EECS 1720
Building Interactive Systems

Lecture 4 :: Java Classes & Objects (3)

YORK UNIVERSITÉ UNIVERSITY

Announcements → Lab 1 notes

- Walkthrough Videos **<u>are given in the lab session</u>**
  - posted (with delay) in labs & lecture notes section on eclass
  - So far there is one for lab0 (last week), and lab1 (this week). Note lab0 not worth marks but still important (as it shows you how to import *.zip projects into eclipse properly

- ArrayList should be in java.util (not java.lang) package

- Error with tester for test19_getCourseName():
  - replace test19 (in TestAllUtils.java) with the following:

```
@Test
public void test19_getCourseName() {
  assertEquals("getCourseName() failed", "Building Interactive Systems", StrUtils.getCourseName());
}
```

- StringBuilder is listed as a possible class for use (resources section of lab1.pdf)
  - StringBuilder acts like an ArrayList… starts as an empty string when instantiated
  - can use append() method to join additional strings to the end
  - Better than using str1 = str1 + “additional bit” ;
    - when there are many strings to join together (java slows down as each + makes a new string, which takes up memory.
    - it does not modify existing strings because strings are “immutable” (cannot change them, only create new ones)
    - Relevant for alternatingCaps() method (if you join strings this way)

Recall:

- Connecting to a String

```
Scanner inStr = new Scanner("some string here");
```

- Connecting to Keyboard Input (System.in)

```
Scanner in =   new Scanner(System.in);
```

- Parsing inputs typed in by user at run time:

```
System.out.println("Enter a value between 50 and 100: ");
int val1 = in.nextInt();
int val2 = in.nextInt();
```

OR

```
System.out.println("Enter a value between 50 and 100: ");
int val1 = Integer.parseInt(in.next());
int val2 = Integer.parseInt(in.next());
```

YORK UNIVERSITÉ UNIVERSITY

Recall: Invalid strings?

- E.g.

```
String str = “1es.14”;            // invalid integer string
Scanner in = new Scanner(str);
int value = in.nextInt();

// same thing for
int value2 = Integer.parseInt(in.next());
```

```
// lets look at API:
```

parseInt

```
public static int parseInt(String s)
                    throws NumberFormatException
```

Parses the string argument as a signed decimal integer. The characters in the string must all be decimal digits, except that the first character may be an ASCII minus sign '-' ('\u002D') to indicate a negative value or an ASCII plus sign '+' ('\u002B') to indicate a positive value. The resulting integer value is returned, exactly as if the argument and the radix 10 were given as arguments to the parseInt(java.lang.String, int) method.

Parameters:

s - a String containing the int representation to be parsed

Returns:

the integer value represented by the argument in decimal.

Throws:

NumberFormatException - if the string does not contain a parsable integer.

Example (checking before reading)

```
import java.util.Scanner;


public class GuessingGame {

    public static void main(String[] args) {

        Scanner in = new Scanner(System.in);

        System.out.println(“.. can you guess the number? ”);
        double guess;

        if (in.hasNextDouble()) {
          guess = in.nextDouble();
          System.out.println(“you entered: “ + guess);
        }

        System.out.println(“program is ending now”);
    }
}
```

Other useful Scanner methods (… later)

- Can ”look” ahead (before scanning)

	nextByte() method.
boolean	**hasNextDouble**() Returns true if the next token in this scanner's input can be interpreted as a double value using the **nextDouble()** method.
boolean	**hasNextFloat**() Returns true if the next token in this scanner's input can be interpreted as a float value using the **nextFloat()** method.
boolean	**hasNextInt**() Returns true if the next token in this scanner's input can be interpreted as an int value in the default radix using the **nextInt()** method.
boolean	**hasNextInt**(int radix) Returns true if the next token in this scanner's input can be interpreted as an int value in the specified radix using the **nextInt()** method.
boolean	**hasNextLine**() Returns true if there is another line in the input of this scanner.
boolean	**hasNextLong**() Returns true if the next token in this scanner's input can be interpreted as a long value in the default radix using the **nextLong()** method.

