

LGE Internal Use Only
SW College
학습자용

Lambda

sanghyuck.na@lge.com

도전하세요!

- 다음 요구사항을 만족하는 프로그램을 작성하세요
 - 새로운 Thread myThread를 생성하세요
 - myThread를 실행 시키면 화면에 "Running"을 1회 출력합니다.
 - Thread는 실행을 마치면 종료합니다.

정답

- Runnable instance는 Anonymous instance로써 가장 핵심이 되는 Business logic을 구성하기 위해해서는 추가 JAVA 문법규칙을 지켜야는 부분이 코드의 절반을 차지합니다.
- Lambda는 코드를 단순하게 함으로써 가독성을 높이는 기본 기법입니다.

```
Thread myThread = new Thread(new Runnable() {  
    @Override  
    public void run() {  
        System.out.println("running");  
    }  
});  
myThread.start();  
myThread.join();
```

미리 보는 학습결과

```
Thread myThread = new Thread(new Runnable() {  
    @Override  
    public void run() {  
        System.out.println("running");  
    }  
})
```



```
new Thread(() -> System.out.println("running"));
```

Lambda

- Lambda 표현식은 arguments, code, data처럼 함수에 전달하는 방식
 - 익명 클래스의 핵심 이슈는 만약 익명 클래스에 구현해야 할 메소드가 단 하나인 경우 단순하지만, 그 익명 클래스의 문법은 불편하고 복잡하게 만든다
 - Lambda 표현식은 단일 메소드 클래스의 인스턴스를 더 압축해서 표현하여 단순화
- Lambda Expression 장점
 - 표현적 요소: 가독성
 - 성능적 요소: 임시 변수 제거, 스케줄링 선택,
 - 재사용 요소: 구체적인 Interface class를 Code로 작성하지 않게 되어 의존성 제거

```
LambdaExpression:  
LambdaParameters -> LambdaBody  
  
LambdaBody:  
Expression  
Block
```

```
LambdaParameters:  
( [LambdaParameterList] )  
Identifier  
  
LambdaParameterList:  
LambdaParameter {, LambdaParameter}  
Identifier {, Identifier}  
  
LambdaParameter:  
{VariableModifier} LambdaParameterType  
VariableDeclaratorId  
VariableArityParameter  
  
LambdaParameterType:  
UnannType  
var
```

Step1 – Functional Interface

- Functional interface만 Lambda로 바꿀수 있습니다
 - Lambda 표현식으로 바꿀 익명클래스가 @FunctionalInterface인지 확인
- @FunctionalInterface
 - Lambda로 변형 할 수 있는 Interface임을 알려주는 정보(informative annotation type)
 - Interface이면서 단 1개 abstract method (단, default methods는 제외)

Interface Runnable

All Known SubInterfaces:

RunnableFuture<V>, RunnableScheduledFuture<V>

All Known Implementing Classes:

AsyncBoxView.ChildState, ForkJoinWorkerThread, FutureTask, RenderableImageProducer, SwingWorker, Task, Thread, TimerTask

1. Functional Interface!

```
@FunctionalInterface  
public interface Runnable
```

Method Summary

2. Single abstract method

All Methods	Instance Methods	Abstract Methods
Modifier and Type	Method and Description	
void	run() When an object implementing interface Runnable is used to create a thread, starting the thread causes the object's run method to be called in that separately executing thread.	

Step2 – Lambda Parameters

- 삭제
 - 익명클래스생성표현식의 new 부터 method name까지 모두삭제
 - -> 추가
- Lambda parameter list 기술
 - Parameter가 있는 경우 내에 Comma로 구분하고, Parameter의 이름을 기술
 - Type(Class, var, 식별자)은 모두 기술하거나 모두 생략합니다.
 - Parameter가 1개인 경우 괄호 "()" 생략 가능합니다.
 - Parameter가 없는 경우 빈 괄호로 하고, 그 이후 기호 "->" 추가: ()->lambda_body
- LambdaParameters -> LambdaBody
 - VariableModifier: Annotation, final
 - LambdaParameterType: UnannType(type 생략), var¹¹

```
new Thread(new Runnable {  
    public void run() {  
  
    }  
})
```

```
new Thread(new Runnable {  
    public void run() -> {  
  
    }  
})
```

