PROGRAMAÇÃO AVANÇADA EM JAVA

F. Mário Martins

2004/2005

JAVA Avançado 1

CLASSES JAVA DE TIPO PARTICULAR

- **Todas as classes que foram estudadas até agora, são classes de nível superior, ou seja, estão incluídas apenas em packages, existindo por si próprias e possuindo uma posição particular na hierarquia de classes.**
- **Estas classes são definidas em ficheiros de texto específicos que devem possuir o mesmo nome da classe e, ao serem compiladas, é automaticamente gerado pelo compilador de JAVA um ficheiro de código com o mesmo nome, ainda que com outra extensão.**
- **Porém, a linguagem JAVA permite que certas classes possam ser definidas como classes auxiliares à definição de outras, sendo o seu código definido dentro do mesmo ficheiro de definição da classe de alto nível. Estas classes possuem, portanto, definições "aninhadas", ou seja, incluídas nas**

definições de outras classes de mais alto nível. Designam-se em JAVA, por tal razão, inner classes.

- As inner classes constituem um mecanismo elegante e poderoso de JAVA, já que servem de classes auxiliares à definição das classes de alto nível sem que venham a pertencer à hierarquia final de JAVA, apesar de poderem ser igualmente utilizadas por outras classes para além da que as inclui.

- É relativamente comum no desenvolvimento de aplicações JAVA, em especial se forem de grande dimensão, utilizar inner classes. A sua importância e utilidade pode ser directamente observada ao analisar-se o código fonte de certas classes de JAVA. Principalmente por este motivo, melhor compreender certo código fonte de JAVA, e apesar da complexidade que deverá ser introduzida para que as mesmas sejam correctamente apresentadas neste momento, vamos estudar as inner classes, garantindo, assim, que todo o leque de diferentes possíveis utilizações de classes de JAVA fica completamente apresentado.

- **Assim, a abordagem das inner classes neste capítulo resulta de uma necessidade de completar a apresentação das classes de JAVA, e não tanto de uma necessidade de, neste momento, o leitor adquirir um conjunto de conhecimentos fundamentais para que possa prosseguir o seu estudo da PPO.**

A compreensão do papel destas classes é, apesar de tudo, importante sempre que pretendermos compreender o próprio código fonte de algumas classes de JAVA, por vezes construído à custa de tais classes auxiliares.

São, de facto, quatro os tipos diferentes de inner classes de JAVA: static member classes, member classes, local classes e anonymous classes.

Vejamos, em seguida, as suas principais características e formas de utilização.

CLASSES MEMBRO ESTÁTICAS

São classes (ou interfaces) que são definidas como membros de outras classes sob a forma de static classes, ou seja, usando adicionalmente o modificador `static` no usual cabeçalho da sua definição enquanto classe, sendo a sua definição aninhada no código da classe principal.

Uma classe deste tipo comporta-se como uma classe normal de alto nível, excepto no facto de que tem acesso a todos os membros static da classe que a inclui, ou seja, às variáveis e métodos de classe desta e, também, de outras classes static igualmente incluídas na principal. Para além destes, a classe static pode ainda aceder às variáveis de instância definidas como private na classe principal, ainda que não as possa referenciar usando this. O inverso é também verdadeiro, isto é, a classe principal tem mesmo acesso a todos os membros da classe estática.

Vejamos um exemplo de utilização de classes membro estáticas. A classe JAVA predefinida `java.util.LinkedList`, que implementa uma lista de objectos sob a forma de uma lista duplamente ligada,

usa uma private static class de nome `Entry` para definir não apenas a estrutura de cada célula da lista ligada mas também o conjunto de métodos que irão permitir manipular tais células.

Para que se compreenda o exemplo, vejamos parte do código da classe principal, a classe `LinkedList`:

```
public class LinkedList extends AbstractSequentialList
                  implements List, Cloneable,
                         Serializable {

  // variáveis de instância
  private Entry header = new Entry(null, null, null);
  private int size = 0;

  // construtores
  public LinkedList() {
    header.next = header.previous = header;
  }
```

```
public LinkedList(Collection c) {
   this();
   this.addAll(c);
}
   ......................................................................
```

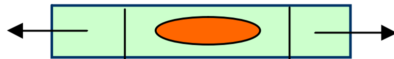
A classe define a variável `header` **como sendo do tipo** `Entry`**, ou seja, vai ser uma instância da classe auxiliar** `Entry`**.**

O construtor `LinkedList()` **atribui os valores iniciais aos campos de header, compreendendo-se, analisando tal código, que cada célula da lista ligada terá dois campos de referência para células,** `next` **e** `previous`**, ambos inicializados referenciando a célula** `header`**, valor inicial que é o correspondente à lista vazia.**

A classe `Entry`**, definida no mesmo ficheiro, possui a definição apresentada, parcialmente, em seguida, sendo de notar que possui como atributos de acesso** `private static`**:**

```java
private static class Entry {

 // variáveis de instância
```



```java
Object element;
Entry next;
Entry previous;

  // construtor

  Entry(Object elem, Entry next, Entry prev) {
     this.element = elem;
     this.next = next; this.previous = prev;
  }

  // método de instância exemplo

  private void remove(Entry e) {
     if (e == header) throw new RemoveException();
        e.previous.next = e.next;
        e.next.previous = e.previous;
        size--;
  }
```

```
    // outros métodos de instância - sem corpo

    private Entry addBefore(Object o, Entry e) {___}
    private Object clone() {___}
    private Object[] toArray() {___}
    ......................................................
}
```

Como se pode compreender pelo código, esta classe `static` é uma classe auxiliar da classe `LinkedList`, que define a própria estrutura de cada uma das células da lista, e cujos métodos irão permitir manipular as células da lista uma a uma, por exemplo, realizando inserções, remoções, cópias, etc.

Como se pode ver pelo código do método `remove(Entry e)`, os métodos desta classe têm acesso às variáveis de instância `private` da classe principal, ou seja, podem referenciar e modificar quer `header` quer `size`, ainda que através de uma referência directa ao seu nome e não pela utilização da referência `this.header`, por exemplo, dado que this apenas tem sentido no código dos métodos de instância da própria classe `LinkedList`.

Recorrendo aos métodos desta classe auxiliar, o código dos métodos de instância de `LinkedList`, tornam-se muito mais simples e claros, desde que compreendida a definição da classe auxiliar.

Por outro lado, a classe Entry, apesar de não fazer parte da hierarquia de classes, o que é, aliás, compreensível dada a sua especificidade, poderá ser importada por uma qualquer outra classe de JAVA, desde que referenciada pelo seu qualificador absoluto, neste caso, java.util.LinkedList.Entry. Caso pretendêssemos importar todas as inner classes de LinkedList, escreveríamos LinkedList.*.

A especificidade de Entry pode ser comprovada pelo facto de que, por exemplo, a classe TreeMap, de java.util, usa, igualmente, uma classe interna static de nome Entry, mas que neste caso implementa a estrutura e o comportamento de células necessárias à representação de nodos de árvores.

Note-se que a utilidade da classe auxiliar Entry relativamente ao ambiente JAVA e à classe LinkedList, é equivalente à utilidade

relativa da classe Produto que foi anteriormente definida para que mais facilmente pudéssemos definir a classe MaquinaVenda. Ou seja, Produto, sendo uma classe específica que não apresenta qualquer interesse particular para que seja acrescentada ao ambiente JAVA, poderia ter sido definida como uma inner class da classe MaquinaVenda.

Este tipo de raciocínio e de pragmatismo relativamente a uma dada classe, que tem que ser desenvolvida por forma a facilitar a construção de uma outra que será uma classe de alto nível, está na base deste mecanismo de inner classes de JAVA.

É também comum em JAVA que static member classes sejam interfaces, e que uma dada classe auxiliar da classe principal implemente tal interface. Vejamos um exemplo concreto.

Consideremos que pretendíamos criar uma fila de espera genérica sob a forma de uma lista ligada de células, classe que designaremos por FilaLigada.

