

Character Sets, Encodings, Java and Other Headaches

Brian Clapper
ArdenTex, Inc.
bmc@ardentex.com

Introduction

- Java can read, write, and convert among various character encodings.
- But many people don't entirely understand:
 - the difference between Unicode and UTF-16
 - how ISO 8859-1, Windows 1252, and ASCII are related
 - how Java translates between character encodings
 - what pitfalls await the unwary, when dealing with databases, web browsers, and data files

Introduction

This talk attempts to demystify the terminology, the technology, and the trials and tribulations associated with handling multiple character sets in Java applications.

What This Talk Is, and Is *Not*

- This talk is not a discussion about internationalization, localization, or multinationalization.
 - Those are worthy topics, but beyond the scope of this presentation
- This talk *is* a discussion of character sets, encodings, and conversions

First, some terminology

- The terms *character set* and *encoding* are often used interchangeably.
 - This conflation of terms is incorrect.
- Most people do not make a distinction between the numbers assigned to characters and the way those numbers are stored in the computer.
- I think that distinction can be clarifying.

First, some terminology

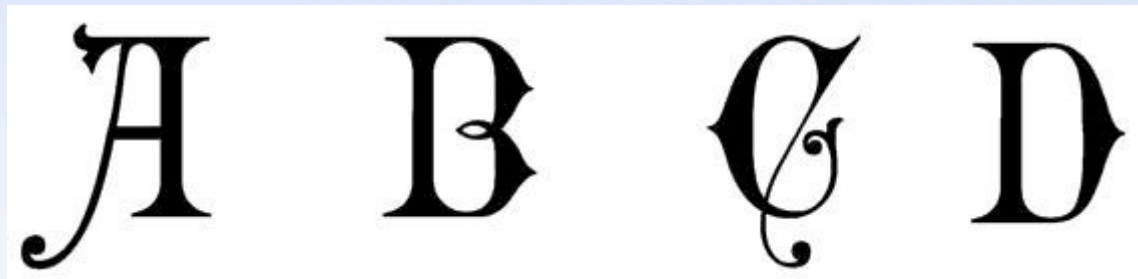
Some terms

- **Character:** An abstraction, or an idea with semantic meaning to humans.
 - The letter “A” is a character.
 - So is the “&” symbol.
- **Glyph:** A visual representation of a character, in *any* medium.
 - Think of a calligraphy alphabet

First, some terminology



A B C D E



A B C D

These are glyphs.

First, some terminology

- **Character Set:** A collection of characters
- **Font:** A collection of glyphs, usually in a specific style (Arial, Helvetica, Times, Old English, etc.)
- **Coded Character Set:** A character set where each character has been assigned a specific numeric value, or *code point*.
 - Unfortunately, the term “character set” is also sometimes used here, leading to confusion.
 - Think of this as a “chart on the wall”.

First, some terminology

- **Character Layout:** My term, used in this presentation as an alternative to “Coded Character Set”.
 - I introduce this non-standard term solely to avoid confusion between “character set” and “coded character set”.
 - (Terminology is a real problem in this area.)
- **Encoding:** How the numbers in a character layout are encoded
 - Are they stored in 8 bits? A word? A multibyte sequence?

Common Character Layouts

- We'll discuss four common character layouts in this presentation:
 - ASCII
 - ISO 8859-1
 - Windows 1252
 - Unicode

Character Layouts: ASCII

ASCII (American Standard Code for Information Interchange), a.k.a., US-ASCII

- First edition: 1963
- Based on ordering of English alphabet
- Consists of 128 characters, including non-printing control characters
- Commonly encoded as 7 bits in an 8-bit byte
- That's not the only historical encoding, though (as we'll see)

Character Layouts: ISO 8859-1

ISO 8859-1, a.k.a, Latin-1

- Part of ISO/IEC 8859 series of ASCII-based character sets
- Generally, we treat ISO 8859-1 as an encoding, but for the moment, let's treat it as a character layout.
- Identical to US-ASCII from 0 to 128.
- Has more characters (including many with accent marks), from 160 through 255.
- Codes 128 through 159 are undefined

