

# Strings, Datas e Números

Instituto Metrópole Digital

Disciplina: IMD0040 - Linguagem de Programação II

Docente: Emerson Alencar



# Strings

- Armazenam conjuntos de caracteres
- As strings são objetos (não são tipos primitivos), logo podem ser construídas como qualquer outro objeto

```
String s = new String();
```

String vazia

```
String s = new String("abc");
```

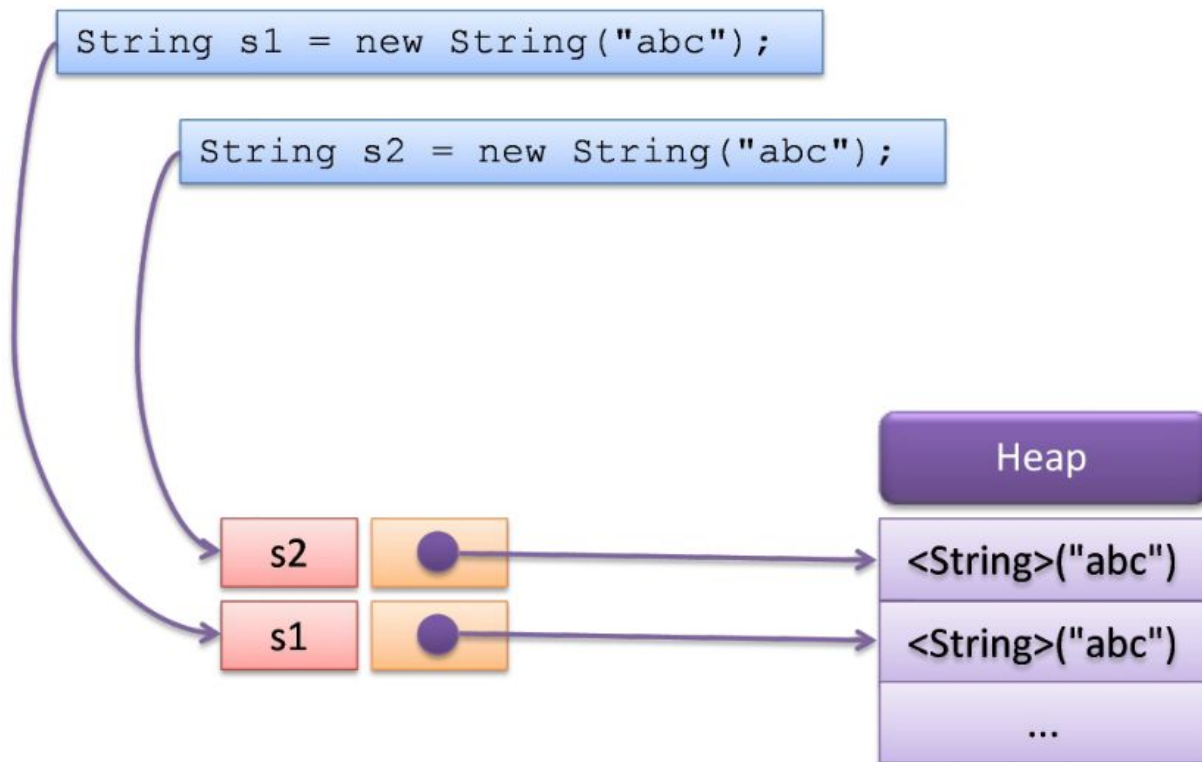
String "abc"

```
String s = "abc";
```

String "abc"

Qual a diferença?

