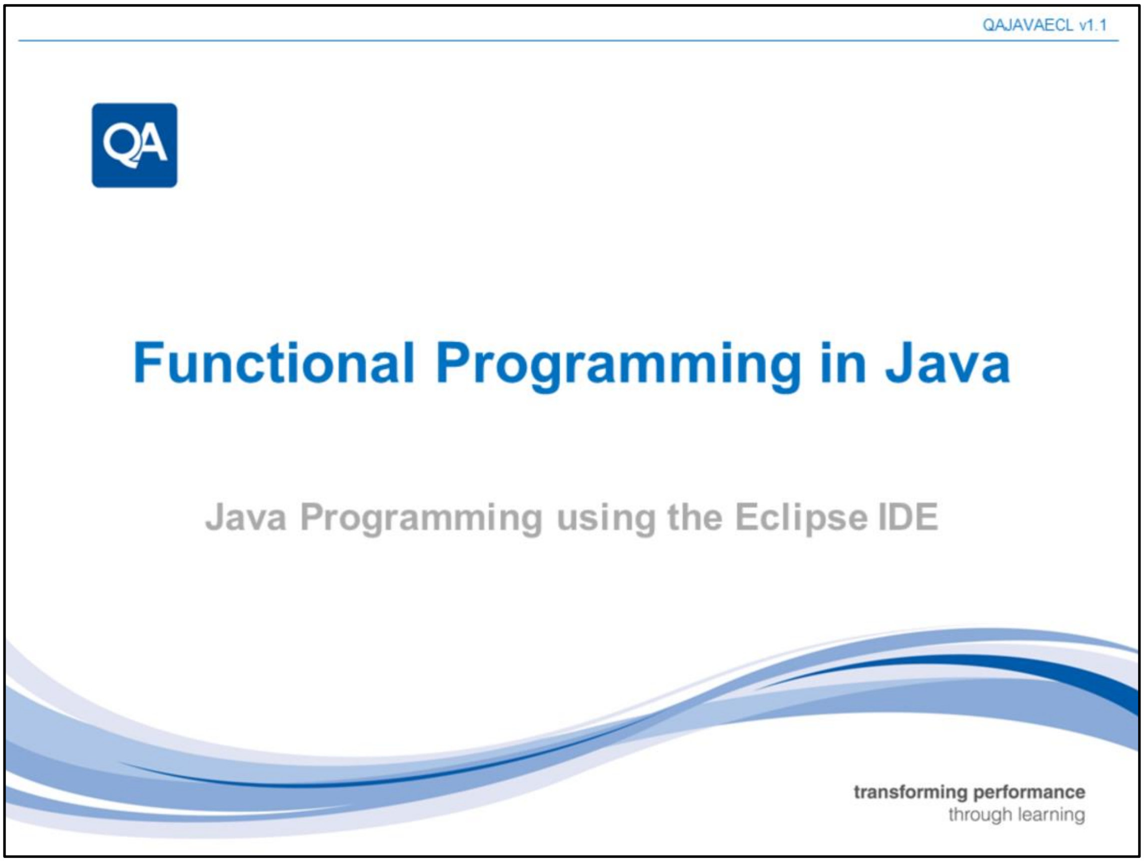
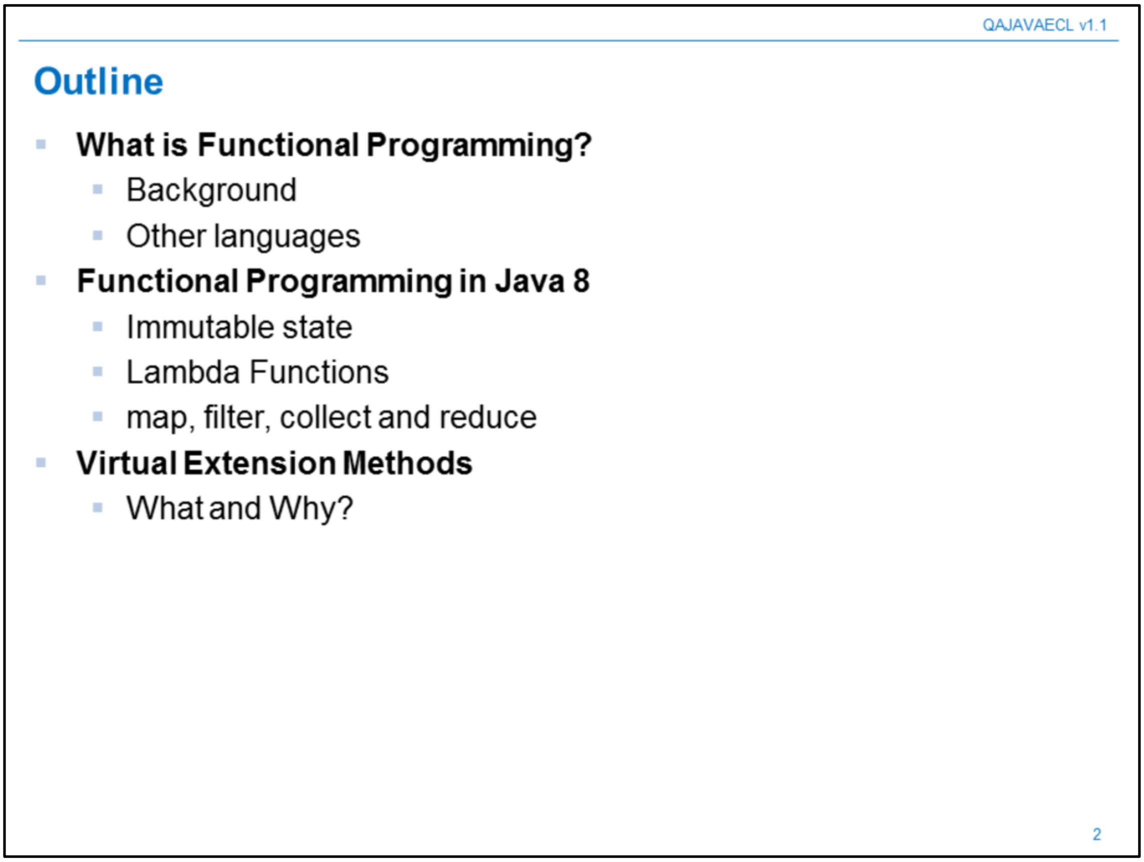


QAJAVAECL v1.1

Functional Programming in Java

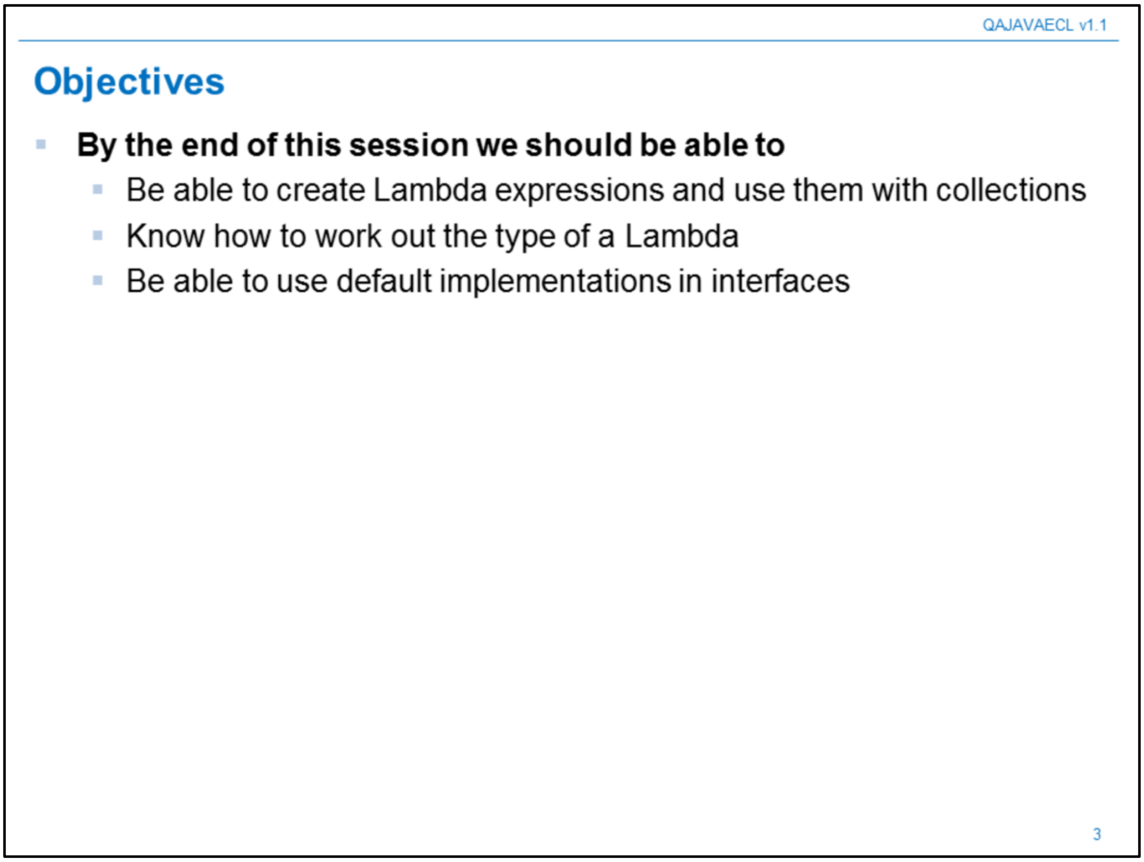
Java Programming using the Eclipse IDE

transforming performance
through learning



Outline

- **What is Functional Programming?**
  - Background
  - Other languages
- **Functional Programming in Java 8**
  - Immutable state
  - Lambda Functions
  - map, filter, collect and reduce
- **Virtual Extension Methods**
  - What and Why?

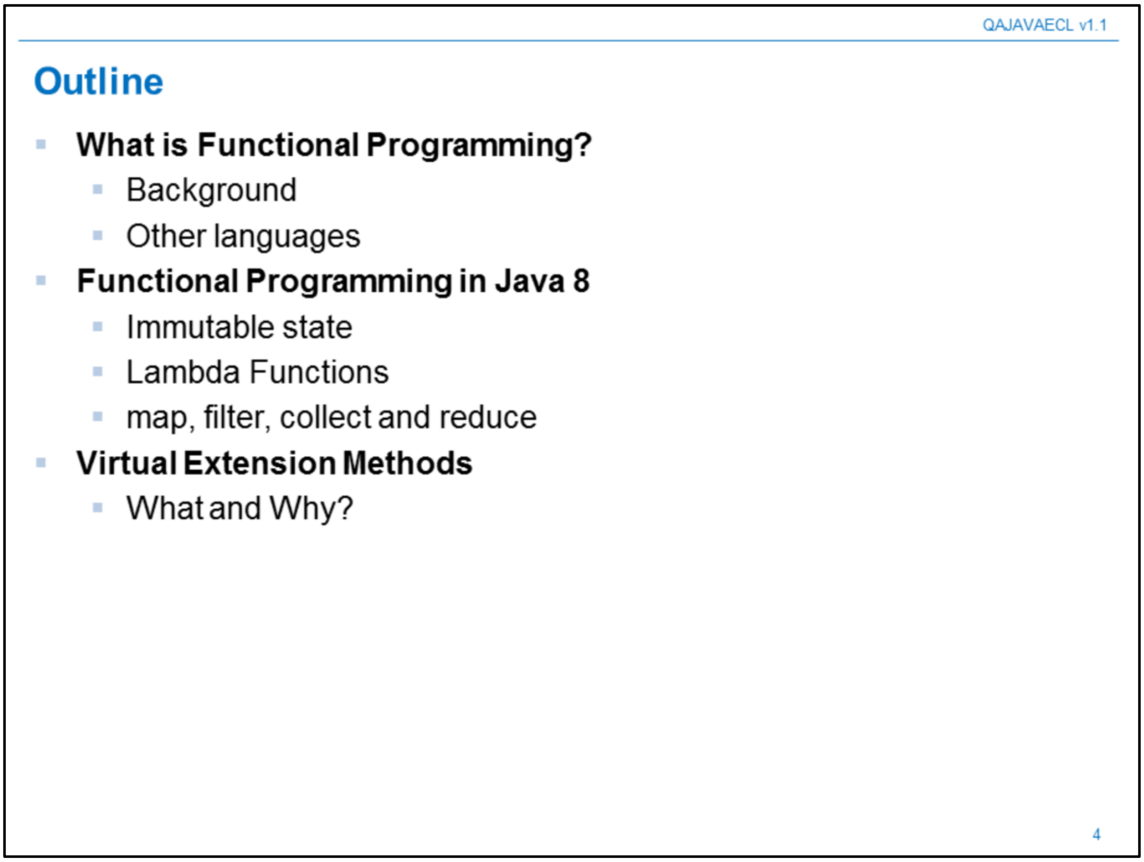


QAJAVAECL v1.1

Objectives

- **By the end of this session we should be able to**
  - Be able to create Lambda expressions and use them with collections
  - Know how to work out the type of a Lambda
  - Be able to use default implementations in interfaces

