KB금융그룹 | 국민의 평생 금융파트너

2025년 상반기 K-디지털 트레이닝

람다식

[KB] IT's Your Life

수학에서는 매개 인자라는 뜻

함수도 데이터처럼 다루자

보편적인 형식
변수 = 데이터;

함수도 데이터처럼
변수 = 함수;

함수가 변수에 대입되게금 .매개변수에 변수에 반환에 사용

함수적 프로그래밍 패러다임이 좋은데 자바는 함수라는 개념이 없고
객체기준이며 객체 중심에서 메서드라고 한다.

자바에 함수형 프로그래밍이 잘 안맞음.
자바 새로운 버전에서 함수형 프로그래밍에 잘 맞게끔 변경함.
함수와 제일 유사한게 뭐가 있을까 거기에다가 함수형 프로그래밍 매커니즘을
적용시키자. => 메서드가 하나 밖에 없는 객체가 형태적으로 유사
=>람다라는 형식이 탄생

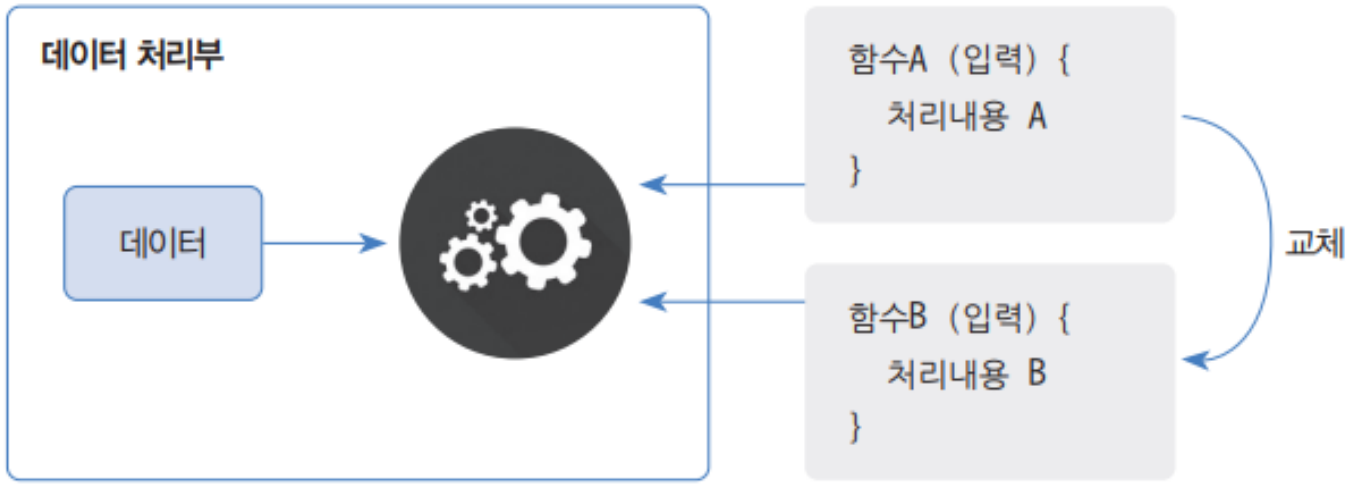
KB 국민은행

KB 국민은행

람다식

Calculable func = (x,y) -> {return x+y; }

- 함수형 프로그래밍: 함수를 정의하고 이 함수를 데이터 처리부로 보내 데이터를 처리하는 기법
- 데이터 처리부는 제공된 함수의 입력값으로 데이터를 넣고 함수에 정의된 처리 내용을 실행
- 람다식: 데이터 처리부에 제공되는 함수 역할을 하는 매개변수를 가진 중괄호 블록이다.
- 자바는 람다식을 익명 구현 객체로 변환



