

9장 예외처리 및 제네릭 프로그래밍

목차

- 예외
- 예외 처리 방법
- 제네릭 클래스와 인터페이스
- 제네릭 상속과 타입 한정
- 제네릭 메서드

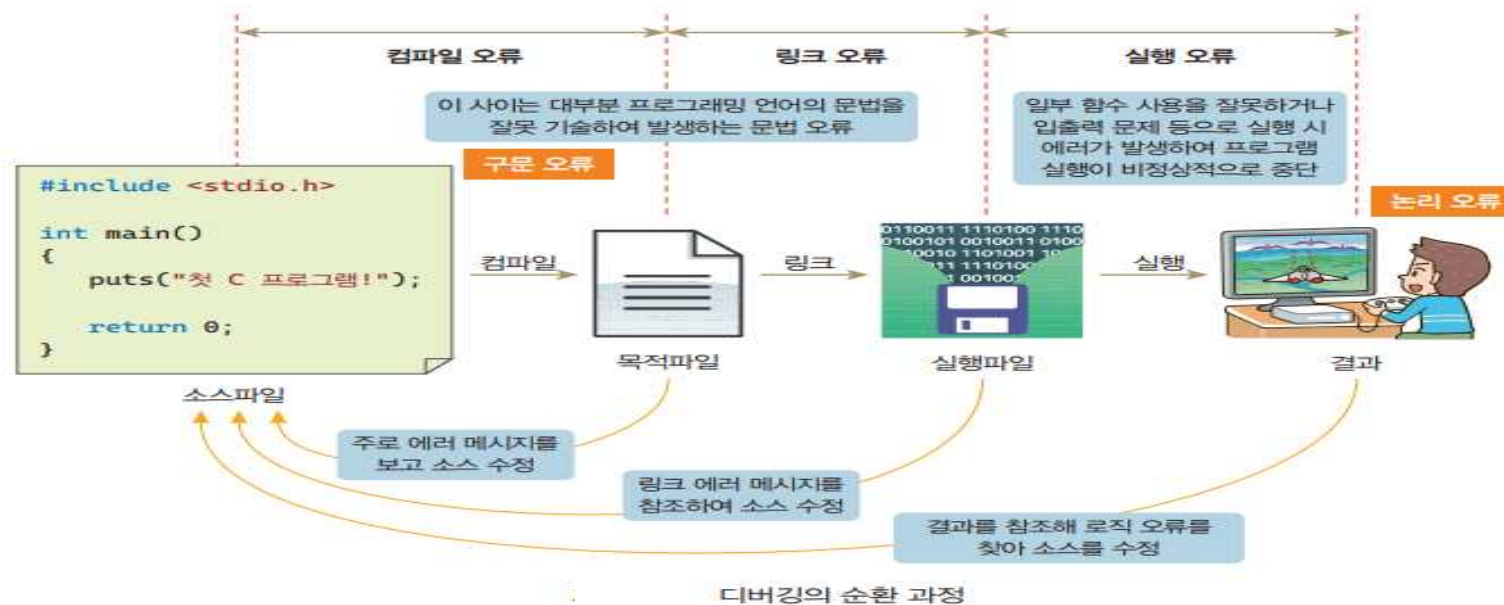
예외

■ 의미

- 에러(error) : 개발자가 해결할 수 없는 치명적인 오류=> 제어 불가능
 - 반복문의 무한 반복, 메모리 부족 등
- 컴파일 오류
 - 문법에 맞지 않게 작성된 코드
 - 컴파일할 때 발견
- 예외(exception) : 실행 중 발생하는 오류: 개발자가 해결할 수 있는 오류 => 제어 가능
 - 자바에서 오동작이나 결과에 악영향을 미칠 수 있는 실행 중 발생하는 오류를 예외라 함
 - 문법적 오류는 사전에 컴파일러에 의해 컴파일 오류로 걸러지지만 실행 중에 오류는 걸러지지않기에 예외 처리를 한다.
 - 정수를 0으로 나누는 경우
 - 배열보다 큰 인덱스로 배열의 원소를 접근하는 경우
 - 존재하지 않는 파일을 읽으려고 하는 경우
 - 정수 입력을 기다리는 코드가 실행되고 있을 때, 문자가 입력된 경우
- 자바에서 예외 처리 가능
 - 예외 발생 -> 자바 플랫폼 인지 -> 응용프로그램에서 전달
 - 응용프로그램이 예외를 처리하지 않으면, 응용프로그램 강제 종료
 - 예외가 발생하면 비정상적인 종료를 막고, 프로그램을 계속 진행할 수 있도록 우회 경로를 제공 하는 것이 바람직

오류

- 오류 또는 에러(error)
 - 프로그램 개발 과정에서 나타나는 모든 문제
- 발생 시점에 따른 구분
 - 컴파일(시간) 오류 : 구문 오류
 - 개발환경에서 오류 내용과 오류 발생 위치를 어느 정도 알려주므로
 - 오류를 찾아 수정하기가 비교적 용이
 - 링크(시간) 오류
 - 컴파일 오류보다 상대적으로 희소
 - main() 함수 이름이나 라이브러리 함수 이름을 잘못 기술하여 발생
 - 실행(시간) 오류
 - 실행하면서 오류가 발생해 실행이 중지되는 경우



예외

■ 예외 종류(실행 중에 발생하는 오류)

- 일반 예외와 실행 예외



■ 실행 예외

- 개발자의 실수로 발생 할 수 있으며, 예외 처리를 하지 않아도 컴파일 할 수 있는 비검사형 예외
- 컴파일러가 예외 처리 여부를 확인하지 않음. 따라서 개발자가 예외 처리 코드의 추가 여부를 결정

■ 일반 예외

- 예외 처리를 하지 않으면 컴파일 오류가 발생하므로 꼭 처리해야하는 검사형 예외
- 컴파일러는 일반 예외가 발생할 가능성을 발견하면 컴파일 오류를 발생
- 개발자는 예외 처리 코드를 반드시 추가

- 예외를 구분하는 이유는 프로그램의 성능 때문에 모든 상황에서 예외 처리를 한다면 과부하가 걸리게
기에 일반 예외만 컴파일러가 확인하고 실행 예외는 코드에서 처리하든지 JVM에 맡기든지 개발자가 선택해야 한다.

