



CS 113 – Computer Science I

Lecture 4 – Loops

Adam Poliak

09/13/2022

Announcements

- Assignment 00
 - Good job!
 - Grades coming out soon
- Assignment 01
 - Due Thursday 09/15
- Office hours:
 - Adam's: 10:30-11:30am on Wednesdays



Agenda

- Announcements
- Recap
- String Comparison
- Loops

Comparing strings

- In Java, you cannot directly compare strings: use **compareTo**
 - Javadocs: <https://docs.oracle.com/javase/7/docs/api/java/lang/String.html>

compareTo

```
public int compareTo(String anotherString)
```

Compares two strings lexicographically. The comparison is based on the Unicode value of each character in the strings. The character sequence represented by this `String` object is compared lexicographically to the character sequence represented by the argument string. The result is a negative integer if this `String` object lexicographically precedes the argument string. The result is a positive integer if this `String` object lexicographically follows the argument string. The result is zero if the strings are equal; `compareTo` returns 0 exactly when the `equals(Object)` method would return `true`.

This is the definition of lexicographic ordering. If two strings are different, then either they have different characters at some index that is a valid index for both strings, or their lengths are different, or both. If they have different characters at one or more index positions, let k be the smallest such index; then the string whose character at position k has the smaller value, as determined by using the `<` operator, lexicographically precedes the other string. In this case, `compareTo` returns the difference of the two character values at position k in the two string -- that is, the value:

$$\text{this.charAt}(k) - \text{anotherString.charAt}(k)$$

If there is no index position at which they differ, then the shorter string lexicographically precedes the longer string. In this case, `compareTo` returns the difference of the lengths of the strings -- that is, the value:

$$\text{this.length}() - \text{anotherString.length}()$$

Specified by:

`compareTo` in interface `Comparable<String>`

Parameters:

`anotherString` - the `String` to be compared.

Returns:

the value 0 if the argument string is equal to this string; a value less than 0 if this string is lexicographically less than the string argument; and a value greater than 0 if this string is lexicographically greater than the string argument.

compareTo

```
public int compareTo(String anotherString)
```

Compares two strings lexicographically. The comparison is based on the Unicode value of each character in the strings. The character sequence represented by this `String` object is compared lexicographically to the character sequence represented by the argument string. The result is a negative integer if this `String` object lexicographically precedes the argument string. The result is a positive integer if this `String` object lexicographically follows the argument string. The result is zero if the strings are equal; `compareTo` returns 0 exactly when the `equals(Object)` method would return `true`.

This is the definition of lexicographic ordering. If two strings are different, then either they have different characters at some index that is a valid index for both strings, or their lengths are different, or both. If they have different characters at one or more index positions, let k be the smallest such index; then the string whose character at position k has the smaller value, as determined by using the `<` operator, lexicographically precedes the other string. In this case, `compareTo` returns the difference of the two character values at position k in the two string -- that is, the value:

$$\text{this.charAt}(k) - \text{anotherString.charAt}(k)$$

If there is no index position at which they differ, then the shorter string lexicographically precedes the longer string. In this case, `compareTo` returns the difference of the lengths of the strings -- that is, the value:

$$\text{this.length}() - \text{anotherString.length}()$$

Specified by:

`compareTo` in interface `Comparable<String>`

Parameters:

`anotherString` - the `String` to be compared.

Returns:

the value 0 if the argument string is equal to this string; a value less than 0 if this string is lexicographically less than the string argument; and a value greater than 0 if this string is lexicographically greater than the string argument.

```
public int compareTo(String anotherString)
```

Parameters:

`anotherString` - the `String` to be compared.

Returns:

- the value 0 if the argument string is equal to this string;
- a value less than 0 if this string is lexicographically less than the string argument;
- and a value greater than 0 if this string is lexicographically greater than the string argument.

Comparing strings

- In Java, you cannot directly compare strings: use **compareTo**

```
String a = "apple";  
String b = "banana";  
if (a.compareTo(b) == 0) {  
    System.out.println("a and b match!");  
}  
if (a.compareTo(b) != 0) {  
    System.out.println("a and b DO NOT match!");  
}
```


Lexicographic Values/Order

- Strings are **ordered lexicographically**
 - Generally, the same order as alphabetical order, with some caveats
 - The characters of a string each correspond to a number

ASCII

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040	 	Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	\$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	((72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051))	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[[123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174	|	
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135]]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177		DEL

Source: www.LookupTables.com

<https://www.asciitable.com/>

StringCompare.java

```
String first = "a";
String second = "A";
int asciia = (int) first.charAt(0);
int asciib = (int) second.charAt(0);
System.out.println("ASCII Code for "+first+" is " + asciia);
System.out.println("ASCII Code for "+second+" is " + asciib);

if (first.compareTo(second) == 0) {
    System.out.println(first+" is equal to "+second);
}
else if (first.compareTo(second) < 0) {
    System.out.println(first+" is less than "+second);
}
else if (first.compareTo(second) > 0) {
    System.out.println(first+" is greater than "+second);
}
```

```
$ java StringCompare
ASCII Code for a is 97
ASCII Code for A is 65
a is greater than A
```

Exercise: IsPrimary

Write a program that asks the user for a color and prints whether the color is primary or not.