... action(Calculable arg){ ... }

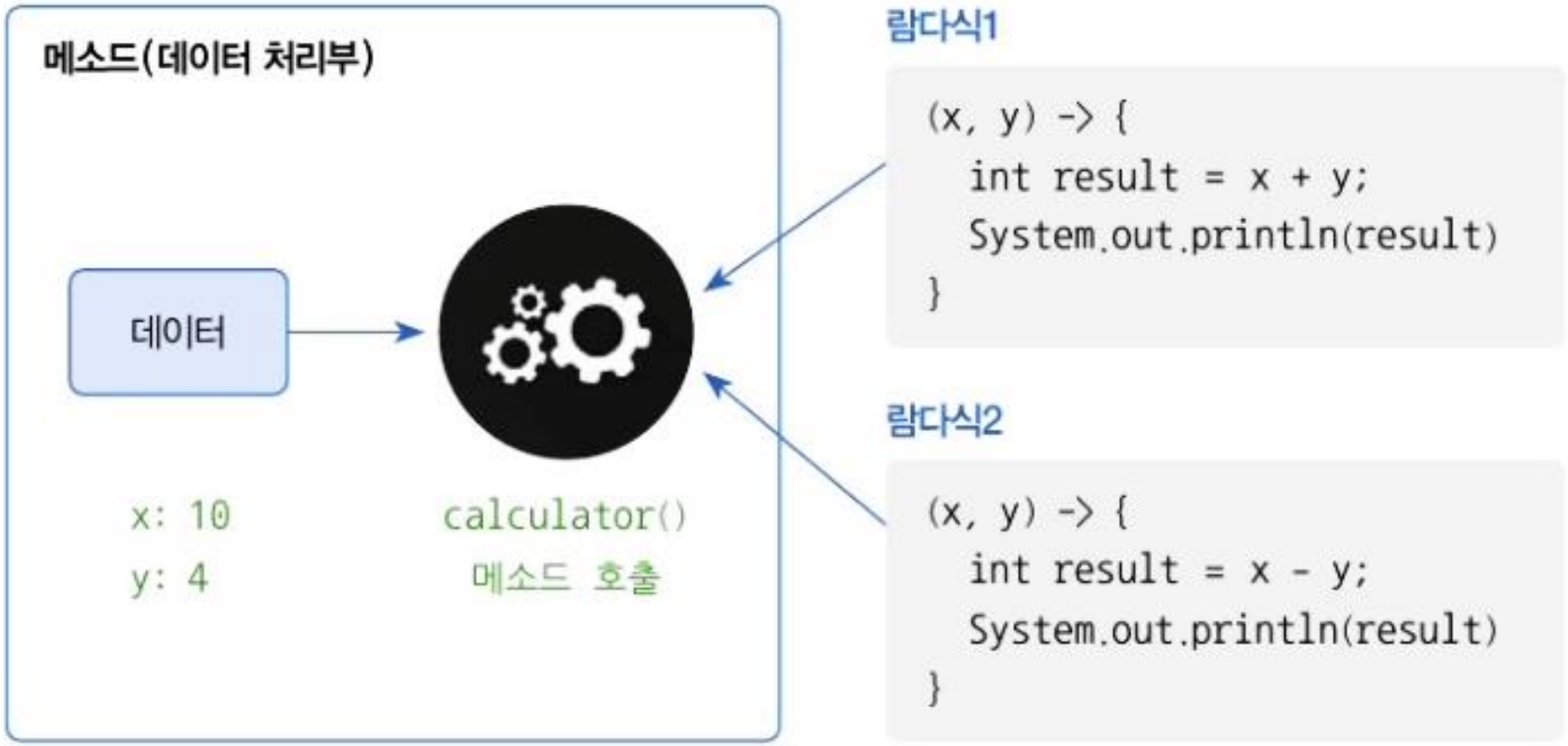
```
람다식:  (매개변수, …) -> { 처리 내용 }
```

```
public interface Calculable {
  //추상 메소드
  void calculate(int x, int y);
}
```

익명 구현 객체

```
action( (x, y) -> {
  int result = x + y;
  System.out.println(result);
});
```

람다식


메소드(데이터 처리부)
데이터
x: 10
y: 4
calculator()
메소드 호출
람다식1
(x, y) -> {
int result = x + y;
System.out.println(result)
}
람다식2
(x, y) -> {
int result = x - y;
System.out.println(result)
}

함수형 인터페이스

- 인터페이스가 단 하나의 추상 메소드를 가지는 것

인터페이스

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

람다식

```
( ) -> { ··· }
```

인터페이스

```
@FunctionalInterface
public interface Calculable {
  void calculate(int x, int y);
}
```

람다식

```
( x, y ) -> { ··· }
```

- @FunctionalInterface: ~ lombok이 아닌 그냥 annotation
  - 인터페이스가 함수형 인터페이스임을 보장
  - 컴파일 과정에서 추상 메소드가 하나인지 검사 --> 위배시 에러 처리

Calculable.java

```
package ch16.sec01;

@FunctionalInterface
public interface Calculable {
  //추상 메소드
  void calculate(int x, int y);
}
```

