Università degli Studi dell'Aquila

Dipartimento di Ingegneria e Scienze dell'Informazione e Matematica

Università degli Studi dell'Aquila

Corso di Algoritmi e Strutture Dati con Laboratorio

The String and Scanner classes

The `String` class

- A `String` object is a variable that contains a string (a sequence of characters) and can call methods in the `String` class
- Objects cannot be explicitly declared in Java; instead, programmers declare reference variables

```
String s;
```

- In this declaration, `s` is not a `String` object, but rather a `String` reference, that is, a variable that can hold the address of a `String` object.
- **Remark: Strings are constant;** their values cannot be changed after they are created.

The `String` class: constructors

- **`public String()`**

  Initializes a newly created String object so that the String object represents an **empty string**.

- **`public String(String original)`**

  Initializes a newly created String object so that it represents the same sequence of characters as the argument; in other words, the newly created string is a **copy of the argument string**.

The String class: constructors

- **`String(char[] value)`**

  Allocates a new String so that it represents the sequence of characters currently contained in the character array argument.

- **`String(char[] value, int offset, int count)`**

  Allocates a new String that contains characters from a subarray of the character array argument, starting at index `offset` for length `count`

The `String` class: methods

The class String includes methods for:

- examining individual characters of the sequence,
- comparing strings,
- searching strings,
- extracting substrings,
- creating a copy of a string with all characters translated to uppercase or to lowercase.

The `String` class: methods

`boolean equals(Object anObject) //override`

Compares this string to the specified object.

`boolean equalsIgnoreCase(String anotherString)`

Compares this String to another String, ignoring case considerations.

`String toString() //override`

This object (which is already a string!) is itself returned.

`int compareTo(String anotherString)`

Compares two strings lexicographically.

`int compareToIgnoreCase(String str)`

Compares two strings lexicographically, ignoring case differences.

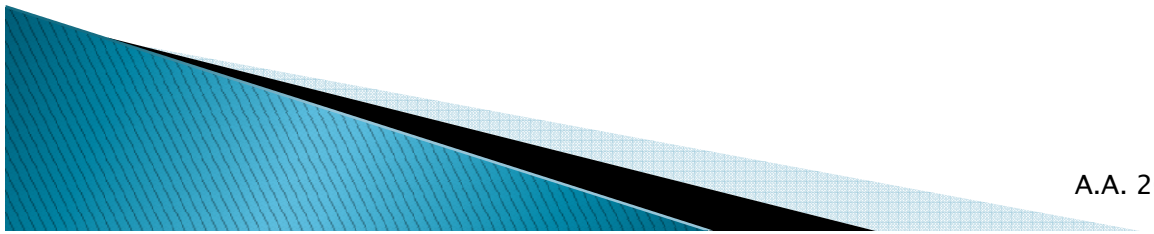
The `String` class: methods

`int length()`
Returns the length of this string.

`boolean isEmpty()`
Returns true if, and only if, `length()` is 0.

`char charAt(int index)`
Returns the char value at the specified index.

`string concat(String str)`
Concatenates the specified string to the end of this string.



The `String` class: methods

`int indexOf(String str)`
Returns the index within this string of the first occurrence of the specified substring.
`int indexOf(String str, int fromIndex)`
Returns the index within this string of the first occurrence of the specified substring, starting at the specified index.
`int lastIndexOf(String str)`
Returns the index within this string of the last occurrence of the specified substring.
`int lastIndexOf(String str, int fromIndex)`
Returns the index within this string of the last occurrence of the specified substring, searching backward starting at the specified index.

The `String` class: methods

`String replace(char oldChar, char newChar)`

Returns a new string resulting from replacing all occurrences of oldChar in this string with newChar.

`String substring(int beginIndex)`

Returns a new string that is a substring of this string.

`String substring(int beginIndex, int endIndex)`

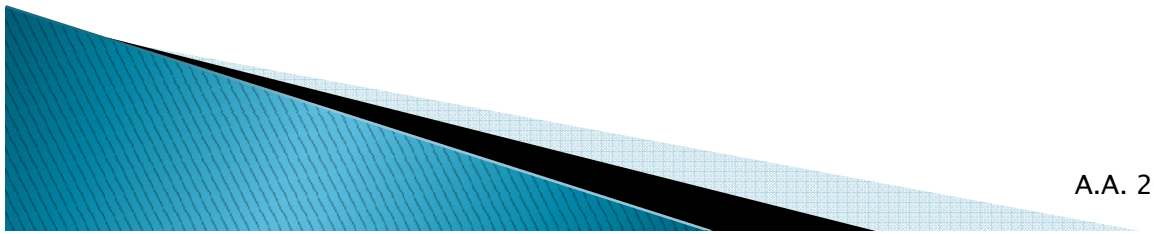
Returns a new string that is a substring of this string.

