JAVA 상속

상속의 기본 개념과 문법
상속과 생성자 호출
상속을 적용할 수 있는 두 클래스의 관계
메서드 오버라이딩(Method Overriding)
다형성(Polymorphism)
오버라이딩과 다형성, 그리고 상속의 의미

상속을 이용하여 클래스를 선언하면 상속의 대상이 되는 클래스의 맴버변수와 메서드를 선언하고 있는 클래스에서 사용할 수 있다.

```
class Man {
    String name;
    public void tellName(){
        System.out.println("name is " + name);
    }
}

class BusinessMan extends Man {
    String company;
    public void tellCompany(){
        System.out.println("Company is " + company);
    }
}
```

```
public static void main(String[] args){
    BusinessMan m = new BusinessMan();
    m.name = "kim";
    System.out.println(m.name);
    m.tellName();
    m.tellCompany();
}
```

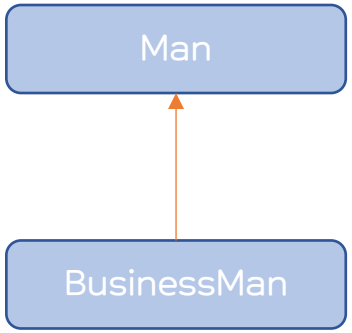
클래스 선언 시 클래스명 뒤에 'extends 상속할클래스명'을 작성하면

extends 키워드 뒤에 작성한 클래스의 맴버변수 및 메서드를 상속받는다.

상속 받은 맴버변수와 메서드는 마치 자신의 것처럼 사용할 수 있다.

```
class Man {
   ...
}

class BusinessMan extends Man {
  ...
}
```



도식화

상속의 대상이 되는 클래스(현 예제에서는 Man 클래스)를 상위 클래스, 부모 클래스, 수퍼 클래스라 부른다.

상속을 하는 클래스(현 예제에서는 BusinessMan 클래스)를 하위 클래스, 자식 클래스, 서브 클래스라 부른다.

클래스 선언 시 모든 맴버변수는 생성자를 통해 초기값을 설정하는 것이 좋으므로 BusinessMan클래스에 아래와 같이 생성자를 추가하자.

```
class BusinessMan extends Man {
  String company;

  public BusinessMan(String name, String company){
    this.name = name;
    this.company = company;
  }

  public void tellCompany(){
    System.out.println("Company is " + company);
  }
}
```

BusinessMan 클래스는 Man클래스를 상속하므로 Man 클래스에서 선언한 맴버변수 name을 가지고 있다. 그렇기 때문에 생성자에서는 상속받은 name변수의 값도 초기화하고 있다.

하지만 이렇게 상속받은 맴버변수의 값을 초기화하는 것은 좋은 코드가 아니다. 클래스 설계 관점에서 보면 아무리 상속을 받았다고 하더라도, 맴버변수의 초기화는 맴버변수를 선언한 클래스가 책임지고 초기화하는 것이 맞기 때문이다. 그렇기 때문에 맴버변수는 name은 Man클래스에서 초기화해주는 것이 좋다.

이런 이유로 Man클래스와 BusinessMan 클래스 각각에 생성자를 만들어 맴버변수를 초기화하는 코드를 다음과 같이 작성하였다.

```
class Man {
  String name;

  public Man(String name){
    this.name = name;
  }

  public void tellName(){
    System.out.println("name is " + name);
  }
}
```

```
class BusinessMan extends Man {
  String company;

  public BusinessMan(String company){
    this.company = company;
  }

  public void tellCompany(){
    System.out.println("Company is " + company);
  }
}
```

Man 클래스에서 위와 같이 생성자를 추가하면 BusinessMan 클래스에서 오류가 발생한다.

그 이유에 대해 다음장에서 설명하겠다.

어떠한 클래스를 상속받는 자식클래스의 생성자 첫 줄에는 super()라는 명령어가 숨겨져 있다.

```
class Man {
  String name;

  public Man(String name){
    this.name = name;
  }
  ...
}
```

```
class BusinessMan extends Man {
  String company;

  public BusinessMan(String company){
    super();
    this.company = company;
  }
  ...
}
```

BusinessMan 클래스는 Man 클래스를 상속하고 있다.

