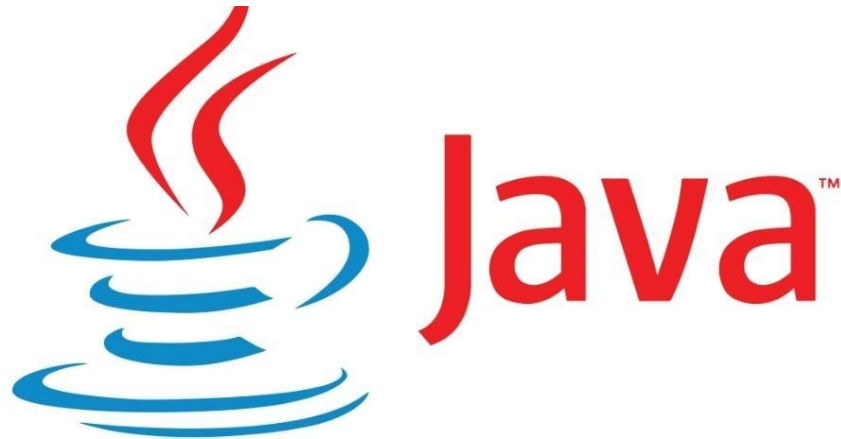


# Desarrollo Avanzado con Java



# Índice

<b>Recordando cosas útiles</b>	<b>4</b>
Array copy	4
Strings – Métodos útiles	4
Librería Math	5
Clase Arrays	6
<b>Strings Mutables</b>	<b>7</b>
Constructores	7
Algunos métodos	7
<b>Excepciones</b>	<b>8</b>
Tipo de excepciones	9
¿Cómo declarar excepciones?	9
Lanzar Excepciones	10
Capturar excepciones	11
<b>Opcionales</b>	<b>14</b>
<b>Api de Fecha y Hora</b>	<b>15</b>
Date	15
Clase Calendar	15
Ejemplo de seteo de fecha	16
Ejemplo de GregorianCalendar	16
DateFormat	17
Api java.time	18
Instant	18
Duration	18
Algunos métodos de Duration	19
Local Date y Times	20
Parseo y Formateo de fechas	20
<b>Api Stream</b>	<b>22</b>
Pipeline	22
Operaciones intermedias	23
Filter	23
Map y FlatMap	23

Distinct	24
Skip y Limit	24
Sorted	26
Operaciones Terminales	26
Collectors	26
ForEach	27
Otras operaciones finales	28
Operaciones terminales de búsqueda	28
Reduce	29

## Recordando cosas útiles

### Array copy

Es un método public y static de la clase System que permite copiar el array completo o una porción del mismo.

**public static void arraycopy**(Object src, int posSrc, Object dest, int destPos, int length)

*src*: arreglo origen

*posSrc*: la posición desde la que se desea comenzar a copiar

*dest*: arreglo destino

*destPos*: la posición desde la que se desea empezar a copiar en el destino

*length*: cantidad de posiciones a copiar

Un ejemplo

```
public class app {  
    Run | Debug  
    public static void main(String[] args) {  
        String [] heroes= {"Spiderman", "Ironman", "Capitan America", "Black Widow", "Vision"};  
        String [] copia= new String[2];  
  
        System.out.println("Arreglo completo: "+Arrays.toString(heroes));  
        System.arraycopy(heroes, srcPos: 2, copia, destPos: 0, length: 2);  
        System.out.println("Copia de heroes: "+Arrays.toString(copia));  
    }  
}
```

```
Arreglo completo: [Spiderman, Ironman, Capitan America, Black Widow, Vision]  
Copia de heroes: [Capitan America, Black Widow]
```

### Strings – Métodos útiles

- **equals(Object obj)**: retorna true si los strings son iguales. Distingue entre mayúscula y minúscula.
- **equalsIgnoreCase(String s)**: igual que equals pero no distingue entre mayúscula y minúscula.
- **charAt(int index)**: retorna el carácter en la posición pasada por parámetro.
- **indexOf(char c)**: retorna el índice de la primera aparición de c.
- **length()**: retorna el número de caracteres que componen la cadena.
- **compareTo(String s)**: compara lexicográficamente los strings. Retorna un número menor a 0 si es la instancia es menor que el parámetro, 0 si son iguales y un número mayor a 0 si la instancia es mayor que el parámetro.

- **substring(int ini):** retorna un nuevo string, que es una subcadena del string original, comenzando desde la posición ini hasta el final.
- **substring(int ini, int end):** retorna un nuevo string, que es una subcadena del string original, comenzando desde la posición ini hasta la posición end.
- **toLowerCase()** o **toUpperCase():** retorna un nuevo String pero todo en minúscula o mayúscula respectivamente.
- **replace(char oldChar, char newChar):** retorna un nuevo string con las apariciones de oldChar reemplazadas por newChar.
- **concat(String s):** retorna un nuevo String compuesto por el string representado por el objeto y el string pasado por parámetro.
- **trim():** retorna un nuevo string sin los espacios en blanco del principio y el final.

Ejemplos:

```
public static void main(String[] args) {
    String s = "ABBA";
    String s2 = "ALLA";

    System.out.println("s CHAR AT: "+s.charAt(index: 0));
    System.out.println("s2 CHAR AT: "+s2.charAt(index: 1));
    System.out.println("INDEX OF: "+s.indexOf(ch: 'B'));
    System.out.println("s COMPARE TO s2: "+s.compareTo(s2));
    System.out.println("s2 COMPARE TO s: "+s2.compareTo(s));
    System.out.println("s SUBSTRING: "+s.substring(beginIndex: 0, endIndex: 2));
    System.out.println("s2 SUBSTRING: "+s2.substring(beginIndex: 0, endIndex: 2));

    String texto= "Manejo de Strings con JAVA";
    System.out.println("UPPER CASE: "+ texto.toUpperCase());
    System.out.println("LOWER CASE: "+ texto.toLowerCase());
    System.out.println("REPLACE: "+ texto.replace(oldChar: 'A', newChar: 'X'));
    System.out.println("TRIM: "+ "      HOLA      ".trim());
}
```

```
s CHAR AT: A
s2 CHAR AT: L
INDEX OF: 1
s COMPARE TO s2: -10
s2 COMPARE TO s: 10
s SUBSTRING: AB
s2 SUBSTRING: AL
UPPER CASE: MANEJO DE STRINGS CON JAVA
LOWER CASE: manejo de strings con java
REPLACE: Manejo de Strings con JXVX
TRIM: HOLA
```

## Librería Math

### Métodos

- **abs(int i):** retorna el valor absoluto de i.

