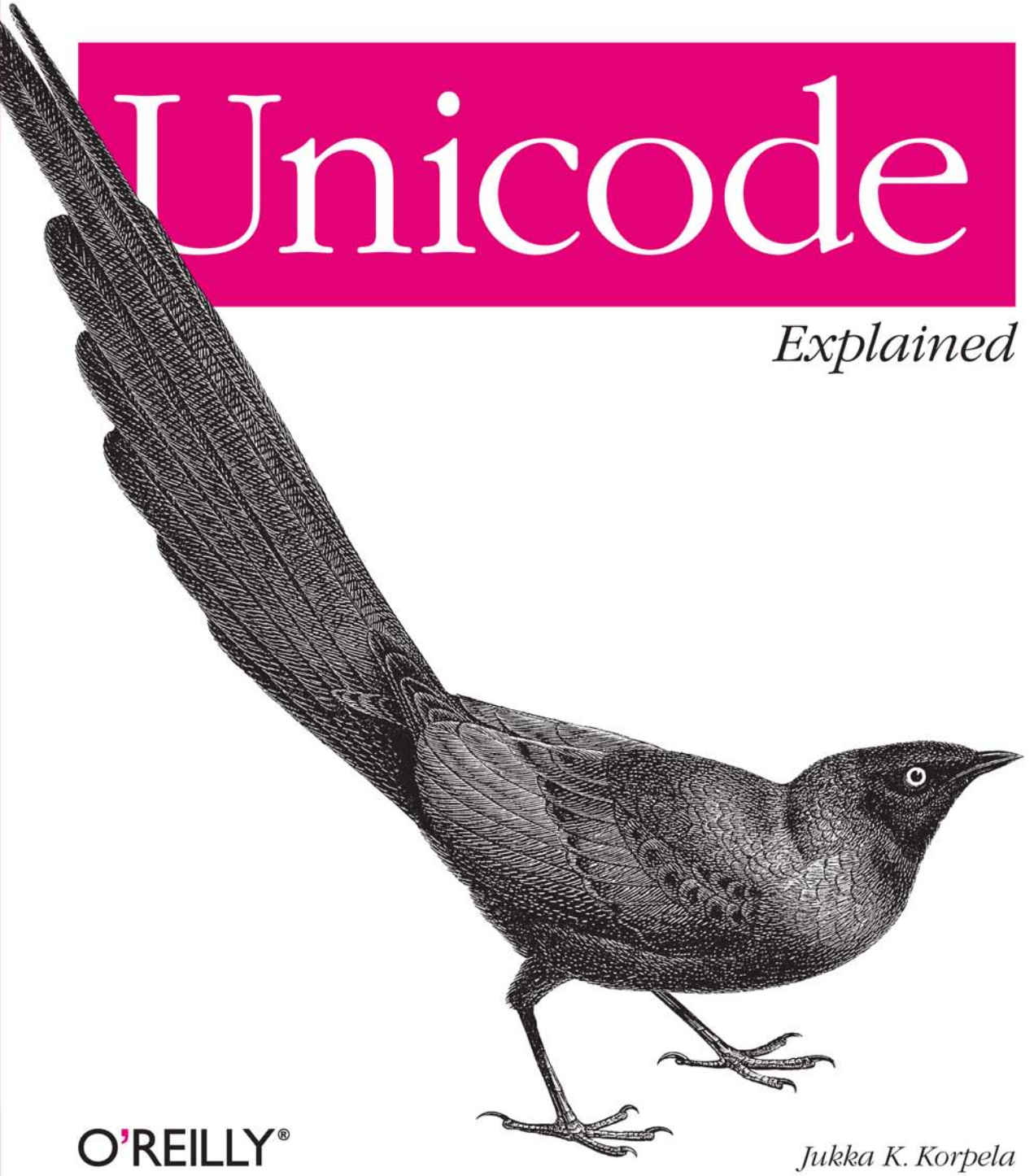
Internationalize Documents, Programs, and Web Sites

Unicode

Explained



O'REILLY®

Jukka K. Korpela

O'REILLY®

Unicode Explained

Do you need to write a single software product or web site to target multiple platforms, languages, and character sets without re-engineering? There are hundreds of encoding systems for mapping characters to numbers, but Unicode promises a single mapping, which makes a single worldwide product solution possible. It's no wonder that industry giants like Apple, Hewlett-Packard, IBM, and Microsoft have all adopted Unicode.

This comprehensive reference contains everything you need to understand Unicode. It takes you on a detailed tour through the complex character world. For starters, it explains how to identify and classify characters—from the common to the specialized. Then it shows you how to type these characters, interpret their properties, and process character data in a robust manner.

The first few chapters teach the basics of Unicode and character data. They provide a firm grasp of the terminology you need to make reference to various components, including:

- Character sets
- Fonts and encodings
- Glyphs and character repertoires

The middle section offers more detailed information about using Unicode and other character codes:

- Principles and methods behind defining character codes
- Some of the widely used codes
- Code conversion techniques
- Properties of characters
- Collation and sorting
- Line-breaking rules
- Unicode encodings

The final four chapters cover more advanced material, such as programming to support Unicode.

You simply can't afford to be without the nuggets of valuable information detailed in *Unicode Explained*.

Jukka Korpela is a consultant who specializes in character codes, localization, orthography, usability, and accessibility. After graduating from the University of Helsinki, he taught these subjects at Helsinki University of Technology and other institutions. Later, he worked on localization and accessibility issues at TIEKE, the Finnish Information Society Development Center.

www.oreilly.com

US $59.99 **CAN $77.99**

ISBN: 978-0-596-10121-3

9 780596 101213 55999

Unicode Explained

Unicode Explained

Jukka K Korpela

O'REILLY®
Beijing • Cambridge • Farnham • Köln • Sebastopol • Taipei • Tokyo

Unicode Explained
by Jukka K Korpela

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://safari.oreilly.com*). For more information, contact our corporate/institutional sales department:(800) 998-9938 or *corporate@oreilly.com* .

Editor: Simon St.Laurent
Production Editor: Adam Witwer
Copyeditor: Linley Dolby

Indexer: Joe Wizda
Cover Designer: Karen Montgomery
Interior Designer: David Futato
Illustrators: Robert Romano and Jessamyn Read

Printing History:
June 2006: First Edition.

This book uses RepKover™, a durable and flexible lay-flat binding.

ISBN: 978-0-596-10121-3
1216069064

Table of Contents

Part III. Advanced Unicode Topics

Preface

Characters often seem simple on the surface, but they are at the heart of a wide variety of data communications and data processing problems, including text processing, typesetting, styling text, text databases, and the transmission of textual information.

Computers were invented just for computing. For quite some time, they were so expensive that their use was limited to the most important numerical calculations that would have been impossible otherwise. Text was used mainly to add legends and headings to numeric output, often using a very limited character repertoire, maybe even lacking lowercase letters. As the cost of computing has dropped, computers have become extensively used for human communication in text format. Most people think of computers as communicators rather than calculators. People want to communicate in different languages, and we also use notation systems that may require rich repertoires of characters.

Unicode was developed to help make this both possible and smooth. Unicode was first defined in the early 1990s, but its use has progressed fairly slowly. Modern computers often use Unicode internally, but applications and users still tend to work with older character codes, which are often very limited. It has been rather complicated to work with Unicode in text processing, for example. At long last, however, these problems are becoming easier to solve. Information technology is becoming really multinational, supporting different languages, writing systems, and conventions. IT products need to be at least potentially suitable for use in different cultural environments, or "localizable." Unicode itself is just part of the technical basis for all this, but it is an indispensable part.

The technological basis of using Unicode, though still imperfect, is much better than most people's capabilities for making use of it. Even computer professionals often don't know how to work with large repertoires of characters. The bottleneck is lack of a basic knowledge and skills, not a lack of hardware or software.

The concept of a character is one of the most difficult basic concepts in information technology, yet fundamental to text processing, databases, the Web, XML-based markup, internationalization, and other areas. People who encounter Unicode when studying such topics often run into serious difficulties. They mostly find material that assumes that the reader already knows what Unicode is. It might be even worse: it is very

easy to find incorrect or seriously confusing information about Unicode and characters, even in new books. People find themselves in a maze of twisty little passages of characters, fonts, encodings, and related concepts.

This book guides you through the Unicode and character world. It explains how to identify and classify characters—whether common, uncommon, or exotic—and to type them, to use their properties, and to process character data in a robust manner. It helps you to live in a world with several character encodings.

Audience

Readers of this book are expected to be familiar with computers and how computers work, broadly speaking. They are not expected to know computer programming, though many readers will use the contents in system design and programming.

This book is intended for people with different backgrounds and needs, including:

- An end user of multilingual or specialized text-related applications. For example, anyone who works with texts containing mathematical or special symbols, or uses a multilingual database. These readers should probably explore Chapters 1 through 3 first, practice with that content, and then read Chapters 7 and 8.
- An IT professional who needs to understand Unicode and work with it. The need might arise from text data conversion tasks, from creating internationalized software or web sites, or from system design or programming in an environment that uses Unicode.
- An IT teacher who needs a better understanding of character code issues, both to understand the subject area better and to disseminate correct information. There is rather little about character codes in curricula, and this is largely a chicken-and-egg problem: there are no good textbooks, and teachers themselves don't know the topic well enough. The first three chapters of the book could provide the foundation for a course, optionally coupled with other chapters relevant to a particular curriculum.
- An IT student, hobbyist, or professional who keeps hearing about Unicode and needs to work with technologies that use Unicode, such as XML.

Assumptions and Approach

Previous knowledge about character codes is not assumed. If you already know about them, you may need to change your mental model a bit.

This book starts at the ordinary computer user's level. Thus, it unavoidably contains explanations that look trivial to some readers. However, these discussions might help in explaining things to others when needed. The book also contains practical instructions on actually working with "special" characters, and an IT professional might find

this irrelevant. However, studying such issues and practicing with them will help a lot in creating a background for more technical work with the infrastructures of character usage.

In explaining practical ways of doing things, this book often uses Microsoft Windows and Microsoft Office programs as examples. This is because so many people use such software and need to know how to use Unicode in them. Moreover, even if you personally prefer other software, odds are good that you need to work with Windows and Office at times. Information on using Unicode in some other environments can be found in the following:

- Markus Kuhn: "UTF-8 and Unicode FAQ for Unix/Linux," which is available at *http://www.cl.cam.ac.uk/~mgk25/unicode.html*
- Tom Gewecke: "Unleash Your Multilingual Mac," which is available at *http://hometown.aol.com/tg3907/mlingos9.html*

After the first three chapters, this book gets more technical, and many of the issues discussed are abstract and even formal. Therefore, understanding most of the material in the initial chapters is essential for the rest. To most people, it is very difficult to read about abstract things if you lack a concrete background that lets you map the abstract concepts and rules to specific practice.

This book explores Unicode processing generally, but cannot go into great detail on all parts of the Unicode character space. For much more information on ideographic characters and processing of East Asian languages, see Ken Lunde's *CJKV Information Processing* (O'Reilly).

Except for the last chapter (Chapter 11), this book does not assume that the reader knows about computer programming. However, some references to programming are made throughout the book.

Contents of This Book

The book has three parts:

Part I
: Chapters 1 through 3 provide a self-contained tutorial presentation of Unicode and character data. It is aimed at anyone who has a basic understanding of computing, and introduces characters in information technology, with some historical background. Although much of this part is well-known to many IT professionals, it provides a consistent terminology that could give professionals (and especially teachers) a model for talking to laymen about characters.

Part II
: Chapters 4 through 6 give detailed information about using Unicode and other character codes. These chapters are especially aimed at computer science students and teachers, information technology professionals, and people involved in lin-

guistic data processing and databases containing string data. Together with the first part, this covers what every IT professional should know about characters. It explains the principles and methods of defining character codes, describes some of the widely used codes, presents code conversion techniques, and takes a detailed look at Unicode. This includes properties and classification of characters, collation and sorting, line breaking rules, and Unicode encodings.

Part III
: Chapters 7 through 11 discuss relatively independent topics, to be read according to each reader's specific needs. They are topics that are important and even crucial to many, but not necessary to all. For example, if you need to author or administer multilingual web sites, you should read the section on characters in HTML and XHTML. To be honest, I would suggest that most people need to read it at least twice. Character code problems are intrinsically difficult, and very widely misunderstood. It takes time to digest the concepts and principles before you can really start working with the algorithms and tools.

The chapters can be characterized as follows:

Chapter 1, *Characters as Data*
: This chapter describes, at a general level but exemplified by simple and typical cases, how computers represent and process characters. It defines fundamental concepts like character set, code position, encoding, glyph, and font. At this point, Unicode is the only character set discussed, to avoid confusion. To make the discussion more concrete and motivating, some features of writing systems are described. The historical development of character codes is presented to the extent that is necessary for understanding why even apparently simple characters, such as dashes and é, still cause problems. The use of different encodings is illustrated by examples of viewing email messages and web pages, using commands to select the encoding if needed. The basic methods for finding, installing, and selecting fonts are described.

Chapter 2, *Writing Characters*
: This is a practical presentation of some common methods of entering characters, including keyboard variation, special keys, changing keyboard settings, virtual keyboards, character maps, automatic "correction" of character sequences, program commands, and different escape notations. It is largely a collection of recipes, useful, for example, to people who work daily with texts containing "difficult" characters. For this reason, some quick reference tables for very commonly needed characters are presented. However, it is also relevant to IT specialists who need to understand the possible input methods when designing applications and systems. The examples used are mostly from MS Windows and MS Office environments but various alternatives (such as "Unicode editors") are also discussed. HTML and XML character reference and entity reference techniques are presented as well. The chapter ends with an exercise for writing some specialized texts using some of the techniques presented.

Chapter 3, *Character Sets and Encodings*

This chapter describes some very widely used character codes and encodings, mainly ASCII, ISO-8859-1 and other ISO-8859 standards, Windows Latin 1 and relatives, and UTF-8. (However, the semantics of characters are described in Chapter 8.) Some less common encodings such as DOS code pages are described in order to give some basics for working with legacy data and legacy systems. A few widely used multibyte encodings for East Asian languages are briefly described, too. The section describes how conversions between the encodings can be performed, either with the functions of commonly used programs or separate converters. It also discusses practical feasibility of the character sets in different contexts, such as email, Internet discussion forums, and document interchange. MIME is presented to the extent needed for dealing with the charset issue.

Chapter 4, *The Structure of Unicode*

An in-depth presentation of the fundamentals of Unicode, including design principles, coding space, and special terminology. The nature of Unicode as an umbrella standard based on a large number of older standards is explained, as well as its relationship to ISO 10646. The unification principle as well as criticism of it is described.

Chapter 5, *Properties of Characters*

This chapter describes the various properties defined for characters in the Unicode standard and their relationship with some programming concepts. This is, in part, a companion to the much more formal definitions in the standard itself. In particular, compatibility, decompositions, collation, sorting, directionality, and linebreaking properties as well as Unicode normalization forms are described.

Chapter 6, *Unicode Encodings*

This chapter describes UTF-8 and other Unicode encodings in detail, including the algorithmic descriptions and the practical considerations on choosing an encoding.

Chapter 7, *Characters and Languages*

The chapter describes some IT-related requirements of different languages and writing systems, such as how to deal with right-to-left writing. This includes conversions between writing systems (transliteration or transcription). The interaction between encoding, language, and font settings is described. Moreover, language codes, language metadata, and language markup are described, illustrated with XML examples.

Chapter 8, *Character Usage*

This chapter consists of sections devoted to different character blocks and collections that are practically important especially in the Western world. The first section is more generic and discusses the relationship of character standards, orthography, and typography. (Even in purely English-language text, typographically correct punctuation requires characters beyond ASCII.) The chapter contains detailed information about the semantics and usage of individual characters, although the level of detail depends greatly on the importance of the character. All

the major blocks are briefly characterized to give an overview, but the emphasis is on ASCII, different Latin supplements, general punctuation, and mathematical and technical symbols.

Chapter 9, *The Character Level and Above*
: Characters form but one "protocol level," above which there are higher levels such as markup level, record structure level, and application level. This chapter provides guidelines for the coding of information at different levels when there is choice, such as using markup versus character difference (largely still an open problem despite the efforts of the W3C and the Unicode Consortium). This is particularly important for processing of legacy data and for avoiding overly fine distinctions at the character level. The chapter ends with a section on media types for text and the difference between plain text, other subtypes of text, and application types such as text-processing formats.

Chapter 10, *Characters in Internet Protocols*
: This chapter describes how character encoding information is transmitted using Internet protocols, including MIME and HTTP, and how content negotiation works on the Web (for the purposes of negotiating on character encoding). This constitutes a basis for a presentation of some fundamentals of multilingual web authoring at the technical level. Moreover, the use of characters in the protocols themselves, such as Internet message headers and URLs, is described, with focus on the partial shift from pure ASCII to Unicode. In particular, the technical basis of Internationalized Domain Names and Internationalized URLs is described.

