Scala is Java8.next()

Twitter - @blueiur
La-scala - λdaewon

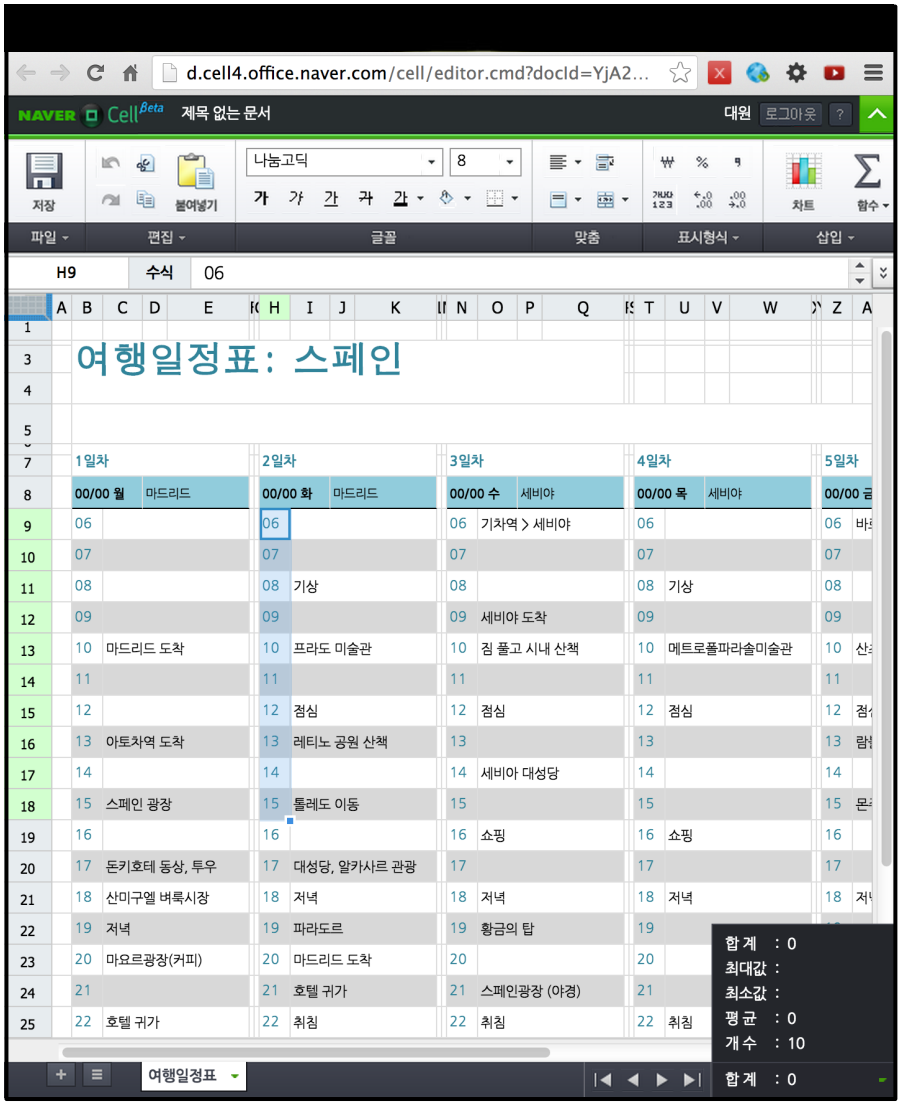
```java
// Before Java 8
Collections.sort(strings, new Comparator<String>() {
  @Override
  public int compare(String s1, String s2) {
    return s1.compareTo(s2);
  }
});
```

```java
// Java 8
Collections.sort(strings, (s1, s2) -> s1.compareTo(s2));
```

Same thing different syntax

Programmer

- http://office.naver.com



Programmer

- *http://daewon.github.io/*

Overview

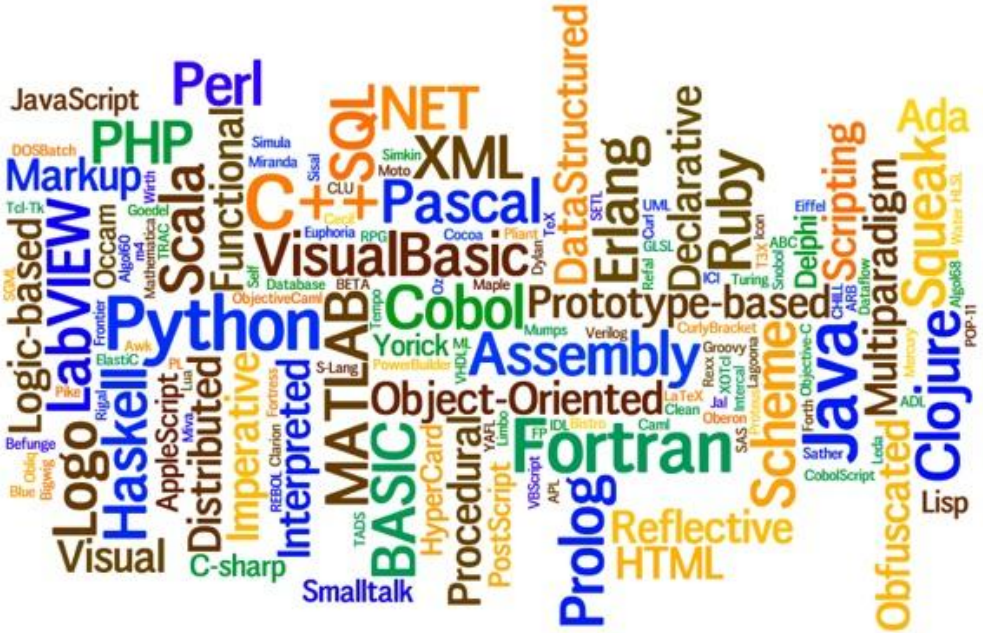
- *Scala*
- *Java8 features*
  - ***lambda***
    - *benefit of lambda expressions*
  - *comparison to Scala*

Scala

- ***Sca**lable **La**nguage*
- *Born in **2003***
- *Object-Oriented Meets **Functional***

Martin odersky

- *Designed the Scala programming language*
- *Generic Java, and built the current generation of javac, the Java compiler*



Features

- *SEAMLESS JAVA INTEROP*
- *TYPE INFERENCE*
- *CONCURRENCY & DISTRIBUTION*
- *TRAITS*
- *PATTERN MATCHING*
- *HIGHER-ORDER FUNCTIONS*

