SAMSUNG SW ACADEMY FOR YOUTH

함께가요 미래로!
Enabling People

삼성 청년 SW 아카데미

Java

SAMSUNG SW ACADEMY FOR YOUTH

Programming Language

객체지향 프로그래밍

- 인터페이스
- 제네릭

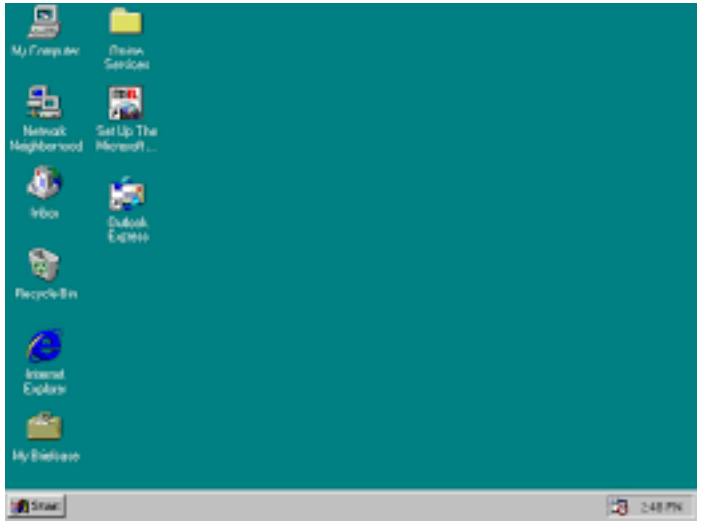
SAMSUNG SW ACADEMY FOR YOUTH

함께가요 미래로!
Enabling People

인터페이스

인터페이스

- 일반적인 의미의 인터페이스(Interface)
  - 정의: 두 시스템, 장치 또는 구성 요소가 만나는 지점으로 상호작용을 할 수 있는 경계 또는 접점
  - 기능: 서로 다른 장치나 시스템을 연결하고 소통할 수 있도록 하는 표준화된 규격이나 방법
  - 약속: 두 시스템 간의 상호작용 방법과 규칙에 대한 사전 합의



인터페이스

인터페이스(Interface)

- 완벽히 추상화된 설계도
- 클래스와 유사하게 작성되지만 class 대신 interface 키워드 사용
- 기본적으로 모든 메서드가 추상 메서드 (JDK8 부터 default 메서드와 static 메서드도 포함될 수 있음)
- 인터페이스 내에 선언된 메서드는 public abstract가 기본으로 생략되어 있음.
- 인터페이스 내에 정의된 변수는 자동으로 public static final로 간주되며, 생략할 수 있음.

```
public interface 인터페이스이름 {
  public static final 타입 상수이름1 = 10;
  타입 상수이름 상수이름2 = 10;

  public abstract 반환형 메서드이름1([매개변수들]);
  반환형 메서드이름2([매개변수들]);
}
```

인터페이스

인터페이스의 구현

- 인터페이스는 그 자체로 인스턴스를 생성할 수 없음 (구현부가 없으므로)
- 객체를 생성하기 위해서는 먼저 인터페이스를 구현하는 클래스를 만들고, 클래스를 이용해 객체 생성
- 클래스가 인터페이스를 구현할 경우 implements 키워드를 사용
- 클래스는 여러 개의 인터페이스를 다중 구현 가능
- 클래스는 인터페이스의 추상메서드를 모두 구현(재정의)해야 객체 생성 가능
- 클래스는 인터페이스의 추상메서드를 모두 재정의하지 않을 경우 추상 클래스가 됨

```
interface Shape {}
class Circle extends Shape {}
class Circle implements Shape {}
```

인터페이스

인터페이스와 다형성

- 인터페이스를 구현한 클래스로 만든 객체는 해당 인터페이스 타입으로 참조할 수 있음
- 동적 바인딩: 런타임 시점에서는 실제 객체의 메서드가 호출

```
public interface AlarmSound {
    void playAlarm();
}
```

```
public class GalaxyPhone implements AlarmSound {
    @Override
    public void playAlarm() {
        System.out.println("Beep Beep Beep!");
    }
}
```

```
public class IPhone implements AlarmSound {
    @Override
    public void playAlarm() {
        System.out.println("Ding Ding Ding!");
    }
}
```

```
AlarmSound galaxyPhone = new GalaxyPhone();
AlarmSound iPhone = new IPhone();

galaxyPhone.playAlarm();
iPhone.playAlarm();
```

인터페이스

인터페이스 상속

- extends 키워드를 이용하여 상속
- 클래스와 달리 인터페이스는 다중 상속이 가능

```
interface Movable {
    public abstract void move();
}

interface Cookable {
    public abstract void cook();
}

interface Chef extends Movable, Cookable {
}
```

```
class Robot implements Chef {

    @Override
    public void move() { // 생략 }

    @Override
    public void cook() { // 생략 }
}
```

