagLayout	O GridBagLayout é um gerenciador sofisticado e flexível. Ele alinha componentes posicionando-os dentro de uma grade de células, permitindo aos componentes ocupar mais de uma célula. As linhas na grade podem ter alturas diferentes, assim como as colunas podem ter larguras diferentes.
GridLayout	O GridLayout simplesmente cria um grupo de componentes iguais em tamanho e os mostra em uma quantidade especificada de linhas e colunas.

# Componentes Swing



## Java e Orientação a Objetos

Com exceção dos **top-level containers**, todos os componentes do Swing cujo nome começa com “J” são descendentes da classe **JComponent**. Por exemplo, **JPanel**, **JScrollPane**, **JButton** e **JTable** – Todos herdam de **JComponent**. Já o **JFrame** e **JDialog** não, porque eles são containers top-level.

A classe **JComponent** provê as seguintes características a seus descendentes:

> **Tool tips**: especificando uma string (texto) você pode prover ajuda aos usuários sobre determinado componente.

> **Bordas e desenho**: permite especificar as bordas que um componente mostra ao redor das extremidades. É possível, também, desenhar dentro de um componente.

> **Look-and-feel**: Possibilidade de mudança de visual e de comportamento para cada componente.

> **Propriedades customizadas**: você pode associar uma ou mais propriedades a cada Jcomponent.

> **Suporte a layout**: possibilidade de mudar características de layout como tamanho mínimo do componente, alinhamento etc.

> **Drag and drop**: suporte a arrastar e soltar para componentes.

> **Buffer duplo**: suaviza o aparecimento dos componentes na tela.

> **Key binding**: associa teclas do teclado a eventos dos componentes.

## Campo de texto (TextField)

Esse componente é um dos mais básicos, muito comum em formulários ou em interfaces onde o usuário digita um texto como uma informação a ser enviada.

## Rótulo (label)

Esse é um outro componente bastante comum. Podemos dizer até que está presente em toda interface gráfica. O Rótulo, é todo o texto que encontramos na interface que tem a função de informar o usuário.

## Botões (Button)

Você, ao utilizar programas com interface gráfica em ambientes como Windows e/ou Linux, por exemplo, certamente, já se deparou com botões em suas interfaces. Portanto, o seu uso é bastante óbvio, pois os botões permitem a ativação de certas ações.

## O componente ‘Botão de opção’ (RadioButton)

O componente ‘Botão de opção’, também conhecido como RadioButton, é utilizado para você selecionar uma e somente uma opção entre várias oferecidas em um determinado grupo, ou seja, um grupo desses botões não permite mais de uma seleção por vez. Esse componente deverá ser utilizado em situações onde existam pelo menos duas opções de escolha.

## O componente ‘Grupo de botões’ (ButtonGroup)

Para resolver o problema dos ‘Botões de opção’ primeiramente devemos acomodar os botões (no mínimo dois) dentro de um painel, que é um recipiente (contêiner) que tem por finalidade separar esses botões de opção dos outros grupos de botões de opção (se houver na mesma aplicação). Depois disso, entra em campo um outro componente chamado ‘Grupo de botões’ (ButtonGroup), cuja finalidade é exatamente agrupar e dar um nome a esse grupo de botões de opção e gerenciar o estado de cada um deles que fizerem parte do mesmo grupo, garantido que um único botão do grupo possa ser selecionado de cada vez. Geralmente esses componentes dependem uns dos outros.

## O componente ‘Caixa de combinação’ (ComboBox)

Nesse componente a lista de itens de um componente ‘Caixa de combinação’ fica escondida, restrita apenas a uma linha, ocupando, dessa forma, menos espaço do que outros componentes que você irá conhecer no decorrer desse curso. Esse componente é utilizado para armazenar informações em forma de uma lista, sendo cada item em uma linha, e apenas uma seleção será permitida.

