# 제네릭(Generic)

# 제네릭(Generic) 클래스/ 인터페이스(class/interface)의 필요성

# 제네릭(Generic) 클래스/인터페이스(class/interface)의 필요성

☞ Generic의 필요성

1 - **상품**을 저장할 수 있는 클래스 생성

사과(Apple)만 저장가능한 클래스

```
class Apple{ }

class Goods1 {
  private Apple apple = new Apple();
  public Apple get() {
    return apple;
  }
  public void set(Apple apple) {
    this.apple = apple;
  }
}
```

2

사과를 저장(set)하고 가져오기(get)

```
Goods1 goods1 = new Goods1();
goods1.set(new Apple());
Apple apple = goods1.get();
```

3

연필(Pencil)만 저장가능한 클래스

```
class Pencil{ }

class Goods2 {
  private Pencil pencil = new Pencil();
  public Pencil get() {
    return pencil;
  }
  public void set(Pencil pencil) {
    this.pencil = pencil;
  }
}
```

4

연필을 저장(set)하고 가져오기(get)

```
Goods2 goods2 = new Goods2();
goods2.set(new Pencil());
Pencil pencil = goods2.get();
```

5

6 상품이 추가될 때 마다 클래스를 생성해야 할까?

클래스 하나로 모든 상품을 저장할 수 없을까?

# 제네릭(Generic) 클래스/인터페이스(class/interface)의 필요성

☞ Generic의 필요성

1

상품마다 클래스를 생성해야 하는 문제점에 대한 해결책

**해결책 #1** **필드(field)의 타입**을 모든 객체를 저장할 수 있는 **Object로 정의**

모든 타입 객체 저장 가능

2

```
class Apple{ }
class Pencil{ }

class Goods {
  private Object object = new Object();
  public Object get() {
    return object;
  }
  public void set(Object object) {
    this.object = object;
  }
}
```

3

```
Goods goods1 = new Goods();
goods1.set(new Apple()); //Apple 저장
Apple apple = (Apple)goods1.get(); //Object→Apple

Goods goods2 = new Goods();
goods2.set(new Pencil()); //Pencil 저장
Pencil pencil = (Pencil)goods2.get(); //Object→Pencil

//wrong casting
//Goods goods3 = new Goods();
//goods3.set(new Apple());
//Pencil pen = (Pencil)goods3.get(); //ClassCastException
```

4

문제점: 약한 타입체크

출력시 입력된 객체 타입으로 캐스팅 필요 → 잘못된 객체 타입 캐스팅: RuntimeException 발생 → 컴파일 오류발생 안함

5

이 모든 것의 해결책 : 제네릭 타입 (Generic Type)

# 제네릭(Generic) 클래스/ 인터페이스(class/interface)의 문법

# 제네릭(Generic) 클래스/인터페이스(class/interface)의 문법

**제네릭 타입변수**

제네릭 타입 변수명
다수개 타입변수 사용가능
일반적으로 영대문자 하나를 사용

| 제네릭타입변수 | 의미 |
|---|---|
| T | 타입(Type) |
| K | 키(Key) |
| V | 값(Value) |
| N | 숫자(Number) |
| E | 원소(Element) |

3

1 ☞ Generic 클래스/인터페이스 **정의** 문법 구조

**제네릭 타입 변수명 한 개인 경우**

```
접근지정자 class 클래스명< T > {
  //타입 T를 사용한 코드
}
```

2

```
접근지정자 interface 클래스명< T > {
 //타입 T를 사용한 코드
}
```

**제네릭 타입 변수명 두 개인 경우**

```
접근지정자 class 클래스명< K, V > {
  //타입 K, V를 사용한 코드
}
```

```
접근지정자 interface 클래스명< K, V > {
  //타입 K, V를 사용한 코드
}
```

**예시**

4

```
public class MyClass<T>{
  private T t;
  public T get() {
    return t;
  }
  public void set(T t) {
    this.t = t;
  }
}
```

```
public interface MyInterface<K,V>{

  public abstract void setKey(K k);
  public abstract void setValue(V v);
  public abstract K getKey();
  public abstract V getValue();

}
```