3

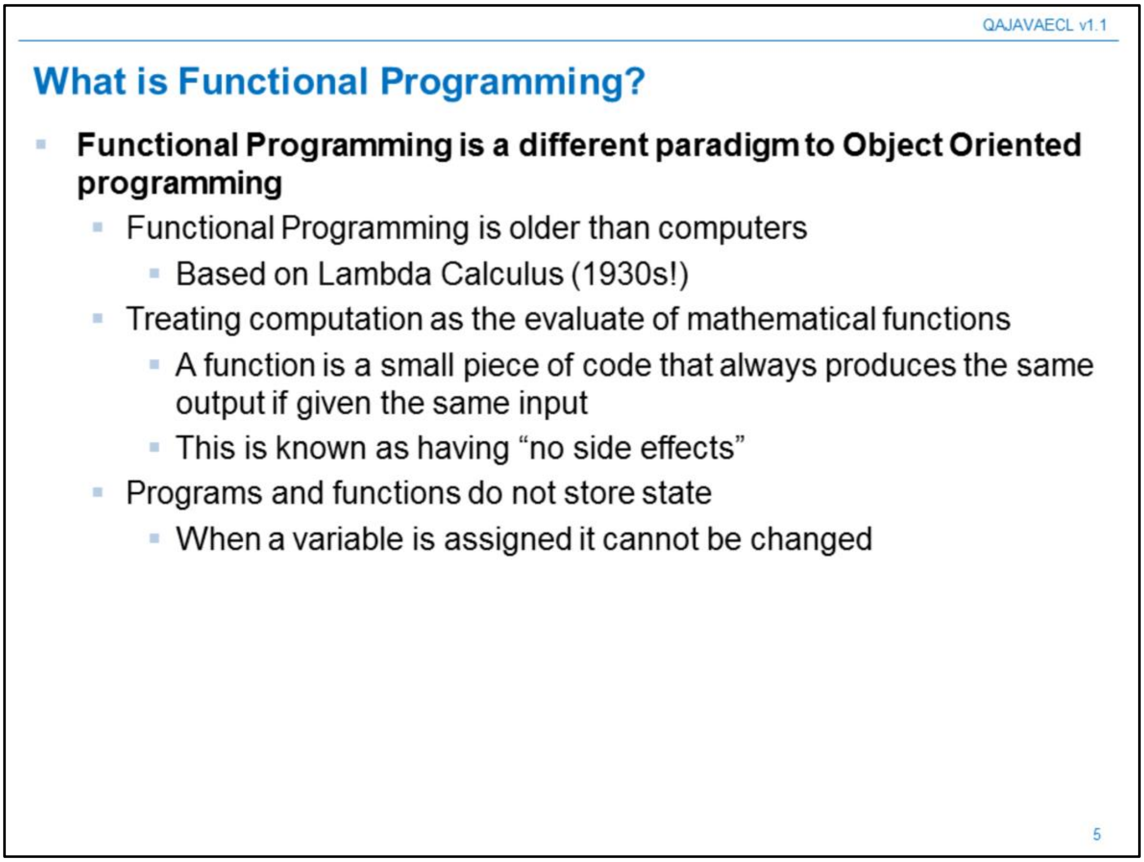


QAJAVAECL v1.1

Outline

- **What is Functional Programming?**
  - Background
  - Other languages
- **Functional Programming in Java 8**
  - Immutable state
  - Lambda Functions
  - map, filter, collect and reduce
- **Virtual Extension Methods**
  - What and Why?

4

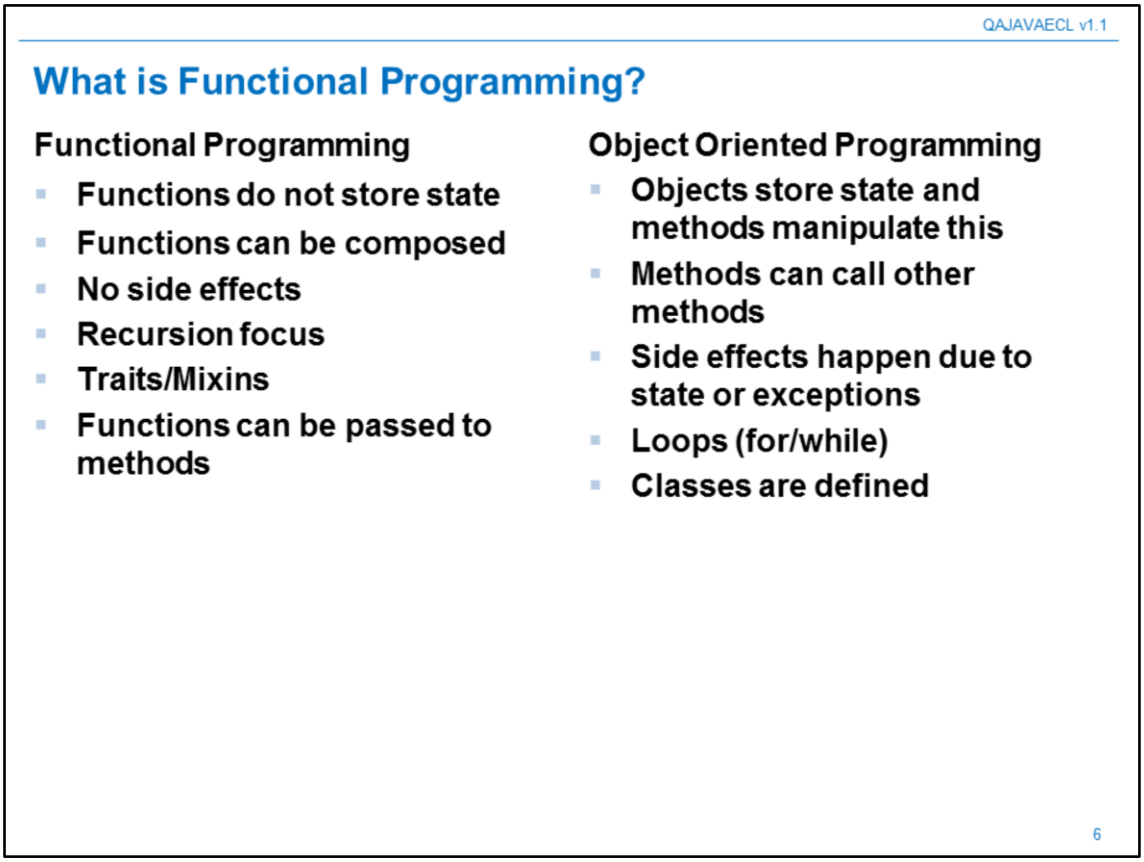


QAJAVAECL v1.1

What is Functional Programming?

- **Functional Programming is a different paradigm to Object Oriented programming**
  - Functional Programming is older than computers
    - Based on Lambda Calculus (1930s!)
  - Treating computation as the evaluate of mathematical functions
    - A function is a small piece of code that always produces the same output if given the same input
    - This is known as having “no side effects”
  - Programs and functions do not store state
    - When a variable is assigned it cannot be changed

5



QAJAVAECL v1.1

What is Functional Programming?

Functional Programming

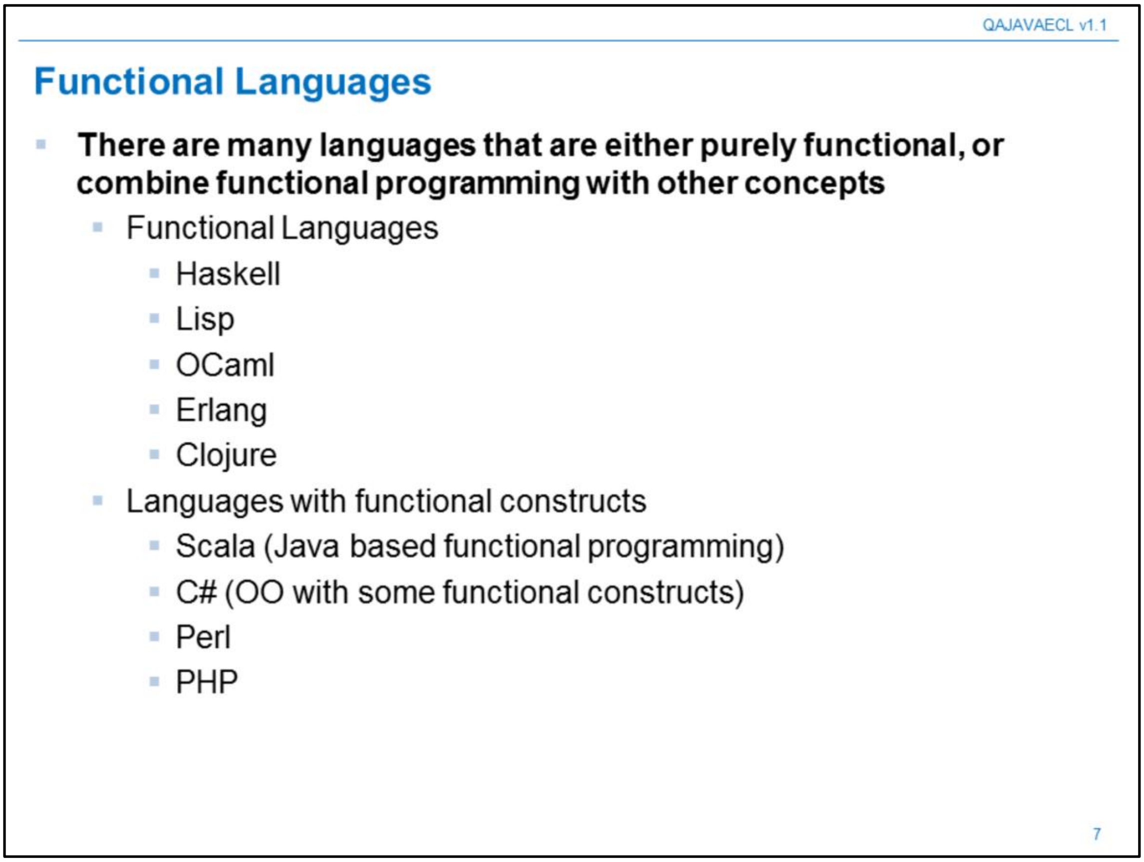
- Functions do not store state
- Functions can be composed
- No side effects
- Recursion focus
- Traits/Mixins
- Functions can be passed to methods

Object Oriented Programming

- Objects store state and methods manipulate this
- Methods can call other methods
- Side effects happen due to state or exceptions
- Loops (for/while)
- Classes are defined

6

This is not specific to Java!

