Eric Roberts
CS 106A

Handout #29
April 27, 2012

# Characters and Strings

## Characters and Strings

Eric Roberts
CS 106A
April 27, 2012

## Early Character Encodings

- The idea of using codes to represent letters dates from before the time of Herman Hollerith, whose contribution is described in the introduction to Chapter 8.
- Most of you are familiar with the work of Samuel F. B. Morse, inventor of the telegraph, who devised a code consisting of dots and dashes. This scheme made it easier to transmit messages and paved the way for later developments in communication.

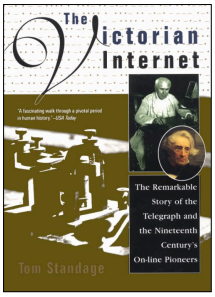
Samuel Morse (1791-1872)

| | | | | | |
|---|---|---|---|---|---|
| A | ·− | J | ·−−− | S | ··· |
| B | −··· | K | −·− | T | − |
| C | −·−· | L | ·−·· | U | ··− |
| D | −·· | M | −− | V | ···− |
| E | · | N | −· | W | ·−− |
| F | ··−· | O | −−− | X | −··− |
| G | −−· | P | ·−−· | Y | −·−− |
| H | ···· | Q | −−·− | Z | −−·· |
| I | ·· | R | ·−· | | |

Alphabetic Characters in Morse Code

## The Victorian Internet

What you probably don't know is that the invention of the telegraph also gave rise to many of the social phenomena we tend to associate with the modern Internet, including chat rooms, online romances, hackers, and entrepreneurs—all of which are described in Tom Standage's 1998 book, *The Victorian Internet.*




## The Principle of Enumeration

- Computers tend to be good at working with numeric data. When you declare a variable of type `int`, for example, the Java virtual machine reserves a location in memory designed to hold an integer in the defined range.
- The ability to represent an integer value, however, also makes it easy to work with other data types as long as it is possible to represent those types using integers. For types consisting of a finite set of values, the easiest approach is simply to number the elements of the collection.
- For example, if you want to work with data representing months of the year, you can simply assign integer codes to the names of each month, much as we do ourselves. Thus, January is month 1, February is month 2, and so on.
- Types that are identified by counting off the elements are called ***enumerated types***.

## Enumerated Types in Java

- Java offers two strategies for representing enumerated types:
  - Defining named constants to represent the values in the enumeration
  - Using the `enum` facility introduced in Java 5.0
- Although I cover the `enum` syntax briefly in the book, I remain convinced that it is easier for beginning programmers to use the older strategy of defining integer constants to represent the elements of the type and then using variables of type `int` to store the values.
- For example, you can define names for the major compass points as follows:

```
public static final int NORTH = 0;
public static final int EAST = 1;
public static final int SOUTH = 2;
public static final int WEST = 3;
```

## Characters

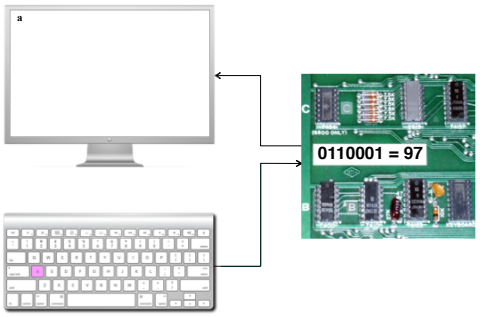
- Computers use the principle of enumeration to represent character data inside the memory of the machine. There are, after all, a finite number of characters on the keyboard. If you assign an integer to each character, you can use that integer as a code for the character it represents.
- Character codes, however, are not particularly useful unless they are standardized. If different computer manufacturers use different coding sequences (as was indeed the case in the early years), it is harder to share such data across machines.
- The first widely adopted character encoding was ASCII (*American Standard Code for Information Interchange*).
- With only 256 possible characters, the ASCII system proved inadequate to represent the many alphabets in use throughout the world. It has therefore been superseded by Unicode, which allows for a much larger number of characters.

