KB금융그룹 | 국민의 평생 금융파트너

2025년 상반기 K-디지털 트레이닝

제네릭

[KB] IT's Your Life

KB 국민은행

제너릭

내용물로 content 필드를 선언할 때 타입을 무엇으로 해야 하나?

```
public class Box {
   public ? content;
}
```

- 어떠한 타입도 가능해야 한다면 → Object 타입

```
public class Box {
   public Object content;
}
```

```
Box box = new Box();
box.content = 모든 객체;
```

```
String content = (String) box.content;
```

제네릭

- 결정되지 않은 타입을 파라미터로 처리
- 실제 사용할 때 파라미터를 구체적인 타입으로 대체시키는 기능

- <T>는 T가 타입 파라미터임을 뜻하는 기호.
  - 타입이 필요한 자리에 T를 사용할 수 있음을 알려줌

```
public class Box<T> {
    public T content;
}
```

```
Box<Integer> box = new<Integer>();
box.content = 100;
int content = box.content;  // 강제 타입 변환이 필요없이 100을 바로 얻을 수 있음
```

제네릭

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

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

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

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

Box.java

```
package ch13.sec01;

public class Box<T> {
  public T content;
}
```

GenericExample.java

```
package ch13.sec01;

public class GenericExample {
  public static void main(String[] args) {
    //Box<String> box1 = new Box<String>();
    Box<String> box1 = new Box<>();  // Box 생성시 타입파라미터 대신 String으로 대체
    box1.content = "안녕하세요.";
    String str = box1.content;
    System.out.println(str);

    //Box<Integer> box2 = new Box<Integer>();
    Box<Integer> box2 = new Box<>();  // Box 생성시 타입파라미터 대신 Integer로 대체
    box2.content = 100;
    int value = box2.content;
    System.out.println(value);
  }
}
```

```
안녕하세요.
100
```

제네릭 타입

- 결정되지 않은 타입을 파라미터로 가지는 클래스와 인터페이스
- 선언부에 '< >' 부호가 붙고 그 사이에 타입 파라미터들이 위치

```
public class 클래스명<A, B, ···> { ... }
public interface 인터페이스명<A, B, ···> { ... }
```

- 타입 파라미터는 일반적으로 대문자 알파벳 한 글자로 표현
- 외부에서 제네릭 타입을 사용하려면 타입 파라미터에 구체적인 타입을 지정.
- 지정하지 않으면 Object 타입이 암묵적으로 사용

Product.java

```
package ch13.sec02.exam01;

//제네릭 타입
public class Product<K, M> {  // 타입 파라미터로 K와 M 정의
  //타입 파라미터를 필드 타입으로 사용
  private K kind;
  private M model;

  //타입 파라미터를 리턴 타입과 매개 변수 타입으로 사용
  public K getKind() { return this.kind; }
  public M getModel() { return this.model; }
  public void setKind(K kind) { this.kind = kind; }
  public void setModel(M model) { this.model = model; }
}
```

TV.java

```
package ch13.sec02.exam01;

public class Tv {
}
```

Car

```
package ch13.sec02.exam01;

public class Car {
}
```

GenericExample.java

```
package ch13.sec02.exam01;

public class GenericExample {
  public static void main(String[] args) {
    //K는 Tv로 대체, M은 String으로 대체
    Product<Tv, String> product1 = new Product<>();

    //Setter 매개값은 반드시 Tv와 String을 제공
    product1.setKind(new Tv());
    product1.setModel("스마트Tv");

    //Getter 리턴값은 Tv와 String이 됨
    Tv tv = product1.getKind();
    String tvModel = product1.getModel();

    //K는 Car로 대체, M은 String으로 대체
    Product<Car, String> product2 = new Product<>();

    //Setter 매개값은 반드시 Car와 String을 제공
    product2.setKind(new Car());
    product2.setModel("SUV자동차");

    //Getter 리턴값은 Car와 String이 됨
    Car car = product2.getKind();
    String carModel = product2.getModel();
  }
}
```

Rentable.java 인터페이스에서의 제너릭 사용

```
package ch13.sec02.exam02;

public interface Rentable<P> {
  P rent();
}
```

