



# Trabalhando com String

## A Classe String

No Java, String é um objeto representado como uma sequência de caracteres. Manipular “cadeias” de caracteres é um aspecto fundamental da maioria das linguagens de programação. A implementação de strings como objetos internos permite que o Java forneça um complemento completo de recursos que tornam conveniente o manuseio de strings. Por exemplo, Java possui métodos para comparar duas cadeias, procurar uma subsequência, concatenar duas cadeias e alterar o caso das letras em uma cadeia. A classe String possui 11 construtores e mais de 40 métodos utilitários. Objetos de sequência são meios imutáveis. Após a criação de um objeto, não é possível alterar os caracteres que compõem essa sequência. Isso não é uma limitação da classe String, mas será muito útil em um aplicativo multithread. Você pode executar toda a operação em seu objeto String, apenas as coisas alteradas, se um novo objeto String for criado a cada vez. Essa abordagem é usada porque cadeias fixas e imutáveis podem ser implementadas com mais eficiência do que as variáveis.

## Criando um objeto String

Em Java, strings são objetos. Assim como outros objetos, você pode criar uma instância de uma String com a nova palavra-chave, da seguinte maneira:

### Passo 1

```
String nome = new String ("Abel");  
String nome = "Abel";
```

Essa linha de código cria um objeto da classe String e o atribui ao nome da variável de referência. Existem algumas diferenças vitais entre essas opções que discutiremos mais adiante, mas o que elas têm em comum é que ambas criam um novo objeto String, com o valor “Abel”, e o atribuem a um nome de variável de referência. Agora, digamos que você queira uma segunda referência ao objeto String referido por s, e estamos chamando o método concat no nome do

objeto, isso criará um novo objeto String com o valor “Abel”, enquanto a referência s2 ainda aponta para o objeto “Abel”.



### Passo 2

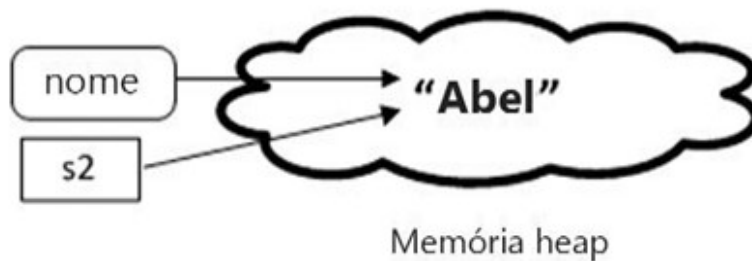
```
String s2 = nome; // refere-se s2 à mesma String que nome
```

### Passo 3

```
nome = nome.concat (" Cardoso"); // o método concat () 'acrescenta'  
um literal ao final
```

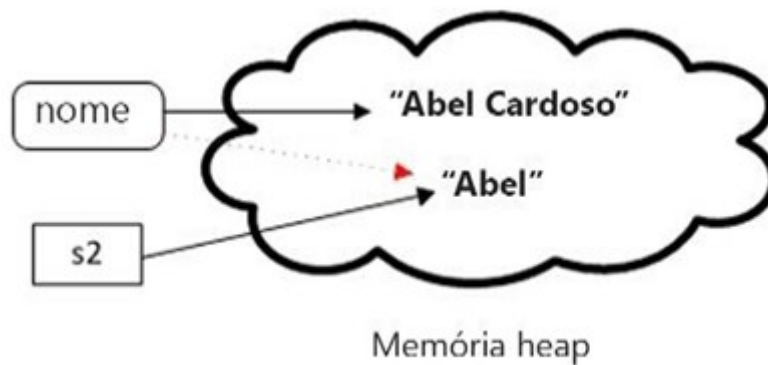
Vamos entender esse conceito graficamente,

#### Etapa 1 e Etapa 2



Fonte: Autoral

#### Etapa 3



Fonte: Autoral

Como sabemos que as cadeias nos objetos do tipo String são imutáveis, significa que o conteúdo da instância da String não pode ser alterado após a criação. No entanto, uma variável declarada como referência de String pode ser alterada para apontar para outro objeto String a qualquer momento.



