REPORT

과 목 명 : 디자인패턴
담담교수 : 박제호 교수님
소　　속 : 소프트웨어학과
학　　번 : 32151671
이　　름 : 박민혁

단국대학교
Dankook University

1. 자바 4장 복습

(1) 객체 지향 프로그래밍 : 객체 지향 프로그래밍은 절차 지향 프로그래밍에 비해 다른 난이도를 가지고 있음

(2) 객체 지향의 특성
- 캡슐화(Encapsulation) : 보안성
- 상속(Inheritance) : 기존에 쓰던 코드를 가지고 와서 씀 (cost low)
- 다형성(Polymorphism) : 기존에 쓰던 코드를 이용 하되 변화를 주고 싶을 때 편리함
- 객체 지향 언어의 목적 = 생산성 향상(논의가 될 만한 이슈)
- Class : 붕어 틀 객체 : 붕어빵 (가면 갈수록 섞임)
- 필드 : 변수(static), 메소드 : 함수

(3) 생성자
- 생성자는 new를 통해 객체를 생성할 때, 객체당 한 번 호출
- 생성자는 리턴 타입을 지정할 수 없음
- 객체 초기화
- 객체가 생성될 때 반드시 호출 됨
- 개발자가 생성자를 작성하지 않았으면 컴파일러가 자동으로 기본 생성자를 삽입
- Overloading

(4) this
- 하나의 객체 안에서 사용(포인터), 생성자가 여러 개일 때, 생성자를 호출, 객체는 메모리가 할당이 되어야 한다.

(5) 인자 전달 방식
- Call by value : 메모리가 만들어 지면서 원래 있던 값(변화x)에 의해 processing
- Call by reference : 메모리를 공유 할 수 있게 메모리 주소에 접근
- Overloading : 이름을 같게 하고 싶을 때

(6) 자바의 접근 지정자
- Private : 같은 메모리 영역 안에서
- Protected : 상속 관련
- Public : 밖에 있는 것을 다른 곳에 호출
- Static : 객체를 만들지 않아도 사용 할 수 있는 메소드
- Final : 값 변경 x

2. 자바 5장 상속

(1) 객체 지향의 상속

- 부모클래스에 만들어진 필드, 메소드를 자식클래스가 물려받음
- 상속을 통해 간결한 자식 클래스 작성 (동일한 특성을 재 정의할 필요가 없어 자식 클래스가 간결해짐)

```
#상속 예제
class Tv
{
	boolean power; // 전원상태(on/off)
	int channel; // 채널

	void power() { power=!power; }
	void channelUp() { ++channel; }
	void channelDown() [ --channel }
}

class CaptionTv extends Tv
{
	boolean caption; // 캡션상태(on/off)
	void displayCaption(String text) {
		if(caption)
			System.out.println(text);
	}
}

class Caption TvTest
{
	public static void main(String args[]) {
		CaptionTv ctv = new CaptionTv();
		ctv.channel = 10; // 부모클래스로부터 상속받은 멤버
		ctv.channelUp(); // 부모클래스로부터 상속받은 멤버

		System.out.println(ctv.channel);
		ctv.displayCaption("Hello World");
		ctv.caption = true; // 캡션기능 on
		ctv.displayCaption("Hello World); // 캡션을 화면에 보여 준다.
	}
}
```

```
[실행결과]
11
Hello, World
```

(2) 객체 지향에서 상속의 장점
- 클래스의 간결화 : 멤버의 중복 작성이 불필요
- 클래스 관리 용이 : 클래스들의 계층적 분류
- 소프트웨어의 생산성 향상 : 재사용성과 확장 용이

-> 결국 디자인이 엉망이면 안 됨.

(3) 자바의 상속 선언
- extend keyword 사용 (원래 class 이름 extends 상위 class 이름)

(4) 자바 상속의 특징
- 클래스의 다중 상속을 지원하지 않는다. (디자인이 복잡해짐)

(5) 상속과 접근 지정자 4가지
- private : 상속 불가
- protected : 상속 가능(패키지가 달라도 됨)
- public : 모든 클래스에 상속 가능
- default : 패키지내 모든 클래스 상속 가능

