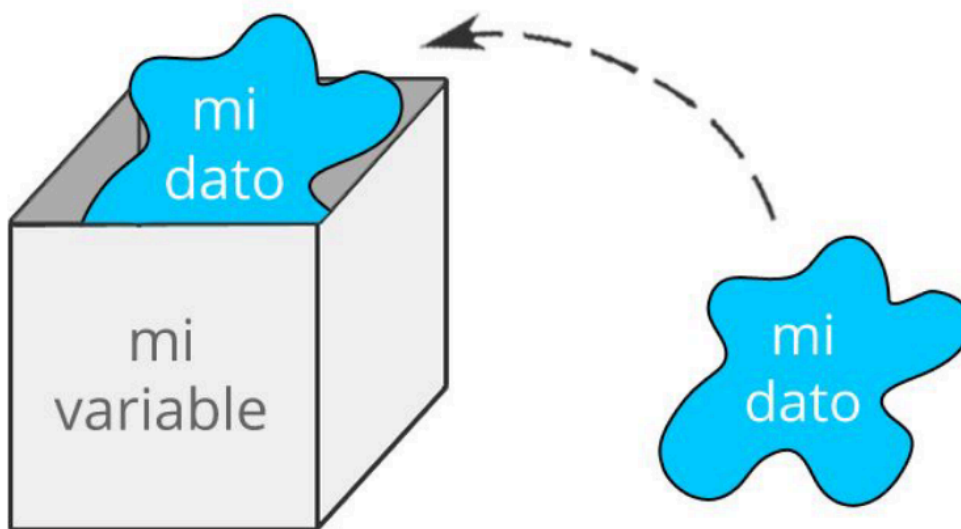


Variables

Definición

Cuando trabajamos con información, necesitamos un espacio en nuestra memoria para contenerla.

Cuando programamos también necesitamos almacenar información en la memoria de la computadora para reusarla o simplemente para trabajar con ella.



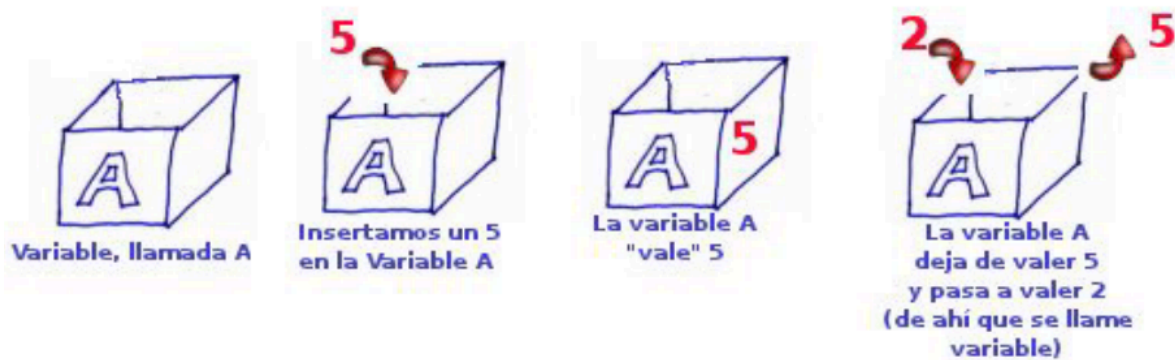
El espacio en la memoria de la computadora donde guardamos un dato es lo que llamamos "variable".

No es importante para nosotros conocer exactamente en qué lugar de la computadora se guardan nuestros datos, siempre y cuando la computadora los encuentre cuando los necesitamos.

Tengamos en cuenta la enorme cantidad de información que tiene almacenada la computadora.

Es por esto que este espacio o lugar dentro de la computadora donde guardamos información, debe tener un nombre: para que la computadora pueda encontrarlo entre la enorme cantidad de información que maneja.

En conclusión, una variable es un espacio en la memoria de la computadora, que tiene un nombre propio, definido por el programador.



Como en matemática, una variable es un dato que tiene un nombre y cuyo valor puede variar.

Cuando le ponemos "nombre" a un dato, podemos mencionar repetidamente por su nombre, y reusar nuestros cálculos en casos diferentes, simplemente cambiando el valor que referimos con ese "nombre" ⇒ Las variables permiten que los programas puedan adaptarse y reusarse para realizar las mismas tareas sobre diferentes casos.

