



# Chapter 7: Strings & Wrapper Classes

Yepang LIU (刘烨庞)

[liuyp1@sustech.edu.cn](mailto:liuyp1@sustech.edu.cn)

# Recall the Primitive Type char

- ▶ The char data type is a single 16-bit Unicode character
  - ‘\u0000’ – ‘\uffff’: 65536 characters, covering characters for almost all modern languages, and a large number of symbols

0000	00D0	00F0	0141	0142	0160	0161	00DD	00FD	0009	000A	00DE	00FE	000D	017D	017E
	Đ đ	Ł ł	Š š	Ý ý							Ɔ ɔ		Ž ž		
0010	0011	0012	0013	0014	00BD	00BC	00B9	00BE	00B3	00B2	00A6	2212	00D7	001E	001F
					½	¼	⅓	¾	³	²	¡	—	×		
0020	0021	0022	0023	0024	0025	0026	0027	0028	0029	002A	002B	002C	002D	002E	002F
	!	"	#	\$	%	&	'	(	)	*	+	,	-	.	/
0030	0031	0032	0033	0034	0035	0036	0037	0038	0039	003A	003B	003C	003D	003E	003F
0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
0040	0041	0042	0043	0044	0045	0046	0047	0048	0049	004A	004B	004C	004D	004E	004F
@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
0050	0051	0052	0053	0054	0055	0056	0057	0058	0059	005A	005B	005C	005D	005E	005F
P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
0060	0061	0062	0063	0064	0065	0066	0067	0068	0069	006A	006B	006C	006D	006E	006F
`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
0070	0071	0072	0073	0074	0075	0076	0077	0078	0079	007A	007B	007C	007D	007E	007F
p	q	r	s	t	u	v	w	x	y	z	{		}	~	
00C4	00C5	00C7	00C9	00D1	00D6	00DC	00E1	00E0	00E2	00E4	00E3	00E5	00E7	00E9	00E8
Ä	Å	Ç	É	Ñ	Ö	Ü	á	à	â	ä	ã	å	ç	é	è

# Recall the Primitive Type char

- ▶ The char data type is a single 16-bit Unicode character
  - ‘\u0000’ – ‘\uffff’: 65536 characters, covering characters for almost all modern languages, and a large number of symbols
- ▶ Programs often contain char literals (in single quotes)

```
char c1 = '\u0030';
```

```
char c2 = '\u0041';
```

```
char c3 = '\u4e2d';
```

```
char c4 = '\u56fd';
```

```
System.out.printf("%c %c %c %c", c1, c2, c3, c4);
```

What is (int)c2 ?

Prints: 0 A 中 国

# String

- ▶ A string is a sequence of characters

```
"I like Java programming"
```

- ▶ A string may include letters, digits and various **special characters**, such as +, -, \*, / and \$.

```
"I \u2665 Java programming"
```

```
I ♥ Java programming
```

Unicode escape sequence for chars you cannot find on keyboard:

\u + a code point in hexadecimal (十六进制码位)



# Objectives

- ▶ Immutable character-string objects of class `String`
- ▶ Mutable character-string objects of class `StringBuilder`

# The String Type

- ▶ String represents a string of characters
- ▶ String is a predefined class in Java.
- ▶ String is a *reference type*

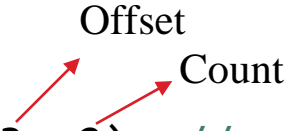
String, like any class, has **fields**, **constructors** and **methods**



```
char[] value;
```

# Creating String Objects (Instantiation)

- ▶ String objects can be created by using the **new** keyword and various **String** constructors

- `String s1 = new String("hello world");`
- `String s2 = new String();` // empty string (length is 0)
- `String s3 = new String(s1);`
- `char[] charArray = {'h', 'e', 'l', 'l', 'o'};`
- `String s4 = new String(charArray);`
- `String s5 = new String(charArray, 3, 2);` // string "lo"  


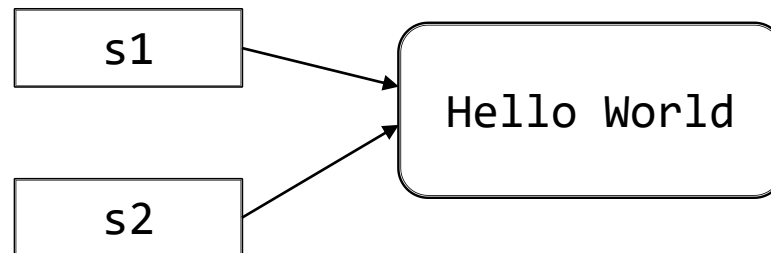
More at: <https://docs.oracle.com/javase/10/docs/api/java/lang/String.html>

