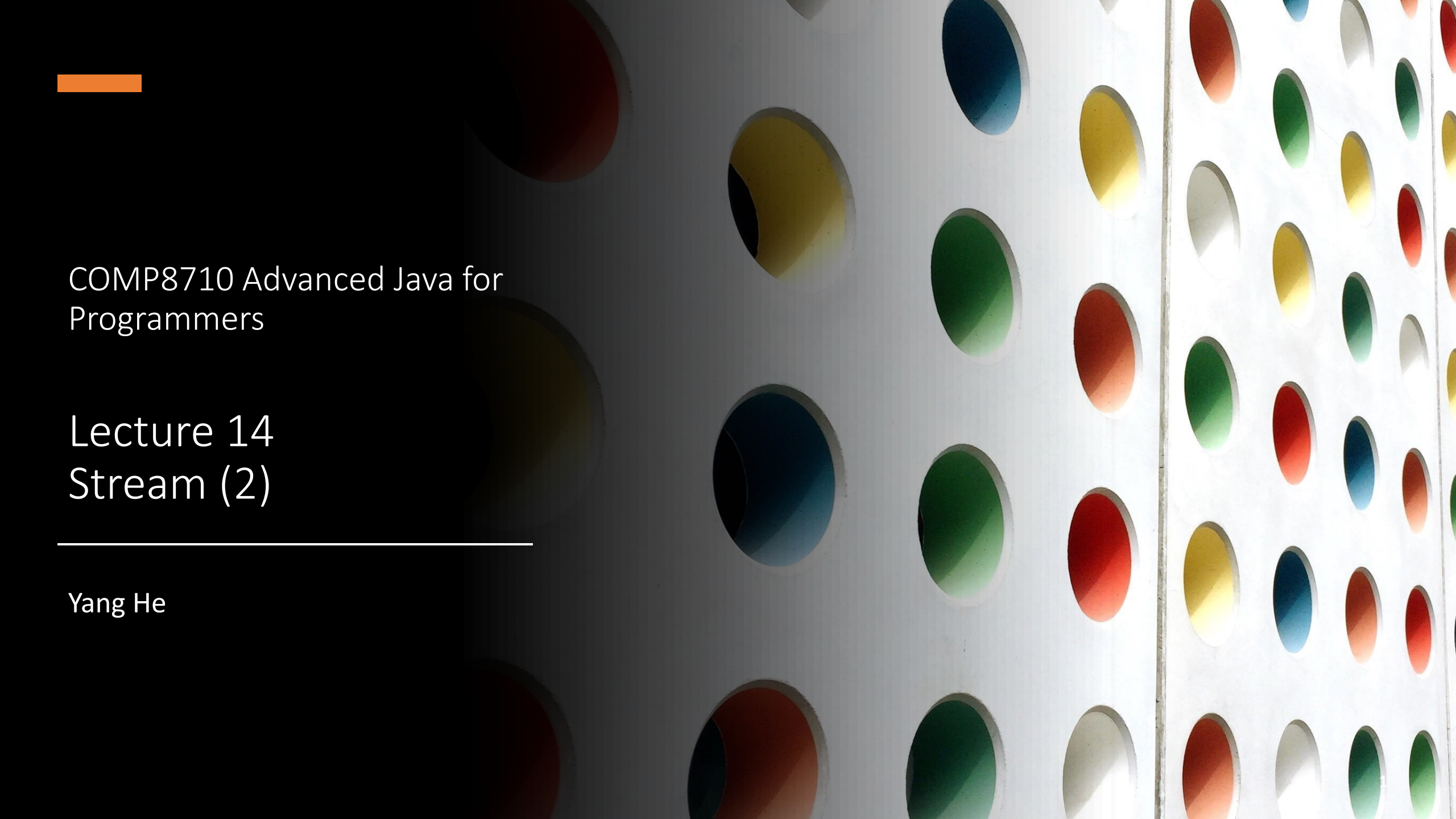


COMP8710 Advanced Java for Programmers

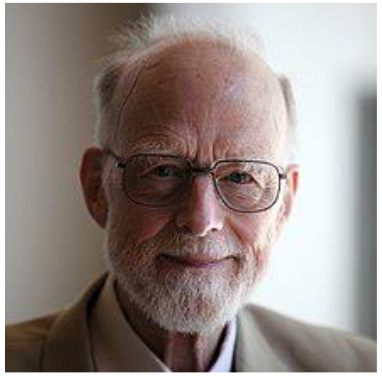
Lecture 14
Stream (2)

Yang He

Topics

- Introduction to Java Streams
- Stream operations
- Collections vs. streams
- Matching with Optional
- Stateless vs. stateful
- Infinite streams

Optional Interlude



Tony Hoare

"I call it my billion-dollar mistake. It was the invention of the null reference in 1965. At that time, I was designing the first comprehensive type system for references in an object oriented language (ALGOL W). My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has *led to innumerable errors, vulnerabilities, and system crashes*, which have probably caused a billion dollars of pain and damage in the last forty years."

