Developing Global Applications in Java

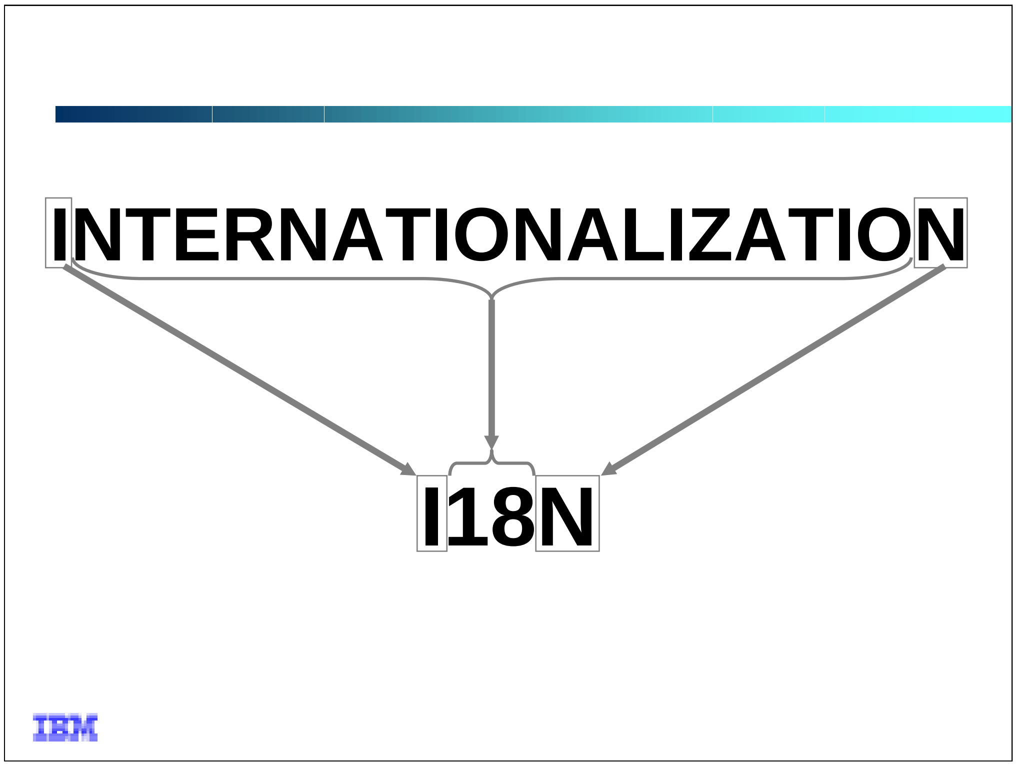
Richard Gillam
Unicode Technology group
Center for Java Technology, Cupertino
IBM

I'm Richard Gillam from the Unicode Technology group in IBM's Center for Java Technology in Silicon Valley, and I'm here today to talk about developing global applications, a procedure that's generally known as "internationalization."

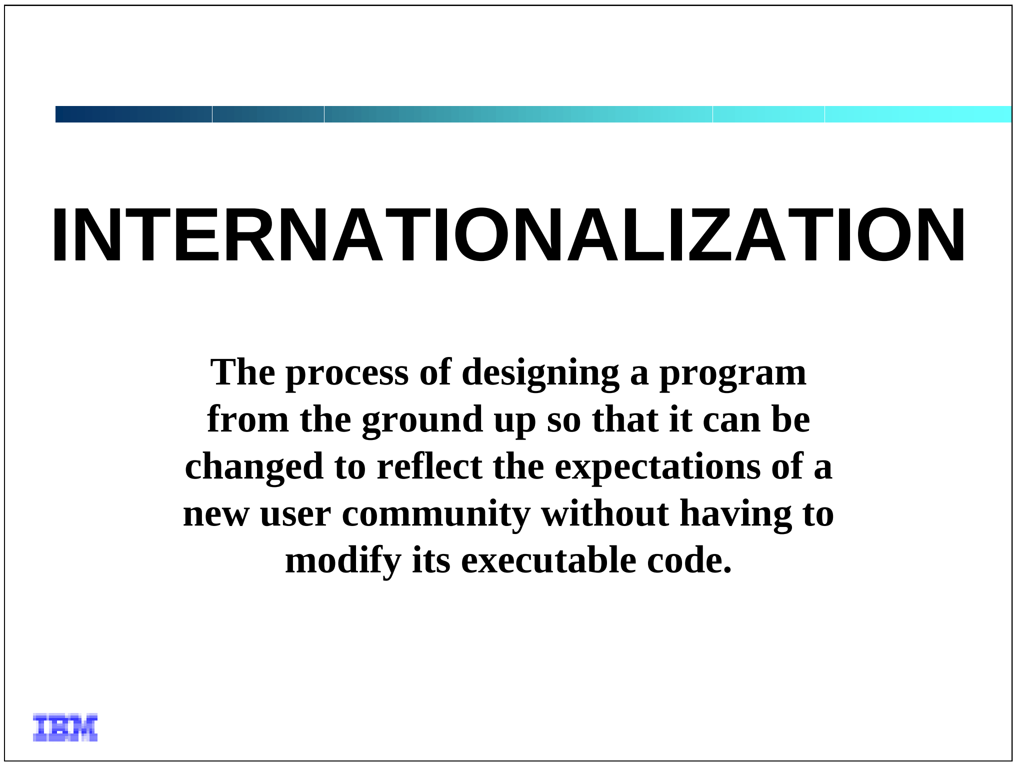
Our group at IBM designed much of the internationalization support in the Java Class Libraries under contract to Sun. What I want to do today is give you a guided tour through these classes, what they do, and how to use them.

INTERNATIONALIZATION

The first thing we should probably do is look at the term “internationalization” and what we mean by it. One of the interesting things about it is that it’s the only 20-letter word I’ve seen that seems like a “normal word”.



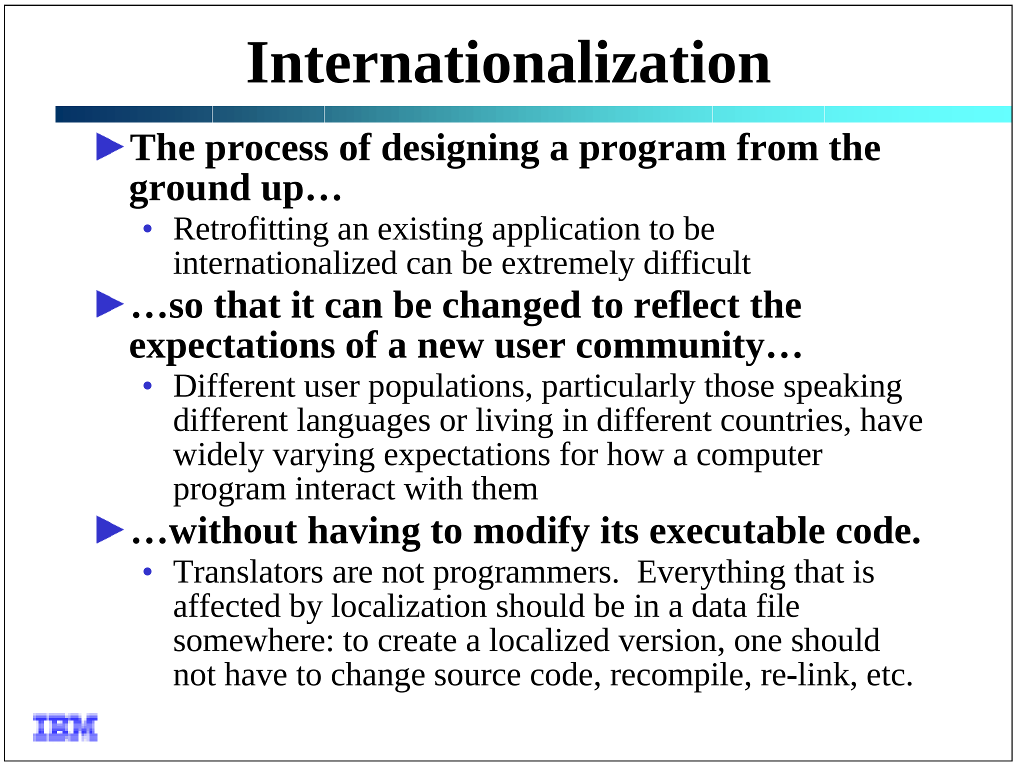
Because it’s such a long word, you’ll often see it abbreviated as I18N (pronounced as “eye eighteen en”). You have an I, 18 other letters, and an N. You’ll also se this type of approach used with other words in this field, such as “localization” (L10N).



INTERNATIONALIZATION

The process of designing a program from the ground up so that it can be changed to reflect the expectations of a new user community without having to modify its executable code.

Internationalization is the process of designing a program from the ground up so that it can be changed to reflect the expectations of a new user community without having to modify its executable code.



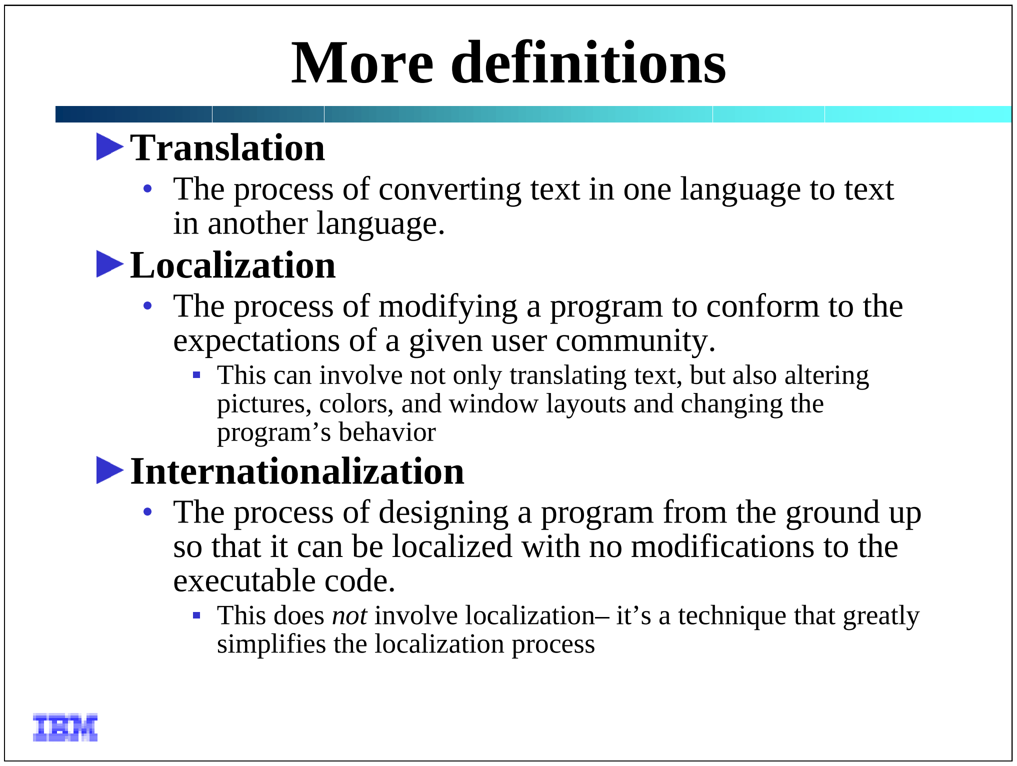
Internationalization

- **The process of designing a program from the ground up…**
  - Retrofitting an existing application to be internationalized can be extremely difficult
- **…so that it can be changed to reflect the expectations of a new user community…**
  - Different user populations, particularly those speaking different languages or living in different countries, have widely varying expectations for how a computer program interact with them
- **…without having to modify its executable code.**
  - Translators are not programmers. Everything that is affected by localization should be in a data file somewhere: to create a localized version, one should not have to change source code, recompile, re-link, etc.

Let's explore that long definition a little more closely. I've purposely used the vague phrase "reflect the expectations of a new user community" because I don't want to suggest that the whole process is about transating programs into other languages. This is the biggest part of it, of course, but it's equally important that a program follow local conventions for things such as how a number or date is written. And, of course, the granularity there isn't always by country. In the U.S., for example, civilians and military personnel write dates and times differently.

Designing the program from the ground up with this kind of customization in mind is vital. Trying to customize a program that hasn't been designed for it is extremely time-consuming and error-prone.

And since most programmers aren't experts on the various communities where their software is used, the translators aren't likely to be the original development team. In fact, they're not likely to be computer programmers at all. Providing translators a way to do their job that doesn't involve recompiling or re-linking the program is essential.

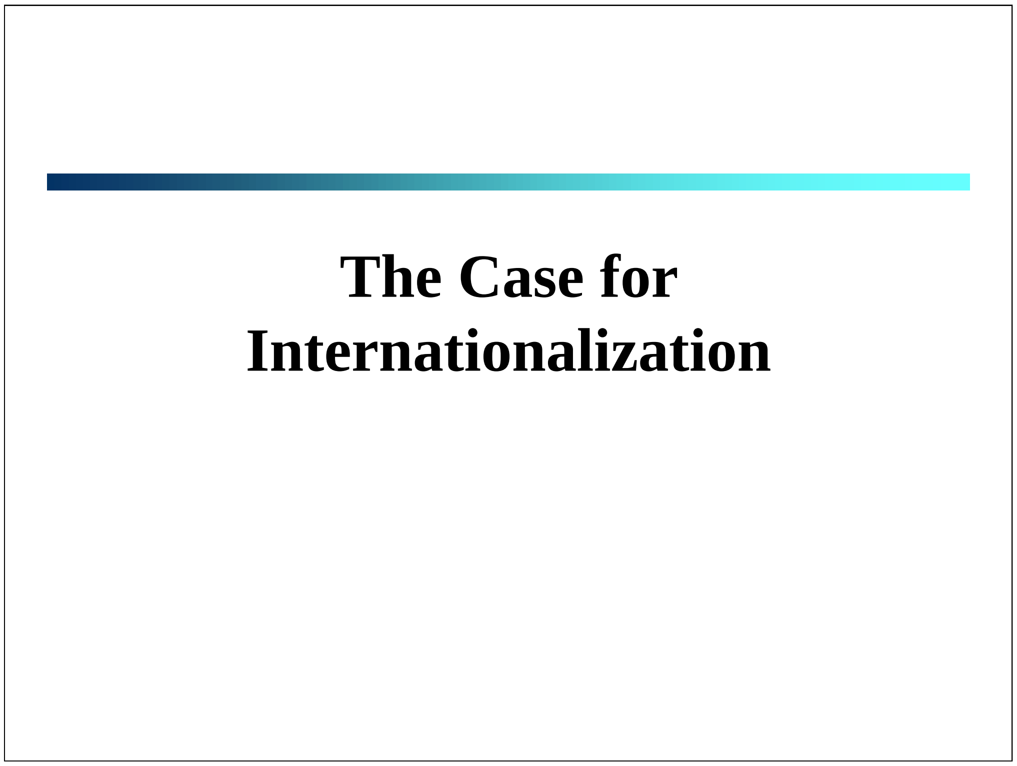


More definitions

- **Translation**
  - The process of converting text in one language to text in another language.
- **Localization**
  - The process of modifying a program to conform to the expectations of a given user community.
    - This can involve not only translating text, but also altering pictures, colors, and window layouts and changing the program’s behavior
- **Internationalization**
  - The process of designing a program from the ground up so that it can be localized with no modifications to the executable code.
    - This does *not* involve localization– it’s a technique that greatly simplifies the localization process

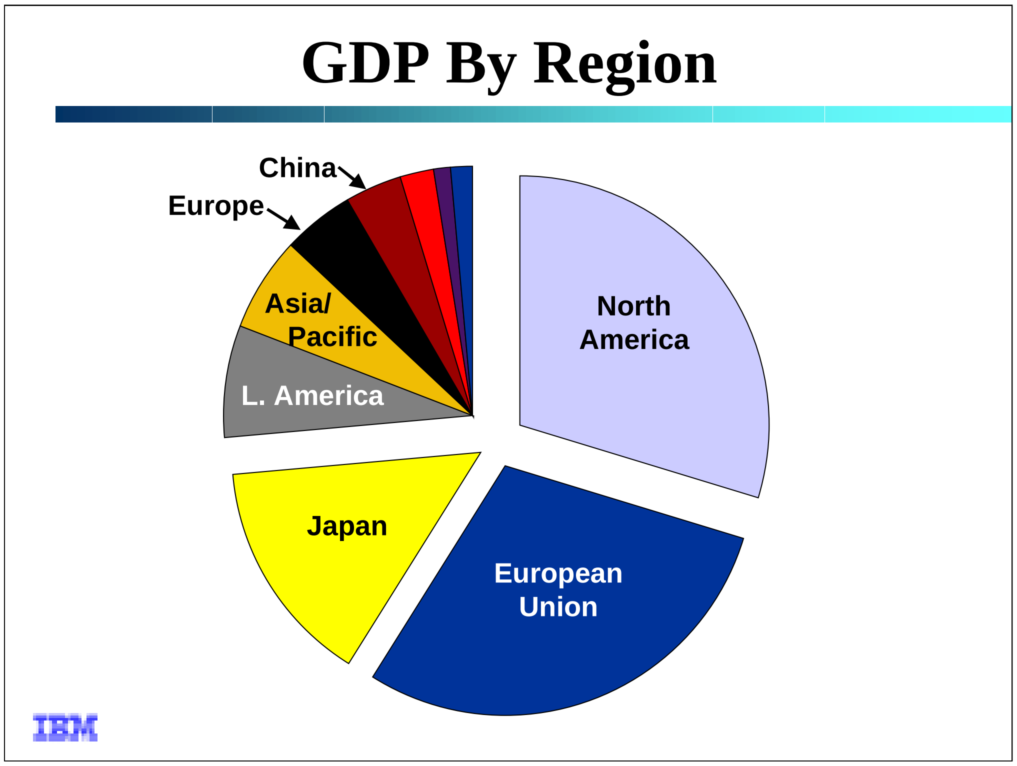
Internationalization is one of the terms you’ll hear lot when talking about global software. The other two are translation and localization. Localization is more than merely translating between languages– it also involves changing everything else about a program’s appearance or behavior that might be affected by a country or other population’s customs, beliefs, and preferences. Pictures may have to change– mailboxes tend to look different in different countries, for example. Colors may have different connotations in different countries. And so on.

Localization is distinct from internationalization. Localization (which includes translation) is what translation houses do to software to prepare it for a particular market. Internationalization is what programmers do to make sure the program can be localized easily.

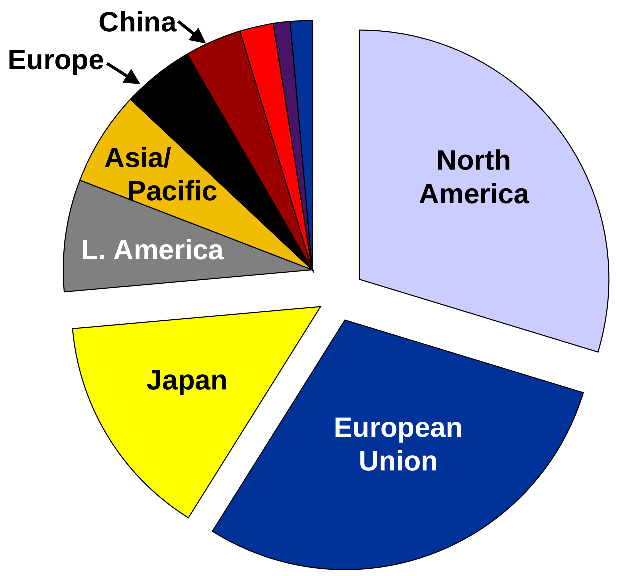


The Case for Internationalization

If you’re at this conference, you probably already know why internationalization is important, but let’s take a few minutes to look at that issue. Even if I’m preaching to the choir, I might be able to supply you with more ammunition to convince other people.

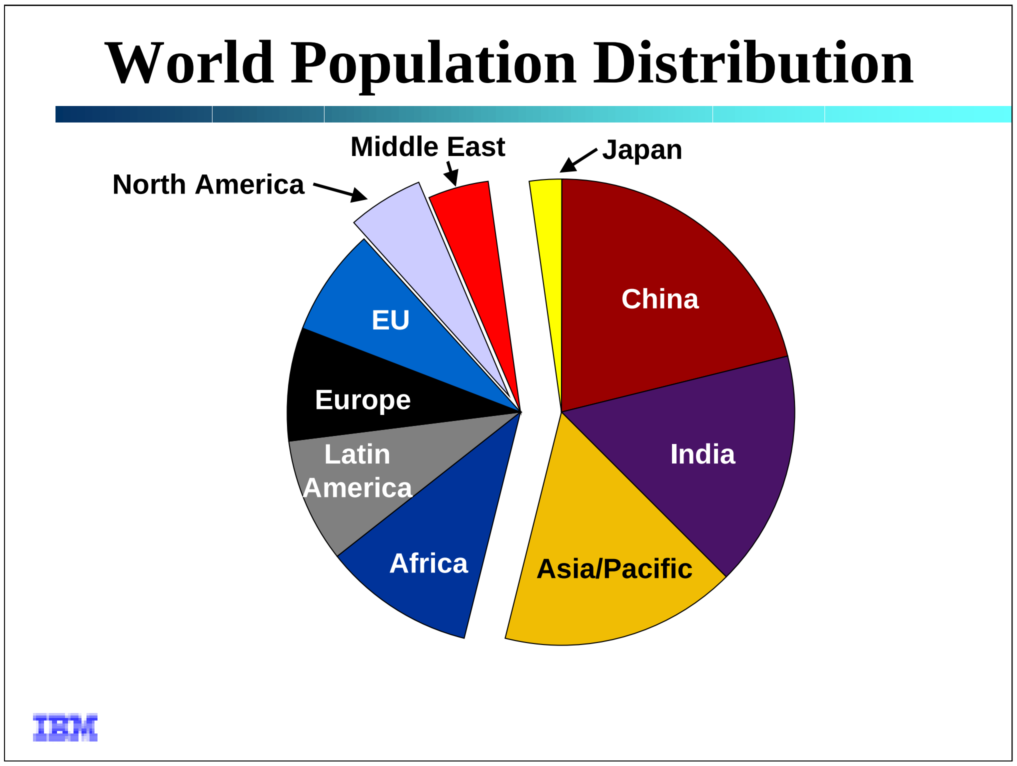


GDP By Region

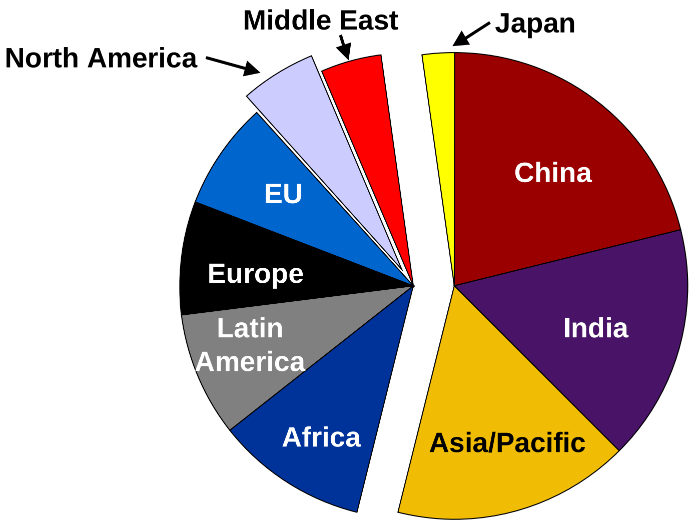


This is a graph of the world GDP distribution. North America is the single largest part, but it still represents only a third of the world's economy. The European Union is actually just as big as North America, representing another third of the world's economy. Obviously, people in most of the EU countries either don't speak English at all or don't speak it natively. Japan represents another fifth of the world's economy, and again most Japanese don't speak English.

The rest of the world accounts for a little over a fourth of the world's economy, so while any one country or region may be small, the whole thing still represents lots of dollars. (And parts of it, such as the rest of Europe, the rest of Asia, and especially Latin America, don't represent that much incremental work.) As you can see, two-thirds of the dollars to be made out there comes from places other than the U.S. and Canada.

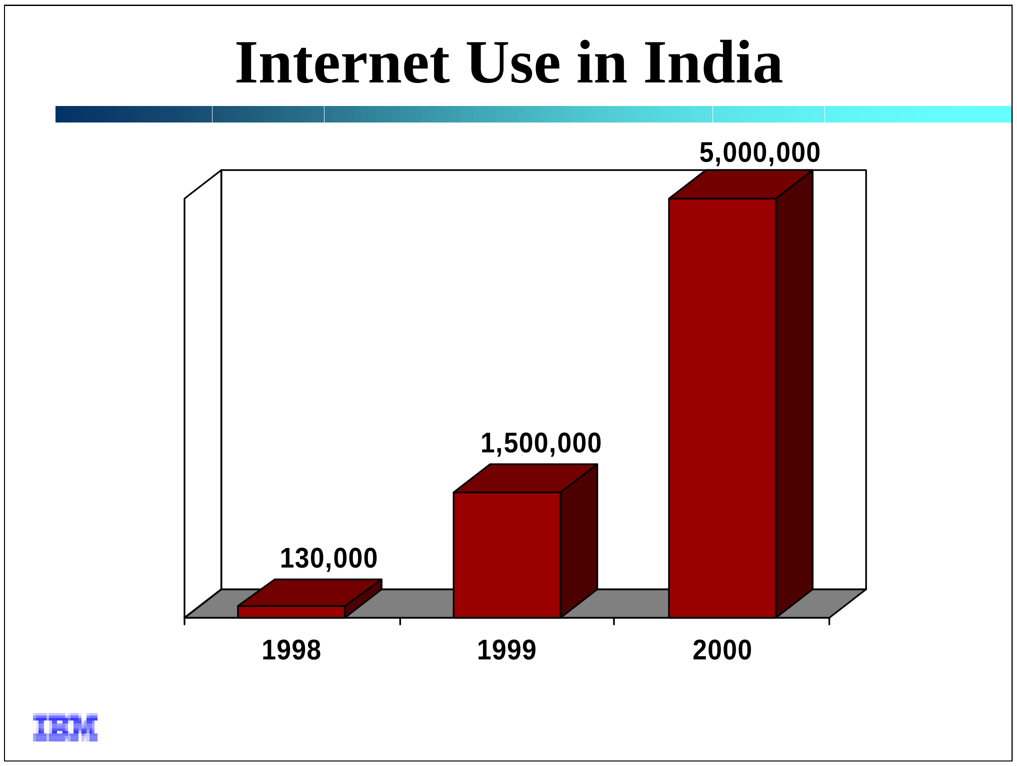


World Population Distribution

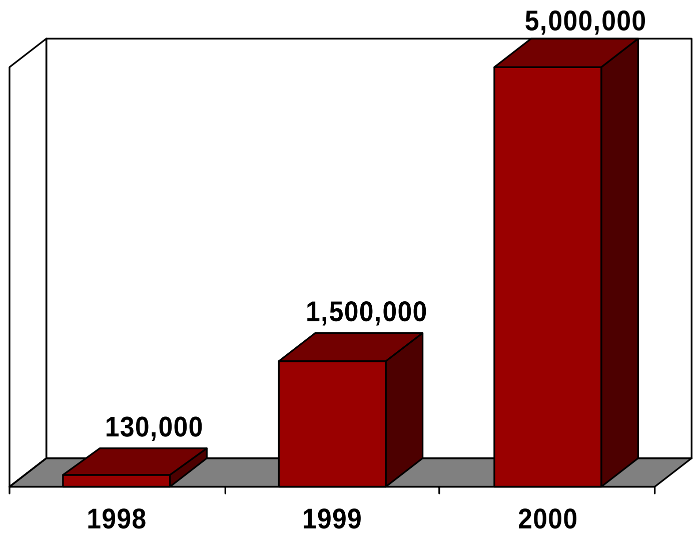


The small slivers also don't tell the whole story. This is a graph of world population distribution. Notice how radically the rankings change. First, notice how small a sliver North America is on this graph. Second, notice that more than half of the world's population is in Asia.

Also notice that China and India, which barely showed up on the previous graph, dominate this graph. Countries with small slivers of the GDP graph and big sections of the population graph represent large potential markets. Of course, much depends on how the economies in these countries are growing and how technological they're becoming, but China and India are both making big pushes to modernize right now.



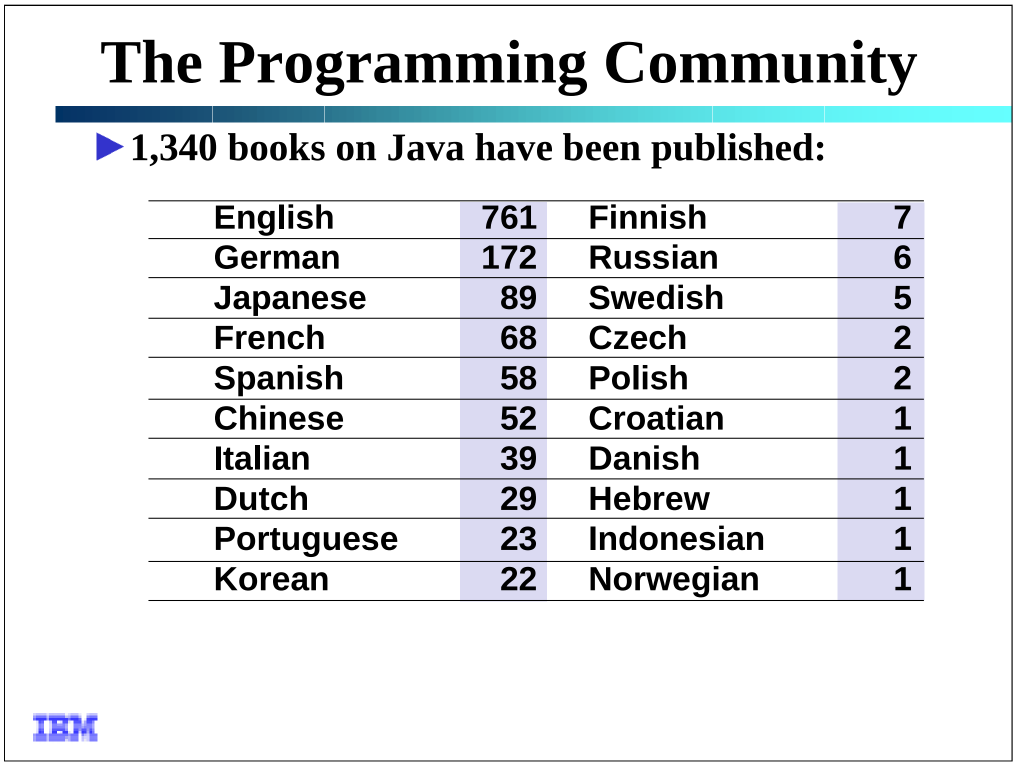
Internet Use in India



IBM

In India, for example, the number of people having user accounts on the Internet is exploding. At the end of last year, analysts estimated that there were about 130,000 registered Internet users in India. By the end of this year, thanks to loosening telecommunications regulation, that number will grow to a million and half. By the end of next year, they're expecting it to grow to five million. Furthermore, these numbers represent only registered Internet users. Analysts estimate that the 130,000 number for last year represents about a million actual users due to doubling up of accounts.

As you can probably imagine, these are the kinds of figures that cause dollar signs to light up in CEOs' eyes.

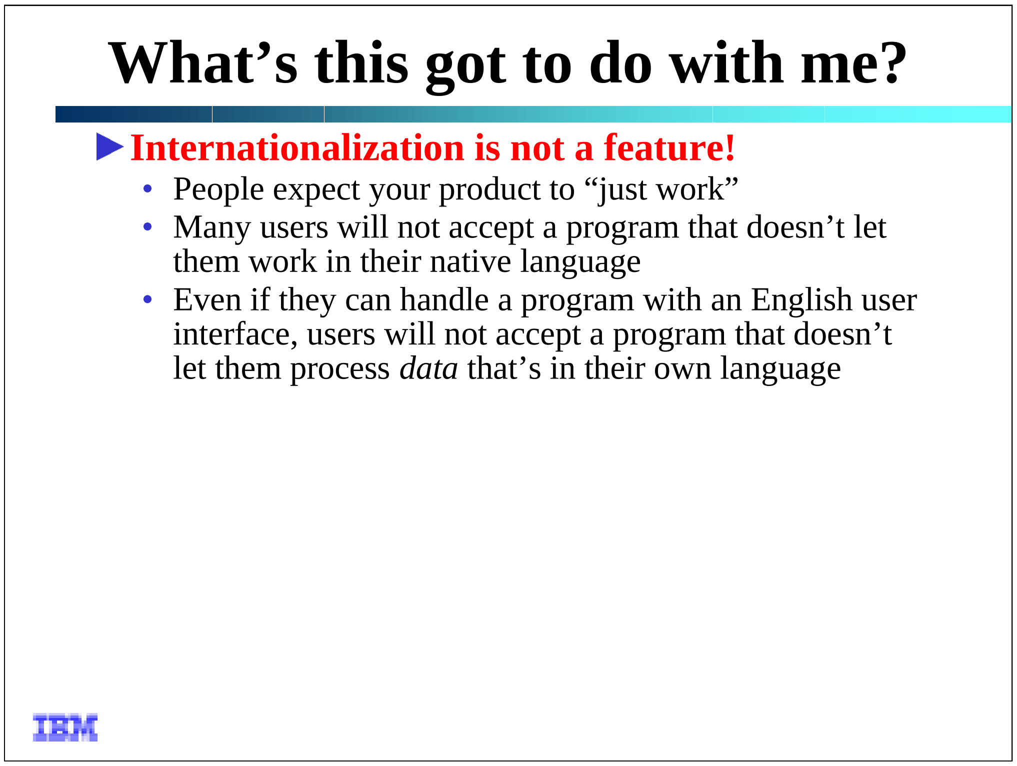


The Programming Community

- 1,340 books on Java have been published:

English	761	Finnish	7
German	172	Russian	6
Japanese	89	Swedish	5
French	68	Czech	2
Spanish	58	Polish	2
Chinese	52	Croatian	1
Italian	39	Danish	1
Dutch	29	Hebrew	1
Portuguese	23	Indonesian	1
Korean	22	Norwegian	1

Finally, just as another point of reference, this graph shows the number of books on Java that have been published in various languages. Java books have been published in twenty languages, and while more than half of them were in English, the number of non-English books is still striking. This gives you a rough idea of the size of the developer community speaking languages other than English, and that in turn gives you a rough idea of the sizes of the user communities represented by the developer communities. (Of course, there are also many English-speaking developers who primarily produce software for use in other countries and languages, too.)



What's this got to do with me?

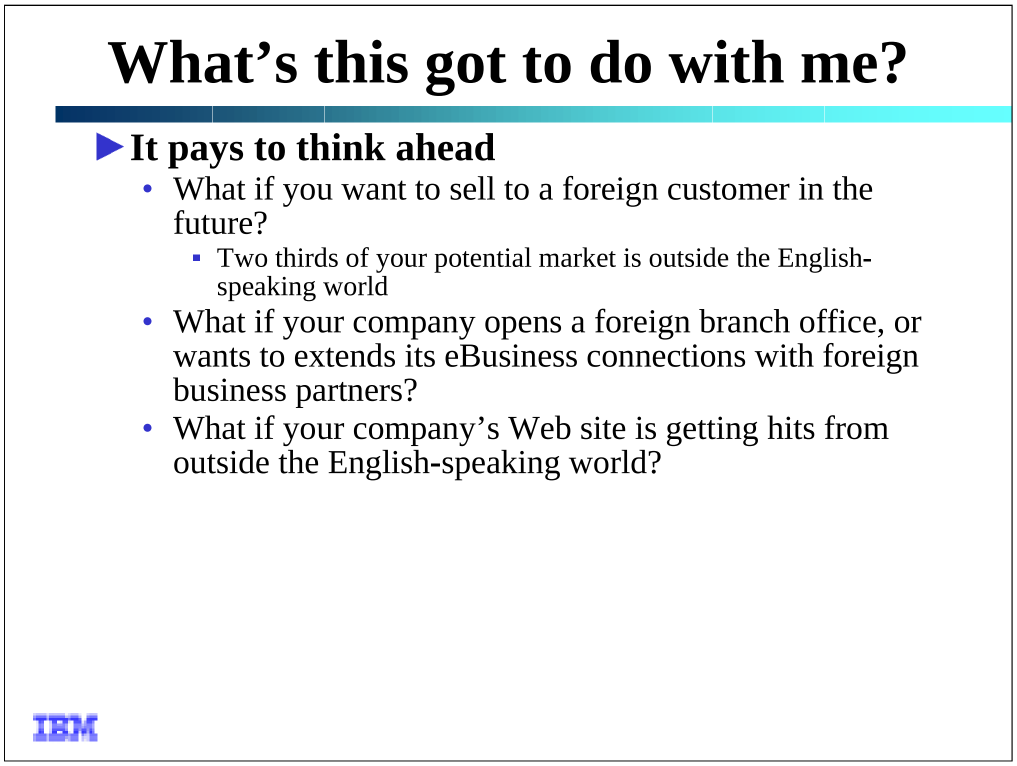
- **Internationalization is not a feature!**
  - People expect your product to "just work"
  - Many users will not accept a program that doesn't let them work in their native language
  - Even if they can handle a program with an English user interface, users will not accept a program that doesn't let them process *data* that's in their own language

IBM

If you're thinking to yourself "But none of my clients has asked for internationalization," you're right. A phrase I liked from the last Java tutorial is "Internationalization isn't a feature." It isn't. You're never going to have a client come up to you and say "I want internationalization." You may not even have the client say "I want this program to work in Japanese."

Clients just expect the software to "work right." They might not even be able to define that unless they're looking at something that "doesn't work right," but it's too late if you wait until then. An English-only product closes you off to a great many users. Even English-speaking users will have their productivity undermined by being forced to operate the product in something other than their native language. Making sure the product is localized is just part of designing a good user interface.

And, of course, even they can accept an English user interface, an overseas business will likely be processing data that's in the language where the company is located. If your product can't handle data in a particular language, you're dead in the water in the countries that speak that language.

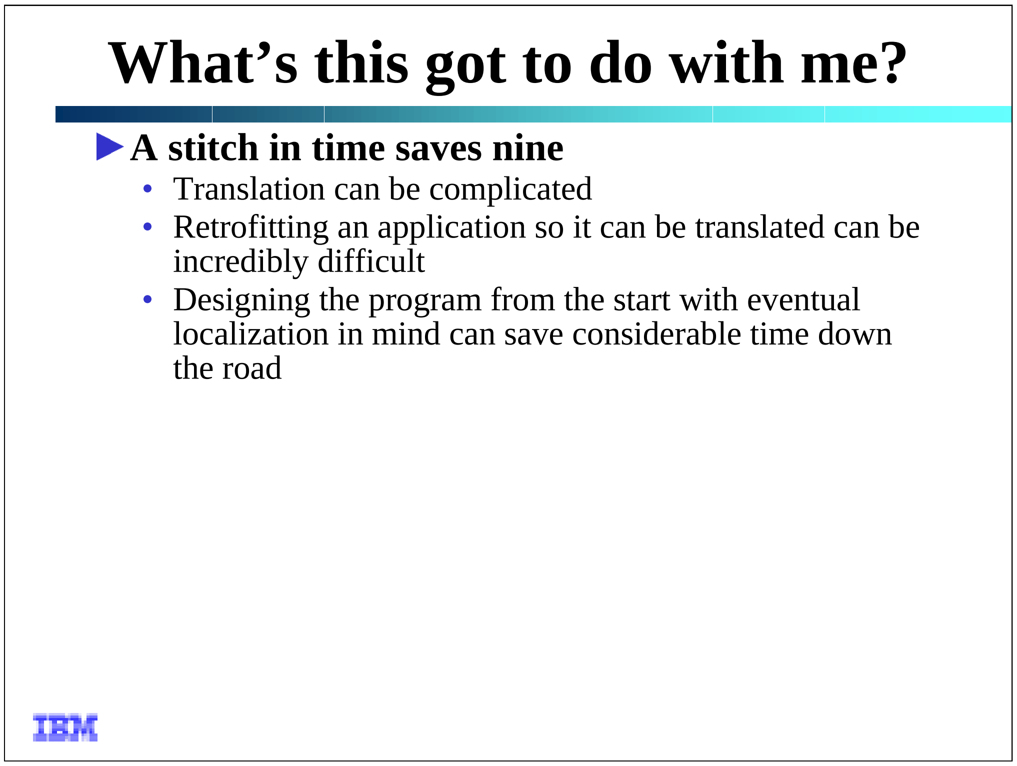


What's this got to do with me?

- **It pays to think ahead**
  - What if you want to sell to a foreign customer in the future?
    - Two thirds of your potential market is outside the English-speaking world
  - What if your company opens a foreign branch office, or wants to extends its eBusiness connections with foreign business partners?
  - What if your company's Web site is getting hits from outside the English-speaking world?

And if you're thinking this material doesn't apply to you because you don't expect to have anyone in foreign countries using the software you're producing, be careful. It pays to think ahead.

What if you land a new client in the future that's based in a foreign country? What if one of your existing clients expands his operation to a foreign country? What if you wind up with suppliers or other business partners in other countries and you want to be able to do electronic commerce with them? What if your Web site is getting lots of hits from overseas, or what if you want your Web site to get lots of hits from overseas? The thing you definitely don't want to do is write your application in such a way as to prevent translating it into other languages in the future. Making sure your program is internationalized doesn't mean you *have* to localize it right away.



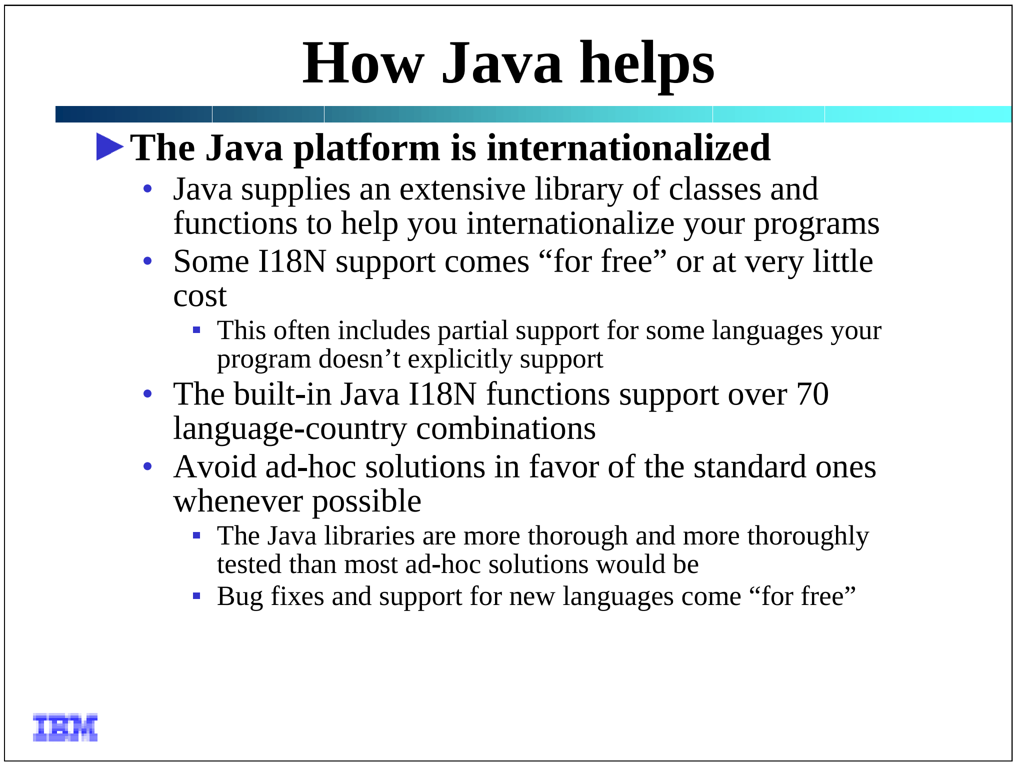
What’s this got to do with me?

- **A stitch in time saves nine**
  - Translation can be complicated
  - Retrofitting an application so it can be translated can be incredibly difficult
  - Designing the program from the start with eventual localization in mind can save considerable time down the road

Again, it pays to think ahead. You may not have to worry about this stuff now, but if in three years one of your customers comes back to you and says “we’re opening a branch office in France. Can you make your application work in French?” you’re in a lot of trouble if you haven’t prepared.

We’ve been getting a lot of hype about the Year 2000 Problem lately, and a big part of what has made this so difficult for IT people to deal with is that they can only deal with it by going through their program’s source code line by line looking for problems. This is incredibly time consuming and error-prone. Retrofitting a program to support foreign-language data or to allow translation of its user interface is exactly the same kind of problem. You want to avoid this as much as possible.

Again, internationalization is not a feature. It’s often an unspoken requirement.



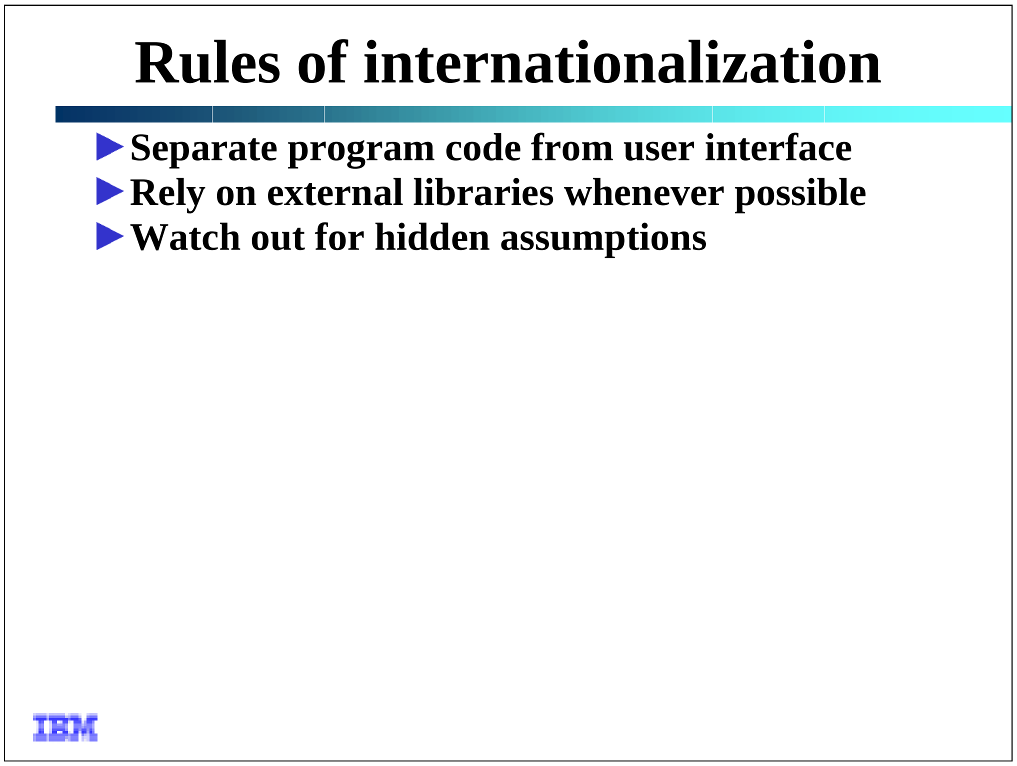
How Java helps

- **The Java platform is internationalized**
  - Java supplies an extensive library of classes and functions to help you internationalize your programs
  - Some I18N support comes “for free” or at very little cost
    - This often includes partial support for some languages your program doesn’t explicitly support
  - The built-in Java I18N functions support over 70 language-country combinations
  - Avoid ad-hoc solutions in favor of the standard ones whenever possible
    - The Java libraries are more thorough and more thoroughly tested than most ad-hoc solutions would be
    - Bug fixes and support for new languages come “for free”

IBM

You’re already several steps into the game if you’re writing in Java. The Java platform is itself internationalized, so you get some degree of internationalization in your code “for free” or for very little incremental work. One good thing this means is that you can get partial support even for languages and countries you haven’t specifically localized for. In fact, the internationalized functions in Java currently support over 70 language-country combinations.

Java provides you with almost everything you need to write properly internationalized software. The main thing you have to remember is to use the right APIs in the right way. Be sure to keep this in mind if you find yourself writing your own international support. If there’s a way to do it with the standard libraries in Java, you’ll save a lot of work and pick up bug fixes and additions of new languages “for free.”

