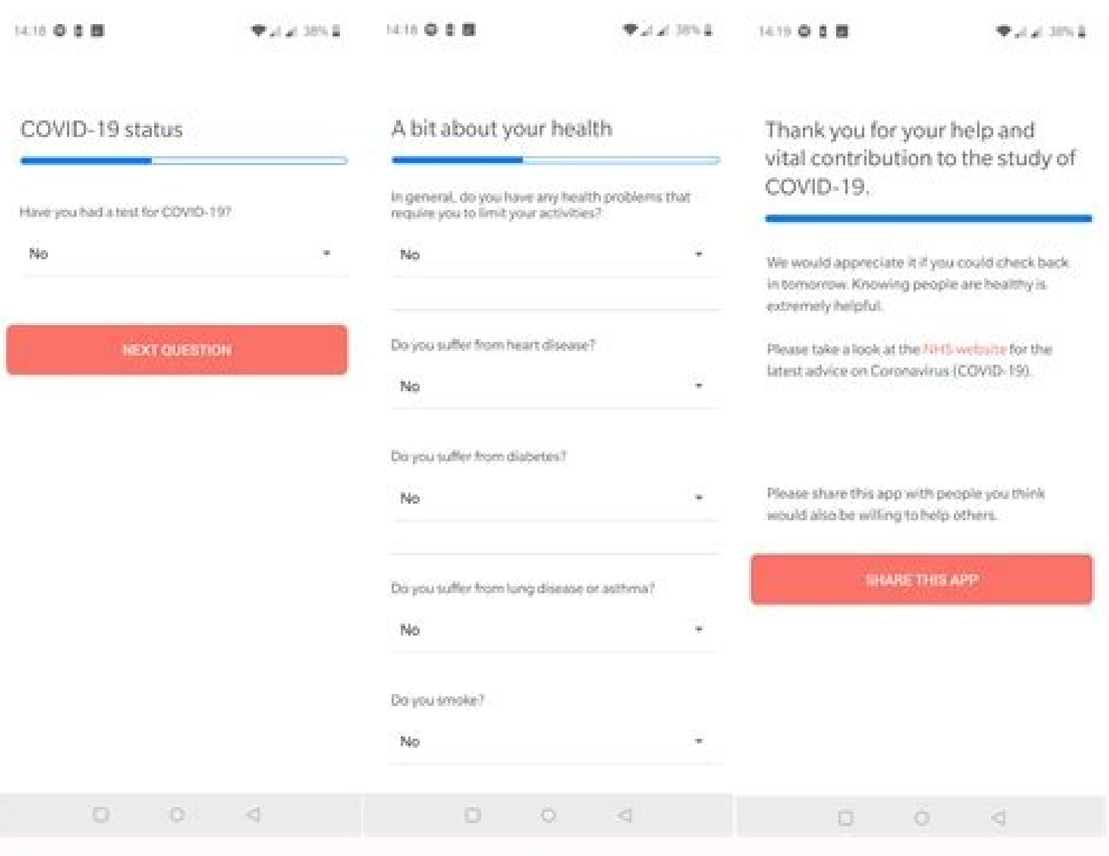
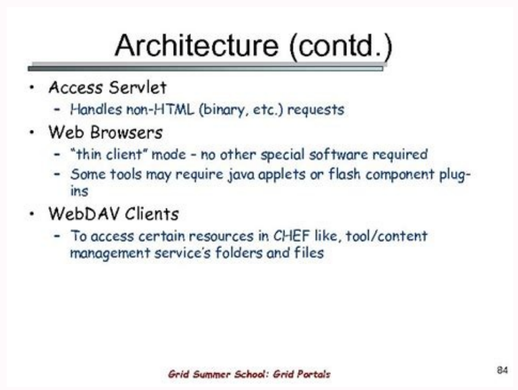
Java resource bundle properties file format





This section steps through a sample program named PropertiesDemo. 1. Create the Default Properties File A properties file is a simple text file. You can create and maintain a properties file with just about any text editor. You should always create a default properties file. The name of this file begins with the base name of your ResourceBundle and ends with the .properties suffix. In the PropertiesDemo program the base name is LabelsBundle. Therefore the default properties file is called LabelsBundle.properties. This file contains the following lines: # This is the default LabelsBundle.properties file s1 = computer s2 = disk s3 = monitor s4 = keyboard Note that in the preceding file the comment lines begin with a pound sign (#). The other lines contain key-value pairs. The key is on the left side of the equal sign and the value is on the right. For instance, s2 is the key that corresponds to the value disk. The key is arbitrary. We could have called s2 something else, like msg5 or diskID. Once defined, however, the key should not change because it is referenced in the source code. The values may be changed. In fact, when your localizers create new properties files to accommodate additional languages, they will translate the values into various languages. 2. Create Additional Properties Files as Needed To support an additional Locale, your localizers will create a new properties file that contains the translated values. No changes to your source code are required, because your program references the keys, not the values. For example, to add support for the German language, your localizers would translate the values in LabelsBundle.properties and place them in a file named LabelsBundle_de.properties. Notice that the name of this file, like that of the default file, begins with the base name LabelsBundle and ends with the .properties suffix. However, since this file is intended for a specific Locale, the base name is followed by the language code (de). The contents of LabelsBundle_de.properties are as follows: # This is the LabelsBundle_de.properties file s1 = Computer s2 = Platte s3 = Monitor s4 = Tastatur The PropertiesDemo sample program ships with three properties files: LabelsBundle.properties LabelsBundle_de.properties LabelsBundle_fr.properties 3. Specify the Locale The PropertiesDemo program creates the Locale objects as follows: Locale[] supportedLocales = { Locale.FRENCH, Locale.GERMAN, Locale.ENGLISH }; These Locale objects should match the properties files created in the previous two steps. For example, the Locale.FRENCH object corresponds to the LabelsBundle_fr.properties file. The Locale.ENGLISH has no matching LabelsBundle_en.properties file, so the default file will be used. 4. Create the ResourceBundle This step shows how the Locale, the properties files, and the ResourceBundle are related. To create the ResourceBundle, invoke the getBundlemethod, specifying the base name and Locale: ResourceBundle labels = ResourceBundle.getBundle("LabelsBundle", currentLocale); The getBundle method first looks for a class file that matches the base name and the Locale. If it can't find a class file, it then checks for properties files. In the PropertiesDemo program we're backing the ResourceBundle with properties files instead of class files. When the getBundle method locates the correct properties file, it returns a PropertyResourceBundle object containing the key-value pairs from the properties file. 5. Fetch the Localized Text To retrieve the translated value from the ResourceBundle, invoke the getString method as follows: String value = labels.getString(key); The String returned by getString corresponds to the key specified. The String is in the proper language, provided that a properties file exists for the specified Locale. 6. Iterate through All the Keys This step is optional. When debugging your program, you might want to fetch values for all of the keys in a ResourceBundle. The getKeys method returns an Enumeration of all the keys in a ResourceBundle. You can iterate through the Enumeration and fetch each value with the getString method. The following lines of code, which are from the PropertiesDemo program, show how this is done: ResourceBundle labels = ResourceBundle.getBundle("LabelsBundle", currentLocale); Enumeration bundleKeys = labels.getKeys(); while (bundleKeys.hasMoreElements()) { String key = (String)bundleKeys.nextElement(); String value = labels.getString(key); System.out.println("key = " + key + ", " + "value = " + value); } 7. Run the Demo Program Running the PropertiesDemo program generates the following output. The first three lines show the values returned by getString for various Locale objects. The program displays the last four lines when iterating through the keys with the getKeys method. Locale = fr, key = s2, value = Disque dur Locale = de, key = s2, value = Platte Locale = en, key = s2, value = disk key = s4, value = Clavier key = s3, value = Moniteur key = s2, value = Disque dur key = s1, value = Ordinateur A resource bundle is a set of properties files with the same base name and a language-specific suffix. For example, if you create file_en.properties and file_de.properties, IntelliJ IDEA will recognize and combine them into a resource bundle.IntelliJ IDEA marks resource bundles with the following icon: . In the Project tool window, select the directory where the new resource bundle should be created.From the main menu, select , or press Alt+Insert and click Resource Bundle.In the dialog that opens, specify the base name of the resource bundle.If necessary, select the Use XML-based properties files.To add locales, click and type the comma-separated suffixes of the required locales.Click OK when ready. A new node Resource Bundle '' appears in the Project tool window:By default, the bundle contains properties files for all specified locales. You can dissociate it to show only the properties files without the bundle. Right-click the resource bundle you want to dissociate.From the context menu, click Dissociate Resource Bundle. Select the properties files to be combined.Right-click the selection.From the context menu, click Combine to Resource Bundle.Specify the base name of the resource bundle. Once you create several properties files with the same name and different locale suffixes, IntelliJ IDEA automatically recognizes them and groups them in the Project view into a resource bundle.IntelliJ IDEA includes a Resource Bundle editor for convenient editing of localizable strings in properties files. Do one of the following:Select a resource bundle in the Project tool window, and press F4.Open a properties file that is a part of a bundle and click the Resource Bundle tab at the bottom of the editor: Open a properties file.Add, change, or delete keys as required. IntelliJ IDEA reflects the changes in the Resource Bundle editor. Use the Resource Bundle editor to change property values, which will ensure that you edit the entire set of property files simultaneously. IntelliJ IDEA creates respective records in each file of the bundle. Select the property key in the left pane of the resource bundle editor.In the target locale frame, edit the value as required. IntelliJ IDEA updates the respective properties file accordingly.Here, you can also add new properties (use the button), sort, and group properties. When editing a resource bundle, keep the following in mind:IntelliJ IDEA highlights properties with no values or those omitted in one of the properties files.To convert between escape sequences (for example, \u00df) and unicode literals (corresponding national characters, such as ß) in properties files and in the Resource Bundle editor, select the Transparent native-to-ascii conversion checkbox on the File Encoding page of the IDE settings.For more information, see Encoding.It is possible to encode non-ASCII symbols using both uppercase and lowercase hex sequences (for example, \u00E3 and \u00e3). By default, IntelliJ IDEA supports only uppercase sequences. To use lowercase hex