Chapter 11, *Characters in Programming*
: This chapter presents a number of ways to represent character and string data in different programming languages, such as FORTRAN, C, C#, Perl, ECMAScript, and Java™, as well as other computer languages such as XML and CSS. It emphasizes both the differences and similarities, which are illustrated with sample programs to perform simple manipulation of string data. The chapter is especially intended for people who teach programming but also for people who study or practice programming in an environment where character data is essential. Programs that cannot distinguish, for example, between an empty string, a space character, the NUL character, and the digit zero will have large problems in a Unicode environment. The chapter also examines requirements for modern processing of character data, including the principle of being prepared to handle a large character repertoire and that of separating internal encoding from input and output encodings. The International Components for Unicode (ICU) activity and its results are described. The chapter also contains a section on Common Locale Data Repository (CLDR) and its future use in disciplined programming. This largely goes beyond the character concept but is motivated by the use of Unicode in CLDR and by the organizational connection with the Unicode Consortium.

Appendix, *Tables for Writing Characters*
: The Appendix provides some commonly needed information useful for entering characters. It includes tables of key sequences, as well as a mapping chart from the Symbol font to Unicode.

Self-Assessment Test

To estimate your progress in knowledge about Unicode, you can perform the following self-assessment test. Read the following statements and comment on each of them with one of the following alternatives (using whatever symbols you find convenient, such as those in parentheses): "I do not understand what the statement says" (??), "I know what it says but I do not know whether it is true" (?), "true" (+), and false (–). Moreover, for any "true" or "false" answer, consider what you would present as an argument in a discussion in which someone says you're wrong.

At any point in reading the book, and especially when you think you have learned enough, reread the statements and perform the test again. You might regard the following as a spoiler, so it has been written backward so that you can hopefully ignore it at this point if you like. It reveals what the test is about: .elpoep ot siht nialpxe ot deen thgim uoy dna ,gnorw era yeht yhw wonk ot laitnesse si ti ecnis ,hguoht ,siht gniwonk htiw deifsitas eb ton dluohs uoY .eslaf lla era yeht tub ,skoob ecnerefer ni neve edam ylnommoc era stnemetats ehT

1. Unicode is a 16-bit character code.
2. Unicode contains all the characters used in the languages of the world.
3. Unicode is meant to replace all the other character codes.
4. Unicode cannot be used in real applications now; it is just a future plan.
5. Using Unicode, the size of a text file gets doubled.
6. We don't need Unicode if we write only in English.
7. Unicode consists of 256 code pages.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic
: Indicates new terms, URLs, email addresses, filenames, and file extensions.

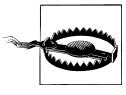
`Constant width`
: Indicates computer code in a broad sense. This includes commands, options, switches, variables, attributes, keys, functions, types, classes, namespaces, methods, modules, properties (does *not* include Unicode "properties"), parameters,

values, objects, events, event handlers, XML tags, HTML tags, macros, the contents of files, and the output from commands.

`Constant width bold`
: Shows commands or other text that should be typed literally by the user.

`Constant width italic`
: Shows text that should be replaced with user-supplied values or by values determined by context.

This icon signifies a tip, suggestion, or general note.



This icon indicates a warning or caution.

The following special notations are used in this book to refer to characters:

"x"
: Refers to character *x* by showing it within double quotation marks. For clarity, characters that might be confused with other characters in the text—i.e., letters a–z, A–Z, and some common punctuation, such as hyphens (-), commas (,), and periods (.)—are enclosed in quotation marks.

U+nnnn
: Refers to a character (or a code point) by its Unicode number. The number *nnnn* is written in hexadecimal notation, usually in four digits using leading zeros if needed.

Web sites and pages are mentioned in this book to help the reader locate online information that might be useful. Normally both the address (URL) and the name (title, heading) of a page are mentioned. Some addresses are relatively complicated, but you can probably locate the pages easily by using your favorite search engine to find a page by its name, typically by typing it inside quotation marks. This may also help if the page cannot be found by its address; it may have moved elsewhere, so the name may work.

Using Code Examples

This book is here to help you get your job done. In general, you may use the code in this book in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing a CD-ROM of examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example

code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Unicode Explained* by Jukka K. Korpela. Copyright 2006 O'Reilly Media, Inc., 0-596-10121-X."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

Safari® Enabled

When you see a Safari® Enabled icon on the cover of your favorite technology book, that means the book is available online through the O'Reilly Network Safari Bookshelf.

Safari offers a solution that's better than e-books. It's a virtual library that lets you easily search thousands of top tech books, cut and paste code samples, download chapters, and find quick answers when you need the most accurate, current information. Try it for free at *http://safari.oreilly.com*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707 829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at:

http://www.oreilly.com/catalog/unicode

To comment or ask technical questions about this book, send email to:

bookquestions@oreilly.com

For more information about our books, conferences, Resource Centers, and the O'Reilly Network, see our web site at:

http://www.oreilly.com

Acknowledgments

The presentation of problems, solutions, and ideas owes much to people with whom I have been in contact in character-related matters through years, such as (roughly in chronological order by their influence) Timo Kiravuo, Alan J. Flavell, Arjun Ray, Roman Czyborra, Bob Bemer, and Erkki I. Kolehmainen.

The reviewers, Andreas Prilop, John Cowan, and Jori Mäntysalo gave a very substantial amount of valuable input, both on content and on presentation. Simon St.Laurent has had an active and supportive role through the entire process as an editor.

PART I

Working with Characters

This part describes the fundamentals of representing character data in computers, including Unicode and other important character codes. It also discusses several practical ways of writing Unicode characters.

CHAPTER 1

Characters as Data

Computers were originally built to process numbers. Over the last few decades, they've become increasingly better at handling text as well, but the transition from human scribbling and beautiful typography to bits and bytes has been complicated. Going from a paper document to a computerized representation of that document means learning about how the computer handles text, and requires learning about characters, character codes, fonts, and encodings. Unicode provides a set of solutions for some of these problems, while retaining presentation flexibility for making text look as we feel it should.

Introduction to Characters and Unicode

Computer programs use two basic data types in most of their processing: characters and numbers. These basic types are combined in various ways to create strings, arrays, records, and other data structures. (Inside the computer, characters are numbers, but the ways that these numbers are handled is very different from numbers meant for calculation.)

Early computers were largely oriented toward numerical computation. However, characters were used early on in administrative data processing, where names, addresses, and other data needed to be stored and printed as strings. Text processing on computers became more common much later, when computers had become so affordable that they replaced typewriters. At present, most text documents are produced and processed using computers.

Originally, character data on computers had limited types and uses. For economic and technical reasons, the repertoire of characters was very small, not much more than the letters, digits, and basic punctuation used in normal English. This constitutes but a tiny fraction of the different characters used in the world's writing systems—about 100 characters out of literally myriads (tens of thousands) of characters. Thus, there was a growing need for a possibility of presenting and handling a large character repertoire on computers; Unicode is the fundamental answer to that.

Why Unicode?

Since you are reading this book, I assume you already have sufficient motivation to learn about Unicode. Nevertheless, a short presentation follows that explains the benefits of Unicode.

Computers internally work on numbers. This means that characters need to be coded as numbers. A typical arrangement is to use numbers from 0 to 255, because that range fits into a basic unit of data storage and transfer, called a *(8-bit) byte* or *octet*.

When you define how those numbers correspond to characters, you define a *character code*. There are quite a number of character codes defined and used in the world. Most of them have the same assignments for numbers 0 to 127, used for characters that appear in English as well as in many other languages: the letters a–z plus their uppercase equivalents, the digits 0–9, and a few punctuation marks. Many of the code numbers in this so-called ASCII set of characters are used for various technical purposes.

For French texts, for example, you need additional characters such as accented letters (é, ô, etc.). These can be provided by using code numbers in the range 128–255 in addition to the ASCII range, and this gives room for letters used in most other Western European languages as well. Thus, you can use a single character code, called Latin 1, even for a text containing a mixture of English, French, Spanish, and German, because these languages all use the Latin characters with relatively few additions.

However, you quickly run out of numbers if you try to cover too many languages within 256 characters. For this reason, different character codes were developed. For example, Latin 1 is for Western European languages, Latin 2 for several languages spoken in Central and Eastern Europe, and additional character codes exist for Greek, Cyrillic, Arabic, etc. When only one language is used, you can usually pick up a suitable character code and use it. In fact, someone probably did that for you when designing the particular computer system (including software) that you use. You may have used a particular character code for years without knowing anything about it.

Character codes that use only the code numbers from 0 to 255 are called *8-bit codes*, since such code numbers can be represented using 8 bits.

Things change when you need to combine languages in one document and the languages are fundamentally different in their use of characters. In an English-German or French-Spanish glossary, for example, you can use Latin 1. In English-Greek data, you can use one of the character codes developed for Greek, since these codes contain the ASCII characters. But what about French-Greek? That's not possible the same way, since the character codes discussed above do not support such a combination. A code either has Latin accented letters in the "upper half" (the range of 128–255), or it has Greek letters (α, β, γ, etc.) there. It would be impractical, and often impossible, to define 256-character codes for all the possible language combinations.

As you probably know, the number of characters needed for Chinese and Japanese is very large. They just would not fit into a set with only 256 characters. Therefore, dif-

ferent strategies are used. For example, 2 bytes (octets) instead of one might be used for one character. This would give 65,536 possible numbers for a character. On the other hand, the character codes developed for the needs of East Asian languages do not contain all the characters used in the world.

The solution to such problems, and many other problems in the world of growing information exchange, is the introduction of a character code that gives every character of every language a unique number. This number does not depend on the language used in the text, the font used to display the character, the software, the operating system, or the device. It is universal and kept unchanged. The range of possible numbers is set sufficiently high to cover all the current and future needs of all languages.

The solution is called *Unicode*, and it gives anyone the opportunity to say, "I want this character displayed and the number is..." and have herself understood by all systems that support Unicode. This does not always guarantee a success in displaying the character, due to lack of a suitable font, but such technical problems are manageable.

Much widely used software, including Microsoft Windows, Mac OS X, and Linux, has supported Unicode for years. However, to use Unicode, all the relevant components must be "Unicode enabled." For example, although Windows "knows Unicode," an application program used on a Windows system might not. Moreover, the display or printing of characters often fails since fonts (software for drawing characters) are still incomplete in covering the set of Unicode characters. This is changing as more complete fonts become available and as programs become more clever in their ability to use characters from different fonts.

Unicode Can Be Easy

Unicode is both very easy and very complicated. The fundamental principles are simple and natural, as the explanation above hopefully illustrated. The actual typing and viewing of Unicode characters can also be easy, when modern tools are used. As we get to complicated issues like sorting Unicode strings or controlling line breaking, you will find some challenges. But this book starts from simple principles and usage.

For example, an average PC running the Windows XP system has a universal tool for typing any Unicode character, assuming that it is contained in some font installed on the system. The tool is called the *Character Map*, or CharMap for short. Figure 1-1 shows the user interface of this program. The program can be launched from the Start menu, although you may need to look for it among "System tools" or something like that. You can select a collection of characters from a menu, and then click on a character to select it. The selected characters can be copied onto the clipboard with a single click, and you can then paste them (e.g., with Ctrl-V) where you like.

There are many other similar tools, often with advanced character search features. There are also ways to configure your keyboard on the fly so that keys and key combinations produce characters that you need frequently.

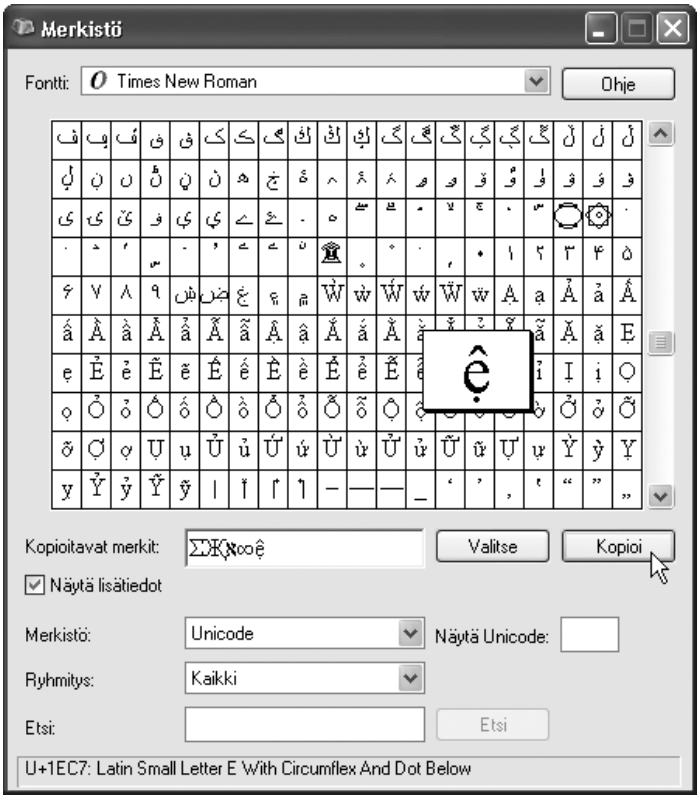


Figure 1-1. Character Map, part of Windows XP, lets you type any Unicode character

What's in a Character?

We use characters daily: we type them, and we read them on screen or on paper. We use text-processing programs routinely, much like people used to use typewriters, pens, or other writing tools. How could characters create problems?

Why Do We Need to Know About Characters?

If English is your native language, you are accustomed to using a small set of characters, consisting of the letters A–Z and a–z, digits 0–9, and a few punctuation characters. Most novels, newspaper articles, and memos contain no other characters. Since you seem to be able to type these characters directly on a keyboard, why should you learn more about characters and get confused? To be honest, character issues *are* confusing.

Suppose you use a computer only to write and edit texts in English, perhaps as a secretary or a technical editor. You still have reasons to know about characters:

- Computer technology has caused a decline in *typography*, and you can make a positive impression by using correct punctuation instead of typewriter-style punctuation. If you use a text-processing program, it probably takes care of using "smart" quotation marks instead of "straight" quotes, but you need to learn how to produce dashes—like this—and how to prevent bad line breaks.
- Normal English texts may contain special characters occasionally. Someone may spell Caesar as Cæsar, or use a word like fiancé, rôle, or garçon the French way, or use the per mille sign ‰ or the euro sign €. Michael Everson writes: "Despite unfounded but widespread belief to the contrary (based doubtless on the prevalence of ASCII), diacritics (usually French ones) are often found in naturalized English words. Examples are: à la carte, abbé, Ægean, archæology, belovèd, café, décor, détente, éclair, façade, fête, naïve, naïvety (but cf. non-naturalized naïveté), noël, œsophagus, résumé, vicuña" (*http://www.evertype.com/alphabets/englishpdf*). You may regard some of these spellings as foreign or obsolete, but people may still use them in English. There are often good reasons to change the spelling to something simpler, but not knowing how to produce the characters is not a good reason.
- Your text may contain *foreign names* with some strange characters. Although it is common to simplify the spelling, you can stand out positively by doing things correctly. Suppose that someone's surname is Hämäläinen and she works in an important international position. She is probably accustomed to seeing her name written as Hamalainen or Haemaelaeinen. But wouldn't she be delighted if someone were polite enough and competent enough to spell her name right, just for a change? However, she might not like it if someone tried to do so and failed, producing Hmlinen or H{m{l{inen.
- You might even be asked to include *quotations* in a foreign language. You might even need to work with a document in a foreign language, because someone has to do that and this is your day for being that someone. In that case, you may need to use foreign punctuation as well and to find a way to enter foreign characters efficiently, in addition to just knowing a universal clumsy way of entering any character.
- Texts increasingly contain technical and scientific special *notations*. Even casual memos and messages may need to mention µm (micrometer) or to use the almost equals sign ≈ or the male sign ♂. In scientific or technical texts, mathematical formulas are often quite crucial and need to be exactly right, down to the choice of each special symbol. The world is getting more technical and symbolic. Even nontechnical texts like bridge columns contain special symbols, such as ♠.

In *multilingual applications*, characters and their codes are a major issue. Even a web site with two or more languages or a bilingual dictionary can be regarded as multilingual applications, and they create the problem of representing the characters of both or all languages. For example, people using French and people using Russian on computers probably work with their own tools, settings, and conventions, but if you need to create

a document that is bilingual in French and Russian, you need to make sure you can work with both Latin letters with diacritic marks and Cyrillic letters. In effect, you would need to use Unicode, one way or another.

If you are a *computer professional*, you need to be prepared to handle data-processing problems that may involve characters of any kind. Someday someone will ask you to work with a system for processing data in a strange language or with strange symbols in it, perhaps even in a writing system where text runs right to left. It will be very difficult if you have no background in working with such issues. Most people need quite some time to digest character problems and techniques. You may find that, with something you thought you knew for years, you have completely misunderstood some basics.