# 제네릭(Generic) 클래스/인터페이스(class/interface)의 문법

☞ Generic 클래스/인터페이스 **객체생성** 문법 구조

**TIP** 5

객체생성시 제네릭타입을 지정하지 않으면
올 수 있는 Type 중 최상위 클래스(Object)로 인식
(즉, 아래 두 코드는 동일한 의미)

```
MyClass mc = new MyClass();
MyClass<Object> mc = new Myclass<>();
```

**- 객체 생성**

1

```
클래스명<실제제네릭타입> 참조변수명 = new 클래스명<실제제네릭타입>();
또는
클래스명<실제제네릭타입> 참조변수명 = new 클래스명<>();
```

생성자의 경우
내부의 타입 생략 가능

**예시 1. 제네릭 타입 변수 1개**

2

```
public class MyClass<T>{
  private T t;
  public T get() {
    return t;
  }
  public void set(T t) {
    this.t = t;
  }
}
```

**Generic 정의**

3

```
MyClass<String> mc1 = new MyClass<String>();
mc1.set("안녕");
System.out.println(mc1.get()); // "안녕"
MyClass<Integer> mc2 = new MyClass<>();
mc2.set(100);
System.out.println(mc2.get()); // 100
MyClass<Integer> mc3 = new MyClass<>();
mc3.set("안녕");   //문법오류(syntax error)
                   //강한 타입체크
```

**Generic 객체 생성 및 사용**

4

**객체생성시 타입 결정**

# 제네릭(Generic) 클래스/인터페이스(class/interface)의 문법

☞ Generic 클래스/인터페이스 **객체생성** 문법 구조

**예시 2. 제네릭 타입 변수 2개**

1

```
class KeyValue<K, V>{
  private K key;
  private V value;
  public K getKey() {
    return key;
  }
  public void setKey(K key) {
    this.key = key;
  }
  public V getValue() {
    return value;
  }
  public void setValue(V value) {
    this.value = value;
  }
}
```

**Generic 정의**

2

```
KeyValue<String, Integer> kv1 = new KeyValue<>();
kv1.setKey("사과");
kv1.setValue(1000);
String key1 = kv1.getKey();
int value1 = kv1.getValue();
System.out.println("key: "+key1+"  value: "+value1);

KeyValue<Integer, String> kv2 = new KeyValue<>();
kv2.setKey(404);
kv2.setValue("Not Found(요청한페이지를 찾을 수 없음)");
int key2 = kv2.getKey();
String value2 = kv2.getValue();
System.out.println("key: "+key2+" value: "+value2);

KeyValue<String, Void> kv3 = new KeyValue<>();
kv3.setKey("키값만 사용");
String key3 = kv3.getKey();
System.out.println("key: "+key3);
```

**Generic 객체 생성 및 사용**

# 제네릭(Generic) 클래스/인터페이스(class/interface)의 문법

☞ 앞의 **Goods** 클래스의 제네릭 정의 및 객체 생성

1 **해결책 #2** 하나의 클래스로 모든 타입을 담을 수 있어야 함 + 강한 타입 체크(문법오류) 필요

2

```
class Goods<T>{
  private T t;
  public T get() {
    return t;
  }
  public void set(T t) {
    this.t = t;
  }
}
```

**Generic 정의**

3

```
Goods<Apple> goods1 = new Goods<Apple>();
goods1.set(new Apple());
Apple apple = goods1.get();

Goods<Pencil> goods2 = new Goods<Pencil>();
goods2.set(new Pencil());
Pencil pencil = goods2.get();

Goods<Apple> goods3 = new Goods<Apple>();
goods3.set(new Apple());
//Pencil pen = goods3.get(); //syntax 에러
```

