Java Programming

Arthur Hoskey, Ph.D.
Farmingdale State College
Computer Systems Department

Today's Lecture

- Lambda expressions
- Streams

- A **lambda expression** is an **anonymous method.**
- Here is a lambda expression that adds 10 to a number and returns the result:

The return type is inferred from the return value (int in this case)

```
(int a) -> { return a + 10; }
```

Input Parameter(s)

Lambda operator

Expression

Lambda Expression

You can do the following with lambda expressions:

- Pass a lambda expression to a method as a parameter
- Assign a lambda expression to a variable
- Return a lambda expression from a method

Lambda Expression

- Syntax for lambda expressions:

```
(int a) -> { return a + 10; }
```

You can omit the parameter data types if you want

```
(a) -> { return a + 10; }
```

You can omit the braces and return if there is only one statement in the body

```
(a) -> a + 10;
```

You can omit the parameter parenthesis if there is only one parameter

```
a -> a + 10;
```

You can omit variable if there are no parameters

```
() -> System.out.println("No parameters in lambda");
```

Lambda Expression Syntax

Functional Inteface

- An interface with only one abstract method.

```
interface MyFunctionalInterface
{
   int square(int x);
}
```

Contains only ONE method

Functional Interface

- The example below declares an instance of the functional interface and populates it using a lambda expression.

```
interface MyFunctionalInterface
{
   int square(int x);
}


MyFunctionalInterface mfi;

mfi = (int x) -> { return x * x; };

int result;
result = mfi.square(3);
```

Declare a variable for the functional inteface

Assign a lambda expression to the functional interface variable

Call the method on the functional interface

Functional Interface and Lambda

Pass Functional Interface to Method

- The example below passes a functional interface to a method which then uses it.

```
void TestMethod(MyFunctionalInterface x)
{
   int result;
   result = x.square(3);
   System.out.println(result);
}

MyFunctionalInterface mfi;
mfi = (int x) -> { return x * x; }
TestMethod(mfi);
```

Call the method using the parameter (MyFunctionalInterface is defined on the previous slide)

Pass in the functional interface variable as a parameter to TestMethod

Lambdas and Enclosing Scope

- Lambda expressions do not have their own scope.
- Variables defined inside lambdas are part of the enclosing scope.
- Code inside a lambda expression has direct access to all variables in its enclosing scope.
- The variables used from the enclosing scope should be final or effectively final (effectively final means the variable is not changed after it is initialized).

```
interface MyInterface {
   int add(int x);
}

MyInterface mfi;
int num = 5;
mfi = (int x) -> { return x + num; };
int result = mfi.add(3);
System.out.println(result);
```

The lambda expression has access to num because num is declared in the enclosing scope. The compiler allows access to num because num is effectively final (its value does not change after initialization)

Prints 8

Lambdas and Enclosing Scope

Target Typing

- You do not have to explicitly declare data types in a lambda expression (compiler figures them out).
- Both parameter and return data types are inferred by the compiler.
- For example:

```
interface MyFunctionalInterface {
   int square(int x);
}

MyFunctionalInterface mfi;
mfi = (x) -> { return x * x; };
```

x is an int because 3 is passed in

```
int result = mfi.square(3);
```

Target type of x in lambda is int because the parameter is an int. The return type will also be an int.

Target Typing

Target Typing Error

This example has a compile error.

```
interface MyFunctionalInterface {
   int square(int x);
}
```

Return type of interface method is int. Any lambda expression used for this method should resolve to an int or there will be a compile error.

```
MyFunctionalInterface mfi;
mfi = (x) -> { return x * x; };
```

x is a double because 3.0 is passed in

~~int result = mfi.square(3.0);~~

ERROR. Passing in a double will cause the parameter type and return type of the lambda expression to be inferred to a double. This causes a compile error because a double is being assigned to result which is an int.

Target Typing

- Now on to streams…

Stream

Stream

- A stream is a sequence of elements that operations can be performed on.
- IntStream – Predefined class that is a stream of int.
- IntStream.of – Static method that creates an IntSteam instance.

- The following code creates a stream of int from an array of int:

```
int[] nums = {1, 2, 3, 4, 5};
```

Create an array of int

```
IntStream myStream;
```

Declare a stream variable

```
myStream = IntStream.of(nums);
```

Create a stream instance that contains the items from the nums array

Note: This stream is different from the file related streams.

