PROGRAMACIÓN FUNCIONAL Y STREAMS

INTERFACES FUNCIONALES

Caso práctico: Comparación de personas

Deseamos ordenar por criterios distintos cada vez (id, fecha, etc.)

- Alternativa 1: definir subclases `PersonaPorNombre`, `PersonaPorFechaNacimiento`...
  - Mucho código repetido (no cumple DRY)
  - Muchos cambios si se añade un nuevo criterio (no cumple OCP)
- Alternativa 2: no usar herencia, sino composición/delegación
  - Factorizar la *función* de comparación
  - No delegar hacia las subclases
  - Delegar en objeto de otra clase que implemente la interfaz `java.util.Comparator`

Usando composición/delegación:

```
class OrdenarPersonaPorId implements java.util.Comparator<Persona> {
    public int compare(Persona o1, Persona o2) {
        return o1.getIdPersona() - o2.getIdPersona();
    }
}

Collections.sort(personas, new OrdenarPersonaPorId());
```

La **función factorizada** (la implementación de `Comparator`) es sustituible en tiempo de ejecución mediante inyección de dependencias

Clases anónimas

Comparador: versión con clases anónimas

```
Collections.sort(personas,
  new java.util.Comparator<Persona>() {
    public int compare(Persona o1, Persona o2) {
      return o1.getIdPersona() - o2.getIdPersona();
    }
  }
);
```

Clases anónimas (Java 7)

```
public class ComparatorTest {
  public static void main(String[] args) {
    List<Person> personList = Person.createShortList();

    Collections.sort(personList, new Comparator<Person>(){
      public int compare(Person p1, Person p2){
        return p1.getLastname().compareTo(p2.getLastname());
      }
    });
    System.out.println("=== Sorted Asc Lastname ===");
    for(Person p: personList){
      p.printName();
    }

    Collections.sort(personList, new Comparator<Person>(){
      public int compare(Person p1, Person p2){
        return p2.getLastname().compareTo(p1.getLastname());
      }
    });
    System.out.println("=== Sorted Desc Lastname ===");
    for(Person p: personList){
      p.printName();
    }
  }
}
```

Lambdas (Java 8)

```
public class ComparatorTest {
  public static void main(String[] args) {

    List<Person> personList = Person.createShortList();

    // Print Asc
    System.out.println("=== Sorted Asc Lastname ===");
    Collections.sort(personList, (Person p1, Person p2) ->
      p1.getLastname().compareTo(p2.getLastname()));
    for(Person p:personList){
      p.printName();
    }

    // Print Desc
    System.out.println("=== Sorted Desc Lastname ===");
    Collections.sort(personList, (p1,  p2) ->
      p2.getLastname().compareTo(p1.getLastname()));
    for(Person p:personList){
      p.printName();
    }
  }
}
```

Clases Internas (*inner*)

- Son clases locales anónimas, declaradas dentro de métodos
- Pueden hacer referencia a identificadores declarados en la clase contenedora y a variables locales `final` (o *effectively final*) del método en que se declaran
- Aglutinan funcionalidades que solo se necesitan una vez en la aplicación

▷ Viven en el *heap*

▷ Capturan el valor de las variables locales

```
public class Enclosing {
  public class Inner {
    public int incrementAndReturn() {
      return counter++;
    }
  }

  private int counter = 0;

  public int getCounter() {
    return counter;
  }

  public static void main(String[] args) {
    var anEnclosing = new Enclosing();
    var anInner = anEnclosing.new Inner();
    int value;

    while ((value = inner.incrementAndReturn()) < 10)
    {
        System.out.println(value);
    }
  }
}
```

Predicados

Ejemplo: partidos de una competición

Queremos iterar sobre una colección de partidos de una competición y quedarnos sólo con los partidos que enfrentan a dos equipos concretos.

- ¿Cómo implementamos el **criterio** de filtrado?
- ¿Cómo resolvemos el retorno de `null` en algún paso de la iteración?