**Generic 객체 생성 및 사용**

4

**즉, 제네릭의 기본 개념은 클래스 내에 사용되는 타입을 클래스의 정의 때가 아닌 객체 생성 때 정의하겠다는 의미**

# 제네릭(Generic) 메서드(Method)

# 제네릭(Generic) 메서드(Method)

☞ 제네릭 메서드

1 - **리턴타입** 또는 **매개 변수의 타입**을 제네릭 타입으로 선언

☞ 제네릭 메서드의 **정의** 문법 구조

2

**제네릭 타입 변수명 한 개인 경우**

```
접근지정자 < T > T 메서드이름 (T t) {
  //타입 T를 사용한 코드
}
```

**제네릭 타입 변수명 두 개인 경우**

```
접근지정자 <T, V> T 메서드이름 (T t, V v) {
  //타입 T를 사용한 코드
}
```

**매개변수에만 제네릭이 사용된 경우**

```
접근지정자 < T > void 메서드이름 (T t) {
  //타입 T를 사용한 코드
}
```

**리턴타입에만 제네릭이 사용된 경우**

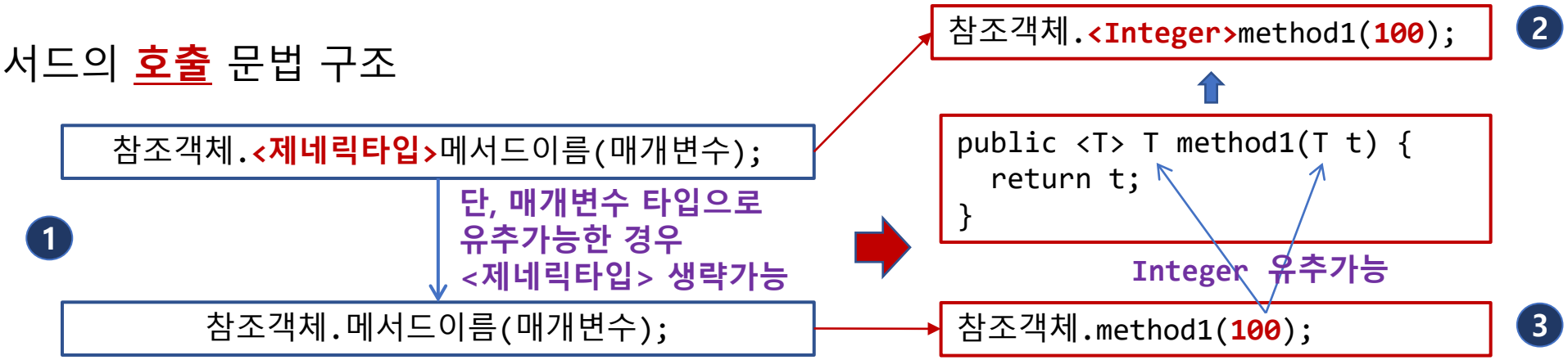
```
접근지정자 < T > T 메서드이름 (int a) {
  //타입 T를 사용한 코드
}
```

3

```
class GenericMethods {

  public <T> T method1(T t) {
    return t;
  }

  public <T> boolean method2(T t1, T t2) {
    return t1.equals(t2);
  }

  public <K, V> void method3(K k , V v){
    System.out.print(k +" : ");
    System.out.println(v);
  }

}
```

# 제네릭(Generic) 메서드(Method)

☞ 제네릭 메서드의 **호출** 문법 구조




```
class GenericMethods {
  public <T> T method1(T t) {
    return t;
  }
  public <T> boolean method2(T t1, T t2) {
    return t1.equals(t2);
  }
  public <K, V> void method3(K k , V v){
    System.out.print(k +" : ");
    System.out.println(v);
  }
}
```

4

**Generic 메서드 정의**

```
GenericMethods gm = new GenericMethods();

String str1 = gm.<String>method1("안녕");
String str2 = gm.method1("안녕");
System.out.println(str1 + ", "+ str2); //안녕, 안녕

boolean bool1 = gm.<Double>method2(2.5,2.5);
boolean bool2 = gm.method2(2.5,2.5);
System.out.println(bool1 + ", "+ bool2); //true, true

gm.<String, Integer>method3("국어", 80); //국어 : 80
gm.method3("국어", 80); //국어 : 80
```

5

**Generic 메서드 호출**

# 제네릭(Generic) 메서드(Method)

☞ 제네릭 메서드 내에서 **사용가능한 메서드**

1

```
class A {
  public <T> void method1(T t) {
    ...
  }
}
```

정의 시점에는 어떤 타입이 들어올지 모름

제네릭 메서드 내부에서는 참조변수 t의 메서드로 Object 클래스의 메서드만 사용 가능

2

나중에 메서드 매개변수로 String을 넣을 예정이라고 하여도 메서드 내부에서 String 메서드 사용 불가

```
A a = new A();

int length = a.<String>method1("안녕");
```

3

```
class A {
  public <T> void method1(T t) {
    System.out.println(t.length()); //(불가능)
  }
}
```