LambdaExample.java

```
package ch16.sec01;

public class LambdaExample {
  public static void main(String[] args) {
    action((x, y) -> {
      int result = x + y;
      System.out.println("result: " + result);
    });

    action((x, y) -> {
      int result = x - y;
      System.out.println("result: " + result);
    });
  }

  public static void action(Calculable calculable) {
    //데이터
    int x = 10;
    int y = 4;
    //데이터 처리
    calculable.calculate(x, y);
  }
}
```

저 매개변수 인자로
1. 구현 클래스를 사용
2. 익명 구현 객체 사용
new Calculable(){
@Override
void calculaotr(){ }
}
3. 위에 두개 너무 오래걸려 만들기
=>람다식

```
result: 14
result: 6
```

매개변수가 없는 람다식

- 함수형 인터페이스의 추상 메소드에 매개변수가 없을 경우 람다식 작성하기
- 실행문이 두 개 이상일 경우에는 중괄호를 생략할 수 없고, 하나일 경우에만 생략할 수 있음

```
( ) -> {
  실행문;
  실행문;
}
```

```
( ) -> 실행문
```

Workable.java

```
package ch16.sec02.exam01;

@FunctionalInterface
public interface Workable {
  void work();
}
```

Person.java

인터페이스로 프로그래밍하는 좋은 예

```
package ch16.sec02.exam01;

public class Person {
  public void action(Workable workable) {
    workable.work();
  }
}
```

LambdaExample.java

```
package ch16.sec02.exam01;

public class LambdaExample {
  public static void main(String[] args) {
    Person person = new Person();

    //실행문이 두 개 이상인 경우 중괄호 필요
    person.action(() -> {
      System.out.println("출근을 합니다.");
      System.out.println("프로그래밍을 합니다.");
    });

    //실행문이 한 개일 경우 중괄호 생략 가능
    person.action(() -> System.out.println("퇴근합니다."));
  }
}
```

```
출근을 합니다.
프로그래밍을 합니다.
퇴근합니다.
```

Button.java

```
package ch16.sec02.exam02;

public class Button {
  //정적 중첩 함수형 인터페이스
  @FunctionalInterface
  public static interface ClickListener {
    //추상 메소드
    void onClick();
  }

  //필드
  private ClickListener clickListener;

  //메소드
  public void setClickListener(ClickListener clickListener) {
    this.clickListener = clickListener;
  }

  public void click() {
    this.clickListener.onClick();
  }
}
```

LambdaExample.java 람다 활용

```
package ch16.sec02.exam02;

public class ButtonExample {
  public static void main(String[] args) {
    Button btnOk = new Button();

    //Ok 버튼 객체에 람다식(ClickListener 익명 구현 객체) 주입
    btnOk.setClickListener(() -> {
      System.out.println("Ok 버튼을 클릭했습니다.");
    });

    btnOk.click(); //Ok 버튼 클릭하기

    Button btnCancel = new Button();

    //Cancel 버튼 객체에 람다식(ClickListener 익명 구현 객체) 주입
    btnCancel.setClickListener(() -> {
      System.out.println("Cancel 버튼을 클릭했습니다.");
    });

    btnCancel.click(); //Cancel 버튼 클릭하기
  }
}
```

```
Ok 버튼을 클릭했습니다.
Cancel 버튼을 클릭했습니다.
```

매개변수가 있는 람다식

- 함수형 인터페이스의 추상 메소드에 매개변수가 있을 경우 람다식 작성하기
- 매개변수를 선언할 때 타입은 생략할 수 있고, 구체적인 타입 대신에 var를 사용할 수 있음

c++ auto와 같음

```
(타입 매개변수, … ) -> {
  실행문;
  실행문;
}
```

```
(var 매개변수, …) -> {
  실행문;
  실행문;
}
```

```
(매개변수, …) -> {
  실행문;
  실행문;
}
```

```
(타입 매개변수, … ) -> 실행문
```

```
(var 매개변수, …) -> 실행문
```

```
(매개변수, …) -> 실행문
```

매개변수가 있는 람다식

- 매개변수가 하나일 경우에는 괄호를 생략 가능. 이때는 타입 또는 var를 붙일 수 없음

```
매개변수 -> {
   실행문;
   실행문;
}
```

```
매개변수 -> 실행문
```

Workable.java

```
package ch16.sec03;

@FunctionalInterface
public interface Workable {
  void work(String name, String job);
}
```

Speakable.java

```
package ch16.sec03;

@FunctionalInterface
public interface Speakable {
  void speak(String content);
}
```

Person.java

```
package ch16.sec03;

public class Person {
  public void action1(Workable workable) {
    workable.work("홍길동", "프로그래밍");
  }

  public void action2(Speakable speakable) {
    speakable.speak("안녕하세요");
  }
}
```