Step3 – Lambda Body

- LambdaBody에 명령어가 1줄인 경우 규칙
 - LambdaBody의 대괄호"{"와 명령문의 ";"(Semicolon) 삭제
 - 반환 값이 있는 경우 "return" keyword 삭제

```
new Thread(() -> {  
    System.out.println("Worker.run() called");  
}).start();
```

```
new Thread(() -> {  
    System.out.println("Worker.run() called");  
}).start();
```


Lambda – 정리

- @FunctionalInterface 인지 확인
- 기존 익명클래스 생성표현식에서 new 부터 method name까지 모두 삭제
- Lambda 기호 추가
- LambdaParameter
 - 없는 경우 빈 괄호로 하고, 그 이후 기호 "->" 추가: ()->lambda_body
 - Parameter는 comma로 구분
 - Type(Class, var, 식별자)는 모두 넣거나 아니거나 선택 사항
- LambdaBody: 명령어가 1줄인 경우 규칙
 - LambdaBody의 대괄호 "{}"와 명령문의 ";"(Semicolon) 삭제
 - 반환 값이 있는 경우 "return" keyword 삭제

도전하세요!

- 다음 Comparator를 Lambda expression으로 변경하세요!

09:40 시작하겠습니다

```
Collections.sort(new LinkedList<String>(),  
    new Comparator<String>() {  
        @Override  
        public int compare(String o1, String o2) {  
            return Integer.compare(o1.length(), o2.length());  
        }  
    });
```

도전하세요!

- 다음 Callable을 Lambda expression으로 변경하세요!

```
Callable<List<String>> c = new Callable<List<String>>() {  
    public List<String> call() {  
        return List.of();  
    }  
};
```

도전하세요!

- 다음 Consumer를 Lambda expression으로 변경하세요!

```
Arrays.asList(5, 2, 6).stream().forEach(  
    new Consumer<Integer>() {  
        @Override  
        public void accept(Integer t) {  
            System.out.println(t);  
        }  
    }  
);
```

도전하세요!

- 다음 Function을 Lambda expression으로 변경하세요!

```
List<String> cos = List.of(Locale.getISOCountries())  
    .stream().map(new Function<String, String>() {  
        @Override public String apply(String s) {  
            return s.toLowerCase();  
        }  
    }).collect(Collectors.toList());
```

LGE Internal Use Only
SW College
학습자용

Method Reference Expression

sanghyuck.na@lge.com

Method Reference Expression⁸

- Method Reference Expression([MRE](#))
 - 기호 " :: "로 대상 method를 특정하여 "메소드" 자체에 대한 참조방법
 - MRE4: Instance와 Array 생성은 마치 메소드 호출인 것과 같이 보이게 처리

	종류	표현식
1	Static method	ContainingClass::staticMethodName
2	instance method	ContainingObject::instanceMethodName
3	Class-instance method	ContainingType::methodName
4	Constructor	ClassName::new

MethodReference:

```
ExpressionName :: [TypeArguments] Identifier
Primary :: [TypeArguments] Identifier
ReferenceType :: [TypeArguments] Identifier
super :: [TypeArguments] Identifier
TypeName . super :: [TypeArguments] Identifier
ClassType :: [TypeArguments] new
ArrayType :: new
```

MRE 1 – ContainingClass::staticMethodName

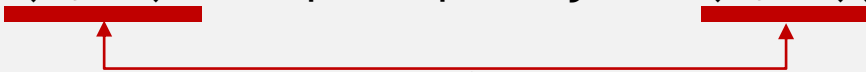
- staticMethodName
 - static으로 선언된 메소드
 - LambdaParameters의 타입+개수가 LambdaBady의 대상함수의 입력Parameter 타입+개수와 동일한 경우 적용
- 변환방법
 - 양쪽 중복Parameters와 괄호 제거
 - 람다표현식 기호 -> 삭제
 - LambdaBady에 대상함수의 오른쪽 마지막 참조기호 "." → "::"로 변경
- JRE: Method Type Argument 추론

```
Collections.sort(new LinkedList<Integer>(),  
                 (l, r) -> Integer.compare(l, r));  
Collections.sort(new LinkedList<String>(), Integer::compare);
```


MRE 2 – ContainingObject::instanceMethodName

- instanceMethodName
 - static method가 아닌 instance method 참조
- 변환방법
 - 양쪽 중복Parameters와 괄호 제거
 - 람다표현식 기호 -> 삭제
 - LambdaBody에 instance variable과 instanceMethodName 사이에 참조기호 "." → "::" 변경

```
class ComparisonProvider {  
    int compareByName(String a, String b) {  
        return a.compareTo(b);  
    }  
}  
String[] stringArray = { "Barbara", "James"};  
  
ComparisonProvider cp = new ComparisonProvider();  
Arrays.sort(stringArray, (l, r) -> cp.compareByName(l, r));  
Arrays.sort(stringArray, cp::compareByName);
```



