Prefer an external PDF viewer to the one integrated in firefox.

The learning outcome of this practical session is to practice Java Streams operations with simple collectors and to acquire a new development skill that consists in writing code with a REPL (Read-Evaluate-Print Loop, jshell, a debugger-like tool).

The first step consists in using Java 16 or higher to benefit from jshell, which allows interactive development. This form of development is especially relevant when prototyping. Code can be copy-pasted and multi-lines instructions can be achieved with Alt-<Enter>.

REPL are particularly suited for prototyping, learning and demonstrations. In particular, the benefits of jshell are:

- faster development due to increased interaction (feedback is obtained with a single keystroke, “enter”), which allows incremental prototyping;
- ability to query intermediate results (similar to debug mode with direct access to variable content);
- removing the need for boilerplate (several classes are already imported);
- automatic completion;
- command history (with arrows);
- many special commands (/open, /vars, /help, etc.).

Similarly to an IDE, automatic completion is available with method signature documentation and moving around previous code can be done easily with up and down arrows.

However, partial redefinition of already declared elements can lead to specific bugs. Moreover, some niceties offered by IDE are not available (automatic declaration of methods to override, syntax highlighting, even though jshell is supported in IntelliJ IDEA).

To facilitate the development further, make sure all interesting methods are available directly with the following two static imports.

```java
import static java.util.Comparator.comparing;
import static java.util.stream.Collectors.toList;
```

Finally, open your navigator on the official pages documenting the methods used in this practical session.

First, you need the version number of your installed Java environment: `java -version`.

Then, you will need the documentation of interface Stream, the documentation of class Comparator, and possibly the documentation of class Collectors.
1 Traders and Transactions Classes

```java
public class Trader {
    private final String name;
    private final String city;
    public Trader(String n, String c) {
        this.name = n;
        this.city = c;
    }
    public String getName() {
        return this.name;
    }
    public String getCity() {
        return this.city;
    }
    public String toString() {
        return "Trader: " + this.name + " in " + this.city;
    }
}

public class Transaction {
    private final Trader trader;
    private final int year;
    private final int value;
    public Transaction(Trader trader, int year, int value) {
        this.trader = trader;
        this.year = year;
        this.value = value;
    }
    public Trader getTrader() {
        return this.trader;
    }
    public int getYear() {
        return this.year;
    }
    public int getValue() {
        return this.value;
    }
    public String toString() {
        return "{" + this.trader + ", " + "year: " + this.year + ", " + "value: " + this.value + "}";
    }
}
```

2 Data Set

```java
Trader raoul = new Trader("Raoul", "Cambridge");
Trader mario = new Trader("Mario", "Milan");
Trader alan = new Trader("Alan", "Cambridge");
Trader brian = new Trader("Brian", "Cambridge");
List<Transaction> transactions = Arrays.asList(
    new Transaction(brian, 2011, 400),
    new Transaction(brian, 2012, 300),
);```
new Transaction(raoul, 2012, 1000),
new Transaction(raoul, 2011, 400),
new Transaction(mario, 2012, 710),
new Transaction(mario, 2012, 700),
new Transaction(alan, 2012, 950)
);

3 Requests

The first 6 requests are essential.

Using only the variable *transactions*:

1. Finds all transactions in 2011 and sort by value (small to high).

   ```java
   List<Transaction> tr2011 =
   transactions.stream()
   .filter(transaction -> transaction.getYear() == 2011)
   .sorted(comparing(Transaction::getValue))
   .toList();
   ```

2. What are all the unique cities where the traders work?

   ```java
   List<String> cities =
   transactions.stream()
   .map(transaction -> transaction.getTrader().getCity())
   .distinct()
   .toList();
   ```

   An alternative with a set:

   ```java
   Set<String> cities =
   transactions.stream()
   .map(Trader::getCity)
   .collect(toSet());
   ```

3. Returns a string of all traders’ names sorted alphabetically.

   ```java
   String traderStr =
   transactions.stream()
   .map(transaction -> transaction.getTrader().getName())
   .distinct()
   .sorted()
   .reduce("", (n1, n2) -> n1 + n2);
   ```
An alternative with a specialized collector:

```java
import static java.util.stream.Collectors.joining;
String traderStr =
transactions.stream()
  .map(transaction -> transaction.getTrader().getName())
  .distinct()
  .sorted()
  .collect(joining());
```

4. Are any traders based in Milan?

```java
boolean milanBased =
transactions.stream()
  .anyMatch(transaction -> transaction.getTrader()
      .getCity()
      .equals("Milan"));
```

5. What’s the highest value of all the transactions?

```java
Optional<Integer> highestValue =
transactions.stream()
  .map(Transaction::getValue)
  .reduce(Integer::max);
```

```java
Optional<Integer> highestValue =
transactions.stream()
  .map(Transaction::getValue)
  .max(comparing(Integer::intValue));
```

```java
OptionalInt highestValue =
transactions.stream()
  .mapToInt(Transaction::getValue)
  .max();
```

6. Prints all unique pair of traders’ names as a list of strings such that the first trader has once made a transaction with lower value than a different second one.

```java
List<String> smallerTrans =
transactions.stream()
  .flatMap(t1 ->
    transactions.stream()
      .filter(t2 -> t1.getTrader() != t2.getTrader()));
```
7. Finds all traders from Cambridge and sort them by name.

```java
List<Trader> traders =
    transactions.stream()
    .map(Transaction::getTrader)
    .filter(trader -> trader.getCity().equals("Cambridge"))
    .distinct()
    .toList();
```

8. Finds all transactions’ values from the traders living in Cambridge.

```java
List<Integer> values =
    transactions.stream()
    .filter(t -> "Cambridge".equals(t.getTrader().getCity()))
    .map(Transaction::getValue)
    .toList();
```

9. Finds the transaction with the smallest value.

```java
Optional<Transaction> smallestTransaction =
    transactions.stream()
    .reduce(((t1, t2) ->
        t1.getValue() < t2.getValue() ? t1 : t2));
```

10. Find all traders that have increased their transactions values over time.
List<Trader> improvTraders =
    transactions.stream()
    .flatMap(
        t1 -> transactions.stream()
            .filter(t2 -> t1.getTrader() == t2.getTrader())
            .filter(t2 -> t1.getYear() < t2.getYear())
            .filter(t2 -> t1.getValue() < t2.getValue())
    )
    .map(Transaction::getTrader)
    .distinct()
    .toList();