Appendix D Characters & Strings

Characters

Characters are represented in Java by variables of type *char*. Characters are a kind of integer with numeric values in the range 0…65535, but their values are not displayed in the way the values of integers are. Instead, there is a standard code, called unicode, that determines how each numeric value is displayed.

There are 94 visible characters that may be typed on a standard keyboard.

plain	shifted
`` `1234567890-= ``	`~!@#$%^&*()_+`
`qwertyuiop[]\`	`QWERTYUIOP{}|`
`asdfghjkl;'`	`ASDFGHJKL:"`
`zxcvbnm,./`	`ZXCVBNM<>?`

These 94 characters are stored using the values 33…126. Some examples appear here.

65 → A	66 → B	67 → C	....	89 → Y	90 → Z	upper case letters
97 → a	98 → b	99 → c	....	121 → y	122 → z	lower case letters
48 → 0	49 → 1	50 → 2	....	56 → 8	57 → 9	digits
33 → !	35 → #	63 → ?	91 → [	125 → }	126 → ~	miscellaneous examples

In addition, a blank space is stored using the value 32.

You can put the values of the blank space and the visible keyboard characters into *char* variables in two ways. Each of the following pairs is equivalent.

```
char cc = 'A';        char dd = 'c';         char ee = ' ';
char cc = 65;         char dd = 99;          char ee = 32;
```

There are two characters that must be preceded by a back slash when entered in non-numeric format:

```
 a single quote         a back slash
char ff = '\'';       char gg = '\\';
char ff = 39;         char gg = 92;
```

Generally, the numeric format is used for characters that cannot be typed on a standard keyboard. For example, the Greek letter π has the value 960 and you must assign a numeric value to use this value.

```
char pi = 960;
char pi = '\u03c0';
```

Many code tables give the numeric codes for unicode characters as hexadecimal numerals. The hexadecimal numeral for the decimal 960 is 3c0. When specifying the hexadecimal code for a character in Java, you start with \u and follow this with four digits. In this case a 0 was placed before 3c0 to make up the required four digits.

The *char* variable *pi* shown here will display as π on almost all modern computers. While unicode specifies code values for just about every language (Arabic, Burmese, Chinese, English, Hebrew …), I'm not sure just what percent of modern computers will display each of these alphabets.

Several non-printing characters may be entered with special symbols:

```
char f = '\n';  (the newline character = 10)
char g = '\e';  (the escape character = 27)
char h = '\t';  (the tab character = 9)
```

Strings

- A String can be entered by placing characters between double quotation marks:

  ```
  String a = "Hello";
  String b = "He said \"Drop that gun!\"";
  String c = "C = 2\u03c0r";
  String d = "";
  ```

  The string *a* contains five letters •Hello•.
  The string *b* contains •He said "Drop that gun!"•
  The string *c* contains •C = 2πr•
  The string *d*, often called an empty string, contains no characters at all.

- A String can be constructed by concatenation. Concatenation is the construction of a longer string by putting two strings together. An addition sign '+' becomes a concatenation operator when either (or both) of the items to be 'added' is a string. If one of the two objects is not a string, a string is constructed from it, and the two strings are then concatenated.

`d = "Hello" + "Ann";`	*d* contains •HelloAnn •
`e = "Hello" + " " + "Ann";`	*e* contains •Hello Ann •
`f = "Ann is " + 3 + " years old";`	*f* contains • Ann is 3 years old•
`g = "Ann is " + 3 + 5 + " years old";`	*g* contains • Ann is 35 years old•
`h = "Ann is " + (3 + 5) + " years old";`	*h* contains • Ann is 8 years old•

  In Java, concatenation precedes from left to right so that example *g* is built up as:

  ```
  g = "Ann is " + 3 + 5 + " years old"
  g = "Ann is 3" + 5 + " years old"
  g = "Ann is 35" + " years old"
  g = "Ann is 35 years old"
  ```

  In example *h*, the parentheses force the 3 and 5 to be added (giving 8) before any strings are encountered:

  ```
  h = "Ann is " + (3 + 5) + " years old"
  h = "Ann is " + 8 + " years old"
  h = "Ann is 8" + " years old"
  h = "Ann is 8 years old"
  ```

- An individual character can be obtained from a string by using the *charAt* method. The characters in a string are numbered beginning with 0.

  ```
  a = "Hello";
  ```

  `a.charAt(0)` is equal to 'H';
  `a.charAt(1)` is equal to 'e';
  `a.charAt(4)` is equal to 'o';
  `a.charAt(5)` is an error;

- The length of a string can be obtained by the length method.

  ```
  a = "Hello";
  ```

  `a.length()` is equal to 5;

  For the purposes of using *charAt* the characters of a string lie in the range 0…length–1.

The example below creates a string containing the characters in the string *s* except for digits (0…9).

```
String s = "The 5 elephants and 17 monkeys ran for 8 hours.";
String noDigits = "";
for (int n = 0; n < s.length() < s++)
   if (s.charAt(n) < '0' || '9' < s.charAt(n))
      noDigits = noDigits + s.charAt(n);
```

noDigits contains "`The  elephants and  monkeys ran for  hours.`"
There are 2 spaces before *elephants*, *monkeys* and *hours* because the spaces before and after each number remain.