- **ceil(double d):** en caso de que d sea un número con coma retorna el entero mayor inmediato. De lo contrario retorna d.
- **floor(double d):** en caso de que d sea un número con coma retorna el entero menor inmediato. De lo contrario retorna d.
- **random():** retorna un número aleatorio.
- **sqrt(double d):** retorna la raíz cuadrada de d.
- **min(int i1, int i2):** retorna el mínimo entre i1 e i2.

### **BigInteger y BigDecimal**

Son tipos de datos numéricos que permiten representar números enteros (BigInteger) o números con coma (BigDecimal) muy grandes, mayor a 64 bytes con precisión arbitraria.

No se pueden aplicar los operadores matemáticos convencionales como + o -, sino que tienen sus propios métodos.

Algunos métodos son los siguientes:

- **add (BigDecimal a):** suma.
- **subtract (BigDecimal a):** resta.
- **multiply (BigDecimal a):** multiplicar.
- **divide (BigDecimal, int roundMode):** división.
- **movePointRight(int posiciones):** mover la coma a la derecha.
- **movePointLeft(int posiciones):** mover la coma a la izquierda.

### Clase Arrays

La clase Arrays contiene métodos estáticos para realizar operaciones comunes con arreglos. Por ejemplo:

- **asList(T a):** transforma un arreglo en lista.
- **binarySearch(char[] a, char key):** realizar búsqueda binaria en el arreglo a.
- **copyOf(Object [] original, int newLength):** copia el arreglo original.
- **fill(long[] a, long val):** rellena el arreglo a con valores aleatorios.
- **sort(int[] a):** ordena el arreglo.

## Strings Mutables

En Java, el tipo de dato String es inmutable, es decir, que cuando realizamos un cambio en un string se crea una nueva instancia de la clase con el cambio realizado y el puntero pasa a referenciar el espacio en memoria de esta instancia.

Cuando se utilizan muchos métodos de modificación de Strings, el garbage collector (recolector de basura) pasará la mayor parte de su tiempo “recolectando” de la memoria todos los Strings no usados, lo cual genera grandes problemas de performance.

Para esto Java introdujo dos tipos de datos, StringBuilder (Java 5) y StringBuffer (Java 7).

La principal diferencia entre uno y otro es que StringBuffer fue hecho exclusivamente para multihilo, es decir para ejecuciones en paralelo, ya que sus métodos tienen sincronización para manejo de escritura concurrente. Stringbuilder es más performante, pero es más recomendable utilizarlo cuando se trabaja con un solo hilo.

Ambas clases tienen los mismos constructores y los mismos métodos.

## Constructores

- **StringBuilder():** crea un StringBuilder vacío con una capacidad de 16 caracteres.
- **StringBuilder(int capacity):** crea un StringBuilder vacío con una capacidad igual a la determinada por parámetro.
- **StringBuilder(String s):** crea un StringBuilder con el String especificado como parámetro.

## Algunos métodos

- **append(Object obj):** si obj es un String lo agrega al final del objeto actual. Si obj es otro objeto, primero llama a obj.toString() y lo agrega al final del objeto actual.
- **insert(int pos, String str):** agrega str en la posición especificada.
- **reverse():** se reemplaza el StrinBuilder por la misma secuencia de caracteres pero en orden inverso.
- **setCharAt(int pos, char c):** se reemplaza el carácter en la posición especificada por c.
- **remove(int ini, int fin):** se eliminan los caracteres desde la posición ini hasta la posición fin.

## Excepciones

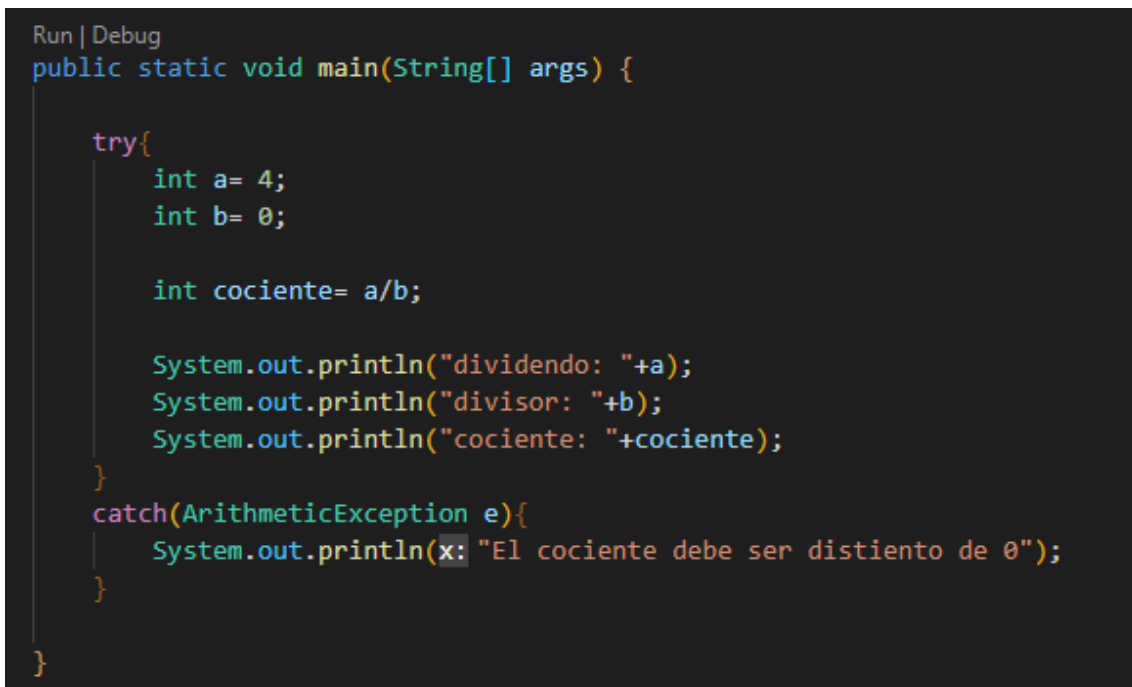
Una excepción es un cierto tipo de error o condición anormal que se produce durante la ejecución de un programa interrumpiendo el flujo normal del mismo.