(6) 서브클래스에서 슈퍼클래스의 생성자 선택
- 상속 관계에서의 생성자 : 슈퍼클래스와 서브클래스 각각 여러 생성자 작성 가능
- 서브클래스 생성자 작성 원칙 : 서브클래스 생성자에서 슈퍼클래스 생성자 하나 선택
- 서브클래스에서 슈퍼클래스의 생성자를 선택하지 않는 경우 : 컴파일러가 자동으로 슈퍼클래스의 기본 생성자 선택
- 서브클래스에서 슈퍼클래스의 생성자를 선택하는 방법 : super(패러미터 값 리스트) 이용
-> 슈퍼클래스의 생성자 중 패러미터 값 리스트와 동일한 시그니쳐를 가지는 생성자를 호출

```
#생성자 예제
class ParentClass { // 부모클래스
	private int age;

	public ParentClass(int age) { // 생성자
		this.age = age;
	}

	public void family() {
		System.out.println("부모입니다. 나이는 " + this.age + "살입니다.");
	}
}
```

```
class ChildClass extends ParentClass {
	private int age;

	public ChildClass(int age) { // 생성자
		super(age+30);
		this.age = age;
	}

	public void family() {
		System.out.println("자식입니다. 나이는 " + this.age + "살입니다.");
	}

	public void childMethod() {
		family(); // 자식클래스에 있는 family()
		super.family(); // 부모클래스에 있는 family()
	}
}

public class InheritanceTest {
	public static void main(String[] args) {

		ChildClass child = new ChildClass(20);
		child.childMethod(); // 자식클래스의 childMethod() 호출
	}
}
```

(7) super()
- 서브클래스에서 명시적으로 슈퍼클래스의 생성자 선택 호출
- 인자를 이용하여 슈퍼클래스의 적당한 생성자 호출
- 반드시 서브클래스 생성자 코드의 제일 첫 라인에 와야 함

(8) 업캐스팅(upcasting)
- 서브클래스 객체를 슈퍼클래스 타입으로 타입 변환 : 메모리 사이즈가 작은 것에서 위쪽으로 올라가는 것은 자동

(9) 다운캐스팅(downcasting)
- 슈퍼클래스 객체를 서브클래스 타입으로 변환
- 개발자의 명시적 타입 변환 필요

(10) instanceof 연산자

- 업캐스팅된 래퍼런스로 객체의 판단이 어렵다.
- 어떤 class를 가지고 만들어진 객체 인지 알 수 있음

```
#instanceof 예제
public class Instanceof {

	public static void main(String[] args) {

		FireEngine f = new FireEngine();
		Ambulance a = new Ambulance();

		Instanceof test = new Instanceof();
		test.doWork(f);
		test.doWork(a);

	}

	public void doWork(Car c) {
		if (c instanceof FireEngine) {
			FireEngine f = (FireEngine)c;
			f.water();

		} else if (c instanceof Ambulance) {
			Ambulance a = (Ambulance)c;
			a.siren();
		}
	}
}

class Car {
	String color;
	int door;

	void drive() { //운전하는 기능
		System.out.println("drive, brrrr~");
	}
void stop() { // 멈추는 기능
		System.out.println("stop!!!");
	}
}
```

```
class FireEngine extends Car {  // 소방차
    void water() {                  // 물 뿌리는 기능
        System.out.println("warter!!!");
    }
}

class Ambulance extends Car {  // 앰뷸런스
    void siren() {                  // 사이렌을 울리는 기능
        System.out.println("siren~~~~");
    }
}
```

실행결과 :

```
warter!!!
siren~~~~
```

(11) 메소드 오버라이딩
- 뒤집는 것(메소드의 시그니처는 바꿀 수 없음) -> body 부분만

(12) 오버라이딩의 목적, 다형성 실현
- 슈퍼클래스에 선언된 메소드를, 각 서브클래스들이 자신만의 내용으로 새로 구현하는 기능
- 상속을 통해 하나의 인터페이스(같은 이름)에 서로 다른 내용 구현이라는 객체 지향의 다형성 실현
- 동적 바인딩을 통해 실행 중에 다형성 실현

