

Introdução à linguagem Java	5
0 que é Java?	7
Um pouco de História	8
Onde encontro Java?	9
O que você pode fazer em Java?.....	9
Plataforma Java	10
Garbage Collection (Coletor de Lixo)	11
Divisão da Plataforma	12
Fases do Programa Java	15
Aplicações mais comuns em Java	17
Identificadores, palavras-chave e tipo	18
JavaDoc	19
Ponto-e-Vírgula, Blocos e Espaço	21
Identificadores e Palavras Reservadas	22
Variáveis, Declaração e Atribuição	24
Tipos de Dados	26
Casting de Tipos Primitivos	28
Classes Wrapper (Empacotadoras)	31
Construtores e método valueOf	32
Auto Boxing - Boxing and Unboxing	33
Operadores	34
Operadores Aritiméticos	35
Operadores Relacionais.....	36
Operadores Lógicos	37
&& (e lógico) e & (e binário)	38
(ou lógico) e (ou binário)	39
^ (ou exclusivo binário)	40
! (Negação)	41
Operadores de Incremento e Decremento	42
Precedência de Operadores	43
Operador Condicional (?:)	45
Estruturas de Controle	46
Estrutura de decisão (if)	47
Estrutura de decisão (if-else).....	48
Estrutura de decisão (if-else-if)	49
Estrutura de decisão (switch)	59
Estrutura de repetição (while)	51

Estrutura de repetição (do-while)	52
Estrutura de repetição (for)	54
Declaração break	55
Declaração continue	56
Array	57
Array?	58
Declarando Array	59
Acessando um elemento do Array	61
Arrays Multidimensionais	62
Acessando um elemento de um Array Multidimensional	63
Percorrendo Arrays com Enhanced-for	64
Manipulando Arrays com java.util.Arrays	65
Bases da programação Java 00	66
Pacotes	68
Modificadores de acesso	70
Classes	72
Definindo Classes	73
Métodos	74
Definindo Métodos	76
Objetos	78
Classe X Objeto	79
Notação UML	80
Notação UML Diagrama de Classe	81
Notação UML Relacionamentos	82
Herança	84
Agregação	85
Métodos, Construtores e Membros Estáticos	86
Declarando Membros: Variáveis e Métodos	87
Referência de Objetos	88
Invocação de Métodos	90
Passagem de Parâmetro por Valor	92
Passagem de Parâmetro por Referência	93
Sobrecarga de métodos (Overloading)	94
Construtores	95
Overloading de Construtores	97
Utilizando o Construtor this()	98
Instância de Classes	100

Membros estáticos	102
Membros estáticos	103
Herança e Polimorfismo	104
Herança	105
SuperClasse e SubClasse	107
Herança - classe Veiculo	108
Herança - classe Carro	109
Modificador de Classe final	110
Polimorfismo	111
Sobreposição de Métodos @Override	113
Referência polimórficas	115
Coleções Heterogêneas de Objetos	116
Determinando a Classe de um Objeto	117
Modificador de método final	119
Encapsulamento	120
Métodos de Configuração e Captura	122
Classes Abstratas, Internas e Interfaces	123
Classes Abstratas	124
Métodos Abstratos	125
Exemplos de implementação	126
Exemplos de implementação	127
Interfaces	128
Criando Interfaces	129
Implementando Interfaces	130
Herança entre interfaces	131
Interface vs. Classe	132
Métodos de Extensão.....	133
Classes Internas (Aninhadas)	135
Classes Internas Anônima.....	137
Java 8 - Lambda.....	138
Tipos Enumerados	139
Java Orientado a Objetos Exceções.....	141
Exceções	142
Categoria de Exceções	144
Exceções Verificadas e Não Verificadas	146
Manipulando Exceções	147
Um catch Múltiplas Exceções.....	149

Try-Com-Recursos.....	150
Throw e Throws	151
Criando Exceções	152
Sobrepondo Métodos e Exceções.....	153
Collections	154
Java Collections	155
Java Collections - Hierarquia	156
Interfaces Set e List	159
Interfaces Map.....	161
Generics e Coleções Java	162
Generics e Coleções Java	164
Interface Iterator	166
Percorrendo Collections	167
Metodo forEach.....	168
Classificando Coleções: Collections.sort	169
Interface Comparable	170
Collections e Streams	171
Lendo e Escrevendo Arquivos	173
Console I/O	174
Usando a classe Scanner	175
java.io.File	177
I/O Stream.....	179
Encadeando I/O Stream.....	180
FileWriter e BufferedWriter.....	181
FileReader e BufferedReader	182
NIO.2 - Path.....	183
NIO.2 - I/O Streams.....	184
NIO.2 - I/O Streams.....	185
NIO.2 - I/O Channel.....	184
Construindo interfaces gráficas com Swing	188
AWT versus Swing	189
Componentes AWT	191
Gerenciadores de layout	193
Componentes Swing	194
Containers JFrame.....	195
Manipulação de Evento.....	196
Classes de Evento.....	197
Criação de aplicações gráficas com Eventos.....	198

Apêndices

Classes Adaptadoras.....	199
Tipos Genéricos	201
Tipos Genéricos.....	142
Declarando uma Classe utilizando Generics	204
Limitação “Primitiva”	206
Limitando Genéricos	207
Coringa <?>	209
Threads	211
Ciclo de vida de uma Thread	213
Criando Threads	215
Iniciando Threads	217
Escalonamento da Thread	218
Prioridades de uma Thread	219
Sincronização	220
Bloqueando acesso Concorrente	222
Containers JFrame	191
Manipulação de Eventos	192
Classes de Evento	194
Criação de aplicações gráficas com Eventos	196
Classes Adaptadoras	197

Java Orientado a Objeto

Introdução à linguagem Java



Java Orientação a Objeto

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.

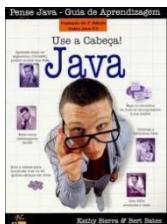
Dúvidas

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

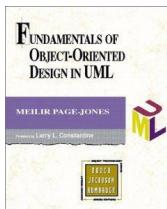
Bibliografia extra:



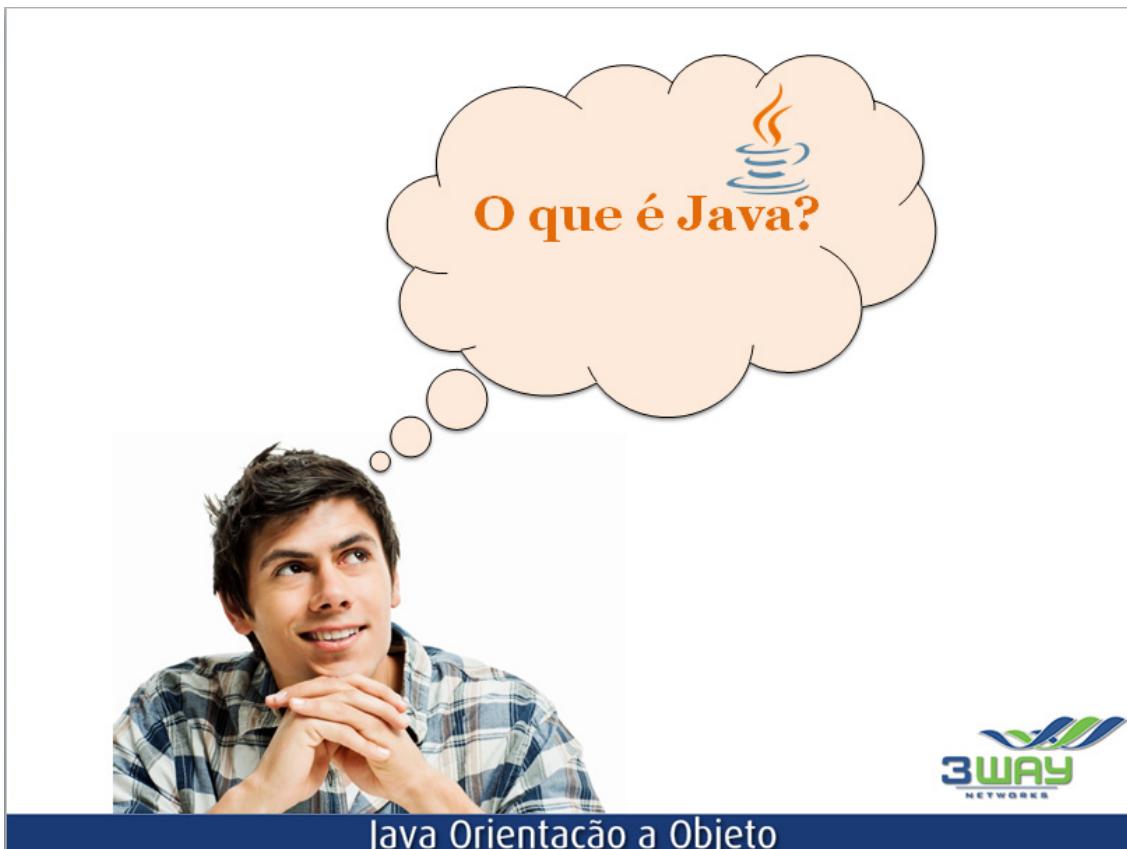
- Java - Como programar, de Harvey M. Deitel;



- Use a cabeça! - Java, de Bert Bates e Kathy Sierra;



- Fundamentals of Object-Oriented Design in UML;



O que é Java?

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, por exemplo:

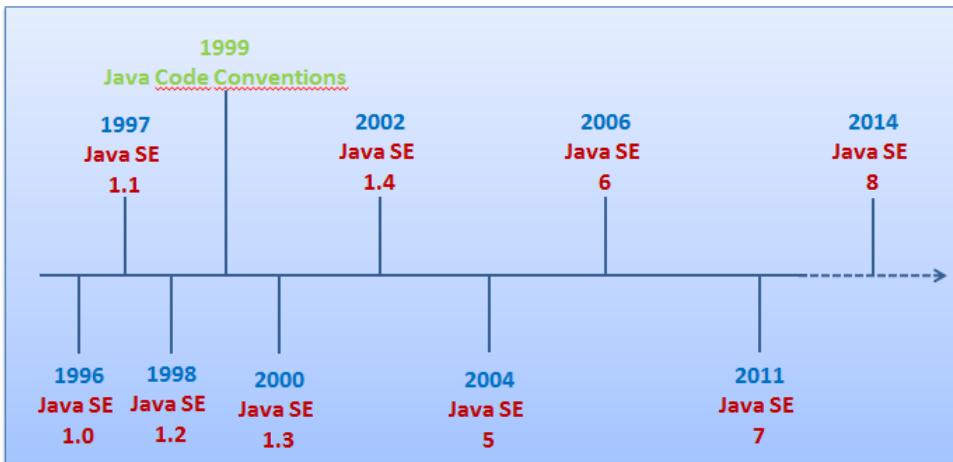
- Java é uma linguagem simples, existem poucas regras, muito bem definidas, **orientada a objetos**, fácil de aprender, e que possui como maior vantagem ser **portável** entre as diversas plataformas existentes.

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 Orientação a Objeto

Um pouco de história

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, a 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 Orientação a Objeto

Onde você encontra Java?

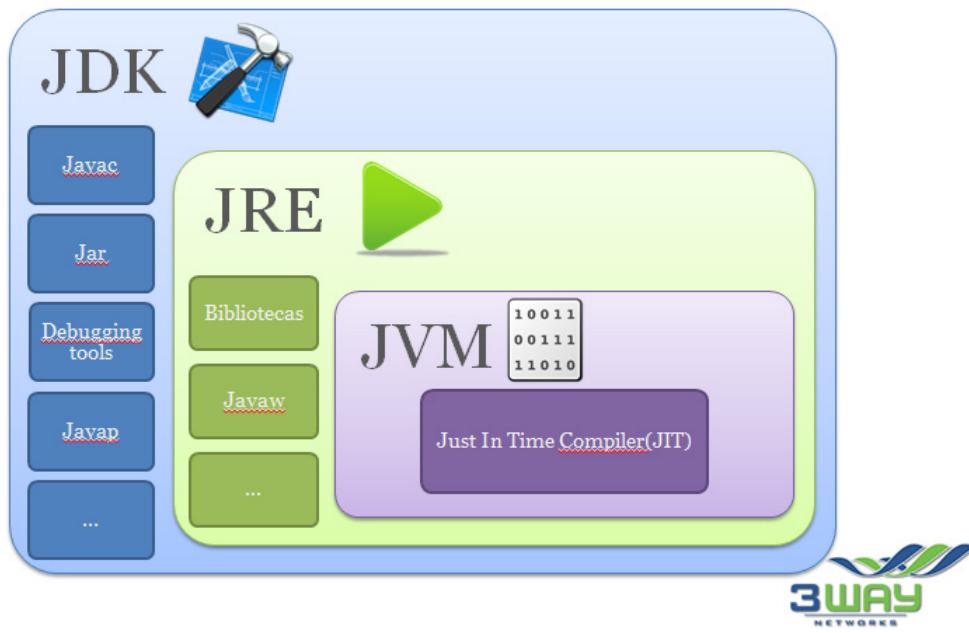
O Java é executado em mais de 850 milhões de computadores pessoais e em bilhões de dispositivos em todo o mundo. A tecnologia Java está em todo lugar! Ela pode ser encontrada inclusive em dispositivos de televisão, laptops, datacenters, consoles de jogo, supercomputadores científicos, telefones celulares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

Plataforma Java

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)

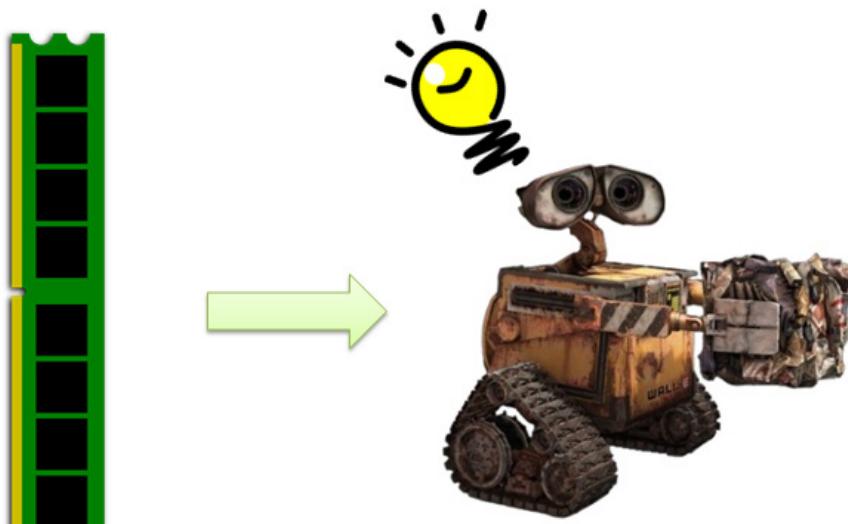
É composto pela JVM e pela biblioteca de classes Java utilizada para execução de aplicações Java, estas bibliotecas são chamadas de APIs Java. Portanto para rodar uma aplicação Java é necessário que você instale uma JRE e computador onde o software foi instalado.

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 de o 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 multiplataforma. Uma JVM pode ser desenvolvida por qualquer organização (comunidades / institutos / empresas), desde que sigam as especificações para a Java Virtual Machine.

Garbage Collector (Colecionador de lixo)



Java Orientação a Objeto

Garbage Collector (Colecionador 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 objetos de um programa que não serão mais acessados no futuro e deslocar os recursos utilizados por tais objetos 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 estando sendo referenciados há algum tempo é liberada. Porém o Garbage Collector funciona aleatoriamente

Divisão da Plataforma



Java Orientação a Objeto

Divisão da plataforma

- **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:

- o JDBC (Java Database Connectivity), utilizado no acesso a banco de dados;

- o JSP (Java Server Pages), um tipo de servidor Web. Grossamente falando, servidores Web são as aplicações que permitem a você acessar um site na internet;
- o 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, por exemplo. Um 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.

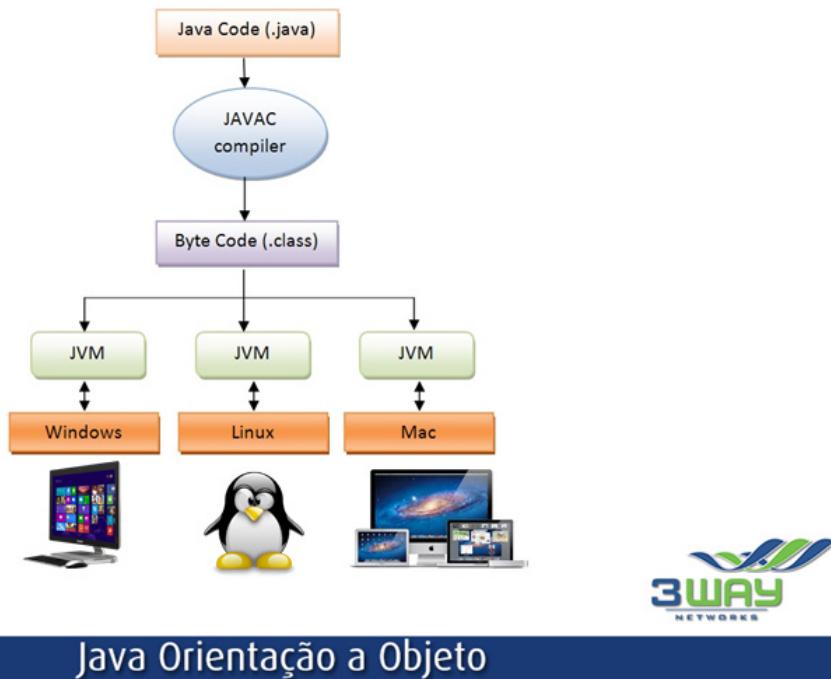
• **WebServices**

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 Orientação a Objeto

Fases do programa Java

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 edi-

tor de texto, é salvo com a extensão “*.java*”, por exemplo, “*Programa.java*”.

2. **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.
3. **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 Orientação a Objeto

Aplicações mais comuns em Java

- **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:

Java Orientado a Objetos

Identificadores, palavras-chave e tipo



Java Orientação a Objeto

Descrição:

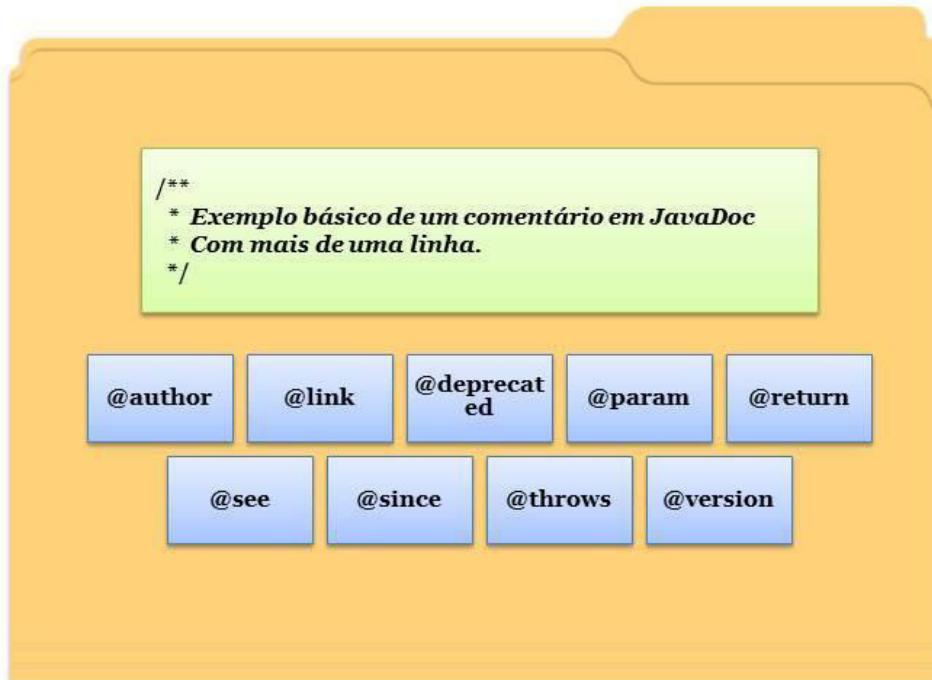
Objetivos Específicos:

Identificadores, palavras-chave e tipo.

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 Orientação a Objeto

JavaDoc

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.

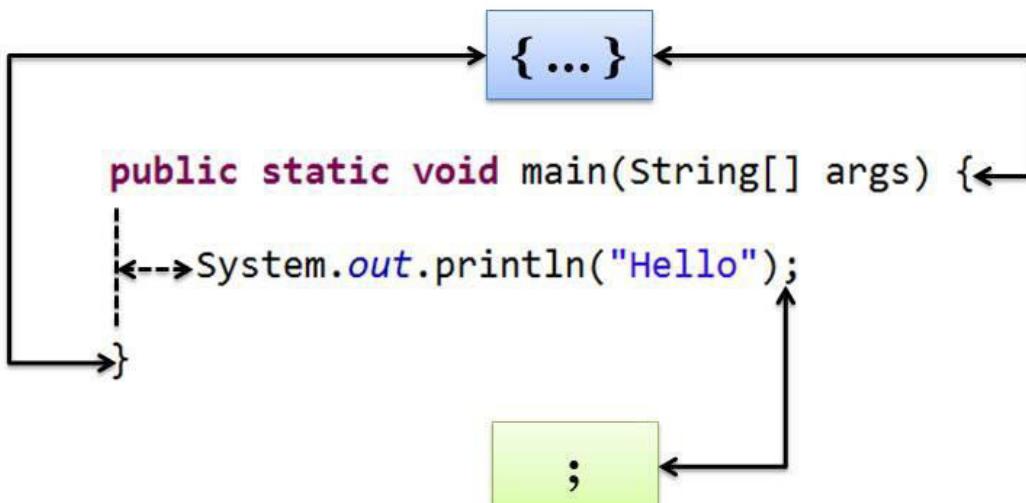
Para fazer um comentário em Java, você pode usar o // para comentar uma linha simples até o final, tudo após o // será ignorado pelo compilador, ou então usar o /* */ para comentar múltiplas linhas, tudo o que estiver entre eles será ignorado pelo compilador.

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 JavaDoc

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

Ponto-e-Vírgula, Blocos e Espaço



Java Orientação a Objeto

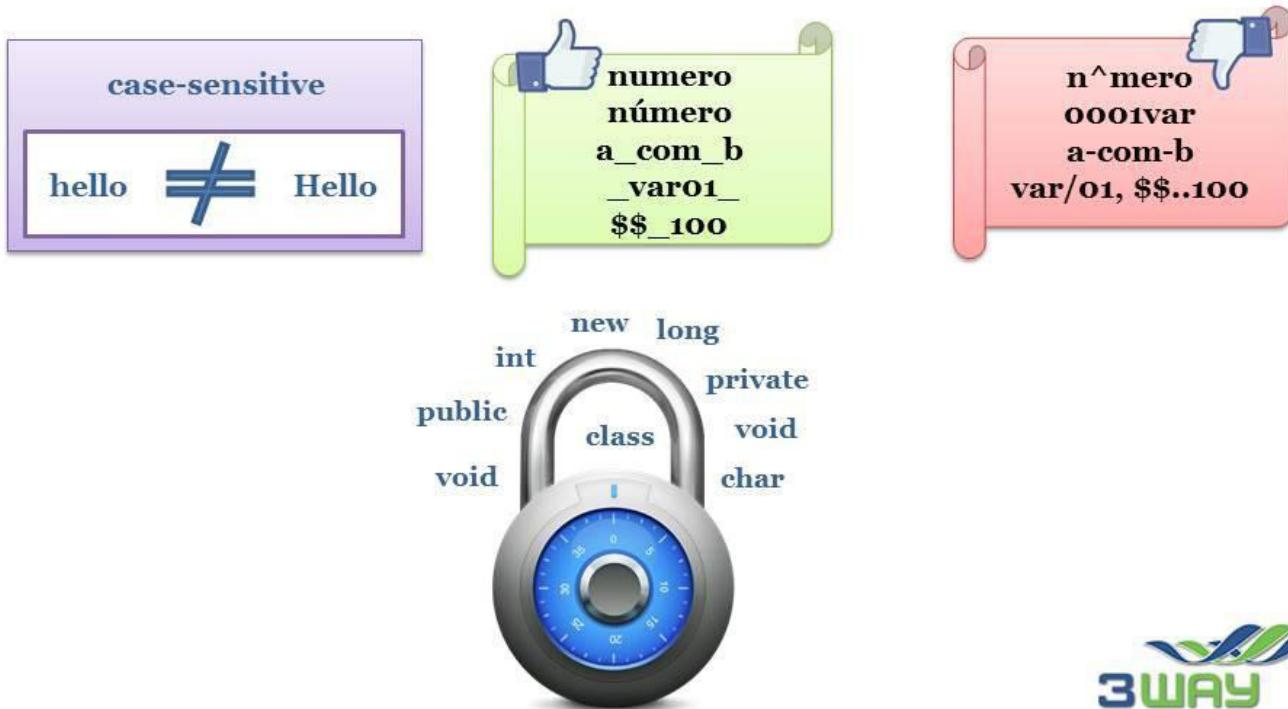
Ponto-e-vírgula, Blocos e Espaço.

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 Orientação a Objeto

Identificadores e Palavras Reservadas.

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:

1. Não pode ser uma palavra-reservada (palavra-chave);
2. Não pode ser **true** nem **false** - literais que representam os tipos lógicos (booleanos);
3. Não pode ser **null** - literal que representa o tipo nulo;
4. Não pode conter espaços em brancos ou outros caracteres de formatação;
5. 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.

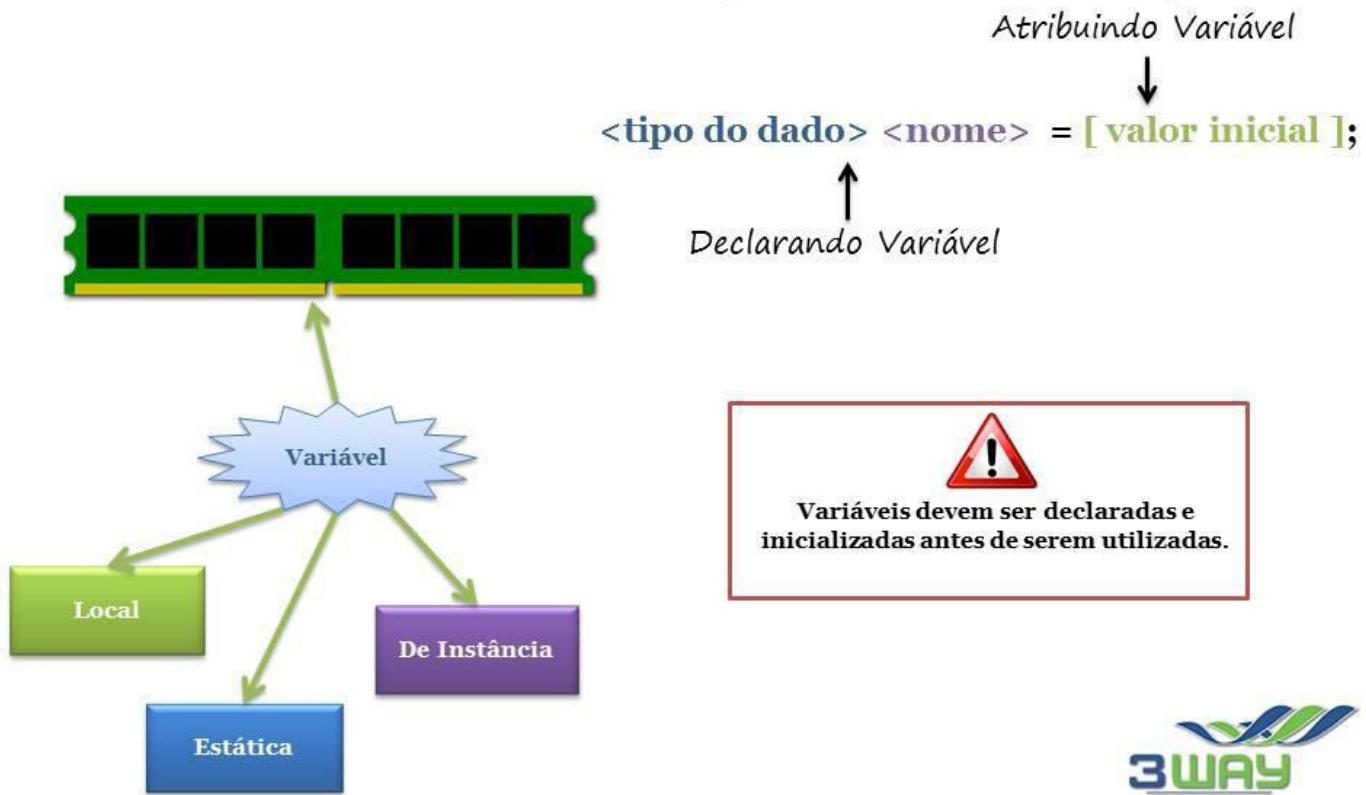
Observação 02: Java é **case-sensitive**, ou seja, letras maiúsculas e minúsculas diferenciam os identificadores, ou seja, **a** é um identificador diferente de **A**, **História** é diferente de **história**, etc.

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*.

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

Variáveis, Declaração e Atribuição



Java Orientação a Objeto

Variáveis, Declaração e Atribuição.

