Java 8

Part Two

Method References

Streams

METHOD REFERENCES

Lambdas

- We know we can use a lambda anytime that an object of a functional interface is expected.

```
Collections.sort(myWords, myComparatorObject);
        OR
Collections.sort(myWords,
        (s1, s2) -> s1.compareToIgnoreCase(s2)  );

myButton.setOnAction( event -> System.out.println(event) );
```

- In these examples, our lambda contains a single line that is a call to an existing method (compareToIgnoreCase or println)

Method References

- If a method already exists that matches what we need for our lambda, we can pass a *method reference* instead of creating a lambda.
- You cannot send any parameters to a method reference.

Type	Syntax
Static method	Class::staticMethodName
Instance method	object::methodName
Instance method	Class::methodName
Constructor	Class::new

Example: A Static Method

We want to sort a list of numbers with a comparator.

Original Syntax:

```
Collections.sort(numberList, new MyIntegerComparator());
```

Lambda Syntax:

MyIntegerComparator is a functional interface with one method:

```
int compareTo(Integer num1, Integer num2)
```

So we can replace this method with a lambda:

```
Collection.sort(numberList, (num1, num2) -> Integer.compare(num1, num2));
```

Example: A Static Method

Lambda Syntax:

```
Collection.sort(numberList, (num1, num2) -> Integer.compare(num1, num2));
```

Method Reference Syntax:

The body of the lambda is a single method invocation that has the same parameters and return type as our lambda: two Integer parameters, return type int

So we can replace the whole lambda with a method reference:

```
Collections.sort(numbers, Integer::compare);
```

Instance Method (Through the Class)

Example: we want to sort a list alphabetically, ignoring case.

Original syntax:

```
Collections.sort(myWords, new IgnoreCaseStringComparator());
```

functional interface with one method: int compareTo(String s1, String s2)

Lambda syntax:

```
Collections.sort(words, (s1,s2)->s1.compareToIgnoreCase(s2));
```

Method Reference syntax:

```
Collections.sort(words, String::compareToIgnoreCase);
```

- The first parameter of the lambda is the invoking object

Instance Method (Through an Object)

Example: print the event object to the console when the button is clicked.

Lambda syntax:

```
button.setOnAction(event -> System.out.println(event));
```

Method reference syntax:

```
button.setOnAction(System.out::println);
```

Instance Method (Through an Object)

Example: invoke your own method on the click of a button.

Lambda syntax:

```
button.setOnAction(event -> handleButtonClick(event));
```

Method reference syntax:

```
button.setOnAction(this::handleButtonClick);
```

Instance Method (Through an Object)

An example that uses Streams (return to this later!)

```
wordStream.forEach(s -> System.out.println(s) );
```

or

```
wordStream.forEach(System.out::println)
```

Constructor

- Can invoke similar to a static method, but instead of the method name, use “new”
- Example from streams (return to this later!)

```
List<String> buttonLabels = List.of(“B1”, “B2”, “B3”);


Stream<Button> buttonStream = labels.stream.map(
        s -> new Button(s)
);
        OR
Stream<Button> buttonStream = labels.stream.map(Button::new);
```

Using Method References

- If your code requires an object of a functional interface…
  - You can use a lambda instead.

- If your lambda body has just a single method call…
  - You can use a method reference instead.

COMPARATOR

The Comparator Interface

- Allows you to specify an ordering of two objects.
- The ordering can be different from a class's natural ordering (which is defined by implementing the Comparable interface and the compareTo method).
- Allows you to have many different ways to order objects.

The Comparator Interface

- Create a class that implements the interface.
  - This is often a private static class.
- Implement the compare method.

```
private static class MyComparator implements Comparator<MyClass> {
   public int compare(MyClass o1, MyClass o2) {
      // return negative if o1 < o2
      // return positive if o1 > 02
      // return 0 otherwise
   }
}
```

The Comparator Interface

- Create an object of that class to use.
  - This is often a private static class.
- Use that object to sort.

```
public final static Comparator<MyClass> MY_CLASS_COMPARATOR
   = new MyComparator();

Collections.sort(myCollection, MY_CLASS_COMPARATOR)
Arrays.sort(myArray, MY_CLASS_COMPARATOR)
```

Practice

- Review the User comparator example.

Methods in Comparator

- The Comparator interface provides many helpful static methods:
  - comparing(…) *(static)*
  - thenComparing(…)
  - comparingInt(…) *(static)*
  - thenComparingInt(…)

- These methods take in a parameter of type Function.
- They all return an object of type Comparator.

