Lambda Expressions in Java 8

COP 4331/5339 – OODP/OOSD

Instructor: Dr. Ionut Cardei

Reading material:

- Lambda Expressions in Java 8, by Cay Horstmann (Dr. Dobbs): http://www.drdobbs.com/jvm/lambda-expressions-in-java-8/240166764
- Java Tutorial: https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html
- Lambda Quick Start: http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/Lambda-QuickStart/index.html
- Lambda Expressions in Java, by Angelika Langer & Klaus Kreft http://www.angelikalanger.com/Lambdas/Lambdas.pdf

Overview

- What is a λ-expression ?
- Functional Interfaces
- Motivation for Lambdas
- Lambda Expression for Comparator
- Lambda Expression Syntax
- Closures and Variable Access
- Dealing with Exceptions
- Method References
- Constructor References
- Further Reading

What is a λ-expression ?

- A block of code with parameters, i.e. a function with no name that can access *effectively final* variables from the surrounding scope (lexically scoped)
- Captures free variables, instance and class variables from the enclosing scope
- Implemented with a **closure**.
- Lambdas can be stored, referenced by variables and passed around for execution at a later time

What is a λ-expression ?

- It can be used anywhere where an object implementing a ***functional interface*** is expected
- Other languages supporting lambdas:
  - JavaScript, C++, Python, Scala, LISP, Scheme, Haskell, Matlab, F#, Perl, Lua, PHP, Ruby, TCL
  - Introduced by functional languages
- Java is very late to this party

Functional Interfaces

- A Java interface with **only one abstract (i.e. unimplemented) method** declared in its definition
- Optional Java annotation: @*FunctionalInterface*
- Examples:

```
public interface Comparable<T>  {
  int compareTo(T other);
}
```

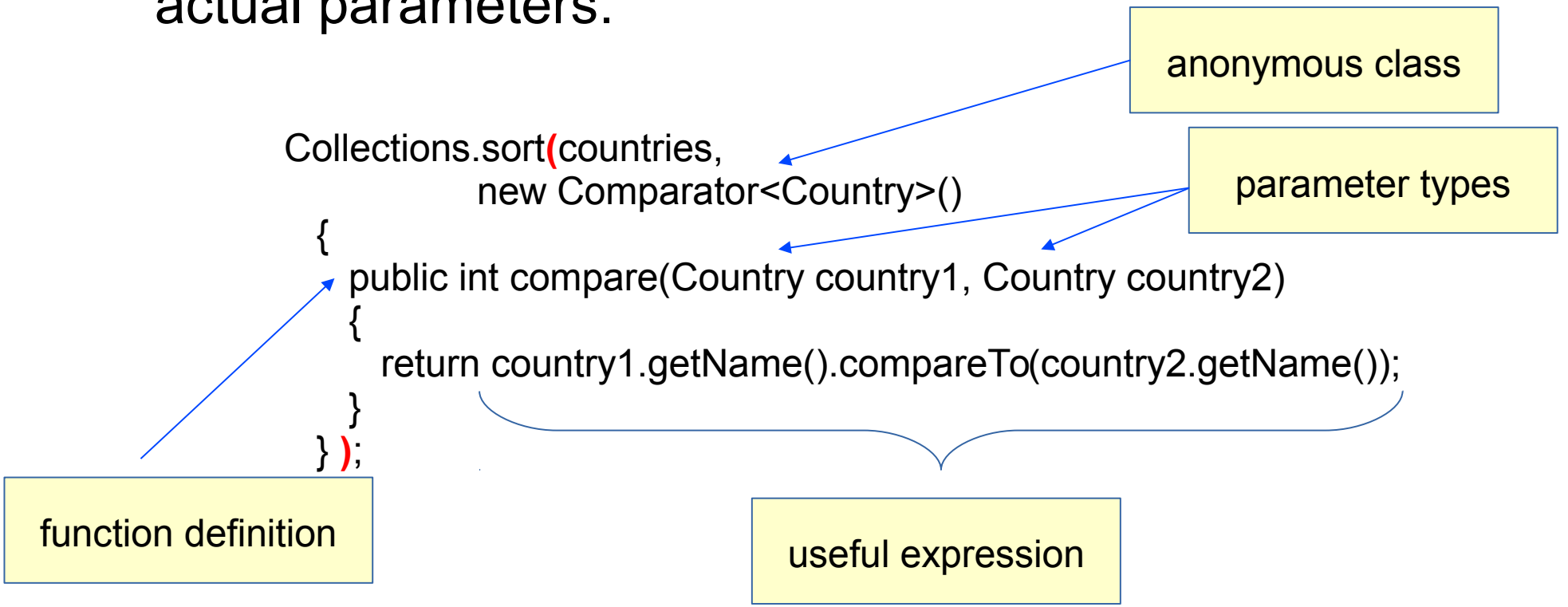
```
public interface Comparator<T>  {
  int compare(T p1, T p2);
}
```

```
@FunctionalInterface
public interface ActionListener
{
   int actionPerformed(ActionEvent event);
}
```

```
public interface Runnable
{
   void run();
}
```

