



yonggyo1125 /
curriculum300H



<> Code

Issues 1

Pull requests

Actions

Projects

Security

Insights



main



curriculum300H / 1.JAVA(84시간) / 8~10일차(9h - 객체지향 프로그래밍2 /



yonggyo1981 객체지향프로그래밍2 동영상 강의 URL 추가

2 years ago



Name

Name

Last commit date



..



day08_10

객체지향 프로그래밍2

2 years ago



images

객체지향 프로그래밍2

2 years ago



README.md

객체지향프로그래밍2 동영...

2 years ago

README.md



강의 동영상 링크

[동영상 링크](#)

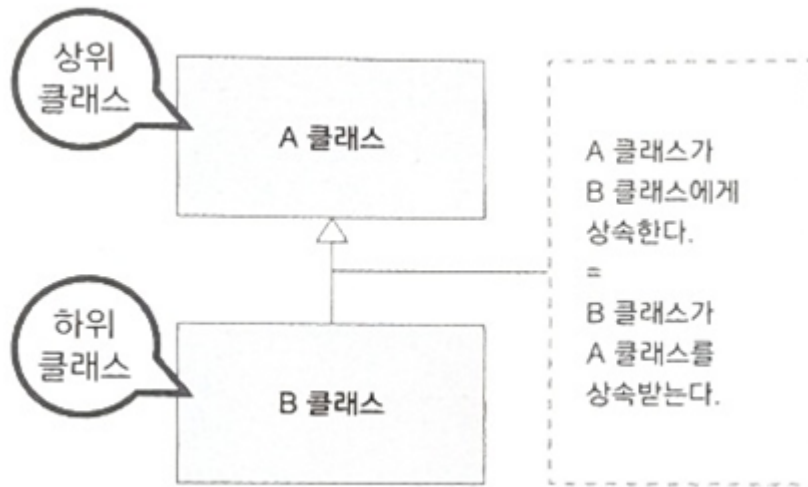
상속과 다형성

상속이란?

- 객체 지향 프로그래밍의 중요한 특징 중 하나가 **상속**(inheritance)입니다.
- 상속은 우리가 일반적으로 알 듯 무엇인가를 물려받는다는 의미 입니다. 일례로 부모가 자식에게 물려주는 재산을 상속이라고 하고, 상속받은 재산은 자신의 것으로 사용할 수 있습니다.
- 객체 지향 프로그램에서도 마찬가지로 B클래스가 A클래스를 상속받으면 B클래스는 A클래스를 상속받으면 B클래스는 A클래스의 메서드를 사용할 수 있습니다.
- 객체 지향 프로그램은 유지보수하기 편하고 프로그램을 수정하거나 새로운 내용을 추가하는 것이 유연한데, 그 기반이 되는 기술이 **상속**입니다.

클래스의 상속

- B 클래스가 A 클래스에서 상속 받는다고 할 때 다음과 같은 그림으로 나타낼 수 있습니다.



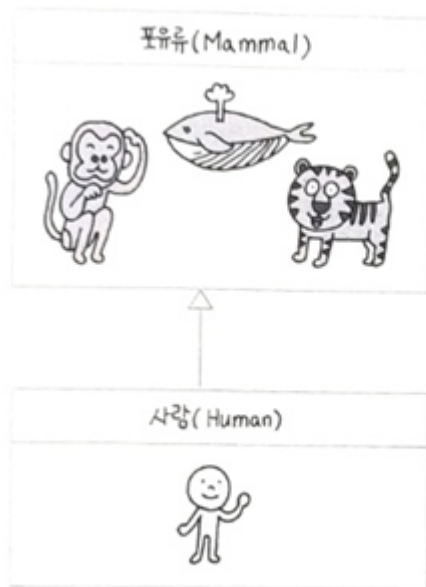
- 클래스 간 상속을 표현할 때는 위 그림에 표현한 것 처럼 상속받는 클래스에서 상속하는 클래스로 화살표가 갑니다.
- 부모 클래스(parent class)를 '상위클래스', 자식 클래스를 '하위 클래스' 등으로 부르기도 합니다.

클래스 상속 문법

- 자바 문법으로 **상속**을 구현할 때는 **extends** 예약어를 사용합니다.
- 이때 사용하는 extends 예약어는 **연장, 확장**하가의 의미입니다. 즉 A가 가지고 있는 속성이나 기능을 추가로 확장하여 B클래스를 구현한다는 뜻
- 그러면 일반적인 클래스 A에서 더 구체적인 클래스 B가 구현됩니다.
- 하기 코드는 **B클래스가 A 클래스를 상속받는다**라고 말합니다.

```
class B extends A {  
  
}
```





```

class Mammal {
    ...
}

class Human extends Mammal {
    ...
}
  
```

- 포유류는 사람보다 일반적인 개념입니다.
- 사람은 포유류의 특징과 기능을 기본으로 더 많거나 다른 특징과 기능을 가지고 있습니다.
- 상속관계에서는 상위 클래스가 하위클래스보다 일반적인 개념이고, 하위 클래스는 상위 클래스보다 구체적인 클래스가 됩니다.

상속을 사용하여 고객 관리 프로그램 구현하기

day08_10/inheritance/Customer.java

```

package day08_10.inheritance;

public class Customer {

    // 멤버 변수
    private int CustomerID; // 고객 아이디
    private String customerName; // 고객 이름
    private String customerGrade; // 고객 등급
    int bonusPoint; // 보너스 포인트
    double bonusRatio; // 적립비율

    // 디폴트 생성자
    public Customer() {
        customerGrade = "SILVER"; // 기본 등급
        bonusRatio = 0.01; // 보너스 포인트 기본 적립 비율
    }

    public int calcPrice(int price) {
        bonusPoint += price * bonusRatio; // 보너스 포인트 계산
        return price;
    }
}
  
```



```

        public String showCustomerInfo() {
            return customerName + "님의 등급은 " + customerGrade +
                "이며, 보너스 포인트는" + bonusPoint + "입니다.";
        }
    }
}

```

- 예제에서 사용한 멤버변수를 살펴보면 다음과 같습니다.

|멤버변수|설명| |customerID|고객 아이디| |customerName|고객 이름| |customerGrade|고객 등급

기본 생성자에서 지정되는 기본 등급은 SILVER입니다. |bonusPoint|고객의 보너스 포인트

- 고객이 제품을 구매할 경우 누적되는 보너스 포인트입니다. |bonusRatio|보너스 포인트 적립 비율

- 고객이 제품을 구매할 때 구매 금액의 일정 비율이 보너스 포인트로 적립됩니다.

- 기본 생성자에서 지정되는 적립 비율은 1%입니다. 즉, 10,000원짜리를 사면 100원이 적립됩니다.

- 모든 멤버 변수를 반드시 private으로 선언할 필요는 없습니다. 필요에 따라서 멤버 변수나 메서드를 외부에 노출하지 않을 목적일 때 private으로 선언합니다.

|메서드|설명| |Customer()|기본 생성자입니다. 고객 한 명이 새로 생성되면

CustomerGrade는 SILVER이고, bonusRatio는 1%로 지정합니다. |calcPrice(int price)|제품에 대해 지불해야 하는 금액을 계산하여 반환합니다. 할인되지 않는 경우 가격을 그대로 반환합니다. 그리고 가격에 대해 보너스 포인트 비율을 적용하여 보너스 포인트를 적립합니다. |showCustomerInfo()|고객 정보를 출력합니다. 고객 이름과 등급, 현재 적립된 포인트를 보여줍니다.

새로운 고객 등급이 필요한 경우

예제 시나리오



고객이 점점 늘어나고 판매도 많아지고 보니 단골 고객이 생겼습니다.

단골 고객은 회사 매출에 많은 기여를 하는 우수 고객입니다.

이 우수 고객에게 좋은 혜택을 주고 싶습니다.

우수 고객은 VIP이고, 다음과 같은 혜택을 제공합니다.

- 제품을 살 때는 항상 **10%** 할인해 줍니다.
- 보너스 포인트를 **5%** 적립해 줍니다.
- 담당 전문 상담원을 배정해 줍니다.

- Customer 클래스에 일반 곡개의 속성과 기능이 이미 구현되어 있기 때문에 VIPCustomer 클래스는 Customer 클래스를 상속받고 VIP고객에게 필요한 추가 속성과 기능을 구현하는 것입니다.

```
package day08_10.inheritance;
```



```
// VIPCustomer 클래스는 Customer 클래스를 상속받음
```

```
public class VIPCustomer extends Customer {  
    private int agentID; // VIP 고객 상담원 아이디  
    double saleRatio; // 할인율
```

```
    public VIPCustomer() {
```

```
        customerGrade = "VIP"; // 상위 클래스가 private 변수 이므로
```

오류 발생

```
        bonusRatio = 0.05;  
        saleRatio = 0.1;
```

```
    }
```

```
    public int getAgentID() {  
        return agentID;
```

```
    }
```

```
}
```

- 간단하게 상속을 통해서 Customer의 멤버변수와 메서드를 공유하는 VIPCustomer 클래스를 작성하였습니다.
- Customer 클래스에 이미 선언되어 있는 customerID, customerName, customerGrade, bonusPoint, bonusRatio 멤버 변수와 calcPrice(), showCustomerInfo() 메서드는 상속을 받아서 사용할 것이기 때문에 구현하지 않았습니다.
- 그러나 customerGrade 변수에 오류가 발생합니다. 상위 클래스에서 customerGrade 는 private 변수이고 외부 클래스에서는 이 변수를 사용할 수 없습니다.

상위 클래스 변수를 사용하기 위한 protected 예약어

- 상위 클래스에 선언한 customerGrade가 private 변수이기 때문에 오류가 발생합니다.
- 상위 클래스에 작성한 변수나 메서드 중 외부 클래스에서 사용할 수 없지만 하위 클래스에서 사용할 수 있도록 지정하는 예약어가 protected 입니다. protected로 변경을 하면 하위 클래스에서는 접근할 수 있게 됩니다.
- 즉, protected는 상속된 하위 클래스를 제외한 나머지 외부 클래스에서는 private과 동일한 역할을 합니다.

day08_10/inheritance/Customer.java

```
package day08_10.inheritance;
```



```
public class Customer {
```

```
    // 멤버 변수
```

```

protected int CustomerID; // 고객 아이디
protected String customerName; // 고객 이름
protected String customerGrade; // 고객 등급
int bonusPoint; // 보너스 포인트
double bonusRatio; // 적립비율

// 디폴트 생성자
public Customer() {
    customerGrade = "SILVER"; // 기본 등급
    bonusRatio = 0.01; // 보너스 포인트 기본 적립 비율
}

public int calcPrice(int price) {
    bonusPoint += price * bonusRatio; // 보너스 포인트 계산
    return price;
}

public String showCustomerInfo() {
    return customerName + " 님의 등급은 " + customerGrade +
"이며, 보너스 포인트는" + bonusPoint + "입니다.";
}

public int getCustomerID() {
    return CustomerID;
}

public void setCustomerID(int customerID) {
    CustomerID = customerID;
}

public String getCustomerName() {
    return customerName;
}

public void setCustomerName(String customerName) {
    this.customerName = customerName;
}

public String getCustomerGrade() {
    return customerGrade;
}

public void setCustomerGrade(String customerGrade) {
    this.customerGrade = customerGrade;
}
}

```

- Customer클래스에 있는 private 변수를 다른 하위 클래스에서도 사용할 수 있도록 모두 protected로 변경하였습니다.
- protected로 선언한 customerID, customerName, customerGrade 변수를 사용하기 위해 get(), set()메서드를 추가 하였습니다.

- protected 예약어로 선언한 변수는 외부 클래스 private 변수처럼 get() 메서드를 사용해 값을 가져올 수 있고, set() 메서드를 사용해 값을 지정할 수 있습니다.
- Customer클래스를 상속받은 VIPCustomer 클래스는 protected로 선언한 변수를 상속받게 되고, 나머지 public 메서드도 상속받아 사용할 수 있습니다. 상기 코드와 같이 protected로 선언하면 VIPCustomer 부분의 오류는 사라집니다.

day08_10/inheritance/CustomerTest1.java

```
package day08_10.inheritance;

public class CustomerTest1 {
    public static void main(String[] args) {
        Customer customerLee = new Customer();
        // CustomerID와 customerName은 protected 변수이므로 set()

        customerLee.setCustomerID(10010);
        customerLee.setCustomerName("이순신");
        customerLee.bonusPoint = 1000;
        System.out.println(customerLee.showCustomerInfo());

        VIPCustomer customerKim = new VIPCustomer();
        // CustomerID와 customerName은 protected 변수이므로 set()

        customerKim.setCustomerID(10020);
        customerKim.setCustomerName("김유신");
        customerKim.bonusPoint = 10000;
        System.out.println(customerKim.showCustomerInfo());
    }
}
```

실행결과

이순신 님의 등급은 SILVER이며, 보너스 포인트는1000입니다.
 김유신 님의 등급은 VIP이며, 보너스 포인트는10000입니다.

상속에서 클래스 생성과 형 변환

- 하위 클래스가 생성될 때는 상위 클래스의 생성자가 먼저 호출됩니다.
- 상속관계에서 클래스의 생성과정을 살펴보면 하위클래스가 상위클래스의 변수와 메서드를 사용할 수 있는 이유와 하위클래스가 상위클래스의 자료형으로 형 변환을 할 수 있는 이유를 이해할 수 있습니다.

하위 클래스가 생성되는 과정

- 상속을 받은 하위 클래스는 상위클래스의 변수와 메서드를 사용할 수 있습니다.

- 즉, CustomerTest예제를 살펴보면 VIPCustomer 클래스로 선언한 customerKim 인스턴스는 상속받은 상위 클래스의 변수를 자기 것 처럼 사용할 수 있습니다.
- 변수를 사용할 수 있다는 것은 그 변수를 저장하고 있는 메모리가 존재한다는 뜻입니다.
- 그런데 VIPCustomer 클래스의 코드를 보면 해당 변수가 존재하지 않습니다. 단순히 Customer 클래스를 상속받았을 뿐 입니다.
- 여기에서 상속된 하위 클래스가 생성되는 과정을 다시 생각해 볼 필요가 있습니다.
- 테스트를 하기 위해 Customer와 VIPCustomer 클래스 생성자에 출력문을 추가하겠습니다.

day08_10/inheritance/Customer.java

```
package day08_10.inheritance;
```



```
public class Customer {
```

```
    // 멤버 변수
```

```
    protected int CustomerID; // 고객 아이디
```

```
    protected String customerName; // 고객 이름
```

```
    protected String customerGrade; // 고객 등급
```

```
    int bonusPoint; // 보너스 포인트
```

```
    double bonusRatio; // 적립비율
```

```
    // 디폴트 생성자
```

```
    public Customer() {
```

```
        customerGrade = "SILVER"; // 기본 등급
```

```
        bonusRatio = 0.01; // 보너스 포인트 기본 적립 비율
```

```
        // 상위 클래스 생성할 때 콘솔 출력문
```

```
        System.out.println("Customer() 생성자 호출");
```

```
    }
```

```
    public int calcPrice(int price) {
```

```
        bonusPoint += price * bonusRatio; // 보너스 포인트 계산
```

```
        return price;
```

```
    }
```

```
    public String showCustomerInfo() {
```

```
        return customerName + " 님의 등급은 " + customerGrade +
```

```
"이며, 보너스 포인트는" + bonusPoint + "입니다.";
```

```
    }
```

```
    public int getCustomerID() {
```

```
        return CustomerID;
```

```
    }
```

```
    public void setCustomerID(int customerID) {
```

```
        CustomerID = customerID;
```

```
    }
```



```

    public String getCustomerName() {
        return customerName;
    }

    public void setCustomerName(String customerName) {
        this.customerName = customerName;
    }

    public String getCustomerGrade() {
        return customerGrade;
    }

    public void setCustomerGrade(String customerGrade) {
        this.customerGrade = customerGrade;
    }
}