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

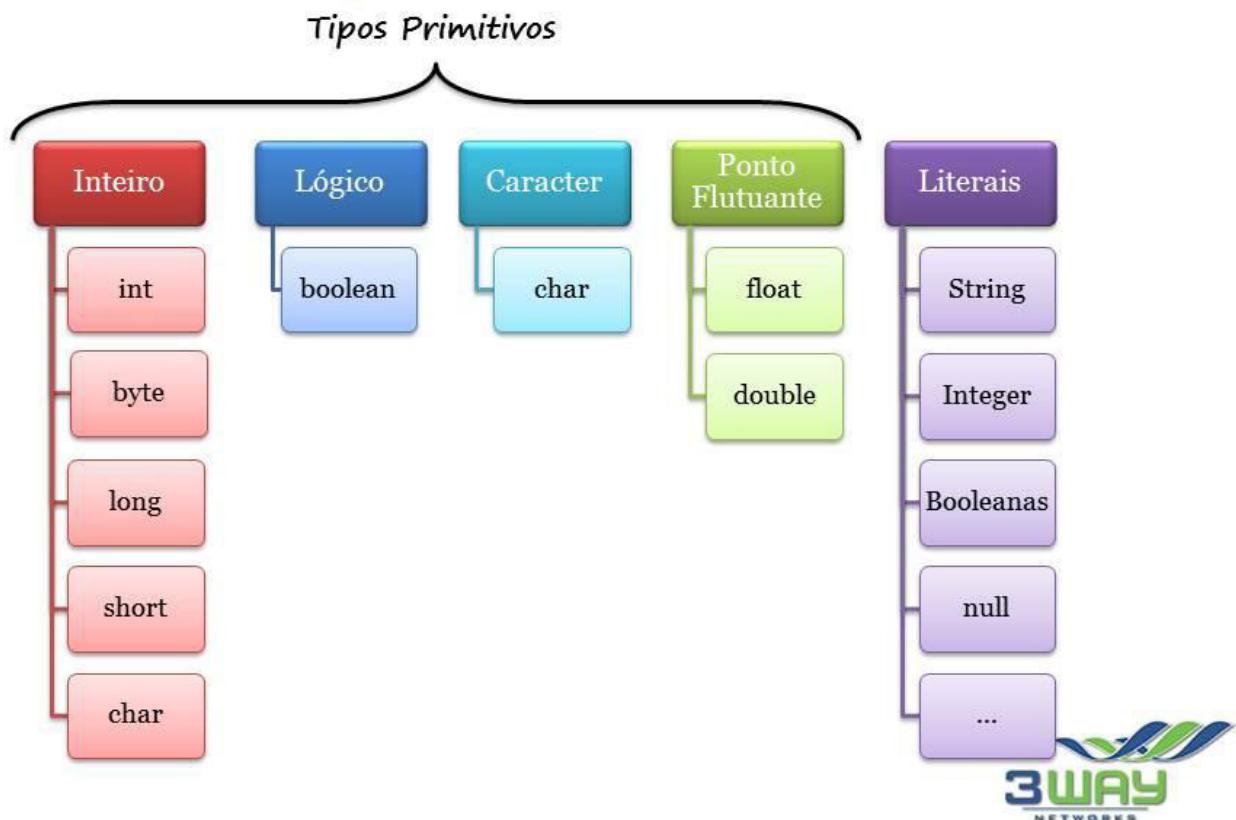
sua utilização e o valor da variável.

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. 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 Orientação a Objeto

TIPOS DE DADOS.

- 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 **CARACTER ASCII** DE **8 BITS**. **UNICODE** PERMITE O USO DE SÍMBOLOS E CARACTERES ESPECIAIS DAS LÍNGUAS.

PARA USAR UM LITERAL DE CARACTER 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

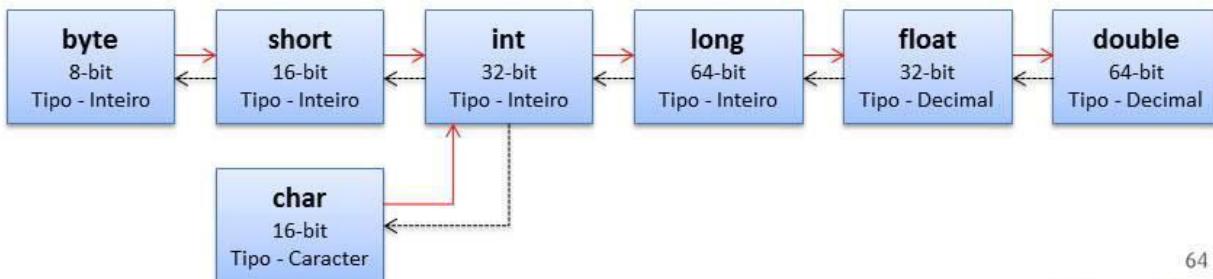
EM JAVA, EXISTEM DUAS CATEGORIAS DE VARIÁVEIS DE PONTO FLUTUANTE: **FLOAT** - ARMAZENA VALORES NUMÉRICOS EM PONTO FLUTUANTE DE PRECISÃO SIMPLES E **DOUBLE** - DE PRECISÃO DUPLA. AMBOS SEGUEM A NORMA: *IEEE Standard for Binary Floating Point Arithmetic*, ANSI/IEEE Std. 754-1985 (IEEE, NEW YORK). O FATO 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).

		<i>Valores possíveis</i>				
<i>Tipos</i>	<i>Primitivo</i>	<i>Menor</i>	<i>Maior</i>	<i>Valor Padrão</i>	<i>Tamanho</i>	<i>Exemplo</i>
Inteiro	<u>byte</u>	-128	127	0	8 bits	byte ex1 = (byte)1;
	<u>short</u>	-32768	32767	0	16 bits	short ex2 = (short)1;
	<u>int</u>	-2.147.483.648	2.147.483.647	0	32 bits	<u>int</u> ex3 = 1;
	<u>long</u>	-9.223.372.036.854.770.000	9.223.372.036.854.770.000	0	64 bits	<u>long</u> ex4 = 1L;
Ponto Flutuante	<u>float</u>	-1,4024E-37	3.40282347E + 38	0	32 bits	<u>float</u> ex5 = 5.5f;
	<u>double</u>	-4,94E-307	1.79769313486231570E + 308	0	64 bits	<u>double</u> ex6 = 10.20d; ou <u>double</u> ex6 = 10.20;
Caractere	<u>char</u>	0	65535	\0	16 bits	<u>char</u> ex7 = 194; ou <u>char</u> ex8 = 'a';
Booleano	<u>boolean</u>	<u>false</u>	<u>true</u>	<u>false</u>	1 bit	<u>boolean</u> ex9 = true;

Casting de Tipos Primitivos



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



Java Orientação a Objeto

CASTING DE TIPOS PRIMITIVOS

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**.

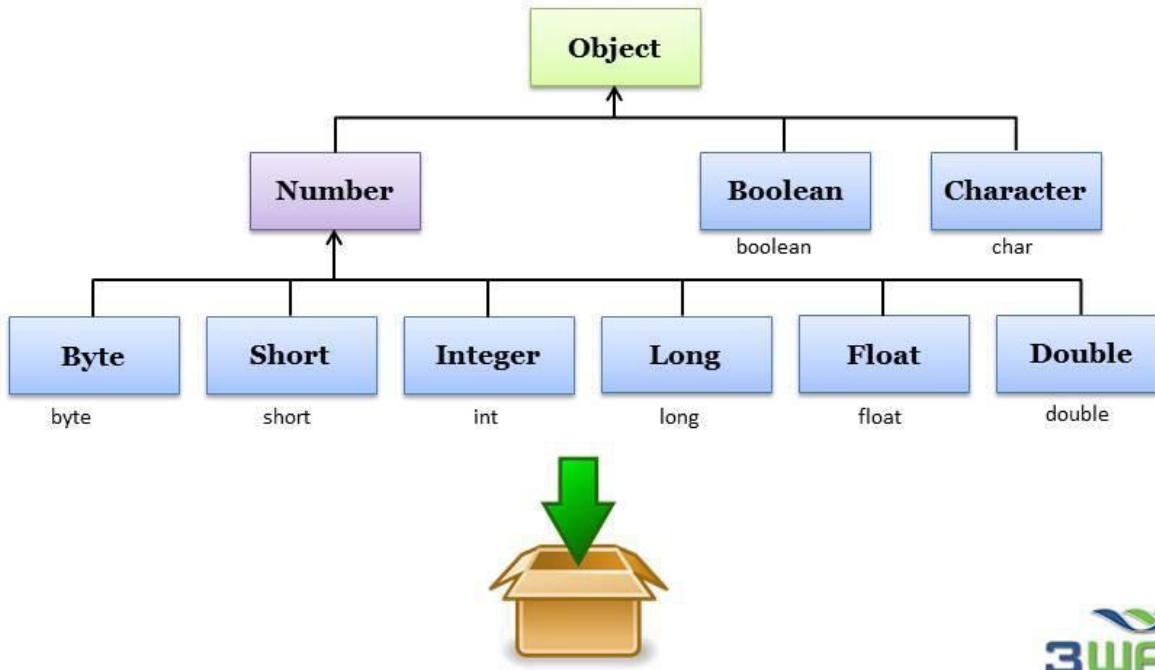
É POSSÍVEL FAZER CONVERSÕES DE TIPOS DE PONTO FLUTUANTE PARA INTEIROS, E INCLUSIVE ENTRE O TIPO CARACTERE, PORÉM ESTAS CONVERSÕES PODEM OCASIONAR A PERDA DE VALORES, QUANDO SE MOLDA UM TIPO DE MAIOR TAMANHO, COMO UM **DOUBLE** DENTRO DE UM **INT**.

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:

<i>double</i>	<i>float</i>	<i>long</i>	<i>int</i>	<i>char</i>	<i>short</i>	<i>byte</i>	<i>DE \ PARA</i>
Implícito	Implícito	Implícito	Implícito	char	Implícito		<i>byte</i>
Implícito	Implícito	Implícito	Implícito	char		byte	<i>short</i>
Implícito	Implícito	Implícito	Implícito		short	byte	<i>char</i>
Implícito	Implícito	Implícito		char	short	byte	<i>int</i>
Implícito	Implícito		int	char	short	byte	<i>long</i>
Implícito		long	int	char	short	byte	<i>float</i>
	float	long	int	char	short	byte	<i>double</i>

Classes Wrapper (Empacotadoras)



Java Orientação a Objeto

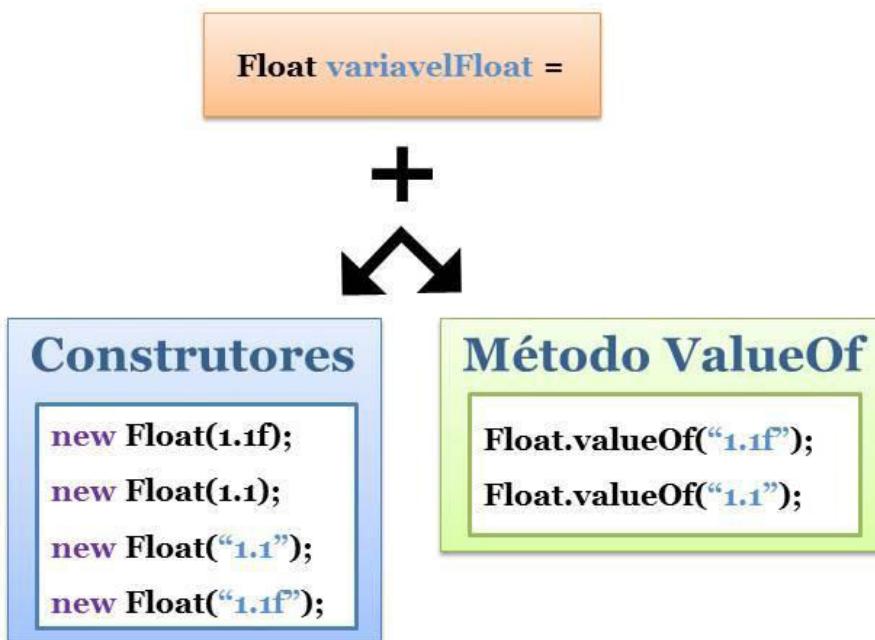
CLASSES WRAPPER (EMPACOTADORAS)

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 Orientação a Objeto

CONSTRUTORES WRAPPER

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 CORRESPONDENTES 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 Orientação a Objeto

AUTOBOXING – BOXING AND UNBOXING

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 Orientado a Objetos Operadores



Java Orientação a Objeto

Operadores

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 Orientação a Objeto

Operadores Aritméticos

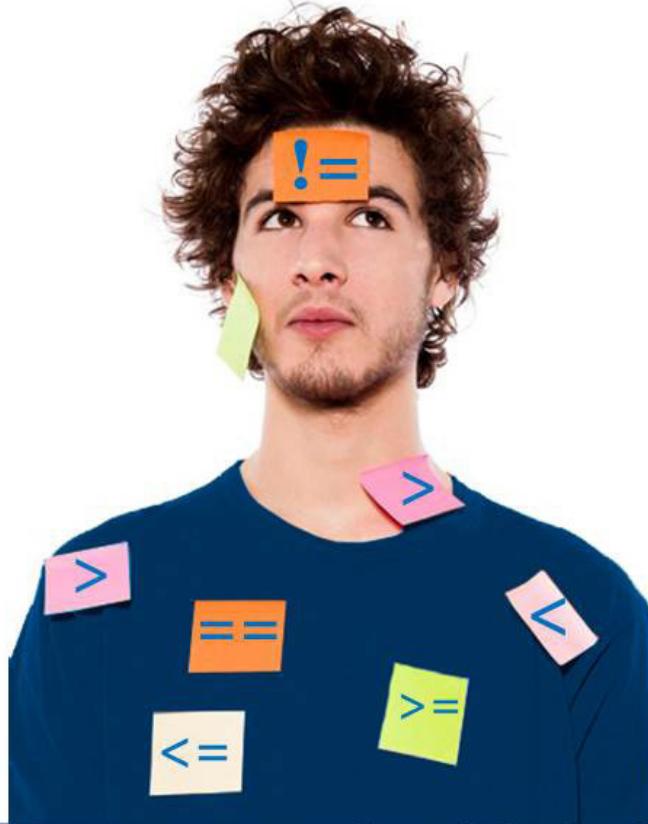
Os operadores aritméticos são **operadores binários**, ou seja, funcionam com dois operandos. Por exemplo, a expressão “**a + 1**” contém o operador binário “**+** (mais)” e os dois operandos “**a**” e “**1**”.

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



	var1 > var2	Maior que
	var1 >= var2	Maior ou igual
	var1 < var2	Menor que
	var1 <= var2	Menor ou igual
	var1 == var2	Igual
	var1 != var2	Diferente



Java Orientação a Objeto

Operadores Relacionais

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

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

Operador	Em Java	Sentença	Exemplo 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
<=	<=	Menor ou igual a	5 <= 4	Falso
≠	!=	Diferente de	7 != 9	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 Orientação a Objeto

Operadores Lógicos

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:

Operador	Descrição	Exemplo
&&	Retorna true se ambos operandos forem true.	true && true
&	Retorna true se ambos operandos forem true, avaliando ambos.	true & false
 	Retorna true se algum dos operandos for true.	false true
 	Retorna true se um dos operandos for true, avaliando ambos.	true true
!	Nega o operando que se passa.	! 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 Orientação a Objeto

&& (e lógico) e & (e binário)

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 Orientação a Objeto

|| (ou lógico) e | (ou binário)

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 Orientação a Objeto

\wedge (ou exclusivo binário)

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

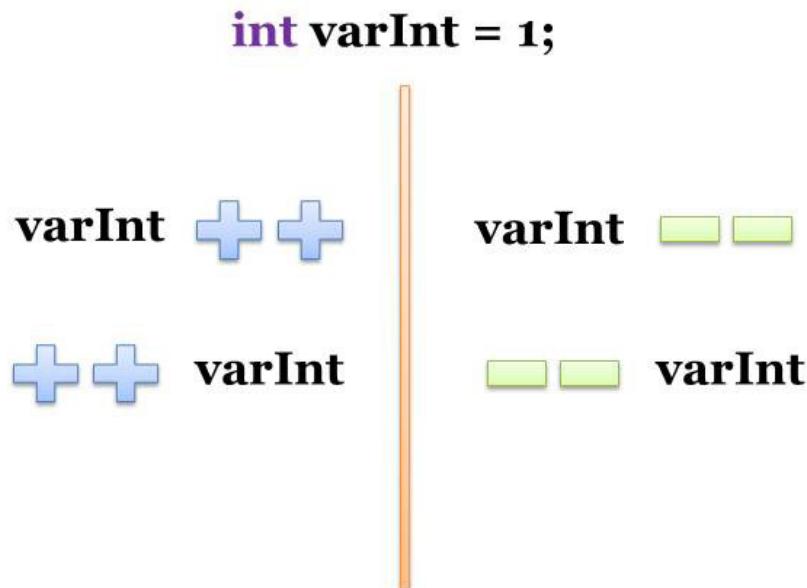


Java Orientação a Objeto

! (Negação)

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 Orientação a Objeto

Operadores de Incremento e Decremento

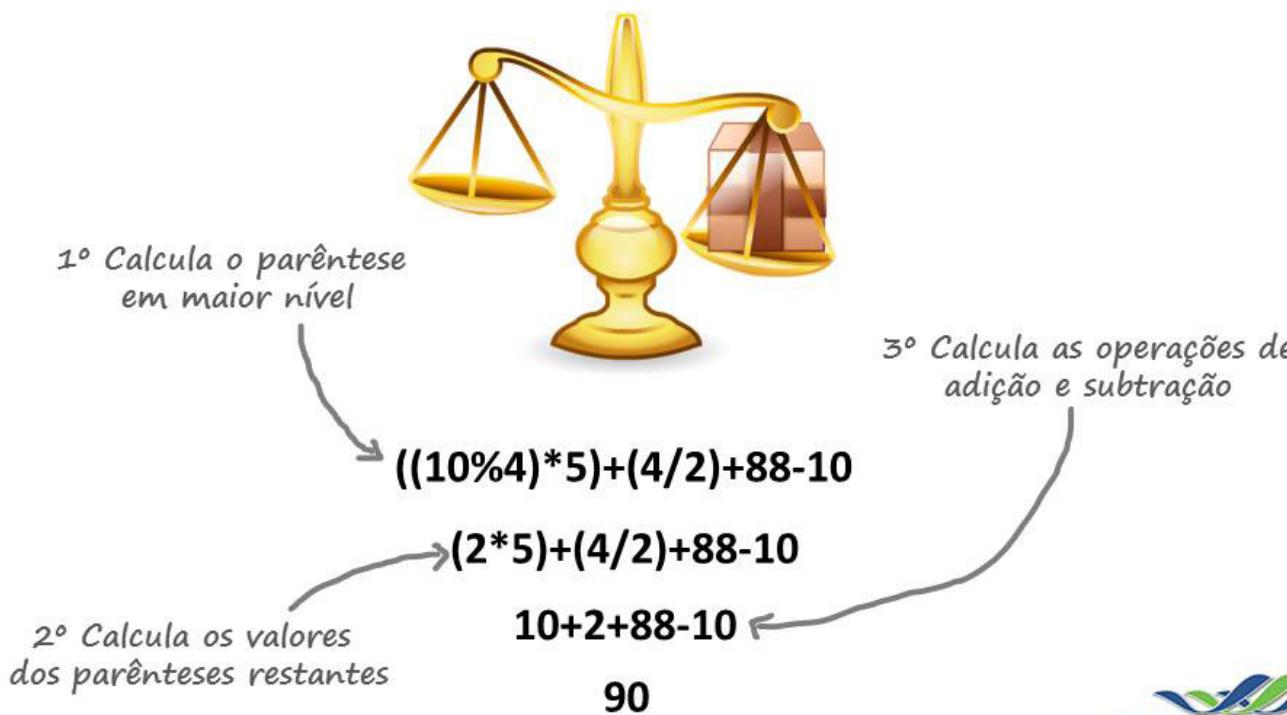
Além dos operadores aritméticos básicos, Java dá suporte ao operador unário de **incremento (++)** e ao operador unário de **decremento (--)**. Operadores ++ ou --, respectivamente aumentam ou diminuem em 1 o valor da variável.

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 Orientação a Objeto

Precedência de Operadores

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).

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

12	<code> </code>	Operador lógico ou condicional
13	<code>? :</code>	Condisional: if-then-else compacto
14	<code>= op =</code>	Atribuição (Combinada)

Considere a expressão:

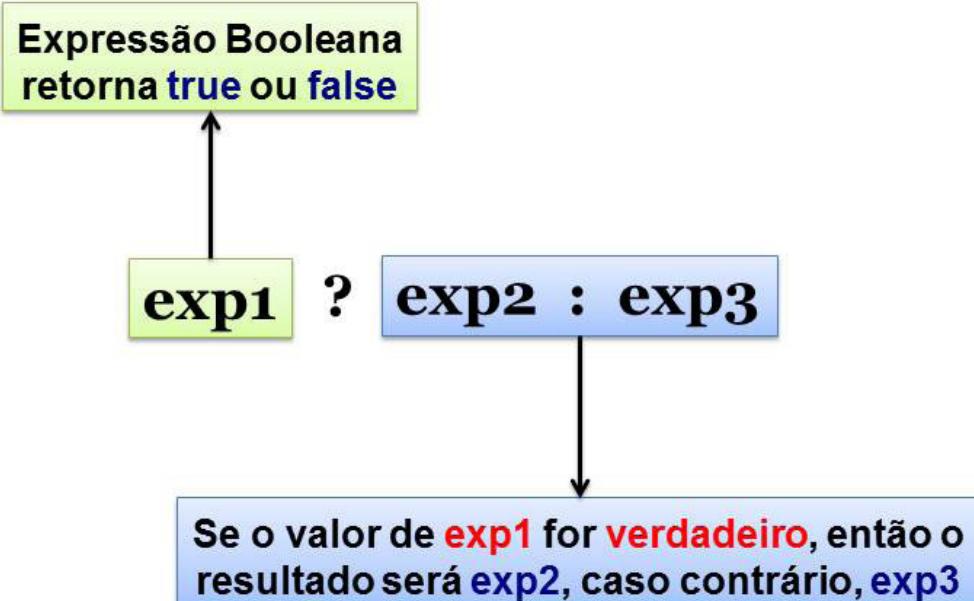
`1 | 2 ^ 3 * 2 & 13 | 2`

1. A análise é feita da esquerda para a direita.
2. O primeiro cálculo a ser executado é **`3 * 2`**, que resulta em **6**.
3. Em seguida, o resultado anterior é comparado com o **13**, uma operação **& (E)**: **6 & 13**, que resulta em **4**.
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**.
5. É executada, então, a operação **| (Ou)** mais a esquerda: **`1 | 6`** que resulta em **7**.
6. 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 (?:)



Java Orientação a Objeto

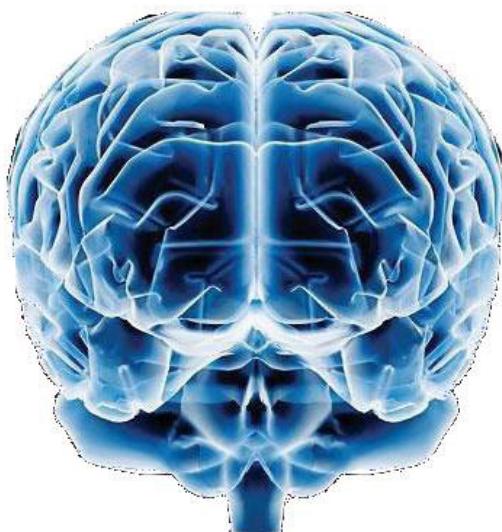
Operador Condicional (?:)

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 Orientado a Objetos

Estruturas de Controle



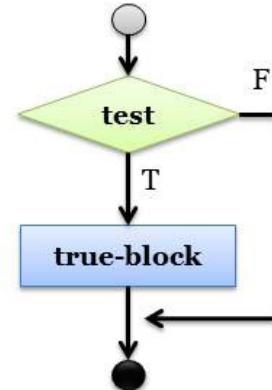
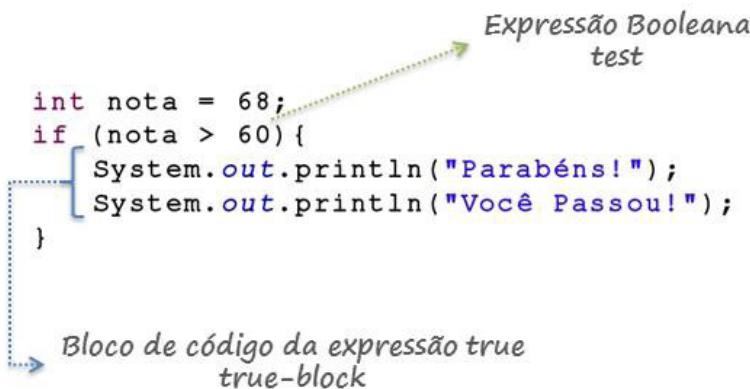
Java Orientado a Objeto

Estruturas de Controle

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)



Java Orientado a Objeto

Estrutura de decisão (if)

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:

ex 1:

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

ex 2:

```
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)

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:

ex 1:

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

ex 2:

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

Estrutura de decisão (if-else-if)

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

```
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)

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** (até Java 5), ele também pode operar sobre tipos **enumerados** (a serem vistos posteriormente) sobre classes empacotadoras **Character**, **Byte**, **Short** e **Integer** (a partir de Java 6) e da classe **String** (a partir de Java 7).

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

ex1:

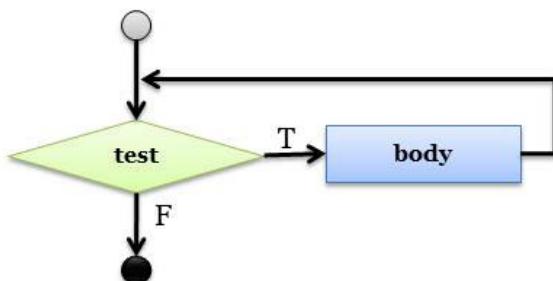
```
switch (<variável_do_tipo_permitido>) {  
    case <valor 1>:  
        [instrução1;  
        [...]  
        [break];]  
    case <valor n>:  
        [instrução1;  
        [...]  
        [break];]  
    default:  
        [instrução1;  
        [...]  
        [break];]  
}
```

ex 2:

```
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 Orientado a Objeto

Estrutura de repetição (while)

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.

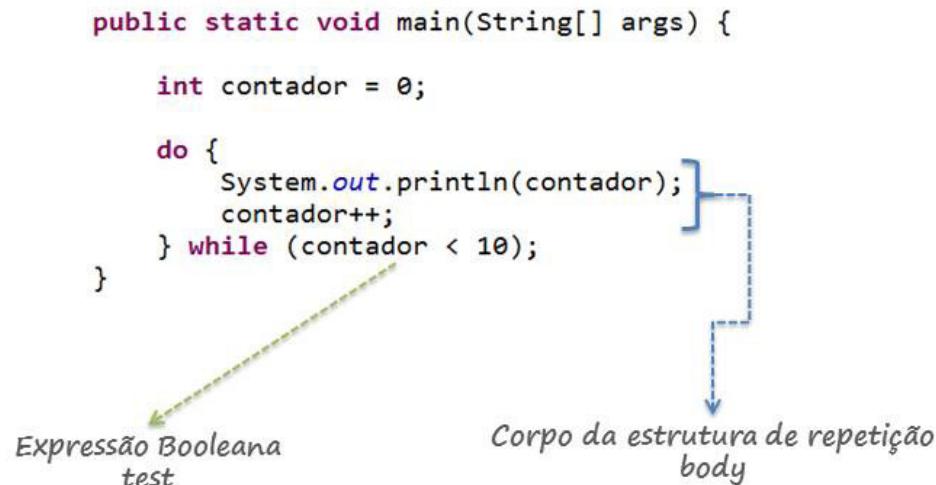
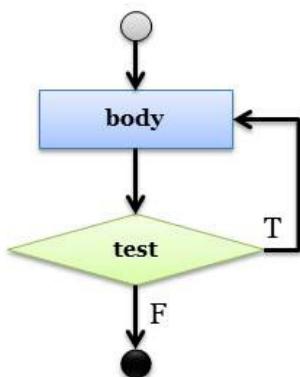
A declaração **while** possui a seguinte forma:

```

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

}

Estrutura de repetição (do-while)



Java Orientado a Objeto

Estrutura de repetição (do-while)

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:

```

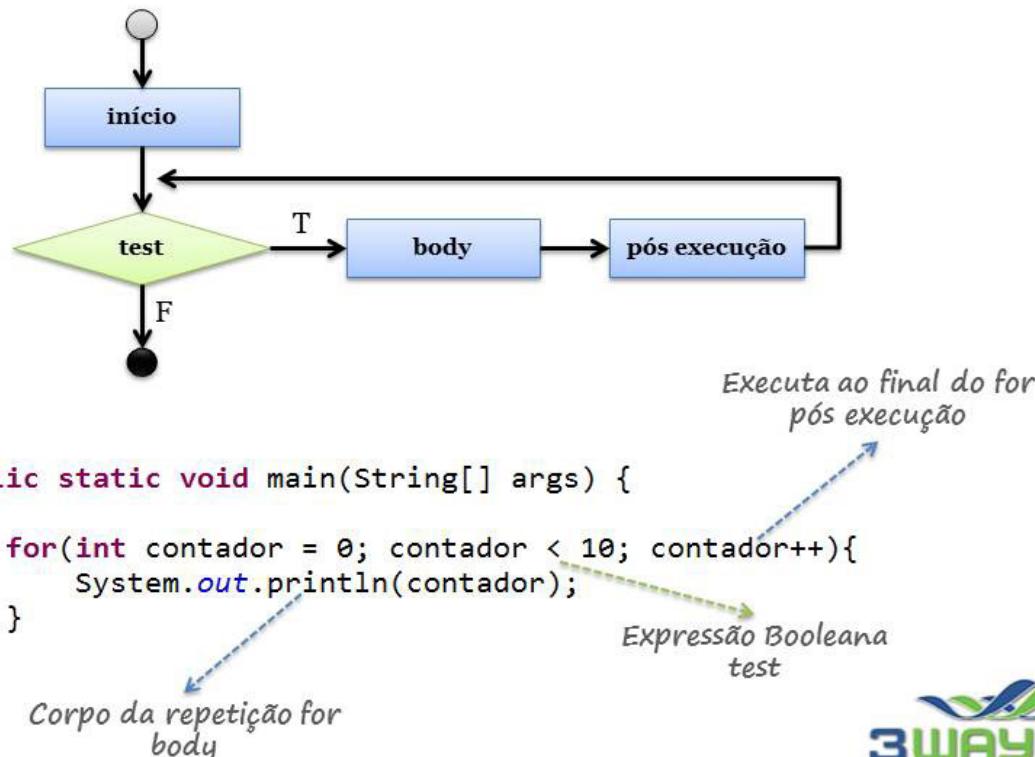
do {
  instrução1;
  instrução2; [...]
} while (expressão_lógica);
  
