



# Pacotes e Encapsulamento

*Helder da Rocha*  
[www.argonavis.com.br](http://www.argonavis.com.br)

- **Pacotes básicos**
  - **java.lang**: classes fundamentais - importado automaticamente
  - **java.util**: classes utilitárias
  - **java.io**: classes para entrada e saída
  - **java.net**: classes para uso em rede (TCP/IP)
  - **java.sql**: classes para acesso via JDBC
  - **java.awt**: interface gráfica universal nativa
  - **javax.swing**: interface gráfica independente de plataforma
  - **java.text**: internacionalização, transformação e formatação de texto

- *Interfaces*
  - *Cloneable*
  - *Runnable*
- *Classes*
  - *Boolean, Number (e subclasses), Character, Void*
  - *Class*
  - *Math*
  - *Object*
  - *Process, Thread, System e Runtime*
  - *String e StringBuffer*
  - *Throwable e Exception (e subclasses)*

# Classe `java.lang.Object` e interface `Cloneable`

- Métodos de `Object` que devem ser sobrepostos

```
public boolean equals(Object obj)
public String toString()
public int hashCode()
protected Object clone()
                    throws CloneNotSupportedException
```

*por ser protected  
TEM que ser sobreposto  
para que possa ser usado  
em qualquer classe*

- `Cloneable`

- Para que um objeto possa ser clonado, através do método `clone()` herdado de `Object`, é preciso que ele implemente a interface `Cloneable`

- Como fazer:

```
class SuaClasse implements Cloneable
class SuaClasse extends SupClasse
    implements Cloneable
```

# Como estender Object

- *Há vários métodos em Object que devem ser sobrepostos pelas subclasses*
  - *A subclasse que você está estendendo talvez já tenha sobreposto esses métodos mas, alguns deles, talvez precisem ser redefinidos para que sua classe possa ser usada de forma correta*
- *Métodos que devem ser sobrepostos*
  - *boolean equals(Object o)*
    - *Defina o critério de igualdade para seu objeto*
  - *int hashCode()*
    - *Para que seu objeto possa ser localizado em Hashtables*
  - *String toString()*
    - *Sobreponha com informações específicas do seu objeto*
  - *Object clone()*
    - *se você desejar permitir cópias do seu objeto*

# Como sobrepor equals()

- *Determine quais os critérios (que propriedades do objeto) que podem ser usados para dizer que um objeto é igual a outro*
  - *O raio, em um objeto Círculo*
  - *O número de série, em um objeto genérico*
  - *O nome, sobrenome e departamento, para um empregado*
  - *A chave primária, para um objeto de negócio*
- *Implemente o equals(), testando essas condições e retornando true apenas se forem verdadeiras (false, caso contrário)*
  - *Verifique que a assinatura seja igual à definida em Object*

# *instanceof* e exemplo com equals()

- *instanceof* é um operador usado para comparar uma referência com uma classe
  - a expressão será true se a referência for do tipo de uma classe ou subclasse testada e false, caso contrário
- Exemplo: sobreposição de equals()

```
class Point {
    private int x, y;
    public boolean equals(Object obj) {
        if (obj instanceof Point) {
            Point ponto = (Point) obj;
            if (ponto.x == this.x && ponto.y == this.y) {
                return true;
            }
        }
        return false;
    }
}
```

# Como sobrepor toString()

- *toString()* deve devolver um *String* que possa representar o objeto quando este for chamado em uma concatenação ou representado como texto
  - Decida o que o *toString()* deve retornar
  - Faça chamadas *super.toString()* se achar conveniente
  - Prefira retornar informações que possam identificar o objeto (e não apenas a classe)
  - *toString()* é chamado automaticamente em concatenações usando a referência do objeto

# Como sobrepor hashCode()

- *hashCode()* deve devolver um número inteiro que represente o objeto
  - use uma combinação de variáveis, uma chave primária ou os critérios usados no *equals()*
  - número não precisa ser único para cada objeto mas dois objetos iguais devem ter o mesmo número.
  - O método *hashCode()* é chamado automaticamente quando referências do objeto forem usadas em coleções do tipo hash (*Hashtable*, *HashMap*)
  - *equals()* é usado como critério de desempate, portanto, se implementar *hashCode()*, implemente *equals()* também.