```

day08_10/inheritance/VIPCustomer.java

```

package day08_10.inheritance;

// VIPCustomer 클래스는 Customer 클래스를 상속받음
public class VIPCustomer extends Customer {
    private int agentID; // VIP 고객 상담원 아이디
    double saleRatio; // 할인율

    public VIPCustomer() {
        customerGrade = "VIP"; // 상위 클래스가 private변수 이므로
        bonusRatio = 0.05;
        saleRatio = 0.1;

        // 하위 클래스를 생성할 때 콘솔 출력문
        System.out.println("VIPCustomer() 생성자 호출");
    }

    public int getAgentID() {
        return agentID;
    }
}

```

오류 발생

day08_10/inheritance/CustomerTest2.java

```

package day08_10.inheritance;

public class CustomerTest2 {
    public static void main(String[] args) {

```

```

VIPCustomer customerKim = new VIPCustomer(); // 하위 클래스 생성

customerKim.setCustomerID(1020);
customerKim.setCustomerName("김유신");
customerKim.bonusPoint = 10000;
System.out.println(customerKim.showCustomerInfo());
}
}

```

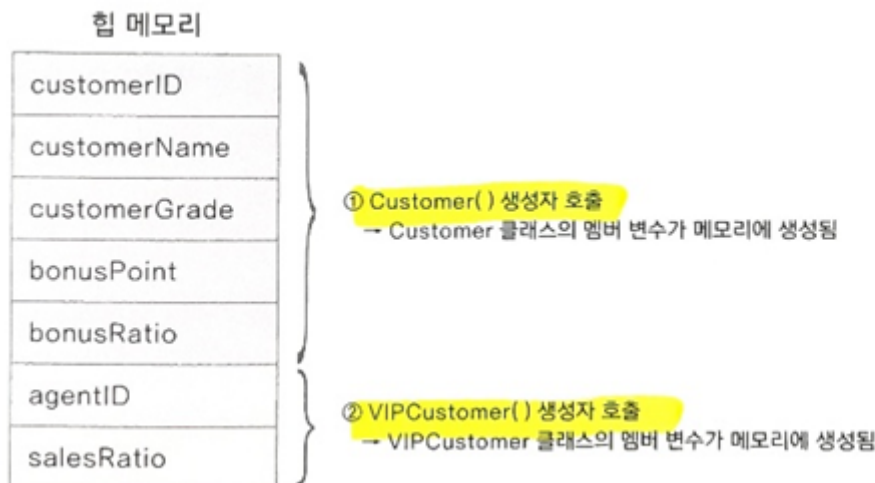
실행결과

Customer() 생성자 호출

VIPCustomer() 생성자 호출

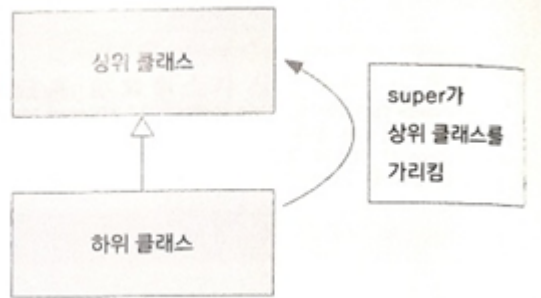
김유신 님의 등급은 VIP이며, 보너스 포인트는 10000입니다.

- 상위클래스의 Customer() 생성자가 먼저 호출되고 그 다음에 VIPCustomer()가 호출되는 것을 알 수 있습니다.
- 상위 클래스를 상속받은 하위 클래스가 생성될 때는 반드시 상위 클래스의 생성자가 먼저 호출됩니다. 그리고 상위클래스의 생성자가 호출될 때 상위 클래스의 멤버 변수가 메모리에 생성되는 것입니다.



- 상위 클래스의 변수가 메모리에서 먼저 생성이 되기 때문에 하위 클래스에서도 이 값들을 모두 사용할 수 있습니다.
- private 변수도 동일하게 상위클래스에서 생성이 되지만, 단지 하위 클래스에서 접근할 수 없다는 점을 제외하고는 동일합니다.

부모를 부르는 예약어 super



- `super`예약어는 하위클래스에서 상위 클래스로 접근할 때 사용합니다.
- 하위 클래스는 **상위클래스의 주소**, 즉 참조 값을 알고 있습니다. 이 참조 값을 가지고 있는 예약어가 바로 `super`입니다.
- `this`가 자기 자신의 참조 값을 가지고 있다는 것과 같다고 생각하면 됩니다.
- 또한 `super`는 **상위 클래스의 생성자를 호출**하는 데도 사용합니다.

상위 클래스 생성자 호출하기

- `CustomerTest2.java`의 예제에서 `VIPCustomer`만 생성 하였는데, `Customer` 상위 클래스도 생성된 것을 알 수 있습니다.
- 하위 클래스 생성자만 호출했는데 상위 클래스의 생성자가 호출되는 이유는 ****하위 클래스 생성자에서 ** `super()`를 자동으로 호출하기 때문**입니다.
- `super()`를 호출하면 **상위 클래스의 디폴트 생성자가 호출**됩니다.

```

public VIPCustomer()
    super(); // 컴파일러가 자동으로 추가하는 코드(상위클래스의
    Customer()가 호출됨)
    CustomerGrade = "VIP";
    bonusRatio = 0.05;
    saleRatio = 0.1;

    System.out.println("VIPCustomer() 생성자 호출");
}
  
```



`super` 예약어로 매개변수가 있는 생성자 호출하기

- `Customer` 클래스를 생성할 때 고객 ID와 이름을 반드시 지정해야 한다고 합니다. 이런 경우에 `set()` 메서드로 값을 지정하는 것이 아니고, 새로운 생성자를 만들어서 매개변수로 값을 전달받을 수도 있습니다.
- 즉, 디폴트 생성자가 아닌 매개변수가 있는 생성자를 직접 구현해야 합니다.
- 다음과 같이 `Customer` 클래스에 새로운 생성자를 추가하고, 기존의 디폴트 생성자는 삭제하거나 주석 처리해 보겠습니다.

day08_10/inheritance/Customer.java



```
...
// 디폴트 생성자
/**
public Customer() {
    customerGrade = "SILVER"; // 기본 등급
    bonusRatio = 0.01; // 보너스 포인트 기본 적립 비율

    // 상위 클래스 생성할 때 콘솔 출력문
    System.out.println("Customer() 생성자 호출");
}
*/

public Customer(int customerID, String customerName) {
    this.customerID = customerID;
    this.customerName = customerName;
    customerGrade = "SILVER";
    bonusRatio = 0.01;
    System.out.println("Customer(int, String) 생성자 호출");
}
...

```

- 그런데 이렇게 Customer 클래스의 디폴트 생성자를 없애고 새로운 생성자를 작성하면, Customer 클래스를 상속받은 VIPCustomer 클래스에서 오류가 발생합니다.

```

2
3 // VIPCustomer 클래스는 Customer 클래스를 상속받음
4 public class VIPCustomer extends Customer {
5     private int agentID; // VIP 고객 상담원 아이디
6     double saleRatio; // 할인율
7
8     public VIPCustomer() {
9         customerGrade = "VIP";
10        bonusRatio = 0.05;
11        saleRatio = 0.1;
12
13        // 하위 클래스를 생성할 때 콘솔 출력문
14        System.out.println("VIPCustomer() 생성자 호출");
15    }
16

```

```

3 // VIPCustomer 클래스는 Customer 클래스를 상속받음
4 public class VIPCustomer extends Customer {
5     private int agentID; // VIP 고객 상담원 아이디
6     double saleRatio; // 할인율
7
8     public VIPCustomer() {
9         customerGrade = "VIP";
10        bonusRatio = 0.05;
11        saleRatio = 0.1;
12
13        // 하위 클래스를 생성할 때 콘솔 출력문
14        System.out.println("VIPCustomer() 생성자 호출");
15    }

```

Implicit super constructor Customer() is undefined. Must explicitly invoke another constructor

- 이 오류 메시지는 묵시적으로 호출될 디폴트 생성자 Customer()가 정의되지 않았기 때문에, 반드시 명시적으로 다른 생성자를 호출해야 한다는 뜻입니다.

하위 클래스가 생성될 때는 상위 클래스의 디폴트 생성자를 호출하는 super()가 자동으로 생성됩니다.

```
public Customer(int customerID, String customerName) {
```



```

    super();
    -> 컴파일시에 super()가 자동생성되고 상위클래스의 디폴트 생성자
    Customer()가 호출되는데
    -> Customer 클래스에는 디폴트 생성자 Customer()가 없으므로 오류가
    발생합니다.
    ...
}

```

- Customer 클래스를 새로 생성할 때 고객 ID와 고객 이름을 반드시 지정하여 생성하기로 했으니 VIPCustomer 클래스를 생성할 때도 이 값이 필요합니다.
- 그리고 VIP 고객만을 위한 상담원 ID도 지정합니다.
- 기존 VIPCustomer 클래스의 디폴트 생성자도 지우거나 주석처리한 후 필요한 매개 변수를 포함하는 생성자를 새로 작성합니다.

day08_10/inheritance/VIPCustomer.java

```

...
public VIPCustomer(int customerID, String customerName, int agentID) {
    super(customerID, customerName);
    customerGrade = "VIP";
    bonusRatio = 0.05;
    saleRatio = 0.1;

    this.agentID = agentID;

    // 하위 클래스를 생성할 때 콘솔 출력문

```



```

        System.out.println("VIPCustomer() 생성자 호출");
    }
    ...

```

- 새로운 생성자는 고객 ID, 고객 이름, 상담원 ID를 매개변수로 받습니다.
- `super` 예약어는 상위클래스의 생성자를 호출하는 역할을 하며, 3행의 `super(customerID, customerName);` 문장으로 상위 클래스 생성자를 호출합니다.
- `super()`를 통해 `Customer(int customerID, String customerName)` 상위 클래스 생성자를 호출하고 코드 순서대로 멤버 변수가 초기화 됩니다.
- 상위 클래스 생성자 호출이 끝나면 `VIPCustomer` 하위 클래스 생성자 내부 코드 수행이 마무리됩니다.

day08_10/inheritance/CustomerTest2.java

```

package day08_10.inheritance;

public class CustomerTest2 {
    public static void main(String[] args) {
        VIPCustomer customerKim = new VIPCustomer(1020, "김유신",
1000);
        customerKim.bonusPoint = 10000;
        System.out.println(customerKim.showCustomerInfo());
    }
}

```

실행결과

```

Customer(int, String) 생성자 호출
VIPCustomer() 생성자 호출
김유신 님의 등급은 VIP이며, 보너스 포인트는10000입니다

```

- VIP 등급인 김유신 고객을 생성할 때는 상위클래스 생성자를 먼저 호출한 후 하위 클래스 생성자 코드 수행이 정상적으로 마무리 되는 것을 알 수 있습니다.

상위 클래스의 멤버 변수나 메서드를 참조하는 `super`

- 상위 클래스에 선언된 멤버 변수나 메서드를 하위 클래스에서 참조할 때도 `super`를 사용합니다.
- `this`를 사용하여 자신의 멤버에 접근했던 것과 비슷합니다.
- 예를 들어 `VIPCustomer` 클래스의 `showVIPInfo()` 메서드에 상위 클래스의 `showCustomerInfo()` 메서드를 참조해 담당 상담원 아이디를 추가로 출력하고자 할 때 다음과 같이 구현할 수 있습니다.

```
public String showVIPInfo() {  
    return super.showCustomerInfo() + "담당 상담원 아이디는 " +  
    agentID + "입니다.";  
}
```



- super 예약어는 상위 클래스의 참조 값을 가지고 있으므로 위 코드 처럼 사용하면 고객 정보를 출력하는 showCustomerInfo() 메서드를 새로 구현하지 않고 상위 클래스의 구현 내용을 활용할 수 있습니다.
- 물론 위 코드의 showVIPInfo() 메서드에서는 굳이 show.showCustomerInfo()라고 호출하지 않고 그냥 showCustomerInfo()라고 호출해도 됩니다.
- 메서드 재정의에서 자세하게 설명을 하겠지만 하위클래스가 상위클래스와 동일한 이름의 메서드를 구현하는 경우도 있습니다. 이러한 경우 하위 클래스에서 동일한 이름의 상위 클래스 메서드를 가리킬 때 super.showCustomerInfo()라고 써야 합니다.

상위 클래스로 묵시적 클래스 형 변환

- 상속을 공부하면서 이해해야 하는 중요한 관계가 클래스 간의 형 변환입니다.
- Customer와 VIPCustomer의 관계를 생각해 보면, 개념 면에서 보면 상위 클래스인 Customer가 VIPCustomer보다 일반적인 개념이고, 기능면에서 보면 VIPCustomer가 Customer보다 기능이 더 많습니다. 왜냐하면 상속받은 클래스는 상위 클래스의 기능을 모두 사용할 수 있고 추가로 많은 기능을 구현하기 때문입니다.
- 따라서 VIPCustomer는 VIPCustomer형 이면서 동시에 Customer 형이기도 합니다.
- 즉, VIPCustomer 클래스로 인스턴스를 생성할 때 이 인스턴스의 자료형을 Customer 형으로 클래스 형 변환하여 선언할 수 있습니다. 왜냐하면 VIPCustomer 클래스는 Customer 클래스를 상속받았기 때문입니다.

클래스형과 클래스의 자료형, 인스턴스형과 인스턴스의 자료형은 모두 비슷한 의미로 사용하는 용어입니다. 이러한 **클래스 형 변환**을 **업캐스팅**(upcasting)이라고도 합니다.

```
Customer vc = new VIPCustomer();
```

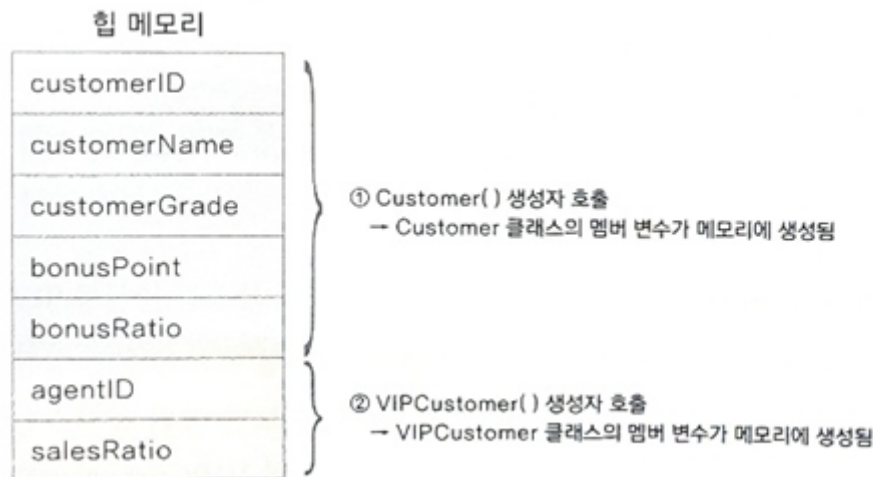


Customer - 선언된 클래스형 (상위 클래스형)

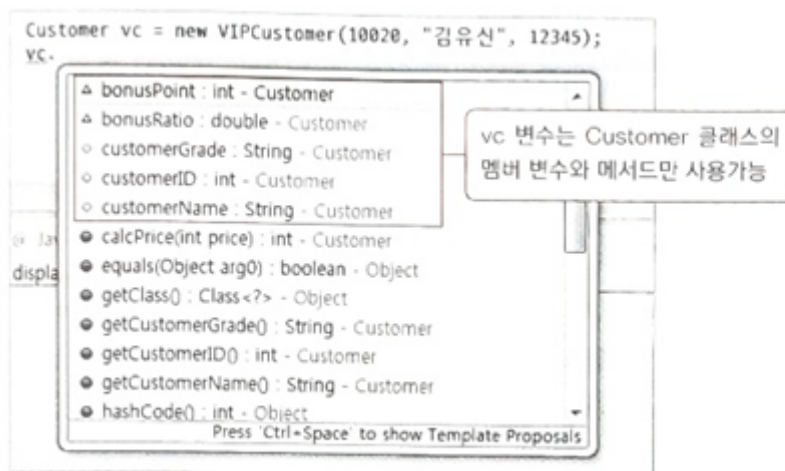
VIPCustomer - 생성된 인스턴스의 클래스형 (하위 클래스 형)

- 반대로 Customer로 인스턴스를 생성할 때 VIPCustomer형으로 선언할 수는 없습니다. 상위 클래스인 Customer가 VIPCustomer 클래스의 기능을 다 가지고 있는 것은 아니기 때문입니다. 요약하면 모든 하위 클래스는 상위 클래스 자료형으로 형변환 될 수 있지만 그 역은 성립하지 않습니다.

