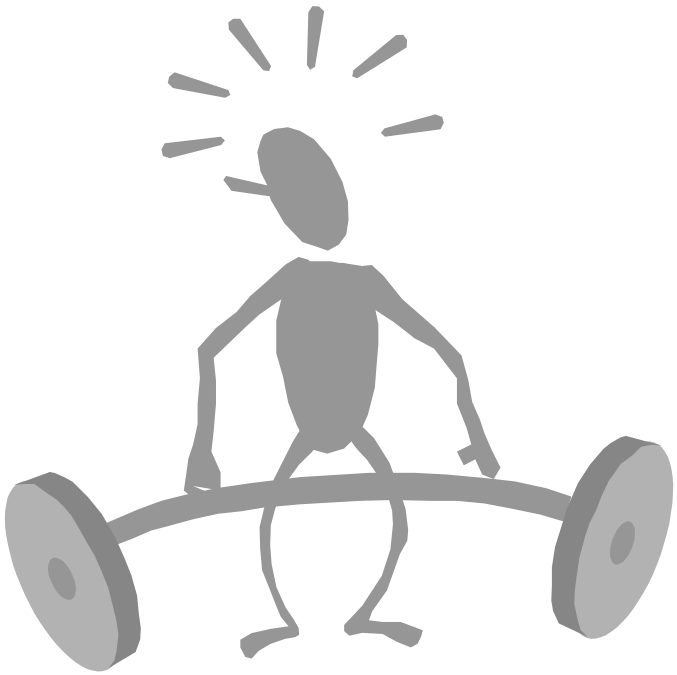
Aula 02 - Organização da Linguagem Java

Programação Orientada a Objetos

Prof. Marcelo Nascimento Costa, MSc
marcelo.nascimento@uva.br

Métodos em Java



- Métodos
  - Organização
  - Modificadores
  - Tipo de retorno
  - Nome
  - Parâmetros
  - Código de um método

Organização de um Método

- Cabeçalho do Método
  - modificadores de método
  - seu tipo de retorno
  - o nome do método
  - seus parâmetros
- Corpo do Método
  - Declaração de variáveis locais
  - Código do método

```
modificadores  tipo-retorno  nomeMétodo(params) {
    // implementação do método
}
```

Organização de um Método

```
public double volume() {
        return 4.0 / 3.0 * Math.PI * raio * raio * raio;
}
```

- Assinatura:
  - public (método acessível para todos)
  - double (método retorna um double)
  - volume (nome do método)
  - () (sem parâmetros)

- Usando o Método
  - Esfera e = new Esfera();
  - double d = e.volume();

Modificadores de Visibilidade

- Indicam os objetos que podem acessar o método
  - O modificador ***public*** indica que um método é acessível por outros objetos
  - O modificador ***private*** indica que um método é acessível apenas pelo próprio objeto
  - O modificador ***protected*** indica que um método é acessível pelo objeto e por descendentes de sua classe (classes de qualquer pacote), ou por classes do mesmo pacote
  - Default: classes do mesmo pacote podem acessar o método

Modificador de Escopo

- Indica a quem pertence o método
  - Método de instância: manipula atributos de instância e classe
  - Método de classe: manipula somente atributos de classe
  - O modificador ***static*** indica que um método pertence à classe
  - Por default, os métodos pertencem às instâncias

Tipo de Retorno de Método

- Pode ser qualquer tipo válido da linguagem
  - Primitivos, vetores ou classes

- Pode ser do tipo ***void***
  - Tipo especial da linguagem Java
  - Utilizado nos retornos de métodos, indicando que o método não retorna nada
  - Não pode ser utilizado na declaração de variáveis nem de parâmetros

Código de Método

- O código de um método Java
  - Muito similar ao código de um método em C++
  - Apresentado logo após o cabeçalho do método
  - Delimitado por um par de chaves

- Principais diferenças:
  - Não existem ponteiros
  - Todos os objetos são dinâmicos
  - Objetos não precisam ser liberados (“*garbage collection*”)

Chamada de Métodos de Outros Objetos

- Na chamada de métodos de outros objetos, devemos especificar o nome do objeto dono do método
- O nome do método deve ser separado do nome do objeto pelo operador ponto (“.”)
- Parâmetros são especificados como em uma chamada de método do próprio objeto

Métodos de Classe

- Algumas operações estão definidas com escopo de classe
- Não precisamos instanciar objetos para enviar mensagem para a classe.
- Exemplo:
  - Classe: **Integer**
  - Método: **parseInt**

```
int x = Integer.parseInt("100");
```

Sobrecarga de operação

```
class Esfera
{
    private static int numeroEsferas;
    private double raio;
    private double xCenter, yCenter, zCenter;

    public double volume() {
        return 4.0 / 3.0 * Math.PI * raio * raio * raio;
    }

    public double volume(double valorPI) {
        return 4.0 / 3.0 * valorPI * Math.pow(raio, 3);
    }
}
```

Sobrecarga de operação

- Invocação é decidida pelo compilador em função do tipo dos parâmetros

```
Esfera e = new Esfera();
double vol;


vol = e.volume();
vol = e.volume(3.14);
```

Operadores Aritméticos

Op	Objetivo	Exemplo	Restrições
+	Soma	a + b	
-	Subtração	a - b	
*	Produto	a * b	
/	Divisão	a / b	
%	Resto da divisão	a % b	**a**,**b** inteiros
++	Incremento	++a	**a** inteiro
--	Decremento	--a	**a** inteiro

Operadores de Atribuição

Op	Objetivo	Exemplo	Restrições
=	Atribuição	a = b	
+=	Atrib. Soma	a += b	
-=	Atrib. Subt	a -= b	
*=	Atrib. Produto	a *= b	
/=	Atrib. Divisão	a /= b	
%=	Atrib. Resto	a %= b	**a**, **b** inteiros

