



Java - Review



Java 7



Antes un cambio de Java 7

En Java < 7 cada Closeable, debía cerrarse, generalmente en un finally

```
public final String readFirstLineFromFile(String path) throws IOException {
    BufferedReader br = new BufferedReader(new FileReader(path));
    try {
        return br.readLine();
    } finally {
        if (br != null) {
            br.close();
        }
    }
}
```

Y el bendito null check!!!

Posible "resource leak"

Para hacer esto menos engorroso se incorporó el [Try with resources](#)

```
public final String readFirstLineFromFile(String path) throws IOException {
    try (BufferedReader br = new BufferedReader(new FileReader(path))) {
        return br.readLine();
    }
}
```



Java 8



Listado de Cambios

- » Functional interfaces
- » Lambdas
- » Inference
- » Method Reference
- » Default Method
- » Static Method
- » Optional
 - ◇ map/flatMap
 - ◇ or
- » Collections
 - ◇ Stream
 - ◇ Date-Time package
- » Concurrency

[Listado completo de cambios.](#)



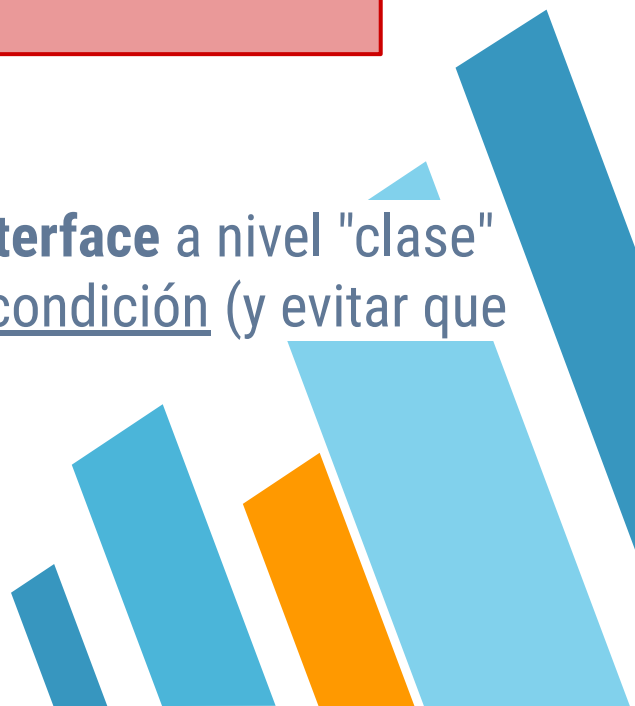


Interfaz funcional

Una interfaz funcional (También llamadas Single Abstract Method) es una interfaz que tiene **un solo método “abstracto” o a implementar.**

abstracto: porque no cuenta métodos defaults, estáticos y los definidos en objects

Opcionalmente se le puede agregar **@FunctionalInterface** a nivel "clase" para que el compilador chequee que se cumple la condición (y evitar que algún futuro cambio rompa esa condición).






Interfaz funcional

```
@FunctionalInterface
public interface MyFunctionalInterface<T> {

    boolean apply(T t);

}
```





Interfaz funcional

Muchas clases ya existentes se pueden utilizar como interfaces funcionales:

- » `java.lang.Runnable`
 - » `java.util.concurrent.Callable`
 - » `java.security.PrivilegedAction`
 - » `java.util.Comparator`
 - » `java.io.FileFilter`
 - » `java.beans.PropertyChangeListener`
- 



Interfaz funcional

Nuevo paquete [java.util.function](#) que contiene algunas interfaces funcionales útiles:

- » [Predicate](#)<T> -- a boolean-valued property of an object
- » [Consumer](#)<T> -- an action to be performed on an object
- » [Function](#)<T,R> -- a function transforming a T to a R
- » [Supplier](#)<T> -- provide an instance of a T (such as a factory)
- » [UnaryOperator](#)<T> -- a function from T to T
- » [BinaryOperator](#)<T> -- a function from (T, T) to T

y otras comunes un poco más específicas como LongPredicate y BooleanSupplier



Interfaz funcional

```
@FunctionalInterface
public interface Predicate<T> {


    /**
     * Evaluates this predicate on the given argument.
     *
     * @param t the input argument
     * @return {@code true} if the input argument matches
     * the predicate, otherwise {@code false}
     */
    boolean test(T t);
}
```



Static Method

Una interfaz se pueden definir métodos estáticos, que sirven de métodos de utilidad, por ejemplo Factories de nuevas instancias por composicion.

```
public interface Predicate<T> {  
    ....  
    static <T> Predicate<T> not(Predicate<? super T> target) {  
        Objects.requireNonNull(target);  
        return (Predicate<T>)target.negate();  
    }  
}
```