예외

■ 실행 예외

- 개발자의 실수로 발생 할 수 있으며, 예외 처리를 하지 않아도 컴파일 할 수 있는 비검사형 예외
- 예외가 발생하면 JVM은 해당하는 실행 예외 객체를 생성
- 실행 예외는 컴파일러가 예외 처리 여부를 확인하지 않음. 따라서 개발자가 예외 처리 코드의 추가 여부를 결정
- 대표적인 실행 예외

실행 예외	발생 이유
ArithmeticException	0으로 나누기와 같은 부적절한 산술 연산을 수행할 때 발생한다.
IllegalArgumentException	메서드에 부적절한 인수를 전달할 때 발생한다.
IndexOutOfBoundsException	배열, 벡터 등에서 범위를 벗어난 인덱스를 사용할 때 발생한다.
NoSuchElementException	요구한 원소가 없을 때 발생한다.
NullPointerException	null 값을 가진 참조 변수에 접근할 때 발생한다.
NumberFormatException	숫자로 바꿀 수 없는 문자열을 숫자로 변환하려 할 때 발생한다.

● 예제

- [sec01/Unchecked1Demo](#)

=> 실행 예외1 : 가져올 토큰이 없는데 요구할 때

- [sec01/Unchecked2Demo](#)

=> 실행 예외2 : 인덱스 범위가 벗어난 경우

```
package sec01;
import java.util.StringTokenizer;
public class UnChecked1Demo {
    public static void main(String[] args) {
        String s = "Time is money";
        StringTokenizer st = new StringTokenizer(s);

        while (st.hasMoreTokens()) {
            System.out.print(st.nextToken() + "+");
        }
        System.out.print(st.nextToken());
        //더 이상 가져올 토큰이 없어 예외 발생
    }
}
```

```
package sec01;
public class UnChecked2Demo {
    public static void main(String[] args) {
        int[] array = { 0, 1, 2 };
        System.out.println(array[3]);
        // 범위를 벗어난 인덱스를 사용해 예외를 발생
    }
}
```

예외

■ 일반 예외

- 예외 처리를 하지 않으면 컴파일 오류가 발생하므로 꼭 처리해야하는 검사형 예외
- 컴파일러는 일반 예외가 발생할 가능성을 발견하면 컴파일 오류를 발생
- 개발자는 예외 처리 코드를 반드시 추가
- 대표적인 일반 예외 예

일반 예외	발생 이유
ClassNotFoundException	존재하지 않는 클래스를 사용하려고 할 때 발생한다.
InterruptedException	인터럽트되었을 때 발생한다.
NoSuchFieldException	클래스가 명시한 필드를 포함하지 않을 때 발생한다.
NoSuchMethodException	클래스가 명시한 메서드를 포함하지 않을 때 발생한다.
IOException	데이터 읽기 같은 입출력 문제가 있을 때 발생한다.

- 예제 : [sec01/CheckedDemo](#)
- 인터럽트가 발생 할 수 있는 실행문을 포함하므로 예외 처리를 해야하는데 예외 처리를 하지 않아 컴파일 오류가 발생한 경우임

```
package sec01;

public class CheckedDemo {
    public static void main(String[] args) {
        Thread.sleep(100);
        //일반 예외가 발생 할 수 있는 코드임에도 예외 처리를
        //하지 않아 컴파일 오류가 발생했음
    }
}
```