Operadores Lógicos e Relacionais

Op	Objetivo	Exemplo	Restrições
>	Maior	a > b	
<	Menor	a < b	
>=	Maior ou Igual	a >= b	
<=	Menor ou Igual	a <= b	
==	Igual	a == b	
!=	Diferente	a != b	
&&	E lógico	a && b	**a**, **b** booleanos
||	Ou lógico	a || b	**a**, **b** booleanos
!	Not lógico	!a	**a** booleano

Categorias de Expressões

- Expressões de Cálculo
  - Utilizam os operadores aritméticos e de atribuição
  - Efetuam cálculos algébricos
  - Normalmente resultam em um valor numérico

- Expressões de Controle
  - Utilizam os operadores lógicos e relacionais
  - Definem condições, normalmente utilizadas para o controle do fluxo de execução
  - Resultam sempre em um valor booleano (true / false)

Analista de Sistemas - 2009 – Tribunal de Justiça/Paraná

12 - Com base no código abaixo, informe qual o valor de y que será escrito quando o programa for executado.

```
public class Compara
{
    public static void main(String args[])
    {
    Int x = 10, y=0;
      if(x < 10)
        y = 1;
      if(x>=10)
        y = 2;
    System.out.println("O valor de y é " + y);
    }
}
```

a) 0

b) 1

c) 2

d) 3

Analista de Sistemas Eletronorte – 2005 – NCE/UFRJ

13 - Sobre a compilação e/ou execução do programa abaixo, podemos afirmar que:

```
class MinhaClasse {
    void meuMetodo(int i) {
            System.out.println(“versao int”);
     }
     void meuMetodo(double i) {
         System.out.println(“versao double”);
   }
public static void main(String args[]) {
     MinhaClasse obj = new MinhaClasse();
     double x = 3;
     obj.meuMetodo(x);
   }
}
```

SUSEP – 2002 - ESAF

14 - Analise as seguintes afirmações relativas à Programação Orientada a Objetos:

I. Em um programa orientado a objetos, as instâncias de uma classe armazenam os mesmos tipos de informações e apresentam o mesmo comportamento.

II. Em uma aplicação orientada a objetos, podem existir múltiplas instâncias de uma mesma classe.

III. Em um programa orientado a objetos, as instâncias definem os serviços que podem ser solicitados aos métodos.

IV. Em um programa orientado a objetos, o método construtor não pode ser executado quando a classe à qual pertence é executada.

Indique a opção que contenha todas as afirmações verdadeiras.

a) I e II

b) II e III

c) III e IV

d) I e III

e) II e IV

TCU – 2002 - ESAF

15 - Analisando o seguinte trecho de código em Java,

```
final class Pagamento
{
. . .
}
```

é correto afirmar que

a) não será possível criar subclasses da classe *Pagamento.*

b) a classe *Pagamento não poderá conter métodos.*

c) falta a palavra-chave *extends na declaração* da classe *Pagamento.*

d) a classe *Pagamento é uma classe abstrata.*

e) a classe *Pagamento é derivada da classe* final.

Blocos de Comandos

- O controle de fluxo trabalha com blocos de comandos
  - Um bloco de comandos pode conter um ou mais comandos
  - Blocos com mais de um comando são delimitados por chaves
  - Blocos com um comando podem ser delimitador por chaves
- Possíveis comandos
  - Expressões
  - Outros controles de fluxo
  - Chamadas de métodos

Condições (If-Else)

```
if (x > 0)
{
        x = x + 10;
        System.out.println (“x foi acrescido de 10”);
}


if (y < 10  &&  y > 0)
        System.out.println (“Y está entre 0 e 10”);
else
        System.out.println (“Y fora do intervalo 0-10”);
```

Condições (Switch)

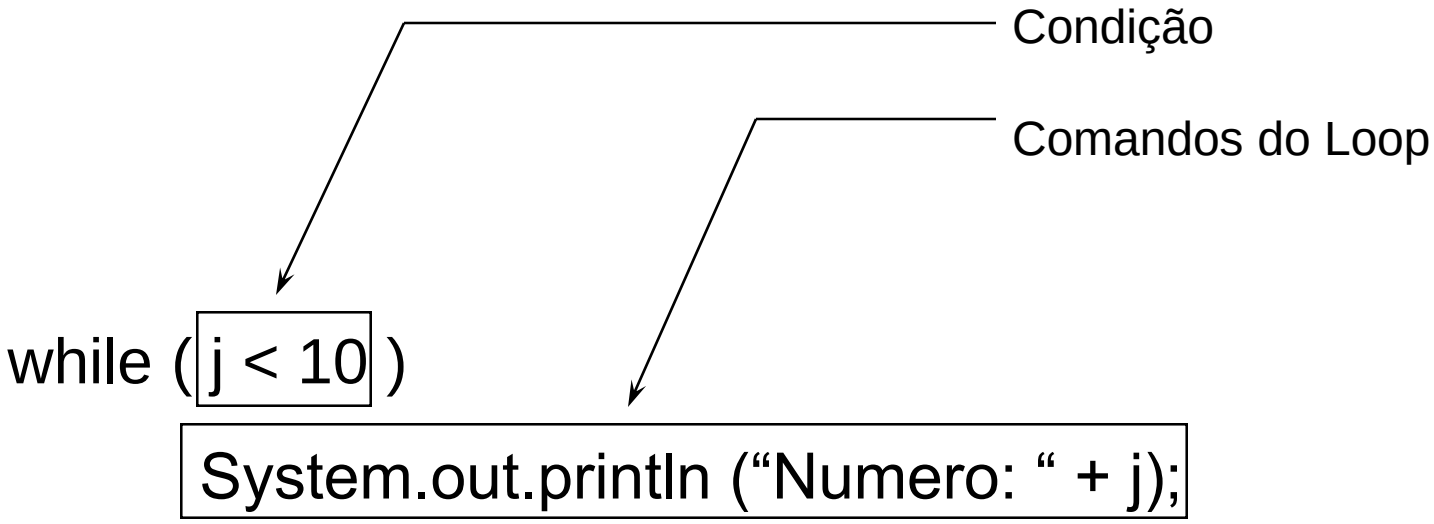
```
switch (month)
{
    case 1:
    case 3:
    case 5:
    case 7:
    case 8:
    case 10:
    case 12:
      numDays = 31;
      break;

    case 4:
    case 6:
.
.
.
    case 9:
    case 11:
      numDays = 30;
      break;

case 2:
      if (AnoBissexto())
        numDays = 29;
      else
        numDays = 28;
      break;
}
```

