# Programación funcional

# Programación funcional (FP)

- **Programación funcional**. Estilo de programación que enfatiza el uso de *funciones* para descomponer una tarea compleja en subtareas.
- Ejemplos de lenguajes funcionales: LISP, Scheme, ML, Haskell, Erlang, F#, Clojure.
- Java es un lenguaje orientado a objetos, no un lenguaje funcional.
- Pero desde Java 8 se agregaron varias características para facilitar un estilo parcial de programación funcional.

# Programación funcional

- Características principales:
- Programación sin efectos secundarios.
- Funciones de primera clase.
- Cerraduras (closures) de funciones.
- Operaciones de orden superior sobre colecciones.

# Código libre de efectos secundarios

- **Efecto secundario**. Un cambio en el estado de un objeto o variable de programa producido por una llamada a un método.
- Ejemplo: modificar el valor de una variable no local.
- Ejemplo: impresión de salida en System.out.
- Ejemplo: leer / escribir datos en un archivo, colección o red.

```
int resultado = f(x) + f(x);
int resultado = 2 * f (x);
```

- ¿Son las dos declaraciones anteriores iguales?
  - Sí, si la función f() no tiene efectos secundarios.
  - Uno de los objetivos de la programación funcional es minimizar los efectos secundarios.

# Ejemplo con efectos secundarios 1

```
public class SideEffect {
    public static int x = 0;
    public static int f(int n) {
        n++;
        return 2 * n;
    }
    public static void main(String[] args) {
        int result = f(x) + f(x);      // comparar las 2 llamadas
        //int result = 2 * f(x);
        System.out.println(result);
    }
}
```

# Ejemplo con efectos secundarios 2

```
public class SideEffect {
    public static int x;
    public static int f(int n) {
        x = x * 2;
        return x + n;
    }
    public static void main(String[] args) {
        x = 5;
        int result = f(x) + f(x);     // comparar las 2 llamadas
        // int result = 2 * f(x);
        System.out.println(result);
    }
}
```

# Funciones de primera clase

- **Ciudadano de primera clase**. Un elemento de un lenguaje de programación que está integrado con el lenguaje y es compatible con la gama completa de operaciones disponibles para otros elementos del lenguaje.
- En los lenguajes que soportan programación funcional, las funciones se tratan como ciudadanos de primera clase:
  - Se puede almacenar una función en una variable.
  - Se puede pasar una función como parámetro a otra función.
  - Se puede devolver un valor de una función.
  - Se puede crear una colección de funciones.

# Ejemplo: tutor de sumas / multiplicaciones

- Considerar un programa que proporcione al usuario problemas de suma y multiplicación:

  9 + 6 = 15

  es correcto

  3 * 7 = 18

  incorrecto... la respuesta es 21

# Código con lambdas

- Se puede representar la operación aritmética como una expresión lambda.

```
Scanner console = new Scanner(System.in);
// examina al usuario con 3 problemas de sumas
giveProblems(console, 3, “+”, (x, y) -> x + y);
// examina al usuario con 3 problemas de multiplicaciones
giveProblems(console, 3, “*”, (x, y) -> x * y);
```

# Método giveProblems

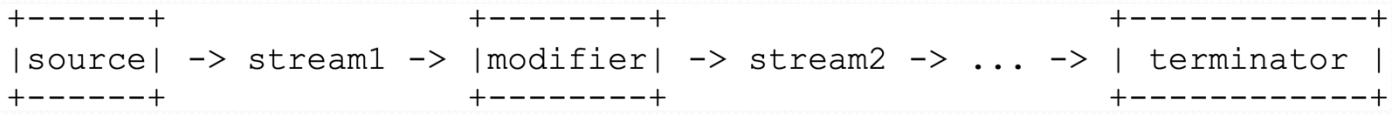
```
public static void givePro blems(Scanner console, int count, String text,
                                 IntBinaryOperator operator) {
    Random r = new Random();
    int numRight = 0;
    for (int i = 1; i <= count; i++) {
        int x = r.nextInt(12) + 1;
        int y = r.nextInt(12) + 1;
        System.out.print(x + " " + text + " " + y + " = ");
        int answer = operator.applyAsInt(x, y);
        int response = console.nextInt();
```

