

# 9. Java의 주요 API (중간 수준 목차)

## 9.1 java.lang

### Object, String, Math, System, Class 등 핵심 클래스

#### ◆ 1. Object 클래스

##### ☀ 개요

- 자바의 모든 클래스의 최상위 부모 클래스 (모든 클래스는 암묵적으로 `Object` 를 상속받음)
- 모든 객체는 최소한 `Object` 의 메서드를 상속받는다.

##### ✂ 주요 메서드

메서드	설명
<code>toString()</code>	객체의 문자열 표현 반환
<code>equals(Object obj)</code>	두 객체의 동등성 비교
<code>hashCode()</code>	객체의 해시코드 반환
<code>getClass()</code>	객체의 클래스 정보를 반환
<code>clone()</code>	객체 복제
<code>finalize()</code>	GC 직전에 호출되는 메서드 (Deprecated)
<code>wait()</code> , <code>notify()</code> , <code>notifyAll()</code>	동기화(스레드) 관련 제어용 메서드

##### ✅ 예시

```
1 class Person {
2     String name;
3     public Person(String name) {
4         this.name = name;
5     }
6
7     @Override
8     public String toString() {
9         return "Person: " + name;
10    }
11
12    @Override
13    public boolean equals(Object obj) {
14        if (obj instanceof Person p) {
15            return this.name.equals(p.name);
16        }
17        return false;
18    }
19 }
```

```

18     }
19
20     @Override
21     public int hashCode() {
22         return name.hashCode();
23     }
24 }
25
26 public class Main {
27     public static void main(String[] args) {
28         Person p1 = new Person("John");
29         Person p2 = new Person("John");
30
31         System.out.println(p1);           // Person: John
32         System.out.println(p1.equals(p2)); // true
33         System.out.println(p1.hashCode()); // 동일한 이름이면 동일 해시
34     }
35 }

```

## ◆ 2. String 클래스

### ☀ 개요

- 불변 객체 (immutable): 한 번 생성된 문자열은 수정 불가.
- 문자열 리터럴은 **String Constant Pool**에 저장됨.
- 문자열 조작은 `+`, `substring()`, `indexOf()` 등으로 수행.

### ✖ 주요 메서드

메서드	설명
<code>length()</code>	문자열 길이 반환
<code>charAt(int)</code>	인덱스 위치의 문자 반환
<code>substring()</code>	부분 문자열 반환
<code>equals()</code>	문자열 비교
<code>contains()</code> , <code>startsWith()</code> , <code>endsWith()</code>	패턴 매칭
<code>split()</code>	문자열 분리
<code>replace()</code>	문자열 대체
<code>toUpperCase()</code> , <code>toLowerCase()</code>	대소문자 변경
<code>trim()</code>	앞뒤 공백 제거

## ✅ 예시

```
1 public class Main {
2     public static void main(String[] args) {
3         String s = " Hello Java ";
4         System.out.println(s.trim().toUpperCase()); // "HELLO JAVA"
5
6         String[] parts = s.trim().split(" ");
7         for (String part : parts) {
8             System.out.println(part); // Hello, Java
9         }
10    }
11 }
```

## ◆ 3. Math 클래스

### ☀ 개요

- 수학 관련 정적 메서드를 제공
- 인스턴스를 만들 수 없음 (private 생성자)

### ✖ 주요 메서드

메서드	설명
<code>abs()</code> , <code>max()</code> , <code>min()</code>	기본적인 수치 계산
<code>pow(x, y)</code>	x의 y 제곱
<code>sqrt(x)</code>	제곱근
<code>round()</code> , <code>ceil()</code> , <code>floor()</code>	반올림, 올림, 내림
<code>random()</code>	0.0 이상 1.0 미만의 난수 반환
<code>sin()</code> , <code>cos()</code> , <code>tan()</code>	삼각 함수
<code>log()</code> , <code>exp()</code>	로그, 지수 함수

## ✅ 예시

```
1 public class Main {
2     public static void main(String[] args) {
3         double r = Math.random();        // 0.0 ~ 1.0
4         int n = (int)(r * 100);           // 0 ~ 99
5         System.out.println(n);
6
7         System.out.println(Math.pow(2, 10)); // 1024.0
8         System.out.println(Math.round(3.6)); // 4
9     }
10 }
```

## ◆ 4. System 클래스

### ☀ 개요

- 시스템 관련 기능을 제공하는 유틸리티 클래스
- `System.out`, `System.err`, `System.in` 등 표준 입출력
- `System.gc()`, `System.currentTimeMillis()` 등 제공

### ✖ 주요 메서드

메서드	설명
<code>System.out.println()</code>	표준 출력 스트림
<code>System.err.println()</code>	에러 출력 스트림
<code>System.in</code>	입력 스트림 (Scanner로 연결)
<code>exit(int)</code>	프로그램 종료
<code>gc()</code>	GC 요청
<code>currentTimeMillis()</code> , <code>nanoTime()</code>	시간 측정

## ✅ 예시

```
1 public class Main {
2     public static void main(String[] args) {
3         long start = System.currentTimeMillis();
4
5         for (int i = 0; i < 1000000; i++) {} // 단순 연산
6
7         long end = System.currentTimeMillis();
8         System.out.println("걸린 시간: " + (end - start) + "ms");
9
10        System.exit(0); // 명시적 종료
11    }
12 }
```

## ◆ 5. Class 클래스

### ☀ 개요

- 자바의 리플렉션 API 핵심 클래스
- 객체 또는 타입의 **메타정보**를 다룰 수 있음
- `getClass()` 혹은 `Class.forName()` 사용

### ✖ 주요 메서드

메서드	설명
<code>getName()</code> , <code>getSimpleName()</code>	클래스 이름
<code>getDeclaredFields()</code> , <code>getDeclaredMethods()</code>	필드/메서드 목록
<code>newInstance()</code> (Deprecated)	객체 생성
<code>getSuperclass()</code> , <code>isInterface()</code>	상속 및 타입 체크

## ✅ 예시

```
1 public class Main {
2     public static void main(String[] args) throws Exception {
3         Class<?> clazz = Class.forName("java.lang.String");
4         System.out.println(clazz.getSimpleName()); // String
5
6         for (var method : clazz.getDeclaredMethods()) {
7             System.out.println(method.getName());
8         }
9     }
10 }
```

## 🧠 심화 포인트 요약

클래스	주요 특징	일반 용도
<code>Object</code>	모든 클래스의 조상	공통 메서드 재정의
<code>String</code>	불변 객체	문자열 처리
<code>Math</code>	수학 함수 모음	수치 연산
<code>System</code>	시스템 제어	입출력, 시간, 종료
<code>Class</code>	리플렉션	메타 정보 획득, 동적 로딩

## StringBuilder, StringBuffer

`StringBuilder`와 `StringBuffer`는 Java에서 문자열을 효율적으로 조작할 수 있도록 만든 클래스이다. `String`은 불변 (immutable) 객체라서 문자열을 수정할 때마다 새로운 인스턴스가 만들어지는 반면, 이 둘은 가변 (mutable) 문자열 클래스로서, 성능 면에서 훨씬 유리하다.

### ◆ 개요 비교: `String` vs `StringBuilder` vs `StringBuffer`

구분	<code>String</code>	<code>StringBuilder</code>	<code>StringBuffer</code>
불변성	✅ 불변 (immutable)	❌ 가변 (mutable)	❌ 가변 (mutable)
스레드 안전	❌ 안전하지 않음	❌ 안전하지 않음	✅ 스레드 안전 (synchronized)
속도	느림 (매번 새 객체 생성)	빠름	느림 ( <code>synchronized</code> 때문)
용도	변경이 거의 없는 문자열	단일 스레드 문자열 조작	멀티스레드 환경에서 문자열 조작

### ◆ 1. `StringBuilder`

#### ☀️ 특징

- Java 5부터 추가
- 비동기 환경에서 빠른 문자열 수정 가능
- `append()`, `insert()`, `delete()` 등 다양한 메서드 제공

## ✅ 예시

```
1 public class Main {
2     public static void main(String[] args) {
3         StringBuilder sb = new StringBuilder("Hello");
4         sb.append(" world");
5         sb.insert(5, ",");
6         sb.replace(6, 11, "Java");
7         sb.delete(0, 1);
8
9         System.out.println(sb.toString()); // ello,Java
10    }
11 }
```

## ✖ 주요 메서드

메서드	설명
<code>append(String)</code>	문자열 끝에 추가
<code>insert(int, String)</code>	특정 위치에 삽입
<code>delete(int, int)</code>	문자열 삭제
<code>replace(int, int, String)</code>	특정 구간 대체
<code>reverse()</code>	문자열 뒤집기
<code>toString()</code>	<code>String</code> 으로 변환

## ◆ 2. `StringBuffer`

### ☀ 특징

- Java 초창기부터 존재
- `StringBuilder` 와 거의 동일하지만, 모든 메서드가 `synchronized` 로 보호됨
- 멀티스레드 환경에서 데이터 일관성 보장

## ✅ 예시

```
1 public class Main {
2     public static void main(String[] args) {
3         StringBuffer sb = new StringBuffer("Good");
4         sb.append(" Morning");
5         sb.insert(4, ",");
6         sb.delete(0, 1);
7
8         System.out.println(sb.toString()); // ood, Morning
9     }
10 }
```

## 🔧 사용 시기 요약

상황	추천 클래스
문자열 변경이 거의 없음	<code>String</code>
단일 스레드 환경에서 많은 문자열 조작	<code>StringBuilder</code>
멀티스레드 환경에서 문자열 조작 필요	<code>StringBuffer</code>

## 🧠 성능 테스트 예시

```
1 public class Test {
2     public static void main(String[] args) {
3         long start, end;
4
5         // StringBuilder
6         StringBuilder sb = new StringBuilder();
7         start = System.currentTimeMillis();
8         for (int i = 0; i < 100000; i++) {
9             sb.append("a");
10        }
11        end = System.currentTimeMillis();
12        System.out.println("StringBuilder: " + (end - start) + "ms");
13
14        // String
15        String s = "";
16        start = System.currentTimeMillis();
17        for (int i = 0; i < 100000; i++) {
18            s += "a"; // 비효율적
19        }
20        end = System.currentTimeMillis();
21        System.out.println("String: " + (end - start) + "ms");
22    }
23 }
```

## 🔍 참고: 용량 증가 방식

- `StringBuilder` / `StringBuffer` 는 내부적으로 `char[]` 배열을 사용
- 용량 초과 시, 자동으로 **기하급수적으로 크기 증가** (기본 16 → 34 → 70...)
- 이 과정은 비용이 크기 때문에 초기 용량을 지정하는 것이 좋음:

```
1 | stringBuilder sb = new stringBuilder(10000); // 예상 용량 지정
```



## ✓ 정리

항목	String	StringBuilder	StringBuffer
변경 가능	✗	✓	✓
스레드 안전	✗	✗	✓
성능 (단일 스레드)	느림	빠름	비교적 느림
사용 목적	정적 문자열	빠른 동적 문자열 처리	동기화 필요 시

## Wrapper 클래스: Integer, Double, Boolean 등

Java의 **Wrapper** 클래스는 기본형(primitive type)을 객체(object)처럼 다루기 위해 만들어진 **기본형 포장 클래스**이다. `int`, `double`, `boolean` 같은 기본 타입은 객체가 아니라서 컬렉션 등에 사용할 수 없다. 그래서 Java는 **기본형을 참조형(클래스)**으로 감싸는 **Wrapper** 클래스를 제공한다.

### ◆ 1. Wrapper 클래스란?

기본형	래퍼 클래스
<code>byte</code>	<code>Byte</code>
<code>short</code>	<code>Short</code>
<code>int</code>	<code>Integer</code>
<code>long</code>	<code>Long</code>
<code>float</code>	<code>Float</code>
<code>double</code>	<code>Double</code>
<code>char</code>	<code>Character</code>
<code>boolean</code>	<code>Boolean</code>

### 🔧 사용 이유

- 컬렉션(`List`, `Set`, `Map`)에 기본형을 넣을 수 없음 → 객체로 감싸야 함
- 제네릭에서 기본형 사용 불가 → `List<int>` ✗, `List<Integer>` ✓
- null 표현 가능 (기본형은 null을 표현 못함)

## ◆ 2. 오토박싱(Auto Boxing) & 오토언박싱(Unboxing)

### ✓ 설명

- 오토박싱: 기본형 → Wrapper 클래스로 자동 변환
- 오토언박싱: Wrapper 객체 → 기본형으로 자동 변환

```
1 int i = 10;
2 Integer iobj = i;           // 오토박싱
3 int j = iobj;               // 오토언박싱
```

## ◆ 3. 주요 메서드와 생성 방법

### ✓ Integer, Double, Boolean 예제

```
1 public class Main {
2     public static void main(String[] args) {
3         Integer i = Integer.valueOf("123");    // 문자열 → Integer
4         int x = Integer.parseInt("456");      // 문자열 → int
5
6         Double d = Double.valueOf("3.14");
7         double y = Double.parseDouble("2.71");
8
9         Boolean b1 = Boolean.valueOf("true");
10        boolean b2 = Boolean.parseBoolean("false");
11
12        System.out.println(i + x); // 579
13        System.out.println(d + y); // 5.85
14        System.out.println(b1 && b2); // false
15    }
16 }
```

## ◆ 4. 주요 메서드 정리

### ✓ Integer 클래스

메서드	설명
<code>valueOf(String)</code>	문자열 → <code>Integer</code> 객체
<code>parseInt(String)</code>	문자열 → <code>int</code>
<code>compareTo(Integer)</code>	두 숫자 비교
<code>equals(Object)</code>	값 비교
<code>toString()</code>	문자열 변환
<code>intValue()</code> , <code>doubleValue()</code>	기본형 추출

## ✅ Boolean 클래스

메서드	설명
<code>parseBoolean(String)</code>	문자열 "true" 면 true
<code>valueOf(String)</code>	문자열을 Boolean 으로
<code>booleanValue()</code>	기본형 반환
<code>toString()</code>	"true" or "false"

## 🧠 박싱과 캐싱

- `Integer` 등은 [-128 ~ 127] 범위의 값을 미리 캐싱함.
- 해당 범위 값은 항상 같은 객체 참조를 반환:

```
1 Integer a = 100;
2 Integer b = 100;
3 System.out.println(a == b); // true (같은 객체)
4
5 Integer c = 1000;
6 Integer d = 1000;
7 System.out.println(c == d); // false (다른 객체)
```

`==` 연산자는 참조 비교임. 값 비교는 `.equals()` 사용해야 함.

## 🔍 null 처리 주의

- 오토언박싱 시 `null` 값을 기본형으로 바꾸려 하면 `NullPointerException` 발생:

```
1 Integer i = null;
2 int x = i; // ⚠ 예외 발생
```

## 🔗 Wrapper 클래스의 활용 예시: 컬렉션

```
1 import java.util.*;
2
3 public class Main {
4     public static void main(String[] args) {
5         List<Integer> numbers = new ArrayList<>();
6         numbers.add(10); // 오토박싱
7         numbers.add(20);
8         numbers.add(30);
9
10        for (int n : numbers) { // 오토언박싱
11            System.out.println(n);
12        }
13    }
14 }
```

```

12     }
13     }
14 }

```

## ✓ 정리 요약

항목	설명
정의	기본형을 객체로 다루기 위한 클래스
목적	컬렉션/제네릭 사용, null 표현, 객체 조작
특징	불변(immutable), 오토박싱/언박싱
주의	캐싱 범위 확인, null 언박싱 주의

## Runtime, Enum, Throwable, assert

### ◆ 1. Runtime 클래스

#### ☀ 개요

- JVM 런타임 환경과 상호작용할 수 있게 해주는 클래스
- `Runtime.getRuntime()` 으로 인스턴스를 얻은 후, 시스템 자원을 제어 가능
- 프로세스 실행, 메모리 정보, 종료 처리 등에 사용

#### ✓ 주요 메서드

메서드	설명
<code>exec(String)</code>	외부 프로그램 실행
<code>gc()</code>	가비지 컬렉터 호출 요청
<code>freeMemory()</code>	사용 가능한 메모리 양
<code>totalMemory()</code>	JVM에 할당된 총 메모리
<code>maxMemory()</code>	JVM이 사용할 수 있는 최대 메모리
<code>addShutdownHook(Thread)</code>	JVM 종료 전에 실행할 작업 등록

#### ✓ 예시

```

1 public class Main {
2     public static void main(String[] args) throws Exception {
3         Runtime rt = Runtime.getRuntime();
4
5         System.out.println("Free Memory: " + rt.freeMemory());
6         System.out.println("Total Memory: " + rt.totalMemory());

```

```

7
8      // 외부 명령 실행 (윈도우: notepad, 리눅스: gedit 등)
9      // Process p = rt.exec("notepad");
10
11     // 종료 후 추가
12     rt.addShutdownHook(new Thread(() -> System.out.println("JVM 종료 직전입니다.")));
13
14     // 가비지 컬렉션 유도
15     rt.gc();
16 }
17 }

```

외부 프로그램 실행 시, `Process` 객체를 반환하며 I/O 스트림을 통해 결과를 처리할 수도 있음.

