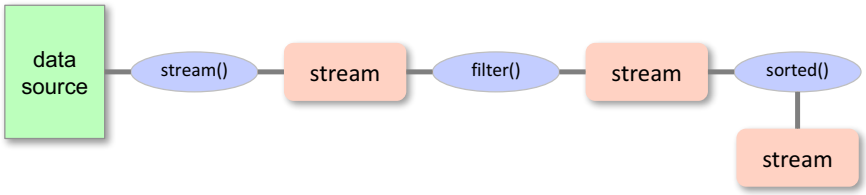
Streams in Java 8

GURUTEAM
specialist ICT learning

Java8 Streams

- Provide a stream processing model for Java
  - can work sequentially or in parallel
- Allow output of one stream to become input to another
  - stream methods can return next stream in chain
  - but one method does not need to complete
  - before output can be used by the next



About Streams

- Functional in nature
  - no storage
  - don't modify the source
- Can chain streams
  - can produce an intermediate stream
  - terminal operations produce output
- Aim to be Lazy / Possibly infinite
  - can be operated on using findFirst() or limit(n)
- Consumable
  - a new stream must be generated to revisit items
- Can execute in parallel

Streams on Collections

- Collections common structure in Java applications
  - streams extensions to collection classes
  - two key methods added to collections
- `stream()`
  - creates a stream object that operates on collection
  - allows for sequential process of contents, filtering, sorting etc.
- `parallelStream()`
  - creates a stream that works in parallel on data
  - collection must be treated as immutable
  - take a copy if required
- Operations are pipelined together and lazily evaluated

Example

- Generate Stream from simple array
- Print out all elements

```
…
import java.util.List;
…

  List<String> theList = Arrays.asList("one", "two", "three");
  …

  theList.stream()
         .forEach(System.out::println);
  …
```

```
one
two
three
```


Not only collections

- It is not necessary to have a collection to use streams
- Can use `Stream.of` method to create a stream
  - takes a group of object references

```
Stream.of("a1", "a2", "a3").forEach(System.out::println);
```

```
a1
a2
a3
```

- Special types of streams for primitive types
  - e.g. IntStream, LongStream and DoubleStream

```
IntStream.range(1, 4).forEach(System.out::println);
```

```
1
2
3
```

```
IntStream.generate(
    ()->{return (int)(Math.random()*100)
          .limit(10).forEach(System.out::println);
```

Generating Infinite Streams

- Stream need not have a finite length

```
IntStream.iterate (1, i -> i + 1  )
         .limit(5)    // or else we go on forever…
         .forEach ( System.out::println);
```

```
1
2
3
4
5
```

- `generate()` function allows arbitrary function to be called to provide next element
  - e.g. for random values

```
IntStream.generate( ()->{return (int)(Math.random()*100 )
          .limit(5)
          .forEach(System.out::println);
```

```
92
61
88
68
11
```


Streams Methods

- Can use functional operations on the stream
  - `map()` applies a function to each element in stream and returns a new stream
  - `filter()` filters contents of stream and returns a new stream
  - `flatMap()` apply a `map()` function and flatten the nested stream that results
  - `reduce()` combine elements in stream to a single value
  - `forEach()` applies operation to element element in stream. Does not return a stream
  - `collect()` ends the stream process and transforms stream data (e.g. into a `List`)
- Other supported methods
  - `count()` number of elements in stream
  - `limit()` only pass first `n` elements to the next stream
  - `sorted()` – sorts the items in the stream
  - `max()` / `min()` returns the maximum or minimum element in the stream
  - `distinct()` ensures all values in the stream are distinct

Example

- Print out the entry in the stream (list) that has the longest length

```
…
import java.util.List;
…

  List<String> theList = Arrays.asList("one", "two", "three");
  …

  theList.stream()
         .map ( s -> s.toUpperCase())
         .max ( Comparator.comparing(s -> s.length()) )
         .ifPresent(s -> System.out.println(s));
  …
```