이렇게 Man 클래스를 상속받는 BusinessMan 클래스의 생성자 첫 줄에서 super()라는 명령어가 숨져져 있다.

super() 명령어는 부모클래스의 생성자를 호출(실행)하는 명령어다.

()안의 내용이 없기 때문에 정확히는 부모클래스에서 매개변수 정보가 없는 생성자를 호출한다는 의미이다.

하지만 현재 Man 클래스에는 매개변수가 없는 생성자가 존재하지 않기 때문에 BusinessMan 클래스에서 오류가 발생하고 있는 것이다.

상속을 받고 있는 자식클래스 생성자의 첫 줄에는 부모클래스의 매개변수 정보가 없는 생성자를 호출하는 super()가 숨겨져 있다.

이렇게 자식클래스 생성자의 첫 줄에 숨어 있는 부모클래스의 생성자 호출 문법은 직접 사용하여 컨트롤 가능하다.

그리고 아래 코드가 Man클래스와 BusinessMan 클래스의 맴버변수를 적절하게 초기화한 문법이다.

```
class Man {
  String name;

  public Man(String name){
    this.name = name;
  }
  ...
}
```

```
class BusinessMan extends Man {
  String company;

  public BusinessMan(String company){
    super("kim");
    this.company = company;
  }
  ...
}
```

생성자 호출도 메서드 호출과 동일하게 매개변수 정보가 일치해야 한다.

BusinessMan 클래스의 생성자에 작성한 super("kim")은 부모클래스의 생성자 중 문자열 하나를 매개변수로 갖는 생성자를 호출하라는 말이다.

이렇게 명시적으로 부모클래스의 생성자를 하나 호출하면 super() 명령어가 실행되지 않는다.

단, 부모클래스의 생성자가 자식 클래스 생성자의 첫 줄에 숨겨져 있으므로, 명시적으로 사용할 때에도 반드시 첫 줄에 작성하여야 한다.

다음 예제의 실행결과를 생각해보자

```
class SuperCLS {
    public SuperCLS(){
        System.out.println(1);
    }
    public SuperCLS(int a){
        System.out.println(2);
    }
    public SuperCLS(int a, int b){
        System.out.println(3);
    }
}
```

```
class SubCLS extends SuperCLS {
    public BusinessMan(){
        System.out.println(4);
    }
     public BusinessMan(int a){
        super(1);
        System.out.println(5);
    }
    public BusinessMan(int a, int b){
        super(1, 2);
        System.out.println(6);
    }
}
```

```
public static void main(String[] args){
    SubCLS sub1 = new SubCLS();
    SubCLS sub2 = new SubCLS(5);
    SubCLS sub3 = new SubCLS(5, 10);
}
```

이러한 상속과 생성자에 대한 개념을 묻는 문제가 정보처리기사 실시 시험에 꽤나 자주 출제 된다.

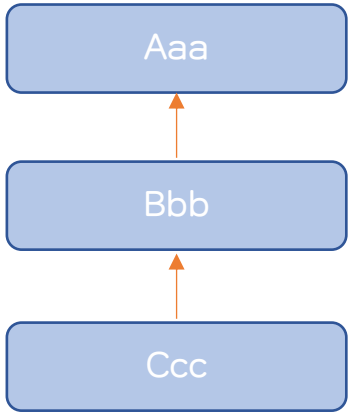
this()라는 키워드도 존재한다. super()가 부모클래스의 생성자 호출 명령이었다면, this()는 자신의 생성자를 호출하는 문법이다.

자바는 문법상으로는 단일 상속만 지원한다. 이 말은 문법상으로는 자식클래스는 하나의 클래스만 부모클래스로 갖는다는 말이다.

```
class Aaa {...}
class Bbb extends Aaa{...}
class Ccc extends Aaa{...}
class Ddd extends Aaa, Bbb{...} -> 이러한 문법은 존재하지 않음
```

하지만 문법상으로는 지원하지 않는 것이지, 코드를 잘 작성하면 다중 상속(여러 클래스의 내용 상속)이 가능하다.

```
class Aaa {...}
class Bbb extends Aaa{...}
class Ccc extends Bbb{...} -> 문법적으로는 Bbb 클래스 하나만을 상속하지만 해석을 해보면 Bbb, Aaa 두 클래스 모두를 상속하고 있다
```



상속을 사용하면 클래스 선언 시 다른 클래스의 맴버변수와 메서드를 아주 편하게 사용할 수 있다.

