yonggyo1125 Update README.md f041dbe · last year History

Name	Name	Last commit date
..		
day15	용어 변경	2 years ago
README.md	Update README.md	last year

README.md

강의 동영상 링크

동영상 링크

지네릭스

지네릭스란?

- 지네릭스는 다양한 타입의 객체들을 다루는 메서드나 컬렉션 클래스에 컴파일시의 타입체크(compile-time type check)를 해주는 기능이다.
- 객체의 타입을 컴파일 시에 체크하기 때문에 객체의 타입 안정성을 높이고 형변환의 번거로움이 줄어든다
- 다룰 객체의 타입을 미리 명시해줌으로써 번거로운 형변환을 줄여준다.

지네릭 클래스의 선언

```
class Box {
        Object item;

        void setItem(Object item) {
                this.item = item;
        }
```

```
        Object getItem() {
                return item;
        }
}
```

지네릭 클레스로 변경

```
class Box<T> { // 지네릭 타입 T를 선언
        T item;

        void setItem(T item) {
                this.item = item;
        }

        T getItem() {
                return item;
        }
}
```

- Box에서 T를 '타입 변수(type variable)'라 하며 Type의 첫 글자에서 따온 것이다.
- 타입 변수는 다른 것을 사용해도 되며, 의미 적으로 E(Element), K(Key), V(Value)등을 관례적으로 많이 사용한다.
    - 예) ArrayList의 경우 타입 변수 E는 Element(요소)의 첫글자를 따서 사용
    - 예) Map<K,V>에서 K는 Key(키)를 의미하고, V는 Value(값)을 의미한다.
    - 기호의 종류만 다를 뿐 **임의의 참조형 타입**을 의미한다는 것은 모두 같다.
- 지네릭 클래스가 된 Box 클래스의 객체를 생성할 때는 다음과 같이 참조변수와 생성자에 T 대신에 사용될 실제 타입을 지정해주어야 한다.

```
Box<String> b = new Box<String>(); // 타입 T 대신, 실제 타입을 지정
b.setItem(new Object()); // 에러, String이외의 타입은 지정불가
b.setItem("ABC"); // OK, String 타입이므로 가능
String item = b.getItem(); // 형변환 필요없음
```

- 상기 코드에서 타입 T대신 String 타입을 지정해줬으므로, 지네릭 클래스 Box는 다음과 같이 정의된 것과 같다.

```
class Box<String> { // 지네릭 타입을 String으로 지정
        String item;

        void setItem(String item) {
                this.item = item;
        }

        String getItem() {
                return item;
        }
}
```

- 지네릭이 도입되기 이전의 코드와 호환을 위해, 타입을 지정하지 않는 예전의 방식으로 객체를 생성하는 것이 허용된다. 다만 지네릭 타입을 지정하지 않아서 안전하지 않다는 경고가 발생한다.

```
Box b = new Box(); // OK, T는 Object로 간주한다.
b.setItem("ABC"); // 경고, unchecked or unsafe operation
b.setItem(new Object()); // 경고, unchecked or unsafe operation
```

- 하기와 같이 타입 변수 T에 Object 타입을 지정하면, 타입을 지정하지 않은 것이 아니라 알고 적은 것이므로 경고는 발생하지 않는다.

```
Box<Object> b = new Box<Object>();
b.setItem("ABC"); // 경고발생 안함
b.setItem(new Object()); // 경고 발생 안함
```

지네릭스의 용어

Class Box<T> {}

- **Box<T>** : 지네릭 클래스, 'T의 Box' 또는 'T Box'라고 읽는다.
- **T** : 타입 변수 또는 타입 매개변수.(T는 타입문자)
- **Box** : 원시 타입(raw type)

Box<String> b = new Box<String>();

- 타입 매개변수에 타입을 지정하는 것을 **지네릭타입 호출**이라 한다.
- 지정된 타입 'String'을 **매개변수화된 타입(parameterized type)** 또는 **대입된 타입**이라고 한다.
- 컴파일 후에는 Box과 Box는 이들의 '원시타입'인 Box로 바뀐다.(즉, 지네릭 타입이 제거된다)

