

7장 중첩 클래스

중첩 클래스

중첩 클래스의 이해와 특징

- 중첩 클래스란? 특정 클래스 내에 또 다른 클래스가 정의되는 것을 의미한다. 이런 중첩 클래스가 필요한 이유는 지금까지 작업해 왔던 클래스들과는 다르게 독립적이지는 않지만 하나의 멤버처럼 사용할 수 있는 특징이 있다. 즉 다른 클래스와 협력할 일이 적고 외부 클래스와 밀접한 관련이 있을 때 사용

중첩 클래스를 정의 시 주의사항이자 장점

- 중첩 클래스는 외부 클래스의 모든 멤버들을 마치 자신의 멤버처럼 사용할 수 있다.
- 외부 클래스와 중첩 클래스가 서로의 멤버에 쉽게 접근할 수 있다.
- 소스의 가독성과 유지보수성을 높일 수 있다.
- 서로 관련 있는 클래스를 논리적으로 묶어서 표현함으로써, 코드의 캡슐화를 증가시킨다.
- 외부에서는 중첩 클래스에 접근할 수 없으므로, 코드의 복잡성을 줄일 수 있다.
- static 중첩 클래스는 제외하고는 다른 중첩 클래스는 항상 외부 클래스를 통해서 생성이 가능하다.



중첩 클래스는 외부 클래스를 상속할 필요 없이 외부 클래스의 private 멤버까지 사용할 수 있다.

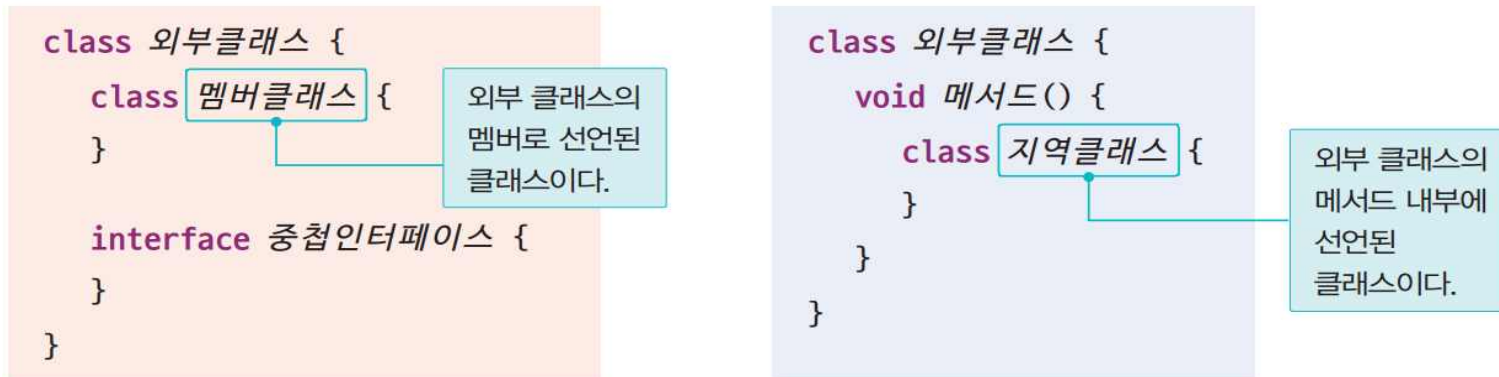
중첩 클래스의 종류

- 멤버 중첩클래스
- 지역 중첩 클래스
- 정적 중첩 클래스
- 익명 중첩 클래스

종류	설명
Member	멤버 변수나 멤버 메서드들과 같이 클래스가 정의된 경우에 사용한다.
Local	특정한 메서드 내에 클래스가 정의된 경우에 사용한다.
Static	static 변수(클래스 변수)와 같이 클래스가 static으로 선언된 경우에 사용한다.
Anonymous	참조할 수 있는 이름이 없는 경우에 사용한다.

중첩 클래스와 중첩 인터페이스

■ 중첩 클래스의 구조



■ 컴파일 후 생성 파일

```
외부클래스$멤버클래스.class  
외부클래스$중첩인터페이스.class  
외부클래스$1지역클래스.class
```

이름이 동일한 지역 클래스가 있다면 \$2 등을 사용한다.

■ 외부 클래스 접근

`외부클래스.this`

=> 중첩 클래스에서의 this는 중첩 클래스의 객체 자신을 참조하는 것

■ 중첩 클래스의 객체 생성

```
외부클래스.내부클래스 변수 = 외부클래스의객체변수.new 내부클래스생성자();  
외부클래스.정적멤버클래스 변수 = new 외부클래스.정적멤버클래스생성자();
```

중첩 클래스

● 내부[Member] 중첩 클래스

- 내부 클래스는 외부 클래스의 객체 내부에 존재하는 클래스를 말한다.
- 객체를 생성해야만 사용할 수 있는 멤버들과 같은 위치에 정의되는 클래스를 말한다. 즉 **중첩 클래스를 생성하려면 외부 클래스의 객체를 생성한 후에 생성할 수 있다.**

● 내부[Member] 중첩 클래스의 구성

```
class A {  
    ...  
    class B {  
        ...  
    }  
    ...  
}  
A a = new A();  
A.B b = a.new B();
```

외부클래스 외부클래스참조변수 = new 외부클래스();

외부클래스.내부클래스 내부클래스참조변수 = 외부클래스참조변수.new 내부클래스();

● Static 중첩 클래스

- static 중첩 클래스로 어쩔 수 없이 정의하는 경우가 있는데 그것은 바로 중첩 클래스 안에 static변수를 가지고 있다면 어쩔 수 없이 해당 중첩 클래스는 static으로 선언하여야 한다.
- static변수나 메서드들은 객체를 생성하지 않고도 접근이 가능하다는 의미이지만 **static 중첩 클래스는 외부 클래스를 생성하지 않고도**
[외부_클래스명.내부_클래스_생성자()]로 생성이 가능하다.

