

15. 컬렉션 프레임워크(Collection Framework)

1) ArrayList

(1) 배열 리스트의 메서드 (add, size, get, toString)

```
10 /*
2 * 자료구조 : 말 그대로 자료를 저장하는 알고리즘 구조로써 사용 목적이나 특성에 따라 다양한 형태가 존재.
3 *
4 * 컬렉션 프레임워크 : 기존의 자료구조의 적용은 기 개발되거나 알려진 자료구조 알고리즘을 활용하여 개발자가 사용목적에 맞게 직접 구현하여 활용하는
5 *          것이 일반적이었으며 이로 인해 개발시간 지연에 따른 비효율성 대두. 이를 해소하기 위해 개별 자료구조에 따른 공통적인 특성과
6 *          기능을 통합 표준화하여 정의함(여기에는 개별 자료구에 따른 확장성까지 포함)으로써 개발자가 목적에 맞는 자료 구조를 선별하여
7 *          활용할 수 있도록 자바에서 컬렉션 프레임워크라는 자료 구조 클래스 제공. 따라서 개발자는 필요한 자료구조를 직접 구현할 필요없이
8 *          목적과 특성에 맞는 자료 구조를 선별만하여 활용함으로써 생산성과 효율성 증대.
9 *
10 * ArrayList : 재할당이 가능한 동적 배열.
11 */
12 import java.util.ArrayList;           // 크기(size) : ArrayList에 저장되어 있는 요소의 수.
13                                // 용량(capacity) : ArrayList에 재할당하지 않고 저장 가능한 크기 .
```

```

14 public class Main {
15     public static void main(String[] args) {
16         ArrayList<String> arName = new ArrayList<String>();           // 디폴드 생성자 : 크기 10의 용량으로 설정.
17         ArrayList<String> arName1 = new ArrayList<String>(5);          // 5크기의 용량 설정.
18
19         arName.add("A");          // add(E e) : 해당 컬렉션에 요소 추가.
20         arName.add("C");
21         arName.add("D");
22     //     arName.add(1);          // 확정된 제네릭 타입( <String> )과 인수의 형식이 일치해야함.
23
24         arName1.add("A");
25         arName1.add("B");
26         arName1.add("C");
27         arName1.add("D");
28
29         System.out.println(arName.size());           // size() : ArrayList에 저장되어 있는 전체 요소의 수(size)를 리턴하며 -
30         System.out.println(arName1.size());          // 용량과 구분. 용량은 저장가능한 크기로 배열의 할당 크기라고 인식하면 됨. 즉,
31         System.out.println();                      // arName은 최초 디폴드로 10크기의 공간(용량)이 설정되어 현재 저장되어 있는 -
32                                         // 요소의 수가 3으로 확인.
33         for (int i = 0; i < arName.size(); i++) {
34             System.out.print(arName.get(i) + " ");    // get(int index) : 해당 인덱스 위치의 요소값 리턴.
35         }
36         System.out.println('\n');
37
38         for (String s : arName1) {                  // 동적배열이므로 foreach문 형식 지원.
39             System.out.print(s + " ");
40         }
41         System.out.println('\n');
42
43         System.out.println(arName1.toString()); // toString()메서드를 통해 33, 38 행과 같이 루프 이용없이 전체 요소값 리턴 가능.
44         System.out.println();
45
46         arName.add("E");                         // add()메서드를 이용해 데이터 추가 시 자동으로 맨 끝 요소 다음에 추가.
47         System.out.println(arName.toString());
48         System.out.println();
49
50         arName.add(1, "B");                     // add(int index, E e) : 해당 인덱스 위치에 요소값 추가 삽입. 즉, 기존
51         System.out.println(arName.toString());   // -인덱스 위치의 값에 덮어져 삭제되는 것이 아니라 삽입하여 이후의 값이 밀려나-
52                                         // 내부적으로 복사가 진행. 따라서 ArrayList의 경우 자료 추가 시 처리 속도면-
53                                         // 에서 불리.
54     }
55 }

```

(2) 배열 리스트의 메서드 (isEmpty, set, indexOf, lastIndexOf, remove, clear)

```
1 import java.util.ArrayList;
2
3 public class Main {
4     public static void main(String[] args) {
5         ArrayList<String> arName = new ArrayList<String>();
6
7         arName.add("A");
8         arName.add("B");
9         arName.add("B");
10        arName.add("D");
11        arName.add("E");
12        arName.add("F");
13        arName.add("B");
14        arName.add("G");
15
16        System.out.println(arName.isEmpty());           // boolean isEmpty() : 배열이 비었는지 확인.
17
18        arName.set(2, "C");                          // set(int index, E element) : 해당 인덱스 위치의 요소 값 변경.
19        System.out.println(arName.toString());
20
21        System.out.println(arName.indexOf("X"));       // indexOf(Object o) : 전달된 인수 객체와 일치하는 배열 요소의 인덱스 리턴.
22        System.out.println(arName.indexOf("B"));       // -존재하지 않으면 -1 리턴. 순차 검색.
23
24        System.out.println(arName.lastIndexOf("B"));   // lastIndexOf(Object o) : 역순 검색.
25
26        arName.remove(arName.lastIndexOf("B"));       // remove(int index) : 해당 인덱스 위치의 요소 값 삭제.
27        System.out.println(arName.toString());
28
29        arName.clear();                            // clear() : 모든 요소 값 삭제.
30        System.out.println(arName.toString());
31
32        System.out.println(arName.isEmpty());
33    }
34 }
```

2) ArrayList와 LinkedList의 비교

```
1 import java.util.ArrayList;
2 import java.util.LinkedList;
3
4 class Main {
5     public static void main(String[] args) {
6         ArrayList<String> arList = new ArrayList<String>();           // 배열리스트는 메모리의 인접한 공간에 자료를 연속적으로 배치하는 반면 연결리스트는-
7         LinkedList<String> lkList = new LinkedList<String>();          // 힙의 임의의 공간에 각각의 자료를 기억시키되 링크로 연결하여 순서를 구분. 이에 따라
8                                         // -삽입, 삭제 시 배열리스트는 지속적인 자료의 복사가 이루어지는 반면, 연결리스트는
9                                         // -해당 링크의 연결 순서만 조작하면 되므로 상대적 속도가 빠름. 이에 반해 자료를 읽어
10                                         // -들일 때는 배열리스트의 경우 자료의 연속성으로 인해 자료크기에 인덱스만 곱하면 쉽게
11                                         // -구할 수 있는 반면 연결리스트는 처음부터 순서에 맞추어 링크를 따라가며 차례대로 검색
12                                         // -해야하므로 상대적 읽는 속도가 느림.
13
14     long start=System.currentTimeMillis();                                // long currentTimeMillis() : System클래스의 정적 메서드로써 특정 연도의 날짜와 시간을-
15
16
17
18     for (int i = 0; i < 100000; i++) {                                     // 기점으로 현재까지 경과한 시간을 1000분의 1초 단위로 쪼개어 일반화된 조리턴. 이는 사람이 인지할 수
19         arList.add(0, String.valueOf(i));                                    // -없는 일차원적인 시간 표현이 되므로 일반적으로 시간의 조사 보다는 조사 시점간의 경과 시간을 조사하는
20
21     }                                                               // -목적으로 활용.
22
23
24     long end=System.currentTimeMillis();                                     // ArrayList arList의 인덱스 0의 위치에 반복적으로 문자열 값 삽입. 반복적인 삽입에 따른 연쇄적인
25     System.out.println("ArrayList의 자료 추가 작업 시간 : " + (end-start)/1000. + "초"); // -값 복사 발생. 제네릭 타입이 String으로 설정되어 있으므로 기본형 i를 참조형으로 변환하기 위해 -
26                                         // valueOf()메서드 이용.
                                         // String valueOf(int i) : String클래스의 정적 메서드로 인수로 전달된 기본형 값을 참조형인 -
                                         // 문자열로 변환하여 리턴.
```

```

27     start=System.currentTimeMillis();
28     for (int i = 0; i < 100000; i++) {
29         arList.get(i);
30     }
31     end=System.currentTimeMillis();
32     System.out.println("ArrayList의 자료 읽기 작업 시간 : " + (end-start)/1000. + "초");
33
34
35     start=System.currentTimeMillis();
36     for (int i = 0; i < 100000; i++) {
37         lkList.add(0, String.valueOf(i));
38     }
39
40
41
42
43
44     end=System.currentTimeMillis();
45     System.out.println("LinkedList의 자료 추가 작업 시간 : " + (end-start)/1000. + "초");
46
47
48     start=System.currentTimeMillis();
49     for (int i = 0; i < 100000; i++) {
50         lkList.get(i);
51     }
52     end=System.currentTimeMillis();
53     System.out.println("LinkedList의 자료 읽기 작업 시간 : " + (end-start)/1000. + "초");
54 }
```

// 배열 리스트는 인덱스에 타입크기를 곱하여 요소 검색하므로
 // -연결 리스트에 비해 상대적인 검색 속도가 빠름.

// 연결 리스트는 순차적인 자료의 추가 시마다 마지막 저장된 데이터의 참조를 별도로 저장. 따라서 이후 -
 // 요소의 끝에 새로운 자료를 추가 시 해당 추가 요소의 전위 링크에 직전 마지막 별도로 저장된 데이터의 -
 // 참조를 지정. 당행과 같이 지정된 위치에 자료를 삽입 시에는 삽입할 자료의 참조를 직전 데이터의 후위 -
 // 링크에 넣고 삽입할 자료의 전위 링크에 직전 데이터의 참조를 또한 삽입할 위치에 존재하는 기존 요소의 -
 // 참조를 삽입할 자료의 후위 링크에 지정하고 마지막으로 삽입할 위치에 존재하는 기존 요소의 전위 링크에 -
 // -삽입할 자료의 참조를 지정하는 비교적 단순한 조작만으로 데이터 추가가 가능함으로써 자료의 추가, -
 // 삽입 속도가 빠른 것이 장점.

3) 반복자(Iterator)

(1) 반복자의 개념과 활용

```
1① /*
2 * 연결리스트의 경우 전체 요소를 검색하기 위해 각각의 전체 요소들을 순차적으로 순회하면서 각각의 요소에 접근하기 위해 항상 최초 위치부터
3 * 링크를 따라 값을 얻는 내부 메커니즘을 통한 이중 루프로 구성되어 있어 비효율적. 이러한 비효율성을 제거하려면 마지막 액세스한 데이터의
4 * 후위 데이터 링크주소를 미리 저장해두었다가 그 주소를 통해 바로 접근함으로써 해결 가능. 이러한 1차적인 데이터 순회를 통해 각 데이터의
5 * 후위 링크 주소를 순차적인 형태로 객체화하여 최초 데이터의 주소를 후위 링크 주소로 하는 직접 가상 데이터의 참조를 리턴해 주는 역할을 하는
6 * 것이 반복자(Iterator). 따라서 반복자 또한 일종의 컬렉션이며 모든 컬렉션은 Iterable인터페이스로부터 구현되어 iterator()
7 * 메서드를 통해 반복자 객체를 리턴.
8 * ( 주의 : iterator()메서드를 통해 반복자 객체를 얻는 것이지 실제 요소의 참조를 얻는 것은 아님에 유의.
9 *         실제 요소의 참조 주소는 반복자 객체를 통한 next()메서드를 호출함으로써 그 주소를 통해 값 리턴 )
10 * 또한 이러한 반복자를 통해 다양한 컬렉션들에 대한 액세스 방식을 표준화여 코드의 일관성을 확보하려는 목적으로 내포.
11 */
12② import java.util.Iterator;
13 import java.util.LinkedList;
14
15 class Main {
16     public static void main(String[] args) {
17         LinkedList<String> lkList = new LinkedList<String>();
18
19         lkList.add("A");
20         lkList.add("B");
21         lkList.add("C");
22         lkList.add("D");
23         lkList.add("E");
24         lkList.add("F");
25
26         Iterator<String> iterator=lkList.iterator();      // iterator()를 통해 반복자 객체 리턴. 즉, 1차적인 순회를 통해 각 데이터의 후위 링크
27         // -주소를 순차적인 형태로 배열하여 최초 데이터의 주소를 후위 링크 주소로 가지는 직접 -
28         // 가상 데이터의 참조를 포함한 객체를 리턴.
29
30         System.out.println(iterator.next());           // next() : 반복자 객체의 현 요소가 가지는 후위 링크 주소를 리턴하며 매 호출 시마다 후위 링크 주소로-
31         System.out.println(iterator.next());           // 이동하여 해당 요소값 리턴. 26행의 반복자 객체를 통해 next()메서드를 호출함으로써 최초 요소에 대한
32         // -직접 가상 데이터의 참조를 받아 가상의 현 요소가 가지는 후위 링크의 주소가 최초 요소를 가리킴으로써 -
33         // 최초 요소의 값을 리턴.
34
```

```

35     while (iterator.hasNext()) {           // boolean hasNext() : 현 요소에 대한 후위 링크 주소가 있는지 조사. next()로 조사 시-
36         System.out.println(iterator.next()); // 후위 링크 주소가 없다면 즉, 더 이상 요소값이 없다면 NoSuchElementException을-
37     }                                     // 발생시키므로 hasNext()메서드를 통해 미리 점검 필수.
38     System.out.println();
39
40
41
42     while (iterator.hasNext()) {           // 35행의 루프 완료 후 반복자의 참조는 맨 끝 요소를 가리키고 있으므로 루프 집입 자체가 불가.
43         System.out.println(iterator.next());
44     }
45     System.out.println();
46
47
48
49     iterator=lkList.iterator();           // 반복자를 재순회하기 위해서는 이와 같이 iterator()메서드를 통해 반복자 객체를 재 생성해야 함.
50     while (iterator.hasNext()) {          // -35행에서 모든 데이터에 대한 접근이 끝난 후 자동으로 최초의 시점으로 포인터를 이동시키면 이렇게-
51         System.out.println(iterator.next()); // -객체를 재 생성해야 하는 불편함을 덜 수 있겠지만 그렇게 하지 않는 가장 큰 이유는 데이터를 단순히 -
52     }
53     System.out.println();                // 읽는 것은 문제가 되지 않으나 반복자 객체를 통한 데이터 순회 중 데이터를 삽입, 삭제하는 경우 그 -
54
55
56
57
58
59
60     iterator=lkList.iterator();           // 데이터에 대한 후위 링크 주소가 추가, 삭제되어야하는데 만일 반복자 객체를 그대로 사용한다면 -
61     while (iterator.hasNext()) {          // 변경된 데이터에 대한 불법적인 접근이나 접근 불가로 이어지는 사태가 발생할 수 있어 당행과 같이 -
62         String temp=iterator.next();      // iterator()메서드를 통해 반복자 객체의 초기화 즉, 1차적인 데이터 재 순회를 통한 변경된 -
63         if(temp=="B" || temp=="C") {       // 데이터를 포함한 후위 링크 주소를 재조사해야 함. 반복자는 모든 데이터에 대한 후위 링크 주소를 -
64             iterator.remove();            // 객체화하여 저장하는 역할만을 담당하며, 호출 시마다 포인터를 순차적으로 따라가면서 읽는 역할은 -
65         }
66     }
67
68     iterator=lkList.iterator();           // -next()메서드가 담당. 따라서 최초 데이터에 대한 직전 가상 데이터의 참조는 반복자 객체를 통해 -
69     while (iterator.hasNext()) {          // 직접 얻을 수 없고 그 객체에 대한 next()메서드를 통해서만 추출 가능함에 항상 주의.
70         System.out.println(iterator.next());
71     }
72 }
73 }
```

(2) 반복자 적용전과 비교

```
1 import java.util.Iterator;
2 import java.util.LinkedList;
3
4 class Main {
5     public static void main(String[] args) {
6         LinkedList<String> lkList = new LinkedList<String>();
7
8         long start=System.currentTimeMillis();
9         for (int i = 0; i < 100000; i++) {
10             lkList.add(0, String.valueOf(i));
11         }
12         long end=System.currentTimeMillis();
13         System.out.println("LinkedList의 자료 추가 작업 시간 : " + (end-start)/1000. + "초");
14
15
16         start=System.currentTimeMillis();
17         for (int i = 0; i < 100000; i++) {
18             lkList.get(i);
19         }
20         end=System.currentTimeMillis();
21         System.out.println("LinkedList의 자료 읽기 작업 시간 : " + (end-start)/1000. + "초");
22
23
24         Iterator<String> iterator=lkList.iterator();      // 반복자를 통한 1차적인 데이터 순회로 후위 데이터 링크만 조사하여 객체화한
25                                         // -데이터를 저장한 후 next()메서드로 포인터 이동을 하여 후위 링크 주소만
26                                         // -바로 바로 참조.
27         start=System.currentTimeMillis();
28         while (iterator.hasNext()) {
29             iterator.next();
30         }
31         end=System.currentTimeMillis();
32         System.out.println("LinkedList에 대한 반복자 적용 자료 읽기 작업 시간 : " + (end-start)/1000. + "초");
33
34
35         start=System.currentTimeMillis();
36         for (String string : lkList) {                  // 데이터를 읽는 작업에 한하여 향상 for(Enhanced for)를 이용하면 반복자와 비슷한-
37             String a=string;                            // 성능을 유지하면서 재 순회 시 반복자 객체를 재 생성하고 요소값을 얻기 위해 별도의 -
38         }                                              // 메서드를 호출하는 불편함이나 요소의 범위를 벗어나는 등의 예외 방지 가능. 단, 전연한
39         end=System.currentTimeMillis();                // -바와 같이 읽기만 가능하며 전체 요소를 대상으로 후위 링크 주소만 따라가므로 속도는-
40                                         // 빠르나 별도의 연속적인 범위 지정이 불가.
41         System.out.println("LinkedList에 대한 Enhanced for 적용 자료 읽기 작업 시간 : " + (end-start)/1000. + "초");
42     }
43 }
```