지네릭스의 제한

- 타입 매개변수를 가지는 참조 변수와 인스턴스는 타입 매개변수에 동일한 자료형을 입력해야 한다.

```
Box<Apple> appleBox = new Box<Apple>(); // OK, Apple 객체만 저장가능
Box<Grape> grapeBox = new Box<Grape>(); // OK, Grape 객체만 저장가능
```

- static멤버에 타입 변수 T를 사용할 수 없다.
  - T는 인스턴스 변수로 간주되기 때문에 모든 객체에 대해 동일하게 동작해야 하는 static 멤버에서는 사용할 수 없다.
  - static멤버는 타입 변수에 지정된 타입, 즉 대입된 타입의 종류에 관계없이 동일한 것이어야 하기 때문이다.
  - 'Box.item'과 'Box.item'이 다른 것이어서는 안된다.

```
class Box<T> {
      static T item; // 에러
```

```
        static int compare(T t1, T t2) { ... } // 에러
        ...
}
```

- 지네릭 타입의 배열을 생성하는 것도 허용하지 않는다.
  - 지네릭 배열 타입의 참조변수를 선언하는 것은 가능하지만, 'new T[10]' 과 같이 배열을 생성하는 것은 불가
  - 지네릭 배열을 생성할 수 없는 것은 new 연산자 때문인데, 이 연산자는 컴파일 시점에 타입 T가 뭔지 정확히 알아야 한다.
  - 하기 코드에 정의된 Box클래스를 컴파일하는 시점에서는 T가 어떤 타입이 될지 전혀 알수 없기 때문이다.(지네릭은 인스턴스가 만들어질때 T 타입이 결정된다.)
  - new 연산자와 같은 이유로 instanceof 연산자도 T를 피연산자로 사용할 수 없다.

```
class Box<T> {
        T[] itemArr; // OK, T타입의 배열을 위한 참조변수
        ...
        T[] toArray() {
                T[] tmpArr = new T[itemArr.length]; // 에러. 지네릭 배열 생성불
가
                ...
                return tmpArr;
        }
        ...
}
```

지네릭 클래스의 객체 생성과 사용

- 참조변수와 생성자에 대입된 타입(매개변수화된 타입)이 일치해야 한다. 일치하지 않으면 에러가 발생한다.

```
Box<Apple> appleBox = new Box<Apple>(); // OK
Box<Apple> appleBox = new Box<Grape>(); // 에러

Apple이 Fruit의 하위 클래스라고 가정
Box<Fruit> appleBox = new Box<Apple>(); // 에러. 대입된 타입이 다르다.
```

- 두 지네릭 클래스의 타입이 상속관계에 있고, 대입된 타입이 같을 때는 가능하다.

```
FruitBox는 Box의 하위 클래스라고 가정

Box<Apple> appleBox = new FruitBox<Apple>(); // OK. 다형성
```

- 추정이 가능한 경우 타입을 생략할 수 있다.

```
Box<Apple> appleBox = new Box<Apple>();
Box<Apple> appleBox = new Box<>();
```

- 타입 매개변수가 상위 클래스인 배열 또는 컬렉션 프레임워크에서는 하위 클래스를 추가할 수 있다(다형성)

```
Box<Fruit> fruitBox = new Box<Fruit>();
fruitBox.add(new Fruit()); // OK
fruitBox.add(new Apple()); // OK. void add(Fruit item)
```

day15/Box.java

```
package day15;

import java.util.ArrayList;

class Fruit {
	public String toString() {
		return "Fruit";
	}
}

class Apple extends Fruit {
	public String toString() {
		return "Apple";
	}
}

class Grape extends Fruit {
	public String toString() {
		return "Grape";
	}
}

class Toy  {
	public String toString() {
		return "Toy";
	}
}


public class Box<T> {
	ArrayList<T> list = new ArrayList<T>();
	void add(T item) {
		list.add(item);
	}

	T get(int i) {
		return list.get(i);
	}

	int size() {
		return list.size();
	}

	public String toString() {
		return list.toString();
```

```
        }
}
```

