CSS 142

Lecture 4

Arkady Retik aretik@uw.edu

TODAY'S CONTENT

1. **Recap + Continue on L3**
2. **Assignments, statements, methods**
3. **Intro to classes and objects; String Class**
4. **Documentation, Coding Style**
5. **Reading**
   - **reading today: Chapter 2 (finish);**
   - **Next week Monday** Chapter 3 (**3.1, 3.2**)
     - **FoR3**

FoR 2:

The order is from most common muddy question to least common muddy questions.

1. The printf() was overall confusing and understanding how to format using the **printf method**. i.e %.2f, %6.2f, %+.2f, %s, etc

2**. IndexOf(String**, startingIndex) and IndexOf() method was confusing

String name = “Mary, Mary quite contrary”
Name.indexOf(“Mary”, 1) ;
This return 6 and not 0 or why it returns 6 was asked.

3. **Escape sequences** and why do we use them was commonly asked too.

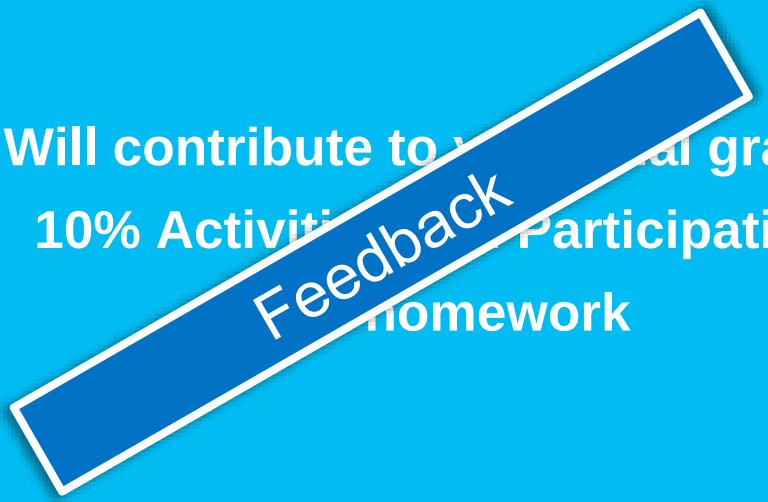
4. What is the difference between **classes, objects, and methods** was asked often.

Hands-on: Class Activity

Will contribute to your final grade:

10% Activities / Participation

Homework



Hands-on: Activity 1

Name/s:

Date:

Instructions: In pairs, work on the following problems **using pencil and paper**.

Problem 1. Write a method that takes as input two integers and returns their sum.

Problem 2. Write a method that takes as input a double and prints it twice.

Problem 3. Write a main method with test calls to the methods you wrote for problems 1 and 2. Predict the output you would get from running your main.

Problem 4. What is the output produced by the following lines of program code?

```
char a, b;
a = 'b';
System.out.println(a);
b = 'c';
System.out.println(b);
a = b;
System.out.println(a);
```

Problem 5. What is the output produced by the following lines of program code? What would be difference if we would use –n instead of n (in line 4).

```
int n = 3;
n++;
System.out.println("n == " + n);
n;  //what will be difference if we use  ++n;
System.out.println("n == " + n);
```

4. Problem 4: 95% of the students got this correct. 5% didn't finish or do it.

5. Problem 5: 20% got it correct by saying it results in an "error". 75% got it incorrect by saying it prints "n==4". 5% didn't get to finish.

Problem 4. What is the output produced by the following lines of program code?

```
char a, b;
a = 'b';
System.out.println(a);
b = 'c';
System.out.println(b);
a = b;
System.out.println(a);
```

Problem 5. What is the output produced by the following lines of program code? What would be difference if we would use –n instead of n (in line 4).

```
int n = 3;
n++;
System.out.println("n == " + n);
n;  //what will be difference if we use  –n;
System.out.println("n == " + n);
```

1. Problem 1.
Looking for this general format (**correct**): (~10%)

```
public static int sum(int x, int y){
    return x+y;
}
```

The students didn't make a method that takes two integer as the parameters **(incorrect**). (~85%)

```
public static void sum(){
    int x = 5;
    int y = 1;
    System.out.println(x+y);
}
```

Also, most had a 'void' as the return type and not an integer. This resulted in no "return" to return the sum. (~85%) same amount as no parameter

```
public static void sum(){
    ....
    System.out.println(x+y);
}
```

~5% remaining didn't know how to do problem 1 or understand methods. i.e. They used main(...) instead.

2. Problem 2.
Looking for this general format **(correct**): (~10%)

```
public static void printTwice(double x){
    System.out.println(x);
    System.out.println(x);
}
```

The student didn't have anything in the parameters **(incorrect**). (~85%)

```
public static void printTwice(){
    double x = 4.0;
    System.out.println(x);
    System.out.println(x);
}
```

~5% remaining didn't know how to do problem 2 or understand methods. i.e They used main(...) instead.

3. Problem 3.
Looking for this general format (correct): (~10%)

```
public static void main(String []args){
    sum(1,3);
    printTwice(5.0);
}
```

The student got this wrong most likely if they got problems 1 and 2 wrong by passing in nothing into the parameters. (~85%)

~5% remaining didn't know how call the method in main.