## The ASCII Subset of Unicode

The following table shows the first 128 characters in the Unicode character set, which are the same as in the older ASCII scheme:

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 00x | \000 | \001 | \002 | \003 | \004 | \005 | \006 | \007 |
| 01x | \b | \t | \n | \011 | \f | \r | \016 | \017 |
| 02x | \020 | \021 | \022 | \023 | \024 | \025 | \026 | \027 |
| 03x | \030 | \031 | \032 | \033 | \034 | \035 | \036 | \037 |
| 04x | *space* | ! | " | # | $ | % | & | ' |
| 05x | ( | ) | * | + | , | - | . | / |
| 06x | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| 07x | 8 | 9 | : | ; | < | = | > | ? |
| 10x | @ | A | B | C | D | E | F | G |
| 11x | H | I | J | K | L | M | N | O |
| 12x | P | Q | R | S | T | U | V | W |
| 13x | X | Y | Z | [ | \ | ] | ^ | _ |
| 14x | ` | a | b | c | d | e | f | g |
| 15x | h | i | j | k | l | m | n | o |
| 16x | p | q | r | s | t | u | v | w |
| 17x | x | y | z | { | \| | } | ~ | \177 |

## Hardware Support for Characters




## Notes on Character Representation

- The first thing to remember about the Unicode table from the previous slide is that you don't actually have to learn the numeric codes for the characters. The important observation is that a character *has* a numeric representation, and not what that representation happens to be.
- To specify a character in a Java program, you need to use a character constant, which consists of the desired character enclosed in single quotation marks. Thus, the constant `'A'` in a program indicates the Unicode representation for an uppercase A. That it has the value $101_8$ is an irrelevant detail.
- Two properties of the Unicode table are worth special notice:
  - The character codes for the digits are consecutive.
  - The letters in the alphabet are divided into two ranges, one for the uppercase letters and one for the lowercase letters. Within each range, the Unicode values are consecutive.

## Special Characters

- Most of the characters in the Unicode table are the familiar ones that appear on the keyboard. These characters are called ***printing characters***. The table also includes several ***special characters*** that are typically used to control formatting.
- Special characters are indicated in the Unicode table by an ***escape sequence***, which consists of a backslash followed by a character of sequence of digits. The most common ones are:

| | |
|---|---|
| `\b` | Backspace |
| `\f` | Form feed (starts a new page) |
| `\n` | Newline (moves to the next line) |
| `\r` | Return (moves to the beginning of the current line without advancing) |
| `\t` | Tab (moves horizontally to the next tab stop) |
| `\\` | The backspace character itself |
| `\'` | The character ' (required only in character constants) |
| `\"` | The character " (required only in string constants) |
| *ddd* | The character whose Unicode value is the octal number *ddd* |

## Useful Methods in the `Character` Class

| |
|---|
| **`static boolean isDigit(char ch)`**<br>Determines if the specified character is a digit. |
| **`static boolean isLetter(char ch)`**<br>Determines if the specified character is a letter. |
| **`static boolean isLetterOrDigit(char ch)`**<br>Determines if the specified character is a letter or a digit. |
| **`static boolean isLowerCase(char ch)`**<br>Determines if the specified character is a lowercase letter. |
| **`static boolean isUpperCase(char ch)`**<br>Determines if the specified character is an uppercase letter. |
| **`static boolean isWhitespace(char ch)`**<br>Determines if the specified character is **whitespace** (spaces and tabs). |
| **`static char toLowerCase(char ch)`**<br>Converts `ch` to its lowercase equivalent, if any. If not, `ch` is returned unchanged. |
| **`static char toUpperCase(char ch)`**<br>Converts `ch` to its uppercase equivalent, if any. If not, `ch` is returned unchanged. |

## Character Arithmetic

- The fact that characters have underlying representations as integers allows you can use them in arithmetic expressions. For example, if you evaluate the expression `'A' + 1`, Java will convert the character `'A'` into the integer 65 and then add 1 to get 66, which is the character code for `'B'`.
- As an example, the following method returns a randomly chosen uppercase letter:

```
public char randomLetter() {
   return (char) rgen.nextInt('A', 'Z');
}
```

