12-1. Generic

1. 제네릭이란?

- JDK 1.5 부터 도입
- 타입 매개변수
- 컴파일 시 클래스나 메서드의 **타입체크**를 해주는 기능
- 컴파일 시 구체적인 **타입**이 결정되도록 하는 것(타입을 파라미터화)
- 클래스 내부에서 사용할 데이터 타입을 외부에서 지정하는 기법

```
제네릭의 장점
 - 타입 안정성을 제공 (컴파일 시 강한 타입체크,
   == 런타임 시에 발생할 수 있는 오류를 미리 방지)
 - 타입체크와 형변환을 생략할 수 있으므로 코드가 간결해짐 (컴파일러가 추가)
 - 코드의 재사용성이 높아짐
```

2. 제네릭 클래스 사용

모든 객체를 담고 꺼낼 수 있는 Box 클래스

```
// 어떤 객체던 담을 수 있지만,
// Object 로 꺼낸 후 캐스팅을 거쳐야 한다
class Box {
  Object items;
  void setItem(Object item) {
    this.item = item;
  }
  Object getItem() {
    return item;
  }
}
```

제네릭으로 변경

```
// 어떤 객체던 담을 수 있으며, 지정한 타입으로 반환된다(캐스팅 불필요)
// 객체 생성 시점에 실제 타입을 지정하여 효율적인 코드 작성이 가능하다
// (동일한 클래스를 중복으로 만들지 않아도 된다)
class Box<T> {
  T items;
  void setItem(T item) {
    this.item = item;
  }
  T getItem() {
    return item;
  }
}
```

외부에서 타입을 String 으로 지정할 경우

```
// Box<String> stringBox = new Box<>();
class Box {
  String items;
  void setItem(String item) {
    this.item = item;
  }
  String getItem() {
    return item;
  }
}
```

외부에서 타입을 SomeClass 로 지정할 경우

```
// Box<SomeClass> stringBox = new Box<>();
class Box {
  SomeClass items;
  void setItem(SomeClass item) {
    this.item = item;
  }
  SomeClass getItem() {
    return item;
  }
}
```

추가 설명 - 중복된 타입 파라미터를 생략한 다이아몬드(<>) 연산자

- Java7 부터 지원
- 컴파일러가 유추하여 해석

```
List<String> items = new ArrayList<>();
Map<String, String> items2 = new HashMap<>();
```

3. 용어

```
class Box<T> {
  T items;
  void setItem(T item) {
    this.item = item;
  }
  T getItem() {
    return item;
  }
}
```

- Box<T>
  - 제네릭 클래스
  - T의 Box 또는 T Box
- T
  - 타입변수 또는 타입 매개변수
  - 타입을 나타내는 문자일 뿐, 다른 문자로 대체 가능
  - 일반적으로 T 를 쓰는 이유는 Type 의 첫문자를 차용한 것
- Box
  - 원시 타입(raw type)

4. 제네릭 클래스의 선언

- 클래스명 옆에 '<타입 변수>' 추가
- 타입 변수가 여러개인 경우에는 '<>' 사이에 ',' 로 구분하여 사용 (멀티타입 파라미터)
- 클래스 뿐만 아니라 인터페이스도 동일하게 사용 가능 (제네릭 인터페이스)

```
// 멀티타입 파라미터 예
public class HashMap<K,V> extends AbstractMap<K,V> implements ... {

  // ...

  static class Node<K,V> implements Map.Entry<K,V> {
        final int hash;
        final K key;
        V value;
        Node<K,V> next;

        // ...
    }
}
```

- ArrayList 의 경우 타입 변수를 Element 의 첫문자 'E' 를 사용

```
public class ArrayList<E> extends AbstractList<E> implements List<E>, ... {

  // ...

  public E get(int index) {
        rangeCheck(index);

        return elementData(index);
    }
}
```

5. 제네릭의 제한

- static 멤버(변수)에는 사용할 수 없다 (static 변수는 하나의 공유변수이기 때문에 타입이 변경 될 수 없는 개념)
  - static 메서드에서는 제네릭 타입 사용이 가능하다 (제네릭 클래스 선언과 제네릭 메서드 선언은 별개)
- 타입 변수로 배열을 생성할 수 없다
  - 정확히 말하면 new 연산자와 instanceof 연산자에 쓰일 수 없다 (컴파일 시점에 타입 T를 명확하게 알아야하는 키워드이기 때문)

6. 제네릭 클래스의 범위 제한

- 상속 및 구현관계를 이용해서 타입을 제한
- 아래의 예제와 같이 'extends' 를 이용하여 T 타입의 범위를 제한할 수 있음

```
// 타입으로 Fruit 클래스를 상속받은 객체만 사용가능
class FruitBox<T extends Fruit> {
  // ...
}
```

```
// '&' 키워드로 인터페이스도 추가 제한 가능
class FruitBox<T extends Fruit & Comparable> {
  // ...
}
```

7. 와일드 카드

- 제네릭 타입을 매개변수나 리턴타입으로 사용할 때 타입 파라미터를 제한할 목적
- 세가지 형태가 존재

```
<? extends T> : 와일드 카드의 상한 제한.
          - T와 자식만 가능
          - T는 올 수 있는 최상위 타입
          - 내부적으로 명시된 객체 자료형으로 인식
<? super T>   : 와일드 카드의 하한 제한.
          - T와 부모만 가능
          - T는 올 수 있는 최하위 타입
          - 내부적으로 Object로 인식
<?>        : 제한 없음. 모든 타입이 가능.
          - <? extends Object>와 동일
          - 내부적으로 Object로 인식
```

8. 제네릭 메서드

- 메서드의 반환타입 앞에 '<타입 변수>' 추가하여 정의
- 제네릭 클래스에 정의된 타입과는 다른 독립적인 메서드 내에서만 사용되는 개체

```
// Collections 클래스의 sort 메서드
public static <T extends Comparable<? super T>> void sort(List<T> list) {
  list.sort(null);
}
```

```
// T 는 단지 문자열.. 이런식의 정의도 가능
static <MJ> List<MJ> test(List<MJ> list, MJ type) {
    list.add(type);
    return list;
}
```

9. 컴파일 시 동작

- 컴파일러는 제네릭 타입을 이용해 소스파일을 체크 후 필요한곳에 형변환을 추가한 후 제네릭 타입을 제거
- 하위 호환성(과거 코드)을 유지하기 위한 동작