

# Métodos da Classe String

[extraído de [https://www.w3schools.com/java/java\\_ref\\_string.asp](https://www.w3schools.com/java/java_ref_string.asp)]

Method	Description	Return Type
<a href="#"><u>charAt()</u></a>	Returns the character at the specified index (position)	char
<a href="#"><u>compareTo()</u></a>	Compares two strings lexicographically	int
<a href="#"><u>compareToIgnoreCase()</u></a>	Compares two strings lexicographically, ignoring case differences	int
<a href="#"><u>concat()</u></a>	Appends a string to the end of another string	String
<a href="#"><u>contains()</u></a>	Checks whether a string contains a sequence of characters	boolean
<a href="#"><u>endsWith()</u></a>	Checks whether a string ends with the specified character(s)	boolean
<a href="#"><u>equals()</u></a>	Compares two strings. Returns true if the strings are equal, and false if not	boolean
<a href="#"><u>equalsIgnoreCase()</u></a>	Compares two strings, ignoring case considerations	boolean
<a href="#"><u>indexOf()</u></a>	Returns the position of the first found occurrence of specified characters in a string	int
<a href="#"><u>isEmpty()</u></a>	Checks whether a string is empty or not	boolean
<a href="#"><u>lastIndexOf()</u></a>	Returns the position of the last found occurrence of specified characters in a string	int
<a href="#"><u>length()</u></a>	Returns the length of a specified string	int

<a href="#">replace()</a>	Searches a string for a specified value, and returns a new string where the specified values are replaced	String
replaceFirst()	Replaces the first occurrence of a substring that matches the given regular expression with the given replacement	String
replaceAll()	Replaces each substring of this string that matches the given regular expression with the given replacement	String
split()	Splits a string into an array of substrings	String[]
<a href="#">startsWith()</a>	Checks whether a string starts with specified characters	boolean
subSequence()	Returns a new character sequence that is a subsequence of this sequence	CharSequence
substring()	Returns a new string which is the substring of a specified string	String
toCharArray()	Converts this string to a new character array	char[]
<a href="#">toLowerCase()</a>	Converts a string to lower case letters	String
toString()	Returns the value of a String object	String
<a href="#">toUpperCase()</a>	Converts a string to upper case letters	String
<a href="#">trim()</a>	Removes whitespace from both ends of a string	String
valueOf()	Returns the string representation of the specified value	String

Criando tipos genéricos

# Tipos genéricos

- Novidade do Java 5.0;
- Funcionalidade já existente em outras linguagens (ex.: *templates* do C++);
- Teoria estudada e solidificada;
- Muitas bibliotecas são genéricas:
- Código complicado de ler e manter;
- Coerção leva a erros em tempo de execução.

```
public class Node<T> {  
    T value;  
    Node<T> next;  
  
    public Node(T value, Node<T> next) {  
        this.value = value;  
        this.next = next;  
    }  
  
    public Node(T value) {  
        this(value, null);  
    }  
}
```

# Funcionamento

- Cria-se uma classe “genérica”:
  - *Trabalha com um tipo T, desconhecido;*
  - *Tipo será atribuído na definição da referência.*

```
public class Casulo<T> {
    private T elemento;

    public void colocar(T elem) {
        elemento = elem;
    }

    public T retirar() {
        return elemento;
    }
}
```

# Funcionamento

- Referência e construtor definem o tipo manipulado pela classe genérica;
- Compilador pode efetuar checagens de tipo.

```
Casulo<String> cs = new Casulo<String>();
cs.colocar("Uma string");
// Erro: cs.colocar(new Integer(10));
String s = cs.retirar();
```

```
Casulo<Object> co = new Casulo<Object>();
co.colocar("Uma string");
co.colocar(new Integer(10));
Object o = co.retirar();
```



# Herança de tipos genéricos

- Os conceitos de herança podem se confundir quando usamos tipos genéricos.

```
Casulo<String> cs = new Casulo<String>();
Casulo<Object> co = cs;
```

- O código acima gera erro;
- Por que? Object não é superclasse de String?

```
co.colocar(new Integer()); // OK!
String s = cs.retirar(); // OK!
```

co e cs são o mesmo objeto e o código acima faria s receber um Integer!

## Curingas (*wildcards*)

- Considere, então, um código genérico:

```
void imprimir(Casulo<Object> c) {
    System.out.println(c.retirar());
}
```

- O código não é tão genérico assim:

```
imprimir(co); // OK!
imprimir(cs); // Erro!
```

Como acabamos de ver, não se pode converter  
Casulo<String> para Casulo<Object>!

## Curingas (*wildcards*)

- Para essa situação, podemos usar curingas:

```
void imprimir(Casulo<?> c) {  
    System.out.println(c.retirar());  
}
```

- Significa: o método `imprimir()` pode receber casulos de qualquer tipo.

# Curingas limitados

- Podemos também limitar o tipo genérico como sendo subclasse de alguma classe;

```
void desenhar(Casulo<? extends Forma> c) {  
    c.retirar().desenhar();  
    c.retirar().inverter();  
}
```

- Significa: o método `imprimir()` pode receber casulos de qualquer subclasse de `Forma` ou casulos de `Forma`;
- O compilador garante que o que retirarmos do casulo será uma `Forma` ou uma subclasse.