● Static 중첩 클래스의 구성

```
class A {  
    ...  
    static class B {  
        ...  
    }  
    ...  
}  
A.B b = new A.B();
```

외부클래스.내부클래스 클래스참조변수 = new 외부클래스.내부클래스();

중첩 클래스

Local 중첩 클래스

- Local 중첩 클래스는 **특정 메서드 안에서 정의되는 클래스**를 말한다. 다시 말해서 특정 메서드 안에서 선언되는 지역변수와 같은 것이다.
- 메서드가 호출될 때 생성할 수 있으며 메서드의 수행력이 끝나면 지역변수와 같이 자동 소멸된다.
- **지역 변수를 사용할 때는 반드시 해당 지역 변수가 final로 선언되어야 한다.** 만약 final로 선언되지 않은 지역 변수를 지역 클래스 내부에서 사용하게 되면 컴파일러가 강제로 해당 지역 변수에 final을 추가한다.

Local 중첩 클래스의 구성

```
class A {  
    ...  
    void abc() {  
        class B { //지역 클래스  
            ...  
        }  
        B b = new B();  
        //지역 클래스 객체 생성  
    }  
    ...  
}
```

지역 내부 클래스 지역 내부 클래스 참조 변수 = new 지역 내부 클래스();
⇒ 지역 클래스는 선언 이후 바로 객체를 생성해 사용하며, 메서드가 호출될 때만 메모리에 로딩된다.

Anonymous(익명) 중첩 클래스

- 익명이란? 이름이 없는 것을 의미한다. 익명 클래스(anonymous class)란 다른 내부 클래스와 달리, 이름을 가지지 않는 클래스를 의미한다.
- 클래스의 선언과 객체의 생성을 동시에 하므로 단 하나의 객체만을 생성할 수 있고, 단 한번만 사용되는 일회용 클래스이다.
- 생성자를 선언할 수도 없으며, 오로지 단 하나의 클래스나 단 하나의 인터페이스를 상속받거나 구현할 수 있다.

Anonymous(익명) 중첩 클래스의 구성

```
class A {  
    C c = new C() { /**/  
        public void bcd() {  
            System.out.println("익명 내부 클래스");  
        }  
    };  
    void abc() {  
        c.bcd();  
    }  
}  
interface C {  
    public abstract void bcd();  
}  
public class AnonymousClass_2 {  
    public static void main(String[] args) {  
        //객체 생성 및 메서드 호출  
        A a = new A();  
        a.abc();  
    }  
}
```

익명 클래스는 선언과 동시에 생성하여 참조변수에 대입함.
클래스 이름 참조 변수 이름 = new 클래스 이름() { //메서드 선언};

중첩 클래스와 중첩 인터페이스 => quiz_1_식별자

■ 멤버 클래스

=> 멤버 클래스는 외부 클래스의 멤버 필드나
멤버 메서드처럼 멤버로 선언된 클래스

=> 멤버 클래스 종류

- 정적 멤버 클래스(static 키워드 지정 O)
- 내부[멤버] 클래스(static 키워드 지정 X)

■ 외부 클래스 접근

외부클래스.this

=> 중첩 클래스에서의 this는 중첩 클래스의 객체 자신을 참조하는 것

=> 중첩 클래스에서 외부 클래스의 객체를 참조하려면 this 앞에 외부 클래스 이름을 명시 할 것

■ 중첩 클래스의 객체 생성

외부클래스.내부클래스 변수 = 외부클래스의객체변수.new 내부클래스생성자();

외부클래스.정적멤버클래스 변수 = new 외부클래스.정적멤버클래스생성자();

■ 예제

- 내부 클래스 사용 :

[sec05/MemberClassDemo](#)

//내부 클래스(인스턴스 멤버 클래스)

Package sec05;

Public class MemberClassDemo {

private String secret = " 비공개 ";

String s = " 외부 ";

class MemberClass {

String s = " 내부 ";

public void show1() {

System.out.println(" 내부 클래스 ");

System.out.println(secret);

System.out.println(s);

System.out.println(MemberClassDemo.this.s);

}

// static String s3 = " 정적 멤버 필드 " ;

// static void show2() {}

//내부 클래스 내부에 정적변수나 메서드를 포한 할 수 없다.

}

public static void main(String[] args) {

MemberClassDemo m = new MemberClassDemo();

MemberClassDemo.MemberClass m1 = m.new MemberClass();

/**중첩 클래스를 생성하려면 외부 클래스의 객체를 생성한 후에 생성

System.out.println(m1.s);

m1.show1();

}

}

내부
내부 클래스
비공개
내부
외부

중첩 클래스와 중첩 인터페이스 => quiz_2_식별자

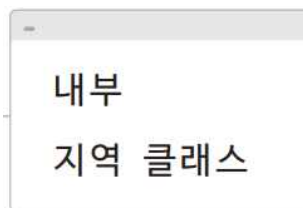
■ 지역 클래스

=> 외부 클래스의 메서드 내부에 선언된 클래스

⇒ 지역 클래스는 지역 변수처럼 메서드 내부에서만 사용하므로 `abstract`나 `final`로만 지정 할 수 있다.

