

Programação de Computadores II

- Linguagem Java -

Alexandre Sztajnberg

Roteiro

- ☐ Introdução
- ☐ Noções de Orientação a Objeto
- ☐ Primeiros exemplos
- ☐ Sintaxe e estruturas
- ☐ Métodos, controle
- ☐ Exceções
- ☐ Threads
- ☐ Serialização
- ☐ Reflexão
- ☐ RMI

Slides baseados em...

- ❑ David J. Barnes & Michael Kölling, Programação orientada a objetos com Java, Pearson Education do Brasil, 2004
- ❑ Introduction to Programming, David J. Barnes Michael Kölling
- ❑ The University of North Carolina at Chapel Hill, Programming Language Concepts, Spring 2002, Felix Hernandez-Campos
- ❑ Nell Dale, Chip Weems and Mark Headington, [Programming and Problem Solving with Java](#), Editora Jones and Bartlett Publishers.
- ❑ Rafael Santos Introdução à Programação Orientada a Objetos Usando Java
- ❑ [Cay Horstmann](#), Big Java: Programming and Practice, Ed. Wiley, 2002.

Exercícios, Projetos, Provas

- ❑ Programas pequenos (muitos!)
 - em dupla (“pair programming”)
- ❑ Projetos (programas um pouco maiores)
 - um ou dois
- ❑ Provas: P1 e P2
- ❑ Obs1: tudo vale nota!
- ❑ Obs2: plágio (cola) - 0,0 (zero) para TODOS envolvidos

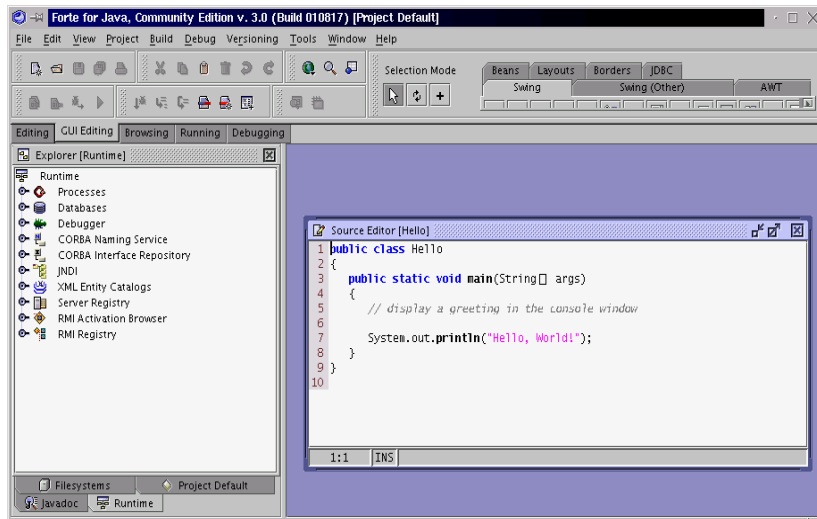
Ferramentas

- ❑ JDK
- ❑ Editores
- ❑ IDEs
- ❑ No LabIME
 - JCreator
 - JDK 1.4
 - Eclipse?
- ❑ www.ime.uerj.br/~alexsz/cursos/pc2

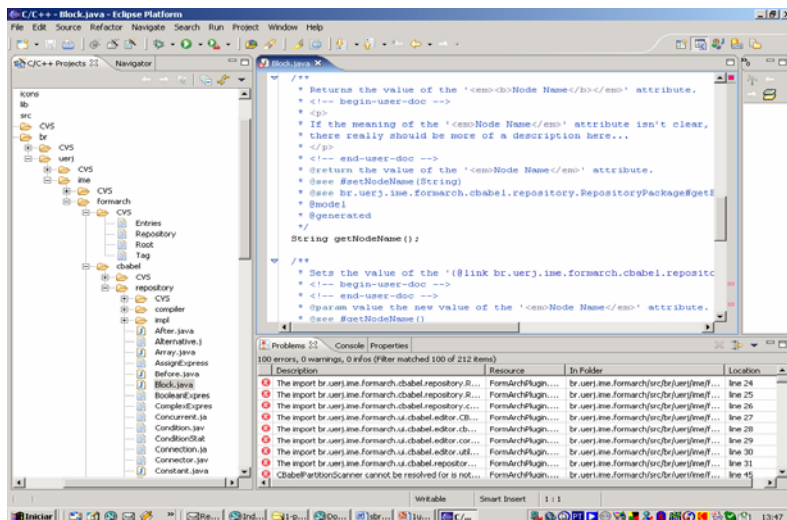
Ambientes de Programação

- ❑ Além de compiladores e interpretadores
 - Assemblers, debuggers, pré-processadores e ligadores (linkers)
 - Editores
 - Style Checkers
 - Version management
 - Profilers
- ❑ Integrated environments
 - Beyond a simple *bus error*
 - *Emacs*

IDE: Integrated Development Environment



Eclipse



Compilação x Interpretação

Do Programa-fonte para o Código Executável

```
program gcd(input, output);  
var i, j: integer;  
begin  
    read(i, j);  
    while i <> j do  
        if i > j then i := i - j;  
        else j := j - i;  
    writeln(i)  
end.
```

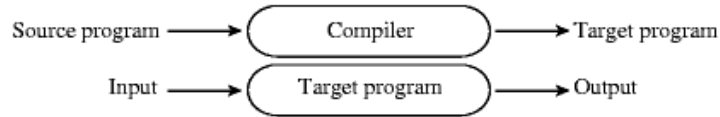


Compilação

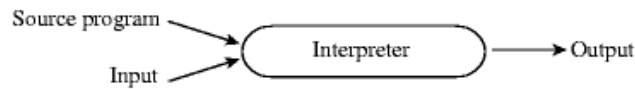
```
27bdffd0 afbf0014 0c1002a8 00000000 0c1002a8 afa2001c 8fa4001c  
00401825 10820008 0064082a 10200003 00000000 10000002 00832023  
00641823 1483fffa 0064082a 0c1002b2 00000000 8fbf0014 27bd0020  
03e00008 00001025
```

Compilação e Interpretação

- O **compilador** é o **programa** que traduz o código em alto nível para o programa executável



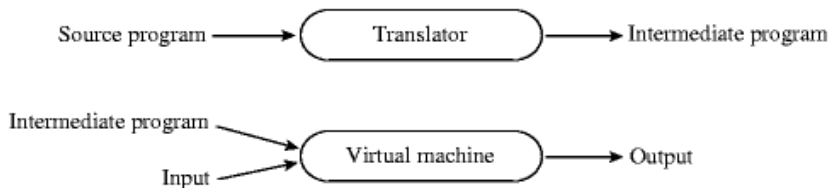
- O **interpretador** é o programa que executa outro programa



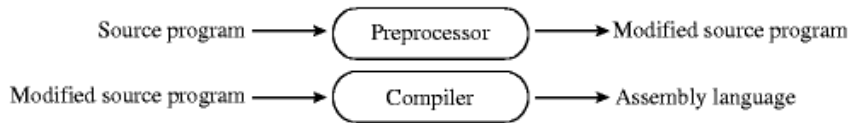
Combinando Compilação e Interpretação

- Diferença sutil:

- A linguagem é interpretada quando a tradução inicial é simples
- A linguagem é compilada quando o processo de tradução é complexo



Pré-processamento

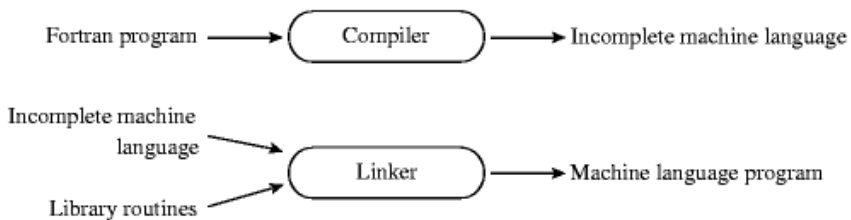


□ Macros

- `#define <macro> <replacement name>`
- `#define FALSE 0`
- `#define max(A,B) ((A) > (B) ? (A):(B))`

Ligação (Linking)

□ Bibliotecas e subrotinas

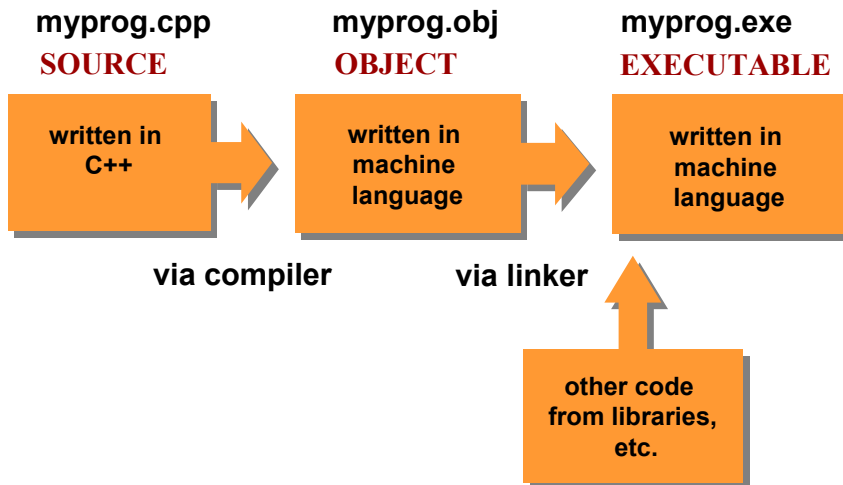


- Java - Características e Portabilidade (JVM)

Linguagens de Alto Nível

- ▣ Are **portable**
- ▣ Are translated into machine code by **compilers**
- ▣ Instructions are written in language similar to natural language
- ▣ Examples -- FORTRAN, COBOL, Pascal, C, C++
- ▣ Many are standardized by ISO/ANSI to provide an official description of the language

Estágios de um Programa em C++



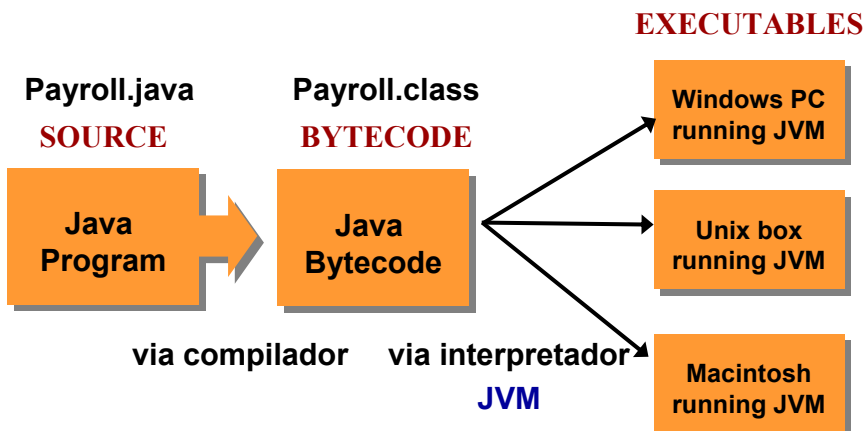
Linguagem Java

- ❑ É uma linguagem orientada a objetos.
- ❑ "Write once, run anywhere" . A linguagem Java é totalmente portátil. O programa é compilado em "bytecodes", e a *Java Virtual Machine* (JVM) interpreta o código (bytecode) para instruções da plataforma sobre a qual a JVM foi instalada.
- ❑ Ideal para aplicações de Rede e Internet (*Applets*).
- ❑ Possui bibliotecas de apoio.
- ❑ Segurança. Como restrições nas applets.
- ❑ Sem uso de ponteiros.
- ❑ Sintaxe semelhante a C / C++.
- ❑ Case sensitive.

