Java Lambda Expressions

Presented by Linda Seiter

Java Lambda Expressions

- Java is a class-based object-oriented language
  - All functionality defined in a class
  - No stand-alone functions

- Lambda Expression added in Java 8 (2014) to support functional-style programming

- Term lambda (**λ**) comes from Lambda Calculus (Alonzo Church, 1930s)
  - Model of computation based on functions
  - A lambda is an anonymous function (has no name)

	Named Function	Anonymous Function
`Define`	sumSquares(x,y) = $x^2 + y^2$	(x,y) = $x^2 + y^2$
`Invoke`	sumSquares(2,3) → $2^2 + 3^2$ → 13	???

Java Lambda Expression Syntax

Syntax	Example
`parameter -> expression`	`x -> x * x`
`parameter -> { codeblock }`	`n -> { int result= 1; for (int i = 1; i <= n; i++) result*= i; return result; }`
`(parameters) -> expression`	`(x, y) -> Math.sqrt(x*x + y*y)`

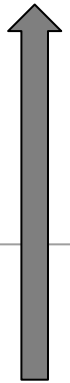
- Expression by default returns a value and cannot contain variable assignments, conditionals, loops, etc.
- Use code block with curly braces for multiple statements.
- The code block should have a `return` statement if it needs to return a value.

Lambda expression implements a functional interface

A functional interface has:

- single abstract method (SAM)
- 0+ abstract methods overriding a public java.lang.Object method (equals, hashCode, toString, etc)
- 0+ default or static methods

```
@FunctionalInterface
public interface StringOperator {
    String apply(String s);
}
```



```
StringOperator exclaim=  s -> s + "!";

System.out.println(exclaim.apply("wow"));
```

Implementing a functional interface

Separate concrete class (JDK 1.0 1996)	StringOperator exclaim=new Exclaim();
	
	System.out.println(exclaim.apply("wow"));
		class Exclaim implements StringOperator {
		  @Override
		  public String apply(String s) {return s + "!";}
		}

Anonymous class (JDK 1.1 1997)	StringOperator exclaim = new StringOperator () {
	  @Override
	  public String apply(String s) {return s + "!";}
	};
	
	System.out.println(exclaim.apply("wow"));	

Lambda Expression (Java 8 2014)	StringOperator exclaim = s -> s + "!"; System.out.println(exclaim.apply("wow"));	

CHALLENGE: Rewrite using lambda expressions

Anonymous Class	Lambda Expression

StringOperator exclaim = new StringOperator () {
public String apply(String s) { return s + "!" ; }
};
	StringOperator exclaim = s -> s + "!";

StringOperator question = new StringOperator () {
public String apply(String s) { return s + "?"; }
};
	StringOperator question =

StringOperator capitalize = new StringOperator () {
public String apply(String s) {
if (s == null || s.isEmpty()) return s ;
return s.substring(0, 1).toUpperCase()
+ s.substring(1);
}
};
	StringOperator capitalize =

Lambda expression - Concise and readable

Anonymous Class	Lambda Expression

StringOperator exclaim = new StringOperator () {
public String apply(String s) { return s + "!" }
};
	StringOperator exclaim = s -> s + "!";

StringOperator question = new StringOperator () {
public String apply(String s) { return s + "?"; }
};
	StringOperator question = s -> s + "?";

StringOperator capitalize = new StringOperator () {
public String apply(String s) {
if (s == null || s.isEmpty()) return s ;
return s.substring(0, 1).toUpperCase()
+ s.substring(1);
}
};
	StringOperator capitalize =

Lambda expression - Concise and readable

Anonymous Class	Lambda Expression

StringOperator exclaim = new StringOperator () {
public String apply(String s) { return s + "!" }
};
	StringOperator exclaim = s -> s + "!";

StringOperator question = new StringOperator () {
public String apply(String s) { return s + "?"; }
};
	StringOperator question = s -> s + "?";