The Function Interface

- Function is a functional interface with one method:
    - interface: Function<T, R>
    - method: public R apply(T)
- Objects take in a parameter of type T and return an object of type R.
    - T and R can be the same or different.

The Function Interface

- An example of a Function would be something that maps from a type to some comparable key for that type.
  - Example: User -> String (to compare Users by email)
  - Example: User -> LocalDate (to compare Users by joinDate)
  - Example: User -> Integer (to compare Users by a numeric id)

- Essentially we are going from type T to type Comparable
  - T is the object we want to sort
  - R is some class that implements Comparable- this is the type of the key we want to sort by (often a String, Integer, etc.)
  - T -> Comparable

Using a Method Reference

- Back to the methods in Comparator:
  - Comparator.comparing(Function)
  - The parameter is type Function
- Remember:
  - If your code requires an object of a functional interface… You can use a lambda instead.
  - If your lambda body has just a single method call… You can use a method reference instead.

Using a Method Reference

- Comparator.comparing(Function)
- Remember:
  - If your code requires an object of a functional interface… You can use a lambda instead.
    - Function is a functional interface. So we can use a lambda!
    - user -> user.getEmail()
    - T is User, R is String
  - If your lambda body has just a single method call… You can use a method reference instead.

Using a Method Reference

- Comparator.comparing(Function)
- Remember:
  - If your code requires an object of a functional interface… You can use a lambda instead.
    - Function is a functional interface. So we can use a lambda!
    - user -> user.getEmail()
    - T is User, R is String
  - If your lambda body has just a single method call… You can use a method reference instead.
    - That lambda has a single method call. So we can use a method reference!
    - User::getEmail

Using a Method Reference

- Comparator.comparing(Function)
  - Comparator.comparing(User::getEmail)
- Remember:
  - If your code requires an object of a functional interface… You can use a lambda instead.
    - Function is a functional interface. So we can use a lambda!
    - user -> user.getEmail()
    - T is User, R is String
  - If your lambda body has just a single method call… You can use a method reference instead.
    - That lambda has a single method call. So we can use a method reference!
    - User::getEmail

Examples

-
  ```
  Collections.sort(userList,
      Comparator.comparing(User::getJoinDate));
  ```
-
  ```
  Arrays.sort(userArray, Comparator.comparing(User::getEmail));
  ```

Methods in Comparator

- We can also use thenComparing to specify a second (or third or fourth or…) characteristic to compare on!
  - thenComparing is invoked on the Comparator returned from comparing

-
  ```
  Collections.sort(userList,
      Comparator.comparing(User::getLastName)
      .thenComparing(User::getFirstName));
  ```

Methods in Comparator

- We can also specify a second Comparator (using a lambda or method reference) to specify how to sort the keys!

-
  ```
  Collections.sort(userList,
        Comparator.comparing(User::getLastName, String::compareToIgnoreCase));
  ```

-
  ```
  Collections.sort(userList,
        Comparator.comparing(User::getEmail, (e1, e2)-> e1.compareToIgnoreCase(e2));
  ```

Methods in Comparator

- If the key you are sorting on is an int, use the comparingInt and thenComparingInt methods.
    - Also comparingDouble and thenComparingDouble

-
  ```
  Collections.sort(employeeList;
      Comparator.comparingInt(Employee::getYearsWorked)
      .thenComparingInt(Employee::getYearsUntilRetirement));
  ```

-
  ```
  Arrays.sort(userArray, Comparator.comparingInt(
      user -> user.getEmail().length()));
  ```

Other Methods

- Comparator.reverseOrder() returns a Comparator that orders based on the reverse of the natural ordering
  - Example: `Arrays.sort(numberArray, Comparator.reverseOrder());`
- myComparatorObject.reversed() returns a Comparator that is the reverse of the invoking object
  - Example:
    ```
    Collections.sort(userList,
                Comparator.comparing(User::getEmail).reversed());
    ```

Practice

- Update the Comparators in the User example with these methods.

