COMP 110/L Lecture 17

Maryam Jalali

Some slides adapted from Dr. Kyle Dewey

Outline

- `Strings`
- `String.length`
- `String.split`
- Multidimensional arrays

Strings

- A string is an ordered sequence of characters.
- However, because of the String class, we never directly interact with this representation.
- Java provides many methods as part of the String class that can be used to process and manipulate strings.
- These methods do not change the strings since strings in Java are immutable.

Basics

- Declare and assign using regular assignment operator

```
String firstName = “Denise";

String lastName = "Ritchie";

 //we can also reassign values

firstName = "Tom";
```

- Note that the reassignment in the last line in the example does not change the original string.
- It just makes the variable firstName point to a new string

`String` methods

Method name	Description
`indexOf(`**`str`**`)`	index where the start of the given string appears in this string (-1 if it is not there)
`length()`	number of characters in this string

`substring(`**`index1`**`,`**`index2`**`)`
or
`substring(`**`index1`**`)`
	the characters in this string from *index1* (inclusive) to *index2* (exclusive);
	if *index2* omitted, grabs till end of string

`toLowerCase()`	a new string with all lowercase letters
`toUpperCase()`	a new string with all uppercase letters

- These methods are called using the dot notation:

```
String gangsta = "Dr. Dre";
System.out.println(gangsta.length());    // 7
```

Modifying strings

- Methods like `substring`, `toLowerCase`, etc. create/return a new string, rather than modifying the current string.

```
String s = "lil bow wow";
s.toUpperCase();
System.out.println(s);   // lil bow wow
```

- To modify a variable, you must reassign it:

```
String s = "lil bow wow";
s = s.toUpperCase();
System.out.println(s);   // LIL BOW WOW
```

Comparing strings

- Relational operators such as < and == fail on objects.

```
Scanner console = new Scanner(System.in);
System.out.print("What is your name? ");
String name = console.next();
if (name == "Barney") {
    System.out.println("I love you, you love me,");
    System.out.println("We're a happy family!");
}
```

- This code will compile, but it will not print the song.

- == compares objects by *references* (seen later), so it often gives `false` even when two `String`s have the same letters.

The `equals` method

- Objects are compared using a method named `equals`.

```
Scanner console = new Scanner(System.in);
System.out.print("What is your name? ");
String name = console.next();
if (name.equals("Barney")) {
    System.out.println("I love you, you love me,");
    System.out.println("We're a happy family!");
}
```

- Technically this is a method that returns a value of type `boolean`, the type used in logical tests.

`String` test methods

Method	Description
`equals(`**str**`)`	whether two strings contain the same characters
`equalsIgnoreCase(`**str**`)`	whether two strings contain the same characters, ignoring upper vs. lower case
`startsWith(`**str**`)`	whether one contains other's characters at start
`endsWith(`**str**`)`	whether one contains other's characters at end
`contains(`**str**`)`	whether the given string is found within this one

```
String name = console.next();
if (name.startsWith("Dr.")) {
    System.out.println("Are you single?");
} else if (name.equalsIgnoreCase("LUMBERG")) {
    System.out.println("I need your TPS reports.");
}
```

Type `char`

- `char` : A primitive type representing single characters.
  - Each character inside a `String` is stored as a `char` value.
  - Literal `char` values are surrounded with apostrophe (single-quote) marks, such as `'a'` or `'4'` or `'\n'` or `'\''`

  - It is legal to have variables, parameters, returns of type `char`

```
char letter = 'S';
System.out.println(letter);                    // S
```

- `char` values can be concatenated with strings.

```
char initial = 'P';
System.out.println(initial + " Diddy");  // P Diddy
```

The `charAt` method

- The `char`s in a `String` can be accessed using the `charAt` method.

```
String food = "cookie";
char firstLetter = food.charAt(0);   // 'c'

System.out.println(firstLetter + " is for " + food);
System.out.println("That's good enough for me!");
```

- You can use a `for` loop to print or examine each character.

```
String major = "CSE";
for (int i = 0; i < major.length(); i++) {
    char c = major.charAt(i);
    System.out.println(c);
}
```

Output:

```
C
S
E
```

`char` **vs.** `String`

- `"h"` is a `String`
  `'h'` is a `char` (the two behave differently)

