Generics

☞ AppleBox와 OrangeBox 클래스의 설계

```
class AppleBox
{
    Apple item;
    public void store(Apple item) { this.item=item; }
    public Apple pullOut() { return item; }
}

class OrangeBox
{
    Orange item;
    public void store(Orange item) { this.item=item; }
    public Orange pullOut() { return item; }
}
```

Apple 상자와 Orange 상자를 구분한 모델

구현의 편의만 놓고보면, FruitBox 클래스가 더 좋아 보인다. 하지만 FruitBox 클래스는 자료형에 안전하지 못하다는 단점이 있다.

물론AppleBox와 OrangeBox는 구현의 불편함이 따르는 단점이 있다. 그러나 자료형에 안전하다는 장점이 있다 .

```
Class FruitBox
{
    Object item;
    public void store(Object item) { this.item=item; }
    public Object pullOut() { return item; }
}
```

무엇이든 담을 수 있는 상자 클래스

AppleBox, OrangeBox의 장점인 자료형의 안전성과 FruitBox의 장점인 구현의 편의성을 한데 모은 것이 제네릭이다!

☞ 자료형의 안전성에 대한 논의

```
public static void main(String[] args)
{
    FruitBox fBox1=new FruitBox();
    fBox1.store(new Orange(10));
    Orange org1=(Orange)fBox1.pullOut();
    org1.showSugarContent();

    FruitBox fBox2=new FruitBox();
    fBox2.store("오렌지");
    Orange org2=(Orange)fBox2.pullOut(); // 예외 발생지점
    org2.showSugarContent();
}
```

실행중간에 Class Casting Exception이 발생한다!
그런데 실행중간에 발생하는 예외는 컴파일과정에서 발견되는 오류상황보다 발견 및 정정이 어렵다!
즉, 이는 자료형에 안전한 형태가 아니다.

```
public static void main(String[] args)
{
    OrangeBox fBox1=new OrangeBox();
    fBox1.store(new Orange(10));
    Orange org1=fBox1.pullOut();
    org1.showSugarContent();

    OrangeBox fBox2=new OrangeBox();
    fBox2.store("오렌지"); // 에러 발생지점
    Orange org2=fBox2.pullOut();
    org2.showSugarContent();
}
```

컴파일 과정에서, 메소드 store에 문자열 인스턴스가 전달될 수 없음이 발견된다. 이렇듯 컴파일 과정에서 발견된 자료형 관련 문제는 발견 및 정정이 간단하다.
즉, 이는 자료형에 안전한 형태이다.

☞ 제네릭(Generics) 클래스 설계

```
class FruitBox
{
   Object item;
   public void store(Object item){this.item=item;}
   public Object pullOut(){return item;}
}
```

T라는 이름이 매개변수화 된 자료형임을 나타냄

```
class FruitBox <T>
{
   T item;
   public void store( T item ) {this.item=item;}
   public T pullOut() {return item;}
}
```

T에 해당하는 자료형의 이름은 인스턴스를 생성하는 순간에 결정이 된다!

제네릭 클래스가 자료형의 안전과 정의의 편의라는 두 마리 토끼를 실제로 잡았는지 생각해 보자!

☞ 제네릭 클래스 기반 인스턴스 생성

```
FruitBox<Orange> orBox=new FruitBox<Orange>();
```

T를 Orange로 결정해서 FruitBox의 인스턴스를 생성하고 이를 참조할 수 있는 참조 변수를 선언해서 참조 값을 저장한다!

```
FruitBox<Apple> apBox=new FruitBox<Apple>();
```

T를 Apple로 결정해서 FruitBox의 인스턴스를 생성하고 이를 참조할 수 있는 참조 변수를 선언해서 참조 값을 저장한다!

```
public static void main(String[] args)
{
    FruitBox<Orange> orBox=new FruitBox<Orange>();
    orBox.store(new Orange(10));
    Orange org=orBox.pullOut();
    org.showSugarContent();

    FruitBox<Apple> apBox=new FruitBox<Apple>();
    apBox.store(new Apple(20));
    Apple app=apBox.pullOut();
    app.showAppleWeight();
}
```

인스턴스 생성시 결정된 T의 자료형에 일치하지 않으면 컴파일 에러가 발생하므로 자료형에 안전한 구조임!

