Instituto de Computação

Programação Orientada a Objetos com Java

Prof. Lauro Eduardo Kozovits, D.Sc.

Departamento de Computação, UFF

Instituto de Computação

Objetivos

- Apresentar conceitos da Programação Orientada a Objetos usando a LP Java como exemplo
- Por que Java?
  - Apresentar os aspectos modernos da linguagem orientada a objetos Java e seu potencial de uso por ser uma linguagem multiplataforma moderna e usada em mais de 3 bilhões de equipamentos.

Instituto de Computação

Público Alvo

- Alunos dos cursos do Departamento de Computação da UFF e desenvolvedores de aplicações, principalmente de aplicações para celulares Android, web e intranets.

Instituto de Computação

Pré-requisitos

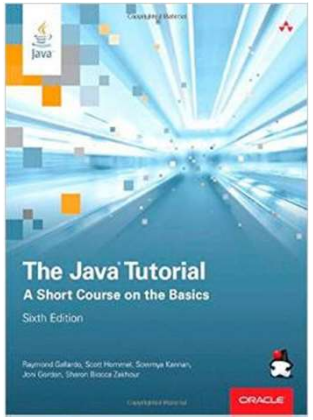
- Noções de alguma linguagem de programação

Instituto de Computação

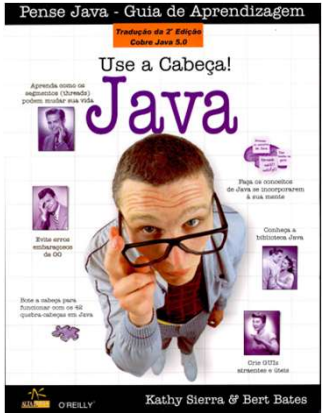
Ementa da disciplina

- Fundamentos da programação orientada a objetos
- Orientação a objetos
- Elementos da linguagem Java
- Relacionamento entre objetos
- Herança e polimorfismo
- Classes abstratas e interfaces
- Exceções
- Threads

Bibliografia do curso



http://docs.oracle.com/javase/tutorial



H. M. Deitel, P. J. Deitel, Java Como Programar, 6ª Edição, Bookman, 2006.

Índice

Módulos
M 1 – Introdução a linguagem Java
M 2 – Sintaxe básica: Tipos de Dados, Controle de Fluxo, Arrays e Strings
M 3 – Orientação a Objetos: Conceitos e uso de Classes e Objetos
M 4 – Herança
M 5 – Uso de Math, Stack e modificador static
M 6 – Classes Abstratas e Polimorfismo
M 7 – Interfaces
M 8 – Tratamento de Exceções
M 9 – Uso de Threads
M 10 – Interfaces Gráficas com o Usuário
M 11 – Manipulação de Arquivos
M 12 – Acesso a Bancos de Dados via JDBC
M 13 – Servlets
M 14 – Java Server Pages (JSP)

Obs: módulos na cor preta fazem parte do conteúdo de POO

Instituto de Computação

MÓDULO 1

Introdução à linguagem Java

Origem

- Desenvolvida em 1991 na Sun Microsystems, no projeto Green, para criação de software para controle de utensílios domésticos inteligentes e interativos
- James Gosling aborreceu-se com C++, criou uma nova linguagem e ao ver uma árvore na sua janela, batizou a linguagem como OAK
- A linguagem era pequena
- A linguagem era segura
- A linguagem era portável
- A linguagem era ...

Origem

- …perfeita para uso na WEB

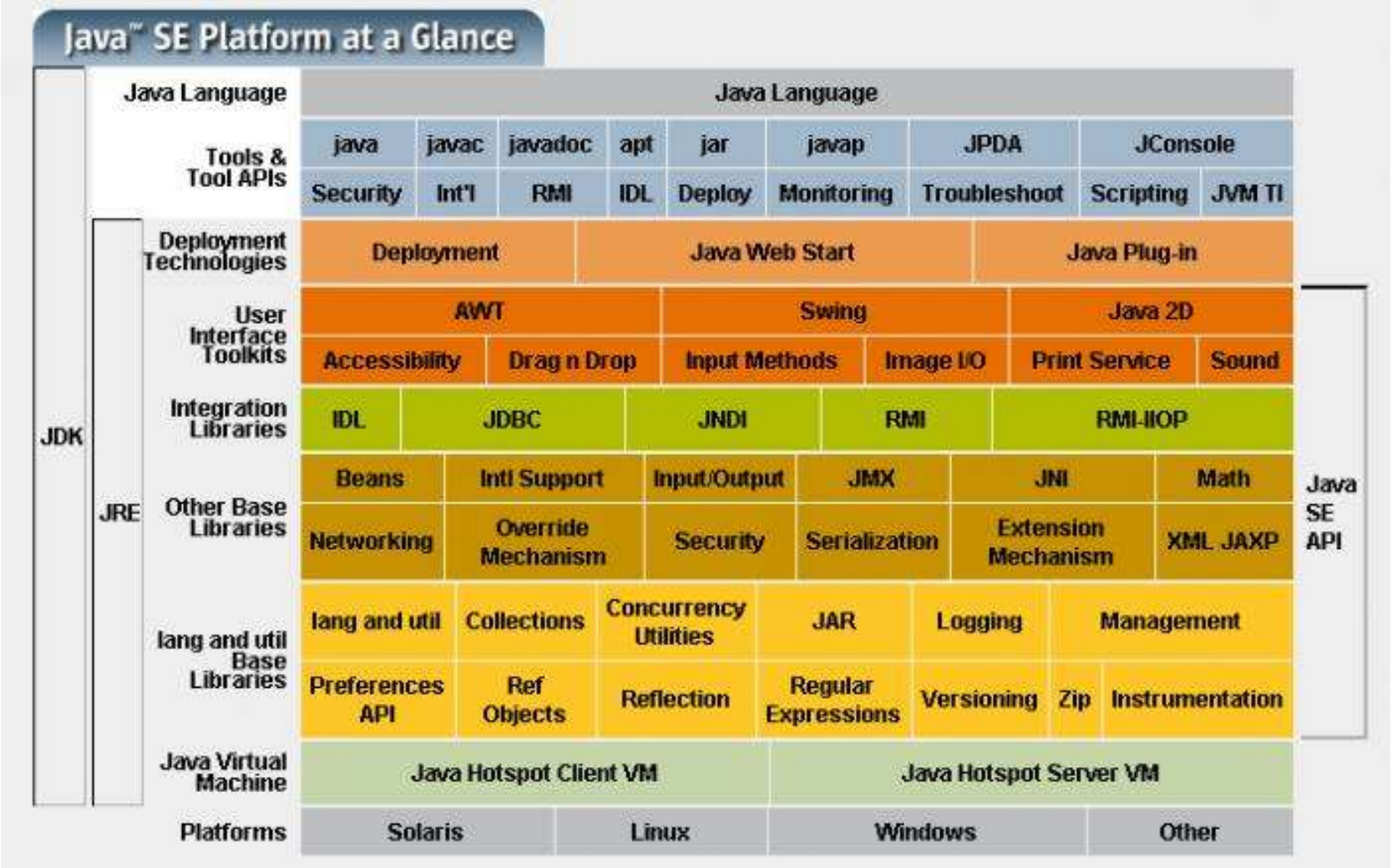
- Para evitar o engavetamento do projeto, seus programadores mudaram o nome para JAVA e a levaram na Netscape.
- A SUN liberou os fontes e estimulou a criação da subsidiária Javasoft
- O apoio de Marc Andreesen da Netscape foi imediato:
- *“A linguagem Java representa uma tremenda oportunidade para todos nós!”*
- ...enquanto isso na M$ um porta voz anunciou:
- *“Java é uma linguagem moderadamente interessante...*”
- começava a guerra Microsoft e Java iniciada com a batalha ActiveX versus JAVA applets

Instituto de Computação

O que pode ser feito com Java?

- Com Java podemos escrever:
- Programas para Internet/intranets
- Programas para uso local (desktop)
- Applets (atualmente em desuso)
- Midlets (programas para celulares, também em desuso)
- E muito mais...

Arquitetura Java



Instituto de Computação

Características da linguagem

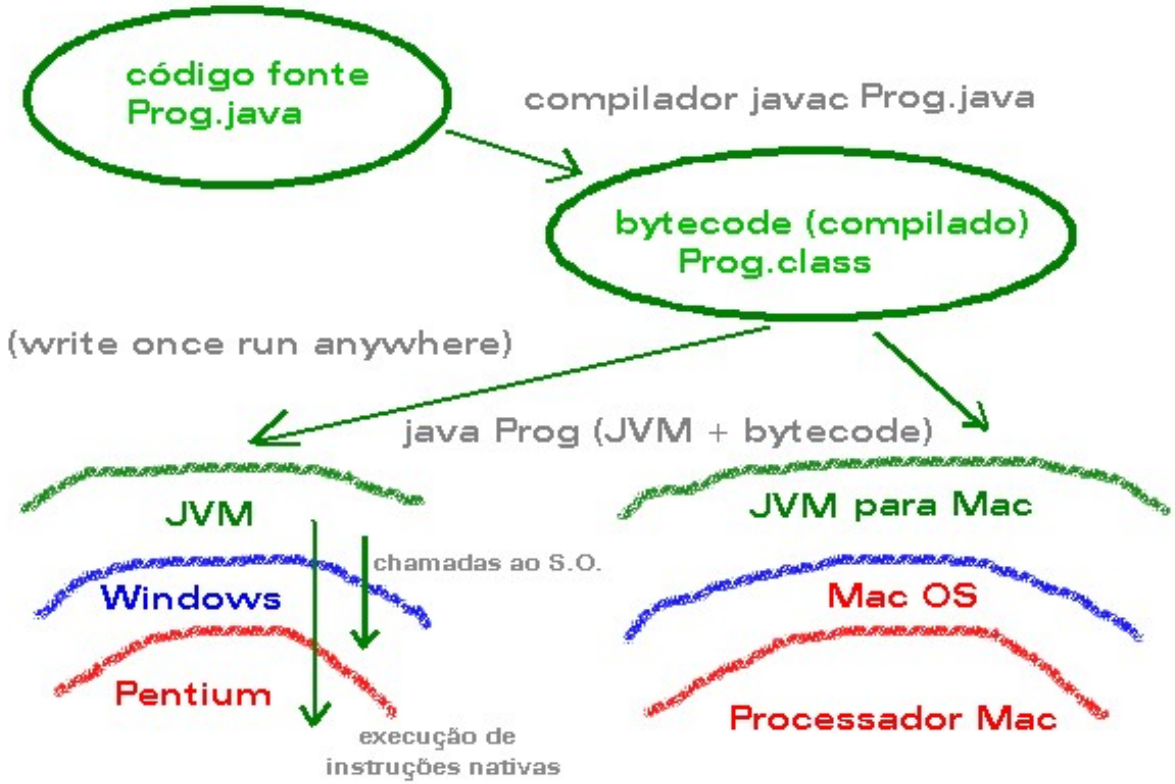
- Orientada a objeto
- Independente de plataforma
- Robusta, segura, multithreaded, suporta objetos distribuídos, garbage collection, etc
- “write once run anywhere”

Instituto de Computação

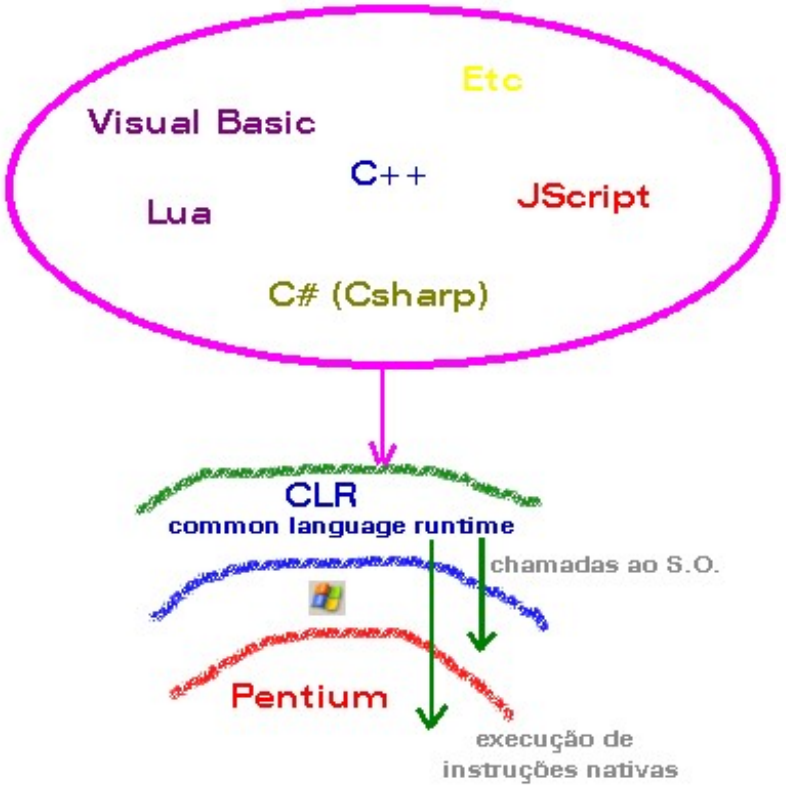
Java Virtual Machine

- É o “microprocessador” Java, ou seja um software que se permite a execução do programa Java compilado numa dada plataforma. A JVM verifica ainda a integridade do programa e a existência de todas as classes necessárias a sua execução.
- Um software compilado em Linux com Java 14 irá rodar sem modificações no Windows (ou Mac) com a JVM 14 e vice-versa.

