



Novedades de Java 8

Índice



1 El paquete java.time	3
2 Interfaces funcionales	7
3 Expresiones Lambda (clausuras)	10
4 Referencias de método	13
5 Referencias de constructor	18
6 Interfaces funcionales predefinidas	20
7 Stream en Java 8	21

1. El paquete java.time

En Java 8 se introduce un nuevo API para fechas contenido en el paquete java.time, prácticamente todo lo que expone este API es inmutable; es decir objetos que una vez creados ya no pueden cambiar su estado, no se puede variar el valor de sus propiedades.

Las clases más importantes que incorpora java.time son:

Clase	Funcionalidad
Instant	Representa en milisegundos el instante actual desde 1 de enero de 1970.
LocalDateTime	Se utiliza para almacenar año, mes, día, hora, minutos y segundos.
LocalDate	Se utiliza para almacenar una fecha sin hora, sólo día, mes y año.
LocalTime	Se utiliza para tiempos con horas, minutos y segundos.
ZonedDateTime	Almacena fecha y hora con información del uso horario.

Tabla 13.1: Clase en java.time.

La relación con las fechas de versiones anteriores **Date**, **Time**, **Calendar** y **GregorianCalendar** se basa en las funciones siguientes:

Clase	Función
Date	public LocalDate toLocalDate() public Instant toInstant() public static Date valueOf(LocalDate date) public LocalTime toLocalTime()
Time	public Instant toInstant() public static Time valueOf(LocalTime time)
Calendar	public final Instant toInstant() public ZonedDateTime toZonedDateTime()
GregorianCalendar	public static GregorianCalendar from(ZonedDateTime zdt)

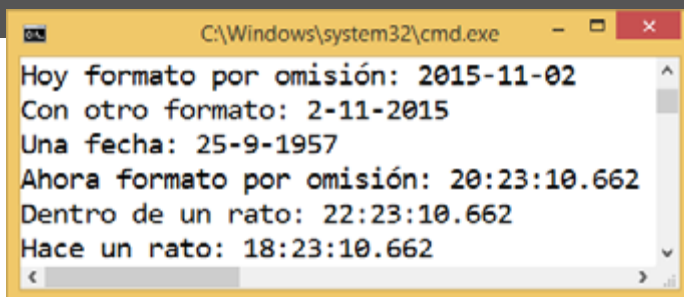
Tabla 13.2: Relación de fechas java.time con versiones anteriores.

Algunos ejemplos de código:

```
import java.time.LocalDate;
import java.time.LocalDateTime;

public class EjemploFechas {
    public static void main(String[] args) {
        LocalDate hoy = LocalDate.now();
        System.out.println("Hoy formato por omisión: "+hoy);
        System.out.println("Con otro formato: "+hoy.
            getDayOfMonth()+"-"+hoy.getMonthValue()+"-"+hoy.getYear());
        LocalDate fecha = LocalDate.of(1957, 9, 25);
        System.out.println("Una fecha: "+fecha.
            getDayOfMonth()+"-"+fecha.getMonthValue()+"-"+fecha.getYear());

        LocalDateTime ahora = LocalDateTime.now();
        System.out.println("Ahora formato por omisión: "+ahora);
        System.out.println("Dentro de un rato: "+ahora.
            plusHours(2));
        System.out.println("Hace un rato: "+ahora.
            minusHours(2));
    }
}
```



Para poder dar un formato determinado a las fechas, se ha incluido la clase:

java.time.format.DateTimeFormatter

Con esta clase se puede dar un formato a una fecha de acuerdo con un patrón formado por caracteres que indican cómo escribir el año, el mes, el día, las horas, los minutos, los segundos y demás características que implican a las fechas y a los tiempos.

Un código ejemplo de su utilización es el siguiente:

```
LocalDate date = LocalDate.now();
DateTimeFormatter formatter =
    DateTimeFormatter.ofPattern("yyyy MM dd");
String text = date.format(formatter);
LocalDate parsedDate = LocalDate.parse(text,
    formatter);
```

IMAGEN 13.1: EJEMPLOS DE CÓDIGO CON FECHAS.

En el código anterior a partir de la fecha actual, se relazan las tres operaciones siguientes:

- La función `static ofPattern`, recibe como parámetro un patrón y devuelve la referencia a un objeto `DateTimeFormatter` de acuerdo con ese patrón.
- Con la función **format** de **LocalDate**, se puede convertir la fecha con el formato creado en un **String**.
- También se puede obtener la referencia a un **LocalDate** a partir de un **String** y un formato.

Algunos de los caracteres que se pueden utilizar para formar los patrones de formato en fechas son:

Letra	Función	Letra	Función
y	yy año del siglo, yyyy año 4 cifras	w	número de la semana en el año
D	número del día en el año	W	número de la semana en el mes
M	M o M número del mes MMM o MMMM mes en letras	H	hora del día con una cifra (0 a 9) o dos, en formato 0 a 23. Si se utiliza "a" en formato AM o PM
d	número del día del mes, d con una cifra (0 a 9), dd dos cifras siempre	m	minutos, m con una cifra (0 a 9), mm con dos cifras siempre
E	Nombre del día EEE o EEEE	s	minutos, s con una cifra (0 a 9), ss con dos cifras siempre

Tabla 13.3: Caracteres de patrones en fechas.



