



AULA DE JAVA 1 - CONCEITOS BÁSICOS

DAS 5316 – Integração de Sistemas Corporativos

Roque Oliveira Bezerra, M. Eng

roque@das.ufsc.br

Prof. Ricardo J. Rabelo



ROTEIRO

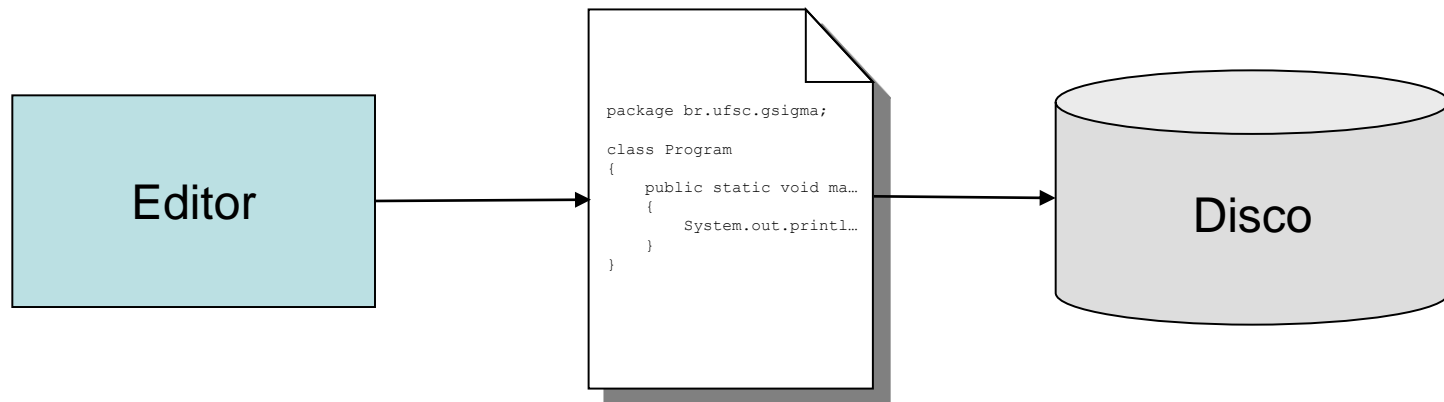
- Introdução
- Fases de um programa Java
- Strings
- Entrada e Saída
- Arrays
- Classes
 - Atributos
 - Métodos
 - Construtores
 - Herança
- Packages
- Interfaces

JAVA

- Java é um ambiente de execução completo, não apenas a linguagem de programação.
- Programas Java são compilados para *bytecode*, ou seja, um código *assembly* independente de arquitetura;
- O *bytecode* é interpretado na *Java Virtual Machine* (JVM);
- Compilação JIT (*Just in Time*)
 - A medida que a JVM detecta que um trecho de código será executado diversas vezes, este é convertido, e passa a executar na CPU real.