Guava y Java

Guava es una aitigua biblioteca open source de Google que proporciona una amplia gama de utilidades para Java, incluyendo colecciones, cachés, primitivas, concurrencia, etc.

Iteración:

- Guava usa una interfaz funcional `com.google.common.base.Predicate` (inspiraron la clase `java.util.function.Predicate` de Java) —Guava define `Predicate::apply()` y Java define `Predicate::test()`
- Guava añade un método `filter` a los `Iterators`, que recibe un `Predicate` como criterio de filtrado
- Guava no comprueba `null` en la programación fluent

Ejemplo con Guava: null en la iteración

```
import java.util.Iterator;
import com.google.common.base.Predicate;
import com.google.common.collect.Iterators;

Iterator<Match> matches = repository.findMatches(); // puede ser null

final Predicate<Match> condition = new Predicate<>() {
    final Team team1 = new Team("Cadiz CF");
    final Team team2 = new Team("RC Betis");

    @Override
    public boolean apply(Match match) {
        return match.getLocalTeam().equals(team1)
            && match.getVisitingTeam().equals(team2);
    }
};

Iterator<Match> matchesByTeam =
    Iterators.filter(matches, condition); // 💥 Excepción si matches == null

while (matchesByTeam.hasNext()) {
    System.out.println(matchesByTeam.next());
}
```

Supongamos que `matches` se obtiene de una API heredada que puede devolver `null`. Entonces al hacer...

```
Iterator<Match> matchesByTeam = Iterators.filter(matches, condition);
```

- El pipeline espera un iterador, no un `null`.
- Si `matches` es `null`, se lanzará una excepción `NullPointerException` al intentar iterar sobre `matchesByTeam`.

Ejemplo con Java 9+: null en la iteración

```
import java.util.Iterator;
import java.util.Spliterators;
import java.util.function.Predicate;
import java.util.stream.Stream;
import java.util.stream.StreamSupport;

class MatchByTeamsPredicate implements Predicate<Match> {
  private final Team localTeam;
  private final Team visitingTeam;

  MatchByTeamsPredicate(Team localTeam, Team visitingTeam) {
    this.localTeam = localTeam;
    this.visitingTeam = visitingTeam;
  }

  @Override
  public boolean test(Match match) {
    return match.getLocalTeam().equals(localTeam)
        && match.getVisitingTeam().equals(visitingTeam);
  }
}
```

```
Stream<Match> toStream(Iterator<Match> iterator) {
    return StreamSupport.stream(
        Spliterators.spliteratorUnknownSize(iterator, 0),
        false
    );
}

void main() {
  Iterator<Match> matches = repository.findMatches(); // puede ser null

  Predicate<Match> condition =
      new MatchByTeamsPredicate(
          new Team("Cadiz CF"),
          new Team("RC Betis")
      );

  Stream.ofNullable(matches)
      .flatMap(this::toStream)
      .filter(condition)
      .forEach(System.out::println);
}
```

Guava y Java 8

Programación fluent y retorno de `null` en los iterables:

- Guava usa `FluentIterable` para encadenar varios `Iterable`
- Java 8 sustituye el `FluentIterable` por los `Predicate` o por el uso de `StreamSupport`
- Java 9 introduce `Stream.ofNullable()`

Lectura recomendada: From Guava's FluentIterable via StreamSupport to Java 8 Streams

Colecciones immutables:

- Implementaciones immutables de colecciones: `ImmutableList`, `ImmutableSet`, `ImmutableMap`... en Guava
- Adoptadas con `List.of()`, `Set.of()`, `Map.of()`... en Java 8

