Java Streams

CMSC132
Java Streams: What are they (not)?

- Not related to Java IO Streams!
  - (FileInputStream, InputStreamReader, etc. -- none of those guys)

- A basic way to do functional programming in Java
What is this... “functional” programming?

- Basic idea: issue **methods** as arguments to other methods,
  - there, execute them with local data as arguments
A Functional Programming Parable

1. You pass a drill to a worker
2. He uses it to drill a hole in the wall next to him
But what is a “Stream”?

- Think “Streaming Collection of Elements”
- Can have different sources
  - Java Collections
  - Arrays
  - A sequence of individual objects
- A sequence of operations can be applied
- Results not available until “terminal” operation
How to make streams?

- Import Stream-related things from java.util.stream
  - `import java.util.stream.*` imports everything related
- Method 1: build from a static array or individual objects using Stream.of
  - `String[] menuItemNames = {"Grits", "Pancakes", "Burrito"};
    Stream.of(menuItemNames); // returns a stream, so needs "=" before it
  - `Stream.of("Hedgehog", "Kitten", "Fox"); // arbitrary argument count`
- Method 2: call the `stream()` method of any `Collection`
  - `List<String> menuItemNameList = Arrays.asList(menuItemNames);
    menuItemNameList.stream();`
- Method 3: use the StreamBuilder class and it’s “accept” method.
forEach

- **Intuition** → iterate over elements in the stream
- Lambda has one argument, return value is ignored
- Terminal operation: does not return another stream!

```java
Stream.of(users).forEach(e -> e.logOut());
```
- Logs out all users in system
forEach

- Loops over stream elements, calling provided function on each element
  - `Stream.of("hello", "world").forEach(word -> System.out.println(word));`
  - A lambda argument is passed

- Can also pass “method references”
  - `Stream.of("hello", "world").forEach(System.out::println);`
  - Syntax: `class::method`
Some More Common Stream Operations

**map**
Applies a function to each element

**filter**
Removes elements that don't satisfy a custom rule

**sorted**
Sorts elements

**limit**
Return the first N elements

**distinct**
Removes duplicates

**collect**
Gets elements out of the stream once we’re done (terminal operation)
collect (Basics)

- Also a terminal method.
- Let’s say we start with
  - `Stream<Integer> stream = Arrays.asList(3, 2, 1, 5, 4, 7).stream();`
- Some basic examples: just output all elements as a collection.
  - `List<Integer> list = stream.collect(Collectors.toList());`
  - `Set<Integer> list = stream.collect(Collectors.toSet());`
- Lots more useful goodies,
  - like `Collectors.groupingBy(f)` and `Collectors.reducing(f);`
map

- **Intuition** → modifies the elements of the stream
- The function takes an element of type T and returns an element of type K.

\[
T \rightarrow f \rightarrow K
\]

Stream\( \langle T \rangle \).map(\( f \)) \rightarrow Stream\( \langle K \rangle \)

- Example:
  
  ```java
  List<Integer> numbersTripled =
      numbers.stream().map(x -> x*3).collect(toList());
  ```
List<Integer> numbersTripled =
    numbers.stream().map(x -> x*3).collect(toList());

[ 1 2 3 4 5 6 ]
List<Integer> numbersTripled =
    numbers.stream().map(x -> x*3).collect(toList());

[ 1 2 3 4 5 6 ]
\[ \rightarrow \]
\[ f \]
\[ \rightarrow \]
\[ [ 3 ] \]
List<Integer> numbersTripled =
    numbers.stream().map(x -> x*3).collect(toList());

\[
\begin{array}{cccccc}
1 & 2 & 3 & 4 & 5 & 6 \\
\end{array}
\]

\[
\xrightarrow{f}
\]

\[
\begin{array}{cccc}
3 & 6 & \\
\end{array}
\]
List<Integer> numbersTripled =
    numbers.stream().map(x -> x*3).collect(toList());