Shorthand Assignment Statements

- Shorthand assignment notation combines the *assignment operator* (**=**) and an ***arithmetic op****erator* (some examples of what `Op` can be are **+**, **-**, *****, **/**, or **%**)
- It is used to change the value of a variable by adding, subtracting, multiplying, or dividing by a specified value
- The general form is

```
Variable Op = Expression
```

  - which is equivalent to

```
Variable = Variable Op (Expression)
```

  - The `Expression` can be another variable, a constant, or a more complicated expression

Examples of Shorthand Assignment Statements

Example:	Equivalent To:
count += 3;	count = count + 3;
sum -= discount;	sum = sum – discount;
bonus *= 2;	bonus = bonus * 2;
time /= rushFactor;	time = time / rushFactor;
change %= 55;	change = change % 55;
amount *= count1 + count2;	amount = amount * (count1 + count2);

Assignment Compatibility

- In general, the value of one type **cannot** be stored in a variable of another type
  - `int intVariable = 2.99; //Illegal`
  - The above example results in a type mismatch because a `double` value cannot be stored in an `int` variable
- However, there are exceptions to this
  - `double doubleVariable = 2;`
  - For example, an `int` value can be stored in a `double` type

Why is that?

Size of the memory allocation

Assignment Compatibility and Type Cast

- More generally, a value of any type in the following list can be assigned to a variable of any type that appears to the right of it
  - `byte→short→int→long→float→double`
  - `char`
  - Note that as your move down the list from left to right, the range of allowed values for the types becomes larger
- An explicit ***type cast*** is required to assign a value of one type to a variable whose type appears to the left of it on the above list (e.g., `double` to `int`)
- **Note** that in Java an `int` **cannot** be assigned to a variable of type `boolean`, nor can a `boolean` be assigned to a variable of type `int`

In some languages
TRUE==1;
FALSE==0;

Constants

- ***Constant*** (or ***literal***): An item in Java which has one specific value that **cannot change**
  - Constants of an integer type may not be written with a decimal point (e.g., `10`)
  - Constants of a floating-point type can be written in ordinary decimal fraction form (e.g., `367000.0` or `0.000589`)
  - Constant of a floating-point type can also be written in *scientific* (or *floating-point*) *notation* (e.g., `3.67e5` or `5.89e-4`)
    - Note that the number before the `e` may contain a decimal point, but the number after the `e` may not

Constants

- Constants of type **`char`** are expressed by placing a **single character** in single quotes (e.g., **`'Z'`**)
- Constants for strings of characters are enclosed by **double quotes** (e.g., **`"Welcome to Java"`**)
- There are only two **`boolean`** type constants, **`true`** and **`false`**
  - Note that they **must be spelled with all lowercase letters**

Arithmetic Operators and Expressions

Precedence Rules

Display 1.3 Precedence Rules

Highest Precedence

First: the unary operators: +, −, ++, −−, and !

Second: the binary arithmetic operators: *, /, and %

Third: the binary arithmetic operators: + and −

Lowest Precedence

Precedence and Associativity Rules

- Unary operators of equal precedence are grouped right-to-left

  `+-+rate` is evaluated as `+(-(+rate))`

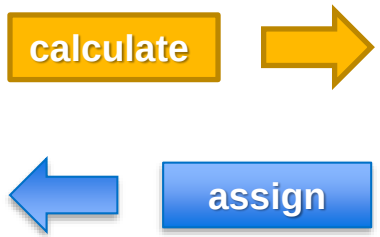
- Binary operators of equal precedence are grouped left-to-right

  `base + rate + hours` is evaluated as

  `(base + rate) + hours`

- Exception: A string of **assignmen**t operators is grouped right-to-left

  `n1 = n2 = n3;` is evaluated as `n1 = (n2 = n3);`



Pitfall: Round-Off Errors in Floating-Point Numbers

- Floating point numbers are only approximate quantities
  - Mathematically, the floating-point number 1.0/3.0 is equal to 0.3333333 . . .
  - A computer has a finite amount of storage space
    - It may store 1.0/3.0 as something like 0.3333333333, which is slightly smaller than one-third
  - Computers actually store numbers in binary notation, but the consequences are the same: floating-point numbers may lose accuracy

What does this means for us?

