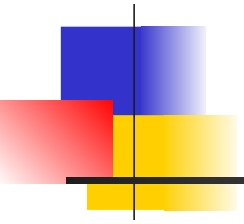
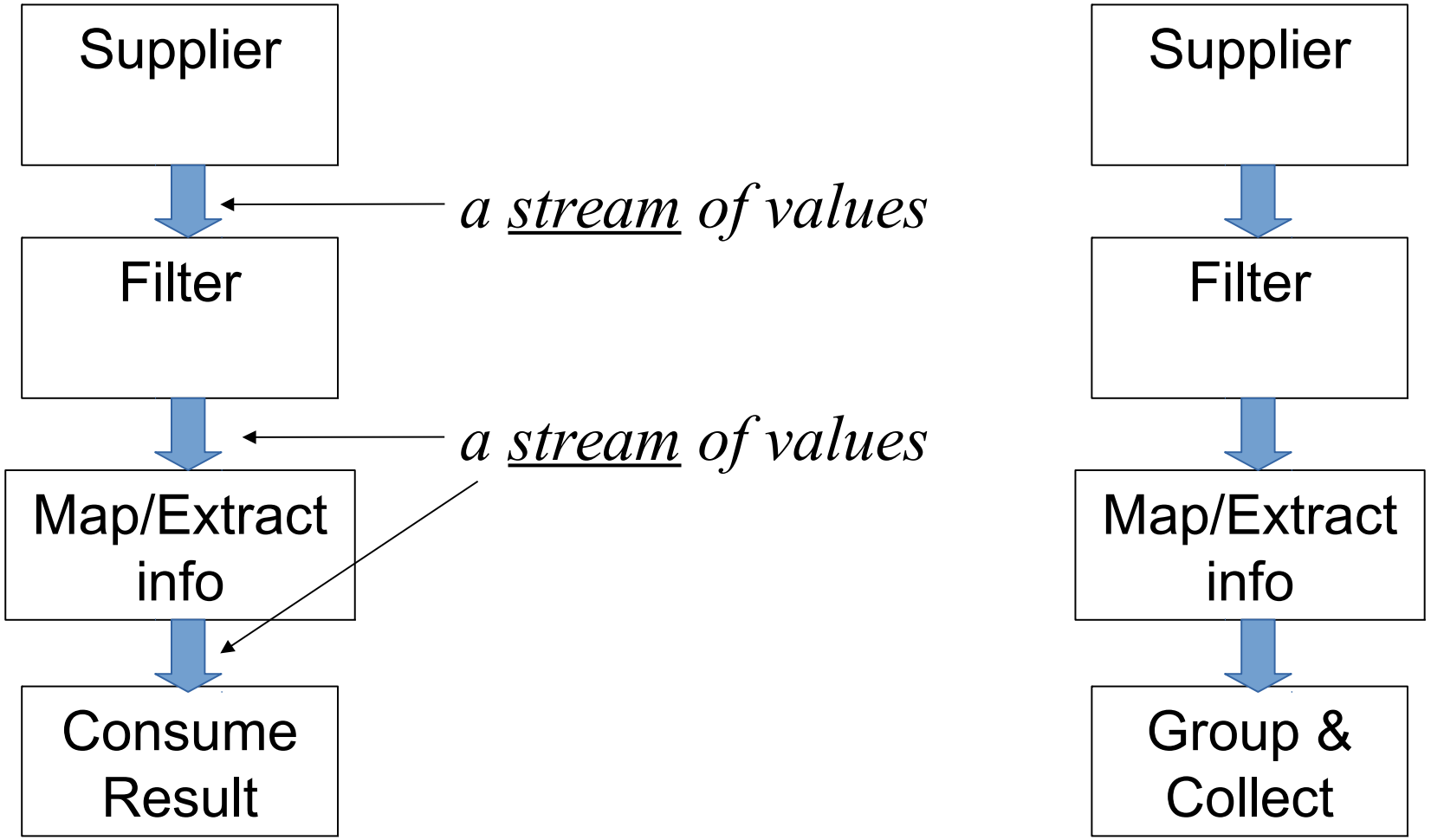


Streams

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:

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

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

Java List Processing

- Using a Loop:

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

Using forEach and Consumer:

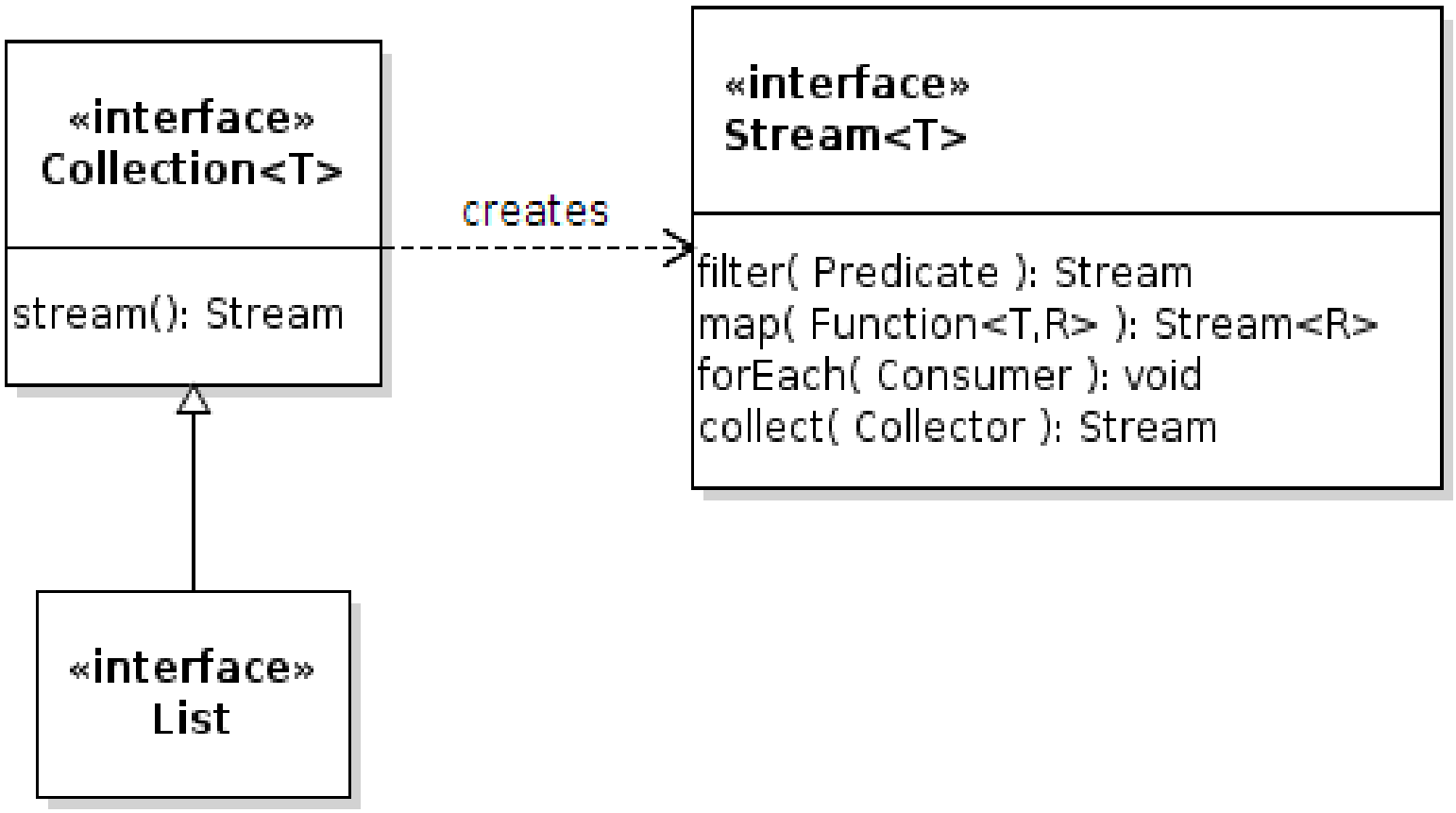
```
List<String> fruit = getFruits();
fruit.forEach( (x)->System.out.println(x) );
```

Consumer written as Lambda expression

What Happened?

Collection has 2 new methods for creating *Streams*.

Stream is an interface for stream processing.



Streams

- Every Java collection has a steam() method to create.
- You can use Stream to process elements

```
List<String> fruit = Arrays.asList(
      "Apple", "Banana", "orange", "pear");
fruit.stream().__________
```

Add operations on the stream to do what you want

Stream methods

- Stream methods mostly return another Stream.
- Use to 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 )
```

<<interface>>
Predicate
test(arg: T): bool

<<interface>>
Consumer
accept(T): void

Writing and using lambda

- Write some Lambdas for the Predicate and Consumer

```
Predicate<String> berries =
                (s) -> s.endsWith("berry");

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

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

Assemble Parts of the Stream

- Now connect the parts to a Stream "pipeline":

```
Predicate<String> berries =
                (s) -> s.endsWith("berry");

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

// Process the List of fruits:
fruit.stream( ).filter( berries )
                   .forEach( print );
```

Stream with Lamdbas Inline

- Don't have to declare type parameter (it is inferred).

```
// Stream with Lambdas defined where used
fruit.stream( )
      .filter( (s) -> s.endsWith("berry") )
      .forEach( System.out::println );
```

Creating a New Collection

Collect the stream result into a new collection (List)

by using: `stream.collect( Collector ).`

The **`Collectors`** class contains useful "collectors".

We want `Collectors.toList()`

```
List<String> result =
      fruit.stream()
           .filter(berries)
           .collect(  Collectors.toList() );
```

Collector

Sort the Fruit & remove duplicates

Using a loop and old-style Java is not so easy:

```
List<String> fruit = getFruits();
Collections.sort( fruit );
String previous = "";
// can't modify list in a for-each loop
// so use an indexed for 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> sortedFruit =
      fruit.stream().sorted().distinct()
            .collect( Collectors.toList() );
```

Exercise: get all currencies

- Use a stream to return the names of all currencies in a list of Valuable. Include each currency only once.

```
List<String> getCurrencies(List<Valuable> money) {
    // use:
    // stream()
    // map( Function<Valuable,String> )
    // 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" }
```

Exercise: try it

Try to solve it yourself before looking at the next slide.

Exercise: get all currencies

```
List<String> getCurrencies(List<Valuable> money) {
   List<String> result =
       money.stream( )
            .map( (m) -> m.getCurrency() )
            .distinct()   // remove duplicates
            .sorted()
            .collect( Collectors.toList() );
   return result;
```

```
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);
// result: { "Baht", "Dollar", "Rupee" }
```