# Curingas limitados

- Mas, não podemos alterar uma classe com um curinga:

```
void teste(Casulo<? extends Forma> c) {
    // Erro: c pode ser Casulo<Retangulo>!
    c.colocar(new Circulo());
}
```

```
void outroTeste(Casulo<?> c) {
    // Erro: c pode ser Casulo<Integer>!
    c.colocar("Uma string!");
}
```

```
void desenhar(Casulo<? extends Forma> c) {
    c.retirar().desenhar();
    c.retirar().inverter();
}
```

```
public class Programa {  
  
    public static void print_node(Node<?> node) {  
        System.out.println("Value: " + node.value + " Next: " + node.next);  
  
        // OK porque aux é inteiro independente do tipo associado ao wildcard  
        node.aux = 3;  
  
        // Erro! Como o metodo não sabe o tipo que será passado no wildcard, não é  
        // possível saber se String é um tipo válido.  
        // node.value = "maria"  
    }  
  
    public static void main(String[] args) {  
        Node<String> n = new Node<String>("agua");  
        n.next = new Node<String>("bola");  
        print_node(n);  
    }  
}
```

# Métodos genéricos

- Novamente, este método não é tão genérico:

```
static void arrayToCollection(Object[] array,
                             Collection<Object> collection) {
    for (Object o : array) collection.add(o);
}
```

- E este gera erro de compilação:

```
static void arrayToCollection(Object[] array,
                             Collection<?> collection) {
    for (Object o : array)
        collection.add(o); // Erro!
}
```

# Métodos genéricos

- A solução:

```
static <T> void arrayToCollection(T[] array,
                                Collection<T> collection) {
    for (T o : array) collection.add(o);
}
```

- O método usa um tipo diferente a cada chamada:

```
// usa Long[] e Collection<Long>:
arrayToCollection(new Long[] {1L, 2L},
                 new ArrayList<Long>());
```



# Conclusões

- Usar tipos genéricos é relativamente simples e traz grandes vantagens;
- Criar tipos genéricos é mais complexo e envolve um entendimento mais aprofundado.

# Uso de Argumentos de Linha de Comando

*Demonstração ao vivo*



UNIVERSIDADE FEDERAL  
DO ESPÍRITO SANTO

Centro Tecnológico  
Departamento de Informática

Prof. Vítor E. Silva Souza  
<http://www.inf.ufes.br/~vitorsouza>

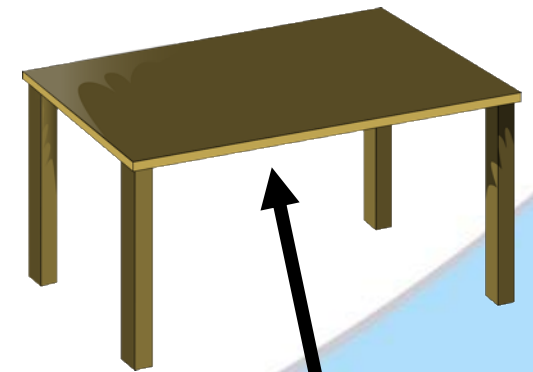
# [Desenvolvimento OO com Java] Organizando suas classes em Pacotes



Esta obra está licenciada com uma licença Creative Commons Atribuição-  
Compartilha Igual 4.0 Internacional: <http://creativecommons.org/licenses/by-sa/4.0/>.

# Por que organizar as classes?

- À medida que **aumenta** o número de classes, aumenta a chance de **coincidência** de nomes;
- Precisamos separar as classes em **espaços** de nomes;
- Java possui o conceito de **pacotes**:
  - *Espaço de nome para **evitar** conflitos;*
  - ***Agrupamento** de classes semelhantes;*
  - *Maneira de construir **bibliotecas** de classes;*
  - *Estabelecimento de políticas de **acesso** às classes.*

