Java 8 introduces a new way to process a collection of items, by using streams. This has nothing to do with file streams or I/O streams. Rather, Java 8 streams are more like internal iterators, in the sense that they allow you to apply operations sequentially on each item in a collection.

Contents

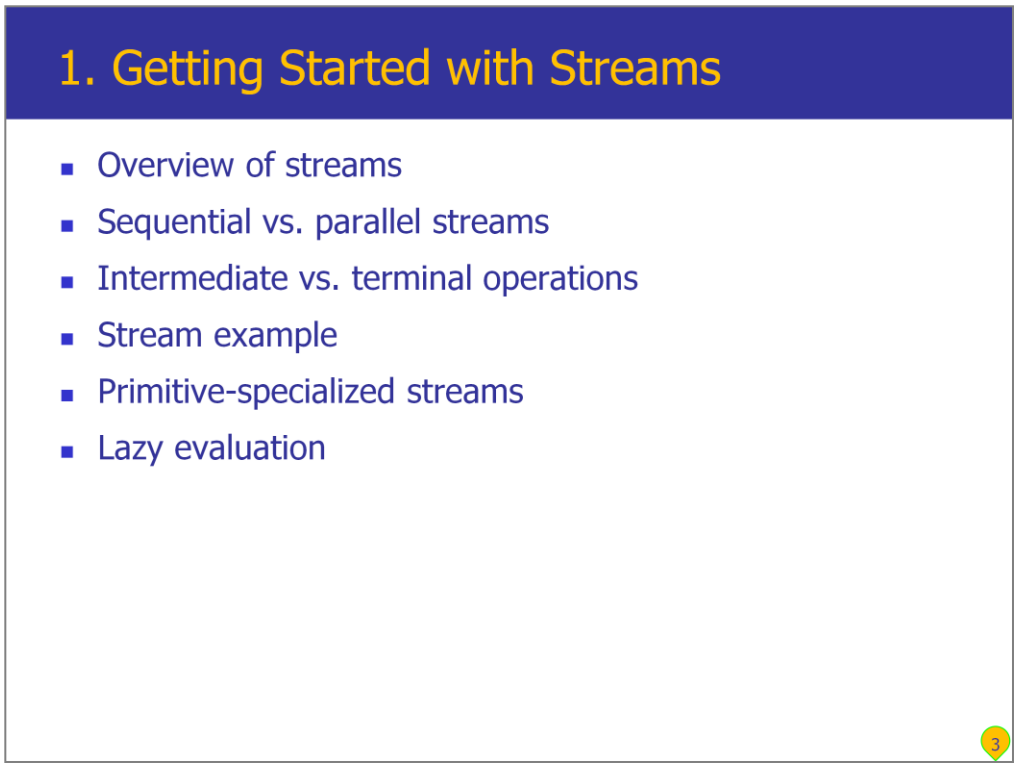
1. Getting started with streams
2. A closer look at stream operations