Por ejemplo:

- Indexación incorrecta de un array.
- Acceso a un archivo no encontrado.
- División por cero.
- Acceso a variable antes de ser instanciada (java.lang.NullPointerException).

Es vital aprender a manejar las excepciones para crear un programa robusto que logre recuperarse de una situación anormal que interrumpa el flujo normal del mismo, permitiéndole volver a un estado estable.

Las excepciones permiten separar el código normal del código que se encarga del tratamiento del error.



```
Run | Debug
public static void main(String[] args) {

    try{
        int a= 4;
        int b= 0;

        int cociente= a/b;

        System.out.println("dividendo: "+a);
        System.out.println("divisor: "+b);
        System.out.println("cociente: "+cociente);
    }
    catch(ArithmeticException e){
        System.out.println(x: "El cociente debe ser distinto de 0");
    }

}
```

Todo lo que se encuentra dentro del bloque try es el flujo normal de la aplicación y el código dentro del bloque catch es el tratamiento del error.

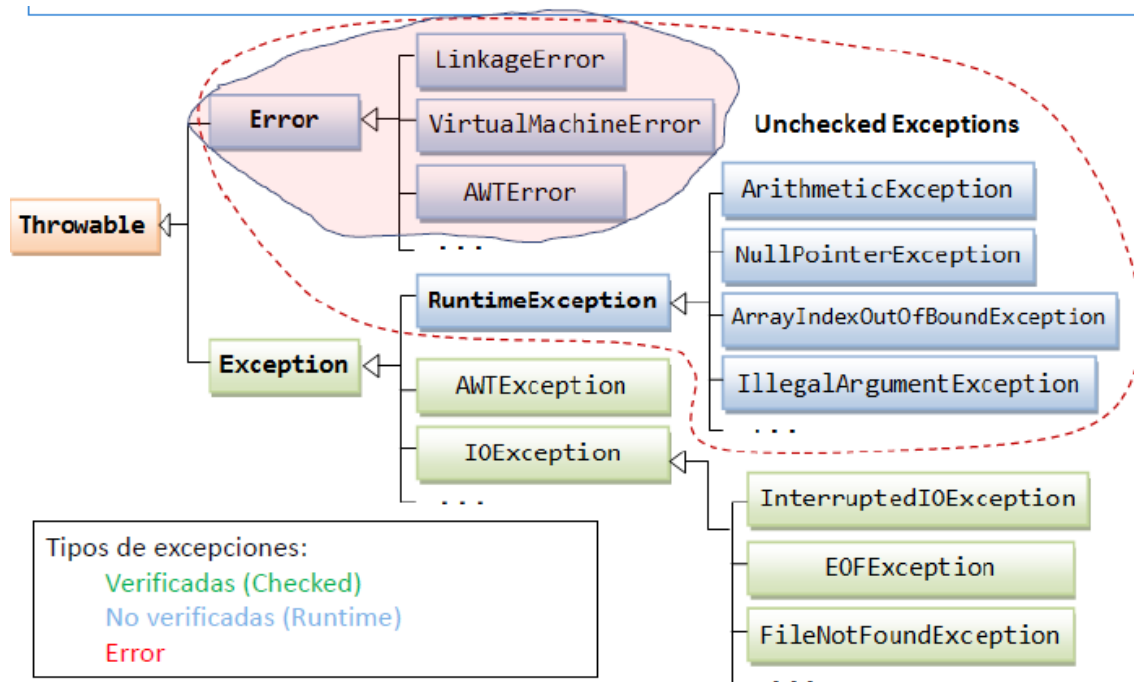
Las excepciones en Java son clases que extienden de java.lang.Throwable de, o de alguna de sus subclases.

Las subclases de Throwable son:

- java.lang.Error: situaciones anormales, usualmente irre recuperables, para las cuales no es necesario agregar algún tratamiento, por ejemplo un error en la JVM.
- java.lang.Exception: situaciones que impiden la ejecución correcta del código, pero que son (hasta cierto punto) previsibles y recuperables. Hay que incluir código para tratarlas.



## Tipo de excepciones



- **Verificadas/Comprobadas (checked):** son excepciones que deben ser manejadas por el desarrollador. Incluye todas las subclases `Exception`, excepto las de la jerarquía `RuntimeException`
- **No verificadas (Runtime):** Son excepciones evitables que deben ser corregidas en tiempo de desarrollo y debugging. Por ejemplo: `NullPointerException`.
- **Error:** se reserva para indicar problemas irrecurables, generalmente asociados con la máquina virtual de java.

`RuntimeException` es la superclase de aquellas excepciones que pueden ser lanzadas durante la operación normal de la JVM. No requiere de un código que las maneje, ya que la JVM tiene un comportamiento por defecto para estas excepciones. Este comportamiento por defecto interrumpe la ejecución del programa, en caso de que esto no sea lo deseado se deberá manejar la excepción.

## ¿Cómo declarar excepciones?

Crear excepciones propias carece de misterio alguno, es como extender cualquier otra clase.

Para crear una excepción propia se debe crear una clase y extenderla de `Exception` si queremos que sea verificable o `RuntimeException` si queremos

que sea no verificable. Dicha clase debe llevar un nombre representativo y por convención debe terminar con la palabra Exception.

