



CODE DSM: JAVA WEEK 10 FUNCTIONAL PROGRAMMING

DMACC Fall 2019

Instructor: Greg Hazen

AGENDA REVIEW



WEEK 9 QUIZ

AGENDA



FUNCTIONAL
PROGRAMMING



FUNCTIONS WITH
COLLECTIONS

METHOD VS FUNCTION

Method

- Called by Name
- Can be passed in data
- Can return data
- Has access to 'this'
- Can use instance variables

Function

- Called by Name
- Can be passed in data
- Can return data
- Independent of any object

FUNCTION

They act like and behave just like any other variable

They have types

They can be passed from one object to another through a method call

METHOD

Return Type Name Input Parameter

↓ ↙ ↘

```
public String doSomething (String value) {  
    // do something...  
}
```

↙ Body

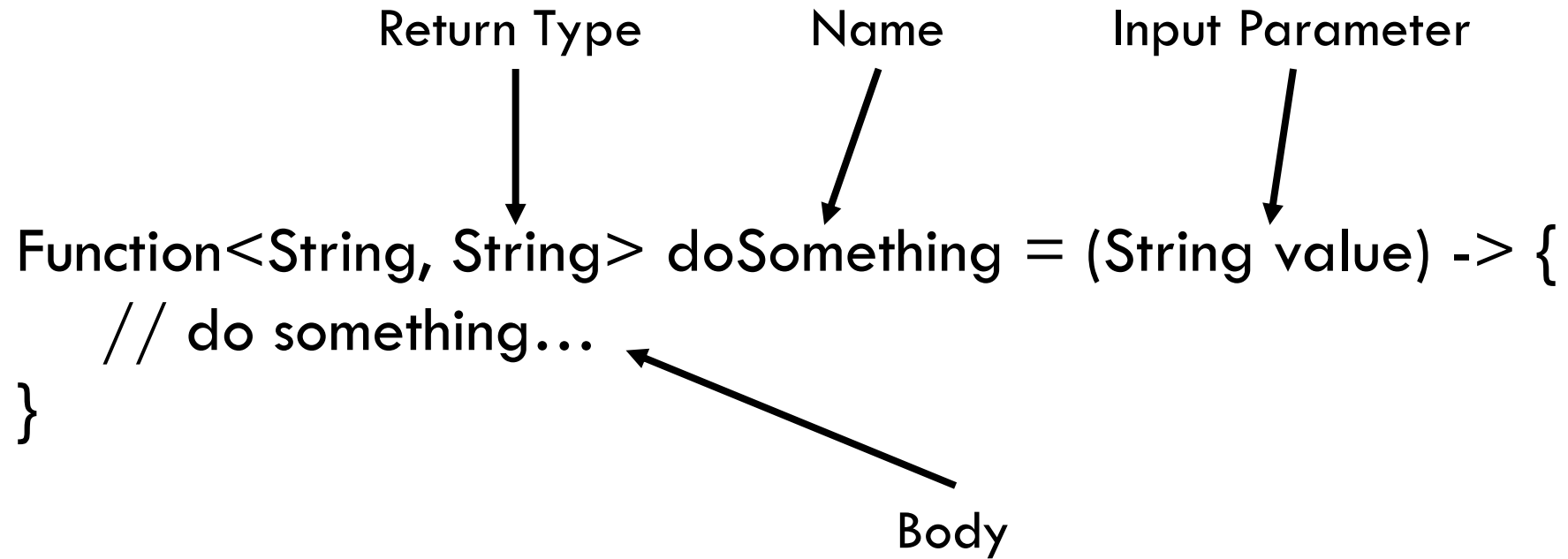
The diagram illustrates the components of a Java method signature. It shows the code: `public String doSomething (String value) {` followed by an indented line `// do something...` and a closing brace `}`. Four labels with arrows point to specific parts: 'Return Type' points to 'String', 'Name' points to 'doSomething', 'Input Parameter' points to '(String value)', and 'Body' points to the indented line of code.

