5차시 정리

```
// 상속
// 객체 지향의 상속 : 부모 클래스에 만들어진 필드, 메소드를 자식 클래스가 물려받음

// 객체 지향에서 상속의 장점
// 클래스의 간결화 : 멤버의 중복 작성 불필요
// 클래스 관리 용이 : 클래스들의 계층적 분류
// 소프트웨어의 생산성 향상 : 클래스 재사용과 확장 용이, 새로운 클래스의 작성 속도 빠름

// 자바에 상속 선언
// extends 키워드 사용
// 부모 클래스 -> 슈퍼 클래스
// 자식 믈래스 -> 서브 클래스

class Point {
	// private으로 선언을 했으므로 x, y값을 기입하기 위해 따로 함수를 이용함.
	private int x, y; // 한 점을 구성하는 x, y 좌표
	public void set(int x, int y) { // private의 x, y를 기입하기 위한 함수
		this.x = x; this.y = y;
	}
	public void showPoint() { // 점의 좌표 출력
		System.out.println("(" + x + "," + y + ")");
	}
}

//Point를 상속받은 ColorPoint 선언
class ColorPoint extends Point {
	private String color; // 점의 색, private으로 선언을 했으므로 color값을 기입하기 위해 따로 함수를 이용해야함
	public void setColor(String color) { // private의 color를 기입하기 위한 함수
		this.color = color;
	}
	public void showColorPoint() { // 컬러 점의 좌표 출력
		System.out.print(color);
		showPoint(); // Point 클래스의 showPoint() 호출
	}
}
public class ColorPointEx {
	public static void main(String[] args) {

		Point p = new Point(); // Point 변수 선언 및 객체 생성
		p.set(1, 2); // Point 클래스의 set() 호출
		p.showPoint();

		ColorPoint cp = new ColorPoint(); // ColorPoint 객체
		cp.set(3, 4); // Point의 set() 호출
		cp.setColor("red"); // color는 private이므로 값을 대입하기 위해 ColorPoint의 setColor() 호출
		cp.showColorPoint(); // 컬러와 좌표 출력
	}
}

// 결과
// (1,2)
// red(3,4)
```

```
// 자바 상속의 특징
// 클래스의 다중 상속 지원하지 않고, 상속 횟수 무제한.
// 상속의 최상위 조상 클래스는 java.lang.Object 클래스이다.

// 자바의 접근 지정자 4가지(public, protected, 디폴트, private)
// 슈퍼클래스의 private 멤버는 다른 모든 클래스에 접근 불허
// 슈퍼클래스의 디폴트 멤버는 패키지내 모든 클래스에 접근 허용
// 슈퍼클래스의 protected 멤버는 같은 패키지 내의 모든 클래스 혹은 다른 패키지의 서브 클래스는 접근 가능
// 슈퍼클래스의 public 멤버는 다른 모든 클래스에서 접근 가능

class Person {

	public String name;
	int age;
	protected int height;
	private int weight; // private 때문에 직접 값 입력 불가, 함수를 통해 입력

	public void setWeight(int weight) { // private 인 weight에 값을 넣기 위해 함수를 사용
		this.weight = weight;
	}

	public int getWeight() {
		System.out.print(weight); // weight 출력
		return weight; // weight값 리턴
	}
}
class Student extends Person {
	public void set() {
		age = 30; // 슈퍼 클래스의 디폴트 멤버 접근 가능
		name = "홍길동"; // 슈퍼 클래스의 public 멤버 접근 가능
		height = 175; // 슈퍼 클래스의 protected 멤버 접근 가능
		// weight = 99; // 오류. 슈퍼 클래스의 private 접근 불가
		setWeight(99); // private 멤버 weight은 setWeight()으로 간접 접근
	}
}
public class InheritanceEx {
	public static void main(String[] args) {
		Student s = new Student(); // 객체 선언
		s.set(); // 값 기입
		s.getWeight(); // weight를 출력
	}
}

// 결과
// 99
```

```
// 슈퍼클래스와 서브 클래스의 생성자 간의 호출 및 실행 관계

// 상속 관계에서의 생성자
// 슈퍼클래스와 서브 클래스 각각 여러 생성자 작성 가능
// 서브 클래스에서 슈퍼클래스의 생성자 하나 선택(super()함수 사용)을 하며 실행,
// 선택하지 않는 경우는 컴파일러가 자동으로 슈퍼 클래스의 기본생성자 선택하여 실행, 기본생성자가 없으면 오류
// 단, super()를 호출하려면 서브클래스의 생성자코드의 제일 첫 라인에서 선언해야 함
class A{
	// public A() { System.out.println("class A 기본생성자"); } // 기본생성자 삭제한다면
