함께가요 미래로!
Enabling People

OOP - 3

목차

abstract class

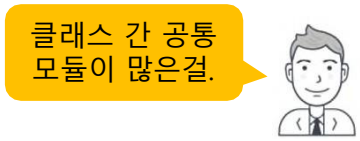
추상 클래

❖ 추상 클래스 정의

- 아래 클래스들의 공통 분모를 뽑아서 상속 구조를 만들자.

```
public class DieselSUV {
  private int curX, curY;

  public void reportPosition() {
    System.out.printf("현재 위치: (%d, %d)%n", curX, curY);
  }

  public void addFuel() {
    System.out.printf("주유소에서 급유");
  }
}
```

```
public class ElectricCar {
  private int curX, curY;

  public void reportPosition() {
    System.out.printf("현재 위치: (%d, %d)%n", curX, curY);
  }

  public void addFuel() {
    System.out.printf("급속 충전");
  }
}
```



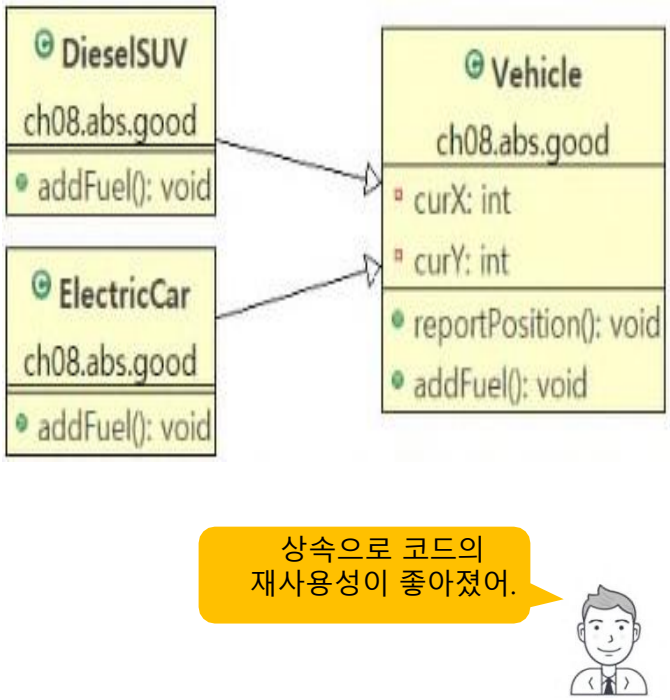
추상 클래

Confidential

❖ 추상 클래스 정의

- 상속 관계 정의를 통한 클래스 정비

```
class Vehicle {
  private int curX, curY;

  public void reportPosition() {
    System.out.printf("현재 위치: (%d, %d)%n", curX, curY);
  }

  public void addFuel() {
    System.out.println("모든 운송 수단은 연료가 필요");
  }
}

class DieselSUV extends Vehicle {
  @Override
  public void addFuel() {
    System.out.println("주유소에서 급유");
  }
}

class ElectricCar extends Vehicle {
  @Override
  public void addFuel() {
    System.out.println("급속 충전");
  }
}
```



추상 클래

❖ 추상 클래스 정의

- 상속 관계 정의를 통한 클래스 정비

```
class Vehicle {
  private int curX, curY;

  public void reportPosition() {
    System.out.printf("현재 위치: (%d, %d)%n", curX, curY);
  }

  public void addFuel() {
    System.out.println("모든 운송 수단은 연료가 필요");
  }
}

class DieselSUV extends Vehicle {
  @Override
  public void addFuel() {
    System.out.println("주유소에서 급유");
  }
}

class ElectricCar extends Vehicle {
  @Override
  public void addFuel() {
    System.out.println("급속 충전");
  }
}
```

```
public class VehicleTest {

  public static void main(String[] args) {
    Vehicle[] vehicles = {
                          new DieselSUV(),
                          new ElectricCar()
                         };
    for (Vehicle v : vehicles) {
      v.addFuel();
      v.reportPosition();
    }
  }
}
```

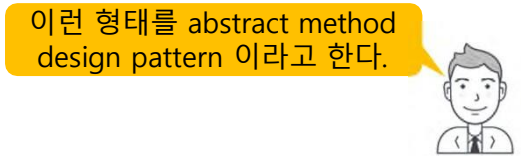
추상 클래

Confidential

❖ 추상 클래스 정의

- DiesselSUV, ElectricCar는 모두 연료가 필요하므로 addFuel은 공통 모듈
  - ◆ 조상 클래스인 Vehicle에 정리하고 각 자손 클래스에서 override 예정
  - ◆ Vehicle에서 힘들게 구현했지만 아무도 Vehicle의 addFuel()에 신경 쓰지 않는다.
    - addFuel()을 Vehicle에서 지우면?
  - ◆ 자손 클래스에서 반드시 재정의해서 사용되기 때문에 조상의 구현이 무의미한 메서드
    - 메서드의 선언부만 남기고 구현부는 세미콜른(;) 으로 대체
    - 구현부가 없다는 의미로 abstract 키워드를 메서드 선언부에 추가
    - 객체를 생성할 수 없는 클래스라는 의미로 클래스 선언부에 abstract를 추가한다.

```
abstract class  Vehicle {
  private int curX, curY;

  public void reportPosition() {
    System.out.printf("현재 위치: (%d, %d)%n", curX, curY);
  }

  public abstract void addFuel();
}
```



