Programação Orientada a Objetos
(Object Oriented Programming)

FLÁVIO DE OLIVEIRA SILVA
flavio@facom.ufu.br
flavio@autoenge.com.br

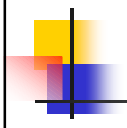
1

OBJETIVOS DA DISCIPLINA

- Entender os principais conceitos do paradigma de Orientação a Objetos e sua importância no processo de desenvolvimento de software
- Estudar uma linguagem de programação que suporte os conceitos de OOP (Java)

EMENTA

1. Introdução à POO
2. Análise Orientada a Objetos
3. Introdução à Linguagem Java
4. Classes, Atributos e Métodos
5. Herança
6. Herança Múltipla
7. Aspectos de Implementação de LOO



BIBLIOGRAFIA

POO, Danny C. C. Object-Oriented Programming and Java; 3ed, Springer, 2001

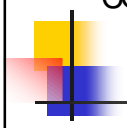
WINBLAD, Ann L. Software Orientado ao Objeto; Makron Books, 1993

DEITEL, H. M. Java como Programar; 3 ed, Bookman, 2001

BOOCH, Grady Object-Oriented Analysis and Design with Applications; 2ed, Benjamin, 1994

KHOSHAFIAN, Setrag Object Orientation: Concepts, languages, database, user interfaces; Wiley, 1990

COAD, Peter Análise Baseada em Objetos; Campus, 1992.



AVALIAÇÃO

- Provas – Teóricas – 50 pontos
  - Primeira Prova - 10/06/2003 – 25 pontos
  - Segunda Prova - 22/07/2003 – 25 pontos
- Testes em sala de Aula – 10 pontos
- Trabalho – Projeto Conclusão de Curso – 40 pontos
  - Apresentação: Relatório e Entrevista de avaliação
  - Critérios: Complexidade; Finalização da proposta; Respeito ao Cronograma; Programação em camadas; Entrevista Individual; Relatório
  - Partes do projeto serão entregues em diferentes datas

OBSERVAÇÕES

- Material Apoio
  - Pasta 122 – Xerox ao lado Bloco B
- Atendimento
  - Terça-Feira 08:50 – 10:40 – Sala 1B54
  - Sexta-Feira 15:00 – 16:30 – Sala 1B54
- Aulas Práticas no laboratório

INTRODUÇÃO A POO

- A Programação Orientada a Objetos (POO) é um paradigma Baseado em objetos
- Paradigma – Modelo, padrão para especificação de um problema
- Paradigmas existentes na programação de computadores: Procedimental; Funcional; Lógico e Orientado a Objetos(OO)

PROCEDIMENTAL (Algorítmica)

- Computação ocorre através da execução de instruções passo a passo. Ex: C; Pascal

```
int fatorial(int n){
   int fat;
   fat = 1;
   if (n == 0)
     return 1;
   while (n >= 1){
     fat = fat * n;
     n = n -1;
   }
   return fat;
}
```


FUNCIONAL

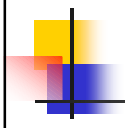
- Computação baseada em cálculo de funções. Ex: LISP; HASKELL

```
;LISP
(defun fatorial (n)
   (cond
     ((= n 0) 1)
     (t (* n (fatorial (- n 1))))
   )
)
-- Haskell
fatorial :: Integer -> Integer
fatorial 0   = 1
fatorial n   = n * fatorial (n - 1)
```

LÓGICA

- Computação baseada em fatos e regras. Ex: Prolog

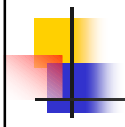
```
factorial(0,1).
factorial(N,X):- M is N - 1,
                 factorial(M,Y),
                 X is N * Y.
```



ORIENTADO A OBJETOS (OO)

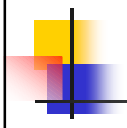
- Computação baseada em objetos que se intercomunicam. Objetos contém dados e métodos. Ex: C++; Java

```
public class Numero {
  private long valor = 0;
  public long Fatorial() {
    if (valor == 0) {
      return 1;
    } else {
      return valor*Fatorial(valor-1);
    }
  }
}
```



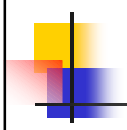
PARADIGMAS

- Cada paradigma representa uma forma de propor uma solução a um dado problema.
- A utilização de um ou outro paradigma depende de vários fatores.



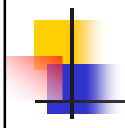
INTRODUÇÃO A POO
Conceitos Básicos

1. OBJETO
2. MÉTODO
3. MENSAGEM
4. CLASSE
5. CLASSIFICAÇÃO
    1. GENERALIZAÇÃO
    2. ESPECIALIZAÇÃO
6. HERANÇA
7. ENCAPSULAMENTO
8. POLIMORFISMO



OBJETO

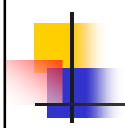
- Entidades que possuem **dados** e **instruções** sobre como manipular estes dados
- Os objetos estão ligado à solução do problema.
  Software Gráfico – Objetos: Circulos; Linhas; etc.
  Software BD – Objetos: Tabelas; Linhas; Campos; etc.
  Software Comercial: Pedidos; Produtos; Clientes; etc.
- Na POO a solução do problema consiste em um primeiro momento estabelecer quais os objetos serão necessários.



OBJETO

- Dados ligados ao objeto – Exemplos:
  Círculo – ponto_centro, raio
  linha – ponto_inicio; ponto_final
  Cliente – Nome; Endereco; Telefone
  Exemplos:

```
//criação de objetos (sintaxe C++)
Ponto p(3,4);
Circle c(p,5.4);
Cliente pessoa;
```

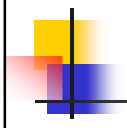


MÉTODO

- Métodos são procedimentos que determinam como o objeto se comporta. Através dos métodos é possível manipular os dados contidos no objeto.
- Os métodos estão ligados ao comportamento do objeto

  Exemplo - Um círculo poderia possuir os métodos:

  draw; move; getArea; getPerimeter; setCenter



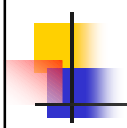
MENSAGEM

- Objetos se comunicam entre si através de mensagens.
- Uma mensagem é uma chamada de um método. A mensagem possui os seguintes componentes:

  Receptor – nome do objeto que irá receber a mensagem
  Método – Método do receptor que será utilizado
  Argumentos – Informação adicional para a execução do método

  Exemplos

```
Point p(0,0), pNewCenter(2,3);
Circle c(p,3);
c.getArea();                  //Exemplo Mensagem
c.setCenter(pNewCenter);      //Exemplo Mensagem
```



CLASSE

- Classe é um agrupamento de objetos
- A classe consiste nos métodos e nos dados que um determinado objeto irá possuir.
- Objetos são criados quando uma mensagem solicitando a criação é recebida pela sua classe.
- A programação orientada a objetos consiste em implementar as classes e na utilização das mesmas, através da sua intercomunicação.
- Um objeto é uma instância da classe.

CLASSIFICAÇÃO

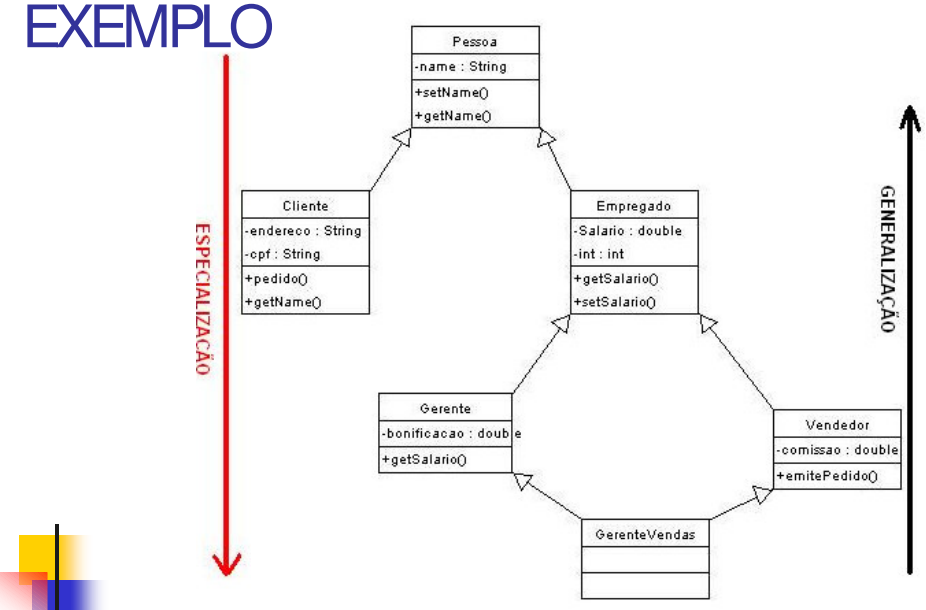
- Na POO classificação consiste em criar classes a partir dos objetos envolvidos em um determinado problema
- Ex: Diferentes tipos de pessoas interagem com um Empresa

COMPORTAMENTO	CLASSE
Pessoas interessadas nos produtos	???
Pessoas que já compraram os produtos	???
Pessoas que são responsáveis por um grupo de trabalhadores	???
Pessoas responsáveis pela demonstração de produtos e sua venda	???
Trabalhadores da linha de produção	???

GENERALIZAÇÃO
ESPECIALIZAÇÃO

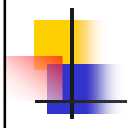
- GENERALIZAÇÃO
  - A generalização consiste em obter similaridades entre as várias classes e partir destas similaridades, novas classes são definidas.
  - Estas classes são chamadas **superclasses**
- ESPECIALIZAÇÃO
  - A especialização por sua vez consiste em observar diferenças entre os objetos de uma mesma classe e dessa forma novas classes são criadas.
  - Estas classes são chamadas **subclasses**.

EXEMPLO



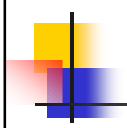
HERANÇA

- Herança é a capacidade de uma subclasse de ter acesso as propriedades da superclasse a ela relacionada.
- Dessa forma as propriedades de uma classe são propagadas de cima para baixo em um diagrama de classes.
- Neste caso dizemos que a subclasse herda as propriedades e métodos da superclasse
- A relação de herança entre duas classes é uma relação da seguinte forma: A "e um tipo de" B, onde A e B são classes. Caso esta relação entre as classes não puder ser construída, em geral, também não se tem uma relação de herança entre a classe A a partir da classe B.



HERANÇA

- Exemplos: Um Carro de Passeio "é um tipo de " veículo; Um caminhão "é um tipo de" veículo; Um círculo "é um tipo de" uma figura geométrica; Um quadrado "é um tipo de" figura geométrica; Um vendedor "é um tipo de" Empregado; Um empregado "e um tipo de" pessoa.
- Herança Múltipla – Uma subclasse herda características de mais uma classe
- Exemplos: Um gerente de vendas "é um tipo" de vendedor e "é um tipo de" gerente;



HERANÇA x USO

- Além da relação de herança entre as classes existe a relação de uso
  - HERANÇA

    classe A "é um tipo de" B
  - USO / AGREGAÇÃO (Relação de Conteúdo)

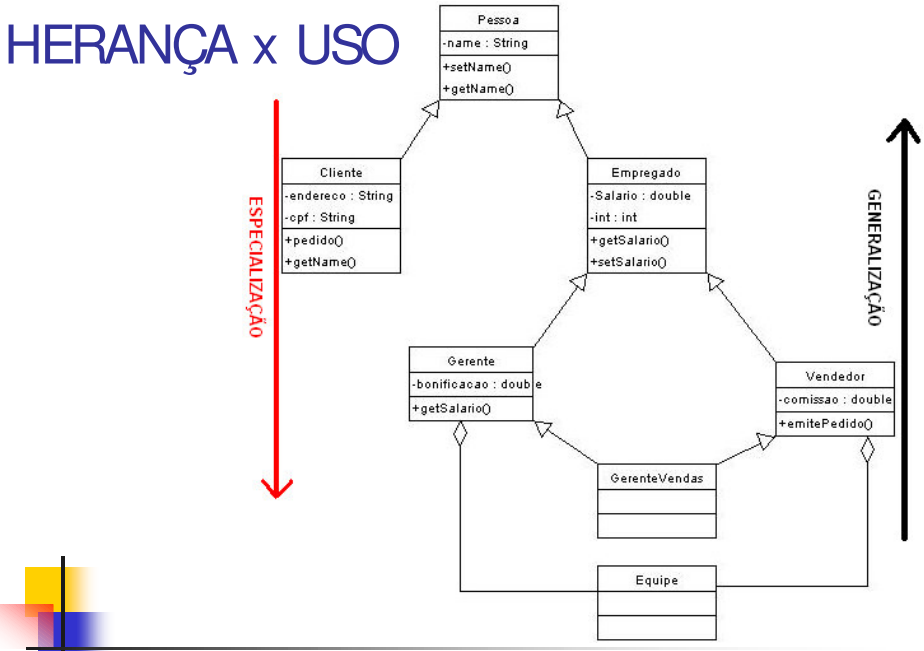
    classe D "contém" classe C"

    classe D "usa" classe C"

    classe C "é parte da" classe D

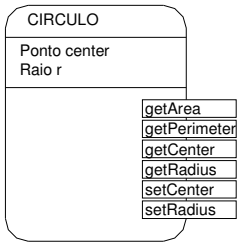
  Exemplo: Uma **equipe** contém um **gerente** e um grupo de **vendedores**

HERANÇA x USO



ENCAPSULAMENTO

- Encapsulamento é um termo que indica que os dados contidos em um objeto somente poderão ser acessados através de seus métodos.
- Dessa forma não é possível alterar os dados diretamente, somente através de métodos. Ex: O raio somente pode ser alterado/recuperado pelos métodos setCenter/getCenter.



POLIMORFISMO

- Os objetos respondem às messagens que eles recebem através dos métodos. A mesma mensagem pode resultar em diferentes resultados. Esta propriedade é chamada de **polimorfismo**

  Exemplo: Método getSalario()

  Para um empregado qualquer → getsalario() = Salario;

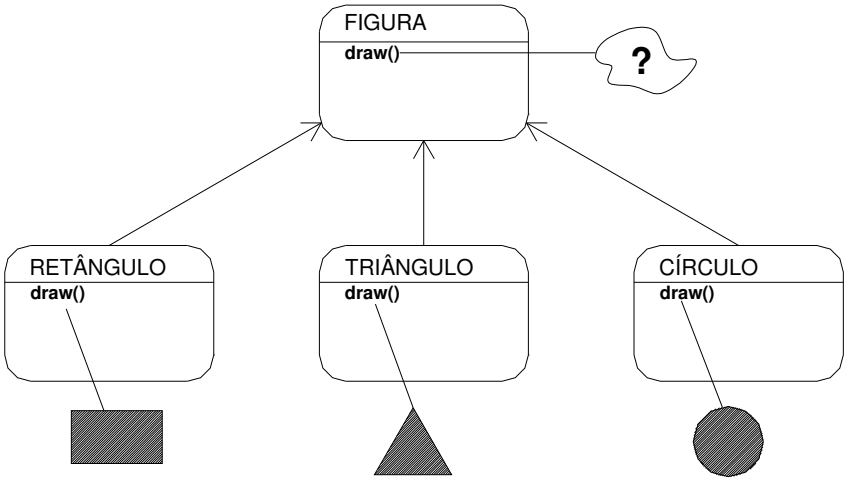
  Para o gerente → getsalario() = salario + bonificacao;

  Exemplo: Método draw()

  Para uma figura qualquer desenha uma forma não definida

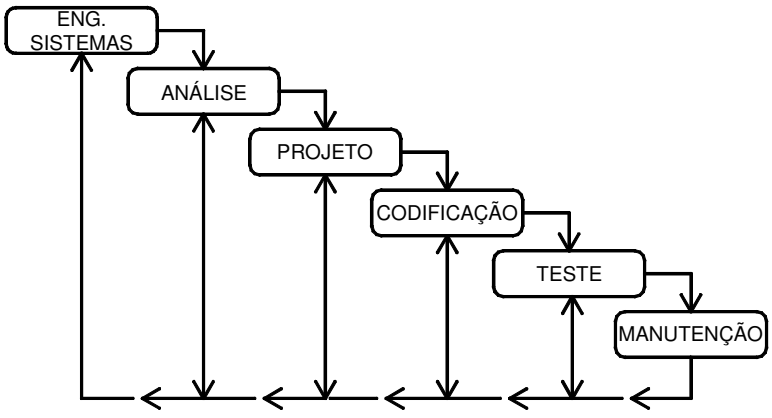
  Para o retângulo, triângulo e círculo o mesmo método responde de uma forma diferente

POLIMORFISMO - Exemplo



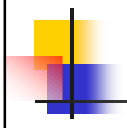
ENGENHARIA DE SOFTWARE

- CICLO DE VIDA DA ENG. SOFTWARE (Mod. Clássico)



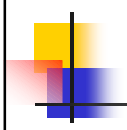
ENGENHARIA DE SOFTWARE

- **ENGENHARIA DE SISTEMAS**
  - Levantamento dos requisitos
  - Inserir o sistema em um contexto maior – Hardware; Pessoas; Outros sistemas
  - Visão geral e ampla do sistema
  - Riscos; Custos; Prazos; Planejamento



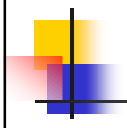
ENGENHARIA DE SOFTWARE

- **ANÁLISE**
  - Continua o processo de coleta de requisitos, porém concentra-se no âmbito do software
  - Modelos – Dados; Funções e comportamentos
  - Particionamento do problema
  - Documentação e Revisão dos requisitos
    - ANÁLISE ESTRUTURADA – DFD
    - ANÁLISE ORIENTADA A OBJETOS – DIAGRAMA DE CLASSES



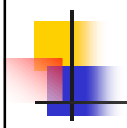
ENGENHARIA DE SOFTWARE

- **PROJETO**
  - "Como" o software irá executar os requisitos
  - Estrutura de dados; Arquitetura do Software; Detalhes de execução; caracterização da interface
  - Produzir um modelo que permita a sua construção posterior
    - PROJETO ESTRUTURADO – Módulos
    - PROJETO ORIENTADO A OBJETOS – Atributos; Especificacão dos Métodos; Mensagens



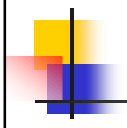
ENGENHARIA DE SOFTWARE

- **CODIFICAÇÃO**
  - "Traduzir" o projeto para uma linguagem de computador
  - Projeto detalhado pode levar a uma codificação mecânica (Ferramenta CASE)
- **TESTES**
  - Verificação se o código atende aos requisitos
  - Aspectos lógicos e internos do software – Teste de todas as instruções
  - Aspectos funcionais externos – entrada produz o resultado esperado



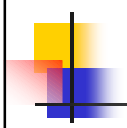
ENGENHARIA DE SOFTWARE

- **MANUTENÇÃO**
  - Mudanças necessárias após a entrega ao cliente
  - Mudanças → Erros; Alteração ambiente externo ou alteração especificação de requisitos.



ANÁLISE ORIENTADA A OBJETOS (OOA)

- Objetivo básico – Identificar classes a partir das quais objetos serão representados como instâncias
- Envolve as seguintes tarefas
  - Identificação de Objetos
  - Especificação de Atributos
  - Definição de métodos
  - Comunicações entre objetos



ANÁLISE ORIENTADA A OBJETOS (OOA)

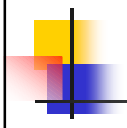
- IDENTIFICAÇÃO DE OBJETOS
  - Entidades externas (Outros sistemas; dispositivos; Pessoas)
  - Coisas ligadas ao domínio do problema (Relatórios; Displays;...)
  - Ocorrências ou Eventos (Conclusão de um movimento; Alarme disparado; Clique do mouse; etc.)
  - Papéis ou funções (Engenheiro; Gerente; Vendedor) desempenhados por pessoas
  - Unidades organizacionais (Grupo; Equipe;...)
  - Lugares (Piso de fábrica; área de descarga)
  - Estruturas (Sensores; veículos de quatro rodas;...)

ANÁLISE ORIENTADA A OBJETOS (OOA)

- IDENTIFICAÇÃO DE OBJETOS – CRITÉRIOS
  1. RETENÇÃO DE INFORMAÇÃO – Objeto deve guardar informação que será utilizada pelo sistema
  2. SERVIÇOS NECESSÁRIOS – Conjunto de operações identificáveis que podem mudar o valor de seus atributos
  3. MÚLTIPLOS ATRIBUTOS – Objeto deve conter mais de um atributo
  4. ATRIBUTOS COMUNS – Conjunto de atributos deve ser aplicado a todos os objetos
  5. OPERAÇÕES COMUNS – Conjunto de operações devem ser aplicáveis a todos os objetos.
  6. REQUISITOS ESSENCIAIS – Entidades externas que aparecem no espaço problema que consomem e/ou produzem informação

ANÁLISE ORIENTADA A OBJETOS (OOA)

- Em uma especificação:
  - NOMES são potenciais objetos
  - VERBOS são potenciais métodos



ANÁLISE ORIENTADA A OBJETOS (OOA)

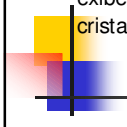
EXEMPLO:

O software SafeHome possibilita que o dono da casa configure o sistema de segurança quando ele for instalado, monitora todos os sensores ligados ao sistema de segurança e interage com o dono da casa através de um teclado (key pad) e teclas de função contidas no painel de controle do SafeHome.

Durante a instalação o painel de controle é usado para "programar" e configurar o sistema. A cada sensor é atribuido um número e um tipo, uma senha mestra é programada para armar e desarmar o sistema e números telefônicos são introduzidos para serem discados quando ocorrer um evento sensor.

Quando um evento sensor é sentido pelo software, ele dispara um alarme sonoro ligado ao sistema. Após um tempo de espera, que é especificado pelo dono da casa durante as atividades de configuração do sistema, o software disca um número telefônico do serviço de monitoração, oferece informações sobre o local, registrando a natureza do evento que foi detectado. O número será novamente discado a 20 segundos até que a ligação telefônica seja completada.

Todas as interações com o SafeHome são gerenciadas por um subsistema de interação com o usuário, que lê a entrada fornecida através do teclado e das chaves de função, exibe mensagens de prompting e informações sobre o status do sistema no mostrador de cristal líquido (LCD). A interação com o teclado assume a seguinte forma...



ANÁLISE ORIENTADA A OBJETOS (OOA)

EXEMPLO:

O software SafeHome possibilita que o ***dono da casa*** configure o **sistema de segurança** quando ele for instalado, monitora todos os ***sensores*** ligados ao **sistema de segurança** e interage com o ***dono da casa*** através de um ***teclado*** e ***teclas de função*** contidas no ***painel de controle*** do SafeHome.

Durante a instalação o ***painel de controle*** é usado para "programar" e configurar o sistema.

A cada ***sensor*** é atribuido um ***número*** e um ***tipo***, uma ***senha mestra*** é programada para armar e desarmar o sistema e ***números telefônicos*** são introduzidos para serem discados quando ocorrer um ***evento sensor***.

Quando um ***evento sensor*** é sentido pelo software, ele dispara um ***alarme sonoro*** ligado ao sistema. Após um ***tempo de espera***, que é especificado pelo ***dono da casa*** durante as atividades de configuração do sistema, o software disca um **número telefônico** do ***serviço de monitoração***, oferece ***informações sobre o local***, registrando a ***natureza do evento*** que foi detectado. O número será novamente discado a 20 segundos até que a ligação telefônica seja completada.

Todas as interações com o SafeHome são gerenciadas por um ***subsistema*** de interação com o ***usuário***, que lê a ***entrada*** fornecida através do ***teclado*** e das ***chaves de função***, exibe ***mensagens*** de prompting e informações sobre o status do ***sistema*** no ***mostrador de cristal líquido*** (LCD). A interação com o teclado assume a seguinte forma...