Una de las razones por las que usamos variables es que nos permiten **reutilizar** fácilmente los valores en diferentes partes de nuestro código.

Cuando reutilizamos un valor, aparecerá en varios lugares en nuestro código.

Volver a escribir ese valor se vuelve tedioso, lo que conduce a errores.

Aquí tenemos un número que reutilizamos para hacer algunos cálculos:

```
847595593392818109495
847595593392818109495 * 2
847595593392818109495 / 4
```

En lugar de escribir el mismo número una y otra vez, podemos guardarlo en una variable llamada **my_number**:

```
my_number = 847595593392818109495
my_number * 2
my_number / 4
```

Nombre de Variables

✗

| Incorrecto | Correcto |
|--------------------|-----------------------------|
| variable | edad |
| A B C | deposito retiro saldo |
| 1numero 2numero | numero1 numero2 |
| caso-1 caso-2 | caso_1 caso_2 |
| input | entrada |

✓




TIP:

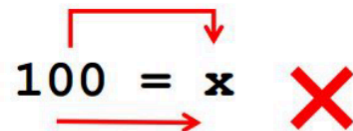
Las variables son Case Sensibles (ej. Nombre <> nombre)

Creación y Asignación de Variables

- La creación de variables se realiza a través de la **asignación** de un **valor** a la misma.
- El operador de asignación en Python es el “=”.



De derecha a izquierda



De izquierda a derecha



TIP:

Una **variable** es un valor que puede cambiar a lo largo de la ejecución de nuestro algoritmo

La idea de la variable en programación, se asemeja mucho a lo que es una variable matemática en una ecuación.

<https://es.wikipedia.org/wiki/Ecuaci%C3%B3n>

Una variable es un lugar de la memoria (ram) de la computadora para almacenar un dato y a este espacio se le asigna un “nombre”.

A través de su nombre, dentro de un programa se puede acceder a este espacio para obtener el valor de la variable

[https://es.wikipedia.org/wiki/Variable_\(programaci%C3%B3n\)](https://es.wikipedia.org/wiki/Variable_(programaci%C3%B3n))

Ejemplo

Por ejemplo, si tenemos las siguientes secuencia ordenada de instrucciones, de definición de variables:

- $y=2$
- $x=3$
- $y=8$

¿Qué valor tiene la variable “y” al finalizar el algoritmo?

La respuesta es 8.

Si pensamos, que cada asignación de una variable es un paso de un algoritmo y las ejecutamos en orden, primero haremos $y=2$, en este momento la variable “y” tiene el valor 2. Luego ejecutamos el segundo paso, $x=3$, donde asignamos a la variable “x” el valor 3. Por último, ejecutamos $y=8$, por lo que la variable “y” queda ahora con el valor 8.

Tipo de sentencia

Declaración

[Tipo de dato] [nombre de variable]

- String varString
- Entero varNumero
- Booleano varBool

Asignación

[nombre variable] = [expresión]

- varString = “hola”
- varNumero = $4+5$
- varBool = VERDADERO
- varNumero = varNumero + 1

A una variable se le puede asignar el valor de otra variable

Por ejemplo, si tenemos las siguientes secuencia ordenada de instrucciones, de definición de variables:

y=2
x=3
y=x
x=5

¿Qué valor tienen las variables x e y al finalizar el algoritmo?

x = 5
y = 3

Notar que la instrucción 4 únicamente modifica la variable x, mientras que y queda sin modificar.

A una variable se le puede asignar el valor resultado de operaciones.

Por ejemplo, si tenemos las siguientes secuencia ordenada de instrucciones, de definición de variables:

y=2 + 2
x= y +1
z=x + y
x=5

¿Qué valor tienen las variables x, y , z al finalizar el algoritmo?

x = 5
y = 4
z = 9

Tipos de datos en Java:

Java tiene 8 tipos primitivos que se dividen en cuatro grupos principales: enteros, punto flotante, caracteres y booleano. Cada uno de estos tipos tiene un tamaño fijo y un valor por defecto.