Optional class

- `Optional<T>`: A clean way to avoid `NullPointerExceptions`
- E.g.

```
public class MemoryCell {
   private Optional<String> value = Optional.empty();
   public Optional<String> getValue() { return value; }
   public void setValue(String value) {
      this.value = Optional.ofNullable(value);
   }
   public static void main(String[] args) {
      var c = new MemoryCell();
      c.setValue("testing");
      // c.setValue(null);
      System.out.println("Value: " + c.getValue().orElse("n/a"));
   }
}
```

Demo: MemoryCellProject

Matching with Optional

- Looking for an element in a stream, e.g.

```
Optional<Dish> dish = menu.stream()
                          .filter(Dish::isVegetarian)
                          .findAny();

Optional<Dish> dish = menu.stream()
                          .filter(Dish::isVegetarian)
                          .findFirst();
```

Why do they return Optional objects?

Demo:
More Stream operations

`StreamDishes.java`

Demo task

- Is there a vegetarian dish with calories under 150?
- If so, print the name of the vegetarian dish under 150 calories, otherwise print “Not found”.

```
var namesOfLightVegDishes = Dish.menu.stream()
                                    .filter(Dish::isVegetarian)
                                    .filter(d -> d.calories() < 150)
                                    .map(Dish::name)
                                    .findAny();

System.out.println("vegetarian dish with calories < 150:");
if (namesOfLightVegDishes.isPresent())
    System.out.println(namesOfLightVegDishes.get());
else
    System.out.println("Not found.");

// Alternatively
System.out.println(namesOfLightVegDishes.orElse("Not found."));
```

Reducing (1)

- So far, we have only seen simple, pre-defined terminal operations
  - E.g. `allMatch, forEach, findAny, collect`
- We can also express more complicated “reduction queries”, e.g.
  - Calculate the sum of the calories of all dishes on the menu
  - What is the highest-calorie dish on the menu? ...

Reducing (2)

- `T reduce(T id, BinaryOperator<T> acc)`
- It takes two parameters:
  - An initial value (`id`), e.g. 0, 1, or ""
  - An accumulator function (`acc`) to combine two stream elements and produce a new value
- E.g. Find the sum of a list of numbers:

```
var numbers = List.of(3, 4, 5, 1, 2);
var total = numbers.stream().reduce(0, (a, b) -> Integer.sum(a, b));
```

- Or use a method reference to be more concise:

```
var total = numbers.stream().reduce(0, Integer::sum));
```

Reducing (3)

- E.g. Print the average number of calories in the menu.

```
var total = Dish.menu.stream()
                        .map(Dish::calories)
                        .reduce(0, Integer::sum);
System.out.println("Average calories: " +
                       String.format("%.2f", 1.0*total/Dish.menu.size()));
```

- E.g. Print a short menu, i.e. a list of dish names separated by “, ”

```
var shortMenu = Dish.menu.stream()
                            .map(Dish::name)
                            .reduce("", (a, b) -> a + b + ", ");
System.out.println(shortMenu);  // beef, chicken, …,
```

Stateless vs. stateful

- Knowing whether operations are stateful or not is important in terms of correctness and performance (e.g. to parallelise algorithms)
- Stateless operations
  - Operations like `map` and `filter` take each element from the input stream and produce zero or one result in the output stream
  - These operations have no internal state, i.e. stateless
- Stateful operations
  - Operations like `reduce` need to have internal state to accumulate the result This internal state is bounded in size.
  - However, for operations like `sort` and `distinct`, it is necessary to hold *all* the elements of the stream, hence the storage requirement for these operation is unbounded

Creating infinite streams

- Since streams are lazily evaluated, we can create infinite ones
- E.g. using the `iterate` and `generate` methods of `Stream`

```
Stream.iterate(0, n -> n + 2)
      .limit(15)
      .forEach(System.out::println);
```

```
Stream.generate(() -> Math.random())
      .limit(10)
      .forEach(System.out::println);
```

Operation `iterate` takes an initial value and a lambda of type UnaryOperator<T>, while operation `generate` takes only a lambda of type Supplier<T>.

