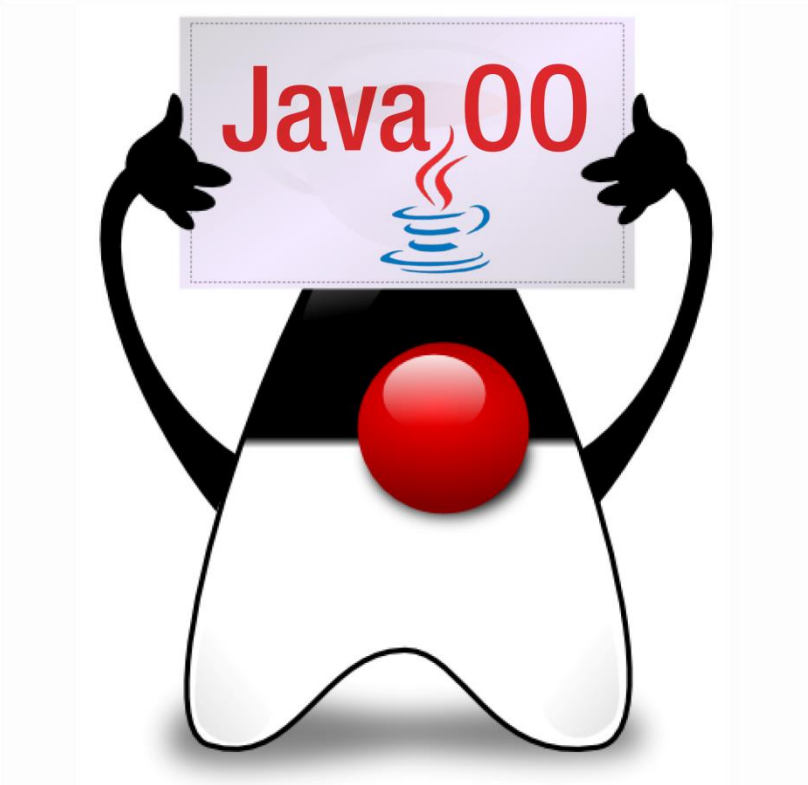
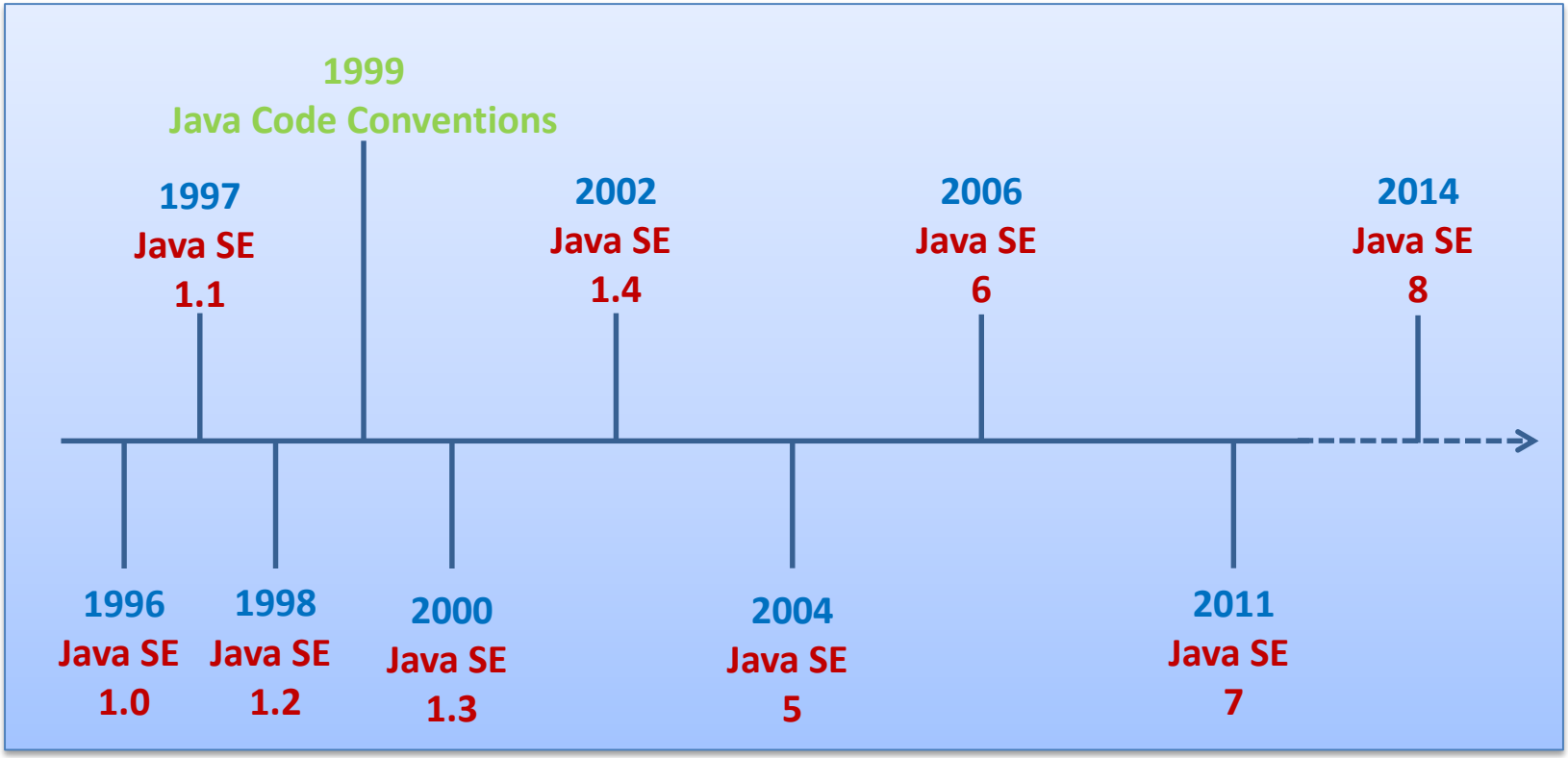
Java Orientado a Objeto
Introdução à linguagem Java



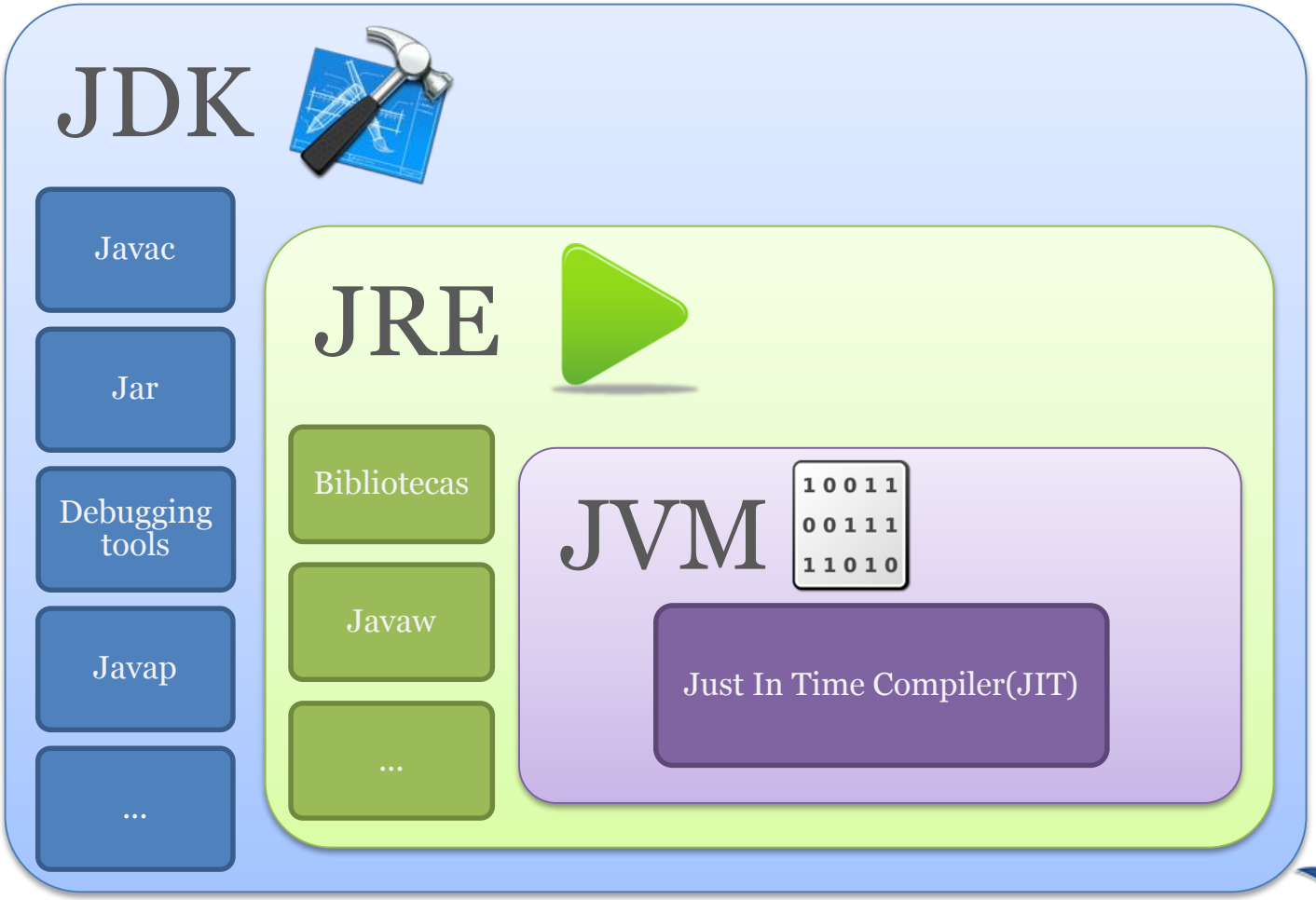
O que é Java?

Um pouco da História



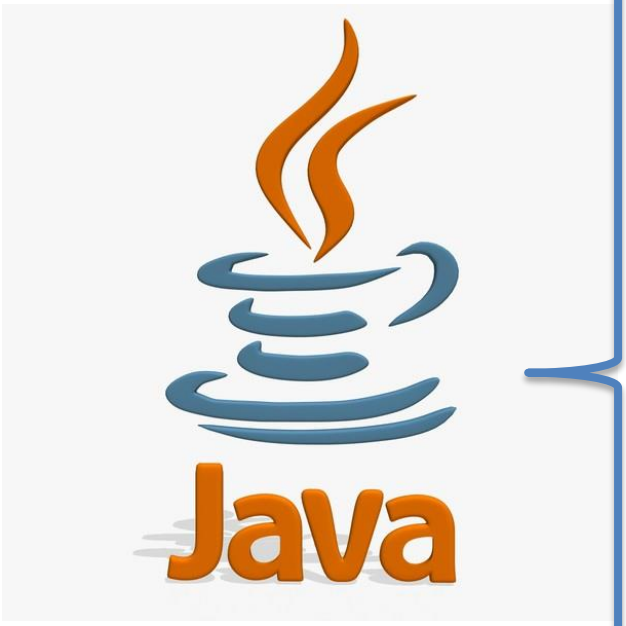
Onde encontro Java?

Plataforma Java



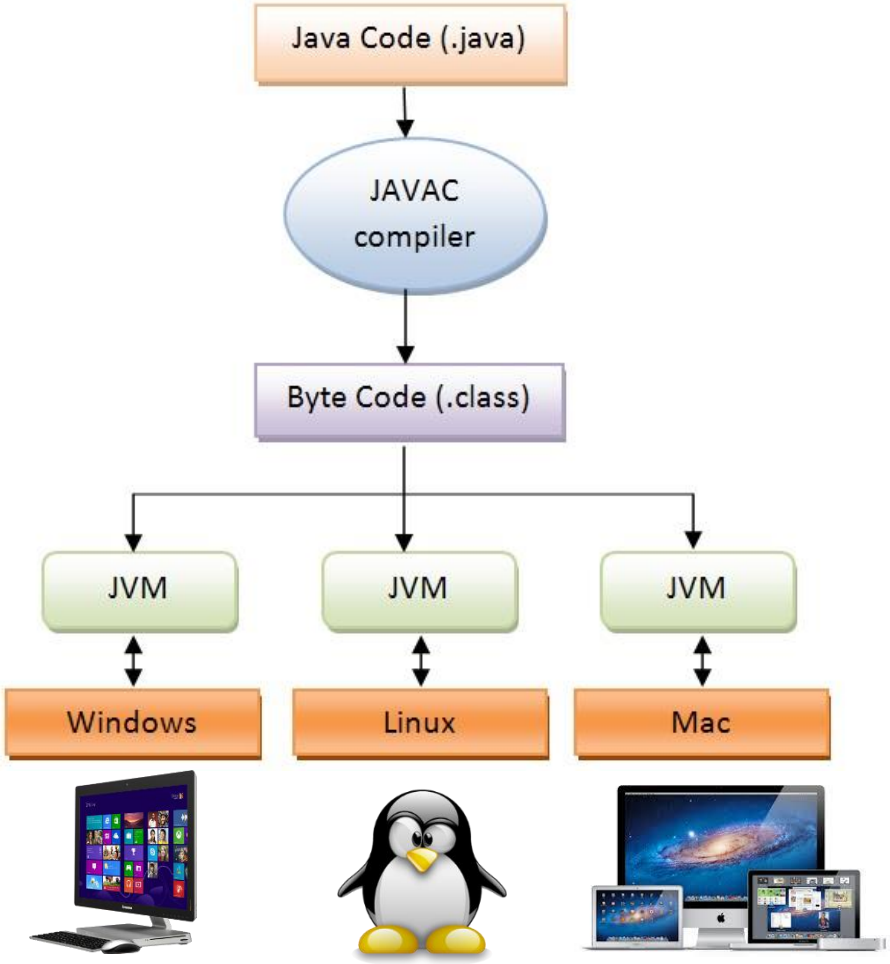
Garbage Collection (Coletor de lixo)

Divisão da Plataforma



- Java EE
- Java FX
- Java SE
- Java ME
- Web Services

Fases do Programa Java



Aplicações mais comuns em Java

Applets

Servlets

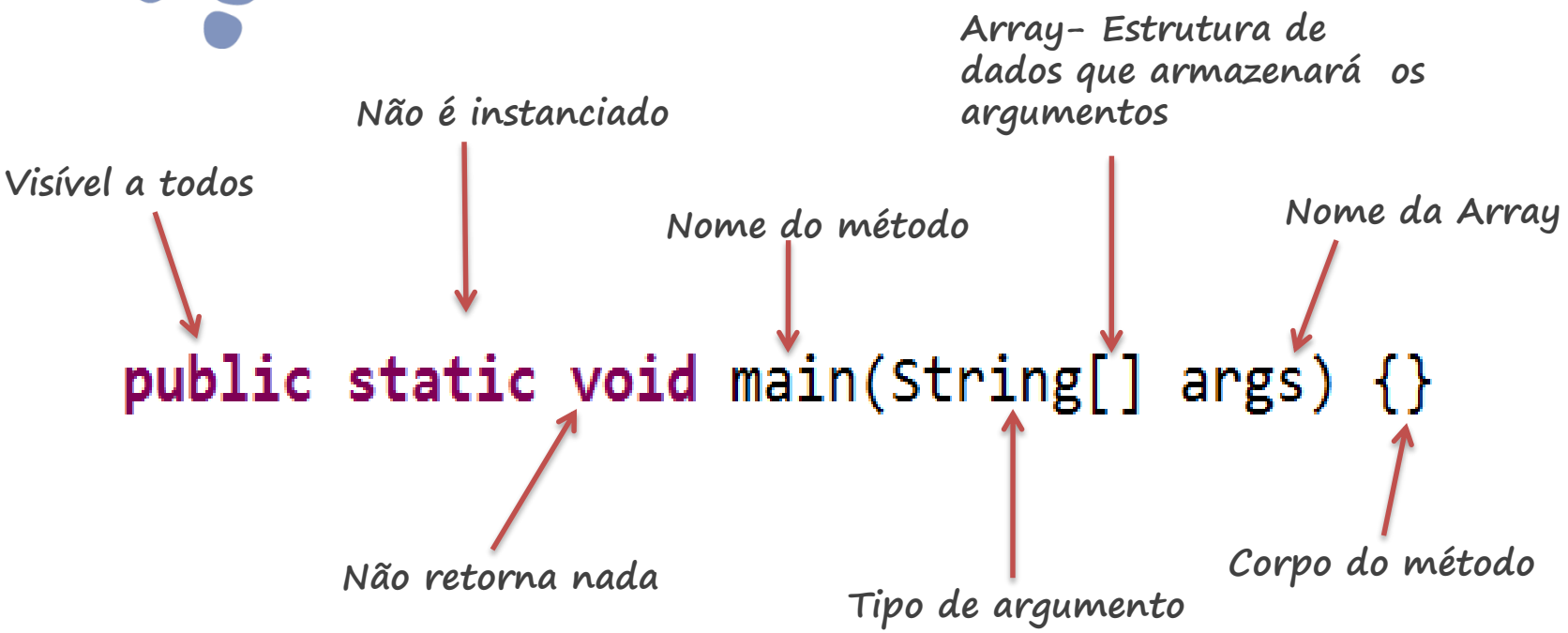
JSP

JSF

EJB

JPA

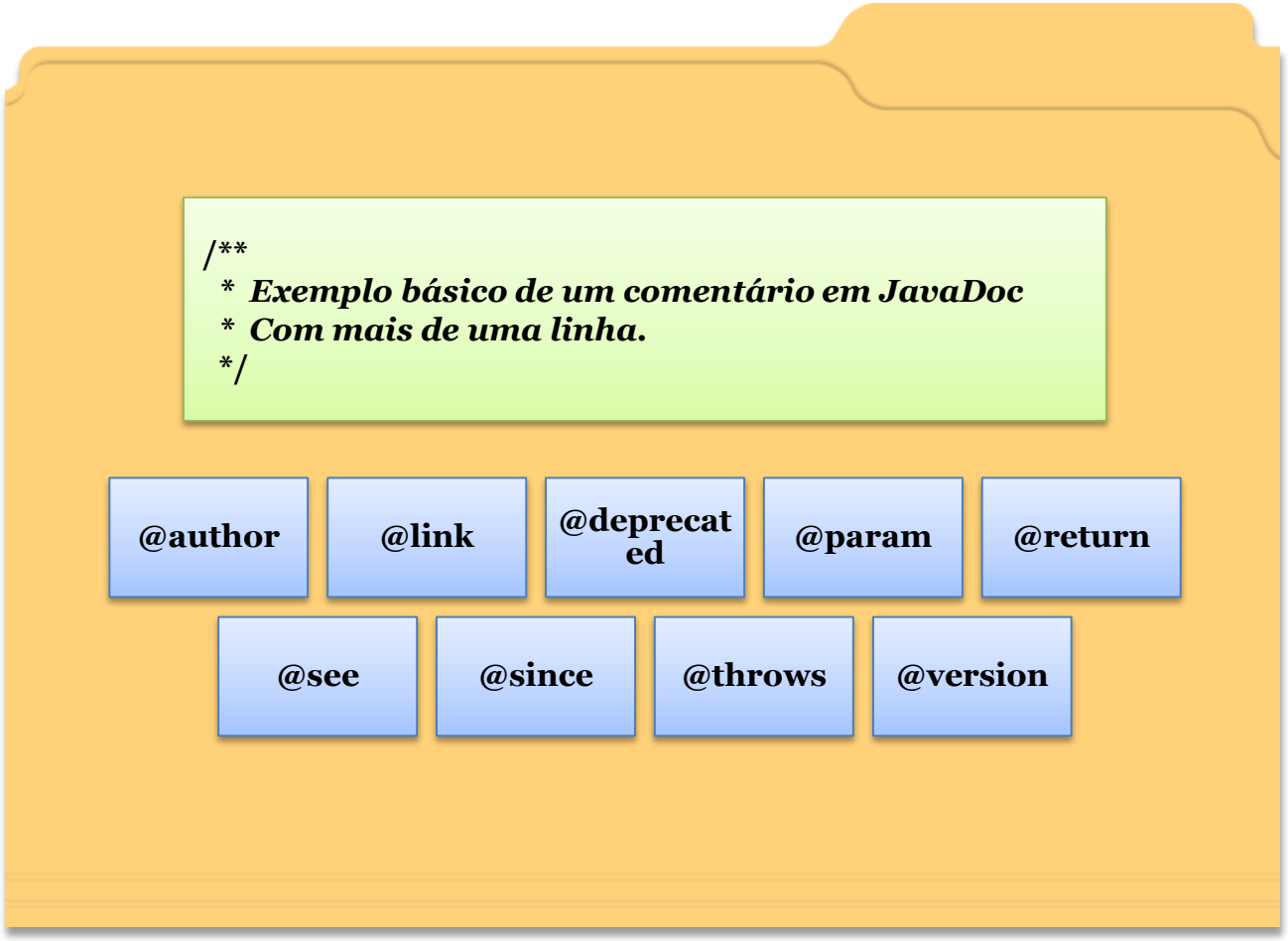
O método main()



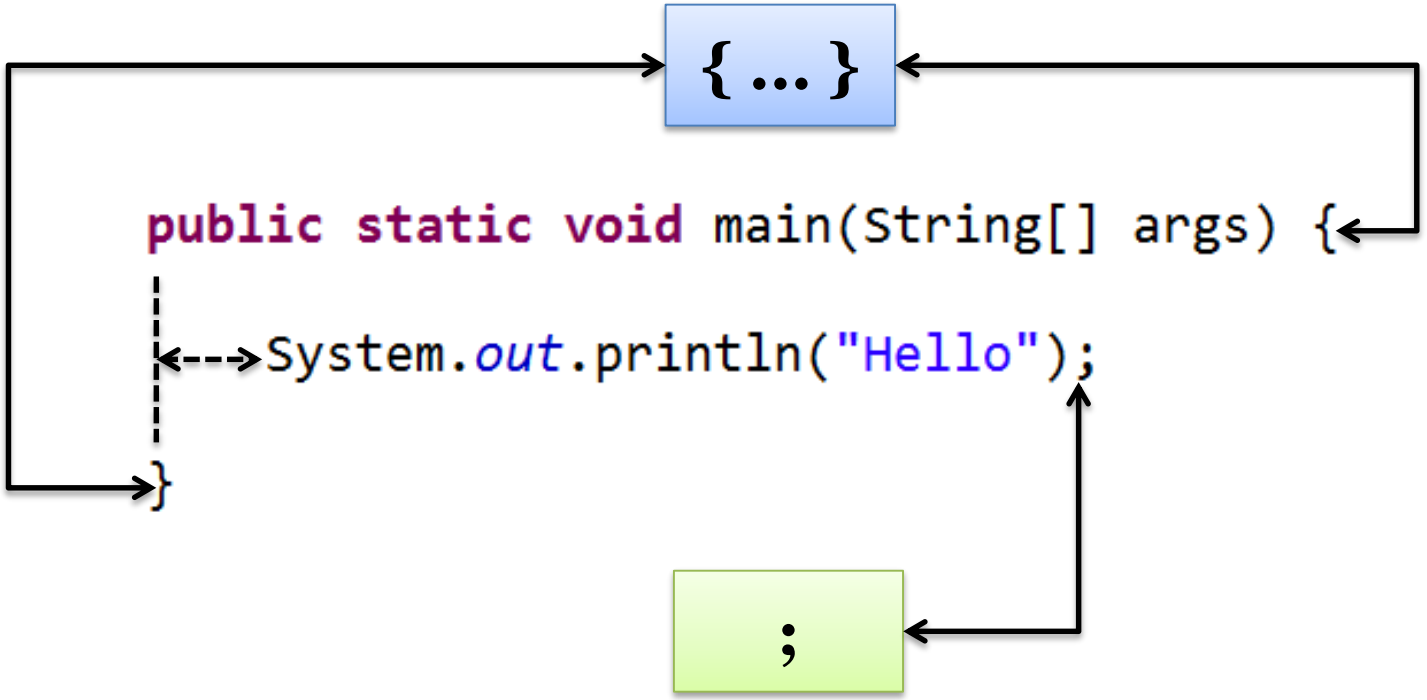
Baby Steps

Java Orientado a Objetos
Identificadores, palavras-chave e tipo

JavaDoc


/**
* Exemplo básico de um comentário em JavaDoc
* Com mais de uma linha.
*/
@author
@link
@deprecated
@param
@return
@see
@since
@throws
@version

Ponto-e-Vírgula, Blocos e Espaço



Identificadores e Palavras Reservadas

case-sensitive

hello ≠ Hello

numero
número
a_com_b
var01
$$_100

n^mero
0001var
a-com-b
var/01, $$..100

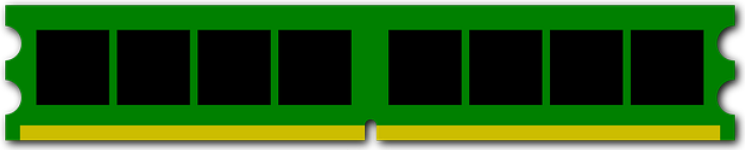
new long
int private
public void
class
void char

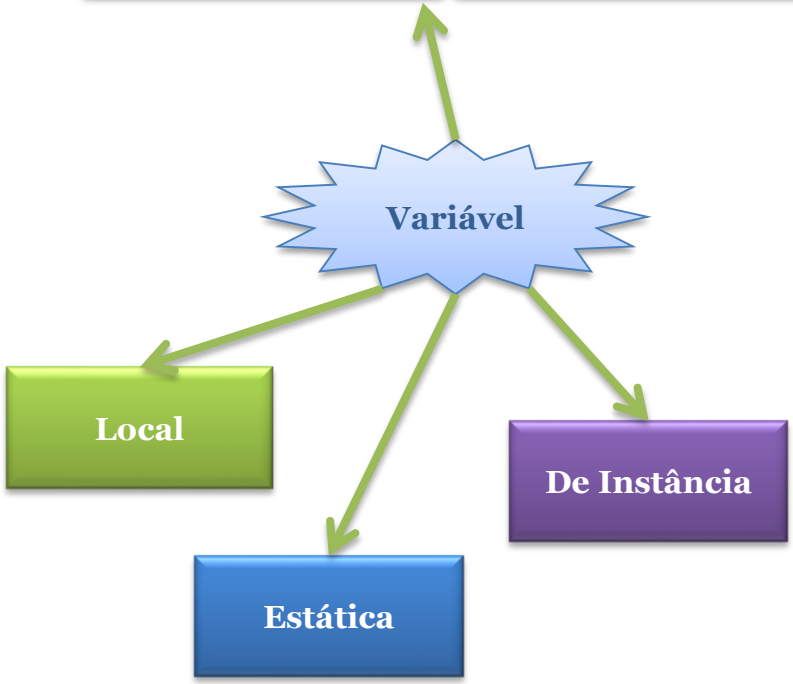
Variáveis, Declaração e Atribuição

Atribuindo Variável

<tipo do dado> <nome> = [valor inicial];

