CHARACTERS & STRINGS & FILES

CITS1001
Outline

• On computers Characters are represented by a standard code: either ASCII or Unicode
• String is one of the classes of the standard Java library
• The String class represents character strings such as “This is a String!”
• Strings are constant (immutable) objects
• StringBuilder is used for changeable (mutable) strings
• Use the right library for your Strings - it makes a difference!

• Reference: Objects First, Ch 5
• This lecture is based on powerpoints by Gordon Royle UWA
In the beginning there was ASCII

- Internally every data item in a computer is represented simply by a bit-pattern
- To store integers this is not a problem, because we can simply store their binary representation
- However for non-numerical data such as characters and text we need some sort of encoding that assigns a number (really a bit-pattern) to each character
- In 1968, the American National Standards Institute announced a code called ASCII - the American Standard Code for Information Interchange
  - This was actually an updated version of an earlier code
ASCII

• ASCII specified numerical codes for 96 printing characters and 32 “control characters” making a total of 128 codes
• The upper-case alphabetic characters ‘A’ to ‘Z’ were assigned the numerical codes from 65 onwards

<table>
<thead>
<tr>
<th></th>
<th>65</th>
<th>66</th>
<th>67</th>
<th>68</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
</tr>
<tr>
<td>B</td>
<td>E</td>
<td>F</td>
<td>G</td>
<td>H</td>
</tr>
<tr>
<td>C</td>
<td>I</td>
<td>J</td>
<td>K</td>
<td>L</td>
</tr>
<tr>
<td>D</td>
<td>I</td>
<td>J</td>
<td>K</td>
<td>L</td>
</tr>
<tr>
<td>E</td>
<td>M</td>
<td>N</td>
<td>O</td>
<td>P</td>
</tr>
<tr>
<td>F</td>
<td>M</td>
<td>N</td>
<td>O</td>
<td>P</td>
</tr>
<tr>
<td>G</td>
<td>Q</td>
<td>R</td>
<td>S</td>
<td>T</td>
</tr>
<tr>
<td>H</td>
<td>Q</td>
<td>R</td>
<td>S</td>
<td>T</td>
</tr>
<tr>
<td>I</td>
<td>U</td>
<td>V</td>
<td>W</td>
<td>X</td>
</tr>
<tr>
<td>J</td>
<td>U</td>
<td>V</td>
<td>W</td>
<td>X</td>
</tr>
<tr>
<td>K</td>
<td>Y</td>
<td>Z</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L</td>
<td>Y</td>
<td>Z</td>
<td></td>
<td></td>
</tr>
<tr>
<td>M</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>O</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>P</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Q</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>U</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>V</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>W</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Y</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Z</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
ASCII cont

- The lower-case alphabetic characters ‘a’ to ‘z’ were assigned the numerical codes from 97 onwards

<table>
<thead>
<tr>
<th>lower-case</th>
<th>code</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>97</td>
</tr>
<tr>
<td>b</td>
<td>98</td>
</tr>
<tr>
<td>c</td>
<td>99</td>
</tr>
<tr>
<td>d</td>
<td>100</td>
</tr>
<tr>
<td>e</td>
<td>101</td>
</tr>
<tr>
<td>f</td>
<td>102</td>
</tr>
<tr>
<td>g</td>
<td>103</td>
</tr>
<tr>
<td>h</td>
<td>104</td>
</tr>
<tr>
<td>i</td>
<td>105</td>
</tr>
<tr>
<td>j</td>
<td>106</td>
</tr>
<tr>
<td>k</td>
<td>107</td>
</tr>
<tr>
<td>l</td>
<td>108</td>
</tr>
<tr>
<td>m</td>
<td>109</td>
</tr>
<tr>
<td>n</td>
<td>110</td>
</tr>
<tr>
<td>o</td>
<td>111</td>
</tr>
<tr>
<td>p</td>
<td>112</td>
</tr>
<tr>
<td>q</td>
<td>113</td>
</tr>
<tr>
<td>r</td>
<td>114</td>
</tr>
<tr>
<td>s</td>
<td>115</td>
</tr>
<tr>
<td>t</td>
<td>116</td>
</tr>
<tr>
<td>u</td>
<td>117</td>
</tr>
<tr>
<td>v</td>
<td>118</td>
</tr>
<tr>
<td>w</td>
<td>119</td>
</tr>
<tr>
<td>x</td>
<td>120</td>
</tr>
<tr>
<td>y</td>
<td>121</td>
</tr>
<tr>
<td>z</td>
<td>122</td>
</tr>
</tbody>
</table>
• Other useful printing characters were assigned a variety of codes, for example the range 58 to 64 was used as follows