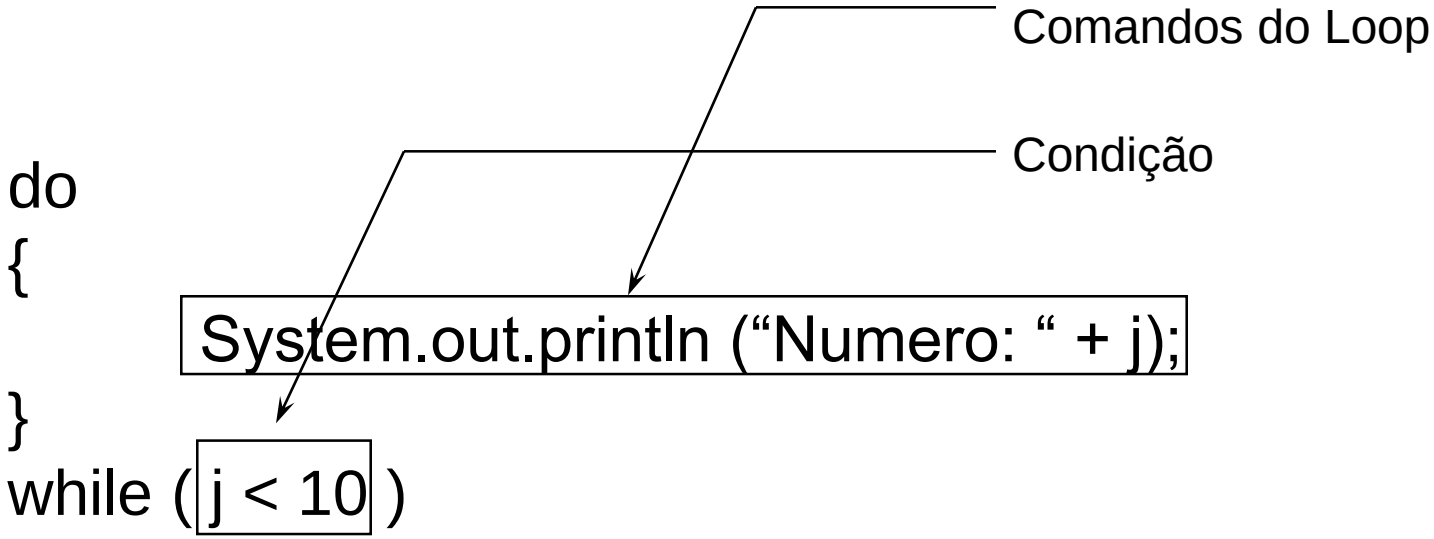
Repetições (While)

- Executa um bloco de comandos enquanto uma condição for verdadeira. A condição é testada no início do loop.



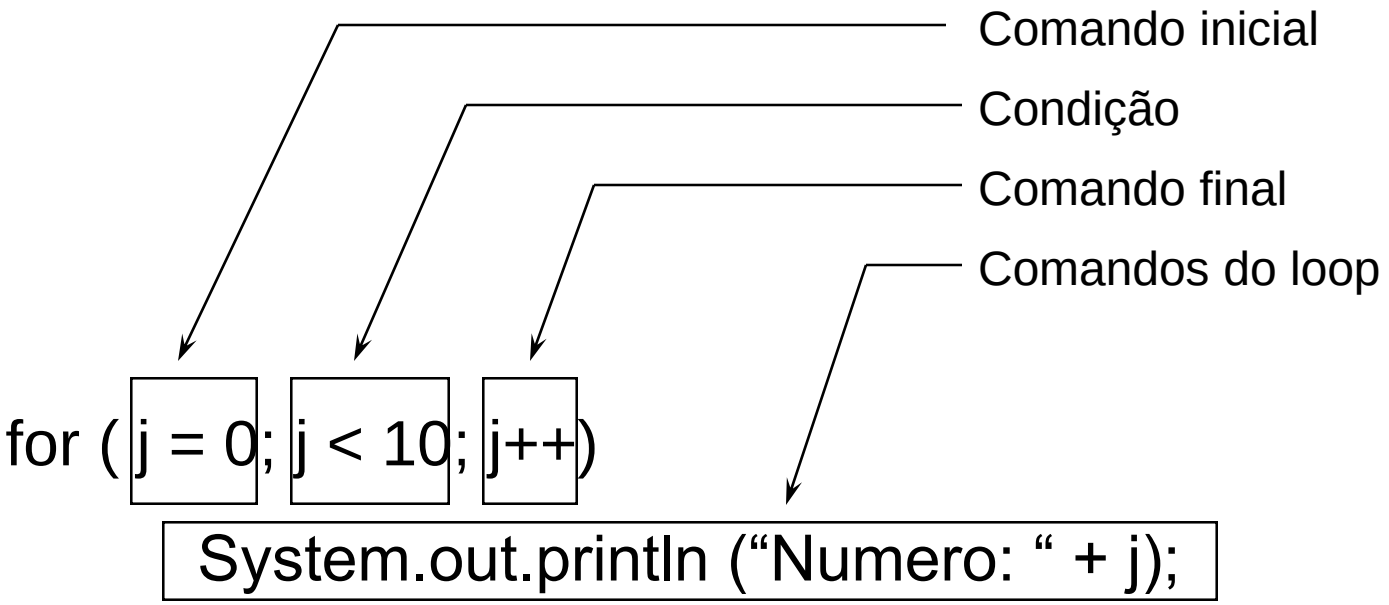
Repetições (Do-While)

- Executa um bloco de comandos enquanto uma condição for verdadeira. A condição é testada no fim do loop.