FUNCTION

Return Type Name Input Parameter

Function<String, String> doSomething = (String value) -> {
 // do something...
}

Body

A diagram illustrating the components of a Kotlin function signature. The function signature is 'Function<String, String> doSomething = (String value) -> { // do something... }'. Four arrows point to specific parts: 'Return Type' points to 'Function<String, String>', 'Name' points to 'doSomething', 'Input Parameter' points to '(String value)', and 'Body' points to the block '{ // do something... }'.

FUNCTION

Input Type



```
Function<String, String> doSomething = (String value) -> {  
    // do something...  
}
```



FUNCTION

```
Function<String, String> doSomething =  
    (String value) -> {  
        // do something...  
    } }
```

} Lambda Expression

FUNCTION

```
Function<String, String> doSomething =  
    (String value) -> // do something on a single line...
```



Lambda Expression

FUNCTION

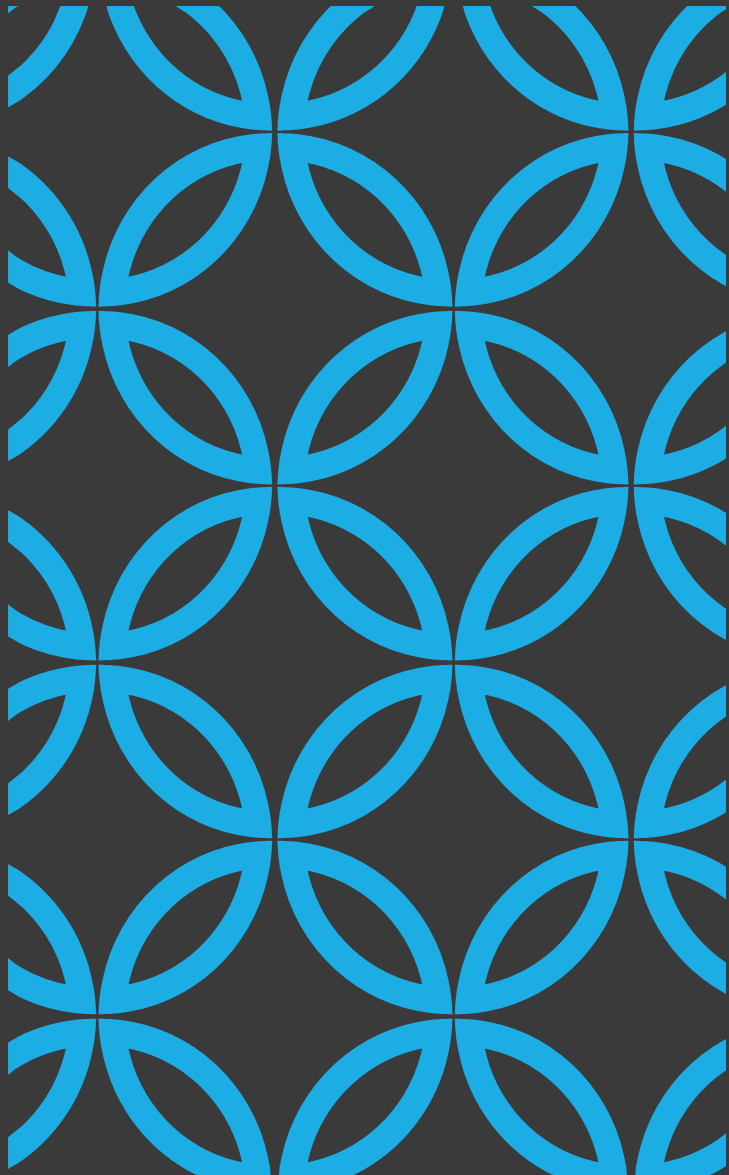
First and
Second
Input Types

Return Type

```
BiFunction<String, String, String> doSomething =  
(String value1, String value2) -> {  
    // do something...  
}
```

FUNCTION

```
BiFunction<String, String, String> doSomething =  
(String value1, String value2) -> {  
    // do something...  
}  
  
// Use the apply method to call it directly  
String result = doSomething.apply("str 1", "str 2");
```