# Creating String Objects (Instantiation)

- ▶ String objects can also be created by string literals (字面常量, a sequence of characters in double quotes)

```
String s1 = "Hello World";
```

```
String s2 = s1;
```







# Using String literal vs new keyword

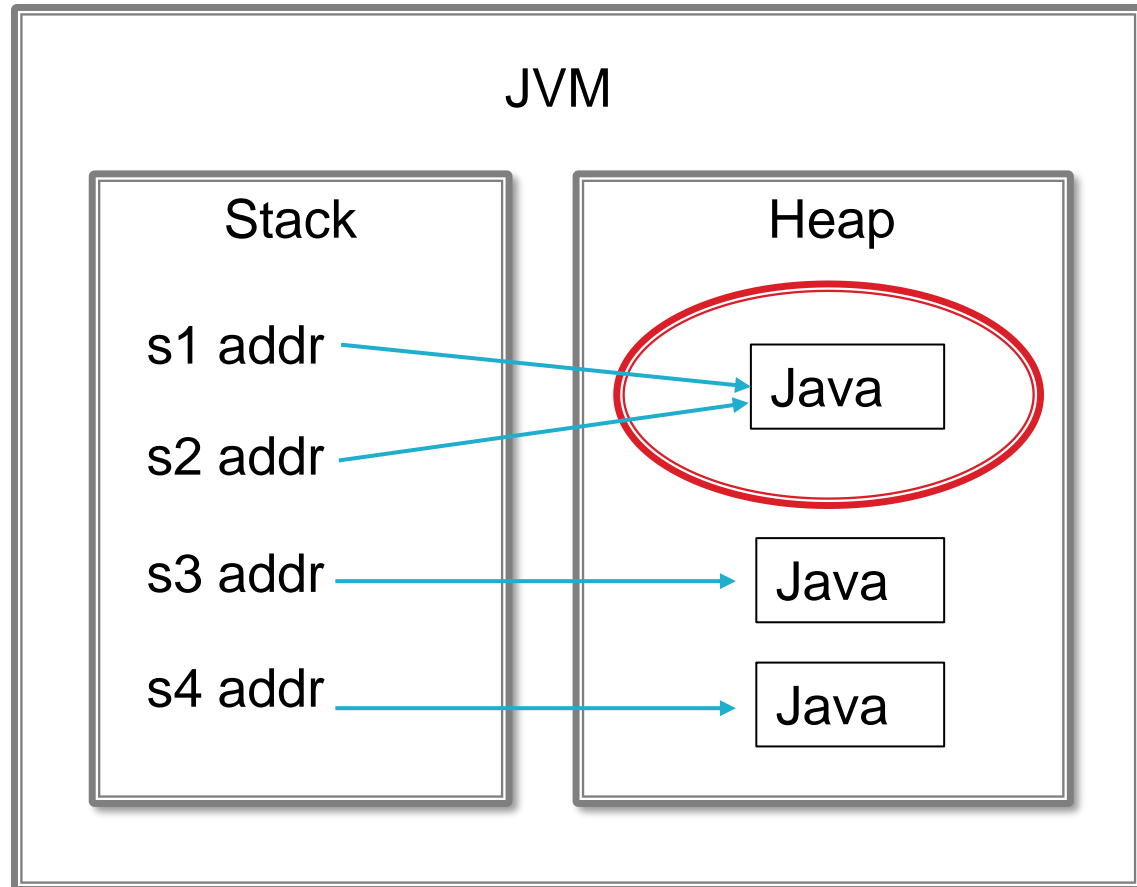
## String Constant Pool:

Store string objects created by string literals

```
String s1 = "Java";  
String s2 = "Java";
```

```
String s3 = new String("Java");  
String s4 = new String("Java");
```

```
System.out.println(s1 == s2); // true  
System.out.println(s3 == s4); // false
```

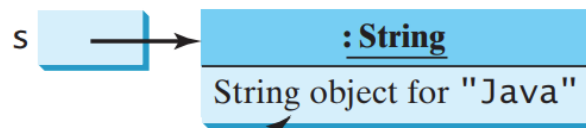


# Immutability (不可变性)

- ▶ In Java, `String` objects are immutable: **their values cannot be changed after they are created.**
- ▶ Any modification creates a new `String` object