Repetições (For)

- Executa um bloco de comandos enquanto uma condição for verdadeira. A condição é testada no início do loop.



Retorno do Método

- Encerra a rotina, indicando seu valor de retorno
  - Se a rotina possui algum tipo de retorno, o valor de retorno é indicado após o ***return***
  - Se a rotina possui tipo de retorno ***void***, o ***return*** não possui parâmetros
  - Nenhum comando da rotina é executado após o ***return***

```
return ( 10.0 );
```

Valor de retorno

```
return;
```

Rotina com retorno ***void***

Métodos (getters)

```
public static int getNumEsferas() {
    return numEsferas;
}
public double[] getCenter() {
    return center;
}
public String getNome() {
    return nome;
}
public double getRaio() {
    return raio;
}
```

Métodos (setters)

```
public static void setNumEsferas(int numEsferas) {
    Esfera.numEsferas = numEsferas;
}
public void setCenter(double[] center) {
    this.center = center;
}
public void setNome(String nome) {
    this.nome = nome;
}
public void setRaio(double raio){
    this.raio = raio;
}
```

Exercícios

1. Transforme um número Racional (formado por numerador e denominador) para um número Real. Antes de dividir, verifique se o denominador é diferente de zero. Emita uma mensagem de alerta ao usuário se for zero.
2. Um banco concede empréstimo a seus clientes no valor máximo de 30% do valor do seu salário liquido. Receba o valor do salário bruto, o valor dos descontos e o valor do possível empréstimo de um cliente, em seguida avise se ele poderá ou não fazer o empréstimo.

Exercícios

3. Implementar o algoritmo da UVA para aprovação do aluno considerando as 2 primeiras avaliações do aluno e a necessidade de fazer a avaliação A3. Caso necessário, ler A3 e verificar se o aluno passou.

4. Verifique a validade de uma data de aniversário (solicite apenas o número do dia e do mês). Além de falar se a data está ok, informe também o nome do mês. Indicar também a idade da pessoa.

Dica: meses com 30 dias: abril, junho, setembro e novembro.

Exercícios

5. Valide um horário composto de horas, minutos e segundos.

6. Exiba todos os números ímpares existentes entre dois números informados pelo usuário.

Dica: use o operador % para calcular o resto da divisão entre dois números.

7. Implementar programa que realize uma das 4 operações (+), (-), (/) e (*) entre duas variáveis, através da escolha do usuário

Analista de Tecnologia da Informação 2009 UFF COSEAC

23 - Analise o seguinte trecho de código na linguagem Java:

```
Int i1 = 5, i2 = 6;
String s1 = (i1>i2) ? “x” : “y”;
```

Após rodar o trecho de código acima apresentado, o valor atribuído a s1 será:

(A) x;

(B) false;

(C) 5;

(D) 6;

(E) y

ANALISTA DE SISTEMAS JÚNIOR – COPEL – 2010 – PUC/PR

29 - A linguagem *Java trabalha com dois tipos de* variáveis: tipos primitivos e objetos. Assinale a alternativa **CORRETA que apresenta os tipos** primitivos da linguagem *Java:*

A) *byte*, *short*, *int*, *long*, *float*, *double*, *boolean*, *String*.

B) *Byte*, *Short*, *Int*, *Long*, *Float*, *Double*, *Boolean*, *String*.

C) *int*, *unsigned int*, *float*, *double*, *boolean*, *char*.

D) *int*, *real*, *boolean e string*.

E) *byte*, *short*, *int*, *long*, *float*, *double*, *boolean*, *char*.

Arrays

- Conceitos
- Declaração
- Construção
- Inicialização

Arrays

- Conceitos
  - Um array é um objeto em Java que armazena múltiplas variáveis de um mesmo tipo
  - Arrays podem armazenar tipos primitivos ou referências para objetos
  - Todo array é um objeto. Mesmo um array de primitivos é um objeto na memória

Arrays

- Instanciando um array
  - Instanciar um array significa criar um novo objeto array na memória
  - Usa-se a palavra chave new
  - O tamanho do array (número de elementos) deve ser especificado neste momento (a JVM precisa saber quanto espaço alocar para o novo objeto)
  - Exemplo:

```
int[] teste;              // Declara um array de ints
teste = new int[4];       // Constroi o array e
                         // atribui a variavel teste
```

Arrays

- Efeitos da instanciação de arrays
  - Quantos objetos são criados na memória depois que as linhas de código a seguir executarem?

```
int[] teste = new int[10];

Thread[] threads = new Thread[5];
```

Arrays

- Arrays multidimensionais

```
int[][] myArray = new int[3][];        // declara e constroi um array
                                       // de 2 dimensoes do tipo int

myArray[0] = new int[2];
myArray[0][0] = 6;
myArray[0][1] = 7;
myArray[1] = new int[3];
myArray[1][0] = 9;
myArray[1][1] = 8;
myArray[1][2] = 5;
```

