[12/19]

1) 다형성(polymorphism) : 객체 지향에서는 객체가 여러 개의 자료형 타입을 가질 수 있는 것
- 자식 인터페이스로 생성한 객체의 자료형은 부모 인터페이스로 사용하는 것이 가능하다.
- 해당 클래스에 포함되는 객체인지를 확인 (if 문에서 T/F로 사용)
- 인터페이스등을 사용해 복잡한 형태의 분기문을 간단히 처리 가능

```
Tiger tiger = new Tiger();  // Tiger is a Tiger
Animal animal = new Tiger();  // Tiger is a Animal
Predator predator = new Tiger();  // Tiger is a Predator
Barkable barkable = new Tiger();  // Tiger is a Barkable
```

2) 추상 클래스(abstract class) : 인터페이스의 역할과 클래스의 기능도 가지고 있는 '돌연변이' 클래스
- 추상 클래스는 일반 클래스와 달리 단독으로 객체를 생성할 수 없다. 반드시 추상 클래스를 상속한 실제 클래스를 통해서만 객체를 생성할 수 있다.

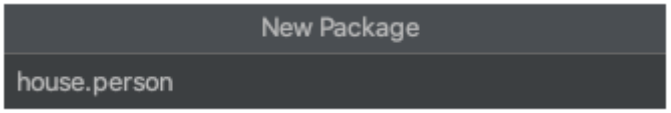
구분		추상 클래스	인터페이스
공통점	객체 생성	객체를 생성(인스턴스화)할 수 없다.	
	추상 메서드	추상 메서드를 포함한다.	
	기능적 목적	상속받는 클래스에서는 추상 메서드를 반드시 재정의하여 구현해야 한다.	
차이점	개념적 목적	상속 받아서 **기능을 확장**시키는데 목적	구현 객체의 **동일한 실행 기능을 보장**하기 위한 목적
	클래스	클래스다.(abstract class)	클래스가 아니다.(interface)
	일반 메서드	일반 메서드 정의가 가능	일반 메서드 정의 불가능 (Java8 이후 static, default 메서드 정의 가능)
	멤버 변수	클래스와 동일하게 변수 선언 및 사용 가능	상수만 사용 가능
	상속 키워드	extends	implements
	다중 상속	불가능	가능

인터페이스 특징 및 차이점

3) 패키지(package) : 비슷한 성격의 클래스들을 모아 놓은 자바의 디렉터리

- 파일관리를 위한 디렉토리 기능
- 클래스를 유일하게 구분할수 있는 식별자 기능
- 패키지가 다르다면 동일한 클래스명을 사용할 수 있다.

3-1) 서브패키지(subpackage) : 패키지 내의 하위 패키지



house 라는 패키지 안에 person 이라는 하위 패키지 생성

- **package** house.person; 가 자동생성

3-2) 패키지 호출

- import house.person; : house 패키지의 person 패키지 호출
- import house.*; : house 패키지 내의 모든 클래스를 사용
    - 만약 person 과 동일한 패키지 내에 있는 클래스라면 person 클래스를 사용하기 위해서 따로 import할 필요는 없다.
    - 같은 패키지 내에서는 import 없이도 사용할 수 있다.

4) 메소드 접근 제어자

- 종류 : public, private, protected, default

4-1) public : 내가 만든 클래스 이외에 다른 곳에서 호출할 수 있도록

- 퍼블릭으로 시작하는 것이 객체지향
- 객체지향 : 내가 만든 곳에서 사용 및 다른 곳에서 동일하게 적용

4-2) private : 현재의 클래스 안에서만 사용

4-3) protected : 필드, 생성자, 메소드가 적용대상이며, 자식클래스만 접근 가능

5) Static : 클래스에서 공유되는 변수나 메서드를 정의할 때 사용

- 메모리에 고정적으로 할당
    - Static이 붙지 않은 메서드나 변수의 경우 객체가 생성될 때마다 호출되어 서로 다른 값을 가지고 있을 수 있습니다. 그렇기 때문에 각 객체들에서 공통적으로 하나의 값이 유지되어야 할 경우 static을 유용하게 사용
- 객체 생성없이 사용 가능
    - static 키워드를 붙이면 객체 생성 없이도 메서드나 변수를 사용 가능
        - 객체생성 : Animal cat = new Animal
        - static이 있으면 바로 Animal. 으로 사용 가능
- 프로그램이 시작되면 메모리의 static 영역에 적재되고, 프로그램이 종료될 때 해제
    - 프로그램이 시작되어 클래스가 메모리에 올라가게 되면 static이 붙은 변수나 메서드는 클래스와 함께 자동으로 메모리의 static 영역에 생성
    - 일반적인 메서드는 객체를 생성하면 메모리의 Heap 영역에 올라가게 됩니다. 메모리의 이 Heap 영역은 Garbage Collector에 의해 자동으로 관리되게 됩니다. 즉 사용하지 않는 객체의 경우 알아서 삭제시킴으로써 메모리를 관리해주나 static 메서드는 static 영역에 존재하기 때문에 이러한 관리를 받지 못함
    - 메모리 할당을 한번만 해 적은 메모리 사용
    - 프로그램이 종료될 때 메모리를 해제하게 되는데, 이 때문에 과도하게 많은 static을 선언할 경우 메모리에 과부하가 올 수 있음

5-1) 같은 메모리 주소를 호출할 수 있기에 static 변수의 값을 공유하게 된다.

- static을 사용하지 않을 경우

```
class Counter  {
    2 usages
    int count = 0;
    2 usages
    Counter() {
        this.count++;
        System.out.println(this.count);
    }
}
public class Sample {
    public static void main(String[] args) {
        Counter c1 = new Counter(); //1
        Counter c2 = new Counter(); //1
    }
}
```

  - c1, c2 는 개별 객체로써 주소값을 가져오게 된다.

- static을 사용할 경우

```
class Counter  {
    2 usages
    static int count = 0;
    2 usages
    Counter() {
        count++;  // count는 더이상 객체변수가 아니므로 this를 제거
        System.out.println(count);  // this 제거
    }
}
public class Sample {
    public static void main(String[] args) {
        Counter c1 = new Counter(); //1
        Counter c2 = new Counter(); //2
    }
}
```

  - c1, c2 가 개별 객체이지만 주소의 값은 공유가 된다.

5-2) 스태틱 메소드(static Method)

```
class Counter  {
    static int count = 0;
    Counter() {
        count++;
        System.out.println(count);
    }

    public static int getCount() {
        return count;
    }
}

public class Sample {
    public static void main(String[] args) {
        Counter c1 = new Counter();
        Counter c2 = new Counter();

        System.out.println(Counter.getCount());  // 스태틱 메서드는
클래스를 이용하여 호출
    }
}
```

- Counter 클래스에 getCount()라는 스태틱 메서드를 추가
- 메서드 앞에 static 키워드를 붙이면 Counter.getCount()와 같이 객체 생성 없이도 클래스를 통해 메서드를 직접 호출할 수 있다.
    - 유틸리티성 메서드는 특정 클래스나 인스턴스에 종속되지 않고, 재사용이 가능하고 범용 기능을 제공하는 스태틱 메서드를 말한다. 이 메소드 들은 코드의 중복을 줄이고 가독성을 향상시킨다
        - ‘오늘의 날짜 구하기’, ‘숫자에 콤마 추가하기’ 등에 주로 사용