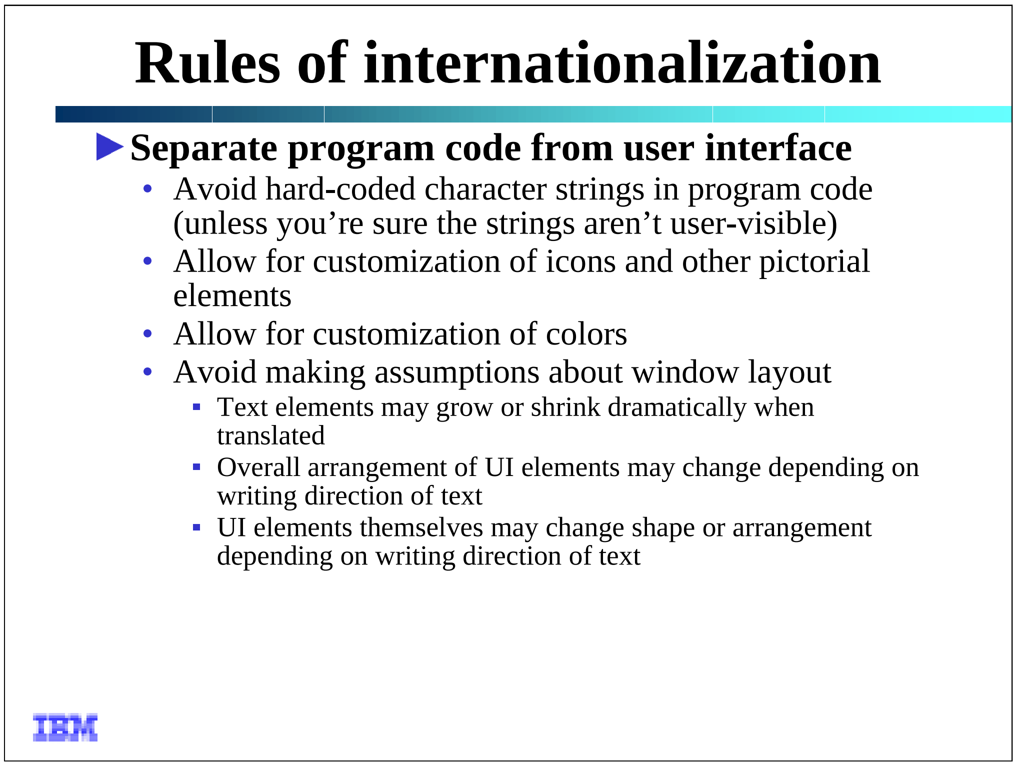


Rules of internationalization

- Separate program code from user interface
- Rely on external libraries whenever possible
- Watch out for hidden assumptions

IBM

So let’s go back and look at just what you have to do to make sure your program is internationalized. The basic idea is to keep your program’s internal logic separated from your program’s user interface: keep user-interface data (labels and messages, pictures, window layouts, etc.) out of program code, keep UI code separate from internal processing code, take advantage of all the UI code your operating environment and external libraries give you, and be careful to keep hidden assumptions about locale and UI out of your internal processing code.



Rules of internationalization

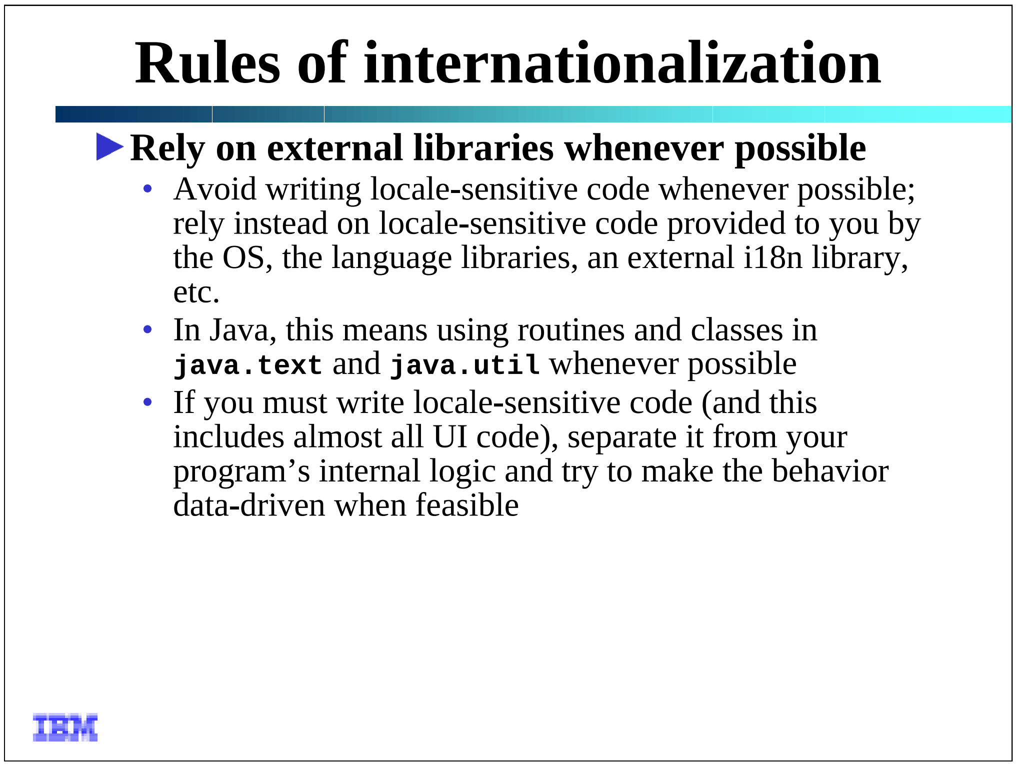
- **Separate program code from user interface**
  - Avoid hard-coded character strings in program code (unless you're sure the strings aren't user-visible)
  - Allow for customization of icons and other pictorial elements
  - Allow for customization of colors
  - Avoid making assumptions about window layout
    - Text elements may grow or shrink dramatically when translated
    - Overall arrangement of UI elements may change depending on writing direction of text
    - UI elements themselves may change shape or arrangement depending on writing direction of text

IBM

Separating program from UI is the first cardinal rule of internationalization. As much as humanly possible, keep the data driving the UI separate from the code.

In particular, be **very** careful about using hard-coded strings in your program code. This is only legal when the strings are completely internal, such as identifiers and tags the user doesn't see. Along the same lines, don't use hard-coded references to icons and pictures, and avoid hard-coded colors.

Window layout can also change based on language. The two big reasons for this are text growing or shrinking when translated and layout being affected by writing direction. An English message can get much smaller when translated into Japanese and much larger when translated into Italian, for example, requiring resizing or rearrangement of various UI elements. Hebrew and Arabic are written from right to left, so speakers of those languages usually expect windows to be arranged in a mirror-image fashion compared to the normal English layout.



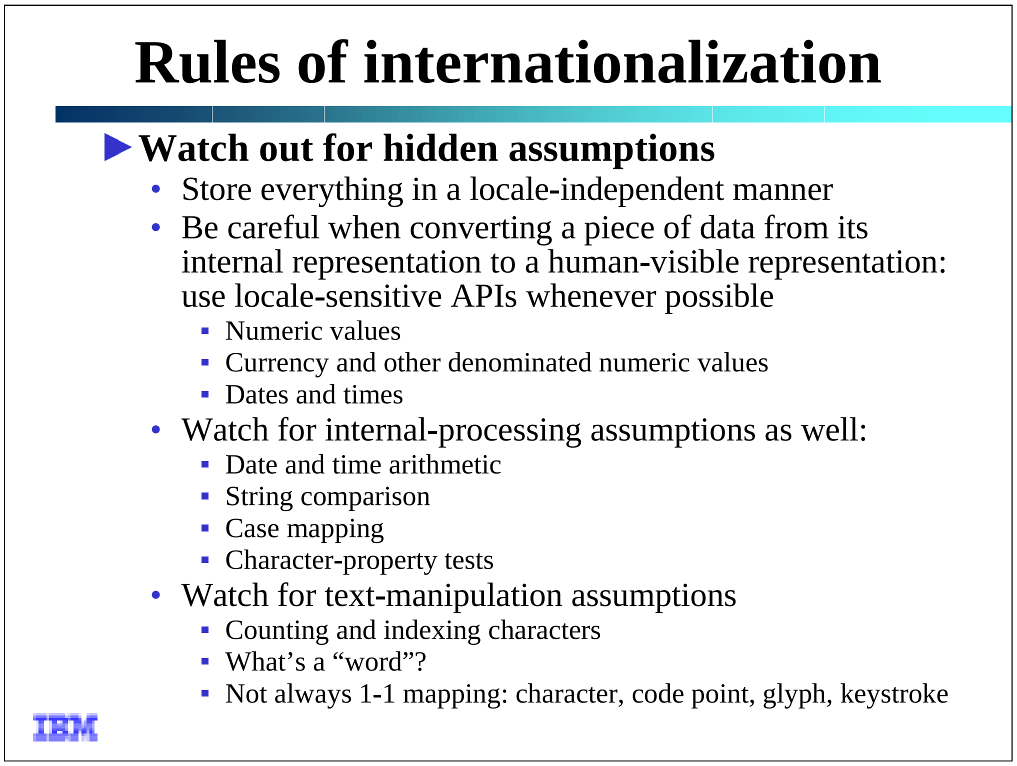
Rules of internationalization

- **Rely on external libraries whenever possible**
  - Avoid writing locale-sensitive code whenever possible; rely instead on locale-sensitive code provided to you by the OS, the language libraries, an external i18n library, etc.
  - In Java, this means using routines and classes in **`java.text`** and **`java.util`** whenever possible
  - If you must write locale-sensitive code (and this includes almost all UI code), separate it from your program's internal logic and try to make the behavior data-driven when feasible

IBM

Proper support for various languages can often be quite involved, so it's usually best to take advantage of whatever international support is provided to you by your operating environment. In Java, think very hard before not using the locale-sensitive APIs for something, and think especially hard before using Java's locale-independent APIs (e.g., Integer.toString(), Integer.valueOf(String), String.compare(), String.equals(), String.indexOf(), etc.).

If you need locale-specific capabilities that Java doesn't provide you and find yourself implementing them yourself, keep them separate from the rest of your program logic, and allow for graceful degradation when you're operating in a language they weren't designed for. Whenever feasible, use a data-driven model: make the code flexible and locale-independent, and have it look to external data for instructions on how to behave.

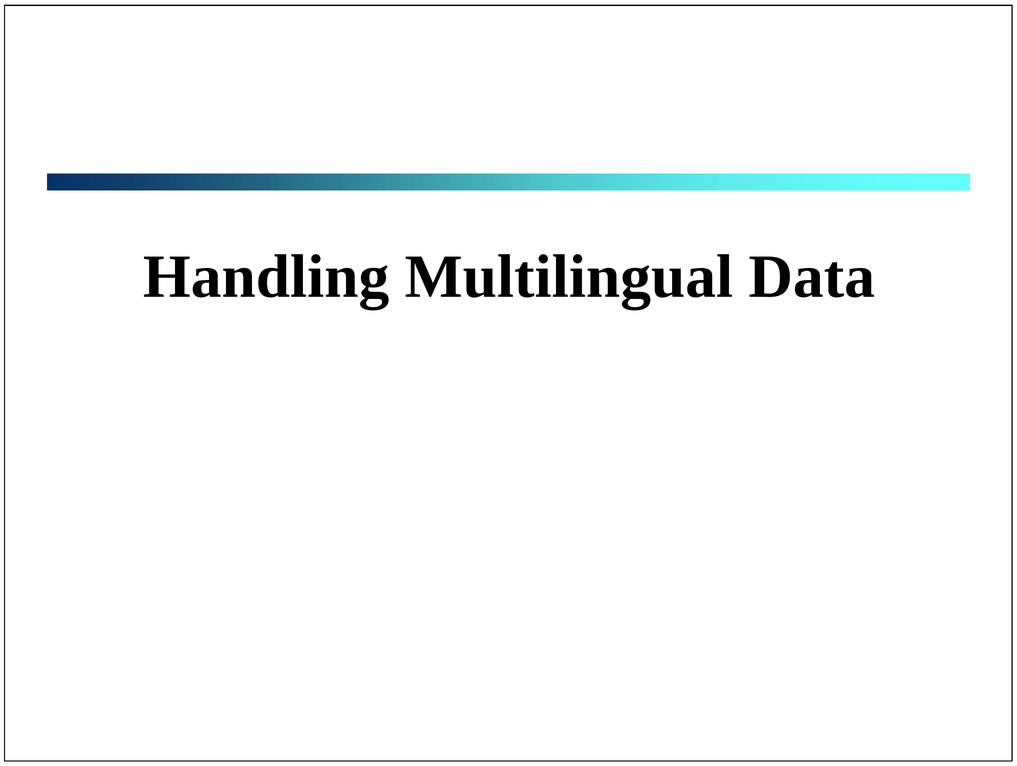


Rules of internationalization

- **Watch out for hidden assumptions**
  - Store everything in a locale-independent manner
  - Be careful when converting a piece of data from its internal representation to a human-visible representation: use locale-sensitive APIs whenever possible
    - Numeric values
    - Currency and other denominated numeric values
    - Dates and times
  - Watch for internal-processing assumptions as well:
    - Date and time arithmetic
    - String comparison
    - Case mapping
    - Character-property tests
  - Watch for text-manipulation assumptions
    - Counting and indexing characters
    - What’s a “word”?
    - Not always 1-1 mapping: character, code point, glyph, keystroke

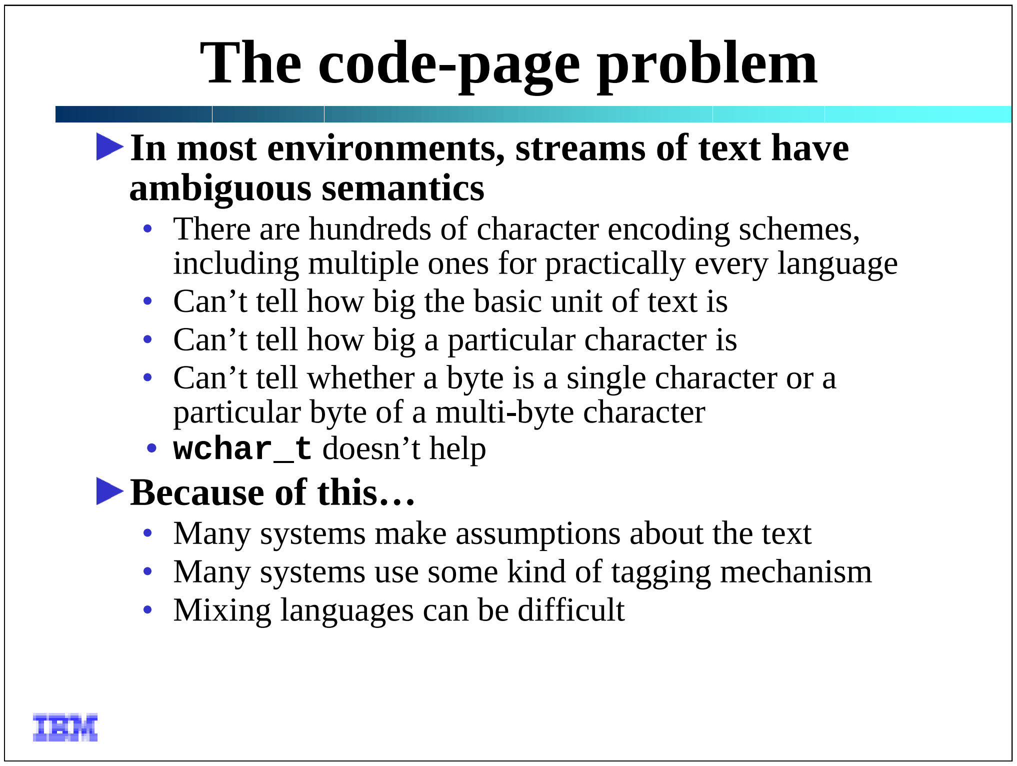
The trickiest internationalization rule is to be on guard against hidden assumptions in your program’s internal processing logic. Make sure your internal storage formats are locale-independent. Be careful when converting between internal storage formats and human-readable formats (and don’t confuse the two). Watch out for naive algorithms for case conversion, string comparison, date arithmetic, and so on. Don’t build up user-visible message by concatenating strings together. And when processing text, keep in mind that there often isn’t a one-to-one mapping between what the user sees as a single character (a “grapheme”), a shape that gets drawn on the screen (a “glyph”), a single input keystroke, and a single storage location within a String. Also keep in mind that units of text such as “words,” “sentences,” and “characters” have definitions that are language-specific.

The most common trouble spots are multilingual text, numbers (especially numbers that carry implicit denominations with them), and dates and times, but there are many potential others. It’s likely you won’t see some of hidden assumptions until somebody complains about them, but avoid the common cases do your best to minimize the others.



Handling Multilingual Data

As I said in the introduction, probably the most important barrier to international use of a computer program is its incompatibility with the data used in a particular place. Let’s talk a little about handling multilingual data properly.



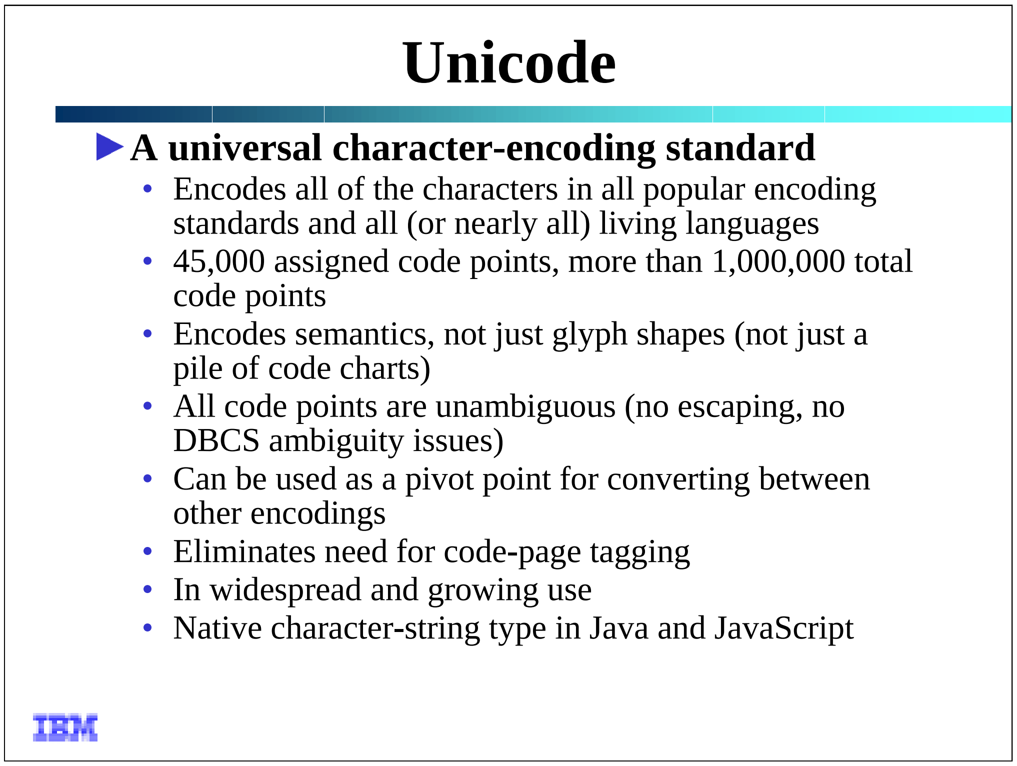
The code-page problem

- **In most environments, streams of text have ambiguous semantics**
  - There are hundreds of character encoding schemes, including multiple ones for practically every language
  - Can't tell how big the basic unit of text is
  - Can't tell how big a particular character is
  - Can't tell whether a byte is a single character or a particular byte of a multi-byte character
  - `wchar_t` doesn't help
- **Because of this…**
  - Many systems make assumptions about the text
  - Many systems use some kind of tagging mechanism
  - Mixing languages can be difficult

In most operating environments, character strings have ambiguous semantics. Since most character encodings are based on single-byte values, you have to have a different encoding (or "code page" in many environments) for every language. There are hundreds of code pages and other encodings in use out there. In fact, there are probably *at least* three or four for every single language. It may be okay to assume that most of them are compatible with 7-bit ASCII, but that's certainly not true of all of them.

To most processes, especially language compilers, that means a character string is just a sequence of arbitrary bytes. This usually means that programs make assumptions as to which character set they're using and expect everything they're communicating with to use the same character set. The way around this is to have some sort of tagging mechanism to identify the character set for a group of characters.

This is a major hindrance to mixing languages in a single file or database. Making assumptions just plain prevents this, and tagging mechanisms add a lot of bookkeeping or limit the granularity of the taggable units (e.g., a whole field might have to be in one language).



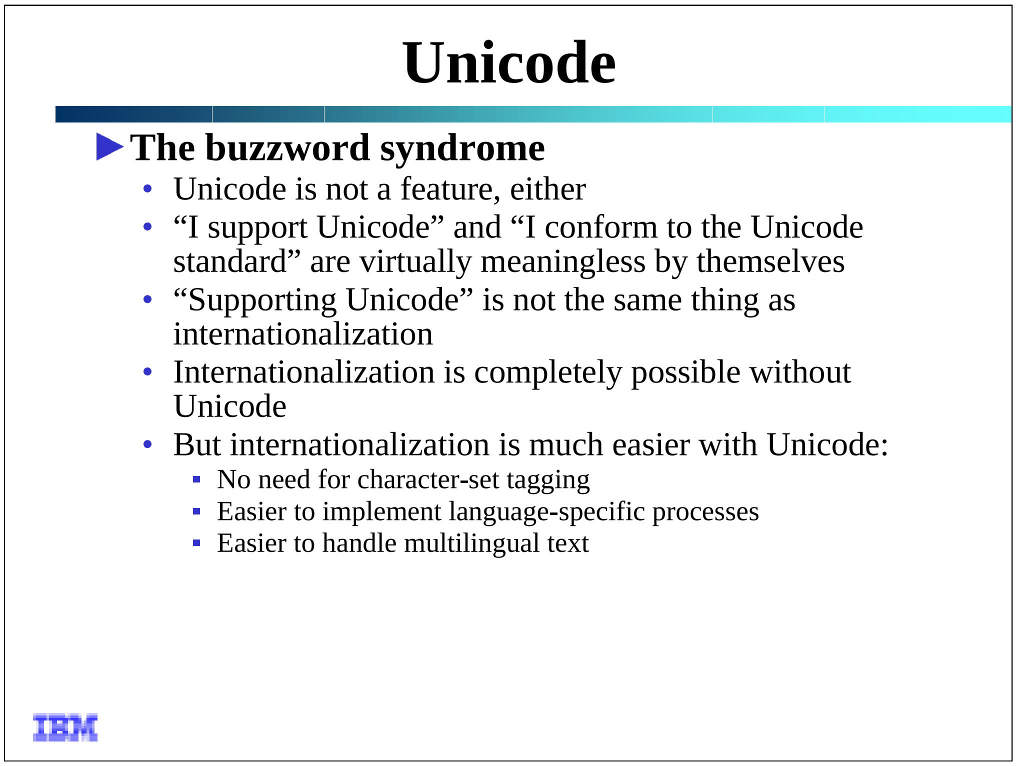
Unicode

- **A universal character-encoding standard**
  - Encodes all of the characters in all popular encoding standards and all (or nearly all) living languages
  - 45,000 assigned code points, more than 1,000,000 total code points
  - Encodes semantics, not just glyph shapes (not just a pile of code charts)
  - All code points are unambiguous (no escaping, no DBCS ambiguity issues)
  - Can be used as a pivot point for converting between other encodings
  - Eliminates need for code-page tagging
  - In widespread and growing use
  - Native character-string type in Java and JavaScript

Unicode, of course, was designed mainbly to solve this very problem. Unicode is a universal character encoding standard, comprising numeric values for some 45,000 characters (with room for more than a million more). All characters in all commonly-used (and most not-so-commonly-used) character encodings are included, as are the characters needed to write virtually all living languages. All are unambiguously encoded, so there's never a question as to which character a particular pattern of bits represents: there are strict limits on how much context a process has to look at to process a particular character--usually it's just that character. Unicode isn't just a pile of code charts; it also includes an extensive set of rules defining what well-formed Unicode text looks like and exactly how a particular code point is to be interpreted: it encodes semantics, not just glyph shapes.

Unicode text is generally easier to process than text in other encodings, and because it includes a huge multitude of characters, it eliminates the need to keep track not only of the characters themselves, but of which encoding scheme was used to encode them.

Unicode is in widespread and growing use. Most newer programming languages (including both Java and JavaScript) are being designed with Unicode as their native character-string format, and Unicode support is appearing in more and more operating systems and applications. Microsoft Windows NT 4.0 and Office 97, for example, support Unicode well. All IBM products, both OSes and applications, are being upgraded to handle Unicode correctly as well.



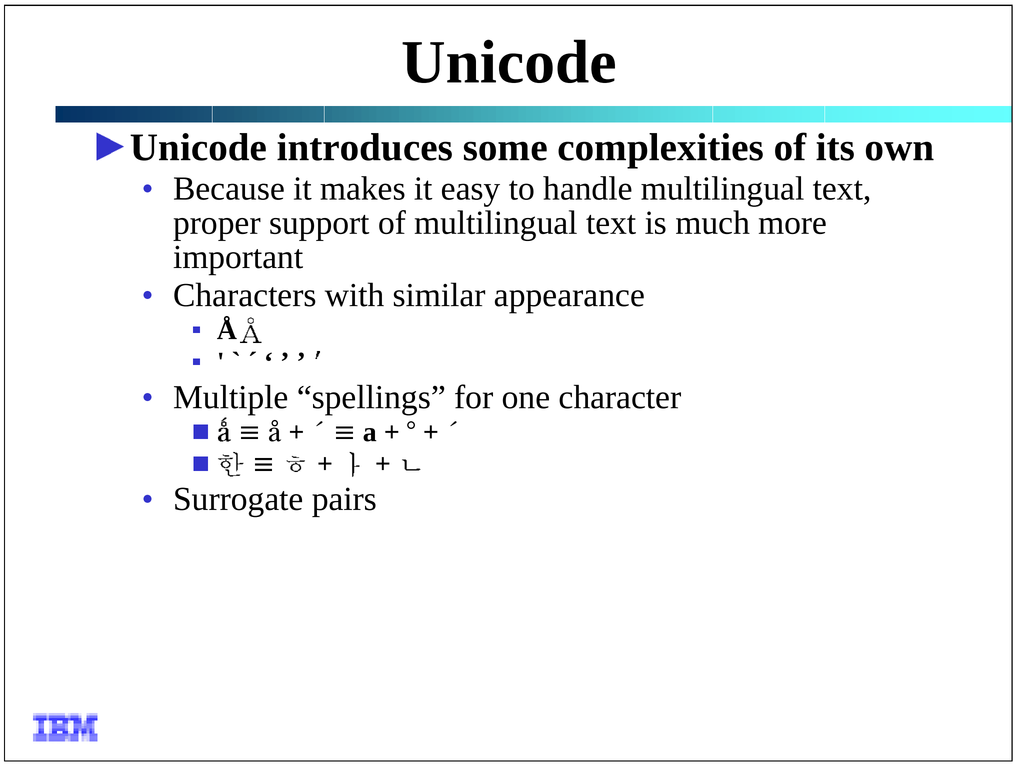
Unicode

- **The buzzword syndrome**
  - Unicode is not a feature, either
  - “I support Unicode” and “I conform to the Unicode standard” are virtually meaningless by themselves
  - “Supporting Unicode” is not the same thing as internationalization
  - Internationalization is completely possible without Unicode
  - But internationalization is much easier with Unicode:
    - No need for character-set tagging
    - Easier to implement language-specific processes
    - Easier to handle multilingual text

The computer industry often falls prey to the “buzzword syndrome”: people starting hearing some word or phrase a lot and they join a frenzy to do something with the word or phrase without bothering to figure out what it means first. Java is a classic example of this. Everybody has been jumping up with some way to tie their products to all of the Java hype, even when Java had nothing to do with them (JavaScript is a favorite example of mine).

Unicode is also a buzzword, although not the kind of über-buzzword that Java is. So it’s important to remember that Unicode is not a feature any more than internationalization is. It’s a means to an end: Unicode is a technology which eases many of the problems involved in implementation good internationalization support. It makes programs easier to internationalize, although internationalization is completely possible without it.

The phrase “This technology supports Unicode” is relatively meaningless. The conformance requirements in the Unicode standard are relatively simple: the main key is that the Unicode standard doesn’t require support for any particular character or set of characters. It basically requires that you follow the rules for any character that you’re claiming to support, and that you not mess up Unicode text you’re passing through to another process. The key is which characters and languages you support. Making sure your string elements are 16 bits wide is far from a complete internationalization solution.



Unicode

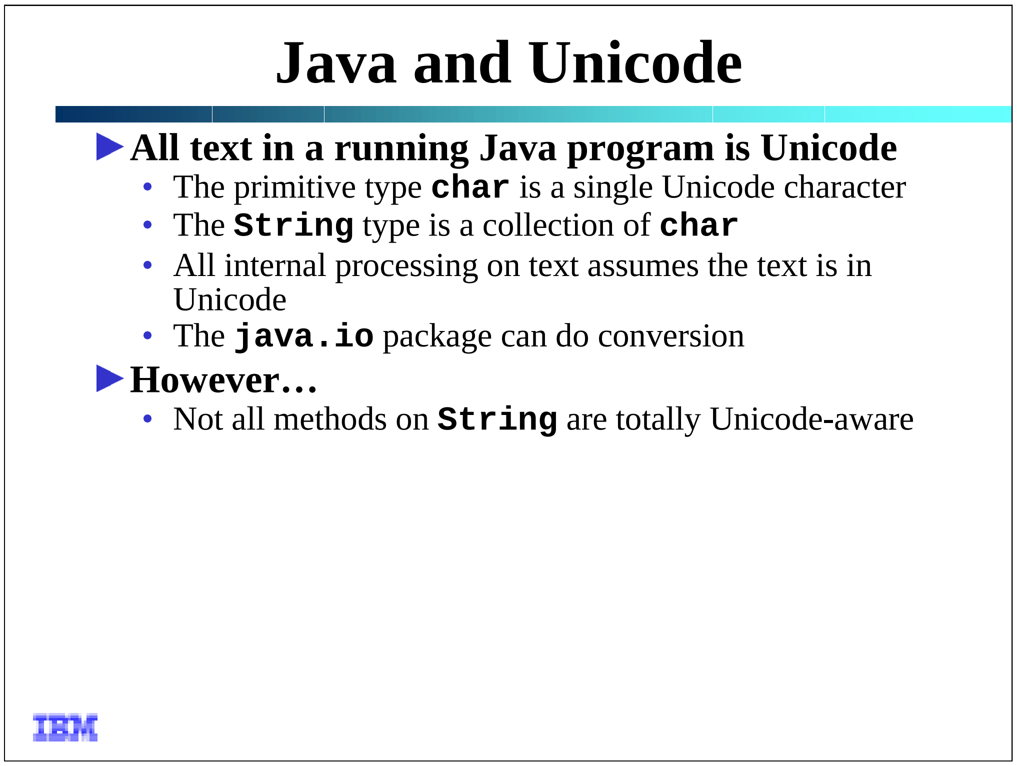
- **Unicode introduces some complexities of its own**
  - Because it makes it easy to handle multilingual text, proper support of multilingual text is much more important
  - Characters with similar appearance
    - **Å** Å
    - ' ` ´ ‘ ’ ’ ′
  - Multiple “spellings” for one character
    - ǻ ≡ å + ´ ≡ **a** + ° + ´
    - 한 ≡ ㅎ + ㅏ + ㄴ
  - Surrogate pairs

Of cuorse, because Unicode makes it possible to mix text in different languages freely, people will start mixing text in different languages freely, increasing the challenge of doing certain things to the text.

In addition, to maintain compatibility with various other encodings, Unicode often has several ways of saying the same thing. In many cases, for example, there are groups of Unicode characters with the same or similar visual appearances. In the first line, for example, the first character is an A with a ring over it. The second character is the symbol for the Angstrom unit.

In the second line, we have a selection of marks that look kind of like apostrophes and quote marks. The first mark is the ASCII straight single quote, which has often been used as a substitute for all these other characters. The second and third characters are acute and grave accent marks. The fourth and fifth are opening and closing quotation marks. The sixth is an apostrophe (although it’s only supposed to be used when the character’s being used as a letter). The last character is the mathematical prime mark. Some types of searches might want to level out these differences.

Some characters can be represented either as a single code point value, or as multiple code point values that combine together into the same character. The different combinations and the single code points that can all used to mean the same thing are supposed to be treated identically. Unicode also has a special kind of combining character sequence called a “surrogate pair” where the individual units don’t have meaning by themselves, but they combine to form a single character.



Java and Unicode

- **All text in a running Java program is Unicode**
  - The primitive type **`char`** is a single Unicode character
  - The **`String`** type is a collection of **`char`**
  - All internal processing on text assumes the text is in Unicode
  - The **`java.io`** package can do conversion
- **However…**
  - Not all methods on **`String`** are totally Unicode-aware

IBM

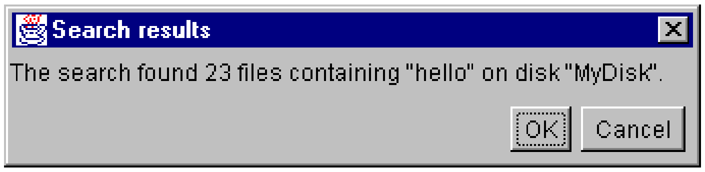
A few slides ago, I mentioned why all this information is relevant: Java's native character encoding is Unicode. Not only is the char type a 16-bit quantity; it's specifically required to represent a Unicode character. This eliminates all the headaches of handling mu8ltiple encodings in a program…

…except, of course, dealing with text coming from outside (or going outside), such as when you're reading a disk file that contains text or receiving text over a network connection. The Java I/O framework automatically handles these kinds of conversions so the rest of the program doesn't have to worry about it.

I should point out, however, that many of the methods on String aren't Unicode aware and just treat the string as a sequence of unsigned 16-bit values. This is fine, but you have to remember to avoid these functions when you're dealing with multilingual text (or make sure you use them right).

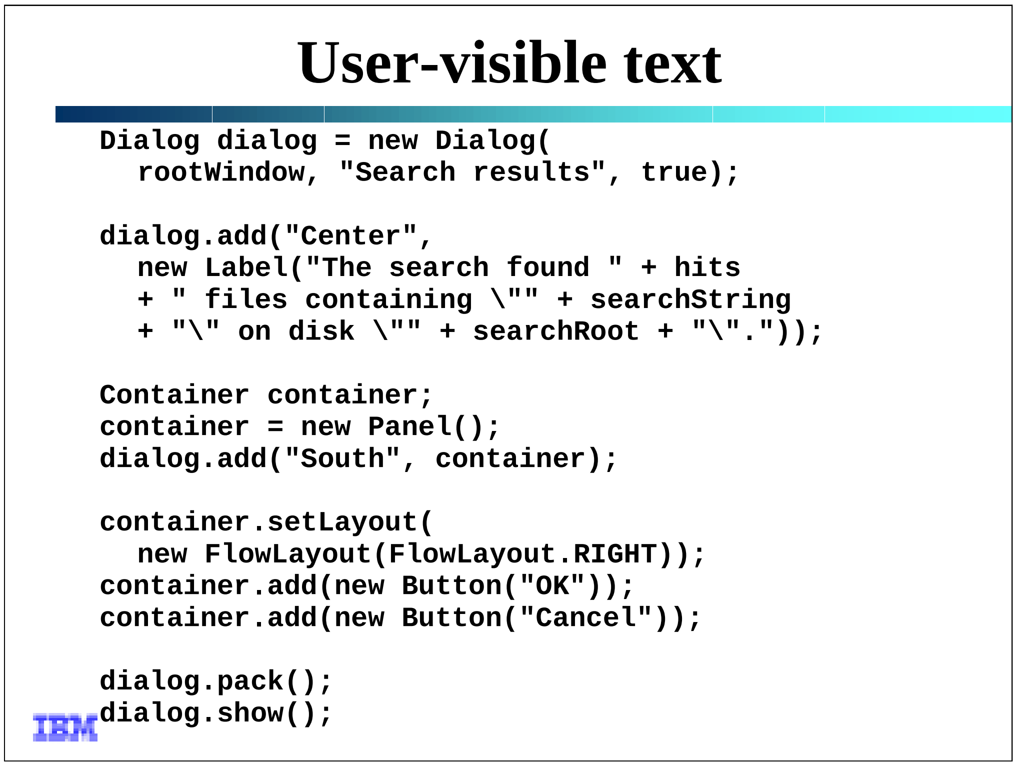
Handling User-Visible Text

User-visible text



The search found 23 files containing "hello" on disk "MyDisk".

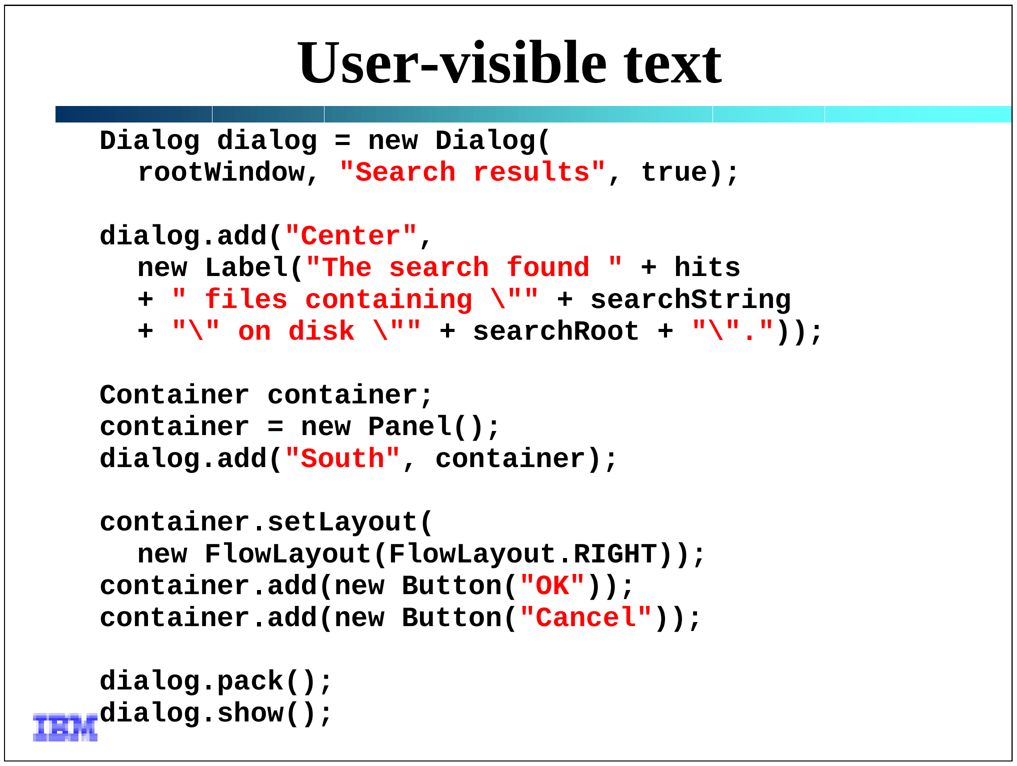
Okay, having gotten a bird's-eye view of everything now, let's delve in and take a close look at internationalization. Say we want our program to display a dialog that looks like this...



User-visible text

```
Dialog dialog = new Dialog(
  rootWindow, "Search results", true);

dialog.add("Center",
  new Label("The search found " + hits
  + " files containing \"" + searchString
  + "\" on disk \"" + searchRoot + "\"."));

Container container;
container = new Panel();
dialog.add("South", container);

container.setLayout(
  new FlowLayout(FlowLayout.RIGHT));
container.add(new Button("OK"));
container.add(new Button("Cancel"));

dialog.pack();
dialog.show();
```

The code to do this usually looks something like this. Now remember that we said in the introduction that hard-coded strings in the source code are a Bad Thing. We have a lot of hard-coded strings here...

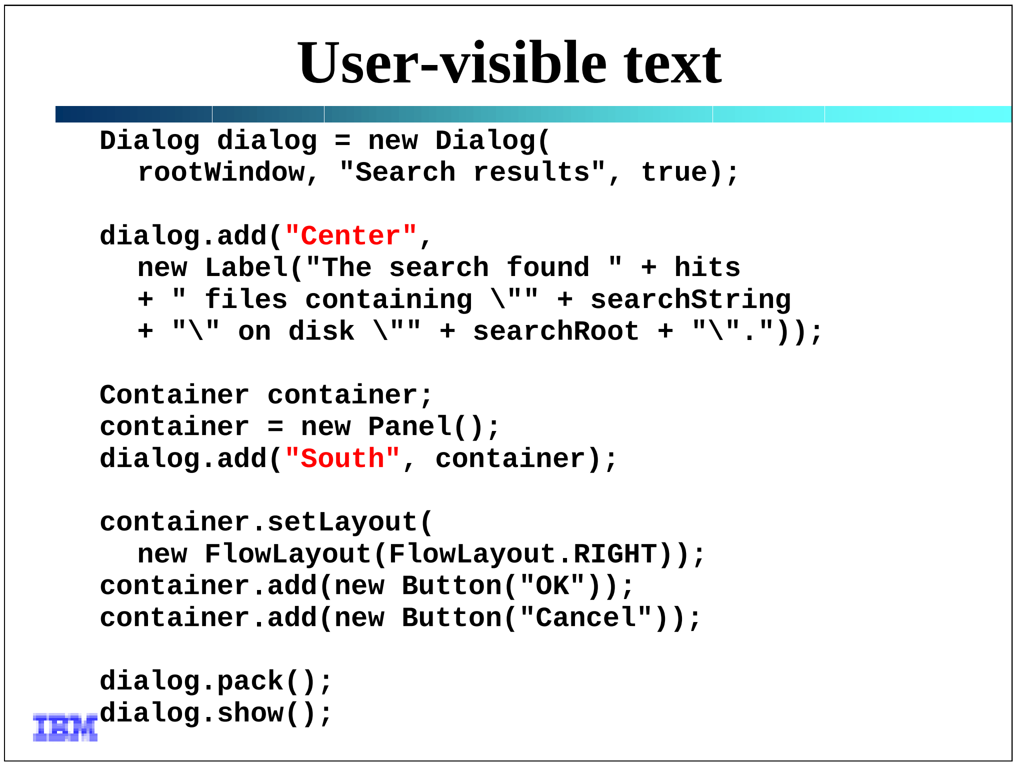


User-visible text

```
Dialog dialog = new Dialog(
  rootWindow, "Search results", true);

dialog.add("Center",
  new Label("The search found " + hits
  + " files containing \"" + searchString
  + "\" on disk \"" + searchRoot + "\"."));

Container container;
container = new Panel();
dialog.add("South", container);

container.setLayout(
  new FlowLayout(FlowLayout.RIGHT));
container.add(new Button("OK"));
container.add(new Button("Cancel"));

dialog.pack();
dialog.show();
```

If you have to go back and translate this program into French, you would have to go through many functions like this one, manually searching for hard-coded strings, then translate all of them and recompile the program.

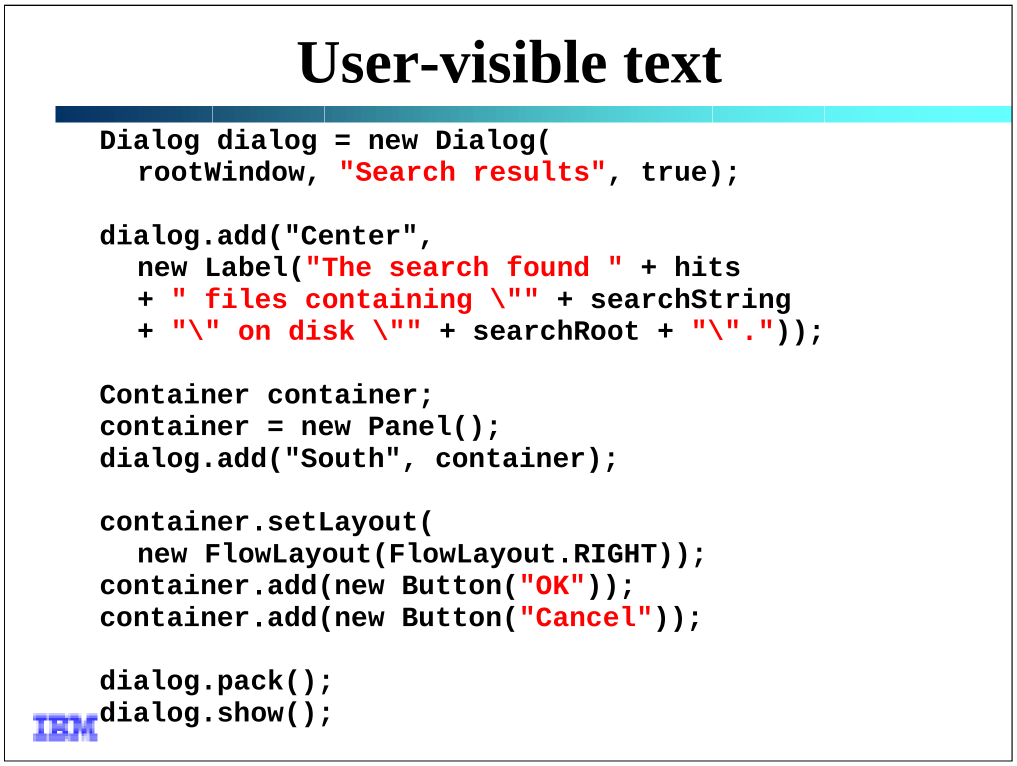
This is painstakingly difficult. It'd be much better if the program source code could stay the same and the strings could actually come from some kind of data file somewhere else. This also has the advantage of collecting everything that needs to be translated into one place.



User-visible text

```
Dialog dialog = new Dialog(
  rootWindow, "Search results", true);

dialog.add("Center",
  new Label("The search found " + hits
  + " files containing \"" + searchString
  + "\" on disk \"" + searchRoot + "\"."));

Container container;
container = new Panel();
dialog.add("South", container);

container.setLayout(
  new FlowLayout(FlowLayout.RIGHT));
container.add(new Button("OK"));
container.add(new Button("Cancel"));

dialog.pack();
dialog.show();
```

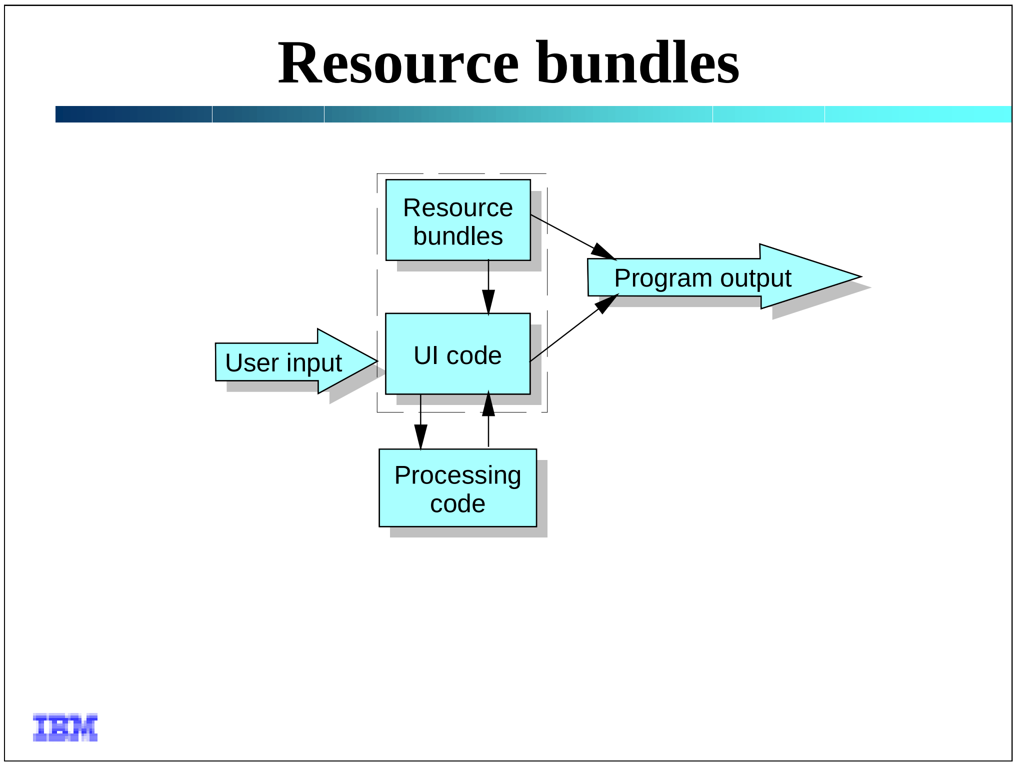
But do we really want to do that with *every* string in this example? Consider "Center" and "South" in our example code and look at the title of the slide. The stuff we need to worry about, and the stuff we need to translate, is user-visible text. These two strings aren't user-visible. They're internal program IDs used to tell the layout manager here to position a new component that's being added. You'll run into this kind of thing in a fair number of places in an average Java program. Strings are often used as internal identifiers to allow an open-ended set of identifiers, something which is very difficult with integers or other types. You definitely don't want to translate these strings; if you do, the program won't work anymore. So these get left alone.