- Inicialização
  - Inicializar o array significa atribuir valor aos seus elementos
  - Array de primitivos: preencher cada elemento primitivo com o valor adequado

```
int[] x = new int[5];
x[4] = 2; // OK
x[5] = 3; // Runtime exception: Não existe índice 5
          // (ArrayIndexOutOfBoundsException)
```

  - Array de objetos: atribuir a cada elemento uma referência para um objeto do tipo (ou subtipo) do array

```
Animal [] pets = new Animal[3];
pets[0] = new Animal();
pets[1] = new Animal();
pets[2] = new Animal();
```

- Inicializando o array em um loop

```
Cao[] meusCaes = new Cao[6];
for (int x = 0; x < meusCaes.length; x++) {
    meusCaes[x] = new Cao(); // atribui um novo Cao a posicao com
                             // indice x
}
```

- Construindo e Inicializando em um único comando (tipo 2)
  - Array anônimo

```
int[] teste;
teste = new int[] {4,7,2}; // nunca especificar o tamanho!
```

  - Exemplo de uso: Passar um array para um método

```
public class Meuteste {
  void recebeUmArray(int []  umArray) {
    // use o parametro array
  }
  public static void main (String [] args) {
    Meuteste o = new Meuteste();
    o.recebeUmArray(new int[] {7,7,8,2,5}); // cria um array
                                            // e passa como parametro
  }}
```

- Atribuindo valores aos elementos do array
  - Array de primitivos

```
int[] lista = new int[5];
byte b = 4;
char c = 'c';
short s = 7;
lista[0] = b;  // OK, byte eh menor que int
lista[1] = c;  // OK, char eh menor que int
lista[2] = s;  // OK, short eh menor que int
```

Arrays

- Atribuindo valores aos elementos do array
  - Array de objetos: é permitido armazenar objetos de qualquer subclasse da classe declarada do array

```
class Car {}
class Subaru extends Car {}
class Ferrari extends Car {}
...
Car [] myCars = {new Subaru(), new Car(), new Ferrari()};
```

  - Segue as mesmas regras da atribuição de variáveis de referência
  - Qualquer objeto que passe no teste “IS-A” executado com a classe declarada do array pode ser atribuído a um elemento do array

ANALISTA DE SISTEMAS JÚNIOR – COPEL – 2010 – PUC/PR

Analise o trecho de código escrito em *Java abaixo:* Assinale a alternativa **CORRETA:**

```
1: public class X {
2:    public static void main(String args[]) {
3:       int [] valor = new int[100];
4:       for(int i=0;i<=valor.length;i++)
5:          valor[i] = i * valor.length;
6:    }
7: }
```

A) O programa não compilará.

B) O programa rodará normalmente.

C) Será lançada uma exceção *IllegalArgumentException em tempo de* execução.

D) Será lançada uma exceção *ArrayIndexOutOfBoundsException em tempo de* execução.

E) Será lançada uma exceção *IllegalStateException* em tempo de execução.

ANALISTA DE SISTEMAS JÚNIOR – COPEL – 2010 – PUC/PR

A API de coleções da linguagem *Java provê um* conjunto de interfaces, implementações e utilitários para manipulação, pesquisa e ordenação de coleções de objetos. Analise o trecho de código abaixo e selecione a classe que implementa a *interface List e apresenta o melhor desempenho para* as características do programa. A classe escolhida preencherá a lacuna da linha 1 do código-fonte:

```
1: List<String> lista = new __________ <String>();
2: Random r = new Random();
3: for(int i=0;i<100000;i++)
4:    lista.add(String.valueOf(r.nextInt()));
5:
6: // Removendo n elementos. Sempre o primeiro da lista
7: for(int i=0;i<1000;i++)
8:    lista.remove(0);
```

A) LinkedList.
B) ArrayList.
C) Vector.
D) HashMap
E) Stack.

Analista de Tecnologia da Informação - Desenvolvimento de Sistemas 2008 CET/SP FAT

Qual é o resultado na execução do seguinte código Java, utilizando os parâmetros 4 e 0 ?

```
public void divide(int a, int b)
{
try {
int c = a / b;
} catch (Exception e) {
System.out.print("Exception ");
} finally {
System.out.println("Finally");
}}
```

(A) Impressão das palavras: *Exception Finally*

(B) Impressão da palavra: *Finally*

(C) Impressão da palavra: *Exception*

(D) Não imprime nada.

Wrappers

- Servem basicamente a dois propósitos:
  - Permitir que tipos primitivos possam ser incluídos em operações exclusivas de objetos, como ser adicionado a uma coleção ou ser retornado de um método que retorna um objeto
  - Fornecer uma gama de funções utilitárias para os tipos primitivos

Wrappers

TABLE 3-2 Wrapper Classes and Their Constructor Arguments

Primitive	Wrapper Class	Constructor Arguments
`boolean`	`Boolean`	`boolean` or `String`
`byte`	`Byte`	`byte` or `String`
`char`	`Character`	`char`
`double`	`Double`	`double` or `String`
`float`	`Float`	`float`, `double`, or `String`
`int`	`Integer`	`int` or `String`
`long`	`Long`	`long` or `String`
`short`	`Short`	`short` or `String`

Wrappers

- Como criar um objeto Wrapper

```
Integer i1 = new Integer(42); // recebe um int (primitivo associado)
Integer i2 = new Integer("42"); // recebe uma representacao do primitivo
                                // em forma de String
Integer i3 = new Integer(“tres"); // Runtime Exception:
                                  // NumberFormatException
Integer i2 = Integer.valueOf("101011", 2);  // converte 101011
                                            // para 43 e
                                            // atribui o valor
                                            // 43 ao objeto Integer

Float f1 = new Float(3.14f); // recebe um float (primitivo associado)
Float f2 = new Float("3.14f");
Float f2 = Float.valueOf("3.14f");
```