# Como sobrepor clone()

- `clone()` é chamado para fazer cópias de um objeto

- `Circulo c = new Circulo(4, 5, 6);`

- `Circulo copia = (Circulo) c.clone();`

*cast é necessário  
porque clone()  
retorna Object*

- Se o objeto apenas contiver tipos primitivos como seus campos de dados, é preciso

- declarar que a classe implementa `Cloneable`

- sobrepor `clone()` da seguinte forma:

*é preciso sobrepor  
clone() porque ele  
é definido como  
protected*

```
public Object clone() {  
    try {  
        return super.clone();  
    } catch (CloneNotSupportedException e) {  
        return null;  
    }  
}
```

## Como sobrepor clone() (2)

- *Se o objeto contiver campos de dados que são referências a objetos, é preciso fazer cópias desses objetos também*

```
public class Circulo {
    private Point origem;
    private double raio;
    public Object clone() {
        try {
            Circulo c = super.clone();
            c.origem = origem.clone(); // Point deve ser clonável!
            return c;
        } catch (CloneNotSupportedException e) {return null;}
    }
}
```

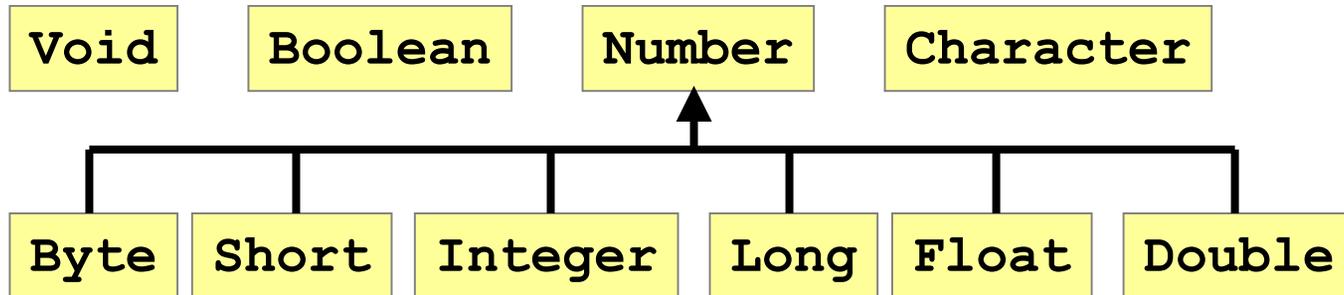
# A classe `java.lang.Math`

- A classe `Math` é uma classe final (não pode ser estendida) com construtor `private` (não permite a criação de objetos)
- Serve como repositório de funções e constantes matemáticas
- Para usar, chame a constante ou função precedida do nome da classe:
  - `double distancia = Math.sin(0.566);`
  - `int sorte = (int) (Math.random() * 1000);`
  - `double area = 4 * Math.PI * raio;`
- Consulte a documentação sobre a classe `Math` e explore as funções disponíveis

# As classes empacotadoras (wrappers)

- *Classes que servem para embutir tipos primitivos para que possam ser manipulados como objetos*
- *Exemplo:*
  - *um vetor de tipos Object pode conter qualquer objeto mas não tipos primitivos*
  - *Solução: colocar os tipos dentro de wrappers*
- *Construtores aceitam literais ou Strings*
- *Há métodos para realizar conversões entre tipos e de tipos para Strings e vice-versa*

# Classes empacotadoras



- *Criação e conversão*

```
Integer n = new Integer(15);
```

```
Integer m = Integer.valueOf(15);
```

```
int q = n.intValue(n);
```

*Mais eficiente*

- *Para conversão de String em tipos primitivos*

```
int i = Integer.parseInt("15");
```

```
double i = Double.parseDouble("15.9");
```

```
boolean b = (new
```

```
Boolean("true")).booleanValue();
```

# Controle e acesso ao sistema

- *Classe System dá acesso a objetos do sistema operacional*

- *System.out - saída padrão*
- *System.err - saída padrão de erro*
- *System.in - entrada padrão*

- *Runtime e Process permitem controlar processos externos (processos do S.O.)*

```
Runtime r = System.getRuntime();
```

```
Process p =
```

```
    r.exec("c:\program~1\micros~1\msie.exe");
```

- *Thread e Runnable lidam com processos internos (processos da aplicação - threads)*