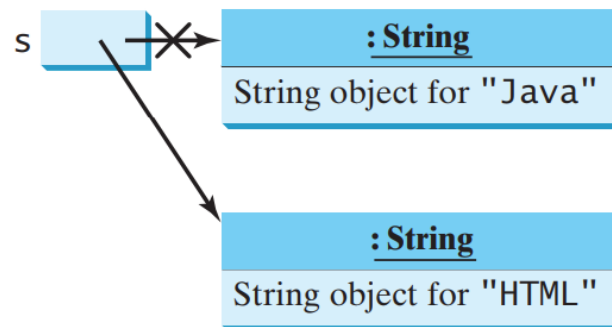
```
String s = "Java";  
s = "HTML";
```

After executing `String s = "Java";`



Contents cannot be changed

After executing `s = "HTML";`



This string object is now unreferenced



# String Methods

- ▶ **length** returns the length of a string (i.e., the number of characters)
- ▶ **charAt** obtains the character at a specific location in a string
- ▶ **getChars** retrieves a set of characters from a string as a char array

These are **instance methods** that can be invoked on specific objects. Calling them requires a non-null object reference.

# The Method `length`

`int length()` Returns the length of this string.

```
public class StringExamples {  
    public static void main(String[] args) {  
        String s1 = "hello world";  
        System.out.printf("s1: %s", s1);  
        System.out.printf("\nLength of s1: %d", s1.length());  
    }  
}
```

```
s1: hello world  
Length of s1: 11
```



# The Method `charAt`

`char`    **`charAt`**(`int index`)    Returns the `char` value at the specified index.

```
public class StringExamples {  
    public static void main(String[] args) {  
        String s1 = "hello world";  
        System.out.printf("s1: %s", s1);  
  
        for(int count = s1.length() - 1; count >=0; count--) {  
            System.out.printf("%c", s1.charAt(count));  
        }  
    }  
}
```

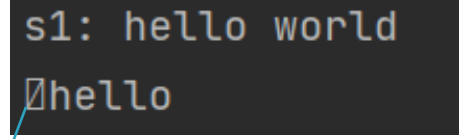
What's the output?

# The Method `getChars`

```
void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)
```

Copies characters from this string into the destination character array.

```
public class StringExamples {  
    public static void main(String[] args) {  
        String s1 = "hello world";  
        char[] charArray = new char[6];  
        System.out.printf("s1: %s\n", s1);  
        s1.getChars(0, 5, charArray, 1);  
        for(char c : charArray) {  
            System.out.print(c);  
        }  
    }  
}
```



```
s1: hello world  
 hello
```

The first Unicode character NUL ('`\u0000`')

# Comparing Strings

- ▶ When primitive-type values are compared with `==`, the result is `true` if both values are identical.

```
int a = 2, b = 2;  
if (a == b) System.out.println("a = b"); // prints a = b
```

- ▶ When references (memory addresses) are compared with `==`, the result is `true` if both references refer to the same object in memory.

```
String s1 = "Hello World";  
String s2 = "Hello World";  
if(s1 == s2) System.out.println("s1 = s2"); // prints s1 = s2
```

# Comparing Strings

```
String s1 = "Hello World";  
String s2 = s1 + "";  
if(s1 == s2) System.out.println("s1 = s2"); // prints s1 = s2?
```

- **No. The condition will evaluate to false** because the `String` variables `s1` and `s2` refer to two different `String` objects, although the strings contain the same sequence of characters.
- To compare the actual contents (or state information) of objects (strings are objects) for equality, a method `equals` must be invoked.



# The Method `equals`

- ▶ Method `equals` tests any two objects for equality—the strings contained in the two `String` objects are identical.

```
String s1 = "Hello World";  
String s2 = s1 + "";  
if(s1.equals(s2)) System.out.println("s1 = s2"); // true
```

```
String s1 = "hello";  
String s2 = "HELLO";  
if(s1.equals(s2)) System.out.println("s1 = s2"); // false
```



# The Method `equalsIgnoreCase`

- ▶ Method `equalsIgnoreCase` ignores whether the letters in each `String` are uppercase or lowercase when performing a comparison.

```
String s1 = "hello";  
  
String s2 = "HELLO";  
  
if(s1.equalsIgnoreCase(s2)) System.out.println("s1 = s2");
```

The condition evaluates to `true` and the program prints “s1 = s2”



# Comparing Strings

- ▶ What does it mean when we say a string `s1` is greater than another string `s2`?
  - When we sort last names, we naturally consider that “Jones” < “Smith”, because the letter ‘J’ comes before ‘S’ in the alphabet of 26 letters.
  - All characters in computers are represented as numeric codes. The characters form an ordered set (a very large alphabet: Unicode Table).
  - When the computer compares `Strings`, it actually compares the numeric codes of the characters in the `Strings`.

