CS112 Lecture: Characters and Strings

Last Revised 3/31/08

Objectives:

1. To introduce the data type char and related basic information (escape sequences, Unicode).
2. To introduce the library classes String and StringBuffer
3. To further discuss the difference between primitive and object (reference) types
4. To show some examples of string processing

Materials:

1. Projectable: Character escape codes
2. Projectable: ASCII Code table
3. Projectable: Unicode character table
4. Dr. Java for demonstrations
5. StringsDemo.java, .class, .dat + printed copy (*Appleworks version*) to hand out
6. EvenBetterReverse program demo from Arrays lecture

I. Introduction

A. The importance of textual information

1. The fact that we call the machines we work with "computers" reflects the fact that computers were originally conceived of as machines for doing numerical computations (e.g. computing artillery range tables during World War II).

2. Early in the history of computers, it was realized that there was some need to provide for processing textual information - first primarily for the purpose of labelling output, but later as information in its own right.

3. Today, processing of textual information has eclipsed numerical computation as a major use of computers. Many computer applications are primarily concerned with the storage, manipulation, and or transmission of text - e.g.

   a) Word processors
   b) Databases
   c) Email
   d) The web (here, of course, text is augmented by the use of graphics)

B. Internally, textual information is represented numerically.

1. Almost all programming languages support a data type called char (or character) which represents individual characters by numeric codes - e.g. in the code used in Java the letter capital A is represented by the numeric code 65.

2. Almost all programming languages also provide some facility for representing *character strings* - or sequences of characters. Typically, a character string is represented by a series of memory cells, each of which holds the numeric code representing one of its characters.

3. Computations involving characters are performed by doing arithmetic on their numeric codes - e.g.

   a) Comparison of characters is done by comparing their numeric codes. Comparison of strings is done by comparing successive characters until a pair that differs is found.

      This works because the numeric codes are assigned in such a way as to correspond to our conventional sense of alphabetical order - e.g. the code for "A" is less than the code for "B", which in turn is less than the code for "C" ...

   b) Conversion from upper case to lower case, or vice versa, is done by adding a constant value to the codes for individual characters.

      This works because the difference between the numeric code for "A" and that for "a" is the same as the difference between the code for "B" and that for "b", which in turn is the same as the difference between the code for "C" and that for "c" ...

II. The Data Type char

A. In Java, the primitive type char is used for representing individual characters.

1. We can declare a variable to be of type char, just as we can declare a variable to be of type int or float.

2. We can write constants of type char - represented in one of two ways:

   a) A single character enclosed in single quotes - e.g.

      'A' is a constant of type char

b) Because certain characters cannot be represented this way, an alternate notation is provided by the use of *escape sequences,* which consist of a backslash followed by one of several special codes:

PROJECT - Character escape codes

B. Java differs from most programming languages that have preceeded it in that Java uses 16 bits to store a char, as compared to 8 in most other languages.

1. This difference represents the evolution of computers from machines that were developed in the Western world, which have traditionally used languages based on relatively small alphabets, to being used globally, thus having to cope with many alphabets as well as languages that have very large alphabets.

2. Until recently, most computer systems used a character encoding known as ASCII (American Standard Code for Information Interchange), which is basically a 7-bit code capable of representing 128 different characters, with the 8th bit used to provide an alternate set of additional characters for special purposes.

   PROJECT: ASCII Code table
   EXPLAIN structure of code

   a) 0..31 + 127 = control characters
   b) 32 = space
   c) 48..57 = digits
   d) 65..90 = upper-case letters
   e) 97..122 = lower-case letters
   f) various punctuation symbols are interspersed wherever there is room

   Trivia question (for homework bonus point): why is it that, while most of the control characters have codes at the beginning of the range, but delete has a code at the every end (all 1's in 7-bit binary)?

3. To accomodate non-Roman alphabets, Java uses an encoding known as Unicode. ASCII is a subset of Unicode, in the sense that the first 256 encodings in Unicode are the same as the corresponding ASCII characters, including the standard extension to 8 bits.

   TRANSPARENCY: Unicode character table

4. However, most platforms used to run Java only support the standard ASCII set. Thus, trying to output a Unicode encoding above 255 is likely to produce incorrect results unless the operating system is equipped with an appropriate extended character set.

