

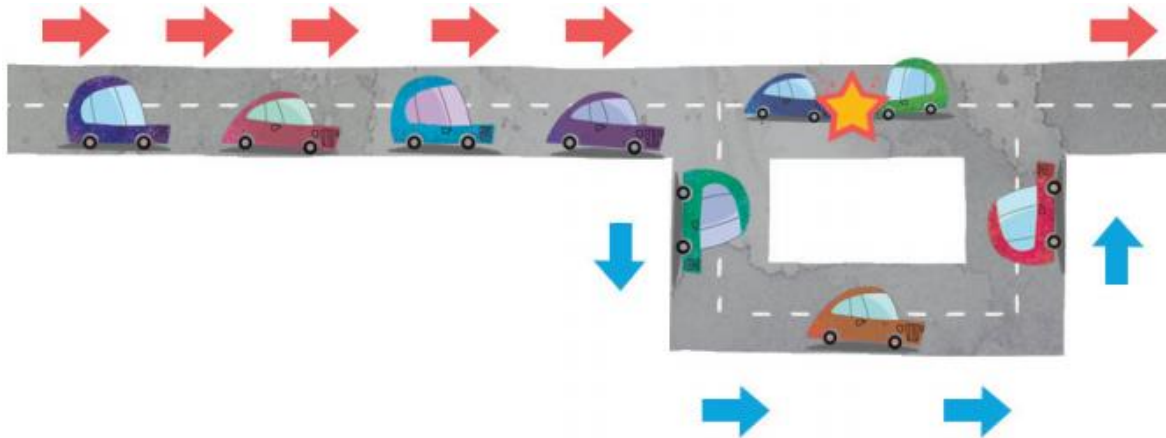
# 예외처리 및 제네릭



# 예외

## ■ 의미

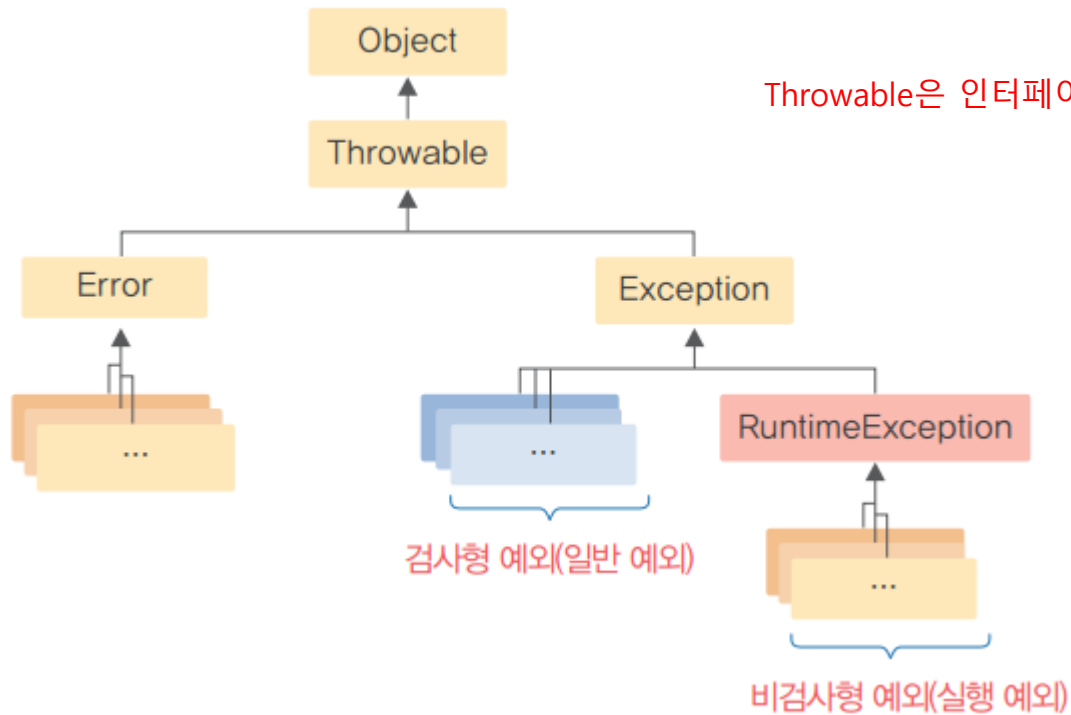
- 에러(error) : 개발자가 해결할 수 없는 치명적인 오류
- 예외(exception) : 개발자가 해결할 수 있는 오류
- 예외가 발생하면 비정상적인 종료를 막고, 프로그램을 계속 진행할 수 있도록 우회 경로를 제공하는 것이 바람직