# Método giveProblems

```
        if (response == answer) {
            System.out.println("es correcto");
             numRight++;
        } else {
            System.out.println("incorrecto... la respuesta es " + answer);
        }
    }
    System.out.println(numRight + " de " + count + " correctas");
    System.out.println();
}
```

# Streams

- **Stream**. Una secuencia de elementos de una fuente de datos que admite operaciones agregadas.
- Los streams operan en una fuente de datos y la modifican:




- Ejemplo: imprimir cada elemento de una colección.
- Ejemplo: sumar cada entero en un archivo.
- Ejemplo: concatenar strings en un string grande.
- Ejemplo: encontrar el valor más grande en una colección.

# Streams

- Un stream consta de una *fuente*, cero o más *operaciones intermedias* y una *operación terminal*.
- La fuente puede ser un arreglo, una colección, una función generadora, un canal de I/O, etc.)
- Las operaciones intermedias transforman un stream en otro stream.
- La operación terminal produce una salida o un efecto secundario.
- Los streams son *flojos* (*lazy*).
- El cálculo de los datos de la fuente solo se realiza cuando comienza la operación terminal.
- Los datos de la fuente se consumen solo cuando es necesario.

# Ejemplo sin streams

```
// Calcula la suma de los cuadrados de los enteros entre 1 y 5
int sum = 0;
for (int i = 1; i <= 5; i++) {
    sum = sum + i * i;
}
```

# Ejemplo con streams

```
// Calcula la suma de los cuadrados de los enteros de 1 a 5
int sum = IntStream.range(1, 6).map(n -> n * n).sum();
```

- IntStream es la fuente. Produce un stream entero con los números 1, 2, 3, 4, 5.
- El modificador map aplica una lambda a cada elemento de un stream y produce un stream de salida.
- Es una **función de orden superior**. Toma una función como argumento.
- sum es la operación terminal.

# Ejemplo con streams

- Las operaciones de stream son como siguen:

IntStream.range(1, 6) -> [1, 2, 3, 4, 5]

-> map -> [1, 4, 9, 16, 25]

-> sum -> 55

# Streams

- La clase Stream<T> representa una secuencia de objetos de tipo T.
- Las clases IntStream, LongStream y DoubleStream representan streams de tipos primitivos int, long y double, respectivamente.

# Fuentes

- Formas de obtener un stream:
- Desde una Collection a través de los métodos stream() y parallelStream().
- Desde un arreglo a través de Arrays.stream(Object []).
- Usando métodos en la clase Stream como Stream.of(Object []), IntStream.range(int, int), Stream.iterate(Object, UnaryOperator), etc.
- Las líneas de un archivo se pueden obtener de BufferedReader.lines().
- También se obtienen de Files.lines(Paths.get("nombre")).
- Se pueden obtener secuencias de números aleatorios de Random.ints().

# Fuentes

- Numerosos otros métodos de stream, incluidos BitSet.stream(), Pattern.splitAsStream(java.lang.CharSequence) y JarFile.stream().

# Operaciones intermedias

- Las operaciones intermedias devuelven otro stream.
- Esto permite llamar varias operaciones en cadena.
- Las operaciones intermedias son flojas.
- No se ejecutan hasta que se invoca la operación terminal.

# Operaciones intermedias

| Operación | Descripción |
|---|---|
| filter() | Devuelve los elementos que coinciden con el predicado dado |
| map() | Aplica la función dada a cada elemento del stream |
| flatMap() | Aplana un stream |
| distinct() | Devuelve elementos únicos del stream |
| sorted() | Devuelve los elementos del stream en orden |
| peek() | Examina solo el primer elemento del stream |
| limit() | Devuelve un número limitado de elementos del stream |
| skip() | Omite un número de elementos del stream |

# Operaciones terminales

- Las operaciones terminales devuelven valores que no son stream.
- Pueden devolver algún tipo primitivo, objeto, colección, o nada (void).
- Un stream solo puede tener una operación terminal y debe ser la última operación.