인터페이스

default 메서드

- 인터페이스에 구현부가 있는 메서드를 작성할 수 있음
- 메서드 앞에 default 라는 키워드를 작성 해야함
- public 접근제한자를 사용해야 하며 public은 생략 가능
- 목적: 인터페이스의 하위 호환성을 유지하면서 새로운 메서드를 추가
- 클래스에서 인터페이스의 default 메서드를 재정의할 수 있음

```
public interface MyInterface {
    void abstractMethod();

    default void defaultMethod() {
        System.out.println("This is a default method.");
    }
}
```

인터페이스

정적(static) 메서드

- 인터페이스 내에 선언된 static 메서드는 클래스의 static 메서드와 사용 방법이 동일함
- 인터페이스 이름으로 메서드에 접근하여 사용
- 특정 인터페이스에 관련된 유틸리티 메서드나 헬퍼 메서드를 제공할 수 있음
- static 메서드는 인터페이스를 구현한 클래스에서 상속되거나 재정의할 수 없음

```
public interface MyInterface {

    static void staticMethod() {
        System.out.println("This is a static method in the interface.");
    }
}
```

인터페이스

인터페이스의 필요성

- 표준화 처리 가능: 여러 클래스들이 동일한 인터페이스를 구현하여 일관된 방식으로 동작 처리
- 개발 기간 단축 가능: 시스템 구조를 먼저 설계하고, 각 부분을 독립적으로 개발하여 개발 기간 단축
- 서로 관계가 없는 클래스들 간의 관계 형성: 관련이 없는 클래스들이 동일한 인터페이스를 구현하여 공통된 동작 공유
- 간접적인 클래스 사용으로 모듈 교체 용이: 구체적인 클래스에 의존하지 않고 인터페이스를 통해 클래스 사용, 모듈 교체 용이
- 독립적 프로그래밍 가능: 각 클래스가 독립적으로 개발 및 테스트 가능, 코드 재사용성과 유지보수성 향상
- 다형성 지원: 같은 인터페이스를 구현하는 객체들을 일관되게 처리하여 코드 유연성과 확장성 증가
- 설계의 유연성 제공: 클래스 간의 강한 결합을 피하여 설계의 유연성 증가, 시스템 변경 및 확장 시 영향 최소화

인터페이스

추상화(Abstraction)

- 불필요한 세부 사항을 숨기고, 중요한 특징이나 기능에 집중하는 것
- 공통의 인터페이스를 정의하고, 구체적인 구현은 하위 클래스에 맡김
- 보다 추상화된 클래스(상위 클래스, 추상 클래스, 인터페이스)에 의존, 연관된 코드 작성
  → 외부의 코드를 단순화하고 관심사의 분리를 강화하는 추상화의 한 유형
- 구체적인 클래스들의 공통점을 뽑아 그룹화하여 상위(추상) 클래스, 또는 인터페이스를 만들고, 상속 및 구현을 이용하여 코드 중복을 해결하는 계층적 코드 체계의 재조직 과정
- 현실 세계의 구체적 실체에서 SW객체를 모델링하는 과정에서 소프트웨어의 목적에 맞는 특징, 행위만 추출하는 과정 → 추상화: 행동과 상태가 있으며, 다른 객체와 상호작용하는 추상적 객체를 정의하는 기능.
- 객체지향 프로그래밍에서, abstraction은 선언되었을 경우(자바에서는 abstract 클래스 혹은 interface) 그 자체로 객체 생성 불가를 의미하며, 객체 생성을 위해서는 프로그래머에게 구현을 강제하는 책임이 있음을 의미
- 캡슐화: 상태 세부 정보를 숨기고, 데이터 타입과 행동을 강하게 연관시키며 데이터 타입 간의 상호작용을 표준화하는 것.
- 다형성: 추상화가 진행되어 서로 다른 타입의 객체에 의해 대체될 수 있게 하는 것.
- 상속: 추상화가 상위 클래스, 추상 클래스, 인터페이스를 만들고 상속, 구현하는 과정에서 이루어짐.

인터페이스 vs 클래스

인터페이스 vs 클래스

클래스와 인터페이스 비교

	클래스	인터페이스
특징	• class 키워드를 사용하여 정의 • 필드와 메서드, 생성자로 이루어짐	• interface 키워드를 사용하여 정의 • static 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(static 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 - 다형성 가능 - 보다 추상화된 설계도에 의존하는 코드를 작성하고 싶을 때 - 타입으로 사용 가능	