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>:</td>
<td>58</td>
<td>;</td>
<td>59</td>
<td>&lt;</td>
<td>60</td>
</tr>
<tr>
<td>&gt;</td>
<td>62</td>
<td>?</td>
<td>63</td>
<td>@</td>
<td>64</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>A</td>
<td>65</td>
</tr>
</tbody>
</table>

• As computers became more ubiquitous, the need for additional characters became apparent and ASCII was extended in various different ways to 256 characters
• However any 8-bit code simply cannot cope with many characters from non-English languages
Unicode

• Unicode is an international code that specifies numerical values for characters from almost every known language, including alphabets such as Braille
• Java’s char type uses 2 bytes to store these Unicode values
• For the convenience of pre-existing computer programs, Unicode adopted the same codes as ASCII for the characters covered by ASCII
To characters and back

- To find out the code assigned to a character in Java we can simply cast the character to an int.

- Conversely we can cast an integer back to a char to find out what character is represented by a certain value.
Character Arithmetic

- Using the codes we can do character “arithmetic”
- For example, it is quite legitimate to increment a character variable as in the following code

```c
char ch;
ch = 'A';
ch++;`n

- Now `ch` has the value ‘B’
Characters as numbers

• As characters are treated internally as numbers, this means they can be freely used in this way.
• A loop involving characters
  ```
  for (char ch = 'a'; ch <= 'z'; ch++) {
    // ch takes the values 'a' through 'z' in turn
  }
  ```
• You can use characters in a `switch` statement
  ```
  switch (ch) {
    case 'N':  // move north
    case 'E':  // move east
    case 'W':  // move west
    case 'S':  // move south
  }
  ```
Unicode notation

- Unicode characters are conventionally expressed in the form $U+\text{dddd}$

- Here $\text{dddd}$ is a 4-digit hexadecimal number which is the code for that character

- We have already seen that ‘A’ is represented by the code 65, which is 41 in hexadecimal

- So the official Unicode code for ‘A’ is $U+0041$
Unicode characters in Java

• Java has a special syntax to allow you to directly create characters from their U-numbers

    char ch;
    ch = '\u0041';

• You can of course do this in BlueJ’s code pad
### More interesting characters

<table>
<thead>
<tr>
<th>2600</th>
<th>Miscel</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Characters" /></td>
<td></td>
</tr>
</tbody>
</table>

See [www.unicode.org](http://www.unicode.org) for these code charts
Strings

• A *string* is a sequence of (Unicode) characters
  ABCDEFGHIJ
  Hello, my name is Hal

• One of the major uses of computers is the manipulation and processing of text and so string operations are extremely important

• Java provides support for strings through two classes in the fundamental *java.lang* package: *String* and *StringBuilder*

• Use *StringBuffer* only for multi-threaded applications
String literals

- You can create a String literal just by listing its characters between quotes

```java
String s = "Hello";
String s = "\u2600\u2601\u2602"
```
java.lang.String

- The class `String` is used to represent *immutable* strings
  - Immutable means that a `String` object cannot be *altered* after it has been created
- In many other languages a string actually *IS* just an array of characters, and so it is quite legal to change a single character with commands like
  ```
  s[23] = 'z'
  ```
- There are a variety of reasons for having `Strings` being immutable including certain aspects of efficiency and security
Methods in the String class

- The `String` class provides a wide variety of methods for creating and using strings
- Two basic methods are

  ```java
  public int length()
  • this returns the number of characters in the String
  public char charAt(int index)
  • This returns the character at the given index, where as usual the indexing starts at 0
  ```
Processing a String

• These two methods give us the fundamental mechanism for inspecting each character of a String in turn