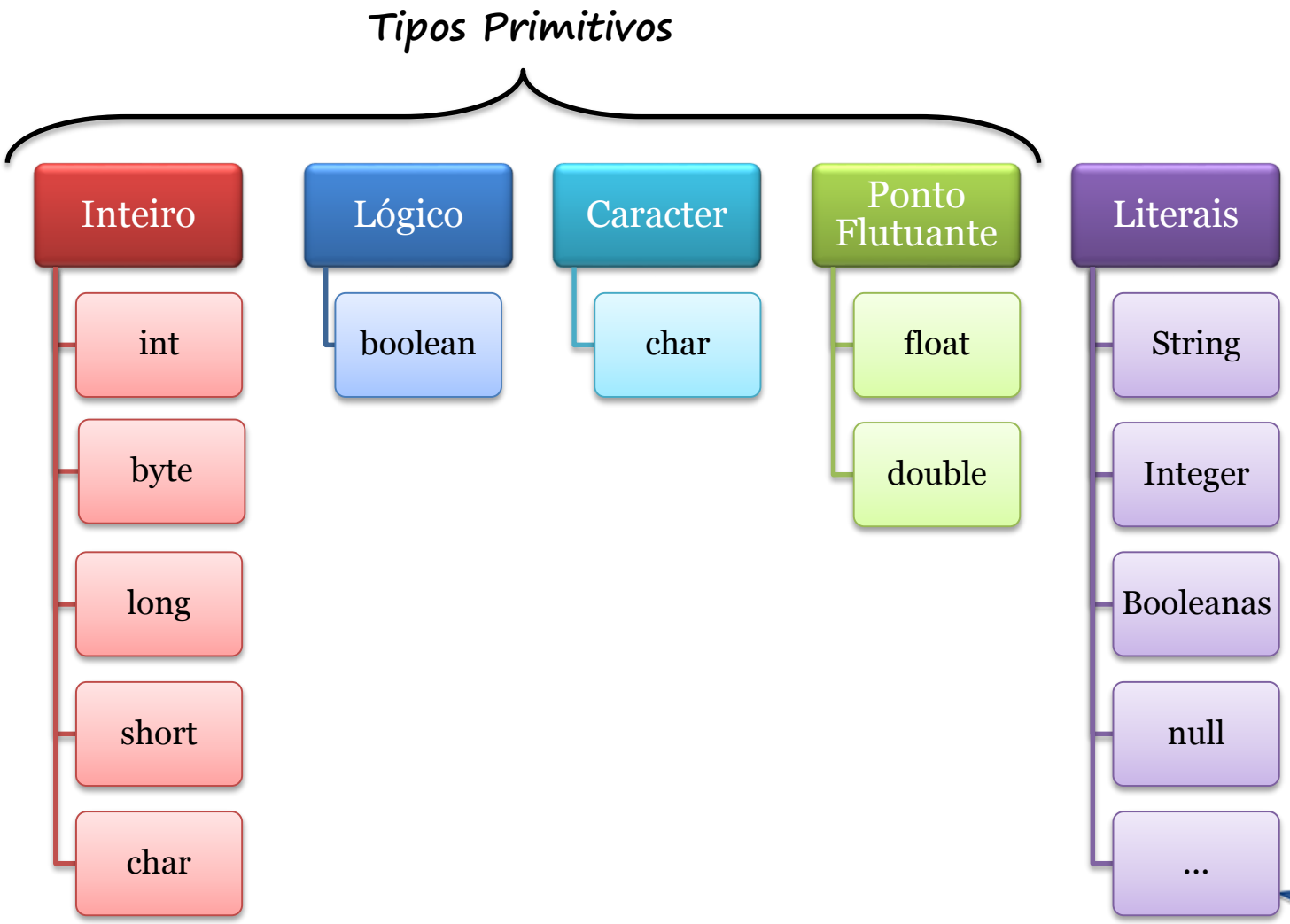
Declarando Variável



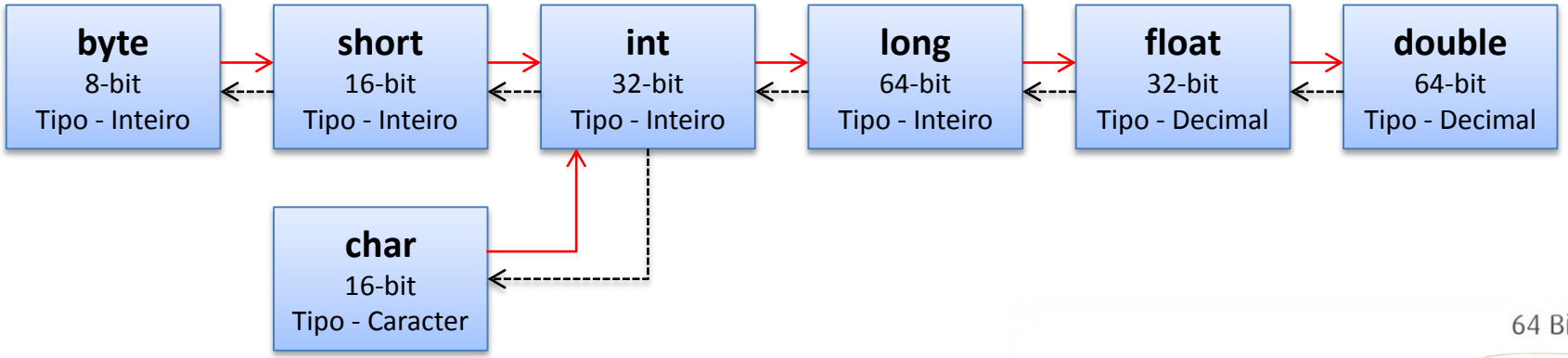


Variáveis devem ser declaradas e inicializadas antes de serem utilizadas.

Tipos de Dados



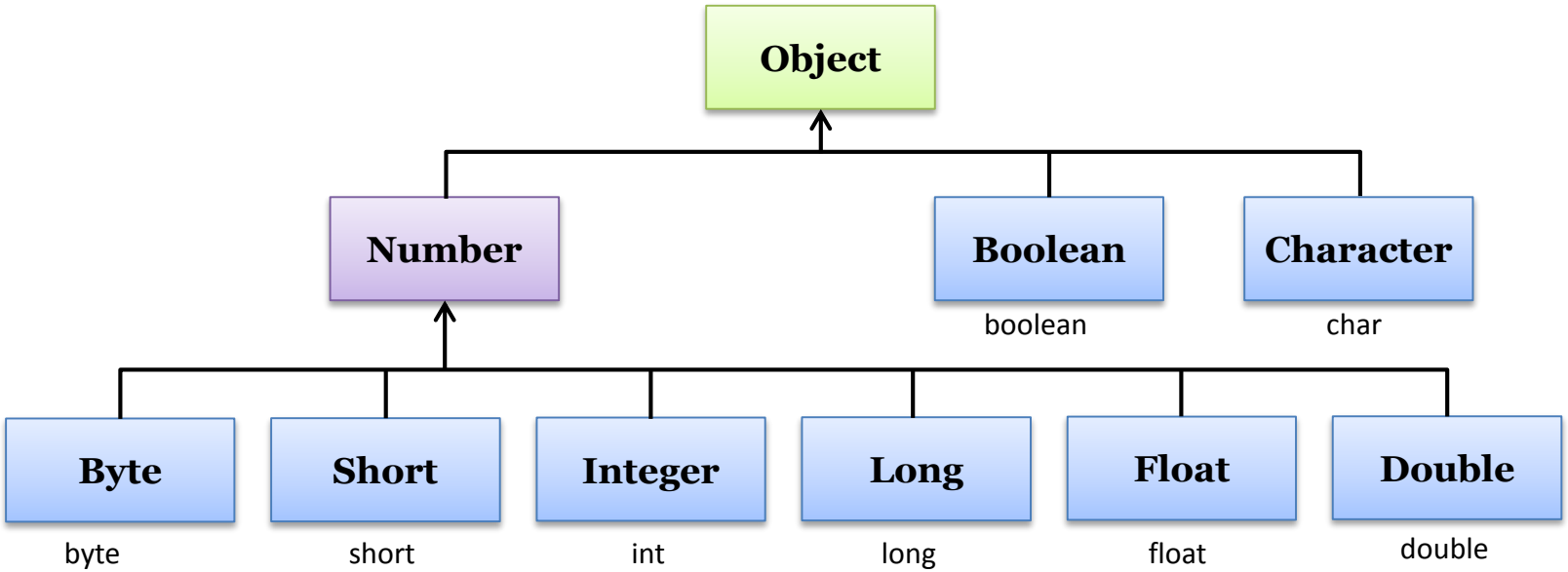
Casting de Tipos Primitivos



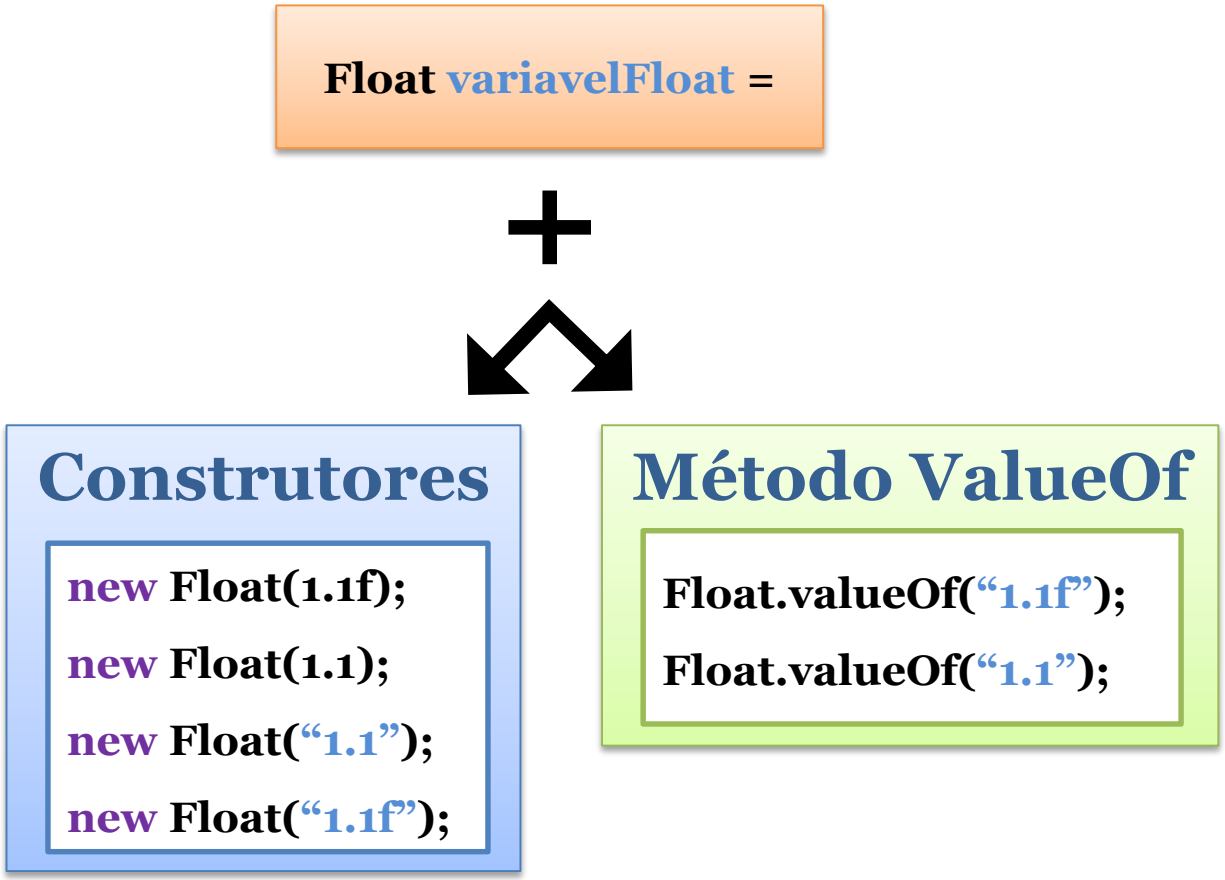
→ **Casting implícito (Automático)**

----> **Casting explícito (Requer a utilizaçao de cast)**

Classes Wrapper (Empacotadoras)



Construtores e método valueOf



AutoBoxing – Boxing and Unboxing

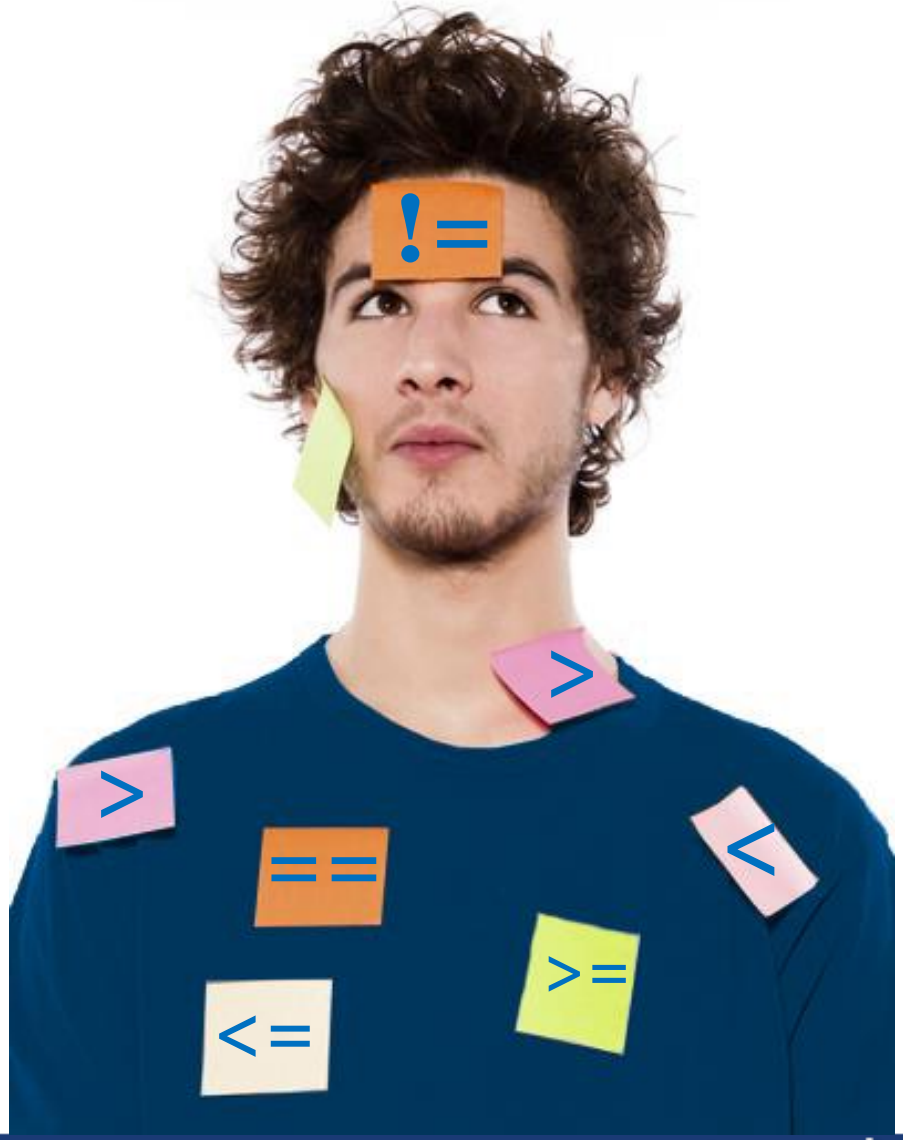
```
int i = 10;
Integer iRef = new Integer(i); // Boxing Explicito
int j = iRef.intValue(); // Unboxing Explicito
iRef = i; // Boxing Automatico
j = iRef; // Unboxing Automatico
```

Java Orientado a Objetos
Operadores

Operadores Aritméticos

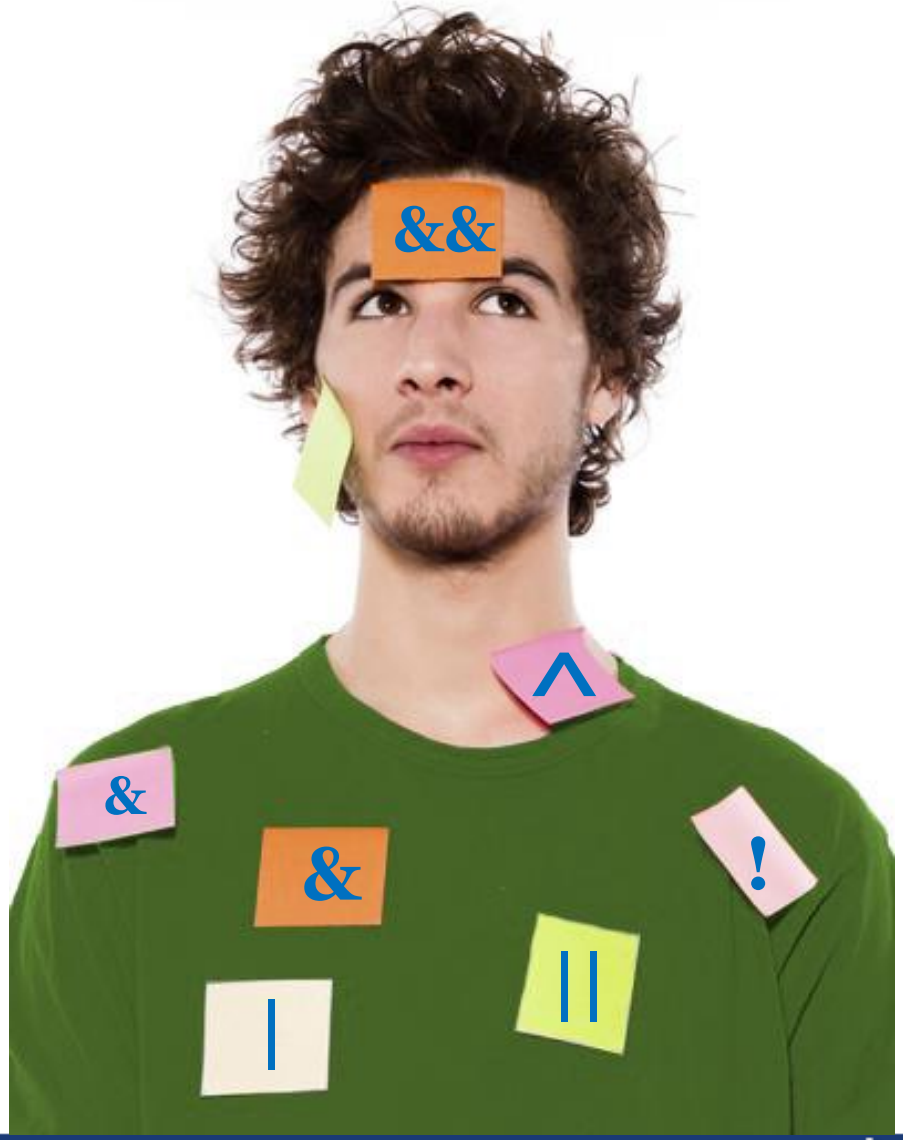
*	var1 * var2	Multiplicação
+	var1 + var2	Adição
-	var1 – var2	Subtração
%	var1 % var2	Resto da divisão
/	var1 / var2	Divisão

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

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)

&& (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.

|| (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.

^ (ou exclusivo binário)

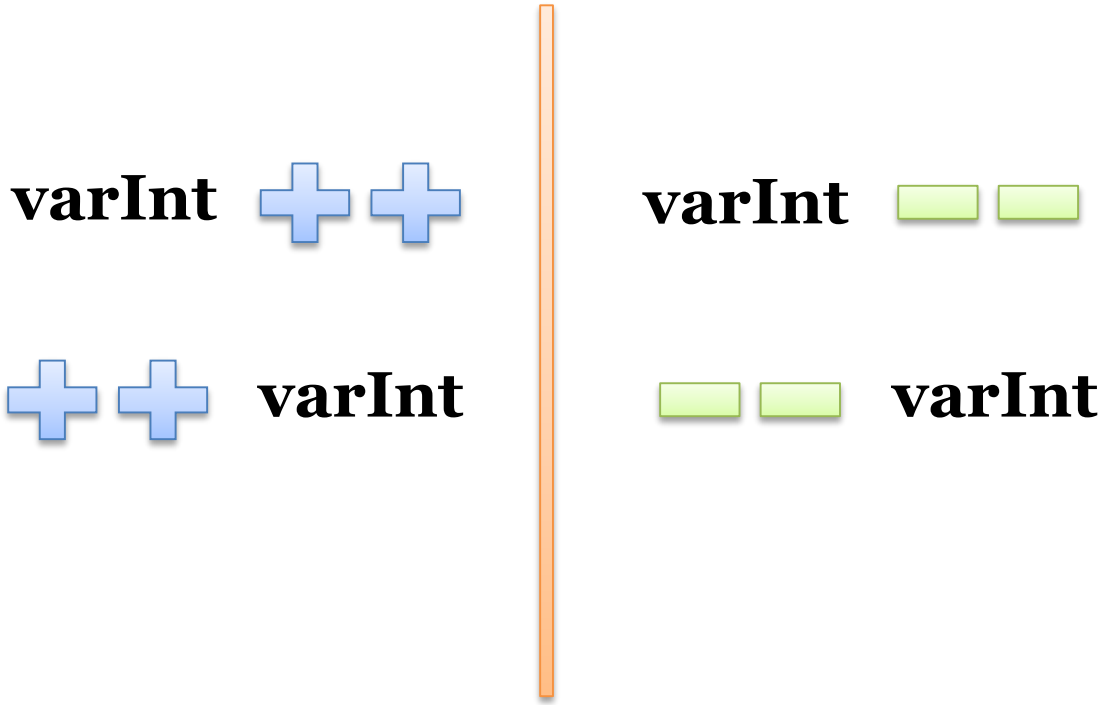
Condição 1	Operador	Condição2	Resultado
true	^	true	false
false	^	true	true
true	^	false	true
false	^	false	false

! (Negação)

Condição	Operador	Resultado
true	!	false
false	!	true

Operadores de Incremento e Decremento

```
int varInt = 1;
```



Precedência de Operadores

1º Calcula o parêntese em maior nível

3º Calcula as operações de adição e subtração

((10%4)*5)+(4/2)+88-10

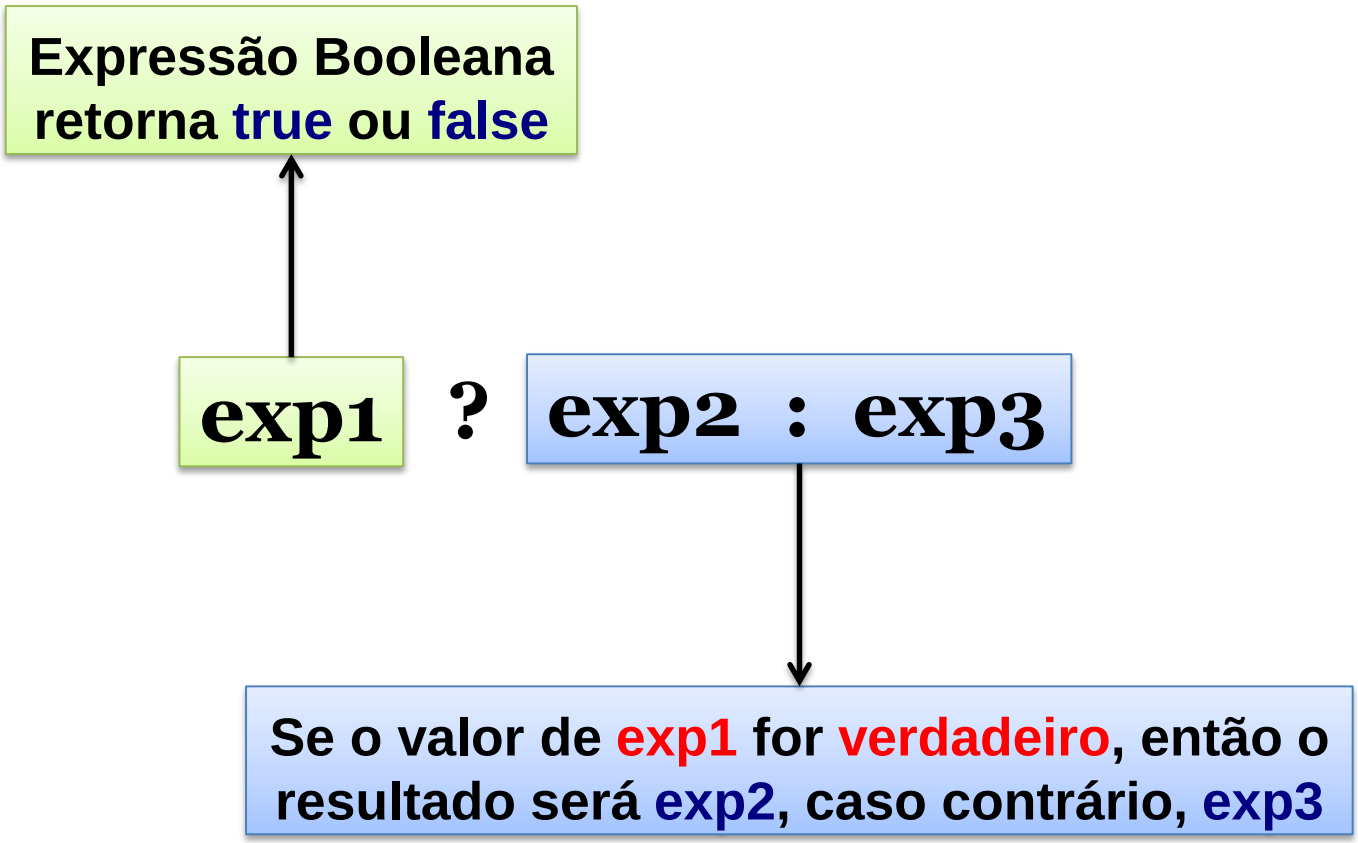
(2*5)+(4/2)+88-10

10+2+88-10

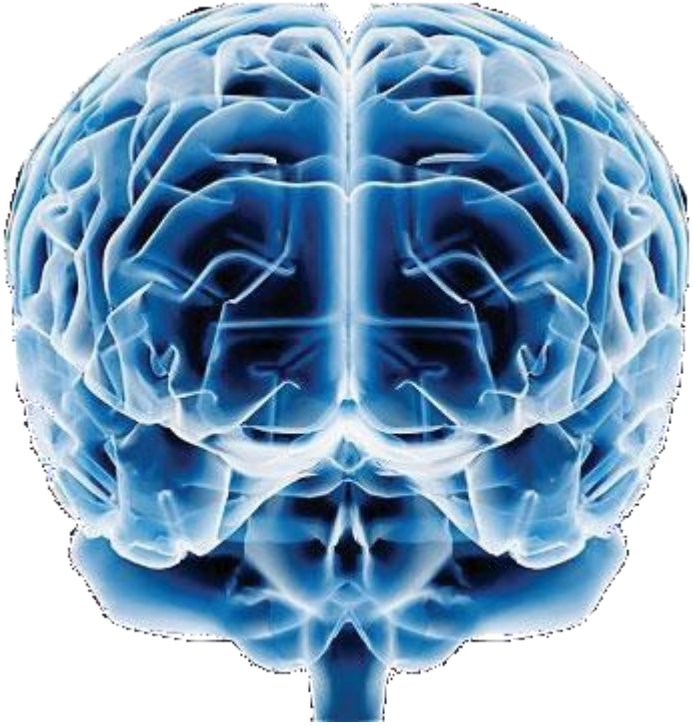
90

2º Calcula os valores dos parênteses restantes

Operador Condicional (?:)



Java Orientado a Objetos
Estruturas de Controle

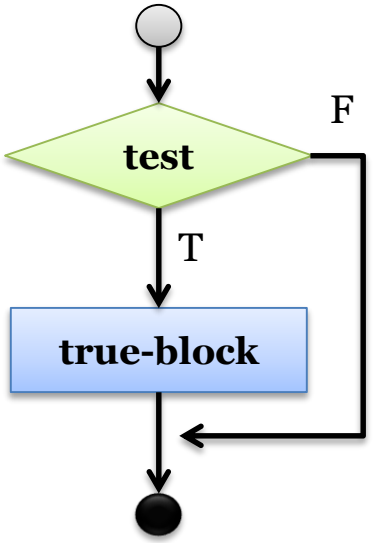


Estrutura de decisão (if)

