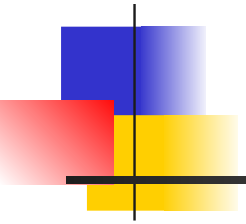
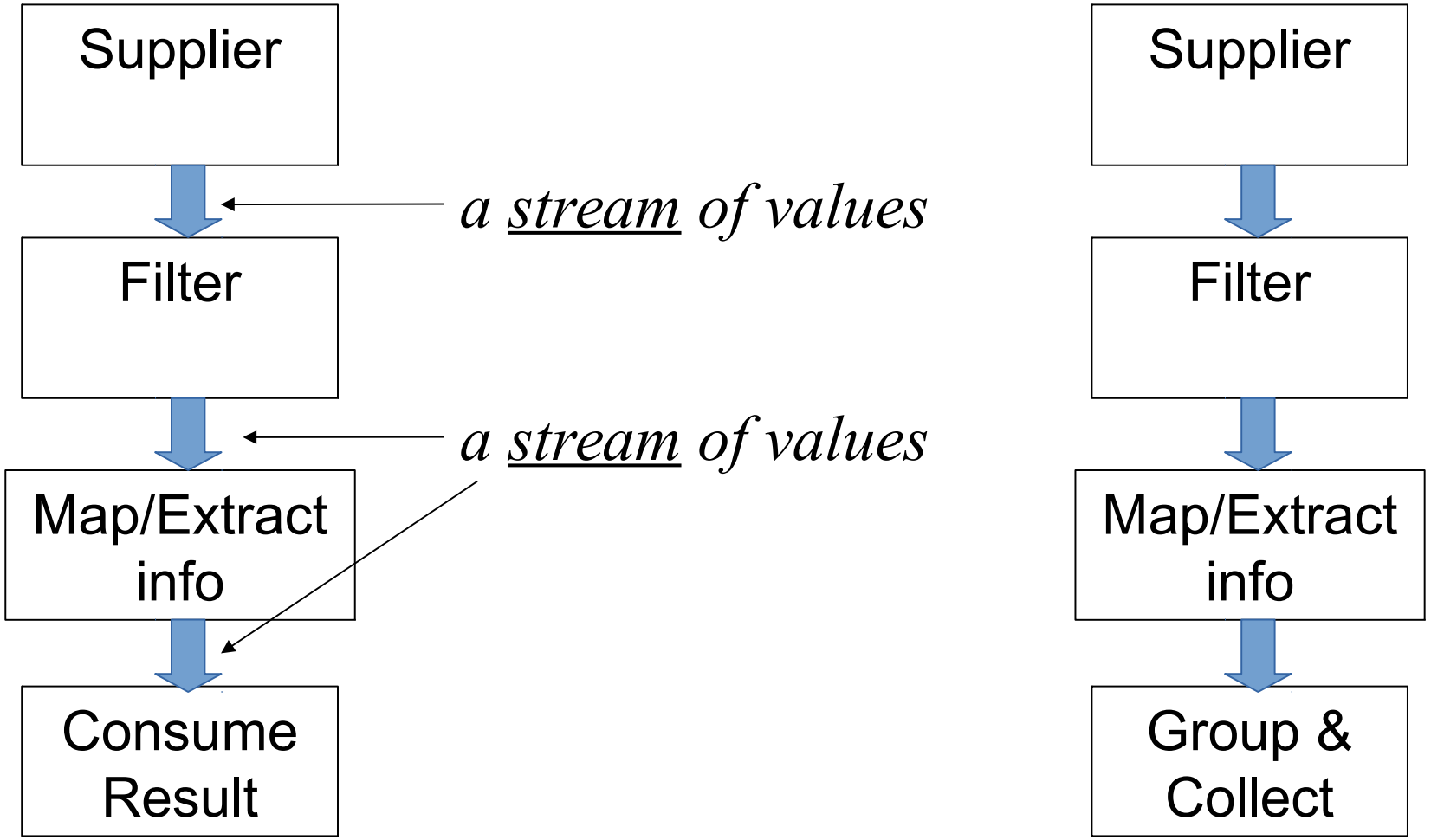


# Streams

# Conceptual view of stream processing

Two common patterns for working with collection of data:



# Linux example using pipes

Read all the lines from a file.

Remove comment lines beginning with #.

Sort the lines.

Eliminate duplicate lines.

Write to a new file.

```
$ cat somefile | grep -v '^#' | sort | uniq > outfile
```

Pipe connects output from one command to input of the next command.

# Java List Processing

- Suppose we have a list of fruit. Print all of them.

```
List<String> fruit = getFruits();
for(String name: fruit) {
    System.out.println( name );
}
```

Same thing using forEach and a Consumer:

```
List<T>:     void forEach( Consumer<T> )

Consumer<T>:    void accept(T arg)
```

# Java List Processing

- Using Loop:

```
List<String> fruit = getFruits();
for(String name: fruit) {
    System.out.println( name );
}
```