Usualmente no se requiere ningún método adicional, pero si es necesario generar uno de los constructores.

```
public class MyException extends Exception {  
    public MyException(String mensaje){  
        super(mensaje);  
    }  
}
```

Los constructores de las excepciones tienen por parámetro el mensaje solo, la causa sola o ambos.

Extender de Exception nos proporciona la ventaja de que el compilador verificará por nosotros que no estemos olvidando manejar alguna situación hipotética que podría hacer dejar de funcionar el programa. Sin embargo, tienen por desventaja que habitualmente pasan a formar parte de la interfaz del método y agregan complejidad.

En la mayoría de los códigos se verán implementaciones que extienden de Exception por la ventaja ya mencionada.

### Lanzar Excepciones

La sentencia throw se utiliza para lanzar excepciones y requiere un argumento de tipo Throwable o de alguna de sus subclases.

Por ejemplo:

```
throw new ArrayIndexOutOfBoundsException("Índice fuera de rango");
```

Si un método lanza una excepción que no extiende de RuntimeException debe ser manejada con bloques try – catch, o bien debe indicar en su firma que lanza dicha excepción propagando la misma hacia el método que lo invocó.

Por ejemplo:

Con RuntimeException

```
public Integer get(int indice){  
    if(indice >= lista.size()){  
        throw new IndexOutOfBoundsException(s: "Indice fuera de rango");  
    }  
    return indice;  
}
```

Con bloque try - catch

```
public Integer get(int indice){  
    if(indice >= lista.size()){  
        try {  
            throw new MyException(mensaje: "Indice fuera de rango");  
        } catch (MyException e) {  
            System.out.println(e.getMessage());  
        }  
    }  
  
    return indice;  
}
```

Propagando la excepción

```
public Integer get(int indice) throws MyException{  
    if(indice >= lista.size()){  
        throw new MyException(mensaje: "Indice fuera de rango");  
    }  
  
    return indice;  
}
```

```
try {  
    ejemplo.get(indice: 5);  
} catch (MyException e) {  
    System.out.println(e.getMessage());  
}
```

### Capturar excepciones

Se requieren tres bloques para capturar excepciones. Try, catch y finally, este último bloque es opcional.

El bloque try es donde se coloca el bloque de código donde se puede llegar a lanzar la excepción. El bloque try debe ir acompañado de al menos un bloque catch o un bloque finally.

El bloque catch se ejecuta cuando se lanza una excepción del tipo indicado en el bloque catch.

```
try {
    ejemplo.get(indice: 5);
} catch (MyException e) {
    System.out.println(e.getMessage());
}
```

En este caso el println se ejecuta si se detecta una excepción del tipo MyException pero no una excepción de otro tipo.

Para un bloque try puede haber más de un bloque catch. Por ejemplo:

```
try {
    ejemplo.get(indice: 5);
} catch (MyException e) {
    System.out.println(e.getMessage());
}
catch (Exception e) {
    e.printStackTrace();
}
```

A partir de java 7 se agregó el operador pipe “|” que permite agrupar varias excepciones en un único bloque catch y por ende dichas excepciones serán tratadas de la misma forma.

```
try {
    ejemplo.get(indice: 5);
} catch (MyException | IndexOutOfBoundsException e) {
    System.out.println(e.getMessage());
}
catch (Exception e) {
    e.printStackTrace();
}
```

El orden de captura de los catch es desde el más específico al menos específico, es decir desde la subclase a la superclase.

```
try {
} catch (FileNotFoundException e1) {
} catch (IOException e2) {
}
```

Correcto porque FileNotFoundException es un posible tipo de IOException

```
try {
} catch (IOException e1) {
} catch (FileNotFoundException e2) {
}
```

Incorrecto porque siempre que ocurra un FileNotFoundException será manejado por IOException dado que está antes y coincide

El bloque finally es opcional, pero si se utiliza siempre debe estar acompañado de un bloque try. Este bloque de código se ejecuta siempre, se produzca o no una excepción. Usualmente se lo utiliza para limpiar variables, métodos o liberar recursos, por ejemplo, cerrar la conexión a la base de datos, liberar un buffer, etc.

Dentro de un bloque finally se puede producir una excepción que no le permite ejecutarse por completo.

## Opcionales

En muchas ocasiones nos encontramos con situaciones en las que un valor puede devolver nulo. Ante esta situación, los programadores están obligados a comprobar si la variable es nulo (null) antes de acceder a su valor. Ya que en el caso de ser nula e intentar acceder a algunas de sus propiedades, el programa falla y lanza una excepción de java.lang.NullPointerException.

Para lidiar de mejor manera con este problema se incluyeron en Java 8 cuatro clases: Optional, OptionalDouble, OptionalInt y OptionalLong.

Estas clases permiten manejar valores que pueden estar presentes o no, sin necesidad de gestionar el valor null. Toda la complejidad del manejo de los valores nulo queda encapsulada dentro de los métodos de estas clases.

### Métodos para inicializar un Optional

- **public static** Optional<T> **empty()**: crea un Opcional vacío.
- **public static** Optional<T> **ofNullable**(T value): crea un Optional con valor nulo. Este opcional también se considera vacío.
- **public static** Optional<T> **of**(T value): crea un Opcional no vacío

### Métodos para determinar si un Optional está vacío

- **boolean isPresent()**: retorna true si el Optional no está vacío
- **boolean isEmpty()**: retorna true si el Optional está vacío

El método isEmpty está disponible a partir de Java 11

### Métodos para obtener el valor

- T **get()**: retorna el valor, si el mismo está presente, o lanza NoSuchElementException.
- T **orElse**(T other): si en un optional un valor está presente , entonces retorna ese valor, sino retorna otro.
- T **orElseThrow()**: si el valor está presente lo retorna, caso contrario lanza una excepción NoSuchElementException.