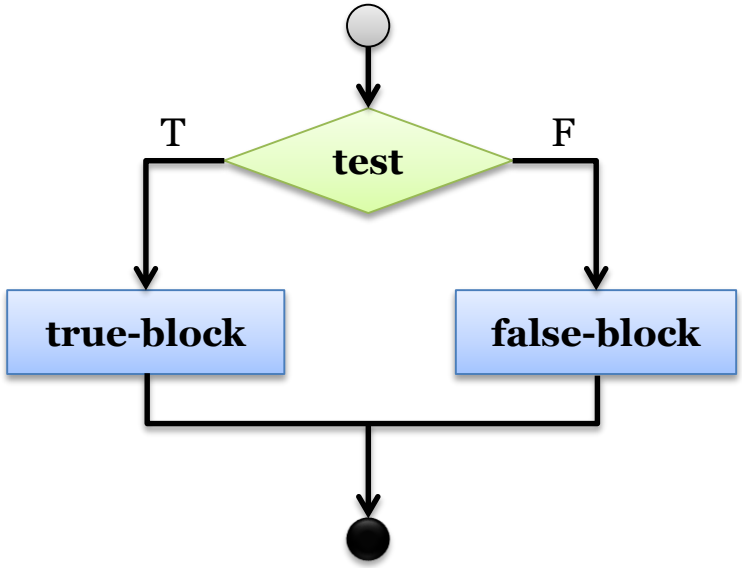
```
int nota = 68;
if (nota > 60){
    System.out.println("Parabéns!");
    System.out.println("Você Passou!");
}
```

Expressão Booleana
test

Bloco de código da expressão true
true-block



Estrutura de decisão (if-else)



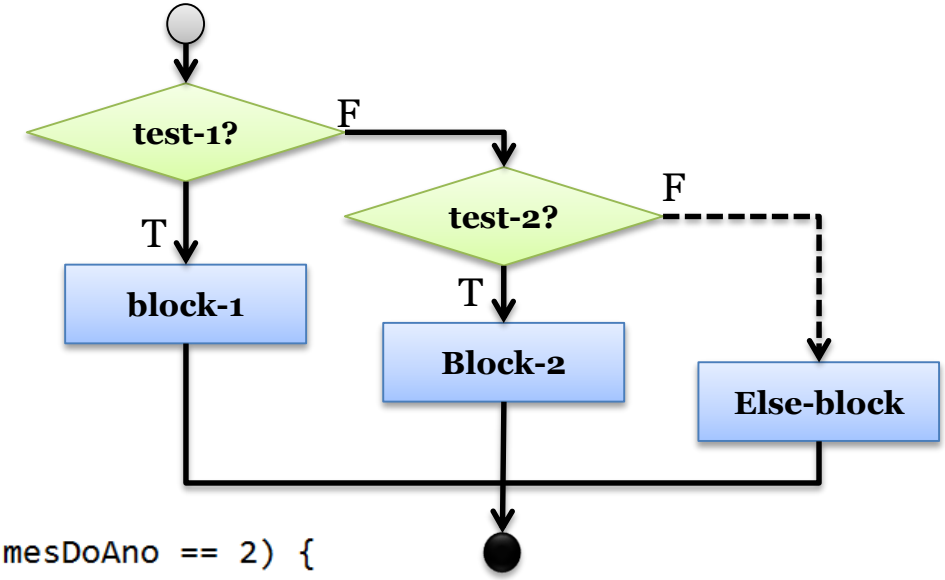
Expressão Booleana
test

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

Bloco de código da expressão true
true-block

Bloco de código da expressão false
false-block

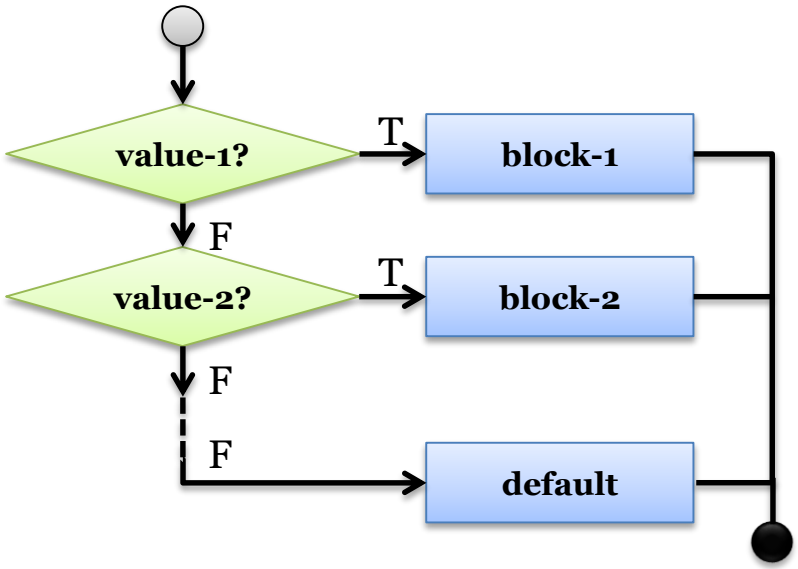
Estrutura de decisão (if-else-if)



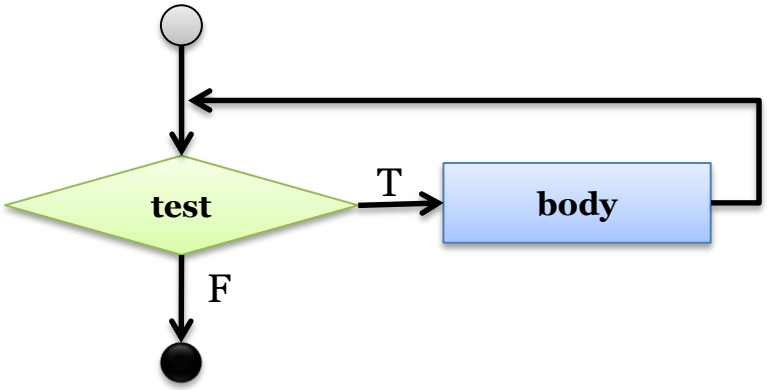

```
public static void main(String[] args) {

    int mesDoAno = 13;

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

}
```

Estrutura de decisão (switch)

```
int mesDoAno = 13;

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



Estrutura de repetição (while)

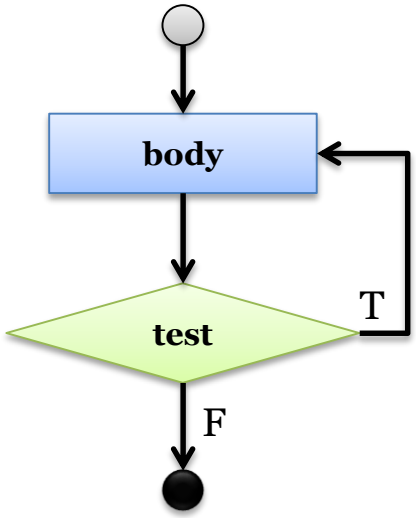



```
public static void main(String[] args) {

    int contador = 0;

    while (contador < 10) {
        System.out.println(contador);
        contador++;
    }
}
```

Expressão Booleana
test

Corpo da estrutura de repetição
body

Estrutura de repetição (do-while)

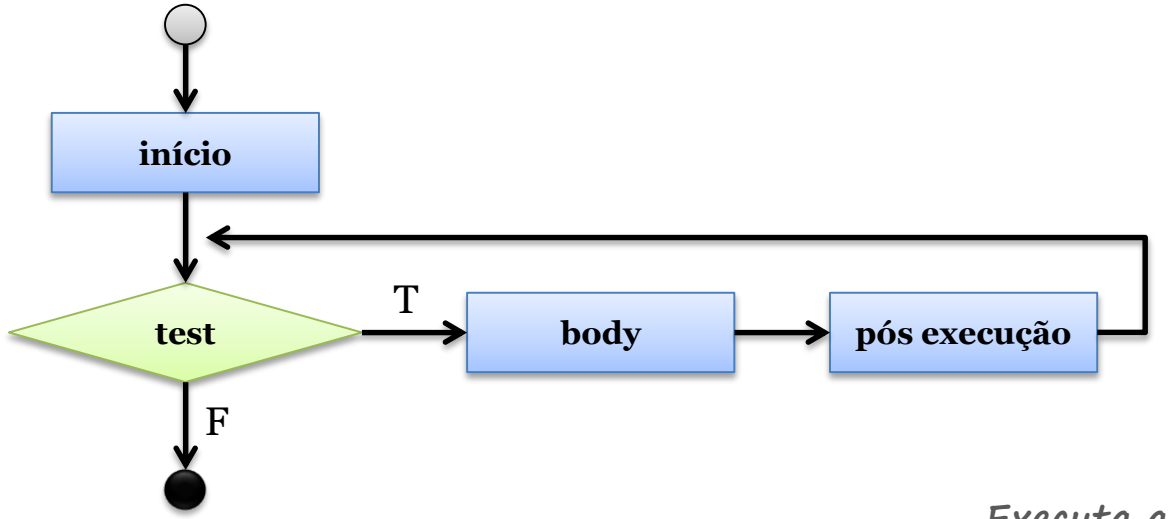



```
public static void main(String[] args) {

    int contador = 0;

    do {
        System.out.println(contador);
        contador++;
    } while (contador < 10);
}
```

Expressão Booleana
test

Corpo da estrutura de repetição
body

Estrutura de repetição (for)




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

    for(int contador = 0; contador < 10; contador++){
        System.out.println(contador);
    }
}
```

Executa ao final do for
pós execução

Expressão Booleana
test

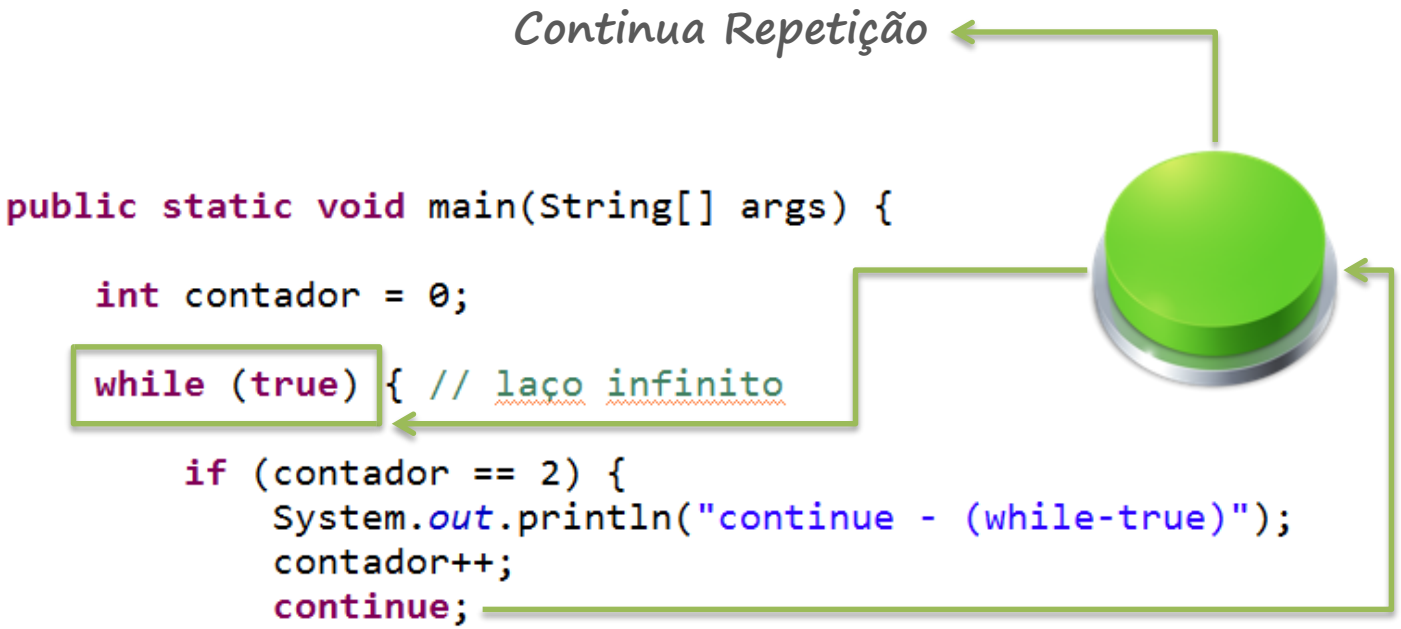
Corpo da repetição for
body

Declaração break

Interrompe a repetição infinita

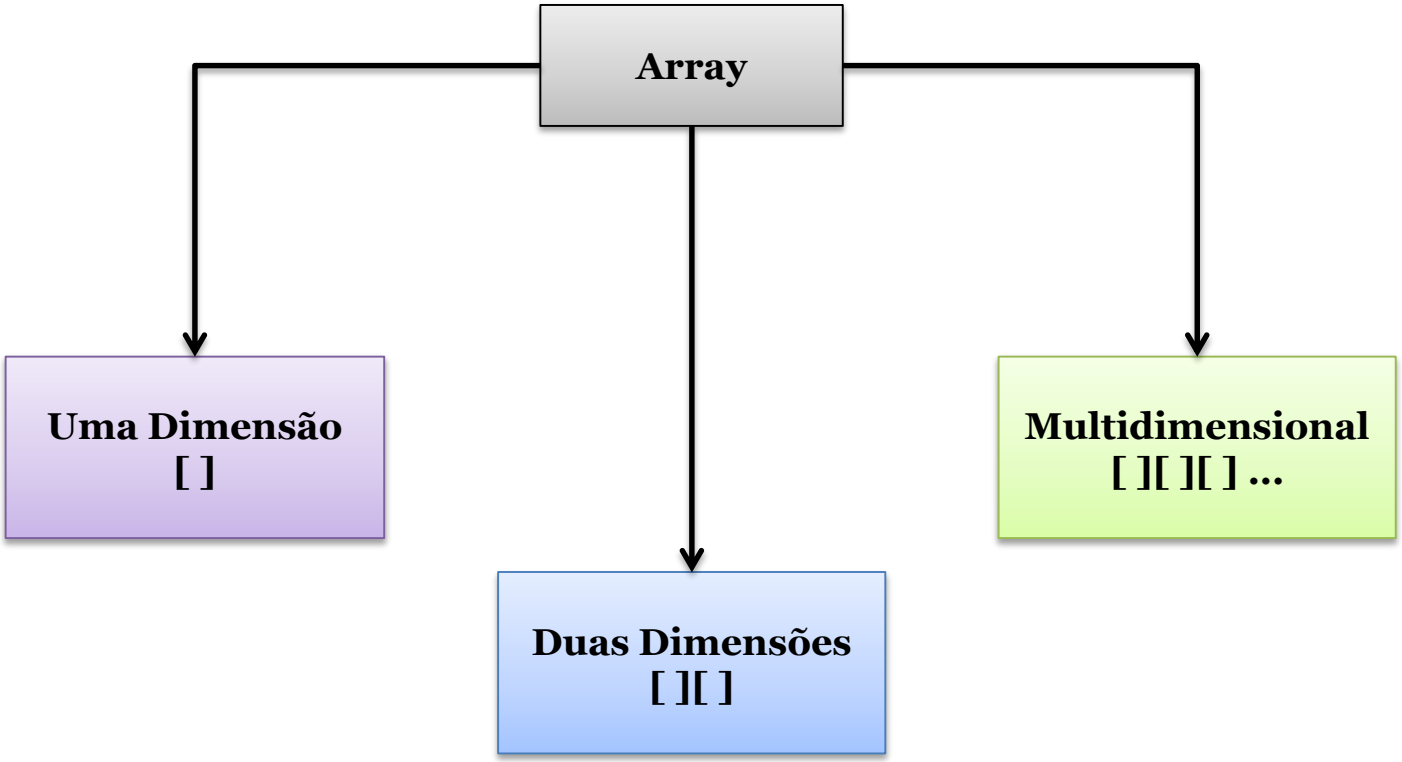
```
public static void main(String[] args) {

    int contador = 0;

    while(true){ //laço infinito
        if(contador == 10){
            System.out.println("break - (while-true)");
            break;
        }
        System.out.println(contador);
        contador ++;
    }
}
```

Declaração continue



Java Orientado a Objetos
Array

Array ?



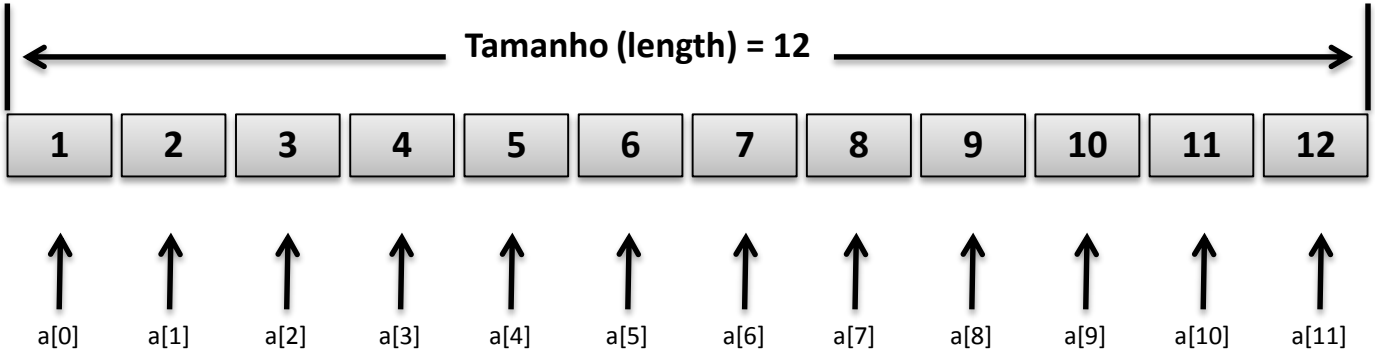
Declarando Array

Inicialização

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

||

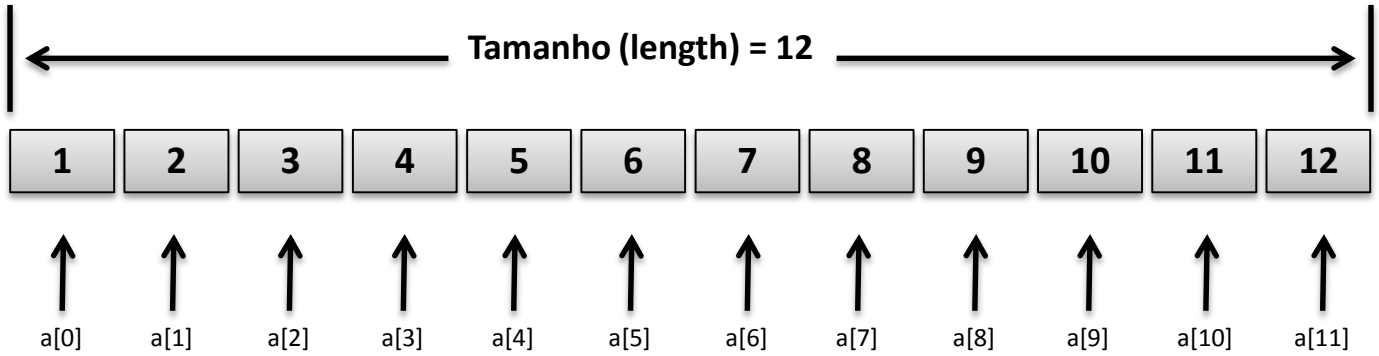
```
int []a = {1,2,3,4,5,6,7,8,9,10,11,12};
```



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

Acessando um elemento do Array

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

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

    System.out.println(a[0]); // acessando primeiro elemento do array

    System.out.println(a[6]); // acessando elemento de indice 6

    System.out.println(a[a.length - 1]); // acessando ultimo elemento do array

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



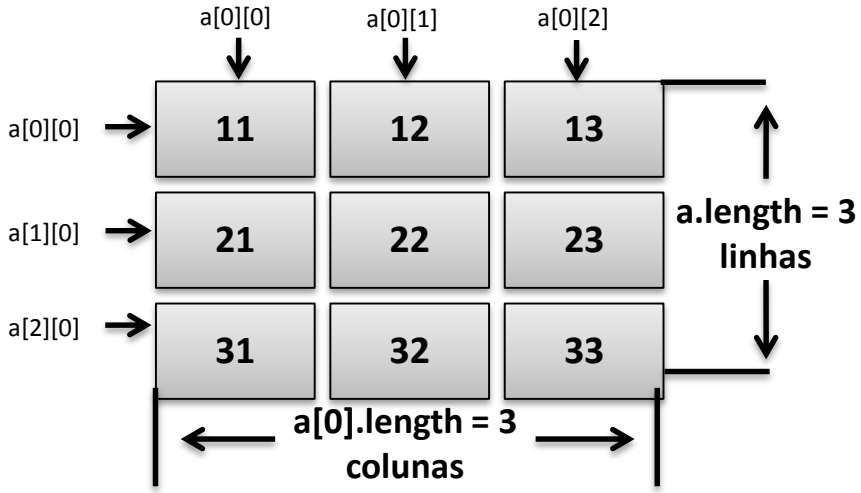
Arrays Multidimensionais

Inicialização

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

||

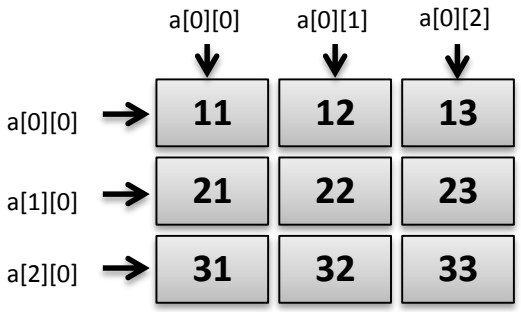
```
int [][]a = {{11,12,13},{21,22,23},{31,32,33}};
```



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

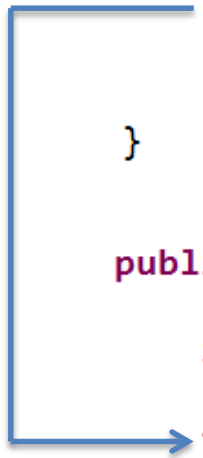
Acessando um elemento de um Array Multidimensional

```
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] + " "); // imprime matriz
        }
        System.out.println("\n");
    }
}
```



Percorrendo Arrays com Enhaced-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) { // ultilizando Enhanced-for
        System.out.println(i); // imprime todos elementos do array
    }
}
```



Manipulando Arrays com java.util.Arrays

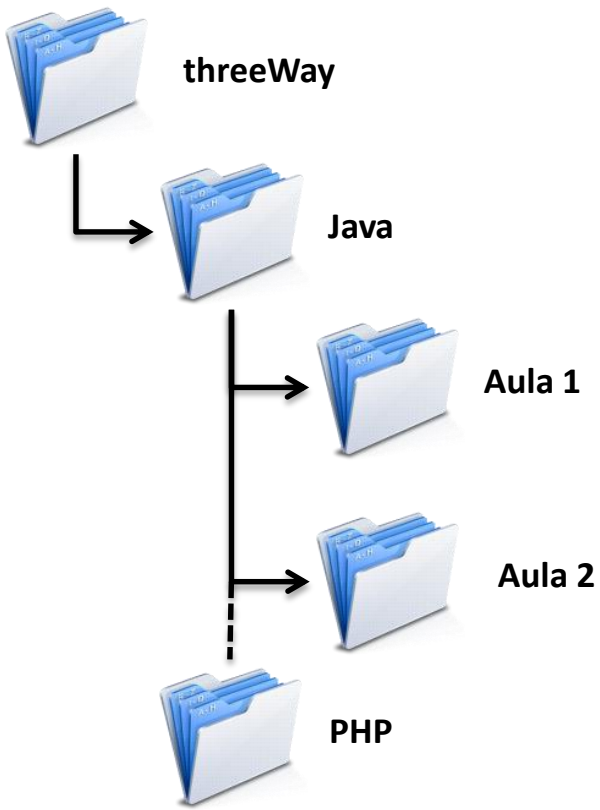

```
public static void main(String[] args) {
    int[] a = { 8, 4, 1, 9, 2, 5, 12, 3, 6, 10, 7, 11 };
    System.out.println(Arrays.toString(a));
    Arrays.sort(a);
    System.out.println(Arrays.toString(a));
    System.out.println("a[" + Arrays.binarySearch(a, 6) + "]");
}
```

Ordenação

Busca Binaria

Java Orientado a Objetos
Bases da programação Java OO

Pacotes



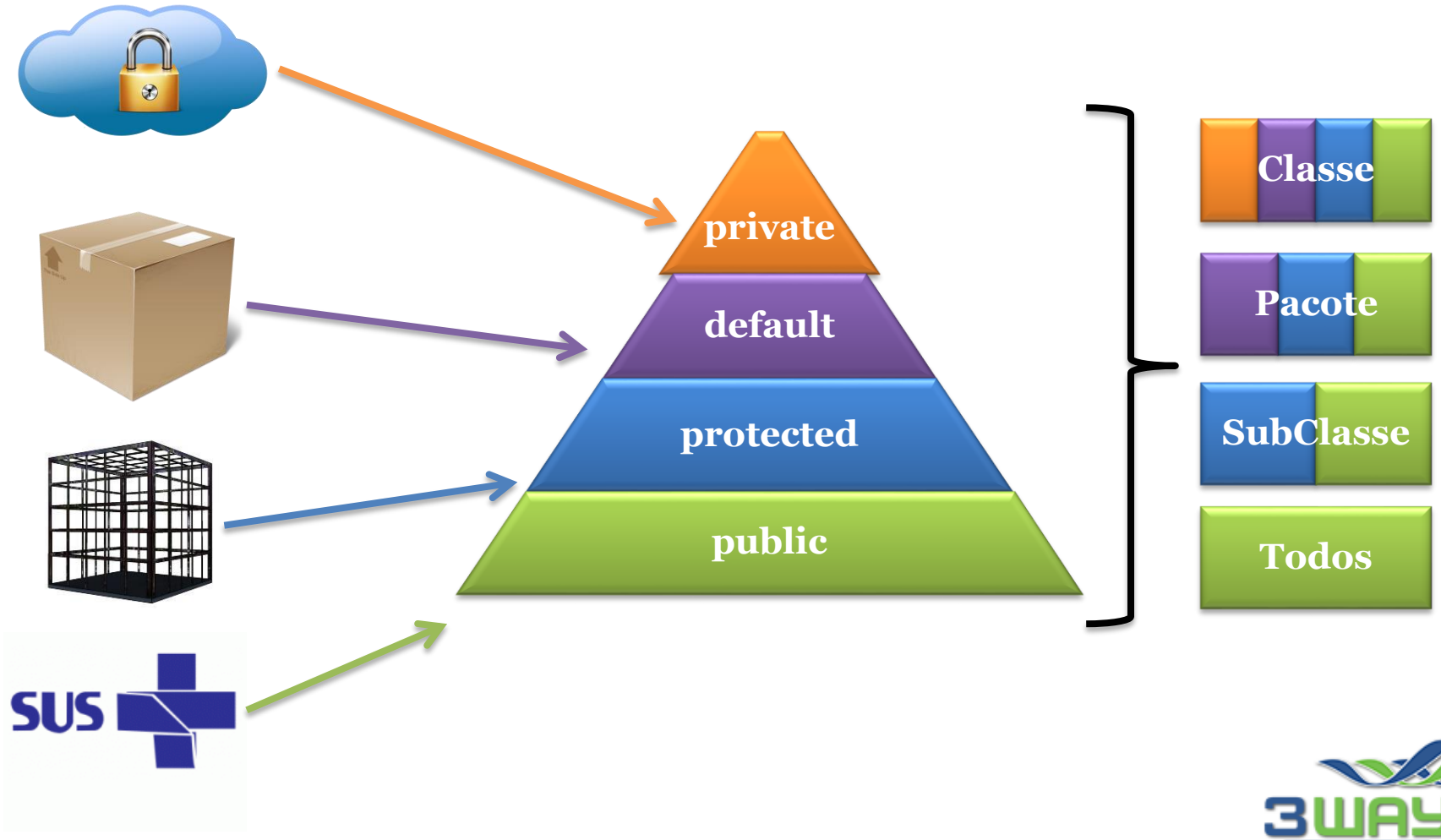
package <nomePacote>;

import <nomePacote.elementoAcessado>;

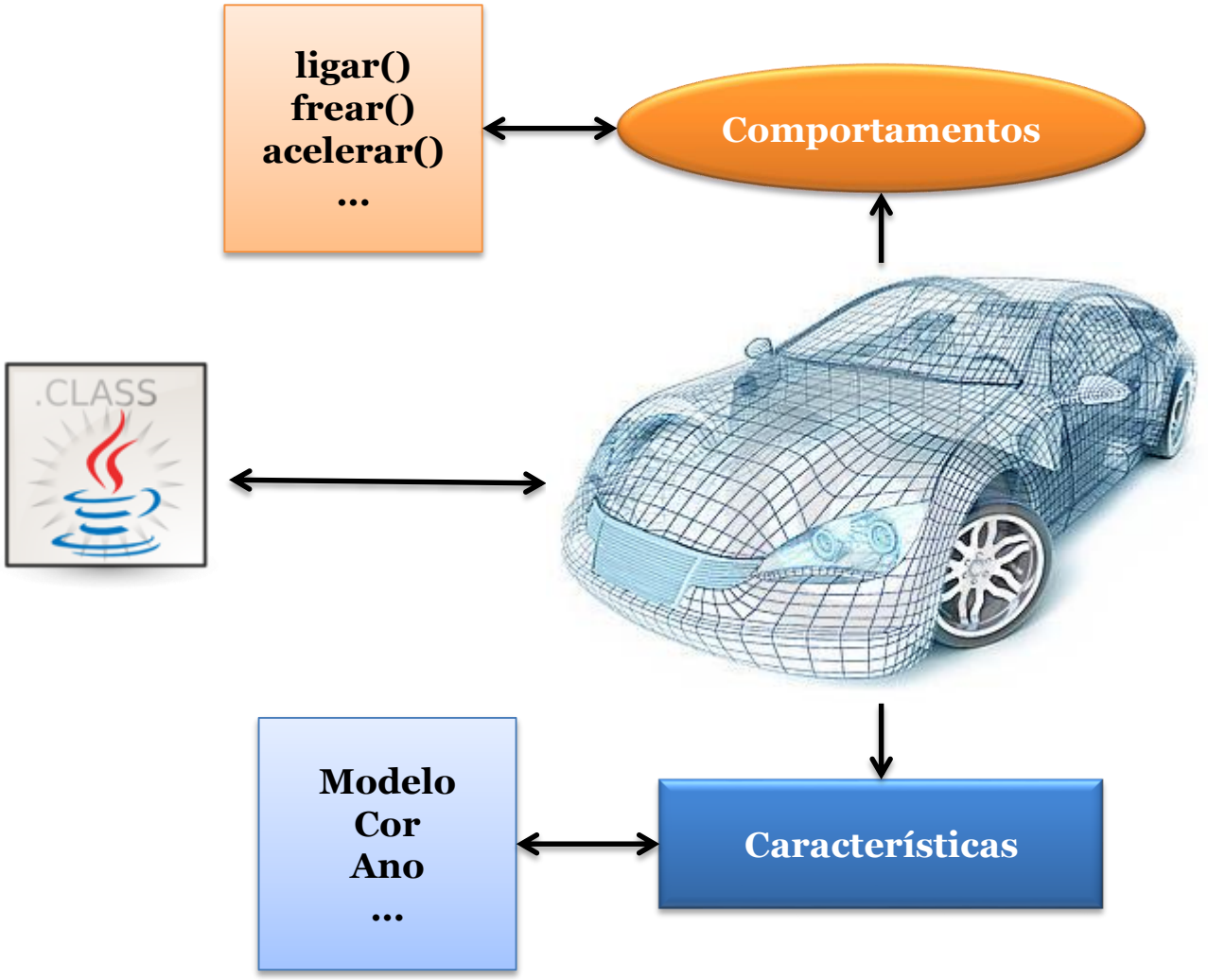
<declaraçãoClasse>

