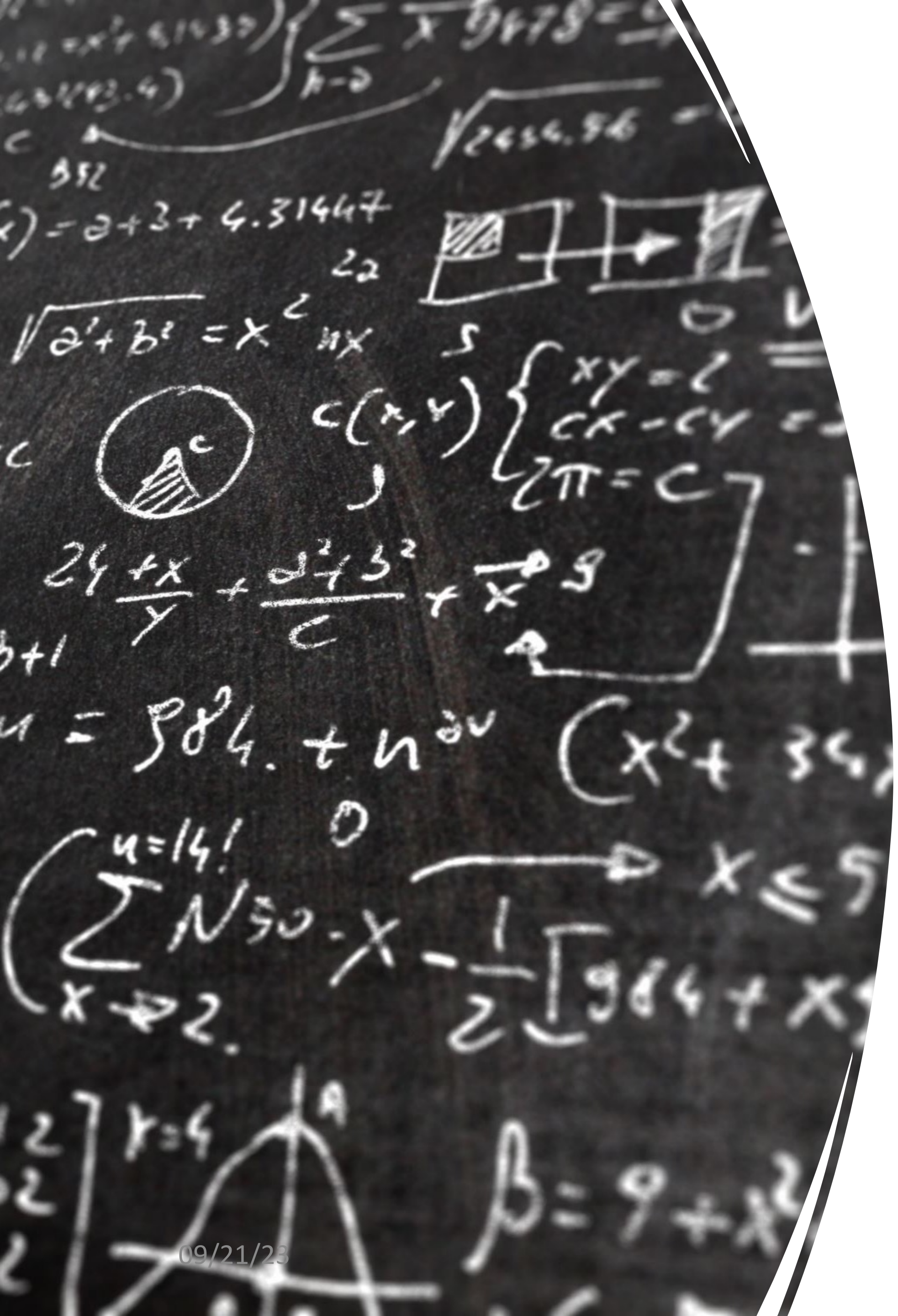


CS 113 – Computer Science I

Lecture 06 – Booleans & Conditionals

Thursday 09/21/2023

Announcements

- HW01:
    - Grading should be done by tomorrow
- HW02 – released
    - Due Tuesday 09/26
- **Read & Follow Instructions**
    - Don’t just skim the labs & homework