Even if you process only "normal" text, character code standards and specifications are more important than they used to be. Modularity of software requires that you isolate character-level processing from other levels. You should not test for a character variable's value being equal to 32 to test whether it is a space character. Often, even a more sensible test, against the character constant `' '`, is suboptimal, and using a built-in function like `isspace` is better, since it takes care of other space-like characters as well. Tools developed for such operations are increasingly based on general specification in character standards, especially the Unicode standard. They are supposed to define, in a systematic and all-compassing way, the fundamental properties of characters, like being space-like, or being a letter, or allowing a line break before or after a character. To use such definitions and software modules that implement them, you don't need to know every detail, but you need to know the principles and the ways to get at the details when needed.

In addition, if you design or develop programs, databases, or systems, you will find that it is extremely difficult to adapt them to processing different character sets, if they were not designed to work that way. If the software is full of code that relies on using 1 byte (octet, 8-bit entity) for one character, it may need an almost complete rewrite if it needs to be modified to process Chinese text as well.

Characters as Units of Text

A character is a basic (or "atomic") unit of written text. A piece of text is a sequence of characters, also called a string. This does not necessarily mean that text is always displayed so that its characters appear linearly one after another, although this is what happens for English text, if we ignore the issue of division into lines. In other writing systems, consecutive characters may be combined into one glyph in complex ways. However, the text is still *logically* a sequence of characters.

Characters as abstractions

To store, process, and transfer data in digital form, we need an abstract concept of a character. It would not be feasible to store the specific appearance of each written character. Instead, we store information that tells which character it is, independent of

the specific visual shape it has. If we wish to affect the way in which our characters are displayed and printed, we use special formatting commands or other tools.

The abstract concept of character is essential in Unicode, in all digital processing of character data, and even in writing itself. The meaning of a piece of text does not change if you change its font, the specific design of its characters. To put it a bit differently, the style and taste—and even the effect—of text might change, but we have an intuitive understanding of something invariant behind such variation. For example, "A," "𝖠," "*A*," and "**A**" are instances of the same character. Since you know the Latin alphabet, you should have no difficulty with this. You might find it more difficult to know whether 𐤀 and א are instances of the same Hebrew character, but people who speak Hebrew are able to recognize that.

Different attempts have been made to describe what characters are. They have even been compared to Platonic forms. The point is that there is so much negative in the concept: it is largely defined by saying what a character is *not*. In a sense, we extract properties and concrete features, until there's very little left—something that could be called the idea of a particular character. Dan Connolly has written in his classical treatise "'Character Set' Considered Harmful": "Note that by the term character, we do not mean a glyph, a name, a phoneme, nor a bit combination. A character is simply an atomic unit of communication. It is typically a symbol whose various representations are understood to mean the same thing by a community of people."

This raises the question of what to do if different people recognize things differently. In some languages, "v" and "w" have been treated as typographic variants of a single character; other languages treat them as completely distinct letters. In such situations, Unicode normally defines separate characters.

To clarify the abstract nature of characters, a Unicode character, or a character defined by some other standard:

- Normally has no particular stylistic appearance but may vary between broad limits, as long as the designs can be recognized as the same character
- Is essentially black and white, though a character as a whole could be colored with any other two colors (making, for example, the ♥ character appear in red), using methods external to character standards
- Has an official name (as described later) but no fixed name across languages, and not necessarily any commonly known name in a particular language
- Has no fixed pronunciation, except for some specifically phonetic characters; however, there are of course correspondences between letters and sounds, even across languages that use the same basic writing system
- May have very specific usage as a special symbol (e.g., © is just a copyright symbol) or a broad range of different uses (e.g., / can be a separator of a kind, a mathematical operator, or something else)

Variation of appearance or different characters?

Problems arise when the concept of an abstract character has to be applied to concrete situations. We know what the letter "A" is, but is it the same as the lowercase letter "a"? That is, is the difference between them just variation in appearance, the same way as the letter "A" in the Times font differs from the letter "A" in the Arial font? In fact, the lowercase letters are a medieval invention, created by people who wrote text by hand and needed forms that are more convenient for that.

We could have defined "A" and "a" as just visual variants of the same abstract character, but we didn't. Quite early in the history of computers, this decision was made. It has far-reaching implications. If you wish to process input data so that upper- and lowercase letters are equivalent, to make things easier to people who type the data, you need to do something special to take care of that.

To take things a bit further, consider the Latin letter "A" and its relationship to the corresponding Cyrillic letter and the corresponding Greek letter, capital alpha. All three letters look the same in most fonts, and they share a common origin. Yet they belong to different alphabets: the Latin alphabet A, B, C, D..., which we use in English and many other languages, the Cyrillic alphabet А, Б, В, Г…, which is used in Russian and many Eastern European languages, and the Greek alphabet Α, Β, Γ, Δ... (alpha, beta, gamma, delta...).

It would have been possible to identify the Latin "A" and its Cyrillic and Greek counterparts. However, it was decided to keep them separate. Generally, Unicode (and character standards in general) do not unify characters across writing system boundaries. We might take this just as a fact of life and live with it. But we might also look at its reasonableness. Consider the operation of converting text from upper- to lowercase. The Latin letter "A" should become "a," whereas the Greek letter alpha "Α" should become α. It would be impossible to do this automatically if it were impossible to tell, from the internal digital representation, whether the original data contains the Latin "A" or the Greek "Α."

Writing systems were invented by people, and characters are creations of mankind, not nature. Thus, the identity of abstract characters is in a sense just a decision made by some people. However, it is usually an informed decision.

Variation in shape turned into a character difference

In many cases, stylistic variation in drawing or printing a character has been "frozen" so that a variant obtains a specific shape and meaning. The ancient Romans used the letter "V" both as a consonant and as vowel. Later, it appeared in different variants, such as a rounded one, like our "U." People started using the original version and different curved variants in different contexts. As such usage became systematic, consistent, and common, the letter "U" was born.

Therefore, we now have the independent characters “V” and “U.” They are, in turn, written with stylistic variation, though now the general idea is that the variation should not obscure the difference between these two characters. Yet, you might still see “V” used for “U” for stylistic reasons, especially to imitate ancient inscriptions (SENATVS POPVLVSQVE ROMANVS).

The letters “U” and “V” have later given birth to new characters that have originally been formed as their typographic variants, as well as the letter “W,” originally a digraph (VV). Special forms of this letter have been recognized as separate characters, such as the modifier letter small w, ʷ. The story goes on. In different areas that need new symbols, characters are created as variants or modifications of old characters. This seems to suit the human mind better than the invention of new character shapes from scratch.

Characters and “abstract characters”

The Unicode standard defines different meanings for the term *character*. The first one is: “The smallest component of written language that has semantic value; refers to the abstract meaning and/or shape, rather than a specific shape (see also *glyph*), though in code tables some form of visual representation is essential for the reader’s understanding.” The second meaning is that “character” is a synonym for “abstract character,” which is defined as “a unit of information used for the organization, control, or representation of textual data.”

Thus, the difference seems to be that an abstract character may have a control purpose only. Control purposes include line breaks, for example. In more common terminology, “character” in Unicode often means a printable (graphic) character, whereas “abstract character” means what is commonly called just “character,” which includes printable and control characters.

On the other hand, the Unicode standard also uses the expression “abstract character” to refer to a symbol that may be perceived by users as a character (“user character”), although it cannot be represented as a single Unicode character (also known as encoded character or coded character). In particular, a symbol with special marks (diacritic marks) on it, such as ó, cannot always be represented as one character in Unicode but may be a sequence of two or more characters.

The expression “semantic value” is somewhat misleading in this context. A character such as a letter can hardly be described as having a meaning (semantic value) in itself. It would be better to say that a character has a *recognized identity* and it may be sometimes used as meaningful in itself (as a symbol or as a one-letter word) but more often as a component of a string that has a meaning. Moreover, the “smallest component” part is somewhat vague. A character such as ú (letter u with an acute accent), which belongs to Unicode, can often be regarded as consisting of smaller components: a letter and a diacritic (acute accent). In fact, in Unicode, the character ú may be regarded *either* as a character on its own or as a combination: as two successive characters, letter “u” and a combining acute accent.

The intuitive concept of character varies by language and cultural background. If you know the letter ä mainly from J. R. R. Tolkien's books, you might regard it just as letter "a" with a special mark that indicates that it is to be pronounced separately. You might even regard the two dots just as optional decoration, as in "naïve" if spelled in the French way. If your native language were Finnish, you would certainly treat ä as a completely separate character, and you would have learned at school that it has its own position in alphabetic order (a, b, c,...x, y, z, å, ä, ö). Similarly, in Swedish, the words "här" ("here"), "har" ("has"), and "hår" ("hair") must be kept clearly separate. To a German, ä is different from "a," but it is treated as primarily equivalent to "a" in alphabetic order and is in a sense a variant of "a" ("a Umlaut").

Unicode, aiming at universality, generally recognizes written forms as separate characters, if at least one language or commonly used notation system makes a difference. Thus, "a" and ä are treated as distinct. If you wish to handle them as equivalent, you need to program code that treats them that way.

Characters and other units of text

Although a character is a natural "atom" of text in data processing, it does not always correspond to people's intuitive idea of the basic constituents of text. Looking at text in English, we might occasionally ask ourselves whether the ligature ﬁ is two characters or one. In other writing systems, similar questions arise more often. Unicode takes a liberal approach to identifying a complex character in many cases. You can represent ﬁ as one character or (more often) as two characters, "f" and "i." As mentioned above, similar principles apply to letters with diacritic marks.

People who speak languages with many diacritic marks or ligatures may regard a symbol like ﬁ or ú as a single character, even though they are often coded as sequences of characters. In some cases, it would not even be possible to code the symbol as a single character in Unicode, since Unicode does not contain all the combinations and ligatures that can be formed.

Moreover, although characters might be written separately, as in "ch," their combination might be understood as a single entity by some people. In English, "ch" denotes a particular sound and has thus some identity of its own. Some other languages treat the combination as an inseparable unit even in alphabetic order: in a dictionary, words would appear in an order like car, czar, char. Such treatment has become less common, though, since it is somewhat more difficult to implement in automated processing. Unicode treats "ch" as two characters but recognizes that it *might* constitute a unit in ordering.

Partly for such reasons, the ordering of characters is rather complex. Unicode does not prescribe a single ordering of characters and strings. Rather, it defines a basic (default) ordering that can be used as basis for defining language-dependent and even application-specific orderings.

Characters Versus Images

Characters are normally represented in graphic form, as something that can be called an image. However, there is a fundamental difference between an image and a character. An image can be a particular rendering of a character, much like a spoken word is a particular presentation of an element of a language. Moreover, most images are not renderings of characters at all.

Character code standards mostly identify a symbol as a character only if it is actually used in texts—e.g., in books, magazines, newspapers, and electronic documents. Characters that are normally used only in product labels and other specialized contexts are often borderline cases. However, they are often identified as characters if they are used in conjunction with symbols that are undeniably characters.

A typical example is the estimated symbol ℮, a stylized variant of the letter "e." It is not used in normal texts, but only in European packaging to claim conformance to certain standards in specifying a quantity. However, it is identified as a character, partly because it is used in packages in relation to text characters—e.g., in "℮ 200 g" (indicating that the mass of the product is 200 grams, within tolerances defined in specific regulation).

On the other hand, logos and identifying symbols are not treated as characters, even though they might be accompanied by texts. By its nature, a logo consists of a name or abbreviation in a particular graphic style. Hence, it would be unnatural to encode it as a character or sequence of characters, although we might use a string of characters as a *replacement* for a logo (e.g., when a document containing a logo needs to be converted to plain text and the logo conveys essential information).

Similarly, most of the various political, ideological, or religious symbols are treated as graphic symbols that are not characters. They are not normally used in texts. Their shape may vary, but not as part of font variation. However, for various reasons, some graphic symbols have been defined as characters in some character codes, contrary to these principles. Unicode therefore contains them as characters, so that existing texts using such characters can be encoded.

Generally, a graphic symbol is encoded as a character in Unicode, if there is need for exchanging it in digital form in plain text. Decisions on this are sometimes difficult and may be affected by tradition.

The distinction between a character and an image is often a practical decision to be made by the author or editor of a document. In many cases, you have a choice between a character and an image. For example, suppose that you are designing a user interface for a document, program, or web page and you need graphic symbols for "Next" and "Previous." It may often be best to use words, but let us assume that you want to use

arrows pointing to the left and to the right. Beware that even at this fairly abstract level, the decision is not culturally neutral: it implies left-to-right writing direction.

In Unicode, there is a largish block of arrow characters. Among them, a few like ← and → are widely available in commonly used fonts. However, they are not very prominent graphically, even if shown in bold, in large font, and in color. Their graphic design is character-like, not iconic. Some other characters in the Arrows block of Unicode look more solid, but they are not as common in fonts. For buttons or links, specially designed images may thus work better. On the other hand, in running texts, the arrow characters often work well. If you wish to make references to other entries in an encyclopedia by using arrows, then "→foobar" works better than a word preceded by a distinctive graphic.

Generally, when deciding between the use of characters and the use of an image for presenting a graphic symbol, the following items should be considered:

- Are there some Unicode characters that could be used, and are they suitable both by their defined semantics and by their typical graphic appearance?
- Is it possible that the document will be rendered so that images are not displayed? If yes, is it possible to specify a textual alternative to the image (such as the `alt` attribute in HTML markup)?
- How safely would the character work, given all the possible problems with encodings, fonts, etc.?
- Is it acceptable, and perhaps desirable, that the symbol changes size, shape, or color when text font size, face, or color is changed?
- Is it possible that the data will be processed as a character string—e.g., stored in a database or used in a search string?

For example, suppose we write about music and wish to refer to F-sharp and B-flat using the conventional musical symbols: F♯, B♭. The Unicode approach would use the special characters: music sharp sign ♯ and music flat sign ♭. However, these characters, although part of Unicode since Version 1.1, are poorly supported in fonts. Even though you could find them in some fonts at your disposal, their appearance might not fit into your typographic design. You might end up using the number sign # and the letter "b" as replacements. In web authoring for example, you might decide that although `B&#x266d;` would be technically quite correct (using a so-called character reference to include the flat sign), it is safer to create a small image, say *flat.gif*, and embed it with markup like `B<img src="flat.gif" alt="-flat">`. This means that the flat symbol remains in constant size if the text size is changed, but this is usually tolerable.

Sometimes character-looking symbols are not characters. Microsoft Word by default changes the three-character sequence "-->" into a kind of arrow symbol (à). However, this arrow is different from any Unicode character: it is just a glyph in the Wingdings font. It is therefore something between a character and an image; as so many compromises, it combines the drawbacks of the alternatives.

Processing of Characters

The previous discussion mentioned that characters can be processed and used in many ways that are not possible (or practical), if information is represented as images, sounds, or in another nontext format. This includes:

- Searching for occurrences of a word or other fragment of text, using either a simple search string or a text pattern
- Performing automatic replacements, such as substituting a string for another in all occurrences
- Indexing the data for efficiency of searching and for creating an alphabetic index or concordance (list of occurrences of words)
- Sorting text data—e.g., for presentation in alphabetic order
- Copying text from an application or data format to another, often via a clipboard
- Modifying text as in a text editor or text-processing application, by deleting, inserting, and replacing characters
- Selecting parts of text by user actions, such as painting or keyboard commands
- Recognizing constructs like words, syllables, morphemes (components of a word with a meaning), and sentences
- Computing statistics on the use of characters, words, phrases, etc.
- Spelling and grammar checks
- Automatic or computer-aided translation
- Presenting texts in audible form, via speech synthesis, which is more natural these days than you might expect from many science fiction films

Even the display of characters on screen or paper involves processing:

- Choice of font, which can be a complex process
- Application of bolding, italics, and other features, if requested
- Selection of contextual forms for characters
- Recognition of character sequences that should or could be rendered using ligatures or other special methods
- Formation of characters with diacritic marks, often requiring complex algorithms
- Adjusting spacing between characters and words, perhaps for justification of lines
- Breaking text into lines, perhaps using hyphenation

In particular, suppose that some document exists on paper only, or as a scanned image only. The above lists of possibilities can be consulted when estimating whether the text should be converted into text format. The conversion may require quite a bit of work, including the identification of special characters occurring in the documents.

Sometimes the benefits of text format turn into drawbacks, or they are regarded as problems. If you send a contract by email and ask the recipient to print, sign, and send it, can you be sure that he does not edit the text before printing, without your noticing? Ease of copying text can be a problem, if it is used to violate your copyright. For such reasons, plain text and even other text forms are sometimes avoided. Perhaps even a printing possibility is undesirable. Some data formats, such as PDF, can be locked, or protected against copying and modification and printing—though in a relative sense only.