Los patrones con letras mayúsculas están localizados al idioma correspondiente, por ejemplo el patrón "EEEE" pondrá el nombre del día en el idioma local.

En la imagen siguiente se muestra el código de una fecha formateada en versión Java 8 y en versión anterior.

```
public static void main(String[] args) {  
    // Escribir la fecha de hoy con Java 8  
    LocalDate hoy = LocalDate.now();  
    System.out.println("JAVA 8 ==> Hoy es: " + hoy);  
    System.out.println("JAVA 8 ==> Hoy es: "  
        + hoy.format(DateTimeFormatter  
            .ofPattern("EEEE d 'de' MMMM 'de' yyyy"))));  
    // Escribir la fecha de hoy con Java 7  
    Date fecha = new Date();  
    System.out.println("JAVA 7 ==> Hoy es: " + fecha);  
    System.out.println("JAVA 7 ==> Hoy es: "  
        + DateFormat.getDateInstance(DateFormat.LONG, new Locale("ES"))  
            .format(fecha));  
}
```

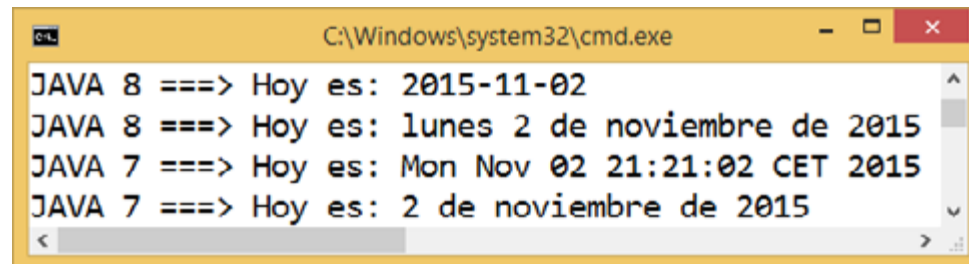


IMAGEN 13.2: FECHAS CON FORMATO.

2. Interfaces funcionales

Una interface funcional es aquella que contiene exactamente un método abstracto (sin definir) pudiendo contener si es necesario varios métodos **default**. Las interfaces Runnable, ActionListener, Comparator o Callable, tienen una característica en común, únicamente tienen un método abstracto. A este tipo de interfaces se les conoce como **Single Abstract Method Interfaces** o **SAM** Interfaces. La forma más común de utilizarlos es implementándolos mediante una clase anónima.

En los códigos siguientes se muestra la implementación Comparator para comparar String y para comparar objetos de una clase Punto, por el criterio de la distancia al punto (0,0)

```
import java.util.Comparator;
public class ComparadorCadenas implements
Comparator<String> {
    @Override
    public int compare(String arg0, String arg1) {
        return arg0.compareTo(arg1);
    }
}
```

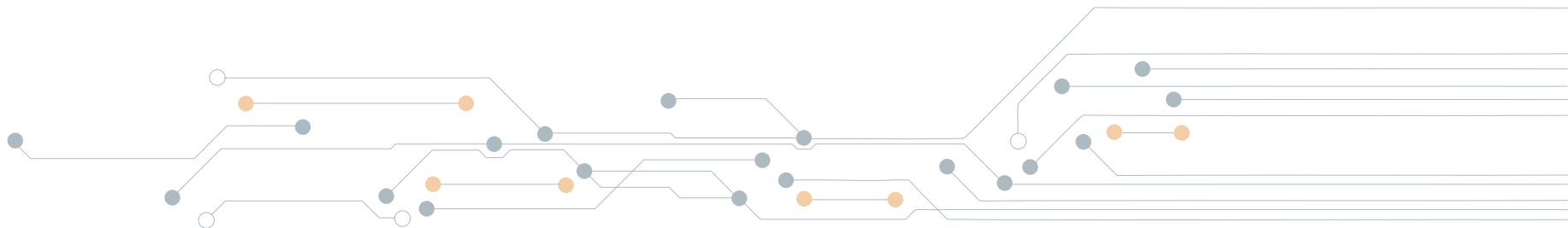
```
import java.util.Comparator;
public class ComparadorPuntos implements
Comparator<Punto>{
    @Override
    public int compare(Punto o1, Punto o2) {
        if(o1.distancia()>o2.distancia())
            return 1;
        if(o1.distancia()<o2.distancia())
            return -1;
        return 0;
    }
}
```

Es frecuente en Java, implementar la única función de estas **SAM Interfaces** en una **clase anónima**, como se muestra en el ejemplo siguiente para ordenar los elementos de una colección de objetos **Punto** mediante el método **static sort** de la clase **Collections**.

```
//Collections.sort(lista, new ComparadorPuntos());
Collections.sort(lista, new Comparator<Punto>(){
    public int compare(Punto o1, Punto o2) {
        if(o1.distancia()>o2.distancia())
            return 1;
        if(o1.distancia()<o2.distancia())
            return -1;
        return 0;
    }
});
```

Los inconvenientes de **las clases anónimas** son su **sintaxis compleja**, son clases sin nombre añadidas en el punto en el que se necesita un objeto de dicha clase y por tanto sólo tienen ese uso.