```

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 Orientado a Objeto

Estrutura de repetição (for)

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:

```

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 Orientado a Objeto

Declaração break

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).

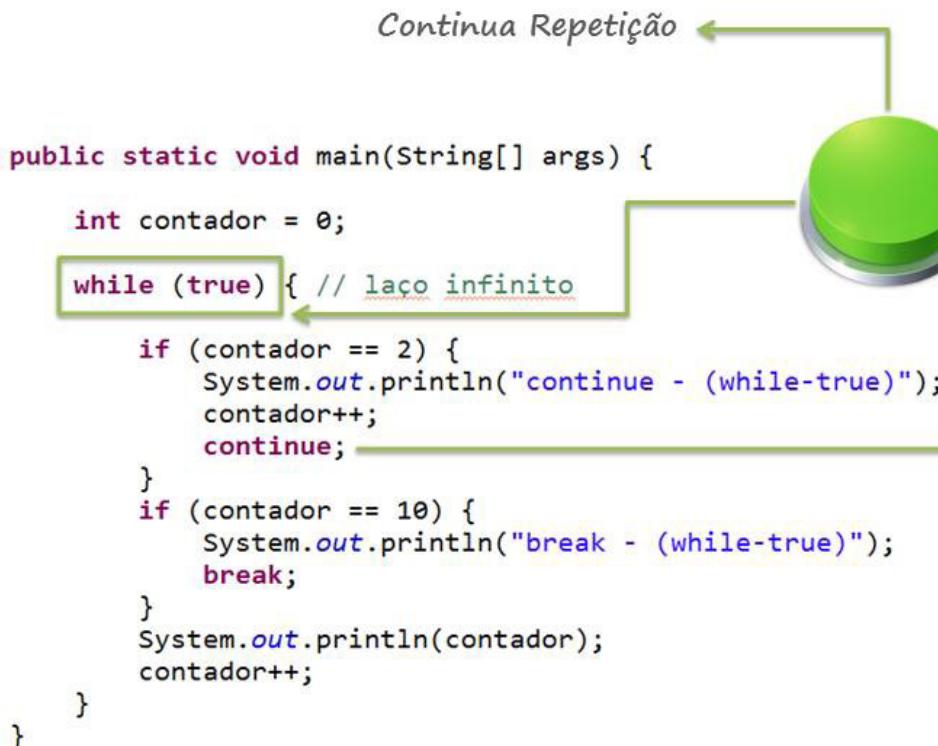
Declaração unlabeled break

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, como por exemplo: ONDE ESTÁ O EXEMPLO?

Declaração continue



Java Orientado a Objeto

Declaração continue

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.

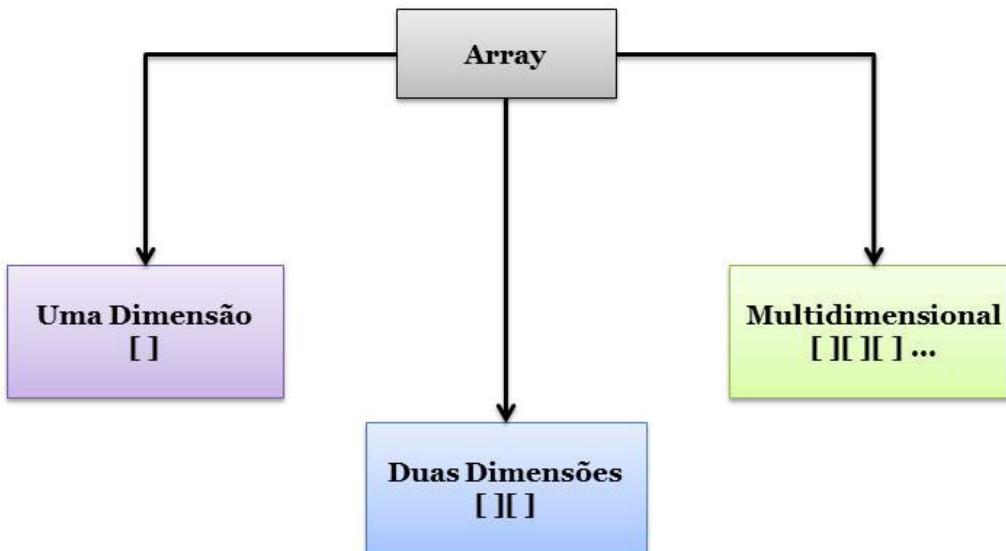
Java Orientado a Objetos

Array



Java Orientado a Objeto

Array ?



Java Orientado a Objeto

Array

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 Orientado a Objeto

Declarando Array

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.

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

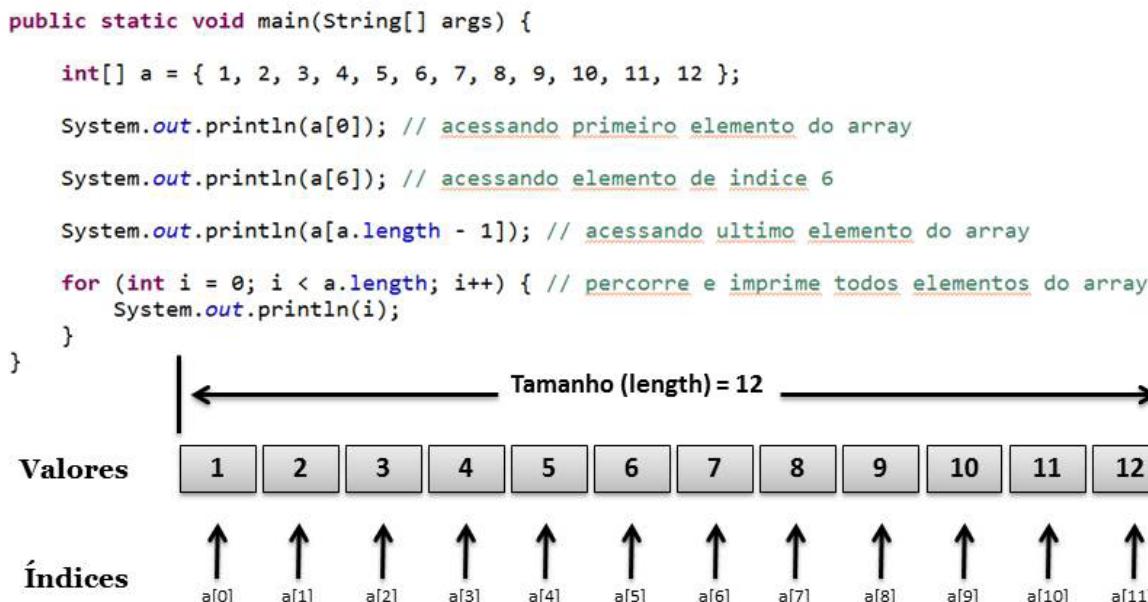
No exemplo, a declaração diz ao compilador Java que o identificador **idades** será usado como um nome de um **array** contendo valores do tipo **int**, e que o novo **array** contém

100 elementos.

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.

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

Acessando um elemento do Array



Java Orientado a Objeto

Acessando um elemento do Array

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

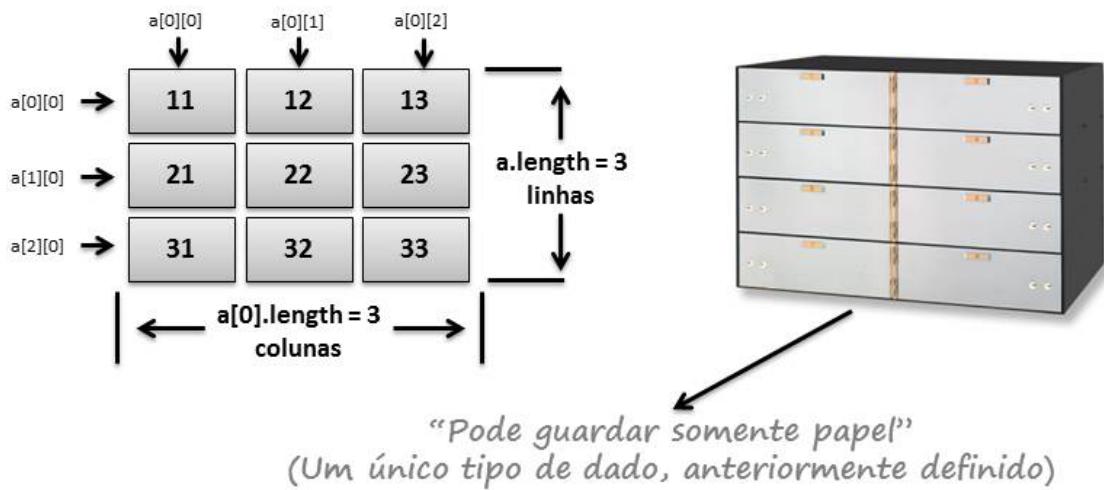
O atributo **length** de um **array** retorna seu tamanho, ou seja, a quantidade de elementos, ou seja, ele é uma constante já que o **array** depois de criado não sofrerá alteração de tamanho.

Arrays Multidimensionais

Inicialização

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

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



Java Orientado a Objeto

Arrays Multidimensionais

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.

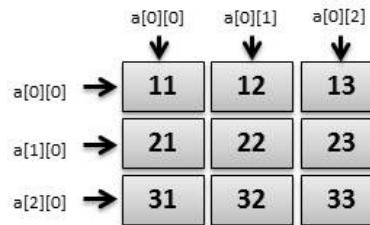
É possível definir n dimensões, porém, na prática, não é comum nem recomendável trabalhar com tantas dimensões. Na prática, é extremamente comum trabalharmos com apenas 1 (uma), e algumas vezes com 2 (duas), e, quase nunca com 3 (três) dimensões, pois há outros recursos na programação orientada a objetos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

Acessando um elemento de um Array Multidimensional

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 Orientado a Objeto

Percorrendo Arrays com Enhanced-for

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.

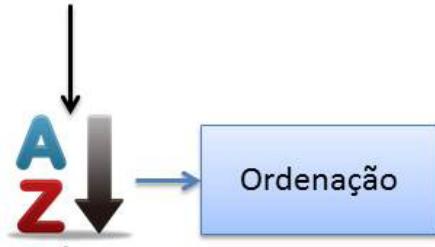
Ponto importante: Deve ser usado Array e conjuntos do mesmo **tipo da declaração (int, double, Object)**, caso contrário, não compila.

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



```
public static void main(String[] args) {
    int[] a = { 8, 4, 1, 9, 2, 5, 12, 3, 6, 10, 7, 11 };
    System.out.println(Arrays.toString(a));
    Arrays.sort(a);
    System.out.println(Arrays.toString(a));
    System.out.println("a[" + Arrays.binarySearch(a, 6) + "]");
}
```



Busca Binaria



Java Orientado a Objeto

Manipulando Arrays com `java.util.Arrays`

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 Orientado a Objetos

Bases da programação Java OO



Java Orientado a Objeto

Bases da programação Java OO

Durante anos, os programadores se dedicaram a construir aplicações muito parecidas que resolviam uma vez ou outra, os mesmo 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 com que outras pessoas possam 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.

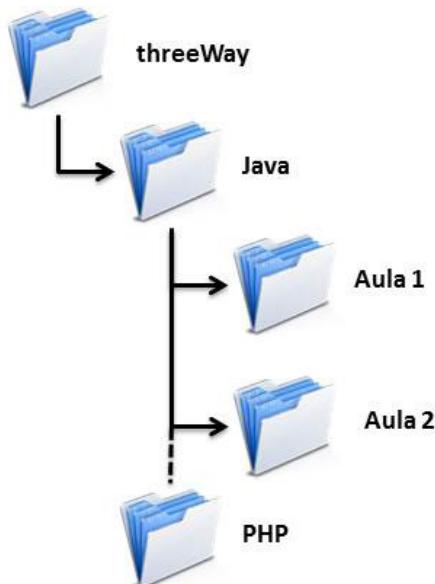
Então em um esquema POO o carro seria o objeto, as propriedades seriam as características como a cor ou o modelo e os métodos seriam as funcionalidades associadas como andar ou parar.

Vou dar outro exemplo, vamos ver como faríamos um modelo em um esquema POO de uma fração, ou seja, essa estrutura matemática que tem um numerador e um denominador que divide ao numerador, por exemplo, $3/2$.

A fração será o objeto e terá duas propriedades, o numerador e o denominador. Logo, poderia ter vários métodos como simplificar, somar com outra fração ou número, subtrair com outra fração, etc.

Estes objetos poderão ser utilizados nos programas, por exemplo, em um programa de matemáticas seria feito o uso de objetos fração e em um programa que providencie uma oficina de carros, seria utilizado o uso de objeto carro. Os programas Orientados a objetos utilizam muitos objetos para realizar as ações que se desejam realizar e eles mesmos também são objetos. Ou seja, a oficina de carros será um objeto que utilizará objetos carro, ferramenta, mecânico, trocas, etc.

Pacotes



```

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

```

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



Java Orientado a Objeto

Pacotes

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.

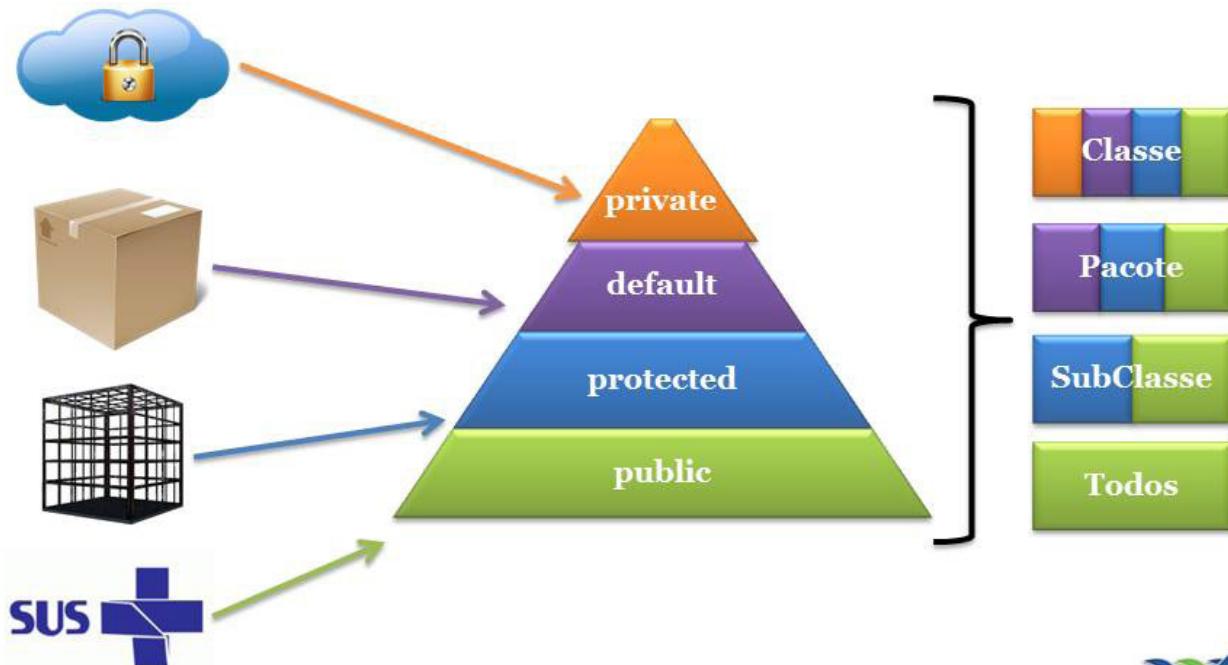
Ter que usar o **nome completamente qualificado** de uma classe não é muito agradável, além de tornar as linhas de nosso código-fonte bem grandes, tornando sua leitura mais difícil, o que não é nada bom para manutenção de um programa.

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 Orientado a Objeto

Modificadores de acesso

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.

Tabela dos modificadores de acesso

	private	default	protected	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 estende-la, nos métodos impede que o mesmo seja sobrescrito (overriding) na subclasse, e nos valores de variáveis não pode ser alterado depois que já tenha sido atribuído um valor.

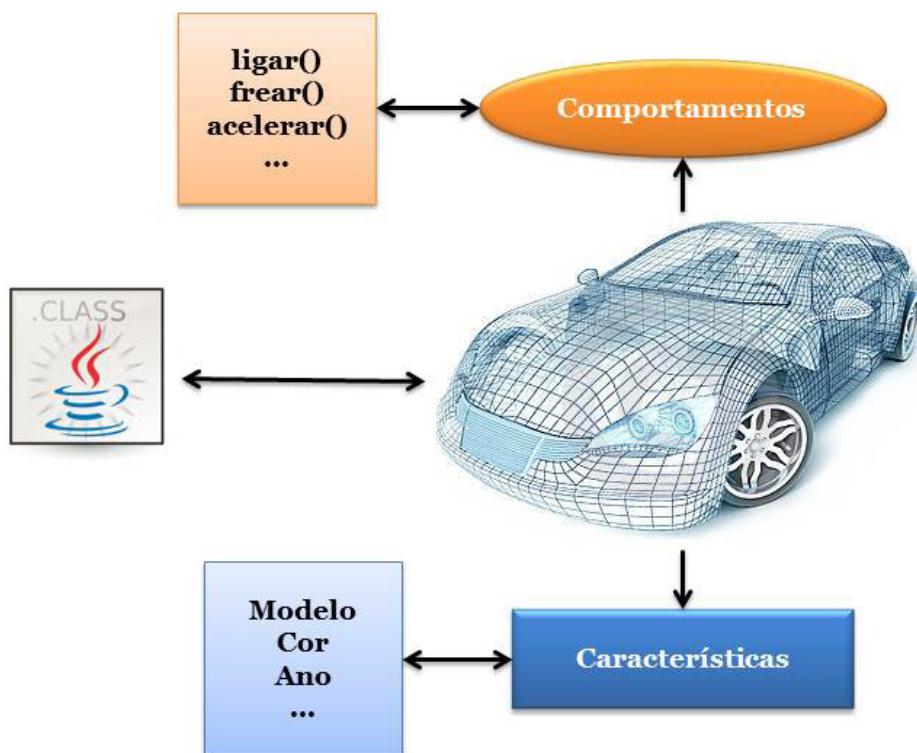
abstract

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 Orientado a Objeto

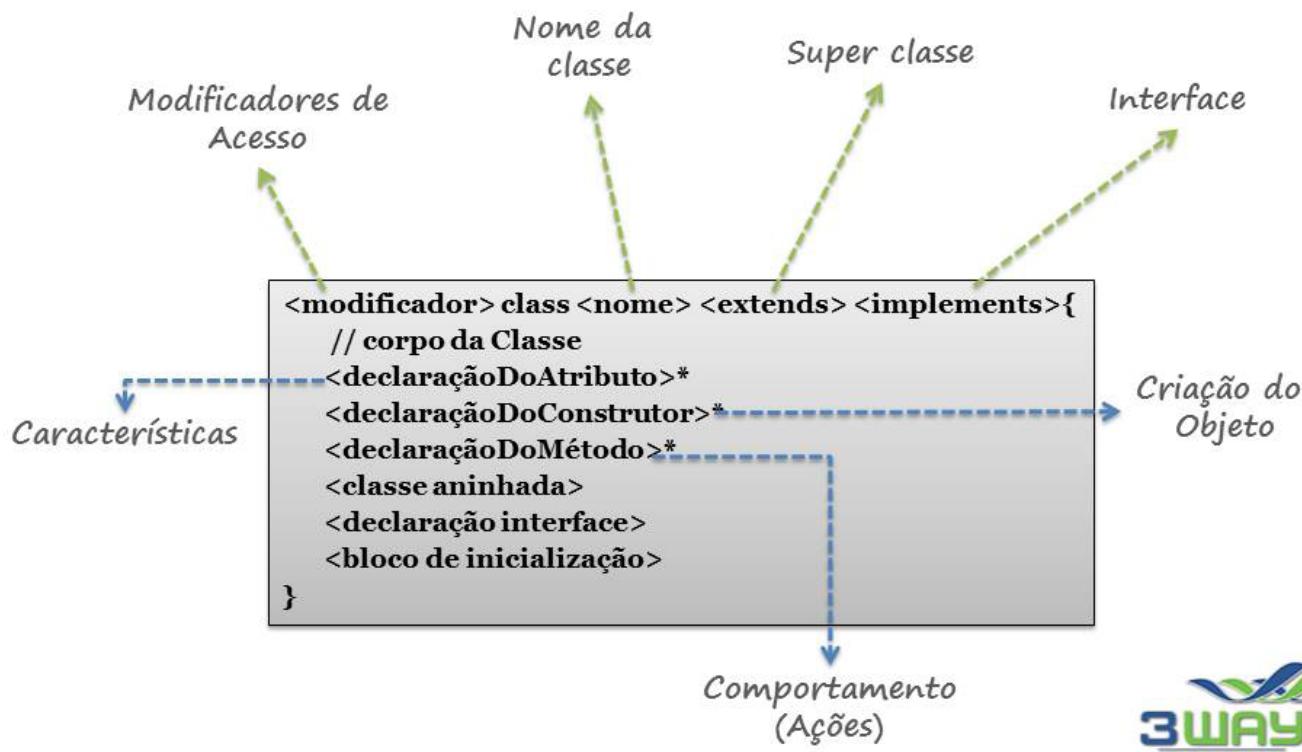
Classes

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 chamados 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 Orientado a Objeto

Definindo Classes

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:

1. As variáveis de classe ou de instância (Atributos).
2. Os construtores de objetos dessa classe.
3. 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 Orientado a Objeto

Métodos

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.

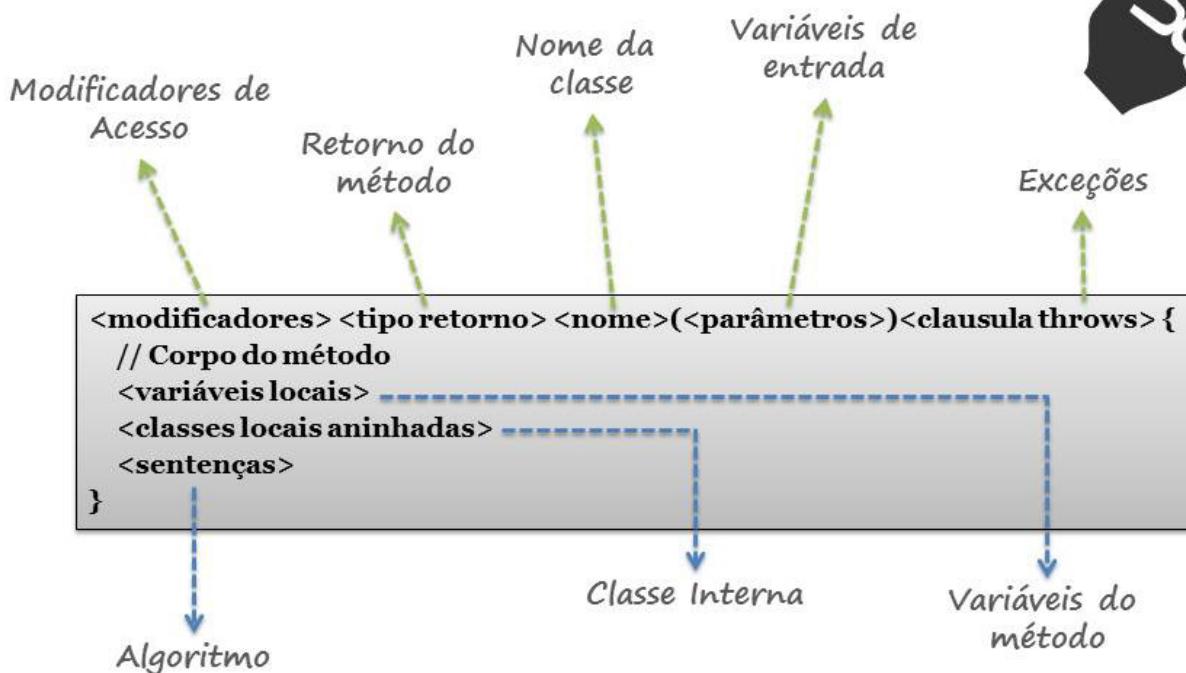
Uma das grandes vantagens, se não a maior, é utilizar e acessar várias vezes os métodos em Java que escrevemos.

- **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



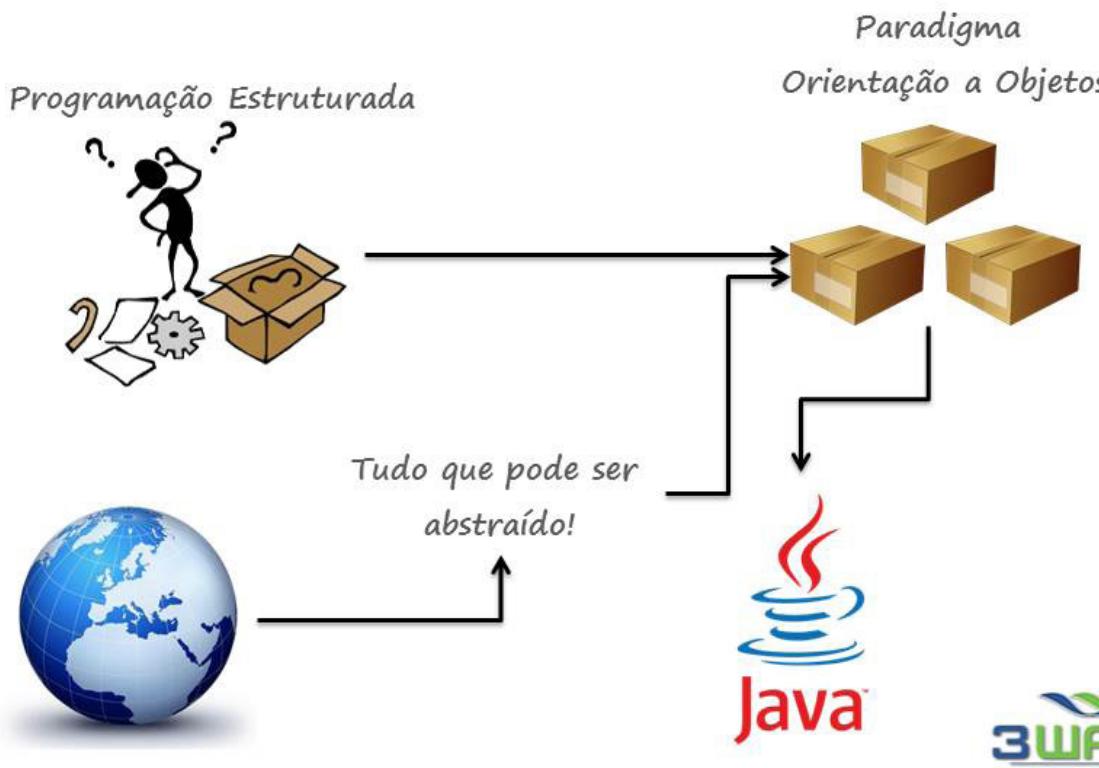
Java Orientado a Objeto

Definindo Métodos

- 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 Orientado a Objeto

Objetos

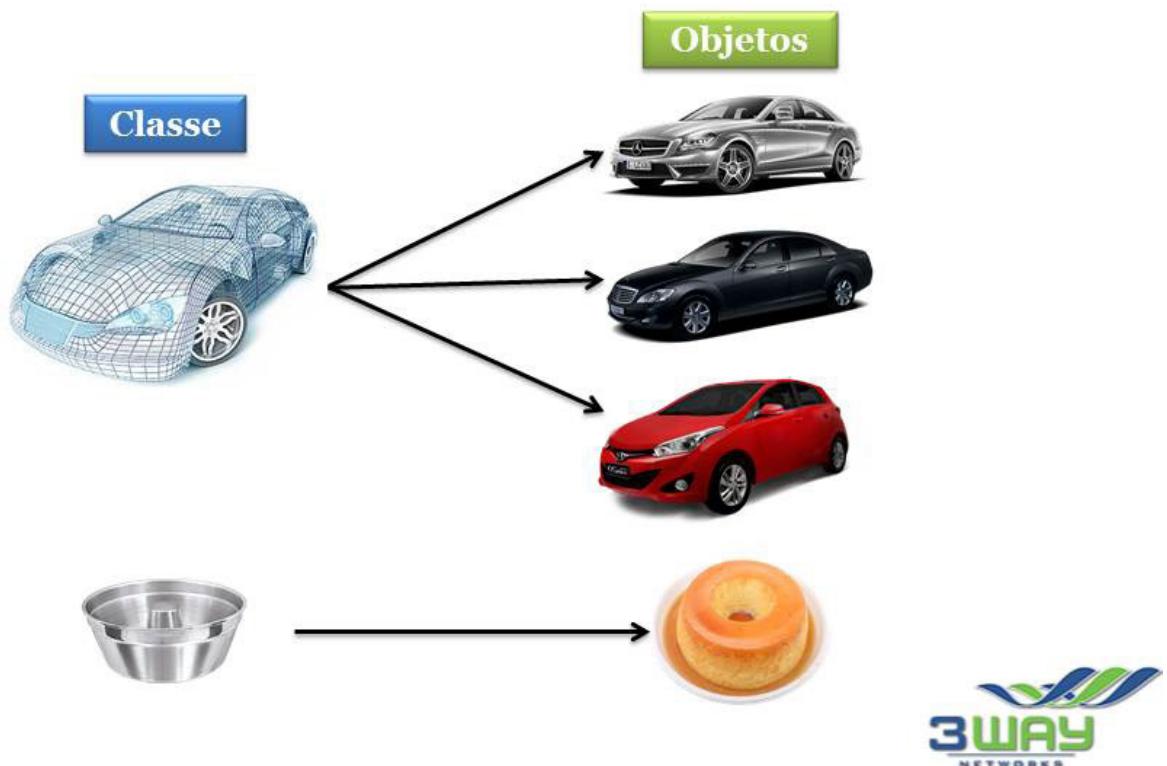
Um **Objeto** é qualquer coisa criada a partir dessa **Classe**, algo contendo as propriedades e comportamentos definidos na **Classe**. Voltando ao exemplo do carro, um grupo de engenheiros mecânicos cria o projeto de um carro (a **Classe**) e a partir deste projeto são reproduzidos milhares de automóveis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

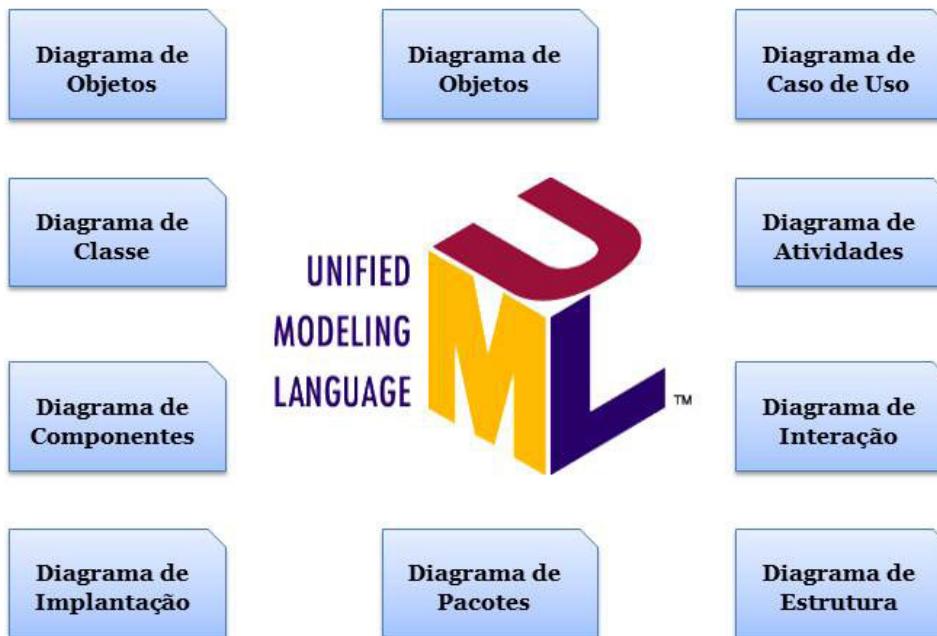
Classe x Objeto

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.

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

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



Java Orientado a Objeto

Notação UML

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:

1. Os diagramas de classes - (Dados estruturais);
2. Os diagramas de casos de uso (Operações funcionais);
3. 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.

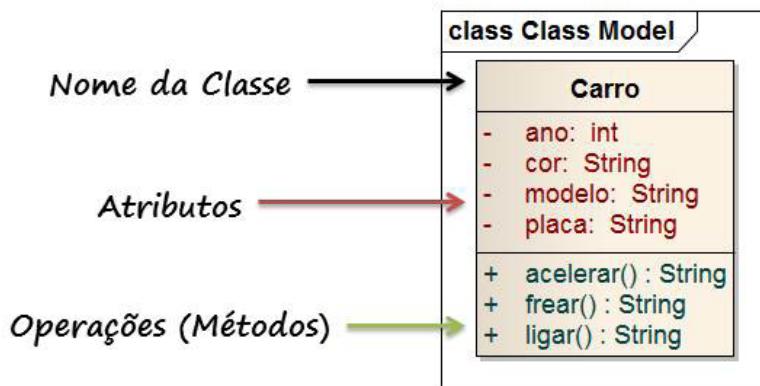
É importante distinguir entre um modelo UML e um diagrama (ou conjunto de diagramas) de UML. O último é uma representação gráfica da informação do primeiro, mas o primeiro pode existir independentemente. O XMI (XML Metadata Interchange) na sua versão corren-

te 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 Orientado a Objeto

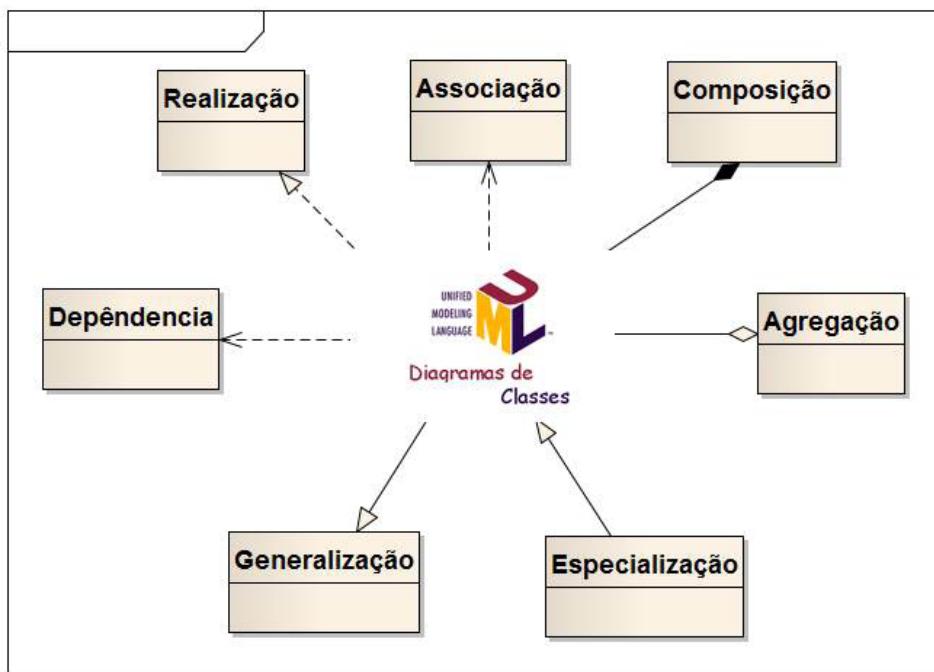
Diagrama de Classes

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 Orientado a Objeto

Relacionamentos

- **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**

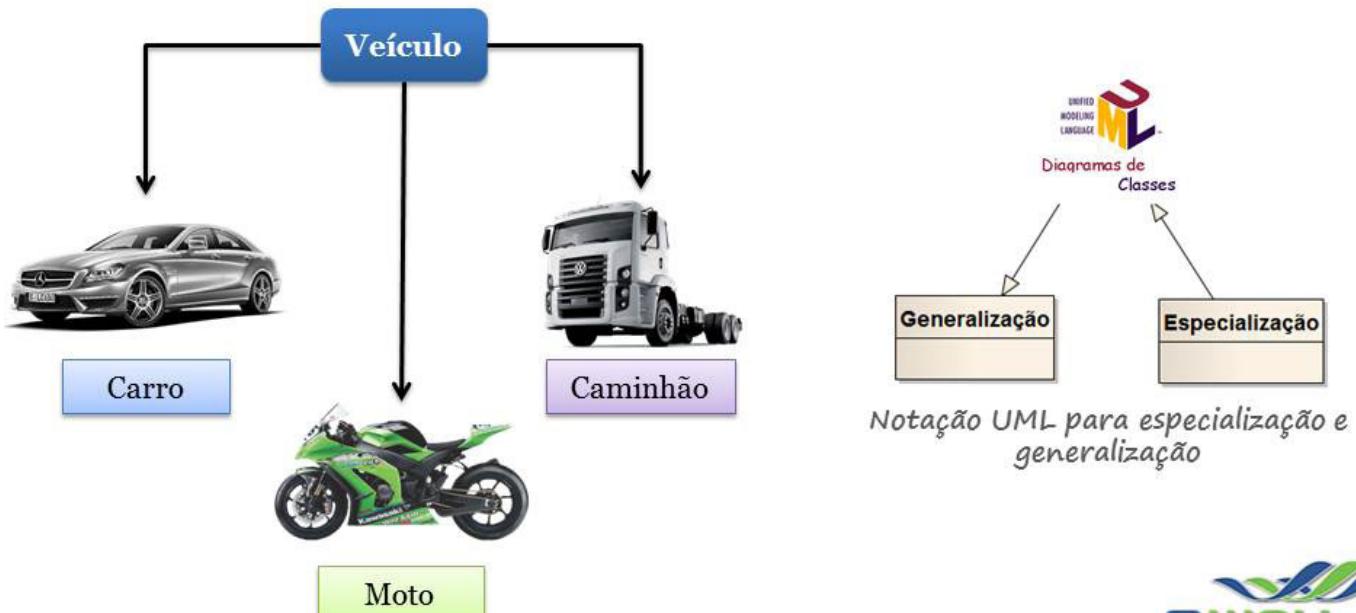
A especialização cria uma classe herdada da generalização onde refina o processo definido na classe pai. Como o próprio nome diz, especializa a classe pai para um tipo específico. Usando a mesma imagem acima da generalização, podemos identificar quem é a generalização “subindo” e quem é a especialização “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 Orientado a Objeto

Herança

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 Orientado a Objeto

Agregação

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

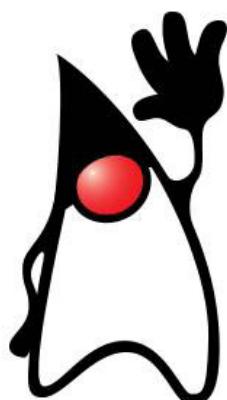
Java implementa o conceito de agregação de objetos pelo uso da referência. Objetos em Java não podem conter outros objetos explicitamente. Objetos somente poderão conter variáveis de tipos primitivos e referências a outros objetos.

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 Orientado a Objetos

Métodos, Construtores e Membros Estáticos



Java Orientado a Objeto

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 Orientado a Objeto

Declarando membros: Variáveis e Métodos

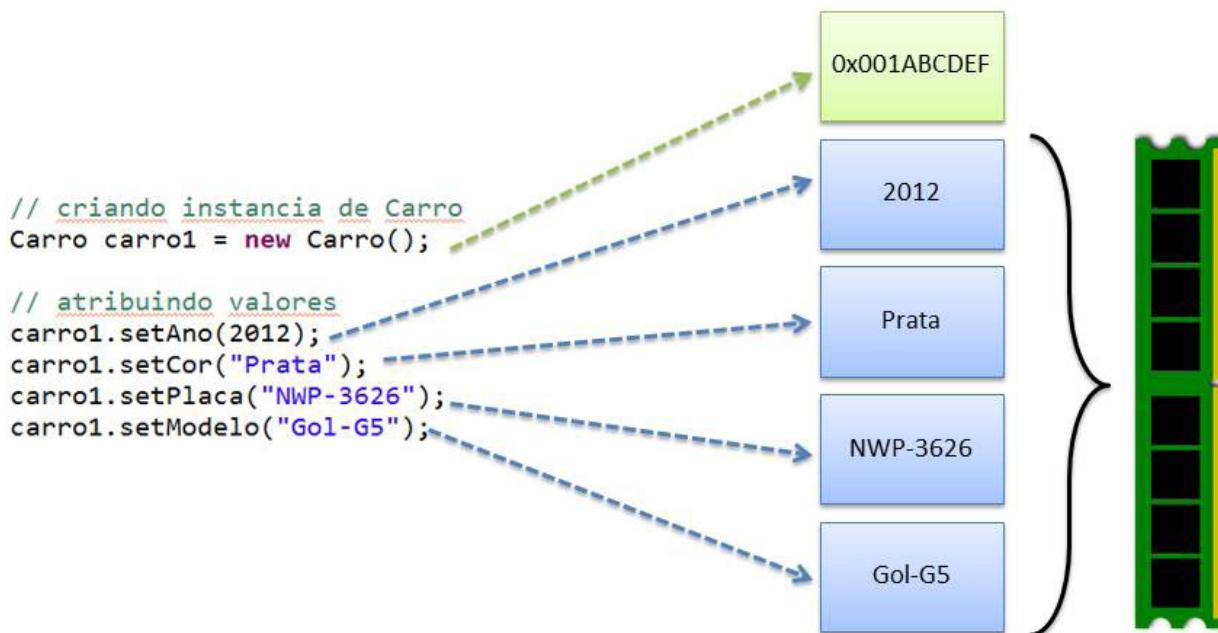
A classe acima mostra a implementação em Java da definição da classe mostrada. O exemplo, apesar de não se preocupar com a utilidade desta implementação, mostra as principais características da definição de uma classe em Java.

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 Orientado a Objeto

Referência de 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.

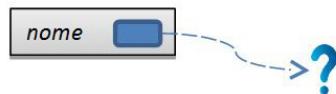
Para trazer em um objeto os seus métodos de instância, é preciso ter referência a algum objeto. As variáveis de referência são inicializadas com o valor “**null**” (nulo).

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 declara-se 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:

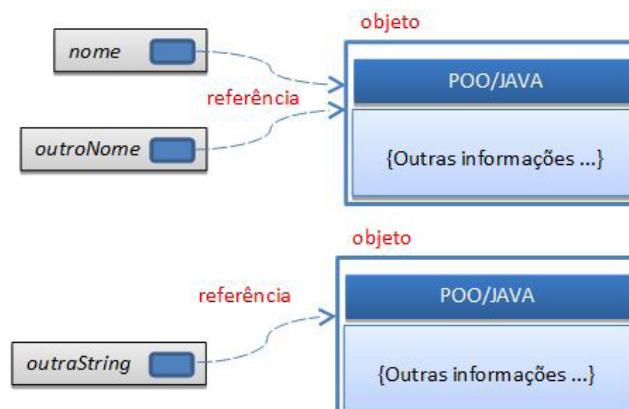
```
String outroNome = nome;
```

Não cia outro objeto, mas simplesmente uma 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 com 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.frear();
```