MRE 3 – ContainingType::methodName

- ContaitngType
 - LambdaParameters의 첫 번째 parameter의 instance methodName의 method참조
- First parameter instance method
 - Second 이후 Parameters는 그 instance method의 input parameter로 배치
- 변환방법
 - 왼쪽 LambdaParameters 제거
 - 오른쪽 LambdaBady 참조변수와"."제거, 두번째 이후 (input parameters)제거
 - 람다표현식 기호 -> 삭제
 - 첫 번째 parameter의 " ClassName:: " 를 Instance methodName 앞에 추가

```
String[] stringArray = { "Barbara", "James"};
```

```
Arrays.sort(stringArray, (l, r) -> l.compareToIgnoreCase(r));
```



```
Arrays.sort(stringArray, String::compareToIgnoreCase);
```

MRE 3 – ContainingType::methodName

- 단일 형식 매개변수 목록의 인자
 - LambdaParameters의 첫 번째 parameter 그대로 메소드 참조에 사용
 - LambdaParameters의 두 번째 parameter 가 없기 때문에 JRE는 추론 종료

```
String[] stringArray = { "Barbara", "James"};

Arrays.stream(stringArray).map((a) -> a.toUpperCase());

Arrays.stream(stringArray).map(String::toUpperCase);
```

도전하세요!

- 메소드 myComparaing()를 Collections.sort()에 적용 해보세요
- myComparaing()의 Lambda 표현식은 몇 번째 MR규칙으로 바꿀 수 있나요?

```
static Comparator<String> myComparaing() {  
    return (o1, o2) ->  
        Integer.compare(o1.length(), o2.length());  
}
```

```
Collections.sort(new LinkedList<String>(),  
    (o1, o2) -> Integer.compare(o1.length(), o2.length())  
);
```

도전하세요!

- 다음 `String::length`는 MRE 몇 번째 인가요?
- 함수의 Input parameter로 `String::length`를 [Function](#)<String, Integer>으로 전달 받을 수 있습니다. 이점을 활용해서 `myComparing2()` 를 완성하세요!

10:35에 시작하겠습니다

1. 강의록 오타 수정해서 github에 올렸습니다. git pull하세요

```
Collections.sort(new LinkedList<String>(),
    myComparing2(s -> s.length()));

Collections.sort(new LinkedList<String>(),
    myComparing2(String::length));

Comparator<String> myComparing2(
    Function<String, Integer> length) {

    // ?

}
```

도전하세요!

- myComparing2 를 복사해서 myComparing3를 만드세요.
- myComparing3를 Generics 으로 일반화 하세요.

```
Collections.sort(new LinkedList<String>(),  
                 myCmp(String::length));  
  
Comparator<String> myComparing3 (  
    Function<String, Integer> length) {  
    // ??  
}
```

도전하세요!

- MR로 바꿔 보세요!

```
List.of(1, 2).forEach(  
    item -> System.out.println(item));
```


```
class Main { void runFor() {}  
Main m = new Main();  
Thread t = new Thread(()->m.runFor());
```

```
Collections.sort(new LinkedList<String>()  
    , (l, r) -> l.compareToIgnoreCase(r));
```

MRE 4 – ClassName::new

- 생성자 참조(Constructor Reference)
 - 객체 생성 Lambda표현식 () -> new Classname<>()
 - Diamond(Generic)가 단지 생략됐기 때문에 raw type이 아닙니다.
- 변환방법
 - 양측 Parameter "()" 부분과 "->" 제거
 - Generics 구문 제거
 - "Classname::"의 뒤에 "new" 결합

```
Arrays.stream(new String[] { "1", "2" })  
    .collect(Collectors.toCollection(()->new ArrayList<>()));  
Arrays.stream(new String[] { "1", "2" })  
    .collect(Collectors.toCollection(ArrayList::new));
```



```
ArrayList<String>::new // Parameterized type constructor  
ArrayList::new // Generic class 전달 추론타입인자  
Foo::<Integer>new // Generic constructor 전달 지정타입인자  
Bar<String>::<Integer>new // Generic class, Generic constructor  
Outer.Inner::new // inner class constructor  
int[]::new // 배열 생성
```