그렇다면 이런 편리한 상속을 모든 클래스, 아무 클래스에 막 사용하면 되는 것일까.

상속을 사용하는 적절한 경우는 두 클래스가 IS-A 관계를 가질 때 이다.

ex1> 모바일폰 클래스와 스마트폰 클래스

-> 모바일폰은 스마트폰이다 x

-> 스마트폰은 모바일폰이다 o

==> class SmartPhone extends MobilePhone{...}

ex2> 사람클래스와 학생클래스

-> 학생은 사람이다. o

-> 사람은 학생이다. x

==> class Student extends Person{...}

ex3> 모니터 클래스와 스마트폰클래스

-> 모니터는 스마트폰이다. x

-> 스마트폰은 모니터다. x

==> 상속 불가

다음과 같이 코드를 구현했을 때 실행 결과를 생각해보자.

```
class Chef {
    public void makeCook(){
        System.out.println("쉐프가 요리를 합니다");
    }
}

class MasterChef extends Chef{
    public void giveOrder(){
        System.out.println("후임 요리사에게 지시를 합니다.")
    }
    public void makeCook(){
        System.out.println("마스터 쉐프가 요리를 합니다");
    }
}
```

```
public static void main(String[] args){
    Chef c1 = new Chef();
    c1.makeCook();

    MasterChef c2 = new MasterChef();
    c2.makeCook();
}
```

c1객체에서 makeCook 메서드가 호출되면 Chef 클래스에서 선언한 메서드가 실행된다.

c2객체에서 makeCook 메서드가 호출되면 MasterChef 클래스에서 선언한 메서드가 실행된다.

언뜻보면 당연한 결과 같지만 중요한 개념이 숨겨져 있다.

Chef 클래스에는 makeCook()메서드가 존재하고, MasterChef 클래스는 Chef클래스를 상속하고 있기 때문에 MasterChef 클래스에는 이미 makeCook() 메서드가 존재하는 것과 마찬가지다. 그럼에도 불구하고 MasterChef클래스에서 makeCook()메서드를 정의하고 있다.

이렇게 상속을 통해 부모클래스로부터 물려받은 메서드의 기능을 새롭게 정의하는 것을 메서드 오버라이딩(Method Overriding)이라 한다.

```
class Chef {
    public void makeCook(){
        System.out.println("쉐프가 요리를 합니다");
    }
}
```

```
class MasterChef extends Chef{
    ...
    // Chef 클래스의 makeCook메서드 재정의
    public void makeCook(){
        System.out.println("마스터 쉐프가 요리를 합니다");
    }
}
```

부모클래스에서 물려받은 메서드를 자식 클래스에서 오버라이딩하면 메서드의 내용을 덮어 써버리게 때문에 자식클래스의 객체는 재정의한 메서드만 호출할 수 있다. 단, 자식 클래스의 객체에서는 재정의한 메서드 호출만 가능하지만 자식클래스를 정의할 때는 부모클래스에서 선언한 메서드를 사용할 수 있다.

자식 클래스 선언 시 부모클래스의 메서드를 사용하려면 메서드명 앞에 super. 키워드를 붙인다.

```
class Chef {
  public void makeCook(){
    System.out.println("쉐프가 요리를 합니다");
  }
}
```

```
class MasterChef extends Chef{
  ...
  public void makeCook(){
    System.out.println("마스터 쉐프가 요리를 합니다");
  }
  public void cookPractice(){
    super.makeCook(); -> Chef 클래스의 makeCook()
    this.makeCook(); -> 자신의 makeCook()
    makeCook(); -> 자신의 makeCook()
  }
}
```

```
public static void main(String[] args){
  Chef c1 = new Chef();
  c1.makeCook();

  MasterChef c2 = new MasterChef();
  //MasterChef의 makeCook() 호출!
  c2.makeCook();
}
```

	this	super	this()	super()
사용처 및 문법	변수명 앞에 사용 메서드명 앞에 사용 ex> this.name ex> this.tellName()	변수명 앞에 사용 메서드명 앞에 사용 ex> super.name ex> super.tellName()	생성자의 첫 줄에 사용 ex> this() ex> this(5)	생성자의 첫 줄에 사용 ex> super() ex> super("java")
의미	해당 클래스에서 선언한 맴버변수/메서드에 접근	부모클래스에서 선언한 맴버변수/메서드에 접근	해당 클래스의 생성자 호출	부모 클래스의 생성자 호출

그렇다면 메서드 오버라이딩 문법은 언제, 왜 사용할까? 메서드 오버라이딩 문법의 사용으로 얻는 이점은 무엇인가.