ANÁLISE ORIENTADA A OBJETOS (OOA)

EXEMPLO - IDENTIFICAÇÃO DE OBJETOS	Classificação Geral
dono da casa	Papel ou entidade externa
sistema de segurança	Coisa
sensores	*Entidade externa*
teclado	*Entidade externa*
teclas de função	*Entidade externa*
painel de controle	*Entidade externa*
número	*Atributo do sensor*
Tipo	*Atributo do sensor*
senha mestra	*Coisa*
números telefônicos	*Coisa*
evento sensor	*Ocorrência*
alarme sonoro	*Entidade externa*
tempo de espera	*Atributo do sistema*
Serviço de monitoração	*Unidade Organizacional ou Ent. Externa*
informações sobre o local	*Atributo do sistema*
natureza do evento	*Atributo do sistema*
subsistema	*Entidade externa*
entrada	*Entidade externa*
chaves de função	*Entidade externa*
mensagens	*Entidade externa*
mostrador de cristal líquido	*Entidade externa*

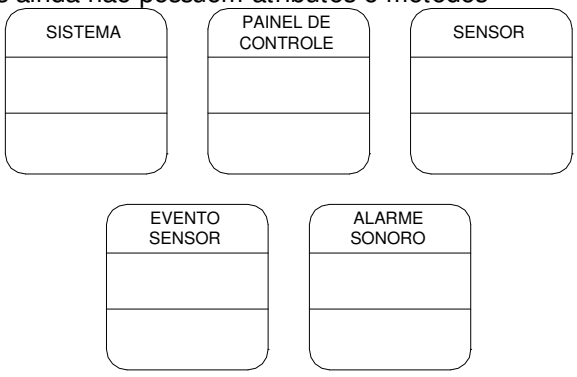
ANÁLISE ORIENTADA A OBJETOS (OOA)

EXEMPLO - IDENTIFICAÇÃO DE OBJETOS	Características
dono da casa	Rejeitado: 1 e 2 falham embora 6 se aplique
sistema de segurança	Aceito: Todas se aplicam
sensores	Aceito: Todas se aplicam
teclado	Aceito: Todas se aplicam
teclas de função	Aceito: Todas se aplicam
painel de controle	Aceito: Todas se aplicam
número	Rejeitado: 3 falha
Tipo	Rejeitado: 3 falha
senha mestra	Rejeitado: 3 falha
números telefônicos	Rejeitado: 3 falha
evento sensor	Aceito: Todas se aplicam
alarme sonoro	Aceito: 2, 3, 4, 5 e 6
tempo de espera	Rejeitado: 3 falha
Serviço de monitoração	Rejeitado: 1 e 2 falham embora 6 se aplique
informações sobre o local	Rejeitado: 3 falha
natureza do evento	Rejeitado: 3 falha
subsistema	Aceito: Todas se aplicam
entrada	Rejeitado: 3 falha
chaves de função	Aceito: Todas se aplicam
mensagens	Aceito: Todas se aplicam
mostrador de cristal líquido	Aceito: Todas se aplicam

ANÁLISE ORIENTADA A OBJETOS (OOA)

EXEMPLO - IDENTIFICAÇÃO DE OBJETOS

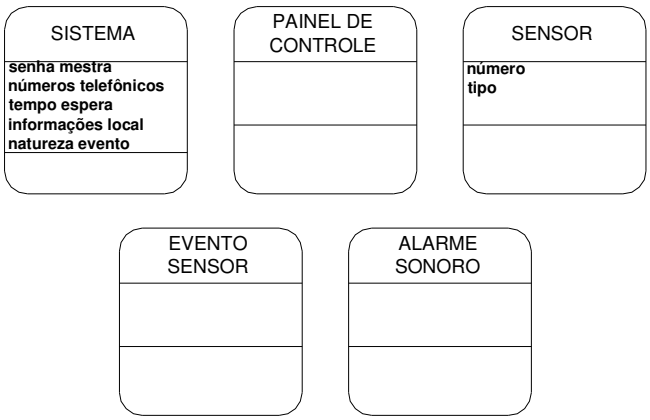
- Os objetos do painel de controle serão considerados separadamente.
- Os objetos abaixo são o ponto de partida para o desenvolvimento do sistema
- Objetos ainda não possuem atributos e métodos



ANÁLISE ORIENTADA A OBJETOS (OOA)

EXEMPLO – ESPECIFICAÇÃO DE ATRIBUTOS

- Necessário estudar e analisar a descrição do sistema
- Pergunta Importante – "Quais informações definem este objeto?"

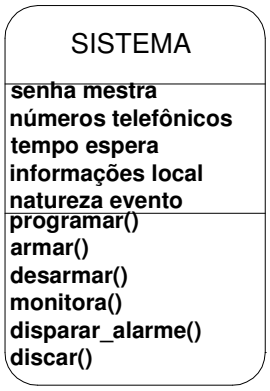


ANÁLISE ORIENTADA A OBJETOS (OOA)

EXEMPLO – DEFINIÇÃO DE MÉTODOS

- Necessário estudar e analisar a descrição do sistema
- Verbos são potenciais operações

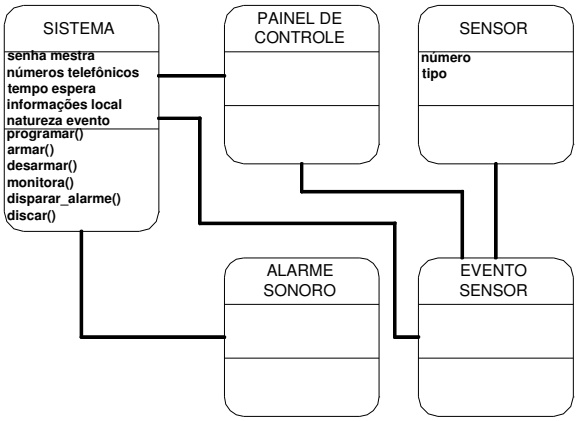
Configure
Instalado
Monitora
Ligados
Interage
Instalação
"programar"
configurar
programada
armar
desarmar
introduzidos
serem discados
ocorrer
sentido
dispara
especificado
configuração
disca
registrando
detectado
Discado
ligação
interações
gerenciadas
interação
lê
fornecida
exibe



ANÁLISE ORIENTADA A OBJETOS (OOA)

EXEMPLO – COMUNICAÇÕES ENTRE OBJETOS

- Representa as comunicações entre os vários objetos.
- Contém todos os caminhos de mensagens entre os objetos



ANÁLISE ORIENTADA A OBJETOS (OOA)

EXEMPLO – PROBLEMA PROPOSTO

O departamento de obras públicas da cidade de Uberlândia decidiu desenvolver um sistema de computador para rastreamento e conserto de buracos de rua (SIRCOB). À medida que são registrados buracos de rua, eles recebem um número de identificação e são armazenados de acordo com o endereço da rua, tamanho (numa escala de 0 a 10), localização (no meio da rua; na calçada; etc.), bairro (determinado a partir do endereço da rua) e prioridade de reparo (determinada a partir do tamanho do buraco). Dados de ordem de trabalho são associados a cada buraco, e eles incluem localização e tamanho do buraco, número de identificação da equipe de reparos, número de pessoas na equipe, equipamentos designados, horas aplicadas ao reparo, status do trabalho (em andamento, concluído, não concluído), quantidade de material de enchimento usado e custo do reparo (computado a partir das horas trabalhadas, número pessoas, material e equipamentos usados). Finalmente, um arquivo de danos ocorridos é criado para guardar informações sobre danos registrados devido ao buraco, o qual inclui o nome do cidadão, endereço, número telefônico, tipo de dano e a quantia em reais a ser paga. O SIRCOB é um sistema on-line; as consultas devem ser feitas interativamente.

1) Utilize a análise orientada a objetos (OOA) para modelar o sistema.
2) Criar um diagrama de classes, considerando que todas as classes partem de uma classe chamada **Object.**
3) Codificar as classes utilizando a linguagem **Java**.

ANÁLISE ORIENTADA A OBJETOS (OOA)

EXEMPLO – SOLUÇÃO – IDENTIFICANDO OS OBJETOS

O departamento de obras públicas da cidade de Uberlândia decidiu desenvolver um **sistema** de computador para rastreamento e conserto de **buracos** de rua (SIRCOB). À medida que são registrados buracos de rua, eles recebem um **número de identificação** e são armazenados de acordo com o **endereço da rua**, **tamanho** (numa escala de 0 a 10), **localização** (no meio da rua; na calçada; etc.), **bairro** (determinado a partir do endereço da rua) e **prioridade** de reparo (determinada a partir do tamanho do buraco). Dados de **ordem de trabalho** são associados a cada buraco, e eles incluem **localização** e **tamanho** do buraco, **número de identificação da equipe de reparos**, **número de pessoas na equipe**, **equipamentos** designados, **horas** aplicadas ao reparo, **status** do trabalho (em andamento, concluído, não concluído), **quantidade** de material de enchimento usado e **custo** do reparo (computado a partir das horas trabalhadas, número pessoas, material e equipamentos usados). Finalmente, um **arquivo** de **danos** ocorridos é criado para guardar informações sobre danos registrados devido ao buraco, o qual inclui o **nome do cidadão**, **endereço**, **número telefônico**, **tipo de dano** e a **quantia** em reais a ser paga. O SIRCOB é um sistema on-line; as consultas devem ser feitas interativamente.

ANÁLISE ORIENTADA A OBJETOS (OOA)

EXEMPLO – SOLUÇÃO – IDENTIFICANDO OS OBJETOS

NOMES	CLASSIFICAÇÃO	ANÁLISE
departamento	Unidade Organizacional	Rejeitado: 1 e 2 Falham
sistema	coisa	Aceito: Todas se aplicam
buracos	coisa	Aceito: Todas se aplicam
número de identificação	coisa	Rejeitado: 3 falha
endereço da rua	coisa	Rejeitado: 3 falha
tamanho	coisa	Rejeitado: 3 falha
localização	coisa	Rejeitado: 3 falha
bairro	bairro	Rejeitado: 3 falha
prioridade	coisa	Rejeitado: 3 falha
ordem de trabalho	coisa	Aceito: Todas se Aplicam
número de identificação da equipe	coisa	Rejeitado: 3 falha
número de pessoas	coisa	Rejeitado: 3 falha
equipamentos	coisa	Rejeitado: 3 falha
horas	coisa	Rejeitado: 3 falha
status	coisa	Rejeitado: 3 falha
quantidade	coisa	Rejeitado: 3 falha
custo	coisa	Rejeitado: 3 falha
arquivo de danos	estrutura	Aceito: Todas se aplicam
nome do cidadão	coisa	Rejeitado: 3 falha
Endereço	coisa	Rejeitado: 3 falha
número telefônico	coisa	Rejeitado: 3 falha
tipo de dano	coisa	Rejeitado: 3 falha
quantia	coisa	Rejeitado: 3 falha

ANÁLISE ORIENTADA A OBJETOS (OOA)

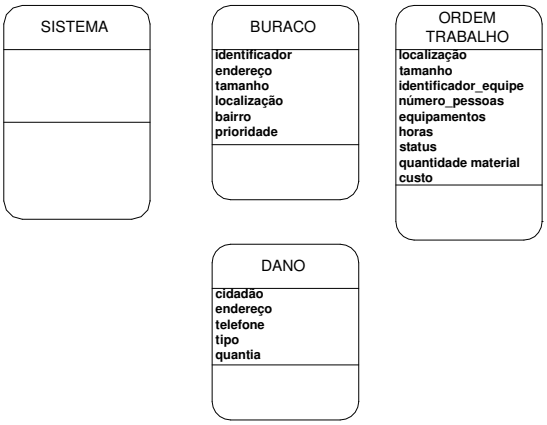
EXEMPLO – SOLUÇÃO – IDENTIFICANDO OS ATRIBUTOS

NOMES	CLASSIFICAÇÃO	ANÁLISE
departamento	Unidade Organizacional	Rejeitado: 1 e 2 Falham
sistema	coisa	Aceito: Todas se aplicam
buracos	coisa	Aceito: Todas se aplicam
número de identificação	coisa	Rejeitado: 3 falha (Atributo)
endereço da rua	coisa	Rejeitado: 3 falha (Atributo)
tamanho	coisa	Rejeitado: 3 falha (Atributo)
localização	coisa	Rejeitado: 3 falha (Atributo)
bairro	coisa	Rejeitado: 3 falha (Atributo)
prioridade	coisa	Rejeitado: 3 falha (Atributo)
ordem de trabalho	coisa	Aceito: Todas se Aplicam
número de identificação da equipe	coisa	Rejeitado: 3 falha (Atributo)
número de pessoas	coisa	Rejeitado: 3 falha (Atributo)
equipamentos	coisa	Rejeitado: 3 falha (Atributo)
horas	coisa	Rejeitado: 3 falha (Atributo)
status	coisa	Rejeitado: 3 falha (Atributo)
quantidade	coisa	Rejeitado: 3 falha (Atributo)
custo	coisa	Rejeitado: 3 falha (Atributo)
arquivo de danos	Estrutura / Coisa	Aceito: Todas se aplicam
nome do cidadão	coisa	Rejeitado: 3 falha (Atributo)
Endereço	coisa	Rejeitado: 3 falha (Atributo)
número telefônico	coisa	Rejeitado: 3 falha (Atributo)
tipo de dano	coisa	Rejeitado: 3 falha (Atributo)
quantia	coisa	Rejeitado: 3 falha (Atributo)

ANÁLISE ORIENTADA A OBJETOS (OOA)

EXEMPLO – SOLUÇÃO – OBJETOS E ATRIBUTOS

- Além dos objetos mostrados a seguir que são identificados diretamente a partir da especificação do sistema outros objetos podem ser necessários dependendo de como o sistema será construído. (Ex.: Equipe: Trabalhador:



ANÁLISE ORIENTADA A OBJETOS (OOA)

EXEMPLO – SOLUÇÃO – IDENTIFICANDO OS MÉTODOS

VERBOS

- **Verbos destacados**

registrados

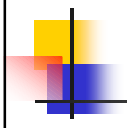
Recebem

Armazenados

Associados

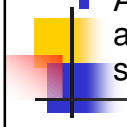
Guardar

- **Outras operações serão necessárias para que o sistema funcione corretamente.**



PROGRAMAÇÃO EM CAMADAS

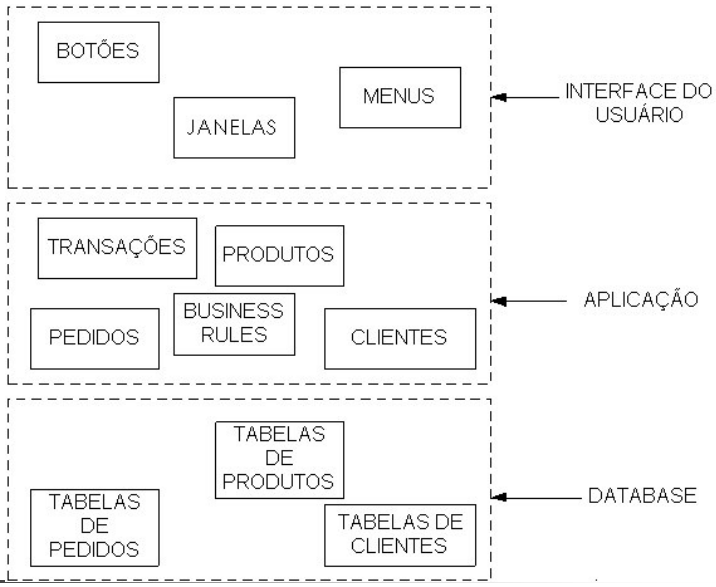
- É uma abordagem, onde a aplicação é dividida em vários níveis. Cada nível, pode conter várias classes, porém em cada um destes níveis existem resposabilidades específicas.
- Um exemplo é a arquitetura de 3 camadas, que possui os seguinte níveis: Interface; Aplicação ou Regras de Negócios e Banco de dados.
- Neste tipo de abordagem as classes de uma camada se comunicam com as camadas adjacentes.
- Uma das vantagens desta abordagem reside no fato de que as classes de cada camada se especializam em determinadas tarefas, facilitando assim a abstração e criação destas classes.
- Além disso qualquer alteração em uma camada não deve afetar a outra, considerando que a comunicação entre elas se mantém fixa.



PROGRAMAÇÃO EM CAMADAS

- Uma outra vantagem deste conceito é a possibilidade de utilizar as várias camadas distribuidas em uma rede, visto que desde a concepção o software foi dividido em partes que se intercomunicam.
- Apesar de ser um modelo com várias vantagens, é necessário um maior esforço a fim conseguir modelar as várias camadas necessárias.
- O número de camadas e o tipo das camadas pode ser variável, dependendo do tipo de aplicação.

PROGRAMAÇÃO EM CAMADAS - Exemplo



PROJETO DE CONCLUSÃO DE CURSO

- Valor: 40 pontos
- Apresentação: Relatório Escrito e Entrevista de avaliação
- O projeto será entregue em etapas durante o curso. Dessa forma partes bem definidas serão entregues em diferentes datas.
- O trabalho será feito em grupo e o mesmo deverá possuir entre 2 e 3 pessoas.
- Cada participante deverá ficar responsável por implementar classes específicas, porém será necessário o entendimento da solução como um todo.
- Inicialmente deverá ser feita uma proposta de trabalho que irá conter uma descrição de um software a ser desenvolvido.
- O projeto deverá utilizar interface gráfica (Applet ou Windows)

PROJETO DE CONCLUSÃO DE CURSO

- CRITÉRIOS DE AVALIAÇÃO
  - COMPLEXIDADE - A complexidade da proposta será um item que será analisado. As propostas que tiverem uma maior complexidade, serão melhor avaliadas.
  - FINALIZAÇÃO DA PROPOSTA – Após um detalhamento da proposta serão especificados quais os requisitos necessários ao software. Ao final será feita uma verificação se todos estes requisitos foram implementados e além disso serão feitos testes para verificar o funcionamento do software como um todo.
  - RESPEITO AO CRONOGRAMA - Neste item, será verificado se o grupo entregou todas as etapas nas datas solicitadas.

PROJETO DE CONCLUSÃO DE CURSO

- CRITÉRIOS DE AVALIAÇÃO (continuação)
  - PROGRAMAÇÃO EM CAMADAS - Será avaliado se a solução proposta pelo grupo utiliza da divisão do problema em camadas, onde cada camada possui responsabilidades específicas.
  - ENTREVISTA INDIVIDUAL - Cada participante deverá apresentar e explicar os conceitos utilizados na parte que ficou sob sua responsabilidade. Os participantes serão arguidos e será verificado o conhecimento adquirido; o entendimento das várias tecnologias; o conhecimento da linguagem e seus recursos. Este item será analizado e pontuado individualmente.
  - RELATÓRIO

PROJETO DE CONCLUSÃO DE CURSO

- RELATÓRIO

  A pontuação do neste item será compartilhada por todos os componentes do grupo. Será verificado se o relatório contém todas as seções necessárias conforme indicado abaixo:
  - INTRODUÇÃO
  - DESENVOLVIMENTO
    - **Especificação de Requisitos**
    - **Diagrama de Classes**
    - **Descrição da solução**
    - **Tecnologias Utilizadas**
  - CONCLUSÃO
  - BIBLIOGRAFIA
- O relatório deverá ser criado a medida que o projeto vai sendo implementado e as seções acima representam o seu conteudo final.

PROJETO DE CONCLUSÃO DE CURSO

- **Primeira Etapa**: Proposta de trabalho e descrição do software a ser desenvolvido. Deverá ser feita uma pesquisa sobre assuntos possíveis; o grupo terá liberdade para fazer suas propostas. As mesmas serão analisadas e poderão ser ou não aprovadas.
- Sugestões de assuntos
  - Banco de dados (SQL - JDBC)
  - Pacote gráfico (Desenhos geométricos em 2D)
  - Criptografia
  - Multimídia: Manipulação de imagens; Sons; Animação; Jogos; etc.
  - Comunicação através da Internet (Sockets)
  - Compactação de dados
  - Programação concorrente (Multithreading)
  - Outras sugestões...
    - http://java.sun.com/j2se/1.4.1/docs/api
    - Livro Deitel; outras publicações;...

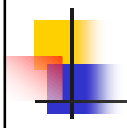
Data de Entrega: 13/05/2003

ABSTRACÃO DE DADOS

- Abstração é o processo de identificar as qualidades ou propriedades importantes do problema que está sendo modelado. Através de um modelo abstrato, pode-se concentrar nas características relevantes e ignorar as irrelevantes.
- Abstração é fruto do raciocínio.
- Uma linguagem de programação é uma abstração do funcionamento do computador
- Importante conceito dentro da POO
- Exemplos: Objeto representando uma pessoa; tipos de dados básicos de uma linguagem – int; double; char; etc.

ESTRUTURA ABSTRATA DE DADOS

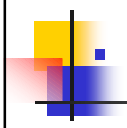
- Estrutura de dados abstrata, contém dados e métodos
- Dados estão ocultos dentro da estrutura e podem ser acessados somente por meio dos métodos (Encapsulamento)
- Nome da estrutura e sua interface são conhecidos mas não a sua implementação.
- A sua implementação encontra-se oculta.
- Métodos que permanecem inalterados, mesmo que a implementação dos mesmos venha a ser alterada.



TIPOS ABSTRATOS DE DADOS (TAD)

ABSTRACT DATA TYPES (ADT)

- TAD contém a definição da estrutura de dados abstrata
- TAD é um conjunto de estruturas de dados abstratas
- Uma particular estrutura de dados abstrata de um TAD é denominada de uma instância do TAD
- TAD deve possuir
  - Nome
  - Dados
  - Métodos
    - Construtores - constrói e inicializa uma instância do TAD
    - Modificadores – Modifica estado (dados) do TAD
    - Acessores – Retorna estado do TAD, sem modificar seus dados
    - Destrutores – Destrói uma instância do TAD
- A criação de uma classe é a definição de um TAD



TIPOS ABSTRATOS DE DADOS (TAD)
ABSTRACT DATA TYPES (ADT)

- TDA deve conter as assinaturas da operações
- Assinatura contém as seguintes informações:
  - Nome
  - Conjunto de entrada (tipos, número e ordem dos parâmetros)
  - Conjunto de saída (o tipo do valor de retorno)
- Cada método deve conter
  - PRÉ-CONDIÇÕES – Condições que deverão ser aplicadas aos dados antes da utilização do método
  - PÓS-CONDIÇÕES – Condições que indicam a modificação que foi efetuada sobre os dados
- Os tipos utilizados no TDA deve ser tipos válidos e conhecidos.

TIPOS ABSTRATOS DE DADOS - Exemplo

- TDA que repesenta um círculo
  - Nome
    Circulo
  - Dados
    raio : número inteiro não negativo
    xCentro : número inteiro que indica a coordenada x do ponto (x,y)
    yCentro : número inteiro que indica a coordenada y do ponto (x,y)
  - Métodos
    Construtor
      Valores Iniciais: O raio do círculo
      Processo: Cria um círculo de raio informado e origem no ponto (0,0)
    Area
      Entrada: Nenhuma
      Pré-condições: Nenhuma
      Processo: Calcula a área do círculo
      Saída: A área do círculo
      Pós-condições: Nenhuma
    Escala
      Entrada: Um número inteiro indicando o fator de escala
      Pré-condições: O fator de escala deve ser maior que zero
      Processo: Multiplica o raio pelo fator de escala e redesenha o círculo
      Saída: Nehuma
      Pós-condições: O valor do raio será o resultado da múltiplicação raio pelo fator de escala

A TECNOLOGIA JAVA

- Tecnologia recente. Formalmente lançada em 1995 pela SUN, sendo neste ano, incorporada ao Netscape Navigator
- Sua concepção está voltada para a internet.
- Sintaxe semelhante à linguagem C++ amplamente conhecida e utilizada
- Não possui recursos existentes em C++ como: Ponteiros; Arquivos de Cabeçalho; `Struct`; `Union`; Sobrecarga de Operadores; etc.
- Possui recursos importantes: Multithreading; Múltimidia; Componentes de Interface gráfica; processamento distribuído

A TECNOLOGIA JAVA

- Puramente Orientada a objetos: Modularidade; Reusabilidade de Código; Tendência Atual;
- Portabilidade: O código de um programa java, chamado **bytecodes**, pode ser executado em qualquer plataforma
- Interpretada: Os **bytecodes** são executados através da máquina virtual java (Java Virtual Machine). Apesar de interpretada a linguagem tem peformance suficiente (Cerca de 10 vezes mais lenta que C)
- Dinâmica: Qualquer classe Java pode ser carregada e instanciada em um interpretador Java a qualquer momento, mesmo quando ele já está rodando.

A TECNOLOGIA JAVA

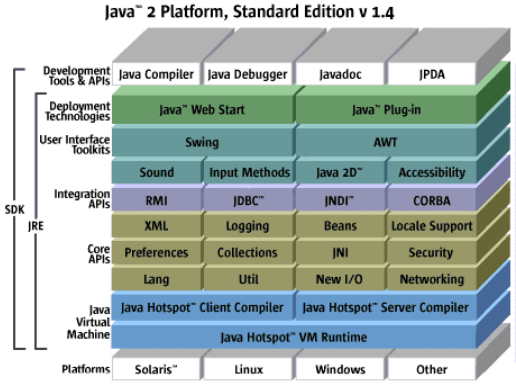
- Distribuída: Proporciona considerável suporte à redes de alto e de baixo nível . Permite a invocação remota de métodos (RMI) e o carregamento de componentes através da rede, por exemplo.
- Tipagem forte (strong typing): Através de uma abrangente checagem em tempo de compilação, problemas em potencial no casamento de tipos, são evitados, tornando o código mais confiável.
- Robusta: Tratamento de exceções e gerenciamento automático de memória permitem a criação de programas robutos

A TECNOLOGIA JAVA

- APPLETS: Programas que podem ser inseridos em uma página Web. O applet é executado automaticamente quando a página HTML que o contém é aberta em um browser habilitado para Java. Este recurso é unicamente oferecido por java.
- Sistemas Embutidos: A plataforma Java está disponível e pode ser executada em equipamentos como: Telefones Celulares; Pagers; Eletrodomésticos; Automóveis; etc. Isto a torna uma importante linguagem para o desenvolvimento de sistemas embutidos (embedded systems)

JAVA SDK

- Possui os recursos necessários para o desenvolvimento de aplicações em java
- O J2SE SDK (Java 2 Standard Edition Software Development Kit) consiste na base para o desenvolvimento utilizando a tecnologia Java



JAVA SDK

- Versão Atual: Java(TM) 2 SDK, Standard Edition 1.4.1_02
- A versão mais atualizaada pode ser obtida no site: http://java.sun.com/j2se/
- Software Development Kit (SDK): Constituído de duas partes básicas:
  - Java Run-time Edition(JRE): Conjunto mínimo para a execução de aplicações escritas em java. Contém Máquina Virtual Java (JVM); Classes básicas e arquivos de suporte.
  - Ferramentas básicas para a criação de aplicações java.