# Operaciones terminales

| Operación | Tipo | Descripción |
|---|---|---|
| forEach | void | Realiza una acción en todos los elementos del stream |
| toArray | Object[] | Regresa un arreglo conteniendo los elementos del stream |
| reduce | tipo T | Realiza una operación de reducción en los elementos del stream usando un valor inicial y una operación binaria |
| collect | Contenedor de tipo T | Regresa un contenedor mutable como List o Set |
| min | Optional<T> | Regresa el elemento mínimo en el stream |
| max | Optional<T> | Regresa el elemento máximo en el stream |
| count | long | Regresa el número de elementos del stream |

# Operaciones terminales

| Operación | Tipo | Descripción |
|---|---|---|
| anyMatch | boolean | Regresa true si alguno de los elementos del stream coincide con el predicado |
| allMatch | boolean | Regresa true si todos los elementos del stream coinciden con el predicado |
| noneMatch | boolean | Regresa true si ninguno de los elementos del stream coincide con el predicado |
| findFirst | Optional<T> | Regresa el primer elemento del stream |
| findAny | Optional<T> | Regresa un elemento aleatorio del stream |

# Operación map

- Devuelve un stream que consta de los resultados de aplicar la función dada a los elementos del stream de entrada.

```
List<Integer> list = Stream.of("Bienvenidos", "a", "Hermosillo", "Sonora")
                          .map(String::length).collect(Collectors.toList());
System.out.println(list);          // [11, 1, 10, 6]
```

# Operación map

```
// Convierte un conjunto de palabras a minúsculas
List<String> words = Arrays.asList("To", "be", "or", "Not", "to", "be");
Set<String> words2 = words.stream().map(String::toLowerCase)
                .collect(Collectors.toSet());
System.out.println("word set = " + words2);
```

**output:**

word set = [not, be, or, to]

# Operación map

- Java tiene diferentes tipos de streams.
- El tipo se conserva a menos que se indique explícitamente que el tipo de stream cambia.
- Si el stream es de objetos y se requiere aplicar una operación con algún tipo primitivo, es necesario indicar que el tipo del stream va a cambiar.

# Operación map

- Variantes de map:
- map(). Mapea un stream de objetos de tipo T a un stream de objetos de tipo R.
- mapToInt(). Mapea un stream de objetos a un stream de primitivos int.
- mapToLong(). Mapea un stream de objetos a un stream de primitivos long.
- mapToDouble(). Mapea un stream de objetos a un stream de primitivos double.

# Operación mapToInt

```
// Encuentra la línea más larga en el archivo
int longest = Files.lines(Paths.get("haiku.txt")).mapToInt(String::length)
                    .max().getAsInt();
```

**operaciones de stream:**

Files.lines -> ["haiku are funny", "but sometimes they don't make sense", "refrigerator"] -> mapToInt -> [15, 35, 12] -> max -> 35

# Operación mapToInt

```
// Suma de las longitudes de las palabras
List<String> words = Arrays.asList(“Ser”, “o”, “no”, “ser”);
int totalLength = words.stream()
    .mapToInt(String::length)
    .sum();              // 9
```

# Operación filter

- La operación filter mantiene elementos del stream que cumplen con un predicado.

```
// Calcula la suma de los cuadrados de los enteros impares
int sum = IntStream.of(3, 1, 4, 1, 5, 9, 2, 6, 5, 3)
            .filter(n -> n % 2 != 0).map(n -> n * n).sum();
```

- Las operaciones de flujo son como siguen:

```
IntStream.of  -> [3, 1, 4, 1, 5, 9, 2, 6, 5, 3]
        -> filter -> [3, 1, 1, 5, 9, 5, 3]
        -> map -> [9, 1, 1, 25, 81, 25, 9]
        -> sum -> 151
```

# Operación flatMap

- La operación flatMap se utiliza para procesar arreglos de dos dimensiones (matrices).
- Reemplaza cada elemento del stream con el contenido de aplicar la función de mapeo indicada a cada elemento.
- Ejemplo con una expresión lambda:

```
int[][] a = { {1, 2, 3}, {4, 5, 6}, {7, 8, 9} };
IntStream stream = Arrays.stream(a).flatMapToInt(x -> Arrays.stream(x));
```

- Ejemplo con una referencia a método:

```
IntStream stream = Arrays.stream(a).flatMapToInt(Arrays::stream);
```

# Operación flatMap

- Operaciones de flujo:

```
IntStream stream = Arrays
        .stream(a)                          // Stream de int[]
        .flatMapToInt(Arrays::stream);      // Stream de enteros
```

- La matriz se procesa por renglón.

