Documento Explicativo: Formateo de Números, Fechas y Horas en Java

1. Introducción General

Cuando programamos, necesitamos mostrar datos de manera legible y consistente para los usuarios finales. Por ejemplo, mostrar números con dos decimales o fechas con un formato específico (“20-10-2022”). Java nos ofrece varias clases y métodos que permiten adaptar esta presentación de datos según nuestras necesidades. Este documento explica los conceptos esenciales para que puedas aplicar formatos a valores numéricos, fechas y horas de forma sencilla en Java.

---

2. Formateo de Números

2.1 NumberFormat y DecimalFormat

Para controlar la forma en que se muestran los números en Java, disponemos de la clase abstracta `NumberFormat` y, especialmente, de la clase concreta `DecimalFormat`. Por ejemplo, si queremos mostrar un número con comas como separadores de miles y un decimal, podemos usar un patrón que indique dónde irán las comas y cuántos decimales deseamos.

- **Patrones comunes**:
    - `#`: Indica que si no hay dígito, esa posición se omite.
    - `0`: Indica que si no hay dígito, se coloca un cero en esa posición.

Ejemplo Práctico

```
double numero = 1234.567;
NumberFormat formato1 = new DecimalFormat("###,###,###.0");
System.out.println(formato1.format(numero)); // Resultado: 1,234.6
```

- En este ejemplo, `###,###,###.0` redondea el número a un decimal (`.0`) y muestra comas en cada grupo de tres dígitos.

Si quisiéramos mostrar siempre tres ceros a la izquierda y cinco decimales, usaríamos:

```
NumberFormat formato2 = new DecimalFormat("000,000,000.00000");
System.out.println(formato2.format(numero)); // Resultado: 000,001,234.56700
```

Con ello, garantizamos el número de dígitos tanto antes como después del decimal.

---

3. Formateo de Fechas y Horas

3.1 Clases Principales

Para manejar fechas y horas, Java utiliza, entre otras, las siguientes clases:

- **`LocalDate`**: Representa una fecha (día, mes y año) sin hora.
- **`LocalTime`**: Representa una hora (horas, minutos y segundos) sin fecha.
- **`LocalDateTime`**: Combina fecha y hora, pero sin información de zona horaria.
- **`ZonedDateTime`**: Incluye fecha, hora y zona horaria.

3.2 DateTimeFormatter

Para mostrar estos valores en un formato específico, usamos la clase `DateTimeFormatter`. Java trae formatos predefinidos, como `ISO_LOCAL_DATE` o `ISO_LOCAL_TIME`, pero también podemos crear formatos personalizados mediante el método `ofPattern(...)`.

Formatos Predefinidos
LocalDate fecha = LocalDate.of(2022, 10, 20);
System.out.println(fecha.format(DateTimeFormatter.ISO_LOCAL_DATE)); // 2022-10-20

Si intentamos formatear un `LocalDate` con un patrón de hora (`ISO_LOCAL_TIME`), obtendremos un error en tiempo de ejecución porque no existe información de hora.

3.3 Uso de Patrones Personalizados

Para personalizar aún más la forma en que se muestra una fecha/hora, podemos crear patrones. Por ejemplo:

LocalDateTime fechaHora = LocalDateTime.of(2022, 10, 20, 11, 12, 34);
DateTimeFormatter formatoPersonalizado = DateTimeFormatter.ofPattern("MMMM dd, yyyy 'a las' hh:mm");
System.out.println(fechaHora.format(formatoPersonalizado));
// Posible salida: "octubre 20, 2022 a las 11:12"

- **`M`** representa el mes.
- **`d`** representa el día.
- **`y`** representa el año.
- **`h`** representa la hora en formato 12 horas.

- `m` representa los minutos.

Observa que se usan comillas simples `'a las'` para insertar texto libre dentro del patrón. Esto evita que Java interprete ese texto como parte del patrón.

3.4 Compatibilidad de Símbolos

Cada símbolo tiene sentido para ciertos tipos de datos:

- **Mes (`M`)** no existe en `LocalTime` (solo tiene hora).
- **Hora (`h`)** no existe en `LocalDate` (solo tiene fecha).

Usar un símbolo que no coincide con el tipo de dato generará un error en tiempo de ejecución.