JAVA SDK – FERRAMENTAS BÁSICAS

- As ferramentas básicas abaixo, são instaladas juntamente com o SDK
- Estas ferramentas podem ser acessadas através da linha de comando do console do S.O.

FERRAMENTA	DESCRIÇÃO
javac	Compilador para a linguagem Java
java	Utilizado para a executar uma aplicação Java
javadoc	Utilitário utilizado para documentação das classes
appletviewer	Permite a execução e debug de Applets sem a utilização de um browser
jar	Cria um arquivo .jar a partir de vários arquivos .class
jdb	Debugger da linguagem Java

HELLO JAVA WORLD!

- A código abaixo mostra uma aplicação básica que pode ser executada na console do S.O

```
/**
* Title:         HELLO JAVA WORLD
* Description:   Aplicação básica(console S.O.)
* Company:      AUTOENGE
* @author       FLAVIO SILVA
*/
public class HelloJavaWorld {
  public static void main(String[] args) {
    System.out.println("Hello Java World!");
  }
}
```

HELLO JAVA WORLD!

- **javac** – Compila o código. Este processo cria o arquivo **.class** que poderá ser executado

c:\ > javac HelloJavaWorld.java

- **java** – Executa o código compilando

c:\ > java HelloJavaWorld

- **jar** – Cria um arquivo compactado **(.jar)** que pode conter classes(*.class), imagens e sons.
- O arquivo mffile.mft, chamado "manifest file" deve conter, pelo menos, o seguinte texto:

  <BeginOfFile>

  Main-Class: HelloJavaWorld<NewLine>

  <EndOfFile>

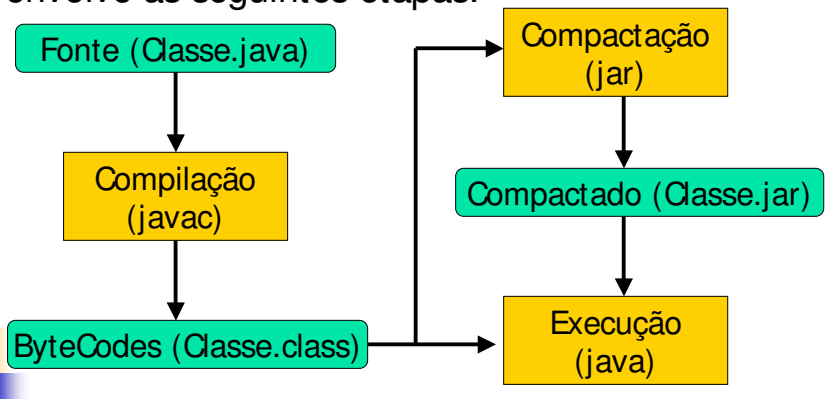
c:\ > jar -cmf mffile.mft HelloJavaWorld.jar *.class

HELLO JAVA WORLD!

- Executando um arquivo .jar

c:\ > java –jar HelloJavaWorld.jar

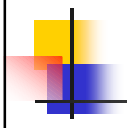
- O processo de criação de um código em java envolve as seguintes etapas:



LINGUAGEM JAVA - COMENTÁRIOS

- O código java pode conter comentários da seguinte forma:

```
/**
 * Todo o texto é ignorado
 *
*/

//Apenas o trecho após as barras é
//ignorado!
```



LINGUAGEM JAVA - IDENTIFICADORES

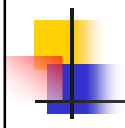
- Um identificador é um nome que identifica unicamente uma variável, método ou classe. As seguintes restrições se aplicam na definição de identificadores:
  - Todos os identificadores devem começar por uma letra, um *underscore* (_) ou o sinal de cifrão ($).
  - Um identificador pode incluir, mas não começar por, um número
  - Não podem existir espaços em branco no nome de um identificador
  - Java, como a linguagem C, faz diferença com relação às letras maiúsculas e minúsculas no nome de um identificador
  - **Palavras reservadas** da linguagem Java não podem ser usadas como identificadores

LINGUAGEM JAVA - LITERAIS

- Um valor constante em Java é criado usando-se uma representação literal desse valor.

  Exemplos:

```
500  // Um valor inteiro.
3.14 // Valor de ponto flutuante.
'Y'// Constante de caractere.
"String de teste" //Constante de string.
```



PALAVRAS RESERVADAS

- A tabela abaixo mostra as palavras reservadas utilizadas pela linguagem java

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

- As palavras: **true false null**; também são reservadas e representam valores definidos por Java

LINGUAGEM JAVA – TIPOS BÁSICOS

- Tipos básicos utilizados por java

TIPO	DESCRIÇÃO	TAMANHO (bits)	FAIXA DE VALORES	
			VALOR MÍNIMO	VALOR MÁXIMO
byte	inteiro	8	-128	127
short	inteiro	16	-32768	32767
int	inteiro	32	-2147483648	2147483647
long	inteiro	64	-9223372036854775808	9223372036854775807
float	ponto flutuante, precisão simples	32	3,4e-038	3,4e+038
double	ponto flutuante, precisão dupla	64	1.7e-308	1.7e-308
char	caracter unicode	16	0	65535
boolean	valor lógico (falso/verdadeiro)		false	true

- Qualquer identificador em Java deve possuir um tipo (strong typing)
- Os tipos básicos representam valores individuais e não objetos. O motivo para isto é a eficiência.

LINGUAGEM JAVA – TIPOS BÁSICOS

- Devido à sua portabilidade todos os tipos de dados têm uma faixa de valores estritamente definida. Por exemplo, um int é sempre de 32 bits, independente da plataforma de execução.

DECLARANDO VARIÁVEIS

- A variável é a unidade básica de armazenamento de um programa Java.
- Uma variável é definida pela combinação de um identificador, um tipo e um inicializador opcional.

  `tipo nomeVariavel [= valorVariavel];`
- A inicialização acima somente é possível para tipos básicos
- Todas as variáveis têm um escopo, que define sua visibilidade e sua duração

```
int iA, iB, iC;
int iD = 3, iE, iF = 5;// inicializando
byte btZ = 22;// Declara e inicializa
double dPi = 3.14159; //ponto flutuante
char chX = 'a'; //caracter
```


CRIANDO OBJETOS

- Para criar uma instância de um objeto utiliza-se a função alocadora de objetos **new**.

  Exemplos:

```
Pessoa p;          //Declara uma varíavel
p = new Pessoa(); //Aloca o objeto
Pessoa p2 = new Pessoa(); //Declara e Aloca
```

- Na declaração de **p**, após a execução da primeira linha a variável contém o valor **`null`**. A declaração cria uma **referência** para um objeto
- Somente após a alocação é que o objeto poderá ser utilizado. Caso contrário ocorrerá um erro de compilação.

OPERADORES

- ARITMÉTICOS – "+" "-" "*" "/"

```
int iA = 13;
int iV = 7;
float f1 = 13;
iA+iV //retorna 20
iA-iV //retorna 6
iA*iV //retorna 91
iA/iV //retorna 1
f1/iV //retorna 1.8571428
```


OPERADORES

- LÓGICOS – "&&" (AND) "||" (OR) "!" (NOT)

```
boolean bX, boolean bY;
```

bX	*bY*	*bX && bY*	*bX || bY*	*!bX*
true	true	true	true	false
true	false	false	true	
false	true	false	true	true
false	false	false	false	

OPERADORES

- RELACIONAIS

IGUALDADE	==	MENOR OU IGUAL QUE	<=
DESIGUALDADE	!=	MENOR OU IGUAL QUE	<=
MENOR QUE	<	MAIOR OU IGUAL QUE	>=

Exemplos:

```
int iH = 4, iJ = 4, iK = 6;
iH == iJ          //Retorna true
iH == iK          //Retorna false
iK == iK          //Retorna true
iH < iK           //Retorna true
iH < iJ           //Retorna false
iH <= iJ          //Retorna true
```


OPERADORES

- RELACIONAIS (cont.)

Exemplos:

```
Person homem = new Person();
Person mulher = new Person();
homem == mulher //Retorna false. Mesmo
//valor porém diferentes objetos
homem == homem  //Retorna true
```

OPERADORES

- OPERADORES BINÁRIOS

"AND" BINÁRIO	&	"OR" INCLSUIVO BINÁRIO	|
"OR" EXCLUSIVO BINÁRIO	^	COMPLEMENTO BINÁRIO	~

Exemplos:

```
int c = 4;          //binário       00000100
int d = 6;          //binário       00000110
int bytRes;
bytRes = c & d;     //retorna 4     00000100
bytRes = c ^ d;     //retorna 2     00000010
bytRes = c | d;     //retorna 6     00000110
bytRes = ~c;        //retorna -5    11111011
bytRes = d >> 2;    //retorna 1     00000001
bytRes = d << 2;    //retorna 24    00011000
```


OPERADORES

- OPERADOR CONDICIONAL(? :)

Exemplo:

```
a ? b : c          //retorna o valor b se a é true,
                   //caso contrário retorna o valor c
```

OPERADORES

- Associatividade da esquerda para direita
- Regras de Precedência

[] . e++ e--
-e !e ~e ++e --e
new
* / %
+ -
== !=
&
^
|
&&
||
? :
= += *= /= %= >>= <<= &= ^= !=

Exemplos:

```
a/b*c    interpretado como (a/b)*c
a+b*c/d  interpretado como a+((b*c)/d)
```


OUTROS IMPORTANTES OPERADORES

- ATRIBUIÇÃO DE VALORES (Operador =)

  Variável (v e v1) à esquerda irá conter o valor E, especificado à direita.

  v = E;

  v = v1 = E; equivale a v = (v1 = E); //Evitar!!!

- ATRIBUIÇÃO COM OPERADOR (op=)

  Sendo op um operador

  v op= f; equivale a v = v op f;

  Ex: a += 2; equivale a a = a + 2;

- ASSESSANDO UM MÉTODO DE UM OBJETO

  Utiliza-se o operador ponto "." Ex.:

  Pessoa p = new Pessoa();

  p.getName();

LINGUAGEM JAVA – TIPO STRING

- Classe que contém a representação de uma de uma sequência de caracteres do tipo "ABC".
- Na linguagem JAVA o tipo String é um objeto
- Existem vários métodos e operadores já definidos para esta classe.

Exemplo:

```
String s1 = new String("FLAVIO");
String s2 = new String("SILVA");
s1 = s1 + s2;  //concatenação de strings
s1.length();   //retorna o tamanho da string;
s1.trim();     //remove espaços em branco
               //inicio e fim
```


BLOCOS DE CÓDIGO

- Um bloco é iniciado pelo caracter **{** e finalizado pelo caracter **}**
- As instruções (métodos; declarações; atribuições) contidas em um bloco são tratadas como **uma única** instrução.

Exemplo:

```
{
    a = a + 2;
    b = circle.area();
    c = a + b;
}
```

BLOCOS DE CÓDIGO

- DECLARAÇÕES LOCAIS – Dentro de um bloco é possível fazer declarações de objetos e variáveis que serão válidas somente dentro deste bloco

```
int a;
Circle circle = new Circle(3);
a = 2;
{
  //Declaração local de a - válida somente
  //dentro do bloco
  double a;
  a = 3.14 + circle.area();
}
//Imprime o valor de a - neste caso será 2
System.out.println("O valor de a é: " + a)
```


ASSINATURA BÁSICA DE UM MÉTODO

- A assinatura de um método é criada da seguite forma:

[*nívelAcesso*] *tRetorno* **nomeMetodo**(T0 P0, ...,Tn Pn)

nívelAcesso - `private`; `public` ou `Protected`

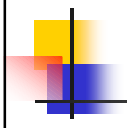
tRetorno - Tipo de retorno. Se não houver nenhum retorno, deve ser utilizado **`void`**. O tipo pode ser básico ou então qualquer objeto da linguagem

T0,...Tn – Tipos dos parâmetros utilizados

P0,...,Pn – Nome dos parâmetros

NÍVEIS DE ACESSO

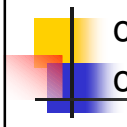
- Os níveis de acesso a atributos, métodos e classes são definidos da seguinte forma:
  - **public:** Método ou atributo visível a todas as classes (público)
  - **protected:** Método ou atributo visível nas subclasses (protegido)
  - **private:** Método ou atributo visível somente na classe onde é utilizado (privado)



SOBRECARREGAMENTO (Overloading)

- Consiste em possuir mais de um método com o mesmo nome, porém com diferentes parâmetros (número, ordem e tipos)
- Este recurso é utilizado para métodos que realizam tarefas semelhantes porém sobre tipos de dados diferentes
- Normalmente este recurso também é utilizado no construtor da classe
- O sobrecarregamento facilita a reutilização de código

Exemplo: Circle c = new Circle();
c.move(x,y);//Recebe a coordenada x e y
c.move(p); //Recebe o objeto ponto



CRIANDO UMA CLASSE EM JAVA

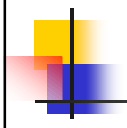
- Definição classe - Necessário keyword **`class`**
- Corpo da Classe – Delimitado por **{** e po r **}**
- A classe contém dados (atributos ou variáveis membro) e métodos(procedimentos)
- Dados (Variáveis Membro)
  - Dados que representam o objeto; Podem ser acessados diretamente dentro da definição da classe, semelhante a uma "variável global"
- MÉTODOS
  - CONSTRUTORES
  - DESTRUTORES
  - MODIFICADORES
  - ACESSORES
  - OUTROS MÉTODOS

CRIANDO UMA CLASSE EM JAVA

- MÉTODOS CONSTRUTORES
  - Método especial. Deve possuir o mesmo nome da classe
  - Inicializa os dados (variáveis membro) do objeto
  - Garante que objetos iniciem em um estado consistente
  - Construtores normalmente são sobrecarregados
  - Construtores não podem retornar um valor
  - O construtor que não recebe nenhum parâmetro é conhecido como "construtor default"

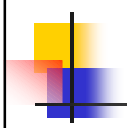
CRIANDO UMA CLASSE EM JAVA

- MÉTODOS DESTRUTORES
  - Devolve os recursos para o sistema
  - Não tem parâmetros, não retorna valor. Sua assinatura é: **protected void finalize()**
  - O método finalize somente é chamado imediatamente antes da coleta de lixo (garbage colletion). Ele não é chamado quando um objeto sai de escopo, por exemplo.
  - Normalmente não é necessário a criação deste método pois o mesmo é herdado da classe **Object**



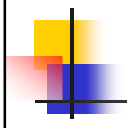
CRIANDO UMA CLASSE EM JAVA

- MÉTODOS MODIFICADORES (setXXX)
  - Permitem a modificação dos dados (variáveis membro) do objeto
  - Devem ser do tipo `public`, pois são acessados de forma externa ao objeto
  - Normalmente começam com o “set” a fim de facilitar o entendimento de seu objetivo.



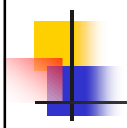
CRIANDO UMA CLASSE EM JAVA

- MÉTODOS ACESSORES (getXXX)
  - Permitem a recuperação dos dados (variáveis membro) do objeto
  - Devem ser do tipo **`public`**, pois são acessados de forma externa ao objeto
  - Normalmente começam com o "get" a fim de facilitar o entendimento de seu objetivo.



CRIANDO UMA CLASSE EM JAVA

- OUTROS MÉTODOS
  - Executam tarefas que são de responsabilidade do objeto e que irão representar o comportamento do mesmo.
  - Estes métodos podem ser públicos (**`public`**), protegidos (**`protected`**) ou privados (**`private`**), conforme sua utilização
  - Para que o método seja acessado externamente o mesmo de ser do tipo público (**`public`**)
  - Caso o método seja apenas auxiliar à classe sem uma ligação direta com o comportamento do objeto o mesmo deve ser do tipo privado (**`private`**). Normalmente este métodos são utilizados por um outro método público



ESTRUTURA BÁSICA DE UMA CLASSE

```
[nívelAcesso] class NomeClasse {
    [nívelAcesso] tRetorno variavel_membro1;
    [nívelAcesso] tRetorno variavel_membroN;
    //Construtores
    [nívelAcesso] NomeClasse(T0 P0, ...., Tn Pn);
    //Destrutor
    protected void finalize();
    //Modificadores
    [nívelAcesso] tRetorno setXXX(T0 P0, ...., Tn Pn);
    //Acessores
    [nívelAcesso] tRetorno getXXX (T0 P0, ...., Tn Pn);
    //Outros métodos (privados; protegidos e públicos)
    [nívelAcesso] tRetorno Metodo(T0 P0, ...., Tn Pn)
}
```

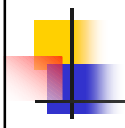

CRIANDO UMA CLASSE - EXEMPLO

```
public class Person{
  private String name;
  //Construtor
  public Person(){ name = ""; }
  //Get Metodos
  public String getName() { return name;}
  //set Metodos
  public void setName(String s){ name = s; }
  //Destrutor
  protected void finalize(){
      name = "";
  }
}
```

CRIANDO UMA CLASSE – Exemplo 2

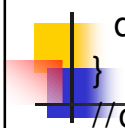
- Neste caso será utilizado o Sobrecarregamento (Overloading) no construtor da classe e também no método setRaio.

```
public class Circle {
  //Variáveis membro ou atributos
  protected double dX;
  protected double dY;
  protected double dRaio;
  //...continua a definição
```



CRIANDO UMA CLASSE – Exemplo 2

```
  //...continuando a classe Circle
  //Métodos Construtores
  public Circle(double dCx, double dCy, double dR){
    System.out.println("Novo Círculo – Construtor 2");
    dX = dCx;
    dY = dCy;
    dRaio = dR;
  }
  protected Circle(double dR){
    System.out.println("Novo Círculo – Construtor 3");
    dX = dY = 0;
    dRaio = dR;
  }
  //continua...
```



CRIANDO UMA CLASSE – Exemplo 2

```
//...continuando a classe Circle
//Métodos Modificadores
public void setCenter(double dCx, double dCy){
  dX = dCx;
  dY = dCy;
}
public void setRaio(double dR){
  dRaio = dR;
}
public void setRaio(int iR){
  dRaio = iR;
}
//continua...
```


CRIANDO UMA CLASSE – Exemplo 2

```
//...continuando a classe Circle
//Métodos Acessores
public double getRaio(){
   return dRaio;
}
public double getXCenter(){
   return dX;
}
public double getYCenter(){
   return dX;
}
//continua...
```

CRIANDO UMA CLASSE – Exemplo 2

```
//...continuando a classe Circle
//Outros métodos
public double perimeter(){
   double perimetro = 2 * dRaio * Math.PI;
      System.out.println("Perimetro: " + perimetro);
      return perimetro;
 }
 public double area(){
   double dArea = Math.PI* Math.pow(dRaio,2);
   System.out.println("Area: " + dArea);
   return dArea;
 }
}
```


CRIANDO UMA CLASSE – Exemplo 2

```
public class Circle {
  //Variáveis membro ou atributos
  //Métodos Construtores
  public Circle(double dCx, double dCy, double dR){...}
  public Circle(double dR){...}
  //Métodos Modificadores
  public void setCenter(double dCx, double dCy){...}
  public void setRaio(double dR){...}
  public void setRaio(int iR){...}
  //Métodos Acessores
  public double getRaio(){...}
  public double getXCenter(){...}
  public double getYCenter(){...}
  //Outros métodos
  public double perimeter(){...}
  public double area(){...}
}
```

CRIANDO UMA CLASSE – Exemplo 2

```
//Exemplo da Utilização de Sobrecarga
...
Circle c1 = new Circle(3);
Circle c2 = new Circle(3,5.4,8);
double dP, dA;
double dRaio = 7.5;
int iRaio = 5;
//Utilizando sobrecarga no método setRaio
c1.setRaio(dRaio);
c2.setRaio(iRaio);
dP = c1.perimeter();
dA = c2.area();
```


CRIANDO UMA CLASSE - Exemplo 3

```
class Person{
  protected String name;
  //Evite criar variáveis membro como public!!!!
  public String global; //Perde o encapsulamento de dados
  //Construtor
  public Person(){ name = ""; }
  //Get Metodos
  String getName(){ return name; }
  String getGlobal(){ return global; }
  //set Metodos
  void setName(String s){name = s};
  }
}
```

CRIANDO UMA CLASSE - Exemplo 3

```
//Exemplo de Utilização do objeto
...
Person p = new Person();
int i;
String s, s2;
p.setName("Flavio");
//EVITAR!!!! – A linha abaixo mostra a perda do
//encapsulamento de dados
p.global = “Silva";
s = p.getName();
s2 = p.getGlobal();
```


ESTRUTURAS DE CONTROLE

- ESTRUTURA DE SELEÇÃO – **if** / **if-else**
  - Uma instrução (ou bloco de instruções) somente será executada caso uma condição (expressão condicional E) resultar em verdadeiro (true)

```
if (E)
   S;
```

ESTRUTURAS DE CONTROLE

- ESTRUTURA DE SELEÇÃO – **if** / **if-else**
  - Uma variação da estrutura acima é a estrutura **if-else**. Onde uma escolha é feita. **Se** a expressão condicional (E) for verdadeira (true) então um o bloco de instruções logo após a expressão será executado. **Caso contrário**, sendo a expressão falsa (false) então o bloco que se encontra após a palavra **else** será executado.

```
if (E)
   S;
else
   R;
```


ESTRUTURAS DE CONTROLE

- ESTRUTURA DE SELEÇÃO – **if** / **if-else**
  - **EXEMPLO**

```
if (Saldo >= 0)
  System.out.println("Ok!");
else
  System.out.println("Depositar!");
```

ESTRUTURAS DE CONTROLE

- ESTRUTURA DE SELEÇÃO MÚLTIPLA – **switch**
  - A estrutura **if/ else** permite a seleção no máximo entre 2 blocos diferentes.
  - Caso seja necessário um número maior de opções então deve ser utilizado a estrutura **switch.** Exemplo:

```
switch (E) {
   case c1 : S1;
             break;
   case c2 : S2;
             break;
   ...
   default : Sd;
```


ESTRUTURAS DE CONTROLE

- Inicialmente a expressão **E** é avaliada. Caso a expressão **E**, resulte na constante **c1**, o bloco de instruções **S1** será executado, até que a palavra **break** seja encontrada. Neste caso o próxima instrução após o `switch` será executada. Da mesma forma caso **E**, resulte na constante **c2** o bloco **S2** será executado. E assim, sucessivamente.
- Caso a expressão **E** não resulte em nenhum valor constante presente no swith (**c1, c2, ...**) então o bloco padrão (**default**) Sd será executado
- O bloco default (**Sd**) é opcional, não sendo necessário sua presença.
- Os valores constantes utilizados no **switch** podem ser do tipo **btye; short; int, long e char**

ESTRUTURAS DE CONTROLE

- **Exemplo:**

```
int a = 2;
switch (a){
  case 1 : {
   System.out.println("A é igual a 1");
   break;
  }
  case 2 : {
   System.out.println("A é igual a 2");
   break;
  }
  case 3 : {
   System.out.println("A é igual a 3");
    break;
  }
  default : {
    System.out.println("A é diferente de 1,2e3");
  }
}
```


ESTRUTURAS DE CONTROLE

- ESTRUTURA DE REPETIÇÃO – **while**
  - Esta estrutura indica que a instrução ou bloco de instruções (**S**) que se encontra logo após uma expressão condicional (**E**) será executado **ENQUANTO** tal expressão por verdadeira (**true**).

```
while (E)
   S;
```

  - Quando a última instrução do bloco é executada a expressão (**E**) será novamente executada e caso continue verdadeira o bloco será executado mais uma vez, e assim, sucessivamente.

ESTRUTURAS DE CONTROLE

- ESTRUTURA DE REPETIÇÃO – **while**
  - A fim de evitar que a execução do bloco de instruções (**S**) prossiga indefinidamente é necessário que dentro deste bloco o valor da expressão condicional seja em algum momento alterado para falso (**`false`**).
  - No caso da instrução **`while`**, sempre é feito um teste antes da execução do bloco. Dessa forma para que o bloco seja executado, pelo menos uma vez, o resultado da expressão inicialmente seja verdadeiro.
  - Ao terminar a repetição a primeira instrução após a instrução **`while`** é executada.

ESTRUTURAS DE CONTROLE

- EXEMPLO ESTRUTURA DE REPETIÇÃO – **while**

```
int a = 0;
//Se a instrução abaixo fosse executada
//não permitiria a repetição
//a = 10
while (a < 10){
   System.out.println("Valor de A:" + a);
   System.out.println("A ainda é menor que
  10!");
  //Sem a instrução abaixo a repetição nunca
  terminaria
   a++;
}
//Primeira instrução a ser executada após o while
System.out.println("Agora, a é igual a 10!");
```

ESTRUTURAS DE CONTROLE

- ESTRUTURA DE REPETIÇÃO – **do / while**
  - Esta estrutura é semelhante ao **`while`**, porém sempre o bloco de instruções (**S**) será executado pelo menos uma vez. Em seguida a expressão condicional (**E**) é avaliada e o bloco de instruções será executada **ENQUANTO** seu valor for verdadeiro.

```
do {
S;
} while (E);
```

  - Caso a expressão seja verdadeira a primeira instrução do bloco será executada novamente

ESTRUTURAS DE CONTROLE

- ESTRUTURA DE REPETIÇÃO – **for**
  - Neste caso a repetição e controlada por um contador da seguinte forma:

```
For (C; E; I)
  S;
```

  - O bloco de instruções **S**, será executado enquanto a expressão condicional **E**, for verdadeira. O contador é inicializado pela expressão **C** e a expressão **I** é responsável por alterar o valor do contador.