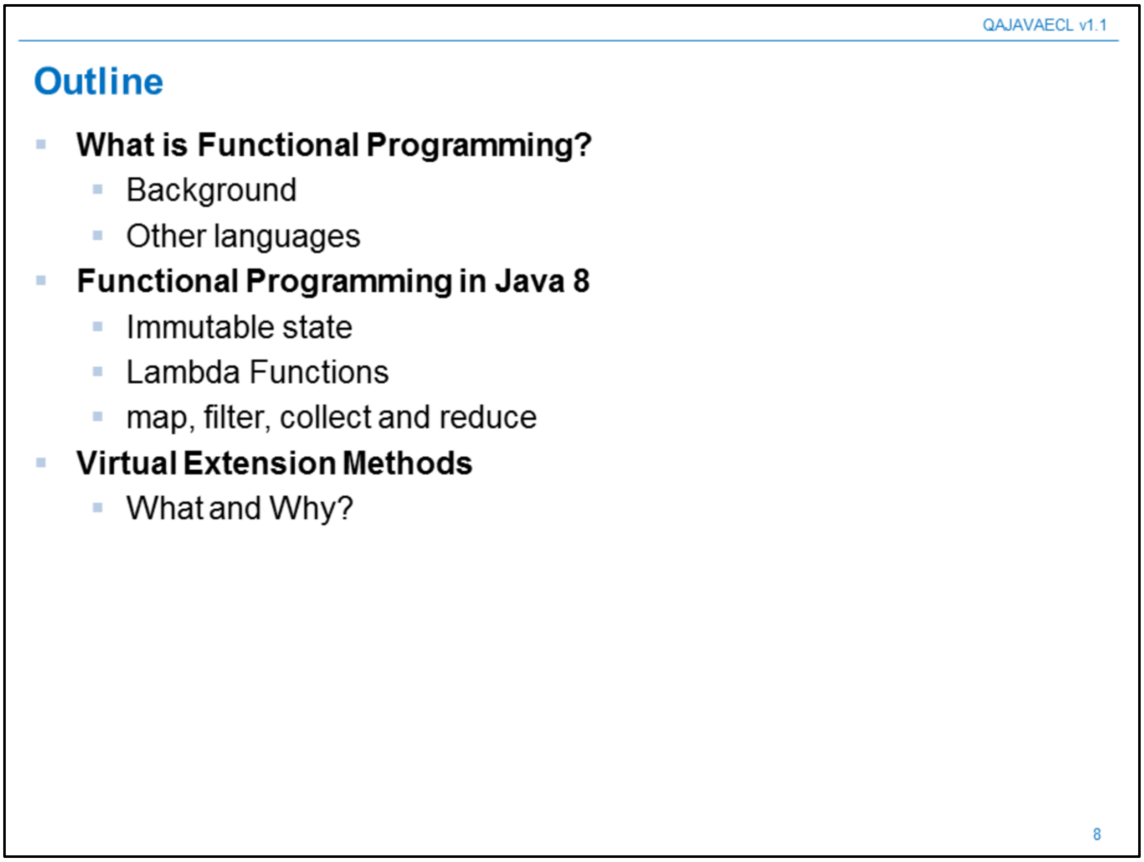


QAJAVAECL v1.1

Functional Languages

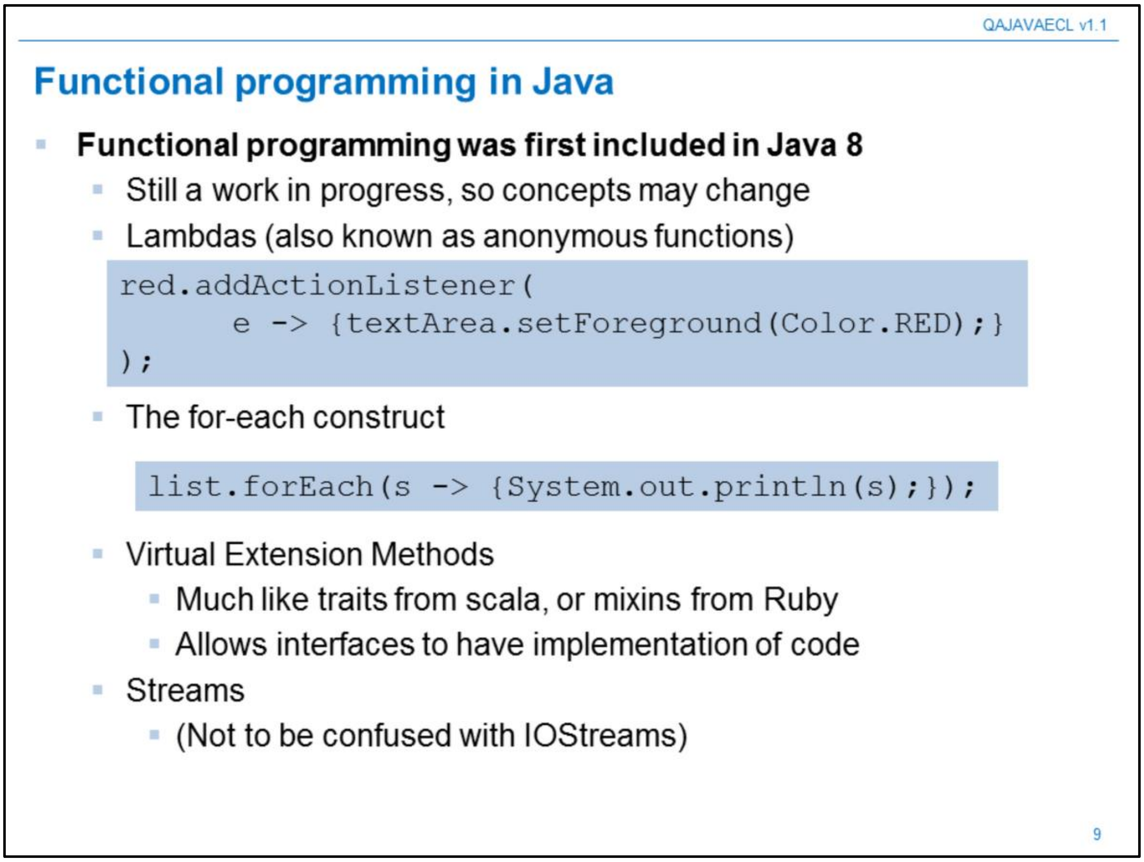
- **There are many languages that are either purely functional, or combine functional programming with other concepts**
  - Functional Languages
    - Haskell
    - Lisp
    - OCaml
    - Erlang
    - Clojure
  - Languages with functional constructs
    - Scala (Java based functional programming)
    - C# (OO with some functional constructs)
    - Perl
    - PHP

7



Outline

- **What is Functional Programming?**
  - Background
  - Other languages
- **Functional Programming in Java 8**
  - Immutable state
  - Lambda Functions
  - map, filter, collect and reduce
- **Virtual Extension Methods**
  - What and Why?



QAJAVAECL v1.1

Functional programming in Java

- **Functional programming was first included in Java 8**
  - Still a work in progress, so concepts may change
  - Lambdas (also known as anonymous functions)

```
red.addActionListener(
      e -> {textArea.setForeground(Color.RED);}
);
```

  - The for-each construct

```
list.forEach(s -> {System.out.println(s);});
```

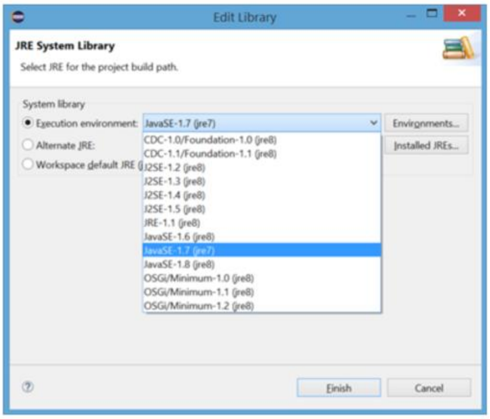
  - Virtual Extension Methods
    - Much like traits from scala, or mixins from Ruby
    - Allows interfaces to have implementation of code
  - Streams
    - (Not to be confused with IOStreams)

9

Lambdas can be used wherever we had anonymous inner classes before, but they remove some of the boiler plate.

Changing Java Version in Eclipse

- **The version of Java you are using can be changed in the properties for the folder**
  - Right click the project folder
  - Go to properties
  - Select "Java Build Path"
  - Go to the Libraries tab
  - Double click the library
- **If Java 8 isn't setup then click on environments and tell Eclipse where to find the java 8 JDK and JRE**



Edit Library

JRE System Library

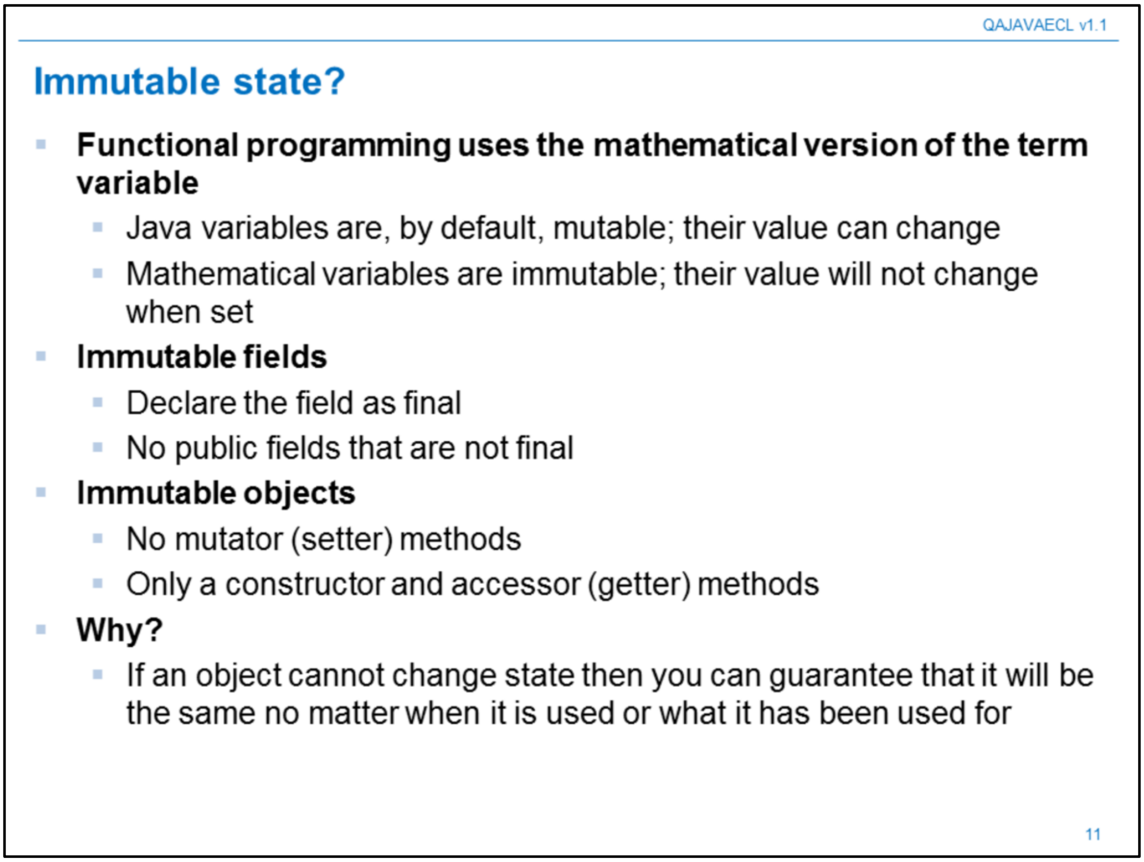
Select JRE for the project build path.