C. Most of the standard Java operators may be used with values of type char, which are treated as a 16 bit unsigned integer. For example, if c and d are declared as variables of type char, the following are all legal:

DEMO using Dr. Java

```
char c = 'C', c1 = 'C', d = 'D';
```

1. == (boolean result) true just when two characters have the same integer code (represent the same char)

```
DEMO        c == d
            c == c1
            c == 'C'
```

2. < (boolean result) true just when first character's integer code is less than second - i.e. when first alphabetically preceeds second

   (Can also use <=, >, >=, !=)

```
DEMO        c < d
            c < c1
            c <= c1
            d > c
```

3. Integer arithmetic operations (+, -, *, /, %) on chars and/or ints (int result) based on the integer code for the character(s)

```
DEMO        c + 1
            c * 2
            c + d
```

4. Note that the result of the above operations is either boolean or int, not char!

D. It is possible to convert between types int and char by using casts

1. DEMO

```
            c + 1
            (char) (c + 1)
            (char) (c + 'a' - 'A')
```

2. A comment on practice - letting the computer calculate the difference, rather than "hard-wiring" in a value (which happens to be 32 in this case) is <u>much</u> safer and more readable! If one were doing many conversions, an approach like this might be used:

```
static final int LC_UC_DIFFERENCE = 'a' - 'A';
(char) (c + LC_UC_DIFFERENCE)
```

Demo with Dr. Java - omit static!

III. Character Strings

A. We have noted that, in addition to supporting a representation for individual characters, most programming languages also support some representation for character strings. The type string, however, is always somewhat problematic.

1. On the one hand, it seems like the type string should be a primitive type, of the same sort as integers, floats, etc.

2. On the other hand, the amount of storage needed for representing a string varies widely.

   a) A true primitive type always requires a fixed number of bits to store - e.g. 32 bits for an int or a float; 64 for a long or a double.

   b) The number of bits needed for a string is a function of the number of characters in it - in the case of Java, 16 bits per character. Thus, one string in a given program may require (say) 64 bits, while another may require 16,000!

B. In Java, strings have a special status. The package java.lang defines a String class. For the most part, strings are treated as objects, like any other objects. However, the compiler "knows" the class String in a way that it does not "know" other classes.

1. The compiler recognizes any sequence of characters (including the escape characters) enclosed in double quotes as a String constant, and creates an appropriate String object when it compiles the program.

   Note: Two quote characters in succession, with nothing in between ("") are valid, and represent an empty string.

2. The compiler recognizes the + operator, when applied to strings, as standing for string concatenation, and generates code to call the concat() method of class String.

   e.g. if s and t are variables of class String, then the following two lines of code are equivalent:

```
DEMO:     String s = "Hello", t = "World";
          s + t
          s.concat(t);
```

Note: + is the only operator allowed with Strings, and String is the only class it is allowed with - the compiler recognizes it specially

3. Under certain circumstances, the compiler generates code to automatically convert a primitive type or another to a string.

   a) In particular, this is done whenever the compiler encounters the "+" operator with either operand being a String.

   b) The compiler "knows" how to generate code to convert any primitive type to a String.

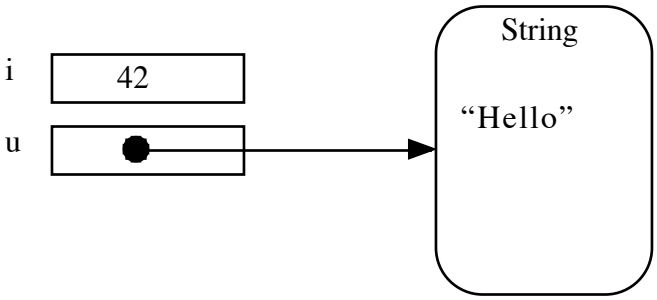
   c) Any object can be converted to a string because the class Object - which is the base class of all classes - includes a method toString() - which can be overridden to perform the conversion more appropriately.

4. If the same string constant occurs twice in the same program, the compiler *may* use the same String object to represent both strings, but is not guaranteed to.

C. Although strings behave in some ways like primitive types, they are actually reference types

1. Consider a state of memory diagram for the following sequence of statements:

```
int i = 42;
String u = "Hello";
```



One consequence of this is that the following comparison is *not necessarily* true (though it might be)

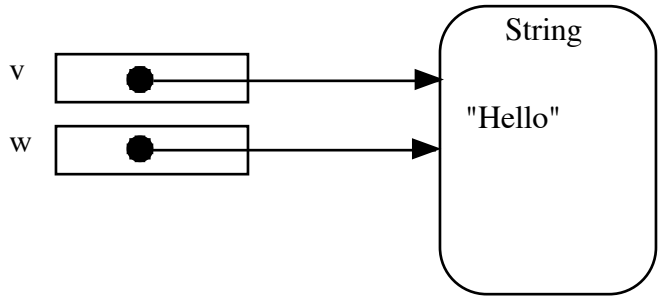
```
u == "Hello"
```

(Whether it is true depends on whether the compiler uses the same String object for both instances of the "Hello" string. This might happen, because - to economize on storage - the JVM sometimes tries to re-use an existing string object for a given value, rather than creating a new one.