The `String` class: methods

`char[] toCharArray()`
Converts this string to a new character array.

`String toLowerCase()`
Converts all of the characters in this String to lower case using the rules of the default locale.

`String toUpperCase()`
Converts all of the characters in this String to upper case using the rules of the default locale.



The `String` class: example

```
String s = new String();
```

- Actually, the argument `s` is a reference to `""`

```
String t = new String ("Aloha");
```

- Actually, the argument `t` is a reference to `"Aloha"`

```
s.length() // returns 0
```

```
t.toLowerCase()
```

- returns (a reference to) `"aloha"`
- t is still a reference to `"Aloha"`

The `String` class: example

- Determine the output:

```
System.out.println (t.indexOf ("ha"));
System.out.println (t.indexOf ("a"));
System.out.println (s.indexOf ("ha"));
```

Hint: Indexes start at 0.

The `String` class: example

```
String y1 = "Aloha";
String y2 = "Aloha";
```

- **These statements create two references, `y1` and `y2`, to the same string object, so**

```
y1 == y2   // returns true
y1 == t    // returns false
```

- **but**

```
y1.equals (t)   // returns true
```

The `String` class: example

`String z = new String ("Aloha");`

- Determine the result returned in each case:

```
s.equals ("")
s == ""
t.equals ("Aloha")
t == "Aloha"
t.equals (null)
t.equals (z)
t == z
w.equals (null)
w == null
```

The `String` class: example

```
System.out.println("abc");
String cde = "cde";
System.out.println("abc" + cde);
String c = "abc".substring(2,3);
String d = cde.substring(1, 2);
```

The `Scanner` class

- The **`Scanner`** class allows users easy access to text data. A *text* is a sequence of lines, separated by end-of-line markers. A Scanner object skips over irrelevant characters called *delimiters* (spaces, tabs, end-of-line markers, …) to access *tokens* (primitive types and strings; for example, integers).
- The default whitespace delimiter used by a scanner is as recognized by **`Character.isWhiteSpace`**

The `Scanner` class

- The text can be entered from the keyboard, entered from a file, or consist of a string of characters. The `Scanner` class has constructors to initialize each of the three kinds of Scanner object.

```
Scanner keyboardScanner = new Scanner (System.in);
int n = keyboardScanner.nextInt();
```

- Suppose the input from the keyboard is 74
- Then the token 74 will be stored in the variable n.

The Scanner class

```
Scanner keyboardScanner=new Scanner (System.in);
int j, k, m, n;
j = keyboardScanner.nextInt();
k = keyboardScanner.nextInt();
m = keyboardScanner.nextInt();
```

- Suppose the input from the keyboard is: `74 58 0`
- They are skipped over, and the tokens 74, 58 and 0 will be assigned to the variables j, k and m, respectively.

The `Scanner` class

- The `hasNextInt` method tests to see if the next token is an `int` value:

```
Scanner keyboardScanner = new Scanner (System.in);
int bonus;
if (keyboardScanner.hasNextInt())
    bonus = keyboardScanner.nextInt();
else
    bonus = 0;
```

- The Scanner class also has methods to scan in and check for other primitive values, such as `nextDouble()`, `nextLong()`, `hasNextDouble()`, …

The `Scanner` class

- The `next()` method scans in the next token as a string of characters:

```
Scanner keyboardScanner = new Scanner (System.in);
String s = keyboardScanner.next();
```

- Suppose the input from the keyboard is `gentle`
- Then the variable `s` will contain a reference to the string "gentle".
- The `next()` method can help with the scanning of dirty data. Assume the keyboard input is supposed to consist of positive int values, ending with a value of -1 (such a terminal value is called a *sentinel*)

The Scanner class

```
final int SENTINEL = -1;
Scanner keyboardScanner = new Scanner (System.in);
int sum = 0, score;
while (true)
      if (keyboardScanner.hasNextInt()) {
            score = keyboardScanner.nextInt();
            if (score == SENTINEL)
               break; // terminate execution of loop
            sum += score;

      } // if next token is an int

      else keyboardScanner.next();

System.out.println (sum);
```

The `Scanner` class

- Suppose the input entered from the keyboard is

```
90   100 50
7z   80
5f
-1
```

- The preceding loop would be executed 7 times, but the erroneous values `7z` and `5f` would be skipped over. The output would be **320**
- If the else part of the preceding if statement were omitted, an infinite loop would occur because 7z would fail the `hasNextInt()` condition.