[ 1 2 3 4 5 6 ]
  ↓
    f
  ↓

[ 3 6 9 ]
List<Integer> numbersTripled =
    numbers.stream().map(x -> x*3).collect(toList());

[ 1 2 3 4 5 6 ]
    ↓
    f

[ 3 6 9 12 ]
List<Integer> numbersTripled =
    numbers.stream().map(x -> x*3).collect(toList());

[  1  2  3  4  5  6  ]
    ↓
      f

[  3  6  9 12 15  ]
List<Integer> numbersTripled =
    numbers.stream().map(x -> x*3).collect(toList());

[  1  2  3  4  5  6  ]  
    ↓  ↓
[  3  6  9 12 15 18  ]
The function \( f \) can be a...

- One-liner lambda expression
  \[
  \text{map}(x \rightarrow x/2)
  \]

- More complex lambda expression
  \[
  \text{map}(x \rightarrow \{
      \ldots \text{some code} \ldots
      \text{return something;}
  \})
  \]

- Just any function
  \[
  \text{map}(	ext{String::toUpperCase})
  \]
• **Intuition** → keeps elements satisfying some condition
• Lambda has one argument and produces a boolean
• Value of boolean determines whether item should be kept

```java
List<Integer> goodYears = years
    .stream().filter(y -> y != 2020).collect(toList());
```
List<Integer> goodYears = years
    .stream().filter(y -> y != 2020).collect(toList());
List<Integer> goodYears = years
    .stream().filter(y -> y != 2020).collect(toList());

For each element $y$, what does $y \neq 2020$ evaluate to?
List<Integer> goodYears = years.filter(y -> y != 2020).collect(toList());


y != 2020 evaluates to true

For each element y, what does y != 2020 evaluate to?
List<Integer> goodYears = years
    .stream().filter(y -> y != 2020).collect(toList());


y != 2020 evaluates to true

[ 2000

For each element y, what does y != 2020 evaluate to?
List<Integer> goodYears = years
  .stream().filter(y -> y != 2020).collect(toList());
^ 
y != 2020 evaluates to true

[ 2000 2005 ]

For each element \( y \), what does \( y \neq 2020 \) evaluate to?
List<Integer> goodYears = years
    .stream().filter(y -> y != 2020).collect(toList());

```
```

y != 2020 evaluates to true

```
[ 2000 2005 2010 ]
```

For each element \( y \), what does \( y \neq 2020 \) evaluate to?
List<Integer> goodYears = years.stream().filter(y -> y != 2020).collect(toList());


y != 2020 evaluates to true


For each element y, what does y != 2020 evaluate to?
List<Integer> goodYears = years
    .stream().filter(y -> y != 2020).collect(toList());

For each element \( y \), what does \( y \neq 2020 \) evaluate to?
List<Integer> goodYears = years
  .stream().filter(y -> y != 2020).collect(toList());


y != 2020 evaluates to true


For each element y, what does y != 2020 evaluate to?
List<Integer> goodYears = years
    .stream().filter(y -> y != 2020).collect(toList());

Result: new stream only containing values satisfying \( y \neq 2020 \)

List<Integer> leapYears =
    years.stream().filter(y -> {
        if (y % 400 == 0) return true;
        if (y % 100 == 0) return false;
        if (y % 4 == 0) return true;
        return false;
    }).collect(toList());

- No requirement to have simple or one-liner condition
- Reminder: lambda is anonymous class implementing functional interface
- Implements Predicate<T> which has boolean test(T t)
var numbers = Arrays.asList(3, 2, 1, 5, 4, 7);
numbers.stream().sorted().forEach(System.out::println);

[  3  2  1  5  4  7  ]

Result: new stream only containing values

[  1  2  3  4  5  7  ]
var numbers = Arrays.asList(3,3,1,1,4,7,8);
numbers.stream().distinct().forEach(System.out::println);

[ 3 3 1 1 4 7 8 ]

Result: new stream only containing values

[ 3 1 4 7 8 ]
```java
var numbers = Arrays.asList(3, 2, 2, 3, 7, 3, 5);
numbers.stream().limit(4).forEach(System.out::println);
```