■ 예제

- 지역 클래스와 지역 변수 관계 :
- [sec05/LocalClassDemo](#)



```
//지역 클래스
package sec05;
public class LocalClassDemo {
    private String s1 = "외부";

    void method() {
        int x = 1;
        //지역 클래스가 참조하는 지역변수는final로 간주
        class LocalClass {
            String s2 = "내부";
            String s3 = s1;

            public void show() {

                System.out.println("지역 클래스");
                // x = 2;
                //x는 method()지역변수이므로 final이기에 참조가능하나
                //x는 final이여서 변경 불가
            }
        }

        LocalClass lc = new LocalClass();
        System.out.println(lc.s2);
        lc.show();
    }

    public static void main(String[] args) {
        LocalClassDemo lcd = new LocalClassDemo();
        lcd.method();
    }
}
```

중첩 클래스와 중첩 인터페이스 => quiz_3_식별자

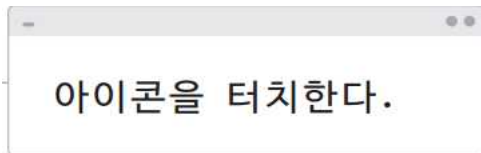
■ 중첩 인터페이스

=> 클래스의 멤버로 선언된 인터페이스

=> 그래픽 프로그래밍에서 이벤트를 처리할 때 자주 활용

■ 예제

- 중첩 인터페이스 사용 :
- [sec05/InnerInterfaceDemo](#)



```
//중첩 인터페이스
package sec05;

class Icon { //클래스의 멤버로 선언된 중첩 인터페이스
    interface Touchable {
        void touch();
    }
}

public class InnerInterfaceDemo implements Icon.Touchable {
    // Icon.Touchable 는 Icon 클래스의 멤버 Touchable 인터페이스 의미

    public void touch() { //메서드 오버라이딩
        System.out.println("아이콘을 터치한다.");
    }

    public static void main(String[] args) {
        Icon.Touchable btn = new InnerInterfaceDemo();
        // 부모                자식
        btn.touch();
    }
}
```


익명 클래스

■ 개념

- 익명 클래스는 **한번만 사용되는 클래스로** 이름이 없는 클래스이다.
- 익명 클래스는 **클래스 선언과 객체 생성 실행문을 하나로 합친 것**
- 익명 클래스는 단독으로 정의 할 수 없고 클래스를 상속하거나 인터페이스를 구현해서 작성
- 중첩 클래스의 특수한 형태로 코드가 단순해지기 때문에 이벤트 처리나 스레드 등에서 자주 사용

```
class OnlyOnce [ extends  
                implements ] Parent {  
    // Parent가 클래스라면 오버라이딩한 메서드  
    // Parent가 인터페이스라면 구현한 메서드  
}  
  
Parent p = new OnlyOnce();
```



OnlyOnce가 한번만 사용 된다면 생성자의 이름을 부모 클래스 이름으로 대신하고 본체를 구현하는 익명 클래스 사용

```
Parent p = new Parent() {  
    // Parent가 클래스라면 오버라이딩한 메서드  
    // Parent가 인터페이스라면 구현한 메서드  
};
```

무명 클래스 본체로서 OnlyOnce 클래스의 본체와 동일하다.

하나의 실행문이므로 세미콜론(;)으로 끝난다.

익명 클래스 => quiz_4_식별자

■ 활용

- 익명 클래스의 부모로 사용할 클래스 : [sec06/Bird](#)
- 기명 멤버 클래스 : [sec06/MemberDemo](#)
- 익명 멤버 클래스 : [sec06/Anonymous1Demo](#)
- 기명 지역 클래스 : [sec06/LocalDemo](#)
- 익명 지역 클래스 : [sec06/Anonymous2Demo](#)

```
package sec06;
public class Bird { //부모로 사용할 클래스
    void move() {
        System.out.println("새가 움직인다~~~.");
    }
}
// public interface Bird {
// 인터페이스로 바꾸어도 Bird클래스와 결과 유사
// void move();
// }
```

//기명 멤버 중첩 클래스

```
package sec06;
public class MemberDemo {

    class Eagle extends Bird { //멤버 중첩 클래스
        public void move() { //메서드 오버라이딩
            System.out.println("독수리가 난다~~~.");
        }
        public void sound() { //Eagle자식클래스에서추가
            System.out.println("휘익~~~.");
        }
    }

    Eagle e = new Eagle(); // MemberDemo의 멤버

    public static void main(String[] args) {
        MemberDemo m = new MemberDemo(); /**
        m.e.move();
        m.e.sound();
    }
}
```

독수리가 난다~~~
휘익~~~

//익명 멤버 중첩 클래스

```
package sec06;
public class Anonymous1Demo {
    Bird e = new Bird() {
        //생성자의 이름을 부모 클래스 이름으로 대신하고 본체를 구현하는 익명
        //클래스

        public void move() { //메서드 오버라이딩
            System.out.println("독수리가 난다~~~.");
        }
        void sound() { //익명 자식 클래스에서 추가
            System.out.println("휘익~~~.");
        }
    };

    public static void main(String[] args) {
        Anonymous1Demo a = new Anonymous1Demo();
        a.e.move();
        // a.e.sound();
        //a.e가 부모 클래스이므로 자식 클래스의 sound()를 호출할 수 없다.
    }
}
```

독수리가 난다~~~

익명 클래스

=> quiz_5_식별자

■ 활용

- 익명 클래스의 부모로 사용할 클래스 : [sec06/Bird](#)
- 기명 멤버 클래스 : [sec06/MemberDemo](#)
- 익명 멤버 클래스 : [sec06/Anonymous1Demo](#)
- 기명 지역 클래스 : [sec06/LocalDemo](#)
- 익명 지역 클래스 : [sec06/Anonymous2Demo](#)

```
package sec06;
public class Bird {
    void move() {
        System.out.println("새가 움직인다~~~.");
    }
}
// public interface Bird {
// 인터페이스로 바꾸어도 Bird클래스와 결과 유사
// void move();
// }
```

//기명 지역 중첩 클래스

```
package sec06;
```

```
public class LocalDemo {
    public static void main(String[] args) {
        class Eagle extends Bird {
            //main()메서드내에 정의된 기명 지역 클래스

            public void move() { //메서드 오버라이딩
                System.out.println("독수리가 난다~~~.");
            }
        }
        Bird e = new Eagle(); //main()메서드의 지역변수
        e.move();
    }
}
```

독수리가 난다~~~

//익명 지역 중첩 클래스

```
package sec06;
```

```
public class Anonymous2Demo {
    public static void main(String[] args) {
        Bird b = new Bird() { // b는 지역변수
            public void move() {

                System.out.println("독수리가 난다~~~.");
            }
        };
        b.move();
    }
}
```