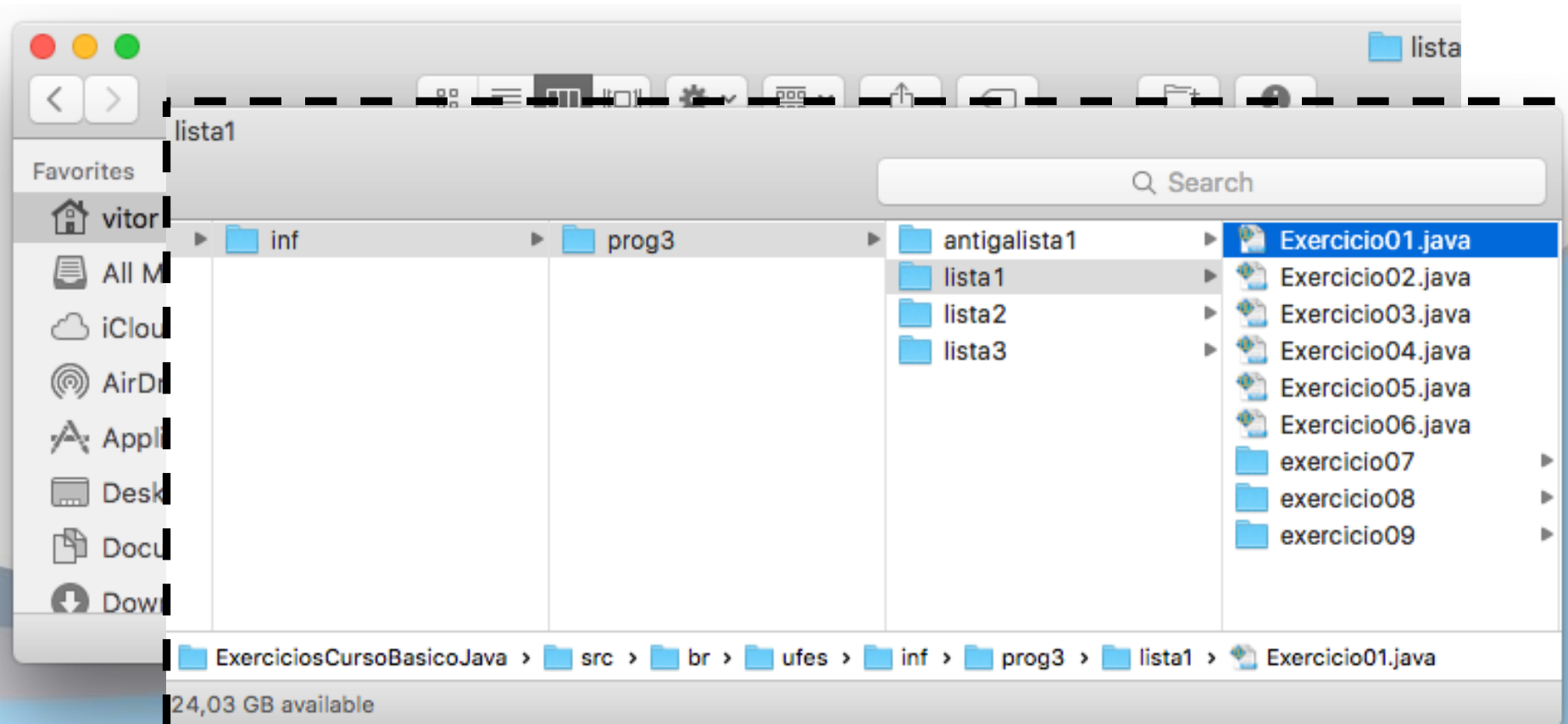


Pumpid	Date	pressure	temperature	volumeflow	rotational
1	2012-05-10	22222.0	33333.0	44444.0	2000.0
2	2012-05-11	22222.0	33333.0	44444.0	1500.0
3	2012-05-12	22222.0	33333.0	44444.0	2500.0
4	2012-05-13	22222.0	33333.0	44444.0	3000.0
5	2012-05-15	22222.0	33333.0	44444.0	2700.0
6	2012-05-04	22222.0	33333.0	44444.0	22.0
7	2012-05-11	22222.0	33333.0	44444.0	2000.0
8	2012-05-04	22222.0	33333.0	44444.0	5000.0
9	1970-01-01	22222.0	33333.0	44444.0	2000.0
10	1970-01-01	22222.0	33333.0	44444.0	1500.0

# Inspiração: organização de arquivos

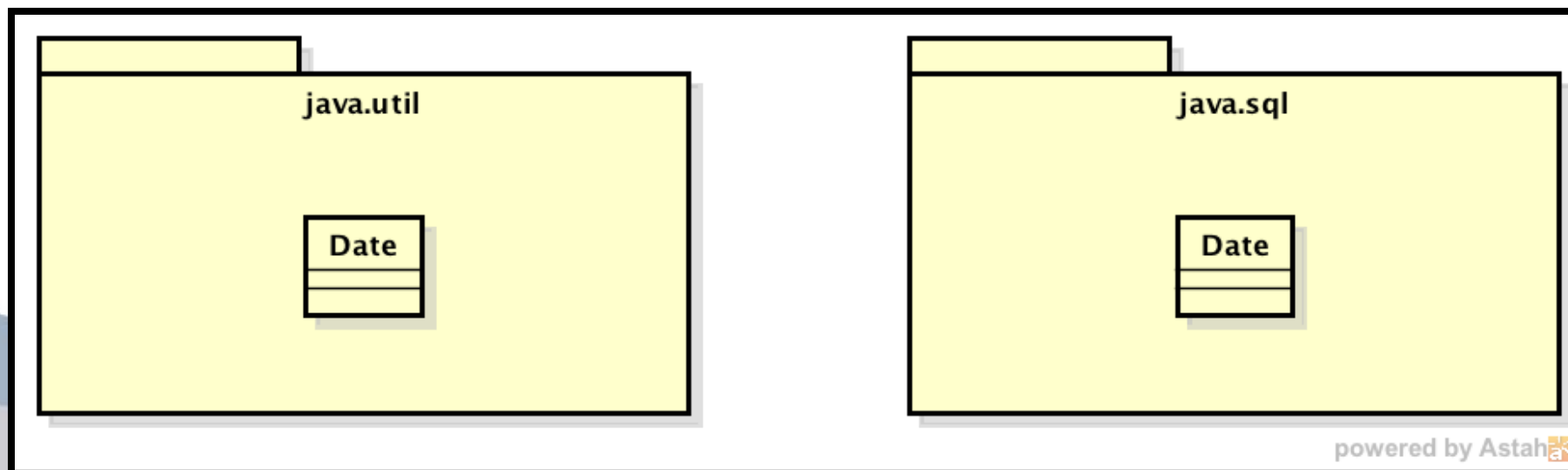
- Mesmo **problema**: não podemos ter 2 arquivos com mesmo **nome** na mesma **pasta**;
  - *Pastas definem espaços de nome, org. hierárquica.*

Em Java,  
pacotes se  
refletem em  
pastas no  
sistema



# Pacotes da API Java

- As **APIs** Java (ex.: Java SE) são **divididas** em pacotes:
  - *java.lang*: classes do **núcleo** da plataforma;
  - *java.util*: classes **utilitárias**;
  - *java.io*: classes para **I/O** (entrada/saída);
  - *Dentre muitos outros...*



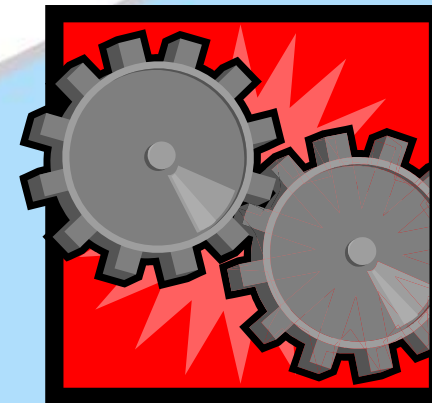
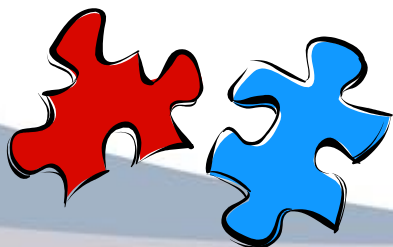
# Modularidade

- Decomposição do sistema em **módulos**:
  - *Coesos (baixo acoplamento);*
  - *Autônomos;*
  - *De interface simples e e*
- Fundamental