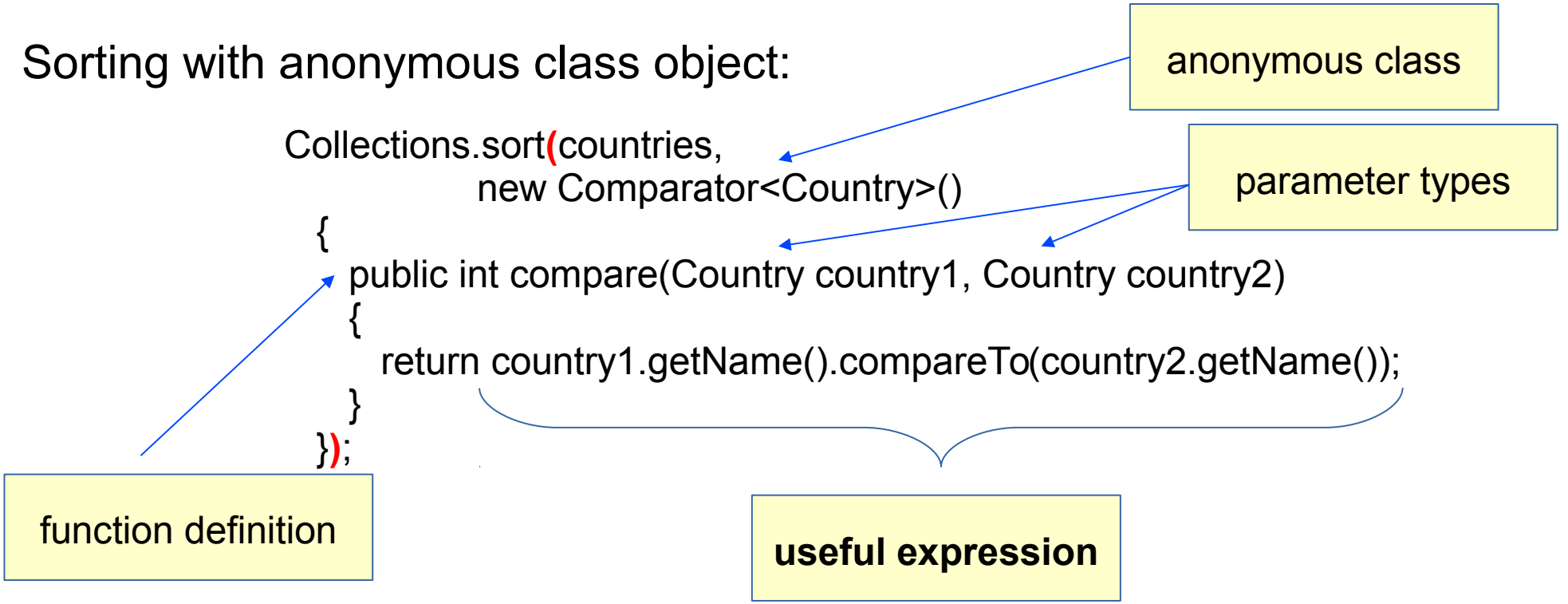
Motivation for Lambdas

- Using named or anonymous classes for instances of functional interfaces is verbose
- The compiler can infer parameter and return types from actual parameters.