Demo project:
DemoStreams

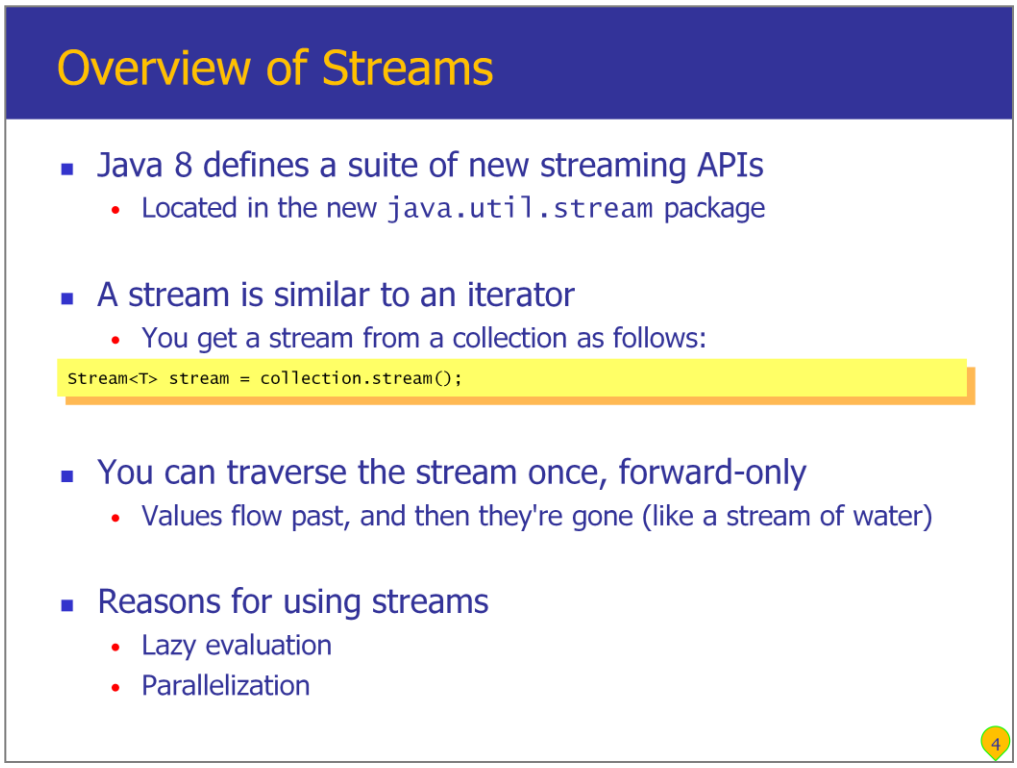
2

Section 1 describes how to obtain a stream on a collection, and shows some simple operations you can apply.

Section 2 takes a closer look at the two types of operations you can apply on a stream, i.e. intermediate operations and terminal operations.



This section describes what a stream is, how to get a stream on a collection, and how to perform simple operations on a stream.

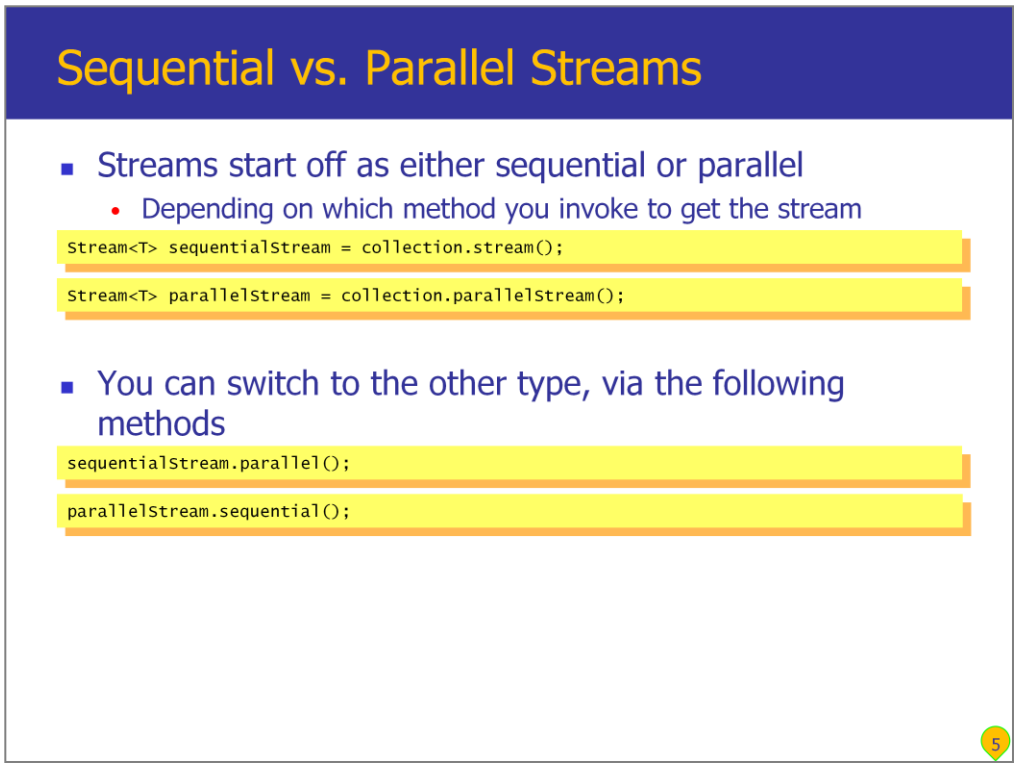


The Java 8 streaming APIs are defined in the `java.util.stream` package, which is new in Java 8.