Java Orientado a Objeto

Invocação de Métodos

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:

Declaração do método	Definição
<code>public char charAt(int index)</code>	Retorna o caractere especificado no índice.
<code>public boolean equalsIgnoreCase (String anotherString)</code>	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 true
boolean result = str1.equalsIgnoreCase(str2);
```

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 Orientado a Objeto

Passagem de parâmetro por valor

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 Orientado a Objeto

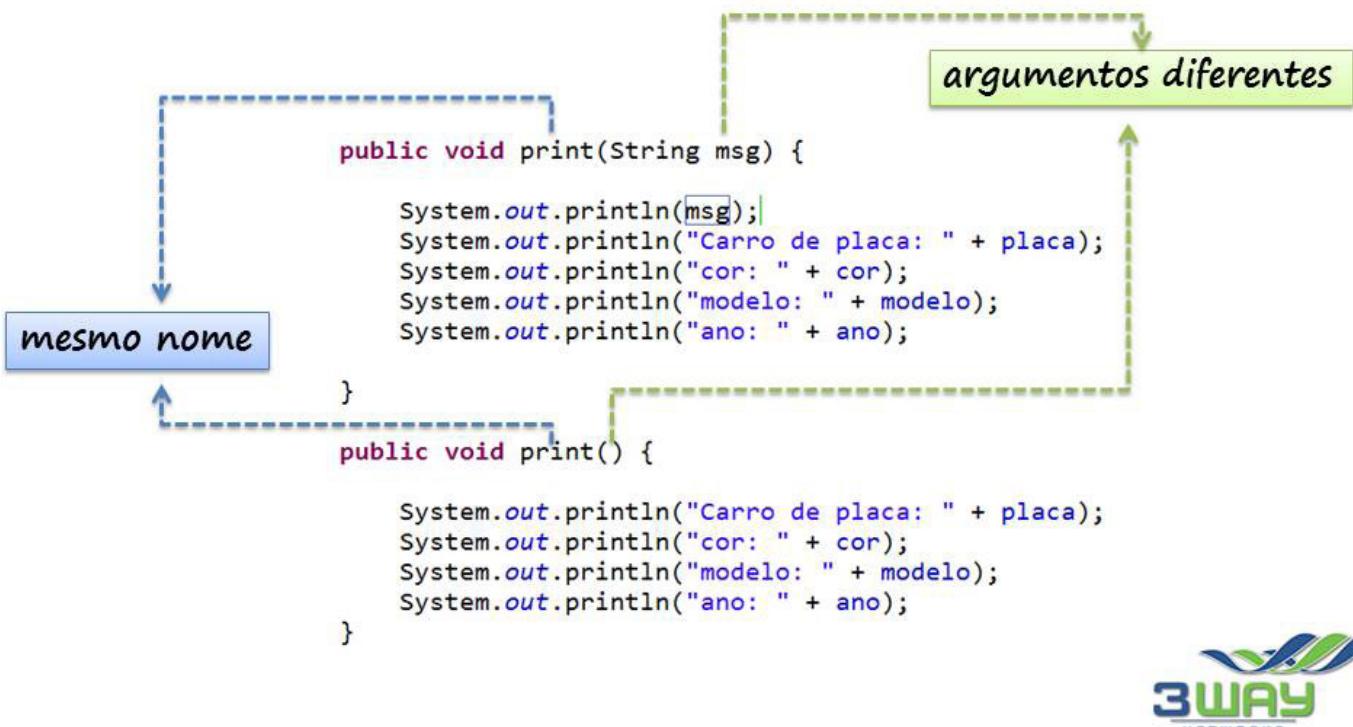
Passagem de parâmetro por referência

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 Orientado a Objeto

Sobrecarga de métodos (Overloading)

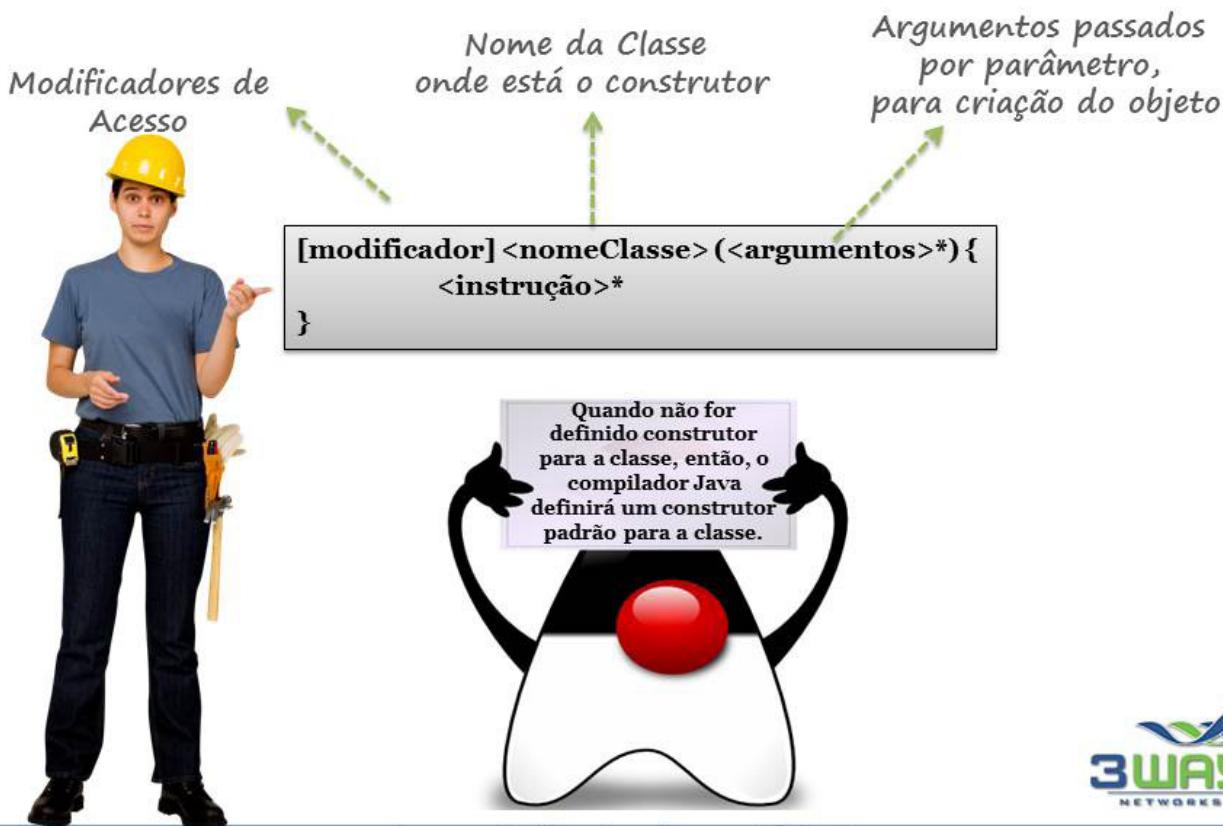
O termo sobrecarga vem do fato de declararmos vários métodos com o mesmo nome, estamos carregando o aplicativo com o ‘mesmo’ método. A única diferença entre esses métodos são seus parâmetros e/ou tipo de retorno.

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 Orientado a Objeto

Construtores

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.

Utilizado quando há uma chamada com o operador **new**. São como métodos, mas a assinatura de um construtor é diferente de um método, temos as propriedades de um construtor:

1. Devem ter o **mesmo nome da classe**
2. 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)
3. 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:

1. O espaço para o objeto é alocado e seu conteúdo é inicializado **null**.
2. O construtor da classe base é invocado. Se a classe não tem uma superclasse definida explicitamente, a classe Object é a classe base.
3. Os membros da classe são inicializados para o objeto, seguindo a ordem em que foram declarados na classe.
4. 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 - Sobre carga)



Java Orientado a Objeto

Overloading de Construtores

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 Orientado a Objeto

Utilizando o Construtor this()

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

Esse conceito de auto referência 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.

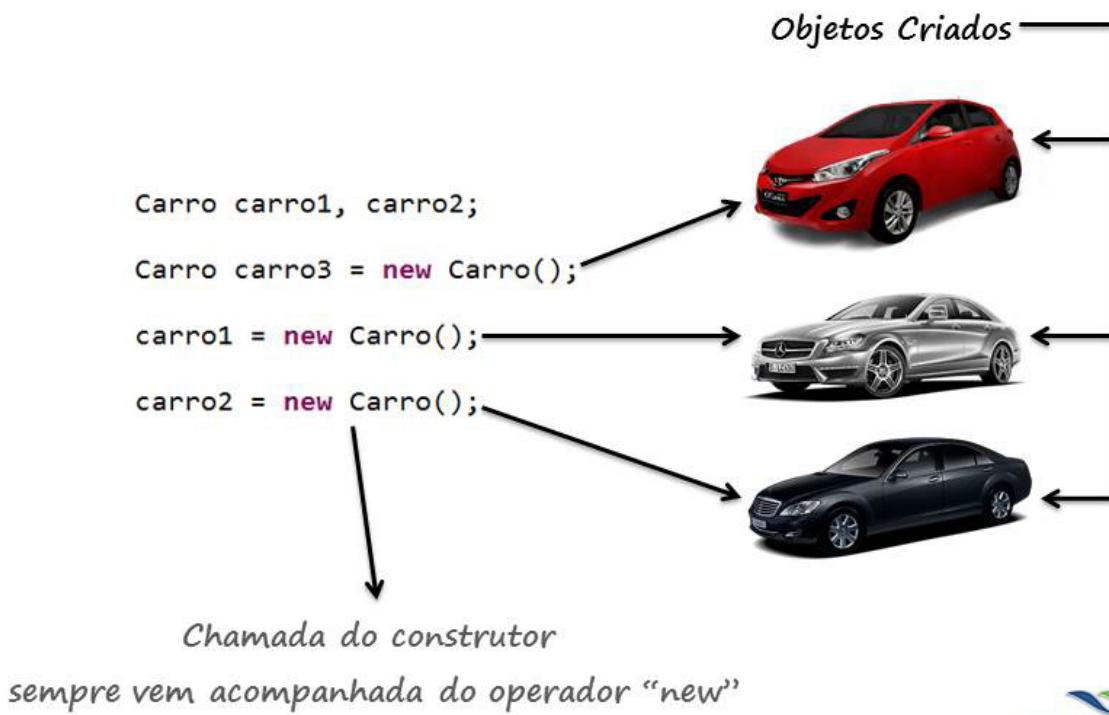
Chamadas a construtores podem ser encadeados, o que significa ser possível chamar um construtor a partir de outro construtor. Usamos uma invocação a **this()** para isso.

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

1. A chamada ao construtor DEVE SEMPRE OCORRER NA PRIMEIRA LINHA DE INSTRUÇÃO
2. 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 Orientado a Objeto

Instância de Classes

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;

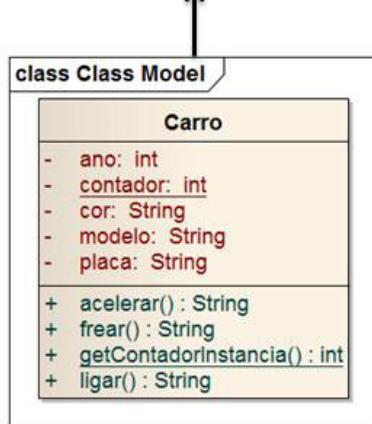
    // (2)Construtor
    // modificação para implementar contador de instâncias
    public Carro() {
        contador++;
        System.out.println("Criando objeto Carro");
    }

    // (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 Orientado a Objeto

Membros estáticos

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 Orientado a Objeto

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

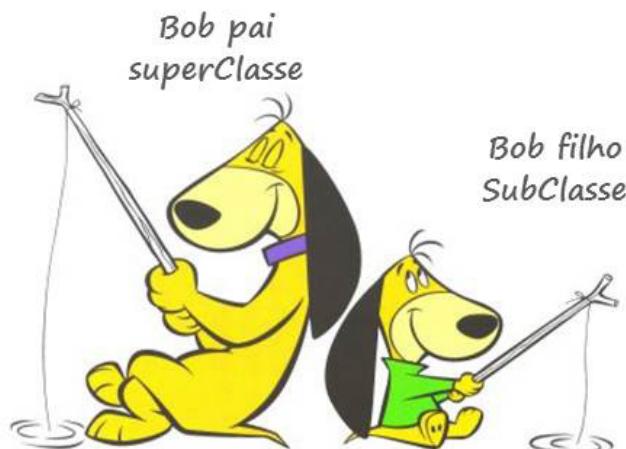
Principalmente quando você quiser ter um controle sobre os objetos ou quando todos os objetos devem partilhar uma informação (evitar ter que fazer Composição ou chamar métodos de outros objetos).

- **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 Orientado a Objetos Herança e Polimorfismo



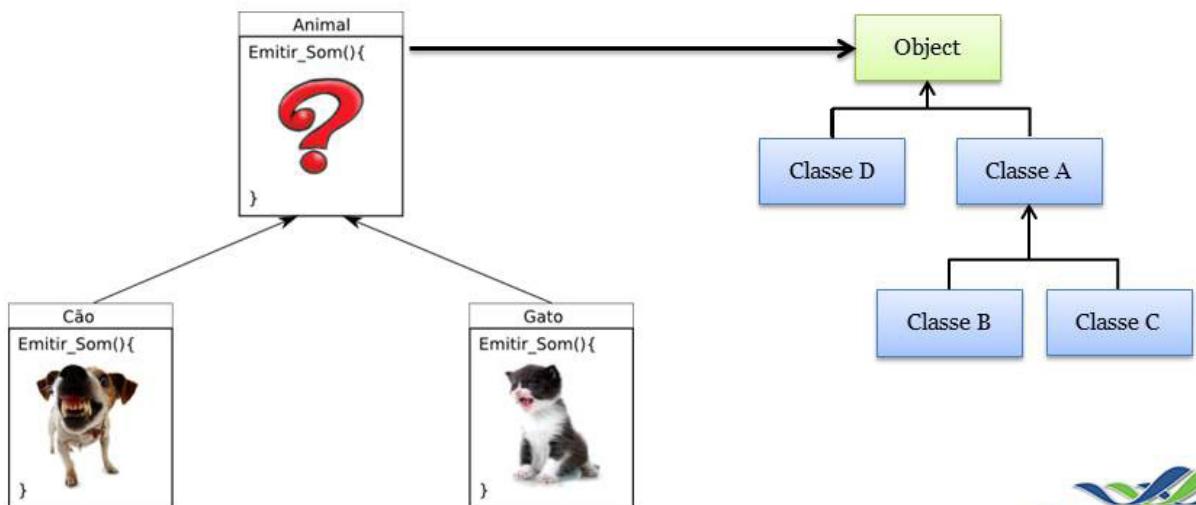
Java Orientado a Objeto

Herança e Polimorfismo

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 Orientado a Objeto

Herança

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.

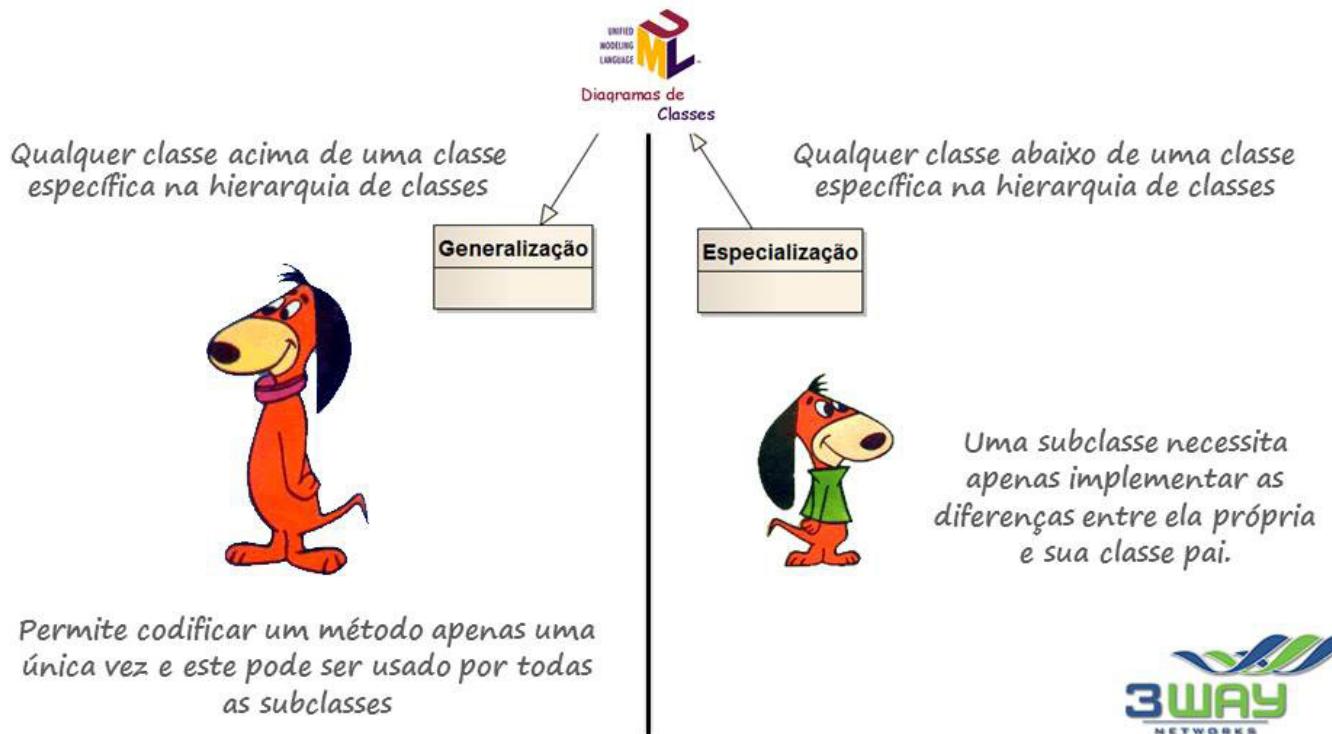
Por exemplo, Imagine que dentro de uma organização empresarial, o sistema de RH tenha que trabalhar com os diferentes níveis hierárquicos da empresa, desde o funcionário de baixo escalão até o seu presidente. Todos são funcionários da empresa, porém cada um com um cargo diferente. Mesmo a secretaria, 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 quan-

to 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 Orientado a Objeto

Superclasse e Subclasse

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, subtípo, 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 Orientado a Objeto

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;

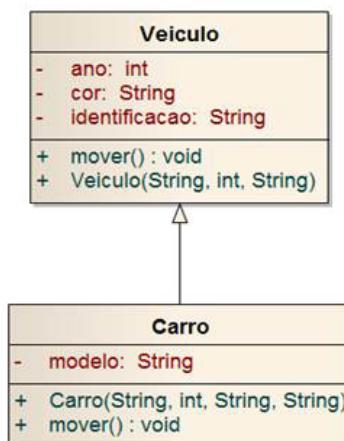
        System.out.println("Criando objeto Carro");
    }

    @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 Orientado a Objeto

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 Orientado a Objeto

Modificador de classe **final**

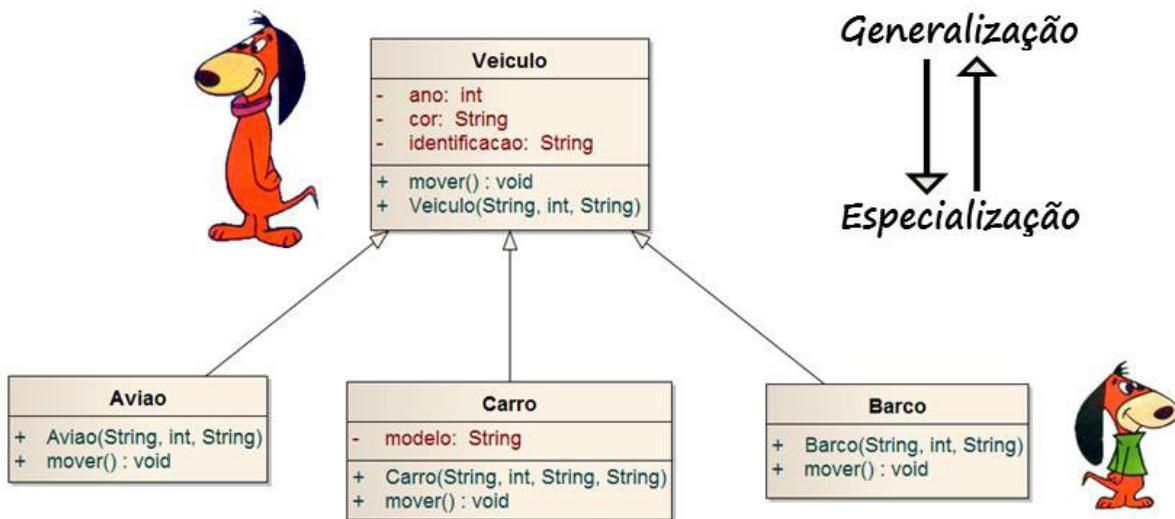
O modificador final pode ser aplicado a variáveis, métodos e classes. O comportamento descrito pelo operador final varia de elemento para elemento, mas tem um conceito em comum, o conceito de imutabilidade, isto é, elementos final não podem ser mudados.

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 classes 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 final.

Polimorfismo



Java Orientado a Objeto

Polimorfismo

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:

1. Métodos declarados como final não podem ser redefinidos e, portanto não são passíveis de invocação polimórfica da parte de seus descendentes;
2. 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 sobrescrita 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 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 ser não 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 do método sobreposto na superclasse.



Java Orientado a Objeto

Sobreposição de Métodos @Override

Substituir um método como o próprio nome já diz, significa cancelar um método de uma superclasse em uma subclasse dando-o 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 **Veiculo** possui como descendentes as classes **Carro**, **Avião** e **Barco**, conforme ilustrado. Observe que as classes filhas sobrepõem o método **mover()** da classe **Veiculo**, modificando seu comportamento.

As seguintes regras para sobreposição de métodos devem ser seguidas:

1. O tipo de retorno do método na subclasse deve ser o mesmo ou um subtipo do tipo de retorno do método sobreposto na superclasse.
2. O método que está sobrepondo não pode ser menos acessível que o seu método sobreposto.
3. O método que está sobrepondo não pode lançar exceções de tipos diferentes das exceções 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 Orientado a Objeto

Referências polimórficas

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**, **Aviao** 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, Template Methodetc;

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 Orientado a Objeto

Coleções Heterogêneas de 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 Canario e 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).

