Stream API
and
Lambda Expression

Basics Revisited

- Interfaces
- Collections Framework
- Anonymous Inner Classes

Interfaces

"Ordinary" interfaces

Marker interfaces (e.g. `Serializable` or `Runnable`)

Functional interfaces (annotated with `@FunctionalInterface`)

`@interface` classes (Annotations)

New in Java 8:

`static` methods

`default` methods

Default methods[1]

"A default method is a method that is declared in an interface with the `default` modifier; its body is always represented by a block. It provides a default implementation for any class that implements the interface without overriding the method. Default methods are distinct from concrete methods (§8.4.3.1), which are declared in classes."

[1][Gosling 2015, p. 288]

Collections Framework

Two things to remember:

1. There are Lists, Sets, and Maps:

```
List<T>:        ArrayList<T>  or  LinkedList<T>
Set<T>:         TreeSet<T>  or  HashSet<T>
Map<K, V>:      TreeMap<K, V>  or  HashMap<K, V>
```

2. Use “loosly coupled” references:

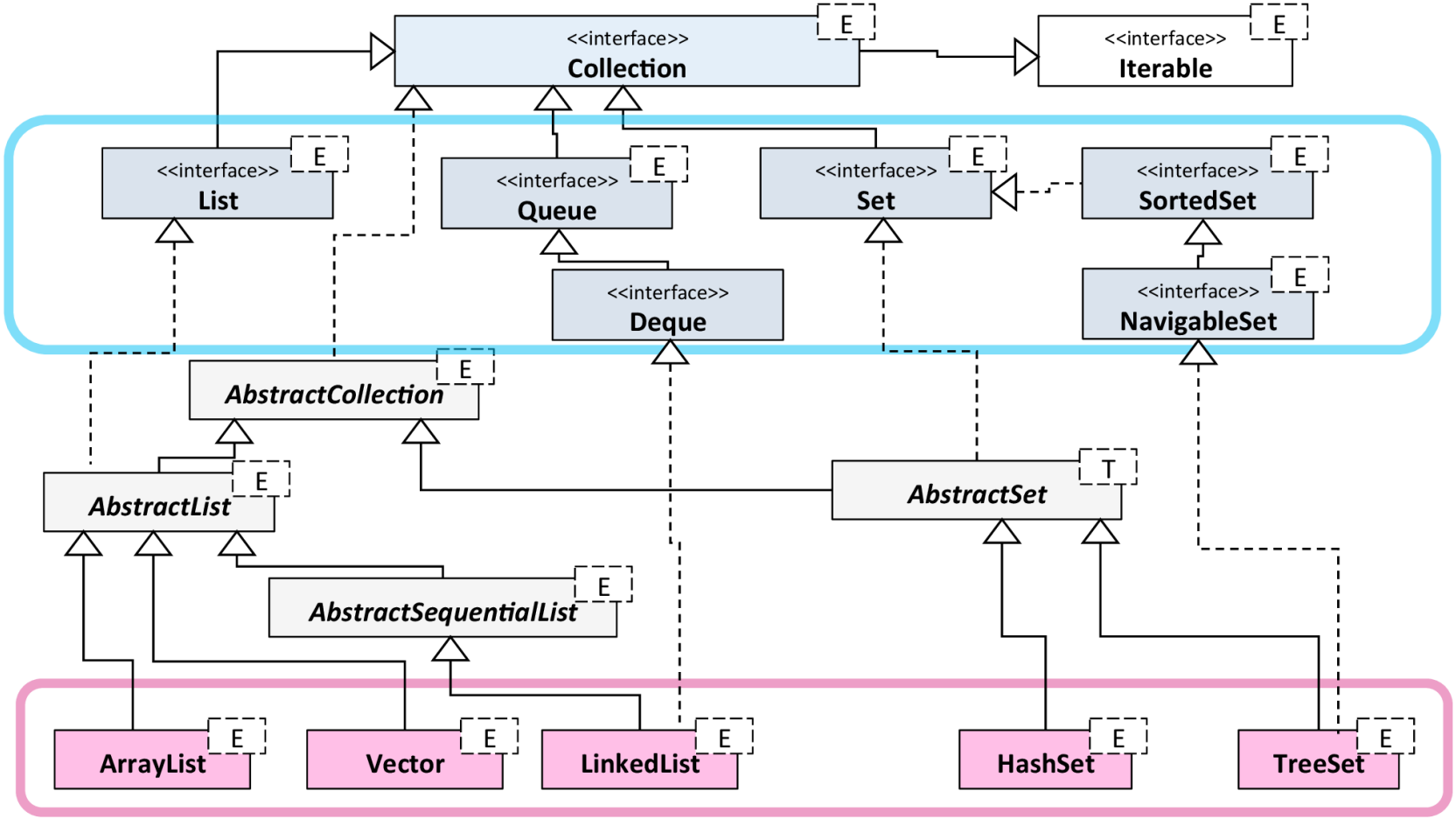
```
List<String> = new ArrayList<>();
```