Tipos de Datos Enteros:

byte:

- Tamaño: 8 bits (1 byte).
- Rango: -128 a 127.
- Utilizado para ahorrar memoria en grandes arrays, principalmente en lugar de `int`, ya que su tamaño es la cuarta parte de un entero.

short:

- Tamaño: 16 bits (2 bytes).
- Rango: -32,768 a 32,767.
- Se puede utilizar en lugar de **byte** donde el rango de **byte** no es suficiente.

int:

- Tamaño: 32 bits (4 bytes).
- Rango: -2^{31} a $2^{31}-1$.
- Generalmente, la opción predeterminada para números enteros, a menos que no sea suficiente.

long:

- Tamaño: 64 bits (8 bytes).
- Rango: -2^{63} a $2^{63}-1$.
- Utilizado cuando se necesita un rango de valores más amplio que el que ofrece **int**.

Tipos de Datos de Punto Flotante:

float:

- Tamaño: 32 bits (4 bytes).
- Rango: aproximadamente $\pm 3.40282347E+38F$ (6-7 dígitos decimales significativos).
- Utilizado principalmente para ahorrar memoria en grandes arrays de números de punto flotante.

double:

- Tamaño: 64 bits (8 bytes).
- Rango: aproximadamente $\pm 1.79769313486231570E+308$ (15 dígitos decimales significativos).
- Es el tipo por defecto para números de punto flotante en Java.

Tipo de Datos de Caracteres:

char:

- Tamaño: 16 bits (2 bytes).
- Rango: 0 a 65,536 (sin valores negativos).
- Utilizado para almacenar caracteres únicos, como letras, números o símbolos. Utiliza el formato Unicode, permitiendo almacenar cualquier carácter.

Tipo de Datos Booleano:

boolean:

- No se define específicamente el tamaño.
- Valores: solo puede ser `true` o `false`.
- Utilizado para banderas que rastrean condiciones verdaderas/falsas.

Características y Usos:

- Eficiencia: Los tipos primitivos son muy eficientes en términos de almacenamiento y velocidad porque están altamente optimizados por la JVM.
- Valor por defecto: Cada tipo primitivo tiene un valor por defecto, por ejemplo, `0` para los tipos numéricos, `false` para `boolean` y `'\u0000'` (el carácter nulo) para `char`.
- No son objetos: A diferencia de los objetos, los primitivos no tienen métodos; son valores puros sin propiedades ni comportamientos.
- Stack Allocation: Los tipos primitivos se almacenan en la pila, lo que los hace mucho más rápidos que los objetos, que se almacenan en el montón.

Usar tipos de datos primitivos en lugar de objetos puede resultar en una aplicación más rápida y menos demandante de recursos. Sin embargo, carecen de la flexibilidad y los beneficios de los objetos, como poder invocar métodos o ser null (excepto en el caso de envoltorios o "wrappers" que permiten utilizarlos como objetos, por ejemplo, `Integer` para `int`, `Double` para `double`, etc.). Estos envoltorios son parte de la API de Java Collections y permiten trabajar con tipos primitivos en colecciones que solo soportan objetos.

Algunos ejemplos del uso de tipo de datos

```
public class Primitivos {  
    public static void main(String[] args) {  
        // Ejemplo de byte  
        byte numeroByte = 100;  
        System.out.println("Valor byte: " + numeroByte);  
  
        // Ejemplo de short  
        short numeroShort = 30000;  
        System.out.println("Valor short: " + numeroShort);  
  
        // Ejemplo de int  
        int numeroInt = 200000;  
        System.out.println("Valor int: " + numeroInt);  
  
        // Otra forma de escribirlos para separar miles  
        numeroInt=1_000_000;  
        System.out.println("Valor de int: "+numeroInt);  
  
        // Ejemplo de long  
        long numeroLong = 15000000000L; // La 'L' al final indica que es un literal long  
        System.out.println("Valor long: " + numeroLong);  
    }  
}
```

Salida por la consola:

```
Valor byte: 100  
Valor short: 30000  
Valor int: 200000  
Valor de int: 1000000  
Valor long: 15000000000
```

```
public class PuntoFlotante {  
    public static void main(String[] args) {  
        // Ejemplo de float  
        float numeroFloat = 3.14f; // La 'f' al final indica que es un literal float  
        System.out.println("Valor float: " + numeroFloat);  
  
        // Ejemplo de double  
        double numeroDouble = 3.141592653589793;  
        System.out.println("Valor double: " + numeroDouble);  
    }  
}
```