# Unicode Table

Line Feed  
'\n' (LF)  
(a white-space char)

Digits

Letters

Operators

0000	00D0	00F0	0141	0142	0160	0161	00DD	00FD	0009	000A	00DE	00FE	000D	017D	017E
	Đ	đ	Ł	ł	Š	š	Ý	ý			Þ	þ		Ž	ž
0010	0011	0012	0013	0014	00BD	00BC	00B9	00BE	00B3	00B2	00A6	2212	00D7	001E	001F
					½	¼	⅓	¾	⅓	²	¡	—	×		
0020	0021	0022	0023	0024	0025	0026	0027	0028	0029	002A	002B	002C	002D	002E	002F
	!	"	#	\$	%	&	'	(	)	*	+	,	-	.	/
0030	0031	0032	0033	0034	0035	0036	0037	0038	0039	003A	003B	003C	003D	003E	003F
0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
0040	0041	0042	0043	0044	0045	0046	0047	0048	0049	004A	004B	004C	004D	004E	004F
@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
0050	0051	0052	0053	0054	0055	0056	0057	0058	0059	005A	005B	005C	005D	005E	005F
P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
0060	0061	0062	0063	0064	0065	0066	0067	0068	0069	006A	006B	006C	006D	006E	006F
`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
0070	0071	0072	0073	0074	0075	0076	0077	0078	0079	007A	007B	007C	007D	007E	007F
p	q	r	s	t	u	v	w	x	y	z	{		}	~	
00C4	00C5	00C7	00C9	00D1	00D6	00DC	00E1	00E0	00E2	00E4	00E3	00E5	00E7	00E9	00E8
Ä	Å	Ç	É	Ñ	Ö	Ü	á	à	â	ä	ã	å	ç	é	è
00EA	00EB	00ED	00EC	00EE	00EF	00F1	00F3	00F2	00F4	00F6	00F5	00FA	00F9	00FB	00FC
ê	ë	í	ì	î	ï	ñ	ó	ò	ô	ö	õ	ú	ù	û	ü
2020	00B0	00A2	00A3	00A7	2022	00B6	00DF	00AE	00A9	2122	00B4	00A8	2260	00C6	00D8
†	°	¢	£	§	•	¶	ß	®	©	™	'	''	≠	Æ	Ø
221E	00B1	2264	2265	00A5	00B5	2202	2211	220F	03C0	222B	00AA	00BA	03A9	00E6	00F8
∞	±	≤	≥	¥	μ	∂	Σ	Π	π	∫	ª	º	Ω	æ	ø
00BF	00A1	00AC	221A	0192	2248	2206	00AB	00BB	2026	00A0	00C0	00C3	00D5	0152	0153
¿	¡	¬	√	ƒ	≈	Δ	«	»	...		À	Ã	Õ	Œ	œ



# The Method compareTo

```
String s1 = "hello";  
String s2 = "HELLO";  
int result = s1.compareTo(s2); // value of result?
```

compareTo compares two strings (lexicographical comparison):

- ▶ Returns 0 if the Strings are equal (identical contents).
- ▶ Returns a negative number if the String that invokes compareTo (s1) is **less than** the String that is passed as an argument (s2).
- ▶ Returns a positive number if the String that invokes compareTo (s1) is **greater than** the String that is passed as an argument (s2).