Otro aspecto que hay que tener en cuenta es que en métodos como sort de **Collections** el segundo parámetro, el parámetros del objeto de la clase que implementa la única función de **Comparator**, es un **parámetro puramente** funcional, en realidad de lo que se trata es que ejecute una funcionalidad cada vez que tenga que comparar dos objetos de la clase Punto.



Java 8 va a resolver la complejidad de las clases anónimas y de los métodos que reciben un parámetro netamente funcional mediante las interfaces funcionales y las expresiones lambda.

Para indicar que una interface es funcional se utiliza la anotación **@FunctionalInterface**, se utiliza para declarar la intención que la interface sea funcional. Esta notación la utiliza el compilador para indicar que se debe de incluir una única función abstracta. Denota intención como **@Override**.

Una consideración que hay que tener en cuenta con las interfaces funcionales, es la siguiente:

En la definición de interfaz funcional se indica que en un interfaz pueden existir múltiples métodos abstractos siempre que todos menos uno sobrescriban un método público de la clase Object.

En el ejemplo se muestra una interfaz funcional con su único método abstracto y un método default.

```
@FunctionalInterface
public interface InterfazUno {
    //método que caracteriza esta interfaz funcional
    public void Pintar();
    default public void HacerAlgo(){
        System.out.println("InterfazUno ==>
HacerAlgo");
    }
    //método abstracto que sobrepasa al de Object
    public String toString();
}
```

```
<<interface>>
InterfazFuncional

+ pintar(): void <<abstract>>
+ hacerAlgo():void <<default>>
```

Una interfaz funcional tiene una única función abstracta y tanta funciones default como se considere.

```
<<interface>>
Runnable

+run(): void <<abstract>>
```

```
<<interface>>
Comparator

+compare(Object, Object): int <<abstract>>
```

```
<<interface>>
Callable<V>

+call(): V <<abstract>>
```

```
<<interface>>
ActionListener

+ actionPerformed(ActionEvent ): void <<abstract>>
```

IMAGEN 13.3: INTERFACES FUNCIONALES.

3. Expresiones Lambda (clausuras)

Una **expresión lambda** es un método anónimo que **implementa** el método abstracto definido en una **interfaz funcional**. En Java 8 se añaden como una aproximación a los lenguajes de programación funcional. En realidad en Java la programación funcional se simula con las clases anónimas, pero debido a su compleja sintaxis se pretende sea sustituido su uso por las expresiones lambda.

Se basan en la utilización de un nuevo operador, llamado lambda "->". su sintaxis es la siguiente:

```
(<lista_argumentos> -> {<sentencias>})  
<lista_argumentos> puede estar vacía, son los  
parámetros de llamada a la función.  
{<sentencias>} si es un única sentencia no hace falta  
utilizar las "{}".
```

Algunos ejemplos de expresiones lambda:

```
(String s) -> System.out.println(s);  
(int i, int j) -> i * j;  
(Persona p) -> {p.getEdad();}  
(String nombre, Saludo saludo) -> {  
    String di= saludo.getSaludo() + nombre;  
    System.out.println(di);  
}  
x->System.out.println(x);
```

Para utilizar una expresión lambda esta se tiene que colocar en el punto en el que se pueda utilizar una referencia a la interfaz funcional que implementa. Por ejemplo:v

```

@FunctionalInterface
interface UnDato{
    String getData();
}

//Formas de crear la expresión lambda
//Se crea una referencia a UnDato
UnDato undato;
//Se asigna una expresión lambda a la referencia al
interface
undato = () -> "Es Java 8";
//Se inicializa la referencia con la expresión lambda
Undato otrodato = () -> "Las expresiones lambda";

//Invocar a la función que ha implementado la
expresión lambda
System.out.println("Los datos son: "+undato.getData()
+
                " y " + otrodato.getData());

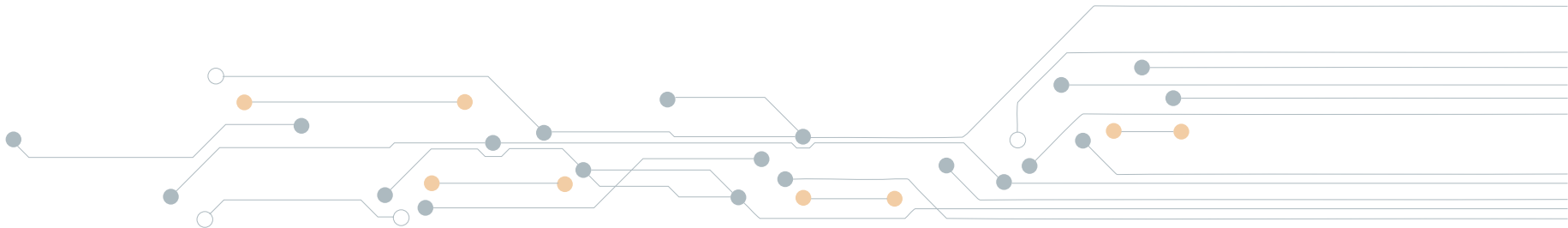
```