day15/FruitBoxEx1.java

```
package day15;

public class FruitBoxEx1 {
        public static void main(String[] args) {
                Box<Fruit> fruitBox = new Box<Fruit>();
                Box<Apple> appleBox = new Box<Apple>();
                Box<Toy> toyBox = new Box<Toy>();
                // Box<Grape> grapeBox = new Box<Apple>(); //  에러. 타입 불일치

                fruitBox.add(new Fruit());
                fruitBox.add(new Apple()); // OK.  void add(Fruit item)

                appleBox.add(new Apple());
                appleBox.add(new Apple());
                // appleBox.add(new Toy()); // 에러. Box<Apple>에는 Apple 만 담을
수 있음.

                toyBox.add(new Toy());
                // toyBox.add(new Apple() // 에러, Box<Toy>에는 Apple을 담을 수 없
다.

                System.out.println(fruitBox);
                System.out.println(appleBox);
                System.out.println(toyBox);
        }
}

실행경과
[Fruit, Apple]
[Apple, Apple]
[Toy]
```

제한된 지네릭 클래스

타입 문자로 사용할 타입을 명시하면 한 종류의 타입만 저장할 수 있도록 제한할 수 있지만, 여전히 모든 종류의 타입을 지정할 수 있는 것에는 변함이 없다. 타입 매개변수 T에 지정할 수 있는 타입의 종류를 제한할 수 있는 방법이 있다.

- **extends** 를 사용하면, 특정 타입의 하위클래스만 대입할 수 있게 제한 할 수 있다.

```
class FruitBox<T extends Fruit> { // Fruit의 하위클래스만 타입으로 지정가능
        ArrayList<T> list = new ArrayList<T>();
        ...
}
```

```
FruitBox<Apple> appleBox = new FruitBox<Apple>(); // OK
FruitBox<Toy> toyBox = new FruitBox<Toy>(); // 에러. Toy는 Fruit의 하위클래스가 아님
```

```
FruitBox<Fruit> fruitbox = new FruitBox<Fruit>();
fruitBox.add(new Apple()); // OK. Apple이 Fruit의 하위클래스
fruitBox.add(new Grape()); // OK. Grape가 Fruit의 하위클래스
```

- 만약 클래스가 아니라 인터페이스를 구현해야 한다는 제약이 필요하다면, 이때에도 **extends**를 사용한다.

```
interface Eatable {}
class FruitBox<T extends Eatable> { ... }
```

- 클래스 Fruit의 하위 클래스이면서 Eatable 인터페이스도 구현해야 한다면 아래와 같이 **&**기호로 연결한다.

```
class FruitBox<T extends Fruit & Eatable> { ... }
```

FruitBox에는 Fruit의 하위클래스면서 Eatable을 구현한 클래스만 타입 매개변수 T에 대입될 수 있다.

day15/fruitboxex2/Box.java

```
package day15.fruitboxex2;

import java.util.ArrayList;

class Fruit implements Eatable {
	public String toString() {
		return "Fruit";
	}
}

class Apple extends Fruit {
	public String toString() {
		return "Apple";
	}
}

class Grape extends Fruit {
	public String toString() {
		return "Grape";
	}
}

class Toy {
	public String toString() {
		return "Toy";
	}
```

```
}

interface Eatable {}

class FruitBox<T extends Fruit & Eatable> extends Box<T> {}

public class Box<T> {
        ArrayList<T> list = new ArrayList<T>();
        void add(T item) {
                list.add(item);
        }

        T get(int i) {
                return list.get(i);
        }

        int size() {
                return list.size();
        }

        public String toString() {
                return list.toString();
        }
}
```

day15/fruitboxex2/FruitBoxEx2.java

```
package day15.fruitboxex2;

public class FruitBoxEx2 {
        public static void main(String[] args) {
                FruitBox<Fruit> fruitBox = new FruitBox<Fruit>();
                FruitBox<Apple> appleBox = new FruitBox<Apple>();
                FruitBox<Grape> grapeBox = new FruitBox<Grape>();
                //FruitBox<Grape> grapeBox = new FruitBox<Apple>(); // 에러, 타입
불일치
                //FruitBox<Toy> toyBox = new FruitBox<Toy>();  // 에러.

                fruitBox.add(new Fruit());
                fruitBox.add(new Apple());
                fruitBox.add(new Grape());
                appleBox.add(new Apple());
                //appleBox.add(new Grape()); // 에러, Grape는 Apple의 하위클래스가
아님
                grapeBox.add(new Grape());

                System.out.println("fruitBox-" + fruitBox);
                System.out.println("appleBox-" + appleBox);
                System.out.println("grapeBox-" + grapeBox);
        }
}
```