4) 제네릭(Generic)

(1) Raw type

```
1 import java.util.ArrayList;
2
3 class Main {
4     public static void main(String[] args) {
5         ArrayList arrayList = new ArrayList(); // ArrayList : 자바5 이전의 컬렉션 형태로 ArrayList<Type>의 Raw Type.
6         int t1; // 데이터 저장 시 Object타입으로 저장하여 데이터 유형에 관계없이 저장 가능. -
7         String t2; // 문제는 Object타입으로 저장되어 있어 컴파일 시 타입 체크가 안되는 즉, -
8         arrayList.add(1); // 실행중에야 타입 체크가 되기 때문에 오 캐스팅에 따른 예외 발생 위험성을 내포하며-
9         arrayList.add("a"); // 데이터를 꺼낼 때마다 타입에 맞춰 정확히 캐스팅을 해야하는 불편을 감수해야 함.
10
11
12 //     t1=arrayList.get(0);
13 //     t1=(int)arrayList.get(0); // 배열리스트에 저장된 데이터 타입이 Object타입이므로 9, 10행과 같이 타입이 달라도 모두-
14 //     System.out.println(t1); // 저장이 가능하지만 배열 요소값을 12, 16행과 같이 특정 타입에 직접 대입 시 컴파일 에러가 -
15 //                             // 발생하므로 13, 17행과 같이 반드시 타입에 맞추어 캐스팅 필수.
16 //     t2=arrayList.get(1);
17 //     t2=(String)arrayList.get(1);
18 //     System.out.println(t2);
19
20 //     t2=(String)arrayList.get(0); // 배열리스트에 저장된 데이터가 Object형이고 변수 t2의 타입이 String형이므로 String형-
21 // } // 으로 캐스팅하는 것 자체는 문제가 되지 않음. 즉, 컴파일 중에는 전혀 문제가 발생되지 않은나
22 } // -실제 데이터 타입은 정수형임으로 인해 실행중에야 오 캐스팅이식별되어 예외를 발생시키는 -
23 // 심각한 문제 초래. 이러한 문제를 해결하기 위해 제네릭(Generic)이 도입됨으로써 캐스팅에-
24 // 따른 불편과 불안정성을 해소함과 동시에 컴파일 중 타입체크가 바로 가능해짐으로써 타입 안정성
25 // -확보.
```

(2) 제네릭 구조 분석

<1> 임의 타입별 자료구조 클래스의 생성

```
1 class IntBox {                                // 정수형 값을 저장하는 클래스로써 12행 같이 생성자의 실 인수의 타입이 불일치하면
2     int value;                               // -당연히 컴파일 에러처리가 되며 이는 7행의 메서드를 통한 저장 또한 동일. 이러한
3
4     IntBox(int value) { this.value = value; } // -형태의 다양한 데이터 형식을 저장 가능한 클래스를 만든다면 기본형 뿐만이 아니라
5
6     int get() { return value; }               // -임의의 객체도 저장 가능해야 하므로 무한대의 클래스가 생성되어야 하는 비현실적
7     void set(int value) { this.value = value; } // -상황 발생.
8 }
9
10 class Main {
11     public static void main(String[] args) {
12         //         IntBox errIntBox = new IntBox(12.34);
13         IntBox intBox = new IntBox(1234);
14
15         System.out.println(intBox.get());
16     }
17 }
```

<2> Raw type의 실체

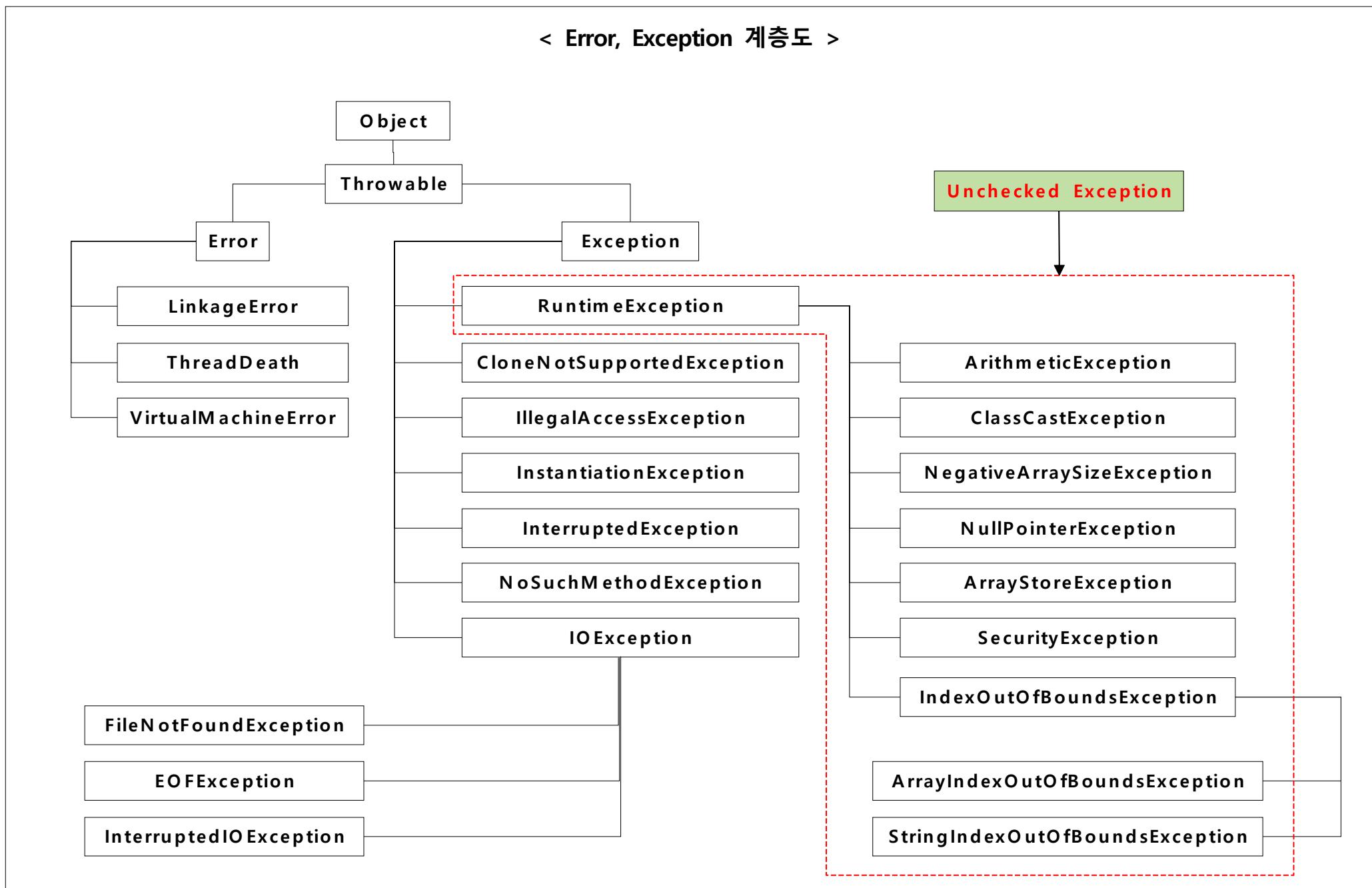
```
1 class ObjectBox {           // 로우타입은 실제 이와 같이 Object형으로 데이터를 받아들여 모든 데이터를
2     Object value;          // -받아들일 수 있도록 설계. 하지만 상기한 바와 같이 캐스팅이 강제되어야하고
3                         // -컴파일 중 타입체크가 되지 않는 타입 불안정성 내재.
4
5     ObjectBox(Object value) { this.value = value; }
6
7     Object get() { return value; }
8     void set(Object value) { this.value = value; }
9 }
10
11 class Main {
12     public static void main(String[] args) {
13         ObjectBox intBox = new ObjectBox(1234);
14
15         int t1 = (int)intBox.get();
16         System.out.println(t1);
17
18
19         ObjectBox dblBox = new ObjectBox(12.34);
20
21         double t2 = (double)dblBox.get();
22         System.out.println(t2);
23     }
24 }
25 }
```

<3> 제네릭 타입

```
1  /*
2   *  < 형식인수와 실인수의 구분 >
3   *  - 형식 인수(Type parameter), 매개 변수
4   *  - 실 인수 (Type argument)
5   *
6   *  < 제네릭 인수 타입 >
7   *  제네릭 형식 인수 타입은 강제적인 것은 아니며 관례상 아래와 같이 대문자 이니셜 형태로 표기. 객체 생성 시
8   *  "클래스명 <실 인수 타입 >"과 같은 형태로 실 인수 타입을 지정하면 클래스 내 모든 제네릭 형식 인수 타입이
9   *  설정된 곳에 대체 삽입되어 실행. 실 인수 타입은 참조형만 가능하며 기본형의 경우 래퍼 클래스를 이용. 형식
10  *  인수 타입 설정은 "클래스명<T, K>"와 같은 형태로 쉼표로 구분하여 다수 설정 가능.
11  *
12  *  T : Type
13  *  E : Element
14  *  K : Key
15  *  V : Value
16  *  N : Number
17  */
18 class GenericBox<T> {    // 제네릭 타입의 클래스 선언 => "클래스명 <형식인수 타입 >"
19     T value;
20     int temp=5;           // 제네릭은 형식인수가 실인수로 대체되는 일종의 매크로 타입일뿐 클래스 내에서
21                         // -무조건 제네릭 타입만 써야 하는 것은 아니며 다른 타입의 사용도 가능.
22
23     GenericBox(T value) { this.value = value; }
24
25     T get() { return value; }
26     void set(T value) { this.value = value; }
27 }
28 }
```

```
29 class Main {  
30     public static void main(String[] args) {  
31         GenericBox<Integer> intBox = new GenericBox<Integer>(1234); // 모든 제네릭 타입 "T"에 실 인수 타입  
32                                         // -"Integer"로 대체되어 객체 생성.  
33 //         String t3 = intBox.get();  
34         int t1 = intBox.get();           // 제네릭 타입 설정으로 인해 캐스팅이 필요치 않으며 33행과 같이 타입 불일치 시 바로 컴파일  
35         System.out.println(t1);       // -에러 처리되어 타입 안정성 확보. 일반적인 참조형의 경우 기본형과의 상호 직접 대입이 -  
36                                         // 불가능하나 래퍼클래스의 경우 예외적으로 오토박싱과 오토언박싱이 수행되어 상호 직접 대입  
37                                         // -가능.  
38  
39         GenericBox<Double> dblBox = new GenericBox<Double>(12.34);  
40         double t2 = dblBox.get();  
41         System.out.println(t2);  
42  
43 //         GenericBox<Double> dblBox2 = new GenericBox<Double>(1234); // 생성자의 실인수 타입이 제네릭 실인수 타입과  
44 }                                         // -불일치 시 바로 컴파일 에러 처리.  
45 }
```

16. 예외(Exception)



- Error : 장치결함 또는 메모리 부족 등 프로그램 외적 요인에 의해 프로그램 내부적으로 복구가 불가능한 심각한 오류.
- Exception : 프로그램 내부적으로 오류 발생 조건만 해결된다면 복구가 가능한 경미한 오류.
- Checked Exception : 문법적으로 예외 처리가 강제된 예외로 프로세스 충돌, 사용자의 오 입력 등 프로그램 외적 요인에 의해 발생 가능한 예외.
- Unchecked Exception : 예외 처리를 생략 가능한 예외로 개발자의 단순 실수 등 프로그램 자체의 논리적 오류에 의해 발생 가능한 예외.

1) 예외 복구(try ~ catch 블럭을 통한 예외 처리)

```
1 public class Main {  
2     public static void main(String[] args) {  
3         int ar[]={1,2,3,4,5};  
4         int ix=12;  
5  
6         try {  
7             System.out.println(ix/2);           // 예외가 발생하지 않는다면 try블럭 내부를 모두 처리 후  
8             System.out.println("요기두 처리"); // -모든 catch블럭을 건너 뛰고 다음 구문 실행.  
9  
10        } catch(ArithmeticException e){      // catch블럭은 중첩이 가능하며 선별 예외에서 넓은  
11            // -범위의 예외 순으로 배치. switch문과 유사.  
12  
13            System.out.println(e.getMessage()); // getMessage() : 예외 객체  
14            // -발생에 대한 원인이 되는 값리턴.  
15        System.out.println("연산끝");  
16        System.out.println();  
17  
18        try {  
19            System.out.println(ix/0);          // 예외가 발생하면 해당 예외 객체가 자동 생성되고 그 예외 객체가 catch블럭의 인수로  
20            System.out.println("요기는??"); // -전달되어 예외 처리 후 다음 구문 실행. 단, try블록 내 예외 발생 시점 이후의 -  
21        } catch(ArithmeticException e){      // 처리는 무시됨.  
22  
23            System.out.println(e.getMessage());  
24        } catch(Exception e){  
25            System.out.println(e.getMessage());  
26        }  
27        System.out.println("연산끝");  
28        System.out.println();  
29    }
```

```

30     try {
31         System.out.println(ix/0);
32     } catch(ArrayIndexOutOfBoundsException e){           // 예외 객체가 상위 선별 예외에 해당하지 않으면
33         System.out.println(e.getMessage());             // -하위 catch문으로 예외 객체 전달.
34     } catch(Exception e){
35         System.out.println(e.getMessage());
36     }
37     System.out.println("연산끝");
38     System.out.println();
39
40     try {
41         System.out.println(ar[ix]);
42     } catch(ArrayIndexOutOfBoundsException e){
43         System.out.println(e.getMessage());
44         System.out.println(e.toString());               // toString() : getMessage()보다 자세한 예외-
45                                         // 정보 리턴.
46
47         e.printStackTrace();                         // printStackTrace() : 가장 자세한 예외 정보 출력.
48     } catch(Exception e){                        // -리턴 값이 없고 내부적으로 예외 정보 직접 출력.
49         System.out.println(e.getMessage());
50         System.out.println(e.toString());
51         e.printStackTrace();
52     }
53     System.out.println("연산끝");
54     System.out.println();
55
56     try {
57         System.out.println(ar[ix]);
58         System.out.println(ix/0);
59     } catch(ArrayIndexOutOfBoundsException | ArithmeticException e){ // catch( 예외 클래스1 | 예외 클래스2 형식인수 ) :
60         e.printStackTrace();                                     // " | "파이프라인 연산자를 통해 동일 catch블럭 내 예외 선별 가능.
61     }
62     System.out.println("연산끝");
63     System.out.println();
64 }
65 }
```

2) 예외 회피

(1) throw를 통한 임의예외 발생

```
1 public class Main {  
2     public static void main(String[] args) {  
3         try {  
4             ClassCastException ex = new ClassCastException(); // 예외 객체를 생성한다 해서 예외가 발생하는 것은 아니며 8행과 같이-  
5                                         // throw를 통해 예외 객체를 던져야만 당행의 예외 객체가 활성화 되어  
6                                         // -임의 예외 발생.  
7  
8         //           throw ex; // throw를 통해 예외 객체를 던짐으로써 당행에서 예외 발생. 예외 발생이 예상되는 시점은 반드시-  
9         } catch (ClassCastException e) { // 예외 처리를 함으로써 예외로 인한 프로그램의 중단을 방지해야 하므로 당행의 경우 try블럭 내 -  
10            e.printStackTrace(); // 필히 존재해야 함이 원칙이나 예외 발생의 주체 원인이 되는 예외 객체의 생성은 4행과 같이 반드시-  
11        } // try블럭 내 생성해야 하는 것은 아님. 예외 객체를 예외 블럭 내 뿐만이 아닌 외부에서도 참조해야  
12        // -한다면 오히려 예외 처리 블럭 외부에 선언을 해야함이 옳으나 4행과 같이 예외 블럭 내부에 생성을  
13        System.out.println("실행 완료"); // -하면 예외 블럭 내 지역 변수로 설정되어 외부에서의 참조가 불가. 또한 외부에서 예외클래스 타입  
14    } // 변수를 먼저 선언하고 예외 블럭 내부에서 객체를 생성하여 대입하여도 예외가 발생되는 경우만 -  
15 } // 객체가 생성되므로 외부에서 예외 변수를 참조해봐야 초기화되지 않은 변수를 참조함으로써 컴파일 -  
16 } // 예러 발생.
```

(2) 예외 종류에 따른 try~catch 적용

```
1 public class Main {
2     public static void main(String[] args) {
3         // try {
4             throw new ArithmeticException();
5         } catch (Exception e) {
6             e.printStackTrace();
7         }
8
9         // try {
10            throw new Exception(); // 실행 가능 여부와 관계없이 체크드 예외는 try~catch블럭이 문법적으로
11        } catch (Exception e) { // -강제되어 미적용 시 당행과 같이 컴파일 자체가 거부됨. 하지만 4행과 -
12            e.printStackTrace(); // 같은 언체크드 예외는 실행 가능 여부와 관계없이 문법적으로 try~catch
13        }
14    }
15 }
```