En los ejemplos siguientes se muestran como utilizar las interfaces Runnable y Comparator con expresiones lambda.

```

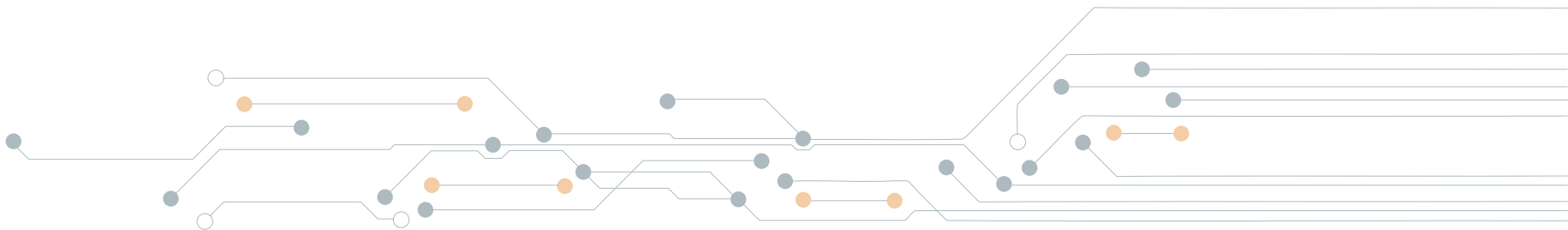
public class CreateThreadRunnableExample {
    public static void main(String[] args) {
        //Antes de Java 8:
        new Thread(new Runnable() {
            public void run() {
                System.out.println("Demasiado
código para tan poco");
            }
        }).start();
        //En Java 8:
        new Thread(() ->
            System.out.println("Con expresiones
Lambda")).start();
    }
}

```



```
import java.util.Comparator;
public class ComparaString {
    public static void main(String[] args) {
        Comparator<String> comp =
            (String s1, String s2) ->
s1.compareTo(s2);
        String[] strings = { "Antonio", "Zacarias",
"Carlos" };
        System.out.println(strings[0] + " comparado con "
+ strings[1] + " ==> resultado es:"
+ comp.compare(strings[0],
strings[1]));
        System.out.println(strings[1] + " comparado con "
+ strings[0] + " ==> resultado
es:"
+ comp.compare(strings[1],
strings[0]));
        System.out.println(strings[0] + " comparado
con "
```

```
+ strings[0] + " ==> resultado es:"
+ comp.compare(strings[0],
strings[0]));
        System.out.println(strings[0] + " comparado
con "
+ strings[2] + " ==> resultado
es:"
+ comp.compare(strings[0],
strings[2]));
        System.out.println(strings[2] + " comparado
con "
+ strings[0] + " ==> resultado
es:"
+ comp.compare(strings[2],
strings[0]));
    }
}
```



4. Referencias de método

Una referencia de método permite hacer referencia a un método sin que este se ejecute. Como con las expresiones lambda, cuando se evalúa una referencia de método, también se crea una instancia de una interfaz funcional. Su utilización fundamental es para poder pasar métodos como parámetros a funciones.

Su formato es:

```
<identificador_clase> :: <identificador_metodo> //  
referencia a método static  
<referencia_clase> :: <identificador_metodo> //referencia  
a método de instancia
```

Algunos ejemplos:

```
String::ValueOf      //Método static ValueOf de String  
// equivale a la expresión lambda x -> String.valueOf(x)  
x::toString          //Método de instancia toString del  
objeto x  
// equivale a la expresión lambda () -> x.toString()
```



En el ejemplo siguiente se muestra el uso de referencia a método static:

