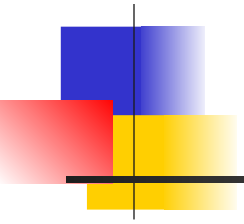
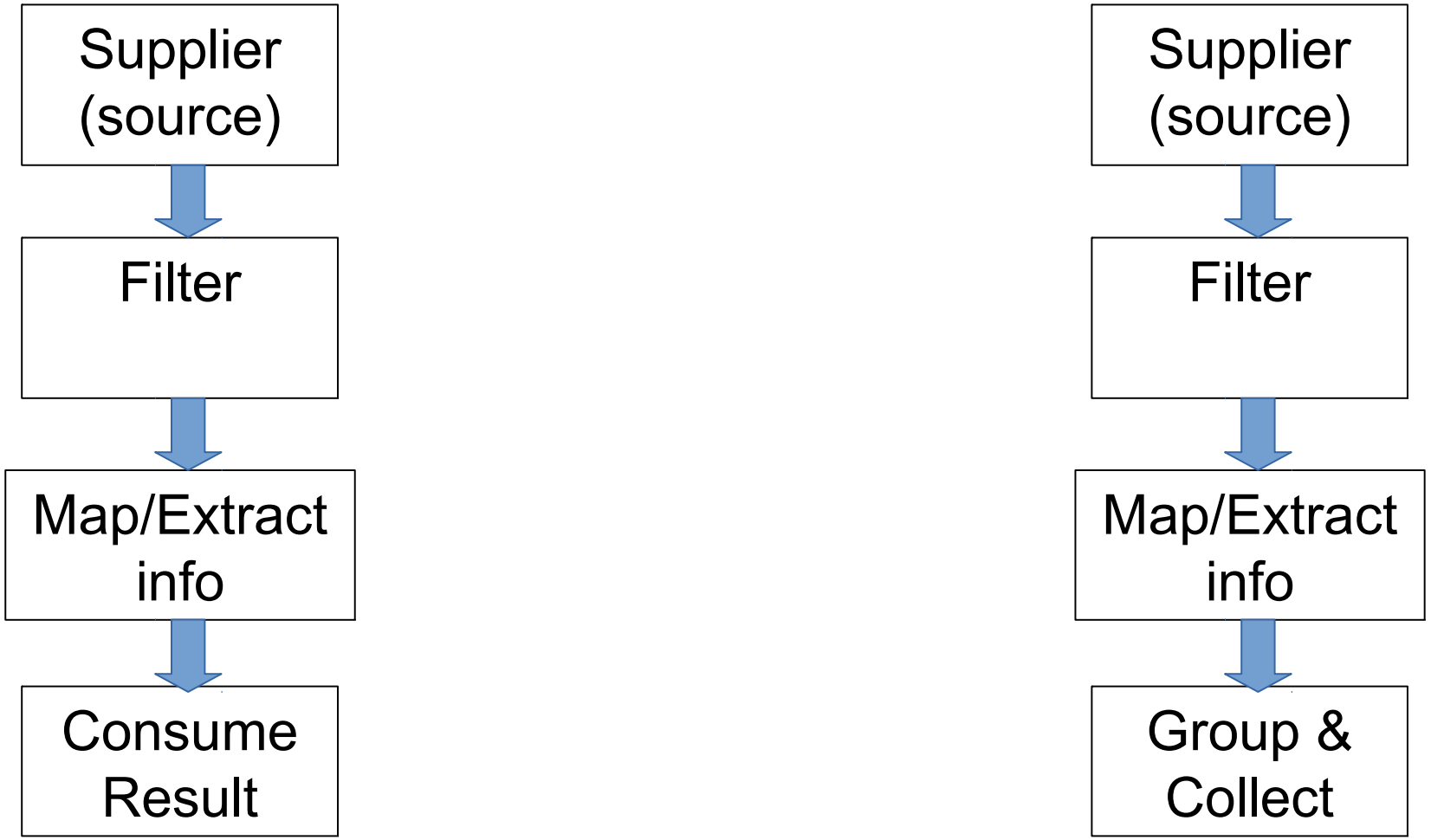