(3) throws(언체크드 예외)

```
1 public class Main {
2     public static void main(String[] args) {
3         call(); // 8행에서 throws를 통해 예외 처리를 위임받았지만 해당 예외 타입이 언체크드 예외이므로 -
4         System.out.println("다음 문장"); // 당행에 대한 예외 처리는 강제성이 없음. 또한 throws절은 예외를 명시만 할 뿐 예외를 -
5         System.out.println("메서드 호출"); // 발생시키는 것은 아니고 메서드 내에서 예외 객체가 실제 발생되지도 않았으므로 실행에는-
6     } // 문제가 되지 않음.
7
8     static void call() throws RuntimeException { // 메서드() throws 예외 클래스 : throws옆에 전달(위임)할 예외 클래스 타입을-
9     } // 명시함으로써 메서드 내 예외 발생 객체에 대한 예외 처리 생략 가능. 즉, 메서드에 내 예외-
10    // 발생으로 인한 예외 처리를 throws를 통해 호출부로 예외 처리를 위임함으로써 해당 메서드
11 } // -내에서 예외처리를 할 필요가 없지만 해당 메서드를 호출한 호출부에 대한 예외 처리는 필수.
12
13
14
15 // -단, 이 때 전달할 예외 타입이 언체크드 예외인 경우 실행 여부와 관계없이 호출부에 대한 -
// 예외 처리(try~catch)는 강제적이지 않으나, 체크드 예외인 경우에는 반드시 try ~ -
// catch블럭 내 호출부를 구성해야 함. throws는 예외를 명시할 뿐 예외를 발생시키지는 -
// 않음에 유의.
```

(4) throws(체크드 예외)

```
1 public class Main {  
2     public static void main(String[] args) {  
3         // try {  
4             call();  
5             System.out.println("다음 문장"); // call()메서드 내에서 실제 예외가 발생했다면 예외 객체를 던기 위해 catch블럭으로  
6         } // 바로 전환됨으로써 당행의 처리가 안되겠지만, 13행에서 예외에 대한 명시만 하였고 실제  
7         catch (InterruptedException e) { // 예외 객체가 발생되지는 않았으므로 체크드 예외에 대한 예외처리만 강제할 뿐 예외가 발생된  
8             e.printStackTrace(); // 것은 아니기 때문에 당행은 정상적으로 실행.  
9         }  
10        System.out.println("메서드 호출");  
11    }  
12  
13     static void call() throws InterruptedException{ // 이와 같이 throws에 명시한 예외 객체가 체크드 예외인 경우 호출부에서 반드시 -  
14     } // 예외 처리가 문법적으로 강제됨. 여기서 throws는 예외가 발생할 것임을 명시하여  
15 } // 예외 처리를 강제할 뿐 예외를 발생시키는 것은 아님에 유의.  
16
```

(5) main메서드에서의 throws

<1> main메서드의 체크드 예외 처리

```
1 public class Main {  
2     public static void main(String[] args) {  
3         System.out.println("대기 2초");  
4         try {  
5             Thread.sleep(2000);          // 체크드 예외인 인터럽트 예외에 대한 예외 처리 필수. 단, 싱글  
6         } catch (InterruptedException e) { // -스레드인 경우 인터럽트가 걸릴 소지가 적고 예외 발생이 되지-  
7             e.printStackTrace();      // 않는다는 확신이 있다면 throws를 통해 JVM에게 예외를 명시  
8         }                            // -함으로써 예외 처리 생략 가능.  
9         System.out.println("작업 완료");  
10    }  
11 }
```

<2> main메서드의 체크드 예외 처리 생략

```
1 public class Main {  
2     public static void main(String[] args) throws InterruptedException{  
3         System.out.println("대기 2초");  
4         Thread.sleep(2000);          // 2행과 같이 throws를 통해 JVM에게 예외를 떠넘김으로써 당행에 대한 예외 처리  
5         System.out.println("작업 완료"); // -생략 가능. 단, 예외가 발생하지 않아야 함이 전제되어야 하며 예외가 실제 발생  
6     }                            // -한다면 throws를 통해 예외 객체가 JVM에게 전달되어 프로그램을 강제 종료시켜  
7 }                                // -버림.즉, 이러한 방식은 실전에서는 지양되어야 하나 테스팅코드를 실행하기 위해  
8 }                                // -매번 예외 처리하는 번거로움을 해소하기 위한 임시 방편임에 유의.
```

(6) 메서드의 호출부에 대한 예외 처리

```
1 public class Main {  
2     public static void main(String[] args) {  
3         // try {  
4             call();  
5             System.out.println("다음 문장");  
6         }  
7         catch (Exception e) {  
8             e.printStackTrace();  
9         }  
10        System.out.println("메서드 호출");  
11    }  
12  
13    static void call() {  
14        throw new ArithmeticException();  
15    }  
16 }  
17  
18 |
```

// throw를 이용해 예외 객체를 생성하면 throws를 통해 예외 객체를
// -명시하지 않아도 예외 발생부(14행)와 호출부(4행)에서 예외가 -
// 검출되어 정상적인 처리 불가. 따라서 호출부 또는 예외 발생부 둘 중
// -하나를 try~catch블럭 내에 구성해야 함. 만약 당행에서 체크드 -
// 예외가 발생한다면 컴파일 자체가 불가.

(7) 메서드 내의 예외 발생부에 대한 예외 처리

```
1 public class Main {
2     public static void main(String[] args) {
3         call();
4         System.out.println("다음 문장");
5         System.out.println("메서드 호출");
6     }
7
8     static void call() {
9         try {
10             throw new ArithmeticException();           // 메서드 내에서 직접 예외 발생부에 대한 예외처리를 함으로써 메서드
11         } catch (ArithmetricException e) {          // -호출부에 대한 예외처리가 불필요해짐에 따라 메서드 호출부에 대한
12             e.printStackTrace();                   // -예외처리 시 실행 불가하였던 4행의 처리가 가능하게 됨.
13         }
14     }
15 }
```

(8) throws를 통한 예외 회피

```
1 public class Main {
2     public static void main(String[] args) {
3         //     try {
4         //         call();
5         //         System.out.println("다음 문장");
6         //     }
7         //     catch (Exception e) {
8         //         e.printStackTrace();
9         //     }
10        System.out.println("메서드 호출");
11    }
12
13     static void call() throws Exception{      // 메서드 내부에서 예외가 발생됨에 따라 예외 처리를 함에 있어
14         throw new ArithmetricException();      // -그 예외 발생 원인이 외부 요인에 의한 즉, 내부적 해결이 -
15     }                                         // 불가한 경우 이와 같이 throws를 통해 예외 처리를 회피함
16 }
```

// -으로써 호출부에서 직접 예외 처리를 강제함.

3) 사용자 정의 예외

```
1 /*  
2 * < 사용자 정의 예외 >  
3 *  
4 * 개발자의 실수나 외부 상황에 의해 발생할 수 있는 일반적 예외가 아닌, 프로그램 내부의 고유 논리에  
5 * 의해 발생할 수 있는 예외로 이는 JVM이 검출할 수 없으므로 개발자가 직접 예외를 정의.  
6 */  
7 import java.util.Scanner;  
8  
9 class UserDefException extends Exception { // 사용자 정의 예외.  
10    UserDefException(String exContent) {  
11        super(exContent); // 표준 예외 객체나 사용자 정의 예외 객체 생성 시 24행과 같이 생성자를 통해 문자열 전달이 가능. 이에 따라 49행과 같이 -  
12    } // -getMessage()를 이용 상세한 예외 내용을 직접 명시 가능. 단, 표준 예외 객체의 경우 예외 객체 생성 시 생성자의 //  
13 } // 실인수로 예외 내용을 전달만 하면 되지만 사용자 정의 예외 객체에 문자열을 전달하려면 상속 규칙에 의거 생성자는 상속되지 //  
14 // 않으므로 필히 10행과 같이 생성자를 정의하여 super키워드를 통해 문자열을 전달해야 함에 유의.  
15  
16 abstract class Sum { // 객체 생성이 목적이 아닌 내부 메서드만을 활용하기 위해 추상 클래스로 선언. 따라서 내부 메서드 또한 18행과 같이 정적으로 선언.  
17  
18    static void sum(int st, int ed) throws UserDefException { // throws 예외 객체, ..., ... : 지정한 메서드에 예외가 발생한다면 -  
19        int tot = 0; // 호출부로 예외 객체를 던져 예외가 발생 했음을 명시. 해당 호출부에서는 예외에 대한 -  
20  
21        if (st > ed) { // 처리를 책임져야 하므로 47행의 메서드 호출부는 try블럭 내에 구성되어함. 즉, -  
22            throw new UserDefException("입력 오류!! 작은 수가 먼저 입력되어 합니다.\n"); // throws키워드가 명시된 메서드를 호출할 때 그 예외가 체크드 예외라면 호출부에서 // 반드시 try블럭으로 구성해야함.  
23        }  
24  
25        for (int i = st; i <= ed; i++) {  
26            tot += i;  
27        }  
28  
29        System.out.printf("%d에서 %d까지의 합은 %d입니다.", st, ed, tot);  
30    }  
31 }  
32 }
```

```

34
35 public class Main {
36     public static void main(String[] args) {
37         Scanner input = new Scanner(System.in);
38         int st, ed;
39
40         for (;;) {
41             try {
42                 System.out.print("작은 수 입력 : ");
43                 st = input.nextInt();           // 입력 시 숫자가 아닌 문자가 입력되는 것을 방지하기 위해 예외 블럭 내 설정.
44                 System.out.print("큰 수 입력 : ");
45                 ed = input.nextInt();
46
47                 Sum.sum(st, ed);
48             } catch (UserDefException e) {
49                 System.out.print(e.getMessage());
50                 System.out.println("재입력 바랍니다.\n");
51                 continue;
52             } catch (Exception e) {
53                 System.out.println("숫자만 입력 가능합니다.\n");
54
55                 input.nextLine();           // String nextLine() : 문자열 입력 메서드. 단 공백, 탭, 개행문자까지 포함한 문자열 인식. nextLine메서드 -
56                 continue;                  // 이전에 입력 메서드가 존재하고 그 메서드가 개행 문자를 포함하지 않는다면 버퍼에 남아 있던 개행문자가 nextLine-
57             }                           // 메서드로 전이되어 입력이 무시되는 효과 발생. 43, 45행의 nextInt()는 개행문자를 포함하지 않는 숫자만 입력 받는 -
58             input.close();              // 메서드인데 문자를 입력받게 되면 InputMismatchException 예외가 발생되지만 예외 블럭 내 구성을 함으로써 루프를
59             break;                     // -돌려 재입력을 받도록 하면 되지만, 그 문자와 개행문자는 입력 버퍼에 그대로 남게 됨으로써 차 입력 시 버퍼의 값이-
60         }                           // 전달되어 입력 기회를 잃고 그 버퍼의 값은 또 전달되지 않는 무한 반복 발생. 이를 해결하기 위해 개행문자를 포함하는 -
61     }                           // nextLine()을 통해 버퍼의 개행문자를 비워야 함.
62 }
63 }
```

4) 자원(Resource) 해제

(1) finally를 이용한 자원 해제

```
1 import java.util.Scanner;
2
3 public class Main {
4     public static void main(String[] args) {
5         Scanner sc = new Scanner(System.in);
6         String str=null;
7
8
9         try {
10             System.out.print("문자열 입력 : ");
11             str = sc.nextLine();
12         } catch (Exception e) {
13             e.printStackTrace();
14         } finally {           // 네트워크, DB접속등의 비메모리 관련 외부 자원에 대한 리소스 해제.
15             sc.close();        // -단, 이 때 finally블럭은 예외 발생 여부와 상관없이 처리됨.
16         }
17
18         System.out.println(str);
19
20     } // str = sc.nextLine();           // 예외 발생 여부와 관계없이 14행의 finally블럭에 의해 기존 Scanner객체의
21     // -입력 스트림이 이미 해제된 상태이므로 불법적인 접근으로 처리.
22 }
```

(2) 예외 발생과 finally

```
1 import java.util.Scanner;
2
3 public class Main {
4     public static void main(String[] args) {
5         Scanner sc = new Scanner(System.in);
6         int num=0;
7
8         try {
9             System.out.print("숫자 입력 : ");
10            num = sc.nextInt();
11        } catch (Exception e) {
12            System.out.println("예외 발생");
13        } finally {                                // 예외 발생 여부와 상관없이 처리.
14            System.out.println("스캐너 자원 해제");
15            sc.close();
16        }
17
18        System.out.println(num);
19    }
20 }
```

(3) finally 활용(비밀번호 입력)

<1> finally 미적용

```
1 import java.util.Scanner;
2
3 public class Main {
4     static final int correctpw = 1234;           // 저장되어있는 비밀번호로 가정.
5     static int pw=0;                            // 입력 비밀번호.
6     static int fMth=0;                          // inputPw메서드에 대한 최초 호출 명시 스위치.
7
8     public static void main(String[] args) {
9         Scanner sc = new Scanner(System.in);
10
11         inputPw(sc);      // 비밀번호 입력을 위한 일회성 메서드. 메서드 재 호출 시 스캐너 객체를 닫아 호출 불가 상태이어야 함으로 설계.
12             // -즉, 비밀번호가 일치하던 불일치하던 3회 입력 후에는 스캐너 객체를 close하여 15행의 메서드 호출이 -
13             // 불가하도록 설계.
14
15         inputPw(sc);      // 비밀번호 3회 불일치 시 재입력 시도 가정.
16     }
17 }
```

```

18 static void inputPw(Scanner sc) {
19     for (int i = 0; i < 3; i++) {
20         try {
21
22             if(fMth == 0) {                                // inputPw메서드 최초 호출 시에만
23                 System.out.print("패스워드 4자리 숫자 입력 : ");    // -23행의 문자열이 출력되도록 설정.
24             }
25
26             pw = sc.nextInt();
27
28             if ( pw == correctpw) {
29                 System.out.println("패스워드 일치 로그인 성공");
30
31                 fMth=1;          // inputPw메서드에 대한 최초 호출이 아님 명시 스위치.
32
33                 sc.close();      // 비밀번호가 일치하던 불일치 하던 당행과 51행에서 각각 스캐너 객체를 닫아야함.
34                 return;
35             }
36
37         } catch (Exception e) {
38             sc.nextLine();                                // 26행에서 문자 입력 시 예외 발생으로 인한 입력버퍼 개행문자
39             System.out.println(i+1+"회 입력 오류");        // -제거. 또한 비밀번호 일치 여부를 떠나 15행의 메서드 재-
40             System.out.println("숫자만 입력 가능합니다.");   // 호출 시 이미 소멸된 스캐너 객체를 26행에서 불접적 접근을-
41             continue;                                     // 하게됨으로써 예외가 발생되어 catch블럭에 의해 처리가 되나
42         }                                              // -당행에서 다시 예외가 발생되어 강제 종료.
43
44         System.out.println(i+1+"회 입력 오류");
45     }
46
47     System.out.println("비밀번호 3회 오류 접속 불가");
48
49     fMth=1;          // inputPw메서드에 대한 최초 호출이 아님 명시 스위치.
50
51     sc.close();      // 비밀번호 3회 입력 오류 시 스캐너 객체 해제.
52 }
53 }
```

<2> finally 적용

```
1 import java.util.Scanner;
2
3 public class Main {
4     static final int correctpw = 1234;           // 저장되어있는 비밀번호로 가정.
5     static int pw=0;                           // 입력 비밀번호.
6     static int fMth=0;                         // inputPw메서드에 대한 최초 호출 명시 스위치.
7
8     public static void main(String[] args) {
9         Scanner sc = new Scanner(System.in);
10
11         inputPw(sc);
12         inputPw(sc);
13     }
14 }
```

```

15 static void inputPw(Scanner sc) {
16     try {                                // try ~ finally : 예외 처리를 위한 목적이 아닌 공통 자원요소에 대한 해제를 목적.
17         for (int i = 0; i < 3; i++) {      // -21행부터의 try ~ catch블럭에 finally블럭을 설정하여 스캐너 객체 해제를 하면
18             // -비밀번호 오입력 또는 문자 입력으로 인한 루프의 최초 회전이 종료되는 시점에 해제를 -
19             // 함으로써 차회전부터는 스캐너 객체를 사용할 수 없게 되는 문제가 발생하므로 이와같이 -
20             // 외부에 try ~ finally블럭을 설정하여 30행의 리턴 시 또는 루프 종료 후 최초 한 -
21         try {                            // 번만 해제가 되도록 설정.
22             if(fMth == 0) {
23                 System.out.print("패스워드 4자리 숫자 입력 : ");
24             }
25
26             pw = sc.nextInt();
27
28             if ( pw == correctpw) {
29                 System.out.println("패스워드 일치 로그인 성공");
30                 return;
31             }
32
33         } catch (Exception e) {
34             sc.nextLine();
35             System.out.println(i+1+"회 입력 오류");
36             System.out.println("숫자만 입력 가능합니다.");
37             continue;
38         }
39
40         System.out.println(i+1+"회 입력 오류");
41     }
42
43     System.out.println("비밀번호 3회 오류 접속 불가");
44
45 } finally {          // finally블럭은 예외 발생 여부와 관계없이 실행됨은 물론,
46                     // -30행에서 메서드에 대한 리턴이 됨에도 무조건 실행.
47
48     fMth=1;           // 스위치 설정과 스캐너 객체에 대한 해제는 비밀번호의 일치 여부를 떠나 반드시
49     sc.close();        // -설정되어야 하는 공통 코드이므로 이와 같이 finally블럭에 설정.
50 }
51 }
52 }

```

<3> 메서드 재 호출 예외 처리

```
8@  public static void main(String[] args) {
9      Scanner sc = new Scanner(System.in);
10
11     inputPw(sc);
12
13     try {                                // 비밀번호가 일치하던 불일치 하던 스캐너 객체가 닫혀 메서드
14         inputPw(sc);                      // -재 호출 시 예외 발생을 방어하기 위한 예외 처리.
15     } catch (Exception e) {
16         if (pw != correctpw) {
17             System.out.println("\n\n비밀번호 3회 입력 오류입니다. 가까운 은행을 방문해주세요 ㅋㅋ");
18         }
19     }
20 }
```