STREAMS

Some Functional Interfaces

Functional Interface	Parameter Types	Return Type
Supplier<T>	None	T
Consumer<T>	T	void
Predicate<T>	T	boolean
Function<T, R>	T	R
BiFunction<T, U, R>	T, U	R
UnaryOperator<T>	T	T
BinaryOperator<T>	T, T	T

Streams

- Allow you to process collections
- You specify ***what you want to do***
  - Instead of *how to do it*
  - The *how* is left to the stream library to optimize

From Iteration to Stream

- The traditional way to process collections was to iterate over them and apply some process to each element.
- This works, but doesn’t allow any optimization behind the scenes.
- And it’s overly prescriptive in how a task must be accomplished.

- Example:
  - If you want to find the total of numbers in a list, it doesn’t really matter that you iterate over the list in order.
  - But when we use iteration, we specify the order as part of the operation.
  - We don’t need to do this!

A Simple Example

- Count the number of times a target value appears in a list of random numbers.
  - Use iteration
  - Then use a stream.
- Specify *what not how*.
  - Specify: What do we need from the collection?
  - Ignore: How will this be done? (e.g., in which order, in which thread)

Using Streams

1. Create the stream.
2. Transform the stream.
    - Also called: *intermediate operations.*
    - Stream-producing
    - Always lazy
3. Produce a result.
    - Also called: *terminal operations.*
    - Value or side-effect producing
    - Forces the execution of lazy operations that precede it.
    - Terminates the stream- it can no longer be used.
        - Not caught at compile time- a runtime exception (IllegalStateException)

A Stream is Not a Collection

- A stream does not store its elements.
  - They are stored in an underlying collection or generated on demand.
- Stream operations do not mutate the source.
  - Instead they return *new streams* with a result.
- Stream operations are *lazy*.
  - Operations are not executed until a result is needed.
  - Example: If you ask for the first five matches, the filter method stops once it finds the fifth match!

Stream Terminology

- Pipelines: A stream pipeline consists of a source, zero or more intermediate operations, and a terminal operation.
- Stateless vs stateful (more to come on this)
- Non-interference: The data source is not modified during the execution of a stream pipeline.
  - But note the lazy factor!
  - This means you can modify as long as it is before the terminal operation.

CREATING STREAMS

Using Streams

1. Create the stream.
2. Transform the stream.
    - Also called: *intermediate operations.*
    - Stream-producing
    - Always lazy
3. Produce a result.
    - Also called: *terminal operations.*
    - Value or side-effect producing
    - Forces the execution of lazy operations that precede it.
    - Terminates the stream- it can no longer be used.
        - Not caught at compile time- a runtime exception (IllegalStateException)

Stream Quick Reference

Creating	Transformation (Intermediate)	Terminal

stream()
parallelStream()
Stream.of(array)
Stream.of(…)
Stream.generate(Supplier)
Stream.iterate(UnaryOperator)
	filter(Predicate)
	map(Function)
	mapToInt(IntFunction)
	limit(int)
	skip(int)
	sorted()
	sorted(Comparator)
	distinct()
		count()
		forEach(Consumer)
		collect(Collectors.toList()) toMap()
		collect(Collectors.joining(“delim”))
		toArray()
		summarizingInt()
		anyMatch(Predicate), allMatch, noneMatch
		findFirst()
		reduce()

Creating Streams

- For any Collection object:
  - invoke `stream()`
  - invoke `parallelStream()`
- For arrays:
  - invoke `Stream.of(myArray)`
  - invoke `Arrays.stream(array, from, to)`

Infinite Streams

- Because streams are lazy, you can have infinite streams.
  - Example: The stream is infinitely long, but if you are only asking for the first 100 elements, that’s okay, because the processing will stop after those 100 elements are found.
- You *can* get intro trouble if invoke methods that ask for something to be done on all elements of an infinite stream.
  - It’s your job to avoid this- Java won’t stop you!

Creating Infinite Streams

- `Stream.generate(Supplier<T> supplier)`
  - Supplier: no parameter, returns T
  - Example:

```
Stream<Double> randoms = Stream.generate(
     () -> Math.random()    );
```

or

```
Stream<Double> randoms = Stream.generate(Math::random);
```

method reference to a static method
(the method random takes no
parameters and returns T, so
it fits in place of the lambda)

Creating Infinite Streams

- `Stream.iterate(T seed, UnaryOperator<T> f)`
    - UnaryOperator: T parameter, returns T
    - The lambda is repeatedly applied to each successive value
    - Example:

```
Stream<Integer> counters = Stream.iterate(
     5, n -> n+2;  );
// generates the Stream 5, 7, 9, 11, 13
```

Standard Library Methods

- Split a String or CharSequence by a regular expression:
  - `Stream<String> words = Pattern.compile(“[\\P{L}]+”).splitAsStream(myString);`
  - This would split the string on all non-letters
- Split a file into lines:
  - `Stream<String> lines = Files.lines(path);`
  - This can go inside a try-with-resources block, which will close the file appropriately

Intermediate and Terminal Operations Needed For Testing

- limit(20)

- count()
- forEach(System.out::println);

Practice

- Create an infinite stream of random integers.
- Create an infinite stream of the odd, positive numbers.
- Create a stream of dictionary words from a file.
- Create a stream of eviction objects.
- Create a stream of job objects.