LET'S SEE IT IN INTELLIJ

FUNCTIONAL INTERFACE

Any interface with a single method is considered a functional interface and can be treated as a lambda

Use '->' to declare an implementation

```
@FunctionalInterface
public interface Greeter {
    String greet(String name);
}
```

```
Greeter g = s -> "Hi " + s + "!";
String result = g.greet("Greg");
//result is "Hi Greg!"
```

FUNCTIONAL INTERFACE

The Pluralsight video says the annotation is simply for “convenience”

Helpful to prevent developers from breaking your design

@FunctionalInterface

```
public interface Greeter {  
    String greet(String name);  
}
```

```
Greeter g = s -> "Hi " + s + "!";  
String result = g.greet("Greg");  
//result is "Hi Greg!"
```

FUNCTIONAL INTERFACE

Any interface with a single method is considered a functional interface

Important to make previous Java interfaces functional

```
@FunctionalInterface
public interface FileFilter {
    boolean accept(File pathname);
}
```


FUNCTIONAL INTERFACE

Using lambdas
can make code
cleaner and
faster to read

```
FileFilter javaFileFilter =  
new FileFilter() {  
    @Override  
    public boolean accept(File file) {  
        return file.getName().endsWith(".java");  
    }  
};
```



```
FileFilter javaFileFilter =  
(File file) -> file.getName().endsWith(".java");
```

FUNCTIONAL INTERFACE

Lambdas don't
require as much
memory or
processing time
to create

You're not
actually create a
new object

```
FileFilter javaFileFilter =  
new FileFilter() {  
    @Override  
    public boolean accept(File file) {  
        return file.getName().endsWith(".java");  
    }  
};
```



```
FileFilter javaFileFilter =  
(File file) -> file.getName().endsWith(".java");
```

EXERCISE

Create a map called `agesByName` with the following names as key to the corresponding age

- Alex age 47
- Bob age 30
- Carry age 10
- Deb age 23

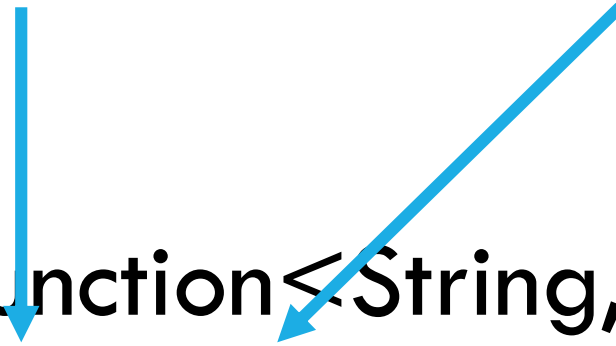
EXERCISE

1. Create a lambda that takes in the name and age and returns it formatted like “Alex is 30”
2. Create a method that takes in `agesByName` and the lambda. Loop over each name in the map and print the formatted name and age using the lambda.

JAVA CAN INFER PARAMETER TYPES

From our exercise

```
BiFunction<String, Integer, String> format =  
(String name, Integer age) -> name + " is " + age;
```



```
BiFunction<String, Integer, String> format =  
(name, age) -> name + " is " + age;
```

TYPES OF FUNCTIONS

Instead of creating a custom FunctionalInterface, try to use existing ones from `java.util.function`

Too many to memorize, review them to know what's available

TYPES OF FUNCTIONS

All `java.util.function` functions can be categorized into

- Supplier: Takes no parameters but returns an object
- Consumer: Takes a parameter but returns nothing
- Predicate: Takes a parameter and returns a boolean
- Function: Takes a parameter and returns an object

TYPES OF FUNCTIONS

Consumer, Predicate and Function can also take two parameters

- BiConsumer: Takes 2 parameters but returns nothing
- BiPredicate: Takes 2 parameters and returns a boolean
- BiFunction: Takes 2 parameters and returns an object

MAP FOREACH

Executes a given function for each key/value pair in the map

Requires the function takes the key/value types as parameter and returns nothing

- This is the BiConsumer function

EXERCISE

Change your code to use *Map's forEach* method instead of your *printEach* method

