Java string format number decimal places

You can use DecimalFormat classes to format decimal numbers into locally specific strings. This class allows you to control leading zero displays and trailing, prefire and enough, groups (thousands) of separatists, and decimal separatists. If you want to change the formatting symbol, such as a decimal separation, you can use DecimalFormatSymbols in conjunction with the DecimalFormat class. These classes offer a lot of flexibility in number formatting, but they can make your code more complex. The text follows using examples showing DecimalFormat and DecimalFormatSymbols classes. Examples of codes in this material are from a sample program called DecimalFormatDemo. Building Your Pattern determines the formatting properties of DecimalFormat with a string of patterns. The pattern determines what the formatted number looks like. For a full description of the syntax patterns, see Syntax Pattern Format Number. The following example creates the formatter by passing a string of patterns to the DecimalFormat developer. The format method receives multiple values as an argument and returns the number formatted in the string: DecimalFormat myFormatter = New DecimalFormat (pattern); String output = myFormatter.format (value); System.out.println (value + + pattern + output); Output for the previous line of code is described in the following table. The value is numbers, twice as much , that is to be formatted. The pattern is a string that determines the nature of the formatting. The output, which is a string, represents a formatted number. Output from DecimalFormatDemo Program pattern output value of Explanation 123456.789 ###,#.### 123,456.789 Pound marks (#) marks the digits, the comma is a placeholder for group separatists, and that period is a placeholder for decimal separatists. 123456.789 ###.# 123456.79 Its value has triple digits to the right of the decimal point, but the pattern has only two. The format method handles this by comparing. 123.78 0000000.000 000123.780 Pattern determines the leading zero and tracks, since 0 characters are used instead of pound marks (#). 12345.67 $#,#.### $12,345.67 The first character in a pattern is a dollar mark ($). Note that it immediately precedes the left digits in formatted output. 12345.67 \u00A5#,#.### ¥12,345.67 Pattern determines the currency mark for the Japanese yen (¥) with Unicode 00A5 value. Local-Sensitive formatting Previous examples created a DecimalFormat object for the default Local. If you want a DecimalFormat object for a bad Local, you as well as numberFormat law and then send it to DecimalFormat. Here's an example: NumberFormat nf = NumberFormat.getNumberInstance (loc); DecimalFormat df = (DecimalFormat)nf; df.applyPattern(pattern); String output = df.format (value); System.out.println(pattern + output + + loc.toString(); Conduct results of previous code examples in the following output. Formatted numbers, which are in the second column, varies with Locale: ###.### 123,456.789 en_US ###.### 123.456,789 de_DE ###,#.### 123 456,789 fr_FR So far the formatting pattern discussed here follows the A. For example, in the #pattern, the coma is a thousands separatist and the duration represents a decimal point. The convention is fine, provided your end user is not exposed to it. However, some applications, such as spreadsheets and report generators, allow end users to determine their own formatting patterns. For this application, the formatting patterns specified by the end user should use local notations. In this case, you will want to use the ApplyLocalizedPattern method on the DecimalFormat object. Changing Formatting Symbols You can use the DecimalFormatSymbols class to change the symbols that appear in the format number generated by the format method. These symbols include decimal separatists, group separatists, push marks, and percent marks, among other things. The next example shows the DecimalFormatSymbols class by using a strange format on the numbers. The incredible format is the result of calls to setDecimalSeparator, setGroupingSeparator, and setGroupingSize methods. Extraordinary DecimalFormatSymbols = New DecimalFormatSymbols (duringLocale); unusualSymbols.setDecimalSeparator('|'); unusualSymbols.setGroupingSeparator('^'); Strange string = #,##0.###; Weird DecimalFormate = New DecimalFormat (strange, unusual); weirdFormatter.setGroupingSize(4); Strange string = weirdFormatter.format (12345.678); System.out.println (strange); When carried out, this example prints the number in a strange format: Syntax Pattern Format Number You can design your own format patterns for numbers by following the rules specified by the following BNF diagram: pattern := subpattern{;subpattern} subpattern:= {prefix}integer{.fraction}prefix := '\u000.' - SpecialCharacters enough := '\u0000'.' \\uFFFD' - integer SpecialCharacters := '#'* '0'* '0' breakdown := '0'* '#'* The notation used in the previous diagram is described in the following table: Notification Description X* 0 or more example X (X | Y) either X or Y X.. Y any characters from X up to Y, including S - Characters T in S, except those in T {X} X are options In previous BNF diagrams, the first subpattern determines the format for positive numbers. The second subpattern, which is an option, determines the format for negative numbers. Although unanticed in bnF diagrams, a coma may appear in the integer section. In subpatterns, you specify formatting with special symbols. This symbol is described next table: 0-digit Description Symbol #a indicates as absent. placeholder for decimal separation, placeholder for group E separation separates mantissa and exponents for exponential format; separate the format - the default negative prefifit % multiplied by 100 and shows as a percentage? multiplied by 1000 and show according to ¤ currency; replaced by a currency symbol; if twice, replaced by an international currency symbol; if found in a pattern, a monetary decimal separater is used instead of the X decimal separater any other character can be used in prefixes or suffix' used to quote special characters in the prefix or previous suffix you see the use of print and println methods to print strings to standard output (System.out). Since all numbers can be converted to a string (as you can see later in this lesson), you can use this method to print an arbitrary mix of strings and numbers. The Java programming language has other methods, however, that allows you to run more control over your print output when the number is included. The Print methods and format package java.io include a