추상 클래

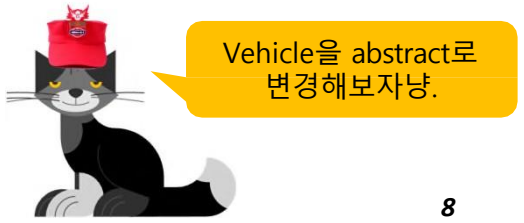
Confidential

❖ 추상 클래스의 특징

- abstract 클래스는 상속 전용의 클래스
  - ◆ 클래스에 구현부가 없는 메서드가 있으므로 객체를 생성할 수 없음
  - ◆ 하지만 상위 클래스 타입으로써 자식을 참조할 수는 있다.

```
// Vehicle v = new Vehicle(); // abstract 클래스는 객체를 생성할 수 없다.

Vehicle v = new DieselSUV();   // 자식을 참조하는 것은 문제 없음
```

- 조상 클래스에서 상속받은 abstract 메서드를 재정의 하지 않은 경우
  - ◆ 클래스 내부에 abstract 메서드가 있는 상황이므로 자식 클래스는 abstract 클래스로 선언되어야 함

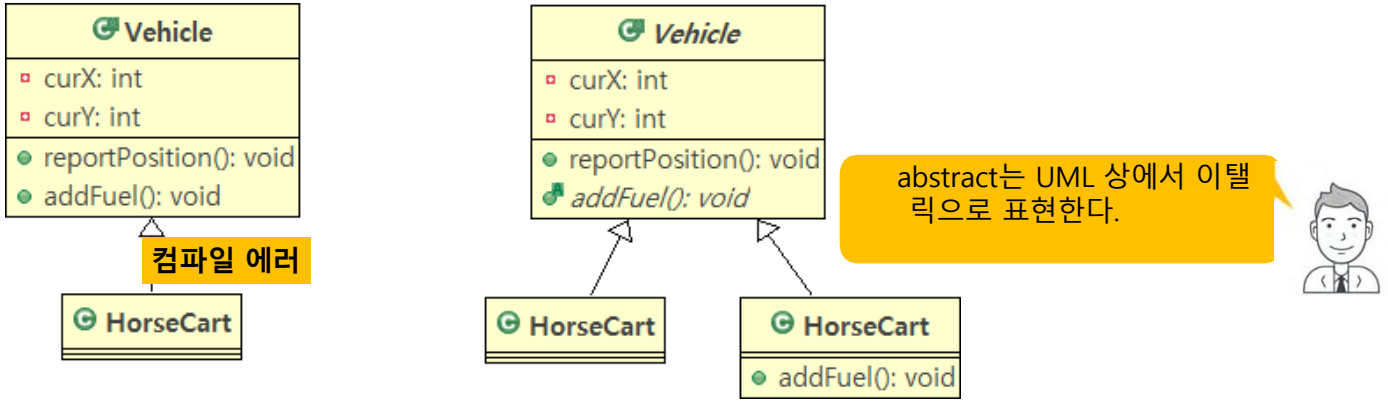


추상 클래

Confidential

❖ 추상 클래스를 사용하는 이유

- abstract 클래스는 구현의 강제를 통해 프로그램의 안정성 향상



Vehicle의 addFuel() 그대로 사용

반드시 addFuel() 재정의 필요

```
//The type HorseCart must implement the inherited abstract method Vehicle.addFuel()
class HorseCart extends Vehicle{}
```

- interface에 있는 메서드 중 구현할 수 있는 메서드를 구현해 개발의 편의 지원

함께가요 미래로!
Enabling People

Interface

인터페이

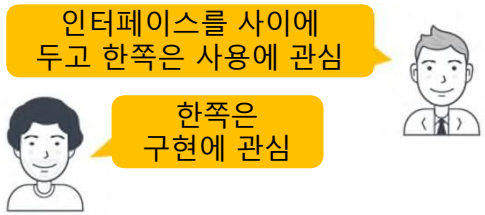
Confidential

❖ 인터페이스란?

[국어사전]

인터페이스 (interface)

1 서로 다른 두 시스템, 장치, 소프트웨어 따위를 서로 이어 주는 부분. 또는 그런 접속 장치.



- GUI – Graphic User Interface
  - ◆ 프로그램과 사용자 사이의 접점

누르면 → **실행:호출**

저장 → **결과: 리턴**

되겠지? → **동작: 메서드**

선언부만 필요

How?.. 는 궁금치 않다!

```
public abstract void save();
```

누르겠지? 그럼 저장 해야지!

```
public void save(){
  //1. 변경 정보 확인
  //2. OutputStream을 통해 내용 저장
  //3. 화면상에 변경 완료 표시
}
```

인터페이

❖ 인터페이스 작성

- 최고 수준의 추상화 단계 : 모든 메서드가 abstract 형태
  - ◆ JDK 8에서 default method와 static method 추가
- 형태
  - ◆ 클래스와 유사하게 interface 선언
  - ◆ 맴버 구성
    - 모든 멤버변수는 public static final 이며 생략 가능
    - 모든 메서드는 public abstract 이며 생략 가능