---

4. Consejos Prácticos y Ejemplificaciones

1. **Cuidar el Rango de Valores**
   Si formateas un número muy grande o con demasiados decimales, piensa en el rendimiento y en si realmente necesitas mostrarlos todos.

2. **Respetar el Tipo de Objeto**
   Asegúrate de usar patrones apropiados para la clase que tengas:
   - `LocalDate` → símbolos de fecha (año, mes, día).
   - `LocalTime` → símbolos de hora, minutos, segundos.
   - `LocalDateTime` → ambos tipos de símbolos.
3. **Zonas Horarias**
   Si necesitas mostrar la hora con una zona horaria (por ejemplo, UTC, EST), considera usar `ZonedDateTime` en lugar de `LocalDateTime`. Esto evita errores en la presentación de la hora a usuarios que se ubiquen en diferentes zonas.

4. **Texto Adicional**
   Colocar texto dentro de comillas simples `'...'` en el patrón permite incluir cualquier palabra o frase, como “a las” o “horario local”. Si necesitas mostrar una comilla simple en la salida, usa dos comillas juntas `''` para “escaparla”.

5. **Validación y Errores**
   Si observas una excepción tipo `DateTimeException` o `IllegalArgumentException`, revisa que tu patrón sea compatible con el tipo de objeto de fecha/hora y que no contenga caracteres no reconocidos por el formateador (por ejemplo, la letra `T` si no está escapada correctamente).

---

5. Conclusión

Aplicar formatos a números, fechas y horas es esencial para mostrar información de manera clara y profesional. Java proporciona herramientas robustas (`DecimalFormat`, `NumberFormat`, `DateTimeFormatter`, etc.) que, con el uso adecuado de patrones y símbolos, permiten personalizar casi cualquier estilo de presentación.

Internacionalización y Localización en Java

1. Introducción General

Cuando desarrollamos aplicaciones que podrían utilizarse en diferentes países o idiomas, debemos considerar cómo adaptarlas para que cada usuario reciba la información de una manera clara y familiar. A esto se le llama **internacionalización** (I18N) y **localización** (L10N):

- **Internacionalización (I18N)**: Preparar nuestro programa para que pueda adaptarse a múltiples idiomas o convenciones sin tener que reescribir la lógica principal.
- **Localización (L10N)**: Implementar en la práctica el soporte para idiomas y formatos concretos (por ejemplo, traducir textos y formatear valores según el país).

Por ejemplo, si mostramos una fecha como “4/1/22” en Estados Unidos, la gente pensaría que es el 1 de abril de 2022. Sin embargo, en otros países (como Reino Unido), interpretarían esa fecha como el 4 de enero de 2022. Además, ciertas palabras en inglés (“behaviors”) cambian su ortografía en inglés británico (“behaviours”). Esta guía te ayudará a manejar estas diferencias de una forma ordenada y escalable.

---

2. Comprendiendo la Clase Locale

2.1 Formato de `Locale`

La clase `Locale` (en el paquete `java.util`) es fundamental para entender en qué idioma y región se debe basar nuestra aplicación. Un **locale** identifica el idioma y, opcionalmente, el país. Por ejemplo:

- `Locale.getDefault()` podría devolver `en_US` (inglés y Estados Unidos).
- Formato estándar:
    - Solo idioma (minúsculas), ej. `fr` (francés).
    - Idioma y país, ej. `en_US` (inglés de Estados Unidos). Requiere el idioma en minúsculas, y luego el país en mayúsculas.

Ejemplo

```
Locale locale = Locale.getDefault();
System.out.println(locale); // Muestra algo como en_MX
```

Si necesitas crear un `Locale` específico, puedes:

- Usar constantes predefinidas como `Locale.GERMANY`.
- Crear uno con `new Locale("fr")` o `new Locale("hi", "IN")`.
- Usar la clase interna `Locale.Builder()` para mayor flexibilidad.

---

3. Formateo y Análisis (Parsing) de Números

3.1 Número General, Moneda y Porcentajes

Para mostrar o leer números según un locale, utilizamos la clase `NumberFormat` del paquete `java.text`. Dependiendo de si queremos formatear valores generales, monetarios o porcentuales, usamos distintos métodos estáticos:

- `NumberFormat.getInstance(Locale locale)`
- `NumberFormat.getCurrencyInstance(Locale locale)`
- `NumberFormat.getPercentInstance(Locale locale)`

Ejemplo: Números con Separadores

```
int asistentesPorAño = 3_200_000;
int asistentesPorMes = asistentesPorAño / 12;

NumberFormat usFormat = NumberFormat.getInstance(Locale.US);
System.out.println(usFormat.format(asistentesPorMes)); // 266,666

NumberFormat deFormat = NumberFormat.getInstance(Locale.GERMANY);
```

System.out.println(deFormat.format(asistentesPorMes)); // 266.666

Observa cómo el separador de miles cambia según la convención local.