- The following code implements the `isDigit` method from the `Character` class:

```
public boolean isDigit(char ch) {
   return (ch >= '0' && ch <= '9');
}
```

## Exercise: Character Arithmetic

- Implement a method `toHexDigit` that takes an integer and returns the corresponding hexadecimal digit as a character. Thus, if the argument is between 0 and 9, the method should return the corresponding character between `'0'` and `'9'`. If the argument is between 10 and 15, the method should return the appropriate letter in the range `'A'` through `'F'`. If the argument is outside this range, the method should return `'?'`.

## Strings as an Abstract Idea

- Ever since the very first program in the text, which displayed the message `"hello, world"` on the screen, you have been using strings to communicate with the user.
- Up to now, you have not had any idea how Java represents strings inside the computer or how you might manipulate the characters that make up a string. At the same time, the fact that you don't know those things has not compromised your ability to use strings effectively because you have been able to think of strings holistically as if they were a primitive type.
- For most applications, the abstract view of strings you have held up to now is precisely the right one. On the inside, strings are surprisingly complicated objects whose details are better left hidden.
- Java supports a high-level view of strings by making `String` a class whose methods hide the underlying complexity.

## Using Methods in the `String` Class

- Java defines many useful methods that operate on the `String` class. Before trying to use those methods individually, it is important to understand how those methods work at a more general level.
- The `String` class uses the receiver syntax when you call a method on a string. Instead of calling a static method (as you do, for example, with the `Character` class), Java's model is that you send a message to a string.
- None of the methods in Java's `String` class change the value of the string used as the receiver. What happens instead is that these methods ***return*** a new string on which the desired changes have been performed.
- Classes that prohibit clients from changing an object's state are said to be ***immutable***. Immutable classes have many advantages and play an important role in programming.

## Strings *vs.* Characters

- The differences in the conceptual model between strings and characters are easy to illustrate by example. Both the `String` and the `Character` class export a `toUpperCase` method that converts lowercase letters to their uppercase equivalents.
- In the `Character` class, you call `toUpperCase` as a static method, like this:

```
ch = Character.toUpperCase(ch);
```

- In the `String` class, you apply `toUpperCase` to an existing string, as follows:

```
str = str.toUpperCase();
```

- Note that both classes require you to assign the result back to the original variable if you want to change its value.

## Selecting Characters from a String

- Conceptually, a string is an ordered collection of characters.
- In Java, the character positions in a string are identified by an ***index*** that begins at 0 and extends up to one less than the length of the string. For example, the characters in the string `"hello, world"` are arranged like this:

| h | e | l | l | o | , |  | w | o | r | l | d |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |

- You can obtain the number of characters by calling `length`.
- You can select an individual character by calling `charAt(`*k*`)`, where *k* is the index of the desired character. The expression

```
str.charAt(0);
```

returns the `'h'` that appears at index position 0 in `str`.

## Concatenation

- One of the most useful operations available for strings is ***concatenation***, which consists of combining two strings end to end with no intervening characters.
- The `String` class exports a method called `concat` to signify concatenation, although that method is hardly ever used. Concatenation is built into Java in the form of the `+` operator.
- If you use `+` with numeric operands, it signifies addition. If at least one of its operands is a string, Java interprets `+` as concatenation. When it is used in this way, Java performs the following steps:
  - If one of the operands is not a string, convert it to a string by applying the `toString` method for that class.
  - Apply the `concat` method to concatenate the values.

