5. 패키지

#0.강의/1.자바로드맵/2.자바-기본

패키지 - 시작

여러분이 쇼핑몰 시스템을 개발한다고 가정해보자. 다음과 같이 프로그램이 매우 작고 단순해서 클래스가 몇게 없다면 크게 고민할 거리가 없겠지만, 기능이 점점 추가되어서 프로그램이 아주 커지게 된다면 어떻게 될까?

아주 작은 프로그램

```
Order
User
Product
```

큰 프로그램

```
User
UserManager
UserHistory
Product
ProductCatalog
ProductImage
Order
OrderService
OrderHistory
ShoppingCart
CartItem
Payment
PaymentHistory
Shipment
ShipmentTracker
```

매우 많은 클래스가 등장하면서 관련 있는 기능들을 분류해서 관리하고 싶을 것이다.
컴퓨터는 보통 파일을 분류하기 위해 폴더, 디렉토리라는 개념을 제공한다. 자바도 이런 개념을 제공하는데, 이것이 바로 패키지이다.
다음과 같이 카테고리를 만들고 분류해보자.

```
* user
  * User
  * UserManager
  * UserHistory
* product
  * Product
  * ProductCatalog
  * ProductImage
* order
  * Order
  * OrderService
  * OrderHistory
* cart
  * ShoppingCart
  * CartItem
* payment
  * Payment
  * PaymentHistory
* shipping
  * Shipment
  * ShipmentTracker
```

여기서 `user`, `product` 등이 바로 패키지이다. 그리고 해당 패키지 안에 관련된 자바 클래스들을 넣으면 된다.
패키지(package)는 이름 그대로 물건을 운송하기 위한 포장 용기나 그 포장 묶음을 뜻한다.

패키지 사용

패키지 사용법을 코드로 확인해보자.
패키지를 먼저 만들고 그 다음에 클래스를 만들어야 한다.
패키지 위치에 주의하자.

pack.Data

```
package pack;

public class Data {
    public Data() {
        System.out.println("패키지 pack Data 생성");
    }
}
```

- **패키지를 사용하는 경우 항상 코드 첫줄에 `package pack`과 같이 패키지 이름을 적어주어야 한다.**
- 여기서는 `pack` 패키지에 `Data` 클래스를 만들었다.
- 이후에 `Data` 인스턴스가 생성되면 생성자를 통해 정보를 출력한다.

pack.a.User

```
package pack.a;

public class User {

    public User() {
        System.out.println("패키지 pack.a 회원 생성");
    }
}
```

- `pack` 하위에 `a`라는 패키지를 먼저 만들자.
- `pack.a` 패키지에 `User` 클래스를 만들었다.
- 이후에 `User` 인스턴스가 생성되면 생성자를 통해 정보를 출력한다.

참고: 생성자에 `public`을 사용했다. 다른 패키지에서 이 클래스의 생성자를 호출하려면 `public`을 사용해야 한다. 자세한 내용은 접근 제어자에서 설명한다. 지금은 코드와 같이 생성자에 `public` 키워드를 넣어두자.

pack.PackageMain1

```
package pack;

public class PackageMain1 {

    public static void main(String[] args) {
        Data data = new Data();
```

```
        pack.a.User user = new pack.a.User();
    }
}
```

`pack` 패키지 위치에 `PackageMain1` 클래스를 만들었다.

실행 결과

```
패키지 pack Data 생성
패키지 pack.a 회원 생성
```

- **사용자와 같은 위치:** `PackageMain1`과 `Data`는 같은 `pack`이라는 패키지에 소속되어 있다. 이렇게 같은 패키지에 있는 경우에는 패키지 경로를 생략해도 된다.
- **사용자와 다른 위치:** `PackageMain1`과 `User`는 서로 다른 패키지다. 이렇게 패키지가 다르면 `pack.a.User`와 같이 패키지 전체 경로를 포함해서 클래스를 적어주어야 한다.

패키지 - import

import

이전에 본 코드와 같이 패키지가 다르다고 `pack.a.User`와 같이 항상 전체 경로를 적어주는 것은 불편하다. 이때는 `import`를 사용하면 된다.

```
package pack;

import pack.a.User;

public class PackageMain2 {

    public static void main(String[] args) {
        Data data = new Data();
        User user = new User(); //import 사용으로 패키지 명 생략 가능
    }
}
```

실행 결과

```
패키지 pack Data 생성
패키지 pack.a 회원 생성
```

코드에서 첫줄에는 `package`를 사용하고, 다음 줄에는 `import`를 사용할 수 있다.
`import`를 사용하면 다른 패키지에 있는 클래스를 가져와서 사용할 수 있다.
`import`를 사용한 덕분에 코드에서는 패키지 명을 생략하고 클래스 이름만 적을 수 있다.

참고로 특정 패키지에 포함된 모든 클래스를 포함해서 사용하고 싶으면 `import` 시점에 `*(별)`을 사용하면 된다.

패키지 별(*) 사용

```
package pack;

import pack.a.*; //pack.a의 모든 클래스 사용

public class PackageMain2 {

    public static void main(String[] args) {
        Data data = new Data();
        User user = new User(); //import 사용으로 패키지 명 생략 가능
    }
}
```

이렇게 하면 `pack.a` 패키지에 있는 모든 클래스를 패키지 명을 생략하고 사용할 수 있다.