More Detailed Example: parsing input using ArrayLists…

- Let's say we want to process a string that has a mixture of doubles and ints, and we want to count and extract them (while ignoring all the other stuff)

ECLIPSE DEMO

Recall: Arrays are a bit confusing ArrayLists are much easier to work with

- Arrays have one field (length), but no methods

```
String[] myStringArray = new String[10]; // fixed size
myStringArray[0] = “eecs1710”;
myStringArray[1] = “eecs1720”;
out.println(“length = ” + myStringArray.length);
```

- ArrayLists are a reference type by design, thus support fields and methods

```
ArrayList<String> myArrayList = new ArrayList<String>(); // can grow

// no length field, only size() method
out.println(myArrayList.size());              // empty, so size = 0

myArrayList.add(“eecs1710”);
myArrayList.add(“eecs1720”);

out.println(myArrayList.size());
```

Note on ArrayLists

- Must always use a Reference type as the element <>
  - Cannot use primitive types

- ArrayList<Integer>
- ArrayList<Double>
- etc

- Why? ArrayList makes use of certain methods in these classes to function correctly
  - E.g. to compare elements (e.g. equals() method is needed), also when searching for an element in the collection..

```java
public static void scanKeyboardInput() {

        Scanner in = new Scanner(System.in);                                // keyboard
        System.out.print("Please enter a set of space separated values: ");  // prompt

        // create some counters
        int countIntegers = 0;
        int countReals = 0;

        // create some storage for numbers found (containers must use wrapper classes)
        ArrayList<Integer> intList = new ArrayList<Integer>();
        ArrayList<Double> realList = new ArrayList<Double>();

        // grab entire line of input from keyboard, make a string based scanner
        Scanner line = new Scanner(in.nextLine());

        while (line.hasNext()) {

                // look ahead
                if (line.hasNextInt()) {
                        countIntegers++;
                        intList.add(line.nextInt());
                }
                else if (line.hasNextDouble()) {
                        countReals++;
                        realList.add(line.nextDouble());
                }
                else {
                        line.next();
                }
        }
        line.close();         // good practice to close scanner objects when done
        in.close();

        System.out.println("found " + countIntegers + " integers, and " + countReals + " reals");
        System.out.println("\nints:  \n" + intList.toString());
        System.out.println("\nreals: \n" + realList.toString());

}
```

```
Please enter a set of space separated values: 234.5235 4.23 5252 344 6 2 r4g5 glsdkjh &%&# 32 4.522 0.342 -56
found 6 integers, and 4 reals

ints:
[5252, 344, 6, 2, 32, -56]

reals:
[234.5235, 4.23, 4.522, 0.342]
```

More (advanced) String Examples (StringBuilder, Regex)

String → immutable type (cannot modify)

StringBuilder → a modifiable string type

Regex → Regular Expressions

YORK UNIVERSITÉ UNIVERSITY

String

- String type is an “immutable” type
  - This means, once created, a String cannot be modified
  - Some methods ”appear” to modify, but in actuality, they return entirely new String objects

- E.g.
  - replace()
  - substring()

`String`	`replace(char oldChar, char newChar)` Returns a string resulting from replacing all occurrences of `oldChar` in this string with `newChar`.
`String`	`replace(CharSequence target, CharSequence replacement)` Replaces each substring of this string that matches the literal target sequence with the specified literal replacement sequence.
`String`	`replaceAll(String regex, String replacement)` Replaces each substring of this string that matches the given **regular expression** with the given replacement.
`String`	`replaceFirst(String regex, String replacement)` Replaces the first substring of this string that matches the given **regular expression** with the given replacement.

`String`	`substring(int beginIndex)` Returns a string that is a substring of this string.
`String`	`substring(int beginIndex, int endIndex)` Returns a string that is a substring of this string.

substring() method

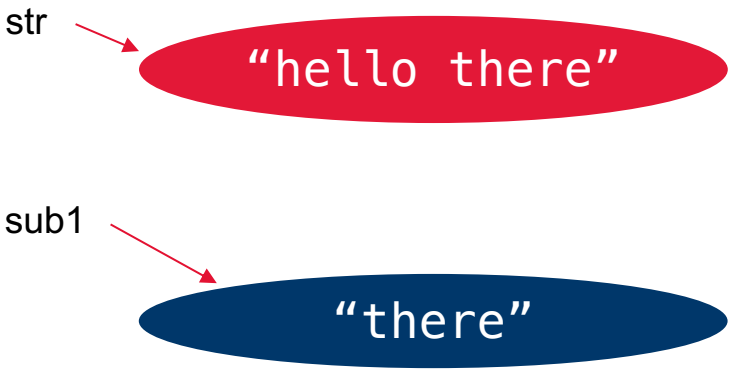
- extracts a new String (as a substring of another)

```
String str, sub1, sub2, sub3;
str = “hello there”;

sub1 = str.substring(1, 6);

str = str.substring(1, 6);

sub2 = str.substring(3);
```



palindrome example

- Palindrome → a string that when reversed is the same
  - E.g. “civic”, “madam”, “racecar”

