Streams

Antonio Espín Herranz

Streams

- Antes de Java 1.8. El cálculo en colecciones, lo realizamos:

```
List<Integer> numeros = Arrays.asList(1,2,3,4,5);
int suma = 0, cuadrado;
for (int n : numeros){
    if (n % 2 == 1){
        cuadrado = n * n;
        suma += cuadrado;
    }
}
System.out.println(suma);
```

Streams

- Con streams:

```
List<Integer> numeros = Arrays.asList(1,2,3,4,5);
int suma = numeros.stream()
.filter(n->n%2==1)
.map(n->n*n)
.reduce(0, Integer::sum);
```

Streams

- Dentro de programación funcional un stream se define como una lista infinita.
- Los stream no tiene almacenamiento.
  - En colección los elementos deben estar antes en memoria, en un stream no.
- El diseño de los stream se basa en iteraciones internas:
  - No necesitamos bucles para procesarlas.
  - Las operaciones que aplicamos realizan las iteraciones de forma interna.
- Soportan programación funcional.
- Están diseñados para ser procesados en paralelo sin un trabajo adicional de los desarrolladores.
- NO se pueden reutilizar como una colección.
- Pueden ser ordenados y desordenados.
- Soportan operaciones retardadas (lazy).

Streams

- Si queremos un stream en paralelo, hay que indicarlo:

```
int sum = Arrays.asList(1,2,3,4,5).parallelStream()
.filter(n -> n % 2 == 1)
.map(n -> n * n)
.reduce(0, Integer::sum);
```

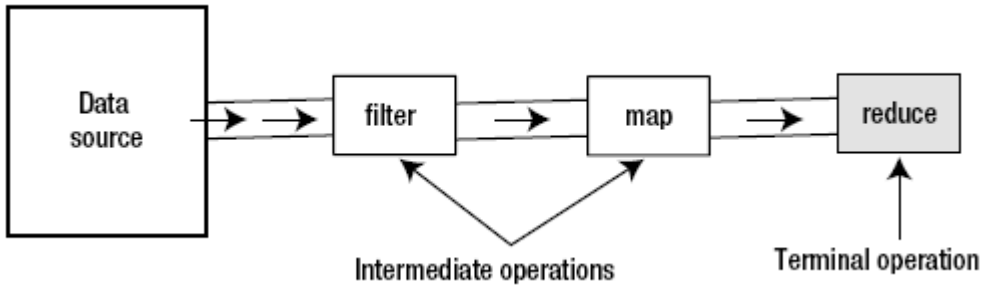
Streams

- Proporcionan un iterator() para iterar de forma externa pero no es necesario:

```
// Get a list of integers from 1 to 5
List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
...
// Get an iterator from the stream
Iterator<Integer> iterator = numbers.stream().iterator();
while(iterator.hasNext()) {
int n = iterator.next();
...
}
```

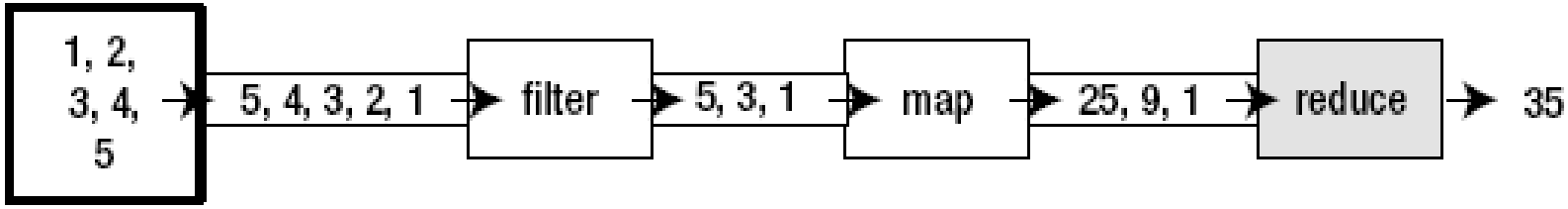
Operaciones

- Una stream soporta dos tipos de operaciones:
  - Operaciones intermedias (operaciones Lazy)
  - Operaciones finales (eager).
  - Conectan las operaciones a modo de pipe, de tal forma que los datos salen de una fuente de datos y pasando por cada operación.