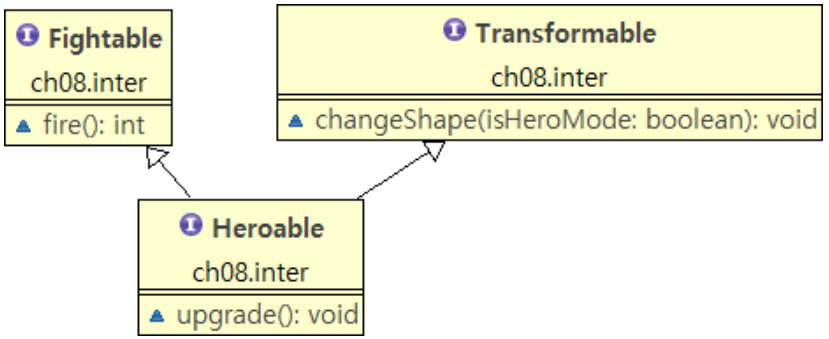
```
public interface MyInterface {
  public static final int MEMBER1 = 10;
  int MEMBER2 = 10;

  public abstract void method1(int param);
  void method2(int param);
}
```

인터페이

❖ 인터페이스 상속

- 클래스와 마찬가지로 인터페이스도 extends를 이용해 상속이 가능
- 클래스와 다른 점은 인터페이스는 다중 상속이 가능
  - ◆ 헷갈릴 메서드 구현 자체가 없다.

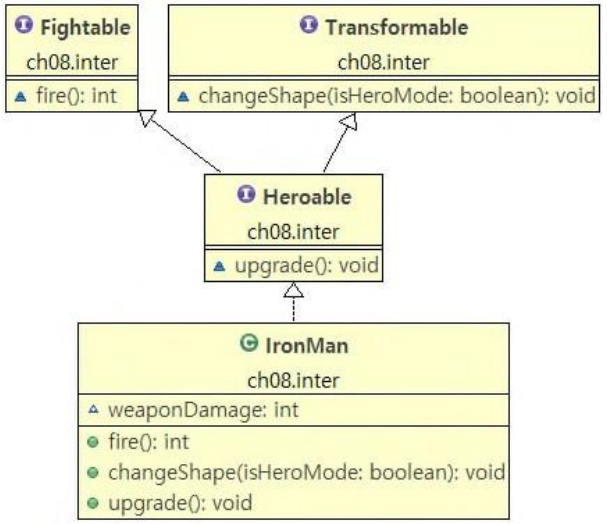
```
interface Fightable{
  int fire();
}

interface Transformable{
  void changeShape(boolean isHeroMode);
}

public interface Heroable extends Fightable, Transformable{
  void upgrade();
}
```



인터페이

❖ 인터페이스 구현과 객체 참조

- 클래스에서 implements 키워드를 사용해서 interface 구현
- implements 한 클래스는
  - ◆ 모든 abstract 메서드를 override해서 구현하거나
  - ◆ 구현하지 않을 경우 abstract 클래스로 표시해야 함
- 여러 개의 interface implements 가능




```
public class IronMan implements Heroable {
  int weaponDamage = 100;
  @Override
  public int fire() {
    System.out.printf("빔 발사: %d만큼의 데미지를 가함%n");
    return this.weaponDamage;
  }
  @Override
  public void changeShape(boolean isHeroMode) {
    String status = isHeroMode?"장착":"제거";
    System.out.printf("장갑 %s%n", status);
  }
  @Override
  public void upgrade() {
    System.out.printf("무기 성능 개선" );
  }
}
```

```
SpiderMan4 extends Person implements Heroable{
The type SpiderMan4 must implement the inherited abstract method Heroable.upgrade()
2 quick fixes available:
  Add unimplemented methods
  Make type 'SpiderMan4' abstract
                                                    Press 'F2' for focus
```

인터페이

❖ 인터페이스 구현과 객체 참조

- 다형성은 조상 클래스 뿐 아니라 조상 인터페이스에도 적용

```
public class IronManTest {

  public static void main(String[] args) {
    IronMan iman = new IronMan();
    Object obj = iman;
    Heroable hero= iman;
    Fightable fight = iman;
    Transformable trans = iman;
  }
}
```