Vejamos uma possibilidade de implementação usando uma static interface member:

```
import java.util.*;

// Fila de espera genérica em lista ligada

public class FilaLigada {

  // Interface estática que define como se devem percorrer
  // os elementos que estão ligados entre si.

  public static interface Ligavel {
     public Ligavel daSeguinte();
     public void fixaSeguinte(Ligavel celula);
  }
```

Note-se que a classe `FilaLigada` não declara implementar qualquer interface. Porém, a interface `Ligavel` é uma declaração interna à própria classe, pelo que os métodos de instância podem usar os dois métodos especificados, daSeguinte() e fixaSeguinte(). Terão mesmo que o fazer já que as variáveis de instância de FilaLigada, designadas frente e cauda, são do tipo Ligavel.

```
// variáveis de instância

private Ligavel frente; // início da fila (para remoção)
private Ligavel cauda;  // fim da fila (para inserção)
private int dim;        // comprimento actual

// construtor da classe principal. frente e cauda são do
// tipo Ligavel, ou seja, são instâncias de uma qualquer
// classe que implemente a interface Ligavel.

public FilaLigada() {
   frente = null;
   cauda  = null;
   dim = 0;
}
```

```
// métodos de instância de FilaLigada

public boolean vazia() {
     return frente == null;
}

public void remove() throws FilaVaziaException {
    if (this.vazia()) { throw new FilaVaziaException()
      else {
            frente = frente.daSeguinte(); dim--;
           }
}

public Ligavel daPrimeiro() {
   return frente;
}

public void insere(Ligavel c) {
    if (dim == 0) { frente = c; cauda = c;}
     else
        {c.fixaSeguinte(frente); frente = c;};
    dim++;
}

public int size1() {  // serve de teste de percurso da
```

```
    int t = 0;        // lista; usar antes size().
    Ligavel c = frente;
    while(!(c==null)) {
        t++; c = c.daSeguinte();
    }
    return t;
  }

  public size() { return dim; }

  public Vector lista() { // para enumeração externa
    Vector fila = new Vector();
    Ligavel c = frente;
    while(!(c==null)) {
        fila.addElement(c);
        c = c.daSeguinte();
    }
    return fila;
  }
} // fim da definição de FilaLigada


// Classe interna que implementa a interface Ligavel.
// Esta classe implementa células de tipo Ligavel que
```

```
// contêm uma String e a referência para a seguinte.

class LigavelString implements FilaLigada.Ligavel {

   // variáveis de instância
   private String nome;
   private FilaLigada.Ligavel seguinte;

   // construtor
   public LigavelString(String s) {
       this.nome = s; this.seguinte = null;
   }

   // métodos de instância

   public String daValor() {return this.nome;}
   public String toString() { return this.nome;}




   // métodos que implementam a interface Ligavel

   public FilaLigada.Ligavel daSeguinte() {
```

```
        return seguinte;
    }

    public void fixaSeguinte(FilaLigada.Ligavel ref) {
         this.seguinte = ref;
    }
}
```

Note-se que nenhum método da classe `FilaLigada` manipula dados contidos nas células, dado desconhecerem quais estes são. Apenas são usadas as referências para as células seguintes e, mesmo para estas, de forma abstracta, ou seja, usando os métodos especificados na interface `Ligavel`.

Deste modo a classe `FilaLigada` oferece, simultaneamente, um comportamento que é específico do funcionamento de uma fila de espera implementada usando uma lista ligada, um comportamento genérico a que cada implementação específica da fila de espera deverá obedecer implementando-o à sua maneira, definido através da interface `Ligavel`, e, ainda, a flexibilidade de cada classe poder estruturar as suas células ligadas da forma que mais lhe convier.

Assim, tal como se pode verificar através da classe exemplo `LigavelString`, qualquer classe que pretenda ver as suas células inseridas e manipuladas segundo uma `FilaLigada`, apenas definirá a estrutura de tais células e o código dos dois métodos que implementam `Ligavel`.

Note-se ainda que a classe LigavelString, embora pertença ao mesmo ficheiro fonte de FilaLigada, é uma classe externa a esta. Assim, e por não ser membro desta classe, tem que prefixar qualquer referência a membros desta, usando, por exemplo, FilaLigada.Ligavel, para se referir à interface (ou tipo) definido na classe que lhe é exterior.

A criação de uma FilaLigada de dado tipo, e a sua utilização, é exemplificada em seguida, usando pequenos extractos de código de um programa de teste:

```
import java.lang.*;
public class TstFilaLigada {
```

```
// teste de FilaLigada

 public static void main(String args[]) {

 // Criação de células do tipo LigavelString para inserir
 // numa FilaLigada.

    LigavelString c1 = new LigavelString("ISABEL");
    LigavelString c2 = new LigavelString("RITA");
    LigavelString c3 = new LigavelString("NUNO");
    LigavelString c4 = new LigavelString("PEDRO");

 // Criação de uma instância vazia de FilaLigada

    FilaLigada fila1 = new FilaLigada();

 // Inserção das células na FilaLigada
    fila1.insere(c1);
    fila1.insere(c2);
    fila1.insere(c3);


// Outras operações diversas

    System.out.println(fila1.size());
    fila1.remove();
```

Em conclusão, foi criada uma classe genérica, com comportamento genérico, que oferece uma determinada estruturação de objectos, mas que teve que impor às classes que pretendam ver as suas instâncias estruturadas dessa forma, a satisfação de certos requisitos, designadamente, a implementação de alguns métodos básicos, imposição realizada através da incorporação interna, ou seja, na sua definição, de uma interface.

A classe oferece um conjunto de serviços genéricos, mas tem que garantir que os clientes de tais serviços cumprem certos requisitos de comportamento. Assim, estes têm que satisfazer a interface ditada pela classe, enquanto de esta não. Esta é a única forma de tal contrato ser garantido, não o podendo nunca ser através do uso de interfaces definidas externamente à classe.

O compilador de JAVA ao compilar a classe FilaLigada, cria dois ficheiros, um de nome `FilaLigada.class` e outro de nome

`FilaLigada$Ligavel.class` que contém a implementação da interface estática contida na definição da classe.

Todas as questões relacionadas com acessos são resolvidas pelo compilador.

CLASSES MEMBRO NÃO ESTÁTICAS

São classes igualmente definidas como membros de outras classes mas que não são definidas usando o modificador static. Assim, este tipo de classe auxiliar pode, tal como uma variável de instância ou um método de instância, ser instanciada, e, deste modo, uma instância sua estará sempre associada a qualquer instância criada da classe principal.

Assim sendo, o código desta classe auxiliar pode aceder a todas as variáveis e métodos, estáticos e não estáticos, da classe que a inclui, e vice-versa.

Enquanto que static member classes de uma dada classe podem ser quer classes quer interfaces, apenas classes pode ser member classes, dado que as interfaces não permitem criar instâncias.

Procurando exemplificar a utilidade da utilização de member classes em JAVA, vamos analisar parte do código da classe Vector. Um dos métodos de instância de Vector é o método elements() que devolve uma Enumeration do vector, segundo o código:

```
public Enumeration elements() {
 new VectorEnumerator(this);
}
```

Da análise do código torna-se imediato constatar que é invocado um construtor de nome `VectorEnumerator`, ao qual é passada uma referência do receptor da mensagem `elements()` que é uma instância da classe Vector.

Procurada a classe `VectorEnumerator` nos vários packages de JAVA, tal classe não aparece.

Analisado mais em pormenor o código da classe Vector, verifica-se que a classe VectorEnumerator se encontra, de facto, definida no mesmo ficheiro fonte da classe Vector, e com atributos que não a tornam pública, sendo o seu código o seguinte:

```
final class VectorEnumerator implements Enumeration {
    Vector vector;
    int count;
    public VectorEnumerator(Vector v) {
    vector = v;
    count = 0;
   }

   public boolean hasMoreElements() {
    return count < vector.elementCount;
   }

   public Object nextElement() {
    synchronized (vector) {
      if (count < vector.elementCount) {
          return vector.elementData[count++];
      } }
    throw new NoSuchElementException("VectorEnumerator");
   }}
```

Trata-se, portanto, de uma inner class do tipo member class da classe Vector, que se encontra definida no ficheiro fonte Vector.java, e que serve de auxiliar para a implementação dos métodos necessários à transformação de um Vector numa Enumeration.