# Comparing Strings

0000	0000	00F0	0141	0142	0160	0161	00DD	00FD	0009	000A	00DE	00FE	000D	017D	017E
	Đ	đ	Ł	ł	Š	š	Ý	ý			Þ	þ		Ž	ž
0010	0011	0012	0013	0014	00BD	00BC	00B9	00BE	00B3	00B2	00A6	2212	00D7	001E	001F
					½	¼	⅓	¾	³	²	¹	—	×		
0020	0021	0022	0023	0024	0025	0026	0027	0028	0029	002A	002B	002C	002D	002E	002F
	!	"	#	\$	%	&	'	(	)	*	+	,	-	.	/
0030	0031	0032	0033	0034	0035	0036	0037	0038	0039	003A	003B	003C	003D	003E	003F
	0	1	2	3	4	5	6	7	8	9	.	.	.	.	.
0040	0041	0042	0043	0044	0045	0046	0047	0048	0049						
	@	A	B	C	D	E	F	G	H	I					
0050	0051	0052	0053	0054	0055	0056	0057	0058	0059	00DA	00DB	00DC	00DD	00DE	00DF
	P	Q	R	S	T	U	V	W	X	Y	¿	¸	¹	º	»
0060	0061	0062	0063	0064	0065	0066	0067	0068	0069						
	`	a	b	c	d	e	f	g	h	i					
0070	0071	0072	0073	0074	0075	0076	0077	0078	0079	007A	007B	007C	007D	007E	007F
	p	q	r	s	t	u	v	w	x	y	z	{		}	~
00C4	00C5	00C7	00C9	00D1	00D6	00DC	00E1	00E0	00E2	00E4	00E3	00E5	00E7	00E9	00E8
	Ä	Å	Ç	É	Ñ	Ö	Ü	á	à	â	ä	ã	å	ç	é
00EA	00EB	00ED	00EC	00EE	00EF	00F1	00F3	00F2	00F4	00F6	00F5	00FA	00F9	00FB	00FC
	ê	ë	í	ì	î	ï	ñ	ó	ò	ô	ö	õ	ú	ù	û
2020	00B0	00A2	00A3	00A7	2022	00B6	00DF	00AE	00A9	2122	00B4	00A8	2260	00C6	00D8
	†	°	¢	£	§	•	¶	®	©	™	´	¨	≠	Æ	Ø
221E	00B1	2264	2265	00A5	00B5	2211	220F	03C0	222B	00AA	00BA	03A9	00E6	00F8	
	∞	±	≤	≥	¥	μ	∂	Σ	Π	π	∫	ª	º	Ω	æ
00BF	00A1	00AC	221A	0192	2248	2206	00AB	00BB	2026	00A0	00C0	00C3	00D5	0152	0153
	¿	¡	¬	√	ƒ	≈	Δ	«	»	...	À	Ã	Õ	Œ	œ

```
String s1 = "hello", s2 = "HELLO";  
int result = s1.compareTo(s2));
```

32 = 0068 (HEX) – 0048 (HEX) (s1 > s2)

```
String s1 = "HE", s2 = "HELLO";  
int result = s1.compareTo(s2));
```

-3 (s1 < s2, s2 has three more letters)

```
String s1 = "HEL", s2 = "HEllo";  
int result = s1.compareTo(s2));
```

-32 (s1 < s2)



# Methods `startsWith` & `endsWith`

The methods `startsWith` and `endsWith` determine whether a string starts or ends with the method argument, respectively

```
String s1 = "Hello World";  
if(s1.startsWith("He")) System.out.print("true"); // true
```

```
String s1 = "Hello World";  
if(s1.startsWith("llo", 2)) System.out.print("true"); // true
```

```
String s1 = "Hello World";  
if(s1.endsWith("ld")) System.out.print("true"); // true
```

# Locating Characters in Strings

```
String s = "abcdefghijklmabcdefghijklm";  
System.out.println(s.indexOf('c')); // 2  
System.out.println(s.indexOf('$')); // -1  
System.out.println(s.indexOf('a', 1)); // 13
```

- ▶ `indexOf` locates the **first occurrence** of a character in a `String`.
  - If the method finds the character, it returns the character's index in the `String`; otherwise, it returns `-1`.
- ▶ Two-argument version of `indexOf`:
  - Take one more argument: the starting index at which the search should begin.



# Locating Characters in Strings

```
String s = "abcdefghijklmabcdefghijklm";  
System.out.println(s.lastIndexOf('c')); // 15  
System.out.println(s.lastIndexOf('$')); // -1  
System.out.println(s.lastIndexOf('a', 8)); // 0
```

- ▶ `lastIndexOf` locates the **last occurrence** of a character in a `String`.
  - The method searches from the end of the `String` toward the beginning.
  - If it finds the character, it returns the character's index in the `String`; otherwise, it returns `-1`.
- ▶ Two-argument version of `lastIndexOf`:
  - The character and the index from which to begin searching backward.