sequences, set the idea.native2ascii.lowercase property in the idea.properties file to true.For more information, see Platform properties. When editing properties files, you can press Alt+Enter to use the context actions for copying the key or value of the property at the caret to the clipboard. Last modified: 26 July 2022 java.util.ResourceBundle class enables you to choose and read the properties file specific to the user's locale and look up the values.A ResourceBundle object has a base name. In order for a ResourceBundle object to pick up a properties file, the filename must be composed of the ResourceBundle base name, followed by an underscore, followed by the language code, and optionally followed by another underscore and the country code.The format for the properties file name is as follows:basename_languageCode_countryCode For example, suppose the base name is MyResources and you define the following three locales:US-en DE-de CN-zh Then you would have these three properties files:MyResources_en_US.properties MyResources_de_DE.properties MyResources_zh_CN.properties Page 2 ResourceBundle class is used to store text and objects which are locale sensitive. Generally we use property files to store locale specific text and then represent them using ResourceBundle object. Following are the steps to use locale specific properties file in a java based application. Step 1: Create properties files. Suppose we need properties file for English locale. Then create a properties file name XXX_en_US.properties where XXX is the name of the file and en_US represents the locale for English(US). Messages_en_US.properties message=Welcome to TutorialsPoint.COM! Let's now create properties file for French locale. Then create a properties file name XXX_fr_FR.properties where XXX is the name of the file and fr_FR represents the locale for French(France). Messages_fr_FR.properties message=Bienvenue sur TutorialsPoint.COM! Here you can figure out that the key is same but the value is locale specific in both the properties file. Step 2: Create ResourceBundle object Create ResourceBundle object with properties file name and locale using following syntax. ResourceBundle bundle = ResourceBundle.getBundle("Messages", Locale.US); Step 3: Get the value from ResourceBundle object. Get the value from ResourceBundle object by passing the key. String value = bundle.getString("message"); Example Following example illustrate the use of ResourceBundle objects to display locale specific values from properties files. IOTester.java import java.util.Locale; import java.util.ResourceBundle; public class I18NTester { public static void main(String[] args) { ResourceBundle bundle = ResourceBundle.getBundle("Messages", Locale.US); System.out.println("Message in "+Locale.US +": "+bundle.getString("message")); bundle = ResourceBundle.getBundle("Messages", Locale.FRANCE); System.out.println("Message in "+Locale.FRANCE +": "+bundle.getString("message")); } } Output It will print the following result. Message in en_US: Welcome to TutorialsPoint.COM! Message in fr_FR: Bienvenue sur TutorialsPoint.COM! Notes for Naming Conventions Following are the naming conventions for the properties file. For properties file mapped to default locale, no prefix is mandatory. message_en_US.properties is equivalent to message.properties. For properties file mapped to locale, prefix can be attached in two ways. message_fr.properties is equivalent to message_fr_FR.properties. Print Page 2 The java.text.NumberFormat class is used for formatting numbers and currencies as per a specific Locale. Number formats varies from country to country. For example, In Denmark fractions of a number are separated from the integer part using a comma whereas in England they use a dot as separator. Example - Format Numbers In this example, we're formatting numbers based on US locale and Danish Locale. IOTester.java import java.text.NumberFormat; import java.util.Locale; public class I18NTester { public static void main(String[] args) { Locale enLocale = new Locale("en", "US"); Locale daLocale = new Locale("da", "DK"); NumberFormat numberFormat = NumberFormat.getInstance(daLocale); System.out.println(numberFormat.format(100.76)); numberFormat = NumberFormat.getInstance(enLocale); System.out.println(numberFormat.format(100.76)); } } Output It will print the following result. 100,76 100.76 Print Page 3 In this example, we're formatting currencies based on US locale and Danish Locale. IOTester.java import java.text.NumberFormat; import java.util.Locale; public class I18NTester { public static void main(String[] args) { Locale enLocale = new Locale("en", "US"); Locale daLocale = new Locale("da", "DK"); NumberFormat numberFormat = NumberFormat.getCurrencyInstance(daLocale); System.out.println(numberFormat.format(100.76)); numberFormat = NumberFormat.getCurrencyInstance(enLocale); System.out.println(numberFormat.format(100.76)); } } Output It will print the following result. kr 100,76 $100.76 Print Page 4 In this example, we're formatting numbers in percentage format. IOTester.java import java.text.NumberFormat; import java.util.Locale; public class I18NTester { public static void main(String[] args) { Locale enLocale = new Locale("en", "US"); NumberFormat numberFormat = NumberFormat.getPercentInstance(enLocale); System.out.println(numberFormat.format(0.76)); } } Output It will print the following result. 76% Print Page 5 In this example, we're setting min and max digits for both integer as well as fractional part of a number. IOTester.java import java.text.NumberFormat; import java.util.Locale; public class I18NTester { public static void main(String[] args) { Locale enLocale = new Locale("en", "US"); NumberFormat numberFormat = NumberFormat.getInstance(enLocale); numberFormat.setMinimumIntegerDigits(2); numberFormat.setMaximumIntegerDigits(3); numberFormat.setMinimumFractionDigits(2); numberFormat.setMaximumFractionDigits(3); System.out.println(numberFormat.format(12234.763443)); } } Output It will print the following result. 234.763 Print Page 6 In this example, we're showcasing Rounding Mode. IOTester.java import java.math.RoundingMode; import java.text.NumberFormat; import java.util.Locale; public class I18NTester { public static void main(String[] args) { Locale enLocale = new Locale("en", "US"); NumberFormat numberFormat = NumberFormat.getInstance(enLocale); numberFormat.setMinimumFractionDigits(0); numberFormat.setMaximumFractionDigits(0); System.out.println(numberFormat.format(99.50)); numberFormat.setRoundingMode(RoundingMode.HALF_DOWN); System.out.println(numberFormat.format(99.50)); } } Output It will print the following result. 100 99 Print Page 7 In this example, we're showcasing parsing of number present in different locale. IOTester.java import java.text.NumberFormat; import java.text.ParseException; import java.util.Locale; public class I18NTester { public static void main(String[] args) throws ParseException { Locale enLocale = new Locale("en", "US"); Locale daLocale = new Locale("da", "DK"); NumberFormat numberFormat = NumberFormat.getInstance(daLocale); System.out.println(numberFormat.parse("100,76")); numberFormat = NumberFormat.getInstance(enLocale); System.out.println(numberFormat.parse("100,76")); } } Output It will print the following result. 100.76 100.76 Print Page 8 The java.text.DecimalFormat class is used for formatting numbers as per customized format and as per locale. Example - Format Numbers In this example, we're formatting numbers based on a given pattern. IOTester.java import java.text.DecimalFormat; public class I18NTester { public static void main(String[] args) { String pattern = "####,####.##"; double number = 123456789.123; DecimalFormat numberFormat = new DecimalFormat(pattern); System.out.println(number); System.out.println(numberFormat.format(number)); } } Output It will print the following result. 1.23456789123E8 1,2345,6789.12 Print Page 9 Followings is the use of characters in formatting patterns. Sr.No.Class & Description 10To display 0 if less digits are present. 2#To display digit ommitting leading zeroes. 3.Decimal separator. 4,Grouping separator. 5EMantissa and Exponent separator for exponential formats. 