OOP(객체 지향 프로그래밍)의 특징

특징1. 캡슐화

참고) SW 공학에서 요구사항 변경에 대처하는 고전적인 설계 원리

높은 응집도와 낮은 결합도를 유지할 수 있도록 설계해야 요구사항을 변경할 때 유연하게 대처할 수 있다.

1.응집도(Cohesion)

클래스나 모듈 안의 요소들이 얼마나 밀접하게 관련되어 있는지를 나타낸다.

2.결합도(Coupling)

어떤 기능을 실행하는 데 다른 클래스나 모듈들에 얼마나 의존적인지를 나타낸다.

캡슐화는 낮은 결합도 를 유지할 수 있도록 해주는 객체지향 설계 원리다.

캡슐화는 정보 은닉 을 통해 높은 응집도와 낮은 결합도를 갖도록 한다.

정보 은닉(information hiding)

필요가 없는 정보는 외부에서 접근하지 못하도록 제한하는 것

자바에서는 캡슐화된 멤버를 노출시킬 것인지, 숨길 것인지를 결정하기 위해 접근 제한자(Access Modifier)를 사용한다.

특징2. 일반화(프로그래밍에서는 상속)

하위 클래스/객체 에서 상위 클래스/객체 로부터 필드와 메소드를 물려받아 하위 객체가 사용할 수 있도록 해주는 것. 상위 클래스 메소드의 수정으로 하위 클래스 객체들의 수정 효과를 가져오므로 유지 보수 시간을 줄여줄 수 있다.

+Peter Coad의 상속 규칙

상속의 오용을 막기 위해 상속의 사용을 제한하는 규칙

1. 자식 클래스와 부모 클래스 사이에는 '역할 수행(is role played by) 관계가 아니어야 한

다. ex) driver, 회사원 클래스와 사람 클래스의 관계

2. 한 클래스의 인스턴스는 다른 서브 클래스의 객체로 변환할 필요가 절대 없어야 한다.
 자식 클래스의 인스턴스 사이에 변환 관계가 필요한지 점검해보자
‘운전자’는 어떤 시점에서 ‘회사원’이 될 필요가 있으며, ‘회사원’ 역시 ‘운전자’가 될 필요가 있다. 이런 경우 객체의 변환 작업이 필요하므로 규칙에 위배된다.

3. 자식 클래스가 부모 클래스의 책임을 무시하거나 재정의하지 않고 확장만 해야한다.

4. 자식 클래스가 단지 일부 기능을 재사용할 목적으로 유틸리티 역할을 수행하는 클래스를 상속하지 않아야 한다.

두 자식 클래스 사이에 “is a kind of” 관계가 성립되지 않을 때 상속을 사용하면 불필요한 속성이나 연산(빚이라고 해도 될 것이다)도 물려받게 된다.
많은 사람들이 일반화 관계를 속성이나 기능의 상속, 즉 재사용을 위해 존재한다고 오해하고 있다. 그러나 이는 사실이 아니다!

예시)
```
public class MyStack<String> extends ArrayList<String> {
  public void push(String element) { add(element); }
  public String pop() { return remove(size() - 1); }
}
```

ArrayList의 isEmpty, size, add, remove 등의 메서드를 자신이 구현하지 않고 그대로 사용할 수 있다.
그러나 ArrayList 클래스에 정의된 Stack과 전혀 관련 없는 수많은 연산이나 속성도 같이 상속받게 된다.
문제점
Stack의 무결성 조건인 LIFO(Last In First Out)에 위배된다.
```
public static void main(String[] args) {
  MyStack<String> st = new MyStack<String>();
  st.push("1");
  st.push("2");
  st.set(0, "3"); // 허용되어서는 안됨. LIFO 위배!
  System.out.println(st.pop());
  System.out.println(st.pop());
}
```

Stack “is a kind of” ArrayList 관계가 아니기 때문에 일부 기능만을 사용하기 위해 부모로 만들지 않는다.
ArrayList 대신 Stack을 사용할 수 없으므로 위와 같이 사용하는 것은 바람직하지 못하다.

어떤 클래스의 일부 기능만 재사용하고 싶은 경우
위임(delegation) 을 사용한다.
자신이 직접 기능을 실행하지 않고 다른 클래스의 객체가 기능을 실행하도록 위임하는 것
따라서 일반화 관계는 클래스 사이의 관계지만 위임은 객체 사이의 관계다.
즉, 기능을 재사용할 때는 위임을 이용하라.
위임을 사용해 일반화(상속)을 대신하는 과정
자식 클래스에 부모 클래스의 인스턴스를 참조하는 속성을 만든다.
이 속성 필드를 this로 초기화한다.
자식 클래스에 정의된 각 메서드에 1번에서 만든 위임 속성 필드를 참조하도록 변경한다.
자식 클래스에서 일반화 관계 선언을 제거하고 위임 속성 필드에 부모 클래스의 객체를 생성해 대입한다.
자식 클래스에서 사용된 부모 클래스의 메서드를 추가하고 해당 메서드에도 속성 필드를 참조하도록 변경한다.
컴파일하고 잘 동작하는지 확인한다.
위의 잘못된 일반화 예시 코드를 수정하는 과정

```
public class MyStack<String> extends ArrayList<String> {
  public void push(String element) { add(element); }
  public String pop() { return remove(size() - 1); }
}
```