Character Layouts: Windows 1252

- Also called “CP 1252”, for “code page 1252”
- The normal character set on Windows
- Again, most people think of it as an encoding. I say it's also a character layout.
- Identical to US-ASCII from 0 to 128.
- Identical to ISO 8859-1, from 160 through 255.
- Codes 128 through 159 are Windows-specific

Character Layouts: Windows 1252

- In the Windows world, sometimes called the “ANSI code page”
- Never standardized by ANSI. Name was taken from an early ANSI draft, later modified to become ISO 8859-1.
- Microsoft says: "The term ANSI as used to signify Windows code pages is a historical reference, but is nowadays a misnomer that continues to persist in the Windows community."

Character Layouts: Unicode

- A universal character set, maintained by the Unicode Consortium (unicode.org)
- Provides “the basis for processing, storage, and interchange of text data in any language in all modern software and information technology protocols.”
- Unicode Basic FAQ calls it an encoding, but I say it's a character layout. ;-)
- Characters are also called *code points* and are denoted by U+xxxx, where xxxx is the hexadecimal character code.

Character Layouts: Unicode

- Very large, containing all the characters for all the writing systems of the world, ancient and modern
- Codes 0 through 127 are the same as ASCII
- Codes 160 through 255 are the same as Windows 1252 and ISO 8859-1
 - Note, though, that the encodings of these non-ASCII characters turn out *never* to be the same as Windows 1252 or ISO 8859-1
- Most common encodings: UTF-16, UTF-8

Character Layouts: Unicode

- Unicode divides characters into 17 *planes*, each with 65,536 (i.e., 2^{16}) characters.
 - Each plane can fit in a 16-bit unsigned integer
- Plane 0, a.k.a., the *Basic Multilingual Plane* (BMP):
 - contains characters U+0000 through U+FFFF
 - consists of most of the character assignments so far
- If you delve into this stuff at any level, you'll see lots of references to the BMP

Common Encodings

- Okay, so what's the difference between this odd *character layout* term and an *encoding*?
- Let's start by looking at each of the previously mentioned character layouts.

Common Encodings: ASCII

Typically encoded as 7 bits in an 8-bit byte

- Not the only historical encoding, though.
- e.g., 1970s-era Control Data Corporation Cyber mainframes:
 - Word size was 60 bits
 - Native character set was 6 bits (no lower case), packed 10 per word. Not byte-addressable.
 - For interoperability, they had an ASCII encoding: 7-bit ASCII characters in a 12-bit byte (packed 5 per word)

Common Encodings: ISO 8859-1

- An 8-bit encoding of the ISO 8859-1 character layout.
- There really aren't any other encodings.
- On US-based Unix systems, the JVM typically assumes the host operating system is using ISO 8859-1, unless told otherwise.
- Called `ISO8859_1` in Java

Common Encodings: CP 1252

- An 8-bit encoding of the Windows 1252 character layout. (There really aren't any other encodings.)
- On US-based Windows systems, the JVM typically assumes the host operating system is using CP 1252, unless told otherwise.
- Called **Cp1252** in Java

Common Encodings: Unicode

- “Unicode” is *not* an encoding (at least, not the way I define “encoding”).
- There are several encodings of Unicode.
- You can freely convert between them without losing information.

Common Encodings: Unicode

Let's examine each of these encodings in more detail, starting with UTF-16.

Unicode Encodings: UTF-16

- Java uses UTF-16 as its internal character representation.
- Variable-length word encoding of Unicode
- Each code point (character) is mapped to a sequence of 16-bit words.
- The characters in the Basic Multilingual Plane are encoded in one 16-bit word.
- For characters in other planes, the encoding is a pair of 16-bit words
 - called a *surrogate pair*

Unicode Encodings: UTF-16

- UCS-2 (2-byte Universal Character Set) is an older, similar encoding.
 - Identical to UTF-16, except that it doesn't support surrogate pairs
 - Can only encode the Basic Multilingual Plane
 - Fixed 16-bit length
 - Java used UCS-2 initially, but added UTF-16 supplementary character support in J2SE 5.0

Unicode Encodings: UTF-16

There are three UTF-16 encodings, to handle different byte orderings.