LambdaExample.java

```
package ch16.sec03;

public class LambdaExample {
  public static void main(String[] args) {
    Person person = new Person();

    //매개변수가 두 개일 경우
    person.action1((name, job) -> {
      System.out.print(name + "이 ");
      System.out.println(job + "을 합니다.");
    });
    person.action1((name, job) -> System.out.println(name + "이 " + job + "을 하지 않습니다."));

    //매개변수가 한 개일 경우
    person.action2(word -> {
      System.out.print("\"" + word + "\"");
      System.out.println("라고 말합니다.");
    });
    person.action2(word -> System.out.println("\"" + word + "\"라고 외칩니다."));
  }
}
```

```
홍길동이 프로그래밍을 합니다.
홍길동이 프로그래밍을 하지 않습니다.
"안녕하세요"라고 말합니다.
"안녕하세요"라고 외칩니다.
```

리턴값이 있는 람다식

- 함수형 인터페이스의 추상 메소드에 리턴값이 있을 경우 람다식 작성하기
- return 문 하나만 있을 경우에는 중괄호와 함께 return 키워드를 생략 가능
- 리턴값은 연산식 또는 리턴값 있는 메소드 호출로 대체 가능

```
(매개변수, … ) -> {
  실행문;
  return 값;
}
```

```
(매개변수, …) -> return 값;
(매개변수, …) -> 값
```

Calcuable.java

```
package ch16.sec04;

@FunctionalInterface
public interface Calcuable {
  double calc(double x, double y);
}
```

Person.java

```
package ch16.sec04;

public class Person {
  public void action(Calcuable calcuable) {
    double result = calcuable.calc(10, 4);
    System.out.println("결과: " + result);
  }
}
```

LambdaExample.java

```
package ch16.sec04;

public class LambdaExample {
  public static void main(String[] args) {
    Person person = new Person();

    //실행문이 두 개 이상일 경우
    person.action((x, y) -> {
      double result = x + y;
      return result;
    });

    //리턴문이 하나만 있을 경우(연산식)
    //person.action((x, y) -> {
    // return (x + y);
    //});
    person.action((x, y) -> (x + y));

    //리턴문이 하나만 있을 경우(메소드 호출)
    //person.action((x, y) -> {
    // return sum(x, y);
    //});   다른 메서드 호출도 가능.
    person.action((x, y) -> sum(x, y));
  }

  public static double sum(double x, double y) {
    return (x + y);
  }
}
```

```
결과: 14.0
결과: 14.0
결과: 14.0 반환형식이 달라도 변환된다면 ㄱㅊ
```

그냥 표현식이면 다 가능

메소드 참조

메소드 참조

- 메소드를 참조해 매개변수의 정보 및 리턴 타입을 알아내 람다식에서 불필요한 매개변수를 제거

이런 형식일 경우

```
(left, right) -> Math.max(left, right);
```

-> 단순히 두개의 값을 Math.max( )메소드의 매개값으로 전달하는 역할만 함.

- 메소드 참조로 단순화 가능

이런 형식으로 대체 가능

```
Math :: max;
```

정적 메소드와 인스턴스 메소드 참조

정적 메소드를 참조 시

- 클래스 이름 뒤에 :: 기호를 붙이고 정적 메소드 이름을 기술

```
클래스 :: 메소드
```

인스턴스 메소드일 경우

- 객체를 생성한 다음 참조 변수 뒤에 :: 기호를 붙이고 인스턴스 메소드 이름을 기술

```
참조변수 :: 메소드
```

Calcuable.java

```
package ch16.sec05.exam01;

@FunctionalInterface
public interface Calcuable {
  double calc(double x, double y);
}
```

리턴과 매개변수 주의
더블 더블2개

Person.java

```
package ch16.sec05.exam01;

public class Person {
  public void action(Calcuable calcuable) {
    double result = calcuable.calc(10, 4);
    System.out.println("결과: " + result);
  }
}
```

Computer.java

```
package ch16.sec05.exam01;

public class Computer {
  public static double staticMethod(double x, double y) {
    return x + y;
  }

  public double instanceMethod(double x, double y) {
    return x * y;
  }
}
```

리턴과 매개변수가
더블 더블2개

MethodReferenceExample.java

```
package ch16.sec05.exam01;

public class MethodReferenceExample {
  public static void main(String[] args) {
    Person person = new Person();

    //정적 메소드일 경우
    //람다식
    //person.action((x, y) -> Computer.staticMethod(x, y));
    //메소드 참조
    person.action(Computer :: staticMethod);

    //인스턴스 메소드일 경우
    Computer com = new Computer();
    //람다식
    //person.action((x, y) -> com.instanceMethod(x, y));
    //메소드 참조
    person.action(com :: instanceMethod);
  }
}
```