1) 스레드(Thread)의 개념과 생성

(1) Thread 상속

```

18 /*
2  * < 프로세스(Process)와 멀티스레드(Multi-thread) >
3 *
4 * 프로세스란 하나의 프로그램이 CPU에 할당되어 실행중인 상태를 의미하며 그 프로세스를 다시 여러 가닥으로 조각 즉,
5 * 하나의 실행 흐름을 여러 개로 분할하여 각각의 독립적인 실행 흐름을 가지도록 한 것이 스레드임. 그렇다고 해서 실제
6 * 개별적인 흐름을 가지는 것은 아니며 프로세스가 CPU로부터 할당받은 시간을 공유하고 그 시간을 아주 잘게 퍼개어 각
7 * 스레드에 할당함으로써 마치 각각의 독립적인 스레드가 별도의 실행흐름을 가지는 것처럼 기만하는 것임.
8 * 스레드는 클래스 단위로 생성되며 Thread클래스로부터 직접 상속을 받아 스레드 객체를 생성하는 방법과 Runnable
9 * 인터페이스에 대한 구현 클래스 객체를 생성하여 Thread객체의 생성자 인수로 전달하여 생성하는 2가지 형태로 분류.
10 */
11 class TestThread extends Thread { // 사용자(작업) 스레드.
12
13     public void run() { // 스레드 실행을 위한 run()메서드 재정의. 메인 클래스(메인 스레드)의 -
14         for (int i = 0; i < 10; i++) { // 메인 메서드가 프로그램의 시작지점 이듯이 run()메서드가 사용자 스레드의
15             System.out.println("바브"); // -시작지점. 따라서 run()메서드 내에서 return되면 사용자 스레드 또한
16             try { // 종료.
17                 Thread.sleep(400); // sleep(long millis) : Thread 클래스의 정적 메서드로써 지정된-
18             } catch (Exception e) { // 인수(1000분의 1초)의 시간만큼 해당 스레드를 대기시킴. 즉, CPU의 -
19             } // 큐에 할당하지 않고 자연시간 만큼 대기 시킴. sleep의 대상은 항상 현재 -
20         } // 스레드를 의미하므로 정적이며 따라서 객체에 대한 적용이 불가하여 sleep을
21     } // 통한 외부 스레드의 제어 불가. 주의할 것은 인수에 지정된 대기 시간은 그 -
22 } // 시간만큼 대기를 해달라는 요청에 불과할 뿐 실제 시스템의 성능이나 상황에 -
23 } // 따라 더 자연될 수 있음.
24

```

```
25
26 class Main { // 메인 스레드.
27
28 public static void main(String[] args) {
29     TestThread testThread = new TestThread(); // 사용자 스레드 객체 생성.
30
31     testThread.start(); // start()메서드를 통한 생성된 스레드의 실행. 주의할 점은 start()메서드를
32 // -호출함으로써 생성된 스레드가 바로 실행되는 것은 아니며, 호출 직후 CPU의 -
33     for (int i = 0; i < 10; i++) { // 큐에 실행 대기 상태를 유지시키다가 실행 할당이 되는 순간 스레드 객체의 run()
34         System.out.print("박성연 "); // -메서드를 호출하여 실제 스레드의 실행이 진행.
35         try {
36             Thread.sleep(200);
37         } catch (Exception e) { }
38     }
39 }
40 }
```

(2) 익명이너클래스를 활용한 스레드 생성

```
1 class Main {
2     public static void main(String[] args) {
3         Thread testThread=new Thread() { // 하나의 사용자 스레드를 생성하기 위해 별도의 클래스를 정의하여-
4             public void run() { // 객체를 생성할 필요없이 담행과 같이 익명이너클래스 형태로 정의.
5                 for (int i = 0; i < 10; i++) {
6                     System.out.println("바보");
7                     try {
8                         Thread.sleep(400);
9                     } catch (Exception e) { }
10                }
11            }
12        };
13        testThread.start();
14
15        for (int i = 0; i < 10; i++) {
16            System.out.print("박성연 ");
17            try {
18                Thread.sleep(200);
19            } catch (Exception e) { }
20        }
21    }
22 }
23 }
24 }
```

(3) Runnable 인터페이스

```
1 class TestThread implements Runnable {           // Runnable인터페이스로 부터 구현한 클래스 타입의 객체를-
2     public void run() {                         // 14행과 같이 Thread 객체의 생성자 인수로 전달하여 스레드
3         for (int i = 0; i < 10; i++) {          // -객체를 기동. 이러한 방식을 굳이 지원하는 이유는 자바의 -
4             System.out.println("바보");        // -특성상 다중 상속을 지원하지 않기 때문. 즉, 어떤 클래스의
5             try {                            // -메서드를 활용하여 별도의 독립적인 스레드 기동을 구현하고자
6                 Thread.sleep(400);           // -한다면, 메서드가 정의되어 있는 클래스로부터 상속을 받아야-
7             } catch (Exception e) { }       // 하고 또한 Thread클래스로부터도 상속을 받아야 하는 다중-
8         }                                     // 상속이 되어야 하는 문제점이 발생하므로 이를 극복하기 위한 방식.
9     }
10 }
11
12 class Main {
13     public static void main(String[] args) {
14         Thread testThread = new Thread(new TestThread());
15
16         testThread.start();
17
18         for (int i = 0; i < 10; i++) {
19             System.out.print("박성연 ");
20             try {
21                 Thread.sleep(200);
22             } catch (Exception e) { }
23         }
24     }
25 }
```

(4) 익명이너클래스를 활용한 Runnable 구현

```
1 class Main {
2     public static void main(String[] args) {
3         String str="바보";
4
5         Runnable runnable = new Runnable() {
6             @Override
7             public void run() {
8                 for (int i = 0; i < 10; i++) {
9                     System.out.println(str);
10                    try {
11                         Thread.sleep(400);
12                     } catch (Exception e) { }
13                 }
14             }
15         };
16         Thread testThread = new Thread(runnable);
17
18 //         Thread testThread = new Thread(new Runnable() { // Thread객체 생성자의 인수에 5행의 new연산자부터
19 //             @Override
20 //             public void run() {
21 //                 for (int i = 0; i < 10; i++) {
22 //                     System.out.println(str);
23 //                     try {
24 //                         Thread.sleep(400);
25 //                     } catch (Exception e) { }
26 //                 }
27 //             }
28 //         });
29 //
30         testThread.start();
31
32         for (int i = 0; i < 10; i++) {
33             System.out.print("박성연 ");
34             try {
35                 Thread.sleep(200);
36             } catch (Exception e) { }
37         }
38     }
39 }
```

// 익명이너클래스 내부에 필드 설정이 가능하지만 9, 22행의 공통 -
// 요소 출력을 위해 내부 필드를 별도로 중복 설정하는 것을 회피하기 -
// 위해 외부에 선언한 것일뿐 의미없음. 중요한 것은 익명이너클래스 -
// 또한 내부에 필드 설정이 가능하다는 것인데 생성자 설정은 불가함에-
// 유의. 생성자는 해당 클래스의 이름과 동일해야 하는데 익명이너 -
// 클래스는 맙그대로 이름이 없는 클래스이기 때문에 생성자 설정이 불가.
// 따라서 익명이너클래스는 내부에 필드 설정은 가능하지만 생성자를 통한
// -내부 필드의 초기화는 불가.

// Thread객체 생성자의 인수에 5행의 new연산자부터 -
// 세미콜론만 뺀 15행까지를 그대로 넣은 형태. 이러한-
// 익명이너클래스 문법을 활용함으로써 단순히 임의의 작업
// -스레드를 하나 생성하기 위해 굳이 클래스를 별도로 -
// 정의할 필요가 없어짐과 더불어 주 클래스의 멤버를 직접
// -참조 가능. Thread클래스로 직접 익명이너클래스를
// -생성해도 무방하나 이 경우, 직접 run()메서드를 -
// 재정의해야 하는 반면 Runnable인터페이스로부터 -
// 생성을 하면 자동 유포라이팅됨. 임의의 슈퍼클래스로 -
// 상속을 받아 Runnable인터페이스를 통해 스레드를 -
// 구현해야 하는 상황이 아니더라도 이처럼 Runnable -
// 인터페이스를 익명이너클래스로 활용함으로써의 장점 존재.

(5) Runnable 활용

```
1 class SuperClass{  
2     public void specialMethod() { // 어떤 클래스의 아주 특별하고 잘 정의된 메서드로 가정.  
3         for (int i = 0; i < 25; i++) {  
4             System.out.println("Run");  
5             try {  
6                 Thread.sleep(200);  
7             } catch (InterruptedException e) {}  
8         }  
9     }  
10 }  
11  
12 class MyThread extends SuperClass implements Runnable{ // 다중상속을 지원하지 않는 자바의 특성상 어떤 임의-  
13     @Override // 클래스의 잘 정의된 메서드를 스레드로 구현하고자 할  
14     public void run() { // 때, 임의의 클래스와 Thread클래스로부터 다중-  
15         specialMethod(); // 상속을 받아야 하는 문제점 발생. 이를 극복하기 -  
16     } // 위해 이와같이 필요한 클래스로 부터 상속을 받되 -  
17 } // Runnable인터페이스로 부터 구현을 하고 run()-  
18 // 메서드를 재정의하여 그 내부에서 상속받은 메서드를-  
19 class Main { // 사용함으로써 해결.  
20     public static void main(String[] args) {  
21         Thread testThread = new Thread(new MyThread());  
22  
23         testThread.start();  
24  
25         for (int i = 0; i < 5; i++) {  
26             System.out.println("Main Thread Run");  
27             try {  
28                 Thread.sleep(1000);  
29             } catch (InterruptedException e) {}  
30         }  
31     }  
32 }
```

2) 스레드 관리

(1) 스레드 정보 관리

```
1 class Thread1 extends Thread{  
2     public Thread1() { }           // 19행의 인수 없는 객체 생성을 위해 반드시 디폴트 생성자 지정.  
3  
4     public Thread1(String name) {    // 39행의 생성자 인수를 super()의 생성자 인수로 전달하여 객체명 지정.  
5         super(name);  
6     }  
7 }  
8  
9 class Thread2 extends Thread{  
10    public Thread2() { }  
11  
12    public Thread2(String name) {  
13        super(name);  
14    }  
15 }  
16 }
```

```
17 class Main {  
18     public static void main(String[] args) {  
19         Thread1 thread1 = new Thread1();  
20         Thread2 thread2 = new Thread2();  
21  
22         System.out.println(thread1.getName());           // getName() : 스레드 객체의 이름 조사. 별도의 지정이 없는 경우 생성된 순서대로  
23         System.out.println(thread2.getName());           // -Thread-n...의 형태로 지정.  
24  
25         System.out.println(Thread.currentThread().getName());    // currentThread() : Thread 클래스의 정적 메서드로 현재 -  
26         System.out.println();                            // 스레드 객체를 리턴. 메인 스레드의 경우 별도의 객체 생성을 안하므로  
27                                         // -이와같이 currentThread() 메서드를 통해 스레드 객체 관련 -  
28                                         // 메서드 적용.  
29  
30         thread1.setName("스레드1");                  // setName(String name) : 스레드 객체의 이름 지정.  
31         thread2.setName("스레드2");  
32         Thread.currentThread().setName("메인 스레드");  
33         System.out.println(thread1.getName());  
34         System.out.println(thread2.getName());  
35         System.out.println(Thread.currentThread().getName());  
36         System.out.println();  
37  
38  
39         Thread1 thread1_2 = new Thread1("스레드1-2");    // 생성자를 통한 스레드 객체명 지정.  
40         Thread2 thread2_2 = new Thread2("스레드2-2");  
41  
42         System.out.println(thread1_2.getName());  
43         System.out.println(thread2_2.getName());  
44     }  
45 }
```

(2) 스레드의 우선순위 조사

```
1  /*
2   * < 스레드의 우선순위 >
3   *
4   * 스레드의 우선순위는 시스템 스케줄링에 의한 CPU로부터의 할당받는 시간 점유율을 지정하는 Thread의 속성(field).
5   * 하지만 스레드의 우선순위는 절대적이지는 않으며 시스템 스케줄링 상황에 따라 달라질 수 있음. 개별 스레드의 우선순위는
6   * 1 ~ 10까지 설정이 가능하며 10이 가능 높은 우선순위를 가짐. Thread클래스의 우선순위 속성에 대응되는
7   * MIN_PRIORITY(1), NORM_PRIORITY(5), MAX_PRIORITY(10)의 세가지 정적 상수 필드가 제공되며
8   * Thread클래스에 소속됨. 우선순위 미 지정 시 디폴트 5로 설정.
9  */
10 class Thread0 extends Thread{
11     public void run() {
12         for (int i = 0; i < 5; i++) {
13             try {
14                 Thread.sleep(100);
15             } catch (InterruptedException e) {}
16
17             System.out.println(Thread.currentThread().getName());
18         }
19     }
20 }
21
22 class Thread1 extends Thread{
23     public void run() {
24         for (int i = 0; i < 5; i++) {
25             try {
26                 Thread.sleep(100);
27             } catch (InterruptedException e) {}
28
29             System.out.println(Thread.currentThread().getName());
30         }
31     }
32 }
```

```
34 class Main {  
35     public static void main(String[] args) {  
36         Thread0 thread0 = new Thread0();  
37         Thread1 thread1 = new Thread1();  
38  
39         System.out.println(Thread.currentThread().getPriority());  
40         System.out.println(thread0.getPriority());  
41         System.out.println(thread1.getPriority());      // getPriority() : 스레드 객체의 우선 순위 조사.  
42         System.out.println();  
43  
44         thread0.start();  
45         thread1.start();  
46  
47         for (int i = 0; i < 5; i++) {  
48             try {  
49                 Thread.sleep(100);  
50             } catch (InterruptedException e) {}  
51  
52             System.out.println(Thread.currentThread().getName());  
53         }  
54     }  
55 }
```

(3) 스레드의 우선순위 지정

```
1 class Thread0 extends Thread{  
2     public void run() {  
3         for (int i = 0; i < 5; i++) {  
4             try {  
5                 Thread.sleep(100);  
6             } catch (InterruptedException e) {}  
7             System.out.println(Thread.currentThread().getName());  
8         }  
9     }  
10 }  
11 }  
12  
13 class Thread1 extends Thread{  
14     public void run() {  
15         for (int i = 0; i < 5; i++) {  
16             try {  
17                 Thread.sleep(100);  
18             } catch (InterruptedException e) {}  
19             System.out.println(Thread.currentThread().getName());  
20         }  
21     }  
22 }  
23 }  
24 }
```

```
25 class Main {  
26     public static void main(String[] args) {  
27         Thread0 thread0 = new Thread0();  
28         Thread1 thread1 = new Thread1();  
29  
30         Thread.currentThread().setPriority(Thread.MIN_PRIORITY);  
31         thread0.setPriority(Thread.NORM_PRIORITY);  
32         thread1.setPriority(Thread.MAX_PRIORITY); // setPriority(int newPriority) : 스레드 객체에 대한 우선 순위  
33                                         // -지정. 스레드 실행 전 먼저 변경하는 것이 원칙. 여기서는 실습 환경상 -  
34         thread0.start();  
35         thread1.start();  
36  
37         for (int i = 0; i < 5; i++) {  
38             try {  
39                 Thread.sleep(100);  
40             } catch (InterruptedException e) {}  
41  
42             System.out.println(Thread.currentThread().getName());  
43         }  
44     }  
45 }  
46 }
```

3) 스레드 제어

(1) yield

```
1 import java.util.ArrayList;
2
3 class Insert {
4     public void insertTest() {
5         ArrayList<Character> str = new ArrayList<Character>();
6
7         long st = System.currentTimeMillis();
8         for (int i = 0; i < 10000; i++) {
9             for (char j = 'a'; j <= 'z'; j++) {
10                 str.add(0, j);
11             }
12         }
13         long ed = System.currentTimeMillis();
14
15         System.out.println(Thread.currentThread().getName() + " : " + (ed - st) / 1000.);
16     }
17 }
18
19 class Insert1 extends Insert implements Runnable {
20     @Override
21     public void run() {
22         insertTest();
23     }
24 }
25
```

```
26 class Main {  
27     public static void main(String[] args) {  
28         Thread insert1 = new Thread(new Insert1(), "Insert1"); // Thread( Runnable객체, 스레드명 ) :  
29                                         // -Thread클래스의 오버로딩 생성자.  
30         insert1.start();  
31         Thread.currentThread().setName("Insert0");  
32  
33  
34         ArrayList<Character> str = new ArrayList<Character>();  
35  
36         long st = System.currentTimeMillis();  
37         for (int i = 0; i < 10000; i++) {  
38             for (char j = 'a'; j <= 'z'; j++) {  
39                 Thread.yield();  
40                 str.add(0, j);  
41             }  
42         }  
43         long ed = System.currentTimeMillis();  
44         System.out.println(Thread.currentThread().getName() + " : " + (ed - st) / 1000.);  
45     }  
46 }  
47 }
```