Scala *is used in* ***many worldclass*** *companies*

Rod johnson *joins* ***typesafe***

- *Creator of Spring*
- *Keynote at ScalaDays 2013*
  - *Scala in 2018*

New Features in Java 8

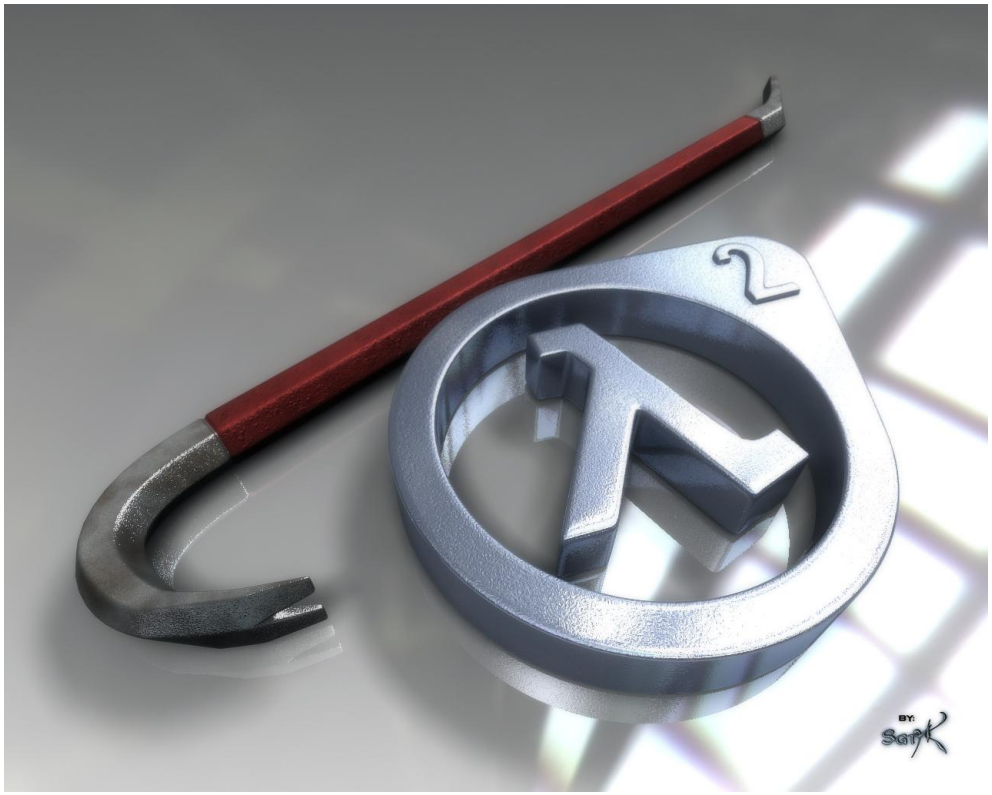
- ***Lambda Expressions***
- ***Method References***
- ***Virtual Extension Methods***

New Features in Java 8

- ***Lambda Expressions***
- ***Method References***
- ***Virtual Extension Methods***

Project Lambda - JSR 335

*2006 - **Gosling**: "We will never have closures in Java"*
*2008 - **Reinhold**: "We will never have closures in Java"*



Project Lambda

2007 - 3 different proposals for closures in Java
*2009 - Start of project Lambda (**JSR 335**)*

Project Lambda

JDK 8: General Availability - 2014/04/18

- *2006 - **Gosling**: "We will never have closures in Java"*
- *2007 - 3 different proposals for closures in Java*
- *2008 - **Reinhold**: "We will never have closures in Java"*
- *2009 - Start of project Lambda (**JSR 335**)*

λ *Lambda?*

- *lambda calculus*
- *anonymous function*
- *function literal*
- *closure*

λ *Lambda?*

- *lambda calculus*
- ***anonymous function***
- *function literal*
- ***closure***

λ *Anonymous Function*

Wikipedia

- *function defined, and possibly called,* ***without being bound to an identifier.***

λ 익명 함수

Wikipedia

- ***특정 식별자 없이 정의되거나 호출될 수 있는 함수***

λ 익

Wikipedia

- **특정 식**

Languages that support anonymous functions

ActionScript
Ada
C
C#
C++
Clojure
Curl
D
Dart
Dylan
Erlang
Elixir
F#
Frink
Go
Gosu
Groovy
Haskell
Java
JavaScript
Lisp
Logtalk
Lua
Mathematica
Maple
Matlab
Maxima
OCaml
Octave
Object Pascal
Objective-C
Pascal
Perl
PHP
Python
R
Racket
Ruby
Scala
Scheme
Smalltalk
Standard ML
TypeScript
Tcl
Vala
Visual Basic .NET
Visual Prolog

Languages that **no support** anonymous functions

ActionScript
Ada
C
C#
C++
Clojure
Curl
D
Dart
Dylan
Erlang
Elixir
F#
Frink
Go
Gosu
Groovy
Haskell
Java
JavaScript
Lisp
Logtalk
Lua
Mathematica
Maple
Matlab
Maxima
OCaml
Octave
Object Pascal
Objective-C
Pascal
Perl
PHP
Python
R
Racket
Ruby
Scala
Scheme
Smalltalk
Standard ML
TypeScript
Tcl
Vala
Visual Basic .NET
Visual Prolog

*Languages that **no support** anonymous functions*

ActionScript
Ada (1977)
C (1969)
C#
C++
Clojure
Curl
D
Dart
Dylan
Erlang
Elixir
F#
Frink
Go
Gosu
Groovy
Haskell
Java
JavaScript
Lisp
Logtalk
Lua
Mathematica
Maple
Matlab
Maxima
OCaml
Octave
Object Pascal
Objective-C
Pascal (1969)
Perl
PHP
Python
R
Racket
Ruby
Scala
Scheme
Smalltalk
Standard ML
TypeScript
Tcl
Vala
Visual Basic .NET
Visual Prolog

Languages that **no support** anonymous functions

ActionScript
Ada
C
C#
C++
Clojure
Curl
D
Dart
Dylan
Erlang
Elixir
F#
Frink
Go

Gosu
Groovy
Haskell
Java (1995)
JavaScript
Lisp

Object Pascal

Objective-C
Pascal
Perl
PHP
Python
R
Racket
Ruby
Scala
Scheme
Smalltalk
Standard ML
TypeScript
Tcl
Vala
Visual Basic .NET
Visual Prolog