- A portion of a string can be obtained by using the *substring* method. The positions between the characters in a string are numbered as shown. It is an error to use a position outside of the range 0…length of string.

```
a = "Hello";
positions:   H   e   l   l   o
           ^   ^   ^   ^   ^   ^
           0   1   2   3   4   5
```

`a.substring(1, 3)`	is equal to	`"el"`	(between positions 1 and 3)
`a.substring(2, 2)`	is equal to	`""`	(empty string—between 2 and 2)
`a.substring(2)`	is equal to	`"llo"`	(from 2 to the end)
`a.substring(0, a.length())`	is equal to	`"Hello"`	(between 0 and 5)

- These is no way to change a character in a string. When this is desired, one creates a new string from the old one with the undesired letter omitted and a new one inserted. For example, to create "George Smith" from "George smith":

```
String s = "George smith";
String t = s.substring(0,7) + 'S' + s.substring(8);
```

- Strings should not be compared with the == or != operators. Java will let you use == and != with strings; this is almost never the right thing. These operators do not examine the letters in the string.

  The right way to compare two strings (s and t) for equality is:

```
if (s.equals(t))
   ....
```

  The right way to compare two strings (s and t) for inequality is:

```
if (!s.equals(t))
   ....
```

  More commonly, it is better to use *equalsIgnoreCase* rather than equals.

```
if (s.equalsIgnoreCase(t))
   ....
if (!s.equalsIgnoreCase(t))
   ....
```

- Strings can not be compared with the <, <=, > or >= operators.

  The right way to ask if *s* is lexicographically less than *t* is:
  ```
  if (s.compareTo(t) < 0)
      ....
  ```
  The right way to ask if *s* is lexicographically less than or equal to *t* is:
  ```
  if (s.compareTo(t) <= 0)
      ....
  ```
  The right way to ask if *s* is lexicographically greater than *t* is:
  ```
  if (s.compareTo(t) > 0)
      ....
  ```
  The right way to ask if *s* is lexicographically greater than or equal to *t* is:
  ```
  if (s.compareTo(t) >= 0)
      ....
  ```
  There is a similar method *compareToIgnoreCase* available in all but very old versions of Java.
  ```
  if (s.compareToIgnoreCase(t) < 0)
      ....
  ```

Partial List of String Methods

`public int length()`

Returns the length of this string.

`public int charAt(int index)`

Parameters:

the index of the character.

Returns:

the character at the specified index of this string. The first character is at index 0.

Throws:

StringIndexOutOfBoundsException if the index is out of range.

`public boolean equals(String anotherString)`

Parameters:

the string to be compared.

Returns:

true if the strings are equal; *false* otherwise.

See Also:

compareTo, equalsIgnoreCase

`public boolean equalsIgnoreCase(String anotherString)`

Parameters:

the string to be compared.

Returns:

true if the strings are equal, ignoring case; *false* otherwise.

See Also:

toLowerCase, toUpperCase

`public int compareTo(String anotherString)`

Parameters:

the string to be compared.

Returns:

the value 0 if this string is equal to the argument string; a value less than 0 if this string is lexicographically less than the string argument; and a value greater than 0 if this string is lexicographically greater than the string argument.

`public int compareToIgnoreCase(String anotherString)`

Parameters:

anotherString - the String to be compared.

Returns:

Similar to *compareTo*.

`public String trim()`

Returns:

a string derived from this string by removing 'white space', (includes spaces and tabs), from both ends.

`public boolean startsWith(String prefix)`

Parameters:

the prefix.

Returns:

true if the character sequence represented by the argument is a prefix of the character sequence represented by this string; *false* otherwise.

`public boolean endsWith(String suffix)`

Parameters:

the suffix.

Returns:

true if the character sequence represented by the argument is a suffix of the character sequence represented by this string; *false* otherwise.

`public int indexOf(char ch)`

Parameters:

a character.

Returns:

the index of the first occurrence of the character in the character sequence represented by this string, or -1 if the character does not occur.

`public int lastIndexOf(char ch)`

Parameters:

a character.

Returns:

the index of the last occurrence of the character in the character sequence represented by this string, or -1 if the character does not occur.

`public int` `indexOf(String str)`

Parameters:

the substring to search for.

Returns:

if the string argument occurs as a substring within this object, then the index of the first character of the first such substring is returned; if it does not occur as a substring, -1 is returned.

`public int` `lastIndexOf(String str)`

Parameters:

the substring to search for.

Returns:

if the string argument occurs one or more times as a substring within this object, then the index of the first character of the last such substring is returned. If it does not occur as a substring, -1 is returned.

`public` `String substring(`**`int`** `beginIndex)`

Parameters:

the beginning index, inclusive.

Returns:

a string containing the specified portion of this string.

Throws:

StringIndexOutOfBoundsException if the *beginIndex* is out of range.

`public` `String substring(`**`int`** `beginIndex,` **`int`** `endIndex)`

Parameters:

beginIndex - the beginning index, inclusive.

endIndex - the ending index, exclusive.

Returns:

a string containing the specified portion of this string.

Throws:

StringIndexOutOfBoundsException if the *beginIndex* or the *endIndex* is out of range.

`public` `String replace(`**`char`** `oldChar,` **`char`** `newChar)`

Parameters:

oldChar - the old character.

newChar - the new character.

Returns:

a string derived from this string by replacing every occurrence of *oldChar* with *newChar*.

`public` `String toLowerCase()`

Returns:

a string derived from this string by converting characters of this string to lowercase.

`public` `String toUpperCase()`

Returns:

a string derived from this string by converting characters of this string to uppercase.