- UTF-16: Bytes can be big-endian or little-endian
 - Requires 16-bit leading byte-order mark (BOM)
 - BOM is encoded version of the Zero-width Non-breaking Space (ZWNBSpace) character, U+FEFF
 - 0xFE 0xFF means big-endian
 - 0xFF 0xFE means little-endian
 - Java name for this encoding: **UTF16**

Unicode Encodings: UTF-16BE

- Bytes are in big-endian order
- No byte-order mark
- Java name for this encoding: **UTF-16BE**

Unicode Encodings: UTF-16LE

- Bytes are in little-endian order
- No byte-order mark
- Java name for this encoding: **UTF-16LE**

Unicode Encodings: UTF-8

- A variable-length byte-oriented encoding
- Encodes each character (code point) in 1 to 4 octets (8-bit bytes)
- There's a straightforward mapping that dictates how many bytes a character needs.
 - Depends on the code point number
 - Good description at:
<http://en.wikipedia.org/wiki/UTF-8#Description>

Unicode Encodings: UTF-8

- First 128 characters (US-ASCII) require a single byte.
 - Indistinguishable from ASCII encoding for these characters
- Next 1,920 characters use two bytes
 - © is one byte in CP 1252 & ISO 8859-1: 0xA9
 - In UTF-8, © is encoded as: 0xC2 0xA9
- Three bytes needed for remainder of most common characters
- Compatibility with ASCII makes UTF-8 popular for web pages, documents, etc.

Unicode Encodings: UTF-8

BOM Idiocy™

- Many Windows programs (e.g., Notepad) add the bytes 0xEF 0xBB 0xBF at the start of files saved as UTF-8
 - This is the UTF-8 encoding of a Unicode byte-order mark (U+FEFF)
 - A BOM is silly for byte-oriented UTF-8.
 - Ostensibly, the intent is to allow code to distinguish UTF-8 files from ISO 8859-1 or Windows 1252 ones.
 - Java does **not** recognize this byte-order mark.
 - You can't “cat” two such files together.

Unicode Encodings: UTF-32

- UTF-32 uses exactly 32 bits for each Unicode code point.
- All Unicode characters can be encoded in 32 bits.
 - Thus, UTF-32 is a fixed-width encoding.
- Highly space-inefficient, so not used very often
- Like UTF-16, there are three related encodings to handle endian issues
- Java names: **UTF-32**, **UTF-32BE**, **UTF32-LE**

How Java Handles Encodings

- Java stores characters internally as UTF-16
- Java uses translation tables to map between external encodings and UTF-16.
 - Map *from* external encoding *to* UTF-16 on input.
 - Map *from* UTF-16 *to* external encoding on output.
- These translations can be lossy. More on this later.

How Java Handles Encodings

For example, what does this do?

```
Reader r = new InputStreamReader(  
    new FileInputStream(  
        path), "UTF-8"));
```

How Java Handles Encodings

For example, what does this do?

```
Reader r = new InputStreamReader(  
    new FileInputStream(  
        path), "UTF-8"));
```

It:

- opens a stream of bytes from a file, via the **FileInputStream**
- wraps the **FileInputStream** in a **Reader** that treats those bytes as UTF-8 encoded characters
- converts the UTF-8 characters to Java's internal UTF-16

How Java Handles Encodings

- Java *only* deals with UTF-16 internally.
- Inbound characters are converted to UTF-16.
- Outbound characters are converted from UTF-16 to whatever the output encoding is.
- Unless you're reading and writing UTF-16, *all* character I/O requires conversion to and from Java's canonical UTF-16 encoding.
- This is a perfectly reasonable and sound approach.

How Java Handles Encodings

Java supports a large number of encodings.
See:

<http://java.sun.com/javase/6/docs/technotes/guides/intl/encoding.doc.html>

Pitfalls

With that out of the way, let's look at some encoding-related pitfalls.

