Object-oriented Programming in Java

Session: 12

Internationalization and Localization

Objectives

- Describe internationalization
- Describe localization
- Describe the Unicode character encoding
- Explain the internationalization process
- Define the internationalization elements

Introduction

- With the advent of the Internet, globalization of software products has become an imminent requirement.
- When the input and output operations of an application is made specific to different locations and user preferences, users around the world can use it with ease.
- This can be achieved using the processes called internationalization and localization.
- The adaptation is done with extreme ease because there are no coding changes required.

Internationalization

- An application is accessible to the international market when the input and output operations are specific to different locations and user preferences.
- The process of designing such an application is called internalization.
- Internationalization is commonly referred to as i18n.
- 18 in i18n refer to the 18 characters between the first letter i and the last letter n.
- Java includes a built-in support to internationalize applications.

Localization

- Localization deals with a specific region or language.
- In localization, an application is adapted to a specific region or language.
- Locale-specific components are added and text is translated in the localization process.
- Localization is commonly referred as l10n.
- 10 in l10n refers to the 10 letters between the first letter l and the last letter n.
- Primarily, in localization, the user interface elements and documentation are translated.

Benefits of I18N and L10N

- **No Recompilation of New Languages**: New languages are supported without recompilation.
- **Same Executable File**: The localized data needs to be incorporated in the application and the same executable runs worldwide.
- **Dynamic Retrieval of Textual Elements**: Textual elements are not hardcoded in the program.
- **Conformation to the End User's Region and Language**: Region specific information such as currencies, numbers, date and time follow the specified format of the end user's region and language.
- **Easy Localization**: The application can be easily and quickly localized.

ISO Codes [1-6]

- In the internationalization and localization process, a language is represented using the alpha-2 or alpha-3 ISO 639 code, such as `es` that represents Spanish.
- The code is always represented in lower case letters.
- A country is represented using the ISO 3166 alpha-2 code or UN M.49 numeric area code. It is always represented in upper case. For example, `ES` represents Spain.
- If an application is well internationalized, it is easy to localize it for a character encoding scheme.

ISO Codes [2-6]

The following Code Snippet illustrates the use of Japanese language for displaying a message:

Code Snippet

```
import java.util.Locale;
import java.util.ResourceBundle;
public class InternationalApplication {
/**
* @param args the command line arguments
*/
public static void main(String[] args) {
  // TODO code application logic here
  String language;
  String country;
```

ISO Codes [3-6]

```
if (args.length != 2) {
language = new String(“en”);
country = new String(“US”);
  } else {
language = new String(args[0]);
country = new String(args[1]);
  }

  Locale currentLocale;
  ResourceBundle messages;

currentLocale = new Locale(language, country);
```

ISO Codes [4-6]

```
messages =
ResourceBundle.getBundle("internationalApplication/
MessagesBundle", currentLocale);
System.out.println(messages.getString("greetings"));
System.out.println(messages.getString("inquiry"));
System.out.println(messages.getString("farewell"));
  }
}
```

- In the code, two arguments are accepted to represent country and language.
- Depending on the arguments passed during execution of the program the message corresponding to that country and language is displayed.
- For this, five properties file have been created.

ISO Codes [5-6]

Following are the content of five properties files:

```
MessagesBundle.properties
greetings = Hello.
farewell = Goodbye.
inquiry = How are you?
```

```
MessagesBundle_de_DE.properties
greetings = Hallo.
farewell = Tschüß.
inquiry = Wiegeht’s?
```