Integer and Floating-Point Division

- When one or both operands are a floating-point type, division results in a floating-point type
  - `15.0/2` evaluates to `7.5`
- When both operands are integer types, division results in an integer type
  - Any fractional part is discarded
  - The number is not rounded
    - `15/2` evaluates to `7`
- Be careful to make at least one of the operands a floating-point type if the fractional portion is needed

The Modulo Operator: `%`

- The Module (or Mod) `%` operator is used with operands of type **`int`** to recover the information lost after performing integer division
  - **`15/2`** evaluates to the quotient **`7`**
  - **`15%2`** evaluates to the remainder **`1`**
- The `%` operator can be used to count by 2's, 3's, or any other number
  - To count by twos, perform the operation **`number`** `%` **`2`**, and when the result is **`0`**, **`number`** is even

Type Casting

- A ***type cast*** takes a value of one type and produces a value of another type with an "equivalent" value
  - If `n` and `m` are integers to be divided, and the fractional portion of the result must be preserved, at least one of the two must be type cast to a floating-point type **before** the division operation is performed
    - `double anws = n / (double)m;`
  - Note that the desired type is placed inside parentheses immediately in front of the variable to be cast
  - Note also that the type and value of the variable to be cast does not change
    - i.e. `m` will remain to be integer

More Details About Type Casting

- When type casting from a floating-point to an integer type, the number is **truncated**, not rounded
  - `(int)2.9` evaluates to `2`, not `3`
- When the value of an integer type is assigned to a variable of a floating-point type, Java performs an automatic type cast called a *type coercion*
  - `double d = 5;`
- In contrast, it is illegal to place a `double` value into an `int` variable without an explicit type cast
  - `int i = 5.5; // Illegal`
  - `int i = (int)5.5 // Correct`

Increment and Decrement Operators

- The *increment operator* (`++`) adds one to the value of a variable
  - If `n` is equal to `2`, then `n++` or `++n` will change the value of `n` to `3`
- The *decrement operator* (`--`) subtracts one from the value of a variable
  - If `n` is equal to `4`, then `n--` or `--n` will change the value of `n` to `3`

Increment and Decrement Operators

- When either operator precedes its variable, and is part of an expression, then the expression is evaluated using the changed value of the variable
  - If `n` is equal to `2`, then `2*(++n)` evaluates to `6`
- When either operator follows its variable, and is part of an expression, then the expression is evaluated **using the original value of the variable**, and only then is the variable value changed
  - If `n` is equal to `2`, then `2*(n++)` evaluates to `4`

Appendix 2
in Absolute Java

Precedence and Associativity Rules 2

PRECEDENCE	ASSOCIATIVITY
From highest at top to lowest at bottom. Operators in the same group have equal precedence.	
Dot operator, array indexing, and method invocation: `.`, `[]`, `()`	Left to right
`++` (postfix, as in `x++`), `--` (postfix)	Right to left
The unary operators: `+`, `-`, `++` (prefix, as in `++x`), `--` (prefix), `!`, `~` (bitwise complement)[1]	Right to left
`new` and type casts (*Type*)	Right to left
The binary operators `*`, `/`, `%`	Left to right
The binary operators `+`, `-`	Left to right
The binary operators `<<`, `>>`, `>>>` (shift operators)[1]	Left to right
The binary operators `<`, `>`, `<=`, `>=`, `instanceof`	Left to right
The binary operators `==`, `!=`	Left to right
The binary operator `&`	Left to right
The binary operator `^` (exclusive or)[1]	Left to right
The binary operator `|`	Left to right
The binary operator `&&`	Left to right
The binary operator `||`	Left to right
The ternary operator (conditional operator) `?:`	Right to left
The assignment operators `=`, `*=`, `/=`, `%=`, `+=`, `-=`, `&=`, `|=`, `^=`, `<<=`, `>>=`, `>>>=`	Right to left

[1] Not discussed in this book.

The Class `String`

The Class `String`

- There is no primitive type for strings in Java
- The class `String` is a **predefined class** in Java that is used to store and process strings
- **Objects** of type `String` are made up of strings of characters that are written within double quotes
  - Any quoted string is a constant of type `String`

```
"Live long and prosper."
```

- A variable of type `String` can be given the value of a `String` object

```
String blessing = "Live long and prosper.";
```