독수리가 난다~~~

익명

10장 람다식과 함수인터페이스

목차

- 람다식 기초
- 란다식 유의 사항과 활용
- 함수형 인터페이스 활용

람다식 기초

■ 람다식 의미(자바 8부터는 함수형 프로그래밍 기법인 람다식을 지원)

- 함수를 분명하고 간결한 방법으로 설명하기 위해 고안한 것으로 나중에 실행할 목적으로 다른 곳에 전달 할 수 있는 코드
- 메서드를 하나의 식(expression)으로 표현한것
- 메서드를 람다식으로 표현하면 메서드의 이름이 없어지므로, 람다식을 '익명함수(anonymous function)'라고 함 (메서드를 포함하는 익명 구현 객체를 전달할 수 있는 코드)
- 반환 타입이 없음
- 함수를 변수처럼 사용할 수 있는 개념

CF) 함수 : 특정한 작업을 수행하는 독립적인 부분

메서드 : 객체 지향의 개념에서는 함수 대신 객체의 행위나 동작을 의미하는 메서드

=> 메서드는 함수와 같은 의미지만 JDK8 이전의 자바에서는 객체(혹은 클래스)를 통해서만 접근이 가능하고, 메소드 그 자체를 변수로 사용하지는 못하였다.
=> 람다식은 JDK8에서 지원하는 것으로 함수를 변수처럼 사용할 수 있기 때문에, 매개변수로 다른 메소드의 인자로 전달할 수 있고, 리턴값으로 함수를 받을 수도 있다.

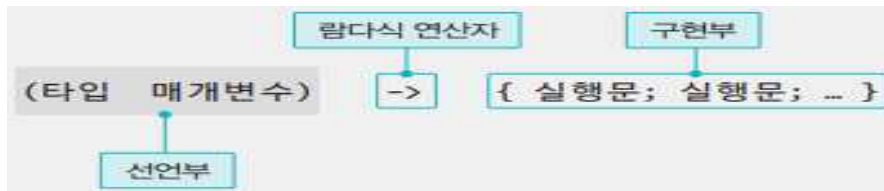
● 특징

- 메서드와 달리 이름이 없다.
- 메서드와 달리 특정 클래스에 종속되지 않지만, 매개변수, 반환 타입, 본체를 가지며, 심지어 예외도 처리할 수 있다.
- 메서드의 인수로 전달될 수도 있고 변수에 대입될 수 있다.
- 익명 구현 객체와 달리 메서드의 핵심 부분만 포함한다

람다식 기초

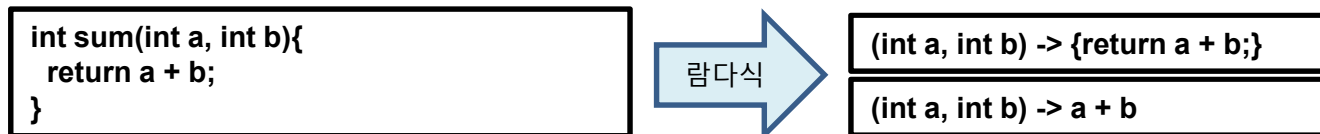
■ 람다식 문법

- 메서드를 하나의 식(expression)으로 표현한것
- 메서드를 람다식으로 표현하면 메서드의 이름이 없게되므로, 람다식을 '익명함수(anonymous function)'라고 함 (메서드를 포함하는 익명 구현 객체를 전달할 수 있는 코드)



■ 람다식 작성

- 람다식에 선언된 매개변수의 타입은 추론이 가능한 경우는 생략할 수 있음, 반환타입이 없는 이유도 항상 추론이 가능하기 때문
- 익명함수답게 메서드에서 이름과 반환타입을 제거, 선언부와 몸통사이에 `->`를 추가



- 반환값이 있는 메서드의 경우, **return문 대신 식으로 대신할 수 있음**, 식의 연산결과가 자동적으로 반환값이 됨. 이때 문장이 아닌 식이므로 끝에 `;`를 붙이지 않음
- 선언된 매개변수가 하나일 경우 괄호 생략가능, 단 매개변수의 타입을 선언했다면 괄호를 생략할 수 없음

`a -> a * a`

`(int a) -> a * a`

- 괄호 안의 문장이 하나일 때는 괄호를 생략할 수 있음, 문장의 끝에 `;`를 붙이지 않으며 return 문일 경우 괄호는 생략할 수 없음

`a, b -> a + b`

`a, b -> {return a + b;}`

함수 인터페이스

■ 함수인터페이스 의미

- 람다식은 **interface**의 **익명 구현 객체(익명 클래스)**를 **생성하기 때문에** interface와 밀접한 관련이 있다.

=> interface는 직접 객체화할 수 없기 때문에 상속을 통해 구현 클래스가 필요하지만 (**교재 7장 인터페이스 참고**) 람다식은 **사용 대상 interface의 익명 구현 클래스를 생성하고 객체화한다.**

- 오직 1개의 추상 메서드로 구성된 인터페이스 구현 객체만 람다식으로 표현 할 수 있는데 이 인터페이스를 **함수형 인터페이스(functional interface)**라고 한다.

= 하나의 추상메서드만이 선언된 interface만이 람다식의 **타겟타입**이 될 수 있는데, 이런 인터페이스를 **함수형 인터페이스(functional interface)**라고 한다.

=> 람다식이 대입될 interface를 람다식의 '타겟타입(target type)' 이라고 한다

=> 람다식은 하나의 메서드만 정의 할 수 있기 때문에 모든 인터페이스의 구현 객체를 람다식으로 표현할 수 없다.

// 타겟타입(인터페이스)은 함수형 인터페이스(무조건 하나의 메서드만 가짐)여야 한다.