# Strings e a Memória

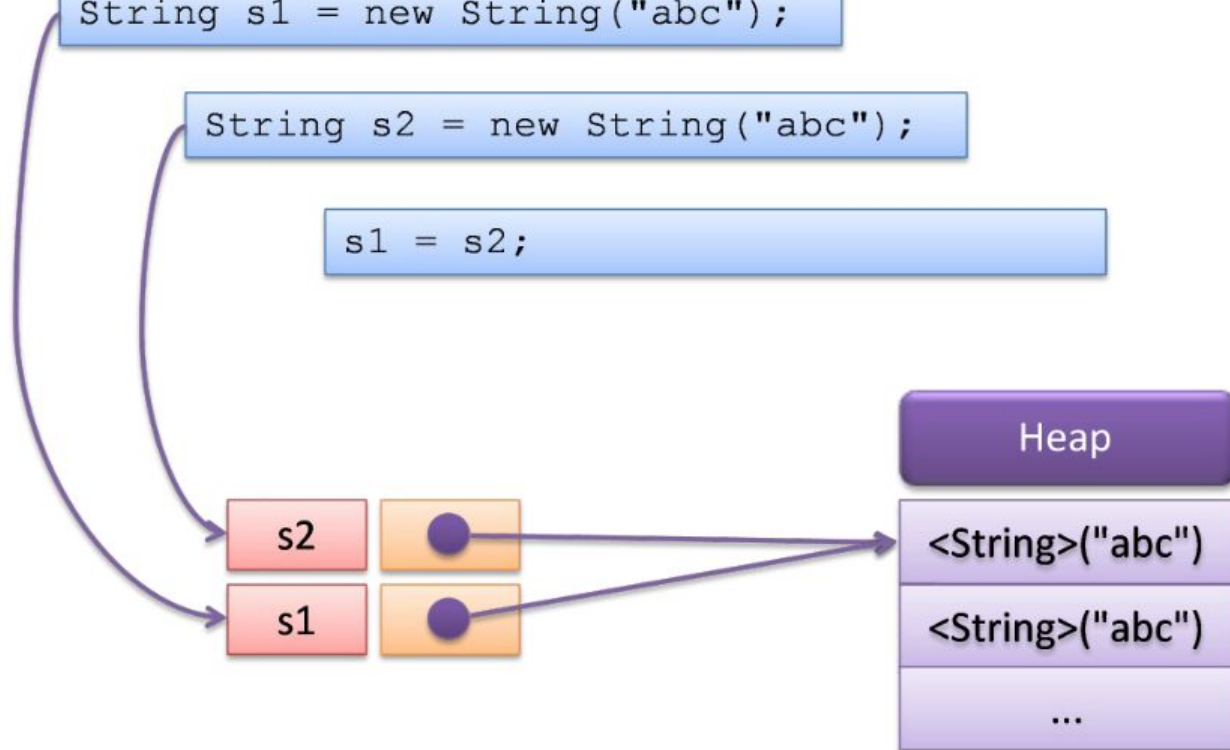


# Strings e a Memória

```
String s1 = new String("abc");
```

```
String s2 = new String("abc");
```

```
s1 = s2;
```



# Strings e a Memória

A JVM cria a string no heap  
e a adiciona ao pool

```
String s1 = "abc";
```

JVM

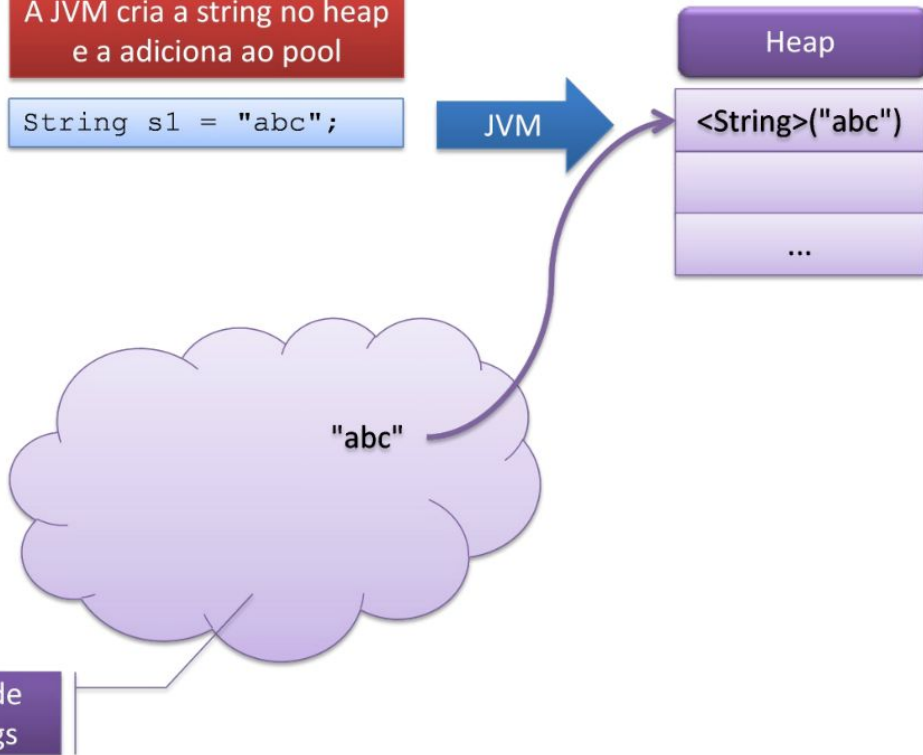
Heap

<String>("abc")