# 예외

## ■ 종류

- 일반 예외와 실행 예외



# 예외

## ■ 실행 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.



# 예외

- 예제

- [sec01/Unchecked1Demo](#)

```
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());
    }
}
```

- [sec01/Unchecked2Demo](#)

```
public class UnChecked2Demo {
    public static void main(String[] args) {
        int[] array = { 0, 1, 2 };
        System.out.println(array[3]);
    }
}
```



# 예외

## ■ 일반 예외

- 컴파일러는 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](#)

```
public class CheckedDemo {  
    public static void main(String[] args) {  
        // Thread.sleep(100);  
    }  
}
```



# 예외 처리 방법

---

- 두 가지 방법
  - 예외 잡아 처리하기
  - 예외 떠넘기기

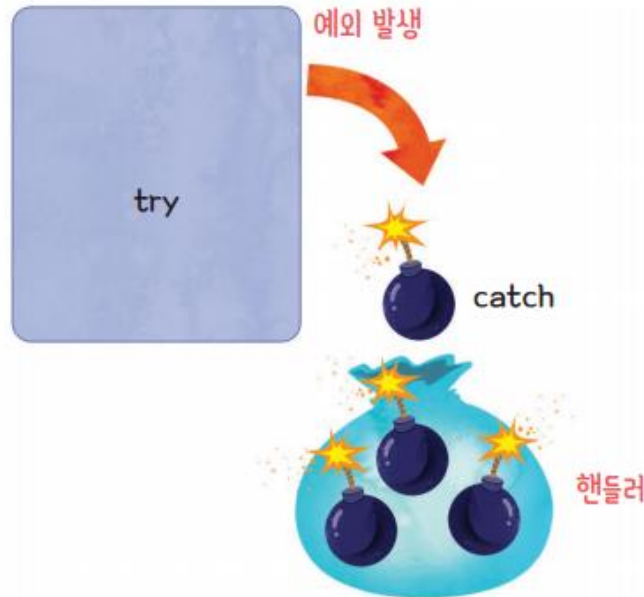


# 예외 처리 방법

## ■ 예외 잡아 처리하기



(a) 일반적인 코드



(b) try~catch 코드





# 예외 처리 방법

## ■ 예외 잡아 처리하기

```
try {
```

예외 발생

```
} catch (예외클래스1 참조변수) {  
    핸들러;
```

```
} catch (예외클래스2 참조변수) {  
    핸들러;
```

```
}
```

예외가 발생하면  
예외 객체를 catch 블록의  
참조 변수로 전달한다.

```
try {
```

예외가 발생할 수 있는 실행문;

```
} catch (예외 클래스1 | 예외 클래스2 변수) {
```

핸들러;

```
} catch (예외 클래스3 변수) {
```

핸들러;

```
} finally {
```

예외 발생 여부와 관계없이 수행할 실행문;

```
}
```

다수의 예외를 한꺼번에 잡으려면 | 연산자로 연결하면 된다.

여러 개의 catch 블록이 있을 수 있다.

없어도 상관없다. 있다면 예외 발생  
여부와 관계없이 실행된다.