Flattening streams (1)

- Task: Get a list of all unique characters from a list of words.
- E.g. A list containing 2 words:

```
["Hello", "World"]
```

- We want to find all unique characters:

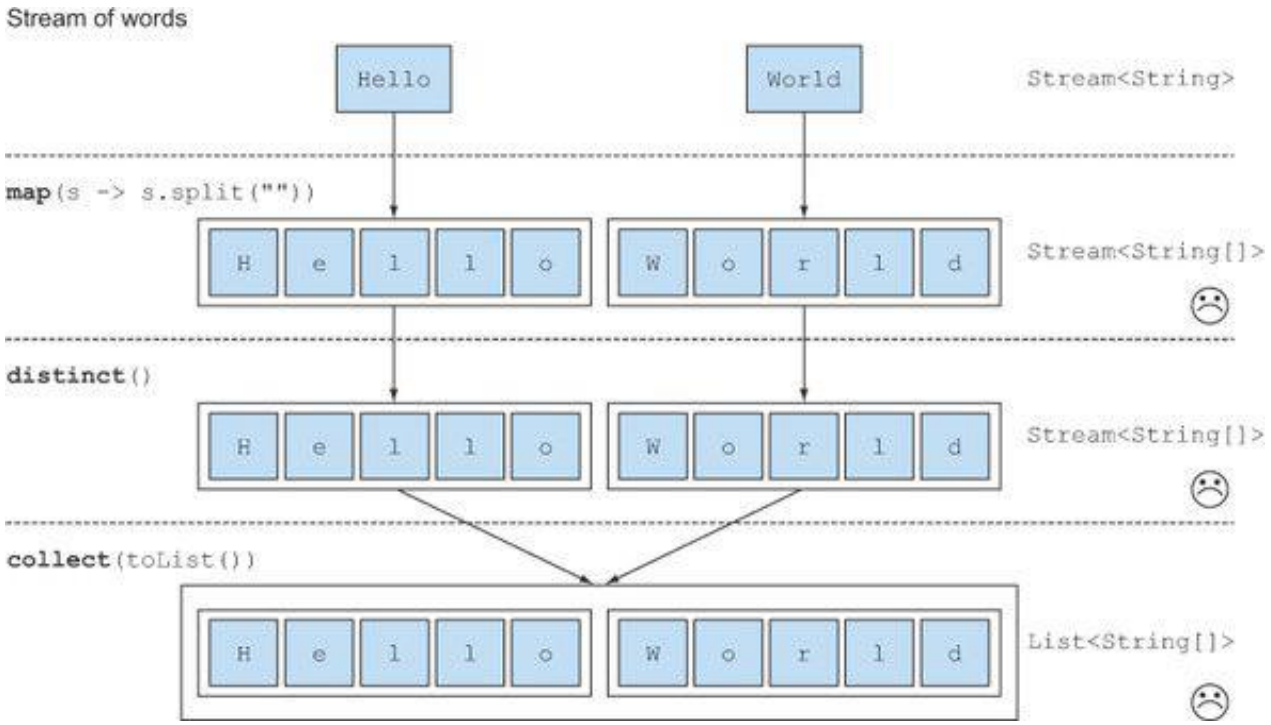
```
["H", "e", "l", "o", "W", "r", "d"]
```

Flattening streams (2)

- Using map:

```
var words = List.of("Hello", "World");

words.stream()
     .map(w -> w.split(""))
     .distinct()
     .toList();
```



Map returns a stream consisting of the results of applying the given function to the elements of this stream

`Stream<R> map(Function<T,R> mapper)`

Flattening streams (3)

- We have to use `flatMap` instead:

```
words.stream()
     .flatMap(w -> Stream.of(w.split("")))
     .distinct()
     .toList();

     // ["H", "e", "l", "o", "W", "r", "d"]
```

FlatMap returns a stream consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function

```
Stream<R> flatMap(Function<T,Stream<R>> mapper)
```

- flatMap applies a “one-to-many transformation” to the elements of the stream, and then flattening the resulting elements into a new stream

Stream reduction using collect()

- public R collect(Collector<T, A, R> collector)
  - Performs a mutable reduction operation on the elements of a stream
  - Input: a Collector
  - Output: some result of type R
- Interface Collector<T, A, R>
  - T - the type of the input elements
  - A - the intermediate accumulation type of the reduction operation (often hidden as an implementation detail)
  - R - the type of the result

Stream reduction using static methods of Collectors

- A few typical examples of collecting/reducing streams:

```
var numberOfDishes = menu.stream().collect(counting());   // or menu.stream().count();

var averageCalories = menu.stream().collect(averagingInt(Dish::calories));

var shortMenuCommaSeparated = menu.stream().map(Dish::name).collect(joining(", "));
```