6;Format separator. 7-Negative number prefix. 8%Shows number as percentage after multiplying with 100. 9?Shows number as mille after multiplying with 1000. 10XTo mark character as number prefix/suffix. 11'To mark quote around special characters. In this example, we're formatting numbers based on different patterns. IOTester.java import java.text.DecimalFormat; public class I18NTester { public static void main(String[] args) { String pattern = "###.###"; double number = 123456789.123; DecimalFormat numberFormat = new DecimalFormat(pattern); System.out.println(number); //pattern ###.### System.out.println(numberFormat.format(number)); //pattern ###.# numberFormat.applyPattern("###.#"); System.out.println(numberFormat.format(number)); //pattern ###,###.## numberFormat.applyPattern("###,###.##"); System.out.println(numberFormat.format(number)); number = 9.34; //pattern 000.### numberFormat.applyPattern("000.##"); System.out.println(numberFormat.format(number)); } } Output It will print the following result. 1.23456789123E8 1,2345,6789.12 Print Page 10 By default, DecimalFormat object is using the JVM's locale. We can change the default locale while creating the DecimalFormat object using NumberFormat class. In the example below, we'll use same pattern for two different locale and you can spot the difference in the output. IOTester.java import java.text.DecimalFormat; import java.text.NumberFormat; import java.util.Locale; public class I18NTester { public static void main(String[] args) { String pattern = "###.##"; double number = 123.45; Locale enlocale = new Locale("en", "US"); Locale dalocale = new Locale("da", "DK"); DecimalFormat decimalFormat = (DecimalFormat) NumberFormat.getNumberInstance(enlocale); decimalFormat.applyPattern(pattern); System.out.println(decimalFormat.format(number)); decimalFormat = (DecimalFormat) NumberFormat.getNumberInstance(dalocale); decimalFormat.applyPattern(pattern); System.out.println(decimalFormat.format(number)); } } Output It will print the following result. 123.45 123,45 Print Page 11 Using DecimalFormatSymbols class, the default separator symbols, grouping separator symbols etc. can be changed. Following example is illustrating the same. IOTester.java import java.text.DecimalFormat; import java.text.DecimalFormatSymbols; public class I18NTester { public static void main(String[] args) { String pattern = "#,###.###"; double number = 126473.4567; DecimalFormat decimalFormat = new DecimalFormat(pattern); System.out.println(decimalFormat.format(number)); DecimalFormatSymbols decimalFormatSymbols = new DecimalFormatSymbols(); decimalFormatSymbols.setDecimalSeparator(';'); decimalFormatSymbols.setGroupingSeparator(':'); decimalFormat = new DecimalFormat(pattern, decimalFormatSymbols); System.out.println(decimalFormat.format(number)); } } Output It will print the following result. 126,473.457 126:473;457 Print Page 12 Using setGroupingSize() method of DecimalFormat, default grouping of numbers can be changed. Following example is illustrating the same. IOTester.java import java.text.DecimalFormat; public class I18NTester { public static void main(String[] args) { double number = 121223232473.4567; DecimalFormat decimalFormat = new DecimalFormat(); System.out.println(number); System.out.println(decimalFormat.format(number)); decimalFormat.setGroupingSize(4); System.out.println(decimalFormat.format(number)); } } Output It will print the following result. 1.212232324734567E11 121,223,232,473.457 1212,2323,2473.457 Print Page 13 java.text.DateFormat class formats dates as per the locale. As different coutries use different formats to display dates. This class is extremely useful in dealing with dates in internalization of application. Following example show how to create and use DateFormat Class. IOTester.java import java.text.DateFormat; import java.util.Date; import java.util.Locale; public class I18NTester { public static void main(String[] args) { Locale locale = new Locale("da","DK"); DateFormat dateFormat = DateFormat.getDateInstance(); System.out.println(dateFormat.format(new Date())); dateFormat = DateFormat.getDateInstance(DateFormat.DEFAULT, locale); System.out.println(dateFormat.format(new Date())); } } Output It will print the following result. Nov 29, 2017 29-11-2017 Print Page 14 DateFormat class provides various formats to format the date. Following is list of some of the formats. DateFormat.DEFAULT DateFormat.SHORT DateFormat.MEDIUM DateFormat.LONG DateFormat.FULL In following example we'll show how to use different formats. IOTester.java import java.text.DateFormat; import java.util.Date; public class I18NTester { public static void main(String[] args) { DateFormat dateFormat = DateFormat.getDateInstance(DateFormat.DEFAULT); System.out.println(dateFormat.format(new Date())); dateFormat = DateFormat.getDateInstance(DateFormat.SHORT); System.out.println(dateFormat.format(new Date())); dateFormat = DateFormat.getDateInstance(DateFormat.MEDIUM); System.out.println(dateFormat.format(new Date())); dateFormat = DateFormat.getDateInstance(DateFormat.LONG); System.out.println(dateFormat.format(new Date())); dateFormat = DateFormat.getDateInstance(DateFormat.FULL); System.out.println(dateFormat.format(new Date())); } } Output It will print the following result. Nov 29, 2017 11/29/17 Nov 29, 2017 November 29, 2017 Wednesday, November 29, 2017 Print Page 15 DateFormat class provides various formats to format the time. DateFormat.getTimeInstance() method is to be used. See the example below. In following example we'll show how to use different formats to format time. IOTester.java import java.text.DateFormat; import java.util.Date; public class I18NTester { public static void main(String[] args) { DateFormat dateFormat = DateFormat.getTimeInstance(DateFormat.DEFAULT); System.out.println(dateFormat.format(new Date())); dateFormat = DateFormat.getTimeInstance(DateFormat.SHORT); System.out.println(dateFormat.format(new Date())); dateFormat = DateFormat.getTimeInstance(DateFormat.MEDIUM); System.out.println(dateFormat.format(new Date())); dateFormat = DateFormat.getTimeInstance(DateFormat.LONG); System.out.println(dateFormat.format(new Date())); dateFormat = DateFormat.getTimeInstance(DateFormat.FULL); System.out.println(dateFormat.format(new Date())); } } Output It will print the following result. 4:11:21 PM 4:11 PM 4:11:21 PM 4:11:21 PM IST 4:11:21 PM IST Print Page 16 DateFormat class provides various formats to format the date and time together. DateFormat.getDateTimeInstance() method is to be used. See the example below. In following example we'll show how to use different formats to format date and time. IOTester.java import java.text.DateFormat; import java.util.Date; public class I18NTester { public static void main(String[] args) { DateFormat dateFormat = DateFormat.getDateTimeInstance(DateFormat.DEFAULT, DateFormat.DEFAULT); System.out.println(dateFormat.format(new Date())); dateFormat = DateFormat.getDateTimeInstance(DateFormat.SHORT, DateFormat.SHORT); System.out.println(dateFormat.format(new Date())); dateFormat = DateFormat.getDateTimeInstance(DateFormat.MEDIUM, DateFormat.MEDIUM); System.out.println(dateFormat.format(new Date())); dateFormat = DateFormat.getDateTimeInstance(DateFormat.LONG, DateFormat.LONG); System.out.println(dateFormat.format(new Date())); dateFormat = DateFormat.getDateTimeInstance(DateFormat.FULL, DateFormat.FULL); System.out.println(dateFormat.format(new Date())); } } Output It will print the following result. Nov 29, 2017 4:16:13 PM 11/29/17 4:16 PM Nov 29, 2017 4:16:13 PM November 29, 2017 4:16:13 PM IST Wednesday, November 29, 2017 4:16:13 PM IST Print Page 17 java.text.SimpleDateFormat class formats dates as per the given pattern. It is also used to parse dates from string where string contains date in mentioned format. See the following example of using SimpleDateFormat class. IOTester.java import java.text.ParseException; import java.text.SimpleDateFormat; import java.util.Date; public class I18NTester { public static void main(String[] args) throws ParseException { String pattern = "dd-MM-yyyy"; SimpleDateFormat simpleDateFormat = new