- Say we want to write a method to check if a string is a palindrome (returns true if it is, false if not)?

```
public static boolean isPalindrome(String str) {

          // how to do it?

}
```

YORK UNIVERSITÉ UNIVERSITY

StringBuilder

- Because Strings are immutable str1 = str1 + str2 (creates a new string) – if done a lot, this will fill up memory!

- Facilitates (fast/efficient) mutation of a string

Constructors

Constructor and Description
`StringBuilder()` Constructs a string builder with no characters in it and an initial capacity of 16 characters.
`StringBuilder(CharSequence seq)` Constructs a string builder that contains the same characters as the specified `CharSequence`.
`StringBuilder(int capacity)` Constructs a string builder with no characters in it and an initial capacity specified by the `capacity` argument.
`StringBuilder(String str)` Constructs a string builder initialized to the contents of the specified string.

- Has 3 methods to modify characters:

```
append(anything)
insert(int, anything)
delete(int, int)
```

Faster because these operations do not create an entire new string with each change

Demo (timing comparison)

```
public class DemoStringBuilderTiming {

      public static void main(String[] args) {

            final String WORD = "Test";
            final int REPEATS = 10000;

            // slow way
            long start = System.currentTimeMillis();

            String s = "";

            for (int i=0; i<REPEATS; i++) {
                  s += WORD;
            }
            long elapsed = System.currentTimeMillis() - start ;
            System.out.println("Time using String class: " + elapsed + " ms");


            // fast way
            start = System.currentTimeMillis();

            StringBuilder s2 = new StringBuilder();
            for (int i=0; i<REPEATS; i++) {
                  s2.append(WORD);
            }
            elapsed = System.currentTimeMillis() - start;
            System.out.println("Time using StringBuilder class: " + elapsed + " ms");

      }
}
```

Note: System has a method to check the system clock

Version 1: using String and +=

Version 2: using StringBuilder

How to extract the built string?

```
// … not shown

StringBuilder s2 = new StringBuilder();
for (int i=0; i<REPEATS; i++) {
     s2.append(WORD);
}

String result = s2.toString();
```

e.g. partial UML diagram

```
java.lang :: StringBuilder

// no public fields

// constructors (not all shown)
+ StringBuilder()
+ StringBuilder(String str)

// methods (not all shown)
+ append(String) : StringBuilder
+ append(char) : StringBuilder
// ...
+ insert(int, String) : StringBuilder
+ insert(int, char) : StringBuilder
// ...
+ delete(int,int) : StringBuilder

+ toString() : String
```

Pattern Matching & Regular Expressions

```
String str = “some string sentence here”;
```

- String searches (exact search for a target)

```
boolean targetExists = str.contains(target);
int targetStartIndex = str.indexOf(target);
```

- Less exact searching sometimes useful
  - E.g. count number of vowels

- Use “*pattern matching*”
  - Pattern represented as a “*regular expression*” (aka “*regex*”)
  - E.g. pattern e.g. → java.util.*
    - * acts as a wildcard (i.e. it represents “all classes” in java.util)

Regular Expressions

CHARACTER SPECIFICATIONS	
[a-m]	Characters between a and m, inclusive
[a-mp-t]	Characters a through m, inclusive, or p through t, inclusive
[abc]	The character a, b, or c
[^abc]	Any character except a, b, or c
[a-m&&[^ck]]	The characters a through m but neither c nor k
PREDEFINED SPECIFICATIONS	
.	Any character
\d	A digit, [0-9]
\s	A whitespace character, [\t\bx0B\f\r]
\w	A word character, [a-zA-Z_0-9]
\p{Punct}	A punctuation symbol, [!"#$%&'()*+,-./:;<=>?@[\]^_`{|}~]
QUANTIFIERS	
X?	X, once or not at all
X*	X, zero or more times
X+	X, one or more times
X{n,m}	X, at least n but no more than m times
LOGICIAL OPERATORS	
XY	X followed by Y
X|Y	X or Y
(X)	X as a capturing group

String methods that can use regex:

- split()
- matches()
- replaceAll()
- etc…

YORK U
UNIVERSITÉ
UNIVERSITY

The `matches` method

- from the String class API:

matches

```
public boolean matches(String regex)
```

Tells whether or not this string matches the given regular expression.