Giving Identity to Characters

To represent characters in digital form, we need to encode them using bits, but first we need something to encode. We need a *collection of characters* that are distinguishable from each other. We do not define characters individually but as parts of a collection. The Latin letter "A" is defined, among other things, by designating it as distinct from lowercase "a" or from any Greek or Cyrillic letter.

A character is also described by its *meaning*, or semantics. However, we must be careful about this. A character is usually just an atom of text and normally lacks a meaning in the sense that words or some parts of words have meanings. In the word "singing," the stem "sing" and the suffix "-ing" have meanings, but it would not be natural to say that the letter "g" has a meaning, in any comparable sense.

The meaning of letter "g" is basically that it is one of the (lowercase) Latin letters, used to write words in some writing systems. Its pronunciation may vary (even within one language—compare "get" with "gem"), although it might be possible to indicate some typical phonetic values. Generally, definitions of letters in character standards are independent of pronunciation issues, except for some characters specifically designed for such usage (e.g., characters in the International Phonetic Alphabet, IPA).

As we get to more technical characters, such as the plus sign + or the copyright symbol © or the smiling face ☺, we find characters that can be described as having a meaning of their own. They might even correspond to words, such as "plus" and "copyright."

Definitions of characters in standards

The definition of a character in a standard needs to be unambiguous and definitive, not just loose prose. Old character standards tried avoiding the problem of definition by simply showing the character, assigning a number to it, and possibly naming it. This has turned out to be insufficient for many purposes. How could you tell from just seeing an "A" whether it is meant to be the Latin letter only, or also the Greek or Cyrillic letter?

The most important character standard in the modern world is *Unicode*, so let us take a look at its way of defining characters. Unicode identifies a character by:

- Showing a representative *glyph* for the character—i.e., one specific but typical visual form that the character may have

- Assigning a unique *number* to it; this number will never be changed
- Assigning a unique *Unicode name* for it; this will never be changed either, even if it is found misleading or originally mistyped, and it is best to regard it as a *mnemonic identifier* rather than a name in a normal sense
- Specifying a set of *properties* for it in a rigorous, formalized manner; they describe, for example, the general class (letter, digit, punctuation, etc.) of the character, its uppercase equivalent when applicable, etc.
- Making *annotations*—i.e., prose descriptions that clarify the meaning, often comparing the character with other characters, presenting alternate names for it, and sometimes even describing possible variation in the visual appearance

For example, the plus sign is defined in Unicode as follows:

- The representative glyph looks much like +.
- The number is 2B, often written as 002B for uniformity, in hexadecimal (base 16) notation, which means 43 in decimal (base 10).
- The name is PLUS SIGN.
- The general category is "Sm," which is short for "Symbol, Math." Line breaking is permitted after the character. There are several other formalized properties as well; we will discuss the various properties in detail in Chapter 5.
- There are no annotations for this character.

Annotations used to emphasize differences

The plus sign is not easily confused with any other character, and it has no widely used alternate names in English. Therefore, no annotations were deemed necessary. For the comma character "," character number 002C, for example, there is an annotation that says that the character has the alternative name "decimal separator." This does not mean that the decimal separator should be a comma (although most languages in fact use a comma for that). It just means that in some contexts some people call the comma "decimal separator." This effectively identifies a comma used as a decimal separator with the character number 002C, as opposed to treating it as a separate though similar character. On the other hand, the annotations related to the comma character also contain notes that refer to "Arabic comma," "single low-9 quotation mark," and "ideographic comma" as separate characters. This can be read as a warning against confusing the comma with those visually similar characters. For example, some languages use a single low-9 quotation mark as an opening quote in some contexts (e.g., in German: ‚gut'); without a warning, you might be inclined to think that it's just a special use for the comma.

The representative glyphs

The definitions of characters in Unicode are logical and do not imply any particular presentation of a character, either internally (in digital form, as bits) or visibly on paper or screen. However, a representative glyph is given to clarify the identity of a character.

The Unicode standard explicitly says that the representative glyph is not a prescriptive form of the character, but it lets a "knowledgeable user" recognize the character.

The glyphs used in Unicode code charts tend to be neutral and generic rather than typographically well-designed. They typically lack artistic ambitions, and they have been designed so that differences with other characters have been emphasized. That is, glyphs for characters that are often rather similar in practice, especially if we consider variation across fonts, have usually been designed to be sufficiently different from each other.

The number and the Unicode name as identifiers

The number assigned can be regarded as identification only, although in practice, it is used as a basis for the digital representation. The Unicode name is an alternative, more mnemonic identifier. As a mental exercise, consider the possibility of sending information by telephone so that you utter the names of Unicode characters, in order to express something complicated like a foreign word or a formula. If both participants have access to information about Unicode characters, the communication can be completely successful even though no visible characters are sent and no digital encoding is used.

Thus, when characters are represented in digital form, each character is internally a number, an integer. Numbers in turn are represented as sequences of bits, but this is a different level. When a file contains the string "Hello" (without the quotation marks), it really contains five numbers corresponding to the characters. In most character codes, this is the sequence 72, 101, 108, 108, 111.

A character code can assign numbers to characters arbitrarily, but once assigned in a specification, they should not be changed. In practice, the assignments have been made in a partly systematic way, so that related characters often have consecutive numbers.

Many modern standards, specifications, and instructions identify characters by their Unicode numbers to achieve unambiguity. Previously, documents on matters like mathematical or technical notations or transliteration of texts used to specify the symbols to be used just by showing them as visual forms, as ink on paper. This turned out to be particularly problematic in the computer era, when different people interpreted such signs differently, resulting in incompatible encoding of data.

Suppose that you specify, for example, that in some notation, the double prime character (″), with Unicode number 2033 in hexadecimal, be used (say, to denote seconds as a subdivision of a degree when expressing angles). Actually, the Unicode number alone would suffice, but mentioning the name makes the specification more readable.

In principle, you do not even need to write the character itself, though usually it helps. By identifying the Unicode number, you have achieved several things:

- You have unambiguously *identified* the character you mean. People may still decide to use some similar character instead, if they have difficulty typing the right character. Yet, it is clear which is the right character; others are various replacements.
- You have given a number that can be used as an *index* to large collections of information about the character, such as varying visual shapes for it, its defined properties, fonts containing it, definitions of meaning, and comments on scope of usage.
- The number can be used for *typing* the character by anyone who knows a general input method for Unicode characters in a particular environment. Typical word processors have at least one mechanism that produces a specific character, if you just specify its Unicode number.

Thus, anyone who participates in creating or clarifying notational specifications should know the principles of Unicode and should promote the use of Unicode numbers for characters. You should probably expect resistance, since it is not quite easy to see the benefits.

Unicode is more explicit

Older character standards, such as ASCII and the ISO 8859 family of standards, contain substantially less information about characters. They rely on the names of characters and the representative glyphs—and intuitive understanding related to the traditions of using characters. The same applies to the ISO 10646 standard, which is the official international standard that corresponds to Unicode. This means that we have two standards that are fully in accordance, ISO 10646 and the Unicode standard, but the latter contains a lot of additional information. Moreover, the Unicode standard is freely available on the World Wide Web, which is why people speak about Unicode and not ISO 10646, except in official standards and related documents.

The collection of all Unicode (or ISO 10646) characters is sometimes called the Universal Character Set (UCS). This expression is used especially in formal contexts, when one needs to refer to ISO 10646 and does not want to mention Unicode. In normal prose, we usually refer just to Unicode characters.

Spelling of names and the U+nnnn convention

The Unicode names of characters are written in all uppercase in the Unicode standard, but this is just a convention. In fact, the standard itself spells the names in all lowercase in some contexts. Uppercasing is often used to indicate (or hint) that a character is referred to by its Unicode name. However, in this book, we use normal (mixed) case for the names, except in some quotations.

We will use the conventional style of mentioning a Unicode character by its code number in hexadecimal (base 16) and prefixed with U+—e.g., U+002B. We could use just the number, but then you might not always know whether we use a number for such identification or just as a number.

This notation is used with at least four hexadecimal digits, so there are often leading zeros. All characters in the so-called Basic Multilingual Plane (BMP) can be expressed in four digits, but some newer characters need more.

We will normally mention first the Unicode name, then the code, often with a glyph between them. Thus, while you might see a Unicode character mentioned as U+002B PLUS SIGN in many sources, we will mostly say: the plus sign + U+002B.

Unicode Definitions of Characters

The definition of a character in Unicode is given partly in *code charts*, partly in the Unicode Database, which contains large tables of data on characters, by property, to be discussed in Chapter 5. Here we concentrate on the information in the code charts, which are available via *http://www.Unicode.org/charts/*. Each code chart begins with a table of glyphs, followed by notes on each character. The notes vary greatly in length and nature, but they should always be consulted when in doubt about the identity of character. Note that the code charts have been divided into two major groups, "Scripts" (which contains letters, ideographs, and other characters to write different human languages) and "Symbols and Punctuation." There is some overlap, since some blocks of characters belong to both groups.

The description of a character in a code chart consists of the following, where the first three items are given for every character (on one line), and others may or may not be present:

- Unicode number
- Representative glyph (in normal text size)
- Unicode name, in uppercase; this name is fixed
- Old (Unicode 1.0) name, in uppercase on a line of its own
- Other name(s), preceded by an equals sign = and written in lowercase; these names may be changed
- Comment(s) on usage, preceded by a bullet •
- Cross reference(s) to other characters, preceded by an arrow →; these references often warn against confusing a character with another, similar-looking character
- Information that specifies the character as a decomposable character, using a notation that begins with the symbol ≡ (indicating so-called canonical equivalence) or with the symbol ≈ (indicating weaker correspondence)

Figure 1-2 shows the description of the full stop (period) character in a code chart.

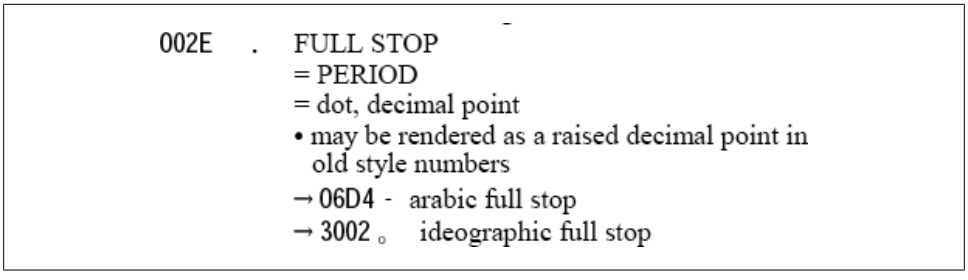


Figure 1-2. Sample description of a character in a Unicode code chart

Definitions of Characters Elsewhere

Characters were defined and used long before Unicode. Even in our times, characters are often used without identifying them with a reference to any character code standards. This creates ambiguity and potential diversity when text data is represented in computer-readable form.

For example, the standards that define the SI, the International System of Units (an extension of the metric system), use several special characters such as μ, ×, and Ω. The authoritative formats of the standards are printed documents, and since they do not specify code numbers or Unicode names for the characters, we are left in some uncertainty. Some characters can be identified rather unambiguously, but it is unclear what the "raised dot" character is, for example. This character, used in notations like N·m (for newton meter), is usually interpreted as the middle dot U+00B7, but it can be argued that a more appropriate interpretation is the dot operator U+22C5.

Similarly, the International Phonetic Alphabet (IPA) was originally defined about a century ago. When it later became relevant to use it on computers, the characters had to be identified as Unicode characters. This was far from trivial, since many IPA characters can be regarded as normal Latin letters, or treated as separate symbols.

Even relatively new standards on transliteration or transcription—i.e., on conversions between writing systems—fail to identify all characters unambiguously. For example, many standards and tables for writing Russian words in Latin letters specify that the so-called hard sign, ъ, is to be translated using a special character, but this character is just shown as a glyph on paper. This is subject to different interpretations including the ASCII quotation mark ", the right double quotation mark ", and the double prime ″ (U+2033). The Unicode standard makes, in a code chart, the following note about the modifier letter double prime ˝ (U+02BA): "transliteration of tverdyj znak (Cyrillic hard sign: no palatalization)." This might seem to resolve the issue in principle, but in practice, that character is not present in most fonts, and we can also ask whether the Unicode standard is authoritative in transliteration issues. Problems similar to this also exist for some apostrophe-like characters in transliteration systems for Arabic, for example.

What's in a Name?

The names of characters in character standards are assigned identifiers rather than definitions. This is particularly true for Unicode, which now has an absolute principle of name stability. A Unicode name will not be changed even if proved wrong.

Typically, the names are selected so that they contain only letters A–Z, spaces, and hyphens; often the uppercase variant is the reference spelling of a character name.

The same character may have different names in different definitions of character repertoires. Generally, the name is intended to suggest a generic meaning and scope of use. However, the Unicode standard warns (mentioning full stop "." as an example of a character with varying usage):

> A character may have a broader range of use than the most literal interpretation of its name might indicate; the coded representation, name, and representative glyph need to be taken in context when establishing the semantics of a character.

Although the Unicode names can be misleading—a price that we pay for their absolute stability—most of them aren't. The great majority of Unicode names describe the character, and the name is often the only description that the Unicode standard gives about a character individually. Thus, the name should be taken as describing the character, unless there is an annotation that says otherwise.

The Unicode name is in English, in a sense. In many cases, it is normal English, but often the name contains elements from other languages, such as the name in another language but as (somehow) adapted to English spelling.

For many purposes, it would be desirable to refer to characters by some widely understood names, in different languages. There will probably be a registry of such names, though mostly only for those characters that are widely used in each language. It will naturally contain English names as well, partly different for U.S. English and British English. They will of course have much similarity to the Unicode names. The naming is expected to take place in the context of Common Locale Data Repository (CLDR), discussed in Chapter 11.

Names of characters vary a lot, even within a language. This applies particularly to characters that are widely used in modern notations, but without much tradition, such as the tilde ~ or the commercial at @. Do not assume that people know from the name alone what you mean, even if you speak the same language.

The Unicode standard mentions some colloquial names for characters, even in languages other than English. For the @ character, it mentions that the "common, humorous German slang name" is "Klammeraffe," which means "clinging monkey." Undoubtedly, in some environments, the character might be better known under that name than under any official name. However, you need to be careful in using the al-

ternate names mentioned in the standard. It is better to look for information on actual usage in a language and a subculture. Slang, by its nature, varies by time and people.

When you need to refer to a character and cannot just show it, try to mention commonly known synonyms for it. It is not constructive to say just "use the reverse solidus." Instead, you can say "use the forward slash (that is, solidus), not the backslash (reverse solidus)." Unicode names alone are often rather useless in difficult situations for identifying characters to people who are not familiar with Unicode. The same applies even more to Unicode numbers.

Thus, you are not supposed to use the Unicode names for all characters in all contexts. If you are used to calling the "." character "period," you need not start calling it "full stop." You need not spell out "capital Latin letter A" every time you mention capital (uppercase) "A." However, the Unicode names appear in many contexts, like in character selection menus in editors, so you need to know the idea.

You may wonder why Unicode assigns two immutable identifiers for a character: a number and a name. If each of them is unique and guaranteed to remain unchanged, what do you need the other one for? The short answer is that numbers are the basic identifiers but names are needed too, since they have been used in programs and data to uniquely identify characters. Although it might not be wise to write code that operates on character names that way, it would be unwise to intentionally break all such code now.

Originally, names of characters were meant to act as identifiers across character codes. Different code may assign different numbers to the character ±, but they can be expected to assign the same name, "plus-minus sign," to it, or at least use names that can be recognized as essentially the same. However, this idea never worked well, since the names were in practice not always the same, or even essentially the same. Moreover, Unicode has made the original idea unnecessary, since nowadays the Unicode numbers are widely used to refer to characters across character codes, even when Unicode is not otherwise used for representing characters.

Should We Be Strict About the Meanings of Characters?

People tend to use characters on the basis of their visual appearance. You see a character like ß in some repertoire, and you start using it for the Greek letter beta, if you need it. You see the character ø and you take it as the diameter sign, so you use it in a technical context like "ø = 0.12 m" (saying that the diameter of something is 0.12 meters).

Unicode has strengthened such tendencies. People browse tables or menus of Unicode characters and pick up the first one that looks right for the purpose they have in their mind. Since Unicode has so many more characters than most old standards, there are far more opportunities for getting lost: it is easy to find a Unicode character that more or less looks like the one you need.

Then comes a purist and says that ß is a letter (sharp s) used in German, not any Greek letter, and that ø is a vowel used in some Nordic languages, not a mathematical symbol. Should we care?

