COSC 1020

Yves Lespérance

Lecture Notes

Week 6 — Java Strings

Recommended Readings:
Horstmann: Ch. 3 Sec. 7 & Ch. 5 Sec. 2.3
Lewis & Loftus: Ch. 2 Sec. 6 & Ch. 3 Sec. 4

Character Strings

A *character string* is a sequence of 0 or more characters. A string can contain a word, a sentence, or any amount of text.

In Java, character strings are objects that are instances of the predefined class `String`. So `String` is not a primitive data type like `int` (the syntax can be misleading).

`String` literals are written between double-quotes, e.g. `"Hi there!"`, `"R2d2"`, `" "`, `""`.

When the + operator is used with `String` arguments, it returns the string obtained by joining together its two arguments; this is called *concatenation*, e.g.

```
String greeting = "Hello! Today is ";
String date = "Sept. 15";
String openingMsg = greeting + date;
IO.println(openingMsg);
date = "Sept. 22";
openingMsg = greeting + date;
IO.println(openingMsg);
```


As soon as one argument of + is a string, it will automatically convert the other argument into a string and concatenate the result, e.g.

```
int day = 15;
String date = "Sept. " + day;
```

The `String` class provides a number of methods for operating on string objects.

`charAt(`*i*`)` returns the *i* th character of the string; characters are numbered starting from 0, e.g.

```
String greeting = "Hello! Today is ";
char c = greeting.charAt(4);
// sets c to 'o'
```

`length()` returns the length of the string, e.g.

`greeting.length()` returns 16

`substring(`*i*`,`*j*`)` returns the substring from the *i* th character up to and excluding the *j* th character, e.g.

```
IO.println(greeting.substring(7,12));
// prints "Today"
```


`substring(`*i*`)` just returns the substring between the *i* th character and the end of the string, e.g.

```
IO.println(greeting.substring(7));
// prints "Today is "
```

The `String` class provides many other methods, e.g. `toUpperCase()`, `toLowerCase()`, `trim()`, etc. See the Java API for details.

To convert a number into a string, you can do

```
int dateNum = 15;
String dateStr = "" + dateNum;
```

or

```
String dateStr = Integer.toString(dateNum);
```

and similarly for `Double`, etc.

To convert a string containing a number into an `int`:

```
String dateStr = "19";
int dateNum = Integer.parseInt(dateStr);
```

and similarly for `double` with `Double.parseDouble(s)`.

Stings are *immutable*, i.e., once a string has been created, it can't be modified. The + operator creates a new string each time it is called, and similarly for other methods. This can be inefficient. Java also supplies the class `StringBuffer` for cases where we want to repeatedly modify a string.

Comparing Strings and Other Objects

You cannot use the relational operators to compare strings and other objects. `o1 == o2` is true only if `o1` and `o2` refer to the same object.

For strings, we use the following methods:

`equals(`*s*`)` returns `true` iff the string object is identical to the string argument *s*, i.e. same length and same characters in corresponding positions; e.g.

```
String s1 = "abc";
```

`s1.equals("abc")` returns `true`,
`s1.equals("abc ")` returns `false`,
`s1.equals("aBc")` returns `false`,
`s1.equalsIgnoreCase("aBc")` returns `true`.

`compareTo(`*s*`)` returns `0` if the object is identical to the string argument *s*, a *negative integer* if the object comes before *s* in the lexicographic ordering, and a *positive integer* if the object comes after *s*; lexicographic ordering is similar to dictionary ordering but Unicode values are used to compare each character, e.g. `"abc"` < `"acc"`, `"abc"` < `"abcd"`, `"Zoo"` < `"at"`; e.g.

`s1.compareTo("abc") == 0` returns `true`,
`s1.compareTo("acc") < 0` returns `true`.

In general, `s1.compareTo(s2)` *op* `0` returns `true` iff `s1` *op* `s2`.

For other object types, see if the class provides an equality testing method.

String Search

`indexOf(`*s2*`,`*p*`)` searches for string *s2* within `this` string object starting at position *p*; if a match is found, the method returns the starting position of the match, otherwise -1 is returned.