Agenda

Review

Con

Unit testing

Verify that method is implemented correctly

Call the method with different inputs and check the results

In a library, we can use the main method to test methods

Top down design

1. Identify features of the program
    1. List them out!
2. Identify verbs and nouns in feature list
    1. Verbs: functions
    2. Nouns: objects/variables
3. Sketch major steps – how features should fit together
    1. Algorithm!
4. Write program skeleton
    1. Include method **stubs** (placeholders for our functions)
    2. method **stub:** empty function with parameters and return type
5. Implement and test method stubs one at a time

Booleans & Conditionals

A new data type: Booleans

- Contains two possible values:
  - `true; false;`
  - `bool isWet = true;`
- Conditional expression

Conditional Expressions & Relational Operators

- Conditional expression produces either `true` or `false`
- Relational Operators:
  - `>`
  - `>=`
  - `<`
  - `<=`
  - `==`
  - `!=`
- Watch out about `==` vs `=`

Exercise: relational expressions

```
int temp = 68;
double val = 10.5;
boolean raining = true;
```

Expression	Value	Type
temp > 80		
val != 5.6		
val >= 10.1		
raining == true		
raining		
raining == false		

Logical Operators

- Way to combine Boolean expressions

- logical Operators:
  - && - and
  - || - or
  - ! - not

Rules of logical operators

1. `X && Y` is true when
    1. Both `X` and `Y` are true

2. `X || Y` is true when
    1. `X` is true or `Y` is true

3. `!X` is true when
    1. `X` is false

4. `!X` false when
    1. `X` is true

Exercise: logical expressions

```
boolean isHappy = true;
boolean knowIt = false;
int temp = 40;
```

Expression	Value	Type
isHappy && knowIt		
isHappy		
isHappy || temp > 80		
isHappy || knowIt		
!knowIt		
isHappy && (temp < 80 || !knowIt)		