ESTRUTURAS DE CONTROLE

- ESTRUTURA DE REPETIÇÃO – **for**
  - Exemplo:

```
int i;
for (i = 0; i < 10, i++)
{
  System.out.println("O contador I
  ainda é menor que 10!");
}
```

  - No exemplo acima, o contador é um número inteiro **i**. A instrução será executada enquanto a expressão for verdadeira (i < 10) e **i** será incrementado de 1 em 1.

ESTRUTURAS DE CONTROLE

- ESTRUTURA DE REPETIÇÃO – **for**
  - No laço pode ser utilizada instrução **break** que irá finalizar o mesmo desviando o controle para fora do laço **for**
  - A instrução **continue** desvia o controle para o ínicio do laço **for**, não executando o restante das instruções do laço. A diferença entre **continue** e **break**, é que a instrução **continue** não finaliza a repetição.
  - **break** e **continue** podem ser utilizados também com **while; do/ while**

ARRAYS (VETORES)

- Importante estrutura de dados. Consiste em um grupo de "OBJETOS" do mesmo tipo, onde a cada objeto é associado um "ÍNDICE" único. Estes objetos estão armazenados em uma posição contígua da memória. O número de objetos contidos é definido como sendo o TAMANHO do vetor. A seguir é mostrada a sintaxe para a criação de um array.

  Declaração: `String clientes[];`

  Alocação: `clientes = new String[7];`

  Outra forma:

  ```
  String clientes[] = new String[7];
  ```

- Na declaração não deve ser colocado o tamanho do vetor.
- O código acima cria um vetor de strings com 7 (sete) elementos.

ARRAYS (VETORES)

- O acesso da cada elemento pode ser feito da seguinte forma:

```
clientes[0] = "MARIA"
   //Primeiro elemento
...
clientes[6] = "JOSÉ MARIA"   //Último
elemento
```

- Os vetores são estruturas "estáticas", sendo que uma vez criados não podem ter o seu tamanho alterado. Porém sua criação é feita de forma dinâmica, durante a execução do código.
- Java possui a classe Vector que é um vetor de objetos onde seu tamanho pode ser ALTERADO durante a execução do código.

ARRAYS (VETORES)

- Outros exemplos na declaração e alocação de um vetor:

```
//Declara dois vetores de doubles
double[] aSalarios, aRecebimentos;
//Declara dois vetores de booleanos
boolean[] bOpcoes;
boolean bRespostas[];
//Declara e Aloca dois vetores de
//inteiros
int aNotas[] = new int[47], a[] =
new int[20];
```


ARRAYS (VETORES)

- Quando um vetor é alocado os tipos primitivos **(`int; double; byte;..`)** recebem o valor zero. Tipos **`boolean`** recebem o valor **`false`** e objetos (tipos não primitivos) recebem o valor **`null`**
- O número de elementos (comprimento/tamanho) de um vetor pode ser obtido da seguinte forma:

```
int iNumeroElementos;
iNumeroElementos = aNotas.length;
System.out.println("O tamanho elementos do
vetor é " + iNumeroElementos);
```

ARRAYS (VETORES)

- Exemplos de manipulação de elementos: É possível manipular em expressões tanto o índice do vetor, quanto cada elemento.

```
//Soma de elementos
iSoma = aNotas[0] + aNotas[3] +
aNotas[3+1]
```


ARRAYS (VETORES)

- Exemplos de manipulação de elementos (continuação)

```
int iPrimeiro, iSegundo, iTerceiro,
iResultado ;
iPrimeiro = 0;
iSegundo = 1;
iTerceiro = 2;
//adiciona 7 a quarto elemento do
//vetor (0+1+2=3)
aNotas[iPrimeiro + iSegundo +
iTerceiro ] += 7;
iResultado = aNotas[5]/2 +
aNotas[3/3]*3
```

ARRAYS (VETORES)

- Percorrendo todos os elementos de um vetor

```
for (int ii = 0; ii < aNotas.length;
  ii++)
{
  if (aNotas[ii] < 15)
    System.out.println("Necessário
  estudar mais\n");
  //Imprime o valor de cada elemento
  //do vetor
  System.out.println("Nota: " +
  aNotas[ii] + "\n");
}
```


ARRAYS (VETORES)

- Utilizando vetor como argumento de um método

Exemplo:

```
public calculaIpvaClientes(Veiculo
aVeiculos[])
```

ARRAYS (VETORES)

- VETORES MULTIDIMENSIONAIS – Neste caso cada elemento do vetor possui mais de um índice associado ao mesmo.
- Uma matriz ou tabela é um vetor de duas dimensões. Neste caso pode ser dito que a informação está organizada em linhas e colunas. Exemplo:

```
int aMatriz[][] = new Int[2][3];
```

aMatriz[0][0]	aMatriz[0][1]	aMatriz[0][2]
aMatriz[1][0]	aMatriz[1][1]	aMatriz[1][2]

ARRAYS (VETORES)

- Exemplo: Um vetor com três dimensões

```
int aCuboInteiros[][][] = new
Int[3][3][3];
```

- Percorrendo todos os elementos de um vetor bidmensional(matriz)

```
for (int i = 0; i < aProvas.length; i++)
{
 //Imprime o valor de cada elemento da
 //matriz
 for (int j = 0; j < aProvas[i].length;
  j++)
   System.out.println("Nota: " +
  aProvas[i][j] + "\n");
}
```

LINGUAGEM JAVA – EXEMPLO

- Vamos considerar uma classe Veiculo que possui os seguintes atributos: sPlaca; dValorMercado; Combustivel; dAliquotaIpva; bIpvaPago
- Implentar a classe. Criar um construtor recebe todas as variáveis membro desta classe. Implementar as funções GETxxx/SETxxx conforme as regras abaixo:
- Regras que devem ser respeitas:
  - A aliquota do ipva (dAliquotaIpva) é um valor `double` entre 1 e 10 A aliquota varia conforme o combustível: "GASOLINA" – 4; "ALCOOL" – 3; "DIESEL" – 2
  - A placa e o combustível deve ser diferente de "" (string vazia)

LINGUAGEM JAVA – EXEMPLO

- O combustivel pode assumir os seguintes valores: "ALCOOL", "GASOLINA" e "DIESEL"
- O valor de mercado é um valor `double`, sempre maior que zero
- O valor bIpvaPago é um `boolean` e indica se o Ipva foi ou não pago durante o ano atual.
- Além dos métodos acima a classe possui o método calculaIpva. (dAliquotaIpva/100 * dValorMercado)

LINGUAGEM JAVA – EXEMPLO

- Vamos implementar a classe ProcessaVeiculo. Esta classe é constuida de um vetor de veiculos.
- O construtor desta classe recebe o número máximo de veiculos (iMaxNumber) que será processado. Este número deverá ser sempre maior que 5.
- O construtor irá inicializar os dados os veiculos da seguinte forma:
  - sPlaca = "PLC-000" + "posicao_no_vetor"
  - dValorMercado = Math.random() * 5000 * "posicao_no_vetor+1"
  - sCombustivel = O primeiro "GASOLINA"; o segundo "ALCOOL" e o terceiro "DIESEL" e assim por diante..

LINGUAGEM JAVA – EXEMPLO

  - blpvaPago = Inicialmente false para todos veiculos
- Outros métodos da classe:
  - imprimir os dados de todos os veiculos.
  - Calcular e imprimir o valor do Ipva de todos os veiculos (utilizando while)
  - Criar um método **pagarIpva** que recebe como parâmetro a placa do veículo. Este método deve localizar o veículo e alterar o valor **blpvaPago** de `false` para `true`.

JAVA – API DISPONÍVEL

- A linguagem JAVA possui uma rica API. A descrição da mesma está disponível no seguinte endereço:

  http://java.sun.com/j2se/1.4.1/docs/api

- A API está dividada em pacotes (Packages). Cada pacote contém um conjunto de classes que oferece funcionalidades relacionadas entre si.
- A linguagem Java é uma plataforma de desenvolvimento e existem classes disponíveis para as mais variadas aplicações e necessidades

JAVA – API DISPONÍVEL

java.Applet	Classes básicas para criar uma Applet
java.awt	Classes para a manipulação de gráficos e desenhos
java.io	Classes para Input/Output. Tratamento de arquivos
java.lang	Classes que contém a linguagem java. "Importada" em qualquer programa java
java.net	Classes para implementar aplicações de rede
java.rmi	Classes para **R**emote **M**ethod **I**nvocation. Permite uma máquina Java conversar com outra
java.security	Classes para implentar regras de segurança. Ex.: métodos de criptografia
java.sql	Classes para acesso e manipulação a banco de dados.
java.util	Classes para uso genérico. Ex.: Coleções; etc.
javax.swing	Classes para criação de interfaces gráficas de usuário(GUI)

JAVA – TRABALHANDO COM STRINGS

- Uma String consiste em uma sequências de caracteres.

  String str = "abc"; é equivalente a:

  char data[] = {'a', 'b', 'c'}; String str = new String(data);
- A linguagem java possui um suporte especial para a concatenação de Strings utilizado o operador "+"
- **CONSTRUTORES PRINCIPAIS**

`String()` – Cria uma string vazia

`String(String original)` – Cria uma cópia de uma string a partir da String original

`String("String_constante")` – Cria uma string a partir de um valor string constante

JAVA – TRABALHANDO COM STRINGS

- **MÉTODOS PRINCIPAIS**

`public boolean equals(Object anObject)` - Compara a String com qualquer outro objeto. Caso o outro objeto seja uma string de igual conteúdo retorna `true`

`public int length()` - Retorna o comprimento de uma string, o comprimento equivale ao número de caracteres.

`public String trim()` - Retorna uma cópia da string, removendo espaços em branco existentes no seu inicio e final.

`public char charAt(int index)` - Retorna o caracter existente na posição `index` da string.

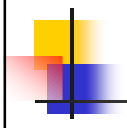
`public String toUpperCase()` - Converte os caracteres da string para sua representação em letra maiúscula.

JAVA – TRABALHANDO COM STRINGS

- A classe **Object** possui o método *toString()*, desta forma todas as outras classes herdam este método e muitas delas o especializam conforme sua necessidade.
- Outro método importante que a classe **String** oferece é o método *valueOf ()*. Este método pode ser utilizado para converter variáveis de diferentes tipos para Strings.

  Exemplo:

  System.out.println(String.valueOf(Math.PI));



JAVA – TRABALHANDO COM STRINGS

- A classe String não possui métodos para se converter em outros tipos como inteiro ou ponto flutuante. Para realizar esta operação é necessário a utilização das classes **Boolean; Byte; Short; Integer; Long e Double** que possuem métodos para realizar tal tarefa.

  Exemplo de Conversão:

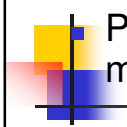
  Convertendo uma String em Float

```
float pi = Float.parseFloat("3.14");
```

  Convertendo Float em String

```
String s = Float.valueOf(3.1415);
```

- Para os outros tipos básicos o procedimento é o mesmo



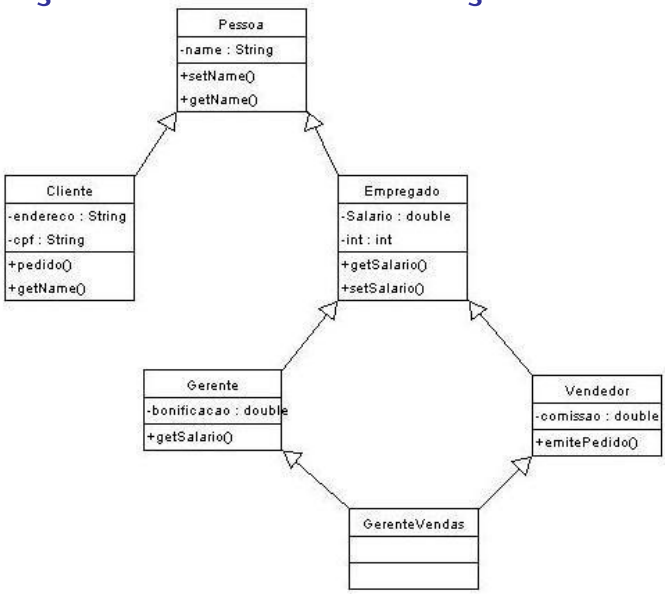
HERANÇA

- Herança é a capacidade de uma subclasse de ter acesso as propriedades da superclasse(também chamada classe base) relacionada a esta subclasse.
- Dessa forma os atributos e métodos de uma classe são propagados de cima para baixo em um diagrama de classes.
- Neste caso dizemos que a subclasse herda as propriedades e métodos da superclasse
- Os construtores da superclasse (classe base) não são herdados pela subclasse.
- A utilização da herança é um importante fator para a "reutilização de código"

HERANÇA

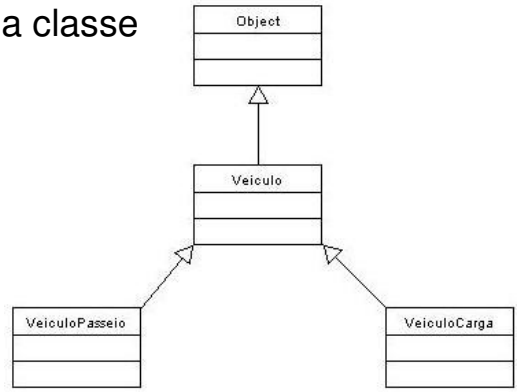
- A herança é uma capacidade característica das linguagens orientadas a objetos.
- A relação de herança entre duas classes é uma relação da seguinte forma: A "e um tipo de" B, onde A e B são classes. Caso esta relação entre as classes não puder ser construída, em geral, também não se tem uma relação de herança entre a classe A a partir da classe B.
- Exemplos: Um Carro de Passeio "é um tipo de " veículo; Um caminhão "é um tipo de" veículo; Um círculo "é um tipo de" figura; Um quadrado "é um tipo de" figura; Um vendedor "é um tipo de" Empregado; Um empregado "e um tipo de" pessoa.

HERANÇA - REPRESENTAÇÃO



HERANÇA - REPRESENTAÇÃO

- A Classe **Object** é a classe base da linguagem Java, ou seja, todas as outras classes herdam a classe **Object**
- Quanto mais alto na hierarquia de classe, mais generalizada é a classe



HERANÇA x USO

- Além da relação de herança entre as classes existe a relação de uso
  - HERANÇA

    classe A "é um tipo de" B
  - USO / AGREGAÇÃO (Relação de Conteúdo)

    classe D "contém" classe C"

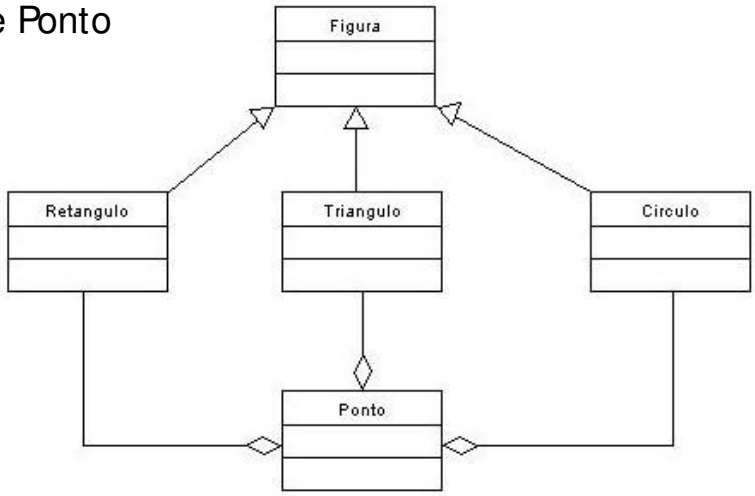
    classe D "usa" classe C"

    classe C "é parte da" classe D

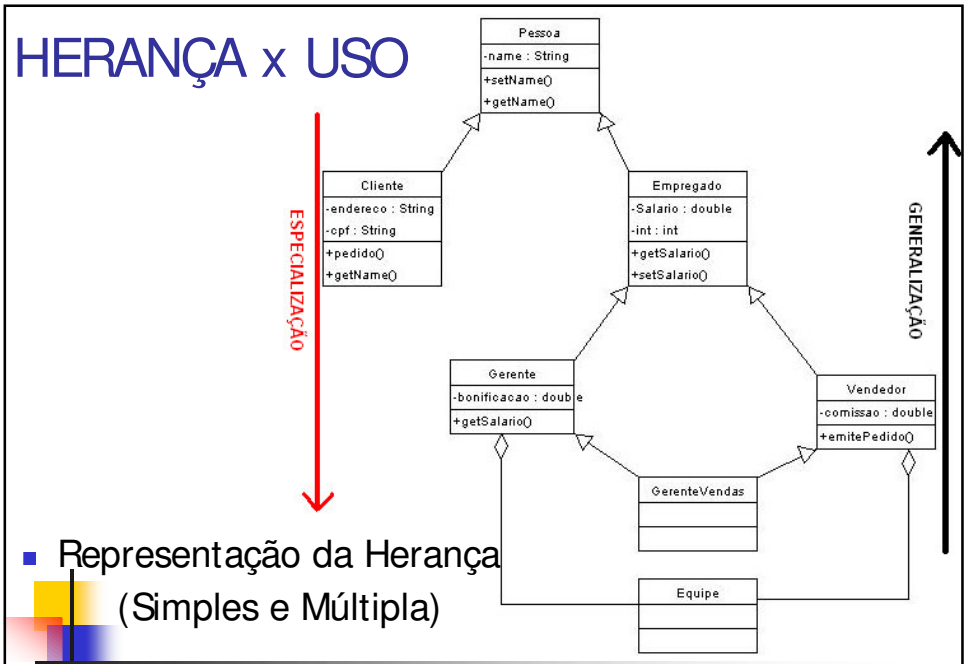
  Exemplo: Uma **Círculo** *contém* um **Ponto** central; Um **Triângulo** é *utiliza* três objetos da classe **Ponto**; Um **Ponto** *é parte* da classe **Quadrado**

HERANÇA x USO - REPRESENTAÇÃO

- Neste exemplo além da herança é representada a relação de uso, onde todos as outras figuras "usam" a classe Ponto



HERANÇA x USO



- Representação da Herança (Simples e Múltipla)

HERANÇA

- Para estabelecer uma relação de herança entre a classe a classe B e a classe A, deve ser utilizada keyword **extends**

  [public] [modTipo] class B ***extends*** A **[...]**
- Quanto mais alto na hierarquia de classe, mais generalizada é a classe
- Normalmente uma classe pode ser vista somente por outras classes no mesmo pacote. O modificador ***public*** indica que a classe será vista por qualquer outra classe
- Um **PACOTE** (**package**) consiste em um conjunto de classes relacionadas entre si.

HERANÇA

- Na criação de uma classe é possível utilizar modificadores de tipo que irão influenciar o comportamento da mesma.
- Modificador ***abstract:*** Indica que a classe não poderá ser instanciada
- Modificador ***final:*** Indica que a classe não poderá ser extendida

Exemplos:

public final class String ***extends*** Object

public class Triangulo ***extends*** Figura

class Empregado ***extends*** Pessoa

class VeiculoCarga ***extends*** Veiculo

abstract class Veiculo ***extends*** Object

HERANÇA – EXEMPLOS

```
class Pessoa {
  protected String name;
  //Construtor
  public Pessoa(){
    name = "";
  }
  public Pessoa(String n){
    name = n;
  }
  //Get Metodos
  String getName(){
    return name;
  }
```

```
//continuação...
//set Metodos
  void setName(String s){
    name = s;
  }
}
```

HERANÇA – EXEMPLOS

```
class Empregado extends
      Pessoa {
  protected double
      dSalario;
  //Construtor
  public Empregado(String
      n, double dS){
    name = n;
    dSalario = dS;
  }
```

```
//continuando...
//Get Metodos
  double getSalario(){
    return dSalario;
  }
//set Metodos
  void setSalario(double
      dS){
    dSalario = dS;
  }
}
```


HERANÇA – ACESSO A SUPERCLASSES

- A palavra reservada ***super***, permite acesso a métodos e construtores da superclasse
- **super(x1, x2, ..., xn)** - Permite a chamada do construtor da superclasse.
- Na utilização da herança é necessário a fim de chamar o construtor da superclasse, sempre que a superclasse não possui um construtor "default".
- Construtor "default" é aquele que não possui parâmetros
- Esta chamada deve ser a ***primeira*** dentro do construtor da classe. O construtor da classe imediatamente superior será chamado

HERANÇA – ACESSO A SUPERCLASSES

- **super.metodo(...)** - Permite que uma classe possa utilizar métodos definidos em sua superclasse
- Exemplos:

  String s;

  s = super.toString();

  Em algum método da classe empregado é possível a seguinte chamada:

  ...

  super.getName();

  ...

HERANÇA – EXEMPLOS

```
class Gerente extends
      Empregado {
  protected double
      dBonificacao;
  //Construtor
  public Gerente(String n,
      double dS, double
      dB){
    super(n,dS);
    dBonificacao = dB;
  }
double getBonificacao(){
    return dBonificacao;
  }
```

```
//continua...
  double getSalario(){
    return (dSalario +
      dBonificacao);
  }
  //set Metodos
  void
      setBonificacao(double
      dS){
    dBonificacao = dS;
  }
}
```

HERANÇA E POLIMORFISMO

- No exemplo anterior pode ser percebido que houve um polimorfismo no método **getSalario**.
- Definição do método na classe **Empregado**

```
double getSalario(){
  return dSalario;
}
```

- Definição no método na classe **Gerente**, que é subclasse de **Empregado**:

```
double getSalario(){
  return (dSalario + dBonificacao);
}
```

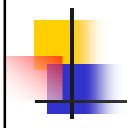
- O mesmo método apresenta um comportamento diferente para diferentes classes que possuem uma relação de herança

HERANÇA - CLASSES ABSTRATAS

- Algumas classes na hieraquia são tão gerais que nehum objeto será criado a partir delas. Neste caso a classe é dita **ABSTRATA**
- Uma classe abstrata não pode ser instanciada ou seja, não é possível criar objetos a partir da mesma
- A classe ABSTRATA é uma classe que está incompleta. Esta classe pode conter métodos abstratos que são aqueles métodos apenas declarados, mas que não foram implementados.
- Os métodos abstratos devem ser obrigatoriamente implementados nas subclasses.

HERANÇA - CLASSES ABSTRATAS

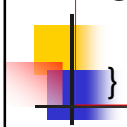
- O método abstrato contém apenas sua assinatura (nome, número e tipo dos seus parâmetros).
- Para a criação de classes e métodos abstratos deve ser utilizado o modificador de tipo "**abstract**"
- Classe CONCRETA é aquela a partir da qual objetos serão instanciados. Neste tipo de classe todos seus métodos devem ser, obrigatoriamente, definidos.



HERANÇA – CLASSES ABSTRATAS

```
abstract class Pessoa {
  protected String name;
  //Construtor
  //public Pessoa(){
  //  name = "";
  //}
  public Pessoa(String n){
    name = n;
  }
  //Get Metodos
  String getName(){
    return name;
  }
```

```
//continua...
//set Metodos
  void setName(String s){
    name = s;
  }
  abstract void
      printName();
}
```



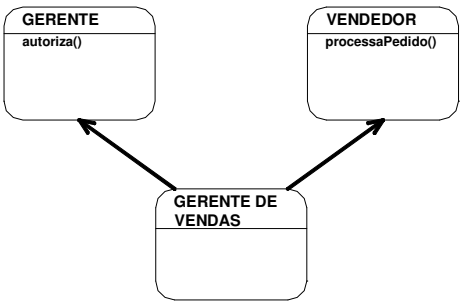
HERANÇA – CLASSES ABSTRATAS

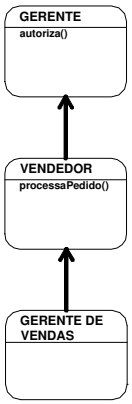
```
class Empregado extends
      Pessoa {
  protected double
      dSalario;
  //Construtor
  public Empregado(String
      n, double dS){
    super(n);
    // name = n;
    dSalario = dS;
  }
```

```
  //Get Metodos
  double getSalario(){
    return dSalario; }
  //set Metodos
  void setSalario(double
      dS){
    dSalario = dS; }
  void printName(){
      System.out.println(
      "Empregado: " +
      name);
  }
}
```


HERANÇA MÚLTIPLA

- Existem casos em que uma classe pode herdar o comportamento de mais de uma classe. Neste caso temos a herança múltipla. Exemplo:

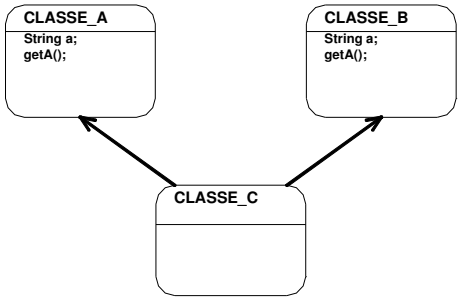




MODELAGEM INCORRETA

HERANÇA MÚLTIPLA

- Como implementar a herança múltipla:



- No exemplo acima, qual cópia do atributo **a** a classe CLASSE_C vai herdar? Qual método **getA()** vai utilizar?
- Java resolve este problema utilizando o conceito de "INTERFACES"

HERANÇA MÚLTIPLA

- Um método possui duas partes: sua assinatura e sua implementação
- Java não suporta a herança múltipla explicitamente, mas possui meios para que os efeitos da herança múltipla seja realizada de forma indireta utilizando o conceito de INTERFACES
- Através deste conceito uma classe pode herdar as assinaturas dos métodos, mas não a sua implementação.
- A implementação, deve por sua vez, ser definida na subclasse.

