

Java의 정석

4판 Java 21

남궁성지음

[ebook – 샘플. 무료 공유] 자바의 정석 4판 Java 21 올컬러 2025. 7. 7 출시
seong.namkung@gmail.com

<https://cafe.naver.com/javachobostudy.cafe> – Q&A 게시판 및 자료실
<https://github.com/castello/javajungsuk4> – 소스코드 & 연습문제
<https://www.youtube.com/@MasterNKS> – 무료 동영상 강좌

도우출판

Foreword

개정 4판을 펴내며...

첫 번째 자바 책을 출간한지 22년이 넘었네요. 이 오랜 시간동안 늘 같은 자리에서 독자분들의 질문에 답변하며 유튜브에 무료 강의, 스터디, 세미나 등 많은 사람이 자바를 쉽게 잘 배울 수 있도록 노력해왔습니다. 저자가 본인 스스로 굳이 이런 글을 쓴다는 것이 내키지 않으나 인터넷에 독자를 혼혹하는 정보가 많아서 이 자리를 빌어 저자의 허심坦회한 생각을 한번 읽어주길 바랄 뿐입니다.

세상의 변화만큼이나 자바도 빠르게 성장해왔는데 몇년이 지나도 이 변화를 온전히 담아주는 책이 나오지 않아서 제대로 된 책을 써보자 다짐하고 4판의 집필을 시작하게 되었습니다. 반년이면 되겠지 싶었으나 어느새 1년이 훌쩍 넘어서야 세상에 나오게 되었습니다.

금전적인 면만 생각하면 대충 구색만 갖춰서 새로운 책으로 출판할 수 있었겠지만, 적어도 누구 하나는 그러지 않아야 하니까요. 누군가 책을 잘 집필해주면 저도 이제 이 무거운 짐을 내려놓고 훌가분해질텐데 이점이 늘 아쉽습니다. 그래도 저는 오랫동안 독자 여러분의 사랑을 많이 받아왔으니 그 보답을 한다는 마음으로 아직까지 버티고 있습니다.

최근 AI의 급격한 발달로 지식을 보다 쉽게 얻을 수 있게 되었습니다. 누구나 AI를 사용하지만, 사람의 능력이 약간만 차이나도 AI가 만들어내는 결과물의 격차가 큽니다. 그래서 사람이 습득해야하는 지식의 양과 질이 더 중요해지고 있습니다.

실무에서는 점점 높은 수준의 실력을 요구하고 있으나 독자들은 점점 쉬운 책을 원하고 그에 맞는 책이 많이 팔리고 있습니다. 정말로 쉬운 것도 아니고 그저 쉬워보이는 책이나 강좌로는 실력이 늘기 어렵습니다. 그에 비해 이 책은 다소 어렵게 느껴질 수도 있으나 중요한 내용을 빠짐없이 담았고, 대신 저자가 직접 답변해주고 강의 영상도 무료로 제공하여 혼자서도 학습할 수 있도록 돋고 있습니다.

이를 잘 활용하면 보다 효율적으로 높은 실력을 갖추실 수 있습니다. 저는 독자 여러분을 항상 묵묵히 응원하며, 늘 같은 자리에서 여러분의 질문을 기다리고 있습니다.

2025년 6월 14일

저자 남궁성
seong.namkung@gmail.com

Contents

Chapter 01

자바를 시작하기 전에

1. 자바(Java programming language)	002
1.1 자바란?	002
1.2 자바의 역사	003
1.3 자바언어의 특징	006
1.4 JVM(Java Virtual Machine)	008
2. 자바개발환경 구축하기	010
2.1 자바 개발도구(JDK)설치하기	010
2.2 인텔리제이(IntelliJ IDEA) 설치하기	020
3. 자바로 프로그램 작성하기	028
3.1 Hello.java	028
3.2 자주 발생하는 에러와 해결방법	031
3.3 자바 프로그램의 실행과정	032
3.4 주석(comment)	033
3.5 이 책으로 공부하는 방법	034

Chapter 02

변수(variable)