## O componente ‘Área de Texto’ (TextArea)

O componente ‘Área de Texto’ é nada mais nada menos do que uma caixa de texto com maiores recursos do que uma caixa de texto convencional ( JTextField ). Você poderá usá-lo para mostrar texto e também editar e receber entrada de dados.

## O componente ‘Caixa de Seleção’ (JCheckBox)

O componente Caixa de Seleção, ou CheckBox, é um pequeno quadrado que pode ser marcado ou desmarcado. Quando marcado, um sinal de checagem aparece na caixinha. Um texto poderá ser adicionado a esse componente, e poderá ser colocado à sua direita, esquerda ou no centro. O mais comum é à direita.

Esse componente é exatamente o oposto do RadioButton, ou seja, é permitido selecionar tantos componentes CheckBox quantos forem necessários, e não apenas uma opção. Ele é muito utilizado onde mais de uma situação é satisfeita à questão em pauta.

## O componente ‘Campo de senha’ (JPasswordField)

Muito comum em qualquer interface que solicite uma senha de acesso, o componente ‘Campo de senha’ é uma caixa de texto em que o conteúdo digitado não é exibido como em uma caixa de texto convencional (JTextField). Nesse componente, o texto digitado geralmente aparece como asteriscos (por exemplo ‘\*\*\*\*\*’, também pode aparecer como bolinhas ‘.....’), independentemente do que seja digitado, a fim de manter o sigilo da informação.

# Containers JFrame

```

public class AppSwing extends JFrame {
    JButton botao;
    JLabel label;
    public AppSwing() {
        super("Primeira aplicação Swing");
        setSize(300, 100);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLayout(new FlowLayout());
        initialize();
    }
    private void initialize() {
        botao = new JButton("Um Botão");
        label = new JLabel("Algum rotulo");
        getContentPane().add(botao);
        getContentPane().add(label);
    }
    public static void main(String[] args) {
        AppSwing app = new AppSwing();
        app.setVisible(Boolean.TRUE);
    }
}

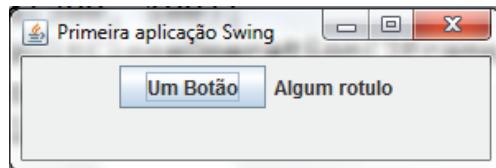
```

Um container é um agrupamento ou uma coleção de JComponents

Construtor

Container que recebe os componentes

Inicia o frame e o deixa visível



## Java e Orientação a Objetos

Veja três maneiras de organizar componentes em uma GUI:

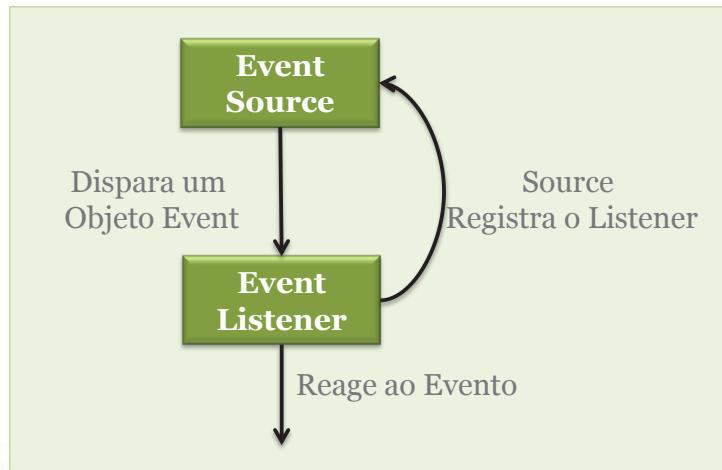
> **Posicionamento absoluto:** fornece o maior nível de controle sobre a aparência de uma GUI, pois configura o layout de um Container como null podendo especificar a posição absoluta de cada componente GUI em relação ao canto superior esquerdo do Container usando os métodos Component setSize e setLocation ou setBounds.