Linguagem Java

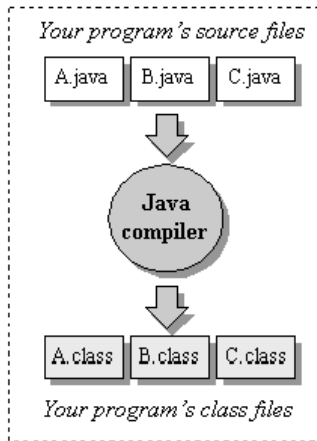
- Achieves portability by using both a **compiler** and an **interpreter**
- Java compiler translates a Java program into an intermediate **Bytecode**--not machine language
- An interpreter program called the Java Virtual Machine (JVM) translates each successive instruction in the Bytecode program to machine language and immediately runs it

Portabilidade de Java

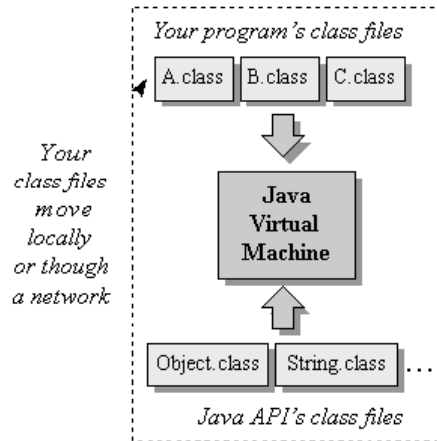


Ambiente de Programação Java

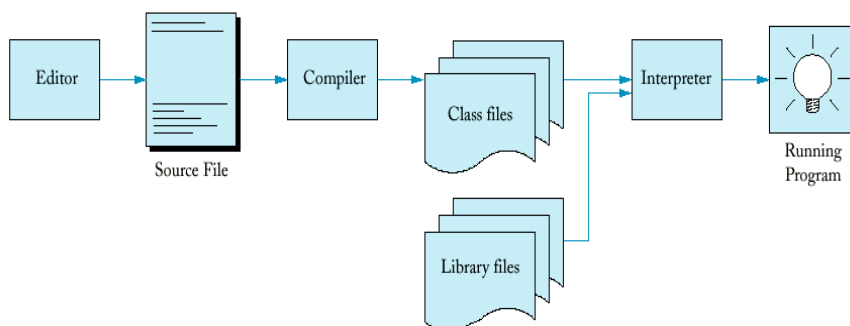
compile-time environment



run-time environment



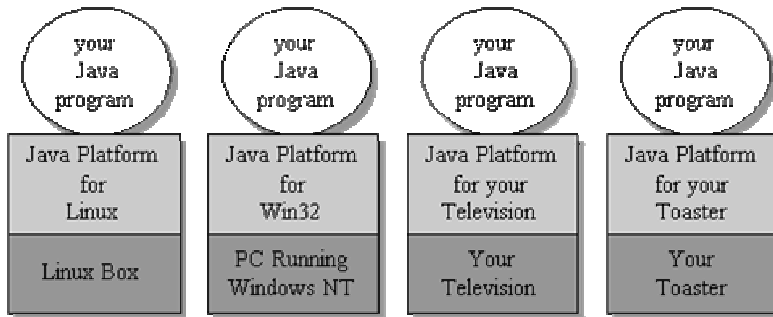
A sequência de passos para desenvolver um programa em Java



A plataforma de Java

The byte code generated by the Java front-end is an *intermediate form*

- ❑ Compact
- ❑ Platform-independent



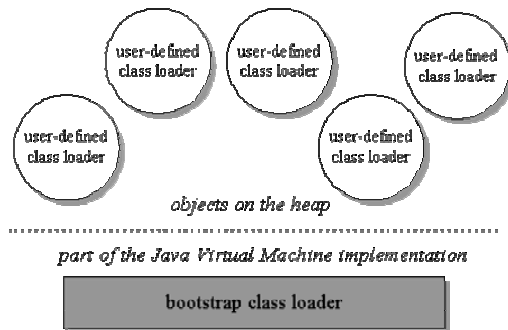
O arquivo *.class* File

Java class file contains

- ❑ Byte code for data and methods (intermediate form, platform independent) (*remember byte code?*)
- ❑ *Symbolic* references from one class file to another
 - Class names in text strings
 - Decompiling/reverse engineering quite easy
- ❑ Field names and descriptors (type info)
- ❑ Method names and descriptors (num args, arg types)
- ❑ Symbolic refs to other class methods/fields, own methods/fields

Class Loaders

- ❑ Bootstrap (default) loader (na JVM)
- ❑ Carregadores definidos pelo usuário (customizado)



Carga dinâmica de classes

- ❑ You don't have to know at compile-time all the classes that may ultimately take part in a running Java application.

User-defined class loaders enable you to dynamically extend a Java app at run-time

- ❑ As it runs, your app can determine what extra classes it needs and load them
- ❑ Custom loaders can download classes across a network (applets), get them out of some kind of database, or even calculate them on the fly.

A máquina de execução

❑ *JVM Simples*

- Interpretação do Byte Code

❑ *Just-in-time compiler*

- Métodos em *ByteCode* são compilados em código de máquina na primeira vez que são invocados
- O código de máquina é guardado para uma invocação subsequente
- Isto requer mais memória

❑ *Otimização adaptativa*

- O interpretador monitora a atividade do programa, compilando a parte do código mais usada do programa em código de máquina
- Isto é muito mais rápido do que uma simples interpretação, **requer um pouco mais de memória**
- A exigência da memória é somente ligeiramente maior devido à regra de 20%/80% da execução de programa **(no geral, 20% do código é responsável por 80% da execução)**

– Alô! –
Primeiro programa

Primeiro programa: Alo.java

```
1 public class Alo
2 {
3     public static void main(String[] args)
4     {
5         // Mostra uma frase no console
6         System.out.println(" Alô,Mundo!");
7         /* Comentario
8            de mais de uma linha */
9     }
10 }
```

O que temos no programa?

- ❑ public class *ClassName*
- ❑ public static void main(String[] args)
- ❑ // comentário de uma linha e (*/* */*) de várias linhas
- ❑ Chamada de método
objeto.metodoName(parametros)
- ❑ Classe *System*
- ❑ Objeto *System.out*
- ❑ Método *println*

Chamada (invocação) de método

□ `objeto.metodoName(parametros)`

□ **Exemplo:**

○ `System.out.println("Hello, Dave!");`

□ **Objetivo:**

○ Invocar um método de um objeto e fornecer qualquer parâmetro adicional.

Compilando e executando o exemplo

- Escrever o programa em um editor de textos
- Salvá-lo com o nomeClass, e extensão .java
- Abrir o prompt (no windows o DOS)
- Compilar em byte codes
`javac Alo.java`
- Executar byte codes
`java Alo`

Erros

❑ Erros de sintaxe

```
System.ouch.print("...");  
System.out.print("Hello");
```

- Detectados pelo compilador

❑ Erros lógicos

```
System.out.print("Hell");
```

- Detectados (assim espero) na fase de teste

Breve introdução a
Orientação a Objetos

Programação Orientada a Objetos (POO)

- ❑ **Objeto:** Objeto de software modela objetos do mundo real, e possui estado (variáveis) e ações (métodos).
- ❑ **Métodos:** São as ações que um objeto pode praticar, e é o modo pelo qual os objetos se comunicam entre si.
- ❑ **Classe:** É o modelo que descreve as variáveis e métodos comuns a todos os objetos de certo tipo.
- ❑ **Herança:** Ocorre quando uma classe "herda" o características e ações de uma superclasse.
- ❑ **Interface:** É como um contrato, onde uma classe que implementa a interface, se compromete a implementar os métodos declarados na interface.

Objetos e classes

- ❑ **Objetos**
 - Representam 'coisas' do mundo real ou do domínio de algum problema (exemplo: "o carro vermelho ali no estacionamento").
- ❑ **Classes**
 - Representam todos os tipos de objetos (exemplo: "carro").

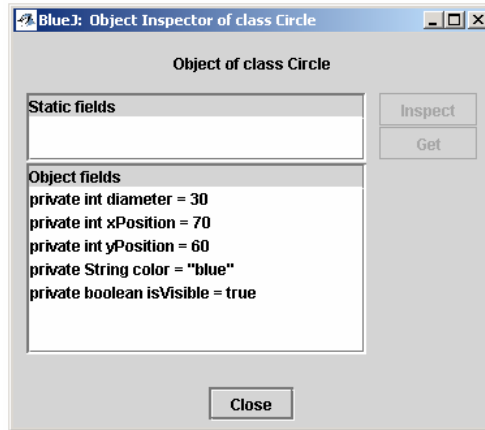
Métodos e parâmetros

- ❑ Objetos têm operações que podem ser invocadas (o Java as chama de *métodos*).
- ❑ Métodos podem ter parâmetros para passar informações adicionais necessárias para sua execução.

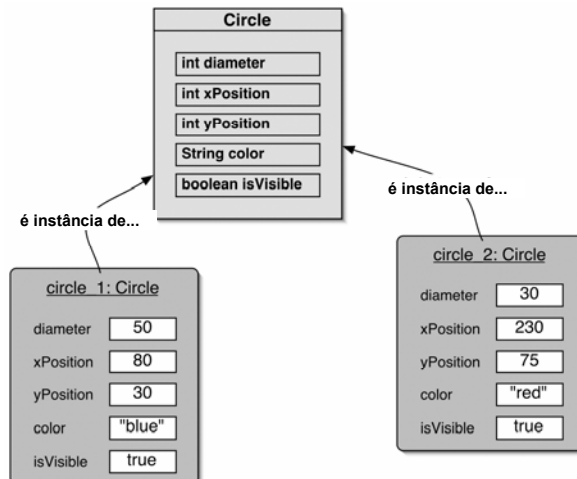
Outras observações

- ❑ Várias *instâncias* podem ser criadas a partir de uma única classe.
- ❑ Um objeto tem *atributos*: valores armazenados em *campos*.
- ❑ A classe define quais campos um objeto tem, mas todo objeto armazena seu próprio conjunto de valores (o *estado* do objeto).