SimpleDateFormat(pattern); Date date = new Date(); System.out.println(date); System.out.println(simpleDateFormat.format(date)); String dateText = "29-11-2017"; date = simpleDateFormat.parse(dateText); System.out.println(simpleDateFormat.format(date)); } } Output It will print the following result. Wed Nov 29 17:01:22 IST 2017 29-11-2017 29-11-2017 Print Page 18 Locale can be used to create locale specific formatting over a pattern in SimpleDateFormat class. See the following example of using locale specific SimpleDateFormat class. IOTester.java import java.text.ParseException; import java.text.SimpleDateFormat; import java.util.Date; import java.util.Locale; public class I18NTester { public static void main(String[] args) throws ParseException { Locale locale = new Locale("da", "DK"); String pattern = "EEEEE MMMMM yyyy"; SimpleDateFormat simpleDateFormat = new SimpleDateFormat(pattern); Date date = new Date(); System.out.println(date); System.out.println(simpleDateFormat.format(date)); simpleDateFormat = new SimpleDateFormat(pattern,locale); System.out.println(simpleDateFormat.format(date)); } } Output It will print the following result. Wed Nov 29 17:48:14 IST 2017 Wednesday November 2017 onsdag november 2017 Print Page 19 Using DecimalFormatSymbols class, the default separator symbols, grouping separator symbols etc. can be changed. Following example is illustrating the same. IOTester.java import java.text.DecimalFormat; import java.text.DecimalFormatSymbols; public class I18NTester { public static void main(String[] args) { String pattern = "#,###.###"; double number = 126473.4567; DecimalFormat decimalFormat = new DecimalFormat(pattern); System.out.println(decimalFormat.format(number)); DecimalFormatSymbols decimalFormatSymbols = new DecimalFormatSymbols(); decimalFormatSymbols.setDecimalSeparator(';'); decimalFormatSymbols.setGroupingSeparator(':'); decimalFormat = new DecimalFormat(pattern, decimalFormatSymbols); System.out.println(decimalFormat.format(number)); } } Output It will print the following result. 126,473.457 126:473;457 Print Page 20 Followings is the use of characters in date formatting patterns. Sr.No.Class & Description 1GTo display Era. 2yTo display Year. Valid values yy, yyyy. 3MTo display Month. Valid values MM, MMM or MMMMM. 4dTo display day of month. Valid values d, dd. 5hTo display hour of day (1-12 AM/PM). Valid value hh. 6HTo display hour of day (0-23). Valid value HH. 7mTo display minute of hour (0-59). Valid value mm. 8sTo display second of minute (0-59). Valid value ss. 9STo display milliseconds of minute (0-999). Valid value SSS. 10ETo display Day in week (e.g Monday, Tuesday etc.) 11DTo display Day in year (1-366). 12FTo display Day of week in month (e.g. 1st Thursday of December). 13wTo display Week in year (1-53). 14WTo display Week in month (0-5) 15aTo display AM / PM 16kTo display Hour in day (1-24). 17KTo display Hour in day, AM / PM (0-11). 18zTo display Time Zone. In this example, we're formatting dates based on different patterns. IOTester.java import java.text.ParseException; import java.text.SimpleDateFormat; import java.util.Date; public class I18NTester { public static void main(String[] args) throws ParseException { String pattern = "dd-MM-yy"; SimpleDateFormat simpleDateFormat = new SimpleDateFormat(pattern); Date date = new Date(); System.out.println(simpleDateFormat.format(date)); pattern = "MM-dd-yyyy"; simpleDateFormat = new SimpleDateFormat(pattern); System.out.println(simpleDateFormat.format(date)); pattern = "yyyy-MM-dd HH:mm:ss"; simpleDateFormat = new SimpleDateFormat(pattern); System.out.println(simpleDateFormat.format(date)); pattern = "EEEEE MMMMM yyyy HH:mm:ss.SSSZ"; simpleDateFormat = new SimpleDateFormat(pattern); System.out.println(simpleDateFormat.format(date)); } } Output It will print the following result. 29-11-17 11-29-2017 2017-11-29 18:47:42 Wednesday November 2017 18:47:42.787+0530 Print Page 21 UTC stands for Co-ordinated Universal Time. It is time standard and is commonly used across the world. All timezones are computed comparatively with UTC as offset. For example, time in Copenhagen, Denmark is UTC + 1 means UTC time plus one hour. It is independent of Day light savings and should be used to store date and time in databases. Conversion of time zones Following example will showcase conversion of various timezones. We'll print hour of the day and time in milliseconds. First will vary and second will remain same. IOTester.java import java.text.ParseException; import java.util.Calendar; import java.util.GregorianCalendar; import java.util.TimeZone; public class I18NTester { public static void main(String[] args) throws ParseException { Calendar date = new GregorianCalendar(); date.setTimeZone(TimeZone.getTimeZone("Etc/UTC")); date.set(Calendar.HOUR_OF_DAY, 12); System.out.println("UTC: " + date.get(Calendar.HOUR_OF_DAY)); System.out.println("UTC: " + date.getTimeInMillis()); date.setTimeZone(TimeZone.getTimeZone("Europe/Copenhagen")); System.out.println("CPH: " + date.get(Calendar.HOUR_OF_DAY)); System.out.println("CPH: " + date.getTimeInMillis()); date.setTimeZone(TimeZone.getTimeZone("America/New_York")); System.out.println("NYC: " + date.get(Calendar.HOUR_OF_DAY)); System.out.println("NYC: " + date.getTimeInMillis()); } } Output It will print the following result. UTC: 12 UTC: 1511956997540 CPH: 13 CPH: 1511956997540 NYC: 7 NYC: 1511956997540 Available Time Zones Following example will showcase the timezones available with the system. IOTester.java import java.text.ParseException; import java.util.TimeZone; public class I18NTester { public static void main(String[] args) throws ParseException { String[] availableIDs = TimeZone.getAvailableIDs(); for(String id : availableIDs) { System.out.println("Timezone = " + id); } } } Output It will print the following result. Timezone = Africa/Abidjan Timezone = Africa/Accra ... Timezone = VST Print Page 22 In java, text is internally stored in Unicode format. If input/output is in differnt format then conversion is required. Conversion Following example will showcase conversion of a Unicode String to UTF8 byte[] and UTF8 byte[] to Unicode byte[]. IOTester.java import java.io.UnsupportedEncodingException; import java.nio.charset.Charset; import java.text.ParseException; public class I18NTester { public static void main(String[] args) throws ParseException, UnsupportedEncodingException { String unicodeString = "\u00C6\u00D8\u00C5" ; //convert Unicode to UTF8 format byte[] utf8Bytes = unicodeString.getBytes(Charset.forName("UTF-8")); printBytes(utf8Bytes, "UTF 8 Bytes"); //convert UTF8 format to Unicode String converted = new String(utf8Bytes, "UTF8"); byte[] unicodeBytes = converted.getBytes(); printBytes(unicodeBytes, "Unicode Bytes"); } public static void printBytes(byte[] array, String name) { for (int k = 0; k < array.length; k++) { System.out.println(name + "[" + k + "] = " + array[k]); } } } Output It will print the following result. UTF 8 Bytes[0] = -61 UTF 8 Bytes[1] = -122 UTF 8 Bytes[2] = -61 UTF 8 Bytes[3] = -104 UTF 8 Bytes[4] = -61 UTF 8 Bytes[5] = -123