Wrappers

- Principais métodos
  - `xxxValue()` – obtém o valor do primitivo armazenado dentro do objeto Wrapper

```
Integer i2 = new Integer(42);
int iPrimitivo = i2.intValue();
```

  - `parseXxx()` – retorna um primitivo a partir de uma String que o representa

```
double d4 = Double.parseDouble("3.14");
```

  - `toXxxString()` – permite converter um número na base 10 para uma String contendo a sua representação hexa, binária ou octal

```
String s3 = Integer.toHexString(254);  // converte 254 para hexa
System.out.println("254 igual a " + s3); // "254 igual a fe"
```

Garbage Collection

- Tem como finalidade fazer o gerenciamento de memória de forma automática
- O garbage collector remove da heap os objetos que não são mais utilizados, liberando espaço na memória
- Utiliza-se o método System.gc() para invocar o garbage collector, mas a sua execução não é garantida
  - Essa chamada apenas solicita que o garbage collector seja executado
  - A decisão de quando ele será executado é da JVM
  - O algoritmo utilizado na rotina de garbage collection varia de acordo com a implementação da JVM

Garbage Collection

- Critério para seleção de objetos
  - Um objeto será alvo do garbage collector caso não possa mais ser acessado, ou seja, quando não existirem mais referências para ele

Executar o código abaixo e monitorar a memória

```
public class A {
        private Object[] objs = new Object[50000];
        public static void main(String[] args){
                A a = new A();
                a.fazAlgo();
        }

        public void fazAlgo(){
                for (int i = 0; i < objs.length; i++)
                        objs[i] = new String("i);

                for (int i = 0; i < objs.length; i++) {
                        objs[i] = null;
                        if ((i%100) == 0)
                                System.gc();
                }
        }
}
```

O **novo** comando for

- O novo for: “for-each”, “enhanced for”, “for-in”
- Simplifica a iteração sobre arrays e coleções
- Exemplos

```
int [] a = {1,2,3,4};

for(int x = 0; x < a.length; x++)    // for basico
  System.out.print(a[x]);

for(int n : a)                       // novo for
  System.out.print(n);
```

O **novo** comando for

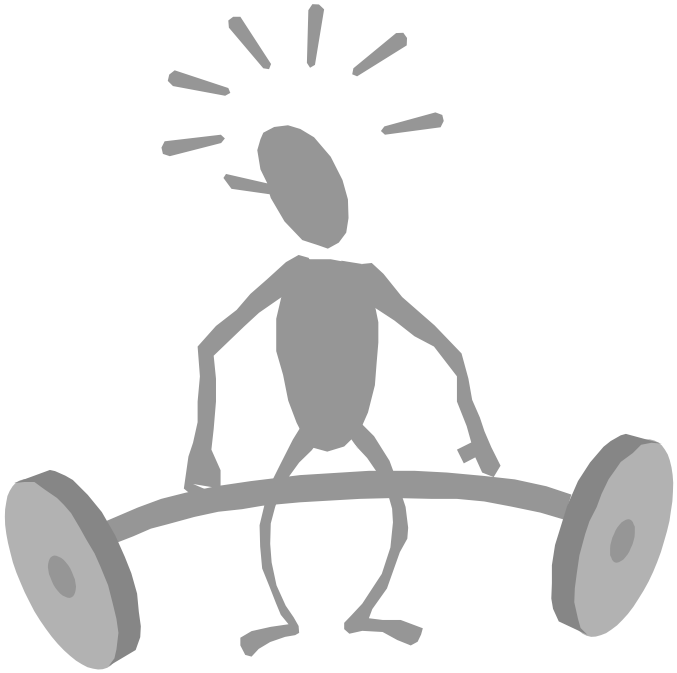
- Sintaxe

```
for(declaracao : expressao)
```

- A expressão é o array ou coleção a qual se deseja percorrer

- A declaração é a variável (escopo de bloco), cujo tipo é compatível com os elementos do array ou coleção. A variável contém o valor do elemento de uma dada iteração