PrintStream class that has two formatting methods that you can use to replace print and print. These methods, formats and prints, the equivalent of each other. The usual system.out you use occurs as a PrintStream object, so you can use the PrintStream method on System.out. Therefore, you can use formats or prints anywhere in your code where you previously used print or print. For example, System.out.format.....); Syntax for both java.io.PrintStream methods are the same: public print format (string format, Objects ... args) where the format is a string that determines which formatting will be used and args are a list of variables to be printed using the formatting. A simple example is System.out.format (Value + float variable is + %f, while the value + integer variable is %d, + and the string is %s, floatVar, intVar, stringVar); The first parameter, the format, is a string of formats that determine how objects in the second parameter, args, will be formatted. The format string contains plain text as well as format specifiers, which are special characters that format object arguments ... args. (Notation Object... args are called varargs, which means that the number of arguments may vary.) The format of specifiers starts with a percent sign (%) and end with the converter. Converters are characters that indicate what kind of argument to be formatted. Between percent signs (%) and converter you can have a preferred flag and specifiers. There are many converters, flags, and specifiers, documented in java.util.Formatter Here are examples int i = 461012; System.out.format (Value i is: %d%n, i); %d determines that a single variable is a decimal integer. %n is new line characters. Output is: Value i is: 461012 Print method and format loaded. Each room has a version with the following storage: public Print Flow format (Locale l, string format, Objects ... args) To print numbers in the French system (where commas are used to replace decimal places in English representation of floating point numbers), for example, you will use: System.out.format (Locale.FRANCE, The floating value + variable is %f, while + the integer variable value + is %d, and the string is %s%n, floatVar, intVar, stringVar); Examples Of The following table lists some of the converters and flags used in sample programs, TestFormat.java, which follows the schedule. Converter and Flag Used in TestFormat.java Clearing Flag d Decimal Integer. f A float. n A new line character that fits the platform that runs the app. You should always use %n, rather than . tB Date &amp;; time change—a full local special name of the moon. td, press Date &amp; time conversion—2-digit days of the month. td has as leading zero as needed, te no. ty, tY date &amp; time conversion—ty = 2 digits of the year, tY = 4 digits of the year. tl Date &amp;; time-conversion—clocks within 12 hours. tM Date &amp;3;00 time conversion—minutes in 2 digits, with leading zero if necessary. tp Date &amp;; time conversion—general/pm specific local (lower case). tm Date &amp;; time conversion—month in 2 digits, with leading zero if necessary. tD Date &amp;; time conversion—date as %tm%td%ty 08 Eight wide characters, with leading zero as needed. + Includes signs, whether positive or negative. , Includes characters of locally designed groups. - Desirable left.. .3 Three spots after decimal point. 10.3 Ten characters wide, desirable, with three spots after decimal point. The following programs show some of the formatting you can do with the format. Output is shown in double quotes in embedded comments: import java.util.Calendar; import java.util.Locale; public class testFormat { public invalid static primary (Rope[] args) { length n = 461012; System.out.format(%d%n, n); --&gt; 461012 System.out.format(%08d%n, n); --&gt; 00461012 System.out.format(%+8d%n, n); --&gt; +461012 System.out.format(%,8d%n, n); --&gt; 461,012 System.out.format (%+,8d%n%n, n); --&gt; +461,012 double pi = Math.PI; System.out.format(%f%n, pi); --&gt; 3.141593 System.out.format(%.3f%n, pi); --&gt; 3.142 System.out.format(%10.3f%n, pi); --&gt; 3.142 System.out.format(%-10.3f%n, pi); --&gt; 3.142 System.out.format (Locale.FRANCE, %-10.4f%n%n, pi); --&gt; 3,1416 Calendar.getInstance(); System.out.format(%tB %te, %tY%n, c, c, c); --&gt; May 29, 2006 System.out.format(%tl:%tM %tp%n, c, c, c); --&gt; 2:34 a.m. (%tD%n, c); (c) --&gt; 05/29/06 } } Note: Discussion in this section only covers the basics of the format and print method. More details can be found in the Basic I/O section of the Important Footprint, in the Formatting page. Using String.format to create protected strings in Strings. DecimalFormat Class You can use java.text.DecimalFormat classes to control leading zero displays and track, prefixes and enough, groups (thousands) of separatists, and decimal separations. DecimalFormat offers a lot of flexibility in number formatting, but it can make your code more complex. The following example creates the DecimalFormat object, myFormatter, by passing a string of patterns to the DecimalFormat developer. Format method(), which DecimalFormat inherits from NumberFormat, is then invoiced by myFormatter—it receives double value as an argument and returns the number formatted in the string: Here's a sample program that describes the use of DecimalFormat: import java.text.*; public class DecimalFormatDemo { customFormat is not valid public static (string pattern, double value) { DecimalFormat myFormatter = DecimalFormat (pattern); String output = myFormatter.format (value); System.out.println (value + + pattern + output); } major invalid public statics (String[] args) { customFormat(#,#.###, 123456.789); customFormat(###, 123456.789); customFormat (000000.000, 123.78); customFormat(#.###, 12345.67} Its output is: 123456.789 ##,#.### 123,456.789 123456.789 #123456.79 123.78 0000000.000 000123.780 12345.67 $#,#.### $12,345.67 The following table describes each output row. DecimalFormat.java Output Value Output Description 123456.789 ##,#.### 123,456.789 Pound marks (#) marking the digits, the comma is a placeholder for group separatists, and that period is a placeholder for decimal separatists. 123456.789 ###.# 123456.79 Its value has triple digits to the right of the decimal point, but the pattern has only two. The format method handles this by comparing. 123.78 0000000.000 000123.780 Pattern determines the leading zero and tracks, since 0 characters are used instead of pound marks (#). 12345.67 $#,#.### $12,345.67 The first character in a pattern is a dollar mark ($). Note that it immediately precedes the left digits in formatted output. Output.