12. 내부 클래스.람다식 외

1. 내부 클래스
2. 람다식
3. 열거형
4. 가변인자
5. 어노테이션

1. 내부 클래스

❖ : 네스티드 클래스의 구분

```
class Outer {   // 외부 클래스
    class Nested {...}   // 네스티드 클래스
}
```

```
class OuterClass {
    static class StaticNestedClass {...}   // Static 네스티드 클래스
}
```

```
class OuterClass {
    class InnerClass {...}   // Non-static 네스티드 클래스, 이너 클래스
}
```

- 멤버 (이너) 클래스 (Member Inner Class)
- 로컬 (이너) 클래스 (Local Inner Class)
- 익명 (이너) 클래스 (Anonymous Inner Class)

1. 내부 클래스

❖ : 네스티드 클래스의 구분

변수

```
class MyClass
{
    static int n1;   // 정적(static) 변수
    int n2;          // 멤버 변수 (인스턴스 변수)

    public void myFunc()
    {
        int n3;      // 지역 변수
        …
    }
}
```

내부 클래스

```
class MyClass          // 외부 클래스
{
    static class NestedButNotInner { }   // static 네스티드 클래스
    class c1 { }       // 멤버 내부 클래스

    public void myFunc()
    {
        class c2 { }   // 지역 내부 클래스
        …
    }
}
```

1. 내부 클래스

Ex01_MemberInner

❖ 멤버 내부 클래스

- 멤버 내부 클래스의 객체는 외부 클래스의 객체에 종속적이다.
- 다른 클래스와는 연관되어 사용되지 않고 해당 클래스에서만 사용할 때 하나의 소스 파일로 관리를 편하게 할 수 있다.
- 외부 클래스는 내부클래스를 멤버변수처럼 사용할 수 있고, 내부 클래스는 외부 클래스의 자원을 직접 사용할 수 있는 장점이 있다.

실습1: 멤버 내부 클래스

```
class Outer1{
    private int speed = 10;

    class MemberInner1 {
        // 외부 클래스의 자원(speed) 사용 가능
        public void move()
        {
            System.out.printf("인간형 유닛이 %d 속도로 이동
합니다.\n", speed);
        }
    }

    public void getMarine()
    {
        MemberInner1 inner = new MemberInner1();
        inner.move();
    }
}
```

```
public class Ex01_MemberInner
{
    public static void main(String[] args)
    {
        Outer1 out = new Outer1();

        // out 기반으로 생성된 객체의 메서드 호출
        out.getMarine();

        // out 기반으로 내부 클래스 객체 생성
        Outer1.MemberInner1 inner = out.new MemberInner1();

        // inner 기반으로 생성된 객체의 메서드 호출
        inner.move();
    }
}
```

1. 내부 클래스

Ex02_LocalInner

❖ 지역 내부 클래스

- 지역 내부 클래스는 클래스의 정의 위치가 **메서드, if문, while** 문 같은 중괄호 블럭 안에 정의된다는 점에서 멤버 내부 클래스와 구분된다.

- 해당 메서드 안에서만 객체 생성이 가능해지므로 클래스의 정의를 깊이 숨기는 효과가 있다.

실습2: 지역 내부 클래스

```
class HumanCamp2 {
    private int speed = 10;

    public void getMarine() {
        class Marine2 {
            // 외부 클래스의 자원(speed) 사용 가능
            public void move() {
                System.out.printf("인간형 유닛이 %d 속도로 이동합니다.\n", speed);
            }
        }

        Marine2 inner = new Marine2();
        inner.move();
    }
}
```

```
public class Ex02_LocalInner
{
    public static void main(String[] args)
    {
        HumanCamp2 hc = new HumanCamp2();
        hc.getMarine();
    }
}
```

1. 내부 클래스

Ex03_AnonymousInner1

Ex04_AnonymousInner2

❖ 익명 내부 클래스

- 지역 내부 클래스는 해당 메서드에서만 클래스의 생성이 가능하므로 클래스의 이름이 상당히 제한적으로 사용된다.
- 그래서 클래스의 이름을 생략해 버리기도 한다.
- 이렇게 클래스의 이름을 생략한 것이 익명 내부 클래스이다.
- 익명 내부 클래스는 이후의 람다식과도 관련이 있다.

❖ 익명 내부 클래스는 예전에 자바 UI에서 이벤트를 처리하는 데 많이 사용했다.

❖ 현재는 안드로이드 프로그래밍에서 위젯의 이벤트 처리하는 핸들러를 구현할 때 사용한다.