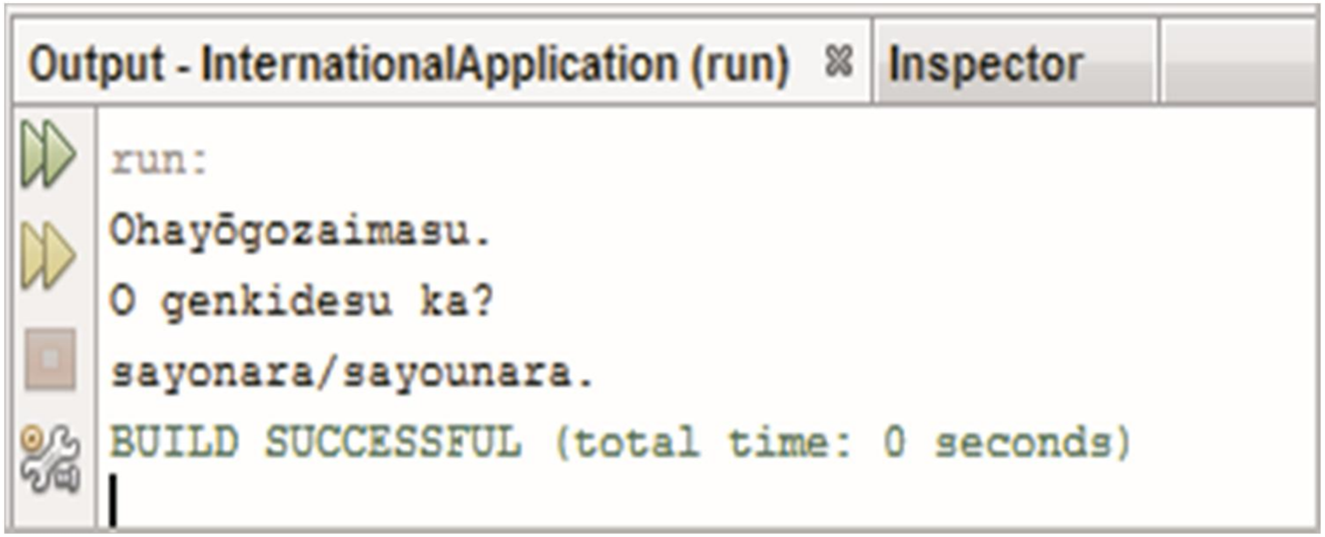
```
MessagesBundle_en_US.properties
greetings = Hello.
farewell = Goodbye.
inquiry = How are you?
```

```
MessagesBundle_fr_FR.properties
greetings = Bonjour.
farewell = Au revoir.
inquiry = Comment allez-vous?
```

```
MessagesBundle_ja_JP.properties
greetings = Ohayōgozaimasu.
farewell = sayonara/sayounara.
inquiry = O genkidesuka?
```

ISO Codes [6-6]

The following displays the output in Japanese language:



```
run:
Ohayōgozaimasu.
O genkidesu ka?
sayonara/sayounara.
BUILD SUCCESSFUL (total time: 0 seconds)
```

Unicode [1-2]

- Unicode provides a unique number for every character irrespective of platform, program, or language.
- The Unicode standard was first designed using 16 bits to encode characters.
- 16-bit encoding supports 216 (65,536) characters where in the hexadecimal they ranged from 0x0000 to 0xFFFF.
- This was insufficient to define all characters in world languages.
- So, the Unicode standard was extended to 0x10FFFF hexadecimal values.
- This new standard supports over one million characters.

Unicode [2-2]

The following list defines the terminologies used in the Unicode character encoding:

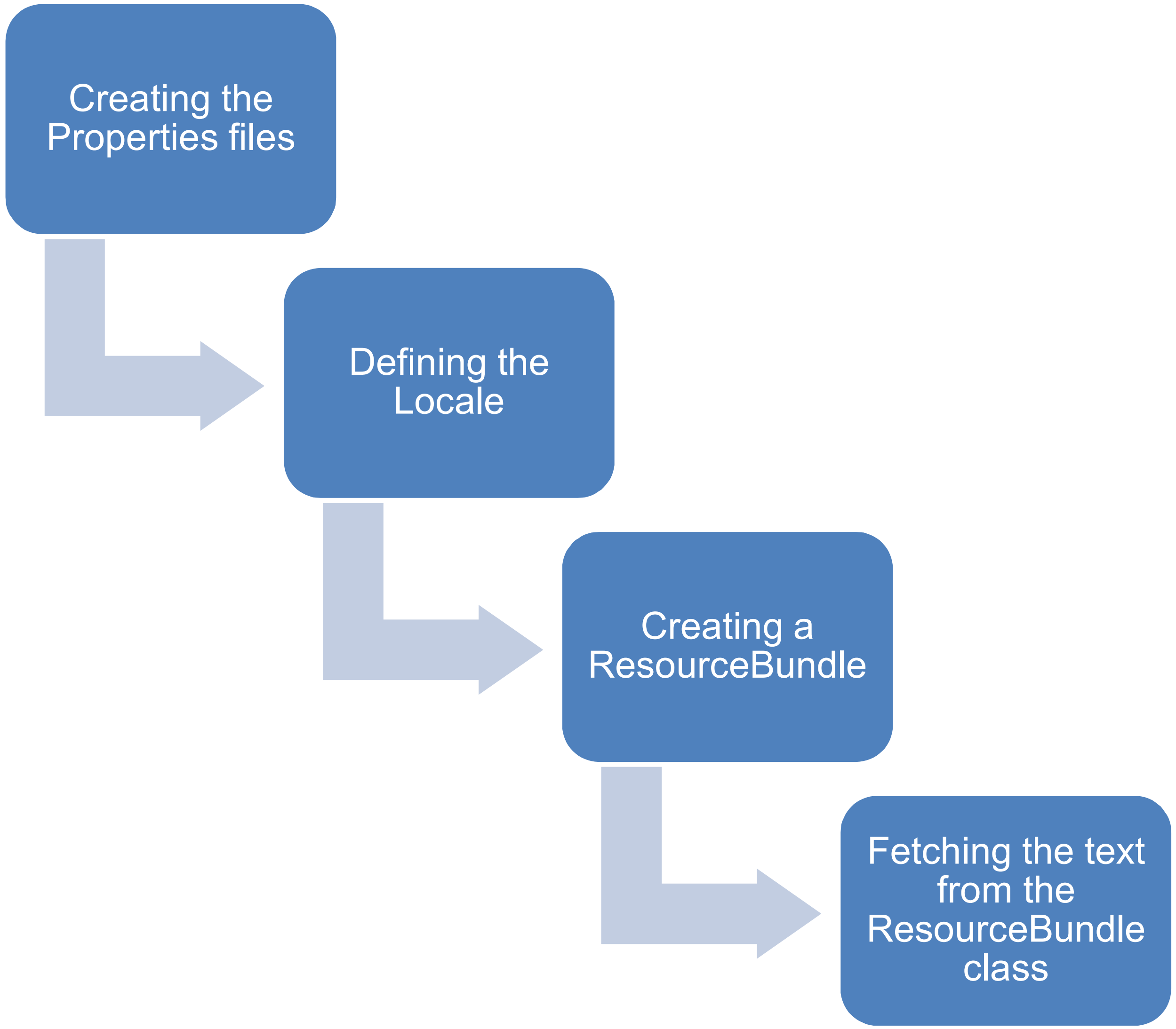
- **Character**: This represents the minimal unit of text that has semantic value.
- **Character Set**: This represents set of characters that can be used by many languages.
- **Coded Character**: This is a character set. Each character in the set is assigned a unique number.
- **Code Point**: This is the value that is used in a coded character set. A code point is a 32-bit int data type. Here, the upper 11 bits are 0 and the lower 21 bits represent a valid code point value.
- **Code Unit**: This is a 16-bit char value.
- **Supplementary Characters**: These are the characters that range from U+10000 to U+10FFFF.
  - Supplementary characters are represented by a pair of code point values called surrogates that support the characters without changing the char primitive data type.
  - Surrogates also provide compatibility with earlier Java programs.
- **Basic Multilingual Plane (BMP):** These are the set of characters from U+0000 to U+FFFF.

Unicode Character Encoding

- Consider the following points for Unicode character encoding:
  - The hexadecimal value is prefixed with the string U+.
  - The valid code point range for the Unicode standard is U+0000 to U+10FFFF.
- The following table shows code point values for certain characters:

Character	Unicode Code Point	Glyph
Latin A	U+0041	A
Latin sharp S	U+00DF	B

Internationalization Process


Creating the Properties files
Defining the Locale
Creating a ResourceBundle
Fetching the text from the ResourceBundle class

Creating the Properties Files [1-2]

- A properties file stores information about the characteristics of a program or environment.
- A properties file is in plain-text format.
- It can be created with any text editor.
- The following example shows the lines included in the default properties file, `MessagesBundle.properties` that needs to be translated:

```
greetings = Hello
farewell = Goodbye
inquiry = How are you?
```

- Since the messages are in the properties file, it can be translated into various languages.
- No changes to the source code are required.