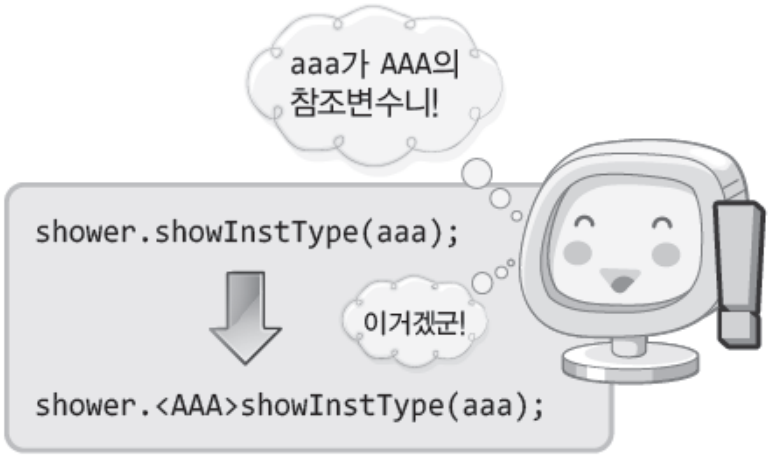
☞ 제네릭 메소드의 정의와 호출

```
class InstanceTypeShower
{
    int showCnt=0;

    public <T> void showInstType(T inst)
    {
        System.out.println(inst);
        showCnt++;
    }

    void showPrintCnt() { · · · · · }
}
```

클래스의 메소드만 부분적으로 제네릭화 할 수 있다!



제네릭 메소드의 호출과정에서 전달되는 인자를 통해서 제네릭 자료형을 결정할 수 있으므로 자료형의 표현은 생략 가능하다!

```
public static void main(String[] args)
{
    AAA aaa=new AAA();
    BBB bbb=new BBB();

    InstanceTypeShower shower=new InstanceTypeShower();
    shower.<AAA>showInstType(aaa);
    shower.<BBB>showInstType(bbb);
    shower.showPrintCnt();
}
```

```
shower.showInstType(aaa);
shower.showInstType(bbb);
```

제네릭 메소드의 호출방법! T를 AAA로, BBB로 결정하여 호출하고 있다!

☞ 제네릭 메소드와 둘 이상의 자료형

```
class InstanceTypeShower2
{
    public <T, U> void showInstType(T inst1, U inst2)
    {
        System.out.println(inst1);
        System.out.println(inst2);
    }
}
```

T, U와 같은 문자는 상징적이다.
따라서 타 문자로도 대체가능!

```
public static void main(String[] args)
{
    AAA aaa=new AAA();
    BBB bbb=new BBB();

    InstanceTypeShower2 shower=new InstanceTypeShower2();
    shower.<AAA, BBB>showInstType(aaa, bbb);
    shower.showInstType(aaa, bbb);
}
```

마찬가지로 메소드 호출 시,
자료형의 정보는 생략이 가능하다!

☞ 매개변수의 자료형 제한

public static <T extends AAA> void myMethod(T param) { . . . . . }

T가 AAA를 상속(AAA가 클래스인 경우) 또는 구현(AAA가 인터페이스인 경우)하는 클래스의 자료형이 되어야 함을 명시함.

인터페이스 이름

```
public static <T extends SimpleInterface> void showInstanceAncestor(T param)
{
    param.showYourName();
}
```

인자는 SimpleInterface를 구현하는 클래스의 인스턴스이어야 함!

클래스이름

```
public static <T extends UpperClass> void showInstanceName(T param)
{
    param.showYourAncestor();
}
```

UpperClass를 상속하는 클래스의 인스턴스이어야 함!

키워드 extends는 매개변수의 자료형을 제한하는 용도로도 사용된다!

☞ 제네릭 메소드와 배열

제네릭 메소드로의 배열 전달

- 배열도 인스턴스이므로 제네릭 매개변수에 전달이 가능하다. 하지만 이렇게 전달을 하면 다음과 같은 문장을 쓸 수 없다!

  System.out.println(arr[i]);

- 다음과 같이 매개변수를 선언하면, 매개변수에 전달되는 참조 값을 배열 인스턴스의 참조 값으로 제한할 수 있다.

  T[ ] arr

- 그리고 이렇게 되면 참조 값은 배열 인스턴스의 참조 값임이 100% 보장되므로 [ ] 연산을 허용한다.

```
public static <T> void showArrayData(T[] arr)
{
    for(int i=0; i<arr.length; i++)
        System.out.println(arr[i]);
}
```

이렇듯 [ ] 연산이 필요하다면 매개변수의 선언을 통해서 전달되는 참조 값을 배열의 참조 값으로 제한해야 한다.