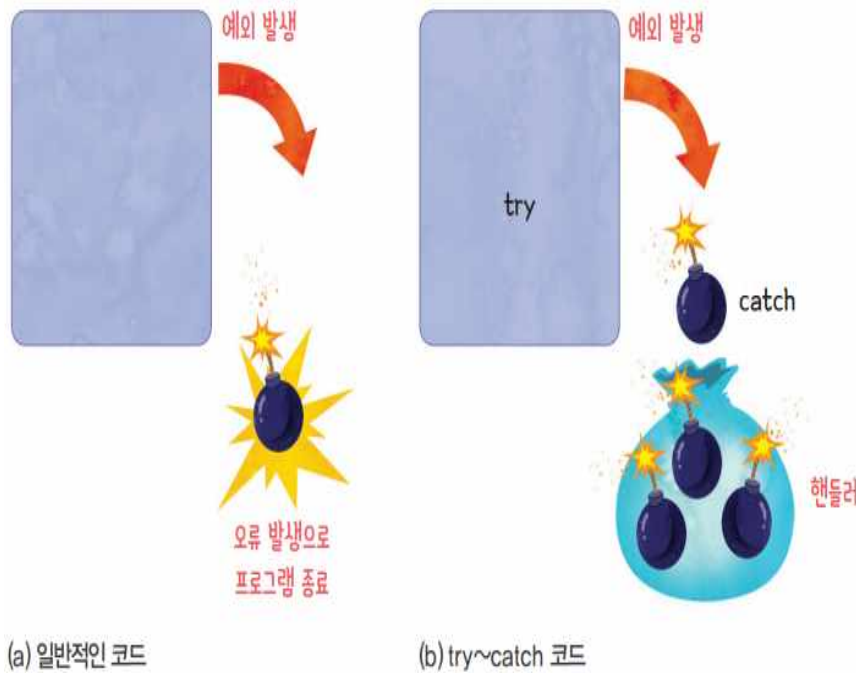
예외 처리 방법

■ 두 가지 방법

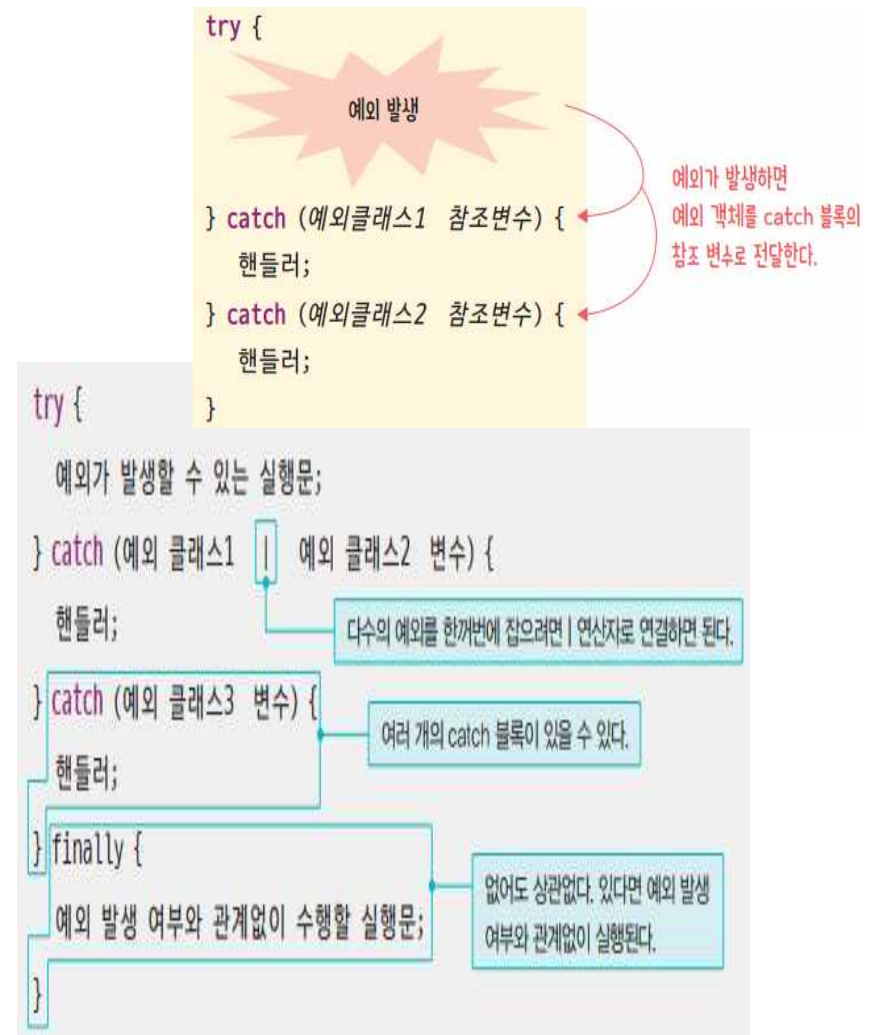
- 예외 처리란 개발자가 작성한 프로그램의 실행 중에 예외가 발생하면 이에 대응하는 것을 말함
- 예외 잡아 처리하기
 - 예외가 발생한 시점에서 발생한 예외 객체를 잡아 바로 처리
 - Try ~ catch ~ finally 문 사용
 - finally문 생략 가능
- 예외 떠넘기기
 - 예외를 발생 시킨 실행문의 상위 코드 블록으로 예외 객체를 떠넘긴다.
 - throws 키워드 사용
 - 예제) 호출된 메서드가 호출한 메인 메서드에 예외 처리를 넘김

예외 처리 방법

■ 예외 잡아 처리하기

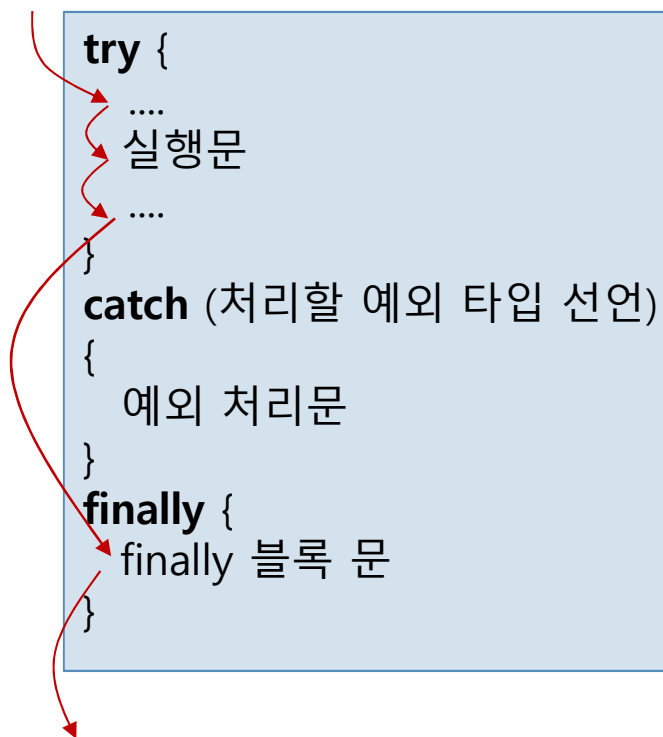


- catch 블록의 순서도 중요

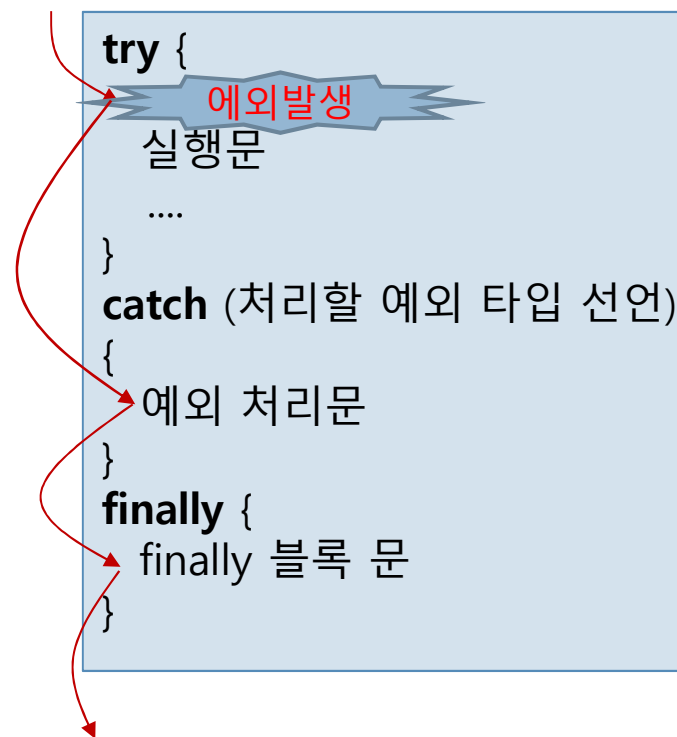


예외에 따른 제어의 흐름

try블록에서 예외가 발생하지 않은 정상적인 경우



try블록에서 예외가 발생한 경우



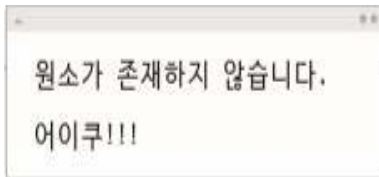
예외 처리 방법

■ 예외 잡아 처리하기

- Throwable 클래스의 주요 메서드

메서드	설명
public String getMessage()	Throwable 객체의 자세한 메시지를 반환한다.
public String toString()	Throwable 객체의 간단한 메시지를 반환한다.
public void printStackTrace()	Throwable 객체와 추적 정보를 콘솔 뷰에 출력한다.