> **Gerenciadores de layout:** é mais simples e mais rápido para criar uma posição GUI com posicionamento absoluto, mas acaba perdendo controle sobre tamanho e o posicionamento dos componentes GUI.

> **Programação visual em uma IDE:** facilita o desenvolvimento em GUI, pois toda IDE tem uma ferramenta de design que permite arrastar e soltar (drag and drop) os componentes para uma área de desenho.

Como mencionado, os containers top-level como o **JFrame** e o **JDialog** no pacote **Swing** são ligeiramente incompatíveis com seus correspondentes AWT. Isso em termos de adição de componentes ao container. Ao invés de adicionar diretamente um componente ao container como nos containers AWT, é necessário primeiro pegar o conteúdo do **ContentPane** do container. Para fazer isso utiliza-se o método **getContentPane()** do container.

# Manipulação de Evento



## Java e Orientação a Objetos

Os componentes **Swing** são construídos com base no padrão de projeto denominado **Model Delegate (Modelo de Delegação)**. Este padrão descreve o modo como sua classe pode responder a uma interação do usuário. Para compreender o modelo, devemos perceber a colaboração entre três componentes:

> **Event Source (Gerador de Evento)** - O **event source** refere-se ao componente da interface que origina o evento. Por exemplo, se o usuário pressiona um botão, o event source neste caso é o botão.

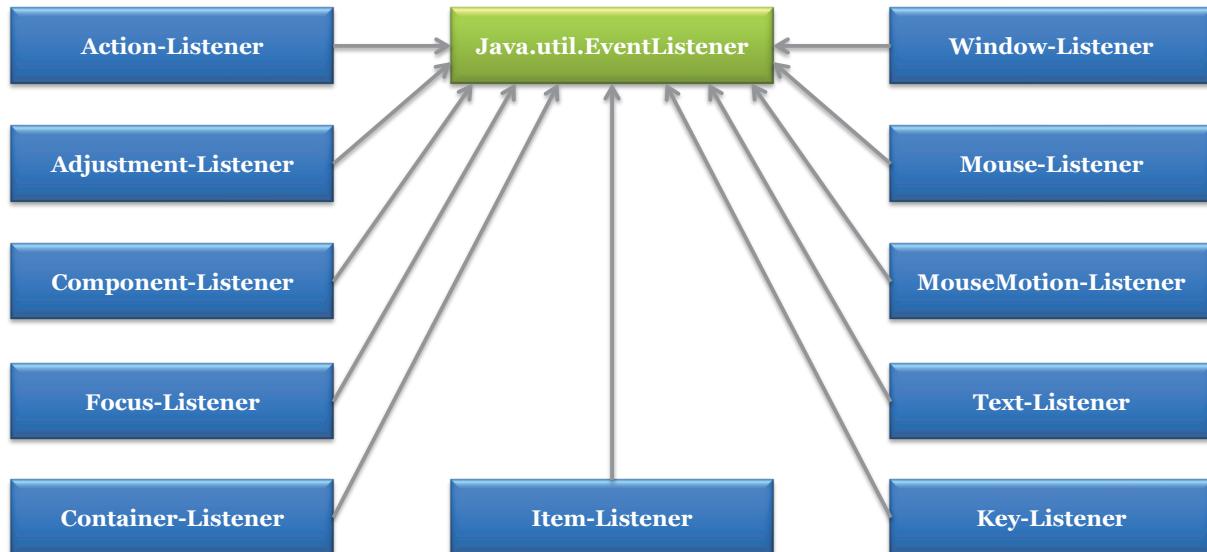
> **Event Listener/Handler (Monitor de Eventos/Manipulador)** - O **event listener** recebe informações de eventos e processa as interações do usuário. Quando um botão é pressionado, o **event listener** pode tratá-lo exibindo uma informação útil ao usuário.