(2) join

<1> 개별 스레드의 실행 순서 제어

```
1① /*
2  *  < 스레드 대기 - join >
3  *
4  *  - sleep은 실행 주체가 되는 현재 스레드의 대기를 지정하는 정적 메서드인 반면, join은 실행 주체가 되는 현재 스레드가 아닌 상대적
5  *  다른 스레드의 대기를 지정하는 비정적 메서드.
6  *
7  *  - join([long millis]) : 임의 스레드 객체에 대한 join메서드 호출 시점 이후에 실행(start)되는 모든 스레드의
8  *  실행을 join의 호출 객체 스레드의 실행이 완료될 때까지 대기. 따라서 join메서드 호출 시점 이전에 실행(start)된 스레드에
9  *  대해서는 적용되지 않으나, join메서드의 호출 배경이 되는 주 스레드는 이미 실행 중임에도 예외가 적용되어 join메서드 호출 시점
10 *  이후의 명령 처리에 대해서만 대기. main스레드가 아닌 외부 스레드 환경에서 join메서드의 호출 주체가 되는 외부 스레드의 실행
11 *  또한 실행 전의 join호출은 실효하지 않음. 인수를 생략하거나 0을 전달하면 호출 스레드의 종료 시점까지 무한대기를 하며 인수로
12 *  시간을 전달하면 지정된 시간만큼 대기를 하되 호출 스레드의 종료시점이 지정된 대기시간보다 먼저 종료되면 지정된 대기시간을 무시하고
13 *  대기중이던 스레드를 실행. 스레드의 대기 중 인터럽트가 걸릴 수 있어 예외 처리는 필수.
14 */
15 class Th1 extends Thread{
16②     public void run() {
17         for (int i = 0; i < 10; i++) {
18             System.out.println("Th1 : " + i);
19             try {Thread.sleep(300);} catch (InterruptedException e) {}
20         }
21         System.out.println("<<<t1완료>>>");
22         System.out.println();
23     }
24 }
25 }
```

```

26 class Th2 extends Thread{
27     public void run() {
28         for (int i = 0; i < 10; i++) {
29             System.out.println("Th2 : " + i);
30             try {Thread.sleep(300);} catch (InterruptedException e) {}
31         }
32         System.out.println("<<<t2완료>>"); 
33         System.out.println();
34     }
35 }
36
37 class Main {
38     public static void main(String[] args) {
39         Th1 th1 = new Th1();
40         Th2 th2 = new Th2();
41
42         th1.start();
43         try {
44             th1.join(100000); // join의 호출 배경이 되는 주 스레드의 경우 join호출 시점 -
45         } catch (InterruptedException e) {} // 이후 부터의 모든 명령 처리가 대기되고 th2의 실행 또한 -
46                                         // join호출 시점 이후이므로 th1의 실행 완료 후 th2 실행.
47         th2.start();
48         try {
49             th2.join(); // join()
50         } catch (InterruptedException e) {} 
51
52         for (int i = 0; i < 10; i++) {
53             System.out.println("메인스레드 : " + i);
54             try {Thread.sleep(300);} catch (InterruptedException e) {}
55         }
56         System.out.println("<<<메인스레드 완료>>"); 
57     }
58 }

```

<2> 주 스레드에 대한 join

```
37 class Main {  
38     public static void main(String[] args) {  
39         Th1 th1 = new Th1();  
40         Th2 th2 = new Th2();  
41  
42         try {  
43             Thread.currentThread().join();          // join에 대한 종료 시점은 호출 스레드에 대한 실행 완료 후가 되므로 이와  
44         } catch (InterruptedException e) { }      // -같이 주 스레드에 대한 join은 호출 시점 이후가 되는 46행부터의 모든-  
45                                         // 처리가 대기가 되어 주 스레드의 종료가 될 수 없는 무한대기 상태가 됨.  
46         th1.start();  
47         try {  
48             th1.join();  
49         } catch (InterruptedException e) { }  
50  
51         th2.start();  
52         try {  
53             th2.join();  
54         } catch (InterruptedException e) { }  
55  
56         for (int i = 0; i < 10; i++) {  
57             System.out.println("메인스레드 : " + i);  
58             try {Thread.sleep(300);} catch (InterruptedException e) {}  
59         }  
60         System.out.println("<<<메인스레드 완료>>>");  
61     }  
62 }
```

(3) 데몬(Daemon) 스레드

<1> 데몬 스레드 지정(setDaemon)

```
1Theta /*
2  *  < 프로그램 종료 방식에 따른 스레드 분류 >
3  *
4  *  * main스레드 단독으로 실행되는 단일 스레드인 경우 main스레드의 종료가 전체 프로그램의 종료를 의미하지만
5  *  멀티스레드 환경에서는 모든 스레드가 종료되어야만 전체 프로그램이 종료.
6  *
7  *  - 사용자 스레드(User Thread) : main스레드의 실행 종료와 관계 없이 독립적으로 실행되는 스레드.
8  *                                즉, main스레드의 종료와 관계 없이 모든 사용자 스레드가 종료
9  *                                되어야만 전체 프로그램이 종료됨.
10 *
11 *  - 데몬 스레드(Daemon Thread) : 모든 사용자 스레드에 종속되어 실행이 결정되는 스레드로 모든 사용자
12 *                                스레드가 종료되면 실행중이더라도 강제 종료. main스레드는 기본적
13 *                                으로 사용자 스레드로 설정됨.
14 *
15 *  * 주 스레드에서 생성되는 종속 스레드는 별다른 지정이 없으면 주 스레드의 속성을 그대로 상속 받음. 즉, 주
16 *  스레드가 사용자 스레드면 종속 스레드도 사용자 스레드로 지정이되고 주 스레드가 데몬 스레드이면 종속 스레드
17 *  또한 데몬 스레드로 설정.
18 */
19 class Th extends Thread{
20Theta     public void run() {
21         for (int i = 0; i < 10; i++) {
22             System.out.println("Th : " + i);
23             try {Thread.sleep(500);} catch (InterruptedException e) {}
24         }
25         System.out.println("<<<Th완료>>>");
26         System.out.println();
27     }
28 }
29 }
```

```
30 class Main {
31     public static void main(String[] args) {
32         Th th = new Th();
33
34         th.setDaemon(true); // setDaemon(boolean on) : 스레드에 대한 데몬 스레드 -
35         th.start();          // 지정이 가능한 메서드로 실인수 on이 true면 데몬 스레드로 지정이 -
36                     // 되고 false면 사용자 스레드로 지정. 단, 반드시 스레드 실행전 -
37                     // 지정되어야 함. 현재 실행중인 주 스레드에 대한 메서드 호출 시 -
38                     // 컴파일 에러가 발생하지는 않으나 막상 실행 시 예외가 발생되므로 -
39                     // 실질적으로 주 스레드에서 생성되는 종속 스레드에 대한 지정만 가능.
40
41     for (int i = 0; i < 10; i++) {
42         System.out.println("메인스레드 : " + i);
43         try {Thread.sleep(100);} catch (InterruptedException e) {}
44     }
45     System.out.println("<<<메인스레드 완료>>>");
46 }
47 }
```

<2> 데몬 스레드 조사(isDaemon)

```
1 class Th extends Thread{  
2     public void run() {  
3         for (int i = 0; i < 10; i++) {  
4             System.out.println("Th : " + i);  
5             try {Thread.sleep(500);} catch (InterruptedException e) {}  
6         }  
7         System.out.println("<<<Th완료>>>");  
8         System.out.println();  
9     }  
10 }  
11  
12 class Main {  
13     public static void main(String[] args) {  
14         Th th = new Th();  
15  
16         /* isDaemon() : 해당 스레드가 데몬 스레드이면 true, 사용자 스레드이면 false 리턴. */  
17  
18         System.out.println( "메인 스레드 : " + ( Thread.currentThread().isDaemon() ? "데몬 스레드":"사용자 스레드" ) );  
19         System.out.println( "th 스레드 : " + ( th.isDaemon() ? "데몬 스레드":"사용자 스레드" ) );  
20  
21         th.setDaemon(true);  
22         System.out.println( "th 스레드 : " + ( th.isDaemon() ? "데몬 스레드":"사용자 스레드" ) );  
23  
24         th.start();  
25  
26         for (int i = 0; i < 10; i++) {  
27             System.out.println("메인스레드 : " + i);  
28             try {Thread.sleep(100);} catch (InterruptedException e) {}  
29         }  
30         System.out.println("<<<메인스레드 완료>>>");  
31     }  
32 }
```

<3> 스레드의 속성 상속

```
1 class Th1 extends Thread{
2     public void run() {
3         for (int i = 0; i < 10; i++) {
4             System.out.println("Th1 : " + i);
5             try {Thread.sleep(1000);} catch (InterruptedException e) {}
6         }
7         System.out.println("<<<Th1완료>>>");
8         System.out.println();
9     }
10 }
11
12 class Th2 extends Thread{
13     public void run() {
14         Th1 th1 = new Th1();
15
16         th1.start();      // main스레드에서 생성된 th2가 33행에서 데온 스레드로 설정됨에 따라 th1 또한 -
17                     // 주 스레드 th2의 속성을 상속받아 별도의 지정이 없지만 데온 스레드로 자동 설정됨.
18
19         for (int i = 0; i < 10; i++) {
20             System.out.println("Th2 : " + i);
21             try {Thread.sleep(500);} catch (InterruptedException e) {}
22         }
23         System.out.println("<<<Th2완료>>>");
24         System.out.println();
25     }
26 }
27
```

```
28 class Main {  
29     public static void main(String[] args) {  
30         Th2 th2 = new Th2();  
31  
32         th2.setDaemon(true);  
33         th2.start();  
34  
35         for (int i = 0; i < 10; i++) {  
36             System.out.println("메인스레드 : " + i);  
37             try {Thread.sleep(100);} catch (InterruptedException e) {}  
38         }  
39         System.out.println("<<<메인스레드 완료>>>");  
40     }  
41 }
```

<4> 데몬 스레드 지정을 통한 스레드 제어

```
1 class Th1 extends Thread{  
2     public void run() {  
3         for (int i = 0; i < 10; i++) {  
4             System.out.println("Th1 : " + i);  
5             try {Thread.sleep(500);} catch (InterruptedException e) {}  
6         }  
7         System.out.println("<<<Th1완료>>>");  
8         System.out.println();  
9     }  
10 }  
11  
12 class Th2 extends Thread{  
13     public void run() {  
14         Th1 th1 = new Th1();  
15  
16         th1.setDaemon(false);  
17         th1.start();  
18  
19         for (int i = 0; i < 10; i++) {  
20             System.out.println("Th2 : " + i);  
21             try {Thread.sleep(1000);} catch (InterruptedException e) {}  
22         }  
23         System.out.println("<<<Th2완료>>>");  
24         System.out.println();  
25     }  
26 }  
27
```

```
28 class Main {  
29     public static void main(String[] args) {  
30         Th2 th2 = new Th2();  
31  
32         th2.setDaemon(true);      // th2는 데몬 스레드로 설정했지만 16행의 지정으로 인해 메인 스레드와 th1스레드 를 다  
33         th2.start();           // -사용자 스레드가 되어 th2가 실행중인 경우 두 스레드가 모두 종료되면 th2도 강제 종료.  
34  
35         for (int i = 0; i < 10; i++) {  
36             System.out.println("메인스레드 : " + i);  
37             try {Thread.sleep(100);} catch (InterruptedException e) {}  
38         }  
39         System.out.println("<<<메인스레드 완료>>>");  
40     }  
41 }
```

4) 동기화(Synchronized)

(1) 공유 객체와 동기화

```
1① /*
2  * 주제 : < 공유 객체 소속의 배열 ar에 대한 개별 스레드의 데이터 교환 저장 >
3  *
4  * 스레드의 실행 순서나 실행 시간은 시스템 상황이나 스케줄링 환경에 종속적이므로 예측 불가하고 그 통제 또한 용이하지 않음. 위 주제에
5  * 맞추어 아래와 같이 InputOne, InputTwo 스레드를 실행시킨다 할 때, 두 스레드 중 스케줄러에 의해 먼저 할당된 스레드가 매우
6  * 빠른 속도로 먼저 배열 ar에 데이터를 저장함으로써 다른 스레드가 실행 할당을 받아 접근을 시도하기도 전에 프로그램이 종료될 것임에
7  * 따라 위 주제에 부합되지 않음. 그러나 공유 객체의 배열 크기를 더 크게 늘린다면 스레드가 할당을 받아 실행되는 시간이 무한정
8  * 지속되지는 않음으로 어느 정도의 시간이 지난후에는 각 스레드의 데이터가 규칙없이 뒤엉켜 저장되는 것을 볼 수 있을 것임. 위 주제에
9  * 맞추어 스레드를 번갈아 가면서 배열에 저장하려면 공유 객체에 대한 개별 스레드의 상대적 점유를 인정하고 적절한 순서 제어가 필요. 즉,
10 * 임의 스레드 하나가 공유 객체를 점유하면 다른 스레드들의 접근을 막고 선점한 스레드의 점유가 끝나면 다른 스레드에게 점유를 양도하는
11 * 과정을 반복함으로써 위 주제를 만족시킬 수 있는데 이러한 동작을 동기화(Synchronized)라 함.
12 */
13 class Data { // 스레드 InputOne, InputTwo의 공유 객체.
14
15     int[] ar; // 각 스레드의 데이터를 번갈아가면서 저장할 공유 배열.
16     int idx; // 각 스레드가 공유 배열 ar에 접근하기 위한 공유 인덱스.
17
18②     public Data(int size) {
19         ar = new int[size];
20         idx = -1;
21     }
22
23③     public void outputData() {
24         for (int i = 0; i < ar.length; i++) {
25             System.out.println(ar[i]);
26         }
27     }
28 }
29
```

```
30 class InputOne extends Thread {  
31     Data data;  
32  
33     public InputOne(Data data) {  
34         this.data = data;           // 개별 스레드가 공유 객체에 접근하기 위해 스레드의 생성자 인수로 공유 객체 전달.  
35     }  
36  
37     public void run() {  
38         for (;;) {  
39             if (data.idx >= data.ar.length - 1)      // 당 공유 인덱스가 공유 객체 소속 배열 ar의 인덱스  
40                 break;                         // -종료값이면 루프 종료와 동시에 스레드 종료.  
41             data.idx++;  
42             data.ar[data.idx] = 1;    // 공유 객체에 저장할 InputOne스레드의  
43         }                                     // -데이터를 특정시키기 위해 숫자 1로 가정.  
44     }  
45 }  
46  
47 class InputTwo extends Thread {  
48     Data data;  
49  
50     public InputTwo(Data data) {  
51         this.data = data;  
52     }  
53  
54     public void run() {  
55         for (;;) {  
56             if (data.idx >= data.ar.length - 1)  
57                 break;  
58             data.idx++;  
59             data.ar[data.idx] = 2;    // 공유 객체에 저장할 InputTwo스레드의  
60         }                           // -데이터를 특정시키기 위해 숫자 2로 가정.  
61     }  
62 }
```

```
63
64 class Main {
65     public static void main(String[] args) {
66         Data data = new Data(20);           // 공유 객체 생성.
67         InputOne inputOne = new InputOne(data);    // 각 스레드가 공유 객체에 접근하기 위해 각-
68         InputTwo inputTwo = new InputTwo(data);    // 스레드의 생성자에 공유 객체 data 전달.
69
70         inputOne.start();
71         inputTwo.start();
72
73     try {
74         inputOne.join();                  // 스레드 inputOne, inputTwo의 모든 실행 완료 후 69행에서
75         inputTwo.join();                // 출력을 제어하기 위해 각 스레드 객체에 대한 join 설정.
76     } catch (InterruptedException e) { }
77
78     data.outputData();
79 }
80 }
```

(2) 동기화 블럭(Critical section)

17-4-2-1

th1										th2									
스레드 텐	1	3	4	5	6	8	스레드 텐	2	7										
n++							n++												
	1	3	4	5	6			2	7										
sum	1	3	4	5		7	sum	2	7										
	n							n											

17-4-2-2

th1										th2									
스레드 텐	1	3	4	5	8	스레드 텐	2	6	7										
n++						n++													
	1	3	4	5			2		6										
sum	1	3	4		6	sum			5	6									
	n							n											