- 예제 : [sec02/TryCatch1Demo](#)



```
package sec02;
public class TryCatch1Demo {
    public static void main(String[] args) {
        int[] array = { 0, 1, 2 }; //1
        try {
            System.out.println("마지막 원소 => " + array[3]); //2
            System.out.println("첫 번째 원소 => " + array[0]); //실행X
        } catch (ArrayIndexOutOfBoundsException e) { //3
            System.out.println("원소가 존재하지 않습니다."); //4
        }
        System.out.println("어이쿠!!!"); //5
    }
}
```

예외 처리 방법

■ 예외 잡아 처리하기

- 예제 : [sec02/TryCatch2Demo](#)



- 예제 : [sec02/TryCatch3Demo](#)

● 다중 catch 블록일 때 catch블록의 순서대로 비교한다. 하지만 예외 객체의 부모가 먼저 나오면 먼저 부모가 가로채서 나중의 catch블록이 실행되지 않는다.

● Exception 객체가 부모이므로 `ArrayIndexOutOfBoundsException`가 실행이 안되어 컴파일 오류가 발생한다.

```
package sec02;
public class TryCatch2Demo {
    public static void main(String[] args) {
        int dividend = 10;
        try {
            int divisor = Integer.parseInt(args[0]);
            System.out.println(dividend / divisor);
        } catch (ArrayIndexOutOfBoundsException e) { //main의 인수가X
            System.out.println("원소가 존재하지 않습니다.");
        } catch (NumberFormatException e) { // 숫자로 바꿀수 없을 때
            System.out.println("숫자가 아닙니다.");
        } catch (ArithmeticException e) { //0으로 나눌수 없을 때
            System.out.println("0으로 나눌 수 없습니다.");
        } finally {
            System.out.println("항상 실행됩니다."); //항상 실행
        }
        System.out.println("종료.");
    }
} // 다중 catch 블록일 때 catch블록의 순서대로 비교한다.
```

```
package sec02;
public class TryCatch3Demo {
    public static void main(String[] args) {
        int[] array = { 0, 1, 2 };
        try {
            int x = array[3];
        } catch (Exception e) {
            System.out.println("어이쿠!!!");
        }
        catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("원소가 존재하지 않습니다.");
        }
        System.out.println("종료.");
    }
}

// 실행 X
//
//
```

예외 처리 방법

■ 예외 잡아 처리하기

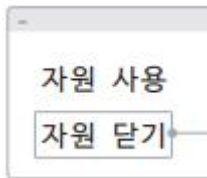
● try~with~resource 문

- try 블록에서 파일 등과 같은 리소스를 사용한다면 try 블록을 실행한 후 자원 반환 필요
- 자원을 닫지 않으면 자원 낭비됨
- 리소스를 관리하는 코드를 추가하면 가독성도 떨어지고, 개발자도 번거롭다.
- JDK 7부터는 예외 발생 여부와 상관없이 사용한 리소스를 자동 반납하는 수단 제공. 단, 리소스는 `AutoCloseable`의 구현 객체

```
try (리소스) {  
    } catch ( ... ) {  
    }
```

- JDK 7과 8에서는 try()의 괄호 내부에서 자원 선언 필요. JDK 9부터는 try 블록 이전에 자원 선언 가능. 단, 선언된 자원 변수는 사실상 final이어야 함

● 예제 : [sec02/TryCatch4Demo](#)



```
package sec02;  
  
public class TryCatch4Demo {  
    public static void main(String[] args) {  
        Reso reso = new Reso();  
  
        try (reso) {  
            reso.show();  
        } catch (Exception e) {  
            System.out.println("예외 처리");  
        }  
    }  
}  
  
class Reso implements AutoCloseable {  
    void show() {  
        System.out.println("자원 사용");  
    }  
  
    public void close() throws Exception {  
        System.out.println("자원 닫기");  
    }  
}
```

예외 처리 방법

■ 예외 떠넘기기

- 메서드에서 발생한 예외를 내부에서 처리하기가 부담스러울 때는 throws 키워드를 사용해 예외를 상위 코드 블록으로 양도 가능
 - 예외를 발생 시킨 실행문의 상위 코드 블록으로 예외 객체를 떠넘긴다.
 - throws 키워드 사용
 - 예제) 호출된 메서드가 호출한 메인 메서드에 예외 처리를 넘김



예외 처리 방법

■ 예외 떠넘기기

● 사용 방법

```
public void write(String filename)
    throws IOException, ReflectiveOperationException {
    // 파일 쓰기과 관련된 실행문 ...
}
```

throws는 예외를 다른 메서드로 떠넘기는 키워드이다.

예외를 1개 이상 선언할 수 있다.

● 예제 : [sec02/ThrowsDemo](#)

호출된 메서드가 호출한 메인 메서드에
예외 처리를 넘김



```
package sec02;
import java.util.Scanner;
public class ThrowsDemo {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        try {
            square(in.nextLine());
        } catch (NumberFormatException e) {
            System.out.println("정수가 아닙니다.");
        }
    }

    private static void square(String s) throws NumberFormatException
    {
        int n = Integer.parseInt(s); //예외가 발생
        System.out.println(n * n);
    }
}
```

- throws 절이 있는 메서드를 오버라이딩할 때는 메서드에서 선언한 예외보다 더 광범위한 검사형 예외를 던질 수 없다. 단 부모 클래스의 메서드에 예외를 떠넘기는 throws절이 없다면 자식 클래스가 메서드를 오버라이딩 할 때 어떤 예외도 떠넘길 수가 없다.

● 자바 API 문서

- 많은 메서드가 예외를 발생시키고 상위 코드로 예외 처리를 떠넘긴다.
- 예를 들면,

```
public static void sleep(long millis, int nanos) throws InterruptedException
```

[quiz_1_식별자] 예외 처리

■ 예외 잡아 처리하기

- 예제 : [sec02/TryCatch2Demo](#)



- 예제 : [sec02/TryCatch3Demo](#)

=> 아래의 결과가 나오도록 프로그램을 수정하십시오.