실습3: 익명의 내부 클래스

```
interface Unit3 {
    void move();
}

class HumanCamp3 {
    private int speed = 10;

    public Unit3 getMarine()  {
        class Marine3 implements Unit3  {
            public void move() {
                System.out.printf("인간형 유닛이 %d 속도로 이동합니다.\n", speed);
            }
        }

        return new Marine3();
    }
}
```

```
public class Ex03_AnonymousInner1 {
    public static void main(String[] args) {
        HumanCamp3 hc = new HumanCamp3();
        Unit3 unit = hc.getMarine();
        unit.move();
    }
}
```

실습4: 익명의 내부 클래스2

```
interface Unit4 {
    void move();
}

class HumanCamp4 {
    private int speed = 10;

    public Unit4 getMarine() {
//        class Marine4 implements Unit4 {
//            public void move()  {
//                System.out.printf("인간형 유닛이 %d 속도로 이동합니다.\n", speed);
//            }
//        }
//        return new Marine4();

        // 이름이 없으므로 부모 클래스나 인터페이스의 이름을 사용
        return new Unit4() {
            public void move() {
                System.out.printf("인간형 유닛이 %d 속도로 이동합니다.\n", speed);
            }
        };    // 하나의 실행문이므로 세미콜론으로 끝납니다.
    }
}
```

```
public class Ex04_AnonymousInner2 {
    public static void main(String[] args) {
        HumanCamp4 hc = new HumanCamp4();
        Unit4 unit = hc.getMarine();
        unit.move();
    }
}
```

2. 람다식 (LAMBDA EXPRESSION)

❖ 람다식이란?

- 자바에서 함수형 프로그래밍(functional programming)을 구현하는 방식
- 자바8부터 지원
- 클래스를 생성하지 않고 함수의 호출만으로 기능을 수행
- 함수형 프로그래밍 :
  - 순수 함수(pure function)를 구현하고 호출함으로써 외부 자료에 부수 적인 영향을 주지 않고 매개 변수만을 사용하도록 만든 함수
  - 함수를 기반으로 구현
  - 입력 받은 자료를 기반으로 수행되고 외부에 영향을 미치지 않으므로 병렬처리등에 가능
  - 안정적인 확장성 있는 프로그래밍 방식

2. 람다식 (LAMBDA EXPRESSION)

❖ **람다의 이해**

일반 클래스 사용	Ex05_Lambda1
익명 내부 클래스로 사용	Ex06_Lambda2
람다식으로 변형	Ex07_Lambda3

실습1 : 람다식 이해

```
interface Unit5 {
    void move(String s);
}

class Human5 implements Unit5{
    public void move(String s)
   {
        System.out.println(s);
    }
}

public class Ex05_Lambda1 {
    public static void main(String[] args)
   {
        Unit5 unit = new Human5();
        unit.move("Lambda : Unit 5");
    }
}
```

```
interface Unit6 {
    void move(String s);
}

public class Ex06_Lambda2
{
    public static void main(String[] args)
    {
        Unit6 unit = new Unit6()
        {  // 익명 클래스
            public void move(String s) {
                System.out.println(s);
            }
        };
        unit.move("Lambda : Unit 6");
    }
}
```

```
interface Unit7
{
    void move(String s);
}

public class Ex07_Lambda3
{
    public static void main(String[] args)
    {
        Unit7 unit = (String s) ->
            {
                System.out.println(s);
            };
        unit.move("Lambda : Unit 7");
    }
}
```

2. 람다식 (LAMBDA EXPRESSION)

❖ 람다식 문법

매개변수가 하나인 경우 자료형과 소괄호를 생략할 수 있다.

```
str -> { System.out.println(str); }
```

중괄호 안의 구현부가 한 문장인 경우 중괄호를 생략할 수 있다.

```
str -> System.out.println(str);
```

중괄호 안의 구현부가 한 문장이라도 return 문이 있다면 중괄호를 생략할 수 없다.

```
str -> return str.length();      // 잘못된 형식
```

매개 변수가 두 개인 경우 소괄호를 생략할 수 없다.

```
x, y -> { System.out.println(x + y); }     // 잘못된 형식
```

2. 람다식 (LAMBDA EXPRESSION)

❖ 람다식 문법2

중괄호 안의 구현부가 반환문 하나라면 return 과 중괄호 모두 생략할 수 있다.

```
(x, y) -> x + y;        // 두 값을 더하여 반환함
str -> str.length();    // 문자열의 길이를 반환함
```