Try with Dr. Java

2. Now consider the following scenario instead:

```
String v = "Hello";
String w = v;
```



In this case, the test v == w would be guaranteed to be true.

DEMO with Dr. Java

D. Methods of class String

The class String provides a large number of methods, including the following. (This is just a partial list)

1. `length()` - returns the length (number of characters) in the string

   DEMO: `v.length()` `// 5`

2. `charAt(int)` - return the character at a specified position in the string.

   a) Zero origin indexing is used - so `charAt(0)` is the first character in the string

b) The parameter must lie between 0 and length - 1

DEMO:
```
v.charAt(1)          // 'e'
v.charAt(5)          // Error
```

3. `equals(String)` - returns true just when two strings represent the same sequence of characters.

   a) Note how this differs from ==, which requires that the two strings be the *same* string

   b) If `s == t` is true, then so is `s.equals(t)`; but the reverse is not necessarily the case

   DEMO: (Note previous declarations of v, u, w)
```
v == u                          // false
v.equals(u)                     // true
v == w                          // true
v.equals(w)                     // true
```

4. `equalsIgnoreCase(String)` - same as equals, but the upper case and lower case version of the same letter are considered the same

DEMO:
```
"abc".equals("ABC")             // false
"abc".equalsIgnoreCase("ABC")   // true
```

5. `compareTo(String)` - compares this string to another string character by character (from left to right), stopping when the first non-matching character is found or all the characters in one string are used up, whichever occurs first, and then returns an int value:

   a) If the return value is zero, the two strings are identical (i.e. `equals()` would be true.)

   DEMO:
```
"ABC".compareTo("ABC")          // 0
```

   b) If the return value is negative, either:

   (1) The first non-matching character in this string was less than the corresponding character in the other string
   (2) Or this string ran out of characters (it was a prefix of the other string)

   I.e. this string alphabetically preceeds the other string

DEMO:
```
"ABC".compareTo("ABD")    // -1
"ABC".compareTo("ABCD")   // -1
```

c) If the return value is positive, either:

(1) The first non-matching character in this string was greater than the corresponding character in the other string
(2) Or the other string ran out of characters (it was a prefix of this string)

I.e. this string alphabetically follows the other string

DEMO:
```
"ABC".compareTo("AB")     // 1
"ABC".compareTo("ABA")    // 2
```

Note well: the return values are <u>not</u> guaranteed to be +/- 1 !
(In fact, what the implementation does is compare characters by subtracting them, and then go until either a non-zero result occurs or one string runs out)

6. indexOf(char), indexOf(String)

a) These methods search a string for the occurrence of a certain pattern, which may be a single character or another string.

b) They return the index of beginning of the *first* match within the string - or -1 if no match is found.

DEMO:
```
"abcabc".indexOf('b')                 // 1
"I feel happy".indexOf("happy")       // 7
"Hello, world".indexOf('x')           // -1
"abc”.indexOf(‘A’)                    // -1
```
(match is case sensitive)

7. indexOf(char, int), indexOf(String, int)

These variants with an additional int parameter allow the caller to specify the place in the string where the search starts - which would otherwise be 0.

DEMO:
```
"abcabc".indexOf('b', 3)              // 4
```

8. `lastIndexOf(char)`, `lastIndexOf(char, int)`, `lastIndexOf(String)`, `lastIndexOf(String, int)`

   These find the index of the *last* occurrence of the character or string. The variants with an int parameter use this to specify a starting position for the search - since the search is done backwards, the search includes all characters from the specified one back to the start of the string.

   DEMO:

```
"abcabc".lastIndexOf('b')              // 4
"abcabc".lastIndexOf('b', 3)           // 1
```

9. `startsWith(String)`

   Returns true just when the string starts with the specified sequence of characters.

   DEMO:
```
"abcabc".startsWith("abc")     // true
"abcabc".startsWith("bc")      // false
```

10. `endsWith(String)`

    Returns true just when the string ends with the specified sequence.

    DEMO:
```
"abcabc".endsWith("abc")       // true
"abcabc".endsWith("ab")        // false
```

11. Since JDK 1.4, Java has included some much more powerful pattern-matching facilities based on the notion of a *regular expression*. The concept of regular expressions will be introduced in CS220.

12. `toUpperCase()` - returns a new string, in which all lower case letters have been changed to upper case letters, and other characters have been left alone.

    IMPORTANT: this does *not* alter the original string - it creates a new one with the changes made. **It is an accessor - not a mutator.**

    DEMO:
```
"aBcD123".toUpperCase()        // "ABCD123"
```

13. `toLowerCase()` - similar, but upper case letters are changed to lower case. (Again, an accessor, not a mutator - creates a new string without altering the original)

    DEMO:
```
"aBcD123".toLowerCase()        // "abcd123"
```

14. `trim()` - creates a new string with leading and trailing spaces removed

    DEMO: `"  abc  ".trim()  // "abc"`

15. `substring(int, int)` - creates a new string that is a portion of this string. The first parameter specifies the position of the first character to *include*; the second, the position of the first character to *exclude*. Zero origin indexing is used for both. As in the above, the original string is <u>not</u> altered.