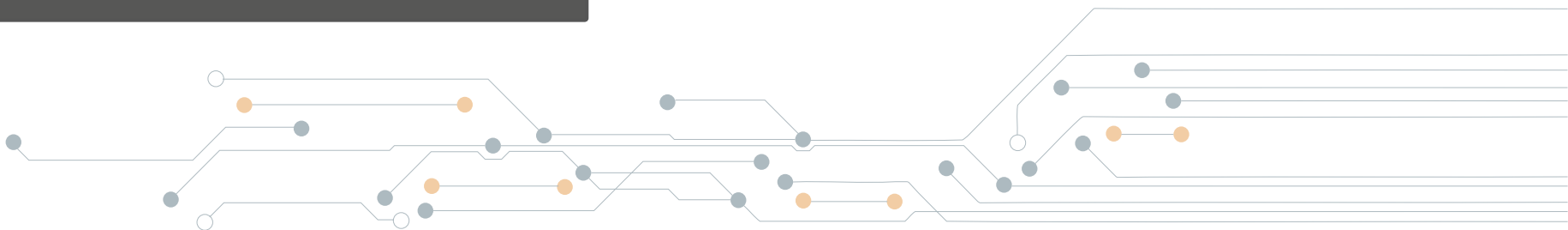
```
public class PruebaReferenciaMetodo {
    // Esta función recibe como primer parámetro una
    // referencia
    // a la interfaz funcional TestearNumero.
    // Por tanto el puede ser cualquier instancia a
    // la interface,
    // incluida una creada por una referencia a
    // método
    static boolean testNumero(TestearNumero p, int n)
    {
        return p.test(n);
    }
    public static void main(String args[]) {
        Random aleatorio = new Random();
        boolean resultado;
        int numero=0;
        for(int i=1; i<=10; i++){
            numero= aleatorio.nextInt(100)-50;
            System.out.print("\n"+numero);
            TestearNumero tn = n
            ->MiTesteadorNumero.isPrimo(n);
            resultado = tn.test(numero);
            //resultado = testNumero
            (MiTesteadorNumero::isPrimo, numero);
            if (resultado) System.out.print(" ES PRIMO
");
            resultado =
            testNumero(MiTesteadorNumero::isPar, numero);
```

```
        if (resultado) System.out.print(" ES PAR
");
        resultado = testNumero
        (MiTesteadorNumero::isImpar, numero);
        if (resultado) System.out.print(" ES IMPAR
");
        resultado = testNumero
        (MiTesteadorNumero::isPositivo,
        numero);
        if (resultado) System.out.print(" ES
POSITIVO ");
        resultado = testNumero
        (MiTesteadorNumero::isNegativo, numero);
        if (resultado) System.out.print(" ES
NEGATIVO ");
        }
    }
```

```
@FunctionalInterface
interface TestearNumero {
    boolean test(int numero);
}

// Clase que define tres métodos estáticos que se
// pasan
// como referencias de método para el "predicado" que
// se
// aplica a el numero que se recibe como segundo
// parámetro
class MiTesteadorNumero {
    static boolean isPrimo(int n) {
        n = Math.abs(n);
        if (n < 2) return false;
        for (int i = 2; i <= n / i; i++)
            if ((n % i) == 0) return false;
        return true;
    }
    static boolean isPar(int n) {
        return (n % 2) == 0;
    }
    static boolean isImpar(int n) {
        return (n % 2) != 0;
    }
}
```

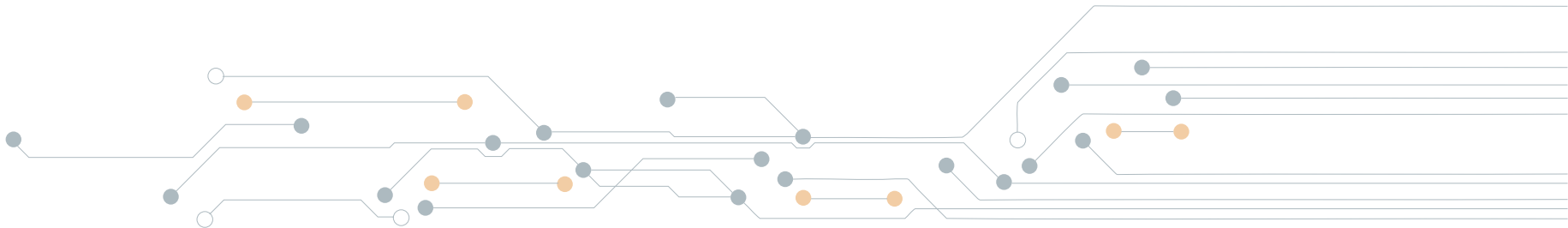
```
static boolean isPositivo(int n) {
    return n > 0;
}
static boolean isNegativo(int n) {
    return n < 0;
}
}
```



En el ejemplo siguiente se muestra el uso de referencia a método de instancia:

```
public class PruebaReferenciaMetodo {
    public static void main(String args[]) {
        Random aleatorio = new Random();
        boolean resultado;
        TestearNumero interfacepredicado;
        int factor=0;
        for(int i=1; i<=10; i++){
            MiTesteadorNumero numero =
                new MiTesteadorNumero(aleatorio.
nextInt(50)+1);
            //interfacepredicado = y -> numero.
isFactor(y);
            interfacepredicado= numero::isFactor;
            resultado =
                interfacepredicado.
test(factor=aleatorio.nextInt(4)+2);
            if (resultado)
                System.out.println( numero.
getNumero()+
```

```
        " ES MULTIPLIO DE " + factor);
    else
        System.out.println( numero.
getNumero()+
        " NO ES MULTIPLIO DE " + factor);
    }
}
```




```

@FunctionalInterface
interface TestearNumero {
    boolean test(int n);
}
class MiTesteadorNumero {
    private int n;
    MiTesteadorNumero(int n) {
        this.n = n;
    }
    int getNumero()      return n;

    boolean isFactor(int x) return (n % x) == 0;
}

```

Se puede usar una referencia a método de instancia con la forma:

```
<identificador_clase> :: <identificador_metodo_instancia>
```