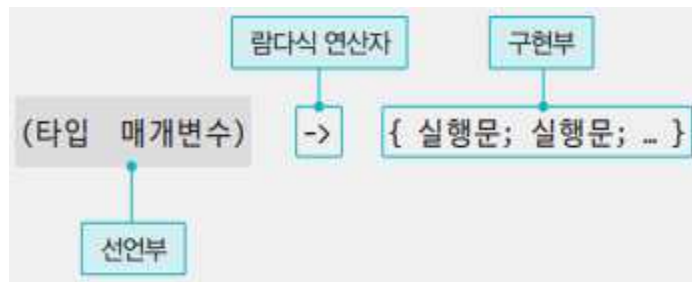
인터페이스 변수이름 = 람다식;

=> 람다식에 대입되는 인터페이스는 하나의 메서드만 가져야 하는데, 이를 명시하기 위한 annotation이 바로 @FunctionalInterface이다. 인터페이스의 선언부에 이 annotation을 붙여놓으면 컴파일러가 이 인터페이스에 두 개 이상의 추상메서드가 선언되지 않도록 검사한다.

함수 인터페이스

■ 람다식 의미와 문법

- 메서드를 포함하는 익명 구현 객체를 전달할 수 있는 코드
- 문법



- 람다식에 선언된 매개변수의 타입은 추론이 가능한 경우는 생략할 수 있음, 반환타입이 없는 이유도 항상 추론이 가능하기 때문

■ 함수인터페이스문법

// 타겟타입(인터페이스)은
//함수형 인터페이스(무조건 하나의 메서드만 가짐)여야 한다.

인터페이스 변수이름 = 람다식;

```
@FunctionalInterface
interface Flyer{
    public void fly();
}
```

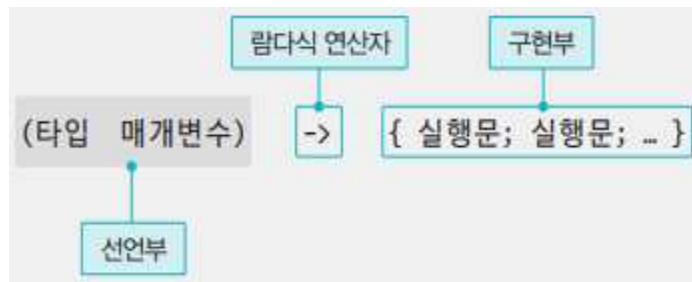
```
//1. 일반
Flyer f = new Flyer() {
    @Override
    public void fly() {
        System.out.println("Fly");
    }
};
f.fly();

//2. 람다
Flyer f2 = () -> { System.out.println("Fly2");};
f2.fly();
//실행문이 하나면 {} 생략 가능.
Flyer f3 = () -> System.out.println("Fly3");
f3.fly();
```

함수 인터페이스

■ 람다식 의미와 문법

- 메서드를 포함하는 익명 구현 객체를 전달할 수 있는 코드
- 문법



- 람다식에 선언된 매개변수의 타입은 추론이 가능한 경우는 생략할 수 있음, 반환타입이 없는 이유도 항상 추론이 가능하기 때문

■ 함수인터페이스문법

// 타겟타입(인터페이스)은
//함수형 인터페이스(무조건 하나의 메서드만 가짐)여야 한다.

인터페이스 변수이름 = 람다식;

- 예제 : [sec01/Lambda2Demo](#)

```
package sec01;
interface Negative { //함수형 인터페이스, 메서드 1개
    int neg(int x);
}
interface Printable { //함수형 인터페이스, 메서드 1개
    void print();
}

public class Lambda2Demo {
    public static void main(String[] args) {
        //모두 같은 의미
        Negative n;
        n = (int x) -> { return -x; };
        n = (x) -> { return -x; };
        n = x -> { return -x; };
        n = (int x) -> -x;
        n = (x) -> -x;
        n = x -> -x;

        Printable p;
        p = () -> { System.out.println("안녕!"); };
        p = () -> System.out.println("안녕!");
        // 매개변수가 없으므로 선언부의 괄호를 사용해야함
        p.print();
    }
}
```

람다식과 함수 인터페이스 => quiz_6_식별자

다음 프로그램들을 결과창이 나오도록 작성하시오.

[quiz_6_1_식별자] 람다식과 함수인터페이스 이용한 프로그램을 빈 곳을 채워 결과창과 같이 나오도록 하시오.

[quiz_6_2_식별자] **익명 지역 클래스**로 정의하여 같은 결과가 나오도록 프로그램을 작성하시오.

[quiz_6_3_식별자] **인터페이스의 상속을 받은 클래스**로 정의하여 같은 결과가 나오도록 작성하시오.

<pre>//람다식과 함수 인터페이스 이용 package ch10_111; interface Negative { //함수형 인터페이스 int neg(int x); } interface Printable { //함수형 인터페이스 void print(); } public class quiz_6_1_111 { public static void main(String[] args) { //모두 같은 의미 Negative n; n = (int x) -> { return -x; }; ???/**** Printable p; p = () -> System.out.println("안녕!"); p.print(); } }</pre>	<pre>package ch10_111; interface Negative { int neg(int x); } interface Printable { void print(); } public class quiz_6_2_111 { public static void main(String[] args) { // 여러 문장 } }</pre>	<pre>package ch10_111; interface Negative { int neg(int x); } interface Printable { void print(); } public class quiz_6_3_111 { public static void main(String[] args) { // 여러 문장 } }</pre>
---	---	---

람다식 기초

■ 메서드 참조

- 전달할 동작을 수행하는 메서드가 이미 정의된 경우에 표현할 수 있는 람다식의 축약형
- 메서드 참조의 종류와 표현 방식

종류	표현 방식
정적 메서드 참조	클래스이름::정적메서드
인스턴스 메서드 참조	객체이름::인스턴스메서드(혹은 클래스이름::인스턴스메서드)
생성자 참조	클래스이름::new 혹은 배열타입이름::new