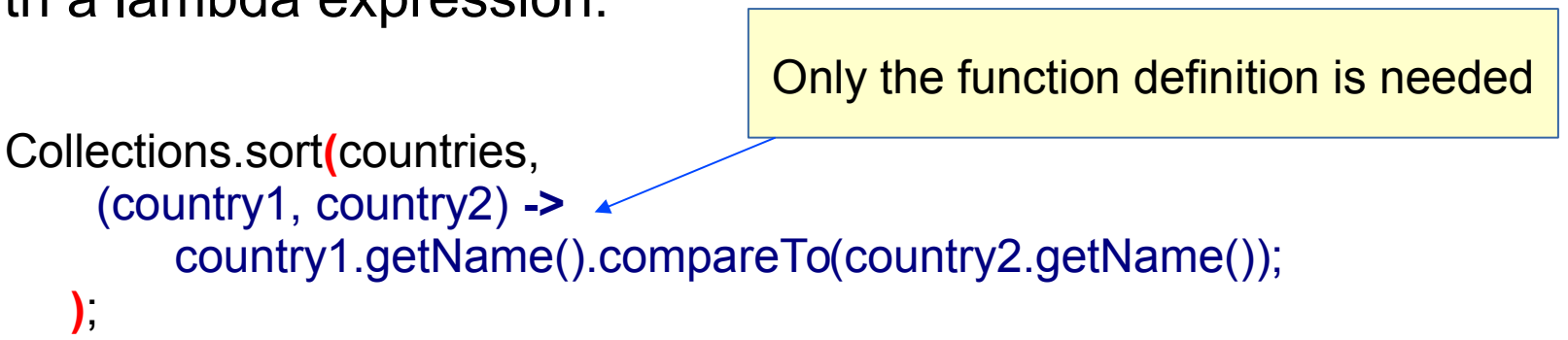


Motivation for Lambdas

Sorting with anonymous class object:



Same thing, with a lambda expression:



Motivation for Lambdas

- Less typing, more convenient
- More readable code
- Support for a **stream** abstraction where objects are processes in a pipeline mode (and not related to the java.io I/O classes).
- Support for lazy evaluation and optimization
  - delayed object creation, on demand
  - implicit parallelization of code, convenient for programmers, with parallel streams
- Support for ***Fluent Programming***, i.e. chaining an operation that returns a value followed by an operation that uses that value and returns a new value, and so on.

Lambda Expression for Comparator

- Same example, now with a lambda-expression (in blue color):

```
Collections.sort(countries,
      (Country country1, Country country2) ->
            country1.getName().compareTo(country2.getName());
    );
```

- If the compiler can infer parameter types, we can omit them:

```
Collections.sort(countries,
      (country1, country2) ->
            country1.getName().compareTo(country2.getName());
    );
```

Lambda Expression Syntax

- For a code block of one statement:

  (paramlist) -> expression

- For a code block of multiple statements:

  (paramlist) -> {
  statement$_1$; …....
  statement$_n$;
  }

  - If the lambda returns a statement: use *return expr;*
  - *(paramlist) is (Type$_1$ p$_1$, … , Type$_n$ p$_n$)*

- If the parameter list has just one parameter with inferred type, you can omit the parentheses
- No need to specify return type

Lambda Expression with Multiple Statements

- A Comparator<String> comparing string lengths

```
Collections.sort(stringlist,
    (String s1, String s2) -> {
        if (s1.length() < s2.length()) return -1;
        if (s1.length() > s2.length()) return +1;
        return 0;
    } );
```

- An ActionListener (from Chapter 4):

```
helloButton.addListener(
    event -> {      // or (event)   or   (ActionEvent event)
        String s = jText1.getText();
        System.out.println("the text was " + s);
        jText1.setText("Hello World");
    } );
```

Parameterless Lambdas

- Simply write () as the parameter list:
- Example that creates a new thread from a *Runnable* object and starts the thread:

```
new Thread(
        ( ) -> {     // Runnable has method void run()
              System.out.println("runs in another thread");
            }
      ).start();
```

(in this one line function example the **{** and **}** are optional.
I wrote them because of the comment.)

Closures and Variable Access

- A *closure* is a function together with its environment:
  - actual parameters, instance and class variables from the enclosing class
  - ... plus local variables that are *effectively final* (called *free variables*)

```
for (int i=0; i<n; i++) {
  final int index = i;              // a final variable
  Date now = new Date();            // an effectively final variable

  new Thread(
        ( ) -> {
           System.out.println("thread started, index " + index + " at
                              time " + now);
        }
      );
}   // for i=...
```

Closures and Variable Access

- Lexical scope: the lambda expression is not a scope of it own, but it is part of the enclosing scope
- “this” and “super” refer to the enclosing class

```
for (int i=0; i<n; i++) {
  final int index = i;      // final, a “free variable”

  new Thread(
        ( ) -> {
           System.out.println(“thread started by “ + this.toString()
                    +  “ object, with loop index “ + index);
         }
      );
}    // for i=...
```

What are Lambdas, Actually ??

- A lambda is represented in the JVM by an instance of an **anonymous and hidden** class generated by the compiler, that implements the target functional interface.
- Lazy instantiation: object is not created unless used at runtime.