- 메서드 오버라이딩은 부모클래스에서 정의한 메서드를 재정의하기 때문에 메서드명과 매개변수를 동일하게 작성해야 한다.

  -> 동일한 메서드명을 작성하기 위해 부모클래스 파일을 열어 메서드명 및 매개변수 정보를 확인해야하는 번거로움 발생

- 메서드 오버라이딩을 적용하면 부모클래스의 메서드 내용을 자식 클래스에서 재정의(덮어써버림)하기 때문에 부모클래스의 메서드 호출 불가

  -> 부모클래스에서 정의한 메서드의 사용을 막아서 이점이 뭔가?

  자식클래스에서 다른 이름으로 메서드를 만들어 혹시 모를 사태에 대비하여 부모클래스에서 정의한 메서드를 사용할 수 있는 놔두는게 좋지 않을까?

그렇다. 우리가 지금까지 배운 내용만으로는 오버라이딩을 사용할 이유가 하나도 없다. 이제서야 말하지만 오버라이딩 개념은 단독으로 사용하라고 만든 개념이 아니다. 다음에 배우게되는 상속의 다형성(Polymorphism)이라는 개념과 함께 사용해야 진가를 발휘한다.

다형성(Polymorphism)이라는 개념도 마찬가지로 엄청 어려운 내용이지만 다형성의 개념을 단독으로 사용하면 어렵기만 하고, 아무런 이점이 없다.

대부분의 자바 서적에서는 오버라이딩과 다형성의 개념을 따로 설명한 후, 이 둘의 동시 적용으로 인한 시너지 효과를 설명하지 않는다.

그 결과,, 대부분의 자바 언어를 학습한 학습자는 상속을 이렇게 이야기한다.(솔직히 이렇게만 말할 수 있어도 대단한 거다!)

상속을 적용하면 부모클래스의 맴버변수와 메서드를 재차 만들지 않고 사용할 수 있기 때문에 코드의 재활용을 위해 상속을 사용한다.

다음에 나오는 다형성 개념과 오버라이딩을 잘 이해하면 상속을 왜 쓰는지에 대한 질문에 여러분들은 이렇게 대답할 것이다.

상속은 연관성이 없는 다수의 클래스에 공통의 연관성을 부여하여, 각기 다른 클래스(자료형) 데이터를 일괄적으로 관리하기 원한 문법이다.

다형성이란 형태가 다양하다는 의미이다.

자바에서의 다형성은 객체가 생성될 때 다양한 형태로 생성이 가능하다는 의미로 해석하면 된다. 다음 예시를 보자.

```
class MobliePhone {
    public void sendMsg(){
        System.out.println("메세지 전송");
    }
}

class SmartPhone extends MobilePhone {
    public void playApp(){
        System.out.println("앱 실행");
    }
}
```

```
public static void main(String[] args){
    //다양한 형태로 객체 생성 : 다형성
    SmartPhone p1 = new SmartPhone();
    MobilePhone p2 = new MobilePhone();
    MobilePhone p3 = new SmartPhone(); //처음보는 객체의 또 다른 생성 문법
    SmartPhone p4 = new MobilePhone(); //오류 발생!
}
```

위 코드처럼 객체은 우리가 알고 있던 형태 뿐 아니라 다른 형태로도 생성 가능하다.

이것을 다형성(Polymorphism)이라 한다.

주의할 것은 위 코드처럼 어떤 코드는 가능하고, 어떤 코드를 불가능하다.

다형성을 학습한다는 것은 이렇게 객체 생성 시 사용할 수 있는 문법과 아닌 문법을 구별하는 것과

우리가 알고 있는 객체 생성 문법과의 차이점, 그리고 이런 다형성을 오버라이딩과 함께

사용했을 때 어떠한 코드를 구현할 수 있는지 느껴보는 것이다.