Ejemplo con Java 8+: criterios de filtrado

Comprobar que un mismo partido (*fixture*) no está repetido ni se enfrenta un equipo contra sí mismo en un grupo de la competición...

```
import java.util.List;
import java.util.function.Predicate;

class FixturePredicate implements Predicate<Match> {
    private final Team localTeam;
    private final Team visitingTeam;

    public FixturePredicate(Team local, Team visiting) {
        this.localTeam = local; this.visitingTeam = visiting;
    }

    @Override
    public boolean test(Match match) {
        return match.getLocalTeam().equals(localTeam)
            && match.getVisitingTeam().equals(visitingTeam);
    }
}

private Predicate<Match> fixture(Team localTeam, Team visitingTeam) {
    return new FixturePredicate(localTeam, visitingTeam);
}
```

```
private void checkMatchesInGroup(List<Match> matchesInGroup) {
    for (Match match : matchesInGroup) {
        Team t1 = match.getLocalTeam();
        Team t2 = match.getVisitingTeam();

        assertNotSame(t1, t2);

        List<Match> firstLeg = matchesInGroup.stream()
            .filter(fixture(t1, t2))
            .toList();

        assertTrue(firstLeg.size() == 1);

        List<Match> secondLeg = matchesInGroup.stream()
            .filter(fixture(t2, t1))
            .toList();

        assertTrue(secondLeg.size() == 0);
    }
}
```

LAMBDAS

Funciónes anónimas o *lambdas*

- Función o subrutina definida y (posiblemente) llamada sin necesidad de asociarla a un identificador o nombre
- Se suelen pasar como argumento a funciones de orden superior
- Son funciones anidadas que permiten acceder a variables definidas en el ámbito de la contenedora (variables no locales a la función anónima)
- Muchos lenguajes las introducen a través de la palabra reservada `lambda`

Expresión lambda

Una expresión *lambda* es una función anónima (con o sin parámetros) que es llamada sin necesidad de asociarle un nombre explícito.

Sirven para pasarlas como argumento a funciones de orden superior, momento en el cual los parámetros de la función anónima toman un valor en el contexto de ejecución de la función contenedora que la recibe y ejecuta.

Por tanto, las funciones anónimas permiten acceder a variables (no locales) definidas en el ámbito de la contenedora.

Lambdas en los lenguajes

Mecanismos de los lenguajes para implementar funciones anónimas:

- En C++: funciones anónimas, objetos función (*functors*) o funciones lambda (desde C++11)
- En Java 8: expresiones lambda
- En Ruby: blocks, procs y lambdas
- En C#: delegates (métodos anónimos y expresiones lambda)
- En Python: generators, comprehensions, lambda expressions

Lambdas en Java

Sintaxis

```
( argumentos ) -> expresión
```

Ejemplos

```
(int x, int y) -> x + y
() -> 42
(String s) -> { System.out.println(s); }
```

Lambdas en Ruby

Blocks

```
{ | argumentos | expresión }
```

Lambdas

```
->(argumentos) { expresión }
```

Ejemplos

```
[1, 2, 3].each { |num| puts num * 2 }
# 2 4 6

times_two = ->(x) { x * 2 }
times_two.call(10)
# 20
```

Lambdas en C++

Sintaxis

```
[capture](parameters) -> return_type { body }
```

capture = entorno de referencia

`[]` – Sin variables externas definidas. Erróneo intentar utilizar cualquier variable externa

`[x, &y]` – `x` se captura por copia, `y` por referencia

`[&]` – Toda variable externa utilizada es capturada implícitamente por referencia

`[=]` – Toda variable externa utilizada es capturadas implícitamente por copia

`[&, x]` – `x` se captura explícitamente por copia; el resto, por referencia

`[=, &z]` – `z` se captura explícitamente por referencia; el resto, por copia

Clausuras o *closures*

- **Clausura** = función o referencia a función junto con un *entorno de referencia*
  - La diferencia entre una función normal y una clausura es que una clausura depende de una o varias **variables libres**.
  - Una clausura permite acceder a las variables libres fuera de su ámbito léxico (i.e. alcance), incluso cuando se invoca desde fuera de ese ámbito.