Stream

Stream vs Collection

- Taken from:

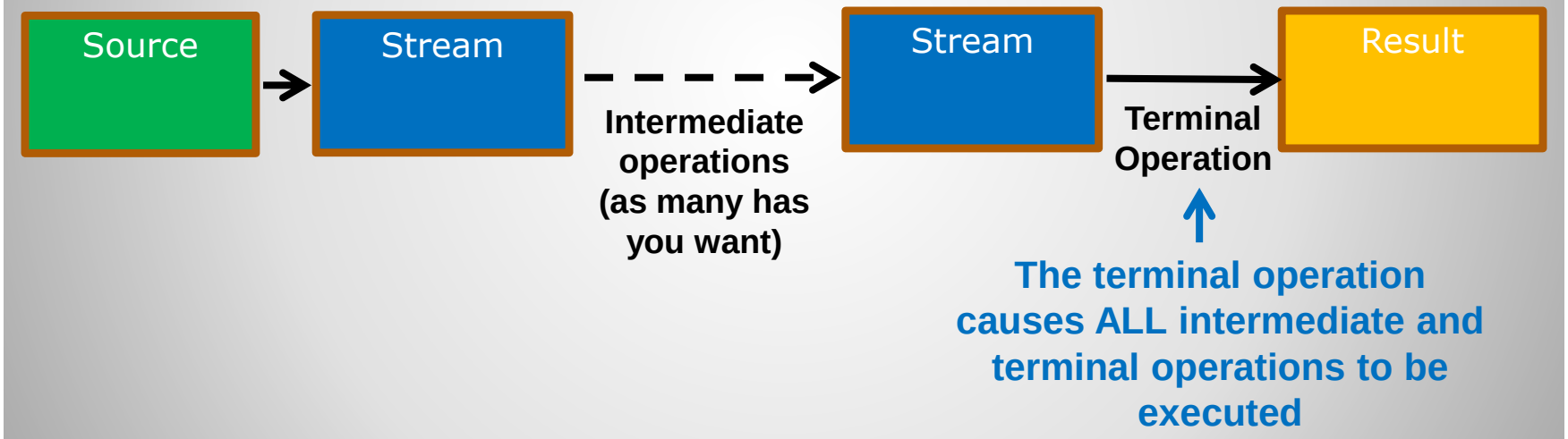
https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html

- "Collections are primarily concerned with the efficient management of, and access to, their elements."
- "By contrast, streams do not provide a means to directly access or manipulate their elements, and are instead concerned with declaratively describing their source and the computational operations which will be performed in aggregate on that source."
- A stream pipeline can be viewed as a query on the stream source.

Stream vs Collection

Stream Processing Flow

1. The data source is to create a stream
2. Any number of intermediate operations are performed on the stream. Intermediate operations produce another stream.
3. A terminal operation is finally performed on the stream. No more operations can be performed after a terminal operation.



Stream Processing Flow

Intermediate Operations

- **Intermediate Operations** – Operations performed on each element of the stream.
- Examples:
  - Square every number in a stream.
  - Apply a method to every number in a stream.
- Each intermediate operation returns a new stream object (allows for creating a pipeline of calls to operate on the stream).
- Intermediate operators are "lazy" (they do not actually initiate processing).
- You must call a terminal operation to execute all intermediate operations (see upcoming slide for terminal operation).

- **Reduction -** Operations that take the elements of a stream and produce one result. For example: sum, average, min, max, etc… The stream is being "reduced" to one value.

- **Terminal Operations** – Actually initiate processing. All processing on a stream is delayed until a terminal operation is called. Uses "eager" evaluation (do immediately).