Although you might realize the importance of using the right character, not just a right-looking character, you may need to explain the issue to others. Moreover, we often need to make compromises, and then it becomes essential to consider their impact. Reasons for using the right character translate into risks that you need to prepare for, when you cannot use the right character. So here are some basic reasons for being strict:

Some people see the difference
: Although the character looks right to you, a specialist may well see a difference between ß and β (sharp s versus small beta) or between ø and ⌀ (letter "o" with stroke versus diameter sign). When you write a foreign word, anyone who speaks that language as her native language is a specialist compared to you.

Font changes make differences noticeable
: When the font is changed, the difference can become clearly visible. A typical example is that the difference between degree sign ° (as in "50 °F" or "10 °C") and masculine ordinal indicator º (superscript letter "o," used in Spanish) is very small or nonexistent in many fonts, but very clear in many other fonts (e.g., ° versus º). Your text might be rendered in different fonts even though you have carefully selected a particular font. This is particularly true in web authoring and in cooperative authoring.

Conversions operate on characters, not appearance
: Automated editing of text is based on defined properties of characters, not on their appearance. For example, text-editing commands that operate on words will (or at least should) treat ø as a letter, not as a technical symbol. Converting text to uppercase would turn "ß-carotene" into "SS-CAROTENE," since "SS" is the defined uppercase version of ß.

Searching looks for characters, not appearance
: A search function in a program, as well as a database search, works on characters. When asked to find the string "β-carotene" (with beta), they will not find "ß-carotene" (with sharp s). The same applies to pattern matching and replace functions. Search routines may use some heuristics in their attempt to help users with common errors in using wrong characters, just as they may help with misspellings —as Google might say "did you mean pseudonym?" when you have typed "psuedonym." But don't rely on such features.

Automated processing generally ignores appearance
: For example, automatic speech synthesis and automatic translation, works on characters as abstract entities, not on their visual appearance. If your text contains "1°", meant to mean "one degree" but incorrectly uses a masculine ordinal indicator, it might be spelled out as "primero" (Spanish word for "first" in masculine gender). Similarly, it might be translated incorrectly.

Sometimes these considerations do not matter, or—more often—they need to be suppressed in favor of other needs. If you only aim at producing a document to be distributed on paper and you have full control up to and including the print operation, then the appearance is all that matters. But more often than not, documents are stored and sent in digital form. Then you may need to take precautions against wrong processing, perhaps document what you have done, and check things after various conversions and other operations.

Characters differ in the definiteness of their meaning. Some well-known characters like the hyphen - (known formally as hyphen-minus in Unicode) have a wide range of uses, and you may need to use them liberally. Computer programs need to be prepared for handling them accordingly. But other characters have specific semantics. The letter ø and the technical symbol ⌀ have limited uses. They should not be confused with each other or used for other purposes without careful consideration.

Ambiguity Among Characters

The identity of characters is defined by the definition of a character repertoire. Thus, it is not an absolute concept but relative to the repertoire; some repertoire might contain a character with mixed usage while another defines distinct characters for the different uses. For instance, the ASCII repertoire has a character called "hyphen." It is also used as a minus sign, as well as a substitute for a dash, since ASCII contains no dashes. Thus, that ASCII character is a generic, multipurpose character, and one can say that in ASCII, hyphen and minus are identical. But in Unicode, there are distinct characters named "hyphen" and "minus sign" (as well as different dash characters). For compatibility, the old ASCII character is preserved in Unicode, too (in the old code position, with the name hyphen-minus).

Similarly, as a matter of definition, Unicode defines characters for micro sign, n-ary product, etc., as distinct from the Greek letters (small mu, capital pi, etc.) from which they originate. This is a logical distinction and does not necessarily imply that different glyphs are used. The distinction is important, for example, when textual data in digital form is processed by a program (which "sees" the code values, through some encoding, and not the glyphs at all). Note that Unicode does not make any distinction, for example, between the Greek small letter pi (π), and the mathematical symbol pi denoting the well-known constant 3.14159... (i.e., there is no separate symbol for the latter). For the ohm sign (Ω), there is a specific character (in the Symbols Area), but it is defined as being canonical equivalent to Greek capital letter omega (Ω)—i.e., there are two separate characters but they are equivalent. On the other hand, Unicode makes a distinction between Greek capital letter pi (Π) and the mathematical symbol n-ary product (∏), so that they are not equivalent.

If you think this doesn't sound quite logical, you are not the only one to think so. The point is that for symbols resembling Greek letters and used in various contexts, there are three possibilities in Unicode:

- The symbol is regarded as identical to the Greek letter (just as its particular usage).
- The symbol is included as a separate character, but it is defined as equivalent to the Greek letter. There are two kinds of equivalence: canonical and compatibility.
- The symbol is regarded as a completely separate character.

You need to check the Unicode references for information about each individual symbol. As a rough rule of thumb about symbols looking like Greek letters, mathematical operators (like summation) exist as independent characters whereas symbols of quantities and units (like pi and ohm) are identical to Greek letters or equivalent to them.

How Do I Find My Character?

Suppose you have been requested to convert some printed or handwritten text into a digital format. (At the end of this chapter, we have such an exercise.) For English text with no special characters, you might be able to use a scanner. But what would you do with characters that the scanner does not recognize reliably?

Such problems are fairly common. For example, you might need to check the spelling of a foreign name from a printed reference book, or you might need to quote some printed material. Even standards on various notations often fail to specify the characters unambiguously: the authoritative format of a standard is usually a printed publication, and all you have got there is ink on paper, glyphs.

The recognition of a character from its glyph can be quite difficult, and it may require both factual and cultural knowledge about the subject area and the text. You also need technical information on character standards, since you ultimately need to identify glyphs as appearances of characters defined in the standards.

Looking for characters through lists or code charts is a rather hopeless task. The amount of characters is huge, and many characters look very similar to each other. For example, how can you know whether a glyph on paper is letter "a" with a caron (ǎ) or letter "a" with a breve (ă)? Thus, you first need some information or guess on the nature of a character. If you know or suspect that the character appears in a Romanian name, you have a good starting point, since the character repertoire used in Romanian can be found in a suitable reference. Similarly, if you know that a glyph like ₣ is a currency symbol, you have almost identified it.

The following list suggests some general online resources for identifying characters:

"Where is my Character?" (http://www.Unicode.org/standard/where/)
: An explanatory document by the Unicode Consortium. It explains some problems caused by the variation of shapes of characters.

Unicode Code Charts (http://www.Unicode.org/charts/)
: This is official information and covers all Unicode characters. It is organized first by division into "Scripts" (writing systems for human languages, containing letters, syllables, and word signs) and "Symbols and Punctuation." These parts are further

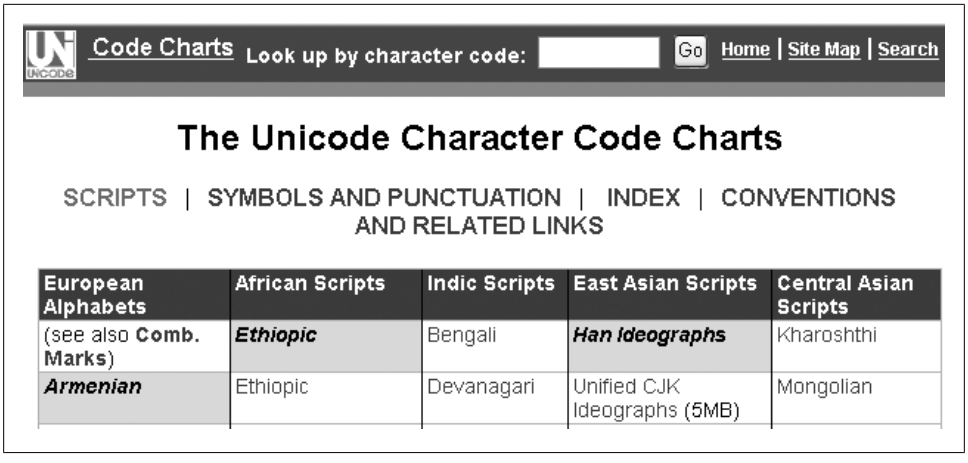


Figure 1-3. Part of the interface to online Unicode code charts

divided into large categories such as "European Alphabets." Figure 1-3 illustrates the appearance of the main page of the Code Charts.

Fileformat.info, section Unicode (http://www.fileformat.info/info/Unicode/)
: This contains data taken from the Unicode site and organized for viewing in different ways. It also contains information on Unicode support in different fonts. As you get down to information on individual characters, their properties are displayed in a compact format, which is great when you are ready to use it.

Database of characters at the EKI (http://www.eki.ee/letter/)
: Although not as exhaustive in character repertoire as the above, this database lets you search for characters in a few ways and shows some essential extra information on usage: it lists languages that use a character and character encodings (charsets) that contain it. Although these lists are not complete, they are often helpful. For example, they tell that letter "a" with a caron (ǎ, U+01CE) is used in Yoruba and in Romanization of Bulgarian and Chinese, whereas the letter "a" with a breve (ă, U+0103) is used in Romanian and Vietnamese and Romanization of Khmer, as shown in Figure 1-4. However, the information is not always completely reliable; in particular, the character used when writing Bulgarian as Romanized—i.e., in Latin letters—is not "a" with a caron but "a" with a breve, according to standards.

Which Characters Does Each Language Use?

For details on the use of characters in different languages, you need to consult grammar guides and textbooks on the languages themselves. However, there is an extensive compilation of basic information in *The World's Writing Systems* by Peter T. Daniels and William Bright (Oxford University Press). There is brief description of character usage in a few languages in *The Chicago Manual of Style*, 15th Edition (The University of Chicago Press). Online, you can find "The Alphabets of Europe," by Michael Ever-

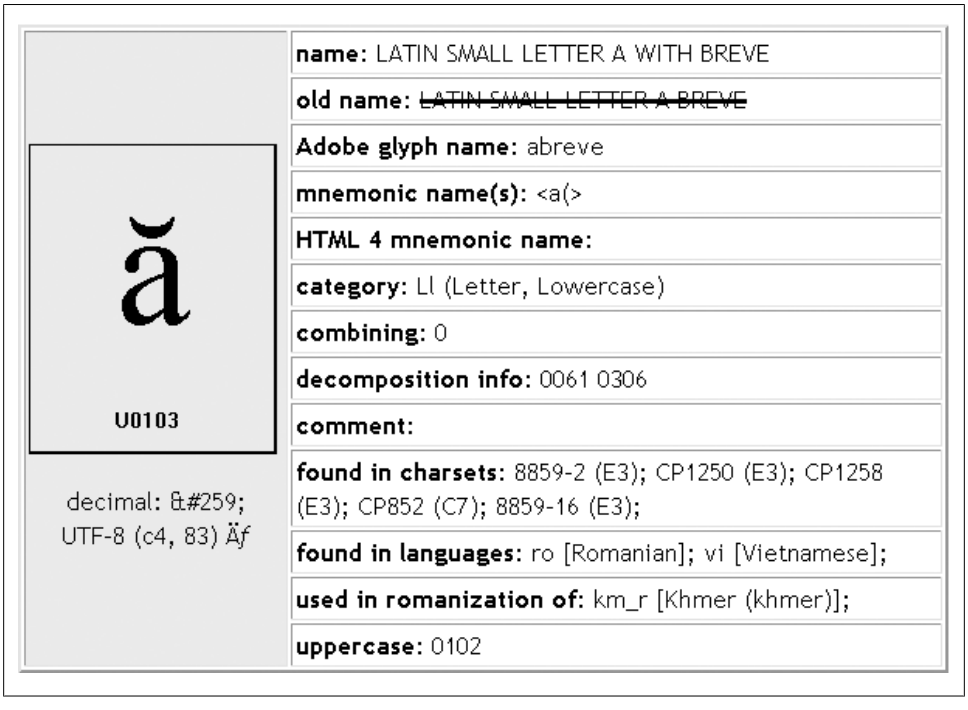


Figure 1-4. Sample information on a character in the eki.ee database

son, at *http://www.evertype.com/alphabets/*. It is extensive and based on detailed research, although it partly applies different criteria to different languages: for some languages, it includes only the basic modern alphabet; for others, it lists historical characters and other characters that are not used in normal writing. The CLDR database, discussed in Chapter 11, contains information on the use of letters in different languages.

Variation of Writing Systems

The most widely used writing systems, or scripts, can be classified as follows:

Alphabetic scripts
: Denote sounds with letters, though usually not in a strict one-to-one manner. Examples: Latin, Greek, and Cyrillic scripts, each of which exists in different versions.

Consonant scripts, or abjads
: Basically denote consonants, leaving vowels to be inferred; however, consonant scripts may have letters for long vowels, and in some situations even short vowels are written using small signs attached to consonants. Examples: Hebrew and Arabic scripts.

Abugida scripts
: These use consonant letters that imply a particular vowel after the consonant, when used in the base form. Alternatives with other vowels or without any vowel are indicated by additional marks. Many South and Southeast Asian scripts belong to this category—e.g., the Devanagari script used for many Indic languages.

Syllabic scripts
: Use basically one character for each syllable. Examples: the Hiragana and Katakana scripts, used for Japanese.

Ideographic scripts
: Use basically one character for one (short) word. The most widely known ideographic script is Han, often known as Chinese script, though it is also used (in part) for other languages as well, especially Japanese and Korean, and therefore often called "CJK."

Consonantal writing may sound impossible, because it introduces so much ambiguity. However, although an individual written form of a word is often ambiguous, the ambiguities are usually resolved easily from the context by a person who understands the language well. Moreover, languages written with a consonantal script typically have a structure that makes this easier than for English, for example. When vowels are mainly used to express variations of a common theme expressed by a word root, consisting of a pattern described by a combination of consonants, the vowels can usually be inferred from the grammatical context.

The word "script" is often used in character code contexts instead of "writing system." It is important to distinguish it from the use of the word "script" to denote a programming concept—a certain type of a computer program, such as a Perl script.

Some scripts, such as the Latin script, are written with spaces between words, and a space is normally a permissible line break point. Hyphenation may introduce other break points. Other scripts may permit line breaks more freely.

The Latin script and many other scripts are written left to right, with lines proceeding from top to bottom. These are not universal properties of human writing, and even the Latin script is historically based on a script that was written right to left. Unicode addresses the problem of left-to-right versus right-to-left writing in two ways: by defining inherent directionality for characters and by defining control characters for affecting writing direction. For example, Hebrew and Arabic letters have inherent right-to-left directionality. Special methods are needed when text in such letters contains names or quotations that have the opposite directionality, or vice versa.

In Latin scripts, each character is normally displayed as a separate image on screen or paper, though the spacing between characters may vary. In other scripts, the formatting of texts for visual presentation can be essentially more difficult: the shape of a character may depend on context; adjacent characters can be written together (using a ligature or using cursive writing where characters join smoothly); and a character might be displayed as an auxiliary symbol above, below, before, or behind another character.

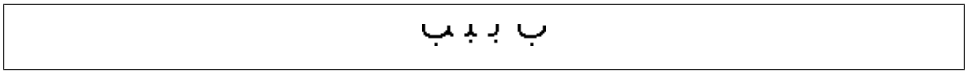


Figure 1-5. The four contextual forms of the Arabic letter "ba"

Glyphs and Fonts

It is important to distinguish the character concept from the glyph concept. A glyph is a presentation of a particular shape a character may have when rendered or displayed. It has even been said that any character is an abstract idea, whereas glyphs for the character are its different visible manifestations.

Each character we use in English normally has the same basic shape, and glyphs for it differ in typographic design only. It is obvious that "T" in the Times font represents the same character as "T" in the Arial font, for example. However, the letter "a" has two rather different shapes (compare "a" in normal Times font and "*a*" in Times italic). When you write literally by hand, you may draw characters differently in different positions of a word. For example, a word-final "s" may be quite different than a word-initial "s." In typewritten or typeset text, or in text displayed or printed on computers, such distinctions are not made, even in so-called handwriting-style fonts.

In Greek writing, a word-final sigma (ς) is rather different from a normal small sigma (σ), although they are logically the same character. The first and last letter of the word σοφός (sophos, "wise") are the same but are written differently. However, since this is a special case, character codes usually solve this by encoding them as two separate characters, and Unicode follows suit, even without defining any equivalence between them.

In other writing systems, the variation can be much bigger, especially if the writing systems imitate handwriting. In Arabic, letters have two or four *contextual forms*, which can be quite different from each other. Figure 1-5 shows the four forms of an Arabic letter, usually called "ba" or more exactly bā', though the Unicode name is Arabic letter beh (U+02BE). The forms are (from right to left!) for use as isolated, at the start of a word, in the middle of a word, and at the end of a word. As you can see, for example, the word-final form (on the left) has a part that helps in joining the character with the previous character. Each of these forms, in turn, can appear differently in different fonts.