형 변환된 vc가 가리키는 것



- Customer vc = new VIPCustomer(); 문장이 실행되면 VIPCustomer 생성자가 호출되므로 클래스 변수가 위와 같이 메모리에 만들어 집니다.
- 그런데 클래스의 자료형이 Customer로 한정되었습니다. 클래스가 형 변환이 되었을 때는 선언한 클래스형에 기반하여 멤버 변수와 메서드에 접근할 수 있습니다.
- 따라서 이 vc 참조 변수가 가리킬 수 있는 변수와 메서드는 Customer 클래스의 멤버 뿐입니다.



- 하위 클래스의 인스턴스가 상위 클래스로 형 변환되는 과정이 묵시적으로 이루어진다. - [다형성 참조](#)

메서드 오버라이딩(재정의)

- 상위 클래스에 정의한 메서드가 하위 클래스에서 구현할 내용과 맞지 않을 경우 하위 클래스에서 이 메서드를 재정의할 수 있습니다.
- 이를 메서드 오버라이딩(method overriding)이라고 합니다.
- **오버라이딩을 하려면 반환형, 메서드 이름, 매개 변수, 매개변수 자료형이 반드시 같아야 합니다.**
- 그렇지 않다면 재정의한 메서드를 기존 메서드와 다른 메서드로 인식합니다.

VIP 고객 클래스의 제품 가격 메서드 재정의하기

- VIPCustomer 클래스에서 calcPrice() 메서드를 재정의해 봅시다.

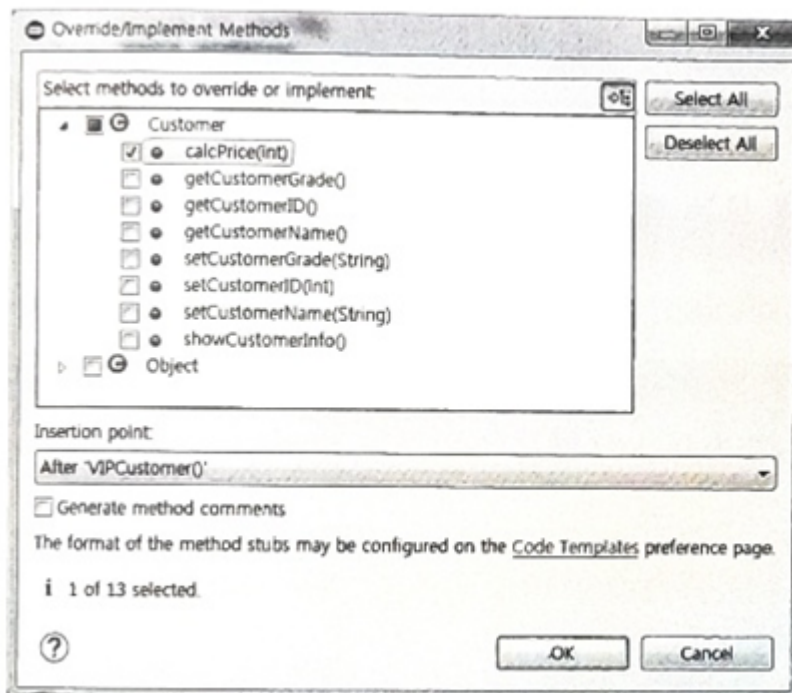
day08_10/inheritance/VIPCustomer.java

```
...
@Override
public int calcPrice(int price) {
    bonusPoint += price * bonusRatio;

    return price - (int)(price * saleRatio); // 할인된 가격을 계산하여
반환
}
...
```



- 하위클래스 VIPCustomer에서 calcPrice() 메서드를 재정의했습니다.
- 상위 클래스의 calcPrice() 메서드와 매개변수의 자료형 및 개수가 같고, 반환형도 int형으로 같습니다. 다만 할인율을 계산하여 정가에서 뺀 후 세일가격을 반환하도록 코드가 변경되었습니다.
- 상위 클래스의 메서드를 재정의 할때는 메서드 이름을 직접 써도 되고, 이클립스의 기능을 활용할 수 있습니다.
- 코드에서 오른쪽 마우스 버튼을 누르고 [Source -> Override/Implement Methods...] 을 누르면 다음과 같은 화면이 나옵니다.



- 상위 클래스 Customer의 메서드 중에서 재정의 할 메서드를 선택할 수 있습니다.

애너테이션(Annotation)

애너테이션은 영어로는 주석이라는 의미 입니다. @기호와 함께 사용하며, '@애너테이션 이름'으로 표현합니다. 자바에서 제공하는 **애너테이션은 컴파일러에게 특정한 정보를 제공해 주는 역할을 합니다.** 예를 들어 **@Override는 이 메서드가 재정의된 메서드임을 컴파일러에게 알려줍니다.** 만약 메서드의 선언부가 다르다면 컴파일된 오류가 발생하여 프로그래머의 실수를 막아 줍니다. 이렇게 미리 정의되어 있는 애너테이션을 표준 애너테이션이라고 합니다.

주로 사용하는 표준애너테이션

애노테이션	설명
@Override	재정의된 메서드라는 정보 제공
@FunctionalInterface	함수형 인터페이스라는 정보 제공
@Deprecated	이후 버전에서 사용되지 않을 수 있는 변수, 메서드에 사용됨
@SuppressWarnings	특정 경고가 나타나지 않도록 함

day08_10/inheritance/OverrideTest1.java

```
package day08_10.inheritance;

public class OverrideTest1 {

    public static void main(String[] args) {
        Customer customerLee = new Customer(10010, "이순신");
        customerLee.bonusPoint = 1000;

        VIPCustomer customerKim = new VIPCustomer(10020, "김유신", 12345);
        customerKim.bonusPoint = 10000;

        int price = 10000;
        System.out.println(customerLee.getCustomerName() + " 님이  
지불해야 하는 금액은 " + customerLee.calcPrice(price) + "원 입니다.");
        System.out.println(customerKim.getCustomerName() + " 님이  
지불해야 하는 금액은 " + customerKim.calcPrice(price) + "원 입니다.");
    }
}
```

실행결과

이순신 님이 지불해야 하는 금액은 10000원 입니다.
김유신 님이 지불해야 하는 금액은 9000원 입니다.



묵시적 형변환과 메서드 재정의

```
Customer vc = new VIPCustomer("10030", "나몰라", 2000);  
vc.calcPrice(10000);
```



- 묵시적 형 변환에 의해 VIPCustomer가 Customer형으로 변환되었습니다.
- 그리고 나서 calcPrice() 메서드가 호출되었습니다.
- calcPrice()는 하위 클래스에서 재정의된 메서드이며 Customer 클래스와 VIPCustomer 클래스에 모두 존재합니다.
- Customer형으로 선언되었다고 하더라도 vc.calcPrice(10000)은 VIPCustomer에서 재정의된 메서드가 호출됩니다.
(멤버 변수와 메서드는 선언한 클래스형에 따라 호출됩니다.)
- 상위 클래스와 하위 클래스에 같은 이름의 메서드가 존재할 때 호출되는 메서드는 인스턴스에 따라 결정됩니다.
- 선언한 클래스형이 아닌 생성된 인스턴스의 메서드를 호출하는 것, 이렇게 메서드가 호출되는 기술을 '가상 메서드(virtual method)'라 합니다.

가상 메서드

- 자바의 클래스는 멤버변수와 메서드로 이루어져 있습니다. 클래스를 생성하여 인스턴스가 만들어 지면 멤버변수는 힙 메모리에 위치합니다.
- 그러나 변수가 사용하는 메모리와 메서드가 사용하는 메모리는 다릅니다.
- 변수는 인스턴스가 생성될 때마다 생성되지만 실행해야 할 명령 집합이기 때문에 인스턴스가 달라도 같은 로직을 수행합니다.
- 즉, 같은 객체의 인스턴스를 여러 개 생성한다고 해서 메서드도 여러 개 생성되지 않습니다.

day08_10/virtualfunction/TestA.java

```
package day08_10.virtualfunction;  
  
public class TestA {  
    int num;  
  
    void aaa() {  
        System.out.println("aaa() 출력");  
    }  
  
    public static void main(String[] args) {  
        TestA a1 = new TestA();  
        a1.aaa();  
    }  
}
```



```

        TestA a2 = new TestA();
        a2.aaa();
    }
}

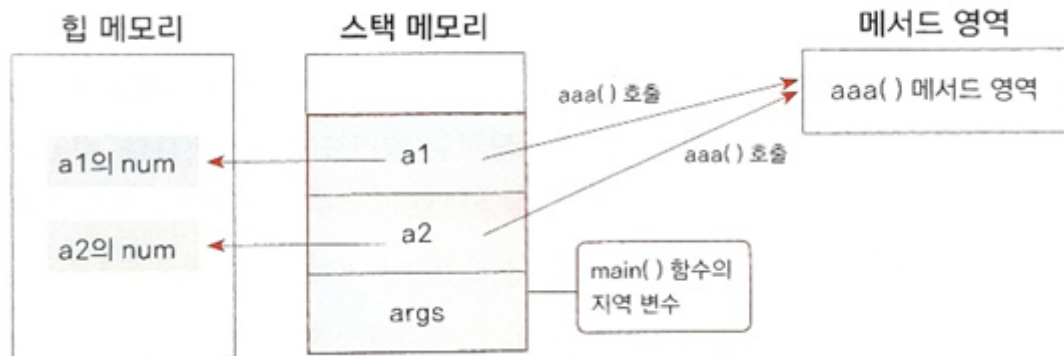
```

실행결과

aaa() 출력

aaa() 출력

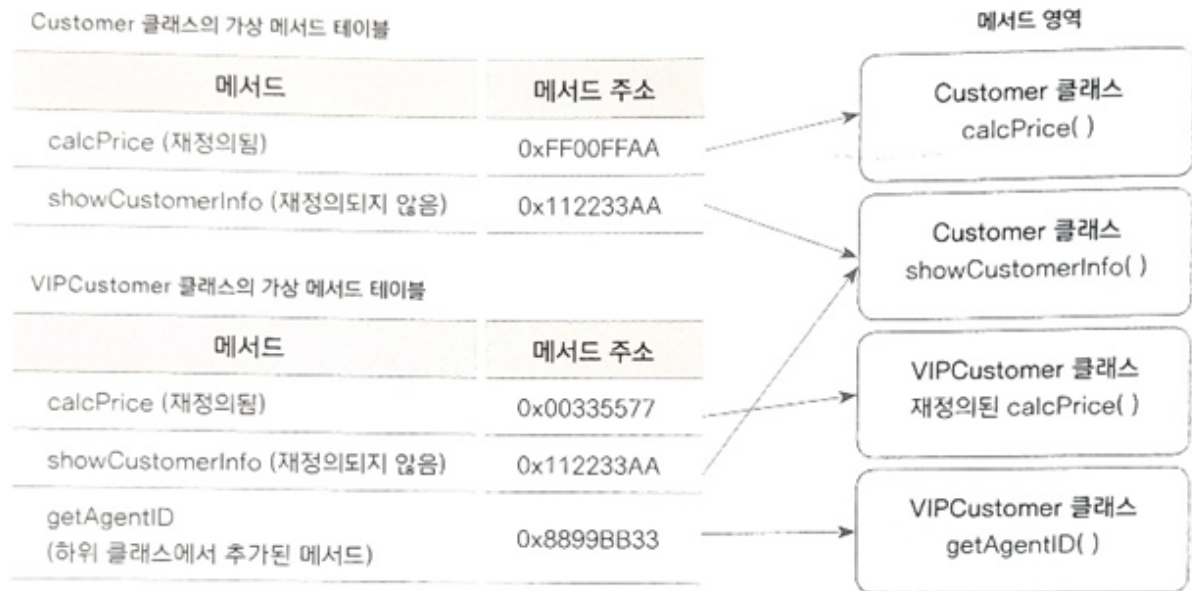
- 상기 코드가 실행되는 메모리 상태를 그림으로 그리면 다음과 같습니다.



- main() 함수가 실행되면 지역 변수는 스택 메모리에 위치합니다.
- 각 참조 변수 a1, a2가 가리키는 인스턴스는 힙 메모리에 생성됩니다.
- 메서드의 명령 집합은 메서드 영역(코드 영역)에 위치합니다.
- 우리가 메서드를 호출하면 메서드 영역의 주소를 참조하여 명령이 실행됩니다. 따라서 인스턴스가 달라도 동일한 메서드가 호출됩니다.

가상 메서드의 원리

- 일반적으로 프로그램에서 메서드를 호출한다는 것은 그 메서드의 명령 집합이 있는 메모리 위치를 참조하여 명령을 실행하는 것입니다.
- 그런데 가상메서드의 경우에는 가상 메서드 테이블이 만들어집니다.
- 가상메서드 테이블은 각 메서드 이름과 실제 메모리 주소가 짝을 이루고 있습니다.
- 어떤 메서드가 호출되면 이 테이블에서 주소 값을 찾아서 해당 메서드의 명령을 수행합니다.



- calcPrice() 메서드는 두 클래스에서 서로 다른 메서드 주소를 가지고 있습니다. 이렇게 재정의된 메서드는 실제 인스턴스에 해당하는 메서드가 호출됩니다.
- showCustomerInfo()와 같이 재정의되지 않은 메서드인 경우는 메서드 주소가 같으며 상위 클래스의 메서드가 호출됩니다.

day08_10/inheritance/OverrideTest3.java

```
package day08_10.inheritance;
```



```
public class OverrideTest3 {
    public static void main(String[] args) {
        int price = 10000;

        Customer customerLee = new Customer(10010, "이순신");
        System.out.println(customerLee.getCustomerName() + " 님이
지불해야 하는 금액은" + customerLee.calcPrice(price) + "원 입니다.");

        VIPCustomer customerKim = new VIPCustomer(10020, "김유
신", 12345);
        System.out.println(customerKim.getCustomerName() + " 님이
지불해야 하는 금액은 " + customerKim.calcPrice(price) + "원 입니다.");

        Customer vc = new VIPCustomer(10030, "나몰라", 2000);
        System.out.println(vc.getCustomerName() + " 님이 지불해야
하는 금액은" + vc.calcPrice(10000) + " 원 입니다.");
    }
}
```

실행결과

이순신 님이 지불해야 하는 금액은10000원 입니다.

김유신 님이 지불해야 하는 금액은 9000원 입니다.
나몰라 님이 지불해야 하는 금액은 9000 원 입니다.

- VIPCustomer로 생성하고 Customer형으로 변환한 vc는 원래 Customer형 메서드가 호출되는 것이 맞지만, 가상 메서드 방식에 의해 VIPCustomer 인스턴스의 메서드가 호출되어 할인가격 9,000원이 출력됩니다.



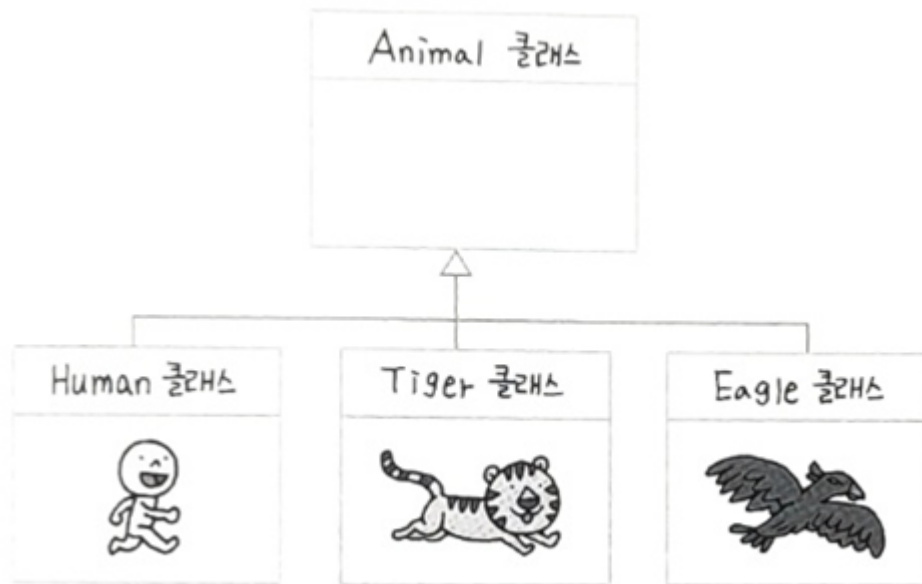
- 상위 클래스(Customer)에서 선언한 calcPrice() 메서드가 있고 이를 하위클래스(VIPCustomer)에서 재정의한 상태에서 하위 클래스 인스턴스(vc)가 상위 클래스로 형 변환이 되었습니다.
- 이때 vc.calcPrice()가 호출되면, vc 변수를 선언할 때 사용한 자료형(Customer)의 메서드가 호출되는 것이 아니라 생성된 인스턴스(VIPCustomer)의 메서드가 호출됩니다.
- 이를 가상 메서드라고 합니다. **자바의 모든 메서드는 가상메서드입니다.**

다형성

다형성이란?