Home.java

인터페이스에서의 제너릭 사용

```
package ch13.sec02.exam02;

public class Home {
  public void turnOnLight() {
    System.out.println("전등을 켭니다.");
  }
}
```

Car.java

```
package ch13.sec02.exam02;

public class Car {
  public void run() {
    System.out.println("자동차가 달립니다.");
  }
}
```

HomeAgency.java 인터페이스에서의 제너릭 사용

```
package ch13.sec02.exam02;

public class HomeAgency implements Rentable<Home> {  // 타입 파라미터 P를 Home으로 대체
  @Override
  public Home rent() { // 리턴 타입이 반드시 Home이어야 함
    return new Home();
  }
}
```

CarAgency.java

```
package ch13.sec02.exam02;

public class CarAgency implements Rentable<Car> {  // 타입 파라미터 P를 Car으로 대체
  @Override
  public Car rent() {// 리턴 타입이 반드시 Car여야 함
    return new Car();
  }
}
```

GenericExample.java 인터페이스에서의 제너릭 사용

```
package ch13.sec02.exam02;

public class GenericExample {
  public static void main(String[] args) {
    HomeAgency homeAgency = new HomeAgency();
    Home home = homeAgency.rent();
    home.turnOnLight();

    CarAgency carAgency = new CarAgency();
    Car car = carAgency.rent();
    car.run();
  }
}
```

```
전등을 켭니다.
자동차가 달립니다.
```

Box.java

제너릭 타입 생략시 Object로 간주

```
package ch13.sec02.exam03;

public class Box<T> {
  public T content;

  //Box의 내용물이 같은지 비교
  public boolean compare(Box<T> other) {
    boolean result = content.equals(other.content);
    return result;
  }
}
```

GenericExample.java 제너릭 타입 생략시 Object로 간주

```
package ch13.sec02.exam03;

public class GenericExample {
  public static void main(String[] args) {
    Box box1 = new Box();
    box1.content = "100";

    Box box2 = new Box();
    box2.content = "100";

    Box box3 = new Box();
    box3.content = 100;

    boolean result1 = box1.compare(box2);
    System.out.println("result1: " + result1);

    boolean result2 = box1.compare(box3);
    System.out.println("result2: " + result2);
  }
}
```

```
result1: true
result2: false
```

제네릭 메소드

- 타입 피라미터를 가지고 있는 메소드. 타입 파라미터가 메소드 선언부에 정의
- 리턴 타입 앞에 < > 기호 추가하고 타입 파라미터 정의 후 리턴 타입과 매개변수 타입에서 사용

```
public  <A, B, ···>  리턴타입  메소드명(매개변수, ...) { ... }
```

타입 파라미터 정의

- 타입 파라미터 T는 매개값의 타입에 따라 컴파일 과정에서 구체적인 타입으로 대체

```
public <T> Box<T> boxing(T t) { ··· }
```

```
① Box<Integer> box1 = boxing(100);
② Box<String> box2 = boxing("안녕하세요");
```

Box.java

```
package ch13.sec03.exam01;

public class Box<T> {
  //필드
  private T t;

  //Getter 메소드
  public T get() {
    return t;
  }

  //Setter 메소드
  public void set(T t) {
    this.t = t;
  }
}
```

GenericExample.java 제너릭 타입 생략시 Object로 간주

```
package ch13.sec03.exam01;

public class GenericExample {
  //제네릭 메소드
  public static <T> Box<T> boxing(T t) {
    Box<T> box = new Box<T>();
    box.set(t);
    return box;
  }

  public static void main(String[] args) {
    //제네릭 메소드 호출
    Box<Integer> box1 = boxing(100);
    int intValue = box1.get();
    System.out.println(intValue);

    //제네릭 메소드 호출
    Box<String> box2 = boxing("홍길동");
    String strValue = box2.get();
    System.out.println(strValue);
  }
}
```

```
100
홍길동
```

제한된 타입 파라미터

- 모든 타입으로 대체할 수 없고, 특정 타입과 자식 또는 구현 관계에 있는 타입만 대체할 수 있는 타입 파라미터

```
public <T extends 상위타입> 리턴타입 메소드(매개변수, ...) { ... }
```