MRE 4 – ClassName::new

- Type-closed method
 - Method가 대상 Concrete class 정의+생성, 초기화실행

```
List<Integer> newSampleList() {  
    List<Integer> list = new ArrayList<>();  
    list.add(1);  
    return list;}  
List<Integer> data = newSampleList();
```

- Type-closed client
 - Client code가 Concrete class 정의+생성,
 - Method가 초기화실행

```
void newSampleList(List<? super Integer> list) {  
    list.add(3);}  
List<Integer> data = new ArrayList<>();  
newSampleList(data);
```

MRE 4 – ClassName::new

- client는 Concrete class정의, method는 생성+초기화실행
 - client는 가독성향상, method는 실행만 하여 최대한 실행지연
 - Method Reference사용으로 JVM 성능 최적화(Directly linked to the target method)
- 변환 방법
 - $\text{new } [\text{TypeArguments}] \text{ ClassType}(A_1, \dots, A_n) \rightarrow \text{ClassType} :: [\text{TypeArguments}] \text{ new}$
 - A_1, \dots, A_n 는 Parameter
- [Constructor Reference]의 type은 Supplier class입니다

```
List<Integer> newSampleList(Supplier<List<Integer>> s) {  
    List<Integer> list = s.get();  
    list.add(3);  
    return list;  
}  
  
List<Integer> data = null;  
data = newSampleList(()->new ArrayList<Integer>());  
data = newSampleList(ArrayList<Integer>::new);  
data = newSampleList(ArrayList::new);
```

도전하세요

- ?를 람다표현식으로 바꿔보세요!

```
Arrays.stream(new String[] { "1", "2" }).map(Integer::new);
```

```
Arrays.stream(new String[] { "1", "2" }).map(?);
```

11:30에 시작하겠습니다

```
IntStream.range(0, 10)  
    .collect(LinkedList::new, LinkedList::add,  
            LinkedList::addAll);
```

```
IntStream.range(0, 10).collect(?, ?, ?);
```

MRE – 4 ClassName::new

- 배열생성표현식
 - 배열 생성구문을 바꾸는 기법. size는 배열크기, k는 차원수
 - Array Constructor Reference
- 변환법
 - (size) -> **new** Type [size] []^{k-1} → Type[]^k :: **new** (k ≥ 1)
 - ()은 생략하지만 []는 Array선언 유지를 위해 생략하지 않습니다.

```
List.of(1, 2).stream()  
    .toArray((int size) -> new Integer[size]);
```

```
List.of(1, 2).stream()  
    .toArray(Integer[]:: new);
```

```
1. list.stream().toArray((int size) → new Integer[size]);  
2. list.stream().toArray(new Integer[]);  
3. list.stream().toArray(Integer[]:: new);
```

MRE – 4 ClassName::new

- Array Constructor Reference type은 IntFunction class입니다
 - Input Parameter는 고정: int
 - R은 사용자정의: Integer[]

```
@FunctionalInterface
interface Stream extends ... {
    <A> A[] toArray(IntFunction<A[]> generator);
}
```

```
@FunctionalInterface
interface IntFunction<R> {
    R apply(int value)
}
```

```
toArray((int size) -> new Integer[size]);
```

조심하세요

- 다음 중 오류는 무엇일까요?

```
List<Integer> list = List.of(1, 2);
```

- ① (Integer[]) list.toArray();
- ② (Integer[]) list.toArray(new Integer[0]);
- ③ list.stream().toArray(Integer[]::new);

정리

	Usage	Expression
1	Static method 참조	ContainingClass::staticMethodName
2	instance method 참조	ContainingObject::instanceMethodName
3	Class-instance method 참조	ContainingType::methodName
4	Constructor 참조	ClassName::new

확인하세요!

- super와 this의 MRE2
 - 자신은 this, 부모는 super

```
class Parent {  
    void run() {}  
}  
  
class Child extends Parent {  
    void run() {}  
  
    void startThread () {  
        new Thread(super::run).start();  
        new Thread(this::run).start();  
    }  
}
```


LGE Internal Use Only
SW College
학습자용

Functional interface type

sanghyuck.na@lge.com

Functional interface type

Capture

Lambda Design pattern

Functional interface type

- 람다표현식을 대표하는 타입
 - 단일 abstract method를 갖는 interface로 람다표현식을 단순히 "대표"하는 타입
 - 연산자 " :: "는 method 추출
- Functional interface naming convention
 - Parameter type과 개수에 대한 규칙과 Return Type에 따라 고유 명사로 결정

```
interface Task { void invoke(); }

void main() {
    Runnable r = () -> System.out.println("hi");

    Task t = r::run;
    Runnable r2 = t::invoke;
}
```