User-visible text

```
Dialog dialog = new Dialog(
  rootWindow, "Search results", true);

dialog.add("Center",
  new Label("The search found " + hits
  + " files containing \"" + searchString
  + "\" on disk \"" + searchRoot + "\"."));

Container container;
container = new Panel();
dialog.add("South", container);

container.setLayout(
  new FlowLayout(FlowLayout.RIGHT));
container.add(new Button("OK"));
container.add(new Button("Cancel"));

dialog.pack();
dialog.show();
```

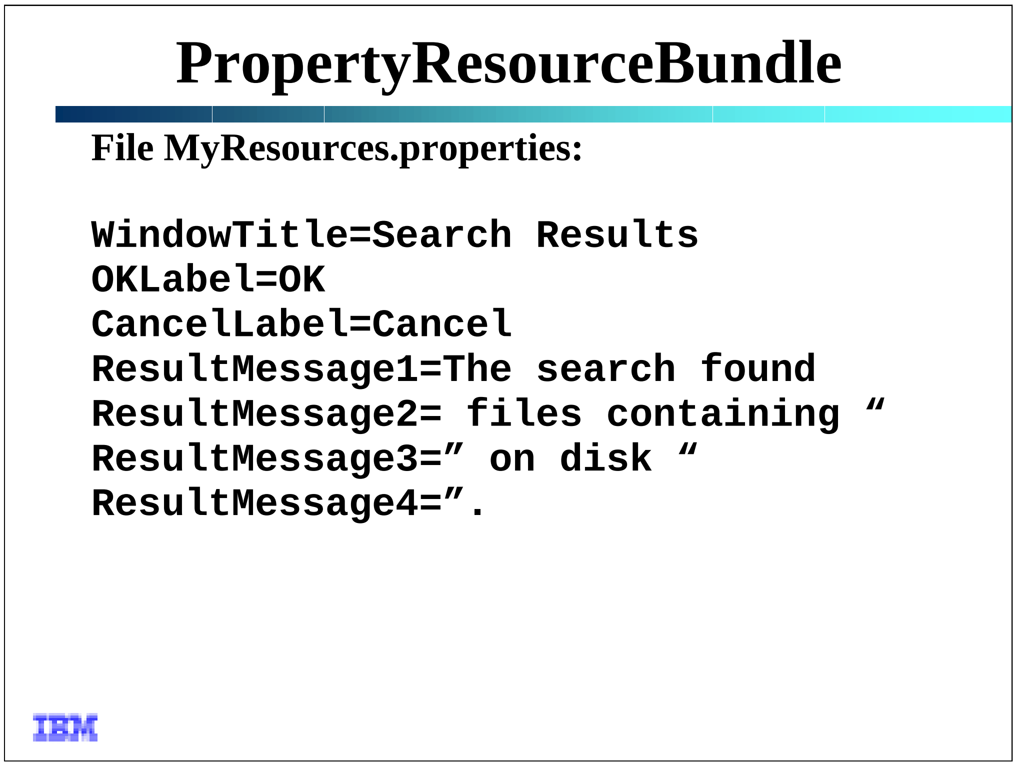
All of the other strings, on the other hand, are user-visible text: “Search results” is the window title, “OK” and “Cancel” are the button labels, and the others make up the message the dialog box displays. All of these must be translated in order to make the dialog box intelligible to a non-English speaker. These are the strings you want to get out of your program code and into some central external repository.



Resource bundles

In Java, that repository is called a resource bundle. This is similar to a message catalog in C, but much more flexible. Resource bundles can contain not only messages and other user-visible strings, but icons and pictures, actual UI elements like menus and buttons, and even whole window layouts. The program’s UI code draws on the items stored in a resource bundle to produce the program’s output and user interface.

Java provides an abstract ResourceBundle object that represents a resource bundle. Subclasses of ResourceBundle provide interfaces to different types of storage for the actual resource data: disk files, database repositories, network resources, or even data embedded into the ResourceBundle code itself.



PropertyResourceBundle

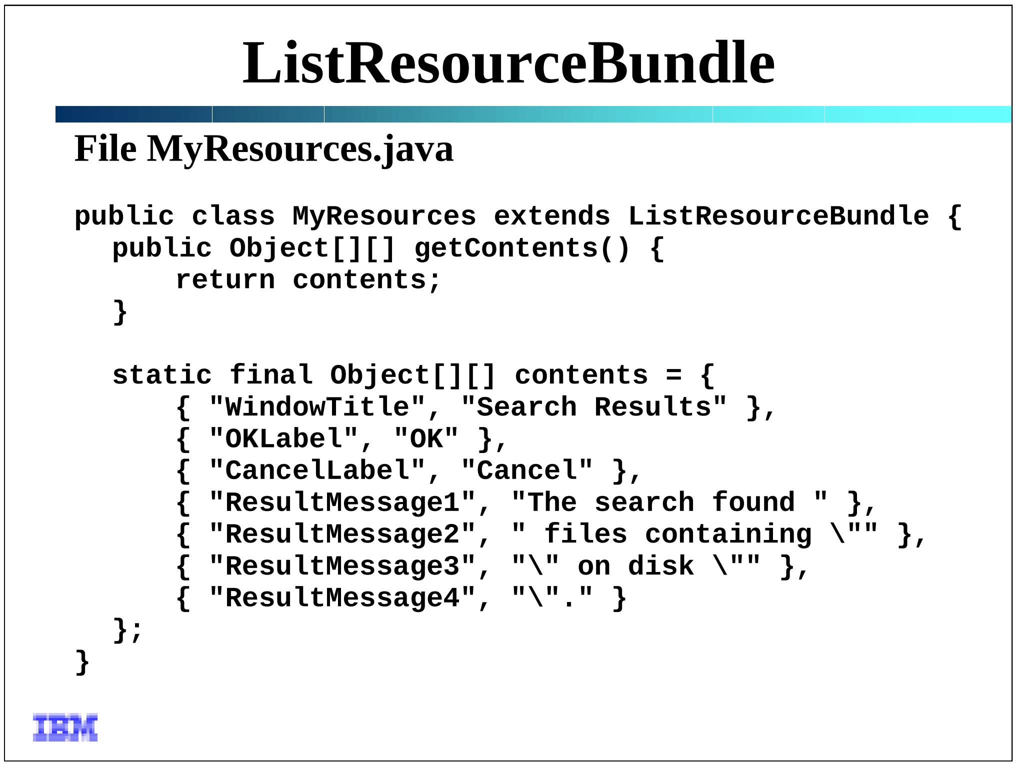
File MyResources.properties:

```
WindowTitle=Search Results
OKLabel=OK
CancelLabel=Cancel
ResultMessage1=The search found
ResultMessage2= files containing "
ResultMessage3=" on disk "
ResultMessage4=".
```

Java provides two concrete subclasses of ResourceBundle. The simpler of these is PropertyResourceBundle. PropertyResourceBundle provides an interface to access resource data in a properties file. A properties file is simply a text file containing a series of key-value pairs. The keys are separated from the values by = signs, and the key-value pairs are separated by carriage returns.

For all types of resource bundles, you give the bundle a name (it's up to you whether you want to keep all of your program's resources in a single resource bundle or spread across several). Then you assign each individual resource a programmatic ID (such as "WindowTitle" or "CancelLabel" in the example above). This ID is how the program will access the bundle (this is another case of a hard-coded string that is for internal use and must not be translated). The ID is the key and the actual resource data is the value.

Properties files are very simple, but have some serious limitations: The first is that you can only put text into a properties file, meaning there's no way to have resources of any other type. Second, there are issues with the character encoding of the file (it isn't Unicode) that make it cumbersome for text in languages that don't use the standard Western European Latin alphabet. Generally, we don't recommend using property resource bundles.

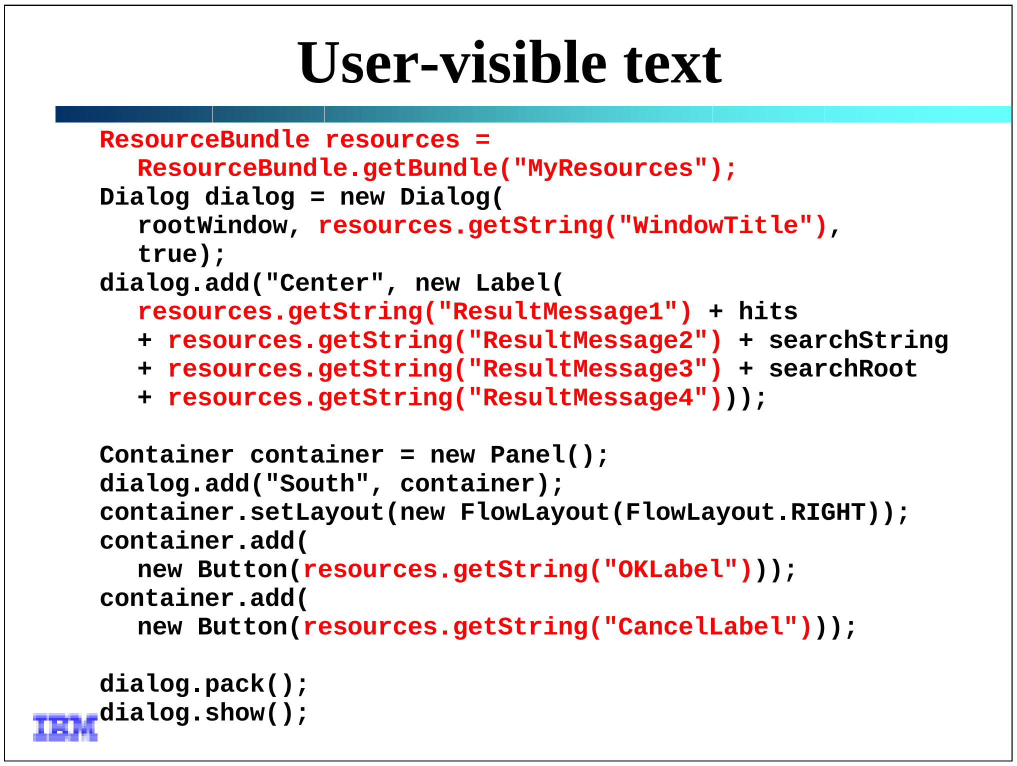


ListResourceBundle

File MyResources.java

```
public class MyResources extends ListResourceBundle {
  public Object[][] getContents() {
      return contents;
  }

  static final Object[][] contents = {
      { "WindowTitle", "Search Results" },
      { "OKLabel", "OK" },
      { "CancelLabel", "Cancel" },
      { "ResultMessage1", "The search found " },
      { "ResultMessage2", " files containing \"" },
      { "ResultMessage3", "\" on disk \"" },
      { "ResultMessage4", "\"." }
  };
}
```

The other built-in subclass of ResourceBundle is ListResourceBundle. ListResourceBundles contain the resource data as static class members. This means each list resource bundle is a new class. In essence, the resource-data file the translators mess with is the source code file itself. That means there's some extra cruft here that the translators don't need to worry about, but the file format is still pretty simple (you only mess with the contents of "contents"), and it can accommodate any character encoding and can contain any type of resource data. Again, the key-value-pair structure of a ListResourceBundle is evident here.



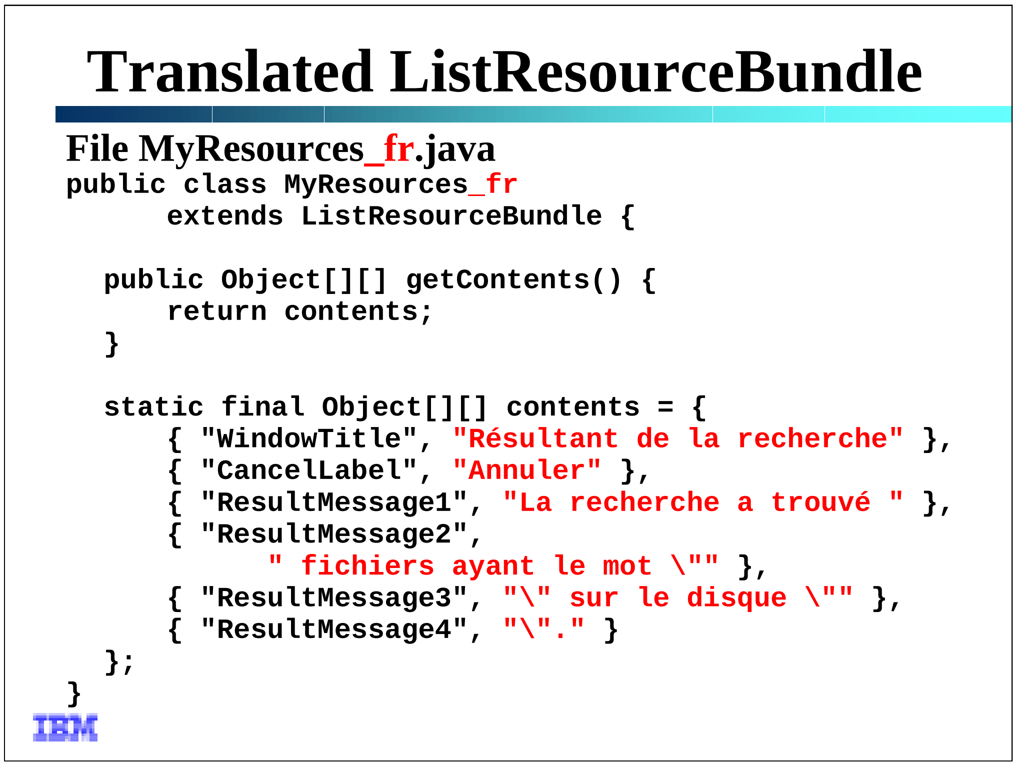
User-visible text

```
ResourceBundle resources =
  ResourceBundle.getBundle("MyResources");
Dialog dialog = new Dialog(
  rootWindow, resources.getString("WindowTitle"),
  true);
dialog.add("Center", new Label(
  resources.getString("ResultMessage1") + hits
  + resources.getString("ResultMessage2") + searchString
  + resources.getString("ResultMessage3") + searchRoot
  + resources.getString("ResultMessage4")));

Container container = new Panel();
dialog.add("South", container);
container.setLayout(new FlowLayout(FlowLayout.RIGHT));
container.add(
  new Button(resources.getString("OKLabel")));
container.add(
  new Button(resources.getString("CancelLabel")));

dialog.pack();
dialog.show();
```

Whichever type of resource bundle you’re using, your code accesses it the same way. Our original code snippet putting up the dialog box would look like this whether the resource data is in a PropertyResourceBundle, a ListResourceBundle, or some program-defined type of resource bundle. The red parts are the parts that changed from the original version of this snippet. Instead of having hard-coded strings, we have calls to fetch a particular value from the resource bundle. We also have to add a line at the beginning of the function to fetch the resource bundle itself.

I ran out of room to show it here, but the ResourceBundle APIs can throw exceptions (what if the resource bundle isn’t there, or a particular resource isn’t in it?), so this code snippet would normally be enclosed in a try-catch construct.

This code is obviously somewhat longer and harder to read, but it completely separates the code from the resource data.



Translated ListResourceBundle

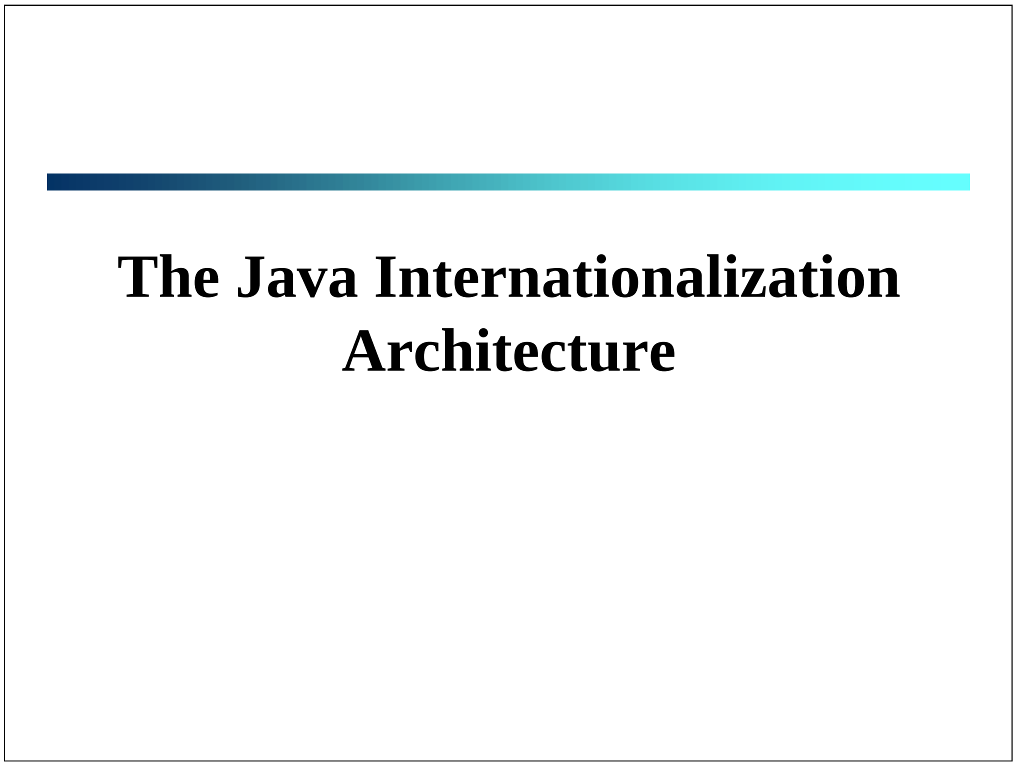
File MyResources_fr.java

```
public class MyResources_fr
      extends ListResourceBundle {

  public Object[][] getContents() {
      return contents;
  }

  static final Object[][] contents = {
      { "WindowTitle", "Résultant de la recherche" },
      { "CancelLabel", "Annuler" },
      { "ResultMessage1", "La recherche a trouvé " },
      { "ResultMessage2",
            " fichiers ayant le mot \"" },
      { "ResultMessage3", "\" sur le disque \"" },
      { "ResultMessage4", "\"." }
  };
}
```

So what happens now when you want to translate the text? Instead of going through the program source code with a fine-toothed comb looking for strings that need translating, all of those strings are collected here in one place. The translator simply copies the untranslated file and translates all the strings into his language.

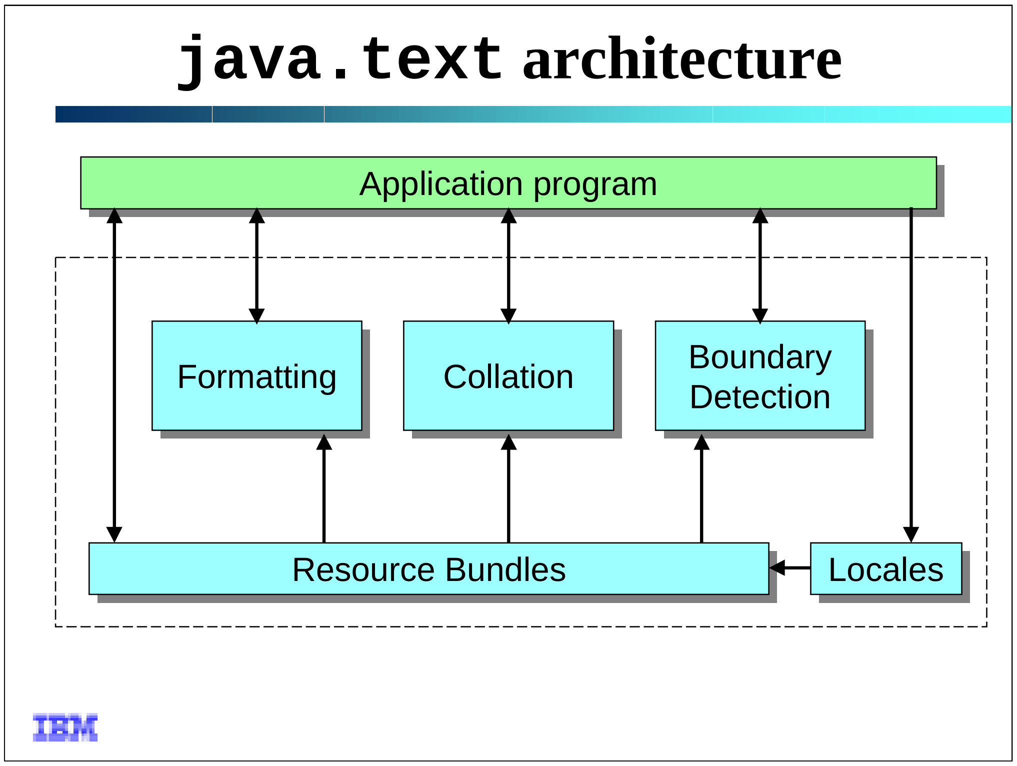
Notice that we've given the resource bundle a new name. This allows a program to carry around resources for several different languages or countries with the same executable, and allows the program to dynamically select the right resources for whatever language a particular user is using at the moment. For each variant of the same resource bundle, the new one has the original one's name tagged with a locale ID (a Locale is a Java object that identifies a particular combination of language and country [and sometimes other distinguishing characteristics]).

Notice that the definition of "OKLabel" is missing here. That's because "OK" is still "OK" when translated into French. Resource bundles are arranged in a hierarchy going from least specific to most specific. Any bundle can omit a resource and the resource-loading mechanism will automatically fall back on the more general bundle for that resource's value. In order for this "inheritance" mechanism to work right (which is especially important when you don't *have* a resource for some locale), there are certain rules you have to follow about which bundles you provide and which resources go into which ones.



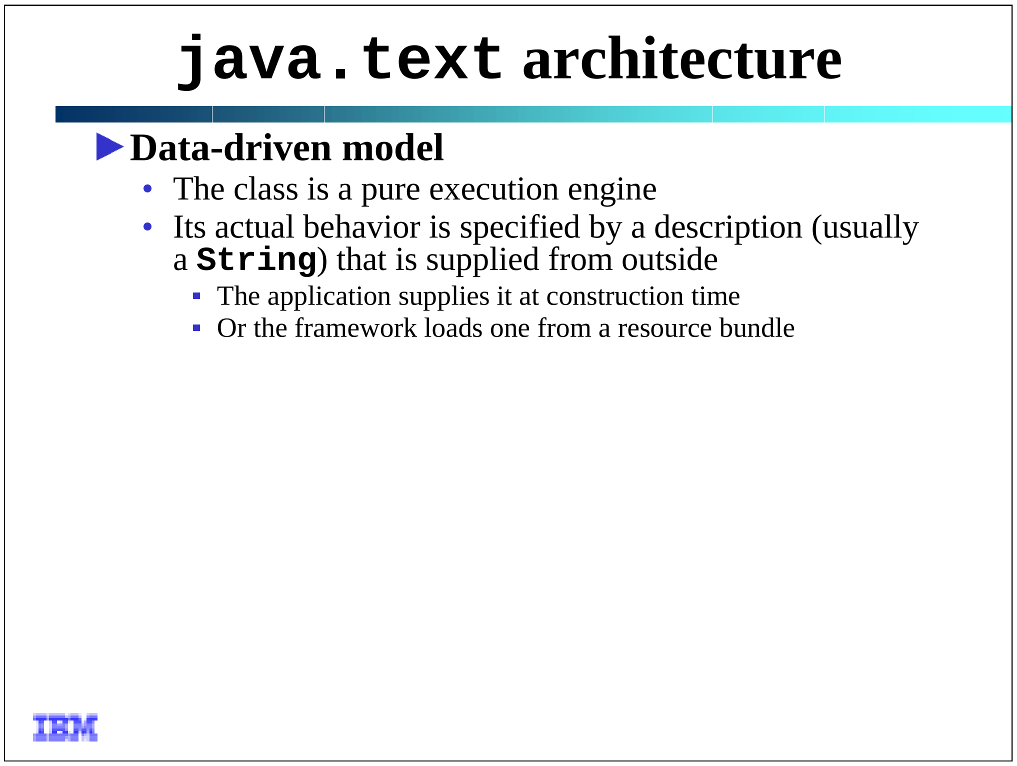
The Java Internationalization Architecture

Now that we’ve taken a few minutes to look at one of the major steps in internationalizing a program and introduced ResourceBundle, I’d like to stop and look at the overall architecture of the Java internationalization frameworks.



`java.text` architecture

There are three major frameworks in the java.text package: formatting, collation, and boundary detection. The application uses them for certain tasks. The application also uses ResourceBundle in the manner we just examined. But the other three frameworks use ResourceBundle in exactly the same way. Each of these frameworks depends on data stored in resource bundles to tell it how to behave and what to produce as output. Java comes with over 120 different resource bundles (often mistakenly called “locales”) for various combinations of language and country. The application program can specify which resource bundle an international API should use by using a Locale object.

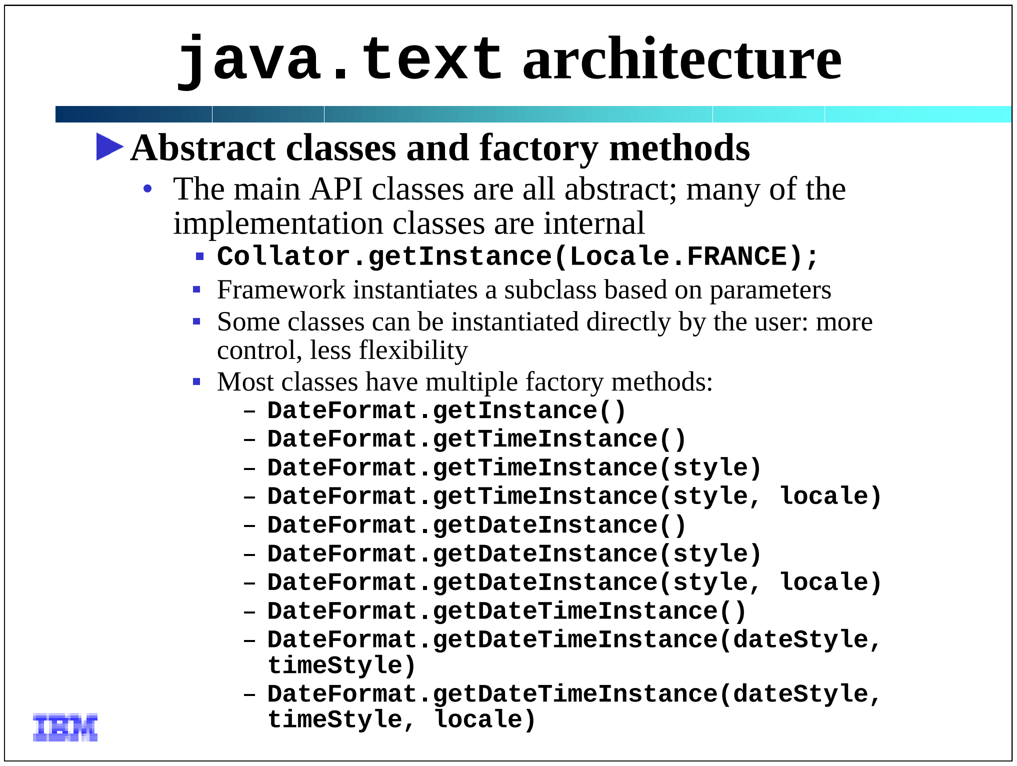


`java.text` architecture

- **Data-driven model**
  - The class is a pure execution engine
  - Its actual behavior is specified by a description (usually a `String`) that is supplied from outside
    - The application supplies it at construction time
    - Or the framework loads one from a resource bundle

IBM

This dependence on resource bundles is one of the central design characteristics of the Java i18n frameworks. That is, they all use a data-driven model. Most of the i18n classes are pure execution engines that derive their exact behavior from some kind of textual description supplied by the caller or fetched from a resource bundle. This allows changes in behavior without touching code. (Some capabilities *do* require different code, but this approach keeps these situations to a minimum.)



`java.text` architecture

- **Abstract classes and factory methods**
  - The main API classes are all abstract; many of the implementation classes are internal
    - **`Collator.getInstance(Locale.FRANCE);`**
    - Framework instantiates a subclass based on parameters
    - Some classes can be instantiated directly by the user: more control, less flexibility
    - Most classes have multiple factory methods:

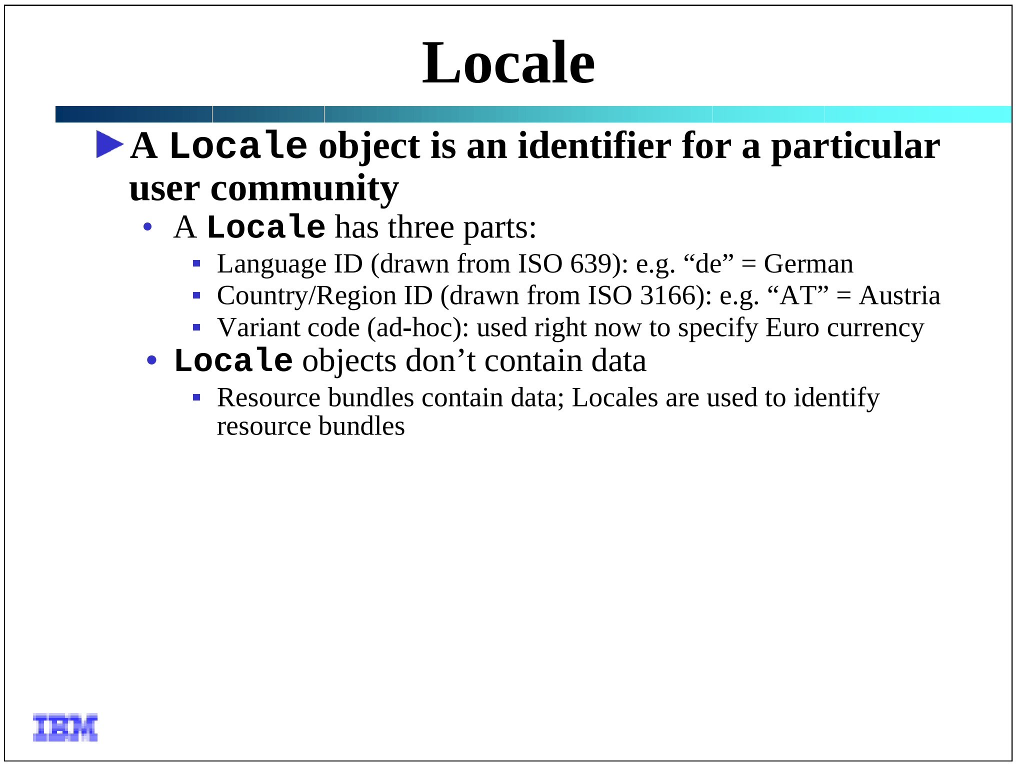
```
– DateFormat.getInstance()
– DateFormat.getTimeInstance()
– DateFormat.getTimeInstance(style)
– DateFormat.getTimeInstance(style, locale)
– DateFormat.getDateInstance()
– DateFormat.getDateInstance(style)
– DateFormat.getDateInstance(style, locale)
– DateFormat.getDateTimeInstance()
– DateFormat.getDateTimeInstance(dateStyle,
  timeStyle)
– DateFormat.getDateTimeInstance(dateStyle,
  timeStyle, locale)
```

IBM

Sometimes, different code is required to support certain locales. To allow for this, the Java i18n frameworks are based on abstract classes and factory methods. That is, the primary API class for a framework is abstract. The implementation class (or classes) can then be made internal to the package. The implementation classes are instantiated by calling a static method on the abstract class instead. This allows us to use different classes in some cases without changing the API.

Of course, many of the implementation classes are also public. The application program can use them when it requires more control over the result, but only at the expense of not being able to use the other implementation classes to handle the special cases.

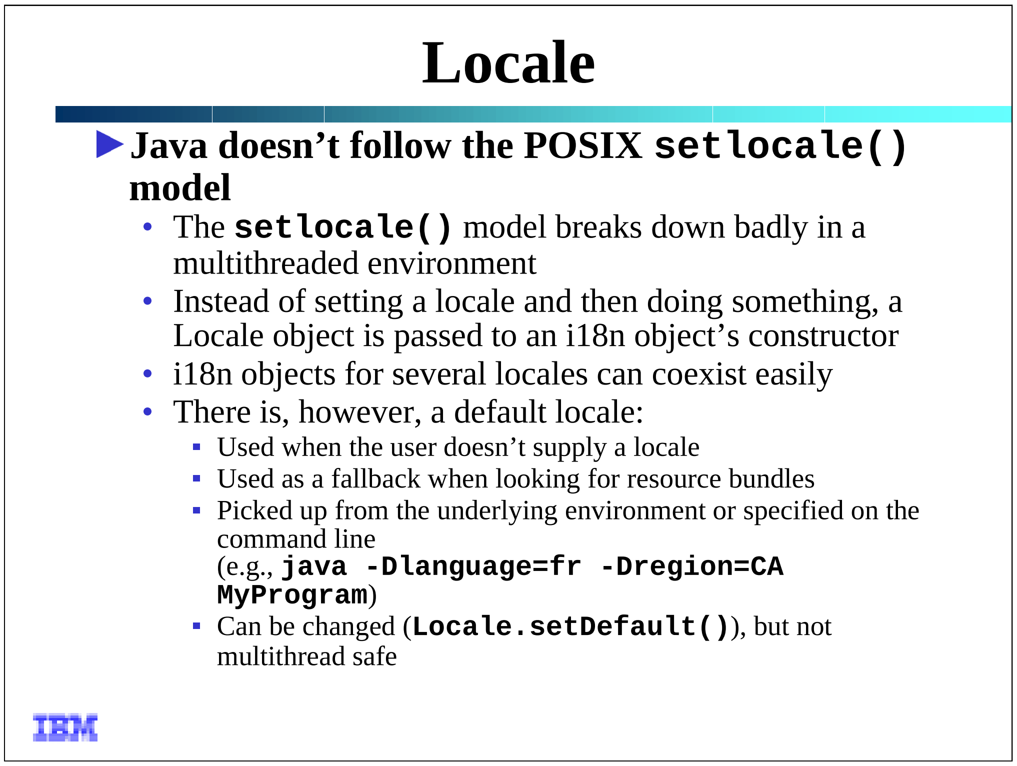
Most classes that supply factory methods supply more than one. This allows the user to achieve a fair amount of control over the result without having to call the implementation class directly.



Locale

- **A `Locale` object is an identifier for a particular user community**
  - A **`Locale`** has three parts:
    - Language ID (drawn from ISO 639): e.g. “de” = German
    - Country/Region ID (drawn from ISO 3166): e.g. “AT” = Austria
    - Variant code (ad-hoc): used right now to specify Euro currency
  - **`Locale`** objects don’t contain data
    - Resource bundles contain data; Locales are used to identify resource bundles

A Locale object is the key that’s used to specify a particular user community. The community is identified by a language code, a region or country code, and an optional variant code. Locale objects don’t contain data; they just identify user communities. The data resides in resource bundles, and the Locales are used to locate appropriate resource bundles. This approach allows different subsystems to support different sets of locales (in particular, it means that an application doesn’t have to support all of the locales the i18n library supports, nor is it limited to just the locales the i18n library supports.



Locale

- **Java doesn't follow the POSIX `setlocale()` model**
  - The **`setlocale()`** model breaks down badly in a multithreaded environment
  - Instead of setting a locale and then doing something, a Locale object is passed to an i18n object's constructor
  - i18n objects for several locales can coexist easily
  - There is, however, a default locale:
    - Used when the user doesn't supply a locale
    - Used as a fallback when looking for resource bundles
    - Picked up from the underlying environment or specified on the command line (e.g., **`java -Dlanguage=fr -Dregion=CA MyProgram`**)
    - Can be changed (**`Locale.setDefault()`**), but not multithread safe

If you've done work in C, you've probably run into the POSIX locale model, where the locale *does* contain data, and where there's only a single active locale in effect for a process at any one time.

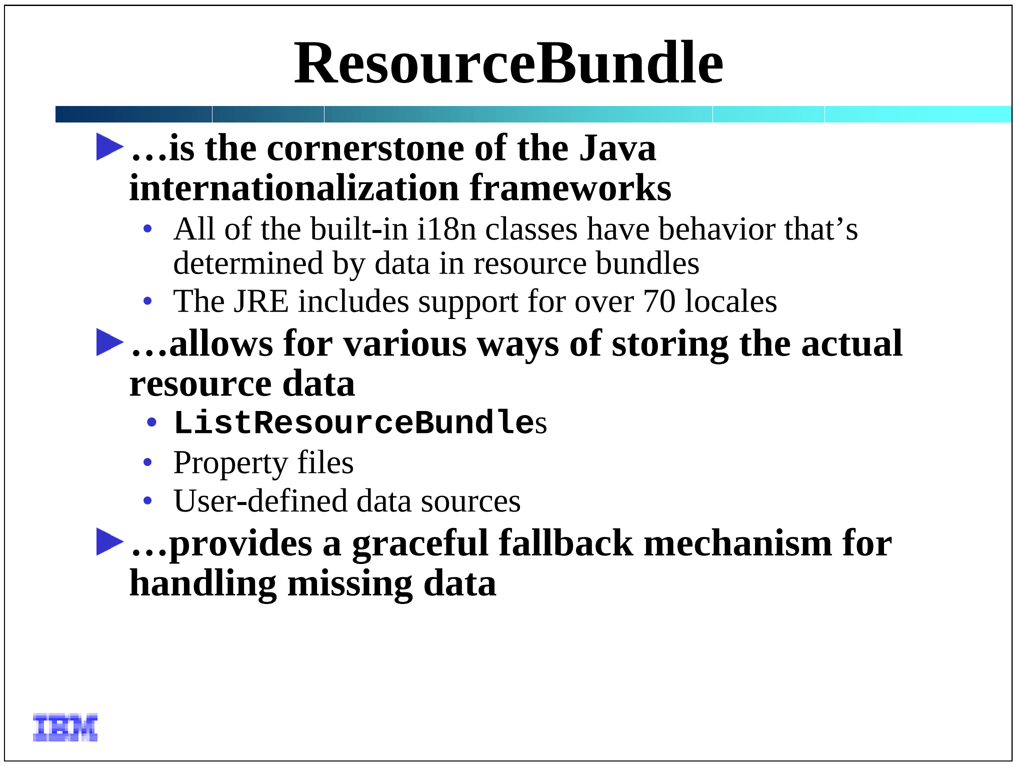
This model breaks down badly in a multithreaded environment, because all i18n operations may have to be wrapped in setlocale() calls, and because multiple threads all share the same locale setting (possibly requiring a locking scheme of some kind).

Instead setting a locale each time you do something, you instantiate one of the i18n objects with a locale, and then use that object every time you want that locale's behavior. There is no global setting.

There *is* a default locale, and it can be set with locale.setDefault(), but you shouldn't use this function the same way you'd use setlocale() in C. The default locale is what locale gets used anywhere you don't specify a locale. It's either picked up from the OS or supplied by the user on the command line. You should pretty much never change the default locale. When you feel the temptation to call Locale.setDefault() a lot, switch to specifying the locale explicitly everywhere you're asked for it and keep track of the locale(s) yourself.

Another reason not to use Locale.setDefault() is that it also isn't multithread safe. This means you're not allowed to use it in an applet at all.

Most applications just want to "work right" for the user, and thus never need to set the locale explicitly or think about the default locale.



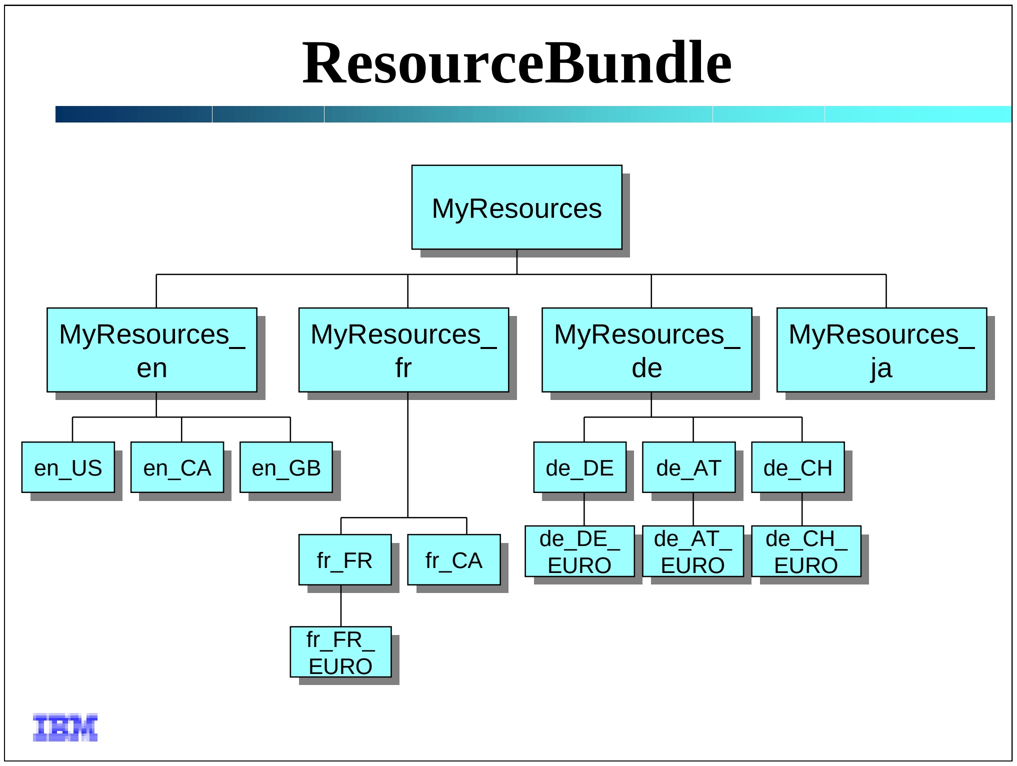
ResourceBundle

- **…is the cornerstone of the Java internationalization frameworks**
  - All of the built-in i18n classes have behavior that's determined by data in resource bundles
  - The JRE includes support for over 70 locales
- **…allows for various ways of storing the actual resource data**
  - `ListResourceBundle`s
  - Property files
  - User-defined data sources
- **…provides a graceful fallback mechanism for handling missing data**

IBM

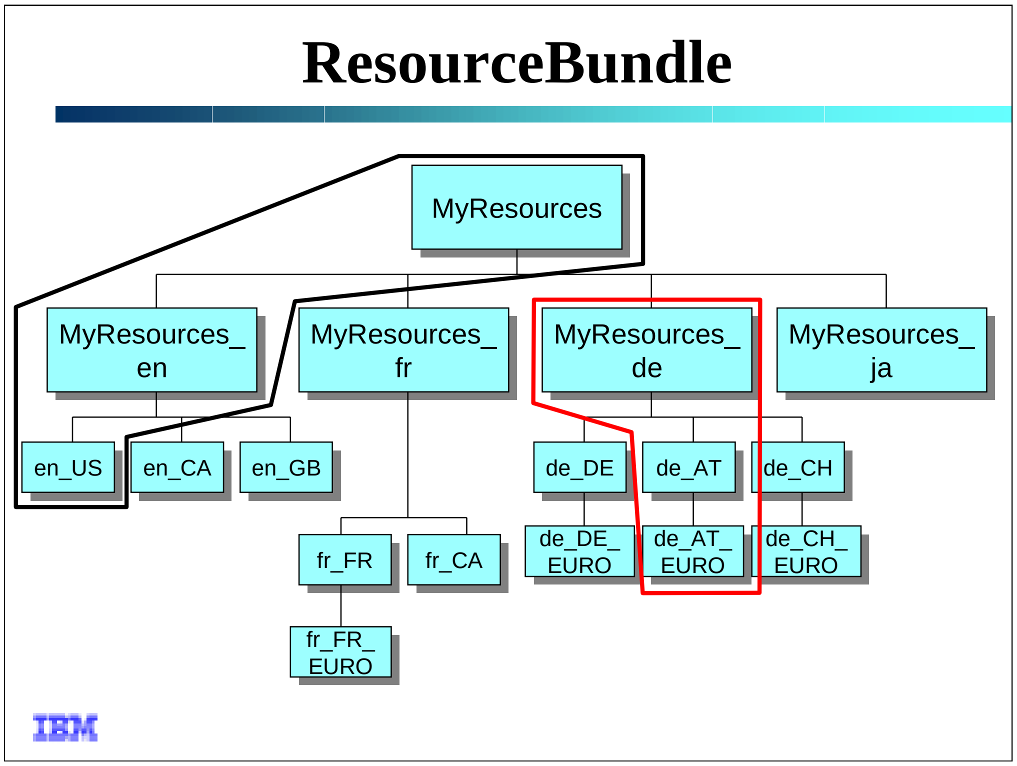
Most of the information on this slide we've already talked about: the whole i8n library is resource-bundle-driven, and ResourceBundle provides a generic interface to any type of actual repository of resource data.

The other main thing ResourceBundle gives you is a graceful fallback in case information for a particular locale isn't there.



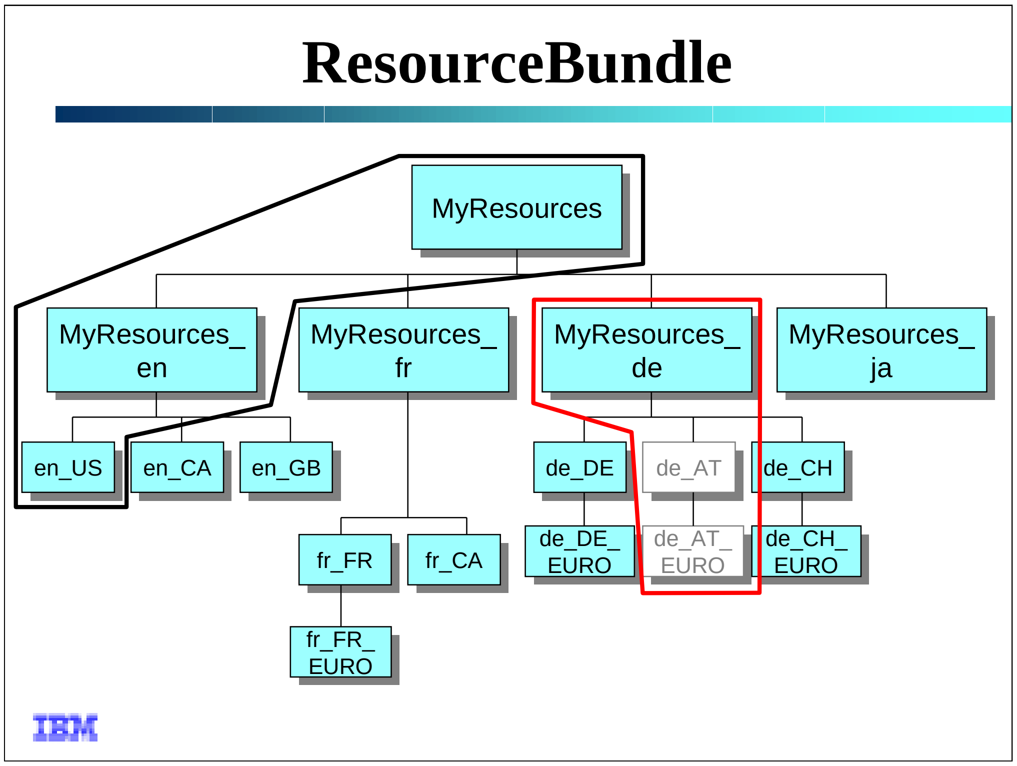
ResourceBundle

This is a resource bundle hierarchy. You have a family of resource bundles called "MyResources". At the top, with no locale name appended, is the *root resource bundle*. The next level down includes all the resource bundles qualified only with a language code. The tier below that is all the resource bundles tagged with both language and country codes, and the bottom tier is all the resource bundles with language, country, and variant codes.



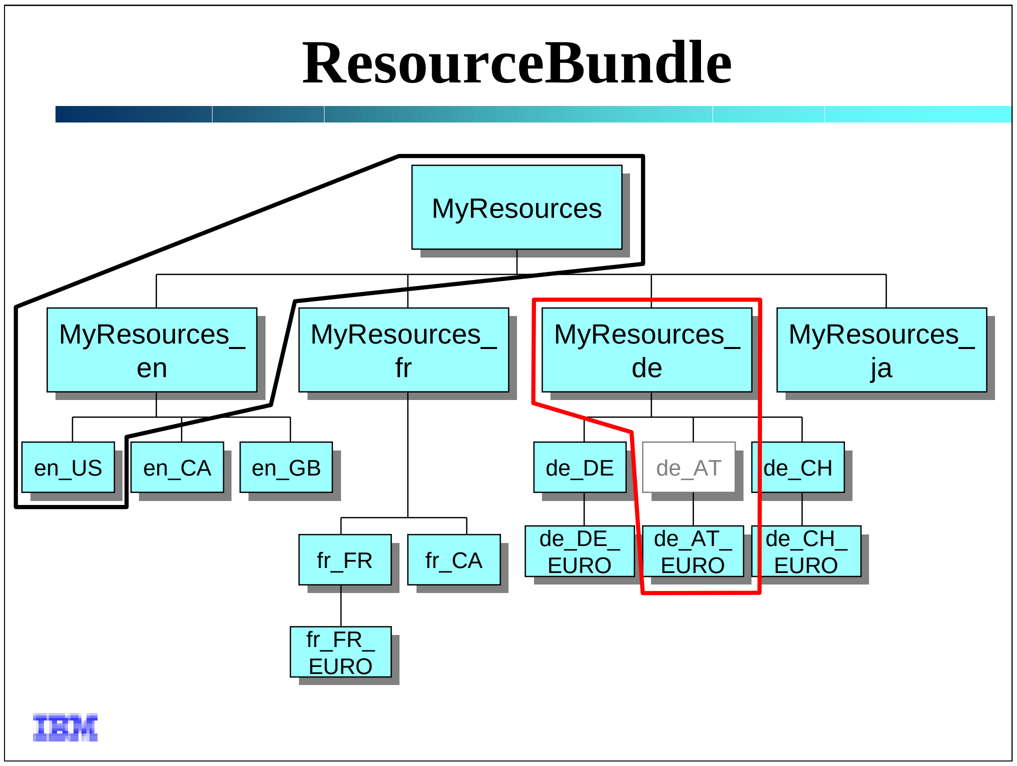
ResourceBundle

This hierarchy is used to define a search path. This diagram shows how the resource bundle engine would search the hierarchy for the resource bundle for a particular locale. The red line shows the search path for the requested locale (in this case, “de_AT_EURO”). It starts at the bottom of the hierarchy and works its way up, progressing to more and more general bundles, until it finds a bundle. If it can’t find one, then it tries the chain leading upward from the default locale (“en_US” here). The root resource bundle is the bundle of last resort. Since the bundle we were looking for (MyResources_de_AT_EURO) is actually here, the search just stops there.