Con esta forma, el primer parámetro de la función de la interfaz funcional coincide con el objeto de invocación y el segundo (si lo hubiera) con el parámetro especificado por el método. Así en el ejemplo anterior la interfaz funcional, la referencia a métodos de instancia y la invocación al método de la interfaz quedaría como sigue:

```

@FunctionalInterface
interface TestearPredicadoNumero {
    boolean test(MiTesteadorNumeroPredicado
predicado, int numero);
}
class MiTesteadorNumeroPredicado {
    private int n;
    MiTesteadorNumeroPredicado(int n) {
        this.n = n;
    }
    int getNumero() {
        return n;
    }
    boolean isFactor(int x) {
        return (n % x) == 0;
    }
}
//En función main
MiTesteadorNumeroPredicado numero;
TestearPredicadoNumero interfacepredicado;
....
numero = new MiTesteadorNumeroPredicado(aleatorio.
nextInt(50)+1);

```

```
//interfacepredicado=(predicado, n)->numero.  
isFactor(n);  
interfacepredicado=  
MiTesteadorNumeroPredicado::isFactor;  
resultado = interfacepredicado.test(numero,  
  
factor=aleatorio.nextInt(4)+2);
```

5. Referencias de constructor

Las referencias a constructor tienen el formato siguiente:

<identificador_clase> :: new

Esta forma de referencia se puede aplicar a cualquier referencia de interfaz funcional que defina un método compatible con el constructor.

En el código siguiente se muestra su aplicación:

```
public class PruebaReferenciaConstructor {  
    public static void main(String[] args) {  
        // Se hace una referencia de constructor de  
        MiClase  
        // Como función en interface tiene un parámetro  
        // el constructor que se ejecuta es el que tiene  
        un parámetro  
        MiInterface constructor = MiClase::new;  
    }  
}
```

```
//MiInterface constructor = (String s) -> new
MiClase(s);
// Se crea una instancia con el constructor con
un parámetro.
MiClase mc =
    constructor.funcion("prueba referencia
a constructor");
System.out.println("str en mc es la " +
mc.getStr());
}
}
@FunctionalInterface
// La interfaz funcional cuya función devuelve una
referencia a MiClase
interface MiInterface {
    MiClase funcion(String s);
}
```

```
// MiClase tiene dos constructores
class MiClase {
    private String str;
    MiClase(String str)    this.str = s;
    MiClase() {
        str = "";
    }
    String getStr()    return str;
}
```



6. Interfaces funcionales predefinidas

En Java 8 se ha añadido el paquete **java.util.function** en el que se encuentran una serie de interfaces funcionales que se pueden utilizar como tipos de destino de las expresiones lambda y referencias a métodos.

Algunas de ellas son las siguientes:

Interfaz	Función	Funcionalidad
Consumer<T>	void accept(T t)	Una operación a un objeto de tipo T, que no devuelve ningún valor.
Supplier<T>	T get()	Devuelve un objeto de tipo T.
Predicate<T>	boolean test(T t)	Determina si un objeto tipo T cumple una condición. su resultado es true o false.
UnaryOperator<T>	R apply(T t)	Una operación unaria a un objeto de tipo T y devuelve como resultado un tipo también T.
BinaryOperator<T>	R apply(T t, U u)	Una operación a dos objetos de tipo T y devuelve su resultado también de tipo T
Function<T, R>	R apply(T t)	Una operación a un objeto de tipo T que devuelve como resultado un objeto de tipo R.

Tabla 13.4: Interfaces funcionales predefinidas.

En la imagen siguiente se muestra la utilización de la interface **Predicate**:

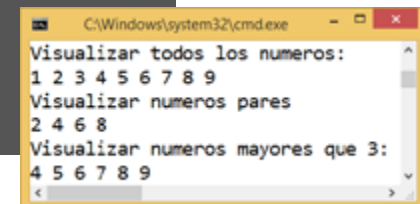
```
public static void main(String args[]){
    List<Integer> list =
        Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9);

    System.out.println("Visualizar todos los numeros:");
    eval(list, n->true);

    System.out.println("Visualizar numeros pares");
    eval(list, n-> n%2 == 0 );

    System.out.println("Visualizar numeros mayores que 3:");
    eval(list, n-> n > 3 );
}
```

```
public static void eval(List<Integer> lista,
    Predicate<Integer>
    predicado){
    for(Integer n: lista) {
        if(predicado.test(n)) {
            System.out.print(n + " ");
        }
    }
    System.out.println();
}
```



7. Stream en Java 8

Un stream desde el punto de vista de programación funcional, no es nada más que un conjunto de funciones que se ejecutan anidadas, unas a continuación de otras, porque el resultado que devuelve cada una es una referencia a un objeto o clase que tiene implementadas todas las funciones que se van anidando.