원소가 존재하지 않습니다.
종료

```
package sec02;
public class TryCatch2Demo {
    public static void main(String[] args) {
        int dividend = 10;
        ??? {
            int divisor = Integer.parseInt(args[0]);
            System.out.println(dividend / divisor);
        } ??? (??? e) { //main의 인수가X
            System.out.println("원소가 존재하지 않습니다.");
        } ??? (??? e) { // 숫자로 바꿀수 없을 때
            System.out.println("숫자가 아닙니다.");
        } ??? (??? e) { //0으로 나눌수 없을 때
            System.out.println("0으로 나눌 수 없습니다.");
        } ??? {
            System.out.println("항상 실행됩니다."); //항상 실행
        }
        System.out.println("종료.");
    }
} // 다중 catch 블록일 때 catch블록의 순서대로 비교한다.
```

```
package sec02;
public class TryCatch3Demo {
    public static void main(String[] args) {
        int[] array = { 0, 1, 2 };
        try {
            int x = array[3];
        } catch (Exception e) {
            System.out.println("어이쿠!!!");
        }
        catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("원소가 존재하지 않습니다.");
        }
        System.out.println("종료.");
    }
}

// 실행 X
//
//
```


[quiz_2_식별자] 예외 처리 : 예외 떠넘기기

- 다음 프로그램의 빈 곳을 채우고 실행 순서를 적으시오.

- 예제 : [sec02/ThrowsDemo](#)

호출된 메서드가 호출한 메인 메서드에 예외 처리를 넘김



```
package sec02;
import java.util.Scanner;
public class ThrowsDemo {
    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);

        ??? {

            square(in.nextLine()); // ???

        } ??? (NumberFormatException e) { // ???
            System.out.println("정수가 아닙니다.");
        }

    }

    private static void square(String s) ??? NumberFormatException { // ???
        int n = Integer.parseInt(s); // ???
        System.out.println(n * n);

    }

}
```

[quiz_3_식별자] 예외 처리

다음과 같은 테스트 프로그램과 실행 결과가 있다. 실행 결과가 나타나도록 MyDate클래스의 빈 곳을 채우고 아래와 같은 프로그램의 오류의 예외 처리를 하시오.

```
package ch9_111;
class MyDate {
    int ??? = 2035;
    int ??? = 12;
    int ??? = 25;
}
public class quiz_3_111 {
    public static void main(String[] args) {
        MyDate d = null;

        System.out.printf("%d년 %d월 %d일\n", d.year, d.month, d.day);
    }
}
```

2035년 12월 25일

Exception in thread "main" java.lang.NullPointerException: Cannot read field "year" because "d" is null

```
package ch9_111;
class MyDate {
    int ??? = 2035;
    int ??? = 12;
    int ??? = 25; }
public class quiz_3_111 {
    public static void main(String[] args) {
        MyDate d = null;
        ??? {
            System.out.printf("%d년 %d월 %d일\n", d.year, d.month, d.day);
        } ??? (??? e) {
            ???;
            ???;
        }
    }
}
```

제네릭 타입

■ 필요성

- 자바는 다양한 종류의 객체를 관리하는 컬렉션이라는 자료구조를 제공
- 초기에는 Object 타입의 컬렉션을 사용
- Object 타입의 컬렉션은 실행하기 전에는 어떤 객체인지?



- 예제(Object 타입)
 - [sec03/Beverage](#), [sec03/Beer](#), [sec03/Boricha](#), [sec03/object/Cup](#)
 - [sec03/GenericClass1Demo](#)



```
package sec03;  
public class Beverage { }
```

```
package sec03;  
public class Beer extends Beverage { }
```

```
package sec03;  
public class Boricha extends Beverage { }
```

```
package sec03.object;  
public class Cup {  
    private Object beverage;  
    public Object getBeverage() {  
        return beverage; }  
    public void setBeverage(Object beverage) {  
        this.beverage = beverage; }  
}
```

```
package sec03;  
import sec03.object.Cup;  
public class GenericClass1Demo {  
    public static void main(String[] args) {  
        Cup c = new Cup();  
        c.setBeverage(new Beer());  
        Beer b1 = (Beer) c.getBeverage();  
        c.setBeverage(new Boricha());  
        // b1 = (Beer) c.getBeverage(); }  
}
```

제네릭 타입

■ 소개

- 제네릭 타입의 의미
 - 하나의 코드를 다양한 타입의 객체에 재사용하는 객체 지향 기법
 - 클래스, 인터페이스, 메서드를 정의할 때 타입을 변수로 사용



- 제네릭 타입의 장점
 - 컴파일할 때 타입을 점검하기 때문에 실행 도중 발생할 오류 사전 방지
 - 불필요한 타입 변환이 없어 프로그램 성능 향상

제네릭 타입

■ 제네릭 타입 선언

```
class 클래스이름<타입매개변수> {  
    필드;  
    메서드;  
}
```

메서드나 필드에 필요한 타입을 타입 매개변수로 나타낸다.

- 타입 매개변수는 객체를 생성할 때 구체적인 타입으로 대체
- 전형적인 타입 매개변수
 - E : Element를 의미하며 컬렉션에서 요소를 표시할 때 많이 사용한다.
 - T : Type을 의미한다.
 - V : Value를 의미한다.
 - K : Key를 의미

타입 매개변수	설명
E	원소(Element)
K	키(Key)
N	숫자(Number)
T	타입(Type)
V	값(Value)

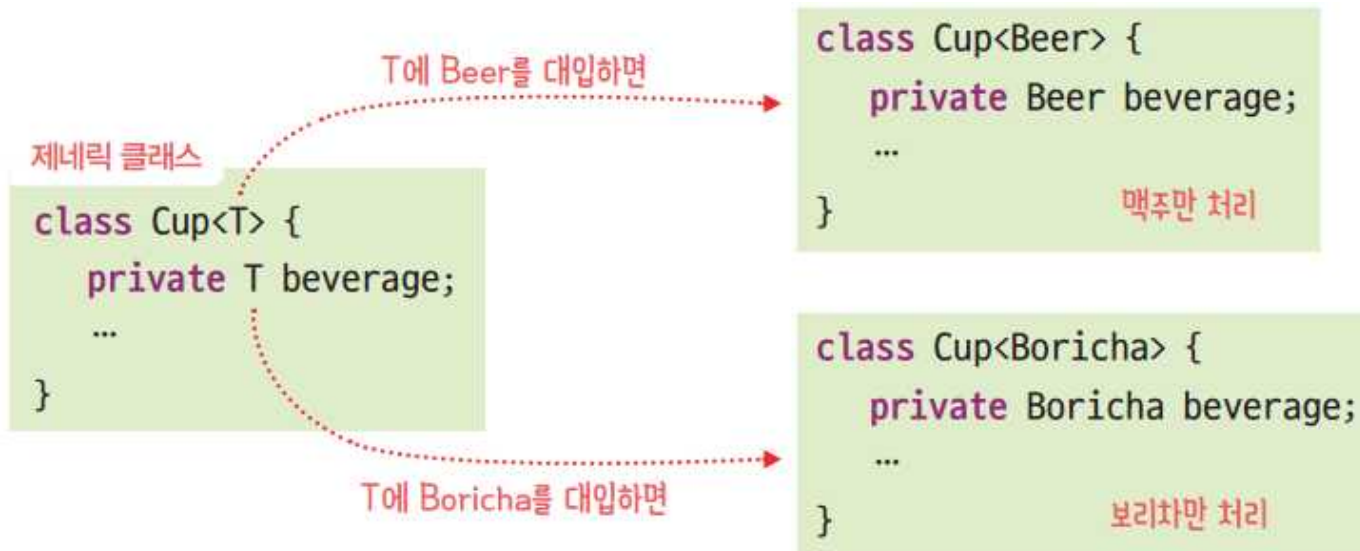
제네릭 타입

■ 제네릭 객체 생성

제네릭클래스 <적용할타입> 변수 = new 제네릭클래스<적용할타입>();

생략할 수 있다.