Streams

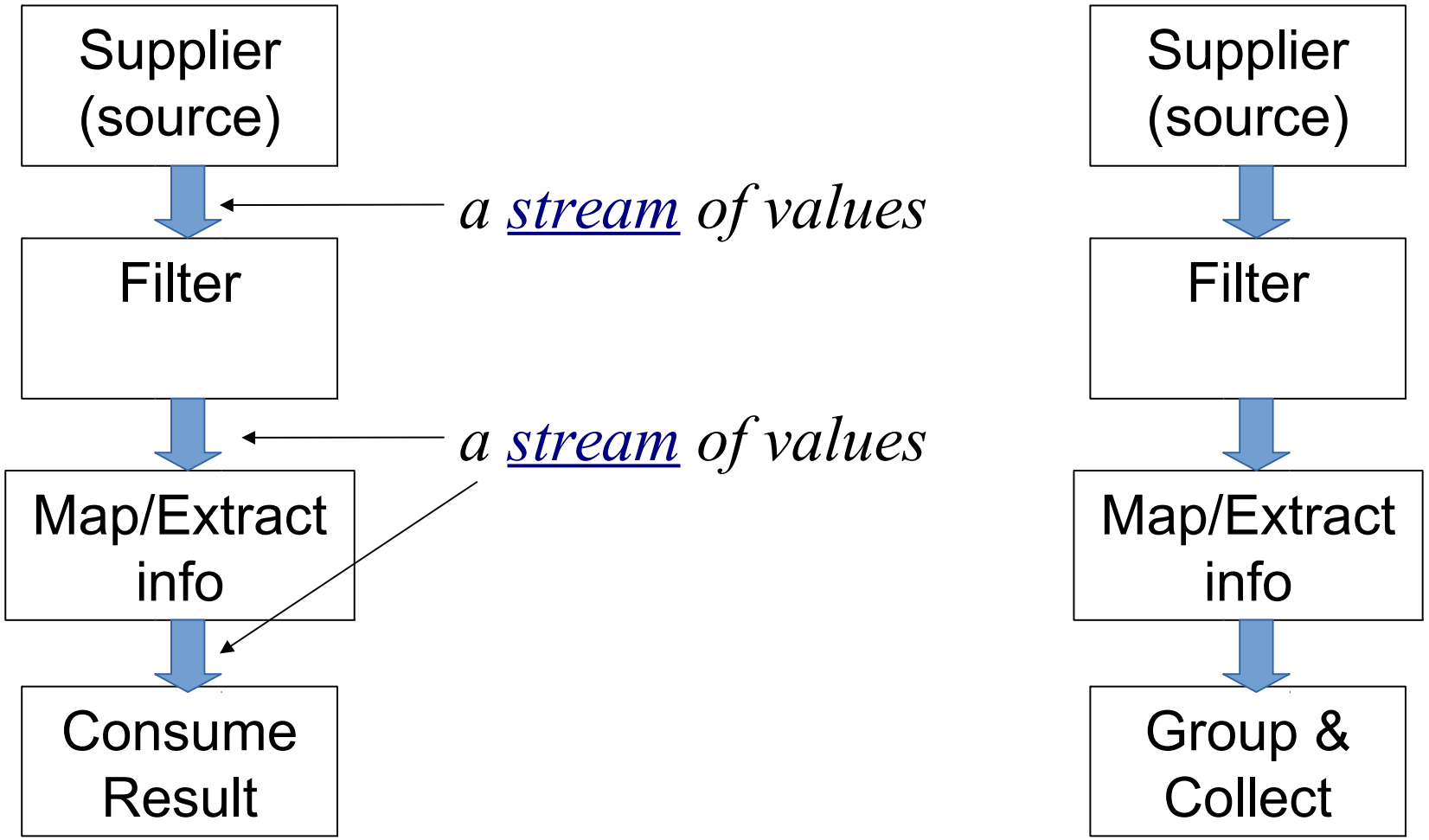
Conceptual view of stream processing

Two common patterns for working with collection of data:



A *Stream* of data

The values flow like a stream of data...



Linux example using pipes

cat	Output all the lines in a file.
grep -v	Remove lines beginning with #.
sort	Sort the lines.
uniq	Eliminate duplicate lines.
> file	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 Strings. Print all of them.

```
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 a Lambda expression

Explanation

- Every Collection and Iterable has a forEach( ) method.