- *serão tratados em capítulo a parte*

- *Objetos da classe String são constantes em Java*
  - *não podem ter seu valor alterado (não existem métodos que permitam a alteração de Strings)*
  - *concatenações sempre geram novos objetos*
  - *podem ser criados através de atribuição de constantes*
- *Classe StringBuffer não pode ser usada no lugar de String, mas*
  - *pode ser transformada em String quando necessário*
  - *pode ter seu conteúdo alterado (existem métodos append(), replace(), etc.)*
  - *concatenações usando append() operam no próprio objeto (maior eficiência - evita criar novos objetos que serão descartados)*
  - *criação só pode ser realizada via construtor*

- *Classes descendentes de `java.lang.Throwable` representam situações de erro*
  - *Erros graves irre recuperáveis (descendentes da classe `java.lang.Error`)*
  - *Exceções de tempo de execução (descendentes da classe `java.lang.Exception`)*
- *São usadas em blocos try-catch*
  - *serão tratadas em capítulo a parte*
- *São classes, portanto...*
  - *... podem ser estendidas*
  - *... podem ser usadas para criar objetos via construtor*
  - *... podem ter métodos chamados*

# Identificação de tipos em tempo de execução

- A classe *Class* contém métodos que permitem enxergar a interface pública de uma classe
  - saber quais métodos existem para serem chamados
  - saber quais os parâmetros e tipos dos parâmetros dos métodos
  - saber quais os construtores e campos públicos
- Com essas informações, é possível carregar uma classe compilada, em tempo de execução, e usá-la
  - Há métodos para carregar uma classe pelo nome, passado como *String* (*forName*)
  - Há métodos (*newInstance*) para criar objetos a partir do nome da classe

# Outros pacotes

- *Outros pacotes mais importantes da API Java 2 serão explorados em aulas futuras*
- *Use a documentação e experimente utilizar classes de algum pacote de seu interesse*
- *Sugestões*
  - *java.text: descubra como formatar texto para impressão (por exemplo, 2 decimais e 3 dígitos fracionários)*
  - *java.io: descubra como abrir um arquivo*
  - *java.sql: descubra como conectar-se a um banco*

- 1. Crie dois *Circulos* e dois *Pontos* em um programa, com as mesmas coordenadas, e teste se são iguais usando `==` e `equals()`. Imprima o valor da referência de cada um (concatene-a com uma *String*).
- 2. Implemente os seguintes métodos para *Circulo* e *Point*:
  - `equals()`
  - `toString()`
  - `hashCode()`
  - `clone()`
- 3. Refaça o exercício 1.

- Um pacote é uma coleção de classes e interfaces Java, agrupadas
  - O pacote faz parte do nome da classe
    - Uma classe **NovaYork** do pacote **simulador.cidades** pode ser usada por outra classe (de pacote diferente) apenas se for usado o nome completo **simulador.cidades.NovaYork**
    - **import**, pode ser usado, para compartilhar os espaços de nomes de pacotes diferentes (assim, não será preciso usar nomes completos)
  - Toda classe pertence a um pacote
    - Se a classe não tiver declaração **package**, ela pertence ao pacote **default**, que, durante a execução, corresponde à raiz do Classpath.

- O **Classpath** é uma propriedade do sistema que contém as localidades onde o JRE irá procurar classes. Consiste de
  - 1. JARs nativos do JRE (API Java 2)
  - 2. Extensões do JRE (subdiretórios `$JAVA_HOME/jre/lib/classes` e `$JAVA_HOME/jre/lib/ext`)
  - 3. Lista de caminhos definidos na variável de ambiente `CLASSPATH` e/ou na opção de linha de comando `-classpath (-cp)` da aplicação java.
- A ordem acima é importante
  - Havendo mais de uma classe com mesmo pacote/Nome somente a **primeira** classe encontrada é usada. Outras são ignoradas
  - Há risco de **conflitos**. API nova sendo carregada depois de antiga pode resultar em classes novas chamando classes antigas!
  - JARs são carregados em ordem alfabética
  - A ordem dos caminhos na variável `CLASSPATH` (ou opção `-cp`) também é significativa

# Variável CLASSPATH e -cp

- Em uma instalação típica, **CLASSPATH** contém apenas "."
  - **Pacotes** iniciados no diretório atual (onde o interpretador java é executado) são encontrados (podem ter suas classes importadas)
  - **Classes** localizadas no diretório atual são encontradas.
- Geralmente usada para definir caminhos para **uma** aplicação
  - Os caminhos podem ser diretórios, arquivos ZIP ou JARs
  - Pode acrescentar novos caminhos mas não pode remover caminhos do Classpath do JRE (básico e extensões)
- A opção **-cp** (**-classpath**) substitui as definições em CLASSPATH
- Exemplo de definição de CLASSPATH
  - no DOS/Windows

```
set CLASSPATH=extras.jar;. ;c:\progs\java
java -cp %CLASSPATH%;c:\util\lib\jsw.zip gui.Programa
```
  - no Unix (sh, bash)

```
CLASSPATH=extras.jar:./home/mydir/java
export CLASSPATH
java -classpath importante.jar:$CLASSPATH Programa
```