결론적으로 말하면, 클래스에 대한 객체는 해당 클래스로 만들 수 있고, 추가적으로 부모클래스로도 생성 할 수 있다.

부모클래스는 모든 자식 클래스에 대한 객체를 생성할 수 있지만, 자식 클래스는 부모 클래스의 객체를 생성해주지 않는다.

```
public static void main(String[] args){
  SmartPhone p1 = new SmartPhone();
   MobilePhone p2 = new MobilePhone();

   //부모클래스로 자식클래스의 객체 생성 가능
   MobilePhone p3 = new SmartPhone();
  //자식클래스로는 부모클래스의 객체 생성 불가능
   SmartPhone p4 = new MobilePhone(); //오류 발생!
}
```

그렇다면, MobilePhone p3 = new SmartPhone(); 코드를 통해 생성된 p3 객체는 모바일폰 객체일까 스마트폰 객체일까?

MobilePhone p3 = new SmartPhone(); 코드를 통해 생성된 p3 객체는 스마트폰 클래스의 객체이다.

그럼 SmartPhone p1 = new SmartPhone(); 코드를 통해 생성된 p2 객체와의 차이점은 무엇일까.

p3 객체의 실체는 스마트폰 객체이지만, 객체 자신은 스스로 모바일폰 객체라 인식하고 있다.

그렇기 때문에 p3 객체는 스마트폰 클래스에서 선언한 맴버변수 및 메서드는 사용하지 못하고, 부모클래스인 모바일폰 클래스에서 선언한 맴버변수와 메서드만 사용 가능하다.

객체의 형 변환

MobilePhone p3 = new SmartPhone(); 코드로 만들어진 p3 객체의 실체는 스마트폰 객체지만, 객체 스스로는 모바일폰 객체라 인식하고 있다. 그렇기에 p3 객체가 스마트폰 객체라는 것을 인지시켜주면 p3 객체는 스마트폰의 맴버변수와 메서드를 사용할 수 있다. 그 방법은 정수와 실수의 형 변환 문법과 동일하다.

SmartPhone p = (SmartPhone)p3; 과 같이 코드를 작성하면 p객체는 스마트폰의 맴버변수와 메서드를 사용할 수 있다.

그럼 main메서드에 다음과 같은 코드를 작성하면 실행이 될까?

```
MobilePhone p = new MobilePhone();
SmartPhone pp = (SmartPhone)p;
```

p객체는 모바일폰 객체이기 때문에 억지로 SmartPhone객체로의 변환 시 오류가 발생하는 것을 확인 할 수 있을 것이다.

중간 정리

객체 생성 문법은 클래스명 객체명 = new 생성자호출() 이다.

하지만 부모클래스명 객체명 = new 자식클래스의 생성자호출() 문법으로도 자식 클래스에 대한 객체를 생성할 수 있다.

이렇게, 객체를 생성하는 형태가 다양하다는 의미에서 이러한 문법을 다형성이라 부른다.

다형성을 사용하면 부모클래스를 통해 자식클래스의 객체는 생성할 수 있지만, 자식클래스를 통해 부모클래스의 객체는 생성할 수 없다.

부모클래스를 통해 생성된 객체의 실체는 자식클래스에 대한 객체이지만, 객체 스스로는 부모클래스에 대한 객체라고 착각하고 있다.

그렇게 때문에 다형성 문법을 통해 생성된 객체는 부모클래스에서 선언한 맴버변수 및 메서드만 사용할 수 있다. 다형성을 통해 생성된 객체는 형 변환 문법을 통해 자료형 변경이 가능하고, 이렇게 변경된 객체는 자신의 클래스에 선언한 맴버변수 및 메서드 사용이 가능하다.

여기까지 학습한 내용으로 다형성을 왜 사용하는지, 언제 유용할지 감이 오는가? 다형성을 사용하면

부모클래스의 변수와 메서드만 사용 가능한데 이건 어려운 문법쓰고 오히려 손해아닌가.

오버라이딩 학습 마지막에도 언급했지만 다형성은 다형성 하나의 개념만으론 내용은 어려운데, 활용 방안이 별로 없는 개념이다.