실행결과

```
fruitBox-[Fruit, Apple, Grape]
```

```
appleBox-[Apple]
grapeBox-[Grape]
```

와일드 카드

매개변수에 과일박스를 대입하면 주스를 만들어 반환하는 Juicer라는 클래스가 있고, 이 클래스에는 과일을 주스로 만들어서 반환하는 makeJuice()라는 static 메서드가 다음과 같이 정의되어 있다.

```
class Juicer {
        static Juice makeJuice(FruitBox<Fruit> box) { // <Fruit>으로 지정
                String tmp = "";
                for (Fruit f : box.getList()) {
                        tmp += f + " ";
                        return new Juice(tmp);
                }
        }
}
```

Juicer클래스는 지네릭 클래스가 아닌데다, 지네릭 클래스라고 해도 static 메서드에서는 타입 매개변수 T를 매개변수에 사용할 수 없으므로 아예 지네릭스를 적용하지 않던가, 상기 코드와 같이 타입 매개변수 대신, 특정 타입을 지정해줘야 한다.

```
FruitBox<Fruit> fruitBox = new FruitBox<Fruit>();
FruitBox<Apple> appleBox = new FruitBox<Apple>();
...
System.out.println(Juicer.makeJuice(fruitBox)); // OK. FruitBox<Fruit>
System.out.println(Juicer.makeJuice(appleBox)); // 에러.  FruitBox<Apple>
```

지네릭 타입을 'FruitBox'로 고정해 놓으면, 상기 코드에서 알 수 있듯이 'FruitBox'타입의 객체는 makeJuice()의 매개변수가 될 수 없으므로, 다음과 같이 여러가지 타입의 매개변수를 갖는 makeJuice()를 만들 수 밖에 없다.

```
static Juice makeJuice(FruitBox<Fruit> box) {
        String tmp = "";
        for(Fruit f : box.getList()) tmp += f + " ";
        return new Juice(tmp);
}

static Juice makeJuice(FruitBox<Apple> box) {
        String tmp = "";
        for(Fruit f : box.getList()) tmp += f + " ";
        return new Juice(tmp);
}
```

그러나 상기 코드와 같이 오버로딩하면 컴파일 에러가 발생한다. **지네릭 타입이 다른 것만으로는 오버로딩이 성립하지 않기 때문이다.** 지네릭 타입은 컴파일러가 컴파일할 때만 사용하고 제거해버린다. 그래서 위의 두 메서드는 오버로딩이 아니라 **메서드 중복 정의**이다. 이럴 때 사용하기 위해 고안된 것이 바로 **와일드 카드**이다.

- **와일드 카드**는 기호 **?**로 표현하는데, 와일드 카드는 어떠한 타입도 될 수 있다.
- **?**만으로는 Object타입과 다를 게 없으므로, 다음과 같이 'extends'와 'super'로 상한(upper bound)와 하한(lower bound)를 제한할 수 있다.
  - **<? extends T>** : 와일드 카드의 상한 제한. T와 그 하위 클래스/인터페이스들만 가능
  - **<? super T>** : 와일드 카드의 하한 제한. T와 그 상위 클래스/인터페이스들만 가능
  - **<?>** : 제한 없음. 모든 타입이 가능. <? extends Object>와 동일
- 지네릭 클래스와 달리 와일드 카드에는 '&'를 사용할 수 없다. 즉, <? extends T & E>와 같이 할 수 없다.

```
static Juice makeJuice(FruitBox<? extends Fruit> box) {
        String tmp = "";
        for(Fruit f : box.getList()) tmp += f + " ";
        return new Juice(tmp);
}
```

```
FruitBox<Fruit> fruitBox = new FruitBox<Fruit>();
FruitBox<Apple> appleBox = new FruitBox<Apple>();
...
System.out.println(Juicer.makeJuice(fruitBox)); // OK. FruitBox<Fruit>
System.out.println(Juicer.makeJuice(appleBox)); // OK. FruitBox<Apple>
```