매개 변수가 없을 경우에는 소괄호를 생략할 수 없습니다.

```
( ) -> System.out.println("Hello~");
```

실습2: 람다식 룰

```
interface Unit8 {
    void move(String s);  // 매개변수 하나, 반환형 void
}

public class Ex08_LambdaRule1 {
    public static void main(String[] args)  {
        Unit8 unit;

        unit = (String s) -> { System.out.println(s); };
        unit.move("Lambda : 줄임 없는 표현 : 앞 예제 동일");

        unit = (String s) -> System.out.println(s);
        unit.move("Lambda : 중괄호 생략");

        unit = (s) -> System.out.println(s);
        unit.move("Lambda : 매개변수 형 생략");

        unit = s -> System.out.println(s);
        unit.move("Lambda : 매개변수 소괄호 생략");
    }
}
```

```
interface Unit9
{
    int calc(int a, int b);  // 매개변수 둘, 반환형 int
}

public class Ex09_LambdaRule2
{
    public static void main(String[] args)
    {
        Unit9 unit;
        unit = (a, b) -> { return a + b; };
        //unit = a, b -> { return a + b; }; // 앞쪽 소괄호 생략 안됨
        //unit = (a, b) -> return a + b;    // 뒤쪽 중괄호 생략 안됨
        int num = unit.calc(10, 20);
        System.out.println(num);

        unit = (a, b) -> a * b;  // 뒤쪽 중괄호와 return 생략 가능
        System.out.println(unit.calc(10, 20));
    }
}
```

실습2: 람다식 룰

```
interface Unit10
{
    String move();  // 매개변수 없슴, 반환형 String
}

public class Ex10_LambdaRule3
{
    public static void main(String[] args)
    {
        Unit10 unit = () -> {
            return "인간형 유닛 이동";
        };

        System.out.println(unit.move());
    }
}
```

2. 람다식 (LAMBDA EXPRESSION)

❖ 함수형 인터페이스

- 람다식을 선언하기 위한 인터페이스
- 익명 함수와 매개 변수만으로 구현되므로 단 하나의 메서드만을 가져야 함
- 두 개 이상의 메서드인 경우 어떤 메서드의 호출인지 모호해 짐
- @FunctionalInterface 어노테이션
- 함수형 인터페이스라는 의미, 여러 개의 메서드를 선언하면 에러남

Ex11_Functional.java

```
1 @FunctionalInterface
2 Invalid '@FunctionalInterface' annotation; Unit11 is not a functional interface
3 {
4     String move();
5     void attack();
6 }
```

← 메서드가 2개이므로 오류 발생

실습3: 함수형 인터페이스

```
@FunctionalInterface
interface Unit11 {
    String move();
//    void attack();
}

public class Ex11_Functional {
    public static void main(String[] args)
    {
        Unit11 unit = () -> {
            return "인간형 유닛 이동";
        };

        System.out.println(unit.move());
    }
}
```

3. 열거형(ENUM)

❖ 열거형이란?

- 관련된 상수들을 같이 묶어 놓은 것. Java는 타입에 안전한 열거형을 제공

```
interface MyNum {
    public static final int SPRING = 0;
    public static final int SUMMER = 1;
    public static final int FALL = 2;
    public static final int WINTER = 3;

    public static final int DO = 0;
    public static final int RE = 1;
    public static final int MI = 2;
    public static final int FA = 4;
    public static final int SOL = 5;
    public static final int RA = 6;
    public static final int SI = 7;
}
```

앞에서 배운 것을 상기해보면 인터페이스에 사용된 변수는 앞에 public final static 이 생략된 것이다.
그러므로 이 모든것들은 final 상수들이다.

3. 열거형(ENUM)

❖ 이전 방식의 문제점

- 다음 코드에서 MAN 과 TANK는 같은 값을 가지고 있다.
- 그러기 때문에 잘못 사용하면 의미 전달에 있어서 모호함이 나타날 수 있다.

- MAN과 TANK 둘은 상수를 정의해서 사용한 의미는 다르지만, 결국은 같은 숫자로 판정된다.

- 섞여서 사용되었다면 비록 실행 중에 에러가 발생하지는 않았지만 의도했다면 잘못 생각한 것이고, 그렇지 않다면 실수가 있는 프로그램이다.

```
interface Human1 {
    int MAN = 1;
    int WOMAN = 2;
}

interface Machine1 {
    int TANK = 1;
    int AIRPLANE = 2;
}
```

실습 : 열거형 1

```
interface Human1 {
   int MAN = 1;
   int WOMAN = 2;
}

interface Machine1 {
   int TANK = 1;
   int AIRPLANE = 2;
}
```

```
public class Ex01_Constants {
    public static void main(String[] args) {
        createUnit(Machine1.TANK);

        createUnit(Human1.MAN); // 잘못된 상수 사용
    }

    public static void createUnit(int kind) {
        switch(kind) {
        case Machine1.TANK:
            System.out.println("탱크를 만듭니다.");
            break;
        case Machine1.AIRPLANE:
            System.out.println("비행기를 만듭니다.");
            break;
        }
    }
} //컴파일 및 실행 과정에서 발견되지 않는 오류
```

3. 열거형(ENUM)

❖ 열거형 기반으로 수정하여 개선

열거형 값 (Enumerated Values)

```
enum Human2 { MAN, WOMAN }

enum Machine2 { TANK, AIRPLANE }

public class Ex02_Enum
{
    public static void main(String[] args)
    {
        welcomeGuest(Machine2.TANK);

//        welcomeGuest(Human2.DOG); // 잘못된 상수 사용 : 에러

        // 참고 : C처럼 숫자로 비교하면 에러가 난다.
//        if (Human2.MAN == 0) {
//
//        }
    }
```

파일 과정에서 자료형 불일치로 인한 오류 발생

Switch 문에서는 case문에서 표현의 간결함을 위해 TANK 와 같이 '열거형 값'의 이름만 명시하기로 약속되어 있다.