## ◆ 2. Enum 클래스

### ☀ 개요

- Java 5부터 도입된 타입 안전한 상수 집합
- 내부적으로 `java.lang.Enum` 클래스를 암묵적으로 상속
- 열거형은 클래스처럼 동작하며, 필드, 생성자, 메서드 포함 가능

### ✅ 기본 사용

```

1  enum Direction {
2      NORTH, SOUTH, EAST, WEST
3  }
4
5  public class Main {
6      public static void main(String[] args) {
7          Direction d = Direction.NORTH;
8          System.out.println(d);           // NORTH
9          System.out.println(d.name());    // NORTH
10         System.out.println(d.ordinal());  // 0
11     }
12 }

```

### ✅ 고급 예시 (필드 + 생성자 + 메서드)

```

1  enum Planet {
2      MERCURY(3.303e+23, 2.4397e6),
3      EARTH(5.976e+24, 6.37814e6);
4
5      private final double mass;    // in kilograms
6      private final double radius;  // in meters
7
8      Planet(double mass, double radius) {
9          this.mass = mass;
10         this.radius = radius;
11     }

```

```

12
13     double surfaceGravity() {
14         final double G = 6.67300E-11;
15         return G * mass / (radius * radius);
16     }
17 }

```

## ✓ 주요 메서드

메서드	설명
<code>values()</code>	모든 열거 상수 배열 반환
<code>valueOf(String)</code>	문자열로부터 열거 상수 반환
<code>ordinal()</code>	정의 순서 (0부터 시작)
<code>name()</code>	상수 이름 반환

## ◆ 3. Throwable 클래스

### ☀ 개요

- 자바의 모든 에러 및 예외의 최상위 클래스
- 하위 클래스:
  - `Error`: 시스템 오류 (예: `OutOfMemoryError`, `StackOverflowError` 등)
  - `Exception`: 일반적인 예외 (`IOException`, `SQLException` 등)
    - `RuntimeException`: 실행 시점 예외 (`NullPointerException` 등)

## ✓ 계층 구조

```

1  Throwable
2  |— Error
3  |   |— OutOfMemoryError, StackOverflowError 등
4  |— Exception
5  |   |— IOException, SQLException
6  |   |— RuntimeException
7  |       |— NullPointerException, IndexOutOfBoundsException 등

```

## ✓ 주요 메서드

메서드	설명
<code>getMessage()</code>	예외 메시지
<code>printStackTrace()</code>	스택 트레이스 출력
<code>getCause()</code>	예외의 원인 반환

메서드	설명
<code>getStackTrace()</code>	스택 트레이스 배열 반환

### ✓ 예시

```

1 public class Main {
2     public static void main(String[] args) {
3         try {
4             int x = 5 / 0;
5         } catch (ArithmeticException e) {
6             System.out.println("예외 발생: " + e.getMessage());
7             e.printStackTrace(); // 어디서 발생했는지
8         }
9     }
10 }

```

## ◆ 4. `assert` 키워드

### ☀ 개요

- Java 1.4부터 도입
- 디버깅용 런타임 조건 검사 도구
- 조건이 `false` 일 경우 `AssertionError` 를 발생시킴
- 기본적으로 비활성화되어 있음 → `-ea` (enable assertions) 옵션 필요

### ✓ 사용법

```

1 int score = -1;
2 assert score >= 0 : "점수는 음수일 수 없습니다.";

```

### ✓ 실행 방법

```

1 java -ea Main

```

### ✓ 주의사항

- 프로덕션 코드에서는 사용하지 않는 것이 일반적
- 반드시 예외 처리를 대체하지 않도록 주의

### ✓ 요약 표

항목	설명	주 용도
<code>Runtime</code>	JVM 실행 환경 제어	외부 명령 실행, 메모리 확인

항목	설명	주 용도
<code>Enum</code>	타입 안정 상수 집합	상태값, 분기 처리
<code>Throwable</code>	예외 및 오류의 최상위 클래스	예외 처리 및 추적
<code>assert</code>	디버깅용 조건 검사	테스트 또는 개발 시 조건 검증

## 9.2 java.util

### 컬렉션 프레임워크: List, Set, Map, Queue

Java의 컬렉션 프레임워크(Collection Framework)는 데이터를 효율적으로 저장하고, 탐색하고, 조작할 수 있도록 다양한 자료구조(자료형)와 알고리즘을 제공하는 표준 API 집합이다. 이 중에서도 가장 많이 사용되는 핵심 인터페이스는 `List`, `Set`, `Map`, `Queue` 네 가지이고, 각각의 구현체가 있다.

#### ◆ 전체 구조 개요

```

1  java.util.Collection (interface)
2  |— List (순서 0, 중복 허용)
3  |   |— ArrayList
4  |   |— LinkedList
5  |   |— Vector (Legacy)
6  |— Set (순서 X, 중복 X)
7  |   |— HashSet
8  |   |— LinkedHashSet
9  |   |— TreeSet (정렬)
10 |— Queue (선입선출 또는 우선순위)
11 |   |— LinkedList
12 |   |— PriorityQueue
13
14 java.util.Map (interface) - Collection과 별개
15 |— HashMap
16 |— LinkedHashMap
17 |— TreeMap (정렬)

```

#### ■ 1. List 인터페이스

##### ✓ 특징

- 순서 유지 (index 기반)
- 중복 허용
- 동적 배열 혹은 연결 리스트 기반 구현체 존재



### ◆ 주요 구현체

구현체	특징
<code>ArrayList</code>	배열 기반, 빠른 인덱스 접근, 삽입/삭제는 느림
<code>LinkedList</code>	연결 리스트 기반, 삽입/삭제 빠름, 인덱스 접근 느림
<code>Vector</code>	<code>ArrayList</code> 와 유사하지만 동기화됨 (거의 사용 X)

### ◆ 예시

```
1 List<String> list = new ArrayList<>();
2 list.add("Java");
3 list.add("Python");
4 list.add("Java"); // 중복 허용
5 System.out.println(list.get(0)); // Java
```

## ■ 2. Set 인터페이스

### ✓ 특징

- 중복 허용하지 않음
- 순서가 없거나, 삽입 순서/정렬 순서 유지

### ◆ 주요 구현체

구현체	특징
<code>HashSet</code>	해시 기반, 순서 없음
<code>LinkedHashSet</code>	삽입 순서 유지
<code>TreeSet</code>	정렬된 순서 유지 ( <code>Comparable</code> 또는 <code>Comparator</code> 기반)

### ◆ 예시

```
1 Set<String> set = new HashSet<>();
2 set.add("Apple");
3 set.add("Banana");
4 set.add("Apple"); // 무시됨
5
6 for (String fruit : set) {
7     System.out.println(fruit);
8 }
```

### 3. Map 인터페이스 (Collection과 별도)

#### ✓ 특징

- Key-Value 쌍으로 데이터 저장
- Key는 중복 불가, Value는 중복 가능

#### ◆ 주요 구현체

구현체	특징
HashMap	해시 기반, 순서 없음
LinkedHashMap	삽입 순서 유지
TreeMap	정렬된 Key 순서 유지
Hashtable	HashMap 과 유사하지만 동기화됨 (거의 사용 X)

#### ◆ 예시

```
1 Map<String, Integer> map = new HashMap<>();
2 map.put("Apple", 1000);
3 map.put("Banana", 1500);
4 map.put("Apple", 1200); // 기존 값 덮어쓰기
5
6 System.out.println(map.get("Apple")); // 1200
```

### 4. Queue 인터페이스

#### ✓ 특징

- FIFO (선입선출) 방식 자료구조
- 대기열, 작업 큐, 캐시 등에 사용

#### ◆ 주요 구현체

구현체	특징
LinkedList	일반 큐와 덱(double-ended queue) 구현 가능
PriorityQueue	우선순위 큐 (정렬 기준 필요)

## ◆ 예시

```
1 Queue<String> queue = new LinkedList<>();
2 queue.offer("Task1");
3 queue.offer("Task2");
4
5 System.out.println(queue.poll()); // Task1 (삭제)
6 System.out.println(queue.peek()); // Task2 (유지)
```

## 🔧 주요 메서드 정리

메서드	List	Set	Map	Queue
add()	✓	✓	✗ (put 사용)	✓
get(index)	✓	✗	✗	✗
contains()	✓	✓	✓ (Key, Value 따로)	✓
remove()	✓	✓	✓	✓
put(key, value)	✗	✗	✓	✗
offer() / poll()	✗	✗	✗	✓

## 🧠 정리 비교표

항목	List	Set	Map	Queue
순서	유지	없음 or 유지	Key 정렬 or 삽입순 서	일반적으로 유지
중복	허용	✗	Key ✗ / Value ✓	일반적으로 허용
주요 구현 체	ArrayList, LinkedList	HashSet, TreeSet	HashMap, TreeMap	LinkedList, PriorityQueue
사용 예	순서 있는 데이터	유일한 데이터	Key-Value 저장	작업 처리 순서 제어

## 🔍 예시: 모든 컬렉션 순회

```
1 // List
2 List<String> list = List.of("a", "b", "c");
3 for (String s : list) System.out.println(s);
4
5 // Set
6 Set<String> set = Set.of("x", "y", "z");
7 set.forEach(System.out::println);
8
9 // Map
10 Map<String, Integer> map = Map.of("A", 1, "B", 2);
11 map.forEach((k, v) -> System.out.println(k + " : " + v));
```

## ✅ 정리 요약

- `List`: 순서가 있고 중복 가능 → 일반적인 배열 형태
- `Set`: 중복 없이 유일한 데이터 저장
- `Map`: 키로 값을 빠르게 찾는 구조
- `Queue`: 순차적인 작업 처리

## 정렬 및 비교: Comparator, Comparable

Java에서 객체를 정렬하려면 **비교 기준**이 필요하다. 이 비교 기준을 제공하는 방식에는 두 가지가 있다:

1. `Comparable` 인터페이스 - 자연 순서 정의
2. `Comparator` 인터페이스 - 외부에서 별도 기준 정의

이 두 개념은 Java 컬렉션에서 정렬을 구현하는 핵심 메커니즘이다.

### ■ 1. `Comparable<T>` - "자연 순서 정의"

#### ☀️ 특징

- 클래스 자체에 정렬 기준을 정의
- 정렬 기준이 고정
- `compareTo()` 메서드만 구현

#### ✅ 사용 예

```
1 class Person implements Comparable<Person> {
2     String name;
3     int age;
4
5     public Person(String name, int age) {
6         this.name = name;
7         this.age = age;
8     }
9 }
```

```

9
10 // 나이 기준 오름차순 정렬
11 @Override
12 public int compareTo(Person other) {
13     return Integer.compare(this.age, other.age);
14 }
15
16 @Override
17 public String toString() {
18     return name + "(" + age + ")";
19 }
20 }

```

```

1 List<Person> list = new ArrayList<>();
2 list.add(new Person("Alice", 30));
3 list.add(new Person("Bob", 25));
4
5 Collections.sort(list); // 내부 compareTo() 기준 사용
6 System.out.println(list); // [Bob(25), Alice(30)]

```

## ■ 2. Comparator<T> - "외부에서 정렬 기준 제공"

### ☀ 특징

- 정렬 기준을 외부에서 동적으로 정의
- 필요에 따라 다양한 정렬 방법 가능
- `compare()` 메서드 구현

### ✓ 사용 예

```

1 class Person {
2     String name;
3     int age;
4     public Person(String name, int age) {
5         this.name = name;
6         this.age = age;
7     }
8     public String toString() {
9         return name + "(" + age + ")";
10    }
11 }

```

```

1 List<Person> list = new ArrayList<>();
2 list.add(new Person("Alice", 30));
3 list.add(new Person("Bob", 25));
4
5 // 이름 기준 정렬 (Comparator 사용)
6 list.sort(Comparator.comparing(p -> p.name));
7 System.out.println(list); // [Alice(30), Bob(25)]
8
9 // 나이 기준 내림차순 정렬
10 list.sort((p1, p2) -> Integer.compare(p2.age, p1.age));
11 System.out.println(list); // [Alice(30), Bob(25)]

```

## Comparable vs Comparator 비교

항목	Comparable	Comparator
위치	클래스 내부	클래스 외부
메서드	<code>compareTo(T o)</code>	<code>compare(T o1, T o2)</code>
유연성	하나의 기준만 가능	여러 기준 구현 가능
수정 범위	원본 클래스 수정 필요	외부에서 분리 가능
대표 예	<code>String</code> , <code>Integer</code> 등 기본형 객체	다양한 정렬 전략 구현 시

## 실전 정렬 예시

### 객체 여러 기준으로 정렬하기

```

1 // 이름 기준 오름차순
2 Comparator<Person> byName = Comparator.comparing(p -> p.name);
3
4 // 나이 기준 오름차순
5 Comparator<Person> byAge = Comparator.comparingInt(p -> p.age);
6
7 // 나이 기준 내림차순
8 Comparator<Person> byAgeDesc = Comparator.comparingInt(p -> p.age).reversed();
9
10 // 이름 → 나이 순으로 정렬
11 Comparator<Person> complex = byName.thenComparing(byAge);

```

```

1 list.sort(complex); // 다중 기준 정렬 적용

```

## 문자열, 숫자 기본 정렬

```
1 List<String> list = Arrays.asList("banana", "apple", "cherry");
2 Collections.sort(list); // 알파벳 순 (Comparable)
3
4 list.sort(Comparator.reverseOrder()); // 역순
```

```
1 List<Integer> nums = Arrays.asList(5, 1, 7, 3);
2 Collections.sort(nums); // 기본 정렬 (오름차순)
3
4 nums.sort(Comparator.naturalOrder());
5 nums.sort(Comparator.reverseOrder()); // 내림차순
```

## 고급 기능 (Java 8 이후)

기능	예시
메서드 체이닝	<code>Comparator.comparing(...).thenComparing(...)</code>
null 처리	<code>Comparator.nullsFirst(...), nullsLast(...)</code>
역순 정렬	<code>.reversed()</code>
람다식 활용	<code>(a, b) -&gt; a.prop - b.prop</code>

```
1 list.sort(Comparator.nullsLast(Comparator.comparing(p -> p.name)));
```

## 요약 정리

구분	Comparable	Comparator
위치	클래스 내부	외부에서 별도 정의
메서드	<code>compareTo()</code>	<code>compare()</code>
기준	고정된 하나	다수 가능
예시	<code>String, Integer</code>	사용자 정의 정렬

## 유틸리티 클래스: Arrays, Collections, Optional

이 클래스들은 컬렉션과 배열을 효과적으로 다루기 위한 핵심 도구들이다.

# 1. Arrays 클래스

## ☀️ 개요

- `java.util.Arrays`
- 배열을 조작하기 위한 유틸리티 메서드 모음
- `sort()`, `fill()`, `copyOf()`, `equals()` 등

## ✅ 주요 메서드

메서드	설명
<code>sort(array)</code>	배열 정렬
<code>copyOf(array, newLength)</code>	배열 복사 (크기 변경 포함)
<code>fill(array, value)</code>	모든 요소를 동일 값으로 채움
<code>equals(arr1, arr2)</code>	두 배열 내용 비교
<code>toString(array)</code>	문자열로 표현
<code>binarySearch(array, key)</code>	이진 탐색 (정렬 전제 조건)

## ✅ 예시 코드

```
1  import java.util.Arrays;
2
3  public class Main {
4      public static void main(String[] args) {
5          int[] nums = {5, 2, 9, 1};
6
7          Arrays.sort(nums); // [1, 2, 5, 9]
8          System.out.println(Arrays.toString(nums));
9
10         int[] copy = Arrays.copyOf(nums, 6); // [1, 2, 5, 9, 0, 0]
11         System.out.println(Arrays.toString(copy));
12
13         Arrays.fill(copy, 100); // [100, 100, ...]
14         System.out.println(Arrays.toString(copy));
15     }
16 }
```



## 2. Collections 클래스

### ☀ 개요

- `java.util.Collections`
- 컬렉션(List, Set, Map 등)의 조작을 위한 유틸리티 메서드 제공
- 정렬, 최소/최대, 동기화, 불변 컬렉션 생성 등

### ✅ 주요 메서드

메서드	설명
<code>sort(List)</code>	정렬
<code>reverse(List)</code>	순서 뒤집기
<code>shuffle(List)</code>	무작위 섞기
<code>min(Collection) / max(Collection)</code>	최소/최대값
<code>frequency(Collection, obj)</code>	등장 횟수
<code>unmodifiableList(List)</code>	읽기 전용 뷰 반환
<code>synchronizedList(List)</code>	스레드 안전한 리스트
<code>fill(List, val)</code>	값 채우기

### ✅ 예시 코드

```
1 import java.util.*;
2
3 public class Main {
4     public static void main(String[] args) {
5         List<String> fruits = new ArrayList<>(List.of("Banana", "Apple", "Cherry"));
6
7         Collections.sort(fruits);
8         System.out.println(fruits); // [Apple, Banana, Cherry]
9
10        Collections.reverse(fruits);
11        System.out.println(fruits); // [Cherry, Banana, Apple]
12
13        Collections.shuffle(fruits);
14        System.out.println(fruits); // 무작위 순서
15
16        System.out.println(Collections.frequency(fruits, "Apple")); // 횟수
17    }
18 }
```

### 3. Optional<T> 클래스

#### ☀ 개요

- `java.util.Optional`
- **NullPointerException** 방지를 위한 값 컨테이너
- 값이 존재하거나 없을 수 있는 상황에서 명시적 표현 제공

#### ✅ 주요 메서드

메서드	설명
<code>Optional.of(value)</code>	null 아닌 값 감쌈
<code>Optional.ofNullable(value)</code>	null일 수도 있는 값 감쌈
<code>isPresent()</code> / <code>isEmpty()</code>	값 유무 확인
<code>get()</code>	값 반환 (값 없으면 예외 발생)
<code>orElse(default)</code>	기본값 반환
<code>orElseGet(Supplier)</code>	지연 실행 기본값
<code>orElseThrow()</code>	예외 던지기
<code>map(Function)</code>	값이 있을 때 함수 적용
<code>filter(Predicate)</code>	조건 만족 시 유지

#### ✅ 예시 코드

```
1 import java.util.Optional;
2
3 public class Main {
4     public static void main(String[] args) {
5         Optional<String> name = Optional.of("Alice");
6
7         System.out.println(name.isPresent()); // true
8         System.out.println(name.get());       // Alice
9
10        Optional<String> empty = Optional.empty();
11        System.out.println(empty.orElse("No Name")); // No Name
12
13        String result = name
14            .filter(n -> n.startsWith("A"))
15            .map(String::toUpperCase)
16            .orElse("DEFAULT");
17        System.out.println(result); // ALICE
18    }
```

## 🧠 Optional 실전 팁

상황	추천 방식
값이 무조건 존재	<code>Optional.of(value)</code>
값이 null일 수도 있음	<code>Optional.ofNullable(value)</code>
값이 없을 수 있음 → 기본값 사용	<code>orElse(default)</code>
값이 없을 수 있음 → 예외 발생	<code>orElseThrow()</code>
값이 있을 때만 가공	<code>map()</code> / <code>flatMap()</code>
값 조건에 따라 필터링	<code>filter()</code>

## ✅ 총정리 요약

클래스	주용도	대표 기능
<code>Arrays</code>	배열 조작	정렬, 복사, 비교, 채우기
<code>Collections</code>	컬렉션 조작	정렬, 섞기, 최소/최대, 동기화
<code>Optional&lt;T&gt;</code>	Null-safe 값 처리	존재 여부 확인, 기본값 대체, 조건 필터링