StringOperator capitalize = new StringOperator () {
public String apply(String s) {
if (s == null || s.isEmpty()) return s ;
return s.substring(0, 1).toUpperCase()
+ s.substring(1);
}
};
	StringOperator capitalize = s -> {
	if(s == null || s.isEmpty()) return s;
	return s.substring(0,1).toUpperCase()
	+ s.substring(1);
	};

CHALLENGE: Explain the compiler error

Lambda Syntax	Example

parameter -> expression
 	
parameter -> { codeblock }
 	
(parameters) -> expression
	@FunctionalInterface
	public interface MyFunction {
	String apply(String s, String t);
	}
	
	MyFunction f = (a,b) -> { a + b }
	
	System.out.println(f.apply("HI","BYE"));

CHALLENGE: Explain the compiler error

Lambda Syntax	Example

parameter -> expression
parameter -> { codeblock }
(parameters) -> expression
	@FunctionalInterface
	public interface MyFunction{
	int m(int s, int t);
	String n(String s, String t);
	}
	
	MyFunction f1 = (s,t) ->s + t
	
	MyFunction f2 = (s,t) ->s - t

CHALLENGE: Explain the compiler error

Lambda Syntax	Example

parameter -> expression
 	
parameter -> { codeblock }
 	
(parameters) -> expression
	@FunctionalInterface
	public interface MyFunction {
	String apply(String s, String t);
	}
	
	MyFunction f = a,b -> a + b
	
	System.out.println(f.apply("hello","there"));

Sorting List Elements

interface List<E>
void sort(Comparator<E> c)

2 < 5

"bye" < "hi"

{name:Abby, age:25} < {name:Albert, age:18}

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

@FunctionalInterface
public interface Comparator<T>
int compare(T o1, T o2)

- Return negative integer if o1 precedes o2
- Return positive integer if o1 follows o2
- Return 0 if o1 equals o2 (in terms of order)

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

Challenge : Rewrite using lambda expressions

```
List<Person> people = Arrays.asList(
                                    new Person("Fred", 25),
                                    new Person("Albert", 18),
                                    new Person("Abby", 25));
people.sort( new Comparator<Person>() {
     public int compare(Person p1, Person p2) {
       return Integer.compare(p1.getAge(),p2.getAge());
     }
} );
people.sort( new Comparator<Person>() {
    public int compare(Person p1, Person p2) {
      return p1.getName().compareTo(p2.getName());
    }
} );
people.sort( new Comparator<Person>() {
    public int compare(Person p1, Person p2) {
       int ageCompare = Integer.compare(p1.getAge(),
                                        p2.getAge());
       return (ageCompare == 0) ?
                 p1.getName().compareTo(p2.getName()):
                 ageCompare;
       }
} );
```

```
List<Person> people = Arrays.asList(
                                    new Person("Fred", 25),
                                    new Person("Albert", 18),
                                    new Person("Abby", 25));
people.sort(



);
people.sort(



);
people.sort(



);
```

Challenge : Rewrite using lambda expressions

```
List<Person> people = Arrays.asList(
                                    new Person("Fred", 81),
                                    new Person("Albert", 25),
                                    new Person("Abby", 25));
people.sort(new Comparator<Person>(){
     public int compare(Person p1, Person p2) {
       return Integer.compare(p1.getAge(),p2.getAge());
     }
});
people.sort(new Comparator<Person>() {
    public int compare(Person p1, Person p2) {
      return p1.getName().compareTo(p2.getName());
    }
});
people.sort(new Comparator<Person>() {
    public int compare(Person p1, Person p2) {
       int ageCompare = Integer.compare(p1.getAge(),
                                        p2.getAge());
       return (ageCompare == 0) ?
                 p1.getName().compareTo(p2.getName()):
                 ageCompare;
       }
});
```

```
List<Person> people = Arrays.asList(
                                    new Person("Fred", 81),
                                    new Person("Albert", 25),
                                    new Person("Abby", 25));
people.sort(

   ( p1, p2) -> Integer.compare(p1.getAge(),p2.getAge())

);
people.sort(



);
people.sort(



);
```