다음과 같은 코드가 있을 때 main메서드에서 사용하지 못하는 코드는 몇 번 코드인가

```
class Cake {
    public void eatCake(){
        System.out.println("케익을 먹는다");
    }
}

class CheaseCake extends Cake{
    public void eatCheaseCake(){
        System.out.println("치즈 케익을 먹는다");
    }
}

class StrawberryCheaseCake extends CheaseCake{
    public void eatStrawberryCheaseCake(){
        System.out.println("딸기 치즈 케익을 먹는다");
    }
}
```

```
public static void main(String[] args){
    StrawberryCheaseCake cake1 = new StrawberryCheaseCake();
    cake1.eatCake(); // 1번
    cake1.eatCheaseCake(); //2번

    Cake cake2 = new StrawberryCheaseCake();  //3번
    CheaseCake cake3 = new StrawberryCheaseCake(); //4번
    CheaseCake cake4= new Cake(); //5번
    StrawberryCheaseCake cake5 = new CheaseCake(); //6번
    Cake cake6 = new CheaseCake(); //7번

    cake2.eatCheaseCake(); //8번
    cake2.eatCake(); //9번
    cake3.eatStrawberryCheaseCake(); //10번
    cake4.eatCheaseCake(); //11번
}
```

오버라이딩과 다형성을 문법적으로 따로 생각하면 문법은 어려운데 제약만 생기는 것을 앞선 학습을 통해 느꼈을 것이다.

다시 한 번 정리하자면

- 메서드 오버라이딩 문법을 적용하면 부모클래스의 메서드를 호출하지 못한다.
- 다형성을 통해 만들어진 객체는 자신의 맴버 변수와 메서드는 사용하지 못하고 부모클래스의 맴버변수와 메서드만 사용할 수 있다.

상속의 이해도에 따라 상속이라는 문법의 사용 목적을 다음과 같이 생각해 볼 수 있을 것이다.

1. 상속을 통해 부모클래스의 맴버 변수와 메서드를 사용할 수 있으므로, 중복된 코드를 제거하고 작성한 코드를 재활용 할 수 있다.
2. 상속은 연관성이 없는 다수의 클래스에 공통의 연관성을 부여하여, 각기 다른 클래스(자료형) 데이터를 일괄적으로 관리하기 원한 문법이다.

앞선 상속의 설명으로 1번 내용에 대해선 어느정도 동의할 것이라 생각이다. 지금부터는 2번 내용이 어떤 말을 하는 것인지 느껴볼 수 있도록 하겠다.

그러기 위해서 오버라이딩과 다형성을 함께 사용하는 코드를 작성해보겠다.

앞서 만든 Cake 클래스를 오버라이딩과 다형성 개념을 적용하지 않고 사용한 코드를 보겠다.

```
class Cake {
    public void eatCake(){
        System.out.println("케익을 먹는다");
    }
}

class CheaseCake extends Cake{
    public void eatCheaseCake(){
        System.out.println("치즈 케익을 먹는다");
    }
}

class StrawberryCheaseCake extends CheaseCake{
    public void eatStrawberryCheaseCake(){
        System.out.println("딸기 치즈 케익을 먹는다");
    }
}
```

```
public static void main(String[] args){
    Cake c1 = new Cake();
    CheaseCake c2 = new CheaseCake();
    StrawberryCheaseCake c3 = new StrawberryCheaseCake();

    c1.eatCake();
    c2.eatCheaseCake();
    c3.eatStrawberryCheaseCake();
}
```

c1, c2, c3 각 객체의 메서드 호출부분을 보면 다음 부분에서 아쉬움이 있다.

- 각 메서드명이 모두 다르기 때문에 코드 구현 시 모든 메서드 각각의 기능을 파악하고 있어야 한다.
- 각 메서드가 조금씩 다른 내용을 실행하지만 '먹는다'는 비슷한 내용인데 메서드를 따로따로 호출해야 한다. 뭔가 공통 부분이 있으면 좋겠다.