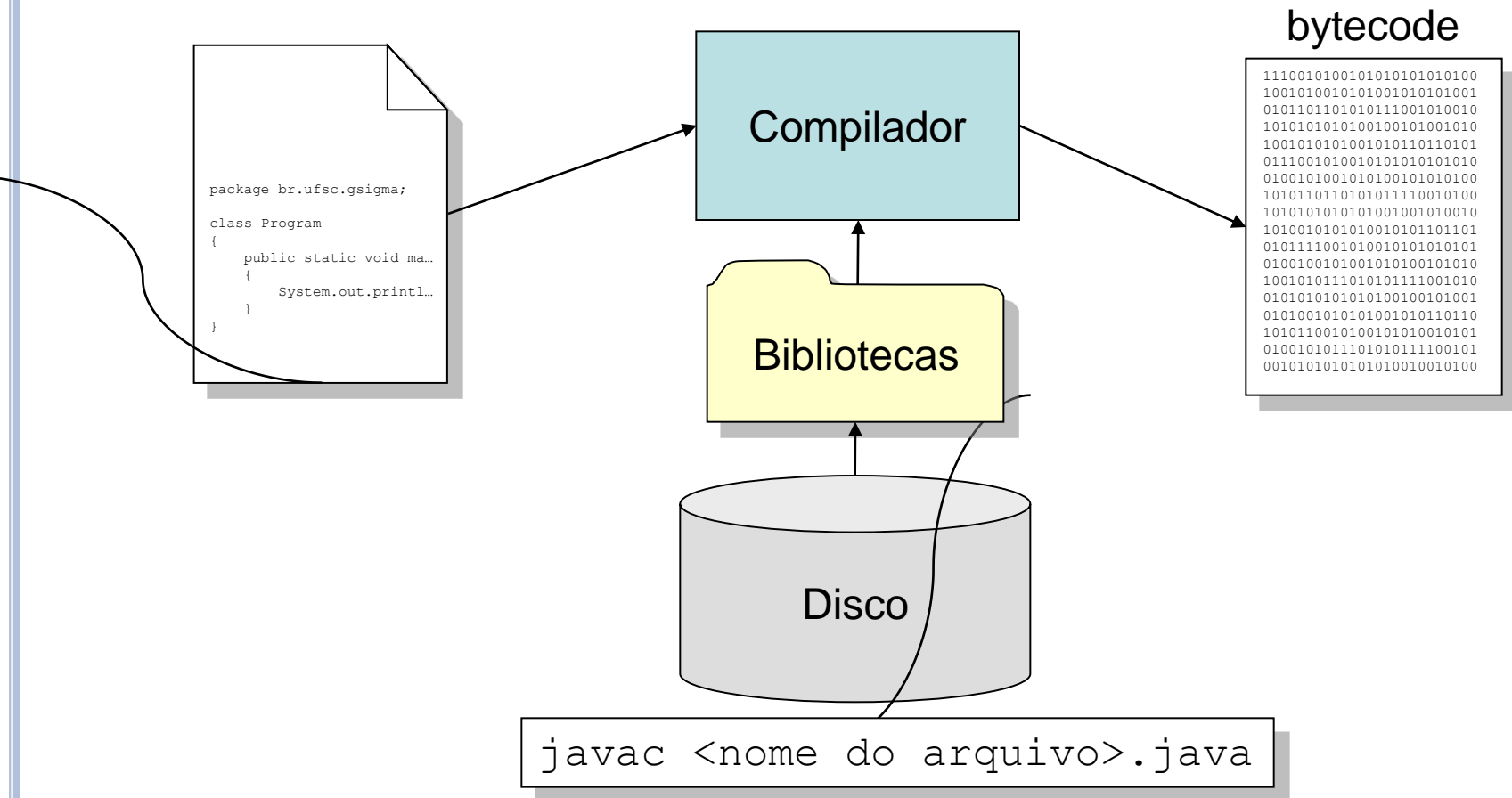
FASES DE UM PROGRAMA JAVA

- Edição



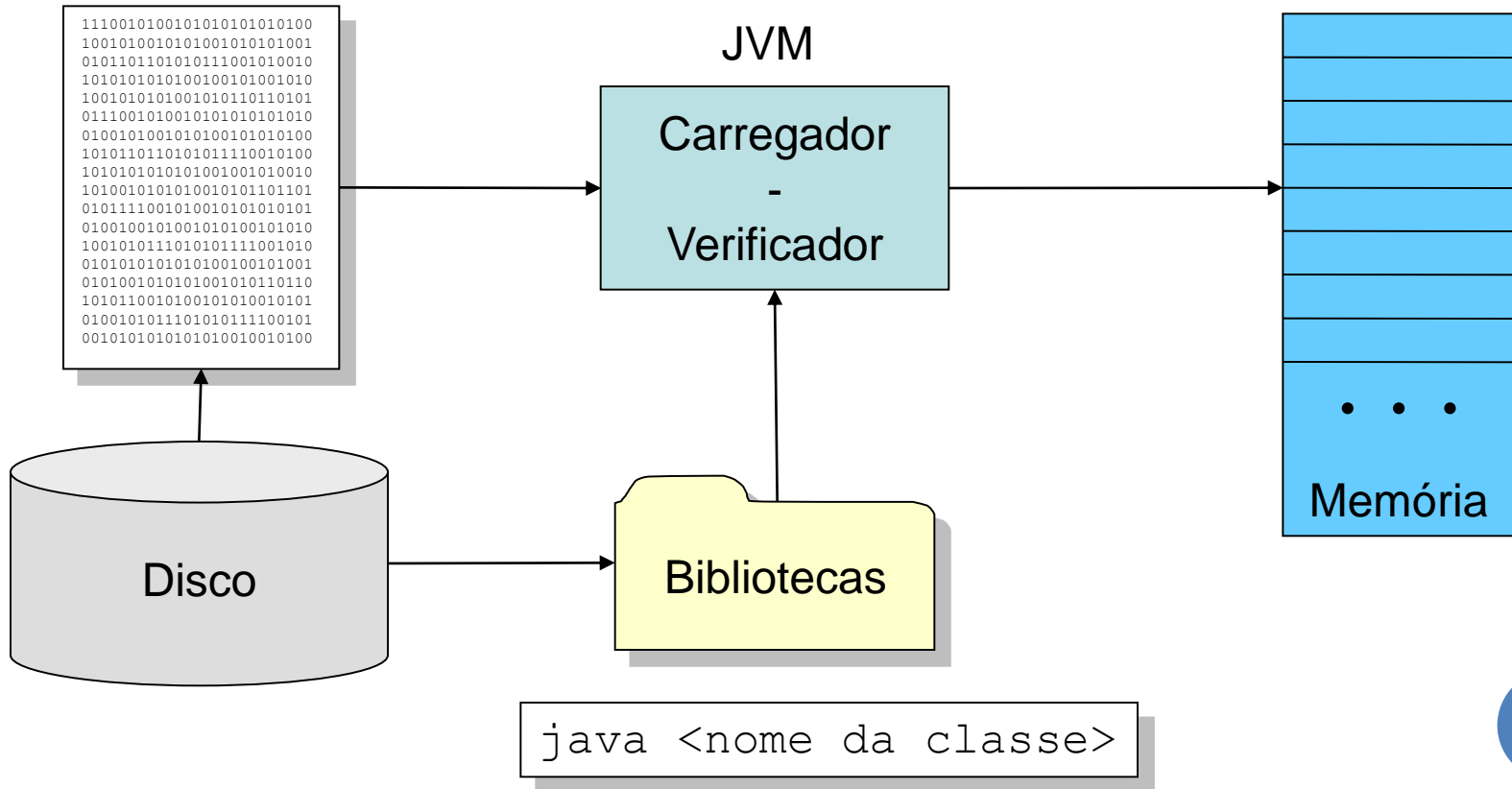
FASES DE UM PROGRAMA JAVA

○ Compilação



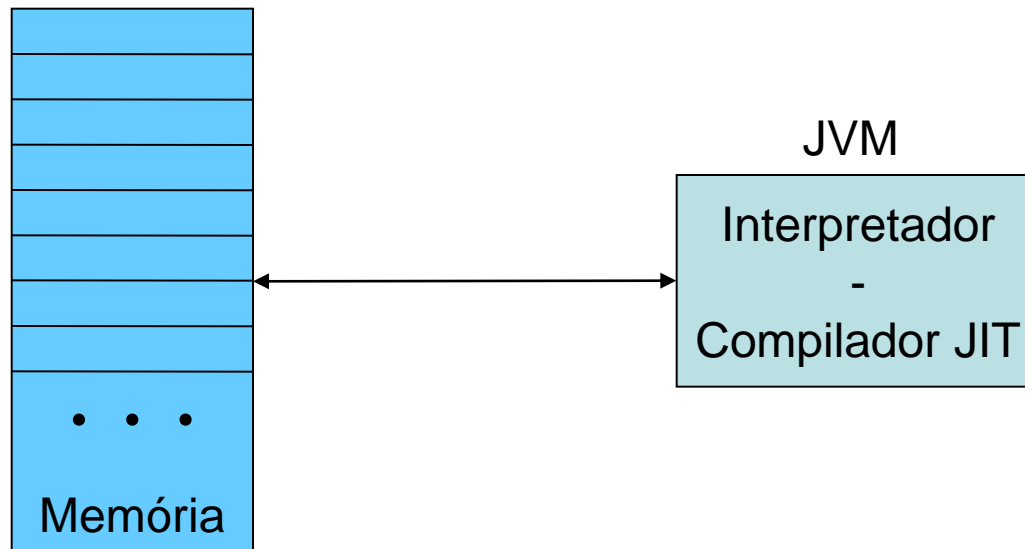
FASES DE UM PROGRAMA JAVA

Carregamento



FASES DE UM PROGRAMA JAVA

- Interpretação



UM EXEMPLO SIMPLES

```
public class Exemplo
{
    public static void main(String[] args)
    {
        System.out.println("Alô, mundo");
    }
}
```


UM EXEMPLO SIMPLES

```
public class Exemplo
```

```
{
```

```
    public
```

```
    {
```

```
        System
```

```
    }
```

```
}
```

class: Indica a declaração de uma classe.

Tudo em Java deve ficar dentro de uma classe, não existem variáveis ou funções globais

```
    args)
```

```
;
```

UM EXEMPLO SIMPLES

```
public class Exemplo
{
    public static void main(String[] args)
    {
        System.out.println("Alô, mundo");
    }
}
```

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

Método *main*, ponto de entrada do programa

UM EXEMPLO SIMPLES

```
public class Exemplo
{
    public static void main(String[] args)
    {
        System.out.println("Alô, mundo");
    }
}
```

Chamada de método

Chamadas de métodos são feitas seguindo a forma:

<objeto>.<método> (<parâmetros>)

Neste caso, o objeto é `System.out`, o método é `println`, e o parâmetro é a string `"Alô, mundo"`

COMENTÁRIOS

- Java suporta três tipos de comentário:

- De linha (//)

```
System.out.println("Alô, mundo"); //Que função legal...
```

- De bloco (*/* ... */*)

```