Operaciones

- Las operaciones intermedias devuelven un Stream que procesa cada elemento.
- Las operaciones finales normalmente se asocian una operación que devuelve un sólo resultado.

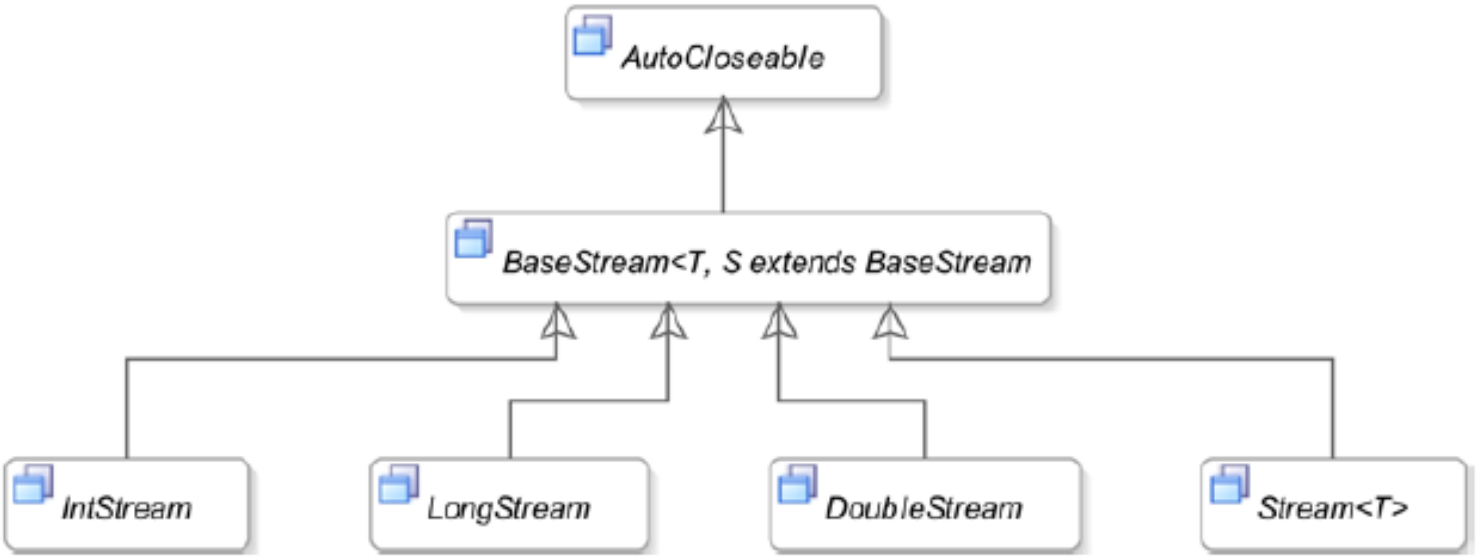



```
numbers.stream( ).filter(n -> n % 2 == 1).map(n -> n * n).reduce(0, Integer::sum)
```

Streams

- Un stream puede ser ordenada o desordenada.
- Pueden preservar el orden de los elementos si partimos de una colección ordenada.
- Si intentamos reutilizar una stream nos lanzará la exception: **IllegalStateException**
- Los Stream se encuentran en el paquete:
  java.util.stream

Arquitectura



Métodos

- Iterator<T> iterator()
  - Devuelve un iterador
- S sequential():
  - Retorna un stream secuencial.
- S parallel()
  - Retorna un stream paralelo.
- boolean isParallel()
  - Es o no un stream paralelo.
- S unordered()
  - Retorna una versión desordenada del stream, o el mismo ya lo está.

- Tenemos Stream de tipos primitivos (**IntegerStream**, **LongStream** y **DoubleStream**) y de tipos definidos por el programador: **Stream<T>** representa otro tipo, Stream<Empleado>, etc.