Entorno de referencia de una clausura

- Tabla que guarda una referencia a cada una de las variables no locales (*libres*) de la función
    - **Variable libre** (*free*): notación lógica matemática que especifica los lugares de una expresión donde tiene lugar una sustitución
    - **Variable asignada** (*bound*): variable que era libre previamente pero a la que le ha sido asignado un valor o conjunto de valores

Anónimas y clausuras en C++

```
std::vector<int> some_list; // assume that contains something
int total = 0;
for (int i=0;i<5;++i) some_list.push_back(i);
std::for_each(
  begin(some_list),
  end(some_list),
  [&total](int x) { total += x; }
);
// Calcula la suma total de los elementos de la lista.
/*
   La variable total se guarda como parte de la clausura de la
   función lambda. Como es una referencia a la variable total de la
   pila de ejecución, puede cambiar su valor.
*/
```

```
[capture](parameters) -> return_type { body }
```

- Una *clausura* en C++ se expresa mediante la parte `[capture]`
- El *entorno de referencia* se expresa por el conjunto de variables externas indicadas dentro de la clausura
- Las variables del entorno de referencia en C++ pueden ser capturadas por copia (`[=]`) o por referencia (`[&]`)
- Mutabilidad de variables en `body`
  - Las variables externas capturadas son inmutables por defecto
  - `mutable` después de los (`parameters`): permite que `body` modifique los objetos capturados por copia

Lecturas recomendadas: Lambdas en C++

- Lambda Functions in C++11 - the Definitive Guide
- La historia de las lambdas en C++: Parte 1 y Parte 2

Tutoriales recomendado:

- Mejorando código con expresiones lambda
- Closures en Scala

Anónimas y clausuras en Java

Captura de variables en lambdas

Una expresión lambda en Java puede **capturar** (o no)...

- variables de instancia no locales (atributos de la clase contenedora) y
- variables locales (declaradas o no `final`, pero cuyo valor no es modificado)

...del ámbito contenedor.

Lambdas y clases anónimas internas

En Java, una expresión lambda y una *inner class* tienen un propósito similar, pero son diferentes en el **ámbito** (*scope*) de definición de las variables locales.

- Con una inner class se crea un **nuevo ámbito** para la clase
  - Se pueden ocultar las variables locales para el ámbito contenedor instanciando nuevas variables con el mismo nombre
  - También se puede usar `this` dentro de una clase anónima para hacer referencia a su instancia
- Las expresiones lambda trabajan con el **ámbito contenedor**
  - No se pueden ocultar las variables del ámbito contenedor dentro del cuerpo de la expresión lambda
  - `this` hace referencia a una instancia de la clase contenedora

En el ejemplo siguiente, ¿qué valor devuelve `scopeExperiment()`?

```
@FunctionalInterface
public interface ClaseFuncional{
  String method(String string);
}

private String variable = "Valor de la contenedora";

public String scopeExperiment() {

  ClaseFuncional unaInnerClass = new ClaseFuncional() {
    String variable = "Valor de la inner class";
    @Override
    public String method(String string) {
      return this.variable;
      /*  Con o sin this, no hay lambdas ni variables libres */
    }
  };
  String resultadoInnerClass = unaInnerClass.method("");

  return "resultadoInnerClass = " + resultadoInnerClass;
}
```

El valor devuelto será:

```
resultadoInnerClass  = Valor de la inner class
```

En el ejemplo siguiente, ¿qué valor devuelve `scopeExperiment()`?

```
@FunctionalInterface
public interface ClaseFuncional{
  String method(String string);
}

private String variable = "Valor de la contenedora";

public String scopeExperiment() {

  ClaseFuncional unaLambda = parametro -> {
    String variable= "Valor de la lambda";
    return this.variable;
    /* Con this, la clausura de la variable libre se produce
       con el valor de ClaseFuncional::variable */
  };
  String resultadoLambda = unaLambda.method("");

  return "resultadoLambda = " + resultadoLambda;
}
```

El valor será:

```
resultadoLambda = Valor de la contenedora
```

Funciones anónimas en Ruby

Bloques (*blocks*)

Sintaxis `do` ... `end`

```ruby
some_list = [ 10, 20, 30 ]
some_list.map do |i|
    i += 1
end
```