1. 변수(variable)	040
1.1 변수(variable)란?	040
1.2 변수의 선언과 초기화	040
1.3 변수의 명명규칙	045
2. 변수의 타입	047
2.1 기본형(primitive type)	048
2.2 상수와 리터럴(constant & literal)	050
2.3 형식화된 출력 – printf()	058
2.4 화면에서 입력받기 – Scanner	062
3. 진법	064
3.1 10진법과 2진법	064
3.2 비트(bit)와 바이트(byte)	065
3.3 8진법과 16진법	066
3.4 정수의 진법 변환	068
3.5 실수의 진법 변환	070
3.6 음수의 2진 표현 – 2의 보수법	072
4. 기본형(primitive type)	076
4.1 논리형 – boolean	076
4.2 문자형 – char	076
4.3 정수형 – byte, short, int, long	083
4.4 실수형 – float, double	089
5. 형변환	095
5.1 형변환(캐스팅, casting)이란?	095
5.2 형변환 방법	095
5.3 정수형 간의 형변환	096
5.4 실수형 간의 형변환	098

5.5 정수형과 실수형 간의 형변환	101
5.6 자동 형변환	103

Chapter 03 연산자(operator)

1. 연산자(operator)	108
1.1 연산자와 피연산자	108
1.2 식(式)과 대입 연산자	108
1.3 연산자의 종류	108
1.4 연산자의 우선순위와 결합규칙	110
1.5 산술 변환(usual arithmetic conversion)	113
2. 단항 연산자	115
2.1 증감 연산자 ++, --	115
2.2 부호 연산자 +, -	118
3. 산술 연산자	119
3.1 사칙 연산자 +, -, *, /	119
3.2 나머지 연산자 %	130
4. 비교 연산자	131
4.1 대소비교 연산자 <, >, <=, >=	131
4.2 등가비교 연산자 ==, !=	131
5. 논리 연산자	136
5.1 논리 연산자 &&, , !	136
5.2 비트 연산자 &, , ^, ~, <<, >>	143
6. 그 외의 연산자	152
6.1 조건 연산자 ?:	152
6.2 대입 연산자 =, op=	154

Chapter 04 조건문과 반복문

1. 조건문 – if, switch	158
1.1 if문	158
1.2 if–else문	162
1.3 if–else if문	163
1.4 중첩 if문	166
1.5 switch문	168
2. 반복문 – for, while, do–while	180
2.1 for문	180
2.2 while문	191
2.3 do–while문	197
2.4 break문	199
2.5 continue문	200
2.6 이름 붙은 반복문	202

Contents

Chapter 05

배열(array)

1. 배열(array)	206
1.1 배열(array)이란?	206
1.2 배열의 선언과 생성	206
1.3 배열의 길이와 인덱스	208
1.4 배열의 초기화	213
1.5 배열의 복사	216
1.6 배열의 활용	220
2. String 배열	230
2.1 String 배열의 선언과 생성	230
2.2 String 배열의 초기화	230
2.3 char 배열과 String 클래스	233
2.4 커맨드 라인을 통해 입력받기	236
3. 다차원 배열	238
3.1 2차원 배열의 선언과 인덱스	238
3.2 2차원 배열의 초기화	238
3.3 가변 배열	243
3.4 다차원 배열의 활용	244

Chapter 06

객체지향 프로그래밍 I

1. 객체지향언어	254
1.1 객체지향언어의 역사	254
1.2 객체지향언어	254
2. 클래스와 객체	255
2.1 클래스와 객체의 정의와 용도	255
2.2 객체와 인스턴스	256
2.3 객체의 구성요소 – 속성과 기능	257
2.4 인스턴스의 생성과 사용	258
2.5 객체 배열	264
2.6 클래스의 또 다른 정의	266
3. 변수와 메서드	270
3.1 선언위치에 따른 변수의 종류	270
3.2 클래스 변수와 인스턴스 변수	271
3.3 메서드	273
3.4 메서드의 선언과 구현	276
3.5 메서드의 호출	278
3.6 return문	282
3.7 JVM의 메모리 구조	285
3.8 기본형 매개변수와 참조형 매개변수	288
3.9 참조형 반환타입	292
3.10 재귀호출(recursive call)	294
3.11 클래스 메서드(static 메서드)와 인스턴스 메서드	301
3.12 클래스 멤버와 인스턴스 멤버 간의 참조와 호출	304

4. 오버로딩(overloading)	307
4.1 오버로딩이란?	307
4.2 오버로딩의 조건	307
4.3 오버로딩의 예	307
4.4 오버로딩의 장점	309
4.5 가변인자(varargs)와 오버로딩	311
5. 생성자(constructor)	315
5.1 생성자란?	315
5.2 기본 생성자(default constructor)	316
5.3 매개변수가 있는 생성자	318
5.4 생성자에서 다른 생성자 호출하기 – this(), this	319
5.5 생성자를 이용한 인스턴스의 복사	322
6. 변수의 초기화	324
6.1 변수의 초기화	324
6.2 명시적 초기화(explicit initialization)	325
6.3 초기화 블록(initialization block)	326
6.4 멤버변수의 초기화 시기와 순서	328