Collections Framework[1]

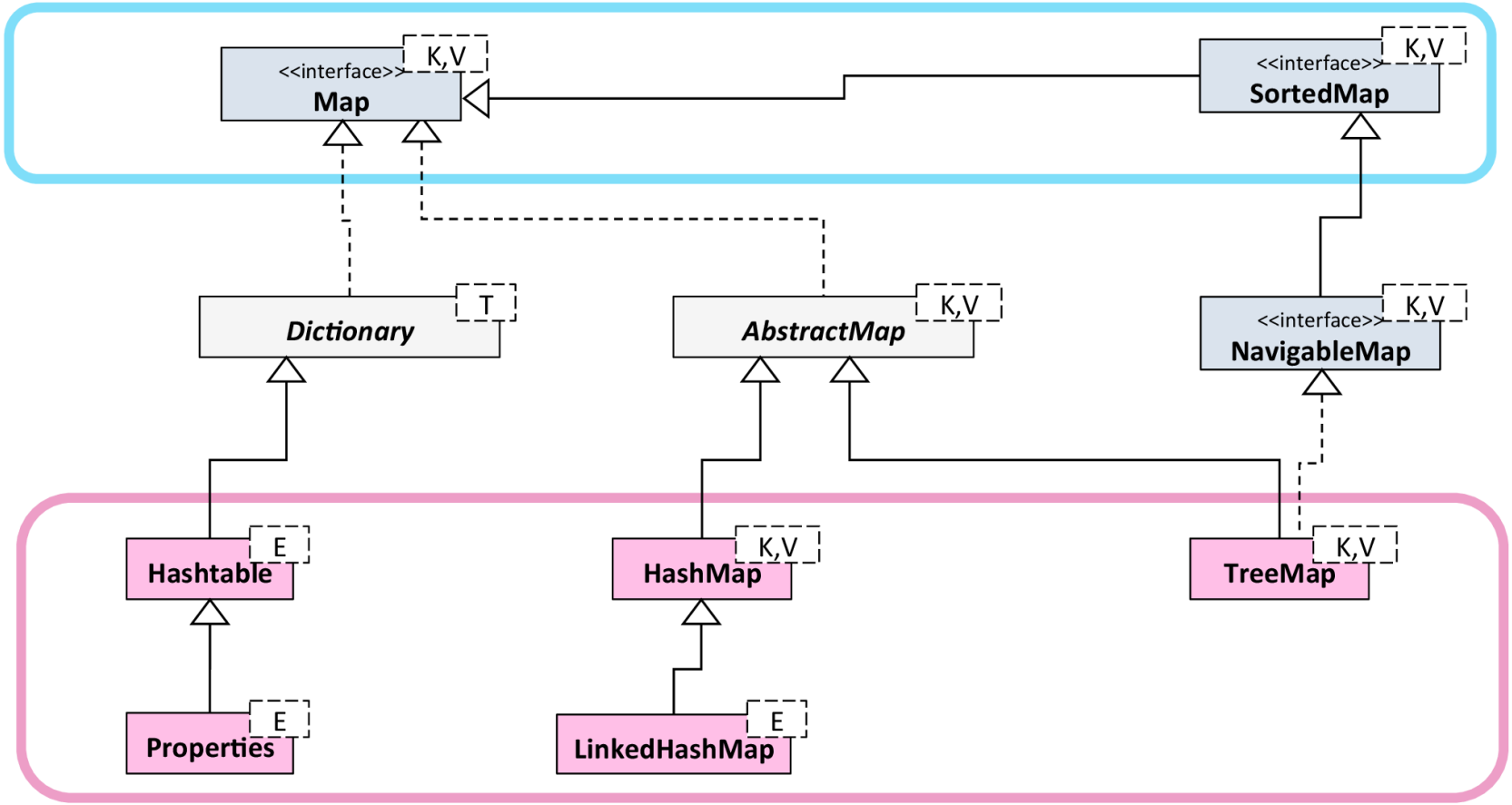
“The collections framework is a unified architecture for representing and manipulating collections, enabling them to be manipulated independently of the details of their representation. It reduces programming effort while increasing performance. […]”

