Lecture 12:
Strings

Sierra College
CSCI-12
Spring 2015
Mon 03/09/15

Announcements

- **General**
  - Still catching up on grading, but making progress…
- **Schedule**
  - Some adjustments made, please see updated schedule
  - I have decided to "push out" the due dates for remaining assignments before the midterm (1 wk for each one)
    - Schedule topics unchanged/"wiggle room", but longer for the indicated assignments
  - Midterm exam in ~2 wks (Weds 3/25), before spring break
- **Past due assignments**
  - HW09: External Input, accepted thru Tues 3/10
- **Current assignments**
  - HW10: Methods, due Monday 3/9
  - HW11: SimpleDate, due Thursday 3/12 (lab time Weds) *[due date pushed out]*
- **New assignments**
  - HW12: Strings, due Tuesday 3/17 *[due date pushed out]*
    - Some manipulations of a fixed String using its API methods

Lecture Topics

- **Last time**:
  - Discussion on methods and calling them (HW10: Methods)
  - Creating and using classes/objects
  - The *SimpleDate* class
- **Today**:
  - Finish up *SimpleDate*
  - The *String* class and its API

Motivations For The ***String*** Class

- Java provides ***char*** as one of the 8 primitive datatypes
  - A *char* is one single Unicode character (2 bytes)
  - Appears in single quotes (or its numerical equivalent):

```
char userInput  =  'Y';
final char EURO_SYMBOL  =  0x20AC;
```

- However, almost all applications have some need for:
  - A datatype handling sequences of characters (i.e., text)
  - All routine manipulations of such sequences
- The following would be completely unacceptable:

```
System.out.println('H' + 'e' + 'l' + 'l' + 'o' + ' ' +
                                    'W' + 'o' + 'r' + 'l' + 'd' + '!');
```

- These are the motivations for some sort of text, or "string", datatype

The ***String*** Class

- A **string** is just any sequence of text within double quotes
  - A *String* is within double quotes: “x”, “I am a String”
  - A *char* is within single quotes: ‘x’
- Java provides ***String*** as a predefined Java API class
  - All constructors and methods are given in its API
  - **It is automatically available to any Java program**
    - Part of the java.lang package, so no need to import
  - But: String is a class, NOT a core datatype!
- From an application perspective, *String* is just another native datatype for text variables and literals

```
String firstName = new String(“Joe”);
System.out.println( “Hello, “ + firstName + “!” );
```

The ***String*** API

- As with any Java class, a full description of the class appears in the Java API
  - How to create objects of the String class (constructors)
  - How to use objects of the String class (methods)
  - **The Java API is always the authoritative source on the API of any Java-provided class**
- A (very) abridged version of the *String* API is also given in your text:
  - Pg.161, Table 7

String Constructors

String Constructors
`String( String str )` allocates a *String* object with the value of *str*, which is a *String* object or a *String* literal
`String( )` allocates an empty *String* object

Examples:

```
String greeting = new String( "Hello" );
String empty = new String( );
String message = “This is another way of creating a String”;
```

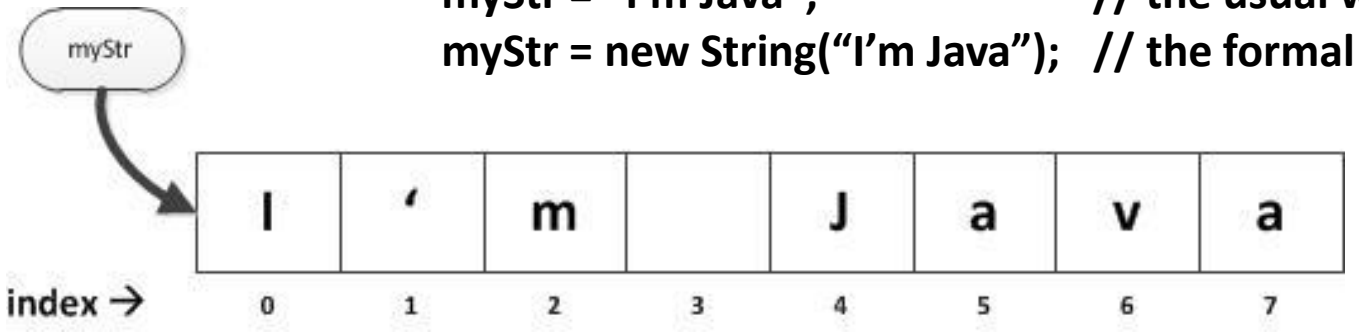
Last example demonstrates special Java support for instantiating new strings (p.156)