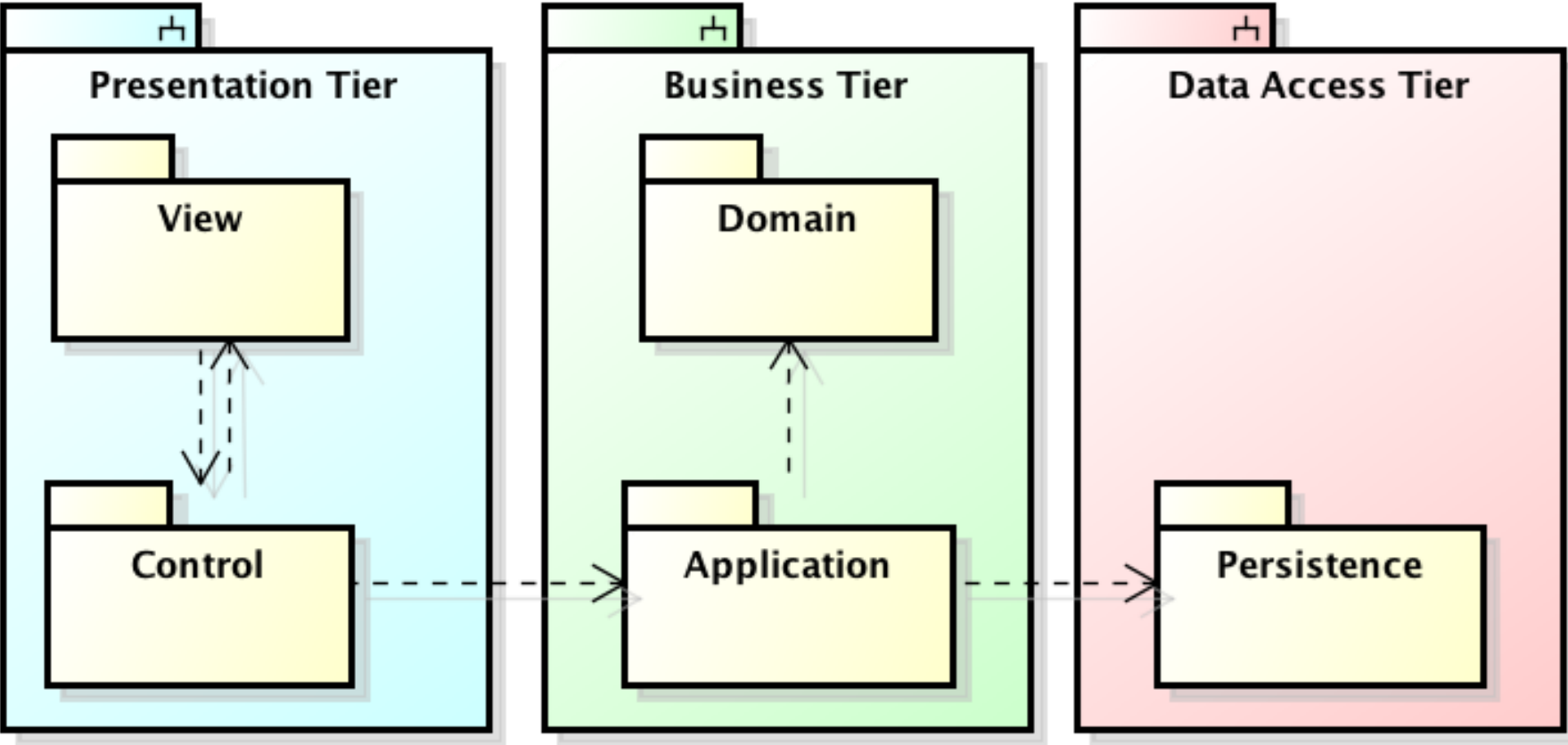


**PACOTES!**

powered by Astah
















# Exemplo: uma arquitetura para a Web





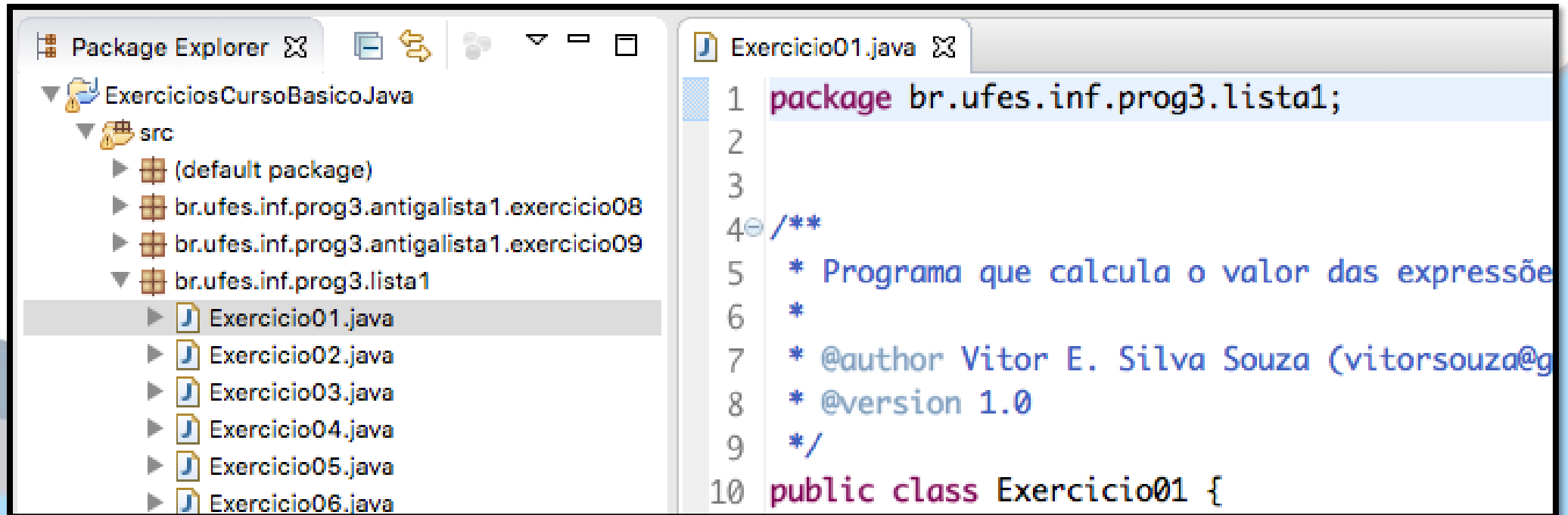
# Exemplo: uma arquitetura para a Web

- ▲  Java Resources
  - ▲  src
    - ▷  br.ufes.inf.nemo.sap
    - ▷  br.ufes.inf.nemo.sap.assignments.application
    - ▷  br.ufes.inf.nemo.sap.assignments.controller
    - ▷  br.ufes.inf.nemo.sap.assignments.domain
    - ▷  br.ufes.inf.nemo.sap.assignments.persistence
    - ▷  br.ufes.inf.nemo.sap.lab.application
    - ▷  br.ufes.inf.nemo.sap.lab.controller
    - ▷  br.ufes.inf.nemo.sap.lab.domain
    - ▷  br.ufes.inf.nemo.sap.lab.domain.persistence
    - ▷  br.ufes.inf.nemo.sap.servlet
    - ▷  META-INF