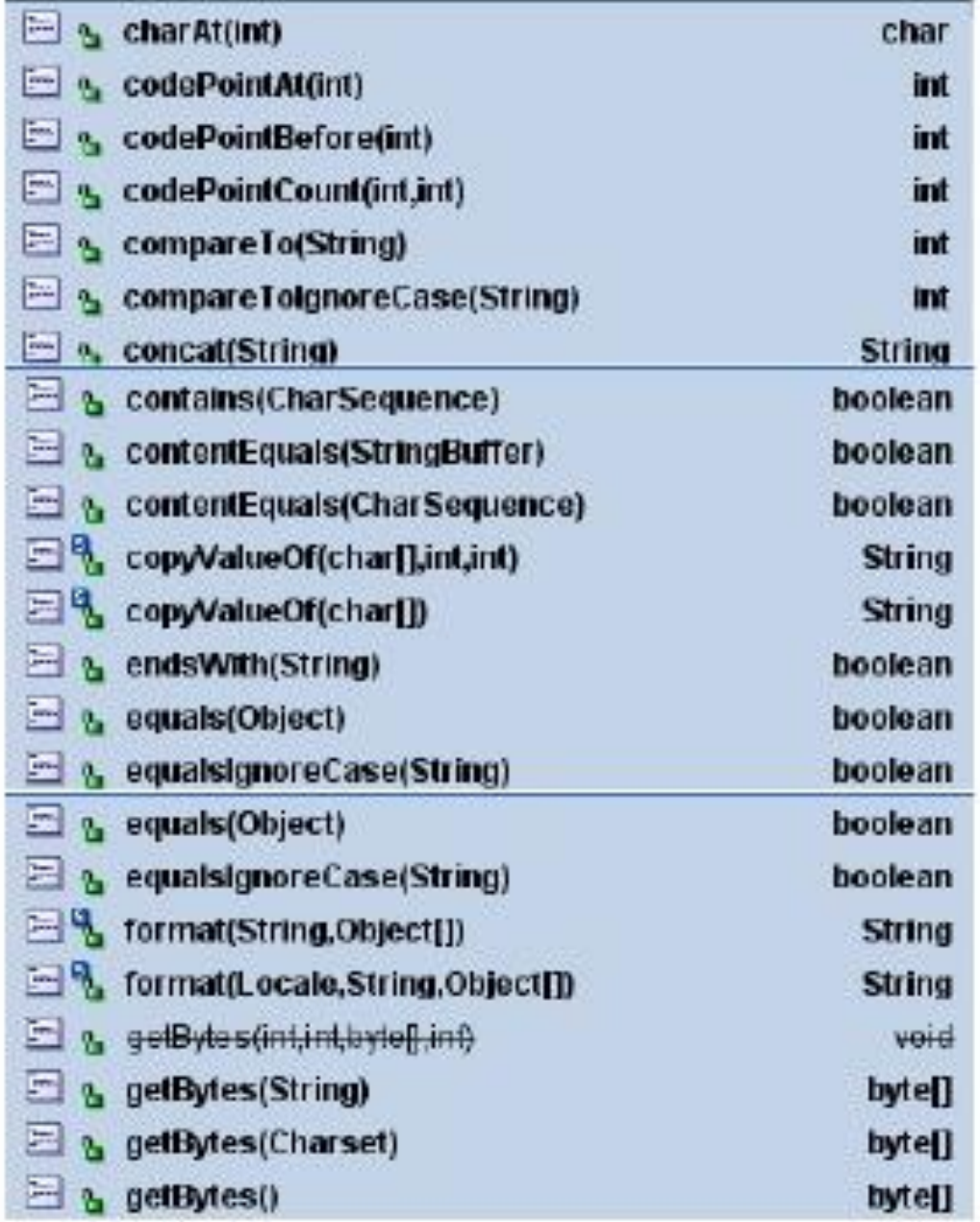
Classes, Objects, and Methods

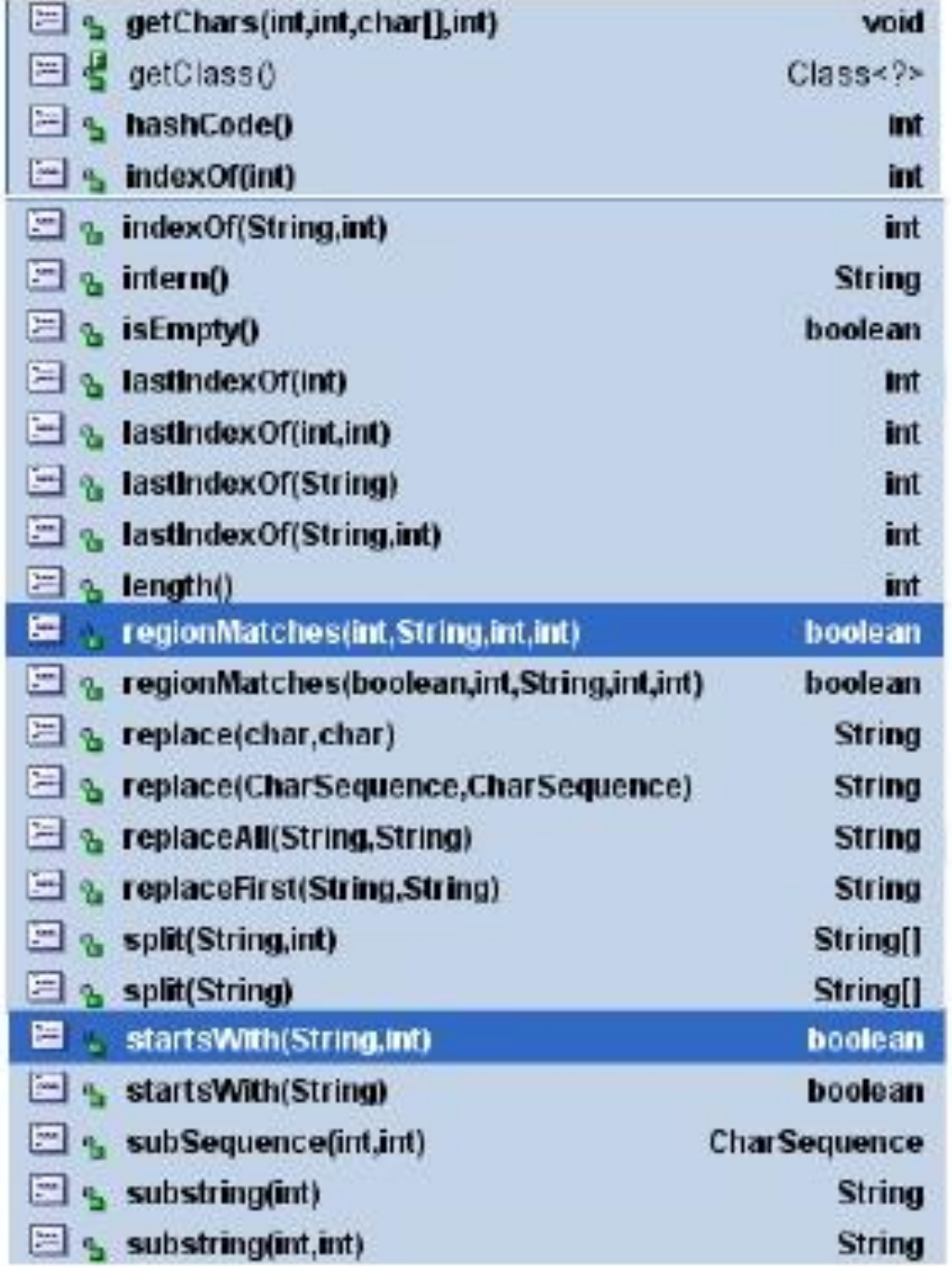
- A ***Class*** is the name for a **type** whose values are objects
  - You can think of *Class* as a *template*
- ***Objects*** are entities that store data and take actions
  - Objects of the `String` class store data consisting of strings of characters
- The actions that an object can take are called ***methods***
  - Methods can return a value of a single type and/or perform an action
  - All objects within a class have the same methods, but each can have different data values