- 다형성이란 하나의 코드가 여러 자료형으로 구현되어 실행되는 것을 말합니다.

다형성은 추상 클래스, 인터페이스에서 구현됩니다. 또한 안드로이드, 스트링 등 자바 기반의 프레임워크에서 응용할 수 있는 객체 지향 프로그래밍의 중요한 개념입니다.



day08_10/polymorphism/AnimalTest1.java

```
package day08_10.polymorphism;

class Animal {
    public void move( ) {
        System.out.println("동물이 움직입니다.");
    }
}

class Human extends Animal {
    public void move() {
        System.out.println("사람이 두 발로 걷습니다.");
    }
}

class Tiger extends Animal {
    public void move() {
        System.out.println("호랑이가 네 발로 뜹니다.");
    }
}

class Eagle extends Animal {
    public void move() {
        System.out.println("독수리가 하늘을 날니다.");
    }
}

public class AnimalTest1 {
    public static void main(String[] args) {
        AnimalTest1 aTest = new AnimalTest1();
        aTest.moveAnimal(new Human());
        aTest.moveAnimal(new Tiger());
        aTest.moveAnimal(new Eagle());
    }
}
```



```

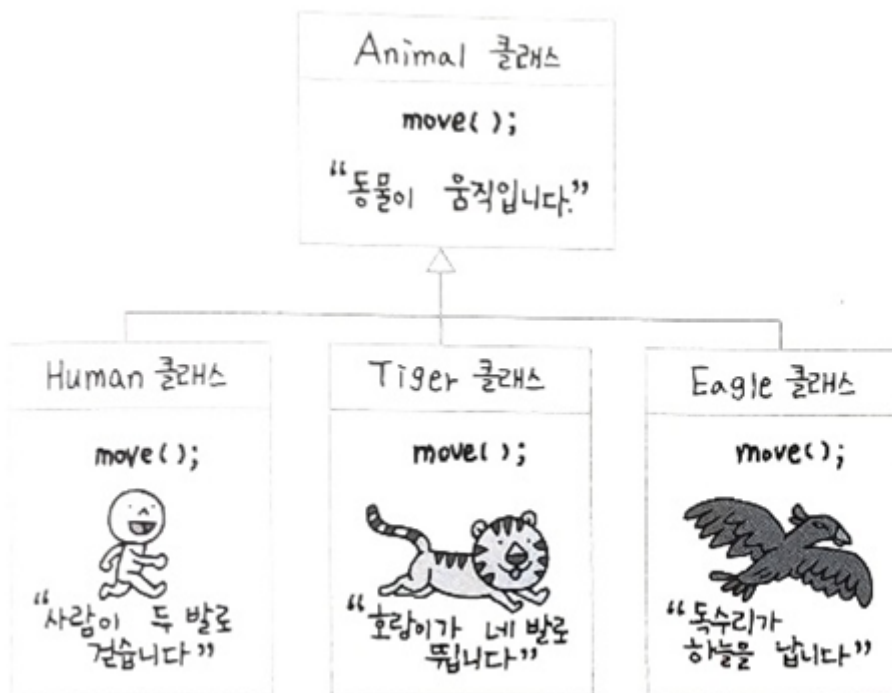
        public void moveAnimal(Animal animal) {
            animal.move();
        }
    }
}

```

실행결과

사람이 두 발로 걷습니다.
 호랑이가 네 발로 뛰니다.
 독수리가 하늘을 납니다.

- 테스트를 하기 위해 AnimalTest1 클래스에 moveAnimal(); 메서드를 만들었습니다.
 이 메서드는 어떤 인스턴스가 매개변수로 넘어와도 모두 Animal형으로 변환합니다.
- 예) Animal ani = new Human();
- Animal에서 상속받은 클래스가 매개변수로 넘어오면 모두 Animal형으로 변환되므로 animal.move() 메서드를 호출할 수 있습니다.
- 가상 메서드의 원리에 따라 animal.move가 아닌 매개변수로 넘어온 실제 인스턴스의 메서드입니다.
- animal.move() 코드는 변함이 없지만 어떤 매개변수가 넘어왔느냐에 따라 출력문이 달라집니다. 이것이 다형성 입니다.



다형성의 장점

다형성을 활용한 프로그램의 확장성 - 상위 클래스에서 공통 부분의 메서드를 제공하고, 하위 클래스에서는 그에 기반한 추가 요소를 덧붙여 구현하면 코드 양도 줄어들고 유지보수도 편리합니다. - 필요에 따라 상속받은 모든 클래스를 하나의 상위 클래스로 처리할 수 있고 다형성에 의해 각 클래스의 여러 가지 구현을 실행 할 수 있으므로 프로그램을 쉽게 확장할 수 있습니다. - 다형성을 잘 활용하면 유연하면서도 구조화된 코드를 구현하여 확장성 있고 유지보수하기 좋은 프로그램을 개발할 수 있습니다.

day08_10/polymorphism/Customer.java - 다형성을 활용해 VIP 고객 클래스 완성하기

```
package day08_10.polymorphism;
```



```
public class Customer {
    protected int customerID;
    protected String customerName;
    protected String customerGrade;
    int bonusPoint;
    double bonusRatio;

    public Customer()
    {
        // 고객 등급과 보너스 포인트 적립률 지정 함수 호출
        initCustomer();
    }

    public Customer(int customerID, String customerName){
        this.customerID = customerID;
        this.customerName = customerName;

        // 고객 등급과 보너스 포인트 적립률 지정 함수 호출
        initCustomer();
    }

    // 생성자에서만 호출하는 메서드이므로 private으로 선언
    // 멤버 변수의 초기화 부분
    private void initCustomer()
    {
        customerGrade = "SILVER";
        bonusRatio = 0.01;
    }

    public int calcPrice(int price){
        bonusPoint += price * bonusRatio;
        return price;
    }

    public String showCustomerInfo(){
        return customerName + " 님의 등급은 " + customerGrade +
        "이며, 보너스 포인트는 " + bonusPoint + "점입니다.";
    }

    public int getCustomerID() {
```

```

        return customerID;
    }

    public void setCustomerID(int customerID) {
        this.customerID = customerID;
    }

    public String getCustomerName() {
        return customerName;
    }

    public void setCustomerName(String customerName) {
        this.customerName = customerName;
    }

    public String getCustomerGrade() {
        return customerGrade;
    }

    public void setCustomerGrade(String customerGrade) {
        this.customerGrade = customerGrade;
    }
}

```

- 기존 Customer 클래스와 달라진 점은 initCustomer() 메서드가 있습니다.
- 이; 메서드는 클래스의 멤버 변수를 초기화하는데, Customer 클래스를 생성하는 두 생성자에서 공통으로 사용하는 코드이므로 메서드로 분리하여 호출했습니다.

day08_10/polymorphism/VIPCustomer.java - 다형성을 활용해 VIP 고객 클래스 완성하기

```

package day08_10.polymorphism;

public class VIPCustomer extends Customer {
    private int agentID;
    double saleRatio;

    public VIPCustomer(int customerID, String customerName, int agentID){
        super(customerID, customerName);

        customerGrade = "VIP";
        bonusRatio = 0.05;
        saleRatio = 0.1;
        this.agentID = agentID;
    }

    // 지불 가격 메서드 재정의
    public int calcPrice(int price){
        bonusPoint += price * bonusRatio;
    }
}

```



```

        return price - (int)(price * saleRatio);
    }

    // 고객 정보 출력 메서드 재정의
    public String showCustomerInfo(){
        return super.showCustomerInfo() + " 담당 상담원 번호는 "
+ agentID + "입니다";
    }

    public int getAgentID(){
        return agentID;
    }
}

```

- VIPCustomer 클래스에서 calcPrice() 메서드와 showCustomerInfo() 메서드를 재정의했습니다.
- 일반 고객 클래스에서 calcPrice()메서드는 정가를 그대로 반환했지만, VIPCustomer 클래스에서는 할인율을 반영한 지불 가격을 반환합니다. 또 일반 고객 클래스에서, showCustomerInfo()메서드는 고객 등급과 이름만 출력했지만 VIPCustomer 클래스에서는 담당 상담원 번호까지 출력합니다.

day08_10/polymorphism/CustomerTest.java - 다형성을 활용해 VIP 고객 클래스 완성하기

```
package day08_10.polymorphism;
```



```

public class CustomerTest {
    public static void main(String[] args) {
        Customer customerLee = new Customer();
        customerLee.setCustomerID(10010);
        customerLee.setCustomerName("이순신");
        customerLee.bonusPoint = 1000;
        System.out.println(customerLee.showCustomerInfo());

        // VIPCustomer를 Customer형으로 선언
        Customer customerKim = new VIPCustomer(10020, "김유신",
12345);

        customerKim.bonusPoint = 1000;
        System.out.println(customerKim.showCustomerInfo());

        System.out.println("===== 할인율과 보너스 포인트 계산
=====");

        int price = 10000;
        int leePrice = customerLee.calcPrice(price);
        int kimPrice = customerKim.calcPrice(price);
        System.out.println(customerLee.getCustomerName() + " 님이
" + leePrice + "원 지불하셨습니다.");
    }
}

```

```

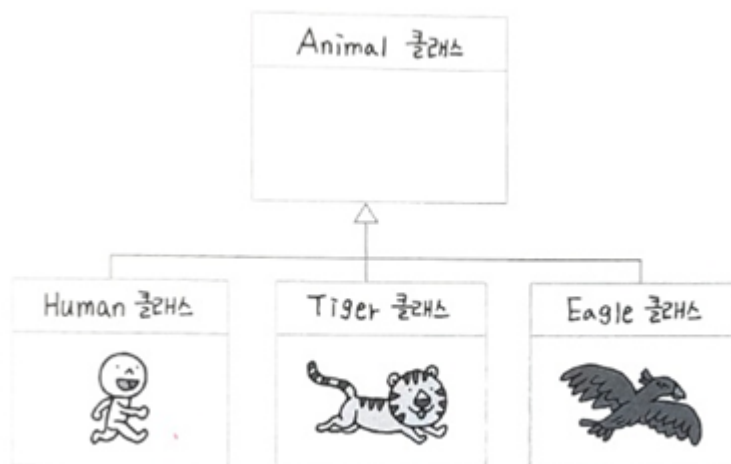
        System.out.println(customerLee.showCustomerInfo());
        System.out.println(customerKim.getCustomerName() + " 님이
" + kimPrice + "원 지불하셨습니다.");
        System.out.println(customerKim.showCustomerInfo());
    }
}

```

- 출력 결과를 보면 10,000원 짜라 상품을 구입했을 때 등급에 따라 다른 할인율과 포인트 적립이 이루어지는 것을 알 수 있습니다.
- 그런데 여기에서 customerLee와 customerKim은 모두 Customer형으로 선언되었고, 고객의 자료형은 Customer형으로 동일하지만 할인율과 보너스 포인트는 각 인스턴스의 메서드에 맞게 계산되었습니다.
- 즉, 상속 관계에 있는 상위 클래스와 하위 클래스는 같은 상위 클래스 자료형으로 선언되어 생성할 수 있지만 재정의된 메서드는 각각 호출될 뿐만 아니라 이름이 같은 메서드가 서로 다른 역할을 구현하고 있음을 알 수 있습니다.

다운 캐스팅과 instanceof

하위 클래스로 형 변환, 다운 캐스팅



- 위와 같은 계층 구조에서 상위 클래스를 자료형으로 선언하는 `Animal ani = new Human();` 코드를 쓸 수 있습니다.
- 이때 생성된 인스턴스 Human은 Animal형 입니다. 이렇게 Animal형으로 형 변환이 이루어진 경우에는 Animal 클래스에서 선언한 메서드와 멤버 변수만 사용할 수 있습니다.
- 다시 말해 Human 클래스에 더 많은 메서드가 구현되어 있고 다양한 멤버 변수가 있다고 하더라도 자료형이 Animal형인 상태에서는 사용할 수가 없습니다. 따라서 필요에 따라 다시 원래 인스턴스의 자료형(여기에서는 Human 형)으로 되돌아가야 하는 경우가 있습니다. 이렇게 상위 클래스로 형 변환 되었던 하위 클래스를 다시 원래 자료형으로 형 변환하는 것을 다운 캐스팅(down casting)이라고 합니다.

instanceof

- 상속 관계를 생각해 보면 모든 인간은 동물이지만 모든 동물이 인간은 아닙니다. 따라서 다운 캐스팅을 하기 전에 상위 클래스로 형 변환된 인스턴스의 원래 자료형을 확인해야 변환할 때 오류를 막을 수 있습니다.
- 이를 확인하는 예약어가 바로 instanceof입니다.

```
Animal hAnimal = new Human();  
if (hAnimal instanceof Human) { // hAnimal 인스턴스 자료형이 Human형이라면  
    Human human = (Human)hAnimal; // 인스턴스 hAnimal을 Human형으로 다  
    운 캐스팅  
}
```



- instanceof 예약어는 왼쪽에 있는 변수의 원래 인스턴스형이 오른쪽 클래스 자료형 인가를 확인합니다.
- instanceof의 반환 값이 true이면 다운 캐스팅을 하는데, 이때는 Human human = (Human)hAnimal; 문장과 같이 명시적으로 자료형을 써 주어야 합니다.
- 상위 클래스로는 묵시적으로 형 변환이 되지만, 하위 클래스로 형 변환을 할 때는 명시적으로 해야 합니다.
- 만약 instanceof로 인스턴스형을 확인하지 않으면 오류가 발생할 수 있습니다.
- 참조 변수의 원래 인스턴스형을 정확히 확인하고 다운 캐스팅을 해야 안전하며 이때 instanceof를 사용합니다.

day08_10/polymorphism/AnimalTest.java

```
package day08_10.polymorphism.instance_of;  
  
import java.util.ArrayList;  
  
class Animal{  
    public void move()  
    {  
        System.out.println("동물이 움직입니다.");  
    }  
}  
  
class Human extends Animal{  
    public void move()  
    {  
        System.out.println("사람이 두 발로 걸읍니다. ");  
    }  
  
    public void readBook()  
    {  
        System.out.println("사람이 책을 읽읍니다. ");  
    }  
}
```



```

class Tiger extends Animal{
    public void move()
    {
        System.out.println("호랑이가 네 발로 뛸니다. ");
    }

    public void hunting()
    {
        System.out.println("호랑이가 사냥을 합니다. ");
    }
}

class Eagle extends Animal{
    public void move()
    {
        System.out.println("독수리가 하늘을 날다 ");
    }

    public void flying()
    {
        System.out.print("독수리가 날개를 쭉 펴고 멀리 날아갑니다");
    }
}

public class AnimalTest {
    ArrayList<Animal> aniList = new ArrayList<Animal>();

    public static void main(String[] args) {
        AnimalTest aTest = new AnimalTest();
        aTest.addAnimal();
        System.out.println("원래 타입으로 다운 캐스팅 ");
        aTest.testCasting();
    }

    public void addAnimal()
    {
        aniList.add(new Human());    //ArrayList에 추가되면서
        aniList.add(new Tiger());
        aniList.add(new Eagle());

        for(Animal ani : aniList){    // 배열의 요소들을 Animal
            ani.move();                // 오버라이딩(재정의)된
        }
    }

    public void testCasting()
    {

```

```

        for(int i=0; i<aniList.size(); i++){ //모든 배열 항목들
            을 하나씩 돌면서

            Animal ani = aniList.get(i); // 일단 Shape
            타입으로 가져옴

            if(ani instanceof Human){ //Circle이면
                Human h = (Human)ani; //Circle형으로
                다운 캐스팅

                h.readBook();
            }
            else if(ani instanceof Tiger){
                Tiger t = (Tiger)ani;
                t.hunting();
            }
            else if(ani instanceof Eagle){
                Eagle e = (Eagle)ani;
                e.flying();
            }
            else{
                System.out.println("지원되지 않는 타입입니
다.");
            }
        }
    }
}

```

실행결과

사람이 두 발로 걷습니다.
 호랑이가 네 발로 뛴니다.
 독수리가 하늘을 날니다
 원래 타입으로 다운 캐스팅
 사람이 책을 읽습니다.
 호랑이가 사냥을 합니다.
 독수리가 날개를 쭉 펴고 멀리 날아갑니다

추상 클래스

추상 클래스

추상 클래스란?

