람다 vs 익명 클래스

소유자	종수 김
태그	

람다 vs 익명 클래스 - 1

자바에서 익명 클래스와 람다 표현식은 모두 간단하게 기능을 구현하거나, 일회성으로 사용할 객체를 만들 때 유용하나 방식과 의도에 차이가 있음.

1. 문법 차이
    - 익명 클래스

```
// 익명 클래스 사용 예
Button button = new Button(); button.setOnClickListener(new OnClickL
        @Override
        public void onClick(View v) {
System.out.println("버튼 클릭"); }
});
```

        - 익명 클래스는 클래스를 선언하고 즉시 인스턴스를 생성하는 방식.
        - 반드시 new 인터페이스명 () {...} 형태로 작성하며 메서드를 오버라이드 해서 구현
        - 익명 클래스도 하나의 클래스
    - 람다 표현식

```
// 람다 표현식 사용 예
Button button = new Button();
button.setOnClickListener(v -> System.out.println("버튼 클릭"));
```

        - 람다 표현식은 함수를 간결하게 표현할 수 있는 방식
        - 함수형 인터페이스(메서드가 하나인 인터페이스)를 간단히 구현할 때 주로 사용

    - 람다는 → 연산자를 사용하여 표현하며, 매개변수와 실행할 내용을 간결하게 작성

2. 코드의 간결함
    - ***익명 클래스***는 문법적으로 더 복잡하고 장황하다. `new 인터페이스명()` 같은 형태와 함께 메서드를 오버라이드 해야 하므로 코드의 양이 상대적으로 많다.
    - ****람다 표현식****은 간결하며, 불필요한 코드를 최소화한다. 또한 많은 생략 기능을 지원해서 핵심 코드만작성할 수 있다.
3. 상속 관계
    - ***익명 클래스***는 일반적인 클래스처럼 다양한 인터페이스와 클래스를 구현하거나 상속할 수 있다. 즉, 여러 메서드를 가진 인터페이스를 구현할 때도 사용할 수 있다.
    - ****람다 표현식****은 메서드를 딱 하나만 가지는 ****함수형 인터페이스****만을 구현할 수 있다.
        - 람다 표현식은 ****클래스를 상속****할 수 없다. 오직 함수형 인터페이스만 구현할 수 있으며, ****상태(필드, 멤버 변수)****나 추가적인 메서드 오버라이딩은 불가능하다.
        - 람다는 단순히 함수를 정의하는 것으로, 상태나 추가적인 상속 관계를 필요로 하지 않는 상황에서만 사용할수 있다.
4. 호환성
    - 익명 클래스는 자바의 오래된 버전에서도 사용할 수 있음.
    - 람다 표현식은 자바 8부터 도입되었기에 그 전 버전은 사용 불가능
5. this 키워드의 의미
    - 익명 클래스 : 내부에서 this는 익명클래스 자신을 가리킨다. 외부 클래스와 별도의 컨텍스트
    - 람다 표현식 : this는 람다를 선언한 클래스의 인스턴스, 즉 락다 표현식은 별도의 컨텍스트를 가지는 것이 아닌 람다를 선언한 클래스의 컨텍스트를 유지한다.
        - 람다 내부의 this는 람다가 선언된 외부 클래스의 this와 동일
    - 예제

```
package lambda.lambda6;

public class OuterMain {
    private String message = "외부 클래스";
```

```
    public void execute() {
        // 1. 익명 클래스 예시
        Runnable anonymousRunnable = new Runnable() {
            private String message = "익명 클래스";
            @Override
            public void run() {
                /// 익명 클래스에서의 this는 익명 클래스의 instance
                System.out.println("[익명 클래스] this :  " + this);
                System.out.println("[익명 클래스] this.class :  " + this.getClass
                System.out.println("[익명 클래스] this.message :  " + this.mess
            }
        };

        // 2. 람다 예시
        Runnable lambda = () -> {
            // 람다에서의 this는 람다가 선언된 클래스의 인스턴스(즉, 외부 클래스)
            System.out.println("[람다] this : " + this);
            System.out.println("[람다] this.class : " + this.getClass());
            System.out.println("[람다] this.message : " + this.message);
        };

        anonymousRunnable.run();
        System.out.println("------------------------_");
        lambda.run();

    }

    public static void main(String[] args) {
        OuterMain outerMain = new OuterMain();
        System.out.println("[외부 클래스]: " + outerMain);
        outerMain.execute();
    }
}

// 결과
[외부 클래스]: lambda.lambda6.OuterMain@5acf9800
[익명 클래스] this :  lambda.lambda6.OuterMain$1@7a81197d
```

```
[익명 클래스] this.class :  class lambda.lambda6.OuterMain$1
[익명 클래스] this.message :  익명 클래스
-------------------------_
[람다] this : lambda.lambda6.OuterMain@5acf9800
[람다] this.class : class lambda.lambda6.OuterMain
[람다] this.message : 외부 클래스
```