λ *Java8 support Lambda*

- ***General Availability - 2014/04/18***

λ Lambda syntax

- *(parameters) -> { body }*

- *(int x, int y) -> { return x + y; }*

λ *Lambda syntax*

- *(param) -> {return 100}*
- *(param) -> {100}*
- *(param) -> 100*
- *param -> 100*

λ *Lambda syntax*

- *(param) -> {return 100}*
- *(param) -> {100}*
- *(param) -> 100*
- *param -> 100*

λ *Lambda syntax*

- *(param) -> {return 100}*
- *(param) -> {100}*
- *(param) -> 100*
- *param -> 100*

λ *Lambda syntax*

- *(param) -> {return 100}*
- *(param) -> {100}*
- *(param) -> 100*
- *param -> 100*

λ Lambda syntax

- *(param) -> {return 100}*
- *(param) -> {100}*
- *(param) -> 100*
- *param -> 100*

λ *Lambda syntax*

- x -> x + 1
- (x) -> x + 1
- (**int** x) -> x + 1
- (**int** x, **int** y) -> x + y
- (x, y) -> { System.out.println(x + y) }
- () -> { System.out.println("runnable!"); }

λ *Storing Lambda*

- int n = 10;
- String name = “daewon”;
- **? adder** = (**int** x, **int** y) -> x + y

λ *Storing Lambda*

- int n = 10;
- String name = “daewon”;
- **?** **adder** = (**int** x, **int** y) -> x + y

@Functional Interface

- ***Lambda Expression -> Anonymous Inner Class***

```
Comparable<String> c = new Comparable<String>() {
    compareTo(String o) {
        1;
    };
}
```

Replace with lambda
Convert to atomic

```
Calculator add = (i, j) -> add(i,j);
Calculator multiply = new Calculator(){
    t i, int j) {
        j);
    } ;

recordCalc(multiply, 3, 3);
recordCalc(add, 5, 5);
```

Replace with lambda
Replace with method reference
Convert to atomic

```
Comparable<String> c = o->{
```

Expand lambda to (String o) -> {...}
Replace lambda with anonymous class
Split into declaration and assignment
Convert to atomic

Replace with Lambda

```
@FunctionalInterface
interface Adder {
  int add(int a, int b);
}
```

```
????? func = (int a, int b) -> { return a + b };
Adder shortFunc = (a, b) -> a + b;
```

Functional Interface

```
@FunctionalInterface
interface Adder {
  int add(int a, int b);
}
```

```
Adder func = (int a, int b) -> { return a + b };
Adder shortFunc = (a, b) -> a + b;
```

Functional Interface

```
@FunctionalInterface
interface Adder {
  int add(int a, int b);
}
```

```
Adder func = (int a, int b) -> { return a + b };
Adder shortFunc = (a, b) -> a + b;
```

Type Inference

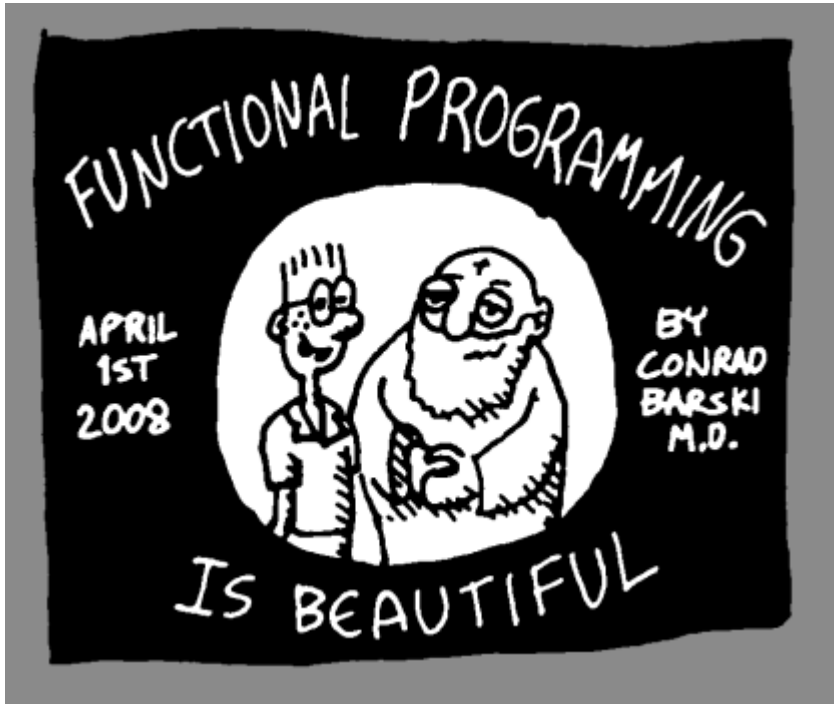
Predefined Functional Interfaces

BiConsumer<T,U>
BiFunction<T,U,R>
BinaryOperator<T>
BiPredicate<T,U>
BooleanSupplier
Consumer<T>
DoubleBinaryOperator
DoubleConsumer
DoubleFunction<R>
DoublePredicate
DoubleSupplier
DoubleToIntFunction
DoubleToLongFunction
DoubleUnaryOperator
Function<T,R>
IntBinaryOperator
IntConsumer
IntFunction<R>
IntPredicate
IntSupplier
IntToDoubleFunction
IntToLongFunction
IntUnaryOperator
LongBinaryOperator
LongConsumer
LongFunction<R>
LongPredicate
LongSupplier
LongToDoubleFunction
LongToIntFunction
LongUnaryOperator
ObjDoubleConsumer<T>
ObjIntConsumer<T>
ObjLongConsumer<T>
Predicate<T>
Supplier<T>


getConnection
connection
SQL = "SELECT
statement.
next()

Lambda in Java

- ***anonymous inner class***
- *lambda expressions*



```java
// Spring
jdbcTemplate.queryForObject("select * from student where id = ?",
  new Object[]{1212l},
  new RowMapper() {
    public Object mapRow(ResultSet rs, int n) throws SQLException {
        return new Student(rs.getString("name"), rs.getInt("age"));
    }
});
```

```java
// Google Guava
Iterables.filter(persons, new Predicate<Person>() {
    public boolean apply(Person p) {
        return p.getAge() > 18;
    }
});
```

Lambda in Java

```java
// Spring
jdbcTemplate.queryForObject("select * from student where id = ?",
  new Object[]{1212l},
  new RowMapper() {
    public Object mapRow(ResultSet rs, int n) throws SQLException {
        return new Student(rs.getString("name"), rs.getInt("age"));
    }
});
```

```java
// Google Guava
Iterables.filter(persons, new Predicate<Person>() {
    public boolean apply(Person p) {
        return p.getAge() > 18;
    }
});
```

Lambda in Java

```java
// Before Java 8
Collections.sort(strings, new Comparator<String>() {
  @Override
  public int compare(String s1, String s2) {
    return s1.compareTo(s2);
  }
});
```

```java
// Java 8
Collections.sort(strings, (s1, s2) -> s1.compareTo(s2));
```

Bulky syntax!

```
// Java 8
Collections.sort(strings, (s1, s2) -> s1.compareTo(s2));
```

Clear syntax!