클래스 이름 중복

패키지 덕분에 클래스 이름이 같아도 패키지 이름으로 구분해서 같은 이름의 클래스를 사용할 수 있다.

```
pack.a.User
pack.b.User
```

이런 경우 클래스 이름이 둘다 `User`이지만 패키지 이름으로 대상을 구분할 수 있다.
이렇게 이름이 같은 경우 둘다 사용하고 싶으면 어떻게 해야할까?

pack.b.User

```
package pack.b;

public class User {

    public User() {
        System.out.println("패키지 pack.b 회원 생성");
    }
}
```

```
package pack;

import pack.a.User;

public class PackageMain3 {

    public static void main(String[] args) {
        User userA = new User();
        pack.b.User userB = new pack.b.User();
    }
}
```

같은 이름의 클래스가 있다면 `import`는 둘중 하나만 선택할 수 있다. 이때는 자주 사용하는 클래스를 `import`하고 나머지를 패키지를 포함한 전체 경로를 적어주면 된다. 물론 둘다 전체 경로를 적어준다면 `import`를 사용하지 않아도 된다.

패키지 규칙

패키지 규칙

- 패키지의 이름과 위치는 폴더(디렉토리) 위치와 같아야 한다. (필수)
- 패키지 이름은 모두 소문자를 사용한다. (관례)
- 패키지 이름의 앞 부분에는 일반적으로 회사의 도메인 이름을 거꾸로 사용한다. 예를 들어, `com.company.myapp`과 같이 사용한다. (관례)

    - 이 부분은 필수는 아니다. 하지만 수 많은 외부 라이브러리가 함께 사용되면 같은 패키지에 같은 클래스 이름이 존재할 수도 있다. 이렇게 도메인 이름을 거꾸로 사용하면 이런 문제를 방지할 수 있다.
    - 내가 오픈소스나 라이브러리를 만들어서 외부에 제공한다면 꼭 지키는 것이 좋다.
    - 내가 만든 애플리케이션을 다른 곳에 공유하지 않고, 직접 배포한다면 보통 문제가 되지 않는다.

패키지와 계층 구조

패키지는 보통 다음과 같이 계층 구조를 이룬다.

- `a`
    - `b`
    - `c`

이렇게 하면 다음과 같이 총 3개의 패키지가 존재한다.
`a`, `a.b`, `a.c`

계층 구조상 `a` 패키지 하위에 `a.b` 패키지와 `a.c` 패키지가 있다.
그런데 이것은 우리 눈에 보기에 계층 구조를 이룰 뿐이다. `a` 패키지와 `a.b`, `a.c` 패키지는 서로 완전히 다른 패키지이다.
따라서 `a` 패키지의 클래스에서 `a.b` 패키지의 클래스가 필요하면 `import` 해서 사용해야 한다. 반대도 물론 마찬가지이다.

정리하면 패키지가 계층 구조를 이루더라도 모든 패키지는 서로 다른 패키지이다.
물론 사람이 이해하기 쉽게 계층 구조를 잘 활용해서 패키지를 분류하는 것은 좋다. 참고로 카테고리는 보통 큰 분류에서 세세한 분류로 점점 나누어진다. 패키지도 마찬가지이다.

패키지 활용

실제 패키지가 어떤 식으로 사용되는지 예제를 통해서 알아보자. 실제 동작하는 코드는 아니지만, 큰 애플리케이션은 대략 이런식으로 패키지를 구성한다고 이해하면 된다. 참고로 이것은 정답은 아니고 프로젝트 규모와 아키텍처에 따라서 달라진다.

전체 구조도

- `com.helloshop`

- `user`
  - `User`
  - `UserService`
- `product`
  - `Product`
  - `ProductService`
- `order`
  - `Order`
  - `OrderService`
  - `OrderHistory`

com.helloshop.user 패키지

```
package com.helloshop.user;

public class User {
    String userId;
    String name;
}
```

```
package com.helloshop.user;

public class UserService {
}
```

com.helloshop.product 패키지

```
package com.helloshop.product;

public class Product {
    String productId;
    int price;
}
```

```
package com.helloshop.product;
```

```
public class ProductService {
}
```

com.helloshop.order 패키지

```
package com.helloshop.order;

import com.helloshop.product.Product;
import com.helloshop.user.User;

public class Order {

    User user;
    Product product;

    public Order(User user, Product product) {
        this.user = user;
        this.product = product;
    }
}
```

다른 패키지의 기능이 필요하면 `import`를 사용하면 된다.
생성자를 보면 `public`이 붙어있다. `public`이 붙어있어야 다른 패키지에서 생성자를 호출할 수 있다.

```
package com.helloshop.order;

import com.helloshop.product.Product;
import com.helloshop.user.User;

public class OrderService {
    public void order() {
        User user = new User();
        Product product = new Product();
        Order order = new Order(user, product);
    }
}
```

```
package com.helloshop.order;

public class OrderHistory {
}
```

패키지를 구성할 때 서로 관련된 클래스는 하나의 패키지에 모으고, 관련이 적은 클래스는 다른 패키지로 분리하는 것이 좋다.

정리