	public A(int x) { System.out.println("class A 생성자1"); }
	public A(int x, int y) { System.out.println("class A 생성자2"); }
}
class B extends A{
	public B() { System.out.println("class B 기본생성자"); }
	public B(int x) { super(x);	System.out.println("class B 생성자1"); }
	public B(int x, int y) { super(x,y);	System.out.println("class B 생성자2"); }
}
class Point {
	private int x, y; // 한 점을 구성하는 x, y 좌표
	public Point() { // private x, y에 값을 대입하기 위한 기본생성자
		this.x = this.y = 0;
	}
	public Point(int x, int y) { // 인수가 2개짜리 생성자1
		this.x = x; this.y = y;
	}
	public void showPoint() { // 점의 좌표 출력
		System.out.println("(" + x + "," + y + ")");
	}
}
class ColorPoint extends Point { // Point를 상속받음
	private String color; // 점의 색
	public ColorPoint(int x, int y, String color) { // 인수 3개짜리 생성자로 입력받은 좌표와 색상을 대입
		super(x, y); // Point에 기본생성자자가 아닌 인수가 2개인 생성자를 호출
		this.color = color;
	}
	public void showColorPoint() { // 컬러 점의 좌표 출력
		System.out.print(color);
		showPoint(); // Point 클래스의 showPoint() 호출
	}
}
public class SuperEx {
	public static void main(String[] args) {
		ColorPoint cp = new ColorPoint(5, 6, "blue");
		cp.showColorPoint();
		// B클래스에서 A클래스로 생성자 호출할 때 super가 없어서 기본생성자를 호출해야 하는데 없어서 오류
		// B b = new B();
		System.out.println();
		B b1 = new B(5);
		System.out.println();
		B b2 = new B(5,5);
	}
}
// 출력
// blue(5,6)
//
// class A 생성자1
// class B 생성자1
//
// class A 생성자2
// class B 생성자2
```

```
// 업캐스팅 : 서브 클래스 객체를 슈퍼 클래스 타입으로 타입 변환
// 업캐스팅 된 래퍼런스(변수)는 객체 내에 슈퍼 클래스의 멤버만 접근 가능
// 업캐스팅이 되었다면 서브 클래스의 필드와 메소드는 사용할 수 없다.
// 사용 예)
// super S ; 슈퍼클래스의 변수 선언
// sub s = new sub; 서브클래스의 변수 선언와 레퍼런스 할당
// S = s; 슈퍼클래스변수에 서브클래스변수를 대입, 업캐스팅

// 다운캐스팅 : 슈퍼 클래스 객체를 서브 클래스 타입으로 변환
// 다운캐스팅을 할때 어떤 서브 클래스로 선언할 지 명시적인 타입변환을 호출해야한다.
// 사용 예)
// super S = new super; 슈퍼클래스의 변수 선언과 레퍼런스 할당
// sub s; 서브클래스의 변수 선언
// s = (sub)S; 서브클래스로 강제 형변환으로 다운 캐스팅

// instanceof 연산자 : 레퍼런스가 가리키는 객체의 타입식별을 위해 사용
// 반환값으로 True / False
// 사용 예)
// 변수 instanceof 클래스 ; 변수이 클래스타입(슈퍼클래스도 가능)이면 True, 아니면 False

class Person { }
class Student extends Person { }
class Researcher extends Person { }
class Professor extends Researcher { }

public class InstanceOfEx {
	static void print(Person p) {
		if(p instanceof Person)
			System.out.print("Person ");
		if(p instanceof Student)
			System.out.print("Student ");
		if(p instanceof Researcher)
			System.out.print("Researcher ");
		if(p instanceof Professor)
			System.out.print("Professor ");
		System.out.println();
	}
	public static void main(String[] args) {
		System.out.print("new Student() ->\t");
		print(new Student()); // st는 p타입 st타입이고, st는 re타입이 pr타입이 아님.
		System.out.print("new Researcher() ->\t");
		print(new Researcher()); // re는 p타입 re타입이고, re는 st타입이 pr타입이 아님.
		System.out.print("new Professor() ->\t");
		print(new Professor()); // pr는 p타입 re타입 pr타입 이고, pr는 st타입이 아님.