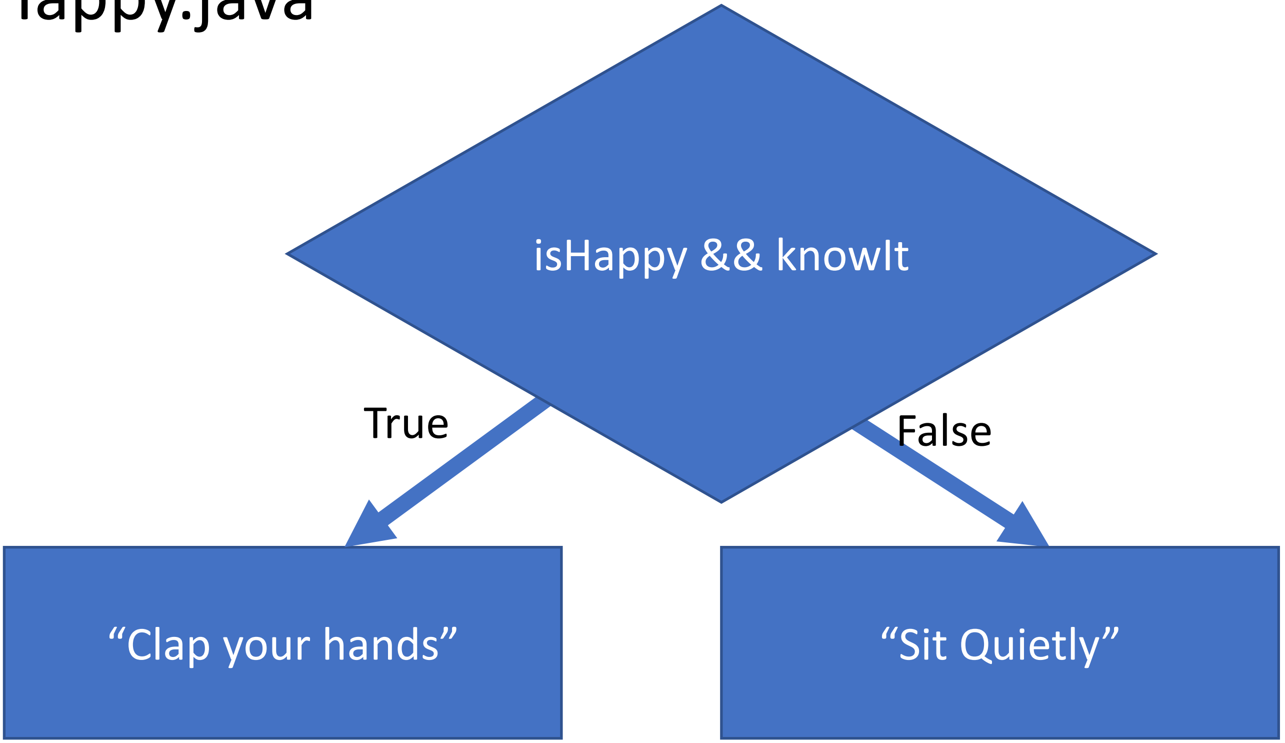
Decision making: if/else

Idea: Branching decision-making based on Boolean expressions

- Example: A **decision tree** for Happy.java

```
if (isHappy && knowIt) {
   System.out.println("Clap your hands!");

} else {
   System.our.println("Sit quietly.");
}
```



Exercise: IsEven

Write a program IsEven which asks the user for an integer and prints whether it is even or not

```
$ java IsEven
Enter an integer: 4
4 is even!

$ java IsEven
Enter an integer: -1
-1 is odd!

$ java IsEven
Enter an integer: 0
0 is even!
```

Decision making: multi-way if statements

```
if (<condition1>) {
  <stmts>
} else if (<condition2>) {
   <stmts>
}
....
else {
   <stmts>
}
```

NOTES:

- Conditions evaluated in order
- First true condition executes
- Only **one** of the conditions can execute!
- the final else statement is optional

Example: Height.java