> **Event Object (Objeto Evento)** - Quando um evento ocorre (por exemplo, quando o usuário interage com um componente da interface gráfica), um objeto de **evento** é criado. Este objeto contém todas as informações necessárias sobre o evento gerado. As informações incluem o tipo de evento, digamos, o clique do mouse. Há diversas classes de evento para diferentes categorias de ações do usuário. Um objeto de **evento** tem o tipo de dado de uma dessas classes.

Inicialmente, um **listener** deverá ser **registrado** pelo **event source**. Assim ele poderá receber informações sobre os eventos quando ocorrerem no **event source**. Somente um **listener** registrado poderá receber notificações dos eventos. Uma vez registrado, um **listener** simplesmente aguarda até que ocorra um evento.

Quando alguma coisa acontece no **event source**, um objeto **event**, que descreve o evento, é criado. O evento é então disparado pelo gerador para os **listener** registrados. Quando o **listener** recebe um objeto **event** (ou seja, uma notificação) de um **event source**, ele executa sua função. Ele decifra a notificação e processa o evento ocorrido.

# Classes de Evento



## Java e Orientação a Objetos

Um **EventObject** tem uma classe de evento que indica seu tipo. Na raiz da hierarquia das classes **Event** estão a classe **EventObject**, que é encontrada no pacote **java.util**. Uma subclasse imediata da classe **EventObject** é a classe **AWTEvent**. A classe **AWTEvent** está declarada no pacote **java.awt**. Ela é a raiz de todos os eventos baseados em AWT. A seguir temos algumas das classes de evento AWT:

### Classes de evento AWT

Classes de evento	Descrição
ComponentEvent	Estende a classe AWTEvent. Instanciada quando um componente é movido, redimensionado, tornado visível ou invisível.
InputEvent	Estende a classe ComponentEvent. Classe abstrata de evento, raiz a partir da qual são implementadas todas as classes de componentes de entrada de dados.
ActionEvent	Estende a classe AWTEvent. Instanciada quando um botão é pressionado, um item de uma lista recebe duplo-clique ou quando um item de menu é selecionado.

## Classes de evento AWT

Classes de evento	Descrição
ItemEvent	Estende a classe AWTEvent. Instanciada quando um item é selecionado ou desmarcado pelo usuário, seja numa lista ou caixa de seleção (checkbox).
KeyEvent	Estende a classe InputEvent. Instanciado quando uma tecla é pressionada, liberada ou digitada (pressionada e liberada).
MouseEvent	Estende a classe InputEvent. Instanciada quando um botão do mouse é pressionado, liberado, clicado (pressionado e liberado) ou quando o cursor do mouse entra ou sai de uma parte visível de um componente.
TextEvent	Estende a classe AWTEvent. Instanciada quando o valor de um campo texto ou área de texto sofre alteração.
WindowEvent	Estende a classe ComponentEvent. Instanciada quando um objeto Window é aberto, fechado, ativado, desativado, minimizado, restaurado ou quando o foco é transferido para dentro ou para fora da janela.

**Listeners** de evento são classes que implementam as interfaces <Tipo>Listener. A tabela a seguir algumas interfaces utilizadas com maior frequência:

## Interfaces mais utilizadas

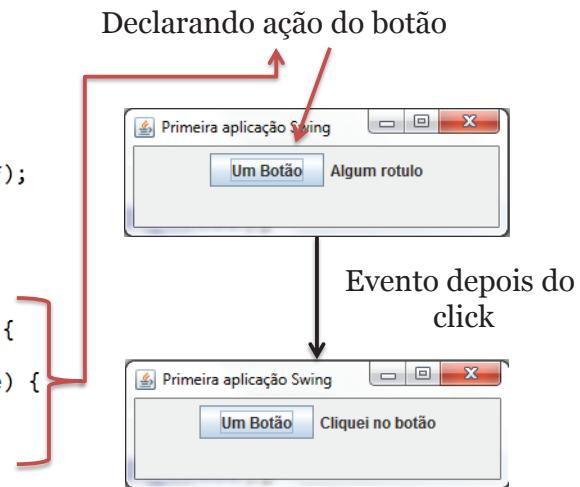
Classes de evento	Descrição
ActionListener	Recebe eventos de ação, estes podem ser um pressionamento do mouse ou da barra de espaço sobre o objeto.
MouseListener	Recebe eventos do mouse.
MouseMotionListener	Recebe eventos de movimento do mouse, que incluem arrastar e mover o mouse.
WindowListener	Recebe eventos de janela (abrir, fechar, minimizar, entre outros).