HERANÇA MÚLTIPLA

- Uma INTERFACE é definida através da palavra "**interface**" conforme mostrado a seguir:

  [public] interface B ***extends*** A

  Neste caso A deve ser outra interface.
- Exemplo – Definição da INTERFACE GerenteInt

```
interface GerenteInt{
  boolean autorizar();
}
```

- A indicação da herança múltipla é feita da seguinte forma:

[public] [modTipo] class B **[*extends*** A] **implements** C

HERANÇA MÚLTIPLA

- Na interface todos os métodos são abstratos e não possuem implentação apenas sua assinatura.
- A uma classe pode utilizar mais de uma interface em sua definição

HERANÇA MÚLTIPLA - EXEMPLO

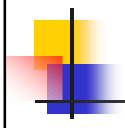
```
class GerenteVendas
   extends Vendedor
   implements
   GerenteInt{
  protected String
   sRegiao;
  //Construtor
  public
   GerenteVendas(String
   n, double dS, double
   dC, String sReg){
    super(n, dS, dC);
    sRegiao = sReg;
  }
  //continua..
  void printName(){

   System.out.println("Ve
   ndedor: " + name);  }

  public boolean
   autorizar(){
   System.out.println("Ve
   nda Autorizada");
    return true;  }
}
```


HERANÇA MÚLTIPLA

- Através da herança múltipla novos métodos, de diferentes classes, podem ser agregados a uma subclasse
- A herança através de interface não possibilita a reutilização do código, visto que o método herdado deve ser implementado para cada subclasse.
- ATRIBUTOS EM UMA INTERFACE: Em uma interface os atributos são implicitamente declarados como **static** e **final**.
- MÉTODOS EM UMA INTERFACE: Todos os métodos são abstratos, não sendo necessário a palavra "**abstract**"

HERANÇA MÚLTIPLA

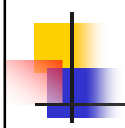
- **static:** Indica que existe apenas uma cópia do método ou varíavel, referenciados pela classe (método ou variável de classe)
- **final:** impede a modificação de um método, variável ou a especialização de uma classe.



HERANÇA MÚLTIPLA

- Exemplo: Métodos e Variáveis em interface

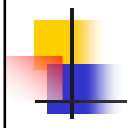
```
interface Cores{
  int RED = 1;
  int GREEN = 2;
  int BLUE = 3;
  void setCor(int c);
  int getCor();
}
```



HERANÇA MÚLTIPLA

- Uma classe abstrata que implenta alguma interface, deve conter a definição deste método. Exemplo:

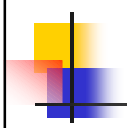
```
abstract class Colorido implements Cores {
   int i;
   Colorido() {}
   public void setCor(int c){
      i = c;
   }
   …
```



HERANÇA MÚLTIPLA

- Diferenças entre classe Abstrata e Interface

CLASSE ABSTRATA	INTERFACE
Pode conter alguns métodos declarados como **abstract**	Somente pode ter métodos abstratos
Pode conter atributos **protected** e métodos **static**	Somente pode conter métodos **public**
Pode conter atributos do tipo **final** e "não-final"	Somente pode conter constantes (implicitamente são public final static)



JAVA – TRATAMENTO DE EXCEÇÕES

- TRATAMENTO DE ERROS TRADICIONAL
  - O Erro é verificado e caso ocorra, é tratado no ponto é possível a sua ocorrência.
  - Ocorre uma mistura entre o código para resolver o problema e o código utilizado no tratamento do erro
  - Tratamento do erro, interfere na lógica do problema e alguns casos torna mais díficil a compreensão do mesmo.
  - Em algumas situações o erro deve ser propagado por vários níveis até que seja tratado.

TRATAMENTO DE EXCEÇÕES

- TRATAMENTO DE ERROS TRADICIONAL
  - Exemplo – Código sem tratamento de erros!

```
ler_um_arquivo
{
  abrir_arquivo;
  determinar_seu_tamanho;
  alocar_memória;
  ler_arquivo;
  fechar_arquivo;
}
```

TRATAMENTO DE EXCEÇÕES

- TRATAMENTO DE ERROS TRADICIONAL - Exemplo

```
errorCode ler_um_arquivo{
  errorCode = 0;
  abrir_arquivo;
  if (arquivo_aberto){

      determinar_seu_tamanho;
    if (tamanho_disponivel){
      alocar_memória;
      if (memoria_alocada){
        ler_arquivo;
```

```
if (erro_Leitura)
          errorCode = -4;
       } else
        errorCode = -3;
    } else
     errorCode = -2;
    fechar_arquivo;
    if (erro_Fechar_arquivo)
      errorCode = -5
  } else
     errorCode = -1;
  return errorCode; }
```


TRATAMENTO DE EXCEÇÕES

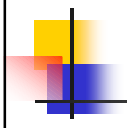
- TRATAMENTO DE ERROS TRADICIONAL - Exemplo

```
errorCode leArquivo{
  errorCode = 0;
  abrir_arquivo;
  if (arquivo_nao_aberto)
    return -1;
  determinar_seu_tamanho;
  if (tamanho_nao_disponivel) {
    fechar_arquivo;
    return -2;}
  alocar_memória;
  if (memoria_nao_alocada){
    fechar_arquivo;
    return -3;}
```

```
ler_arquivo;
 if (erro_Leitura){
   fechar_arquivo;
   return -4;}
 fechar_arquivo;
 if (erro_Fechar_arquivo)
    return -5
 return errorCode;}
```

TRATAMENTO DE EXCEÇÕES

- EXCEÇÃO
  - Indicação de um problema ocorrido durante o processamento
- TIPOS DE EXCEÇÃO
  - Erros de hardware; divisão por zero; tentativa de acessar elementos fora dos limites de um vetor; valores de parâmetros inválidos em um método; esgotamento da memória; utilizar um objeto não criado; etc.



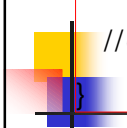
TRATAMENTO DE EXCEÇÕES

- VANTAGENS
  - Separação do código de tratamento de erro do código do programa
  - Propagação do erro através da pilha de funções

```
metodo1 {
 int error;
 error = metodo2();
 if (error)
   //ProcessaErro
 else
   //continua
}
```

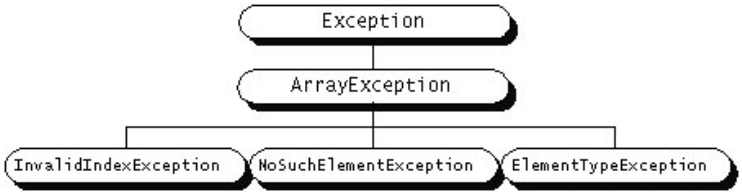
```
int metodo2() {
  int error;
  error = metodo3();
  if (error)
    return error;
  else
   //continua
}
```

```
int metodo3() {
  int error;
  error = leArquivo();
  if (error)
    return error;
  else
   //continua
}
```



TRATAMENTO DE EXCEÇÕES

- VANTAGENS
  - Agrupamento dos vários tipos de erros em classes



TRATAMENTO DE EXCEÇÕES

- QUANDO UTILIZAR
  - Em situações onde o método é incapaz de completar sua tarefa
  - Em situações onde o método não trata as exceções ocorridas, mas apenas sinaliza sua ocorrência, como em bibliotecas de classes.

TRATAMENTO DE EXCEÇÕES

- COMO UTILIZAR
  - Para utilizar o tratamento de exceções, basicamente é necessário "ouvir e capturar" as possíveis exceções que podem ocorrer e além disso "disparar" exceções.
  - Para "ouvir/capturar" utiliza-se os blocos "**`try/catch`**"
  - Para "disparar" exceções utiliza-se as instruções "throws/trow"
  - Além disso é possível criar classes específicas para o tratamento de exceções

OUVINDO / CAPTURANDO EXCEÇÕES

- As exceções que poderão ocorrer serão "ouvidas" através de um bloco que inicia pela instrução **`try`**
- Ao final deste bloco, um ou mais, blocos que iniciam pela instrução **`catch`**, irão "capturar" e tratar as exceções que podem ter sido disparadas.
- Após o último bloco **`catch`**, um bloco **`finally`** opcional fornece um código que sempre será executado e pode ser utilizado para evitar perdas de recurso (Ex. Arquivo aberto)
- Logo que uma exceção ocorre o processamento deixa o bloco **`try`** e começa a pesquisar nos blocos **`catch`** um tratamento específico para aquele tipo de erro

OUVINDO / CAPTURANDO EXCEÇÕES

- Exemplo:

```
try{
    ... }
catch(ExceptionType1 e){
    ... }
catch (ExceptionType2 e){
    ... }
catch (Exception e){
    ... }
finally {
    ...
}
```


DISPARANDO EXCEÇÕES

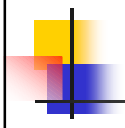
- Para que um método possa disparar uma exceção é necessário colocar a cláusula **`throws`** na definição do mesmo, indicando quais tipos de exceção o mesmo pode retornar
- O método que irá retornar a exceção deve criar a mesma. Uma exceção é também um objeto!
- Após criar a exceção a mesma deve ser disparada com a cláusula **`throw`**
- A classe base de exceções em java é a classe **`Exception`**

CRIANDO CLASSES DE EXCEÇÕES

- Para criar uma classe que será responsável pelo tratamento de exceções a mesma deve extender a classe **`Throwable`**

Exemplo:

```
public class MyException extends
    Throwable
```



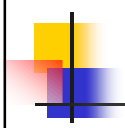
CRIANDO CLASSES DE EXCEÇÕES

- Importante:

catch (Exception e) → Captura todas as exceções

catch (Error err) → Captura todos os erros do tipo Error

catch (Throwable t) → Captura todos os erros do tipo Error e Exception



TRATAMENTO DE EXCEÇÕES - EXEMPLO

```
private double calculaTotal(String sC, String sJ, String sP, String sM) throws
        NumberFormatException {
  if (sC.equals(""))
   throw new NumberFormatException("O CAMPO CAPITAL ESTÁ VAZIO");
  if (sJ.equals(""))
    throw new NumberFormatException("O CAMPO JUROS ESTÁ VAZIO");
  if (sPrazo.equals(""))
    throw new NumberFormatException("O CAMPO ANOS ESTÁ VAZIO");
  //converte o texto para double
  dCapital = Double.parseDouble(sC);
  dJuros = Double.parseDouble(sJ);
  dPrazo = Double.parseDouble(sP);
  dTotal = 0;
  if (dPrazo == 0)
    throw new NumberFormatException("O NÚMERO DE ANOS É IGUAL A ZERO");
  if (sMetodo.equals(aMetodos[0])){
    dTotal = ((dCapital * (1 + (dJuros/100)*dPrazo))/(dPrazo))/12;
  if (sMetodo.equals(aMetodos[1])){
    dTotal = ((dCapital * Math.pow((1 + (dJuros/100)),dPrazo))/dPrazo)/12;
  }
  return dTotal;
}
```


TRATAMENTO DE EXCEÇÕES - EXEMPLO

```
try {
  dTotal = calculaTotal(txtCapI.getText(),txtJur.getText(), txtPra.getText(),sMetodoCalculo);
  ...
}
catch (NumberFormatException ex)
{
  JOptionPane.showMessageDialog(null,ex);
  JOptionPane.showMessageDialog(null,ex.getMessage());
}
```

TRATAMENTO DE EXCEÇÕES - EXEMPLO

```
try {
  dTotal = calculaTotal(txtCapI.getText(),txtJur.getText(), txtPra.getText(),sMetodoCalculo);
  ...
}
catch (NumberFormatException ex)
{
  JOptionPane.showMessageDialog(null,ex);
  JOptionPane.showMessageDialog(null,ex.getMessage());
}
```


ASSINATURA COMPLETA DE UM MÉTODO

- A assinatura de um método é criada da seguite forma:

[*nívelAcesso*] [*modificador*] *tRetorno* **nomeMetodo** (T0 P0,...,Tn Pn)
[***throws*** *tExceção*]

[*nívelAcesso*] - `private`; `public` ou `Protected`

[*modificador*] – Permite alterar a forma como o método se comporta (`static; abstract; final; native; synchronized`)

[*tRetorno*] - Tipo de retorno. Se não houver nenhum retorno, deve ser utilizado **`void`**. O tipo pode ser básico ou então qualquer objeto da linguagem

T0,...Tn – Tipos dos parâmetros utilizados

P0,...,Pn – Parâmetros

[***throws*** *tExceção*] – Tipo de exceção que pode ser disparada pelo método

ESTRUTURA COMPLETA DE UMA CLASSE

```
[nívelAcesso] [modificadorClasse] class NomeClasse
[extends baseClass] [implements interface1, interface2,...] {
    [nívelAcesso] [modificadorAtributo] tipo variavel_membro1;
    [nívelAcesso] [modificadorAtributo] tipo variavel_membroN;
    //Construtores
    [nívelAcesso] NomeClasse(T0 P0, ..., Tn Pn);
    //Destrutor
    protected void finalize();
    //Modificadores
    [nívelAcesso] [modificador] tRetorno setXXX(T0 P0,...,Tn Pn) ...
    //Acessores
    [nívelAcesso] [modificador] tRetorno getXXX(T0 P0,...,Tn Pn) ...
    //Outros métodos (privados; protegidos e públicos)
    [nívelAcesso] [modificador] tRetorno metodo(T0 P0,...,Tn Pn)
    [throws tExceção]
    }
```


ESTRUTURA COMPLETA DE UMA CLASSE

- A seguir é mostrado os possíveis valores utilizados na construção da classe:
  - [*nívelAcesso*] - `private`; `public` ou `Protected`
  - [*modificadorClasse*] - Permite alterar a forma como a classe se comporta (`abstract`;`final`)
  - **baseClass** – Classe base utilizada pela herança
  - **interface$_i$..** – Nome da interface implementada
  - [*modificadorAtributo*] - Permite alterar a forma como o atributo se comporta (`static`; `abstract`; `final`; `transient`; `volatile`)
  - [***tipo***] – Tipo da variável membro ou atributo. O tipo pode ser básico ou então qualquer objeto da linguagem

ESTRUTURA COMPLETA DE UMA CLASSE

[*modificador*] – Permite alterar a forma como o método se comporta (`static; abstract; final; native; synchronized`)

[***tRetorno***] - Tipo de retorno do método. Se não houver nenhum retorno, deve ser utilizado **`void`**. O tipo pode ser básico ou então qualquer objeto da linguagem

T0,...Tn – Tipos dos parâmetros utilizados

P0,...,Pn – Parâmetros

[***throws*** *tExceção*] – Tipo de exceção que pode ser disparada pelo método

PACOTES

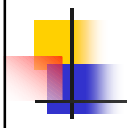
- Um pacote (package) consiste em um conjunto de classes e interfaces, relacionados entre si e normalmente sua criação está ligada a composição de bibliotecas
- Através da utilização de pacotes é possível utilizar classes com mesmo nome, porém proveniente de diferentes pacotes.
- Por convenção os pacotes devem ter o seguinte nome: com.company.package
- Toda classe pertence a um pacote, quando o nome do pacote não é informado, o compilador considera que classe pertence ao pacote "default" que neste caso é o diretório corrente.

PACOTES

- Para criar um pacote basta colocar a seguinte declaração no arquivo .java:

```
package nomeDoPacote
```

- Esta declaração deve ser a primeira declaração existente no arquivo.
- Somente as classes, métodos e variáveis públicas (public) podem ser utilizadas externamente ao pacote.



PACOTES

- Para organizar os arquivos em um pacote a regra é a seguinte:
  - Criar cada classe ou interface em um arquivo .java diferente.
  - Colocar estes arquivos em uma hierarquia de diretório equivalente à estrutura de nome do pacote

Exemplo:

Nome do pacote – com.autoenge.people

Arquivos: **pessoa.java; cliente.java; empregado.java; vendedor.java; gerentel nt.java; gerenteVendas.java**

Diretório onde estão os arquivos: / **com**/ **autoenge**/ **people**

PACOTES

- Uma forma para utilizar uma classe contida em um pacote é qualificando seu nome completamente:

  `java.io.File`

  `com.autoenge.people.vendedor`

  Exemplo:

  File d = new java.io.File("fname.txt");

- Para evitar a qualificação completa do nome do pacote basta utilizar a keyword "**import**"

  `import nomeCompletoDoPacote.className`

- Para importar todas as classes existentes em um pacote utiliza-se a seguinte sintaxe:

  `import nomeCompletoDoPacote.*`

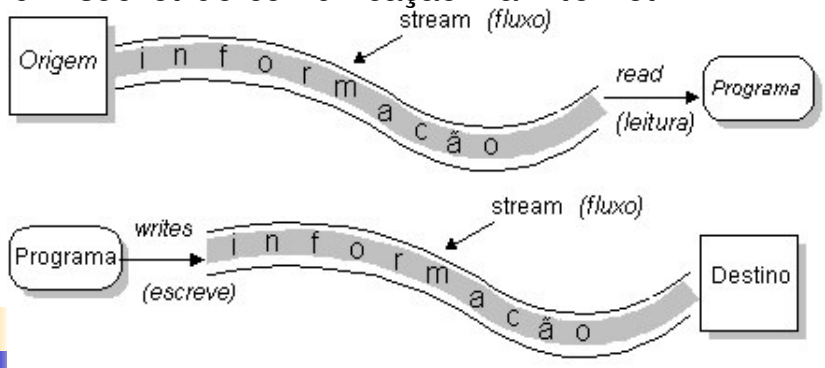
PACOTES

- Dois pacotes são automaticamente carregados quando se trabalha com a linguagem java:
  - O pacote java.lang
  - O pacote default (diretório corrente)
- Exemplo

```
import java.io.*;
File d = new File("fname.txt")
```

ENTRADA E SAÍDA (I/O)

- O conceito de **Stream** pode ser entendido como um fluxo de informação que pode entrar ou sair de um programa para uma fonte de informação que pode ser um arquivo, a memória ou mesmo um socket de comunicação via internet.



ENTRADA E SAÍDA (I/O)

- As operações de entrada e saída podem ser divididas em dois grandes grupos: Entrada (Read) e Saida (Write)
- Independente da origem ou destino o processo de leitura e escrita envolve os seguinte passos:

LEITURA (READ)	ESCRITA (WRITE)
Abre uma Stream while (existe_informação) le_informação (read) Fecha a Stream	Abre uma Stream while (existe_informação) escreve_informação (write) Fecha a Stream

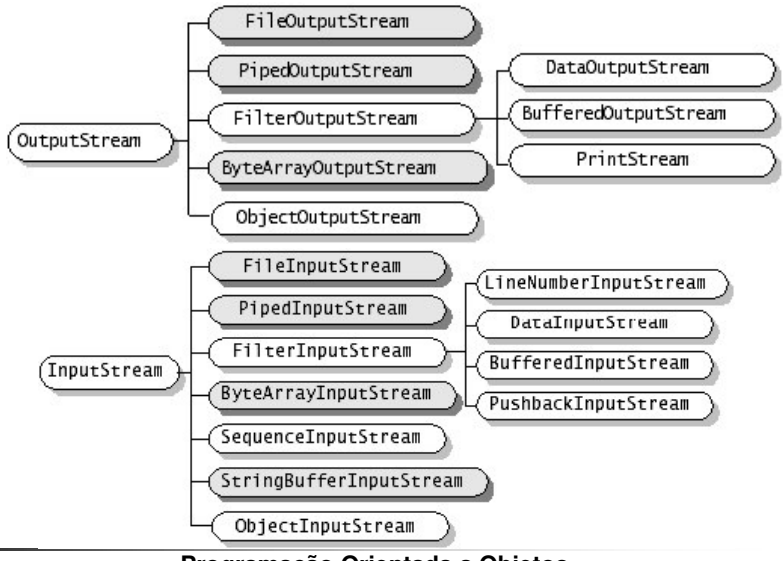
- O pacote **java.io** contém uma série de classes para a manipulação de entrada e saida.

ENTRADA E SAÍDA (I/O)

- Para o trabalho com I/O existem dois tipos de stream: Byte Streams e Character Streams
- Byte Streams (Fluxos de Bytes), permite a escrita e leitura de bytes (8 bits). Este tipo de stream normalmente é utilizado para a manipulação de dados binários como por exemplo, imagens e sons. **InputStream** e **OutputStream** são classes abstratas e super classes das byte streams.
- Character Streams(Fluxos de caracteres) permite a manipulação de caracters com 16 bits (unicode). **Reader** e **Writer** são classes básicas e super classes das Character Streams.
- As classes Bytes Streams são da primeira versão de i/o utilizada por java (desde 1.0)

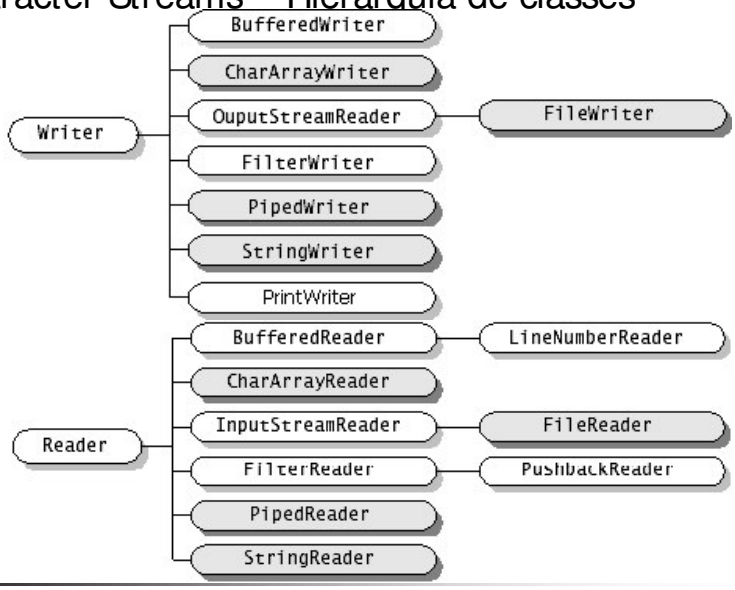
ENTRADA E SAÍDA – CLASSES

- Byte Streams – Hierarquia de classes



ENTRADA E SAÍDA – CLASSES

- Character Streams – Hierarquia de classes



ENTRADA E SAÍDA (I/O) - ARQUIVOS

- Entre as classes de entrada e saída podemos destacar:

Byte Streams	Character Streams
FileInputStream	FileReader
FileOutputStream	FileWriter

- As classes acima são utilizadas para leitura e escrita no sistema de arquivos nativo.
- Normalmente as classes acima utilizam a classe **File** que representa arquivos e diretórios no sistema nativo, permitindo sua manipulação

ENTRADA E SAÍDA (I/O) - BUFFERIZADA

- Para a leitura e escrita bufferizadas em arquivos são utilizadas as seguinte classes:

Byte Streams	Character Streams
BufferedInputStream	BufferedReader
BufferedOutputStream	BufferedWriter

- A utilização de um buffer reduz o número de acessos necessários e desta forma este tipo de i/o é mais efeciente, devendo ser utilizado sempre que possível
- Noramalmente esta classe é utilizada em conjunto com alguma outra classe, com o objetivo de adicionar o buffer na classe em questão.

ENTRADA E SAÍDA (I/O) - BUFFERIZADA

- Para criar objetos das classes acima é necessário utilizar as classes **Reader e Writer** no caso das character streams.

```
BufferedReader in = new BufferedReader(new FileReader("foo.in"));
BufferedWrite out = new BufferedWriter(new FileWrite("foo.in"));
```

- Para ler um arquivo linha por linha deve ser utilizada a classe **LineNumberReader**
- Para escrever arquivo linha por linha deve ser utilizada a classe **PrintWriter/ PrintStream**

```
PrintWriter out = new PrintWriter(new BufferedWriter (new FileWriter
("foo.out")));
```

ENTRADA E SAÍDA (I/O) - TIPOS BÁSICOS

- As classes abaixo permitem que uma aplicação faça a leitura e escrita de tipos primitivos de dados da linguagem java.

Byte Streams	Character Streams
▪ DataInputStream ▪ DataOutputStream	▪ Não apresenta esta funcionalidade!

- As classes acima são um importante recurso para facilitar a leitura/escrita formatada de dados.
- Para realizar a leitura/escrita formatada de dados das classes **Reader** e **Writer** é necessário sua conversão utilizando as classes: **InputStreamReader** e **OutputStreamWriter**

ENTRADA E SAÍDA (I/O) - OBJETOS

- As classes abaixo permitem a leitura e a escrita de objetos em geral.

Byte Streams	Character Streams
▪ ObjectInputStream ▪ ObjectOutputStream	▪ Não apresenta esta funcionalidade!

- O processo de leitura e escrita de objetos é chamado "Serialização de Objetos".
- É possível utilizar um buffer durante a serialização de objetos. Para isto uma stream do tipo **BufferedInputStream** ou **ObjectOutputStream** de ser adicionada ao um fluxo dos tipos acima.