- `String` is an object; it contains methods

```
String s = "h";
s = s.toUpperCase();          // 'H'
int len = s.length();         //  1
char first = s.charAt(0);     // 'H'
```

- `char` is primitive; you can't call methods on it

```
char c = 'h';
c = c.toUpperCase();    // ERROR: "cannot be dereferenced"
```

Comparing `char` values

- You can compare `char` values with relational operators:

```
'a' < 'b'   and   'X' == 'X'   and   'Q' != 'q'
```

  - An example that prints the alphabet:

```
for (char c = 'a'; c <= 'z'; c++) {
    System.out.print(c);
}
```

- You can test the value of a string's character:

```
String word = console.next();
if (word.charAt(word.length() - 1) == 's') {
    System.out.println(word + " is plural.");
}
```

Formatting Output

Use the `printf` statement.

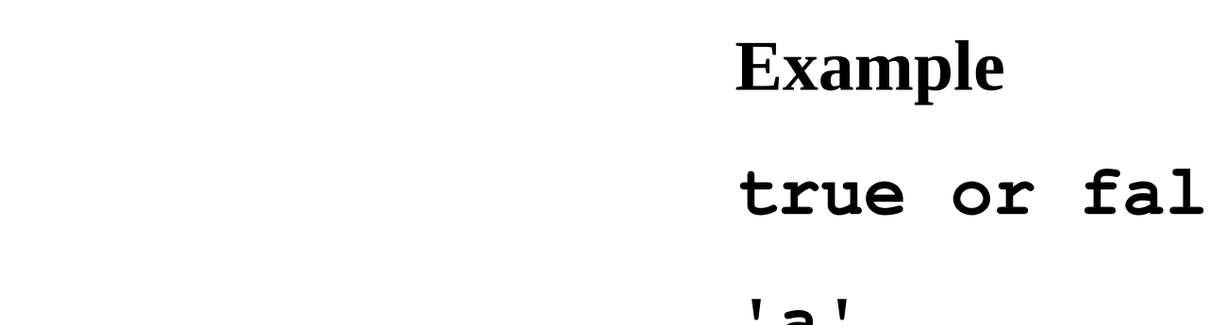
```
System.out.printf(format, items);
```

Where format is a string that may consist of substrings and format specifiers. A format specifier specifies how an item should be displayed. An item may be a numeric value, character, boolean value, or a string. Each specifier begins with a percent sign.

Frequently-Used Specifiers

Specifier	Output	Example
%b	a boolean value	true or false
%c	a character	'a'
%d	a decimal integer	200
%f	a floating-point number	45.460000
%e	a number in standard scientific notation	4.556000e+01
%s	a string	"Java is cool"

```
int count = 5;
double amount = 45.56;
System.out.printf("count is %d and amount is %f", count, amount);
```



display count is 5 and amount is 45.560000

`String.length`

Returns the number of `chars` in the given `String`

`String.length`

Returns the number of `chars` in the given `String`

`"abc".length()`

`String.length`

Returns the number of `chars` in the given `String`

`"abc".length()`

`3`

`String.length`

Returns the number of `chars` in the given `String`

```
"abc".length()
```

```
3
```

```
"".length()
```

`String.length`

Returns the number of `chars` in the given `String`

`"abc".length()`

3

`"".length()`

0

Example:

StringLength.java

String.split

Tokenizing

- It is common to store different pieces of data as a string such that each individual piece of data is demarcated by some delimiter.

```
Smith,Joe,12345678,1985-09-08
```

- Often we need to process such strings to extract each individual piece of data.
- Processing such strings is usually referred to as *parsing*.
- In particular, a string is "split" into a collection of individual strings called tokens (thus the process is also sometimes referred to as tokenizing).

`String.split`

Allows for a `String` to be separated into different parts.

Returns an array of `String`s (`String[]`).

`String.split`

Allows for a `String` to be separated into different parts.

Returns an array of `String`s (`String[]`).

```
"foo,bar".split(",")
```

`String.split`

Allows for a `String` to be separated into different parts.

Returns an array of `String`s (`String[]`).

```
"foo,bar".split(",")
```

```
new String[]{"foo", "bar"}
```

Example:

SplitOnComma.java

What `split` Takes

`split` takes a *regular expression*.

Regular expressions describe different string patterns.

What `split` Takes

`split` takes a *regular expression*.