    DEMO: `"abcdefg".substring(1, 4)  // "bcd"`

    a) The difference between the starting and ending positions is the length of the resulting string (e.g. in the above case 4-1 = 3 is the length of the result).

    b) If the starting position and ending position are equal, a string of length zero is returned.

    DEMO:
    ```
    "abcdefg".substring(1, 1)  // ""
    ```

    c) It is an error for the starting position to be less than 0, or for the ending position to be greater than the length of the string, or for the starting position to be greater than the ending position.

    DEMO:
    ```
    "abcdefg".substring(-1, 1)  // Error
    "abcdefg".substring(3, 7)   // "defg"
    "abcdefg".substring(3, 8)   // Error
    "abcdefg".substring(3, 2)   // Error
    ```

    d) Note that the substring method in Java differs from the equivalent string operation in some other programming languages where the last parameter specifies the *length* of the substring.

16. `replace(char, char)`

    Creates a new string, in which all occurrences of the first character are replaced by the second. As in the above, the original string is <u>not</u> altered.

    DEMO: `"Food".replace('o', 'e')  // “Feed”`

IV. Mutable Strings: the class StringBuffer

A. One important property of a String object is that it is immutable - i.e. the operations on it don't change its contents. The operations described above that seem to alter the content (e.g. +, toUpperCase(), toLowerCase(), trim(), substring(), replace()) don't change the string to which they are applied - they create a new string, leaving the original one unaltered.

B. Sometimes, for reasons of efficiency, it is desirable to use a mutable string - i.e. one whose content can be altered. This avoids the necessity of creating a whole new copy of the string, which can consume a fair amount of time in copying over the existing data to the new string.

C. The class StringBuffer is provided for this purpose. Its methods include the following

1. `StringBuffer(), StringBuffer(String)` - Constructors - initialize the object to empty, or to a copy of the contents of some ordinary string

   DEMO:
   ```
   StringBuffer b = new StringBuffer("Hello")
   ```

2. `toString()` - creates an ordinary string from the buffer. (The buffer itself is unchanged.

   DEMO:
   ```
   b.toString()
   ```

   Note: As is the case with any object, this is done automatically in a context where a String is needed.

   DEMO:
   ```
   "The value of b is " + b
   b    // Consequence of how Dr. Java works
   ```

3. `length(), charAt(int)` - same as for ordinary strings

   DEMO:
   ```
   b.length()
   b.charAt(1)
   ```

4. `setCharAt(int, char)` - change the character at a specified position to be a new value

   DEMO:
   ```
   b.setCharAt(0, 'J')
   b
   ```

5. `append(any type)` - add the representation for the argument to the end of the buffer. (The buffer is expanded if necessary). Contrast with + for ordinary strings, which creates a new string, rather than modifying an existing one.

   DEMO:
   ```
   b.append(" World");
   b
   ```

6. `insert(int, any type)` - insert the representation for the second argument at a specified place in the buffer. (The buffer is expanded if necessary)

   DEMO:
   ```
   b.insert(5, " and GoodBye");
   b
   ```

7. `reverse()` - reverses the order of the characters

   DEMO:
   ```
   b.reverse();
   b
   ```

V. Further Demonstrations of String Processing

A. Distribute copies of the StringsDemo program

B. *DEMO* various option - Use StringsDemo.dat for input

1. Demo search with “the”
2. Demo with “but” - note word is not found, because the only occurrence is But.
3. Repeat with case insensitive search
4. Demo substitution - change fox to raccoon - note “raccoones”
5. Demo word list

VI. Parsing Numbers

A. Sometimes, a program needs to convert the String representation of a number to internal binary form. (We have used several examples in which this is the case, but have skipped the details for now.)

EXAMPLE: Recall the examples we did with reversing the order of an array of integers.

PROJECT `EvenBetterReverse` from arrays lecture. This program makes use of a method (`JOptionPane.showInputDialog("How many integers?")`) that returns whatever the user typed as a character string. In this case, we know that the String has to represent an integer - it would be an error for the user to type anything else.

The method `Integer.parseInt()` converts the string to an int, if this is possible.

DEMO: Show what happens if something that is not a valid integer is typed.

DEMO: Using this method with various valid and invalid strings in Dr. Java

B. A similar method exists for doubles - `Double.parseDouble()`
(There are similar methods for the other numeric classes as well, but we won't discuss them)