Note-se ainda que o construtor da classe auxiliar VectorEnumerator recebe como parâmetro o vector a enumerar, não tendo assim que referenciar a variável de instância da classe Vector. No entanto, caso tivesse que o fazer, JAVA define que sempre que uma classe auxiliar pretende referenciar uma variável de instância da classe que a inclui, deverá usar a sintaxe Idclasse.this.Idvar, tal como, por exemplo, LinkedList.this.header ou Vector.this.elementCount.

CLASSES LOCAIS

São classes muito particulares que são definíveis dentro de um determinado bloco de código de uma outra classe, tipicamente dentro do código de um método. São, naturalmente, apenas acessíveis dentro desse bloco, e apenas têm acesso às constantes e variáveis acessíveis em tal contexto. Em alguns casos, classes locais são definidas dentro dos próprios construtores de uma dada classe.

Considere-se que dentro da classe FilaLigada é necessário criar um método que construa uma enumeração sobre a fila, método designado enumera(). Uma outra possibilidade de o fazer, seria usar uma classe local a tal método, classe que define como tal enumeração vai ser realizada, usando a estrutura da classe principal.

Caso pretendêssemos criar uma enumeração sobre uma FilaLigada, usando uma classe local ao método enumera(), teríamos então que definir o seguinte código de tal método:

```
public Enumeration enumera() {

   classe EnumeradorDeFila implements Enumeration {
      Ligavel actual;
      public EnumeradorDeFila() { actual = frente; }
     public boolean hasMoreElements()
          return (dim > 0);
      }
      public Object nextElement() {
          if (actual == null)
               throw new NoSuchElementException();
          Object seguinte = actual;
          actual = actual.daSeguinte();
           return seguinte;
      }
   }

  return new EnumeradorDeFila(); // resultado do método
}
```

O método enumera() define, portanto, uma classe que lhe é local, classe esta que define a forma de realizar uma enumeração sobre a

instância de que ambos fazem parte, neste caso, uma instância de FilaLigada.

A maior desvantagem de uma classe local é que não é acessível do exterior, e, por isso, nem sequer faz sentido que lhe seja atribuído qualquer modificador de acesso.

CLASSES ANÓNIMAS

Classes anónimas são classes locais sem identificador definido. Possuem uma definição identica à de qualquer outra classe, mas, pelo facto de não possuirem um identificador, são usadas para a imediata criação de uma instância, ou seja, são em geral precedidas por new, ou seja, o resultado da introdução da sua definição será a imediata criação de uma instância. São, em geral, associadas a instruções de return de métodos, e possuem definições muito simples, sendo úteis sempre que há necessidade de realizar adaptações de resultados.

Obedecem a uma estrutura sintáctica simples tal como:

```
new nomeClasse([lista-argumentos]) { corpo }
```

não podendo definir construtores específicos, dado não possuirem nome. Podem, no entanto, definir métodos de instância particulares de inicialização das variáveis de instância, o que supera a restrição.

Classes Especiais de JAVA

Para além destas classes de tipo particular que têm por missão principal auxiliar na definição e construção de classes de topo, permitindo, como vimos, que muitas das classes de topo se nos apresentem como classes genéricas, facilmente reutilizáveis, implementando comportamentos complexos mas altamente parametrizados, outras classes JAVA existem, já predefinidas, que implementam comportamentos muito importantes, merecendo portanto uma referência particular.

Não se pretendendo, nem fazendo sentido neste momento, apresentar em detalhe cada uma de tais classes, vamos fazer-lhes referência dividindo-as por categorias funcionais, ou seja, por categorias de serviços prestados, que são, de certa forma, indexados aos packages de que tais classes fazem parte.

As `java.util.Collections` desenvolvidas para a versão 1.2 de JAVA, são um conjunto de classes que implementam as mais diversas formas de representação de dados como colecções, quer

listas quer conjuntos, bem como correspondências de 1 para 1 ou 1 para n, designadas por mappings.

Destacam-se, pela sua utilidade, as classes Hashtable, HashMap, Vector, ArrayList, TreeMap, TreeSet e LinkedList.

Todas as aplicações apresentadas correspondem a programas puramente sequenciais. JAVA permite, no entanto, programar com relativa facilidade código concorrente, quer usando threads quer usando processos. As threads são tarefas concorrentes, ou simplesmente paralelas, que são executadas sob o controlo de um interpretador de JAVA. Os processos são programas que podem mesmo executar externamente ao interpretador de JAVA, e com os quais um programa JAVA poderá comunicar usando streams. A classe `java.lang.Process` permite criar facilmente processos, sendo estes, no entanto, sempre dependentes da plataforma em que são criados e, portanto, geralmente não portáveis.

As classes do package `java.net` permitem implementar com alguma simplicidade aplicações sobre redes, suportando diversos

protocolos tais como http:, ftp: e file:. A classe URL é particularmente interessante dado suportar qualquer um dos protocolos referidos.

No entanto, caso se pretenda menos abstracação e mais controlo sobre as comunicações em rede, sockets podem ser facilmente usados na comunicação com um servidor.

Os packages `java.security` e `javax.cripto` definem um grande conjunto de classes que permitem implementar diversos mecanismos de segurança, quer sobre a execução de código JAVA quer relativos a transacções de diversos tipos de dados. Algumas destas classes implementam o modelo de execução protegida de JAVA, designado por modelo sand-box, em que código JAVA não garantidamente seguro é executado num ambiente especial, em geral sem acesso a dispositivos físicos, por forma a testar, em segurança, a correcção do mesmo.

Estes packages fornecem igualmente classes que implementam algoritmos especiais de encriptação e desencriptação de mensagens, de criação de assinaturas digitais e sua autenticação, e

que são, hoje em dia, muito importantes em várias áreas que se relacionam com as transacções digitais através da internet.

Applets são pequenas aplicações JAVA desenvolvidas para serem executadas por um qualquer browser ou applet viewer, numa qualquer plataforma, e de forma segura. Uma applet não é muito diferente de um normal programa JAVA, excepto pelos factos, relevantes, de que não pode ser executada de forma autónoma, ou seja, directamente pelo interpretador de JAVA, não tem acesso a operações de I/O e, em geral, é inserida noutra qualquer aplicação. Tipicamente, o seu código faz parte de páginas web, sendo traduzido pelo browser do utilizador.

Uma applet, donde quer que possa ter sido adquirida, terá sempre uma execução segura, pois não tem acesso ao sistema de ficheiros do utilizador.

A classe `Applet` implementa as operações de interface entre uma dada applet e o seu contexto, ou seja, a camada computacional que fornece os dados a serem apresentados.

Em geral, tal como noutras ferramentas de criação de ambientes interactivos, a camada computacional deverá ser criada como sendo uma subclasse de `Applet`, o que não é uma solução estruturalmente muito correcta e conveniente.

O package `java.awt` (Abstract Windowing Toolkit) inclui as classes responsáveis pela criação das interfaces gráficas com o utilizador (GUIs), incluindo classes que implementam componentes de computação gráfica, objectos de interacção tais como botões, menus, etc., e, ainda, de gestão de layout. Os seus componentes são, em geral, complexos e baseados em threads.

O modelo de interacção de JAVA, suportado por este conjunto de classes é, um modelo complexo e de baixo nível, sendo baseado em eventos. Em consequência, a criação de uma camada interactiva gráfica sobre uma qualquer camada funcional, é complexa, dado não seguir um modelo metodologicamente claro, por exemplo, tal como o modelo MVC (Model,View e Controller) de Smalltalk.