Unicode Bytes[0] = -58 Unicode Bytes[1] = -40 Unicode Bytes[2] = -59 Print Page 23 Reader and Writer classes are character oriented stream classes. These can be used to read and convert Unicode characters. Conversion Following example will showcase conversion of a Unicode String to UTF8 byte[] and UTF8 byte[] to Unicode byte[] using Reader and Writer classes. IOTester.java import java.io.ByteArrayInputStream; import java.io.ByteArrayOutputStream; import java.io.IOException; import java.io.InputStream; import java.io.InputStreamReader; import java.io.OutputStreamWriter; import java.io.Reader; import java.io.Writer; import java.nio.charset.Charset; import java.text.ParseException; public class I18NTester { public static void main(String[] args) throws ParseException, IOException { String input = "This is a sample text" ; InputStream inputStream = new ByteArrayInputStream(input.getBytes()); //get the UTF-8 data Reader reader = new InputStreamReader(inputStream, Charset.forName("UTF-8")); //convert UTF-8 to Unicode int data = reader.read(); while(data != -1){ char theChar = (char) data; System.out.print(theChar); data = reader.read(); } reader.close(); System.out.println(); //Convert Unicode to UTF-8 Bytes ByteArrayOutputStream outputStream = new ByteArrayOutputStream(); Writer writer = new OutputStreamWriter(outputStream, Charset.forName("UTF-8")); writer.write(input); writer.close(); String out = new String(outputStream.toByteArray()); System.out.println(out); } } Output It will print the following result. This is a sample text This is a sample text Print Page 24Internalization or I18N refers to the capability of an Application to be able to serve users in multiple and different languages. Java has in-built support for Internalization. Java also provides formatting of numbers, currencies and adjustment of date and time accordingly.Java Internationalization helps to make a java application handle different languages, number formats, currencies, region specific time formatting.Localization or L10N is the adaptability of an application that is how an application adapts itself with a specific language, number formats, date and time settings etc.A java application should be internationalized in order to be able to localize itself.Culturally Dependent Information Following information items often varies with different time zones or cultures. Messages Date Time Number Currency Measurements Phone Numbers Postal Addresses GUI labels Internationalization Classes Java has a set of built-in classes which help in internationalization of an application. These classes are following: Sr.No.Class & Description 1LocaleRepresents a language along with country/region. 2ResourceBundleContains localized text or objects. 3NumberFormatUse to format numbers/currencies as per the locale. 4DecimalFormatUse to format numbers as per customized format and as per locale. 5DateFormatUse to format dates as per locale. 6SimpleDateFormatUse to format dates as per customized format and as per locale. Java Internalization - Environment Setup In this chapter, we will discuss on the different aspects of setting up a congenial environment for Java. Local Environment Setup If you are still willing to set up your environment for Java programming language, then this section guides you on how to download and set up Java on your machine. Following are the steps to set up the environment. Java SE is freely available from the link Download Java. You can download a version based on your operating system. Follow the instructions to download Java and run the .exe to install Java on your machine. Once you installed Java on your machine, you will need to set environment variables to point to correct installation directories − Setting Up the Path for Windows Assuming you have installed Java in c:\Program Files\java\jdk directory − Right-click on 'My Computer' and select 'Properties'. Click the 'Environment variables' button under the 'Advanced' tab. Now, alter the 'Path' variable so that it also contains the path to the Java executable. Example, if the path is currently set to 'C:\WINDOWS\SYSTEM32', then change your path to read 'C:\WINDOWS\SYSTEM32;c:\Program Files\java\jdk\bin'. Setting Up the Path for Linux, UNIX, Solaris, FreeBSD Environment variable PATH should be set to point to where the Java binaries have been installed. Refer to your shell documentation, if you have trouble doing this. Example, if you use bash as your shell, then you would add the following line to the end of your '.bashrc: export PATH = /path/to/java:$PATH' Popular Java Editors To write your Java programs, you will need a text editor. There are even more sophisticated IDEs available in the market. But for now, you can consider one of the following − Notepad − On Windows machine, you can use any simple text editor like Notepad (Recommended for this tutorial), TextPad. Netbeans − A Java IDE that is open-source and free which can be downloaded from . Eclipse − A Java IDE developed by the eclipse open-source community and can be downloaded from . What is Next? Next chapter will teach you how to write and run your first Java program and some of the important basic syntaxes in Java needed for developing applications. Java Internalization - Locale Class A Locale object represents a specific geographical/political/cultural region. Any operation requiring a Locale to perform its task is called locale-sensitive operation and uses the Locale to master information relative to the user. For example, displaying a number is a locale-sensitive operation. The number should be formatted as per the customs and conventions of the user's native country, region, or culture. Locale Contents A Locale object contains the following: Language - ISO 639 alpha-2 or alpha-3 language code, or registered language subtags up to 8 alpha letters. alpha-2 code must be used if both alpha-2 and alpha-3 code are present. The language field is case insensitive, but Locale always canonicalizes to lower case. Script - ISO 15924 alpha-4 script code. The script field is case insensitive, but Locale always canonicalizes to title case. Country (region) - ISO 3166 alpha-2 country code or UN M.49 numeric-3 area code. The country field is case insensitive, but Locale always canonicalizes to upper case. Variant - Any arbitrary value used to indicate a variation of a Locale. Where there are two or more variant values each indicating its own semantics, these values should be ordered by importance, with most important first, separated by underscore('_'). The variant field is case sensitive. Extensions - A map from single character keys to string values, indicating extensions apart from language identification. The extensions in Locale implement the semantics and syntax of BCP 47 extension subtags and private use subtags. The extensions are case insensitive, but Locale canonicalizes all extension keys and values to lower case. Java Internalization - Example - Locale Details In this example, we'll get default locale and print its details. Then create a locale for "fr" and print its details. I18NTester.java import java.util.Locale; public class I18NTester { public static void main(String[] args) { Locale locale =Locale.getDefault(); System.out.println("Default Locale Properties:"); System.out.println(locale.getDisplayCountry()); System.out.println(locale.getDisplayLanguage()); System.out.println(locale.getDisplayName()); System.out.println(locale.getISO3Country()); System.out.println(locale.getISO3Language()); System.out.println(locale.getLanguage()); System.out.println(locale.getCountry()); Locale frenchLocale = new Locale("fr","fr"); System.out.println("fr Locale Properties:"); System.out.println(frenchLocale.getDisplayCountry()); System.out.println(frenchLocale.getDisplayLanguage()); System.out.println(frenchLocale.getDisplayName()); System.out.println(frenchLocale.getISO3Country()); System.out.println(frenchLocale.getISO3Language()); System.out.println(frenchLocale.getLanguage()); System.out.println(frenchLocale.getCountry()); } } Output It will print the following result. Default Locale Properties: United States English English (United States) USA eng en US fr Locale Properties: France French French (France) FRA fra fr FR Java Internalization - Example - Display Language In this example, we'll get display language per locale passed as an argument. I18NTester.java import java.util.Locale; public class I18NTester { public static void main(String[] args) { Locale defaultLocale = Locale.getDefault(); Locale enLocale = new Locale("en", "US"); Locale frLocale = new Locale("fr", "FR"); Locale esLocale = new Locale("es", "ES"); System.out.println(defaultLocale.getDisplayLanguage(enLocale)); System.out.println(defaultLocale.getDisplayLanguage(frLocale)); System.out.println(defaultLocale.getDisplayLanguage(esLocale)); } } Output It will print the following result. English anglais inglés Java Internalization - ResourceBundle Class ResourceBundle class is used to store text and objects which are locale sensitive. Generally we use property files to store locale specific text and then represent them using ResourceBundle object. Following are the steps to use locale specific properties file in a java based application. Step 1: Create properties