Um código fonte, muitas plataformas



...comparado ao .NET



Executando um programa Java no ambiente Apache NetBeans IDE 12.0

Assista no Canal ActiveScienceLK

https://www.youtube.com/watch?v=0hlgvvfxHmc

Instituto de Computação

MÓDULO 2

Sintaxe Básica:

Tipos de Dados, Controle de Fluxo, Arrays e Strings

Instituto de Computação

Em Java tudo são classes e objetos...

...com exceção dos 8 tipos primitivos

- byte, short, int, long
- float, double
- char
- boolean

Exemplos:

```
boolean flag = false;
int soma = 123;
char letra = 'A';
double salario = 8888.88;
```

Inteiros(em qualquer JVM)

- byte -128 a 127
- short 16 bits -32768 a 32767
- int 32 bits (default)
- long 64 bits

Exemplos:

```
long x; int y;
x = 123L;
y = 0123; // valor octal com o zero no início
y = 0x123; // valor hexadecimal
x = 0xABEL; // valor hexadecimal long (L ao final)
```

Reais

- float 32 bits
- double 64 bits

Exemplos:

float x; double y;

x = 123;

x = 123f;

x = 123.0f;

x = 123.0; // erro, pois a constante real default é double

y = 123.0;

y = x; // posso atribuir um float em um real

Type casting

- Conversão explícita é necessária no caso:

```
long maior = 123L;
int menor;
menor = (int) maior;
```

OBS (o código abaixo funciona):

```
char letra = 65;
```

Atribuição e operadores

```
x = y = z = 123; // atribuição múltipla
c = a % b; // resto da divisão
int c = 123; // declaração com inicialização
```

Atribuição e operadores(cont.)

```
int c = 123; // declaração com inicialização
c = c + 1;
c += 1;
c++;
++c;
```

Instituto de Computação

Operadores relacionais

- > maior que
- >= maior ou igual a
- < menor que
- <= menor ou igual a
- == igual
- != diferente

Instituto de Computação

Operadores lógicos

- || OU
- && E
- ! Negação

Comando de decisão: IF

```
if (condição)
   comando ou bloco de comandos
else
   comando ou bloco de comandos
```

exemplo 1

```
if (soma > 123 && situacao != 4)
   System.out. println ("ok");
else
    System.out. println ("falhou teste");
```

exemplo 2

```
if (soma == 3 && (x>3 || !flag) ){
   System.out. println ("ok");
   soma = x = 0;
} else
    System.out. println ("falhou teste");
```

exemplo 3

```
if (a >= 100 ){
    // com um comando apenas voce pode dispensar as chaves
    System.out. println ("3 digitos");
} else if (a >= 10 ){
    System.out. println ("2 digitos");
} else if (a >= 0 ) {
    System.out. println ("1 digito");
} else
    System.out. println ("numero negativo");
```

Comando de decisão: SWITCH

```
switch (expressão inteira){
     case cte1:
              comando ou bloco
              [break]
    case cte2:
              comando ou bloco
              [break]
     ...
     default:
               comando ou bloco
}
```

Instituto de Computação

exemplo com switch

```
switch (dia){
      case 1:
            System.out.println("segunda");
            break;
      case 2:
            System.out.println("terca");
            break;
...
      case 7:
            System.out.println("domingo");
            break;
      default:
             System.out.println("opcao invalida");
}
```

outro exemplo com switch

```
switch (dia){
        case 1:
        case 2:
        case 3:
        case 4:
        case 5:
                System.out.println("dia util");
                break;
        case 6:
        case 7:
                System.out.println(" dia agradavel");
                break;
        default:
                 System.out.println("opcao invalida");
}
```

Comando de repetição WHILE

while (condição verdadeira)

    comando ou bloco de comandos

Instituto de Computação

exemplo

```
int a = 0;
while (a < 10){  // este é um loop simples
   System.out.println(a);
   a++;
}
/* aproveitamos para mostrar o
comentário
multilinha */
```

repetição com DO ... WHILE

```
do
   comando ou bloco de comandos
while (condição verdadeira) ;
```

Instituto de Computação

exemplo

```
int i = 1, fatorial = 1;
do {
    fatorial *= i;
    i++;
} while (i <= 4);
System.out.println("fatorial de 4 igual a " + fatorial);
```

Repetição com FOR

```
for (inicializacoes;testes;incrementos)
    comando ou bloco de comandos
```

exemplo

```
for(i=0;i<10;i++)
   System.out.println(i);
```

Instituto de Computação

outro exemplo

```
int x = 10;
int y = x;
for( ;x<30 && y > -30; x++, y--) {
   System.out.println("x="+ x+"y="+y);
   if (y > -10) x = 0;
}
```

desvios com BREAK e CONTINUE

```
int x=0;
for (;;x++) {
        if (x < 10)
                continue;
        System.out.println(x);
        if (x >= 15)
                break;
}
```

A Classe String

- String nome = new String(“Juliana”);

  nome é uma referência ao endereço de memória onde está alocada a String “Juliana”

  new é o operador para alocação de memória

  “Juliana” é o parâmetro da função construtora que veremos nos próximos tópicos

- String outroNome = “Eduardo”;

  para comodidade do programador, Java oferece a possibilidade de uso do new implícito com Strings

O texto abaixo compila?
O que será impresso?

```
String nome1 = new String(“Sidharta”);
String nome2 = new String(“Sidharta”);
if (nome1 == nome2)
   System.out.println(“nomes iguais”);
else
   System.out.println(“diferentes”);
```

Instituto de Computação

O que ocorre?

- nome1 e nome2 referenciam endereços de memória em geral diferentes. Assim a comparação de um com outro, em geral, será diferente.
- Entretanto, se o que desejamos é a comparação de conteúdo podemos usar o método equals ou o método compareTo. Ambos foram codificados na classe String.

Instituto de Computação

Exemplo

```
if ( nome1.equals(nome2) )
   System.out.println(“conteúdo igual”);
else
   System.out.println(“conteúdo diferente”);

//OBS:
//equals retorna um valor lógico true ou false
```

Outro exemplo

```
String nome1 = “Ana”;
String nome2 = “Beatriz”;
int resultado = nome1.compareTo(nome2) ;
if (resultado > 0)
    System.out.println(“maior”);
else if (resultado < 0)
    System.out.println(“menor”);
else // igual a zero
    System.out.println(“iguais”);
```

comparação lexicográfica

Mas Atenção:

String nome1 = “Juliana”;

String nome2 = “Juliana”;

Por otimização do compilador nome1 e nome2 podem estar apontando para a mesma região de memória onde está definida a constante “Juliana”. Se for o caso, a comparação simples das referências resultará em verdadeiro.

```
if (nome1 == nome2)
    System.out.println(“verdadeiro”);
```

Arrays

- São classes em Java e não tipos simples
- Exemplo:

```
int[] vetor;
vetor = new int[3]; // aloquei três posições começando sempre de zero
vetor[0] = 10;
vetor[1] = 20;
vetor[2] = 30;
System.out.println(“o tamanho alocado e´ ”+ vetor.length);
```

Arrays

- Admitem o segundo e último tipo de new implícito do Java: - apenas na declaração

```
double[ ] vetorReais = { 0.0, 1.0, 2.0};
// aloquei e inicializei três posições reais
```

- Usando dois tipos de new implícito

```
String[ ] nomes = { “Ana”, “Edu”, “Bia”};
// pense: como seria o código acima sem o new implícito?
```

Instituto de Computação

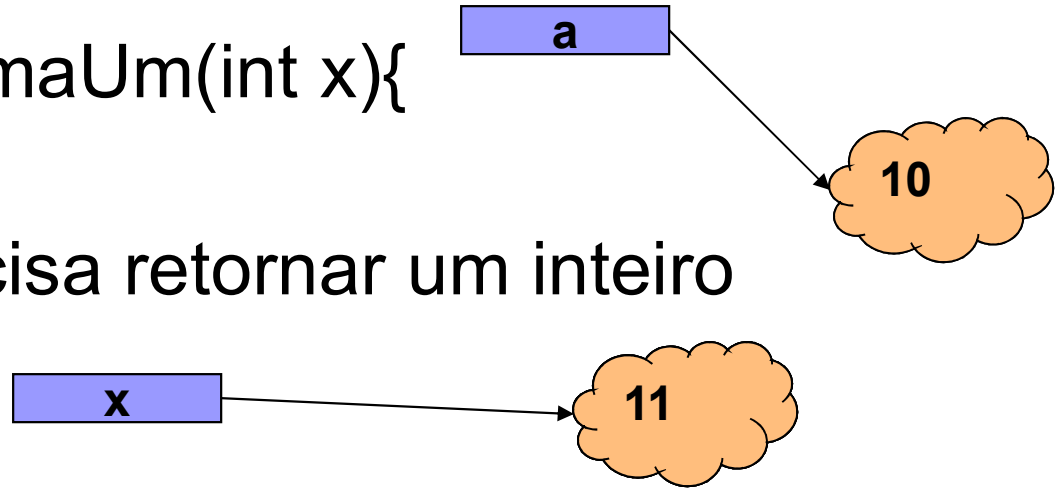
Passagem de Parâmetros

- Tipos simples sempre são passados por valor
- Objetos sempre são passados por referência
- As funções (em O.O. devemos chamar de métodos) podem retornar valores com o uso de return

Um exemplo de passagem por valor...

```
// arquivo Prog.java   (P maiúsculo)
public class Prog{
    public static void main(String[ ] args){
        int a=10;
        int b = somaUm(a);
        System.out.println("a=" + a + " b="+b);
    }
    public static int somaUm(int x){
        x = x+1;
        return x; // precisa retornar um inteiro
    }
}
```

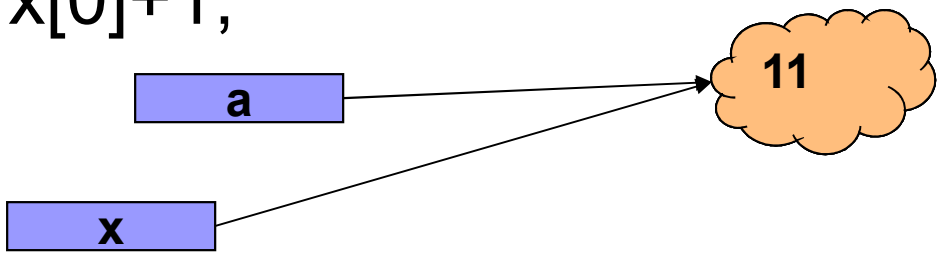
irá imprimir "a=10 b=11", pois a é um int passado por valor



Agora um exemplo de passagem por referência...

```
// arquivo Prog.java   (P maiúsculo)
public class Prog{
    public static void main(String[ ] args){
        int[ ] a = {10};
        somaUm(a);
        System.out.println("a[0]=" + a[0]);
    }
    public static void somaUm(int[ ] x){
        x[0] = x[0]+1;
    }
}
```

irá imprimir "a[0]=11", pois a é um objeto passado por referência



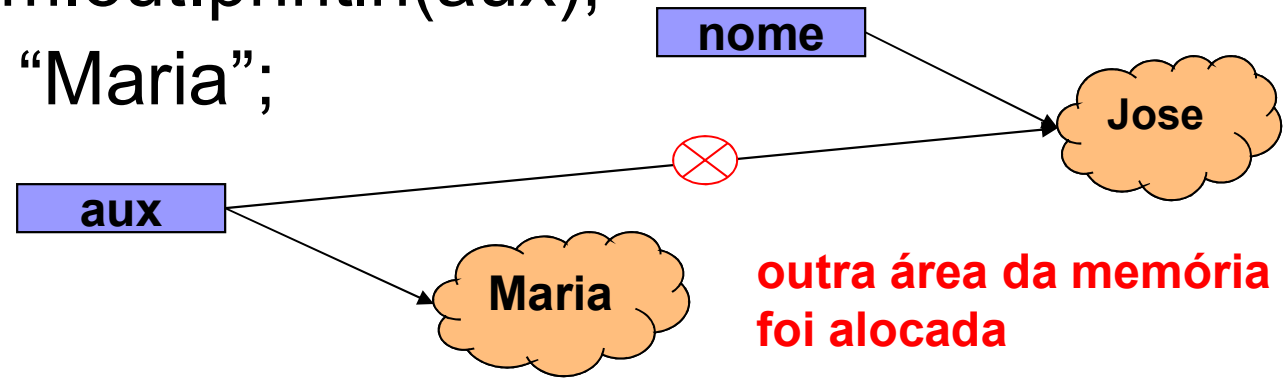
Atenção para o caso particular de Strings:

são sempre constantes

ao modificar instanciamos uma nova

```
// arquivo Prog.java  (P maiúsculo)
public class Prog{
    public static void main(String[ ] args){
        String nome = “Jose”;
        meuMetodo(a);
        System.out.println(nome);
    }
    public static void meuMetodo(String aux){
        System.out.println(aux);
        aux = “Maria”;
    }
}
```

irá imprimir “Jose”, pois String, apesar de objeto, é constante



Indentação

```
public static int meuMetodo(int x){
        int contador=0;
        while (soma < 1000) {
                if (x > 100){
                        soma += 100;
                } else if (x > 10) {
                        soma += 10;
                } else {
                        soma += 1;
                }
                contador++;
        }
}   // para corrigir a identação de um trecho no Eclipse selecione-o e
    digite control+shift+f
```

Regras de nomenclatura

- Nomes de classe:

primeira letra de cada palavra fica em maiúscula. Ex:

public class Funcionario {

- Nomes de métodos e atributos:

tudo em minúscula. Ex:

int soma;

public static void ordenacao(int[ ] vetor) {

Regras de nomenclatura

- Palavras compostas:

Segunda palavra (e as demais) com inicial maiúscula sem usar *underscore*. Ex:

```
public class MeuPrograma {
int totalGeral =0;
public static void minimosQuadrados() {
```

- Constantes

Use tudo em maiúsculas, eventualmente ligando com *underscore*

```
public class Funcionario {
    // a palavra final torna um atributo constante
    public static final double SALARIO_MINIMO=2000.0;
    // obs: valor em algum outro país...
}
```

Exercício LAB 0

Objetivo:
experimentar a linguagem, sua sintaxe básica e o ambiente de desenvolvimento escolhido.
Roteiro:
faça um programa Java (um único arquivo Prog.java) não orientado a objetos para pesquisar a existência de um número inteiro em um vetor que você irá declarar e inicializar.

i)inicialmente use uma função para fazer pesquisa seqüencial
no vetor passado como parâmetro junto ao número procurado
ii)passe a fazer pesquisa binária (precisa ordenar primeiro)
iii)repita o ítem ii pesquisando um vetor de nomes

Instituto de Computação

MÓDULO 3

Orientação a Objetos:

Conceitos e uso de Classes e Objetos

Instituto de Computação

Histórico

- Se qualquer programa é traduzido em linguagem de máquina, por que não programamos diretamente nesta linguagem?
- A resposta (óbvia) é que precisamos trabalhar num nível mais elevado para tratar melhor um problema computacional com um mínimo de esforço e um máximo de eficiência.