Talvez por esta razão, desde JAVA 1.2 que é fornecido o package `javax.Swing`, que vem, em muito, facilitar a criação de interfaces

gráficas com o utilizador, dado que segue o modelo MVC, herdado de Smalltalk, modelo segundo o qual a ligação da camada interactiva à camada computacional é realizada de forma estruturada, ainda que baseada na troca de eventos. Adicionalmente, aspectos de "look and feel" são já contemplados.

Finalmente, não se poderá deixar de referir o package `java.beans`, que inclui um conjunto de classes que permite implementar componentes genéricos (beans), em geral componentes awt (gráficos), que são caracterizados pelas propriedades, eventos e métodos que exportam, e que podem ser incorporados em aplicações, comunicando com estas através de eventos. Os beans elevam a programação por objectos para níveis de meta-programação e de meta-reutilização.

SÍNTESE e OUTROS EXEMPLOS:

INNER CLASSES

Classes que são definidas no interior da definição de outras classes e que, portanto, não ficam a fazer parte da hierarquia de classes de JAVA, mas que são muito úteis na definição de certas funcionalidades internas das classes a que pertencem.

Exº: Uma classe Stack implementada usando um VECTOR e que define um método `enumerator()` **que devolve uma** `Enumeration` **sobre os seus elementos.**

```
public class Stack {

   private Vector valores;

     // código de Stack
   ...................................
```

```
public Enumeration enumerator() {
    return new StackEnum();
  }

class StackEnum implements Enumeration {
     private int num = valores.size()-1;

  public boolean hasMoreElements() {
    return (num >= 0);
  }

  public Object nextElement() {
    if(!hasMoreElements())
      throw new NoSuchElementException();
    else
      return valores.elementAt(num--);
  }
}
}
```

ANONYMOUS CLASSES

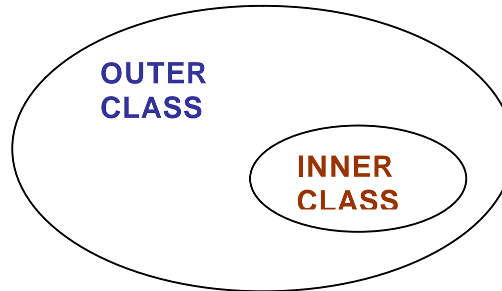
Classes que são definidas no interior da definição de outras mas às quais não é sequer atribuído um nome.

```
public Enumeration enumerator() {
     return new StackEnum()
      {
        int num = valores.size()-1;

         public boolean hasMoreElements() {
     return (num >= 0);
    }

    public Object nextElement() {
     if(!hasMoreElements())
       throw new NoSuchElementException();
     else
        return valores.elementAt(num--);
    }
      }
  }
```

CONTEXTO:




```
public class LinkedList extends AbstractSequentialList
                        implements List, Cloneable, java.io.Serializable {

   // variáveis de instância transientes

    private transient Entry header
          = new Entry(null, null, null);
    private transient int size = 0;
```

```
/** Constructs an empty list. */
  public LinkedList() {
    header.next = header.previous = header;
   }

/** Constructs a list containing the elements of the specified collection, in the order they are returned by the collection's iterator.
     */
   public LinkedList(Collection c) {
   this();
   this.addAll(c);
    }

// Definição do Elemento genérico da Lista Ligada

private static class Entry {

   Object element;
   Entry next;
   Entry previous;
```

```
    Entry(Object element, Entry next, Entry previous) {
        this.element = element;
        this.next = next;
        this.previous = previous;
    }
}
```

// Alguns Métodos de Instância de LINKED LIST

```
     private Entry addBefore(Object o, Entry e) {
    Entry newEntry = new Entry(o, e, e.previous);
    newEntry.previous.next = newEntry;
    newEntry.next.previous = newEntry;
    size++; modCount++;
    return newEntry;
    }

     private void remove(Entry e) {
    if (e == header) throw new NoSuchElementException();
    e.previous.next = e.next; e.next.previous = e.previous;
    size--; modCount++;
    }
```

```
private class ListItr implements ListIterator {

    private Entry lastReturned = header;
    private Entry next;
    private int nextIndex;
    private int expectedModCount = modCount;

    ListItr(int index) {
        if (index < 0 || index > size)
         throw new IndexOutOfBoundsException();
        if (index < (size >> 1)) {  // size/2
         next = header.next;
         for (nextIndex=0; nextIndex<index;nextIndex++)
             next = next.next;
        }
       else {
         next = header;
         for (nextIndex=size; nextIndex>index;
                                                   nextIndex--)
             next = next.previous;
        }
    }
```

```
    public boolean hasNext() {
        return nextIndex != size;
    }

    public Object next() {
        checkForComodification();
        if (nextIndex == size)
         throw new NoSuchElementException();
        lastReturned = next;
        next = next.next;
        nextIndex++;
        return lastReturned.element;
    }
}
```

EM RESUMO: Assim se implementa Parametrização e Programação Genérica em JAVA.

JAVA Avançado 2

JDBC (“Java DataBase Connectivity”) API

The Java Database Connectivity (JDBC)

The Java Database Connectivity (JDBC) API is the industry standard for database-independent connectivity between the Java programming language and a wide range of databases – SQL databases and other tabular data sources, such as spreadsheets or flat files. The JDBC API provides a call-level API for SQL-based database access.

JDBC technology allows you to use the Java programming language to exploit "Write Once, Run Anywhere" capabilities for applications that require access to enterprise data. With a JDBC technology-enabled driver, you can connect all corporate data even in a heterogeneous environment.

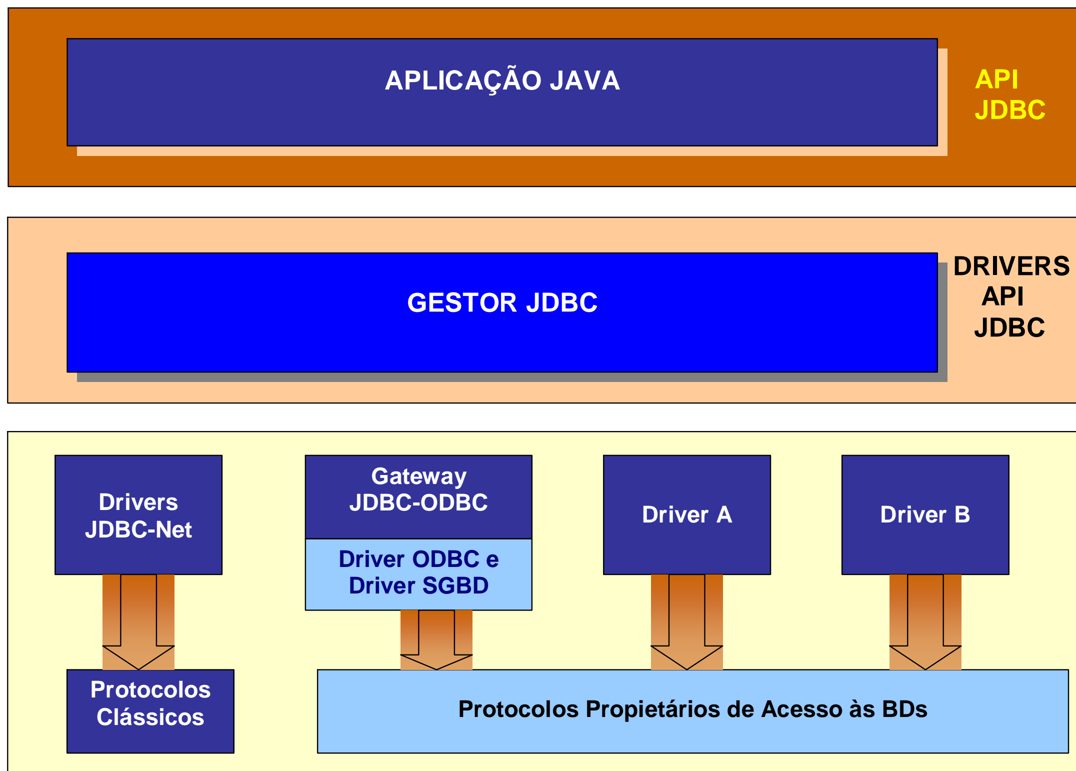
A arquitectura JDBC possui dois conjuntos principais de interfaces:

- JDBC API para quem desenvolve aplicações cuja camada de dados necessita de aceder a bases de dados;
- JDBC driver API para quem necessita de desenvolver “drivers” para as suas bases de dados poderem ser acedidas a partir de Java.