인터페이

❖ 인터페이스의 필요성

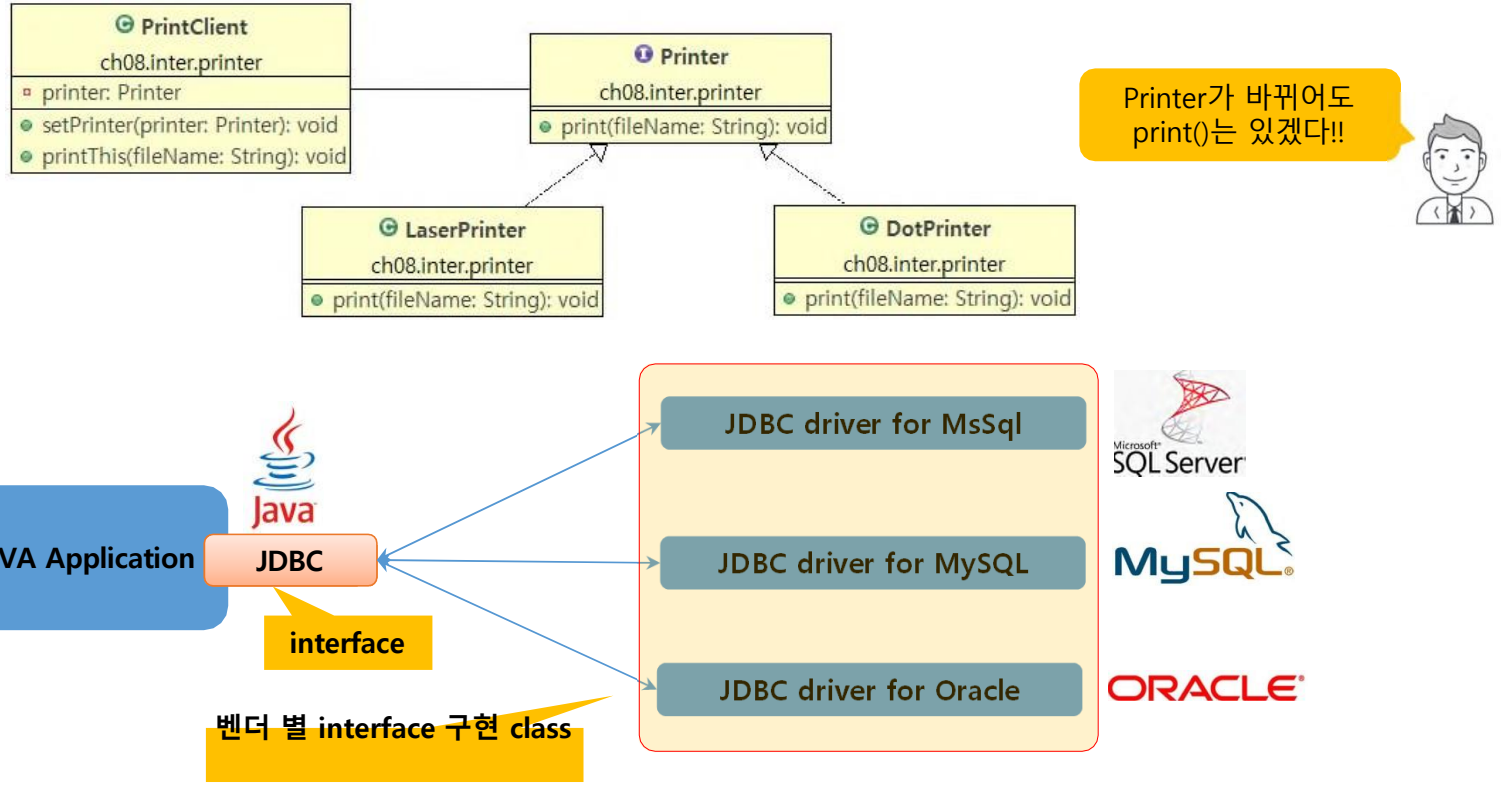
- 구현의 강제로 표준화 처리
  - ◆ abstract 메서드 사용
- 인터페이스를 통한 간접적인 클래스 사용으로 손쉬운 모듈 교체 지원
- 서로 상속의 관계가 없는 클래스들에게 인터페이스를 통한 관계 부여로 다형성 확장
- 모듈 간 독립적 프로그래밍 가능 → 개발 기간 단축

인터페이

Confidential

❖ 인터페이스의 필요성

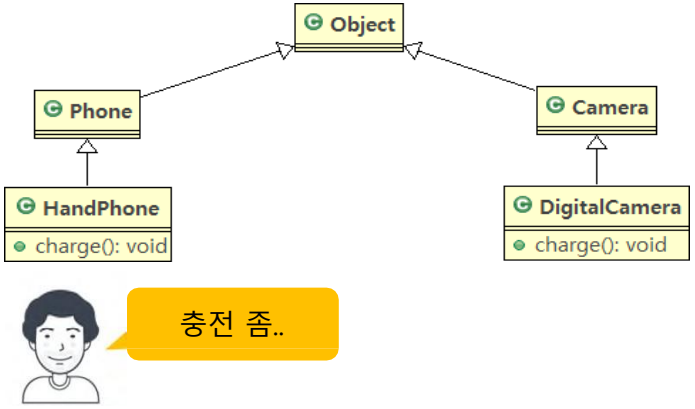
● 구현의 강제로 표준화 처리 → 손쉬운 모듈 교체 지원



인터페이

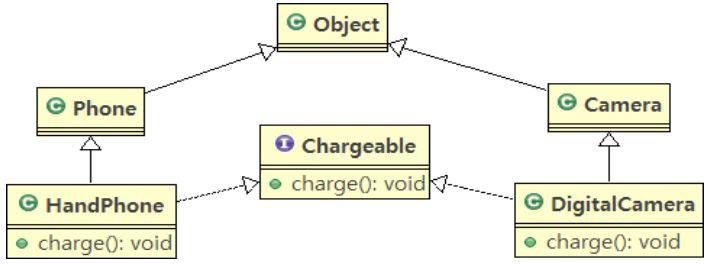
Confidential

❖ 서로 상속의 관계가 없는 클래스들에게 인터페이스를 통한 관계 부여로 다형성 확장




```
void badCase() {
  Object [] objs = {
                      new HandPhone(),
                      new DigitalCamera()
                    };
  for(Object obj: objs) {
    if(obj instanceof HandPhone) {
      HandPhone phone = (HandPhone)obj;
      phone.charge();
    }else if(obj instanceof DigitalCamera) {
      DigitalCamera camera = (DigitalCamera)obj;
      camera.charge();
    }
  }
}
```