Salida por la consola:


```
Valor byte: 100
Valor short: 30000
Valor int: 200000
Valor de int: 1000000
Valor long: 15000000000
```

```
public class TipoCaracter {
    public static void main(String[] args) {
        // Ejemplo de char
        char letra = 'A';
        System.out.println("Valor char: " + letra);

        char unicodeChar = '\u0041'; // Representación Unicode de 'A'
        System.out.println("Valor char con Unicode: " + unicodeChar);

        char asciiChar = 65;
        System.out.println("Valor char con Ascii: " + asciiChar);
    }
}
```

Salida por la consola:

```
Valor char: A
Valor char con Unicode: A
Valor char con Ascii: A
```

```
public class TipoBooleano {
    public static void main(String[] args) {
        // Ejemplo de boolean
        boolean verdadero = true;
        System.out.println("Valor boolean verdadero: " + verdadero);

        boolean falso = false;
        System.out.println("Valor boolean falso: " + falso);
    }
}
```

Salida por la consola:

```
Valor boolean verdadero: true  
Valor boolean falso: false
```

Clase String

La clase String en Java es una de las clases más utilizadas y representa una secuencia de caracteres. Es decir, los objetos de tipo String contienen cadenas de texto. En Java, la clase String es inmutable, lo que significa que una vez que se crea una instancia de un String, no se puede cambiar su contenido.

Ejemplos de uso: `String saludo = "Hola como andan ?";`

Este tipo de variable no es primitivo, ya veremos la explicación más adelante.

Clase LocalDate

La clase LocalDate en Java representa una fecha sin información de hora o zona horaria. Es parte de la API `java.time`, introducida en Java 8 para abordar las deficiencias de las antiguas clases `Date` y `Calendar`. Aquí están los aspectos más importantes de `LocalDate` junto con ejemplos de su uso:

Aspectos Importantes de `LocalDate`

Inmutabilidad: Los objetos `LocalDate` son inmutables, lo cual facilita su manejo, especialmente en aplicaciones multihilo, ya que son seguros en cuanto a hilos y no cambian una vez creados.

Representación de Fecha: Representa una fecha en formato ISO (`aaaa-MM-dd`) sin hora ni zona horaria.

API Fluent: Proporciona métodos que pueden encadenarse para realizar operaciones en una secuencia fluida y legible.

No se preocupa por la Zona Horaria: Es ideal para usar cuando solo necesitas representar una fecha, como un cumpleaños o un día festivo, sin preocuparte por las zonas horarias o la hora exacta.

Ejemplos de Uso de `LocalDate`

Aquí hay algunos ejemplos que ilustran cómo se utiliza `LocalDate` en Java:

```
public class Fecha {  
    public static void main(String[] args) {  
        // Obtener la fecha actual  
        LocalDate hoy = LocalDate.now();  
        System.out.println("Hoy: " + hoy);  
  
        // Crear una fecha específica  
        LocalDate fechaEspecifica = LocalDate.of( year: 2024, month: 1, dayOfMonth: 1);  
        System.out.println("Fecha específica: " + fechaEspecifica);  
  
        // Parsear una fecha de un texto  
        LocalDate fechaDesdeTexto = LocalDate.parse( text: "2024-01-01");  
        System.out.println("Fecha desde texto: " + fechaDesdeTexto);  
    }  
}
```

```
LocalDate hoy = LocalDate.now();  
// Trabajar y manipular fechas  
// Añadir días a la fecha  
LocalDate mañana = hoy.plusDays( daysToAdd: 1);  
System.out.println("Mañana: " + mañana);  
  
// Restar días  
LocalDate ayer = hoy.minusDays( daysToSubtract: 1);  
System.out.println("Ayer: " + ayer);  
  
// Añadir meses  
LocalDate mesSiguiete = hoy.plusMonths( monthsToAdd: 1);  
System.out.println("Mes siguiente: " + mesSiguiete);  
  
// Cambiar el año  
LocalDate siguienteAño = hoy.withYear(2025);  
System.out.println("Siguiete año: " + siguienteAño);
```

salida por la consola:

```
Mañana: 2024-03-01  
Ayer: 2024-02-28  
Mes siguiente: 2024-03-29  
Siguiete año: 2025-02-28
```

```
import java.time.LocalDate;  
  
public class Fecha3 {  
    public static void main(String[] args) {  
        // Comparar fechas  
        boolean esAntes = LocalDate.now().isBefore(LocalDate.of( year: 2025, month: 1, dayOfMonth: 1));  
        System.out.println("Hoy es antes de 2025-01-01? " + esAntes);  
  
        boolean esDespues = LocalDate.now().isAfter(LocalDate.of( year: 2020, month: 1, dayOfMonth: 1));  
        System.out.println("Hoy es después de 2020-01-01? " + esDespues);  
    }  
}
```