## Scanner, Properties, UUID, Random 등

### ■ 1. Scanner 클래스

#### ☀️ 개요

- 입력 처리용 클래스
- 키보드 입력, 파일, 문자열 등 다양한 입력 소스를 다룰 수 있음
- `java.util.Scanner`

#### ✅ 주요 생성자

```

1 new Scanner(System.in);           // 콘솔 입력
2 new Scanner(new File("data.txt")); // 파일 입력
3 new Scanner("1,2,3").useDelimiter(","); // 구분자 지정

```

## ✅ 주요 메서드

메서드	설명
<code>next()</code>	다음 토큰 반환
<code>nextLine()</code>	한 줄 전체 입력
<code>nextInt()</code> , <code>nextDouble()</code> 등	타입별 입력
<code>hasNext()</code>	다음 입력 존재 여부
<code>useDelimiter(regex)</code>	입력 구분자 변경

## ◆ 예시

```
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
7         System.out.print("이름 입력: ");
8         String name = sc.nextLine();
9
10        System.out.print("나이 입력: ");
11        int age = sc.nextInt();
12
13        System.out.println("안녕하세요 " + name + "(" + age + ")");
14    }
15 }
```

## ■ 2. Properties 클래스

### ☀ 개요

- 설정 파일을 키-값 쌍으로 관리하는 용도
- `.properties` 파일은 `key=value` 형식
- `java.util.Properties` 는 `Map<Object, Object>` 를 상속

## ✅ 주요 메서드

메서드	설명
<code>load(InputStream)</code>	설정 파일 읽기
<code>store(OutputStream, comments)</code>	설정 파일 쓰기
<code>getProperty(key)</code>	값 읽기

메서드	설명
<code>setProperty(key, value)</code>	값 설정

### ◆ 예시 (읽기)

```

1 import java.util.Properties;
2 import java.io.FileInputStream;
3
4 public class Main {
5     public static void main(String[] args) throws Exception {
6         Properties prop = new Properties();
7         prop.load(new FileInputStream("config.properties"));
8
9         String host = prop.getProperty("db.host");
10        String user = prop.getProperty("db.user");
11        System.out.println("Host: " + host + ", User: " + user);
12    }
13 }

```

### ◆ 예시 (쓰기)

```

1 Properties prop = new Properties();
2 prop.setProperty("db.host", "localhost");
3 prop.setProperty("db.port", "3306");
4
5 prop.store(new FileOutputStream("config.properties"), "DB Config");

```

## ■ 3. UUID 클래스

### ☀ 개요

- 범용 고유 식별자(Universally Unique Identifier) 생성
- 충돌 가능성이 극히 낮음
- `java.util.UUID`

### ✅ 주요 메서드

메서드	설명
<code>UUID.randomUUID()</code>	임의 UUID 생성
<code>UUID.fromString(String)</code>	문자열 → UUID 객체
<code>toString()</code>	UUID 문자열 반환

## ◆ 예시

```
1 import java.util.UUID;
2
3 public class Main {
4     public static void main(String[] args) {
5         UUID id = UUID.randomUUID();
6         System.out.println(id.toString()); // 예: f81d4fae-7dec-11d0-a765-00a0c91e6bf6
7     }
8 }
```

보통 고유 파일 이름, 세션 토큰, 객체 식별자 등에 사용함.

## ■ 4. Random 클래스

### ☀ 개요

- 의사 난수(Pseudo-random number) 생성
- `java.util.Random` 은 다양한 타입의 난수를 제공
- `ThreadLocalRandom`, `SecureRandom` 등 대안도 있음

### ✅ 주요 메서드

메서드	설명
<code>nextInt()</code>	32비트 정수 난수
<code>nextInt(bound)</code>	0 ~ bound-1 범위
<code>nextDouble()</code>	0.0 ~ 1.0 실수
<code>nextBoolean()</code>	true 또는 false
<code>setSeed(long)</code>	시드값 설정 (재현성 보장)

## ◆ 예시

```
1 import java.util.Random;
2
3 public class Main {
4     public static void main(String[] args) {
5         Random rand = new Random();
6
7         int a = rand.nextInt();           // 무제한 정수
8         int b = rand.nextInt(100);        // 0~99
9         double d = rand.nextDouble();     // 0.0~1.0
10        boolean f = rand.nextBoolean();    // true or false
11
12        System.out.println(a + ", " + b + ", " + d + ", " + f);
13    }
```

## 💡 실전 응용 요약

클래스	사용 예시	실무 활용
<code>Scanner</code>	콘솔 입력, 파일 입력	사용자 입력, 로그 분석
<code>Properties</code>	설정파일 읽기/쓰기	DB 접속 설정, 환경 구성
<code>UUID</code>	고유 ID 생성	토큰, 식별자, 파일 이름
<code>Random</code>	난수 생성	로또 번호, 무작위 추출, 게임 확률

## 9.3 java.time (Java 8+ 날짜/시간 API)

### LocalDate, LocalTime, LocalDateTime

이 API는 **java.time 패키지**에 있고, 기존의 `Date`/`Calendar`의 문제점(불변성 부족, 혼란스러운 월 인덱스, 스레드 안전성 등)을 대체하기 위해 만들어졌다.

#### ■ 개요: java.time API란?

- Java 8부터 추가된 새로운 날짜/시간 처리 API
- 불변(**immutable**) 객체 기반
- `LocalDate`, `LocalTime`, `LocalDateTime`, `ZonedDateTime`, `Instant` 등 다양한 클래스 제공
- 시간대와 상관없이 **순수한 날짜/시간 표현** 가능

#### ✅ 1. LocalDate

##### 📌 개요

- **날짜만 표현**: 연/월/일 (YYYY-MM-DD)
- 시간, 시간대 정보 없음

##### ☀️ 주요 메서드

메서드	설명
<code>now()</code>	현재 날짜
<code>of(y, m, d)</code>	특정 날짜 생성
<code>getYear()</code> , <code>getMonth()</code> , <code>getDayOfMonth()</code>	날짜 정보 추출
<code>plusDays(n)</code> , <code>minusWeeks(n)</code>	날짜 연산

메서드	설명
<code>isBefore()</code> , <code>isAfter()</code> , <code>equals()</code>	날짜 비교

## ◆ 예시

```

1  import java.time.LocalDate;
2
3  public class Main {
4      public static void main(String[] args) {
5          LocalDate today = LocalDate.now();           // 오늘 날짜
6          LocalDate birth = LocalDate.of(1995, 10, 12); // 특정 날짜
7
8          System.out.println("오늘: " + today);
9          System.out.println("내 생일: " + birth);
10         System.out.println("내 생일까지 " + today.until(birth).getDays() + "일 남음");
11
12         LocalDate future = today.plusDays(10);
13         System.out.println("10일 후: " + future);
14     }
15 }

```

## ✓ 2. `LocalTime`

### 📌 개요

- 시간만 표현: 시/분/초/나노초 (HH:MM:SS.nnnnnnnnnn)
- 날짜, 시간대 없음

### ☀️ 주요 메서드

메서드	설명
<code>now()</code>	현재 시간
<code>of(h, m, s)</code>	특정 시간 생성
<code>getHour()</code> , <code>getMinute()</code> , <code>getSecond()</code>	시각 정보 추출
<code>plusHours(n)</code> , <code>minusMinutes(n)</code>	시간 연산
<code>isBefore()</code> , <code>isAfter()</code>	시간 비교



## ◆ 예시

```
1 import java.time.LocalDateTime;
2
3 public class Main {
4     public static void main(String[] args) {
5         LocalDateTime now = LocalDateTime.now();
6         LocalDateTime classStart = LocalDateTime.of(9, 0);
7
8         System.out.println("현재 시간: " + now);
9         System.out.println("수업 시작 시간: " + classStart);
10
11        System.out.println("수업 시작 전인가? " + now.isBefore(classStart));
12    }
13 }
```

## ✓ 3. LocalDateTime

### 📌 개요

- 날짜 + 시간 (시간대 없음)
- `LocalDate` 와 `LocalTime` 을 결합한 형태

### ☀ 주요 메서드

메서드	설명
<code>now()</code>	현재 날짜+시간
<code>of(date, time)</code>	날짜와 시간 조합
<code>toLocalDate()</code> , <code>toLocalTime()</code>	분리 가능
<code>plusDays(n)</code> , <code>minusHours(n)</code>	날짜 및 시간 연산
<code>format(formatter)</code>	포맷 출력 (→ <code>DateTimeFormatter</code> )

## ◆ 예시

```
1 import java.time.LocalDateTime;
2
3 public class Main {
4     public static void main(String[] args) {
5         LocalDateTime now = LocalDateTime.now();
6         LocalDateTime meeting = LocalDateTime.of(2025, 6, 1, 14, 30);
7
8         System.out.println("현재 시각: " + now);
9         System.out.println("회의 시각: " + meeting);
10        System.out.println("지났는가? " + now.isAfter(meeting));
11    }
```

```
12 // 날짜, 시간 각각 추출
13 System.out.println("날짜만: " + meeting.toLocalDate());
14 System.out.println("시간만: " + meeting.toLocalTime());
15 }
16 }
```

## 🧠 비교 정리표

클래스	포함 정보	사용 예
<code>LocalDate</code>	년/월/일	생일, 기념일, 마감일
<code>LocalTime</code>	시/분/초/나노초	알람, 일정 시작 시간
<code>LocalDateTime</code>	날짜 + 시간	회의 시각, 예약 시간

## ✂ 보너스: 날짜 계산 메서드들

```
1 LocalDate date = LocalDate.of(2024, 1, 1);
2 date = date.plusDays(10);           // 10일 후
3 date = date.minusWeeks(2);          // 2주 전
4 date = date.withDayOfMonth(15);     // 날짜를 15일로 설정
```

## ✂ 날짜 비교 메서드들

```
1 if (date1.isAfter(date2)) { ... }
2 if (date1.isBefore(date2)) { ... }
3 if (date1.equals(date2)) { ... }
```

## ✅ 요약 정리

클래스	특징	포함 정보	시간대
<code>LocalDate</code>	날짜만	연, 월, 일	✗
<code>LocalTime</code>	시간만	시, 분, 초	✗
<code>LocalDateTime</code>	날짜 + 시간	연, 월, 일, 시, 분, 초	✗

💡 시간대 정보가 필요한 경우 `ZonedDateTime`, `OffsetDateTime` 사용

# ZonedDateTime, Instant

Java 8의 `java.time` 패키지에서 시간대와 UTC 타임스탬프를 다루기 위한 **핵심 클래스**이다.

- ✓ `ZonedDateTime`
- ✓ `Instant`

이 둘은 전 세계적인 시간 개념(표준시, 시간대, 타임스탬프)을 정밀하게 다루기 위해 만들어졌고, `LocalDateTime`과는 다르게 **시간대(time zone)** 및 **오프셋(offset)**을 명시적으로 포함하고 있다.

## 1. ZonedDateTime

### ☀ 개요

- 날짜와 시간 + **시간대(TimeZone)**를 포함
- 예: `2025-05-29T20:45:00+09:00[Asia/Seoul]`
- 표현력 가장 풍부한 날짜/시간 클래스

### ✓ 주요 메서드

메서드	설명
<code>now()</code>	현재 시간대 기준 <code>ZonedDateTime</code>
<code>of(...)</code>	날짜/시간/Zone을 조합
<code>withZoneSameInstant(zone)</code>	같은 순간을 다른 시간대로 변환
<code>withZoneSameLocal(zone)</code>	같은 지역 시간으로 zone만 변경
<code>toLocalDateTime()</code>	시간대 제거
<code>getZone()</code> , <code>getOffset()</code>	시간대, 오프셋 정보 추출

### ✓ 예시 코드

```
1 import java.time.*;
2 import java.time.format.DateTimeFormatter;
3
4 public class Main {
5     public static void main(String[] args) {
6         ZonedDateTime seoulTime = ZonedDateTime.now(ZoneId.of("Asia/Seoul"));
7         ZonedDateTime utcTime = ZonedDateTime.now(ZoneOffset.UTC);
8
9         System.out.println("서울 시간: " + seoulTime);
10        System.out.println("UTC 시간: " + utcTime);
11
12        // 동일 시각을 다른 시간대로 변환
13        ZonedDateTime nyTime =
            seoulTime.withZoneSameInstant(ZoneId.of("America/New_York"));
```

```
14         System.out.println("뉴욕 시간 (같은 순간): " + nyTime);
15     }
16 }
```

## ✅ 시간대 리스트 조회

```
1 ZoneId.getAvailableZoneIds().forEach(System.out::println); // 가능한 시간대 ID 출력
```

## ■ 2. Instant

### ☀ 개요

- 기준 시점(UTC 1970-01-01 00:00:00) 으로부터 경과한 초/나노초를 표현하는 클래스
- UTC 타임스탬프에 대응 (epoch 기반)
- 시간대 없음, 순수한 "순간"만을 표현
- DB 저장이나 API timestamp 전달에 최적화

## ✅ 주요 메서드

메서드	설명
<code>now()</code>	현재 시각(UTC 기준)
<code>ofEpochSecond(s)</code>	에포크 초로부터 Instant 생성
<code>toEpochMilli()</code>	밀리초 반환 (long)
<code>plusSeconds()</code> , <code>minusNanos()</code>	시점 계산
<code>atZone(ZoneId)</code>	ZonedDateTime으로 변환
<code>compareTo()</code> , <code>isBefore()</code> , <code>isAfter()</code>	비교 연산

## ✅ 예시 코드

```
1 import java.time.*;
2
3 public class Main {
4     public static void main(String[] args) {
5         Instant now = Instant.now(); // UTC 기준 시점
6         System.out.println("현재 Instant: " + now);
7
8         long epochMillis = now.toEpochMilli(); // 1970 이후 밀리초
9         System.out.println("Epoch millis: " + epochMillis);
10
11         ZonedDateTime zoned = now.atZone(ZoneId.of("Asia/Seoul"));
```

```
12         System.out.println("Instant → ZonedDateTime: " + zoned);
13     }
14 }
```

## 

항목	ZonedDateTime	Instant
포함 정보	날짜, 시간, 시간대, 오프셋	UTC 기준 타임스탬프
시간대 포함	✅ 포함 (ZoneId, ZoneOffset)	❌ 없음
사용 목적	사용자 시각 표현, 시간대 변환	시스템/네트워크 시간 기록
변환 대상	LocalDateTime, Instant	ZonedDateTime, LocalDateTime + ZoneId

## 

```
1 // LocalDateTime → ZonedDateTime → Instant
2 LocalDateTime ldt = LocalDateTime.of(2025, 1, 1, 12, 0);
3 ZonedDateTime zoned = ldt.atZone(ZoneId.of("Asia/Seoul"));
4 Instant instant = zoned.toInstant();
5
6 System.out.println("UTC Timestamp: " + instant);
7
8 // Instant → ZonedDateTime → LocalDateTime
9 ZonedDateTime newZoned = instant.atZone(ZoneId.of("Europe/London"));
10 LocalDateTime ldtInLondon = newZoned.toLocalDateTime();
```

## 

클래스	목적	예
ZonedDateTime	날짜 + 시간 + 시간대	예약 시스템, 다국가 서비스
Instant	UTC 시점 (밀리초/나노초)	타임스탬프, 로그 기록

## Duration, Period

- ✅ Duration
- ✅ Period

이 두 클래스는 각각 **시간 단위 간격**, **날짜 단위 간격**을 나타내는 데 사용된다.

## 1. Duration

### ☀ 개요

- 시/분/초/나노초 단위로 시간 차이를 나타냄
- `Instant`, `LocalTime`, `LocalDateTime` 등의 시간 API와 함께 사용
- 단위: 초(Seconds), 나노초(Nanos)

### ✅ 주요 메서드

메서드	설명
<code>Duration.between(start, end)</code>	두 시점 간의 간격 생성
<code>toHours()</code> , <code>toMinutes()</code> , <code>toMillis()</code>	시간 단위 변환
<code>plusXxx()</code> , <code>minusXxx()</code>	간격 연산
<code>isZero()</code> , <code>isNegative()</code>	상태 체크

### ◆ 예시: Duration 사용

```
1 import java.time.*;
2
3 public class Main {
4     public static void main(String[] args) {
5         LocalTime start = LocalTime.of(9, 0);
6         LocalTime end = LocalTime.of(14, 30);
7
8         Duration duration = Duration.between(start, end);
9         System.out.println("총 시간(분): " + duration.toMinutes()); // 330분
10        System.out.println("총 시간(초): " + duration.getSeconds()); // 19800초
11    }
12 }
```

### ◆ 예시: Instant 간격

```
1 Instant before = Instant.now();
2 // 작업 수행...
3 Instant after = Instant.now();
4
5 Duration gap = Duration.between(before, after);
6 System.out.println("작업 소요: " + gap.toMillis() + "ms");
```

## 2. Period

### ☀ 개요

- 년/월/일 단위로 날짜 간격을 나타냄
- `LocalDate` 전용
- 단위: 연(Years), 월(Months), 일(Days)

### ✅ 주요 메서드

메서드	설명
<code>Period.between(start, end)</code>	두 날짜 간격 계산
<code>getYears()</code> , <code>getMonths()</code> , <code>getDays()</code>	기간 정보 추출
<code>plusXxx()</code> , <code>minusXxx()</code>	기간 연산
<code>isZero()</code> , <code>isNegative()</code>	상태 체크

### ◆ 예시: 생일 계산

```
1 import java.time.*;
2
3 public class Main {
4     public static void main(String[] args) {
5         LocalDate birth = LocalDate.of(1995, 5, 29);
6         LocalDate today = LocalDate.now();
7
8         Period age = Period.between(birth, today);
9
10        System.out.println("나이: " + age.getYears() + "세 " +
11                            age.getMonths() + "개월 " +
12                            age.getDays() + "일");
13    }
14 }
```

### 🔄 비교: Duration vs Period

항목	Duration	Period
단위	시간 기반 (초, 나노초)	날짜 기반 (년, 월, 일)
대상	<code>Instant</code> , <code>LocalTime</code> , <code>LocalDateTime</code>	<code>LocalDate</code>
정확도	밀리초/나노초까지	년/월/일까지만
대표 사용	타이머, 작업 수행 시간	생일, 만기일 계산

## 기타 생성 방법

```
1 Duration threeHours = Duration.ofHours(3);
2 Duration tenMinutes = Duration.ofMinutes(10);
3
4 Period oneWeek = Period.ofWeeks(1);
5 Period threeMonths = Period.of(0, 3, 0); // 3개월
```