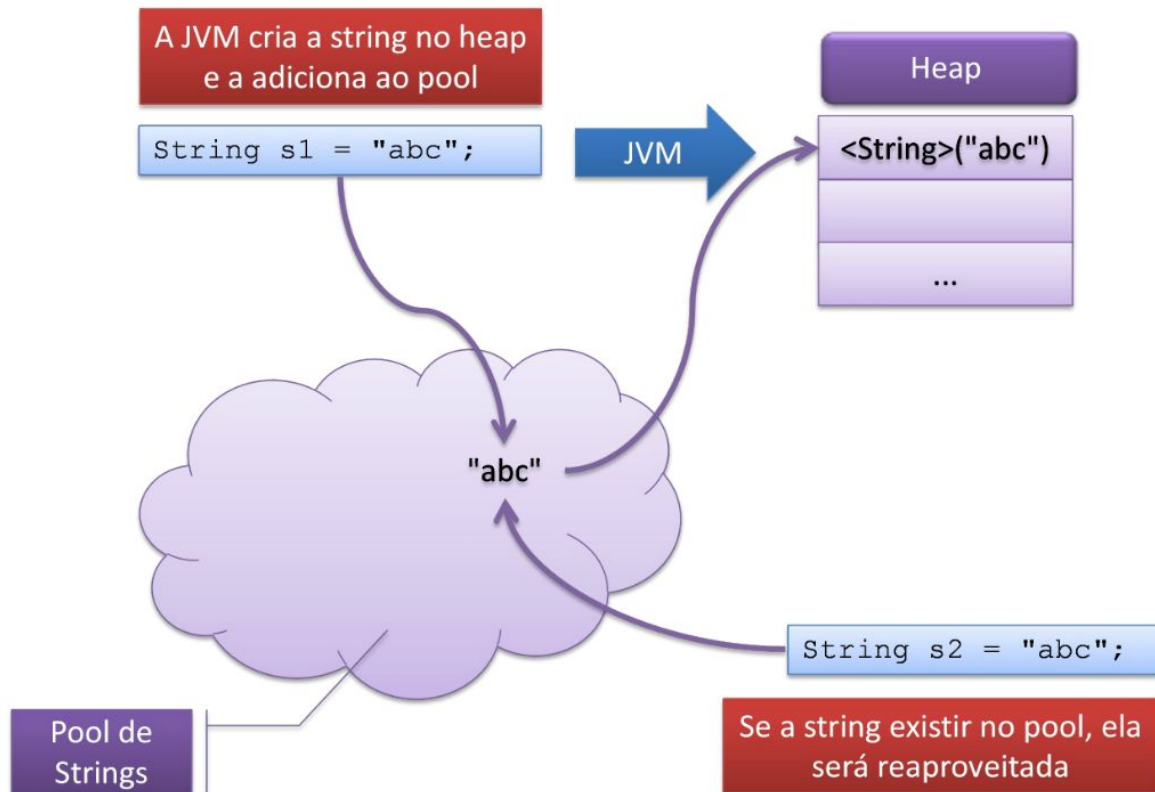
...

"abc"