Parallel Streams

- Streams allow Java to parallelize bulk operations. To do this:

1. Create a parallel stream.
    - `parallelStream()`
    - `parallel()` // e.g., `Stream.of(myArray).parallel()`
2. Ensure that operations:
    - are stateless and
    - can be executed in arbitrary order.
3. Ensure that functions passed are *threadsafe*.

TRANSFORMING STREAMS

Using Streams

1. Create the stream.
2. Transform the stream.
   - Also called: *intermediate operations.*
   - Stream-producing
   - Always lazy
3. Produce a result.
   - Also called: *terminal operations.*
   - Value or side-effect producing
   - Forces the execution of lazy operations that precede it.
   - Terminates the stream- it can no longer be used.
     - Not caught at compile time- a runtime exception (IllegalStateException)

Stream Quick Reference

Creating	Transformation (Intermediate)	Terminal

stream()
parallelStream()
Stream.of(array)
Stream.of(…)
Stream.generate(Supplier)
Stream.iterate(UnaryOperator)
	filter(Predicate)
	map(Function)
	mapToInt(IntFunction)
	limit(int)
	skip(int)
	sorted()
	sorted(Comparator)
	distinct()
		count()
		forEach(Consumer)
		collect(Collectors.toList()) toMap()
		collect(Collectors.joining(“delim”))
		collect(Collectors.groupingBy(…)
		toArray()
		summarizingInt()
		anyMatch(Predicate), allMatch, noneMatch
		findFirst()
		reduce()

Transforming Streams

- Reads data from a stream and puts the transformed data into another stream
- You can pipeline together multiple transformations!

Filter

- Creates a new stream with all elements that match a condition
- The condition is determined by a predicate
    - Predicate<T>: T parameter, returns boolean
- Syntax:

```
    stream.filter( Predicate<T> predicate);
    stream.filter( tObject -> boolean return; );
```

Practice

- Create an infinite stream of random integers.
  - Filter only even random numbers
- Create a stream of dictionary words from a file.
  - Filter only two-letter words
- Create a stream of eviction objects.
  - Filter only evictions due to nuisance, in the Tenderloin, on Market Street.
- Create a stream of job objects.
  - Filter only jobs in the Mayor’s office.

Map

- Transforms/changes the values in a stream
  - Apply a function to every value and put the new values in a new stream
- Function determined by the Function interface:
  - Function<T,R>: T parameter, returns an R
  - T and R can be the same or be different
- Syntax:

```
    stream.map(Function<T, R> function);
    stream.map( myT -> returns myR );
```

Map

- `//myWords is a Stream<String>`

- Example map using same type (String, String)

```
Stream<String> upperWords = myWords.map(s -> s.toUpperCase() );
```

or

```
Stream<String> upperWords = myWords.map(String::toUpperCase);
```

- Example map using different types (String, Character)

```
Stream<Character> firstCharStream = myWords.map(s ->  s.charAt(0) );
```

Practice

- Create an infinite stream of random integers.
- Create a stream of dictionary words from a file.
  - Map the words onto a stream in all upper case
  - Map the words onto a stream of characters of their last letter
  - Filter all words that contain an x or z and map them onto upper case
- Create a stream of eviction objects.
  - Map the eviction objects onto a Stream of Strings that contain their neighborhoods. Then filter out only the Richmond neighborhood.
- Create a stream of job objects.
  - Filter all jobs that have total compensation > 100,000 and then map their job title into a String with spaces removed and only eight chars long.

Extracting and Combining Substreams

- `stream.limit(n)` returns the first n elements of a stream
  - Makes infinite streams useful!
- `stream.skip(n)` skips the first n elements of a stream

- These methods could be *pipelined* together!

- `Stream.concat(stream1, stream2)` combines two streams
  - Make sure the first isn’t infinite!