## Extracting Substrings

- The `substring` method makes it possible to extract a piece of a larger string by providing index numbers that determine the extent of the substring.
- The general form of the `substring` call is

```
str.substring(p1, p2);
```

  where `p1` is the first index position in the desired substring and `p2` is the index position immediately following the last position in the substring.
- As an example, if you wanted to select the substring `"ell"` from a string variable `str` containing `"hello, world"` you would make the following call:

```
str.substring(1, 4);
```

## Checking Strings for Equality

- Many applications will require you to test whether two strings are ***equal***, in the sense that they contain the same characters.
- Although it seems natural to do so, you cannot use the `==` operator for this purpose. While it is legal to write

```
if (s1 == s2) . . .
```

  the `if` test will not have the desired effect. When you use `==` on two objects, it checks whether the objects are ***identical***, which means that the references point to the same address.
- What you need to do instead is call the `equals` method:

```
if (s1.equals(s2)) . . .
```

## Comparing Characters and Strings

- The fact that characters are primitive types with a numeric internal form allows you to compare them using the relational operators. If `c1` and `c2` are characters, the expression

```
c1 < c2
```

  is `true` if the Unicode value of `c1` is less than that of `c2`.
- The `String` class allows you to compare two strings using the internal values of the characters, although you must use the `compareTo` method instead of the relational operators:

```
s1.compareTo(s2)
```

  This call returns an integer that is less than 0 if `s1` is less than `s2`, greater than 0 if `s1` is greater than `s2`, and 0 if the two strings are equal.

## Searching in a String

- Java's `String` class includes several methods for searching within a string for a particular character or substring.
- The method `indexOf` takes either a string or a character and returns the index within the receiving string at which the first instance of that value begins. If the string or character does not exist at all, `indexOf` returns –1. For example, if the variable `str` contains the string `"hello, world"`:

| | | |
|---|---|---|
| `str.indexOf('h')` | *returns* | 0 |
| `str.indexOf("o")` | *returns* | 4 |
| `str.indexOf("ell")` | *returns* | 1 |
| `str.indexOf('x')` | *returns* | –1 |

- The `indexOf` method takes an optional second argument that indicates the starting position for the search. Thus:

| | | |
|---|---|---|
| `str.indexOf("o", 5)` | *returns* | 8 |

## Other Methods in the `String` Class

| |
|---|
| `int lastIndexOf(char ch)` *or* `lastIndexOf(String str)`<br>Returns the index of the last match of the argument, or –1 if none exists. |
| `boolean equalsIgnoreCase(String str)`<br>Returns `true` if this string and `str` are the same, ignoring differences in case. |
| `boolean startsWith(String str)`<br>Returns `true` if this string starts with `str`. |
| `boolean endsWith(String str)`<br>Returns `true` if this string starts with `str`. |
| `String replace(char c1, char c2)`<br>Returns a copy of this string with all instances of `c1` replaced by `c2`. |
| `String trim()`<br>Returns a copy of this string with leading and trailing whitespace removed. |
| `String toLowerCase()`<br>Returns a copy of this string with all uppercase characters changed to lowercase. |
| `String toUpperCase()`<br>Returns a copy of this string with all lowercase characters changed to uppercase |

## Simple String Idioms

When you work with strings, there are two idiomatic patterns that are particularly important:

1. Iterating through the characters in a string.

```
for (int i = 0; i < str.length(); i++) {
   char ch = str.charAt(i);
   . . . code to process each character in turn . . .
}
```

2. Growing a new string character by character.

```
String result = "";
for (whatever limits are appropriate to the application) {
   . . . code to determine the next character to be added . . .
   result += ch;
}
```

## Exercises: String Processing

- As a client of the `String` class, how would you implement `toUpperCase(str)` so it returns an uppercase copy of `str`?

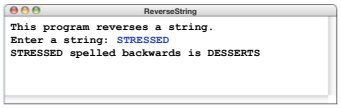
- Suppose instead that you are implementing the `String` class. How would you code the method `indexOf(ch)`?

## The `reverseString` Method

```
public void run() {
```

```
private String reverseString(String str) {
   String result = "";
   for ( int i = 0; i < str.length(); i++ ) {
      result = str.charAt(i) + result;
   }
   return result;
}
```

| result | str | i |
|---|---|---|
| DESSERTS | STRESSED | 8 |




## Exercise: Checking Palindromes

A ***palindrome*** is a string that reads the same forward and backward, such as `"LEVEL"` or `"NOON"`. How would you implement the predicate function `isPalindrome(str)`?