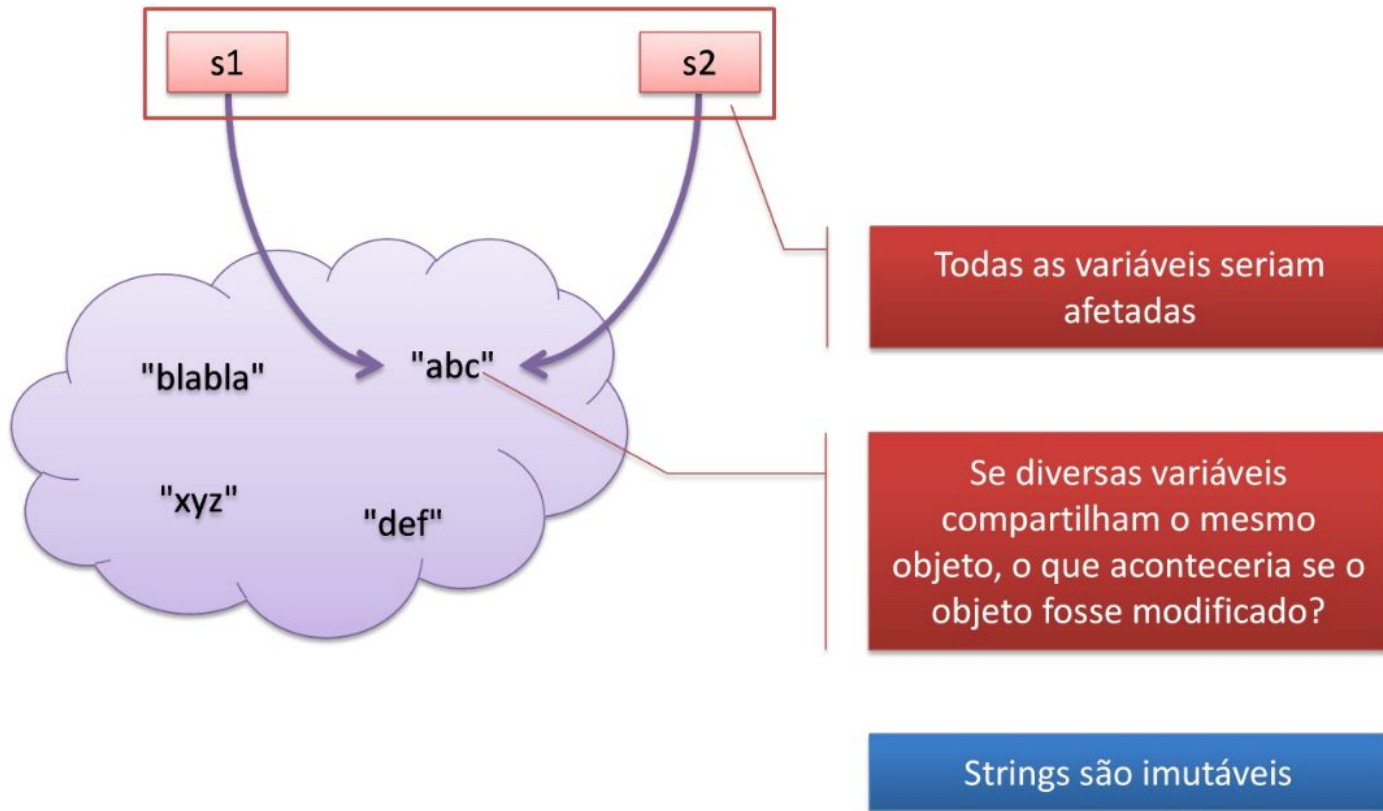
Pool de  
Strings



# Strings e a Memória



# Strings são Imutáveis



# Strings São Imutáveis

- Depois de criada, uma string nunca tem seu valor alterado

```
String s = "abc";  
s.toUpperCase();
```