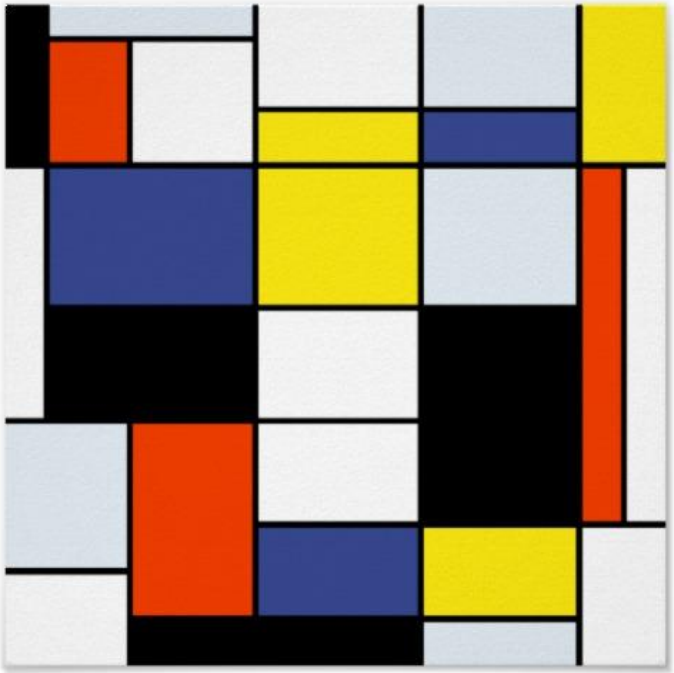
```java
// Before Java 8
Collections.sort(strings, new Comparator<String>() {
  @Override
  public int compare(String s1, String s2) {
    return s1.compareTo(s2);
  }
});
```

```java
// Java 8
Collections.sort(strings, (s1, s2) -> s1.compareTo(s2));
```

Same thing different syntax

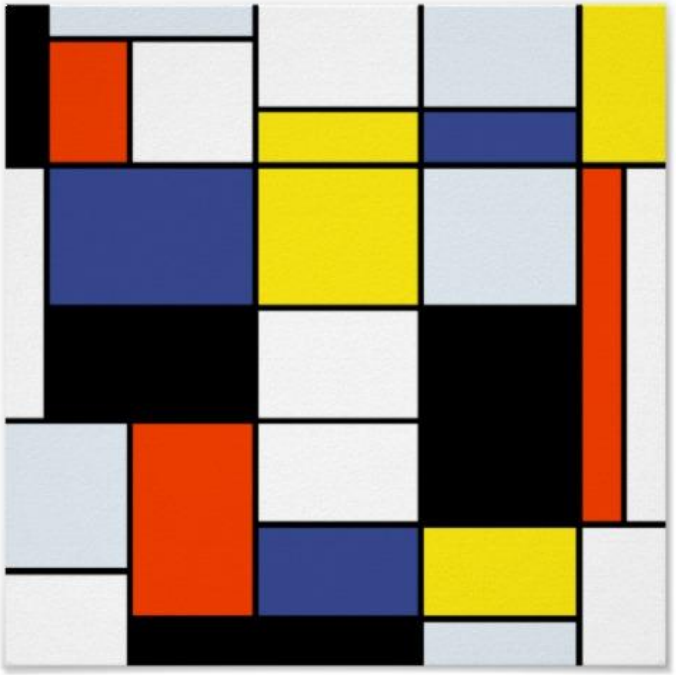
Functions are Data

- *function as parameter*
- *function as return value*
- *lazy eval*



함수는 데이터

- 함수를 다른 함수에 인자로 사용
- 함수를 함수의 반환 값으로 사용
- 지연 평가



함수는 데

- 함수를 다
- 함수를 함
- 지연 평가

Functions as parameter - Loan Pattern

- *scala_for_java_programmer*

```
public void withFile(String fileName) {
  BufferedReader bufferedReader = null;
    try {
      bufferedReader = new BufferedReader(new FileReader(filename));
      String line = null;
      while ((line = bufferedReader.readLine()) != null) {
        System.out.println(line);
      }
    } catch (IOException ex) {
      ex.printStackTrace();
    } finally {
    if (bufferedReader != null)
      bufferedReader.close();
    }
}
```

Print each line to STDOUT

```
public void withFile(String fileName) {
  BufferedReader bufferedReader = null;
    try {
      bufferedReader = new BufferedReader(new FileReader(filename));
      String line = null;
      while ((line = bufferedReader.readLine()) != null) {
        DB.store(line);
      }
    } catch (IOException ex) {
      ex.printStackTrace();
    } finally {
    if (bufferedReader != null)
      bufferedReader.close();
    }
}
```

Store each line to DB

```
public void withFile(String fileName) {
  BufferedReader bufferedReader = null;
    try {
      bufferedReader = new BufferedReader(new FileReader(filename));
      String line = null;
      while ((line = bufferedReader.readLine()) != null) {
        // insert code at here
      }
    } catch (IOException ex) {
      ex.printStackTrace();
    } finally {
    if (bufferedReader != null)
      bufferedReader.close();
    }
}
```

System.out.println(line);

DB.store(line);

Duplicate and Variation

```
public void withFile(String fileName, Consumer<String> work) {
  BufferedReader bufferedReader = null;
    try {
      bufferedReader = new BufferedReader(new FileReader(filename));
      String line = null;
      while ((line = bufferedReader.readLine()) != null) {
       work.accept(line);
      }
    } catch (IOException ex) {
      ex.printStackTrace();
    } finally {
    if (bufferedReader != null)
      bufferedReader.close();
    }
}
```

Refactoring with Lambda

```
public void withFile(String fileName, Consumer<String> work) {
  BufferedReader bufferedReader = null;
    try {
      bufferedReader = new BufferedReader(new FileReader(filename));
      String line = null;
      while ((line = bufferedReader.readLine()) != null) {
        work.accept(line);
      }
    } catch (IOException ex) {
      ex.printStackTrace();
    } finally {
    if (bufferedReader != null)
      bufferedReader.close();
    }
}
```