The collections API has also been extended with various new methods that allow you to obtain a stream on a collection. For example, the `Collection` interface has a couple of new default methods as follows:

```
public interface Collection<T> {
    …
    default Stream<T> stream();
    default Stream<T> parallelStream();
}
```

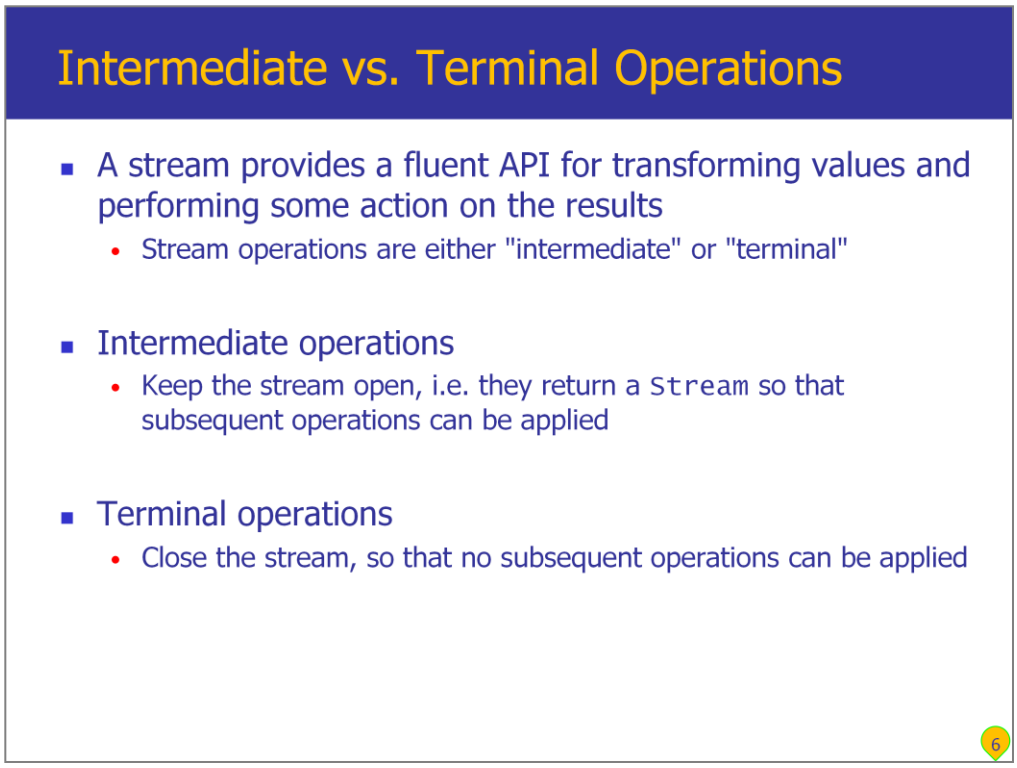
These methods obtain a sequential and parallel stream on a collection, respectively. We describe the difference between sequential and parallel streams on the next slide.



A stream can be either sequential or parallel:

- Actions of a sequential stream occur serially on one thread
- Actions of a parallel stream may occur in parallel on multiple threads

A stream starts off as either sequential or parallel, depending on whether you use the `stream()` or `parallelStream()` method to get the stream. You can switch to the other type, via the `parallel()` and `sequential()` methods.



Once you've obtained a stream on a collection, you can perform operations on each element in the stream. The Stream interface defines many methods that perform a particular operation on each element in turn.

There are two types of operations available:

- Intermediate operations, which can be connected together to perform a pipeline. They can be connected together because their return type is a `Stream`.
- Terminal operations, which close the pipeline and return a non-`Stream` result. For example, the result might be a numeric value, a collection, or even `void`.