- 추상적이라는 것은 구체이지 않고 막연한 것을 뜻합니다.
- 어떤 클래스가 추상적이다라는 말은 구체적인 **않은** 클래스 라는 뜻 입니다.

- 추상 클래스를 영어로 표현하면 **abstract class**이고, 추상 클래스가 아닌 클래스는 **concrete class**라고 합니다. 지금까지 클래스는 모두 **concrete class**였습니다.

추상클래스 문법

- 추상 클래스는 항상 추상 메서드를 포함 합니다.
- 추상 메서드는 구현 코드가 없습니다.
- 함수의 구현코드가 없다는 것은 함수 몸체(body)가 없다는 뜻 입니다.

```
// {} 안의 내용이 함수의 몸체 (구체적인(구현된) 메서드)
int add(int x, int y) {
    return x + y;
}
```



- {}로 감싼 부분을 함수의 **구현부**(implemetatation)라고 합니다. 이 부분이 없는 함수는 추상 함수(abstract function)이고 자바에서는 **추상 메서드**(abstract method)라고 합니다.
- 추상 메서드는 **abstract 예약어**를 사용 합니다.
- {}(구현부)대신 ; 를 씁니다.

```
abstract int add(int x, int y);
```



- ** 자바에서 추상메서드는 abstract 예약어를 사용하여 선언만 하는 메서드 입니다.**

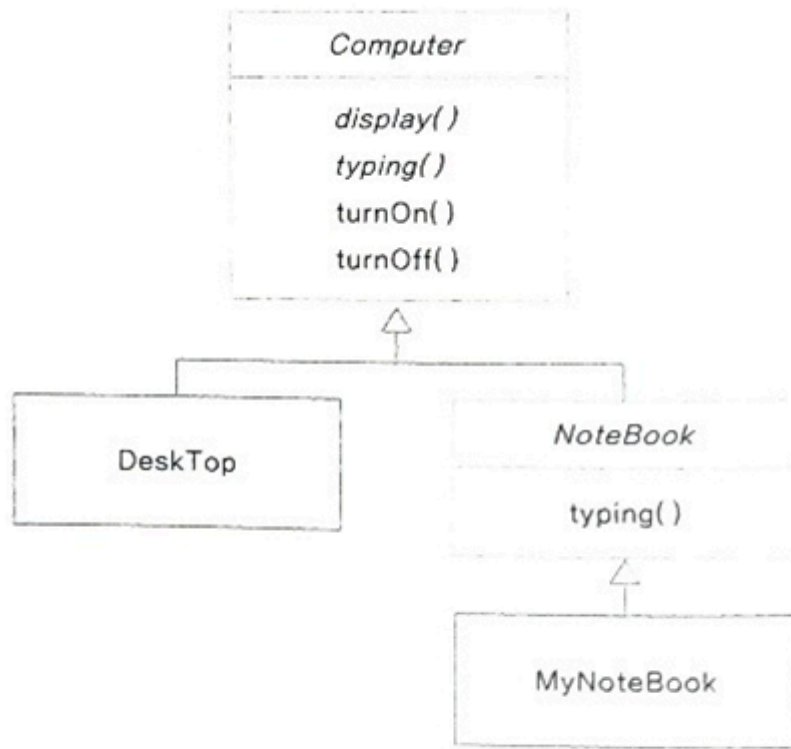
메서드 선언의 의미

```
int add(int num1, int num1);
```



- 상기 코드처럼 선언한 메서드를 보면 두 개의 정수를 입력받은 후 더해서 그 결과 값을 반환한다는 것을 유추할 수 있습니다.
- 즉, 이 메서드의 선언부(declatration)만 봐도 어떤 일을 하는 메서드인지 알 수 있습니다. 함수의 선언부 즉, 반환 값, 함수 이름, 매개변수를 정의한다는 것은 곧 함수의 역할이 무엇인지, 어떻게 구현해야 하는지를 정의한다는 뜻입니다.
- 따라서 함수 몸체를 구현하는 것보다 중요한 것은 **함수 선언부**를 작성하는 것입니다.
- 자바에서 사용하는 메서드 역시 마찬가지로 **메서드를 선언한다는 것은 메서드가 해야 할 일을 명시해 두는 것입니다.**

추상 클래스 구현하기



> 추상 클래스, 추상 메서드는 기울임꼴로 표시

- Computer 클래스는 추상클래스 입니다.
- 컴퓨터 종류에는 데스크톱과 노트북이 있습니다. 그리고 노트북의 종류에는 MyNoteBook이 있습니다.
- Computer 클래스는 추상 클래스이며 이를 상속받은 두 클래스 중 Desktop 클래스는 일반 클래스이고 Notebook 클래스는 추상 클래스입니다. 마지막으로 Notebook 클래스를 상속받은 MyNoteBook 클래스도 일반 클래스 입니다.
- display()와 typing()은 추상 메서드이고 turnOn()과 turnOff()는 구현코드가 있는 메서드입니다.

day08_10/abstractex/Computer.java

```

package day08_10.abstractex;

public class Computer {
    public void display(); // 오류 발생
    public void typing(); // 오류 발생

    public void turnOn() {
        System.out.println("전원을 켭니다.");
    }

    public void turnOff() {
        System.out.println("전원을 끕니다.");
    }
}
  
```

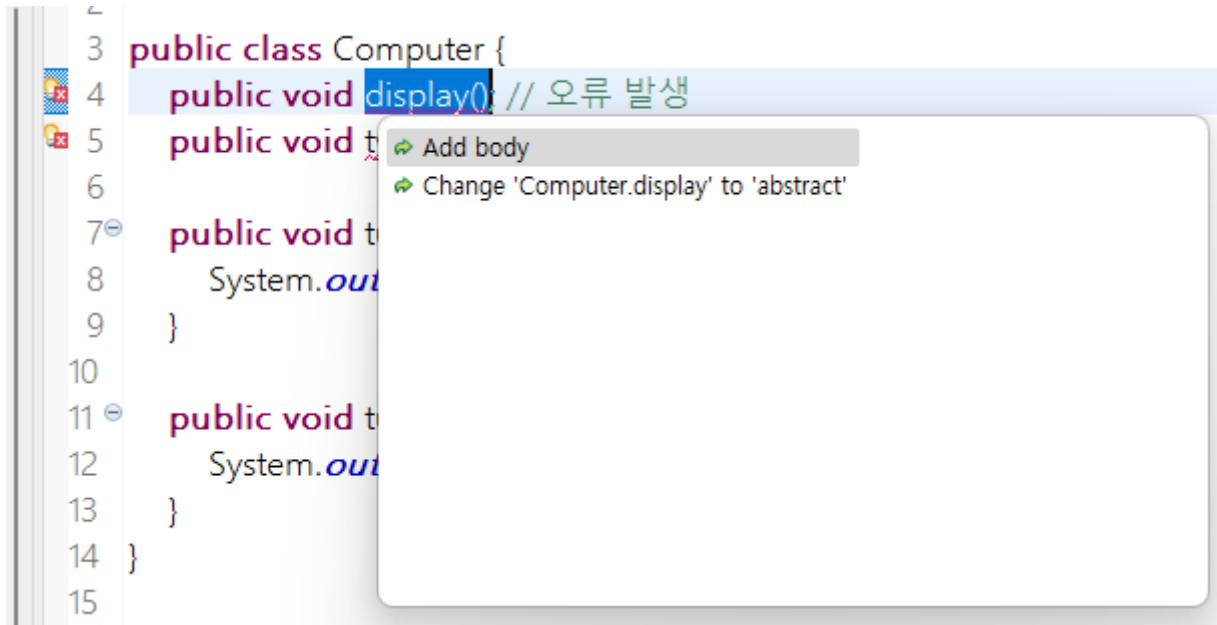


```

    }
}

```

- Computer 클래스 내부에 추상 메서드 display()와 typing()을 선언하고, 구현 메서드 turnOn()과 turnOff()를 작성합니다.
- 그러면 완전하게 구현되지 않은 두 추상메서드에서 오류가 발생합니다.
- display()나 typing() 위에 마우스를 올리면 오류를 해결할 수 있는 방법으로 다음 두 가지를 제시합니다.



- add body - 몸체 부분을 작성하시오.
- Change Computer display to 'abstract' - 이 메서드를 추상 메서드로 바꾸시오.

```
package day08_10.abstractex;
```



```

public class Computer { // 오류 발생
    public abstract void display(); // 오류가 남아 있음
    public abstract void typing(); // 오류가 남아 있음

    ...
}

```

- 이번에는 메서드와 클래스 이름에 모두 오류가 표시됩니다.
- 추상 메서드가 속한 클래스를 추상 클래스로 선언하지 않았기 때문입니다.

```
1 package day08_10.abstractex;
2
3 public class Computer { // 오류 발생
4     public abstract void display(); // 오류가 남아 있음
5     public abstract void typing();
6
7     public void turnOn() {
8         System.out.println("turnOn");
9     }
10
11     public void turnOff() {
12         System.out.println("turnOff");
13     }
14 }
15
```

Remove 'abstract' modifier

Make type 'Computer' abstract

- Remove 'abstract' modifier - 메서드에서 abstract 예약어를 제거하세요.
- Make type 'Computer' abstract - Computer 클래스를 추상 클래스로 만드세요.

```
package day08_10.abstractex;
```



```
public abstract class Computer {
    public abstract void display(); // 더이상 오류 없음
    public abstract void typing(); // 더이상 오류 없음
    ...
}
```

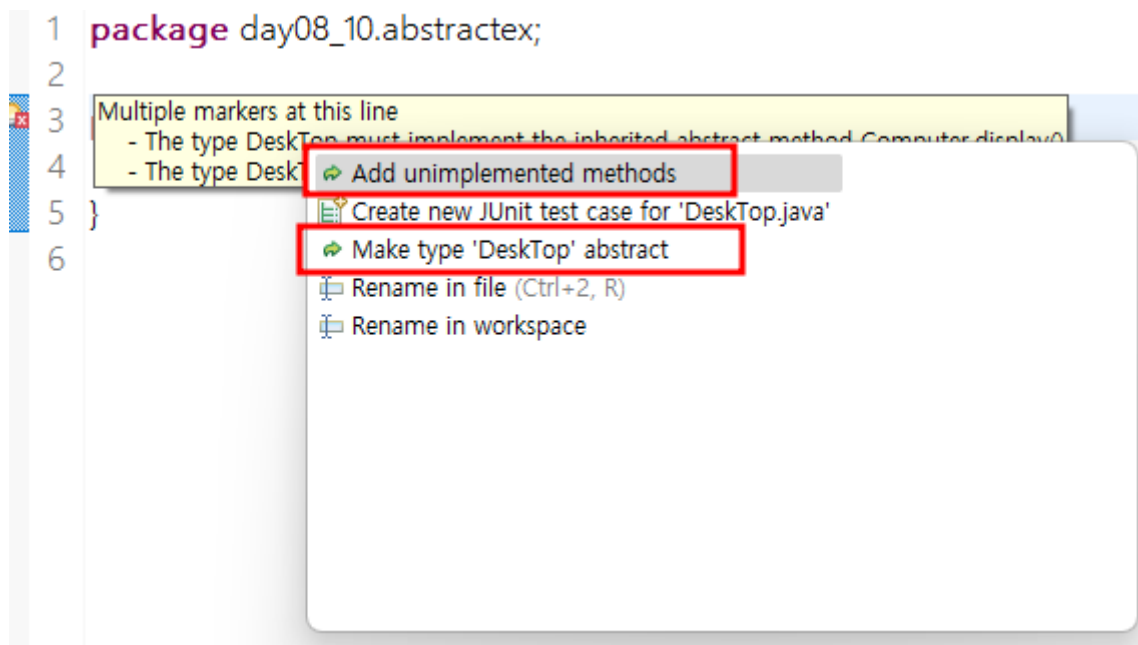
- Computer 클래스를 이와 같이 구현한 의미
 - Computer 클래스를 상속받는 클래스 중 `turnOn()`과 `turnOff()` 구현 코드는 공통이다.
 - 하지만 `display()`와 `typing()`은 하위 클래스에 따라 구현이 달라질 수 있다.
 - 그래서 Computer에서는 구현하지 않고, 이 두 메서드 구현에 대한 책임을 상속받는 클래스에 위임한다.
- Computer 클래스의 추상 메서드는 추상 클래스를 상속받은 DeskTop과 NoteBook에서 실제로 구현하게 됩니다. 이 클래스의 상위 클래스는 하위 클래스도 공통으로 사용할 메서드를 구현하고, 하위 클래스마다 다르게 구현할 메서드는 추상 메서드로 선언해 두는 것입니다.

day08_10/abstractex/DeskTop.java

```
package day08_10.abstractex;
```



```
public class DeskTop extends Computer { // 오류 발생
    ...
}
```



- 상속받은 DeskTop 클래스에 빨간색 줄로 오류 표시가 보입니다.
 - Add unimplemented methods - 구현되지 않은 메서드를 구현하시오.
 - Make type 'DeskTop' abstract - DeskTop 클래스를 추상 클래스로 만드시오.
- 원래 Computer는 추상 클래스입니다. 추상 클래스를 상속받은 추상 클래스가 가진 메서드를 상속받습니다. 따라서 상속받은 클래스는 추상 메서드를 포함합니다.
- 그렇기 때문에 추상메서드를 모두 구현하든가 아니면 DeskTop도 추상 클래스로 만들든가 둘 중 하나를 해야 합니다.
- 즉, 추상 클래스를 상속받은 하위 클래스는 구현되지 않은 추상 메서드를 모두 구현해야 구체적인 클래스가 됩니다.
- Add unimplemented methods 옵션을 눌러보면 비어있던 클래스에 다음과 같은 코드가 생성이 됩니다.

```
...  
@Override  
    public void display() {  
        // TODO Auto-generated method stub  
  
    }  
  
    @Override  
    public void typing() {  
        // TODO Auto-generated method stub  
  
    }  
...
```



- 주석 부분을 제거하고 다음과 같이 몸체 코드를 작성합니다.

day08_10/abstractex/DeskTop.java

```
package day08_10.abstractex;

public class DeskTop extends Computer {

    @Override
    public void display() {
        // 추상 메서드의 몸체 코드 작성
        System.out.println("DeskTop display()");
    }

    @Override
    public void typing() {
        // 추상 메서드의 몸체 코드 작성
        System.out.println("DeskTop typing()");
    }
}
```



day08_10/abstractex/NoteBook.java

```
package day08_10.abstractex;

public abstract class NoteBook extends Computer {
    @Override
    public void display() {
        System.out.println("NoteBook display()");
    }
}
```



- 이 클래스에서는 상속받은 추상 메서드를 모두 구현하지 않고 display() 하나만 구현 하였습니다.
- NoteBook 클래스는 추상메서드를 여전히 하나 가지고 있기 때문에 추상 클래스가 됩니다.
- NoteBook을 상속받은 MyNoteBook 클래스는 다음과 같이 구현할 수 있습니다.

```
package day08_10.abstractex;

public class MyNoteBook extends NoteBook {
    @Override
    public void typing() {
        System.out.println("MyNoteBook typing()");
    }
}
```



모든 추상 메서드를 구현한 클래스를 구현한 abstract 예약어를 사용한다면?

```
package day08_10.abstractex;

public abstract class AbstractTV {
    public void turnOn() {
        System.out.println("전원을 켭니다.");
    }

    public void turnOff() {
        System.out.println("전원을 끕니다.");
    }
}
```



- AbstractTV 클래스는 모든 추상메서드를 구현한 클래스입니다. 하지만 이것으로는 완벽한 TV기능이 구현된 것이 아니고 TV의 **공통 기능만 구현해 놓은 것**입니다.
- 이 클래스는 **생성해서 사용할 목적이 아닌 상속만을 위해 만든 추상클래스**입니다. 이 경우에 **new 예약어로 인스턴스를 생성할 수 없습니다**.