ENTRADA E SAÍDA (I/O) - OBJETOS

- **Exemplos**: Abaixo é mostrado como encadear vários tipos de streams a fim de obter o resultado desejado. Neste caso está sendo feita uma escrita e leitura de objetos em um arquivo do sistema utilizando buffers.
  - **Escrita(encadeando várias tipos de streams)**

```
FileOutputStream fos = new FileOutputStream("t.obj");
BufferedOutputStream bos = new ufferedOutputStream(fos);
ObjectOutputStream oos = new ObjectOutputStream(bos);
```

  - **Leitura(encadeando várias tipos de streams)**

```
FileInputStream fis = new FileInputStream("t.tmp");
BufferedInputStream bis = new BufferedInputStream (fis);
ObjectInputStream ois = new ObjectInputStream(bis);
```


ENTRADA E SAÍDA (I/O) - SERIALIZAÇÃO

- A serialização permite a leitura e escrita em arquivos dos dados contidos em objetos.
- Os dados são gravados em forma de Byte Streams.
- Para que uma classe possa ser serializada basta que a mesma implemente a interface **Serializable**
- Para gravar o objeto deve ser utilizado o seguinte método:

```
public final void writeObject(Object obj)
oos.writeObject(new Veiculo(...));
```

- Para ler o objeto deve ser utilizado o seguinte método:

```
public final Object readObject()
ois.readObject(new Veiculo(...));
```

ENTRADA E SAÍDA (I/O) - CONVERSÃO

- Em muitas situações é necessária a conversão de Byte Stream para Character Stream a vice-versa para isto são utilizadas seguinte classes:

Byte Streams → Character Streams
InputStreamReader

Character Streams → ByteStreams
OutputStreamWriter

- Uma **InputStreamReader** lê bytes de uma InputStream a converte-os para caracteres. Já uma **OutputStreamWriter** converte character para bytes e envia-os para uma OutputStream
- Exemplos:

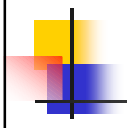
```
Writer out = new BufferedWriter(new OutputStreamWriter ( System.out));
BufferedReader in = new BufferedReader(new InputStreamReader
    (System.in));
```


ENTRADA E SAÍDA (I/O) PADRÃO

- Os dispositivos de entrada e saída padrões são mantidos por variáveis estáticas da classe System e podem ser referenciadas da seguinte forma:
  - Entrada Padrão
    `System.in` (public static final InputStream in)
  - Saida Padrão
    `System.out` (public static final PrintStream out)
  - Saída de Erro Padrão
    `System.err` (public static final PrintStream err)
- Normalmente as variáveis acima apontam para o console.
- Para utilizar as classes **Writer** e **Reader** é necessário utilizar as classes de conversão: **OutputStreamWriter** e **InputStreamReader**

ENTRADA E SAÍDA (I/O)-Acesso Randômico

- As classes vistas anteriormente utilizam acesso sequencial aos dados
- A classe **RandomAccessFile** permite o acesso ao arquivo de forma randômica. Neste tipo de acesso o arquivo se comporta como um longo array de bytes localizado no disco. Um índice (ponteiro do arquivo) permite a manipulação em qualquer posição válida do arquivo.
- Esta classe pode ser utilizada tanto para leitura quando escrita.



ENTRADA E SAÍDA (I/O)-Acesso Randômico

- Criando um arquivo de acesso randômico(aleatório)

RandomAccessFile(String name, String mode) – A string name contém o nome do arquivo e a string mode contém o modo de manipulação do mesmo:

"r" - somente leitura

"rw" - leitura e escrita

- Métodos principais:

void seek(long pos) - Posiciona o ponteiro na posição pos. A posição inicial é igual a 0.

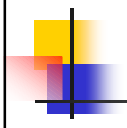
long length() – Retorna o comprimento do arquivo em bytes

void writeBytes(String s) – grava uma string como uma sequência de bytes

String readLine() - Lê uma linha de texto

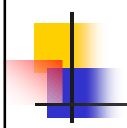
ENTRADA E SAÍDA (I/O) - EXCEÇÕES

- A maioria dos métodos existentes nas classes do pacote **java.io** disparam exceções cuja classe base normalmente é a classe **IOException**
- Entre as sub classes **IOException** podemos citar: **EOFException; FileNotFoundException;...**
- Dessa forma normalmente o código envolvendo operações de entrada e saida quase sempre irá utilizar o bloco **try/ catch**



JAVA – INTERFACE GRÁFICA

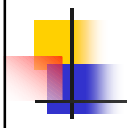
- A Linguagem JAVA e o paradigma orientado a objetos fornecem uma série de recursos que permitem a criação da interface gráfica com usuário (GUI)
- Os conceitos da programação orientada à objetos como: Herança; Polimorfismo e Sobrecarragamento entre outros permitem que a programação seja feita utilizando uma série de classes e métodos que estão disponíveis – Rapidez e qualidade no projeto



JAVA – INTERFACE GRÁFICA

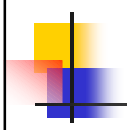
- Podemos classificar as aplicações em três tipos:
  - APLICAÇÕES BASEADAS EM CONSOLE
  - APLICAÇÕES GRÁFICAS BASEADAS EM JANELAS
  - APLICAÇÕES GRÁFICAS BASESADAS NA INTERNET (APPLETs)
- Maiores informações

http://java.sun.com/docs/books/tutorial/uiswing/TOC.html



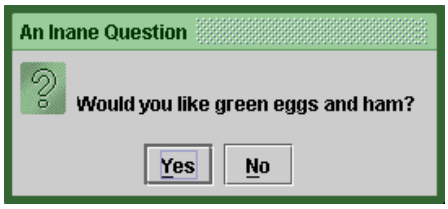
JAVA – INTERFACE GRÁFICA

- APLICAÇÕES BASEADAS EM CONSOLE
  - Não possuem interface gráfica
  - Utilizam o console do sistema
  - Interação é feita através de texto apenas
  - Consiste de uma classe qualquer derivada de Object e devem possuir a função - **public static void main(String args[])**
  - Saída de dados
    - `System.out`
  - Entrada de dados (classes)
    - `System.in`
    - `BufferedReader`
    - `InputStreamReader`



JAVA – INTERFACE GRÁFICA

- APLICAÇÕES BASEADAS EM JANELA WINDOW
  - Possuem interface gráfica
  - Classe normalmente derivada de **`JFrame`** (Window) ou então **`JDialog`** (Caixa de diálogo)



  - Este tipo de aplicação é executada diretamente sobre a plataforma gráfica (Windows; KDE; etc.)

JAVA – INTERFACE GRÁFICA

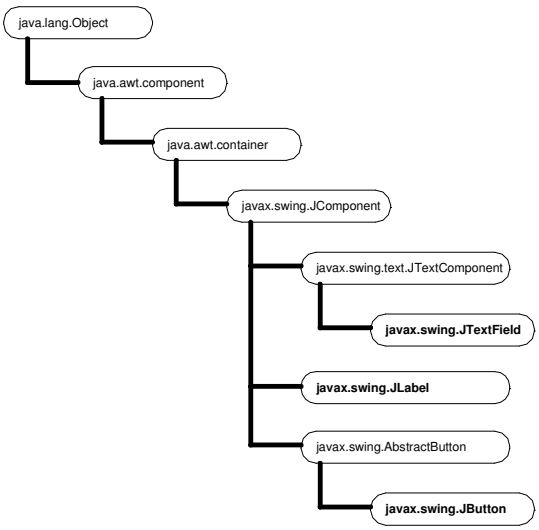
- APLICAÇÕES BASEADAS EM JANELA WINDOW
  - Consiste de uma janela que possui borda, um título e botões (maximizar; minimizar; fechar; etc.
  - JFrame – Janela
  - JDialog – Janela dependente de outra
  - JInternalFrame – Janela interna a uma outra

INTERFACE GRÁFICA

- A linguagem java possui dois pacotes para a criação de interfaces gráficas:
- AWT (Abstract Windowing ToolKit) – Conjunto de classes para criação de aplicações que usam a interface gráfica.
- SWING – Parte da JFC, toda escrita em java, que implementa uma série de componentes gráficos para interface com o usuário. Os componentes podem ser utilizados em multiplataformas. Esta "biblioteca" implementa os componentes existentes no conjunto AWT (Button; Scrollbar; Label; etc.) e outros como (tree view; list box; tabbed panes; etc.)

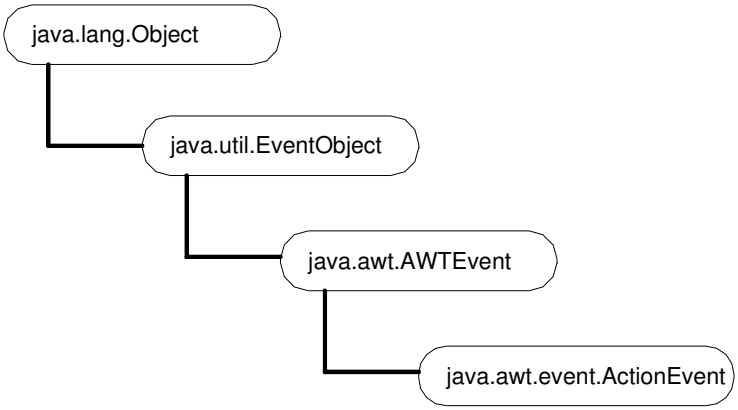
HIERARQUIA DOS COMPONENTES

- Classes definidas pelo pacote Swing (import javax.swing.*;)



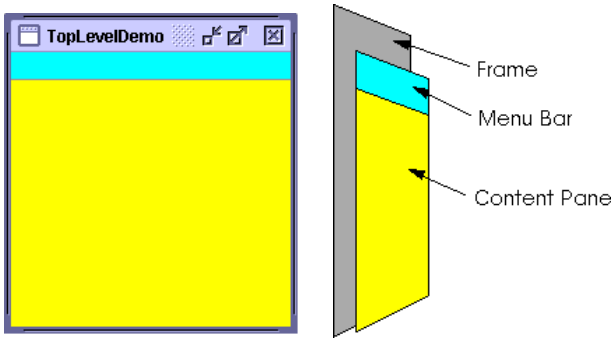
HIERARQUIA DOS EVENTOS

- Classes para manipulação de eventos (import java.awt.event.*;)



INTERFACE GRÁFICA- Window

- ESTRUTURA DE UMA JANELA



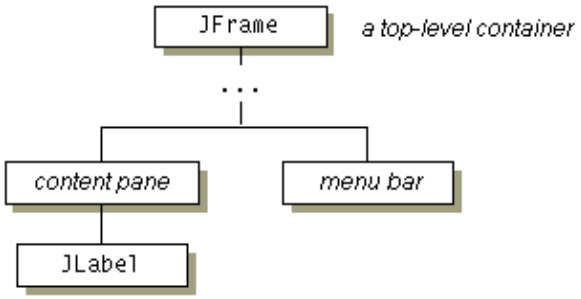
- CONTAINER – Objeto que irá conter os controles da interface do usuário. Para ser visto o controle deve estar ligado a algum **container**.

INTERFACE GRÁFICA- Window

- Uma aplicação sempre possui um **container** principal que é a raiz de todos os outros.
- Pode ser adicionado ao **container** uma barra de menus. Esta barra será posicionada no topo do mesmo
- Para recuperar um container de um JFrame, por exemplo, deve utilizar o seguinte método:
  `Container c = getContentPane();`

INTERFACE GRÁFICA- Window

- ESTRUTURA DE UMA JANELA - continuação



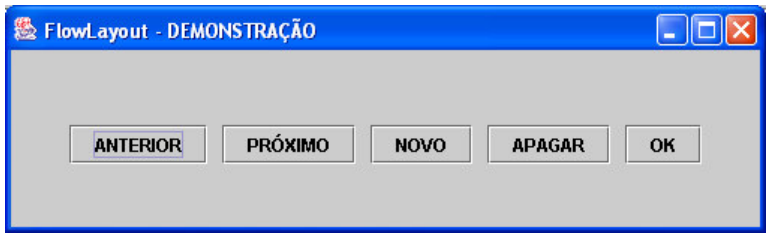
- Existem vários tipos de containers utilizados pela linguagem Java.

INTERFACE GRÁFICA- Window

- Entre os tipos de containers podemos citar:
  - BorderLayout
  - FlowLayout
  - GridLayout
  - BoxLayout
  - CardLayout
  - GridBagLayout
- Container facilita a disposição e o gerenciamento dos objetos que fazem parte da interface gráfica, ao invés de informar a posição específica de cada objeto da interface.

INTERFACE GRÁFICA- LAYOUTS

- **FlowLayout**
  - Componentes dispostos em uma linha, sequencialmente da esquerda para direita na ordem em que foram adicionados.



  - Caso o espaço de uma linha não seja suficiente, múltiplas linhas são utilizadas.

INTERFACE GRÁFICA- LAYOUTS

- **FlowLayout**
  - Os componentes podem ser dispostos da seguinte forma:

CENTRALIZADOS (**`FlowLayout.CENTER`**);

ALINHADOS À ESQUERDA (**`FlowLayout.LEFT`**)

ALINHADOS À DIREITA (**`FlowLayout.RIGHT`**)

```
public FlowLayout(int align, int hgap, int vgap)
align – alinhamento dos componentes
hgap – distância na horizontal entre componentes
vgap – distância na vertical entre componentes
```


INTERFACE GRÁFICA- LAYOUTS

- **FlowLayout – Exemplo**

```
...
//Recupera o container da janela (JFrame)
Container c = getContentPane();
//Ajusta o modo de gerenciamento
c.setLayout(new
FlowLayout(FlowLayout.CENTER,10,50));
//adiciona componentes(botões) ao container
btnAnt = new JButton("ANTERIOR");
c.add(btnAnt);
btnProx = new JButton("PRÓXIMO");
c.add(btnProx);
```

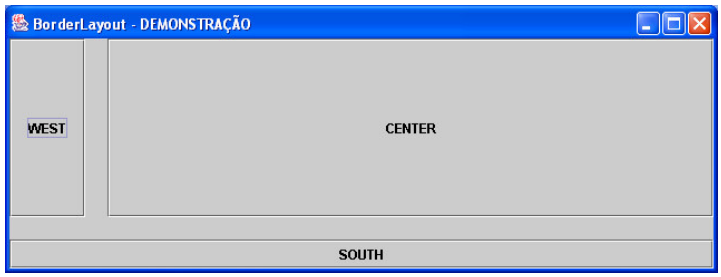
INTERFACE GRÁFICA- LAYOUTS

- **FlowLayout – Exemplo**

```
//continua...
btnNew = new JButton("NOVO");
c.add(btnNew);
btnDelete = new JButton("APAGAR");
c.add(btnDelete);
bntOk = new JButton("OK");
c.add(bntOk);
...
```


INTERFACE GRÁFICA- LAYOUTS

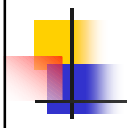
- **BorderLayout**
  - Componentes dispostos cinco regiões: Norte(NORTH); Sul(SOUTH); Leste(EAST); Oeste(WEST) e Centro (CENTER). Cada região pode conter no máximo um componente.



INTERFACE GRÁFICA- LAYOUTS

- **BorderLayout**
  - Um Componente ocupa toda a área de uma região.
  - Componente CENTRAL expande e ocupa áreas não utilizadas (LESTE e/ou OESTE). Se área centro não é utilizada a mesma é deixada vazia.

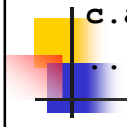
```
public BorderLayout(int hgap, int vgap)
hgap – distância na horizontal entre componentes
vgap – distância na vertical entre componentes
```



INTERFACE GRÁFICA- LAYOUTS

- **BorderLayout – Exemplo**

```
...
//Recupera o container da janela (JFrame)
Container c = getContentPane();
//Ajusta o modo de gerenciamento
c.setLayout(new BorderLayout(20,20));
btnNew = new JButton("WEST");
c.add(btnNew,BorderLayout.WEST);
btnDelete = new JButton("CENTER");
c.add(btnDelete,BorderLayout.CENTER);
bntOk = new JButton("SOUTH");
c.add(bntOk,BorderLayout.SOUTH);
...
```



INTERFACE GRÁFICA- LAYOUTS

- **GridLayout**
  - A àrea é dividida em retângulos iguais, conforme o número de linhas e colunas especificadas. Um único componente ocupa toda a área deste retângulo

INTERFACE GRÁFICA- LAYOUTS

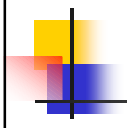
- **GridLayout**
- Quando o número de linhas é especificado o número de colunas é calculado automaticamente, conforme a quantidade de objetos existentes. Se o número de linhas é igual a zero, a quantidade de colunas é respeitada.

```
public GridLayout(int rows, int cols,
int hgap, int vgap)
row – número de linhas
cols – número de colunas
hgap – distância na horizontal entre
componentes
vgap – distância na vertical entre componentes
```

INTERFACE GRÁFICA- LAYOUTS

- **GridLayout – Exemplo**

```
...
//Recupera o container da janela (JFrame)
Container c = getContentPane();
//Ajusta o modo de gerenciamento
//Apesar de ser indiciado 7 colunas
//apenas 3 serão mostradas pois foi
//especificado o número de 2 linhas
c.setLayout(new GridLayout(2,7,10,10));
//continua...
```



INTERFACE GRÁFICA- LAYOUTS

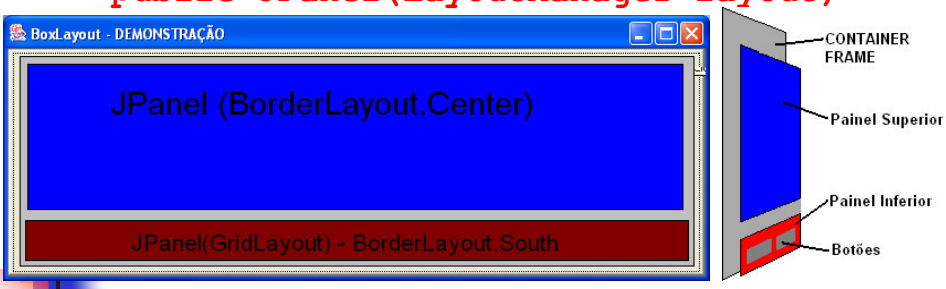
- **GridLayout – Exemplo**

```
btnAnt = new JButton("ANTERIOR");
c.add(btnAnt);
btnProx = new JButton("PRÓXIMO");
c.add(btnProx);
btnNew = new JButton("NOVO");
c.add(btnNew);
btnDelete = new JButton("APAGAR");
c.add(btnDelete);
bntOk = new JButton("OK");
c.add(bntOk);
...
```

INTERFACE GRÁFICA- PAINÉIS

- Normalmente utiliza-se em uma interface mais de um gerenciador de layout.
- O componente **JPanel** é muito útil para organizar os elementos da interface. JPanel é semelhante a um container onde é possível estabelecer como será seu layout interno

```
public JPanel(LayoutManager layout)
```



INTERFACE GRÁFICA- PAINÉIS

```
...
super("Layout - JPanel - DEMONSTRAÇÃO");
Container c = getContentPane();
brdLayout = new BorderLayout();
c.setLayout(brdLayout);
pnlPainel = new JPanel();
c.add(pnlPainel, BorderLayout.CENTER);
pnlPainel.setBackground(Color.blue);
pnlBotes = new JPanel(new
   GridLayout(0,5,20,20));
btnAnt = new JButton("ANTERIOR");
//continua...
```

INTERFACE GRÁFICA- PAINÉIS

```
pnlBotes.add(btnAnt);
btnProx = new JButton("PRÓXIMO");
pnlBotes.add(btnProx);
btnNew = new JButton("NOVO");
pnlBotes.add(btnNew);
bntOk = new JButton("OK");
pnlBotes.add(bntOk);
c.add(pnlBotes, BorderLayout.SOUTH);
pnlBotes.setBackground(Color.red);
...
```

INTERFACE GRÁFICA- EVENTOS

- EVENTO – Ação que ocorre e que pode "percebida" por um objeto
- Quando ocorre um evento o objeto que o recebeu é notificado. Caso o objeto ofereça uma **resposta** ao tipo de evento recebido, o evento então será tratado pelo objeto
- Para se trabalhar com eventos é necessário:
  - Declarar uma classe (handler) que será responsável pelo tratamento.
  - Implementar o tratamento (*handler*) para o evento na classe criada
  - Associar um "ouvinte" (*listener*) para um determinado tipo de evento

INTERFACE GRÁFICA- EVENTOS

- Tipos de eventos

AÇÃO PRODUTORA DO EVENTO	CLASSE OUVINTE (LISTENER)
Clique de um botão; Digitar <enter> após digitar um texto; escolher um item de um menu	ActionListener
Fechar uma janela; Minimizar; Restaurar o tamanho original; Ativar; Destativar; etc.	WindowListener
Pressionar o botão do mouse; Soltar o botão; Passar o Mouse sobre um componente	MouseListener
Movimentar o mouse; Arrastar (clicar e movimentar)	MouseMotionListener
Tornar um componente visível; Alterar a posição de um componente	ComponentListener
Componente recebe o foco (cursor) do teclado; Componente perde o foco	FocusListener
Elemento selecionado em uma lista é alterado (JList; Jtable)	ListSelectionListener
Propriedade de um componente é alterada	PropertyChangeListener
Utilização de teclado (Pressionar uma tecla; digitar uma tecla; soltar uma tecla	KeyListener
Para maiores informações - veja a classe ouvinte base	EventListener

INTERFACE GRÁFICA- EVENTOS

- EXEMPLO – Clique de um botão

```
//Para tratar um clique de botão então será
//utilizada a interface ActionListener
class ButtonHandler implements ActionListener{
//A interface ActionListener possui um método
//actionPerformed que será disparado sempre
//que o evento ocorrer
public void actionPerformed(ActionEvent e){
  String s =  "Botão NEW pressionado. Por
  enquanto só faço isto!";
  //Mostra uma mensagem na tela
  JOptionPane.showMessageDialog(null,s);
  }
}
```

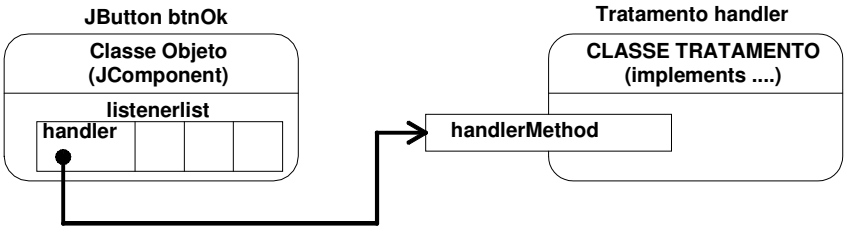
INTERFACE GRÁFICA- EVENTOS

- EXEMPLO – Clique de um botão

```
//Utilização da classe ButtonHandler
private JButton bntOk;
...
//Objeto que será responsável por tratar
//o evento (handler)
ButtonHandler handler = new ButtonHandler();
//Associar um ouvinte de eventos ao
//objeto da interface gráfica
btnNew.addActionListener(handler);
```


INTERFACE GRÁFICA- EVENTOS

- EXEMPLO – Clique de um botão



- Na inicialização da aplicação o objeto é criado (Classe Objeto). Além disso é criado o objeto que será responsável por tratar um determinado tipo de evento (Classe Tratamento)
- O Objeto criado para tratamento é associado ao objeto. A lista de ouvintes do objeto irá então possuir um instância do objeto.