files. Suppose we need properties file for English locale. Then create a properties file name XXX_en_US.properties where XXX is the name of the file and en_US represents the locale for English(US). Messages_en_US.properties message=Welcome to TutorialsPoint.COM! Let's now create properties file for French locale. Then create a properties file name XXX_fr_FR.properties where XXX is the name of the file and fr_FR represents the locale for French(France). Messages_fr_FR.properties message=Bienvenue sur TutorialsPoint.COM! Here you can figure out that the key is same but the value is locale specific in both the properties file. Step 2: Create ResourceBundle object Create ResourceBundle object with properties file name and locale using following syntax. ResourceBundle bundle = ResourceBundle.getBundle("Messages", Locale.US); Step 3: Get the value from ResourceBundle object. Get the value from ResourceBundle object by passing the key. String value = bundle.getString("message"); Example Following example illustrate the use of ResourceBundle objects to display locale specific values from properties files. IOTester.java import java.util.Locale; import java.util.ResourceBundle; public class I18NTester { public static void main(String[] args) { ResourceBundle bundle = ResourceBundle.getBundle("Messages", Locale.US); System.out.println("Message in "+Locale.US +": "+bundle.getString("message")); bundle = ResourceBundle.getBundle("Messages", Locale.FRANCE); System.out.println("Message in "+Locale.FRANCE +": "+bundle.getString("message")); } } Output It will print the following result. Message in en_US: Welcome to TutorialsPoint.COM! Message in fr_FR: Bienvenue sur TutorialsPoint.COM! Notes for Naming Conventions Following are the naming conventions for the properties file. For properties file mapped to default locale, no prefix is mandatory. message_en_US.properties is equivalent to message.properties. For properties file mapped to locale, prefix can be attached in two ways. message_fr.properties is equivalent to message_fr_FR.properties. Java Internalization - NumberFormat Class The java.text.NumberFormat class is used for formatting numbers and currencies as per a specific Locale. Number formats varies from country to country. For example, In Denmark fractions of a number are separated from the integer part using a comma whereas in England they use a dot as separator. Example - Format Numbers In this example, we're formatting numbers based on US locale and Danish Locale. IOTester.java import java.text.NumberFormat; import java.util.Locale; public class I18NTester { public static void main(String[] args) { Locale enLocale = new Locale("en", "US"); Locale daLocale = new Locale("da", "DK"); NumberFormat numberFormat = NumberFormat.getInstance(daLocale); System.out.println(numberFormat.format(100.76)); numberFormat = NumberFormat.getInstance(enLocale); System.out.println(numberFormat.format(100.76)); } } Output It will print the following result. 100,76 100.76 Java Internalization - Format Currencies In this example, we're formatting currencies based on US locale and Danish Locale. IOTester.java import java.text.NumberFormat; import java.util.Locale; public class I18NTester { public static void main(String[] args) { Locale enLocale = new Locale("en", "US"); Locale daLocale = new Locale("da", "DK"); NumberFormat numberFormat = NumberFormat.getCurrencyInstance(daLocale); System.out.println(numberFormat.format(100.76)); numberFormat = NumberFormat.getCurrencyInstance(enLocale); System.out.println(numberFormat.format(100.76)); } } Output It will print the following result. kr 100,76 $100.76 Java Internalization - Format Percentages In this example, we're formatting numbers in percentage format. IOTester.java import java.text.NumberFormat; import java.util.Locale; public class I18NTester { public static void main(String[] args) { Locale enLocale = new Locale("en", "US"); NumberFormat numberFormat = NumberFormat.getPercentInstance(enLocale); System.out.println(numberFormat.format(0.76)); } } Output It will print the following result. 76% Java Internalization - Set Min/Max Precision In this example, we're setting min and max digits for both integer as well as fractional part of a number. IOTester.java import java.text.NumberFormat; import java.util.Locale; public class I18NTester { public static void main(String[] args) { Locale enLocale = new Locale("en", "US"); NumberFormat numberFormat = NumberFormat.getInstance(enLocale); numberFormat.setMinimumIntegerDigits(2); numberFormat.setMaximumIntegerDigits(3); numberFormat.setMinimumFractionDigits(2); numberFormat.setMaximumFractionDigits(3); System.out.println(numberFormat.format(12234.763443)); } } Output It will print the following result. 234.763 Java Internalization - Set Rounding Mode In this example, we're showcasing Rounding Mode. IOTester.java import java.math.RoundingMode; import java.text.NumberFormat; import java.util.Locale; public class I18NTester { public static void main(String[] args) { Locale enLocale = new Locale("en", "US"); NumberFormat numberFormat = NumberFormat.getInstance(enLocale); numberFormat.setMinimumFractionDigits(0); numberFormat.setMaximumFractionDigits(0); System.out.println(numberFormat.format(99.50)); numberFormat.setRoundingMode(RoundingMode.HALF_DOWN); System.out.println(numberFormat.format(99.50)); } } Output It will print the following result. 100 99 Java Internalization - Parsing Numbers In this example, we're showcasing parsing of number present in different locale. IOTester.java import java.text.NumberFormat; import java.text.ParseException; import java.util.Locale; public class I18NTester { public static void main(String[] args) throws ParseException { Locale enLocale = new Locale("en", "US"); Locale daLocale = new Locale("da", "DK"); NumberFormat numberFormat = NumberFormat.getInstance(daLocale); System.out.println(numberFormat.parse("100,76")); numberFormat = NumberFormat.getInstance(enLocale); System.out.println(numberFormat.parse("100,76")); } } Output It will print the following result. 100.76 10076 Java Internalization - DecimalFormat Class The java.text.DecimalFormat class is used for formatting numbers as per customized format and as per locale. Example - Format Numbers In this example, we're formatting numbers based on a given pattern. IOTester.java import java.text.DecimalFormat; public class I18NTester { public static void main(String[] args) { String pattern = "####,####.##"; double number = 123456789.123; DecimalFormat numberFormat = new DecimalFormat(pattern); System.out.println(number); System.out.println(numberFormat.format(number)); } } Output It will print the following result. 1.23456789123E8 1,2345,6789.12 Java Internalization - Format Patterns Followings is the use of characters in formatting patterns. Sr.No.Class & Description 10To display 0 if less digits are present. 2#To display digit ommitting leading zeroes. 3.Decimal separator. 4,Grouping separator. 5EMantissa and Exponent separator for exponential formats. 6;Format separator. 7-Negative number prefix. 8%Shows number as percentage after multiplying with 100. 9?Shows number as mille after multiplying with 1000. 10XTo mark character as number prefix/suffix. 11'To mark quote around special characters. In this example, we're formatting numbers based on different patterns. IOTester.java import java.text.DecimalFormat; public class I18NTester { public static void main(String[] args) { String pattern = "###.###"; double number = 123456789.123; DecimalFormat numberFormat = new DecimalFormat(pattern); System.out.println(number); //pattern ###.### System.out.println(numberFormat.format(number)); //pattern ###.# numberFormat.applyPattern("###.#"); System.out.println(numberFormat.format(number)); //pattern ###,###.## numberFormat.applyPattern("###,###.##"); System.out.println(numberFormat.format(number)); number = 9.34; //pattern 000.### numberFormat.applyPattern("000.##"); System.out.println(numberFormat.format(number)); } } Output It will print the following result. 