Practice

- Concatenate the first 10 random numbers with odd numbers in positions 20-25.
  - Why? Why not!

Stateless Transformations

- When an element is retrieved from the transformed stream, it does not depend on the previous elements.
- Examples:
  - filter
  - map
  - limit
  - skip
  - concat

Stateful Transformations

- When an element is retrieved from the transformed stream, it **does** depend on the previous elements.
- Examples:
  - `distinct` // suppresses duplicates
  - `sorted` // sorts the stream (note: does not sort the collection!)
  - `sorted(Comparator)`

Practice

- Create an infinite stream of random integers.
  - Create a stream of the unique random numbers.
- Create an infinite stream of the odd, positive numbers.
- Create a stream of dictionary words from a file.
- Create a stream of eviction objects.
- Create a stream of job objects.
  - Sort the stream of jobs.
  - Re-sort by overtime.

peek

- The peek method can be very helpful in testing/debugging!
- peek returns another stream with the same elements but is invoked every time an element is retrieved.
- peek takes a Consumer<T> object
    - Consumer<T> has one method: parameter T, void return

```
peek( Consumer<T>)

peek(System.out::println)

peek(s -> System.out.println(“in part X” + s))
```

Streams of Primitive Types

- `IntStream`, `LongStream`, `DoubleStream`
- Creating
  - `IntStream stream = Arrays.stream(numArray, from, to);`
  - static generate and iterate methods
    - `IntStream.generate(IntSupplier)`   `IntStream.iterate(int seed, IntUnaryOperator)`
  - static range method creates a stream with the specific range
    - `IntStream zeroToNine = IntStream.range(0,10);`
    - `IntStream zeroToTen = IntStream.rangeClosed(0,10);`
  - Create from a Random object
    - `randomGenerator.ints()`
  - Convert from object stream
    - `mapToInt(ToIntFunction),mapToLong(ToLongFunction), mapToDouble(ToDoubleFunction)`

Primitive Streams (vs Streams)

- `toArray` returns a primitive array
- `OptionalInt`, `OptionalLong`, `OptionalDouble` type
  - methods `getAsInt`, `getAsLong`, `getAsDouble`

- methods `sum`, `average`, `max`, and `min`

- `boxed()` converts to `Stream<Integer>`
  - Use this if you want to invoke collect or other stream-only methods

`MapToInt` (Double/Long)

- Transforms/changes the values in a stream into an IntStream
- Syntax:

```
stream.mapToInt(ToIntFunction<T> function);
```

  - Function<T>: T parameter, returns an int
- Example:
  - `IntStream lengthStream = myWords.mapToInt(s -> s.length() );`

  or

  - `IntStream lengthStream = myWords.mapToInt(String::length);`

Practice

- Create an infinite stream of random integers.
  - Limit to the 100 numbers and find the max and min.
- Create a stream of dictionary words from a file.
  - Map the words onto a stream to represent how many vowels; find the average.
  - Map the words onto a stream to represent how many “z”s; find the total number of z’s
- Create a stream of eviction objects.
- Create a stream of job objects.
  - Map the jobs onto their overtime amount and sum up all the overtime

PRODUCING A RESULT

Using Streams

1. Create the stream.
2. Transform the stream.
   - Also called: *intermediate operations.*
   - Stream-producing
   - Always lazy
3. Produce a result.
   - Also called: *terminal operations.*
   - Value or side-effect producing
   - Forces the execution of lazy operations that precede it.
   - Terminates the stream- it can no longer be used.
     - Not caught at compile time- a runtime exception (IllegalStateException)

Stream Quick Reference

Creating	Transformation (Intermediate)	Terminal

stream()
parallelStream()
Stream.of(array)
Stream.of(…)
Stream.generate(Supplier)
Stream.iterate(UnaryOperator)
	filter(Predicate)
	map(Function)
	mapToInt(IntFunction)
	limit(int)
	skip(int)
	sorted()
	sorted(Comparator)
	distinct()
		count()
		forEach(Consumer)
		collect(Collectors.toList()) toMap()
		collect(Collectors.joining(“delim”))
		collect(Collectors.groupingBy(…)
		toArray()
		summarizingInt()
		anyMatch(Predicate), allMatch, noneMatch
		findFirst()
		reduce()

forEach

- Allows you to access each element
- Takes a Consumer<T> parameter
  - Consumer<T>: T parameter, void body

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

or

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

- Elements could be processed in arbitrary order.
- `forEachOrdered` ensures elements are processed in order.