INTERFACE GRÁFICA- EVENTOS

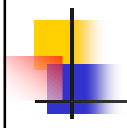
- EXEMPLO – Clique de um botão
- Quando o evento for disparado, o componente é notificado do evento que ocorreu. Caso o objeto possua algum tratamento para aquele tipo de evento o método para tratamento será então executado (handlerMethod)
- No exemplo do botão:
  - **Classe tratamento** – class ButtonHandler implents actionListener
  - **Objeto** – JButton btnOk
  - **Objeto Tratamento** – ButtonHandler handler
  - **handlerMethod** – public void actionPerfomed(ActionEvent e)

INTERFACE GRÁFICA- EVENTOS

- Para o tratamento de eventos existem dois conceitos importantes normalmente são utilizados
  - **CLASSE INTERNA** – Neste caso a classe para o tratamento dos eventos é criada internamente à definição da classe que contém a interface.
  - Esta classe não pode ser acessada externamente.
  - Neste caso é possível acessar os objetos da interface gráfica dentro da definição da classe que irá tratar os eventos

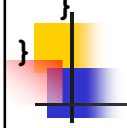
INTERFACE GRÁFICA- EVENTOS

```
public class JButtonSample2 extends JFrame{
  private JButton btnProx, btnAnt, btnNew,
   btnDelete, btnOk;
  public JButtonSample2(){
    ...
    ButtonHandler handler = new
   ButtonHandler();
    btnNew.addActionListener(handler);
    ...
  }
```



INTERFACE GRÁFICA- EVENTOS

```
//Classe Interna para tratamento de eventos
  class ButtonHandler implements
   ActionListener{
    public void actionPerformed(ActionEvent e)
    {
      ...
      if (obj == btnNew)  //btnNew pode ser
  acessado na classe interna!
        s = "Botão NEW pressionado. Por
        enquanto só faço isto!";
    }
  }
}
```



INTERFACE GRÁFICA- EVENTOS

- **CLASSE ANÔMIMA** – Neste caso a classe para o tratamento dos eventos é criada internamente à definição da classe que contém a interface. Porém esta classe não possui nome. Esta classe não pode ser acessada externamente.
- Esta classe permite que os objetos da interface sejam acessados dentro da mesma

```
public class JButtonSample3 extends JFrame{
  private JButton btnProx, btnAnt, btnNew,
   btnDelete, btnOk;
  ...
 public JButtonSample3(){
  //CLASSE ANÔNIMA para o tatamento de
  //eventos
```


INTERFACE GRÁFICA- EVENTOS

```
ActionListener handler = new ActionListener(){
    public void actionPerformed(ActionEvent e){
      ...
      if (obj == btnNew) //btnNew pode ser
      acessado na classe anônima!
        s = "Botão NEW pressionado. Por
        enquanto só faço isto!";
        ...
      }
    };
    btnNew.addActionListener(handler);
    ...
  }
}
```

INTERFACE GRÁFICA- EVENTOS

- EVENTOS DE MOUSE
  - Utilizam as seguintes interfaces: **`MouseListener`** e **`MouseMotionListener`**
  - MouseListener – Interage com os seguintes eventos:

EVENTO PRODUZIDO	MÉTODO PARA O TRATAMENTO DO EVENTO
Mouse é clicado (pressionado e liberado) sobre um componente	`void mouseClicked(MouseEvent e)`
Mouse entra na área de um componente	`void mouseEntered(MouseEvent e)`
Mouse sai da área de um componente	`void mouseExited(MouseEvent e)`
Mouse é pressionado sobre um componente	`void mousePressed(MouseEvent e)`
Mouse é liberado sobre um componente	`void mouseReleased(MouseEvent e)`

  - Para associar o tratamento de eventos de Mouse a um componente deve ser utilizado o método **`addMouseListener`**

INTERFACE GRÁFICA- EVENTOS

- EVENTOS DE MOUSE
  - Todos os métodos acima devem estar presentes mesmo que não estejam sendo utilizados neste caso o código será apenas - **{ }**
  - As informações que o método pode utilizar, como, por exemplo, a posição do mouse, estão contidas no objeto - **`MouseEvent e`**
  - Se o mouse é pressionado sobre um botão os seguintes eventos são disparados: MOUSE_PRESSED; MOUSE_RELEASED;MOUSE_CLICKED

INTERFACE GRÁFICA- EVENTOS

EVENTOS DE MOUSE

- MouseMotionListener – Interage com os seguintes eventos:

EVENTO PRODUZIDO	MÉTODO PARA O TRATAMENTO DO EVENTO
Mouse é clicado sobre um componente e então arrastado	`void mouseDragged(MouseEvent e)`
Mouse foi movimentado sobre um componente mas nenhum botão é clicado	`void mouseMoved(MouseEvent e)`

- Para associar o tratamento de eventos de movimento do Mouse a um componente deve ser utilizado o método **`addMouseMotionListener`**
- As informações que o método pode utilizar, como, por exemplo, a posição do mouse, estão contidas no objeto - **`MouseEvent e`**

INTERFACE GRÁFICA- EVENTOS

- EVENTOS DE MOUSE – Exemplo

```
public class JButtonEvents extends JFrame{
  private JLabel lblMouseStatus,
   lblMouseMsg;
  public JButtonEvents(){
    EventHandler handler = new
   EventHandler();
    this.addMouseListener(handler);
    ...
  }
```

INTERFACE GRÁFICA- EVENTOS

```
class EventHandler implements
 MouseListener, ...{
   public void mouseClicked(MouseEvent e){
     String s = "Mouse foi clicado no ponto -
      (" + e.getX() + " , " + e.getY() + ")";
     lblMouseMsg.setText(s);
     System.out.println(s);
   }
   public void mouseEntered(MouseEvent e){}
   public void mouseExited(MouseEvent e){}
   public void mousePressed(MouseEvent e){...}
   public void mouseReleased(MouseEvent e){...}
      ...
   }
}
```


INTERFACE GRÁFICA- EVENTOS

- EVENTOS DE JANELA (WINDOW)
  - WindowListener – Interage com os seguintes eventos:

EVENTO PRODUZIDO	MÉTODO PARA O TRATAMENTO DO EVENTO
Janela está ativa, ou seja, o cursor do teclado está posicionado sobre a mesma	`void windowActivated(WindowEvent e)`
Janela foi completamente fechada. Disparado depois que a janela foi completamente destruida	`void windowClosed(WindowEvent e)`
Janela está prestes a ser fechada. O primeiro evento a ser disparado antes de fechar a janela	`void windowClosing(WindowEvent e)`
Janela está desativada. Cursor do teclado está posicionado sobre outra janela do sistema	`void windowDeactivated(WindowEvent e)`
Janela minimizada,volta ao seu tamanho original	`void windowDeiconified(WindowEvent e)`
Disparado quando a Janela é minimizada	`void windowIconified(WindowEvent e)`
Disparado a primeira vez que uma janela se torna visível	`void windowOpened(WindowEvent e)`

INTERFACE GRÁFICA- EVENTOS

- EVENTOS DE JANELA (WINDOW)
  - Para associar o tratamento de eventos é necessário utilizar o método **`addWindowListener`**
  - Todos os métodos acima devem estar presentes mesmo que não estejam sendo utilizados neste caso o código será apenas - { }
  - As informações que o método pode utilizar, como o estado da janela, por exemplo, estão contidas no objeto - **`WindowEvent e`**

INTERFACE GRÁFICA- EVENTOS

EVENTOS DE JANELA (WINDOW) – Exemplo

```
public class JButtonEvents extends JFrame{
  public JButtonEvents(){
    //Será associado à janela um ouvinte de
   eventos de Janelas
    this.addWindowListener(handler);
    ...
  }
  class EventHandler implements WindowListener,
   ...{
    ...
    public void windowClosed(WindowEvent e){}
    public void windowActivated(WindowEvent e){
      System.out.println("Janela ativada!);}
    //continua...
```

INTERFACE GRÁFICA- EVENTOS

EVENTOS DE JANELA (WINDOW) – Exemplo

```
public void windowDeactivated(WindowEvent e){
  System.out.println("Janela desativada!);}
  public void windowDeiconified(WindowEvent e){
  System.out.println("O Tamanho Original
   restaurado!"; }
  public void windowIconified(WindowEvent e){
   System.out.println("Janela foi
   minimizada!");}
   public void windowOpened(WindowEvent e){
   //JOptionPane.showMessageDialog(null,"A
    janela foi aberta!");}
  }
  ...
}
```


INTERFACE GRÁFICA- EVENTOS

- CLASSES ADAPTADORAS
  - Todos os métodos definidos nas interfaces para tratamento de eventos devem ser codificados, pois interface possui somente a assinatura do método.
  - Método não utilizado necessita pelo menos da instrução **{ }**
  - Uma forma de resolver este problema é a utilização de classes Adaptadoras. São classes são abstratas que possuem todos os métodos declarados um um código do tipo **{ }**

INTERFACE GRÁFICA- EVENTOS

- CLASSES ADAPTADORAS
  - Para utilizar o tratamento de eventos a partir de classes adaptadoras basta derivar a classe adaptadora correspondente e então redefinir apenas os métodos necessários. Esta classe pode tratar somente um tipo de evento. Exemplo:

```
class EventHandler extends MouseAdapter
```

INTERFACE CLASSE PARA TRATAMENTO DE EVENTOS	CLASSE ADAPTADORA
MouseListener	MouseAdapter
MouseMotionListener	MouseMotionAdapter
WindowListenter	WindowAdapter
FocusListener	FocusAdapter
KeyListener	KeyAdapter
ComponentListener	ComponentAdapter
ContainerListener	ContainerAdapter

INTERFACE GRÁFICA- JLABEL

- Consiste de um rótulo. Pode conter um texto e/ou uma imagem associada. Não pode ser selecionado. Utilizado apenas para exibir informações. O Label pode possui um dica (tooltip) associado ao mesmo. É possível associar um texto HTML a um label
- **Construtor**

`JLabel(Icon image)` – Cria um label apenas com um ícone

`JLabel(String text)` – Cria um label com um determinado texto

```
JLabel(String text, Icon icon,
int horizontalAlignment)
```

Cria um label, utilizando um texto; um imagem e um alinhamento horizontal (LEFT, CENTER, RIGHT)

INTERFACE GRÁFICA- JLABEL

- **Métodos principais**

`public void setText(String text)` – Altera o texto

`public String getText()` – Recupera o texto

- **Eventos principais**

`PropertyChangeListener; ComponentListener; FocusListener; KeyListener; MouseListener`

JLABEL - Exemplo

```
...
//Cria um ícone
Icon icnFace = new ImageIcon("C:\\FACE.gif");
//Cria um label e ajusta algumas propriedades
lblLabel1 = new JLabel("Proprietario");
lblLabel1.setToolTipText("Nome do Proprietário
   do veículo");
lblLabel2 = new JLabel("Proprietário com Ícone",
   icnFace, JLabel.CENTER);
//
lblLabel3 = new JLabel(icnFace);
lblLabel3.setToolTipText("Label sem o ícone");
lblLabel3.setText("Ajusta o texto");
lblLabel3.setHorizontalAlignment(JLabel.RIGHT);
...
```

INTERFACE GRÁFICA- JTextField

- Consiste de uma linha de texto que pode ser digitada e editada. Quando a tecla <ENTER> é pressionada um `actionEvent` é disparado. Outras classes: `JPasswordField`, utilizado para entrada de senhas e `JFormattedTextField`, que permite controlar o tipo de caracter será digitado (somente números) e o uso de máscaras.
- **Construtores principais**

`JTextField(int columns)` – Cria um campo com um número fixo de colunas

`JTextField(String text)` – Cria e inicializa com um texto

`JTextField(String text, int columns)` – Cria um campo, com inicializado com um texto e com um número fixo de colunas(utilizado para calcular o tamanho)

INTERFACE GRÁFICA- JTextField

- **Métodos principais**

`public void setText(String text)` – Altera o texto

`public String getText()` – Recupera o texto

`public void setFont(Font f)` – Ajusta o tipo de letra (fonte) no campo

`public void addActionListener(ActionListener l)`

- **Eventos principais**

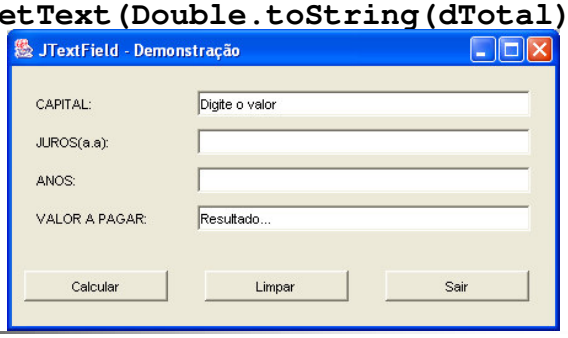
`PropertyChangeListener; ComponentListener; FocusListener; KeyListener; MouseListener; ActionListener`

JTextField - Exemplo

```
public class JTextFieldSample extends JFrame{
  public JTextFieldSample(){
    private JTextField txtCapital, txtJuros,
    txtPrazo, txtValor;
    ...
    txtCapital = new JTextField("Digite o
    valor",20);
    txtJuros = new JTextField(5);
    EventHandler handler = new EventHandler();
    txtPrazo.addActionListener(handler);
  }
  class EventHandler implements ActionListener{
    public void actionPerformed(ActionEvent e){
      ...
```


JTextField - Exemplo

```
    ...
   if (obj == txtPrazo) || (obj == btnCalcular)){
    double dTotal;
    String sCap, sJuros;
    dTotal = calculaTotal(sCap,sJuros,
    txtPrazo.getText());
    txtValor.setText(Double.toString(dTotal));}
    }
  }
  ...
}
```



INTERFACE GRÁFICA- JCheckBox

- Consiste de um componente que pode estar selecionado ou não. Caso esteja selecionado, mostra este estado através de uma marca. Em um conjunto destes componentes, mais de um pode estar selecionado. Para se utilizar este componente em um menu, deve ser utilizada a classe **`JToggleButton`**
- **Construtores principais**

`JCheckBox(String text)` – Cria um checkbox não selecionado e com um texto.

`JCheckBox(String text, boolean selected)` – Cria um checkbox que pode estar selecionado ou não

`JCheckBox(Icon icon, boolean selected)` – Cria um checkbox que possui um inicialmente somente um ícone e que pode estar selecionado ou não.

INTERFACE GRÁFICA- JCheckBox

`JCheckBox(String text, Icon icon, boolean selected)` – Cria um checkbox que possui um texto, um ícone e que pode ou não estar selecionado inicialmente

- **Métodos Principais**

`public void setSelected(boolean b)` – Altera o estado selecionado ou não sem no entanto disparar nenhum evento

`public boolean isSelected()` – Verifica se o checkbox está ou não selecionado

INTERFACE GRÁFICA- JCheckBox

- **Eventos Principais**

  `ActionListener;itemListener;`

  `ComponentListener; FocusListener`

- Neste tipo de componente é normalmente é utilizada a interface **`ItemListener`** para tratamento de eventos:

```
public void addItemListener(ItemListener l)
```

- Um evento ocorre quando o estado do checkbox é alterado e neste caso o seguinte método deve ser codificado:

```
void itemStateChanged(ItemEvent e)
```

- O método `int getStateChange()` da classe ItemEvent retorna se o checkBox está ou não selecionado (`ItemEvent.SELECTED`)

INTERFACE GRÁFICA- JRadioButton

- Consiste de um componente que pode estar selecionado ou não. Caso esteja selecionado, mostra este estado através de uma marca. Normalmente em um cojunto destes componentes, somente um pode estar selecionado ao mesmo tempo, sendo assim deve ser utilizado um objeto da classe **`ButtonGroup`** para criar o relacionamento entre estes botões
- **Construtores principais**

`JRadioButton(String text)` – Cria botão com título

`JRadioButton(String text, boolean selected)` – Botão pode estar selecionado ou não

`JRadioButton(Icon icon, boolean selected)` – Cria um botão que possui um inicialmente somente um ícone e que pode estar selecionado ou não

INTERFACE GRÁFICA- JRadioButton

`JRadioButton(String text, Icon icon, boolean selected)` – Cria um botão que possui um texto, um ícone e que pode ou não estar selecionado inicialmente

- **Métodos Principais**

`public void setSelected(boolean b)` – Altera o estado selecionado ou não sem no entanto disparar nenhum evento

`public boolean isSelected()` – Verifica se o botão está ou não selecionado

- **Eventos Principais**

`ActionListener;ComponentListener; FocusListener; ItemListener` (normalmente utilizado)

INTERFACE GRÁFICA - JComboBox

- Consiste de uma lista de itens onde é possível a seleção de um único item. Somente o item selecionado é visível. O ComboBox pode ser editável ou não.
- **Construtores principais**

`JComboBox()` – Cria um combo box, com uma lista vazia.

`JComboBox(Object[] list)` – Cria um ComboBox, onde a lista é inicializada comum array de objetos. Um exemplo seria um vetor do tipo String[].

`JComboBox(Vector list)` – Cria um ComboBox, onde a lista é inicializada com um vetor. Neste caso é utilizada a classe Vector que representa um vetor de tamanho variável e que pode conter qualquer tipo de objeto, sendo pois de grande flexibilidade

INTERFACE GRÁFICA - JComboBox

- **Métodos principais**

`public void setSelectedIndex(int anIndex)` – Ajusta qual elemento da lista está selecionado. O primeiro elemento possui o índice 0 e um índice igual a –1, indica que nenhum elemento está selecionado.

`public void setEditable(boolean aFlag)` – Permite que o Combo Box seja editável, ou seja, é possível acrescentar itens à lista.

`public void addItem(Object anObject)` – Adiciona um novo item à lista

`public Object getSelectedItem()` – Retorna o objeto atualmente selecionado

INTERFACE GRÁFICA - JComboBox

- **Eventos principais**

`ActionListener; ItemListener`

`ActionListener` – Disparado quando é feita um interação com a lista de itens ou então quando <ENTER> é pressionado em um combo editável.

`public void addActionListener(ActionListener l)`

INTERFACE GRÁFICA - JComboBox

`ItemListener` – Disparado quando o item selecionado é alterado. Quando um elemento já selecionado é novamente selecionado, nenhum evento é disparado. Quando a seleção sai de um item e vai para outro, dois eventos são disparados.

```
public void addItemListener(ItemListener
aListener)
```

JComboBox – Exemplo

```
public class JComboBoxSample extends JFrame{
  protected JComboBox cmbMetodo;
  protected String[] aMetodos = {"Juros
    Simpes", "Juros Compostos"};
  protected String sMetodoCalculo;
  ...
  //Cria o combo Box e inicializa a lista com
    os elementos do array
  cmbMetodo = new JComboBox(aMetodos);
  //Tratamento eventos
  EventHandler handler = new EventHandler();
  cmbMetodo.addActionListener(handler);
  cmbMetodo.addItemListener(handler);
  class EventHandler implements
    ActionListener, ItemListener{
```

JComboBox – Exemplo

```
public void actionPerformed(ActionEvent e){
      ...
      if ((obj ==
    txtPrazo)||(obj==btnCalcular)||(obj ==
    cmbMetodo))
      setTxtValor(dTotal);
     }
     public void itemStateChanged(ItemEvent e){
       if (e.getSource() == cmbMetodo){
         sMetodoCalculo =
    (String)cmbMetodo.getSelectedItem();
     }
  }
}
```


INTERFACE GRÁFICA - JList

- Consiste de uma lista de itens onde é possível a seleção de um ou mais itens. Caso a lista tenha vários itens barras de rolagens devem ser acrescentadas ao componente.
- **Construtores principais**

`JList()` – Cria um lista de seleção vazia.

`JList(ListModel dataModel)` – Cria uma lista onde os elementos estão contidos em um objeto que implementa a interface **`ListModel`**. Um exemplo é a classe **`DefaultListModel`**. Este construtor deve ser utilizado quando se deseja adicionar ou remover itens da lista um vetor de objetos. Um exemplo seria um vetor do tipo String[].

INTERFACE GRÁFICA - JList

`JComboBox(Vector list)` – Cria um ComboBox, onde a lista é inicializada com um vetor. Neste caso é utilizada a classe Vector que representa um vetor de tamanho variável e que pode conter qualquer tipo de objeto, sendo pois de grande flexibilidade

- **Métodos principais**

`public int getSelectedIndex()` - Retorna o índice o primeiro elemento selecionado. Caso nenhum elemento esteja selecionado retorna –1.

`public int[] getSelectedIndices()` - Retorna um vetor com o número dos índices de todos os elementos selecionados

INTERFACE GRÁFICA - JList

`public void setSelectedIndex(int anIndex)` – Ajusta qual elemento da lista está selecionado. O primeiro elemento possui o índice 0 e um índice igual a –1, indica que nenhum elemento da lista está selecionado.

`public Object getSelectedValue()` - Retorna o objeto atualmente selecionado.

`public void setSelectionMode(int selectionMode)` - Ajusta o modo de seleção da lista. Este modo de seleção pode ser: SINGLE_SELECTION (apenas um item por vez; SINGLE_INTERVAL_SELECTION (Um intervalo com vários itens pode ser selecionado) MULTIPLE_INTERVAL_SELECTION (Vários intervalos com vários itens podem ser selecionados).

INTERFACE GRÁFICA - JList

- **Eventos Principais**

`ListSelectionListener` – Ao utilizar esta interface a lista será notificada cada vezv que ouver uma alteração nos itens selecionados na lista

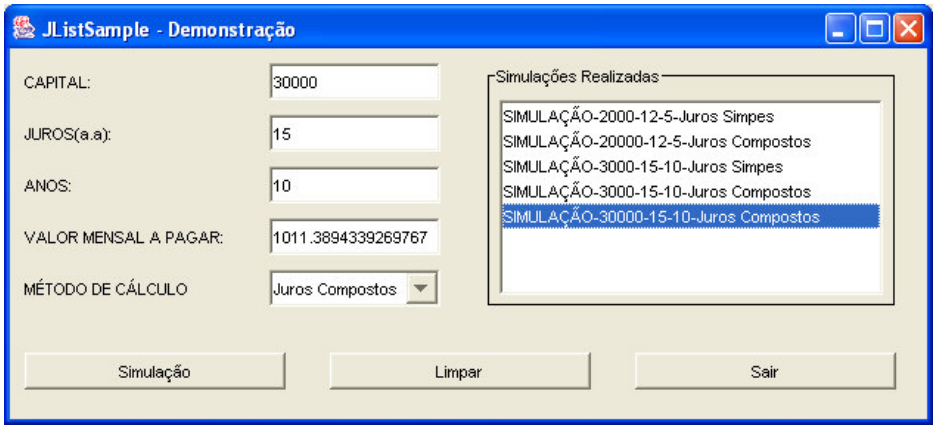
```
public void addListSelectionListener(
ListSelectionListener listener)
```

`MouseListener` – Permite que o componente seja notificado sempre que eventos de Mouse (clique; etc.) ocorrem sobre elementos da lista.

```
public void addMouseListener(MouseListener l)
```


INTERFACE GRÁFICA - JList

- JList - Exemplo



INTERFACE GRÁFICA - JTextArea

- Consiste de uma área que contém multiplas linhas de texto, onde é possível a edição deste texto utilizando mouse e teclado. No caso de várias linhas barras de rolagens devem ser acrescentadas.
- **Construtores principais**

`JTextArea()` – Cria uma área de texto

`JTextArea(String text, int rows, int columns)` – Cria uma área de texto que é inicializada com a string **Text** e que possui um número especificado de linhas (**rows**) e colunas (**columns**)

- **Métodos principais**

`public void setFont(Font f)` - Ajusta a fonte de texto que será utilizada para mostrar o texto.

INTERFACE GRÁFICA - JTextArea

`public void setEditable(boolean b)` - Permite que o texto seja editável.

`public void setText(String t)` - Ajusta o texto contido no componente.

`public String getSelectedText()` - Retorna o texto dentro do componente que está selecionado.

`public void setLineWrap(boolean wrap)` - Caso o valor de **wrap** seja **true** permite que o texto seja quebrado em várias linhas, quando o tamanho do mesmo for maior que a largura do componente.

- **Eventos**

MouseListener; KeyListener

`public void addMouseListener(MouseListener l)`

`public void addKeyListener(KeyListener l)`

INTERFACE GRÁFICA - JTextArea

- Para adicionar barras de rolagens (JSCrollPane) a um área de texto o seguinte código deve ser utilizado.

```
//Cria texta area com 10 linha e 15 colunas
txaEditor = new JTextArea(10,15);
//Cria as barras de rolagens. A classe
//JSCrollPane fornece uma vista do componente
//juntamente com as barras de rolagens
JScrollPane scrTextArea = new
   JScrollPane(txaEditor);
//A fim de exibir o componente é necessário
//adicionar o mesmo ao container, porém neste
//caso é adicionado a vista do componente com
//as barras de rolagens
getContentPane().add(scrTextArea);
```


JTextArea - Exemplo

```
public class JTextAreaSample extends JFrame{
  protected JTextArea txaObservacoes;
  ...
  public JTextAreaSample(){
    ...
    //Cria a área de texto
    txaObservacoes = new JTextArea(3,80);
    //Adiciona barras de rolagens à área de
   texto
    JScrollPane TxtAreaScrollPane = new
   JScrollPane(txaObservacoes);
    pnlObservacoes.add(TxtAreaScrollPane);
    ...
}
```

JTextArea - Exemplo

```
class EventHandler implements ActionListener{
  public void actionPerformed(ActionEvent e){
      ...
    if (obj == btnCalcular){
       String sObs;
       //Recupera o texto da área de texto
       sObs = txaObservacoes.getText();
       ...
    }
  }
}
}
```


INTERFACE GRÁFICA - Menus

Para se trabalhar com menus, várias classes devem ser utilizadas:

- **`JMenuBar`** – Barra de menus. Normalmente uma janela ou possui apenas um objeto deste tipo. Uma barra de menus possui vários objetos da classe Jmenu. Para adicionar uma barra de menu a uma janela deve ser utilizado o método: **`public void setJMenuBar(JMenuBar menubar)`**
- **`JMenu`** – Consiste de uma área que é mostrada assim logo que é clicada. Este objeto é o menu propriamente dito. Um menu contém vários objetos da classe **`JMenuItem`**. Caso um objeto seja classe seja adicionado a outro objeto da classe **`JMenu`** cria-se um sub-menu.
- **`JMenuItem`** – Este objeto representa uma opção do menu.