<1> 멀티스레드의 실행 특성과 공유 객체

```
1① /*
2 * < 두 개의 스레드를 이용한 1부터 100까지 누적합 > - 두 개의 스레드 th1과 th2를 1대 1로 번갈아 가면서 실행시켜 합을 구하기 위함이 주 목적 .
3 *
4 * - 개별 스레드의 너무 빠른 처리속도로 인한 선 점유 스레드의 독점 현상을 방지하기 위해 일정 시간 해당 스레드들을 대기 . 여기서 주목할 점은 공유
5 * 객체의 정보에 대한 개별 스레드들의 동시 접근은 어느 시점에서나 가능하지만 동일 시점이 아니더라도 동시 처리(연산)는 불가능하다는 것 .
6 * 아래의 경우 공유 정보 n의 접근은 각 스레드가 어느 위치에서나 동시 접근이 가능한 반면, 동시 처리는 불가하여 스레드 th1이 n에 접근하여 1을
7 * 증가시킴과 거의 동시에 스레드 th2도 n에 접근하여 1을 증가시킨다면 그 시점에 관계없이 n을 선점한 스레드가 먼저 1을 증가시킨 후 다른
8 * 스레드가 다시 1을 증가시켜 결과적으로 2가 증가된 상태로 두 스레드가 누적합 연산을 처리하게 됨 . 만약 동시 처리가 가능하다면 같은 연산이
9 * 하나로 통합될 것이므로 1만 증가될 것이고 두 스레드가 실제 동시에 실행이 가능하다는 것을 의미하므로 이는 기계(컴퓨터)에 대한 이해 부족으로 인한
10 * 논리적 오류 .
11 * 앞서 언급한 바와 같이 문제는 개별 스레드들의 실행 시점을 예측할 수 없는 상황에서 공유 객체에 대한 스레드 별 순간적인 중복 처리와 딜레이가 문제
12 * 발생의 원인 . 목적에 맞는 정상적 상황이 기반이 된다면 임의의 스레드가 공유 객체에 접근하여 처리를 진행하는 중에는 다른 스레드의 접근이
13 * 허용되서는 안되며 점유 스레드의 모든 연산이 완료된 후에만 다른 스레드의 접근을 허용해야지만 중복 처리와 딜레이로 인한 문제점 해소 가능 . 예를
14 * 들어 스레드 th1이 n을 증가시켜 합을 누적시켜 연산을 모두 완료 후 th2가 연산을 진행한다면 아무런 문제가 되지 않겠으나 스레드의 특성상
15 * 임의의 시점에서 n이 어떤 값(5)을 가지고 있을 때(5턴) 스레드 th1이 n을 증가시킨 직후(6)(6턴) th2도 바로 n을 증가(7)(7턴)시키는
16 * 중복 연산이 처리될 수 있어 th1의 6턴에 해당하는 n값(6)의 누적합을 건너 뛰고 th2에 대한 연산 결과값(7)을 가지고 두 스레드가 합을
17 * 누적시키는 현상 발생 가능 . - < 그림(17-4-2-1) 5 ~ 8턴 참조 >
18 * 또는 th2가 임의의 시점에서 누적합을 구하기 직전 딜레이된 상태(2턴)에서 스레드 th1이 홀로 계속 처리되다가 n증가 연산 직후이면서 합을
19 * 누적시키기 직전 n이 5인 순간(5턴)을 가정할 때, 시스템 상황이나 스케줄러의 판단에 의해 th1의 처리가 할나 지연되어 앞서 딜레이되었던
20 * th2의(2턴) n값(2)이 6턴에서 5로 변경된 상태로 n값을 누적시키고 빠르게 다음 7턴으로 넘어가 n값을 증가(6)시킨 후까지 th1이 아직
21 * 딜레이되어(5턴) 턴이 바뀌지 않고 있다면, 앞서 th2의 빠른 턴에 의한 n의 증가로 th1이 합을 누적시키기 직전(5턴)의 n값(5)을 유지하지
22 * 못하고 변경된 상태(6)(8턴)가 되어 결과적으로 두 스레드가 중복된 n값(6)을 보유한 채로 누적합을 연산하게 됨 .
23 * - < 그림(17-4-2-2) 2턴과 5 ~ 8턴 참조 >
24 * 전자의 경우 공유 객체에 대한 순간적인 중복연산 처리로 해당 n값을 건너뛰고 중복된 다음값으로 누적합 연산이되어 1이 더 증가된 상태의 누적합
25 * 연산이 처리되나, 후자의 경우 하나의 스레드가 합을 누적시키기 직전 n값을 보유한 채 딜레이가 된 상태에서 다른 스레드가 먼저 n값을 증가시킨
26 * 후 두 스레드가 같이 변경된 동일한 중복값을 누적시킴으로써 전자와 같이 건너뛰는 값은 발생되지 않고 동일값이 2번의 턴을 차지한 채로 중복
27 * 누적되어 결과적으로 뒷부분 값들의 누적합 연산이 누락되는 현상 발생 . 이처럼 실행 순서와 시점을 예측할 수 없는 스레드의 특성상 공유 객체에 대한
28 * 스레드간 점유와 연산이 복합적으로 연계되는 상황에서 개별 스레드의 공유 객체에 대한 독립적인 점유상태를 확보하지 못하면 예측할 수 없는 결과 초래 .
29 */
30 class Num {
31     int sum=0;
32     int n=0;
33 }
34 }
```

```
35 class Th1 extends Thread {  
36     Num num;  
37  
38     public Th1(Num num) {  
39         this.num=num;  
40     }  
41  
42     public void run() {  
43         for (int i = 1; i <= 50; i++) {  
44             num.n++;  
45             num.sum += num.n;  
46  
47             System.out.println(i + " : " +num.n); // 대략적인 실행 과정을 확인하기 위한 디버깅 코드.  
48             try {  
49                 Thread.sleep(10);  
50             } catch (InterruptedException e) { }  
51         }  
52     }  
53 }  
54  
55 class Th2 extends Thread {  
56     Num num;  
57  
58     public Th2(Num num) {  
59         this.num=num;  
60     }  
61  
62     public void run() {  
63         for (int i = -1; i >= -50; i--) {  
64             num.n++;  
65             num.sum += num.n;  
66         }
```

```
67     System.out.println(i + " : " +num.n);
68     try {
69         Thread.sleep(10);
70     } catch (InterruptedException e) { }
71 }
72 }
73 }
74
75 class Main {
76     public static void main(String[] args) {
77         Num num = new Num();
78         Th1 th1 = new Th1(num);
79         Th2 th2 = new Th2(num);
80
81         th1.start();
82         th2.start();
83
84         try {
85             th1.join();
86             th2.join();
87         } catch (InterruptedException e) { }
88
89         System.out.println();
90         System.out.println(num.sum);
91     }
92 }
```

<2> 동기화 블럭을 이용한 멀티스레드 제어

```
1  /*
2   *  < 동기화 블럭 - Critical section >
3   *
4   *  형식 : synchronized( 공유 객체 ) { 크리티컬 섹션 }
5   *
6   *  - 해당 스레드에 지정된 공유 객체에 대한 동기화 블럭 내(크리티컬 섹션)에서 공유 객체를 점유하는 동안은 외부 스레드의 공유 객체에 대한 접근을
7   *  일제 차단.
8   *  - 이처럼 동기화 블럭 지정을 통해 공유 객체에 대한 스레드간 상대적 점유를 인정함으로써 공유 객체의 중복 처리로 인한 문제점이 해소되어 매
9   *  실행 시 마다 정상적인 누적합 결과 확인 가능. 하지만 이는 누적합과 항값 n의 결과만 판단한다면 언뜻 정상적인 결과로 볼 수 있겠으나, 회전수
10  *  i의 출력 결과를 분석해보면 스레드간 1:1교환 처리가 정상적으로 일어나지 않고 불규칙하게 처리됨을 확인 가능. 이는 스레드의 딜레이로 인한
11  *  문제가 아직 해결되지 않았음을 의미하며 제대로된 동기화로 볼 수 없음. 동기화 블럭의 역할은 스레드간 중복 점유를 방어할 뿐이므로 스레드
12  *  자체의 딜레이 문제가 해소되지 않음이 당연. 스레드의 딜레이는 시스템 상황과 스케줄링과 같은 외부의 불가항력적인 요소에 의해 발생되는
13  *  현상으로 딜레이 자체를 내부적으로 제거하는 것은 불가. 따라서 이를 해결하고자 한다면 아래와 같은 스레드 제어를 가능케 하는 별도의 내부적
14  *  제어 장치와 함께 스레드의 논리적 제어를 통제할 정교한 알고리즘 필요.
15  *
16  *
17  *  < 교환 멀티 스레드 제어 알고리즘 >
18  *
19  *  A스레드의 실행 -> A스레드 내부 작업 완료 -> B스레드의 대기 해제 -> A스레드 대기 및 공유 객체 점유 해제 ->
20  *  B스레드의 실행 -> B스레드 내부 작업 완료 -> A스레드의 대기 해제 -> B스레드 대기 및 공유 객체 점유 해제 -> 19행으로 반복.
21  *
22  *  - 상기의 스레드 제어 과정 중 주의해야 할 점은 첫 번째, 주 스레드에 대한 상대 스레드의 대기 해제와 주 스레드의 대기 및 공유 객체 점유 해제
23  *  순서를 변경해서는 안된다는 것. 만약 이 순서를 바꾸어 주 스레드의 대기를 먼저 시키면 이후에 처리되는 주 스레드의 모든 작업이 대기 처리되어
24  *  상대 스레드의 해제가 불가하게 됨에 따라 서로 대기 상태가 되는 무한 대기 상태가 유지될 것임. 두 번째 주의해야 할 점은 최종 실행되는 주
25  *  스레드가 실행되는 시점은 당연히 상대 스레드의 모든 실행이 끝난 상태이고 스레드의 제어 논리에 의해 최종 주 스레드의 대기 처리가 진행되어 주
26  *  스레드의 대기를 해제해 줄 상대 스레드가 존재하지 않음으로 인한 프로그램이 종료되지 않는 무한 대기 상태가 될 것임. 따라서 최종 주 스레드에
27  *  대해서는 자신을 해제할 수 있는 별도의 제어 필요.
28  */
29 class Num {
30     int sum=0;
31     int n=0;
32 }
```

```
33
34 class Th1 extends Thread {
35     Num num;
36
37    public Th1(Num num) {
38        this.num=num;
39    }
40
41    public void run() {
42        for (int i = 1; i <= 50; i++) {
43            synchronized (num) { // 동기화 블럭 설정.
44                num.n++;
45                num.sum += num.n;
46
47                System.out.println(i + " : " +num.n); // 대략적인 실행 과정을 확인하기 위한 디버깅 코드.
48                try {
49                    Thread.sleep(10);
50                } catch (InterruptedException e) { }
51            }
52        }
53    }
54 }
55
56 class Th2 extends Thread {
57     Num num;
58
59    public Th2(Num num) {
60        this.num=num;
61    }
62 }
```

```
63④ public void run() {
64    for (int i = -1; i >= -50; i--) {
65        synchronized (num) {
66            num.n++;
67            num.sum += num.n;
68
69            System.out.println(i + " : " + num.n); // 대략적인 실행 과정을 확인하기 위한 디버깅 코드.
70        try {
71            Thread.sleep(10);
72        } catch (InterruptedException e) { }
73    }
74 }
75 }
76 }
77
78 class Main {
79④     public static void main(String[] args) {
80         Num num = new Num();
81         Th1 th1 = new Th1(num);
82         Th2 th2 = new Th2(num);
83
84         th1.start();
85         th2.start();
86
87     try {
88         th1.join();
89         th2.join();
90     } catch (InterruptedException e) { }
91
92         System.out.println(num.sum);
93     }
94 }
```

(3) 공유 객체에 대한 점유 대기와 해제

<1> wait과 notify

```
10 /*
2  *  wait( [long millis] ) : 공유 객체에 직접 wait()지정을 함으로써 호출 시점 이후에 대한 주 스레드의 모든 작업을 동기화 블럭
3  *  내/외부와 관계없이 대기시키고 공유 객체에 대한 점유를 대기 시간동안 해제. 인수로 대기 시간을
4  *  지정가능하며 인수를 0으로 설정하거나 생략 시 해제 전까지 무한대기 상태로 설정. 공유 객체에 대한
5  *  점유 제어를 담당하므로 반드시 동기화 블럭 내에 설정되어야 하며 wait, sleep, join등과 같이
6  *  스레드의 대기를 지정하는 메서드는 외부 인터럽트 발생에 의한 영향을 받을 수 있어 반드시 예외 처리
7  *  필수. 외부 인터럽트 발생으로 CPU점유를 빼앗기는 경우 스레드의 대기를 지속함으로써 프로세스가
8  *  다운될 수 있어 인터럽트 발생 시 강제로 대기를 해제해 버림. 이에 따른 요청 대기가 해제됨으로써 발생할
9  *  수 있는 문제점들을 방지하기 위해 문법적으로 예외 처리를 강제함.
10 *
11 *  notify() : wait과 마찬가지로 공유 객체에 직접 지정하며 공유 객체에 대한 wait()지정으로 인한 임의 스레드의 대기를 해제. 다중
12 *  스레드가 대기 중인 경우는 최초 대기(wait)가 지정된 순으로 호출 시점에 따라 하나씩 순차적으로 해제를 진행하되 대기
13 *  중인 모든 스레드의 대기를 한꺼번에 해제 시는 notifyAll()메서드 이용. 공유 객체에 대한 점유 제어를 담당하므로
14 *  wait()메서드와 마찬가지로 반드시 동기화 블럭 내 설정되어야 하며 공유 객체에 대한 대기를 해제하는 역할이므로 예외 처리는
15 *  불필요.
16 */
17 class Num {
18     int sum=0;
19     int n=0;
20 }
21
22 class Th1 extends Thread {
23     Num num;
24
250     public Th1(Num num) {
26         this.num=num;
27     }
28 }
```

```

29 public void run() {
30     for (int i = 1; i <= 50; i++) {
31         synchronized (num) {
32             try {
33                 num.wait();                                // 주 스레드에 대한 대기 및 공유 객체 점유 해제. 주의할 점은 wait지정을
34             } catch (InterruptedException e) { }        // -한다해도 외부 인터럽트 발생에 의해 대기가 강제 해제될 수 있음에 유의.
35             num.n++;                                 // 즉, th1 스레드에 대한 wait지정으로 인해 th1이 먼저 실행이 될 수
36             num.sum += num.n;                         // -없음에도 먼저 실행되는 경우가 발생한다면 이는 인터럽트 발생에 의해-
37                                         // wait지정이 강제 해제되어 실행되는 예외적인 경우.
38
39             System.out.println(i + " : " +num.n);
40         }
41     }
42 }
43 }

45 class Th2 extends Thread {
46     Num num;
47
48     public Th2(Num num) {
49         this.num=num;
50     }
51
52     public void run() {
53         for (int i = -1; i >= -50; i--) {
54             synchronized (num) {
55                 num.n++;                                // num.n++;
56                 num.sum += num.n;
57
58                 System.out.println(i + " : " +num.n);
59             }
56         }
57     }
58 }
59

```

```
60             num.notify();          // wait()메서드에 의해 대기 중인 스레드의 대기 해제. notify()메서드를 호출함으로써 대기중이던 -
61         }
62     }
63 }
64 }
65
66 // 상대 스레드의 대기 해제는 되었으나, 주 스레드가 동기화 블럭 내 공유 객체를 점유하는 동안은 실행되지
67 // -못하기 때문에 상대 스레드가 해제되어 바로 실행되는 것은 아님. 즉, 주 스레드의 동기화 블럭 내 모든
68 // -처리가 완료되어야만 대기중이던 상대 스레드의 공유 객체 접근이 허용. 이에 따라 주 스레드의 공유 객체
69 // -점유가 끝난 상태에서만 대기하던 상대 스레드의 실행이 가능한데, 대기 상태였던 상대 스레드가 실행되기
70 // -직전의 할당이 지연될과 동시에 주 스레드의 다음 루프 실행이 먼저 빠르게 진행된다면 대기 상태의 상대-
71 // 스레드가 다시 할당 기회를 놓치게 되어 주 스레드만 연속적으로 처리되는 문제 발생 가능.
72
73 class Main {
74     public static void main(String[] args) {
75         Num num = new Num();
76         Th1 th1 = new Th1(num);
77         Th2 th2 = new Th2(num);
78
79         th1.start();
80         th2.start();
81
82         try {
83             th1.join();
84             th2.join();
85         } catch (InterruptedException e) { }
86
87         System.out.println(num.sum);
88     }
89 }
```

<2> 대기와 해제의 순서

```
10 /*
2 * ※ 아래의 경우 각 스레드가 해제(notify) 후 대기(wait)를 진행하였으나 반대로 대기 후 해제를 하게 되면 각 스레드가 최초의 연산 후
3 * 둘 다 대기를 하게 됨으로써 해제를 해 줄 대상이 없어 최초 연산만 진행된 후 프로그램이 종료되지 않고 대기하게 됨. 하지만 아래와 같이 해제
4 * 후 대기를 진행하여도 정상적인 출력 결과가 확인은 되나 최종 실행 스레드의 대기를 해제해 줄 상대가 없어 프로그램이 종료되지 않는 동일
5 * 문제점 발생.
6 */
7 class Num {
8     int sum=0;
9     int n=0;
10}
11
12 class Th1 extends Thread {
13     Num num;
14
15     public Th1(Num num) {
16         this.num=num;
17     }
18
19     public void run() {
20         for (int i = 1; i <= 50; i++) {
21             synchronized (num) {
22                 num.n++;
23                 num.sum += num.n;
24
25                 System.out.println(i + " : " +num.n);
26
27                 num.notify();
28
29                 try {
30                     num.wait();
31                 } catch (InterruptedException e) { }
32             }
33         }
34     }
35 }
```

```
34     }
35 }
36
37 class Th2 extends Thread {
38     Num num;
39
40     public Th2(Num num) {
41         this.num=num;
42     }
43
44     public void run() {
45         for (int i = -1; i >= -50; i--) {
46             synchronized (num) {
47                 num.n++;
48                 num.sum += num.n;
49
50                 System.out.println(i + " : " +num.n);
51
52                 num.notify();
53
54                 try {
55                     num.wait();
56                 } catch (InterruptedException e) { }
57             }
58         }
59     }
60 }
61 }
```

```
62 class Main {  
63     public static void main(String[] args) {  
64         Num num = new Num();  
65         Th1 th1 = new Th1(num);  
66         Th2 th2 = new Th2(num);  
67  
68         th1.start();  
69         th2.start();  
70  
71     try {  
72         th1.join();  
73         th2.join();  
74     } catch (InterruptedException e) { }  
75  
76     System.out.println(num.sum);  
77 }  
78 }
```

<3> 제한적 대기(wait)

```
1 class Num {  
2     int sum=0;  
3     int n=0;  
4 }  
5  
6 class Th1 extends Thread {  
7     Num num;  
8  
9     public Th1(Num num) {  
10         this.num=num;  
11     }  
12  
13     public void run() {  
14         for (int i = 1; i <= 50; i++) {  
15             synchronized (num) {  
16                 num.n++;  
17                 num.sum += num.n;  
18  
19                 System.out.println(i + " : " + num.n);  
20  
21                 num.notify();  
22  
23                 try {  
24                     num.wait(1);  
25                 } catch (InterruptedException e) { }  
26             }  
27         }  
28     }  
29 }  
30
```