Creating the Properties Files [2-2]

- To translate the message in French, the French translator creates a properties file called `MessagesBundle_fr_FR.properties` which contains the following lines:

```
greetings = Bonjour.
farewell = Au revoir.
inquiry = Comment allez-vous?
```

- Notice that the values to the right side of the equal sign are translated.
- The keys on the left side are not changed.
- These keys must not change because they are referenced when the program fetches the translated text.
- The name of the properties file is important.

Defining the Locale [1-3]

- The `Locale` object identifies a particular language and country.
- A `Locale` is simply an identifier for a particular combination of language and region.
- A `java.util.Locale` class object represents a specific geographical, political, or cultural region.
- Any operation that requires a locale to perform its task is said to be locale-sensitive.
- These operations use the `Locale` object to tailor information for the user.

Defining the Locale [2-3]

- A `Locale` object is created using the following constructors:
  - `public Locale(String language, String country)`
  - `public Locale(String language)`
- `Locale` objects are only identifiers.
- After defining a `Locale`, the next step is to pass it to other objects that perform useful tasks, such as formatting dates and numbers.
- These objects are locale-sensitive because their behavior varies according to `Locale`.
- A `ResourceBundle` is an example of a locale-sensitive object.

Defining the Locale [3-3]

Certain important methods of the `Locale` class

- `public static Locale getDefault()`
- `Public final String getDisplayCountry()`
- `public final String getDisplayLanguage()`

Creating a ResourceBundle [1-4]

- `ResourceBundle` objects contain locale-specific objects.
- These objects are used to isolate locale-sensitive data, such as translatable text.
- The `ResourceBundle` class is used to retrieve locale-specific information from the properties file.
- This information allows a user to write applications that can be:
  - Localized or translated into different languages.
  - Handled for multiple locales at the same time.
  - Supported for more locales later.
- The `ResourceBundle` class has a static and final method called `getBundle()` that helps to retrieve a `ResourceBundle` instance.

Creating a ResourceBundle [2-4]

- The `ResourceBundle getBundle(String, Locale)` method helps to retrieve locale-specific information from a given properties file and takes two arguments, a `String` and an object of `Locale` class.
- The object of `ResourceBundle` class is initialized with a valid language and country matching the available properties file.
- The following Code Snippet displays how to create a `ResourceBundle`:

```
messages = ResourceBundle.getBundle("MessagesBundle",
currentLocale);
```

- The arguments passed to the `getBundle()` method identify the properties file that will be accessed.

Creating a ResourceBundle [3-4]

- The first argument, `MessagesBundle`, refers to the following family of properties files:
  - MessagesBundle _ en _ US.properties
  - MessagesBundle _ fr _ FR.properties
  - MessagesBundle _ de _ DE.properties
  - MessagesBundle _ ja _ JP.properties
- The `currentLocale`, which is the second argument of `getBundle()` method, specifies the selected `MessagesBundle` files.
- When the `Locale` was created, the language code and the country code were passed to its constructor.

Creating a ResourceBundle [4-4]

- To retrieve the locale-specific data from the properties file, the `ResourceBundle` class object should first be created.
- Next, the following methods should be invoked:
  - `public final String getString(String key)`
  - `public abstract Enumeration<String>getKeys()`

Fetching the Text from the ResourceBundle Class

- The properties files contain key-value pairs.
- The values consist of the translated text that the program will display.
- The keys are specified when fetching the translated messages from the `ResourceBundle` with the `getString()` method.
- The Code Snippet illustrates how to retrieve the value from the key-value pair using the `getString()` method:

```
String msg1 = messages.getString(“greetings”);
```

- The sample program uses the key greetings because it reflects the content of the message.
- The key is hardcoded in the program and it must be present in the properties files.

Internationalization Elements

- Component Captions
- Numbers, Currencies, and Percentages
- Date and Times
- Messages

Component Captions

- These refer to the GUI component captions such as text, date, and numerals.
- These GUI component captions should be localized because their usage vary with language, culture, and region.
- Formatting the captions of the GUI components ensures that the look and feel of the application is in a locale-sensitive manner.
- The code that displays the GUI is locale-independent. There is no need to write formatting routines for specific locales.