Refactoring with Lambda

```
// print each line to stdout
withFile(filename, line -> System.out.println(line));
```

```
// store each line to db
withFile(filename, line -> storeToDb(line));
```

```
// print and store each line
withFile(filename, line -> {
  System.out.println(line)
  storeToDb(line);
});
```

Refactoring with Lambda

```
// print each line to stdout
withFile(filename, line -> System.out.println(line));
```

```
// store each line to db
withFile(filename, line -> storeToDb(line));
```

Refactoring with Lambda

```scala
// Scala

def withFile(filename: String)(work: String => Unit) {
  Source.fromFile(filename).getLines.forEach work
}
```

Scala

```scala
// Scala

def withFile(filename: String)(work: String => Unit) {
  Source.fromFile(filename).getLines.forEach work
}
```

Scala

```java
// Java8 - print each line to stdout
withFile(filename, line -> System.out.println(line));

// Java8 - store each line to db
withFile(filename, line -> storeToDb(line));
```

```scala
// Scala - print each line to stdout
withFile(filename) {
  line => println(line)
}

// Scala - store each line to db
withFile(filename) {
  line => storeToDb(line)
}
```

Multiple parameter lists(curried function)

Lambda with Collections

- *Imperative - external iteration*
- *Functional - internal iteration*

```
// External Iterator
for (Iterator iter = var.iterator(); iter.hasNext(); ) {
  Object obj = iter.next();
  // Operate on obj
}
```

```
// Internal Iterator
var.each( new Functor() {
  public void operate(Object arg) {
    arg *= 2;
  }
});
```

Internal iterator? External iterator?

```
// External Iterator
for (Iterator iter = var.iterator(); iter.hasNext(); ) {
  Object obj = iter.next();
  // Operate on obj
}
```

```
// Internal Iterator
var.each( new Functor() {
  public void operate(Object arg) {
    arg *= 2;
  }
});
```

When you pass a function object to a method to run over a collection, that is an internal iterator

```
// External Iterator
for (Iterator iter = var.iterator(); iter.hasNext(); ) {
  Object obj = iter.next();
  // Operate on obj
}
```

```
// Internal Iterator
var.each( new Functor() {
  public void operate(Object arg) {
    arg *= 2;
  }
});
```

컬렉션을 순회하는 동안 특정 행동을 하는 함수 객체를 전달하는 방식을

내부 반복자라고 한다

```
// External I
for (Iterato
  Object obj
  // Operate
}
```

```
// Internal I
var.each( n
  public voi
    arg *= 2
  }
});
```

컬렉션을 순회하는 동안 특정 행동을 하는 함수 객체를 전달하는 방식을
내부 반복자라고 한다

```
// Java8:: print each line to stdout
withFile(filename, line -> System.out.println(line));

// Java8 - store each line to db
withFile(filename, line -> storeToDb(line));
```

```
// Scala - print each line to stdout
withFile(filename) {
  line => println(line)
}

// Scala:: store each line to db
withFile(filename) {
  line => storeToDb(line)
}
```

Internal iterator

```
// Java8:: print each line to stdout
withFile(filename, line -> System.out.println(line));

// Java8 - store each line to db
withFile(filename, line -> storeToDb(line));
```

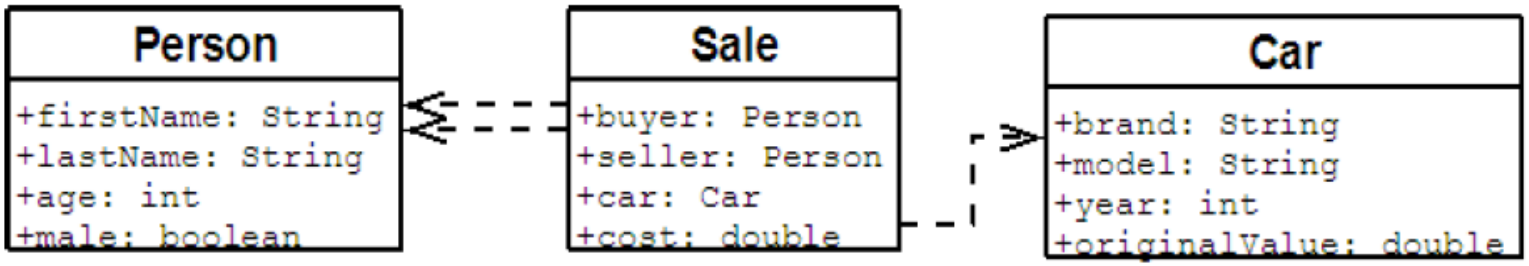
```
// Scala - print each line to stdout
withFile(filename) {
  line => println(line)
}

// Scala:: store each line to db
withFile(filename) {
  line => storeToDb(line)
}
```

Iterator

No more loop with Scala

- *Project Lambda J*
- *Google guava-libraries/*



case class Person(firstName: String, lastName: String, age:Int, male: Boolean)
case class Sale(buyer: Person, seller: Person, car: Car, cost: Double)
case class Car(brand: String, model: String, year: Int, originalValue: Double)

LambdaJ Demo Data Model

```scala
object db {

    // car
    val sonata = Car("Hyundai", "Sonata", 1982, 30000000)
    val santafe = Car("Hyundai", "Santafe", 1990, 50000000)
    val k7 = Car("KIA", "K7", 2000, 50000000)
    val k9 = Car("KIA", "K9", 2008, 70000000)
    val orlando = Car("GM", "Orlando", 2011, 30000000)
    val chevrolet = Car("GM", "Chevrolet", 2010, 50000000)
    val alpheon = Car("GM", "Alpheon", 2012, 70000000)

    def cars = List(sonata, santafe, k7, k9, orlando, chevrolet, alpheon);

    // person
    val daewon = Person("daewon", "jeong", 31, true)
    val youngtek = Person("youngtek", "hong", 32, true)
    val jiwong = Person("jiwong", "kang", 34, true)
    val taehee = Person("taehee", "kim", 32, false)

    def persons = List(youngtek, daewon, jiwong, taehee)

    // sales: map person/car
    def sales = List(Sale(daewon, taehee, sonata, 30000000),
        Sale(daewon, youngtek, santafe, 50000000),
        Sale(daewon, jiwong, santafe, 50000000),
        Sale(jiwong, taehee, santafe, 50000000),
        Sale(taehee, daewon, chevrolet, 50000000),
        Sale(youngtek, daewon, chevrolet, 50000000),
        Sale(youngtek, taehee, orlando, 30000000),
        Sale(taehee, jiwong, chevrolet, 30000000))
}
```

```java
// Iterative version
List<Sale> salesOfAHyundai = new ArrayList<Sale>();
for (Sale sale : sales) {
  if (sale.getCar().getBrand().equals("Hyundai")) {
    salesOfAHyundai.add(sale);
  }
}
```

```java
// Functional version
val salesOfHyundai = db.sales.filter(_.car.brand == "Hyundai")
```