### Comparando Referência de Objetos de String

Como vimos anteriormente, o objeto String pode ser criado usando o construtor, usando uma nova palavra-chave, assim como podemos usar literais String. Há uma diferença fundamental entre os dois modos de criação de String. Vamos entender isso com a ajuda do programa Java. Estamos criando dois objetos String usando literais e dois objetos chamando o construtor da classe String. Estamos comparando objetos String usando o operador ==.

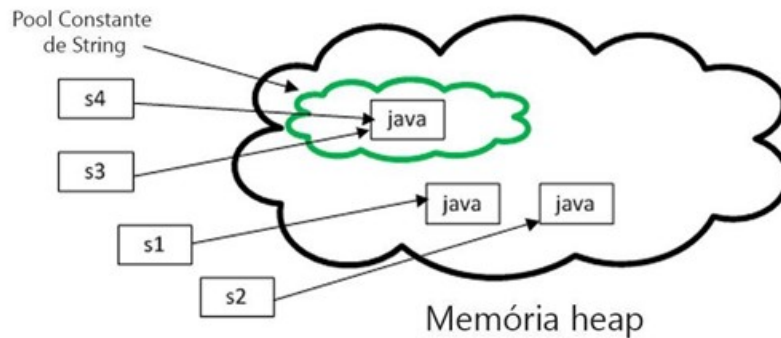
```
public class StringDeepCompareDemo{  
  
    public static void main(String[] args){  
  
        String s1 = new String("java");  
  
        String s2 = new String("java");  
  
        String s3 = "java";  
  
        String s4 = "java";  
  
  
        System.out.print("Comparando S1 and S2 ");  
        System.out.println(s1==s2);  
  
        System.out.print("Comparando S1 and S3 ");  
        System.out.println(s1==s3);  
  
        System.out.print("Comparando S3 and S4 ");  
        System.out.println(s3==s4);  
  
        System.out.print("Comparando S1 and S4 ");  
        System.out.println(s1==s4);  
  
    }  
}
```

**Resultado:**

```
Problems  Javadoc  Declaration  Console  X
<terminated> StringCompareDemo [Java Application] C:\Program Files\
Comparing S1 and S2 false
Comparing S1 and S3 false
Comparing S3 and S4 true
Comparing S1 and S4 false
```

Fonte: Autoral

Como mostrado acima, os objetos String criados usando literais estão passando no resto do teste de igualdade e todos estão falhando. Para a atribuição de String “literal”, se o valor da atribuição for idêntico a outro valor de atribuição de String criado, um novo objeto String não será criado. Uma referência ao objeto String existente é retornada. A figura abaixo explica esse conceito.



Fonte: Autoral

Um dos principais objetivos de qualquer boa linguagem de programação é fazer uso eficiente da memória. À medida que os aplicativos crescem, é muito comum que os literais String ocupem grandes quantidades de memória de um programa, e geralmente há muita redundância no universo de literais String para um programa. Para tornar o Java mais eficiente em memória, a JVM separa uma área especial de memória chamada “Pool constante de String”. Quando o compilador encontra um literal String, ele verifica o pool para ver se já existe um String idêntico. Se uma correspondência for encontrada, a referência ao novo literal será direcionada para a String existente e nenhum novo objeto literal da String será criado.

**Comparando valores do objeto String**

A classe String fornece muitos métodos utilitários, um deles é o método equals. O método equals é usado para comparar o valor do



objeto. Há uma pequena diferença entre o operador `==` e o método `equals()`. A comparação usando `==` é chamada de comparação superficial por causa de `==` retorna `true`, se a referência da variável apontar para o mesmo objeto na memória. A comparação usando o método `equals()` é chamada de comparação profunda porque compara valores de atributo. Vamos entender esse conceito pelo programa java.

```
public class StringCompareDemo{

    public static void main(String[]args)

    {

        String s1 =newString("Hello");

        String s2 =newString("Hello");

        String s3 ="Hello";String s4 ="Java";

        System.out.print("Comparando S1 e S2 ");

        System.out.println(s1.equals(s2));

        System.out.print("Comparando S1 e S3 ");

        System.out.println(s1.equals(s3));

        System.out.print("Comparando S3 e S4 ");

        System.out.println(s3.equals(s4));

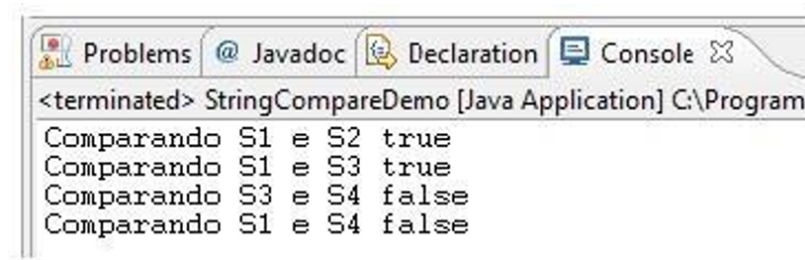
        System.out.print("Comparando S1 e S4 ");

        System.out.println(s1.equals(s4));

    }

}
```