day15/wildcard/Box.java

```
package day15.wildcard;

import java.util.ArrayList;

class Fruit {
        public String toString() {
                return "Fruit";
        }
}

class Apple extends Fruit {
        public String toString() {
                return "Apple";
        }
}

class Grape extends Fruit {
        public String toString() {
                return "Grape";
        }
}

class Juice {
        String name;

        Juice(String name) {
                this.name = name + "Juice";
```

```
        }

        public String toString() {
                return name;
        }
}

class Juicer {
        static Juice makeJuice(FruitBox<? extends Fruit> box) {
                String tmp = "";

                for(Fruit f : box.getList()) {
                        tmp += f + " ";
                }

                return new Juice(tmp);
        }
}

class FruitBox<T extends Fruit> extends Box<T> {}

public class Box<T> {
        ArrayList<T> list = new ArrayList<T>();
        void add(T item) {
                list.add(item);
        }

        T get(int i) {
                return list.get(i);
        }

        ArrayList<T> getList() {
                return list;
        }

        int size() {
                return list.size();
        }

        public String toString() {
                return list.toString();
        }
}
```

day15/wildcard/FruitBoxEx3.java

```
package day15.wildcard;

public class FruitBoxEx3 {
        public static void main(String[] args) {
                FruitBox<Fruit> fruitBox = new FruitBox<Fruit>();
                FruitBox<Apple> appleBox = new FruitBox<Apple>();

                fruitBox.add(new Apple());
                fruitBox.add(new Grape());
```

```
                appleBox.add(new Apple());
                appleBox.add(new Apple());

                System.out.println(Juicer.makeJuice(fruitBox));
                System.out.println(Juicer.makeJuice(appleBox));
        }
}
```

실행결과
```
Apple Grape Juice
Apple Apple Juice
```

지네릭 메서드

- 메서드의 선언부에 지네릭 타입이 선언된 메서드를 지네릭 메서드라 한다.
- 선언 위치는 반환 타입 바로 앞이다.

```
static <T> void sort(List<T> list, Comparator<? super T> c)
```

- 지네릭 클래스에 정의된 타입 매개변수와 지네릭 메서드에 정의된 타입 매개변수는 전혀 별개의 것이다. 같은 타입 문자 T를 사용해도 같은 것이 아니다.

```
class FruitBox<T> {
        ...
        static <T> void sort(LIst<T> list, Comparator<? super T> c) {
                ...
        }
        ...
}
```

- 상기 코드에서 지네릭 클래스 FruitBox에 선언된 타입 매개변수 T와 지네릭 메서드 sort()에 선언된 타입 매개변수 T는 타입 문자만 같을 뿐 서로 다르다.
- sort가 static 메서드로 정의됭 있는데, static 멤버에는 타입 매개변수를 사용할 수 없지만, 메서드에 지네릭 타입을 선언하고 사용하는 것은 가능하다.
- 지네릭 메서드는 지네릭 클래스가 아닌 클래스에도 정의될 수 있다.
- 메서드에 선언된 지네릭 타입은 지역변수를 선언한 것과 같다고 생각하면 된다.(지네릭 클래스의 타입이 인스턴스가 만들어질때 결정되지만, 지네릭 메서드는 메서드가 호출될때 타입이 결정된다.)
- 앞서 다루었던 makeJuice()를 지네릭 메서드로 바꾸면 다음과 같다.

