CPS122 Lecture: Characters and Strings in Java

Last Revised January 8, 2010

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. Javadoc for class String
6. StringsDemo.java, .class, .dat + printed copy (*Appleworks version*) to hand out

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:

5

```
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";
   ```
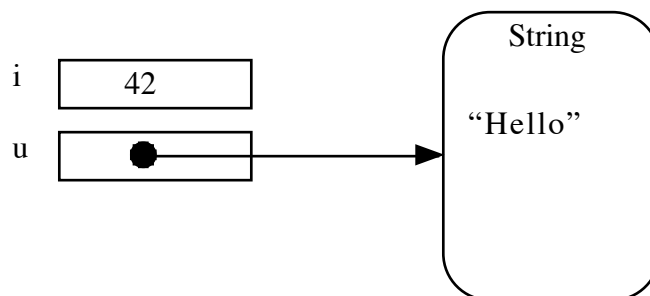

   One consequence of this is that the following comparison is *not necessarily* true (though it might be)

6

```
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;
```

v [•] ———————→ String

"Hello"

w [•] ———————→
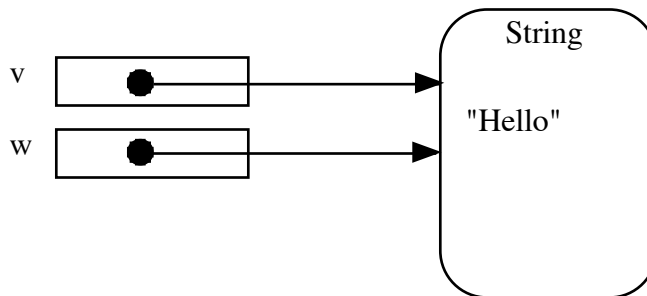
In this case, the test v == w would be guaranteed to be true.

DEMO with Dr. Java

3. To compare two Strings for equality of value, one uses the `equals()` method

DEMO with Dr. Java

```
u.equals("Hello")
```

D. Methods of class String

The class String provides a large number of methods

SHOW Javadoc for String

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

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

2. `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`

3. `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`

4. `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