A Crash Course in Java:

String Processing and Unicode

Unicode

- Sooner or later, you'll need to manipulate non-ASCII characters.
- Java makes this easy and automatic.
- Java is heavily based on the international Unicode standard for encoding characters.
- A character is *not* a byte, or even a pair of bytes!

Character sets

- A **character set** is just a set of characters, each having a unique “id number” within that set.
- Here are a few common character sets:
  - **US-ASCII** has ~128 characters in it, and is principally English letters (upper- and lowercase), numbers, punctuation, and a single space.
  - **Latin-1 (ISO-8859-1)** has ~256 characters: all of ASCII, plus letters used in Western-European languages (like á and ç).
  - **Latin-7 (ISO-8859-7)** has ~256 characters: all of ASCII, plus Greek letters (like π, α, and Ω).

US-ASCII

- Contains 128 characters, or “code points”:
  - **Graphical** characters (0x20-0x7E): English letters, numbers, punctuation, and a single space.
  - **Control** characters (0x00-0x1F, 0x7F): tab, carriage return, linefeed, “bell”, delete, and many others.
- Every character is meant to be stored in 1 byte.
- Common characters:
  - 0x20 single space
  - 0x41 capital 'A'
  - 0x61 lowercase 'a'
  - 0x7E tilde (~)
  - 0x09 tab
  - 0x0A linefeed
  - 0x0D carriage return

Latin-1 (ISO-8859-1)

- Contains 256 characters (0x00-0xFF), meant to be stored in a single byte.
- All of ASCII, *plus* letters used in Western-European languages (French, Spanish, etc.).
- Common characters:
  - 0x41 uppercase 'A' (because Latin-1 contains ASCII)
  - 0x61 lowercase 'a' (ditto)
  - 0xA9 copyright symbol: ©
  - 0xC1 uppercase 'A' with acute accent: Á
  - 0xE1 lowercase 'a' with acute accent: á
  - 0xE7 lowercase 'c' with cedilla: ç

Latin-7 (ISO-8859-7)

- Contains 256 characters (0x00-0xFF), meant to be stored in a single byte.
- All of ASCII, *plus* Greek letters.
- Common characters:
  - 0x41 uppercase 'A' (because Latin-7 contains ASCII)
  - 0x61 lowercase 'a' (ditto)
  - 0xA9 copyright symbol: ©
  - 0xC1 uppercase Greek 'Alpha': Α
  - 0xE1 lowercase Greek 'Alpha': α
  - 0xE7 lowercase Greek 'eta': η

Unicode:

Why these are not enough

- How to represent characters in languages with more than 256 characters (Chinese, Japanese, Korean)?
- Sometimes we need to intermix characters from different sets in the same document:

  Ελληνικά • Français • 日本語 • Русский
- This is not a "font" issue! A font says *how* to display information; a character *is* information.

Enter Unicode

- Unicode is a *very large* character set: 95,000 characters and growing!
- Nearly all modern and historical characters exist somewhere in Unicode.
- Also includes smileys, math symbols, Tolkien's Elvish characters, and Klingon!

Unicode character codes

- Unicode does not dictate how a given character is represented on disk or in memory. *A character is just a number.* Think of it as a unique ID.
- We indicate a Unicode character by "U+" followed by the ID as a hex number:

U+0041	English capital letter "A"
U+00E7	Lowercase C-with-cedilla: ç
U+20AC	Euro symbol: €
U+263A	Smiley: ☺
U+1D160	Eighth note: ♪

- First 256 characters are identical to Latin-1!

Unicode escape sequences

- C++/Perl let you specify characters inside double-quoted strings via octal (**\012**) and hex (**\xAB**)
- Java supports these, but *also* lets you use Unicode (**\uABCD**) to represent U+ABCD.
- Western bias makes some conversion easy:

\u0000-\u007F	*Same as \x00-\x7F in ASCII*
\u0080-\u00FF	*Same as \x80-\xFF in Latin-1*

Using Unicode in Java

- You can use it in string constants, of course:

```
// Print out "François":
System.out.println("Fran\u00E7ois");
```

- Character constants work too:

```
// Assign 'ç' (that char is 2 bytes!):
char cCedilla = '\u00E7';
```

- And so do identifiers!

```
// Create a French waiter:
String gar\u00E7on = "Fran\u00E7ois";
```


Unicode:

A small gotcha

- **Uni**code is **uni**fied: attempt has been made to have each character appear only once. It's not just an amalgam of existing character sets...

A

A

A

Actual glyphs in
Lucida Sans Unicode

...but beware!

Unicode U+0041 is a Roman capital “A”.
Unicode U+0391 is a Greek capital “Alpha”.
Unicode U+0410 is a Cyrillic capital “A”
They look identical!