# Locating Substrings in Strings

```
String s = "abcdefghijklmabcdefghijklm";  
System.out.println(s.indexOf("def"));      // 3  
System.out.println(s.indexOf("def", 7));   // 16  
System.out.println(s.indexOf("hello"));   // -1  
System.out.println(s.lastIndexOf("def")); // 16  
System.out.println(s.lastIndexOf("def", 7)); // 3  
System.out.println(s.lastIndexOf("hello")); // -1
```

- ▶ The versions of methods `indexOf` and `lastIndexOf` that take a `String` as the first argument perform identically to those described earlier except that they search for sequences of characters (or substrings) that are specified by their `String` arguments.

# Extracting Substrings from Strings

```
String s = "abcdefghijklmabcdefghijklm";  
System.out.println(s.substring(20)); // hijklm  
System.out.println(s.substring(3, 6)); // def
```

- ▶ substring methods create a new String object by copying part of an existing String object.
- ▶ The one-integer-argument version specifies the starting index (**inclusive**) in the original String from which characters are to be copied.
- ▶ Two-integer-argument version specifies the starting index (**inclusive**) and ending index (**exclusive**) to copy characters in the original String.

# String Method `replace`

```
String s1 = "Hello";  
System.out.println(s1.replace('l', 'L')); // HeLLo  
System.out.println(s1.replace("ll", "LL")); // HeLLo
```

- ▶ `replace` returns a new **String** object in which every occurrence of the first character argument is replaced with the second character argument.
- ▶ Another version of method `replace` enables you to replace substrings rather than individual characters (every occurrence of the first substring is replaced).



# String Case Conversion Methods

```
String s1 = "Hello";  
System.out.println(s1.toUpperCase()); // HELLO  
System.out.println(s1.toLowerCase()); // hello
```

- ▶ String method `toUpperCase` returns a new `String` object with uppercase letters where corresponding lowercase letters exist in the original.
- ▶ String method `toLowerCase` returns a new `String` object with lowercase letters where corresponding uppercase letters exist in the original.

# String Method `trim`

- ▶ `trim` returns a new `String` object that removes all white-space characters at the beginning or end of the `String` on which `trim` operates.

```
String s1 = " spaces ";  
System.out.println(s1.trim()); //prints "spaces"
```

# String Method `toCharArray`

- ▶ `toCharArray` creates a new character array containing a copy of the characters in the string.

```
String s1 = "hello";  
  
char[] charArray = s1.toCharArray();  
  
for(char c : charArray) System.out.print(c);
```

The for loop prints each of the five chars in “hello”



# String Method `split`

- ▶ When you read a sentence, your mind breaks it into tokens—individual words and punctuation marks that convey meaning to you.
- ▶ String method `split` breaks a `String` into its component tokens, separated from each other by **delimiters** (分隔符), typically white-space characters such as space, tab, new line, carriage return.



# String Method `split`

```
Scanner input = new Scanner(System.in);
System.out.println("Enter a sentence and press Enter");
String sentence = input.nextLine();
String[] tokens = sentence.split(" ");
System.out.printf("Number of tokens: %d\n", tokens.length);
for(String token : tokens) System.out.println(token);
input.close();
```

How about `sentence.split("is")`?

```
Enter a sentence and press Enter
This is a sentence with seven tokens
Number of tokens: 7
This
is
a
sentence
with
seven
tokens
```

# Concatenating Strings

```
String s1 = "Happy ";  
String s2 = "Birthday";  
System.out.println(s1.concat(s2)); // Happy Birthday  
System.out.println(s1); // Happy
```

- ▶ String method `concat` concatenates two `String` objects and returns a new `String` object containing the characters from both original `Strings`.
- ▶ The original `Strings` to which `s1` and `s2` refer are not modified

# Concatenating Strings

```
public static void main(String[] args) {  
    String s = "";  
    for (int i = 0; i < 1000; i++) {  
        s = s + "," + i;  
    }  
}
```

String concatenation '+' in loop

Convert variable 's' from String to StringBuilder

- ▶ We can use “+” for String concatenation
- ▶ However, when + is used in a loop, a new string will be created in every iteration (because of immutability), which is **inefficient**
- ▶ Better to use StringBuilder, which is **mutable**



# Objectives

- ▶ Immutable character-string objects of class `String`
- ▶ Mutable character-string objects of class `StringBuilder`

# Class StringBuilder

- ▶ **String objects are immutable.** Can we create mutable character-string objects in Java?
- ▶ Yes. The class `StringBuilder` helps create and manipulate dynamic string information, i.e., **modifiable, mutable strings**.
- ▶ You can add, insert, or append new contents into `StringBuilder`

# StringBuilder Constructors

- ▶ Every `StringBuilder` is capable of storing a number of characters specified by its **capacity**.
- ▶ If a `StringBuilder`'s capacity is exceeded, the capacity automatically expands to accommodate additional characters.