ResourceBundle

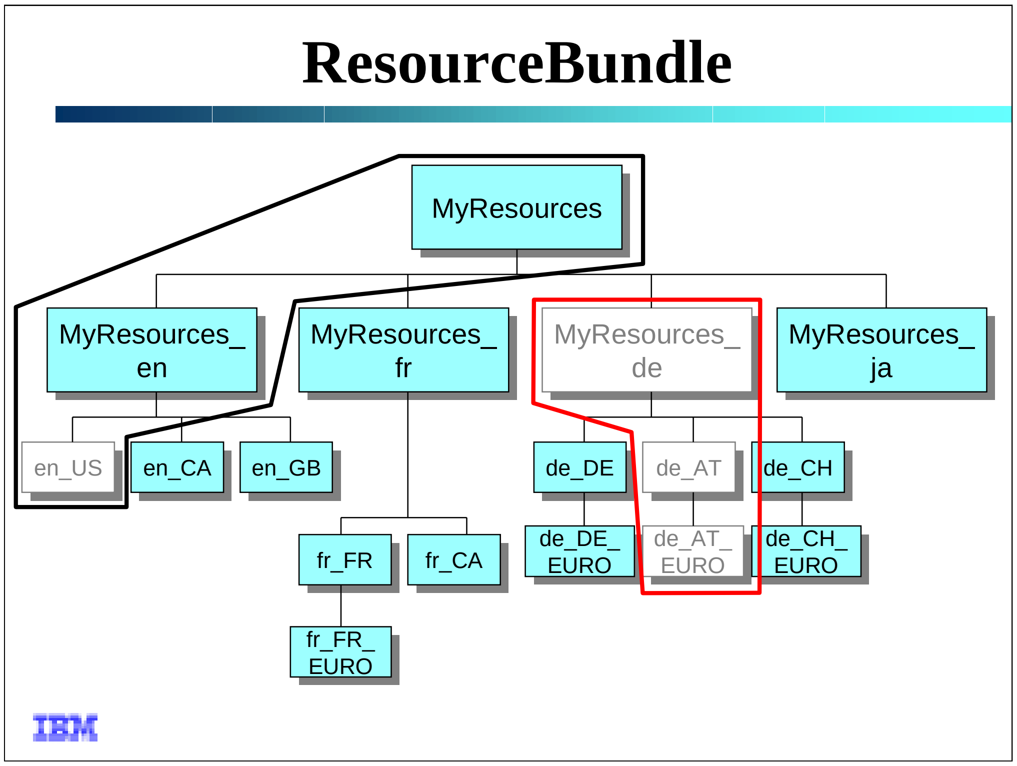
But here, MyResources_de_AT_EURO and MyResources _de_AT are both missing. Since we don’t have information specifically for Austrian German using the Euro currency symbol, we fall back on generic German-language information.



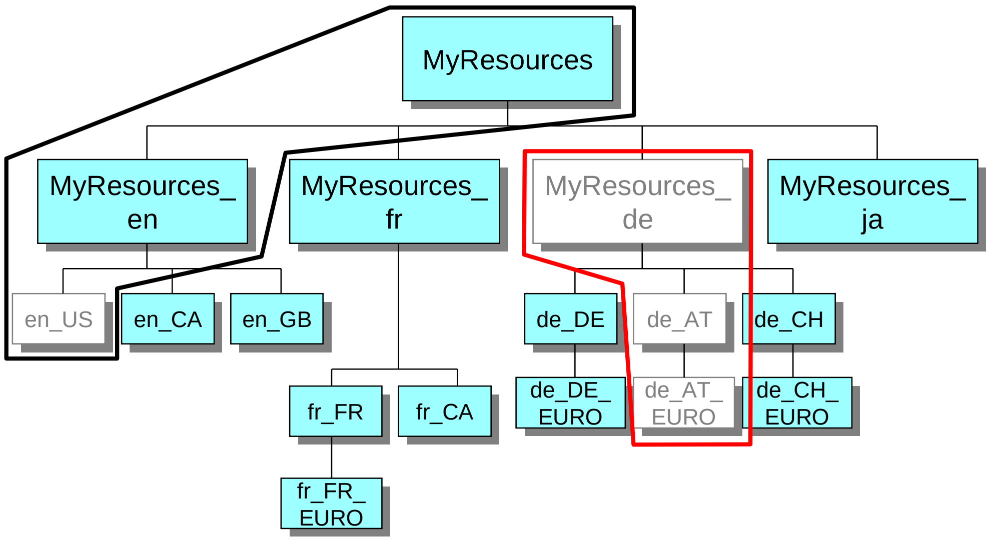
ResourceBundle

Here, we’re missing de_AT, but not de_AT_EURO. We can go straight to the requested bundle, but this is still a malformed hierarchy. This is because if you ask for de_AT, you’ll just get de instead of de_AT_EURO. For any bundle you have in your hierarchy, you *must* have all the bundles that should appear above it in the hierarchy. Things won’t work right otherwise.

In other words, any time you have a resource bundle that’s specific to a particular language and country (for example), you must also supply one that has generic information for just the language (this prevents the fallback mechanism from unnecessarily falling back on a different language). You must always supply a root resource bundle. (In fact, if you only support one locale, you may have *only* a root resource bnudle.)

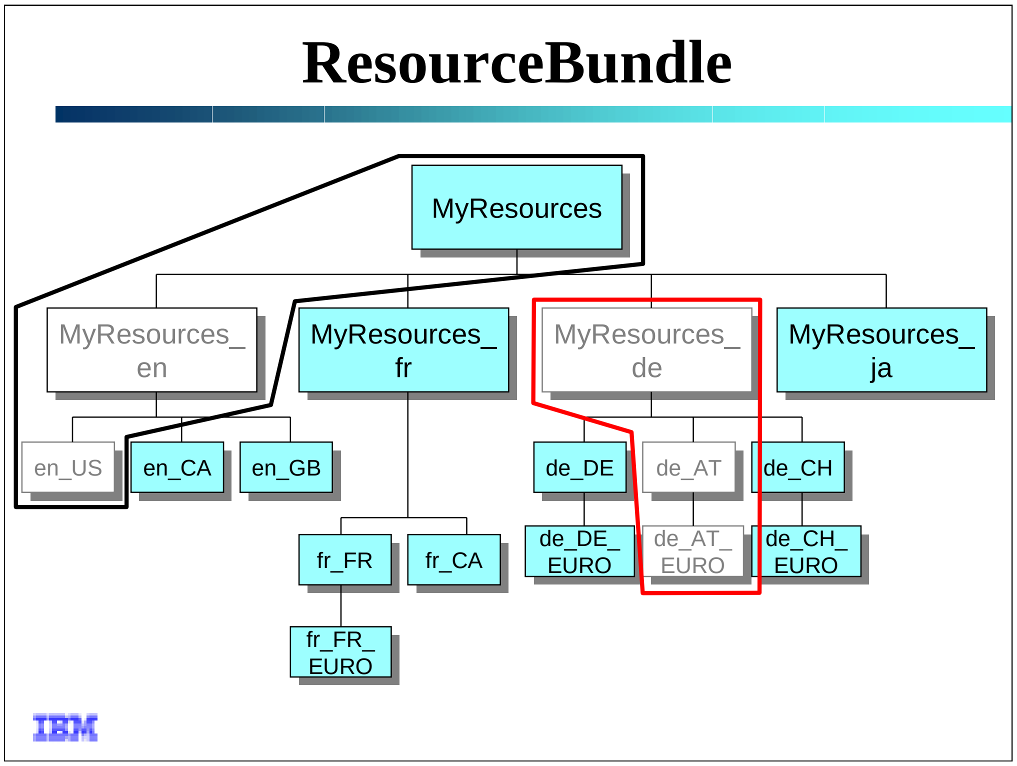


ResourceBundle



IBM

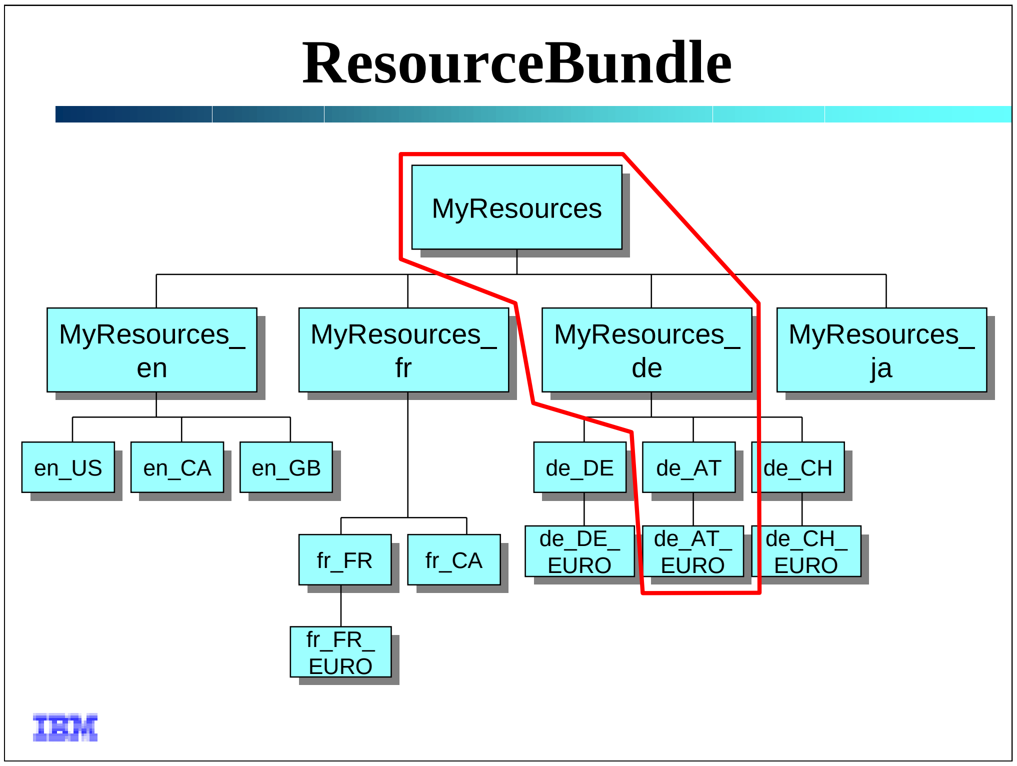
In this example, we don’t have any German-language data at all, and we also don’t have a specific bundle for U.S. English (the default locale). Here, if we look for Austrian German, we’ll end up falling back to generic English data.



ResourceBundle

And if we have neither English nor German data, we fall back on the root resource bundle. The root resource bundle can contain data in any language whatsoever (or data that's language-independent). The choice is up o the programmer. The only thing to keep in mind is that this is the resource bundle of last resort, and you want to make sure it contains reasonable last-resort data.

If you support more than one language, you should generally have a resource bundle explicitly tagged for that language as well as the root bundle. This will keep you from falling back on the default locale when the language you want is actually stored in the root.

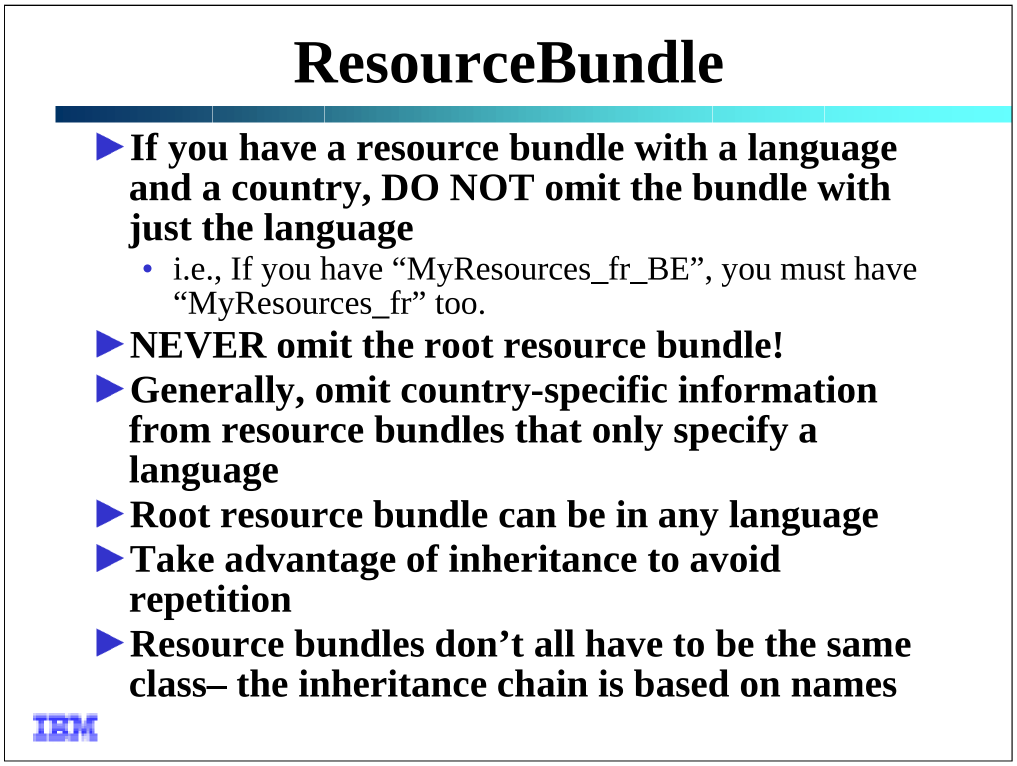


ResourceBundle

Resource bundles can inherit from one another. Once a bundle has been located, this is the hierarchy that's followed when looking for resources in that bundle. The search starts in the specified bundle (or the closest one to it that was found) and proceeds from there up the hierarchy until it reaches the root (it *doesn't* fall back on the default locale first; instead, it throws an error).

This means bundles on the second tier of the h8ierarchy really only need to specify values for resources that differ from the value for that resource in the root resource bundle. Likewise, bundles further down the chain only have to specify values that deviate from the values in their parents.

HOWEVER, all of the resource *must* be present in the root resource bundle. You can't add new resources as you move down the chain.



ResourceBundle

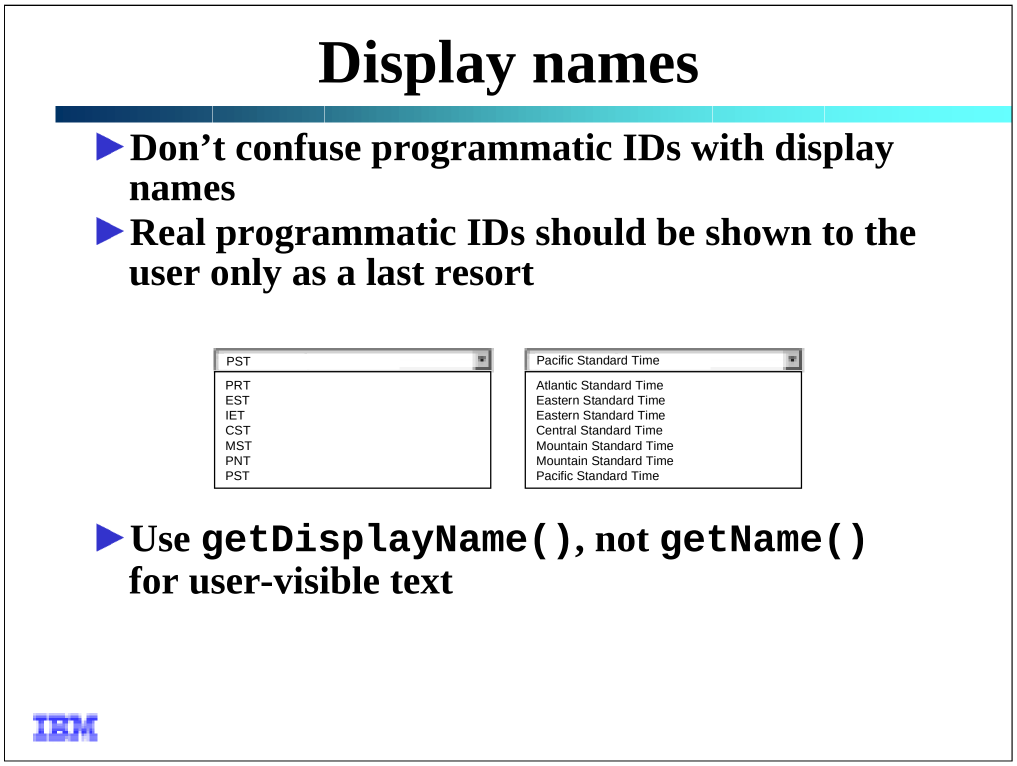
- If you have a resource bundle with a language and a country, DO NOT omit the bundle with just the language
  - i.e., If you have “MyResources_fr_BE”, you must have “MyResources_fr” too.
- NEVER omit the root resource bundle!
- Generally, omit country-specific information from resource bundles that only specify a language
- Root resource bundle can be in any language
- Take advantage of inheritance to avoid repetition
- Resource bundles don’t all have to be the same class– the inheritance chain is based on names

IBM

This is just a recap of the things we’ve already talked about.

One additional point to highlight is that the resource-bundle and resource lookup mechanisms both operate using only name lookup. There is no requirement that all of the bundles in the hierarchy be of the same class, nor that resource bundles inheriting data from other resource bundles have to be subclasses of them.

Generally, just make all of your resource bundles descend directly from ListResourceBundle and not from each other.



Display names

- Don’t confuse programmatic IDs with display names
- Real programmatic IDs should be shown to the user only as a last resort

```
PST
PRT
EST
IET
CST
MST
PNT
PST
```

```
Pacific Standard Time
Atlantic Standard Time
Eastern Standard Time
Eastern Standard Time
Central Standard Time
Mountain Standard Time
Mountain Standard Time
Pacific Standard Time
```

- Use `getDisplayName()`, not `getName()` for user-visible text

IBM

Another important design feature is the distinction between programmatic IDs and display names. For time zones in particular, a programmatic ID can be mistaken for a display name. You don’t want to present programmatic IDs to the user, except as a last resort, even if the display name and the internal ID are the same.

This is because they’re not always the same. In the example above, IET isn’t a real time zone abbreviation; it’s just an ID for the version of Eastern Standard Time used in Indiana, where they don’t observe daylight savings time.

Display names can also be translated, so they’re looked up in resource bundles. Just as with resource names, locale IDs and time zone IDs (and so forth) are meant only for internal programmatic use. Don’t use getName() to get user-visible text; use getDisplayName() instead.

Formatting Text Messages

Okay, now back to issues you encounter while internationalizing.

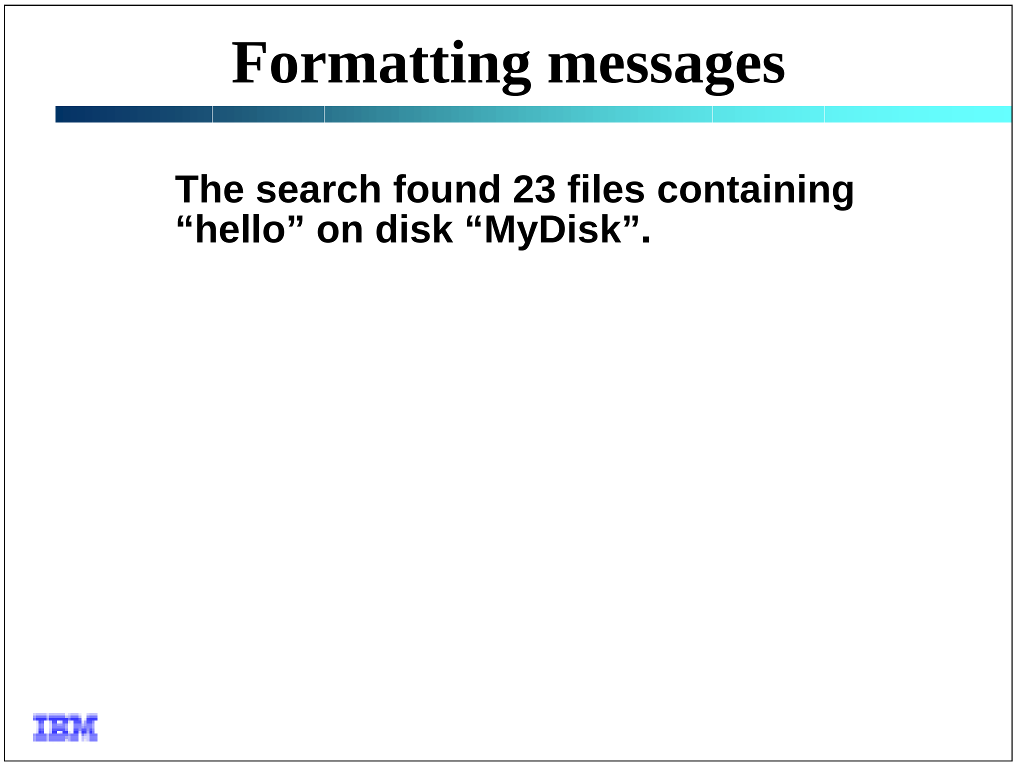
Formatting messages

```
dialog.add("Center",
  new Label("The search found " + hits
  + " files containing \"" + searchString
  + "\" on disk \"" + searchRoot + "\"."));
```

IBM

Putting user-visible text (and other UI elements) into resources is the single largest thing you can do to make your program easier to translate. But it's far from the only thing that must be done. Look at this line here from our ResorceBundle code snippet. This is an example of a hidden assumption. Let's take a look at where the assumption is.

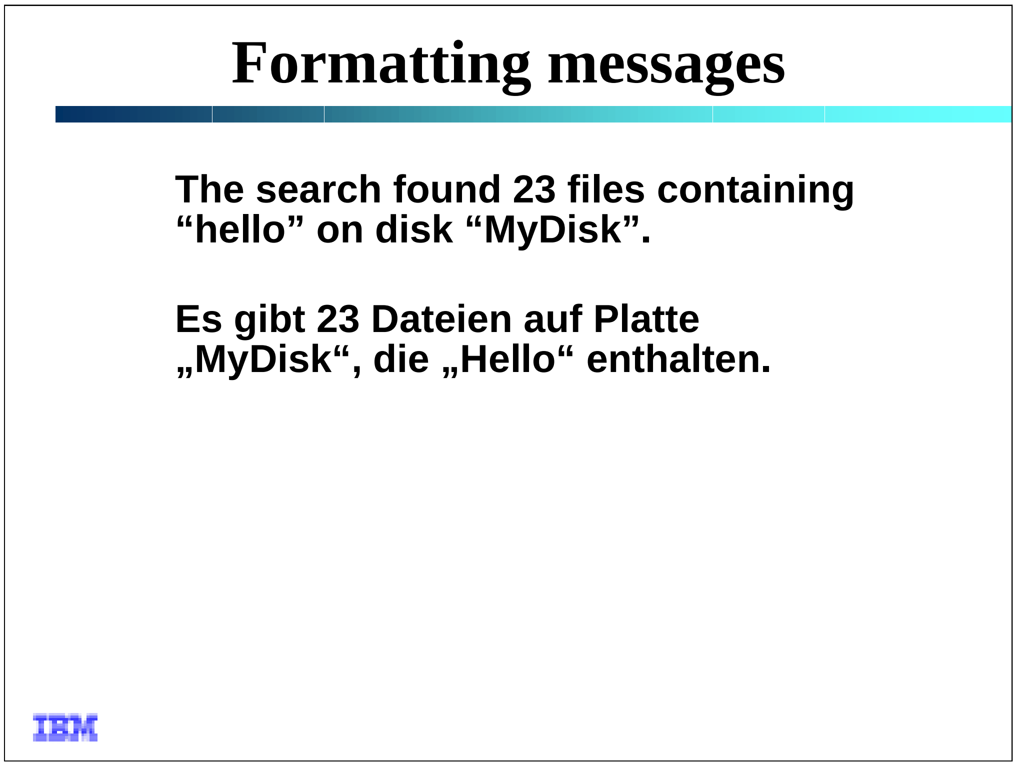
In the previous exercise, we had to take each fixed fragment of this message and translate it individually. But that's not the way the user would be thinking of this message-- he'd be thinking of it as a single sentence with "blanks" that get filled in.



Formatting messages

The search found 23 files containing “hello” on disk “MyDisk”.

In other words, the user will see this: a complete sentence. There are a few dynamic parts of this sentence, but the “fill in the blank” quality doesn’t change the fact that this message is a single unit. Why does this matter?



Formatting messages

The search found 23 files containing “hello” on disk “MyDisk”.

Es gibt 23 Dateien auf Platte „MyDisk“, die „Hello“ enthalten.

Well, consider what would happen to this sentence if we translated it into German. The sentence structure is totally different. The different parts of the sentence go in different places relative to the “blanks,” which means a translator would have to consider the whole sentence together when translating, not just translate the individual fragments. In this case, that’d work, but if you left out a static text string between two “blanks” and in some other language there needed to be a word there, the translator would be stuck. This is one of the hidden assumptions in the example.

Formatting messages

The search found 23 files containing “hello” on disk “MyDisk”.

Es gibt 23 Dateien auf Platte „MyDisk“, die „Hello“ enthalten.

The more serious hidden assumption in the code is that the “blanks” will come in the same order in every language. That isn’t true here. The dynamic parts of the sentence go in a very different order once the sentence is translated into German.

Code that builds up messages needs to take this into account. Therefore, it’s a Bad Idea to build up user-visible messages using string concatenation.

Formatting messages

```
dialog.add("Center",
    new Label("The search found " + hits
    + " files containing \"" + searchString
    + "\" on disk \"" + searchRoot
    + "\"."));
```

So how do you do it? Well, let’s take another look at our code snippet. This is how it looked originally. How do we fix it to output the message in a way that doesn’t make any assumptions about sentence structure? Java provides a class called MessageFormat for this.

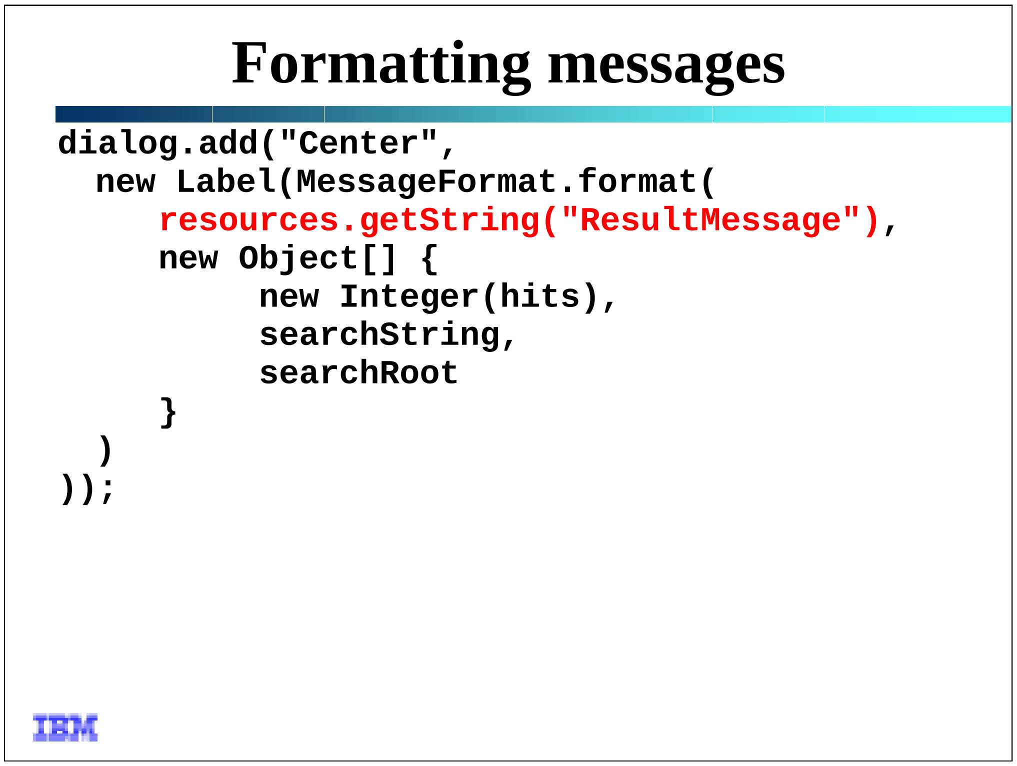
Formatting messages

```
dialog.add("Center",
  new Label(MessageFormat.format(
     "The search found {0} files containing "
     + "\"{1}\" on disk \"{2}\".",
     new Object[] {
          new Integer(hits),
          searchString,
          searchRoot
     }
  )
));
```

With MessageFormat, the code changes to look like this. The static format() method on MessageFormat takes two arguments: a pattern string and an array of arguments. The argument array contains the values that get filled into the "blanks" in the message, in a program-specified order.

The pattern string includes tokens indicating where the "blanks" are: these are the numerals in braces. The numeral tells the formatter which value from the argument array to put in at a particular "blank" position. In every language, "{0}" will always refer to the number of hits, "{1}" will always refer to the search pattern, and "{2}" will always refer to the name of the search root. The program will always supply these arguments in this order. But the pattern string doesn't have to use them in this order. It can rearrange them at will, leave some out, use some twice, and so on.

In other words, the localizable part of this statement is the pattern string.

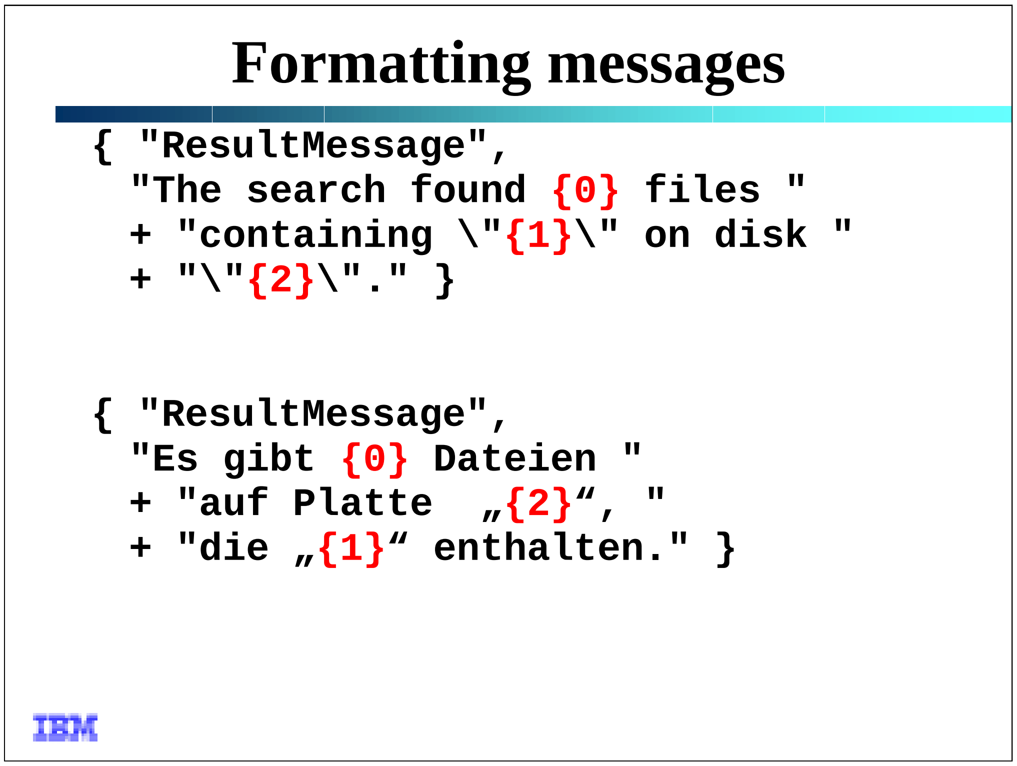


Formatting messages

```
dialog.add("Center",
  new Label(MessageFormat.format(
     resources.getString("ResultMessage"),
     new Object[] {
          new Integer(hits),
          searchString,
          searchRoot
     }
  )
));
```

IBM

So to make this line language-independent, all you have to do is pull the pattern string out of a resource. So the statement ends up looking like this.



Formatting messages

```
{ "ResultMessage",
  "The search found {0} files "
  + "containing \"{1}\" on disk "
  + "\"{2}\"." }


{ "ResultMessage",
  "Es gibt {0} Dateien "
  + "auf Platte  „{2}“, "
  + "die „{1}“ enthalten." }
```

IBM

In the resource-bundle definition, we can now replace the four resources containing fragments of the message with a single resource containing the pattern string. The first line shows the English version of the pattern string, and the second line shows the German version. Note how the German version uses the arguments in a different order than the English version did.

Handling plurals

The search found 1 files containing “hello” on disk “MyDisk”.

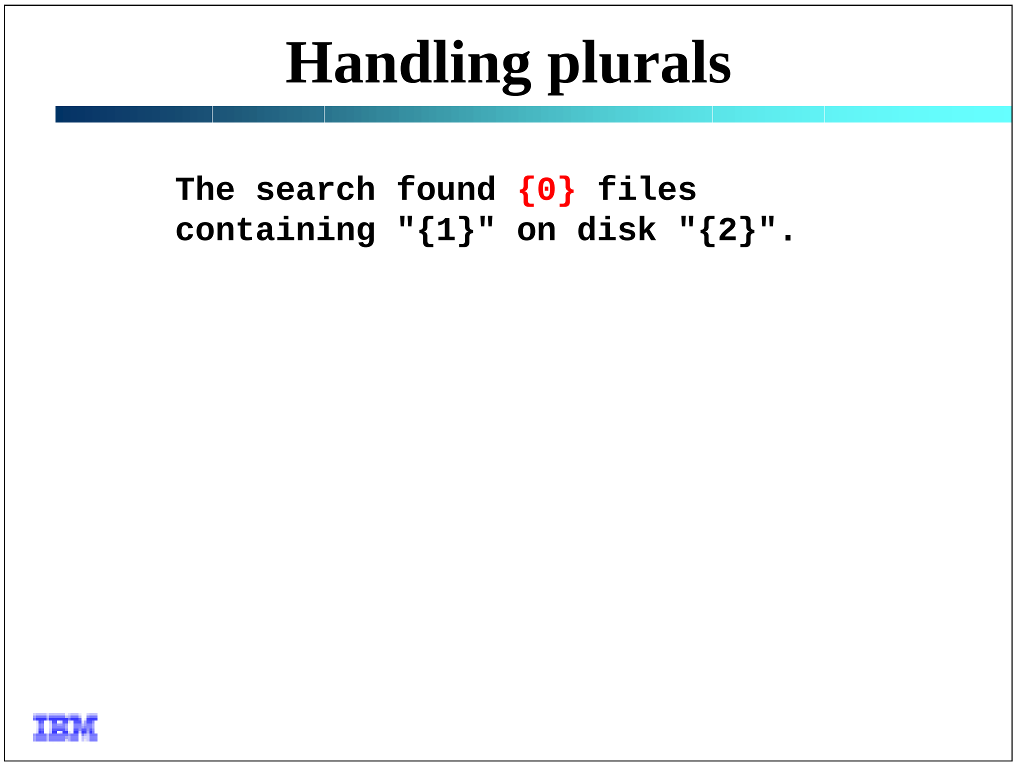
We’re still left with one pesky problem every programmer has encountered many times: Here’s what you get when the number of hits is 1. There are a number of ways programmers deal with this. One is to just leave it this way and forget about it. This produces wrong output, of course, but most users will either ignore it or kind of sneer and keep going. It doesn’t impair understanding. This isn’t necessarily true in other languages.

Handling plurals

The search found 1 file(s) containing "hello" on disk "MyDisk".

Then there's the classic dodge for the problem. I've always thought this looks pretty stupid too, and this definitely won't work in a lot of languages.

The third approach is to just break down and include an "if" statement to select between the singular and plural forms of "file". But this includes a hidden assumption: that your only choices are singular and plural. In some languages, you have singular, dual, and plural, for example. A fixed "if" will leave users of these languages out of luck.

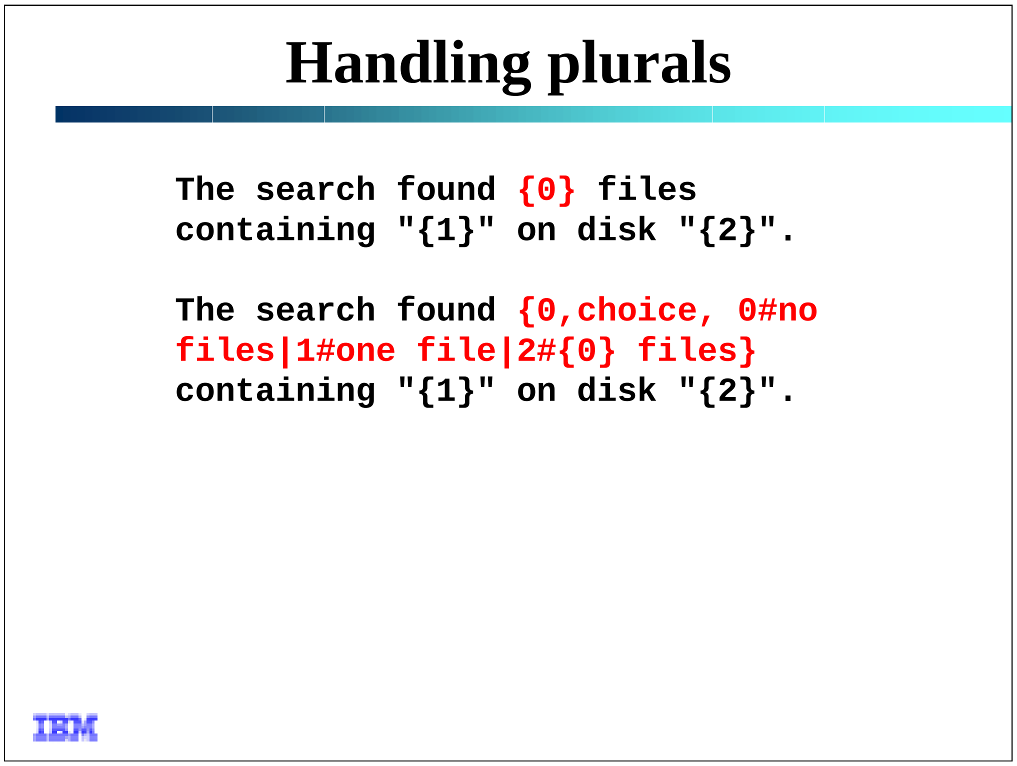


Handling plurals

```
The search found {0} files
containing "{1}" on disk "{2}".
```

MessageFormat is a lot more flexible than it looks at first sight. Each of these substitutions (the numbers in the braces) can contain more than just a number in the brace. They can also extra arguments that tell the formatter what kind of argument it is, and to supply more information on how to format that argument.

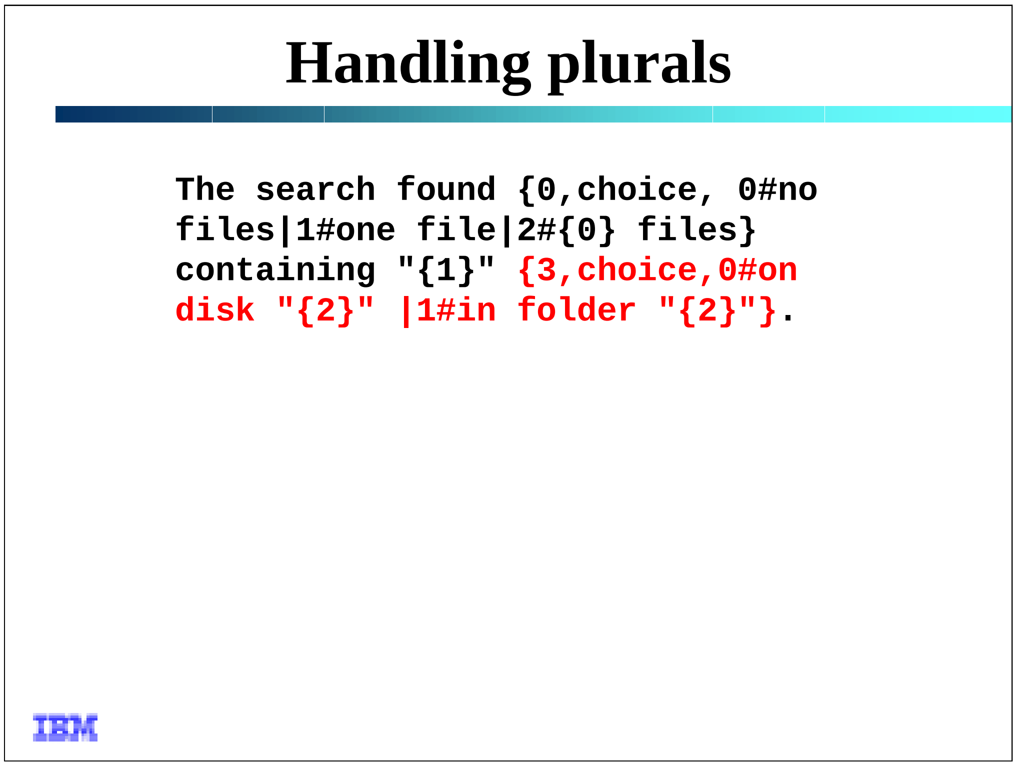
One of our options is to tell the formatter to format a numeric argument as a choice rather than a number. Formatting a number as a choice uses the number to select among several different pattern strings.



Handling plurals

```
The search found {0} files
containing "{1}" on disk "{2}".

The search found {0,choice, 0#no
files|1#one file|2#{0} files}
containing "{1}" on disk "{2}".
```

So here we can deal with the plural problem by having argument 0 be a choice argument. We supply three different pattern strings, separated by vertical bars. The numbers at the beginning of the choice specify the range of values that correspond to that choice. So for 0 or more, the expression evaluates to “no files”. For 1 or more, the expression evaluates to “one file”, and for 2 or more we get “{0} files”. Note that we can re-use the {0} inside the choices, letting us still format the value as a number when the value is 2 or more. Again, this lets us put all the information on the alternative forms of this message into a single pattern string that can be localized all at once.

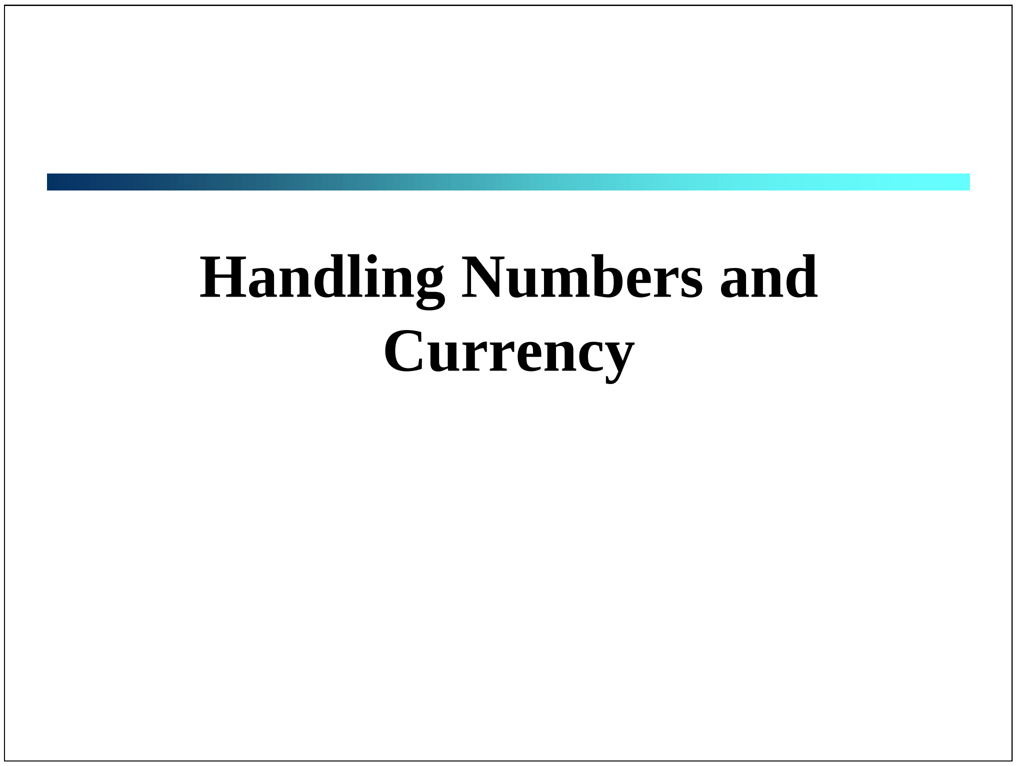


Handling plurals

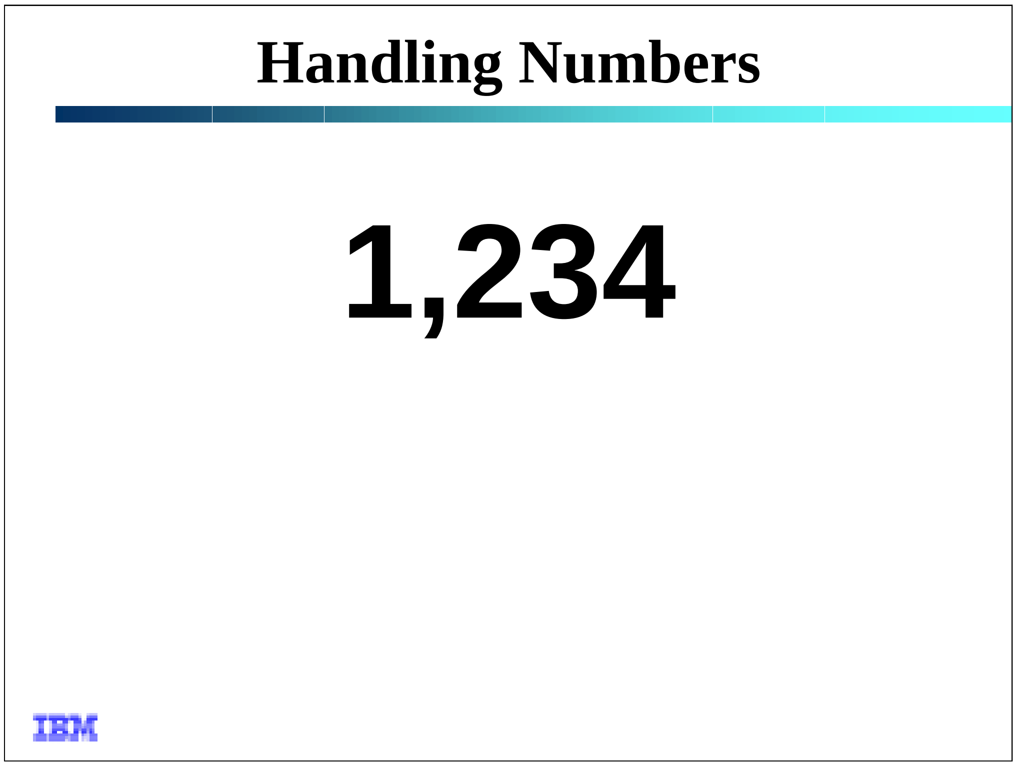
```
The search found {0,choice, 0#no
files|1#one file|2#{0} files}
containing "{1}" {3,choice,0#on
disk "{2}" |1#in folder "{2}"}.
```

Choice formats are useful for some other things, too. Let’s say the root of the search could either be a whole disk or a single folder. Then we’d like to be able to change the message to say either “disk” or “folder” instead of just “disk” all the time. We can use a choice to select between the two (or more) different words.

However, you can’t format a string as a choice– there’s no obvious way for the formatter to look at a string and tell which choice it goes with. We’d have to add another argument to the formatter (argument 3) that’s a selector code. This **does** involve changes to the code, but those changes can apply across all locales.