COLLECTION STREAM

Collections also have the ability to stream the data through multiple functions

```
Set<Integer> mySet = ...;
```

```
mySet.stream(); // returns Stream<Integer>
```

COLLECTION STREAM

Stream class has lots of methods for applying functions to the data

Most of the methods return the Stream again, so you can chain the calls

```
mySet.stream().filter(...).sorted(...);  
// still returns a Stream
```

COLLECTION STREAM

The Stream doesn't execute anything on the data until you end the stream with a *terminal operator*

- Called “lazy execution”

The following will execute each Stream operation and return a count of the elements left in the stream

```
mySet.stream()  
    .filter(...).sorted(...)  
    .count();
```

COLLECTION STREAM

The Stream operators don't change the original data structure or values

If you want the data structure after executing the operations, use a *terminal operator* that returns the new data structure

- `collect(Collectors.toList())` // stream as a list
- `collect(Collectors.toSet())` // stream as a set
- `collect(...)` // create a custom collector

COLLECTION STREAM

Together the stream, operators and terminal operator make up a *pipeline*

EXERCISE

1. Change your code to use the `Map's entrySet().stream()` method
2. Filter out all people less than 25
3. Sort the names by age
4. Use the *terminal operator* `forEach` to print the name and age like before

JAVA CAN INFER PARAMETER TYPES

From our exercise

```
.filter((Map.Entry<String, Integer> nameAndAge)  
      -> nameAndAge.getValue() > 25)
```



```
.filter(nameAndAge  
      -> nameAndAge.getValue() >= 25)
```

Because `agesByName.entrySet().stream()` returns
`Stream<Map.Entry<String, Integer>>`

METHOD REFERENCE

Methods can be passed as a function

- Parameters must match
- Return type must match

Use :: operator

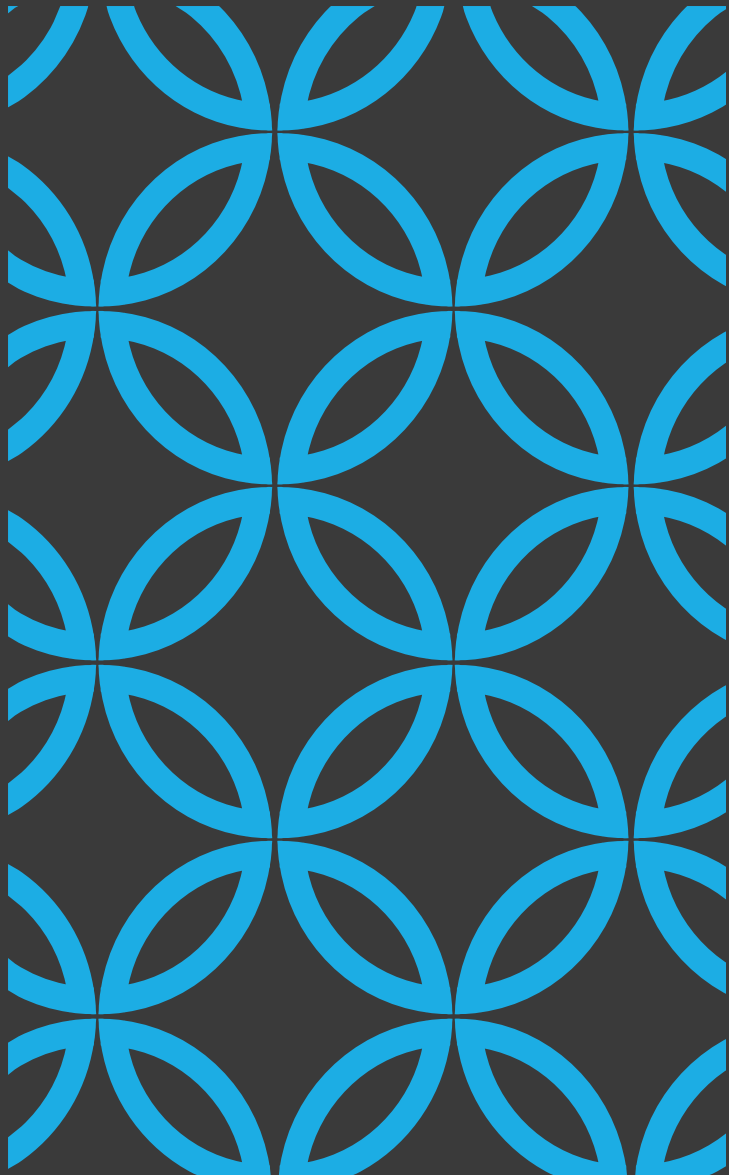
```
Function<String, Integer> functionLength =  
    s -> s.length();
```

```
Function<String, Integer> referenceLength =  
    String::length;
```