**s** continua com o valor "abc"

```
String s = "abc";  
s = s.toUpperCase();
```

**s** deixa de ser "abc" e referencia uma nova string, "ABC"

```
String s = "abc";  
s.concat("def");
```

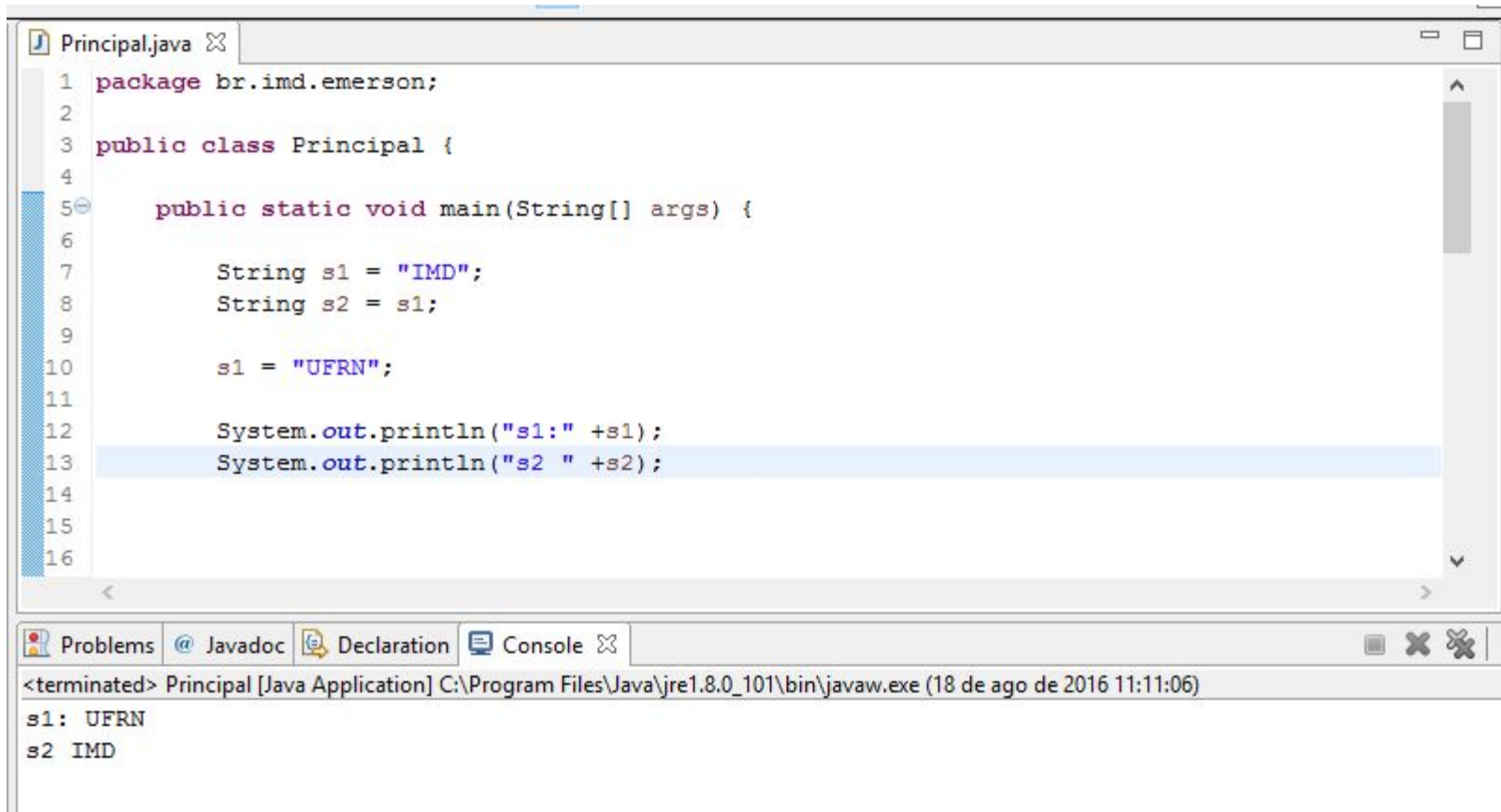
**s** continua com o valor "abc"

```
String s = "abc";  
s = s.concat("def");
```

**s** deixa de ser "abc" e referencia uma nova string, "abcdef"



# Strings São Imutáveis



```
Principal.java
1 package br.imd.emerson;
2
3 public class Principal {
4
5     public static void main(String[] args) {
6
7         String s1 = "IMD";
8         String s2 = s1;
9
10        s1 = "UFRN";
11
12        System.out.println("s1:" + s1);
13        System.out.println("s2 " + s2);
14
15
16    }
```

Problems | Javadoc | Declaration | Console

<terminated> Principal [Java Application] C:\Program Files\Java\jre1.8.0\_101\bin\javaw.exe (18 de ago de 2016 11:11:06)

s1: UFRN  
s2 IMD

# Trabalhando com Strings

- O operador “+” pode ser utilizado na concatenação de strings

```
String s1 = "abc";  
String s2 = "123" + s1;
```

**s2** passa a ter o valor "123abc"

- Para comparar strings, o método equals deve ser utilizado

```
if (s1.equals(s2)) {  
    ...  
}
```

O *equals()* compara o conteúdo ao invés de comparar endereços de memória

# Métodos da Classe String

Método	Descrição
charAt(int)	Retorna o caractere de uma posição
indexOf(String)	Retorna a posição em que uma string aparece pela primeira vez na string principal
length()	Retorna o tamanho da string
split(String)	Divide a string de acordo com um critério
substring(int, int)	Retorna uma parte da string
toLowerCase()	Converte os caracteres para minúsculo
toUpperCase()	Converte os caracteres para maiúsculo