### Resultado



```
<terminated> StringCompareDemo [Java Application] C:\Program
Comparando S1 e S2 true
Comparando S1 e S3 true
Comparando S3 e S4 false
Comparando S1 e S4 false
```

### Explorando métodos da classe String

A manipulação de strings é sem dúvida uma das atividades mais comuns na programação de computadores. A classe String possui uma variedade de métodos para manipulação de strings. Discutiremos métodos básicos com exemplos.



#### **char public charAt (int index)**

Esse método requer um argumento inteiro que indica a posição do caractere que o método retorna. Esse método retorna o caractere localizado no índice especificado da String. Lembre-se de que os índices de string são baseados em zero. Por exemplo,

```
String x = "avião";  
System.out.println (x.charAt (2)); // saída é 'i'  
public String concat (String s)
```

Esse método retorna uma String com o valor da String passado para o método anexado ao final da String usado para invocar o método - por exemplo:

```
String x = "autor";  
System.out.println (x.concat (" do livro")); // output é "autor do livro"
```

Os operadores + e += sobrecarregados executam funções semelhantes ao método concat () - por exemplo:

```
String x = "cartão";  
System.out.println (x + " de biblioteca"); // saída é "cartão de biblioteca"  
String x = "Estados";  
x += " Unidos"  
System.out.println (x); // saída é "Estados Unidos"
```

#### **public boolean equalsIgnoreCase (String s)**

Este método retorna um valor booleano (verdadeiro ou falso), dependendo se o valor da String no argumento é o mesmo que o valor da String usado para invocar o método. Esse método retornará verdadeiro mesmo quando os caracteres nos objetos String comparados tiverem casos diferentes - por exemplo:



```
String x = "Sair";  
System.out.println (x.equalsIgnoreCase ("SAIR")); // é verdade"  
System.out.println (x.equalsIgnoreCase ("tixe")); // é falso"
```

### **public int length ()**

Esse método retorna o comprimento da String usada para invocar o método. Por exemplo:

```
String x = "01234567";  
System.out.println (x.length ()); // retorna "8"
```

### **Substituição pública de String (char old, char new)**

Esse método retorna uma String cujo valor é o da String usada para chamar o método, atualizado para que qualquer ocorrência do caractere no primeiro argumento seja substituída pelo caractere no segundo argumento - por exemplo,

```
String x = "oxoxoxox";  
System.out.println (x.replace ('x', 'X')); // a saída é "oXoXoXoX"
```

### **substring public String (int begin) / substring public String (int begin, int end)**

O método substring () é usado para retornar uma parte (ou substring) da String usada para invocar o método. O primeiro argumento representa o local inicial (baseado em zero) da substring. Se a chamada tiver apenas um argumento, a substring retornada incluirá os caracteres no final da String original. Se a chamada tiver dois argumentos, a substring retornada terminará com o caractere localizado na enésima posição da String original, em que n é o segundo argumento. Infelizmente, o argumento final não é baseado em zero; portanto, se o segundo argumento for 7, o último caractere na String retornado estará na posição 7 da String original, que é o índice 6. Vejamos alguns exemplos:

```
String x = "0123456789"; // o valor de cada caractere é igual ao seu índice!  
System.out.println (x.substring (5)); // saída é "56789"  
System.out.println (x.substring (5, 8)); // saída é "567"
```