- Write a program (called Height.java) that determines if a user can ride a rollercoaster.

- Make sure to ask the user for height in inches.
- Prints out a message if they are taller than 5, 4, 3 feet or are too short for the ride

Exercise: Height.java

```
class CheckHeight2 {
  public static void main(String[] args) {
    System.out.print("Enter a height (inches): ");
    int h = Integer.parseInt(System.console().readLine());

    if (h > 36) {
      println("Taller than 3 ft");
    }
    else if (h > 60) {
      println("Taller than 5 ft");
    }
    else if (h > 48) {
      println("Taller than 4 ft");
    }
    else {
      println("Too small for this ride");
    }
  }
}
```

What is the output of this program:

- if the user enters 62 inches?
- if the user enters 10 inches?

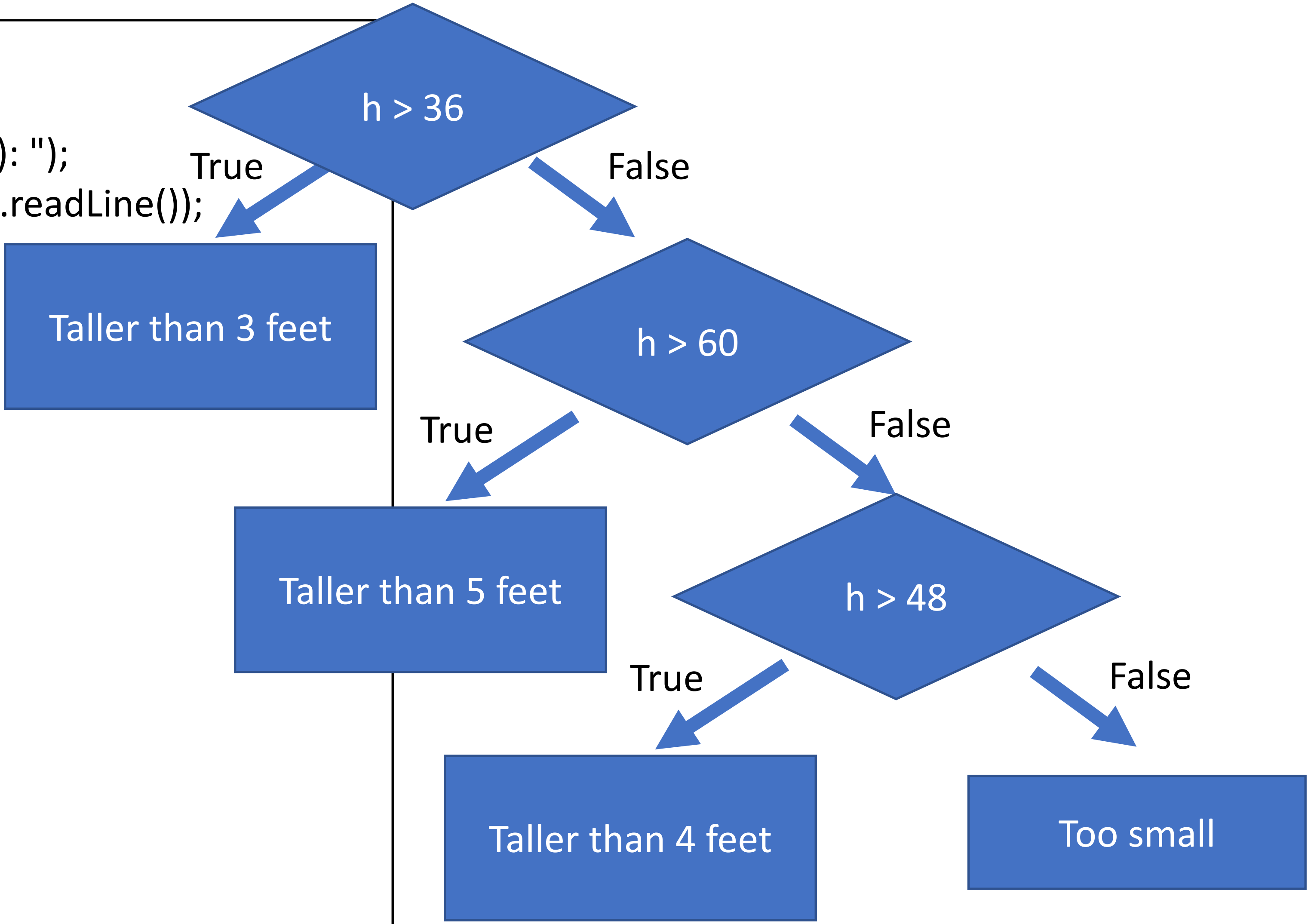
Draw the decision tree for this if statement

