SAMSUNG SW ACADEMY FOR YOUTH

함께가요 미래로!
Enabling People

삼성 청년 SW 아카데미

Java

SAMSUNG SW ACADEMY FOR YOUTH

Programming Language

객체지향 프로그래밍

- 인터페이스
- Generic

SAMSUNG SW ACADEMY FOR YOUTH

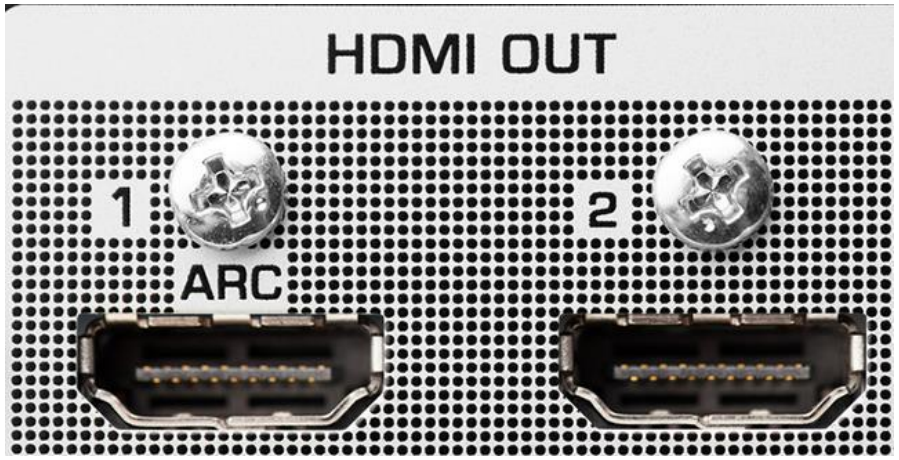
함께가요 미래로!
Enabling People

인터페이스
(Interface)

인터페이스 (Interface)

- 생각해 봅시다.

interface?



인터페이스 (Interface)

- 완벽히 추상화된 설계도 : 모든 메서드가 추상 메서드
- 모든 메서드가 public abstract이며 생략 가능
- 모든 멤버변수가 public static final이며 생략 가능
- interface 키워드를 이용하여 선언
- 클래스에서 해당 인터페이스를 implements 키워드를 이용하여 구현

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

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

인터페이스 (Interface)

- 1. interface 키워드를 이용하여 선언

```
public interface MyInterface { }
```

- 2. 선언되는 변수는 모두 상수로 적용

```
public static final int MEMBER1 = 10;
int MEMBER2 = 10;
```

- 3. 선언되는 메소드는 모두 추상 메소드(public abstract가 생략)

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

- 4. 인터페이스 extends를 이용하여 상속 가능 (다중 상속가능, 구현부가 없음.)

인터페이스 (Interface)

- 5. 객체 생성이 불가능 (추상클래스 동일한 특성)

```
interface MyInterface {}

public class Main {
    MyInterface m = new MyInterface();
}
```

- 6. 클래스가 인터페이스를 구현 할 경우에는 extends 키워드가 아니라 implements 키워드를 이용

```
interface Shape {}

class Circle extends Shape {}
```

```
class Circle implements Shape {}
```

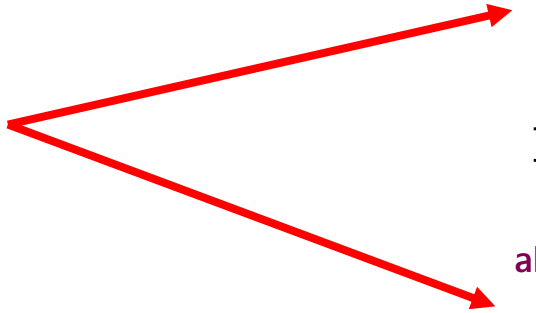
여러 개의 interface implements 가능

인터페이스 (Interface)

- 7. 인터페이스를 상속받는 하위클래스는 추상 메소드를 반드시 오버라이딩(재정의) 해야 한다.