[1][Oracle Corp. 2016]

Collection interfaces (abridged)



Map interfaces (abridged)



Utilities in class Collections

```
public static void reverse(List<?> list)

public static <E> Collection<E> checkedCollection(Collection<E> c,
                                                  Class<E> type)

public static <T> List<T> nCopies(int n, T o)

public static int frequency(Collection<?> c, Object o)

public static void shuffle(List<?> list)

public static void rotate(List<?> list, int distance)

public static void reverse(List<?> list)

public static <T> boolean replaceAll(List<T> list, T oldVal, T newVal)

public static <T> T min(Collection<? extends T> coll,
                        Comparator<? super T> comp)

// others:

Arrays.asList(Object... o)

Arrays.stream(T[] array)

Stream.of(T[] array)
```

Anonymous Inner Classes

```
public interface Comparator<T> { int compare (T obj1, T obj2); }
```

```
public class Collections {
   public static <T> void sort(List<T> l, Comparator<? super T> c)
   {...}
}
```

```
List<String> names = Arrays.asList("John", "Andrew", "Eve");

Collections.sort(names, new Comparator<String>() {
   @Override public int compare(String s1, String s2) {
      return Integer.compare(s1.length(), s2.length());
   }
});

for(String name : names) System.out.print(name + " ");
// Eve John Andrew
```

Streams

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

Streams

- are not data structure
- do not contain storage for data
- are “pipelines” for streams of data (i.e. of objects)
- while in the pipeline data undergo transformation (without changing the data structure holding it)
- wrap collections (lists, set, maps)
- read data from it
- work on copies

What is this... “functional” programming?

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

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 do data get into streams?

Streams are mainly generated based on collections:

```
List<String> names = Arrays.asList("John", "George", "Sue");


Stream<String> stream1 = names.stream();

Stream<String> stream2 = Stream.of("Tom", "Rita", "Mae");

Stream<String> stream3;
stream2 = Arrays.stream( new String[]{"Lisa", "Max", "Anna"} );
```

Or with builder pattern:

```
Stream<String> stream4 = Stream.<String>builder()
                              .add("Mike")
                              .add("Sandra").build();
```

How do data get out of streams?

- The Streaming API provides so called “finalizing” methods (i.e. methods that do not return stream objects)

```
forEach
toArray
collect
reduce
min
max
count
anyMatch
noneMatch
findFirst
findAny
```

Streaming example

"Take all names from the stream that start with the letter "J", map the names into capital letters, skip one, and collect them into a new set"

```
List<String> names = Stream.of("John","George","Joe","Sue","James");


Stream<String> stream1 = names.stream();

______ = stream1.filter( ___________ )
                .map( ___________ )
                .skip( ___________ )
                .collect( ___________ );
```

```
Set<String> result = Stream.of("John", "George", "Joe", "Sue", "James")
        .filter(name -> name.startsWith("J"))
        // Filter names starting with "J"
        .map(String::toUpperCase) // Convert names to uppercase
        .skip(1) // Skip one name
        .collect(Collectors.toSet()); // Collect into a Set


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

Lambda Expressions and Functional Interfaces

- Lambdas
- Functional Interfaces

Lambdas or Closures

"Lambda" = "closure" = record storing a function (functionality, method) and its environment (but without a class or method name)

Roughly: anonymous method

Lambdas represent source code – not data and not object state!

Syntax:

```
( parameter list ) -> { expression(s) }
```

Examples:

```
(int x, int y) -> { return x + y; }
```

```
(long x) -> { return x * 2; }
```

```
() -> { String msg = "Lambda"; System.out.println(msg); }
```

For details on "functional programming" cf. [Huges 1984] or [Turner 2013]

Lambdas and functional interfaces

Functional interfaces are so called *SAM* types (single abstract method)

A functional interface has exactly one abstract method, e.g. `Runnable`, `Comparator<T>`, or `Comparable<T>`

Functional interfaces can define 0..* default methods and 0..* static methods

Using the `@FunctionalInterface` annotation on classes the compiler is required to generate an error message if it is no interface and it doesn't define exactly one SAM.