Regular expressions describe different string patterns.

```
"foo,bar".split(",")
```

What `split` Takes

`split` takes a *regular expression*.

Regular expressions describe different string patterns.

`"foo,bar".split(",")`

`","`:matches only one pattern:a comma

What `split` Takes

`split` takes a *regular expression*.

Regular expressions describe different string patterns.

```
"foo,bar".split(",")
```

",": matches only one pattern: a comma

```
"foo.bar".split(".")
```

What `split` Takes

`split` takes a *regular expression*.

Regular expressions describe different string patterns.

`"foo,bar".split(",")`

`","`: matches only one pattern: a comma

`"foo.bar".split(".")`

`"."`: matches **any** single character

What `split` Takes

`split` takes a *regular expression*.

Regular expressions describe different string patterns.

`"foo,bar".split(",")`

`","`: matches only one pattern: a comma

`"foo.bar".split(".")`

`"."`: matches **any** single character

`"foo.bar".split("\\.")`

What `split` Takes

`split` takes a *regular expression*.

Regular expressions describe different string patterns.

`"foo,bar".split(",")`

`","`: matches only one pattern: a comma

`"foo.bar".split(".")`

`"."`: matches **any** single character

`"foo.bar".split("\\.")`

`"\\."`: matches a period (backslash followed by a period)

Example:

SplitOnAnything.java

Multidimensional Arrays

Recap - Arrays

Arrays are fixed-length sequences
of elements of the same type.

Recap - Arrays

Arrays are fixed-length sequences of elements of the same type.

```
new char[]{'a', 'b', 'c'}

new int[]{1, 2, 3}

new String[]{"foo", "bar"}

new double[]{1.2, 3.4}
```

Motivations

Thus far, you have used one-dimensional arrays to model linear collections of elements. You can use a two-dimensional array to represent a matrix or a table. For example, the following table that describes the distances between the cities can be represented using a two-dimensional array.

Distance Table (in miles)

	Chicago	Boston	New York	Atlanta	Miami	Dallas	Houston
Chicago	0	983	787	714	1375	967	1087
Boston	983	0	214	1102	1763	1723	1842
New York	787	214	0	888	1549	1548	1627
Atlanta	714	1102	888	0	661	781	810
Miami	1375	1763	1549	661	0	1426	1187
Dallas	967	1723	1548	781	1426	0	239
Houston	1087	1842	1627	810	1187	239	0

Motivations

```
double[][] distances = {
    {0, 983, 787, 714, 1375, 967, 1087},
    {983, 0, 214, 1102, 1763, 1723, 1842},
    {787, 214, 0, 888, 1549, 1548, 1627},
    {714, 1102, 888, 0, 661, 781, 810},
    {1375, 1763, 1549, 661, 0, 1426, 1187},
    {967, 1723, 1548, 781, 1426, 0, 239},
    {1087, 1842, 1627, 810, 1187, 239, 0},
};
```

Multidimensional Arrays

Java also allows us to make arrays of *arrays*.

These are often called *multidimensional* arrays.

Multidimensional Arrays

Java also allows us to make arrays of *arrays*.

These are often called *multidimensional* arrays.

```
new int[][]{ new int[]{1, 2, 3},
             new int[]{4, 5},
             new int[]{6},
             new int[0],
             new int[]{7, 8, 9}
             }
```

Multidimensional Arrays

Java also allows us to make arrays of *arrays*.
These are often called *multidimensional* arrays.

```
new int[][]{ new int[]{1, 2, 3},
             new int[]{4, 5},
             new int[]{6},
             new int[0],
             new int[]{7, 8, 9}
             }
```

Corresponding type: `int[][]`

Multidimensional Array Utility

Commonly used for representing tables

Multidimensional Array Utility

Commonly used for representing tables

13	12	19
64	89	247
78	57	21

Multidimensional Array Utility

Commonly used for representing tables

13	12	19
64	89	247
78	57	21

```
new int[][]{ new int[]{13, 12, 19},
             new int[]{64, 89, 247},
             new int[]{78, 57, 21} }
```

Accessing Rows

One row of a two-dimensional array is an array...

Accessing Rows

One row of a two-dimensional array is an array...

```
int[][] array = ...;
int[] row = array[0];
```

Accessing Rows

One row of a two-dimensional array is an array...

```
int[][] array = ...;
int[] row = array[0];
```

Accessing Columns

...and columns are individual elements of rows.