Cake, CheaseCake, StrawberryCheaseCake 클래스에 메서드 오버라이딩 적용

```
class Cake {
    public void eat(){
        System.out.println("케익을 먹는다");
    }
}

class CheaseCake extends Cake{
    public void eat(){
        System.out.println("치즈 케익을 먹는다");
    }
}

class StrawberryCheaseCake extends CheaseCake{
    public void eat(){
        System.out.println("딸기 치즈 케익을 먹는다");
    }
}
```

```
public static void main(String[] args){
    Cake c1 = new Cake();
    CheaseCake c2 = new CheaseCake();
    StrawberryCheaseCake c3 = new StrawberryCheaseCake();

    c1.eat();
    c2.eat();
    c3.eat();
}
```

메서드 오버라이딩 문법을 적용하여 모든 클래스의 메서드명을 동일하게 작성했다.

이 결과로 c1,c2,c3 객체의 메서드 호출 부분도 eat() 메서드 호출로 통일되었다.

메서드 오버라이딩을 통해

- 코드 구현 시 eat 메서드만 알고 있어면 된다.
- '먹는다'는 기능 실행을 위해 'eat()' 이라는 공통의 메서드를 사용한다.

하지만, 각 객체로부터 호출한 eat() 메서드는 모두 Cake 클래스의 메서드가 실행된다.

main 메서드에서 객체 생성 부분에 다형성 문법을 적용한다.

```
class Cake {
    public void eat(){
        System.out.println("케익을 먹는다");
    }
}

class CheaseCake extends Cake{
    public void eat(){
        System.out.println("치즈 케익을 먹는다");
    }
}

class StrawberryCheaseCake extends CheaseCake{
    public void eat(){
        System.out.println("딸기 치즈 케익을 먹는다");
    }
}
```

```
public static void main(String[] args){
    Cake c1 = new Cake();
    Cake c2 = new CheaseCake();
    Cake c3 = new StrawberryCheaseCake();

    c1.eat();    c2.eat();    c3.eat();
}
```

오버라이딩을 단독으로 사용 시 효과

- 부모, 자식 클래스에 공통된 메서드명을 적용할 수 있지만, 부모클래스의 메서드 호출 불가

다형성을 단독으로 사용 시 효과

- 각기 다른 클래스 자료형을 공통된 클래스(부모클래스) 형태로 생성할 수 있지만, 부모 클래스의 맴버변수와 메서드만 사용가능

오버라이딩 + 다형성 사용 시 효과

- 부모, 자식 클래스에 공통된 메서드명을 적용하고, 각기 다른 클래스 자료형을 공통된 클래스(부모클래스) 형태로 생성할 수 있다. 그리고 자식클래스의 메서드가 호출된다.

오버라이딩과 다형성을 함께 사용했을 때의 효과를 정리하면 다음과 같다,

1. ***각기 다른 클래스 자료형을 공통된 자료형(부모클래스) 형태로 생성할 수 있다. (다형성의 개념)***

   이 효과는 자바에서 엄청난 장점을 지닌다. 공부를 할수록 기본자료형과 단일 데이터의 사용률은 낮아지고, 참조자료형과 다수의 데이터를 다루는 경우가 훨씬 많음을 느낄 것이다. 자바에서 다수의 데이터를 관리하기 위해 사용하는 문법은 배열과 리스트이다. 그리고 배열과 리스트는 공통적으로 같은 자료형만 여러개 관리할 수 있다는 것을 이제는 알고 있다.

   즉, 상속의 문법과 다형성을 적용하면 각기 다른 자료형을 공통된 부모클래스 자료형으로 생성함으로써 하나의 자료형처럼 배열과 리스트에서도 사용이 가능해진다!

2. ***부모, 자식 클래스에 공통된 메서드명을 적용할 수 있다.(오버라이딩 개념)***

   메서드명을 통일함으로써 우리가 얻는 이점은 두 가지다.

   첫째, 알고 있어야 하는 메서드의 양이 줄어든다. 즉, 개발자가 신경 쓸 부분을 줄일 수 있다.

   둘째, 메서드명이 같다면 반복문에서 사용 가능하다. 반복문을 사용할 수 있다는 점을 1번 장점과 연결하면 엄청난 시너지 효과가 나온다.