```

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");
Carro    veiculo[3] = new Aviao("Branco", 2010, "Boing 737");

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

Barco      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 Orientado a Objeto

Determinando a Classe de um Objeto

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 cast up 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 cast down é um cast que deve ser explícito, isso acontece porque o compilador não pode garantir a integridade da instância.

Pode-se utilizar os recursos de polimorfismo 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 o da **superclasse** dizemos que houve **up-casting**.

Observe o 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>)<cláusula throws> { ... }
```



Java Orientado a Objeto

Modificador de método **final**

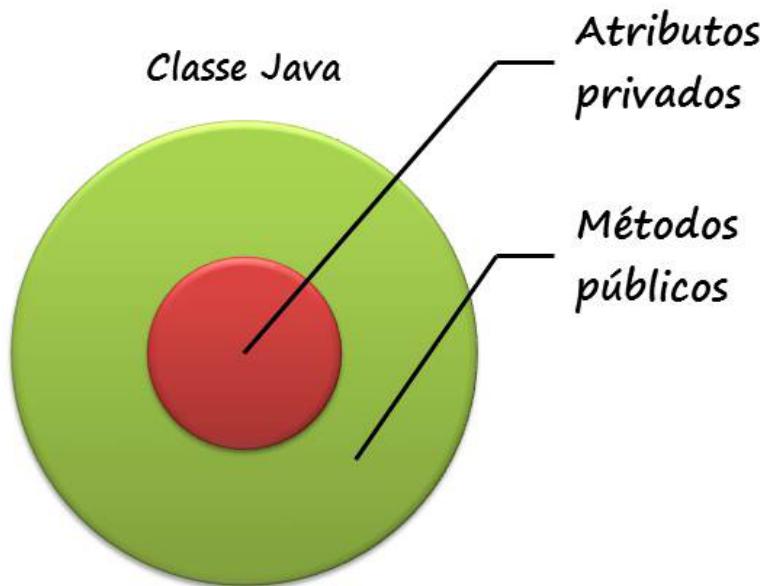
Métodos marcados como **final** são o que chamamos de métodos finais. Para declarar um método final, adicionamos a **palavra-chave final** na declaração do método (na posição do modificador).

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 Orientado a Objeto

Encapsulamento

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 atri-

buição 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;
}

public String getCor() {
    return cor;
}

```

possibilita alteração dos valores (variáveis) por outros objetos.

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



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



Java Orientado a Objeto

Métodos de Configuração e Captura

Até agora víhamos cometendo um grande erro na implementação de nossas classes. Nossa 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

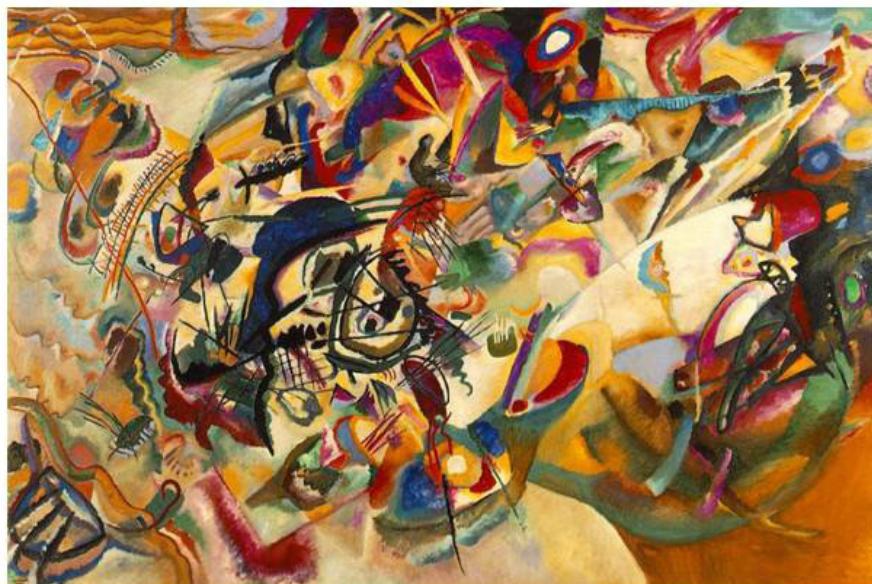
O método de captura recebe o nome de **get<NomeDoAtributo>()**. Ele **retorna** um objeto do mesmo **tipo do atributo** e deve retornar o **valor do atributo** desejado.

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>)**.

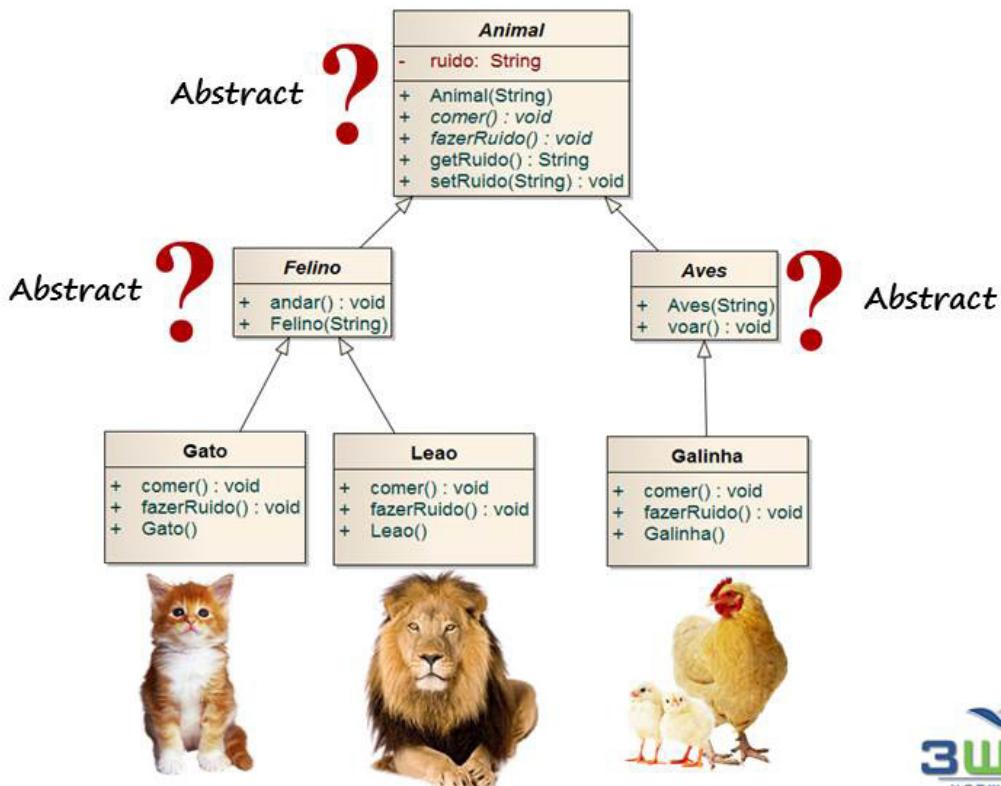
Java Orientado a Objetos

Classes Abstratas, Internas e Interfaces



Java Orientado a Objeto

Classes Abstratas



Java Orientado a Objeto

Classes Abstratas

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.

Como classes abstratas são modelos de classes, então, não podem ser instanciadas diretamente com o operador **new**, elas sempre devem ser **herdadas** por classes concretas. São apenas modelos, abstrações que agrupam e definem um conjunto de objetos com característica 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 Orientado a Objeto

Métodos Abstratos

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 Orientado a Objeto

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 Orientado a Objeto

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

Notação UML

«interface»
Calculos
+ multiplicacao(Double, Double) : Double
+ soma(Double, Double) : Double
+ subtracao(Double, Double) : Double

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



Java Orientado a Objeto

De modo geral uma **interface** é um **tipo de classe abstrata** que só pode conter a assinatura de **métodos abstratos** ou **atributos** marcados com o **modificador final**. Interfaces, por natureza, são abstratas, não necessitando do modificador **abstract**. A partir de Java 8 o tipo interface também aceita métodos default implementados.

Utilizamos interfaces quando queremos que uma classe tenha um comportamento semelhante a outra mas não podemos usar uma relação de herança. 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. Como veremos mais adiante, podemos utilizar uma interface como tipo de dados.
- Utilizar interfaces como mecanismo **alternativo para herança múltipla**, que permitem às classes terem herdarem comportamento além da superclasse.

Criando Interfaces

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



Java Orientado a Objeto

Criando Interfaces

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

Quando uma classe implementa uma interface ela aceita o contrato da interface, então ela tem que implementar todos os métodos abstratos definidos na interface, caso contrário o compilador Java emitirá um erro de compilação. A partir de Java 8 um a interface pode conter métodos **default** e **static** sem quebrar código existente.

Há alguns pontos importantes á 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). Para garantir use a anotação **@Override**
- O nome dos argumentos não precisam ser iguais aos que serão usados na classe.

Implementando Interfaces

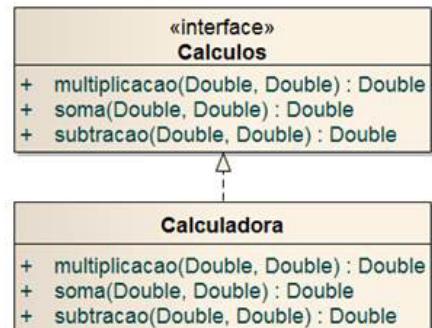
```
public class Calculadora implements Calculos {
    @Override
    public Double soma(Double x, Double y) {
        return x + y;
    }

    @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 Orientado a Objeto

Implementando Interfaces

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**.

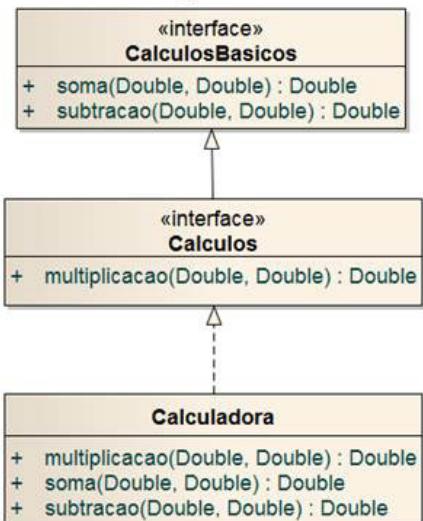
A vantagem principal das interfaces é que não há limites de quantas interfaces uma classe pode implementar, o que ajuda no caso de heranças múltiplas que não é possível ser feito em Java, pois uma classe apenas pode herdar as características de uma outra classe, ou seja, a classe pode apenas **herdar**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

Herança entre Interfaces

Em Java você também pode realizar herança entre interfaces. Uma interface, ao herdar de outra, automaticamente assume os métodos desta de forma implícita.

Um classe Java pode implementar múltiplas interfaces, por exemplo,

```
public class Concreta implements I1, I2, I3{ ... }
```

Isso implica que a classe concreta terá que implementar todos o métodos definidos em I1,I2 e I3, da mesma forma, se temos uma super interface como :

```
interface ISuper extends I1,I2,I3{...}
```

Uma classe concreta que implementa ISuper deverá implementar todos os métodos definidos em ISuper,I1,I2,I3

```
class ConcretaSuper implements ISuper{...}
```

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 Orientado a Objeto

Interface vs. Classe

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

Métodos de Extensão

E se eu adicionar
um novo método
na Interface?

Até **Java 8** – Obrigatório
implementar em todas as classes
concretas, código seria quebrado!



```
public interface Calculos {
    public Double soma(Number x, Number y);
    public Double subtracao(Number x, Number y);
    public Double multiplicacao(Number x, Number y);
    // a partir de Java 8
    default Double divisao(Number x, Number y){
        return helper(x.doubleValue(),y.doubleValue());
    };
    // a partir de Java 8
    static double helper(double x, double y){
        return x/y;
    }
}
```

Métodos **default** e **static** na interface não afeta classes concretas implementadas anteriormente, a interface pode evoluir sem quebrar código existente



Java Orientado a Objeto

Até Java 8 você não podia implementar métodos dentro de interfaces, qualquer adição de assinatura de método dentro de uma interface existente quebraria seu código. Todo método abstrato, que não são default ou static, ainda precisa ser implementado em uma classe concreta.

Você pode agora implementar métodos default dentro de interfaces, esses métodos não afetam classes existentes e permitem adicionar novas funcionalidades à interface. Esses métodos também podem ser sobreescritos na classe concreta.

```
public class CalculosConcreta implements Calculos{
    @Override
    public Double divisao(Number x, Number y) {
        return Calculos.super.divisao(x, y);
    }
}
```

Os métodos estáticos são usados da mesma forma que métodos estáticos dentro de classes, seu principal intuito é facilitar organização de métodos *helper* dentro de suas bibliotecas.

@Override

```
public Double divisao(Number x, Number y) {  
    return Calculos.dividir(x, y);  
}
```

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 Orientado a Objeto

Classes Internas (Aninhadas)

Classe aninhada (*nested class*) 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.

- Agrupar classes que só são utilizadas num único lugar
- Aumentar o encapsulamento.

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 sobrepõem 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 Locais

São classes internas definidas dentro de blocos de sentenças, normalmente são classes definidas dentro de métodos. Veja um exemplo:

```
public class LocalInnerClass{  
    private int data=30;//variavel de instance  
    void display(){  
        class Local{  
            void msg(){System.out.println(data);}  
        }  
        Local l=new Local();  
        l.msg();  
    }  
    public static void main(String args[]){  
        LocalInnerClass obj=new LocalInnerClass();  
        obj.display();  
    }  
}
```

Use classes locais se você precisar criar mais do que uma instância de uma classe interna, com acesso ao seu construtor, ou para criar um novo tipo a ser usado localmente. Atente para o escopo dessa classe, a mesma só pode ser utilizada dentro do método.

Classes Interna Anônima



Classe interna sem nome com propósito de escopo limitado, utilizado para declarar e instanciar um objeto uma única vez

```

public class AnonymousInnerClass {
    // Inner class
    class Fruta {
        public String nome;
    };

    public void ordenar(List<Fruta> frutas) {
        // Ordenar
        Collections.sort(frutas,
            // Anonymous Inner Class
            new Comparator<Fruta>() {
                @Override
                public int compare(Fruta fruta2, Fruta fruta1) {
                    return fruta1.nome.compareTo(fruta2.nome);
                }
            });
    }
}

```

Objeto
definido e
criado no
parâmetro



Java Orientado a Objeto

Classes internas anônimas (anonymous inner classes), usado para tornar seu código mais conciso. Mas infelizmente aumenta a complexidade de entendimento de seu código. Elas permitem que você declare e instancie uma classe ao mesmo tempo. Elas são como classes locais, porém sem nome. Use essa estratégia se você precisar declarar campos ou métodos adicionais. Um uso bem comum é na manipulação de eventos, classes ouvintes, por exemplo manipular o evento de clique de um botão da classe gráfica Swing:

```

//criando um objeto jbutton
JButton showDialogButton = new JButton("Clique aqui");
//adicionar ouvinte para o
//jbutton manipular evento "pressed"
showDialogButton.addActionListener(new ActionListener()
{
    public void actionPerformed(ActionEvent e)
    {
        //mostrar um JDialog
        JDialog d = new JDialog(frame, "Olá", true);
        d.setLocationRelativeTo(frame);
        d.setVisible(true);
    }
});

```

$() \rightarrow \lambda$

Java 8 - Lambda

- É um bloco de código com parâmetro:
(String p, String s) -> Integer.compare(p.length(), s.length())
 podemos usar para simplificar digitação de classes anônimas na implementação de interfaces funcionais.

```
// Ordenar
Collections.sort(frutas,
    // Anonymous Inner Class
    new Comparator<Fruta>() {
        @Override
        public int compare(Fruta fruta2, Fruta fruta1) {
            return fruta1.nome.compareTo(fruta2.nome);
        }
    });

```



@FunctionalInterface
public interface Comparator<T>

Interface Funcional,
 só tem um único
 método abstrato



```
// Ordenar
Collections.sort(frutas,
    (fruta1, fruta2) ->
        fruta1.nome.compareTo(fruta2.nome)
);

```

Java Orientado a Objeto

Expressões lambda

Um problema com classes anônimas é que, se a implementação de sua classe anônima é muito simples, tais como uma interface que contém apenas um método(INTERFACE FUNCIONAL), a sintaxe de classes anônimas pode parecer complicado. Nesses casos, você geralmente está tentando passar a funcionalidade como um argumento para um outro método, bem como a ação que deve ser tomada, como quando alguém clica em um botão. As expressões lambda permitem que você faça isso de uma forma mais simples e clara.

A seção anterior, classes anônimas , mostra como implementar uma classe base sem dar-lhe um nome. Embora isto seja muitas vezes mais conciso do que chamar uma classe de alto nível, para as classes que contém apenas um método, até uma classe anônima parece um pouco exagerado e verboso. As expressões lambda te deixam expressar instâncias de classes com único método mais compacta.

```
showDialogButton.addActionListener( e -> {
    { //mostrar um JDialog
        JDialog d = new JDialog(frame, "Olá", true);
        d.setLocationRelativeTo(frame);
        d.setVisible(true);
    } );
}
```

Tipos Enumerados

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

```
public enum EnumEstacoes {

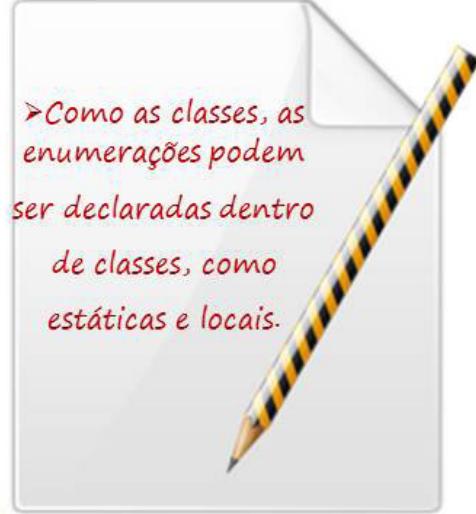
    PRIMAVERA(EnumMes.SETEMBRO, EnumMes.NOVEMBRO),
    VERAO(EnumMes.DEZEMBRO, EnumMes.FEVEREIRO),
    OUTONO(EnumMes.MARCO, EnumMes.MAIO),
    INVERNO(EnumMes.JUNHO, EnumMes.AGOSTO);

    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;
    }
}
```



Java Orientado a Objeto

Tipos Enumerados

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,

static<tipo da sua enum>[] values()

Retorna um array contendo as constantes desta enum, na ordem em que são declaradas.

static<tipo da sua enum> valueOf(String nome)

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 Orientado 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 (0C0000005, 0xBFBABFF1B, 0xB8F61B14, 0x00000000)
*** nv4_disp.dll - Address BHABBF1B base at BF9D4000, Datestamp 4410c8d4

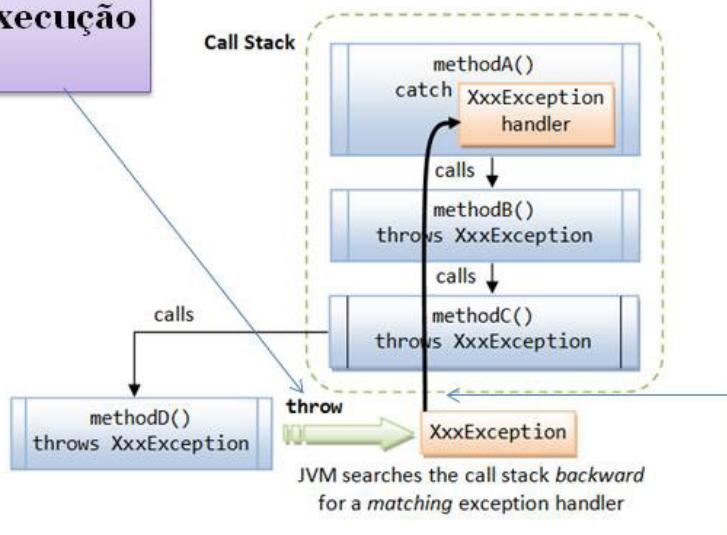
Iniciando despejo de memória física.
Despejo de memória física concluída.
Entre em contato com o administrador.
do sistema ou grupo de suporte técnico para obter a informação.
```



Java Orientado a Objeto

Exceções

Erros que ocorrem durante a execução do programa



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



Java Orientado a Objeto

Exceções

Uma exceção é um **evento EXCEPCIONAL** que **interrompe o fluxo** normal de processamento, podendo terminar de forma inesperada a execução de seu programa.

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

- **ArrayIndexOutOfBoundsException**: ocorre ao acessar um elemento inexistente de um **array**.
- **NumberFormatException**: ocorre ao enviar um parâmetro não-numérico para o método **Integer.parseInt()**.

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

1. 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.
2. Exceções geradas por **violações de contratos** entre **APIs**: O programa tenta fa-

zer 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 exemplo é tentar passar uma URL inválida para o construtor da classe **java.net.URL**, como exemplo **hppt://www.google.com** vai gerar uma **MalformedURLException**. Claro, não existe o protocolo **hppt**.

3. 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.

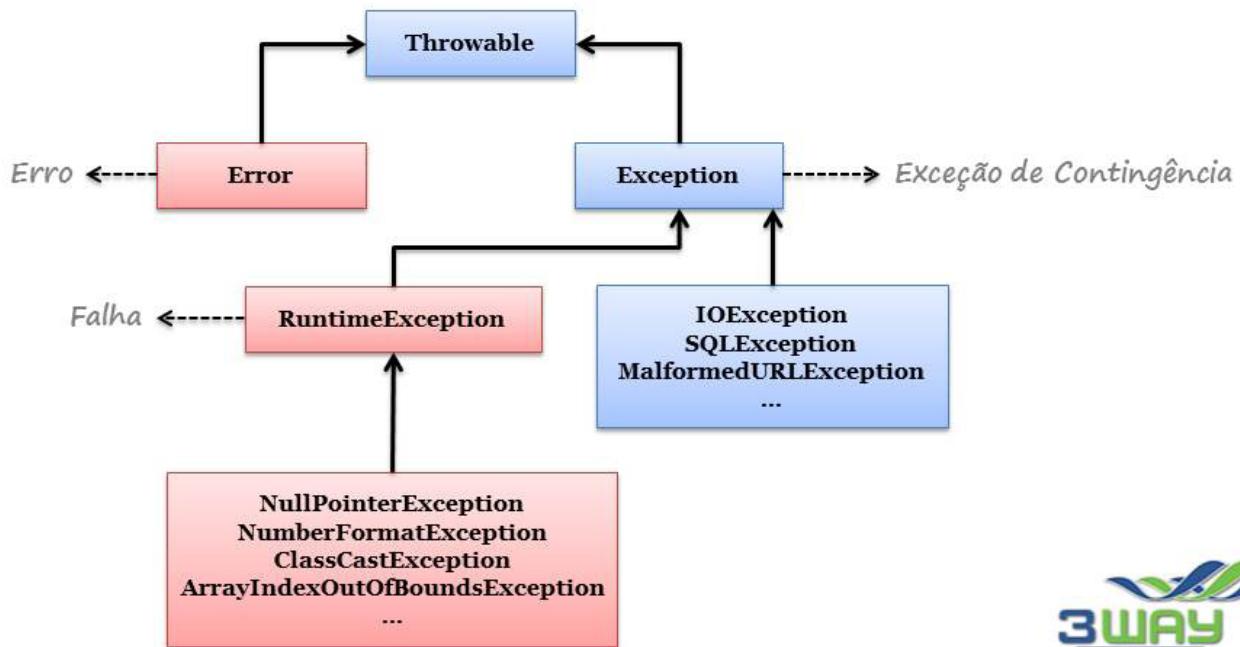
Como funciona

Quando um método lança (**throw**) uma exceção, a JVM dispara um evento de alerta e verifica se o método atual possui o mecanismo de tratamento desta exceção (um bloco **try..catch**). Se o método não trata a exceção então o fluxo de processamento é interrompido e repassado ao próximo método a baixo na pilha de invocação de métodos (**call stack**), fazendo nova verificação e assim consecutivamente até retorna ao método **main**, e então a execução corrente da JVM é morta. Ao longo desse processo, uma pilha com o rastro do caminho (**stacktrace**) percorrido pela JVM é montado e entre (**stacktrace**)

Alguns métodos podem avisar (**throws**) que são passíveis de lançar um ou mais tipos de exceção. O método invocador aceita o “contrato” e então decide se trata (um bloco **try..catch**) ou repassa (**throws**) a exceção.

