Collections Aggregates

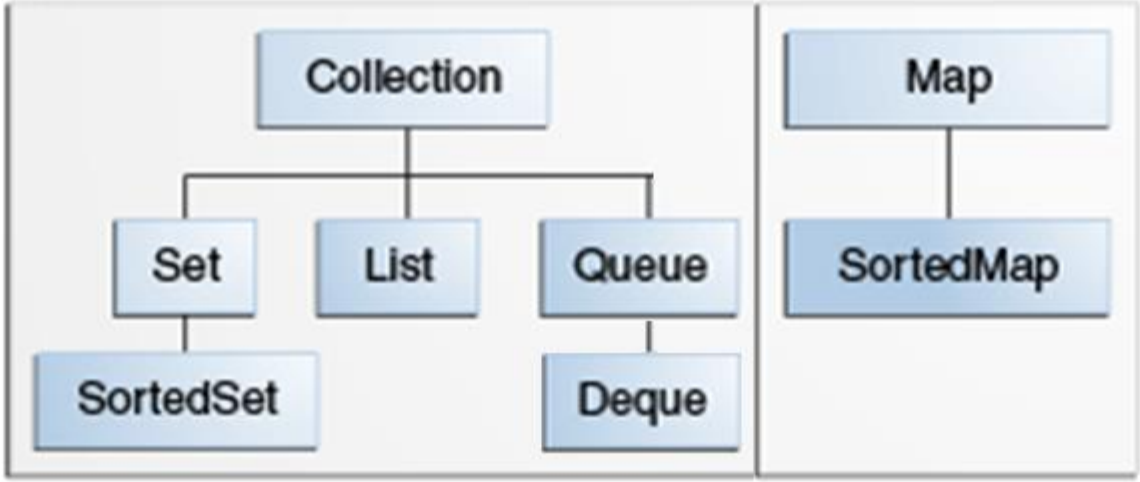
Paul Fodor

CSE260, Computer Science B: Honors

Stony Brook University

http://www.cs.stonybrook.edu/~cse260

Recap: Java Collections Framework



Interface	Hash Table	Resizable Array	Balanced Tree	Linked List	Hash Table + Linked List
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Deque		ArrayDeque		LinkedList	
Map	HashMap		TreeMap		LinkedHashMap

Traversing Collections

- There are multiple ways to traverse collections:
  - (1) by using Iterators
  - (2) with the for-each construct
  - (3) **using aggregate operations** (since JDK 1.8): obtain a stream and perform aggregate operations on it
    - Aggregate operations are often used in conjunction with lambda expressions to make programming more expressive, using less lines of code
    - The following code sequentially iterates through a collection of shapes and prints out the red objects:

```
myShapesCollection.stream()
.filter(e -> e.getColor() == Color.RED)
.forEach(e -> System.out.println(e.getName()));
```

  - (4) only lists can be traversed using indices

Traversing Collections using aggregate operations/streams

- Suppose that you are creating a social networking application:

```
public class Person {
    String name;
    Date birthday;
    Sex gender;
    String emailAddress;
    int age;
    public String getName() {
        ...
    }
    ...
}
```

- Print the name of all members contained in the collection roster with a for-each loop:

```
ArrayList<Person> roster = new ArrayList();
roster.stream()
    .forEach(e -> System.out.println(e.getName());
```

Complete program:

```
import java.util.Date;
public class Person {
	public enum Sex {
		MALE, FEMALE
	}
	String name;
	Date birthday;
	Sex gender;
	String emailAddress;
	int age;

	public Person(String name, Sex gender) {
		this.name = name;
		this.gender = gender;
	}

	public String getName() {
		return name;
	}

	public int getAge() {
		return age;
	}

	public Sex getGender() {
		return gender;
	}
}
```

```java
import java.util.ArrayList;
import java.util.List;

public class TestAggregates1 {

    public static void main(String[] args) {
        List<Person> roster = new ArrayList<>();
        roster.add(new Person("Abe", Person.Sex.MALE));
        roster.add(new Person("Barbara", Person.Sex.FEMALE));
        roster.add(new Person("Chris", Person.Sex.MALE));
        roster.add(new Person("Dorothy", Person.Sex.FEMALE));
        roster.add(new Person("Eugene", Person.Sex.MALE));
        roster.add(new Person("Fabian", Person.Sex.MALE));

        roster.stream()
        .forEach(e -> System.out.println(e.getName()));

        roster.stream()
        .filter(e -> e.getGender() == Person.Sex.MALE)
        .forEach(e -> System.out.println(e.getName()));

    }
}
```