## 정리 요약

클래스	사용 대상	표현 단위	사용 예
<code>Duration</code>	시간/시각 ( <code>Instant</code> , <code>LocalTime</code> )	초, 나노초	타이머, 실행 시간
<code>Period</code>	날짜 ( <code>LocalDate</code> )	년, 월, 일	나이, 대출 만기, 이벤트 D-Day

## DateTimeFormatter, TemporalAdjusters

- ✓ `DateTimeFormatter`: 날짜/시간의 문자열 포맷과 파싱
- ✓ `TemporalAdjusters`: 날짜의 정교한 조정(보정) 연산

### 1. DateTimeFormatter

#### ☀ 개요

- `java.time.format.DateTimeFormatter`
- 날짜/시간을 문자열로 출력하거나, 문자열을 날짜/시간 객체로 파싱
- `SimpleDateFormat` 보다 안전하고 직관적임
- 스레드 안전(thread-safe)

## ✓ 사용 방법

메서드	설명
<code>DateTimeFormatter.ofPattern("yyyy-MM-dd")</code>	커스텀 포맷 생성
<code>format(TemporalAccessor)</code>	날짜/시간 → 문자열
<code>parse(CharSequence)</code>	문자열 → 날짜/시간
<code>DateTimeFormatter.ISO_LOCAL_DATE</code>	사전 정의된 포맷 사용



## 📌 패턴 문자열 예시

패턴	의미	예시
yyyy	연도	2025
MM	월 (2자리)	05
dd	일	29
HH	시 (24시간)	13
mm	분	07
ss	초	45
a	오전/오후	AM, PM
E	요일	Tue
yyyy-MM-dd HH:mm:ss	전체	2025-05-29 14:30:00

### ◆ 예시: 날짜를 포맷하기

```
1 import java.time.*;
2 import java.time.format.*;
3
4 public class Main {
5     public static void main(String[] args) {
6         LocalDateTime now = LocalDateTime.now();
7         DateTimeFormatter fmt = DateTimeFormatter.ofPattern("yyyy년 MM월 dd일
8         HH:mm:ss");
9
10        String formatted = now.format(fmt);
11        System.out.println("포맷된 시간: " + formatted); // 2025년 05월 29일 14:40:23
12    }
13 }
```

### ◆ 예시: 문자열 파싱하기

```
1 String input = "2025-12-01 10:30";
2 DateTimeFormatter parser = DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm");
3 LocalDateTime parsed = LocalDateTime.parse(input, parser);
4 System.out.println("파싱된 객체: " + parsed);
```

## 2. TemporalAdjusters

### ☀ 개요

- `java.time.temporal.TemporalAdjusters`
- `LocalDate`, `LocalDateTime` 등 날짜 객체에 대해 **보정(조정)** 연산을 수행
- "이번 달의 첫 날", "다음 금요일", "마지막 일요일" 같은 **자연어적 날짜 계산** 가능

### ✅ 주요 메서드

메서드	설명
<code>firstDayOfMonth()</code>	해당 달의 첫째 날
<code>lastDayOfMonth()</code>	해당 달의 마지막 날
<code>firstInMonth(DayOfWeek)</code>	해당 요일의 첫 번째
<code>lastInMonth(DayOfWeek)</code>	해당 요일의 마지막
<code>next(DayOfWeek)</code>	다음 해당 요일
<code>previous(DayOfWeek)</code>	이전 해당 요일
<code>nextOrSame(DayOfWeek)</code>	오늘이 그 요일이면 유지, 아니면 다음
<code>previousOrSame(DayOfWeek)</code>	오늘이 그 요일이면 유지, 아니면 이전

### ◆ 예시: 월급날, 금요일 찾기

```
1 import java.time.*;
2 import java.time.temporal.*;
3
4 public class Main {
5     public static void main(String[] args) {
6         LocalDate today = LocalDate.now();
7
8         LocalDate firstDay = today.with(TemporalAdjusters.firstDayOfMonth());
9         LocalDate lastDay = today.with(TemporalAdjusters.lastDayOfMonth());
10
11         LocalDate nextFriday = today.with(TemporalAdjusters.next(DayOfWeek.FRIDAY));
12         LocalDate lastSunday =
13             today.with(TemporalAdjusters.lastInMonth(DayOfWeek.SUNDAY));
14
15         System.out.println("이번 달 첫날: " + firstDay);
16         System.out.println("이번 달 마지막날: " + lastDay);
17         System.out.println("다음 금요일: " + nextFriday);
18         System.out.println("이번 달 마지막 일요일: " + lastSunday);
19     }
20 }
```

## 💡 with() 메서드와 함께 사용

```
1 | LocalDate payday = LocalDate.now().with(TemporalAdjusters.lastDayOfMonth());
```

이렇게 `.with()` 에 `adjuster`를 넘기면 해당 날짜 객체를 보정한 새로운 객체가 반환돼.

## ✅ 요약 비교

항목	<code>DateTimeFormatter</code>	<code>TemporalAdjusters</code>
목적	포맷 및 파싱	날짜 보정 (자연어 계산)
주요 대상	<code>LocalDate</code> , <code>LocalDateTime</code> , <code>ZonedDateTime</code>	<code>LocalDate</code> , <code>LocalDateTime</code>
주요 용도	문자열 변환	첫날, 마지막날, 특정 요일 등 계산
핵심 메서드	<code>format()</code> , <code>parse()</code>	<code>with(...)</code> + 정적 메서드

## 9.4 난수 생성

### Random, SecureRandom, ThreadLocalRandom

✅ `Random`

✅ `SecureRandom`

✅ `ThreadLocalRandom`

이 세 클래스는 모두 난수 생성 기능을 제공하지만, 목적과 내부 구현 방식, 성능이 매우 다르다.

#### ■ 1. Random

##### ☀️ 개요

- `java.util.Random`
- 가장 일반적인 의사 난수 생성기 (Pseudo-random generator)
- `Linear Congruential Generator` 알고리즘 사용
- 단일 스레드에서는 적절하지만, 멀티스레드 환경에서는 경쟁 발생

## ✅ 주요 메서드

메서드	설명
<code>nextInt()</code>	임의의 int 생성
<code>nextInt(bound)</code>	0 이상 bound 미만의 int
<code>nextDouble()</code>	0.0 ~ 1.0 실수

메서드	설명
<code>nextBoolean()</code>	true 또는 false
<code>setSeed(long)</code>	시드값 설정 (결과 재현 가능)

## ◆ 예시

```

1 import java.util.Random;
2
3 Random rand = new Random();
4 int num = rand.nextInt(100);    // 0~99
5 double d = rand.nextDouble();  // 0.0 ~ 1.0
6 boolean b = rand.nextBoolean(); // true or false

```

## ■ 2. SecureRandom

### ☀ 개요

- `java.security.SecureRandom`
- 암호학적으로 안전한 난수 생성기
- 내부적으로 운영체제의 난수 소스 (ex. `/dev/urandom`) 를 사용
- 보안 토큰, 암호 키, UUID 생성에 사용됨
- 느리지만 강력함

### ✅ 주요 메서드

메서드	설명
<code>nextInt()</code> , <code>nextBytes(byte[])</code>	난수 생성
<code>setSeed(byte[])</code>	고유한 시드 설정
<code>generateSeed(int)</code>	시드 생성

## ◆ 예시

```

1 import java.security.SecureRandom;
2
3 SecureRandom secureRand = new SecureRandom();
4 int num = secureRand.nextInt(1000); // 0~999
5 byte[] bytes = new byte[16];
6 secureRand.nextBytes(bytes);        // 랜덤 바이트 배열 생성

```

### 3. ThreadLocalRandom

#### ☀️ 개요

- `java.util.concurrent.ThreadLocalRandom`
- Java 7부터 도입
- 멀티스레드 환경에 최적화된 난수 생성기
- 각 스레드마다 독립적인 시드를 가짐 → 경쟁 없음 → **가장 빠름**
- 직접 생성자 호출 불가, `ThreadLocalRandom.current()` 로 사용

#### ✅ 주요 메서드

메서드	설명
<code>current().nextInt()</code>	난수 생성 (정적 메서드)
<code>nextInt(origin, bound)</code>	특정 범위 난수
<code>nextDouble(origin, bound)</code>	실수 범위 난수

#### ◆ 예시

```
1 import java.util.concurrent.ThreadLocalRandom;
2
3 int n = ThreadLocalRandom.current().nextInt(10, 20); // 10 이상 20 미만
4 double d = ThreadLocalRandom.current().nextDouble(0.0, 5.0);
```

#### 🔄 비교 요약표

항목	<code>Random</code>	<code>SecureRandom</code>	<code>ThreadLocalRandom</code>
위치	<code>java.util</code>	<code>java.security</code>	<code>java.util.concurrent</code>
성능	중간	느림	가장 빠름
스레드 안전성	❌ (공유 상태)	✅ (자체 안전)	✅ (Thread-local)
보안성	낮음	매우 높음	낮음
시드 제어	가능	가능	불가 (자동 관리)
대표 용도	일반 난수	보안 키, 토큰, UUID	멀티스레드 난수, 병렬 알고리즘

## 실전 적용 예시

### 보안용 토큰

```
1 SecureRandom sr = new SecureRandom();
2 byte[] token = new byte[32];
3 sr.nextBytes(token);
4 // → Base64 또는 Hex로 인코딩해 사용
```

### 멀티스레드 로또 번호

```
1 IntStream.generate(() -> ThreadLocalRandom.current().nextInt(1, 46))
2     .distinct()
3     .limit(6)
4     .forEach(System.out::println);
```

## 정리 요약

쓰임새	추천 클래스
단순 테스트, 단일 스레드	<code>Random</code>
보안, 암호학, 세션 토큰	<code>SecureRandom</code>
병렬 처리, 고속 성능	<code>ThreadLocalRandom</code>

## UUID (고유 식별자 생성)

### `UUID (java.util.UUID)`

이 클래스는 데이터베이스 키, 파일 이름, 세션 토큰, 사용자 ID 등 **충돌 없는 고유한 값이 필요한 모든 상황**에서 사용된다.

## 1. UUID란?

### 정의

- **Universally Unique Identifier** (범용 고유 식별자)
- **128비트 길이**의 숫자를 36자 문자열로 표현
- 형식: `"xxxxxxxx-xxxx-Mxxx-Nxxx-xxxxxxxxxxxx"`

예:

```
1 f81d4fae-7dec-11d0-a765-00a0c91e6bf6
```

## ✖ UUID 형식 설명

구성	길이	설명
8자리	32비트	시간/랜덤 정보
4자리	16비트	시간/랜덤 정보
4자리 (Mxxx)	16비트	UUID 버전 (M)
4자리 (Nxxx)	16비트	Variant 정보
12자리	48비트	노드 정보 (MAC or 난수)

## ■ 2. UUID 버전 종류

버전	설명	Java 지원 여부
1	시간 + MAC 주소 기반	✗ (Java 기본 미지원)
3	MD5 해시 기반	✗
4	랜덤 기반 (SecureRandom)	✓ 기본 지원
5	SHA-1 해시 기반	✗

✓ Java의 `UUID.randomUUID()` 는 버전 4 (랜덤 기반) 을 생성해.

## ■ 3. UUID 생성 방법

### ✓ 랜덤 UUID (버전 4)

```
1 import java.util.UUID;
2
3 UUID id = UUID.randomUUID();
4 System.out.println(id.toString());
```

예시 출력:

```
1 3e1d8fc1-b6dc-48e1-b0b4-c8c14b9d7e58
```

### ✓ 문자열 → UUID 객체

```
1 UUID fromStr = UUID.fromString("3e1d8fc1-b6dc-48e1-b0b4-c8c14b9d7e58");
2 System.out.println(fromStr.version()); // 4
3 System.out.println(fromStr.variant()); // 2 (RFC 4122)
```

## 주요 메서드

메서드	설명
<code>randomUUID()</code>	난수 기반 UUID 생성
<code>fromString(String)</code>	문자열로부터 UUID 생성
<code>toString()</code>	UUID를 문자열로 반환
<code>version()</code>	UUID 버전 (ex. 4)
<code>variant()</code>	UUID variant 정보
<code>getMostSignificantBits()</code>	상위 64비트
<code>getLeastSignificantBits()</code>	하위 64비트

## 예시: 고유 ID 리스트 만들기

```
1 List<String> ids = new ArrayList<>();
2 for (int i = 0; i < 5; i++) {
3     ids.add(UUID.randomUUID().toString());
4 }
5 ids.forEach(System.out::println);
```

## SecureRandom과 함께 사용하기

Java의 `UUID.randomUUID()` 는 내부적으로 `SecureRandom` 을 사용해서 **충돌 가능성이** 극히 낮아.  
하지만 **직접 제어된** `SecureRandom` 으로 생성하고 싶다면 다음과 같이 할 수 있어:

```
1 import java.util.UUID;
2 import java.security.SecureRandom;
3
4 SecureRandom sr = new SecureRandom();
5 byte[] bytes = new byte[16];
6 sr.nextBytes(bytes);
7
8 long msb = 0;
9 long lsb = 0;
10 for (int i = 0; i < 8; i++) msb = (msb << 8) | (bytes[i] & 0xff);
11 for (int i = 8; i < 16; i++) lsb = (lsb << 8) | (bytes[i] & 0xff);
12
13 // 버전 4 마스크 적용
14 msb &= 0xffffffffffff0fffL;
15 msb |= 0x0000000000004000L;
16 lsb &= 0x3fffffffffffffffL;
17 lsb |= 0x8000000000000000L;
18
```



```
19 UUID secureUUID = new UUID(msb, lsb);
20 System.out.println(secureUUID);
```

## 💡 실전 활용 예시

활용 사례	설명
파일 이름	report-<UUID>.pdf
RESTful ID	/api/users/3e1d8fc1-...
세션/토큰	쿠키 값, 인증 토큰 등
분산 시스템	전역적으로 충돌 없는 ID 생성 가능

## ⚠ 주의사항

- UUID는 자체적으로 정렬되지 않음  
→ DB 인덱싱 시 비효율적일 수 있음 (→ ULID, KSUID 등 고려)
- `UUID.randomUUID()` 는 보안적으로 충분히 안전하지만, **진짜 암호 키**로 쓰는 건 권장되지 않음
- UUID 문자열은 **36자** ( 32자리 + 4개의 '-' ) → URL 파라미터로 사용 시 인코딩 필요

## ✅ 정리 요약

항목	UUID
정의	전역 고유 식별자 (128비트)
Java 기본 지원	✅ 버전 4 (랜덤 기반)
문자열 형식	xxxxxxxx-xxxx-Mxxx-Nxxx-xxxxxxxxxxxx
충돌 가능성	이론적으로 매우 낮음
실무 활용	사용자 ID, 파일 이름, API 식별자, 로그 트래킹

## 9.5 래퍼 클래스 및 변환

### Boxing/Unboxing

- ✅ **Boxing** (박싱)
- ✅ **Unboxing** (언박싱)

Java의 기본형(primitive type)과 참조형(wrapper class) 사이의 자동 변환 개념이다.

이 개념은 Java의 제네릭, 컬렉션, 연산에서 자주 사용되며 **성능과 오류 가능성**에도 영향을 주는 중요한 주제이다.

## 1. Boxing / Unboxing이란?

개념	설명	예시
Boxing	기본형 → 참조형으로 자동 변환	<code>int → Integer</code>
Unboxing	참조형 → 기본형으로 자동 변환	<code>Integer → int</code>

✅ Java 5부터 자동 변환(오토박싱/언박싱) 지원됨.

### 🔧 주요 Wrapper 클래스 대응표

기본형	Wrapper 클래스
<code>byte</code>	<code>Byte</code>
<code>short</code>	<code>Short</code>
<code>int</code>	<code>Integer</code>
<code>long</code>	<code>Long</code>
<code>float</code>	<code>Float</code>
<code>double</code>	<code>Double</code>
<code>char</code>	<code>Character</code>
<code>boolean</code>	<code>Boolean</code>

## 2. 오토박싱 (Auto-boxing)

### ◆ 예시

```
1 int a = 10;
2 Integer boxed = a; // int → Integer (자동 박싱)
3
4 List<Integer> list = new ArrayList<>();
5 list.add(42);      // 42 → Integer.valueOf(42)
```

→ 내부적으로 다음과 같이 변환됨:

```
1 Integer boxed = Integer.valueOf(a);
```

### 3. 오토언박싱 (Auto-unboxing)

#### 예시

```
1 Integer boxed = Integer.valueOf(123);
2 int b = boxed; // Integer → int (자동 언박싱)
```

→ 내부적으로 다음과 같이 변환됨:

```
1 int b = boxed.intValue();
```

#### ! 주의사항 1: null 언박싱 → `NullPointerException`

```
1 Integer num = null;
2 int result = num; // ❌ NullPointerException 발생!
```

null을 언박싱하려고 하면 예외 발생 → 반드시 null 체크 필요

#### ! 주의사항 2: `==` vs `equals()` 비교

```
1 Integer a = 100;
2 Integer b = 100;
3 System.out.println(a == b); // ✅ true (캐시 범위: -128 ~ 127)
4
5 Integer x = 1000;
6 Integer y = 1000;
7 System.out.println(x == y); // ❌ false (다른 객체)
8
9 System.out.println(x.equals(y)); // ✅ true (값 비교)
```

`==`는 참조 비교, `.equals()`는 값 비교

#### 🧠 내부 캐시: `Integer.valueOf()`

- `-128 ~ 127` 범위는 JVM 내부 캐시
- 이 범위의 Integer는 같은 객체 참조 반환

```
1 Integer a = Integer.valueOf(127);
2 Integer b = Integer.valueOf(127);
3 System.out.println(a == b); // true
4
5 Integer c = Integer.valueOf(128);
6 Integer d = Integer.valueOf(128);
7 System.out.println(c == d); // false
```

## 실전 활용 예시

### 제네릭/컬렉션

```
1 List<Integer> nums = List.of(1, 2, 3); // 오토박싱
2 int sum = 0;
3 for (int n : nums) { // 오토언박싱
4     sum += n;
5 }
```

### 연산 시 자동 언박싱




```
1 Integer a = 10;
2 Integer b = 20;
3 int result = a + b; // a.intValue() + b.intValue()
```

## 요약 정리

항목	Boxing	Unboxing
정의	기본형 → 객체형	객체형 → 기본형
방식	<code>int → Integer.valueOf()</code>	<code>Integer → intValue()</code>
특징	자동 처리 (Java 5+)	성능 약간 저하 가능
주의	<code>null</code> 언박싱 시 NPE 발생	<code>==</code> 은 참조 비교

## Integer, Double 등의 parse, valueOf, toString

Java에서 기본형과 문자열 간 변환에 자주 사용되는 메서드들인

-  `parseXXX()`
-  `valueOf()`
-  `toString()`

에 대해 `Integer`, `Double`, `Boolean` 등의 래퍼 클래스 중심으로 정리한다.