Parameters:

regex - the regular expression to which this string is to be matched

Returns:

true if, and only if, this string matches the given regular expression

e.g. partial UML diagram

```
java.lang :: String

// no public fields

// constructors (not all shown)
+ String()
+ String(String str)

// methods (not all shown)
+ length() : int
+ contains(String) : boolean
+ indexOf(String) : int
+ lastIndexOf(String) : int

+ matches(String) : boolean
```

Exact match

```
import java.io.*;

public class DemoMatches {

     public static void main(String[] str) throws IOException {

          PrintStream output = System.out;

          String s1 = "a";
          String s2 = "b";
          String regex = "a";
          output.println(s1.matches(regex) + " : " + s2.matches(regex));

          s1 = "a4";
          s2 = "4a";
          regex = "[a-z][0-9]";
          output.println(s1.matches(regex) + " : " + s2.matches(regex));

          s1 = "abc7";
          s2 = "abcd789";
          regex = "[a-z]+[0-9]";
          output.println(s1.matches(regex) + " : " + s2.matches(regex));

          s1 = "abc123";
          s2 = "abcd1234";
          regex = "[^0-9]+[0-9]{1,3}";
          output.println(s1.matches(regex) + " : " + s2.matches(regex));
     }
}
```

Range match

```
import java.io.*;

public class DemoMatches {

     public static void main(String[] str) throws IOException {

          PrintStream output = System.out;

          String s1 = "a";
          String s2 = "b";
          String regex = "a";
          output.println(s1.matches(regex) + " : " + s2.matches(regex));

          s1 = "a4";
          s2 = "4a";
          regex = "[a-z][0-9]";
          output.println(s1.matches(regex) + " : " + s2.matches(regex));

          s1 = "abc7";
          s2 = "abcd789";
          regex = "[a-z]+[0-9]";
          output.println(s1.matches(regex) + " : " + s2.matches(regex));

          s1 = "abc123";
          s2 = "abcd1234";
          regex = "[^0-9]+[0-9]{1,3}";
          output.println(s1.matches(regex) + " : " + s2.matches(regex));
     }
}
```

Any lowercase alphabetic character followed by a number

Range match allowing repeats

```
import java.io.*;

public class DemoMatches {

     public static void main(String[] str) throws IOException {

          PrintStream output = System.out;

          String s1 = "a";
          String s2 = "b";
          String regex = "a";
          output.println(s1.matches(regex) + " : " + s2.matches(regex));

          s1 = "a4";
          s2 = "4a";
          regex = "[a-z][0-9]";
          output.println(s1.matches(regex) + " : " + s2.matches(regex));

          s1 = "abc7";
          s2 = "abcd789";
          regex = "[a-z]+[0-9]";
          output.println(s1.matches(regex) + " : " + s2.matches(regex));

          s1 = "abc123";
          s2 = "abcd1234";
          regex = "[^0-9]+[0-9]{1,3}";
          output.println(s1.matches(regex) + " : " + s2.matches(regex));
     }
}
```

One or more lowercase alphabetic characters followed by a number

Range match allowing repeats & exclusions

```
import java.io.*;

public class DemoMatches {

     public static void main(String[] str) throws IOException {

          PrintStream output = System.out;

          String s1 = "a";
          String s2 = "b";
          String regex = "a";
          output.println(s1.matches(regex) + " : " + s2.matches(regex));

          s1 = "a4";
          s2 = "4a";
          regex = "[a-z][0-9]";
          output.println(s1.matches(regex) + " : " + s2.matches(regex));

          s1 = "abc7";
          s2 = "abcd789";
          regex = "[a-z]+[0-9]";
          output.println(s1.matches(regex) + " : "

          s1 = "abc123";
          s2 = "abcd1234";
          regex = "[^0-9]+[0-9]{1,3}";
          output.println(s1.matches(regex) + " : " + s2.matches(regex));
     }
}
```

can have any repeated set of non-numeric characters, followed by 1 to 3 repeated numeric characters