Default Method

Es un método con cuerpo que se puede agregar a una interfaz para ofrecer un comportamiento default a un método de la misma

```
@FunctionalInterface
```

```
public interface Predicate<T> {
```

```
    default Predicate<T> and(Predicate<? super T> other) {  
        Objects.requireNonNull(other);  
        return (t) -> test(t) && other.test(t);  
    }
```

```
    default Predicate<T> negate() {  
        return (t) -> !test(t);  
    }
```

```
}
```

De esta manera se le puede agregar funcionalidad a una jerarquía existente sin tener que modificar todas las implementaciones (ver Collections)

Default Method - Herencia

Al extender una interfaz con métodos default una clase/interfaz puede:

1. No mencionar al método, por lo cual se utilizará el default definido:

```
public interface LazyComparator<T> extends Comparator<T> {}
```

2. Redefinir el método sin un cuerpo transformando el método en abstracto para quien herede.

```
public interface AbstractComparator<T> extends Comparator<T> {  
    Comparator<T> reversed();  
}
```

3. Overridear esa implementación con una nueva (default si el que hereda es otra interfaz)

```
public interface OverrideComparator<T> extends Comparator<T> {  
    default Comparator<T> reversed() {return this; }  
}
```

Default Method - Conflictos Herencia

En caso de que dos clases definan el mismo método default las reglas de resolución del conflicto son:

1. Si son una clase y una interfaz, toma precedencia el método de la clase.

```
class QueuedList extends ArrayList implements  
Queue<String>
```

QueuedList usa el removeAll de ArrayList, no el de Queue (que lo hereda de Collection)

2. Si son dos interfaces (del mismo árbol de herencia) toma precedencia el que está más abajo en el árbol (porque por su rama pisó al superior)

```
class LinkedList<E> implements List<E>, Queue<E>
```

LinkedList usa el removeAll de List (que overrideo el de Collection) y no el de Queue (que lo hereda de Collection)

Default Method - Conflictos Herencia

3. Si dos interfaces independientes definen el mismo default method, la clase que hereda debe overridear el método:

```
public interface OperateCar {  
    // ...  
    default public int  
    startEngine(EncryptedKey key) {  
        // Implementation  
    }  
}
```

```
public interface FlyCar {  
    // ...  
    default public int  
    startEngine(EncryptedKey key) {  
        // Implementation  
    }  
}
```

```
public class FlyingCar implements OperateCar, FlyCar {  
    // ...  
    public int startEngine(EncryptedKey key) {  
        FlyCar.super.startEngine(key);  
        OperateCar.super.startEngine(key);  
    }  
}
```

FlyCar.super.startEngine es como se llama al default method de Flycar.



Method reference


Puedo asignar a interfaces referencias a métodos existentes.
Estas referencias pueden ser a:

- » **Un método estático** de una clase (ClassName::methodName)

```
final IntBinaryOperator adder = Integer::sum;  
adder.applyAsInt(2, 3);
```

- » **Un método de instancia** de una instancia específica (instanceRef::methodName)

```
Set<String> knownNames = ...  
Predicate<String> isKnown = knownNames::contains;  
isKnown.test("value");
```



Method reference

- » **Un método de instancia** sobre un objeto no especificado aún (ClassName::methodName)

```
Function<String, String> upperfier = String::toUpperCase;  
upperfier.apply("hello");
```

- » El constructor de una clase (ClassName::new)

```
Set<Person> rosterSet = transferElements(roster,  
HashSet<Person>::new);
```

- » El constructor de un array de cierta clase (TypeName[]::new)

```
IntFunction<int[]> arrayMaker = int[]::new;  
int[] array = arrayMaker.apply(10); // creates an int[10]
```



Lambdas

Expresiones Lambda

Permite código más simple, sin tanta "burocracia" Entonces se pasa de:

```
EventQueue.invokeLater(  
    new Runnable() {  
        @Override  
        public void run() {  
            System.out.println("Running");  
        }  
    }  
);
```

A que se pueda escribir:


```
EventQueue.invokeLater(() -> System.out.println("Running"));
```



Expresiones Lambda

Una expresión lambda expresa una "instancia" de una interfaz funcional. Para lo cual implementa el único "método abstracto" de la misma.