# StringBuilder

- Como strings são imutáveis, manipular a mesma string diversas vezes pode ocupar muita memória desnecessariamente
  - Bastante comum em concatenação de strings dentro de um loop
- A classe StringBuilder resolve este problema

# Usando a Classe StringBuilder

```
StringBuilder sb = new StringBuilder("abc");  
sb.append("def");  
sb.append("ghi");  
sb.append("jkl");  
String s = sb.toString();
```

O objeto é instanciado com o valor inicial "abc"

Outras strings vão sendo concatenadas

É gerada uma string que contém todas as modificações feitas no objeto sb



"abcdefghijkl"

# Método da Classe StringBuilder

Método	Descrição
append(String)	Concatena uma string
delete(int, int)	Remove parte de uma string
insert(int, String)	Insere uma string em uma determinada posição
reverse()	Inverte os caracteres
toString()	Retorna o conteúdo do objeto como uma string

# Formatando String

- A formatação de strings pode ser feita facilmente através dos métodos `format()` e `printf` da classe `PrintStream`
  - `System.out` é um `PrintStream`, portanto é possível formatar a saída para o console
- A classe `String` também possui o método `format()`



# Formatando Strings

```
System.out.printf("%d, %f", 245, 100.0);
```



245, 100,000000

```
System.out.printf("%.2f", 100.0);
```



100,00

```
System.out.printf(">%7d<\n>%7s<", 2000, "abc");
```



> 2000<  
> abc<

```
System.out.printf("%05d", 25);
```



00025



# Formatando Números

- Java possui a classe `NumberFormat`, utilizada para formatar números
- Possui suporte à localização

# Exemplo de Formatação Numérica

- Formatação do número, considerando separadores de milhar e casas decimais

```
NumberFormat nf = NumberFormat.getInstance();  
String s = nf.format(1000.5);  
System.out.println(s);
```

import java.text.NumberFormat;



1.000,5

- Agora no padrão americano

```
Locale l = new Locale("en", "US");  
NumberFormat nf = NumberFormat.getInstance(l);  
String s = nf.format(1000.5);  
System.out.println(s);
```

import java.text.NumberFormat;



1,000.5

# Exemplos de Formatação de Moeda

- Formatação de moeda no padrão brasileiro

```
Locale l = new Locale("pt", "BR");  
NumberFormat nf = NumberFormat.getCurrencyInstance(l);  
String s = nf.format(1000.5);  
System.out.println(s);
```



R\$ 1.000,50

- Agora no padrão italiano

```
Locale l = new Locale("it", "IT");  
NumberFormat nf = NumberFormat.getCurrencyInstance(l);  
String s = nf.format(1000.5);  
System.out.println(s);
```



€ 1.000,50

# Trabalhando com Datas

- Java possui quatro classes principais para trabalhar com datas

<i>Classe</i>	<i>Descrição</i>
<i>java.util.Date</i>	Representa uma data e hora.
<i>java.util.Calendar</i>	Possibilita a conversão e manipulação de datas e horas.
<i>java.text.DateFormat</i>	Formata datas e horas.
<i>java.util.Locale</i>	Representa uma localidade. É utilizada com datas para formatá-las de acordo com a localidade desejada.

- Uma nova API de datas e horas foi adicionada a partir do java 8

# Exemplos no Uso de Datas

- Obter a data/hora atual

```
Date d = new Date();  
System.out.println(d.toString());
```



Thu Oct 29 18:58:02 BRST 2020

- Somar 7 dias à data atual

```
Calendar c = Calendar.getInstance();  
c.add(Calendar.DAY_OF_MONTH, 7);  
Date d = c.getTime();  
System.out.println(d.toString());
```



Thu Nov 05 18:58:02 BRST 2020

# Exemplos de Formatação de Datas

- Formatação da data atual no padrão curto

```
Date d = new Date();  
DateFormat df = DateFormat.getDateInstance(DateFormat.SHORT);  
String s = df.format(d);  
System.out.println(s);
```



29/10/2020

```
import java.text.DateFormat;  
import java.util.Date;
```

# Exemplos de Formatação de Datas

- Formatação da data atual no padrão longo, de acordo com o francês falado na França

```
Date d = new Date();  
Locale l = new Locale("fr", "FR");  
DateFormat df = DateFormat.getDateInstance(DateFormat.LONG, l);  
String s = df.format(d);  
System.out.println(s);
```



29 octobre 2020

Para formatar a hora,  
use o **getTimeInstance()**

# API de Data e Hora: Pacote java.time

- A partir do Java 8 a linguagem conta com uma nova API para manipulação de datas e horas
- Características
  - Diversas classes para representar diferentes conceitos
  - Classes imutáveis, o que as torna thread-safe, isso significa que uma vez criado os objetos você não pode alterar esse objeto, para alterar que criar um novo objeto.