Con orElseThrow podemos pasar una función anónima como parámetro para retornar una excepción particular, por ejemplo:

```
Optional<Integer> op= Optional.empty();  
op.orElseThrow(() -> new Exception(message: "Excepción lanzada por orElseThrow"));
```

```
Exception in thread "main" java.lang.Exception: Excepción lanzada por orElseThrow
```

## Api de Fecha y Hora

Hasta la versión 8 de java para trabajar con fechas, teníamos 3 clases: Date que representa un punto en el tiempo, Calendar y su subclase GregorianCalendar, que traducen el punto en el tiempo a su representación en un calendario particular (el gregoriano es el más extendido en su uso). Y luego, para formatear la clase en la consola o para convertir de texto a Date, se usa java.text.DateFormat.

### Date

Crea un objeto del tipo Date y lo representa como un long que cuenta los milisegundos que pasaron desde el 1/1/1970.

La mayoría de los métodos están marcados como Deprecated y no deberían continuar utilizándose.

El gran defecto de la clase Date es que se basaba en el formato americano de fecha y no se permitía el uso de formato de fechas de otros países.

Para solucionar esto se crea la clase Calendar y DateFormat.

### Clase Calendar

Calendar es una clase base abstracta para obtener y setear campos de una fecha como Año, Mes, Día, Hora, etc.

**getInstance()** es un método que nos permite obtener una instancia de GregorianCalendar e inicializa cada campo de la fecha con los valores de fecha actual.

Calendar provee métodos para modificar una fecha, cambiando alguno de sus componentes

```
DateFormat format= new SimpleDateFormat(pattern: "dd/MM/yyyy");  
Calendar calendar= Calendar.getInstance();  
  
System.out.println("Fecha actual: "+format.format(calendar.getTime()));  
  
calendar.add(Calendar.DATE, amount: 5);  
  
System.out.println("Fecha 5 días despues: "+format.format(calendar.getTime()));
```

```
Fecha actual: 27/02/2023  
Fecha 5 días despues: 04/03/2023
```

### Ejemplo de seteo de fecha

```
DateFormat format= new SimpleDateFormat(pattern: "dd/MM/yyyy");

int dia= 25, mes= 12, anio= 2022;

Calendar navidad= Calendar.getInstance();

navidad.set(Calendar.YEAR, anio);
navidad.set(Calendar.MONTH, mes-1); //los meses se numeran a partir de 0
navidad.set(Calendar.DAY_OF_MONTH, dia);
navidad.set(Calendar.HOUR, value: 0);
navidad.set(Calendar.MINUTE, value: 0);
navidad.set(Calendar.SECOND, value: 0);

Date navidadDate= navidad.getTime();

System.out.println(format.format(navidadDate));
```

25/12/2022

### Ejemplo de GregorianCalendar

```
Date fechaActual= new Date();

Calendar c= new GregorianCalendar();
c.setTime(fechaActual);

System.out.println("Año: "+c.get(Calendar.YEAR));
System.out.println("Mes: "+c.get(Calendar.MONTH));
System.out.println("Día: "+c.get(Calendar.DAY_OF_MONTH));
```

Año: 2023  
Mes: 1  
Día: 27

### DateFormat

En ejemplos anteriores vimos aplicaciones de DateFormat.

La clase DateFormat es una clase abstracta y su implementación es SimpleDateFormat.

Esta clase tiene dos métodos, **format** que permite formatear un Date en un String y **parse** que permite transformar un String en un Date.

El formato se define en el constructor con un String. Las letras que se utilizan son las siguientes:

d	Día
---	-----

<b>M</b>	Mes
<b>y</b>	Año
<b>MMM</b>	Abreviatura del mes de tres letras
<b>MMMM</b>	Mes en letras
<b>EEEE</b>	Día en letras
<b>EEE</b>	Abreviatura del día en letras
<b>H</b>	Hora
<b>m</b>	Minutos
<b>s</b>	Segundos

El mes y día en letra lo muestra en el idioma del sistema operativo. Pero podemos cambiarlo agregándole un segundo parámetro al constructor.

Locale nos permite cambiar el idioma determinando el país o el idioma, por ejemplo, si queremos cambiar de español a inglés podemos hacer lo siguiente.

Fecha en español:

```
DateFormat dateFormat= new SimpleDateFormat(pattern: "EEEE MMMM");
System.out.println(dateFormat.format(new Date()));
```

lunes febrero

Fecha en inglés determinado el país:

```
DateFormat dateFormat= new SimpleDateFormat(pattern: "EEEE MMMM", Locale.UK);
System.out.println(dateFormat.format(new Date()));
```

Monday February

UK → United Kingdom

Fecha en francés determinando idioma:

```
DateFormat dateFormat= new SimpleDateFormat(pattern: "EEEE MMMM", Locale.FRENCH);
System.out.println(dateFormat.format(new Date()));
```

lundi février



## Api java.time

Como se mencionó anteriormente, la clase Date tiene muchos métodos que han sido marcados como deprecados, la mayoría de ellos cuando se introdujo la clase Calendar. Pero calendar tenía otro problema y es que no resolvía correctamente los segundos bisiestos.

Por esto, a partir de Java 8 se agregó java.time que resuelve estos problemas, y además, los métodos están mejor definidos, permitiendo un comportamiento más claro y esperado. También es posible encadenar los métodos, por ejemplo:

```
LocalDate yesterday= LocalDate.now().minusDays(daysToSubtract: 2).plusDays(daysToAdd: 1);
```

## Instant

Un Instant representa un punto en la línea de tiempo, es equivalente a Date en las versiones anteriores de Java.

Un origen, llamado época, se establece arbitrariamente a la media noche del 1 de enero de 1970 en el meridiano de Greenwich. A partir de ese origen, el tiempo se mide en 86400 segundos por día, hacia adelante y hacia atrás.

Instant puede almacenar hasta mil millones de años hacia adelante o hacia atrás a partir de 1970.