Sintaxis `{` ... `}`

```ruby
some_list = [ 10, 20, 30 ]
some_list.map { |i| i += 1 }
```

El método `map` itera y aplica un bloque repetitivamente a cada elemento de una colección (representado por el parámetro `i`)

Ejemplo: búsqueda en una lista

Sin bloques:

```
class SongList
  def with_title(title)
    for i in 0...@songs.length
      return @songs[i] if title == @songs[i].name
    end
    return nil
  end
end
```

Con bloques `do` ... `end`:

```
class SongList
  def with_title(title)
    @songs.find do |song|
      title == song.name
    end
  end
end
```

Con bloques `{` ... `}`:

```
class SongList
  def with_title(title)
    @songs.find { |song| title == song.name }
  end
end
```

El método `find` itera y aplica el test del bloque a cada elemento `song` de la colección.

Ejecución de bloques

- El bloque debe aparecer al lado de una llamada a método
- No se ejecuta el bloque, sino que se recuerda el contexto (variables locales, objeto actual, etc.) en que aparece
- Al bloque se le pueden pasar parámetros

Ejemplo: fibonacci

```
def fib_up_to(max)
  i1, i2 = 1, 1
  while i1 <= max
    yield i1
    i1, i2 = i2, i1+i2
  end
end
fib_up_to(200) {|f| print f, " " }

#Salida => 1 1 2 3 5 8 13 21 34 55 89 144
```

Ejecución con `yield`:

- Cuando se ejecuta el método, el bloque es invocado donde aparezca `yield`
- El control vuelve al método después del `yield`

Ejemplos de `yield`:

```ruby
def three_times
  yield
  yield
  yield
end
three_times { puts "Hello" }
```

```ruby
class Array
  def find
    for i in 0...size
      value = self[i]
      return value if yield(value)
    end
    return nil
  end
end
```

Ejemplos: iterar con bloques

- Iterar sobre un array con `each`

  `Array#each`: recibe un array y aplica el bloque a cada item, sin modificar el array ni crear un nuevo objeto; devuelve el mismo array.

```
[ 1, 3, 5, 7, 9 ].each {|i| printf i, " " }
#Salida => 1 3 5 7 9
Array a = [ 1, 2, 3, 4 ]
a.each {|i| puts i*2 }
#Salida => 2 4 6 8
#Devuelve => [1, 2, 3, 4]
a
#Devuelve => [1, 2, 3, 4]
```

- Iterar sobre un fichero con `each`

  `File#each`: recibe el contenido de un fichero de texto y aplica el bloque a cada línea.

```
f = File.open("testfile")
f.each do |line|
  puts line
end
f.close
f = File.open("testfile")
f.each {|line| puts line}
f.close
```

- Iterar sobre un array con `collect`

  `Array#collect`: aplica el bloque a todos los items y devuelve el nuevo array modificado; hace lo mismo que `Array#map`

```
["H", "A", "L"].collect {|x| x.succ }
# Salida => [''I'', ''B'', ''M'']
Array a = [ 1, 2, 3, 4 ]
a.collect {|i| puts i*2}
#Salida => 2 4 6 8
#Devuelve => [nil, nil, nil, nil]
a.collect {|i| i.succ}
#Devuelve => [2, 3, 4, 5]
a
#Devuelve => [1, 2, 3, 4]
```

Procs y lambdas

- En Ruby, una función anónima o *lambda* es simplemente un tipo especial de objeto `Proc`
- Definición de procs/lambdas:

```
# sin argumentos:
say_something = -> { puts "This is a lambda" }
say_something = lambda { puts "This is a lambda" }
say_otherwise = Proc.new { puts "This is a proc" }
# con argumentos:
times_two = ->(x) { x * 2 }
```

- Varias formas de llamar a la lambda (es preferible `call`)

```
say_something = -> { puts "This is a lambda" }
say_something.call
say_something.()
say_something[]

say_otherwise = Proc.new { puts "This is a proc" }
say_otherwise.call

times_two = ->(x) { x * 2 }
times_two.call(10)
```