- catch 블록의 순서도 중요



# 예외 처리 방법

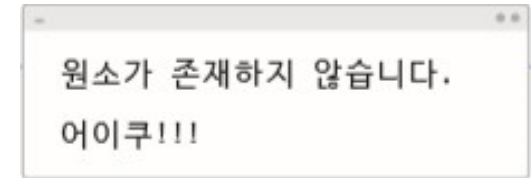
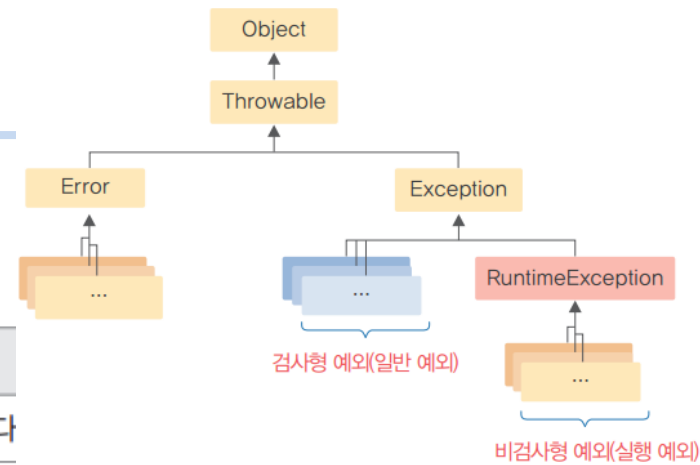
## ■ 예외 잡아 처리하기

- Throwable 클래스의 주요 메서드

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

- 예제 : [sec02/TryCatch1Demo](#)

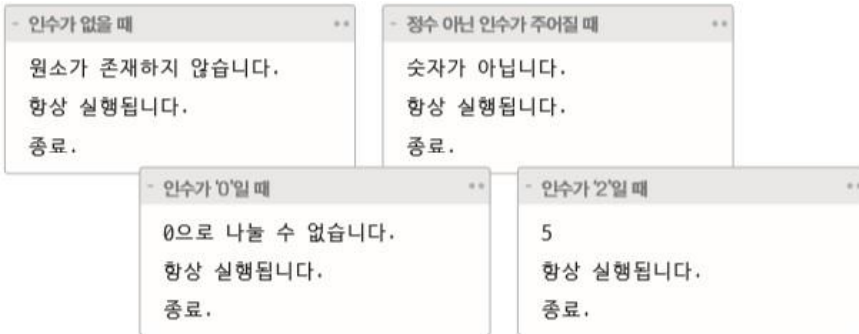
```
public class TryCatch1Demo {  
    public static void main(String[] args) {  
        int[] array = { 0, 1, 2 };  
        try {  
            System.out.println("마지막 원소 => " + array[3]);  
            System.out.println("첫 번째 원소 => " + array[0]);  
        } catch (ArrayIndexOutOfBoundsException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



# 예외 처리 방법

## ■ 예외 잡아 처리하기

- 예제 : [sec02/TryCatch2Demo](#)



```
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) {  
            System.out.println("원소가 존재하지 않습니다.");  
        } catch (NumberFormatException e) {  
            System.out.println("숫자가 아닙니다.");  
        } catch (ArithmeticException e) {  
            System.out.println("0으로 나눌 수 없습니다.");  
        } finally {  
            System.out.println("항상 실행됩니다.");  
        }  
        System.out.println("종료.");  
    }  
}
```



- 예제 : [sec02/TryCatch3Demo](#)

```
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.");  
    }  
}
```



# 예외 처리 방법

## ■ 예외 잡아 처리하기



예제 : [sec02/TryCatch3Demo](#)

```
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.");  
    }  
}
```



# 예외 처리 방법

## ■ 예외 잡아 처리하기

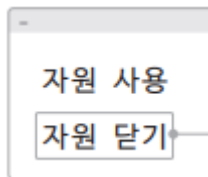
### ● try~with~resource 문

- try 블록에서 파일 등과 같은 리소스를 사용한다면 try 블록을 실행한 후 자원 반환 필요
- 리소스를 관리하는 코드를 추가하면 가독성도 떨어지고, 개발자도 번거롭다.
- JDK 7부터는 예외 발생 여부와 상관없이 사용한 리소스를 자동 반납하는 수단 제공. 단, 리소스는 AutoCloseable의 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](#)



```
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



# 예외 처리 방법

## ■ 예외 떠넘기기

### ● 사용 방법

```
public void write(String filename)
```

```
    throws IOException, ReflectiveOperationException {
```

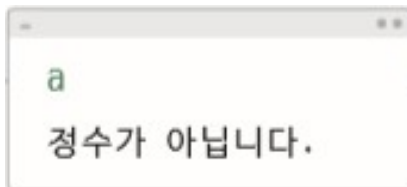
```
    // 파일 쓰기과 관련된 실행문 ...
```

```
}
```

예외를 1개 이상 선언할 수 있다.

throws는 예외를 다른 메서드로 떠넘기는 키워드이다.

