1.
List<names> kNames = names.stream().filter(names ->
names.contains(“k”)).collect(Collectors.toList());

2.
Predicate:
filter()

Function:
map()

Consumer:
printl()

Producer():
stream()

3.
Der übergebene Lambda-Ausdruck darf Zustand haben und man ist selbst für die Thread-Sicherheit des Zustands verantwortlich.

4.
Bei parallelen Streams kann nicht garantiert werden, dass der Stream in der richtigen Reihenfolge zusammen geführt werden kann. Für jedes Element kann eine Action zu jeder Zeit und in jedem Thread, den die Library wählt, ausgeführt werden. Diese unklare Ausführung von Befehlen behindert das Sortieren.

5.
a)
Da die map Funktion parallel ausgeführt wird, kann das Ergebnis von Durchlauf zu Durchlauf variieren, da die Zuteilung der Threadzeit verschieden sein kann. Außerdem ist das Hashset nicht thread-safe was dazu führen kann, dass das gleiche Element zweimal hinzugefügt wird, da der eine Thread nicht auf den andren zugreifen kann.

b)
In diesem Fall macht das ConcurrentHashSet keinen Unterschied, da dies nur den Zugriff über den eigenen Iterator, Spliterator oder Stream synchronisiert. Da von den parallel streams der Collection auf das HashSet zugegriffen wird entsteht das gleiche Problem, wie bei a).

6.

Streams sind faul, weil dazwischenliegende Operationen nicht ausgewertet werden bis die Operation beendet wird.

7.

Die combiner Funktion kombiniert zwei Teilergebnisse um ein neues Teilergebnis zu erzeugen. Der combiner ist nötig um während parallel laufenden geteilte Variablen wieder zu einer Variable zusammen zu führen.

8.
Die unordered() Operation sorgt dafür dass der Stream nicht mehr zwingend die Reihenfolge der Befehle beachten muss. Dadurch können anschließende Operationen optimiert werden, da man die Reihenfolge nicht beachten muss.

9.

a)
Da .forEach(...) eine terminal operation ist wird nach dem ersten Ausführen der stream geschlossen und kann beim zweiten forEach() nicht mehr benutzt werden.

b)
Da .iterate(...) einen unendlichen sequenziert geordneten stream her gibt wird die forEach(...) methode bis in die Unendlichkeit ausgeführt (auch weil kein .limit() gesetzt wurde)..

c)
Da die .iterate(0, i -> (i + 1) % 2) Methode einen stream, welcher nur mit den Werten: 0,1,0,1,... gefüllt ist, wiedergibt und die distinct Methode nur verschiedene Werte zulässt, läuft der Iterator bis ins unendliche, obwohl ein Limit gesetzt wurde. Ähnlich wie bei a) kommt die .limit() Methode nicht auf ihre 10, da .distinct() hier nur max. 2 zulässt.

d)
In diesem Beispiel hier wird eine ConcurrentModificationException geworfen, da die peek(...) Methode eine Änderung am IntStream vornimmt und die forEach(...) merkt, dass etwas an dem IntStream verändert wurde, während sie selbst ausgeführt wird.