Result: new stream only containing values

```java
[ 3 2 2 3 ]
```
collect (Reductions)

- `Stream.collect()` allows us to “reduce” a stream to a single output
- This process is called a “reduction”

Some scenarios:

- A list of vote counts in many districts of a state for two candidates can be reduced to an aggregate vote count for each candidate.
- A list of heights for athletes in a basketball team can be reduced to an average height for the whole team.
- A list of ages of students in a class can be reduced to the maximum (oldest) age in the class.
Create a list of heights (in inches) of team members on a Basketball team
```
List<Integer> teamHeights = List.of(73, 68, 75, 77, 74);
```

Collect using a “reducer” created with `Collectors.reducing`
`Collectors.reducing()` accepts initial accumulator value and a function with two parameters:
current value of accumulator and current stream element value
```
int totalHeight = teamHeights.stream().collect(
    Collectors.reducing(0, (accumulator, curr) -> (accumulator + curr))
);
```

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

- Prints: **367**
collect (Reductions)

Collectors.reducing(0, (accumulator, curr) -> (accumulator + curr))

[ 73, 68, 75, 77, 74 ]

^

Accumulator value: 0
Current stream element: 73
New accumulator value: 73
collect (Reductions)

Collectors.reducing(0, (accumulator, curr) -> (accumulator + curr))

[ 73, 68, 75, 77, 74 ]

Accumulator value: 73
Current stream element: 68
New accumulator value: 141
collect (Reductions)

Collectors.reducing(0, (accumulator, curr) -> (accumulator + curr))

[ 73, 68, 75, 77, 74 ]

^

Accumulator value: 141
Current stream element: 75
New accumulator value: 216
collect (Reductions)

Collectors.reducing(0, (accumulator, curr) -> (accumulator + curr))

[ 73, 68, 75, 77, 74 ]

^

Accumulator value: 216

Current stream element: 77

New accumulator value: 293
collect (Reductions)

Collectors.reducing(0, (accumulator, curr) -> (accumulator + curr))

[73, 68, 75, 77, 74]

Accumulator value: 293
Current stream element: 74
New accumulator value: 367 (Final result)
Some More Common Stream Operations

- **count**
  Counts all elements in a stream (terminal)

- **skip**
  Gets rid of the first N elements

- **findFirst**
  Gets the first stream element wrapped in Optional (terminal)

- **toArray**
  Return elements as an array (terminal)

- **flatMap**
  Flatten the data structure (e.g. on stream consisting of Lists)

- **peek**
  Do something with each item (like forEach, but not terminal)
**Customer asks:**
What are the calorie counts for three of your lowest-calorie breakfast options, excluding salads?

**Diagram:**
- **Collection**
  - Grits
  - Pancakes
  - Burrito
  - Bacon & Eggs
  - Greek Salad
  - Caesar Salad
  - Sandwich

- **Stream**
  - Filters out anything with "salad" in the name

- **Stream**
  - Extracts calorie counts for each food

- **Stream**
  - Sort in ascending order

- **Stream**
  - Remove duplicate values

- **Stream**
  - Return the first three elements

<table>
<thead>
<tr>
<th>Grits</th>
<th>Pancakes</th>
<th>Burrito</th>
<th>Bacon &amp; Eggs</th>
<th>Sandwich</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>235</td>
<td>330</td>
<td>875</td>
<td>450</td>
</tr>
<tr>
<td></td>
<td>330</td>
<td></td>
<td>330</td>
<td>450</td>
</tr>
<tr>
<td></td>
<td>875</td>
<td></td>
<td>450</td>
<td>875</td>
</tr>
</tbody>
</table>

**Sorted Stream:***
- 235
- 330
- 450

**Distinct Stream:***
- 235
- 330
- 450

**Limit Stream:***
- 235
- 330
- 450
Streams in code...

- Allude to code example in Eclipse.
- End of presentation. Questions?