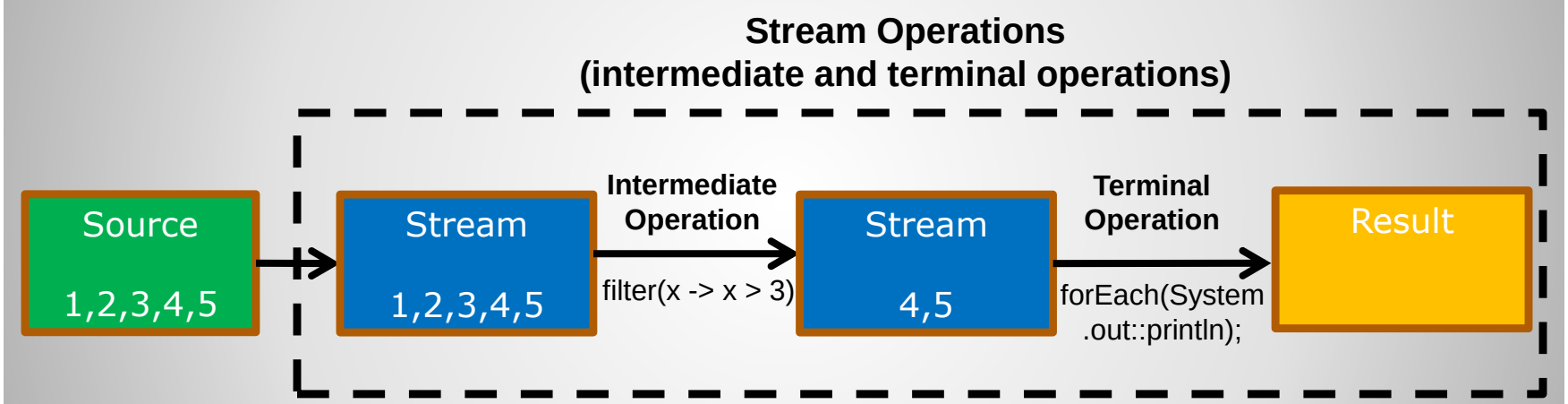
- Once a terminal operation is applied to a stream you basically cannot use that stream again.

- All reductions are terminal operations.
- However, not all terminal operations are reductions. For example, forEach is a terminal operation that does not produce one value.

Reduction and Terminal Operations

Stream Example

```
int[] nums = {1, 2, 3, 4, 5};
IntStream myStream = IntStream.of(nums);
myStream.filter(x -> x > 3).forEach(System.out::println);
```



IntStream.of creates a stream from an array

filter only lets elements that meet a condition go through (greater than 3 in this case)

forEach applies the given method to all stream items (prints items on the screen in this case)

Stream Example

Stream and Other Stream Interfaces

Stream Interface

- **Stream<T>** – A sequence of T type values. T can be any reference type.

Other Stream Interfaces

- **IntStream** – A sequence of primitive int values.
- **DoubleStream** – A sequence of primitive double values.
- **LongStream** – A sequence of primitive long values.

Collection class stream Method (Creating a stream)

- You can use the stream() method of the Collection interface to create a stream.
- The stream() method returns a Stream<T> instance.

```
Integer[] nums = {1, 2, 3, 4, 5};
Collection<Integer> coll = Arrays.asList(nums);
```

Creates a Stream<T> from the list (T stands for Integer in this case)

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

This example does not contain any intermediate operations but you can add as many as you want here. Intermediate operation calls would be placed after stream() but before the terminal forEach.

Collection's stream Method

Reference to Primitive Conversion

- Stream<T> contains only reference types.
- There are times when you need to operate on primitive types.
- Use **mapToInt** or **mapToDouble** to convert Integer and Double wrapper types to their equivalent primitive types.
- Certain methods require a sequence of primitive values (for example average()).

```
Integer[] nums = {1, 2, 3, 4, 5};
Collection<Integer> coll = Arrays.asList(nums);

coll.stream().mapToInt(x -> x).forEach(System.out::println);
```