- 예제 : [sec01/MethodRefDemo](#)

```
package sec01;
interface Mathematical {
    double calculate(double d); }
interface Pickable {
    char pick(String s, int i); }
interface Computable {
    int compute(int x, int y); }
class Utils {
    int add(int a, int b) {
        return a + b; } }
//interface Applyable {
//    boolean apply(Box a, Box b);}
//class Box {
//    int no;
//    public Box(int no) {
//        this.no = no; }
//    boolean isSame(Box b) {
//        return this.no == b.no; } }
```

```
public class MethodRefDemo {
    public static void main(String[] args) {
        Mathematical m;
        Pickable p;
        Computable c;
        // m = d -> Math.abs(d);
        m = Math::abs;
        System.out.println(m.calculate(-50.3));
        // p = (a, b) -> a.charAt(b);
        p = String::charAt;
        System.out.println(p.pick("안녕, 인스턴스 메서드 참조!", 4));
        Utils utils = new Utils();
        // c = (a, b) -> utils.add(a, b);
        c = utils::add;
        System.out.println(c.compute(20, 30));
        // Applyable app;
        /// app = (a, b) -> a.isSame(b);
        // app = Box::isSame;
        // System.out.println(app.apply(new Box(1), new Box(2)));
        // System.out.println(app.apply(new Box(1), new Box(1)));
    }
}
```

람다식 기초

■ 메서드 참조

- 전달할 동작을 수행하는 메서드가 이미 정의된 경우에 표현할 수 있는 람다식의 축약형
- 메서드 참조의 종류와 표현 방식

종류	표현 방식
정적 메서드 참조	클래스이름::정적메서드
인스턴스 메서드 참조	객체이름::인스턴스메서드(혹은 클래스이름::인스턴스메서드)
생성자 참조	클래스이름::new 혹은 배열타입이름::new

- 예제 : [sec01/ConstructorRefDemo](#)

```
package sec01;
interface NewObject<T> {
    T getObject(T o);
}

interface NewArray<T> {
    T[] getArray(int size);
}

public class ConstructorRefDemo {
    public static void main(String[] args) {
        NewObject<String> s;
        NewArray<Integer> i;

        //    s = x -> new String(x);

        s = String::new;
        String str = s.getObject("사과");

        //    i = x -> new Integer[x];

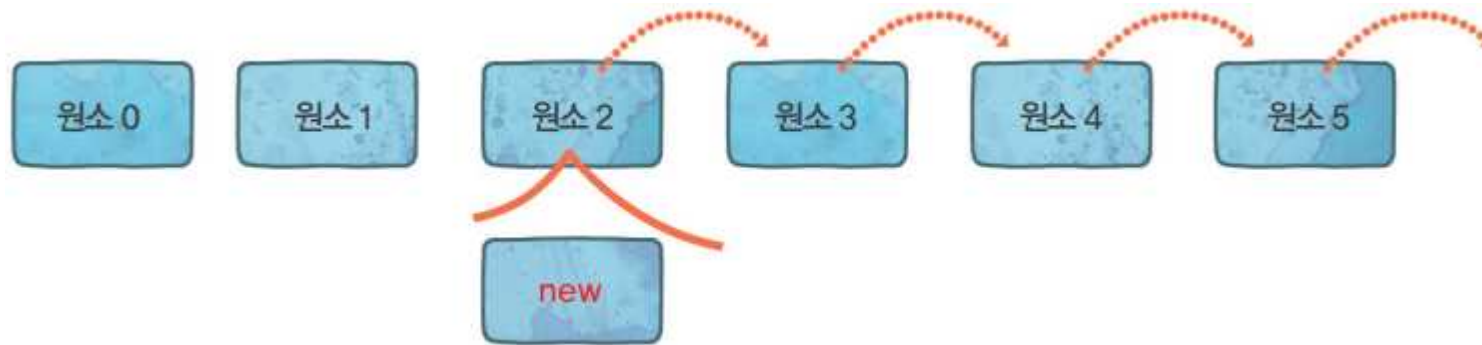
        i = Integer[]::new;
        Integer[] array = i.getArray(2);
        array[0] = 10;
        array[1] = 20;    }
}
```