Categoria de Exceções

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



Java Orientado a Objeto

Categoria de Exceções

Objetos que podem ser lançados para indicar que algo anormal aconteceu, e capturados para lidar com esta situação. Em java uma exceção é uma instância de uma subclasse de `java.lang.Throwable`, entre outros, os métodos comumente usados são:

- **`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.

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**

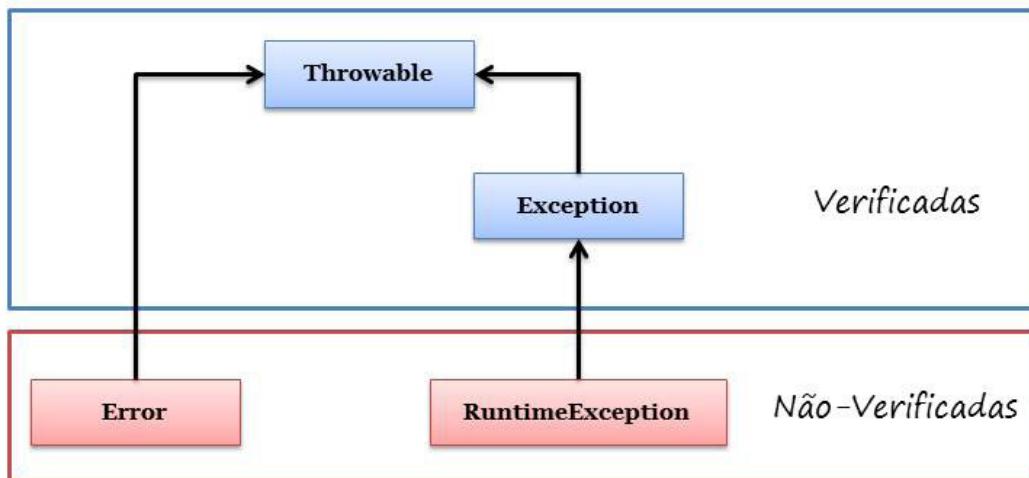
É a raiz das classes derivadas de Throwable que indica situações que a aplicação não deve tentar tratar. Usualmente indica situações anormais, que não deveriam ocorrer.

Exemplos de exceções já definidas em Java incluem:

1. **java.lang.ArithmaticException**: Indica situações de erros em processamento aritmético, tal como uma divisão inteira por 0.
2. **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 subclass de **java.lang.IllegalArgumentException**.
3. **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 subclass de **java.lang.IndexOutOfBoundsException**, assim como a classe **java.lang.StringIndexOutOfBoundsException**.
4. **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.
5. **java.lang.ClassNotFoundException**: Indica que a aplicação tentou carregar uma classe, mas não foi possível encontrá-la.
6. **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.

Exceções Verificadas e Não-Verificadas



Java Orientado a Objeto

Exceções Verificadas e Não-Verificadas

Em Java temos dois tipos de exceção: ***checked (verificadas)*** e ***unchecked (Não verificadas)***. Entende-se por ***exceções rificadas***, aquelas exceções que devem ser obrigatoriamente tratadas pelo seu programa usando ***try..catch*** ou delegadas através da clausula ***throws***. Uma exceção *verificada* é a forma que o método chamado tem para avisá-lo de algo pode sair errado e você deverá fazer alguma coisa!. Todas as subclasses de ***Exception*** que não sejam subclasses de ***RuntimeException*** 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). Todas as classes que são filhas de ***RuntimeException*** não precisam ser tratadas, assim como aquelas que são filhas da classe ***Error***. Esse tipo de exceção assinala problemas de programação, erros que você ou alguém que escreveu uma API, não conseguiu tratar adequadamente ou que de alguma forma não foi possível contornar.

Manipulando Exceções

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

```
public static void main(String... args) {
    PrintStream ps = System.out;
    InputStreamReader leitor = new InputStreamReader(System.in);
    int[] array = { 1, 2, 3, 4 };
    try // IOException
        Character ch = (char) leitor.read();
        // NumberFormatException
        int i = Integer.parseInt(ch.toString());
        // ArrayIndexOutOfBoundsException
        ps.println(array[i]);
    } catch (ArrayIndexOutOfBoundsException e) {
        ps.printf("Índice fora do limite [0..3] : %s\n", e.getMessage());
    } catch (NumberFormatException e) {
        ps.printf("Erro de conversão : %s\n", e.getMessage());
    } catch (IOException e) {
        ps.printf("Erro de entrada/saída : %s\n", e.getMessage());
    } finally {
        ps.println("Sempre passo aqui para fechar todos os recursos");
    }
}
```

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 Orientado a Objeto

Manipulando Exceções

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 o trabalho de captura da exceção é executado pelos blocos **catch**. Um bloco **try** pode possuir vários blocos **catch**, dependendo do número de exceções que podem ser lançadas pelo método invocado.

catch

O bloco **catch** recebe da JVM um argumento que é o objeto da exceção lançada. O tipo do objeto é checado com o tipo da classe definida entre parênteses no **catch()**. A classe especificada deve ser do mesmo tipo do objeto de exceção ou uma superclasse. Deste modo, é possível capturar qualquer exceção se usarmos **Throwable** como parâmetro para o **catch**, mas pode não ser interessante, já que você não saberá o que realmente aconteceu especificamente.

finally

O bloco **finally** é opcional e único, sempre colocado após o último **catch**. Quando houver um **finally**, o fluxo de execução **sempre** passará por ele. Assim, é possível executar alguma ação antes que fluxo de execução seja repassado ao próximo método da pilha de chamada.

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.

Um catch Múltiplas Exceções

Um catch com Múltiplas classes de Exceção

```

public static void main(String... args) {
    PrintStream ps = System.out;
    InputStreamReader leitor = new InputStreamReader(System.in);
    int[] array = { 1, 2, 3, 4 };
    try {// IOException
        Character ch = (char) leitor.read();
        // NumberFormatException
        int i = Integer.parseInt(ch.toString());
        // ArrayIndexOutOfBoundsException
        ps.println(array[i]);
    } catch (ArrayIndexOutOfBoundsException |
             NumberFormatException |
             IOException e) {
        ps.printf("Um erro aconteceu : %s \n", e);
    } finally {
        ps.println("Sempre passo aqui para fechar todos os recursos");
    }
}

```

O bloco catch recebe um argumento de vários tipos de exceção separados pelo operador (|)

Tratamento da exceção

Sempre será executado



JAVA7: Para cada bloco try, pode haver um único catch, com muitos tipos de Exceção



Java Orientado a Objeto

Manipulando Exceções

A partir de Java 7 você pode capturar múltiplas exceções em um único bloco **catch**. Você pode declarar vários parâmetros tipo de classes de exceção usando o operador barra vertical (|). A vantagem é poder agrupar exceções semelhantes que precisem de uma mesmo tratamento, reduzindo código duplicado e, talvez, te incentivar a não capturar tudo em um único catch com um tipo de exceção muito genérico. Exemplo, ao invés de:

```

catch (Exception) {

    logger.log(ex);

    throw ex;

}

```

usar :

```

catch (IOException|SQLException ex) {

    logger.log(ex);

    throw ex;

}

```

try-com-recursos

```

InputStreamReader leitor = new InputStreamReader(System.in);
try { // IOException
    Character ch = (char) leitor.read();
} catch (IOException e) {
    ps.printf("Um erro aconteceu : %s \n", e);
} finally {
    if (leitor != null) {
        try { // fecha recurso
            leitor.close();
        } catch (Exception e) {
            ps.println("Sempre fechar o recurso");
        }
    }
}
try (InputStreamReader leitor =
      new InputStreamReader(System.in)) {
    // IOException
    Character ch = (char) leitor.read();
} catch (IOException e) {
    ps.printf("Um erro aconteceu : %s \n", e);
}

```

Fechando recursos,
tratamento
convencional, até
Java6, finally explícito
e você invoca o método
close() do recurso

Java 7, o recurso é
declarado no try() o finally
é implícito, método close()
de AutoCloseable é invocado
automaticamente



Recursos como arquivos, conexão de banco de dados, socket de rede, etc., que implementam interface AutoCloseable o finally é implícito



Java Orientado a Objeto

Finalizando Recursos

Recursos são objetos que precisam ser finalizados após o término do programa. Como exemplo de recursos temos, objetos de conexão com banco de dados, objetos de manipulação de arquivos, objetos de conexão com servidores http, ftp, etc.

O primeiro exemplo demonstra como nós usamos o **finally**, até Java 6, para garantir que o recurso seja encerrado, no caso um recurso de arquivo.

A partir de Java 7 é possível declarar e inicializar classes de recurso como argumento no **try()**. Qualquer objeto que implemente **java.lang.AutoCloseable**, podem ser declarados inicializados no **try-com-recursos**. Veja no segundo exemplo, não há necessidade do bloco finally. Você pode declarar um ou mais recursos no **try-com-recurso**:

try (

```

java.util.zip.ZipFile zf =
    new java.util.zip.ZipFile(zipFileName);
java.io.BufferedReader writer =
    java.nio.file.Files.newBufferedReader(outputFilePath, charset)
)
{
    .....
}
```

Throw e Throws

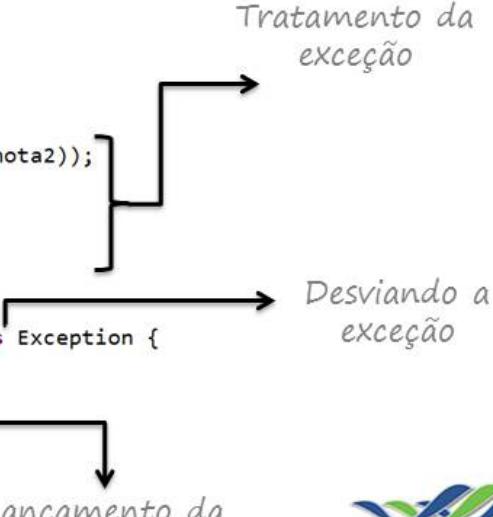


Se um método causar uma exceção mas não capturá-la, então deve-se utilizar a palavra-chave **throws**

```
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 Orientado a Objeto

Throw e Throws

As cláusulas **throw** e **throws** são ações que propagam exceções, ou seja, em alguns momentos existem exceções que não quer que sejam tratadas no mesmo método que a gerou. 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 do tipo **checked**, deve-se utilizar a palavra-chave **throws** para marcar o método como sendo passível de lançar um ou mais tipos de exceção, informando o método invocador que ele pode terá que tratar/relançar essa exceção.

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



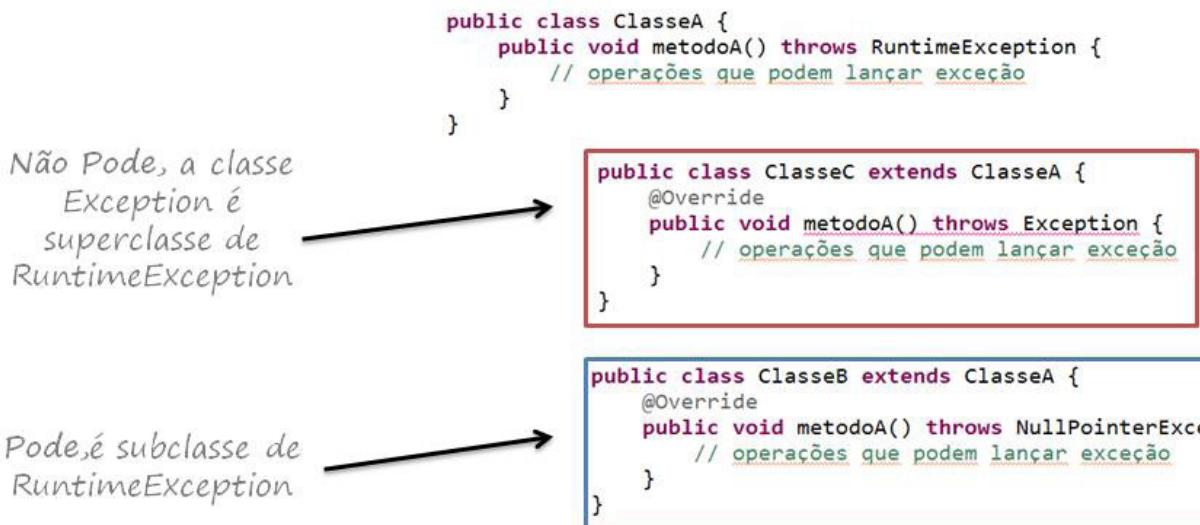
Java Orientado a Objeto

Criando Suas Próprias Classes de Exceções

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.

Para criar nossa própria classe de exceção ela precisa herdar 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 no exemplo.

Sobrepondo Métodos e Exceções



Ao sobrepor métodos com throws, o método deve lançar a mesma exceção ou um de suas subclasses e não pode ser adicionado tipos diferentes.



Java Orientado a Objeto

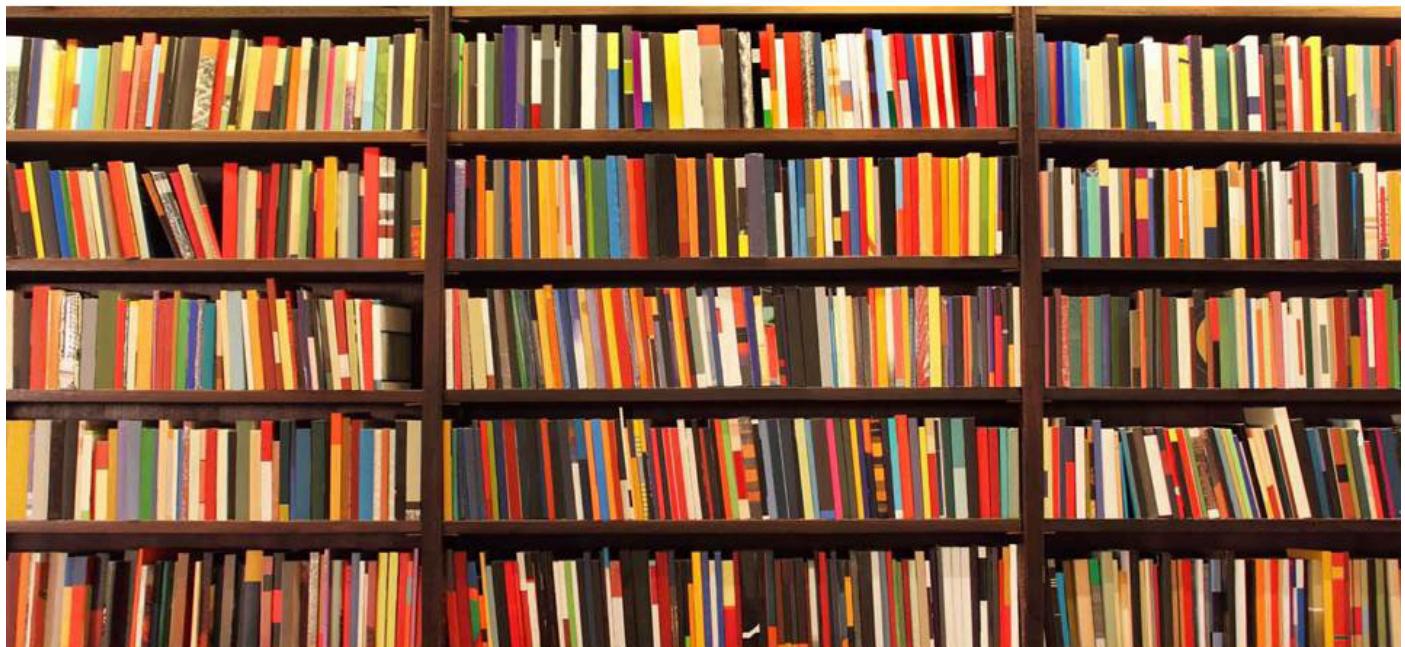
Quando sobreponemos um método que contenham a cláusula throws na sua definição, temos que tomar o

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

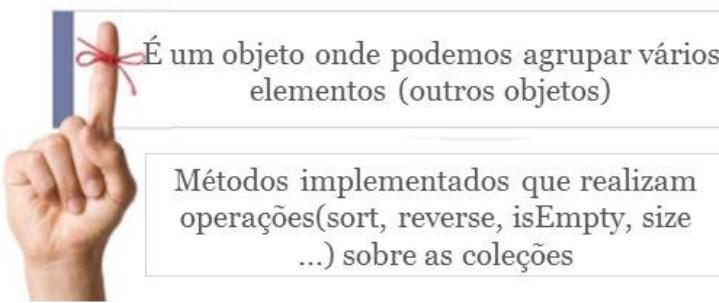
Java Orientado a Objetos

Collections



Java Orientado a Objeto

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 Orientado a Objeto

Java Collections

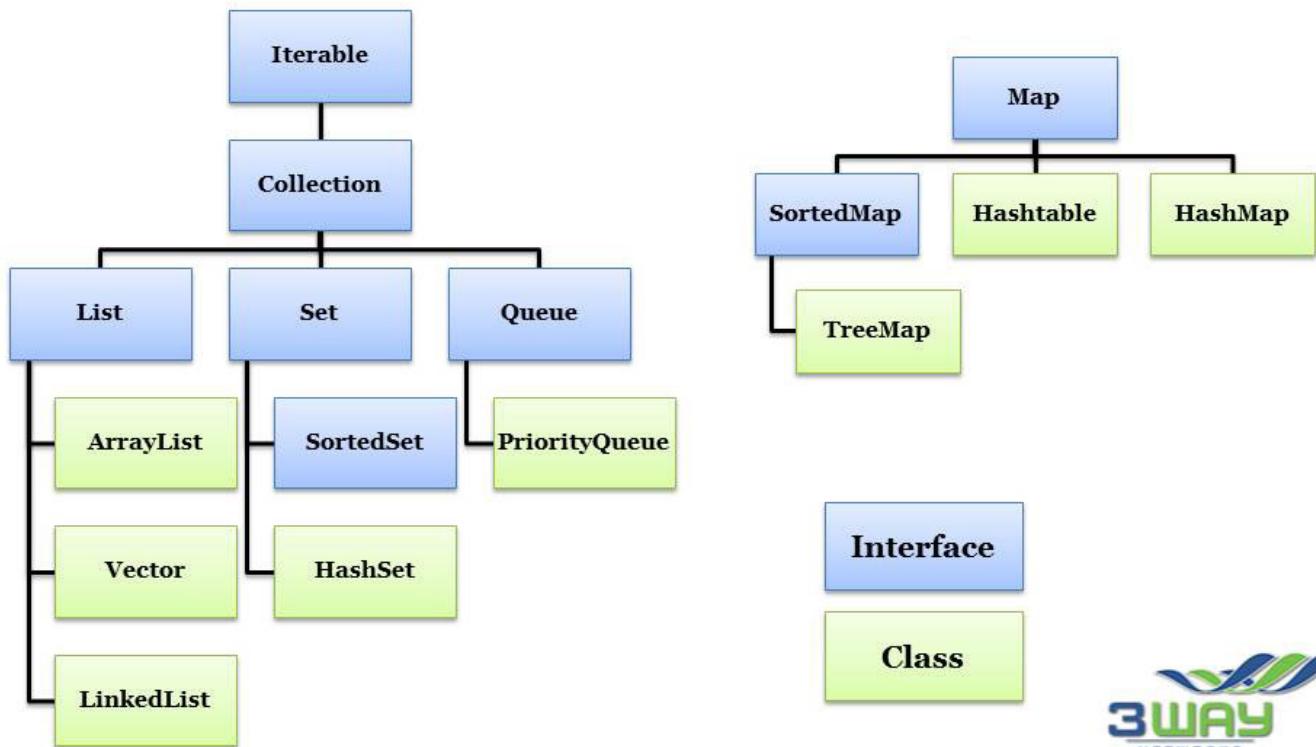
As estruturas de dados são muito utilizadas na programação em geral. Quando desenvolvemos algum programa, frequentemente utilizamos **arrays**. Em algoritmos complexos, listas (ligadas ou não), mapas, pilhas e filas são essenciais.

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 métodos 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 Orientado a Objeto

Java Collections – Hierarquia

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**
- **Não-organizada:** **HashSet**, **TreeSet**, **PriorityQueue**, **HashMap**, **HashTable**, **TreeMap**

- **Ordenada:** TreeSet, PriorityQueue, TreeMap
- **Não-ordenada:** HashSet, LinkedHashSet, ArrayList, Vector, LinkedList, HashMap, HashTable, LinkedHashMap.

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

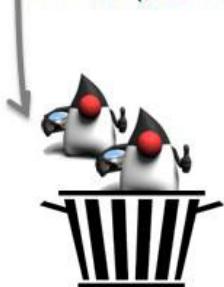
- **Set:** Não aceita itens duplicados, aceita nulos, não possui índice, rápido para inserir e pesquisar elementos;
- **HashSet:** esta é uma implementação concreta de Set não organizada, ou seja, os elementos são percorridos aleatoriamente, e também não é ordenada, não há regras de ordenação. Além disso, 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 duplicado 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.

- **HashTables**: 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 indices diferentes, é ordenado.

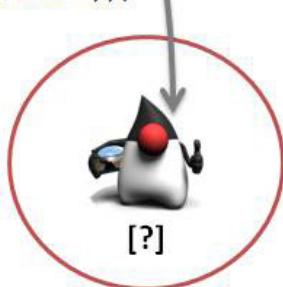
Interfaces Set e List

Interface Set

```
Set<String> set = new HashSet<>();
set.add(new String("Duke"));
set.add(new String("Duke"));
set.add(new String("Duke"));
```

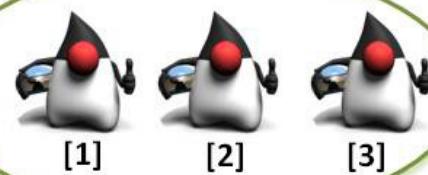


Coleções não ordenadas que não contém duplicidades



Interface List

```
List<String> list = new ArrayList<>();
list.add(new String("Duke"));
list.add(new String("Duke"));
list.add(new String("Duke"));
```



Coleções de classes ordenadas onde as duplicidades são permitidas.



Java Orientado a Objeto

Interface List

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. Veja alguns:

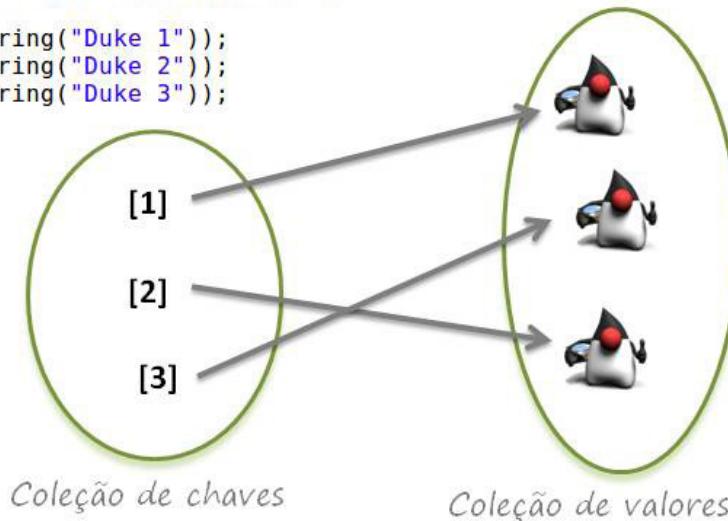
- **sort** – Ordena uma lista usando o algoritmo de ordenação Merge.
- **reverse** – inverte a ordem dos elementos.
- **replaceAll** – substitui todas as ocorrências de um valor especificado com outro.
- **fill** – substitui cada elemento em uma lista com o valor especificado.
- **copy** – copia a lista 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.

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. Você deve sobrescrever esses métodos.

Interface Map

```
Map<Integer, Object> map = new HashMap<>();
map.put(1, new String("Duke 1"));
map.put(2, new String("Duke 2"));
map.put(3, new String("Duke 3"));
```



Use `map.get([chave])` para recuperar objetos



Java Orientado a Objeto

Interface Map

Mapas (também chamados de dicionários) 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 8

```

default Spliterator<E> spliterator()
default Stream<E> stream()
default Stream<E> parallelStream()

```



Java Orientado a Objeto

Generics e Coleções Java

A interface `Collection<E>` é um exemplo de tipo Generic, o `<E>` é um tipo genérico a ser definido em tempo de compilação, veja:

```
Collection<Integer> collection = new ArrayList<Integer>;
```

o `E` será substituído pelo tipo `Integer` durante a compilação. O caractere `<?>`, visto na definição de alguns métodos define que o mesmo pode ser usado com qualquer objeto.

Para você ver como Generics ajuda a melhorar nosso código considere um objeto `ArrayList`. Um objeto

`ArrayList` possui a capacidade de armazenar referências de objetos, elementos de qualquer tipo de classe. Uma instância de `ArrayList`, entretanto, sempre te forçava a realizar um **casting** nos objetos que recuperamos a partir da coleção. Considere a seguinte linha de instrução:

```

ArrayList umArrayList = new ArrayList();
umArrayList.add("objeto");
String umaString = (String) umArrayList.get(0);

```

quando você invoca o método **add()** a referência da string “**objeto**” sofre um *casting* para o tipo *Object*, para recuperar a referência do objeto armazenado usamos o método **get()**, mas o método retorna uma referência do tipo *Object* daí você vai precisar fazer **casting** da referência para o tipo *String*. Agora note a versão utilizando Generics, não há **casting**.

```
ArrayList<String> umArrayList = new ArrayList<String>();  
umArrayList.add("objeto");  
String umaString = umArrayList.get(0);
```

Neste exemplo o tipo Genérico é *ArrayList<E>*, onde **E** é tipo da classe aceita pela coleção, no caso um *String*. Em tempo de compilação o tipo é checado e a coleção agora fará restrição sobre quais tipos de objeto podem ser adicionados pelo método **add()**, assim temos a garantia de que objetos retornados pelo método **get()**, neste caso, sempre serão do tipo *String*, portanto você não irá precisar fazer **casting**.

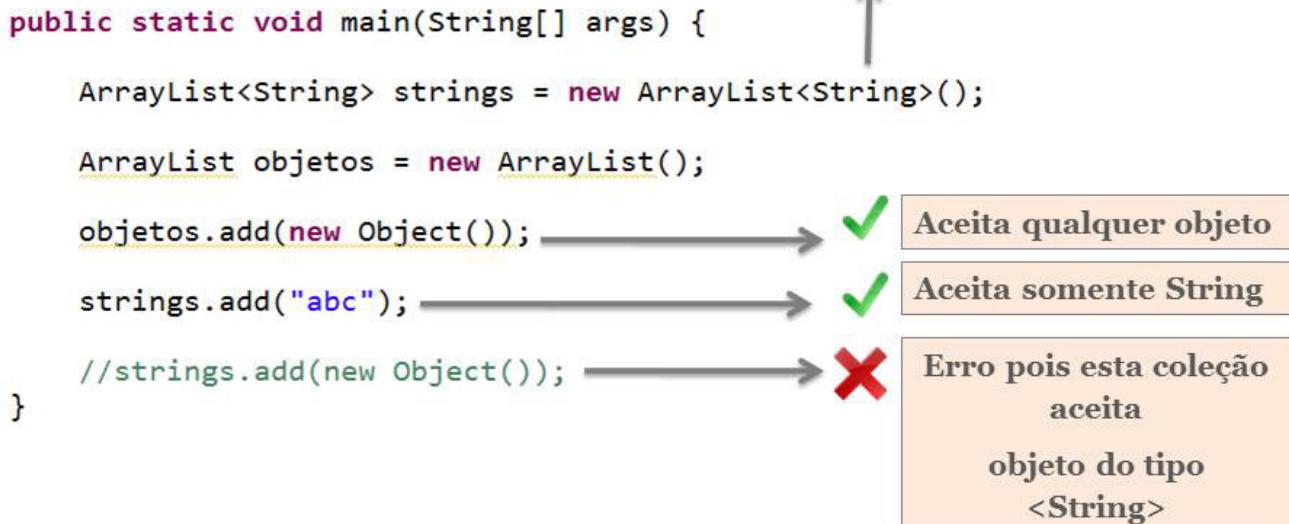
Os Principais métodos da Classe Collection

- **add(E e)** - Adiciona a coleção a um determinado objeto
- **addAll(Collection<? extends E> c)** – 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<?> c)** – Retorna true caso todos os elementos desta coleção estejam presentes em determinada coleção.
- **iterator<E>()** - Retorna o objeto de iteração com os elementos desta coleção.
- **remove(Object objeto)** – Remove o objeto da coleção
- **removeAll(Collection<?> c)** – 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.

A partir de Java 8, foram adicionados os métodos default:

- **default Stream<E> parallelStream()** - retorna um fluxo possivelmente paralelo tendo esta coleção como fonte;
- **default Stream<E> Stream()** - retorna um fluxo seqüencial tendo esta coleção como fonte;
- **default Spliterator<E> spliterator()** - retorna um objeto que permite percorrer e particionar os elementos da coleção.

Generics e Coleções Java



Java Orientado a Objeto

Generics e Coleções Java

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

```

List<String> list = new ArrayList<String>();
list.add(new Integer(5));
list.add(new String("another word"));
list.add(new Pessoa("Fulano"));
  
```

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, você precisaria testar com *instanceof* para saber qual o tipo do objeto antes fazer o casting.