이 메서드들은 문자열을 숫자(또는 불리언)으로 바꾸거나, 숫자를 문자열로 바꾸는 핵심 도구이다. 실무에서 입력 처리, 설정값 파싱, 파일 입출력, JSON 처리 등에 매우 자주 사용된다.

### 1. `parseXXX(String)`

문자열 → 기본형(primitive) 으로 변환

## 📌 특징

- 리턴 타입이 `int`, `double`, `boolean` 등 기본형
- 내부적으로 문자열을 해석(parse)

## ✅ 예시

```
1 int n = Integer.parseInt("123");           // → 123
2 double d = Double.parseDouble("3.14");     // → 3.14
3 boolean b = Boolean.parseBoolean("true"); // → true
```

## ■ 2. `valueOf(String)`

문자열 → 래퍼 객체(wrapper) 로 변환

## 📌 특징

- 리턴 타입이 `Integer`, `Double`, `Boolean` 등 객체
- 내부적으로 캐싱 활용 (`Integer.valueOf()` 는 -128 ~ 127까지 캐싱)

## ✅ 예시

```
1 Integer n = Integer.valueOf("123");        // → Integer 객체
2 Double d = Double.valueOf("3.14");        // → Double 객체
3 Boolean b = Boolean.valueOf("false");      // → Boolean 객체
```

💡 내부적으로 `parseXXX()` 호출 + Boxing

## ■ 3. `toString()`

숫자 → 문자열 변환

## 📌 특징

- 래퍼 클래스의 인스턴스 또는 정적 메서드로 제공
- 문자열 출력, 로그 기록 등에 사용

## ✅ 예시

```
1 Integer n = 100;
2 String s1 = n.toString();           // 인스턴스 메서드
3 String s2 = Integer.toString(200); // 정적 메서드
```

## 전체 변환 흐름 비교

방향	메서드	반환
문자열 → 기본형	<code>parseXXX("123")</code>	<code>int</code>
문자열 → 래퍼 객체	<code>valueOf("123")</code>	<code>Integer</code>
기본형 → 문자열	<code>String.valueOf(123)</code> 또는 <code>Integer.toString(123)</code>	<code>"123"</code>
래퍼 객체 → 문자열	<code>obj.toString()</code>	<code>"123"</code>

## 예제 통합

```
1 String input = "42";
2
3 // 1. 문자열 → 기본형
4 int a = Integer.parseInt(input);
5
6 // 2. 문자열 → 객체
7 Integer b = Integer.valueOf(input);
8
9 // 3. 기본형 → 문자열
10 String s1 = Integer.toString(a);
11
12 // 4. 객체 → 문자열
13 String s2 = b.toString();
14
15 // 5. 문자열 → boolean
16 boolean tf = Boolean.parseBoolean("true");
```

## 주의사항

### ◆ `parseXXX()` 는 `NumberFormatException` 발생 가능

```
1 Integer.parseInt("abc"); // ❌ 예외 발생
```

→ 항상 `try-catch` 또는 유효성 체크 필요

### ◆ `Boolean.parseBoolean()` 의 특징

- `"true"` (대소문자 무시) → `true`
- `"false"`, `"abc"`, `null` → 모두 `false`

```
1 Boolean.parseBoolean("TRUE"); // true
2 Boolean.parseBoolean("yes"); // false
3 Boolean.parseBoolean(null); // false
```

✅ 요약 표

메서드	입력	출력	용도
<code>parseInt("123")</code>	문자열	<code>int</code>	문자열 숫자 → 정수
<code>valueOf("123")</code>	문자열	<code>Integer</code>	문자열 숫자 → 객체
<code>Integer.toString(123)</code>	정수	문자열	숫자 → 문자열
<code>new Integer(123).toString()</code>	객체	문자열	숫자 객체 → 문자열
<code>String.valueOf(123.45)</code>	double	문자열	범용 숫자 → 문자열

# NumberFormat, DecimalFormat

Java에서 숫자를 문자열로 포맷하거나, 문자열을 숫자로 파싱할 때 사용하는 두 가지 핵심 클래스인

- ✅ `NumberFormat`
- ✅ `DecimalFormat`

을 용도별, 기능별, 예제 중심으로 정리한다.

## 1. `NumberFormat` - 숫자 포맷의 상위 추상 클래스

### ☀ 개요

- `java.text.NumberFormat`
- 숫자, 통화(currency), 퍼센트(%) 등 다양한 국제화 포맷 지원
- 하위 클래스인 `DecimalFormat` 이 실제 포맷 처리 담당
- 언어/국가별 숫자 표현 방식을 자동으로 맞춤

✅ 주요 메서드

메서드	설명
<code>getInstance()</code>	기본 숫자 포맷 생성
<code>getCurrencyInstance()</code>	통화 포맷
<code>getPercentInstance()</code>	퍼센트 포맷
<code>format(Number)</code>	숫자 → 문자열
<code>parse(String)</code>	문자열 → 숫자 ( <code>Number</code> )

## ◆ 예시

```
1 import java.text.NumberFormat;
2 import java.util.Locale;
3
4 public class Main {
5     public static void main(String[] args) throws Exception {
6         NumberFormat nf = NumberFormat.getInstance(Locale.US);
7         System.out.println(nf.format(1234567.89)); // 1,234,567.89
8
9         NumberFormat cf = NumberFormat.getCurrencyInstance(Locale.KOREA);
10        System.out.println(cf.format(5000)); // ₩5,000
11
12        NumberFormat pf = NumberFormat.getPercentInstance();
13        pf.setMinimumFractionDigits(1);
14        System.out.println(pf.format(0.753)); // 75.3%
15    }
16 }
```

## ■ 2. DecimalFormat - 사용자 정의 숫자 포맷

### ☀ 개요

- `NumberFormat` 을 상속한 구체 클래스
- 숫자 포맷을 **패턴 문자열**로 정밀하게 정의 가능
- `"0"`, `"#"`, `"."`, `","`, `"E"`, `"%"`, `"₩"` 등의 **패턴 기호** 사용

### ✓ 포맷 패턴 기호

기호	의미	예시
0	자릿수 채움 (0으로 채움)	"0000" → 42 → 0042
#	자릿수 표시 (0이면 생략)	"###" → 42 → 42, 0 → ""
.	소수점	"0.00" → 3.1 → 3.10
,	천 단위 구분	"#,###" → 10000 → 10,000
%	퍼센트 변환	"0%" → 0.25 → 25%
₩	통화 기호	"₩#,###0" → 1000 → ₩1,000



## ◆ 예시

```
1 import java.text.DecimalFormat;
2
3 public class Main {
4     public static void main(String[] args) {
5         DecimalFormat df1 = new DecimalFormat("0.00");
6         System.out.println(df1.format(3.1));          // 3.10
7
8         DecimalFormat df2 = new DecimalFormat("#,###");
9         System.out.println(df2.format(1234567));      // 1,234,567
10
11        DecimalFormat df3 = new DecimalFormat("₩#,##0");
12        df3.setCurrency(java.util.Currency.getInstance("KRW"));
13        System.out.println(df3.format(1000));         // ₩1,000
14    }
15 }
```

## 🔧 문자열 → 숫자 파싱

```
1 DecimalFormat df = new DecimalFormat("0.00");
2 Number n = df.parse("1234.56");
3 double d = n.doubleValue();
4 System.out.println(d); // 1234.56
```

`parse()` 는 문자열 → `Number` → `int`, `double` 등으로 형변환 필요

## 🔄 `format()` vs `parse()` 비교

메서드	방향	예시
<code>format(Number)</code>	숫자 → 문자열	<code>1234.5</code> → <code>"1,234.50"</code>
<code>parse(String)</code>	문자열 → 숫자	<code>"1,234.5"</code> → <code>1234.5</code>

## ✅ 실전 활용 정리

상황	추천 클래스	설명
단순 숫자 포맷	<code>NumberFormat.getInstance()</code>	기본 숫자
통화 출력	<code>getCurrencyInstance(Locale)</code>	₩, \$ 등
퍼센트 포맷	<code>getPercentInstance()</code>	<code>0.25</code> → <code>25%</code>
사용자 지정 형식	<code>DecimalFormat("0.000")</code>	원하는 형태 직접 지정

상황	추천 클래스	설명
입력 파싱	<code>parse()</code>	문자열을 숫자로 안전하게 변환

## 9.6 입출력 API (`java.io`, `java.nio`)

### `InputStream`, `OutputStream`, `Reader`, `Writer`

Java에서 입출력(I/O)을 처리하는 가장 근본적인 클래스들인

- ✓ `InputStream`, `OutputStream` (바이트 스트림)
- ✓ `Reader`, `Writer` (문자 스트림)

이 네 가지를 정리한다. 이들은 Java I/O의 핵심이며, 파일 읽기, 네트워크 통신, 콘솔 입출력 등 모든 I/O 처리의 기반이다.

#### ◆ 전체 구조 개요

Java I/O는 크게 두 가지 계층으로 나뉘어:

스트림 종류	설명	대표 클래스
바이트 스트림	1바이트 단위 처리	<code>InputStream</code> , <code>OutputStream</code>
문자 스트림	문자(UTF-16) 단위 처리	<code>Reader</code> , <code>Writer</code>

### 1. `InputStream` & `OutputStream` (바이트 기반)

#### 📌 특징

- 이미지, 파일, 바이너리 데이터 등을 다룰 때 사용
- 1바이트(byte) 또는 바이트 배열로 입출력 처리
- 최상위 추상 클래스: `java.io.InputStream`, `java.io.OutputStream`

#### ✓ 주요 메서드 (`InputStream`)

메서드	설명
<code>int read()</code>	한 바이트 읽기 (0~255, EOF: -1)
<code>int read(byte[])</code>	바이트 배열로 읽기
<code>int available()</code>	읽을 수 있는 바이트 수
<code>void close()</code>	스트림 닫기

```

1  InputStream in = new FileInputStream("test.bin");
2  int data;
3  while ((data = in.read()) != -1) {
4      System.out.print((char)data);
5  }
6  in.close();

```

## 주요 메서드 (OutputStream)

메서드	설명
<code>void write(int b)</code>	한 바이트 쓰기
<code>void write(byte[])</code>	배열 쓰기
<code>void flush()</code>	버퍼 비우기
<code>void close()</code>	스트림 닫기

```

1  OutputStream out = new FileOutputStream("out.bin");
2  out.write("Hello".getBytes());
3  out.close();

```

## 2. Reader & Writer (문자 기반)

### 특징

- 텍스트, 유니코드 문자 처리에 최적화
- `InputStreamReader`, `OutputStreamWriter` 로 바이트 ↔ 문자 변환 가능

## 주요 메서드 (Reader)

메서드	설명
<code>int read()</code>	한 문자 읽기 (EOF: -1)
<code>int read(char[])</code>	문자 배열로 읽기
<code>void close()</code>	닫기

```

1  Reader reader = new FileReader("test.txt");
2  int ch;
3  while ((ch = reader.read()) != -1) {
4      System.out.print((char) ch);
5  }
6  reader.close();

```

✓ 주요 메서드 (Writer)

메서드	설명
<code>void write(int)</code>	문자 하나 출력
<code>void write(String)</code>	문자열 출력
<code>void flush()</code>	버퍼 비우기
<code>void close()</code>	닫기

```
1 | Writer writer = new FileWriter("output.txt");
2 | writer.write("안녕하세요!");
3 | writer.close();
```

↔ InputStream ↔ Reader 변환

- 문자 스트림은 내부적으로 바이트 스트림을 감싸서 동작
- 변환 클래스: `InputStreamReader`, `OutputStreamWriter`

```
1 | InputStream in = new FileInputStream("text.txt");
2 | Reader reader = new InputStreamReader(in, "UTF-8"); // 인코딩 명시 가능
```

📌 주요 서브클래스 요약

목적	입력	출력
파일	<code>FileInputStream</code>	<code>FileOutputStream</code>
문자 파일	<code>FileReader</code>	<code>FileWriter</code>
메모리 버퍼	<code>ByteArrayInputStream</code>	<code>ByteArrayOutputStream</code>
문자 버퍼	<code>CharArrayReader</code>	<code>CharArrayWriter</code>
문자열	<code>StringReader</code>	<code>StringWriter</code>
변환	<code>InputStreamReader</code>	<code>OutputStreamWriter</code>
성능 개선	<code>BufferedInputStream</code>	<code>BufferedOutputStream</code>
문자 버퍼링	<code>BufferedReader</code>	<code>BufferedWriter</code>

## ✓ 정리 비교표

항목	InputStream / OutputStream	Reader / Writer
단위	바이트(byte)	문자(char)
인코딩 처리	✗ 없음	✓ 필요
사용 대상	이미지, 바이너리, PDF 등	텍스트 파일, JSON, XML 등
변환	InputStreamReader	OutputStreamWriter

## File, BufferedReader, PrintWriter

Java IO에서 파일을 직접 다루거나, 텍스트 입출력을 효율적으로 처리할 때 많이 사용하는 3가지 핵심 클래스이다.

- ✓ `File` - 파일 경로 및 메타 정보 관리
- ✓ `BufferedReader` - 문자 입력 버퍼링 (라인 단위 읽기)
- ✓ `PrintWriter` - 포매팅된 출력 (문자 기반 쓰기)

### 1. File 클래스

#### ☀ 개요

- `java.io.File`
- 실제 파일, 디렉토리의 경로를 추상화한 객체
- 입출력은 하지 않지만, 파일 존재 여부, 크기, 경로, 삭제, 생성 등 메타 정보 조작에 사용

#### ✓ 주요 생성자

```
1 new File("test.txt")
2 new File("/path/to", "file.txt")
```

#### ✓ 주요 메서드

메서드	설명
<code>exists()</code>	파일 존재 여부
<code>createNewFile()</code>	새 파일 생성
<code>delete()</code>	파일 삭제
<code>isFile()</code> / <code>isDirectory()</code>	파일 or 디렉토리 구분
<code>getName()</code> / <code>getPath()</code> / <code>getAbsolutePath()</code>	파일 정보
<code>length()</code>	파일 크기

메서드	설명
<code>listFiles()</code>	디렉토리 안의 파일 목록

## ◆ 예시

```

1  import java.io.File;
2
3  public class Main {
4      public static void main(String[] args) throws Exception {
5          File file = new File("hello.txt");
6
7          if (!file.exists()) {
8              file.createNewFile();
9              System.out.println("파일 생성됨");
10         }
11
12         System.out.println("파일 크기: " + file.length() + " bytes");
13         System.out.println("절대 경로: " + file.getAbsolutePath());
14     }
15 }

```

## ■ 2. `BufferedReader`

### ☀ 개요

- `java.io.BufferedReader`
- 문자 스트림을 버퍼링하여 성능 향상
- 라인 단위 읽기 (`readLine()`) 가능
- `Reader` 하위 클래스 (`FileReader`, `InputStreamReader` 등과 함께 사용)

### ✅ 생성자

```

1  BufferedReader br = new BufferedReader(new FileReader("file.txt"));

```

### ✅ 주요 메서드

메서드	설명
<code>read()</code>	한 문자 읽기
<code>readLine()</code>	한 줄 읽기 (개행 문자는 제거됨)
<code>ready()</code>	읽을 데이터가 준비되었는지 확인

메서드	설명
<code>close()</code>	스트림 닫기

## ◆ 예시

```

1  import java.io.*;
2
3  public class Main {
4      public static void main(String[] args) throws IOException {
5          BufferedReader reader = new BufferedReader(new FileReader("data.txt"));
6          String line;
7
8          while ((line = reader.readLine()) != null) {
9              System.out.println(line);
10         }
11
12         reader.close();
13     }
14 }

```

## ■ 3. PrintWriter

### ☀ 개요

- `java.io.PrintWriter`
- `Writer` 하위 클래스
- 포매팅된 문자 출력, 자동 개행, `System.out` 대체 가능
- 내부적으로 `BufferedWriter`를 사용

### ✅ 생성자

```

1  PrintWriter pw = new PrintWriter("output.txt");           // 기본 생성
2  PrintWriter pw = new PrintWriter(new FileWriter("a.txt")); // 버퍼링 없이
3  PrintWriter pw = new PrintWriter(new BufferedWriter(new FileWriter("a.txt"))); // 고속

```

### ✅ 주요 메서드

메서드	설명
<code>print()</code> , <code>println()</code>	출력 및 줄바꿈
<code>printf(format, args...)</code>	포맷 출력 ( <code>System.out.printf</code> 와 동일)
<code>write()</code>	문자 단위 출력

메서드	설명
<code>flush()</code>	버퍼 비움
<code>close()</code>	스트림 닫기

## ◆ 예시

```

1 import java.io.*;
2
3 public class Main {
4     public static void main(String[] args) throws IOException {
5         PrintWriter writer = new PrintWriter("output.txt");
6
7         writer.println("첫 줄입니다.");
8         writer.printf("숫자: %d, 실수: %.2f\n", 42, 3.14);
9
10        writer.close(); // flush 포함됨
11    }
12 }

```

## 🧠 실전 예제: `BufferedReader` + `PrintWriter` 조합

```

1 BufferedReader reader = new BufferedReader(new FileReader("input.txt"));
2 PrintWriter writer = new PrintWriter("output.txt");
3
4 String line;
5 while ((line = reader.readLine()) != null) {
6     writer.println("읽은 줄: " + line);
7 }
8
9 reader.close();
10 writer.close();

```

## ✅ 요약 비교표

클래스	목적	주요 기능
<code>File</code>	파일/디렉토리 정보	경로, 존재 확인, 삭제, 생성
<code>BufferedReader</code>	입력 성능 향상	줄 단위 읽기, 버퍼링
<code>PrintWriter</code>	출력 성능 + 편의성	print, println, printf 지원, 버퍼링



# Files, Paths, ByteBuffer, FileChannel (NIO)

Java NIO(New I/O)에서 파일 입출력을 처리하기 위해 사용하는 핵심 클래스들인

- ✓ Files
- ✓ Paths
- ✓ ByteBuffer
- ✓ FileChannel

을 정리한다. 이들은 전통적인 IO보다 빠르고 유연한 고성능 입출력 방식을 제공하며, 대용량 파일 처리나 메모리 매핑, 버퍼 기반 처리가 필요할 때 특히 유용하다.