`indexOf(`*s2*`)` works as `indexOf(`*s2*`,0)`.

E.g.

```
String s1 = "abracadabra";
String s2 = "br";
int pm = s1.indexOf(s2);
IO.println(pm); // prints 1
pm = s1.indexOf(s2,pm+s2.length());
IO.println(pm); // prints 8
pm = s1.indexOf(s2,pm+s2.length());
IO.println(pm); // no match, prints -1
```

Also `lastIndexOf` which searches from the right end of the string.

E.g. Date Conversion

Convert a long form date to a short form, e.g.

```
zebra 355 % java ShortenDate
Enter long date: September 5, 2002
Short form is : 05/09/02
```

```
import type.lang.*;
public class ShortenDate
{  public static void main(String[] args)
   {  IO.print("Enter long date: ");
      String longDate = IO.readLine();
      int pSep1 = longDate.indexOf(" ");
      int pSep2 = longDate.indexOf(",");
      String month = longDate.substring(0,pSep1);
      month = month.toLowerCase();
      final String monthTbl = "01january02february"
         + "03march04april05may06june07july08august"
         + "09september10october11november12december";
      int pTbl = monthTbl.indexOf(month);
      month = monthTbl.substring(pTbl-2,pTbl);
      String day = longDate.substring(pSep1+1,pSep2);
      day = day.trim();
      if(day.length() < 2)
      { day = "0" + day;
      }
```



```
      String year = longDate.substring(pSep2+1)
      year = year.trim();
      year = year.substring(year.length()-2,
                            year.length());
      String shortDate = day+"/"+month+"/"+year;
      IO.println("Short form is : " + shortDate);
   }
}
```


E.g. Replace All Occurrences

Replace all occurrences of string *pat* in string *s* by string *rep*, e.g.

```
zebra 354 % java ReplaceAll
Enter original string...
abracadabra
Enter substring to be replaced...
br
Enter replacement string...
brr
Modified string is...
abrracadabrra
```



```
import type.lang.*;
public class ReplaceAll
{  public static void main(String[] args)
   {  IO.println("Enter original string...");
      String s = IO.readLine();
      IO.println("Enter substring to be replaced...");
      String pat = IO.readLine();
      IO.println("Enter replacement string...");
      String rep = IO.readLine();
      int patl = pat.length();
      int repl = rep.length();
      int pos = 0;
      boolean done = false;
      for(; !done ;)
      {  pos = s.indexOf(pat,pos);
         if(pos >= 0)
         {  s = s.substring(0,pos) + rep
                + s.substring(pos+patl);
            pos = pos + repl;
         }
         else
         {  done = true;
         }
      }
      IO.println("Modified string is...");
      IO.println(s);
   }
}
```

Conditional Loops

In many cases, you don't know in advance how many repetitions are required and you want to repeat the operations as long as *some condition holds*. Java provides several constructs for this.

E.g. Determine how many months it takes to pay back a loan given the loan amount, monthly payment amount, and interest rate.

Solution Algorithm

1. Initialize *months_required* to 0.
2. Repeat (a), (b), and (c) while *amount_owed* is greater than 0:
   (a) Add *monthly_ interest* to *amount_owed*.
   (b) Subtract *monthly_payment* from *amount_owed*.
   (c) Increment *months_required* by 1.
3. Report *months_required* as the answer.

Java code

```
IO.print("Enter loan amount: ");
double amountOwed = IO.readDouble();
IO.print("Enter monthly payment: ");
double monthlyPayment = IO.readDouble();
IO.print("Enter interest rate: ");
double interestRate = IO.readDouble();
int monthsRequired = 0;
while (amountOwed > 0)
{  amountOwed = amountOwed +
               amountOwed * interestRate;
   amountOwed = amountOwed - monthlyPayment;
   monthsRequired++;
}
IO.println("It will take " + monthsRequired +
   " months to pay the loan.");
```

Depending on the application, you may need to check the loop condition at the *beginning* of the loop body, at the *end*, or in the *middle*. For each of these, you use different control structures.

