Streams

Prof. Michele Loreti

Programmazione Avanzata

Corso di Laurea in Informatica (L31)

Scuola di Scienze e Tecnologie

Iterating collections...

Operations on data stored in a collection are typically based on an iteration.

Iterating collections...

Operations on data stored in a collection are typically based on an iteration.

Example Suppose that we want to count all the long words in a book.

Iterating collections...

Operations on data stored in a collection are typically based on an iteration.

Example Suppose that we want to count all the long words in a book.

Solution This operation can be performed in three steps:

1. read all the data from a file;
2. store words in a list;
3. iterate over elements in the list and count the elements that are longer than 12 chars.

Iterating collections...

```
//Read file into string
String contents = new String(
  Files.readAllBytes(
    Paths.get("alice.txt"),
    StandardCharsets.UTF_8
  )
);

//Split into words, nonletters are delimiters.
List<String> words = List.of(contents.split("\\PL+"));

//Iterate and count
long count = 0;
for (String w: words) {
  if (w.length() > 12) {
    count++;
  }
}
```

Streams. . .

Streams provide a view of data that lets you specify computations at a higher conceptual level than with collections.

Streams...

Streams provide a view of data that lets you specify computations at a higher conceptual level than with collections.

With the stream we specify what we want to do leaving the scheduling to the underling implementation.

Streams...

Streams provide a view of data that lets you specify computations at a higher conceptual level than with collections.

With the stream we specify what we want to do leaving the scheduling to the underling implementation.

Example To count the long words we can just write:

```
long count = words
        .parallelStream()
        .filter(w -> w.length() > 12)
        .count();
```

Streams...

Streams provide a view of data that lets you specify computations at a higher conceptual level than with collections.

With the stream we specify what we want to do leaving the scheduling to the underling implementation.

Example To count the long words we can just write:

```
long count = words
        .parallelStream()
        .filter(w -> w.length() > 12)
        .count();
```

Streams follow the "what, not how" principle!

Streams vs Collections...

Streams vs Collections...

1. A stream does not store its elements. They may be stored in an underlying collection of generated on demand.

Streams vs Collections...

1. A stream does not store its elements. They may be stored in an underlying collection of generated on demand.

2. Stream operations do not change their source. For instance, the filter method does not remove elements from a stream, but it yields a new stream in which they are not present.

Streams vs Collections...

1. A stream does not store its elements. They may be stored in an underlying collection of generated on demand.

2. Stream operations do not change their source. For instance, the filter method does not remove elements from a stream, but it yields a new stream in which they are not present.

3. Stream operations are lazy when possible. This means that they are not executed until their result is needed. **We can have infinite streams!**

Workflow...

Typical workflow when we work with streams is:

Workflow...

Typical workflow when we work with streams is:

1. Create a stream.

Workflow...

Typical workflow when we work with streams is:

1. Create a stream.
2. Specify immediate operations for transforming the initial stream into others (possibly in multiple steps).

Workflow...

Typical workflow when we work with streams is:

1. Create a stream.
2. Specify immediate operations for transforming the initial stream into others (possibly in multiple steps).
3. Apply a terminal operation to produce a result. This operation forces the execution of the lazy operations that precede it.

Workflow...

Typical workflow when we work with streams is:

1. Create a stream.
2. Specify immediate operations for transforming the initial stream into others (possibly in multiple steps).
3. Apply a terminal operation to produce a result. This operation forces the execution of the lazy operations that precede it.
4. The stream cannot be longer used.

Stream creation...

Interface Collection<T> provides method:

```
Stream<E> stream()
```

that returns a sequential Stream with this collection as its source.

Stream creation...

Interface `Collection<T>` provides method:

```
Stream<E> stream()
```

that returns a sequential `Stream` with this collection as its source.

A stream can be built form an array by using the utility method:

```
Stream<T> Stream.of( T ... values )
```

Stream creation...

Infinite streams can be built by using the (static) utility methods provided class Stream:

```
Stream<T> generate(Supplier<? extends T> s)

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

Stream<T> Stream.iterate(T seed,
  Predicate<? super T> hasNext,
  UnaryOperator<T> next)
```

Stream transformations...

Methods of class Stream<T>

A stream transformation produces a stream whose elements are derived from those of another stream.