Handling Numbers and Currency



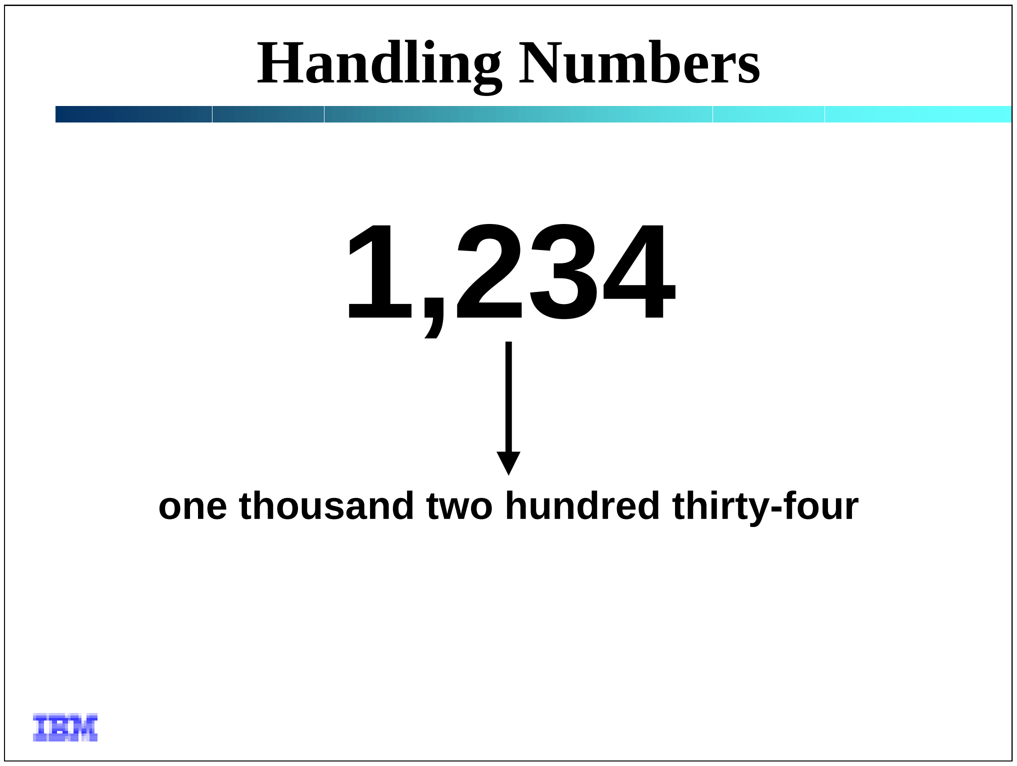
Handling Numbers

1,234

IBM

Now the whole reason we need something like MessageFormat is so that we can intersperse static text with dynamically-generated text. We've now fixed it so that the static text is off in a separate place where it can be translated, but what about the dynamically-generated text?

Take numbers, for example. What number is this?



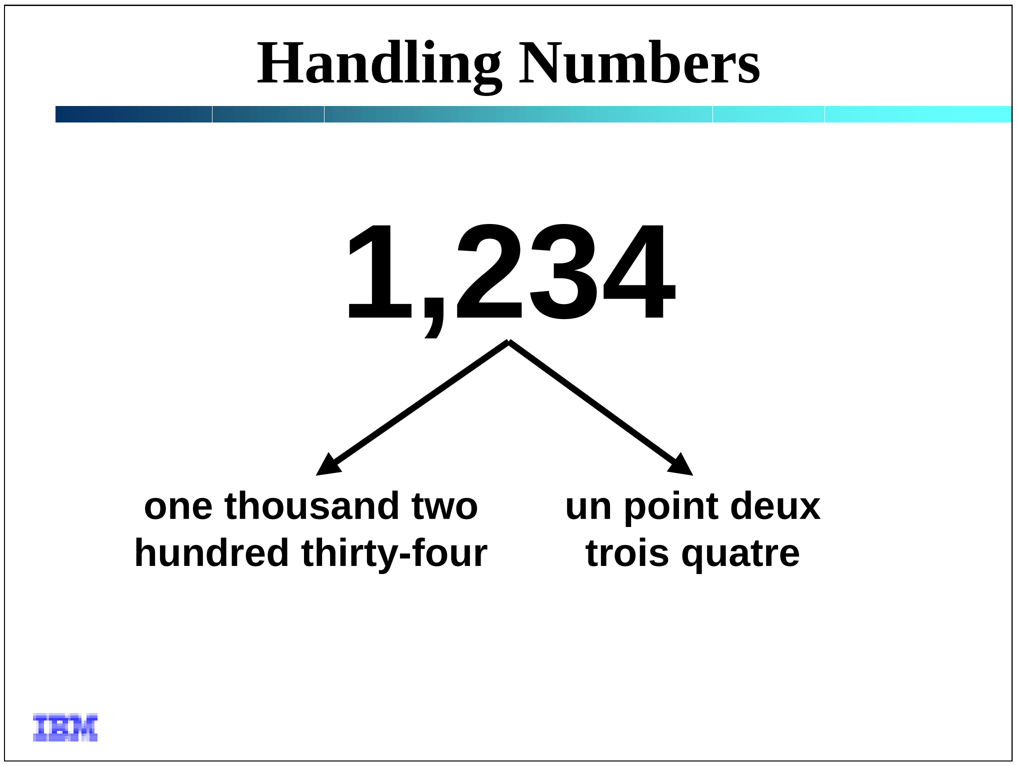
Handling Numbers

1,234

$\downarrow$

one thousand two hundred thirty-four

Well, if you’re an American, you probably looked at this number and saw one thousand two hundred thirty-four. But that’s not what everybody would see.



Handling Numbers

1,234

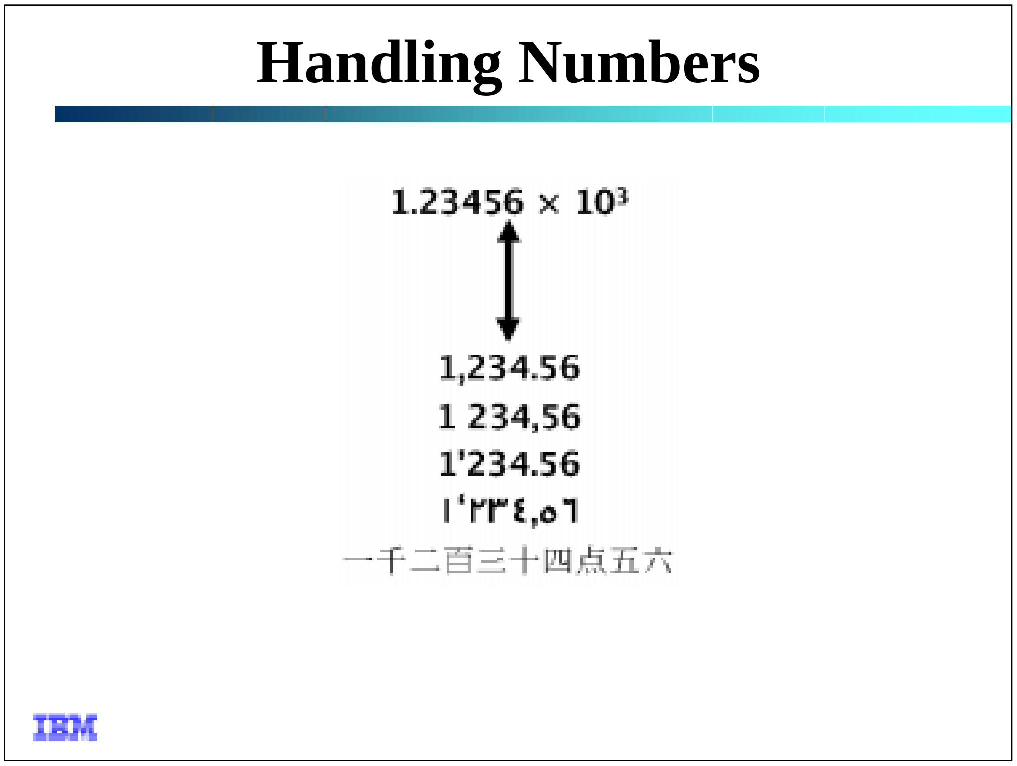
one thousand two hundred thirty-four

un point deux trois quatre

IBM

If you’re French, you’ll see this as one point two three four. That’s because in France, the decimal point is a comma instead of a period. In fact, the comma is used in many European countries, including Great Britiain.

Obviously, there’s a thousandfold difference between the American and European interpretations of this sequence of characters. This could obviously lead to some serious misunderstandings if you’re operating across country boundaries. Clearly, your program needs to worry about this kind of thing if it displays numbers.



Handling Numbers

1.23456×10^3

1,234.56

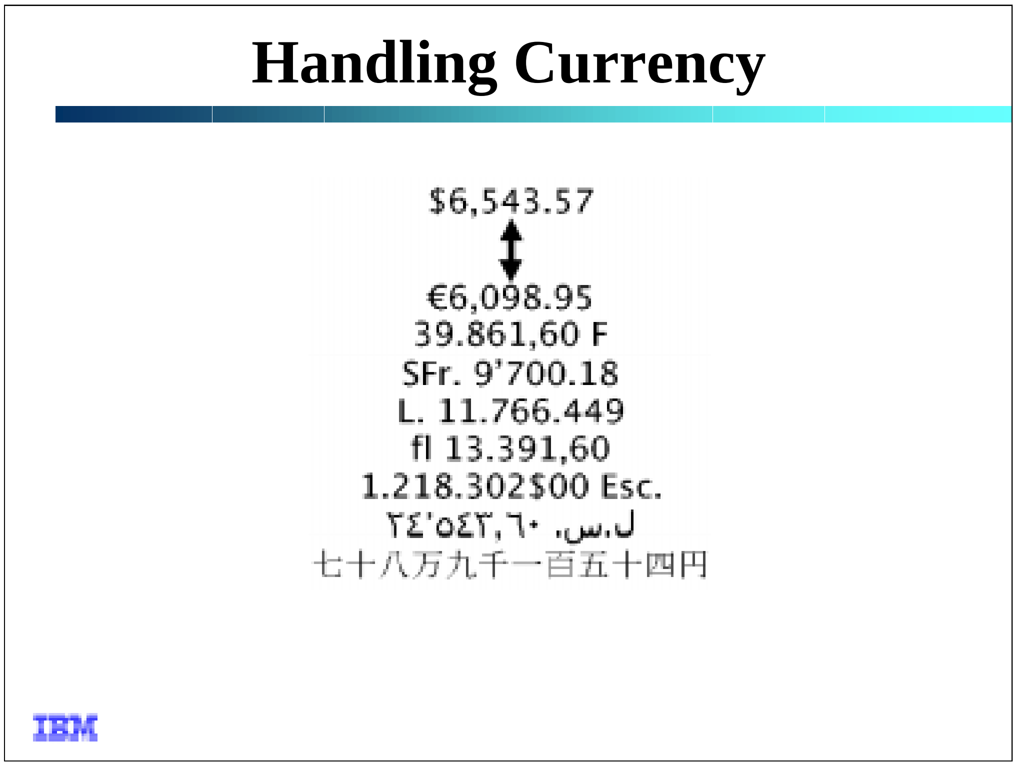
1 234,56

1'234.56

١'٢٣٤,٥٦

一千二百三十四点五六

The decimal-point character isn’t the only character that can vary. This slide shows five different ways the same numeric value can be rendered. The first is the American format. The second is French and the third is Swiss German. So here we have three different combinations of decimal-point and thousands-separator characters. In Arabic, the characters for all the digits have changed as well, and in Japanese, the whole way a number is written is different.



Handling Currency

$6,543.57

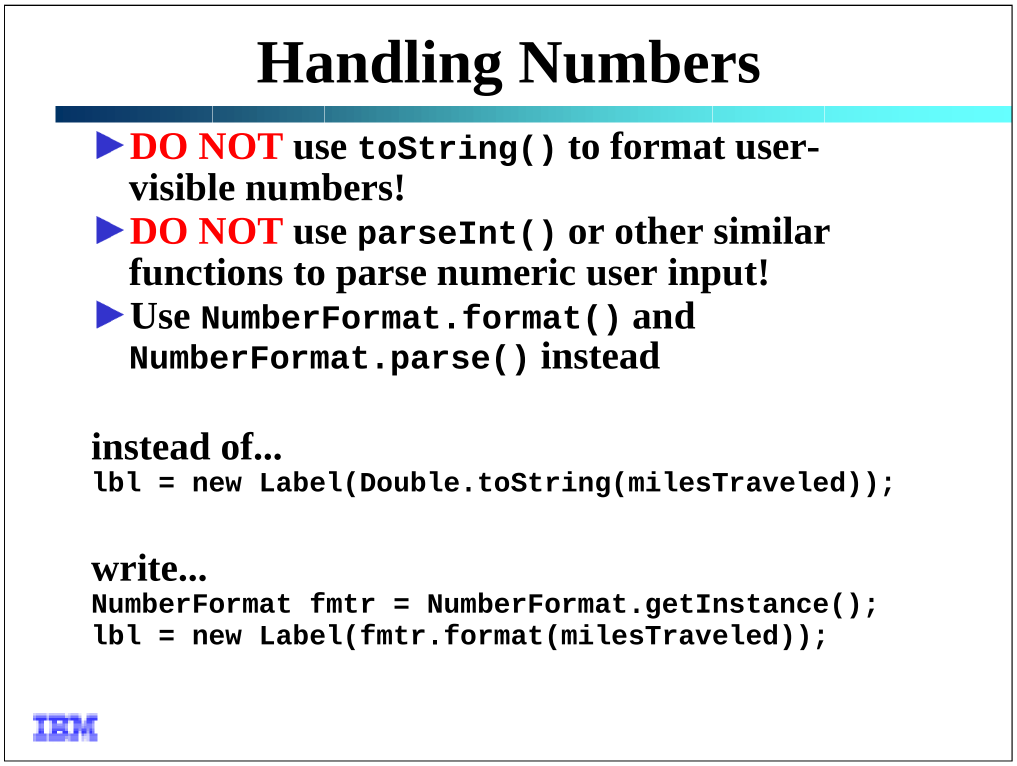
↕

€6,098.95
39.861,60 F
SFr. 9'700.18
L. 11.766.449
fl 13.391,60
1.218.302$00 Esc.
٢٤'٥٤٢,٦٠ ل.س.
七十八万九千一百五十四円

IBM

Handling currency can be even more difficult than handling other kinds of numbers. Now you have to worry about currency symbols and where they get placed relative to the number, alternate decimal-point characters, how many decimal places to show, and how much to round the value.

As if all that weren't complicated enough, you may also have to worry about the exchange rates between different currencies. This is particularly true when an application has to display monetary values in more than one currency.



Handling Numbers

- **DO NOT use `toString()` to format user-visible numbers!**
- **DO NOT use `parseInt()` or other similar functions to parse numeric user input!**
- **Use `NumberFormat.format()` and `NumberFormat.parse()` instead**

instead of...

```
lbl = new Label(Double.toString(milesTraveled));
```

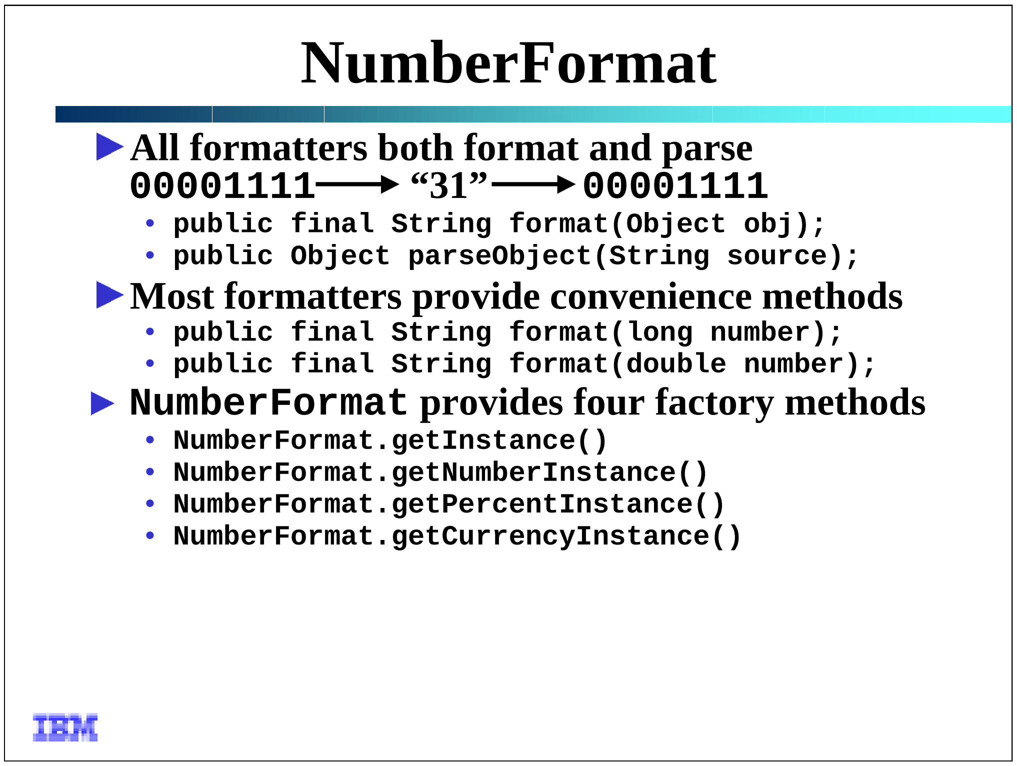
write...

```
NumberFormat fmtr = NumberFormat.getInstance();
lbl = new Label(fmtr.format(milesTraveled));
```

Java’s built-in number formatting engine will handle this for you, but only if you call the right APIs. The main thing to remember is not to use toString() or similar methods to convert numbers into strings that the user will see, and not to use parseInt() or similar methods to parse user input. (toString() and what-not are useful, but only internally and for printing debugging messages.)

Instead of using these APIs, use the NumberFormat object. This will automatically format numbers in a way that’s appropriate for the user’s locale.

The example shows that you have to go through the extra hassle of creating a number formatter to use, but in real life, you’d probably just create a single number formatter and let it sit around in a static variable where everyone can get to it.



NumberFormat

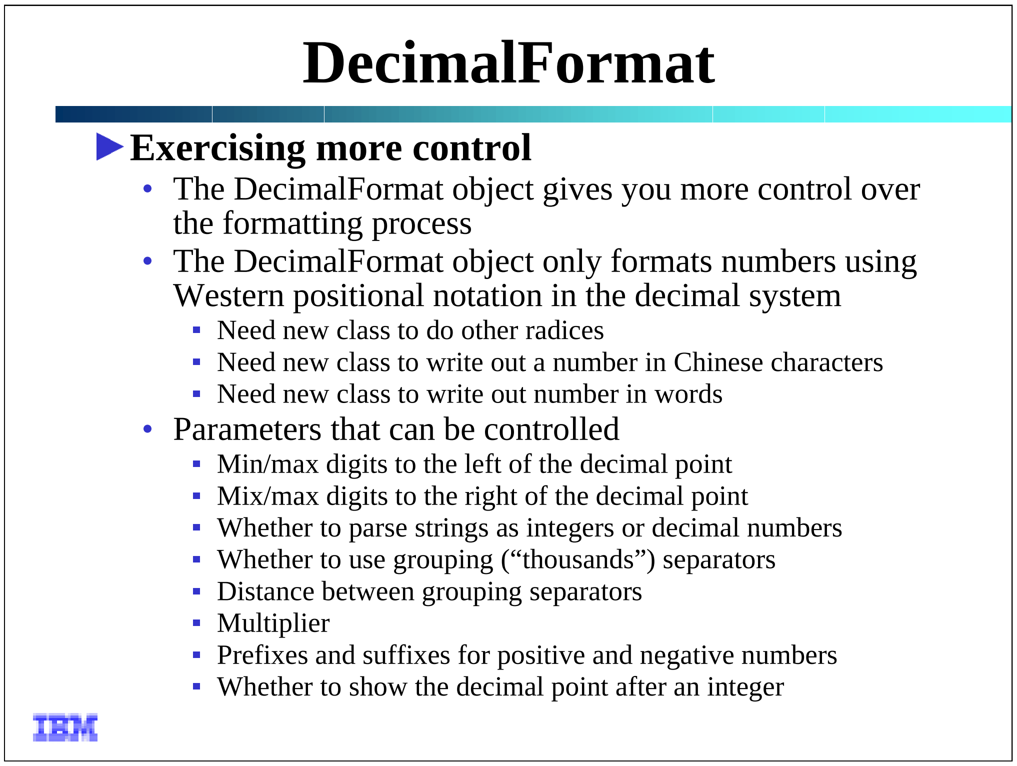
- **All formatters both format and parse**
  `00001111` ⟶ “31” ⟶ `00001111`
  - `public final String format(Object obj);`
  - `public Object parseObject(String source);`
- **Most formatters provide convenience methods**
  - `public final String format(long number);`
  - `public final String format(double number);`
- **`NumberFormat` provides four factory methods**
  - `NumberFormat.getInstance()`
  - `NumberFormat.getNumberInstance()`
  - `NumberFormat.getPercentInstance()`
  - `NumberFormat.getCurrencyInstance()`

NumberFormat and all other formatting objects are designed both to format (convert data from the internal format into user-readable text) and parse (convert user-readable text back into the internal format). There is a family of format() and parse() functions to do these things.

Note that parsing will often, but not always, produce the same value you passed into the formatter when you parse its output. It generally depends on whether all of the information in the original value is still present in the formatted output.

Most formatters will provide convenience methods that take parameters more specific than Object. NumberFormat has methods that take a long and a double (the other types are all automatically upconverted).

There are four factory methods on NumberFormat: One formats numbers in a generic format, one formats them as currency values, and one formats them as percentages. getInstance() does the same thing as getNumberInstance().



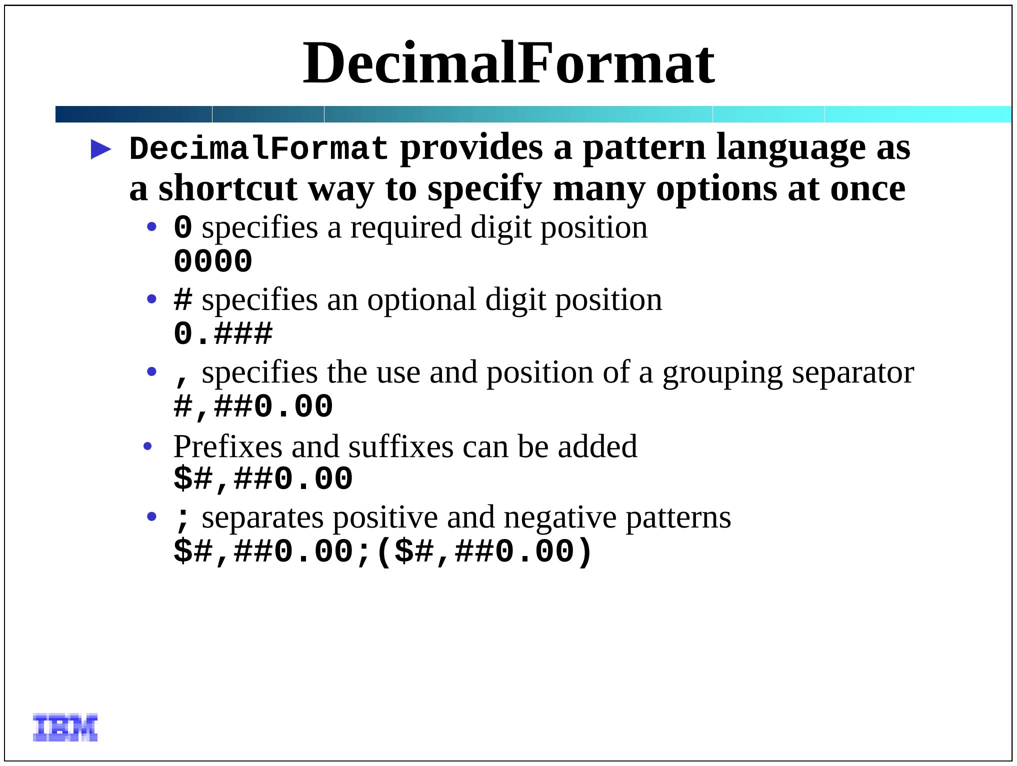
DecimalFormat

- **Exercising more control**
  - The DecimalFormat object gives you more control over the formatting process
  - The DecimalFormat object only formats numbers using Western positional notation in the decimal system
    - Need new class to do other radices
    - Need new class to write out a number in Chinese characters
    - Need new class to write out number in words
  - Parameters that can be controlled
    - Min/max digits to the left of the decimal point
    - Mix/max digits to the right of the decimal point
    - Whether to parse strings as integers or decimal numbers
    - Whether to use grouping ("thousands") separators
    - Distance between grouping separators
    - Multiplier
    - Prefixes and suffixes for positive and negative numbers
    - Whether to show the decimal point after an integer

If you want more control over the result than just the generic format for a given type and locale, you can use DecimalFormat, the main implementation class for NumberFormat directly. This class is used to format numbers using standard Western positional notation and the decimal numeration system. This covers almost all languages.

DecimalFormat lets you control many aspects of the output, including the minimum and maximum number of digits on either side of the decimal point, whether to separate thousands, ten-thousands, or nothing, whether to add prefixes or suffixes to numbers, whether to use a scaling factor (percentage formatters use a scaling factor of 100), and many other things.

You'll need a different subclass of NumberFormat to do some things, such as formatting in non-decimal radices, formatting numbers in Chinese characters, or formatting numbers into words.

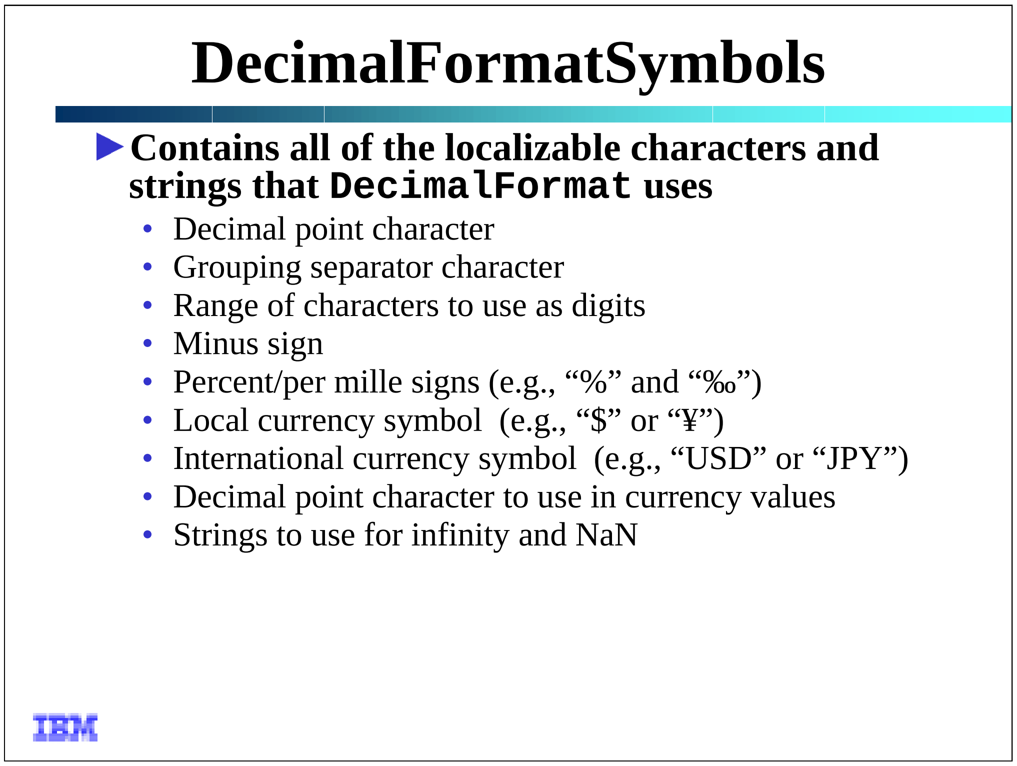


DecimalFormat

- **`DecimalFormat` provides a pattern language as a shortcut way to specify many options at once**
  - `0` specifies a required digit position
    `0000`
  - `#` specifies an optional digit position
    `0.###`
  - `,` specifies the use and position of a grouping separator
    `#,##0.00`
  - Prefixes and suffixes can be added
    `$#,##0.00`
  - `;` separates positive and negative patterns
    `$#,##0.00;($#,##0.00)`

You can also change many of these settings in a single call by using a pattern: a template describing the desired result. In fact, the built-in number formatters produced by NumberFormat's factory methods all load patterns from resource bundles to get their behavior.

This slide shows a sampling of the most important pattern characters and how they work together to specify different formats.

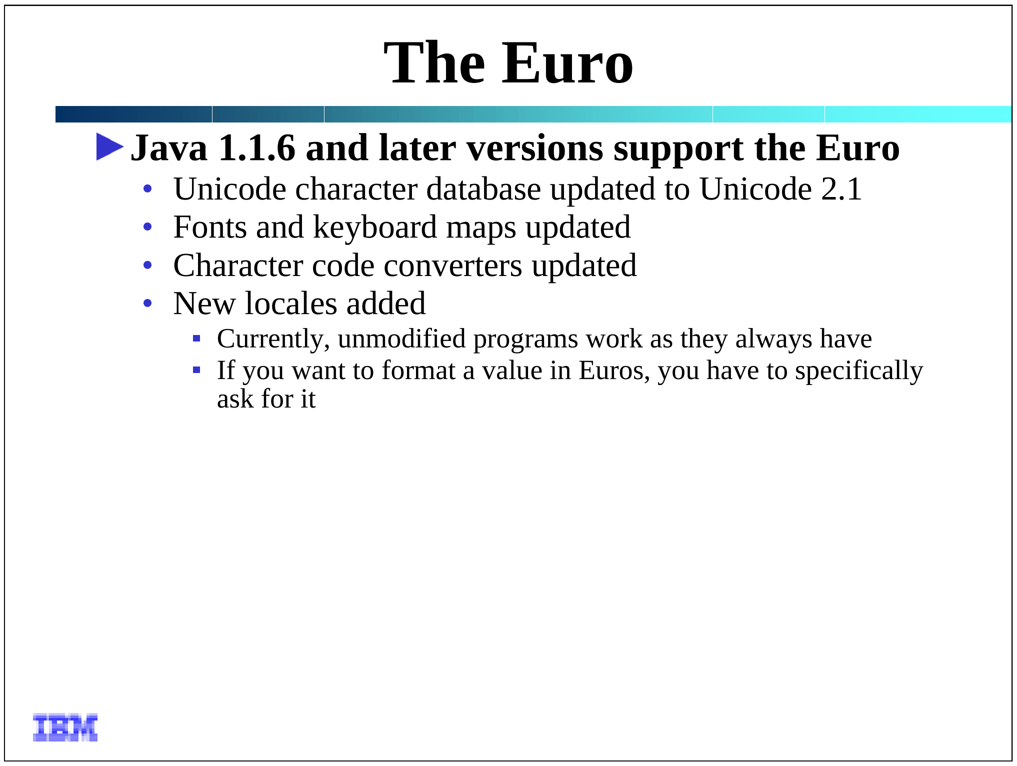


DecimalFormatSymbols

- **Contains all of the localizable characters and strings that `DecimalFormat` uses**
  - Decimal point character
  - Grouping separator character
  - Range of characters to use as digits
  - Minus sign
  - Percent/per mille signs (e.g., “%” and “‰”)
  - Local currency symbol (e.g., “$” or “¥”)
  - International currency symbol (e.g., “USD” or “JPY”)
  - Decimal point character to use in currency values
  - Strings to use for infinity and NaN

IBM

The actual characters to use in the output are specified using a DecimalFormatSymbols object, which is also usually loaded from a resource. This slide shows the various parameters stored in a DecimalFormatSymbols object.

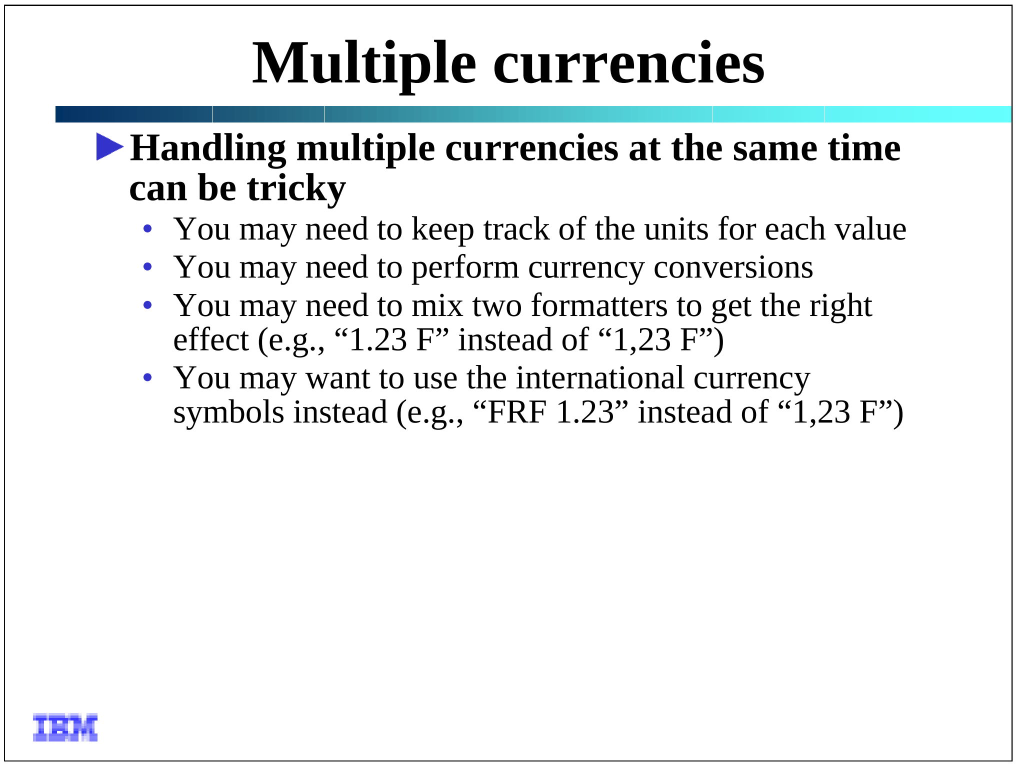


The Euro

- **Java 1.1.6 and later versions support the Euro**
  - Unicode character database updated to Unicode 2.1
  - Fonts and keyboard maps updated
  - Character code converters updated
  - New locales added
    - Currently, unmodified programs work as they always have
    - If you want to format a value in Euros, you have to specifically ask for it

Support for the Euro currency was added to Java in version 1.1.6. This involved updating fonts and keyboard layouts so you could display and type it, updating character converters to support other encodings that support the Euro, and updating the internal Unicode tables to conform to Unicode version 2.1, which added the Euro to Unicode.

In addition, new resource bundles were added for the countries using the Euro currency. You can get a currency formatter that formats numbers as numbers of Euros by specifying a locale ID with a variant code of “EURO.” Currently, we only support those countries actually using the Euro; we don’t yet support those (such as the UK) that might support it in the future.

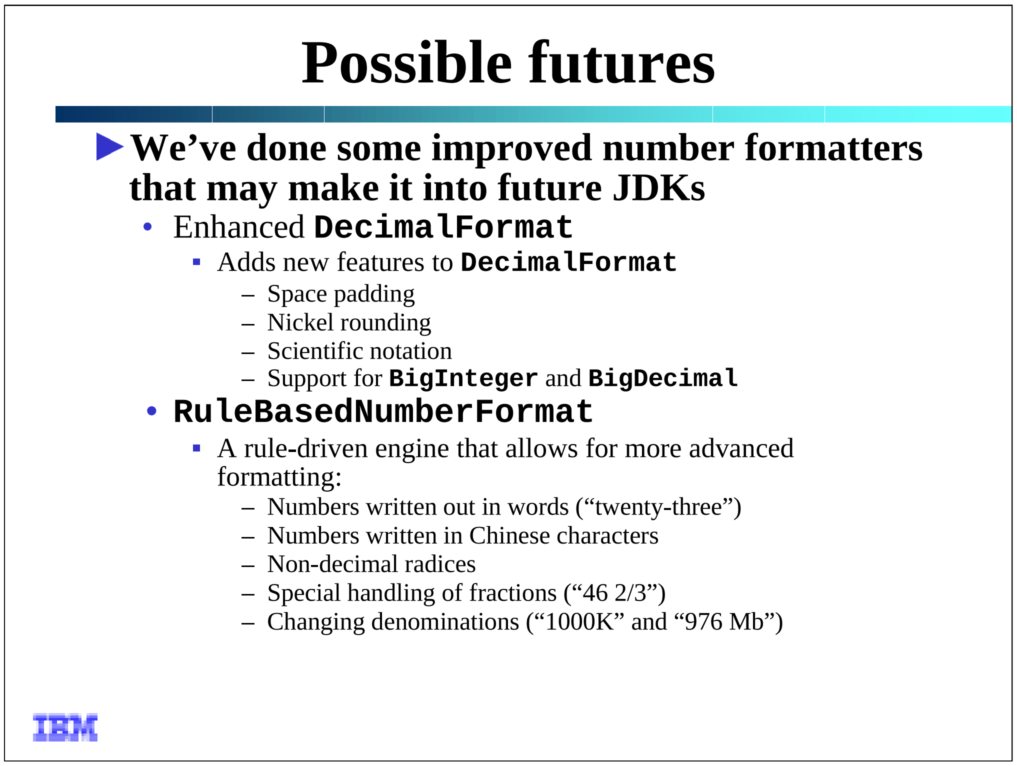


Multiple currencies

- **Handling multiple currencies at the same time can be tricky**
  - You may need to keep track of the units for each value
  - You may need to perform currency conversions
  - You may need to mix two formatters to get the right effect (e.g., “1.23 F” instead of “1,23 F”)
  - You may want to use the international currency symbols instead (e.g., “FRF 1.23” instead of “1,23 F”)

Some applications may want to format currency values denominated in different units. The values may all be stored in the same currency and just translated on output, or they may also be stored in different currencies. In either case, you have to do a currency conversion, something Java can’t do for you.

If different values are denominated in different currencies, you’ll probably also need to tag each value with the currency it’s in. You may want to mix two currency formats (to show a value in French francs to an American user using the American decimal-point character, for example), or you may just want to fall back on the ISO three-letter currency symbols. This is all possible, but requires some extra work: there are no convenience methods to help with this.



Possible futures

- **We've done some improved number formatters that may make it into future JDKs**
  - Enhanced **`DecimalFormat`**
    - Adds new features to **`DecimalFormat`**
      - Space padding
      - Nickel rounding
      - Scientific notation
      - Support for **`BigInteger`** and **`BigDecimal`**
  - **`RuleBasedNumberFormat`**
    - A rule-driven engine that allows for more advanced formatting:
      - Numbers written out in words ("twenty-three")
      - Numbers written in Chinese characters
      - Non-decimal radices
      - Special handling of fractions ("46 2/3")
      - Changing denominations ("1000K" and "976 Mb")

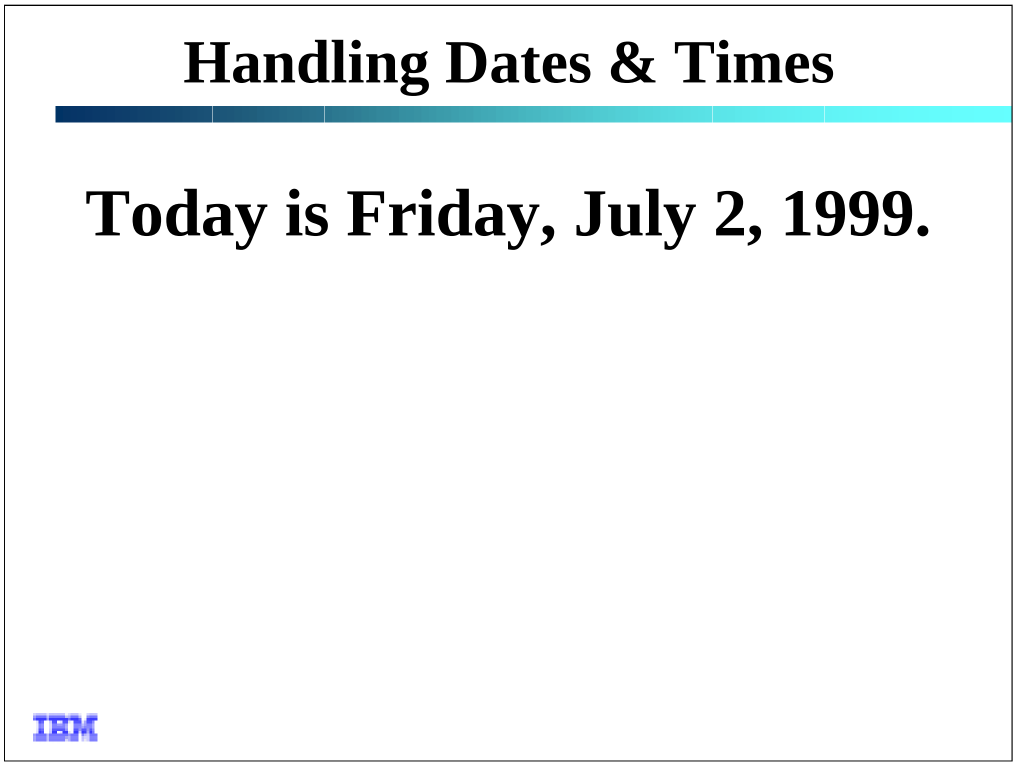
IBM

Our group at IBM has done two new number formatters of our own. One is an enhanced version of DecimalFormat that adds space padding, scientific notation, nickel rounding, and support for BigInteger and BigDecimal to the current version of DecimalFormat. We're still negotiating to get this into the JDK.

We also have something called RuleBasedNumberFormat, a more complicated formatting engine that allows for more exotic formats such as Chinese characters, words, alternate radices, special handling of fractions, and values with changing denominations, among other things. We're still not sure of the ultimate fate of this object.

Trial versions of both formatters are available at IBM's AlphaWorks Web site.

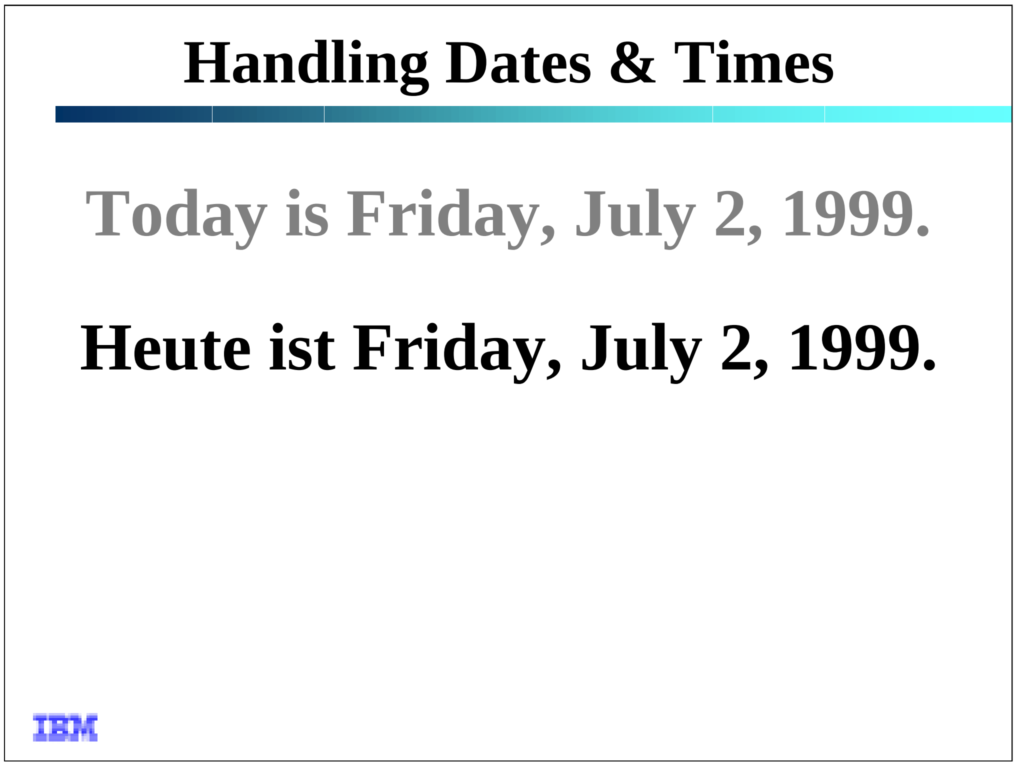
Handling Dates and Times



Handling Dates & Times

Today is Friday, July 2, 1999.

Displaying dates and times has many of the same challenges as displaying numbers. Consider a message like this. Again, it consists of both a static and a dynamic part...

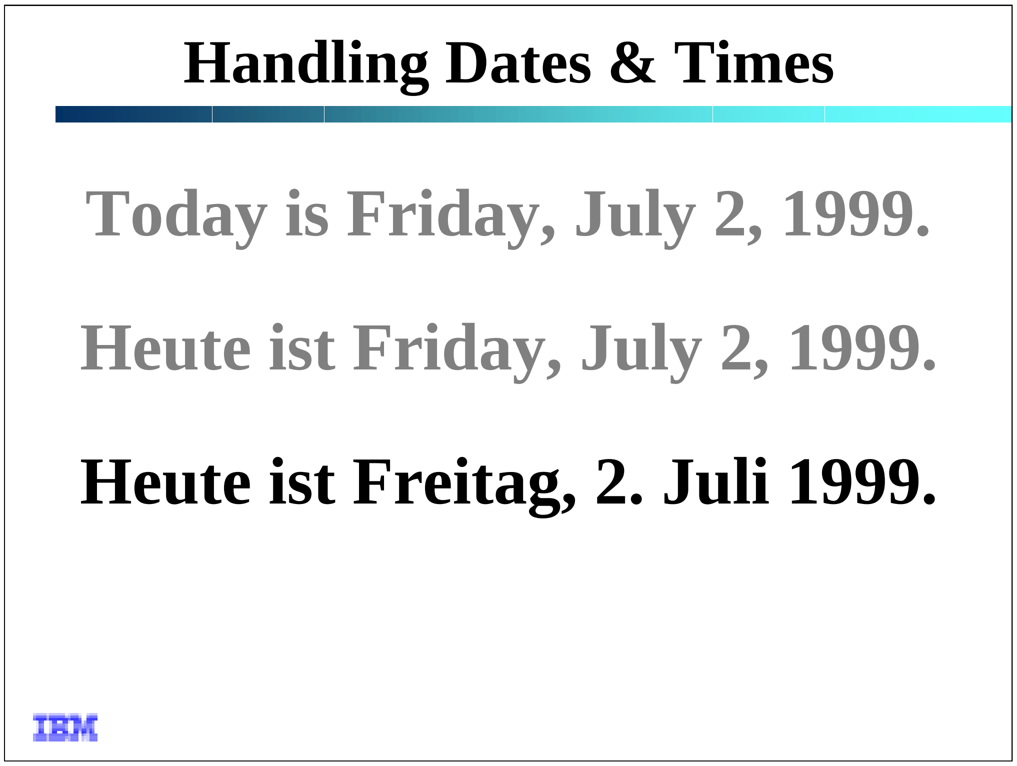


Handling Dates & Times

Today is Friday, July 2, 1999.

Heute ist Friday, July 2, 1999.

...and it doesn't work to just translate the static part.



Handling Dates & Times

Today is Friday, July 2, 1999.

Heute ist Friday, July 2, 1999.

Heute ist Freitag, 2. Juli 1999.

What a German speaker would really like to see is this, with both the message and the date itself translated into German.

Handling Dates & Times

Today is Friday, July 2, 1999.

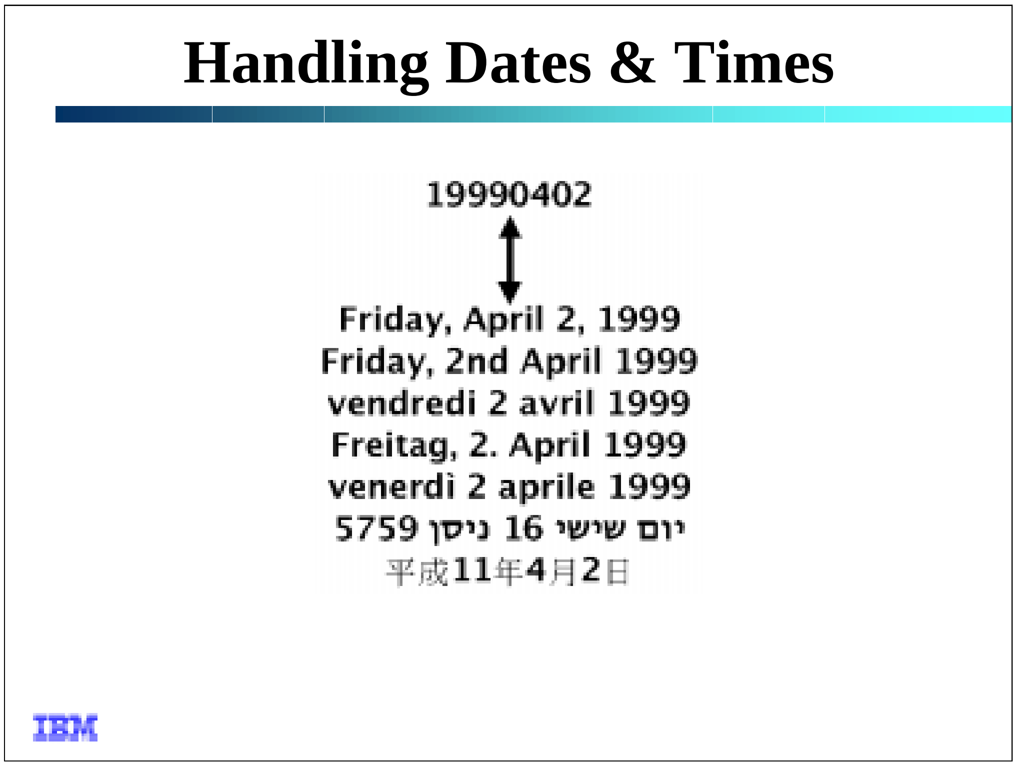
Heute ist Friday, July 2, 1999.

Heute ist Freitag, 2. Juli 1999.

Today is Freitag, 2. Juli 1999.

IBM

In fact, if you're a German and you're running a program that hasn't actually been translated into German, it's probably more desirable to see this.