Pitfall #1: Misleading terminology

- Many people and tools use the term “Unicode” inappropriately.
- When Microsoft Office Excel presents you with a *Save As Unicode Text* option, what does that mean?
 - Does it mean UTF-8? (This answer seems logical.)
 - UTF-16?
 - UTF-32? (Man, I hope not...)

Pitfall #1: Misleading terminology

- It turns out to mean UTF-16.
- If you don't know that, you'll have trouble getting your Java program to read a “Unicode” CSV file produced by Excel.

Pitfall #2: Lossy Conversions

- You can lose information when writing to non-Unicode encodings. Example:
 - You read a file encoded in Windows 1252
 - The file has a ™ symbol (0x99 in Windows 1252)
 - Java translates the ™ to U+2122 in UTF-16
 - You write the data out as ISO 8859-1
 - Java cannot translate the ™ to ISO 8859-1, because the character does not exist in that encoding
 - Java writes a “?” character, instead

Pitfall #2: Lossy Conversions

Moral: Choose your encodings carefully.

Pitfall #3: The UTF-8 BOM Idiocy™

- Customer sends you a UTF-8 CSV file.
- You don't realize it has a UTF-8 byte-order mark.
- You decode the CSV file into an internal set of rows and columns.
- You find that the first column of the first row has an unexpected U+FEFF character.
 - If you write the file back out, those characters get translated to the target encoding (or replaced with “?”)
 - The U+FEFF character will screw up numeric parsing

Pitfall #3: The UTF-8 BOM Idiocy™

One solution:

- Open the file as UTF-8
- Wrap your **Reader** in a **PushbackReader**
- Read the first character, which converts it from UTF-8 to UTF-16
 - If the character is U+FEFF (the BOM), ignore it and move on.
 - If the character is *not* U+FEFF, push it back and move on.

Pitfall #4: Browsers

Consider this scenario:

- Your Grand Web Application delivers web pages encoded in ISO 8859-1.
- A user, in Internet Explorer, enters data in one of your forms and includes a TM symbol (0x99 in Windows 1252).
- On submission, IE converts the form data from Windows 1252 to ISO 8859-1, due to the document's encoding.
- ISO 8859-1 has no TM. IE doesn't map the TM to a “?”. Instead, it leaves it alone.

Pitfall #4: Browsers

- Java, reading the form, converts the incoming data from ISO 8859-1 to UTF-16. When it sees the (illegal) 0x99 character, it doesn't barf or convert it; it just passes the character along as a Unicode U+0099.
- U+0099 is a control character in Unicode. The user's TM just became an obscure non-printable.
- If you write the data as ISO 8859-1 or CP 1252, it'll go out as a 0x99. If you write it as UTF-8, you get 0xC2 0x99. In ASCII, you'll write a “?”.

Pitfall #5: Databases

Database vendors handle character sets differently.

Pitfall #5: Databases

- With Oracle, the encoding is associated with the instance.
 - Create the database as ISO 8859-1, and VARCHAR2 columns hold ISO 8859-1 characters.
 - Create the database as UTF-8, and those columns hold UTF-8.
 - This approach is nice, because the DDL remains the same.
 - Oracle also supports NCHAR and NVARCHAR2, for column-specific encodings.

Pitfall #5: Databases

- With SQL Server, VARCHAR and CHAR are 8-bit entities.
 - You *could* use them to store UTF-8, but you'd have to keep track of conversions yourself.
 - The character sizes would be off, as well.
- If you change your DDL to use NVARCHAR and NCHAR, you can store “Unicode” (encoded as UTF-16).

Pitfall #5: Databases

- MySQL allows you to specify character sets at the server, database, table, and column level.

- Supports many encodings, including UTF-8.