Functional interface type

- Naming Convention

- Prefix : parameter 타입과 그 개수에 특화 된 수량 및 타입 표현
- Suffix: return type 특화되어 고유명사 표현
- Type parameter T R U: 입력은 T와 U, 리턴은 R

	Function Interface	Method signature
Basic	Function<T,R>	R (T t)
	Consumer<T>	void (T t)
	Predicate<T>	boolean Test(T t)
	Supplier<T>	T ()
	BiFunction<T,U,R>	R apply(T, U)
Derived	UnaryOperator<T>	T apply(T) derived from Function<T,T>
	BinaryOperator<T>	T apply(T, T) derived from BiFunction<T,T,T>
	ToIntFunction	int applyAsInt(T value)
	DoubleConsumer	void accept(double value)

도전하세요!

- `void sort(List<T> list, Comparator<? super T> c)`
- 다음 보기에서 오류가 발생합니다. 원인은?
- 선택한 답을 수정해서 오류를 제거해보세요

```
Comparator<String> com = (o1, o2) ->
    Integer.compare(o1.length(), o2.length());

ToIntBiFunction<String, String> tib = (o1, o2) ->
    Integer.compare(o1.length(), o2.length());

LinkedList<String> lst = new LinkedList<>();
① Collections.sort(lst, com);
② Collections.sort(lst, tib);
```

도전하세요!

- 다음 ?을 완성하세요

```
? task = () -> System.out.println("h");

? myWork = () -> 0;

? myOperand = (Integer e) -> e + 1;

? myBinary = (double l, double r) -> l * r;

? myPinter = (String e) -> System.out.println(e);

? criteria = (String e) -> e.length() > 5;

? myException = () -> {
    if (5 % 2 > 10) { return 0;}
    throw new Exception();
};
```

정리

- Functional interface type

13:30에 시작하겠습니다

1. 강의록 수정해서 업로드했습니다. download하세요!

<https://github.com/kariosm/ajava1911>

도전하세요!

- 다음 myComparing3를 완성하세요

```
Collections.sort(new LinkedList<String>(),  
                 myCmp(String::length));  
  
Comparator<String> myComparing3 (  
    Function<String, Integer> length) {  
    // ??  
}
```

Capture

- 람다표현식 내부에서 인접 변수참조
 - Local class body 또는 Lambda body 안에서 인접 지역변수와 매개변수 참조 (Read)
- final
 - 내부 Local class, Lambda body는 외부Local variable과 Parameter은 final여야 함
 - 명시적 Final선언이 없다면 Effective final(자동 final상승)이고 final과 동일
 - final 임에 불구하고 외부에서 수정된다면 내부에서 사용될 수없음(Compile error)
- Design issue 3

```
void memberMethod(int param) {  
    class LocalClass {  
        void child() {System.out.println(param);}  
    }  
    new LocalClass().child();  
  
    int localVar = 2;  
    Runnable r = () -> {System.out.println(localVar)};  
    r.run();  
}
```


Undefined Behavior

- Captured variable dst
 - dst는 내부상태수정은 가능하며, 표준은 이경우 예상치 못한 결과가 발생할 수 있음
 - dst.add()는 compile오류가 발생하지 않으며 데이터 추가함
 - Effective final로 잘못된 참조를 쉽게 인지하기 어렵습니다

```
List<Integer> src = List.of(1, 2, 3);
List<Integer> dst = new Vector<>(); // Fail-fast
List<Thread> ts = new ArrayList<>();

for (Integer p : src) {
    Thread t = new Thread(() -> {dst.add(p)});
    t.start();
    ts.add(t);
}

ts.stream().forEach(t -> {
    try {t.join();} catch (InterruptedException e) { }
});
```

Memory Leak

- 외부 this
 - 내부에서 this접근은 인접 class Main reference으로 접근유지로 인한 leak
 - 내부에서 외부 this를 유지하는 동안 Main은 참조되고 있어 JVM generation에서 Garbage collection 되지 않아 leak

```
class Main {  
    static void main(String[] args) {  
        Runnable r = () ->{Main my_this = Main.this;};  
        new Thread(r).start(); ...  
    }  
};
```

```
class Main {  
    Runnable r = () -> {  
        final WeakReference<Main> my_this  
            = new WeakReference<>(this);  
    }  
};
```