Accessing Rows

One row of a two-dimensional array is an array...

```
int[][] array = ...;
int[] row = array[0];
```

Accessing Columns

...and columns are individual elements of rows.

```
int[][] array = ...;
int[] row = array[0];
int columnElement = row[5];
```

Accessing Rows

One row of a two-dimensional array is an array...

```
int[][] array = ...;
int[] row = array[0];
```

Accessing Columns

...and columns are individual elements of rows.

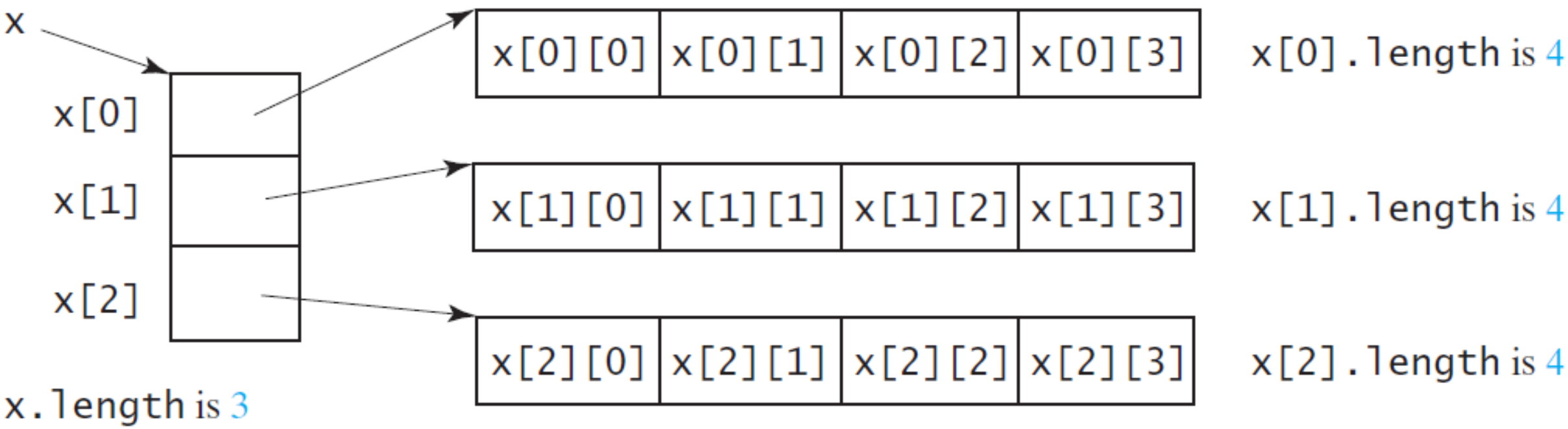
```
int[][] array = ...;
int[] row = array[0];
int columnElement = row[5];
```

```
int[][] array = ...;
int columnElement = array[0][5];
```

	Column 0	Column 1	Column 2
Row 0	x[0][0]	x[0][1]	x[0][2]
Row 1	x[1][0]	x[1][1]	x[1][2]
Row 2	x[2][0]	x[2][1]	x[2][2]

Lengths of Two-dimensional Arrays

```
int[][] x = new int[3][4];
```



Lengths of Two-dimensional Arrays, cont.

```
int[][] array = {
                    {1, 2, 3},
                    {4, 5, 6},
                    {7, 8, 9},
                    {10, 11, 12}
                };
```

array.length

array[0].length

array[1].length

array[2].length

array[3].length

array[4].length ArrayIndexOutOfBoundsException

Ragged Arrays

Each row in a two-dimensional array is itself an array. So, the rows can have different lengths. Such an array is known as *a ragged array*.

For example,

```
int[][] matrix = {
  {1, 2, 3, 4, 5},
  {2, 3, 4, 5},
  {3, 4, 5},
  {4, 5},
  {5}
};
```

matrix.length is 5
matrix[0].length is 5
matrix[1].length is 4
matrix[2].length is 3
matrix[3].length is 2
matrix[4].length is 1

Example:

`AccessTwoDimensionalElement.java`

More 2D Array Examples

- `PrintRow2D.java`
- `PrintCol2D.java`