Laziness

- Let’s more closely examine laziness using forEach.
- What is printed?

```
String[] words = {"apple", "ball", "banana"};
Stream.of(words)
   .map(s -> {
         System.out.println("in the map with " + s);
         return s.toUpperCase(); } )
   .forEach(s -> System.out.println("in the forEach with " + s));
```

Simple Reductions

- `long count()`
- `Optional<T> min(Comparator)`
- `Optional<T> max(Comparator)`
- `Optional<T> findFirst()`
- `Optional<T> findAny()`
- `boolean anyMatch(Predicate<T>)`
- `boolean allMatch(Predicate<T>)`
- `boolean noneMatch(Predicate<T>)`

Optional<T>

- A wrapper for an object of type T or for no object
- The preferred alternative to null when working with streams

```
    Optional<T> optionalValue = …;
    optionalValue.get()
```

    // either returns the wrapper element or throws a NoSuchElementException

- There is an `isPresent()` method that checks if the value is present…

Treating Optional the Same as Null…

```
T value = …
if(value != null)
     value.method();

Optional<T> optionalValue = …
if(optionalValue.isPresent())
      optioanlValue.get().method();
```

- How is this any better?? It’s not…

Optional: Consuming the Value

- One way to use optional values is to specify what should be done only if the value is present by using the `ifPresent(Consumer<T>)` method.
  - Consumer: T parameter, void return

```
Optional<T> optionalValue = …

optionalValue.ifPresent( v -> v.method() );
```

- If a value is present, the method is invoked.
- If no value is present, nothing happens.

Optional: Produce an Alternative

- Another way to use optional values is to provide an alternative when the value is not present with the `orElse(T)` or `orElseGet(Supplier<T> s)` methods.
  - Supplier: no parameter, returns T

```
Optional<String> optionalVal = …
String result = optionalVal.orElse(“default text”);
String result = optionalVal.orElseGet(()->return str);
```

Optional: Throwing an Exception

- You can also throw an exception if no value exists with the `orElseThrow(Supplier<T extends Exception)`
  - Supplier: no parameter, returns T- but T must extend Exception

```
Optional<String> optionalVal = …
String expectedResult =
optionalVal.orElseThrow(new IllegalArgumentException());
```

or

```
optionalVal.orElseThrow(IllegalArgumentException::new);
```

Practice

- Create an infinite stream of random integers.
    - Find the smallest of the first 100 numbers.
- Create an infinite stream of the odd, positive numbers.
- Create a stream of dictionary words from a file.
    - Count the two-letter words.
    - Find a word with the largest number of z's.
    - Determine if there are any words with the letter combination "qi."
    - Find a word with a q but no u. Then with a q, no u, and an x.

Practice

- Create a stream of eviction objects.
  - Count how many evictions due to nuisance in the Tenderloid on Market.
  - Determine if there were any evictions on Phelan.
- Create a stream of job objects.
  - Find the jobs with the min and max total compensation amounts.

Collecting Results

- Allows you to look at the results as a collection

- `iterator()`
- `toArray() // returns an Object[]`
- `toArray(T[]::new) // returns a T[]`
- `collect(Collectors.toList())`
- `collect(Collectors.toSet())`

Collecting Results to a Map

-
  ```
  stream.collect(Collectors.toMap(
     Function<T,K>  keyFunction,
     Function<T,V> valueFunction)
     );
  ```
  - Function: T parameter, returns K/V (the key or the value)
  - To return the actual element (usually as a value), use Function.identity()

Practice

- Create a stream of eviction objects.
  - Collect a list of all evictions in the tenderloin.
  - Collect nuisance-caused eviction objects to a map, keyed by id.
- Create a stream of job objects.
  - Collect all jobs with overtime into a list.
  - Collect all job’s in the mayor’s office to a map, keyed by their id.