Memory Leak

- Standalone
 - "WeakReference"로 Strong reference를 피함

```
class LeakActivity extends Activity {  
    final Handler handler = new Handler() {  
        @Override  
        public void handleMessage(Message msg) {  
            Activity ref = LeakActivity.this;  
        }  
    }  
}
```

```
class NonLeakActivity extends Activity {  
    Handler handler = new Handler() {  
        WeakReference<NonLeakActivity> ref  
            = new WeakReference<>(NonLeakActivity.this);  
  
        public void handleMessage(Message msg) {  
            NonLeakActivity act = ref.get();  
            if (act != null) { }  
        }  
    }  
}
```

Shadowing Scope

- [Shadowing](#)
 - 지역변수가 같은 이름의 Attribute, Parameter을 가리는 현상
 - 지역변수, 생성자와 메소드의 매개변수, 클래스는 Shadow관계
- Lambda body 참조 유효 범위
 - LambdaParameters의 선언은 지역변수 Shadow 규칙을 적용받음
 - Lambda body의 참조는 shadow 규칙에 따라 인접한 동일이름 대상이 참조됨

```
Path f = Paths.get("C:\\Log.log"); // p1

void testMethod () {
    Integer f = Integer.valueOf(0); // p2

    Comparator<String> c = (f, s) // p3
        -> Integer.compare(f.length(), s.length());
}
```

도전 하세요!

- 다음 코드를 실행해보세요
- 잘못 사용된 부분을 확인하세요. 총 몇 개이고, 그 원인은 무엇인가요?

```
final Random R = new Random();
ArrayList<Integer> ints = new ArrayList<>();

@Test public void test() {
    ArrayList<Integer> ints = new ArrayList<>();

    R.ints().limit(10).forEach((v) -> {
        Main main = this;
        main.ints.add(v);
    });

    System.out.println(ints);
}
```

정리

- Lambda Capture
- Undefined Behavior
- Memory Leak
- Shadowing Scope

Deferred

- Design pattern 2
- Deferred execution
 - 최종결정 시까지 실행을 지연. Parameter와 그 Code는 상황에 따라 실제로 실행되지 않을 수 있고(0) 여러 번 실행(*) 해 할 수 있습니다.
 - "x = " + 2는 info() method호출 시 결합 처리됐지만, 사용되지 않습니다

```
Logger.info("x =" + 2);  
class Logger {  
    enum Level { Error, Info, Verbose }  
    static final Level gLevel = Level.Error;  
    static void info(String msg) {  
        if (Level.Info.compareTo(gLevel) <= 0) {  
            // gLevel < Error  
        }  
    }  
}
```

```
Logger.info(() -> "x = " + 3);  
public static void info(Supplier<String> msg) {  
    if (Level.Info.compareTo(gLevel) <= 0) {  
        System.out.println(msg.get());  
    }  
}
```

Composition

- Lambda의 핵심원리 Composition

- 메소드 단위호출을 결합하는 방법.
- 동일한호출이 지속된다면 그것을 결합하는 게 낫습니다. 더불어 메소드단위호출은 임시 결과를 유지의무도 수반합니다.
- Lambda는 여러 메소드호출을 하나로 묶을 수(Composition) 있습니다. 메소드호출을 결합하면 중간결과와 임시저장소에 대한 JVM에게 최적화작업을 전가함

```
UnaryOperator<Integer> plus = (f) -> f + 10;  
UnaryOperator<Integer> mul = (s) -> s * 5;  
Integer v = Integer.valueOf(1);  
Integer mid = plus.apply(begin);  
Integer end = mul.apply(mid);
```

```
UnaryOperator<Integer> all  
                        = t -> mul.apply(plus.apply(t));  
Integer end = all.apply(v);
```


도전하세요!

- 다음 모든 UnaryOperator<String> 을 하나로 결합하세요
 - 아래 코드는 "-- ** ++ Lambda ++ ** --" 반환
- Random r의 r.nextInt()결과가 짝수만 그 결합 함수를 호출하도록 하세요.

```
String txt = "Lambda";

UnaryOperator<String> minus = t -> "-- " + t + " --";
UnaryOperator<String> mul   = t -> "** " + t + " **";
UnaryOperator<String> plus  = t -> "++ " + t + " ++";

System.out.println(minus.apply(mul.apply(plus.apply(txt))));

Random r = new Random();
```

정리

- Deferred
- Composition