Creación de Stream

- Podemos crear Stream de **valores**: **of**

```
// Creates a stream with one string elements
Stream<String> stream = Stream.of("Hello");

// Creates a stream with four strings
Stream<String> stream = Stream.of("Ken", "Jeff", "Chris",
    "Ellen");

int sum = Stream.of(1, 2, 3, 4, 5)
      .filter(n -> n % 2 == 1)
      .map(n -> n * n)
      .reduce(0, Integer::sum);
```

Creación de Stream

- Se pueden crear de arrays:
  - String[] names = {"Ken", "Jeff", "Chris", "Ellen"};
  - Stream<String> stream = Stream.of(names);

  - String str = "Ken,Jeff,Chris,Ellen";
  - Stream<String> stream = Stream.of(str.split(","));

  - También dispone de un **Builder**:
  - Stream.Builder<String> builder = Stream.builder();

Builder

```
Stream<String> stream = Stream.<String>builder()
    .add("Ken")
    .add("Jeff")
    .add("Chris")
    .add("Ellen")
    .build();
```

Creación de Streams

- Métodos static que generan un stream con un **intervalo** de números enteros ordenados:
  - IntStream **range**(int start, int end)
    - Intervalo abierto.
  - IntStream **rangeClosed**(int start, int end).
    - Intervalo cerrado.
  - También disponible en LongStream.

Empty Stream

- // Creates an empty stream of strings
- Stream<String> stream = Stream.empty();

- IntStream, LongStream, y DoubleStream contienen también el método static empty() para crear un stream vacío de tipos primitivos.

- // Creates an empty stream of integers
- IntStream numbers = IntStream.empty();

Stream de funciones

- Podemos tener un stream que puede generar un número infinito de valores bajo demanda:
  - Métodos **static**

  <T> Stream<T> **iterate**(T seed, UnaryOperator<T> f)

  <T> Stream<T> **generate**(Supplier<T> s)
- Tenemos que indicar una semilla y una función.
- También están disponibles dentro de IntStream, LongStream y DoubleStream.

Stream.iterate()

// Números naturales:

- Stream<Long> naturalNumbers = Stream.iterate(1L, n -> n + 1);

//Números impares:

- Stream<Long> oddNaturalNumbers = Stream.iterate(1L, n -> n + 2);

Se puede especificar el tamaño con **.limit(n)**

Stream.iterate()

- Se puede aplicar la operación **forEach**

```
Stream.iterate(1L, n -> n + 2)
.limit(5)
.forEach(System.out::println);
```

- Se puede aplicar skip para saltar una serie de valores:

```
Stream.iterate(2L, PrimeUtil::next)
.skip(100)
.limit(5)
.forEach(System.out::println);
```

Stream.generate()

- Se indica un método que genere una serie de valores desordenados de forma infinita.

```
Stream.generate(Math::random)
.limit(5)
.forEach(System.out::println);
```

- Java 8, ha añadido métodos a Random que trabajan con Streams.

```
new Random().ints()
.limit(5)
.forEach(System.out::println);
```

Stream.generate()

- Stream.**generate**(new Random()::nextInt)
  - .limit(5)
  - .forEach(System.out::println);

Streams de Arrays

- Crear un stream con 3 números:
  - IntStream numbers = Arrays.stream(new int[]{1, 2, 3});
- Crear un stream con 3 String:
  - Stream<String> names = Arrays.stream(new String[] {"Ken", "Jeff"});

Streams de Colecciones

- import java.util.HashSet;
- import java.util.Set;
- import java.util.stream.Stream;
- ...
- // Create and populate a set of strings
- Set<String> names = new HashSet<>();
- names.add("Ken");
- names.add("jeff");
- // Create a sequential stream from the set
- **Stream<String>** sequentialStream = **names.stream**();

- // Create a parallel stream from the set
- **Stream<String>** parallelStream = **names.parallelStream**();

Stream de ficheros

- Java 8 ha añadido métodos para trabajar con operaciones I/O que utilizan streams.
- En los paquetes: java.nio.file.*

- Se puede leer un fichero de texto como un stream de String, donde cada elemento representa una línea de texto del fichero.