# Operación limit

- Devuelve un stream que consta de los elementos del stream de entrada truncado para que no tenga una longitud más grande de la indicada.

```
Random random = new Random();
// Genera un arreglo con 10 números aleatorios dobles
double[] lista = random.doubles().limit(10).toArray();
```

# Operación sorted

- Hay dos variantes:
- sorted() – devuelve un stream que consta de los elementos del stream de entrada, ordenados según el orden natural.
- sorted(Comparator) – devuelve un stream que consta de los elementos del stream de entrada, ordenados según el Comparator proporcionado.

# Operación sorted

```
// Genera una lista con 10 números aleatorios enteros, entre 100 y 999, ordenados de menor a mayor
Random random = new Random();
List<Integer> lista1 = random.ints(10, 100, 1000).sorted().boxed().toList();

// Ordenados de mayor a menor
List<Integer> lista2 = random.ints(10, 100, 1000).boxed()
        .sorted(Collections.reverseOrder()).toList();
```

# Operación skip

- Devuelve un stream que consta de los elementos restantes del stream de entrada después de descartar el número de elementos indicado.

```
Stream.of("one", "two", "three", "four", "five").skip(2)
        .forEach(System.out::println);    // three, four, five
```

# Operación reduce

- La operación terminal reduce permite producir un único resultado a partir de una secuencia de elementos, aplicando repetidamente una operación de combinación a los elementos.
- Hay tres variantes:

1. Optional<T> **reduce**(BinaryOperator<T> accumulator)
2. T **reduce**(T identity, BinaryOperator<T> accumulator)
3. <U> U **reduce**(U identity, BiFunction<U, ? super T, U> accumulator, BinaryOperator<U> combiner)

# Elementos de reduce

- Identity: un elemento que es el valor inicial de la operación de reduce y el resultado por default si el stream está vacío.
- Accumulator: una función para incorporar un elemento adicional al resultado.
- Combiner: una función para combinar dos valores, los cuáles deben ser compatibles con la función del acumulador. No es necesario si los tipos de los argumentos del acumulador coinciden (*match*) con los tipos de su implementación.

# Versión 1

```
Optional<T> reduce(BinaryOperator<T> accumulator)
// Acumula nombres en minúscula en un string
String[] names = {“Ana”, “Blanca”, “Carlos”, “Daniel”, “Erika”, “Fernando”, “Gabriela”, “Héctor”, “Isabel”, “Juan”};
Optional<String> reducedValue = Arrays.stream(names).map(String::toLowerCase)
        .reduce((name1, name2) -> name1 + “, ” + name2);
if (reducedValue.isPresent()) {
    String s = reducedValue.get();
    System.out.println(s);  // ana, blanca, carlos, daniel, erika, fernando, gabriela, hector, isabel, juan
}
```

# Versión 2

```
T reduce(T identity, BinaryOperator<T> accumulator)
// Suma los elementos de un arreglo
int[] numbers = {123, 456, 789, 246, 135, 802, 791};
int suma = Arrays.stream(numbers).reduce(0, (x, y) -> (x + y));
System.out.println("suma = " + suma);          // 3342
```

# Versión 2

```
// Regresa n!, = 1 * 2 * 3 * ... * (n - 1) * n.
// Se supone que n es no negativa.
public static int factorial(int n)
{
    return IntStream.range(2, n + 1).reduce(1, (a, b) -> a * b);
}
```

# Versión 3

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

```
// Suma las edades de una lista de usuarios
List<User> users = Arrays.asList(new User("Juan", 30),
                                 new User("Julia", 35));
// Esto no compila
int computedAges = users.stream()
   .reduce(0, (partialAgeResult, user) -> partialAgeResult + user.getAge());
```

**Operator '+' cannot be applied to 'User', 'int'**

# Versión 3

- Se tiene un stream de objetos User y los tipos de argumentos del acumulador son Integer y User.
- La implementación del acumulador es una suma de enteros.
- El compilador no puede inferir el tipo de parámetro User.
- El problema se resuelve usando un combiner.

```
int result = users.stream()
        .reduce(0, (partialAgeResult, user) -> partialAgeResult + user.getAge(), Integer::sum);
System.out.println("Resultado: " + result);     // 65
```