Lambda expression x->x is applied to each element

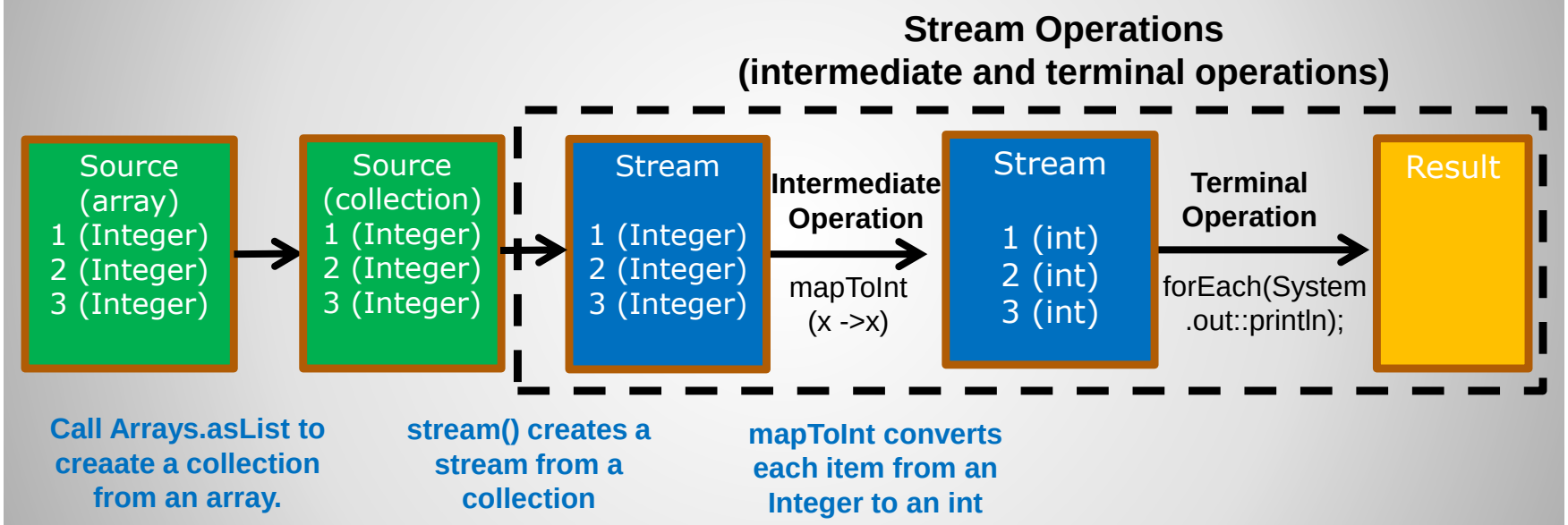
Stream() returns a sequence of Integer type elements (not int)

mapToInt generates an IntStream instance (sequence of int)

Reference to Primitive Conversion

Stream – Reference to Primitive Conversion

```
Integer[] nums = {1, 2, 3};
Collection<Integer> coll = Arrays.asList(nums);
coll.stream().mapToInt(x -> x).forEach(System.out::println);
```



Stream Reference to Primitive Conversion

Reference to Primitive Conversion – Another Example

```
int[] data = {33, 45};
int sum = Arrays.stream(data)
   .sum();

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

Stream is created from an array which contains primitive types (no need to convert to a primitive type for sum).

Sum requires primitive types

```
int sum2 = Arrays.asList(33, 45)
   .stream()
   .mapToInt(i -> i)
   .sum();

System.out.println(sum2);
```

Creates a stream of Integer objects since nums are in a List. A List can only store Object types so the int values 33 and 45 must be boxed into Integer objects. The boxing forces us to unbox when doing calculations. This is why mapToInt must be called. mapToInt unboxes the Integer objects.

mapToInt unboxes Integer objects so sum can have primitives to operate on

Reference to Primitive Conversion

range and rangeClosed

- IntStream and LongStream classes have helper methods to easily create collections of numbers (range and rangeClosed).

- **range** – Create a stream in the given range. It does NOT include the ending value.

Creates a stream with the elements 1, 2, 3, 4

```
IntStream.range(1, 5)
            .forEach(System.out::println);
```

- **rangeClosed** – Create a stream in the given range. It does include the ending value.

Creates a stream with the elements 1, 2, 3, 4, 5

```
IntStream.rangeClosed(1, 5)
            .forEach(System.out::println);
```

range and rangeClosed

forEach

- forEach is a terminal operation that performs an action on all elements of a stream.
- For example, printing - If you want to print all elements of a stream you can apply a print method to each element of a stream.

```
int[] nums = {1, 2, 3, 4, 5};
IntStream myStream = IntStream.of(nums);
```

forEach applies the passed in method to each item in the stream

Pass in a reference to the println method

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

:: is used for a method reference in Java

Note: The compiler will convert System.out::println to x->System.out.println(x). For example:

```
myStream.forEach(x->System.out.println(x)); // Same as forEach above
```

Stream - forEach

filter

- An intermediate operation that generates another stream based on a test (based on a predicate).
- The test is applied to all elements in stream.