System library

● Execution environment: JavaSE-1.7 (jre7)

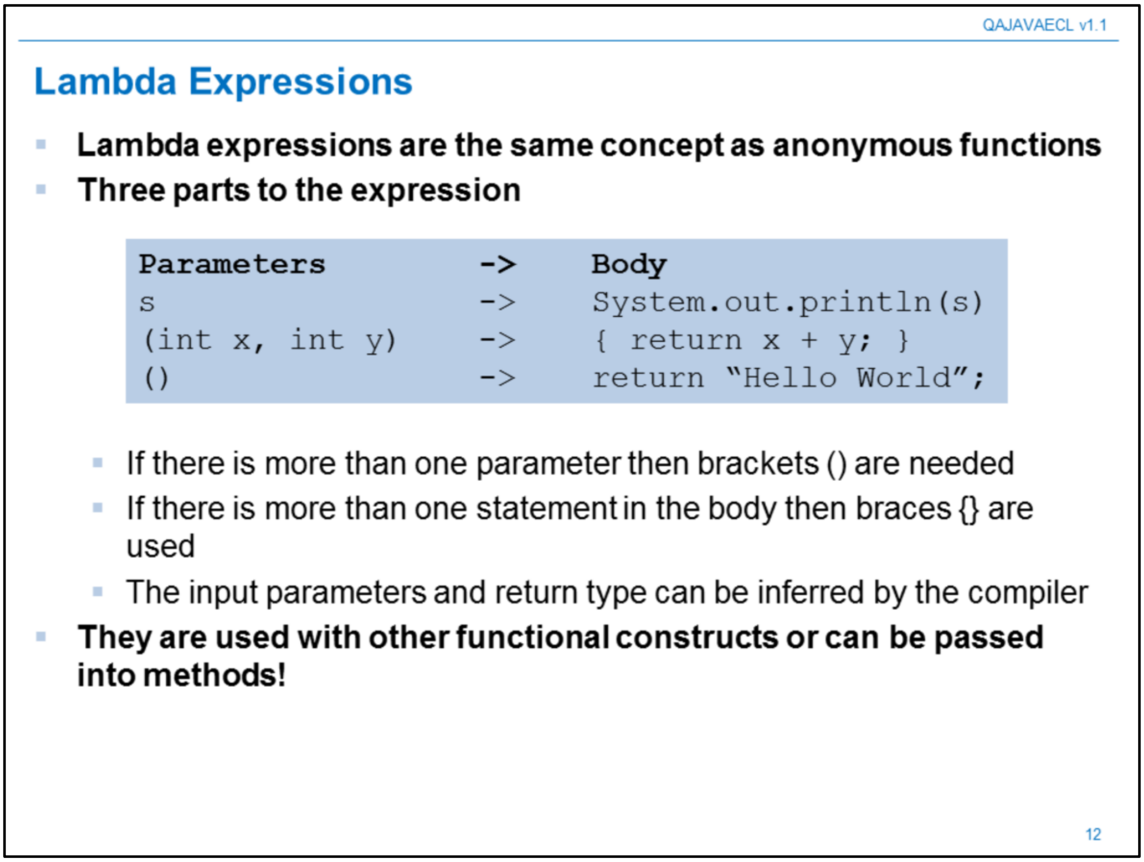
○ Alternate JRE:

○ Workspace default JRE (



Immutable state?

- **Functional programming uses the mathematical version of the term variable**
  - Java variables are, by default, mutable; their value can change
  - Mathematical variables are immutable; their value will not change when set
- **Immutable fields**
  - Declare the field as final
  - No public fields that are not final
- **Immutable objects**
  - No mutator (setter) methods
  - Only a constructor and accessor (getter) methods
- **Why?**
  - If an object cannot change state then you can guarantee that it will be the same no matter when it is used or what it has been used for



Lambda Expressions

- **Lambda expressions are the same concept as anonymous functions**
- **Three parts to the expression**

```
Parameters          ->     Body
s                   ->     System.out.println(s)
(int x, int y)      ->     { return x + y; }
()                  ->     return "Hello World";
```

  - If there is more than one parameter then brackets () are needed
  - If there is more than one statement in the body then braces {} are used
  - The input parameters and return type can be inferred by the compiler
- **They are used with other functional constructs or can be passed into methods!**

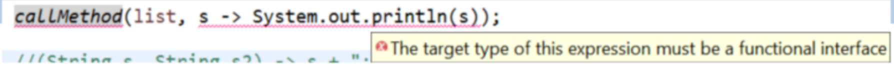
The type of a Lambda?

- How would we write the method for this call?

```
callMethod(list, s -> System.out.println(s));
```

- We could rely on Eclipse to automatically generate it for us, but IDEs are not always able to infer the correct type

```
private static void callMethod(ArrayList<String> stringList,
                                                  Object object) {
        // TODO Auto-generated method stub
}
```



```
callMethod(list, s -> System.out.println(s));
//(String s, String s2) -> s + "
```

The target type of this expression must be a functional interface

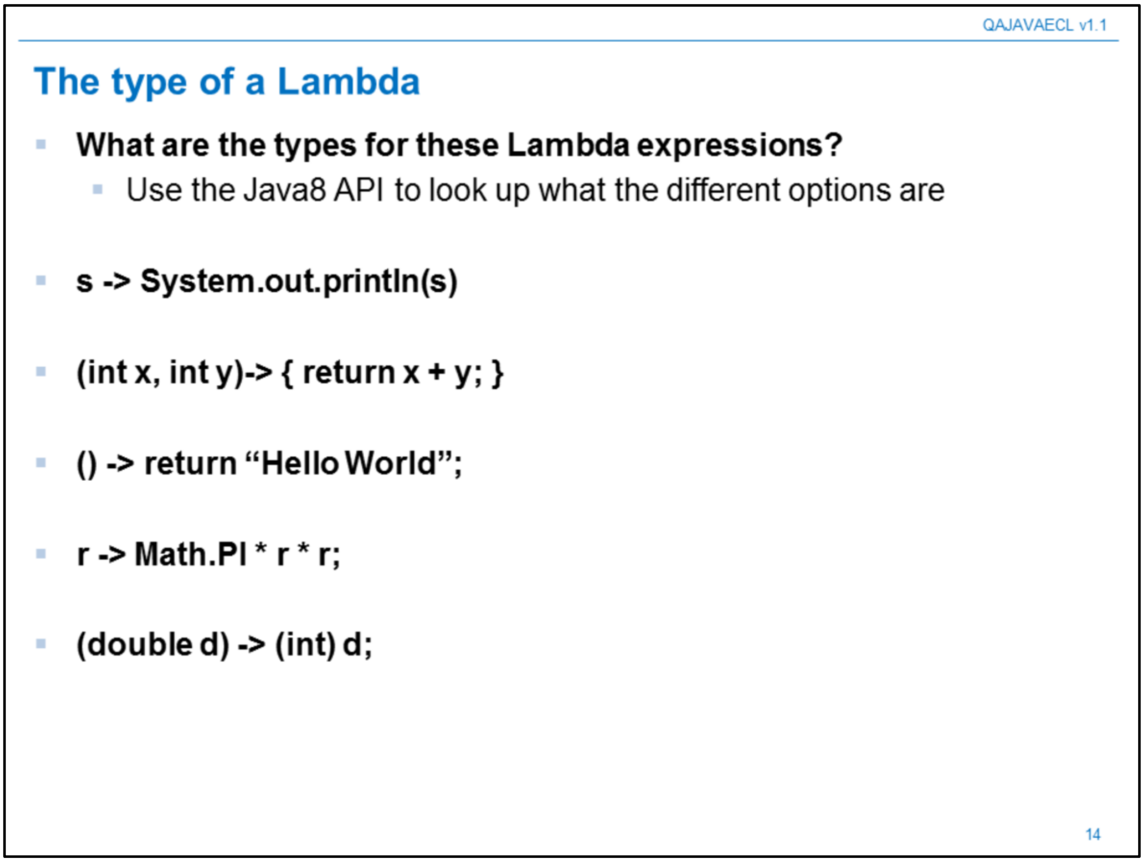
- java.util.function has a set of new object types which apply in this situation

```
private static void callMethod(
                ArrayList<String> string,
                Consumer<? super String> p) {
```

There are many different types available at

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

As of the 8/1/2015 Eclipse does not correctly infer types for lambda expressions, it is on their list of ‘things to do’ for future updates.