실습 : 열거형2

```
enum Human2 { MAN, WOMAN }

enum Machine2 { TANK, AIRPLANE }
```

```
public class Ex02_Enum {
    public static void main(String[] args) {
                createUnit(Machine2.TANK);

//        createUnit(Human2.DOG); // 잘못된 상수 사용 : 에러
        // 참고 : C처럼 숫자로 비교하면 에러가 난다.
//              if (Human2.MAN == 0) {
//
//              }
    }

    public static void createUnit(Machine2 kind) {
        switch(kind) {
        case TANK:
            System.out.println("탱크를 만듭니다.");
            break;
        case AIRPLANE:
            System.out.println("비행기를 만듭니다.");
            break;
        }
    }
} //컴파일 및 실행 과정에서 발견되지 않는 오류
```

4. 가변인자

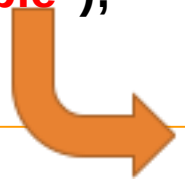
Ex03_Varargs

❖ 가변 인자 선언에 대한 컴파일러 처리

```
public static void showAll(String...vargs) {...}

public static void main(String[] args) {
   showAll("Box");
   showAll("Box", "Toy");
   showAll("Box", "Toy", "Apple");
}
```

vargs를 배열의 참조변수로 간주하고 코드를 작성하면 된다.



컴파일러가 다음과 같이 배열 기반 코드로 수정을 한다.

```
public static void showAll(String[] vargs) {...}

public static void main(String[] args) {
   showAll(new String[]{"Box"});
   showAll(new String[]{"Box", "Toy"});
   showAll(new String[]{"Box", "Toy", "Apple"});
}
```

실습 : 가변인자

```
public class Ex03_Varargs
{
    public static void helloEverybody(String... vargs)
    {
        for (String s : vargs)
            System.out.print(s + '\t');
        System.out.println();
    }

    public static void main(String[] args)
    {
        helloEverybody("홍길동");
        helloEverybody("홍길동", "전우치");
        helloEverybody("홍길동", "전우치", "손오공");
    }
}
```

5. 어노테이션

@Override	오버라이딩을 올바르게 했는지 컴파일러가 체크하게 한다. 오버라이딩할 때 메서드이름을 잘못적는 실수를 하는 경우가 많은데 이런 점을 방지!
@Deprecated	문제의 발생 소지가 있거나 개선된 기능의 다른 것으로 대체되어서 더 이상 필요 없게 되었음을 뜻함 따라서 아직은 호환성 유지를 위해서 존재하지만 이후에 사라질 수 있는 클래스 또는 메서드를 가리켜 Deprecated 되었다고 한다.
@SuppressWarnings	deprecation 관련 경고 메시지를 생략하라는 의미

실습 : @OVERRID,DEPRECATED, SUPPRESSWARNING

```
interface Unit4 {
    public void move(String str);
}

class Human4 implements Unit4 {
    @Override
    public void move(String str) {
        System.out.println(str);
    }
}

public class Ex04_Override {
    public static void main(String[] args){
        Unit4 unit = new Human4();
        unit.move("인간형 유닛이 이동합니다.");
    }
}
```

```
interface Unit5 {
    @Deprecated
    public void move(String str);
    public void run(String str);
}

class Human5 implements Unit5 {
    @Override
    public void move(String str) {
        System.out.println(str);
    }
    @Override
    public void run(String str) {
        System.out.println(str);
    }
}

public class Ex05_Deprecated {
    public static void main(String[] args)
    {
        Unit5 unit = new Human5();
        unit.move("인간형 유닛이 이동합니다.");
        unit.run("인간형 유닛이 달립니다.");
    }
}
```

```
interface Unit6 {
    @Deprecated
    public void move(String str);
    public void run(String str);
}

class Human6 implements Unit6 {
    @Override
    @SuppressWarnings("deprecation")
    public void move(String str)    {
        System.out.println(str);
    }
    @Override
    public void run(String str) {
        System.out.println(str);
    }
}

public class Ex06_SuppressWarnings
{
    @SuppressWarnings("deprecation")
    public static void main(String[] args)
    {
        Unit6 unit = new Human6();
        unit.move("인간형 유닛이 이동합니다.");
        unit.run("인간형 유닛이 달립니다.");
    }
}
```