### ● 예제 : [sec02/ThrowsDemo](#)



### ● 자바 API 문서

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

```
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);
```

```
        System.out.println(n * n);
```

```
    }
```

```
}
```

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



# 제네릭 타입

## ■ 필요성

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



```
public class Beverage {  
}  
public class Boricha extends Beverage {  
}  
public class Beer extends Beverage {  
}
```

- 예제(Object 타입)



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

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

```
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

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



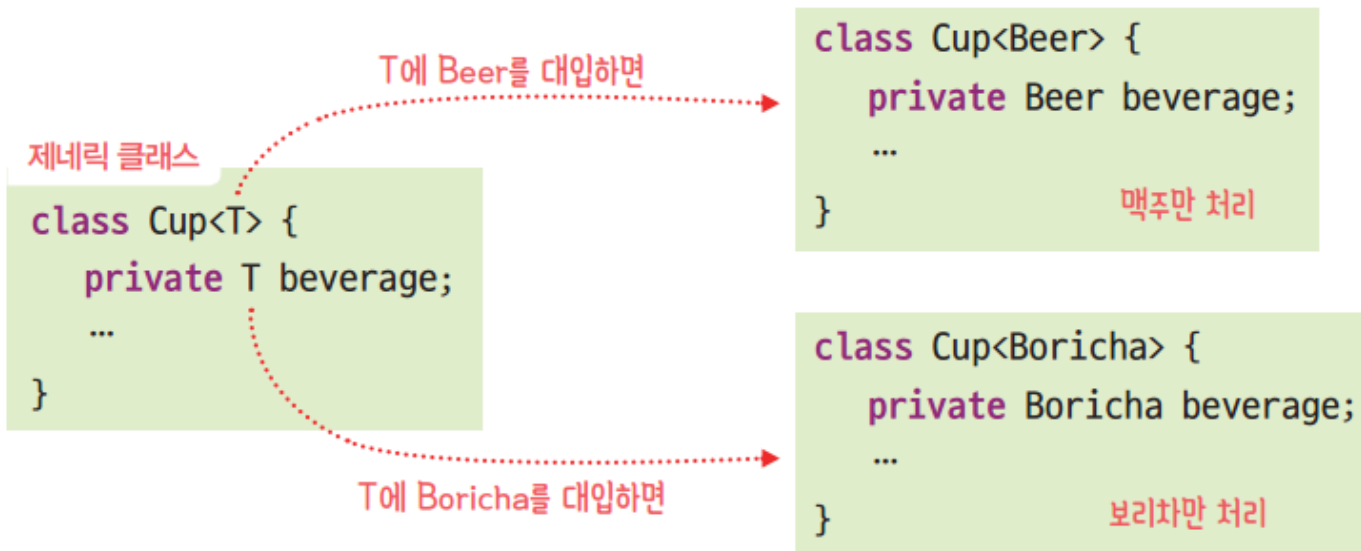
# 제네릭 타입

## ■ 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

## ■ 제네릭 타입 응용



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

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

```
public class GenericClass2Demo {  
    public static void main(String[] args) {  
        Cup<Beer> c = new Cup<Beer>();  
  
        c.setBeverage(new Beer());  
        Beer b1 = c.getBeverage();  
  
        //      c.setBeverage(new Boricha());  
        b1 = c.getBeverage();  
    }  
}
```



# 제네릭 타입

## ■ 제네릭 타입 응용

### ● 예제(2개 이상의 타입 매개변수)

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

```
김선달 20  
기타 등등
```

```
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;  
    }  
}
```

```
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");  
  
        // Entry<int, String> e3 = new Entry<>(30, "아무개");  
  
        System.out.println(e1.getKey() + " " + e1.getValue());  
        System.out.println(e2.getKey() + " " + e2.getValue());  
    }  
}
```



# 제네릭 타입



## Raw 타입의 필요성 및 의미

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

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