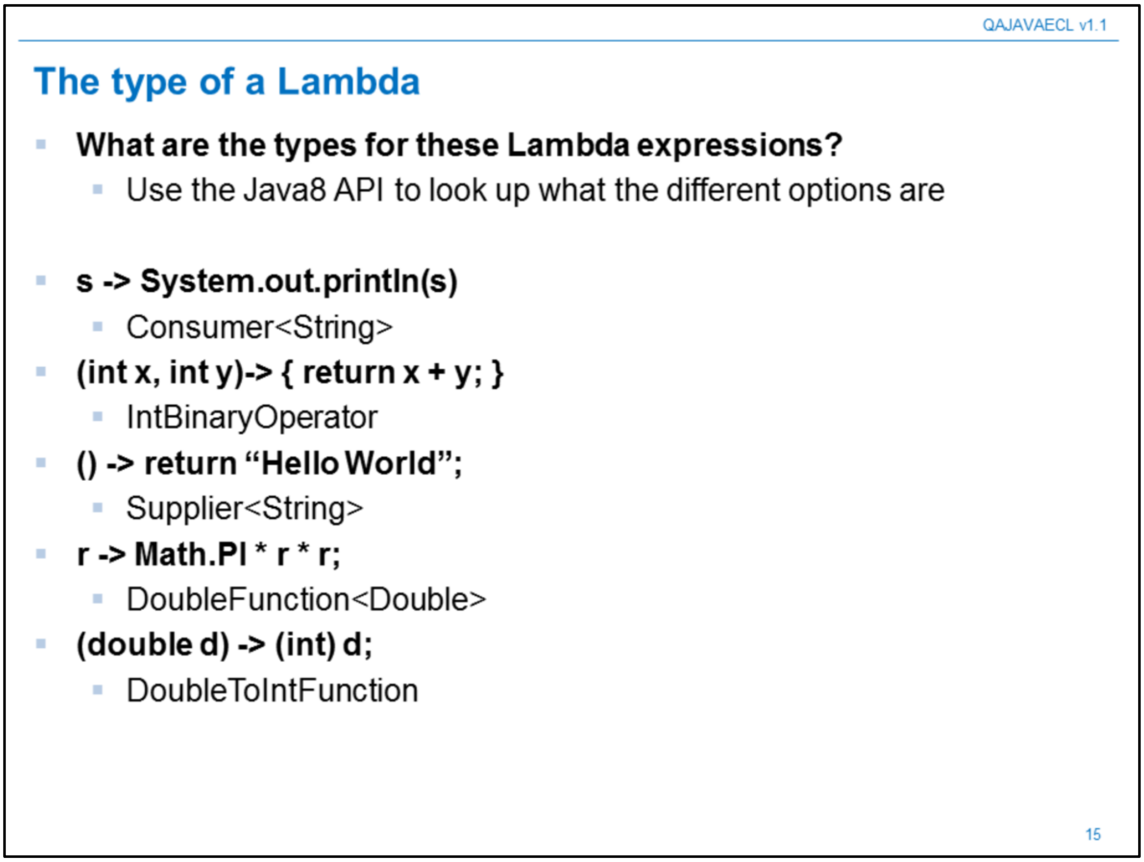


QAJAVAECL v1.1

The type of a Lambda

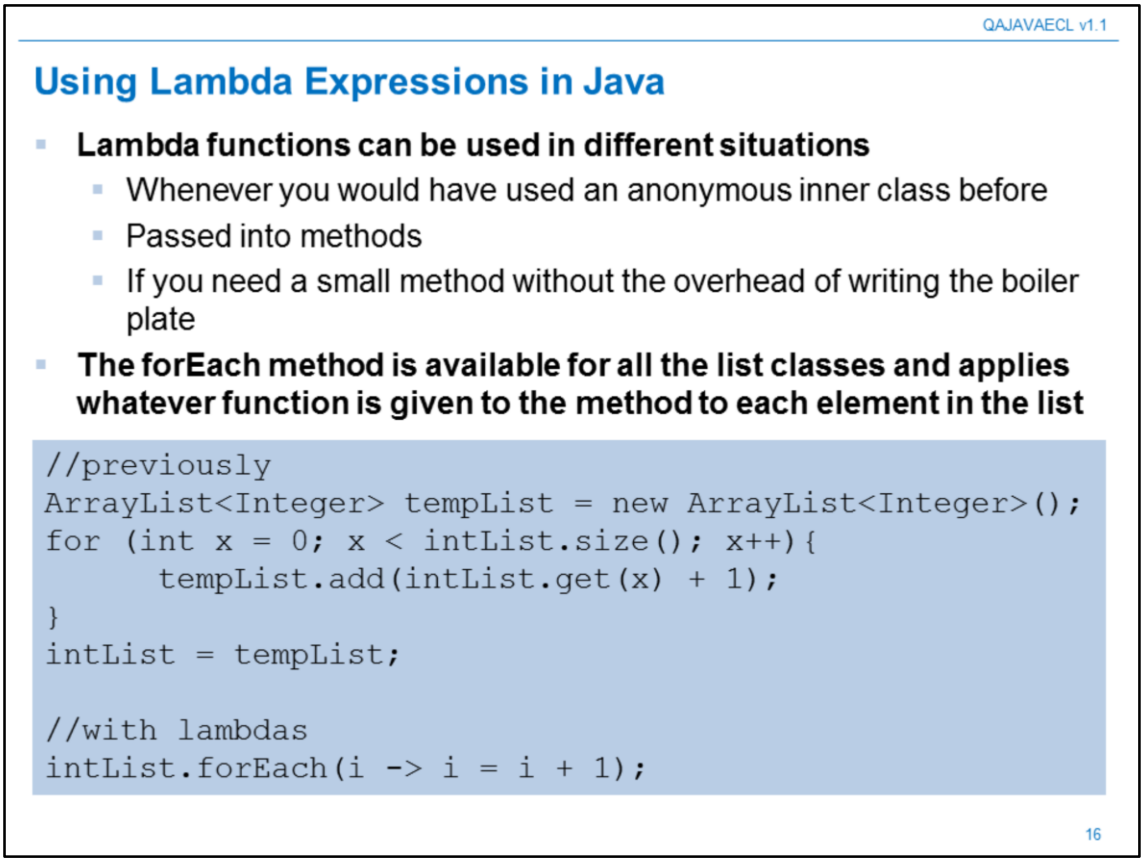
- **What are the types for these Lambda expressions?**
  - Use the Java8 API to look up what the different options are
- **s -> System.out.println(s)**
- **(int x, int y)-> { return x + y; }**
- **() -> return “Hello World”;**
- **r -> Math.PI * r * r;**
- **(double d) -> (int) d;**

14



The type of a Lambda

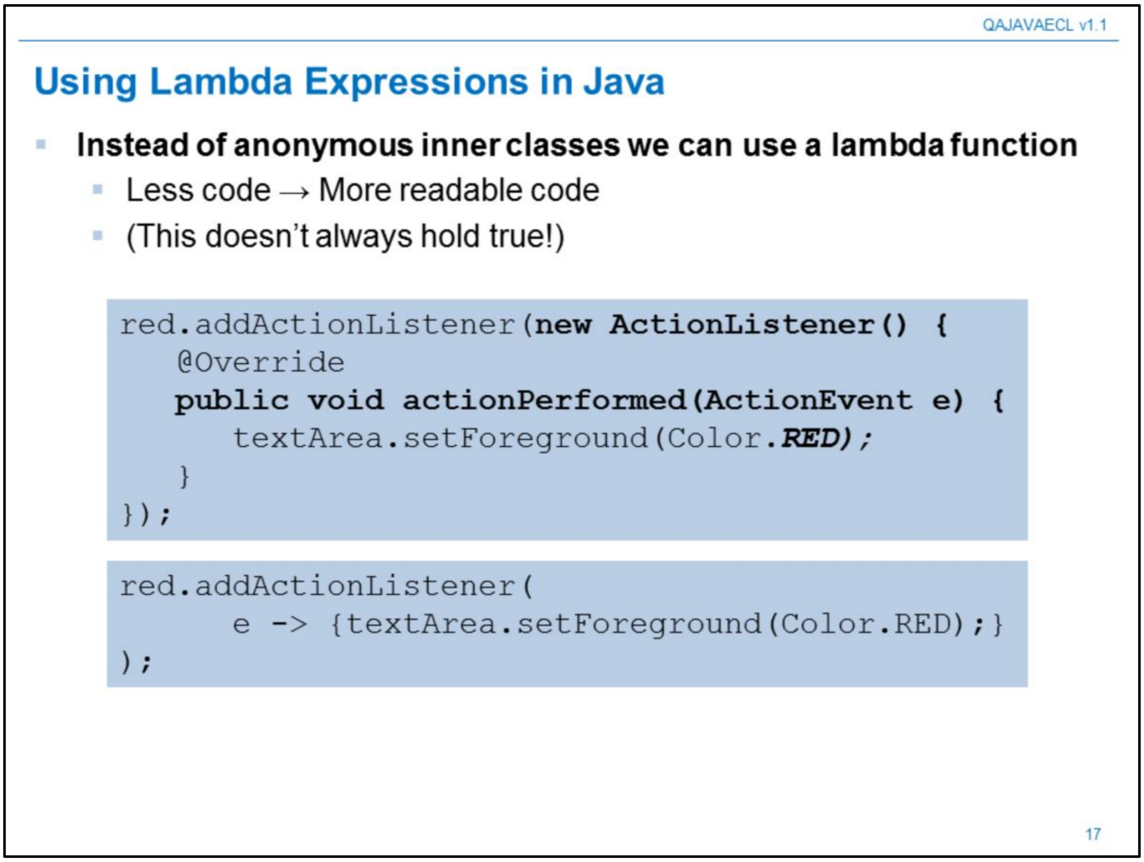
- **What are the types for these Lambda expressions?**
  - Use the Java8 API to look up what the different options are
- **s -> System.out.println(s)**
  - Consumer<String>
- **(int x, int y)-> { return x + y; }**
  - IntBinaryOperator
- **() -> return “Hello World”;**
  - Supplier<String>
- **r -> Math.PI * r * r;**
  - DoubleFunction<Double>
- **(double d) -> (int) d;**
  - DoubleToIntFunction



Using Lambda Expressions in Java

- **Lambda functions can be used in different situations**
  - Whenever you would have used an anonymous inner class before
  - Passed into methods
  - If you need a small method without the overhead of writing the boiler plate
- **The forEach method is available for all the list classes and applies whatever function is given to the method to each element in the list**

```
//previously
ArrayList<Integer> tempList = new ArrayList<Integer>();
for (int x = 0; x < intList.size(); x++){
      tempList.add(intList.get(x) + 1);
}
intList = tempList;

//with lambdas
intList.forEach(i -> i = i + 1);
```



QAJAVAECL v1.1

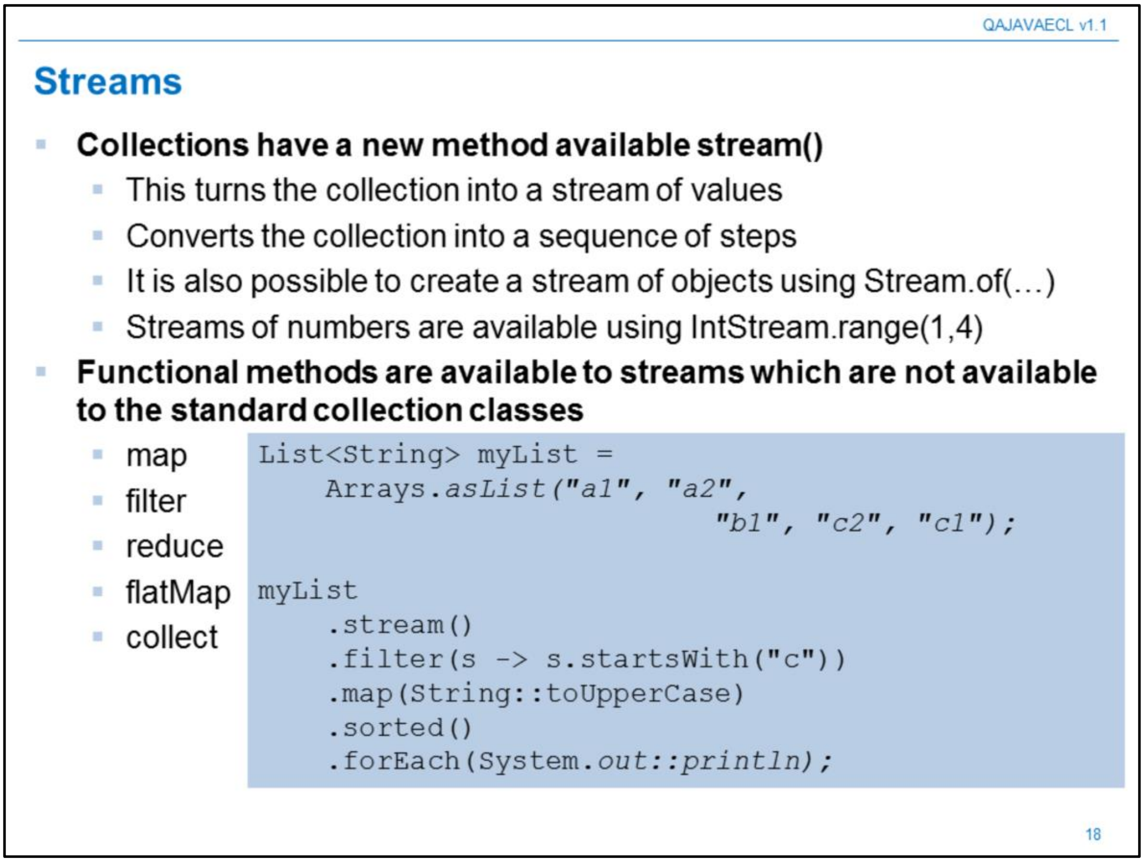
Using Lambda Expressions in Java

- **Instead of anonymous inner classes we can use a lambda function**
  - Less code → More readable code
  - (This doesn't always hold true!)

```
red.addActionListener(new ActionListener() {
   @Override
   public void actionPerformed(ActionEvent e) {
      textArea.setForeground(Color.RED);
   }
});
```

```
red.addActionListener(
      e -> {textArea.setForeground(Color.RED);}
);
```