```
// give each element to a "Consumer"
list.forEach( Consumer<String> consumer );
```

Consumer is an interface with 1 abstract method.

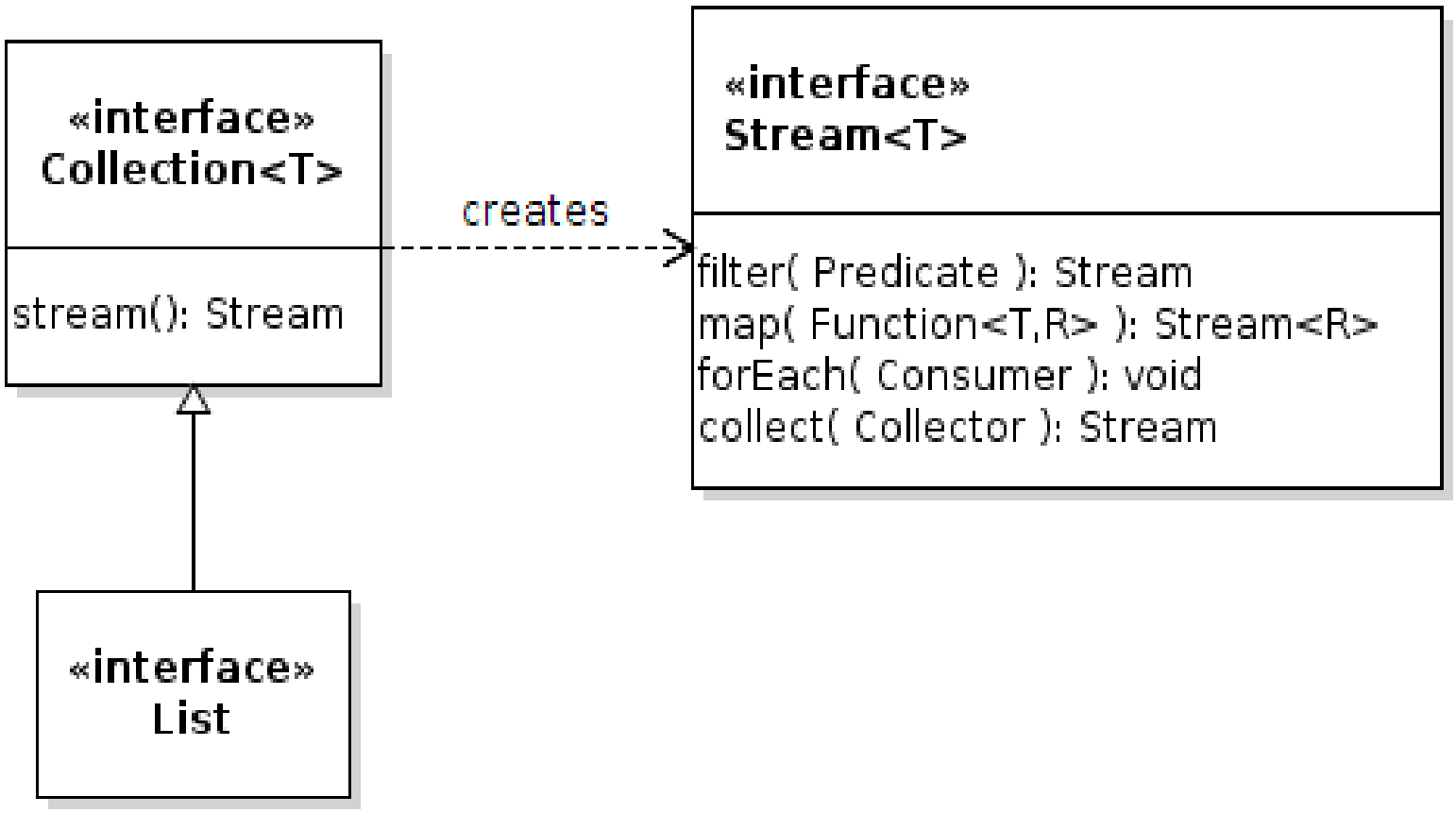
It "consumes" the argument (returns nothing).

```
interface Consumer<T> {
    public void accept(T arg);
}
```

Streams

Collection has 2 new methods for creating *Streams*.

Stream is an interface for stream processing.



Streams

Use a Stream to process elements

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

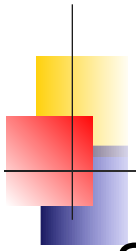

Add operations on the stream to do what you want

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 );
```

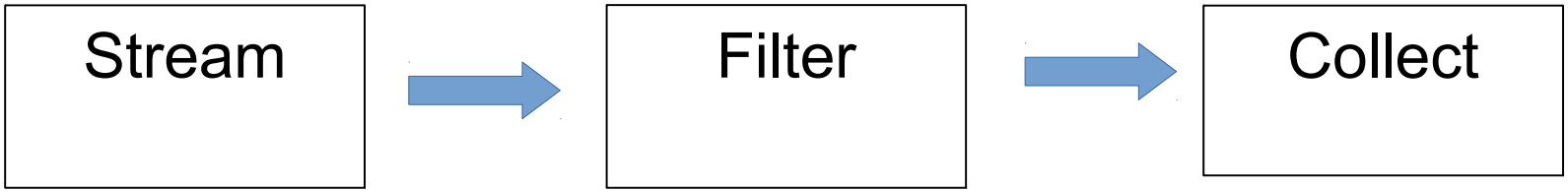
Stream with Inline Lambda

- 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 );
```

Stream with Collector

Instead of *consuming* the Stream, collect the elements.



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

- Remove duplicate items from the list