```

Pessoa pes = null;
Integer i = null;
  
```

```
for(int i=0; i < list.size(); i++){  
    Object obj = list.get(i);  
    if(obj instanceof Pessoa)  
        pes = (Pessoa) obj;  
    else if(obj instanceof Integer)  
        i = (Integer) obj;  
}
```

Geralmente, não nos interessa uma lista com vários tipos de objetos misturados; no dia-a-dia, usamos listas de objetos conhecidos. 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*, e qualquer erro já se apresenta na compilação.

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*:

```
List<String> list = new ArrayList<String>(); //até Java7
```

```
List<String> list = new ArrayList<>(); //a partir de Java7
```

Interface Iterator, Iterable

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



```

public interface Iterable<T> {
    Iterator<T> iterator();
    default void forEach(Consumer<? super T> action)
    default Spliterator<T> spliterator()
}

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();
}
  
```

A partir de Java 8



Java Orientado a Objeto

Interface Iterator

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 Collection's:

- **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.

Interface Iterable

Collection é uma sub-interface de *Iterable*. Você obtém um objeto *Iterator* a partir do método:

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

A partir de Java 8, você pode usar o método **default forEach()**, para percorrer uma 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<String> iterator = strings.iterator(); while (iterator.hasNext()) { System.out.println(iterator.next()); }</pre>



Java Orientado a Objeto

Percorrendo Collections

São usadas diferentes estrutura de dados na implementação das coleções, por exemplo, `ArrayList` implementa uma coleção com base em `Array`, permitindo que você recupere um objeto diretamente usando o método `get(posição)`. O mesmo já não é tão interessante para um `LinkedList`, que usa uma estrutura de dados de lista duplamente encadeada, num lista encadeada o acesso aleatório tende a ser bastante ineficiente.

Nesses caos há duas maneiras básicas de percorrer coleções:

1. Usando o laço **for-enhanced** (ou **for-each**)
2. **Iteradores**

Um `Iterator` precisa invocar `hasNext()` antes de `next()` para ter certeza de que ainda há um próximo elemento a ser capturado, caso contrário seria lançada uma exceção.

O For-each é um ciclo for, mas que é adaptado para utilização em `Collections`. Ele serve para percorrer todos os elementos de qualquer `Collection` contida na API Collections. No exemplo, `strings` é a fonte contendo os elementos, cada ciclo do for-each captura o objeto na seqüência e salva sua referência na variável local `str`

Java 8 – método forEach

```

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

strings.forEach(new Consumer<String>() {
    @Override
    public void accept(final String str) {
        System.out.println(str);
    }
});

strings.forEach((str) -> System.out.println(str));

strings.forEach(System.out::println);

```

Usando forEach de forma imperativa com anonymous inner class

Usando forEach com Lambada

Usando forEach com Lambada, da forma mais reduzida, com métodos de referência



Java Orientado a Objeto

Operações de Agregação

A partir de Java 8, o método indicado para percorrer uma coleção é obter uma stream e realizar uma operação de agregação sobre ela. Operações de agregação são usadas em conjunto com expressões lambda tornando sua programação mais poderosa usando menos linha de código.

Compare os exemplos, na primeira versão invocamos o método `forEach()` e passamos um objeto anônimo, na segunda versão é usada uma expressão lambda e por final você pode simplesmente informar um método de referência.

Você pode usar operações de agregação para filtrar, ordenar e iterar sobre a coleção, por exemplo, imprimir somente string que terminam com caractere “1”.

```
collection.stream()
```

```
.sorted()
```

```
.filter(e -> e.endsWith("1"))
```

```
.forEach(e -> System.out.println(e));
```

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 Orientado a Objeto

Classificando Coleções *Collections.sort*

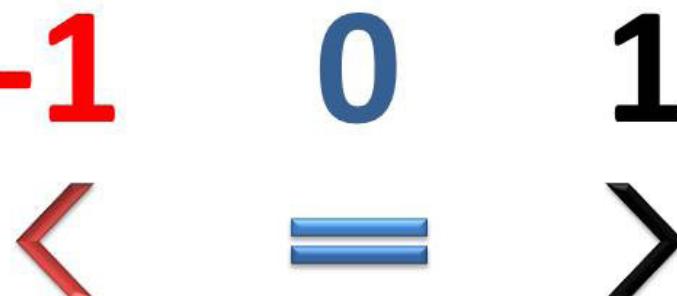
Vimos anteriormente que as listas são percorridas de maneira pré-determinada de acordo com a inclusão dos itens. Mas muitas vezes queremos **percorrer** a nossa lista de maneira **ordenada**, classificando os elementos de acordo com nossa necessidade.

A classe **Collections**(com s no final) nos permite **ordenar** coleções, através do um método estático **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 Orientado a Objeto

Interface Comparable

Essa interface fornece o método, **compareTo(T o)** , você deverá sobrescrevê-lo para que defina a ordem de comparação da sua classe.

O método **compareTo()** deverá ser implementado. Ele retorna:

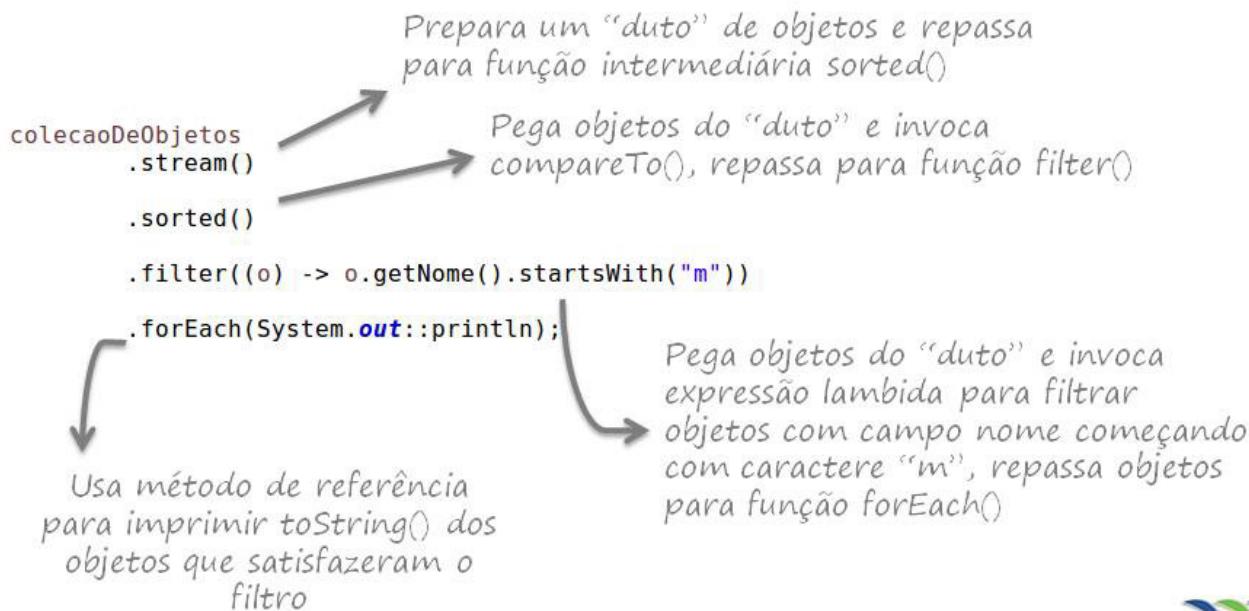
- Um inteiro **menor que zero** se objeto atual for “menor” que o recebido como parâmetro
- Um inteiro **maior que zero** se objeto atual for “maior” que o recebido como parâmetro
- **Zero** se objetos forem iguais

Exemplo:

```
public int compareTo(Point other) {  
    //Java7 sem risco de overflow  
    int diff = Integer.compare(x, other.x);  
    if (diff != 0) return diff;  
    return Integer.compare(y, other.y);  
}
```

Java 8 – Collections e Streams

A classe **java.util.Stream**, representa uma seqüência de elementos. **Streams** são como “dutos” jorrando objetos, você pode aplicar uma ou mais operações



Java Orientado a Objeto

Pipelines e Streams

Um *pipeline* é uma sequencia de operações de agregação, como no exemplo. O retorno da invocação do método **stream()** é utilizado pelo próximo método **sorted()**, que fará a ordenação dos elementos da coleção, e seu retorno será enviado para o próximo método **filter()**, que executa a expressão *lambda* associada, que por fim retorna para o método **forEach()** que por sua vez usa o método estático de referência **System.out::println()**, enviado o resultado de **toString()**, para saída padrão, de cada elemento após o filtro. Compare com o código que usa um laço **for-each**:

```

Collections.sort(colecaoDeObjetos);

for(String str: colecaoDeObjetos){

    if(str.startsWith("m")){
        System.out.println(str);
    }
}
  
```

Um *pipeline* contém três componentes:

- Uma fonte: esta pode ser uma coleção, um array, um I/O channel, ou uma função de geração de *streams*.
- Stream<String> s = Stream.of("seja", "agil", "com", "stream");
- Zero ou mais operações intermediárias: operações intermediárias produzem novas streams, tais como **sorted()**, **filter()**, **map()**.
- Um stream é uma sequencia de elementos. Diferentemente de uma coleção, streams não são estrutura de armazenamento de dados. Ao invés disso, streams transportam objetos de uma fonte através do pipeline.
- Operação terminal: são operações como o **forEach()**, que não produzem stream com resultado. Exemplos de operações terminais são **forEach()**, **anyMatch()**, **allMatch()**, **noneMatch()**, **count()**, **reduce()**.

Java Orientado a Objetos

Lendo e Escrevendo Arquivos



Java Orientado a Objeto

Console I/O

```
***** CUNIT CONSOLE - MAIN MENU *****
(C)Run all, (<S>elect suite, <L>ist suites, Show <P>failures, <Q>uit
Enter Command : r
Running Suite : Suite_success
  Running test : successful_test_1
  Running test : successful_test_2
  Running test : successful_test_3
WARNING - Suite init failed for Suite_init_failure.
Running test : successful_test_4
  Running test : failed_test_2
  Running test : failed_test_1
WARNING - Suite cleanup failed for Suite_clean_failure.
Running Suite : Suite_mixed
  Running test : successful_test_2
  Running test : failed_test_4
  Running test : failed_test_2
  Running test : successful_test_4
--Run Summary: Type Total Ran Passed Failed
suites   1      4     3    1    2
tests    13     18     7    3
asserts 10     10     7    3
***** CUNIT CONSOLE - MAIN MENU *****
(C)Run all, (<S>elect suite, <L>ist suites, Show <P>failures, <Q>uit
Enter Command :
```

```
import java.io.Console;
public class Teste {
    Console con = System.console();
}
```



System.in



System.out



Java Orientado a Objeto

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 Orientado a Objeto

Usando a classe Scanner

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.

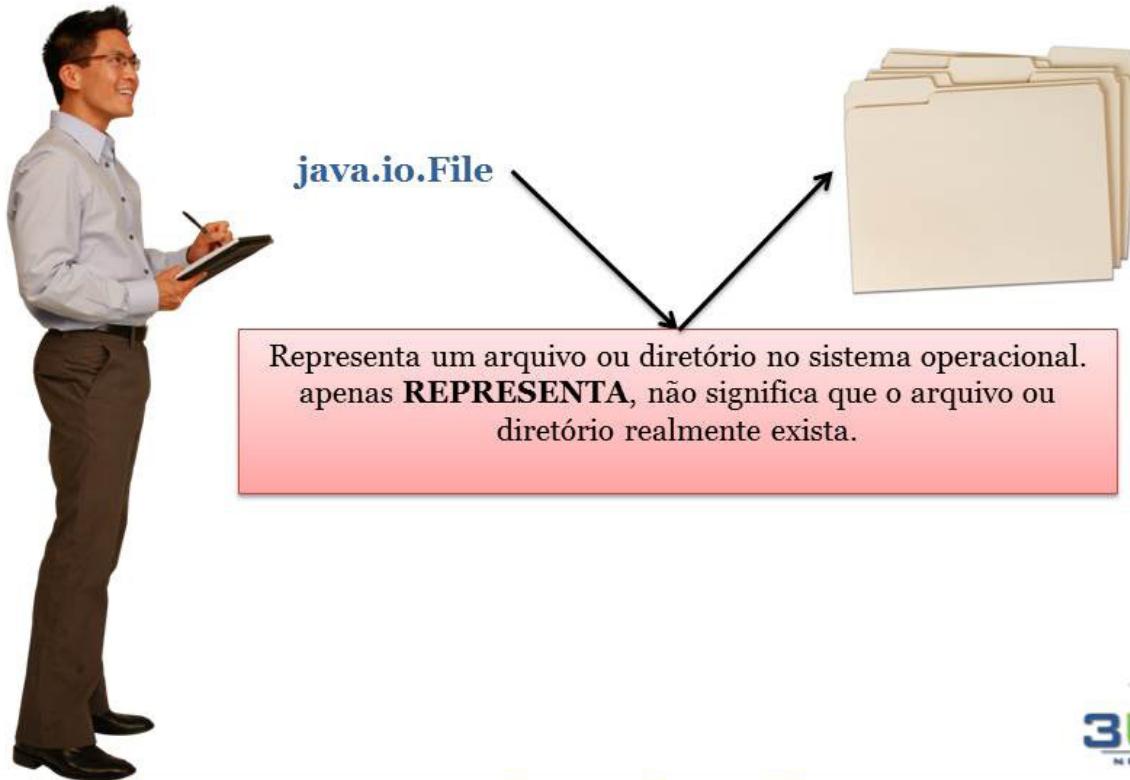
A classe Scanner é do pacote `java.util`. Ela possui métodos muito úteis para trabalhar com Strings, em especial, diversos métodos já preparados para pegar números e palavras já formatadas através de expressões regulares.

Para utilizar a classe Scanner em uma aplicação Java deve-se proceder da seguinte maneira:

[1]	importar o pacote <code>java.util</code> : <code>import java.util.Scanner;</code>
[2]	Instanciar e criar um objeto Scanner: <code>Scanner ler = new Scanner(System.in);</code>
[3]	Lendo valores através do teclado:
[3.1]	Lendo um valor inteiro: <code>int n;</code> <code>System.out.printf("Informe um número para a tabuada: ");</code> <code>n = ler.nextInt();</code>

[3.2]	Lendo um valor real: <u>float</u> preco; System.out.printf("Informe o preço da mercadoria = R\$ "); preco = ler.nextFloat();
[3.3]	Lendo um valor real: <u>double</u> salario; System.out.printf("Informe o salário do Funcionário = R\$ "); salario = ler.nextDouble();
[3.4]	Lendo uma String: <u>String</u> s; System.out.printf("Informe uma cadeia de caracteres:\n"); s = ler.nextLine();

java.io.File



Java Orientado a Objeto

Java.io.File

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.

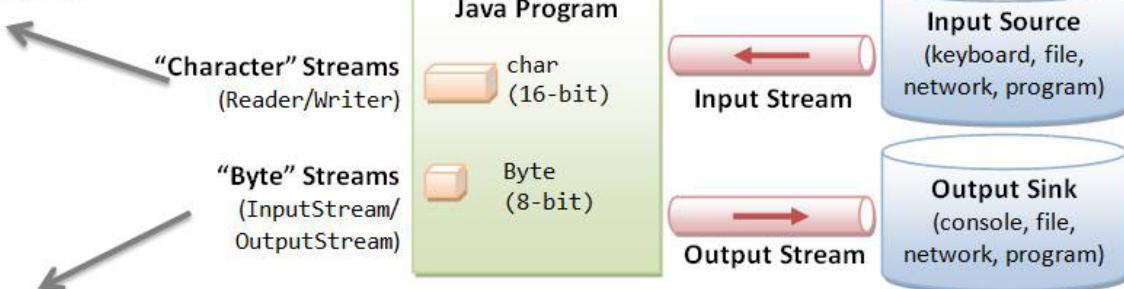
Alguns métodos da classe File:

- 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

I/O Stream

Usado para I/O arquivos de
Texto Classes InputStream
OutputStream



Usado para I/O arquivos de
binários Classes
InputStream OutputStream

Internal Data Formats:

- Text(char): UCS-2
- int, float, double, etc.

External Data Formats:

- Text in various encodings (US-ASCII, ISO-8859-1, UCS-2, UTF-8, UTF-16, UTF-16BE, UTF16-LE, etc.)
- Binary (raw bytes)



Stream, é uma cadeia de bits, como uma mangueira jorrando 'bits'! Lê escreve um caractere por vez.



Java Orientado a Objeto

Stream I/O no Java I/O Padrão

Uma Stream é um fluxo sequencial e contínuo de dados (igual água ou óleo passando por duto).

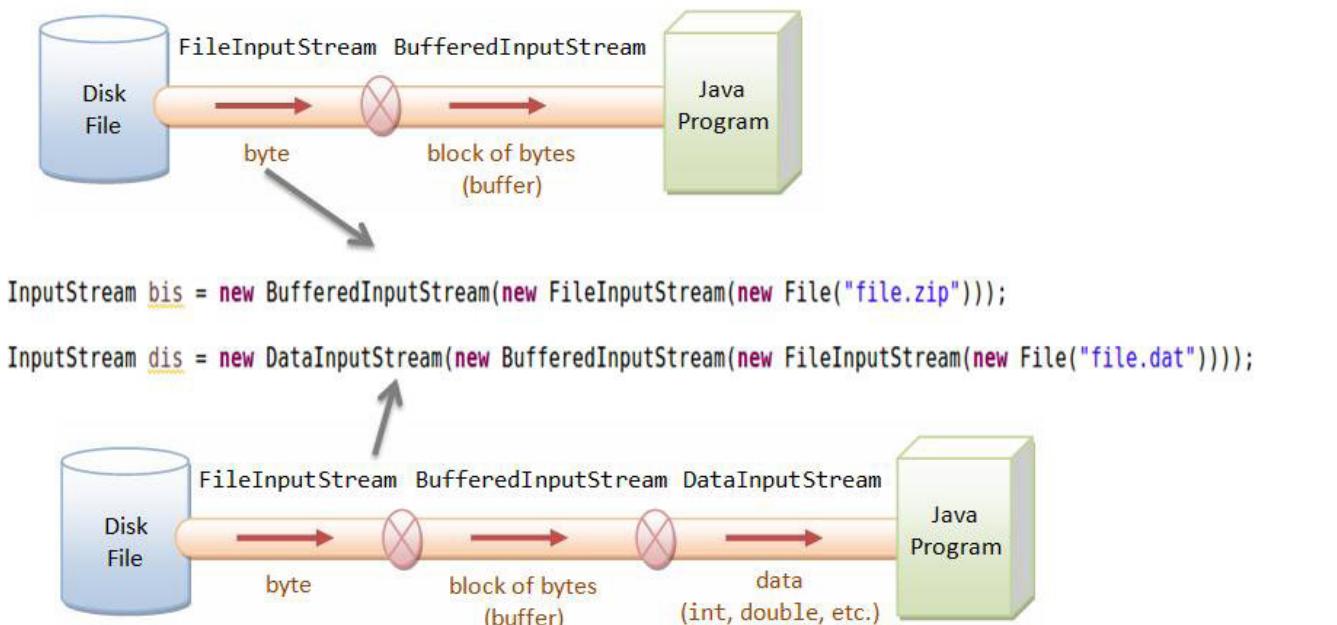
As classes abstratas `InputStream` e `OutputStream` definem, respectivamente, o comportamento padrão dos fluxos em Java: em um fluxo de entrada, é possível ler bytes e, no fluxo de saída, escrever bytes.

As classes abstratas `Reader` e `Writer` definem, respectivamente, fluxos de entrada de caractere e fluxo de saída caractere.

Operações de I/O Stream envolve três partes:

1. Abrir um stream de entrada/saída associado com um dispositivo físico (rede, arquivo, console/teclado, outro programa, etc), pela construção de uma instância de I/O stream apropriado.
2. Ler a partir do stream de entrada até chegar ao “fim-de-stream”, ou escrever para o stream de saída aberto.
3. Fechar o stream entrada/saída.

Encadeando I/O Stream



Java Orientado a Objeto

Encadeamento de I/O Streams

Objetos de I/O Stream normalmente são encadeados com outro objeto de stream que fazem a bufferização, filtragem ou conversão de bytes em tipos de dados primitivo java. Fazendo uma analogia, você está colocando um “tubo” de stream dentro de outro.

No exemplo, nós fizemos os dados de `FileInputStream` serem “derramados” dentro de um `BufferedInputStream` - fazendo um buffer para melhorar o desempenho da leitura - e depois “derramamos” o conteúdo de `BufferedInputStream` dentro de um `DataInputStream` - para conseguirmos que as seqüências de bytes que estão entrando já cheguem formatados em um tipo de dado java.

Os métodos de leitura/escrita `InputStream/OutputStream` são projetados para ler/escrever um único byte de dados em cada chamada. Isto é muito ineficiente, porque cada chamada é manipulada em operação de baixo nível do sistema operacional. *Buffering*, é fundamental para melhorar o desempenho das operações de leitura/escrita, na biblioteca padrão de I/O do Java você terá que encadear (direta ou indiretamente) seus objetos de stream para melhorar a performance.

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!");
    }
}

```

Um arquivo, caminho absoluto
Encadeando para FileWriter
Encadeando para BufferedWriter



Java Orientado a Objeto

FileWriter e BufferedWriter

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()`.

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!");
    }
}

```

Um arquivo, caminho absoluto
Encadeando para FileReader
Encadeando para BufferedReader



Java Orientado a Objeto

FileReader e BufferedReader

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().

Classe de java.io	Estende de	Argumento dos construtores	Principais métodos
FileReader	Reader	String File	read() close()
BufferedReader	Reader	Reader	read () readLine()

NIO.2 - Path

Path pode ser um arquivo no diretório atual, caminho relativo ao programa

```
Path p1 = Paths.get("in.txt");
Path p2 = Paths.get("c:\\\\projetos\\\\java\\\\Hello.java");
Path p3 = Paths.get("/use/local");
```

Path pode ser um arquivo, caminho absoluto, no windows use caractere de escape \\

Path pode ser um diretório

JAVA 7 NIO.2, mais simples, Buffered, sem bloqueio de I/O. A classe `java.nio.Path`, especifica a localização de um arquivo, ou diretório, ou link simbólico.
Substitui `java.io.File`



Java Orientado a Objeto

JAVA 7 NIO.2 - I/O

Java 7, trouxe melhorias para processos de I/O, através de uma nova API dentro do pacote `java.nio.file`.

A classe Path substitue a classe File, melhorando o suporte ao tratamento de arquivos e suas permissões. Um instância de Path representa a localização de um arquivo , um diretório ou um link simbolico.

Classes de suporte

Paths - para criar um `Path` você deve usar a classe **Helper Paths** e o método estático `get()` , esse método recebe uma string com a URI do recurso e retorna o Path associado.

Files - a classe contém métodos exclusivamente estáticos para operações de arquivo, diretório e links simbólicos, tais como criar, apagar, ler, escrever, copiar, mover, etc. As operações de leitura e escrita são utilizadas com arquivos pequenos para os quais você deseja fazer uma leitura completa em um único passo.

NIO.2 – I/O Streams

```

InputStream in = Files.newInputStream(path);
OutputStream out = Files.newOutputStream(path);
Reader reader = Files.newBufferedReader(path);
Writer writer = Files.newBufferedWriter(path);

try (OutputStream out = new BufferedOutputStream(
    Files.newOutputStream(p, StandardOpenOption.CREATE,
        StandardOpenOption.APPEND))) {
    out.write(data, 0, data.length);
} catch (IOException x) {
    System.err.println(x);
}

```

Compatibilidade
com Java I/O
Básico

Leitura ou escrita
orientada por
streams, um
caractere por vez.

*Use com arquivos Grandes, para usar streams
compatíveis com java IO*



Java Orientado a Objeto

Compatibilidade com Java I/O Padrão

A classe de suporte *Files* te permite obter objetos de I/O Stream, biblioteca padrão de I/O java.

O exemplo mostra o processo de escrita, usando as facilidades das classes *Path* e *Files* ao invés de usar o encadeamento de *FileInputStream*, *FileOutputStream*, *BufferedReader* ou *BufferedWriter*.

Para abrir um arquivo para leitura, você pode usar o método *newInputStream (Path, OpenOption ...)*. Esse método retorna um fluxo de entrada sem buffer para a leitura de bytes do arquivo.

Você pode criar um arquivo, anexar um arquivo, ou escrever para um arquivo usando o método *newOutputStream (Path, OpenOption ...)*. Este método abre ou cria um arquivo para escrever bytes e retorna um fluxo de saída sem buffer.

Os métodos recebem um parâmetro opcional do tipo *OpenOption*. Se não houver opções abertas, e o arquivo não existir, um novo arquivo é criado. Se o arquivo existir, é truncado. Esta opção é equivalente a chamar o método com as opções *CREATE* e *TRUNCATE_EXISTING*.

NIO.2 – I/O Streams

```

String fileStr = "small_file.txt";
Path path = Paths.get(fileStr);
List<String> lines = new ArrayList<String>();
lines.add("Olá, 您好! Olá, 吃饱了没有?");

try {
    Files.write(path, lines, Charset.forName("UTF-8"));
} catch (IOException ex) {
    ex.printStackTrace();
}

byte[] bytes;
try {
    bytes = Files.readAllBytes(path);
    for (byte aByte: bytes) {
        System.out.printf("%02X ", aByte);
    }
    System.out.printf("%n%n");
} catch (IOException ex) { }

List<String> inLines;
try {
    inLines = Files.readAllLines(path, Charset.forName("UTF-8"));
    for (String aLine: inLines) {
        for (int i = 0; i < aLine.length(); ++i) {
            char charOut = aLine.charAt(i);
            System.out.printf("[%d] '%c'(%04X) ", (i+1), charOut, (int)charOut);
        }
        System.out.println();
    }
} catch (IOException ex) {}

```

Um arquivo no diretório atual, caminho relativo ao programa

Escreve dados para arquivo texto

Lê dados do arquivo como bytes

Lê dados do arquivo como caracteres UTF-8

Use com arquivos pequenos



Java Orientado a Objeto

Lendo e Escrevendo para Arquivos Pequenos

A classe *Files* faz o trabalho rápido de operações que são comuns com arquivos. Por exemplo, você

pode facilmente ler todo o conteúdo de um arquivo:

```
byte[] bytes = Files.readAllBytes(path);
```

Se você desejar ler com uma string, basta chamar:

```
String content = new String(bytes, StandardCharsets.UTF_8);
```

Mas se você deseja ler o arquivo como uma seqüência de linhas:

```
List<String> lines = Files.readAllLines(path);
```

Inversamente se você deseja escrever para o arquivo como uma string, chame:

```
Files.write(path,content.getBytes(StandardCharsets.UTF_8));
```

Você pode escrever um seqüência de linhas:

```
Files.write(path, lines);
```

Ou anexar uma seqüência de linhas ao final do arquivo:

```
Files.write(path, lines, StandardOpenOption.APPEND);
```

NIO.2 – I/O Channel