/* Esse comando serve pra fazer isso,  
depois de chamado, vai acontecer aquilo */
```

- De documentação (*/** ... */*)

```
/**  
Esse método calcula xyz, baseado em abc  
@param abc o abc usado no cálculo  
*/  
public void metodoX(int abc) { ... }
```

TIPOS DE DADOS

○ Inteiros

- **byte** 1 byte, -128 a 127
- **short** 2 bytes, -32.768 a 32.767
- **int** 4 bytes, -2.147.483.648 a 2.147.483.647
- **long** 8 bytes, -9.223.372.036.854.775.808 a 9.223.372.036.854.775.807

○ Ponto Flutuante

- **float** 4 bytes, ~ ±3.40282347E+38
- **double** 8 bytes, ~ ±1.79769313486231570E+308

○ Caracter

- **char** 2 bytes, '\u0000' a '\uffff'

○ Booleano

- **boolean** true ou false

VARIÁVEIS

- Todas as variáveis precisam ter um tipo associado
- Variáveis são declaradas indicando primeiramente o tipo, e em seguida o nome
- Pode-se opcionalmente inicializar uma variável durante sua declaração
- Usar uma variável antes de atribuir um valor a ela é um erro de programação (Null Pointer Exception)

```
int diasDeFolga = 30;  
boolean completo;  
  
completo = false;
```

STRINGS

- *Strings* são seqüências de caracteres
- Java não possui um tipo primitivo específico para representar *strings*, em vez disso elas são encapsuladas pela classe `String`
- Até mesmo *strings* literais (delimitadas por aspas), são instâncias da classe `String`
- *Strings* podem ser criadas a partir de literais, ou pela concatenação de *strings* com outras variáveis

```
String str = "Alô";  
int x = 30;  
String str2 = str + " " + x;  
  
//str2 == "Alô 30"
```