El formato para la ejecución de dos funciones en forma de stream sería el siguiente:

```
<refe_objeto>.<funcion_01>(<parametros>).  
<funcion_02>(<parametros>);
```

Sin hacer uso de los stream incorporados en Java 8 se pueden ejecutar funciones anidadas tal y como muestra el código siguiente:

```
public class PruebaStreamNoJava8 {  
    public static void main(String args []){  
        Persona p = new Persona("pepe", 45);  
        OperacionesPersona o = new OperacionesPersona(p);  
        o.setNombre("antonio").setEdad(46).pintaPersona();  
    }  
}
```

La clase OperacionesPersona tiene esta implementación:

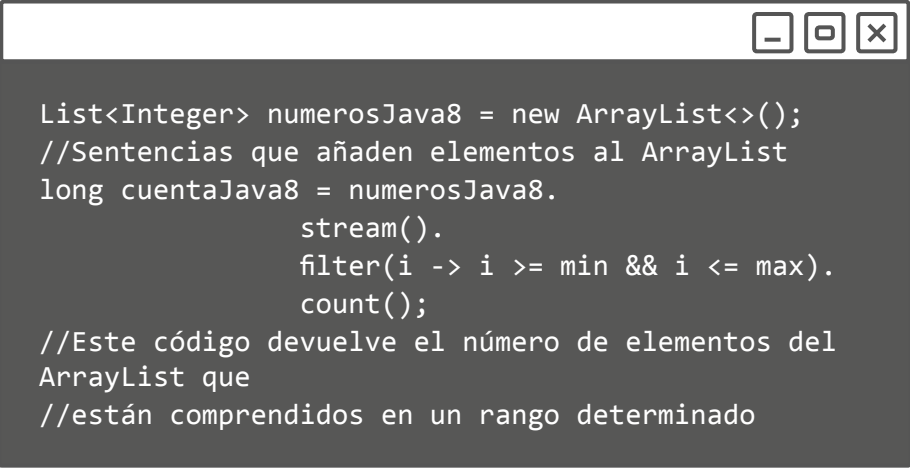
```
public class OperacionesPersona {  
    private Persona persona;  
    public OperacionesPersona(Persona persona) {  
        this.persona = persona;  
    }  
    public Persona getPersona() {  
        return persona;  
    }  
    public void setPersona(Persona persona) {  
        this.persona = persona;  
    }  
    public OperacionesPersona setNombre(String  
nombre){  
        persona.setNombre(nombre);  
        return this;  
    }  
    public OperacionesPersona setEdad(int edad){  
        persona.setEdad(edad);  
        return this;  
    }  
}
```

```
public OperacionesPersona pintaPersona(){
    System.out.println(persona);
    return this;
}
}
```

Como se puede apreciar las funciones que se pueden encadenar en esta forma de stream son las que devuelven una referencia a un tipo que implementa la función siguiente. en este ejemplo las funciones setNombre, setEdad y pintaPersona devuelven la referencia this al objeto OperacionesPersona al que se invocan dichas funciones.

En Java 8 se añade el API de stream que en gran parte permite a todas las clases Collection aplicar esta forma de encadenar llamadas a funciones.

Un ejemplo de utilización de stream se muestra en el código siguiente:



```
List<Integer> numerosJava8 = new ArrayList<>();
//Sentencias que añaden elementos al ArrayList
long cuentaJava8 = numerosJava8.
    stream().
    filter(i -> i >= min && i <= max).
    count();
//Este código devuelve el número de elementos del
//ArrayList que
//están comprendidos en un rango determinado
```

Todas la clases que implementen el interfaz **Collection** o uno de sus derivados (**List**, **Set** o **Queue**) pueden aplicar algunas de las funcionalidades que se describen a continuación. De ellas algunas son **intermedias que producen el stream de acuerdo a su funcionamiento** y otras son **finales que no modifican el stream**:

- **filter**, intermedia que devuelve un stream con los elementos que cumplan la regla de un predicado.

Su parámetro es la interfaz funcional Predicate.

Ejemplo:

```
List<Integer> pares = numeros.stream().filter(n -> n % 2 == 0);
myList.stream().filter(s -> s.startsWith("c"));
```

- **map**, intermedia que devuelve el stream después de aplicar una funcionalidad a cada uno de sus elementos.

Su parámetro es la interfaz funcional Function.

Ejemplo:

```
Arrays.stream(new int[] {1, 2, 3}).map(n -> 2 * n + 1);
Stream.of("a1", "a2", "a3").map(s -> s.substring(1));
```

- **sorted**, intermedia que devuelve el stream ordenado según el orden natural del origen o de acuerdo con el orden establecido con el **Comparator** pasado como parámetro.

Ejemplo:

```
Stream<String> str = Stream.of("aBc", "d", "ef",  
"123456");  
List<String> ordenInverso =  
    str.sorted(Comparator.reverseOrder()).  
    collect(Collectors.toList());  
Stream<String> str = Stream.of("aBc", "d", "ef",  
"123456");  
List<String> ordenNatural = names3.sorted();
```

- **collect**, intermedia que transforma los elementos del stream en por ejemplo List o Set.

Su parámetro suele ser una de las funciones de **Collectors**, clase final del paquete **java.util.stream**, como por ejemplo **toList()** o **toSet()**.

Ejemplos:

```
Stream<Integer> intStream = Stream.of(1,2,3,4);  
List<Integer> intList = intStream.collect(Collectors.  
toList());  
List<Integer> pares = numeros.stream().filter(n -> n %  
2 == 0)  
    .sorted().collect(Collectors.  
toList());
```



- **foreach**, final que recorre todos los elementos del stream realizando una operación con cada uno de ellos.

Su parámetro es la interfaz funcional Consumer.