# Principais Elementos

Nome da Classe	O que representa
<i>LocalDate</i>	Uma data (com dia, mês e ano)
<i>LocalTime</i>	Uma hora (com hora, minuto, segundo e milissegundo)
<i>LocalDateTime</i>	Uma data e hora
<i>Period</i>	Um período de tempo (em anos, meses, dias, semanas)
<i>Duration</i>	Uma duração de tempo (em dias, horas, minutos, segundos)
<i>MonthDay</i>	Um par de mês e dia (Ex: dia de aniversário)
<i>YearMonth</i>	Um par de ano e mês (Ex: data de validade do cartão de crédito)
<i>Instant</i>	Um instante no tempo, com precisão de nanossegundos

Nome do Enum	O que representa
<i>ChronoUnit</i>	Unidades de tempo (dias, meses, anos, horas, minutos, etc.)

# Classes LocalDate e LocalTime

- Data/Hora atual do Sistema

```
LocalDate d = LocalDate.now();
```

```
LocalTime t = LocalTime.now();
```

- Data/Hora juntando as partes

```
LocalDate d = LocalDate.of(2020, Month.DECEMBER, 10);
```

```
LocalTime t = LocalTime.of(13, 45, 0);
```

- Data/Hora através de parse

```
LocalDate d = LocalDate.parse("04/03/2020",  
    DateTimeFormatter.ofPattern("dd/MM/yyyy"));
```

```
LocalTime t = LocalTime.parse("16:00",  
    DateTimeFormatter.ofPattern("HH:mm"));
```

# Operações com Datat

- LocalDate

```
LocalDate d = LocalDate.now();  
LocalDate d1 = d.plusDays(5);  
LocalDate d2 = d.minus(1, ChronoUnit.WEEKS);
```

- LocalTime

```
LocalTime t = LocalTime.now();  
LocalTime t2 = t.plusHours(2).plusMinutes(30);  
LocalTime t3 = t.minus(100, ChronoUnit.MILLIS);
```

- LocalDateTime

```
LocalDateTime d = LocalDateTime.now();  
LocalDateTime d2 = d.plusDays(2).plusHours(30);
```

# Classes Period e Duration

- Período/duração juntando as parte

```
Period p = Period.of(0, 1, 7);  
LocalDate d = LocalDate.now().plus(p);
```

0 ano 1 mês e 7 dias

```
Duration d = Duration.ofMinutes(15);  
LocalTime t = LocalTime.now().minus(d);
```

15 minutos

- Período/duração entre datas/horas

```
LocalDate d1 = LocalDate.now();  
LocalDate d2 = LocalDate.parse("2000-01-05");
```

```
Period p = Period.between(d2, d1);  
int years = p.getYears();  
int months = p.getMonths();  
int days = p.getDays();
```

```
LocalTime t1 = LocalTime.now();  
LocalTime t2 = LocalTime.parse("04:30:00");
```

```
Duration d = Duration.between(t2, t1);  
long seconds = d.getSeconds();
```

# Enum ChronoUnit

- Intervalo em meses

```
LocalDate d1 = LocalDate.of(2000, Month.JANUARY, 1);  
LocalDate d2 = LocalDate.of(2100, Month.DECEMBER, 31);  
long months = ChronoUnit.MONTHS.between(d1, d2);
```

- Intervalo em nanossegundos

```
LocalTime t1 = LocalTime.of(8, 0);  
LocalTime t2 = LocalTime.now();  
long nanos = ChronoUnit.NANOS.between(t1, t2);
```

- Intervalo em horas

```
LocalTime t1 = LocalTime.of(8, 0);  
LocalTime t2 = LocalTime.now();  
long nanos = ChronoUnit.HOURS.between(t1, t2);
```

# Classe Instant

- Tempo de execução

```
Instant start = Instant.now();  
  
//...  
  
Instant end = Instant.now();  
  
Duration d = Duration.between(start, end);  
long seconds = d.getSeconds();
```

**Por hoje é só...**