11장 컬렉션 프레임워크

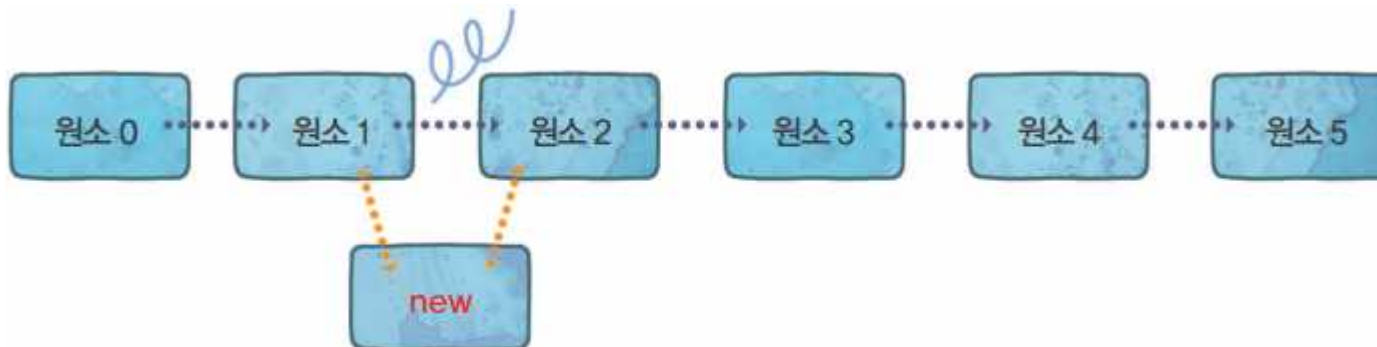
컬렉션 프레임워크 기초

■ 필요성

- 유사한 객체를 여러 개 저장하고 조작해야 할 때가 빈번
- 고정된 크기의 배열의 불편함



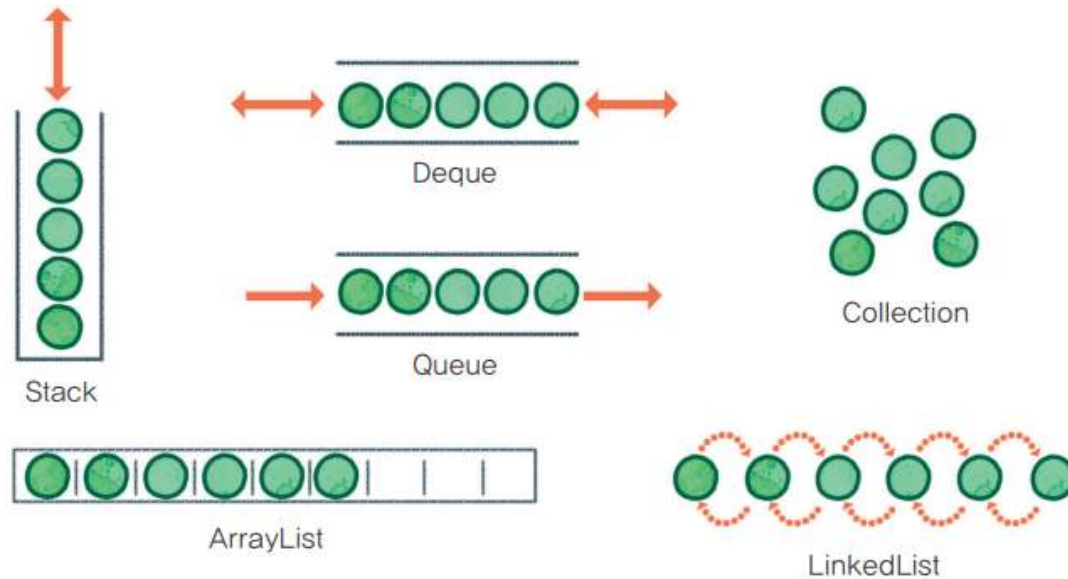
- 연결 리스트를 사용하면 배열보다 쉽게 추가된다.



컬렉션 프레임워크 기초

■ 의미

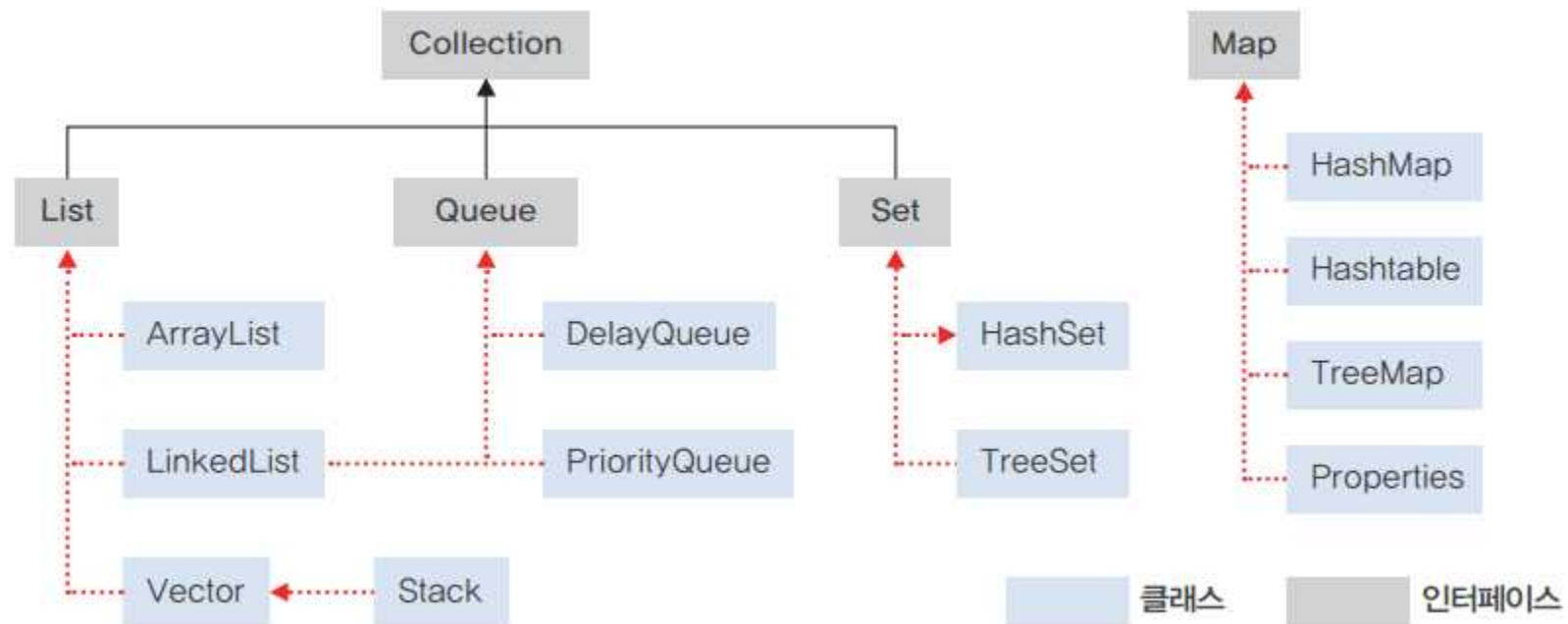
- 유사한 객체의 집단을 효율적으로 관리할 수 있도록 컬렉션 프레임워크를 제공
- 컬렉션 : 데이터를 한곳에 모아 편리하게 저장 및 관리하는 가변 크기의 객체 컨테이너
- 컬렉션 프레임워크 : 객체를 한곳에 모아 효율적으로 관리하고 편리하게 사용할 수 있도록 제공하는 환경



컬렉션 프레임워크 기초

■ 구조

- 컬렉션 프레임워크는 인터페이스와 클래스로 구성
- 인터페이스는 컬렉션에서 수행할 수 있는 각종 연산을 제네릭 타입으로 정의해 유사한 클래스에 일관성 있게 접근하게 함
- 클래스는 컬렉션 프레임워크 인터페이스를 구현한 클래스
- java.util 패키지에 포함 (DelayQueue는 java.util.concurrent 패키지에 포함)