Example

- Count occurrences of vowels?

```
import java.util.Scanner;

public class DemoCountVowels {

      public static void main(String[] args) {

            Scanner input = new Scanner(System.in);

            System.out.println("Enter a string ...");
            String s = input.nextLine().toLowerCase();

            int count = 0;

            for (int i = 0; i < s.length(); ++i) {

                  if ( s.substring(i, i + 1).matches("[aeiou]") ) {

                        count++;
                  }
            }

            System.out.printf("Number of vowels = %d\n", count);
            input.close();

      }

}
```

Example

- Count punctuation characters in a string?

```
import java.util.Scanner;

public class DemoCountPunct {

      public static void main(String[] args) {

            Scanner input = new Scanner(System.in);

            System.out.println("Enter a string ...");
            String s = input.nextLine();

            int count = 0;

            for (int i = 0; i < s.length(); ++i) {

                  if ( s.substring(i, i + 1).matches("\\p{Punct}") ) {

                        count++;
                  }
            }

            System.out.printf("Number of punctuation characters = %d\n", count);
            input.close();

      }

}
```

Example

- Checking if a String is a valid postcode:

```
//postcode example (1710 final F2020)

s1 = "M3K 2G6";
s2 = "2G6M3K";
regex = "[A-Z][0-9][A-Z]\\s*[0-9][A-Z][0-9]";

output.println(s1.matches(regex) + " : " +
s2.matches(regex));
```

Appendix :: String.format() & System.out.printf()

A way of formatting your strings/prints

YORK UNIVERSITÉ UNIVERSITY

formatting a string → without nf() or nfs() ??

```
System.out.print(String)
```

```
System.out.println(String)
```

String type is an "argument" (input) for the methods print() and println()

- Another version of the print method:

```
System.out.printf(String, ..list of variables)
```

String type holds a special formatted String which embeds where to place and how to format the list of variables

YORK UNIVERSITÉ UNIVERSITY

String syntax for using `printf(String s, …)`

```
int x = 5;
float y = 6.234f;
System.out.printf("x = %d, y = %f \n", x, y);
```

Formatted string
% denotes where to embed variables from the remaining list of arguments (i.e. x, y)

Outputs:

```
x = 5, y = 6.234
```

YORK UNIVERSITÉ UNIVERSITY

String syntax for using `printf(String s, …)`

```
%[argument_index$][flags][width][.precision]conversion
```

[ ] → means optional

So minimum usage would be: %*conversion*

String syntax for using `printf(String s, …)`

```
%[argument_index$][flags][width][.precision]conversion
```

[ ] → means optional

So minimum usage would be: %*conversion*

conversion: a character that indicates how the argument should be formatted:

e.g.
‘c’ - character
‘s’ - string
‘d’ - integer (formatted as a decimal integer)
‘f’ - floating point (formatted as a decimal)
‘n’ - line separator (new line)

YORK UNIVERSITÉ UNIVERSITY

String syntax for using `printf(String s, …)`

```
%[argument_index$][flags][width][.precision]conversion
```

precision: a number that indicates how many decimal places to print:

```
double var1 = 14.52;
System.out.printf( “The value of var1 is %.6f  !! ”, var1);
```

```
The value of var1 is 14.520000  !!
```

YORK UNIVERSITÉ UNIVERSITY

String syntax for using `printf(String s, …)`

```
%[argument_index$][flags][width][.precision]conversion
```

width: a number that indicates how many characters to print for the entire number (i.e. inclusive of “.”)

```
double var1 = 14.52;
System.out.printf( “The value of var1 is %5.3f  !! ”, var1);
```

```
The value of var1 is  14.520  !!
```

width = 5 characters

YORK UNIVERSITÉ UNIVERSITY

String syntax for using `printf(String s, …)`

```
%[argument_index$][flags][width][.precision]conversion
```

flags: character(s) that modify output: e.g. '+' always includes a sign; '-' left justifies, '(' encloses negatives in parentheses, etc.

```
double var1 = 14.52;
System.out.printf( "The value of var1 is %+5.3f !! ", var1);
```

```
The value of var1 is  +14.520  !!
```

YORK UNIVERSITÉ UNIVERSITY

String syntax for using `printf(String s, …)`

```
%[argument_index$][flags][width][.precision]conversion
```

argument_index: specifies the index of the variable in the comma separated list of arguments.

```
double var1 = 14.52;
String str = “dollars”;
System.out.printf( “The value of var1 is %2$s %1$+5.3f !! ”, var1, str);
```

```
The value of var1 is dollars   +14.520 !!
```

YORK UNIVERSITÉ UNIVERSITY