## 1. Paths

### ☀ 개요

- `java.nio.file.Paths`: 경로를 나타내는 객체
- `Path` 객체를 만들기 위한 헬퍼 클래스
- `File` 보다 더 유연하고 OS 독립적인 경로 조작 가능

### ✓ 예시

```
1 import java.nio.file.*;
2
3 Path path = Paths.get("data", "example.txt"); // 경로 연결
4 System.out.println(path.toAbsolutePath());    // 절대 경로 출력
5 System.out.println(Files.exists(path));       // 존재 여부
```

## 2. Files

### ☀ 개요

- `java.nio.file.Files`: 파일 조작 기능 모음 (정적 메서드)
- `Path` 객체 기반으로 파일을 읽고 쓰거나, 복사, 삭제, 이동 등 수행

### ✓ 주요 메서드

메서드	설명
<code>readAllLines(Path)</code>	모든 줄을 <code>List&lt;String&gt;</code> 으로 읽기
<code>readAllBytes(Path)</code>	파일을 <code>byte[]</code> 로 읽기
<code>write(Path, byte[])</code>	<code>byte[]</code> 를 파일에 쓰기
<code>copy()</code> , <code>move()</code> , <code>delete()</code>	파일 복사, 이동, 삭제
<code>isDirectory()</code> , <code>exists()</code>	파일 상태 확인
<code>lines(Path)</code>	<code>Stream&lt;String&gt;</code> 반환

## ◆ 예시: 파일 쓰고 읽기

```
1 Path path = Paths.get("sample.txt");
2
3 // 쓰기
4 Files.write(path, "Hello, NIO!".getBytes());
5
6 // 읽기
7 String content = Files.readString(path); // Java 11+
8 System.out.println(content);
```

## 3. ByteBuffer

### ☀ 개요

- `java.nio.ByteBuffer`: 바이트 데이터를 읽고 쓰기 위한 버퍼
- `FileChannel`, `SocketChannel` 등과 함께 사용
- 직접 할당 (`allocateDirect`) vs JVM 메모리 (`allocate`)

### ✅ 주요 메서드

메서드	설명
<code>allocate(int)</code>	JVM 힙 메모리에 버퍼 생성
<code>allocateDirect(int)</code>	OS 메모리(Direct 메모리)
<code>put(byte)</code> , <code>put(byte[])</code>	데이터 쓰기
<code>get()</code> , <code>get(byte[])</code>	데이터 읽기
<code>flip()</code>	읽기 모드로 전환
<code>clear()</code>	쓰기 모드로 초기화
<code>rewind()</code>	읽기 위치를 처음으로 되돌림

## ◆ 예시

```
1 ByteBuffer buffer = ByteBuffer.allocate(1024);
2 buffer.put("Hello ByteBuffer".getBytes());
3 buffer.flip();
4
5 while (buffer.hasRemaining()) {
6     System.out.print((char) buffer.get());
7 }
```

## 4. FileChannel

### ☀ 개요

- `java.nio.channels.FileChannel`: 파일에 직접 접근 가능한 채널
- `RandomAccessFile`, `FileInputStream`, `FileOutputStream`으로부터 생성
- **seek, position, truncate, memory-mapped I/O** 지원

### ✅ 주요 메서드

메서드	설명
<code>read(ByteBuffer)</code>	버퍼로 읽기
<code>write(ByteBuffer)</code>	버퍼에서 쓰기
<code>position()</code>	읽기/쓰기 위치 확인
<code>size()</code>	파일 크기 반환
<code>truncate(long)</code>	파일 크기 자르기
<code>map()</code>	메모리 맵핑 I/O (고성능)

### ◆ 예시: 버퍼 + 채널을 이용한 파일 복사

```
1  import java.io.*;
2  import java.nio.*;
3  import java.nio.channels.*;
4
5  public class Main {
6      public static void main(String[] args) throws IOException {
7          try (
8              FileInputStream fis = new FileInputStream("source.txt");
9              FileOutputStream fos = new FileOutputStream("dest.txt");
10             FileChannel inChannel = fis.getChannel();
11             FileChannel outChannel = fos.getChannel()
12         ) {
13             ByteBuffer buffer = ByteBuffer.allocate(1024);
14             while (inChannel.read(buffer) > 0) {
15                 buffer.flip();
16                 outChannel.write(buffer);
17                 buffer.clear();
18             }
19         }
20     }
21 }
```

## 정리 비교표

클래스	설명	주요 용도
<code>Paths</code>	<code>Path</code> 객체 생성 도우미	경로 조작
<code>Files</code>	정적 메서드로 파일 처리	간단한 읽기/쓰기/삭제
<code>ByteBuffer</code>	바이트 저장/조작용 버퍼	채널 기반 입출력
<code>FileChannel</code>	파일 직접 접근 채널	대용량 파일, 버퍼 입출력, 메모리 매핑

### ✅ 요약 예시: "파일 → `ByteBuffer` → 채널 → 파일"

```
1 Path source = Paths.get("source.txt");
2 Path dest = Paths.get("copy.txt");
3
4 try (
5     FileChannel in = FileChannel.open(source);
6     FileChannel out = FileChannel.open(dest, StandardOpenOption.CREATE,
7     StandardOpenOption.WRITE)
8 ) {
9     ByteBuffer buffer = ByteBuffer.allocate(1024);
10    while (in.read(buffer) > 0) {
11        buffer.flip();
12        out.write(buffer);
13        buffer.clear();
14    }
15 }
```

## 객체 직렬화: `Serializable`, `ObjectInputStream` 등

Java에서 객체를 파일, 네트워크 등으로 저장하거나 전송할 수 있도록 만드는 기능인

✅ **객체 직렬화(Serialization)**에 대해 정리해준다.

이 기능은 다음 요소들로 구성된다:

- ✅ `Serializable` 인터페이스
- ✅ `ObjectOutputStream` (객체를 저장/전송)
- ✅ `ObjectInputStream` (객체를 복원/수신)
- 🚫 직렬화 대상 제외: `transient` 키워드
- 💡 직렬 버전 관리: `serialVersionUID`

## 1. 객체 직렬화(Serialization)란?

객체의 상태(state)를 바이트 형태로 변환하여 저장하거나 전송할 수 있게 하는 것

📦 직렬화 → 파일/네트워크로 전송

🔄 역직렬화 → 저장된 바이트 데이터를 다시 객체로 복원

## 2. Serializable 인터페이스

### ☀ 개요

- `java.io.Serializable`은 마커 인터페이스 (메서드 없음)
- 이 인터페이스를 구현한 객체만 직렬화할 수 있음

```
1 import java.io.Serializable;
2
3 public class User implements Serializable {
4     private String name;
5     private int age;
6
7     public User(String name, int age) {
8         this.name = name;
9         this.age = age;
10    }
11 }
```

## 3. ObjectOutputStream & ObjectInputStream

### ✅ 객체 저장 (직렬화)

```
1 import java.io.*;
2
3 User user = new User("Alice", 30);
4 ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("user.ser"));
5 oos.writeObject(user);
6 oos.close();
```

### ✅ 객체 복원 (역직렬화)

```
1 ObjectInputStream ois = new ObjectInputStream(new FileInputStream("user.ser"));
2 User restored = (User) ois.readObject();
3 ois.close();
4
5 System.out.println(restored.getName());
```

## ⚠ 4. transient 키워드

직렬화 대상에서 제외하고 싶은 필드에 사용

```
1 class User implements Serializable {
2     private String name;
3     private transient String password; // 저장되지 않음
4 }
```

## 🧠 5. serialVersionUID

클래스의 버전을 식별하는 정수값.

직렬화된 객체를 정확히 복원하기 위한 ID.

```
1 private static final long serialVersionUID = 1L;
```

- 자동으로 부여되지만, 클래스 구조가 바뀌면 **호환되지 않음**
- 명시적으로 선언해서 버전 충돌 방지

## 🔧 실전 예제: 직렬화 + 역직렬화 전체 흐름

```
1 class Person implements Serializable {
2     private static final long serialVersionUID = 1L;
3     private String name;
4     private transient String secret;
5
6     public Person(String name, String secret) {
7         this.name = name;
8         this.secret = secret;
9     }
10
11     @Override
12     public String toString() {
13         return name + ", secret=" + secret;
14     }
15 }
16
17 public class Main {
18     public static void main(String[] args) throws Exception {
19         // 직렬화
20         ObjectOutputStream out = new ObjectOutputStream(new
21         FileOutputStream("person.ser"));
22         out.writeObject(new Person("Bob", "1234"));
23         out.close();
24
25         // 역직렬화
26         ObjectInputStream in = new ObjectInputStream(new
27         FileInputStream("person.ser"));
```

```

26     Person p = (Person) in.readObject();
27     in.close();
28
29     System.out.println(p); // secret=null (transient 적용됨)
30 }
31 }

```

## Serializable vs Externalizable

항목	Serializable	Externalizable
방식	자동 직렬화	개발자가 수동으로 제어
성능	상대적으로 느림	빠름 (직접 구현)
인터페이스 메서드	없음	<code>writeExternal()</code> , <code>readExternal()</code>
사용성	매우 간편	고급 제어 가능

## 요약 정리표

개념	설명
<code>Serializable</code>	직렬화 대상임을 표시
<code>ObjectOutputStream</code>	객체 → 파일/바이트로 저장
<code>ObjectInputStream</code>	바이트 → 객체로 복원
<code>transient</code>	직렬화 대상 제외 필드
<code>serialVersionUID</code>	클래스 버전 ID (명시 권장)

## 9.7 기타 주요 API 개요

### `java.util.concurrent`: Executor, Future, Lock 등

Java의 고성능 동시성 프로그래밍을 위한 핵심 패키지인

 `java.util.concurrent`에 대해 정리한다.

이 패키지는 멀티스레드 환경에서 더 안전하고 유연하게 작업을 분산, 조율, 동기화할 수 있게 도와주는 다양한 도구들을 제공한다.

### 1. 주요 구성 요소 개요

범주	주요 클래스 / 인터페이스	설명
스레드 풀	<code>Executor</code> , <code>ExecutorService</code> , <code>ThreadPoolExecutor</code>	스레드 풀 기반 작업 실행

범주	주요 클래스 / 인터페이스	설명
작업 결과 처리	<code>Future</code> , <code>Callable</code>	비동기 작업 실행 및 결과 조회
락/동기화	<code>Lock</code> , <code>ReentrantLock</code> , <code>ReadWriteLock</code>	고급 동기화 제어
동시성 컬렉션	<code>ConcurrentHashMap</code> , <code>CopyOnWriteArrayList</code>	멀티스레드 환경에서 안전한 컬렉션
스케줄링	<code>ScheduledExecutorService</code>	정기적/지연 작업 스케줄링
동기화 도구	<code>CountDownLatch</code> , <code>CyclicBarrier</code> , <code>Semaphore</code> , <code>Exchanger</code>	스레드 간 협업 제어

## 2. Executor, ExecutorService

### ☀ 기본 구조

```
1 Executor executor = Executors.newSingleThreadExecutor();
2 executor.execute(() -> System.out.println("Hello Executor!"));
```

### ☀ `ExecutorService` - 결과 처리와 종료 지원

```
1 ExecutorService service = Executors.newFixedThreadPool(4);
2 service.submit(() -> {
3     System.out.println("작업 실행 중");
4 });
5 service.shutdown();
```

## 3. Callable & Future

### ✅ Callable: `Runnable` 과 달리 값을 반환함

```
1 Callable<Integer> task = () -> {
2     return 5 + 10;
3 };
```

### ✅ Future: 비동기 작업의 결과를 받을 수 있음

```
1 ExecutorService pool = Executors.newSingleThreadExecutor();
2 Future<Integer> future = pool.submit(task);
3
4 Integer result = future.get(); // 블로킹 호출
5 System.out.println(result);
6 pool.shutdown();
```



## 4. Lock 인터페이스

### 기본 Lock 사용

```
1 Lock lock = new ReentrantLock();
2 lock.lock();
3 try {
4     // 임계 영역
5 } finally {
6     lock.unlock();
7 }
```

### 주요 메서드

메서드	설명
<code>lock()</code>	락 획득 (블로킹)
<code>tryLock()</code>	락 시도 (즉시 실패 가능)
<code>unlock()</code>	락 해제
<code>lockInterruptibly()</code>	인터럽트 가능한 락

## 5. ReadWriteLock

- 읽기 작업은 동시에 가능, 쓰기 작업은 단독으로 실행됨

```
1 ReadWriteLock rwLock = new ReentrantReadWriteLock();
2 rwLock.readLock().lock();
3 // 공유 자원 읽기
4 rwLock.readLock().unlock();
```

## 6. Concurrent 컬렉션

컬렉션	설명
<code>ConcurrentHashMap</code>	쓰레드 안전한 해시맵
<code>CopyOnWriteArrayList</code>	읽기 빈도 높고 쓰기 적은 리스트
<code>ConcurrentLinkedQueue</code>	락 없는 대기열
<code>BlockingQueue (LinkedBlockingQueue, ArrayBlockingQueue)</code>	생산자-소비자 패턴에 유용

```
1 ConcurrentHashMap<String, Integer> map = new ConcurrentHashMap<>();
2 map.put("one", 1);
3 System.out.println(map.get("one"));
```

## 7. 동기화 도구

도구	설명
<code>CountDownLatch</code>	여러 스레드 작업 완료 대기
<code>CyclicBarrier</code>	여러 스레드가 동시에 다음 단계로
<code>Semaphore</code>	리소스 접근 동시 개수 제한
<code>Exchanger</code>	두 스레드 간 데이터 교환

```
1 CountDownLatch latch = new CountDownLatch(3);
2
3 Runnable task = () -> {
4     // 작업 수행
5     latch.countDown();
6 };
7
8 for (int i = 0; i < 3; i++) new Thread(task).start();
9 latch.await(); // 모든 스레드가 끝날 때까지 대기
```

## ✓ 실전 예제: 멀티스레드 작업 + Future + 락

```
1 import java.util.concurrent.*;
2 import java.util.concurrent.locks.*;
3
4 class Counter {
5     private int count = 0;
6     private Lock lock = new ReentrantLock();
7
8     public void increment() {
9         lock.lock();
10        try {
11            count++;
12        } finally {
13            lock.unlock();
14        }
15    }
16
17    public int get() {
18        return count;
19    }
20 }
21
22 public class Main {
23     public static void main(String[] args) throws Exception {
24         Counter counter = new Counter();
25         ExecutorService pool = Executors.newFixedThreadPool(5);
```

```

26
27     Callable<Void> task = () -> {
28         for (int i = 0; i < 1000; i++) counter.increment();
29         return null;
30     };
31
32     List<Future<Void>> futures = new ArrayList<>();
33     for (int i = 0; i < 5; i++) {
34         futures.add(pool.submit(task));
35     }
36
37     for (Future<Void> f : futures) f.get();
38
39     pool.shutdown();
40     System.out.println("최종 카운트: " + counter.get());
41 }
42 }

```

## 🧠 핵심 요약

개념	설명
<code>Executor</code>	작업 실행 인터페이스
<code>Future</code>	비동기 결과 확인 도구
<code>Lock</code> , <code>ReadWriteLock</code>	고급 동기화 제어
<code>ConcurrentHashMap</code>	병렬 환경에서도 안전한 맵
<code>CountDownLatch</code> , <code>Semaphore</code>	스레드 협력/제어 도구

## java.sql: JDBC와 DB 연동

Java에서 데이터베이스와 연결하는 표준 API인

✅ `java.sql` 기반 **JDBC(Java Database Connectivity)** 를 깊이 있게 다뤄본다.