- Examples:

```
CREATE DATABASE foo CHARACTER SET utf8
```

```
CREATE TABLE tbl CHARACTER SET utf8
```

```
CREATE TABLE tbl (  
    col1 VARCHAR(5) CHARACTER SET latin1,  
    col2 VARCHAR(20) CHARACTER SET utf8  
)
```

Pitfall #5: Databases

- PostgreSQL supports individual database-level encoding
 - Supports many encodings, including UTF-8.
 - **See** <http://www.postgresql.org/docs/8.4/static/multibyte.html>

- Example:

```
CREATE DATABASE db WITH ENCODING 'UTF8'  
$ createdb -E UTF8 db
```

The Nightmare Scenario

- I actually had to debug this situation in a former life.
- It combines many of the previous pitfalls and illustrates
 - how hairy this stuff can get
 - how you might debug some of the problems

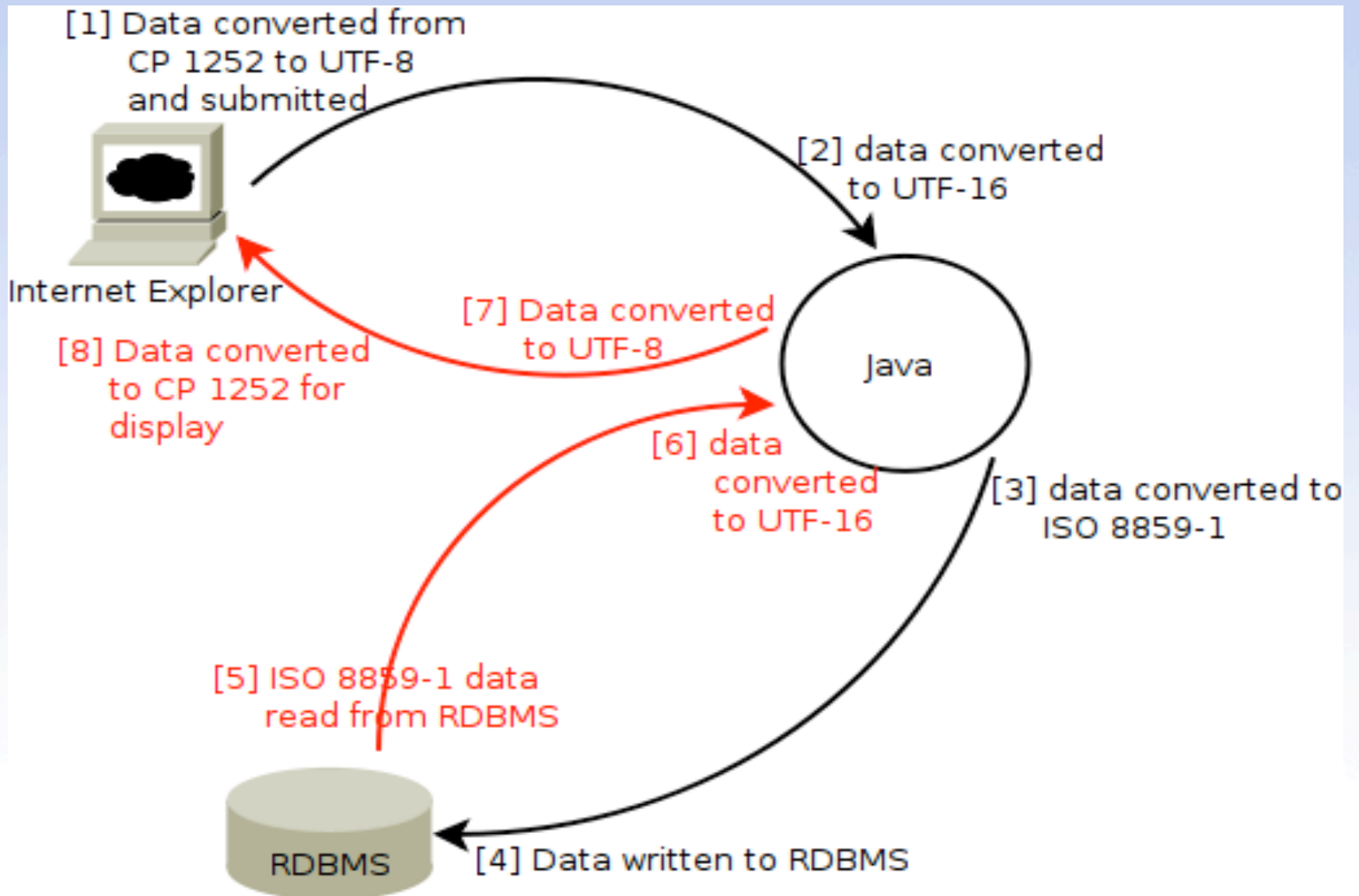
The Nightmare Scenario

- Application: A product data management tool
- Components we care about here:
 - Web-based user interface, backed by Java servlets
 - Support for Oracle or SQL Server