```
int[] nums = {1, 2, 3, 4, 5};
IntStream myStream;
myStream = IntStream.of(nums);

myStream.filter(x -> x > 3).forEach(System.out::println);
```

Only use numbers greater than 3

Prints the following

4
5

- All intermediate and terminal operations must be performed in one chain together (stream pipeline). For example, the following will not work:

```
myStream.filter(x -> x > 3);
myStream.forEach(System.out::println);
```

Cannot do operations on the same stream in different statements (they must be "chained" together in the same statement)

filter

map

- Returns a stream that is the result of applying a given function.
- The map method can be applied as many times as you want as long as those calls are before a terminal operation.

```
int[] nums = {1, 2, 3, 4, 5};
IntStream myStream;
myStream = IntStream.of(nums);
```

Squares each number in the stream. The lambda is being applied to all stream elements.

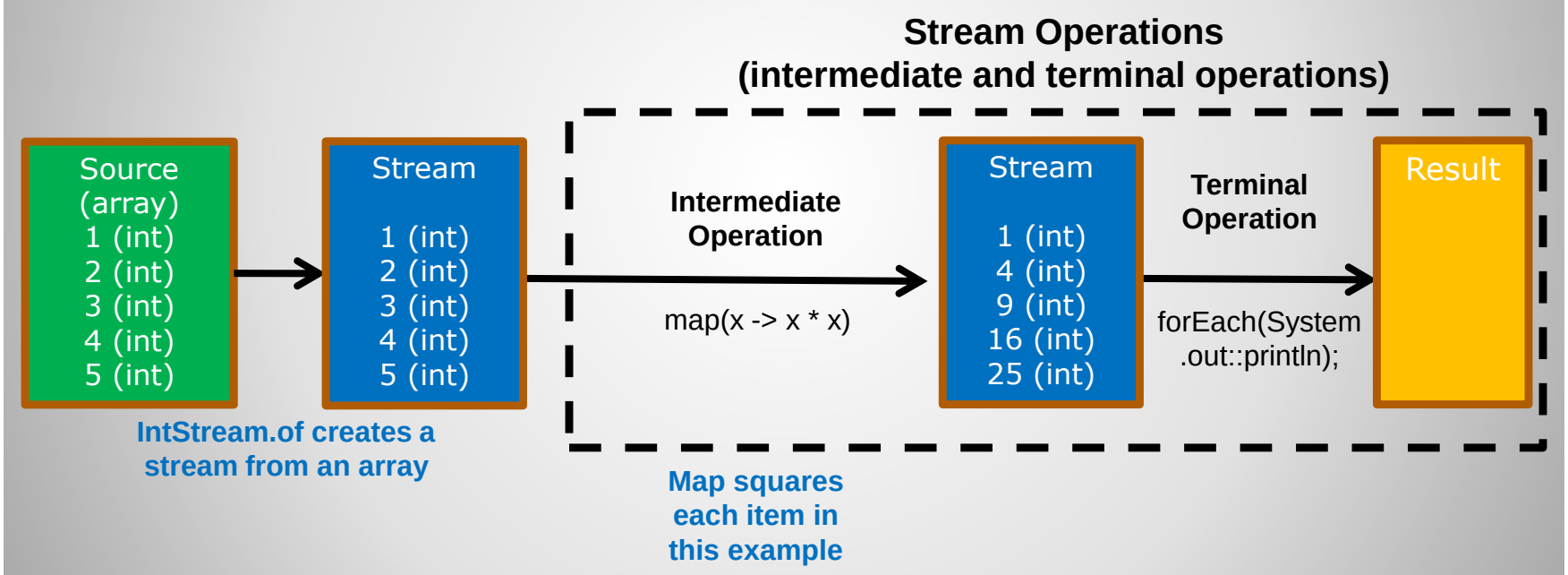
```
myStream.map(x -> x * x)
            .forEach(System.out::println);
```

Print out all the elements

map

Stream – map

```
int[] nums = {1, 2, 3, 4, 5};
IntStream myStream;
myStream = IntStream.of(nums);
myStream.map(x -> x * x).forEach(System.out::println);
```



map

map (change type String to Integer)

- You can use the map operation to change the stream elements to a different data type.

Create a list of String

```
List<String> numbers = List.of("1", "2", "3");
```

Converts each element to an Integer