Stream transformations...

Methods of class Stream<T>

A stream transformation produces a stream whose elements are derived from those of another stream.

Method filter can be used to select only some of the elements in a stream:

```
Stream<T> filter(Predicate<? super T> predicate)
```

Stream transformations...

Methods of class Stream<T>

A stream transformation produces a stream whose elements are derived from those of another stream.

Method filter can be used to select only some of the elements in a stream:

```
Stream<T> filter(Predicate<? super T> predicate)
```

Method map transforms a stream by applying a function to each element in the stream:

```
Stream<S> map(Function<? super T,? extends R> mapper)
```

Extracting substreams...

Methods of class Stream<T>

Given a stream we can extract a substream.

Extracting substreams...

Methods of class Stream<T>

Given a stream we can extract a substream.

Method `limit` can be used to select only the first `n` elements of a stream:

```
Stream<T> limit( long n )
```

Extracting substreams...

Methods of class Stream<T>

Given a stream we can extract a substream.

Method limit can be used to select only the first n elements of a stream:

```
Stream<T> limit( long n )
```

Method skip can be used to ignore the first n elements of a stream:

```
Stream<T> skip( long n )
```

Extracting substreams...

Methods of class Stream<T>

Given a stream we can extract a substream.

Method limit can be used to select only the first n elements of a stream:

```
Stream<T> limit( long n )
```

Method skip can be used to ignore the first n elements of a stream:

```
Stream<T> skip( long n )
```

Methods takeWhile and dropWhile selects (resp. discharge) all the elements of a stream while a given predicate is satisfied:

```
Stream<T> takeWhile(Predicate<? super T> predicate)
```

```
Stream<T> dropWhile(Predicate<? super T> predicate)
```

Combining streams...

Methods of class Stream<T>

Static method concat can be used to build a new string resulting from the concatenation of two streams:

```
static <T> Stream<T> concat(Stream<? extends T> a,
                            Stream<? extends T> b)
```

Other Stream transformations...

Methods of class Stream<T>

Remove duplicates from a stream:

```
Stream<T> distinct()
```

Other Stream transformations...

Methods of class Stream<T>

Remove duplicates from a stream:

```
Stream<T> distinct()
```

Sort elements in a stream:

```
Stream<T> sorted() //T implements Comarable<T>
```

```
Stream<T> sorted(Comparator<? super T> comparator)
```

Other Stream transformations...

Methods of class Stream<T>

Remove duplicates from a stream:

```
Stream<T> distinct()
```

Sort elements in a stream:

```
Stream<T> sorted() //T implements Comarable<T>
```

```
Stream<T> sorted(Comparator<? super T> comparator)
```

Build a stream consisting of the elements of a stream, additionally performing the provided action on each element as elements are consumed from the resulting stream:

```
Stream<T> peek(Consumer<? super T> action)
```

Optional values...

An Optional<T> object is a wrapper for either an object of type T or no object.

Optional values...

An Optional<T> object is a wrapper for either an object of type T or no object.

The key to using optinal is to use a method that either produces an alternative if the value is not present, or consumes the value only if it is present.

Optional values...

An Optional<T> object is a wrapper for either an object of type T or no object.

The key to using optinal is to use a method that either produces an alternative if the value is not present, or consumes the value only if it is present.

Optional<T> methods:

```
T orElse(T other)
T orElseGet(Supplier<? extends T> supplier)
T orElseThrow(Supplier<? extends X> exceptionSupplier)
void ifPresent(Consumer<? super T> action)
void ifPresentOrElse(Consumer<? super T> action,
    Runnable emptyAction)
```

Reductions...

Methods of class Stream<T>

Reductions are terminal operations that reduce a stream to a nonstream value that can be used in our program.

Reductions...

Methods of class Stream<T>

Reductions are terminal operations that reduce a stream to a nonstream value that can be used in our program.

- Get the maximum element of this stream according to the provided Comparator:

```
Optional<T> max(Comparator<? super T> comparator)
```

- Get the minimum element of this stream according to the provided Comparator

```
Optional<T> min(Comparator<? super T> comparator)
```

- Get the first element of this stream:

```
Optional<T> findFirst()
```

- Get some element of the stream:

```
Optional<T> findAny()
```

Collecting results...