Ejemplo: Moneda
double precio = 48;
NumberFormat moneda = NumberFormat.getCurrencyInstance(Locale.US);
System.out.println(moneda.format(precio)); // $48.00

Si usáramos `Locale.UK`, obtendríamos `£48.00`.

Ejemplo: Porcentajes
double tazaExito = 0.802;
NumberFormat porcent = NumberFormat.getPercentInstance(Locale.US);
System.out.println(porcent.format(tazaExito)); // 80%

3.2 Análisis (Parsing) de Números

`NumberFormat.parse(String fuente)` convierte una cadena en un número (retorna un objeto de tipo `Number`). Debes tener en cuenta la convención local, ya que un "." o "," podría significar cosas distintas según el país.

String valor = "40.45";
NumberFormat enFormat = NumberFormat.getInstance(Locale.US);
System.out.println(enFormat.parse(valor)); // 40.45

NumberFormat frFormat = NumberFormat.getInstance(Locale.FRANCE);
System.out.println(frFormat.parse(valor)); // 40 (trunca tras el punto porque para Francia "." no indica decimal)

3.3 CompactNumberFormat

Para mostrar números con un formato abreviado, Java ofrece `CompactNumberFormat`, accesible por medio de `NumberFormat.getCompactNumberInstance(...)`. Muestra valores en una forma reducida dependiendo de la región (por ejemplo, "7M" para 7 millones en inglés o "7 Mio." en alemán).

- **SHORT**: Usado por defecto, usa símbolos como K, M, B (thousand, million, billion).
- **LONG**: Usa palabras completas (por ejemplo, "7 million").

NumberFormat compUs = NumberFormat.getCompactNumberInstance(Locale.US,
NumberFormat.Style.LONG);
System.out.println(compUs.format(7_123_456)); // "7 million"

---

4. Formateo de Fechas y Horas

4.1 Clases Clave

Para representar fechas y horas en Java:

- `LocalDate`: Solo fecha (año, mes, día).
- `LocalTime`: Solo hora (horas, minutos, segundos).
- `LocalDateTime`: Fecha y hora combinadas.
- `ZonedDateTime`: Incluye fecha, hora y zona horaria.

4.2 `DateTimeFormatter`

Permite formatear o analizar valores de fecha/hora. A diferencia de patrones personalizados (`ofPattern("dd/MM/yyyy")`), para localización usamos métodos como:

- `DateTimeFormatter.ofLocalizedDate(FormatStyle style)`
- `DateTimeFormatter.ofLocalizedTime(FormatStyle style)`
- `DateTimeFormatter.ofLocalizedDateTime(FormatStyle dateStyle, FormatStyle timeStyle)`

El `FormatStyle` puede ser `SHORT`, `MEDIUM`, `LONG` o `FULL`. También podemos combinar con `withLocale(...)` para aplicar un idioma/país específico.

Ejemplo

```
LocalDateTime fechaHora = LocalDateTime.of(2022, 10, 20, 15, 12, 34);

DateTimeFormatter dtf = DateTimeFormatter.ofLocalizedDateTime(FormatStyle.SHORT, FormatStyle.SHORT);
System.out.println(dtf.format(fechaHora)); // 10/20/22, 3:12 PM (en_US)

DateTimeFormatter dtfIt = dtf.withLocale(new Locale("it", "IT"));
System.out.println(dtfIt.format(fechaHora)); // 20/10/22, 15:12 (en Italia)
```

---

5. Configurando el Locale por Categorías

5.1 Categorías DISPLAY y FORMAT

La clase `Locale` incluye una enumeración `Locale.Category`, que permite establecer distintos locales para la presentación (DISPLAY) y para el formateo (FORMAT). Por defecto, cuando usas `Locale.setDefault(new Locale("en", "US"))`, se cambian ambos. Sin embargo, podrías hacer:

Locale.setDefault(Locale.Category.DISPLAY, new Locale("es", "ES"));
Locale.setDefault(Locale.Category.FORMAT, Locale.US);

- **DISPLAY** controla cómo se muestran textos (por ejemplo, el idioma con que se indica "español" o "Spanish").
- **FORMAT** controla cómo se formatean números, fechas y monedas.