A JDBC API vai permitir:

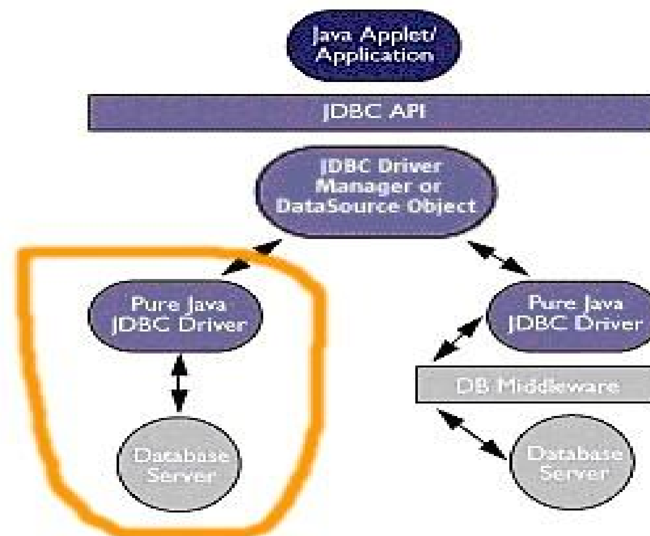
- Estabelecer ligação com uma base de dados ou aceder a tabelas;
- Enviar instruções SQL;
- Receber e processar o conjunto de resultados.

ARQUITECTURA GERAL DE ACESSO A DADOS COM JDBC

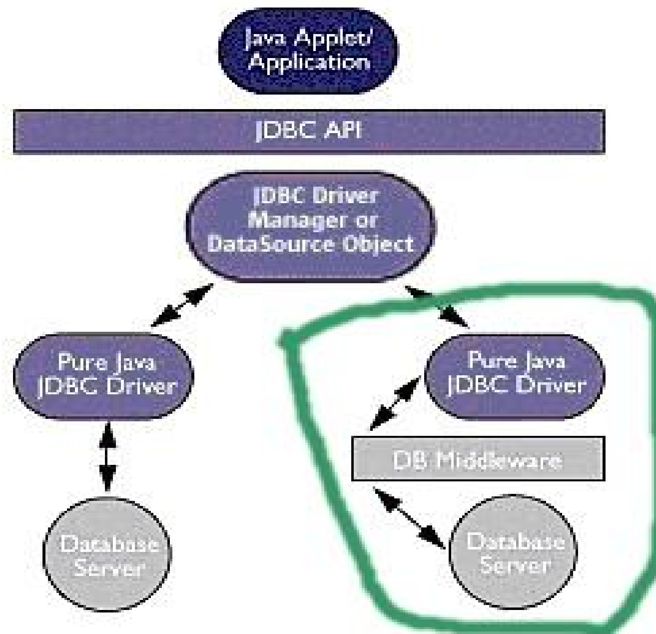


JDBC "drivers" podem ser classificados em 4 categorias, tal como ilustrado nas figuras seguintes:

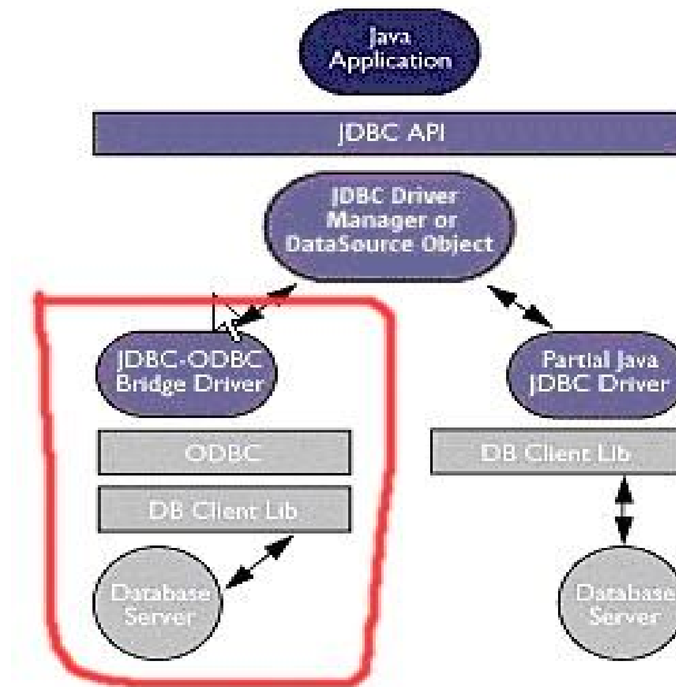
1) Applets e e aplicações podem aceder a bases de dados usando a JDBC API através de "drivers" puros de tecnologia JDBC. Estes "drivers" traduzem as chamadas JDBC para o protocolo de rede usado pelos DBMS, o que corresponde a uma chamada directa da máquina cliente ao servidor de bases de dados ("intranet access").



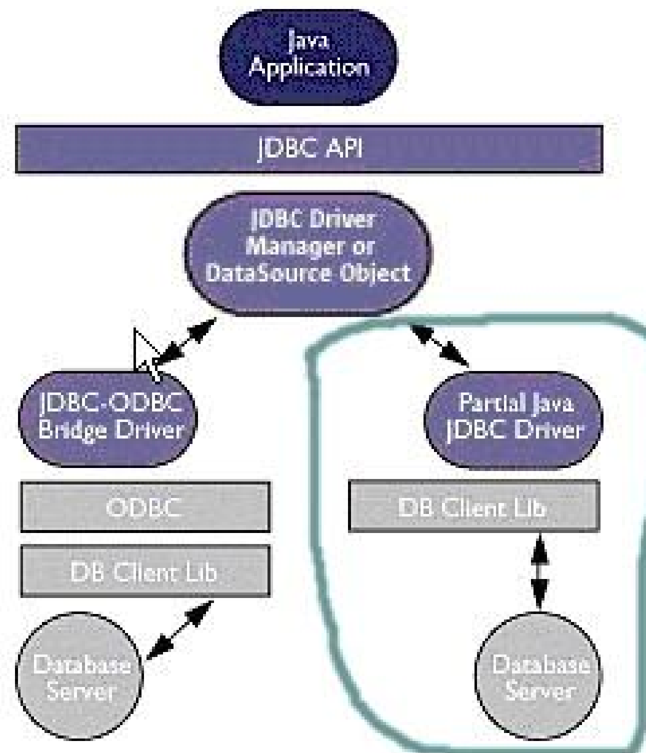
2) “Drivers” puros de tecnologia JDBC ligam a “middleware” específico, traduzindo “chamadas” JDBC para os protocolos intermédios, que por sua vez são traduzidos para os protocolos usados pelos DBMS. O “middleware” fornece ligação a múltiplas distintas bases de dados.



3) O acesso à bases de dados faz-se através de "drivers" ODBC cujo código é carregado em cada máquina cliente que usa JDBC-ODBC. A Sun fornece uma "bridge" JDBC-ODBC que pode ser usada em casos experimentais quando nenhum "driver" está disponível.