```
@FunctionalInterface
public interface Counter<T> {
    int count(T obj);
}
```

```
Counter<String> strCount = (String s) -> { return s.length(); };
```

[https://docs.oracle.com/javase/8/docs/api/java/lang/FunctionalInterface.html]

Your first lambda with Comparator<T>

```
@FunctionalInterface
public interface Comparator<T> { int compare(T obj1, T obj2); }
```

```
public class Collections {
   public static <T> void sort(List<T> l, Comparator<? super T> c)
   {...}
}
```

Your task:
Prepare your first lambda expression
to compare two String objects by length

```
List<String> names = Arrays.asList("John", "Andrew", "Eve");

Collections.sort(names,   ????   );

for(String name : names) System.out.print(name + " ");
// Eve John Andrew
```

Functional interfaces in JDK

Selection of most used interfaces from package `java.util.function`:

```
// Computes a single input with no result
public interface Consumer<T>  {
    void accept(T t);
    default Consumer<T> andThen(Consumer<? super T> after) {...}
}

// Represents a supplier of results.
interface Supplier<T> {
   T get();
}

// Computes a single output, prodeces output of different type
interface Function<T,R>  {
   R apply(T t);
   default <V> Function<T,V> andThen(Function<? super R,? extends V> after)  {...}
   default <V> Function<V,R> compose(Function<? super V,? extends T> before) {...}
}

// Represents a predicate (boolean-valued function) of one argument
interface Predicate<T> {
   boolean test(T t);
   default Predicate<T> and(Predicate<? super T> other) {...}
   default Predicate<T> negate() {...}
   // ...
}
```

BiConsumer<T,U>
BiFunction<T,U,R>
BinaryOperator<T>
BiPredicate<T,U>
BooleanSupplier
Consumer<T>
DoubleBinaryOperator
DoubleConsumer
DoubleFunction<R>
DoublePredicate
DoubleSupplier
DoubleToIntFunction
DoubleToLongFunction
DoubleUnaryOperator
Function<T,R>
IntBinaryOperator
IntConsumer
IntFunction<R>
IntPredicate
IntSupplier
IntToDoubleFunction
IntToLongFunction
IntUnaryOperator
LongBinaryOperator
LongConsumer
LongFunction<R>
LongPredicate
LongSupplier
LongToDoubleFunction
LongToIntFunction
LongUnaryOperator
ObjDoubleConsumer<T>
ObjIntConsumer<T>
ObjLongConsumer<T>
Predicate<T>
Supplier<T>
ToDoubleBiFunction<T,U>
ToDoubleFunction<T>
ToIntBiFunction<T,U>
ToIntFunction<T>
ToLongBiFunction<T,U>
ToLongFunction<T>
UnaryOperator<T>

[https://docs.oracle.com/javase/8/docs/api/java/util/function/package-amountmary.html]

Type inference

Lambda expressions allow for minimal syntax if compiler can deduct type information (so called *type inference*), e.g.:

```
@FunctionalInterface
public interface BiFunction<T, U, R> {
    R apply(T t, U u);
}
```

R = Integer could be inferred from return type of expression t.length() + u.length() → Integer (e.g. when used directly in typed method call)

```
BiFunction<String, Object, Integer> bf;
bf = (String txt, Object obj) -> { return t.length() + u.hashCode() ); }
// can be even shorter:
bf = (txt, obj) -> t.length() + u.hashCode(); // se below
```

compiler can infer types from declaration of f

Further syntax shortening examples:

for single return statements keyword return together with {}-pair can be dropped

```
(int x, int y) -> { return x * y; }  // shorter:   (x, y) -> x * y
(long x) -> { return x * 2; }        // shortest:  x -> x * 2
```

()-pair can be dropped with only one parameter

Lambda as parameters and return types

Further examples for type inference using `Comparator<T>` as functional interface:

```
List<String> names = Arrays.asList("Ahab", "George", "Sue");
Collections.sort(names, (s1, s2) -> s1.length() - s2.length() );
```

*T :: **String** can be deducted from names being a List<**String**>*

Signature of Collections::sort method is:
*public static <**T**> void sort(List<**T**> list, Comparator<? super **T**> c)*