- 상위 타입은 클래스뿐만 아니라 인터페이스도 가능

```
public <T extends Number> boolean compare(T t1, T t2) {
  double v1 = t1.doubleValue();   //Number의 doubleValue() 메소드 사용
  double v2 = t2.doubleValue();   //Number의 doubleValue() 메소드 사용
  return (v1 == v2);
}
```

GenericExample.java

```
package ch13.sec04;

public class GenericExample {
  //제한된 타입 파라미터를 갖는 제네릭 메소드
  public static <T extends Number> boolean compare(T t1, T t2) {
    //T의 타입을 출력
    System.out.println("compare(" + t1.getClass().getSimpleName() + ", " +
        t2.getClass().getSimpleName() + ")");

    //Number의 메소드 사용
    double v1 = t1.doubleValue();
    double v2 = t2.doubleValue();

    return (v1 == v2);
  }
```

GenericExample.java

```
  public static void main(String[] args) {
    //제네릭 메소드 호출
    boolean result1 = compare(10, 20);
    System.out.println(result1);
    System.out.println();

    //제네릭 메소드 호출
    boolean result2 = compare(4.5, 4.5);
    System.out.println(result2);
  }
}
```

```
compare(Integer, Integer)
false

compare(Double, Double)
true
```

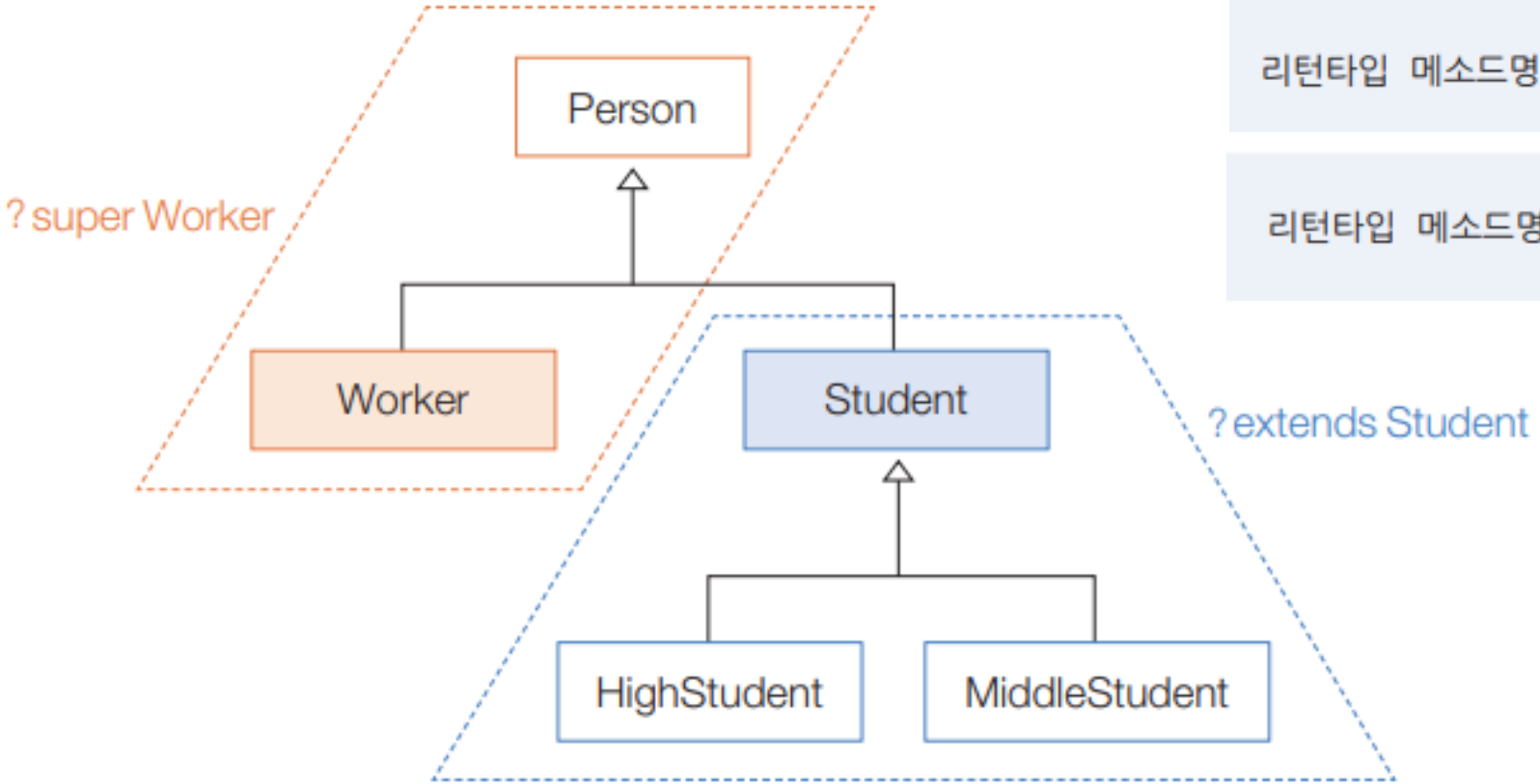
와일드카드 타입 파라미터

- 제네릭 타입을 매개값이나 리턴 타입으로 사용할 때 범위에 있는 모든 타입으로 대체할 수 있는 타입 파라미터.
- ?로 표시

```
리턴타입 메소드명(제네릭타입<? extends Student> 변수) { … }
```

```
리턴타입 메소드명(제네릭타입<? super Worker> 변수) { … }
```

```
리턴타입 메소드명(제네릭타입<?> 변수) { … }
```



Person.java

```
package ch13.sec05;

public class Person {
}

class Worker extends Person {
}

class Student extends Person {
}

class HighStudent extends Student {
}

class MiddleStudent extends Student{
}
```

Application.java

```
package ch13.sec05;

public class Applicant<T> {
  public T kind;

  public Applicant(T kind) {
    this.kind = kind;
  }
}
```

Course.java

```
package ch13.sec05;

public class Course {
  //모든 사람이면 등록 가능
  public static void registerCourse1(Applicant<?> applicant) {
    System.out.println(applicant.kind.getClass().getSimpleName() +
        "이(가) Course1을 등록함");
  }

  //학생만 등록 가능