// 최종 실행 주 스레드의 무한 대기를 방지하고자 대기 시간 지정. 미와 같이 공유 -
// 객체에 대한 주 스레드의 대기 시간을 지정함으로써 매 실행 시마다 정상적인 연산
// -결과와 최종 스레드의 실행 대기로 인한 문제점이 해결되는 것을 확인 가능. -
// 하지만 각 스레드의 실행시간이 내부적으로 딜레이 되지 않고 정상적으로 처리 -
// 된다면 우연히 정상적인 결과를 확인 가능도 하나 각 스레드의 실행시간이란 것이
// -시스템 상황에 따라 지연될 수 있는 예측 불가능한 시간이므로 정상적 스레드간
// -교환 제어를 보장할 수 없음. 매 실행 시 회전수 i를 분석해 보면 언뜻 정상적인

```

31 class Th2 extends Thread {
32     Num num;
33
34     public Th2(Num num) {
35         this.num=num;
36     }
37
38     public void run() {
39         for (int i = -1; i >= -50; i--) {
40             synchronized (num) {
41                 num.n++;
42                 num.sum += num.n;
43
44                 System.out.println(i + " : " +num.n);
45
46                 num.notify();
47
48
49                 try {
50                     num.wait(1);
51                 } catch (InterruptedException e) { }
52
53 //                 try {
54 //                     Thread.sleep(100);
55 //                 } catch (InterruptedException e) { }
56
57         }
58     }
59 }
60

```

// -교환이 일어나는 것처럼 보이나 예외적인 스레드 딜레이 상황을 연출하기 위한 -
 // 54행의 주석처리를 해제 후 실행해 보면 1:1교환 실행이 정상적으로 일어나지 -
 // 않음을 확인 가능. 예컨대 th1이 정상적인 시간 대기를 하는 상황에서 th2가 -
 // 대기 해제 후 동기화 블럭을 벗어난 상태에서 딜레이가 발생한다면 th1의 대기가 -
 // 자동 해제되어 실행이 진행됨으로써 동기화 블럭을 벗어 나게되어 두 스레 모두 -
 // 동기화 블럭을 벗어나는 상태의 통제 불능의 상황 발생 가능. -
 // 즉, 주 스레드가 정상적인 대기 시간에 따라 실행되는 반면 상대 스레드의 실행 -
 // 자체가 딜레이됨으로 인한 주 스레드의 지정 시간 대기가 자동 해제됨으로써 두 -
 // 스레드 모두 공유 객체의 점유를 벗어난 상태가 되어 이후 스레드의 실행 제어는 -
 // 통제 밖의 상태로 전락. 미처럼 스레드의 대기 시간 지정은 개별 스레드의 실행미 -
 // -지연없이 정상적이라면 문제가 되지 않겠지만 그렇지 않다면 wait()메서드를 -
 // 지정하지 않았을 때와 크게 상황이 다르지 않음. 어쨌든 공유 객체에 대한 -
 // 제한적인 대기 지정으로 최종 실행 스레드의 무한대기는 해결되었으나 1:1교환 -
 // 실행은 미해결. 이를 위해서는 좀 더 정교한 별도의 스레드 제어 알고리즘 필요.

```
61 class Main {  
62     public static void main(String[] args) {  
63         Num num = new Num();  
64         Th1 th1 = new Th1(num);  
65         Th2 th2 = new Th2(num);  
66  
67         th1.start();  
68         th2.start();  
69  
70         try {  
71             th1.join();  
72             th2.join();  
73         } catch (InterruptedException e) { }  
74  
75         System.out.println(num.sum);  
76     }  
77 }
```

<4> 교환 실행 멀티 스레드의 완성

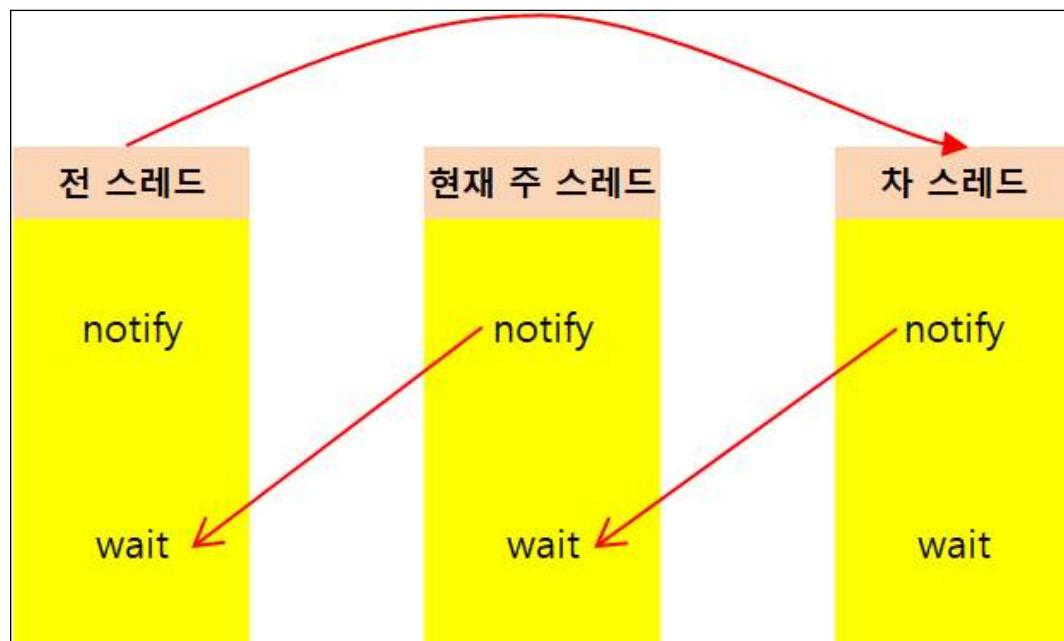
```
1 class Num {  
2     int sum=0;  
3     int n=0;  
4 }  
5  
6 class Th1 extends Thread {  
7     Num num;  
8  
9     public Th1(Num num) {  
10         this.num=num;  
11     }  
12  
13     public void run() {  
14         for (int i = 1; i <= 50; i++) {  
15             synchronized (num) {  
16                 num.n++;  
17                 num.sum += num.n;  
18  
19                 System.out.println(i + " : " +num.n);  
20  
21                 num.notify();  
22  
23                 try {  
24                     num.wait( (i <= 49) ? 0 : 1 );  
25                 } catch (InterruptedException e) { }  
26             }  
27         }  
28     }  
29 }  
30 // 최종 실행 스레드 전까지는 무한대기를 통해 1:1교환 실행을-  
// 성립시키되 최종 실행 스레드의 경우만 무한대기를 회피하기-  
// 위해 대기 시간을 지정함으로써 모든 문제점 해결.
```

```
31 class Th2 extends Thread {  
32     Num num;  
33  
34     public Th2(Num num) {  
35         this.num=num;  
36     }  
37  
38     public void run() {  
39         for (int i = -1; i >= -50; i--) {  
40             synchronized (num) {  
41                 num.n++;  
42                 num.sum += num.n;  
43  
44                 System.out.println(i + " : " +num.n);  
45  
46                 num.notify();  
47  
48                 try {  
49                     num.wait( (i >= -49) ? 0 : 1 );  
50                 } catch (InterruptedException e) { }  
51             }  
52  
53             try {  
54                 Thread.sleep(10);  
55             } catch (InterruptedException e) { }  
56         }  
57     }  
58 }  
59
```

```
61 class Main {  
62     public static void main(String[] args) {  
63         Num num = new Num();  
64         Th1 th1 = new Th1(num);  
65         Th2 th2 = new Th2(num);  
66  
67         th1.start();  
68         th2.start();  
69  
70         try {  
71             th1.join();  
72             th2.join();  
73         } catch (InterruptedException e) { }  
74  
75         System.out.println(num.sum);  
76     }  
77 }
```

(4) 교환 실행 멀티 스레드의 제어 알고리즘 분석

17-4-4



1^{Theta} /*
2 * < 교환 실행 멀티 스레드의 대기 해제와 대기 알고리즘에 따른 스레드 대기 해제의 상대적 주체 > - 그림(17-4-4) 참조.
3 */

4 * 앞서 검토한 바와같이 멀티 스레드의 동시 대기가 발생하는 문제로 인해 선 상대 스레드의 대기 해제 후 주 스레드 자신을 대기시켜야 함을 인지. 이와 연관되어
5 * 시간흐름에 따른 상대적 대기 해제의 대상을 분석해 봄으로써 멀티 스레드 제어에 대한 구조적 특징을 확인 가능. 두 스레드가 교환 실행이 되어야 하는 상황에서
6 * 먼저 주 스레드의 대기 해제 대상은 주 스레드의 직전 스레드만 가능한데, 이는 과거에 이미 실행되고 있던 직전 스레드가 대기함으로써 현재의 주 스레드가 대기
7 * 해제를 시켜주는 것이 당연한 논리적 흐름이며 아직 실행되지도 않은 차 스레드의 대기 해제는 불가능함을 인지. 또한 역으로 현재 주 스레드에 대한 대기 해제의
8 * 주체는 차 스레드만 가능한데, 이 또한 대기 해제와 대기의 상대적 제어 순서에 따라 현재 대기가 된 주 스레드의 대기 해제를 과거의 직전 스레드가 해준다는
9 * 것은 불가능. 이유인 즉슨 상대 스레드의 대기 해제 이후 자신을 대기시켜야 하는 논리상 직전 스레드는 주 스레드가 대기가 되는 시점에서 대기가 해제되는
10 * 시점이 되므로 차 스레드로 전환이 되어야만 대기 해제 명령이 가능. 결론적으로 주 스레드의 대기 해제 대상은 직전 스레드만 가능하고 주 스레드에 대한 대기
11 * 해제 주체는 차 스레드만 가능하며 여기서 차 스레드는 직전 스레드가 전환된 스레드를 의미.
12 */

```
13
14 class Data {           // 스레드 InputOne, InputTwo의 공유 객체.
15
16     int[] ar;          // 각 스레드의 데이터를 번갈아가면서 저장할 공유 배열.
17     int idx;           // 각 스레드가 공유 배열 ar에 접근하기 위한 공유 인덱스.
18
19 public Data(int size) {
20     ar = new int[size];
21     idx = -1;
22 }
23
24 public void outputData() {
25     for (int i = 0; i < ar.length; i++) {
26         System.out.println(ar[i]);
27     }
28 }
29 }
30
31 class InputOne extends Thread {
32     Data data;
33
34 public InputOne(Data data) {
35     this.data = data;
36 }
37
38 public void run() {
39     int time;           // 70행의 스레드 대기(wait) 시간.
40
41     for (;;) {
42         synchronized (data) {
43             data.idx++;
44 }
```

```

45     if (data.idx >= data.ar.length) { // 차인덱스가 배열크기 이상이라는 것은 당인덱스가 배열인덱스 종료값 이상임을 의미하고-
46         break; // 이는 최종 스레드를 의미하므로 실행 종료. 여기서 차인덱스의 점유는 스레드의 교환 -
47                                         // 구조상 최종 스레드가 67행과 같이 직전 스레드를 대기 해제시킨 후 70행과 같이 대기함-
48                                         // 으로써 최종 스레드의 직전 스레드가 차 스레드로 전환되어 점유한 상태임. 따라서 -
49                                         // 차인덱스를 점유한 차 스레드(전환된 직전 스레드)의 종료시점 이후가 실제 최종 스레드가
50                                         // -종료되는 시점이 되므로 이때 최종 스레드는 차차인덱스를 점유한 상태가 될것임.
51                                         // (중요) ※ 구조 변수의 해석은 항상 구하는식에서 종료시점까지 해석함에 유의. 55행의
52                                         // -경우 종료 시점이 아니므로 45행의 종료시점과 연결해서 해석해서는 안되며 최초 초기값
53                                         // -의미로 해석해야 함.
54
55     } else if (data.idx >= data.ar.length - 1) { // 당인덱스가 배열인덱스 종료값 이상이면 이는 최종 스레드를 의미하며-
56         time = 1; // 스레드 교환 알고리즘에 따라 최종 스레드를 기준으로 하여 직전 스레드는
57                                         // -최종 스레드가 67행과 같이 대기 해제를 처리해줌으로써 차 스레드로-
58                                         // 전환되어 정상적인 종료가 가능하나 70행과 같이 대기된 최종 스레드는-
59                                         // 앞서 대기 해제된 직전 스레드가 차 스레드로 전환되어 소멸된 상태이므로
60                                         // -해제하여 줄 대상 스레드가 존재하지 않아 56행과 같이 임의의 -
61                                         // 제한적인 대기 시간을 지정함으로써 최종 스레드의 무한 대기 발생 회피.
62     } else {
63         time = 0; // 최종 스레드가 아닌 직전 이하의 모든 스레드들은 각 시점의 차 스레드가 대기 해제를 처리해줌으로써 -
64     } // 안정적인 교환 실행을 위해 반드시 무한 대기로 설정해야 함. 해당 주제의 경우 63행에 임의의 시간을
65                                         // -지정하고 반복 실행을 하였을 때 우연히 정상적인 결과를 확인 가능도 하나 이는 해당 프로그램 자체가
66         data.ar[data.idx] = 1; // -단순하고 시스템 내부 상황이 안정적인 상황에 한하며, 그렇지 못한 경우 개별 스레드의 작업시간이
67         data.notify(); // -딜레이 되거나 작업 종료 후 동기화 블럭을 벗어난 상태에서 딜레이 되면 각 스레드의 지정된 대기 -
68                                         // 시간이 도달하면서 서로 자동 해제된 상태가 되어 두 스레드 모두 동기화 블럭을 벗어난 상태가 됨으로써 -
69     try { // 투제 불능의 상태가 될 수 있으므로 유의해야 함.
70         data.wait(time);
71     } catch (InterruptedException e) { }
72     }
73   }
74 }
75 }
76

```

```
77 class InputTwo extends Thread {  
78     Data data;  
79  
80     public InputTwo(Data data) {  
81         this.data = data;  
82     }  
83  
84     public void run() {  
85         int time;  
86  
87         for (;;) {  
88             synchronized (data) {  
89                 data.idx++;  
90  
91                 if (data.idx >= data.ar.length) {  
92                     break;  
93                 } else if (data.idx >= data.ar.length - 1) {  
94                     time = 1;  
95                 } else {  
96                     time = 0;  
97                 }  
98  
99                 data.ar[data.idx] = 2;  
100                data.notify();  
101  
102                try {  
103                    data.wait(time);  
104                } catch (InterruptedException e) { }  
105            }  
106  
107            try {  
108                Thread.sleep(10);  
109            } catch (InterruptedException e) { }  
                                         // 스레드 딜레이 상황을 연출하기 위한 임시 코드.
```

```
110      }
111  }
112 }
113
114 class Main {
115     public static void main(String[] args) {
116         Data data = new Data(20);
117         InputOne inputOne = new InputOne(data);
118         InputTwo inputTwo = new InputTwo(data);
119
120         inputOne.start();
121         inputTwo.start();
122
123     try {
124         inputOne.join();
125         inputTwo.join();
126     } catch (InterruptedException e) { }
127
128     data.outputData();
129 }
130 }
```

(5) 스레드 대기 해제 알고리즘의 최적화

```
1 class Data {
2     int[] ar;
3     int idx;
4
5     public Data(int size) {
6         ar = new int[size];
7         idx = -1;
8     }
9
10    public void outputData() {
11        for (int i = 0; i < ar.length; i++) {
12            System.out.println(ar[i]);
13        }
14    }
15 }
16
17 class InputOne extends Thread {
18     Data data;
19
20     public InputOne(Data data) {
21         this.data = data;
22     }
23
24     public void run() {
25         for (;;) {
26             synchronized (data) {
27                 data.notify(); // 스레드 대기 해제와 대기의 순서는 유지하되 대기 해제의 시점을 당행과 같이 스레드 종료 제어 시점 이전으로 -
28                 // 당김으로써 40행에서 대기된 최종 스레드를 차 스레드로 전환된 직전 스레드가 종료시점 이전에 대기 해제를 -
29                 // 하여 최종 스레드에 대한 무한 대기 현상을 해소시킬 수 있고 이에 따라 최종 스레드에 대한 제한적 스레드 -
30                 // 대기에 대한 추가 제어 구조 불필요.
31 }
```

```

32         if (data.idx >= data.ar.length - 1) { // 36행의 인덱스 생성식 위치 변화에 따른 스레드 종료 제어 시점 변경과 스레드 제어
33             break; // -알고리즘에 따라 당인덱스가 배열인덱스 종료값 이상이라는 것은 40행에서 최종-
34         }
35         data.idx++;
36         data.ar[data.idx] = 1; // 스레드가 대기된 상태에서 대기 해제된 직전 스레드가 차 스레드로 전환되어 최종-
37 // -스레드의 공유 인덱스를 동시에 보유한 상태가 되므로 스레드 종료. 최종 스레드
38 // -또한 차 스레드로 전환된 직전 스레드의 27행에서의 대기 해제로 대기가 -
39 // 해제되면서 루프를 돌아 공유 인덱스를 유지한 채로 스레드 종료.
40
41     try {
42         data.wait(0);
43     } catch (InterruptedException e) { }
44 }
45 }
46
47 class InputTwo extends Thread {
48     Data data;
49
50     public InputTwo(Data data) {
51         this.data = data;
52     }
53
54     public void run() {
55         for (;;) {
56             synchronized (data) {
57                 data.notify();
58
59                 if (data.idx >= data.ar.length - 1) {
60                     break;
61                 }
62
63                 data.idx++;
64                 data.ar[data.idx] = 2;