- Los `proc` no se preocupan de los argumentos:

```
t = Proc.new { |x,y| puts "I don't care about args!" }
t.call #Salida: I don't care about args!
t.call(10) #Salida: I don't care about args!
t.call(10,10) #Salida: I don't care about args!
t.call(10,10) #Salida: I don't care about args!

s = ->(x,y) { puts "I care about args" }
s.call # ArgumentError: wrong number of arguments (given 0, expected 2)
s.call(10) # ArgumentError: wrong number of arguments (given 1, expected 2)
s.call(10,10) # Salida: I care about args
```

- Los `proc` retornan del método actual; los lambda retornan de la función anónima:

```
# funciona:
my_lambda = -> { return 1 }
puts "Lambda result: #{my_lambda.call}"

# eleva una excepción:
my_proc = Proc.new { return 1 }
puts "Proc result: #{my_proc.call}"
```

- Si el `proc` está dentro de un método, la llamada a `return` es equivalente a retornar de ese método:

```ruby
def call_proc
  puts "Before proc"
  my_proc = Proc.new { return 2 }
  my_proc.call
  puts "After proc"
end
puts call_proc
# Prints "Before proc" but not "After proc"

def call_lambda
  puts "Before lambda"
  my_lambda = lambda { return 2 }
  my_lambda.call
  puts "After lambda"
end
puts call_lambda
# Prints "Before lambda" and "After lambda"
```

Diferencias entre `Proc` y `lambda`:

- Las lambdas se definen con `-> {}` y los procs con `Proc.new {}`
- Los `Proc` retornan del método actual, las lambdas retornan de la propia función lambda
- Los `Proc` no se preocupan del número correcto de argumentos, las lambdas elevan una excepción

Paso de bloques como parámetros

- Simplemente, se añade al final de la llamada a un método
- ¿Dónde se llama al bloque? Donde el método indique con `yield`
- El bloque (realmente un objeto `Proc`) se pasa como una especie de parámetro no declarado

Ejemplos de paso de bloques:

- Llamada a un bloque sin parámetros

```ruby
def run_it
  puts("Before the yield")
  yield
  puts("After the yield")
end
```

```ruby
run_it do
  puts('Hello')
  puts('Coming to you from inside the block')
end

# Salida =>
#  Before the yield
#  Hello
#  Coming to you from inside the block
#  After the yield
```

- Cualquier método puede recibir un bloque como parámetro implícito, pero no lo ejecuta si no hace `yield`:

```ruby
def run_it
end

run_it do
  puts('Hello')
end

# => No genera salida
```

- Con `yield`:

```ruby
def run_it
  yield if block_given?
end

run_it do
  puts('Hello')
end
# Salida =>
#   Hello
```

- Llamada a un bloque con parámetros:

```ruby
def run_it_with_parameter
  puts('Before the yield')
  yield(24)
  puts('After the yield')
end

run_it_with_parameter do |x|
  puts('Hello from inside the proc')
  puts("The value of x is #{x}")
end

# Salida =>
#  Before the yield
#  Hello from inside the proc
#  The value of x is 24
#  After the yield
```

- Hacer explícito el bloque pasado como parámetro usando *ampersand*: explicitamos que se espera que el método reciba un parámetro de tipo bloque

```
def run_it_with_parameter(&block)
  puts('Before the call')
  block.call(24)
  puts('After the call')
end
```

- Convertir un `Proc` o un lambda en un bloque pasado como parámetro:

```
my_proc = Proc.new {|x| puts("The value of x is #{x}")}
run_it_with_parameter(&my_proc)
my_lambda = lambda {|x| puts("The value of x is #{x}")}
run_it_with_parameter(&my_lambda)

# Salida (en ambos casos) =>
#  Before the call
#  The value of x is 24
#  After the call
```

Lecturas recomendadas

- M. Williams: Java SE 8: Lambda Quick Start, Oracle Learning Library, 2013.
- D. Thomas & A. Hunt: Programming Ruby. The Pragmatic Programmer's Guide, Addison-Wesley, 2005.

