

301AA - Advanced Programming

Lecturer: **Andrea Corradini**

andrea@di.unipi.it

<http://pages.di.unipi.it/corradini/>

AP-18: Lambdas and Streams in Java 8

Java 8: language extensions

Java 8 is the biggest change to Java since the inception of the language. Main new features:

- **Lambda expressions**
 - Improved type inference
 - Anonymous functions
 - Method references
 - Default methods in interfaces
- **Stream API**

A big challenge was to introduce lambdas without requiring recompilation of existing binaries

Benefits of Lambdas in Java 8

- Enabling functional programming
 - Being able to pass behaviors as well as data to functions
 - Introduction of lazy evaluation with stream processing
- Writing cleaner and more compact code
- Facilitating parallel programming
- Developing more generic, flexible and reusable APIs

Lambda expression syntax:

Print a list of integers with a lambda

```
List<Integer> intSeq = Arrays.asList(1,2,3);  
intSeq.forEach(x -> System.out.println(x));  
           x -> System.out.println(x)
```

is a **lambda expression** that defines an **anonymous function (method)** with one parameter named **x** of type Integer

```
// equivalent syntax  
intSeq.forEach((Integer x) -> System.out.println(x));  
intSeq.forEach(x -> {System.out.println(x);});  
intSeq.forEach(System.out::println); //method reference
```

- Type of parameter inferred by the compiler if missing

Multiline lambda, local variables, no new scope

```
List<Integer> intSeq = Arrays.asList(1,2,3);  
// multiline: curly brackets necessary  
intSeq.forEach(x -> {  
    x += 2;  
    System.out.println(x);  
});  
  
// local variable declaration  
intSeq.forEach(x -> {  
    int y = x + 2;  
    System.out.println(y);  
});  
  
// no new scope!!!  
int x = 0;  
intSeq.forEach(x -> {           //error: x already defined  
    System.out.println(x + 2);  
});
```

Local and Static Variable Capture

- Local variables used inside the body of a lambda must be **final** or **effectively final**

```
public class LVCEExample {    // local variable capture
    public static void main(String[] args) {
        List<Integer> intSeq = Arrays.asList(1,2,3);

        int var = 10;        // must be [effectively] final
        intSeq.forEach(x -> System.out.println(x + var));
        // var = 3; // uncommenting this line it does not compile
    }
}
```

This is a fundamental design choice, as it makes closures not necessary

```
public class SVCEExample {    // static variable capture
    private static int var = 10;
    public static void main(String[] args) {
        List<Integer> intSeq = Arrays.asList(1,2,3);

        intSeq.forEach(x -> System.out.println(x + var));
        var = 3;            // it compiles
    }
}
```

Implementation of Java 8 Lambdas

- The Java 8 compiler conceptually first converts a lambda expression into a function, compiling its code
- Then it generates code to call the compiled function where needed
- For example, `x -> System.out.println(x)` could be converted into a generated static function

```
public static void genName(Integer x) {  
    System.out.println(x);  
}
```
- But what type should be generated for this function? How should it be called? What class should it go in?

Functional Interfaces

- Design decision: Java 8 lambdas are instances of *functional interfaces*.
- A **functional interface** is a Java interface with **exactly one abstract method**. E.g.,

```
public interface Comparator<T> { //java.util
    int compare(T o1, T o2);
}

public interface Runnable { //java.lang
    void run();
}

public interface Consumer<T>{ //java.util.function
    void accept(T t)
}

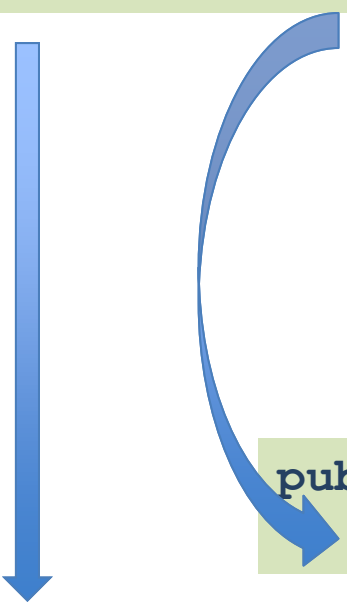
public interface Callable<V> { //java.util.concurrent
    V call() throws Exception;
}
```


Functional interfaces and lambdas

- Functional Interfaces can be used as *target type* of lambda expressions, i.e.
 - As type of variable to which the lambda is assigned
 - As type of formal parameter to which the lambda is passed
- The compiler uses type inference **based on target type**
- Arguments and result types of the lambda must match those of the unique abstract method of the functional interface
- The lambda is invoked by calling the only abstract method of the functional interface
- Lambdas can be interpreted as **instances** of **anonymous inner classes** implementing the functional interface

Expanding a lambda

```
List<Integer> intSeq = Arrays.asList(1,2,3);  
intSeq.forEach(x -> System.out.println(x));
```



```
// List<T> extends Iterable<T>  
interface Iterable<T>{ //java.lang  
    default void forEach(Consumer<? super T> action)  
        for (T t : this)  
            action.accept(t);
```

```
public interface Consumer<T>{ //java.util.function  
    void accept(T t); } //functional interface
```

```
List<Integer> intSeq = Arrays.asList(1,2,3);  
for (Integer t:intSeq)  
    System.out.println(t);
```