  public static void registerCourse2(Applicant<? extends Student> applicant) {
    System.out.println(applicant.kind.getClass().getSimpleName() +
        "이(가) Course2를 등록함");
  }

  //직장인 및 일반인만 등록 가능
  public static void registerCourse3(Applicant<? super Worker> applicant) {
    System.out.println(applicant.kind.getClass().getSimpleName() +
        "이(가) Course3을 등록함");
  }
}
```

Course.java

```
package ch13.sec05;

public class GenericExample {
  public static void main(String[] args) {
    //모든 사람이 신청 가능
    Course.registerCourse1(new Applicant<Person>(new Person()));
    Course.registerCourse1(new Applicant<Worker>(new Worker()));
    Course.registerCourse1(new Applicant<Student>(new Student()));
    Course.registerCourse1(new Applicant<HighStudent>(new HighStudent()));
    Course.registerCourse1(new Applicant<MiddleStudent>(new MiddleStudent()));
    System.out.println();

    //학생만 신청 가능
    //Course.registerCourse2(new Applicant<Person>(new Person())); (x)
    //Course.registerCourse2(new Applicant<Worker>(new Worker())); (x)
    Course.registerCourse2(new Applicant<Student>(new Student()));
    Course.registerCourse2(new Applicant<HighStudent>(new HighStudent()));
    Course.registerCourse2(new Applicant<MiddleStudent>(new MiddleStudent()));
    System.out.println();
```

Course.java

```
    //직장인 및 일반인만 신청 가능
    Course.registerCourse3(new Applicant<Person>(new Person()));
    Course.registerCourse3(new Applicant<Worker>(new Worker()));
    //Course.registerCourse3(new Applicant<Student>(new Student()));   (x)
    //Course.registerCourse3(new Applicant<HighStudent>(new HighStudent()));  (x)
    //Course.registerCourse3(new Applicant<MiddleStudent>(new MiddleStudent()));  (x)
  }
}
```

```
Person이(가) Course1을 등록함
Worker이(가) Course1을 등록함
Student이(가) Course1을 등록함
HighStudent이(가) Course1을 등록함
MiddleStudent이(가) Course1을 등록함

Student이(가) Course2를 등록함
HighStudent이(가) Course2를 등록함
MiddleStudent이(가) Course2를 등록함

Person이(가) Course3을 등록함
Worker이(가) Course3을 등록함
```