추상 클래스를 만드는 이유

day08_10/abstractex/ComputerTest.java

```
package day08_10.abstractex;

public class ComputerTest {
    public static void main(String[] args) {
        Computer c1 = new Computer(); // 클래스를 인스턴스로 생성
할 수 없음

        Computer c2 = new DeskTop();
        Computer c3 = new NoteBook(); // 클래스를 인스턴스로 생성
할 수 없음

        Computer c4 = new MyNoteBook();
    }
}
```



- Computer 클래스형 인스턴스를 4개 생성했습니다. 그러나 Computer와 NoteBook에서 오류가 납니다.
- 오류 메시지를 확인해 보면 Computer클래스와 NoteBook 클래스를 인스턴스로 생성할 수 없다고 나옵니다.

1. 추상 클래스는 인스턴스로 생성할 수 없다.

- 추상 클래스는 모든 메서드가 구현되지 않았으므로 인스턴스로 생성할 수 없습니다.

2. 추상 클래스에서 구현하는 메서드

- 생성할 수 없는 추상 클래스는 상속을 하기 위해 만든 클래스입니다.
- 추상클래스에는 추상메서드와 구현된 메서드가 함께 사용 될 수 있습니다.
- 구현된 메서드는 하위 클래스에서도 사용할 즉, 하위 클래스에서도 구현 내용을 공유할 메서드를 구현합니다.
- 실제 하위 클래스에서 내용을 각각 다르게 구현해야 한다면, 구현 내용을 추상 메서드로 남겨 두고 하위 클래스에 구현을 위임하는 것입니다.
- 구현된 메서드 : 하위 클래스에서 공통으로 사용할 구현 코드, 하위 클래스에서 재정의할 수도 있음.
- 추상 메서드 : 하위 클래스가 어떤 클래스냐에 따라 구현 코드가 달라짐
- 앞에서 구현한 Computer 클래스에서 turnOn()과 turnOff()의 구현은 하위 클래스에서 공유할 수 있지만 display()와 typing()의 구현 내용은 NoteBook인지 DeskTop인지에 따라 달라지므로 Computer 클래스에서는 구현하지 않은 것입니다.

추상클래스와 다형성

- DestkTop은 상위 클래스인 Computer의 클래스 자료형으로 선언하고 대입될 수 있습니다. 마찬가지로 MyNoteBook역시 상위 클래스인 Computer의 클래스 자료형으로 선언하고 대입될 수 있습니다.
- 상위 클래스인 추상 클래스는 하위에 구현된 여러 클래스를 하나의 자료형(상위 클래스 자료형)으로 선언하거나 대입할 수 있습니다.
- 추상 클래스에 선언된 메서드를 호출하면 가상 메서드에 의해 각 클래스에 구현된 기능이 호출됩니다.
- 즉, 하나의 코드가 다양한 자료형을 대상으로 동작하는 다형성을 활용할 수 있습니다.

final 예약어

- final은 마지막이라는 의미입니다.
- 즉, 마지막으로 정한 것이니 더 이상 수정할 수 없다는 뜻입니다.
- 자바 프로그램에서는 final 예약어는 변수, 메서드, 클래스에 사용할 수 있습니다.

사용 위치	설명
변수	final 변수는 상수를 의미합니다.
메서드	final 메서드는 하위 클래스에서 재정의할 수 없습니다.
클래스	final 클래스는 상속할 수 없습니다.

상수를 의미하는 final 변수

day08_10/finalex/Constant.java

```
package day08_10.finalex;

public class Constant {
    int num = 10;
    final int NUM = 100; // 상수 선언

    public static void main(String[] args) {
        Constant cons = new Constant();
        cons.num = 50;
        //cons.NUM = 200; // 상수 값에 대입하면 오류 발생

        System.out.println(cons.num);
        System.out.println(cons.NUM);
    }
}
```

여러 자바 파일에서 공유하는 상수 값 정의하기

- 하나의 자바 파일에서만 사용하는 상수 값은 해당 파일 안에서 정의해서 사용할 수 있습니다. 그러나 프로젝트를 하다 보면 **여러 파일에서 똑같이 공유해야 하는 상수 값**도 있습니다.
- 자바로 프로젝트를 진행할 때 여러 파일에서 공유해 하는 상수 값은 한 파일에 모아 **public static final로 선언하여 사용할 수** 있습니다.

day08_10/finalex/Define.java

```
package day08_10.finalex;

public class Define {
    public static final int MIN = 1;
    public static final int MAX = 99999;
    public static final int ENG = 1001;
    public static final int MATH = 2001;
    public static final double PI = 3.14;
    public static final String GOOD_MORNING = "Good Morning!";
}
```

day08_10/finalex/UsingDefine.java

```
package day08_10.finalex;

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



```
// static으로 선언했으므로 인스턴스를 생성하지않고 클래스
이름으로 참조 가능
System.out.println(Define.GOOD_MORNING);
System.out.println("최소값은 " + Define.MIN + "입니다.");
System.out.println("최대값은 " + Define.MAX + "입니다.");
System.out.println("수학 과목 코드 값은 " + Define.MATH +
"입니다.");
System.out.println("영어 과목 코드 값은 " + Define.ENG +
"입니다.");
    }
}
```

실행결과

```
Good Morning!
최소값은 1입니다.
최대값은 99999입니다.
수학 과목 코드 값은 2001입니다.
영어 과목 코드 값은 1001입니다.
```

상속할 수 없는 final 클래스

클래스를 final로 선언하면 상속할 수 없습니다.

재정의 할 수 없는 final 메서드

메서드를 final로 선언하면 하위클래스에서 재정의 할 수 없습니다.

인터페이스

인터페이스란?

구현 코드가 없는 인터페이스

- 인터페이스(interface)는 클래스 혹은 프로그램이 제공하는 기능을 명시적으로 선언하는 역할을 합니다.
- 인터페이스는 추상 메서드와 상수로만 이루어져 있습니다.
- 구현된 코드가 없기 때문에 인스턴스를 생성할 수 없습니다.

인터페이스 만들기

- 이클립스에서 인터페이스를 만들려면 패키지에서 마우스 오른쪽버튼을 클릭하고 New -> Interface를 클릭

- Name 항목에 만들려는 목적에 맞는 인터페이스 이름을 입력하고 **Finish**를 클릭하면 인터페이스가 만들어집니다.

day08_10/interfaceex/Calc.java

```
package day08_10.interfaceex;

public interface Calc {
    // 인터페이스에서 선언한 변수는 컴파일 과정에서 상수로 변환됨
    double PI = 3.14;
    int ERROR = -9999999;

    // 인터페이스에서 선언한 메서드는 컴파일 과정에서 추상 메서드로 변환됨
    int add(int num1, int num2);
    int subtract(int num1, int num2);
    int times(int num1, int num2);
    int divide(int num1, int num2);
}
```

- 이 인터페이스는 계산기를 만들기 위해 선언한 코드입니다.
- Calc 인터페이스에는 원주율을 뜻하는 PI 변수와 오류가 났을 때 사용할 ERROR 변수, 그리고 사칙연산을 수행하기 위해 add(), subtract(), times(), divide() 메서드를 선언했습니다.
- 인터페이스에 선언된 메서드는 모두 구현코드가 없는 추상메서드입니다.
- 메서드는 public abstract 예약어를 명시적으로 쓰지 않아도 컴파일 과정에서 자동으로 추상메서드로 변환됩니다.
- 인터페이스에 선언한 변수는 모두 컴파일 과정에서 값이 변하지 않는 상수로 자동 변환됩니다. public static final 예약어를 쓰지 않아도 무조건 상수로 인식합니다.

클래스에서 인터페이스 구현하기

- 인터페이스를 클래스가 사용하는 것을 ****클래스에서 인터페이스를 구현한다 (implements)****라고 표현합니다.
- 인터페이스에서는 인터페이스에 선언한 기능을 클래스가 구현한다는 의미로 **implements 예약어**를 사용합니다.

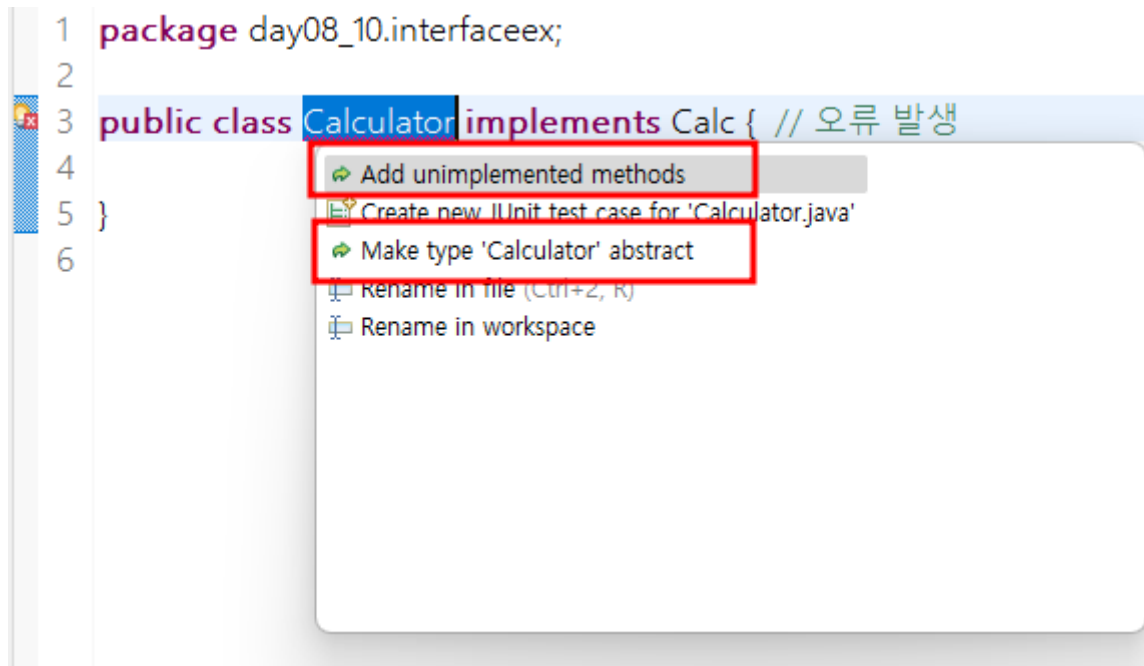
day08_10/interfaceex/Calculator.java

```
package day08_10.interfaceex;

public class Calculator implements Calc { // 오류 발생

}
```

- 그러나 상기 코드는 다음과 같은 오류가 표시됩니다.



- Add unimplemented methods - 추상 메서드를 구현하십시오. - Make type 'Calculator' abstract - Calculator 클래스를 추상 클래스로 만드십시오.

- Calculator 클래스에서 Calc 인터페이스를 구현한다고 했으므로 Calculator 클래스는 추상 메서드 4개(add(), subtract(), times(), divide() 메서드)를 포함합니다.
- 이 추상 메서드를 구현하지 않으면 Calculator 클래스도 추상 클래스가 됩니다.
- 위 두 오류 메시지는 Calc 인터페이스에 포함된 추상 메서드를 구현하거나 Calculator 클래스를 추상 클래스로 만들라는 의미입니다.
- Add unimplemented method 옵션을 클릭하여 Calc 인터페이스에 선언된 4개 추상 메서드 중 add()와 subtract() 2개만 구현하여 추상 클래스를 만들어 보겠습니다.

day08_10/interfaceex/Calculator.java

```
package day08_10.interfaceex;

public abstract class Calculator implements Calc {

    @Override
    public int add(int num1, int num2) {
        // TODO Auto-generated method stub
        return 0;
    }

    @Override
    public int subtract(int num1, int num2) {
        // TODO Auto-generated method stub
        return 0;
    }
}
```



```
}  
}
```

- 추상 메서드 times()와 divide()를 구현하지 않았으므로 Calculator는 추상 클래스입니다.

클래스 완성하고 실행하기

- 아직 구현하지 않은 times()와 divide() 추상 메서드를 이 클래스에서 구현합니다.

day08_10/interfaceex/CompleteCalc.java

```
package day08_10.interfaceex;  
  
public class CompleteCalc extends Calculator {  
    @Override  
    public int times(int num1, int num2) {  
        return num1 * num2;  
    }  
  
    @Override  
    public int divide(int num1, int num2) {  
        if (num2 != 0)  
            return num1/num2;  
        else  
            return Calc.ERROR; // num2가 0, 즉 나누는 수가 0  
            인 경우에 대해 오류 반환  
    }  
  
    // CompleteCalc에서 추가로 구현한 메서드  
    public void showInfo() {  
        System.out.println("Calc 인터페이스를 구현하였습니다.");  
    }  
}
```

day08_10/interfaceex/CalculatorTest.java

```
package day08_10.interfaceex;  
  
public class CalculatorTest {  
    public static void main(String[] args) {  
        int num1 = 10;  
        int num2 = 5;  
  
        CompleteCalc calc = new CompleteCalc();  
        System.out.println(calc.add(num1, num2));  
        System.out.println(calc.subtract(num1, num2));  
        System.out.println(calc.times(num1, num2));  
        System.out.println(calc.divide(num1, num2));  
    }  
}
```

```

        calc.showInfo();
    }
}

```

실행결과

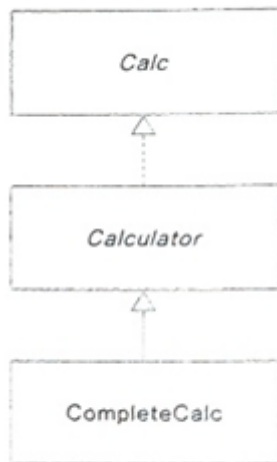
```

0
0
50
2

```

Calc 인터페이스를 구현하였습니다.

인터페이스 구현과 형변환(다형성)



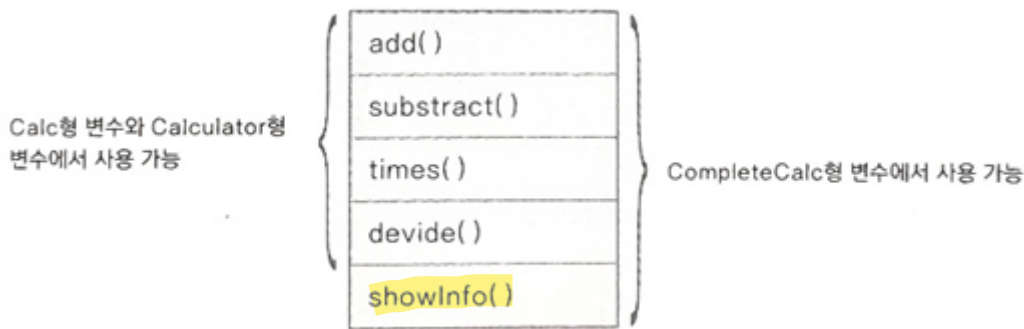
- **Calculator 클래스**는 인터페이스에서 선언한 추상 메서드 중 일부 메서드만 구현했으므로 추상 클래스입니다.
- 이를 상속받은 **CompleteCalc 클래스**는 Calculator 클래스에서 구현하지 않은 나머지 추상 메서드를 모두 구현하고 **showInfo()** 메서드를 추가로 구현 했습니다.
- 상속 관계에서 하위 클래스는 상위클래스 자료형으로 묵시적으로 형 변환할 수 있다고 했습니다. 인터페이스도 마찬가지 입니다.
- CompleteCalc 클래스는 상위 클래스인 Calculator형이면서, Calc 인터페이스를 구현 하였으므로 Calc형이기도 합니다. 따라서 별다른 조치 없이 다음처럼 Calc형으로 선언한 변수에 대입할 수 있습니다.