17



QAJAVAECL v1.1

Streams

- **Collections have a new method available stream()**
  - This turns the collection into a stream of values
  - Converts the collection into a sequence of steps
  - It is also possible to create a stream of objects using Stream.of(…)
  - Streams of numbers are available using IntStream.range(1,4)
- **Functional methods are available to streams which are not available to the standard collection classes**
  - map
  - filter
  - reduce
  - flatMap
  - collect

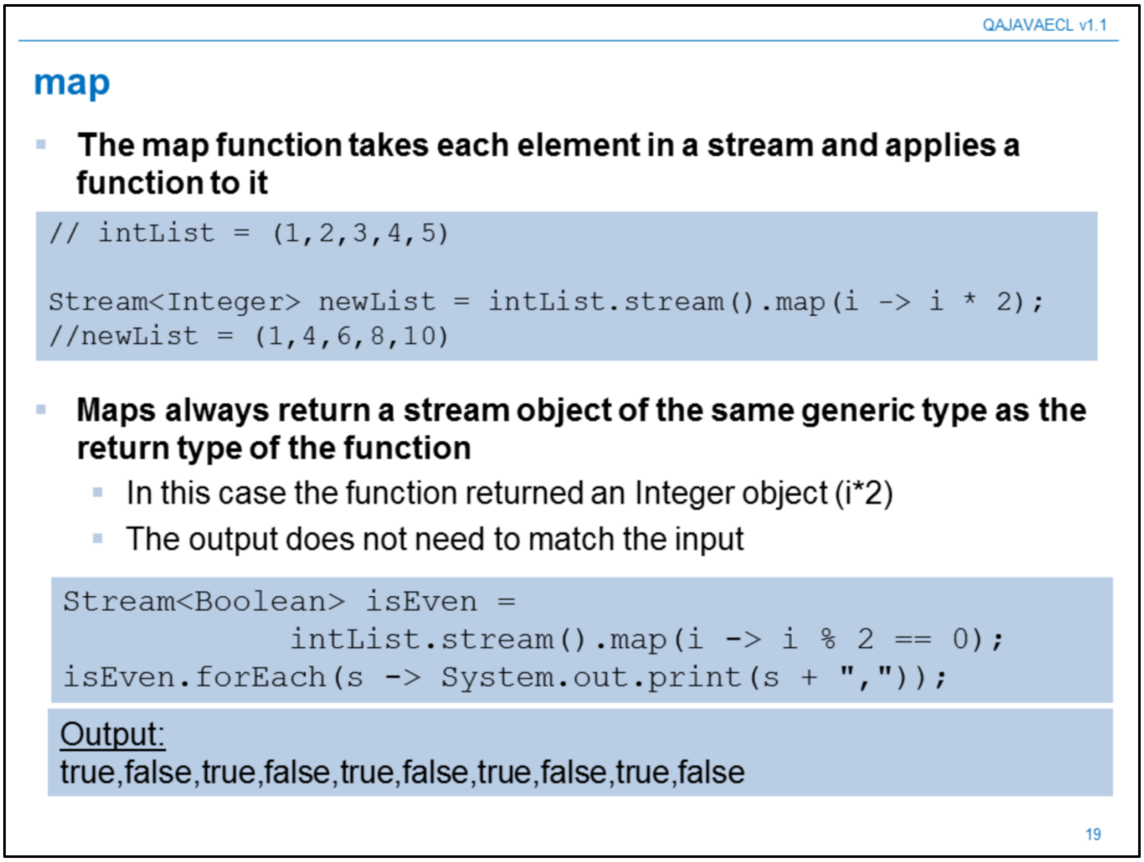
```
List<String> myList =
    Arrays.asList("a1", "a2",
                           "b1", "c2", "c1");

myList
    .stream()
    .filter(s -> s.startsWith("c"))
    .map(String::toUpperCase)
    .sorted()
    .forEach(System.out::println);
```

18

Example used from:http://winterbe.com/posts/2014/07/31/java8-stream-tutorial-examples/

This is a good set of examples and explanation about streams in Java8

None of the methods used change the state of myList, so they can be chained together with the output of one operation feeding into the next.



QAJAVAECL v1.1

map

- **The map function takes each element in a stream and applies a function to it**

```
// intList = (1,2,3,4,5)

Stream<Integer> newList = intList.stream().map(i -> i * 2);
//newList = (1,4,6,8,10)
```

- **Maps always return a stream object of the same generic type as the return type of the function**
  - In this case the function returned an Integer object (i*2)
  - The output does not need to match the input

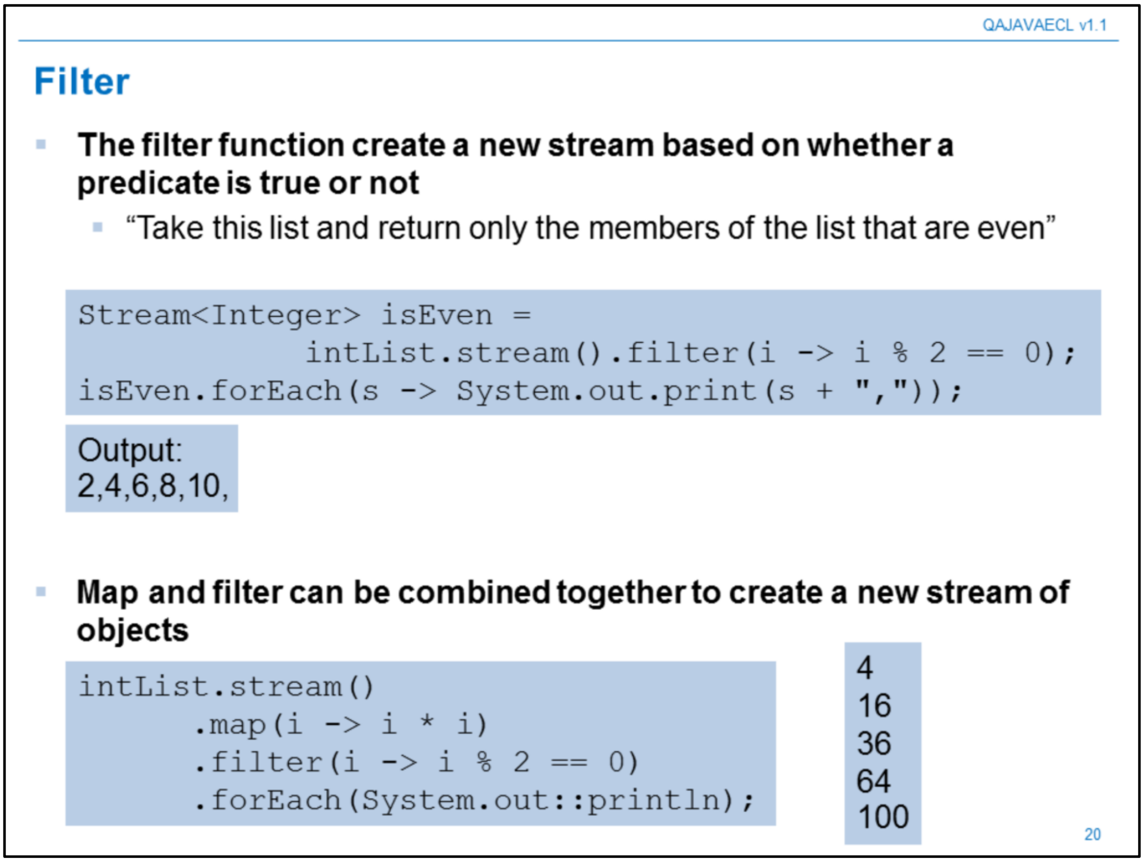
```
Stream<Boolean> isEven =
              intList.stream().map(i -> i % 2 == 0);
isEven.forEach(s -> System.out.print(s + ","));
```

Output:

```
true,false,true,false,true,false,true,false,true,false
```

19

The collection class Map should not be confused with the functional method map. Look for the capitalisation of the word!



QAJAVAECL v1.1

Filter

- **The filter function create a new stream based on whether a predicate is true or not**
  - “Take this list and return only the members of the list that are even”

```
Stream<Integer> isEven =
            intList.stream().filter(i -> i % 2 == 0);
isEven.forEach(s -> System.out.print(s + ","));
```

```
Output:
2,4,6,8,10,
```

- **Map and filter can be combined together to create a new stream of objects**

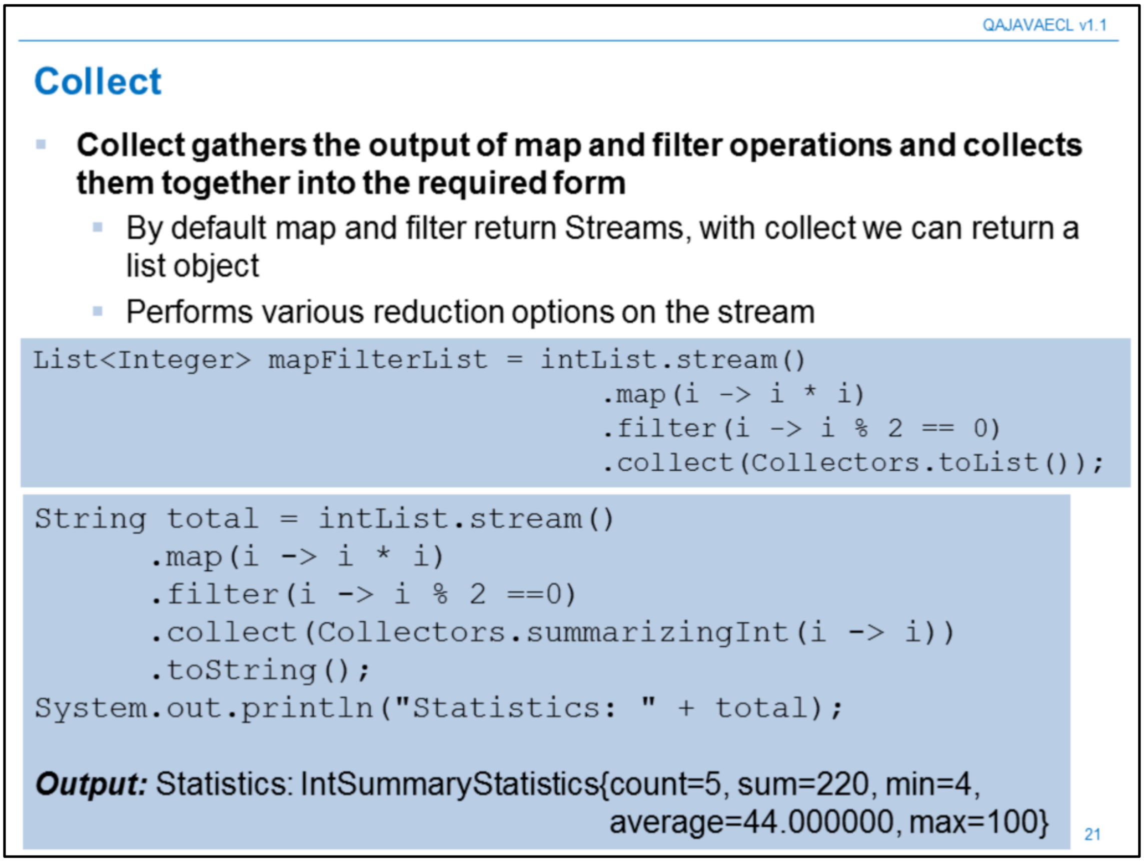
```
intList.stream()
      .map(i -> i * i)
      .filter(i -> i % 2 == 0)
      .forEach(System.out::println);
```

```
4
16
36
64
100
```

20

System.out::println calls the println static method on all the elements in the mapped and filtered intList. This is an instance of a terminal operation, it closes the stream after it has completed, so nothing else can operate on the stream.

To get around this we would create a new, separate stream.