```

private void leia(Path path) {
    try (SeekableByteChannel sbc = Files.newByteChannel(path)) {
        ByteBuffer buf = ByteBuffer.allocate(64);
        while (sbc.read(buf) > 0) {
            buf.rewind();
            System.out.print(Charset.forName("UTF-8").decode(buf));
            buf.flip();
        }
    } catch (IOException e) {
        log.warning(e.toString());
    }
}

```

I/O de arquivos conectados a um channel, dados são lidos para o buffer

O método `rewind()` muda o ponteiro para inicio do buffer e o deixa pronto para leitura

O método `flip()` muda o ponteiro e permite ler dados a partir do buffer, use antes de escrever para arquivo

Use com arquivos Grandes. Channel lê um buffer por vez



Java Orientado a Objeto

Lendo e Escrevendo com Channels

Um stream I/O lê um caractere ou um byte por vez, channel I/O lê um buffer por vez. Imagine um channel com um tubo através do qual os dados são transferidos e buffers são as fontes ou destinos para onde os dados são transferidos. A interface `ByteChannel` fornece funcionalidade básicas `read` e `write`. Um `SeekableByteChannel` é um `ByteChannel` que possui a capacidade de fazer um acesso aleatório dentro do arquivo te permitindo mudar a posição de leitura/escrita dentro do arquivo.

Os métodos `rewind()`, `flip()`, da classe `ByteBuffer`, devem usados para reposicionar os ponteiros de controle de leitura/escrita do buffer. Use `rewind()`, para iniciar nova leitura completa do buffer. Use `flip()` antes de ler dados do buffer.

Há dois métodos para leitura ou escrita com channel I/O:

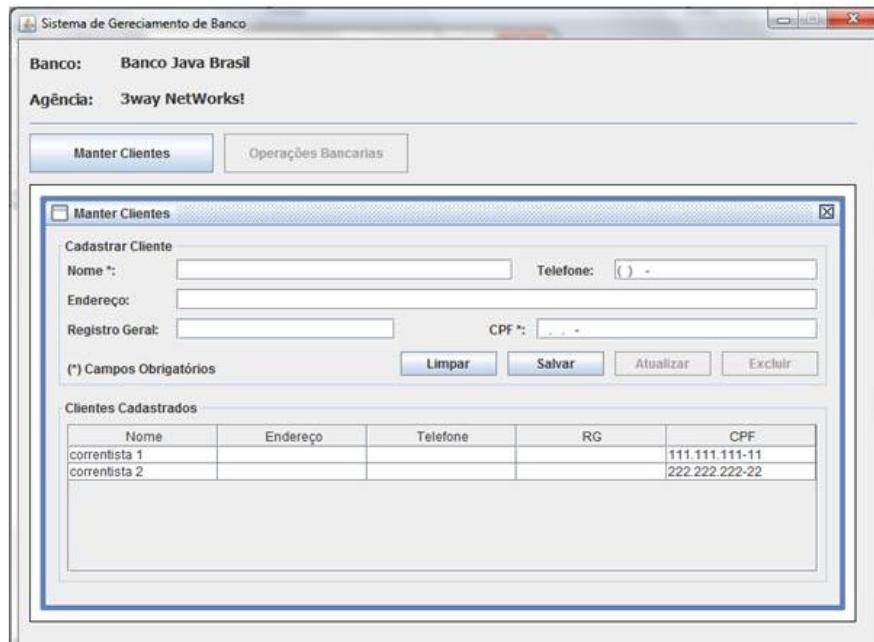
`newByteChannel(Path, OpenOption...)`

`newByteChannel(Path, Set<? extends OpenOption>, FileAttribute<?>...)`

os dois métodos possuem um parâmetro do tipo `java.nio.file.OpenOption` (`WRITE,APPEND,CREATE,etc`), os mesmo de `newOutputStream` e mais a opção `READ`, pois `SeekableByteChannel` dá suporte tanto de leitura como de escrita.

Java Orientado a Objetos

Construindo interfaces gráficas com Swing



Java Orientado a Objeto

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 Orientado a Objeto

AWT versus Swing

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 contém um “J” na frente, como em JButton por exemplo. Componentes AWT não contém 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.

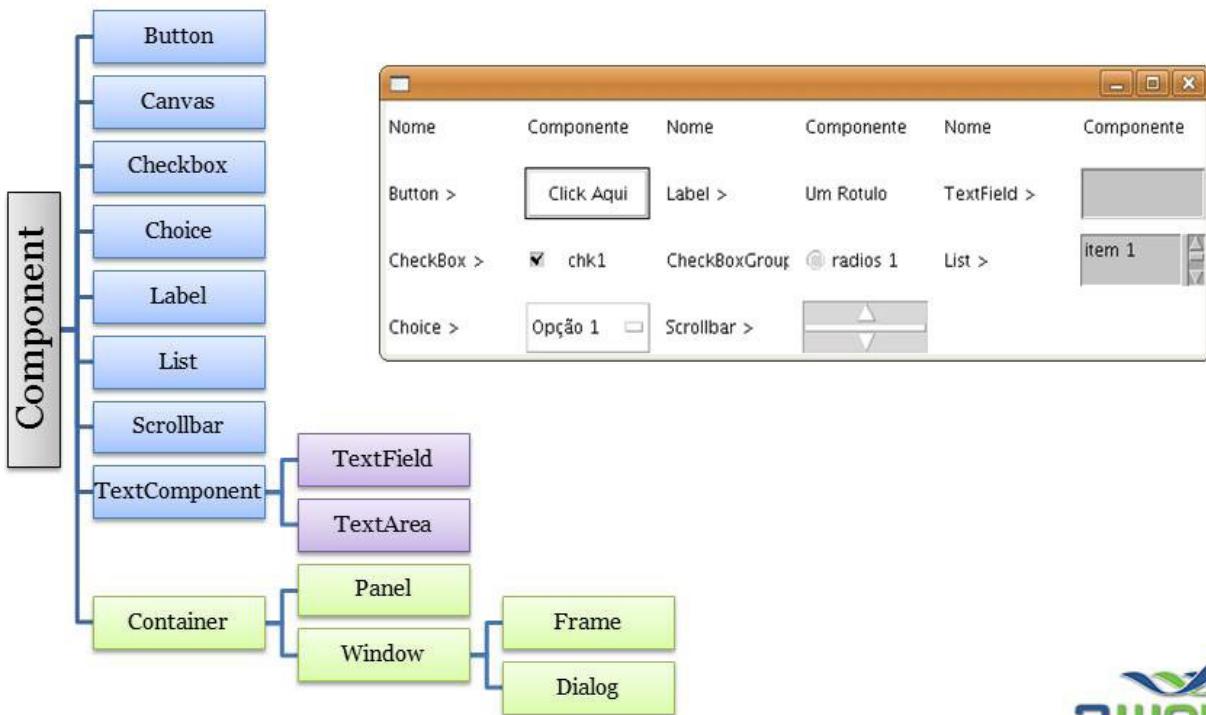
Swing

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 Orientado a Objeto

Componentes AWT

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**.

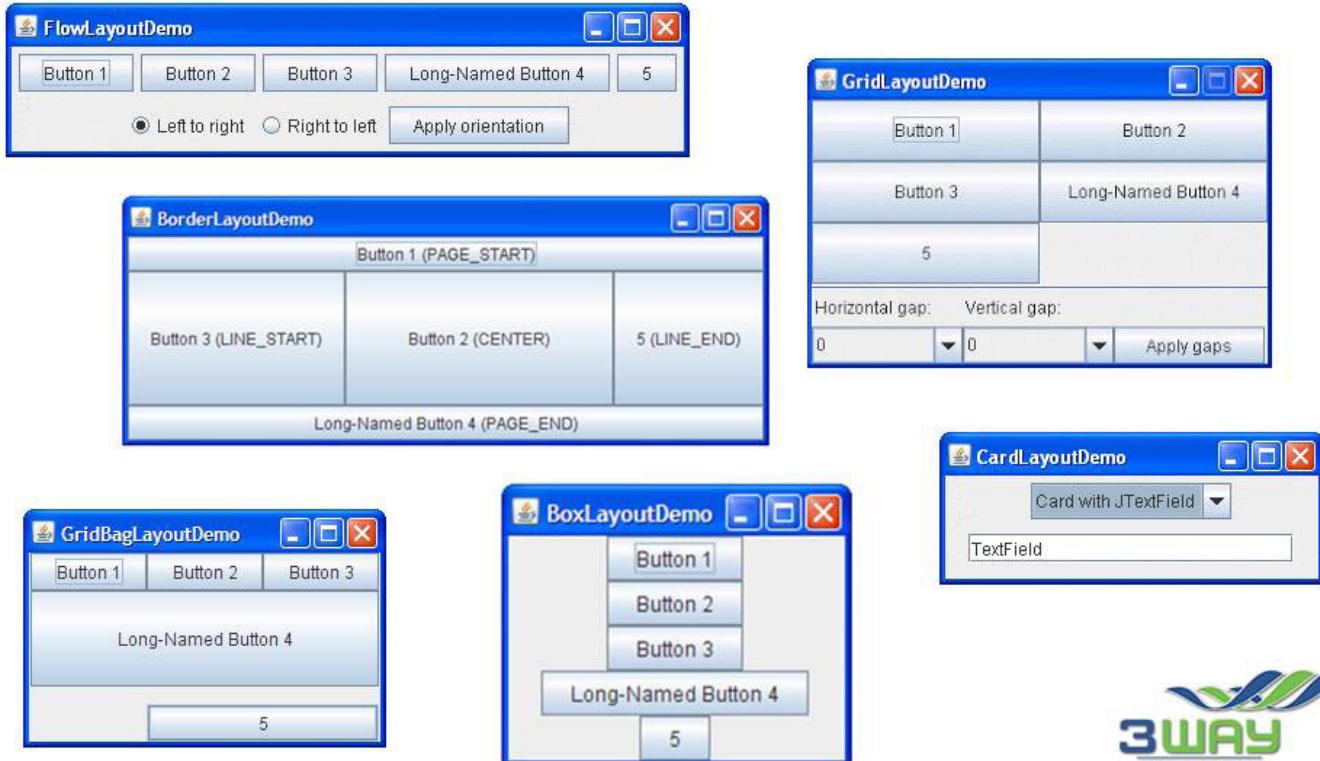
Fundamental Window Classes

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.

Classe AWT	Descrição
Component	<i>Uma classe abstrata para objetos que podem ser exibidos no console interagir com o usuário. A raiz de todas as outras classes AWT.</i>
Container	<i>Uma subclasse abstrata da classe Component. Um componente que pode conter outros componentes.</i>
Panel	<i>Herda a classe Container. Uma área que pode ser colocada em um Frame, Dialog ou Window. Superclasse da classe Applet.</i>

Window	Também herda a classe <i>Container</i> . 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: <i>Frame()</i> <i>Frame(String title)</i>

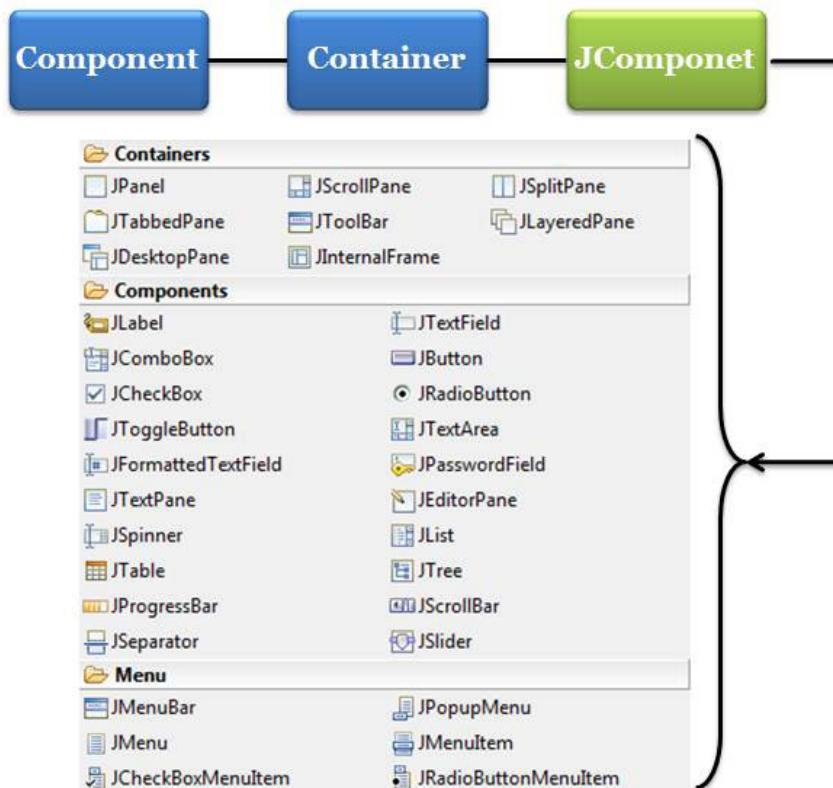
Gerenciadores de layout



Java Orientado a Objeto

Gerenciadores de Layout

Componentes Swing



Java Orientado a Objeto

Componentes Swing

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);
    }
}

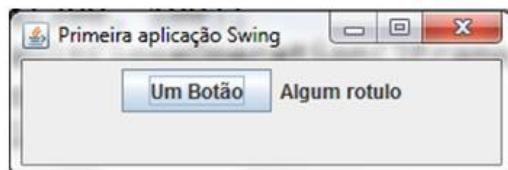
```

Um container é um agrupamento ou uma coleção de JComponents

Construtor

Container que recebe os componentes

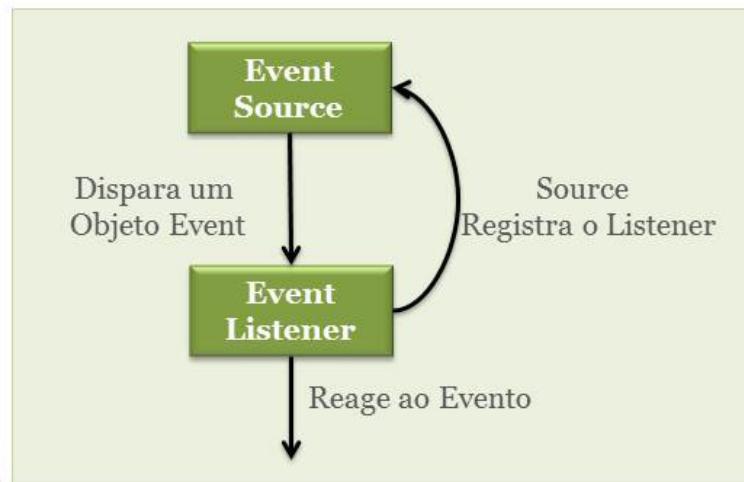
Inicia o frame e o deixa visível



Java Orientado a Objeto

Containers JFrame

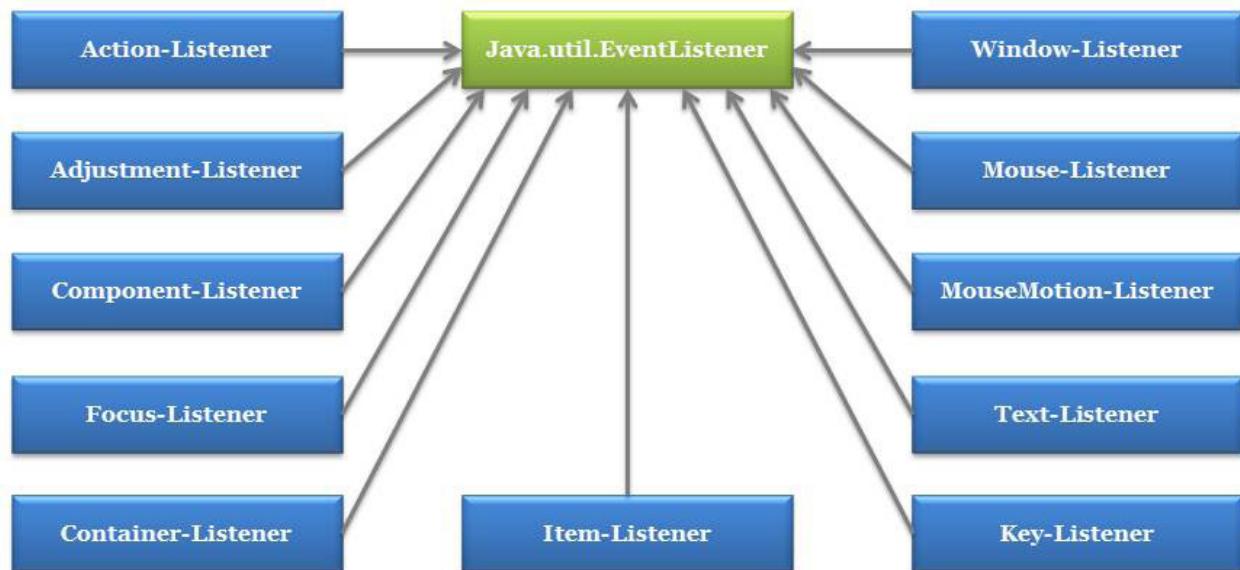
Manipulação de Evento



Java Orientado a Objeto

Manipulação de evento

Classes de Evento



Java Orientado a Objeto

Classes de Evento

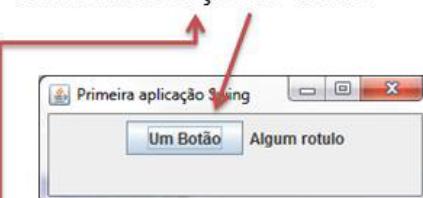
Criação de aplicações gráficas com Eventos

```

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);
    }
}

```

Declarando ação do botão



Evento depois do click



Java Orientado a Objeto

Criação de aplicações gráficas com eventos

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 Orientado a Objeto

Classes Adaptadoras

Apêndices

Java Orientado a Objetos

Tipos Genéricos



Java Orientado a Objeto

Java Orientado a Objetos

Tipos Genéricos



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



Java Orientado a Objeto

Tipos Genéricos

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> { // Contem o parâmetro <?>
    private T[] array; // atributo de tipo generic
    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 generico
    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

Declarando uma Classe utilizando Generics

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.

3. 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 sobre carregados forem idênticas para cada tipo de argumento, os métodos sobre carregados podem ser codificados mais compacta e convenientemente com um método genérico.

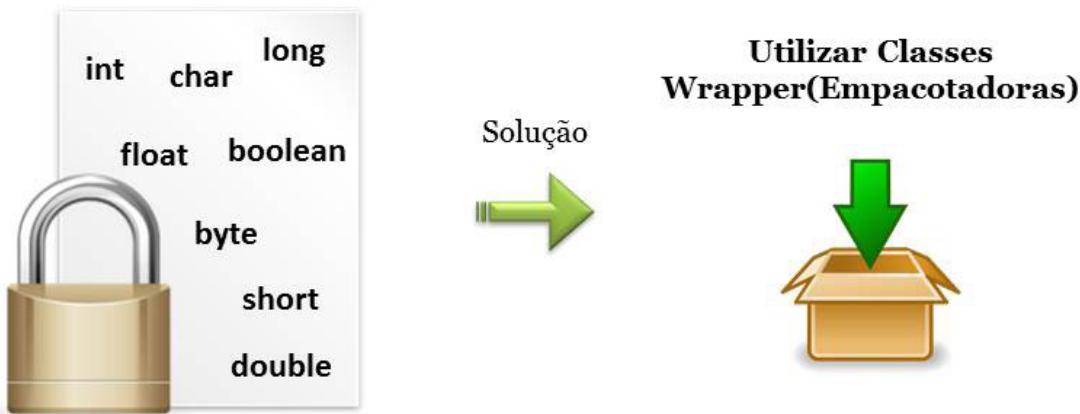
Você pode escrever uma única declaração de método genérico que pode ser chamada com argumentos de tipos diferentes.

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



Java Orientado a Objeto

Limitação “Primitiva”

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.

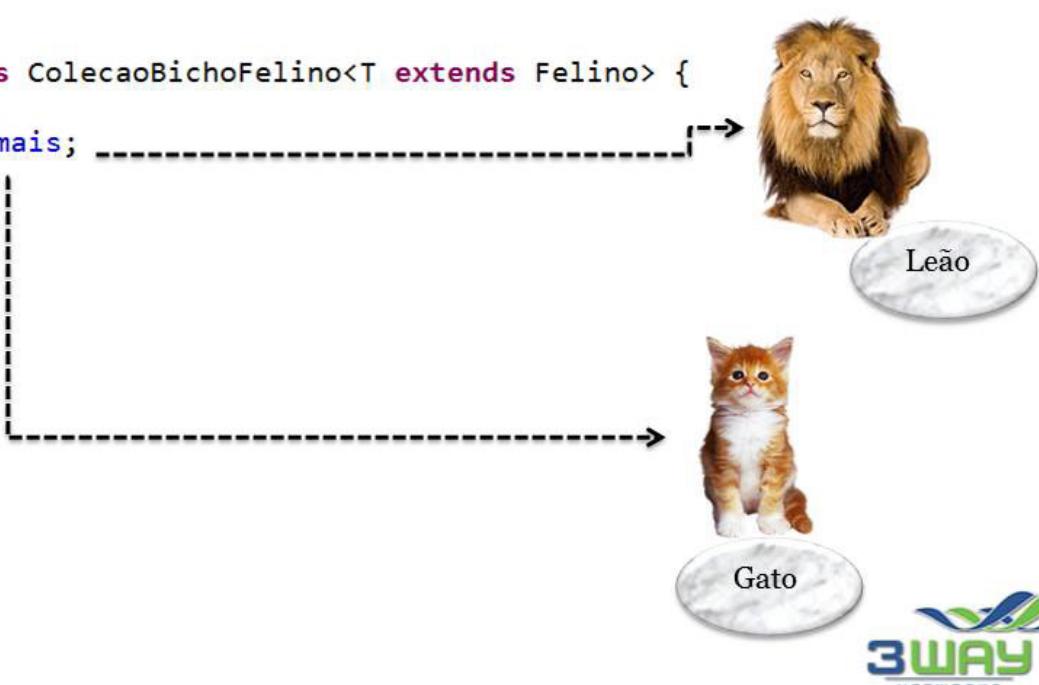
```
ArrayList <int> listaInt = new ArrayList <int>();
```

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

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

Limitando Genéricos

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



Java Orientado a Objeto

Limitando Genéricos

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 <?>

<? extends T>	←	<? super T>
Aceita T e todos os seus descendentes		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

Coringa <?>

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.

Threads

Pense em **threads** como processos do sistema operacional



Java Orientado a Objeto

Threads

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.

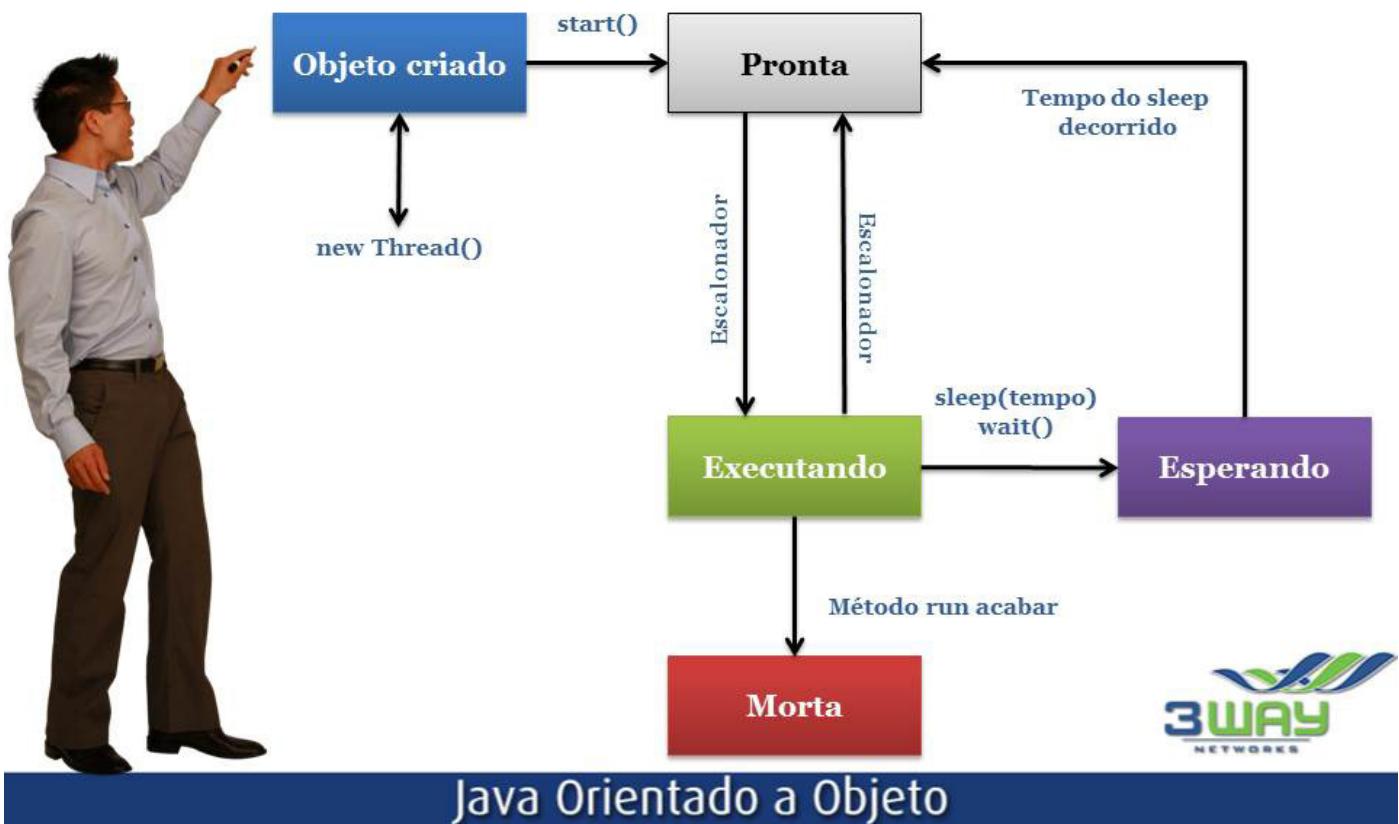
Em um programa só (um processo só), se queremos executar coisas em paralelo, normalmente falamos de **Threads**.

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



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

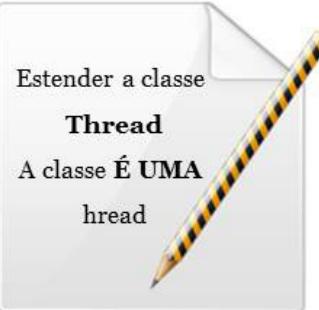
Uma Thread é finalizada quando acabar a execução do seu método run, e então ela vai para o estado Morta, onde o Sistema de Execução Java poderá liberar seus recursos e eliminá-la .

Verificando se Threads estão Executando/Pronta/Esperando ou Novas/Mortas

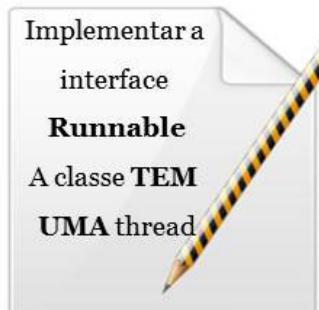
A classe Thread possui o método isAlive, o qual permite verificar se uma thread está nos estados **Executando/Pronta/Esperando** ou nos estados **Nova/Morta**. Quando o retorno do método for true a thread esta participando do processo de escalonamento e o retorno for false a thread está fora do processo 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 Orientado a Objeto

Criando Threads

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 linguagem 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

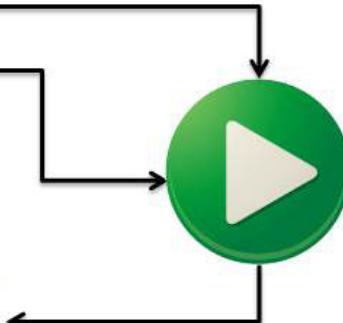
O principal fator a ser levado em consideração na escolha entre um dos dois métodos de criação de thread é que a linguagem Java não permite herança múltipla, assim quando uma classe que já for subclasse de outra precisar implementar thread é obrigatório que isso seja feito através da implementação da 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 Orientado a Objeto

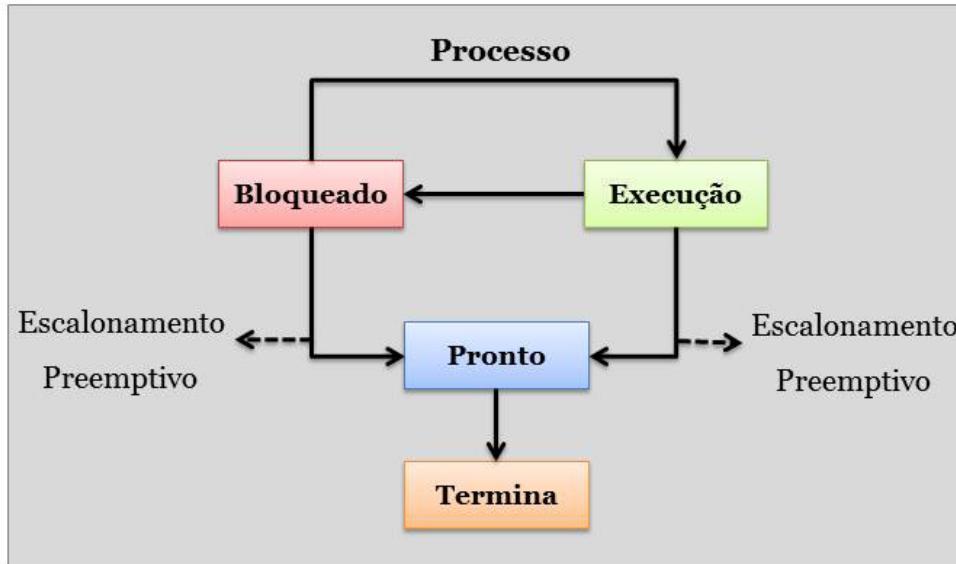
Iniciando Threads

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.

Escalonamento da Thread



O Escalonador seleciona um entre os processos em memória prontos para executar e aloca a CPU para ele



Java Orientado a Objeto

Escalonamento de Threads

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 escala threads baseado na sua prioridade.

As threads escalonáveis são aquelas que estão nos estados **Executando** ou **Pronta**, 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 Orientado a Objeto

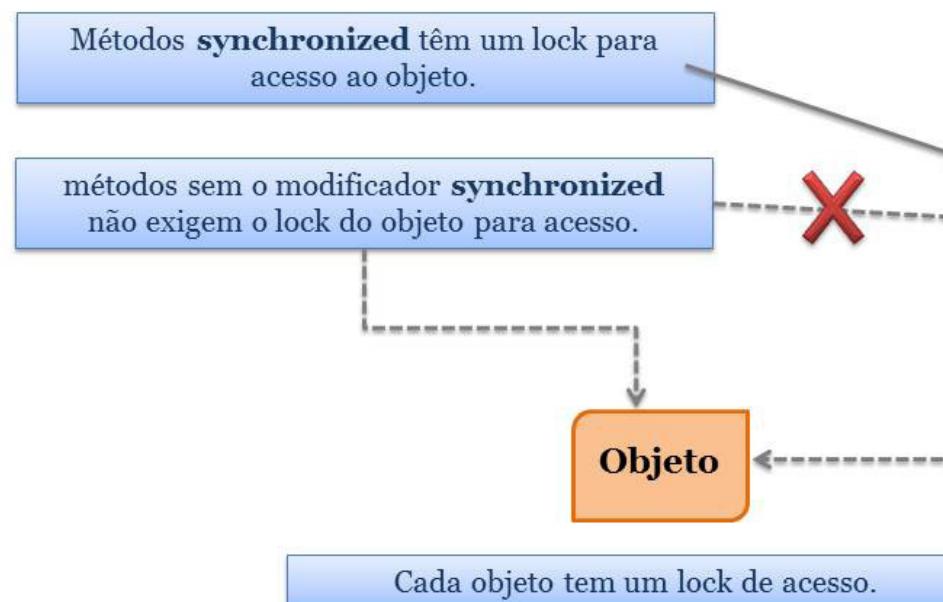
Propriedades de uma Thread

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:

1. Quando várias threads estiverem Prontas, aquela que tiver a maior prioridade será executada.
2. Quando existir várias threads com prioridades iguais, as mesmas serão escalonadas segundo o algoritmo Round-Robin de escalonamento.
3. 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.
4. Threads com prioridades mais baixas terão direito garantido de serem executadas para que situações de starvation não ocorram.

Sincronização



Java Orientado a Objeto

Sincronização

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.

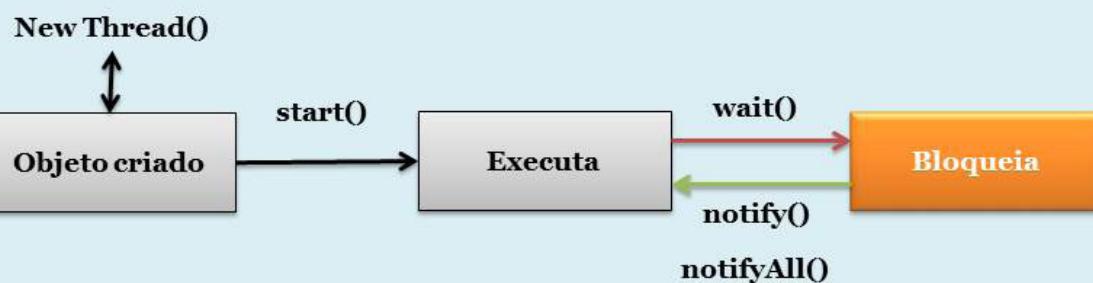
Bloqueando acesso Concorrente

A coordenação é feita com métodos da classe Object:

`wait()`

`notify()`

`notifyAll()`



Java Orientado a Objeto

Bloqueando acesso Concorrente

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 :

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.