```
Collections.sort(names, createComparator() );
```

```
public Comparator<String> createComparator() {
    return (s1, s2) -> s1.length() - s2.length();
}
```

Statement returned here is of functional interface type Comparator<String>

Method references (“function pointers”)

Syntax: `Classname::methodName` `objectReferenceName::methodName`

Lambdas can be replaced by method references whenever there would not further actions within the the lambda

Examples:

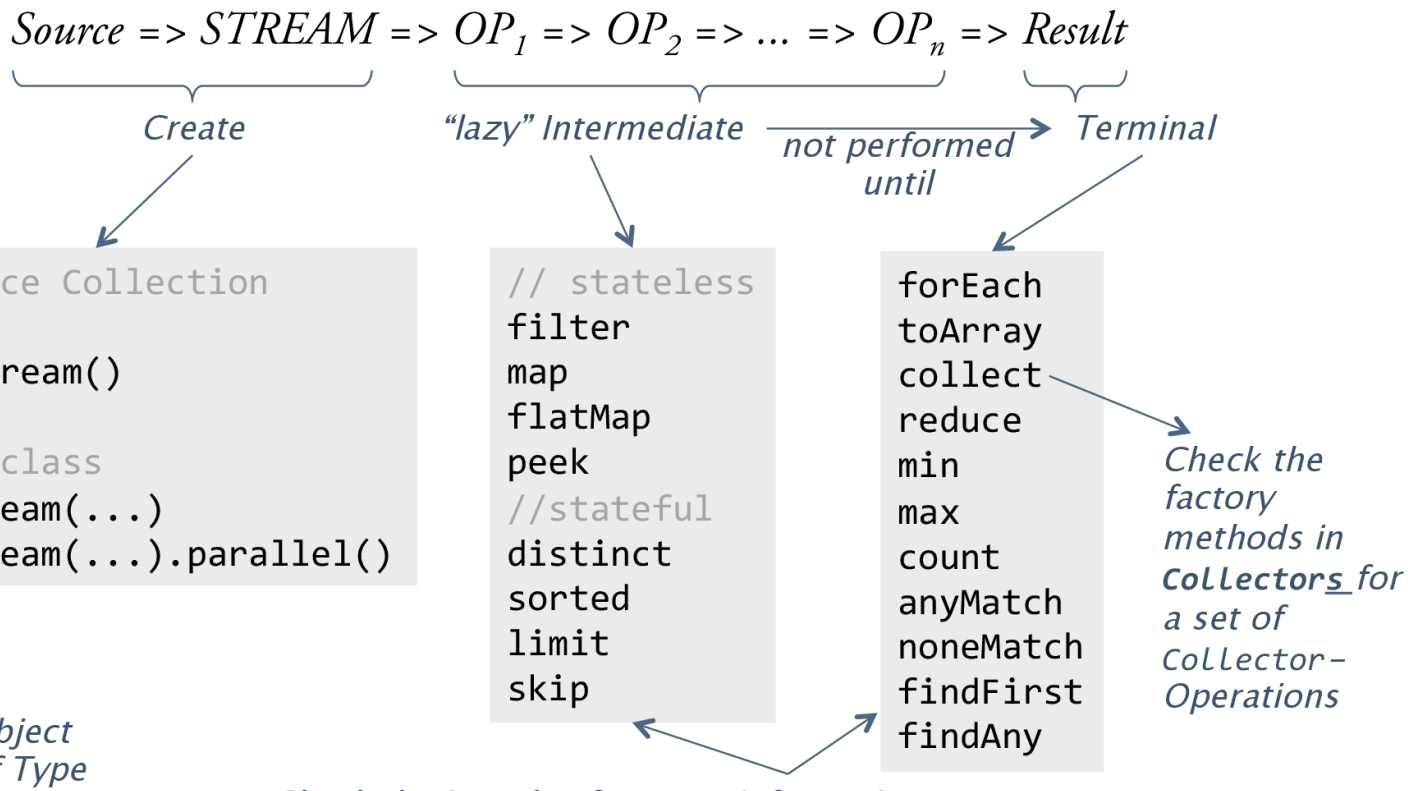
Reference	Method reference…	…replacing Lambda
static method	`String::valueOf`	`obj -> String.valueOf(obj)`
instance method (via class)	`String::compareTo`	`(s1, s2) -> s1.comapreTo(s2)`
instance method (via object ref)	`person::getName`	`() -> person.getName()`
Constructor	`ArrayList::new`	`() -> new ArrayList<>()`

[Table taken from Inden 2015, p. 812]

Streaming API

- Creating Streams
- Fluently working with streams
- Finalize Streams

Stream operations



[Inden 2015, p. 825]

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!
- ```
  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