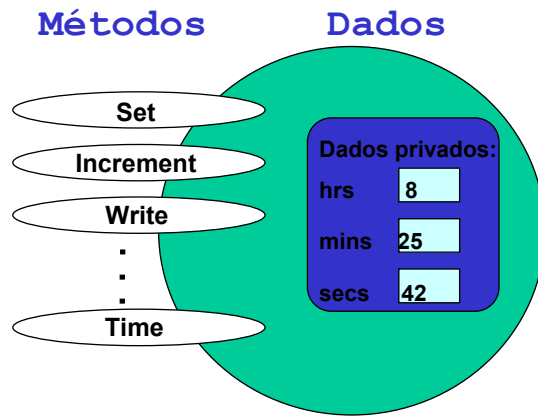
Estado



Dois objetos circle



Um objeto da classe Time



Características da POO

- ☐ Encapsulamento
- ☐ Abstração dos Dados
- ☐ Esconder informações
- ☐ Reutilização do Código
- ☐ Polimorfismo
- ☐ Dynamic method binding

Encapsulamento

- ❑ Permite ao programador esconder os detalhes da representação dos dados por trás de um simples conjunto de operações (como a interface).
- ❑ Quais os benefícios da abstração de dados?
 - Reduz a carga conceitual (ter que conhecer as “internas”)
 - Programadores precisam conhecer menos sobre o resto do programa
 - Encapsula falhas (dentro da classe/objeto)
 - Bugs (erros) são localizados em componentes independentes
 - Fornece um grau significativo de independência de componentes do programa
 - Separa os diferentes tipos de programadores

Encapsulamento, Classes, Objetos e Métodos

- ❑ A unidade de encapsulamento na OOP é a classe, um tipo abstrato de dados
- ❑ O conjunto de valores é o conjunto de objetos (ou instâncias)
- ❑ Classes podem ter:
 - Conjunto de atributos de classes
 - Conjunto de métodos de classe
- ❑ Objetos podem ter:
 - Conjunto de atributos de instâncias (que se relacionam)
 - Conjunto de métodos de instâncias

Herança

- ❑ O encapsulamento melhora a reusabilidade do código
 - Tipos de dados abstratos
 - Módulos
 - Classes
- ❑ É geralmente o caso em que o código que o programador quer reusar, é próximo mas não é exatamente aquilo que ele necessita.
- ❑ Herança fornece um mecanismo para estender ou refinar unidades de encapsulamento
 - Adicionando ou sobrescrevendo métodos
 - Adicionando atributos

Polimorfismo

- ❑ The is-a relationship supports the development of generic operations that can be applied to objects of a class and all its subclasses
 - This feature is known as polymorphism
 - E.g. paint() method
- ❑ The binding of messages to method definition is instance-dependent, and it is known as dynamic binding
 - It has to be resolved at run-time
- ❑ Dynamic binding requires the virtual keyword in C++
- ❑ Static binding requires the final keyword in Java

Java: Sintaxe, Classes, Objetos, etc.

Java

- ❑ Nomes de variáveis, objetos, constantes, métodos, etc.
- ❑ O identificador em Java deve começar com uma letra ou underscore ou \$, e deve ser seguido por zero ou mais letras (A-Z, a-z), dígitos (0-9), underscores (_), ou \$.

Válidos:

age_of_dog	taxRateY2K	HourlyEmployee
ageOfDog		

Inválidos: (Por que?)

age#	2000TaxRate	Age-Of-Dog
------	-------------	------------

Outros aspectos dos identificadores

- Java é uma linguagem **case-sensitive**, ou seja, ela diferencia letras maiúsculas e minúsculas. Ex: nome, Nome, nOMe são distintos.
- O uso de identificadores **significativos** é uma boa prática de programação.

31 Palavras Reservadas em Java

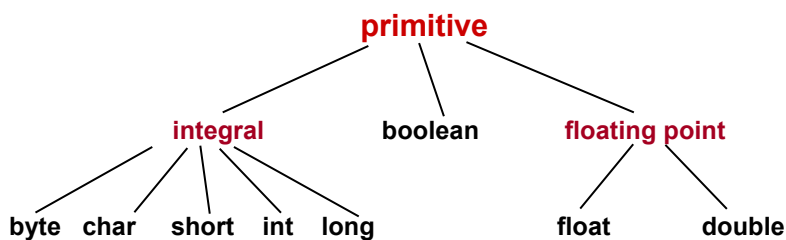
abstract	boolean	break	byte	case
catch	char	class	const	continue
default	do	double	else	extends
false	final	finally	float	for
goto	if	implements	import	instanceof
int	interface	long	native	new
null	package	private	protected	public
return	short	static	strictfp	super
switch	synchronized	this	throw	throws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

Não podem ser usadas como identificadores.

Java tem tipos de dado “primitivos”

- **Tipos Inteiros**
 - Podem representar números positivos e negativos quando declarados como **byte**, **int**, **short**, ou **long**
 - Podem representar caracteres simples quando declarados como **char**
- **Tipos de ponto-flutuante**
 - Representam números reais com ponto decimal
São declarados como **float**, ou **double**
- **Tipo Booleano**
 - Recebe o valor verdadeiro (true) ou falso (false)
Declarado como **boolean**.

Tipos de dados primitivos em Java



Exemplos de valores de dados em Java

int sample values

4578 -4578 0

double sample values

95.274 95. .265

float sample values

7.4f 95.2f -> Obs: *f* no final do n° para
indicar *float*.

char sample values

'B' 'd' '4' '?' '*'

ASCII e Unicode

- ❑ **ASCII** é um antigo conjunto de caracteres usados para representar caracteres como inteiros.
- ❑ ASCII é um subconjunto do novo conjunto de caracteres **Unicode**.
- ❑ O caractere 'A' em Unicode é internamente armazenado como inteiro 65, e as letras sucessivas do alfabeto são armazenadas como inteiros consecutivos.
- ❑ É válida a comparação de caracteres: 'A' menor que 'B', etc.

Tipos de Inteiros

Tipo	Tam. em Bits	Valor Mínimo	até	Valor Máximo
byte	8	-128	até	127
short	16	-32,768	até	32,767
int	32	-2,147,483,648	até	2,147,483,647
long	64	-9,223,372,036,854,775,808	até	+9,223,372,036,854,775,807

Tipos de Ponto-flutuante

Tipo	Tam. em Bits	Alcance dos Valores
float	32	$\pm 1.4 \times 10^{-45}$ até $\pm 3.4028235 \times 10^{38}$
double	64	$\pm 4.9 \times 10^{-324}$ até $\pm 1.7976931348623157 \times 10^{308}$

Java String Class

- ❑ A **string** is a sequence of characters enclosed in **double quotes**.

- ❑ **string** sample values

`"Today and tomorrow"`

`"His age is 23."`

`"A"` (a one character string)

- ❑ The empty string contains no characters and is written as `""`

A Classe String

Ao contrário de outras linguagens de programação, em Java *String* não é um Array de caracteres mas uma classe.

Porém sua declaração é semelhante aos tipos de dados primitivos:

`String nome = "Sinforosa";` ou

`String nome = new String("Sinforosa");`

Como classe, uma string possui métodos, exemplo:

- `length()` – Retorna o nº de caracteres da String
- `charAt(int i)` – Retorna o caractere da posição i da String.

Métodos de uma String

- O método *length* retorna um valor int que é o número de caracteres da String.

```
String nome = "Donald Duck";
```

```
int numChars;
```

```
numChars = nome.length();
```

instância

método

length é um método da instância

Métodos de uma String (Cont.)

- ❑ Método `indexOf` procura na **string** uma **substring** particular, e retorna um valor int que é a posição inicial para a primeira ocorrência da substring dentro da string.
- ❑ Obs: Caracteres começam na posição **0 (zero)**, não 1.
- ❑ O argumento **substring** pode ser uma String literal, uma expressão String, ou um valor do tipo char.
- ❑ Se a **substring** não for encontrada, o método `indexOf` retorna **-1**.

Métodos de uma String (Cont.)

- ❑ Método **substring** retorna a substring da string, porém não altera a string.
- ❑ O primeiro parâmetro é um int que especifica a posição inicial dentro da string
- ❑ O segundo parâmetro é um int que é 1 unidade maior que a posição final da substring
- ❑ **Lembre-se:** posições de caracteres contidas na string são numeradas **iniciando-se a partir do 0**, não de 1.

Actions of Java's String class

- ❑ **String operations include**
 - joining one string to another (concatenation)
 - converting number values to strings
 - converting strings to number values
 - comparing 2 strings

Declaração de tipos

Dentro de qualquer parte do código a declaração é feita da seguinte forma:

```
int a=1,b,c; //podem ser ou não inicializadas
```

```
char ch1,ch2='b'; //variáveis separadas por vírgula
```

```
String nome1, nome2=null; /*Em ambos a referência  
é nula */
```

```
String frase1 = "Hoje acordei";
```

```
String frase2 = new("Hoje acordei"); /* as duas  
declarações possuem o mesmo efeito */
```

Simples classe em Java

Cabeçalho

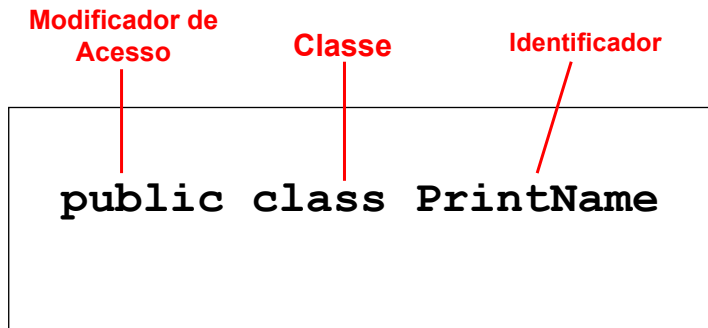
```
class FazNada
```

```
{
```

Corpo

```
}
```

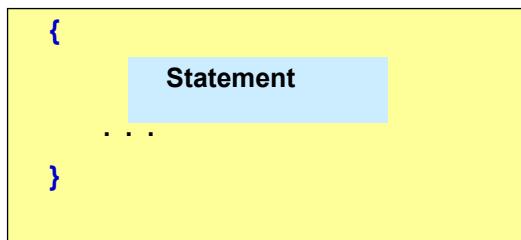

Cabeçalho da classe



Bloco (Compound Statement)

- ❑ Bloco é a seqüência de zero ou mais instruções envolvidas por um par de chaves { }.

Bloco



What is a Variable?

- ❑ A **variable** is a location in memory to which we can refer by an identifier, and in which a **data value** that can be **changed** is stored
- ❑ Declaring a variable means specifying both its name and its data type or class

What Does a Variable Declaration Do?

```
int ageOfDog;
```

A declaration tells the compiler to **allocate enough memory** to hold a value of this data type, and to **associate the identifier** with this location.



4 bytes for ageOfDog

Syntax for Declarations

Variable Declaration

```
Modifiers TypeName Identifier , Identifier . . . ;
```

Constant Declaration

```
Modifiers final TypeName Identifier = LiteralValue;
```

What is a Named Constant?

- ❑ A **named constant** is a location in memory to which we can refer by an identifier, and in which a **data value that cannot be changed** is stored.

VALID NAMED CONSTANT DECLARATIONS

```
final String STARS = "*****";  
  
final float  NORMAL_TEMP = 98.6;  
final char   BLANK = ' '  
  
final int    VOTING_AGE = 18;  
final double MAX_HOURS = 40.0;
```

Giving a value to a variable

You can assign (give) a value to a variable by using the **assignment operator =**

VARIABLE DECLARATIONS

```
String firstName;  
char   middleInitial;  
char   letter;  
int    ageOfDog;
```

VALID ASSIGNMENT STATEMENTS

```
firstName = "Fido";  
middleInitial = 'X';  
letter = middleInitial;  
ageOfDog = 12;
```

❑ *Why is String uppercase and char lower case?*

- char is a built in type
- String is a class that is provided
- Class names begin with uppercase by convention

Assignment Statement Syntax

```
Variable = Expression;
```

First, Expression on right is evaluated.

Then the resulting value is stored in the memory location of Variable on left.

NOTE: The value assigned to Variable must be of the same type as Variable.