Draw the decision tree for this if statement

Exercise: Height.java

```
class CheckHeight2 {
 public static void main(String[] args) {
  System.out.print("Enter a height (inches): ");
  int h = Integer.parseInt(System.console().readLine());

  if (h > 36) {
   println("Taller than 3 ft");
  }
  else if (h > 60) {
   println("Taller than 5 ft");
  }
  else if (h > 48) {
   println("Taller than 4 ft");
  }
  else {
   println("Too small for this ride");
  }
 }
}
```

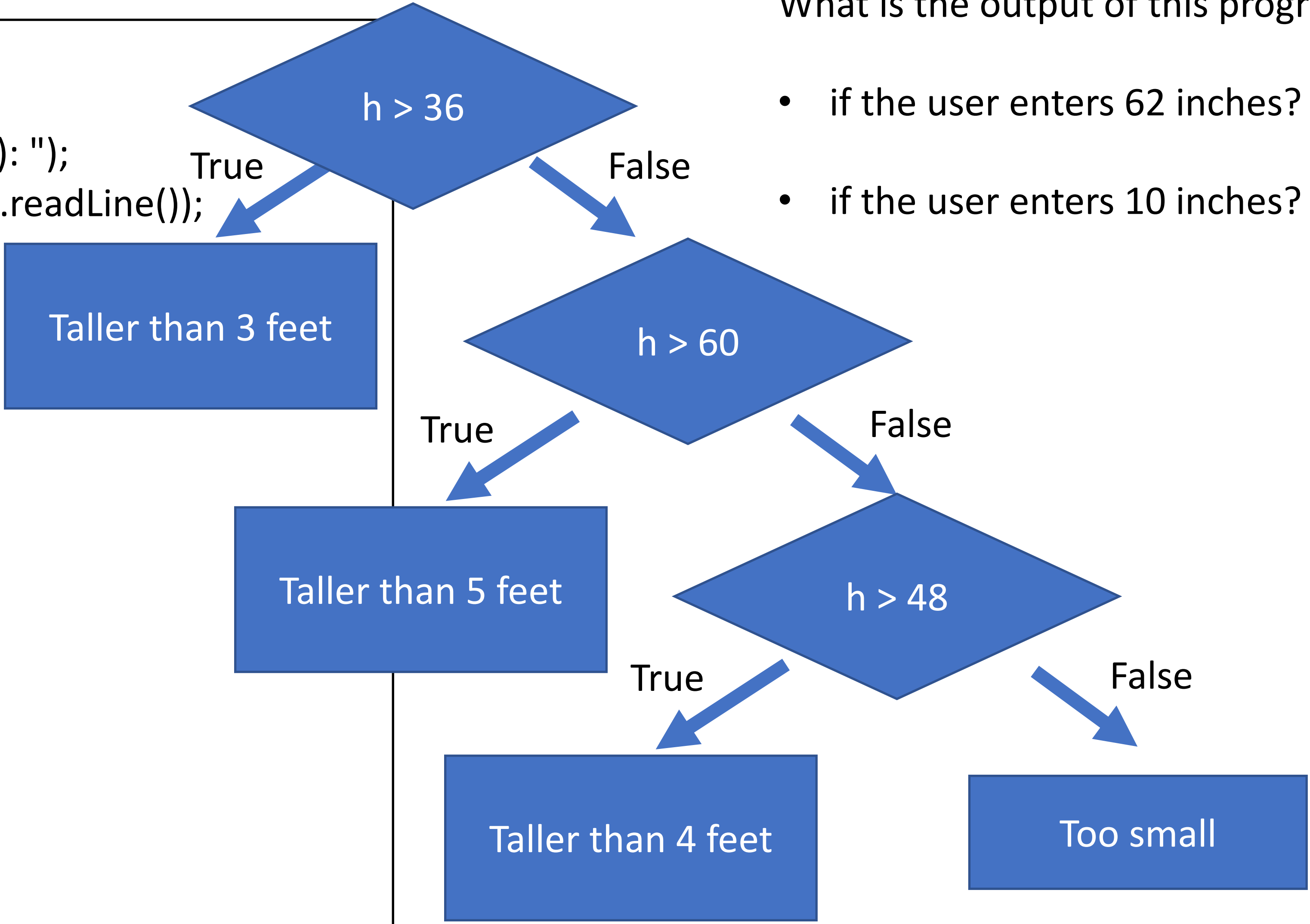


Exercise: Height.java

```
class CheckHeight2 {
  public static void main(String[] args) {
    System.out.print("Enter a height (inches): ");
    int h = Integer.parseInt(System.console().readLine());

    if (h > 36) {
      println("Taller than 3 ft");
    }
    else if (h > 60) {
      println("Taller than 5 ft");
    }
    else if (h > 48) {
      println("Taller than 4 ft");
    }
    else {
      println("Too small for this ride");
    }
  }
}
```

Draw the decision tree for this if statement



What is the output of this program

- if the user enters 62 inches?
- if the user enters 10 inches?