Las expresiones permiten

- » Crear funciones sin la necesidad de crear una clase que la contenga.
 - » Asignarlas a variables y pasarlas como argumentos de funciones.
 - » Ejecutar su funcionalidad "on demand" en runtime.
- 

Expresiones Lambdas - Sintaxis

Parámetros

```
(String x, int y) -> {  
    System.out.println("recibi" + x);  
    return y + 1;  
}
```

Cuerpo

Sintaxis Sugars:

- `(x, y) -> x + y` ← No es necesario el "return" ni los tipos en parámetros
- `x -> "hola"` ← Con un parámetro los paréntesis son necesarios
- `() -> 42` ← Si no recibo parámetros
- `(String s) -> { System.out.println(s); }` ← Void lambda

Lambdas

Instancias de Interfaces Funcionales.

```
final BinaryOperator<Integer> operacion = (x, y) -> {  
    System.out.println("recibi" + x);  
    return y + 1;  
};  
operacion.apply(1, 2);
```

Ejecutando la función para los valores 1 y 2

```
final Function<Integer, String> func = (x) -> "hola";  
func.apply(1);  
  
operacion.andThen(func).apply(2, 3);
```

Componiendo las funciones

Igual este no es el uso más común de lambdas




Lambdas - Usos (más) comunes

```
//variable declarations and assignment
Callable<String> c = () -> "done";

//method parameter
button.addActionListener(e -> ui.dazzle(e.getModifiers()));

//method return
...
return () -> { System.out.println("later") };
```



Lambdas - Usos (menos) comunes

//array initialization

```
filterFiles(new FileFilter[] {  
    f -> f.exists(), f -> f.canRead(),  
    f -> f.getName().startsWith("q")  
});
```

//body of other lambda

```
Supplier<Runnable> c = () -> () -> { System.out.println("hi");};
```

//choice in conditional expresion

```
Callable<Integer> c = flag ? (() -> 23) : (() -> 42);
```




Type Inference


Type inference se agrega en Java 7, y es el proceso por el cual Java puede deducir los tipos no especificados de una variable basados en el contexto.

```
// Java < 7
```

```
Map<String, Map<String, String>> mapOfMaps = new HashMap <String, Map  
<String, String>> ();  
List <String> strList = Collections.<String> emptyList();  
List <Integer> intList = Collections.<Integer> emptyList();
```

```
// Java 8
```

```
Map<String, Map<String, String>> mapOfMaps = new HashMap <> ();  
List<String> strListInferred = Collections.emptyList();  
List<Integer> intListInferred = Collections.emptyList();
```



Lambdas - Target Typing & Type Inference

Type inference se mejora en [Java 8](#) para que pueda inferir tipos a partir del contexto y a quien se asigna al valor (ya sea variable o parámetro de función).

```
//variable declarations and assignment (same but not the same)
```

```
Callable<String> c = () -> "done";
```

```
PrivilegedAction<String> a = () -> "done";
```

```
//asignation on Object
```

```
Object o = (Runnable) () -> { System.out.println("hi"); };
```

```
Object o = () -> { System.out.println("hi"); }; // Illegal
```

```
//arguments inferred from the parameter type
```

```
List<Integer> intList = Arrays.asList(5, 2, 4, 2, 1);
```

```
Collections.sort(intList, (a, b) -> a.compareTo(b));
```

```
List<String> strList = Arrays.asList("Red", "Blue", "Green");
```

```
Collections.sort(strList, (a, b) -> a.compareTo(b));
```

Lambdas - Lexical scoping

Las lambdas no son inner classes, por lo tanto no generan un nuevo scope para sus variables, sino que utilizan el scope de quién los contiene

```
public class Hello {  
    Runnable r1 = () -> { System.out.println(this); };  
    Runnable r2 = () -> { System.out.println(toString());  
};  
  
public String toString() { return "Hello, world!"; }  
  
public static void main(String... args) {  
    new Hello().r1.run();  
    new Hello().r2.run();  
}  
}
```

Imprime 2 veces "Hello, world!"

Si fueran inner classes cada "this" referiría a esa clase por lo que el toString sería el de Object no el de Hello.

Variable effectively final

Java 8 agrega el concepto de effectively final: variables no marcadas como final pero que como no se modifican son consideradas "final"

```
String hello = "Hi";
```

```
public String sayHi(String name) {  
    return hello + ": " + name;  
}
```

hello es "effectively final"

```
private int count = 0;
```

```
int countBy(int by) {  
    count += by  
    return count;  
}
```

Por que se modifica count no es final ni "effectively final"

Lambda - Variable capture

En las lambdas

Si debo usar una variable definida por quien la contiene, esa variable debe ser final o effectively final.

```
List<List<Integer>> list = new ArrayList<>();  
int sum = 0; // La línea de abajo no compila  
list.forEach(e -> { sum += e.size(); });
```

esto se resuelve
con “reduction”
(más adelante)


la variables definidas en un lambda no pueden hacer “shadowing” de las variables del contenedor.

```
List<User> users = new ArrayList<>();  
User user = null; // La línea de abajo no compila  
users.forEach(user -> user.toString());
```



Lambda - Menos riesgos

Debido al tipo de captura y el scope de las variables que utilizan las lambdas no necesitan tener una **strong reference** a la clase que lo contiene lo que **evita los memory leaks** que se tienen al mal utilizar las inner classes.