## `java.lang.StringBuilder`

```
+StringBuilder()  
+StringBuilder(capacity: int)  
+StringBuilder(s: String)
```

Constructs an empty string builder with capacity 16.  
Constructs a string builder with the specified capacity.  
Constructs a string builder with the specified string.



# StringBuilder Constructors

Default initial capacity is 16 chars

```
StringBuilder buffer1 = new StringBuilder();  
StringBuilder buffer2 = new StringBuilder(10);  
StringBuilder buffer3 = new StringBuilder("hello");  
System.out.printf("buffer1 = \"%s\"\\n", buffer1);  
System.out.printf("buffer2 = \"%s\"\\n", buffer2);  
System.out.printf("buffer3 = \"%s\"\\n", buffer3);
```

```
buffer1 = ""  
buffer2 = ""  
buffer3 = "hello"
```

# StringBuilder Method **append**

- ▶ Class `StringBuilder` provides several `append` methods to **allow values of various types to be appended** to the end of a `StringBuilder` object.
- ▶ Overloaded `append()` are provided for each of the primitive types, and for character arrays, Strings, Objects, and more.

```
append(boolean b)
```

```
append(char c)
```

```
append(char[] str)
```

```
append(char[] str, int offset, int len)
```

```
append(double d)
```

```
append(float f)
```

```
append(int i)
```

```
append(long lng)
```

```
append(CharSequence s)
```

```
append(CharSequence s, int start, int end)
```

```
append(Object obj)
```

```
append(String str)
```

```
append(StringBuffer sb)
```





```
1. String string = "goodbye";
2. char[] charArray = {'a', 'b', 'c', 'd', 'e', 'f'};
3. boolean booleanValue = true;
4. char charValue = 'Z';
5. int intValue = 7;
6. long longValue = 10000000000L;
7. float floatValue = 2.5f;
8. double doubleValue = 33.3333;
9. StringBuilder buffer = new StringBuilder();
10. StringBuilder lastBuffer = new StringBuilder("last buffer");

11. buffer.append(string); buffer.append("\n");
12. buffer.append(charArray); buffer.append("\n");
13. buffer.append(charArray, 0, 3); buffer.append("\n");
14. buffer.append(booleanValue); buffer.append("\n");
15. buffer.append(charValue); buffer.append("\n");
16. buffer.append(intValue); buffer.append("\n");
17. buffer.append(longValue); buffer.append("\n");
18. buffer.append(floatValue); buffer.append("\n");
19. buffer.append(doubleValue); buffer.append("\n");
20. buffer.append(lastBuffer);

21. System.out.printf("buffer contains:\n%s", buffer.toString());
```

```
buffer contains:
goodbye
abcdef
abc
true
Z
7
10000000000
2.5
33.3333
last buffer
```

Here we still use the same `StringBuilder` object reference, because `StringBuilder` objects are mutable.

# Other StringBuilder Method

## java.lang.StringBuilder

```
+toString(): String  
+capacity(): int  
+charAt(index: int): char  
+length(): int  
+setLength(newLength: int): void  
+substring(startIndex: int): String  
+substring(startIndex: int, endIndex: int):  
    String  
+trimToSize(): void
```

Returns a string object from the string builder.

Returns the capacity of this string builder.

Returns the character at the specified index.

Returns the number of characters in this builder.

Sets a new length in this builder.

Returns a substring starting at `startIndex`.

Returns a substring from `startIndex` to `endIndex-1`.

Reduces the storage size used for the string builder.

```
+deleteCharAt(index: int): StringBuilder  
+insert(index: int, data: char[], offset: int,  
    len: int): StringBuilder  
+insert(offset: int, data: char[]):  
    StringBuilder  
+insert(offset: int, b: aPrimitiveType):  
    StringBuilder  
+insert(offset: int, s: String): StringBuilder  
+replace(startIndex: int, endIndex: int, s:  
    String): StringBuilder  
+reverse(): StringBuilder  
+setCharAt(index: int, ch: char): void
```

Deletes a character at the specified index.

Inserts a subarray of the data in the array into the builder at the specified index.

Inserts data into this builder at the position offset.

Inserts a value converted to a string into this builder.

Inserts a string into this builder at the position offset.