Salida por la consola:

Hoy es antes de 2025-01-01? true

Hoy es después de 2020-01-01? true

```
public class Fecha4 {  
    public static void main(String[] args) {  
        LocalDate hoy = LocalDate.now();  
        // Obtener año, mes y día  
        int año = hoy.getYear();  
        System.out.println("Año: " + año);  
  
        Month mes = hoy.getMonth();  
        System.out.println("Mes: " + mes);  
  
        int día = hoy.getDayOfMonth();  
        System.out.println("Día: " + día);  
  
        // Ajustar la fecha al primer día del mes  
        LocalDate primerDiaDelMes = hoy.with(TemporalAdjusters.firstDayOfMonth());  
        System.out.println("Primer día del mes: " + primerDiaDelMes);  
    }  
}
```

Salida por la consola:

Año: 2024

Mes: FEBRUARY

Día: 29

Primer día del mes: 2024-02-01

Cada versión de Java, tenemos para ver su correspondiente link al que podemos acceder a las consultas sobre el contenido de cualquier clase en Java.

El link para acceder a la versión 17 de java es :

<https://docs.oracle.com/en/java/javase/17/docs/api/index.html>

Method Summary

| All Methods | Static Methods | Instance Methods | Concrete Methods |
|-------------------|---|--|------------------|
| Modifier and Type | Method | Description | |
| Temporal | <code>adjustInto(Temporal temporal)</code> | Adjusts the specified temporal object to have the same date as | |
| LocalDateTime | <code>atStartOfDay()</code> | Combines this date with the time of midnight to create a Local of this date. | |
| ZonedDateTime | <code>atStartOfDay(ZoneId zone)</code> | Returns a zoned date-time from this date at the earliest valid rules in the time-zone. | |
| LocalDateTime | <code>atTime(int hour, int minute)</code> | Combines this date with a time to create a LocalDateTime. | |
| LocalDateTime | <code>atTime(int hour, int minute, int second)</code> | Combines this date with a time to create a LocalDateTime. | |
| LocalDateTime | <code>atTime(int hour, int minute, int second, int nanoOfSecond)</code> | Combines this date with a time to create a LocalDateTime. | |
| LocalDateTime | <code>atTime(LocalTime time)</code> | Combines this date with a time to create a LocalDateTime. | |
| OffsetDateTime | <code>atTime(OffsetTime time)</code> | Combines this date with an offset time to create an OffsetDat | |

Por ejemplo, acá estamos viendo parte de los métodos de la clase `LocalDate`.

Clase Period

La clase `Period` en Java es utilizada para modelar una cantidad de tiempo en términos de años, meses y días. Es muy útil para calcular diferencias entre fechas, como por ejemplo, la edad de una persona. Aquí te muestro cómo puedes utilizar `Period` junto con `LocalDate` para calcular la edad de alguien:

```
public class Edad {
    public static void main(String[] args) {
        // Fecha de nacimiento
        LocalDate fechaNacimiento = LocalDate.of( year: 1990, month: 5, dayOfMonth: 15);
        // Fecha actual
        LocalDate fechaActual = LocalDate.now();
        // Calcular el periodo entre la fecha de nacimiento y la fecha actual
        Period edad = Period.between(fechaNacimiento, fechaActual);
        // Mostrar la edad calculada
        System.out.println("Edad: " + edad.getYears() + " años, " + edad.getMonths() + " meses, y " + edad.getDays() + " días.");
    }
}
```

Salida por la consola:

```
Edad: 33 años, 9 meses, y 14 días.
```

En este ejemplo:

- Se crea una `LocalDate` que representa la fecha de nacimiento.
- Se obtiene la fecha actual utilizando `LocalDate.now()`.

- Se usa `Period.between()` para calcular el período entre la fecha de nacimiento y la fecha actual, lo que nos da la edad en años, meses y días.
- Finalmente, se extraen los años, meses y días del `Period` utilizando los métodos `getYears()`, `getMonths()`, y `getDays()` respectivamente.