- Assign a *String* literal to a *String* object reference: **String str = “String text”;**
- No use of the *new* keyword required
- Frequent, common usage warrants this special support for a “shortcut”

String Visualization

- Internally, a *String* is some organized collection of ***chars***
  - Array? Linked list? (we don't know, don't care)
  - Think of it as a group of indexed *chars* (0-based)
- Declaring a *String* results in a **string reference** which points to the data (just like any other object)

```
String  myStr;
myStr = "I'm Java";                  // the usual way
myStr = new String("I'm Java");   // the formal way
```



Things With *Strings*

- So now that we can create *Strings*… what can we do with them??
- As with any Java class, its API is the official last word (here, the Java API)
- Some of the typical things we want to do with a *String*:
  - How long is it?
  - How can we find specific patterns in it?
  - How can we extract specific characters or substrings?
  - How do we manipulate its case? (upper/lower)
  - How do we build up bigger strings from smaller strings?
- See ***StringUsage.java*** in **Example Source Code**
  - All of the following method examples demonstrated there

The *length* Method

Return type	Method name and argument list
`int`	`length( )` returns the number of characters in the *String*

Examples:

```
String greeting = new String("Hello“);
int len = greeting.length( );     // len is now 5
```

The *charAt* Method

Return type	Method name and argument list
`char`	`charAt( int index )` returns the character at position *index*

- Note: strings in Java (as in C, C++) are **0-based**
  - First character is at position 0 (not 1)
  - Last character is at position **str.length() - 1**

Examples:

H	e	l	l	o
0	1	2	3	4

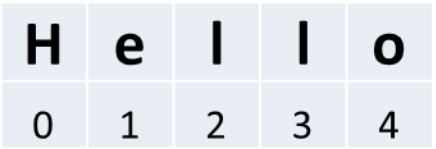
```
String greeting = new String("Hello“);
char firstChar = greeting.charAt( 0 );          // firstChar = ‘H’
char lastChar = greeting.charAt( 4 );           // lastChar = ‘o’ but hardwired
lastChar = greeting.charAt(greeting.length()-1); // lastChar = ‘o’ but more general
```

The last form is preferred (more general), and will work for ANY *String*

The *indexOf* Method

Return type	Method name and argument list
`int`	`indexOf( String searchString )` returns the index of the first occurrence of *searchString*, or -1 if not found
`int`	`indexOf( char searchChar )` returns the index of the first occurrence of *searchChar*, or -1 if not found

Examples:




```
String greeting = new String("Hello“);
int index = greeting.indexOf( 'e' );        // index = 1
index = greeting.indexOf(‘l’);              // index = 2
index = greeting.indexOf(“lo”);             // index = 3
```

The *substring* Method

Return type	Method name and argument list
`String`	`substring( int startIndex, int endIndex )` returns a substring of the *String* object beginning at the character at index ***startIndex,*** and ending at the character at index ***end Index – 1***

Examples:

H	e	l	l	o
0	1	2	3	4

```
String greeting = "Hello";
String substr1 = greeting.substring(0, 2);                  // substr1 = “He”
String substr2 = greeting.substring( 3, greeting.length( ) ); // substr2 = “lo”
```

The *toUpperCase* and *toLowerCase* Methods

Return type	Method name and argument list
`String`	`toUpperCase( )` returns a **copy** of the *String* with all letters uppercase; original string is **unchanged** unless return copy is saved
`String`	`toLowerCase( )` returns a **copy** of the *String* with all letters lowercase; original string is **unchanged** unless return copy is saved

Examples:

```
String greeting = new String("Hello“);
greeting.toUpperCase();                          // nothing happens to new copy
System.out.println(greeting);                    // still “Hello”
System.out.println(greeting.toUpperCase());      // “HELLO”, original unchanged
System.out.println(greeting.toLowerCase());      // “hello”, original unchanged
greeting = greeting.toUpperCase();               // greeting is now “HELLO”
```

String Concatenation

- **String concatenation** refers to the splicing together of smaller substrings into a larger string
  - The substrings may be any combination of *String* variables and/or *String* literals
- Two string concatenation operators are provided
  - **Concatenation operator**: **+**
    - At least one operand must be a *String*
  - **Shortcut concatenation operator**: **+=**
    - The LHS operand must already be a *String*