Numbers, Currencies, and Percentages [1-8]

- The format of numbers, currencies, and percentages vary with culture, region, and language.
- Hence, it is necessary to format them before they are displayed.
- For example, the number 12345678 should be formatted and displayed as 12,345,678 in the US and 12.345.678 in Germany.
- Similarly, the currency symbols and methods of displaying the percentage factor also vary with region and language.
- Formatting is required to make an internationalized application, independent of local conventions with regards to decimal-point, thousands-separators, and percentage representation.
- The `NumberFormat` class is used to create locale-specific formats for numbers, currencies, and percentages.

Numbers, Currencies, and Percentages [2-8]

The following Code Snippet shows how to create locale-specific format of number for the country Japan:

Code Snippet

```
importjava.text.NumberFormat;
import java.util.Locale;
import java.util.ResourceBundle;
public class InternationalApplication {


static public void printValue(Locale currentLocale) {
  Integer value = new Integer(123456);
  Double amt = new Double(345987.246);
NumberFormatnumFormatObj;
  String valueDisplay;
  String amtDisplay;
```

Numbers, Currencies, and Percentages [3-8]

```
numFormatObj = NumberFormat.getNumberInstance(currentLocale);
valueDisplay = numFormatObj.format(value);
amtDisplay = numFormatObj.format(amt);
System.out.println(valueDisplay + " " + currentLocale.toString());
System.out.println(amtDisplay + " " + currentLocale.toString());
 }

 /**
 * @param args the command line arguments
 */
public static void main(String[] args) {
   // TODO code application logic here
   String language;
   String country;
```

Numbers, Currencies, and Percentages [4-8]

```
if (args.length != 2) {
language = new String(“en”);
country = new String(“US”);
  } else {
language = new String(args[0]);
country = new String(args[1]);
  }
    Locale currentLocale;
    ResourceBundle messages;
currentLocale = new Locale(language, country);
messages =
ResourceBundle.getBundle(“internationalApplication/MessagesBu
ndle”, currentLocale);
```

Numbers, Currencies, and Percentages [5-8]

```
System.out.println(messages.getString(“greetings”));
System.out.println(messages.getString(“inquiry”));
System.out.println(messages.getString(“farewell”));
printValue(currentLocale);
  }
}
```

- The `NumberFormat` class has a static method `getCurrencyInstance()` which takes an instance of `Locale` class as an argument.
- The `getCurrencyInstance()` method returns an instance of a `NumberFormat` class initialized for the specified locale.

Numbers, Currencies, and Percentages [6-8]

- The syntax for some of the methods to format currencies are as follows:
  - `public final String format(double currency`
  - `public static final NumberFormat getCurrencyInstance()`
  - `public static NumberFormat getCurrencyInstance(Locale inLocale)`
- The following Code Snippet shows how to create locale-specific format of currency for the country, France:

Code Snippet

```
NumberFormat currencyFormatter;
String strCurrency;
// Creates a Locale object with language as French and
country
// as France
Locale locale = new Locale("fr", "FR");
// Creates an object of a wrapper class Double
Double currency = new Double(123456.78);
```

Numbers, Currencies, and Percentages [7-8]

```
// Retrieves the CurrencyFormatterinstance
currencyFormatter = NumberFormat.
getCurrencyInstance(locale);
// Formats the currency
strCurrency = currencyFormatter.format(currency);
messages =
ResourceBundle.getBundle("internationalApplication/Mes
sagesBundle", currentLocale);
```

- The `getPercentInstance()` method returns an instance of the `NumberFormat` class initialized to the specified locale.
- The syntax for some of the methods to format percentages are as follows:
  - `public final String format(double percent)`
  - `public static final NumberFormat getPercentInstance()`
  - `public static NumberFormat getPercentInstance(Locale inLocale)`

Numbers, Currencies, and Percentages [8-8]

The following Code Snippet shows how to create locale-specific format of percentages for the country, France:

Code Snippet

```
NumberFormatpercentFormatter;
String strPercent;
// Creates a Localeobject with language as French and country
// as France
Locale locale = new Locale("fr", "FR");
// Creates an object of a wrapper class Double
Double percent = new Double(123456.78);
// Retrieves the percentFormatter instance
percentFormatter = NumberFormat. getPercentInstance(locale);
// Formats the percent figure
strPercent = percentFormatter.format(percent);
```

Date and Times [1-4]

- The date and time format should conform to the conventions of the end user's locale.
- The date and time format varies with culture, region, and language.
- Hence, it is necessary to format them before they are displayed.
- In German, the date can be represented as 20.04.07, whereas in US it is represented as 04/20/07.
- Java provides the `java.text.DateFormat` and `java.text.SimpleDateFormat` class to format date and time.
- The `DateFormat` class is used to create locale-specific formats for date.