```
List<String> fruit = getFruits();
List<String> sortedFruit =
      fruit.stream()
           .sorted()
           .distinct()
           .collect( Collectors.toList() );
```

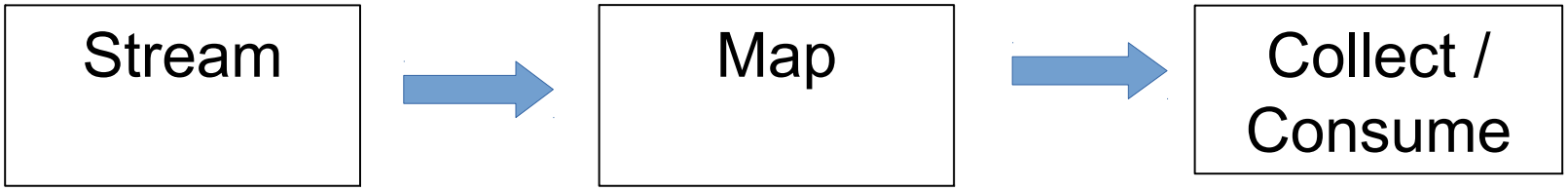
MoneyUtil.filterByCurrency

- Filter all the Valuable in the parameter (money) that have the requested currency:

```
static <E extends Valuable> List<E>
    filterByCurrency(List<E> money,String currency){


    return money.stream( )
                    .________________________________________
                    .________________________________________

}
```

Stream with Mapping

Map one kind of value to another kind of value.




```
stream.map( Function<T,R> mapper )
```

T = type of input values (the stream type)

R = type of output value (output stream type)

Get the student names

For every student in class, get the student's name and put the names into a new List

```
Function<Student,String> mapToName =
                s -> s.getName();
List<String> names =
    students.stream()
            .map( mapToName )
            .collect( Collectors.toList() );
```

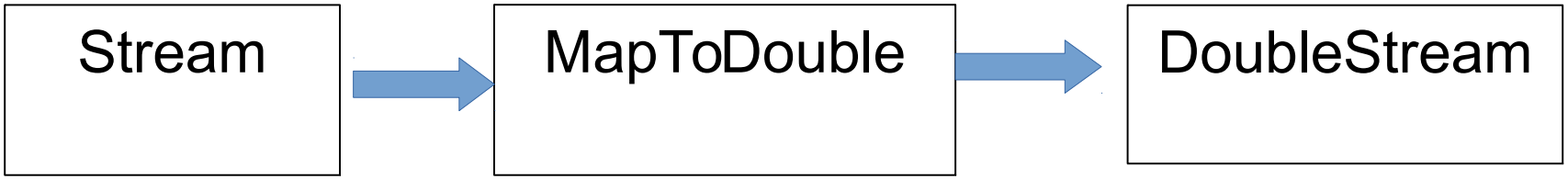
How to Sum Valuables?

Can we use a Stream to *sum* all the valuables in a List?

```
public double sumForCurrency(
            List<Valuable> money, String currency ) {
    double total = 0.0;
    for(Valuable v: money) {
        if ( v.getCurrency().equals(currency) )
            total += v.getValue();
    }
    return total;
}
```

Special Interfaces for Primitives

ToDoubleFuntion, DoubleStream, for "double" primitive




```
stream.mapToDouble( ToDoubleFunction<T> f )
```

A **`ToDoubleFunction<T>`** maps **`T`** to **`"double"`**.

```
ToDoubleFunction<Valuable> getValue =
        v -> v.getValue();  // returns v.getValue()
```

Sum All Valables

Can we use a Stream to *sum* all the valuables in a List?

```
public double sumByCurrency( List<Valuable> money )
{
    double total =
       money.stream()
            .mapToDouble( Valuable::getValue )
            .sum( ); // a method of DoubleStream
    return total;
}
```

Sum for One Currency

Can you sum money just for one currency, using a Stream?

```
public double sumForCurrency(
    List<Valuable> money, String currency ) {
    return money
            .stream()
            .____________________________________

            .____________________________________

            .____________________________________
}
```

How to Sum Valuables?

Can we use a Stream to *sum* all the valuables in a List?

```
public double sumByCurrency(
    List<Valuable> money, String currency ) {
    double total = 0.0;
    for(Valuable v: money) {
        if ( v.getCurrency().equals(currency) )
            total += v.getValue();
    }
    return total;
}
```

How to Sum Valuables?

Can we use a Stream to *sum* all the valuables in a List?

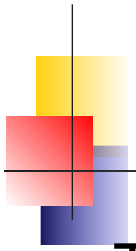
```
public double sumByCurrency(
    List<Valuable> money, String currency ) {
    double total = 0.0;
    for(Valuable v: money) {
        if ( v.getCurrency().equals(currency) )
            total += v.getValue();
    }
    return total;
}
```

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) {
    // .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" }
```