# Classpath do JRE

- Colocar JARs no subdiretório **ext** ou **classes** e pacotes no diretório **classes** automaticamente os inclui no Classpath para todas as aplicações da JVM
  - Carregados **antes** das variáveis CLASSPATH e -cp
  - **Use com cuidado!** Pode provocar **conflitos**. Coloque nesses diretórios apenas os JARs e classes usados **na maior parte** de suas aplicações
- Exemplo: suponha que o Classpath seja
  - Classpath JRE: %JAVA\_HOME%\jre\lib\rt.jar;
  - Classpath Extensão JRE: %JAVA\_HOME%\jre\lib\ext\z.jar
  - Variável de ambiente CLASSPATH: .;c:\programas;e que uma classe, localizada em **c:\exercicio** seja executada. Se esta classe usar a classe **arte.fr.Monet** o sistema irá procurá-la em
  1. %JAVA\_HOME%\jre\lib\rt.jar\arte\fr\Monet.class
  2. %JAVA\_HOME%\jre\lib\ext\z.jar\arte\fr\Monet.class
  3. c:\exercício\arte\fr\Monet.class
  4. c:\programas\arte\fr\Monet.class

# Como criar e usar um pacote

- Para **criar** um pacote é preciso
  1. Declarar o nome do pacote em cada unidade de compilação
  2. Guardar a classe compilada em uma localidade (caminho) compatível com o pacote declarado
    - O "caminho de pontos" de um pacote, por exemplo, **simulador.cidade.aeroporto** corresponde a um caminho de diretórios **simulador/cidade/aeroporto**
- Para **usar** um pacote (usar suas classes) é preciso
  1. Colocar a raiz do pacote (pasta "simulador", por exemplo) no Classpath
  2. Importar as classes do pacote (ou usar suas classes pelo nome completo)
    - A instrução `import` é opcional

- Geralmente, aplicações Java são distribuídas em arquivos **JAR**
  - São extensões do formato ZIP
  - armazenam pacotes e preservam a hierarquia de diretórios
- Para **usar** um JAR, é preciso incluí-lo no Classpath
  - via CLASSPATH no contexto de execução da aplicação, ou
  - via parâmetro `-classpath` (`-cp`) do interpretador Java, ou
  - copiando-o para `$JAVA_HOME/jre/lib/ext`
- Para **criar** um JAR

```
jar cvf classes.jar C1.class C2.class xyz.gif abc.html
jar cf mais.jar -C raiz_onde_estao_pacotes/ .
```
- Para **abrir** um JAR

```
jar xvf classes.jar
```
- Para **listar o conteúdo** de um JAR

```
jar tvf classes.jar
```

# Criação de bibliotecas

- Uma **biblioteca** é um conjunto de classes base (para criar objetos ou extensão) ou classes utilitárias (contendo métodos úteis)
- Para **distribuir** uma biblioteca
  - Projete, crie e compila as classes (use pacotes)
  - Guarde em arquivo JAR
- Para **usar** a biblioteca
  - importe as classes da sua biblioteca em seus programas
  - rode o interpretador tendo o JAR da sua biblioteca no Classpath