# Streams y métodos

- Usando streams como parte de un método regular:

```
// Regresa true si el entero dado es primo.
// Se supone n >= 0.
public static boolean isPrime(int n)
{
    return IntStream.range(1, n + 1).filter(x -> n % x == 0).count() == 2;
}
```

# Resultados opcionales

- Algunos terminadores de streams, como max, devuelven un resultado "opcional" porque el stream puede estar vacío o no contener el resultado:

```
// imprime el múltiplo más grande de 10 en lista
// (no compila)
int largest = IntStream.of(55, 20, 19, 31, 40, -2, 62, 30)
                       filter(n -> n % 10 == 0).max();
System.out.println(largest);
```

# Resultados opcionales

- Para extraer el resultado opcional, se usa un terminador “get as” para convertir el tipo OptionalInt en Integer.

```
// imprime el múltiplo de 10 más grande en lista
// (esta versión compila y funciona.)
int largest = IntStream.of(55, 20, 19, 31, 40, -2, 62, 30).
                        filter(n -> n % 10 == 0).max().getAsInt();
System.out.println(largest);
```

# Variables libres / ligadas

- En una expresión lambda:
- Los parámetros son **variables ligadas**.
- Las variables del ámbito externo son **variables libres**.

# Variables libres / ligadas

- Ejemplo:

```
// Imprime el número de factores
int n = 10;
long count = IntStream.range(1, n + 1)
        .filter(x -> n % x == 0)
        .count();
System.out.println("count = " + count);          // 4 (1, 2, 5, 10)
```

- En la instrucción filter(x -> n % x == 0), x es una variable ligada y n es una variable libre.

# Cerraduras de función

- **Cierre de función (function closure)**. Un bloque de código que define una función junto con las variables libres definidas en su ámbito.
- Ejemplo 1:

```
compute((x, y) -> x + y);
```

- `x` y `y` son parámetros, es decir, son variables ligadas.
- Este código no tiene variables libres.

# Cerraduras de función

- Ejemplo 2:

```
int min = 10;
int max = 50;
int multiplier = 3;
compute((x, y) -> Math.max(x, min) * Math.max(y, max) * multiplier);
```

- x y y son variables ligadas porque son parámetros.
- min, max y multiplier son variables libres.
- Java necesita incluir las definiciones de las variables ligadas y las variables libres para que compute haga su trabajo.

# Cerraduras de función

- Una cerradura de función puede visualizarse así:

```
parameters     : (x, y)
free variables: min = 10, max = 50, multiplier = 3
code           : Math.max(x, min) * Math.max(y, max) * multiplier
```

- Las variables libres se incluyen para que el código puede ejecutarse.
- Se dice que, al formarse esta cerradura, han sido *capturadas*.
- Las variables libres deben estar declaradas `final` o ser *efectivamente* finales (una sola asignación).

# Funciones de orden superior

- **Función de orden superior (higher-order function)**. Es una función que toma como argumento otra función.
- Ejemplos: los métodos map, filter y reduce son funciones que aceptan otras funciones como argumentos.
- Otros ejemplos. List.sort, ArrayList.forEach, Arrays.compare, etc.
- Las funciones de orden superior se pueden aplicar a colecciones (arreglos, listas, conjuntos, mapas).

# Funciones de orden superior en arreglos

- Un arreglo puede ser fuente de un stream usando Arrays.stream.

```
// Suma los valores absolutos de los enteros pares quitando los duplicados
int[] numbers = {3, -4, 8, 4, -2, 17, 9, -10, 14, 6, -12};
int sum = Arrays.stream(numbers)
        .map(Math::abs)          // función de orden superior
        .filter(n -> n % 2 == 0) // función de orden superior
        .distinct()
        .sum();                  // 56
```

# Funciones de orden superior en arreglos

- Varias clases de stream tienen un terminador llamado toArray que recopila el contenido del stream en un arreglo.
- Ejemplo:

```
int[] numbers = {3, -4, 8, 4, -2, 17, 9, -10, 14, 6, -12};
int[] sublist = Arrays.stream(numbers)
    .map(Math::abs)                 // función de orden superior
    .filter(n -> n % 2 == 0)        // función de orden superior
    .distinct()
    .toArray();       // [4, 8, 2, 10, 14, 6, 12]
```