String 클래스 메서드

4

```
class A {
  public <T> void method1(T t) {
    System.out.println(t.equals("안녕")); //(가능)
  }
}
```

Object 클래스 메서드

5

Q. 만일 메서드 내에 특정 클래스의 메서드를 사용하고 싶다면?
→ 제네릭 타입의 범위 제한

# The End

# 제네릭(Generic) 타입 범위 제한(Bound)

# 제네릭(Generic) 타입 범위 제한(Bound)

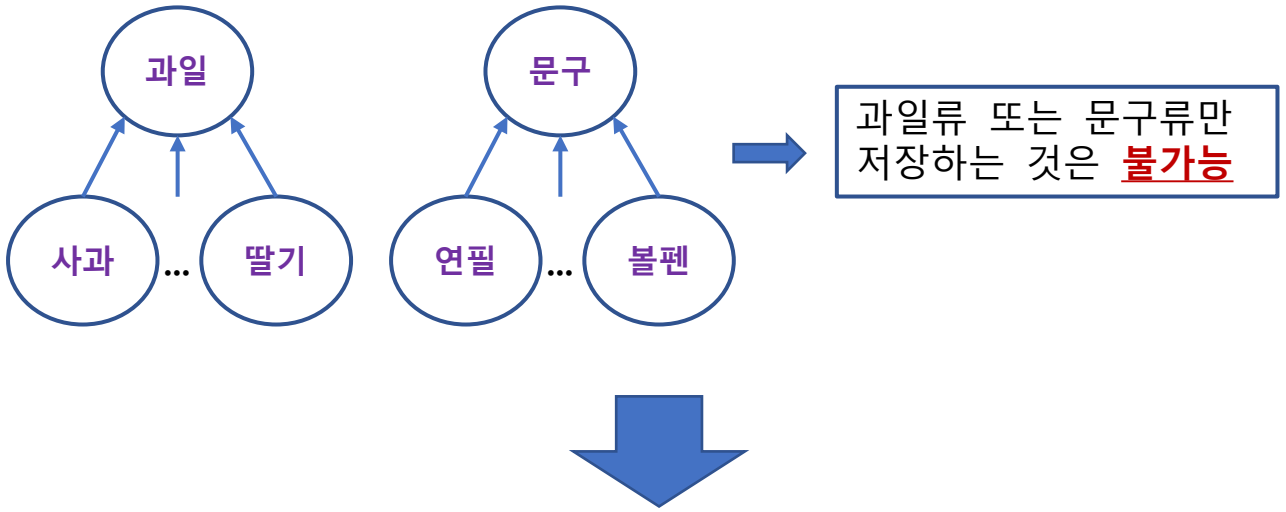
1 ☞ 제네릭(Generic) 타입 범위 **제한(Bound)의 필요성**

**제네릭 클래스**

2

```
class Goods<T>{
  private T t;
  public T get() {
    return t;
  }
  public void set(T t) {
    this.t = t;
  }
}
```