# Uma biblioteca (simples)

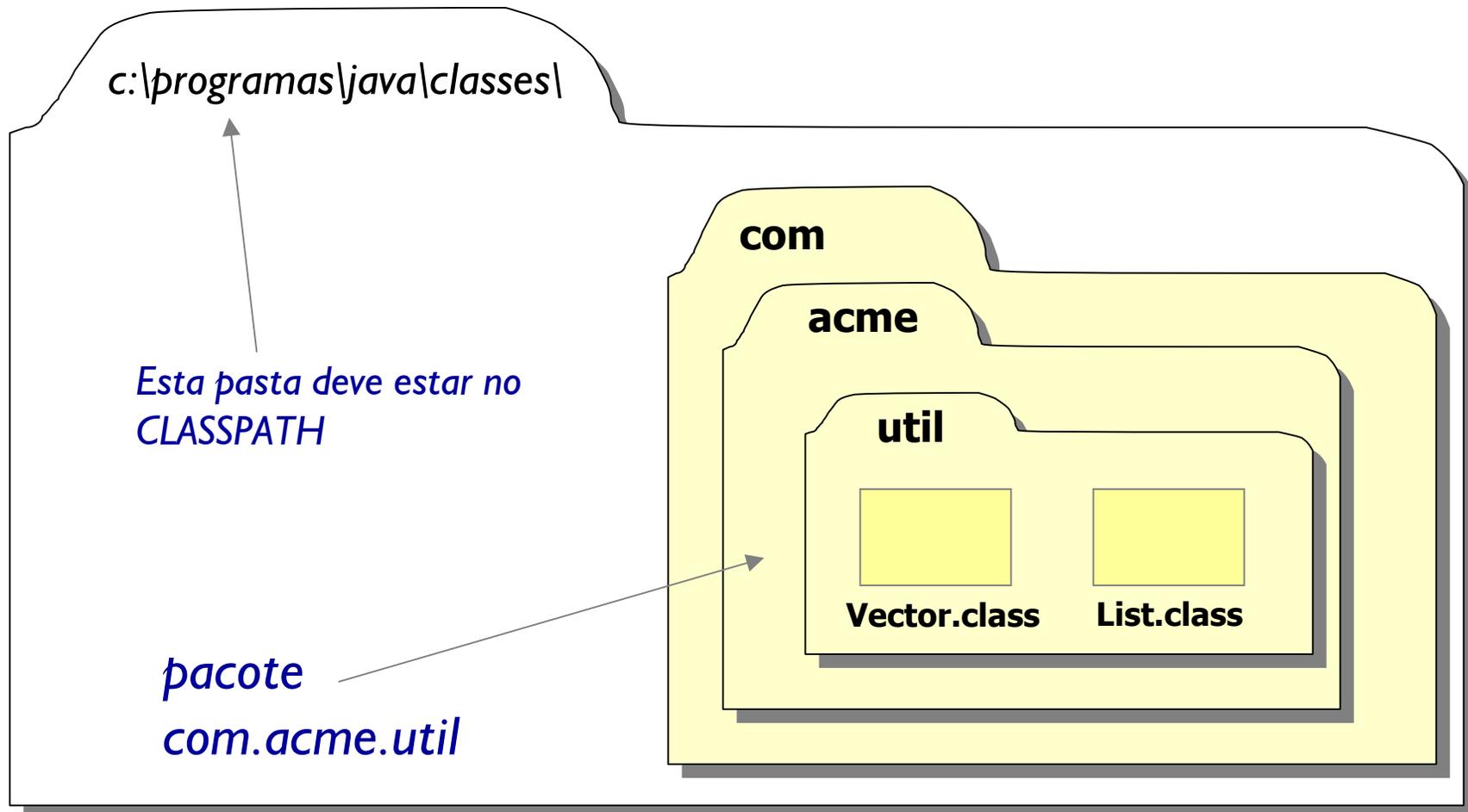
## List.java

```
package com.acme.util;
public class List {
    public List() {
        System.out.println("com.acme.util.List");
    }
}
```

## Vector.java

```
package com.acme.util;
public class Vector {
    public Vector() {
        System.out.println("com.acme.util.Vector");
    }
}
```