충전하기 어렵다..ㅜㅜ




```
void goodCase() {
  Chargeable [] objs = {
                       new HandPhone(),
                       new DigitalCamera()
                      };
  for(Chargeable obj: objs) {
    obj.charge();
  }
}
```

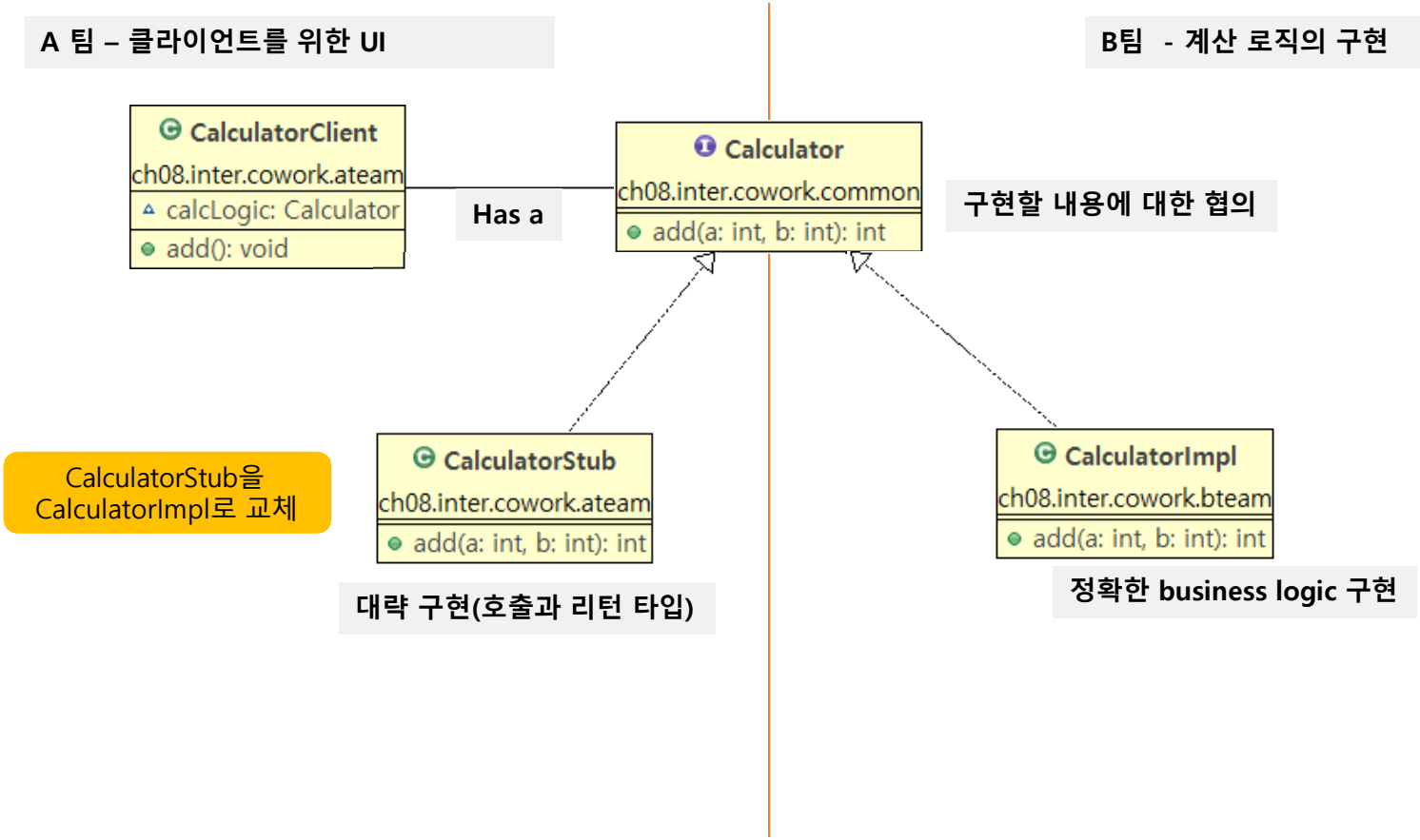
충전 부탁한다냥.