	}
}

// 출력
// new Student() ->Person Student
// new Researcher() ->	Person Researcher
// new Professor() ->	Person Researcher Professor
```

```
// 메소드 오버라이딩(무시하기, 덮어쓰기)
// 수퍼 클래스에 선언된 메소드를 각 서브 클래스들이 자신만의 내용으로 새로 구현하는 기
// 동적 바인딩을 통해 실행 중에 다형성 실현되며, 슈퍼 클래스의 메소드를 서브 클래스에서 재정의

// 동적 바인딩
// 업캐스팅 시 재정의로 인하여 슈퍼클래스에 정의된 메서드를 호출했지만 서브 클래스에 오버라이딩된 메소드가 무조건 실행되는 것
// 이와 반대로 컴파일러에서  super의 접근은 정적 바인딩으로 처리

class Shape { // 슈퍼 클래스
	public Shape next;
	public Shape() {
		next = null;
	}
	public void draw() {
		System.out.println("Shape");
	}
}
class Line extends Shape {
	public void draw() { // shape를 line에 맞게 메소드 오버라이딩
		System.out.println("Line");
	}
}
class Rect extends Shape {
	public void draw() { // shape를 rect에 맞게 메소드 오버라이딩
		System.out.println("Rect");
	}
}
class Circle extends Shape {
	public void draw() { // shape를 circle에 맞게 메소드 오버라이딩
		System.out.println("Circle");
	}
}
public class MethodOverridingEx {
	static void paint(Shape p) {
		p.draw(); // p가 가리키는 객체 내에 오버라이딩된 draw() 호출. 동적 바인딩
	}
	public static void main(String[] args) {
		Line line = new Line();
		paint(line);
		paint(new Shape());
		paint(new Line()); // 메소드 오버라이딩 사용
		paint(new Rect());  // 메소드 오버라이딩 사용
		paint(new Circle());  // 메소드 오버라이딩 사용
	}
}

// 출력
// Line
// Shape
// Line
// Rect
// Circle
```

// 동적바인딩 : 컴파일러가 동적으로 실행할 메소드를 실행할 시에 결정
// 업캐스팅 시 오버라이딩이 호출된 함수가 원래 슈퍼클래스의 함수를 대체하는 상황

// super는 슈퍼클래스의 멤버를 접근할 때 사용되는 레퍼런스로 서브 클래스에서만 사용한다.
// 컴파일러는 super의 접근을 정적바인딩으로 처리

// 메소드 오버로딩(생성자 오버로딩) : super키워드를 사용하여 슈퍼클래스의 생성자를 호출 시 정적 바인딩
// 이름이 같은 여러 개의 메소드를 중복 작성하여 사용의 편리성 향상, 다형성 실현
// 메소드 오버라이딩 (메소드 재정의) : 업캐스팅 후 오버라이딩 된 메서드 호출시 동적 바인딩
// 슈퍼 클래스에 구현된 메소드를 무시하고 서브 클래스에서 새로운 기능의 메소드를 재정의. 다형성 실현

비교요소	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 규현된 메소드를 무시하고, 서브 클래스에서 새로운 기능의 메소드를 재정의하고자 함, 다형성 실현
조건	메소드 이름은 반드시 통일하고, 매개변수 타입이나 개수가 달라야 성립	메소드의 이름, 매개변수 타입과 개수, 리턴타입이 모두 동일하여야 성립
바인딩	정적 바인딩, 호출될 메소드는 컴파일 시에 결정	동적 바인딩 실행 시간에 오버라이딩된 메소드 찾아 호출

```
class Weapon {
	protected int fire() {
		return 1; // 무기는 기본적으로 한 명만 살상
	}
}
class Cannon extends Weapon {
	@Override
	protected int fire() { // 오버라이딩
		return 10; // 대포는 한 번에 10명을 살상
	}
}

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

		Weapon weapon = new Weapon();
		System.out.println("기본 무기의 살상 능력은 " + weapon.fire());

		weapon = new Cannon();
		// weapon.fire()를 하면서 weapon.fire가 cannon.fire로 오버라이딩 되는 상황
		// 동적 바인딩이 실행
		System.out.println("대포의 살상 능력은 " + weapon.fire());
	}
}

// 출력
// 기본 무기의 살상 능력은 1
// 대포의 살상 능력은 10
```

```
// 추상 메소드 (abstract)
// 선언되어 있느나 구현되어 있지 않은 메소드로 서브 클래스에서 오버라이딩하여 구현해야함.

// 추상 메소드의 종류
// 추상 메소드를 하나라도 포함하는 클래스
// 추상 메소드가 하나도 없지만 추상 클래스로 선언한 클래스

// 추상클래스의 상속
// 그냥 단순 상속시 서브 클래스도 추상 클래스이므로 abstract로 선언해야 함
// 서브 클래스에서 오버라이딩을 하여 추상메소드를 재정의 하지 않으면 서브 클래스도 추상 클래스이다.