Methods of class Stream<T>

Class Stream<T> provides many methods that can be used to use data in a stream:

```
void forEach(Consumer<? super T> action)
```

```
<A> A[] toArray(IntFunction<A[]> generator)
```

```
<R,A> R collect(Collector<? super T, A, R> collector)
```

Collecting results...

Methods of class Stream<T>

Class Stream<T> provides many methods that can be used to use data in a stream:

```
void forEach(Consumer<? super T> action)
```

```
<A> A[] toArray(IntFunction<A[]> generator)
```

```
<R,A> R collect(Collector<? super T, A, R> collector)
```

Interface Collector<T,A,R> represents a mutable reduction operation that accumulates input elements into a mutable result container.

Collecting results...

Methods of class Stream<T>

Class Stream<T> provides many methods that can be used to use data in a stream:

```
void forEach(Consumer<? super T> action)
```

```
<A> A[] toArray(IntFunction<A[]> generator)
```

```
<R,A> R collect(Collector<? super T, A, R> collector)
```

Interface Collector<T,A,R> represents a mutable reduction operation that accumulates input elements into a mutable result container.

Standard collector are provided via utility methods in class Collectors:

- Collectors.toList()
- Collectors.toSet()
- Collectors.joining()
- ...

Reduction Operations

Method reduce provides a general mechanism for computing a value from a stream.

Reduction Operations

Method reduce provides a general mechanism for computing a value from a stream.

```
Optional<T> reduce(BinaryOperator<T> accumulator)

T reduce(T identity, BinaryOperator<T> accumulator)

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

Reduction Operations

Method reduce provides a general mechanism for computing a value from a stream.

```
Optional<T> reduce(BinaryOperator<T> accumulator)

T reduce(T identity, BinaryOperator<T> accumulator)

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

Example:

```
List<Integer> values = ....
Optional<Integer> sum = values.stream().reduce((x,y)->x+y);
```

Parallel Streams

Streams make it easy to parallelise bulk operations. To take advantage of these mechanisms we have to pay attention to few rules.

Parallel Streams

Streams make it easy to parallelise bulk operations. To take advantage of these mechanisms we have to pay attention to few rules.

We can build a parallel stream by using method `paralleStream()` in interface `Collection`.

Parallel Streams

Streams make it easy to parallelise bulk operations. To take advantage of these mechanisms we have to pay attention to few rules.

We can build a parallel stream by using method `paralleStream()` in interface `Collection`.

Any sequential stream can be made parallel via the method `parallel()`.

Parallel Streams

Streams make it easy to parallelise bulk operations. To take advantage of these mechanisms we have to pay attention to few rules.

We can build a parallel stream by using method `paralleStream()` in interface `Collection`.

Any sequential stream can be made parallel via the method `parallel()`.

In a parallel stream, when the terminal method executes, all intermediate stream operations are parallelised.

Parallel Streams

Streams make it easy to parallelise bulk operations. To take advantage of these mechanisms we have to pay attention to few rules.

We can build a parallel stream by using method `paralleStream()` in interface `Collection`.

Any sequential stream can be made parallel via the method `parallel()`.

In a parallel stream, when the terminal method executes, all intermediate stream operations are parallelised.

To guarantee that the obtained result is the same as in sequential settings, all the operations must be stateless!

Parallel Streams

Bad example:

```
int[] shortWords = new int[12];
words.parallelStream().forEach(
  s -> { if (s.length()<12) shortWords[s.length()]++ }
);
```

Parallel Streams

Bad example:

```
int[] shortWords = new int[12];
words.parallelStream().forEach(
  s -> { if (s.length()<12) shortWords[s.length()]++ }
);
```

There is a race condition on shortWords!

Parallel Streams

Bad example:

```
int[] shortWords = new int[12];
words.parallelStream().forEach(
  s -> { if (s.length()<12) shortWords[s.length()]++ }
);
```

There is a race condition on shortWords!

Correct code:

```
Map<Integer,Long> shortWordCounts
  = words.parallelStream()
    .filter( s -> s.length()<12 )
    .collect(groupingBy(
      String::length,
      counting()))
    );
```

To be continued...