Challenge : Rewrite using lambda expressions

```
List<Person> people = Arrays.asList(
                                    new Person("Fred", 81),
                                    new Person("Albert", 25),
                                    new Person("Abby", 25));
people.sort(new Comparator<Person>(){
     public int compare(Person p1, Person p2) {
       return Integer.compare(p1.getAge(),p2.getAge());
     }
});
people.sort(new Comparator<Person>() {
    public int compare(Person p1, Person p2) {
      return p1.getName().compareTo(p2.getName());
    }
});
people.sort(new Comparator<Person>() {
    public int compare(Person p1, Person p2) {
       int ageCompare = Integer.compare(p1.getAge(),
                                        p2.getAge());
       return (ageCompare == 0) ?
                 p1.getName().compareTo(p2.getName()):
                 ageCompare;
       }
});
```

```
List<Person> people = Arrays.asList(
                                    new Person("Fred", 81),
                                    new Person("Albert", 25),
                                    new Person("Abby", 25));
people.sort(

   ( p1, p2) -> Integer.compare(p1.getAge(),p2.getAge())

);
people.sort(

    (p1, p2) -> p1.getName().compareTo(p2.getName());

);
people.sort(









);
```

Lambda expressions: Concise and readable

```
List<Person> people = Arrays.asList(
                                    new Person("Fred", 81),
                                    new Person("Albert", 25),
                                    new Person("Abby", 25));
people.sort(new Comparator<Person>(){
     public int compare(Person p1, Person p2) {
       return Integer.compare(p1.getAge(),p2.getAge());
     }
});
people.sort(new Comparator<Person>() {
    public int compare(Person p1, Person p2) {
      return p1.getName().compareTo(p2.getName());
    }
});
people.sort(new Comparator<Person>() {
    public int compare(Person p1, Person p2) {
       int ageCompare = Integer.compare(p1.getAge(),
                                                p2.getAge());
       return (ageCompare == 0) ?
                 p1.getName().compareTo(p2.getName()):
                 ageCompare;
       }
});
```

```
List<Person> people = Arrays.asList(
                                    new Person("Fred", 81),
                                    new Person("Albert", 25),
                                    new Person("Abby", 25));
people.sort(

   ( p1, p2) -> Integer.compare(p1.getAge(),p2.getAge())

);
people.sort(

    (p1, p2) -> p1.getName().compareTo(p2.getName());

);
people.sort(
    ( p1, p2) -> {
       int ageCompare = Integer.compare(p1.getAge(),
                                                p2.getAge());
       return (ageCompare == 0) ?
                 p1.getName().compareTo(p2.getName()):
                 ageCompare;
       }
);
```

Still some code duplication in lambda expressions

```
List<Person> people = Arrays.asList(
                                    new Person("Fred", 81),
                                    new Person("Albert", 25),
                                    new Person("Abby", 25));


people.sort(  ( p1, p2) -> Integer.compare(p1.getAge(),p2.getAge())  );

people.sort(  ( p1, p2) -> p1.getName().compareTo(p2.getName()); );

people.sort(  ( p1, p2) ->  {
                    int ageCompare = Integer.compare(p1.getAge(), p2.getAge());
                   return (ageCompare == 0) ?
                              p1.getName().compareTo(p2.getName()):
                              ageCompare;
        }
);
```

Eliminating code redundancy

```
Comparator<Person> ageComparator = (p1, p2) -> Integer.compare(p1.getAge(),p2.getAge());
people.sort(ageComparator);


Comparator<Person> nameComparator =  (p1, p2) -> p1.getName().compareTo(p2.getName());
people.sort(nameComparator);


//Sort by age then name
Comparator<Person> ageThenNameComparator = ageComparator .thenComparing( nameComparator ));
```