# Funciones de orden superior en listas

- La interface List incluye el método stream para crear un stream de valores a partir de una lista.

```
List<String> words = Arrays.asList("To", "be", "or", "Not", "to", "be");
System.out.print("words:");
words.stream()
    .forEach(s -> System.out.print(" " + s));      // función de orden superior
System.out.println();
```

# Funciones de orden superior en listas

- Se obtiene el mismo resultado usando el método map y una referencia al método System.out.print.

```
System.out.print("words:");
words.stream()
    .map(s -> " " + s)
    .forEach(System.out::print);
System.out.println();
```

# Funciones de orden superior en listas

```
// Convierte a minúsculas, quita duplicados y ordena las palabras
System.out.print("words:");
words.stream()
    .map(String::toLowerCase)    // función de orden superior
    .distinct()
    .sorted()
    .map(s -> " " + s)           // función de orden superior
    .forEach(System.out::print); // función de orden superior
System.out.println();            // words: be not or to
```

# Funciones de orden superior en listas

- Los valores de un stream se pueden recolectar en una colección usando el método collect.
- Se necesita crear un objeto Collectors llamando al método apropiado como toList o toSet.
- Ejemplo:

```
Set<String> words2 = words.stream()
    .map(String::toLowerCase)          // función de orden superior
    .collect(Collectors.toSet());      // función de orden superior
System.out.println("word set = " + words2);   // word set = [not, be, or, to]
```

# Funciones de orden superior en listas

- Se puede especificar el tipo de colección pasando una referencia al constructor apropiado.
- Por ejemplo, para garantizar un TreeSet que mantiene las llaves en orden:

```
Set<String> words2 = words.stream()
    .map(String::toLowerCase)    // función de orden superior
    .collect(Collectors.toCollection(TreeSet::new)); // función de orden superior
System.out.println("word set = " + words2);   // word set = [be, not, or, to]
```

# Funciones de orden superior en archivos

```
// Imprime cada línea del archivo
try {
    Files.lines(Paths.get("haiku.txt"))
        .forEach(System.out::println);      // función de orden superior
}
catch (IOException e) {
    System.out.println("Error al leer el archivo: " + e);
}
```

# Funciones de orden superior en archivos

- Se pueden procesar las líneas usando map, reduce y filter.
- Ejemplo: encontrar la longitud de la línea más larga.

```
int longest = Files.lines(Paths.get("haiku.txt"))
    .mapToInt(String::length)      // función de orden superior
    .max()
    .getAsInt();      // 35
```

# Resumen

- La programación funcional es un estilo que enfatiza el uso de funciones o métodos para descomponer problemas.
- Java 8 agregó elementos al lenguaje para soportar programación funcional.
- Un efecto secundario es un cambio al estado del programa producido cuando una función es invocada.
- Ejemplos de efectos secundarios: modificar una variable global, escribir en un archivo, imprimir en la consola.
- Los programadores funcionales tratan de evitar los efectos secundarios lo más que pueden.

# Resumen

- Una función de primera clase es una función que puede ser tratada como otros tipos de datos.
- Por ejemplo, ser asignada o ser pasada como argumento.
- Java proporciona una sintaxis abreviada para definir funciones anónimas llamadas expresiones lambda.
- Un stream es una secuencia de elementos de una fuente de datos que soporta operaciones agregadas.
- Arreglos, colecciones, strings, rangos de enteros, archivos y otras fuentes pueden ser convertidos en streams.

# Resumen

- Las operaciones típicas sobre streams incluyen `map` (aplicar una operación a cada elemento), `filter` (mantener o remover elementos basado en algún criterio) y `reduce` (combinar múltiples elementos en un solo elemento).
- Una cerradura (closure) es una definición de una función junto con las definiciones de cualquier variable declarada fuera de la función (variables libres) que la función utilice.
- Una función de orden superior es una función que acepta otra función como argumento.
- Java soporta una forma limitada de funciones de orden superior a través de las referencias a métodos.