  (구현하지 않을 경우 abstract 클래스로 표시해야 함)

```
interface Chef {
    void cook();
}
```



```
class KFoodChef implements Chef {

    @Override
    public void cook() {
        System.out.println("한식을 요리한다.");
    }
}
```

```
abstract class JFoodChef implements Chef {

}
```

- 8. 인터페이스 다형성 적용.

```
public static void main(String[] args) {
    Chef chef = new KFoodChef();
}
```

인터페이스 (Interface)

- 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

SAMSUNG SW ACADEMY FOR YOUTH

함께가요 미래로!
Enabling People

인터페이스 vs. 클래스

인터페이스 vs. 클래스

클래스와 인터페이스 비교

	클래스	인터페이스
특징	class 키워드를 사용하여 정의 필드와 메서드, 생성자로 이루어짐	interface 키워드를 사용하여 정의 상수와 추상메소드(메소드 선언부)로 이루어짐 public static fianl 생략 public abstract 생략
관계	인터페이스를 구현함	클래스에 의해 구현됨
멤버 변수	선언 가능	상수만 가능
다중 상속	클래스는 하나의 클래스만 상속 가능	인터페이스는 여러 개의 인터페이스 상속 가능 (구현부가 없으므로 헷갈리지 않음)
다중 구현	클래스는 여러 개의 인터페이스를 다중으로 구현(implements) 가능	
인스턴스	생성 가능	생성 불가
타입	타입으로 사용 가능	타입으로 사용 가능

인터페이스 vs. 클래스

추상클래스와 인터페이스 비교

	추상 클래스	인터페이스
객체생성	불가	불가
일반 메소드	가능	불가
일반 필드	가능	불가(상수만 가능)
메서드	abstract를 붙여야만 추상 메소드	모든 메서드는 추상 메서드
사용	- 추상적인 클래스의 성격을 가질 때(일부 메서드만 미완성인 설계도) - 서로 유사한 클래스 사이에 코드를 공유하고 싶을 때	- 서로 관련없는 클래스 사이에 공통으로 적용되는 인터페이스를 구현하기를 원할 때, ex) Comparable, Serializable - 객체(클래스)의 성격이라기보다 어떤 기능을 구현하고 있다는 약속의 성격이 있을 때
공통점	특정 기능의 구현을 강제하고 싶을 때	

SAMSUNG SW ACADEMY FOR YOUTH

함께가요 미래로!
Enabling People

Generic

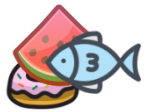
제네릭?

- 타입 파라미터를 이용하여 클래스, 인터페이스, 메서드를 다양한 타입을 처리할 수 있도록 작성하는 기법

제네릭 클래스란?

- 클래스를 정의할 때
- 클래스 안에서 사용되는 자료형(타입)을 구체적으로 명시하지 않고
- T와 같이 타입 매개변수를 이용하는 클래스

- 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



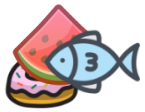
뭐든지 담을 수 있는 박스!

장난감 or 식료품?

담을 때는 편하지만 뺄 때는 번거롭다.

장난감

어떤 것을 담을 것인지 한정



식료품

담을 때는 번거롭지만 뺄 때는 편하다.

표현

- 클래스 또는 인터페이스 선언 시 <> 에 타입 파라미터 표시
- ClassName: Raw Type
- ClassName<T>: Generic Type

```
public class ClassName<T>{}
public interface InterfaceName<T>{}
```

```
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 {… }
```

- 타입 파라미터: 특별한 의미의 알파벳 보다는 단순히 임의의 참조형 타입을 말함
- T : reference Type
- E : Element
- K : Key
- V : Value

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;
  }
}
```

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

사용

- 컴파일 시 타입 파라미터들이 대입된 타입으로 대체됨

```java
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

```java
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 extends와 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 extends로 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

SAMSUNG SW ACADEMY FOR YOUTH

함께가요 미래로!
Enabling People

다음 방송에서 만나요!

삼성 청년 SW 아카데미