QAJAVAECL v1.1

Collect

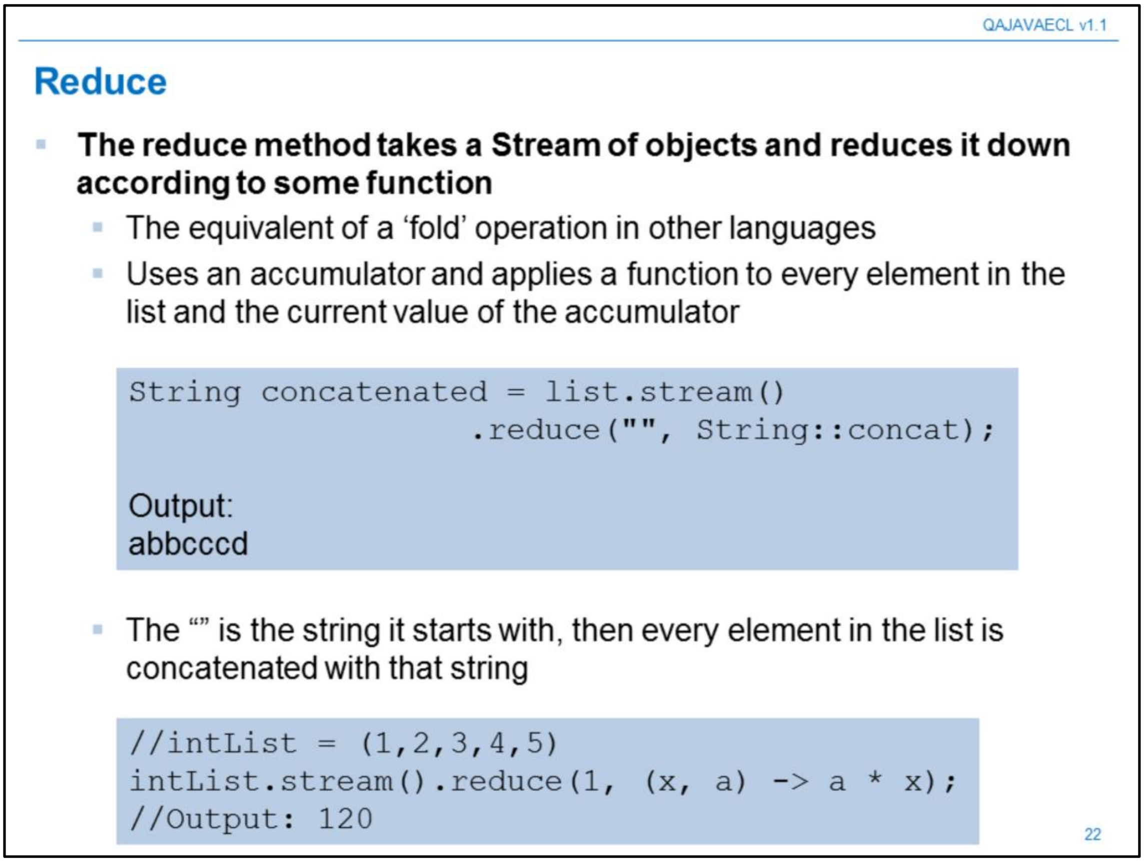
- **Collect gathers the output of map and filter operations and collects them together into the required form**
  - By default map and filter return Streams, with collect we can return a list object
  - Performs various reduction options on the stream

```
List<Integer> mapFilterList = intList.stream()
                                     .map(i -> i * i)
                                     .filter(i -> i % 2 == 0)
                                     .collect(Collectors.toList());
```

```
String total = intList.stream()
      .map(i -> i * i)
      .filter(i -> i % 2 ==0)
      .collect(Collectors.summarizingInt(i -> i))
      .toString();
System.out.println("Statistics: " + total);
```

Output: Statistics: IntSummaryStatistics{count=5, sum=220, min=4, average=44.000000, max=100}

21



QAJAVAECL v1.1

Reduce

- **The reduce method takes a Stream of objects and reduces it down according to some function**
  - The equivalent of a ‘fold’ operation in other languages
  - Uses an accumulator and applies a function to every element in the list and the current value of the accumulator

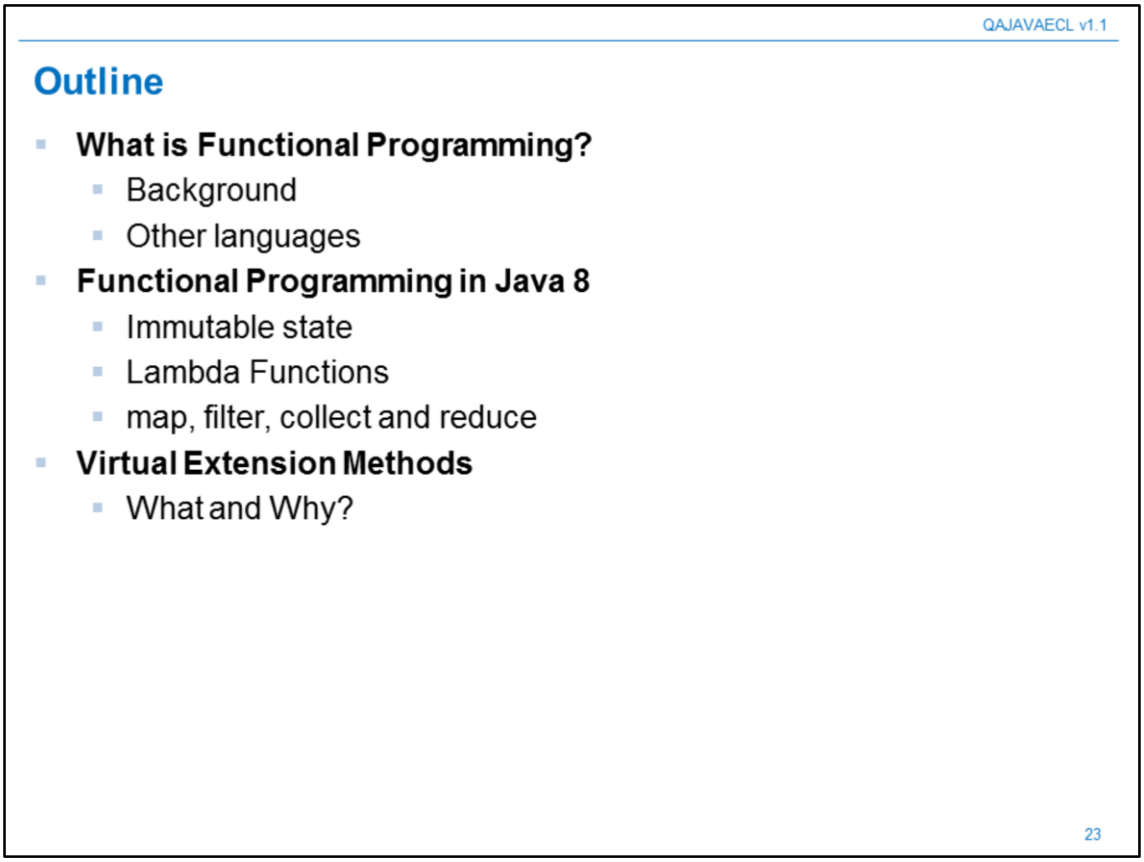
```
String concatenated = list.stream()
                          .reduce("", String::concat);
```

Output:
abbcccd

  - The “” is the string it starts with, then every element in the list is concatenated with that string

```
//intList = (1,2,3,4,5)
intList.stream().reduce(1, (x, a) -> a * x);
//Output: 120
```

22

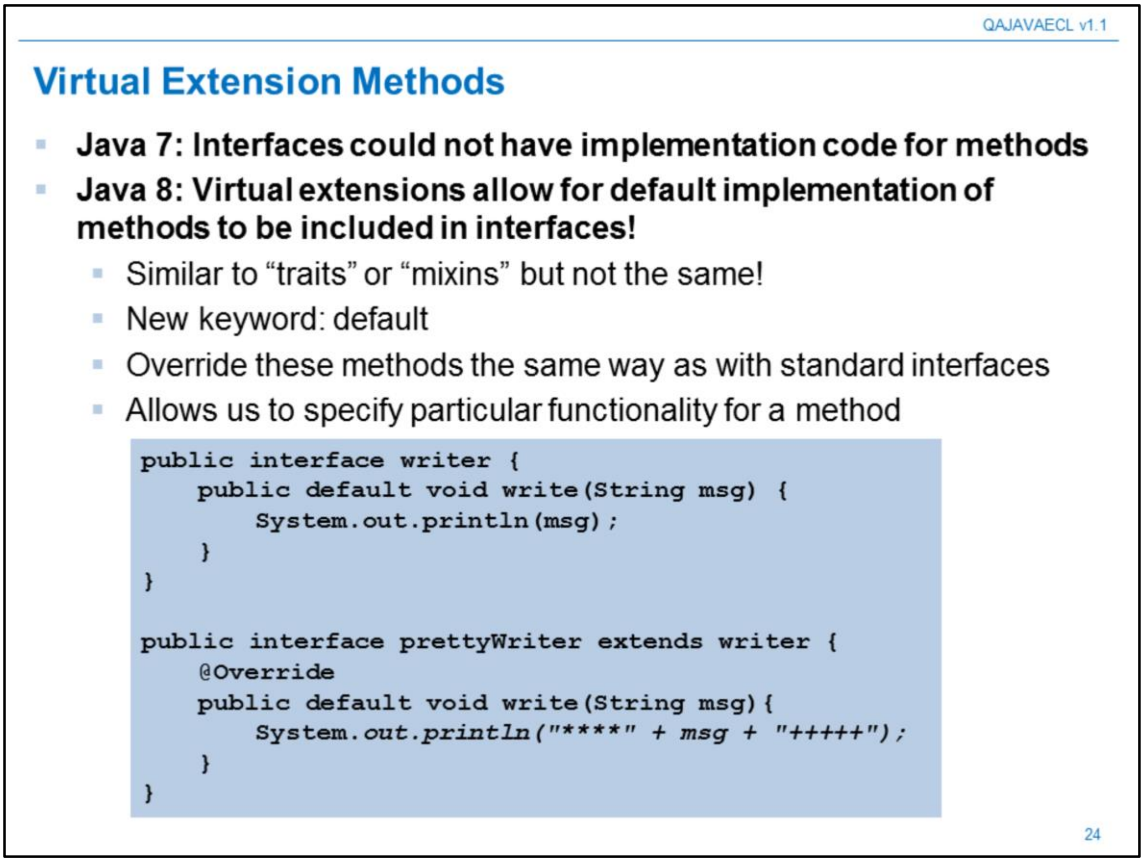


QAJAVAECL v1.1

Outline

- **What is Functional Programming?**
  - Background
  - Other languages
- **Functional Programming in Java 8**
  - Immutable state
  - Lambda Functions
  - map, filter, collect and reduce
- **Virtual Extension Methods**
  - What and Why?