```java
public void inspectString(String s) {
    int len = s.length();
    for (int i=0; i<len; i++) {
        char ch = s.charAt(i);
        // Do something with ch
    }
}
```
Counting vowels

```java
public int countVowels(String s) {
    int numVowels = 0;
    for (int i=0; i<s.length(); i++) {
        char ch = s.charAt(i);
        if (ch == 'a' || ch == 'A')
            numVowels++;
        if (ch == 'e' || ch == 'E')
            numVowels++;
        if (ch == 'i' || ch == 'I')
            numVowels++;
        if (ch == 'o' || ch == 'O')
            numVowels++;
        if (ch == 'u' || ch == 'U')
            numVowels++;
    }
    return numVowels;
}
```
**String comparison**

**compareTo**

```java
public int compareTo(String anotherString)
```

Compares two strings lexicographically. The comparison is based on the Unicode value of each character in the strings. The character sequence represented by this `String` object is compared lexicographically to the character sequence represented by the argument string. The result is a negative integer if this `String` object lexicographically precedes the argument string. The result is a positive integer if this `String` object lexicographically follows the argument string. The result is zero if the strings are equal; `compareTo` returns 0 exactly when the `equals(Object)` method would return `true`.

This is the definition of lexicographic ordering. If two strings are different, then either they have different characters at some index that is a valid index for both strings, or their lengths are different, or both. If they have different characters at one or more index positions, let `k` be the smallest such index; then the string whose character at position `k` has the smaller value, as determined by using the `<` operator, lexicographically precedes the other string. In this case, `compareTo` returns the difference of the two character values at position `k` in the two string -- that is, the value:

```java
this.charAt(k) - anotherString.charAt(k)
```

If there is no index position at which they differ, then the shorter string lexicographically precedes the longer string. In this case, `compareTo` returns the difference of the lengths of the strings -- that is, the value:

```java
this.length() - anotherString.length()
```
Lexicographic ordering

• Lexicographic ordering is like alphabetic ordering
• First we order the alphabet
  \[ a, b, c, d, e, f, \ldots, z \]
• The following words are alphabetically ordered
  aardvark, apple, applet, ban, band
• What are the rules for alphabetic ordering of two words?
  • Find the \textit{first character} where the two words are different and use that character to order the words e.g. \texttt{aardvark} before \texttt{apple}
  • If there are no such characters, then use the \textit{length} of the words to order them e.g \texttt{ban} before \texttt{band}
compareTo

- It is the *Unicode value* of the characters that determines their ordering, so for example
  \[ \text{Xylophone} \text{ comes before apple} \]
- The method just specifies that it returns either a negative number, 0, or a positive number as follows:
  - A negative number if the target occurs *before* the argument
  - A positive number if the target occurs *after* the argument
  - Zero if the target is equal to the argument
Other methods

• To convert a `String` to lower case:
  
  ```java
  public String toLowerCase()
  ```

• Hey, I thought `Strings` were immutable! How can you change it to lower case?
  
  • I haven’t because this creates a NEW `String` that is a lower-case version of the old one
  
  • This duplication of `Strings` can be very memory-intensive
Many other methods

public int indexOf(char ch)
public int indexOf(String s)
  • Find the first occurrence in the target string of the character ch, or substring s and return their location

public String replace(char oldChar, char newChar)
  • Create a new String by replacing all occurrences of oldChar with newChar

public char[] toCharArray()
  • Retrieve the characters in the String as an array of chars
Concatenation

- We have already seen that the + operator can be used to concatenate strings
  
  ```java
  String s1 = "Hello";
  String s2 = " there";
  String s = s1 + s2;
  ```

- The immutability of Strings can have serious consequences for memory usage that may catch out the unaware - suppose for example that we had to create a single String consisting of all the words in a book
**Slow code**