# Onde armazenar

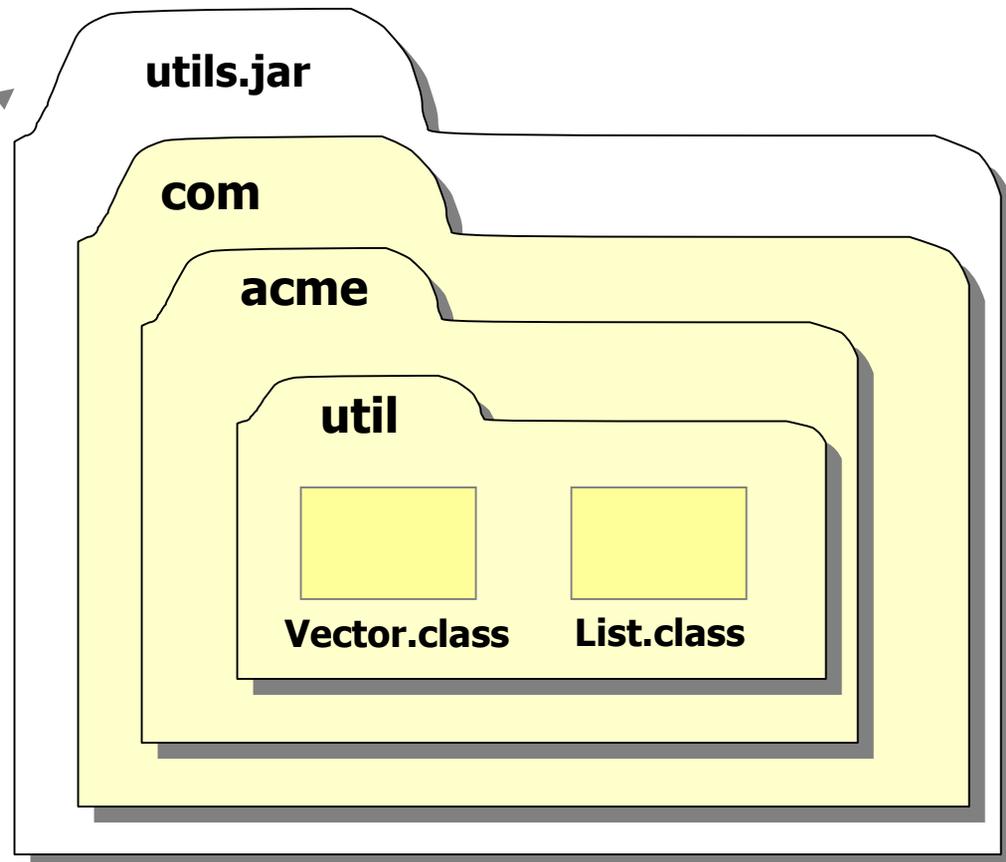


# Ou em um JAR

O caminho até a localização do JAR deve estar no CLASSPATH (pode ser especificado no momento da execução:

```
java -classpath  
    %CLASSPATH%;utils.jar  
    NomeProgExecutavel
```

JAR também pode ser jogado no diretório ext do JRE



# Como usar a biblioteca

- Programa que usa biblioteca pode estar em qq lugar
  - É preciso que caminho até a biblioteca (pasta onde começa pacote ou caminho até JAR) esteja definido no Classpath

```
CLASSPATH=%CLASSPATH%;c:\programas\java\classes
```

```
CLASSPATH=%CLASSPATH%;c:\jars\utils.jar
```

```
// Programa que usa a biblioteca
import com.acme.util.*;
public class LibTest {
    public static void main(String[] args) {
        Vector v = new Vector();
        List m = new List();
    }
}
```

- Pode também usar

```
java -cp %CLASSPATH%;c:\...\java\classes LibTest
```

```
java -cp %CLASSPATH%;c:\jars\utils.jar LibTest
```

# Como criar um executável

- Todo JAR possui um arquivo chamado **Manifest.mf** no subdiretório `/META-INF`.
  - Lista de pares **Nome: atributo**
  - Serve para incluir informações sobre os arquivos do JAR, `CLASSPATH`, classe `Main`, etc.
  - Se não for indicado um arquivo específico, o sistema gerará um `Manifest.mf default` (vazio)
- Para tornar um JAR executável, o `Manifest.mf` deve conter a linha:
  - `Main-Class: nome.da.Classe`
- Crie um arquivo de texto qualquer com a linha acima e monte o JAR usando a opção `-manifest`:

```
jar cfm arq.jar arquivo.txt -C raiz
```
- Para executar o JAR em linha de comando:

```
java -jar arq.jar
```

# Colisões entre classes

- Se houver, no mesmo espaço de nomes (Classpath + pacotes), duas classes com o mesmo nome, não será possível usar os nomes das classes de maneira abreviada
- Exemplo: classes *com.acme.util.List* e *java.util.List*

```
import com.acme.util.*;
import java.util.*;
class Xyz {
    // List itens; // NAO COMPILA!
    // java.util.List lista;
}
```

```
import com.acme.util.*;
class Xyz {
    List itens; // com.acme.util.List
    java.util.List lista;
}
```

- *1. Crie um JAR executável:*
  - *Use uma aplicação gráfica (pegue algum exercício anterior)*
  - *Crie o manifest a partir de um arquivo de texto contendo apenas a linha Main-class.*
- *2. Use o Apache Ant para criar um roteiro de construção para sua aplicação contendo os alvos:*
  - *init: para criar um diretório onde ficarão os binários*
  - *clean: para apagar esse diretório*
  - *compile: para compilar a aplicação*
  - *jar: para empacotar tudo em um JAR.*