```
public class GenericClass3Demo {  
    public static void main(String[] args) {  
        Cup c = new Cup();  
  
        c.setBeverage(new Beer());  
  
        //      Beer beer = c.getBeverage();  
        Beer beer = (Beer) c.getBeverage();  
    }  
}
```



# 제네릭 상속 및 타입 한정

## ■ 제네릭 타입의 상속 관계

### ● 예를 들어

```
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](#)

```
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<Beer> list2 = new ArrayList<>();  
        list2.add(new Beer());  
        //    beverageTest(list2);  
    }  
  
    static public void beverageTest(ArrayList<Beverage> list) { }  
}
```







# 제네릭 상속 및 타입 한정

## ■ 타입 한정

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

부모가 인터페이스라도 extends를 사용한다.

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

```
public class Beverage {  
}  
public class Boricha extends Beverage {  
}  
public class Beer extends Beverage {  
}
```

```
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<>();  
    }  
}
```

```
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.

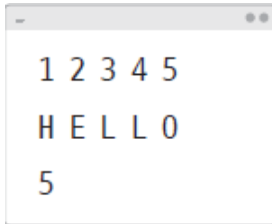
- 제네릭 메서드를 호출할 때는 구체적인 타입 생략 가능
- JDK 7과 JDK 8의 경우 익명 내부 클래스에서는 다이아몬드 연산자 사용 불허
- JDK 9부터는 익명 내부 클래스에서도 다이아몬드 연산자 사용 가능



# 제네릭 메서드

## ■ 예제

- 배열의 타입에 상관없이 모든 원소 출력
- [sec05/GenMethod1Demo](#)



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

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

        public static <T> T getLast(T[] a) {
            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);
        Utils.<Character>showArray(ca);

        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](#)



#### ■ [sec05/GenMethod3Demo](#)



```
public class GenMethod2Demo {  
    static class Utils {  
        public static <T extends Number> void showArray(T[] a) {  
            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);  
    }  
}
```



# 제네릭 메서드

<T **extends** 특정클래스> 반환타입 메서드이름(...) { ... }  
<T **extends** 인터페이스> 반환타입 메서드이름(...) { ... }  
부모가 인터페이스라도 extends를 사용한다.

## ■ 제네릭 타입에 대한 범위 제한

### ● 예제

- [sec05/GenMethod3Demo](#)



```
class Ticket implements Comparable {
    int no;

    public Ticket(int no) {    this.no = no;    }

    public int compareTo(Object o) {
        Ticket t = (Ticket) o;
        return no < t.no ? -1 : (no > t.no ? 1 : 0);
    }
}

public class GenMethod3Demo {
    public static <T extends Comparable> int countGT(T[] a, T elem) {
        int count = 0;
        for (T e : a)
            if (e.compareTo(elem) > 0)        ++count;
        return count;
    }

    public static void main(String[] args) {
        Ticket[] a = { new Ticket(5), new Ticket(3), new Ticket(10), new Ticket(7), new Ticket(4) };
        System.out.println(countGT(a, a[4]));
    }
}
```



# 제네릭 메서드

<T **extends** 특정클래스> 반환타입 메서드이름(...) { ... }  
<T **extends** 인터페이스> 반환타입 메서드이름(...) { ... }  
부모가 인터페이스라도 extends를 사용한다.

## ■ 제네릭 타입에 대한 범위 제한

### ● 예제

#### ■ [sec05/GenMethod3Demo](#)



```
class Ticket implements Comparable {
    int no;

    public Ticket(int no) {    this.no = no;    }

    public int compareTo(Object o) {
        Ticket t = (Ticket) o;
        return no < t.no ? -1 : (no > t.no ? 1 : 0);
    }
}

public class GenMethod3Demo {
    public static <T extends Comparable> int countGT(T[] a, T elem) {
        int count = 0;
        for (T e : a)
            if (e.compareTo(elem) > 0)        ++count;
        return count;
    }

    public static void main(String[] args) {
        Ticket[] a = { new Ticket(5), new Ticket(3), new Ticket(10), new Ticket(7), new Ticket(4) };
        System.out.println(countGT(a, a[4]));
    }
}
```