THREE

max() returns Optional<String>

Example

- Print out the entry in the stream (list) that has the longest length
- Use method handles as more concise notation

```
…
import java.util.List;
…

  List<String> theList = Arrays.asList("one", "two", "three");
  …

  theList.stream()
         .map( String::toUpperCase )
         .max( Comparator.comparing( String::length ) )
         .ifPresent( System.out::println );
  …
```

THREE

max() returns Optional<String>

Terminal and Non Terminal Operations

- Stream operations are either terminal or intermediate
- Terminal operations
  - return a void or a non stream object
  - examples include `collect` and `forEach`
- Intermediate (or non terminal operations)
  - return another stream of a specific type
  - type inferred
  - allows for chaining of operations

Example

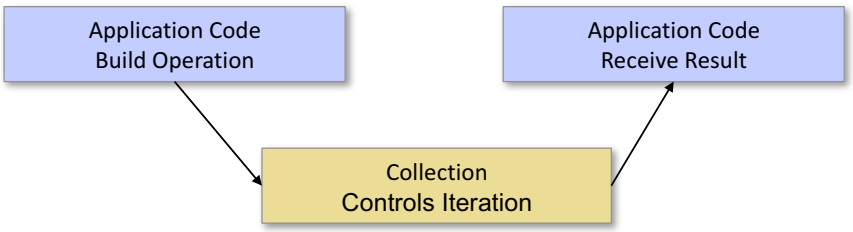
- `Person` is a simple class with a name and age

```
public class Person {
  private String name;
  private int age;
  public Person ( String n, int a ) {
    this.name = n;
    this.age = a;
  }
  public int getAge() {
    return this.age;
  }
  public String getName() {
    return this.name;
  }
}
```

```
Person people[] = {
  new Person("George", 21),
  new Person("John", 32),
  new Person("Jane", 21),
  new Person("Fred", 40)
};

List<Person> thePeople Arrays.asList(people);
```

Internal and External Iteration

- Historically in Java, iteration over a collection controlled via **Iterators**
- Loops make it difficult to sub divide problems
- Large amount of boiler plate code



Map

- Applies a function to transform each element
  - non terminal operation
  - example transforms a Person object to the corresponding name

```
List<String> names = new ArrayList<>();
for ( Person p: thePeople ) {
  names.add(p.getName());
}
```

```
List<String> names = thePeople.stream()
                              .map(Person::getName)
                              .collect(Collectors.toList());
```

Collectors

- `collect` operation is a terminal operation
- Used to transform stream elements into
  - lists, sets, maps etc.
- Accepts a Collector
  - object that handles transformation into result
  - can build your own - implement `java.util.stream.Collector`
- Various built-in collector classes available
  - `Collectors.toSet()`
  - `Collectors.toList()`
  - `Collectors.groupingBy(func) provides  Map<key, List>`
  - `Collectors.averagingInt(func)`

Collectors

- Can transform into a Map

```
Map<Integer, List<Person>> agesMap =
                thePeople.stream()
                         .collect( Collectors.groupingBy(
                                      Person::getAge,
                                      Collectors.toList()
                                   )
                                 );
for ( Integer i : agesMap.keySet() ) {
  System.out.println(i + ": " + agesMap.get(i));
}
```

```
32: [John[32]]
21: [George[21], Jane[21]]
40: [Fred[40]]
```

- Basic statistics can be retrieved
  - using summarizingInt,
  - summarizingLong and
  - summarizingDouble

forEach

- Located on `Iterable` interface and on Stream
  - in place operation on collection
  - *terminal* operation to work on each items on a stream
  - Expects a Consumer argument
    - void return
    - side effect may cause modification of underlying data

```
thePeople.stream().forEach( System.out::println );

thePeople.stream().forEach(
                    p -> p.setName( p.getName()
                                    .toUpperCase() )
                  );

System.out.println("-");
thePeople.stream().forEach( System.out::println );
```

```
George[21]
John[32]
Jane[21]
Fred[40]
-
GEORGE[21]
JOHN[32]
JANE[21]
FRED[40]
```