Replaces the characters in this builder from `startIndex` to `endIndex-1` with the specified string.

Reverses the characters in the builder.

Sets a new character at the specified index in this builder.

# Read the Documentation!

- ▶ <https://docs.oracle.com/javase/8/docs/api/java/lang/String.html>
- ▶ <https://docs.oracle.com/javase/8/docs/api/java/lang/StringBuilder.html>



# Read the Documentation in IDEA

```
String str = "Java";
String str2 = str;
str.concat(" course");
```

Result of 'String.concat()' is ignored

```
@NotNull
public String concat(
    @NotNull String str
)
```

Concatenates the specified string to the end of this string.  
If the length of the argument string is 0, then this String object is returned.  
Otherwise, a String object is returned that represents a character sequence that is the concatenation of the character sequence represented by this String object and the character sequence represented by the argument string.

Examples:

```
"cares".concat("s") returns "caress"
"to".concat("get").concat("her") returns "together"
```

Params: str – the String that is concatenated to the end of this String.

Returns: a string that represents the concatenation of this object's characters followed by the string argument's characters.

```
String s1 = "Java";
String s2 = "Java";
```

s1.l

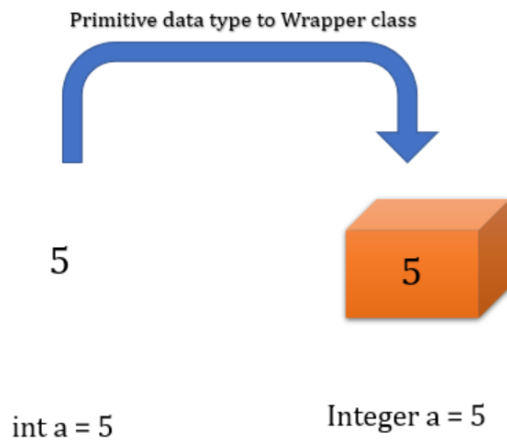
- m length()
- m lastIndexOf(int ch)
- m lines()
- m lastIndexOf(String str)
- m lastIndexOf(int ch, int fromIndex)
- m lastIndexOf(String str, int fromIndex)
- m toLowerCase(Locale.ROOT)
- m toLowerCase(Locale locale)
- m toLowerCase()



## ► Part II: Wrapper Classes

# Wrapper Classes

- ▶ Java has 8 primitive types: boolean, char, double, float, byte, short, int and long
- ▶ Java also provides 8 type-wrapper classes—Boolean, Character, Double, Float, Byte, Short, Integer and Long—that enable primitive-type values to be treated as objects.



Be careful: not Int or Char



# Character Class

- ▶ The class `Character` is the type-wrapper class for the primitive type `char`
- ▶ `Character` provides methods (mostly **static** ones) for convenience in processing individual `char` values
  - `isDigit(char c)`
  - `isLetter(char c)`
  - `isLowerCase(char c)`



# Useful Character Methods

```
Scanner sc = new Scanner(System.in);
System.out.println("Enter a character and press Enter:");
String input = sc.next();
char c = input.charAt(0);

System.out.printf("is digit: %b\n", Character.isDigit(c));
System.out.printf("is identifier start: %b\n", Character.isJavaIdentifierStart(c));
System.out.printf("is letter: %b\n", Character.isLetter(c));
System.out.printf("is lower case: %b\n:", Character.isLowerCase(c));
System.out.printf("is upper case: %b\n", Character.isUpperCase(c));
System.out.printf("to upper case: %c\n", Character.toUpperCase(c));
System.out.printf("to lower case: %c\n", Character.toLowerCase(c));

sc.close();
```



# Useful Character Methods

Enter a character and press Enter:

A

is digit: false

is identifier start: true

is letter: true

is lower case: false

is upper case: true

to upper case: A

to lower case: a

Enter a character and press Enter:

8

is digit: true

is identifier start: false

is letter: false

is lower case: false

is upper case: false

to upper case: 8

to lower case: 8

Java identifiers can only start with a letter, an underscore (`_`), or a dollar sign (`$`)



# Other Useful Methods

- ▶ `Integer.parseInt(String s)` : parses the string argument as a decimal integer value
  - `Integer.parseInt("123")` returns an integer 123
  - `Integer.parseInt("123abc")` returns a `NumberFormatException`
- ▶ `Double.parseDouble(String s)...`
- ▶ Check the Java API documentation for more details
  - <https://docs.oracle.com/javase/10/docs/api/>