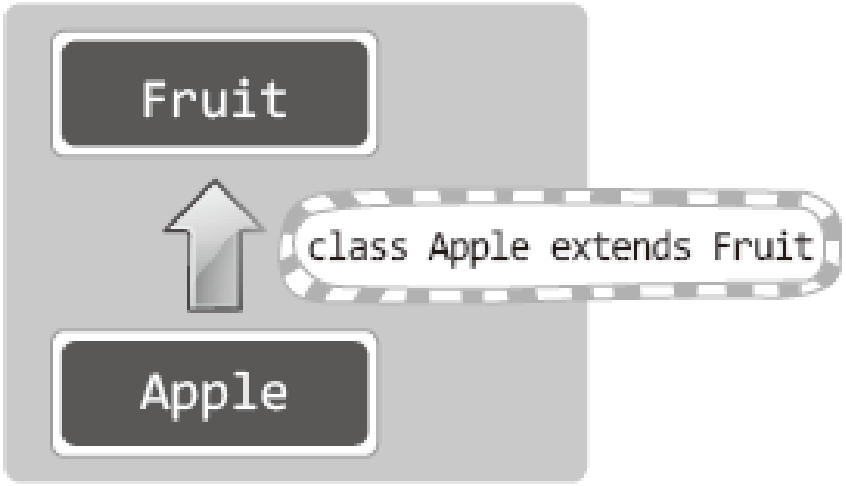
☞ 제네릭 변수의 참조와 상속의 관계

public void ohMethod (FruitBox<Fruit> param) { . . . . }

ohMehod의 인자로 전달될 수 있는 참조 값의 자료형은

-> FruitBox<Fruit>의 인스턴스 참조 값

-> FruitBox<Fruit>을 상속하는 인스턴스의 참조 값



왼쪽과 같은 상속 구조에서 FruitBox<Apple> 클래스는 ohMethod의 인자가 될 수 있을까?

NO!

반드시 키워드 extends를 이용해서 상속이 명시된 대상만 인자로 전달될 수 있다.

... extends FruitBox<Fruit>

☞ 와일드카드와 제네릭 변수의 선언

< ? extends Fruit >

<? extends Fruit>가 의미하는 바는 “Fruit을 상속하는 모든 클래스”이다.

FruitBox<?> box;

- 자료형에 상관없이 FruitBox<T>의 인스턴스를 참조에 사용하는 참조변수.
- 다음 선언과 동일.

=> FruitBox<? extends Object> box;

☞ 하위 클래스를 제한하는 용도의 와일드 카드

FruitBox<? extends Apple> boundedBox;

- ~을 상속하는 클래스라면 무엇이든지
- Apple을 상속하는 클래스의 인스턴스라면 무엇이든지 참조 가능한 참조변수 선언

FruitBox<? super Apple> boundedBox;

- ~이 상속하는 클래스라면 무엇이든지
- Apple이 상속하는 클래스의 인스턴스라면 무엇이든지 참조 가능한 참조변수 선언

☞ 제네릭 클래스의 다양한 상속 방법

```
class AAA<T>
{
    T itemAAA;
}
class BBB<T>extends AAA<T>
{
    T itemBBB;
}
```

왼쪽과 같이 제네릭 클래스도 상속이 가능하다. 그리고 이 경우 다음과 같이 인스턴스를 생성하게 된다.

```
BBB<String> myString=new BBB<String>( );
BBB<Integer> myInteger=new BBB<Integer>( );
```

이 경우 T는 각각 String과 Integer로 대체되어 인스턴스가 생성된다 .

```
class AAA<T>
{
    T itemAAA;
}

class BBB extends AAA<String>
{
    int itemBBB;
}
```

왼쪽과 같이 제네릭 클래스의 자료형을 결정해서 상속하는 것도 가능하다.

이 경우, BBB 클래스는 제네릭과 관련 없는 클래스가 되어, 일반적인 방법으로 인스턴스를 생성하면 된다.

☞ 제네릭 인터페이스의 구현 방법

```
interface MyInterface<T>
{
    public T myFunc(T item);
}
```

제네릭 인터페이스

```
class MyImplement<T> implements MyInterface<T>
{
    public T myFunc(T item)
    {
        return item;
    }
}
```

제네릭 인터페이스 구현모델1

일반적인 인터페이스의 구현과 차이가 없다!

```
class MyImplement implements MyInterface<String>
{
    public String myFunc(String item)
    {
        return item;
    }
}
```

제네릭 인터페이스 구현모델2

왼쪽과 같이 제네릭 인터페이스의 자료형을 지정해서 구현하는 것도 가능하다.

☞ 기본 자료형의 이름은 제네릭에 사용 불가!

컴파일 불가능한 문장들!

```
FruitBox<int> fb1= new FruitBox<int>();
FruitBox<double> fb1= new FruitBox<double>();
```

기본 자료형 정보를 이용해서는 제네릭 클래스의 인스턴스 생성이 불가능하다!

제네릭은 클래스와 인스턴스에 관한 이야기이다!

- 그럼 기본 자료형 정보를 대상으로는 제네릭 인스턴스의 생성이 필요한 상황에서는 어떻게 하죠?

  => 위 질문에 Wrapper 클래스를 떠올리기 바란다! 이와 관련된 이야기는 이후에 계속된다!