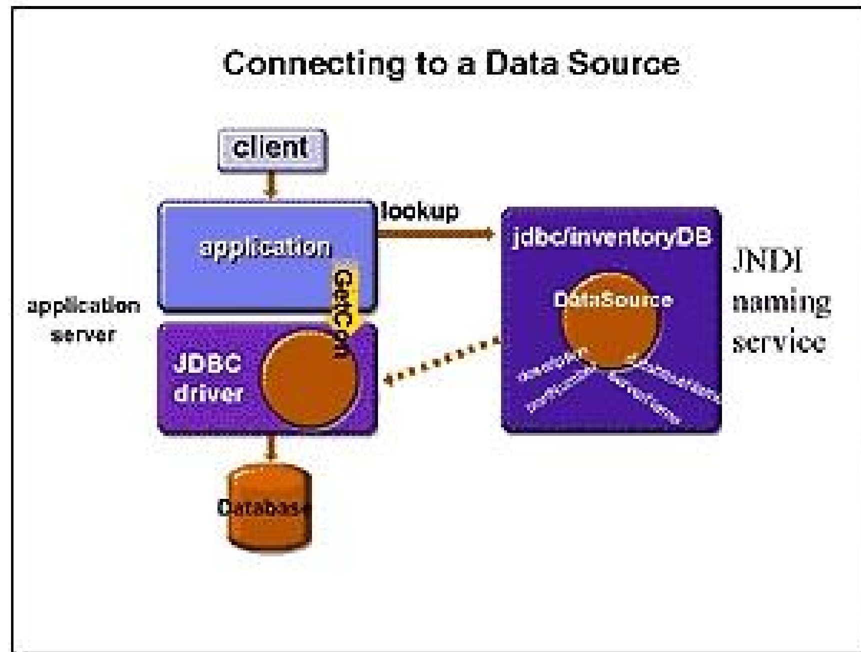


4) “Driver” para API nativa, que converte chamadas JDBC em chamadas da API de Oracle, Sybase, DB2, Informix, etc.



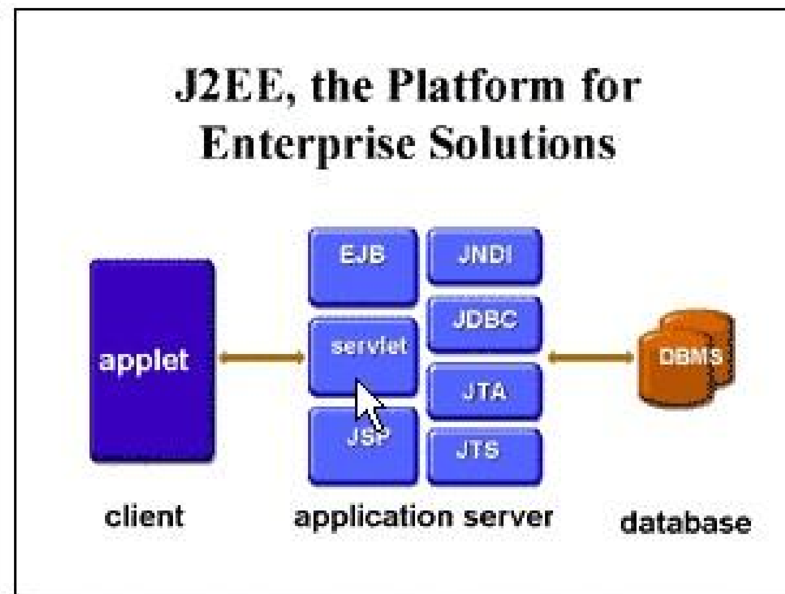
DATA SOURCE OBJECTS

Tecnologia que aproveita as vantagens da standardização dos URL para estabelecer ligações a bases de dados. Estes DSO possibilitam “pooling” e transacções distribuídas, em ambientes empresariais.



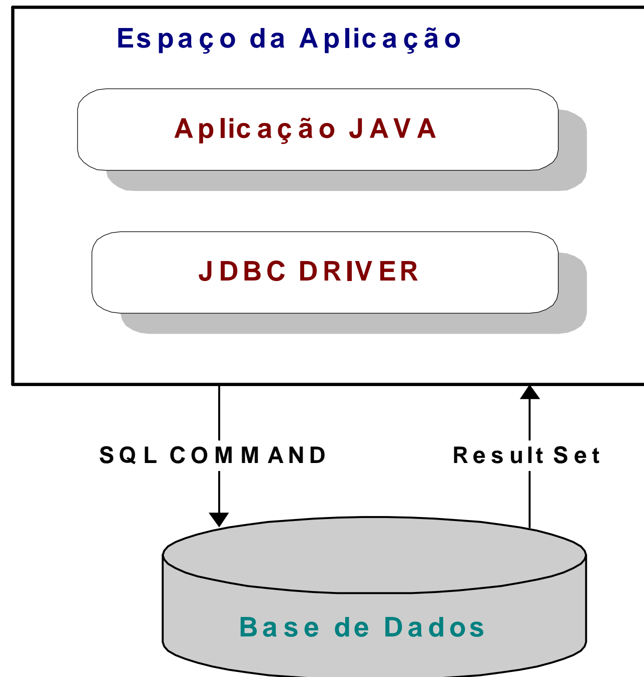
A JDBC API está incluída nas plataformas J2SE e J2EE

Nestas plataformas qualquer aplicação pode aceder a dados contidos nas mais diversas bases de dados, quer locais quer remotas, de uma forma quase transparente.

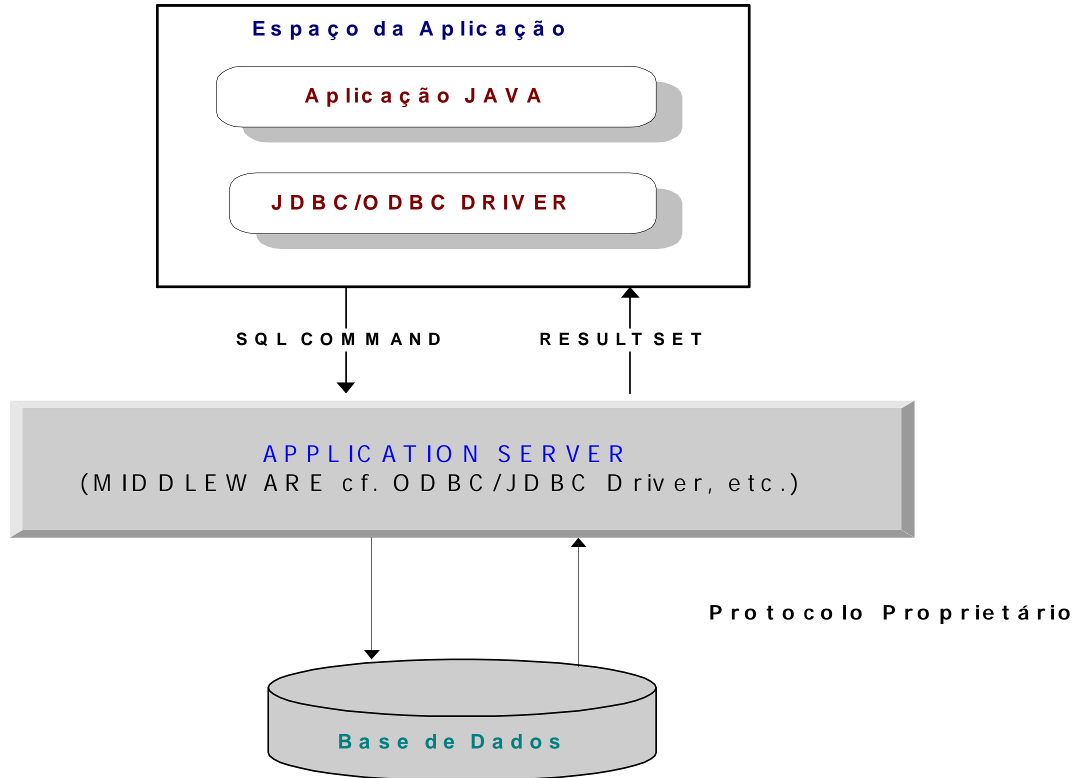


ARQUITECTURAS TÍPICAS: VISTA DA PROGRAMAÇÃO

JDBC - Modelo de 2 camadas
"Two-tier" Model



JDBC - Modelos de 3 camadas
"Three-tier" Model



JDBC – 4 Passos Básicos

1.- Estabelecer a ligação à Base de Dados.
2.- Enviar comando SQL.
3.- Processar resultados.
4.- Fechar a ligação à Base de Dados.