Grouping

- Often, you want to group results together by some characteristic.
- Use `Collectors.groupingBy(Function<V,K> grouper)` to create a `Map(<K>, List<V>)` object
  - The function takes parameter of type V and returns a key of type T- this is the key used to group the elements together

```
Map<Key,Value> map =
stream.collect(Collectors.groupingBy(
    valueObject -> return grouping key
);
```

Grouping

- Use `Collectors.groupingBy(Function<V,K> grouper, Collector collector)` to create a `Map(<K>,Int/Long/Double)`
  - static methods that return Collector objects:
    - counting()
    - averagingInt(ToIntFunction<T>)
    - summingInt(ToIntFunction<T>)
    - maxBy(Comparator) and minBy(Comparator)

```
Map<Key, Long> map = stream.collect(Collectors.groupingBy(
         value -> return key,
         Collectors.counting()
      ));
```

Practice

- Create a stream of eviction objects.
  - Collect a map with a list of all evictions for each neighborhood.
  - Collect a map with the number of evictions for each neighborhood.
- Create a stream of job objects.
  - Collect a map with a list of jobs for each department.
  - Collect a map with the sum of the salaries of all jobs in each department.

Reduce

- You can reduce a stream to a single result.
- Three methods:
  - reduce(BinaryOperator<T> accumulator)
    - returns Optional<T>
  - reduce(T identity, BinaryOperator<T> accumulator)
  - reduce(T identity, BiFunction<U, T, U> accumulator, BinaryOperator<U> combiner>
    - this third version can often be written more clearly with a map and reduce
- BinaryOperator<T>: parameter T and T, return T
- BiFunction<T,U,R>: parameter T and U, return R
- The accumulator must be an associative function.

Practice

- Create an infinite stream of random integers.
  - Find the sum of the first 100 integers.
- Create an infinite stream of the odd, positive numbers.
- Create a stream of dictionary words from a file.
  - Find and print the “largest” word.
- Create a stream of eviction objects.
- Create a stream of job objects.
  - Sum up all benefit pay. First use the three-parameter reduce, then use a map and reduce.

String Results

-
  ```
  String result =
  stream.collect(Collectors.joining())
  ```

-
  ```
  String result =
  stream.collect(Collectors.joining(“,”));
  ```

-
  ```
  String result = stream
      .map(Object::toString)
      .collect(Collectors.joining());
  ```

Practice

- Create an infinite stream of random integers.
- Create an infinite stream of the odd, positive numbers.
- Create a stream of dictionary words from a file.
  - Print a comma-separated list of all q without u words.
- Create a stream of eviction objects.
- Create a stream of job objects.

IntSummaryStatistics Results (Long and Double)

- Use Collectors.summarizingInt(ToIntFunction<T> to create an object of type IntSummaryStatistics
  - Then invoke methods getAverage, getMin, getMax, getSum

```
IntSummaryStatistics summary = stream.
    collect(Collectors.summarizingInt(
        str -> str.length()
            );
```

`IntSummaryStatistics` Results (Long and Double)

```
IntSummaryStatistics summary = stream.
    collect(Collectors.summarizingInt(
        str -> str.length()));

double average = summary.getAverage();
int max = summary.getMax();
int min = summary.getMin();
long sum = summary.getSum();
```

Practice

- Create an infinite stream of random integers.
- Create an infinite stream of the odd, positive numbers.
- Create a stream of dictionary words from a file.
- Create a stream of eviction objects.
- Create a stream of job objects.
    - Find all statistics about overtime pay.

“Reusing” Streams

- Streams cannot be reused once there is a terminal operation.
- Also be careful that intermediate operations do not actually transform the stream, but create a *whole new stream.*
- You can create a *stream supplier* and then invoke get() to obtain a new stream.
- Syntax:

```
Supplier<Stream<String>> streamSupplier = () ->
     myList.stream()
     .filter(…)
     .map(…);
Stream<String> stream1 = streamSupplier.get();
Stream<String> stream2 = streamSupplier.get();
```

Some Functional Interfaces

Functional Interface	Parameter Types	Return Type
Supplier<T>	None	T
Consumer<T>	T	void
Predicate<T>	T	boolean
Function<T, R>	T	R
BiFunction<T, U, R>	T, U	R
UnaryOperator<T>	T	T
BinaryOperator<T>	T, T	T

Stream Quick Reference

Creating	Intermediate	Terminal

stream()
parallelStream()
Stream.of(array)
Stream.of(…)
Stream.generate(Supplier)
Stream.iterate(UnaryOperator)
	filter(Predicate)
	map(Function)
	mapToInt(IntFunction)
	limit(int)
	skip(int)
	sorted(),
	sorted(Comparator)
		count()
		forEach(Consumer)
		collect(Collectors.toList()) toMap()
		collect(Collectors.joining(“delim”))
		collect(Collectors.groupingBy(…)
		toArray()
		summarizingInt()
		anyMatch(Predicate), allMatch, noneMatch
		findFirst()
		reduce()