Ejemplo:

```
Stream<Integer> numeros = Stream.of(1,2,3,4,5);
numeros.forEach(i -> System.out.print(i+","));
Stream.of("d2", "a2", "b1", "b3", "c").map(s -> {
    System.out.println("map: " + s);
    return s.toUpperCase();
})
    .filter(s -> {
        System.out.println("filter: " + s);
        return s.startsWith("A");
    }).forEach(s -> System.out.println("forEach: " + s));
```

- **Otras funciones** terminales son count(), max (Comparator<? super T> comparator), min(Comparator<? super T> comparator), anyMatch(Predicate<? super T> predicate), allMatch(Predicate<? super T> predicate), noneMatch(Predicate<? super T> predicate), findFirst() y findAny().

Ejemplos:

```
Stream.of("d2", "a2", "b1", "b3", "c")
    .map(s -> { System.out.println("map: " + s);
                return s.toUpperCase();
            })
    .anyMatch(s -> { System.out.println("anyMatch: " + s);
                    return s.startsWith("A");
                });
Stream<Integer> numeros = Stream.of(1,2,3,4,5);
System.out.println("¿Stream tiene algún 4? "+
    numeros.anyMatch(i -> i==4));
Stream<Integer> numeros = Stream.of(1,2,3,4,5);
System.out.println("¿todos los elementos del Stream
son menores que 10?" +
    numeros.allMatch(i -> i<10));
Stream<Integer> numeros = Stream.of(1,2,3,4,5);
System.out.println("(¿Stream no tiene algún 10? "+
    numeros.noneMatch(i -> i==10));
```


Hay otro tipo de stream que es **streamParallel**, este tiene la particularidad de dividir en “trozos” el stream y trabajar con cada uno de ellos de forma separada y un thread diferente. cada uno. Todas las funcionalidades anteriores se pueden aplicar a este tipo de stream. En este ejemplo se muestra la diferencia de trabajar con stream en serie o stream en paralelo.

```
public class StreamParalelo {
    public static void main(String args[])
        throws InterruptedException {
        Random aleatorio = new Random();
        List<Integer> numeros = new ArrayList<Integer>();
        for (int i = 0; i < 1_000_000; i++) {
            numeros.add(aleatorio.nextInt());
        }
        long inicio = System.currentTimeMillis();
        List<Integer> pares = numeros.stream()
            .filter(n -> n % 2 == 0)
            .sorted().collect(Collectors.toList());
        System.out.println("SERIE: "+pares.size()
            + " elementos computados en "
            + (System.currentTimeMillis() - inicio) + " con "
            + Thread.activeCount() + " threads");
        inicio = System.currentTimeMillis();
        List<Integer> paresParalelo = numeros.parallelStream()
            .filter(n -> n % 2 == 0)
            .sorted().collect(Collectors.toList());
        System.out.println("PARALELO: "+paresParalelo.size()
            + " elementos computados en "
            + (System.currentTimeMillis() - inicio) + " con "
            + Thread.activeCount() + " threads");
    }
}
```

Una posible respuesta de este código es la siguiente:

SERIE: 499985 elementos computados
en 399 con 1 threads
PARALELO: 499985 elementos computados
en 229 con 4threads

En el código siguiente se muestra la diferencia de procesar una lista de cientos de miles de números, para obtener el número de ellos que se encuentran en un rango determinado.

```
List<Integer> numerosJava8 = new ArrayList<>();
List<Integer> numerosJava7 = new ArrayList<>();
Random r = new Random();
int cantidad = r.nextInt(100000000);
int min = r.nextInt(100);
int max = r.nextInt(100 - min) + min;
// En Java 8
long t1J8 = System.currentTimeMillis();
r.ints(cantidad, 0, 100).
forEach(numerosJava8::add);
    long cuentaJava8 = numerosJava8.stream()
        .filter(i -> i >= min && i <= max).
count();
    long totalJ8 = System.currentTimeMillis() - t1J8;
// En Java 7
long t2J8 = System.currentTimeMillis();
for (int i = 0; i < cantidad; i++)
```

```
        numerosJava7.add(r.nextInt(100));
int cuentaJava7 = 0;
for (Integer i : numerosJava7)
    if (i >= min && i <= max)    cuentaJava7++;
long totalJ7 = System.currentTimeMillis() - t1J8;
System.out.println("Java8 ==> De un total de " +
cantidad
    + " numeros, hay entre " + min + " y " +
max + " : "
    + cuentaJava8 + " contados en "
    + totalJ8 + " milisegundos");
System.out.println("Java7 ==> De un total de " +
cantidad
    + " numeros, hay entre " + min + " y " +
max + " : "
    + cuentaJava7 + " contados en "
    + totalJ7 + " milisegundos");
    }
}
```

Una posible respuesta de este código es la siguiente:

Java8 ==> De un total de 92587677 numeros,
hay entre 60 y 91 : 29624510 contados
en 3218 milisegundos

Java7 ==> De un total de 92587677 numeros,
hay entre 60 y 91 : 29620380 contados
en 8306 milisegundos



Telefonica

EDUCACIÓN DIGITAL