Para crear un Instant tenemos los métodos static, **now()** o el método **ofEpoch()** que permite crear un instant desde una cantidad de milisegundos.

```
System.out.println(Instant.now());  
System.out.println(Instant.ofEpochMilli(System.currentTimeMillis()));  
System.out.println(Instant.ofEpochMilli(System.currentTimeMillis()-1000*60*60*24*5));
```

```
2023-02-27T15:05:43.771Z  
2023-02-27T15:05:43.861Z  
2023-02-22T15:05:43.861Z
```

Los métodos **plus** y **minus** permiten aumentar o disminuir los instants. Estos están sobrecargados para poder usarlos en milisegundos o estableciendo la unidad y el valor.

## Duration

La clase Duration, es una clase que modela una duración de tiempo en términos de segundos y milisegundos y nos puede permitir entre otras cosas almacenar la diferencia entre 2 instantes.

Además, se puede crear con diversas versiones de “of” como un período de N horas, días, minutos, segundos.

```

Instant start= Instant.now();
Thread.currentThread().sleep(millis: 3452);
Instant end= Instant.now();

Duration duration= Duration.between(start, end);

System.out.println(duration.getSeconds()+" "+ duration.toMillis());
3: 3452

```

### Algunos métodos de Duration

- **static** Duration **ofMinutes**(long minutos): crea un Duration con el tiempo pasado por parámetro. Además de minutes existe millis, seconds, hours y days.
- **static** Duration **between**(Temporal start, Temporal end): crea un Duration con la diferencia entre end y start.
- boolean **isZero**(): retorna true si la duración es cero.
- **boolean isNegative**(): retorna true si la duración es negativa.

### Local Date y Times

LocalDate representa una fecha, como el 20 de enero.

LocalTime representa un tiempo sin ninguna información de fecha, como una alarma que suena a las 7:15 cada mañana.

LocalDateTime almacena valores de fecha y hora.

Para crear instancias de estas clases tenemos variantes de los métodos **now**, **parse** y **of**

```

System.out.println(LocalDate.of(year: 2019, month: 5, dayOfMonth: 4));
System.out.println(LocalDate.parse(text: "2019-05-04"));
System.out.println(LocalDate.now());
System.out.println(LocalDate.ofYearDay(year: 2020, dayOfYear: 96));

System.out.println(LocalTime.of(hour: 7, minute: 15));
System.out.println(LocalTime.parse(text: "07:15"));
System.out.println(LocalTime.now());
System.out.println(LocalTime.of(hour: 22, minute: 45, second: 53));

System.out.println(LocalDateTime.of(year: 2019, month: 5, dayOfMonth: 4, hour: 7, minute: 0));
System.out.println(LocalDateTime.parse(text: "2019-05-04T07:15"));

```

```
2019-05-04
2019-05-04
2023-02-27
2020-04-05
07:15
07:15
12:47:34.244
22:45:53
2019-05-04T07:00
2019-05-04T07:15
```

### Parseo y Formateo de fechas

***java.time.format.DateTimeFormatter*** permite crear String con un patrón que se utilizan para tanto para el parsing (String → Fecha) como para la conversión a String (Fecha → String)

Posee tres grupos de formatos:

- Predefinidos: posee un conjunto de enumerados para los formatos predefinidos.
- Específicos de una localización: posee un conjunto de enumerados que definen formatos de fecha por zona.
- Personalizados: los formatos personalizados se crean con Strings con patrones iguales a los de SimpleDateFormat.

Ejemplo fecha a texto:

```
LocalDate anotherSummerDay = LocalDate.of(year: 2018, month: 1, dayOfMonth: 23);
String full= DateTimeFormatter.ofLocalizedDate(FormatStyle.FULL).format(anotherSummerDay);
String lon= DateTimeFormatter.ofLocalizedDate(FormatStyle.LONG).format(anotherSummerDay);
String medium= DateTimeFormatter.ofLocalizedDate(FormatStyle.MEDIUM).format(anotherSummerDay);
String shor= DateTimeFormatter.ofLocalizedDate(FormatStyle.SHORT).format(anotherSummerDay);

System.out.println(full);
System.out.println(lon);
System.out.println(medium);
System.out.println(shor);
```

```
martes 23 de enero de 2018
23 de enero de 2018
23/01/2018
23/01/18
```

Ejemplo texto a fecha:

```
DateTimeFormatter formatter = DateTimeFormatter.ofPattern(pattern: "yyyyMM-ddHHmm");
String fecha1 = "201907-222019";
LocalDate date = LocalDate.parse(fecha1, formatter);
System.out.println(date);
LocalDateTime dateTime = LocalDateTime.parse(fecha1,formatter);
System.out.println(dateTime);
```

```
2019-07-22
2019-07-22T20:19
```

## Api Stream

El api de colecciones tiene un diseño flexible que permite sacar ventaja del uso de grupos de datos en diferentes contextos con diferentes necesidades. Pero está lejos de ser perfecto, por ejemplo, para buscar el máximo de una colección no es posible mediante un método, sino que se debe escribir una secuencia lógica. Por ello, en Java 8 se agregó el Api Stream.

El concepto principal introducido es el de Stream que representa un flujo de datos que se pueden analizar de forma secuencial. Un Stream no es una estructura de datos que almacena datos, sino que toma los elementos de una estructura y los pasa por un canal para realizar transformaciones y operaciones.

Las operaciones sobre el flujo se ejecutan en forma lazy (perezosa), es decir, se difiere su ejecución hasta que se invoca una operación terminal, de esta forma los elementos que no son necesarios para completar las operaciones nunca se calculan.

## Pipeline

Las operaciones del Api Stream se dividen en operaciones intermedias y operaciones finales. Las operaciones intermedias permiten combinar el resultado con otra operación y refinar el trabajo. Las operaciones terminales pueden producir un resultado o modificar un dato. Luego de que se ejecuta una operación terminal el Stream está consumido y no se pueden ejecutar más operaciones.

La combinación de N operaciones intermedias y una operación terminal se llama Stream Pipeline o simplemente pipeline.