와일드 카드 방식

```
static Juice makeJuice(FruitBox<? extends Fruit> box) {
        String tmp = "";
        for(Fruit f : box.getList()) tmp += f + " ";
```

```
        return new Juice(tmp);
}
```

지네릭 메서드

```
static <T extends Fruit> Juice makeJuice(FruitBox<T> box) {
        String tmp = "";
        for(Fruit f : box.getList()) tmp += f + " ";
        return new Juice(tmp);
}
```

- 지네릭 메서드를 호출할 때는 하기와 같이 타입 변수에 타입을 대입해야 한다.

```
FruitBox<Fruit> fruitBox = new FruitBox<Fruit>();
FruitBox<Apple> appleBox = new FruitBox<Apple>();
...
System.out.println(Juicer.<Fruit>makeJuice(fruitBox));
System.out.println(Juicer.<Apple>makeJuice(appleBox));
```

- 그러나 대부분의 경우 컴파일러가 타입을 추정할 수 있기 때문에 생략해도 된다.(상기 코드에서도 fruitBox와 appleBox의 선언부를 통해 대입된 타입을 컴파일러가 추정할수 있다.)

```
System.out.println(Juicer.makeJuice(fruitBox)); // 대입된 타입을 생략할 수 있다.
System.out.println(Juicer.makeJuice(appleBox));
```

- 지네릭 메서드를 호출할 때, 대입된 타입을 생략할 수 없는 경우에는 참조변수나 클래스의 이름을 생략할 수 없다.

```
System.out.println(<Fruit>makeJuice(fruitBox)); // 에러. 클래스 이름 생략불가
System.out.println(this.<Fruit>makeJuice(fruitBox)); // OK
System.out.println(Juicer.<Fruit>makeJuice(fruitBox)); // OK
```

- 같은 클래스 내에 있는 멤버들끼리는 참조변수나 클래스이름, 즉, **this.**나 **클래스이름.**을 생략하고 메서드 이름만으로 호출이 가능하지만, **대입된 타입이 있을때는 반드시 써줘야 한다.**
- 지네릭 메서드는 매개변수의 타입이 복잡할때 단순하게 변경할 수 있다. 와일드카드 방식 매개변수

```
public static void printAll(ArrayList<? extends Product> list, ArrayList<? extends
Product> list2) {
        for (Unit u : list) {
                System.out.println(u);
        }
}
```

지네릭 메서드 형태로 변환

```
public static <T extends Product> void printAll(ArrayList<T> list, ArrayList<T>
list2) {
        for (Unit u : list) {
                System.out.println(u);
        }
}
```

지네릭 타입의 제거

- 컴파일러는 지네릭 타입을 이용해서 소스파일을 체크하고 필요한 곳에 형변환을 넣어준다. 그리고 지네릭 타입을 제거한다. 즉, 컴파일된 파일(*.class)에는 지네릭에 대한 정보가 없는것이다.
- 이렇게 하는 주된 이유는 지네릭이 도입되기 이전의 소스코드와의 호환성을 유지하기 위해서이다.
- JDK1.5부터 지네릭스가 도입되었지만, 아직도 원시 타입을 사용해서 코드를 작성하는 것을 허용하고는 있다.

지네릭 타입 제거과정

1. 지네릭 타입의 경계(bound)를 제거한다. 지네릭 타입이 라면 T는 Fruit로 치환된다. 인 경우는 T는 Object로 치환된다. 그리고 클래스 옆의 선언은 제거된다.
    - 변환 전

```
class Box<T extends Fruit> {
        void add(T t) {
                ...
        }
}
```

    - 변환 후

```
class Box {
        void add(Fruit t) {
                ...
        }
}
```

2. 지네릭 타입을 제거한 후에 타입이 일치하지 않으면 형변환을 추가한다. List의 get()은 Object 타입을 반환하므로 **형변환**이 필요하다.
    - 변환 전

```
T get(int i) {
        return list.get(i);
}
```

    - 변환 후

```
Fruit get(int i) {
        return (Fruit)list.get(i);
}
```

와일드 카드가 포함되어 있는 경우에는 다음과 같이 적절한 타입으로 형변환이 추가된다.

- 변환 전

```
static Juice makeJuice(FruitBox<? extends Fruit> box) {
        String tmp = "";
        for(Fruit f : box.getList()) tmp += f + " ";
        return new Juice(tmp);
}
```

- 변환 후

```
static Juice makeJuice(FruitBox box) {
        String tmp = "";
        Iterator it = box.getList().iterator();
        while(it.hasNext()) {
                tmp += (Fruit)it.next() + " ";
        }
        return new Juice(tmp);
}
```