Resumiendo lo visto hasta ahora...

```
//arrancamos de...
```

```
List<Person> people = ...
```

```
Collections.sort(people, new Comparator<Person>() {  
    public int compare(Person x, Person y) {  
        return x.getLastName().compareTo(y.getLastName());  
    }  
});
```

```
// using Lambda
```

```
Collections.sort(people,  
    (Person x, Person y) ->  
        x.getLastName().compareTo(y.getLastName()));
```

```
//using static method
```

```
Collections.sort(people,  
    Comparator.comparing((Person p) -> p.getLastName()));
```



Resumiendo lo visto hasta ahora...

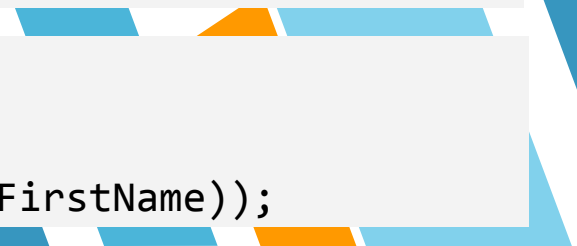
```
//using static import for Comparator::comparing  
Collections.sort(people, comparing((Person p) -> p.getLastName()));
```

```
//using inference por lambda parameter (and as it is one parenthesis are  
gone)  
Collections.sort(people, comparing(p -> p.getLastName()));
```

```
//using method reference  
Collections.sort(people, comparing(Person::getLastName));
```

```
//using a default method to combine comparators  
Collections.sort(people,  
comparing(Person::getLastName).thenComparing(Person::getFirstName));
```

```
//using new list api  
people.sort(comparing(Person::getLastName)  
            .thenComparing(Person::getFirstName));
```





Optional

Optional

Optional es un contenedor que puede o no tener un valor. Al mismo que se le puede preguntar la existencia del valor o generar comportamiento de acuerdo a si lo tiene o no.

```
Map<String, Integer> values= ...;
Optional<String> opt1 = Optional.empty();
Optional <Integer> opt2 = Optional.ofNullable(values.get("key"));
Optional <Integer> opt3 = Optional.of(values.get("key"));
```

Potencial Null Pointer

Condicionales:

```
Optional<Soundcard> soundcard = ...
if(soundcard.isPresent()){
    System.out.println(soundcard.get());
}
```

ó

Obtener el valor

```
soundcard.ifPresent(System.out::println);
```



Optional

//Transformaciones:

```
Optional<Soundcard> op = soundcard.filter(usb ->
```

```
"3.0".equals(usb.getVersion()));
```

```
Optional<USB> usb = soundcard.map(Soundcard::getUSB); //getUSB retorna USB
```

```
Optional<USB> opt = soundcard.flatMap(Soundcard::getUSB); //getUSB retorna  
Optional<USB>
```

//Ors (si está vacío entonces...):

```
soundcard.orElse(otherSoundcard);
```

```
soundcard.orElseGet(() -> createSoundCard())
```

```
soundcard.orElseThrow(IllegalStateException::new);
```



Optional - Ejemplo

```
public class Computer {  
    private Optional<Soundcard> soundcard;  
    public Optional<Soundcard> getSoundcard() {  
        return soundcard;  
    }  
}  
  
public class Soundcard {  
    private Optional<USB> usb;  
    public Optional<USB> getUSB() {  
        return usb;  
    }  
}  
  
public class USB {  
    public String version;  
    public String getVersion(){  
        return version;  
    }  
}
```

Optional

//Lo que se escribía:

```
String version = "UNKNOWN";  
if(computer != null){  
    Soundcard soundcard = computer.getSoundcard();  
    if(soundcard != null){  
        USB usb = soundcard.getUSB();  
        if(usb != null){  
            version = usb.getVersion();  
        }  
    }  
}
```

//ahora se puede escribir de la siguiente manera

```
String name = computer  
    .flatMap(Computer::getSoundcard)  
    .flatMap(Soundcard::getUSB)  
    .map(USB::getVersion)  
    .orElse("UNKNOWN");
```



Collections





Collections

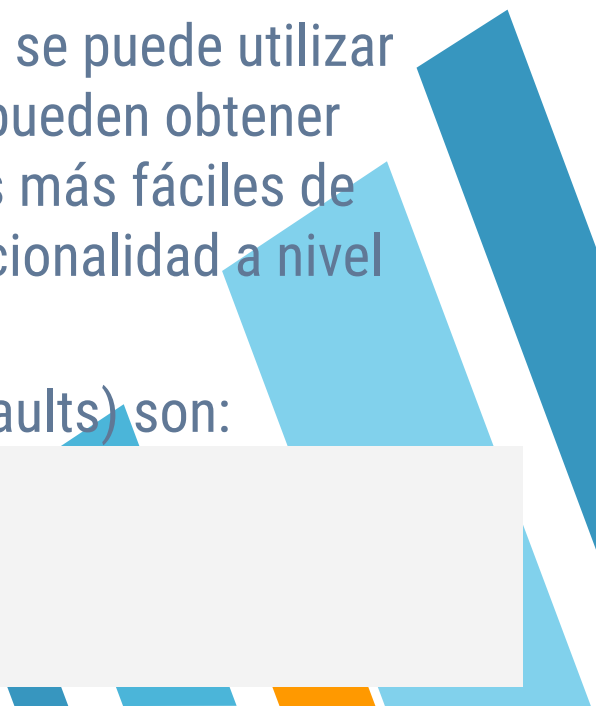
La API de colecciones de Java fue altamente mejorada para la versión 8 del lenguaje.