- <적용할타입>에서 적용할 타입을 생략할 경우 <>를 다이아몬드 연산자라고 함
- 제네릭 클래스의 적용



제네릭 타입 => quiz_4_식별자

- 예제(Object 타입)



- [sec03/Beverage](#), [sec03/Beer](#), [sec03/Boricha](#), [sec03/object/Cup](#)
- [sec03/GenericClass1Demo](#)

- 제네릭 타입 응용



- 예제 : [sec03/generic/Cup](#),
[sec03/GenericClass2Demo](#)

```
package sec03;  
public class Beverage { }
```

```
package sec03;  
public class Beer extends Beverage { }
```

```
package sec03;  
public class Boricha extends Beverage { }
```

```
package sec03.object;  
public class Cup {  
    private Object beverage;  
    public Object getBeverage() {  
        return beverage; }  
    public void setBeverage(Object beverage) {  
        this.beverage = beverage; }  
}
```

```
package sec03;  
import sec03.object.Cup;  
public class GenericClass1Demo {  
    public static void main(String[] args) {  
        Cup c = new Cup();  
        c.setBeverage(new Beer()); //어떤 객체도 가능  
        Beer b1 = (Beer) c.getBeverage(); //Beer타입으로 변환  
        c.setBeverage(new Boricha()); //가능하나 c 객체 모름  
        // b1 = (Beer) c.getBeverage(); //변환 불가, 실행오류 }  
}
```

제네릭

```
package sec03.generic;  
public class Cup<T> {  
    private T beverage;  
    public T getBeverage() {  
        return beverage; }  
    public void setBeverage(T beverage) {  
        this.beverage = beverage; }  
}
```

```
package sec03;  
import sec03.generic.Cup;  
public class GenericClass2Demo {  
    public static void main(String[] args) {  
        Cup<Beer> c = new Cup<Beer>();  
        //Beer타입의 Cup객체 생성  
        c.setBeverage(new Beer()); //Beer객체가 반환되므로  
        Beer b1 = c.getBeverage(); //타입 변환이 필요 없다.  
        // c.setBeverage(new Boricha());  
        // Beer타입의 Cup객체에 Boricha객체를 담을수없다.컴파일 오류  
        b1 = c.getBeverage(); }  
}
```

제네릭 타입

■ Raw 타입의 필요성 및 의미=> 사용하지 말 것!

- 이전 버전과 호환성을 유지하려고 Raw 타입을 지원
- Raw타입 : 제네릭 타입에서 타입 매개변수를 전혀 사용하지 않을 때를 말함
- 제네릭 클래스를 Raw 타입으로 사용하면 타입 매개변수를 쓰지 않기 때문에 Object 타입이 적용
- 예제 : [sec03/GenericClass3Demo](#)

```
package sec03.generic;
public class Cup<T> {
    private T beverage;
    public T getBeverage() {
        return beverage; }
    public void setBeverage(T beverage) {
        this.beverage = beverage; }
}
```

```
package sec03;
import sec03.generic.Cup;
public class GenericClass2Demo {
    public static void main(String[] args) {
        Cup<Beer> c = new Cup<Beer>();
        //Beer타입의 Cup객체 생성
        c.setBeverage(new Beer()); //Beer객체가 반환되므로
        Beer b1 = c.getBeverage(); //타입 변환이 필요 없다.
        // c.setBeverage(new Boricha());
        // Beer타입의 Cup객체에 Boricha객체를 담을수없다.컴파일 오류
        b1 = c.getBeverage(); }
}
```

Raw타입
사용

```
package sec03;
import sec03.generic.Cup;
public class GenericClass3Demo {
    public static void main(String[] args) {
        Cup c = new Cup();
        //구체적인 타입이 없으므로 Raw타입의 제네릭 클래스
        c.setBeverage(new Beer());

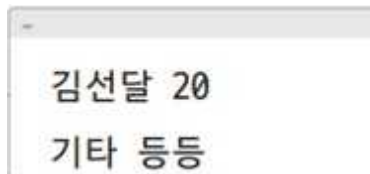
        // Beer beer = c.getBeverage();
        //어떤 타입이 반환되는지 알 수 없으므로 타입 변환이 필요
        Beer beer = (Beer) c.getBeverage();//타입 변환해야함
    }
}
```


제네릭 타입

■ 제네릭 타입 응용

- 예제(2개 이상의 타입 매개변수)

- [sec03/Entry.java](#)
- [sec03/EntryDemo](#)



```
package sec03;
public class Entry<K, V> {
    private K key;
    private V value;
    public Entry(K key, V value) {
        this.key = key;
        this.value = value;
    }
    public K getKey() {
        return key;
    }
    public V getValue() {
        return value;
    }
}
```

```
package sec03;

public class EntryDemo {
    public static void main(String[] args) {
        Entry<String, Integer> e1 = new Entry<>("김선달", 20);
        Entry<String, String> e2 = new Entry<>("기타", "등등");
        // <>는 <String, Integer>와 동일
        // Entry<int, String> e3 = new Entry<>(30, "아무개");
        //타입 변수로 기초 타입을 사용할 수 없다. int의 wrapper클래스로 사용
        System.out.println(e1.getKey() + " " + e1.getValue());
        System.out.println(e2.getKey() + " " + e2.getValue());
    }
}
```

제네릭 상속 및 타입 한정

■ 제네릭 타입의 상속 관계

- 제네릭도 자식 객체를 부모 타입 변수에 대입할 수 있다.
- 예를 들어 ArrayList는 제네릭 클래스이다.

```
ArrayList<Beverage> list = new ArrayList<>();  
list.add(new Beer());           // OK  
list.add(new Boricha());       // OK
```