3. ***다형성 문법으로 객체를 생성하면 부모클래스의 메서드만 실행되지만, 오버라이딩의 적용으로 자식클래스의 메서드 호출 가능(다형성 + 오버라이딩)***

   다형성을 단독으로 사용하면 자료형이 다른 클래스를 동일한 자료형으로 표현할 수 있지만 자신의 메서드는 사용 못하고, 부모 클래스에서 선언한 메서드만 사용할 수 있는 단점이 있었다. 하지만 오버라이딩을 함께 사용하면 부모클래스의 메서드 내용을 덮어써버리기 때문에 다형성에서의 단점을 상쇄할 수 있다.

인맥 관리 프로그램을 상속 미적용 버전과 상속 적용 버전으로 만들어보며 마지막으로 상속의 사용 목적에 대해 생각해보자.

프로그램 요구사항

인맥관리 프로그램은 대학동창과 직장동료의 정보를 관리할 수 있는 프로그램이다. 이 프로그램은 최대 10명의 인맥 정보를 관리할 수 있다.

인맥관리 프로그램은 지인의 정보를 저장할 수 있고, 저장된 모든 지인의 정보를 확인할 수도 있어야 한다.

```
class UnivFriend {    // 대학 동창
  private String name;
  private String major;
  private String phone;

  public UnivFriend(String name, String major, String phone) {
    this.name = name;
    this.major = major;
    this.phone = phone;
  }

  public void showUnivInfo() {
    System.out.println("이름: " + name);
    System.out.println("전공: " + major);
    System.out.println("전화: " + phone);
  }
}
```

```
class CompFriend {    // 직장 동료
  private String name;
  private String department;
  private String phone;

  public CompFriend(String name, String dept, String phone) {
    this.name = name;
    this.dept = dept;
    this.phone = phone;
  }

  public void showCompInfo() {
    System.out.println("이름: " + name);
    System.out.println("부서: " + dept);
    System.out.println("전화: " + phone);
  }
}
```

```
public static void main(String[] args) {
  UnivFriend[] ufs = new UnivFriend[5];
  int ucnt = 0;

  CompFriend[] cfs = new CompFriend[5];
  int ccnt = 0;

  ufs[ucnt++] = new UnivFriend("LEE", "Computer", "010-333-555");
  ufs[ucnt++] = new UnivFriend("SEO", "Electronics", "010-222-444");

  cfs[ccnt++] = new CompFriend("YOON", "R&D 1", "02-123-999");
  cfs[ccnt++] = new CompFriend("PARK", "R&D 2", "02-321-777");

  for(int i = 0; i < ucnt; i++) {
    ufs[i].showUnivInfo();
    System.out.println();
  }

  for(int i = 0; i < ccnt; i++) {
    cfrns[i].showCompInfo();
    System.out.println();
  }
}
```

이런 코드에서 만약 거래처 인맥도 관리해야 한다면??

```
class Friend {
  private String name;
  private String phone;

  public Friend(String name, String phone) {
    this.name = name;
    this.phone = phone;
  }
  public void showInfo() {
    System.out.println("이름: " + name);
    System.out.println("전화: " + phone);
  }
}
```

```
class CompFriend extends Friend {
  private String dept;

  public CompFriend(String name, String dept, String phone) {
    super(name, phone);
    this.dept = dept;
  }
  public void showInfo() {
    super.showInfo();
    System.out.println("부서: " + dept);
  }
}

class UnivFriend extends Friend {
  private String major;

  public UnivFriend(String name, String major, String phone) {
    super(name, phone);
    this.major = ma;jor
  }
  public void showInfo() {
    super.showInfo();
    System.out.println("전공: " + major);
  }
}
```

```
public static void main(String[] args) {
  Friend[] fs = new Friend[10];
  int cnt = 0;

  fs[cnt++] = new UnivFriend("LEE", "Computer", "010-333-555");
  fs[cnt++] = new UnivFriend("SEO", "Electronics", "010-222-444");
  fs[cnt++] = new CompFriend("YOON", "R&D 1", "02-123-999");
  fs[cnt++] = new CompFriend("PARK", "R&D 2", "02-321-777");

  // 모든 동창 및 동료의 정보 전체 출력
  for(int i = 0; i < cnt; i++) {
    fs[i].showInfo();     // 오버라이딩 한 메소드가 호출된다.
    System.out.println();
  }
}
```