Gran parte de los cambios presentados hasta ahora se incorporan por ser útiles en sí mismo pero además porque **combinados con las API de colección** se puede procesar la información de **una manera mucho más funcional**.

Entonces a partir de convertir la colección en un stream se puede utilizar lambdas para convertir y operar sobre la colección. Se pueden obtener como resultados Optionals. Y para hacer estos cambios más fáciles de agregar se utilizan métodos default para agregar la funcionalidad a nivel interfaz.

Los métodos más importantes agregados (también defaults) son:

```
default Stream<E> stream()  
default Stream<E> parallelStream()  
default Spliterator<E> spliterator()
```






Stream - Ejemplo

La clase Stream es una secuencia de elementos a la cual se le puede realizar una secuencia (o pipeline) de operaciones.

Permite simplificar las transformaciones de conjuntos de datos :

```
for (Person p : roster) {  
    if (p.getAge() > 17) {  
        System.out.println(p.getName());  
    }  
}
```


```
roster  
    .stream()  
    .filter(e -> p.getAge() > 17)  
    .forEach(e ->  
        System.out.println(e.getName()));
```






Stream - Sintaxis

Un stream no es una estructura que contiene datos sino un pipeline que se arma mediante:

- **Una fuente:** que puede ser una colección o una función “generador” o una conexión I/O (en este caso roster al cual se le genera el stream mediante a `::stream()`)
 - **Una cadena (opcional) de operaciones intermedias:** donde cada una genera un nuevo stream al que se puede aplicar más operaciones (en este caso filter)
 - **Una operación terminal:** que no genera un nuevo stream sino una colección nueva o un valor.
- 



Stream - Factories

- » La interface Collection provee un método `::stream()` para obtener un stream sobre el contenido de la colección.
 - » Hay métodos estáticos en Stream :
 - ◇ `::of(T), ::of(T...)`: crea a partir de elementos
 - ◇ `::empty()` : stream vacío
 - ◇ `::concat(Stream, Stream)`: concatena los streams
 - ◇ `::generate(Supplier)`: obtiene los valores del supplier
 - ◇ `::iterate(seed, UnaryOperator)`: genera a partir del seed aplicando el operator.
 - » También se puede usar **Stream.Builder** para ir construyendo el stream por elementos.
- 



Stream - Specific Types

Por otro lado hay stream de tipos específicos como:

- **IntStream**
- **DoubleStream**
- **LongStream**

vienen con operaciones relacionadas a esos tipos:

- summaryStatistics
- Sum
- min
- max
- average

Estos Streams se pueden transformar entre ellos usando los métodos mapToX (donde X es algún otro tipo como Int, Double o Long).

Si se quiere obtener un Stream “normal” entonces se debe usar mapToObj().





Stream - Operaciones Intermedias

Las operaciones intermedias generan un nuevo stream que toma de fuente el stream anterior y lo modifican (mediante filtros y/o operaciones).


Estas operaciones no se calculan hasta que no son requeridas por una **operación terminal** (son **lazy**).





Stream - Operaciones Intermedias

Algunos ejemplos de operaciones intermedias:

- **map:** Aplica una función $T \rightarrow W$ a cada elemento T del stream
 - **flatMap:** Aplica una función $T \rightarrow \text{Stream}(W)$ a cada elemento y aplana los resultados.
 - **limit:** limita la cantidad de elementos del stream
 - **distinct:** genera un stream sin elementos repetidos (según `::equals`).
 - **filter:** aplica un predicado a cada elemento dejando solo los que cumplen.
 - **sorted:** ordena los elementos de acuerdo a un Comparator
- 




Stream - Operaciones Terminales

Son operaciones que **retornan una colección nueva o un valor** (que puede ser un Optional) a partir de la cadena de operaciones del pipeline.

Una vez ejecutada esta operación el stream se cierra (implementa AutoCloseable) por lo que no se puede reutilizar.