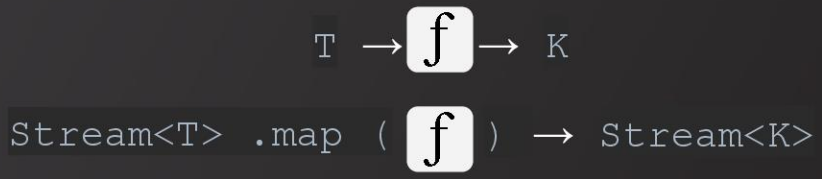
Gets elements out of the stream once we're done (terminal operation)

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(`  `)` and `Collectors.reducing(`  `);`

map

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



- Example:

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

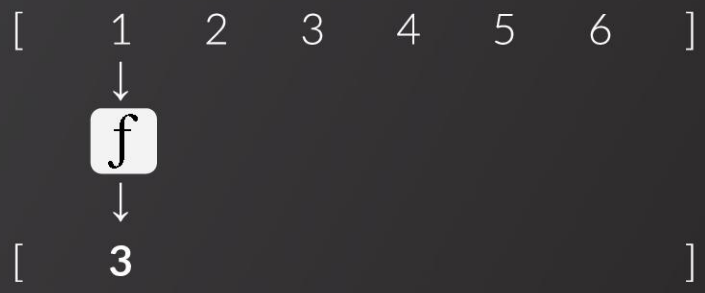
map

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

[1 2 3 4 5 6]