- 그러나 ArrayList<Beverage> 타입과 ArrayList<Beer>의 경우는 상속 관계가 없다.
- 예제 : [sec04/GenericInheritanceDemo](#)

```
package sec04;  
import java.util.ArrayList;  
  
public class GenericInheritanceDemo {  
    public static void main(String[] args) {  
        ArrayList<Beverage> list1 = new ArrayList<>();  
        list1.add(new Beer());  
        beverageTest(list1); // ArrayList<Beverage>타입의 객체이기 때문에 정상 실행  
  
        ArrayList<Beer> list2 = new ArrayList<>();  
        list2.add(new Beer());  
        // beverageTest(list2);  
        // ArrayList<Beer>는 ArrayList<Beverage>의 자식 타입이 아니기 때문에 컴파일 오류 발생  
    }  
    static public void beverageTest(ArrayList<Beverage> list) { }  
}
```

제네릭 상속 및 타입 한정

■ 제네릭의 제약

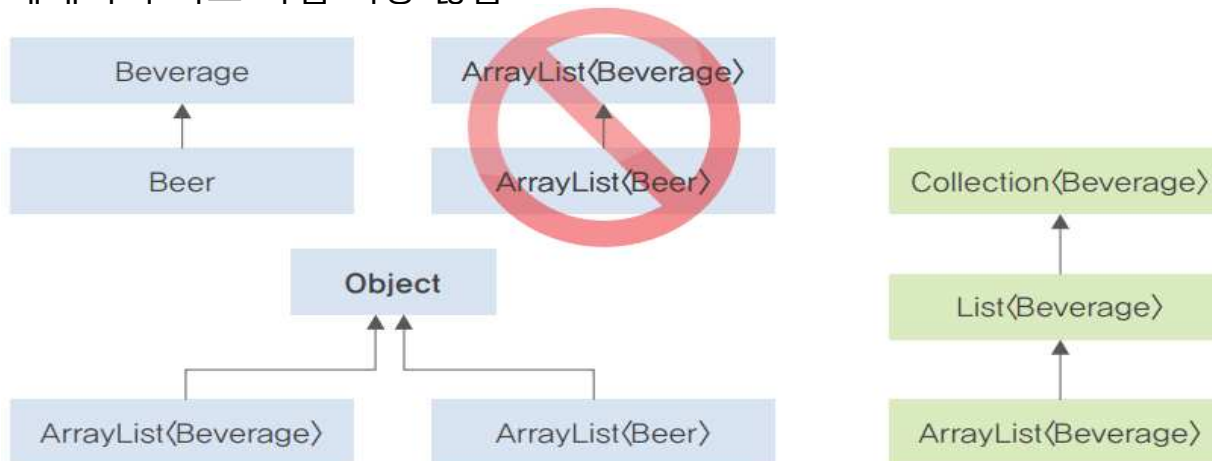
- 기초 타입을 제네릭 인수로 사용 불가

```
Vector<int> vi = new Vector<int>(); // 컴파일 오류. int 사용 불가
```



```
Vector<Integer> vi = new Vector<Integer>(); // 정상 코드
```

- 정적 제네릭 타입 금지
- 제네릭 타입의 인스턴스화 금지. 즉, `T a = new T()` 등 금지
- 제네릭 타입의 배열 생성 금지
- 실행 중에 제네릭 타입 점검 금지. 예를 들어, `a instanceof ArrayList<String>`
- 제네릭 클래스의 객체는 예외로 던지거나 잡을 수 없다
- 제네릭의 서브 타입 허용 않음



제네릭 상속 및 타입 한정

■ 타입 한정

- 타입 매개변수의 범위를 특정 타입으로 제한 할 수 있다. 이때는 extends 키워드로 타입 매개변수의 경계를 전한다.
- 특정 클래스의 자식 타입이나 인터페이스의 구현 타입으로 타입 매개변수를 제한하는 방법

```
<T extends 특정클래스> 반환타입 메서드이름(...) { ... }  
<T extends 인터페이스> 반환타입 메서드이름(...) { ... }
```

부모가 인터페이스라도 extends를 사용한다.

- 예제 : [sec04/bound/BoundedTypeDemo](#)

=> 제네릭 Cup클래스라면

Cup<String>클래스도 사용되지만

음료수만을 위한 제네릭 Cup클래스
로만 사용하려면 Cup클래스의

타입매개변수를 Beverage의

자식 타입으로 제한한다.

```
package sec04;  
public class Beverage { }
```

```
package sec04;  
public class Beer extends Beverage { }
```

```
package sec04;  
public class Boricha extends Beverage{ }
```

```
package sec04.bound;  
import sec04.Beer;  
import sec04.Beverage;  
import sec04.Boricha;  
public class BoundedTypeDemo {  
    public static void main(String[] args) {  
        Cup<Beer> c1 = new Cup<>();  
        Cup<Boricha> c2 = new Cup<>();  
        // Cup<String> c3 = new Cup<>(); //자식 타입이 아니므로 컴파일 오류  
    } }  
class Cup<T extends Beverage> {  
    private T beverage;  
    public T getBeverage() {  
        return beverage;  
    }  
    public void setBeverage(T beverage) {  
        this.beverage = beverage;  
    }  
}
```

제네릭 메서드

■ 의미와 선언 방법

- 타입 매개변수를 사용하는 메서드
- 제네릭 클래스뿐만 아니라 일반 클래스의 멤버도 될 수 있음
- 제네릭 메서드를 정의할 때는 타입 매개변수를 반환 타입 앞에 위치

```
< 타입매개변수 > 반환타입 메서드이름(...) {  
    ...  
}
```

2개 이상의 타입 매개변수도 가능하다.