```
numbers.stream()
        .map(Integer::valueOf)
        .forEach(System.out::println);
```

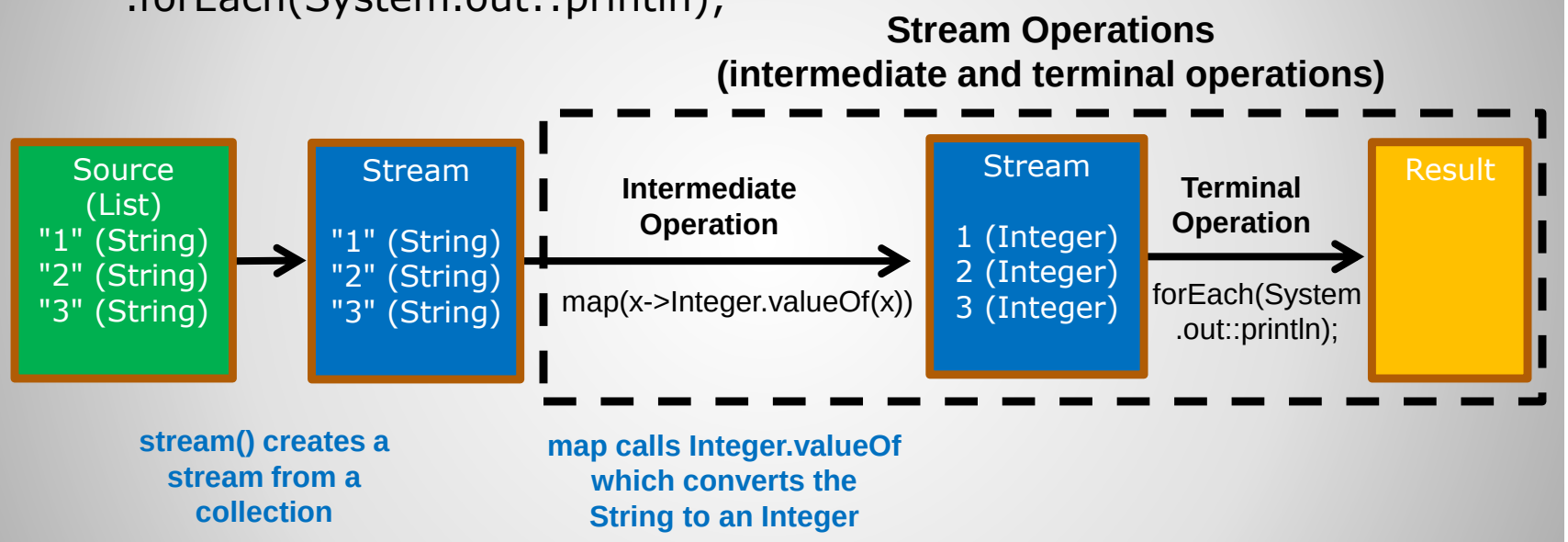
```
numbers.stream()
    .map(x->Integer.valueOf(x))
    .forEach(System.out::println);
```

Integer::valueOf is the same as: x->Integer.valueOf(x)

map – Change Type

map - Change Type

```
List<String> numbers = List.of("1", "2", "3");
numbers.stream()
        .map(x->Integer.valueOf(x))
        .forEach(System.out::println);
```



map - Change Type

map (change type Employee to String)

- The following example creates a stream from a List of Employee objects and then converts them to String objects.

```
class Employee {
    public Employee(String name, int id) { this.name = name; this.id = id; }
    private String name;
    private int id;
    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
    public int getId() { return id; }
    public void setId(int id) { this.id = id; }
}