Instituto de Computação

Histórico

- Na década de 60, vimos o surgimento da programação estruturada e de centenas de linguagens de alto nível. Buscava-se reaproveitar o código por meio de subrotinas e facilitar seu entendimento evitando-se o uso de GO TO e programas monolíticos.

Instituto de Computação

Histórico

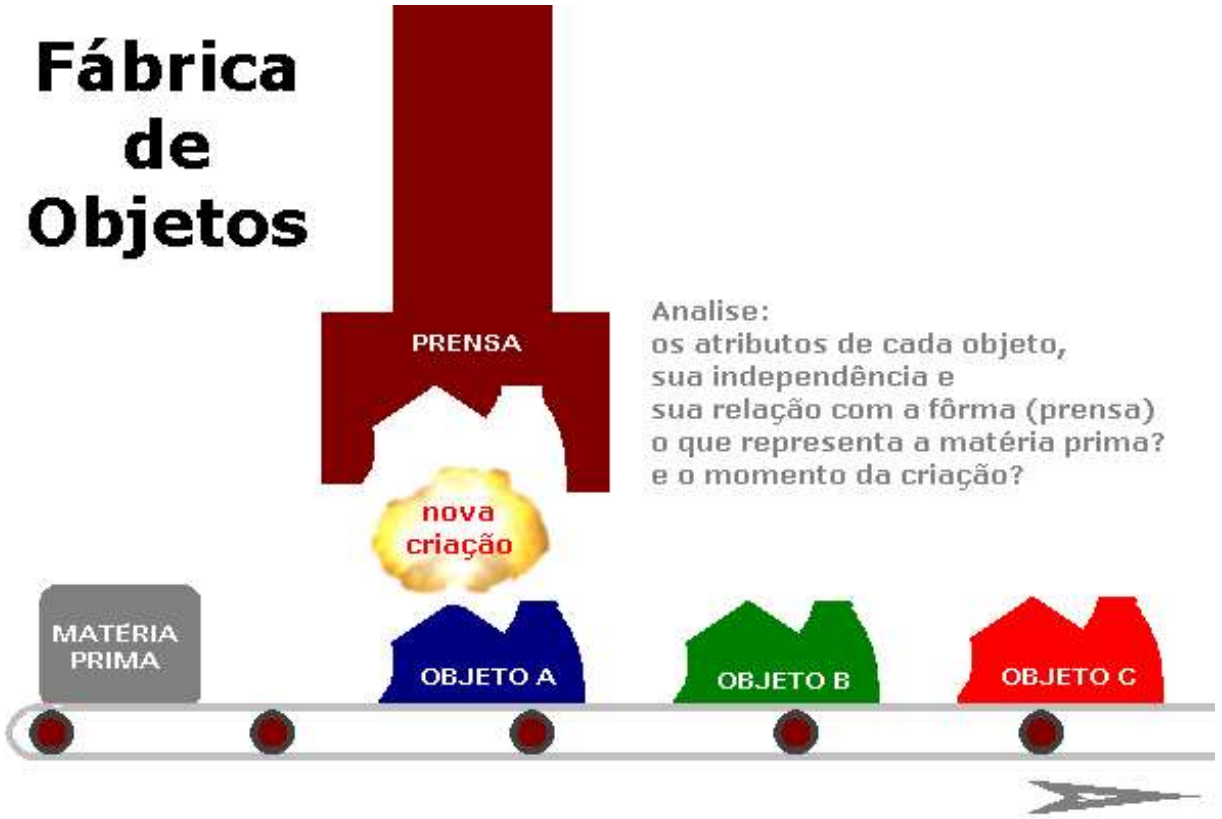
- Na década de 70 a ênfase foi nos dados com o amadurecimento do modelo de banco de dados relacional.
- Surge a idéia de tipos abstratos de dados como uma forma precursora de objetos. A idéia era entender que um conjunto de funções e variáveis tinham um papel de representar uma única entidade, por exemplo, uma pilha ou uma fila.
- Em paralelo vimos o surgimento do UNIX e do C como tentativas de padronizar a Babel da diversidade de computadores, formatos e padrões

Instituto de Computação

Histórico

- Nos anos 80, surge a idéia de orientação a objetos como forma de imitar o sucesso de outras indústrias.
- A indústria eletrônica já aplicava o modelo de componentes. As demais também já usavam com sucesso a idéia da linha de montagem, da fábrica de objetos.
- Desta forma, trabalhar com o conceito de objeto pareceu ser algo mais intuitivo e natural para o ser humano.

a Classe visualizada como padrão para produção de objetos



Exemplo de classe

```
// arquivo Funcionario.java

public class Funcionario {
    public String nome;
    public double salario;
    public int matricula;
}
```

Exemplo de uso

```
// arquivo Programa.java
public class Programa {
    public static void main(String[ ] args) {
        Funcionario f1;
        f1 = new Funcionario( ); // construtor padrão
        f1.nome = “Jose”;
        f1.salario = 3000.0;
        f1.matricula = 123;
        System.out.println(f1.nome);
    }
}
```

Instituto de Computação

Desmistificando o conceito de Classe

- Observe no exemplo anterior que a classe só tem campos (atributos) e todos eles públicos
- Veja também que ao criarmos um funcionário alocamos a memória com new e usamos a variável f1 para referenciar esta posição.
- Criamos na verdade uma estrutura equivalente ao record do Pascal ou struct do C
- Uma classe é isto: - um record estendido. Além de atributos, teremos métodos (funções) também no contexto da estrutura criada.

Modificadores de visibilidade

- No exemplo anterior, o programa que manipula funcionário tem toda a liberdade de alterar o valor de salário, definir um número negativo para matricula ou mesmo criar um nome com extensão superior àquela usada no armazenamento permanente.
- Isto pode ser altamente indesejável.

public vs private

- Para encapsular os atributos defendendo-os de acessos indevidos ou foras do modelo definido para a Classe Funcionario, torna-se necessário do modificador de visibilidade private
- Podemos imaginar uma classe Funcionario modelada no serviço de processamento de dados federal para ser entregue (bytecode) às prefeituras de todo o país para que estas criem suas próprias folhas de pagamento.
- Na lógica da classe Funcionario projetada, não será possível nenhum salário acima do presidente e nem abaixo do mínimo.

```
public class Funcionario {
    private double salario;
    public double getSalario( ) {
            return salario;
     }
    public void setSalario(double salario) {
            this.salario = validaSalario(salario);
            // observe o this. evitando ambigüidade do nome do atributo com
            // a variável local salario
     }
    public static final double SALARIO_MINIMO = 380.0; // observe o final para criar
    public static final double SALARIO_PRESIDENTE = 24000.0; // constantes
    private double validaSalario(double salario) {
            if (salario > SALARIO_PRESIDENTE) // observe o tratamento simplificado
                    return SALARIO_PRESIDENTE; // sem mensagem de erro ou log
            else if (salario < SALARIO_ MINIMO )
                    return SALARIO_ MINIMO;
            return salario;
    }
    private String nome; // Exercício: crie os métodos get e set para
    private int matricula; // nome e matrícula também
}
```

O que aprendemos no exemplo anterior?

- Métodos get / set

  São usados para dar acesso aos atributos encapsulados. No exemplo, o set faz com que o programa de folha de pagamento não consiga fazer nenhuma alteração indevida de salário. Usa-se o get / set seguido do nome do atributo com inicial maiúscula.
- Uso do this.

  O this é um apelido para o endereço de memória do objeto. O this. referencia um atributo ou método se necessário.
- Uso de final

  São três os usos de final. No exemplo, quando aplicado a atributos define seu valor inicial como imutável (constante)

Como fica a Folha de Pagamento?

```
// arquivo GeraFolhaPagamento.java
public class GeraFolhaPagamento {
    ...
    public void aplicaAumento(Funcionario f1, double percentual) {
        double novoSalario = f1.getSalario ();
        novoSalario *=  (1.0 + percentual / 100.0);
        f1.setSalario (novoSalario);
        // não importa os adicionais da prefeitura, pois o salário
        // resultante ficará dentro da faixa prevista na legislação.
        // Observe uma questão de modelagem e coesão:
        // tudo que for referente a funcionário deve ficar dentro da
        // classe Funcionario !
    }
}
```

Instituto de Computação

Construtores

- São métodos sem valor de retorno (nem mesmo void) usados somente com o new para a criação de um objeto.
- Se você não definir um construtor estará usando um construtor padrão (sem parâmetros)
- Construtores servem para inicializar um objeto: criar estruturas de dados, abrir conexões à Internet ou BD para recuperar os valores iniciais do objeto ou mesmo, como em nossos exemplos, para definir com parãmetros os valores iniciais de atributos. OBS: você poderia usar o construtor padrão e métodos set para esta última tarefa!

```
public class Funcionario {
    private double salario;
    private String nome;
    private int matricula; // get e set não representados
    public Funcionario (String nome, double salario, int matricula){
            this. salario = verificaSalario(salario);
            this. nome = nome; // demais verificações omitidas
            this. matricula = matricula;
    }
    // e se precisarmos instanciar um funcionario prestador de serviços
    // que não tem número de matrícula?

    /* inserimos outro construtor sobrecarregado. A sobrecarga é mera comodidade para o
    programador de aplicação que não precisa inventar parâmetros ou usar o que não for preciso
    ou adequado à sua lógica. */
    public Funcionario (String nome, double salario){
                this(nome, salario , PRESTADOR_SERVICOS );
                this.matricula = PRESTADOR_SERVICOS ;
    }
    public static final int PRESTADOR_SERVICOS = -1; // ATENÇÂO: aqui mais uma questão de
    // lógica e coesão, pois quem sabe o código de prestador é a classe Funcionário
    /* Observe que chamamos o método construtor já criado para evitar repetir tudo o que já foi
    escrito no construtor: foram apenas 3 linhas, mas poderia ser muito mais. Entretanto, usar
    this(...) é a única forma de se fazer isto. Não podemos chamar um construtor diretamente a não
    ser com o new. Este this(...) serve, portanto, para chamar algum outro construtor em
    sobrecarga e só pode ser empregado na primeira linha (primeira instrução) de um construtor.
}   OBS: não confunda o uso de this. com this(...) */
```

Conceitos

- Sobrecarga ou Overload

  Ocorre quando métodos, construtores ou não, numa mesma classe têm o mesmo nome, porém apresentam número e/ou tipo diferente de parâmetros.

- this., this(...) e this

  o primeiro destina-se a resolver ambigüidades entre nomes de variáveis e atributos; o segundo para chamar construtor em sobrecarga e o terceiro para que o objeto possa informar, quando necessário, o seu endereço onde ele está alocado. Observe que no mundo externo ao objeto já temos esta referência. Ex:

```
Funcionario f1;
f1 = new Funcionario( ); /* f1 é o nome usado para apontar.
Dentro da instância o objeto não sabe seu nome.
As pessoas te chamam por seu nome, você, entretanto se
referencia com o pronome EU. O this tem função semelhante */
```

Exercício LAB 1

Objetivo:
experimentar a programação O.O. , o uso de construtores, métodos get/set, usos de this, final e modificadores de visibilidade
Roteiro:
faça um programa Java composto de 3 classes:
a classe Funcionario com todos os atributos e métodos discutidos e um atributo “vetor de Dependentes” (Dependente[ ] vetDep) com o qual você irá referenciar os dependentes de cada funcionário (no máximo 10);
a classe Dependente que, por simplicidade, terá apenas os atributos nome e idade e
a classe Lab1 onde o seu programa irá iniciar (tem o método main). Nesta classe você irá criar alguns funcionários e alguns dependentes. Crie o método addDep(Dependente d) em Funcionario para ir adicionando dependentes ao vetor de Dependentes de cada funcionário. Ao final, chame o método imprime() de Funcionario para listar seus atributos e o vetor de seus dependentes.