- **min/max(Comparator)**
- **count**
- **anyMatch/allMatch/noneMatch(Predicate)**
- **findFirst/findAny()**

En los específicos se agregan otros como: **IntStream::average**, **Int::summaryStatistics**, **DoubleStream::Average**, **DoubleStream::summaryStatistics**, etc





Stream - Aggregators - Reduce

Las operaciones terminales se pueden considerar


```
reduce(BinaryOperator<T> accumulator)

reduce(T identity, BinaryOperator<T> accumulator)

reduce(U identity,
        BiFunction<U,? super T,U> accumulator,
        BinaryOperator<U> combiner)
```

reduce aplica la operación accumulator entre lo ya acumulado y el siguiente valor del stream.

La segunda versión usa identity como el valor inicial (por eso siempre devuelve valor). La tercera versión solo sirve en Parallel Streams y aplica el combiner entre los resultados parciales



Stream - Aggregators - Reduce

Ejemplos:

Concatenar textos

```
Stream<String> words = Stream.of("this", "is", "a", "phrase");  
String phrase = words.reduce("", (x, y) -> x + y);
```

Sumar las edades de las personas

```
Integer totalAgeReduce = roster  
    .stream()  
    .map(Person::getAge)  
    .reduce(0, (a, b) -> a + b);
```

Sumar los costos de las facturas (usando BigDecimal)

```
BigDecimal sum = invoices.stream()  
    .map(x -> x.getQty().multiply(x.getPrice()))  
    .reduce(BigDecimal.ZERO, BigDecimal::add);
```


Stream - Aggregators - Collect

Stream::collect es otro tipo de operación terminal que permite modificar/mutar los valores de un stream para obtener un valor o una colección de valores.

```
collect(Collector<? super T,A,R> collector)
collect(Supplier<R> supplier, BiConsumer<R,? super T> accumulator,
BiConsumer<R,R> combiner)
```

La primera versión utiliza un Collector que engloba los elementos de la segunda versión (que son “similares” a los de `::reduce`).

```
Stream<Integer> ages = Stream.of(5, 12, 33, 23, 51, 78, 15);

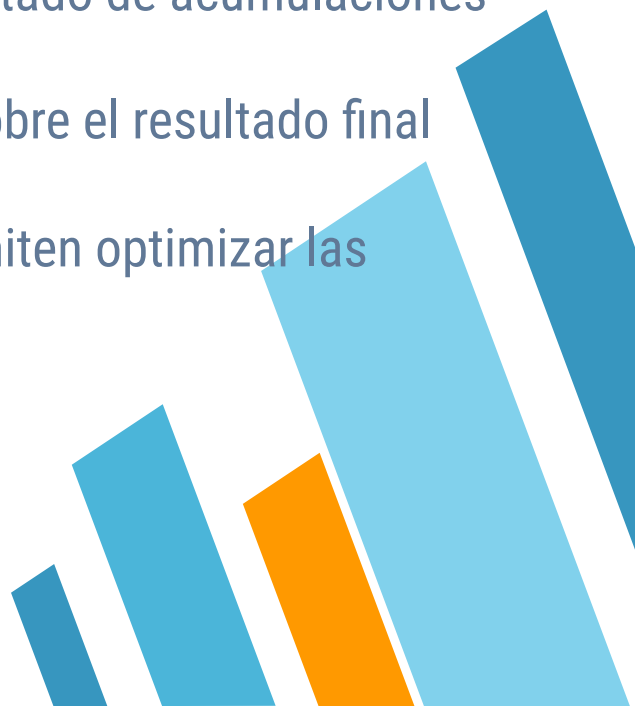
ArrayList<Integer> agesList = ages.collect(ArrayList::new,
                                           ArrayList::add, ArrayList::addAll);

List<Integer> agesList2 = ages.collect(Collectors.toList());
```



Stream - Aggregators - Collectors

La clase Collector recibe:


- » **Supplier<A> supplier:** para obtener la clase que acumulará
 - » **BiConsumer<A, T> accumulator:** Una función de la clase acumulador para “acumular”
 - » **BinaryOperator<A> combiner:** Cómo combinar resultado de acumulaciones paralelas
 - » **Function<A, R> finisher:** Una función para aplicar sobre el resultado final (típicamente Identidad).
 - » **Characteristics... characteristics:** Valores que permiten optimizar las transformaciones.
- 



Stream - Aggregators - Collectors

El SDK ya provee algunos de estos Collectors que se pueden obtener de la clase Collectors:

```
Collectors.toList()
Collectors.toCollection(TreeSet::new)
Collectors.joining(", ")
Collectors.summingInt(Employee::getSalary)
Collectors.groupingBy(Employee::getDepartment)
Collectors.groupingBy(Employee::getDepartment,
                      Collectors.summingInt(Employee::getSalary))
Collectors.partitioningBy(s -> s.getGrade() >= PASS_THRESHOLD)
```





Java Time

Java Time - Yet another new API

La idea es una nueva API con los siguientes principios:

- » **Clara**: con métodos bien definidos y con comportamiento esperable (parámetro null -> NullPointerException).
- » **Fluent**: para que sea fácil de leer

```
LocalDate today = LocalDate.now();  
LocalDate payday =  
    today.with(TemporalAdjusters.lastDayOfMonth()).minusDays(2);
```

- » **Inmutable**: las operaciones sobre fechas no modifican el objeto sino que devuelven otros.

```
LocalDate dateOfBirth = LocalDate.of(2012, Month.MAY, 14);  
LocalDate firstBirthday = dateOfBirth.plusYears(1);
```

Java Time - Factory Methods

- » of: crea a partir de parámetros (fecha y hora como ints, etc)
- » from: crea a partir de convertir los parámetros
- » parse: genera una fecha a partir de una cadena.

```
//Current Date
```

```
LocalDate today = LocalDate.now();
```

```
//Creating LocalDate with values
```

```
LocalDate firstDay_2014 = LocalDate.of(2014, Month.JANUARY, 1);
```

```
LocalDate.ofEpochDay(365);
```

```
LocalDate.ofYearDay(2014, 100);
```

```
LocalDate localDate = LocalDate.from(Instant.now());
```

```
//parse
```

```
LocalDateTime dt = LocalDateTime.parse("27::Apr::2014 21::39::48",  
    DateTimeFormatter.ofPattern("d::MMM::uuuu HH::mm::ss"));
```

Java Time - Objects

- » Tipos de momento “fechas”: la X es la Localización
 - ◇ XDate: representa una fecha (día, mes, año).
 - ◇ XTime: representa una hora (hora, minutos, segundos, etc.)
 - ◇ XDateTime: la suma de date y time
- » Localización:
 - ◇ LocalX: una “fecha” sin timezone
 - ◇ ZonedX: una “fecha” con timezone
 - ◇ ZoneId/ZoneOffset: Para setear zonas


```
LocalDateTime todayInBA =  
    LocalDateTime.now(ZoneId.of("America/Argentina/Buenos_Aires"));  
  
ZonedDateTime zonedInIndianapolis =  
    todayInBA.atZone(ZoneId.of("America/Indiana/Indianapolis"));  
  
ZonedDateTime gregorianCalendarDateTime = new  
    GregorianCalendar().toZonedDateTime();
```



Java Time - Objects

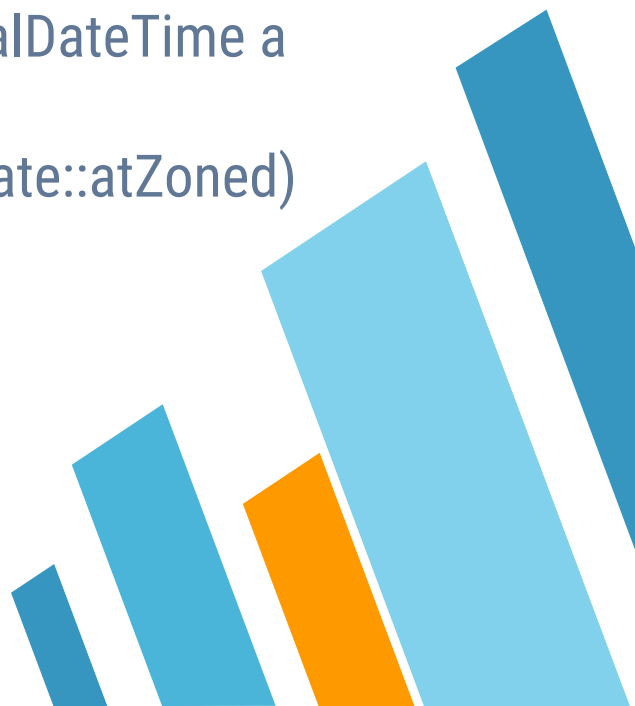
- » Time-Ranges: (para operar o como resultado de operaciones con fechas)
- » Period: Rango con precisión a “día”
 - ◇ Duration: Rango con precisión a “nano segundos”

```
LocalDate today = LocalDate.now();  
LocalDate birthday = LocalDate.of(1960, Month.JANUARY, 1);  
  
Period p = Period.between(birthday, today);  
long p2 = ChronoUnit.DAYS.between(birthday, today);
```





Java Time - Methods

- » **format**: transforma en cadena string la instancia.
 - » **getX**: devuelve una “parte” de la fecha.
 - » **isX**: pregunta acerca del estado de la fechas.
 - » **withX**: retorna una nueva fecha con el campo X modificado.
 - » **plus**: suma una fecha o un rango
 - » **minus**: resta una fecha
 - » **to**: transforma la instancia a otro tipo (de LocalDateTime a LocalDate)
 - » **at**: combina una instancia contra otra (LocalDate::atZoned)
- 