Handling Dates & Times

19990402

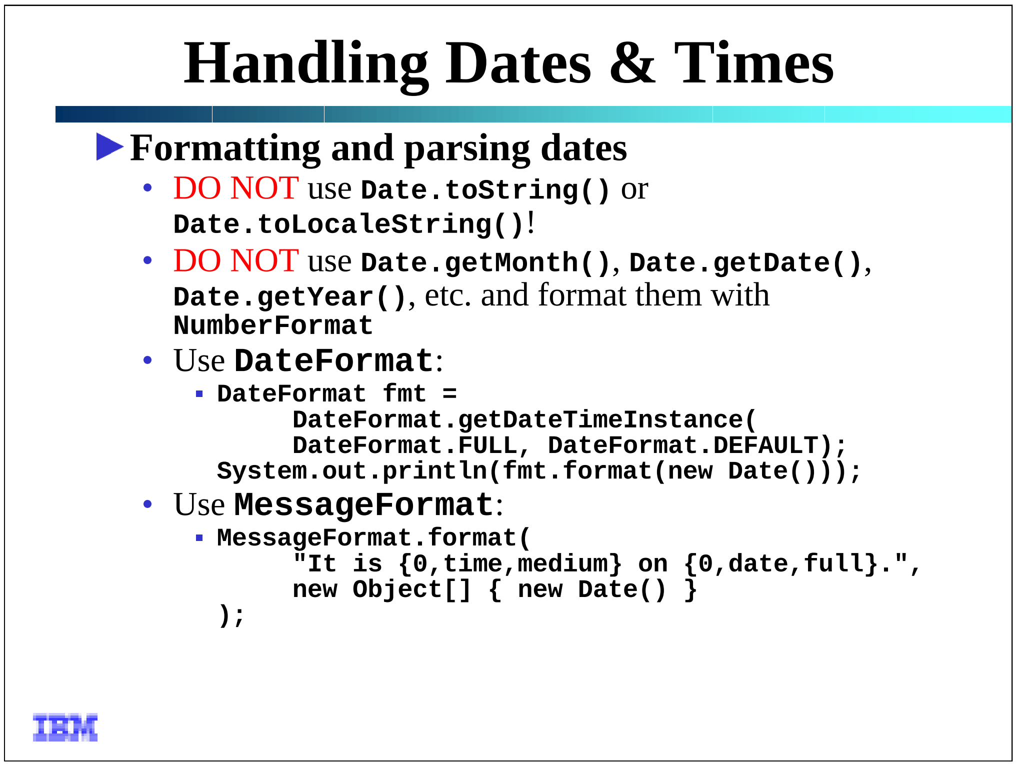
$\updownarrow$

Friday, April 2, 1999
Friday, 2nd April 1999
vendredi 2 avril 1999
Freitag, 2. April 1999
venerdì 2 aprile 1999
יום שישי 16 ניסן 5759
平成11年4月2日

IBM

Again, the displayed forms of the dates can vary quite a bit from language to language. Not only do the words for the days and months change, but so does the order of the fields themselves and the punctuation around them. In fact, in some countries, the calendar system in use also changes: In Hebrew, for example, April 2, 1999 is the 16th of Nisan, 5759. Japan has changed to use our Gregorian calendar, but they number their years by the reigns of the emperors: 1999 is 11 Heisei in Japan.

So, just as with numbers, you don’t want to do date and time formatting on your own. Again, Java provides an extensive framework of tools to let you handle dates and times properly.



Handling Dates & Times

Formatting and parsing dates

- DO NOT use `Date.toString()` or `Date.toLocaleString()`!
- DO NOT use `Date.getMonth()`, `Date.getDate()`, `Date.getYear()`, etc. and format them with `NumberFormat`
- Use `DateFormat`:

```
DateFormat fmt =
     DateFormat.getDateTimeInstance(
     DateFormat.FULL, DateFormat.DEFAULT);
System.out.println(fmt.format(new Date()));
```

- Use `MessageFormat`:

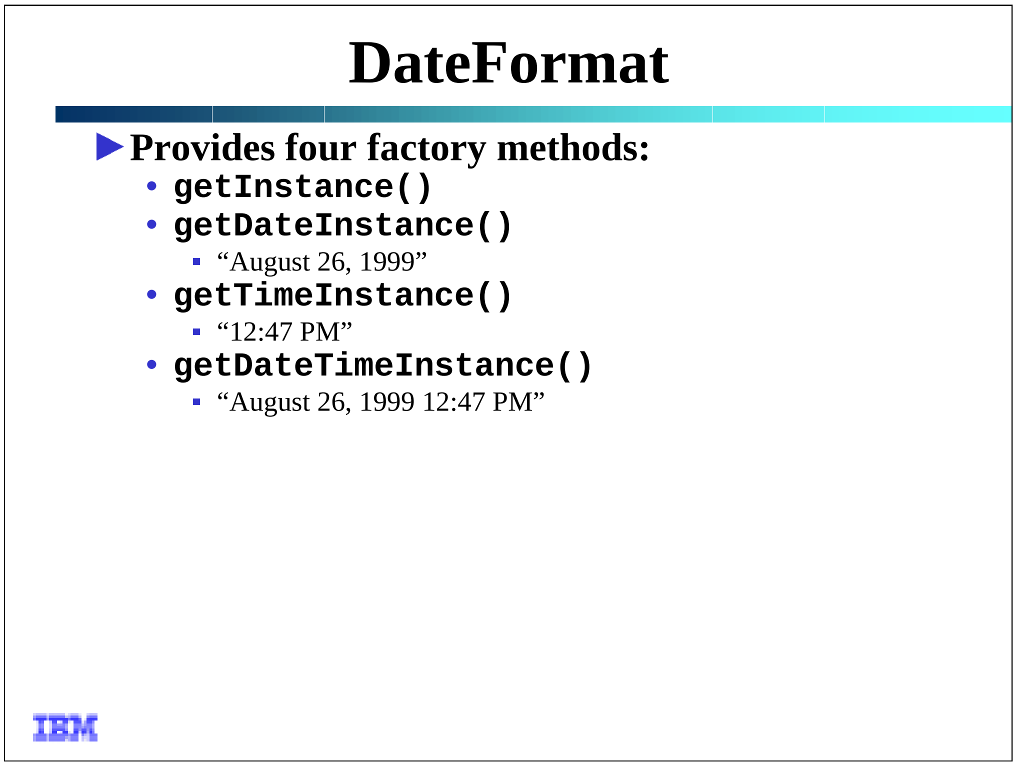
```
MessageFormat.format(
     "It is {0,time,medium} on {0,date,full}.",
     new Object[] { new Date() }
);
```

So you want to avoid using functions like toString(). Even more importantly, you don’t want to decompose the date into fields using the methods on Date and then format each field individually.

Instead, use DateFormat. Again, it has factory methods you call to get appropriate DateFormat objects for your locale. A DateFormat actually formats both dates and times (which are stored together in a Date object), so there are separate factory methods to get objects that control whether you see just the date, just the time, or both. DateFormat also offers a selection of formats (short, medium, long, and full), and you can set them independently for the date and time.

Again, you can also access all these features through MessageFormat by specifying extra options in the {} sequences. In the example, we’re using the same parameter (a Date object that has been initialized to “now”) in two different substitutions: the first shows just the time and the second shows just the time.

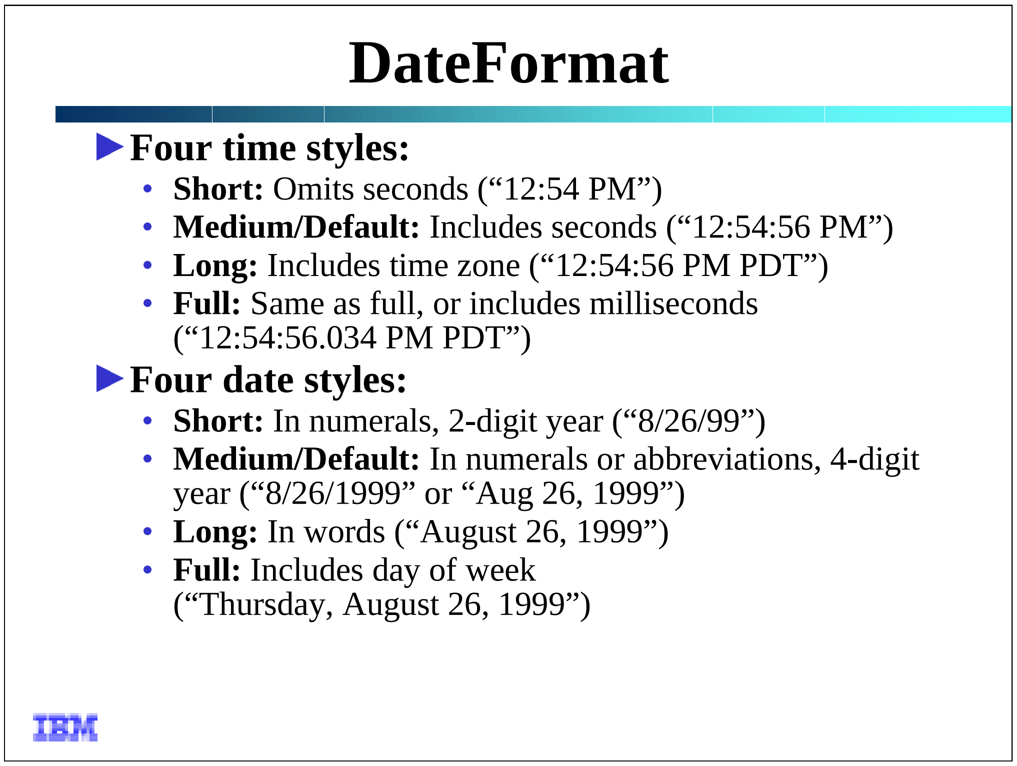
Just as with NumberFormat, there is extensive API on DateFormat and its concrete subclass SimpleDateFormat for customizing the output or behavior of the formatter.



DateFormat

- **Provides four factory methods:**
  - **`getInstance()`**
  - **`getDateInstance()`**
    - “August 26, 1999”
  - **`getTimeInstance()`**
    - “12:47 PM”
  - **`getDateTimeInstance()`**
    - “August 26, 1999 12:47 PM”

The abstract DateFormat class has four factory methods, that produce formatters that show either the “date” part of the date value (a Date specifies a point of time within a range of millennia with millisecond resolution, meaning it contains both date and time information), the “time” part of the value, or both. (getInstance() is the same as getDateTimeInstance().)

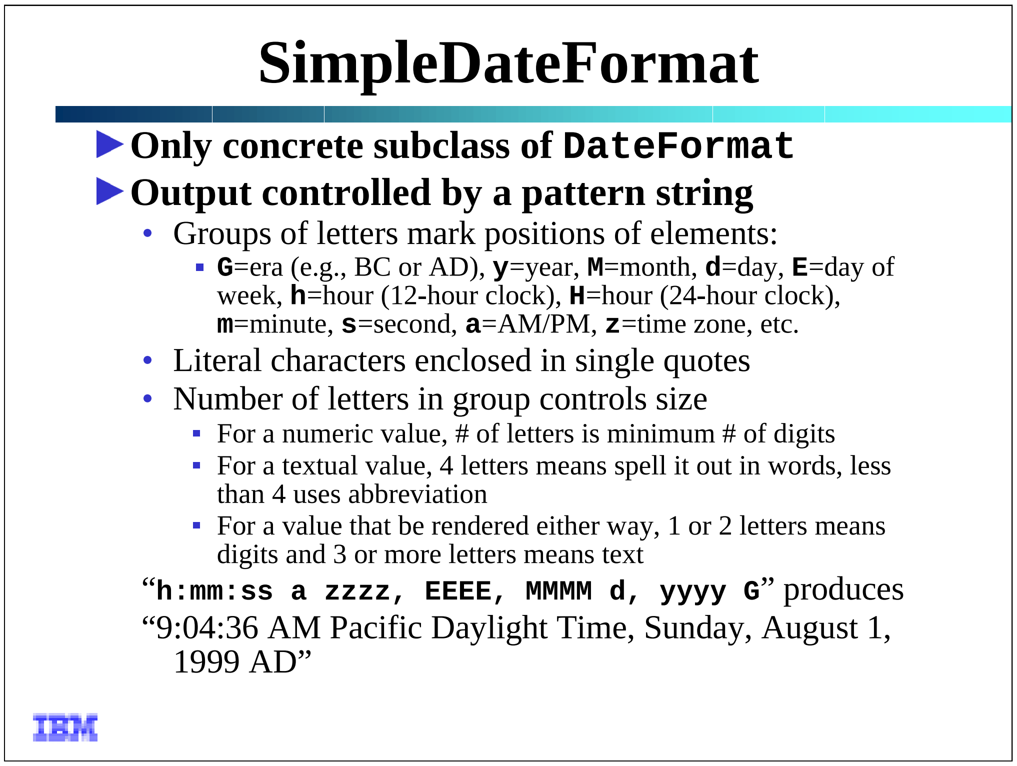


DateFormat

- **Four time styles:**
  - **Short:** Omits seconds (“12:54 PM”)
  - **Medium/Default:** Includes seconds (“12:54:56 PM”)
  - **Long:** Includes time zone (“12:54:56 PM PDT”)
  - **Full:** Same as full, or includes milliseconds (“12:54:56.034 PM PDT”)
- **Four date styles:**
  - **Short:** In numerals, 2-digit year (“8/26/99”)
  - **Medium/Default:** In numerals or abbreviations, 4-digit year (“8/26/1999” or “Aug 26, 1999”)
  - **Long:** In words (“August 26, 1999”)
  - **Full:** Includes day of week (“Thursday, August 26, 1999”)

IBM

Each factory method lets you specify a style for the time part and a separate style for the date part. The meanings of the various styles are shown above. (“Medium” and “Default” are always the same.)



SimpleDateFormat

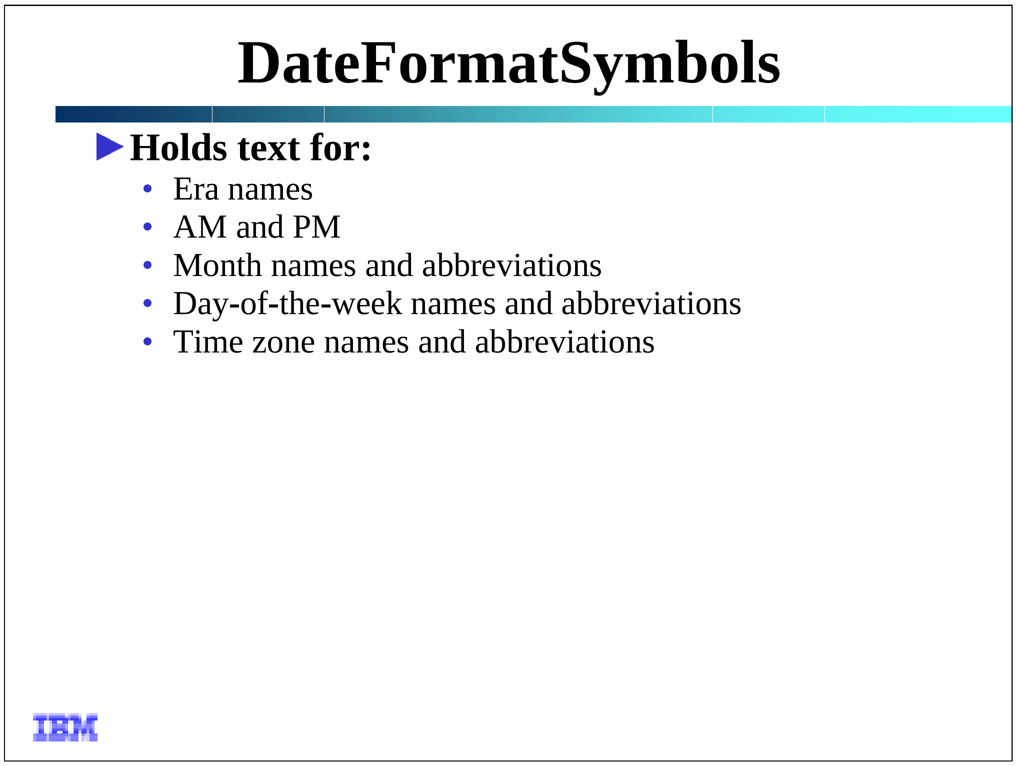
- **Only concrete subclass of `DateFormat`**
- **Output controlled by a pattern string**
  - Groups of letters mark positions of elements:
    - **`G`**=era (e.g., BC or AD), **`y`**=year, **`M`**=month, **`d`**=day, **`E`**=day of week, **`h`**=hour (12-hour clock), **`H`**=hour (24-hour clock), **`m`**=minute, **`s`**=second, **`a`**=AM/PM, **`z`**=time zone, etc.
  - Literal characters enclosed in single quotes
  - Number of letters in group controls size
    - For a numeric value, # of letters is minimum # of digits
    - For a textual value, 4 letters means spell it out in words, less than 4 uses abbreviation
    - For a value that be rendered either way, 1 or 2 letters means digits and 3 or more letters means text

"**`h:mm:ss a zzzz, EEEE, MMMM d, yyyy G`**" produces "9:04:36 AM Pacific Daylight Time, Sunday, August 1, 1999 AD"

Again, the implementation class of DateFormat, SimpleDateFormat, is also public, allowing finer-grained control over date/time formatting than the DateFormat factory methods give you.

SimpleDateFormat's behavior is controlled by a pattern string that acts as a template for the desired result. The pattern string includes tokens specifying different possible "fields" of the date, such as day or month or hour of day (there are many to choose from), punctuation or boilerplate text, and their relative orders. The tokens also usually allow for several alternative representations for their value.

The canned date formats are all based on pattern strings in resources.

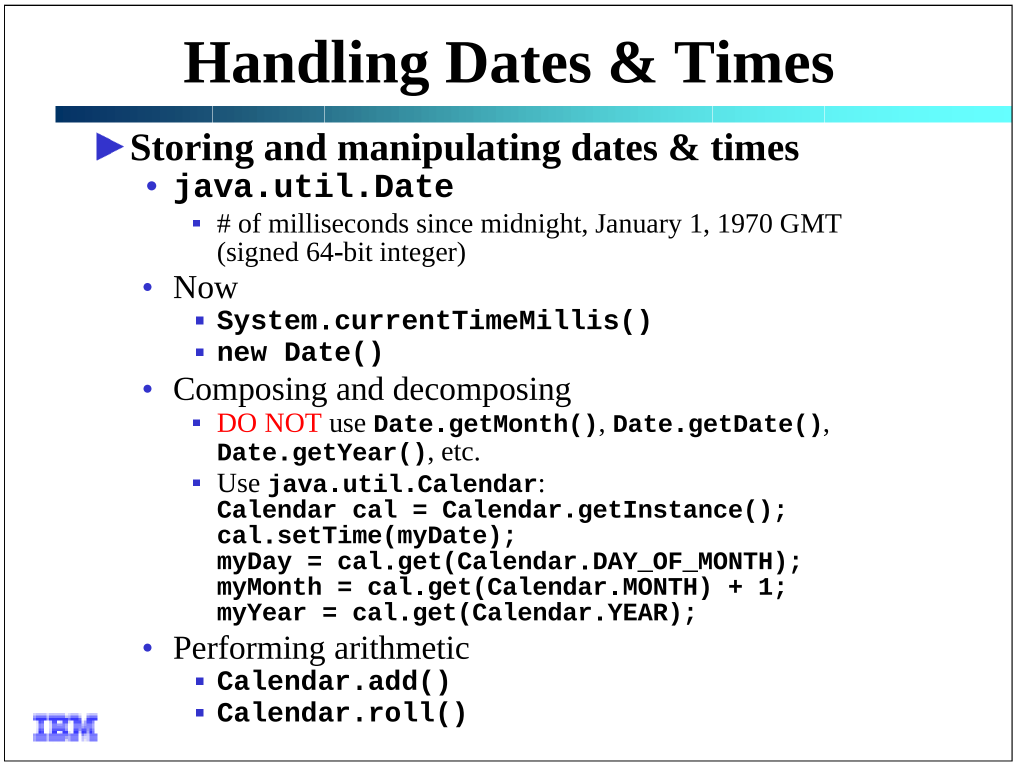


DateFormatSymbols

- **Holds text for:**
  - Era names
  - AM and PM
  - Month names and abbreviations
  - Day-of-the-week names and abbreviations
  - Time zone names and abbreviations

IBM

SimpleDateFormat has a DateFormatSymbols object associated with it that contains the actual words and abbreviations used for certain field values. The symbols object is also usually loaded from a resource bundle.



Handling Dates & Times

- **Storing and manipulating dates & times**
  - `java.util.Date`
    - # of milliseconds since midnight, January 1, 1970 GMT (signed 64-bit integer)
  - Now
    - `System.currentTimeMillis()`
    - `new Date()`
  - Composing and decomposing
    - DO NOT use `Date.getMonth()`, `Date.getDate()`, `Date.getYear()`, etc.
    - Use `java.util.Calendar`:

```
Calendar cal = Calendar.getInstance();
cal.setTime(myDate);
myDay = cal.get(Calendar.DAY_OF_MONTH);
myMonth = cal.get(Calendar.MONTH) + 1;
myYear = cal.get(Calendar.YEAR);
```

  - Performing arithmetic
    - `Calendar.add()`
    - `Calendar.roll()`

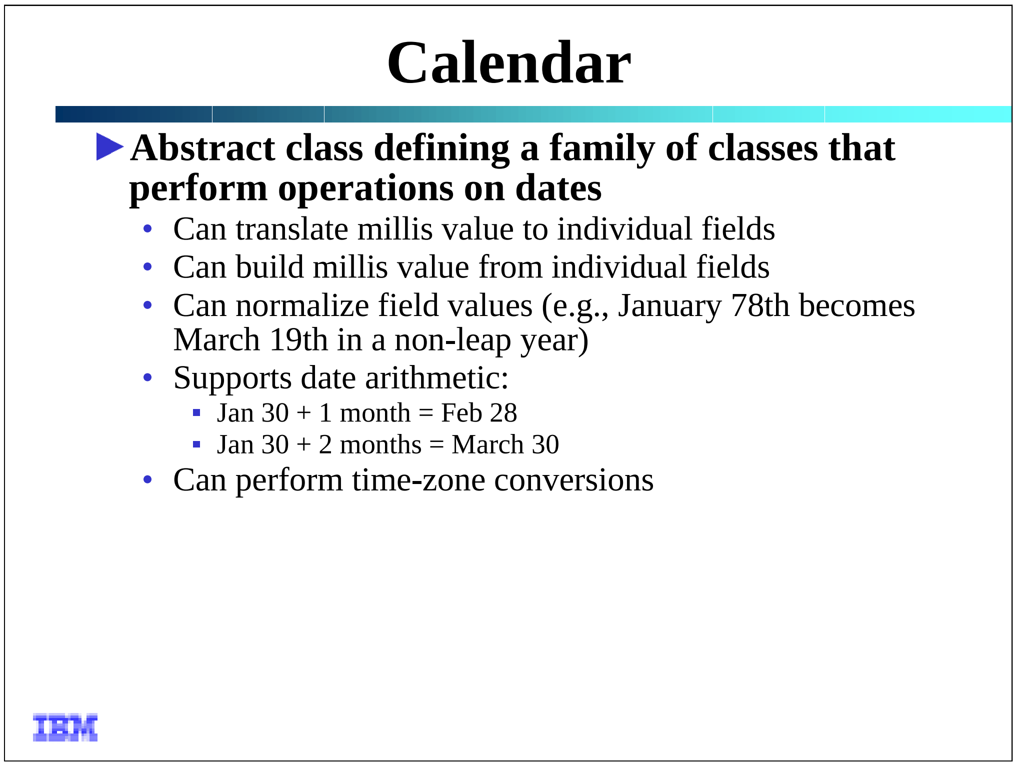
With dates, you also have the additional problem of making sure your internal processing code doesn’t contain any hidden locale assumptions when manipulating the values.

Java provides a built-in class called Date that’s used for storing dates and times. Be sure you use this for all date storage, not some ad-hoc format. The Java Date format is completely locale-independent and Y2K-proof. All it is is the number of milliseconds before or since midnight, January 1, 1970 GMT. Notice that dates are always stored internally as GMT regardless of time zone.

There are two APIs for obtaining the current date and time. One, System.currentTimeMillis(), returns a number of milliseconds, and this can’t be formatted without converting it into a date (the raw value is useful for things like timing tests, but not for displayable dates and times). The default constructor on the Date object, on the other hand, creates a Date object using System.currentTimeMillis().

Date provides a pretty good API for decomposing a date into individual fields and the reverse. **DON’T USE IT.** This is because Date contains the hidden assumption that all countries use the Gregorian calendar. Instead, use the Calendar object to convert between fields and millis. The API is better, and it’ll work with multiple calendar systems.

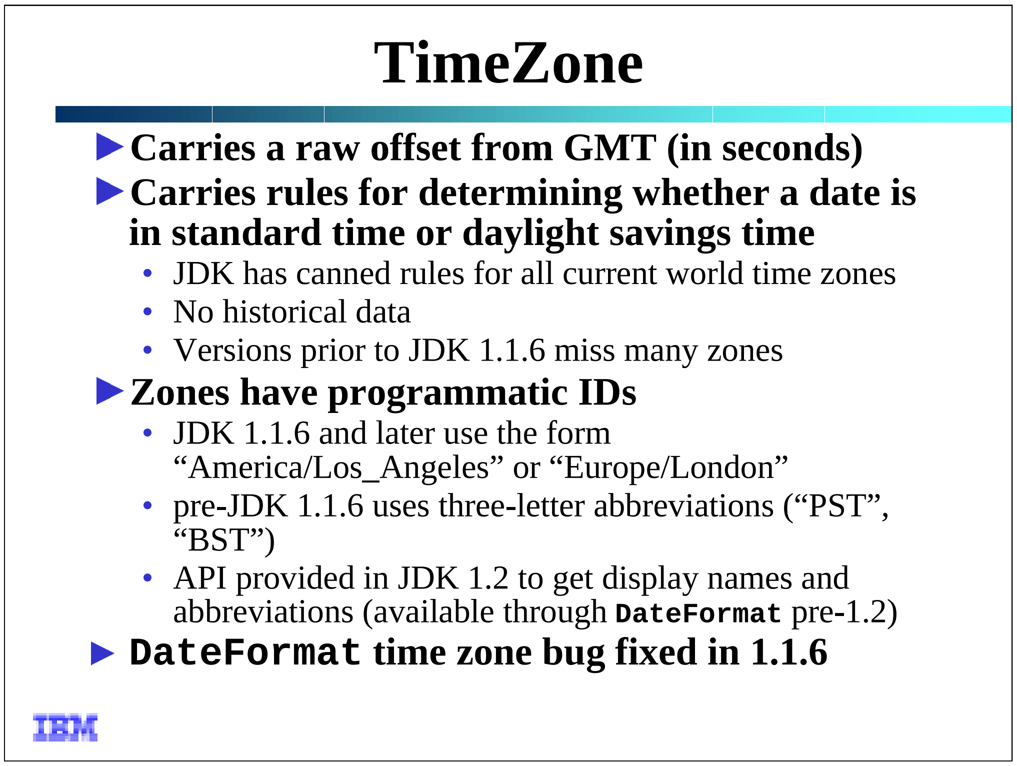
You also don’t want to do arithmetic directly on a number of millis. Consider adding a month. You have no way of knowing how long any particular month is. Non-Gregorian calendars also have this type of problem, but in different fields and in different ways. Calendar provides add() and roll() methods for altering individual fields, and also provides API to do things like time zone conversions.



Calendar

- **Abstract class defining a family of classes that perform operations on dates**
  - Can translate millis value to individual fields
  - Can build millis value from individual fields
  - Can normalize field values (e.g., January 78th becomes March 19th in a non-leap year)
  - Supports date arithmetic:
    - Jan 30 + 1 month = Feb 28
    - Jan 30 + 2 months = March 30
  - Can perform time-zone conversions

The Calendar class defines the algorithms to be used for deriving field values (such as hour of day or day or month) from a number of millis (a raw date value), or deriving a number of millis from a set of field values. These capabilities can be used to perform accurate date arithmetic., including time zone conversions.



TimeZone

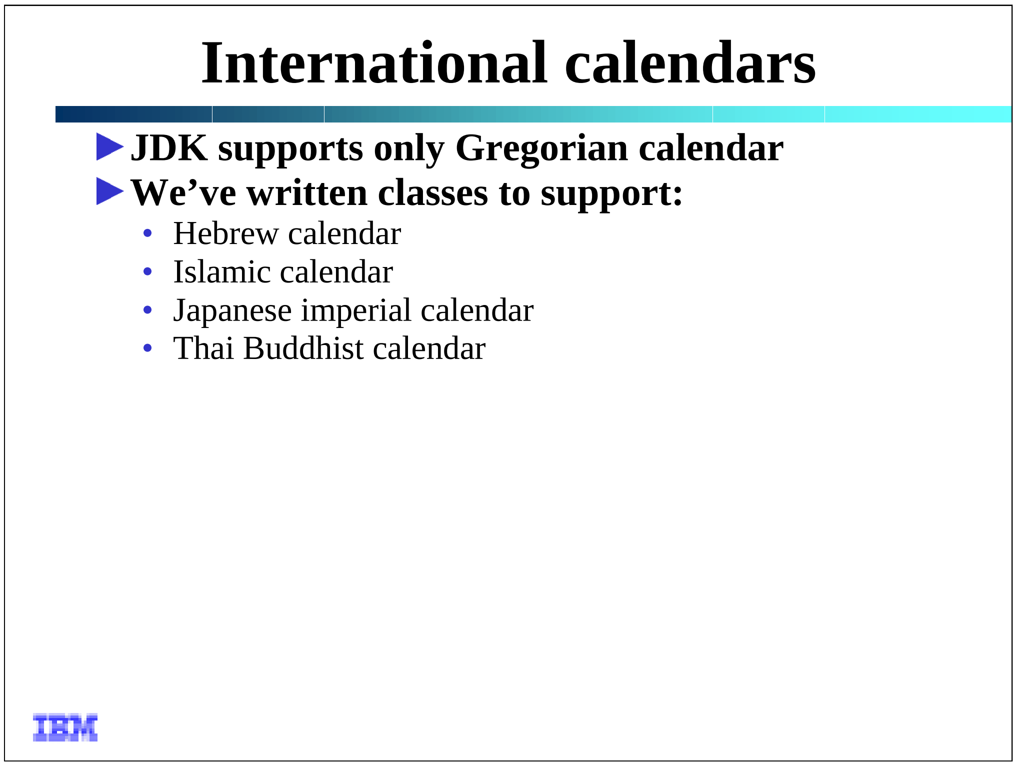
- **Carries a raw offset from GMT (in seconds)**
- **Carries rules for determining whether a date is in standard time or daylight savings time**
  - JDK has canned rules for all current world time zones
  - No historical data
  - Versions prior to JDK 1.1.6 miss many zones
- **Zones have programmatic IDs**
  - JDK 1.1.6 and later use the form "America/Los_Angeles" or "Europe/London"
  - pre-JDK 1.1.6 uses three-letter abbreviations ("PST", "BST")
  - API provided in JDK 1.2 to get display names and abbreviations (available through `DateFormat` pre-1.2)
- **`DateFormat` time zone bug fixed in 1.1.6**

IBM

Calendar also uses an auxiliary object called TimeZone to specify a time zone (the internal value is always in GMT). A TimeZone object carries not only an offset (in seconds) from GMT, but also rules for calculating the beginning and end of Daylight Savings Time (and an additional offset to use during Daylight Savings Time). In JDK 1.1.6, we provide a complete set of canned TimeZones representing all the current world time zones (there's a different TimeZone for every jurisdiction with different DST rules). These TimeZones all have standard internal identifiers, and many aliases are also supplied (for example, "Asia/Tokyo" and "Asia/Seoul", although they have the same offset and DST rules, are both valid identifiers).

Prior to 1.1.6, a lot of time zones were missing, causing some weird behavior, contrived three-letter IDs were used (they're still supported for compatbility), and there was a bug in DateFormat that caused it to format everything according to an arbitrary default time zone for the default locale. This bug was fixed, so that TimeZone.getDefault(), which returns the current time zone setting from the underlying host environment, is used instead.

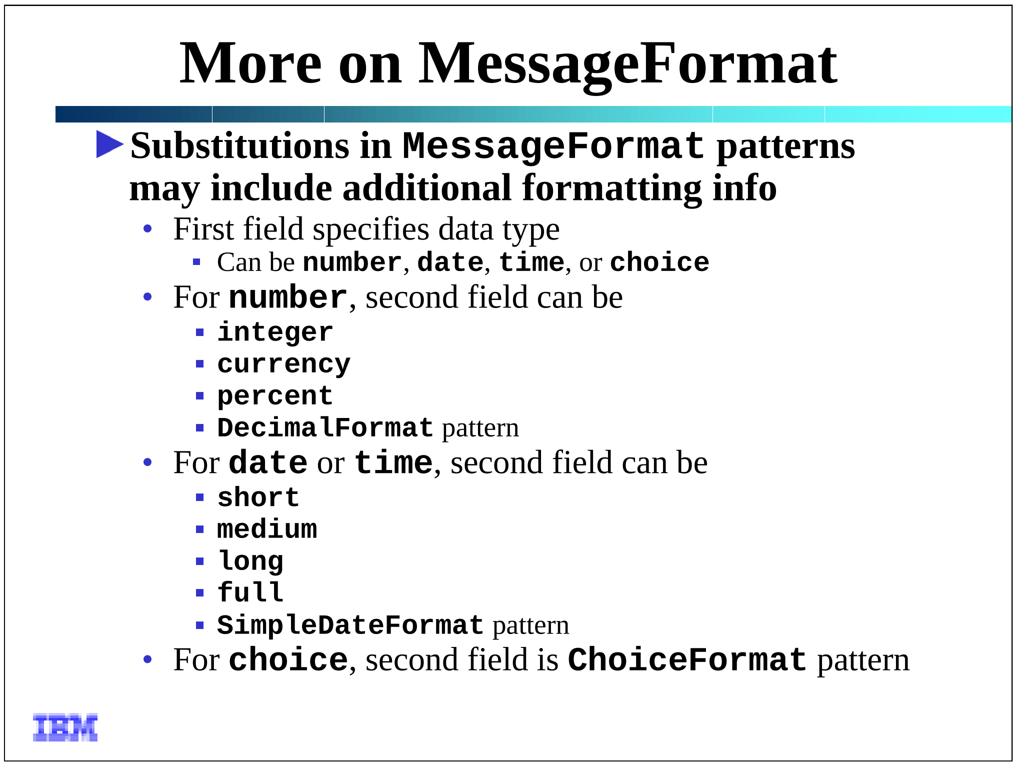
In JDK 1.2, we also added a getDisplayName() function to TimeZone. In prior versions, the display names could be accessed through SimpleDateFormat, but that wasn't obvious to anyone.



International calendars

- ▶ **JDK supports only Gregorian calendar**
- ▶ **We’ve written classes to support:**
  - Hebrew calendar
  - Islamic calendar
  - Japanese imperial calendar
  - Thai Buddhist calendar

The only concrete subclass of Calendar in the JDK is GregorianCalendar, which is fine for most things, but it’s not the only calendar system in use in the world (it’s not even the only one in use in business). We’ve put together classes that handle several other calendar systems, including the Hebrew, Islamic, and Japense calendars. There are also available on AlphaWorks.



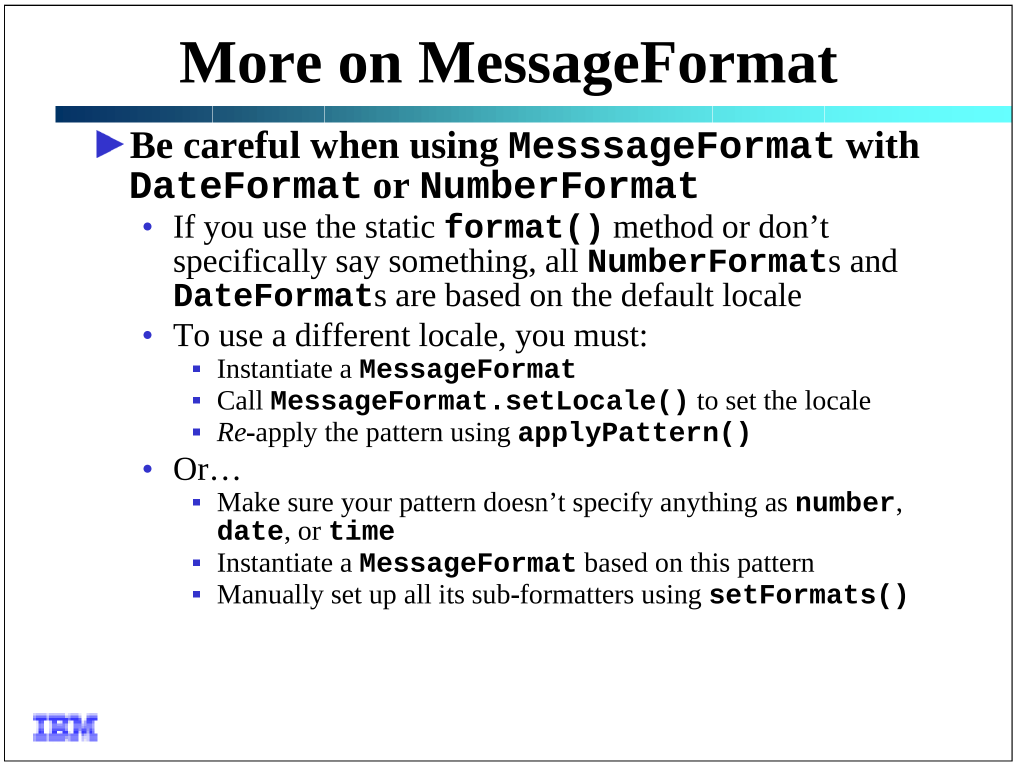
More on MessageFormat

- **Substitutions in `MessageFormat` patterns may include additional formatting info**
  - First field specifies data type
    - Can be **`number`**, **`date`**, **`time`**, or **`choice`**
  - For **`number`**, second field can be
    - **`integer`**
    - **`currency`**
    - **`percent`**
    - **`DecimalFormat`** pattern
  - For **`date`** or **`time`**, second field can be
    - **`short`**
    - **`medium`**
    - **`long`**
    - **`full`**
    - **`SimpleDateFormat`** pattern
  - For **`choice`**, second field is **`ChoiceFormat`** pattern

Now that we've had a chance to look at NumberFormat and DateFormat, I'd like to take another look at MessageFormat.

As I mentioned briefly before, the substitutions in a MessageFormat pattern can be qualified with information as to their type and desired output format. The type can be number, date, time, or choice. (Strings are always formatted as themselves.)

For "number," you can specify the format to be integer, currency, or percent, or you can specify a DecimalFormat pattern string. Likewise, "date" and "time" can both be qualified with short, medium, long, or full, or with a SimpleDateFormat pattern string.

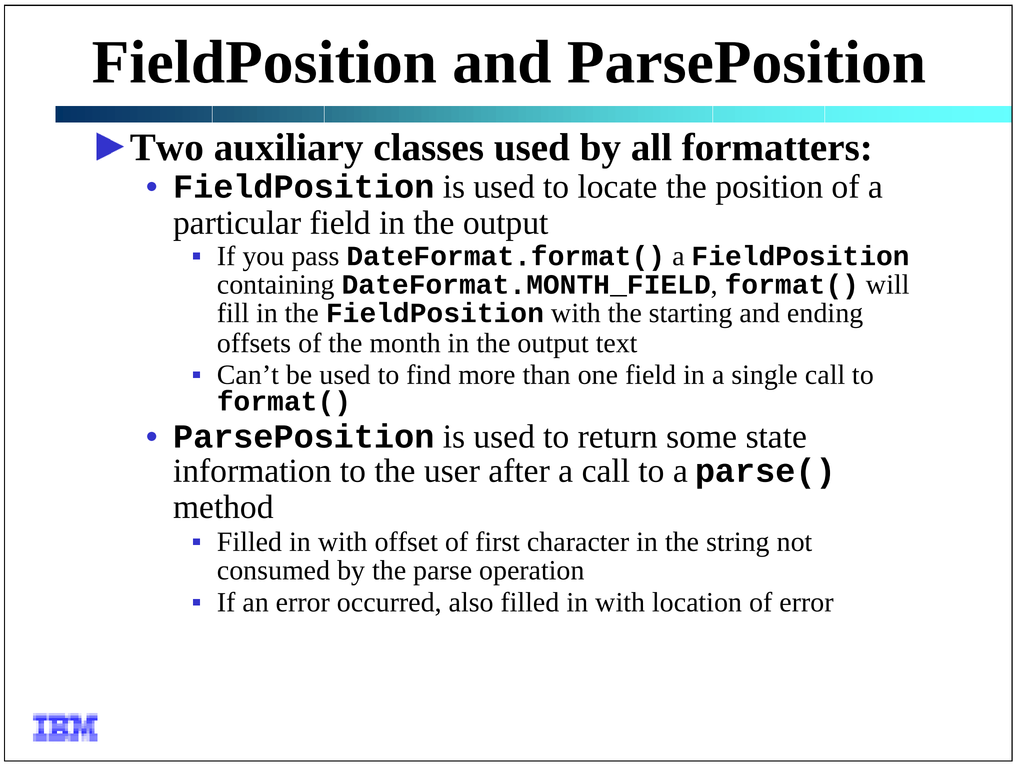


More on MessageFormat

- **Be careful when using `MesssageFormat` with `DateFormat` or `NumberFormat`**
  - If you use the static **`format()`** method or don't specifically say something, all **`NumberFormat`**s and **`DateFormat`**s are based on the default locale
  - To use a different locale, you must:
    - Instantiate a **`MessageFormat`**
    - Call **`MessageFormat.setLocale()`** to set the locale
    - *Re*-apply the pattern using **`applyPattern()`**
  - Or…
    - Make sure your pattern doesn't specify anything as **`number`**, **`date`**, or **`time`**
    - Instantiate a **`MessageFormat`** based on this pattern
    - Manually set up all its sub-formatters using **`setFormats()`**

But you have to be careful sometimes when you have number or date/time fields in a MessageFormat pattern. If you just specify the formats in the pattern, you always get the default locale's behavior. If you want some *other* locale's behavior instead, you have call setLocale() on the MessageFormat (which precludes using the static format() function), and *then* call applyPattern() to set the pattern (you can't do this in the opposite order).

Or you can simply manually create the subformatters yourself and pass them to the MessageFormat using its setFormats() method (this is useful in more exotic cases, such as when the fields aren't all going to use the same locale).



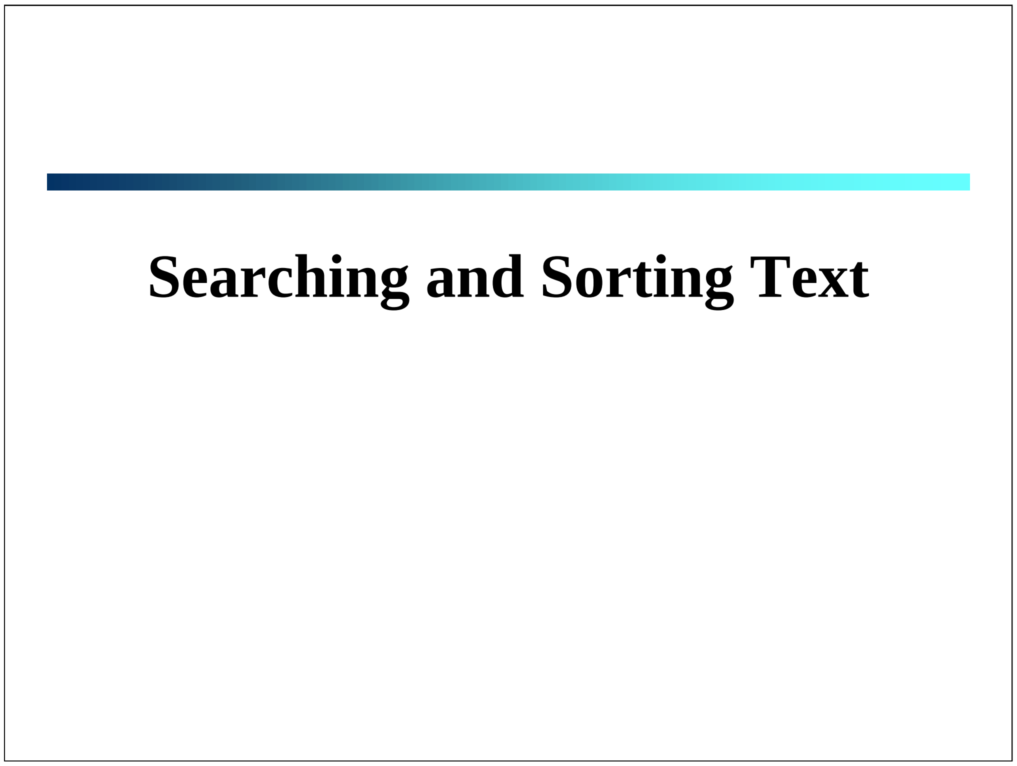
FieldPosition and ParsePosition

- **Two auxiliary classes used by all formatters:**
  - **`FieldPosition`** is used to locate the position of a particular field in the output
    - If you pass **`DateFormat.format()`** a **`FieldPosition`** containing **`DateFormat.MONTH_FIELD`**, **`format()`** will fill in the **`FieldPosition`** with the starting and ending offsets of the month in the output text
    - Can't be used to find more than one field in a single call to **`format()`**
  - **`ParsePosition`** is used to return some state information to the user after a call to a **`parse()`** method
    - Filled in with offset of first character in the string not consumed by the parse operation
    - If an error occurred, also filled in with location of error

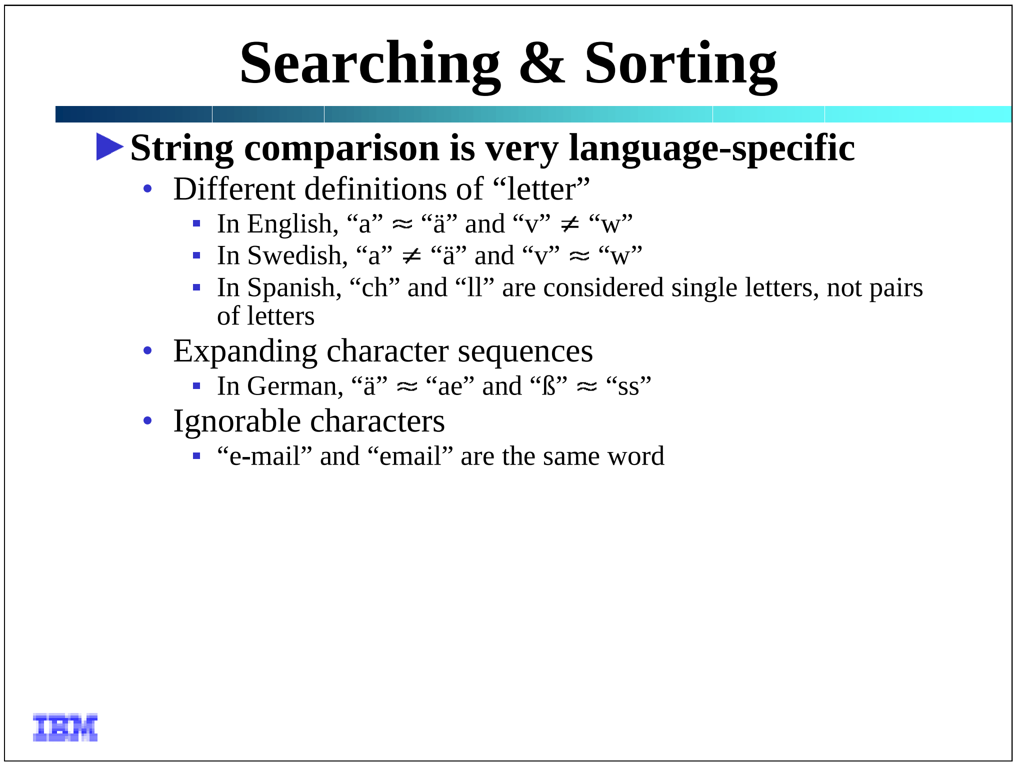
The formatting framework also defines two auxiliary classes.

The client can use a FieldPosition object in conjunction with a formater's format() methods to locate a particular "field" in the formatted result (the "month" field in a date format, or the integral part of a number).

The client can also use a ParsePosition object in conjunction with a formatter's parse() methods to specify the starting parse location in a string and keep track of how many characters from the string were consumed by the parse. If there's a parse error, the ParsePosition also shows where in the string the error occurred.