### **public String toLowerCase ()**

Esse método retorna uma String cujo valor é a String usada para invocar o método, mas com qualquer caractere maiúsculo convertido em minúsculo - por exemplo,

```
String x = "Um novo livro Java";  
System.out.println (x.toLowerCase ()); // output é "um novo livro java"
```

### **public String toUpperCase ()**

Esse método retorna uma String cujo valor é a String usada para chamar o método, mas com qualquer caractere minúsculo convertido em maiúsculo - por exemplo,

```
String x = "Um novo livro Java";  
System.out.println (x.toUpperCase ()); // saída é "UM NOVO LIVRO  
JAVA"
```

### **public String trim ()**

Esse método retorna uma String cujo valor é a String usada para invocar o método, mas com os espaços em branco à esquerda ou à direita removidos - por exemplo,

```
String x = "oi ";  
System.out.println (x + "x"); // resultado é "oi x"  
System.out.println (x.trim () + "x"); // resultado é "oix"
```

### **char public [] toCharArray ()**

Este método produzirá uma matriz de caracteres a partir de caracteres do objeto String. Por exemplo:





```
String s = "Java";  
char [] arrayChar = s.toCharArray (); // isso produzirá uma matriz de  
tamanho 4
```

```
public boolean contains ("searchString")
```

Esse método retorna true do destino. A String está contendo a String de pesquisa fornecida no argumento. Por exemplo,

```
String x = "Java é linguagem de programação";  
System.out.println (x.contains ("Abel")); // Isso imprimirá falso  
System.out.println (x.contains ("Java")); // Isso será verdadeiro
```

### Classe StringBuffer / StringBuilder

No Java String, objetos imutáveis significam que, uma vez criado, ele não pode ser alterado, a referência aponta para um novo objeto. Se nosso aplicativo tiver atividade de manipulação de strings, haverá muitos objetos de strings descartados na memória heap, que podem resultar em impacto no desempenho. Para contornar essas limitações, você pode usar a classe StringBuilder ou StringBuffer. Você usa uma dessas classes, que são alternativas à classe String, quando você sabe que uma String será modificada; geralmente, você pode usar um objeto StringBuilder ou StringBuffer em qualquer lugar em que usaria uma String. StringBuffer pode ter caracteres e substrings inseridos no meio ou anexados ao final. O StringBuffer cresce automaticamente para abrir espaço para essas adições e geralmente possui mais caracteres pré-alocados do que o necessário para permitir espaço para crescimento. Semelhante à classe String, essas duas classes fazem parte do pacote java.lang e são importadas automaticamente para todos os programas. As classes são idênticas, exceto pelo seguinte:

- StringBuilder é mais eficiente.
- StringBuffer é seguro para threads devido a todos os métodos sincronizados.

Vamos ver abaixo o programa que está comparando o desempenho dos objetos de StringBuffer e String. Neste programa, estamos

criando um novo objeto de string interativamente dentro do loop. No próximo programa, estamos substituindo o objeto String por StringBuffer. O tempo de aplicação é calculado e exibido em um milissegundo.



```
import java.util.Date;

import java.sql.Timestamp;

public class StringComparePerformance{

    public static void main(String[] args){

        Date sData=new Date();

        long sTempo=sData.getTime();

        System.out.println("Hora de início do StringBuffer: "+new
Timestamp(sTempo));

        StringBuffer s =new StringBuffer("AA");

        for(int i=0; i<10000; i++){

            s.append(i);

        }

        Date eData=new Date();

        long eTempo=eData.getTime();

        System.out.println("Fim do tempo do StringBuffer: "+new
Timestamp(eTempo));

        System.out.println("Tempo gasto para executar
StringBufferprocess "+(eTempo-sTempo)+"ms");

        System.out.println("=====

        Date strData=new Date();

        long strTime=strData.getTime();

        System.out.println("Hora de início para String: "+new
Timestamp(strTime));

        String str=new String("AA");

        for(int i=0; i<10000; i++){

            str+=i;
```



```

    }

    Date eStrDate=new Date();

    long eStrTime=eStrDate.getTime();

    System.out.println("Fim do tempo para String: "+new
Timestamp(eStrTime));

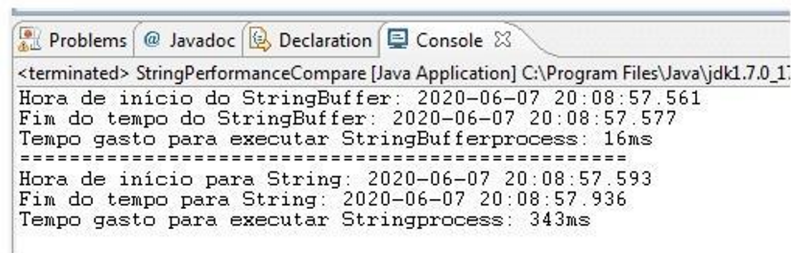
    System.out.println("Tempo gasto para executar
Stringprocess "+(eStrTime-strTime)+"ms");

    }

}