대응한다. 저렇게 대체 가능하다

리턴과 매개변수가 맞아야지
저렇게 대응(대체)가능

```
결과: 14.0
결과: 40.0
```

매개변수의 메소드 참조

- 람다식에서 제공되는 a 매개변수의 메소드를 호출해서 b 매개변수를 매개값으로 사용

```
(a, b) -> { a.instanceMethod(b); }
```

- a의 클래스 이름 뒤에 :: 기호를 붙이고 메소드 이름을 기술

```
클래스 :: instanceMethod
```

어떻게 구별해
클래스 메소드 인지 클래스::스태틱클래스메소드
인스턴스 메소드 인지 인스턴스::인스턴스메소드
클래스::인스턴스 메소드인지

Comparable.java

```
package ch16.sec05.exam02;

@FunctionalInterface
public interface Comparable {
  int compare(String a, String b);
}
```

Person.java

```
package ch16.sec05.exam02;

public class Person {
  public void ordering(Comparable comparable) {
    String a = "홍길동";
    String b = "김길동";

    int result = comparable.compare(a, b);

    if(result < 0) {
      System.out.println(a + "은 " + b + "보다 앞에 옵니다.");
    } else if(result == 0) {
      System.out.println(a + "은 " + b + "과 같습니다.");
    } else {
      System.out.println(a + "은 " + b + "보다 뒤에 옵니다.");
    }
  }
}
```

MethodReferenceExample.java

```
package ch16.sec05.exam02;

public class MethodReferenceExample {
  public static void main(String[] args) {
    Person person = new Person();
    person.ordering(String :: compareToIgnoreCase);  // (a, b) -> a.compareToIgnoreCase(b)
  }
}
```

햇갈리다 다시 공부

홍길동은 김길동보다 뒤에 옵니다.

매개변수 3개일 때도 가능

(a,b,c) -> a.method(b,c)

생성자 참조

- 객체를 생성하는 것. 람다식이 단순히 객체를 생성하고 리턴하도록 구성되면 람다식을 생성자 참조로 대치 가능

```
(a, b) -> { return new 클래스(a, b); }
```

- 클래스 이름 뒤에 :: 기호를 붙이고 new 연산자를 기술

```
클래스 :: new
```

타입과 갯수가 많는 생성자가 있을 경우 가능

- 생성자가 오버로딩되어 여러 개가 있을 경우, 컴파일러는 함수형 인터페이스의 추상 메소드와 동일한 매개변수 타입과 개수를 가지고 있는 생성자를 찾아 실행
- 해당 생성자가 존재하지 않으면 컴파일 오류 발생

Creatable1.java

```
package ch16.sec05.exam03;

@FunctionalInterface
public interface Creatable1 {
  public Member create(String id);
}
```

string 매개인자를 받아서 Member리턴

Creatable2.java

```
package ch16.sec05.exam03;

@FunctionalInterface
public interface Creatable2 {
  public Member create(String id, String name);
}
```

Computer.java

```
package ch16.sec05.exam03;

public class Member {
  private String id;
  private String name;

  public Member(String id) {
    this.id = id;
    System.out.println("Member(String id)");
  }

  public Member(String id, String name) {
    this.id = id;
    this.name = name;
    System.out.println("Member(String id, String name)");
  }

  @Override
  public String toString() {
    String info = "{ id: " + id + ", name: " + name + " }";
    return info;
  }
}
```

생성자들

Person.java

```
package ch16.sec05.exam03;

public class Person {
  public Member getMember1(Creatable1 creatable) {
    String id = "winter";
    Member member = creatable.create(id);
    return member;
  }

  public Member getMember2(Creatable2 creatable) {
    String id = "winter";
    String name = "한겨울";
    Member member = creatable.create(id, name);
    return member;
  }
}
```

ConstructorReferenceExample.java

```
package ch16.sec05.exam03;

public class ConstructorReferenceExample {
  public static void main(String[] args) {
    Person person = new Person();

    Member m1 = person.getMember1( Member :: new );
    System.out.println(m1);
    System.out.println();

    Member m2 = person.getMember2( Member :: new );
    System.out.println(m2);
  }
}
```

Member(String id)
{ id: winter, name: null }

Member(String id, String name)
{ id: winter, name: 한겨울 }