COMPARANDO STRINGS

- Para comparar a igualdade de duas *strings* deve-se usar o método *equals*:

```
if ( str.equals("Alô") ) ...  
  
if ( "Alô".equals(str) ) ...
```

- Caso se queira comparar *strings* sem levar em conta a diferença entre maiúsculas e minúsculas, pode-se usar o método *equalsIgnoreCase*:

```
if ( str. equalsIgnoreCase("Alô") )  
    ...
```

- NÃO** se deve comparar *strings* com `==`

CONVERSÃO DE STRINGS

- Para converter tipos simples para *string* existe o método `valueOf()`, da classe `String`:
 - `String str1 = String.valueOf(23);`
 - `String str2 = String.valueOf(50.75);`
- Para a conversão de *strings* para tipos simples também existem métodos:
 - `int x = Integer.parseInt("42");`
 - `float f = Float.parseFloat("3.14159");`
- Se a conversão não for possível, uma exceção é lançada.

STRING: ALGUNS MÉTODOS

- **int** `length()`
 - Comprimento da *string*
- **char** `charAt(int index)`
 - Retorna o caracter na posição requerida
- **int** `indexOf(String str)`
 - Retorna a posição onde *str* fica na *string*, ou -1 se não encontrar
- `String substring(int beginIndex, int endIndex)`
 - Cria uma *substring*, com os caracteres contidos entre `beginIndex` e `endIndex`
- **int** `compareTo(String other)`
 - Compara com outra *string*, e retorna 0 se forem iguais, -1 se esta for menor que a outra, ou 1 em caso contrário

ENTRADA E SAÍDA

- A leitura e a escrita de dados com o usuário é feita, respectivamente, pelos objetos:
 - `System.in`
 - `System.out`
- Os principais métodos de `System.out` são
 - `print(...)`
 - Imprime o conteúdo de uma variável ou expressão
 - `println(...)`
 - Imprime o conteúdo de uma variável ou expressão, e uma quebra de linha

ENTRADA E SAÍDA

- A partir da versão 5.0 (1.5), Java provê a classe `java.util.Scanner` para leitura de dados de `System.in`

```
import java.util.Scanner;

public class InputTest
{
    public static void main(String[] args)
    {
        Scanner entrada = new Scanner(System.in);

        System.out.print("Qual é seu nome? ");
        String nome = entrada.nextLine();

        System.out.print("Quantos anos você tem? ");
        int idade = entrada.nextInt();

        System.out.println("Olá, " + nome + ". Sua idade é: " + idade);
    }
}
```

ARRAYS

- *Arrays* são estruturas de dados que armazenam uma seqüência de tamanho fixo de valores de um mesmo tipo.

```
int[] numeros; //array de int  
  
String[] nomes; //array de String
```

- Assim como qualquer variável, *arrays* precisam ser inicializados antes de serem usados, e isso é feito:
 - Usando o operador **new**, e o tamanho desejado
 - Fornecendo os valores diretamente

```
String[] nomes = new String[1024];  
int[] numeros;  
numeros = new int[100];  
char[] abc = { 'a', 'b', 'c' };
```

ARRAYS, ACESSANDO ELEMENTOS

- Após a inicialização, valores podem ser atribuídos a índices do *array* ou pode-se ler o valor atribuído a um índice
- Índices começam em 0
- O tamanho de um *array* sempre pode ser obtido pelo atributo (de apenas leitura) `length`

```
nomes[0] = "Arthur";  
nomes[1] = "Ford";  
  
int tamanho = nomes.length;  
  
String ultimo = nomes[nomes.length-1];
```

ITERANDO SOBRE ARRAYS

- Pode-se iterar sobre *arrays* de duas formas

- Acessando os elementos por seus índices

```
String[] array = ...;

for (int i = 0; i < array.length; i++)
{
    String str = array[i];
    //Usa str
}
```

- Navegando diretamente pelos elementos

```
for (String str : array)
{
    //Usa str
}
```

ARRAYS MULTIDIMENSIONAIS

- Java não possui uma construção explícita para *arrays* multidimensionais.
- Porém, é permitido criar *arrays* de *arrays*, o que é equivalente
- Além disso, há uma sintaxe especial para inicializar estes *arrays*

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

```
String dados[][][] = new String[300][10][50];
```


ENUMERAÇÕES

- Uma enumeração é um tipo cujos valores possíveis pertencem a um conjunto limitado, pré-definido

```
enum Naipe { Espadas, Ouros, Copas, Paus }  
  
Naipe n = Naipe.Espadas;
```

- Tipos enumerados podem ser usados em *switches*

```
switch (n)  
{  
    case Espadas:  
        ...  
        break;  
    ...  
}
```

ENUMERAÇÕES

- Uma enumeração é um tipo cujos valores possíveis pertencem a um conjunto limitado, pré-definido

```
enum Naipe { Espadas, Ouros, Copas, Paus }
```

```
Naipe n = Naipe.Espadas;
```

- Tipos enumeração

Sempre que se for usar um dos possíveis valores de uma enumeração, deve-se qualificá-lo com o nome da enumeração.