- También se puede obtener un stream de JarEntry de un JarFile.

- También entradas de directorios a partir de un path.

Ejemplo de fichero

```
Path path = Paths.get(filePath);
if (!Files.exists(path)) {
    System.out.println("The file " + path.toAbsolutePath() + " does
        not exist.");
    return;
}
try (Stream<String> lines = Files.lines(path)) {
    // Read and print all lines
    lines.forEach(System.out::println);
} catch (IOException e) {
    e.printStackTrace();
}
```

Ejemplo: listar una carpeta

```
Path dir = Paths.get("");
System.out.printf("%nThe file tree for %s%n",
    dir.toAbsolutePath());
try (Stream<Path> fileTree = Files.walk(dir)) {
    fileTree.forEach(System.out::println);
} catch (IOException e) {
    e.printStackTrace();
}
```

Otras fuentes

```
String str = "5 apples and 25 oranges";
str.chars()
    .filter(n -> !Character.isDigit((char)n) &&
       !Character.isWhitespace((char)n))
    .forEach(n -> System.out.print((char)n));
```

```
String str = "Ken,Jeff,Lee";
Pattern.compile(",")
    .splitAsStream(str)
    .forEach(System.out::println);
```

Operaciones de los Streams

- Operaciones **Intermedias**:
  - Se aplican a un Stream y devuelven un Stream.
    - **distinct**: sólo elementos distintos.
    - **filter**: los elementos que cumplen la condición
    - **limit**: Número de iteraciones.
    - **map**: Aplicar una función a los elementos del stream
    - **peek**: Para depurar, devuelve los elementos del stream.
    - **skip**: Saltar un número de iteraciones. Es como for (int i = 100 ..
    - **sorted**: Devuelve los elementos ordenados.

Operaciones de los Streams

- Operaciones **Terminales**:
  - Devuelven un único valor.
    - **allMatch**: Devuelve true si el Stream es vacío, o si todos los elementos cumplen un predicado. False en caso contrario.
    - **anyMatch**: Si algún elemento cumple el predicado. False si está vacío o no lo cumple.
    - **findAny**: Devuelve cualquier elemento del Stream, un objeto opcional vacío si el Stream está vacío.
    - **findFirst**: El primer elemento del Stream.
    - **noneMatch**: Devuelve true si ningún elemento coincide con el predicado.
    - **forEach**: Se aplica una acción para cada elemento del Stream.
    - **reduce**: Aplica una operación de reducción para llegar a un único valor, por ejemplo: sumar todos los elementos del Stream.

Debug

- Se puede depurar los elementos que va tomando el Stream para ello disponemos del método peek().

```
int sum = Stream.of(1, 2, 3, 4, 5)
    .peek(e -> System.out.println("Taking integer: " + e))
    .filter(n -> n % 2 == 1)
    .peek(e -> System.out.println("Filtered integer: " + e))
    .map(n -> n * n)
    .peek(e -> System.out.println("Mapped integer: " + e))
    .reduce(0, Integer::sum);
System.out.println("Sum = " + sum);
```

Ejemplos

- Utilizar Optional para evitar **NullPointerException**.

```
Optional<Integer> max = Stream.of(1, 2, 3, 4, 5)
.reduce(Integer::max);
if (max.isPresent()) {
    System.out.println("max = " + max.get());
}
else {
    System.out.println("max is not defined.");
}
```

Ejemplos

```
OptionalDouble income = Person.persons()
.stream()
.mapToDouble(Person::getIncome)
.max();

if (income.isPresent()) {
    System.out.println("Highest income: " + income.getAsDouble());
}
else {
    System.out.println("Could not get the highest income.");
}
```

Ejemplos

```
long personCount = Person.persons()
    .stream()
    .count();
System.out.println("Person count: " +
    personCount);
```

Collectors

- Partiendo de un Stream obtener colecciones.

- Por ejemplo, si tenemos una colección de objetos, podemos filtrar y después si queremos obtener una nueva colección podemos utilizar Collectors.