In the ISO-8859-6 character code (Latin/Arabic), for example, each Arabic letter has one code position only. This leaves it to rendering engines to determine the context (position within a word) and to use the correct contextual form. Unicode, on the other hand, contains both such characters (effectively, taken from ISO-8859-6) and each of the contextual forms as a separately coded character. This lets you write Arabic so that the rendering process can be very simple, at the cost of extra work in writing. However, even using Unicode, you are normally supposed to use the more abstract Arabic letters.

It is ultimately a matter of definition whether two graphic presentations are glyphs for the same character or distinct characters. However, it is normally not an individual's

decision but a collective agreement. The definition of a character repertoire specifies the "identity" of characters, among other things. One could define a repertoire where uppercase "Z" and lowercase "z" are just two glyphs for the same character. On the other hand, one could define that italic "Z" is a character different from normal "Z," not just a different glyph for it.

In fact, in Unicode for example there are several characters that could be regarded as typographic variants of letters only, but for various reasons, Unicode defines them as separate characters. For example, mathematicians use a variant of letter "N" to denote the set of natural numbers (0, 1, 2,...), and this variant is defined as being a separate character (double-struck capital N, ℕ, U+2115) in Unicode.

The design of glyphs has several aspects, both practical and esthetic. For a review of a major company's description of its principles and practices, see Microsoft's "Character design standards" on its typography pages at *http://www.microsoft.com/typography/*.

Some discussions, such as ISO 9541-1 and ISO/EC TR 15285, make a further distinction between "glyph image," which is an actual appearance of a glyph, and "glyph," which is a more abstract notion. In such an approach, "glyph" is close to the concept of "character," except that a glyph may present a combination of several characters. Thus, in that approach, the characters "f" and "i" might be represented using an abstract glyph that combines the two characters into a ligature ﬁ, which itself might have different physical manifestations. Such approaches need to be treated as different from the issue of treating ligatures as (compatibility) characters.

Allowed Variation of Glyphs

When a character repertoire is defined (e.g., in a standard), some particular glyph is often used to describe the appearance of each character, but this should be taken as an example only. The Unicode standard specifically says the glyphs used for a character can be quite different from the "representative glyph," but within cultural conventions:

> Consistency with the representative glyph does not require that the images be identical or even graphically similar; rather, it means that both images are generally recognized to be representations of the same character. Representing the character U+0061 Latin small letter a by the glyph "X" would violate its character identity.

Thus, the definition of a character repertoire is not a matter of just listing glyphs. In fact, it's the exception rather than the rule that a character repertoire definition explicitly says something about the meaning and use of a character. For example, the description of the dollar sign $ says that the character may have one or two vertical bars, to make it clear that such variation does not change the character's identity. On the other hand, the pound sign £ has one crossbar, in contrast with the lira sign ₤, which is identified as a separate character.

Fonts and Their Properties

A font contains a repertoire of glyphs. In a more technical sense, as the implementation of a font, a font is an organized set of glyphs. The glyphs may have names that identify them; this is the way used in PostScript fonts. More often, glyphs are identified by their numbers, which typically correspond to code positions of the characters (presented by the glyphs). Thus, a font in that sense is character-code dependent. An expression like *Unicode font* refers to such issues of basic structure and does not imply that the font contains glyphs for all Unicode characters. In fact, such comprehensive fonts are very rare at present.

A font may contain the same glyph for distinct characters. For example, although characters such as Latin uppercase "A," Cyrillic uppercase "A," and Greek uppercase alpha are regarded as distinct characters (with distinct code values) in Unicode, a font might contain just one "A" that is used to present all of them. In fact, this applies to most fonts. On the other hand, a font may contain alternative glyphs for a character, for use in different contexts.

Fonts have names, which are often trademarks. The name of a font can be a single word like "Times" or it may consist of two or more words, such as "Times New Roman." It is not uncommon to see fonts that are very similar to each other but have completely different names such as "Helvetica" and "Arial."

Fonts can be classified in many ways, and this belongs to typography rather than our topic. However, some basic classifications as indicated in Table 1-1 are relevant for our purposes, since they appear in program settings for selecting fonts for displaying characters. For example, a program may have one choice of a font for serif font, another choice for sans serif font. These font classes are distinguished by the presence or absence (in French, "sans" means "without") of short strokes that terminate the lines of many letters. Usually there is also the difference that in a sans serif font, the lines of letters have (almost) equal thickness, whereas in a serif font, the thickness varies (e.g., the vertical line of "T" is thicker than the horizontal line).

Table 1-1. Some basic classes of fonts

Class of fonts	Characteristics	Sample font(s)
Serif	Widely used for copy text in books	Times, Georgia
Sans serif	Often used on screen and for small print	Arial, Verdana
Monospace	Equal-width characters, often used for code	Courier New
Cursive	Letters join to each other as in handwriting	Cooper BlkIt BT
Fantasy	Exotic, artistic (font)	Comic Sans MS

The attribute "proportional" refers to any font where the width of character varies, as opposed to monospace fonts. Monospace fonts are often used for computer code, and sometimes to imitate old typewriter text. To be exact, there can be variation in width

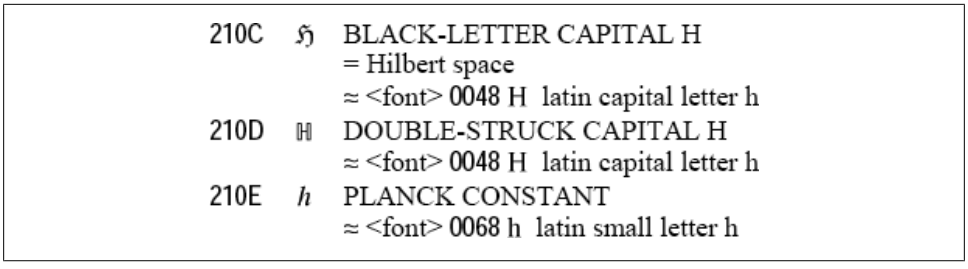


Figure 1-6. Some descriptions of characters in the Letterlike Symbols block in the Unicode standard

even in a monospace font: some Unicode characters are defined to be invisible, so they need to have a width of zero, and some characters such as fixed-width spaces have a specific width by definition.

There are many online services for viewing samples of fonts and for identifying the font of some text you have seen. They often tell how to download or buy the fonts, too. See, for example, *http://www.identifont.com* and *http://www.linotype.com*.

Typographers often use the term *typeface* to denote the basic design of glyphs, reserving the word "font" for particular implementations and variants. For example, the Times typeface is available as normal (regular), as bold, as italic, and bold italic, as well as in different sizes. Variants of a typeface in different sizes may differ in their details—i.e., they are not just formed from a basic size by simple scaling.

Font Variation Versus Characters

As mentioned above, variants such as normal, bold, and italic do not normally constitute a character difference. That is, a normal "A" is the same character as a bold "**A**" or an italic "*A*." Neither does changing the typeface change the identity of a character, as a rule. However, some Unicode characters have been defined essentially as variants of other characters, although this difference could have been made at the font level only. Such characters are defined in the Unicode standard as having *compatibility decompositions*, using notations as in Figure 1-6. The symbol ≈ stands for compatibility equivalence, and <font> indicates font variation—similar to what you could achieve using the `font` element in HTML, but here <font> is just a general notation, not markup. The <font> notation in the Unicode standard does not specify what kind of a font is to be used, and as you can see from the descriptions of U+210C and U+210D, <font> can mean quite different things for different characters. For example, U+210E is essentially "h" in italics, but in the Unicode standard, this is just implicit in its representative glyph.

Fonts in Implementations

The implementation of fonts is relevant to our topic, since it affects the practical availability of characters. If a character is only available in a font that is poorly implemented,

we may need to look for other approaches. For example, high-quality printing may require the use of certain font technologies.

The most important font technologies at present are:

Bitmap fonts
: Also known as raster fonts, system fonts, or screen fonts, these fonts essentially present a character as a matrix or raster of pixels, or bits indicating the presence or absence of a pixel. Bitmap fonts are more or less obsolete, though they are still used as "system fonts," often in window titles and dialog boxes.

PostScript Type 1
: This technology, developed by Adobe, is widely used in the print industry and in desktop publishing. On your PC, you may find Type 1 fonts, too.

TrueType
: This technology was developed by Apple, and then licensed to Microsoft. Probably most fonts on your PC are TrueType fonts (with filenames ending in *.ttf*).

OpenType
: This is a new technology developed jointly by Microsoft and Adobe. It is Unicode oriented and more platform-independent than older technologies.

Fonts other than bitmap fonts are effectively computer programs of a kind, controlling the drawing of lines that constitute a glyph. Fonts are generally protected by copyright laws, although the scope and terms of protection vary by country.

If you use Windows, you will probably benefit from downloading and installing the software from *http://www.microsoft.com/typography/TrueTypeProperty21.mspx*, "Font properties extension." It enhances the functionality of Windows so that when you open the Fonts folder (via Start → Control Panel), you can right-click on the icon of a font file and select Properties to get rather detailed information on the font. However, the amount of information depends on the technology of the font. Figure 1-7 shows some properties of a TrueType font. The properties include the ranges of Unicode characters that the font supports. Beware, however, that such support is not always exhaustive; it may lack some characters of the range, especially if the Unicode standard has been extended since the creation of the font. (The figure contains some Finnish words, too. Such things may happen if you install a program that uses English on an operating system that uses a different language in its interface.)

Failures to Display a Character

In addition to the fact that the appearance of a character may vary, it is quite possible that some program fails to display a character at all. Perhaps the program cannot interpret the character encoding of the data, either because it was not properly informed about the encoding or because it has not been programmed to handle the particular encoding.

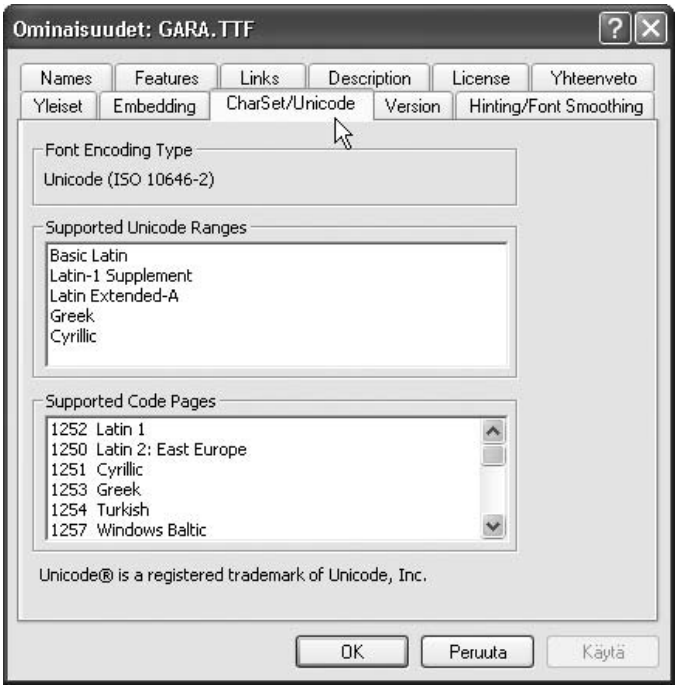


Figure 1-7. Properties of a font (Garamond), as viewed with the Font properties extension

Even if a program recognizes some data as denoting a character, it may well be unable to display it since it lacks a glyph for it. Often it will help if the user manually checks the font settings, perhaps trying to find a rich enough font. Advanced programs could be expected to do this automatically and even to pick up glyphs from different fonts, but such expectations are often unrealistic at present. However, it is quite possible that no such font can be found. As an important detail, the possibility of seeing, for example, Greek characters on some Windows systems depends on whether "multilingual support" has been installed.

A well-designed program will in some appropriate way indicate its inability to display a character. For example, a small rectangular box, the size of a character, could be used to indicate that there is a character that was recognized but cannot be displayed. Some programs use a question mark, but this is risky—how is the reader expected to distinguish such usage from the real "?" character? Advanced browsers may display a symbol that indicates the general class (e.g., Latin letter or mathematical symbol) of the character.

Font Embedding

To overcome a situation in which a recipient of a document might not have a font needed for the characters in it, techniques have been developed for embedding fonts into documents themselves. This is quite different from what word processors normally do with fonts: they include information about fonts (by font name), not fonts themselves.

Font embedding does not normally mean the inclusion of an entire font but only an extract from the font data, as needed for a particular document. The technique may prevent the recipient from using the embedded font for anything but viewing the particular document. This makes font designers more willing to allow embedding.

Another reason for font embedding is the desire to have a document presented exactly as designed. If you create a document using fonts that you like and send it, the recipient's program may well be capable of displaying all the characters but by using different fonts, in part. Usually if you specify a font that is not present in the recipient's system, the program used for viewing the document will use its default font instead. This might be regarded as a serious problem especially by visual designers.

The Font properties extension that was illustrated in Figure 1-7 gives access to information about font embedding possibilities, in the Embedding pane. If embedding is allowed for a TrueType font, you can, for example, set Microsoft Word to embed the font. For this, you would select Tools → Settings → Save, and then check the box about font embedding. Remember to reset this setting after saving the document, since otherwise Word will keep embedding all TrueType fonts, which is generally unnecessary.

For the Web, Microsoft has developed the Web Embedding Fonts Tool (WEFT) for use with HTML and CSS. However, it has not gained much popularity, partly due to its relative complexity. Instead, the usual approach is to use the PDF format, since common PDF creation tools allow easy font embedding. In addition to commercial products such as Adobe Acrobat, there are free tools like PDFCreator, which adds a "virtual printer" to your system. You can then use the Print command in various programs to generate a PDF version of a document, and in this context, you can check settings that make the tool embed the fonts you have used.

Font embedding has its drawbacks, too. Often it would be desirable for the user to change the font for legibility, but font embedding has more or less been designed to prevent this. A special character may look odd to a user, who might well recognize it if he could view it using some font he knows well. The PDF format does not allow easy font resizing, which would be crucial to many people. Therefore, it is best to distribute your material in alternative formats in accordance with recipients' choices, such as Microsoft Word, RTF, HTML, or PDF.

Definitions of Character Repertoires

The implementation of Unicode support is a long and mostly gradual process. Unicode can be supported by programs on any operating systems, although some systems may allow much easier implementation than others; this mainly depends on whether the system uses Unicode internally so that support to Unicode is built in.

Even in circumstances where Unicode is supported in principle, the support usually does not cover all Unicode characters. For example, an available font may cover some part of Unicode that is only practically important in some area. When text data produced in one program is to be processed in another, we should be prepared for difficulties with any unusual characters. For data transfer, it is essential to know which Unicode characters the recipient is able to handle.

Thus, although Unicode contains a huge number of characters, not all of them can be used safely. Among the 100,000 or so characters, usually only a small subset can be used in a particular application and context without a serious risk of distorting information.

Formally Defined Repertoires

Each character code, by itself, defines a character repertoire: the collection of characters that can be represented in the code. In addition to this, subsets of such collections can be defined.

A character repertoire is any collection of characters, without implying any particular implementation even at the level of code numbers. However, in practice, the simplest way to define a character repertoire is to use Unicode as the basis and simply list the code numbers. Such a definition specifies a *closed collection*, which does not change if the Unicode standard is enhanced. In contrast, by listing a set of Unicode blocks you define an *open collection*, which is fixed at any given moment of time but will automatically expand if new characters are added to any of those blocks in a revision of the Unicode standard.

For example, there are three Multilingual European Subsets (MES-1, MES-2, MES-3), defined in a CEN Workshop Agreement, CWA 13873. Among them, MES-2 is the most important. It is a closed collection, covering Latin, Greek, and Cyrillic scripts. The CWA is available at *http://www.evertype.com/standards/iso10646/pdf/cwa13873.pdf* or via *http://www.cenorm.be/cenorm/businessdomains/businessdomains/isss/cwa/*.

Practical Repertoires

In addition to international standards, there are company policies that define various subsets of the character repertoire. A practically important one, especially in regards to support in widely used fonts, is Microsoft's "Windows Glyph List 4" (WGL4), also known as "PanEuropean" character set, listed on the page "Using special characters

from Windows Glyph List 4 (WGL4) in HTML" at *http://www.alanwood.net/demos/wgl4.html*. Contrary to what you might expect, the characters in it have not all been included in MES-2.

In data-processing contexts, a character can be considered "safe" if it is certain or very probable that it will be correctly transmitted and presented to the recipient. In a broader sense, being safe entails more: the sender should be sure of the character he means, and the recipient should understand it correctly. Mostly, however, we consider the technical problems: difficulties in presenting the character in a digital form, in sending it over network connections and possibly to a different program and operating environment, and in rendering it visually. Nowadays, it's usually the last phase that poses most problems.

From a practical point of view, we can distinguish the following repertoires of characters. Each repertoire listed here contains all the previous repertoires. The list can be useful when you design an application, or instructions on writing things, or a computer language. When selecting which repertoire you use or support, it is advisable to proceed slowly in the list and consider whether the usefulness of extra characters outweighs the risks. The names used for the repertoires here are practical descriptions, not official names. They make liberal use of encoding names, which will be described in more detail in Chapter 3.