Collection 인터페이스

■ Collection 인터페이스와 구현 클래스

인터페이스		특징	구현 클래스
Collection	List	객체의 순서가 있고, 원소가 중복될 수 있다.	ArrayList, Stack, Vector, LinkedList
	Queue	객체를 입력한 순서대로 저장하며, 원소가 중복될 수 있다.	DelayQueue, PriorityQueue, LinkedList
	Set	객체의 순서가 없으며, 동일한 원소를 중복할 수 없다.	HashSet, TreeSet, EnumSet

Collection 인터페이스

■ Collection 인터페이스

● 주요 메서드

메서드	설명
<code>boolean add(E e)</code>	객체를 맨 끝에 추가한다.
<code>void clear()</code>	저장된 모든 객체를 제거한다.
<code>boolean contains(Object o)</code>	명시한 객체의 저장 여부를 조사한다.
<code>boolean isEmpty()</code>	리스트가 비어 있는지 조사한다.
<code>Iterator<E> iterator()</code>	Iterator를 반환한다.
<code>boolean remove(Object o)</code>	명시한 첫 번째 객체를 제거하고, 제거 여부를 반환한다.
<code>int size()</code>	저장된 전체 객체의 개수를 반환한다.
<code>T[] toArray(T[] a)</code>	리스트를 배열로 반환한다.

● 이외에도 Collection 인터페이스는 다음과 같은 유용한 디폴트 메서드를 제공

```
default void forEach(Consumer<? super T> action)
default boolean removeIf(Predicate <? super E> filter)
default <T> T[] toArray(IntFunction<T[]> generator)
```

→ 자바 11부터

Collection 인터페이스

■ 컬렉션의 반복 처리

- Collection 인터페이스는 iterator() 메서드를 통하여 반복자를 제공
- 키-값 구조의 Map 컬렉션은 반복자를 제공하지 않는다.
- Iterator 인터페이스가 제공하는 주요 메서드

메서드	설명
boolean hasNext()	다음 원소의 존재 여부를 반환한다.
E next()	다음 원소를 반환한다.
default void remove()	마지막에 순회한 컬렉션의 원소를 삭제한다.

- 예제 : [sec02/IteratorDemo](#)



```
package sec02;

import java.util.*;

public class IteratorDemo {
    public static void main(String[] args) {
        Collection<String> list = Arrays.asList("다람쥐", "개구리", "나비");

        Iterator<String> iterator = list.iterator();
        while (iterator.hasNext())
            System.out.print(iterator.next() + "-");
        System.out.println();

        while (iterator.hasNext())
            System.out.print(iterator.next() + "+");
        System.out.println();

        iterator = list.iterator();
        while (iterator.hasNext())
            System.out.print(iterator.next() + "=");
    }
}
```

Collection 인터페이스

■ List 컬렉션

- 순서가 있는 객체를 중복 여부와 상관없이 저장하는 리스트 자료구조를 지원. 배열과 매우 유사하지만 크기가 가변적. 원소의 순서가 있으므로 원소를 저장하거나 읽어올 때 인덱스를 사용

메서드	설명
void add(int index, E element)	객체를 인덱스 위치에 추가한다.
E get(int index)	인덱스에 있는 객체를 반환한다.
int indexOf(Object o)	명시한 객체가 있는 첫 번째 인덱스를 반환한다.
E remove(int index)	인덱스에 있는 객체를 제거한다.
E set(int index, E element)	인덱스에 있는 객체와 주어진 객체를 교체한다.
List<E> subList(int from, int to)	범위에 해당하는 객체를 리스트로 반환한다.

- 디폴트 메서드

```
default void replaceAll(UnaryOperator<E> operator)
default void sort(Comparator<? super E> c)
```

- 팩토리 메서드(자바 9부터)

```
static <E> List<E> of(E... elements)
```

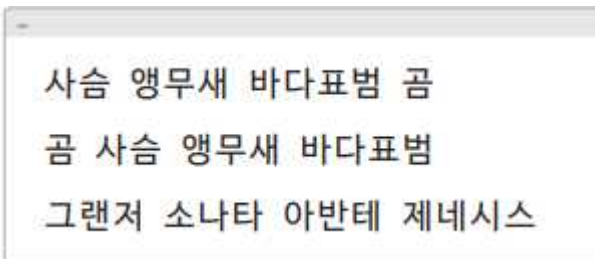
Collection 인터페이스

■ List 컬렉션

- List 타입과 배열 사이에는 다음 메서드를 사용하여 상호 변환

```
public static <T> List<T> asList(T... a) // java.util.Arrays 클래스의 정적 메서드
<T> T[] toArray(T[] a) // java.util.List 클래스의 메서드
```

- 예제 : [sec02/ListDemo](#)



사슴 앵무새 바다표범 곰
곰 사슴 앵무새 바다표범
그랜저 소나타 아반테 제네시스

- 대표적인 List 구현 클래스 : ArrayList, Vector

```
package sec02;

import java.util.Arrays;
import java.util.List;

public class ListDemo {
    public static void main(String[] args) {
        String[] animals1 = { "사슴", "호랑이", "바다표범", "곰" };

        List<String> animals2 = Arrays.asList(animals1);
        animals2.set(1, "앵무새");
        // animals2.add("늑대");

        for (String s : animals2)
            System.out.print(s + " ");
        System.out.println();

        animals2.sort((x, y) -> x.length() - y.length());
        String[] animals3 = animals2.toArray(new String[0]);
        for (int i = 0; i < animals3.length; i++)
            System.out.print(animals3[i] + " ");
        System.out.println();

        List<String> car = List.of("그랜저", "소나타", "아반테", "제네시스");
        // car.set(1, "싼타페");
        car.forEach(s -> System.out.print(s + " "));

        // List<Object> objects = List.of("a", null);
    }
}
```