Instituto de Computação

MÓDULO 4

Herança

A herança como forma de reaproveitamento de código

- Vimos nos tópicos anteriores a classe Funcionario como um record e fomos, progressivamente, enriquecendo-a de forma a encapsular atributos, fazer programação defensiva etc.
- Estamos agora interessados em especializar um pouco mais esta classe para poder representar, por exemplo, um Gerente ou um Coordenador (ambos funcionários).

A herança como forma de reaproveitamento de código

- Suponhamos que o Coordenador tenha tudo que o Funcionario (comum) tenha mais um nome de projeto. O Gerente, por sua vez, tem os atributos bônus anual e carro que recebe da empresa. Este Gerente, apesar das mordomias, é também um Funcionario.

A herança como forma de reaproveitamento de código

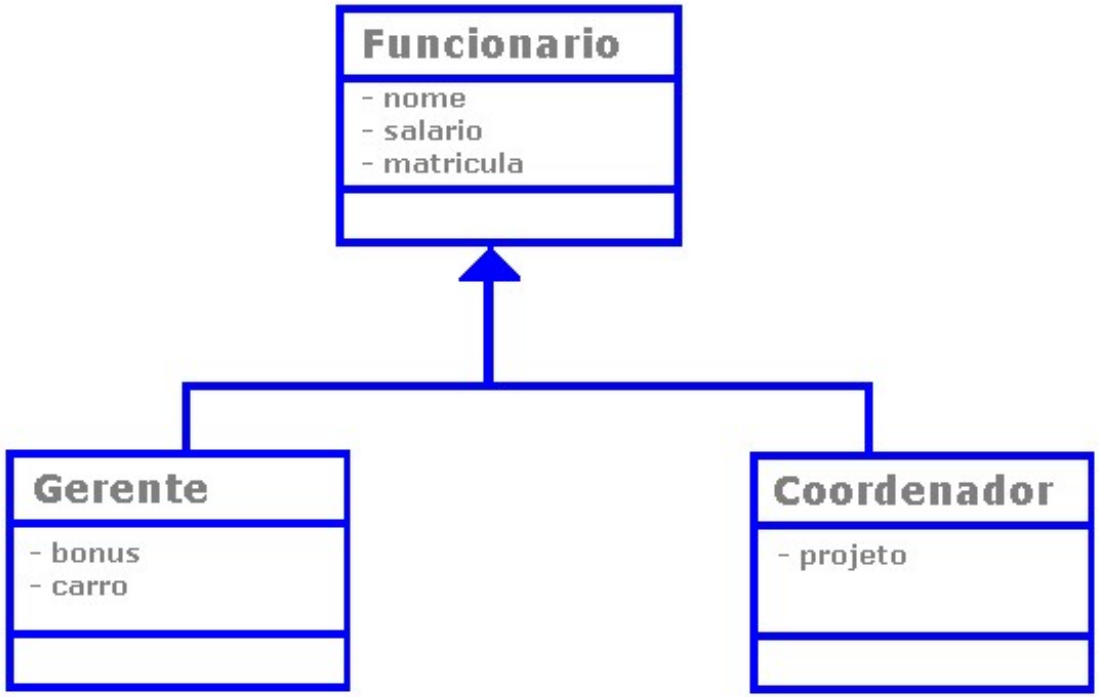
- Parece pouco razoável fazer um cut&paste para pegar todo o código criado para Funcionario e usar em Gerente e Coordenador.
- Também não faz muito sentido ir adicionando à classe Funcionario atributos que um funcionário comum jamais irá usar. Não podemos correr o risco de colocar todo o universo do problema dentro de uma única classe ou deixaremos de programar O.O.

Qual a solução?

- A solução é a criação de uma subclasse ou classe filha de Funcionario que irá herdar o que é comum a Funcionario e definir os atributos de sua especialização.
- Teremos uma espécie de cut&paste invisível e controlado pela linguagem ao declarar:

```
public class Coordenador extends Funcionario {
        private String projeto;
        public Coordenador (String nome, double salario, int matricula, String projeto ){
                super(nome, salario, matricula); // reaproveito construtor da superclasse
                this. projeto = projeto;
        }
        public void imprime(){
                super.imprime(); // tambem aproveitei o imprime da superclasse.
                System.out.println("atributos especiais do cargo: projeto "+projeto);
        }
} // e isto é tudo! Observe que eu posso usar tudo (que for visível) de Funcionario
```

Diagrama de Classes



Conceitos

- A classe situada acima é dita superclasse e a classe abaixo é chamada de subclasse
- Sempre que pudermos empregar a expressão "É UM" estamos lidando com uma situação candidata a herança.
- Diagrama de classes não é organograma. O funcionário comum estará, deste modo, sempre acima de seu Gerente e pode com segurança chamá-lo de subclasse! ;-)
- Observe que no exercício anterior o funcionário "TEM UM" (ou mais) dependente(s). Assim, do ponto de vista O.O. , não há herança entre Funcionario e Dependente mesmo se os humanos envolvidos forem pais e filhos!

Uso de protected

- protected é um modificador de visibilidade que permite manter um atributo (ou método) invisível (como em private) ao mundo externo à classe, porém possibilita dar acesso ilimitado às classes filhas (como se estivesse na mesma classe). Isto dá comodidade aos programadores da classe filha, supostamente mais confiáveis que os programadores da aplicação.

Exemplo: fica mais fácil usar

```
salario *= 1.2;
```

do que

```
double   aux = getSalario( );
aux *= 1.2;
setSalario(aux);
```

Uso de super(...) e super.

- *super( )* tem função equivalente ao *this( )* , invocando porém um construtor da superclasse. OBS: tem que estar na primeira linha de um construtor (1a instrução)
- *super.* serve, como o *this.* para resolver ambigüidades. Por exemplo, ao chamarmos o método super.imprime() em Coordenador, evitamos a chamada recursiva do método imprime() local. Use quando necessário.

Instituto de Computação

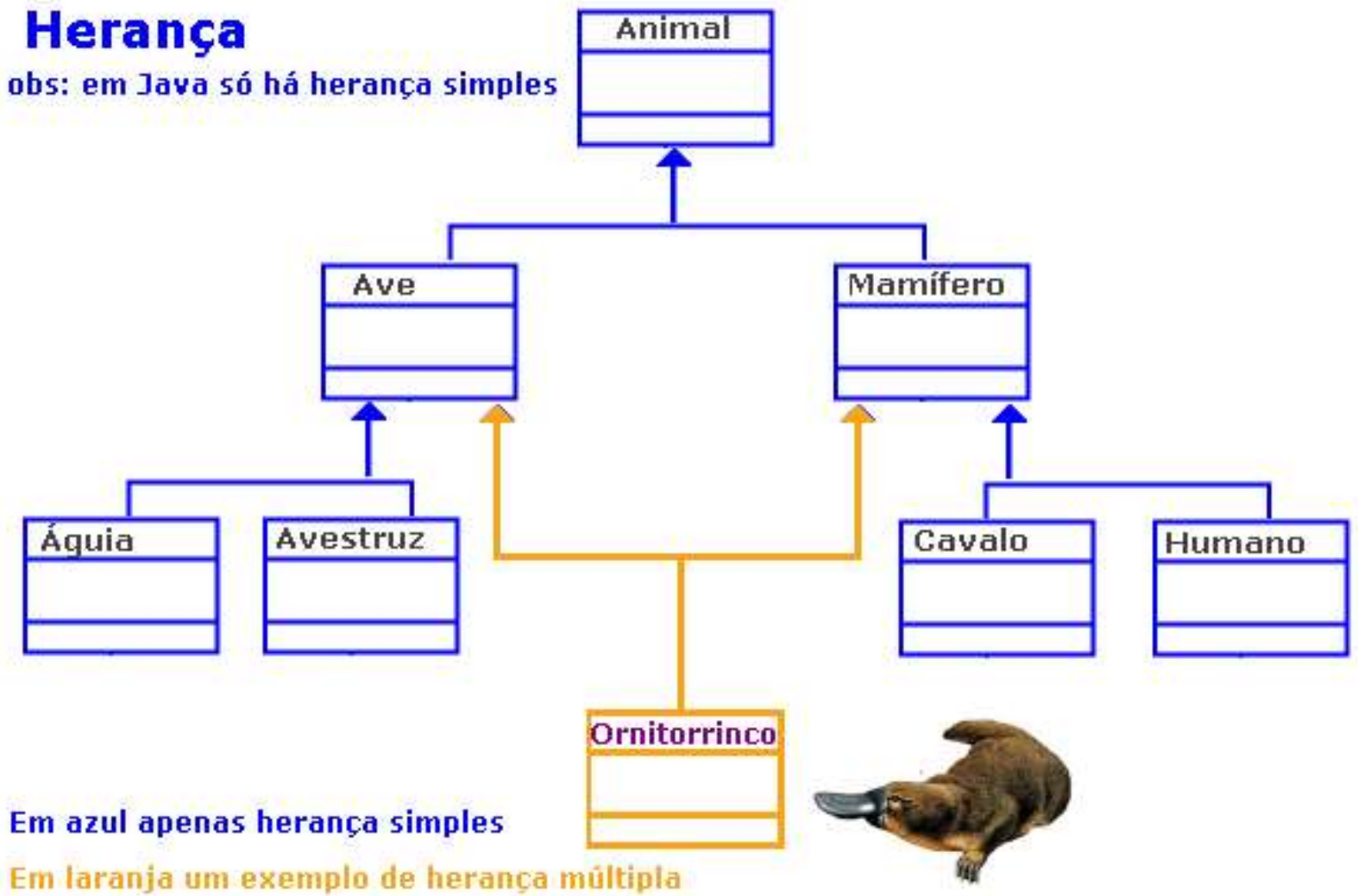
Conceitos

- Assinatura de método: é o nome do método, o tipo e a quantidade de parâmetros recebidos.
- Sobreescrita, override ou anulação: quando temos métodos com mesma assinatura em duas classes ligadas por herança temos um exemplo de sobreescrita.
- O mecanismo de sobreescrita é FUNDAMENTAL para a existência de polimorfismo como veremos nos próximos slides

Observe o código de Funcionario e Coordenador para entender este exemplo ilustrando a anulação (sobreescrita).

```
// arquivo Programa.java
public class Programa {
    public static void main(String[ ] args) {
        Funcionario f1;
        f1 = new Funcionario("Jose", 3000.0, 123 );
        f1.imprime(); // chama o método imprime() de Funcionario
        Coordenador c1;
        c1 = new Coordenador ("Maria", 3500.0, 124, "intranet depto" );
        c1.imprime(); /* o imprime() de Funcionario não é chamado e
         sim o imprime() do coordenador. Para efeito da aplicação é
        como se o método original tivesse sido anulado */
    }
}
```

Herança Simples vs Herança Múltipla



Exercício LAB 2

Objetivo:
experimentar a programação O.O. , o uso de herança, protected e sobreescrita
Roteiro:
faça um programa Java composto de classes adicionais àquelas do LAB1:
as classes Gerente e Coordenador com todos os atributos e métodos discutidos neste módulo;
a classe Departamento com um atributo nome e um “vetor de Funcionarios” (Funcionario[ ] vetFuncionario) com o qual você irá referenciar os funcionarios (comuns, gerentes e coordenadores) (defina um máximo de 50);
a classe Lab2 onde o seu programa irá iniciar (tem o método main). Nesta classe você irá criar alguns funcionários(dos 3 tipos), um ou dois departamentos e alguns dependentes. Crie o método addFuncionario(Funcionario f) em Departamento para ir adicionando funcionarios ao vetor de Funcionarios de cada departamento. Ao final, chame o método imprime() de Departamento para listar seus atributos e o vetor de funcionarios. OBS: ao imprimir cada funcionario, seus dependentes serão também impressos.

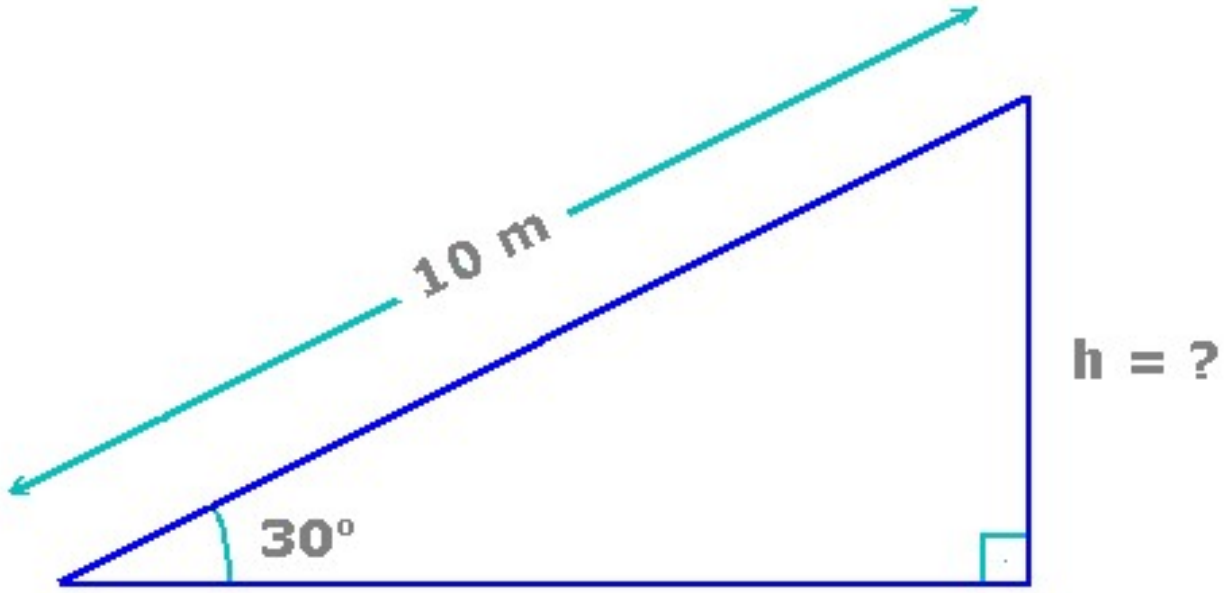
Instituto de Computação

MÓDULO 5

Modificador static
A Classe Math e a Classe Stack

Lembrando de geometria...