- Example:

```
Comparator <String> comp = (s1, s2) -> {
        if (s1.length() < s2.length()) return -1;
        if (s1.length() > s2.length()) return +1;
        return 0;
      } ;
```

Class *Object*'s Methods

- Methods (toString, hashCode, equals, clone...) are inherited by all classes → they are NOT abstract.
- A functional interface may include them:

```
@FunctionalInterface
interface Function1 <R,T> {
    String toString();                  // inherited from Object

    boolean equals(Object ob);   // inherited from Object

    R apply(T t);                       // function with one param
};
```

Dealing with Exceptions

- The lambda expression (code block) must comply with the function signature in its interface definition.
- If a function does not throw *checked* exception, than we must use try-catch:

```
new Thread(
        ( ) -> {     // Runnable's method void run()
              System.out.println("runs in another thread");
              Thread.sleep(1000);
            }
      ).start();
```

Trouble ! sleep() is declared to throw checked InterruptedException

Dealing with Exceptions

- The lambda expression (code block) must comply with the function signature in its interface definition.
- If a function does not throw *checked* exception, than we must use try-catch:

```
new Thread(
        ( ) -> {     // Runnable's method void run()
              System.out.println("runs in another thread");
              try { Thread.sleep(1000);
              } catch (InterruptedException ex) { }
            }
      ).start();
```

Solution: try – catch block

Method References

- If a proper method already exists, we can use it
- Example:
  - When mouse click on JButton, display the event. With a lambda expression:

```
jButton.addActionListener(
        event -> System.out.println(event)
    );
```

  - With a method reference:

```
jButton.addActionListener(
        System.out::println
    );
```

Method References

- Method reference wrapped in compatible lambda expression
- Three cases:
  - object::instanceMethod e.g. System.out::println
  - Class::staticMethod e.g. Math::sqrt
  - Class::instanceMethod e.g. Double::compareTo

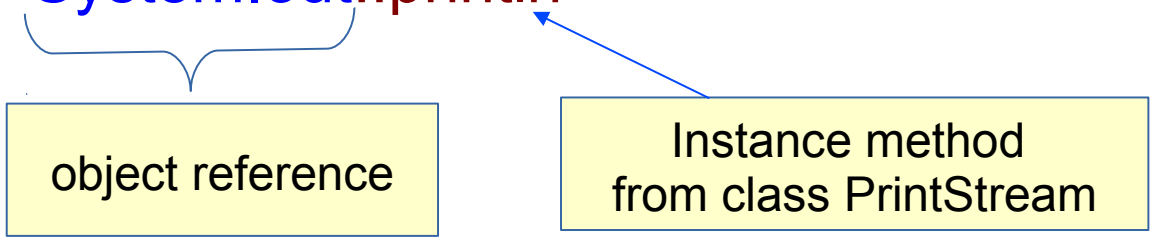
Method References

- object::instanceMethod

```
jButton.addActionListener(
        event -> System.out.println(event)
    );
```

same as:

```
jButton.addActionListener(
        System.out::println
    );
```



Method References

- Class::staticMethod

- *Suppose we want to create a **stream** of Doubles, compute square roots for them, then display them to System.out:*

```
List<Double> dl = Arrays.asList(0.5, 1.0, 3.0, -2.0);
```

- With lambda expression :

```
dl.stream().map( x -> Math.sqrt(x) )
           .forEach( y -> System.out.println(y) );
```

- Same with Class::staticMethod reference (Math::sqrt) :

```
dl.stream().map( Math::sqrt )
           .forEach( y -> System.out.println(y) );
```

Method References

- Class::instanceMethod
  - Suppose the functional interface abstract function is defined as:

    Type1 instanceMethod(Type2 y, Type3 z);
  - The first parameter in the lambda call becomes the target of the method (the implicit parameter), the second becomes y, the third becomes z, etc.
- E.g. instance method *compareTo()* from class *Double* is defined as:

  int compareTo(Double other)

Method References

- Class::instanceMethod (continued)
- Sorting List <Double>:

```
List<Double> dl = Arrays.asList(3.0, 4.0, 2.0, 1.0);
```

- Sorting with a lambda matching a *Comparator<Double>*:

```
Collections.sort(dl, (x,y) -> x.compareTo(y) );
```

- Equivalent, with a method reference, Double::compareTo

```
Collections.sort(dl, Double::compareTo );
```

Constructor Reference

- Similar to method references where the method name is **“new”**.
- Example: create array of *JButtons* from a list of labels
  - Function stream() creates a stream of objects
  - function *map*() calls the passed lambda on each stream element and returns the new stream)

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

labels.add(“Hello”); labels.add(“Goodbye”);

Stream<JButton> stream = labels.stream().map( JButton::new);

List<Button> buttons = stream.collect(Collectors.toList());
```

same as

```
Stream<JButton> stream =labels.stream().map( (String s) -> new JButton(s));
```

Further Reading

- The java.util.function package
  - https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html
- The Java Stream API
  - https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html
  - Tutorials:

    http://www.oracle.com/technetwork/articles/java/ma14-java-se-8-streams-2177646.html
  - http://winterbe.com/posts/2014/07/31/java8-stream-tutorial-examples/