The `Scanner` class

- Suppose the entire body of the loop were replaced with

```
score = keyboardScanner.nextInt();
if (score == SENTINEL)
  break; // terminate execution of loop
sum += score;
```

- Then an error (technically, an exception, as defined later) would occur at run time because `7z` is not an `int` value.

The `Scanner` class

- Sometimes the remainder of an input line should be skipped over if an incorrect value is discovered during scanning.
- For example, it might be that each input line is supposed to contain a name, grade point average, class year and age, with "***" as the sentinel. If the grade point average is not a double value (or the class year or age is not an int value), the rest of the line should be skipped.

The Scanner class

```
final String SENTINEL = "***";
Scanner keyboardScanner = new Scanner (System.in);
String name;
int classYear, age;
double gpa;
while (true){
      //name
      name = keyboardScanner.next();
      if (name.equals (SENTINEL)) break;
       // grade point average
      if (!keyboardScanner.hasNextDouble()) {
            keyboardScanner.nextLine();
            continue; // start another iteration of the loop
      } // if next token is not a double
      gpa = keyboardScanner.nextDouble();
```

The Scanner class

```
        // class year
        if (!keyboardScanner.hasNextInt())  {
              keyboardScanner.nextLine();
              continue; // start another iteration of loop
        } // if next token is not an int
        classYear = keyboardScanner.nextInt();
        // age
        if (!keyboardScanner.hasNextInt()){
               keyboardScanner.nextLine();
               continue; // start another iteration of loop
        } // if next token is not an int
        age = keyboardScanner.nextInt();
        // process name, gpa, classYear and age …
} // while
```

The `Scanner` class

- For scanning over a file, the constructor is different from keyboard scanning, but the "`hasNext()`, `hasNextInt()`, `next()`, `nextDouble()`, …" methods are still available.
- For example:

```
Scanner fileScanner =new Scanner(new File ("data"));
if (fileScanner.hasNextDouble())
      double gpa = fileScanner.nextDouble();
else
      fileScanner.next();
```

The `Scanner` class

- Sentinels are not used in file scanning because it is too easy to forget to append the sentinel to the end of the file. (With keyboard input, a scan loop will continue until the sentinel is entered.) So a typical scanning loop with `fileScanner` will start with

```
while (fileScanner.hasNext())
or
while (fileScanner.hasNextLine())
or
while (fileScanner.hasNextInt())
```

The Scanner class

- Scanning over a line is fairly straightforward. For example, suppose we want to add up the int values in a line, and skip over the non-int values.

```
Scanner lineScanner = new Scanner ("70 o2 50");
int sum = 0;
while (lineScanner.hasNext())
      if (lineScanner.hasNextInt())
            sum += lineScanner.nextInt();
      else
            lineScanner.next(); // skip non-int
```

The `Scanner` class

- Often a program needs all three kinds of Scanner object: a keyboard scanner to get the name of a file, a file scanner to access each line in that file, and a line scanner to access the tokens in a line.
- Scanner defines where a token starts and ends based on a set of **delimiters**.
- The default delimiters are the whitespace characters.
- You can specify the delimiters for your scanner with the **`useDelimiter` method**.

The Scanner class

- For example, in order to set delimiters to space and comma: `", *"` tells Scanner to match a comma and zero or more spaces as delimiters.

```
Scanner src = new Scanner(new File ("Test.txt"));
src.useDelimiter(", *");
```

- For example, if you want the tokens in a string line to be upper- or lower-case letters, any other character will be a delimiter:

```
Scanner sc =new Scanner (line).useDelimiter ("[^a-zA-Z]+");
```

- the '+' can be read as "one or more occurrences" and '^' means "except". So a delimiter is one or more occurrences of any character except a letter.