```
package threeWay.Java.Aula;

import java.util.Arrays;

public class Teste { }
```

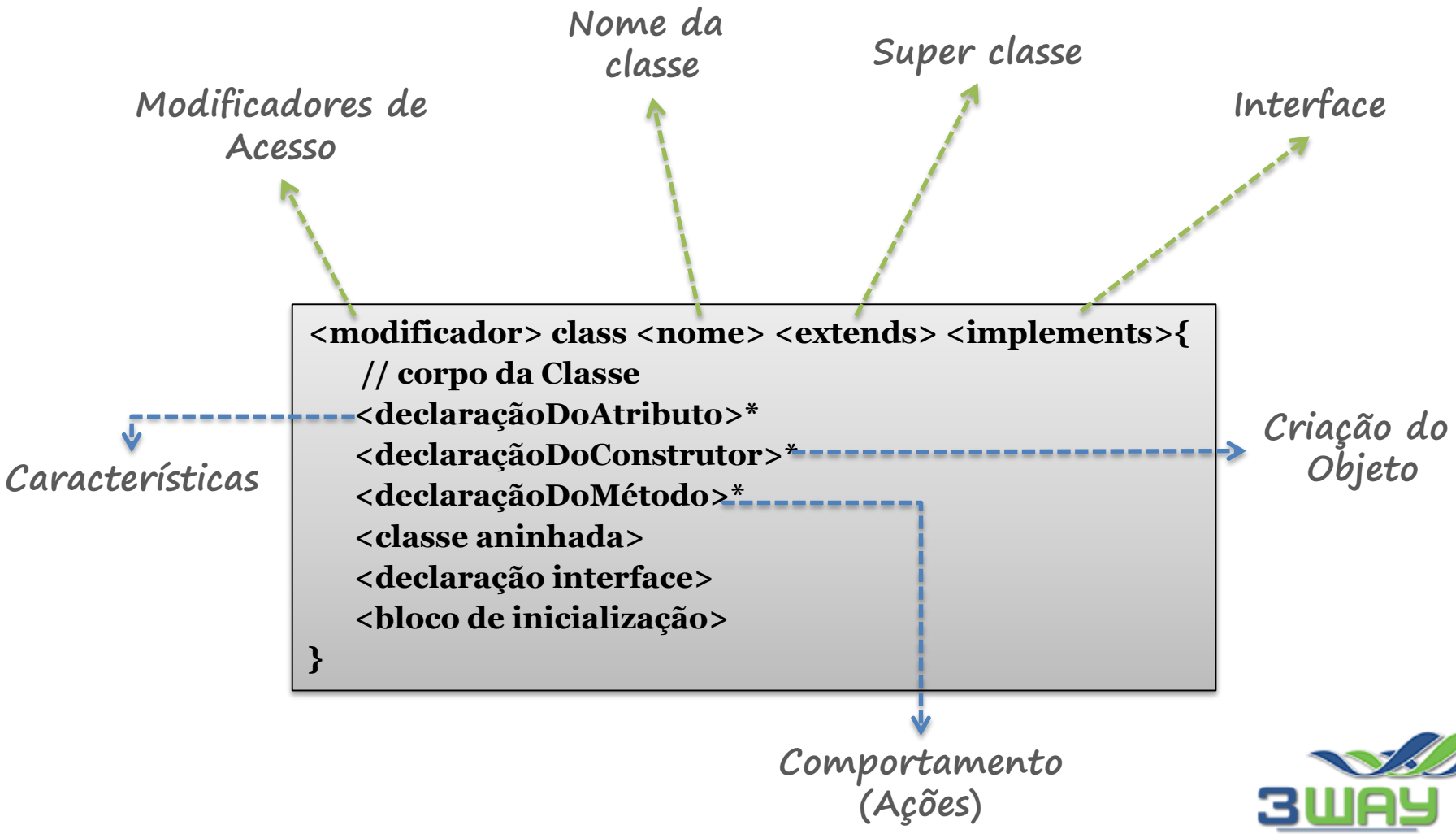
Modificadores de acesso



Classes


ligar()
frear()
acelerar()
...
Comportamentos
.CLASS
Modelo
Cor
Ano
...
Características

Definindo Classes



Métodos

Decomposição e a Solução
para problemas

Separa o problema
em partes menores e
reaproveitáveis

Métodos resolve
uma parte especifica
desses problemas

Na Orientação a Objetos
os métodos referenciam comportamentos das classes.

Definindo Métodos

Modificadores de Acesso

Retorno do método

Nome da classe

Variáveis de entrada

Exceções

```
<modificadores> <tipo retorno> <nome>(<parâmetros>)<clausula throws> {
   // Corpo do método
   <variáveis locais>
   <classes locais aninhadas>
   <sentenças>
}
```

Algoritmo

Classe Interna

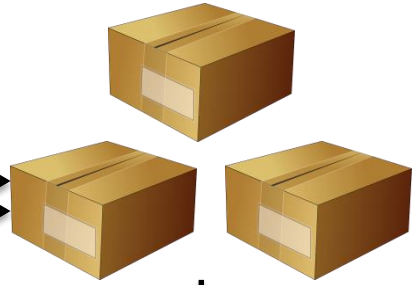
Variáveis do método

Objetos

Programação Estruturada

Paradigma

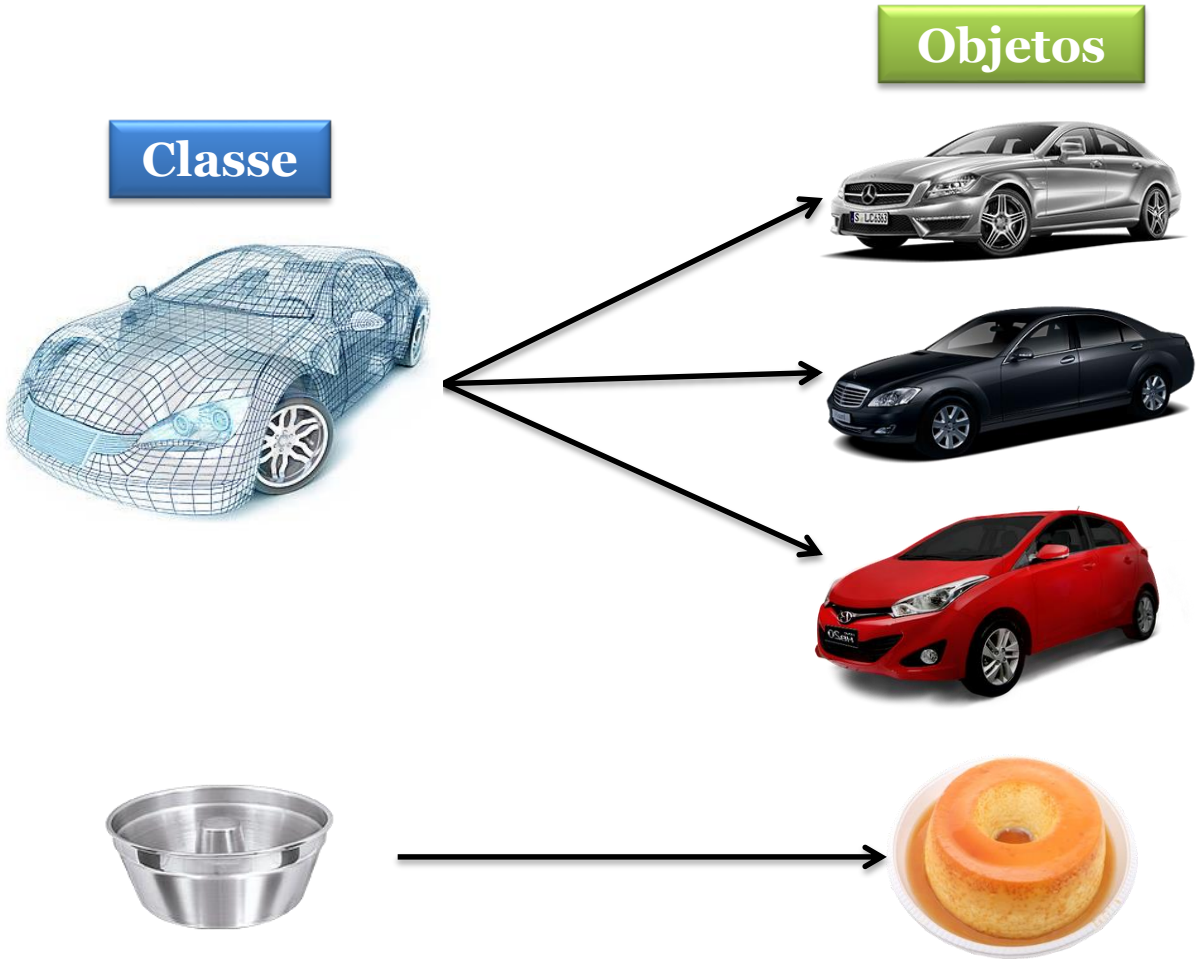
Orientação a Objetos



Tudo que pode ser abstraído!

3WAY NETWORKS

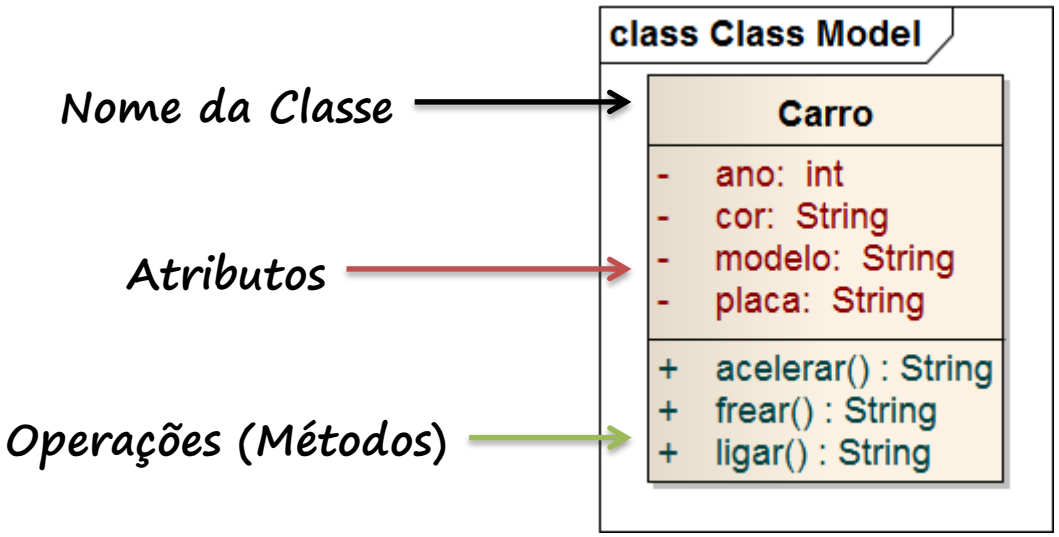
Classe x Objeto



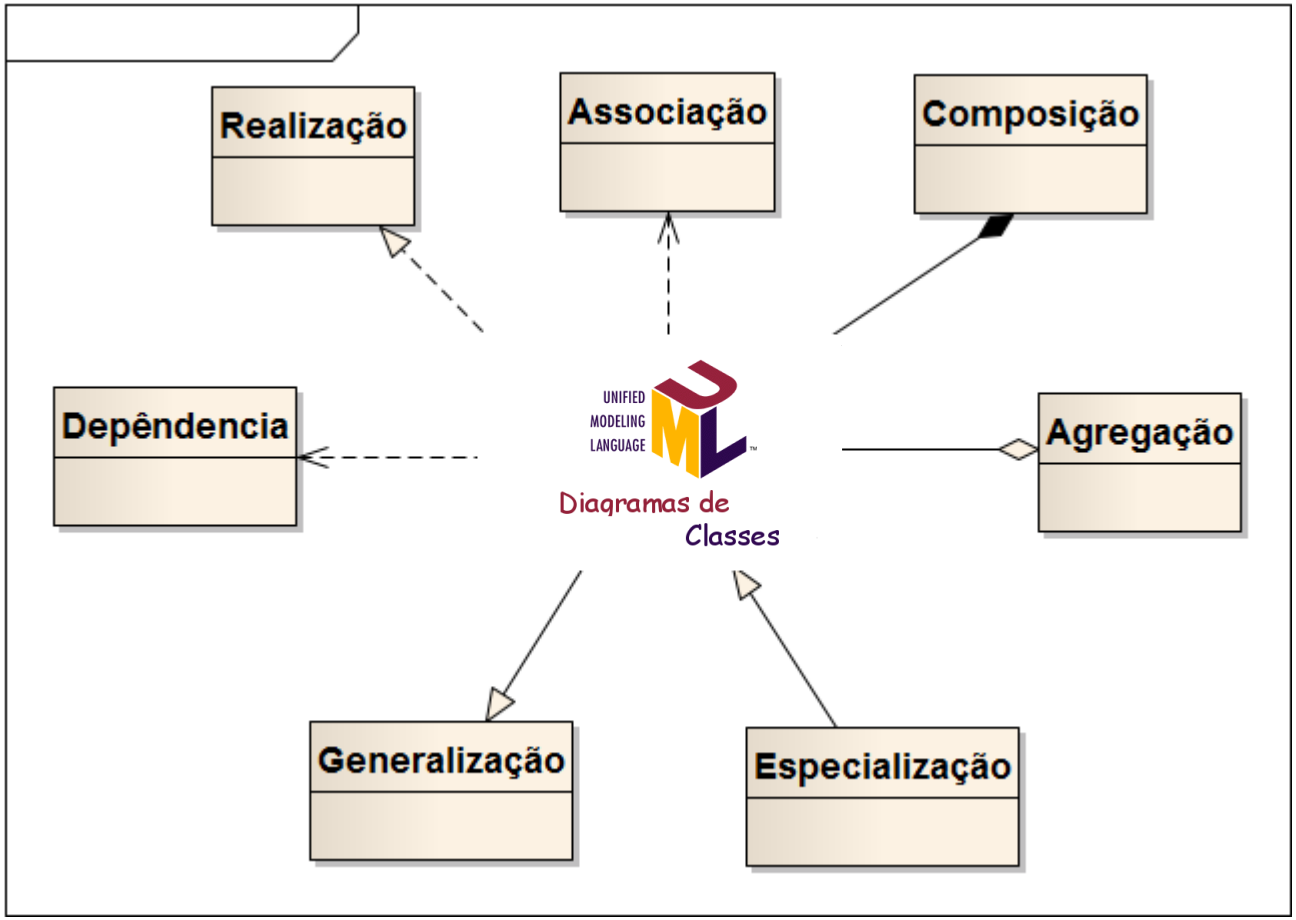
Notação UML

- Diagrama de Objetos
- Diagrama de Objetos
- Diagrama de Caso de Uso
- Diagrama de Classe
- Diagrama de Atividades
- Diagrama de Componentes
- Diagrama de Interação
- Diagrama de Implantação
- Diagrama de Pacotes
- Diagrama de Estrutura

Notação UML
Diagrama de Classe

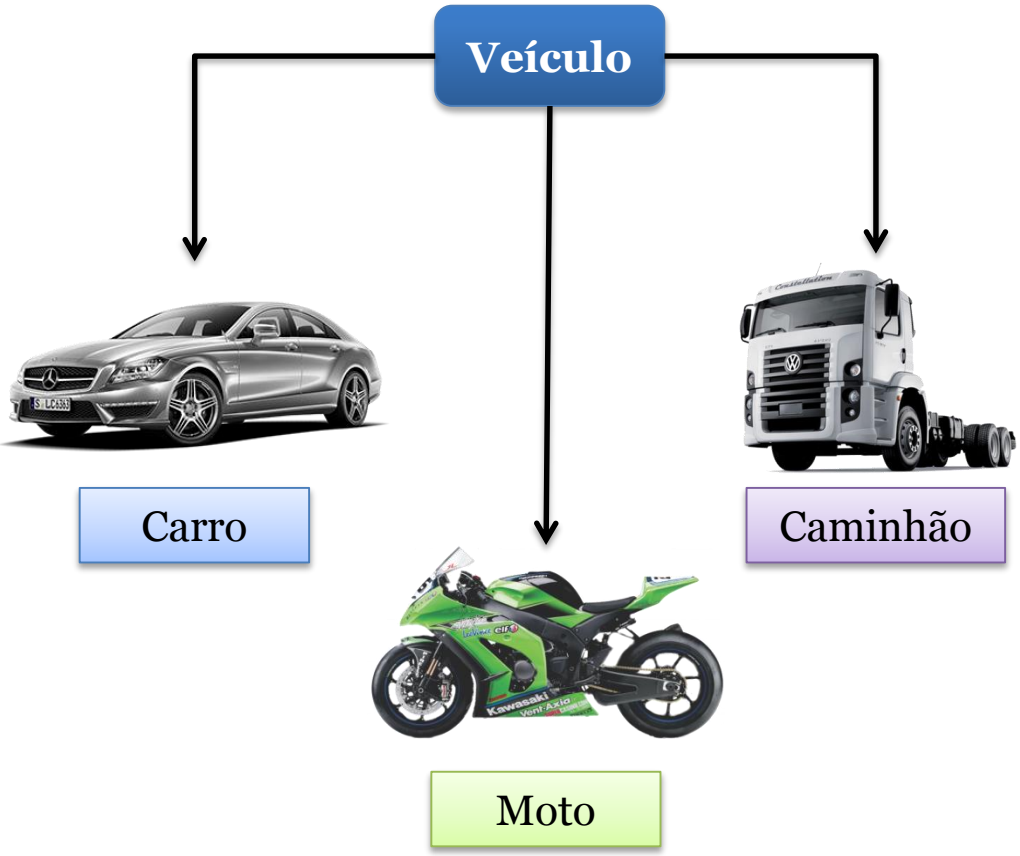


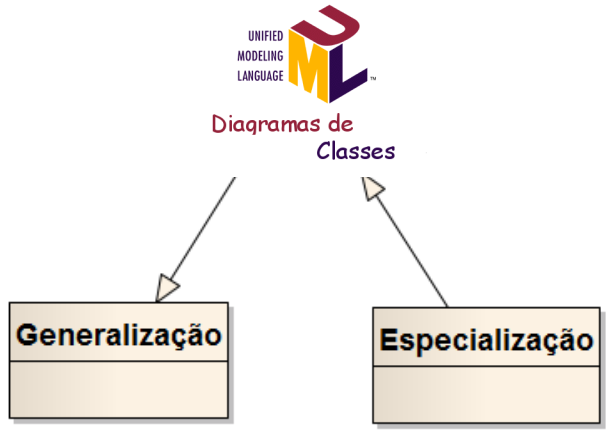
Notação UML
Relacionamentos



Herança

Relacionamentos do tipo é um

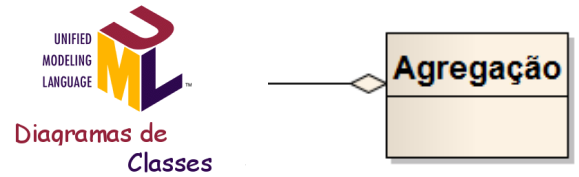




Notação UML para especialização e generalização

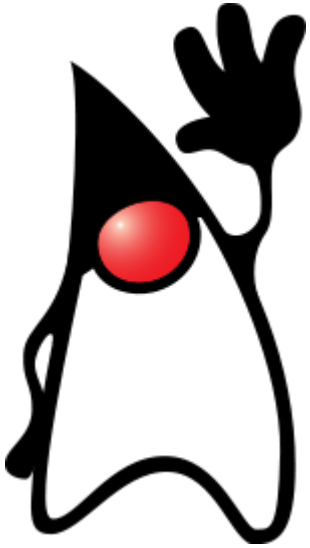
Agregação

Relacionamentos do tipo tem um



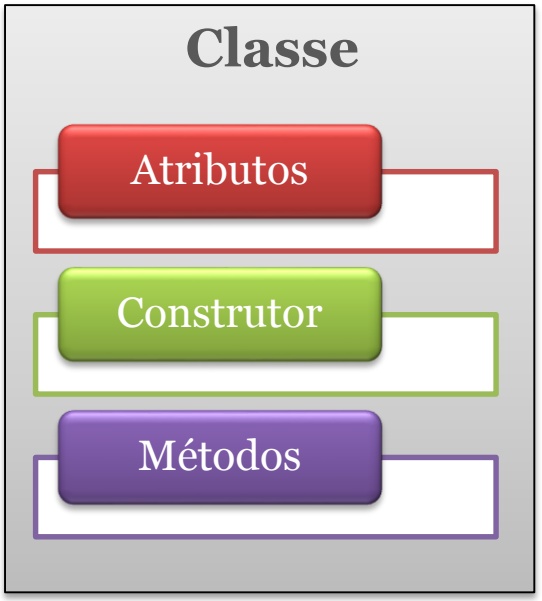
Notação UML para agregação

Java Orientado a Objetos
Métodos, Construtores e Membros Estáticos



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";
    }
}
```



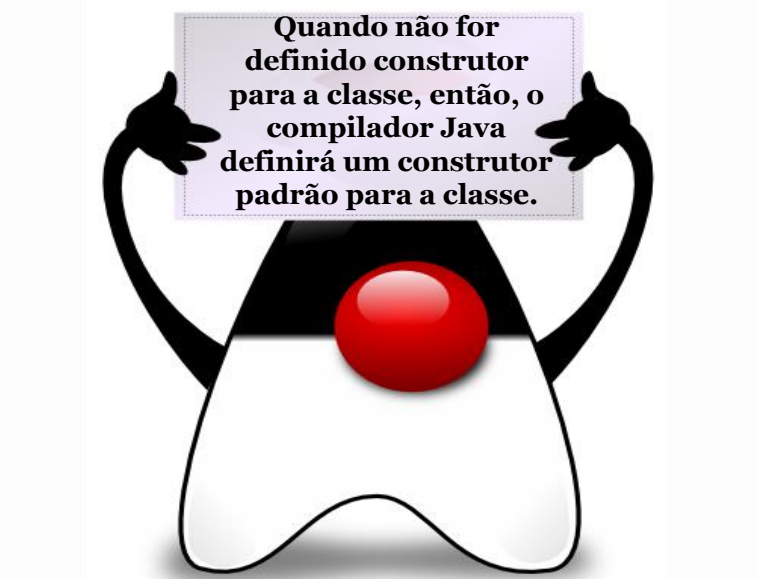
Construtores

Modificadores de Acesso

Nome da Classe onde está o construtor

Argumentos passados por parâmetro, para criação do objeto

```
[modificador] <nomeClasse> (<argumentos>*) {
        <instrução>*
}
```

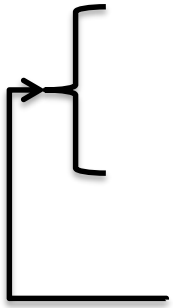


Overloading de Construtores

```
public Carro() {

    //Qualquer código de inicialização aqui
}

public Carro( String placa ) {

    this.placa = placa;
}

public Carro( String modelo, String placa ) {

    this.modelo = modelo;
    this.placa = placa;
}

public Carro( String modelo, String cor, int ano, String placa

    this.modelo = modelo;
    this.cor = cor;
    this.ano = ano;
    this.placa = placa;
}
```

Construtores com diferentes tipos de parâmetros (Overloading - Sobrecarga)