An example: From inner classes pre Java 8...

```
public class Calculator1 { // Pre Java 8
    interface IntegerMath { // (inner) functional interface
        int operation(int a, int b);
    }
    public int operateBinary(int a, int b, IntegerMath op) {
        return op.operation(a, b);
    } // parameter type is functional interface

    // inner class implementing the interface
    static class IntMath$Add implements IntegerMath{
        public int operation(int a, int b){
            return a + b;
        }
    }

    public static void main(String... args) {
        Calculator1 myApp = new Calculator1();
        System.out.println("40 + 2 = " +
            myApp.operateBinary(40, 2, new IntMath$Add()));
    } // anonymous inner class implementing the interface
    IntegerMath subtraction = new IntegerMath(){
        public int operation(int a, int b){
            return a - b;
        }
    };

    System.out.println("20 - 10 = " +
        myApp.operateBinary(20, 10, subtraction)); } }
```

... to lambda expressions

```
public class Calculator {  
  
    interface IntegerMath { // (inner) functional interface  
        int operation(int a, int b);  
    }  
  
    public int operateBinary(int a, int b, IntegerMath op) {  
        return op.operation(a, b);  
    } // parameter type is functional interface  


---

  
    public static void main(String... args) {  
        Calculator myApp = new Calculator();  
        // lambda assigned to functional interface variables  
        IntegerMath addition = (a, b) -> a + b;  
        System.out.println("40 + 2 = " +  
            myApp.operateBinary(40, 2, addition));  
        // lambda passed to functional interface formal parameter  
        System.out.println("20 - 10 = " +  
            myApp.operateBinary(20, 10, (a, b) -> a - b));  
    }  
}
```

From Lambdas to Bytecode

- Lambdas can, in principle, be compiled as instances of anonymous inner classes
- Neither JLS 8 nor JVM 8 prescribe a specific compilation strategy for lambdas
- The strategy is left to the designer of the compiler, which can exploit this freedom on behalf of efficiency
- Current compilers use the **invokedynamic** instruction of the JVM. It allows to defer to runtime the computation of a call-site.
- Let's explore with godbolt.org

Other examples of lambdas: Runnable

```
public class ThreadTest {// using functional interface Runnable
    public static void main(String[] args) {
        Runnable r1 = new Runnable() { // anonymous inner class
            @Override
            public void run() {
                System.out.println("Old Java Way");
            }
        };

        new Thread(r1).start();
        // using lambda expression
        Runnable r2 = () -> { System.out.println("New Java Way"); };

        new Thread(r2).start();
    }
}
```

```
// constructor of class Thread

public Thread(Runnable target)
```

Other examples of lambdas: Listener

```
JButton button = new JButton("Click Me!");

// pre Java 8
button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent evt) {
        System.out.println("Handled by anonymous class listener");
    }
});

// Java 8
button.addActionListener(
    e -> System.out.println("Handled by Lambda listener"));
```

New Functional Interfaces in package java.util.function

```
public interface Consumer<T> { //java.util.function
    void accept(T t);
}
public interface Supplier<T> { //java.util.function
    T get();
}
public interface Predicate<T> { //java.util.function
    boolean test(T t);
}
public interface Function <T,R> { //java.util.function
    R apply(T t);
}
```


Other examples of lambdas

```
List<Integer> intSeq = new ArrayList<>(Arrays.asList(1,2,3));

// sort list in descending order using Comparator<Integer>
intSeq.sort((x,z) -> z - x); // lambda with two arguments
intSeq.forEach(System.out::println);

// remove odd numbers using a Predicate<Integer>
intSeq.removeIf(x -> x%2 == 1);
intSeq.forEach(System.out::println); // prints only '2'
```

But, where do the `sort`, `removeIf` and `forEach` methods come from?

```
// default method of Interface List<E>
default void sort(Comparator<? super E> c)
// default method of Interface Collection<E>
default boolean removeIf(Predicate<? super E> filter)
// default method of Interface Iterable<T>
default void forEach(Consumer<? super T> action)
```

Default Methods

Problem: Adding new abstract methods to an interface breaks existing implementations of the interface

Java 8 allows interface to include

- Abstract (instance) methods, as usual
- **Static methods**
- **Default methods**, defined in terms of other possibly abstract methods

Java 8 uses lambda expressions and default methods in conjunction with the Java collections framework to achieve *backward compatibility* with existing published interfaces

Method References

- Method references can be used to pass an existing function in places where a lambda is expected
- The signature of the referenced method needs to match the signature of the functional interface method

Method Reference Type	Syntax	Example
static	ClassName::StaticMethodName	String::valueOf
constructor	ClassName::new	ArrayList::new
specific object instance	objectReference::MethodName	x::toString
arbitrary object of a given type	ClassName::InstanceMethodName	Object::toString

Summary

- Lambda expressions allow to define anonymous methods and to handle them as first-class objects
- Their types are functional interfaces, using type inference
- Certain typical coding patterns can be greatly simplified
- They are widely used in the Stream API
- Neither JLS 8 nor JVM 8 prescribe a specific compilation strategy for lambdas
- For a discussion about the possible compilation strategies and the choice of using **invokedynamic** to defer the choice to runtime, see **From Lambdas to Bytecode** by Brian Goetz