Searching and Sorting Text



Searching & Sorting

- **String comparison is very language-specific**
  - Different definitions of “letter”
    - In English, “a” ≈ “ä” and “v” ≠ “w”
    - In Swedish, “a” ≠ “ä” and “v” ≈ “w”
    - In Spanish, “ch” and “ll” are considered single letters, not pairs of letters
  - Expanding character sequences
    - In German, “ä” ≈ “ae” and “ß” ≈ “ss”
  - Ignorable characters
    - “e-mail” and “email” are the same word

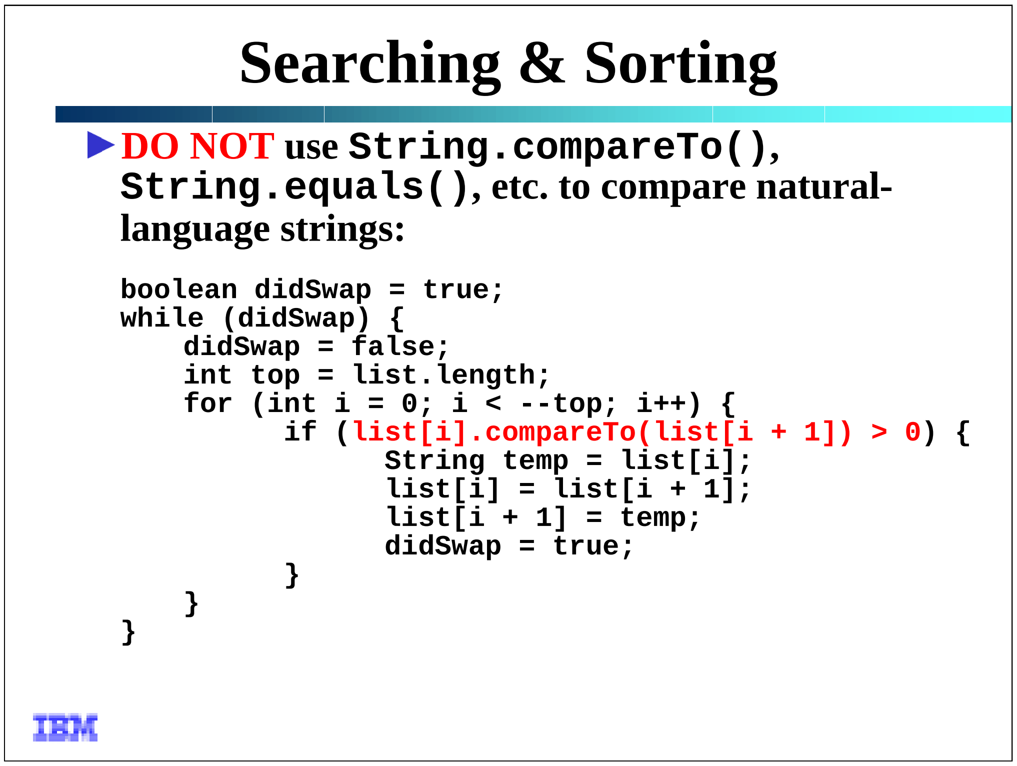
Just as it’s important to watch for hidden assumptions about language when displaying text on the screen, it’s important to watch for hidden assumptions when analyzing or manipulating text internally. The most important analysis operations done on text are searching and sorting, which both rely on string comparison and have highly language-dependent behavior.

For example, in English, a-umlaut is just an a with an umlaut added to it, while v and w are completely different letters. In Swedish, on the other hand, a-umlaut is a completely different letter from an unadorned a, and actually sorts after z. v and w, on the other hand, are variant forms of the same letter in Swedish.

Some languages treat sequences of characters as though they were one character: for instance “ch” and “ll” are considered single letters, not pairs of letters, in Spanish.

Some languages treat some single letters as though they were sequences of characters: for instance, a-umlaut in German is equivalent to “ae”, and the sharp S is equivalent to “ss”.

Most languages also have the concept of characters that are “ignorable” for searching or sorting purposes: for instance, in English, “email” is the same word whether or not it’s spelled with a hyphen.

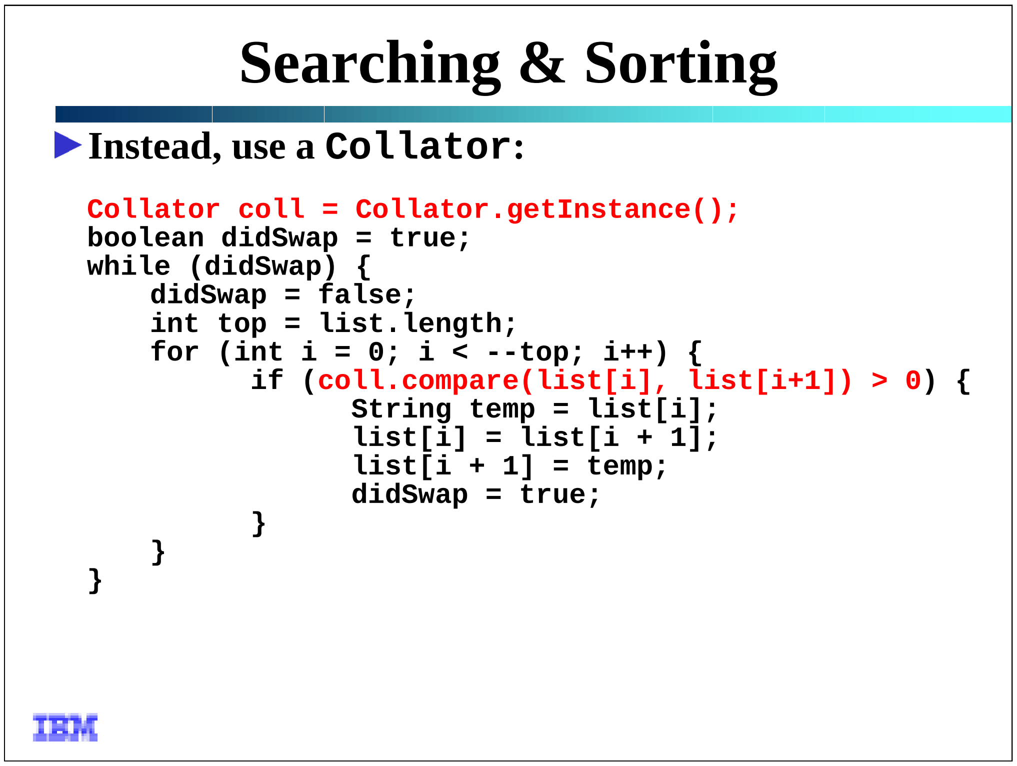


Searching & Sorting

- **DO NOT use `String.compareTo()`, `String.equals()`, etc. to compare natural-language strings:**

```
boolean didSwap = true;
while (didSwap) {
    didSwap = false;
    int top = list.length;
    for (int i = 0; i < --top; i++) {
          if (list[i].compareTo(list[i + 1]) > 0) {
                String temp = list[i];
                list[i] = list[i + 1];
                list[i + 1] = temp;
                didSwap = true;
          }
    }
}
```

The bottom line, therefore, is that you shouldn’t use String.compareTo(), String.equals(), or similar functions to compare natural-language text. These functions can be fine for things like internal IDs, but not for natural-language text. The problem is that these functions perform a bitwise lexicographic compare, which is not language-sensitive (and, in fact, doesn’t conform to any language’s sort order with Unicode values).

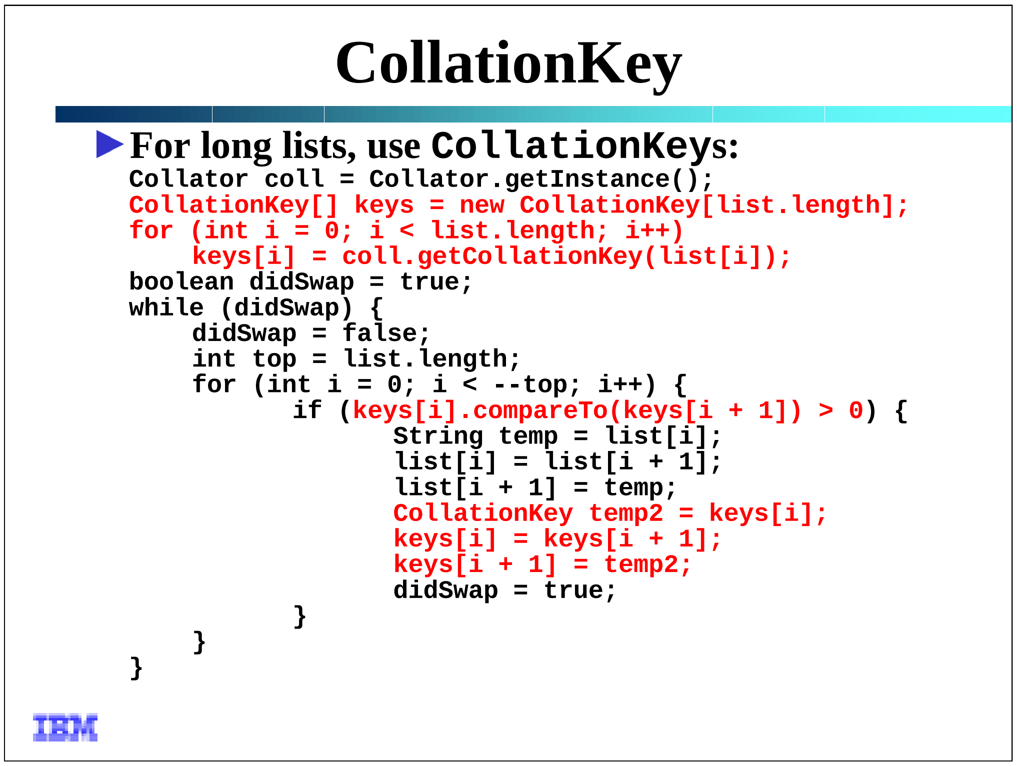


Searching & Sorting

- Instead, use a `Collator`:

```
Collator coll = Collator.getInstance();
boolean didSwap = true;
while (didSwap) {
    didSwap = false;
    int top = list.length;
    for (int i = 0; i < --top; i++) {
          if (coll.compare(list[i], list[i+1]) > 0) {
                String temp = list[i];
                list[i] = list[i + 1];
                list[i + 1] = temp;
                didSwap = true;
          }
    }
}
```

Instead, Java provides a class called Collator that knows how to compare strings in a language-sensitive way. When comparing natural-language strings, create a Collator object and use its compare() method to compare strings.



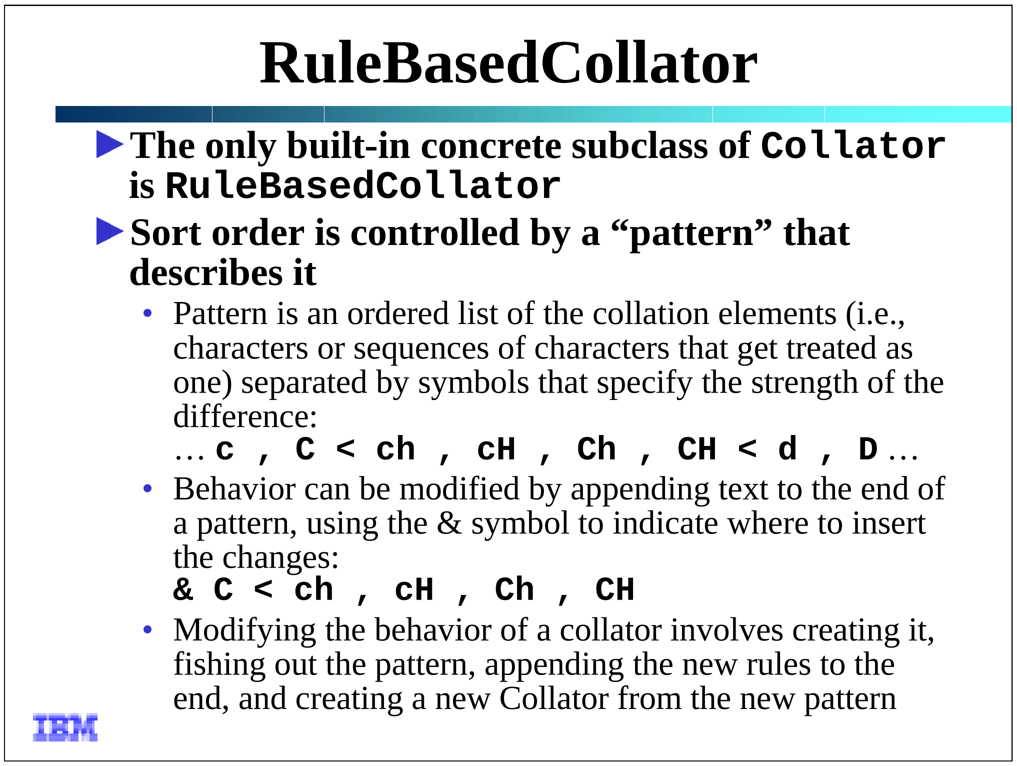
CollationKey

- For long lists, use `CollationKeys`:

```
Collator coll = Collator.getInstance();
CollationKey[] keys = new CollationKey[list.length];
for (int i = 0; i < list.length; i++)
    keys[i] = coll.getCollationKey(list[i]);
boolean didSwap = true;
while (didSwap) {
    didSwap = false;
    int top = list.length;
    for (int i = 0; i < --top; i++) {
           if (keys[i].compareTo(keys[i + 1]) > 0) {
                  String temp = list[i];
                  list[i] = list[i + 1];
                  list[i + 1] = temp;
                  CollationKey temp2 = keys[i];
                  keys[i] = keys[i + 1];
                  keys[i + 1] = temp2;
                  didSwap = true;
           }
    }
}
```

The previous two examples showed a simple snippet of code doing a bubble sort on a list of strings. As you might imagine, it's significantly slower to do a language-sensitive comparison than it is to do a bitwise comparison. Internally, each compare operation partially translates the strings it's comparing into sort keys--sequences of integers that can be compared with a bitwise compare. You can speed up a sort operation (or any other operation that compares the same strings repeatedly) by creating sort keys for all of your strings before doing the sort.

Java has a class called CollationKey that represents a sort key. It's a sequence of integers based on a String that can be bitwise compared with another CollationKey created by the same Collation and will yield the same result for the same two strings as calling Collator.compare() itself. So to do a faster sort on a long list, you'd create a temporary array of CollationKeys using Collator.getCollationKey() and then do the sort by doing bitwise comparison on the keys. For long lists, the time savings in the sort loop swamps the extra time spent building the key list.

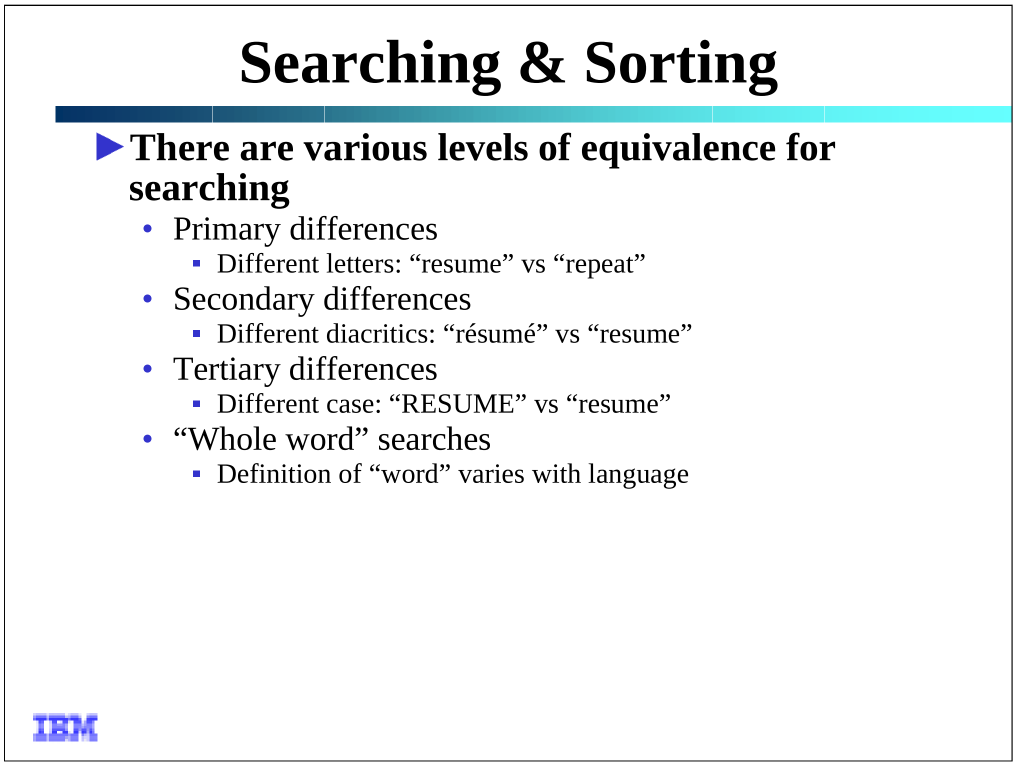


RuleBasedCollator

- **The only built-in concrete subclass of `Collator` is `RuleBasedCollator`**
- **Sort order is controlled by a “pattern” that describes it**
  - Pattern is an ordered list of the collation elements (i.e., characters or sequences of characters that get treated as one) separated by symbols that specify the strength of the difference:
    … `c , C < ch , cH , Ch , CH < d , D` …
  - Behavior can be modified by appending text to the end of a pattern, using the & symbol to indicate where to insert the changes:
    `& C < ch , cH , Ch , CH`
  - Modifying the behavior of a collator involves creating it, fishing out the pattern, appending the new rules to the end, and creating a new Collator from the new pattern

The implementation class for Collator is called RuleBasedCollator, and it can be used directly when the programmer wants to specify a particular sort order. Again, a pattern string specifies the collator’s behavior.

The pattern string just consists of the various tokens (either single characters or groups of characters to treat as a single character) separated by characters that specify the level of difference (see next slide) between them. You can also modify an existing set of rule by appending rules at the end that start with &. The & symbol allows you to insert new rules at arbitrary positions earlier in the rules.



Searching & Sorting

- **There are various levels of equivalence for searching**
  - Primary differences
    - Different letters: “resume” vs “repeat”
  - Secondary differences
    - Different diacritics: “résumé” vs “resume”
  - Tertiary differences
    - Different case: “RESUME” vs “resume”
  - “Whole word” searches
    - Definition of “word” varies with language

When you’re doing a search or comparing two strings for equality, you also care about the degree of equivalence; for example, you may or may not want to take case differences into account.

Java’s Collator class defines three levels of equivalence:

- Two strings have a “primary difference” if somewhere they have different “letters” (according to the language) in corresponding positions

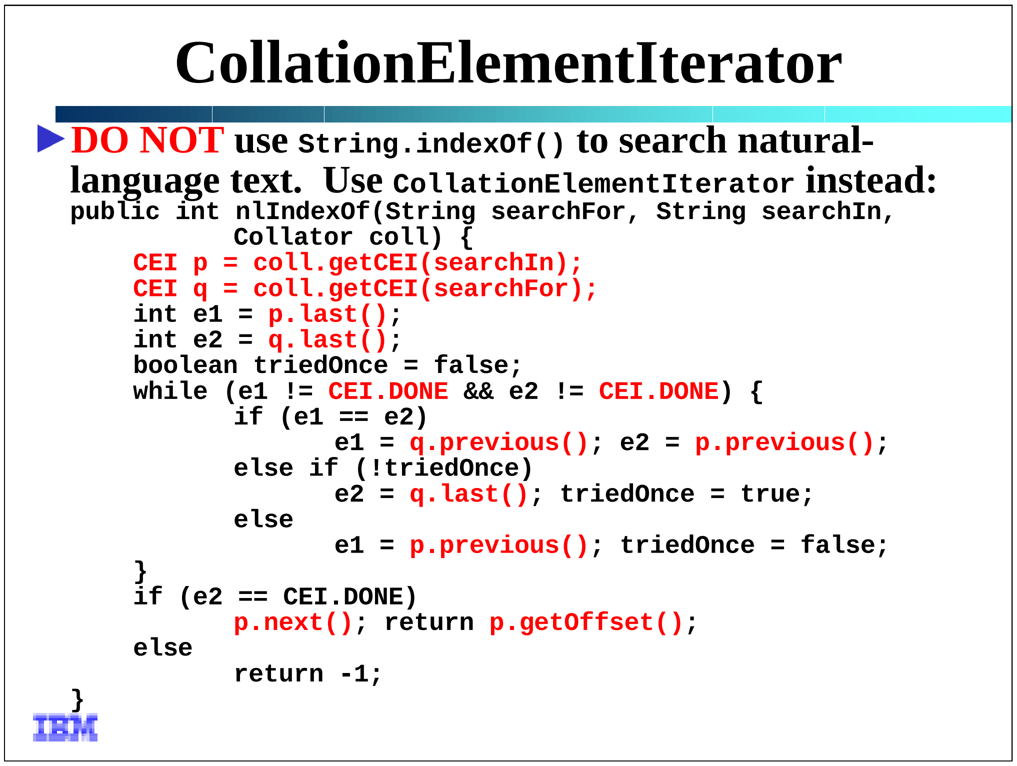
- Two strings have a “secondary difference” if they don’t have a primary difference, but do have two corresponding letters with a diacritic or variant-form difference.

- Two strings have a “tertiary difference” if they don’t have a primary or secondary difference, but two corresponding letters have different case.

There’s also a fourth level of difference, “identity difference,” which is when there are no tertiary differences, but the strings still are different in terms of the actual hex codes. This usually happens when you have two otherwise equal strings that contain characters from outside the language-- those characters sort after everything else and are sorted relative to each other based on their hex value (this helps ensure there’s always a well-defined ordering in a list).

When you’re searching, you may also only be interested in search hits that are “whole words,” i.e., whose ends both fall on word boundaries. But the definition of “word” also varies between languages.

There are many types of searches that require other more complicated types of equivalence, but there isn’t much built-in support for these fancier processes in Java.



CollationElementIterator

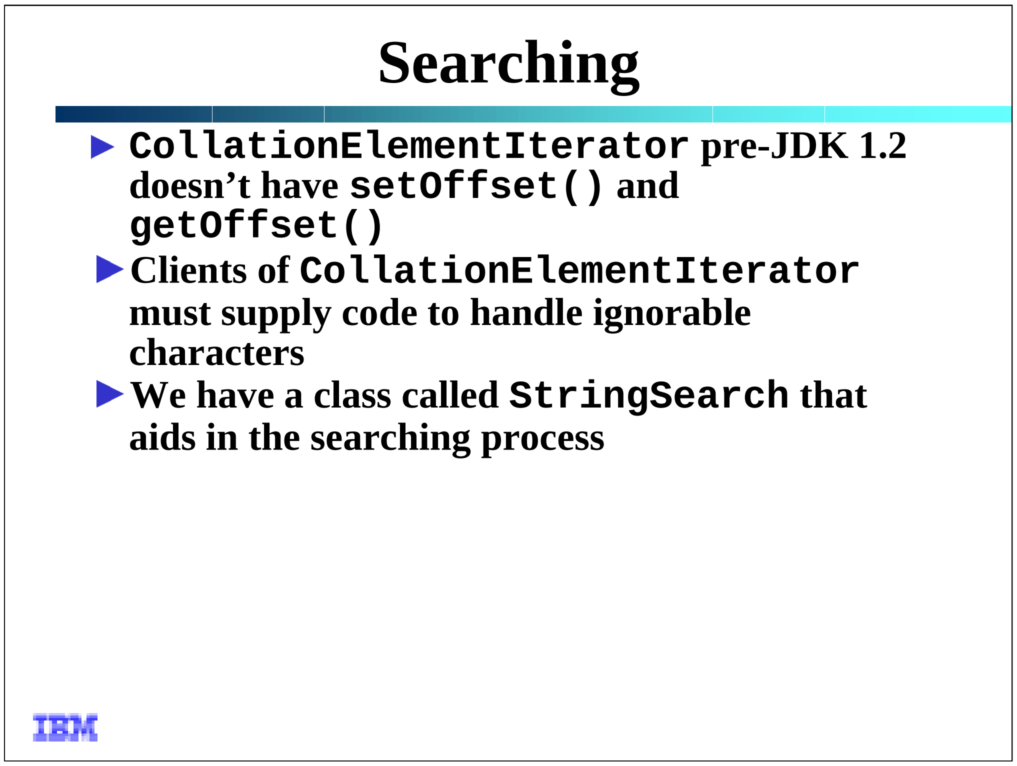
▶ **DO NOT use `String.indexOf()` to search natural-language text. Use `CollationElementIterator` instead:**

```
public int nlIndexOf(String searchFor, String searchIn,
           Collator coll) {
    CEI p = coll.getCEI(searchIn);
    CEI q = coll.getCEI(searchFor);
    int e1 = p.last();
    int e2 = q.last();
    boolean triedOnce = false;
    while (e1 != CEI.DONE && e2 != CEI.DONE) {
           if (e1 == e2)
                  e1 = q.previous(); e2 = p.previous();
           else if (!triedOnce)
                  e2 = q.last(); triedOnce = true;
           else
                  e1 = p.previous(); triedOnce = false;
    }
    if (e2 == CEI.DONE)
           p.next(); return p.getOffset();
    else
           return -1;
}
```

So again, you don't want to use String.indexOf() or any of its brothers when you're searching natural-language text (although again it may be just fine for internal IDs and other things like that).

Instead, Java provides a class called CollationElementIterator that can be used to perform language-sensitive searching. A CollationKey is a sequence of integers. These individual integers are called collation elements, and there isn't a one-to-one mapping of characters to collation elements. To do a natural-language search, you have to match collation elements, not characters. CollationElementIterator provides you a way to obtain collation elements one at a time and a way to map back from a collation-element position to an actual character position in the original string.

This example (which I've had to squoosh terribly to get on one slide) shows one simple way of doing a search using a CollationElementIterator. Instead of pulling characters from the strings, you use the iterator to get collation elements (which are ints) one at a time. Then you use getOffset() at the end to tell where the hit is. This algorithm does it backwards so that we're sitting at the beginning of the hit when we drop out of the loop. Unfortunately, that means this function really matches lastIndexOf(). Doing indexOf() correctly is more complicated. This example also ignores strength differences and ignorable characters, both of which complicate things somewhat.



Searching

- **`CollationElementIterator` pre-JDK 1.2 doesn’t have `setOffset()` and `getOffset()`**
- **Clients of `CollationElementIterator` must supply code to handle ignorable characters**
- **We have a class called `StringSearch` that aids in the searching process**

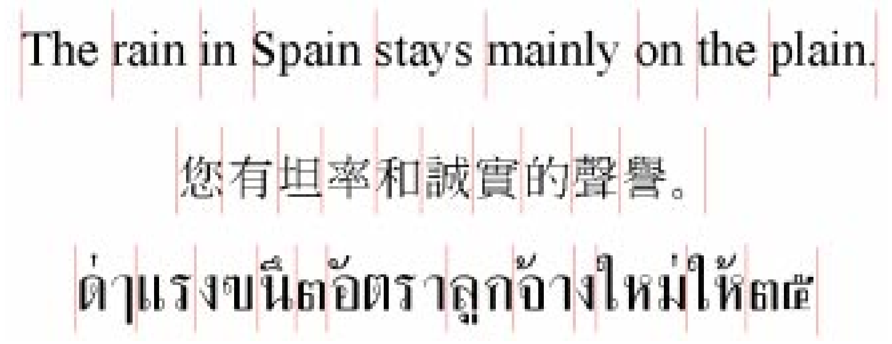
A few other random points: CollationElementIterator doesn’t have getOffset() and setOffset() in pre-1.2 Java implementations. Searching can be done without them, but only with difficulty and slowly.

CollationElementIterator also leaves handling of some of the details, such as ignorable characters, to you. The example on the previuos slide omits this because I ran out of room.

We have a convenience class called StringSearch that handles all the extra bookkeeping for you and also implements a fast-search algorithm. It’s also available via AlphaWorks.

Locating Word and Character Boundaries

Locating text boundaries


The rain in Spain stays mainly on the plain.

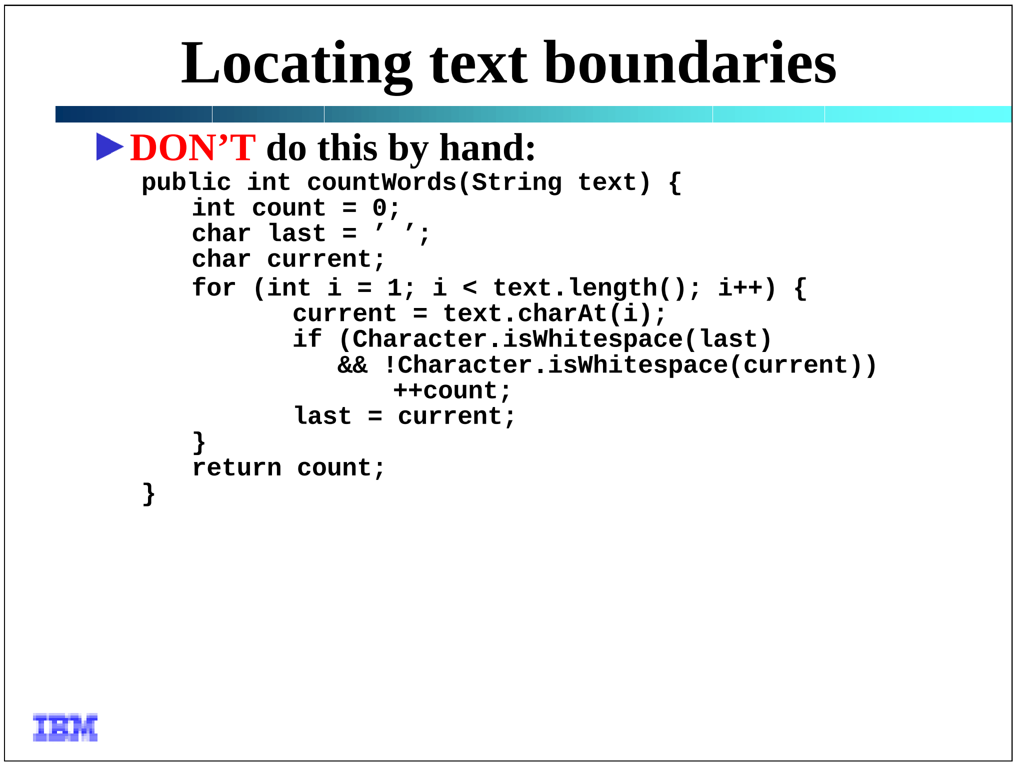
您有坦率和誠實的聲譽。

ค่าแรงขั้นต่ำอัตราลูกจ้างใหม่ให้๓๕

IBM

Locating boundaries between words isn't as simple as it seems. The definition of "word" varies from language to language. In fact, the definition of "word" depends on what you're doing. In this example, we're looking for positions where it'd be legal to wrap text onto the next line. In English you can (generally, but not always), break the line at the boundary between a run of whitespace characters and a run of non-whitespace characters. But in Thai, spaces aren't used between words, o you have to do a lot more work to determine the word boundaries.

In Chinese, there also aren't spaces between words, but lines can be broken almost anywhere, not just on actual word boundaries. The only restriction is that certain punctuation marks must be kept with the character before or after them.



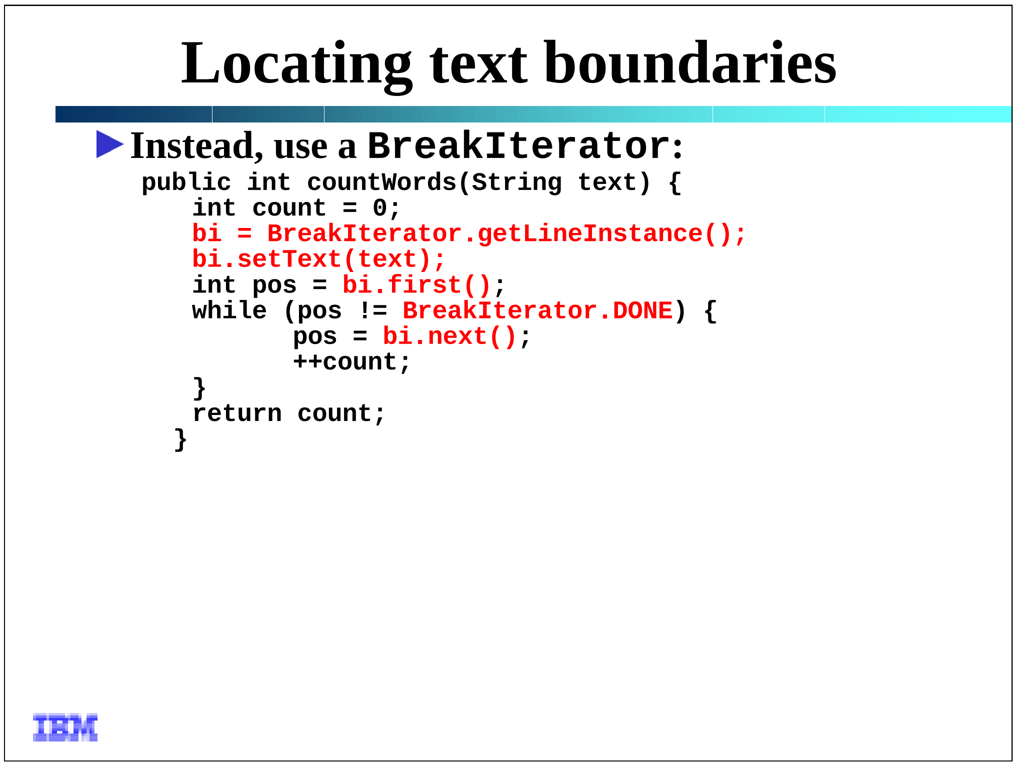
Locating text boundaries

▶ **DON'T do this by hand:**

```
public int countWords(String text) {
   int count = 0;
   char last = ' ';
   char current;
   for (int i = 1; i < text.length(); i++) {
          current = text.charAt(i);
          if (Character.isWhitespace(last)
             && !Character.isWhitespace(current))
                 ++count;
          last = current;
   }
   return count;
}
```

IBM

Clearly, then, if you try to locate word boundaries using a simple algorithm such as this, you'll break down in some languages and occasionally get wrong results in all languages.

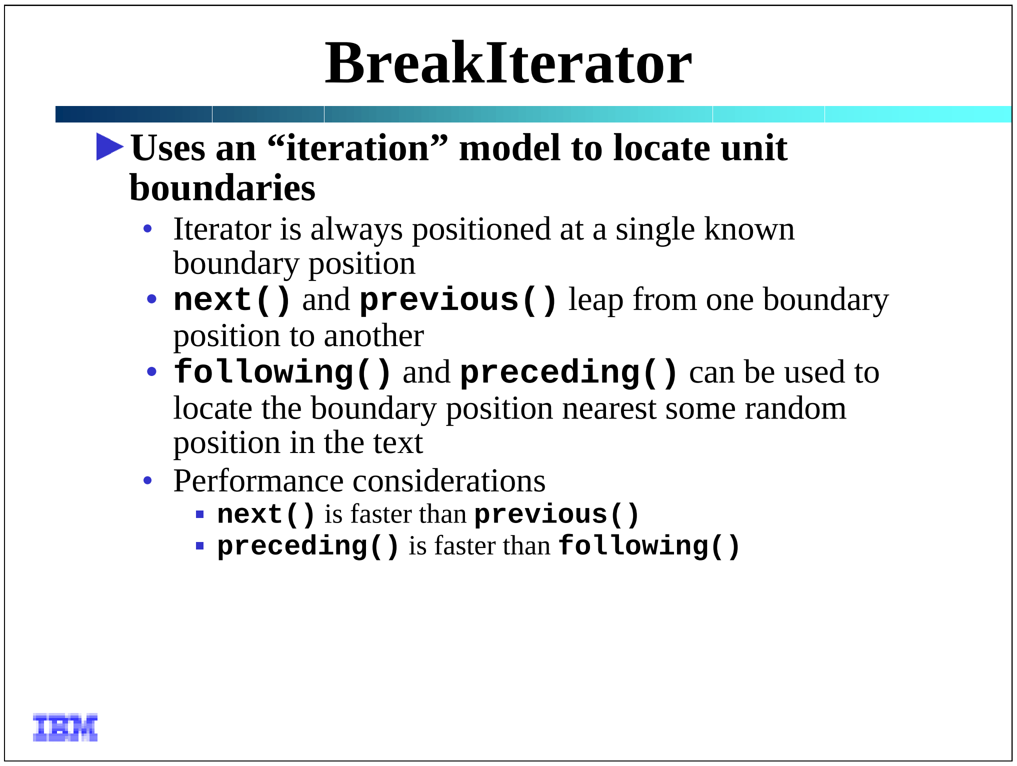


Locating text boundaries

- **Instead, use a `BreakIterator`:**

```
public int countWords(String text) {
   int count = 0;
   bi = BreakIterator.getLineInstance();
   bi.setText(text);
   int pos = bi.first();
   while (pos != BreakIterator.DONE) {
          pos = bi.next();
          ++count;
   }
   return count;
  }
```

Java provides a class called BreakIterator to take care of this problem. This how the code from the preceding slide looks if you use a BreakIterator. Note that it's totally different, but similar to code that iterates across a collection using an Enumeration object.

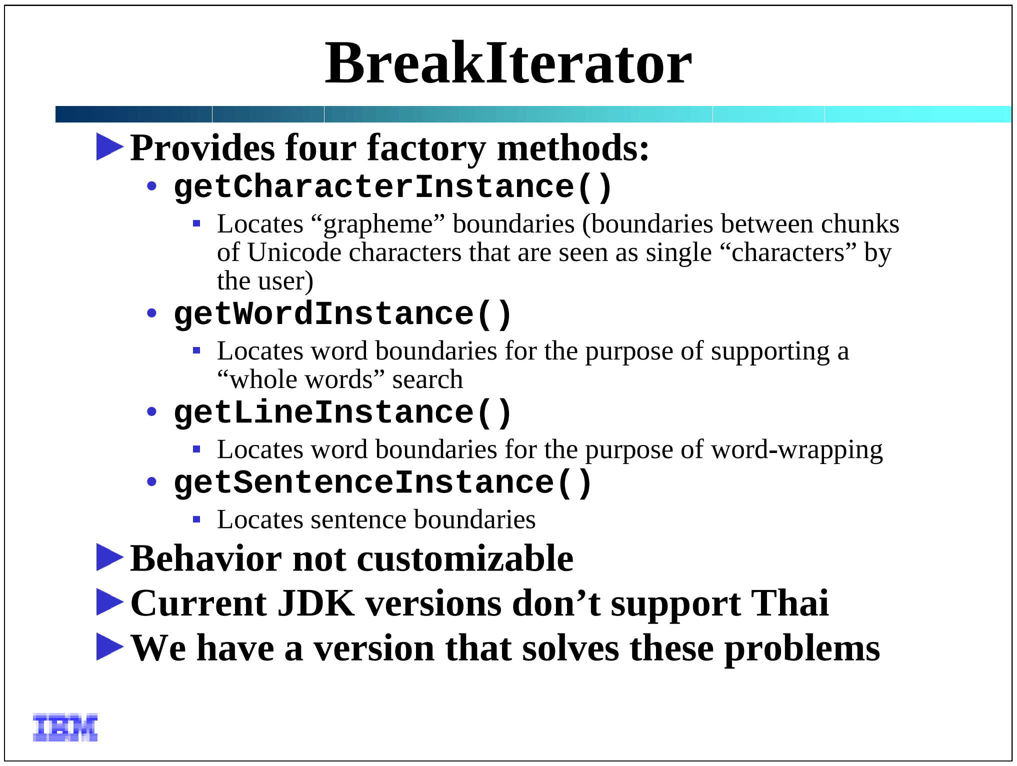


BreakIterator

- **Uses an "iteration" model to locate unit boundaries**
  - Iterator is always positioned at a single known boundary position
  - **`next()`** and **`previous()`** leap from one boundary position to another
  - **`following()`** and **`preceding()`** can be used to locate the boundary position nearest some random position in the text
  - Performance considerations
    - **`next()`** is faster than **`previous()`**
    - **`preceding()`** is faster than **`following()`**

That iteration idiom is the one that BreakIterator uses to return boundary positions to the client. The BreakIterator just jumps from boundary to boundary, returning their positions in order. You can move through them forward or backward (although forward is faster).

You can also use preceding() and following() to locate the nearest boundary to some arbitrary position in the text (preceding() is faster than following()).



BreakIterator

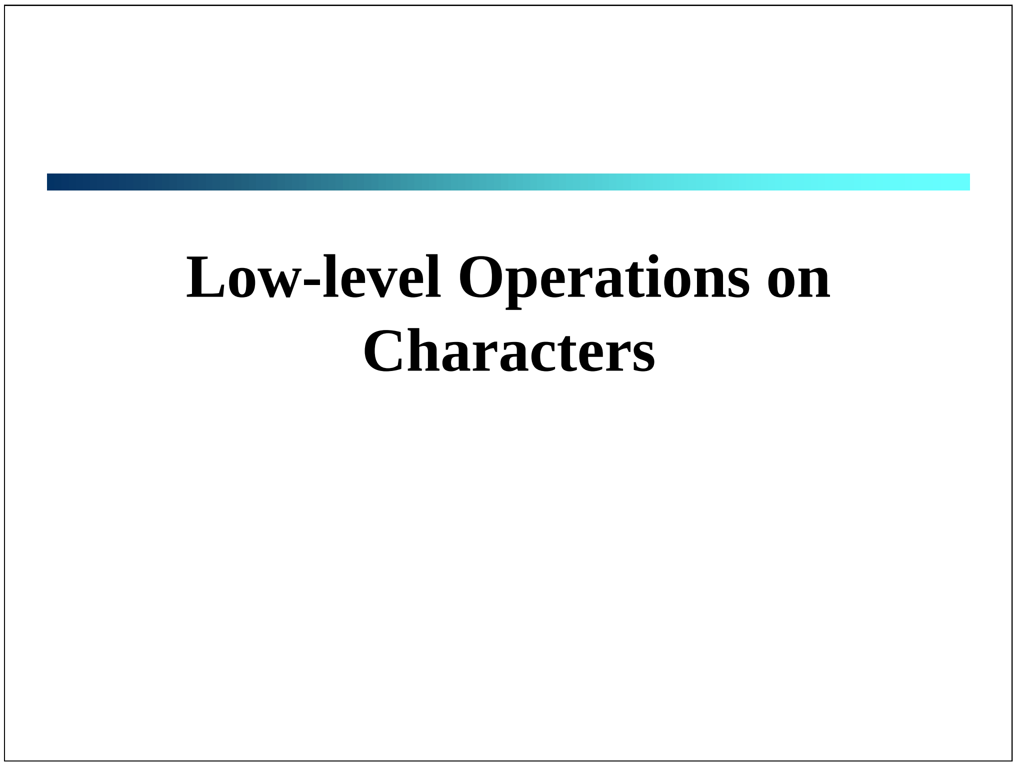
- **Provides four factory methods:**
  - `getCharacterInstance()`
    - Locates “grapheme” boundaries (boundaries between chunks of Unicode characters that are seen as single “characters” by the user)
  - `getWordInstance()`
    - Locates word boundaries for the purpose of supporting a “whole words” search
  - `getLineInstance()`
    - Locates word boundaries for the purpose of word-wrapping
  - `getSentenceInstance()`
    - Locates sentence boundaries
- **Behavior not customizable**
- **Current JDK versions don’t support Thai**
- **We have a version that solves these problems**

IBM

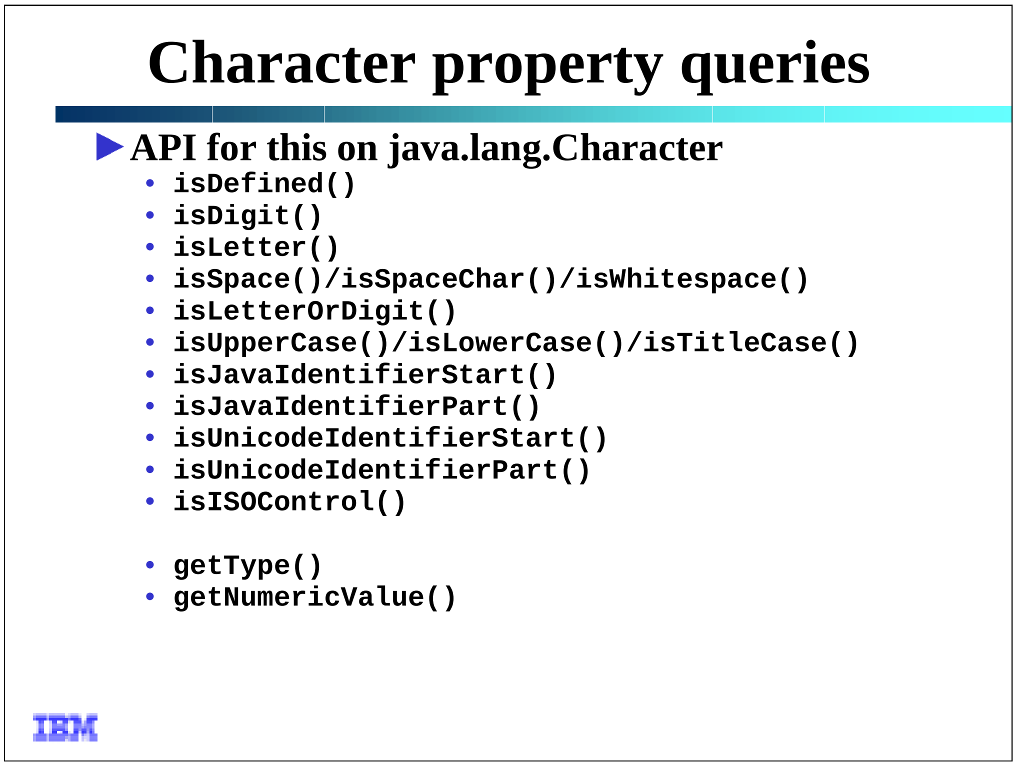
BreakIterator isn’t restricted to just finding word boundaries. It provides factory methods for locating boundaries between four different types of units (including two different types of word boundaries). Others are possible.

Currently, BreakIterator’s behavior is not customizable (although the canned behavior is correct for almost all languages), and Thai (which requires a more sophisticated algorithm) isn’t supported.

IBM’s JDK 1.2 release includes a version of BreakIterator that is customizable and supports Thai, and this is also available now on AlphaWorks. Thai support is also planned for future JDKs from Sun.



Low-level Operations on Characters



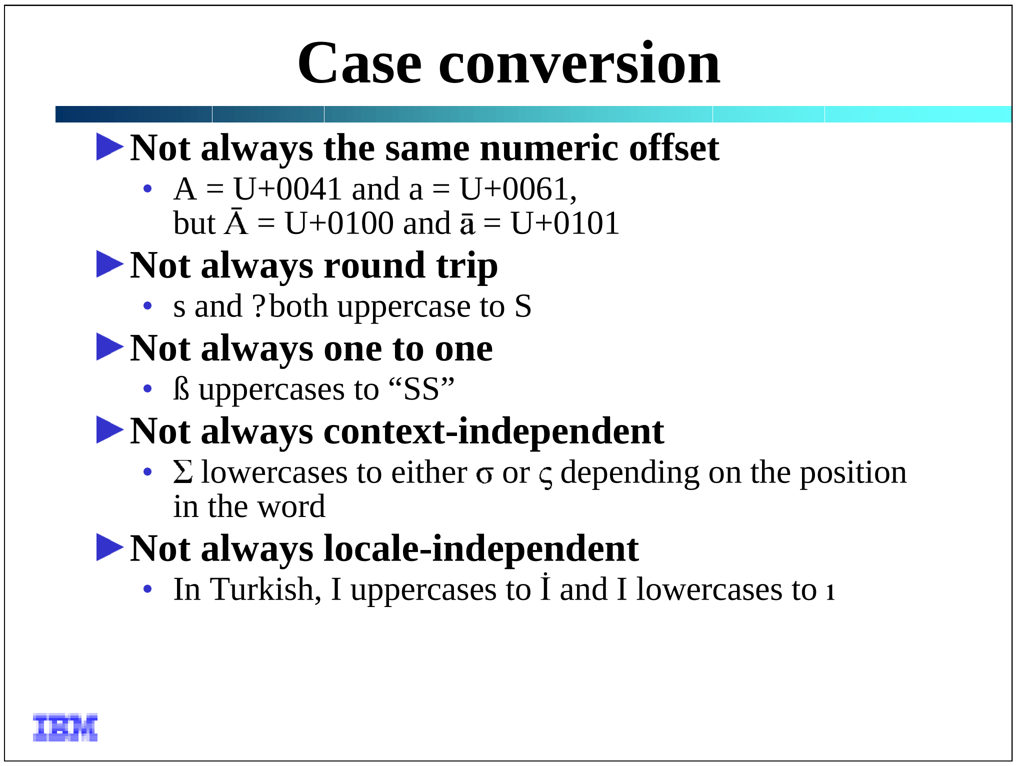
Character property queries

- API for this on java.lang.Character
  - `isDefined()`
  - `isDigit()`
  - `isLetter()`
  - `isSpace()/isSpaceChar()/isWhitespace()`
  - `isLetterOrDigit()`
  - `isUpperCase()/isLowerCase()/isTitleCase()`
  - `isJavaIdentifierStart()`
  - `isJavaIdentifierPart()`
  - `isUnicodeIdentifierStart()`
  - `isUnicodeIdentifierPart()`
  - `isISOControl()`

  - `getType()`
  - `getNumericValue()`

IBM

The java.lang.Character class provides a whole variety of functions to query various properties on a character. getType() returns a code representing one of the types defined in the Unicode Character Database. getNumericValue() returns the numeric value of a character according to the Unicode Character Database.