Filter

- Applies a predicate to remove false elements
  - Items where predicate is true are passed as output stream

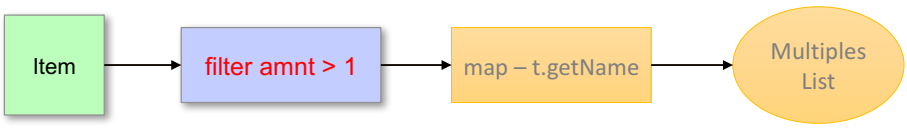
```
Item shoppingList [] = {
  new Item("Tin Beans", 0.50, 4),
  new Item("Milk", 0.60, 1),
  new Item("Sausages", 2.75, 1),
  new Item("Potatoes", 0.45, 2)
};

List<Item> myList = Arrays.asList(shoppingList);
```

```
List<Item> bulkItems = myList.stream()
                      .filter( i -> i.getAmount() > 3 )
                      .collect( Collectors.toList() );

System.out.println(bulkItems);
```

```
[4 of Tin Beans @ 0.5]
```

Pipelining

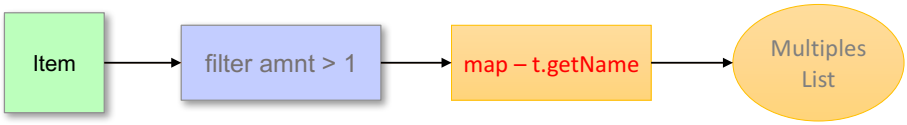
- Create a pipeline of operations and apply to the stream
- Find names of all Items where quantity greater than 1




```
List<String> multiples = myList.stream()
                              .filter( i -> i.getAmount() > 1 )
                              .map(Item::getName)
                              .collect(Collectors.toList());
```


Pipelining

- Create a pipeline of operations and apply to the stream
- Find names of all Items where quantity greater than 1




```
List<String> multiples = myList.stream()
                              .filter( i -> i.getAmount() > 1 )
                              .map(Item::getName)
                              .collect(Collectors.toList());
```

```
[Tin Beans, Potatoes]
```

Pipelining

- Can generate as complex a pipeline as required

```
List<String> processedList =
    myList.stream()
          .filter(t -> t.getAmount() > 1)
          .map(t -> t.getName())
          .map(String::toLowerCase)
          .sorted()
          .collect(Collectors.toList());

System.out.println(processedList);
```

```
[potatoes, tin beans]
```


FlatMap

- `flatmap` applies map operation and "flattens" results
  - useful for dealing with nested streams
- Consider contents of a text file read into a collection

```
String [] lines = {
  "Here is the first line of the file",
  "Here is the second line",
  "And here is the third line"
};
List<String> contents = Arrays.asList(lines);
```

- We are interested in the individual words in each line

FlatMap

- `String::split` can be used to split line into words
  - returns array of String
  - can be turned into a stream of String
- Use `flatMap` to combine into a single stream
  - then a single `List`

```
List<String> allWords =
      contents.stream()
              .flatMap( l -> Arrays.asList(l.split(" "))
                                   .stream() )
              .collect(Collectors.toList());

System.out.println(allWords);
```

```
[Here, is, the, first, line, of, the, file, Here, is, the,
second, line, And, here, is, the, third, line]
```


Count

- It is also possible to get a count of elements in the stream

```
long numWords = contents.stream()
          .flatMap( l -> Arrays.asList(l.split(" ")).stream())
          .count();
```

- Another useful option is `.distinct()`
  - uses `.equals(Object o)` to determine distinct members of the stream

```
long numDistinct = contents.stream()
        .flatMap( l -> Arrays.asList(l.split(" ")).stream())
        .distinct()
        .count();
System.out.println( numWords + " words (" + numDistinct + " distinct)");
```

```
19 words (11 distinct)
```