현대차를 판매하는 모든 Sales객체를 선택

```
// Iterative version
List<Sale> salesOfAHyundai = new ArrayList<Sale>();
for (Sale sale : sales) {
  if (sale.getCar().getBrand().equals("Hyundai")) {
    salesOfAHyundai.add(sale);
  }
}
```

```
// Functional version
val salesOfHyundai = db.sales.filter(_.car.brand == "Hyundai")
```

sales 목록에서 car brand가 Hyundai인 것만 필터링

```java
// Iterative version
List<Sale> salesOfAHyundai = new ArrayList<Sale>();
for (Sale sale : sales) {
  if (sale.getCar().getBrand().equals("Hyundai")) {
    salesOfAHyundai.add(sale);
  }
}
```

```java
// Functional version
val salesOfHyundai = db.sales.filter(_.car.brand == "Hyundai")
```

Select all sales of Hyundai

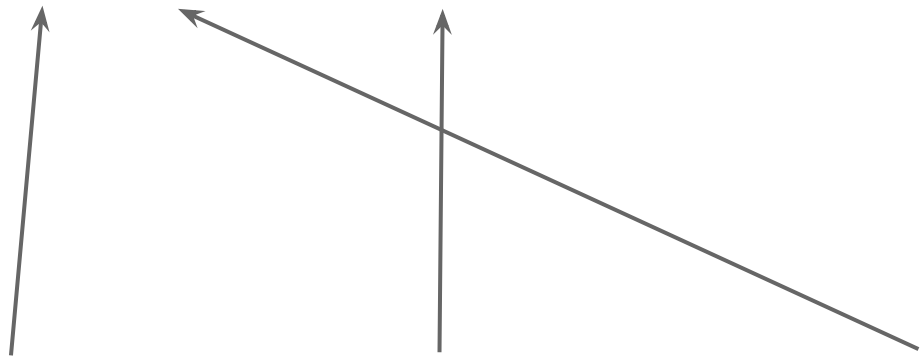
```
// Iterative version
List<Double> costs = new ArrayList<Double>();
for (Car car : cars) {
  costs.add(car.getOriginalValue());
}
```

```
// Functional version
val costs = db.cars.map(_.originalValue)
```

자동차에 Original 가격 목록을 작성

```
// Iterative version
List<Double> costs = new ArrayList<Double>();
for (Car car : cars) {
  costs.add(car.getOriginalValue());
}
```

```
// Functional version
val costs = db.cars.map(_.originalValue)
```



cars 목록으로부터 originalValue를 추출해서 새 목록을 작성

```
// Iterative version
List<Double> costs = new ArrayList<Double>();
for (Car car : cars) {
  costs.add(car.getOriginalValue());
}
```

```
// Functional version
val costs = db.cars.map(_.originalValue)
```

Extract car's original cost

```
// Iterative version
double maxCost = 0.0;
for (Sale sale : sales) {
  double cost = sale.getCost();
  if (cost > maxCost) {
    maxCost = cost;
  }
}
```

```
// Functional version
val maxCost = db.sales.maxBy(_.cost).cost
```

가장 비싸게 판매한 값은?

```java
// Iterative version
double maxCost = 0.0;
for (Sale sale : sales) {
  double cost = sale.getCost();
  if (cost > maxCost) {
    maxCost = cost;
  }
}
```

```java
// Functional version
val maxCost = db.sales.maxBy(_.cost).cost
```

Find most costly sale

```java
// Iterative version
// 01. find youngest person in persons
Person youngest = null;
for (Person person : persons){
  if (youngest == null || person.getAge() < youngest.getAge()) {
    youngest = person;
  }
}
// 02. find buyer have age equal to younggest person
List<Sale> buys = new ArrayList<Sale>();
for (Sale sale : sales){
  if (sale.getBuyer().equals(youngest)) {
    buys.add(sale);
  }
}
```

```java
// Functional version
val youngestPerson = db.persons.minBy( _.age )
val buys = db.sales.filter( _.buyer == youngestPerson )
```

가장 어린 사람과 나이가 같은 판매자들 찾기

```
// Iterative version
// 01. find youngest person in persons
Person youngest = null;
for (Person person : persons){
  if (youngest == null || person.getAge() < youngest.getAge()) {
    youngest = person;
  }
}
// 02. find buyer have age equal to younggest person
List<Sale> buys = new ArrayList<Sale>();
for (Sale sale : sales){
  if (sale.getBuyer().equals(youngest)) {
    buys.add(sale);
  }
}
```

```
// Functional version
val youngestPerson = db.persons.minBy( _.age )
val buys = db.sales.filter( _.buyer == youngestPerson )
```

Find buys of youngest person

```
// 01. 가장 어린 사람을 찾고
val youngestPerson = db.persons.minBy( _.age )
```

```
// 02. 가장 어린 사람과 나이가 같은 판매자들을 찾는다.
val buys = db.sales.filter( _.buyer == youngestPerson )
```

Find buys of youngest person

```java
// Iterative version
Map<String, List<Car>> carsByBrand = new HashMap<>();
for (Car car : db.getCars()) {
  List<Car> carList = carsByBrand.get(car);
  if (carList == null){
      carList = new ArrayList<Car>();
      carsByBrand.put(car.getBrand(), carList);
  }
  carList.add(car);
}
```

```java
// Functional version
val carsByBrand = db.cars.groupBy(_.brand)
```

brand 기준으로 car를 그루핑

```
// Iterative version
Map<String, List<Car>> carsByBrand = new HashMap<>();
for (Car car : db.getCars()) {
  List<Car> carList = carsByBrand.get(car);
  if (carList == null){
      carList = new ArrayList<Car>();
      carsByBrand.put(car.getBrand(), carList);
  }
  carList.add(car);
}
```

```
// Functional version
val carsByBrand = db.cars.groupBy(_.brand)
```

Select * from cars group by brand

```
// Iterative version
Map<String, List<Car>> carsByBrand = new HashMap<>();
for (Car car : db.getCars()) {
  List<Car> carList = carsByBrand.get(car);
  if (carList == null){
      carList = new ArrayList<Car>();
      carsByBrand.put(car.getBrand(), carList);
  }
  carList.add(car);
}
```

```
// Functional version
val carsByBrand = db.cars.groupBy(_.brand)
```