Nightmare Scenario #1

- Customer A called support
- Problem:
 - User entered product data, including a TM symbol
 - When product data redisplayed on the page, the TM had been replaced with a “?”
- What happened, and how do we fix it?
- First, what's the configuration?
 - User is on Windows, with Internet Explorer
 - Database is using ISO 8859-1 encoding

Nightmare Scenario: #1



Nightmare Scenario #1

What actually happened:

- IE converted the CP 1252 TM (0x99) to a UTF-8 TM symbol
- Java converted it to a UTF-16 version
- The JDBC driver used the JDK's encoding tables to convert the TM to ISO 8859-1, so it was converted to a “?” and stored that way in the database.

Nightmare Scenario #2

- Variation of Nightmare Scenario #1: Customer B called support
- Problem:
 - User entered product data, including a TM symbol
 - When product data redisplayed on the page, the TM had been replaced with a double quote.
 - User is on Windows, with Internet Explorer
 - Database is using Windows 1252 encoding
 - Different database engine than Customer A

Nightmare Scenario #2

What actually happened:

- IE converted the CP 1252 TM (0x99) to a UTF-8 TM symbol
- Java converted it to a UTF-16 version (0x2122)
- The JDBC driver passed the unconverted UTF-16 TM over the wire to the RDBMS.
- The RDBMS truncated it to 8 bits, yielding 0x22 (a double quote in Windows 1252).
- A double quote was stored in the database.

Lessons to Learn

- This stuff is a complete pain in the ass.
- But, because there's no one character set that is perfect for all occasions, this stuff is also unavoidable.
 - One way or another, you'll always be converting characters.
- Understanding how it all fits together can save valuable time and confusion.
- Lots of people *don't* understand this stuff, so with a small amount of effort, *you* can become the resident expert.

Specific Tips and Tricks

- If you encounter this kind of problem, get as much information as possible about the environment.
 - What browser are they using, if applicable?
 - Where are they running Java (so you know the default encoding)?
 - What databases is Java talking to, and how are they encoded?
 - What are the encodings of the files being read?
 - What encoding is used when HTML forms are submitted?

Specific Tips and Tricks

- Reproducing the problem in an isolated test case is the gold standard.
- Wire-sniffing tools like *tcpdump*, Ethereal and WireShark are invaluable: They will show you what's actually going over the wire, so you don't have to guess.
- You absolutely *must* have a good hex dump utility. (e.g., on Unix: `hexdump -C`)
- Write a small program that converts between encodings. It's easy to do, and it's damned useful to have around.

Useful Links

- <http://www.unicode.org/>
- <http://www.joelonsoftware.com/articles/Unicode.html>
- <http://en.wikipedia.com/wiki/Unicode>
- <http://en.wikipedia.com/wiki/UTF-16>
- <http://en.wikipedia.com/wiki/UTF-8>
- Tim Bray's "Characters vs. Bytes": <http://www.tbray.org/ongoing/When/200x/2003/04/26/UTF>
- This presentation will be on the Philly JUG site, as well as my web site: <http://www.ardentex.com/publications/charsets-encodings-java.pdf>

Acknowledgments

The following individuals reviewed the original draft(s) of this presentation and provided valuable feedback:

Mark Chadwick

Matt Dymek

Tom Hjellming

Steve Sapovits

Drew Sudell

Jon Tulk

Thank you!

- If you made it this far, and your brain didn't explode, pat yourself on the back.
- Feel free to drop me an email, if you have questions or comments about this presentation: *bmc@ardentex.com*