(13) 오버라이딩 vs 오버로딩

비교 요소	메소드 오버로딩	메소드 오버라이딩
선언	같은 클래스나 상속 관계에서 동일한 이름의 메소드 중복 작성	서브 클래스에서 슈퍼 클래스에 있는 메소드와 동일한 이름의 메소드 재작성
관계	동일한 클래스 내 혹은 상속 관계	상속 관계
목적	이름이 같은 여러 개의 메소드를 중복 작성하여 사용의 편리성 향상. 다형성 실현	슈퍼 클래스에 구현된 메소드를 무시하고 서브 클래스에서 새로운 기능의 메소드를 재정의하고자 함. 다형성 실현
조건	메소드 이름은 반드시 동일하고, 매개변수 타입이나 개수가 달라야 성립	메소드의 이름, 매개변수 타입과 개수, 리턴 타입이 모두 동일하여야 성립
바인딩	정적 바인딩. 호출될 메소드는 컴파일 시에 결정	동적 바인딩. 실행 시간에 오버라이딩된 메소드 찾아 호출

```
#오버라이딩 예제
class Woman{ //부모클래스
	public String name;
	public int age;

	//info 메서드
	public void info(){
		System.out.println("여자의 이름은 "+name+", 나이는 "+age+"살입니다.");
	}

}

class Job extends Woman{ //Woman클래스(부모클래스)를 상속받음 :

	String job;

	public void info() {//부모(Woman)클래스에 있는 info()메서드를 재정의
		super.info();
		System.out.println("여자의 직업은 "+job+"입니다.");
	}
}

public class OverTest {

	public static void main(String[] args) {

		//Job 객체 생성
		Job job = new Job();

		//변수 설정
		job.name = "유리";
		job.age = 30;
		job.job = "프로그래머";

		//호출
		job.info();

	}

}
```

(14) 추상 메소드
- 추상 메소드 : 선언 되어 있으나 구현 되어 있지 않은 메소드(body가 존재하지 않음)
- 추상 메소드는 서브클래스에서 오버라이딩 하여 구현해야 함

(15) 추상 클래스
- 추상 메소드를 하나라도 가진 클래스 : 클래스 앞에 반드시 abstract라고 선언해야 함
- 추상 메소드가 하나도 없지만 abstract로 선언 된 클래스
- 추상 메소드가 유무에 상관없이, 추상 클래스를 사용하고 싶으면 abstract를 반드시 명시

(16) 추상 클래스의 용도
- 설계와 구현 분리 -> 형태의 원형 형태를 정의
- 계층적 상속 관계를 갖는 클래스 구조를 만들 때

(17) 자바의 인터페이스
- 클래스가 구현해야 할 메소드들이 선언되는 추상형
- 인터페이스 선언 : class 대신에 interface 선언
- 다중 상속이 가능하다.

(18) 인터페이스의 목적
- 인터페이스는 스펙을 주어 클래스들이 그 기능을 서로 다르게 구현할 수 있도록 하는 클래스의 규격 선언이며, 클래스의 다형성을 실현하는 도구이다.

(19) 추상 클래스와 인터페이스 비교
- 유사점 : 객체를 생성할 수 없고, 상속을 위한 슈퍼클래스로만 사용, 클래스의 다형성을 실현하기 위한 목적

- 다른점 :

비교	목적	구성
추상 클래스	추상 클래스는 서브 클래스에서 필요로 하는 대부분의 기능을 구현하여 두고 서브 클래스가 상속받아 활용할 수 있도록 하되, 서브 클래스에서 구현할 수밖에 없는 기능만을 추상 메소드로 선언하여, 서브 클래스에서 구현하도록 하는 목적(다형성)	• 추상 메소드와 일반 메소드 모두 포함 • 상수, 변수 필드 모두 포함
인터페이스	인터페이스는 객체의 기능을 모두 공개한 표준화 문서와 같은 것으로, 개발자에게 인터페이스를 상속받는 클래스의 목적에 따라 인터페이스의 모든 추상 메소드를 만들도록 하는 목적(다형성)	• 변수 필드(멤버 변수)는 포함하지 않음 • 상수, 추상 메소드, 일반 메소드, default 메소드, static 메소드 모두 포함 • protected 접근 지정 선언 불가 • 다중 상속 지원

```
#추상클래스와 추상 메소드 예제
public class AbstractTest {
	public static void main(String[] args) {
		FirstCat fc = new FirstCat();
		SecondCat sc = new SecondCat();

		fc.call();
		sc.call();
	}
}//AbstractTest

abstract class Cat{ // 추상 메서드를 포함하므로 추상클래스로 선언
	abstract void call(); // 추상 메서드 선언(구현x)
	void call2(){
		System.out.println("일반 메서드");
	}
}

//Cat 추상클래스를 상속한 클래스들
class FirstCat extends Cat{
	void call(){ //추상메서드는 서브클래스에서 반드시 재정의 되어야 함
		System.out.println("첫번째 야옹이");
	}
}

class SecondCat extends Cat{
	void call(){
		System.out.println("두번째 야옹이");
	}
}
```

실행 결과 화면

```
#interface 예제
interface catWorld{
	public void call();
}


public class InterfaceTest implements catWorld{

	public void call() { //오버라이드
		System.out.println("야옹야옹!");
	}

	public static void main(String[] args) {
		InterfaceTest it = new InterfaceTest();

		it.call();
	}

}
```

실행 결과 화면