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

Java 8 - Standard Functional Interfaces

Predicate<T>	T → Boolean	Predicate<Integer> isOdd= i -> i % 2 == 1; System.out.println(isOdd.**test**(7));
UnaryOperator<T>	T → T	UnaryOperator<String> exclaim =s -> s + "!"; System.out.println(exclaim.**apply**("hello"));
BinaryOperator<T>	(T,T) → T	BinaryOperator<Integer> subtract= (x1, x2) -> x1 - x2; System.out.println(subtract.**apply**(7, 3));
Function<T,R> BiFunction<T,U,R>	T → R (T,U) → R	Function<Integer, Double> half = i -> i / 2.0; System.out.println(half.**apply**(10)); BiFunction<Integer, Integer, Integer> sumSquares = (x,y) -> x*x + y*y; System.out.println(sumSquares.**apply**(2,3));
Supplier<T>	() → R	Supplier<Integer> randomDigit = () -> new Random().nextInt(10); System.out.print(randomDigit.**get**());
Consumer<T> BiConsumer<T,U>	T→ () (T,U) → ()	Consumer<String> printLength = (s) -> System.out.println(s.length()); printLength.**accept**("hello");

Passing lambdas to Stream methods

Stream<T> java.util.stream.Stream.**filter**(Predicate<T> predicate)

void java.util.stream.Stream.**forEach**(Consumer<T> action)

```
//Print odd numbers in a list
List<Integer> numberList = Arrays.asList(7,14,31,35,6);
numberList.stream()
        .filter( i -> i % 2 == 1 )
        .forEach( i -> System.out.println(i) );    //System.out::println
```

```
List<Employee> employees = …..;
double percent=0.1;
employees.forEach( e  -> e.updateSalary(percent) );
```

Function composition - andThen, compose

```
UnaryOperator<String> exclaim = s ->  s+"!" ;
UnaryOperator<String> question =s ->  s+"?" ;
UnaryOperator<String> capitalize = str ->  {
        if(str == null || str.isEmpty()) return str;
        return str.substring(0, 1).toUpperCase()+str.substring(1);
};
System.out.println(exclaim.andThen(question).apply("goodbye"));        //goodbye!?
System.out.println(exclaim.compose(question).apply("goodbye"));        //goodbye?!
UnaryOperator<String> emphaticQuestion =
                                s-> capitalize.andThen(question).andThen(exclaim).apply(s);
System.out.println(emphaticQuestion.apply("seriously"));                    //Seriously?!
```

Comparator factory methods and Java 8 Method References

Lambda Expression	Method Reference
//Accepts function that extracts an int sort key, returns Comparator that compares by that key. Comparator<Person> ageComparator = Comparator.comparingInt(**p -> p.getAge()**);	Comparator<Person> ageComparator = Comparator.comparingInt(**Person::getAge**);
//Accepts function that extracts a Comparable sort key, returns Comparator that compares by that key Comparator<Person> nameComparator = Comparator.comparing(**p-> p.getName()**);	Comparator<Person> nameComparator = Comparator.comparing(**Person::getName**);

//Sort by age then name
Comparator<Person> ageThenNameComparator=
ageComparator .thenComparing(nameComparator));
	Comparator<Person> ageThenNameComparator=
	ageComparator .thenComparing(nameComparator));

Anonymous Class ⇎ Lambda Expression

Anonymous Class	Lambda Expression

● declare + instantiate unnamed class
  ○ instance/static variables, methods
● implement/extend any interface/class
  ○ 0 + more abstract methods
 	
● Nested class scope
  ○ this - inner class instance
  ○ **Can** redeclare local variable of enclosing method
  ○ Can't access local variables in enclosing scope that are not final or effectively final
 	
● generate separate class file after compilation
	● declare unnamed method
	
	● extend a functional interface
	  ○ single abstract method
	
	● Block scope
	  ○ this - enclosing class instance
	  ○ **Can't** redeclare local variable of enclosing method
	  ○ Can't access local variables in enclosing scope that are not final or effectively final
	
	● convert to private method dynamically bound using **invokedynamic** bytecode

Summarizing Lambda Expressions

Pros	Cons
● Less boilerplate code ● Conciseness ● Readability ● Simplified variable scope ● JAR file size reductions ● Enhanced iterative syntax ● Parallel processing opportunities	● Possible slight performance hit ● Restricted to functional interface