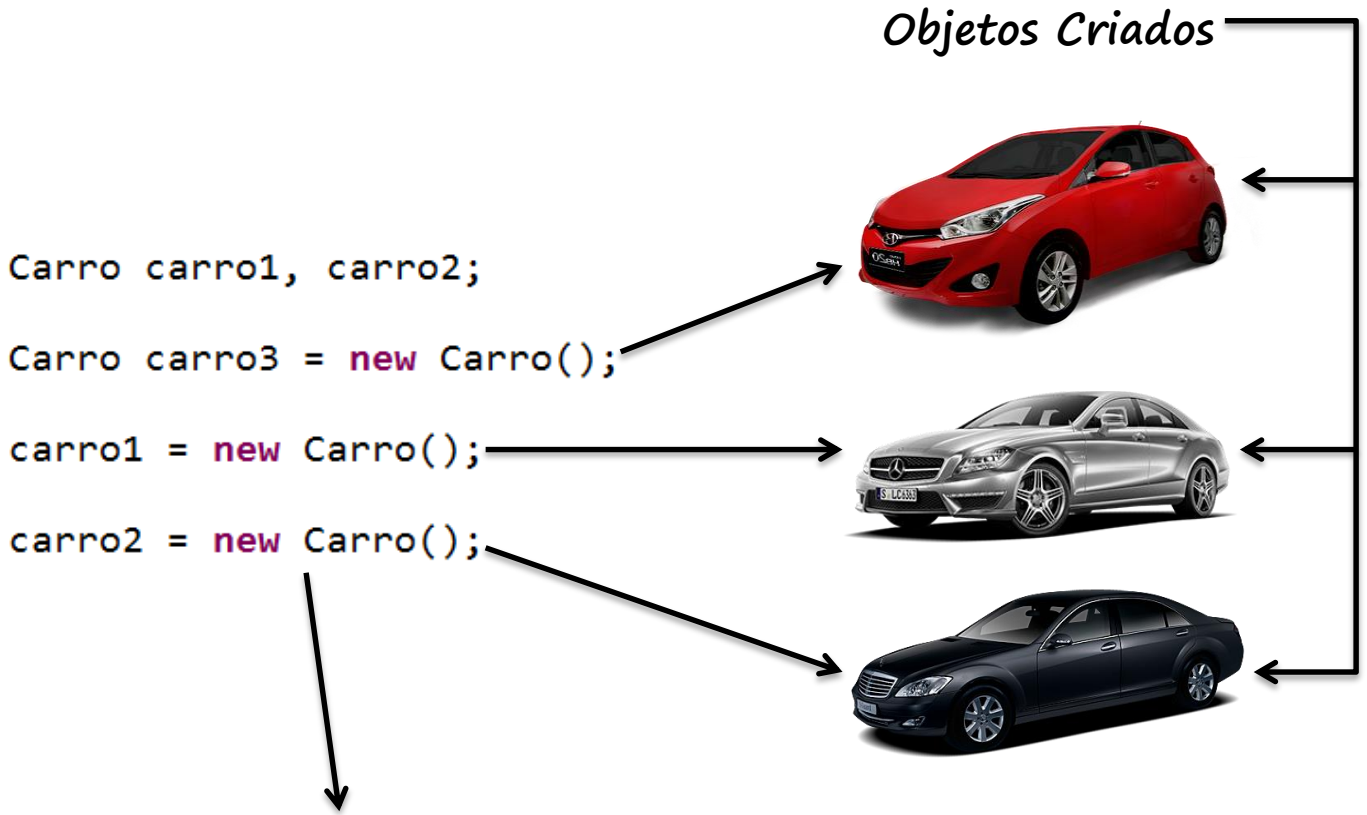
Utilizando o Construtor this()



```
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.

Instância de Classes



Chamada do construtor

sempre vem acompanhada do operador "new"

Invocação de Métodos

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

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

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

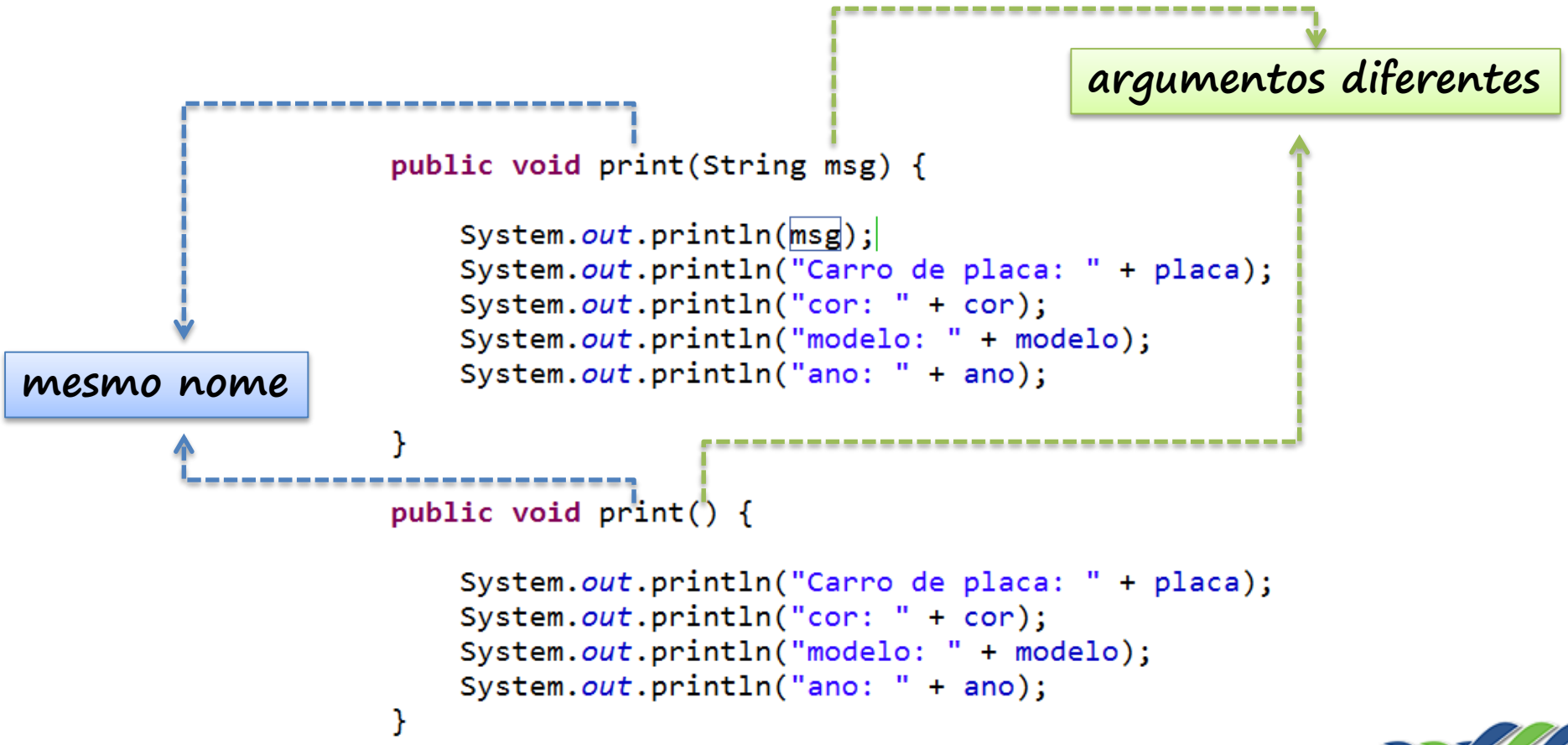
Passagem de Parâmetro por Valor

```
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;
      }
}
```

Passagem de Parâmetro por Referência

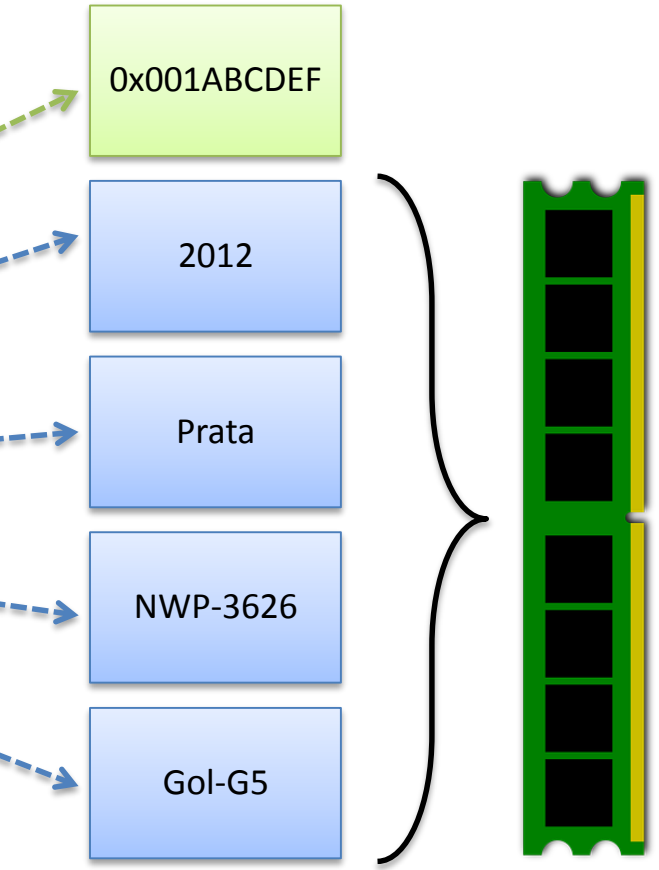
```
public  class PassagemPorReferencia {
      public static void main(String[] args) {
             // criar um array de inteiros
            int[] idades = { 10, 11, 12 };
            // exibir os valores do array
            for (int i = 0; i < idades.length; i++) {
                  System.out.println(idades[i]);
            }
            // 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;--------------------+
            }
      }
}
```

Sobrecarga de métodos (Overloading)



Referência de Objetos

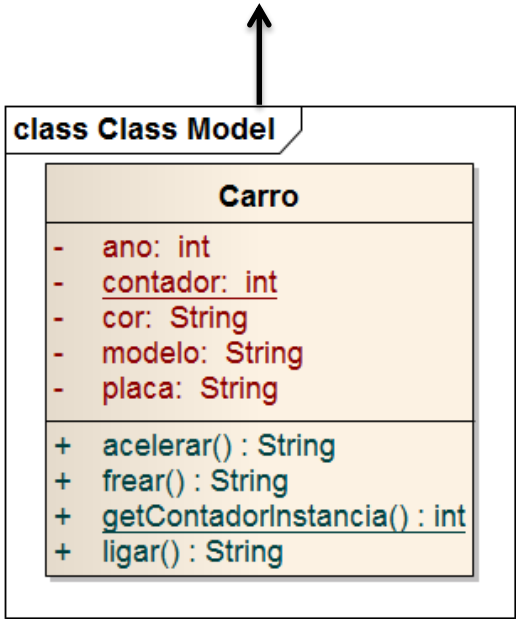
```
// criando instancia de Carro
Carro carro1 = new Carro();

// atribuindo valores
carro1.setAno(2012);
carro1.setCor("Prata");
carro1.setPlaca("NWP-3626");
carro1.setModelo("Gol-G5");
```



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



Membros estáticos

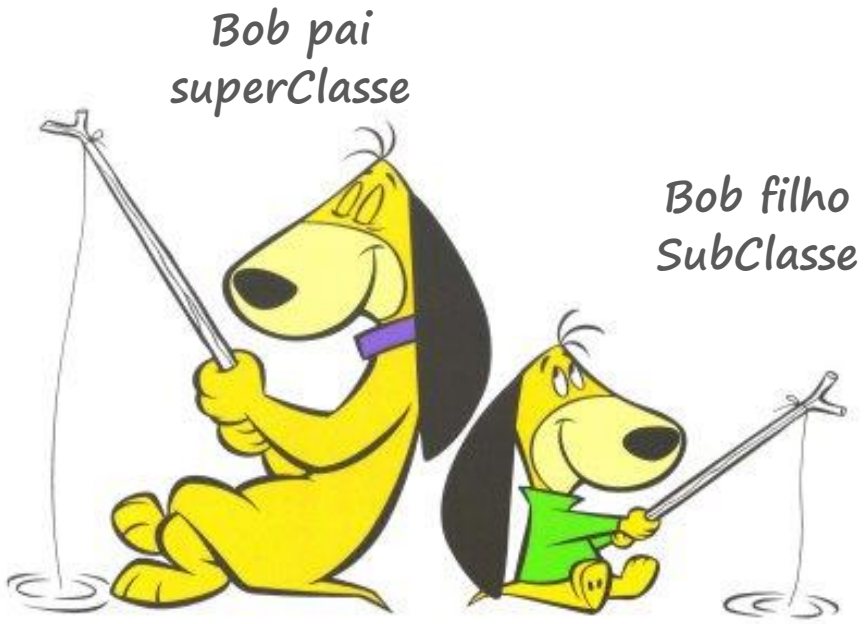
NomeClasse.nomeMetodoEstatico(argumentos);

*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*

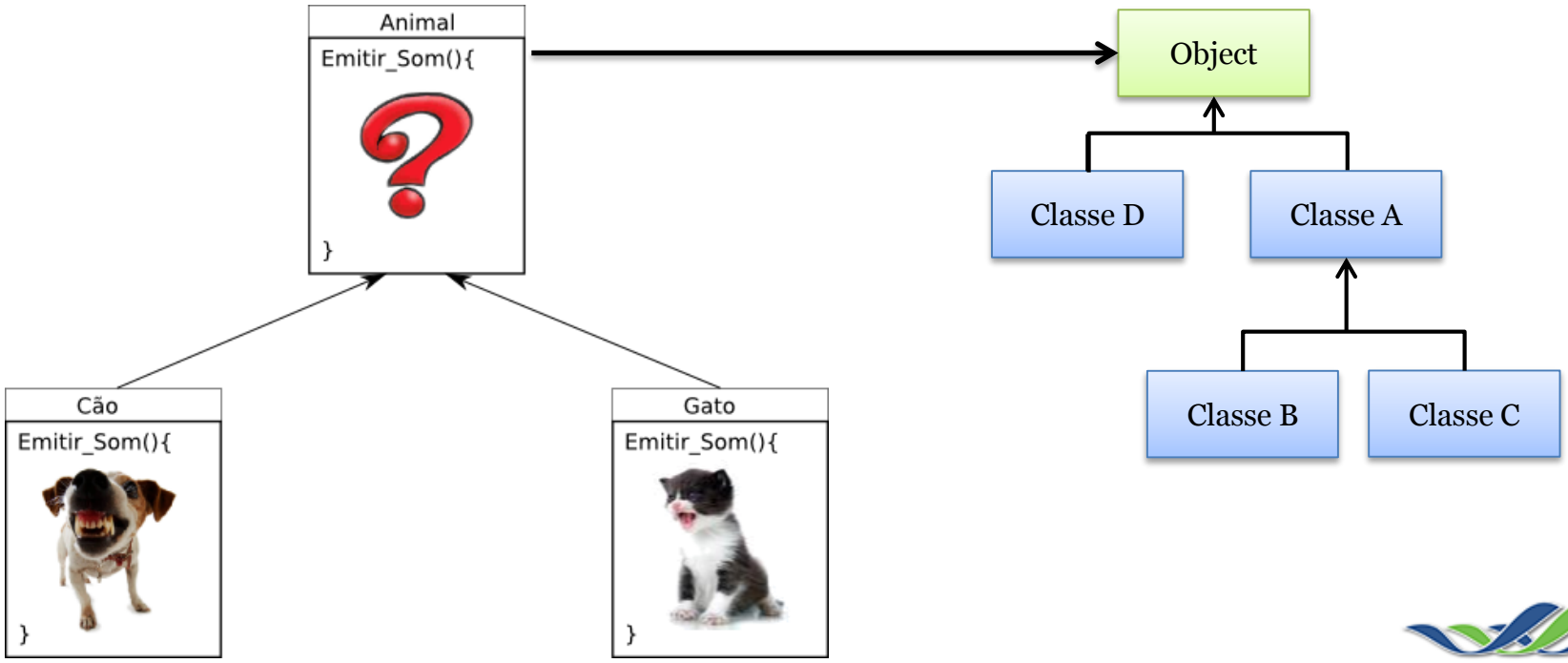
```
Carro carro1, carro2;

Carro carro3 = new Carro();

carro1 = new Carro();

carro2 = new Carro();

System.out.println(Carro.getContadorInstancia() + " instâncias criadas");
```

Java Orientado a Objetos
Herança e Polimorfismo



Herança

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



SuperClasse e SubClasse

Qualquer classe acima de uma classe específica na hierarquia de classes

Generalização

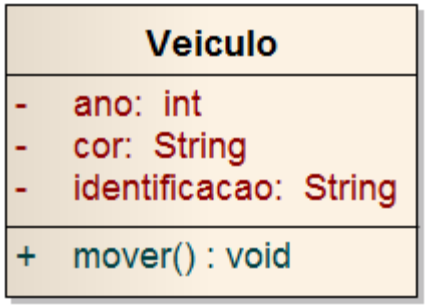
Permite codificar um método apenas uma única vez e este pode ser usado por todas as subclasses

Qualquer classe abaixo de uma classe específica na hierarquia de classes

Especialização

Uma subclasse necessita apenas implementar as diferenças entre ela própria e sua classe pai.

Herança – classe Veiculo

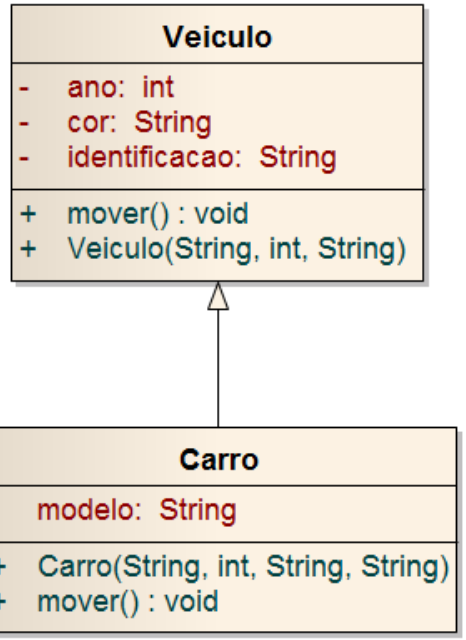



```java
public class Veiculo {// nome da classe

    // (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");
    }

}
```

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.



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**

Polimorfismo

Generalização

Especialização

Sobreposição de Métodos @Override

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

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

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

```java
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**.*

Referências polimórficas

Referência

Instância

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

    Veiculo veiculo = new Carro("Cinza", 2012, "NWP-2552", "Gol");
    veiculo.mover();
    veiculo = new Aviao("Prata", 2013, "NWP-2552");
    veiculo.mover();
    veiculo = new Barco("Branco", 2008, "NWP-2552");
    veiculo.mover();
}
```

Tem que correr

Tem que voar

Tem que navegar

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

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.

Determinando a Classe de um Objeto

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

Carro

↑ -------`System.out.println(veiculo[0].getClass().getSimpleName());`

↓ -------`System.out.println(veiculo[1].getClass().getSimpleName());`

Barco

```
System.out.println(veiculo[0] instanceof Carro);
System.out.println(veiculo[1] instanceof Barco);
```

Retorna true se retornar objeto do tipo Especificado

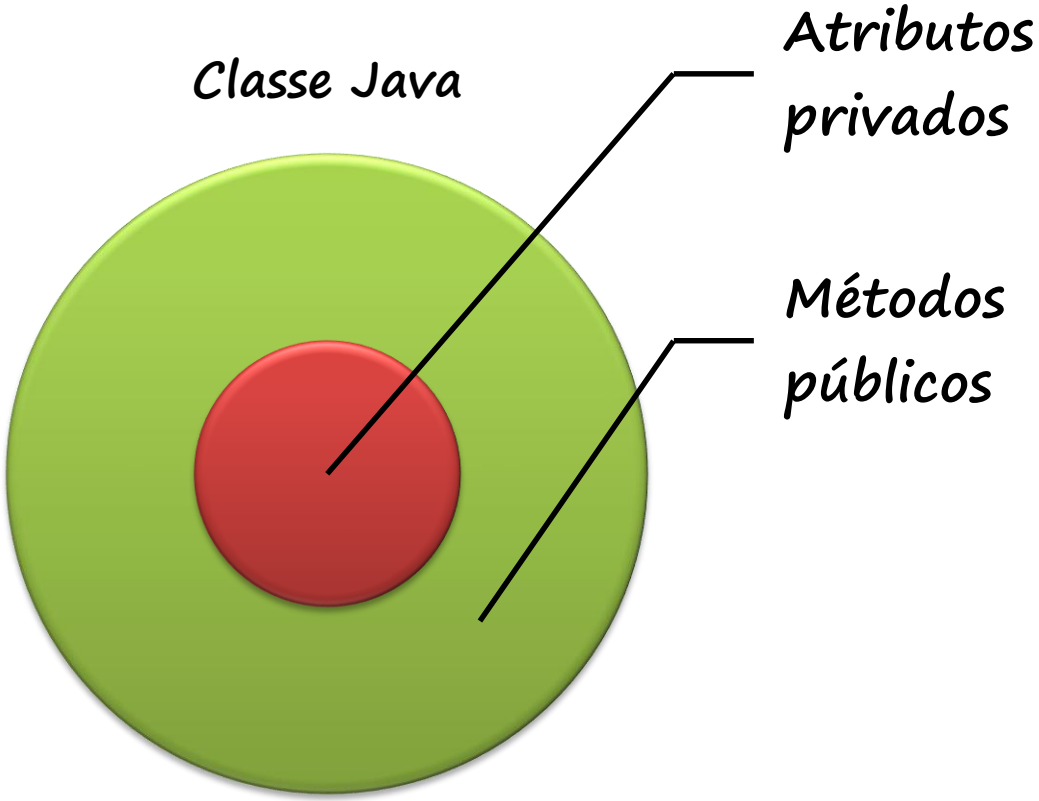
*Um dos problemas que enfrentaremos ao lidar com **referências genéricas** para **objetos** de **subclasses**, é que não sabemos mais a qual classe **pertence a referência** armazenada.*

Modificador de método *final*

Impede o polimorfismo das subclasses por override

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

Encapsulamento



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

Métodos de Configuração e Captura

```
private String cor;
```

```
public void setCor(String cor) {

    this.cor = cor;
}
```

possibilita ***alteração*** *dos valores (variáveis) por outros objetos.*

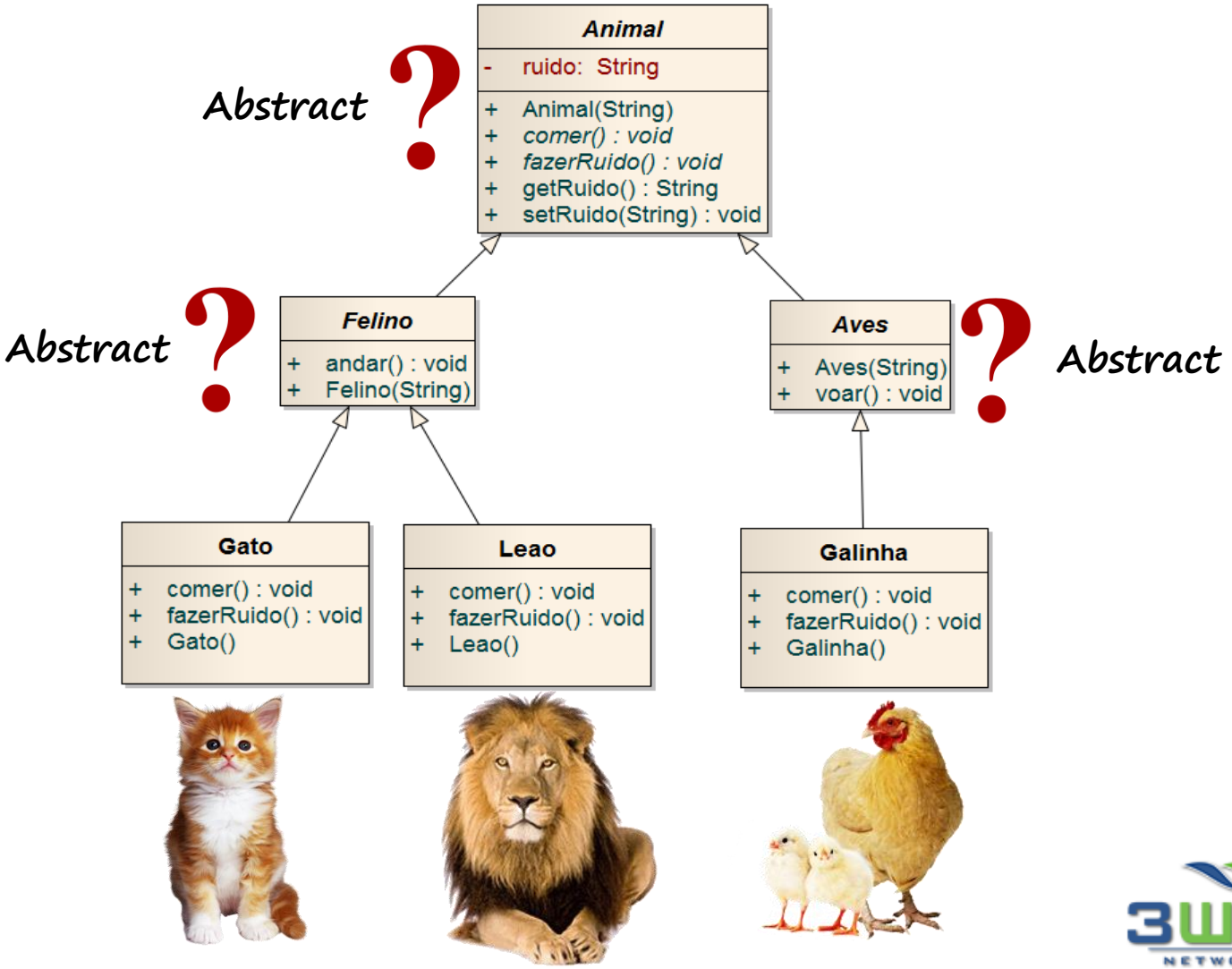
set<NomeAtributo>(<tipo dado> <parâmetro>)

```
public String getCor() {

    return cor;
}
```

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

Java Orientado a Objetos
Classes Abstratas, Internas e Interfaces

Classes Abstratas



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

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");}
}
```

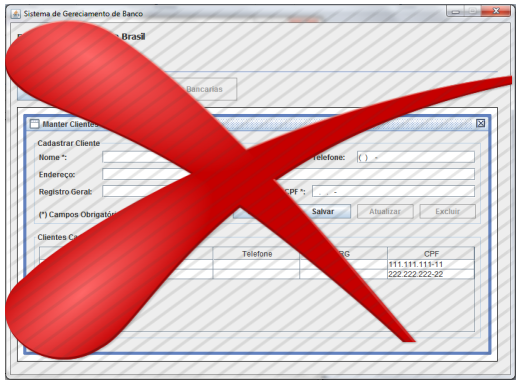
Exemplos de implementação

```
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");
    }

}
```

É um Felino

É um Animal

Interfaces



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

Interfaces por natureza **são abstratas**

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

Notação UML

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

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>*){...}

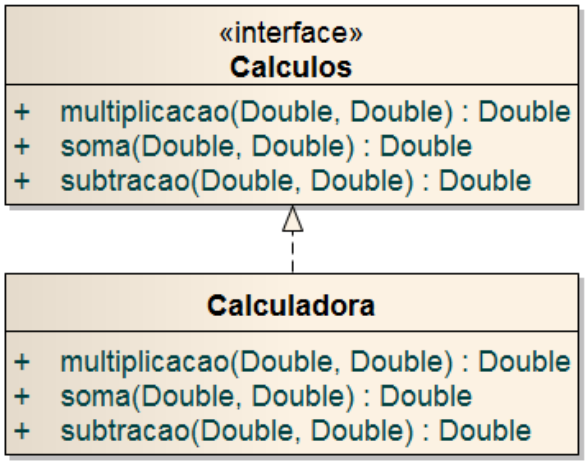
```
public interface Calculos {

    public Double soma(Number x, Number Y);

    public Double subtracao(Number x, Number Y);

    public Double multiplicacao(Number x, Number Y);
}
```

Implementando Interfaces

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

```
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;
    }
}
```