STREAMS

Un stream representa una secuencia de elementos que soportan diferentes tipos de operaciones para realizar cálculos sobre ellos

Operaciones

Las operaciones sobre un stream pueden ser intermediarias o terminales

- Las operaciones **intermediarias** devuelven un nuevo stream permitiendo encadenar múltiples operaciones intermediarias sin usar punto y coma
- Las operaciones **terminales** son nulas o devuelven un resultado de un tipo diferente, normalmente un valor agregado a partir de cómputos anteriores

Ejemplo v0.1

[Probar en paiza.io]

```
void main() {
  List<String> myList =
    Arrays.asList("a1", "a2", "b1", "c2", "c1");

  myList
    .stream()
    .filter(s -> s.startsWith("c"))
    .map(String::toUpperCase)
    .sorted()
    .forEach(System.out::println);
}
```

Streams con interfaces funcionales

- Las operaciones que se aplican sobre un *stream* aceptan algún parámetro en forma de:
  - **interfaz funcional**: objeto cuyo tipo (clase) representa a una función ejecutable con un cierto número de parámetros (normalmente 0, 1 o 2)
  - **expresión lambda**: interfaz funcional anónima, que especifica el comportamiento de la operación, pero sin especificar formalmente su nombre y tipo de parámetros
- Las operaciones aplicadas no pueden modificar el *estado* del stream original

En el ejemplo anterior, se puede observar que:

- `filter`, `map` y `sorted` son operaciones intermediarias
- `forEach` es una operación terminal
- Ninguna de las operaciones modifica el estado de `myList` añadiendo o eliminando elementos
- Sólo se filtran ciertos elementos, se transforman a mayúsculas, se ordenan (por defecto, alfabéticamente) y se imprimen por pantalla

Ejemplo v0.2

```
void main() {
  List<String> myList =
    Arrays.asList("a1", "a2", "b1", "c2", "c1");

  myList
    .stream()
    .filter(s -> s.startsWith("c"))
    .map(String::toUpperCase)
    .sorted()
    .forEach(System.out::println);

  myList
    .stream()
    .reduce( (a,b) -> a + " " + b )
    .ifPresent(System.out::println);
}
```

Ejemplo práctico: partidos de una competición con lambdas

Reutilizamos el caso de filtrado de partidos que vimos en el apartado de **Predicados**, pero implementando `fixture()` como una factoría con una expresión lambda, en lugar de una clase concreta:

```
import java.util.List;
import java.util.function.Predicate;

private Predicate<Match> fixture(Team local, Team visiting) {
    return match -> match.getLocalTeam().equals(local)
                    && match.getVisitingTeam().equals(visiting);
}
```

La lambda captura los parámetros `local` y `visiting` y devuelve un `Predicate<Match>` funcional. No hace falta clase auxiliar; la lógica de filtrado está integrada en la expresión lambda.

Ahora usamos la factoría en un stream para validar que en una competición no hay partidos repetidos ni un equipo enfrentado a sí mismo:

```
private void checkMatchesInGroup(List<Match> matchesInGroup) {
  for (Match match : matchesInGroup) {
    Team t1 = match.getLocalTeam();
    Team t2 = match.getVisitingTeam();
    assertNotSame(t1, t2); // no juega contra sí mismo

    List<Match> firstLeg = matchesInGroup.stream()
        .filter(fixture(t1, t2))
        .toList();
    assertTrue(firstLeg.size() == 1);

    List<Match> secondLeg = matchesInGroup.stream()
        .filter(fixture(t2, t1))
        .toList();
    assertTrue(secondLeg.size() == 0);
  }
}
```

Más información

- Winterbe: Java 8 stream tutorial
- Oracle: Procesamiento de datos con streams de Java
- Oracle: Introducción a Expresiones Lambda y API Stream en Java