Classes, Objects, and Methods

- *Invoking* or *calling a method*: a method is called into action by writing the name of the calling object, followed by a dot, followed by the method name, followed by parentheses
  - This is sometimes referred to as *sending a message to the object*
  - The parentheses contain the information (if any) needed by the method
  - This information is called an *argument* (or *arguments*)

- Remember example in L2? What's object and what's method here?
  - **`System.out.println ("Hello CSS 161");`**





String Methods

String Methods

- The **`String`** class contains many useful methods for string-processing applications
  - A **`String`** method is called by writing a **`String`** object, a dot, the name of the method, and a pair of parentheses to enclose any arguments
  - If a **`String`** method returns a value, then it can be placed anywhere that a value of its type can be used, i.e.

```
String greeting = "Hello";

int count = greeting.length();

System.out.println("Length is " + greeting.length());

[what will be the results ]?
```

  - Note: *length i*s different from the *position* and *index*. Always count from zero when referring to the *position* or *index* of a character in a string

Some Methods in the Class `String` (Part 1 of 8)*

Display 1.4 Some Methods in the Class `String`

```
int length()
```

Returns the length of the calling object (which is a string) as a value of type `int`.

EXAMPLE

After program executes `String greeting = "Hello!";`
`greeting.length()` returns 6.

```
boolean equals(Other_String)
```

Returns `true` if the calling object string and the *Other_String* are equal. Otherwise, returns `false`.

EXAMPLE

After program executes `String greeting = "Hello";`
`greeting.equals("Hello")` returns `true`
`greeting.equals("Good-Bye")` returns `false`
`greeting.equals("hello")` returns `false`

Note that case matters. `"Hello"` and `"hello"` are not equal because one starts with an uppercase letter and the other starts with a lowercase letter.

(continued)

* All following 8 examples are from Savitch “Absolute Java” Copyright Pearson Addison-Wesley..

Some Methods in the Class `String` (Part 2 of 8)

Display 1.4 Some Methods in the Class `String`

`boolean equalsIgnoreCase(`*Other_String*`)`

Returns `true` if the calling object string and the *Other_String* are equal, considering uppercase and lowercase versions of a letter to be the same. Otherwise, returns `false`.

EXAMPLE

After program executes `String name = "mary!";`
`greeting.equalsIgnoreCase("Mary!")` returns `true`

`String toLowerCase()`

Returns a string with the same characters as the calling object string, but with all letter characters converted to lowercase.

EXAMPLE

After program executes `String greeting = "Hi Mary!";`
`greeting.toLowerCase()` returns `"hi mary!"`.

(continued)

Some Methods in the Class `String` (Part 3 of 8)

Display 1.4 Some Methods in the Class `String`

`String toUpperCase()`

Returns a string with the same characters as the calling object string, but with all letter characters converted to uppercase.

EXAMPLE

After program executes `String greeting = "Hi Mary!";`
`greeting.toUpperCase()` returns `"HI MARY!"`.

`String trim()`

Returns a string with the same characters as the calling object string, but with leading and trailing white space removed. Whitespace characters are the characters that print as white space on paper, such as the blank (space) character, the tab character, and the new-line character `'\n'`.

EXAMPLE

After program executes `String pause = "  Hmm  ";`
`pause.trim()` returns `"Hmm"`.

(continued)

Some Methods in the Class `String` (Part 4 of 8)

Display 1.4 Some Methods in the Class `String`

`char charAt(Position)`

Returns the character in the calling object string at the *Position*. Positions are counted 0, 1, 2, etc.