Esto te permite, por ejemplo, mostrar etiquetas en español pero usar el formato numérico de Estados Unidos, y viceversa.

---

6. Conclusión

La internacionalización y localización permiten que tu aplicación sea utilizada en cualquier parte del mundo, manejando adecuadamente las diferencias de idioma, formato de fechas, representación de números y moneda. En Java, lograrlo es sencillo con las clases y métodos adecuados (`Locale`, `NumberFormat`, `DateTimeFormatter`, entre otros).

Para programadores lo más importante es:

- Familiarizarse con `Locale` y su uso.
- Conocer métodos para formatear números, monedas y fechas acorde a la región/idioma.
- Manejar archivos de propiedades para traducir textos y no dejar en código duro cadenas en el código.
- Probar la aplicación con múltiples configuraciones de locale para confirmar que todo funciona según lo esperado.

Con estas bases, tendrás la capacidad de escalar tus aplicaciones para soportar un público internacional y ofrecer una experiencia más personalizada y profesional a tus usuarios.

Carga de Propiedades con Resource Bundles en Java

1. Introducción General

Cuando se internacionaliza (I18N) una aplicación, es habitual mantener los textos que se muestran al usuario en archivos externos llamados ***properties files***, en lugar de incrustar estas cadenas dentro del código. De esta manera, resulta sencillo traducir o adaptar el contenido según el ***locale*** (idioma y región) deseado, sin tener que recompilar la aplicación.

En Java, estos archivos se denominan ***resource bundles***. Se manejan a través de la clase `ResourceBundle`, que actúa de forma similar a un **mapa** (`Map<String, String>`), donde cada línea representa un par de clave=valor.

---

2. Creación de Resource Bundles

2.1 Estructura de Archivos .properties

Para cada idioma/región de tu aplicación, se suele crear un archivo `.properties`. El nombre del archivo sigue la convención:

NombreBase_ll_CC.properties

Donde:

- `NombreBase` es el nombre de tu *resource bundle* (por ejemplo, `Zoo` o `Messages`).
- `ll` es el código de idioma en minúsculas (por ejemplo, `en`, `fr`, `es`).
- `CC` es el código de país en mayúsculas (por ejemplo, `US`, `FR`, `ES`). Esto puede omitirse si no te interesa un país en concreto.

Dentro del archivo `.properties`, cada línea define un par `clave=valor`. Por ejemplo, para un archivo `Zoo_en.properties`:

hello=Hello

open=The zoo is open

Y para `Zoo_fr.properties`:

hello=Bonjour

open=Le zoo est ouvert

Al llamar a estos archivos desde Java, se mostrará el idioma adecuado según el *locale* que seleccionemos en nuestra aplicación.

---

3. Uso de `ResourceBundle.getBundle(...)`

Para cargar los recursos desde un archivo `.properties`, llamamos al método:

ResourceBundle rb = ResourceBundle.getBundle("Zoo", locale);

- **`"Zoo"`**: Es el nombre base del archivo de propiedades (sin extensión ni el sufijo de idioma/país).
- **`locale`**: Objeto `Locale` que especifica el idioma y, opcionalmente, la región.

Una vez tenemos el `ResourceBundle`, podemos acceder a los valores de las claves con:

String valor = rb.getString("hello");

3.1 Búsqueda de Archivos

Java intenta ubicar el archivo adecuado según el ***locale*** indicado. Por ejemplo, si solicitamos un `ResourceBundle` llamado `"Zoo"` con el locale francés de Francia (`new Locale("fr", "FR")`), el orden de búsqueda es:

1. `Zoo_fr_FR.properties`
2. `Zoo_fr.properties`
3. `Zoo_en_US.properties` (si el *locale* por defecto del sistema es `en_US`)
4. `Zoo_en.properties`
5. `Zoo.properties` (sin especificar idioma ni región)
6. Si no encuentra ningún archivo, lanza `MissingResourceException`.