- The primary colors are “red”, “yellow”, “blue”
- All other inputs are non-primary

```
$ java IsPrimary  
Enter a color: green  
green is not primary
```

```
$ java IsPrimary  
Enter a color: blue  
blue is primary
```



Agenda

- Announcements
- Recap
- String Comparison
- **Loops**

Exercise

Suppose we wanted to ask the user for 6 numbers (int) and output their sum?

Loops

- Easy way to repeat some computation
- Two kinds of loops:
 - While
 - For
- Loops repeat block of code until the condition becomes false

Example: While Loop <

```
int val = 0;
String valStr = "";
int sum = 0;

int count = 0;
while (count < 6) {
    System.out.print("Enter a number: ");
    valStr = System.console().readLine();
    val = Integer.parseInt(valStr);
    sum = sum + val;
    count = count + 1;
}
System.out.println("The sum is "+sum);
```

Tracing Loops

```
int sum = 1;
int count = 0;
while (count < 3) {
    sum = sum + 2;
    count = count + 1;
}
```

Iteration	Count < 6	count	sum

Tracing Loops

```
int sum = 1;
int count = 0;
while (count < 3) {
    sum = sum + 2;
    count = count + 1;
}
```

Iteration	Count < 6	count	sum
0	T	0	1

Tracing Loops

```
int sum = 1;
int count = 0;
while (count < 3) {
    sum = sum + 2;
    count = count + 1;
}
```

Iteration	Count < 6	count	sum
0	T	0	1
1	T	1	3

Tracing Loops

```
int sum = 1;
int count = 0;
while (count < 3) {
    sum = sum + 2;
    count = count + 1;
}
```

Iteration	Count < 6	count	sum
0	T	0	1
1	T	1	3
2	T	2	5

Tracing Loops

```
int sum = 1;
int count = 0;
while (count < 3) {
    sum = sum + 2;
    count = count + 1;
}
```

Iteration	Count < 6	count	sum
0	T	0	1
1	T	1	3
2	T	2	5
3	T	3	7

Exercise: Tracing loops

```
int sum = 10;  
int count = 0;  
while (count < 6) {  
    sum = sum - 1;  
    count = count + 2;  
}
```

Iteration	Count < 6	count	sum

Accumulator pattern

Idea: Repeatedly update a variable (typically in a loop)

Pattern:

1. Initialize accumulator variable
2. Loop until done
 1. Update the accumulator variable

Convenience syntax: Assignment

Because updating variable values is so common, language such as Java provide shorthand syntax for it

- Analogy: contractions in English

```
sum = sum + 2
```

```
count = count + 1
```

```
count = count - 1
```

```
product = product * 2
```

```
divisor = divisor / 2
```

```
message = message + "lol!"
```

Convenience syntax: Assignment

Because updating variable values is so common, language such as Java provide shorthand syntax for it

- Analogy: contractions in English

<code>sum = sum + 2</code>	
<code>count = count + 1</code>	
<code>count = count - 1</code>	
<code>product = product * 2</code>	
<code>divisor = divisor / 2</code>	
<code>message = message + " lol"</code>	

Convenience syntax: Assignment

Because updating variable values is so common, language such as Java provide shorthand syntax for it

- Analogy: contractions in English

<code>sum = sum + 2</code>	<code>sum += 2</code>
<code>count = count + 1</code>	<code>count += 1</code>
<code>count = count - 1</code>	<code>count -= 1</code>
<code>product = product * 2</code>	<code>product *= 2</code>
<code>divisor = divisor / 2</code>	<code>divisor /= 2</code>
<code>message = message + " lol"</code>	<code>message += " lol"</code>

Exercise: Write a program that computes powers of 2

Write a program, LoopPow2.java, that computes powers of twos. For example,

```
$ java LoopPow2
Enter an exponent: 0
2 to the power of 0 is 1

$ java LoopPow
Enter an exponent: 1
2 to the power of 1 is 2

$ java LoopPow
Enter an exponent: 4
2 to the power of 4 is 16
```

Example: For Loop



```
int val = 0;
String valStr = "";
int sum = 0;

for (int count = 0; count < 6; count = count +1) {
    System.out.print("Enter a number: ");
    valStr = System.console().readLine();
    val = Integer.parseInt(valStr);
    sum = sum + val;
}
System.out.println("The sum is "+sum);
```

Exercise: Tracing loops

```
String pattern = "";  
for (int i = 0; i < 3; i++) {  
    pattern = pattern + "*";  
}  
System.out.println(pattern);
```

Exercise: LoopPattern

```
$ java LoopPattern
```

```
Enter a length: 5
```

```
*_*_*
```

```
$ java LoopPattern
```

```
Enter a length: 10
```

```
*_*_*_*_*_*
```

```
$ java LoopPattern
```

```
Enter a length: 0
```

```
$ java LoopPattern
```

```
Enter a length: 1
```

```
*
```

Exercise: Nested loops

```
$ java Square
```

```
Enter a size: 5
```

```
*****
```

```
*****
```

```
*****
```

```
*****
```

```
*****
```

```
$ java Square
```

```
Enter a size: 1
```

```
*
```

```
$ java Square
```

```
Enter a size: 0
```