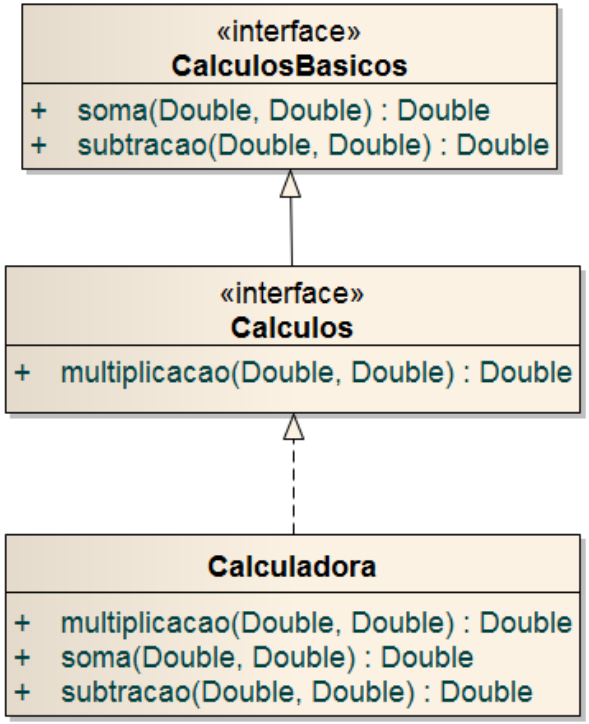
Notação UML



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);

}
```

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***

Métodos de Extensão

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

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
```

Classes Interna Anônima

Classe interna sem nome com propósito de escopo limitado, utilizado para declarar e instanciar um objeto um 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 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);
            }
        });
```

Menos código

```
@FunctionalInterface
public interface Comparator<T>
```

Interface Funcional, só tem um único método abstrato

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

Lambda Métodos de Referência

Lambda com métodos que já existem, não são anônimos.
Você pode referenciar pelo nome pode ser

Classe::nomeMétodoEstático
instanciaClasse::nomeMétodoInstância
Classe::new (referência ao construtor)

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

Método anônimo na expressão lambda

```
public int compararPeloNome(Fruta f1, Fruta f2) {
    return f1.nome.compareTo(f2.nome);
}
```

Método de instância definido numa classe

```
Collections.sort(frutas,refClass::compararPeloNome);

    frutas.forEach(System.out::print);
```

Método de referência usado numa expressão lambda

Método de referência, estático

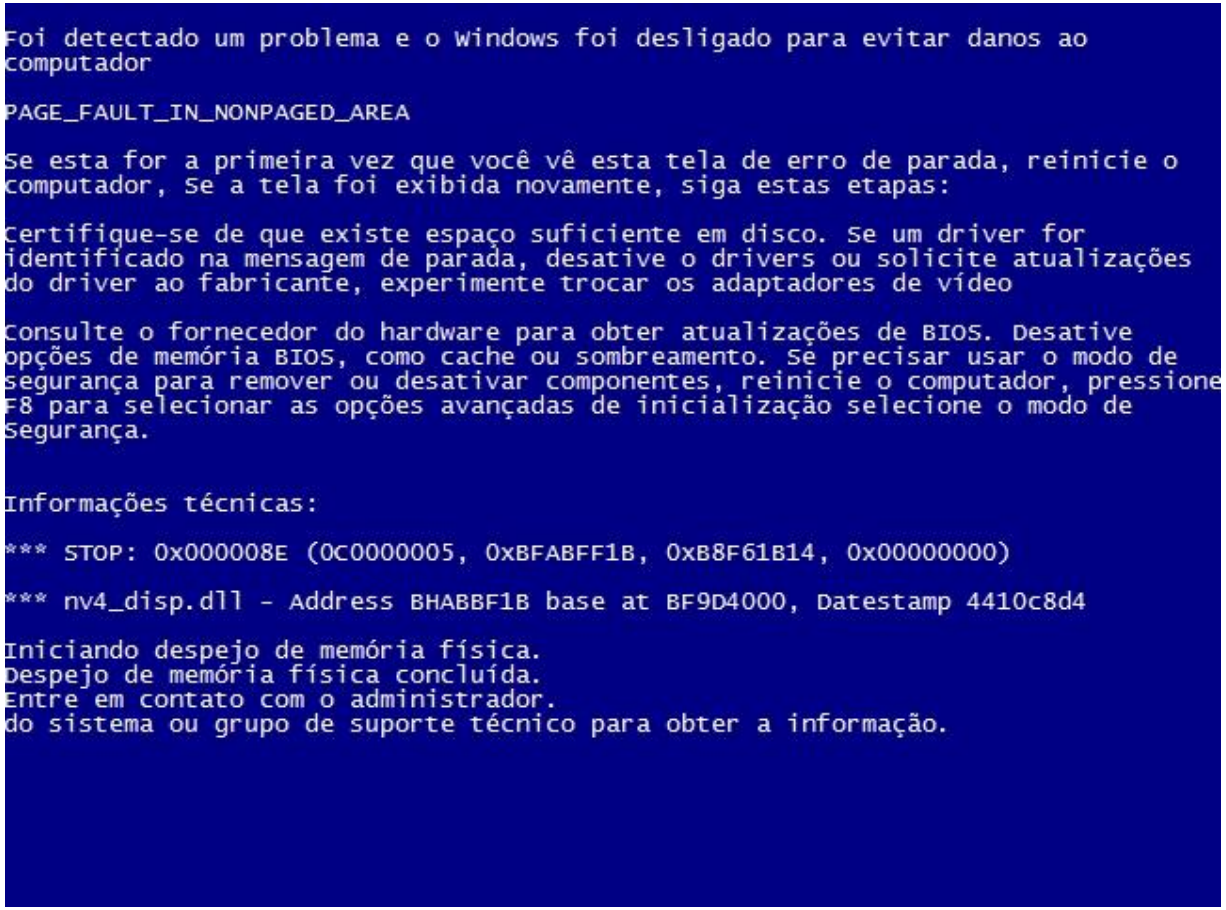
Tipos Enumerados

O tipo **enum** estende implicitamente a classe **java.lang.Enum.** As enumerações podem ter construtores, métodos, variáveis. Cada elemento é uma instância do enum.

```
public enum EnumEstacoes {

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

    private EnumMes inicio, fim;

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

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

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

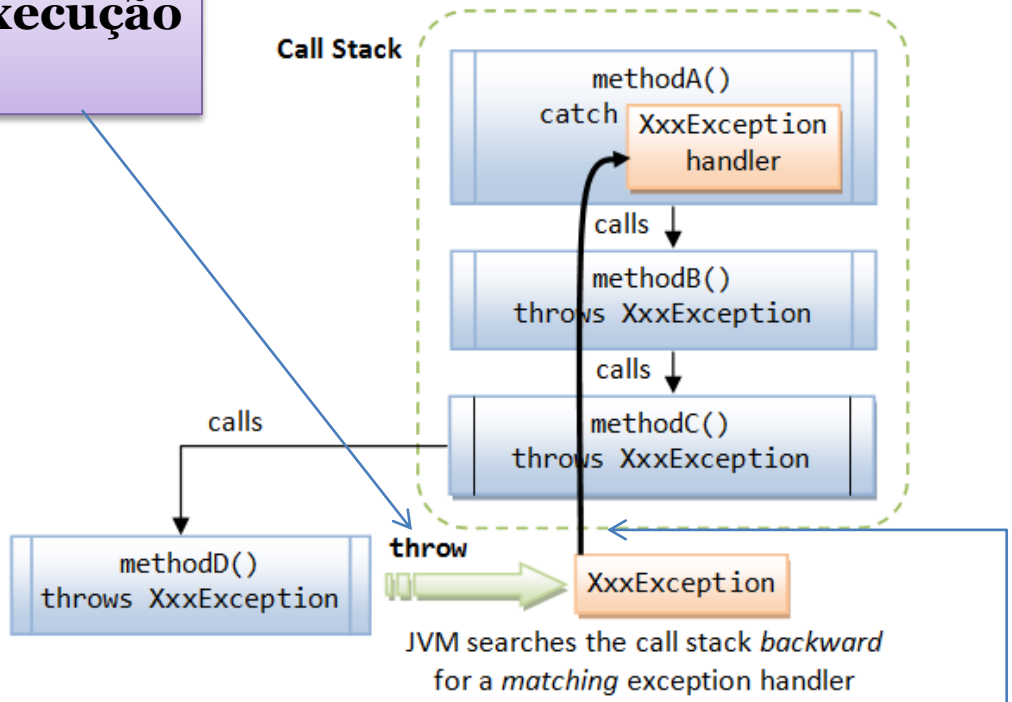
- Como as classes, as enumerações podem ser declaradas dentro de classes, como estáticas e locais.

Java Orientado a Objetos
Exceções



Exceções

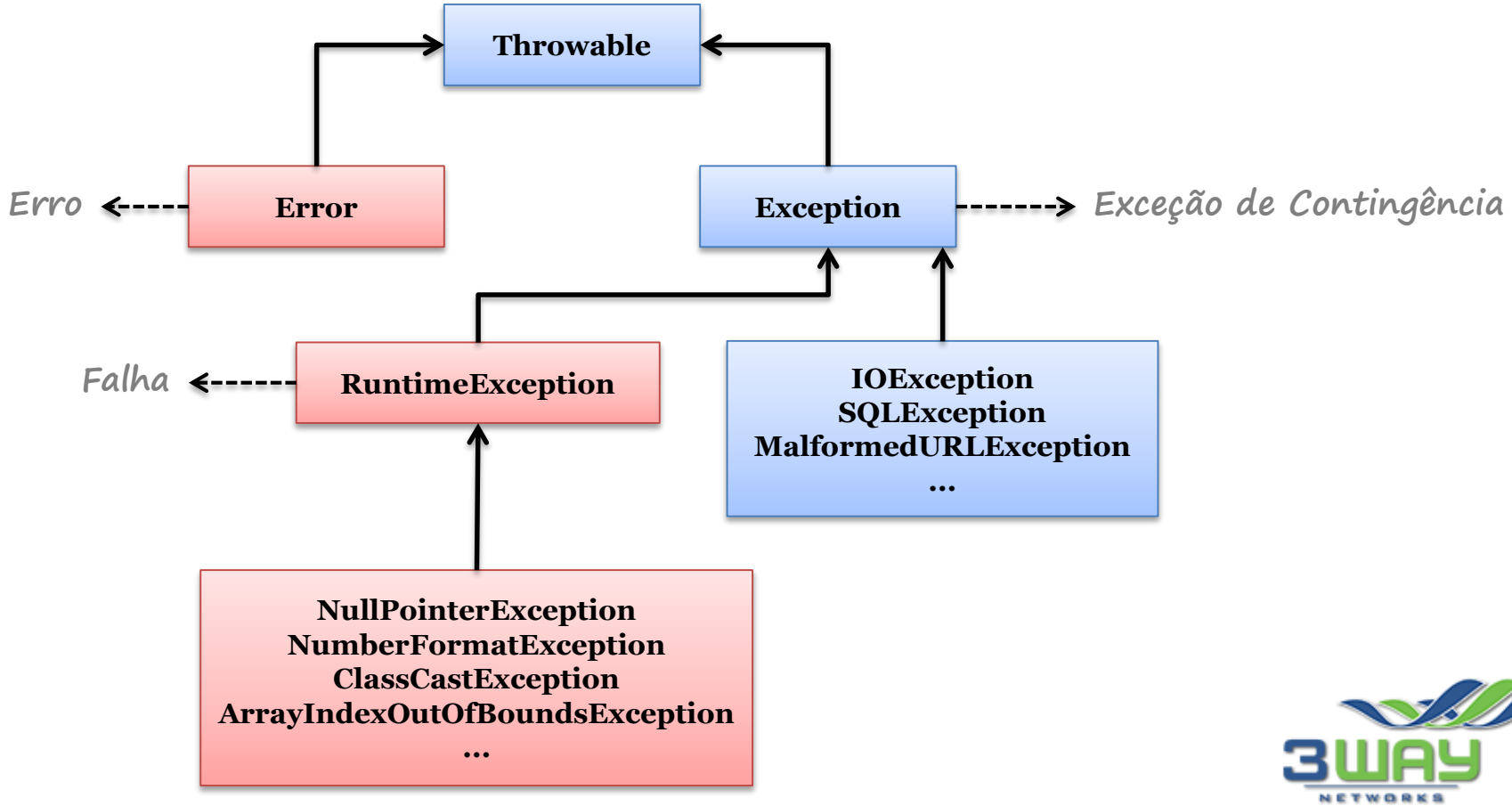
Erros que ocorrem **durante** a **execução** do programa



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

Categoria de Exceções

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



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("Indice 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***

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

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");
        }
    }
}
```

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

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

Throw e Throws

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

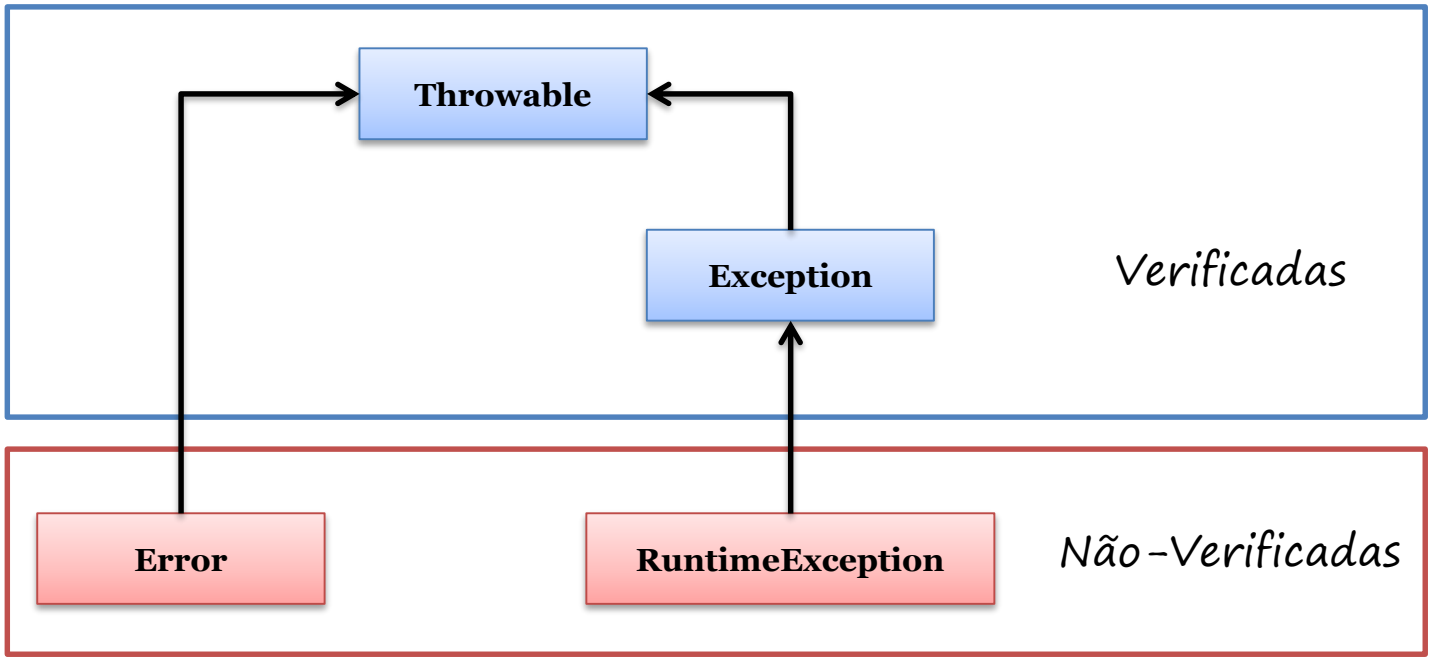
```
public class Calculadora {

    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;
    }
}
```

Tratamento da exceção

Desviando a exceção

Lançamento da exceção

Exceções Verificadas e Não-Verificadas



Criando Exceções

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

Atributos de objetos e construtores podem ser adicionados à classe

```
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;
}
```

Sobrepondo Métodos e Exceções

```java
public class ClasseA {
    public void metodoA() throws RuntimeException {
        // operações que podem lançar exceção
    }
}
```

Não Pode, a classe Exception é superclasse de RuntimeException

```java
public class ClasseC extends ClasseA {
    @Override
    public void metodoA() throws Exception {
        // operações que podem lançar exceção
    }
}
```

Pode,é subclasse de RuntimeException

```java
public class ClasseB extends ClasseA {
    @Override
    public void metodoA() throws NullPointerException {
        // operações que podem lançar exceção
    }
}
```

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 Objetos
Tipos Genéricos

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**
Destrói benefícios de uma linguagem com **tipos fortemente definidos**	Preserva benefícios da checagem de tipos

Declarando uma Classe utilizando Generics

Classe não trabalha com nenhuma referência a um tipo específico

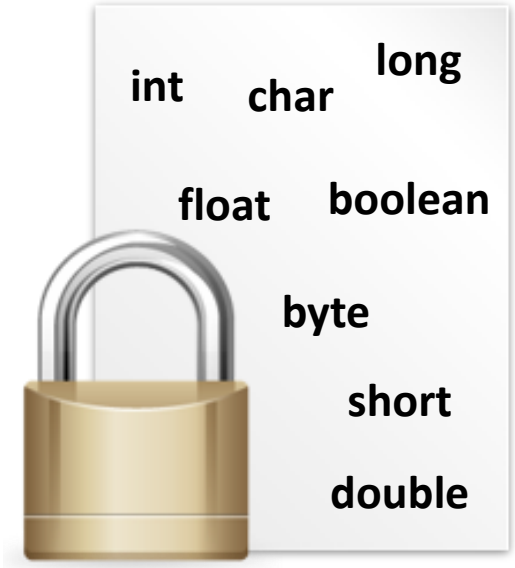
```
public class ManipulaArray<T> { // Contem o parâmetro <?>

    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

Limitação “Primitiva”

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



Solução

Utilizar Classes Wrapper(Empacotadoras)

Limitando Genéricos

```
public class ColecaoBichoFelino<T extends Felino> {

    T[] animais;
}
```

Leão

Gato

Coringa <?>

<? extends T>

Aceita T e todos os seus descendentes

<? super T>

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 e 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]

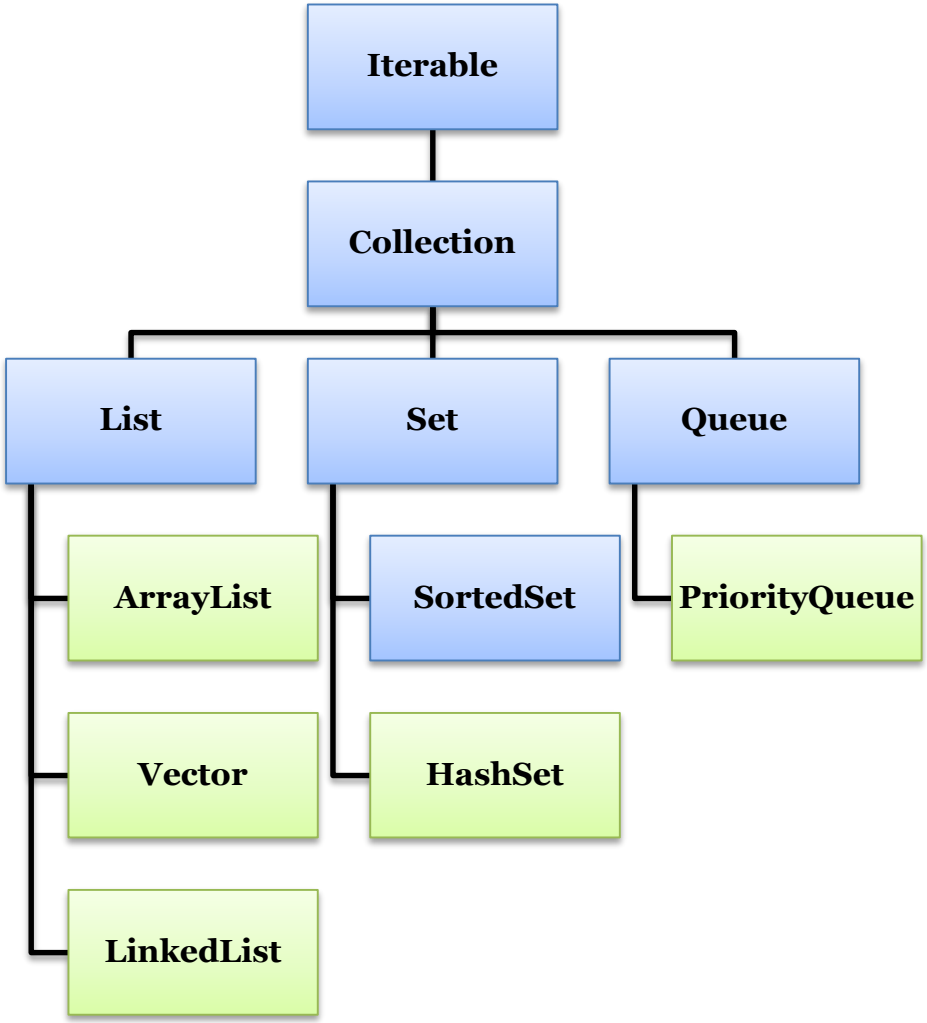
Java Orientado a Objetos
Collections

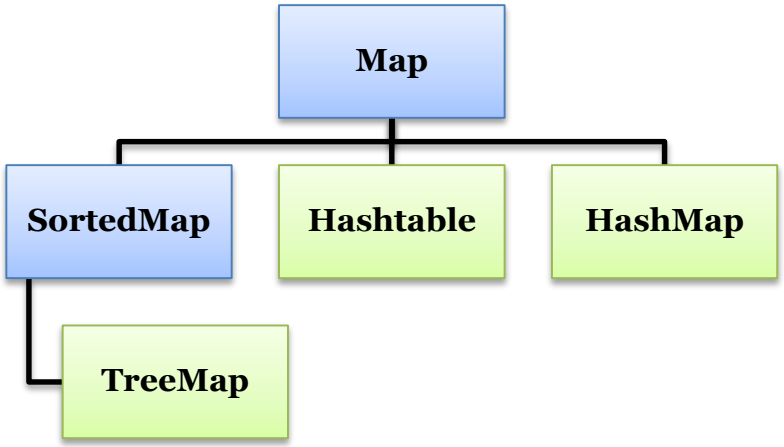
Java Collections

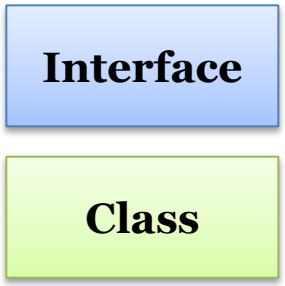
É um objeto onde podemos agrupar vários elementos (outros objetos)

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

Java Collections - Hierarquia



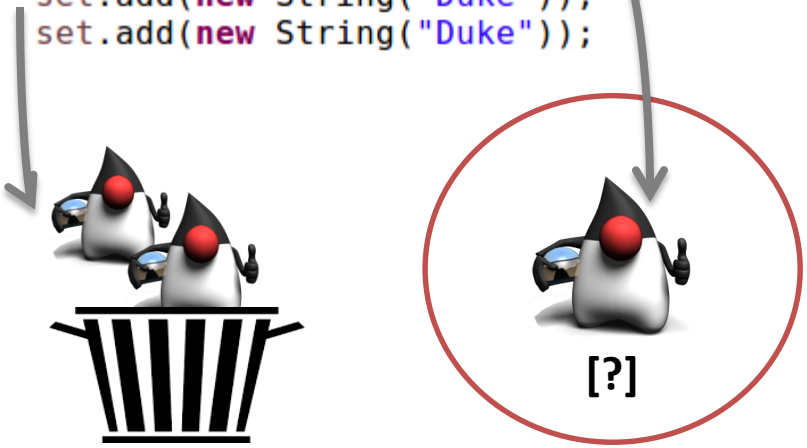




Interfaces Set e List

Interface Set

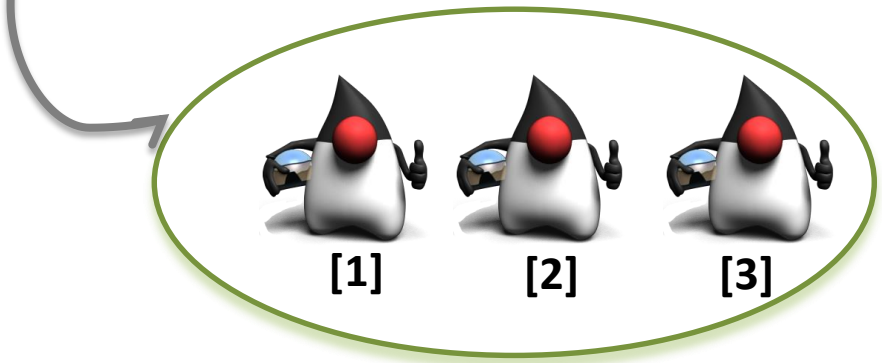
```
Set<String> set = new HashSet<>();

set.add(new String("Duke"));
set.add(new String("Duke"));
set.add(new String("Duke"));
```



Coleções não ordenadas que não contém duplicidades

Interface List

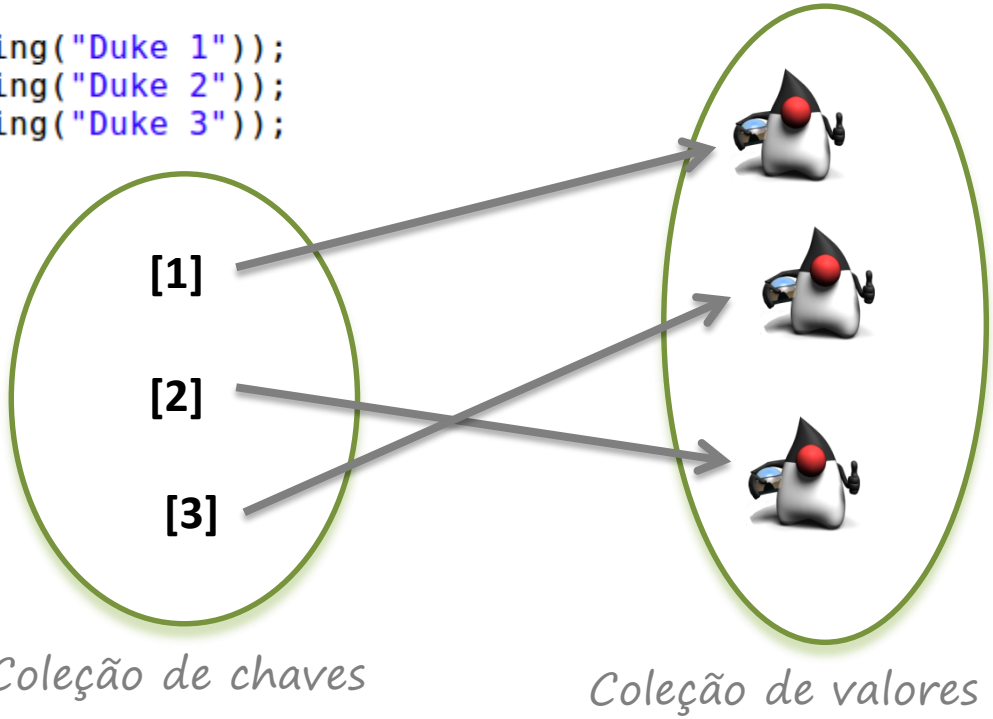
```
List<String> list = new ArrayList<>();

list.add(new String("Duke"));
list.add(new String("Duke"));
list.add(new String("Duke"));
```



Coleções de classes ordenadas onde as duplicidades são permitidas.