- 제네릭 메서드를 호출할 때는 컴파일러가 메서드의 인자를 통해 이미 타입을 알고 있으므로 구체적인 타입 생략 가능
- JDK 7과 JDK 8의 경우 익명 내부 클래스에서는 다이아몬드 연산자 사용 불허
- JDK 9부터는 익명 내부 클래스에서도 다이아몬드 연산자 사용 가능

제네릭 메서드

■ 예제

- 배열의 타입에 상관없이 모든 원소를 출력할 수 있는 제네릭 메서드
- [sec05/GenMethod1Demo](#)



```
1 2 3 4 5
H E L L O
5
```

```
package sec05;
public class GenMethod1Demo {
    static class Utils {
        public static <T> void showArray(T[] a) { //T : 타입 매개 변수
            for (T t : a)
                System.out.printf("%s ", t);
            System.out.println();
        }

        public static <T> T getLast(T[] a) { //T : 반환 타입
            return a[a.length - 1];
        }
    }

    public static void main(String[] args) {
        Integer[] ia = { 1, 2, 3, 4, 5 };
        Character[] ca = { 'H', 'E', 'L', 'L', 'O' };

        Utils.showArray(ia); //제네릭 메서드의 구체적 타입을 생략해도 됨
        Utils.<Character> showArray(ca);
        // <Character> : 호출할 때 구체적인 타입을 명시해도 됨

        System.out.println(Utils.getLast(ia));
    }
}
```

제네릭 메서드

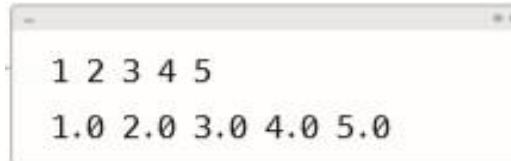
■ 제네릭 타입에 대한 범위 제한

● 사용 방법

```
<T extends 특정클래스> 반환타입 메서드이름(...) { ... }  
<T extends 인터페이스> 반환타입 메서드이름(...) { ... }
```

부모가 인터페이스라도 extends를 사용한다.

● 예제 [sec05/GenMethod2Demo](#)



```
1 2 3 4 5  
1.0 2.0 3.0 4.0 5.0
```

```
package sec05;  
public class GenMethod2Demo {  
    static class Utils {  
        public static <T extends Number> void showArray(T[] a) {  
            //Number클래스의 자식 타입으로 제한  
            for (T t : a)  
                System.out.printf("%s ", t);  
            System.out.println();  
        }  
    }  
    public static void main(String[] args) {  
        Integer[] ia = { 1, 2, 3, 4, 5 };  
        Double[] da = { 1.0, 2.0, 3.0, 4.0, 5.0 };  
        Character[] ca = { 'H', 'E', 'L', 'L', 'O' };  
  
        Utils.showArray(ia);  
        Utils.showArray(da);  
        // Utils.<Character>showArray(ca); // Number클래스의 자식 타입이 아니기에 컴파일 오류  
    }  
}
```

제네릭 만들기

□ 제네릭 클래스와 인터페이스

▣ 클래스나 인터페이스 선언부에 일반화된 타입 추가

```
public class MyClass<T> {  
    T val;  
    void set(T a) {  
        val = a;  
    }  
    T get() {  
        return val;  
    }  
}
```

val의 타입은 T

제네릭 클래스 MyClass 선언, 타입 매개 변수 T

T 타입의 값 a를 val에 지정

T 타입의 값 val 리턴

▣ 제네릭 클래스 레퍼런스 변수 선언

```
MyClass<String> s;  
List<Integer> li;  
Vector<String> vs;
```

□ 구체화

▣ 제네릭 타입의 클래스에 구체적인 타입을 대입하여 객체 생성

▣ 컴파일러에 의해 이루어짐

```
MyClass<String> s = new MyClass<String>(); // 제네릭 타입 T에 String 지정  
s.set("hello");  
System.out.println(s.get()); // "hello" 출력  
MyClass<Integer> n = new MyClass<Integer>(); // 제네릭 타입 T에 Integer 지정  
n.set(5);  
System.out.println(n.get()); // 숫자 5 출력
```

▣ 구체화된 MyClass<String>의 소스 코드

```
public class MyClass<T> {  
    T val;  
    void set(T a) {  
        val = a;  
    }  
    T get() {  
        return val;  
    }  
}
```

→ T가 String으로 구체화

```
public class MyClass<String> {  
    String val; // 변수 val의 타입은 String  
    void set(String a) {  
        val = a; // String 타입의 값 a를 val에 지정  
    }  
    String get() {  
        return val; // String 타입의 값 val을 리턴  
    }  
}
```


[quiz_4_식별자] 제네릭 만들기

- 예제(Object 타입)



- [sec03/Beverage](#), [sec03/Beer](#), [sec03/Boricha](#), [sec03/object/Cup](#)
- [sec03/GenericClass1Demo](#)

- 제네릭 타입 응용



- 예제 : [sec03/generic/Cup](#), [sec03/GenericClass2Demo](#)

```
package sec03;
public class Beverage { }
```

```
package sec03;
public class Beer extends Beverage { }
```

```
package sec03;
public class Boricha extends Beverage { }
```

```
package sec03.object;
public class Cup {
    private Object beverage;
    public Object getBeverage() {
        return beverage; }
    public void setBeverage(Object beverage) {
        this.beverage = beverage; }
}
```

```
package sec03;
import sec03.object.Cup;
public class GenericClass1Demo {
    public static void main(String[] args) {
        Cup c = new Cup();
        c.setBeverage(new Beer()); //어떤 객체도 가능
        Beer b1 = (Beer) c.getBeverage(); //Beer타입으로 변환
        c.setBeverage(new Boricha()); //가능하나 c 객체 모름
        // b1 = (Beer) c.getBeverage(); //변환 불가, 실행오류 }
}
```

제네릭

```
package sec03.generic;
public class Cup<???> {

    //여러 문장

}
```

```
package sec03;
import sec03.generic.Cup;
public class GenericClass2Demo {
    public static void main(String[] args) {
        ???; //Beer타입의 Cup객체 생성

        c.setBeverage(new Beer());
        Beer b1 = ??? ; //타입 변환이 필요 없음
        // c.setBeverage(new Boricha());
        // 오류 이유???
        b1 = c.getBeverage(); }
}
```

[quiz_5_식별자] 제네릭 만들기

다음 프로그램과 실행 결과를 참고하고 다음 조건을 이용하여 Pair 클래스를 작성하시오.

- Pair 클래스는 2개의 필드와 2개의 메서드를 가진다.
- 2개의 필드는 **숫자를 나타내는 어떤 타입도** 될 수 있다.
- 2개의 메서드는 **first()와 second()로 각각 첫 번째 필드 값, 두 번째 필드 값을 반환**한다.

```
package ch9_111;
class Pair<T ??? > {
    private ??? v1, v2;

    public Pair(??? v1, ??? v2) {
        ???;
        ???;
    }

    public ??? first() {
        return v1;
    }

    public ??? second() {
        return v2;
    }
}

public class quiz_5_111{
    public static void main(String[] args) {
        Pair<Integer> p1 = new Pair<>(10, 20);
        Pair<Double> p2 = new Pair<>(10.0, 20.0);

        System.out.println(p1.first());
        System.out.println(p2.second());
    }
}
```

```
10
20.0
```