Usa-se `Naipe.Espadas`, e não simplesmente `Espadas`

Isso acontece porque mais de uma enumeração pode ter o valor `Espadas`. (Ex.: `Armas.Espadas`)

```
}
```

ENUMERAÇÕES

Em *switches*, entretando, o compilador sabe, pelo tipo da variável *n*, a qual **enum** este *Espadas* pertence. Então não é necessário indicar o tipo.

Aliás, estranhamente, é proibido qualificar um valor de enumeração em um *switch*. O compilador gera um erro nestes casos

- Tipos enumerados podem ser usados em ***switches***

```
switch (n)
{
    case Espadas:
        ...
        break;
    ...
}
```

CLASSES EM JAVA

- A unidade básica da Linguagem Java é a Classe;
- Programas Java são compostos de objetos que interagem entre si trocando mensagens (invocando métodos).

EXEMPLO

```
public class Motor
{
    //Atributos
    private int marcha = 0;
    private int rotação = 0;

    //Construtores
    public Motor(int marcha) { this.marcha = marcha; }

    public Motor() { }

    //Métodos
    public void sobeMarcha() { marcha++; }

    public void desceMarcha() { marcha--; }

    public int getMarcha() { return marcha; }

    //Outros métodos...
}
```

CLASSES

- O corpo de uma classe pode conter:
 - Atributos;
 - Métodos;
 - Construtores.

ATRIBUTOS

- Atributos são variáveis que expressam o estado de um objeto;
- Como qualquer variável, podem ser de tipos simples (**int**, **float**, **boolean**, etc.), um tipo referência (classe ou interface), ou ainda um *array*. Ex:

```
private int x;  
private boolean[] b;  
private Motor motor1;  
private Acelerável[] ac;
```

- Atributos podem ser inicializados em sua declaração. Ex:

```
private int x = 20;  
private Motor motor1 = new Motor();  
private Acelerável[] ac = new Bicicleta[5];
```

ATRIBUTOS (CONT)

- É recomendável que atributos sejam declarados como **private**, garantindo assim o encapsulamento dos dados;
 - Os valor do atributo deve ser acessado através de *getters* e *setters*;
 - Exemplo:

```
private int idade;  
  
public int getIdade() {  
    return idade;  
}  
  
public void setIdade(int idade) {  
    this.idade = idade;  
}
```


MÉTODOS

- Métodos são ações que objetos podem executar;
- Podem possuir parâmetros, que assim como atributos podem ser de qualquer tipo simples, tipo referência, ou *array*;
- Métodos podem executar operações que retornam ou não resultados. No primeiro caso seu tipo de retorno deve ser indicado, no segundo ele deve ser declarado como **void**.

```
public void fazCoisa(int param) { ... }  
public int calculaValor(int p1, float p2) { ... }
```

- Métodos podem ter o mesmo nome, desde que tenham número e/ou tipo de parâmetros diferentes entre si.

MÉTODOS (CONT.)

- Dentro de métodos pode-se usar a palavra chave **this** para fazer referência ao objeto sobre o qual o método foi chamado
- Métodos que retornam algum valor devem fazê-lo utilizando a palavra-chave **return** seguida do valor a ser retornado.
- Métodos **void** podem também utilizar **return;** para encerrar sua execução a qualquer momento;
- Métodos em geral são declarados como **public**, para que sejam acessíveis externamente. Mas métodos que são apenas utilizados internamente devem ser declarados como **private**.

EXEMPLO

```
public class Motor
{
    private int marcha = 1;

    public void sobeMarcha() {
        marcha++;
    }

    public void mudaMarcha(int marcha) {
        if (rotaçãoAdequada())
            this.marcha = marcha;
    }

    public int getMarcha() {
        return marcha;
    }

    private boolean rotaçãoAdequada() {
        //...
    }
}
```

CONSTRUTORES

- Um construtor é um tipo especial de método;
- Um construtor não tem tipo de retorno (nem mesmo **void**) e pode possuir quantos parâmetros forem necessários;
- Um objeto pode possuir vários construtores.

```
public class Motor
{
    private int marcha = 0;
    private int rotação = 0;

    public Motor(int marcha) { this.marcha = marcha; }

    public Motor() { }
}
```

CONSTRUTORES (CONT.)

- Se nenhum for declarado, um construtor padrão, vazio, é criado implicitamente;
- Dentro dos construtores pode ser feita a inicialização de atributos e qualquer outra operação necessária para o objeto;
- O ideal é que depois de construído, o objeto esteja pronto para operar;
- Objetos são criados usando a palavra chave **new**, seguida do nome da classe e dos parâmetros do construtor.

```
public class Carro {  
    private Motor motor;  
    public Carro() {  
        motor = new Motor();  
    }  
}
```

MÉTODOS ESTÁTICOS

- Métodos estáticos são métodos que não operam em objetos

```
double x = Math.pow(3.5, 2);  
  
int[] array = ...;  
Arrays.sort(array);  
  
String x = String.valueOf(2341);
```

- São definidos pela palavra chave **static**

```
public static int max(int a, int b)  
{  
    return a > b ? a : b;  
}
```

- Métodos estáticos não podem acessar atributos de objeto, pois estes são relativos a uma instância da classe, que não existe neste contexto

MÉTODO MAIN

- O método *main* é um método estático especial, usado como ponto de partida de um programa Java;
- Deve ser declarado como:

```
public static void main(String[] args)
{
    //comandos...
}
```