Ejemplo

```
List<String> names = Person.persons()
    .stream().map(Person::getName)
    .collect(ArrayList::new, ArrayList::add,
    ArrayList::addAll);

// Mas sencillo:
List<String> names = Person.persons().stream()
.map(Person::getName)
.collect(Collectors.toList());
// También soporta: toSet()
// toCollection(TreeSet::new)
```

Contar

- long count = Person.persons()
- .stream()
- .count();
- System.out.println("Persons count: " + count);

Recuperar Estadísticas Resumidas

- DoubleSummaryStatistics
- LongSummaryStatistics
- IntSummaryStatistics

```
public class SummaryStats {
        public static void main(String[] args) {
        DoubleSummaryStatistics stats = new DoubleSummaryStatistics();
        stats.accept(100.0);
        stats.accept(500.0);
        stats.accept(400.0);

        // Get stats
        long count = stats.getCount();
        double sum = stats.getSum();
        double min = stats.getMin();
        double avg = stats.getAverage();
        double max = stats.getMax();
        System.out.printf("count=%d, sum=%.2f, min=%.2f, average=%.2f, max=%.2f%n",
        count, sum, min, max, avg);
```

Ejemplo, sobre un Stream de objetos

```
DoubleSummaryStatistics incomeStats =
Person.persons()
.stream()
.map(Person::getIncome)
.collect(DoubleSummaryStatistics::new,
DoubleSummaryStatistics::accept,
DoubleSummaryStatistics::combine);
System.out.println(incomeStats);
```

- Salida:

```
DoubleSummaryStatistics{count=6, sum=26000.000000,
   min=0.000000, average=4333.333333, max=8700.000000}
```

Otra forma

```
DoubleSummaryStatistics incomeStats =
Person.persons()
.stream()
.collect(Collectors.summarizingDouble(Person::g
   etIncome));
```

Volcar datos a un Mapa

- Map<Long,String> idToNameMap = Person.persons()
- .stream()
- .collect(Collectors.toMap(Person::getId, Person::getName));

Agrupando datos

- Map<Person.Gender, List<Person>> personsByGender =
- Person.persons()
- .stream()
- .collect(Collectors.**groupingBy**(Person::getGender));

- System.out.println(personsByGender);

- {**FEMALE**=[(3, Donna, FEMALE, 1962-07-29, 8700.00), (5, Laynie, FEMALE, 2012-12-13, 0.00)],

- **MALE**=[(1, Ken, MALE, 1970-05-04, 6000.00), (2, Jeff, MALE, 1970-07-15, 7100.00), (4, Chris, MALE,
- 1993-12-16, 1800.00), (6, Li, MALE, 2001-05-09, 2400.00)]}

Parallel Streams

- Los Streams pueden secuenciales o paralelos.
- Las operaciones secuenciales se ejecutan con un solo hilo.
- La operaciones en paralelo se ejecutan en más de hilo.
- Se selecciona un tipo u otro llamando al método adecuado.
- Por debajo utiliza fork para el cálculo en paralelo.

Ejemplo: juntar en una cadena los nombres de las personas

- **Secuencial**:

```
String names = Person.persons() // The data source
.parallelStream() // Produces a parallel stream
.filter(Person::isMale) // Processed in parallel
.map(Person::getName) // Processed in parallel
.collect(Collectors.joining(", ")); // Processed in parallel
```

- **Paralelo**:

```
String names = Person.persons() // The data source
.stream() // Produces a sequential stream
.filter(Person::isMale) // Processed in serial
.parallel() // Produces a parallel stream
.map(Person::getName) // Processed in parallel
.collect(Collectors.joining(", ")); // Processed in parallel
```

Pruebas de tiempo

- **// Tiempo de ejecución 758 sg.**

```
long count = IntStream.rangeClosed(2,
    Integer.MAX_VALUE/10)
.filter(PrimeUtil::isPrime).count();
```

- **// Tiempo de ejecución 181 sg.**

```
long count = IntStream.rangeClosed(2,
    Integer.MAX_VALUE/10)
.parallel().filter(PrimeUtil::isPrime).count();
```