    - 람다에서 선언한 this는 외부 클래스를 가리킨다.
    - 새로운 클래스가 아니라는 것.

6. 캡처링(capturing)
    - 익명 클래스 : 익명 클래스는 외부 변수에 접근할 수 있지만, 지역 변수는 반드시 final 혹은 사실상 final인 변수만 캡처 가능
    - 람다 표현식 : 람다도 익명 클래스와 같이 캡처링을 지원, 지역 변수는 반드시 final 혹은 사실상 final인 변수만 캡처 가능.
7. 생성 방식
    - 자바 내부 동작 방식으로 크게 중요하지 않음. 참고
    - 익명 클래스 : 새로운 클래스를 정의하여 객체를 생성하는 방식, 컴파일 시 새로운 내부 클래스로 변환.
      진짜 클래스가 만들어지는 것, 클래스가 메모리 상에서 별도로 관리되므로, 메모리 상에 약간의 추가 오버헤드 발생
    - 람다 : 내부적으로 invokeDynamic 이라는 메커니즘을 사용하여 컴파일 타임에 실제 클래스 파일을 생성하지 않고, 런타임 시점에 동적으로 필요한 코드를 처리
        - 익명 클래스보다 메모리 관리가 더 효율적이며, 생성된 클래스 파일이 없으므로 클래스 파일 관리의 복잡성이 감소.

쉽게 정리하면 다음과 같다.

- **익명 클래스**
    - 컴파일 시 실제로 `OuterClass$1.class` 와 같은 클래스 파일이 생성된다.
    - 일반적인 클래스와 같은 방식으로 작동한다.
    - 해당 클래스 파일을 JVM에 불러서 사용하는 과정이 필요하다.
- **람다**
    - 컴파일 시점에 별도의 클래스 파일이 생성되지 않는다.
    - 자바를 실행하는 실행 시점에 동적으로 필요한 코드를 처리한다.

▼ 예시

대략적으로 설명하면 다음과 같이 작동한다.
참고로 이 부분은 자바 스펙에 명시된 것이 아니기 때문에 자바 버전과 자바 구현 방식에 따라 내용이 달라질 수 있다.
따라서 대략 이런 방식으로 작동하는구나라고 참고만 해두자.

원본 코드

```
public class FunctionMain {
    public static void main(String[] args) {
        Function<String, Integer> function = x -> x.length();
        System.out.println("function1 = " + function.apply("hello"));
```

```
    }
}
```

- 람다가 포함된 코드가 있다면 자바는 다음과 같이 컴파일 한다.

컴파일 코드

```
public class FunctionMain {
    public static void main(String[] args) {
        Function<String, Integer> function = 람다 인스턴스 생성(구현 코드는 lambda1()
연결)
        System.out.println("function1 = " + function.apply("hello"));
    }

    // 람다를 private 메서드로 추가
    private Integer lambda1(String x) {
        return x.length();
    }
}
```

- 컴파일 단계에서 람다를 별도의 클래스로 만드는 것이 아니라, `private` 메서드로 만들어 숨겨둔다.
  - 참고로 자바 내부에서 일어나는 일이므로 개발자가 이렇게 만들어진 코드를 확인하기는 어렵다.
- 그리고 실행 시점에 동적으로 람다 인스턴스를 생성하고, 해당 인스턴스의 구현 코드로 앞서 만든 `lambda1()` 메서드가 호출되도록 연결한다.

8. 상태 관리

- 익명 클래스 : 인스턴스 내부에 상태(필드, 멤버 변수)를 가질 수 있음.
  Ex) 익명 클래스 내부에 멤버 변수를 선언하고 해당 변수의 값을 변경하거나 상태를 관리할 수 있음.