Index cars by brand

```java
// Iterative version
Map<Car, Integer> carsBought = new HashMap<Car, Integer>();
for (Sale sale : sales) {
  Car car = sale.getCar();
  int boughtTimes = carsBought.get(car);
  carsBought.put(car, boughtTimes == null ? 1 : boughtTimes+1);
}

Car mostBoughtCarIterative = null;
int boughtTimesIterative = 0;

for (Entry<Car, Integer> entry : carsBought.entrySet()) {
  if (entry.getValue() > boughtTimesIterative) {
    mostBoughtCarIterative = entry.getKey();
    boughtTimesIterative = entry.getValue();
  }
}
```

가장 많이 구매한 차

```
// Functional version
val carsBought = db.sales.groupBy( _.car ).mapValues( _.length )
val mostBoughtCar = carsBought.maxBy {
    case (_, boughtCount) => boughtCount
}._1

val boughtTimes = carsBought(mostBoughtCar)
```

Find most bought car

```
// Functional version
val carsBought = db.sales.groupBy( _.car ).mapValues( _.length )
val mostBoughtCar = carsBought.maxBy {
    case (_, boughtCount) => boughtCount
}._1

val boughtTimes = carsBought(mostBoughtCar)
```

Find most bought car

```
// Iterative version
Map<Person,Map<Person,Sale>> map = new HashMap<Person, Map<Person,Sale>>();
for (Sale sale : sales) {
  Person buyer = sale.getBuyer();
  Map<Person, Sale> buyerMap = map.get(buyer);
  if (buyerMap == null) {
    buyerMap = new HashMap<Person, Sale>();
    map.put(buyer, buyerMap);
  }
  buyerMap.put(sale.getSeller(), sale);
}

Person youngest = null;
Person oldest = null;
for (Person person : persons) {
  if (youngest == null || person.getAge() < youngest.getAge()){
    youngest = person;
  }
  if (oldest == null || person.getAge() > oldest.getAge()) {
    oldest = person;
  }
}

Sale saleFromYoungestToOldest = map.get(youngest).get(oldest);
```

Group sales by buyer and sellers

```
// Functional version
val map = db.sales.groupBy(_.buyer).mapValues{
  ls => ls.groupBy(_.seller).mapValues(_.head)
}

val youngest = db.persons.minBy(_.age)
val oldest = db.persons.maxBy(_.age)
val saleFromYoungestToOldest = map(youngest)(oldest)
```

Group sales by buyer and sellers

Java8 Streams *vs* ***Scala For Comprehension***

- *streams and sql*
  - *how-java-emulates-sql*
- *for comprehension*


Stream.of(1, 2,
map(x -> x * x)
filter(x -> x % 2 == 0
(sum, x)
reduce(0,

```sql
// sql
select sum(salary)
from   people
where  gender = 'FEMALE'
```

```java
// Java8 stream with lambda
people.stream()
  .filter(p -> p.getGender() == Person.Gender.FEMALE)
  .mapToInt(p -> p.getSalary())
  .sum();
```

모든 여성의 급여의 합

```
// sql
select sum(salary)
from   people
where  gender = 'FEMALE'
```

```
// scala for comprehension
(for (p <- people;
  (if p.gender == Person.Gender.FEMALE))
  yield p.salary).sum()
```

Find the sum of all the salaries of all the females.

```sql
// sql
select sum(salary)
from   people
where  gender = 'FEMALE'
```

```scala
// scala for comprehension
(for (p <- people;
  (if p.gender == Person.Gender.FEMALE))
  yield p.salary).sum()
```

Find the sum of all the salaries of all the females.

```sql
// sql
select distinct(firstName)
from   people
where  gender = 'MALE'
```

```java
// Java8 stream with lambda
people.stream()
  .filter(p -> p.getGender() == Person.Gender.MALE)
  .map(p -> p.getFirstName())
  .distinct()
```

중복을 제거한 모든 남성의 이름

```
// sql
select distinct(firstName)
from   people
where  gender = 'MALE'
```

```
// scala for comprehension
(for (p <- people;
  if (p.getGender == Person.Gender.MALE))
  yield p.firstName).distinct
```

Prepare a list of all (distinct) first names of the males

```
// sql
select distinct(firstName)
from   people
where  gender = 'MALE'
```

```
// scala for comprehension
(for (p <- people;
  if (p.getGender == Person.Gender.MALE))
  yield p.firstName).distinct
```

Prepare a list of all (distinct) first names of the males

나이가 가장 많은 남자들

```sql
// sql
select   *
from     people
where  age =
         (select max(age) from people where  gender ='MALE' )
and      gender ='MALE'
```

```java
// Java8
int maxAge = people.stream().max(
     (p1, p2) -> p1.getAge() - p2.getAge()).get());

     people.stream()
  .filter(p -> p.getGender() == Person.Gender.MALE
     && p.getAge() == max_age)
  .collect(Collections.toList()));
```

```scala
// scala for comprehension
for {
  maxAge <- Seq(person.maxBy(_.age));
  p <- people;
  if (p.age == maxAge && p.gender == Person.Gender.MALE)
} yield p
```


Functional Programming
A bad day in () is better than a good day in {}.

Is Java8 a functional language?

- ***What is Functional Programming?***
  - *Immutable state*
  - *Functions as first class citizens*
  - *Persistent data structure*