More examples

- Sum the salaries of all employees in a company:

```
int total = employees.stream()
.collect(Collectors.summingInt(Employee::getSalary)));
```

  - stream() is optional, you can apply the aggregate directly on the collection

- Convert the elements of a Collection to String objects, then join them, separated by commas:

```
String joined = elements.stream()
.map(Object::toString)
.collect(Collectors.joining(", "));
```

- A parallel stream (which might make sense if the collection is large enough and your computer has enough cores):

```
myShapesCollection.parallelStream()
.filter(e -> e.getColor() == Color.RED)
.forEach(e -> System.out.println(e.getName()));
```

Pipeline

- A *pipeline* is a sequence of aggregate operations
  - For example: print the male members contained in the collection roster with a pipeline that consists of the aggregate operations **filter** and **forEach**:

```
roster.stream()
  .filter(e -> e.getGender() == Person.Sex.MALE)
  .forEach(e -> System.out.println(e.getName()));
```

is similar with the for-each loop:

```
for (Person p : roster) {
    if (p.getGender() == Person.Sex.MALE) {
        System.out.println(p.getName());
    }
}
```

Source, Intermediate and Terminal Operations

- A ***pipeline*** contains the following components:
  - A ***source***: this could be a collection, an array, a generator function, or an I/O channel.
  - Zero or more ***intermediate*** operations, such as **`filter`**, that produces a **new stream**
    - A stream is a sequence of elements, but unlike a collection, it is not a data structure that stores elements. Instead, a stream carries values from a source through a pipeline.
  - A ***terminal*** operation that produces a non-stream result, such as: a primitive value (like a double value), a collection, or in the case of **`forEach`**, no value at all.
    - the parameter of a **`forEach`** operation is the lambda expression **`e->System.out.println(e.getName())`**, which invokes the method **`getName`** on the object **`e`**. (The Java runtime and compiler infer that the type of the object **`e`** is **`Person`**.)

`mapToInt` *and Method references*

- Calculate the average age of all **`male`** members contained in the collection roster with a pipeline that consists of the aggregate operations **`filter`**, **`mapToInt`**, and **`average`**:

```
double average = roster.stream()
      .filter(p -> p.getGender() == Person.Sex.MALE)
      .mapToInt(Person::getAge)
      .average()
      .getAsDouble();
```

- The **`mapToInt`** operation returns a new stream of type **`IntStream`** (which is a stream that contains only integer values).
  - The operation applies the function specified in its parameter to each element in a particular stream
  - The function **`Person::getAge`**, is a ***method reference*** that returns the age of the member

`mapToInt` *and Method references*

- Alternatively, we could use the lambda expression **`e -> e.getAge()`**

```
double average = roster.stream()
        .filter(p -> p.getGender() == Person.Sex.MALE)
        .mapToInt(e -> e.getAge())
        .average()
        .getAsDouble();
```

Reduction operations

- The JDK contains many ***terminal*** operations such as average that return one value by combining the contents of a stream
  - These operations are called ***reduction operations*** (more: **`sum`**, **`min`**, **`max`** and **`count`**)

```
double average = roster.stream()
      .filter(p -> p.getGender() == Person.Sex.MALE)
      .mapToInt(Person::getAge)
      .average()
      .getAsDouble();
```

  - The **`average`** operation calculates the average value of the elements contained in a stream of type **`IntStream`**.
  - It returns an object of type **`OptionalDouble`**.
  - If the stream contains no elements, then the **`average`** operation returns an empty instance of **`OptionalDouble`**, and invoking the method **`getAsDouble`** throws a **`NoSuchElementException`**

Differences Between Aggregate Operations and Iterators

- ***Aggregate operations*** do not contain a method like **`next`** to instruct them to process the next element of the collection
- Aggregation can more easily take advantage of parallel computing, which involves dividing a problem into subproblems, solving those problems simultaneously, and then combining the results of the solutions to the subproblems
- Aggregate operations process elements from a stream, not directly from a collection. Consequently, they are also called ***stream operations***.
- Aggregates support behavior as parameters: we can specify lambda expressions as parameters for most aggregate operations

General-purpose reduction operations reduce and collect

- The JDK provides us with the general-purpose reduction operations reduce and collect: **`Stream.reduce`**

```
Integer totalAgeReduce = roster.stream()
    .map(Person::getAge)
    .reduce(
        0,
        (a, b) -> a + b);
```

similar to:

```
Integer totalAge = roster.stream()
     .mapToInt(Person::getAge)
     .sum();
```