Sorting a Stream

- Streams are easily sorted
  - default is to use natural ordering of sort key type

```
List<String> sortedWords = contents.stream()
              .flatMap( l -> Arrays.asList(l.split(" ")).stream())
              .sorted()
              .collect(Collectors.toList());

System.out.println(sortedWords);
```

```
[And, Here, Here, file, first, here, is, is, is, line, line,
line, of, second, the, the, the, the, third]
```


Sorting a Stream

- Specify a sort function using the static function
  `Comparator.comparing`

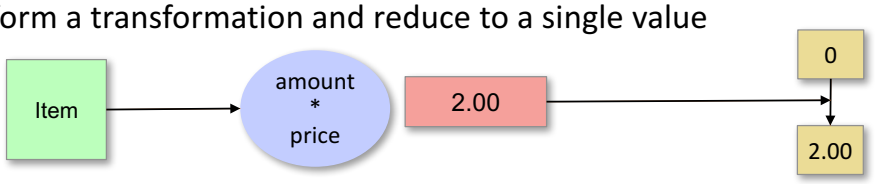
```
import static java.util.Comparator.comparing;

List<String> sortedWords2 = contents.stream()
              .flatMap( l -> Arrays.asList(l.split(" ")).stream() )
              .sorted( comparing(String::toLowerCase) )
              .collect( Collectors.toList());

System.out.println(sortedWords2);
```

```
[And, file, first, Here, Here, here, is, is, is, line, line,
line, of, second, the, the, the, the, third]
```

- Can also be used outside Streams infrastructure

Map Reduce

- Perform a transformation and reduce to a single value

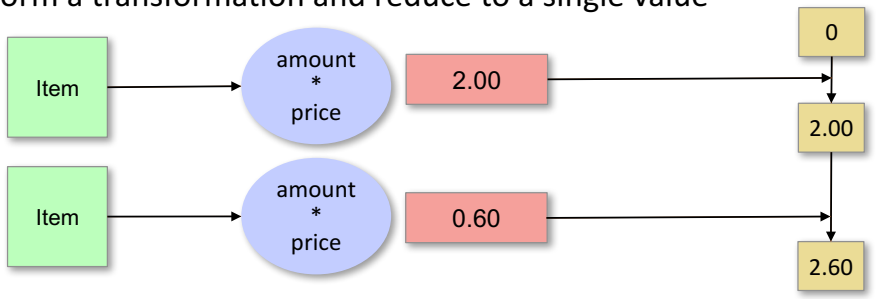



```
Item shoppingList [] = {
  new Item("Tin Beans", 0.50, 4),
  new Item("Milk", 0.60, 1),
  new Item("Sausages", 2.75, 1),
};
List<Item> myList = Arrays.asList(shoppingList);
```

```
double totalCost = myList.stream()
                         .map( it -> it.getAmount() * it.getPrice() )
                         .reduce( 0.0, (a,b) -> a + b );
```