// 추상의 부분을 오버라이딩 한다면 완전한 클래스가 되면서 그냥 클래스가 된다.
// 서브 클래스에서 오버라이딩을 하여 추상메소드를 재정의하면 완전해지며 사용가능하다.

// 추상 클래스의 특징
// 추상 클래스는 객체를 생성 할 수 없다.
// 설계와 구현의 분리(슈퍼 클래스에서 개념 정의, 서브 클래스에서 구현)
// 계층적 상속 관계를 갖는 클래스 구조를 만들 때

abstract class Calculator {
	public abstract int add(int a, int b); // 추상 메소드
	public abstract int subtract(int a, int b); // 추상 메소드
	public abstract double average(int[] a); // 추상 메소드
}
public class GoodCalc extends Calculator {
	@Override // 제정의 하여 추상 해제
	public int add(int a, int b) { // 추상 메소드 구현
		return a + b;
	}
	@Override // 제정의 하여 추상 해제
	public int subtract(int a, int b) { // 추상 메소드 구현
		return a - b;
	}
	@Override // 제정의 하여 추상 해제, 추상 메소드가 없어서 완전한 클래스를 선언함
	public double average(int[] a) { // 추상 메소드 구현
		double sum = 0;
		for (int i = 0; i <a.length; i++)
			sum += a[i];
		return sum/a.length;
	}

	public static void main(String[] args) {
		GoodCalc c = new GoodCalc();
		System.out.println(c.add(2,3));
		System.out.println(c.subtract(2,3));
		System.out.println(c.average(new int [] { 2,3,4 }));
	}
}

// 출력
// 5
// -1
// 3.0
```

```
// 실세계의 인터페이스
// 정해진 규격(인터페이스)에 맞기만 하면 연결 가능. 각 회사마다 구현방법은 다름
// 정해진 규격(인터페이스)에 맞지않으면 연결 불가.

// 자바의 인터페이스(interface 키워드로 선언)란
// 클래스가 구현해야 할 메소드들이 선언되는 추상형
// 스펙을 주어 클래스들이 그 기능을 서로 다르게 구현할 수 있도록 하는 클래스의 규격 선언이며
// 클래스의 다형성을 실현하는 도구이다.
// 설계를 하는 것처럼 이것이것을 여기서 해야된다 라는 선언을 하는 것

// 여전히 인터페이스에는 필드 선언 불가
// 인터페이스의 객체 생성 불가, 변수 선언 가능
// new PhoneInterface(); : 불가 / PhoneInterface galaxy; : 가능

// 다른 인처페이스 상속 가능, 다중 상속 가능
// 상속 : interface MobilePhoneInterface extends PhoneInterface
// 다중상속 : interface MusicPhoneInterface extends MobilePhoneInterface, MP3Interface
// 인터페이스에 추상 메소드를 모두 구현하지 못했으면 단순 상속 extends 키워드 사용
// 인터페이스에 모든 추상 메소드를 모두 구현하면 implements 키워드 사용

// 인터페이스의 구성요소들
// 상수 : public 만 허용, public static final 생략
// 추상 메소드 : public abstract 생략
// default 메소드 : 인터페이스에 코드가 작성되는 메소드. public 접근 지정자만 허용, 생략가능
// private 메소드 : 인터페이스 내에 있는 메소드 코드가 작성되야함. 같은 인터페이스 내 다른 메소드에 의해서만 호출가능
// static 메소드 : public, private 모두 지정 가능. 생략하면 public

interface PhoneInterface { // 인터페이스 선언
	final int TIMEOUT = 10000; // 상수 필드 선언, public static 생략
	void sendCall(); // 추상 메소드, public abstract 생략
	void receiveCall(); // 추상 메소드, public abstract 생략
	default void printLogo() { // default 메소드, public 생략
		System.out.println("** Phone **");
	}
}
class SamsungPhone implements PhoneInterface { // 인터페이스 구현 완료 후 implements 추가
	// PhoneInterface의 모든 추상 메소드 구현 후 implements 붙임
	@Override
	public void sendCall() {
		System.out.println("띠리리리링");
	}
	@Override
	public void receiveCall() {
		System.out.println("전화가 왔습니다.");
	}
	// 메소드 추가 작성
	public void flash() {
		System.out.println("전화기에 불이 켜졌습니다.");
	}
}

public class InterfaceEx {
	public static void main(String[] args) {
		SamsungPhone phone = new SamsungPhone();
		phone.printLogo();
		phone.sendCall(); // 오버라이딩으로 재정의 후
		phone.receiveCall();
		phone.flash();
	}
}
```

```
// 출력
// ** Phone **
// 띠리리리링
// 전화가 왔습니다.