Stream Example

- When using a stream, you typically follow these steps:
  1. Obtain a stream from some source
  2. Perform one or more intermediate operations
  3. Perform one terminal operation

- Example (see `SimpleStreamDemo.java`)

```
import java.util.stream.Stream;
…

List<Double> hoursWorked = Arrays.asList(7.5, 8.25, 9.0, 7.0, 6.5);

Stream<Double> stream = hoursWorked.stream();

double sumExtraHours = stream.filter(h -> h > 7.5)          // Intermediate operation.
                             .mapToDouble(h -> h - 7.5)     // Intermediate operation.
                             .sum();                        // Terminal operation.

System.out.println("Sum of extra hours worked this week: " + sumExtraHours);
```


This slide shows a simple example of how to get a stream from a collection, and then apply operations on the elements in the collection. The example performs three operations on the stream:

- `filter()`
  This is an intermediate operation in the `Stream<T>` interface. It takes a `Predicate<T>` as a parameter, and returns a `Stream<T>` consisting of the elements of the original stream that match the predicate.
- `mapToDouble()`
  This is another intermediate operation in the `Stream<T>` interface. It takes a `ToDoubleFunction<T>` as a parameter, and returns a `DoubleStream` consisting of the results of applying the given function to the elements of the input stream. We'll discuss `DoubleStream` on the next slide.
- `sum()`
  This is terminal operation in the `DoubleStream` interface (and other similar interfaces). It returns the sum of all the `double` values in the stream.

Primitive-Specialized Streams

- There are primitive-specialized versions of `Stream`:
  - `IntStream` is a stream of `int`
  - `LongStream` is a stream of `long`
  - `DoubleStream` is a stream of `double`
- Example:
  - See `PrimitiveStreamsDemo.java`

```
List<Employee> staff = Employee.generateStaff();
Stream<Employee> empStream = staff.stream();

DoubleStream salaryStream = empStream.mapToDouble(Employee::getSalary);

salaryStream.forEach(s -> System.out.println("  " + s));
```