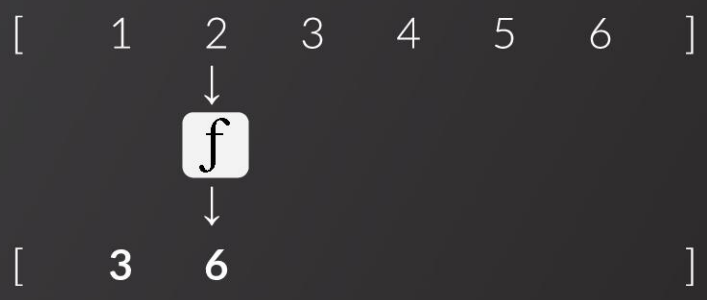
map

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



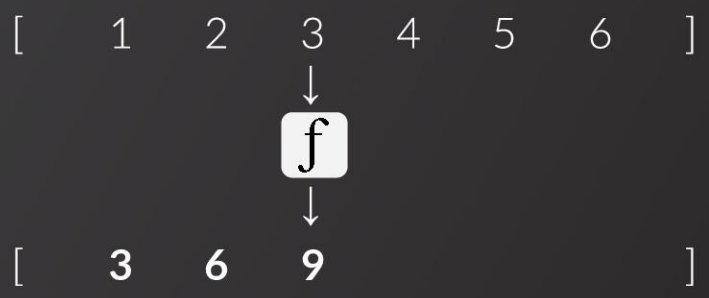
map

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



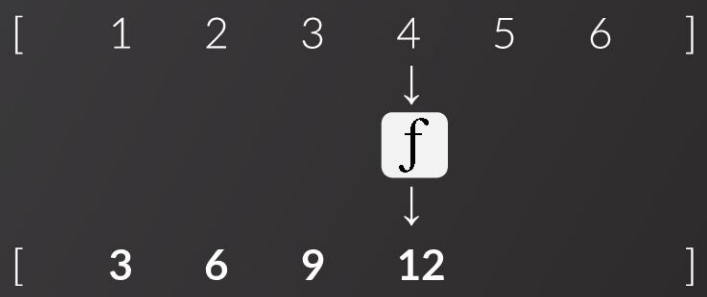
map

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



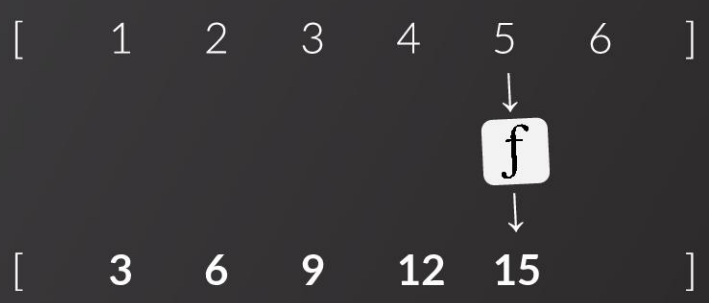
map

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



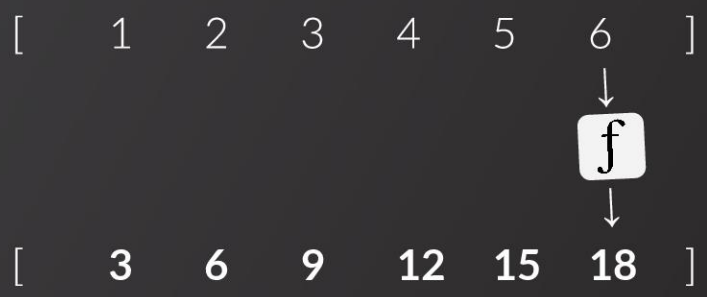
map

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



map

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



map

The function  can be a...

- One-liner lambda expression

```
.map(x -> x/2)
```

- More complex lambda expression

```
.map(x -> {
    … some code …
    return something;
})
```

- Just any function

```
.map(String::toUpperCase)
```

filter

- ***Intuition*** → keeps elements satisfying some condition
- Lambda has one argument and produces a boolean
- Value of boolean determines whether item should be kept

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

filter

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

[2000 2005 2010 2015 2020 2025]

filter

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

[2000 2005 2010 2015 2020 2025]

For each element `y`, what does `y != 2020` evaluate to?

filter

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

[**2000** 2005 2010 2015 2020 2025]
^

`y != 2020` evaluates to true

For each element `y`, what does `y != 2020` evaluate to?

filter

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

[**2000** 2005 2010 2015 2020 2025]
^

`y != 2020` evaluates to true

[2000

For each element `y`, what does `y != 2020` evaluate to?

filter

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

[2000 **2005** 2010 2015 2020 2025]

^

`y != 2020` evaluates to true

[2000 2005

For each element `y`, what does `y != 2020` evaluate to?

filter

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

[2000 2005 **2010** 2015 2020 2025]

^

`y != 2020` evaluates to true

[2000 2005 2010

For each element `y`, what does `y != 2020` evaluate to?

filter

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

[2000 2005 2010 **2015** 2020 2025]
^

`y != 2020` evaluates to true

[2000 2005 2010 2015

For each element `y`, what does `y != 2020` evaluate to?

filter

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

[2000 2005 2010 2015 **2020** 2025]
^

`y != 2020` evaluates to false

[2000 2005 2010 2015

For each element `y`, what does `y != 2020` evaluate to?

filter

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

[2000 2005 2010 2015 2020 **2025**]
^

`y != 2020` evaluates to true

[2000 2005 2010 2015 2025

For each element `y`, what does `y != 2020` evaluate to?

filter

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

[2000 2005 2010 2015 2020 2025]

Result: new stream only containing values satisfying `y != 2020`

[2000 2005 2010 2015 2025]

filter

- No requirement to have simple or one-liner condition

```
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());
```

- Reminder: lambda is anonymous class implementing functional interface
- Implements `Predicate<T>` which has `boolean test(T t)`

sorted

```
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]

distinct

```
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]

limit

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

[3 2 2 3 7 3 5]

Result: new stream only containing values

[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.

collect (Reductions)