1) 부모 클래스의 인스턴스를 참조하는 속성(this)을 만들고
2) 위임 속성 필드를 참조하도록 변경한다.

```
public class MyStack<String> extends ArrayList<String> {
  // 1. 부모 클래스의 인스턴스를 참조하는 속성(this)
  private ArrayList<String> arrayList = this;
  // 2. arrayList.~ 추가
  public void push(String element) { arrayList.add(element); }
  public String pop() { return arrayList.remove(size() - 1); }
}
```

3) 일반화 관계를 제거하고 슈퍼 클래스 객체를 생성 후 대입한다.
4) 자식 클래스에서 사용된 부모 클래스의 메서드에도 위임 속성 필드를 참조하도록 변경한다.

```
// 3. 일반화 관계 제거
public class MyStack<String> {
  // 3. 슈퍼 클래스 객체를 생성 후 대입
  private ArrayList<String> arrayList = new ArrayList<String>();
  // 동일
  public void push(String element) { arrayList.add(element); }
  public String pop() { return arrayList.remove(size() - 1); }
  // 4. 사용된 메서드 추가 및 위임 속성 필드를 참조하도록 변경
```

```
  public boolean isEmpty() { return arrayList.isEmpty(); }
  public int size() { return arrayList.size(); }
}
```

5. 자식 클래스가 '역할(role)', '트랜잭션(transaction)', '디바이스(device)' 등을 특수화(부모 클래스에서 자식 클래스를 추출하는 과정)해야 한다.

특징3. 다형성

다형성은 서로 다른 클래스의 객체가 같은 메시지를 받았을 때 각자의 방식으로 동작하는 능력이다.
예) ‘talk’ Method를 가지고 있는 Abstract Class ‘동물’을 상속받은 Cat,Dog,Tiger 클래스가 있다고 가정하자.

다형성을 사용한 경우

```
// 부모 클래스
public abstract class Pet {
  public abstract void talk();
}
// 자식 클래스
public class Cat extends Pet {
  public void talk(){ System.out.println("야옹"); }
}public class Dog extends Pet {
  public void talk(){ System.out.println("멍멍"); }
}public class Tiger extends Pet {
  public void talk(){ System.out.println("안녕"); }
}
Cat cat;
Dog dog;
Tiger tiger;
```

cat.talk(), dog.talk(), tiger.talk()는 각각 다르게 동작한다.

접근 지정자

접근 지정자	접근 범위	동일 클래스	동일 패키지	다른 패키지의 자식 클래스	다른 패키지
public	접근 제한 없음	O	O	O	O
protected	동일 패키지와 상속 받은 클래스 내부	O	O	O	X
default	동일 패키지 내에서만	O	O	X	X
private	동일 클래스 내에서만	O	X	X	X

메소드 재정의(Overriding)

메소드를 오버라이딩 할때는 다음과 같은 규칙에 주의해서 작성해야 한다.
1. 부모의 메소드와 동일한 리턴 타입,메소드 이름,매개 변수 리스트를 가져야 한다.
2. 접근 제한을 더 강하게 오버라이딩 할 수 없다.(부모의 메소드는 public,자식의 메소드는 private)
3. 새로운 예외를 throws할 수 없다.(부모의 메소드와 비교하여)
+) 당연할수도 있지만 final은 Overriding 할 수 없다.

추상 클래스(abstract class)

클래스 중 객체를 직접 생성할 수 있는 클래스 : 실체 클래스
실체 클래스들의 공통적인 특성을 추출해서 선언한 클래스를 추상 클래스라고 한다.

```
public abstract class abClass{
//필드,생성자,메소드
}
```
와 같이 선언한다.

추상 클래스의 용도

1. 실체 클래스들의 공통된 필드와 메소드의 이름을 통일할 목적
2. 실체 클래스를 작성할 때 시간 절약

추상 클래스와 인터페이스의 공통점,차이는?

추상 클래스와 인터페이스는 선언만 있고 구현 내용이 없는 클래스이다.
따라서 이 둘을 가지고 새로운 인스턴스를 생성할 수 없다.
추상클래스를 extends로 상속받아 구현한 자식클래스나 인터페이스를 implements하고 구현한 자식클래스에서 객체를 생성할 수 있다.

차이점 : 추상 클래스와 인터페이스의 차이점은 목적이다.
추상클래스의 목적은 공통적인 기능을 하는 객체들의 추상화다.
인터페이스는 인터페이스 메서드를 구현하게 하는 것이 목적이다.
둘의 차이는 초점이 어디에 있느냐이다.
다중상속 가능/불가, 멤버 변수 존재 가능, 구현된 메서드 존재 가능(jdk8부터 인터페이스는 default method 구현 가능해짐. 일반적으로는 인터페이스는 구현이 없다.)의 차이가 있지만 제일 중요한 차이는 둘의 사용 목적이다.