### 1. JDBC란?

Java에서 관계형 데이터베이스(RDBMS)와 연결하여 SQL을 실행하고 결과를 처리할 수 있게 해주는 API

#### 🔗 주요 기능:

- DB 연결 (Connection)
- SQL 실행 (Statement / PreparedStatement)
- 결과 처리 (ResultSet)
- 트랜잭션 처리
- 자원 정리

## 2. JDBC 주요 클래스 구조

계층	클래스 / 인터페이스	설명
연결	<code>Driver</code> , <code>DriverManager</code> , <code>Connection</code>	DB 연결 및 세션 생성
실행	<code>Statement</code> , <code>PreparedStatement</code> , <code>CallableStatement</code>	SQL 실행
결과	<code>ResultSet</code>	쿼리 결과 탐색
설정	<code>ResultSetMetaData</code> , <code>DatabaseMetaData</code>	DB 및 결과 정보
기타	<code>SQLException</code> , <code>BatchUpdateException</code> 등	예외 처리

## 3. JDBC 사용 단계 요약

### 1 드라이버 로딩

```
1 | Class.forName("com.mysql.cj.jdbc.Driver"); // 최신 JDBC 드라이버는 생략 가능
```

### 2 DB 연결

```
1 | Connection conn = DriverManager.getConnection(  
2 |     "jdbc:mysql://localhost:3306/mydb",  
3 |     "username", "password"  
4 | );
```

### 3 SQL 실행

```
1 | Statement stmt = conn.createStatement();  
2 | ResultSet rs = stmt.executeQuery("SELECT * FROM users");
```

### 4 결과 처리

```
1 | while (rs.next()) {  
2 |     System.out.println(rs.getString("name"));  
3 | }
```

### 5 자원 해제

```
1 | rs.close();  
2 | stmt.close();  
3 | conn.close();
```

## 4. PreparedStatement 사용 (권장)

```
1 String sql = "SELECT * FROM users WHERE id = ?";
2 PreparedStatement pstmt = conn.prepareStatement(sql);
3 pstmt.setInt(1, 10);
4 ResultSet rs = pstmt.executeQuery();
```

### 장점:

- SQL 인젝션 방지
- 성능 향상 (쿼리 캐시)
- 가독성 향상

## 5. 트랜잭션 처리

```
1 conn.setAutoCommit(false); // 수동 트랜잭션 시작
2
3 try {
4     // 여러 쿼리 수행
5     conn.commit(); // 성공 시 커밋
6 } catch (Exception e) {
7     conn.rollback(); // 실패 시 롤백
8 }
```

## 6. ResultSet 결과 탐색

메서드	설명
<code>rs.next()</code>	다음 행으로 이동
<code>rs.getString("컬럼명")</code>	문자열 조회
<code>rs.getInt("컬럼명")</code>	정수 조회
<code>rs.getDate("컬럼명")</code>	날짜 조회
<code>rs.isNull()</code>	NULL 여부 확인

## 7. 자주 쓰는 JDBC URL 예시

DBMS	URL 예시
MySQL	<code>jdbc:mysql://localhost:3306/dbname</code>
PostgreSQL	<code>jdbc:postgresql://localhost:5432/dbname</code>

DBMS	URL 예시
Oracle	<code>jdbc:oracle:thin:@localhost:1521:xe</code>
H2	<code>jdbc:h2:mem:testdb</code>

## 예제 코드 (MySQL 기준)

```

1  import java.sql.*;
2
3  public class JdbcExample {
4      public static void main(String[] args) throws Exception {
5          Class.forName("com.mysql.cj.jdbc.Driver");
6          try (Connection conn = DriverManager.getConnection(
7              "jdbc:mysql://localhost:3306/testdb", "root", "password");
8              PreparedStatement pstmt = conn.prepareStatement("SELECT * FROM users WHERE
id = ?")) {
9
10             pstmt.setInt(1, 1);
11             try (ResultSet rs = pstmt.executeQuery()) {
12                 while (rs.next()) {
13                     System.out.println("이름: " + rs.getString("name"));
14                 }
15             }
16         }
17     }
18 }

```

## 고급 개념 요약

항목	설명
<code>Batch</code>	다수의 쿼리 묶음 실행 ( <code>addBatch()</code> , <code>executeBatch()</code> )
<code>Connection Pool</code>	재사용 가능한 연결 관리 (ex. HikariCP)
<code>Metadata</code>	DB 구조/정보 탐색 ( <code>DatabaseMetaData</code> , <code>ResultSetMetaData</code> )
<code>CallableStatement</code>	저장 프로시저 호출
<code>try-with-resources</code>	자원 자동 해제 구조 (Java 7+)

## JDBC 실전 팁

- 항상 `PreparedStatement` 를 사용할 것
- `try-with-resources` 로 연결 누수 방지
- 트랜잭션 경계 (`commit/rollback`) 는 명확하게

- 커넥션 풀 라이브러리(HikariCP, Apache DBCP)와 함께 사용 권장

## java.net: Socket, URL 등 네트워크 프로그래밍

### 1. java.net 패키지 개요

Java의 `java.net`은 **TCP/IP 기반 네트워크 프로그래밍**을 위한 기본 API를 제공하며, 다음 기능을 포함한다:

- 소켓 통신 (TCP/UDP)
- URL 접근 및 웹 데이터 수신
- 서버 및 클라이언트 구현
- HTTP/FTP 등 프로토콜 핸들링

### 2. 소켓 통신 개요 (TCP 기반)

#### ◆ 기본 구조

역할	클래스	설명
클라이언트	<code>Socket</code>	서버에 연결 요청
서버	<code>ServerSocket</code>	연결 수락 및 통신 소켓 반환

### 3. TCP 클라이언트 예제 (`Socket` 사용)

```
1  import java.io.*;
2  import java.net.*;
3
4  public class Client {
5      public static void main(String[] args) throws IOException {
6          try (Socket socket = new Socket("localhost", 12345);
7              PrintWriter out = new PrintWriter(socket.getOutputStream(), true);
8              BufferedReader in = new BufferedReader(new
9                  InputStreamReader(socket.getInputStream())) {
10
11              out.println("Hello from client");
12              String response = in.readLine();
13              System.out.println("서버 응답: " + response);
14          }
15      }
16  }
```

## 4. TCP 서버 예제 (ServerSocket 사용)

```
1 import java.io.*;
2 import java.net.*;
3
4 public class Server {
5     public static void main(String[] args) throws IOException {
6         try (ServerSocket serverSocket = new ServerSocket(12345)) {
7             System.out.println("서버 대기 중...");
8
9             try (Socket clientSocket = serverSocket.accept();
10                 BufferedReader in = new BufferedReader(new
11                     InputStreamReader(clientSocket.getInputStream()));
12                 PrintWriter out = new PrintWriter(clientSocket.getOutputStream(),
13                     true)) {
14
15                 String message = in.readLine();
16                 System.out.println("클라이언트로부터 받은 메시지: " + message);
17                 out.println("Hello Client!");
18             }
19         }
20     }
21 }
```

## 5. URL 클래스: 웹 자원 접근

```
1 import java.io.*;
2 import java.net.*;
3
4 public class UrlExample {
5     public static void main(String[] args) throws Exception {
6         URL url = new URL("https://example.com");
7         BufferedReader in = new BufferedReader(new
8             InputStreamReader(url.openStream()));
9
10        String line;
11        while ((line = in.readLine()) != null) {
12            System.out.println(line);
13        }
14        in.close();
15    }
16 }
```

### 🔗 URL 주요 메서드

메서드	설명
<code>getProtocol()</code>	프로토콜 (http, https, ftp)
<code>getHost()</code>	호스트 이름



메서드	설명
<code>getPort()</code>	포트 번호
<code>getPath()</code>	경로
<code>openStream()</code>	입력 스트림 반환 (GET 요청)

## 6. UDP 통신 (비연결형)

클라이언트: `DatagramSocket`, `DatagramPacket`

```

1 DatagramSocket socket = new DatagramSocket();
2 InetAddress address = InetAddress.getByName("localhost");
3 byte[] buf = "Hello UDP".getBytes();
4
5 DatagramPacket packet = new DatagramPacket(buf, buf.length, address, 12345);
6 socket.send(packet);

```

서버:

```

1 DatagramSocket socket = new DatagramSocket(12345);
2 byte[] buf = new byte[256];
3
4 DatagramPacket packet = new DatagramPacket(buf, buf.length);
5 socket.receive(packet);
6 System.out.println("수신 메시지: " + new String(packet.getData()));

```

## 7. 고급 기능

기능	클래스	설명
IP 정보 확인	<code>InetAddress</code>	도메인/IP 확인
멀티캐스트	<code>MulticastSocket</code>	다수 대상에 브로드캐스트
프록시 설정	<code>Proxy</code>	프록시 서버 통한 요청
연결 시간 설정	<code>HttpURLConnection.setConnectTimeout()</code>	타임아웃 제어

### 실전 팁

- TCP는 신뢰성 있는 연결, UDP는 빠르지만 순서보장 없음
- 서버는 항상 `accept()` 후 멀티스레드로 클라이언트 처리
- URL 클래스는 간단한 웹 크롤링/REST API 사용에 적합
- Java 11+ 부터는 `HttpClient` 사용이 권장됨 (REST API 사용 시)


## 요약 비교

항목	TCP ( Socket )	UDP ( DatagramSocket )
연결 방식	연결형	비연결형
신뢰성	보장됨	보장되지 않음
전송 속도	느림	빠름
사용 목적	채팅, 파일 전송	스트리밍, 브로드캐스트

## java.util.stream: Stream API

### 1. Stream API란?

데이터 소스를 추상화하여 **함수형 스타일로 처리**할 수 있도록 제공하는 Java 8의 API

 `java.util.stream` 패키지에 포함

- 데이터 컬렉션(List, Set 등)을 **선언적 방식**으로 처리
- 내부 반복 사용 → 명령형 for-loop보다 **가독성/효율성 우수**
- 병렬 처리 지원 → `parallelStream()`

### 2. Stream의 특징

특징	설명
파이프라인 처리	연산들이 연결되어 한 흐름에서 작동 ( <code>map().filter().collect()</code> )
지연 연산	중간 연산은 실제 계산을 미루며 최종 연산 시 실행됨
무상태/불변성	데이터 소스를 변경하지 않고, 새 값을 리턴함
병렬 처리 가능	<code>parallelStream()</code> 사용 시 내부적으로 ForkJoinPool 활용

### 3. Stream 생성 방법

```
1 // 컬렉션 기반
2 List<String> list = List.of("a", "b", "c");
3 Stream<String> stream1 = list.stream();
4
5 // 배열 기반
6 String[] arr = {"x", "y", "z"};
7 Stream<String> stream2 = Arrays.stream(arr);
8
9 // 값 직접
10 Stream<Integer> stream3 = Stream.of(1, 2, 3);
11
```

```

12 // 무한 스트림
13 Stream<Integer> stream4 = Stream.iterate(0, n -> n + 2); // 0, 2, 4, ...
14 Stream<Double> stream5 = Stream.generate(Math::random);

```

## 4. 중간 연산 (Intermediate Operations)

메서드	설명
<code>filter(Predicate)</code>	조건을 만족하는 요소만 통과
<code>map(Function)</code>	각 요소를 다른 값으로 매핑
<code>flatMap(Function)</code>	중첩된 구조를 평탄화
<code>distinct()</code>	중복 제거
<code>sorted()</code>	정렬
<code>limit(n)</code>	n개까지만 자르기
<code>skip(n)</code>	처음 n개 건너뛰기
<code>peek(Consumer)</code>	디버깅용 중간 확인

예시:

```

1 list.stream()
2   .filter(s -> s.length() > 2)
3   .map(String::toUpperCase)
4   .sorted()
5   .forEach(System.out::println);

```

## 5. 최종 연산 (Terminal Operations)

메서드	설명
<code>forEach(Consumer)</code>	각 요소에 작업 수행
<code>collect(Collector)</code>	리스트, 셋 등으로 수집
<code>reduce(...)</code>	누적 계산 수행
<code>count()</code>	요소 수 반환
<code>min()</code> , <code>max()</code>	최소, 최대값 반환
<code>anyMatch()</code> , <code>allMatch()</code> , <code>noneMatch()</code>	조건 만족 여부 판단
<code>findFirst()</code> , <code>findAny()</code>	요소 하나 반환 (Optional)

 예시:

```
1 int sum = List.of(1, 2, 3).stream().reduce(0, Integer::sum);
2 long count = list.stream().filter(s -> !s.isEmpty()).count();
```

## ■ 6. Collectors 클래스와 수집 연산

```
1 List<String> result = list.stream()
2     .filter(s -> s.length() > 3)
3     .collect(Collectors.toList());
```

### 주요 Collectors

메서드	설명
<code>toList()</code> , <code>toSet()</code>	컬렉션 수집
<code>joining(", ")</code>	문자열 연결
<code>groupingBy(Function)</code>	그룹핑 (Map<key, List>)
<code>partitioningBy(Predicate)</code>	조건 기반 2분할
<code>counting()</code> , <code>summingInt()</code>	수치 집계
<code>mapping()</code>	매핑 후 collect

 그룹핑 예시:

```
1 Map<Integer, List<String>> grouped = list.stream()
2     .collect(Collectors.groupingBy(String::length));
```

## ■ 7. Optional + Stream 조합

```
1 Optional<String> opt = list.stream()
2     .filter(s -> s.startsWith("A"))
3     .findFirst();
```

## ■ 8. 병렬 스트림 (Parallel Stream)

```
1 list.parallelStream()
2     .map(String::toLowerCase)
3     .forEach(System.out::println);
```

- 내부적으로 `ForkJoinPool` 사용
- 데이터 의존성이 없는 연산에 적합

- 병렬성에 따른 부작용/성능 고려 필요

## 9. flatMap vs map 비교

```
1 List<List<String>> nested = List.of(List.of("a", "b"), List.of("c"));
2 List<String> flat = nested.stream()
3     .flatMap(List::stream)
4     .collect(Collectors.toList());
```

연산	설명
<code>map()</code>	요소 1:1 변환
<code>flatMap()</code>	요소 1:N → 1:1 평탄화 변환

### ✓ 실무 팁 요약

- `stream().filter().map().collect()` 패턴에 익숙해질 것
- `peek()` 은 디버깅용으로만 사용할 것
- `Optional` 과 함께 쓰면 안전한 연산 가능
- Stream은 단 한 번만 소비됨 (재사용 불가)
- 성능이 중요하다면 `for` 문과 비교 벤치마크 필수
- `Collectors.groupingBy()` 를 적극 활용할 것

## java.util.function: 함수형 인터페이스

### 1. 함수형 인터페이스란?

하나의 추상 메서드만 가지는 인터페이스

→ 랴다식 또는 메서드 참조의 대상이 될 수 있음

→ `@FunctionalInterface` 어노테이션으로 명시 가능 (선택적)

#### 예시

```
1 @FunctionalInterface
2 interface MyFunc {
3     int apply(int x); // 단 하나의 추상 메서드
4 }
```

```
1 MyFunc square = x -> x * x;
2 System.out.println(square.apply(5)); // 25
```

## 2. 주요 함수형 인터페이스 4종

인터페이스	메서드	설명
<code>Function&lt;T, R&gt;</code>	<code>R apply(T t)</code>	T를 받아 R을 리턴
<code>Consumer&lt;T&gt;</code>	<code>void accept(T t)</code>	T를 소비(출력/저장 등)
<code>Supplier&lt;T&gt;</code>	<code>T get()</code>	아무것도 받지 않고 T 반환
<code>Predicate&lt;T&gt;</code>	<code>boolean test(T t)</code>	T를 받아 boolean 판단

## 3. `Function<T, R>` 예제

```
1 Function<String, Integer> strLength = s -> s.length();
2 System.out.println(strLength.apply("hello")); // 5
```

### 함수 합성

```
1 Function<Integer, Integer> f1 = x -> x + 1;
2 Function<Integer, Integer> f2 = x -> x * 2;
3
4 Function<Integer, Integer> composed = f1.andThen(f2);
5 System.out.println(composed.apply(3)); // (3+1)*2 = 8
```

## 4. `Consumer<T>` 예제

```
1 Consumer<String> printer = s -> System.out.println("출력: " + s);
2 printer.accept("Hello"); // 출력: Hello
```

### `andThen()` 체이닝 가능

```
1 Consumer<String> c1 = s -> System.out.println("1: " + s);
2 Consumer<String> c2 = s -> System.out.println("2: " + s);
3 c1.andThen(c2).accept("Test");
```

## 5. `Supplier<T>` 예제

```
1 Supplier<Double> randomSupplier = () -> Math.random();
2 System.out.println(randomSupplier.get());
```

- 주로 초기화 지연, 랜덤 값, 캐시, 객체 생성 지연 등에 사용

## 6. Predicate<T> 예제

```
1 Predicate<String> isEmpty = s -> s.isEmpty();
2 System.out.println(isEmpty.test("")); // true
```

### 조합 메서드

메서드	의미
<code>and()</code>	AND 조건
<code>or()</code>	OR 조건
<code>negate()</code>	NOT 조건

```
1 Predicate<String> isEmpty = isEmpty.negate();
2 System.out.println(isEmpty.test("abc")); // true
```

## 7. 기본형 특화 인터페이스

인터페이스	기본형	설명
<code>IntFunction&lt;R&gt;</code>	<code>int → R</code>	
<code>ToIntFunction&lt;T&gt;</code>	<code>T → int</code>	
<code>IntSupplier</code> , <code>DoubleSupplier</code> 등	반환값만 기본형	
<code>IntPredicate</code> , <code>LongPredicate</code> 등	입력만 기본형	

예시:

```
1 IntPredicate isEven = x -> x % 2 == 0;
2 System.out.println(isEven.test(4)); // true
```

## 8. Bi계열 인터페이스

인터페이스	메서드	설명
<code>BiFunction&lt;T, U, R&gt;</code>	<code>R apply(T, U)</code>	두 값 → 한 결과
<code>BiConsumer&lt;T, U&gt;</code>	<code>void accept(T, U)</code>	두 값 소비
<code>BiPredicate&lt;T, U&gt;</code>	<code>boolean test(T, U)</code>	두 값 조건 판별

```
1 BiFunction<String, Integer, String> repeat = (s, n) -> s.repeat(n);
2 System.out.println(repeat.apply("Hi", 3)); // HiHiHi
```

## 9. UnaryOperator, BinaryOperator

◆ `UnaryOperator<T> = Function<T, T>`

```
1 | UnaryOperator<Integer> square = x -> x * x;
```

◆ `BinaryOperator<T> = (T, T) -> T`

```
1 | BinaryOperator<Integer> sum = (a, b) -> a + b;
```

 `reduce()` 와 공합 좋음:

```
1 | int result = List.of(1, 2, 3).stream().reduce(0, Integer::sum);
```

## 10. 실무 활용 예시

✓ Stream과 함께

```
1 | List<String> names = List.of("a", "bb", "ccc");
2 |
3 | List<Integer> lengths = names.stream()
4 |     .filter(s -> s.length() > 1)           // Predicate
5 |     .map(s -> s.length())                  // Function
6 |     .collect(Collectors.toList());         // Collector
```

✓ 정리 요약

역할	인터페이스	설명
$T \rightarrow R$	<code>Function&lt;T, R&gt;</code>	변환
$T \rightarrow \text{void}$	<code>Consumer&lt;T&gt;</code>	소비
$() \rightarrow T$	<code>Supplier&lt;T&gt;</code>	공급
$T \rightarrow \text{boolean}$	<code>Predicate&lt;T&gt;</code>	조건 판별
$T \rightarrow T$	<code>UnaryOperator&lt;T&gt;</code>	동일 타입 변환
$(T, T) \rightarrow T$	<code>BinaryOperator&lt;T&gt;</code>	동일 타입 연산



# java.lang.reflect: 리플렉션 API

## 1. 리플렉션(Reflection)이란?

런타임에 클래스/메서드/필드 등의 정보를 동적으로 조사하거나 조작할 수 있는 기능

- 일반적으로 컴파일 타임에 결정되는 구조를, 런타임에 제어
- 프레임워크 개발, 의존성 주입(DI), 애노테이션 처리, 직렬화, 테스트, ORM(JPA) 등에 핵심적으로 사용됨

## 2. 핵심 클래스들 (java.lang.reflect 패키지)

클래스	설명
<code>Class&lt;?&gt;</code>	클래스 정보의 루트. 메타데이터 접근에 사용
<code>Field</code>	멤버 변수에 대한 정보
<code>Method</code>	메서드에 대한 정보
<code>Constructor</code>	생성자에 대한 정보
<code>Modifier</code>	접근 제어자 정보 분석
<code>Parameter</code>	매개변수 정보

## 3. Class 객체 획득 방법

```
1 Class<?> clazz1 = Class.forName("com.example.MyClass"); // 완전한 경로
2 Class<?> clazz2 = MyClass.class;
3 MyClass obj = new MyClass();
4 Class<?> clazz3 = obj.getClass();
```

## 4. 필드(Field) 접근

```
1 Field field = clazz.getDeclaredField("name"); // private도 포함
2 field.setAccessible(true); // 접근 허용
3 field.set(obj, "NewValue"); // 값 설정
4 System.out.println(field.get(obj)); // 값 조회
```