인터페이

Confidential

❖ 독립적인 프로그래밍으로 개발 기간 단축

● 계산기를 구현하는 두 팀의 작업



인터페이

❖ 독립적인 프로그래밍으로 개발 기간 단축

```
public interface Calculator {
  int add(int a, int b);
}
```

공통 비즈니스 로직

```
class CalculatorStub implements Calculator {
  public int add(int a, int b) {
    System.out.printf("파라미터 확인: %d, %d%n", a, b);
    return 0;
  }
}
class CalculatorClient{
  Calculator calcLogic = new CalculatorStub();
  public void add() {
    System.out.println("첫 번째 정수를 입력하시오.");
    Scanner scanner = new Scanner(System.in);
    int a = scanner.nextInt();
    System.out.println("두 번째 정수를 입력하시오.");
    int b = scanner.nextInt();
    System.out.printf("결과: %d+%d=%d%n", a, b, calcLogic.add(a, b));
  }
}
```

A 팀 – 대충 구현된 Stub

A 팀 – 클라이언트를 위한 UI

```
public class CalculatorImpl implements Calculator{
  public int add(int a, int b) {
    System.out.printf("파라미터 확인: %d, %d%n", a, b);
    return a + b;
  }
}
```

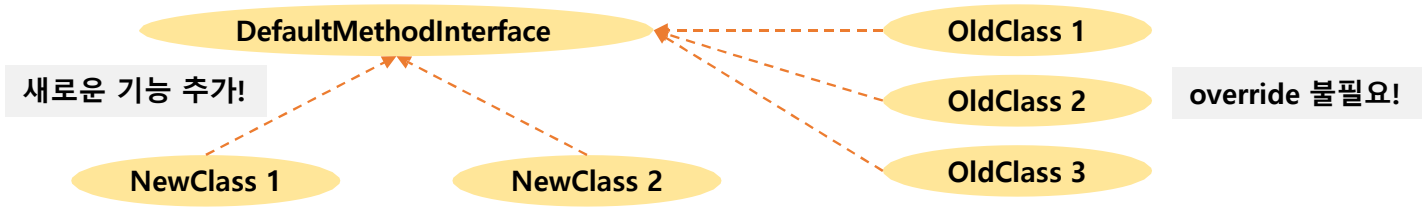
B팀 - 계산 로직의 구현

❖ default method

- 인터페이스에 선언 된 구현부가 있는 일반 메서드
  - ◆ 메서드 선언부에 default modifier 추가 후 메서드 구현부 작성
    - 접근 제한자는 public으로 한정됨(생략 가능)

```
interface DefaultMethodInterface {
  void abstractMethod();

  default void defaultMethod() {
    System.out.println("이것은 기본 메서드입니다.");
  }
}
```

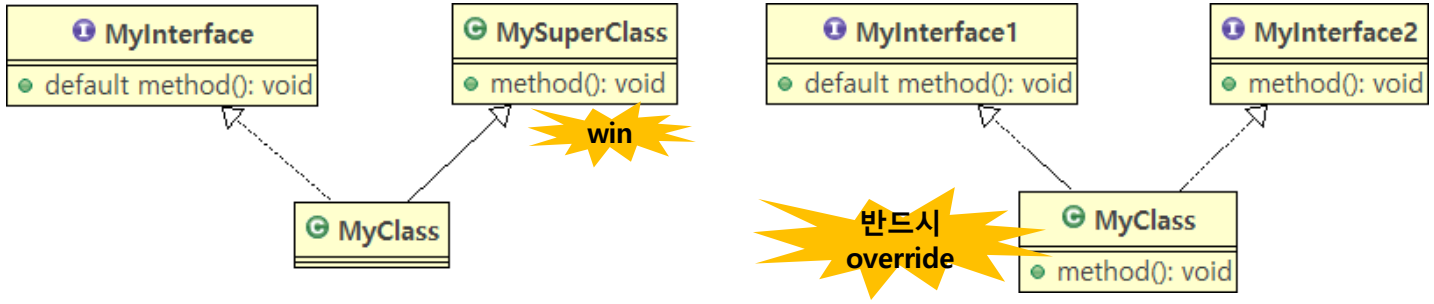
  - ◆ 필요성
    - 기존에 interface 기반으로 동작하는 라이브러리의 interface에 추가해야 하는 기능이 발생
    - 기존 방식으로라면 모든 구현체들이 추가되는 메서드를 override 해야 함
    - default 메서드는 abstract가 아니므로 반드시 구현 해야 할 필요는 없어짐



❖ default method

● default method의 충돌

- ◆ JDK 1.7 이하의 java에서는 interface method에 구현부가 없으므로 충돌이 없었음
- ◆ 1.8 부터 default method가 생기면서 동일한 이름을 갖는 구현부가 있는 메서드가 충돌
- ◆ method 우선 순위
  - super class의 method 우선 : super class가 구체적인 메서드를 갖는 경우 default method는 무시됨
  - interface간의 충돌 : 하나의 interface에서 default method를 제공하고 다른 interface에서도 같은 이름의 메서드(default 유무와 무관)가 있을 때 sub class는 반드시 override 해서 충돌 해결!!