Identical-looking Unicode characters have been used in domain spoofing attacks (the “IDN homograph attack”). Think about that the next time you click on a link to “Amazon.com”...

Unicode:

Transformation Formats

- Remember, a character is just a number, but computers don't store “numbers”: they store *bits and bytes.*
- A scheme for representing a sequence of Unicode characters as a sequence of bits is called a **transformation format.**
- It's just a way of “encoding” the character information, for when you need to...
  - Represent the character in memory
  - Store the character in a file
  - Write the character through a network pipe

Unicode:

The UTF-16 Encoding

- A common Unicode representation inside a running software application.
- Characters U+0000 to U+FFFF are stored as a 2 byte sequence which contains the code number:
  - U+00E7 is the 2-byte sequence [00][E7].
  - "Cat" is 6 bytes: [00][43][00][41][00][74].
- Characters U+10000 and beyond are represented by special two-character (4 byte) combinations.
- Simple and straightforward for nearly all common characters.

Unicode:

Problems with UTF-16

- ASCII / ISO-8859 text now takes *twice* the space, since each character needs 2 bytes instead of 1.
- ASCII data can't be processed by common tools when stored as Unicode -- especially due to “zero” bytes (0x00), which in many applications signal “end of string”!
- Fortunately, other formats exist...

Unicode:

The UTF-8 Encoding

- Most popular Unicode representation. Based on # of data bits in Unicode character:

Data bits	UTF-8 representation
0 - 7	`0aaaaaaa`
8 - 11	`110bbbaa 10aaaaaa`
12 - 16	`1110bbbb 10bbbbaa 10aaaaaa`
17 - 21	`11110ccc 10ccbbbb 10bbbbaa 10aaaaaa`

- Many nice features:
  - 7-bit ASCII already *is* UTF-8, and all 8-bit UTF-8 bytes are always *non*-ASCII data.
  - Already "compressed" for ASCII/ISO-8859.
  - Unambiguous bit patterns allow synch/repair.
  - Sort order is preserved.

Local character encodings

- UTF-16 is convenient for representing characters internally, but most existing file formats are *not* 16-bit-Unicode based!
- When Java's I/O layer writes characters to an output stream, it *automatically* converts them from the internal **16-bit Unicode encoding** to your **local character encoding.**
- Vice-versa when reading characters.
- Characters which can't be converted to the local encoding may be distorted or lost.

Probably ASCII or Latin-1 for you

Unicode:

Automatic conversion

- Compile and run the following in America:

```
public class Hello {
        public static void main(String[] args) {
                System.out.println("Fran\u00E7ois");
        }
}
```

```
% java Hello > Hello.out
% less Hello.out
  Fran<E7>ois
% ls -l Hello.out
  -rw-r--r--     8     Hello.out
```

- If your local charset is ISO-8859-1,
you'll see these 8 bytes: *Fran[E7]ois*
If it's UTF-8, you'll see 9 bytes: *Fran[C2][A7]ois*

Unicode:

Character corruption

- Characters must be turned into a *sequence of bits* if you want to store them in memory, save them on disk, or transmit them through the Internet.
- But what if the software *reading* those bits mistakenly assumes a different character encoding than the software *writing* those bits?
  - Reading UTF-8 bytes as if they were US-ASCII
  - Reading ISO-8859-1 bytes as if they were UTF-8
- The result is **character corruption**...

Unicode:

Example of UTF-8 corruption

Ç

Suppose you have a file in UTF-8, containing a c-cedilla (U+00E7)...

0	0	E	7	U+00E7 as hex digits
0000	0000	1110	0111	U+00E7 as bits
bbbb	bbbb	aaaa	aaaa	Unicode byte name for each bit

110bbbaa	10aaaaaa	UTF-8 encoding for 8-11 data bits
11000011	10100111	UTF-8 encoding for U+00E7

1100	0011	1010	0111	U+00E7 as UTF-8 bits
C	3	A	7	U+00E7 as UTF-8 hex digits

Ã §

Here is what you'd see if you tried to display bytes [C3][A7] on a terminal which assumes that data written to it is ISO-8859-1 instead of UTF-8.

Unicode:

Corruption is everywhere

- In a web application that takes form data and stores it in a database table, the same character may be encoded, transmitted, parsed, and re-encoded many times until it lands in the table!
- Some symptoms:
  - You see garbled text where you expected a non-ASCII character (like “FranÃ§ois”). That's just a **misinterpretation** of the bytes.
  - You see text with missing characters, or characters replaced with “?” (like “Franois” or “Fran?ois”): that's **data loss**, usually caused by stuffing Unicode into a more limited byte representation like US-ASCII.

Project: Unicode

Write a Java program which outputs some text in various languages. Be sure to include a Roman "A" and a Greek capital Alpha (\u0391), as well as some Cyrillic or Hebrew characters. Capture the output to a file, and examine it with a word processor and/or a binary dump.