Ejemplo:

```
lista.stream()
    .filter(t -> t.getTipo() == 5)
    .sorted((t1, t2) -> t1.getValue().compareTo(t2.getValue()))
    .map(Transaccion::getId)
    .collect(Collectors.toList());
```

Como puede verse filter, sorted y map son operaciones intermedias y collect es la operación terminal que retorna una lista con el resultado de las operaciones. En el ejemplo, se puede ver el operador de dos puntos :: que es un operador de referencia de método. Este se usa para llamar un método referenciando al mismo directamente con su clase. Funciona de la misma forma que una expresión lambda, la única diferencia es que este es una referencia directa a un método. Se puede utilizar tanto para métodos estáticos como para objetos.

## Operaciones intermedias

### **Filter**

Retorna un flujo que retiene los elementos que cumplen con una condición

`Stream<T> filter(Predicate<? super T> predicate)`

```
List<Integer> numeros = Arrays.asList(...a: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

List<Integer> numerosPares = numeros.stream()
    .filter(n -> n % 2 == 0)
    .collect(Collectors.toList());

System.out.println("Numeros pares: " + numerosPares);

Numeros pares: [2, 4, 6, 8, 10]
```

### **Map y FlatMap**

Map es una función muy útil y permite aplicar una función de transformación para mapear los elementos de un flujo a otro tipo de elementos

<code>&lt;R&gt; Stream&lt;R&gt;</code>	<code>map(Function&lt;? super T,? extends R&gt; mapper)</code>	Returns a stream consisting of the results of applying the given function to the elements of this stream.
<code>DoubleStream</code>	<code>mapToDouble(ToDoubleFunction&lt;? super T&gt; mapper)</code>	Returns a DoubleStream consisting of the results of applying the given function to the elements of this stream.
<code>IntStream</code>	<code>mapToInt(ToIntFunction&lt;? super T&gt; mapper)</code>	Returns an IntStream consisting of the results of applying the given function to the elements of this stream.
<code>LongStream</code>	<code>mapToLong(ToLongFunction&lt;? super T&gt; mapper)</code>	Returns a LongStream consisting of the results of applying the given function to the elements of this stream.

La operación flatMap y sus variantes ToDouble, ToInt y ToLong permiten transformar cada elemento del Stream en otro Stream y al final concatenarlos todos en uno solo. Es útil cuando tenemos atributos del tipo lista o colección sobre los que queremos iterar, dado que tendremos un `Stream<Stream<R>>` y con flatMap podemos convertirlo en `Stream<R>`

```
List<Persona> personas = Arrays.asList(
    new Persona(nombre: "Juan", Arrays.asList(...a: "111-1111", "111-2222")),
    new Persona(nombre: "Maria", Arrays.asList(...a: "222-1111", "222-2222")),
    new Persona(nombre: "Pedro", Arrays.asList(...a: "333-1111", "333-2222")));

List<String> numerosDeTelefono = personas.stream()
    .map(p -> p.getNumerosDeTelefono())
    .flatMap(n -> n.stream())
    .collect(Collectors.toList());

System.out.println("Numeros de telefono: " + numerosDeTelefono);
```

```
Numeros de telefono: [111-1111, 111-2222, 222-1111, 222-2222, 333-1111, 333-2222]
```

### ***Distinct***

Retorna un Stream sin elementos repetidos. Depende de la implementación del método equals que tengamos.

```
List<Integer> numeros = Arrays.asList(...a: 1, 2, 3, 4, 5, 1, 2, 3, 4, 5);

List<Integer> numerosUnicos = numeros.stream()
    .distinct()
    .collect(Collectors.toList());

System.out.println("Numeros unicos: " + numerosUnicos);
```

```
Numeros unicos: [1, 2, 3, 4, 5]
```

### ***Skip y Limit***

- Limit: Retorna un Stream cuyo tamaño no es mayor al número pasado por parámetro. Los elementos son limitados hasta ese tamaño.
- Skip: Retorna un Stream que descarta los primeros N elementos, donde N es el número pasado por parámetro. Si el Stream contiene menos elementos que N, entonces retorna un Stream vacío.

```
List<Integer> numeros = Arrays.asList(...a: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

List<Integer> primerosTresNumeros = numeros.stream()
    .limit(maxSize: 3)
    .collect(Collectors.toList());

List<Integer> ultimosCincoNumeros = numeros.stream()
    .skip(n: 5)
    .collect(Collectors.toList());

System.out.println("Primero tres números: " + primerosTresNumeros);
System.out.println("Ultimos cinco núemros: " + ultimosCincoNumeros);
```

```
Primero tres números: [1, 2, 3]
Ultimos cinco núemros: [6, 7, 8, 9, 10]
```

## Sorted

Retorna un stream ordenado de acuerdo a la expresión lambda que se le pasa por parámetro.

```
List<Persona> personas = Arrays.asList(  
    new Persona(nombre: "Juan", edad: 20),  
    new Persona(nombre: "Maria", edad: 30),  
    new Persona(nombre: "Pedro", edad: 15),  
    new Persona(nombre: "Ana", edad: 25),  
    new Persona(nombre: "Luis", edad: 18));  
  
List<String> nombresMayoresDe18 = personas.stream()  
    .filter(p -> p.getEdad() > 18)  
    .sorted((p1, p2) -> p1.getNombre().compareTo(p2.getNombre()))  
    .map(Persona::getNombre)  
    .collect(Collectors.toList());  
  
System.out.println("Nombres mayores de 18: " + nombresMayoresDe18);  
Nombres mayores de 18: [Ana, Juan, Maria]
```

## Operaciones Terminales

### Collectors

El api de Stream ofrece el método collect que recibe como argumento un método static de la clase Collectors.

La clase Collectors tiene implementación de numerosos métodos que permiten sumarizar, o realizar operaciones terminales de reducción sobre el stream y retornarlo.