Map Reduce

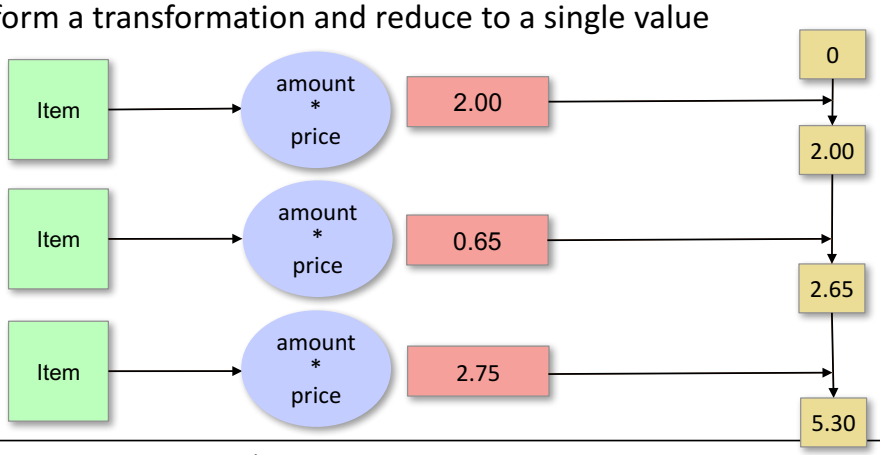
- Perform a transformation and reduce to a single value




```
double totalCost = myList.stream()
                         .map( it -> it.getAmount() * it.getPrice() )
                         .reduce(0.0, (a,b) -> a+b);
```

Map Reduce

- Perform a transformation and reduce to a single value




```
double totalCost = myList.stream()
                         .map( it -> it.getAmount() * it.getPrice() )
                         .reduce(0.0, (a,b) -> a+b);
```


Using `Optional<T>` with Streams

- Provides convenient way of handling empty Stream

```
…
 public static int largestPositive(List<Integer> l ) {
  return l.stream()
          .filter(i -> i >= 0 )
          .sorted((x,y) -> y - x)
          .collect(Collectors.toList())
          .get(0);
 }
…
```

```
…
 List<Integer> values = Arrays.asList(1,-4, -3, 4, -10, 11, 0);
 List<Integer> allNegative = Arrays.asList(-1, -4, -3, -4,-10, -11);
…
 System.out.println( largestPositive(values) );
 System.out.println( largestPositive(allNegative) );
…
```

```
11
Exception in thread "main" java.lang.IndexOutOfBoundsException: Index: 0, Size: 0
        at java.util.ArrayList.rangeCheck(ArrayList.java:653)
        at java.util.ArrayList.get(ArrayList.java:429)
…
```

Using `Optional<T>` with Streams

- Use alternative method to extract head of Stream

```
…
 public static Optional<Integer> safeLargest(List<Integer> l ) {
  return l.stream()
          .sorted((x,y) -> y - x)
          .findFirst();
 }
…
```

```
…
public static Optional<Integer> safeLargestPositive(
                                                List<Integer> l ) {
  return l.stream()
          .filter(i -> i >= 0 )
          .sorted((x,y) -> y - x)
          .findFirst();
 }
…
```


Using `Optional<T>` with Streams

- Values now processed successfully

```
…
 System.out.println( safeLargest(values)
                                    .map( n -> n * n)
                                    .orElse(-1) );
 System.out.println( safeLargest(empty)
                                    .map( n -> n * n)
                                    .orElse(-1) );
 System.out.println( safeLargestPositive(values)
                                    .map( n -> n * n)
                                    .orElse(-1) );
 System.out.println( safeLargestPositive(allNegative)
                                    .map( n -> n * n)
                                    .orElse(-1) );
…
```

```
121
-1
121
-1
```

Spliterator and parallel streams

- Streams make it very easy to execute code in parallel
  - Utilising app-wide fork-join thread pool
  - how much work without streams?

```
trades.parallelStream()
      .filter(t -> t.getQuantity() > 20)
      .map(t -> t.getSymbol())
      .limit( 5 )
      .collect( Collectors.toList() );
```

- Collections should be immutable
  - for safe parallel processing
  - either make immutable (Collections.unmodifiableList) or take a copy
- The spliterator is used under the hood
  - to divide the stream up into chunks for processing

Notes on Parallel Streams

- Making a stream parallel may not always result in speedup
- Beware Amdahl's Law
- Certain stream operations may require serial operation
  - Stateful (e.g. `distinct()`)
  - `sorted()`
  - `findFirst()` (use `findAny()` instead)
- Cost of splitting stream may be significant
  - Arrays easy, linked list very slow