- O *array* de *strings* é a lista de argumentos de linha de comando;
- Pode-se declarar métodos *main* em qualquer classe, sendo isto muito usado para testar classes individualmente

HERANÇA

- Para declarar uma classe derivada de outra utiliza-se a palavra chave **extends**:
- Uma subclasse enxerga tudo o que não foi declarado como **private** na superclasse

```
public class Carro
{
    private int velocidade;

    public int getVelocidade() { return velocidade; }
}

public class Formula1 extends Carro
{
    public int calculoQualquer() { return getVelocidade() * 20; }
}
```


HERANÇA

- Para declarar uma classe derivada de outra utiliza-se a palavra chave **extends**:
- Uma subclasse ~~herda~~ **herda** tudo o que não foi declarado

Caso se tentasse acessar diretamente o atributo `velocidade`, ocorreria um erro de compilação

```
private int velocidade;

public int getVelocidade() { return velocidade; }
}

public class Formula1 extends Carro
{
    public int calculoQualquer() { return getVelocidade() * 20; }
}
```

HERANÇA (CONT.)

- Uma subclasse pode redefinir um método da superclasse, se ele não for **private**;
 - Isso é chamado de sobrescrita (*override*)
- Esta característica é chamada de *polimorfismo*: diferentes objetos podem ter comportamentos diferentes em relação a um mesmo método.
- Por exemplo, a classe *Carro* pode calcular seu deslocamento de uma certa forma. A classe *Formula1* precisa levar mais dados em consideração, como a pressão aerodinâmica em seus aerofólios. Então ela reimplementa o método para o cálculo do deslocamento;
- Pode-se usar a palavra chave **super** para chamar métodos e construtores da superclasse.

EXEMPLO SUPER

```
public class Carro
{
    ...
    public Carro(Motor m) { ... }

    public int deslocamento() { return motor.getAceração() * 20; }
}

public class Formula1 extends Carro
{
    ...
    public Formula1(Motor m, float inclAerofólio)
    {
        super(m);
        ...
    }

    public int deslocamento() { return motor.getAceração() * inclAerofólio; }
}
```

PROTECTED

- A palavra-chave **protected** é um meio termo entre **public** e **private**, para a declaração de membros
 - Eles são, em geral, vistos como se fossem **private**
 - Mas para subclasses eles são como **public**
- Exeto em casos muito especiais, deve-se evitar o uso de **protected**, pois ele quebra o encapsulamento da superclasse

VINCULAÇÃO DINÂMICA

- A criação de hierarquias de classes permite que se trate, de forma abstrata, objetos de classes especializadas como se fossem de classes mais gerais
- Pode-se fazer:

```
Carro[] carros = new Carro[2];

carros[0] = new Carro();
carros[1] = new Formula1();

for (Carro c : carros)
{
    int desloc = c.deslocamento();
    System.out.println( desloc );
}
```

- Quando um método é chamado, não importa o tipo *declarado* da variável, a máquina virtual invoca o método com base do tipo *real* dela

VINCULAÇÃO DINÂMICA

- A criação de hierarquias de classes com classes especializadas como subclasses
- Pode-se fazer:

O ambiente de execução “sabe” que neste ponto, se o *Carro* em questão for um *Formula1*, ele deve chamar a versão do método definida nesta classe, e não a versão mais geral, definida na classe *Carro*

```
Carro[] carros = new Carro[2];  
  
carros[0] = new Carro();  
carros[1] = new Formula1();  
  
for (Carro c : carros)  
{  
    int desloc = c.deslocamento();  
    System.out.println( desloc );  
}
```

- Quando um método é chamado, não importa o tipo *declarado* da variável, a máquina virtual invoca o método com base do tipo *real* dela

COERÇÃO DE OBJETOS

- Suponha que o seguinte método seja adicionada à classe Formula1:

```
String[] getPatrocinadores() {...}
```

- Caso se queira chamar esse método a partir de uma variável do tipo Carro, é preciso informar ao compilador explicitamente que aquela variável guarda um Formula1 mesmo, e não um carro qualquer

```
Carro[] carros = ...;  
//erro de compilação  
String[] p = carros[0].getPatrocinadores();  
  
//Compilador aceita  
Formula1 f = (Formula1) carros[0];  
  
String[] p = f.getPatrocinadores();
```

COERÇÃO DE OBJETOS

- Suponha que o seguinte método seja adicionada à classe Formula1:

```
String[] getPatrocinadores() {...}
```

- Caso se queira chamar esse método a partir de uma variável do tipo Carro, é preciso informar ao compilador explicitamente que aquela variável guarda um Formula1 mesmo, e não um carro qualquer

```
Carro[] carros = ...;  
//erro de compilação  
String[] p = carros[0].getPatrocinadores();  
  
//Compilador aceita  
Formula1 f = (Formula1) carros[0];  
String[] p = f.getPatrocinadores();
```

Typecast,
ou coerção

COERÇÃO DE OBJETOS

- Suponha que o seguinte método seja adicionada à classe Formula1:

```
String[] getPatrocinadores() {...}
```

- Caso se queira chamar esse método a partir de uma variável do tipo Carro, é preciso informar ao compilador explicitamente que aquela variável guarda um Formula1