EXAMPLE

After program executes `String greeting = "Hello!";`
`greeting.charAt(0)` returns `'H'`, and
`greeting.charAt(1)` returns `'e'`.

`String substring(Start)`

Returns the substring of the calling object string starting from *Start* through to the end of the calling object. Positions are counted 0, 1, 2, etc. Be sure to notice that the character at position *Start* is included in the value returned.

EXAMPLE

After program executes `String sample = "AbcdefG";`
`sample.substring(2)` returns `"cdefG"`.

(continued)

Some Methods in the Class `String` (Part 5 of 8)

Display 1.4 Some Methods in the Class `String`

`String substring(`*Start*`, `*End*`)`

Returns the substring of the calling object string starting from position *Start* through, but not including, position *End* of the calling object. Positions are counted 0, 1, 2, etc. Be sure to notice that the character at position *Start* is included in the value returned, but the character at position *End* is not included.

EXAMPLE

After program executes `String sample = "AbcdefG";`
`sample.substring(2, 5)` returns `"cde"`.

`int indexOf(`*A_String*`)`

Returns the index (position) of the first occurrence of the string *A_String* in the calling object string. Positions are counted 0, 1, 2, etc. Returns `-1` if *A_String* is not found.

EXAMPLE

After program executes `String greeting = "Hi Mary!";`
`greeting.indexOf("Mary")` returns `3`, and
`greeting.indexOf("Sally")` returns `-1`.

(continued)

Some Methods in the Class `String` (Part 6 of 8)

Display 1.4 Some Methods in the Class `String`

`int indexOf(A_String, Start)`

Returns the index (position) of the first occurrence of the string *A_String* in the calling object string that occurs at or after position *Start*. Positions are counted 0, 1, 2, etc. Returns −1 if *A_String* is not found.

EXAMPLE

After program executes `String name = "Mary, Mary quite contrary";`
`name.indexOf("Mary", 1)` returns 6.
The same value is returned if 1 is replaced by any number up to and including 6.
`name.indexOf("Mary", 0)` returns 0.
`name.indexOf("Mary", 8)` returns −1.

`int lastIndexOf(A_String)`

Returns the index (position) of the last occurrence of the string *A_String* in the calling object string. Positions are counted 0, 1, 2, etc. Returns −1, if *A_String* is not found.

EXAMPLE

After program executes `String name = "Mary, Mary, Mary quite so";`
`greeting.indexOf("Mary")` returns 0, and
`name.lastIndexOf("Mary")` returns 12.

What are these methods are useful for?

(continued)

Some Methods in the Class `String` (Part 7 of 8)

Display 1.4 Some Methods in the Class `String`

`int compareTo(A_String)`

Compares the calling object string and the string argument to see which comes first in the lexicographic ordering. Lexicographic order is the same as alphabetical order but with the characters ordered as in Appendix 3. Note that in Appendix 3 all the uppercase letters are in regular alphabetical order and all the lowercase letters are in alphabetical order, but all the uppercase letters precede all the lowercase letters. So, lexicographic ordering is the same as alphabetical ordering provided both strings are either all uppercase letters or both strings are all lowercase letters. If the calling string is first, it returns a negative value. If the two strings are equal, it returns zero. If the argument is first, it returns a positive number.

EXAMPLE

After program executes `String entry = "adventure";`
`entry.compareTo("zoo")` returns a negative number,
`entry.compareTo("adventure")` returns `0`, and
`entry.compareTo("above")` returns a positive number.

(continued)

Some Methods in the Class `String` (Part 8 of 8)

Display 1.4 Some Methods in the Class `String`

`int compareToIgnoreCase(A_String)`

Compares the calling object string and the string argument to see which comes first in the lexicographic ordering, treating uppercase and lowercase letters as being the same. (To be precise, all uppercase letters are treated as if they were their lowercase versions in doing the comparison.) Thus, if both strings consist entirely of letters, the comparison is for ordinary alphabetical order. If the calling string is first, it returns a negative value. If the two strings are equal ignoring case, it returns zero. If the argument is first, it returns a positive number.

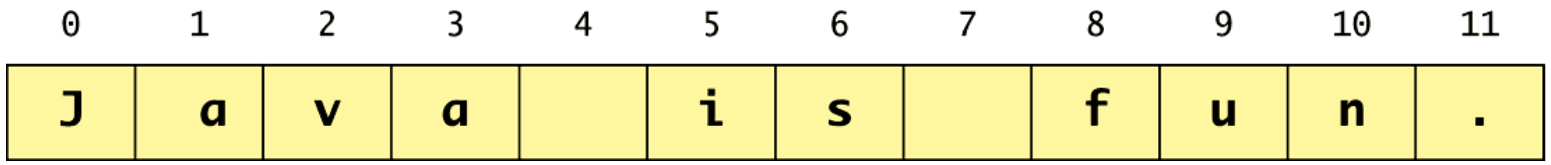
EXAMPLE

After program executes `String entry = "adventure";`
`entry.compareToIgnoreCase("Zoo")` returns a negative number,
`entry.compareToIgnoreCase("Adventure")` returns `0`, and
`"Zoo".compareToIgnoreCase(entry)` returns a positive number.