# Declaração do pacote

- Uso da palavra-chave **package**;
- Primeira linha não comentada da classe:

```
package br.ufes.inf.prog3.lista1;  
  
public class Exercicio01 { /* ... */ }
```



The screenshot shows an IDE interface. On the left, the Package Explorer displays a project named 'ExerciciosCursoBasicoJava' with a 'src' folder containing several packages and files. The package 'br.ufes.inf.prog3.lista1' is expanded, showing files 'Exercicio01.java' through 'Exercicio06.java'. On the right, the code editor shows the content of 'Exercicio01.java'. The first line is 'package br.ufes.inf.prog3.lista1;', which is highlighted. The second line is blank. The third line is blank. The fourth line is a comment '/\*\*'. The fifth line is a comment '\* Programa que calcula o valor das expressões'. The sixth line is a comment '\*'. The seventh line is a comment '\* @author Vitor E. Silva Souza (vitorsouza@)'. The eighth line is a comment '\* @version 1.0'. The ninth line is a comment '\*/'. The tenth line is 'public class Exercicio01 {'.

# Convenção de nomes

- Para não haver **conflito** com absolutamente ninguém, sugere-se usar seu **domínio** na Internet ao contrário:

`http://nemo.inf.ufes.br`



`br.ufes.inf.nemo.sistema1`

`br.ufes.inf.nemo.sistema2`

- Usar apenas letras **minúsculas**;
- Esse **padrão não** se aplica à **API** Java.

# Importação

- Acesso **direto** dentro do **mesmo** pacote:

```
package br.ufes.inf.prog3.lista1.exercicio07;
class Ponto { /* ... */ }
```

```
package br.ufes.inf.prog3.lista1.exercicio07;
class Triangulo {
    private Ponto vertice1;    /* ... */
}
```

```
package br.ufes.inf.prog3.lista1.exercicio07;
public class Exercicio07 {
    public static void main(String[] args) {
        Triangulo triangulo;
        /* ... */
    }
}
```

# Importação

- O mesmo **não** ocorre em pacotes **diferentes**:

```
package br.ufes.inf.prog3.lista1.exercicio07.dominio;  
class Ponto { /* ... */ }
```

```
package br.ufes.inf.prog3.lista1.exercicio07.dominio;  
class Triangulo {  
    private Ponto vertice1; /* ... */  
}
```

```
package br.ufes.inf.prog3.lista1.exercicio07;  
public class Exercicio07 {  
    public static void main(String[] args) {  
        Triangulo triangulo;  
        /* ... */  
    }  
}
```

error: Triangulo cannot be resolved to a type

# Importação

- Resolve-se a questão **importando** a **classe** que encontra-se em **outro pacote**:

```
package br.ufes.inf.prog3.lista1.exercicio07;  
  
import br.ufes.inf.prog3.lista1.exercicio07.dominio.Triangulo;  
  
public class Exercicio07 {  
    public static void main(String[] args) {  
        Triangulo triangulo;  
        /* ... */  
    }  
}
```

Uma IDE ajuda nesta tarefa! Eclipse: "Organize Imports".


```
error: The type br.ufes.inf.prog3.lista1.exercicio07.  
        dominio.Triangulo is not visible
```

# Importação

- A classe **importada**, no entanto, precisa ser **pública**!

```
package br.ufes.inf.prog3.lista1.exercicio07.dominio;  
  
public class Triangulo {  
    private Ponto vertice1;    /* ... */  
}
```

```
package br.ufes.inf.prog3.lista1.exercicio07;  
  
import br.ufes.inf.prog3.lista1.exercicio07.dominio.Triangulo;  
  
public class Exercicio07 {  
    public static void main(String[] args) {  
        Triangulo triangulo;  
        /* ... */  
    }  
}
```



# Importação

- Pode-se **importar** classe por classe ou um **pacote inteiro**:

```

package br.ufes.inf.prog3.lista1.exercicio07;

import br.ufes.inf.prog3.lista1.exercicio07.dominio.*;

public class Exercicio07 {
    public static void main(String[] args) {
        Triangulo triangulo;
        Ponto vertice1;
        /* ... */
    }
}

```



# Importação: e se der conflito?

- Se uma classe precisa **usar** outras **duas classes** de **mesmo nome**, só poderá **importar uma** delas:

```

package com.tables.tablessystem.gui;

import com.tables.tablessystem.guicomponents.Table;

public class ManageTablesWindow {
    public static void main(String[] args) {
        Table productTable;
        com.tables.tablessystem.domain.Table product;
        product = new com.tables.tablessystem.domain.Table();
        /* ... */
    }
}

```

A outra classe deverá ser referida pelo seu nome completo, também conhecido como FQN (Fully Qualified Name).

## Alguns detalhes

- Ordem das declarações num arquivo `.java`:
  - `package [0..1];`
  - `import [0..*];`
  - `class [1..*];`
- Importação de **pacote inteiro** (`import pacote.*`):
  - Não há perda de *desempenho*;
  - Pode haver problema de *conflito de nomes*;
  - Importar *classe por classe* é considerado *boa prática*, pois *facilita a leitura*;
  - "*Organize Imports*" do Eclipse faz assim por *padrão*.

