Fundamentos de Java

Strings, Datas e Números

Softblue
cursos online

Tópicos Abordados

- Strings
  - Strings na memória
  - Métodos importantes da classes *String*
- *StringBuilder*
- Formatando strings
- Trabalhando com datas
  - Manipulação de datas
  - Date and Time API (pacote *java.time*)
- Formatando números
  - Formatação de moeda
- Números randômicos
- Métodos importantes da classe *Math*

Strings

- Armazenam conjuntos de caracteres
- As strings são objetos, logo podem ser construídas como qualquer outro objeto

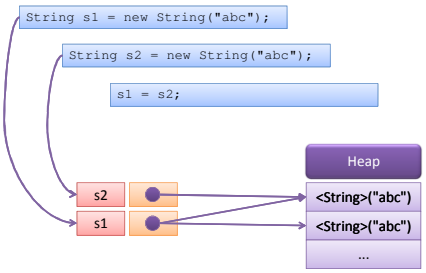
```
String s = new String();
```
String vazia

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

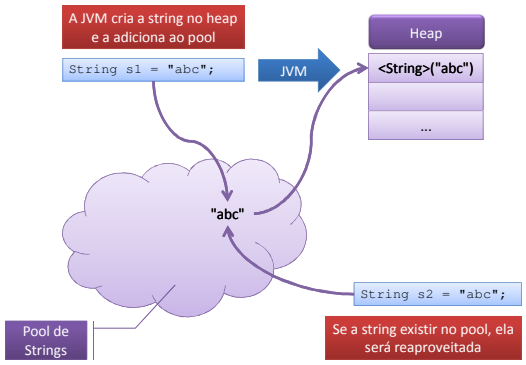
```
String s = "abc";
```
String "abc"

Qual a diferença?

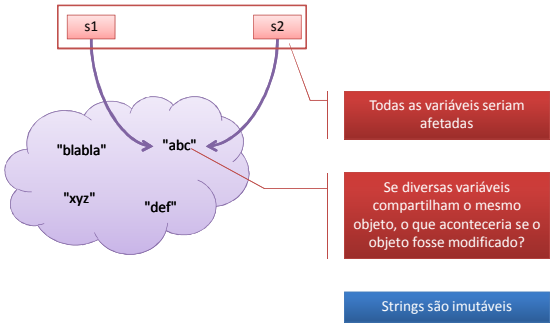
Strings e a Memória



Strings e a Memória



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"

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

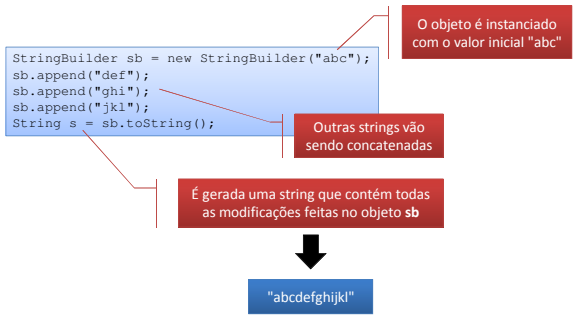
Métodos da Classe *String*

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
  - Existe também a classe *StringBuffer*, que tem exatamente a mesma função mas que possui todos os seus métodos sincronizados

Usando a Classe *StringBuilder*



Métodos 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 Strings

- 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()*

Sintaxe → `printf("<string>", <argumentos>)`

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);
```

```
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);
```

```
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

Classe	Descrição
java.util.Date	Representa uma data e hora.
java.util.Calendar	Possibilita a conversão e manipulação de datas e horas.
java.text.DateFormat	Formata datas e horas.
java.util.Locale	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
```

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

Principais Elementos

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

Nome do Enum	O que representa
ChronoUnit	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 Datas

- *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 *Durarion*

- Período/duração juntando as partes

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

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

- Período/duração entre datas 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();
```

Integração com Código Legado

- As classes *Date* e *Calendar* ganharam métodos para converter a sua representação para um objeto *Instant*

Date -> LocalDateTime

```
Date date = new Date();
Instant instant = date.toInstant();
LocalDateTime ldt = LocalDateTime.from(instant);
```

LocalDateTime -> Date

```
LocalDateTime ldt = LocalDateTime.now();
Instant instant = ldt.atZone(ZoneId.systemDefault()).toInstant();
Date date = Date.from(instant);
```

Integração com Código Legado

Calendar -> LocalDateTime

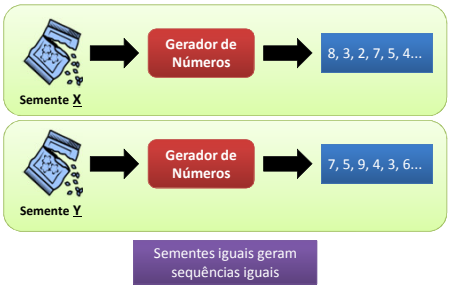
```
Calendar calendar = Calendar.getInstance();
Instant instant = calendar.toInstant();
LocalDateTime ldt = LocalDateTime.from(instant);
```

LocalDateTime -> Calendar

```
LocalDateTime ldt = LocalDateTime.now();
Instant instant = ldt.atZone(ZoneId.systemDefault()).toInstant();
Calendar calendar = Calendar.getInstance();
calendar.setTime(Date.from(instant));
```

Números Randômicos

- Java é capaz de gerar números randômicos, que na verdade são pseudo-randômicos



A Classe *Random*

- A classe *Random* pode ser utilizada para gerar números randômicos

```
Random r = new Random();
```
Cria um gerador de números randômicos. É possível especificar uma semente

```
double n = r.nextDouble();
```
Gera um novo número *double*. É possível gerar números de outros tipos

```
int n = r.nextInt(10);
```
Gera um novo *int* entre 0 e 9

O Método *Math.random()*

- Outra opção é utilizar o método *Math.random()*
  - Gera números do tipo *double*
  - Os números são distribuídos entre 0 e 0,99999...

```
double n = Math.random();
```
Gera o próximo *double* da sequência. Utiliza internamente a classe *Random*

O Método *Math.random()*

- Como implementar um método que define um intervalo de geração de números?

```
int gerarInt(int inicio, int fim) {
  int intervalo = fim - inicio;
  int n = (int)(Math.random() * intervalo) + inicio;
  return n;
}
```

Algoritmo

1. Calcula o intervalo entre os números
2. Gera um *double* randômico entre 0 e 0,9999...
3. Multiplica o valor pelo intervalo
4. Trunca as casas decimais
5. Soma o resultado com o início do intervalo

A Classe *Math*

- Possui uma série de métodos estáticos voltados para operações matemáticas comuns

Método	Descrição
abs()	Valor absoluto (sem sinal)
max()	Valor máximo entre dois valores
min()	Valor mínimo entre dois valores
ceil()	Arredonda um valor para cima
floor()	Arredonda um valor para baixo
round()	Arredonda um valor para cima ou para baixo
sqrt()	Raiz quadrada
pow()	Potenciação
toDegrees()	Ângulo de radianos para graus
toRadians()	Ângulo de graus para radianos

Softblue