- How did the Alpha translate? Same as "A", or different?
- What about the Cyrillic/Hebrew letters?
- Some UNIX shells have environment variables which let you change the locale. Try doing this, and re-running your tests. Any change?

Strings

- Java has a number of utilities for dealing with [Unicode] strings.

java.lang.String

- Not a primitive type, but it *is* a special class:
  - String literals are automatically converted to instances of **String**.
  - Java uses **+** operator for concatenation.
  - **+** will promote to String if either operand is a String (e.g., "Super" + 8 = "Super8").
  - Special ***toString()*** method of Object for coercing objects to strings when type-promoted.
- Not like C strings!
  - No NUL-terminator.
  - Bounds-checked (**RuntimeException** thrown).

Strings:

String access

charAt(i)	Get **char** at given index (0-based).
compareTo(s)	Is this string <, =, > the other?
equals(s) **equalsIgnoreCase**(s)	Does this string have the *same characters* as the other string.
indexOf(sub) **lastIndexOf**(sub)	0-based pos'n of leftmost/rightmost occurrence of given substring/char.
length()	Get number of characters.
startsWith(s) **endsWith**(s)	Does this string start/end with given string?
substring(from, thru)	Return **new** substring of this.

String "manipulation"

replace(oldc, newc)	Return **new** string with char replaced.
toUpperCase() **toLowerCase**()	Return **new**, up/down-cased string.
trim()	Return **new** string with leading and trailing whitespace removed.

*...Hey, why do these all create **new** Strings...?*

Strings are constant!

- There's no way to actually *modify* a String.
  - Rationale: when you know object is constant, *tremendous* amount of optimization is achievable, esp. in a multithreaded environment.
  - Since many apps are string-heavy, this gives major performance gains.
  - Think of this classname as really being "StringConstant".
- So how do you change a String?
  - Assign contents to **StringBuffer**, modify *that*, and assign it back...

Strings:

java.lang.StringBuffer

- Editable String-like object.
- Lacks many "interesting" methods of String.
- Primarily **append(*x*)** and **insert(index, *x*)**:
  - Work for most primitive types *x*, and also any Object (stringified with **toString**()).
  - Return **this**; can method-chain for efficiency.
  - This is how **+** really works! Compiler translates:

```
"Hello " + pPerson + '\n'
```

```
(new StringBuffer())
    .append("Hello ").append(pPerson).append('\n')
    .toString()
```

StringBuffer optimization

- Predeclare approx size in constructor.
  - Minimizes reallocs.
  - Especially important for many small appends.
- Defer toString() until "edits" are done.
  - StringBuffer's toString() does **copy-on-write:** lets new String *share* its internal char buffer, copies buffer *only* if subsequent edits made.
  - If no edits, no copy!
- Don't mix appends with **+** .
  - Redundant and inefficient. Pick one or the other.

Strings:

Beware substrings!

To avoid unnecessary memory allocations, **substring()** returns String with start/end "pointers" into parent String.

- Perfectly legit, since Strings are constant.
  (See? Optimizations abound!)
- **However**, that means parent *can't be garbage-collected* until all children are gone!

```
static String[] prefix = new String[10000];
...
while (aVeryLongLine = readLine(input))  {
        prefix[i++] = aVeryLongLine.substring(1, 3);
}
```

Constructing safer substrings

- **Solution** is simple: create *copy* of substring and let substring be GC'ed:

```
while (aVeryLongLine = readLine(input))  {
        prefix[i++] = new String(aVeryLongLine.substring(1, 3));
}
```

"Subclassing" strings

- String and StringBuffer are **final classes**: you can't subclass them.
- **Problem**: weak typing leads to accidents:

```
// Set country (US, GB, CA...):
public void setCountry(String pCountryCode) { ... }
```

```
address.setCountry("USA");    // d'oh! can't stop this!
```

- **Solution**: wrap class *around* String.

```
public class CountryName {
        private String mText;
        public String toString() { return mText; }    ...
```


Strings vs. byte[]s

- Some String methods punt on the chars-are-not-bytes issue: they have been deprecated.

```
while (aVeryLongLine = readLine(input))  {
        prefix[i++] = new String(aVeryLongLine.substring(1, 3));
}
```

- Remember **character-encoding**!
  If converting between Strings and bytes, do one of these...
  - **Affirm in comments** that you really do want to convert based on current locale's default character-encoding (e.g., "UTF8")
  - **Specify** character-encoding explicitly (last arg).

Project: sprintf

Java lacks C's **sprintf().** Write code for this very convenient utility.

- You had at least two possible strategies: static vs. non-static method. Which seemed best, and why?
- How did you deal with variable-length argument lists?
- Did you consider internationalization?
If so, did it have an affect on your implementation?