# Uso do pacote `java.lang`

- As **classes** do pacote `java.lang` são importadas automaticamente;
- Não é necessário:
  - `import java.lang.String;`
  - `import java.lang.Math;`
  - `import java.lang.*;`

# Importação estática

- A partir do **Java 5** é possível importar os membros **estáticos** de uma classe:
- Antes:

```
/* ... */  
r = Math.exp(x) + Math.log(y) - Math.sqrt(Math.pow(Math.PI,  
y));
```

- Depois:

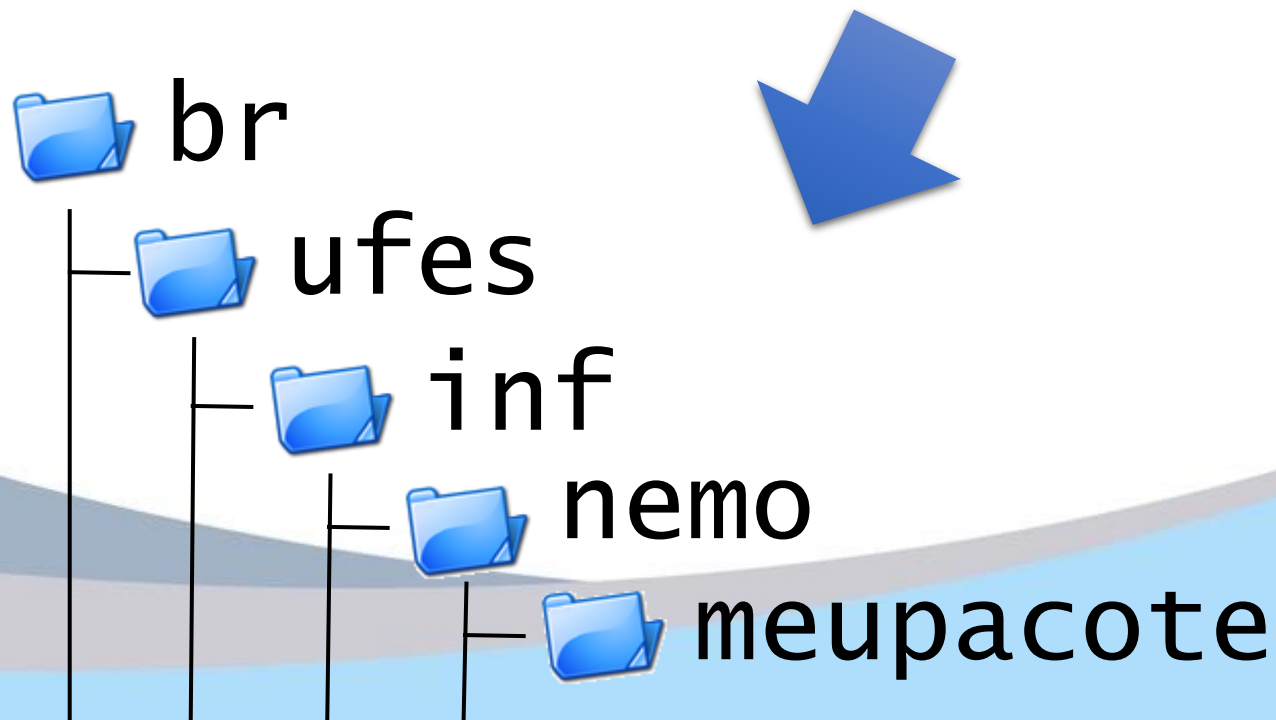
```
import static java.lang.Math.*;  
  
/* ... */  
r = exp(x) + log(y) - sqrt(pow(PI, y));
```

Também pode importar  
somente um específico.

# Localização de pacotes

- A JVM **carrega classes** dos arquivos `.class`;
- Como a JVM **encontra** as classes em **diferentes pacotes?**

`br.ufes.inf.nemo.meupacote`



# Localização de pacotes

```
package br.ufes.inf.nemo.meupacote;
import java.util.Date;
public class MinhaClasse {
    public static void main(String[] args) {
        System.out.println(new Date());
    }
}
```



br



ufes



inf



nemo



meupacote



MinhaClasse.java

# Localização de pacotes

```
$ ls  
br
```

```
$ javac br/ufes/inf/nemo/meupacote/MinhaClasse.java
```



MinhaClasse.java



MinhaClasse.class

# Localização de pacotes

```
$ java -cp . br.ufes.inf.nemo.meupacote.MinhaClasse
```

```
Wed Jun 05 21:01:29 BRT 2013
```

 br

 ufes

 inf

 nemo

 meupacote



MinhaClasse.java



MinhaClasse.class

Inclusão do diretório atual no caminho de classes! (Desnecessário no Java 5+)



# Classpath

- O “**caminho** de classes” ou “**trilha** de classes” é onde as ferramentas do JDK e a JVM **procuram** classes;
  - *A partir dos **diretórios** do classpath procura-se as classes segundo seus **pacotes** (usa a 1ª encontrada).*
- Estão por **padrão** no *classpath*:
  - *A biblioteca de classes da **API** Java SE;*
  - *O diretório **atual**.*
- O *classpath* pode ser **alterado**:
  - *Variável de **ambiente** (não recomendado);*
  - ***Opção** `-classpath` ou `-cp`.*

# Compilação automática

- Ao **compilar** uma classe, se ela faz referência a outra que não foi compilada, esta última é **compilada** se o código está **disponível**;
- Se já foi **compilada**, mas o arquivo fonte está com data mais **recente**, ela é recompilada.
- Uso de **IDEs**:
  - *Utilizar uma IDE **abstrai** todas estas preocupações;*
  - *A IDE cuida de todo o **processo** de compilação.*

# O pacote padrão

- Toda classe que **não** especifica o pacote pertence ao **pacote padrão**;
- Seu `.class` deve estar numa **pasta raiz** do *classpath*.

```
public class Bo1o {
    public static void main(String[] args) {
        // Não há import, estão no mesmo pacote.
        Torta t = new Torta();
        t.f();
    }
}

class Torta {
    void f() { System.out.println("Torta.f()"); }
}
```