El resultado mostrará la edad de la persona en años, meses y días. Es importante tener en cuenta que `Period` calcula la diferencia basada en las fechas solamente, sin considerar las horas, minutos o segundos, por lo que es perfecto para calcular la edad.

Clase Date

La clase `Date` es una de las clases originales en Java para manejar fechas y tiempos, mientras que `LocalDate` es parte de la moderna API de fecha y hora introducida en Java 8 (paquete `java.time`). Ambas clases se utilizan para trabajar con fechas, pero tienen diferencias significativas en términos de diseño, funcionalidad y uso. Vamos a explorar cómo se relacionan y cómo puedes convertir entre ellas.

Diferencias Clave

- **Mutabilidad:** `Date` es mutable, lo que significa que su valor puede cambiar después de haber sido creado. Por el contrario, `LocalDate` es inmutable, lo que proporciona una mayor seguridad en aplicaciones multihilo y evita cambios no deseados en el valor de la fecha.
- **Precisión:** `Date` representa un punto específico en el tiempo, incluyendo fecha y hora, y se mide en milisegundos desde la "epoch" Unix (1 de enero de 1970). `LocalDate`, en cambio, representa solo una fecha (año, mes, día) sin hora ni zona horaria.
- **Uso:** `Date` puede ser más difícil de manejar debido a su mutabilidad y su legado de métodos obsoletos o poco claros. `LocalDate` forma parte de una API de fecha y hora mucho más coherente y bien diseñada, que facilita el manejo de fechas.

Conversión entre Date y LocalDate

Si estás trabajando con código antiguo que utiliza `Date`, pero quieres aprovechar las características de `LocalDate`, puedes necesitar convertir entre estos dos tipos. Aquí te muestro cómo:

Para convertir un objeto `Date` en un `LocalDate`, primero debes convertir el `Date` a un `Instant`, y luego al `LocalDate` correspondiente, ajustándolo a la zona horaria adecuada (usualmente la zona horaria por defecto del sistema).

```
public class ClaseDate {  
    public static void main(String[] args) {  
  
        // Crear un objeto Date (representa fecha y hora)  
        Date fechaDate = new Date();  
        // Convertir Date a LocalDate  
        LocalDate fechaLocalDate = fechaDate.toInstant().atZone(ZoneId.systemDefault()).toLocalDate();  
        System.out.println("LocalDate: " + fechaLocalDate);  
        // Crear un objeto LocalDate  
        LocalDate fecha = LocalDate.now();  
        // Convertir LocalDate a Date  
        Date fechaNueva = Date.from(fecha.atStartOfDay(ZoneId.systemDefault()).toInstant());  
  
        System.out.println("Date: " + fechaNueva);  
    }  
}
```

salida por la consola:

LocalDate: 2024-02-29

Date: Thu Feb 29 00:00:00 UYT 2024

Clase Calendar

- **Antigüedad:** Calendar es parte de las API más antiguas de Java (introducida en Java 1.1) y se encuentra en el paquete `java.util`.
- **Mutabilidad:** Es mutable, lo que significa que los objetos Calendar pueden cambiar de estado. Esto puede llevar a problemas, especialmente en entornos multihilo, donde las mutaciones inesperadas pueden causar errores difíciles de rastrear.
- **Complejidad:** Proporciona una amplia gama de funciones, pero su uso puede ser complejo debido a su mutabilidad y a la necesidad de configurar explícitamente el tiempo y la zona horaria.
- **Zona horaria:** Maneja explícitamente las zonas horarias y puede representar tanto la fecha como la hora.


```
public class FechaCalendar {  
    public static void main(String[] args) {  
        // Crear una instancia de Calendar  
        Calendar calendar = new GregorianCalendar();  
        // Convertir Calendar a LocalDate  
        LocalDate localDate = LocalDate.ofInstant(calendar.toInstant(), calendar.getTimeZone().toZoneId());  
        System.out.println("LocalDate: " + localDate);  
        // De LocalDate a Calendar  
        LocalDate localFecha = LocalDate.now();  
  
        // Convertir LocalDate a Calendar  
        Calendar calendar_nuevo = GregorianCalendar.from(localFecha.atStartOfDay(ZoneId.systemDefault()));  
  
        System.out.println("Calendar: " + calendar_nuevo.getTime());  
    }  
}
```

Salida de consola:

LocalDate: 2024-02-29

Calendar: Thu Feb 29 00:00:00 UYT 2024