List<Employee> empList = List.of (
   new Employee("Rose", 100),
   new Employee("Mateo", 101));
empList.stream()
          .map(e -> e.getName())
          .forEach(System.out::println);
```

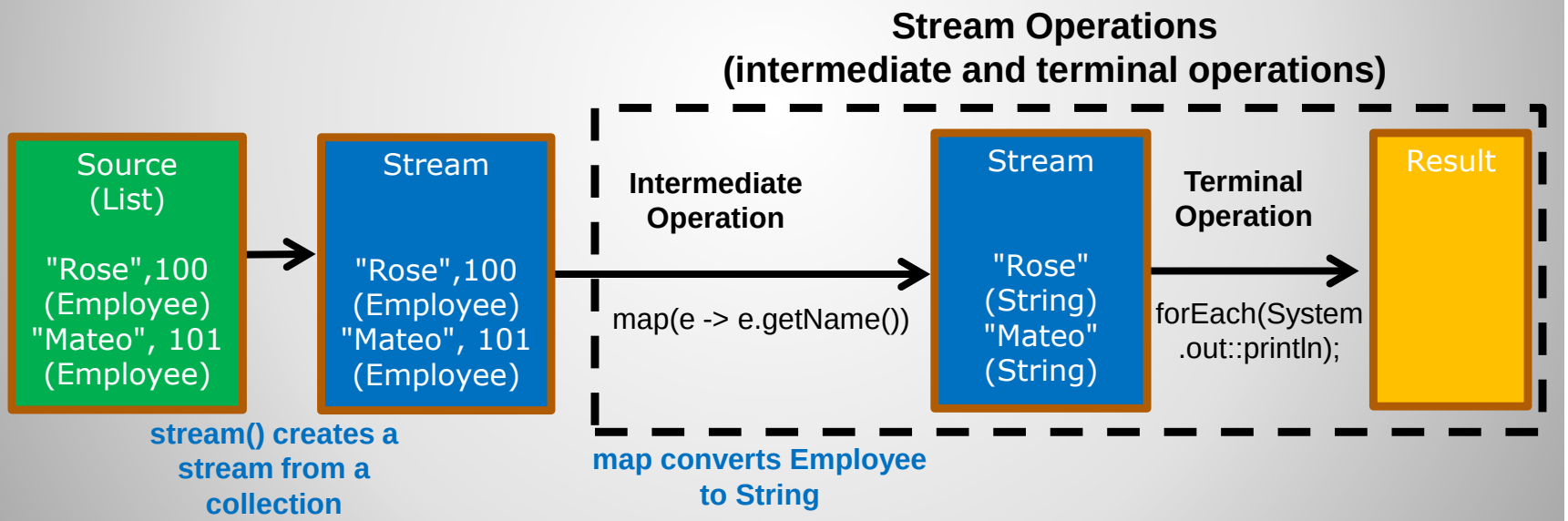
Create a list of Employee

Converts each element to a String

map – Change Type

map - Change Type (Employee to String)

```
List<Employee> empList = List.of (
   new Employee("Rose", 100),
   new Employee("Mateo", 101));
empList.stream()
          .map(e -> e.getName())
          .forEach(System.out::println);
```



map - Change Type

sorted

- Sorts the elements in the stream.

```
int[] nums = {3, 1, 4, 5, 2};
IntStream myStream;
myStream = IntStream.of(nums);

myStream.sorted()
            .forEach(System.out::println);
```

Unsorted array of numbers

Sorts the numbers in ASCENDING order

The numbers will be printed in sorted order (1, 2, 3, 4, 5)

OR

```
myStream.boxed()
            .sorted(Collections.reverseOrder())
            .forEach(System.out::println);
```

Sorts the numbers in DESCENDING order (they must be boxed to do this)

The numbers will be printed in sorted order (5, 4, 3, 2, 1)

sorted

average

- Use **average()** to find the average of a sequence of primitive double elements.
- average() returns an instance of OptionalDouble.
- OptionalDouble has a method getAsDouble which returns a primitive double.

```
List<Double> numbers = List.of(1.0, 2.0, 3.0, 4.0, 5.0);

double avg = numbers.stream()
                    .mapToDouble(x -> x)
                    .average()
                    .getAsDouble();
```

Convert Double to primitive

Calculate the average

Convert result to a primitive double type

average

collect

- You can convert a stream back to a collection.

```
List<Integer> list;

list = IntStream.range(1,5).boxed().collect(Collectors.toList());

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

Creates a stream with the elements 1, 2, 3, 4

Collect Creates a List<Integer> instance

boxed creates Integer class elements from the int primitive elements in the IntSteam

Print all elements in the list collection instance (list is NOT a stream, it is a List<Integer>)

collect

Use map to change type and then save to a list

```
class Employee {
    public Employee(String name, int id) { this.name = name; this.id = id; }
    private String name;
    private int id;
    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
    public int getId() { return id; }
    public void setId(int id) { this.id = id; }
}

List<Employee> empList = List.of (
   new Employee("Rose", 100),
   new Employee("Mateo", 101));
List<String> nameList =
   empList.stream().map(e ->e.getName()).collect(Collectors.toList());

for (String n: nameList) {
   System.out.println(n);
}
```

Create a list of Employee using List.of

map takes the Employee instances and pulls out the names. Only string instances are in the stream after map runs.

collect converts the strings in the stream to a List<String>

Changing Type and Saving to List

- End of Slides

End of Slides