INTERFACE GRÁFICA - Menus

- **Construtores principais**

`JMenuBar()` – Cria uma barra de menu

`JMenu(String s)` – Cria um menu, cujo nome é dado pela string s

`JMenuItem(String text, int mnemonic)` – Cria um item de menu, cujo nome é dado pela string **s** e utiliza uma tecla de atalho indicada pelo inteiro **mnemonic**

- **Métodos principais**

`public JMenu add(JMenu c)` – Adiciona um menu a um objeto JMenuBar

`public JMenuItem add(JMenuItem menuItem)` – Adiciona um item de menu a um objeto da classe JMenu

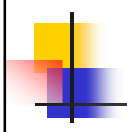
JAVA – INTERFACE GRÁFICA - Menus

- **Eventos**

Os menus trabalham com eventos da mesma forma que os Objetos da classe JButton (botões)

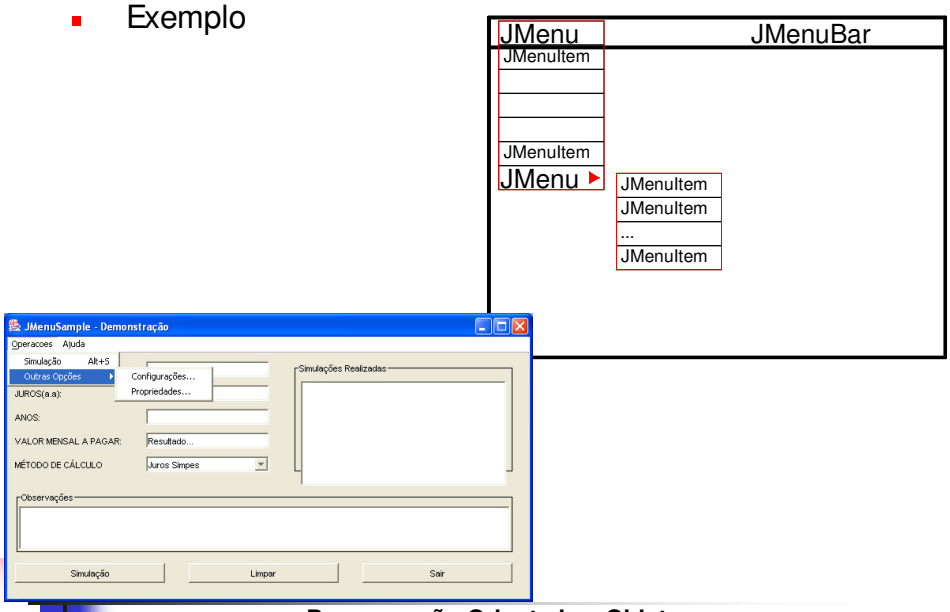
ActionListener – Ações e cliques em itens de menu. A cada item do menu (JMenuItem) deve ser associada uma ação diferente através do método:

```
public void addActionListener(ActionListener l)
```



INTERFACE GRÁFICA - Menus

- Exemplo



Menus - Exemplo

```
public class JMenuSample extends JFrame{
  JMenuBar menuBar;  JMenu menu, submenu;
  JMenuItem menuItem, menuItemSimulacao,
   mnuItmSobre;
 public JMenuSample(){
   menuBar = new JMenuBar();
   setJMenuBar(menuBar);
   menu = new JMenu("Operacoes");
   menu.setMnemonic(KeyEvent.VK_O);
   menuBar.add(menu);
   menuItemSimulacao = new
   JMenuItem("Simulação", KeyEvent.VK_S);
   menu.add(menuItemSimulacao);
   submenu = new JMenu("Outras Opções");
   menuItem = newJMenuItem("Configurações...");
```

Menus - Exemplo

```
menuItem = new JMenuItem("Propriedades...");
submenu.add(menuItem);
menu.add(submenu);
//Um segundo menu será criado
menu = new JMenu("Ajuda"); menuBar.add(menu);
menuItem = new JMenuItem("Ajuda do
  aplicativo"); menu.add(menuItem);
mnuItmSobre = new JMenuItem("Sobre...");
menu.add(mnuItmSobre);
//Eventos
menuItemSimulacao.addActionListener(handler);
mnuItmSobre.addActionListener(handler);
    ...
  }
}
```


CRIANDO UMA APPLET

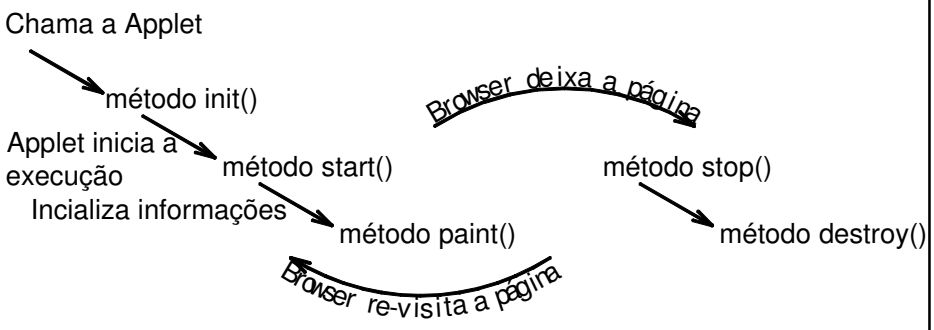
- Uma Applet é um aplicativo gráfico, criado na linguagem java, que pode ser executado em um navegador (Browser).
- Possibilita um aumento das capacidades da WWW uma vez que permite:
  - Animações
  - Imagens com som
  - Efeitos gráficos
  - Programas interativos, como jogos
  - Possibilitam a criação de conexões de rede com o host de origem

CRIANDO UMA APPLET

- `public void init()` - É o primeiro método executado e é executado apenas uma vez.
- `public void start()` - É executado toda vez que o applet aparece no browser.
- `public void paint()` - É executado toda vez que o applet aparece no browser. Recebe umainstância da classe Graphics. (Applet)
- `public void stop()` - É executado toda vez que o applet passa a não ser exibido pelo browser.
- `public void destroy()` - É executado quando o browser não precisa mais do applet.
- Os métodos acima podem ser especializados conforme a necessidade não sendo obrigatória sua presença.

CRIANDO UMA APPLET

- Sequência de métodos chamados pela Applet



CRIANDO UMA APPLET

- As applets podem ser criadas a partir da classe Applet (**`java.applet.Applet`**) ou então a partir da classe JApplet
- A classe JApplet (**`javax.swing.JApplet`**) é uma especialização da classe Applet e permite a utilização dos componentes SWING para a criação da interface com usuário.
- Todos os componentes e recursos vistos até aqui para a criação de interface de usuário podem ser utilizados para a criação de applets (JApplets), inclusive menus.

A CLASSE JApplet

- **Métodos principais**

`public Container getContentPane()` – Recupera o container principal da Applet

`public URL getCodeBase()` – Recupera a URL onde a applet está sendo executada, este método pode ser utilizado para carregar arquivos, de imagens, por exemplo.

```
Image image = getImage(getCodeBase(), "imgDir/a.gif");
```

`public void showStatus(String msg)` – Permite mostrar na barra de status do nagevador uma mensagem de texto.

`public AudioClip getAudioClip(URL url, String name)` – Retorna um objeto AudioClip(arquivo de som) que pode pode ser executado. O arquivo se encontra na URL especificada e possui o nome indicado pela String.

A CLASSE JApplet

`public Container getContentPane()` – Recupera o container principal da Applet

`public URL getCodeBase()` – Recupera a URL onde a applet está sendo executada, este método pode ser utilizado para carregar arquivos, de imagens, por exemplo.

```
Image image = getImage(getCodeBase(), "imgDir/a.gif");
```

`public void showStatus(String msg)` – Permite mostrar na barra de status do nagevador uma mensagem de texto.

`public AudioClip getAudioClip(URL url, String name)` – Retorna um objeto AudioClip(arquivo de som) que pode pode ser executado. O arquivo se encontra na URL especificada e possui o nome indicado pela String.

A CLASSE JApplet

`public String getParameter(String name)`– Retorna uma string que contém o valor de um parâmetro que foi passado para a applet através do código HTML. O nome do parâmetro deve informado para o método.

`public Image getImage(URL url, String name)`– Retorna um objeto do tipo Image que poderá ser visualizado na tela. O argumento URL equivale ao endereço do arquivo e o argumento String representa seu nome.

EXIBINDO UMA APPLET

- A applet é chamada a partir de um arquivo HTML. Para isto é utilizada a tag <applet>
- Exemplo:

```
<html>
<applet  code=AppletSample.class
         width=640
         height=240>
</applet>
</html>
```

- O utilitário AppletViewer(.exe), fornecido juntamente com J2SDK, permite a visualização de applets independente do navegador

EXIBINDO UMA APPLET

- É possível passar parâmetros para uma applet dentro de um arquivo HTML. Isto pode ser feito da seguinte forma:

```
//Código HTML que chama a applet
<APPLET>
<APPLET CODE= "AudioApplet.class" WIDTH=50 HEIGHT=50>
<PARAM NAME=Arquivo VALUE="AudioCom.au">
...
</APPLET>
//Código para recuperar os parâmetros
this.getParameter("Arquivo") retorna o valor "AudioCom.au“
```

O QUE UMA APPLET NÃO PODE FAZER

- Uma Applet não pode carregar bibliotecas ou definir métodos nativos (usam keyword `native`)
- Uma applet não pode ler ou escrever arquivos no cliente que a está executando
- Não pode iniciar outros programas
- Não tem acesso a certas propriedades do sistema
- As restrições de segurança acima não se aplicam caso a applet seja carregada a partir do sistema de arquivos local em um diretório que esteja presente na variável CLASSPATH

CRIANDO UMA APPLET – Código básico

```
import java.applet.Applet;
import java.awt.Graphics;
public class AppletApp extends Applet{
  public void paint (Graphics g){
    g.drawString("AppletApp - derived from
    Applet class",25,25);
  }
}
//Utilizando a classe JApplet (Swing) como base
import javax.swing.JApplet;
import java.awt.Graphics;
public class JAppletApp extends JApplet{
  public void paint (Graphics g){
    g.drawString("JAppletApp - derived from
    JApplet class",25,25);
  }
}
```

CRIANDO UMA APPLET – Exemplo 2

```
import javax.swing.*;
import java.awt.*;
import java.net.*;
public class AppletVeiculo extends JApplet {
  private int iPasso;
  private final String sPasso= " Passo ";
  private final String sMsg = "Executando
  método ";
  private JPanel pnlBackGround;
  private Container c;
  String sMessage;
  //continua...
```


CRIANDO UMA APPLET – Exemplo 2

```
  public void init(){
    iPasso = 1;
    sMessage=sMsg + " INIT: " +sPasso +iPasso;
    this.showStatus(sMessage);
    System.out.println(sMessage);
    iPasso++;
    //Cria o Painel
    pnlBackGround = new JPanel();
    //Recupera o container
    Container c = getContentPane();
  }
  //continua...
```

CRIANDO UMA APPLET – Exemplo 2

```
public void start(){
   sMessage = sMsg +" START: "+sPasso+iPasso;
   this.showStatus(sMessage );
   System.out.println(sMessage);
   URL urlCodeBase;
   urlCodeBase = this.getCodeBase();
   sMessage = "Codebase: " +
   urlCodeBase.toString();
   this.showStatus(sMessage);
   System.out.println(sMessage);
   iPasso++;
   //Adiciona um painel vermelho ao container
   Container c = getContentPane();
   //continua...
```


CRIANDO UMA APPLET – Exemplo 2

```
   c.setLayout(new BorderLayout());
   //Ajusta a cor do Painel
   pnlBackGround.setBackground(Color.RED);
   //adiciona painel ao container
   c.add(pnlBackGround,BorderLayout.CENTER);
}
public void paint(){
   sMessage = sMsg+" PAINT: "+sPasso + iPasso;
   this.showStatus(sMessage);
   System.out.println(sMessage);
   iPasso++;
}
//continua...
```

CRIANDO UMA APPLET – Exemplo 2

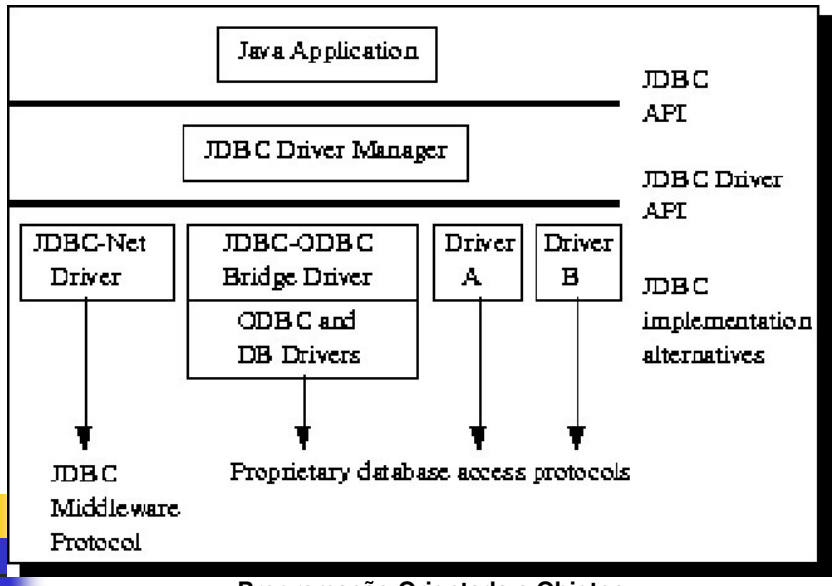
```
public void stop(){
  sMessage = sMsg +" STOP: "+ sPasso +iPasso;
  this.showStatus(sMessage);
  System.out.println(sMessage);
  iPasso++;
}
public void destroy(){
  sMessage = sMsg+" DESTROY: "+sPasso+iPasso;
  this.showStatus(sMessage );
  System.out.println(sMessage );
  iPasso++;
}
}
```


JAVA – JDBC

- JDBC – JAVA DATABASE CONNECTIVITY
  - Permite o acesso a banco de dados
  - Uma das formas de acesso é utilizando o driver JDBC-ODBC que permite a conexão através de um DRIVER OBDC
  - O **ODBC** (Open Database Connectivity) é um padrão para acesso aos banco de dados mais utilizados no mercado: SQLSERVER; ORACLE; MYSQL; POSTGRES; MS ACCESS;
  - COMO FUNCIONA
  - TIPOS DE DRIVERS
  - COMO UTILIZAR

JAVA – JDBC

- COMO FUNCIONA



JAVA – JDBC

- Os drivers baseados na tecnologia JBDC são divididos em quatro tipos ou categorias.
- Os drivers do tipo 1, podem ser utilizados sempre que não houver um driver específico para um determinado banco de dados

1. *JDBC-ODBC + ODBC driver:* Java acessa o banco através de drivers ODBC. O driver deve ser carregado em cada cliente que realiza acesso ao banco.
2. *Driver Java com API-Nativa:* Neste caso as chamadas JDBC são convertidas diretamente em chamadas para a API dos banco de dados. Neste caso também é necessário que um código binário específico esteja presente no cliente

JDBC – TIPOS DE DRIVERS

3. *Driver Java Puro, JDBC-Net :* Este driver traduz chamadas JDBC em chamadas para um protocolo de Rede/DBMS independente que em seguida é traduzido para o DBMS por um servidor. Este Middleware permite que cliente java "puros" se conectem com diferentes BD
4. *Protocolo Nativo – Driver Java Puro:* Neste caso as chamadas JDBC são convertidas diretamente para o protocolo utilizado pelo DBMS, permitindo uma chamada direta do cliente para o servidor. A maioria destes drivers é proprietário.

JDBC – COMO UTILIZAR

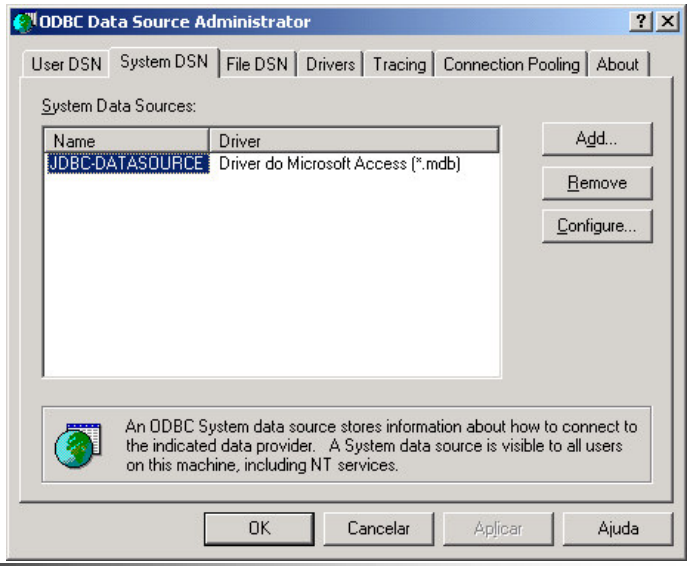
- OS seguintes passos são necessários para utilizar a tecnologia JDBC-ODBC
  1. CRIANDO A CONEXÃO ODBC COMO BD
  2. **CARREGAR O DRIVER**
  3. **CRIAR A CONEXÃO**
  4. **CRIAR COMANDOS SQL (STATEMENTS)**
  5. **PROCESSAR COMANDOS**
  6. **FINALIZAR A CONEXÃO COM O BANCO DE DADOS**
- Os passos de 2 a 6, são executados diretamente no código java.

JDBC – CRIANDO A CONEXÃO COM O BD

- Para utilizar os drivers do tipo 1, inicialmente é necessário carregar o driver na máquina do cliente, onde o aplicativo java será utilizado.
- Existem scripts para a instalação de drivers ODBC. No windows a instalação de um driver basicamente necessita de modificações no registro e a utilização de Dlls específicas para cada banco de dados.
- No windows através do painel de controle – "Fontes de dados OBDC" é possível criar novas conexões para os mais diversos banco de dados.

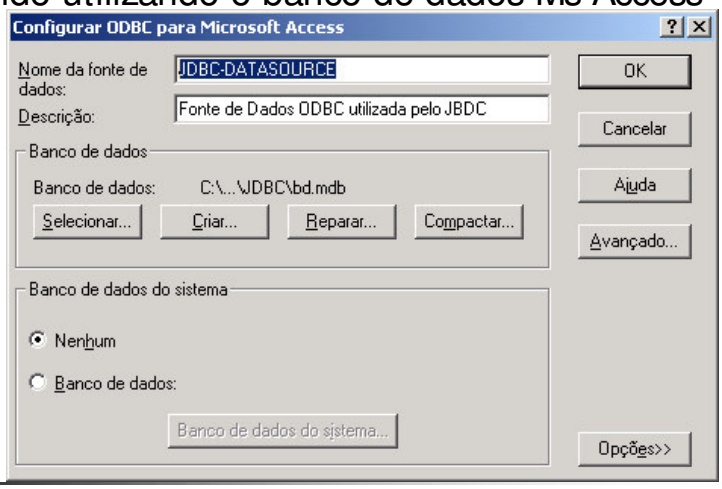
JDBC – CRIANDO A CONEXÃO COM O BD

- Abaixo é mostrado como criar uma fonte de dados ODBC



JDBC – CRIANDO A CONEXÃO COM O BD

- É necessário informar o nome da fonte de dados e a localização do banco de dados. Neste caso está sendo utilizando o banco de dados Ms Access



JDBC – CRIANDO A CONEXÃO COM O BD

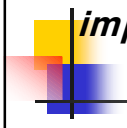
- Ao clicar no botão "Avançado..." (mostrado na tela anterior) é possível configurar o Nome de Logon e senha utilizada pelo banco de dados

JDBC – CARREGANDO O DRIVER

- O primeiro passo para trabalhar com o JBDC é carregar o driver
- Inicialmente o driver deve ser carregado. No caso do JDBC-ODBC isto é feito da seguinte forma:

  Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
- O código acima deve estar dentro de um bloco try/catch.
- A chamada acima cria uma instância do driver e registra o mesmo juntamente como DriverManager
- Para a utilização do JBDC é necessário o pacote:

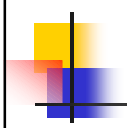
  import java.sql.*;



JDBC – CRIANDO A CONEXÃO

- Em seguida deve ser criada a conexão com o banco de dados. Para isto é utilizado o método getConnection da classe DriverManager.

  connection = DriverManager.getConnection(sBdName, sUserName, sPassword);
- Quando o driver é carregado uma instância da classe DriverManager é criada.
- A conexão retornada pelo método já está aberta e pronta para ser utilizada.



JDBC – Exemplos: Passo 2 e Passo 3

```
...
final String sBdName = "jdbc:odbc:Jdbc-
   Datasource";
protected Connection connection;
final String sUserName = "anonymous";
final String sPassword = "guest";
//Cria a conexão com o banco de dados
try{
  Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
  connection = DriverManager.getConnection
  (sBdName, sUserName, sPassword);
}
catch (ClassNotFoundException clex){
  ...
}
```


JDBC – CRIANDO COMANDOS SQL

- Um comando SQL consiste em um objeto da classe Statement.
- O primero passo para executar um comando é criar um objeto da classe Statement. Isto deve ser feito a partir da conexão criada anteriormente:

  statement = connection.createStatement();
- Para executar um SELECT deve ser utilizado o método executeQuery(), conforme mostrado abaixo:

```
public ResultSet executeQuery(String sql)
throws SQLException
```

- Para realizar uma modificação (INSERT; DELETE; UPDATE) no banco deve ser utilizado o método

```
public int executeUpdate(String sql)
throws SQLException
```

JDBC – Exemplo: Passo 4

```
...
//Cria um comando SQL
Statement statement;
//Cria um ResultSet
ResultSet resultset;
try{
 statement = connection.createStatement();
 resultset = statement.executeQuery(sSqlCommand);
 //displayResultSet(resultset);
 statement.close();
}
catch (SQLException sqlex){
...
}
```


JDBC – PROCESSANDO COMANDOS

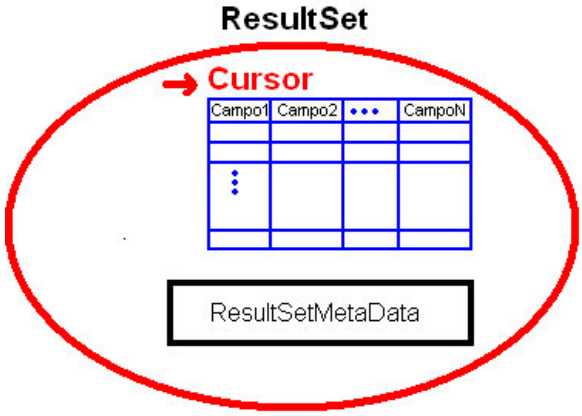
- Comandos o tipo "SELECT" retornam um objeto da classe ResultSet.
- Este objeto representa uma tabela e mantém um cursor apontando para uma linha de dados, além de informações sobre os campos da tabela.
- Inicialmente o cursor está posicionado antes da primeira linha. Para obter o primeiro registro é necessário utilizar o método –

  ```
  public boolean next() throws SQLException
  ```

- Para obter informações sobre a estrutura da tabela (nome; tipo e número de campos) deve ser utilizado um objeto do tipo ResultSetMetaData.

JDBC – PROCESSANDO COMANDOS

- A figura abaixo mostra um esquema do ResultSet
- Para obter todos os dados de um ResultSet é necessário percorrer todas as linhas, obtendo a informação de todas as colunas daquela linha



JDBC – PROCESSANDO COMANDOS

- Métodos para manipulação do cursor

  `public boolean next() throws SQLException` – Move próxima linha

  `public boolean previous() throws SQLException` – Move para a linha anterior

  `public boolean isLast() throws SQLException` – Verifica se a linha é a ultima

- Para obter o ResultSetMedaData o seguinte método do ResultSet deve ser utilizado:

```
public ResultSetMetaData getMetaData()
throws SQLException
```

JDBC – PROCESSANDO COMANDOS

- A partir do ResulSetMetaData é possível obter o número, o tipo e o nome das colunas da tabela:

  `public int getColumnCount() throws SQLException` – Recupera o número de colunas do ResultSet

  `public int getColumnType(int column) throws SQLException` – Recupera o tipo de dado contido na coluna.

  `public String getColumnName(int column) throws SQLException` – Recupera o nome da Coluna

- Os tipos das colunas são representados por um objeto da classe Types.

JDBC – Exemplo: Passo 5

```
...
//coloca o cursor no primeiro registro
boolean moreRecords;
//Como inicialmente o cursor esta antes da
//primeira linha, então deve ser movido,
//inicialmente para a primeira
moreRecords = rs.next();
if (!moreRecords){
  JOptionPane.showMessageDialog(null,"Fim dos
    registros");
  return;
}
//Vetor que irá conter campos(colunas)e as linhas
Vector colunas = new Vector();
Vector linhas = new Vector();
```

JDBC – Exemplo: Passo 5

```
//continua...
Vector linha;
try{
  //obtem o nome dos campos da tabela
  ResultSetMetaData rsmd = rs.getMetaData();
  for (int i = 1; i <= rsmd.getColumnCount();
    ++i){
     colunas.addElement(rsmd.getColumnName(i));
  }
  //obtem os dados de cada campo
  do {
     linha = getRowData(rs, rsmd);
     linhas.addElement(linha);
  } while (rs.next());
```


JDBC – Exemplo: Passo 5

```
  //continua...
  //Adiciona tabela ao JFrame
  table = new JTable(linhas,colunas);
  JScrollPane scrollTable = new
  JScrollPane(table);
  getContentPane().add(scrollTable,
       BorderLayout.CENTER);
  //Reposiona os componentes no container
  validate();
}
catch (SQLException sqlex){
    JOptionPane.showMessageDialog(null,
    sqlex.getMessage());
}
```

JDBC – Exemplo: Passo 5

```
public Vector getRowData(ResultSet rs,
   ResultSetMetaData rsmd) throws SQLException
{
  Vector linhaAtual = new Vector();
  for (int i = 1; i <= rsmd.getColumnCount();
   ++i){
    switch (rsmd.getColumnType(i)){
      case Types.VARCHAR:
        linhaAtual.addElement(rs.getString(i));
        break;
    case Types.INTEGER:
        linhaAtual.addElement(newLong(
        rs.getLong(i)));
        break;
    //continua...
```


JDBC – Exemplo: Passo 5

```
    case Types.DOUBLE:
      linhaAtual.addElement(newDouble(
      rs.getDouble()));
      break;
    default:
      JOptionPane.showMessageDialog(null,"Tipo
      não suportado"+ rsmd.getColumnTypeName(i));
      break;
    }
  }
  return linhaAtual;
}
```

JDBC – FINALIZANDO A CONEXÃO

- Após o processamento a conexão com o banco de dados, que foi criada inicialmente deve ser fechada.
- Para isto é utilizado o seguinte método:

```
public void close() throws SQLException
```