```
Calc calc = new CompleteCalc();
```



- calc 변수가 사용할 수 있는 메서드 목록에 Calc에서 선언한 추상 메서드 add(), subtract(), times(), divide()는 있지만 CompleteCalc 클래스에서 추가로 구현한 showInfo() 메서드는 Calc 인터페이스에 선언한 메서드뿐 입니다.

- 정리하면, 인터페이스를 구현한 클래스가 있을 때 그 클래스는 해당 인터페이스형으로 묵시적 형변환이 이루어지며, **형 변환되었을 때 사용할 수 있는 메서드는 인터페이스에서 선언한 메서드뿐입니다.**



인터페이스의 요소 살펴보기

인터페이스 상수

- 인터페이스는 추상 메서드로 이루어지므로 인스턴스를 생성할 수 없으며, 멤버 변수도 사용할 수 없습니다.**
- 그런데 인터페이스에 다음 코드와 같이 변수를 선언해도 오류가 발생하지 않습니다.

```
public interface Calc {
    double PI = 3.14;
    int ERROR = -99999999;
    ...
}
```



- 그 이유는 **인터페이스에 선언한 변수를 컴파일하면 상수로 변환되기 때문입니다.**
- Calc 인터페이스에 선언한 변수 PI를 컴파일하면 `public static final double PI = 3.14;` 즉 상수 3.14로 변환 됩니다.
- 그리고 `int`형 변수 ERROR 역시 `public static final int ERROR = -99999999`로 변환되어 상수로 취급됩니다.

디폴트 메서드와 정적 메서드

- JDK1.7까지는 인터페이스에서 추상 메서드와 상수, 이 두 가지 요소만 선언해서 사용할 수 있었습니다. 그런데 어떤 인터페이스를 구현한 여러 클래스에서 사용할 메서드가 클래스마다 같은 기능을 제공하는 경우가 있었습니다. JDK1.7까지는 기능이 같다고 해도 인터페이스에서 코드를 구현할 수 없으므로 추상 메서드를 선언하고 각 클래스마다 똑같이 그 기능을 반복해 구현해야 해서 굉장히 번거로웠습니다. 또한 클래스를 생성하지 않아도 사용할 수 있는 메서드(정적 메서드)가 필요한 경우가 있는데, 인터페이스만으로는 메서드를 호출할 수가 없어 불편했었습니다.

- JDK1.8부터 이런 부분에서 인터페이스의 활용성을 높이기 위해 디폴트 메서드와 정적메서드 기능을 제공합니다.
- **디폴트 메서드** : 인터페이스에서 구현 코드까지 작성한 메서드 입니다. 인터페이스를 구현한 클래스에 기본적으로 제공할 메서드 입니다.
- **정적 메서드** : 인스턴스 생성과 상관없이 사용할 수 있는 메서드 입니다.
- 디폴트 메서드나 정적 메서드를 추가했다고 해서 인스턴스를 생성할 수 있는 것은 아닙니다.

디폴트 메서드

- 디폴트 메서드란 말 그대로 기본으로 제공되는 메서드입니다.
- 디폴트 메서드는 인터페이스에서 구현하지만, 이후 인터페이스를 구현한 클래스가 생성되면 그 클래스에서 사용할 기본 기능입니다.
- 디폴트 메서드를 선언할 때는 **default 예약어**를 사용합니다.

day08_10/interfaceex/Calc.java

```
package day08_10.interfaceex;

public interface Calc {
    // 인터페이스에서 선언한 변수는 컴파일 과정에서 상수로 변환됨
    double PI = 3.14;
    int ERROR = -9999999;

    // 인터페이스에서 선언한 메서드는 컴파일 과정에서 추상 메서드로 변환됨
    int add(int num1, int num2);
    int subtract(int num1, int num2);
    int times(int num1, int num2);
    int divide(int num1, int num2);

    default void description() {
        System.out.println("정수 계산기를 구현합니다.");
    }
}
```

day08_10/interfaceex/CalculatorTest.java

```
package day08_10.interfaceex;

public class CalculatorTest {
    public static void main(String[] args) {
        int num1 = 10;
        int num2 = 5;
```

```

        CompleteCalc calc = new CompleteCalc();
        System.out.println(calc.add(num1, num2));
        System.out.println(calc.subtract(num1, num2));
        System.out.println(calc.times(num1, num2));
        System.out.println(calc.divide(num1, num2));
        calc.showInfo();
        calc.description(); // 디폴트 메서드 호출
    }
}

```

- 디폴트 메서드는 인터페이스에 이미 구현되어 있으므로 인터페이스를 구현한 추상 클래스 Calculator나 추상 클래스를 상속받은 CompleteCalc 클래스에서 코드를 구현할 필요가 없습니다.

디폴트 메서드 재정의하기

- 이미 인터페이스에 구현되어 있는 디폴트 메서드가 새로 생성한 클래스에서 원하는 기능과 맞지 않는다면, 하위 클래스에서 디폴트 메서드를 재정의 할 수 있습니다.

```

public class CompleteCalc extends Calculator {
    ...
    @Override
    public void description() {
        // TODO Auto-generated method stub
        super.description();
    }
}

```



- super.description()은 인터페이스에 선언한 메서드를 의미합니다.
- 이 코드를 사용하지 않을 거라면 지우고 새 코드를 작성하면 됩니다. 이제 CompleteCalc 클래스로 인스턴스를 생성하여 호출하면 재정의된 메서드가 호출됩니다.

정적 메서드

- 정적 메서드는 static 예약어를 사용하여 선언하며 클래스 생성과 무관하게 사용할 수 있습니다.
- 정적 메서드를 사용할 때는 인터페이스 이름으로 직접 참조하여 사용합니다.

day08_10/interfaceex/Calc.java

```

package day08_10.interfaceex;

public interface Calc {
    ...
    // 인터페이스에 정적 메서드 total() 구현
    static int total(int[] arr) {

```




```

        int total = 0;

        for(int i : arr) {
            total += i;
        }

        return total;
    }
}

```

day08_10/interfaceex/CalculatorTest.java

```

package day08_10.interfaceex;

public class CalculatorTest {
    public static void main(String[] args) {
        int num1 = 10;
        int num2 = 5;

        CompleteCalc calc = new CompleteCalc();
        System.out.println(calc.add(num1, num2));
        System.out.println(calc.subtract(num1, num2));
        System.out.println(calc.times(num1, num2));
        System.out.println(calc.divide(num1, num2));
        calc.showInfo();
        calc.description(); // 디폴트 메서드 호출

        int[] arr = {1, 2, 3, 4, 5};
        System.out.println(Calc.total(arr));
    }
}

```

실행결과

```

0
0
50
2
Calc 인터페이스를 구현하였습니다.
정수 계산기를 구현합니다.
15

```

private 메서드

- 자바 9부터 인터페이스에 private 메서드를 구현할 수 있습니다.
- private 메서드는 인터페이스를 구현한 클래스에서 사용하거나 재정의할 수 없습니다.

- 기존에 구현된 코드를 변경하지 않고 인터페이스를 구현한 클래스에서 공통으로 사용하는 경우에 `private` 메서드로 구현하면 코드의 재사용성을 높일 수 있습니다.
- 추상메서드에는 `private` 예약어는 사용할 수 없지만, `static` 예약어는 함께 사용할 수 있습니다.

day08_10/interfaceex/Calc.java

```
package day08_10.interfaceex;

public interface Calc {
    // 인터페이스에서 선언한 변수는 컴파일 과정에서 상수로 변환됨
    double PI = 3.14;
    int ERROR = -9999999;

    // 인터페이스에서 선언한 메서드는 컴파일 과정에서 추상 메서드로 변환됨
    int add(int num1, int num2);
    int subtract(int num1, int num2);
    int times(int num1, int num2);
    int divide(int num1, int num2);

    default void description() {
        System.out.println("정수 계산기를 구현합니다.");
        // 디폴트 메서드에서 private 메서드 호출
        myMethod();
    }

    // 인터페이스에 정적 메서드 total() 구현
    static int total(int[] arr) {
        int total = 0;

        for(int i : arr) {
            total += i;
        }
        // 정적 메서드에서 private static 메서드 호출
        myStaticMethod();
        return total;
    }

    // private 메서드
    private void myMethod() {
        System.out.println("private 메서드 입니다.");
    }

    // private static 메서드
    private static void myStaticMethod() {
        System.out.println("private static 메서드입니다.");
    }
}
```

CalculatorTest 클래스 실행 결과

```
0
0
50
2
Calc 인터페이스를 구현하였습니다.
정수 계산기를 구현합니다.
private 메서드 입니다.
private static 메서드입니다.
15
```

인터페이스 활용하기

한 클래스가 여러 인터페이스를 구현하는 경우

- 한 클래스가 여러 클래스를 상속받으면 메서드 호출이 모호해지는 문제가 발생할 수 있습니다.
- 하지만 인터페이스는 한 클래스가 여러 인터페이스를 구현할 수 있습니다.



day08_10/interfaceex/Buy.java

```
package day08_10.interfaceex;

public interface Buy {
    void buy();
}
```



day08_10/interfaceex/Sell.java

```
package day08_10.interfaceex;

public interface Sell {
    void sell();
}
```



Buy 인터페이스에 추상 메서드 buy()가 선언되어 있고, Sell 인터페이스에 추상 메서드 sell()이 선언되어 있습니다. Customer 클래스가 두 인터페이스를 구현하는 코드는 다음과 같습니다.

day08_10/interfaceex/Customer.java

```
package day08_10.interfaceex;

// Customer 클래스는 Buy와 Sell 인터페이스를 모두 구현함
public class Customer implements Buy, Sell {

    @Override
    public void sell() {
        System.out.println("판매하기");
    }

    @Override
    public void buy() {
        System.out.println("구매하기");
    }
}
```

- 인터페이스는 구현 코드나 멤버 변수를 가지지 않기 때문에 여러개를 동시에 구현할 수 있습니다.
- 두 인터페이스에 이름이 같은 메서드가 선언되었다고 해도 구현은 클래스에서 이루어지므로, 어떤 메서드를 호출해야 하는지 모호하지 않습니다.
- 이렇게 두 인터페이스를 구현한 Customer 클래스는 Buy형이자 Sell형 이기도 합니다.

day08_10/interfaceex/CustomerTest.java

```
package day08_10.interfaceex;

public class CustomerTest {
    public static void main(String[] args) {
        Customer customer = new Customer();

        // Customer 클래스형인 customer를 Buy 인터페이스형인 buyer
        //에 대입하여 형 변환,
        // buyer는 Buy 인터페이스의 메서드만 호출 가능
        Buy buyer = customer;
        buyer.buy();

        // Customer 클래스형인 customer를 Sell 인터페이스형인
        //seller에 대입하여 형 변환,
        // seller는 Sell 인터페이스의 메서드만 호출 가능
        Sell seller = customer;
        seller.sell();
    }
}
```

```

        if (seller instanceof Customer) {
            // seller를 하위 클래스형인 Customer로 다시 형 변환
            Customer customer2 = (Customer)seller;
            customer2.buy();
            customer2.sell();
        }
    }
}

```

실행결과

구매하기
판매하기
구매하기
판매하기

- `Buy buyer = customer;` 처럼 `customer`를 `Buy` 인터페이스형 변수에 대입하면 형 변환이 일어나 `Buy` 인터페이스에 선언한 메서드만 호출할 수 있습니다.
- `Sell`형으로 변환될 때도 마찬가지 입니다. 또한 상속 관계에서와 마찬가지로 원래의 인스턴스 자료형으로 다운 캐스팅하기 위해서는 `instanceof`를 사용하여 본래 인스턴스 자료형으로 안전하게 변환할 수 있습니다.

두 인터페이스의 디폴트 메서드가 중복되는 경우

- **정적 메서드**는 인스턴스 생성과 상관없이 사용할 수 있습니다. `Customer` 클래스가 `Buy`, `Sell` 두 인터페이스를 구현하고 `Buy` 인터페이스와 `Sell` 인터페이스에 똑같은 `pay()` 정적 메서드가 있다고 할때, `Buy.pay()`와 `Sell.pay()`로 특정하여 호출할 수 있기 때문에 문제가 되지 않습니다.
- 그러나 **디폴트 메서드**는 인스턴스를 생성해야 호출할 수 있는 메서드이기 때문에 다음처럼 **이름이 같은 디폴트 메서드가 두 인터페이스에 있으면 문제가 됩니다.**

day08_10/interfaceex/Buy.java

```

package day08_10.interfaceex;

public interface Buy {
    void buy();

    default void order() {
        System.out.println("구매 주문");
    }
}

```



day08_10/interfaceex/Sell.java

```
package day08_10.interfaceex;

public interface Sell {
    void sell();

    default void order() {
        System.out.println("판매 주문");
    }
}
```



- 위 오류 메시지는 **디폴트 메서드가 중복되었으니 두 인터페이스를 구현하는 Customer 클래스에서 재정의하라**는 뜻입니다.

day08_10/interfaceex/Customer.java

```
package day08_10.interfaceex;

public class Customer implements Buy, Sell {
    ...
    // 디폴트 메서드 order()를 Customer 클래스에서 재정의함
    @Override
    public void order() {
        System.out.println("고객 판매 주문");
    }
}
```



- Customer 클래스에서 **디폴트 메서드를 재정의**하면, Customer 클래스를 생성하여 사용할때 재정의된 메서드가 호출됩니다.

day08_10/interfaceex/CustomerTest.java

```
package day08_10.interfaceex;

public class CustomerTest {
    public static void main(String[] args) {
        Customer customer = new Customer();

        Buy buyer = customer;
        buyer.buy();
        buyer.order(); // 재정의된 메서드 호출됨

        Sell seller = customer;
        seller.sell();
        seller.order(); // 재정의된 메서드 호출됨

        if (seller instanceof Customer) {
```



```

        Customer customer2 = (Customer)seller;
        customer2.buy();
        customer2.sell();
        customer2.order(); // 재정의된 메서드 호출됨
    }
}

```

실행결과

```

구매하기
고객 판매 주문
판매하기
고객 판매 주문
구매하기
판매하기
고객 판매 주문

```

- customer가 Buy형으로 변환되고 buyer.order()를 호출하면 Buy에 구현한 디폴트 메서드가 아닌 Customer 클래스에 재 정의한 메서드가 호출됩니다.
- 자바 가상메서드 원리와 동일합니다.

인터페이스 상속하기

- 인터페이스 간 o 네도 상속이 가능합니다.
- 인터페이스 간 상속은 구현 코드를 통해 기능을 상속하는 것이 아니므로 **형 상속** (type inheritance)이라고 부릅니다.
- **클래스의 경우에는 하나의 클래스만 상속받을 수 있지만, 인터페이스는 여러 개를 동시에 상속받을 수 있습니다.**
- 한 인터페이스가 여러 인터페이스를 상속받으면, 상속받은 인터페이스는 **상위 인터페이스에 선언한 추상 메서드를 모두 가지게 됩니다.**

day08_10/interfaceex/X.java

```

package day08_10.interfaceex;

public interface X {
    void x();
}

```



day08_10/interfaceex/Y.java

```

package day08_10.interfaceex;

public interface Y {

```



```
        void y();  
    }  
}
```

day08_10/interfaceex/MyInterface.java

```
package day08_10.interfaceex;  
  
public interface MyInterface extends X, Y {  
    void myMethod();  
}
```



day08_10/interfaceex/MyClass.java

```
package day08_10.interfaceex;  
  
public class MyClass implements MyInterface {  
  
    // X 인터페이스에서 상속받은 x() 메서드 구현  
    @Override  
    public void x() {  
        System.out.println("x()");  
    }  
  
    // Y 인터페이스에서 상속받은 y() 메서드 구현  
    @Override  
    public void y() {  
        System.out.println("y()");  
    }  
  
    @Override  
    public void myMethod() {  
        System.out.println("myMethod()");  
    }  
}
```



day08_10/interfaceex/MyClassTest.java

```
package day08_10.interfaceex;  
  
public class MyClassTest {  
    public static void main(String[] args) {  
        MyClass mClass = new MyClass();  
  