String concatenation (+)

- ❑ Concatenation uses the + operator.
- ❑ A built-in type value can be concatenated with a string because Java automatically converts the built-in type value for you to a string first.

Concatenation Example

```
final int DATE = 2003;

final String phrase1 = "Programming and Problem ";
final String phrase2 = "Solving in Java ";
String bookTitle;

bookTitle = phrase1 + phrase2;

System.out.println(bookTitle + " has copyright " + DATE);
```

Usando os dispositivos de saída

SINTAXE NA CHAMADA DO MÉTODO

```
System.out.print (StringValue);  
System.out.println (StringValue);
```

Esses exemplos fornecem a mesma saída.

- 1) `System.out.print("The answer is, ");`
`System.out.println("Yes and No.");`
- 2) `System.out.println("The answer is, Yes and No.");`

Entrada: argumentos da linha de comando

A chave para os argumentos da linha de comando está no método main:

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

String[] args , significa um array de Strings que são uma lista de argumentos da linha de comandos fornecida pelo usuário na chamada do programa.

Como um exemplo veremos uma aplicação que recebe 2 números reais para o cálculo da área de um retângulo.

Exemplo de passagem de parâmetros

```
public class Area
{
    public static void main(String[] args)
    {
        //verifica se foi passado 2 argumentos
        if(args.length==2)
        {
            double a=Double.parseDouble(args[0]);
            double b=Double.parseDouble(args[1]);
            double area = a * b;
            System.out.println("Area = " +area);
        }
    }
}
```

Na chamada do programa na linha de comandos:

```
>java Area 2.5 5
>Area = 12.5
```

Dispositivo de Entrada

- ❑ Mais complexo que os dispositivos de saída
- ❑ Deve se ajustar mais de um dispositivo primitivo

```
InputStreamReader inStream;
```

```
inStream = new InputStreamReader(System.in);
```

```
// declare device inData
```

```
BufferedReader inData;
```

```
inData = new BufferedReader(inStream)
```

Usando o dispositivo de entrada

```
//Criando o dispositivo em uma instrução
```

```
inData = new BuffredReader(new  
    InputStreamReader(System.in));
```

```
String umaLinha;
```

```
// Armazena uma linha do texto em umaLinha
```

```
umaLinha = inData.readLine();
```

Da onde o texto vem?

Interactive Input

- ❑ `readLine` é o valor retornado na classe `BufferedReader`
- ❑ `readLine` vai na janela do `System.in` e recebe como entrada o que o usuário digitou

Como o usuário sabe o que digitar?

- ❑ O programa (você) informa o usuário
`System.out.print("informe...");`

Interactive Input (cont.)

```
BufferedReader inData;  
inData = new BufferedReader(new InputStreamReader(System.in));  
String nome;  
System.out.print("Digite seu nome: ");  
nome = inData.readLine();
```

Nome contém o que o usuário digitou em resposta no prompt.

Programa exemplo, calcula a área de um retângulo.

```
import java.io.*;
public class Area2
{
    public static void main(String[] args) throws
IOException
    {
        BufferedReader inData;
        inData = new BufferedReader(new
InputStreamReader(System.in));
        String aux;
        double a,b,area;

        System.out.print("Digite um lado: ");
        aux = inData.readLine();
        a = Double.parseDouble(aux);

        System.out.print("Digite outro lado: ");
        aux = inData.readLine();
        b = Double.parseDouble(aux);
```

```
        // continuacao
        area = a * b;
        System.out.println("Area = " +area);
    }
}
```

Na chamada do programa na linha de comandos:

```
>java Area2
>Digite um lado: 2.5
>Digite outro lado: 5
>Area = 12.5
```

A Java Application

- ❑ Must contain a method called `main()`
- ❑ Execution always begins with the first statement in method `main()`
- ❑ Any other methods in your program are subprograms and are not executed until they are sent a message

Java Program

```
// *****  
// PrintName prints a name in two different formats  
// *****  
public class PrintName  
{  
    public static void main (String[ ] args)  
    {  
        BufferedReader inData;  
        String first;           // Person's first name  
        String last;           // Person's last name  
        String firstLast;      // Name in first-last format  
        String lastFirst;      // Name in last-first format  
        inData = new BufferedReader(new  
            InputStreamReader(System.in));
```

Java program continued

```
System.out.print("Enter first name: ");
first = inData.readLine();

System.out.print("Enter last name: ");
last = inData.readLine();

firstLast = first + " " + last;
System.out.println("Name in first-last format is "
                  + firstLast);
lastFirst = last + ", " + first;
System.out.println("Name in last-first format is "
                  + lastFirst);

    }
}
```


Sintaxe da declaração de métodos

Declaração de método

```
Modifiers void Identifier (ParameterList)
{
    Statement
}
```

Statement Syntax Template

Statement



- NullStatement
- LocalConstantDeclaration
- LocalVariableDeclaration
- AssignmentStatement
- MethodCall
- Block

NOTE: This is a partial list.

One Form of Java Comments

- Comments between `/*` and `*/` can extend over several lines.

`/* This is a Java comment. It can extend over more
than one line. */`

`/* In this second Java comment the asterisk on the next line
* is part of the comment itself.
*/`

Another Form of Java Comment

- ❑ Using two slashes // makes the rest of the line become a comment.

```
// *****  
// PrintName prints a name in two different formats  
// *****  
  
String first;    // Person's first name  
String last;    // Person's middle initial
```

Operadores

A linguagem Java fornece um conjunto amplo de operadores, sendo eles divididos nos grupos de :

- ❑ Operadores Aritméticos
- ❑ Operadores Booleanos ou Lógicos – retornam os valores **true** ou **false**.
- ❑ Operadores de Atribuição
- ❑ Operador de String

Operadores Aritméticos

Operador	Significado	Exemplo
*	Multiplicação	A * B
/	Divisão	A / B
+	Adição	A + B
-	Subtração	A - B
%	Resto da divisão inteira	A % B
-	Sinal negativo (- unário)	-A
+	Sinal positivo (+ unário)	+A
++	Incremento unitário	++A ou A++
--	Decremento unitário	--A ou A--

Operadores Aritméticos (cont.)

Diferença entre operadores sufixo e prefixo:

```
int w=7,x=18,y,z;  
y = w + x--; // operador sufixo: x--  
z = w + --x; // operador prefixo: --x
```

Na 1ª expressão o valor resultante de y é $7+18 = 25$, depois x é diminuído de 1, pois x é diminuído com sufixo.

Na 2ª expressão o valor resultante de z é $7+16 = 23$, pois x que foi diminuído na expressão anterior é diminuído novamente com prefixo.

Operadores Booleanos ou Lógicos

Operador	Significado	Exemplo
==	Igual	A == B
!=	Diferente	A != B
>	Maior que	A > B
>=	Maior ou Igual a	A >= B
<	Menor que	A < B
<=	Menor ou igual a	A <= B
&&	E Lógico (AND)	A && B
	Ou Lógico (OR)	A B
!	Negação (NOT)	!A

Operadores de Atribuição

Atribuição permite definir o valor de uma variável através de uma constante, ou do resultado de uma expressão.

Em JAVA é válida a atribuição do tipo: a=b=c=d=5;

Operador	Significado	Exemplo
=	Atribuição Simples	A = B
+=	Adição	A+=B A=A+B
-=	Subtração	A-=B A=A-B
=	Multiplicação	A=B A=A*B
/=	Divisão	A/=B A=A/B
%=	Resto da divisão inteira	A%=B A=A%B

Operador de String

O Operador de string é o + que resulta na concatenação de Strings. Exemplo:

```
String frase,nome="Almeida";  
  
int idade=56;
```

```
frase="O nome dele é "+nome+", e  
sua idade é de "+idade+" anos.";
```


Como resultado, frase fica com o texto: *“O nome dele é Almeida, e sua idade é de 56 anos.”*.

Precedência de Operadores

Maior precedência

```
()  
! unário - unário + ++ --(post)  
++ --(pre)  
* / %  
+ -  
< <= > >=  
== !=  
&&  
||  
=
```

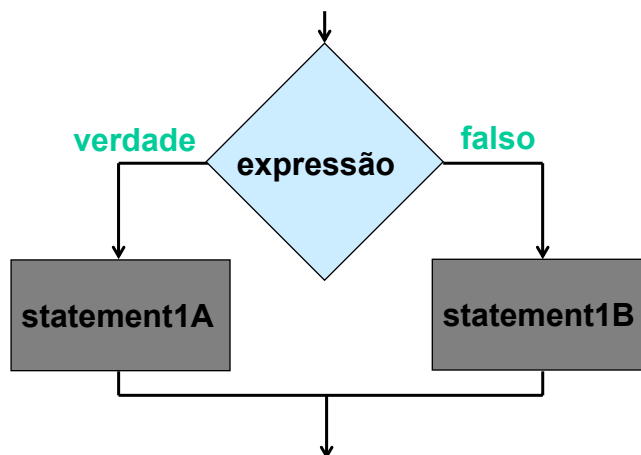
Lowest precedence



Estruturas de Desvio de Fluxo

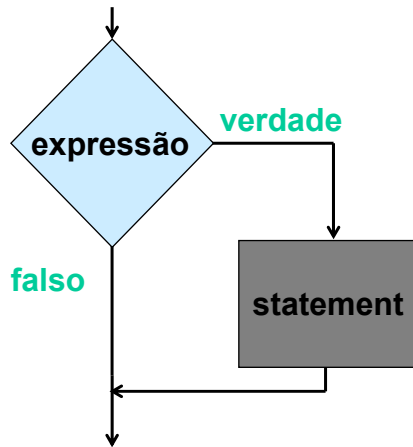
Estrutura de Controle *if-else*

Fornece a seleção em dois sentidos



if é uma instrução de seleção

Executa ou salta a instrução



if-else

if (*Expressão*)

Statement1A

else

Statement1B

```
if (Expressão)
```

```
{
```

```
}
```

```
else
```

```
{
```

```
}
```

Obs: Statement1A e Statement1B cada um pode ser uma simples instrução, uma instrução nula, ou um bloco de instruções.

if-else aninhados

```
if (Expressão1 )  
    Bloco1  
else if (Expressão2 )  
    Bloco2  
    ...  
else if (ExpressãoN )  
    BlocoN  
else  
    Bloco N+1
```

EXATAMENTE 1 desses blocos será executado.

if-else Aninhados

- Cada expressão é avaliada em seqüência, até que uma expressão verdadeira seja encontrada.
- Apenas o bloco de instruções específico que segue a expressão verdadeira é executado.
- Se nenhuma expressão for verdadeira, o bloco seguinte ao else final é executado.
- Realmente, o else final e o bloco final são opcionais. Se forem omitidos, e nenhuma expressão for verdadeira, então nenhuma instrução é executada.

UM EXEMPLO . . .

Uma única resposta possível

```
if (numero > 0)

    System.out.println("Positivo");

else
    if (numero < 0)

        System.out.println("Negativo");

    else

        System.out.println("Zero");
```

Em ausência de chaves:

Um *else* forma par sempre com o *if* precedente mais próximo que não tenha um *else* emparelhado com ele.

```
double average=100.0;

if (average >= 60.0)
    if (average < 70.0) // forma par com o else
        System.out.println("Marginal PASS");
else
    System.out.println("FAIL");
```

Por que a Saída é “FAIL” ?