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

Generics e Coleções Java

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

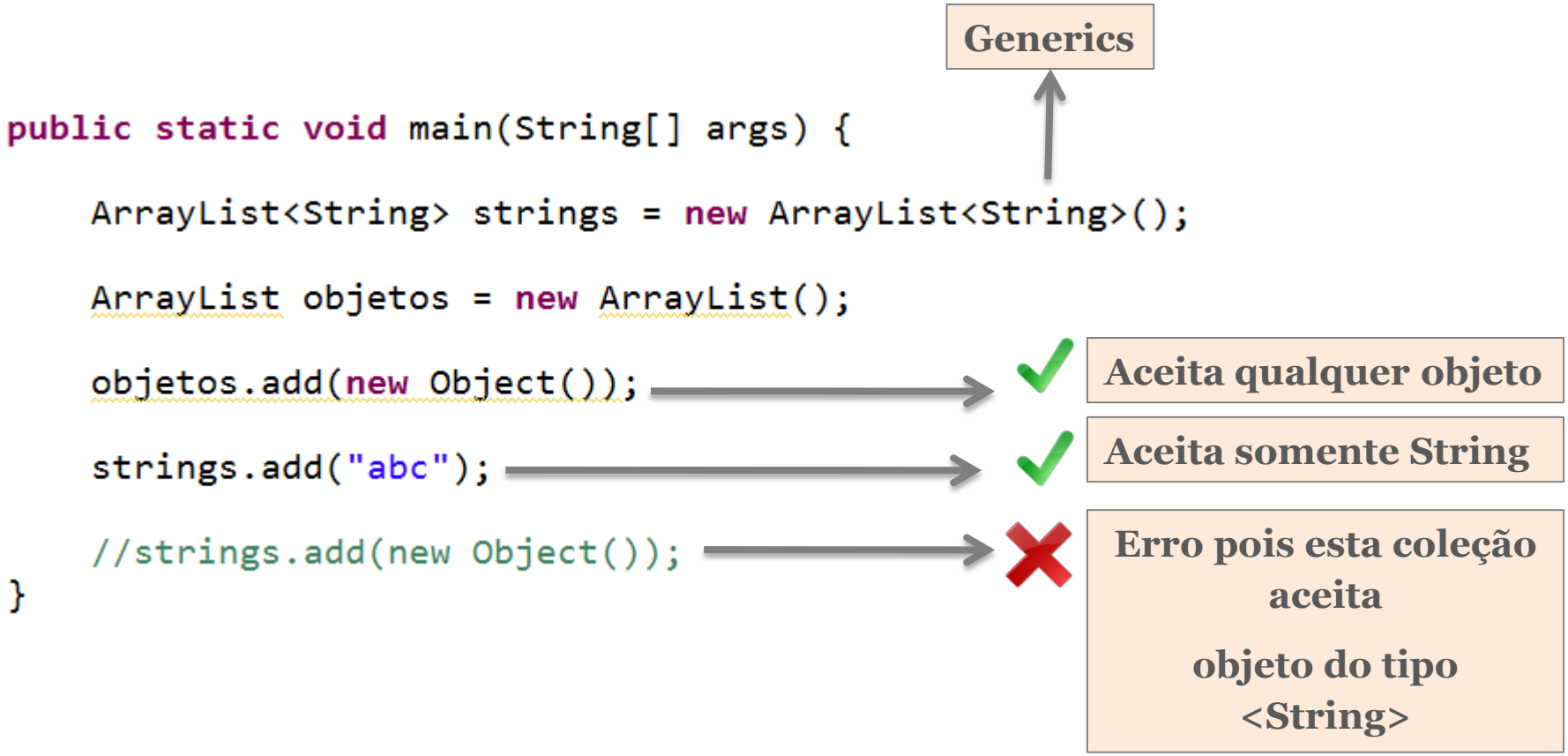
```
boolean add(E e)
boolean addAll(Collection<? extends E> c)
boolean contains(Object o)
boolean containsAll(Collection<?> c)
boolean remove(Object o)
boolean removeAll(Collection<?> c)
boolean retainAll(Collection<?> c)
void clear()
int size()
boolean isEmpty()
Object[] toArray()
<T> T[] toArray(T[]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

Generics

<?> coringa

```
//JAVA 8
default Spliterator<E> spliterator()
default Stream<E> stream()
default Stream<E> parallelStream()
```

Generics e Coleções Java



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()
}
```

A partir de Java 8

```
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();
}
```

Percorrendo Collections

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

Enhaced-for	Iterator

`for (String str : strings) {`
`    System.out.println(str);`
`}`
	`Iterator<String> iterator = strings.iterator();`
	`while (iterator.hasNext()) {`
	`    System.out.println(iterator.next());`
	`}`

Java 8 – método forEach

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

Usando forEach de forma imperativa com anonymous inner class

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

Usando forEach com Lambada

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

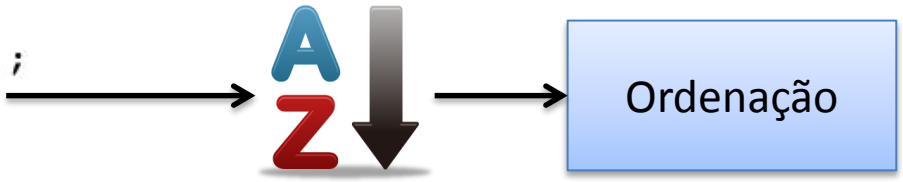
```
strings.forEach(System.out::println);
```

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

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);
            // lista ordenada
            System.out.println(lista);

      }
}
```



Interface Comparable

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

-1 **0** **1**

< **=** **>**

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

Prepara um "duto" de objetos e repassa para função intermediária sorted()

Pega objetos do "duto" e invoca compareTo(), repassa para função filter()

```
colecaoDeObjetos
        .stream()

        .sorted()

        .filter((o) -> o.getNome().startsWith("m"))

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

Usa método de referência para imprimir toString() dos objetos que satisfazeram o filtro

Pega objetos do "duto" e invoca expressão lambida para filtrar objetos com campo nome começando com caractere "m", repassa objetos para função forEach()

Java Orientado a Objetos
Lendo e Escrevendo Arquivos

Console I/O

```
*************** CUNIT CONSOLE - MAIN MENU ***********************
(R)un all, (S)elect suite, (L)ist suites, Show (F)ailures, (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 initialization failed for Suite_init_failure.
Running Suite : Suite_clean_failure
        Running test : successful_test_4
        Running test : failed_test_2
        Running test : successful_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        4       3     n/a       2
               tests        13      10       7       3
               asserts      10      10       7       3

*************** CUNIT CONSOLE - MAIN MENU ***********************
(R)un all, (S)elect suite, (L)ist suites, Show (F)ailures, (Q)uit
Enter Command :
```

```java
import java.io.Console;

public class Teste {

    Console con = System.console();
}
```

System.in

System.out

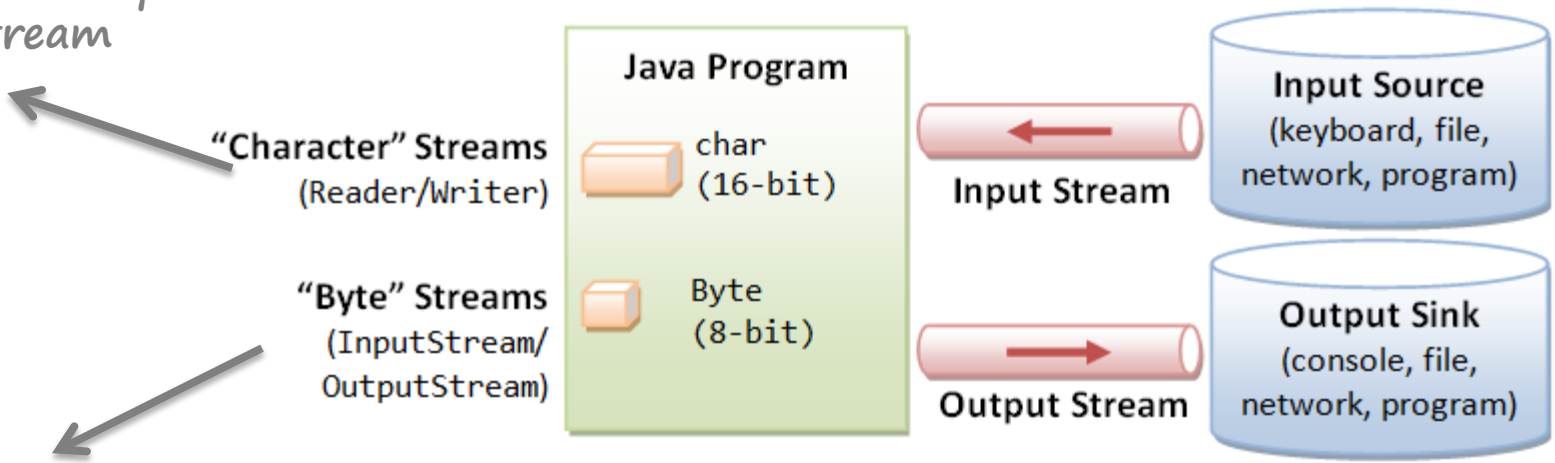
Usando a classe Scanner

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);
}
```

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:

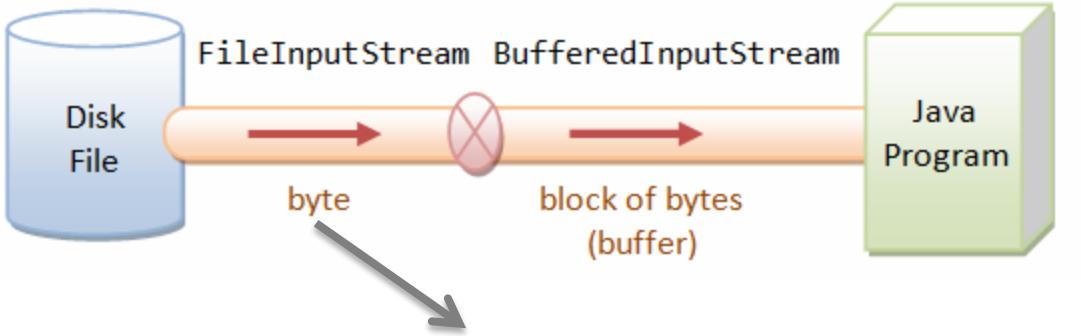
- Text (char): UCS-2
- int, float, double, etc.

External Data Formats:

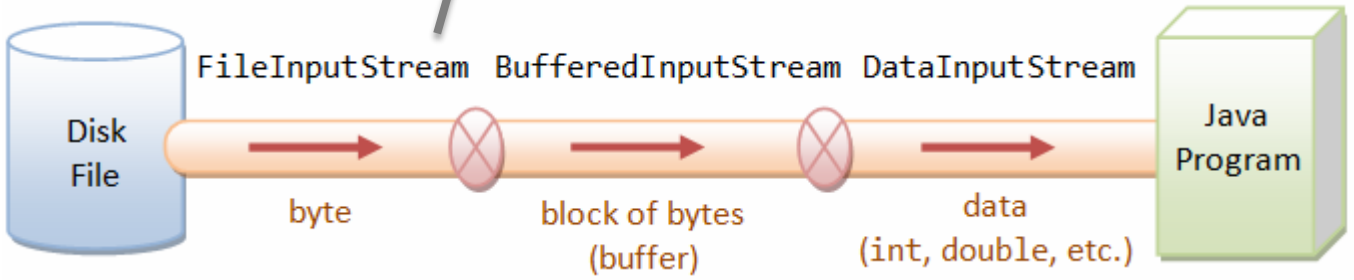
- Text in various encodings (US-ASCII, ISO-8859-1, UCS-2, UTF-8, UTF-16, UTF-16BE, UTF16-LE, etc.)
- Binary (raw bytes)

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

Encadeando I/O Stream




```
InputStream bis = new BufferedInputStream(new FileInputStream(new File("file.zip")));

InputStream dis = new DataInputStream(new BufferedInputStream(new FileInputStream(new File("file.dat"))));
```



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

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

NIO.2 - Path

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

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

```
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 diretório

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

NIO.2 – I/O Streams

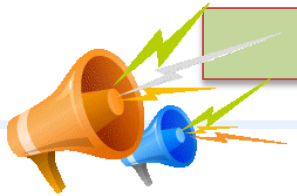
```
String fileStr = "small_file.txt";
Path path = Paths.get(fileStr);
List<String> lines = new ArrayList<String>();
lines.add("Oi,您好! Olá,吃饱了没有?");

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

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);
```

Compatibilidade com Java I/O Básico

```
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);
}
```

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

Use com arquivos Grandes, para usar streams compativeis com java IO

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());
    }
}
```

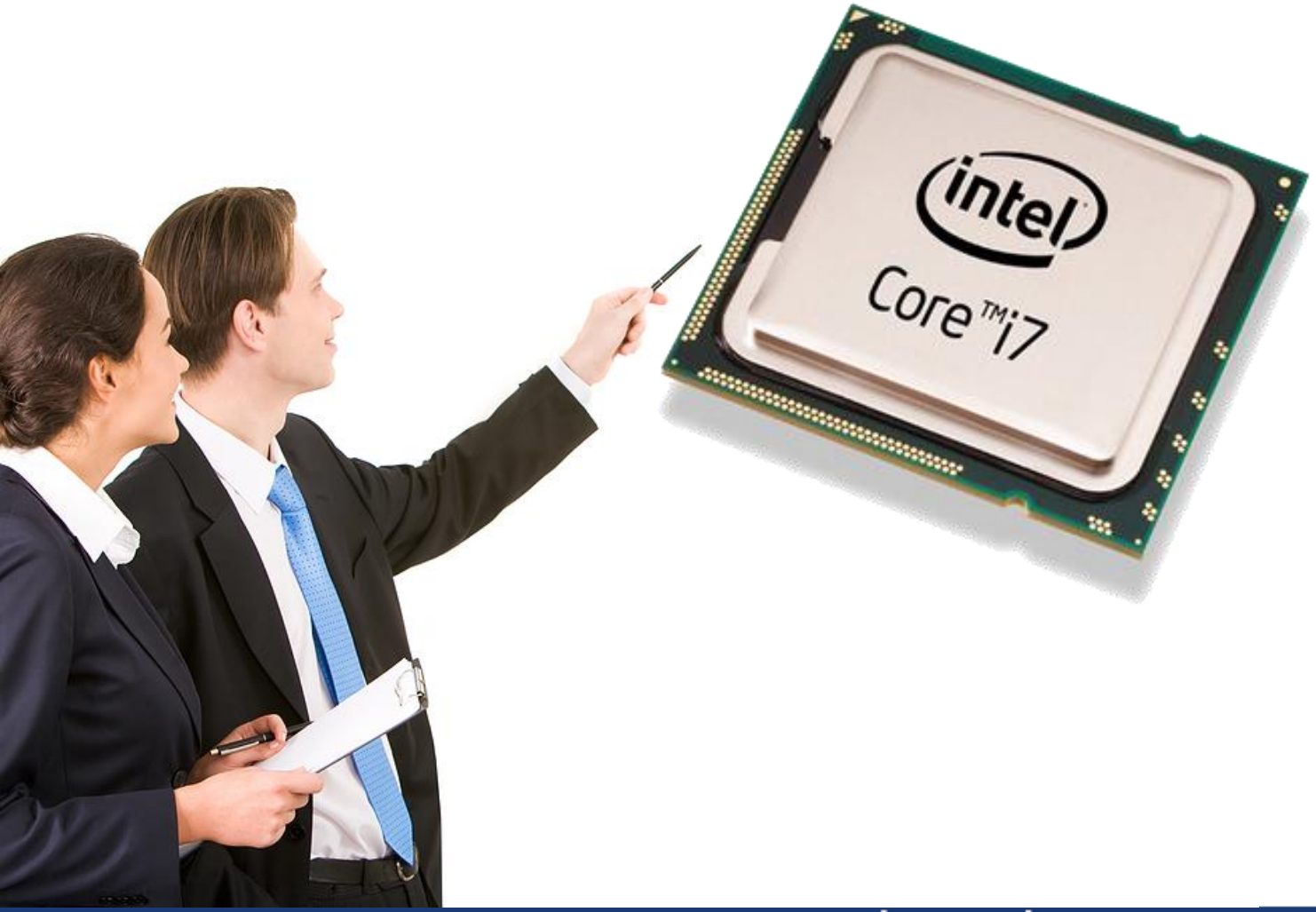
I/O de arquivos conectados a um channel, dados são lidos para o buffer

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 Objetos
Threads

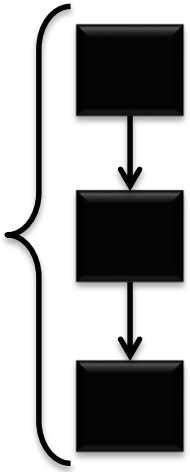


Threads

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

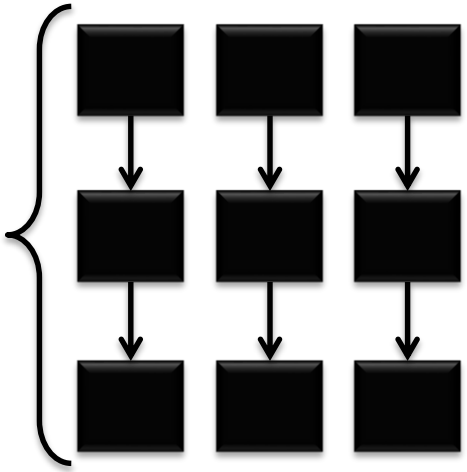
Sequencial

Programa

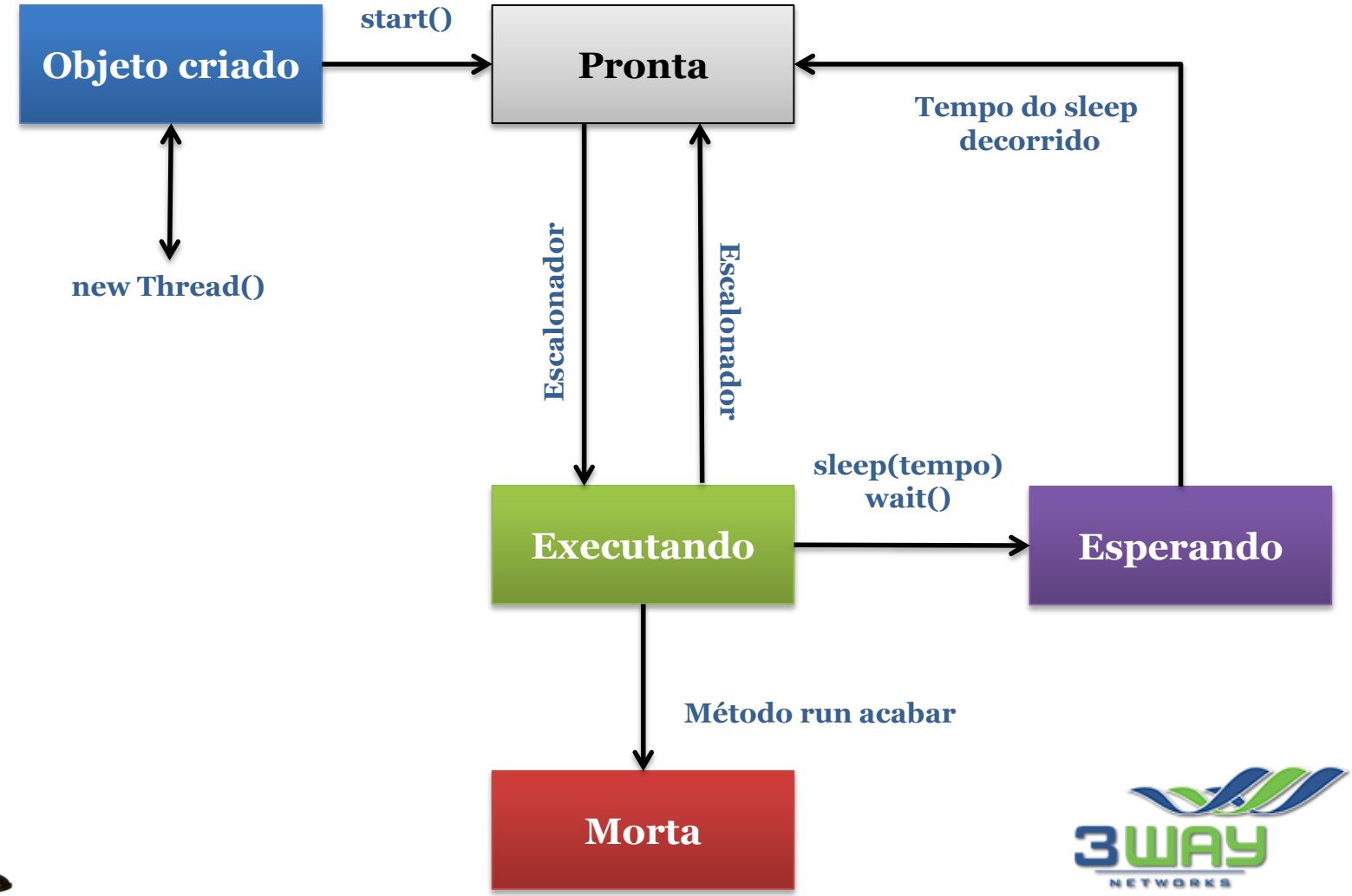


MultiThread

Programa

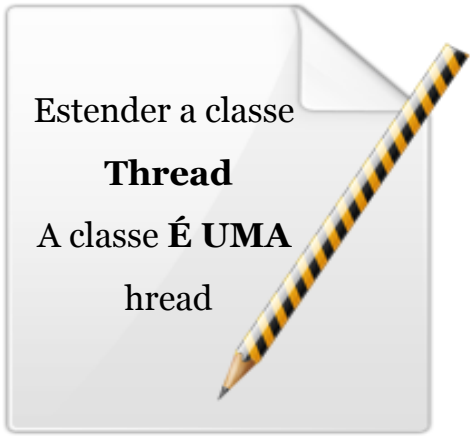


Ciclo de vida de uma Thread

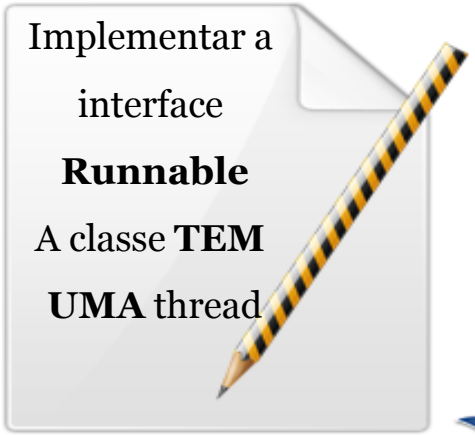


Criando Threads

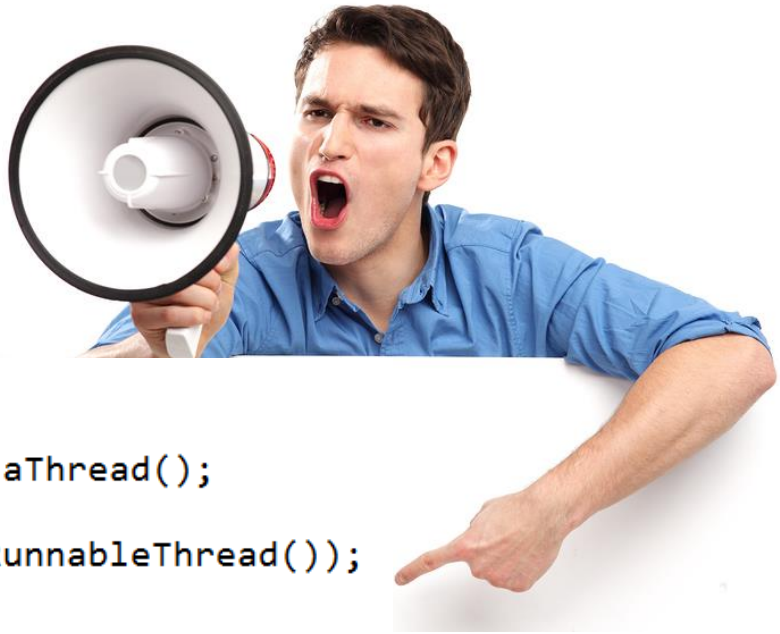
```java
public class HerancaThread extends Thread {

    @Override
    public void run() {
        // conteudo da thread
        System.out.println("extends Thread");
        for (int i = 0; i < 10; i++) {
            System.out.println(i);
        }
    }
}
```




```java
public class RunnableThread implements Runnable {

    @Override
    public void run() {
        // conteudo da thread
        System.out.println("implements Runnable");
        for (int i = 0; i < 10; i++) {
            System.out.println(i);
        }
    }

}
```



Iniciando Threads



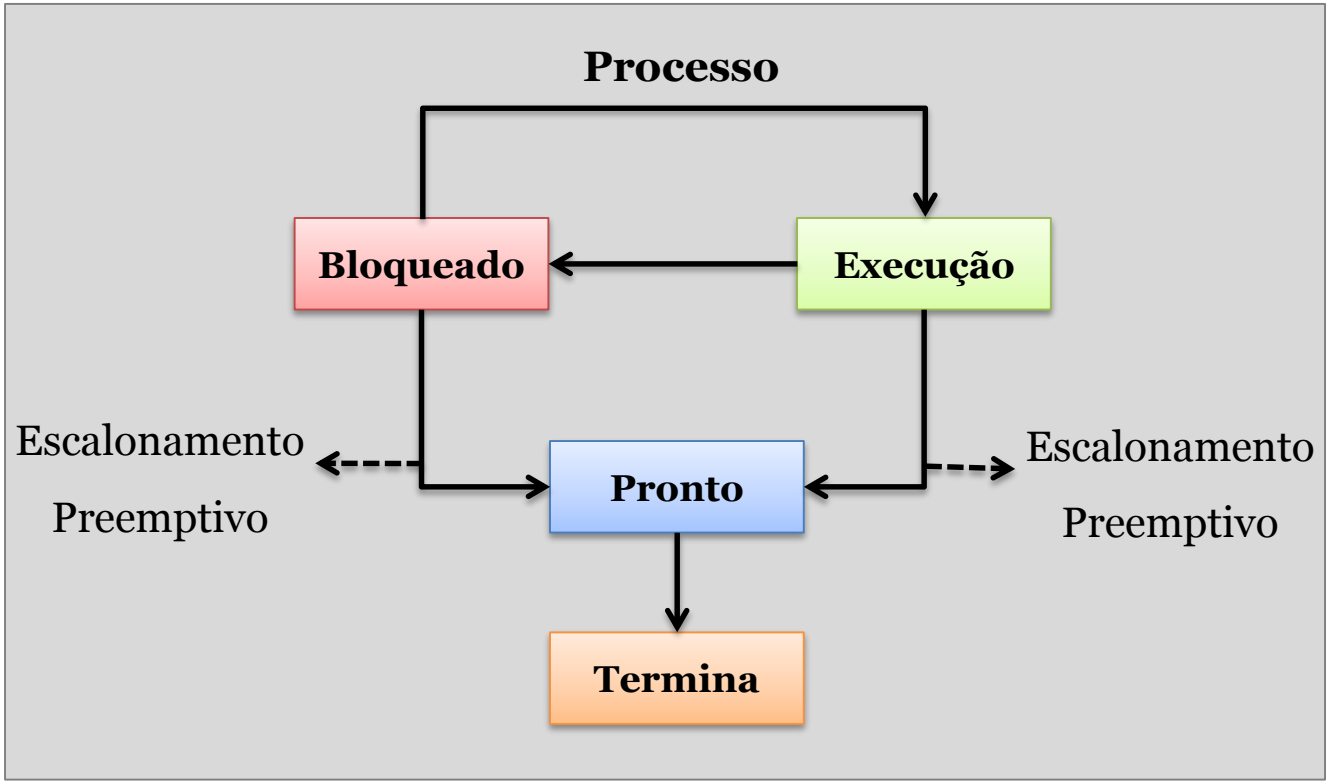
```
public static void main(String[] args) {

    HerancaThread threadSimples = new HerancaThread();