- 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

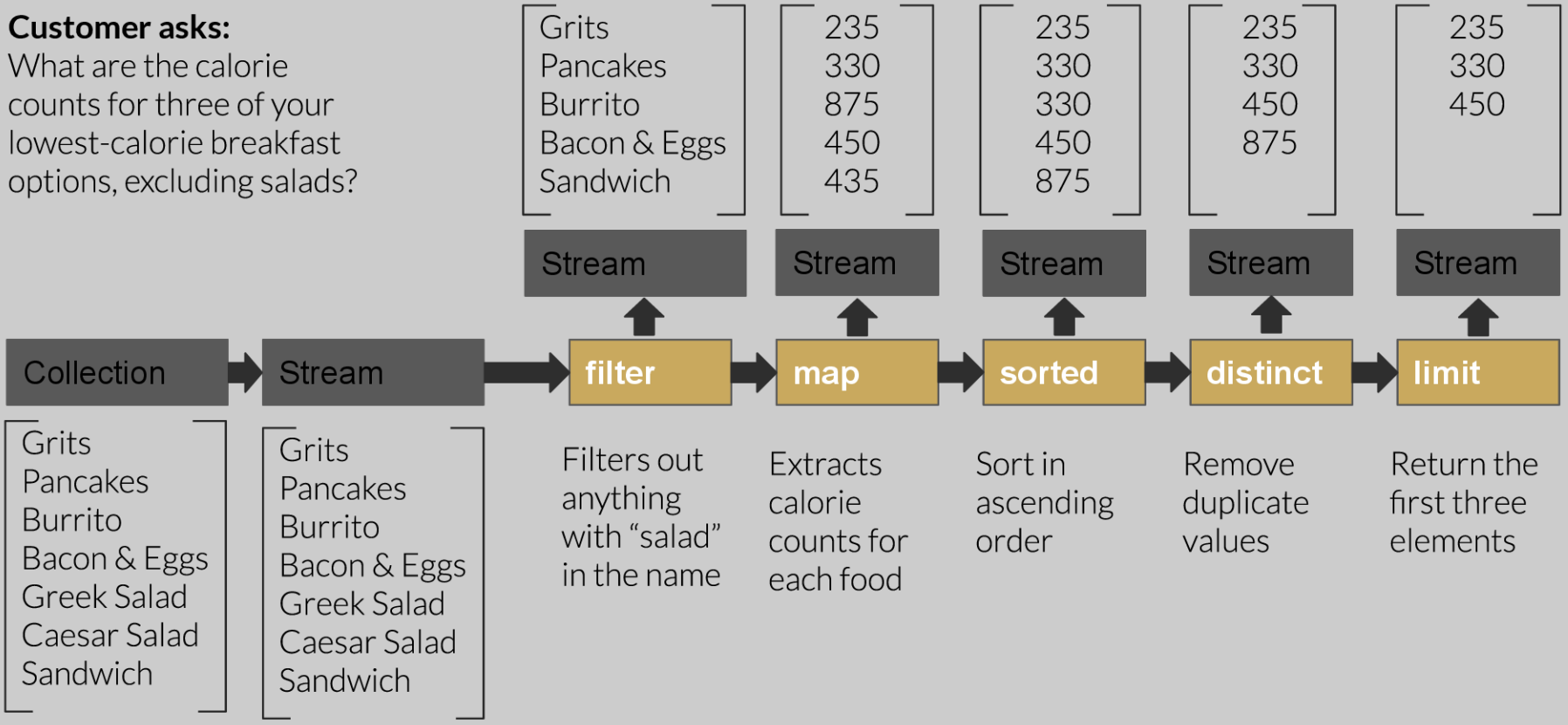
Flatten the data structure (e.g. on stream consisting of Lists)

peek

Do something with each item (like forEach, but not terminal)

Restaurant Example

Customer asks:
What are the calorie counts for three of your lowest-calorie breakfast options, excluding salads?



```
// Step 1: Create the list of breakfast items with their respective calories
List<MenuItem> menu = Arrays.asList(
new MenuItem("Grits", 235),
new MenuItem("Pancakes", 330),
new MenuItem("Burrito", 875),
new MenuItem("Bacon & Eggs", 450),
new MenuItem("Greek Salad", 350), // Exclude as it is a salad
new MenuItem("Caesar Salad", 300), // Exclude as it is a salad
new MenuItem("Sandwich", 435)
);

// Step 2: Use Stream to process the menu and get the calorie counts of the lowest
three breakfast items (excluding salads)
List<Integer> lowestCalories = menu.stream()
.filter(item -> !item.getName().toLowerCase().contains("salad")) // Exclude salads
.map(MenuItem::getCalories) // Extract calorie count
.sorted()         // Sort in ascending order
.distinct()        // Remove duplicates if any
.limit(3)         // Get the first 3 items
.collect(Collectors.toList()); // Collect the result

// Step 3: Output the result
System.out.println("Three lowest-calorie breakfast options: " + lowestCalories);
```

Class `Optional<T>`

The class Optional<T> is a container wrapped around an object and is useful if you do not know whether its content is `null` or not (e.g. when using in fluent programming style).