1.23456789123E8 1,2345,6789.12 Java Internalization - Locale Specific DecimalFormat By default, DecimalFormat object is using the JVM's locale. We can change the default locale while creating the DecimalFormat object using NumberFormat class. In the example below, we'll use same pattern for two different locale and you can spot the difference in the output. IOTester.java import java.text.DecimalFormat; import java.text.NumberFormat; import java.util.Locale; public class I18NTester { public static void main(String[] args) { String pattern = "###.##"; double number = 123.45; Locale enlocale = new Locale("en", "US"); Locale dalocale = new Locale("da", "DK"); DecimalFormat decimalFormat = (DecimalFormat) NumberFormat.getNumberInstance(enlocale); decimalFormat.applyPattern(pattern); System.out.println(decimalFormat.format(number)); decimalFormat = (DecimalFormat) NumberFormat.getNumberInstance(dalocale); decimalFormat.applyPattern(pattern); System.out.println(decimalFormat.format(number)); } } Output It will print the following result. 123.45 123,45 Java Internalization - DecimalFormatSymbols Class Using DecimalFormatSymbols class, the default separator symbols, grouping separator symbols etc. can be changed. Following example is illustrating the same. IOTester.java import java.text.DecimalFormat; import java.text.DecimalFormatSymbols; public class I18NTester { public static void main(String[] args) { double number = 126473.4567; DecimalFormat decimalFormat = new DecimalFormat(pattern); System.out.println(decimalFormat.format(number)); DecimalFormatSymbols decimalFormatSymbols = new DecimalFormatSymbols(); decimalFormatSymbols.setDecimalSeparator(';'); decimalFormatSymbols.setGroupingSeparator(':'); decimalFormat = new DecimalFormat(pattern, decimalFormatSymbols); System.out.println(decimalFormat.format(number)); } } Output It will print the following result. 126,473.457 126:473;457 Java Internalization - Grouping Digits Using setGroupingSize() method of DecimalFormat, default grouping of numbers can be changed. Following example is illustrating the same. IOTester.java import java.text.DecimalFormat; public class I18NTester { public static void main(String[] args) { double number = 121223232473.4567; DecimalFormat decimalFormat = new DecimalFormat(); System.out.println(number); System.out.println(decimalFormat.format(number)); decimalFormat.setGroupingSize(4); System.out.println(decimalFormat.format(number)); } } Output It will print the following result. 1.21223232473456 7E11 121,223,232,473.457 1212,2323,2473.457 Java Internalization - DateFormat Class java.text.DateFormat class formats dates as per the locale. As different coutries use different formats to display dates. This class is extremely useful in dealing with dates in internalization of application. Following example show how to create and use DateFormat Class. IOTester.java import java.text.DateFormat; import java.util.Date; import java.util.Locale; public class I18NTester { public static void main(String[] args) { Locale locale = new Locale("da","DK"); DateFormat dateFormat = DateFormat.getDateInstance(); System.out.println(dateFormat.format(new Date())); dateFormat = DateFormat.getDateInstance(DateFormat.DEFAULT, locale); System.out.println(dateFormat.format(new Date())); } } Output It will print the following result. Nov 29, 2017 29-11-2017 Java Internalization - Formatting Dates DateFormat class provides various formats to format the date. Following is list of some of the formats. DateFormat.DEFAULT DateFormat.SHORT DateFormat.MEDIUM DateFormat.LONG DateFormat.FULL In following example we'll show how to use different formats. IOTester.java import java.text.DateFormat; import java.util.Date; public class I18NTester { public static void main(String[] args) { DateFormat dateFormat = DateFormat.getDateInstance(DateFormat.DEFAULT); System.out.println(dateFormat.format(new Date())); dateFormat = DateFormat.getDateInstance(DateFormat.SHORT); System.out.println(dateFormat.format(new Date())); dateFormat = DateFormat.getDateInstance(DateFormat.MEDIUM); System.out.println(dateFormat.format(new Date())); dateFormat = DateFormat.getDateInstance(DateFormat.LONG); System.out.println(dateFormat.format(new Date())); dateFormat = DateFormat.getDateInstance(DateFormat.FULL); System.out.println(dateFormat.format(new Date())); } } Output It will print the following result. Nov 29, 2017 11/29/17 Nov 29, 2017 November 29, 2017 Wednesday, November 29, 2017 Java Internalization - Formatting Time DateFormat class provides various formats to format the time. DateFormat.getTimeInstance() method is to be used. See the example below. In following example we'll show how to use different formats to format time. IOTester.java import java.text.DateFormat; import java.util.Date; public class I18NTester { public static void main(String[] args) { DateFormat dateFormat = DateFormat.getTimeInstance(DateFormat.DEFAULT); System.out.println(dateFormat.format(new Date())); dateFormat = DateFormat.getTimeInstance(DateFormat.SHORT); System.out.println(dateFormat.format(new Date())); dateFormat = DateFormat.getTimeInstance(DateFormat.MEDIUM); System.out.println(dateFormat.format(new Date())); dateFormat = DateFormat.getTimeInstance(DateFormat.LONG); System.out.println(dateFormat.format(new Date())); dateFormat = DateFormat.getTimeInstance(DateFormat.FULL); System.out.println(dateFormat.format(new Date())); } } Output It will print the following result. 4:11:21 PM 4:11 PM 4:11:21 PM 4:11:21 PM IST 4:11:21 PM IST Java Internalization - Formatting Date and Time DateFormat class provides various formats to format the date and time together. DateFormat.getDateTimeInstance() method is to be used. See the example below. In following example we'll show how to use different formats to format date and time. IOTester.java import java.text.DateFormat; import java.util.Date; public class I18NTester { public static void main(String[] args) { DateFormat dateFormat = DateFormat.getDateTimeInstance(DateFormat.DEFAULT, DateFormat.DEFAULT); System.out.println(dateFormat.format(new Date())); dateFormat = DateFormat.getDateTimeInstance(DateFormat.SHORT, DateFormat.SHORT); System.out.println(dateFormat.format(new Date())); dateFormat = DateFormat.getDateTimeInstance(DateFormat.MEDIUM, DateFormat.MEDIUM); System.out.println(dateFormat.format(new Date())); dateFormat = DateFormat.getDateTimeInstance(DateFormat.LONG, DateFormat.LONG); System.out.println(dateFormat.format(new Date())); dateFormat = DateFormat.getDateTimeInstance(DateFormat.FULL, DateFormat.FULL); System.out.println(dateFormat.format(new Date())); } } Output It will print the following result. Nov 29, 2017 4:16:13 PM 11/29/17 4:16 PM Nov 29, 2017 4:16:13 PM November 29, 2017 4:16:13 PM IST Wednesday, November 29, 2017 4:16:13 PM IST Java Internalization - SimpleDateFormat Class java.text.SimpleDateFormat class formats dates as per the given pattern. It is also used to parse dates from string where string contains date in mentioned format. See the following example of using SimpleDateFormat class. IOTester.java import java.text.ParseException; import java.text.SimpleDateFormat; import java.util.Date; public class I18NTester { public static void main(String[] args) throws ParseException { String pattern = "dd-MM-yyyy"; SimpleDateFormat simpleDateFormat = new SimpleDateFormat(pattern); Date date = new Date(); System.out.println(date); System.out.println(simpleDateFormat.format(date)); String dateText = "29-11-2017"; date = simpleDateFormat.parse(dateText); System.out.println(simpleDateFormat.format(date)); } } Output It will print the following result. Wed Nov 29 17:01:22 IST 2017 29-11-2017 29-11-2017 Java Internalization - Locale specific SimpleDateFormat Locale can be used to create locale specific formatting over a pattern in SimpleDateFormat class. See the following example of using locale specific SimpleDateFormat class. IOTester.java import java.text.ParseException; import java.text.SimpleDateFormat; import java.util.Date; import java.util.Locale; public class I18NTester { public static void main(String[] args) throws ParseException { Locale locale = new Locale("da", "DK"); String pattern = "EEEEE MMMMM yyyy"; SimpleDateFormat simpleDateFormat = new SimpleDateFormat(pattern); Date date = new Date(); System.out.println(date); System.out.println(simpleDateFormat.format(date)); simpleDateFormat = new SimpleDateFormat(pattern,locale); System.out.println(simpleDateFormat.format(date)); } } Output It will print the following result. Wed Nov 29 17:48:14 IST 2017 Wednesday November 2017 onsdag november 2017 Java Internalization - DecimalFormatSymbols Class Using DecimalFormatSymbols class, the default separator symbols, grouping separator symbols etc. can be changed. Following example is illustrating the same. IOTester.java import java.text.DecimalFormat; import java.text.DecimalFormatSymbols; public class I18NTester { public static void main(String[] args) { String pattern = "#,###.