SAMSUNG SW ACADEMY FOR YOUTH

함께가요 미래로!
Enabling People

제네릭

제네릭

- 다양한 종류의 객체, 데이터를 처리할 수 있도록 클래스, 인터페이스, 메서드를 작성하는 기법
- 타입 매개변수를 사용하여 코드의 재사용성을 높이고 타입 안정성을 보장

```
public class GlassJar {
    private Object content;

    public void setContent(Object content) {
        this.content = content;
    }

    public Object getContent() {
        return content;
    }
}
```


```
GlassJar jar = new GlassJar();
CandyGlassJar candyJar = new CandyGlassJar();
JellyGlassJar gellyJar = new JellyGlassJar();
IntGlassJar intJar = new IntGlassJar();
StrGlassJar strJar = new StrGlassJar();
```

제네릭 클래스

- 클래스 정의 시 타입 매개변수를 사용하여 다양한 타입을 처리할 수 있는 클래스
- 타입 매개변수는 <> 안에 적는다

```
public class GlassJar<T> {
    private T content;


    public void setContent(T content) {
        this.content = content;
    }


    public T getContent() {
        return content;
    }
}
```


```
GlassJar<Object> jar = new GlassJar<>();
GlassJar<Candy> candyJar = new GlassJar<>();
GlassJar<Jelly> jellyJar = new GlassJar<>();
GlassJar<Integer> intJar = new GlassJar<>();
GlassJar<String> strJar = new GlassJar<>();
```

제네릭 클래스 선언

- 클래스 또는 인터페이스 선언 시 <> 에 타입 파라미터 표시

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

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

제네릭 클래스 객체 생성

- 변수와 생성 쪽의 타입은 반드시 일치해야 함. (상속관계에 있어도 마찬가지)

```
Box<Student> box = new Box<Student>(); (O)
Box<Person> box = new Box<Student>();  (X)
```

- 추정이 가능한 경우 타입 생략 가능 (생성자 쪽 생략 가능 JDK 1.7 부터)

```
Box<Student> box = new Box<>();
```

- 제네릭 타입을 지정하지 않고 생성이 가능하지만 권장 X (자동으로 T는 Object)
- Raw 타입: 타입 매개 변수가 없는 제네릭 클래스의 타입, 타입 인수 <>를 생략한다면 Raw 타입이 됨
- JDK 5.0 이전에는 제네릭이 없었기 때문에 호환을 위해 도입, 사용하지 않는 것을 권장
  (형변환 필요, 제네릭의 장점이 모두 사라짐)

제네릭 메서드

- 타입 파라미터를 사용하는 메서드
- 클래스의 타입 파라미터와는 별개로, 메서드 레벨에서 제네릭 타입을 정의하고 사용할 수 있음
- 제네릭 메서드는 메서드의 매개변수나 반환 타입을 타입 파라미터로 지정하여 다양한 타입을 처리
- 제네릭 메서드를 정의하려면 메서드의 반환 타입 앞에 타입 파라미터를 선언

```
public <U> void printClassName(U item) {
    System.out.println("Item type: " + item.getClass().getName());
}
```

- 메서드를 호출할 때는 타입 파라미터를 생략 가능(컴파일러가 추론)할 수도 있지만 명시할 수도 있음
- 명시할 때는 메서드명 앞, .연산자 사이에 <타입>을 표시

```
integerBox.printClassName("Test String");
integerBox.<Double>printClassName(12.5);
```

한정된 타입 매개변수

- 제네릭 클래스를 정의할 때
- 모든 종류의 타입에 대해서 작성하는 것이 아니라 특정한 종류의 타입에 대해서만 작성하고 싶은 경우
- 구체적인 타입의 제한이 필요할 때 extends 키워드를 사용할 수 있음
- 타입 파라미터를 한정할 때는 하한 경계(super)의 사용이 불가하며, 상한 경계(extends)만 사용 가능

```
class Box<T extends Person> {
    private T obj;

    public T getObj() { return obj; }

    public void setObj(T obj) { this.obj = obj; }
}
```

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

와일드 카드

- 제네릭 클래스 인스턴스 변수의 참조 타입을 작성할 때
- 와일드 카드로서 문자 ?를 사용
- 불특정 타입을 나타내기 위해 사용하는 특수한 타입 매개변수
- 타입이 구체적으로 정해진 제네릭 클래스들의 부모 타입으로서 사용 가능
- ex) 제네릭 클래스 List에 대하여 Box<A>, Box<B>, Box<C>가 있을 경우 Box<?>는 Box<A>, Box<B>, Box<C>의 부모 타입
- 제네릭 클래스에 다형성 적용 가능
- 와일드 카드의 종류

종류	표현	설명
제한없는 와일드 카드	GenericType<?>	타입에 제한이 없음
상한 경계 와일드 카드	GenericType<? extends T>	T 와 T를 상속받은 타입들만 사용 가능
하한 경계 와일드 카드	GenericType<? super T>	T 와 T의 조상 타입만 사용 가능

SAMSUNG SW ACADEMY FOR YOUTH

함께가요 미래로!
Enabling People

다음 방송에서 만나요!

삼성 청년 SW 아카데미