Um veículo movendo-se 10m sobre um plano inclinado em 30 graus do solo terá se elevado quantos metros ao fim do trajeto?



Escreva um algoritmo Java para resolver o problema.

double h = 10.0 * seno(30 graus); ???

Instituto de Computação

Static e Math

- Se tudo são classes e objetos, então não devem existir bibliotecas de funções “soltas” no Java como em outras linguagens.
- Deve existir então uma classe Matemática para permitir o uso de métodos matemáticos.

Static e Math

- Se a classe for Math então teremos

```
Math aux = new Math(); // criação do objeto
double h = 10.0 * aux.sin( aux.PI / 6.0); // ângulo em radianos
```

- Ok?
- NÃO! O problema é que se tivermos que criar um objeto a cada uso de uma função ou constante matemática vamos ter que escrever muito, perder a legibilidade do código e alocar memória desnecessariamente.
- A solução é dispormos de atributos e métodos estáticos, já disponíveis em tempo de compilação. Podemos assim, prescindir da criação de objetos para usar o código da classe.

Como ficaria o código e a classe Math?

- double h = 10.0 * Math.sin(Math.PI / 6.0);

// sin é um método static e PI uma constante static

- A classe Math poderia ser implementada assim:

```
public final class Math {
    public static final double PI = 3.1415926536;
    public static double sin(double anguloRad){
            double aux = // faz o calculo do seno
            return aux;
    }
    public static final double E = 2.7182818;
    // etc outros métodos e constantes
    private Math( ) {
            // ... e construtor privado para evitar que alguém instancie
    }
}
```

Instituto de Computação

Usos de final

- você já usou o final para tornar constantes alguns atributos
- no exemplo anterior o final foi usado junto a classe Math para impedir que criemos uma classe filha. Portanto, final aplicado à classes impede a herança.
- Finalmente (sem trocadilhos) o uso de final aplicado a métodos impede sua sobreescrita

static: ampliando a explicação

```
// suponha a classe Livro em Livro.java
public class Livro{
    private String nome; // e get/set
    private double preco; // idem
    private double desconto; // idem
}
// e um programa livraria com um trecho de código no arquivo Livraria.java
public class Livraria {
    ...
    public double calculaPrecoFinal(Livro book1) {
        ...
        double valor = book1.getPreco() * (1.0 - book1.getDesconto() );
        return valor;
    }
}
```

Se o livreiro tiver a política de descontos únicos para toda a loja, pode ser que o atributo desconto tenha alguma inconsistência, algum valor diferente de um livro para o outro o que seria indesejável.

Como garantir que todos os livros tenham o mesmo desconto ?

```
// suponha a classe Livro em Livro.java
public class Livro{
    private String nome; // e get/set
    private double preco; // idem
    private static double desconto;
    public static double getDesconto(){
        return desconto;
    }
    public static void setDesconto(double d ){
        desconto = d; // aqui não poderíamos usar o this. ! Por quê?
    }
}
// e um programa livraria com um trecho de código no arquivo Livraria.java
public class Livraria {
    ...
    public double calculaPrecoFinal( Livro book1) {
        ...
        return book1.getPreco() * (1.0 - Livro.getDesconto() );
        // agora o desconto é da classe Livro como um todo
    }
}
```

- Atenção: um método de instância pode usar um atributo de classe (static), mas um método de classe não pode usar um atributo de instância. Pense no porquê disto...

É possível instanciar uma classe com construtor privado?

```
// arquivo Game.java
public class Game {
    private String gameName;
    private Game(String name){
        gameName = name;
    }
}

// arquivo Prog.java  (a propósito, por que main é static?)
public class Prog {
    public static void main(String[ ] p){
        Game g = new Game("Air Attack"); // ???
    }
}
```

Sim, é possível!

```
// arquivo Game.java
public class Game {
    private String gameName;
    private Game(String name){
        gameName = name;
    }

    private static Game singleton = null;
    public static Game criaUnicoGame(String name){ // design pattern singleton
        if (singleton == null)
            singleton = new Game(name); // cria uma única instância e sempre devolve a mesma
        return singleton;
    }
}
// arquivo Prog.java
public class Prog {
    public static void main(String[ ] p){
        Game g = Game.criaUnicoGame("Air Attack");
    }
}
```

Outros atributos e métodos da classe Math

Instituto de computação

java.lang

Class Math

java.lang.Object
  └ java.lang.Math

Acostume-se a ler a documentação (JAVADOC) presente no diretório doc de sua instalação do JDK.

Não reinvente a roda. Se você procura alguma biblioteca útil, seja científica, financeira ou de propósito geral pode ter uma certeza:
- ela já está disponível e bem documentada!

public final class **Math**
extends Object

Field Summary

static double	E The double value that is closer than any other to e, the base of the natural logarithms.
static double	PI The double value that is closer than any other to *pi*, the ratio of the circumference of a circle to its diameter.

Method Summary

static double	abs(double a) Returns the absolute value of a double value.
static float	abs(float a) Returns the absolute value of a float value.
static int	abs(int a) Returns the absolute value of an int value.
static long	abs(long a) Returns the absolute value of a long value.
static double	ceil(double a) Returns the smallest (closest to negative infinity) double value that is not less than the argument and is equal to a mathematical integer.
static double	cos(double a) Returns the trigonometric cosine of an angle.
static double	floor(double a) Returns the largest (closest to positive infinity) double value that is not greater than the argument and is equal to a mathematical integer.
static double	max(double a, double b) Returns the greater of two double values.

Instituto de Computação

Classe Stack

- Todas as classes são filhas da classe mais elementar do Java, a classe Object.
- Ao invés de criar um array de um certo tipo de objetos como fizemos, poderíamos ter utilizado classes utilitárias como pilhas, filas por exemplo
- Há outras muito úteis como filas, hash tables etc. Neste curso, iremos exemplificar com a classe Stack.

Classe Stack

```
import java.util.EmptyStackException;
import java.util.Stack;

public class StackDemo {
    public static void main(String args[]) {
        Stack<Integer> st = new Stack<Integer>(); // cria uma pilha de inteiros
        System.out.println("stack: " + st);
        st.push(1);       st.push(2);       st.push(3);  //  empilha
        while (true){
            System.out.println("stack: " + st);
            try {
                st.pop(); // desempilha
            } catch ( EmptyStackException e){ // não consegue desempilhar
                System.out.println("stack vazia " );
                break; // sai do loop
            }
        }
    }
}
```

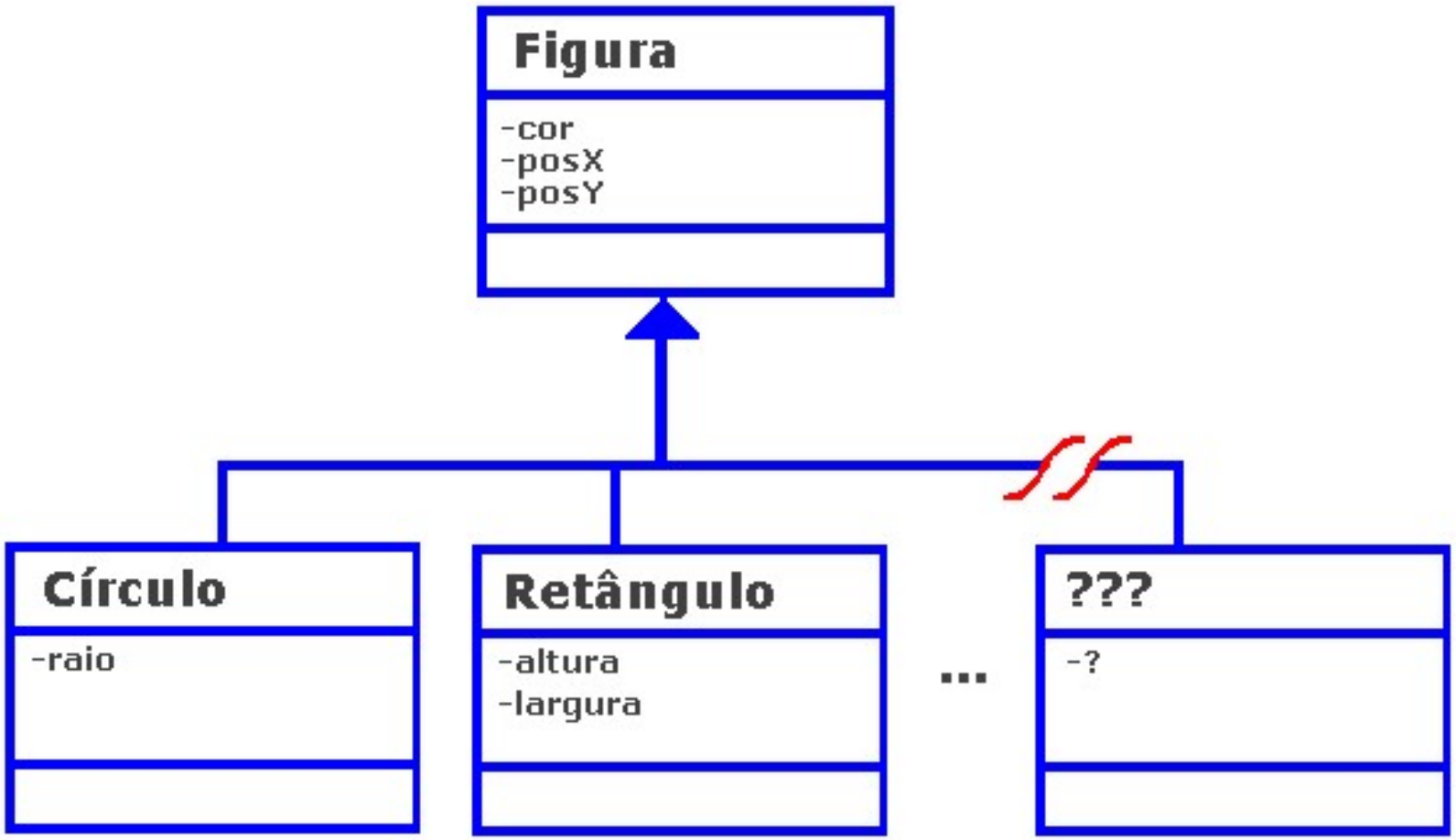
Instituto de Computação

MÓDULO 6

Classes Abstratas e Polimorfismo;

Declaração e Uso de Pacotes

Você consegue desenhar uma Figura?



Classe Abstrata

- A classe Figura do diagrama anterior faz todo o sentido, pois há muitos atributos e métodos comuns as demais subclasses cujo código não deve ser repetido
- Ela nos permite dar um tratamento uniforme para os elementos de desenho de um editor gráfico da mesma forma que fizemos com Gerentes, Coordenadores e Funcionarios no vetor de funcionários.
- Entretanto, o que você não vai conseguir fazer é desenhar, numa folha de papel ou no computador, uma figura (nem salvar, nem recuperar, etc). Isto porque Figura é algo abstrato ao contrário de um retângulo ou círculo por exemplo.

```
public abstract class Figura {
    protected int posX; // e métodos get / set não representados
    protected int posY;
    protected int cor;
    public Figura(int cor, int x, int y){
            posX = x;
            posY = y;
            this.cor = cor;
    }
    // os métodos abaixo são abstratos: só possuem a assinatura
    public abstract void draw(Graphics screen) ;
    public abstract void salvar( ) ;
    public abstract void carregar( ) ;
}
```

Os métodos abstratos devem ser implementados pelas subclasses concretas

Instituto de Computação

```
public class Circulo extends Figura {
    protected int raio; // e métodos get / set não representados
    public Circulo(int cor, int x, int y, int raio){
            super(cor, x, y);
            this.raio = raio;
    }
    // os métodos abaixo são concretos de modo a cumprir o contrato
    public void draw(Graphics screen) {
            // usa primitivas para desenhar um círculo na tela
            ...
    }
    public void salvar( ) {
            // grava o estado corrente do objeto círculo
            ...
    }
    public void carregar( ) {
            // recupera o estado corrente do objeto círculo
            ...
    }
}
```

Instituto de Computação

Conceitos

- Uma classe abstrata pode ter zero, um ou todos os seus métodos abstratos.
- Entretanto, se uma classe tiver pelo menos um método abstrato ela terá que ser abstrata.
- Um método abstrato é um contrato, uma implementação obrigatória repassada à subclasse. Esta subclasse deverá implementar o corpo deste método ou terá que ser abstrata também.
- Uma classe abstrata não pode ser instanciada e nem ser final.