O Compilador ignora a indentação e emparelha o *else* com o segundo *if*

Versão Corrigida

```
double average;  
  
average = 100.0;  
  
if (average >= 60.0)  
{  
    if (average < 70.0)  
        System.out.println("Marginal PASS");  
}  
else  
    System.out.println("FAIL");
```

Classes Revisited

class Name

```
{  
    String first;  
    String second;  
}
```

Classes are active; actions, called methods, are bound (encapsulated) with the class variables

Methods

❑ Method heading and block

```
void setName(String arg1, String arg2)
{
    first = arg1;
    second = arg2;
}
```

❑ Method call (invocation)

```
Name myName;

myName.setName("Nell", "Dale");
```

Some Definitions

❑ **Instance field** A field that exists in every instance of a class

```
String first;
String second;
```

❑ **Instance method** A method that exists in every instance of a class

```
void setName(String arg1, String arg2);
myName.setName("Chip", "Weems");
String yourName;
yourName.setName("Mark", "Headington");
```

More Definitions

- ❑ **Class method** A method that belongs to a class rather than its object instances; has modifier **static**

```
Date.setDefaultFormat(Date.MONTH_DAY_YEAR);
```

- ❑ **Class field** A field that belongs to a class rather than its object instances; has modifier **static**

More Definitions

- ❑ **Constructor method** Special method with the same name as the class that is used with `new` when a class is instantiated

```
public Name(String frst, String lst)
```

```
{
```

```
    first = frst;
```

```
    last = lst;
```

```
}
```

```
    Name name;
```

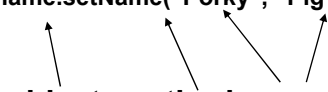
```
    name = new Name("John", "Dewey");
```

Note: argument cannot be the same as field

Void Methods

- ❑ **Void method** Does not return a value

```
System.out.print("Hello");  
System.out.println("Good bye");  
name.setName("Porky", "Pig");
```



object method arguments

The diagram shows three arrows pointing from the labels 'object', 'method', and 'arguments' to the corresponding parts of the `name.setName("Porky", "Pig");` call. The arrow from 'object' points to `name`, the arrow from 'method' points to `setName`, and the arrow from 'arguments' points to the opening parenthesis of the argument list.

Value-Returning Methods

- ❑ **Value-returning method** Returns a value to the calling program

```
String first; String last;  
Name name;  
System.out.print("Enter first name: ");  
first = inData.readLine();  
System.out.print("Enter last name: ");  
last = inData.readLine();  
name.setName(first, last);
```

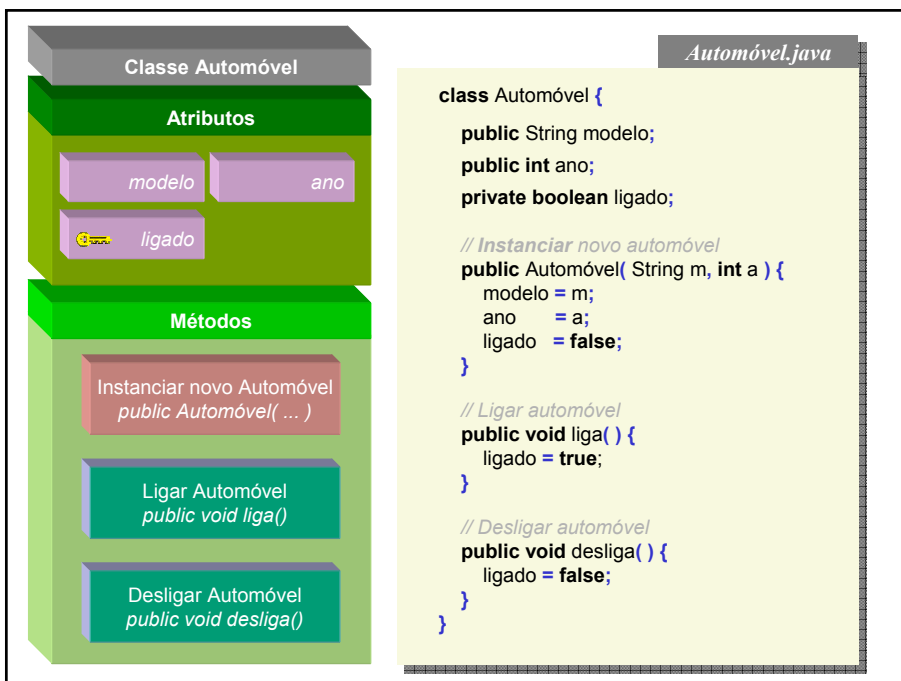
Value-returning example

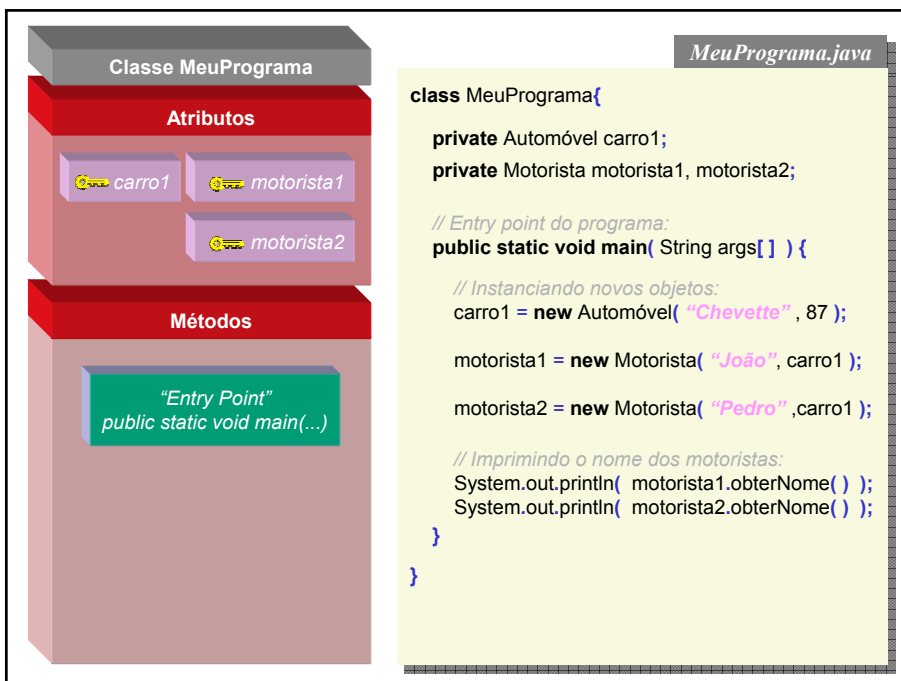
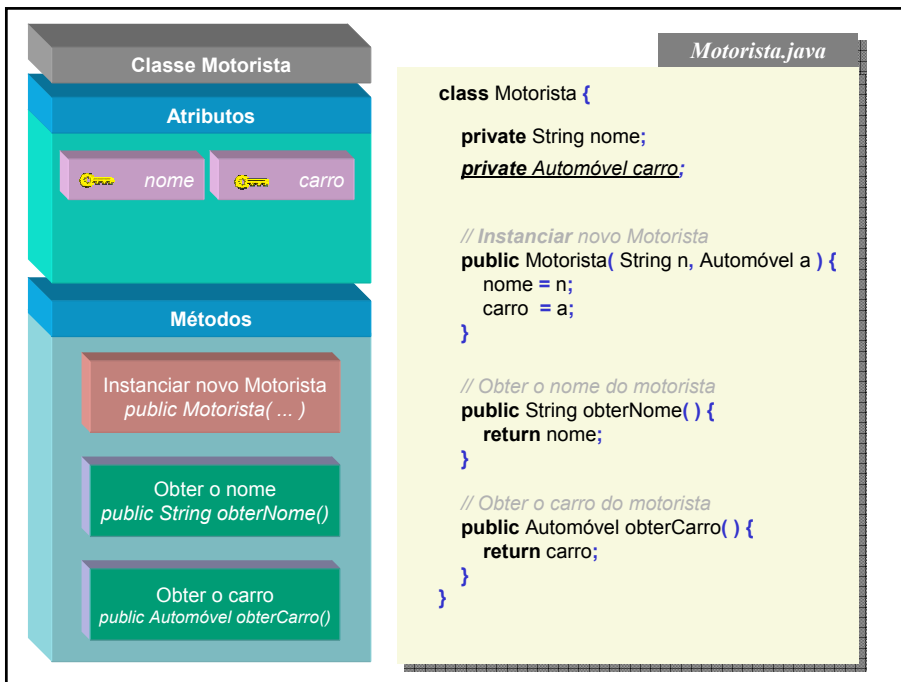
```
public String firstLastFormat()  
{  
    return first + " " + last;  
}
```

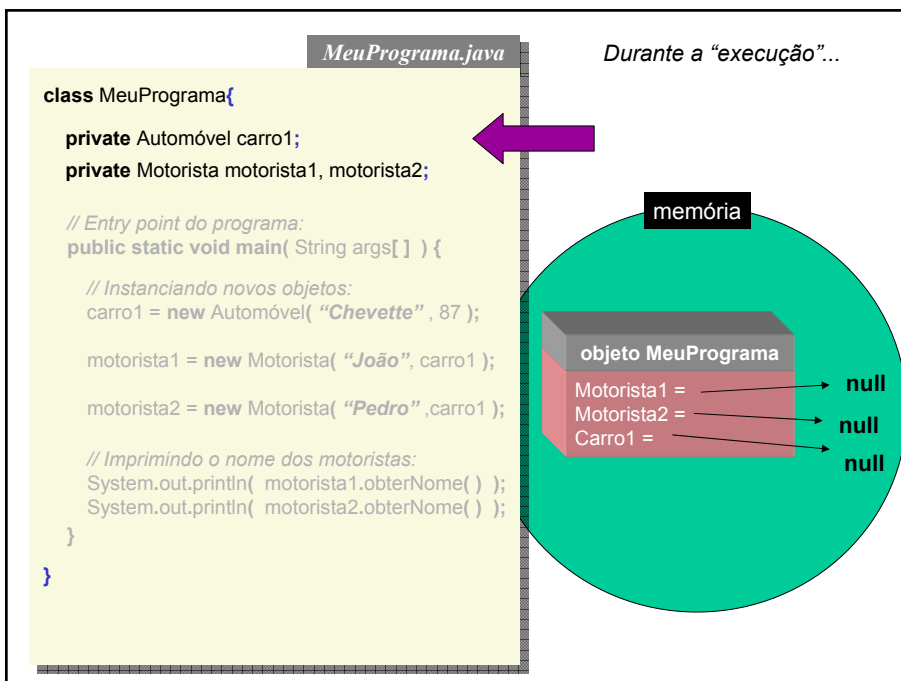
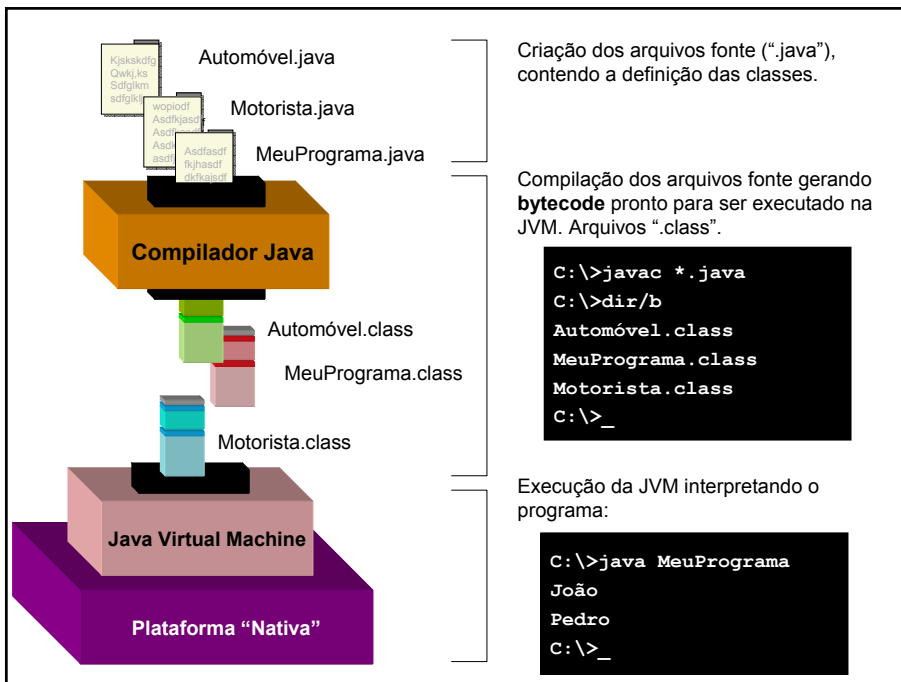
```
System.out.print(name.firstLastFormat());
```