Este mecanismo permite combinar archivos cada vez más “genéricos”: primero se busca la coincidencia exacta, luego solo por idioma, después por la configuración predeterminada, etc.

---

4. Jerarquía y Herencia de Resource Bundles

4.1 Padres e Hijos

Cuando Java encuentra el *bundle* más específico (p. ej., `Zoo_fr_FR.properties`), cualquier clave no encontrada ahí se busca en el siguiente nivel (por ejemplo, `Zoo_fr.properties`). Si tampoco está ahí, se usa `Zoo.properties` y así sucesivamente, hasta encontrar la clave o agotar los archivos.

Importante: Una vez que Java haya encontrado un conjunto de archivos que cubra el idioma indicado, **no** recurre a otros *locales*. Es decir, si se detecta `Zoo_fr` para un idioma francés, no se mezclan claves de `Zoo_en`.

---

5. Ejemplo Práctico

Supongamos que tenemos estos archivos:

`Zoo.properties`:

name=Vancouver Zoo

`Zoo_en.properties`:

hello=Hello

open=is open

`Zoo_en_US.properties`:

name=The Zoo

`Zoo_en_CA.properties`:

visitors=Canada visitors

Al pedir:

```
Locale.setDefault(new Locale("en", "US"));      // Predeterminado: en_US
Locale locale = new Locale("en", "CA");         // Solicitamos en_CA
ResourceBundle rb = ResourceBundle.getBundle("Zoo", locale);
System.out.print(rb.getString("hello") + ". "); // Buscado en Zoo_en_CA -> no está
                                                // Pasa a Zoo_en -> "Hello"
System.out.print(rb.getString("name") + " ");  // Zoo_en_CA -> no está
                                               // Zoo_en -> no está
                                               // Zoo.properties -> "Vancouver Zoo"
```

System.out.print(rb.getString("open") + " "); // Zoo_en_CA -> no está

// Zoo_en -> "is open"

System.out.print(rb.getString("visitors"));   // Zoo_en_CA -> "Canada visitors"

Salida:

Hello. Vancouver Zoo is open Canada visitors

Observa cómo Java usa cada archivo en cascada, buscando las claves desde `Zoo_en_CA.properties` hasta `Zoo.properties`, sin usar el locale predeterminado (`en_US`) porque encontró un archivo específico para `en_CA`.

---

6. Formato de Mensajes con `MessageFormat`

Si necesitas insertar parámetros en las cadenas, usa la clase `MessageFormat`. Por ejemplo, en tu archivo `.properties` podrías tener:

helloByName=Hello, {0} and {1}

Para formatear:

String formato = rb.getString("helloByName");

String resultado = MessageFormat.format(formato, "Ana", "Carlos");

// Resultado: "Hello, Ana and Carlos"

De esta manera puedes insertar variables en tus mensajes, en el orden que necesites.

---

7. Uso de la Clase `Properties`

La clase `Properties` (ubicada en `java.util`) se utiliza como una estructura `Map<String, String>` especializada para cargar o almacenar valores de configuración. Internamente, `ResourceBundle` utiliza esta clase para procesar los archivos `.properties`.

Properties props = new Properties();

props.setProperty("name", "Our zoo");

props.setProperty("open", "10am");

Puedes obtener valores con:

System.out.println(props.getProperty("name")); // "Our zoo"

System.out.println(props.getProperty("camel", "Bob")); // "Bob" si no existe la clave "camel"

Aunque `Properties` y `ResourceBundle` son distintas, ambas pueden manejar archivos `.properties`. `ResourceBundle` está más orientado a la localización y se integra directamente con la lógica de carga de locales.

---

8. Conclusión

El uso de ***resource bundles*** (archivos de propiedades) es clave para **internacionalizar** y **localizar** una aplicación Java. Algunas ventajas:

1. **Facilita la traducción**: No es necesario reescribir el código para añadir o modificar un idioma.
2. **Separa la lógica de negocio de los textos**: Los mensajes no residen en el código, sino en archivos externos.
3. **Extensibilidad**: Puedes agregar nuevos locales o regiones fácilmente, solo añadiendo más archivos `.properties`.

Lo más importante es comprender los fundamentos:

- Cómo se nombran los archivos `.properties`.
- Cómo Java decide cuál archivo usar en función del `Locale`.
- Cómo recuperar cadenas y, de ser necesario, complementar con `MessageFormat` para incluir variables.