Java Time - Ejemplos

//Construction and transformations

```
LocalDateTime timeInThePast = LocalDateTime.now().withDayOfMonth( 5 )  
.withYear( 2005 );
```

```
LocalDateTime moreInThePast = timeInThePast.minusWeeks(2).plus( 3,  
ChronoUnit.DAYS );
```

```
LocalDate now = LocalDate.now();  
LocalDate adjusted = now.with(TemporalAdjusters.lastDayOfMonth());
```

//Zoned

```
ZoneId zoneIdParis = ZoneId.of( "Europe/Paris" );  
ZoneId zoneIdAGT = ZoneId.of( ZoneId.SHORT_IDS.get( "AGT" ) );  
LocalDateTime dateTime = LocalDateTime.now( zoneIdAGT );  
ZonedDateTime zonedDateTimeAGT = ZonedDateTime.of(dateTime, zoneIdAGT  
);
```



Java Time - Ejemplos

//Periods

```
Period period = Period.of( 3, 2, 1 );  
Period period4Months = Period.ofMonths( 4 );  
Period periodB = Period.between( LocalDate.now(), LocalDate.of( 2015,  
Month.JANUARY, 1 ) );  
  
period4Weeks.get( ChronoUnit.DAYS );  
  
LocalDate newDate = LocalDate.now().plus( period4Months );
```

//Format

```
LocalDateTime dateTime = LocalDateTime.of( 2014, Month.DECEMBER, 15,  
15, 0, 30 );  
String isoDateTime = dateTime.format( DateTimeFormatter.ISO_DATE_TIME  
);
```





Java 1Xs

Java - Text Block

Text Block es una alternativa que tiene Java a la hora de definir Strings multilineales sin tanta burocracia.

```
String name = ""  
    red  
    green  
    blue  
    "";
```

// salto de línea obligatorio

// salto de línea opcional

Los bloques respetan la indentación y los espacios.

```
String html = ""  
..... <html>  
..... <body>  
..... <p>Hello World.</p>  
..... </body>  
..... </html>  
..... "";
```

Java - Switch Expression

Una Switch expression es un switch que retorna un valor

```
public enum Day { SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY,
SATURDAY; }

Day day = Day.WEDNESDAY;
System.out.println(
    switch (day) {
        case MONDAY, FRIDAY, SUNDAY -> 6;
        case TUESDAY -> 7;
        case THURSDAY, SATURDAY -> 8;
        case WEDNESDAY -> 9;
        default -> throw new IllegalStateException("Invalid day: " + day);
    }

);
```

Java - Switch Expression Yield

Yield permite indicar el valor de retorno para un bloque del case.

```
Day day = Day.WEDNESDAY;

int numLetters = switch (day) {

    case MONDAY, FRIDAY, SUNDAY -> {

        System.out.println(6);

        yield 6;

    }

    case TUESDAY , WEDNESDAY , THURSDAY, SATURDAY -> {

        System.out.println(8);

        yield 8;

    }

    default -> {

        throw new IllegalStateException("Invalid day: " + day);

    }

};
```

Java - Pattern Matching for instanceof

Se puede definir una variable del tipo correspondiente al hacer un instanceof

```
public static double getPerimeter(Shape shape) throws
IllegalArgumentException {
    if (shape instanceof Rectangle s) {
        return 2 * s.length() + 2 * s.width();
    } else if (shape instanceof Circle s) {
        return 2 * s.radius() * Math.PI;
    } else {
        throw new IllegalArgumentException("Unrecognized
shape");
    }
}
```





Java - Records

Los Records son clases inmutables de Java.

```
public record Person (String name, String address) {}
```

Con esta definición tenemos una clase:

- Con constructor que setea esas variables.
 - Con getters
 - Equals, hashCode y toString.
- 



Referencias

- » [Java 1Xs cambios](#)
 - » [Java 8 - What's new](#)
 - » [Java 8 - Enhancements on the language](#)
 - » [Java - Lambda Tutorials](#)
 - » [Java 8 - Enhancements - Due to lambdas](#)
 - » Blog de Brian Goetz en [state of lambdas](#) & [lambdas in collections](#)
 - » [Tutorial on Method Reference](#)
 - » [Tutorial Default Method](#) & [Inheritance](#)
 - » [Article on Optional](#)
 - » [Tutorial Collection Streams Pipeline and Aggregation](#)
 - » [Tutorial Parallel Streams](#) (bueno para revisar links a concurrencia).
 - » [Article new Date](#) & [Tutorial Date API](#)
 - » [Article Concurrency](#)
 - » <http://www.nurkiewicz.com/2013/05/java-8-definitive-guide-to.html>
 - » <http://www.deadcoderising.com/java8-writing-asynchronous-code-with-completablefuture/>
 - » <http://codingjunkie.net/completable-futures-part1/>
- 