object method object method

Argument to print method is string returned from firstLastFormat method



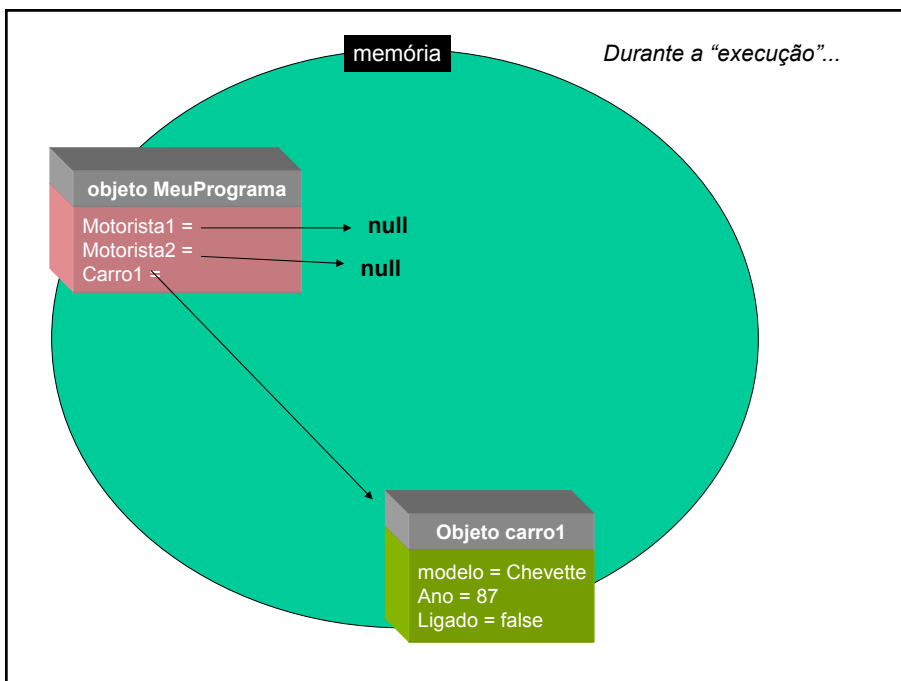




MeuPrograma.java

Durante a "execução"...

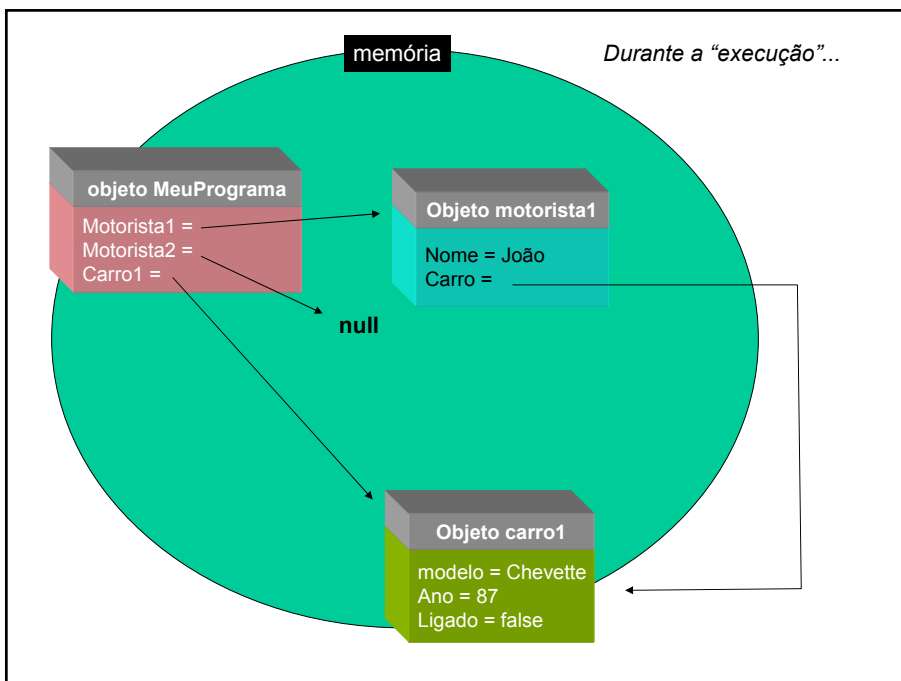
```
class MeuPrograma{  
    private Automóvel carro1;  
    private Motorista motorista1, motorista2;  
  
    // Entry point do programa:  
    public static void main( String args[ ] ){  
        // Instanciando novos objetos:  
        carro1 = new Automóvel( "Chevette", 87 );  
  
        motorista1 = new Motorista( "João", carro1 );  
  
        motorista2 = new Motorista( "Pedro", carro1 );  
  
        // Imprimindo o nome dos motoristas:  
        System.out.println( motorista1.obterNome( ) );  
        System.out.println( motorista2.obterNome( ) );  
    }  
}
```



MeuPrograma.java

```
class MeuPrograma{  
    private Automóvel carro1;  
    private Motorista motorista1, motorista2;  
  
    // Entry point do programa:  
    public static void main( String args[] ){  
        // Instanciando novos objetos:  
        carro1 = new Automóvel( "Chevette", 87 );  
  
        motorista1 = new Motorista( "João", carro1 );  
  
        motorista2 = new Motorista( "Pedro", carro1 );  
  
        // Imprimindo o nome dos motoristas:  
        System.out.println( motorista1.obterNome( ) );  
        System.out.println( motorista2.obterNome( ) );  
    }  
}
```

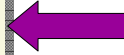
Durante a "execução"...



MeuPrograma.java

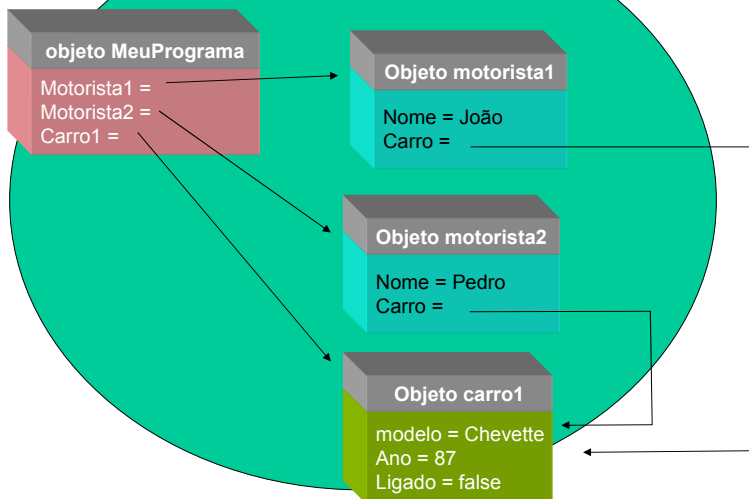
```
class MeuPrograma{  
  
    private Automóvel carro1;  
    private Motorista motorista1, motorista2;  
  
    // Entry point do programa:  
    public static void main( String args[] ){  
  
        // Instanciando novos objetos:  
        carro1 = new Automóvel( "Chevette", 87 );  
  
        motorista1 = new Motorista( "João", carro1 );  
        motorista2 = new Motorista( "Pedro", carro1 );  
  
        // Imprimindo o nome dos motoristas:  
        System.out.println( motorista1.obterNome( ) );  
        System.out.println( motorista2.obterNome( ) );  
    }  
}
```

Durante a "execução"...



memória

Durante a "execução"...



MeuPrograma.java

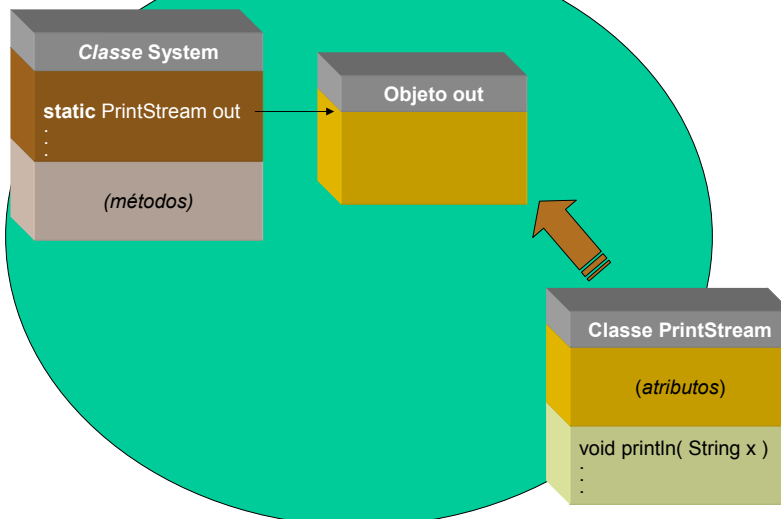
```
class MeuPrograma{  
    private Automóvel carro1;  
    private Motorista motorista1, motorista2;  
  
    // Entry point do programa:  
    public static void main( String args[] ){  
        // Instanciando novos objetos:  
        carro1 = new Automóvel( "Chevette", 87 );  
  
        motorista1 = new Motorista( "João", carro1 );  
  
        motorista2 = new Motorista( "Pedro", carro1 );  
  
        // Imprimindo o nome dos motoristas:  
        System.out.println( motorista1.obterNome() );  
        System.out.println( motorista2.obterNome() );  
    }  
}
```

Durante a "execução"...



memória

Durante a "execução"...



As três características da orientação à objetos:

(de forma simplificada e direta)



Encapsulamento

"Classes são estruturas que definem e guardam tanto dados quanto os algoritmos para tratar esses dados."



Herança

"Classes podem ser definidas a partir de outras classes."



Polimorfismo

"Uma dada variável pode em momentos distintos dentro da execução de um programa guardar tipo de dados diferentes."