23



QAJAVAECL v1.1

Virtual Extension Methods

- **Java 7: Interfaces could not have implementation code for methods**
- **Java 8: Virtual extensions allow for default implementation of methods to be included in interfaces!**
  - Similar to “traits” or “mixins” but not the same!
  - New keyword: default
  - Override these methods the same way as with standard interfaces
  - Allows us to specify particular functionality for a method

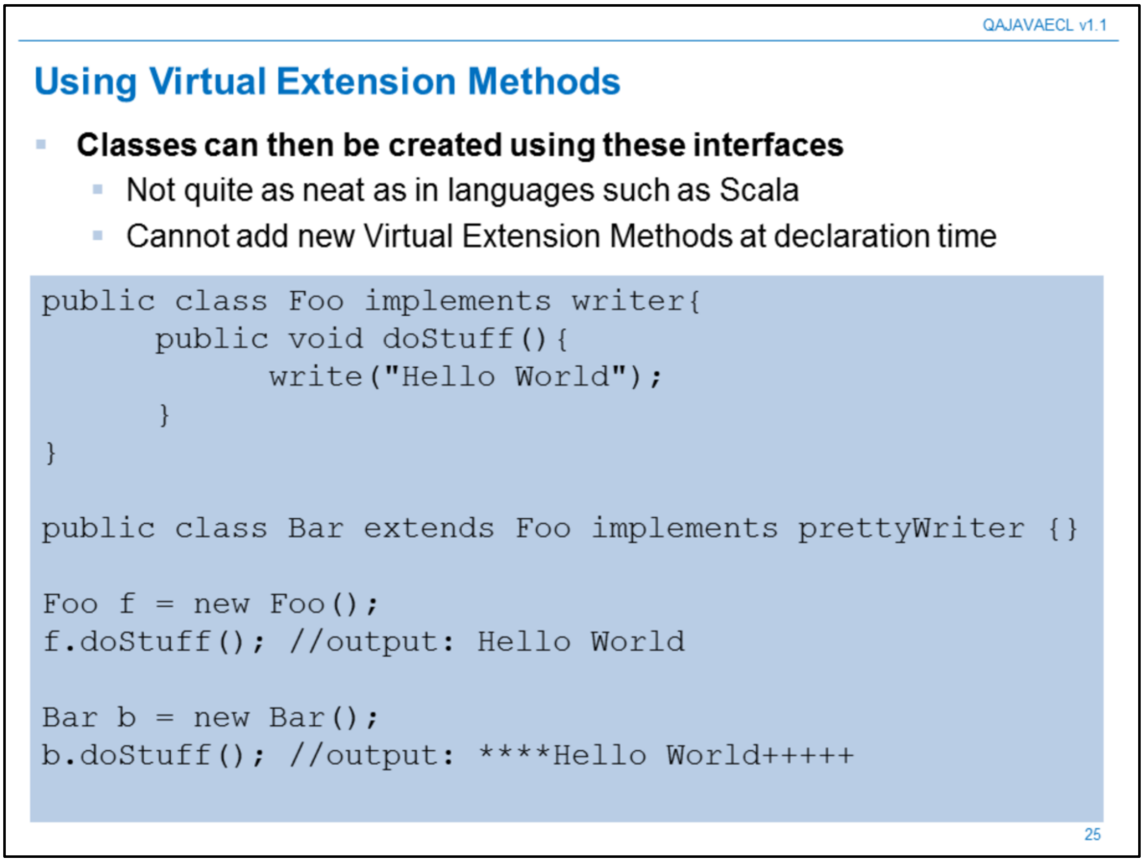
```
public interface writer {
    public default void write(String msg) {
        System.out.println(msg);
    }
}

public interface prettyWriter extends writer {
    @Override
    public default void write(String msg){
        System.out.println("****" + msg + "+++++");
    }
}
```

24

Sometimes known as ‘defender methods’

They allow interfaces to be changed without affecting the sub-classes that rely on the interfaces by providing a default operation for any new method.

http://viralpatel.net/blogs/java-8-default-methods-tutorial/ gives a good tutorial on how virtual methods work



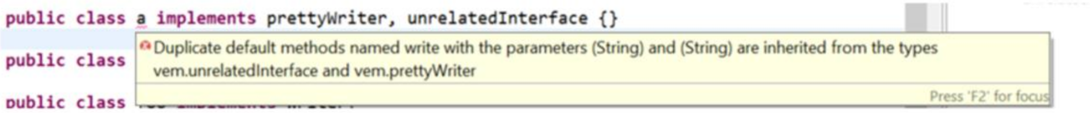
Using Virtual Extension Methods

- **Classes can then be created using these interfaces**
  - Not quite as neat as in languages such as Scala
  - Cannot add new Virtual Extension Methods at declaration time

```
public class Foo implements writer{
      public void doStuff(){
            write("Hello World");
      }
}

public class Bar extends Foo implements prettyWriter {}

Foo f = new Foo();
f.doStuff(); //output: Hello World

Bar b = new Bar();
b.doStuff(); //output: ****Hello World+++++
```

Methods with the same name

- **If you have two unrelated interfaces (one does not extend the other) that both have methods with the same name, your code will not compile**

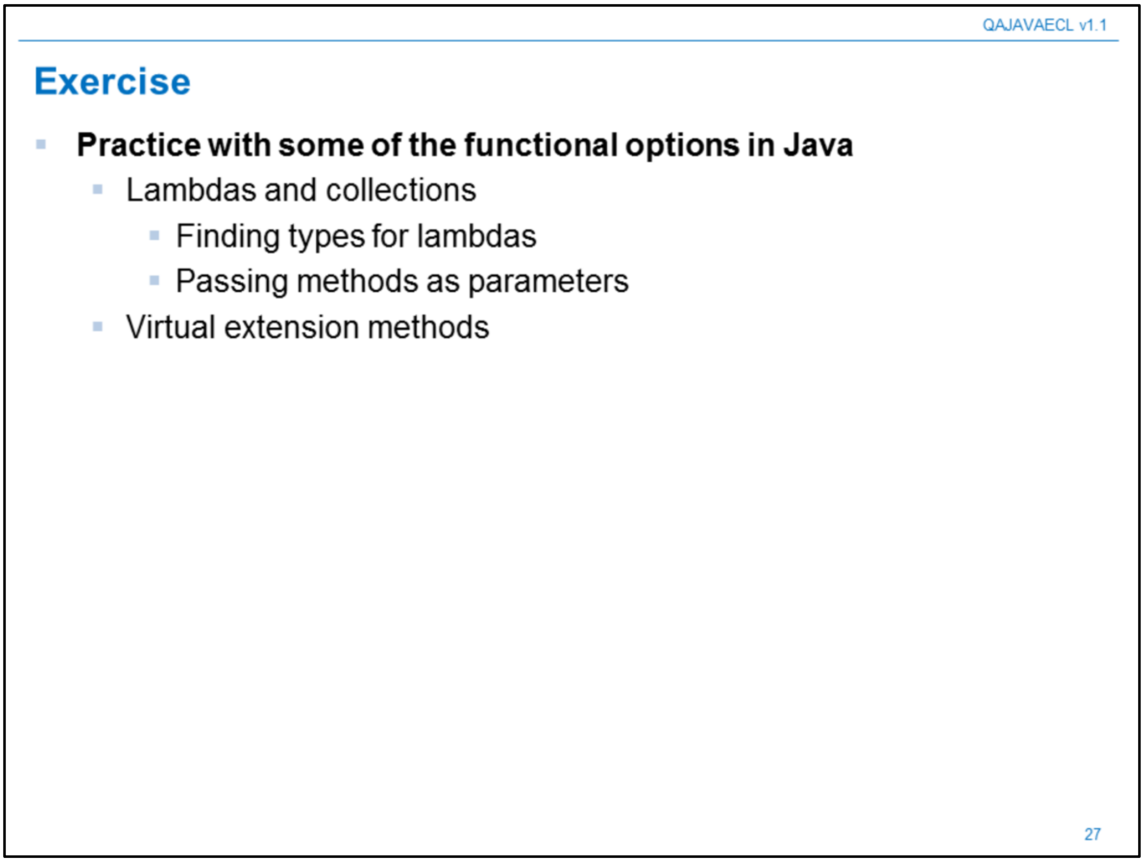


```
public class a implements prettyWriter, unrelatedInterface {}

public class

public class
```

Duplicate default methods named write with the parameters (String) and (String) are inherited from the types vem.unrelatedInterface and vem.prettyWriter

- **To solve this problem we need to override the write() method in the class a and tell it specifically what to do**
  - Use the super call to access the methods in the UnrelatedInterface interface

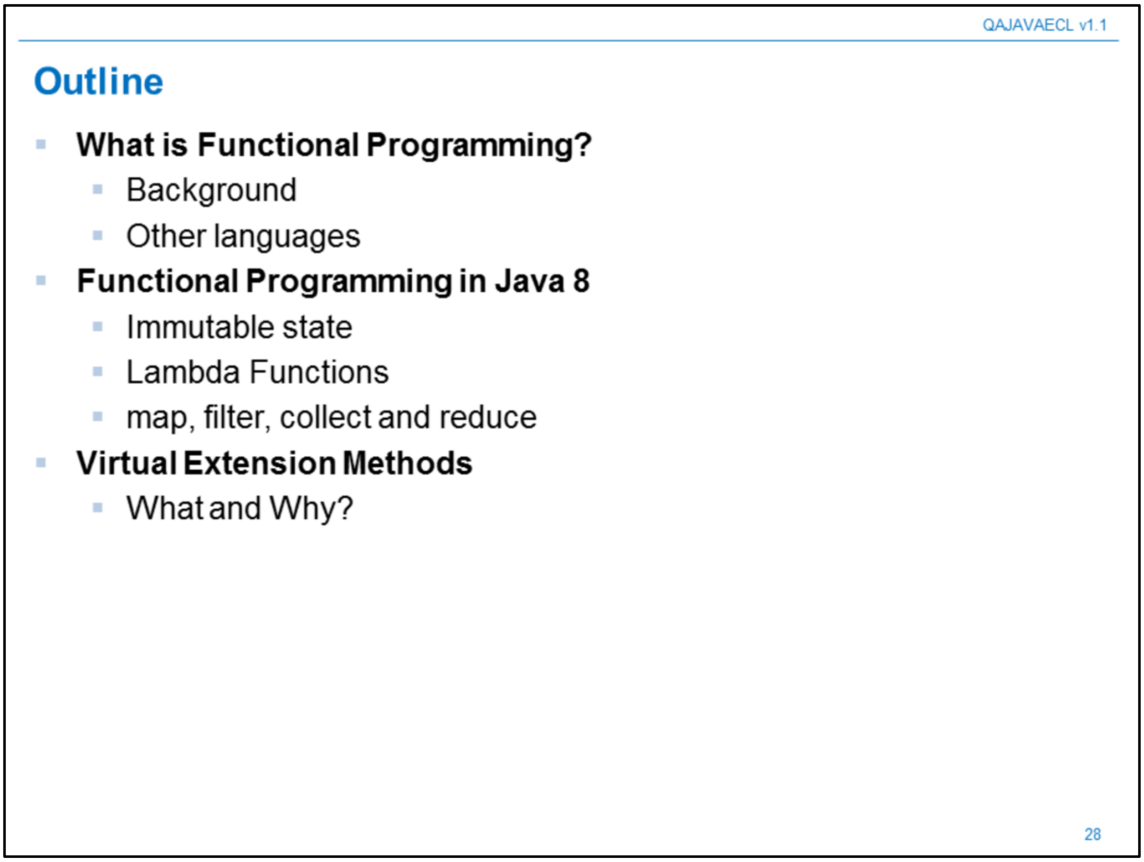
```
public class A implements prettyWriter, UnrelatedInterface {
        public void write(String msg){
                UnrelatedInterface.super.write(msg);
        }
}
```



QAJAVAECL v1.1

Exercise

- **Practice with some of the functional options in Java**
  - Lambdas and collections
    - Finding types for lambdas
    - Passing methods as parameters
  - Virtual extension methods



QAJAVAECL v1.1

Outline

- **What is Functional Programming?**
  - Background
  - Other languages
- **Functional Programming in Java 8**
  - Immutable state
  - Lambda Functions
  - map, filter, collect and reduce
- **Virtual Extension Methods**
  - What and Why?

28