String Indexes

Display 1.5 String Indexes

The 12 characters in the string `"Java is fun."` have indexes `0` through `11`.



Notice that the blanks and the period count as characters in the string.

Escape Sequences

- A backslash (\) immediately preceding a character (i.e., without any space) denotes an *escape sequence* or an ***escape character***
  - The character following the backslash does not have its usual meaning
  - Although it is formed using two symbols, it is regarded as a single character
  - Examples:

    \n; \r; \\; \’

Escape Sequences

Display 1.6 Escape Sequences

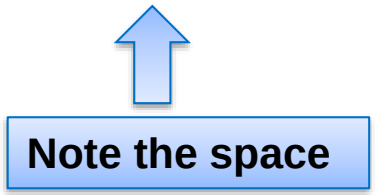
```
\" Double quote.
\' Single quote.
\\ Backslash.
\n New line. Go to the beginning of the next line.
\r Carriage return. Go to the beginning of the current line.
\t Tab. White space up to the next tab stop.
```

Backslash vs Forwardslash

Unix vs Windows

String Processing

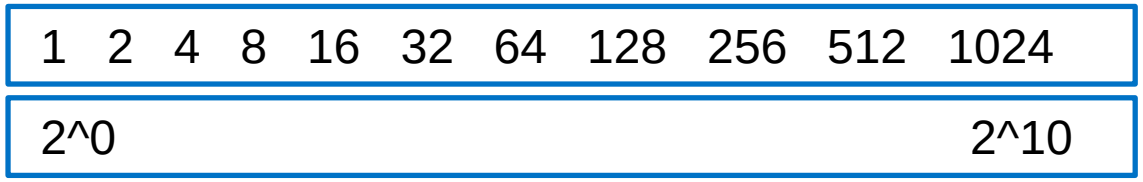
- A **`String`** object in Java is considered to be immutable, i.e., the characters it contains cannot be changed
- There is another class in Java called **`StringBuffer`** that has methods for editing its string objects
- However, it is possible to change the value of a **`String`** variable by using an assignment statement
  - `String name = "Savitch";`
  - `name = "Walter " + name;`



Character Sets

- *ASCII*: A character set used by many programming languages that contains all the characters normally used on an English-language keyboard, plus a few special characters
  - Each character is represented by a particular number
- *Unicode*: A character set used by the Java language that includes all the ASCII characters plus many of the characters used in languages with a different alphabet from English

IBM: type writer => 7bits ASCII set => 8 Bits => UNICODE | 2 words [8 + 8 Bytes]



Appendix 3
in *Absolute Java*

ASCII Character Set 3

The characters shown here form the ASCII character set, which is the subset of the Unicode character set that is commonly used by English speakers. The numbering is the same whether the characters are considered to be members of the Unicode character set or of the ASCII character set. Character number 32 is the blank. Printable characters only are shown.

32		56	8	80	P	104	h
33	!	57	9	81	Q	105	i
34	"	58	:	82	R	106	j
35	#	59	;	83	S	107	k
36	$	60	<	84	T	108	l
37	%	61	=	85	U	109	m
38	&	62	>	86	V	110	n
39	'	63	?	87	W	111	o
40	(	64	@	88	X	112	p
41	)	65	A	89	Y	113	q
42	*	66	B	90	Z	114	r
43	+	67	C	91	[	115	s
44	,	68	D	92	\	116	t
45	-	69	E	93	]	117	u
46	.	70	F	94	^	118	v
47	/	71	G	95	-	119	w
48	0	72	H	96	`	120	x
49	1	73	I	97	a	121	y
50	2	74	J	98	b	122	z
51	3	75	K	99	c	123	{
52	4	76	L	100	d	124	|
53	5	77	M	101	e	125	}
54	6	78	N	102	f	126	~
55	7	79	O	103	g		

Example of Java to String conversion:

http://beginnersbook.com/2015/05/java-ascii-to-string-conversion/

Full ASCII Table

Dec	Hex	Oct	Chr	Dec	Hex	Oct	HTML	Chr	Dec	Hex	Oct	HTML	Chr	Dec	Hex	Oct	HTML	Chr
0	0	000	NULL	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	Start of Header	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	Start of Text	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	End of Text	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	End of Transmission	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	Enquiry	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	Acknowledgment	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	Bell	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	Backspace	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	Horizontal Tab	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	Line feed	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	Vertical Tab	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	Form feed	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	Carriage return	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	Shift Out	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	Shift In	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	Data Link Escape	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	Device Control 1	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	Device Control 2	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	Device Control 3	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	Device Control 4	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	Negative Ack.	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	Synchronous idle	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	End of Trans. Block	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	Cancel	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	End of Medium	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	Substitute	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	Escape	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	File Separator	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	Group Separator	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	Record Separator	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	Unit Separator	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	Del