# Criação de aplicações gráficas com Eventos

```

public class AppSwing extends JFrame {
    JButton botao;
    JLabel label;
    public AppSwing() {
        super("Primeira aplicação Swing");
        setSize(300, 100);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLayout(new FlowLayout());
        initialize();
    }
    private void initialize() {
        botao = new JButton("Um Botão");
        botao.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                label.setText("Cliquei no botão");
            }
        });
        label = new JLabel("Algum rotulo");
        getContentPane().add(botao);
        getContentPane().add(label);
    }
    public static void main(String[] args) {
        AppSwing app = new AppSwing();
        app.setVisible(Boolean.TRUE);
    }
}

```



## Java e Orientação a Objetos

Como foi visto, as **anonymous inner class**, são classes internas que são declaradas sem um nome. Normalmente são utilizadas em contextos onde o reuso da classe não é importante. Por exemplo podemos declarar uma **anonymous inner class** como um argumento de método ou atribuição de uma variável.

Se você observar a listagem anterior verá que o manipulador de evento **ActionListener** foi implementado pela própria classe **AppSwing**, mas essa estratégia pode causar alguns transtornos quando você tem que manipular as ações de vários botões.

Estes são os passos a serem lembrados ao criar uma aplicação gráfica com tratamento de eventos:

- > Criar uma classe que descreva e mostre a aparência da sua aplicação gráfica.
- > Criar um objeto em tempo de execução para o evento **new ActionListener()**.
- > No objeto criado, **sobrepor o método actionPerformed(ActionEvent event)**.

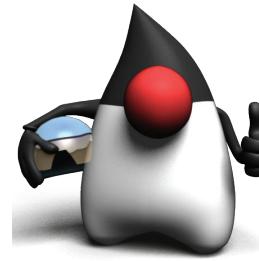
Este se encarregará de executar as instruções.

- > Descrever no método sobreescrito como o evento deve ser tratado.

# Classes Adaptadoras

Com a utilização de Classes Adaptadoras a classe que implementa o manipulador de um evento apenas herda da classe adaptadora e sobrescreve os métodos que precisar.

```
public class AcaoTecla extends KeyAdapter {
    public void keyPressed(KeyEvent e) {
        System.out.println("Tecla pressionada");
    }
    // Não é necessário declarar os outros métodos da
    // interface KeyListener (keyReleased e keyTyped)
}
```



## Java e Orientação a Objetos

Algumas interfaces listeners apresentam mais de um método. Entretanto, nem sempre o projetista da interface requer a utilização de todos os métodos. Para esses casos, há classes que fornecem implementações default para cada uma destas interfaces com múltiplos métodos, a implementação default é ignorar os eventos, definindo todos os métodos com um corpo vazio.

Essas classes adaptadoras são classes abstratas definidas no pacote de eventos AWT, com nome <nome>Adapter. Assim, para que a aplicação use uma classe adaptadora para tratar os eventos do tipo <nome>Event, uma classe que estende a classe <nome>Adapter deve ser definida; nessa classe derivada, os métodos relacionados aos eventos de interesse são redefinidos (por overriding). Como a classe adaptadora implementa a interface correspondente <nome>Listener, assim o fará a classe derivada. Os métodos não redefinidos herdarão a definição original, que simplesmente ignora os demais eventos.