    Thread threadRunnable = new Thread(new RunnableThread());

    threadSimples.start();

    threadRunnable.start();
}
```

Inicia execução do método run()

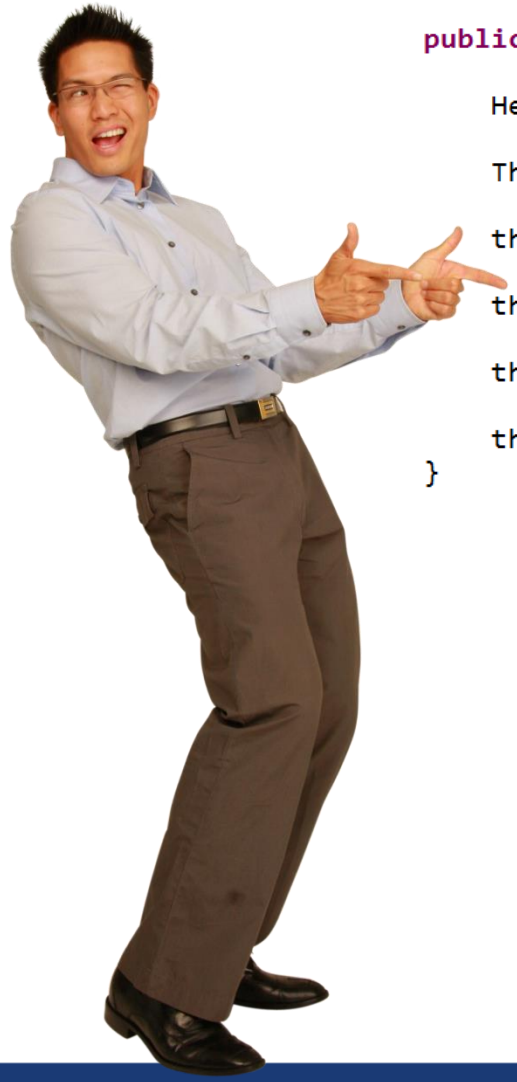
presente nas Threads

Escalonamento da Thread

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



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

Sincronização

Métodos **synchronized** têm um lock para acesso ao objeto.

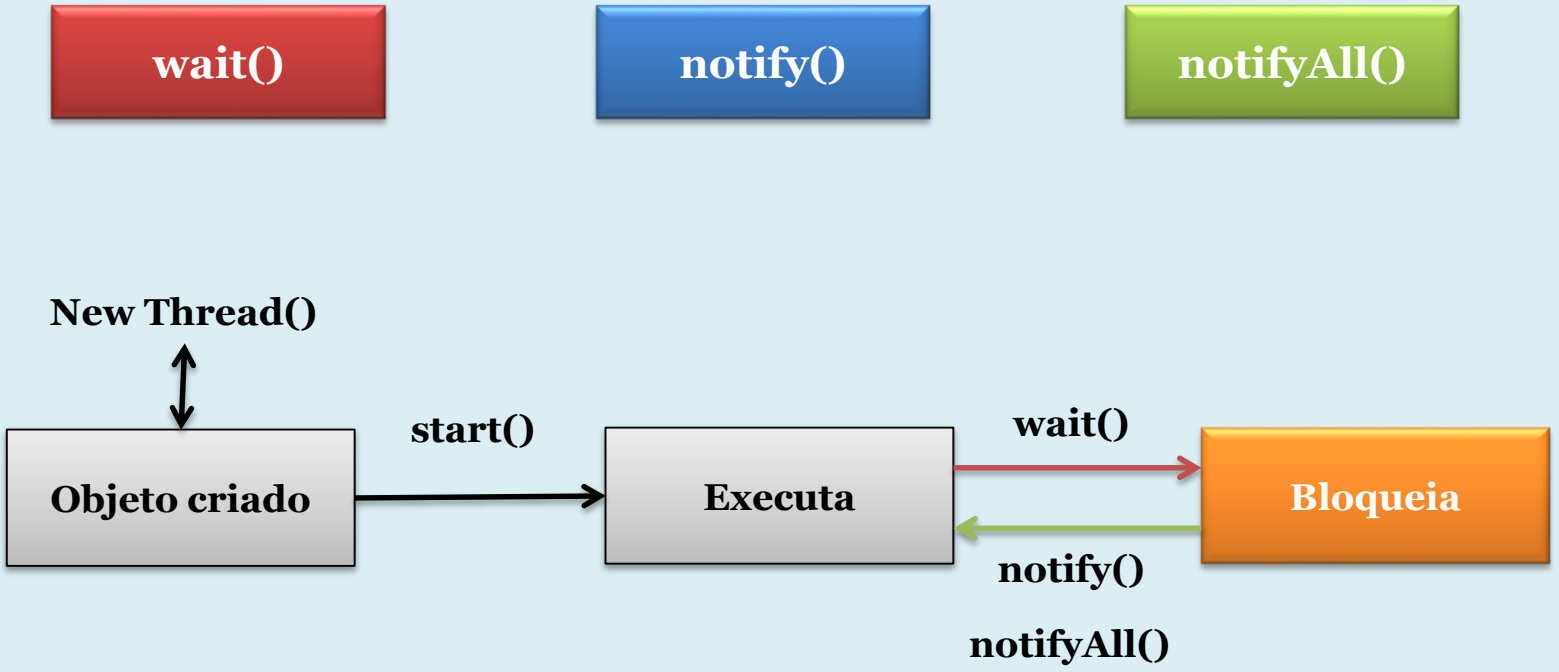
métodos sem o modificador **synchronized** não exigem o lock do objeto para acesso.

Objeto

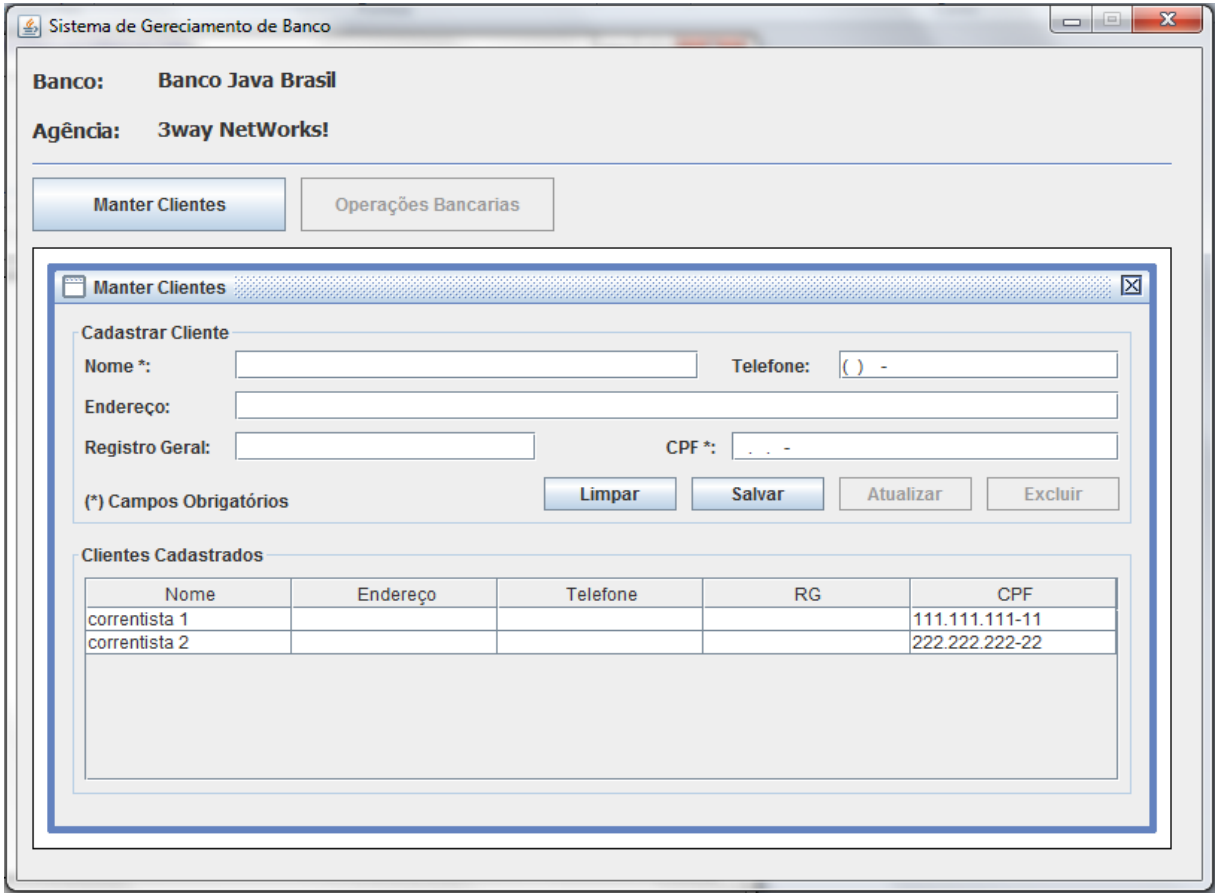
Cada objeto tem um lock de acesso.

Bloqueando acesso Concorrente

A coordenação é feita com métodos da classe Object:



Java Orientado a Objetos
Construindo interfaces gráficas com Swing



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

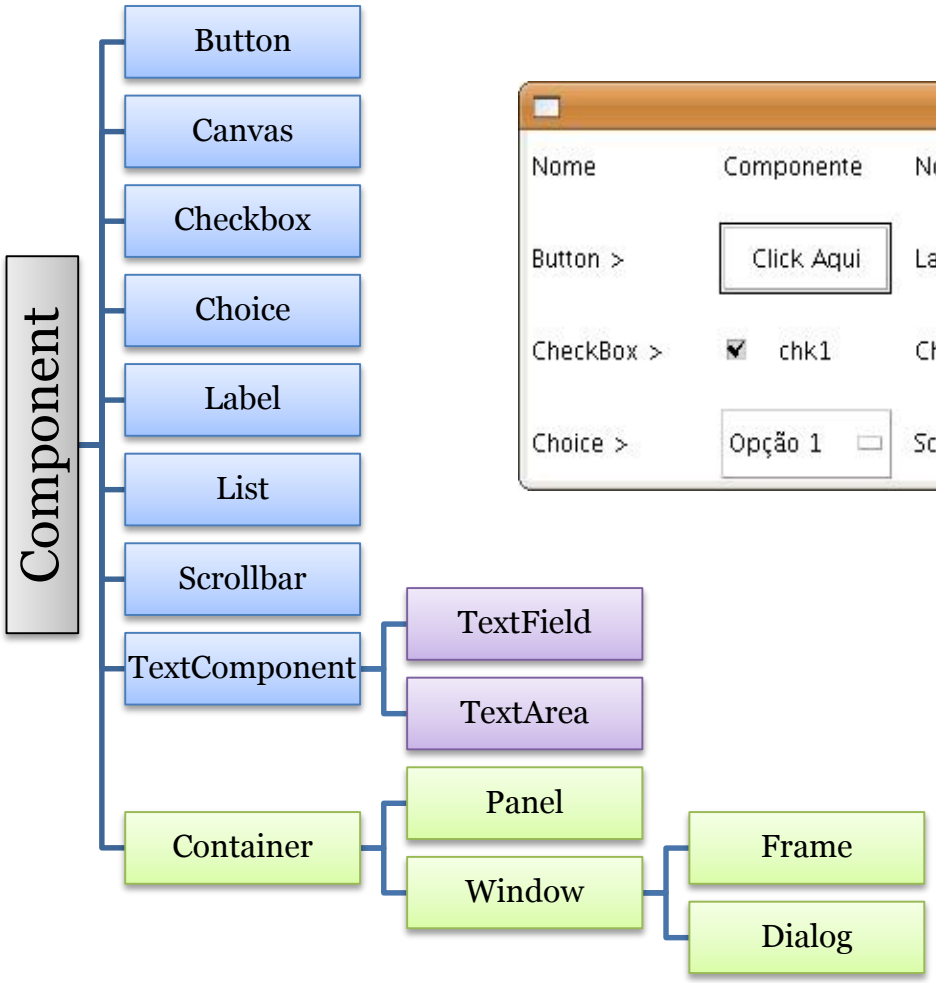
Swing

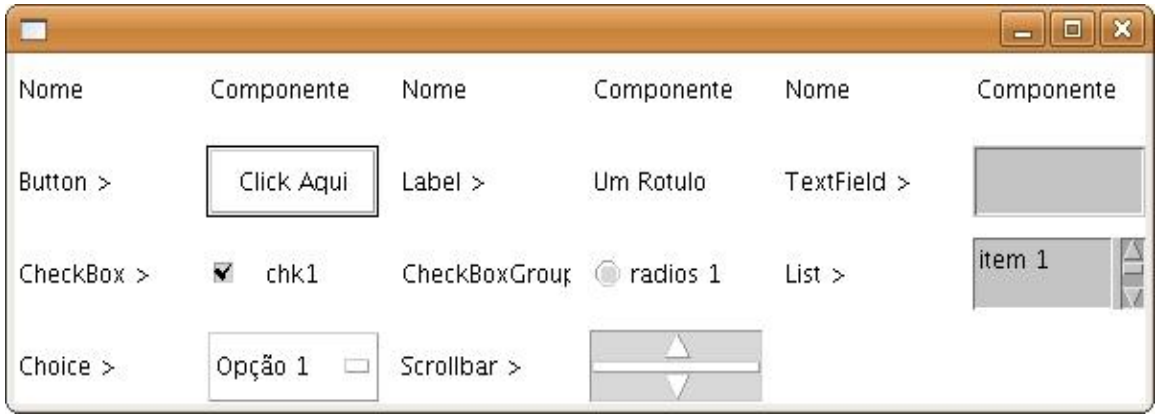
Escrito em Java

Independente de plataforma

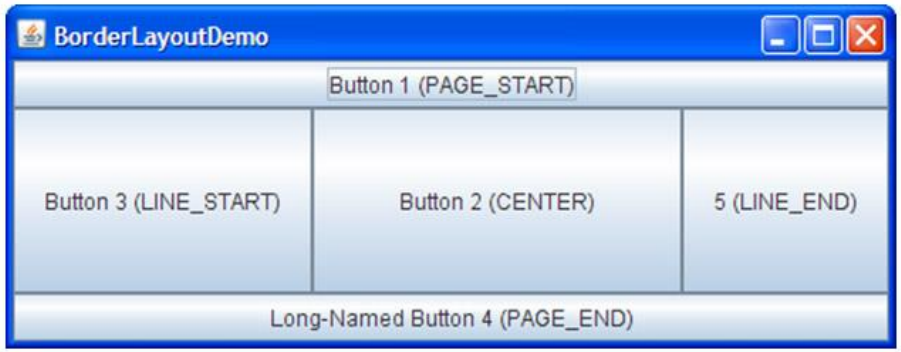
Mesma aparência em plataformas diferentes

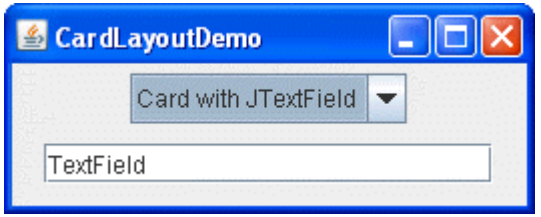
Componentes AWT

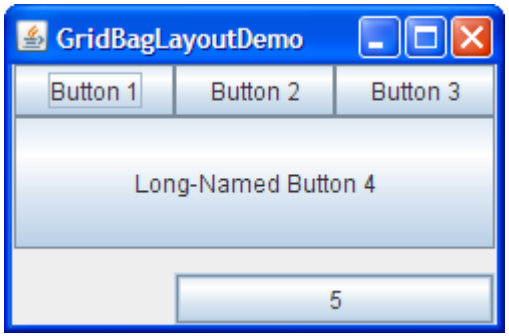


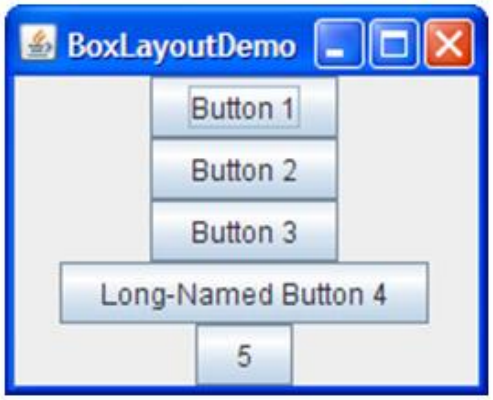


Gerenciadores de layout

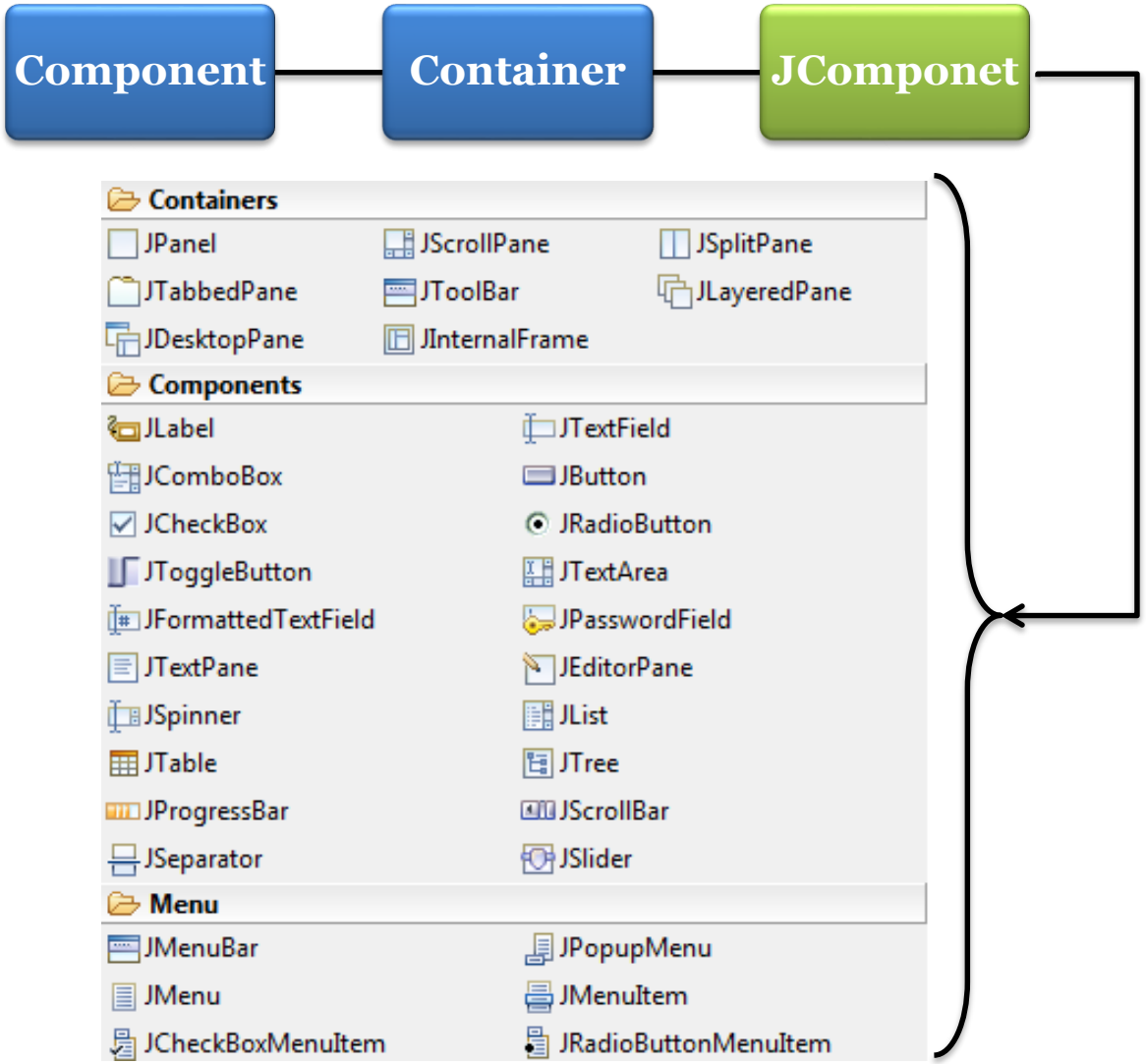








Componentes Swing



Containers JFrame

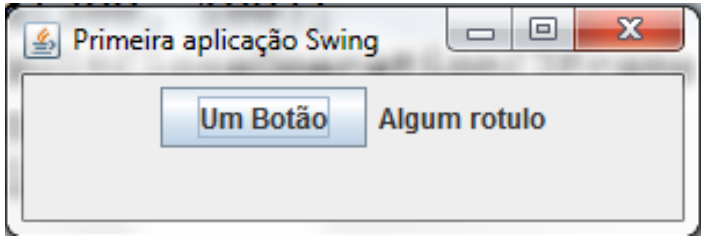
```
public class AppSwing extends JFrame {
    JButton botao;
    JLabel label;
    public AppSwing() {
        super("Primeira aplicação Swing");
        setSize(300, 100);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLayout(new FlowLayout());
        initialize();
    }
    private 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

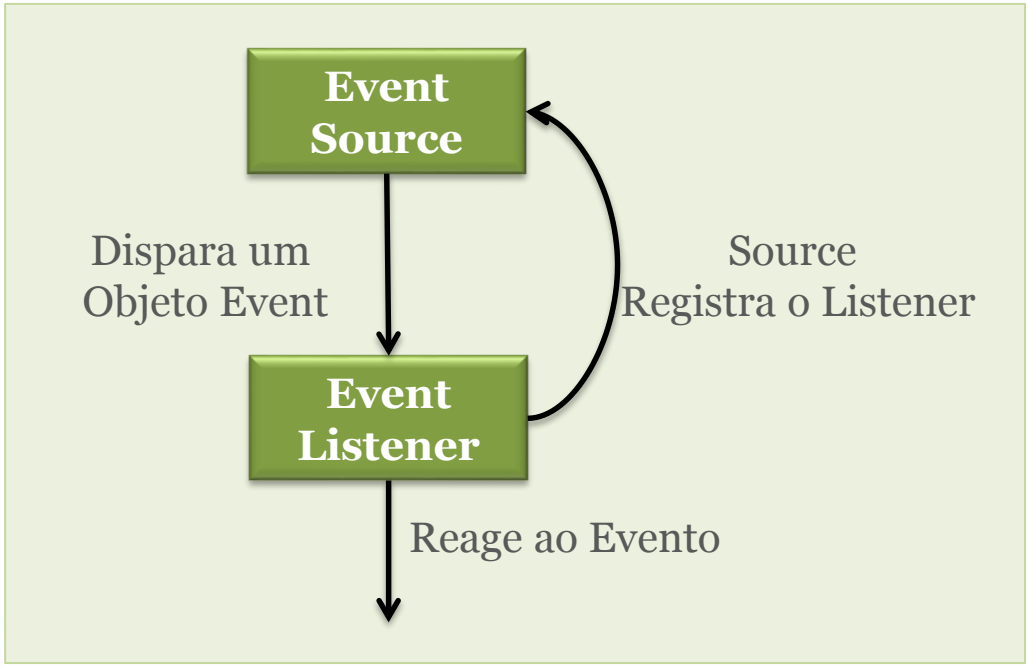
Construtor

Container que recebe os componentes

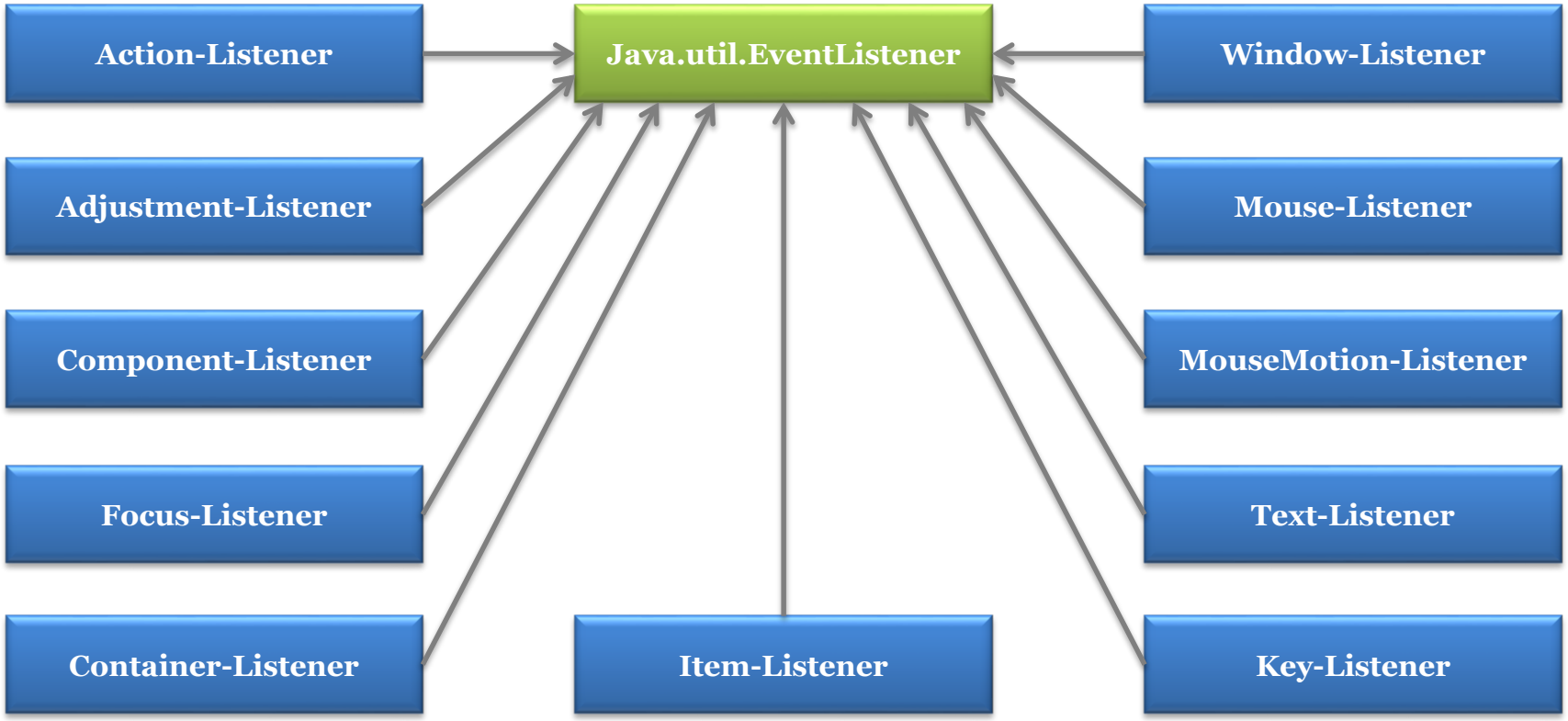
Inicia o frame e o deixa visível



Manipulação de Evento



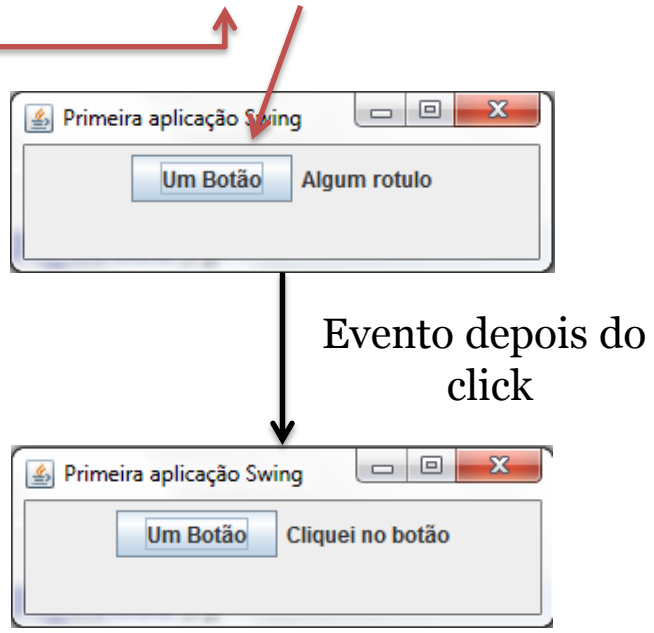
Classes de Evento



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



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)
}
```