Erro em tempo de execução caso
carros[0] não seja um Formula1

ClassCastException

```
Carro[] carros;
```

```
//erro de compilação
```

```
String[] p = carros[0].getPatrocinadores();
```

```
//Compilador aceita
```

```
Formula1 f = (Formula1) carros[0];
```

```
String[] p = f.getPatrocinadores();
```

CHECAGEM DE TIPOS

- Para se certificar que um objeto é mesmo de um dado tipo, e assim evitar erros, pode-se checar em tempo de execução o real tipo de um objeto.
- Pode-se checar explicitamente se a classe de um objeto é uma dada classe, ou usar o operador **instanceof**.

```
if ( carros[0].getClass() == Formula1.class )
{
    Formula1 f = (Formula1)carros[0];
}

if ( carros[0] instanceof Formula1 )
{
    Formula1 f = (Formula1)carros[0];
}
```

CHECAGEM DE TIPOS

- Para se certificar que um objeto é mesmo de um dado tipo, e assim evitar erros, pode-se checar em tempo de execução o real tipo de um objeto.
- Pode-se checar explicitamente se a classe de um objeto é uma dada classe, ou usar o operador **instanceof**.

```
if ( carros[0].getClass() == Formula1.class )  
{  
    Formula1 f = (Formula1)carros[0];  
}
```

Checa se o objeto é desta
classe em específico

```
if ( carros[0] instanceof Formula1 )  
{  
    Formula1 f = (Formula1)carros[0];  
}
```

CHECAGEM DE TIPOS

- Para se certificar que um objeto é mesmo de um dado tipo, e assim evitar erros, pode-se checar em tempo de execução o real tipo de um objeto.
- Pode-se checar explicitamente se a classe de um objeto é uma dada classe, ou usar o operador **instanceof**.

```
if ( carros[0].getClass() == Formula1.class )  
{
```

Checa se o objeto é desta classe, ou de uma classe descendente

```
    Formula1 f = (Formula1)carros[0];  
}
```

```
if ( carros[0] instanceof Formula1 )  
{  
    Formula1 f = (Formula1)carros[0];  
}
```

PACKAGES

- *Packages* criam escopos para declaração de classes;

```
package instrumentos;  
  
public class Teclado  
{  
    void tocar();  
}
```

```
package perifericos;  
  
public class Teclado  
{  
    char ultimaTecla();  
}
```

- A *package* faz parte do nome da classe.

```
instrumentos.Teclado ti;  
perifericos.Teclado tp;  
  
ti.tocar();  
char c = tp.ultimaTecla();
```

PACKAGES

- Declarações de *import* permitem usar classes sem a qualificação da *package*.

```
package instrumentos;  
  
public class Teclado  
{  
    void tocar();  
}
```

```
package teste;  
  
import instrumentos.Teclado;  
...  
Teclado t;  
t.tocar();
```

- Pode-se importar todas as classes de uma package ou apenas uma classe específica;

```
import nome.da.package.*;  
import nome.da.package.NomeDaClasse;
```

- A *package* `java.lang` é importada implicitamente.

PACKAGES

- Arquivos com declaração de *package* devem ter estrutura de diretório especial.

Package	Diretório
exemplo	exemplo/
br.ufsc.gsigma	br/ufsc/gsigma/

- Recomenda-se que nomes de package sejam em minúsculas, e sigam o nome do domínio de internet do desenvolvedor.

PACKAGES E DECLARAÇÕES DE CLASSES/INTERFACES

- Classes e interfaces podem ou não ser declaradas como públicas;
 - As não declaradas como públicas são visíveis apenas por outras classes e interfaces declaradas na mesma package;
 - As públicas tem visibilidade externa total;
 - Arquivo deve ter o mesmo nome da classe/interface;
 - Apenas uma classe/interface pública por arquivo.

INTERFACES

- Permitem expressar comportamento sem se preocupar com a implementação.

```
interface Voador
{
    void voar(int tempo);
}
```

```
class Ave implements Voador
{
    public void voar(int tempo){...}
    public void comer(){...}
}
```

```
class Avião implements Voador
{
    public void voar(int tempo){...}
    public void abastecer(){...}
}
```

```
class DiscoVoador implements Voador
{
    public void voar(int tempo){...}
    public void piscar(){...}
}
```

INTERFACES

- Permitem expressar comportamento sem se preocupar com a implementação.

```
interface Voador
{
    void voar(int tempo);
}
```

```
class Ave implements Voador
{
    public void voar(int tempo){...}
    public void comer(){...}
}
```

Todas as classes que implementam a interface Voador precisam prover um método voar

```
class Avião implements Voador
{
    public void voar(int tempo){...}
    public void abastecer(){...}
}
```

```
class DiscoVoador implements Voador
{
    public void voar(int tempo){...}
    public void piscar(){...}
}
```

INTERFACES

- Clientes usam a interface sem saber qual a classe que a implementa.

```
class Testador
{
    public void testar(Voador v)
    {
        for (int i=0; i<5; i++)
            v.voar(10 * i);
    }
}
```

```
Ave a = new Ave();
Avião v = new Avião();
DiscoVoador d = new DiscoVoador();
...
Testador t = new Testador();
...
t.testar(a);
t.testar(v);
t.testar(d);
```