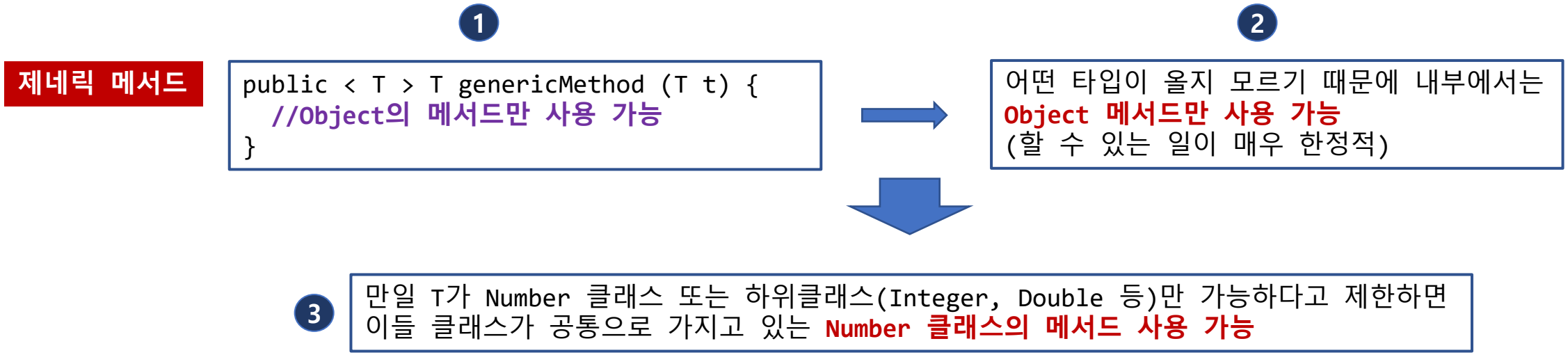
3




4 만일 T가 **과일 클래스 또는 그 하위 클래스만 올 수 있도록 한정**하면 Goods는 **과일류만을 저장**하는 제네릭 클래스로 정의 가능

# 제네릭(Generic) 타입 범위 제한(Bound)

☞ 제네릭(Generic) 타입 범위 **제한(Bound)의 필요성**



# 제네릭(Generic) 타입 범위 제한(Bound)

☞ 제네릭(Generic) 타입 범위 **제한(Bound)의 종류**

1

- **#1. 제네릭 클래스**의 타입 제한
- **#2. 제네릭 메서드**의 타입 제한
- **#3.** 일반메서드 **매개변수**로서의 **제네릭 클래스** 타입제한

☞ **#1. 제네릭 클래스**의 타입 제한

- 타입제한 기본 문법 구조

4

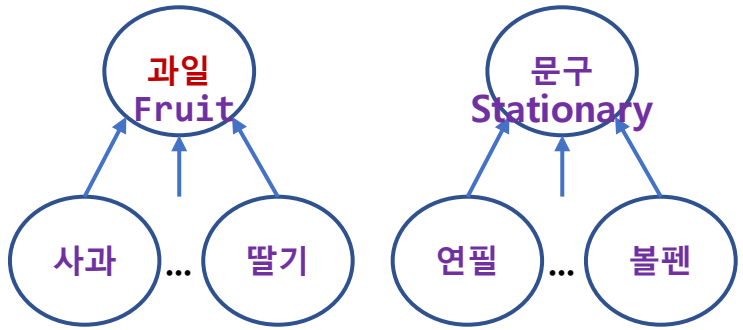
클래스/인터페이스 상관없이 항상 **extends** 사용

2

```
접근지정자 class 클래스명<T extends Fruit> {
  //타입 T를 사용한 코드
}
```

→ **Fruit 또는 그 하위클래스만 가능**

3




```
class Goods<T extends Fruit>{
  private T t;
  public T get() {
    return t;
  }
  public void set(T t) {
    this.t = t;
  }
}
```

5

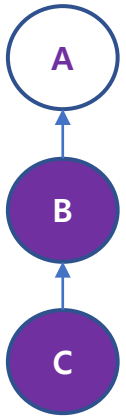
```
Goods<Apple> goods = new Goods<>();
```

6

```
Goods<Pencil> goods = new Goods<>();
```

# 제네릭(Generic) 타입 범위 제한(Bound)

☞ **#1. 제네릭 클래스**의 타입 제한




1

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

class D <T extends B>{
  private T t;
  public T get() {
    return t;
  }
  public void set(T t) {
    this.t = t;
  }
}
```

**TIP**

3

객체생성시 제네릭 타입을 지정하지 않으면
올 수 있는 Type 중 최상위 클래스(B)로 인식
(즉, 아래 두 코드는 동일한 의미)

```
D d = new D();
D<B> d = new D<B>();
```

2

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

  //D<A> d1 = new D<>(); //(불가능)
  D<B> d2 = new D<>();
  D<C> d3 = new D<>();
  D d4 = new D(); //D<B> d4 = D<>();와 동일

  d2.set(new B());
  d2.set(new C());
  //d3.set(new B()); //(불가능)
  d3.set(new C());
  d4.set(new B());
  d4.set(new C());

}
```

# 제네릭(Generic) 타입 범위 제한(Bound)

☞ **#2. 제네릭 메서드**의 타입 제한

- 타입제한 기본 문법 구조

4 클래스/인터페이스 상관없이 항상 **extends** 사용

```
접근지정자 <T extends 최상위클래스명> T 메서드이름(T t) {
  //부모클래스의 메서드 사용 가능
}
```

```
class GenericMethods {

  public <T> void method1(T t) {
    char c = t.charAt(0); //Object 메서드만 사용 가능
    System.out.println(c);
  }

  public <T extends String> void method1(T t) {
    char c = t.charAt(0); //String의 메서드 사용 가능
    System.out.println(c);
  }

}
```