    - 상태를 필요로 하는 경우 익명 클래스
- 람다 : 클래스는 그 내부에 상태(필드, 멤버 변수)와 기능(메서드)를 가진다. 반면에 함수는 그 내부에 상태(필드)를 가지지 않고 기능만 제공.
    - 함수인 람다는 기본적으로 필드(멤버 변수가)없으므로 스스로 상태를 유지 X

9. 용도 구분

- 익명 클래스
    - 상태를 유지하거나, 다중 메서드를 구현할 때
    - 기존 클래스 또는 인터페이스를 상속하거나 구현할 때
    - 복잡한 인터페이스 구현이 필요할 때
- 람다
    - 상태를 유지할 필요가 없고, 간결함이 중요할 때
    - 단일 메서드만 필요한 간단한 함수형 인터페이스 구현 시
    - 간결한 코드가 필요한 경우.

정리, 요약

- 대부분의 경우 익명 클래스를 람다로 대체할 수 있다. 하지만 여러 메서드를 가진 인터페이스나 클래스의 경우에는 여전히 익명 클래스가 필요할 수 있다.
- 자바 8 이후에는 익명 클래스, 람다 둘 다 선택할 수 있는 경우라면 익명 클래스 보다는 람다를 선택하는 것이 간결한 코드, 가독성 관점에서 대부분 더 나은 선택이다.

정리

정리

1. **문법 및 가독성**
   - **익명 클래스**는 새로운 클래스를 정의하고 바로 인스턴스를 생성한다. 문법적으로 다소 복잡하고 장황하며, 여러 메서드를 동시에 오버라이드할 수도 있다.
   - **람다**는 단일 메서드만을 가지는 함수형 인터페이스를 구현할 때 사용되며, 문법이 간결하여 읽기 쉽다.

2. `this` **키워드 해석**
   - **익명 클래스** 내부의 `this`는 익명 클래스 자체의 인스턴스를 가리킨다.
   - **람다** 내부의 `this`는 람다가 선언된 외부 클래스의 인스턴스를 가리킨다.

3. **상속 및 상태 관리**
   - **익명 클래스**는 클래스를 상속하거나 여러 메서드를 가진 인터페이스를 구현할 수 있고, 내부에 **상태(필드)**를 가질 수 있다.

   - **람다**는 오직 함수형 인터페이스만 구현 가능하며, 내부 상태(필드)를 유지하기 어렵다(함수로서 동작).

4. **호환성과 내부 동작**
   - **익명 클래스**는 자바의 오래된 버전(람다 이전 버전)에서도 사용 가능하며, 컴파일 시 실제 익명 클래스(`OuterClass$1.class` 등)가 생성된다.
   - **람다**는 자바 8 이상에서 사용 가능하며, 내부적으로 `invokeDynamic`을 활용하여 별도의 클래스 파일이 아닌, 런타임 시점에 동적으로 람다 인스턴스를 생성한다.

5. **캡처링(capturing) 규칙**
   - 둘 다 외부 로컬 변수를 참조할 때, `final` 또는 "사실상 final"인 지역 변수만 참조할 수 있다.
   - 값이 변경되는 지역 변수는 캡처할 수 없다.

6. **언제 어떤 것을 사용할까?**
   - **복잡한 인터페이스 구현**(메서드가 여러 개)이 필요하거나, **상태를 유지**해야 하는 경우는 **익명 클래스**를 사용한다.
   - **간결성**과 **함수형 방식**이 필요한 경우(함수형 인터페이스 하나만 구현)에는 **람다**가 훨씬 직관적이며, 코드량을 줄일 수 있다.

정리

자바 8 이후 **람다**가 등장하면서 **익명 클래스**가 대부분 람다로 대체되었지만, 상황에 따라(여러 메서드 구현이 필요하거나 상태 유지가 필요한 경우 등) 여전히 **익명 클래스**를 사용해야 할 때가 있다.
선택 시 **가독성**, **유지보수성**, **필요 기능**(상태 관리 여부, 인터페이스 메서드 수 등)을 기준으로 판단하면 된다.
물론 둘 다 선택할 수 있다면 람다를 선택하는 것이 대부분 더 나은 선택이다.