# Especificadores de Acesso



# Membros públicos

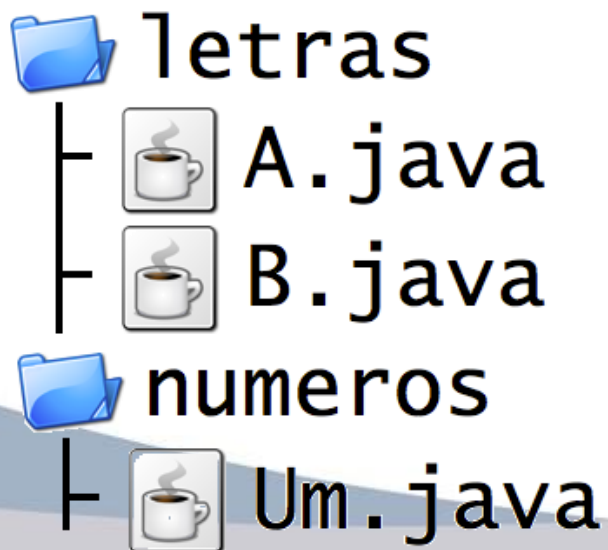
Membro	Resultado
Classes	Classes públicas* podem ser importadas por qualquer classe.
Atributos	Atributos públicos podem ser lidos e alterados por qualquer classe.
Métodos	Métodos públicos podem ser chamados por qualquer classe.

\* Só pode haver uma classe pública por arquivo-fonte e os nomes (da classe e do arquivo) devem ser iguais.

# Membros públicos

```
package letras;
public class A {
    public int x = 10;
    public void print() {
        System.out.println(x);
    }
}
```

```
package numeros;
import letras.B;
public class Um {
    B b = new B();
    public void g() {
        b.f();
    }
}
```



```
package letras;
public class B {
    public A a = new A();
    public void f() {
        a.x = 15;
        a.print();
    }
}
```

# Finalmente, PSVM!

- O método `main()` é:
  - *public*, pois deve ser chamado pela *JVM*;
  - *static*, pois pertence à *classe* como um todo (a *JVM* não instancia um objeto para chamá-lo);
  - *void*, pois não retorna *nada*.
- A *classe* que possui o método `main()` deve ser:
  - *public*, pois deve ser acessível pela *JVM*.

# JavaDoc

- **Comentários** são ignorados pelo compilador;
  - Usados pelo *programador* para melhorar a *legibilidade* do código;
  - Comentários de uma *linha*: `// ...;`
  - Comentários de *múltiplas linhas*: `/* ... */;`
- Um **tipo**, porém, é especial:
  - Comentários *JavaDoc*: `/** ... */` – utilizados pela ferramenta *javadoc* para criar uma *documentação HTML* das classes, atributos e métodos.
  - A *ferramenta javadoc* vem com o *JDK*;
  - Mais informações na *apostila da Caelum*.



# JavaDoc: exemplo

```

/** <i>Documentação da classe</i>.
 * @author Fulano da Silva
 * @see java.io.File
 */
public class FileData extends File {
    /** Documentação de atributo. */
    private double tamanho;

    /* Comentário
     de múltiplas linhas. */

    /** Documentação de método. */
    public void excluir() {
        int x = 1; // Comentário de uma linha.
    }
}

```

No Eclipse: Project > Generate Javadoc...



UNIVERSIDADE FEDERAL  
DO ESPÍRITO SANTO

# A documentação da API do Java

The screenshot shows a web browser displaying the Java API documentation for the `String` class. The browser's address bar shows the file path `file:///Applications/dev/java-se-8-api/api/index.html`. The page title is "Java™ Platform Standard Ed. 8". The navigation menu includes "OVERVIEW", "PACKAGE", "CLASS" (highlighted), "USE TREE", "DEPRECATED", "INDEX", and "HELP". Below the navigation menu, there are links for "PREV CLASS", "NEXT CLASS", "FRAMES", and "NO FRAMES". The main content area shows the class name "String" and its inheritance hierarchy: `java.lang.Object` and `java.lang.String`. It also lists "All Implemented Interfaces": `Serializable`, `CharSequence`, and `Comparable<String>`. The class signature is `public final class String`, which extends `Object` and implements `Serializable`, `Comparable<String>`, and `CharSequence`. A paragraph explains that the `String` class represents character strings and that all string literals in Java programs are implemented as instances of this class. Another paragraph states that strings are constant and their values cannot be changed after they are created. A code example shows `String str = "abc";` and notes that it is equivalent to `char data[] = {'a', 'b', 'c'};` and `String str = new String(data);`. The page also includes a list of classes in the left sidebar and a section for "Enums" at the bottom.

# Datas

- Em Java, existem duas classes para manipulação de datas: `Date` e `Calendar` (`java.util`);
- `java.util.Date`:
  - *Representa um instante do tempo com precisão de milissegundos como um número longo (ms passados desde 01/01/1970 00:00:00);*
  - *`new Date()` representa o instante atual, existe um construtor `new Date(long)`;*
  - *Métodos `before()` e `after()` comparam datas e retornam valores booleanos;*
  - *`getTime()` e `setTime(long)` obtém e alteram o valor interno da data. Partes da data podem ser acessados usando `getDay()`, `getMonth()`, etc.*

# Datas

- `java.util.Calendar`:
  - *`Calendar.getInstance()` obtém um calendário;*
  - *Um calendário funciona com campos: `YEAR`, `MONTH`, `DAY_OF_MONTH`, `DAY_OF_WEEK`, `HOURL`, etc.*
  - *`set(int, int)` atribui um valor a um campo; O primeiro `int` é o campo a ser alterado.*
  - *`get(int)` obtém o valor de um campo;*
  - *`add(int, int)` adiciona um valor a um campo;*
  - *`getTime()` e `setTime(Date)` alteram a data do calendário.*