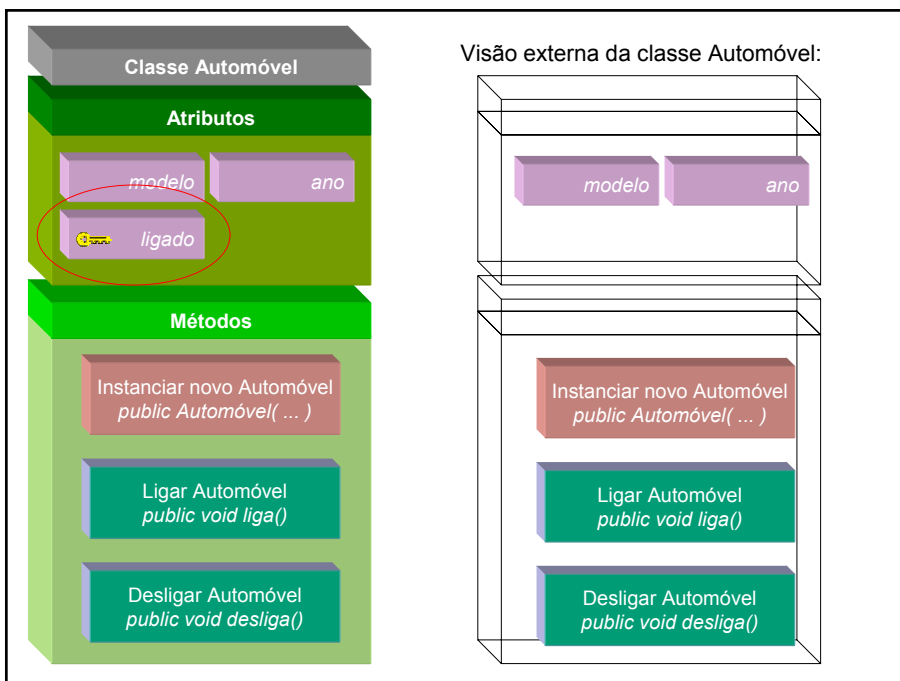
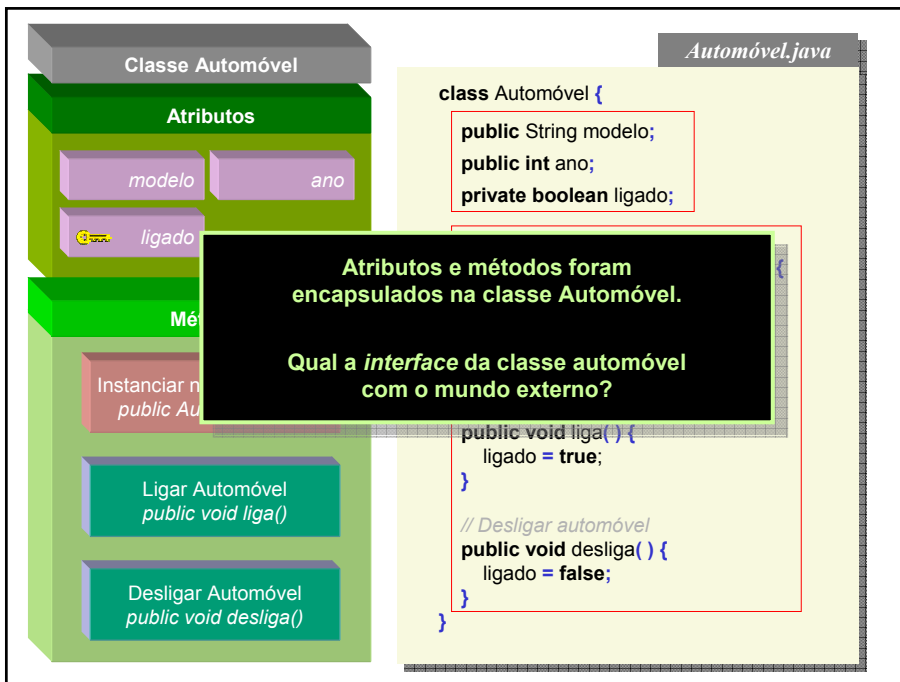
A linguagem Java e o encapsulamento:



A linguagem Java e a herança:



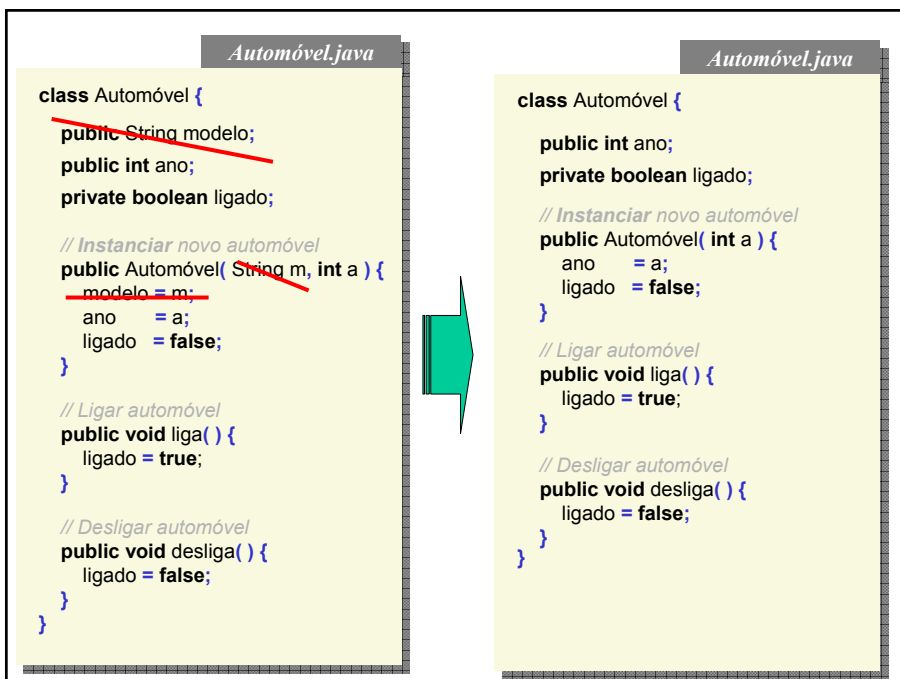
A linguagem Java e o polimorfismo:

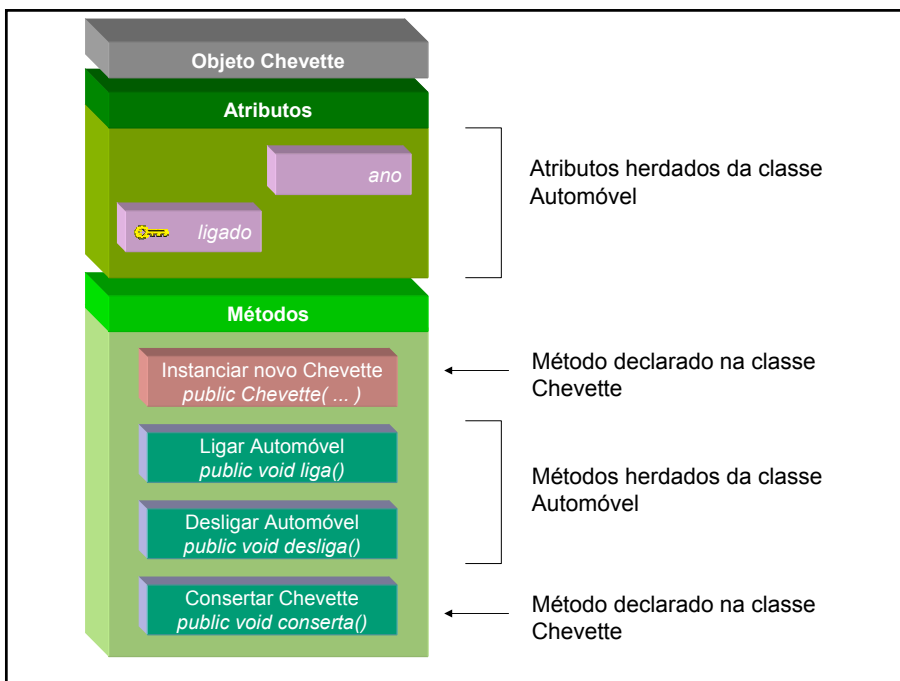
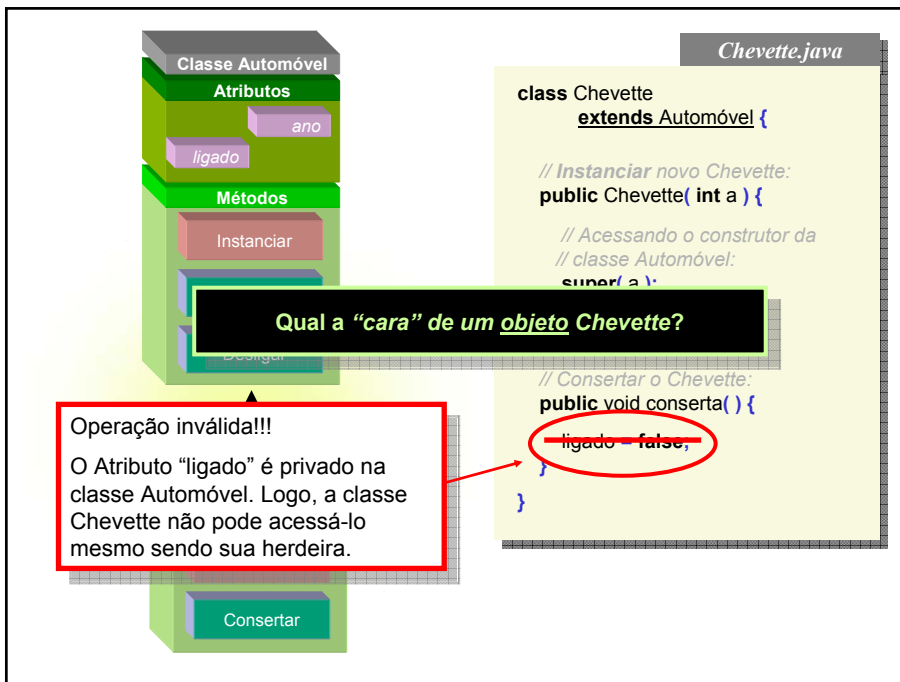


➡ A linguagem Java e o encapsulamento:

➡ A linguagem Java e a herança:

➡ A linguagem Java e o polimorfismo:





Operações válidas com um objeto Chevette:

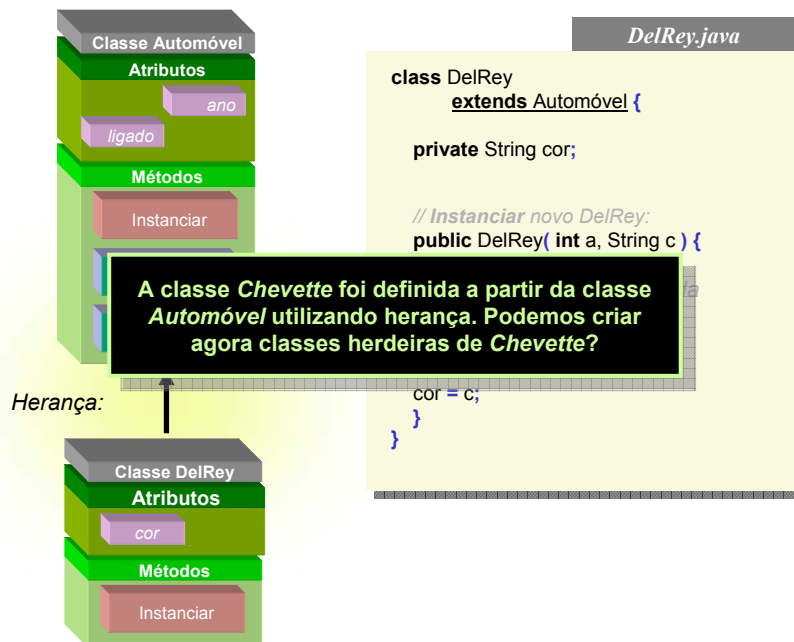
```
// Instanciando um novo Chevette:  
Chevette meuChevette = new Chevette( 88 );
```