Polimorfismo

Observe o trecho código abaixo:

```
...
private Figura[ ] vetorFiguras = new Figura[1000];
...
public void menuCirculo( ) {
    vetorFiguras[numeroObjetosGraficos++] = (Figura) new Circulo(...);
}
public void menuRetangulo( ) {
    vetorFiguras[numeroObjetosGraficos++] = (Figura) new Retangulo(...);
}
...
public void desenha(Graphics screen) {
        /* percorre todo o vetor de objetos criados pelo usuário
        para fazer o desenho na tela. Você consegue dizer que objeto está
        na posição 13 do vetor ? */
        for (int i=0; i<numeroObjetosGraficos;i++)
                vetorFiguras[i].draw(screen);
}
```

Instituto de Computação

Polimorfismo

- Quando precisamos executar o programa, ainda que mentalmente, para saber o método que será invocado temos polimorfismo
- Diz-se que um trecho de código apresenta comportamento polimórfico quando um trecho de código pode ser executado de muitas formas diferentes (POLI + MORPHOS)
- O código gerado é mais flexível e elegante
- É a base para criação de plugins

Polimorfismo

- Quando há sobreescrita de um método e atribuímos um objeto de uma subclasse a uma referência da classe, o compilador gera código para permitir o polimorfismo.
- Isto é chamado de *late-binding* ou ligação tardia (em tempo de execução) da função.

outro exemplo:

```
Funcionario f;
if ( responsavelDepartamento )
    f = new Gerente(...);
else
    f = new Coordenador(...);
f.imprime();// imprime poderia até ser um método abstrato
```

Instituto de Computação

Pacotes

- Num sistema muito grande, seria inadequado a colocação de todas as classes num mesmo subdiretório.
- Da mesma forma que agrupamos métodos e atributos relacionados a um mesmo tipo de objeto numa classe, faz sentido agruparmos classes relacionadas criando um pacote.
- Um pacote é implementado na forma de um subdiretório do sistema de arquivos empregado (Unix, Windows, etc) contendo estas classes com algum propósito semelhante.

Pacotes

- Para criarmos um pacote colocamos as classes componentes num mesmo diretório e definimos a instrução package no início do arquivo, antes mesmo da definição da classe. Ex:

```
package financeiro.banco.correntistas;
```

- Um pacote é implementado na forma de um subdiretório do sistema de arquivos empregado (Unix, Windows, etc).
- A classe que usar alguma outra classe presente num pacote deverá usar a instrução import. Ex:

```
import financeiro.banco.correntistas.Conta;
```

Pacotes

- O pacote pode ou não estar dentro de um arquivo .JAR (formato compatível com ZIP). Entretanto, é importante que ele seja acessível à aplicação a partir do diretório corrente, no diretório default das classes do Java ou por meio do *classpath* definido como parâmetro de execução da JVM (ex: java -classpath c:\projeto)
- O nome de pacote pode resolver a ambigüidade de nomes de classes criadas por duas empresas diferentes. Se você for criar a classe Conecta (nome bem original, não é?) do pacote util, por exemplo para a empresa www.ActiveTecnologia.com.br, basta defini-la como:

```
package br.com.activetecnologia.util;
public class Conecta {
...
}
```

Exercício LAB 3

Objetivo:
experimentar a programação O.O. , o uso de classes abstratas, polimorfismo e pacotes
Roteiro:
faça um programa Java composto das seguintes classes:
* Conta, uma classe abstrata contendo nome, cpf, saldo e número da conta;
* Comum, subclasse de Conta;
* Poupanca, subclasse de Conta, possuindo o atributo taxa de juros pagos;
* Especial, subclasse de Conta, possuindo o atributo taxa de juros cobrados e o atributo limite do cheque especial;
* Banco (ou Agencia) é uma classe para conter Contas (use arrays ou Vector)
* Principal.java; aqui você irá declarar contas, um banco, realizar operações nas contas e listar o banco como feito no exercício do Departamento
OBS: Em Conta haverá o método
protected abstract double creditoDebito(double valor);
destinado a receber operações de crédito (positivas) ou débito (negativas). A implementação em Especial leva em conta o limite do cheque especial do correntista ao aceitar ou recusar a operação de débito.

Instituto de Computação

MÓDULO 7

Interfaces

Definição:

- Uma interface compila como uma classe abstrata onde todos os métodos são abstratos e todos os atributos constantes. Exemplo:

```
// arquivo Comparador.java
public interface Comparador {
    int MENOR=-1;
    int MAIOR=1;
    int IGUAL=0;
    int comparadoA(Comparador x);
}
```

Interface

- A interface não interfere no mecanismo de herança. Uma classe pode estender outra e implementar N interfaces. Exemplo:

```
public class Mercadoria extends Item implements
    Comparador, TabelaMoedas {
    ...
}
```

Interface

- A interface é útil para servir como repositório de constantes para todo o projeto,
- Serve como um contrato para forçar a implementação de métodos por parte da equipe que a implementa.
- Entretanto, a principal utilidade está na criação de novos tipos de dados compatíveis com uma dada implementação. Neste sentido, a interface é útil para separar a especificação de uma implementação em particular.

Exemplo

- Suponha que desejemos criar uma classe Util para fornecer um método de ordenação para a classe Funcionario. Exemplo:

```
public class Util {
    public static void ordenaRuim(Funcionario[ ] vet){
        for(int k=0;k<vet.length-1;k++) {
            for(int j=k+1;j<vet.length;j++){
                if ( vet[k].getSalario() > vet[ j ].getSalario( ) ){
                    Funcionario aux=vet[k];
                    vet[k]=vet[j];
                    vet[j]=aux;
                }
            }
        }
    }
    ...
} // por que esta classe é tão ruim???
```

O Problema

- A classe anterior é ruim pelo fato de usar o método da bolha e de usar ordenação que é algo ja´disponível em bibliotecas do Java.

- Tirando este fato o método ordena só serve para ordenar funcionários e ainda assim
- somente em ordem crescente por salários. Ou seja, nada genérica.

A Solução

```
public class Util {
    public static void ordena(Comparador[] vet){
        for(int k=0;k<vet.length-1;k++)
            for(int j=k+1;j<vet.length;j++)
                if (vet[k].comparadoA(vet[j]) == Comparador.MAIOR){
                    Comparador aux=vet[k];
                    vet[k]=vet[j];
                    vet[j]=aux;
                }
    }
    ...
} /* por que esta classe é melhor?
    É melhor por ordenar vetores genéricos do tipo Comparador
    usando comparações que somente o tipo Comparador sabe
    fazer (método comparadoA retorna MAIOR, MENOR ou IGUAL)
    Conseguimos com isto um desacoplamento entre o método
    utilitário e o tipo das classes a serem ordenadas */
```

A interface define um tipo

```
// arquivo Comparador.java
public interface Comparador {
      int MENOR=-1;
      int MAIOR=1;
      int IGUAL=0;
      int comparadoA(Comparador x);
}
// arquivo Funcionario.java
public class Funcionario implements Comparador { ...

// arquivo Mercadoria.java
public class Mercadoria extends Item implements Comparador { ...

// Funcionario e Mercadoria são do tipo Comparador, logo seus vetores podem ser ordenados
      por um mesmo método, além disso cada um deles provê o resultado e o tipo da
      comparação que se quer na ordenação
```

```
public class Mercadoria extends Item implements Comparador {
    private String nome; // e get / set
    double preco;
    public Mercadoria(String nome, double preco){
        this.nome = nome;
        this.preco = preco;
    }
    public int comparadoA ( Comparador x) { // contrato de implementação
        Mercadoria m = (Mercadoria) x; // type cast
        if (preco > m.getPreco() )  // OU if (nome.compareTo(m.getNome()) > 0)
            return MAIOR;
        else if (preco < m.getPreco()) // OU else if (nome.compareTo(m.getNome()) < 0)
            return MENOR;
        return IGUAL;
    }
}
// de modo análogo ocorre a implementação de comparadoA em Funcionario
// vide o exemplo InterfaceSample.zip
```

```
public class Prog {
    public static void main(String[] args) {
        Funcionario[] vetf = new Funcionario[3];
        vetf[0] = new Gerente("Juliana", 8000.0, 1, "Fox", 3000.0);
        vetf[1] = new Coordenador("Bia", 3000.0, 2, "Mobile Game Project");
        vetf[2] = new Funcionario("Edu", 2000.0);
        Util.ordena(vetf); // método para ordenar vetores do tipo Comparador
        System.out.println("\n\nordenação de funcionários");
        for (int k = 0; k < vetf.length; k++)
           System.out.println(vetf[k].getNome() + " " + vetf[k].getSalario());

        Mercadoria[] vet = new Mercadoria[3];
        vet[0] = new Mercadoria("profundimetro", 1000.0);
        vet[1] = new Mercadoria("mascara", 300.0);
        vet[2] = new Mercadoria("snorkel", 100.0);
        Util.ordena(vet); // método para ordenar vetores do tipo Comparador
        System.out.println("\n\nordenação de mercadorias");
        for (int k = 0; k < vet.length; k++)
           System.out.println(vet[k].getNome() + " " + vet[k].getPreco());
    }
}
```

MÓDULO 8

Tratamento de Exceções

Erros ou Exceções?

- Erros são irrecuperáveis
- Exceções são problemas que podem acontecer com freqüência e que demandam um procedimento de alto nível separando a lógica do programa do código de tratamento

Quando e como ocorrem?

- Ao ocorrer o evento inesperado um objeto representando a exceção é instanciado
- Um método lança uma exceção
- O problema é capturado e tratado em uma região separada da lógica fundamental do programa.

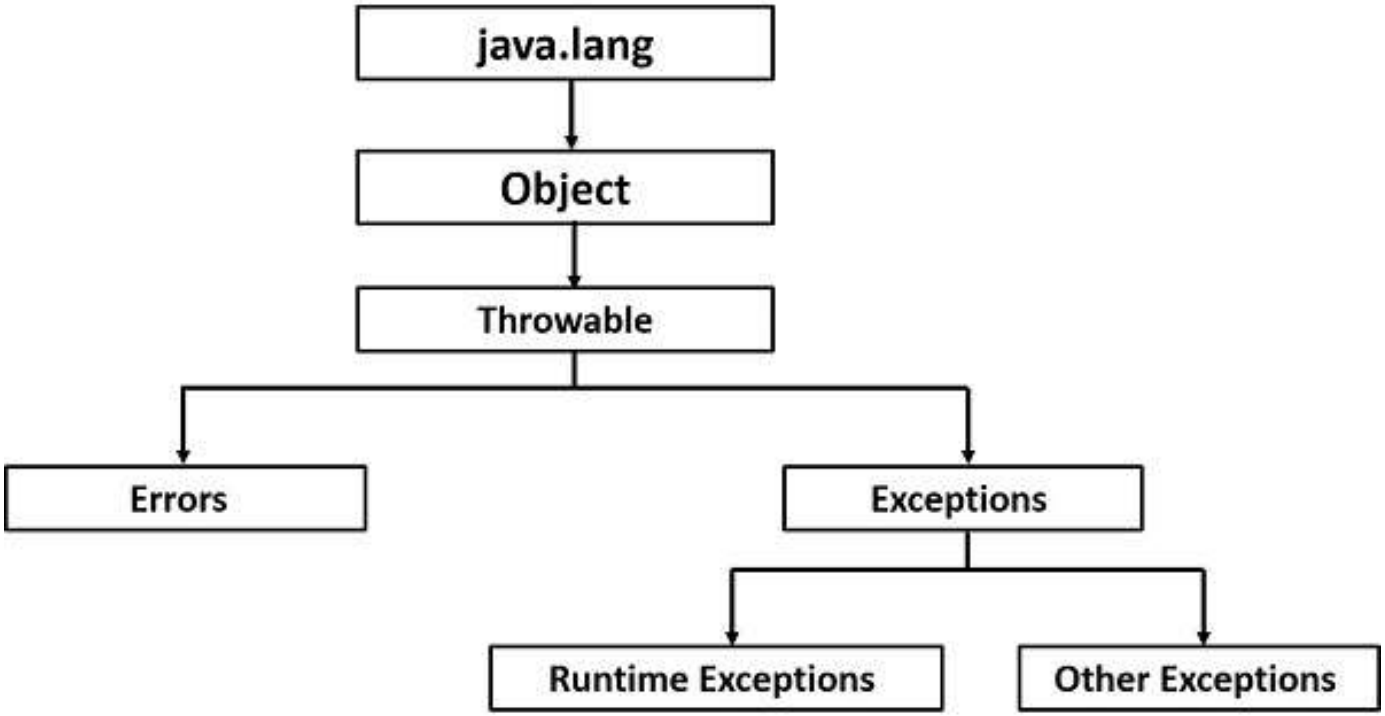
Exemplo

```
public static void listaConteudoZIP(){
        try {
                // Open the ZIP file
                ZipFile zf = new ZipFile("inexistente.zip");

                // Enumerate each entry
                int i=0;
                for (Enumeration entries = zf.entries(); entries.hasMoreElements();) {
                        // Get the entry name
                        String zipEntryName = ((ZipEntry)entries.nextElement()).getName();
                        System.out.println(++i + " "+zipEntryName);
                }
        } catch (IOException e) {
            System.out.println(e.getMessage());
            e.printStackTrace();
        }
}
```

```
O sistema não pode encontrar o arquivo especificado
java.util.zip.ZipException: O sistema não pode encontrar o arquivo especificado
at java.util.zip.ZipFile.open(Native Method)
at java.util.zip.ZipFile.<init>(ZipFile.java:203)
at java.util.zip.ZipFile.<init>(ZipFile.java:84)
at ZIPdemo.listaConteudoZIP(ZIPdemo.java:60)
at ZIPdemo.main(ZIPdemo.java:16)
```