Date and Times [2-4]

- Next, the `format()` method of the `NumberFormat` class is also invoked.
- The date to be formatted is passed as an argument.
- The `DateFormat getDateInstance(style, locale)` method returns an instance of the class `DateFormat` for the specified style and locale.
- Consider the following syntax:

Code Snippet

```
public static final DateFormatgetDateInstance(int
style, Locale locale)
```

  - `style` is an integer and specifies the style of the date.
  - Valid values are `DateFormat.LONG`, `DateFormat.SHORT`, and `DateFormat.MEDIUM`.
  - `locale` is an object of the `Locale` class, and specifies the format of the locale.

Date and Times [3-4]

- DateFormatobject includes a number of constants such as:
  - **SHORT**: Is completely numeric such as 12.13.45 or 4 :30 pm
  - **MEDIUM**: Is longer, such as Dec 25, 1945
  - **LONG**: Is longer such as December 25, 1945
  - **FULL**: Represents a complete specification such as Tuesday, April 12, 1945 AD
- The following Code Snippet demonstrates how to retrieve a DateFormat object and display the date in Japanese format:

```
importjava.text.DateFormat;
import java.util.Date;
import java.util.Locale;

public class DateInternationalApplication {

public static void main(String[] args) {
  Date today;
  String strDate;
```

Date and Times [4-4]

```
DateFormatdateFormatter;
  Locale locale = new Locale(“ja”, “JP”);
dateFormatter =
DateFormat.getDateInstance(DateFormat.MEDIUM, locale);
today = new Date();
strDate = dateFormatter.format(today);
System.out.println(strDate);
 }
}
```

Messages [1-4]

- Displaying messages such as status and error messages are an integral part of any software.
- The `MessageFormat` class helps create a compound message.
- To use the `MessageFormat` class, perform the following steps:
    1. Identify the variables in the message.
    2. Create a template.
    3. Create an Object array for variable arguments.
    4. Create a `MessageFormat` instance and set the desired locale.
    5. Apply and format the pattern.
- The `MessageFormat` class has a method `applyPattern()` to apply the pattern to the `MessageFormat` instance.
- Once the pattern is applied to the `MessageFormat` instance, invoke the `format()` method.

Messages [2-4]

The following Code Snippet when executed will display the message in Danish using `MessageFormater` class.

```
import java.text.MessageFormat;
import java.util.Date;
import java.util.Locale;
import java.util.ResourceBundle;
public class MessageFormatterInternationalApplication
{


  /**
  * @param args the command line arguments
  */
 public static void main(String[] args) {
   // TODO code application logic here
   String template = “At {2,time,short} on
{2,date,long}, we detected {1,number,integer} virus on
the disk {0}”;
```

Messages [3-4]

```
 MessageFormat formatter = new MessageFormat("");
 String language;
 String country;

if (args.length != 2) {
  language = new String("en");
  country = new String("US");
} else {
  language = new String(args[0]);
  country = new String(args[1]);
}

Locale currentLocale;
currentLocale = new Locale(language, country);
formatter.setLocale(currentLocale);
```

Messages [4-4]

```
  ResourceBundle messages = 
ResourceBundle.getBundle("messageformatterin
ternationalapplication/MessageFormatBundle",
currentLocale);

  Object[] messageArguments = 
{messages.getString("disk"), new Integer(7), new 
Date()};

  formatter.applyPattern(messages.getString("template"
));

  String output = formatter.format(messageArguments);

  System.out.println(output);


 }

}
```

Summary

- In the internationalization process, the input and output operations of an application are specific to different locations and user preferences. Internationalization is commonly referred to as i18n.
- In localization, an application is adapted to a specific region or language.
- A locale represents a particular language and country.
- Localization is commonly referred to as l10n.
- A language is represented using the alpha-2 or alpha-3 ISO 639 code in the internationalization and localization process.
- Unicode is a computing industry standard to uniquely encode characters for various languages in the world using hexadecimal values.
- No recompilation is required for localization.