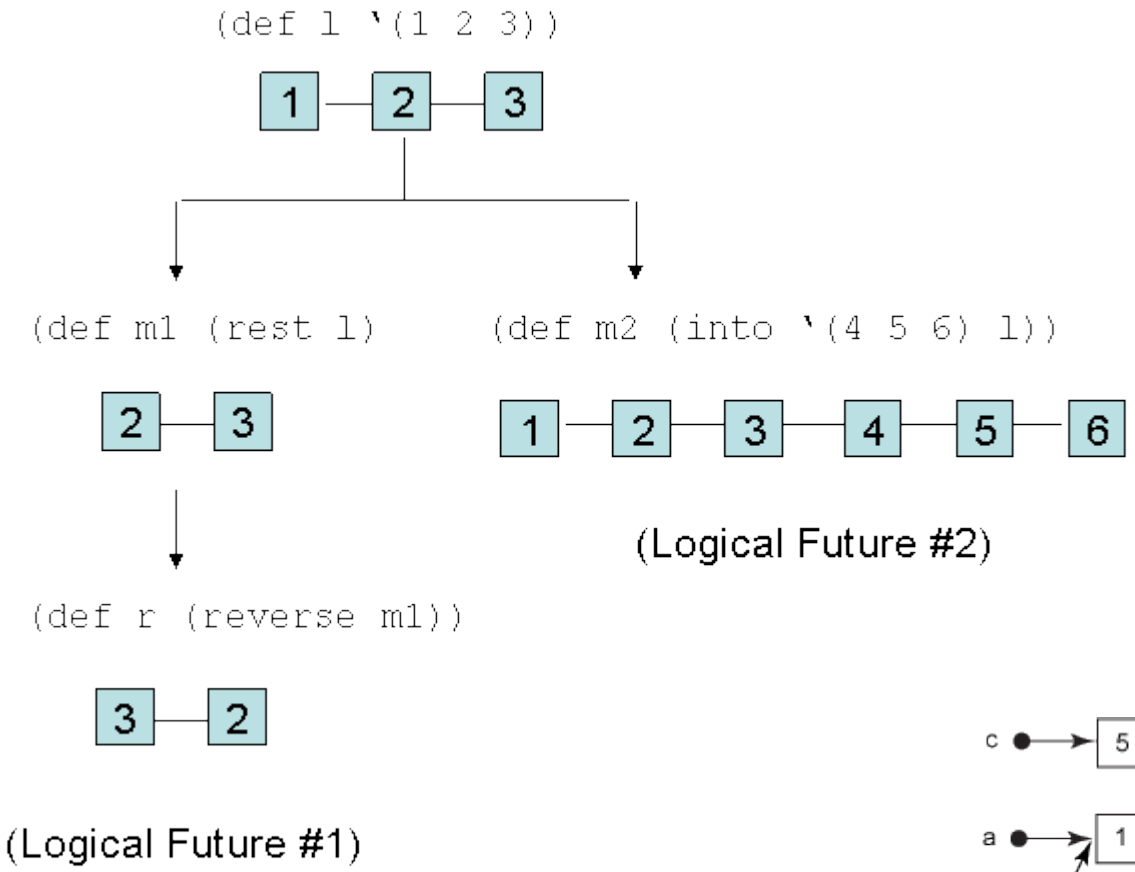
Immutable

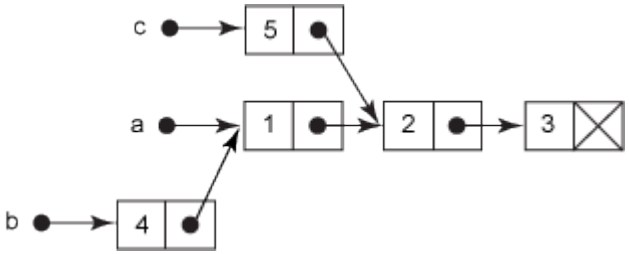
- *No setter*
- *Variable must be final*

```
final int i = 0;

while(i < 5) {
  System.out.println("compile error!");
  i++;
}
```

Persistent Data Structure





Java8

- ~~*Immutable state*~~
- ~~*Functions as first class citizens*~~
- ~~*Persistent data structure*~~

Scala

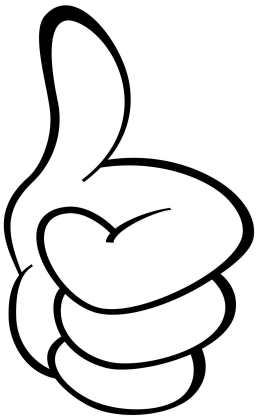
- *Immutable state*
- *Functions as first class citizens*
- *Persistent data structure*

*Java8 is **not** a functional language.* ***So what?***

- ***Real-world programming isn't always perfectly functional***
- ***Java is steel the most popular language in the world***
- ***Functional programming is not a silver bullet***

자바는 함수형 언어가 아닙니다. 그래서요?

- Real-world 문제가 항상 함수형에 적합하지는 않습니다
- 자바는 여전히 세계에서 가장 많이 쓰이고 있는 언어입니다
- 함수형 언어는 은탄환이 아닙니다



Java8 is Good!

---

*Scala is **Best!***

One more thing

- ***Lambda? Closure?***

λ *Lambda? Closure?*

```
// lambda is just an anonymous function
Collection.filter(lists, e -> e.length() >= 5)


// closure is function that closes over the environment it was defined
int min = 5;
Collection.filter(lists, e -> e.length() >= min);
```

λ *Lambda? Closure?*

```
// lambda is just an anonymous function
Collection.filter(lists, e -> e.length() >= 5)


// closure is function that closes over the environment it was defined
int min = 5;
Collection.filter(lists, e -> e.length() >= min);
```

Lazy eval - *Better logging with Lambda*

```java
// log message only debug mode
public void debug(String message) {
  if (log.isDebugEnabled()) {
    log.log(message);
  }
}
```

```java
debug(some.expensive("operation"));
```

log expensive operation

```
// log message only debug mode
public void debug(String message) {
  if (log.isDebugEnabled()) {
    log.log(message);
  }
}
```

```
debug(some.expensive("operation"));
```

log expensive operation

```java
// log message only debug mode
public void debug(Consumer<String> consumer) {
  if (log.isDebugEnabled()) {
    log.log(consumer.accept());
  }
}
```

```java
debug(() -> some.expensive("operation"));
```

*log expensive operation with **Lambda***

```java
// log message only debug mode
public void debug(Consumer<String> consumer) {
  if (log.isDebugEnabled()) {
    log.log(consumer.accept());
  }
}
```

```java
debug(() -> some.expensive("operation"));
```

*log expensive operation with **Lambda***

```
// without Lambda
debug(some.expensive("operation"));
```

```
// with Lambda
debug(() -> some.expensive("operation"));
```

bulk syntax!

```scala
// Scala
def debug(message: => String) {
    if (log.isDebugEnabled) log.log(message)
}
```

```scala
debug(some.expensive("operation"))
```

Lazy eval in Scala

```scala
// Scala
def debug(message: => String) {
    if (log.isDebugEnabled) log.log(message)
}
```

```scala
debug(some.expensive("operation"))
```

Lazy eval in Scala

```
// Java:: Eager eval
debug(some.expensive("operation"));
```

```
// Java8:: Lazy eval
debug(() -> some.expensive("operation"));
```

```
// scala:: Lazy eval
debug(some.expensive("operation"))
```

Question => Answer

// quiz

Lambda를 저장시 사용되는 인터페이스에 사용되는 Annotation은?

경품!