Full ASCII Table

128	Ç	144	É	160	á	176	░	193	┴	209	╤	225	ß	241	±
129	ü	145	æ	161	í	177	▒	194	┬	210	╥	226	Γ	242	≥
130	é	146	Æ	162	ó	178	▓	195	├	211	╙	227	π	243	≤
131	â	147	ô	163	ú	179	│	196	─	212	╘	228	Σ	244	⌠
132	ä	148	ö	164	ñ	180	┤	197	┼	213	╒	229	σ	245	⌡
133	à	149	ò	165	Ñ	181	╡	198	╞	214	╓	230	µ	246	÷
134	å	150	û	166	ª	182	╢	199	╟	215	╫	231	τ	247	≈
135	ç	151	ù	167	º	183	╖	200	╚	216	╪	232	Φ	248	°
136	ê	152	ÿ	168	¿	184	╕	201	╔	217	┘	233	Θ	249	∙
137	ë	153	Ö	169	⌐	185	╣	202	╩	218	┌	234	Ω	250	·
138	è	154	Ü	170	¬	186	║	203	╦	219	█	235	δ	251	√
139	ï	156	£	171	½	187	╗	204	╠	220	▄	236	∞	252	ⁿ
140	î	157	¥	172	¼	188	╝	205	═	221	▌	237	φ	253	²
141	ì	158	₧	173	¡	189	╜	206	╬	222	▐	238	ε	254	■
142	Ä	159	ƒ	174	«	190	╛	207	╧	223	▀	239	∩	255	
143	Å	192	└	175	»	191	┐	208	╨	224	α	240	≡		

Source: www.asciitable.com

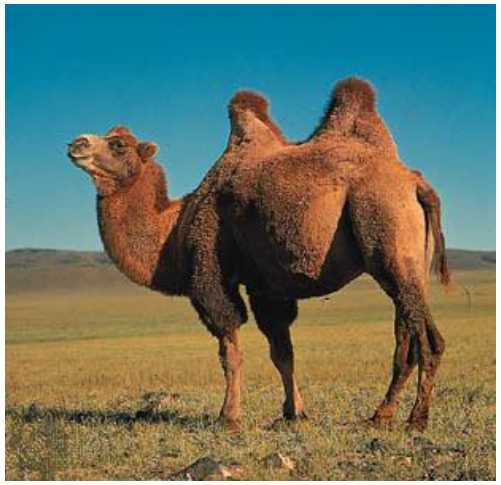
Intro to Java Coding Style

Naming Constants

- Instead of using "anonymous" numbers in a program, always declare them as named constants, and use their name instead
  - `public static final int INCHES_PER_FOOT = 12;`
  - `public static final double RATE = 0.14;`
  - This prevents a value from being changed inadvertently
  - It has the added advantage that when a value must be modified, it need only be changed in one place
  - Note the naming convention for constants: **Use all uppercase letters, and designate word boundaries with an underscore character**

Naming Variables, Classes, Methods, and Objects

- Start the names of **variables, ~~classes~~, methods, and objects** with a lowercase letter, indicate "word" boundaries with an uppercase letter, and restrict the remaining characters to digits and lowercase letters
  - `topSpeed` `bankRate1` `timeOfArrival` } Camel Style
- Start the **names of classes** with an uppercase letter and, otherwise, adhere to the rules above
  - `FirstProgram` `MyClass` `String`



Comments

- A ***line comment*** begins with the symbols `//`, and causes the compiler to ignore the remainder of the line
  - This type of comment is used for the code writer or for a programmer who modifies the code
- A ***block comment*** begins with the symbol pair `/*`, and ends with the symbol pair `*/`
  - The compiler ignores anything in between
  - This type of comment can span several lines
  - This type of comment **provides documentation** for the users of the program

Comments

```
// Lab1b.java
// This is a practice lab to output a few verses of "99 bottles of beer on the wall"
// Authors: Carol Zander, Clark Olson, you

public class Lab1b {

   public static void main (String[] args) {
      int numBottles;                  // number of bottles currently on the wall

      // display first verse
      numBottles = 5;
```

Program Documentation

- Java comes with a program called `javadoc` that will automatically extract documentation from block comments in the classes you define
  - As long as their opening has an extra asterisk (`/**`)
- Ultimately, a well written program is self-documenting
  - Its structure is made clear by the choice of identifier names and the indenting pattern
  - When one structure is nested inside another, the inside structure is indented one more level

Comments and a Named Constant

Display 1.8 Comments and a Named Constant

```
1  /**
2   Program to show interest on a sample account balance.
3   Author: Jane Q. Programmer.
4   E-mail Address: janeq@somemachine.etc.etc.
5   Last Changed: September 21, 2004.
6  */
7  public class ShowInterest
8  {
9      public static final double INTEREST_RATE = 2.5;

10     public static void main(String[] args)
11     {
12         double balance = 100;
13         double interest; //as a percent

14         interest = balance * (INTEREST_RATE/100.0);
15         System.out.println("On a balance of $" + balance);
16         System.out.println("you will earn interest of $"
17                                                 + interest);
18         System.out.println("All in just one short year.");
19     }
20
21 }
```

Although it would not be as clear, it is legal to place the definition of INTEREST_RATE here instead.

SAMPLE DIALOGUE

```
On a balance of $100.0
you will earn interest of $2.5
All in just one short year.
```