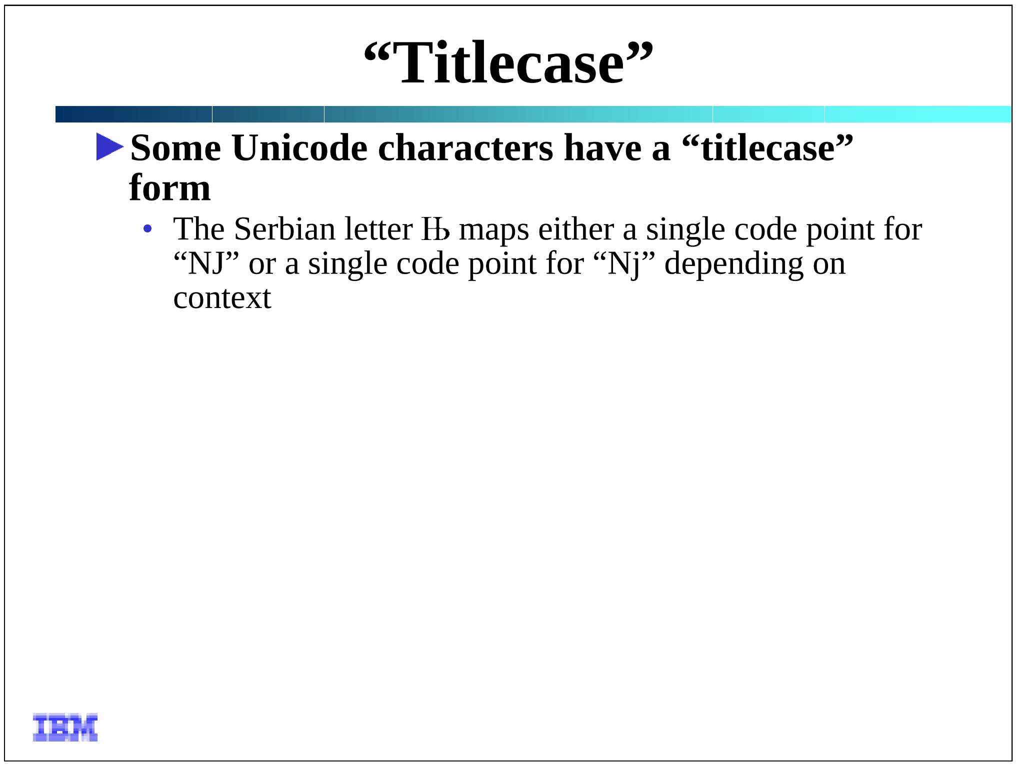


Case conversion

- **Not always the same numeric offset**
  - A = U+0041 and a = U+0061,
    but Ā = U+0100 and ā = U+0101
- **Not always round trip**
  - s and ?both uppercase to S
- **Not always one to one**
  - ß uppercases to “SS”
- **Not always context-independent**
  - Σ lowercases to either σ or ς depending on the position in the word
- **Not always locale-independent**
  - In Turkish, I uppercases to İ and I lowercases to ı

IBM

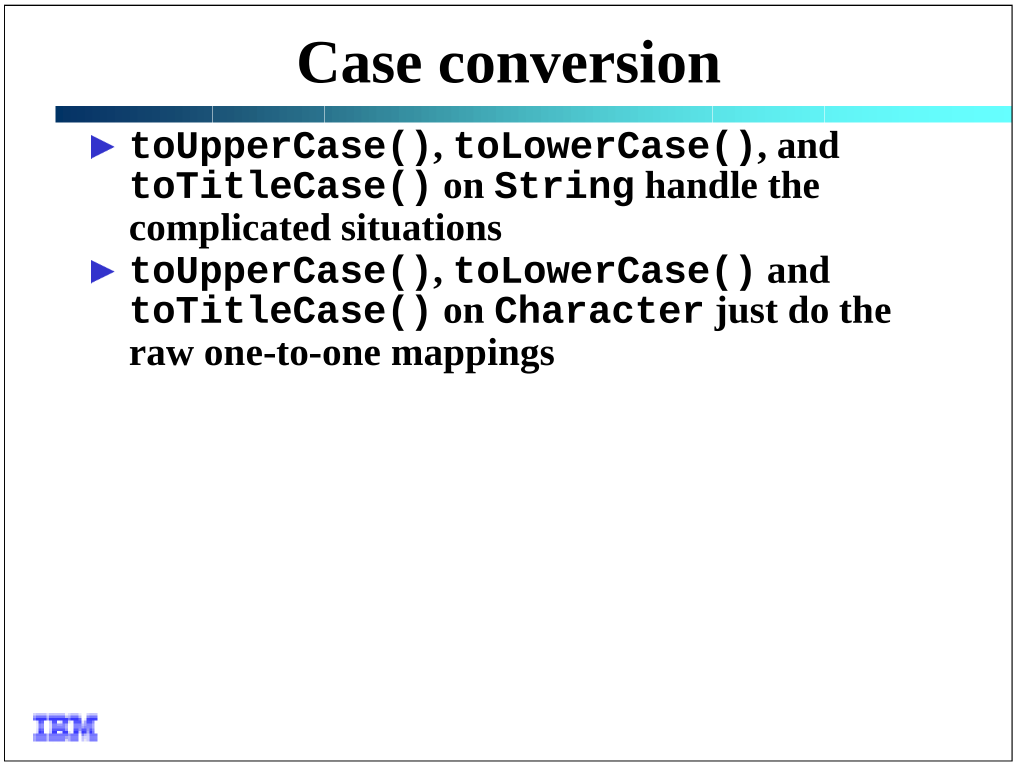
Converting a character from lowercase to uppercase and vice versa also isn’t straightforward. Case pairs aren’t all positioned in the same way relative to each other (again, because of compatibility issues with other encodings). Some times there are mappings between a character in one case and multiple counterparts in the other case. Sometimes a character will expand into two when converted to a different case. And sometimes the conversion is locale-dependent.



“Titlecase”

- **Some Unicode characters have a “titlecase” form**
  - The Serbian letter Њ maps either a single code point for “NJ” or a single code point for “Nj” depending on context

Then there’s “titlecase”, which is used with some single character codes that really represent two characters and represents the situation where the first “character” is uppercase and the second “character” is lowercase.

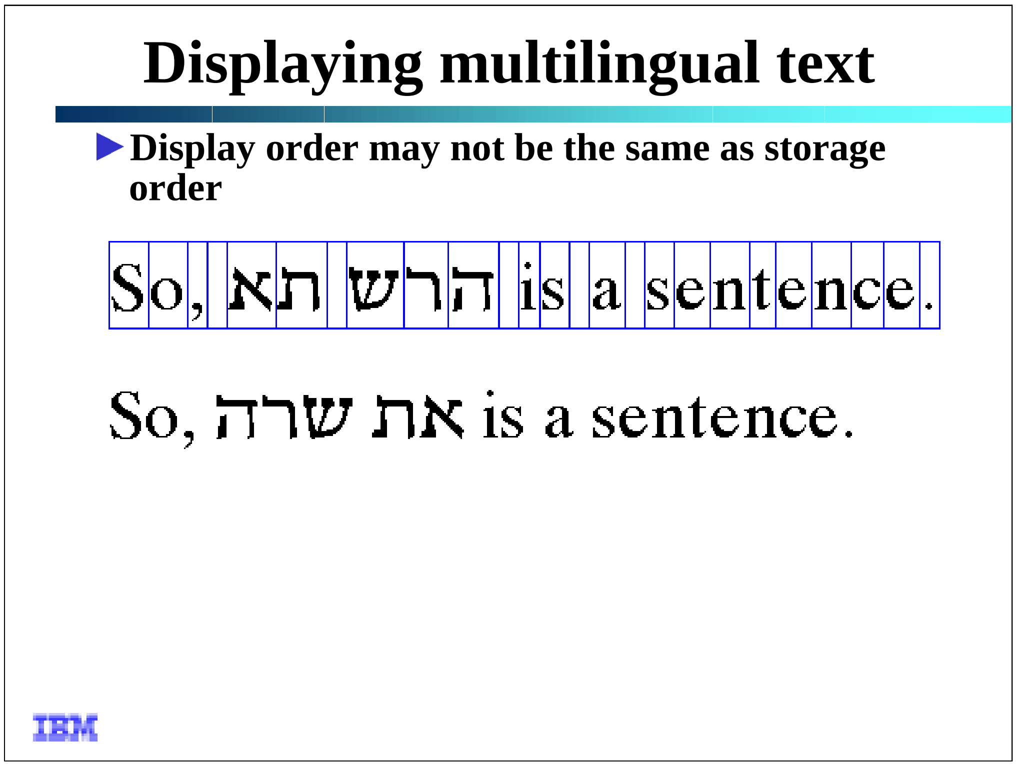


Case conversion

- **`toUpperCase()`, `toLowerCase()`, and `toTitleCase()` on `String` handle the complicated situations**
- **`toUpperCase()`, `toLowerCase()` and `toTitleCase()` on `Character` just do the raw one-to-one mappings**

There's API on both String and Character to perform case conversions. The String API takes into account all the complicated situations discussed before. The Character API only handles the simple one-to-one mappings.

Displaying and Editing Multilingual Text



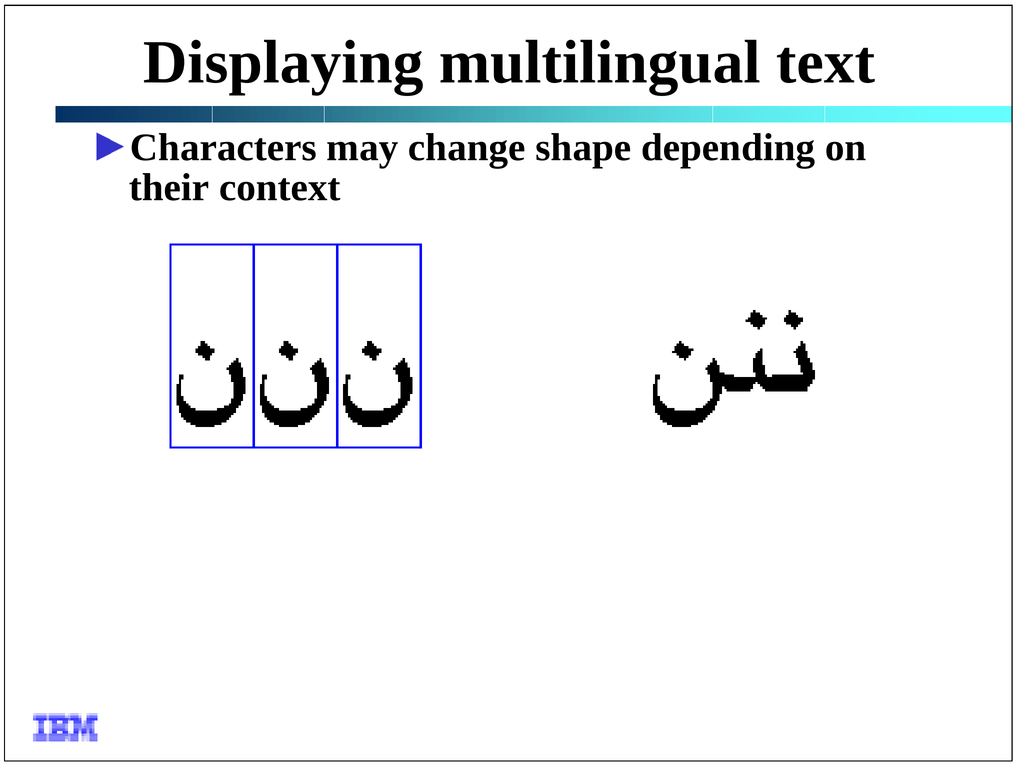
Displaying multilingual text

- **Display order may not be the same as storage order**

So, את שרה is a sentence.

So, את שרה is a sentence.

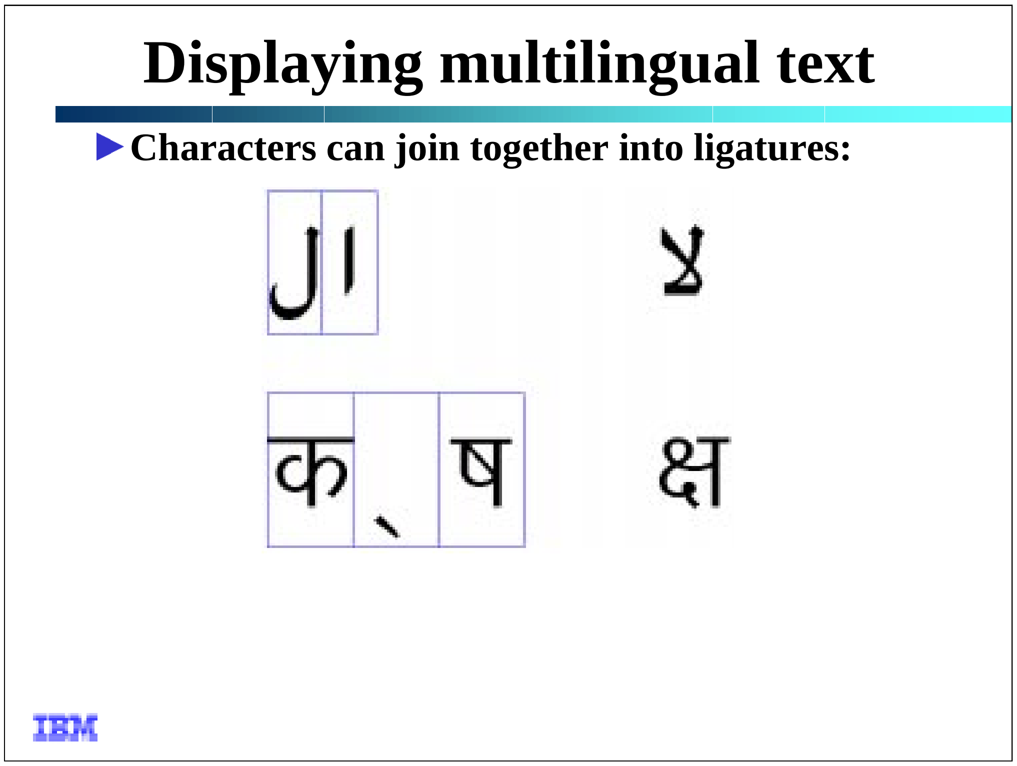
Displaying multilingual text can also be quite complicated. When mixing English and Hebrew text, which have opposite writing directions, on the same line, extra work must be done to keep the various character runs in the right visual positions relative to each other.



Displaying multilingual text

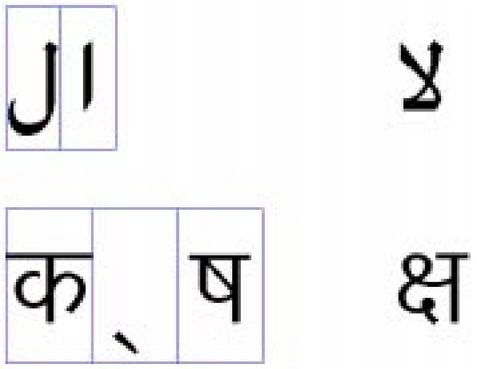
- Characters may change shape depending on their context

Sometimes, characters change their shape depending on the surrounding characters. Here, we show three of the same Arabic letter. When positioned next to one another, they assume three different shapes, none of which is the same as the form the letter takes when it stands alone.



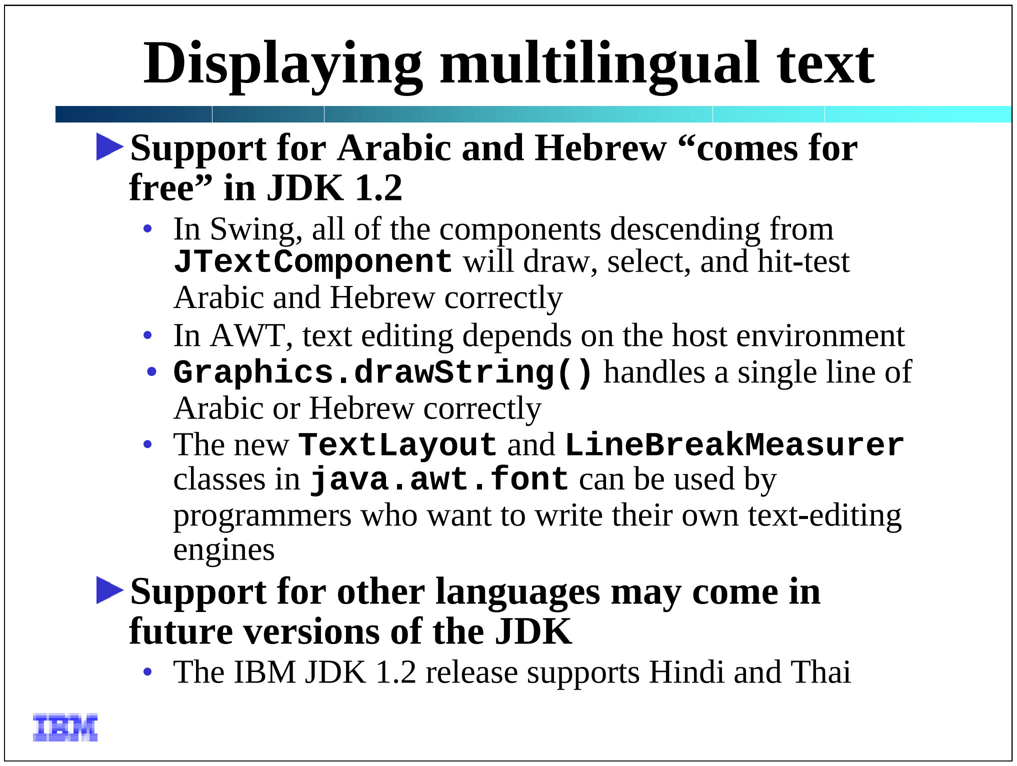
Displaying multilingual text

- Characters can join together into ligatures:



IBM

Sometimes, adjacent characters join together into a whole new shape called a “ligature.” This slide shows two examples (one in Arabic, one in Hindi) of how characters stored adjacently in memory appear on screen.



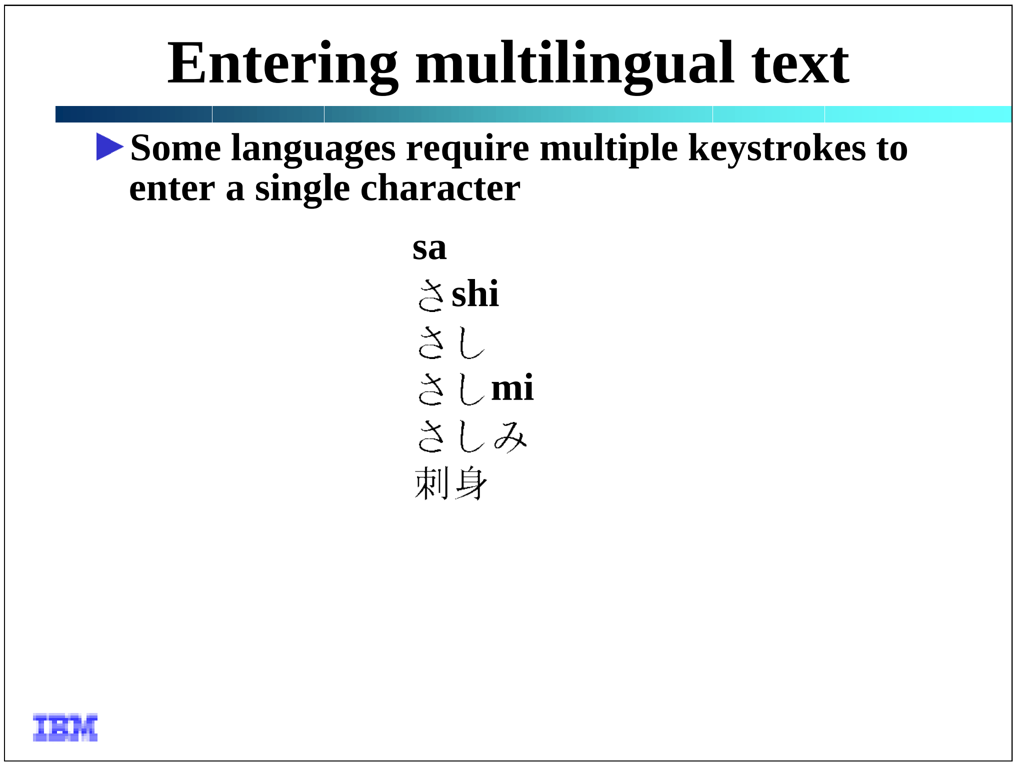
Displaying multilingual text

- **Support for Arabic and Hebrew "comes for free" in JDK 1.2**
  - In Swing, all of the components descending from **`JTextComponent`** will draw, select, and hit-test Arabic and Hebrew correctly
  - In AWT, text editing depends on the host environment
  - **`Graphics.drawString()`** handles a single line of Arabic or Hebrew correctly
  - The new **`TextLayout`** and **`LineBreakMeasurer`** classes in **`java.awt.font`** can be used by programmers who want to write their own text-editing engines
- **Support for other languages may come in future versions of the JDK**
  - The IBM JDK 1.2 release supports Hindi and Thai

The text-rendering engine in JDK 1.2 properly supports Arabic and Hebrew. All of the Swing text components have been updated, so support comes "for free." AWT, on the other hand, relies on the underlying host, so support for various languages doesn't depend on the JDK. The low-level Graphics.drawString() call also handles Arabic and Hebrew correctly.

The Java 2D framework now includes new classes, TextLayout and LineBreakMeasurer, which allow programmers writing their own text editors to draw and hit-test multilingual text correctly.

The IBM JDK 1.2 release will also support Hindi and Thai, and we are negotiating to get this into the next Sun JDK as well.



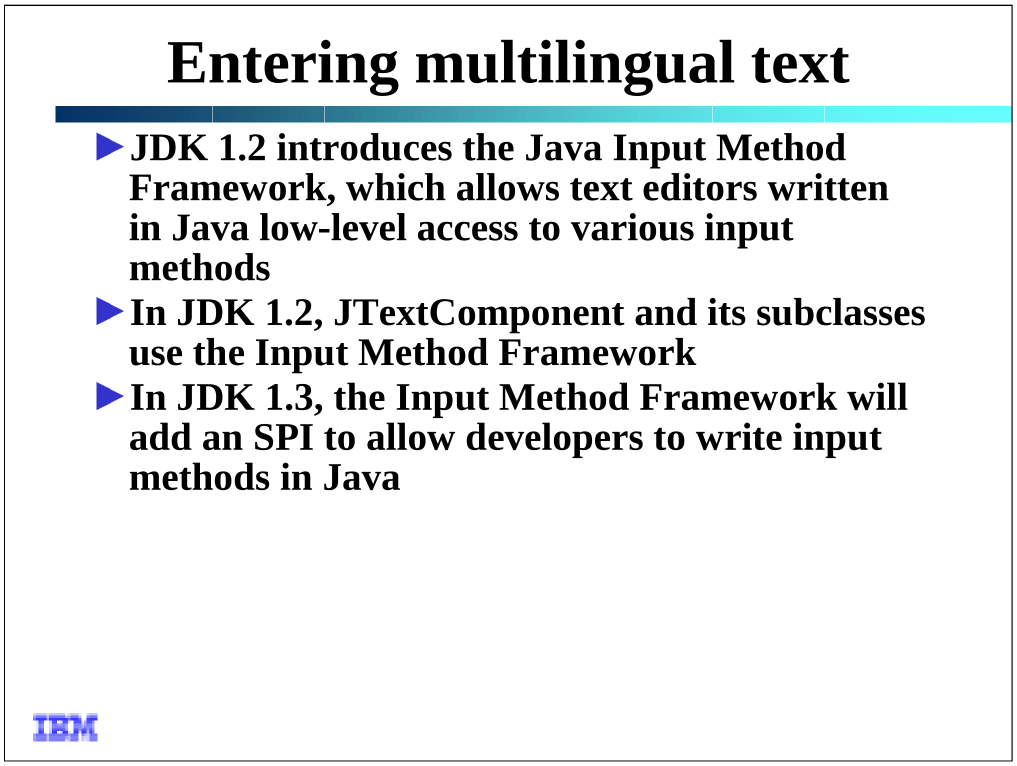
Entering multilingual text

- **Some languages require multiple keystrokes to enter a single character**

sa
さ**shi**
さし
さし**mi**
さしみ
刺身

IBM

Languages such as Japanese and Chinese with many different characters require special procedures (usually involving multiple keystrokes) to enter the text. These are called “input methods.”

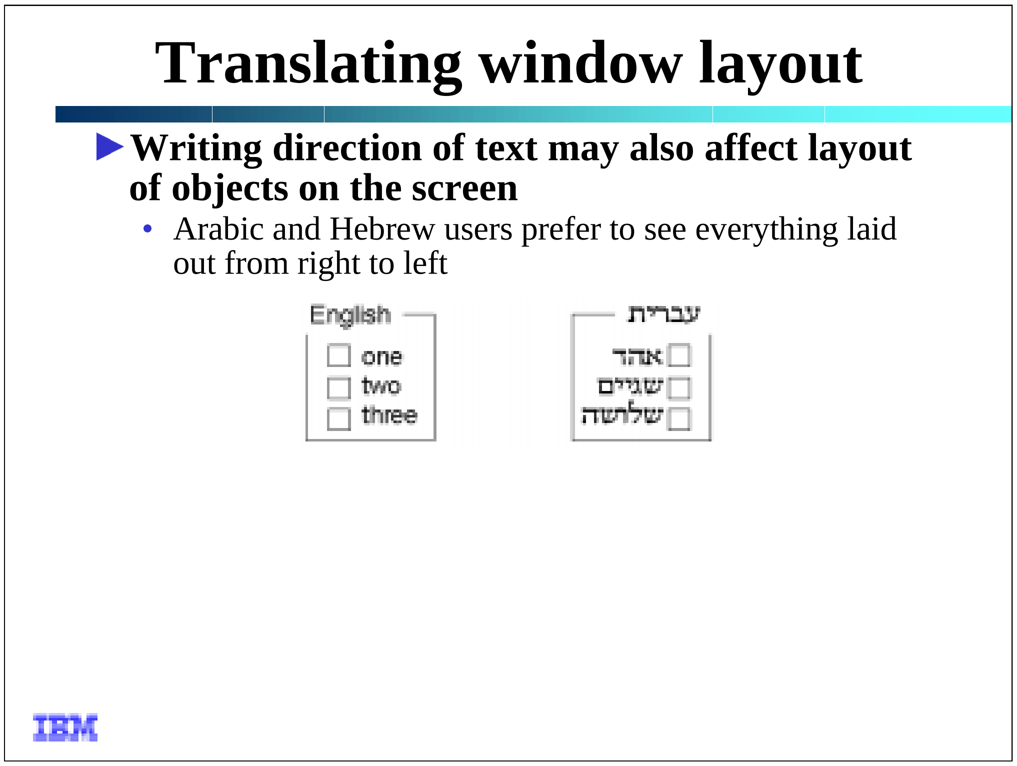


Entering multilingual text

- JDK 1.2 introduces the Java Input Method Framework, which allows text editors written in Java low-level access to various input methods
- In JDK 1.2, JTextComponent and its subclasses use the Input Method Framework
- In JDK 1.3, the Input Method Framework will add an SPI to allow developers to write input methods in Java

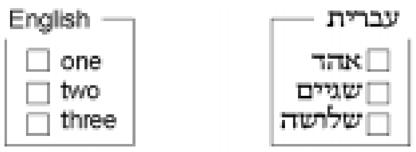
Pre-1.2 Java versions relied on the underlying host for input method support and didn’t provide a way for application programs to access the host’s input method engine. JDK 1.2 introduced the Java Input Method Framework, which does provide low-level access to the input method engine (allowing text editors written in Java to provide a better UI for interacting with input methods). In JDK 1.3, the Input Method Framework will add an SPI to allow programmers to write new input methods in Java.

Translating Window Layouts



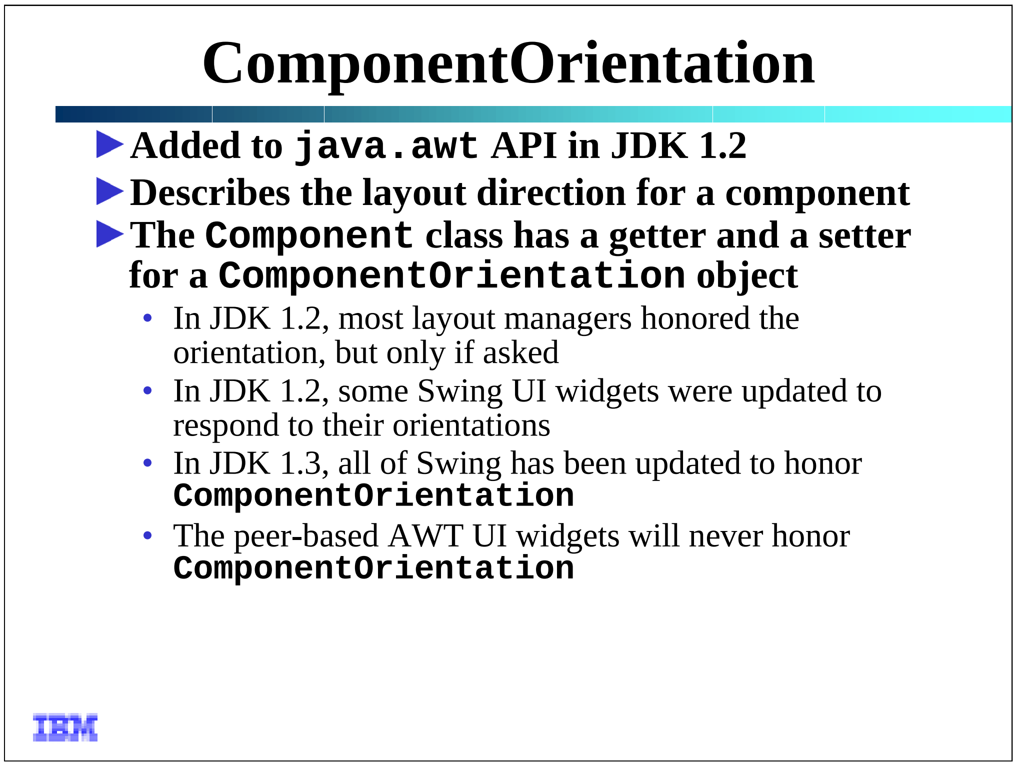
Translating window layout

- **Writing direction of text may also affect layout of objects on the screen**
  - Arabic and Hebrew users prefer to see everything laid out from right to left



IBM

Writing direction usually can affect window layout. Arabic and Hebrew speakers usually prefer for the whole UI to be the mirror image of its English layout: Individual UI widgets gets reversed relative to each other, and the UI widgets themselves also reverse: for example, the check box now appears to the right of its label.

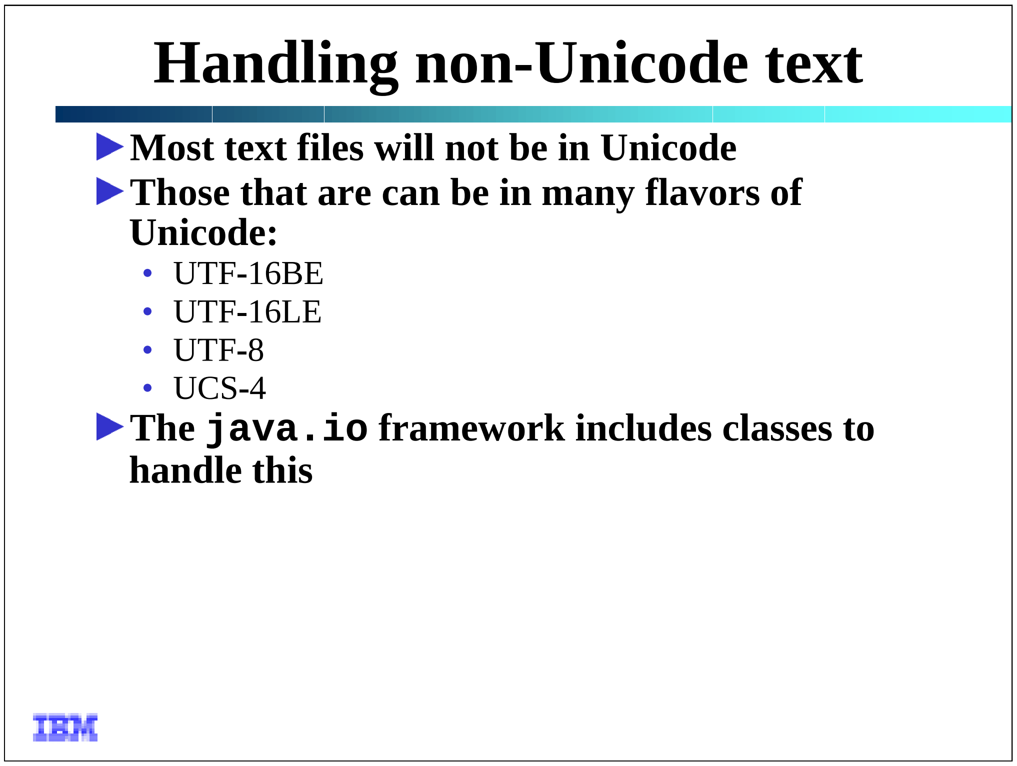


ComponentOrientation

- **Added to `java.awt` API in JDK 1.2**
- **Describes the layout direction for a component**
- **The `Component` class has a getter and a setter for a `ComponentOrientation` object**
  - In JDK 1.2, most layout managers honored the orientation, but only if asked
  - In JDK 1.2, some Swing UI widgets were updated to respond to their orientations
  - In JDK 1.3, all of Swing has been updated to honor **`ComponentOrientation`**
  - The peer-based AWT UI widgets will never honor **`ComponentOrientation`**

JDK 1.2 was updated to handle this by adding a new object called Component Orientation that specifies a UI widget's layout direction. Some layout managers and UI widgets were updated to respond to this setting in JDK 1.2, and the rest will be updated in JDK 1.3. The AWT UI widgets depend on the host environment for this support and are unaffected by changes in the JDK.

Handling Non-Unicode text

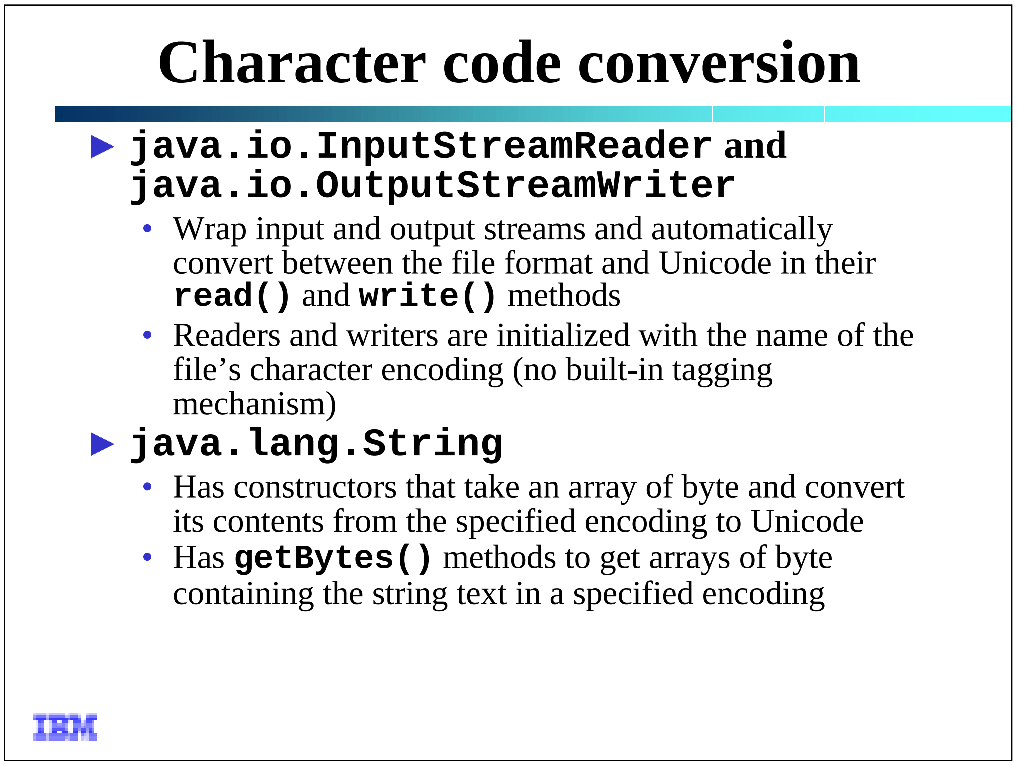


Handling non-Unicode text

- **Most text files will not be in Unicode**
- **Those that are can be in many flavors of Unicode:**
  - UTF-16BE
  - UTF-16LE
  - UTF-8
  - UCS-4
- **The `java.io` framework includes classes to handle this**

IBM

It’s nice that Java uses Unicode for text storage, but the whole world hasn’t seen the light yet, so there has to be a way for a Java program to read files and receive data that’s in other character encodings (in fact, various flavors of Unicode also count as foreign character encodings). Support for character code conversion is in the java.io package.

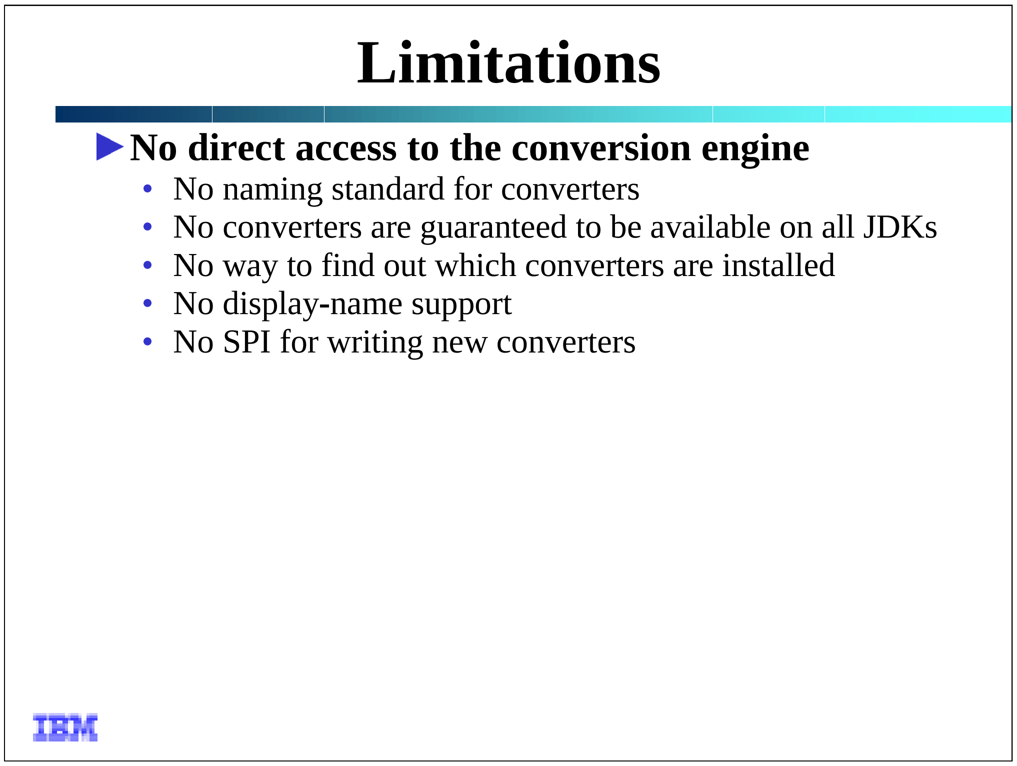


Character code conversion

- ▶ **`java.io.InputStreamReader`** **and** **`java.io.OutputStreamWriter`**
  - Wrap input and output streams and automatically convert between the file format and Unicode in their **`read()`** and **`write()`** methods
  - Readers and writers are initialized with the name of the file’s character encoding (no built-in tagging mechanism)
- ▶ **`java.lang.String`**
  - Has constructors that take an array of byte and convert its contents from the specified encoding to Unicode
  - Has **`getBytes()`** methods to get arrays of byte containing the string text in a specified encoding

IBM

The Java I/O library has classes called “readers” and “writers” that can be wrapped around streams to perform the conversions transparently. The String class also has API to convert from Unicode to something else and back.



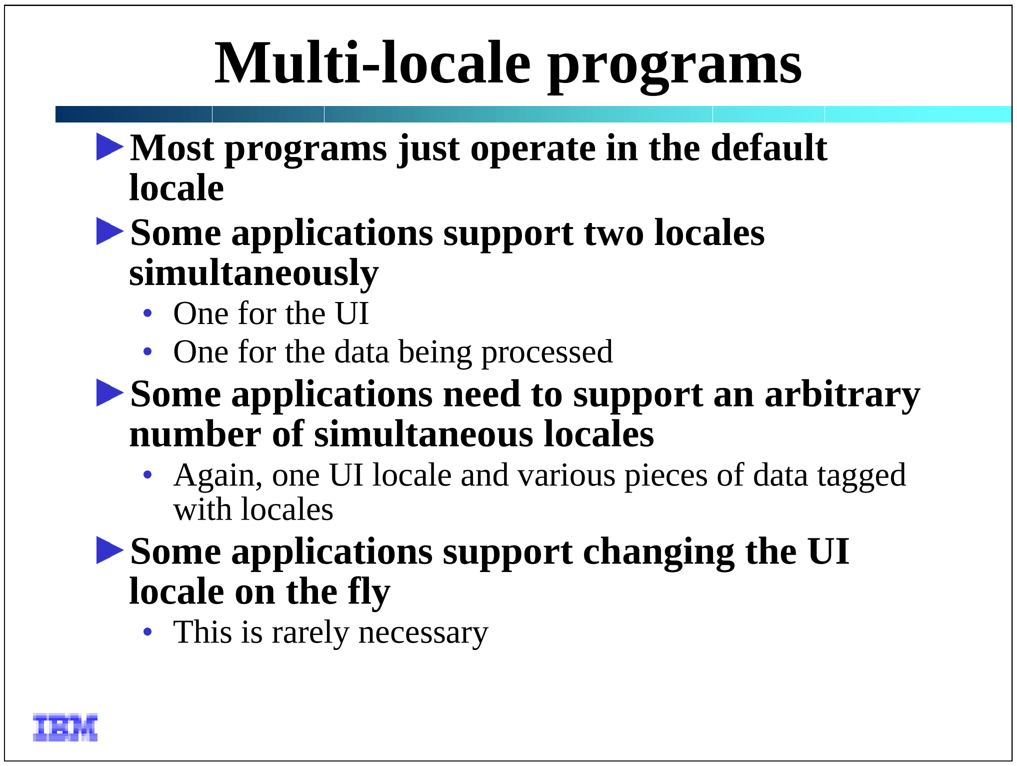
Limitations

- **No direct access to the conversion engine**
  - No naming standard for converters
  - No converters are guaranteed to be available on all JDKs
  - No way to find out which converters are installed
  - No display-name support
  - No SPI for writing new converters

IBM

There's no direct access to the converters in the API, which limits the control an application has. You can't write your own converters for the I/O framework to use, and since there's no way to get a list of available converters and no Java implementation is required to support any particular converter, the conversion library is only useful if you don't need to have the user pick an encoding and you either have control over all of the environments where your program will be run or can use the exception-handling mechanism to deal with situations where the desired converter isn't available.

Server-Side Internationalization



Multi-locale programs

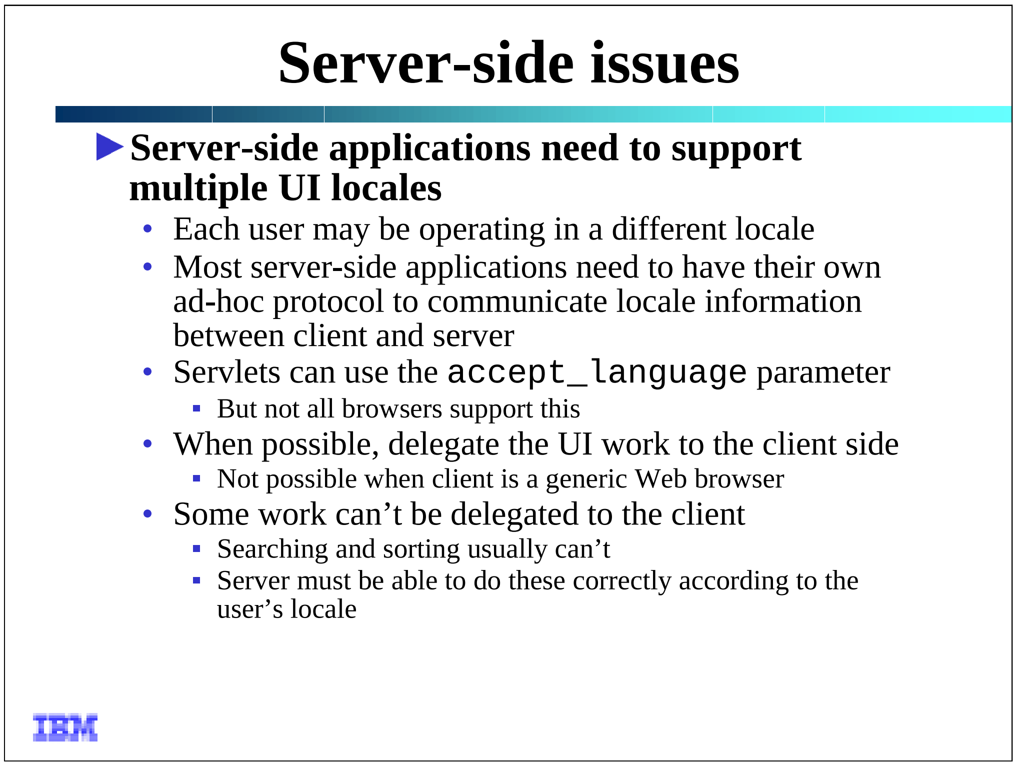
- **Most programs just operate in the default locale**
- **Some applications support two locales simultaneously**
  - One for the UI
  - One for the data being processed
- **Some applications need to support an arbitrary number of simultaneous locales**
  - Again, one UI locale and various pieces of data tagged with locales
- **Some applications support changing the UI locale on the fly**
  - This is rarely necessary

IBM

There are a few issues related to supporting multiple locales at the same time that are worth commenting on. Most applications don't need to worry about locale; they can just use the default locale (which is picked up from the host environment) and never explicitly deal with locales.

Sometimes, it's nice if you can use one locale for the UI and a different UI for the data you're working on. In fact, maybe the data itself will need multiple locales (all specified in the data somewhere). Again, you'd use the default locale for the UI and explicitly specify the locale when dealing with the data.

It's usually not necessary to support multiple UI locales. Some applications allow the user to change the UI locale on the fly while the program is running. This makes for a cool demo, but usually isn't necessary in real life. The big exception to this is a program that is never shut down but may be used at different times by different users.



Server-side issues

- **Server-side applications need to support multiple UI locales**
  - Each user may be operating in a different locale
  - Most server-side applications need to have their own ad-hoc protocol to communicate locale information between client and server
  - Servlets can use the `accept_language` parameter
    - But not all browsers support this
  - When possible, delegate the UI work to the client side
    - Not possible when client is a generic Web browser
  - Some work can't be delegated to the client
    - Searching and sorting usually can't
    - Server must be able to do these correctly according to the user's locale

IBM

…which is a great description for a server-side application. If you're writing an application that runs on a server and deals with remote users, then it has to be able to handle multiple simultaneous users that may all be operating in different locales. This requires some kind of protocol for the client and the server to exchange locale information. Java doesn't provide a built-in way to do this, because the protocol is so application-specific. (If the communication protocol is HTTP, you can use its accept_language tag, but not all browsers support this properly.)

Generally, as much locale-specific processing as possible (number and date formatting, for example) should be performed on the client side (the protocol between client and serve is locale-independent). But this isn't always possible. For some operations, such as searching and sorting, it's very rarely possible. Here, the locale-specific code has to reside on the server.

Other talks to hear

- **For useful tips on using the Java I18N APIs**
  - Helena Shih, Thursday,10:00 AM, Track A
- **For more on the text-editing APIs**
  - Brian Beck, Wednesday, 2:05 PM, Track C
  - Doug Felt & John Raley, Wednesday, 4:50 PM, Track C
- **For more on the Input Method Framework**
  - Norbert Lindenberg, Thursday, 3:50 PM, Track C
- **For more on searching in Unicode text**
  - Laura Werner, Wednesday, 10:00 AM, Track B
- **For more on international calendars**
  - Laura Werner, Thursday, 10:45 AM, Track A
- **For more on BreakIterator**
  - Richard Gillam, Wednesday, 10:45 AM, Track B

For More Info...

- **JDK class and method documentation**
  - `http://www.java.sun.com/products/jdk/1.2/docs/api/index.html`
  - `http://www.java.sun.com/products/jdk/1.2/docs/guide/internat/index.html`
- **Java internationalization tutorial**
  - `http://java.sun.com/docs/books/tutorial/i18n/index.html`
- **Additional IBM classes for internationalization**
  - `http://www.alphaworks.ibm.com/tech` (click on "International")
- **IBM's Classes for Unicode**
  - `http://www.ibm.com/java/tools/international-classes/`
- **Various internationalization-related IBM papers**
  - `http://www.ibm.com/java/education/papers.html`
  - `http://www2.software.ibm.com/developer/papers.nsf/java-papers-bytopic` (click on "Unicode")
- **Unicode home page**
  - `http://www.unicode.org`
- **Me (`rgillam@us.ibm.com`)**