1 ✗

2 ○

3

**char charAt(int index)**
-String 클래스의 메서드
-String 문자열의 index 위치의 문자를 리턴

# 제네릭(Generic) 타입 범위 제한(Bound)

☞ **#2. 제네릭 메서드**의 타입 제한

```
class A {
  public <T extends String> void method1(T t) {
    System.out.println(t.charAt(0));
  }
}
```

1

```
interface MyInterface{
  public abstract void print();
}

class B {
  public <T extends MyInterface> void method1(T t) {
    t.print();
  }
}
```

2

인터페이스이어도 extends 사용

```
public static void main(String[] args) {
  A a = new A();
  a.method1("안녕"); //안

  B b = new B();
  b.method1(new MyInterface() {
    @Override
    public void print() {
      System.out.println("print() 구현");
    }
  });
}
```

3

```
안
print() 구현
```

# 제네릭(Generic) 타입 범위 제한(Bound)

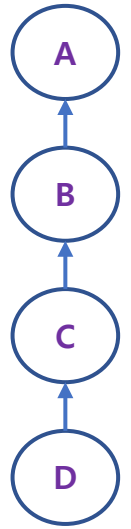
- ☞ **#3.** 메서드 **매개변수**로서의 **제네릭 클래스** 타입제한

1

- 메서드의 **매개변수**로 제네릭 클래스 객체가 오는 경우의 타입제한
- 기본 문법 구조

```
class Goods<T>{
  private T t;
  public T get() {
    return t;
  }
  public void set(T t) {
    this.t = t;
  }
}
```

2




3

**Case #1**

```
접근지정자 메서드이름( 제네릭클래스명<제네릭타입명> 참조변수명){
  //…
}
```

4 → `method(Goods<A> v)` → **A 만 가능**

**Case #2**

```
접근지정자 메서드이름( 제네릭클래스명<?>  참조변수명) {
  //…
}
```

→ `method(Goods<?> v)` → **A, B, C, D 가능**

**Case #3**

```
접근지정자 메서드이름( 제네릭클래스명<? extends 상위클래스> 참조변수명) {
  //…
}
```

→ `method(Goods<? extends B> v)` → **B, C, D만 가능**

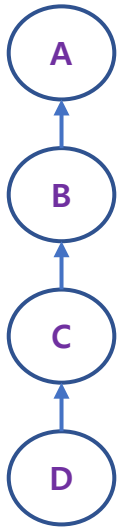
**Case #4**

```
접근지정자 메서드이름( 제네릭클래스명<? super 하위클래스> 참조변수명) {
  //…
}
```

→ `method(Goods<? super B> v)` → **A, B만 가능**

# 제네릭(Generic) 타입 범위 제한(Bound)

- ☞ **#3.** 메서드 **매개변수**로서의 **제네릭 클래스** 타입제한




1

```
class A{}
class B extends A{}
class C extends B{}
class D extends C{}

class Goods<T>{
  private T t;
  public T get() {
    return t;
  }
  public void set(T t) {
    this.t = t;
  }
}
```

2

```
class Test {
  void method1(Goods<A> g) {} //#case1
  void method2(Goods<?> g) {} //#case2
  void method3(Goods<? extends B> g) {} //#case3
  void method4(Goods<? super B> g) {} //#case4
}
```

3

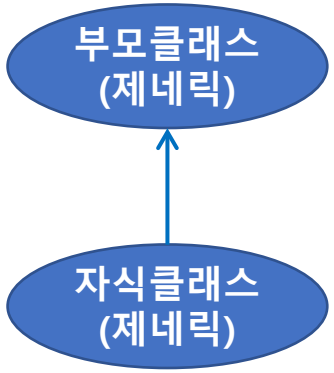
```
public static void main(String[] ar) {
  Test t = new Test();

  //#1. Case1. method1(Goods<A> g)
  t.method1(new Goods<A>()); //O
  t.method1(new Goods<B>()); //X
  t.method1(new Goods<C>()); //X
  t.method1(new Goods<D>()); //X

  //#2. Case2. method2(Goods<?> g)
  t.method2(new Goods<A>()); //O
  t.method2(new Goods<B>()); //O
  t.method2(new Goods<C>()); //O
  t.method2(new Goods<D>()); //O

  //#3. Case3. method3(Goods<? extends B> g)
  t.method3(new Goods<A>()); //X
  t.method3(new Goods<B>()); //O
  t.method3(new Goods<C>()); //O
  t.method3(new Goods<D>()); //O

  //#4. Case4. method4(Goods<? super B> g)
  t.method4(new Goods<A>()); //O
  t.method4(new Goods<B>()); //O
  t.method4(new Goods<C>()); //X
  t.method4(new Goods<D>()); //X
}
```