Tipos de Exceções



Instituto de Computação

try catch finally

```
try {
    // abre conexão com Internet
    // abre arquivo
    // tenta salvar conteúdo obtido
} catch (IOException e) {
    // tratamento referente a IO
} catch (OutraExcecao e){
    // tratamento de outro problema
} catch (Exception e){
    // super classe mais genérica por último
} finally {
    // fecha arquivo: testar se não é nulo
    // fecha conexão: testar se não é nula
    // OBS: finally sempre irá executar independente de ter ou não
    // ocorrido uma Exception
}
```

Instituto de Computação

Lançamento de exceções

- Pode ser útil quando não queremos dar tratamento local e sim repassar o problema para outro trecho chamando nosso método
- Usa-se o comando **throw** para lançar uma exceção
- Junto à assinatura de nosso método declaramos as exceções lançadas com a palavra reservada **throws**

Lançamento de Exceções

```
public int verificaLimitesVetor(int indice) throws IndexOutOfBoundsException {
      if (indice < 0 || ( indice > Departamento.MAX_FUNCIONARIOS ) )
            throw new IndexOutOfBoundsException();
}
```

```
... em outro objeto
Departamento d = new Departamento();
Funcionario f = new Funcionario();
...
try {
      d.verificaLimitesVetor(ix);
      d.adicionaFuncionario(f, ix);
} catch(IndexOutOfBoundsException e) {
      // dá o tratamento adequado
}
...
```

Criação de suas próprias Exceções

- Extender a classe Exception
- Possibilita tratamento customizado de um problema
- Ex:

```
public class FundosInsuficientes extends Exception{
    public FundosInsuficientes (String mensagem){
            super(mensagem); // repassa ao construtor de Exception
    }
}
```

```
public class Conta {
        private double saldo;
        private int numero;
        public Conta(int numero) {
                this.numero = numero;
         }
        public void depositar(double valor) {
                saldo+= valor;
         }
        public void sacar(double quantia) throws FundosInsuficientes {
                 if(quantia <= saldo) {
                        saldo -= quantia;
                } else {
                        double diferenca = quantia - saldo;
                        throw new FundosInsuficientes("R$ " + diferenca);
                }
        }
        // ... Demais métodos
}
```

MÓDULO 9

Uso de Threads

Instituto de Computação

O que é uma thread

- É uma linha de execução da CPU
- Funciona como uma CPU virtual executando trechos do programa, aparentemente "ao mesmo tempo" que outra thread
- São controladas pela JVM, mas dependem da implementação disponível no S.O. subjacente.

Instituto de Computação

Por que usar threads?

- Permitem interfaces mais interativas com o usuário dando-lhe a chance de executar mais tarefas, aparentemente, ao mesmo tempo.
- Representa um uso mais eficiente da CPU. Por exemplo, entre uma tecla e outra apertada pelo usuário, há milhões de ciclos da CPU que podem e devem ser utilizados para realização de outras tarefas para o próprio usuário.
- Isto representa maior produtividade

Instituto de Computação

Como criar threads

- Estender a classe Thread
  - forma simples
- Implementar a interface Runnable
  - este último modo é muitas vezes o preferido pelo fato de que a interface não atrapalha o mecanismo de herança

Estendendo a classe Thread

```
public class MinhaCPU extends Thread{
        public void run(){
                int i=0;
                while (i<30){
                   System.out.print(i++ + " ");
                }
                // ao terminar o método run a thread termina
        }
        public static void main(String[ ] p){
                Thread t = new MinhaCPU();
                t.start(); // a thread fica preparada para executar
                for(char letra='a';letra<'z';letra++){
                   System.out.print(letra);
                }
        }
}
```

Implementando a interface Runnable

```
public class OutraCPU implements Runnable{
        public void run() {
                int i=0;
                while (i<20){
                   System.out.print(i++ + " ");
                }
                // ao terminar o método run a thread termina
        }
        public static void main(String[] p){
                OutraCPU o = new OutraCPU();
                Thread t = new Thread(o);// objeto o implementa Runnable
                t.start();
        }
}
```

Métodos de Thread

- **sleep(long miliseconds)**

```
try {
  Thread.sleep(200);//atua na thread atual
} catch (InterruptedException e) {
}
```

- **currentThread()**

```
// retorna a thread atual
```

vide exemplo MinhaCPUSleep.java

Métodos de Thread

- **join(long miliseconds)**

faz com que a thread atual espere até que a thread xyz invocada termine

```
xyz.join(); // trecho executado pela thread atual
```

É possível passar um timeout em join.

- **getPriority()/setPriority()**

retorna ou define a prioridade de uma thread. O parâmetro varia de 1 a 10.

Constantes:

```
Thread.MAX_PRIORITY;// 10
Thread.MIN_PRIORITY;// 1
Thread.NORM_PRIORITY;// 5
```

Métodos de Thread

- `yield()`

Permite que outras threads de mesma prioridade tenham uma chance de serem executadas.

Instituto de Computação

Sincronização

- é obtida pelo uso da palavra `synchronized` associada a métodos ou blocos
- é possível porque qualquer subclasse de Object tem um **lock** chamado de **monitor**
- quando uma thread obtém este **lock** outras não podem executar o mesmo trecho sincronizado

Demonstração: TestaColisaoThreads.java

Instituto de Computação

Interação entre as threads

- wait() : bloqueia uma thread a espera de uma notificação
- notify() e notifyAll() : fazem a notificação

Mais demonstrações junto com o assunto da próxima aula...

Instituto de Computação

MÓDULO 10 (2ª parte)

Introdução ao Uso de Interfaces Gráficas com o Usuário

Instituto de Computação

AWT e Swing

- Abstract Window Toolkit
  - contém componentes e classes para criação de interfaces gráficas com o usuário
- Swing
  - conjunto de componentes de interface gráfica mais rica que a AWT introduzida a partir da versão 1.2
  - não elimina completamente o uso de AWT

Instituto de Computação

Abrindo Janelas com Swing

- A janela base do Swing é definida pela classe JFrame
- Nela é possível incorporar menus, elementos de diálogo ou mesmo executar primitivas gráficas

Instituto de Computação

Tratamento de eventos

- Baseada no *design pattern* Observer
- é necessário que um objeto seja registrado como listener (ou observador, tratador) dos eventos
- Demonstração: projeto NetBeans

  ProcessaImagem.zip

Instituto de Computação

Desenhando na janela

- Vamos experimentar em ShowColors.java alguns métodos da classe Graphics :
- `setColor, drawString, fillRect, drawRect, drawPolygon e fillPolygon`
- Demonstração: projeto NetBeans

  PongGame2.zip

Exercício LAB

Objetivo:
experimentar a programação de Threads e o básico da interface gráfica.

Roteiro:
faça um programa Java para gerar um novo círculo rebatendo nos limites da janela a partir de cada novo click do mouse. Cada novo círculo terá atributos diferentes (velocX, velocY, cor, raio, etc)

Jogo Squash: faça uma raquete (retângulo) controlável pelas setas do teclado para defender uma "bola" evitando gol (ultrapassar a coordenada Y da raquete)

Jogo Breakout: faça o jogo anterior destruindo retângulos arrumados na parte superior da tela. Você perde se deixar passar 3 bolas. Incremente o jogo com mudanças de direção e velocidade, etc.

Instituto de Computação

MÓDULO 11

Manipulação de Arquivos

Classe StringBuffer

- Ao contrário da classe String, StringBuffer representa um conjunto mutável de caracteres
- Ideal para tarefas de concatenação** ou leitura de arquivos texto.

Ex:

```
public String encriptacaoFraca(String s){
    StringBuffer sb = new StringBuffer();
    for (int i=0;i<s.length();i++){
            char ch = mapa( s.charAt(i) ); // troca chars tabelados num mapa
            sb.append(ch); // concatenação char a char
    }
    return sb.toString();// converte String
}
```

*** OBS: a concatenação de Strings vai gerando novos objetos sendo ineficiente portanto.*

Classes Wrapper

- Permitem tratar tipos primitivos como objetos
- Correspondência:

int	Integer
long	Long
short	Short
byte	Byte

double	Double
float	Float
char	Character
boolean	Boolean

Exemplos de uso:

```
int total = Integer.parseInt("12000"); // extrai int da representação String
String aux = Integer.toHexString(total); // cria string da representação hexadecimal
Double real = Double.valueOf("123.45"); // cria objeto do tipo Double
short numero = Short.MAX_VALUE; // 32767
```

Instituto de Computação

Arquivos e Fluxos

- Java vê cada arquivo como um fluxo seqüencial de bytes
- Independente do sistema de arquivos
- É possível associar fluxos a dispositivos que não arquivos
- Usa pacote java.io

Instituto de Computação

java.io

- É um conjunto grande de classes e interfaces
- As principais são:
  - File
  - InputStream e OutputStream
  - RandomAccessFile
  - Reader e Writer (lidam com caracteres)

File

- abstração para lidar com arquivos e diretórios dos sistemas de arquivos
- útil para navegação e teste de existência destas entidades

Demonstração: abrir projeto "ZIP" classe "EOF"

InputStream e OutputStream

- Lidam com fluxos de bytes
- As subclasses mais comumente usadas para leitura e escrita de arquivos binários são
  - DataInputStream (em conjunto com FileInputStream)
  - DataOutputStream (em conjunto com FileOutputStream)

Demonstração: abrir projeto "ZIP" classe "EOF" (2a parte)

e classe "DataStreams"

RandomAccessFile

- quando houver necessidade de entrada/saída sucessivas e em diferentes posições de um mesmo arquivo
- possui métodos para leitura ou escrita dos tipos de dados mais comumente usados

Demonstração: abrir projeto "ZIP" classe "RandomFileAccess"

classe ZipFile e ZipEntry

- já disponível como pacote java.util.zip
- usa interface Enumeration para obter entradas no ZIP
- OBS: "em Java" o que não estiver já disponível no JDK pode ser comprado ou procurado em pacotes open source

Demonstração: abrir projeto "ZIP" classe "ZIPdemo"

abrir documentação JDK

Exercício LAB 11

Objetivo:
experimentar a manipulação de arquivos.

Roteiro:
suponha que você tenha recebido um arquivo zipado contendo 3 arquivos : nomes.txt telefones.txt e cpfs.dat.

Os arquivos texto nomes e telefones possuem dados de 10 pessoas, um em cada linha. Somente o arquivo cpfs.dat possui o dado na forma binária no formato
XXX.XXX.XXX-XX
sem separação entre os CPFs
i) Gere um arquivo texto de saída SAIDA.TXT contendo os dados de cada pessoa numa mesma linha separados por ponto e vírgula
ii) Repita o ítem i desta vez sabendo que os CPFs foram gerados na ordem inversa (do décimo ao primeiro).
OBS: não abuse da memória prepare seu programa para lidar com arquivos realmente grandes.

Instituto de Computação

MÓDULO 12

Acesso a Bancos de Dados via JDBC

Instituto de Computação

JDBC

- O **Java Database Connectivity** ou **JDBC** é composto por um conjunto de classes e interfaces padrão para conexão a BDs relacionais
- É uma API incluída no pacote java.sql

Instituto de Computação

Vantagens do JDBC

- Mais simples e fácil do que o ODBC
- Seguro e multiplataforma por ser uma API Java

Instituto de Computação

Modelos de aplicação

- Modelo two-tier ou cliente servidor onde a aplicação conversa direto com o servidor
- Modelo three-tier (ou N-tier) onde os comandos SQL são enviados para uma camada intermediária (de serviços) que faz a interação com o banco de dados. Este modelo é mais flexível por permitir maior controle de acesso e também por permitir uma aplicação cliente escrita em mais alto nível.