General-purpose reduction operations reduce and collect

- The **`Stream.collect`** modifies an existing stream:
  - Consider how to find the average of values in a stream
    - We require two pieces of data: the total number of values and the sum of those values
    - We can create a new data type that contains member variables that keep track of the total number of values and the sum of those values:

```
class Averager implements IntConsumer{
    private int total = 0;
    private int count = 0;
    public double average() {
        return count > 0 ? ((double) total)/count : 0;
    }
    public void accept(int i) { total += i; count++; }
    public void combine(Averager other) {
        total += other.total;
        count += other.count;
    }
}
```

General-purpose reduction operations reduce and collect

- The following pipeline uses the **`Averager`** class and the **`collect`** method to calculate the average age of all male members:

```
Averager averageCollect = roster.stream()
        .filter(p -> p.getGender() == Person.Sex.MALE)
        .map(Person::getAge)
        .collect(Averager::new, Averager::accept,
             Averager::combine);
System.out.println("Average age of male members: " +
        averageCollect.average());
```

- We can use the **`collect`** operations with parallel streams
  - the **`collect`** method with a parallel stream creates a new thread whenever the combiner function creates a new object, such as an **`Averager`** object in this example
  - Consequently, we do not have to worry about synchronization

General-purpose reduction operations reduce and collect

- The **`collect`** operation in the example takes three arguments:
  - ***supplier***: is a factory function: it constructs new instances of the result container
    - In the example, it is a new instance of the **`Averager`** class
  - ***accumulator***: function that incorporates a stream element into a result container
    - In the example, it modifies the **`Averager`** result container by incrementing the **`count`** variable by one and adding to the **`total`** member variable the value of the stream element, which is an integer representing the age of a male member
  - ***combiner***: function that takes two result containers and merges their contents
    - In the example, it modifies an **`Averager`** result container by incrementing the **`count`** variable by the **`count`** member variable of the other **`Averager`** instance and adding to the **`total`** member variable the value of the other **`Averager`** instance's **`total`** member variable

General-purpose reduction operations reduce and collect

- The **`collect`** operation is best suited for getting collections:
  - The following example puts the names of the male members in a collection with the **`collect`** operation:

```
List<String> namesOfMaleMembersCollect = roster.stream()
    .filter(p -> p.getGender() == Person.Sex.MALE)
    .map(p -> p.getName())
    .collect(Collectors.toList());
```

- This version of the **`collect`** operation takes one parameter of type **`Collector`**
  - The **`Collectors`** class contains many useful reduction operations, such as accumulating elements into collections and summarizing elements according to various criteria
  - **`Collectors.toList`** operation accumulates the stream elements into a new instance of **`List`**

groupingBy

- Group members of the collection roster by gender:

```
Map<Person.Sex, List<Person>> byGender =
  roster.stream()
   .collect(Collectors.groupingBy(Person::getGender));
```

- The **groupingBy** operation returns a map whose keys are the values that result from applying the lambda expression specified as its parameter (which is called a classification function).
  - In this example, the returned map contains two keys, **Person.Sex.MALE** and **Person.Sex.FEMALE**
  - The keys' corresponding values are instances of **List** that contain the stream elements that, when processed by the classification function, correspond to the key value

groupingBy

- Retrieve the names of each member in the collection roster and group them by gender:

```
Map<Person.Sex, List<String>> namesByGender =
 roster.stream()
  .collect(Collectors.groupingBy(
                Person::getGender,
                Collectors.mapping(
                    Person::getName,
                    Collectors.toList())));
```

- The **`groupingBy`** operation in this example takes two parameters, a classification function and an instance of **`Collector`** that applies the collector mapping, which applies the mapping function **`Person::getName`** to each element of the stream

groupingBy

- Retrieve the total age of members of each gender:

```
Map<Person.Sex, Integer> totalAgeByGender =
 roster.stream()
  .collect(Collectors.groupingBy(
                Person::getGender,
                Collectors.reducing(
                    0,
                    Person::getAge,
                    Integer::sum)));
```

- The **`groupingBy`** operation in this example takes three parameters
  - ***identity***, like the **`Stream.reduce`** operation, is both the initial value of the reduction and the default result if there are no elements in the stream.
  - ***mapper***: reducing operation that applies this mapper function to all stream elements
  - ***operation*** function used to reduce the mapped values