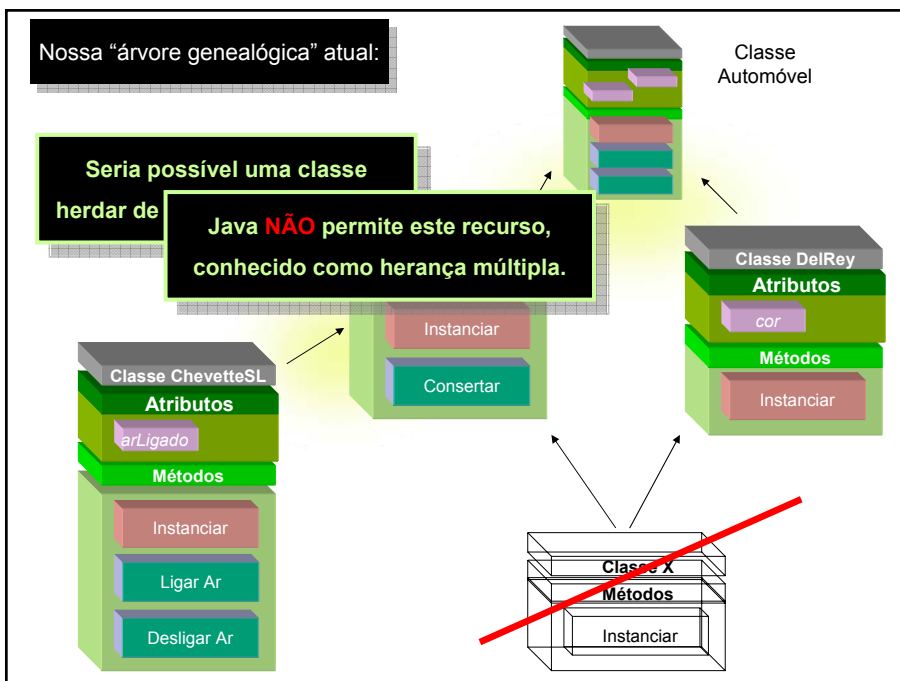
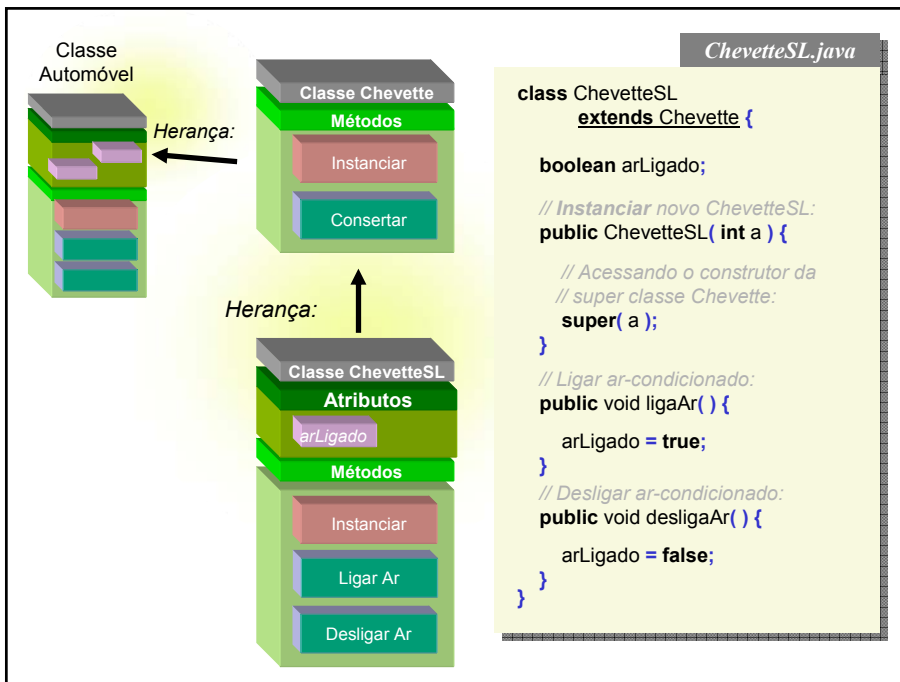
```
// Acessando o método declarado na classe
```

A classe *Chevette* é definida a partir da classe *Automóvel* utilizando herança. A classe *Automóvel* pode possuir outras herdeiras?

```
meuChevette.liga();  
meuChevette.desliga();
```

```
// Acessando os atributos declarados na super  
// classe Automóvel:  
meuChevette.ano = 2005;
```

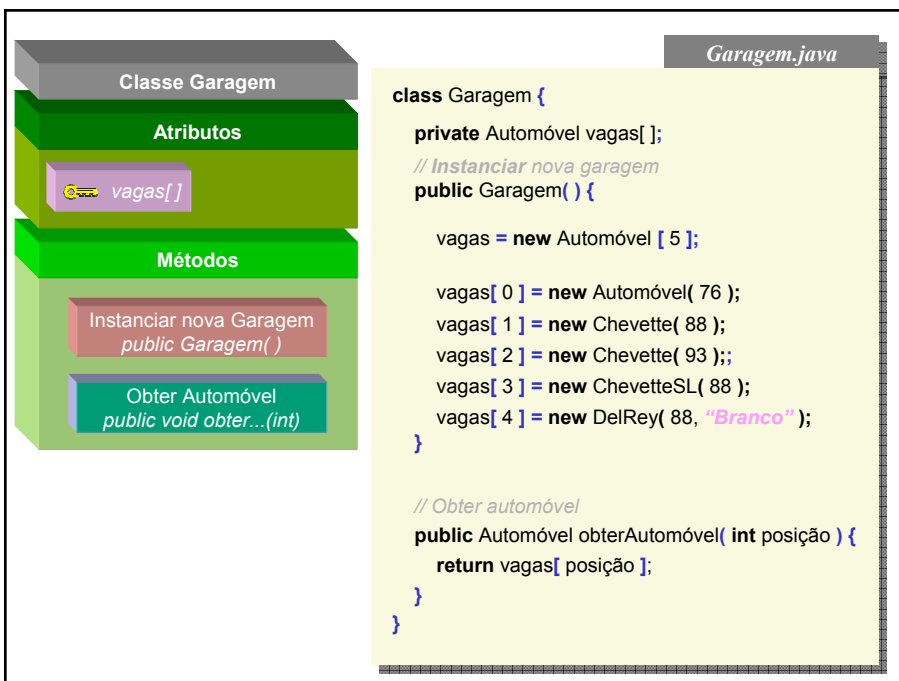




➡ A linguagem Java e o encapsulamento:

➡ A linguagem Java e a herança:

➡ A linguagem Java e o polimorfismo:

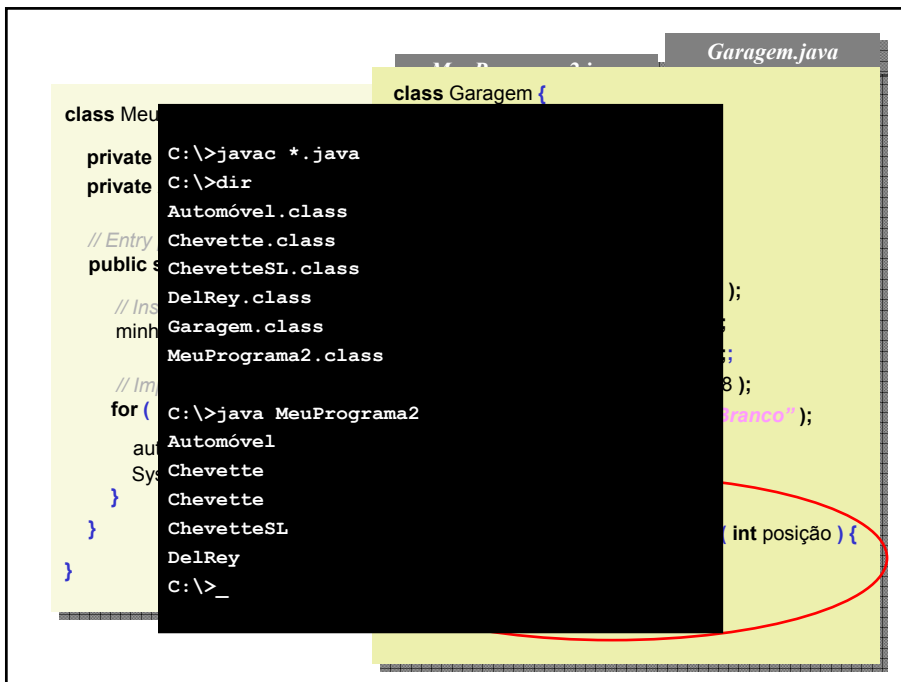


```
Garagem.java
class Garagem {
    // ...
    public void obterAutomovel( int posicao ) {
        // ...
    }
}

MeuPrograma2.java
class MeuPrograma2 {
    private Garagem minhaGaragem;
    private Automovel automovel;

    // Entry point do programa:
    public static void main( String args[] ){
        // Instanciando um novo objeto Garagem::
        minhaGaragem = new Garagem( );

        // Imprimindo a classe de cada automovel:
        for ( int i = 0; i < 5; i++ ){
            automovel = minhaGaragem.obterAutomovel( i );
            System.out.println( automovel.getClass().getName() );
        }
        (continuando...)
    }
}
```



```
MeuPrograma2.java
class MeuPrograma2 {
    private Garagem minhaGaragem;
    private Automovel automovel;

    // Entry point do programa:
    public static void main( String args[] ){
        // Instanciando um novo objeto Garagem::
        minhaGaragem = new Garagem( );

        // Imprimindo a classe de cada automovel:
        for ( int i = 0; i < 5; i++ ){
            automovel = minhaGaragem.obterAutomovel( i );
            System.out.println( automovel.getClass().getName() );
        }
        (continuando...)
    }
}
```


MeuPrograma2.java

```
// Imprimindo a classe de cada automóvel:
for ( int i = 0; i < 5; i++ ){

    automóvel = minhaGaragem.obterAutomóvel( i );
    System.out.println( automóvel.getClass().getName() );
}
(continuando...)

// Sabemos que a posição 3 possui um ChevetteSL.
// Vamos tentar obter esse carro e ligar o ar condicionado:
automóvel = minhaGaragem.obterAutomóvel( 3 );
automóvel.ligaAr( true );

// O código correto seria:
automóvel = (ChevetteSL) automóvel;
automóvel.ligaAr( true );
}
}
```

ERRO DE COMPILAÇÃO!!

Embora a “runtime class” da variável *automóvel* seja *ChevetteSL*, essa variável foi declarada como sendo *Automóvel*, e a classe *Automóvel* não possui o método “*ligaAr(...)*”