Calendários já calculam anos bissextos, trocas de hora, dia, mês, etc. Use-o sempre para manipular datas!

```
static void exemploCalendar() {  
    Calendar hoje = Calendar.getInstance();  
  
    // Como recuperar campos de data  
    System.out.println("\nData de hoje: " + hoje);  
    System.out.println("\nDia do mes: " + hoje.get(Calendar.DAY_OF_MONTH));  
    System.out.println("Mes: " + hoje.get(Calendar.MONTH));  
    System.out.println("Ano: " + hoje.get(Calendar.YEAR));
```

```
    // Como atualizar campos de data  
    Calendar outra_data = Calendar.getInstance();  
    outra_data.set(Calendar.DAY_OF_MONTH, 30);  
    outra_data.set(Calendar.MONTH, 12);  
    outra_data.set(Calendar.YEAR, 2000);  
    System.out.println("\nOutra data: " + outra_data);
```

Outra data:

```
java.util.GregorianCalendar[time=?,areFieldsSet=false,areAllFieldsSet=true, lenient=true,zone=sun.util.calendar.ZoneInfo[id="America/Sao_Paulo",offset=-10800000,dstSavings=0,useDaylight=false,transitions=93,lastRule=null],firstDayOfWeek=1,minimalDaysInFirstWeek=1,ERA=1,YEAR=2000,MONTH=12,WEEK_OF_YEAR=28,WEEK_OF_MONTH=2,DAY_OF_MONTH=30,DAY_OF_YEAR=187,DAY_OF_WEEK=4,DAY_OF_WEEK_IN_MONTH=1,AM_PM=1,HOUR=4,HOUR_OF_DAY=16,MINUTE=9,SECOND=55,MILLISECOND=105,ZONE_OFFSET=-10800000,DST_OFFSET=0]
```

```
    // Somando dias a uma data (e.g., previsão de entrega de um produto)  
    Calendar entrega = Calendar.getInstance();  
    entrega.add(Calendar.DATE, 7);  
    System.out.println("\n7 dias no futuro: " + entrega);
```

```
}
```

# Datas

```
import java.util.*;
import static java.util.Calendar.*;

public class Teste {
    public static void main(String[] args) {
        Calendar cal = Calendar.getInstance();
        cal.set(YEAR, 1981);
        cal.set(MONTH, JUNE);
        cal.set(DAY_OF_MONTH, 15);

        String[] dias = {"", "Dom", "Seg", "Ter",
                        "Qua", "Qui", "Sex", "Sab"};

        int diasem = cal.get(DAY_OF_WEEK);
        System.out.println(dias[diasem]);
    }
}
```

# Datas

```
// Dentro do main()
// importando java.util.* e java.util.Calendar.*
Calendar cal = Calendar.getInstance();

// Thu Jul 13 22:45:39 BRT 2006
cal.setTime(new Date());
System.out.println(cal.getTime());

// Wed Feb 13 22:45:39 BRST 2008
cal.add(YEAR, 2);
cal.set(MONTH, FEBRUARY);
System.out.println(cal.getTime());

// Sat Mar 01 22:46:19 BRT 2008
cal.add(DAY_OF_MONTH, 17);
System.out.println(cal.getTime());
```

# Nova API de datas/horas

- Em Java 1.0, existia só a classe Date;
- Como era complicada de manipular, Java 1.1 adicionou Calendar;
- Calendar não agradava muitos programadores. Surgiu a biblioteca JodaTime ([joda.org/joda-time/](http://joda.org/joda-time/));
- Com base em JodaTime, uma nova API de datas/horas vem sendo trabalhada para Java desde 2007;
- Finalmente esta API foi incorporada à API Java, no pacote java.time.



# Nova API de datas/horas: exemplo (1)

- Datas para computadores: representação interna continua igual.

```
// 2014-06-26T22:16:30.175Z (formato ISO-8601)
Instant agora = Instant.now();
System.out.println(agora);

// Duração (ms): 5
Instant inicio = Instant.now();
for (int i = 0; i < Integer.MAX_VALUE; i++);
Instant fim = Instant.now();
Duration duracao = Duration.between(inicio, fim);
Long duracaoEmMilissegundos = duracao.toMillis();
System.out.println("Duração (ms): " + duracaoEmMilissegundos);

// Porque não era necessário mudar a representação:
// Sun Aug 17 04:12:55 BRT 292278994
System.out.println(new Date(Long.MAX_VALUE));
```

## Nova API de datas/horas: exemplo (2)

- Datas para seres humanos: substitui a classe Calendar;
- Dá suporte a diferentes calendários.

```
LocalDate hoje = LocalDate.now();  
System.out.println(hoje); // 2014-06-26 (formato ISO-  
8601)  
  
LocalDate lancamentoJava8 = LocalDate.of(2014, 3, 18);  
// Ou: lancJava8 = LocalDate.of(2014, Month.MARCH, 18);  
  
System.out.println(lancamentoJava8); // 2014-03-18
```

## Nova API de datas/horas: exemplo (3)

- Datas para seres humanos: substitui a classe Calendar;
- Dá suporte a diferentes calendários.

```
// 3 meses e 8 dias pra eu falar de Java 8
Period periodo = Period.between(lancamentoJava8, hoje);
System.out.printf("%s meses e %s dias pra eu falar de
Java 8%n", periodo.getMonths(), periodo.getDays());
```

```
LocalTime horarioDeEntrada = LocalTime.of(9, 0);
System.out.println(horarioDeEntrada); // 09:00
```

```
LocalDateTime aberturaDaCopa = LocalDateTime.of(2018,
Month.JUNE, 14, 11, 30);
System.out.println(aberturaDaCopa);
// 2018-06-14T11:30
```

# Difícil manter-se atualizado...

- Novidades do Java 9: <http://blog.caelum.com.br/o-minimo-que-voce-deve-saber-de-java-9/>
  - *JShell, Reactive Streams, Jigsaw (JPMS), novas APIs;*
- Novidades do Java 10: <http://blog.caelum.com.br/o-minimo-que-voce-deve-saber-de-java-10/>
  - *Novo versionamento, inferência de tipos (var).*
- Java 11, 12, 13, 14, 15, ...  
<https://dzone.com/articles/a-guide-to-java-versions-and-features>