Instituto de Computação

Drivers JDBC

- Existem 4 tipos de drivers
  - JDBC-ODBC bridge : requer driver ODBC nas máquinas cliente
  - Driver API nativo : converte chamadas JDBC para API nativa requerendo instalação na máquina cliente
  - JDBC Net Driver : é um middleware flexível por traduzir as chamadas JDBC para protocolos de rede independente do SGBD
  - Driver de Protocolo Nativo

  OBS: as duas últimas categorias são totalmente Java

Instituto de Computação

SQLException

- É uma subclasse de Exception
- Possui três métodos relevantes:
  - *getMessage()* retornando uma String descrevendo o problema ocorrido
  - *getErrorCode()* retorna o código de erro retornado pelo SGBD
  - *getSQLState()* retorna a instrução SQL responsável pela exceção ocorrida

URL de conexão

- O JDBC precisa de uma URL para identificar a conexão ao SGBD
- Seu formato geral é composto de 3 partes:

jdbc:<subprotocolo>:<subnome>

A primeira parte é padrão, a segunda é o nome do driver e a terceira a string de identificação do banco de dados

Registro e Conexão

- a forma mais comum de registro do driver JDBC dá-se através do uso de

Class.forName(driver)

- para a conexão propriamente dita, usa-se o método getConnection() de DriverManager

Connection con = DriverManager(URL, user, pwd);

Exemplo

```
String URL = "jdbc:derby://localhost:1527/sample";
String username = "app";
String password = "app";
try {
      Class.forName("org.apache.derby.jdbc.ClientDriver");
      System.out.println("driver carregado");
} catch (Exception e){
      System.out.println("Erro na carga do driver");
      System.exit(0);
}

Statement stmt = null;
Connection conexao = null;
try{
      conexao = DriverManager.getConnection(URL,username,password);
      System.out.println("Conexao criada");
      stmt = conexao.createStatement();
      System.out.println("Statement criado");
} catch (Exception e){
      System.out.println("problemas de conexao com o banco");
}
```

Instituto de Computação

Interface Connection

- Representa a conexão com o BD
- Usada na criação de comandos SQL
- Dá suporte a transações
- Permite acesso a metadados via método getMetaData()

prática: vamos ver agora o exemplo JdbcDataBaseMetaData.java

Instituto de Computação

Execução de comandos SQL

Veremos a seguir:

- Interface Statement
- Método executeQuery()
- Interface ResultSet
- Método executeUpdate()
- Método execute()
- Interface PreparedStatement
- declaração e uso de stored procedures

Interface Statement

- Envia comandos SQL para o BD
- **Statement stmt = conexao.createStatement();**
- O objeto Statement criado necessita dos métodos
  - executeQuery()
  - executeUpdate()
  - execute()

  para enviar comandos SQL simples (não parametrizados) ao SGBD

Método executeQuery()

- Usa-se em consultas retornando ResultSet

Exemplo:

```
ResultSet rs =
        stmt.executeQuery("select * from aluno");
```

Obs: uma nova consulta usando o objeto stmt fecha o result set anterior

Interface ResultSet

- Utilizada para recuperação dos resultados de uma consulta
- Usa o conceito de cursor apontando para o registro atual e o método **next( )** para andar para o próximo
- Os métodos getXXX( ) são usados para recuperação dos tipos de dados (conversão entre tipo SQL e Java)

Exemplo

```
...
while (rs.next() ){
    double salario = rs.getDouble(1); // pode-se recuperar pela posição
    if (rs.wasNull( )  ||  salario < SALARIO_MINIMO){
        // testa para saber se o campo era nulo ou continha valor invalido
        salario = SALARIO_MINIMO;
    }
    String professor = rs.getString("PROFESSOR");// recuperação pelo nome
}
// OBS: após uso fechar ResultSet, Statement e conexao
rs.close();
stmt.close();
conexao.close();
...
```

Método executeUpdate()

- Usa-se em consultas que modifiquem linhas nas tabelas (INSERT, UPDATE, DELETE) ou ainda nos comandos DDL (data definition language)

Exemplo:

```
int numeroRemovidos =
        stmt.executeUpdate("delete from aluno");
```

Obs: o inteiro retornado refere-se ao número de registros alterados ou zero como é o caso da DDL a seguir: "CREATE TABLE aluno ..."

prática: vamos ver agora os exemplos JdbcBasico.java e JdbcTransaction.java

Método execute()

- Usa-se em consultas que retornem mais de um ResultSet ou mais de um indicador de linhas afetadas

Exemplo:

```
boolean gerouResultSet =
        stmt.execute(StringSQL);
```

Obs: o valor true retornado indica que o resultado é um ResultSet (obtido com getResultSet()) ou um contador de linhas (obtido com getUpdateCount()). O método getMoreResults() posiciona o statement para a recuperação do próximo conjunto eventualmente retornado.

Interface PreparedStatement

- filha de Statement
- Usa declarações SQL pré-compiladas
- Tem-se assim, eficiência em seu reuso com simples alteração de parâmetros (sinal de interrogação)

Exemplo:

```
PreparedStatement ps =
        conexao.prepareStatement
("update aluno set mensalidade=? where nome=? ");
```

Interface PreparedStatement

- Após a preparação, resta passar os valores adequados (com métodos setXXX) aos parâmetros

```
ps.setDouble(1, 280.0);
ps.setString(2, "Ze");
```

- Feito isto, executa-se, conforme o caso, o comando com dos métodos abaixo:

```
ps.executeUpdate();
ps.executeQuery();
```

prática: vamos ver agora o exemplo JdbcPreparedStatement.java

Demonstração: Criação e Uso de Stored Procedures

Iremos mostrar a codificação de stored procedures no servidor com o exemplo:

StoredProceduresProfessorDisciplina.java

Também sua declaração e uso no ambiente do SGBD:

criaStoredProcedureProfessorDisciplina.sql

e sua chamada nos programas

CallSpProfessorDisciplina.java e JdbcCriaSP.java

Exercício LAB

Objetivo:
experimentar a programação para BDs relacionais
Roteiro:
Combinar um projeto com a turma.
Criar tabelas e seus relacionamentos. Codificar uma classe com métodos para consulta, inserção, atualização e exclusão.

Instituto de Computação

MÓDULO 13

Servlets

Instituto de Computação

O que são Servlets?

- são pequenos programas Java rodando no contexto de um servidor web
- implementam a interface Servlet
- recebem e respondem a solicitações de clientes web
- em geral, ao falar de Servlets estamos interessados em Servlets HTTP

Instituto de Computação

Servlets HTTP

- Estendem a classe abstrata HTTPServlet
- Há processamento de pedidos concorrentes em múltiplas threads
- As instâncias de um servlet são reutilizadas quando for necessário.
- A resposta é rápida e eficiente

Instituto de Computação

Tratamento de POST e GET

- Executado por meio dos métodos doGet() e doPost()
- Os dois métodos recebem como parâmetros:
  - HttpServletRequest
  - HttpServletResponse

Classe HttpServletRequest

- Mantém as informações sobre o cliente solicitante:
  - parâmetros passados
  - nome do host que fez o pedido
  - demais dados de entrada
- Métodos
  - getRemoteHost()
  - getServerName()
  - getParameterValues(String name)
  - getParameter(String name)
  - etc

Classe HttpServletResponse

- Guarda a resposta do servlet:
  - tamanho
  - MIME-types
  - dados de saída
- Métodos
  - setContentLength(int length)
  - setContentType(String mimeType)
  - getWriter()

Instituto de Computação

Exemplos

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">
<html>
    <head>
    <meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
    <title>WWF Web Workflow</title>
    </head>
<body>
<form name="login" action="LoginServlet" method="post">
    nome<input type="text" name="loginName" maxlength=6><br/>
    senha<input type="password" name="loginPassword" maxlength=6><br/>
    <input type="submit" name="submit" value="ok">
</form>
</body>
</html>
```

Exemplos

```
public class LoginServlet extends javax.servlet.http.HttpServlet implements
      javax.servlet.Servlet {

        protected void doGet(HttpServletRequest request, HttpServletResponse response) throws
            ServletException, IOException {
            PrintWriter out = response.getWriter();
            response.setContentType("text/html");   //gera saída html
            out.println("<html><body>");
            out.println("bem vindo "+ request.getParameter("loginName"));
            out.println("</body></html>");
        }

        protected void doPost(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
            doGet(request, response);
        }
}
```

Gerenciamento de Sessão

- Não há estado preservado numa conexão http
- É necessário para qualquer aplicação web saber se um usuário é ou não válido, que ítens ele/ela já selecionou e que perguntas já respondeu.
- Portanto, torna-se necessário a existência de um mecanismo de sessão para garantir a unicidade de cada usuário

Gerenciamento de Sessão

- A criação de uma sessão e de variáveis de sessão é possível por meio dos comandos

```
HttpSession ss = request.getSession();
ss.setMaxInactiveInterval(20);
// 20 segundos no máximo de inatividade
ss.setAttribute("usuario", loginName);
```

Gerenciamento de Sessão

- Em cada página da aplicação podemos testar se a sessão existe ou não

```
String usuario = (String) ss.getAttribute("usuario");
if (usuario != null )
    welcome(request, response, "sessao ok");
else
    response.sendRedirect("login.jsp");
```

Cookies

- Cookies são dados escritos por um website deixados na máquina local do usuário visando sua identificação posterior
- Uma utilidade de cookies é guardar as preferências do usuário
- Em geral, o mecanismo de gerenciamento de sessão faz uso de cookies para guardar um identificador único por usuário.

Escrevendo cookies

```
if (lembrarUsuario != null){
   System.out.println("evita ter que redigitar senha");
   Cookie user = new Cookie("user", loginName);
   Cookie pwd = new Cookie("pwd", senha);
   user.setMaxAge(10*60);//cookie válido por 10 mins
   pwd.setMaxAge(10*60);
   response.addCookie(user);
   response.addCookie(pwd);
}
```

Problemas com os Servlets

- O único grande problema com Servlets é o fato de que não é prático escrever uma página web completa dentro de um out.println:

```
PrintWriter out = response.getWriter();
response.setContentType("text/html");   //gera saída html
out.println("<html><body bgcolor=\"orange\">");
out.println("Seja Bem Vindo "+usuario+"<br/><br/>"+modo);
out.println("<br/><br/><a
   href=\"LoginServlet?logout=ok\">clique para sair da
   aplicação</a>");
out.println("</body></html>");
```

- Surgem então as ...

MÓDULO 14

Java Server Pages (JSP)

Instituto de Computação

Criando um JSP

- Mude a extensão de uma página HTML qualquer para .jsp
- Coloque-a em um servidor capaz de processar JSPs.
- Está criada assim sua primeira página JSP
- Internamente o servidor converte o JSP para um Servlet sem que o usuário se preocupe com isto
- A idéia é ter uma página HTML com comandos Java.
- Isto é o oposto da idéia de Servlets

Instituto de Computação

Objetos Implícitos

- Alguns objetos já estão instanciados e prontos para usar numa página JSP:
  - out
  - request
  - response
  - application entre outros

Expressions

- Código disposto entre <%= e %>

```
<html>
<body>
Olá!  A hora atual é <%= new java.util.Date() %>
</body>
</html>
```

Scriptlets

- Código Java colocado entre <% e %>

```
<html>
<body>
<%
    // Este é um scriptlet.  Observe que a variável "date"
    // pode ser usada em outro trecho na mesma página.
    System.out.println( "debug: data hora com scriptlet" );
    java.util.Date date = new java.util.Date();
%>
olá!  A data agora é <%= date %>
<%
// outra scriptlet usando o objeto "out" já disponível
    out.println( "<br/><br/>o endereço de sua máquina é " );
    out.println( request.getRemoteHost());
%>
</body>
</html>
```

Mais scriptlets

```
<body>
<%
     String linhas = request.getParameter("linhas");
     int numeroLinhas = 0;
     try {
        numeroLinhas = Integer.parseInt(linhas);
     } catch (Exception e){
        System.out.println("deu erro");
     }
     if (numeroLinhas <= 0){
        out.println("experimente passar o parâmetro linhas=3 para esta
        página");
     }else {
        out.println("<table border=2>");
        for(int i=0;i<numeroLinhas;i++){
%>
          <tr><td><%= "linha" %></td><td><%= i %></td></tr>
<%
        }
        out.println("</table>");
     }
%>
</body>
```

Diretivas

- page directive: para importação de pacotes

```
<%@ page import="java.util.*,java.text.*" %>
```

- include directive: observe que funciona como cut & paste de código

```
<html><body> inclui a página...<br/>
<%@ include file="scriptlet.jsp" %>
</body></html>
```

Instituto de Computação

Processamento de Forms

- O uso de formulários é algo quase que obrigatório em qualquer aplicação web
- JSP fornece um meio de trabalhar com formulários por meio de um JavaBean
- Veja o exemplo ***form.jsp*** que faz uso de uma classe Java (um JavaBean) para coletar os dados de um formulário

Usando um Bean

- ```
  <jsp:useBean id="user"
      class="user.UserData"
      scope="session"/>
  ```
- No exemplo acima ***id*** define o nome da instância; ***class*** o nome completo (incluindo pacote) da classe; ***scope*** define o escopo (session, page, request ou application)
- vide exemplo (***form.jsp***)
  ```

Instituto de Computação

Considerações ao usar JSP

- Da mesma forma que não é conveniente colocar muito código HTML dentro de um servlet, também se torna muito confuso e de manutenção custosa incorporar muito código Java a um JSP
- A solução para isto está no uso de bibliotecas de tags (taglibs fora do escopo de nosso curso) e
- O uso do padrão MVC

Instituto de Computação

Padrão MVC

- Model View Controler é um padrão de projeto que visa definir, como o nome indica, responsabilidades, ou seja, que módulos irão tratar da representação do modelo, da interface com o usuário (view) e quem faz o papel de controlador
- Em nosso exemplo usaremos um controle de "login" para validar o acesso à nossa aplicação
- A página JSP fará o papel de View, uma classe Java irá representar o modelo do usuário que se deseja validar e um servlet irá controlar todo o processo

Instituto de Computação

Demonstração

- projeto _MySite para controle de acesso
- http://localhost:8080/_MySite

Exercício LAB

Objetivo:
Exercitar o uso de bancos de dados em conjunto com Servlets e JSP

Roteiro:
Criar um aplicativo web de abertura de ordens de serviço DE-PARA e DESCRIÇÃO. Cada usuário poderá ver as ordens por ele abertas ou a ele encaminhadas. Cada destinatário poderá fechar uma ordem e/ou alterar sua descrição. Somente uma descrição simples será feita e modificada.

Sugestão para os alunos: transforme o exercício num aplicativo de workflow mais completo permitindo o andamento de um pedido entre os usuários que irão adicionar descrição do que foi feito e re-encaminhar as ordens para outros usuários. O solicitante poderá fechar a ordem assim que considerar a tarefa cumprida. A cada passo o sistema deverá adicionar data/hora e nome do usuário.

Instituto de Computação

Programação Orientada a Objetos com Java

Prof. Lauro Eduardo Kozovits, D.Sc.

Departamento de Computação, UFF