PACKAGE FUNDAMENTAL **import java.sql.***

java.sql.DriverManager
java.sql.Connection
java.sql.Statement
java.sql.ResultSet
java.sql.SQLException
java.sql.ResultSetMetaData

TEMPLATES TÍPICOS

PASSO 1: Estabelecer a ligação à Base de Dados.

a) Carregar o driver pretendido
b) Abrir a ligação à base de dados

```
// carregar a classe correspondente ao “driver”
// neste caso o “driver” ODBC
Class.forName(“sun.jdbc.odbc.JdbcOdbcDriver”);
```

ou, de forma mais segura:

```
try {
        Class.forName(“sun.jdbc.odbc.JdbcOdbcDriver”);
    }
catch(ClassNotFoundException e) {
        // tratamento do erro
}
```

// abrir a ligação através de …

```
DriverManager.getConnection(String url);
DriverManager.getConnection(String url, String pass, String user);
DriverManager.getConnection(String url, Properties.info);
```

Exº:

```
Connection sessao1;
try {
    sessao1 =
        DriverManager.getConnection(”jdbc:odbc:Catalogo”,””, ””);
     }
catch(SQLException e) {
       // tratamento do erro
}
```

Nota: É assim criada uma sessão designada sessao1 sobre a base de dados de nome Catalogo.

PASSO 2: Criar e Enviar comandos SQL.

```
// criar o comando SQL, executar e receber resultados
// a) createStatement()
// b) execute???()


String strSql = "SELECT titulo FROM livros WHERE ano > 2000";
Statement instrucao1;
ResultSet result;

try {
        instrucao1 = sessao1.createStatement();
        result = instrucao1.executeQuery(strSql);
   }
catch(SQLException exc) { ...  }
```

e também, cf. java.sql.Statement

int executeUpdate(String sql), para INSERT, UPDATE e DELETE
boolean execute(String sql), para comandos genéricos CREATE
ResultSet getResultSet(), associado a execute()

PASSO 3: Processar resultados.

```
// processar resultados
while( result.next() ) {                    // iteração sobre linhas
   System.out.println("Tipo: " + result.getInt("type_id")); ........
}
```

PASSO 4: Fechar ligações e sessão.

```
// fechar ligações
sessao1.close();
```

EXEMPLO COMPLETO PARA ANÁLISE:

http://www.eas.asu.edu/~cse494db/IonJDBC/JDBCExample.html

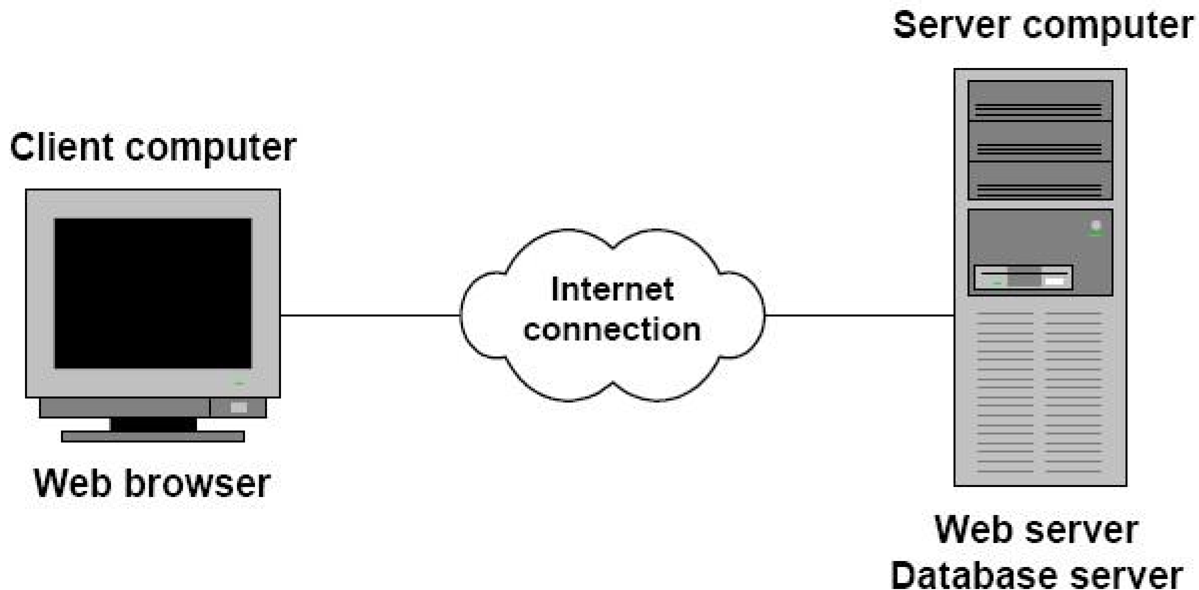
JAVA Avançado 3

Notas introdutórias básicas sobre

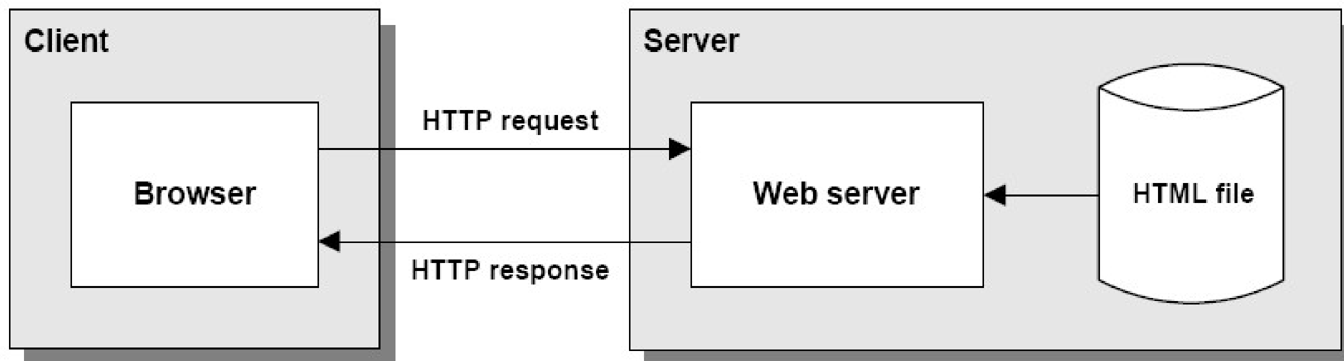
- Java Web Programming
- Servlets e Java Server Pages

QUESTÕES BÁSICAS ESSENCIAIS:

- **Uma aplicação Web é um conjunto de páginas web que são geradas em resposta aos pedidos do utilizador da aplicação;**

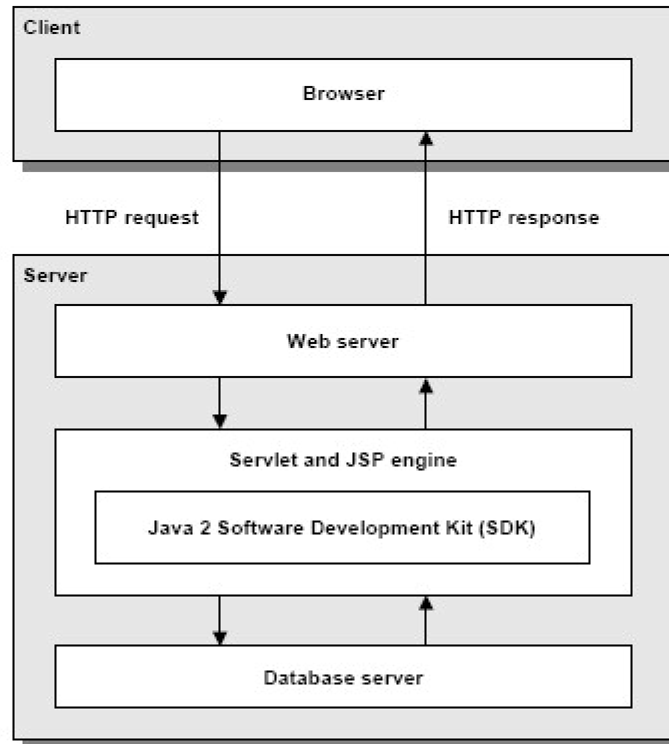


- Para executar uma aplicação web o cliente necessita de ter um web browser e o servidor necessita de ter software de serviço (servidor) web;