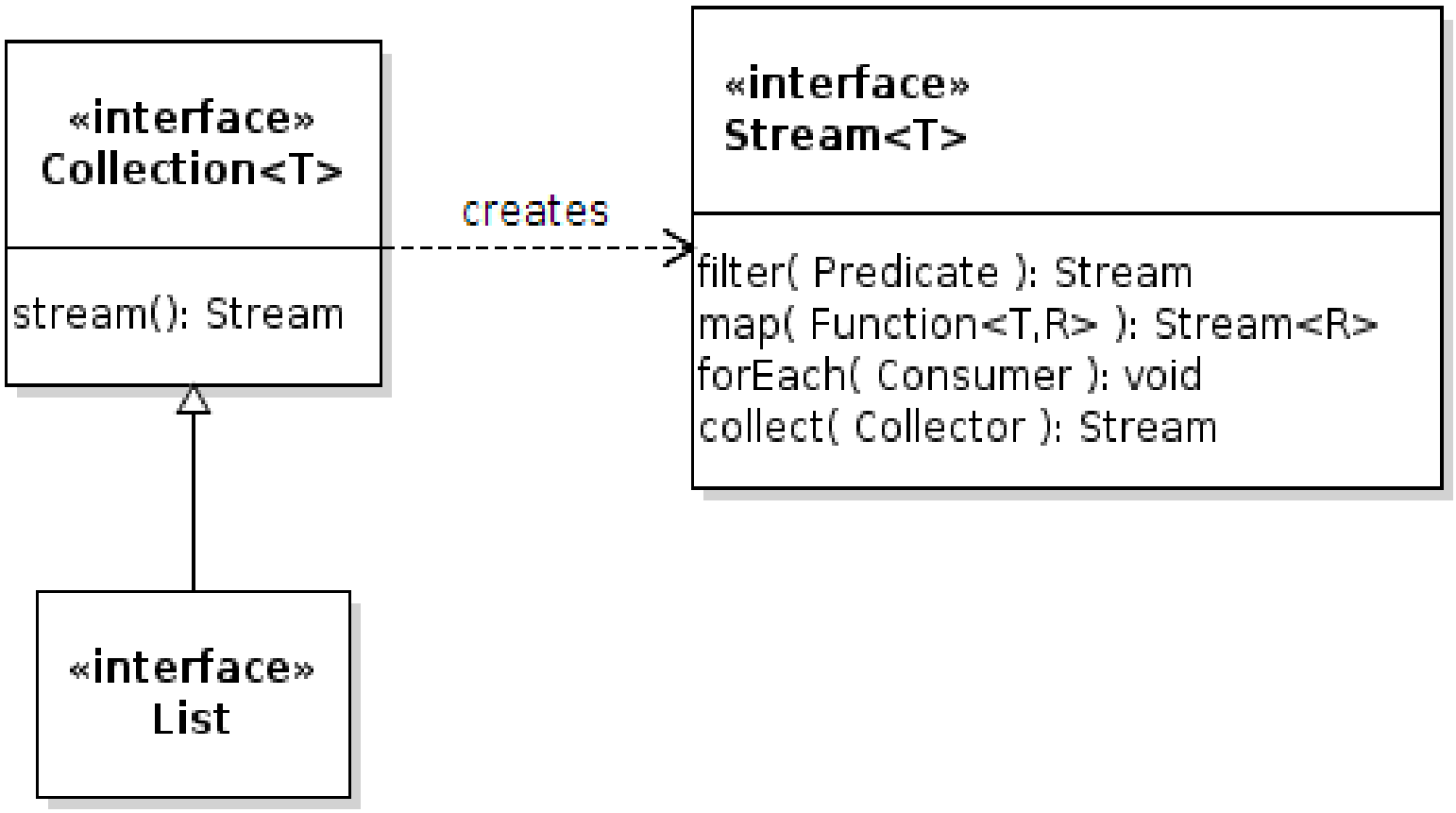
Using forEach:

```
List<String> fruit = getFruits();
fruit.stream().forEach(
     (x)->System.out.println(x) );
```

# What Happened?

*Collection* has 2 new methods for creating *Streams*.

*Stream* is an interface for stream processing.



# Stream methods

- Stream methods mostly return another Stream.
- Use build pipelines: list.stream().filter( p ).map( f ).
- **`forEach`** "consumes" the Stream so it returns nothing

```
filter( Predicate test ): Stream

map( Function<T,R> fcn ): Stream<R>

sorted( Comparator<T> ): Stream

limit( maxSize ): Stream

peek( Consumer ): Stream

collect( Collector<T,A,R> ): R

forEach( Consumer ): void
```

# Composing Streams

Since Stream methods return another Stream, we can "chain" them together. Like a pipeline.

Example: find all the fruit that end with "berry".

What we want:

```
fruit.stream()
     .filter( ends with "berry" ).forEach( print )
```

| Predicate |
| --- |
| test(arg: T): bool |

| Consumer |
| --- |
| accept(T): void |

# Writing and using lambda

- Write some Lambdas for the Predicate and Consumer

```
Predicate<String> filter =
                (s) -> s.matches(".*berry$");

Consumer<String> print =
                (s) -> System.out.println(s);

// or, using a Function Reference
Consumer<String> print =
                System.out::println;
```

# Sort the Fruit & remove duplicates

- Using a loop and old-style Java

```
List<String> fruit = getFruits();
Collections.sort( fruit );
String previous = "";
// can't modify list in a for-each loop
for(int k=0; k<fruit.size();  ) {
    compare this fruit with previous fruit
    if same then remove it.
    Be careful about the index (k)!
}
```

# Sort the Fruit & remove duplicates

- Using a stream

```
List<String> fruit = getFruits();
List<String> sorted =
      fruit.stream().sorted().distinct();
```

# Exercise: get all currencies

- Use a stream to return the names of all currencies in a list of valuable.

```
List<String> getCurrencies(List<Valuable> money) {
    // use:
    // stream()
    // map()
    // distinct()
    // sorted()
    // collect( Collectors.toList() )
```

```
List<Valuable> money = Arrays.asList(
     new Coin(5,"Baht"), new Banknote(10,"Rupee"),
     new Coin(1,"Baht"), new Banknote(50,"Dollar"));

List<String> currencies = getCurrencies(money);
// should be: { "Baht", "Dollar", "Rupee" }
```