❖ static method

- interface에 선언된 static method
  - ◆ 일반 static 메서드와 마찬가지로 별도의 객체가 필요 없음
  - ◆ 구현체 클래스 없이 바로 인터페이스 이름으로 메서드에 접근해서 사용 가능

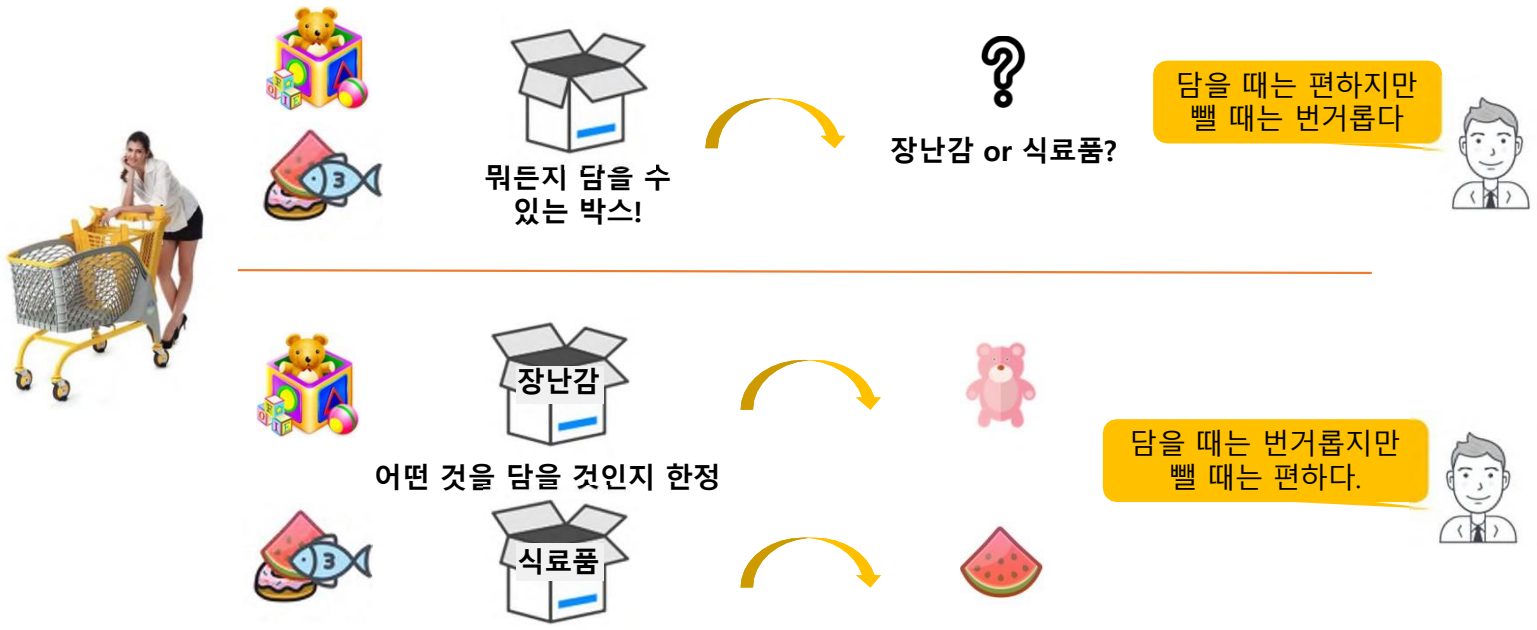
```
package ch08.inter.method;

interface StaticMethodInterface{
  static void staticMethod() {
    System.out.println("Static 메서드");
  }
}

public class StaticMethodTest {
  public static void main(String[] args) {
    StaticMethodInterface.staticMethod();
  }
}
```

함께가요 미래로!
Enabling People

Generic

❖ Generics

- 다양한 타입의 객체를 다루는 메서드, 컬렉션 클래스에서 컴파일 시에 타입 체크
  - ◆ 미리 사용할 타입을 명시해서 형 변환을 하지 않아도 되게 함
    - 객체의 타입에 대한 안전성 향상 및 형 변환의 번거로움 감소



❖ 표현

- 클래스 또는 인터페이스 선언 시 <>에 타입 파라미터 표시

```
public class Class_Name<T>{}
public interface Interface_Name<T>{}
```

  - ◆ Class_Name: Raw Type
  - ◆ Class_Name<T>: Generic Type
- 타입 파라미터
  - ◆ 특별한 의미의 알파벳 보다는 단순히 임의의 참조형 타입을 말함
  - ◆ T : reference Type, E : Element, K : Key, V : Value

```
public class ArrayList<E> extends AbstractList<E> implements List<E>, RandomAccess, Cloneable, java.io.Serializable{… }

public class HashMap<K,V> extends AbstractMap<K,V> implements Map<K,V>, Cloneable, Serializable {… }
```

- 객체 생성
  - ◆ 변수 쪽과 생성 쪽의 타입은 반드시 같아야 함

```
Class_Name<String> generic = new Class_Name<String>();
Class_Name<String> generic2 = new Class_Name<>();
Class_Name generic3 = new Class_Name();
```

```
Class_Name generic3 = new Class_Name();
Class_Name is a raw type. References to generic type Class_Name<T> should be parameterized
```

❖ 클래스 생성

```
class NormalBox{
  private Object some;

  public Object getSome() {
    return some;
  }

  public void setSome(Object some) {
    this.some = some;
  }
}
```

```
class GenericBox<T> {
  private T some;

  public T getSome() {
    return some;
  }

  public void setSome(T some) {
    this.some = some;
  }
}
```