```

### Resultado:



```

<terminated> StringPerformanceCompare [Java Application] C:\Program Files\Java\jdk1.7.0_11
Hora de início do StringBuffer: 2020-06-07 20:08:57.561
Fim do tempo do StringBuffer: 2020-06-07 20:08:57.577
Tempo gasto para executar StringBufferprocess: 16ms
=====
Hora de início para String: 2020-06-07 20:08:57.593
Fim do tempo para String: 2020-06-07 20:08:57.936
Tempo gasto para executar Stringprocess: 343ms

```

Como visto na tela de saída, os objetos String têm desempenho um pouco mais lento devido à criação de novos objetos.

### Métodos importantes das classes StringBuffer e StringBuilder

As classes StringBuffer e StringBuilder têm uma diferença de que StringBuffer está tendo todos os métodos sincronizados, enquanto StringBuilder não é seguro para threads, mas tem melhor desempenho. Aqui discutiremos métodos do StringBuilder que também são aplicáveis ao StringBuffer.

#### Excluir StringBuilder (int startIndex, int endIndex)

Este método exclui uma parte da sequência de caracteres do StringBuilder. Precisamos fornecer dois argumentos int, startIndex e endIndex. Por exemplo:

```

StringBuilder sb = new StringBuilder ("JavaWorld");
sb.delete (4, 8); // o valor sb seria Javad

```

#### Inserção StringBuilder (deslocamento int, String s)



Este método é usado quando queremos inserir uma sequência de caracteres específica, uma sequência de caracteres ou qualquer tipo primitivo em um índice específico dentro da sequência de caracteres StringBuilder. Por exemplo:

```
StringBuilder sb = new StringBuilder ("ABC");  
sb.insert (2, "xyz"); // Aqui o valor sb é ABxyzC
```

String Builder replace (int start, intend, String s)

Este método substitui parte específica da sequência de caracteres pelo terceiro argumento (String) mencionado na chamada do método. Exemplo de sintaxe:

```
StringBuilder sb = new StringBuilder ("ABCDEF");  
sb.replace (1, 3, "XYZ");  
System.out.println (sb); // aqui o valor sb é XYZDEF
```

StringBuilder reverse ()

Como o nome sugere, esse método reverterá a sequência de caracteres no objeto StringBuilder de destino. Exemplo de sintaxe:

```
StringBuilder sb = new StringBuilder ("ABCDEF");  
sb.reverse (); // aqui o valor sb é FEDCBA
```

void setCharAt (int index, char ch)

Este método deve ser usado quando queremos substituir um caractere em um índice específico pelo segundo argumento do método. Exemplo de sintaxe,

```
StringBuilder sb = new StringBuilder ("ABCDEF");  
sb.setCharAt (3, 'x'); // Este valor do ponto sb é ABCxEF
```

### Atividade Extra

- [Artigo: Strings no Java - Não Use o Operador de Igualdade!](#)

- [Site: Java Online Compiler](#)



### Referências Bibliográficas

BARNES, D. J. KOLLING, M. **Programação orientada a objetos com java: uma introdução prática usando o bluej**. 4.ed. Pearson: 2009.

FELIX, R. (Org.). **Programação orientada a objetos**. Pearson: 2017.

MEDEIROS, L. F. de. **Banco de dados: princípios e prática**. Intersaberes: 2013;

ORACLE. Java Documentation, 2021. **Documentação oficial da plataforma Java**. Disponível em: < <https://docs.oracle.com/en/java/> >.

**Ir para exercício**