- `getFields()` : public만
- `getDeclaredFields()` : private 포함

## ■ 5. 메서드(Method) 접근

```
1 Method method = clazz.getDeclaredMethod("sayHello", String.class);
2 method.setAccessible(true);
3 String result = (String) method.invoke(obj, "John");
```

- 오버로딩된 메서드는 정확한 매개변수 타입 명시해야 함
- 리턴값은 Object 타입 → 캐스팅 필요

## ■ 6. 생성자(Constructor) 접근

```
1 Constructor<MyClass> constructor = MyClass.class.getConstructor(String.class);
2 MyClass instance = constructor.newInstance("hi");
```

- private 생성자도 `setAccessible(true)` 로 접근 가능

## ■ 7. Modifier 클래스 활용

```
1 int mod = field.getModifiers();
2 System.out.println(Modifier.isPrivate(mod)); // true
3 System.out.println(Modifier.isStatic(mod)); // false
```

- 접근 제어자(public, private 등) 및 기타 속성(static, final 등)을 분석할 수 있음

## ■ 8. Parameter 정보

Java 8부터 `-parameters` 옵션을 켜야 이름이 보임

```
1 for (Parameter p : method.getParameters()) {
2     System.out.println(p.getName() + " : " + p.getType());
3 }
```

## ■ 9. 애노테이션과 리플렉션

```
1 if (clazz.isAnnotationPresent(MyAnnotation.class)) {
2     MyAnnotation anno = clazz.getAnnotation(MyAnnotation.class);
3     System.out.println(anno.value());
4 }
```

- 메서드, 필드에도 동일하게 사용 가능
- 런타임 유지 정책 (`@Retention(RetentionPolicy.RUNTIME)`) 필수

## 10. 프라이빗 멤버 접근 요약

대상	접근 방식
필드	<code>getDeclaredField()</code> → <code>setAccessible(true)</code>
메서드	<code>getDeclaredMethod()</code> → <code>setAccessible(true)</code>
생성자	<code>getDeclaredConstructor()</code> → <code>setAccessible(true)</code>

## 11. 실무 활용 사례

- 프레임워크 내부
  - Spring DI: `Field.set()` 을 통한 의존성 주입
  - JUnit: `@Test` 메서드 실행
  - JPA: `@Entity` 클래스의 필드 추출 및 매핑
- 라이브러리
  - Jackson: private 필드 접근하여 JSON 직렬화
  - Lombok: 코드 자동 생성에 리플렉션 활용

## 12. 리플렉션의 단점 및 주의사항

단점	설명
성능 저하	직접 호출보다 느림 (JIT 최적화 우회)
보안 취약	private 멤버 조작 가능 (보안 매니저 환경 주의)
유지보수 어려움	구조 변경 시 에러 발생 가능성 ↑

## ✓ 마무리 요약

기능	메서드
클래스 정보	<code>Class.forName()</code> , <code>getName()</code> , <code>getSuperclass()</code>
필드	<code>getDeclaredFields()</code> , <code>field.set/get()</code>
메서드	<code>getDeclaredMethods()</code> , <code>method.invoke()</code>
생성자	<code>getConstructor()</code> , <code>newInstance()</code>
접근 허용	<code>setAccessible(true)</code>
애노테이션	<code>isAnnotationPresent()</code> , <code>getAnnotation()</code>

# javax.xml, javax.json: XML/JSON 처리

## 1.1 DOM (Document Object Model) 방식

메모리에 전체 XML 구조를 트리로 로딩 후 조작

### 🔧 예제: XML 파일 읽기

```
1 <!-- sample.xml -->
2 <person>
3   <name>John</name>
4   <age>30</age>
5 </person>
```

```
1 DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
2 DocumentBuilder builder = factory.newDocumentBuilder();
3 Document doc = builder.parse(new File("sample.xml"));
4
5 Node nameNode = doc.getElementsByTagName("name").item(0);
6 System.out.println(nameNode.getTextContent()); // John
```

### 📌 특징

- 읽고 쓰기 모두 가능
- 전체 XML을 메모리에 올림 → 무겁고 느릴 수 있음
- `org.w3c.dom` API와 함께 사용

## 1.2 SAX (Simple API for XML)

이벤트 기반 파서, 빠르고 메모리 효율적 (읽기 전용)

```
1 SAXParserFactory factory = SAXParserFactory.newInstance();
2 SAXParser parser = factory.newSAXParser();
3
4 DefaultHandler handler = new DefaultHandler() {
5     public void startElement(String uri, String localName, String qName, Attributes
6         attributes) {
7         System.out.println("Start: " + qName);
8     }
9
10    public void characters(char[] ch, int start, int length) {
11        System.out.println("Text: " + new String(ch, start, length));
12    }
13
14    public void endElement(String uri, String localName, String qName) {
15        System.out.println("End: " + qName);
16    }
17};
```

```
18 parser.parse(new File("sample.xml"), handler);
```

🔴 특징:

- 읽기 전용, 빠름
- 콜백 구조로 구성

## 1.3 JAXB (Java Architecture for XML Binding)

XML ↔ Java 객체 매핑

🔧 예시

```
1 @XmlElement
2 public class Person {
3     public String name;
4     public int age;
5 }
```

```
1 // Unmarshalling (XML → Object)
2 JAXBContext ctx = JAXBContext.newInstance(Person.class);
3 Person p = (Person) ctx.createUnmarshaller().unmarshal(new File("person.xml"));
4
5 // Marshalling (Object → XML)
6 ctx.createMarshaller().marshal(p, System.out);
```

🔴 JAXB는 자바 9 이후 `jakarta.xml.bind` 로 이전됨

## 2. JSON 처리: `javax.json`

Java EE 7+ 표준 JSON 처리 API (JSR 353)

### 2.1 JSON 읽기

```
1 {
2     "name": "Alice",
3     "age": 25
4 }
```

```
1 InputStream is = new FileInputStream("data.json");
2 JsonReader reader = Json.createReader(is);
3 JsonObject obj = reader.readObject();
4
5 System.out.println(obj.getString("name")); // Alice
6 System.out.println(obj.getInt("age")); // 25
```

- 사용 클래스: `Json`, `JsonReader`, `JsonObject`

## 2.2 JSON 쓰기

```
1  JsonObject json = Json.createObjectBuilder()
2      .add("name", "Bob")
3      .add("age", 30)
4      .build();
5
6  JsonWriter writer = Json.createWriter(new FileOutputStream("output.json"));
7  writer.writeObject(json);
8  writer.close();
```

## 2.3 JSON 배열 다루기

```
1  JsonArray arr = Json.createArrayBuilder()
2      .add("apple")
3      .add("banana")
4      .add("cherry")
5      .build();
6
7  for (JsonValue value : arr) {
8      System.out.println(value.toString());
9  }
```

## 2.4 주요 인터페이스

인터페이스	설명
JsonObject	JSON 객체 구조 (Map 유사)
JsonArray	배열 구조
JsonReader	JSON 입력
JsonWriter	JSON 출력
JsonBuilderFactory, JsonParser	커스텀 빌더 및 스트리밍

## 3. javax.xml 과 javax.json 비교 요약

항목	XML ( javax.xml )	JSON ( javax.json )
표준 API	DOM, SAX, JAXB	JsonObject, JsonReader 등
사용성	복잡하고 구조적	간결하고 직관적
성능	SAX > DOM > JAXB	매우 빠름

항목	XML ( javax.xml )	JSON ( javax.json )
쓰기	지원 (JAXB 등)	<code>JsonWriter</code> , <code>ObjectBuilder</code>
Java EE 포함 여부	O	Java EE 7+ 이후 포함

## ✅ 추가로 실무에서는?

- JSON은 `Jackson`, `Gson`, `Moshi` 같은 외부 라이브러리를 더 많이 씀
- XML은 `JAXB`, `DOM4J`, `JAXP`, `StAX`, `XStream` 등이 널리 사용됨

# java.util.logging, SLF4J 등 로깅 API

## 1. 로깅(logging)이란?

시스템 실행 중의 상태나 이벤트를 기록하여 추적, 디버깅, 모니터링 등을 가능하게 하는 방법

- `System.out.println()` 보다 효율적이고 유연함
- 로그 수준, 출력 포맷, 대상(파일/콘솔/원격 서버) 등 조절 가능

## 2. java.util.logging (JUL: Java Util Logging)

자바 표준에 포함된 기본 로깅 API

### ✅ 주요 구성 요소

구성요소	설명
<code>Logger</code>	로그를 생성하는 객체
<code>Handler</code>	로그를 출력하는 수단 (콘솔, 파일 등)
<code>Formatter</code>	로그 출력 형식 지정
<code>Level</code>	로그 수준 ( <code>SEVERE</code> , <code>WARNING</code> , <code>INFO</code> , <code>CONFIG</code> , <code>FINE</code> 등 )

### 🔧 기본 사용 예시

```

1 import java.util.logging.*;
2
3 public class LoggingExample {
4     private static final Logger logger =
5         Logger.getLogger(LoggingExample.class.getName());
6
7     public static void main(String[] args) {
8         logger.setLevel(Level.INFO);
9
10        logger.severe("심각한 에러 발생");
11        logger.warning("경고 메시지");

```

```

11     logger.info("정보 메시지");
12     logger.fine("디버그 메시지"); // 출력되지 않음 (기본레벨보다 낮음)
13 }
14 }

```

## 파일로 로그 저장

```

1 FileHandler fileHandler = new FileHandler("app.log");
2 fileHandler.setFormatter(new SimpleFormatter());
3 logger.addHandler(fileHandler);

```

## 단점

- 설정이 복잡하고 유연성 부족
- 출력 형식이 제한적
- 실무에선 거의 안 씀

## 3. SLF4J (Simple Logging Facade for Java)

로깅 API 추상화 계층 – 로깅 구현체를 바꿔도 코드 수정 없이 사용 가능

## SLF4J는 인터페이스, 구현체는 Logback 또는 Log4j

```

1 import org.slf4j.Logger;
2 import org.slf4j.LoggerFactory;
3
4 public class App {
5     private static final Logger logger = LoggerFactory.getLogger(App.class);
6
7     public static void main(String[] args) {
8         logger.info("시작됨");
9         logger.warn("경고");
10        logger.error("에러 발생", new RuntimeException("예외"));
11    }
12 }

```

## 4. Logback – SLF4J의 대표 구현체

- 성능, 설정 유연성, XML 기반 구성 지원
- 실무에서 가장 많이 사용됨



## 🔧 예시: logback.xml

```
1 <configuration>
2   <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
3     <encoder>
4       <pattern>[%d{HH:mm:ss}] %-5level %logger{36} - %msg%n</pattern>
5     </encoder>
6   </appender>
7
8   <root level="info">
9     <appender-ref ref="STDOUT"/>
10  </root>
11 </configuration>
```

- %d, %level, %logger, %msg 등 포맷 설정 가능
- 로그 파일로도 저장 가능 (RollingFileAppender)

## ■ 5. 로그 수준 (공통)

레벨	의미
TRACE	매우 상세한 정보 (디버깅용)
DEBUG	개발 중 상세 로그
INFO	일반적인 실행 흐름 정보
WARN	주의가 필요한 상황
ERROR	예외나 치명적 오류

## ■ 6. SLF4J + Logback 설정 구조 요약

### 1. 의존성 추가 (Maven 예시)

```
1 <dependency>
2   <groupId>org.slf4j</groupId>
3   <artifactId>slf4j-api</artifactId>
4   <version>1.7.36</version>
5 </dependency>
6 <dependency>
7   <groupId>ch.qos.logback</groupId>
8   <artifactId>logback-classic</artifactId>
9   <version>1.2.11</version>
10 </dependency>
```

1. 코드에서는 SLF4J 인터페이스만 사용
2. 설정은 logback.xml 로 분리하여 관리

## 7. Spring Boot에서 로깅

- 기본 내장: SLF4J + Logback
- `application.yml`이나 `logback-spring.xml`로 설정 가능
- 실시간 로그 레벨 변경 가능 (`/actuator/loggers`)

```
1 logging:
2   level:
3     root: INFO
4     com.example.myapp: DEBUG
```

### ✓ 비교 요약

기능	<code>java.util.logging</code>	SLF4J + Logback
표준 API 포함	O	X
실무 활용	드물	매우 많음
확장성/유연성	낮음	높음
설정 방식	코드 기반	외부 설정 (XML, YAML)
로그 수준	제한적	세분화 (TRACE~ERROR)
포맷/출력 제어	제한적	매우 유연

## 보안 관련: `java.security`, `javax.crypto`

### 1. `java.security` 패키지 개요

자바의 기본 보안 인프라를 제공하는 패키지

주요 기능: 메시지 다이제스트, 서명, 키 생성, 인증서 처리 등

#### 1.1 메시지 다이제스트 (해시 함수)

입력 데이터를 일정 길이의 고정된 값으로 변환

```
1 import java.security.MessageDigest;
2
3 MessageDigest md = MessageDigest.getInstance("SHA-256");
4 byte[] hash = md.digest("hello".getBytes());
5
6 for (byte b : hash)
7     System.out.printf("%02x", b); // SHA-256 해시 출력
```

알고리즘	설명
MD5	빠르지만 취약
SHA-1	더 안전하지만 최근엔 취약 판정
SHA-256	현재 실무에서도 많이 사용

## 1.2 키 생성과 서명

개인키로 서명 → 공개키로 검증

```

1 // 키 쌍 생성
2 KeyPairGenerator keyGen = KeyPairGenerator.getInstance("RSA");
3 keyGen.initialize(2048);
4 KeyPair pair = keyGen.generateKeyPair();
5
6 // 서명
7 Signature signature = Signature.getInstance("SHA256withRSA");
8 signature.initSign(pair.getPrivate());
9 signature.update("message".getBytes());
10 byte[] signedData = signature.sign();
11
12 // 검증
13 signature.initVerify(pair.getPublic());
14 signature.update("message".getBytes());
15 boolean valid = signature.verify(signedData);
16 System.out.println("서명 검증 결과: " + valid);

```

## 1.3 인증서 및 키스토어

인증서 파일(.cer, .jks)을 사용해 공개키 관리

```

1 KeyStore ks = KeyStore.getInstance("JKS");
2 ks.load(new FileInputStream("keystore.jks"), "password".toCharArray());
3 Certificate cert = ks.getCertificate("myalias");
4 PublicKey publicKey = cert.getPublicKey();

```

## 2. javax.crypto 패키지 개요

암호화/복호화, 대칭/비대칭 암호화, 패스워드 기반 암호화(PBE) 등을 지원하는 고수준 API

## 2.1 대칭키 암호화 (AES 등)

```
1 KeyGenerator keyGen = KeyGenerator.getInstance("AES");
2 keyGen.init(128);
3 SecretKey secretKey = keyGen.generateKey();
4
5 Cipher cipher = Cipher.getInstance("AES");
6 cipher.init(Cipher.ENCRYPT_MODE, secretKey);
7 byte[] encrypted = cipher.doFinal("secret".getBytes());
8
9 cipher.init(Cipher.DECRYPT_MODE, secretKey);
10 byte[] decrypted = cipher.doFinal(encrypted);
11 System.out.println(new String(decrypted)); // secret
```

특징	내용
속도 빠름	대량의 데이터에 적합
키 분배 문제 존재	네트워크상 안전한 키 교환 필요

## 2.2 비대칭키 암호화 (RSA)

```
1 KeyPairGenerator keyGen = KeyPairGenerator.getInstance("RSA");
2 keyGen.initialize(2048);
3 KeyPair pair = keyGen.generateKeyPair();
4
5 Cipher cipher = Cipher.getInstance("RSA");
6 cipher.init(Cipher.ENCRYPT_MODE, pair.getPublic());
7 byte[] encrypted = cipher.doFinal("topsecret".getBytes());
8
9 cipher.init(Cipher.DECRYPT_MODE, pair.getPrivate());
10 byte[] decrypted = cipher.doFinal(encrypted);
11 System.out.println(new String(decrypted)); // topsecret
```

특징	내용
키 분리 (공개/개인)	키 분배가 쉽고 안전
처리 속도 느림	대량 데이터 암호화에는 부적합

## 2.3 비밀번호 기반 암호화 (PBE)

```
1 char[] password = "mypassword".toCharArray();
2 byte[] salt = "12345678".getBytes();
3
4 PBEKeySpec spec = new PBEKeySpec(password, salt, 65536, 128);
5 SecretKeyFactory skf = SecretKeyFactory.getInstance("PBKDF2WithHmacSHA256");
6 SecretKey key = skf.generateSecret(spec);
7
8 SecretKeySpec aesKey = new SecretKeySpec(key.getEncoded(), "AES");
```

PBE는 사용자의 패스워드를 기반으로 **암호화용 키를 안전하게 생성할 때** 사용돼.

## 2.4 CBC 모드, IV 사용

블록 암호 방식의 보안성을 높이기 위해 초기화 벡터(IV)를 사용

```
1 IvParameterSpec iv = new IvParameterSpec(new byte[16]); // 무작위 추천
2 Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
3 cipher.init(Cipher.ENCRYPT_MODE, secretKey, iv);
```

# 3. 실제 보안 적용 예시

시나리오	API 사용
비밀번호 저장	PBKDF2 + Salt + Base64
데이터베이스 암호화	AES 대칭 암호화
클라이언트 인증	RSA 서명 검증
네트워크 암호화	SSL (보통 javax.net.ssl 활용)
파일 서명	java.security.Signature

## 보안 API 계층 요약

API 계층	제공 기능
java.security	해시, 서명, 키 생성, 인증서, SecureRandom
javax.crypto	암호화/복호화, PBE, Cipher, SecretKey
javax.net.ssl	HTTPS, TLS/SSL 소켓 통신
java.security.cert	X.509 인증서 처리
java.security.spec	키 사양 (KeySpec 등)

API 계층	제공 기능
<code>java.security.interfaces</code>	RSA/DSA/ECDSA 전용 인터페이스

## ✔ 마무리 요약

기술	설명
MessageDigest	단방향 해시 함수
Signature	디지털 서명
Cipher	양방향 암호화 (AES, RSA 등)
KeyStore	키와 인증서 저장소
SecretKey / KeyPair	대칭/비대칭 키 객체
PBKDF2	비밀번호 기반 키 생성 (안전)
IV	CBC 모드 등에서 블록 암호를 더 안전하게