```java
public String concatenate(String[] words) {
    String text = words[0];
    for (int i=1; i<words.length; i++) {
        text = text + " " + words[i];
    }
    return text;
}
```

This code is disastrously slow if the number of words is even moderately large (a few thousand) because every single time through the loop creates an entirely new `String` with just one word added, hence a vast amount of copying is done and a large number of orphaned objects are created.
Mutable Strings

- The class `StringBuilder` is used to represent strings that can be efficiently altered.
- Internally a `StringBuilder` is (essentially) an array of characters.
- It provides efficient ways to `append` and `insert` with a whole range of methods of the following form:

  ```java
  public StringBuilder append(String s)
  public StringBuilder insert(String s, int offset)
  ```

- `StringBuilder` is a single-threaded, non-synchronised class.
- Instances of `StringBuilder` are not safe for use by multiple threads. If synchronisation is required then use `StringBuffer` instead.
Appending

```java
public StringBuilder append(String s)
    • Appends the String s to the end of the target StringBuilder
    • It then returns a reference to the newly altered StringBuilder
    • Notice that the method both alters the target object and also returns a reference to it

StringBuilder s1 = new StringBuilder("Hello");
s1.append(" there");
```
Using a `StringBuilder` to concatenate

```java
public String concatenate(String[] words) {

    StringBuilder text = new StringBuilder(words[0]);

    for (int i=1; i<words.length; i++) {
        text.append(" ");
        text.append(words[i]);
    }

    return new String(text);
}
```
How much difference does it make?

<table>
<thead>
<tr>
<th>Number of words</th>
<th>Using String</th>
<th>Using StringBuilder</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>5 ms</td>
<td>1 ms</td>
</tr>
<tr>
<td>2000</td>
<td>17 ms</td>
<td>1 ms</td>
</tr>
<tr>
<td>4000</td>
<td>71 ms</td>
<td>2 ms</td>
</tr>
<tr>
<td>8000</td>
<td>278 ms</td>
<td>2 ms</td>
</tr>
<tr>
<td>16000</td>
<td>1126 ms</td>
<td>2 ms</td>
</tr>
<tr>
<td>32000</td>
<td>4870 ms</td>
<td>3 ms</td>
</tr>
</tbody>
</table>
Inserting

- A `StringBuilder` also permits characters or strings to be inserted into the *middle* of the string it represents.

  ```java
  public StringBuilder insert(int offset, String s)
  ```

- This inserts the string `s` into the `StringBuilder` starting at the location `offset` - the other characters are “shifted along”

  0123456789

  Hello John
Inserting

StringBuilder s = new StringBuilder("Hello John");
s.insert(5," to");

| 0123456789 | 01234 | 56789 |
| Hello John   | Hello | John  |

| 0123456789... |
| Hello to John |

```java
StringBuilder s = new StringBuilder("Hello John");
s.insert(5," to");
```
Inside a StringBuilder

- The `StringBuilder` internally maintains an array to store the characters.
- Usually the array is a bit longer than the number of characters currently stored.
- If `append` or `insert` causes the number of characters to exceed the capacity, then the `StringBuilder` automatically creates a new bigger array and copies everything over.
- This basic mechanism is used in all of Java’s “growable” classes.
Files

- Java provides new and simplified API for reading and writing files.
- There is an excellent tutorial here:

- Week 10 lab: Write a utility class to read and write all lines to and from simple text files using the “commonly used small files” methods.
- You will need this file class in the project.

- Also see Barnes and Kölling Chapter 12.9
File I/O Methods from Simple to Complex

Use these for CITS1001 project

- `ReadAllBytes` readAllLines
- `newBufferedReader`, `newBufferedWriter`
- `newInputStream`, `newOutputStream`
- `newByteChannel`
- `FileChannel`

- Commonly used
- Small files
- Text files
- Streams, unbuffered
- Buffers
- Channels and
- Advanced features
- File-locking,
- Memory-mapped I/O

Source: http://docs.oracle.com/javase/tutorial/essential/io/file.html
Review

- On computers Characters are represented by a standard code: either ASCII or Unicode
- `String` is one of the classes of the standard Java library
- The `String` class represents character strings such as “This is a String!”
- Strings are constant (immutable) objects
- `StringBuilder` is used for changeable (mutable) strings
- Be aware that Java7 has a new (better) API for reading and writing files
- Use the right library for your Strings and Files - it makes a difference!