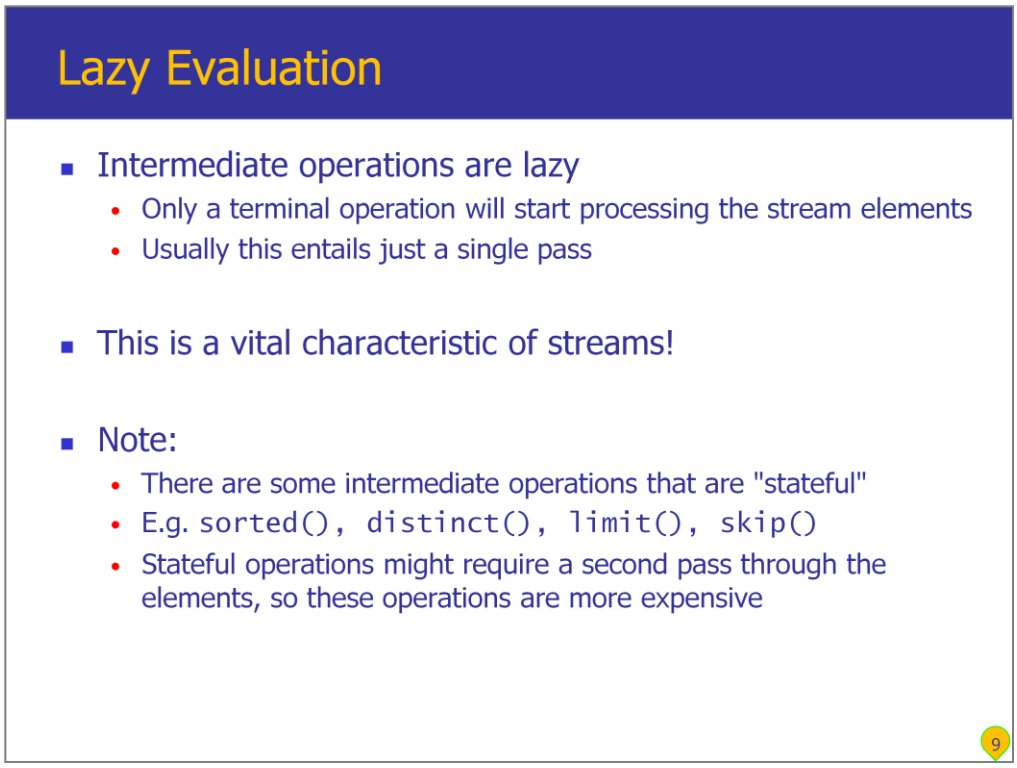
8

`IntStream`, `LongStream`, and `DoubleStream` are type-specific specializations of the `Stream<T>` interface. These three interfaces provide additional methods that deal specifically with `int`, `long`, and `double` values. Consider the example in the slide.

- First we create a collection of `Employee` objects. Each employee has a name, salary, and so on. You can find the full definition of the `Employee` class in the demo project.
- Next we create a stream on the collection, to allow us to apply operations on each `Employee` object.
- Then we call `mapToDouble()` to map the stream of employees into a stream of `double` values. `mapToDouble()` takes a `ToDoubleFunction<T>` parameter; this functional interface expects a method that takes a `T` and returns a `double`. To satisfy this expectation, we pass in a reference to the `Employee::getSalary` instance method. Therefore, `mapToDouble()` will invoke `Employee::getSalary` on each `Employee` object, to convert the stream of employees into a stream of `double` values.
- Finally we call `forEach()` on the stream, to perform an action on each element. `forEach()` takes a `Consumer<T>` parameter (or a specialization in this case, `DoubleConsumer`). We pass in a simple lambda expression to display values on the console.

Note: `PrimitiveStreamsDemo.java` also includes examples of `mapToLong()`

and `mapToInt()`. See the demo project for details.



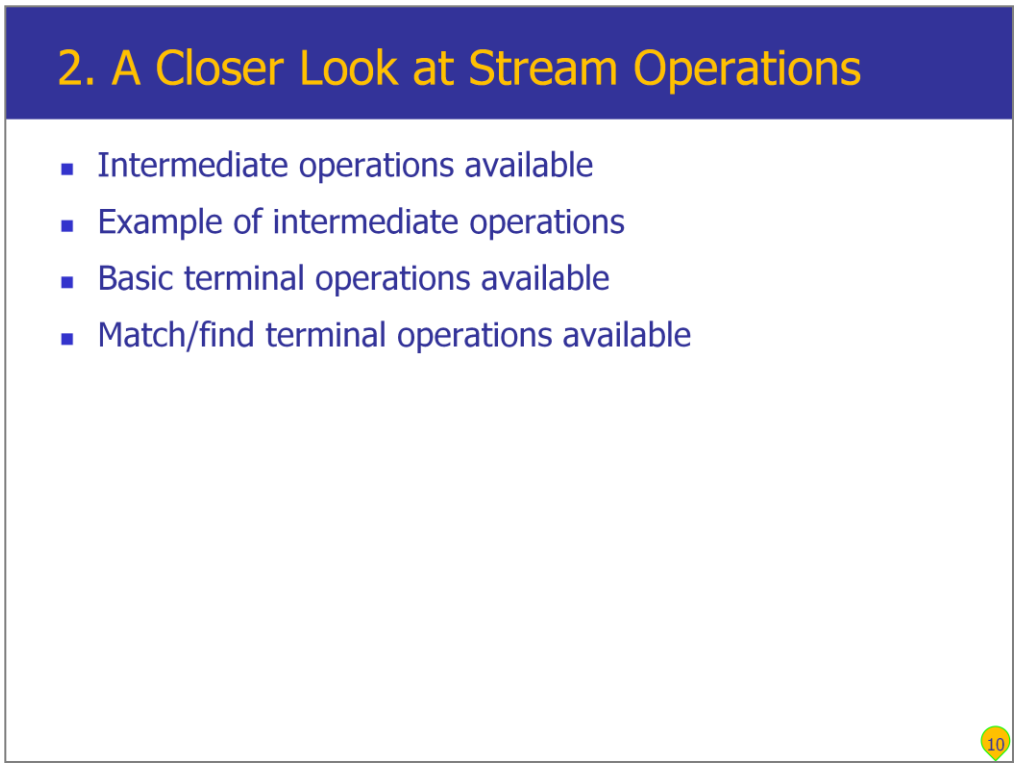
All intermediate operations are "lazy", i.e. expressions are only evaluated when required. Conversely, an algorithm is "eager" if it is evaluated or processed immediately. Intermediate operations are lazy because they don't start processing the contents of the stream until the terminal operation commences.