        // 상위 인터페이스 X형으로 대입하면  
        // X에 선언한 메서드만 호출 가능  
        X xClass = mClass;  
        xClass.x();  
    }  
}
```




```

        // 상위 인터페이스 Y형으로 대입하면
        // Y에 선언한 메서드만 호출 가능
        Y yClass = mClass;
        yClass.y();

        // 구현할 인터페이스형 변수에 대입하면
        // 인터페이스가 상속한 모든 메서드 호출 가능
        MyInterface iClass = mClass;
        iClass.myMethod();
        iClass.x();
        iClass.y();
    }
}

```

실행결과

```

x()
y()
myMethod()
x()
y()

```

인터페이스 구현과 클래스 상속 함께 쓰기

- 생성한 클래스는 상위 인터페이스 형으로 변환할 수 있습니다.
- 다만 상위 인터페이스로 형 변환을 하면 인터페이스에 선언한 메서드만 호출할 수 있습니다.
- 예제를 보면 mClass가 MyClass로 생성되었어도, X 인터페이스형으로 선언된 xClass에 대입되면 xClass가 호출할 수 있는 메서드는 X의 메서드인 x() 뿐입니다.
- 인터페이스를 정의할 때 기능상 계층구조가 필요한 경우 상속을 사용하기도 합니다.

인터페이스 구현과 클래스 상속 함께 쓰기

- 한 클래스에서 클래스 상속과 인터페이스 구현을 모두 할 수 도 있습니다.

day08_10/interfaceex/Shop.java

```

package day08_10.interfaceex;

public class Shop {
    public void shopInfo() {
        System.out.println("shopInfo()");
    }
}

```



```
package day08_10.interfaceex;

public class Customer extends Shop implements Buy, Sell {
    ...
}
```



실무에서 인터페이스를 사용하는 경우

- 인터페이스는 클래스가 제공할 기능을 선언하고 설계하는 것입니다.
- 만약 여러 클래스가 같은 메서드를 서로 다르게 구현한다면, 우선 인터페이스에 메서드를 선언한 다음 인터페이스를 구현한 각 클래스에서 같은 메서드에 대해 다양한 기능을 구현하면 됩니다.
- 이것이 바로 인터페이스를 이용한 다형성의 구현입니다.
- 인터페이스를 잘 정의하는 것이 확장성 있는 프로그램을 만드는 시작입니다.

내부 클래스

내부 클래스 정의와 유형

- 내부 클래스(inner class)는 말 그대로 **클래스 내부에 선언한 클래스**입니다.
- 내부에 클래스를 선언하는 이유는 대개 이 클래스와 외부 클래스가 밀접한 관련이 있기 때문입니다.
- 또한 그 밖의 다른 클래스와 협력할 일이 없는 경우에 내부 클래스를 선언해서 사용합니다.

```
class ABC { // 외부 클래스
    class In { // 인스턴스 내부 클래스
    }

    static class Sin { // 정적 내부 클래스
    }

    public void abc() {
        class Local { // 지역 내부 클래스
        }
    }
}
```



인스턴스 내부 클래스

- 인스턴스 내부 클래스(Instance Inner Class)는 인스턴스 변수를 선언할 때와 같은 위치에 선언하며, 외부 클래스 내부에서만 생성하여 사용하는 객체를 선언할 때 씁니다.
- 인스턴스 내부 클래스는 외부 클래스 생성 후 생성됩니다.
- 따라서 외부 클래스를 먼저 생성하지 않고 인스턴스 내부 클래스를 사용할 수는 없습니다. 이는 이후 설명하는 정적 내부 클래스와 다른 점입니다.

day08_10/innerclass/InnerTest.java

```
package day08_10.innerclass;

class OutClass { // 외부 클래스
    private int num = 10; // 외부 클래스 private 변수
    private static int sNum = 20; // 외부 클래스 정적 변수

    private InClass inClass; // 내부 클래스 자료형 변수를 먼저 선언

    // 외부 클래스 디폴트 생성자, 외부 클래스가
    // 생성된 후 내부 클래스 생성 가능
    public OutClass() {
        inClass = new InClass();
    }

    class InClass { // 인스턴스 내부 클래스
        int inNum = 100; // 내부 클래스의 인스턴스 변수
        // JDK15까지는 인스턴스 내부 클래스에 정적변수 선언이 불가능, JDK16부터는 가능
        //static int sInNum = 200;

        void inTest() {
            System.out.println("OutClass num = " + num + "(외부 클래스의 인스턴스 변수)");
            System.out.println("OutClass sNum = " + sNum + "(외부 클래스의 정적 변수)");
        }

        //JDK15까지는 인스턴스 내부 클래스에 정적메서드 선언이 불가능, JDK16부터는 가능
        //static void sTest() {
        //}

        public void usingClass() {
            inClass.inTest();
        }
    }
}
```



```

public class InnerTest {
    public static void main(String[] args) {
        OutClass outClass = new OutClass();
        System.out.println("외부 클래스 이용하여 내부 클래스 기능
호출");
        outClass.usingClass(); // 내부 클래스 기능 호출
    }
}

```

실행 결과

외부 클래스 이용하여 내부 클래스 기능 호출
 OutClass num = 10(외부 클래스의 인스턴스 변수)
 OutClass sNum = 20(외부 클래스의 정적 변수)

- 외부 클래스를 먼저 생성해야 내부 클래스를 사용할 수 있습니다.
- 클래스의 생성과 상관없이 사용할 수 있는 정적 변수는 인스턴스 내부 클래스에서 선언할 수 없습니다(JDK15버전까지, JDK16버전 이후 가능)
- 마찬가지로 이유로 정적 메서드도 인스턴스 내부 클래스에서 선언할 수 없습니다.

정적 내부 클래스

- 내부 클래스가 외부 클래스 생성과 무관하게 사용할 수 있어야 하고, 정적 변수도 사용할 수 있어야 한다면 정적 내부 클래스를 사용하면 됩니다.
- 정적 내부 클래스는 인스턴스 내부 클래스처럼 외부 클래스의 멤버 변수와 같은 위치에 정의하며 **static 예약어**를 함께 사용합니다.

day08_10/innerclass/InnerTest.java

```

package day08_10.innerclass;

class OutClass { // 외부 클래스
    private int num = 10; // 외부 클래스 private 변수
    private static int sNum = 20; // 외부 클래스 정적 변수

    private InClass inClass; // 내부 클래스 자료형 변수를 먼저 선언

    static class InStaticClass { // 정적 내부 클래스
        int inNum = 100; // 정적 내부 클래스의 인스턴스 변수
        static int sInNum = 200; // 정적 내부 클래스의 정적 변수

        // 정적 내부 클래스의 일반 메서드
        void inTest() {
            //num += 10; // 외부 클래스의 인스턴스 변수는 사용
            System.out.println("InStaticClass inNum = " +
inNum + "(내부 클래스의 인스턴스 변수 사용)");
            System.out.println("InStaticClass sInNum = " +

```



```

sInNum + "(내부 클래스의 정적 변수 사용)");
        System.out.println("OutClass sNum = " + inNum +
"(외부 클래스의 정적 변수 사용)");
    }

    // 정적 내부 클래스의 정적 메서드
    static void sTest() {
        // 외부 클래스와 내부 클래스의 인스턴스 변수는 사용
할 수 없다.

        //num += 10;
        //inNum += 10
        System.out.println("OutClass sNum = " + sNum + "
(외부 클래스의 정적 변수 사용)");
        System.out.println("InStaticClass sInNum = " +
sInNum + "(내부 클래스의 정적 변수 사용)");

    }
}
...
}

public class InnerTest {
    public static void main(String[] args) {
        ...
        OutClass.InStaticClass sInClass = new
OutClass.InStaticClass();
        System.out.println("정적 내부 클래스 일반 메서드 호출");
        sInClass.inTest();
        System.out.println();
        System.out.println("정적 내부 클래스의 정적 메서드 호출");
        OutClass.InStaticClass.sTest();
    }
}

```

실행결과

```

외부 클래스 이용하여 내부 클래스 기능 호출
OutClass num = 10(외부 클래스의 인스턴스 변수)
OutClass sNum = 20(외부 클래스의 정적 변수)
정적 내부 클래스 일반 메서드 호출
InStaticClass inNum = 100(내부 클래스의 인스턴스 변수 사용)
InStaticClass sInNum = 200(내부 클래스의 정적 변수 사용)
OutClass sNum = 100(외부 클래스의 정적 변수 사용)

```

```

정적 내부 클래스의 정적 메서드 호출
OutClass sNum = 20(외부 클래스의 정적 변수 사용)
InStaticClass sInNum = 200(내부 클래스의 정적 변수 사용)

```

- 정적 메서드에서는 인스턴스 변수를 사용할 수 없습니다. 따라서 정적 내부 클래스에서도 외부 클래스의 인스턴스 변수는 사용할 수 없습니다.

- 내부 클래스를 만들고 외부 클래스와 무관하게 다른 클래스에서도 사용하려면 정적 내부 클래스를 생성하면 됩니다.
- 하지만 정적 내부 클래스를 private으로 선언했다면 이것 역시 다른 클래스에서 사용할 수 없습니다.

정적 내부 클래스 메서드	변수 유형	사용 가능 여부
일반 메서드 void inTest()	외부 클래스의 인스턴스 변수(num)	X
	외부 클래스의 정적변수(sNum)	O
	정적 내부 클래스의 인스턴스 변수(inNum)	O
	정적 내부 클래스의 정적 변수(sInNum)	O
정적 메서드 static void sTest()	외부 클래스의 인스턴스 변수(num)	X
	외부 클래스의 정적변수(sNum)	O
	정적 내부 클래스의 인스턴스 변수(inNum)	X
	정적 내부 클래스의 정적 변수(sInNum)	O

지역 내부 클래스

- 지역 내부 클래스는 지역 변수 처럼 메서드 내부에 클래스를 정의하여 사용하는 것을 말합니다.
- 따라서 이 클래스는 메서드 안에서만 사용할 수 있습니다.

day08_10/innerclass/LocalInnerTest.java

```
package day08_10.innerclass;

class Outer {
    int outNum = 100;
    static int sNum = 200;

    Runnable getRunnable(int i) {
        int num = 100; // 지역변수

        class MyRunnable implements Runnable { // 지역 내부 클래스
            int localNum = 10; // 지역 내부 클래스의 인스턴스 변수

            @Override
```



```

        public void run() {
            //num = 200; 지역변수는 상수로 바뀌므로 값을
            //변경할 수 없어 오류 발생
            //i = 100; // 매개변수도 지역변수처럼 상수
            //로 바뀌므로 값을 변경할 수 없어 오류 발생

            System.out.println("i = " + i);
            System.out.println("num = " + num);
            System.out.println("localNum = " +
            localNum);

            System.out.println("outNum = " + outNum
            + "(외부 클래스 인스턴스 변수)");
            System.out.println("Outer.sNum = " +
            Outer.sNum + "(외부 클래스 정적 변수)");
        }

        return new MyRunnable();
    }
}

public class LocalInnerTest {
    public static void main(String[] args) {
        Outer out = new Outer();
        Runnable runner = out.getRunnable(10); // 메서드 호출
        runner.run();
    }
}

```

실행결과

```

i = 10
num = 100
localNum = 10
outNum = 100(외부 클래스 인스턴스 변수)
Outer.sNum = 200(외부 클래스 정적 변수)

```

- 메서드 안에 정의한 MyRunnable 클래스가 바로 지역 내부 클래스입니다.
- LocalInnerTest 클래스의 Outer 클래스를 생성한 후 Runnable형 객체로 getRunnable()을 호출합니다.
- 즉 MyRunnable 메서드 호출을 통해 생성된 객체를 반환받아야 합니다.

지역 내부 클래스에서 지역변수 유효성

- 지역 변수는 메서드가 호출될 때 스택 메모리에서 생성되고 메서드의 수행이 끝나면 메모리에서 사라집니다.
- 지역 내부 클래스에 포함된 getRunnable() 메서드의 매개변수 i와 메서드 내부에 선언한 변수 num은 지역변수입니다.

```
Outer out = new Outer();
Runnable runner = out.getRunnable(10); // getRunnable() 메서드의 호출이
// 끝남
runner.run(); // run()이 실행되면서 getRunnable() 메서드의 지역 변수가 사용
// 됨
```



- run() 메서드는 getRunnable() 메서드의 지역 변수 i와 num을 사용합니다.
- 그런데 지역 내부 클래스를 가지고 있는 getRunnable() 메서드 호출이 끝난 후에도 run() 메서드가 정상적으로 호출됩니다.
- 이는 getRunnable() 메서드 호출이 끝나고 스택 메모리에서 지워진 변수를 이후에 또 참조 할 수 있다는 것 입니다.
- 즉, 지역 내부 클래스에서 사용하는 지역 변수는 상수로 처리됩니다.
- 상수로 처리하기 위해 JDK1.7까지는 final 예약어를 꼭 함께 써야 했으나, JDK1.8 부터는 직접 써 주지 않아도 코드를 컴파일 하면 final 예약어가 자동으로 추가 됩니다.
- 그러므로 num과 i 변수 값을 다른 값으로 바꾸려고 하면 오류가 발생합니다.
- 지역 내부 클래스에서 사용하는 메서드의 지역변수는 모두 상수로 바뀝니다.

익명 내부 클래스

- 클래스 이름을 사용하지 않는 클래스가 있습니다. 이런 클래스를 익명 클래스라고 부릅니다.

day08_10/innerclass/AnonymousInnerTest.java

```
package day08_10.innerclass;

class Outer2 {
    Runnable getRunnable(int i) {
        int num = 100;

        return new Runnable() { // 익명 내부 클래스 Runnable 인터
// 페이스 생성
            @Override
            public void run() {
                // 지역변수는 상수화 되므로 변경 불가
                //num = 200;
                //i = 10;

                System.out.println(i);
                System.out.println(num);
            }
        }; // 클래스 끝에 ;를 씀
    }

    Runnable runner = new Runnable() { // 익명 내부 클래스를 변수에 대
// 입
```




```

        @Override
        public void run() {
            System.out.println("Runnable의 구현된 익명 클래스
변수");
        }
    }; // 클래스 끝에 ;를 씀
}

```

```

public class AnonymousInnerTest {
    public static void main(String[] args) {
        Outer2 out = new Outer2();
        Runnable runnable = out.getRunnable(10);
        runnable.run();
        out.runner.run();
    }
}

```

실행결과

```

10
100

```

Runnable의 구현된 익명 클래스 변수

- 익명 내부 클래스는 단 하나의 인터페이스 또는 단 하나의 추상 클래스를 바로 생성할 수 있습니다.
- Runnable 인터페이스를 생성할 수 있으려면 인터페이스 몸체가 필요합니다.
- Runnable 인터페이스에서 반드시 구현해야 하는 run()메서드가 포함되어 있습니다. 마지막에 세미콜론(;)을 사용해서 익명 내부 클래스가 끝났다는 것을 알려줍니다.
- 정리하면 익명 내부 클래스는 변수에 직접 대입하는 경우도 있고 메서드 내부에서 인터페이스나 추상 클래스를 구현하는 경우도 있습니다.
- 이때 사용하는 지역변수는 상수화되므로 메서드 호출이 끝난 후에도 사용할 수 있습니다.

종류	구현위치	사용할 수 있는 외부 클래스 변수	생성방법
인스턴스 내부 클래스	외부 클래스 멤버 변수와 동일	외부 인스턴스 변수 외부 전역 변수	외부 클래스를 먼저 만든 후 내부 클래스 생성
정적 내부 클래스	외부 클래스 멤버 변수와 동일	외부 전역 변수	외부 클래스와 무관하게 생성

종류	구현위치	사용할 수 있는 외부 클래스 변수	생성방법
지역 내부 클래스	메서드 내부에 구현	외부 인스턴스 변수 외부 전역 변수	메서드를 호출할 때 생성
익명 내부 클래스	메서드 내부에 구현 변수에 대입하여 직접 구현	외부 인스턴스 변수 외부 전역 변수	메서드를 호출할 때 생성되거나, 인터페이스 타입 변수에 대입할 때 new 예약어를 사용하여 생성