```

```
65             try {
66                 data.wait(0);
67             } catch (InterruptedException e) { }
68         }
69     }
70 }
71 }
72 }
73
74 class Main {
75     public static void main(String[] args) {
76         Data data = new Data(20);
77         InputOne inputOne = new InputOne(data);
78         InputTwo inputTwo = new InputTwo(data);
79
80         inputOne.start();
81         inputTwo.start();
82
83         try {
84             inputOne.join();
85             inputTwo.join();
86         } catch (InterruptedException e) { }
87
88         data.outputData();
89     }
90 }
```

(6) 동기화 메서드

```
1① /*
2  * <동기화 메서드와 동기화 블럭의 차이>
3  *
4  * 동기화 블럭은 공유객체를 지정하여 동기화 블럭이 설정되어있는 스레드가 점유함을 명시함으로써 공유객체에 대한 다른 스레드의 접근 차단.
5  *
6  * 동기화 블럭은 임의의 스레드 내에서 공유객체를 지정하는 반면 동기화 메서드는 공유객체 자기 자신 즉, this에 대한 메서드 자체에 동기화
7  * 블럭을 설정. 따라서 동기화 메서드는 동기화 블럭처럼 별도의 공유객체를 지정을 하지 않으며 단지 메서드에 synchronized 키워드를
8  * 추가함으로써 간단히 동기화를 설정. 단, 동기화 메서드는 별도의 공유 객체 지정이 없으므로 wait, notify의 주체가 다른 객체가 될수
9  * 없고 공유객체 자기 자신(this)으로만 사용이 제한되므로 공유객체가 아닌 서로 다른 객체나 개별 스레드에 적용하는 것은 무의미하며 또한,
10 * 동기화 메서드 내부에 존재한다해서 그 객체가 공유객체가 되는 것도 아님. 결과적으로 동기화 메서드는 동기화 메서드가 설정된 주 객체가
11 * 공유객체가 되므로 공유객체 내에 다수의 동기화 메서드가 존재하는 경우 클래스 내의 메서드들을 서로 동기화 시켜 임의의 A스레드가 그 중
12 * 어떤 하나의 동기화 메서드를 호출하여 사용하고 있다면 임의의 다른 B스레드가 A가 사용중인 동기화 메서드를 포함한 다른 모든 동기화
13 * 메서드들의 사용이 제한.
14 */
15 class Data implements Runnable {
16     String ar = "01234567890123456789";
17     int idx = -1;
18
19②     public synchronized void run() {
20         for (;;) {
21             this.notify();
22             if (idx >= ar.length() - 1) {
23                 break;
24             }
25
26             idx++;
27             System.out.print(ar.charAt(idx) + " ");
28                         // char charAt(int index) : String클래스의
29                         // -메서드로 인수에 해당하는 인덱스의 문자값 리턴.
30
31             System.out.println(Thread.currentThread().getName());
32     }
33 }
```

```
31         try {
32             this.wait(0);
33         } catch (InterruptedException e) { }
34     }
35 }
36 }
37 }
38
39 class Main {
40     public static void main(String[] args) {
41         Data data = new Data();
42         Thread thread1 = new Thread(data);
43         Thread thread2 = new Thread(data);
44
45         thread1.start();
46         thread2.start();
47     }
48 }
```

(7) 사용자 인터럽트

```
1 class Event extends Thread {  
2     public void run() {  
3         for(;;) {  
4             System.out.println("이벤트 처리중...");  
5             try {  
6                 Thread.sleep(1000);  
7             } catch (InterruptedException e) {  
8                 System.out.println("인터럽트 발생");  
9                 return;  
10            }  
11        }  
12    }  
13 }  
14  
15 class Main {  
16     public static void main(String[] args) {  
17         Event event = new Event();  
18  
19         event.start();  
20  
21         try {  
22             Thread.sleep(5000);  
23         } catch (InterruptedException e) { }  
24  
25         event.interrupt();          // sleep, wait, join등과 같이 스레드를 대기시키고 있을 때, 만약 더 이상 스레드의 -  
26     }                            // 대기가 필요없는 경우 미와 같이 interrupt()메서드를 통해 외부에서 강제로 인터럽트를-  
27 }                            // 발생시킴으로써 스레드의 대기를 해제 가능.
```

18. 입출력 스트림(I/O Stream)

- 스트림이란 키보드, 모니터, 파일, 오디오, 메모리, 네트워크등의 입출력 장치와 데이터를 주고받는 논리적 통로를 의미하는 것으로써 스트림이란 논리적인 장치를 이용하여 모든 입출력을 통일시킴으로써 일관성과 재사용성 증대. 입출력 스트림은 입출력 데이터를 동시에 전송이 불가능한 단방향의 성질을 지님.

(1) 이진(Binary) 출력 스트림

```
10 import java.io.FileOutputStream;
2 import java.io.IOException;
3
4 class Main {
5     public static void main(String[] args) {
6         byte[] data= { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
7
8         FileOutputStream outObj=null;          // 파일은 실제 메모리 내부에 존재하는 것이 아니라 외부에 존재하여 이에 따른 입출력 스트림에 대한 생성 및 제어 시 예외
9                                         // -발생 소지가 높아 15, 18, 24행과 같이 예외 처리 필수. 하지만 당행과 같이 입출력 스트림 변수는 예외 처리 블럭-
10                                         // 외부에 먼저 선언되어야 하고 또한 null로 초기화 필수. 이유인즉슨 입출력 스트림 변수를 예외 블럭 내부에 선언을 하면
11                                         // -예외 발생 시 변수 선언 자체가 무효화되어 24행의 outObj 객체의 인식 자체가 불가. 또한 null로 초기화를 하지 -
12                                         // 않으면 예외 발생 시 15행의 입출력 객체 생성 자체가 무효화되어 24행에서 쓰레기 값에 대한 close메서드 호출이 되어
13                                         // -컴파일 자체가 불가. 첨언하여 close메서드는 내부적으로 null포인터에 대한 예외 처리 포함.
14     try {
15         outObj=new FileOutputStream("Binaryout.bin");    // FileOutputStream(String name) : 출력할 파일의 경로와 파일 이름을 문자열-
16                                         // 형태로 전달. 전달된 경로에 파일을 생성하면서 출력 스트림 개방.
17
18         outObj.write(data);                            // void write(byte[] b) : 바이트형의 배열 형태로 인수를 전달받아 대상 객체에 출력.
19         System.out.println("파일에 데이터 출력 완료!!!");
20     } catch (IOException e) {
21         System.out.println("파일에 데이터 출력 실패!!!");
22     } finally {
23         try {
24             outObj.close();                          // 파일의 입출력 스트림 미해제 시 파일에 대한 스트림이 개방된 상태라 파일이 손상될 수
25         } catch (IOException e2) {                  // -있으며 메모리 상주로 인한 자원 소모가 될수 있어 반드시 스트림 닫기 필수.
26
27             System.out.println("파일 닫기 실패!!!");
28         }
29     }
30 }
31 }
```

(2) 출력 offset

```
1① /*
2  * < offset > : 상쇄하다.
3  *
4  * 배열의 경우 인덱스의 시작이 0부터 설정되는데 아래 13행에서처럼 그 위치를 조정하여 재설정하는 것 등을 의미.
5  */
6 import java.io.FileOutputStream;
7
8 class Main {
9②     public static void main(String[] args) throws Exception {
10        byte[] data = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
11        FileOutputStream outObj = new FileOutputStream("Binaryout2.bin");
12
13        outObj.write(data, 2, 5);                                // void write(byte[] b, int off, int len) : 인수가 되는 바이트 배열의
14                                         // -offset 위치를 설정하여 그 위치부터 len크기만큼 범위를 설정하여 출력.
15        System.out.println("파일에 데이터 출력 완료!!!");
16        outObj.close();
17    }
18 }
```

(3) 이진(Binary) 입력 스트림

```
1④ import java.io.FileInputStream;
2 import java.io.IOException;
3
4 class Main {
5④     public static void main(String[] args) {
6         FileInputStream inObj=null;
7
8         try {
9             inObj=new FileInputStream("Binaryout.bin"); // FileInputStream(String name) : 입력받을 파일의 경로와 파일 이름을
10            int data;                                // -문자열형태로 전달. 전달된 경로의 파일로부터 입력 스트림 스트림 개방.
11
12         for(;;) {
13             data=inObj.read();                      // int read() : 1byte씩 읽은 데이터를 리턴하되 파일의 끝이면 -1 반환.
14
15             if( data == -1 ) break;
16
17             System.out.println(data);
18         }
19     } catch (IOException e) {
20         System.out.println("파일 읽기 실패");
21     } finally {
22         try {
23             inObj.close();
24         } catch (IOException e2) {
25             System.out.println("파일 닫기 실패!!!");
26         }
27     }
28 }
29 }
```

(4) 파일 복사

```
1① import java.io.FileInputStream;
2 import java.io.FileOutputStream;
3 import java.io.IOException;
4
5 class Main {
6②     public static void main(String[] args) {
7         FileInputStream inObj=null;
8         FileOutputStream outObj=null;
9
10    try {
11        inObj=new FileInputStream("Binaryout.bin");
12        outObj=new FileOutputStream("Binarycopy.bin");
13        int data;
14
15        for(;;) {
16            data=inObj.read();
17
18            if( data == -1 ) break;
19
20            outObj.write(data);
21        }
22        System.out.println("파일 복사 성공!!!");
23    } catch (IOException e) {
24        System.out.println("파일 복사 실패!!!");
25    } finally {
26        try {
27            inObj.close();
28            outObj.close();
29        } catch (IOException e2) {
30            System.out.println("파일 닫기 실패!!!");
31        }
32    }
33 }
34 }
```

(5) 버퍼(Buffer) 입출력 스트림

```
1① /*  
2  * < Buffer >  
3  *  
4  * 버퍼란 장치와 장치 사이의 속도차 해결을 위한 임시 메모리로써 CPU처럼 엄청나게 빠른 장치와 상대적으로 대단히 느린  
5  * 입출력 장치사이간에 직접 데이터 전송 시 입출력 장치가 데이터를 읽고 쓰는 시간 자체가 CPU에게는 상대적으로 엄청나게  
6  * 더딘 시간이므로 CPU의 유휴시간이 발생되어 그 만큼 처리시간의 지연이 발생. 따라서 내부 주기억장치에 버퍼란 임시  
7  * 기억 공간을 할당하고 입출력 데이터를 버퍼에 저장해 두었다가 한꺼번에 전송을 함으로써 CPU의 유휴시간을 해소할 수  
8  * 있고 더불어 데이터의 전송 효율을 높임으로써 장치간 직접 전송보다 상당히 빠른 처리가 가능.  
9 */  
10② import java.io.BufferedInputStream;  
11 import java.io.BufferedOutputStream;  
12 import java.io.FileInputStream;  
13 import java.io.FileOutputStream;  
14 import java.io.IOException;  
15  
16 class Main {  
17     public static void main(String[] args) {  
18         BufferedInputStream bufInObj = null;  
19         BufferedOutputStream bufOutObj = null;  
20         int data;  
21  
22         try {  
23             bufInObj=new BufferedInputStream(new FileInputStream("Binaryout.bin")); // 버퍼 입출력 스트림에 파일 입출력 스트림  
24             bufOutObj=new BufferedOutputStream(new FileOutputStream("Bufout.bin")); // -객체만 전달하면 이후의 데이터를 읽고-  
25             // 쓰는 등의 제어는 파일 입출력 스트림의 -  
26             for(;;) { // 메서드와 동일.  
27                 data=bufInObj.read();  
28  
29                 if(data == -1) break;  
30  
31                 bufOutObj.write(data);  
32             }  
33  
34             System.out.println("버퍼를 이용한 파일 데이터 입출력 완료!!");  
}
```

```
35
36     } catch (IOException e) {
37         System.out.println("버퍼를 이용한 파일 데이터 입출력 실패!!!");
38     } finally {
39         try {
40             bufInObj.close();           // 파일 입출력 스트림을 별도로 닫을 필요없이 버퍼 입출력
41             bufOutObj.close();        // -스트림만 닫아도 파일 입출력 스트림은 자동 해제.
42         } catch (IOException e2) {
43             System.out.println("버퍼를 이용한 파일 닫기 실패!!!");
44         }
45     }
46 }
47 }
```

(6) 파일 직접 입출력과 버퍼 입출력 비교

```
1⑧ import java.io.BufferedInputStream;
2 import java.io.BufferedOutputStream;
3 import java.io.FileInputStream;
4 import java.io.FileOutputStream;
5 import java.io.IOException;
6
7 class Main {
8⑨     public static void main(String[] args) {
9         FileInputStream input=null;
10        FileOutputStream out=null;
11        BufferedInputStream bufInput=null;
12        BufferedOutputStream bufOutput=null;
13        long std;
14        int data;
15
16        std=System.currentTimeMillis();
17        try {
18            input=new FileInputStream("자바교본정리3.pdf");
19            out=new FileOutputStream("copytest1.pdf");
20
21            for(;;) {
22                data=input.read();
23
24                if(data == -1) break;
25
26                out.write(data);
27            }
28        } catch (IOException e) {
29            System.out.println("파일 복사 실패!!");
30        } finally {
31            try {
32                input.close();
33                out.close();
```

```
34 } catch (IOException e2) {
35     System.out.println("파일 닫기 실패!!!");
36 }
37 }
38 System.out.printf("copytest1 복사 시간 %.3f초", (System.currentTimeMillis()-std)/1000.);
39
40
41 std=System.currentTimeMillis();
42 try {
43     bufInput=new BufferedInputStream(new FileInputStream("자바교본정리3.pdf"));
44     bufOutput=new BufferedOutputStream(new FileOutputStream("copytest2.pdf"));
45
46     for(;;) {
47         data=bufInput.read();
48
49         if(data == -1) break;
50
51         bufOutput.write(data);
52     }
53 } catch (IOException e) {
54     System.out.println("파일 복사 실패!!!");
55 } finally {
56     try {
57         bufInput.close();
58         bufOutput.close();
59     } catch (IOException e2) {
60         System.out.println("파일 닫기 실패!!!");
61     }
62 }
63 System.out.printf("\ncopytest2 복사 시간 %.3f초", (System.currentTimeMillis()-std)/1000.);
64 }
65 }
```

(7) Scanner

```
10 /*
2  *  < Scanner >
3  *
4  *  파일이나 문자열로부터 정보를 추출하는 기능 제공.
5  */
6 import java.util.Scanner;
7
8 public class Main {
9     public static void main(String[] args) {
10         Scanner scanner = new Scanner(System.in);           // 스캐너 객체의 생성자의 인수로 System클래스의 in객체(필드)를 전달하면 키보드로 부터
11
12
13
14         System.out.print("정수입력 : ");
15         int i=scanner.nextInt();                         // -입력받을 수 있는 입력 스트림이 개방되어 별도의 예외처리 없이 입력 정보 전달 가능. -
16         System.out.println("정수출력 : "+i+'\n');        // 첨부하여 System의 in객체(필드)는 추상클래스 InputStream의 서브 클래스 타입의
17
18
19         System.out.print("문자열입력(next) : ");
20         String str=scanner.next();                      // - 객체.
21         System.out.println("문자열출력(next) : "+str+'\n'); // int nextInt() : 정수형 입력 메서드. 다른 기본형에 대해서도 정의.
22
23
24
25         System.out.print("문자열입력(nextLine) : ");
26         String str1=scanner.nextLine();                 // String next() : 문자열 입력 메서드. 공백, 탭, 개행문자를 포함하지
27         System.out.println("문자열출력(nextLine) : "+str1+'\n'); // - 않는 직전 문자열까지만 인식.
28
29
30
31
32
33
34
```

// String nextLine() : 문자열 입력 메서드. 공백, 탭, 개행 문자까지 -
// 포함한 문자열 인식을 하되 마지막 입력 완료를 알리는 개행문자는 입력 데이터에 포함
// -되지 않음. 단, nextLine메서드 이전에 입력 메서드가 존재하고 그 메서드가
// -개행 문자를 포함하지 않는다면, 입력 버퍼에 남아 있던 개행문자가 nextLine-
// 메서드로 전이되어 입력이 무시되는 효과 발생. 따라서 19행의 next메서드는 -
// 개행문자를 포함하지 않으므로 입력 버퍼에 남아 있던 개행 문자가 당행의 nextLine-
// 메서드로 전이되어 입력 기회를 빼버리고 개행문자만 전달되며 마지막 개행문자는 -
// 입력 데이터로 인정되지 않아 26행의 str1은 널문자만 포함된 빈 문자열이 출력. -
// 이러한 문제점을 해결하려면 22행과 같이 nextLine메서드를 단순히 호출만 하여 -
// 입력 버퍼를 비움으로써 간단히 해결 가능.

```
35     System.out.print("문자열입력(nextLine) : ");
36     String str2=scanner.nextLine();           // 25행의 nextLine메서드가 개행문자를 포함하므로 당행의 메서드는 정상적으로 실행.
37     System.out.println("문자열출력(nextLine) : "+str2+'\n');
38
39     scanner.close();                         // System.in 입력 스트림을 해제. 미 해제 시 가지지컬렉터에 의해 할당이 해제되기-
40 }                                         // 전까지는 메모리 누수 발생.
41 }
```