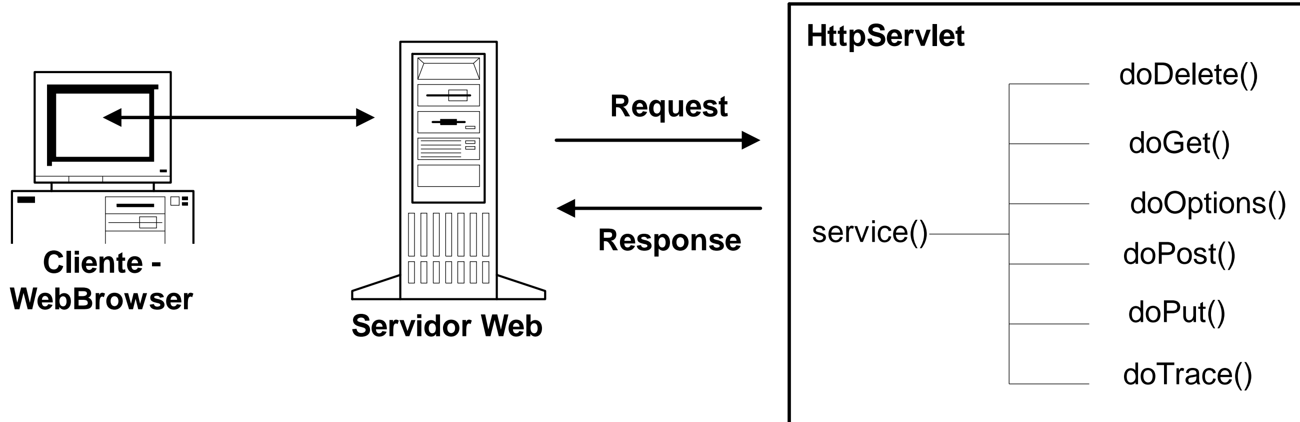
- HTML é a linguagem que o browser sabe converter em Interface com o Utilizador; HTTP é o protocolo de comunicação entre servidores e clientes web;
- Um browser solicita uma página a um servidor enviando um http request, ao qual o servidor envia uma http response.

- **Para executar aplicações Java para a web, o servidor necessita de ter instalada uma arquitectura J2SE ou J2EE, juntamente com uma máquina/motor para «servlets» e JSP, como Tomcat.**



- Uma JSP ou «java server page» é HTML com código Java embebido. Quando uma JSP é solicitada, é convertida numa «servlet» e compilada pela máquina/motor JSP.
- Uma «servlet» é uma classe Java cujas instâncias são executadas no servidor. Para aplicações web, uma «servlet» é uma subclasse da classe `HttpServlet`. Curiosamente, para uma «servlet» passar HTML ao browser, usa o método `println()` para criar a String correcta em geral sobre uma `PrintWriter`.
- Numa aplicação web de Java, servlets são usados para realizar os processamentos necessários e JSPs para criar as páginas HTML.

ARQUITECTURA JAVA SERVLETS 2.2

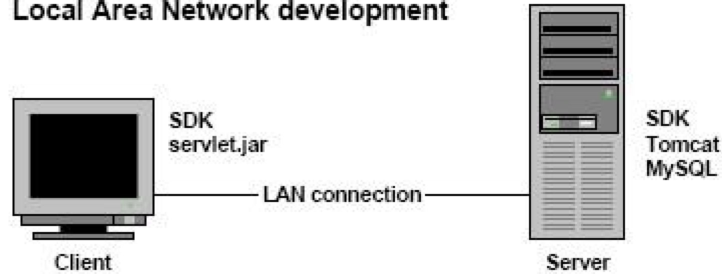


protected void service(HttpServletRequest req,
HttpServletResponse resp)
throws ServletException, IOException;

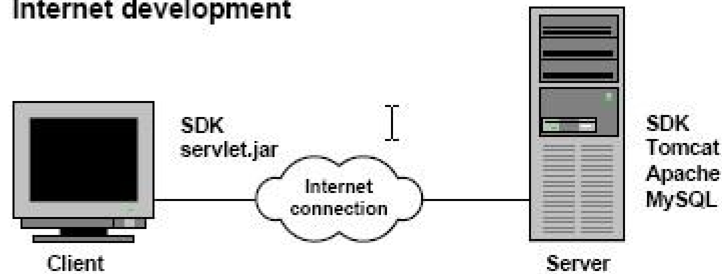
CONFIGURAÇÕES CONCRETAS

Stand-alone development

Local Area Network development

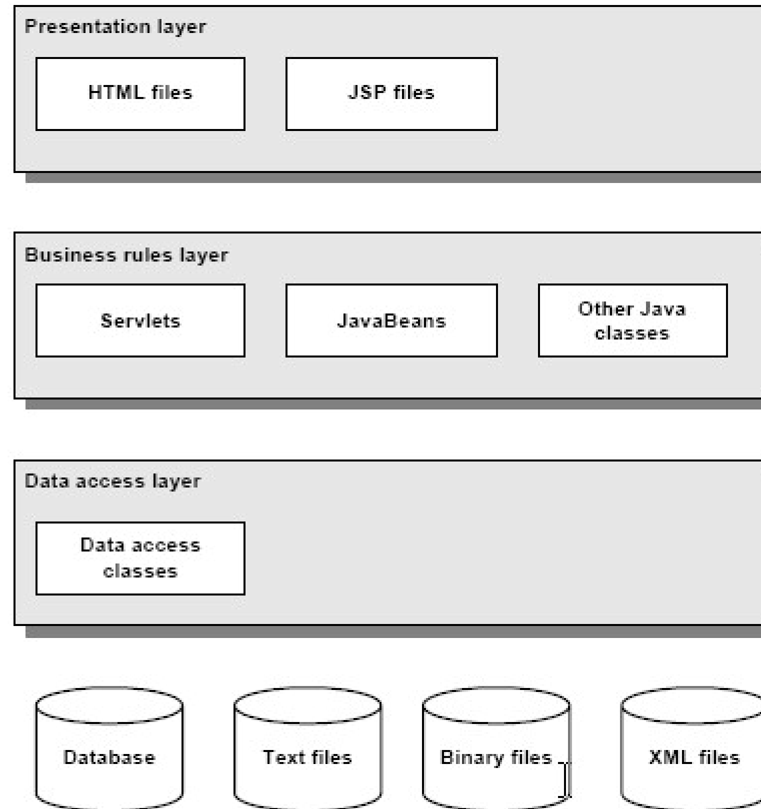


Internet development



- **Quando se usa a Internet como meio de comunicação entre cliente e servidor, é necessário ter um «web server» para além do servidor de JSP e de «servlets».**
- **Como em qualquer outra aplicação, uma aplicação web deve ser divida em três camadas: apresentação, lógica e dados.**

APLICAÇÃO TÍPICA JAVA WEB



JSP – JAVA SERVER PAGES vs. ASP

	ASP	JSP
Plataformas	Windows	Windows, Mac, Linux
Web Server	IIS ou PWS	Qualquer
Cross-platform comp.	Não	Beans, Ebeans, JSP+
Segurança - crashes	Não	Sim
Protecção de Memória	Não	Sim
Linguagem de Script.	VBScript, JScript	JAVA, JScript
Legacy Databases	Sim (COM)	SIM (JDBC)
Integração com BDs	Via ODBC	Via ODBC/JDBC
Componentes	COM	Beans, Ebeans, JSP+
Ferramentas	Sim	Sim