INTERFACES

- Clientes usam a interface sem saber qual a classe que a implementa.

```
class Testador
{
    public void testar(Voador v)
    {
        for (int i=0; i<5; i++)
            v.voar(10 * i);
    }
}
```

O método `testar` quer algum objeto que implemente o comportamento de um `Voador`, não importa qual

```
Ave a = new Ave();
Avião v = new Avião();
DiscoVoador d = new DiscoVoador();
...
Testador t = new Testador();
...
t.testar(a);
t.testar(v);
t.testar(d);
```

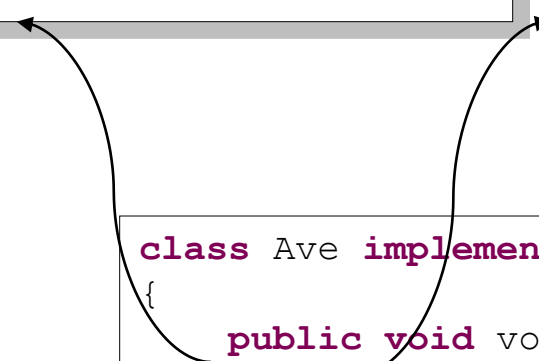
INTERFACES

- Uma classe pode implementar várias interfaces

```
interface Voador
{
    void voar(int tempo);
}
```

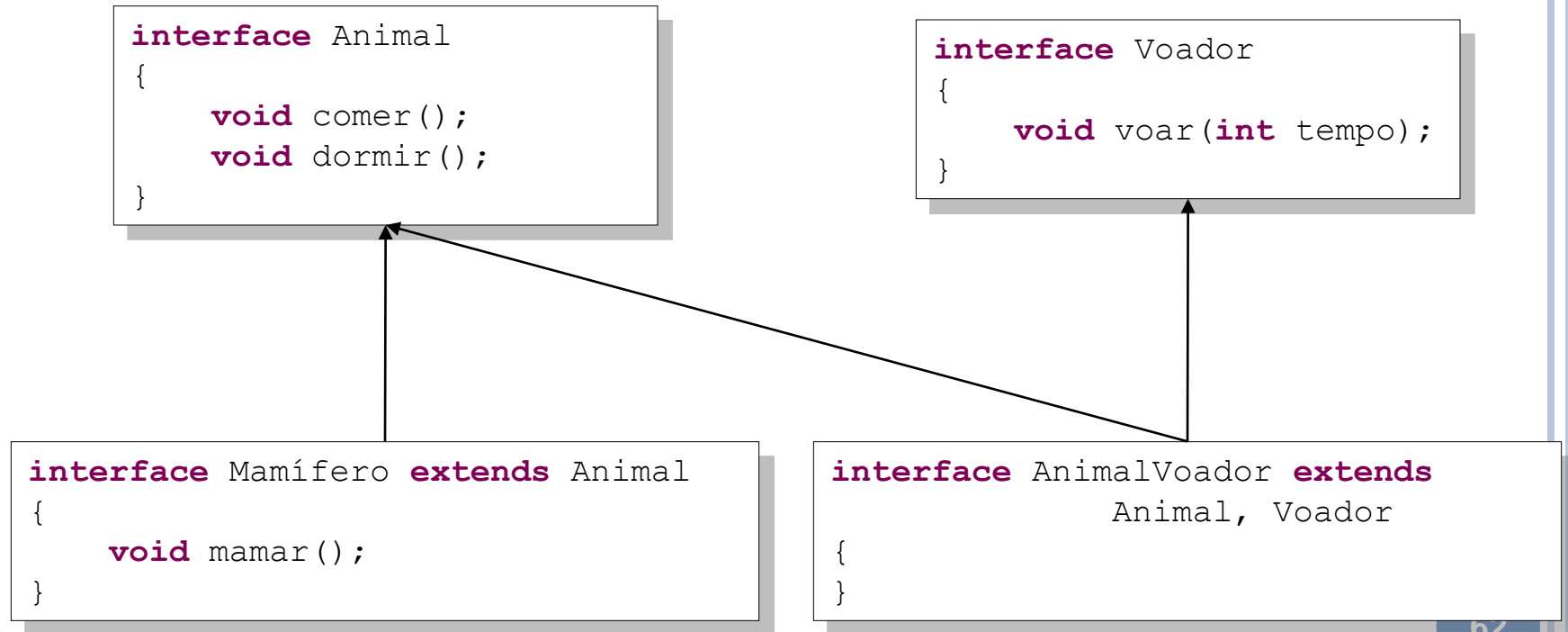
```
interface Animal
{
    void comer();
    void dormir();
}
```

```
class Ave implements Voador, Animal
{
    public void voar(int tempo){...}
    public void comer(){...}
    public void dormir(){...}
}
```



INTERFACES

- Interfaces podem herdar outras interfaces



INSTALAÇÃO DE AMBIÊNTE DE PROGRAMAÇÃO

- Máquina virtual

- Java da Sun - <http://java.sun.com/>
 - Java SE - <http://java.sun.com/j2se/>
 - JDK (Java Development Kit)

- Ambiente de Desenvolvimento

- Eclipse - <http://www.eclipse.org/>
 - Eclipse SDK 3.6 (ou superior)
- Netbeans - <http://www.netbeans.org/>
 - Netbeans IDE 6.5 (ou superior)