ASCII name characters: English letters A–Z and a–z and digits 0–9
: These are the safest characters and often the only characters you can use in names or *identifiers* in a computer language. Often a few extra characters like underline _, hyphen - and full stop "." are allowed, too. Be careful with any extra characters when selecting a name for a file, a username, or a data item name. The naming rules you have learned in some context may not apply in others. For example, Unicode names for characters use just letters A–Z without case distinction, digits 0–9, space, and hyphen (hyphen-minus, to be exact).

*The invariant subset of ASCII: the above, plus characters ! " % & ' ( ) * + , - . / : ; < = > ? and the space character*
: This can be described as the rock-bottom repertoire of characters in data processing. However, in different transfer and transformations, even these characters may get changed somehow. A common example is the ampersand &, which often needs to be written in some special way (e.g., as `&` in HTML and XML).

The full ASCII repertoire: the above, plus characters # $ @ [\ ] ^ _ ` { | } and ~
: This repertoire, called Basic Latin in Unicode, usually works well across programs, computer platforms, and network connections. The characters listed here work mostly just as well as the other ASCII characters, but some standards allow national variation that may make them unsafe. Moreover, producing some of these characters can be a nontrivial task on a non-U.S. keyboard.

The ISO Latin 1 repertoire consists of the above plus 96 additional characters, such as à, é, Ô, £, §, µ, ©, and ¥

This repertoire is also called ISO 8859-1, and it will be described in more detail in Chapter 3 and Chapter 8. It is sufficient for writing most Western European languages, except for some typographic issues. It is widely available in the Western world, but not necessarily elsewhere. Some characters in it still cause problems to some Mac users.

The Windows Latin 1 repertoire, which adds the dashes "–" and "—" as well as English (curly) quotation marks and apostrophe, and a few other characters

This repertoire is generally available on Windows systems and on most other systems as well. The extra characters usually need to be produced using special key combinations or other tools such as word processor functions. Due to character code differences between systems, the extra characters are generally not safe in email, for example.

The WGL4 repertoire

Although the repertoire has been defined by a private company and not in any standard, the characters in it are standard and rather widely available in environments other than Windows, too. The repertoire has a total of 652 characters. In addition to the characters mentioned above, it contains additional Latin letters, the basic characters used in modern Greek, a repertoire of Cyrillic letters sufficient for several languages, a mixed collection of mathematical and other symbols, and some line drawing characters.

The Unicode 2.0 repertoire

There is quite a jump from the WGL4 repertoire to the Unicode 2.0 repertoire, but there are few intermediate general purpose repertoires. Since Unicode is an evolving standard, there are considerable differences between its versions. For example, a font that purportedly supports "full Unicode" might actually support just Unicode 2.0. Newer versions are much more extensive. At the time Unicode 4.1 was published (March 2005), no widely used font supported essentially more than Unicode 2.0 (published in July 1996).

The full Unicode repertoire(s)

Unicode as currently defined is very large, but anything beyond Unicode 2.0 (except for the euro sign €, defined in 2.1) is rather unsafe. Experimental use, as well as use for well-defined limited applications, can be possible and interesting. When designing such use, select and document clearly the Unicode version you need. In the future, things can be expected to change, as font support to (at least) Unicode 4.1 will be shipped with important operating systems.

To illustrate the repertoire of characters that is reasonably "safe" in many situations, Table 1-2 shows all WGL4 characters. This is just an overview. Many of the characters cannot be identified by their shape only. The classification of the characters used in the table is a practical one, rather than formal.

Table 1-2. WGL4 characters

Classification	Characters
Basic Latin letters	ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstu vwxyz
Variants of Latin letters	ª º ⁿ ƒ
Ligatures	ﬁ ﬂ
Added Latin letters	ÆæŒœØøÐđÐðÞþßĸŊŋ
Latin letters with diacritics	ÀÁÂÃÄÅÇÈÉÊËÌÍÎÏÑÒÓÔÕÖÙÚÛÜÝàáâãäåçèéêëìíîï- ñòóôõöùúûüýÿĀāĂăĄąĆćĈĉĊċČčĎď- ĐđĒēĔĕĖėĘęĚěĜĝĞğĠġĢģĤĥĦħĨĩĪīĬĭĮįİıIJijĴĵĶķĸĹĺĻļĽľĿŀŁłŃńŅņŇňŉŌōŎŏŐőŔŕŖŗŘřŚśŜŝ ŞşŠšŢţŤťŦŧŨũŪūŬŭŮůŰűŲųŴŵŶŷŸŹźŻżŽžÅåÆæØøŴŵẀẁẂẃŸỳ
Greek characters	ΑΒΓΔΕΖΗΘΙΚΛΜΝΞΟΠΡΣΤΥΦΧΨΩΆΈΉΊΌΎΏΐΫαβγδεζηθικλμνξοπρςστυφχψωάέήίό ύώϊϋΐΰ·΄ ̈
Cyrillic letters	АБВГДЕЖЗИЙКЛМНОПРСТУФХЦЧШЩЪЫЬЭЮЯЁЂЃЄЅІЇЈЉЊЋЌЎЏабвгдежзийкл мнопрстуфхцчшщъыьэюяёђѓєѕіїјљњћќўџґґ
Digits	0123456789¹²³
Fractions	½ ¼ ¾ ⅛ ⅜ ⅝ ⅞
Punctuation	, : ; . ! ¡ ? ¿ " « » ‹ › “ ” „ ' ‚ ‘ ’ … – — ― _ † ‡ • ‰ ′ ″ ‼ ‾ ⁄
Space characters	space (U+0040), no-break space (U+00A0)
Parentheses	( ) [ ] { }
Multiple-use characters	# % & * - / \ @ ^ _ ` | ¦ ~ § ¨ ¶ · °
Spacing modifier letters	´ ¨ ¸ ˆ ˇ ˘ ˙ ˚ ˛ ˝
Currency symbols	$ ¢ £ ¤ ¥ ₣ ₤ ₱ €
Letterlike symbols	© ® ℅ ℓ № ™ µ Ω ℮
Arrows	← ↑ → ↓ ↔ ↕ ↨
Mathematical operators	+ − ± × ÷ < = > ≈ ≠ ≡ ≤ ≥ ¬ ∂ ∆ ∏ ∑ ⁄ · √ ∞ ∟ ∩ ∫
Miscellaneous technical	⌂ ⌐ ⌠ ⌡
Box drawing	─ │ ┌ ┐ └ ┘ ├ ┤ ┬ ┴ ┼ ═ ║ ╒ ╓ ╔ ╕ ╖ ╗ ╘ ╙ ╚ ╛ ╜ ╝ ╞ ╟ ╠ ╡ ╢ ╣ ╤ ╥ ╦ ╧ ╨ ╩ ╪ ╫ ╬
Block elements	▀ ▄ ▌ ▐ ░ ▒ ▓ ▲ ► ▼ ◄
Geometric shapes	■ □ ▪ ▫ ▬ ○ ● ◘ ◙ ◦ ◊
Miscellaneous symbols	☺ ☻ ☼ ♀ ♂ ♠ ♣ ♥ ♦ ♪ ♫

Numbering Characters

Definitions in character standards assign a number to each character. The numbers are unique in each standard, but different standards assign the numbers differently. Some commonly used standards are mutually compatible, in part: the numbers of characters in ASCII (ranging from 0 to 127) are the same as in the ISO 8859 standards, and the numbers of characters in ISO 8859-1 (ranging from 0 to 255) are the same as in Unicode.

The numbers are nonnegative integers 0, 1, 2,..., but are not necessarily consecutive; there can be gaps in the assignment. For example, in ISO 8859 standards, numbers in the range 128 to 159 are unassigned; more specifically, they are reserved for control purposes, leaving it up to other standards to define them. Unicode contains a lot of gaps, due to the coding structure, partly in order to leave space for future extensions.

It might sound natural to use the first few code numbers for digits 0, 1,..., but character standards use different assignments. Don't expect to find much logic in it. The code number of a character should be treated as fairly arbitrary, but fixed.

The number assigned to a character in a character standard has many different names: *code number*, *code position*, *code value*, *code element*, *code point*, *code set value*, as well as simply *code*. In the Unicode standard, the term "code point" is used both about a number and about a location in the coding space where a character could reside. Some code points are allocated for characters, a few have been explicitly designated as not corresponding to characters (now or ever), and most code points are still not assigned in any way.

Since characters are internally represented by their code numbers, a character can also be treated as an integer. In fact, many old programming languages lack a data type for characters and use an integer type instead. However, the code numbers are usually not used in arithmetic operations, since they mostly lack numeric meaning. If a character's number is smaller than another character's number, this by no means implies a corresponding relation in alphabetic order. For some small regions of code numbers, the order actually corresponds to alphabetic order, though.

For example, in Unicode, the numbers for the characters "a," 0 (digit zero), ! (exclamation mark), ä (letter a with umlaut), and ‰ (per mille sign) are 97, 48, 33, 228, and 8240 in decimal notation. More often, hexadecimal notation is used: 61, 30, 21, E4, and 2030. The code number assignments are essentially arbitrary: the code number has no relationship with the meaning of a character.

Normally, you do not need to memorize the numbers; you check them from suitable references. However, if you use some code numbers frequently, you will probably learn to remember some of them by heart. This explains the sarcastic saying: "Real Programmers might or might not know their spouse's name. They do, however, know the entire ASCII (or EBCDIC) code table."

Hexadecimal Notation

As mentioned above, character numbers are usually specified in hexadecimal notation, or *hex notation*. The phrase *hexadecimal number* is often used, but in fact, it is just a convention for writing numbers. The hexadecimal notation FF denotes the same number as the decimal notation 255.

In hexadecimal notation, letters "A" through "F" (or "a" through "f") are used to denote numbers from 10 to 15 (10 to 15 in decimal notation). The number denoted by a two-

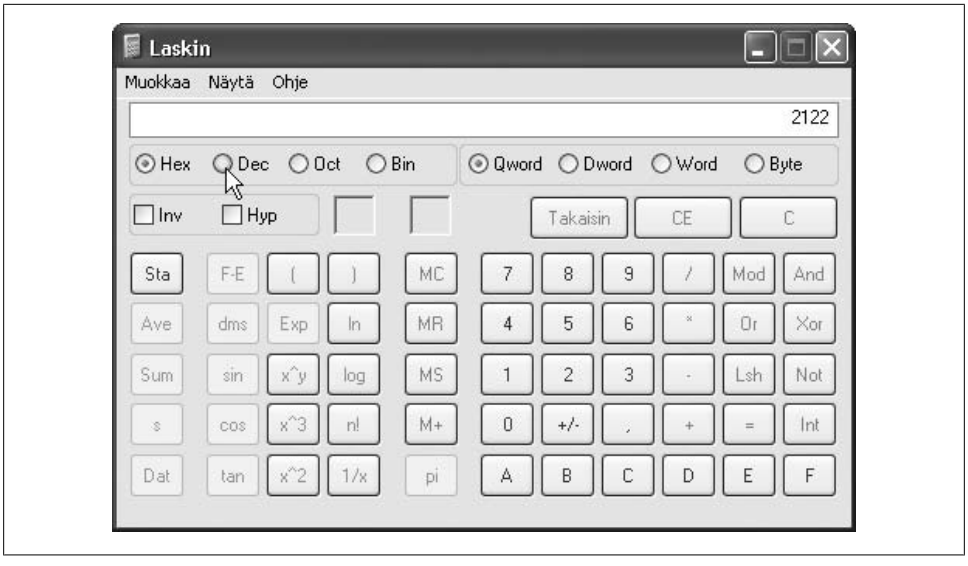


Figure 1-8. The Calculator in Windows XP, in Scientific mode

digit hexadecimal notation is the value of the first digit times 16 plus the value of the second digit. For example, hexadecimal 2E means $2 \times 16 + 14 = 46$ in decimal. Similarly, the four-digit hexadecimal notation 215A means $2 \times 16^3 + 1 \times 16^2 + 5 \times 16 + 10 = 8{,}538$ in decimal. The largest four-digit hexadecimal number is FFFF, which is 65,535 in decimal.

It is usually evident from context whether a number is presented as hexadecimal or decimal. In particular, Unicode code numbers written as U+*nnnn* are always in hexadecimal. When necessary, some special convention is used to indicate the base. In plain text, it is common to use a "0x" (digit zero, letter "x") prefix for hexadecimal numbers, such as in "0x215A." In mathematical notations, the base is often written (in decimal) as a subscript, as in $215A_{16}$ or $8{,}538_{10}$.

It is easy but boring to convert between decimal and hexadecimal, so we mostly use computers for that. In Windows, the Calculator program can be used for such conversions, when set in "scientific" mode. As shown on Figure 1-8, you can, for example, set the Calculator to hexadecimal mode, enter a number, and click on "Dec" to get the value in decimal.

The reason for using hexadecimal notation in character code issues is that Unicode and other standards use that notation. This in turn reflects the design decisions of using 8-bit bytes and grouping characters into 256-character sets. For example, the Unicode number U+205F denotes the character in relative position 5F inside the set U+2000..U+20FF. Such handy things are not possible if decimal numbers are used.

Another reason is that it is trivial and fast to convert between hexadecimal and binary, and computers internally use binary. Each hexadecimal digit corresponds to 4 bits: 0 = 0000, 1 = 0001, 2 = 0010, 3 = 0011,..., E = 1110, F = 1111.

Numbers as Indexes

We can regard a character code as a row of boxes, each capable of containing one character. In many widely used old character codes, the sequence has 256 boxes. In Unicode, the sequence is about a million boxes long. Although Unicode is often presented as a set of code tables (arrays), each consisting of 256 elements, its fundamental structure is essentially linear.

The code numbers are ordinal numbers, or indexes, of the boxes, starting from zero. They can also be understood as indexes to tables of properties of characters. Thus, to find out whether a particular character is a letter in the most general sense, you would conceptually use the character's code number to access a table that contains information about the general category of each character. Actual implementations do not necessarily use such table lookup techniques, but the idea illustrates the point of using code numbers.

There are some things to note on this model, however:

- Not all boxes contain a character. That is, not all code points correspond to a character. In Unicode, *most* code points are currently unassigned, and some have been explicitly defined as "noncharacters"—i.e., as not corresponding to any character, ever.
- Not all characters have a box of their own, or a code point. Some characters containing a diacritic mark can only be written as decomposed—i.e., as a base character followed by one or more combining diacritic marks. For example, the letter "e" with acute accent, é, has a box of its own; but the Cyrillic letter ю with an acute accent on it (ю́), though used as a character in dictionaries, for example, has no code point—it can only be represented as ю followed by a combining acute accent.
- Although the numbering implies an order, this order is mostly arbitrary and not used much. For instance, if a character's code point is numerically smaller than another character's code point, this implies in general nothing about the mutual order of the characters in alphabetic or sorting order.

Thus, although characters are identified by their code points, which are numbers (unsigned integers), the numeric (arithmetic) value is usually irrelevant. That is, we mostly don't operate on them as numbers, with arithmetic operations. For most purposes, the numbers are just indexes. It is not a pure coincidence, though, that some characters have code points that correspond to their mutual alphabetic order. Many character codes have put letters into alphabetic order, and Unicode has tried to preserve much of that.

Making Use of Character Numbers

There are several ways to use the Unicode number of a character. The methods of writing characters will be discussed in Chapter 2, but here are some possibilities:

- In HTML and XML authoring, you can use a *character reference* of the form `&#x` *`number`* `;`—e.g., `&#x212e;`. That way, you can include any character, no matter what your keyboard is or what your document's encoding is.
- On Microsoft software that uses the so-called *Uniscribe* input (e.g., many programs under Windows XP), you can type a character's number in hexadecimal, such as 212e, and then type Alt-X and see how the number is replaced by the character.
- You can use the number as an index to information on characters in different tables, databases, and services, including the Unicode standard.
- You can select a character by its number in user interfaces such as the Character Map in Windows, as illustrated earlier in Figure 1-1, or the window that opens in Microsoft Word when you select Insert → Symbol. The latter is illustrated in Figure 1-9, which shows the window in a Finnish version of Word. As you can see, the character name shown is still the Unicode name as such—in this case, ESTIMATED SYMBOL.

Encoding Characters as Octet Sequences

When we need to store character data on a computer, we might consider storing it in an exact visual shape. Some people would call this a very naive idea, but it is in fact quite feasible, even necessary—for some purposes. If you have an old manuscript to be stored digitally, you need to scan it with high resolution and store it in some image format. Sometimes you would do that for individual characters as well. On web pages, for example, it is common to use images containing text for logos, menu items, buttons, etc., in order to produce a particular visual appearance.