Processing streams lazily enables the Java compiler and runtime to optimize how they process streams. E.g. consider the `filter()`,`mapToDouble()`,`sum()` example described a few slides earlier; the `sum()` operation could obtain the first few doubles from the stream created by `mapToDouble()`, which obtains elements from the `filter()` operation. The `sum()` operation would repeat this process until it had obtained all required elements from the stream.

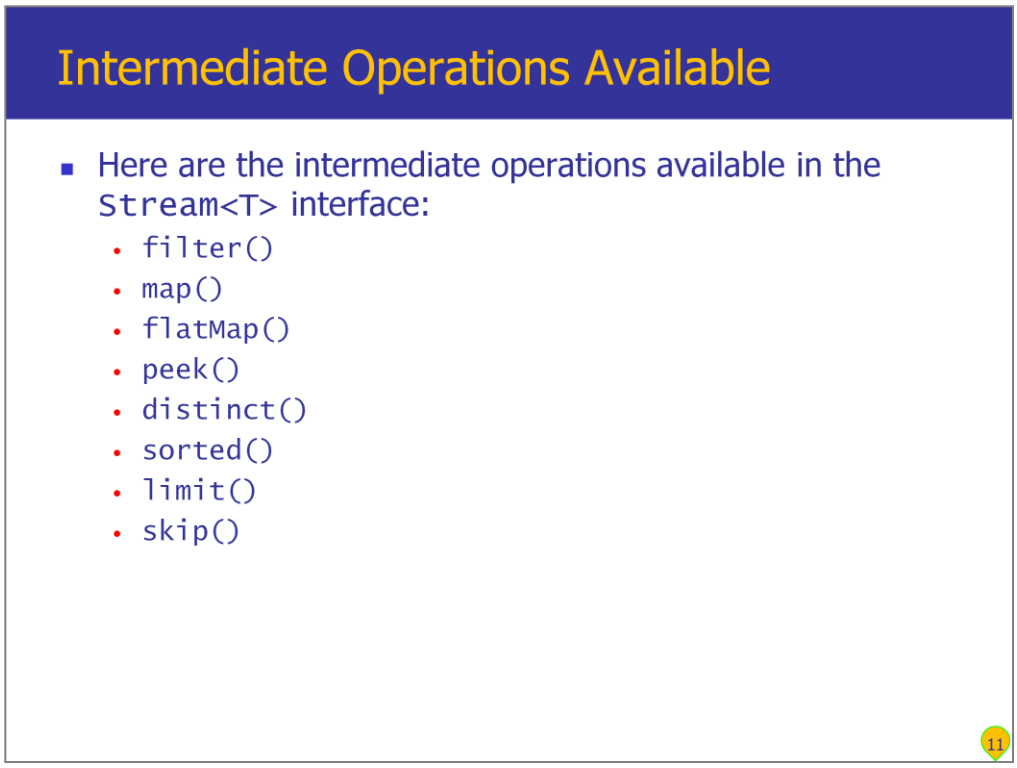
Intermediate operations are divided into stateless and stateful operations:

- Stateless operations, e.g. `filter()` and `map()`, retain no state from previously seen element when processing a new element. Elements can be processed independently of operations on other elements.
- Stateful operations, e.g. `sorted()` and `distinct()`, may incorporate state from previously seen elements when processing new elements. Stateful operations may need to process the entire input before producing a result. E.g. it's not possible to produce any results from sorting a stream until all the elements have been seen. As a result, under parallel computation, some pipelines containing stateful operations may require multiple passes on the data or may need to buffer significant data. In contrast, pipelines containing just stateless intermediate operations can be processed in a single pass,

whether sequential or parallel, with minimal data buffering.