# 제네릭(Generic)의 상속

# 제네릭(Generic)의 상속

☞ 제네릭 클래스의 상속

1




부모클래스가 제네릭인 경우

자식클래스 또한 제네릭임
(부모의 **제네릭변수를 그대로 물려받음**)

2

```
class Parent<K,V>{
   …
}
```

```
class Child<K,V> extends Parent<K,V>{
   …
}
```

3

```
class Parent<K>{
   …
}
```

```
class Child<K,V> extends Parent<K>{
   …
}
```

4

**자식클래스의 제네릭 변수의 개수는
부모보다 같거나 클 수 있음**

# 제네릭(Generic)의 상속

1
```
class Parent<T> {
  T t;
  public T getT() {
    return t;
  }
  public void setT(T t) {
    this.t = t;
  }
}
```

2
```
class Child1<T> extends Parent<T> {

}
```

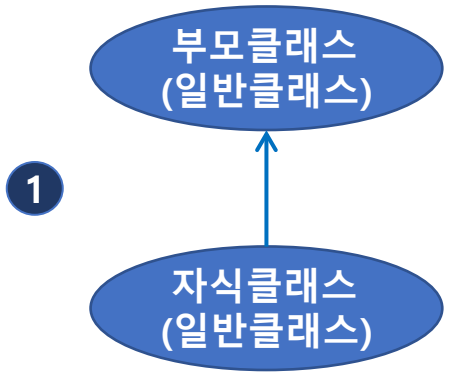
3
```
class Child2<T, V> extends Parent<T> {
  V v;
  public V getV() {
    return v;
  }
  public void setV(V v) {
    this.v = v;
  }
}
```

4
```
public static void main(String[] ar) {

  //#1. 부모제네릭 클래스
  Parent<String> p = new Parent<>();
  p.setT("부모제네릭클래스");
  System.out.println(p.getT());

  //#2. 자식1 제네릭 클래스
  Child1<String> c1 = new Child1<>();
  c1.setT("자식1 제네릭클래스");
  System.out.println(c1.getT());

  //#3. 자식2 제네릭 클래스
  Child2<String, Integer> c2 = new Child2<>();
  c2.setT("자식2 제네릭클래스");
  c2.setV(100);
  System.out.println(c2.getT());
  System.out.println(c2.getV());
}
```

```
부모제네릭클래스
자식1 제네릭클래스
자식2 제네릭클래스
100
```

# 제네릭(Generic)의 상속

☞ 제네릭 메서드를 가진 일반클래스의 상속




1

**부모클래스가**
**제네릭메서드를 가진 경우**

**자식클래스 또한**
**제네릭메서드를 그래도 물려받음**

2

```
class Parent {
  public <T> void print(T t){
     System.out.println(t);
  }
}
```

```
class Child extends Parent {

}
```

3

```
Parent p = new Parent();
p.<String>print("안녕");

Child c = new Child();
c.<String>print("안녕");
```

# 제네릭(Generic)의 상속

☞ 제네릭 메서드를 가진 일반클래스의 상속

1

```
class Parent {
  <T extends Number> void print(T t) {
     System.out.println(t);
   }
}
```

2

```
class Child extends Parent {

}
```

3

```
public static void main(String[] ar) {

  //#1. 부모 클래스의 제네릭 메서드 사용
  Parent p = new Parent();
  p.<Integer>print(10);
  p.print(10);

  //#2. 자식 클래스의 제네릭 메서드 사용
  Child c = new Child();
  c.<Double>print(5.8);
  p.print(5.8);

}
```

```
10
10
5.8
5.8
```

# The End