String Concatenation Examples

- *String* variable + *String* variable

```
String str1 = new String("Thing 1");
String str2 = new String("Thing 2");
String str3 = str1 + " + " + str2;                    // "Thing 1 + Thing 2"
```

- *String* literal + numerical variables

```
int num = 40;
System.out.println("num = " + num);                   // "num = 40"
```

- *String* literal + *String* variable(s)

```
String toGet = " list: ";                             // "list: "
String item = "milk";
toGet += (item + ", ");                               // "list: milk, "
item = "bananas";
toGet += item;                                        // "list: milk, bananas"
```

- *String* literal + object

```
SimpleDate myBirthday = new SimpleDate(7, 18, 2015);
System.out.println("my BD = " + myBirthday.toString() );   // explicit toString()
System.out.println("my BD = " + myBirthday);               // implicit toString()
```

Strings and *println*()

- Do the prior examples look familiar??
  - We've been using this technique for weeks, to display program outputs to the console
- System.out.println() expects some *String* arg that is to be printed
  - If any of the arguments are non-*String*, they are **implicitly** converted to a *String* equivalent
  - Every Java class has either:
    - An explicit **toString()** method, or…
    - An inherited **toString()** method from its parent class, or…
    - An inherited **toString()** method from its ultimate ancestor class, ***Object***
  - This also works with the 8 primitive datatypes:
    - Numbers get implicitly converted to their *String* equivalents

Common ***String*** "Gotchas"

- The **first character** is at **0**, not **1** **(0-based)**
- The **last character** is at **length()-1**, not **length()**
- When extracting substrings, specify an end character index of 1 past where you really want
- The following errors will generate a ***StringIndexOutOfBoundsException***
  - Negative start or end index
  - Start or end index past the last character of the *String*

String Parsing

- **String parsing** (or **tokenizing**) refers to the process by which a given string is **parsed** (separated) into smaller chunks called **tokens**, using **delimiters**
- **Default delimiters** are usually the standard Java whitespace characters
  - Space, tab, and newline are the typical ones
- **Other common delimiters** include
  - Commas, periods, colons, semicolons
  - Or any other user-specified character
- String parsing ties together multiple *String* capabilities
  - But to accomplish parsing, we need a few more capabilities from other *String* methods

Delimited String Examples

- *What is the delimiter in each case??*

- Date:
  10/11/2015
- IP address:
  194.154.46.14
- Directory path:
  C:/Program Files/Java/jdk1.7.0_71/bin/javac
- Config file or settings:
  LOGFILE: logfile.txt
- Database dump or CSV file
  Janice:Smith:F:7/2/1982:Engineering:95295.00
  Janice,Smith,F,7/2/1982,Engineering,95295.00
- Email address:
  smith.janice@engineering.widgetworld.com

More General *indexOf* Method

Return type	Method name and argument list
`int`	`indexOf( String searchString, int fromIndex )` returns the index of the first occurrence of *searchString*, starting at *fromIndex*, or -1 if not found
`int`	`indexOf( char searchChar, int fromIndex )` returns the index of the first occurrence of *searchChar*, starting at *fromIndex*, or -1 if not found

Examples:

```
String parseStr= new String("lions, tigers, bears, oh my");
int index1, index2;
index1 = parseStr.indexOf( ',' );              // index1 = 5
index1 = parseStr.indexOf( ',' , 0 );          // index1 = 5 again
index2 = parseStr.indexOf(',' , index1+1 );    // index2 = 13, 2nd comma
```

trim Method

Return type	Method name and argument list
`String`	`trim( )` returns a copy of the string, with leading and trailing whitespace omitted