In this section we're going to explore the various stream operations available in Java 8. We'll discuss intermediate operations first, then we'll take a look at terminal operations.



Here's a brief description of the intermediate operations available in `Stream<T>`. Each of these operations is stateless, unless indicated below (the JavaDoc for each method indicates if it's stateful or stateless):

- `filter()` – Exclude elements that don't match a `Predicate<T>`
- `map()` – Transform elements to new values via a `Function<T,R>`
- `flatMap()` – Transform each element into zero or more elements
- `peek()` – Perform action on each element, e.g. for debugging
- `distinct()` – Exclude duplicate elements (stateful)
- `sorted()` – Sort elements (stateful)
- `limit()` – Limit number of elements to subsequent ops (stateful)
- `skip()` – Skip the first n elements in the stream (stateful)

Examples of Intermediate Operations

- Here are some examples of intermediate operations
  - See `IntermediateOperationsDemo.java`

```
List<Integer> numbers = generateRandomNumbers();

Stream<Integer> stream = numbers.stream();

stream.filter(n -> n > 70)
      .peek(n -> System.out.println("Raw mark: " + n))
      .mapToInt(n -> n - 70)
      .distinct()
      .sorted()
      .limit(10)
      .forEach(n -> System.out.println("Score above A-grade threshold: " + n));
```

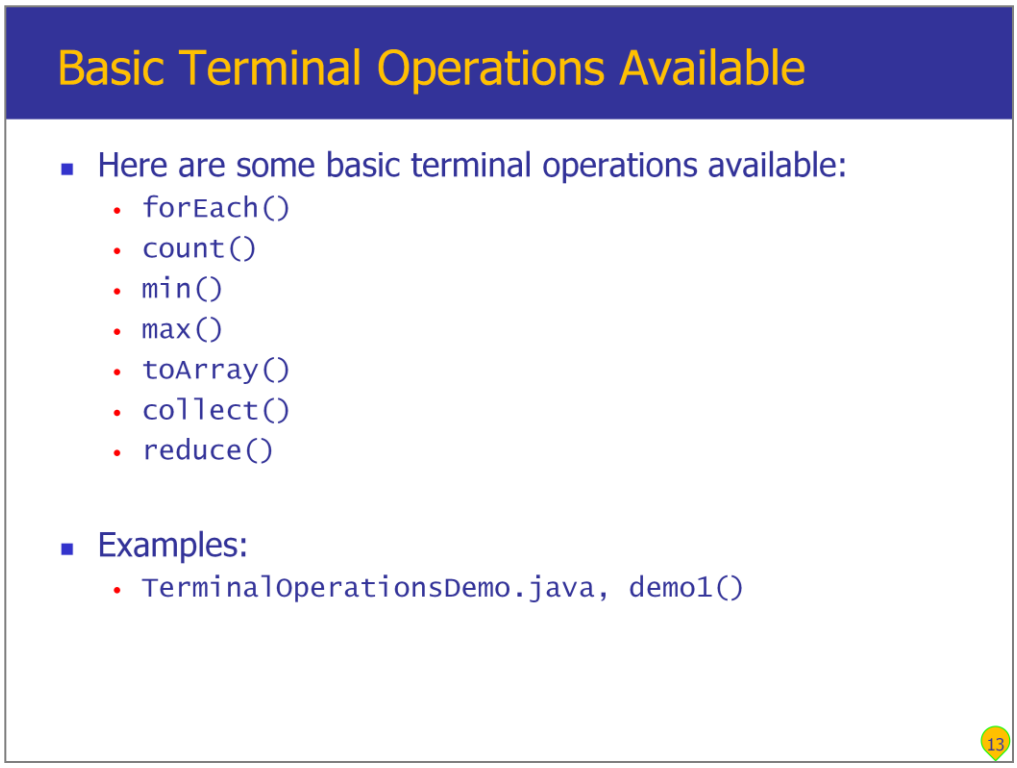
```
System.out.println("Example of flatmap()");
Stream<List<Integer>> intListStream = Stream.of(
    Arrays.asList(1, 2),
    Arrays.asList(3, 4),
    Arrays.asList(5)
);

Stream<Integer> intStream = intListStream .flatMap((intList) -> intList.stream());

intStream.forEach(System.out::println);
```