❖ 사용

- 컴파일 타입에 타입 파라미터들이 대입된 타입으로 대체됨

```
public class NormalBoxTest {

  public static void main(String[] args) {
    NormalBox nBox1 = new NormalBox();
    nBox1.setSome("Hello");
    nBox1.setSome(new Toy());

    Object some = nBox1.getSome();

    if(some instanceof Toy) {
      Toy toy = (Toy)some;
      // toy 사용
    }else if(some instanceof Grocery) {
      Grocery grocery = (Grocery)some;
      // grocery 사용
    }else {
      System.out.println("알수 없음");
    }
  }
}
```

**Object를 파라미터로 사용 →
어떤 객체든지 수용 가능**

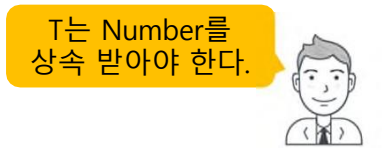
```
public class GenericBoxTest {

  public static void main(String[] args) {
    GenericBox<Toy2> gBox1 = new GenericBox<>();

    gBox1.setSome(new Toy2());

    // gBox1.setSome(new Grocery2());

    Toy2 toy = gBox1.getSome();
    // toy 사용

    GenericBox<Grocery2> gBox2 = new GenericBox<>();
    gBox2.setSome(new Grocery2());
    Grocery2 grocery = gBox2.getSome();
    // grocery 사용
  }
}
```

**무언가 T로 객체를 한정
→ T의 자식까지만 허용 됨**

❖ type parameter의 제한

- 필요에 따라 구체적인 타입 제한 필요
  - ◆ 계산기 프로그램 구현 시 Number 이하의 타입(Byte, Short, Integer…)로만 제한
    - type parameter 선언 뒤 extends 와 함께 상위 타입 명시

```
class NumberBox<T extends Number> {
  public void addSomes(T... ts) {
    double d = 0;
    for (T t : ts) {
      d += t.doubleValue();
    }
    System.out.println("총 합은: " + d);
  }
}
```




```
public class ExtendsTest {

  public static void main(String[] args) {
    NumberBox<Number> numBox = new NumberBox<>();
    numBox.addSomes(1.5, 5, 4L);

    NumberBox<Integer> intBox = new NumberBox<>();
    intBox.addSomes(1,2,3);

    //NumberBox<String> strBox = new NumberBox<>();
  }
}
```

  - ◆ 인터페이스로 제한할 경우도 extends 사용
  - ◆ 클래스와 함께 인터페이스 제약 조건을 이용할 경우 & 로 연결

```
class TypeRestrict1<T extends Cloneable>{}

class TypeRestrict2<T extends Number & Cloneable & Comparable<String>>{}
```

❖ Generic Type 객체를 할당 받을 때 와일드 카드(?) 이용

- generic type에서 구체적인 타입 대신 사용

표현	설명
Generic type<?>	타입에 제한이 없음
Generic type<? extends T>	T 또는 T를 상속받은 타입들만 사용 가능
Generic type<? super T>	T 또는 T의 조상 타입만 사용 가능

```
public class WildTypeTest {
  public void wildCardTest() {
    PersonBox<Object> pObj = new PersonBox<>();
    PersonBox<Person> pPer = new PersonBox<>();
    PersonBox<SpiderMan> pSpi = new PersonBox<>();

    PersonBox<?> pAll = pPer;
    pAll = pSpi;
    pAll = pObj;

    PersonBox<? extends Person> pChildPer = pPer;
    pChildPer = pSpi;
    //pChildPer = pObj;

    PersonBox<? super Person> pSuperPer = pPer;
    //pSuperPer = pSpi;
    pSuperPer = pObj;
  }
}
```

```
class Person {}
class SpiderMan extends Person {}
class PersonBox<T> {}
```

❖ Generic Method

- 파라미터와 리턴타입으로 type parameter를 갖는 메서드
  - ◆ 매서드 리턴 타입 앞에 타입 파라미터 변수 선언

```
[제한자] <타입_파라미터, […]> 리턴_타입 메서드_이름(파라미터){
  // do something
}
```

```
public class TypeParameterMethodTest<T> {
  T some;
  public TypeParameterMethodTest(T some){
    this.some = some;
  }
  public <P> void method1(P p) {
    System.out.println("클래스 레벨의 T"+some.getClass().getName());
    System.out.println("파라미터: " + p.getClass().getName());
  }

  public <P> P method2(P p) {
    return p;
  }

  public static void main(String[] args) {
    TypeParameterMethodTest<String> tpmt = new TypeParameterMethodTest<>("Hello");
    tpmt.method1(10);
    tpmt.<Long>method2(20L);
  }
}
```

객체 생성 시점에 T의 타입 결정

메서드 호출 시점에 P의 타입 결정

❖ 다음 메서드 선언을 읽어보고 사용해봅시다.

```
public interface List<E> extends Collection<E> {
    default void sort(Comparator<? super E> c) {
        ...
    }

    boolean addAll(int index, Collection<? extends E> c);
}


public class Collections {
    private static <T> T get(ListIterator<? extends T> i, int index) {
        . . .
    }

    public static <T> void copy(List<? super T> dest, List<? extends T> src) {
        . . .
    }
}
```