The Scanner class: example

```
String input = "1 fish 2 fish red fish blue fish";
Scanner s = new Scanner(input).useDelimiter("\\s*fish\\s*");
System.out.println(s.nextInt());
System.out.println(s.nextInt());
System.out.println(s.next());
System.out.println(s.next());
s.close();
```

- prints the following output:

```
1
2
Red
blue
```

Espressioni regolari

- Le espressioni regolari rappresentano uno strumento molto potente per lavorare sulle stringhe ed elaborare testi
- Consentono di specificare modelli complessi di testo (pattern) che possono essere cercati in una stringa
- Possono essere utilizzate, sia per convalidare i dati, sia per effettuare ricerche all'interno di un testo. La sintassi di questo pseudo-linguaggio è molto flessibile e consente di creare espressioni in base alle proprie esigenze.
- Dalla versione 1.4 di Java è stato introdotto il package **`java.util.regex`** composto dalle classi Pattern e Matcher che permettono di validare una stringa, o ricercare un testo al suo interno, a partire da un'espressione regolare.

Espressioni regolari

- Per definire un'espressione regolare è necessario conoscere alcune regole base:
- [...] Insieme di caratteri validi alternativi;
- | Modelli alternativi
- [^...] Insieme negato di caratteri validi;
- - Intervallo;
- && Intersezione;
- . Qualunque carattere;
- + Concatenazione;

Espressioni regolari: cardinalità multipla

- RE* (0 o più occorrenze dell'espressione RE);
- RE{n} (esattamente n occorrenze dell'espressione RE);
- RE{n,} (almeno n occorrenze dell'espressione RE);
- RE{n,m}
  (almeno n occorrenze dell'espressione RE, ma non più di m).

Espressioni regolari: forme abbreviate

- \d Carattere numerico. Corrisponde all'insieme [0-9];
- \D Carattere diverso da un numero. Corrisponde all'insieme [^0-9];
- \s White space (' ', tab (\t), carriage return (\r), newline (\n), form feed (\f) and vertical tab \x0B) .
- \S Carattere diverso dai white spaces. Corrisponde all'insieme [^\s];
- \w Parola alfanumerica. Corrisponde all'insieme [a-zA-Z_0-9];
- \W Parola costituita da caratteri speciali. Corrisponde all'insieme [^\w].

Espressioni regolari: forme abbreviate

- Remark: la stringa delimiter "`\\s+`" denota tutti i white spaces. Cioè "`\\s`" è equivalente a:

  "`[ \\t\\n\\x0B\\f\\r]+`"
- Poiché \ è un carattere speciale in Java, bisogna includere un \ aggiuntivo
- Il carattere * nelle espressioni regolari è quantificatore. Per indicare il carattere vero e proprio si scrive * e dunque per saltare un carattere * la stringa delimiter corretta sarà "*"

Esempi

Di seguito riportiamo alcune espressioni comunemente utilizzate:

- indirizzo email
  [a-zA-Z0-9._%-]+@[a-zA-Z0-9.-]+\\.[a-zA-Z]{2,4}
- data in formato mm/gg/aaaa
  (0[1-9]|1[012])[- /.](0[1-9]|[12][0-9]|3[01])[-/.](19|20)\d\d
- url http
  http\://[a-zA-Z0-9\-\.]+\.[a-zA-Z]{2,3}(/\S*)?
- codice fiscale
  [a-zA-Z]{6}\d\d[a-zA-Z]\d\d[a-zA-Z]\d\d\d[a-zA-Z]

Exercises

- Write and run a small program in which an input string is read in and the output is the original string with each occurrence of the word “is” replaced by “was”.

  No replacement should be made for an embedded occurrence, such as in “this” or “isthmus”.

Exercises

- Write and run a small program in which the end user enters three lines of input. The first line contains a string, the second line contains a substring to be replaced, and the third line contains the replacement substring. The output is the string in the first line with each occurrence of the substring in the second line replaced with the substring in the third line. No replacement should be made for an embedded occurrence, in the first line, of the substring in the second line. For example, suppose the original string is "The snow is now on the ground.", the target string is "now", and the replacement string is "melting". The output will be "The snow is melting on the ground.".

Exercises

- Create a keyboard scanner in which the tokens are unsigned integers, and write the code to determine the sum of the integers.

  Note: -5 will be scanned as the unsigned integer 5, and the minus sign will be skipped over as a delimiter.