// 전화기에 불이 켜졌습니다.
```

```
// 다른 인처페이스 상속 가능, 다중 상속 가능
// 상속 : interface MobilePhoneInterface extends PhoneInterface
// 다중상속 : interface MusicPhoneInterface extends MobilePhoneInterface, MP3Interface
// 인터페이스에 추상 메소드를 모두 구현하지 못했으면 단순 상속 extends 키워드 사용
// 인터페이스에 모든 추상 메소드를 모두 구현하면 implements 키워드 사용

// 추상 클래스와 인터페이스 비교
// 유사점
// 객체를 생성할 수 없고, 상속을 위한 슈퍼 클래스로만 사용, 클래스의 디형성을 실현하기 위한 목적
// 다른점
// 추상 클래스 : 추상 메소드와 일반 메소드, 상수 변수 필드 모두 포함
// 인터페이스 : 변수 필드는 포함하지 않음, protected 접근 지정 선언 불가, 다중 상속 지원
```

비교	목적	구성
추상클래스	추상 클래스는 서브 클래스에서 필요로 하는 대부분의 기능을 구현하여 두고, 서브 클래스가 상속받아 활용할 수 있도록 하되, 서브 클래스에서 구현할 수 밖에 없는 기능만을 추상 메소드로 선언하여 서브 클래스에서 규현하도록 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 추상메소드를 만들도록 하는 목적(다형성)	- 변수 필드(멤버 변수)는 포함하지 않음 - 상수, 추상 메소드, 일반 메소드, default 메소드, static 메소드 모두 포함 - proteced 접근 지정 선언 불가 - 다중 상속 지원

```
interface PhoneInterface { // 인터페이스 선언
	final int TIMEOUT = 10000; // 상수 필드 선언, public static 생략
	void sendCall(); // 추상 메소드, public abstract 생략
	void receiveCall(); // 추상 메소드, public abstract 생략
	default void printLogo() { // default 메소드, public 생략
		System.out.println("** Phone **");
	}
}
interface MobilePhoneInterface extends PhoneInterface { // 인터페이스를 상속받아 총 4개의 추상메소드 선언
	void sendSMS(); // 추상 메소드, public abstract 생략
	void receiveSMS(); // 추상 메소드, public abstract 생략
}
interface MP3Interface { // 인터페이스 선언, 2개의 추상메소드 선언
	public void play(); // 추상 메소드, public abstract 생략
	public void stop(); // 추상 메소드, public abstract 생략
}
class PDA { // 클래스 작성
	public int calculate(int x, int y) {
		return x + y;
	}
}
```

```
// SmartPhone 클래스는 PDA를 상속받고,
// 다중 인터페이스 상속으로 MobilePhoneInterface와 MP3Interface 인터페이스를 상속받고
// 두 인터페이스에 선언된 6개의 추상 메소드를 모두 구현한다.
class SmartPhone extends PDA implements MobilePhoneInterface, MP3Interface {

	// MobilePhoneInterface의 추상 메소드 구현
	@Override
	public void sendCall() {
		System.out.println("따르릉따르릉~~");
	}
	@Override
	public void receiveCall() {
		System.out.println("전화 왔어요.");
	}
	@Override
	public void sendSMS() {
		System.out.println("문자갑니다.");
	}
	@Override
	public void receiveSMS() {
		System.out.println("문자왔어요.");
	}

	// MP3Interface의 추상 메소드 구현
	@Override
	public void play() {
		System.out.println("음악 연주합니다.");
	}
	@Override
	public void stop() {
		System.out.println("음악 중단합니다.");
	}

	// 추가로 작성한 메소드
	public void schedule() {
		System.out.println("일정 관리합니다.");
	}
}
public class InterfaceEx2 {
	public static void main(String[] args) {
		SmartPhone phone = new SmartPhone();
		phone.printLogo(); // default 메소드
		phone.sendCall(); // MobilePhoneInterface 인터페이스의 오버라이딩한 메소드 사용
		phone.play(); // MP3Interface 인터페이스의 오버라이딩한 메소드 사용
		System.out.println("3과 5를 더하면 " + phone.calculate(3,5)); // 슈퍼클래스의 메소드 사용
		phone.schedule(); // 추가로 작성한 메소드 사용
	}
}
```