12

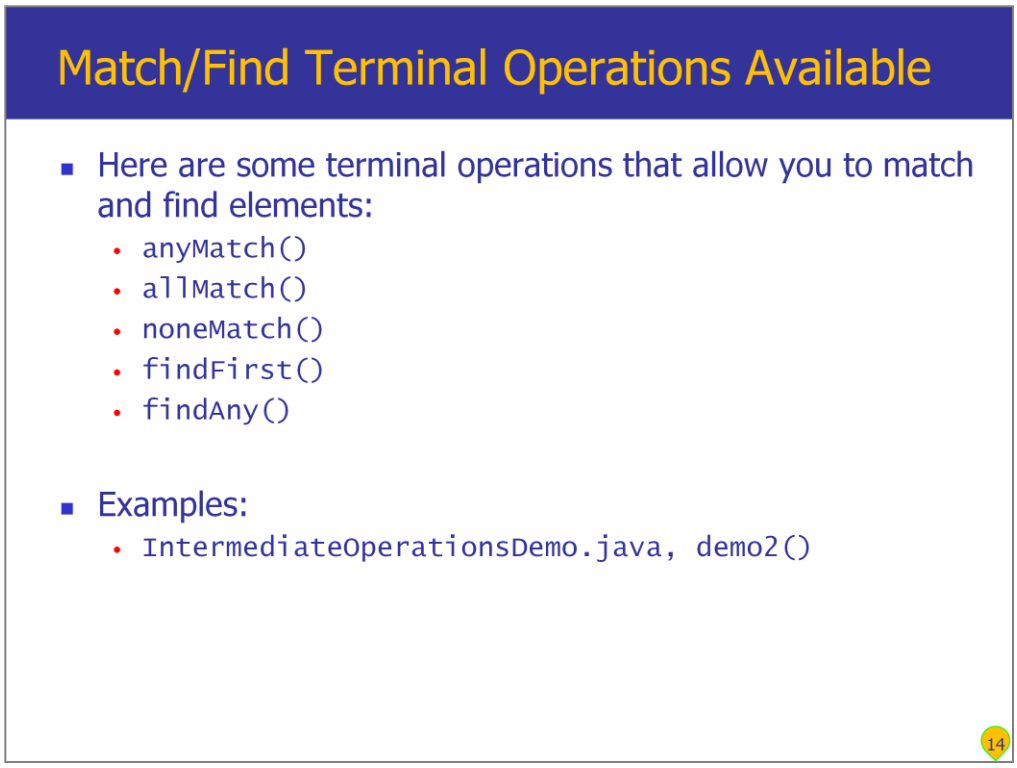
This slide shows how to use some of the intermediate operations listed on the previous slide. At the end of the pipeline of intermediate operations, we have the terminal `forEach()` operation.



Here's a brief description of some of the basic terminal operations available in the `Stream<T>` interface:

- `forEach()` – Perform an action on each element
- `count()` – Return the number of elements
- `min()` – Return the minimum element
- `max()` – Return the maximum element
- `toArray()` – Copy the elements to an array
- `collect()` – Collect the elements to a collection / map
- `reduce()` – Combine elements into one via a `BinaryOperator`

For an example of how to use these terminal operators, see the `demo1()` method in `TerminalOperationsDemo.java`.



`Stream<T>` has terminal operations that allow you to match elements and to find elements. Here's a brief description:

- `anyMatch()` – Determine if any element matches a `Predicate`
- `allMatch()` – Determine if all elements match a `Predicate`
- `noneMatch()` – Determine if no elements match a `Predicate`
- `findFirst()` – Find the first element in the stream
- `findAny()` – Find any element in the stream (may be cheaper)

For an example of how to use these terminal operators, see the `demo2()` method in `TerminalOperationsDemo.java`.

Any Questions?

15