Por ejemplo:

```
List<Persona> personas = Arrays.asList(  
    new Persona(nombre: "Juan", edad: 30),  
    new Persona(nombre: "Maria", edad: 25),  
    new Persona(nombre: "Pedro", edad: 30),  
    new Persona(nombre: "Carlos", edad: 25));  
  
Map<Integer, List<Persona>> personasPorEdad = personas.stream()  
    .collect(Collectors.groupingBy(Persona::getEdad));  
  
System.out.println("Personas agrupadas por edad: " + personasPorEdad);  
Personas agrupadas por edad: {25=[Maria, Carlos], 30=[Juan, Pedro]}
```

Todos los métodos de collectors se encuentran en el siguiente link de la documentación de Java

<https://docs.oracle.com/javase/8/docs/api/java/util/stream/Collectors.html>

## ForEach

Es una operación terminal que ejecuta una acción por cada elemento de la secuencia.

```
Tarea t1 = new Tarea();
Tarea t2 = new Tarea();
Tarea t3 = new Tarea();

t1.setFechaEstimada(LocalDate.now().minus(amountToSubtract: 5, ChronoUnit.DAYS));
t2.setFechaEstimada(LocalDate.now().plus(amountToAdd: 5, ChronoUnit.DAYS));
t3.setFechaEstimada(LocalDate.now().minus(amountToSubtract: 2, ChronoUnit.DAYS));
t3.setFechaFin(LocalDate.now());

List<Tarea> listaTareas = Arrays.asList(t1, t2, t3);

listaTareas.stream()
    .map(Tarea::getAtrasada)
    .forEach(System.out::println);

listaTareas.stream()
    .forEach(t -> t.setAtrasada(t.getFechaFin() == null &&
        t.getFechaEstimada().isAfter(LocalDate.now())));

listaTareas.stream()
    .map(Tarea::getAtrasada)
    .forEach(System.out::println);
```

Antes de setear si está atrasada

```
null
null
null
```

Luego de setear si está atrasada

```
false
true
false
```



### Otras operaciones finales

- Count: retorna la cantidad de elementos del stream
- Max y min: retornan un Optional con el máximo o mínimo elemento de un stream basado en un comparado basado por parámetro.

```
List<Empleado> area1= Arrays.asList(
    new Empleado(nombre: "Martin", sueldo: 1000.0, horasExtras: 5, antiguedad: 11),
    new Empleado(nombre: "Marta", sueldo: 500.0, horasExtras: 0, antiguedad: 15),
    new Empleado(nombre: "Tony", sueldo: 1250.0, horasExtras: 0, antiguedad: 25)
);

List<Empleado> area2= Arrays.asList(
    new Empleado(nombre: "Martin2", sueldo: 2000.0, horasExtras: 4, antiguedad: 8),
    new Empleado(nombre: "Marta2", sueldo: 600.0, horasExtras: 6, antiguedad: 12),
    new Empleado(nombre: "Tony2", sueldo: 1500.0, horasExtras: 9, antiguedad: 18)
);

List<List<Empleado>> empresa = Arrays.asList(area1, area2);

Long cantidadEmpleados = empresa.stream()
    .flatMap(List::stream)
    .count();

System.out.println("Hay " + cantidadEmpleados + " empleados");

Optional<Empleado> empleadoConMasAntiguedad = empresa.stream()
    .flatMap(List::stream)
    .max(Comparator.comparingInt(Empleado::getAntiguedad));

System.out.println("El empleado con más antiguedad es: " + empleadoConMasAntiguedad.get());

Optional<Empleado> empleadoConPeorPaga = empresa.stream()
    .flatMap(List::stream)
    .min(Comparator.comparingDouble(Empleado::getSueldo));

System.out.println("El empleado con menor paga es: " + empleadoConPeorPaga.get());
```

Hay 6 empleados  
El empleado con más antiguedad es: { nombre='Tony', sueldo='1250.0', horasExtras='0', antiguedad='25'}  
El empleado con menor paga es: { nombre='Marta', sueldo='500.0', horasExtras='0', antiguedad='15'}

### Operaciones terminales de búsqueda

Podemos en lugar de verificar condiciones para producir un nuevo stream con “filter” usar operaciones finales, que retornan un valor boolean si se cumplen determinadas condiciones para “todos”, “alguno”, o “ninguno” de los elementos del stream.

- **allMatch:** Verifica si todos los elementos del Stream satisfacen el predicado pasado por parámetro.
- **anyMatch:** Verifica si alguno de los elementos del Stream satisface el predicado pasado por parámetro.
- **noneMatch:** verifica si todos los elementos del Stream NO satisfacen el predicado pasado por parámetro.

Si durante la verificación alguno lo cumple entonces se detiene la verificación y retorna verdadero, es decir, no requiere procesar todo el Stream para producir el resultado.

```
System.out.println("Salario mayor a 300: " +
    empresa.stream().flatMap(List::stream)
        .allMatch(t -> {
            return t.getSueldo() > 300.0;
        }));

System.out.println("Alguno gana menos de 1000: " +
    empresa.stream().flatMap(List::stream)
        .anyMatch(t -> {
            return t.getSueldo() < 1000.0;
        }));

System.out.println("Nadie gana más de 5000: " +
    empresa.stream().flatMap(List::stream)
        .noneMatch(t -> {
            return t.getSueldo() > 5000.0;
        }));
```

```
Salario mayor a 300: true
Alguno gana menos de 1000: true
Nadie gana más de 5000: true
```

Otras operaciones de búsqueda son:

- `findAny`: retorna algún elemento del Stream.
- `findFirst`: retorna el primer elemento del Stream.

## Reduce

La expresión “reduce” permite reducir el stream a un único valor, con lo visto hasta aquí podemos reducirlo a un valor booleano o a un valor cualquiera del stream, pero mediante el método “reduce” lo podemos reducir a un valor personalizado pasando la lógica de esta reducción como parámetro.

Reduce recibe dos parámetros:

- **Identity**: es la identidad del elemento, el valor inicial y por defecto si el stream está vacío.
- **Acumulador**: función que toma dos parámetros, el resultado parcial de la reducción y el siguiente elemento del Stream. Retorna un nuevo resultado parcial

```
List<Integer> numbers = Arrays.asList(...a: 1, 2, 3, 4, 5);  
  
int sum = numbers.stream().reduce(identity: 0, (a, b) -> a + b);  
  
System.out.println("La suma de todos los números es: " + sum);
```

La suma de todos los números es: 15