###"; double number = 126473.4567; DecimalFormat decimalFormat = new DecimalFormat(pattern); System.out.println(decimalFormat.format(number)); DecimalFormatSymbols decimalFormatSymbols = new DecimalFormatSymbols(); decimalFormatSymbols.setDecimalSeparator(';'); decimalFormatSymbols.setGroupingSeparator(':'); decimalFormat = new DecimalFormat(pattern, decimalFormatSymbols); System.out.println(decimalFormat.format(number)); } } Output It will print the following result. 126,473.457 126:473;457 Java Internalization - Date Format Patterns Followings is the use of characters in date formatting patterns. Sr.No.Class & Description 1GTo display Era. 2yTo display Year. Valid values yy, yyyy. 3MTo display Month. Valid values MM, MMM or MMMMM. 4dTo display day of month. Valid values d, dd. 5hTo display hour of day (1-12 AM/PM). Valid value hh. 6HTo display hour of day (0-23). Valid value HH. 7mTo display minute of hour (0-59). Valid value mm. 8sTo display second of minute (0-59). Valid value ss. 9STo display milliseconds of minute (0-999). Valid value SSS. 10ETo display Day in week (e.g Monday, Tuesday etc.) 11DTo display Day in year (1-366). 12FTo display Day of week in month (e.g. 1st Thursday of December). 13wTo display Week in year (1-53). 14WTo display Week in month (0-5) 15aTo display AM / PM 16kTo display Hour in day (1-24). 17KTo display Hour in day, AM / PM (0-11). 18zTo display Time Zone. In this example, we're formatting dates based on different patterns. IOTester.java import java.text.ParseException; import java.text.SimpleDateFormat; import java.util.Date; public class I18NTester { public static void main(String[] args) throws ParseException { String pattern = "dd-MM-yy"; SimpleDateFormat simpleDateFormat = new SimpleDateFormat(pattern); Date date = new Date(); System.out.println(simpleDateFormat.format(date)); pattern = "MM-dd-yyyy"; simpleDateFormat = new SimpleDateFormat(pattern); System.out.println(simpleDateFormat.format(date)); pattern = "yyyy-MM-dd HH:mm:ss"; simpleDateFormat = new SimpleDateFormat(pattern); System.out.println(simpleDateFormat.format(date)); pattern = "EEEEE MMMMM yyyy HH:mm:ss.SSSZ"; simpleDateFormat = new SimpleDateFormat(pattern); System.out.println(simpleDateFormat.format(date)); } } Output It will print the following result. 29-11-17 11-29-2017 2017-11-29 18:47:42 Wednesday November 2017 18:47:42.787+0530 Java Internalization - UTC UTC stands for Co-ordinated Universal Time. It is time standard and is commonly used across the world. All time zones are computed comparatively with UTC as offset. For example, time in Copenhagen, Denmark is UTC + 1 means UTC time plus one hour. It is independent of Day light savings and should be used to store date and time in databases. Conversion of time zones Following example will showcase conversion of various time zones. We'll print hour of the day and time in milliseconds. First will vary and second will remain same. IOTester.java import java.text.ParseException; import java.util.Calendar; import java.util.GregorianCalendar; import java.util.TimeZone; public class I18NTester { public static void main(String[] args) throws ParseException { Calendar date = new GregorianCalendar(); date.setTimeZone(TimeZone.getTimeZone("Etc/UTC")); date.set(Calendar.HOUR_OF_DAY, 12); System.out.println("UTC: " + date.get(Calendar.HOUR_OF_DAY)); System.out.println("UTC: " + date.getTimeInMillis()); date.setTimeZone(TimeZone.getTimeZone("Europe/Copenhagen")); System.out.println("CPH: " + date.get(Calendar.HOUR_OF_DAY)); System.out.println("CPH: " + date.getTimeInMillis()); date.setTimeZone(TimeZone.getTimeZone("America/New_York")); System.out.println("NYC: " + date.get(Calendar.HOUR_OF_DAY)); System.out.println("NYC: " + date.getTimeInMillis()); } } Output It will print the following result. UTC: 12 UTC: 1511956997540 CPH: 13 CPH: 1511956997540 NYC: 7 NYC: 1511956997540 Available Time Zones Following example will showcase the time zones available with the system. IOTester.java import java.text.ParseException; import java.util.TimeZone; public class I18NTester { public static void main(String[] args) throws ParseException { String[] availableIDs = TimeZone.getAvailableIDs(); for(String id : availableIDs) { System.out.println("Timezone = " + id); } } } Output It will print the following result. Timezone = Africa/Abidjan Timezone = Africa/Accra ... Timezone = VST Java Internalization - Unicode Conversion from/to String In java, text is internally stored in Unicode format. If input/output is in different format then conversion is required. Conversion Following example will showcase conversion of a Unicode String to UTF8 byte[] and UTF8 byte[] to Unicode byte[]. IOTester.java import java.io.UnsupportedEncodingException; import java.nio.charset.Charset; import java.text.ParseException; public class I18NTester { public static void main(String[] args) throws ParseException, UnsupportedEncodingException { String unicodeString = "\u00C6\u00D8\u00C5" ; //convert Unicode to UTF8 format byte[] utf8Bytes = unicodeString.getBytes(Charset.forName("UTF-8")); printBytes(utf8Bytes, "UTF 8 Bytes"); //convert UTF8 format to Unicode String converted = new String(utf8Bytes, "UTF8"); byte[] unicodeBytes = converted.getBytes(); printBytes(unicodeBytes, "Unicode Bytes"); } public static void printBytes(byte[] array, String name) { for (int k = 0; k < array.length; k++) { System.out.println(name + "[" + k + "] = " + array[k]); } } } Output It will print the following result. UTF 8 Bytes[0] = -61 UTF 8 Bytes[1] = -122 UTF 8 Bytes[2] = -61 UTF 8 Bytes[3] = -104 UTF 8 Bytes[4] = -61 UTF 8 Bytes[5] = -123 Unicode Bytes[0] = -58 Unicode Bytes[1] = -40 Unicode Bytes[2] = -59 Java Internalization - Unicode Conversion from/to Reader/Writer Reader and Writer classes are character oriented stream classes. These can be used to read and convert Unicode characters. Conversion Following example will showcase conversion of a Unicode String to UTF8 byte[] and UTF8 byte[] to Unicode byte[] using Reader and Writer classes. IOTester.java import java.io.ByteArrayInputStream; import java.io.ByteArrayOutputStream; import java.io.IOException; import java.io.InputStream; import java.io.InputStreamReader; import java.io.OutputStreamWriter; import java.io.Reader; import java.io.Writer; import java.nio.charset.Charset; import java.text.ParseException; public class I18NTester { public static void main(String[] args) throws ParseException, IOException { String input = "This is a sample text" ; InputStream inputStream = new ByteArrayInputStream(input.getBytes()); //get the UTF-8 data Reader reader = new InputStreamReader(inputStream, Charset.forName("UTF-8")); //convert UTF-8 to Unicode int data = reader.read(); while(data != -1){ char theChar = (char) data; System.out.print(theChar); data = reader.read(); } reader.close(); System.out.println(); //Convert Unicode to UTF-8 Bytes ByteArrayOutputStream outputStream = new ByteArrayOutputStream(); Writer writer = new OutputStreamWriter(outputStream, Charset.forName("UTF-8")); writer.write(input); writer.close(); String out = new String(outputStream.toByteArray()); System.out.println(out); } } Output It will print the following result. This is a sample text This is a sample text Print Page 25 The following resources contain additional information on Java. Please use them to get more in-depth knowledge on this topic. Useful Links on Java Useful Books on Java To enlist your site on this page, please drop an email to contact@tutorialspoint.com Print