Herança em Java

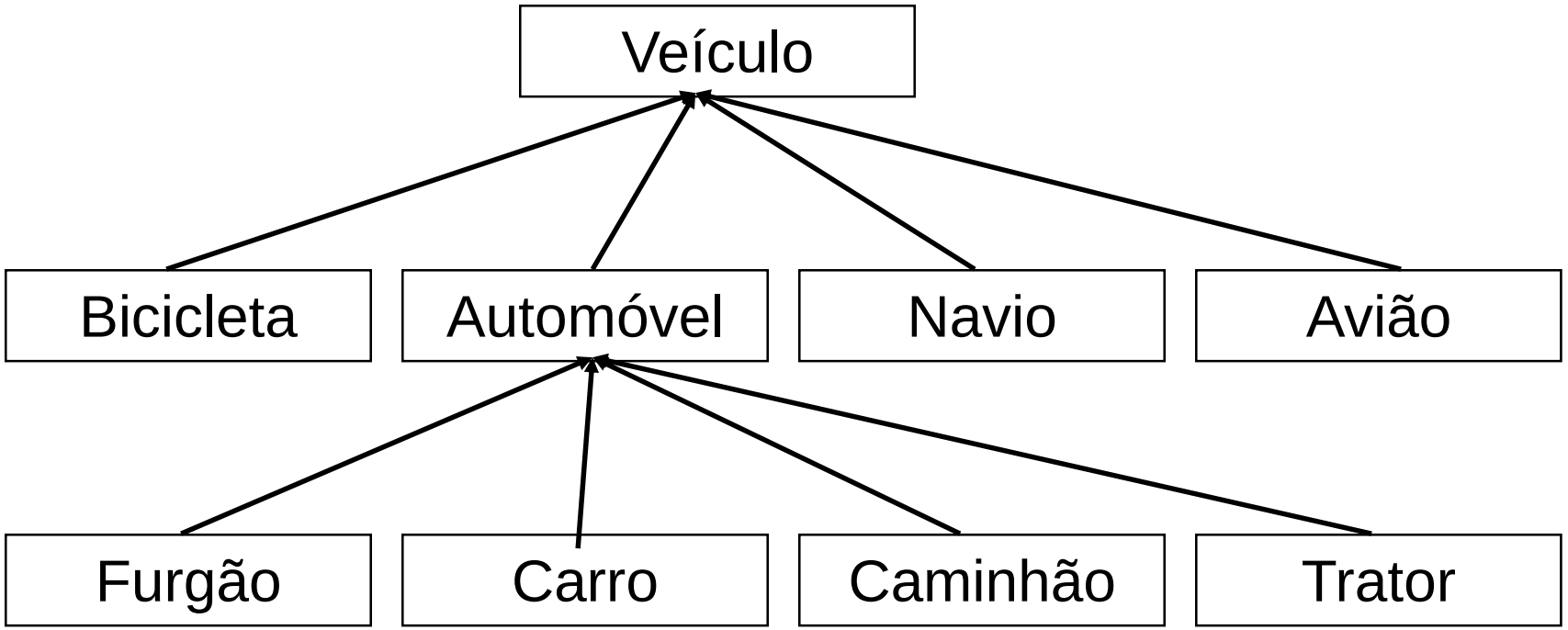


- Tópicos
  - Herança
  - Redefinição de Métodos
  - Herança e Visibilidade

Herança

- Classes são organizadas em estruturas hierárquicas
  - Uma classe pode herdar características e comportamento de outras classes
  - A classe que forneceu os elementos herdados é chamada de **superclasse**
  - A classe herdeira é chamada de **subclasse**
  - A subclasse herda todos os métodos e atributos de suas superclasses
  - A subclasse pode definir novos atributos e métodos específicos

Exemplo de Herança



Tipos de Herança

- Herança Simples x Herança Múltipla
  - Se uma classe herda de apenas uma superclasse, temos uma herança simples
  - Se uma class herda de diversas superclasses, temos uma herança múltipla
  - Java suporta apenas herança simples, mas simula herança múltipla através de interfaces

Herança

- Java suporta herança simples
  - Todas as classes Java possuem uma única superclasse
  - O nome da superclasse é declarado após o nome da classe
  - A palavra reservada ***extends*** é utilizada nesta declaração
  - Se nenhuma superclasse for especificada, o compilador Java assume que a classe herda da superclasse genérica ***Object***

Herança

```
public class Animal {
    private String tipo;
    private int idade;
    public Animal(String tipo, int idade) {
        this.tipo = tipo;
        this.idade = idade;
    }
    public int getIdade() {
        return idade;
    }
    public String toString() {
        return "Isto é um " + tipo;
    }
}
```

Herança

```
public class Cachorro extends Animal {
    private String nome;
    private String raça;

    public Cachorro (String nome, String raça, int idade) {
        super("Cachorro", idade); // chamada do construtor da superclasse

        this.nome = nome;
        this.raça = raça;
    }
}
```

Herança

```
public class MeusAnimais  {
    public static void main(String args[]) {
            Cachorro c1 = new Cachorro ("Rex", "Pastor Alemão", 5);
            Cachorro c2 = new Cachorro ("John", "Poodle", 4);

            System.out.println(c1 + " - idade = " + c1.getIdade());
            System.out.println(c2 + " - idade = " + c2.getIdade());
    }
}
```

Adicionando métodos

```
public class Cachorro extends Animal {

    ...

    public void imprime() {

        System.out.println( "Isto é um Cachorro" + " de nome:  " + nome
        + ", raça: " + raça + " e idade = " + getIdade();

    }

}
```

Adicionando métodos

```
public class MeusAnimais  {
    public static void main(String args[]) {
        Cachorro c1 = new Cachorro ("Rex", "Pastor Alemão", 5);
        Cachorro c2 = new Cachorro ("John", "Poodle", 4);
        c1.imprime();
        c2.imprime();
    }
}
```

Redefinindo métodos na subclasse

```
public class Cachorro extends Animal {

    ...

    public String toString() {

        return "Isto é um Cachorro" + " de nome:  " + nome + " e raça: " +
        raça;

    }

}
```

Redefinindo métodos na subclasse

```
public class MeusAnimais  {
    public static void main(String args[]) {
        Cachorro c1 = new Cachorro ("Rex", "Pastor Alemão", 5);
        Cachorro c2 = new Cachorro ("John", "Poodle", 4);
        System.out.println(c1);
        System.out.println(c2);
    }
}
```

Redefinindo métodos na subclasse

```
public class Cachorro extends Animal {

    ...

    public String toString() {

        return super.toString() + " de nome:  " + nome + " e raça: " +
        raça;

    }

}
```

- Redefinição não pode reduzir a visibilidade do método. Ex: **public** na superclasse e **protected** na subclasse (erro).

Herança e visibilidade

- Subclasses não têm acesso a métodos ou atributos com visibilidade **private.**

- Apenas as subclasses (em qualquer pacote) ou classes do mesmo pacote têm acesso a métodos ou atributos com visibilidade **protected.**

Polimorfismo

- Quando você envia uma mensagem solicitando que uma subclasse execute um método com determinados parâmetros acontece o seguinte:
  - A subclasse checa se ela possui um método com o mesmo nome e com exatamente os mesmos parâmetros.
  - Caso possua, o método é executado.
  - Caso não possua, a classe pai fica responsável por manipular a mensagem, isto é, por procurar em seu domínio por um método com o mesmo nome e com exatamente os mesmos parâmetros.
  - Se o método for encontrado, será executado.