Exercise: Blackjack

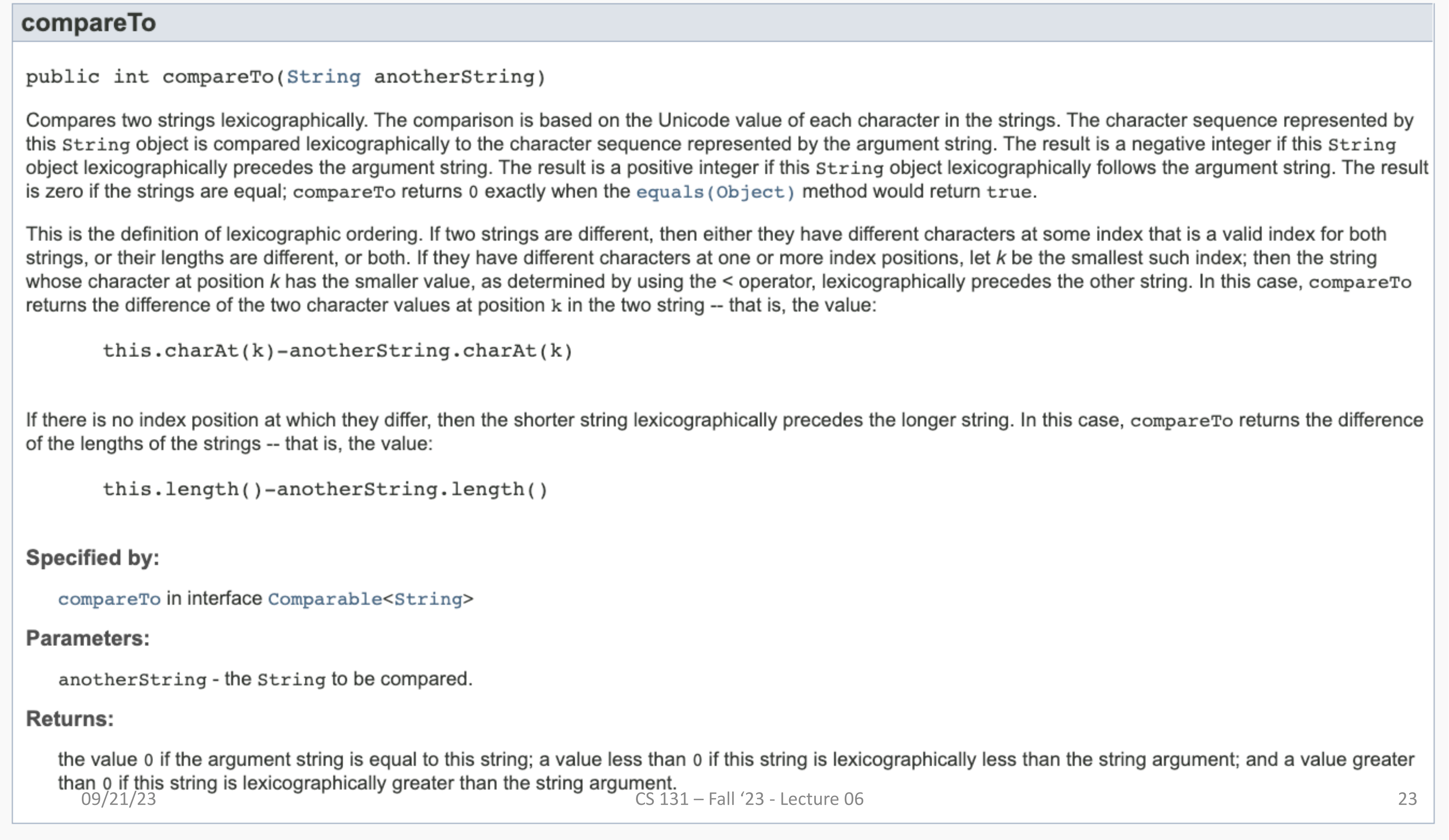
Write a program Blackjack.java which generates a random value between 2 and 21

- If the value is 21, print the value and “Blackjack” to the console
- If the value is between 17 and 20, print the value and “Stand” to the console
- If the value is less than 17, print the value and “Hit me!” to the console

Comparing strings

- In Java, you cannot directly compare strings using ==

- Instead, 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:

```
this.charAt(k)-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:

```
this.length()-anotherString.length()
```

Specified by:

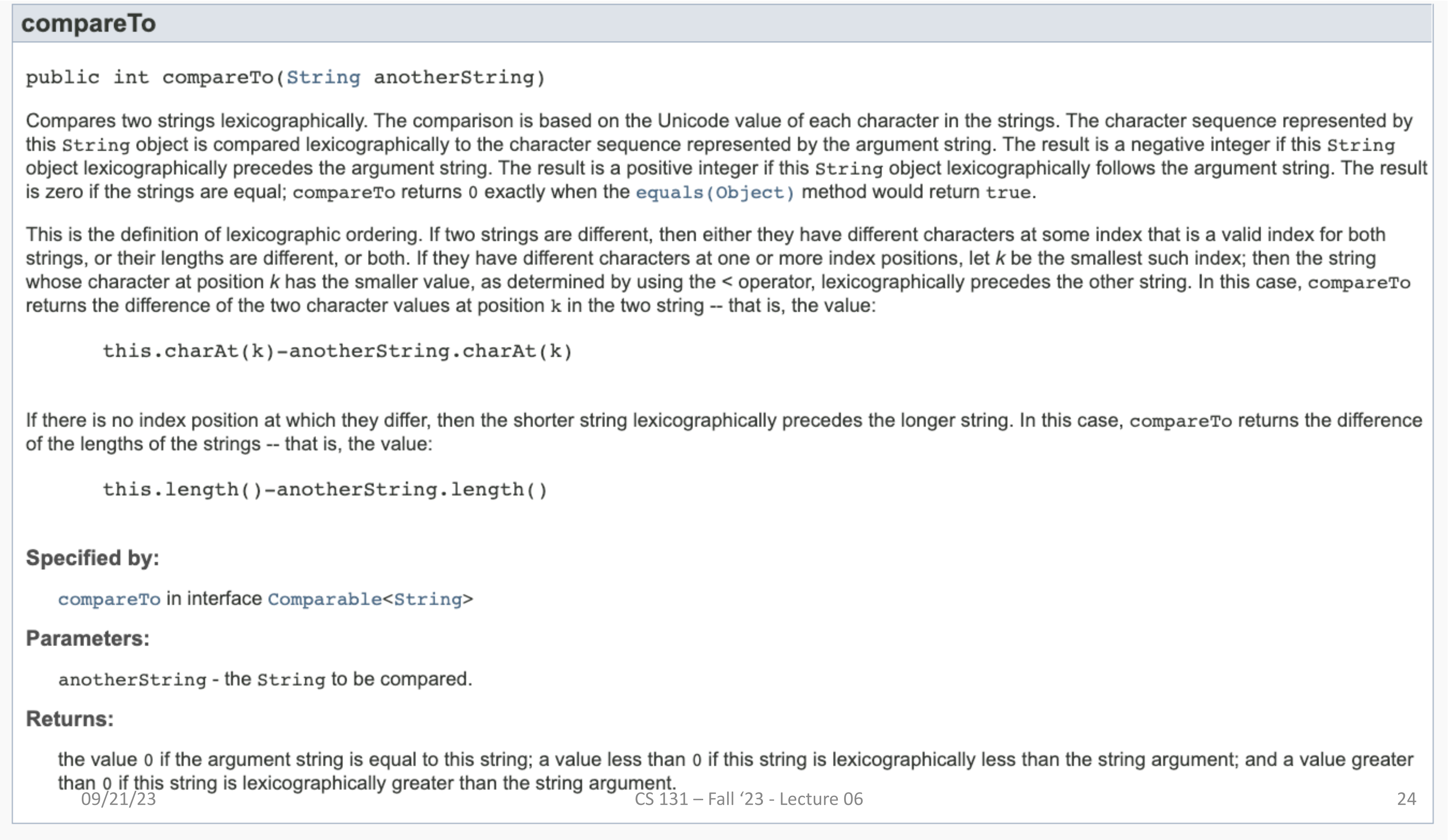
`compareTo` in interface `Comparable<String>`

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:

```
this.charAt(k)-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:

```
this.length()-anotherString.length()
```

Specified by:

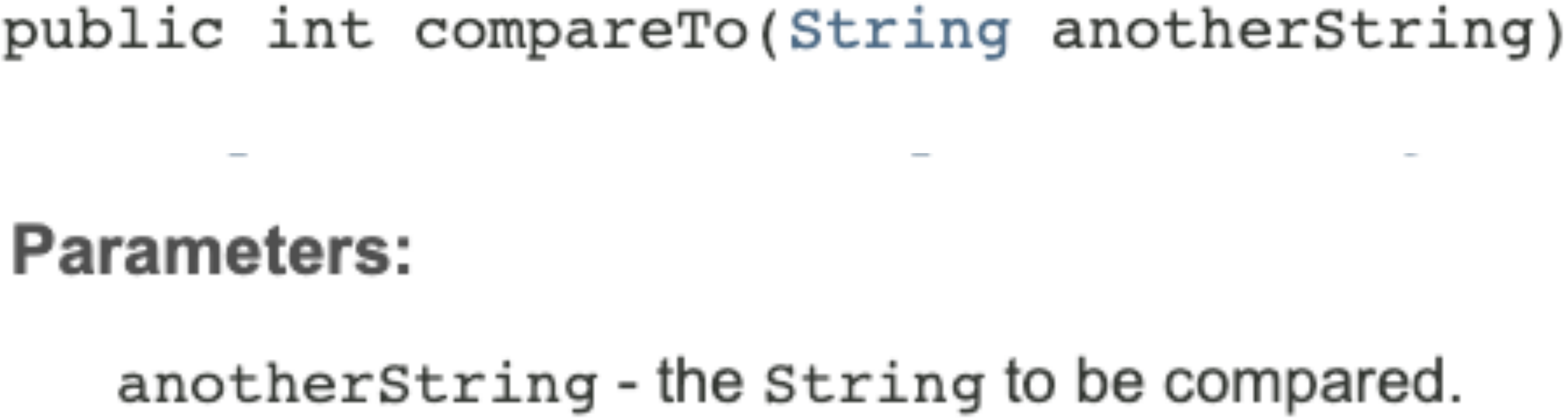
`compareTo` in interface `Comparable<String>`