COMPOSING FUNCTIONS

Two or more functions can be combined into a new function that

- Takes the parameters of the first function
- Returns the type of the last function



LET'S SEE IT IN INTELLIJ

COLLECTION STREAM

Methods that take a function and use it are called *Higher Order Functions*

Stream operators are considered Higher Order Functions

Common operators include

- `map(Function<T, R> f)`
- `flatMap(Function<T, Stream<R>> f)`
- `filter(Predicate<T> f)`
- `reduce(BiFunction<T, U, R> f)`

COLLECTION STREAM: MAP

Apply the given function to each value

Example: Map a list of integers to a list of integers + 1

```
List<Integer> addedOne =  
    values.stream()  
        .map(x -> x + 1)  
        .collect(Collectors.toList());
```

COLLECTION STREAM: FLATMAP

Example:

Assume you have a collection of orders and each order contains a collection of items

The following produces a list of items from all the orders:

```
List<Item> items = orders  
    .flatMap(order -> order.getItems().stream())  
    .collect(Collectors.toList());
```

COLLECTION STREAM: FILTER

Keep each item only if the given function returns true for that item

Example: Filter a list of integers for evens

```
List<Integer> evens =  
    values.stream()  
        .filter(x -> x % 2 == 0)  
        .collect(Collectors.toList());
```


COLLECTION STREAM: REDUCE

Pass each value into the given function and reduce the list to a single value returned from that function

Example – Reduce a list of integers to its sum

```
Integer sum = values.stream().reduce(  
    (value, total) -> element + accumulator)  
    .orElse(0); // If values is empty, return 0
```

COLLECTION STREAM: REDUCE

Pass each value into the given function and reduce the list to a single value returned from that function

Example – Reduce a list of integers to its sum

```
Integer sum = values.stream().reduce(0,  
    (value, total) -> value + total);
```

EXERCISE

Provided (in Slack) classes and grades like:

```
HashMap<String, List<Integer>> gradesByClass =  
    new HashMap<>();
```

```
gradesByClass.put("Morning Class", Arrays.asList(99, 70, 81));  
gradesByClass.put("Evening Class", Arrays.asList(95, 91, 74));
```

EXERCISE

Provided classes and grades...

1. Create a flattened and sorted list of grades from all classes (e.g. [70, 74, 81, 91, 95, 99]) . Print the list.
2. Use reduce function to get the sum of all grades and size method to get a count. Print the average grade (sum / count).

WHY USE STREAMS

We could continue using loops, but...

Streams avoid extra computation

Streams support parallel processing

More readable/compact

OPTIONALS

What happens when a stream is empty?

```
int total = stream.reduce(0, sum);
```

```
int max =
```

```
stream.max(Comparator.naturalOrder());
```

OPTIONALS

If the reduce doesn't have a default value, an Optional is returned

```
Optional<Integer> max =  
stream.max(Comparator.naturalOrder());
```

```
max.isPresent()
```

```
max.get()
```

```
max.orElse(defaultValue);
```

EXERCISE

Using the grades exercise from earlier:

1. Print the max grade; default it to 100 if there's no grades.
2. Print the min grade; default it to 0 if there's no grades.



LET'S SEE HOW TO USE INTELLIJ'S DEBUG MODE

ASSIGNMENT

Review assignment for any clarifications

Clone it together

SEE YOU DECEMBER 20TH!

Don't forget to ask question early in the week