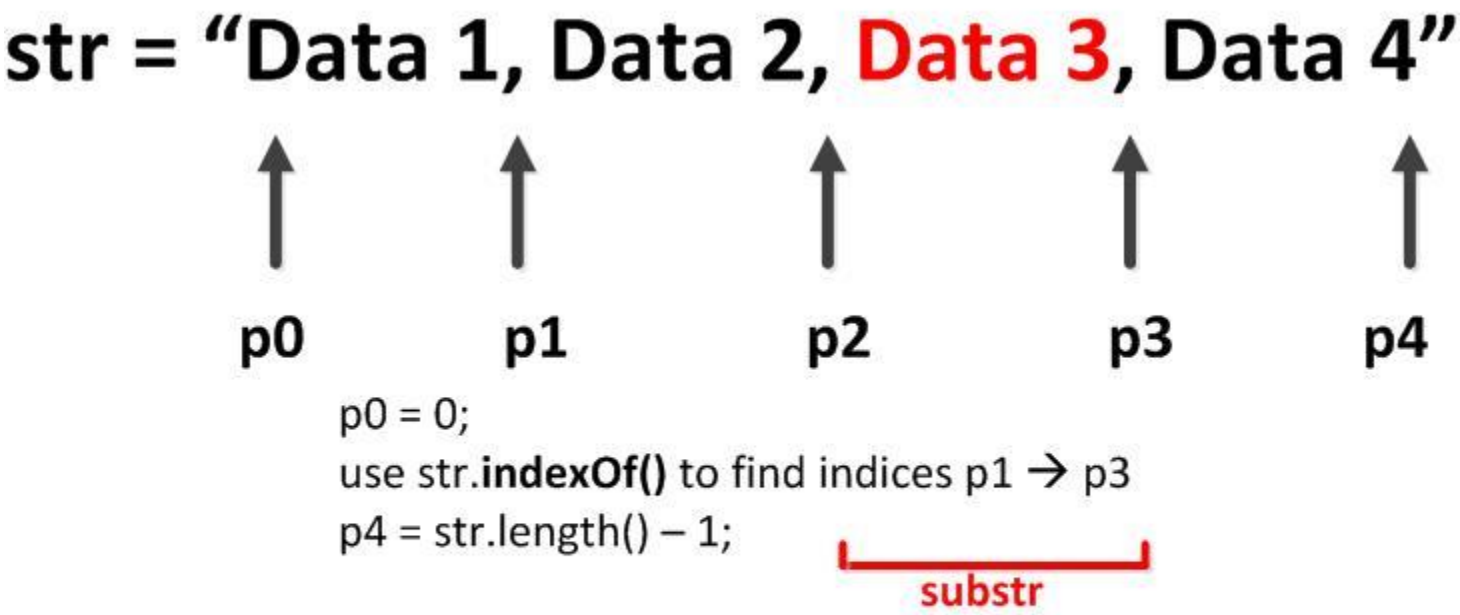
Examples:

```
String  beforeString = “  trim me  “;
String afterString = beforeString.trim();          // now “trim me”
System.out.println( beforeString.length() );    //  11
System.out.println( afterString.length() );      //   7
```

Parsing Algorithm, Algorithmically

- A general approach to parsing a string (as in Strings HW)
  - What is the **string** to be parsed?
  - What is the desired **delimiter**?
  - Find the delimiter **locations**
    - Use **indexOf**() methods
    - Alternate form likely needed (see prior slides or *String* API)
  - Extract **substrings** between delimiter locations
    - Use **substring**() method
  - **Clean up** substrings, if needed
    - Use **trim**() method, if non-whitespace delimiters (see prior slides or *String* API)

Parsing Algorithm, Visually



String Parsing Example

```
14 public class StringParsing {
15
16     public static void main(String [] args) {
17
18         // objective: extract tigers as TIGERS
19
20         // declarations
21         String inStr = new String("lions, tigers, bears, oh my");
22         int loc1, loc2;
23         char sepChar = ',';
24         String tigerStr;
25
26         // find the delimiters (the substring bounds)
27         loc1 = inStr.indexOf(sepChar);
28         loc2 = inStr.indexOf(sepChar, loc1+1);
29
30         // extract the substring, clean it up, upper case it
31         tigerStr = inStr.substring(loc1+1, loc2);
32
33         // clean up the extracted substring
34         tigerStr = tigerStr.trim();
35
36         // convert the string to upper case
37         tigerStr = tigerStr.toUpperCase();
38
39         // output the string and its length, to be certain it's right
40         System.out.println("substring: " + tigerStr +
41                            "\nlength: " + tigerStr.length());
42
43     } // end main
44
45 } // end class
```

Notes:

See ***StringParsing.java*** in **Example Source Code**

Line 28: 2nd search is from one past first delimiter

Line 31: Extract is from one past first comma

```
----jGRASP exec: java StringParsing

substring: TIGERS
length: 6

----jGRASP: operation complete.
```

For Next Time

- **Lecture Prep**
  - Text readings and lecture notes
- **Assignments**
  - See slide 2 for new/current/past due assignments