Note: methods in red are the static methods of the Collectors class

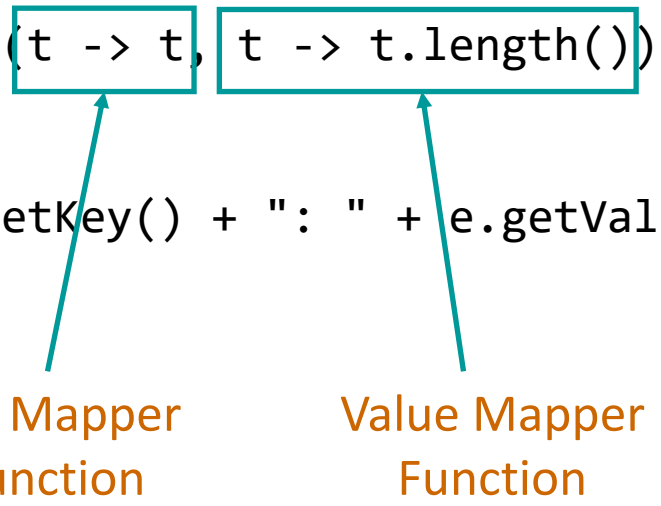
List to a Map (1)

- toMap is a static method of Collectors class; it returns a Collector
- E.g.

```
var fruitList = Arrays.asList("Apple", "Banana", "Pear", "Orange");

Map<String, Integer> fruitMap = fruitList.stream()
                                         .collect(toMap(t -> t, t -> t.length()));

fruitMap.entrySet().stream()
                   .forEach(e -> System.out.println(e.getKey() + ": " + e.getValue()));
```



Output:

```
Apple: 5
Pear: 4
Orange: 6
Banana: 6
```

List to a Map (2)

- toMap method can take an extra parameter, a merge function, which is useful where there are duplicates
- E.g. Find the number of occurrence of each character in the list

```
var chars = Arrays.asList('a','b','c','a','c','d','a');
Map<Character, Integer> count = chars.stream()
                                     .collect(toMap(k -> k, v -> 1, Integer::sum));

// count: {a=3, b=1, c=2, d=1}
```

Key Mapper Function

Value Mapper Function

Merger Function

groupingBy (1)

- A predefined collector in `java.util.stream.Collectors`
- It allows us to group items of a collection by a given key (similar to SQL's GROUP BY clause), and store the results in a Map instance
- E.g. Grouping dishes by type:

```
Map<Dish.Type, List<Dish>> groupedDishes = menu.stream()
                                               .collect(groupingBy(Dish::type));

groupedDishes.forEach((k, v) -> {
    System.out.println(k + ": ");
    v.forEach(dish -> System.out.println("  " + dish.name()));
});
```

Output:

```
FISH:
  prawns
  salmon
MEAT:
  beef
  …
```

groupingBy (2)

- Grouping dish names by type:

```
menu.stream()
    .collect(groupingBy(Dish::type, mapping(Dish::name, toList())));
```

It returns a Map<Dish.Type, List<String>>

- Counting dishes in per type:

```
menu.stream ()
    .collect(groupingBy(Dish::type, counting()));
```

This returns a Map<Dish.Type, Long>

```
// {FISH=2, MEAT=3, OTHER=4}
```

Reducing/Combining

- E.g. get the total calories in each type:

```
Map<Dish.Type, Integer> totalCaloriesByType =
                menu.stream().collect(groupingBy(Dish::type,
                                      reducing(0, Dish::calories, Integer::sum)));
```

- `reducing` is a static method of Collectors. It may take 3 parameters:
  - Identity: like the reduce operation, the identity element is both the initial value of the reduction and the default result if there are no elements in the stream
  - Mapper: the reducing operation applies this mapper function to all stream elements
  - Operation: the operation function is used to reduce the mapped values

collect vs. reduce

- `collect` and `reduce` often produce the same result
- But they differ in crucial ways
  - `reduce` combines two values and produce a new one (*immutable* reduction)
  - `collect` is designed to mutate a container to accumulate the result it produces
  - Using `reduce` with a mutable accumulator will cause problems when we work on parallel streams later

Partitioning streams

- Special case of grouping: partition a stream into two according to a predicate (or classification function)

```
Map<Boolean, List<Dish>> partitionByVegeterian =
                                menu.stream()
                                    .collect(partitioningBy(Dish::isVegetarian));
```

- What is the main advantage compared to filtering the stream?

Finding/Defining your collector

- There are lot more pre-defined collectors, see JavaDoc
- You can also create your own ones by implementing the Collector interface (*out-of-scope for this course*)