Regra fundamental relativa a herança

- Um método definido em uma subclasse com o mesmo nome e lista de parâmetros de um método definido em uma classe ancestral, esconde o método da classe ancestral da subclasse.
- Por exemplo, o método aumentaSalario da classe Gerente é chamado em vez do método aumentaSalario da classe Empregado quando você envia uma mensagem para um objeto da classe Gerente.

A idéia por trás do polimorfismo..

- Objetos pertencentes a classes diferentes (em uma mesma hierarquia) podem responder diferentemente a mensagens iguais.
- A técnica utilizada para se conseguir que um método polimórfico seja executado se chama “late binding”. Isto significa que o compilador não gera o código para a chamada de um método em tempo de compilação.
- Sempre que um método é aplicado a um objeto o compilador gera o código necessário para calcular que método deve ser chamado.

Exemplo

```
Empregado[] vetEmpregados = new Empregado[2];
vetEmpregados [0] = new Gerente ("Ricardo Silva", 7500,
new GregorianCalendar(1987, 12, 15));
vetEmpregados [1] = new Empregado ("Luis Alberto", 2500,
new GregorianCalendar(1988, 10, 11));

vetEmpregados[0].aumentaSalario(5);
vetEmpregados[1].aumentaSalario(5);
```

Modificador **final**

- Classe com modificador **final**: não pode ser estendida com subclasses.
- Método com modificador **final**: não pode ser redefinido nas subclasses.

Analista de Sistemas Eletronorte – 2005 – NCE/UFRJ

43 - Observe a definição a seguir da classe MinhaClasse:

```
package meuPacote.seuPacote;
public class MinhaClasse {
// código da classe
}
```

A maneira correta de referenciar a classe MinhaClasse de fora do pacote meuPacote.seuPacote é:

A) importe o pacote meuPacote.* e faça referência à classe como: seuPacote.MinhaClasse;

B) simplesmente referencie a classe como: MinhaClasse;

C) refira-se à classe como: meuPacote.seuPacote.MinhaClasse;

D) importe o pacote seuPacote.* e faça referência à classe como: MinhaClasse;

E) importe o pacote java.lang e faça referência à classe como: MinhaClasse.

Analista de Sistemas - 2009 – Tribunal de Justiça/Paraná

44 - Em Java, a palavra-chave que implementa uma relação de herança de classes é:

a) implements.

b) package.

c) inherits.

d) extends.

O método equals()

- Exemplo 1

```
public class TesteEquals0
{
   public static void main(String[] args)
   {
     Pessoa p1 = new Pessoa("123");
     Pessoa p2 = p1;

     System.out.println("p1.equals(p2)? "+ (p1.equals(p2)));
     System.out.println("p1 == p2? "+ (p1 == p2));

   }
}
class Pessoa
{
   private String cpf;
   Pessoa(String sCpf)
   {
     cpf = sCpf;
   }
   public String getCpf()
   {
     return cpf;
   }
}
```

O método equals()

- Exemplo 1 – Resultado

```
p1.equals(p2)? true
p1 == p2? true
```

- O método equals() da classe Object usa o operador == para fazer a comparação

- Pela implementação de Object, dois objetos só vão ser considerados semanticamente equivalentes se as referências “apontarem” para o mesmo objeto!

O método equals()

- Exemplo 2

```
public class TesteEquals1
{
   public static void main(String[] args)
   {
     Pessoa p1 = new Pessoa("123");
     Pessoa p2 = new Pessoa("123");

     System.out.println("p1.equals(p2)? "+ (p1.equals(p2)));
     System.out.println("p1 == p2? "+ (p1 == p2));

   }
}
class Pessoa
{
   private String cpf;
   Pessoa(String sCpf)
   {
     cpf = sCpf;
   }
   public String getCpf()
   {
     return cpf;
   }
}
```

O método equals()

- Exemplo 2 – Resultado

```
p1.equals(p2)? false
p1 == p2? false
```

O método equals()

- Exemplo 3

```
public class TesteEquals
{
   public static void main(String[] args)
   {
     Pessoa p1 = new Pessoa("123");
     Pessoa p2 = new Pessoa("123");
     System.out.println("p1.equals(p2)? "+ (p1.equals(p2)));
     System.out.println("p1 == p2? "+ (p1 == p2));
   }
}
class Pessoa
{
   private String cpf;
   Pessoa(String sCpf) { cpf = sCpf; }
   public String getCpf() { return cpf; }

   public boolean equals(Object o)
   {
     if ((o instanceof Pessoa) && (((Pessoa)o).getCpf().equals(cpf)))
               return true;
     return false;
   }
}
```

O método equals()

- Exemplo 3 – Resultado

```
p1.equals(p2)? true
p1 == p2? false
```

O método equals()

- Sobrescrevendo equals()
  - Utilizar o operador instanceof para ter certeza que a classe do objeto passado por parâmetro é apropriada
  - Comparar os atributos significativos dos objetos

O método equals()

- Contrato de equals()
  - **Reflexivo**: x.equals(x) é verdadeiro
  - **Simétrico**: x.equals(y) é verdadeiro se e somente se y.equals(x) é verdadeiro
  - **Transitivo**: Se x.equals(y) é verdadeiro e y.equals(z) é verdadeiro, então x.equals(z) é verdadeiro
  - **Consistente**: Múltiplas chamadas a x.equals(y) sempre retornarão o mesmo resultado, dado que as informações usadas na comparação não se alterem
  - **Nulo**: Se x é não nulo, então x.equals(null) é falso