For most processing of texts on computers, however, we need a more abstract presentation. It would be highly impractical to work on scanned images of characters in storing and transferring text, not to mention comparing strings for example. We do not want to do the process of recognizing a character's identity every time we use the character. Instead, we use characters as atoms of information, identified by their code numbers or some other simple way. This is really what "abstract characters" are about.

Plain Text and Other Formats for Text

Plain text is a technical term that refers to data consisting of characters only, with no formatting information such as font face, style, color, or positioning. However, formatting such as line breaks and simple spacing using space characters may be included, to the extent that it can be expressed using control characters only. Moreover, all char-

Figure 1-9. Character insertion window in Microsoft Word lets you select a character by its Unicode number, as one possibility

acters are to be taken as such, without interpreting them as formatting instructions or tags. For example, HTML or XML is not plain text.

Plain text is a format that is readable by human beings when displayed as such. The reader needs to know the human language used in the text, of course. The display of plain text depends on the font that happens to be used. This can often be changed within a program, but such settings change the font of all text. (As an exception, if the font chosen does not contain all the characters used in the text, a clever program might use other fonts as backup for missing characters.)

Plain text is a primitive format, but it is extremely important. In processing texts, the ultimate processing such as searching, comparison, editing, or automatic translation takes place at the plain text level. Databases often use plain text format for strings for simplicity, even though data extracted from them is presented as formatted.

Plain text is universal, since no special software is needed for presenting it, as long as a suitable font can be used. In Windows, you can use Notepad, the very simple editor, to write and display plain text, as illustrated in Figure 1-10. There are many plain text editors, some of which contain fairly sophisticated processing tools. However, their character repertoire is often very small.

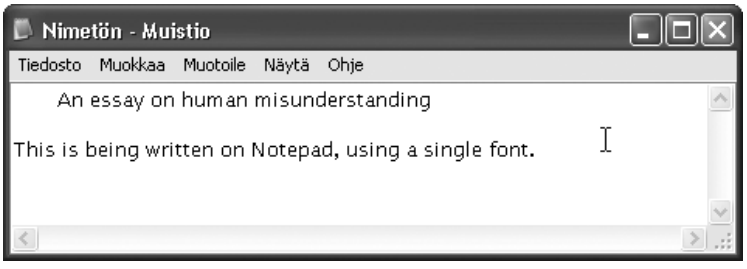


Figure 1-10. Using Windows Notepad, a simple plain text editor

Plain text is commonly used, and normally should be used, in email, Internet discussion forums, simple textual documents (like *readme.txt* files shipped with software), and many other purposes. For example, the RFC (Request for Comments) series of documents, describing the standards, protocols, and practices on the Internet, have ASCII plain text as their format.

The phrase "ASCII file" or "ASCII text" is often used to denote plain text in general, even in contexts where ASCII is obviously not used and could not be used. This is because ASCII has been used so long and so widely, and quite often "ASCII" is mentioned as an opposed to anything that is not plain text (e.g., "ASCII" versus "binary" transfer).

Text-processing programs such as Microsoft Word normally process and store text data in a format that is somehow "enhanced" with formatting and other information. This includes the use of different fonts for different pieces of text, specific positioning and spacing, and invisible metadata ("data about data") such as author name, program version, and revision history. Figure 1-11 illustrates the use of Microsoft Word.

If you create a file containing just the word "Hello" using a plain text editor, the file size will be five octets, or maybe 10 octets, depending on encoding. If you do the same using a text-processing program such Microsoft Word, you might get, for example, a file size of 24 kilobytes (24,576 octets), because a text-processing program inserts a lot of basic information in its internal format, in addition to the text itself and some information about its appearance (font face and size). This means that if you accidentally open a Word file in a plain text editor or process it with a program prepared to deal with plain text only, you get a mess. This is illustrated in Figure 1-12, which contains a Word document with just the text "This is a simple document." written into it, opened in Notepad.

Similar considerations apply to widely used document formats such as *PDF* (Portable Document Format). There are also intermediate formats, which contain text and formatting information, such as *RTF* (Rich Text Format). It was designed for purposes like exchanging text data between different text-processing programs.

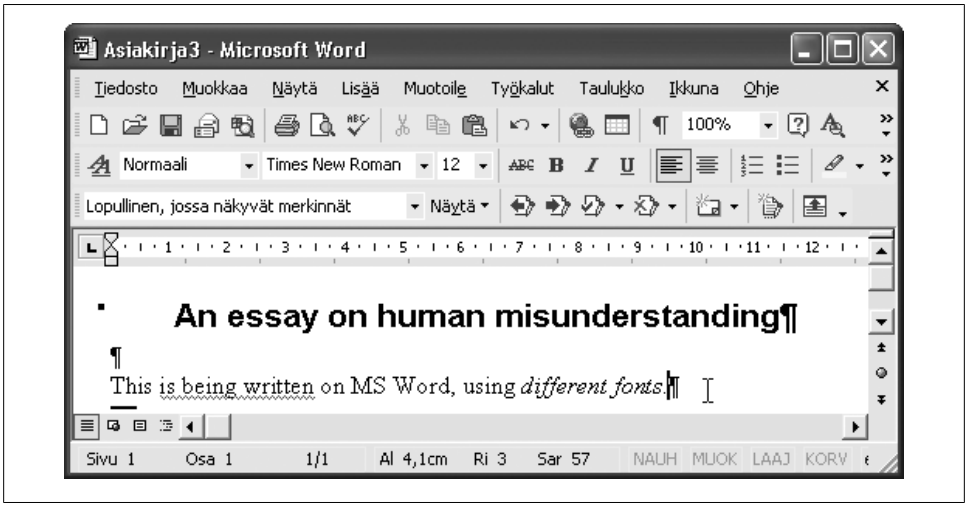


Figure 1-11. Using Microsoft Word, a text-processing program

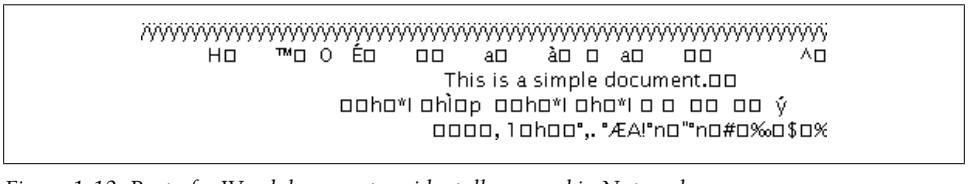


Figure 1-12. Part of a Word document accidentally opened in Notepad

If you are not familiar with the idea of formats like plain text, Word format (often called "*.doc* format" due to the common filename extension), and RTF, you could launch your favorite text-processing program. Write a short document and save it, and then use the "Save As" option in the "File" menu (or equivalent), and study and test the alternative save formats you find there. Save the document in each format in the same folder, and then view the contents of the folder, with file sizes displayed. You might then see what happens if you open each format in a text editor like Notepad, just to get a general idea. Such exercises will prepare you to deal with requests like "Please send your proposal in RTF format." This will also aid you in recognizing situations like receiving an RTF file when you have asked for something else.

Bytes and Octets

In computers and in data transmission between them—i.e., in digital data processing and transfer—data is internally presented as octets, as a rule. An *octet* is a small unit of data with a numerical value between 0 and 255, inclusively. The numerical values are presented in the normal (decimal) notation here, but other presentations are widely

used too, especially octal (base 8) or hexadecimal (base 16) notation. The hexadecimal values of octets thus range from 0 to FF.

Internally, an octet consists of 8 bits—hence the name, from Latin octo "eight." A *bit*, or binary digit, is a unit of information indicating one of two alternatives, commonly denoted with the digits 0 and 1 in writing but internally represented by some small physical entity that has two distinguishable states, such as the presence or absence of a small hole or magnetization. The digits 0 and 1 that symbolize the values are also called bits (0 bit, 1 bit). When a bit has the value 1, we often say that the bit has been set.

In character code contexts, we rarely need to go into bit level. We can think of an octet as a small integer, usually without thinking how it is internally represented. However, in some contexts the concept of *most significant bit* (MSB), also called *first bit* or *sign bit* is relevant. If the most significant bit of an octet is set (1), then in terms of numerical values of octets, the value is greater than 127 (i.e., in the range 128–255). In various contexts, such octets are sometimes interpreted as *negative* numbers, and this may cause problems, unless caution is taken.

The word "byte" is more common than "octet," but the octet is a more definite concept. A *byte* is a sequence of bits of a known length and processed as a unit. Nowadays it is almost universal to use 8-bit bytes, and therefore a distinction between a byte and an octet is seldom made. However, in character code standards and related texts, the term octet is normally used.

There is nothing in an octet that tells how it is to be interpreted. It is just a bit pattern, a sequence of eight 0s or 1s, with no indication of whether it represents an integer, a character, a truth value, or something else. For example, the octet with value 33 in decimal, 00010001 in binary (as bits), could represent the exclamation sign character !, or it could mean just the number 33 in some numeric data. It could well be just part of the internal representation of a number or a character. Information about the interpretation needs to be kept elsewhere, or implied by the definition of some data structure or file format.

Character Encodings

A character encoding can be defined as a method (algorithm) for presenting characters in digital form as sequences of octets. We can also say that an encoding maps code numbers of characters into octet sequences. The difference between these definitions is whether we conceptually start from characters as such or from characters that already have code numbers assigned to them.

There are hundreds of encodings, and many of them have different names. There is a standardized procedure for registering an encoding, and this means that a primary name is assigned to it, and possibly some alias names. For example, ASCII, US-ASCII, ANSI_X3.4-1986, and ISO646-US are different names for an encoding. There are also many unregistered encodings and names that are used widely. The Windows Latin 1 encod-

ing, which is very common in the Western world, has only one registered name, windows-1252, but it is often declared as cp-1252 or cp1252.

The case of letters is not significant in character encoding names. Thus, "ASCII" and "Ascii" are equivalent. Hyphens, on the other hand, are significant in the names.

Single-Octet Encodings

For a character repertoire that contains at most 256 characters, there is a simple and obvious way of encoding it: assign a number in the range 0–255 to each character and use an octet with that value to represent that character. Such encodings, called single-octet or 8-bit encodings, are widely used and will remain important. There is still a large amount of software that assumes that each character is represented as one octet.

Various historical reasons dictate the assignments of numbers to characters in a single-octet encoding. Usually letters A–Z are in alphabetic order and digits are in numeric order, but the assignments are otherwise more or less arbitrary. Besides, any extra Latin letters, as used in many languages, are most probably assigned to whatever positions that were "free" in some sense. Thus, if you compare characters by their code numbers in a single-octet encoding, you will generally not get the right alphabetic ordering by the rules of, for example, French or German.

Multi-Octet Encodings

As you may guess, the next simpler idea of using *two* octets for a single character has been invented, formalized, and used. It is not as common as you might suspect, though.

A simple two-octet encoding is sufficient for a character repertoire that contains at most 65,536 characters. An octet pair (*m*,*n*) represents the character with number $256 \times m + n$. Alternatively, we can say that the number is represented by its 16-bit binary form. This makes processing easy, but there are two fundamental problems with the idea:

- Each character requires two octets, which is rather uneconomical if the text mostly consists of characters that could be presented in a single-octet encoding.
- Unicode is no longer limited to the code number range 0..65,536, so extra methods would anyway be needed to represent characters outside it.

Thus, encodings that use a *variable* number of octets per character are more common. The most widely used among such encodings is *UTF-8* (UTF stands for Unicode Transformation Format), which uses one to four octets per character. For those writing in English, the good news is that UTF-8 represents each ASCII character as one octet, so there is no increase in data size unless you use characters outside ASCII.

If you accidentally view an UTF-8 encoded document in a program that interprets the data as ASCII or windows-1252 encoded, you will notice no difference as long as the data contains ASCII only. In fact, any ASCII data can trivially be declared as UTF-8 encoded as well. If the data contains characters other than ASCII, they would in this

case be displayed each as two or more characters, which have no direct relationship with the real character in the data. This is because consecutive octets would be interpreted as each indicating a character, instead of being treated according to the encoding as a unit.

The "Character Set" Confusion

Character encodings are often called *character sets*, and the abbreviation *charset* is used in Internet protocols to denote a character encoding. This is confusing because people often understand "set" as "repertoire." However, *character set* means a very specific *internal representation* of characters, and for the same repertoire, several different "character sets" can be used. A character set implicitly defines a repertoire, though: the collection of characters that can be represented using the character set.

It is advisable to avoid the phrase "character set" when possible. The term *character code* can be used instead when referring to a collection of characters and their code numbers. The term *character encoding* is suitable when referring to a particular representation.

For example, the word "ASCII" can mean a certain collection of characters, or that collection along with their code numbers 0–127 as assigned in the ASCII standard, or even more concretely, those code numbers (and hence the characters) represented using an 8-bit byte for each character.

Working with Encodings

When you use characters on a computer, some software will internally encode them in binary format. Most users never need to know the details of this, still less need to actually handle the encoding process, but it is essential to know that there are different encodings, with different properties. In transferring data between applications and computers, you may need to change the encoding or select a suitable encoding.

Selecting the Encoding When Saving

Text editors and many other programs typically have a File menu, with a Save function for storing data onto disk. Normally, this function uses the file format and the character encoding that is typical of the program. However, there is usually also a Save As function, which lets the user select the format and encoding. This function is often used because it lets you save an edited document under a different filename.

The Save As function is often the simplest way to *convert* between different encodings (and file formats). You simply open a file and save it differently. For example, suppose you have used Notepad to create a plain text file. If you use, for example, an English version of Windows, the default encoding that Notepad uses is Windows Latin 1. Now suppose that a friend has asked you to send your text in the UTF-8 encoding for some

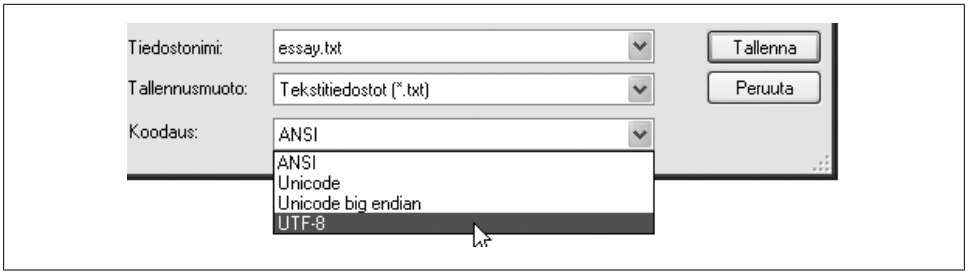


Figure 1-13. An extract from a Save As dialog in Notepad

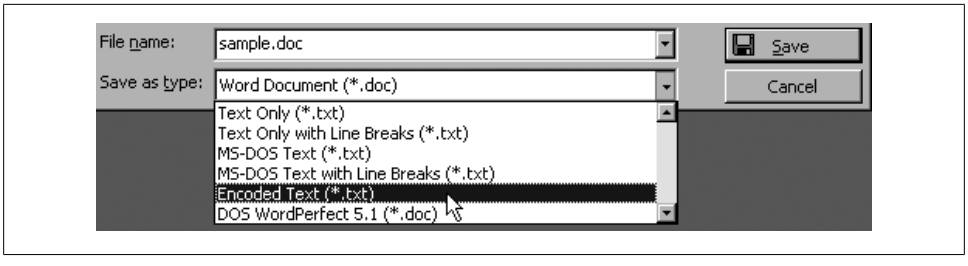


Figure 1-14. An extract from a Save As dialog in Microsoft Word

reason. You simply open your file in Notepad, select File → Save As and then choose the UTF-8 encoding from the menu of encodings, as shown in Figure 1-13. It illustrates the three basic things you can (and need to) specify in Save As dialogs: the filename, the file format, and the encoding.

The list of possible encodings in a Save As dialog varies greatly, and the names of the encodings are not always official names. For example, in Microsoft products, "ANSI" often appears as denoting the character code that the system uses as its normal 8-bit code, such as the Windows Latin 1 encoding, which should be called "windows-1252." The word "Unicode" may denote different encodings used for Unicode, typically UTF-16. Use the UTF-8 encoding for Unicode text, unless you have a good reason for doing otherwise.

When using a text-processing program, the situation is usually different. There is a file format menu in the Save As dialog but often no encoding menu. The reason is that in text processing, the overall format is crucial, and the encoding is often coupled with the format.

In Microsoft Word, for example, the list of formats may contain alternatives as shown in Figure 1-14, with options corresponding to the internal formats of different programs and some plain text formats. Here, too, it may require some guesswork or study to identify what the options really mean. On Windows systems, "*.txt" is associated with several different encodings, and "*.ans" refers to ANSI (e.g., windows-1252). The notation "*.asc" may suggest ASCII encoding, but in fact it refers to an old DOS encoding, a *code page*, which is a single-octet encoding and may vary from one system to another.