`while` Loops

Often, we want the condition to be tested *at the beginning of each cycle*. For this, Java provides the `while` structure:

`while (`*condition*`)`
    statement

The body of the loop is repeatedly executed, as long as the condition remains true. The condition is tested at the beginning of every cycle. If the condition is false *initially*, the body is *never* executed. If the condition becomes false *during* a loop cycle, the cycle is *completed* nonetheless.

`do` Loops

In other cases, the condition must be tested *at the end of each cycle*. For this, Java provides the `do` structure:

`do`
    statement
`while (`*condition*`);`

The body of the loop is always executed at least once.

E.g. a program to compute a total:

```
IO.println(
   "This program adds up the amounts you enter.");
double total = 0;
double amount;
String response;
do
{  IO.print("Enter an amount to add: ");
   amount = IO.readDouble();
   total = total + amount;
   IO.print("Continue adding amounts (y/n)? ");
   response = IO.readLine();
} while (response.charAt(0) == 'y');
IO.println("Total is $" + total);
```

Exiting in the Middle of the Cycle

In many cases, the natural place to test the loop condition is somewhere *in the middle of the cycle*.

E.g. suppose we want to modify the program to add amounts so that it exits when a negative amount is entered; here's an algorithm in pseudocode:

Initialize *total* to 0.
`loop`
  Read an *amount*.
  `if` *amount* $<$ 0 `then exit the loop`.
  Add *amount* to *total*.
`endLoop`
Print *total*.

A natural way to code this in Java is:

```
double total = 0;
double amount;
while (true)
{  IO.print("Enter an amount: ");
   amount = IO.readDouble();
   if (amount < 0) break;
   total = total + amount;
}
IO.println("Total is $" + total);
```

`break` exits immediately from nearest enclosing `while` (or `switch`, `for`, or `do`). It is easy to write code that is hard to understand with `break`. It should only be used as above and in `switch` (another selection construct).

Another way to write a loop that exits from the middle is to use a boolean variable:

```
double total = 0;
double amount;
boolean done = false;
while (!done)
{  IO.print("Enter an amount: ");
   amount = IO.readDouble();
   if (amount < 0)
      done = true;
   else
      total = total + amount;
}
IO.println("Total is $" + total);
```



```
import type.lang.*;
public class ReplaceAll2
{  public static void main(String[] args)
   {  IO.println("Enter original string...");
      String s = IO.readLine();
      IO.println("Enter substring to be replaced...");
      String pat = IO.readLine();
      IO.println("Enter replacement string...");
      String rep = IO.readLine();
      int patl = pat.length();
      int repl = rep.length();
      int pos = 0;
      while(true)
      {  pos = s.indexOf(pat,pos);
         if(pos < 0)
            break;
         s = s.substring(0,pos)
             + rep + s.substring(pos+patl);
         pos = pos + repl;
      }
      IO.println("Modified string is...");
      IO.println(s);
   }
}
```

Note that

```
for (init; test; step)
    statement
```

is equivalent to

```
{   init;
    while (test)
    {   statement
        step;
    }
}
```


Which Looping Construct to Use?

If number of iterations is known before loop starts use `for` (counted loop).

If repeating as long as a conditon holds (conditional loop):

- test at beginning use `while(`*condition*`){...}`,
- test at end use `do{...}while(`*condition*`)`,
- test in the middle use `while(true){...if(`*condition*`)break;...}`.

If iterating over a collection or set of input records, can use `for`.

Don't use `for` for weird iteration constructs such as:

```
for(a = a/2; count < ITERATIONS; IO.println(xnew))
```

Instead write

```
a = a / 2;
while(count < ITERATIONS)
{   ...
    IO.println(xnew);
}
```


Infinite Loops

You have to make sure that a loop eventually exits, otherwise you have an *infinite loop*! E.g. 1:

```
double total = 0;
double amount;
while (true)
{  // input left out
   if (amount < 0) break;
   total = total + amount;
}
```

E.g. 2, a poorly written countdown program:

```
int count = IO.readInt();
while (count != -1)
{  IO.println(count);
   count--;
}
```