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

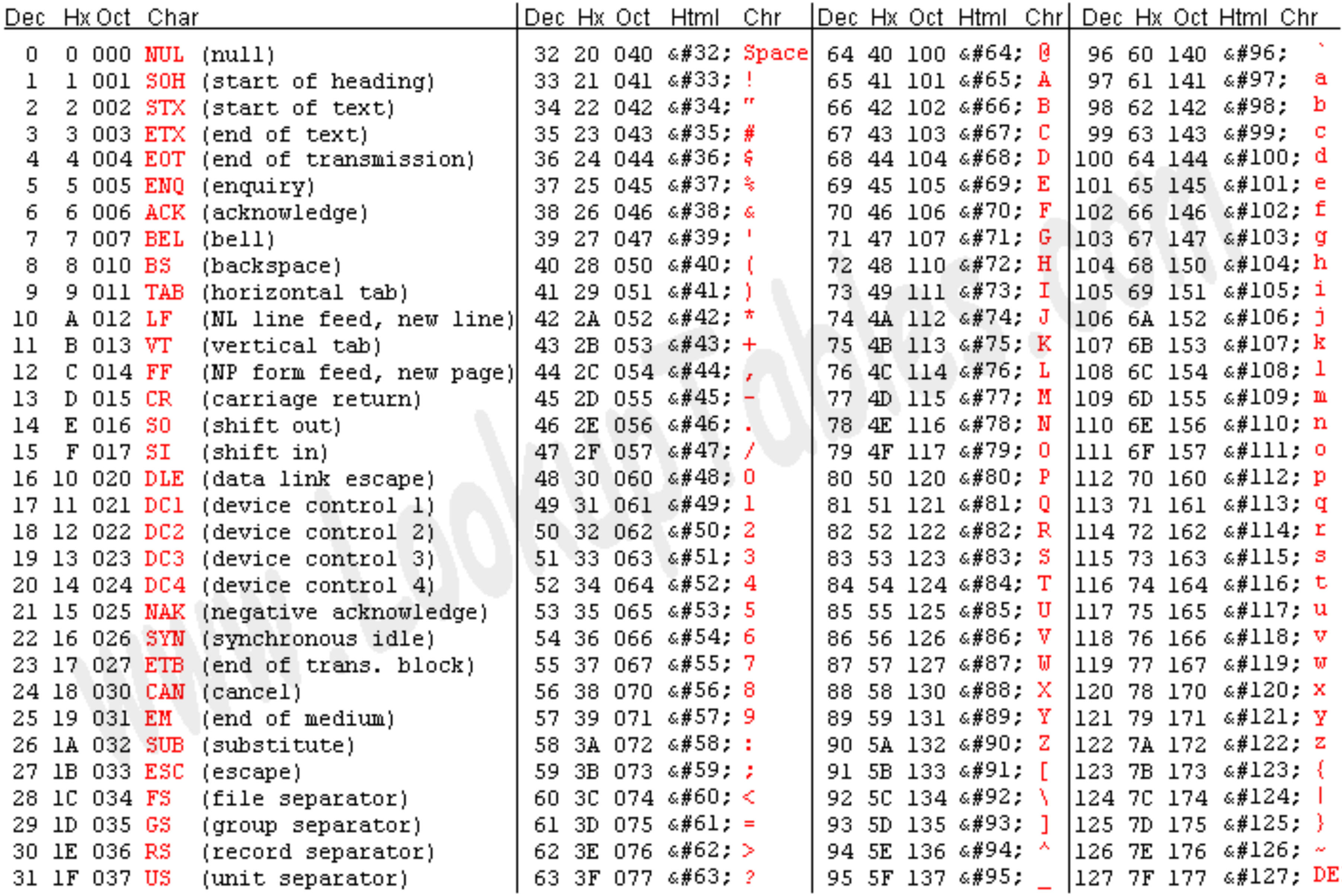
- In Java, you cannot directly compare strings: use **compareTo**

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