Chapter 07

객체지향 프로그래밍 II

1. 상속(inheritance)	334
1.1 상속의 정의와 장점	334
1.2 클래스간의 관계 – 포함 관계	340
1.3 클래스간의 관계 결정하기	341
1.4 단일상속(single inheritance)	347
1.5 Object클래스 – 모든 클래스의 조상	349
2. 오버라이딩(overriding)	351
2.1 오버라이딩이란?	351
2.2 오버라이딩의 조건	352
2.3 오버로딩 vs. 오버라이딩	353
2.4 super	354
2.5 super() – 조상 클래스의 생성자	356
3. package와 import	360
3.1 패키지(package)	360
3.2 패키지의 선언	361
3.3 import문	364
3.4 import문의 선언	364
3.5 static import문	366
4. 제어자(modifier)	368
4.1 제어자란?	368
4.2 static – 클래스의, 공통적인	368
4.3 final – 마지막의, 변경될 수 없는	369
4.4 abstract – 추상의, 미완성의	371
4.5 접근 제어자(access modifier)	372
4.6 제어자(modifier)의 조합	377

Contents

5. 다형성(polymorphism)	378
5.1 다형성이란?	378
5.2 참조변수의 형변환	380
5.3 instanceof 연산자	386
5.4 참조변수와 인스턴스의 연결	394
5.5 매개변수의 다형성	397
5.6 여러 종류의 객체를 배열로 다루기	400
6. 추상 클래스(abstract class)	405
6.1 추상 클래스란?	405
6.2 추상 메서드(abstract method)	405
6.3 추상 클래스의 작성	407
7. 인터페이스(interface)	411
7.1 인터페이스란?	411
7.2 인터페이스의 작성	411
7.3 인터페이스의 상속	412
7.4 인터페이스의 구현	412
7.5 인터페이스를 이용한 다중 상속	415
7.6 인터페이스를 이용한 다형성	417
7.7 인터페이스의 장점	420
7.8 인터페이스의 이해	426
7.9 디폴트 메서드, static메서드, private메서드	430
8. 내부 클래스(inner class)	434
8.1 내부 클래스란?	434
8.2 내부 클래스의 종류와 특징	435
8.3 내부 클래스의 선언	435
8.4 내부 클래스의 제어자와 접근성	436
8.5 익명 클래스(anonymous class)	441

Chapter 08

예외 처리(exception handling)

1. 예외 처리(exception handling)	444
1.1 프로그램 오류	444
1.2 예외 클래스의 계층구조	445
1.3 예외 처리하기 – try-catch문	446
1.4 try-catch문에서의 흐름	449
1.5 예외의 발생과 catch블럭	450
1.6 예외 발생시키기	454
1.7 메서드에 예외 선언하기	457
1.8 finally블럭	464
1.9 자동 자원 반환 – try-with-resources문	466
1.10 사용자정의 예외 만들기	469
1.11 예외 되던지기(exception re-throwing)	472
1.12 연결된 예외(chained exception)	474

Chapter 09

java.lang패키지와 유용한 클래스

1. java.lang패키지	480
1.1 Object클래스	480
1.2 String클래스	494
1.3 StringBuffer와 StringBuilder	508
1.4 Math클래스	514
1.5 래퍼(wrapper) 클래스	521
2. 유용한 클래스	526
2.1 java.util.Objects	526
2.2 java.util.Random	530
2.3 정규식(regular expression) – java.util.regex	535
2.4 java.util.Scanner	540
2.5 java.util.StringTokenizer	543
2.6 java.math.BigInteger	548
2.7 java.math.BigDecimal	551

Chapter 10

날짜와 시간 & 형식화

1. 날짜와 시간	558
1.1 Calendar와 Date	558
2. 형식화 클래스	570
2.1 DecimalFormat	570
2.2 SimpleDateFormat	574
2.3 ChoiceFormat	578
2.4 MessageFormat	579
3. java.time패키지	582
3.1 java.time패키지의 핵심 클래스	582
3.2 LocalDate와 LocalTime	585
3.3 Instant	590
3.4 LocalDateTime과 ZonedDateTime	591
3.5 TemporalAdjusters	595
3.6 Period와 Duration	597
3.7 파싱과 포맷	602

Chapter 11

컬렉션 프레임워크

1. 컬렉션 프레임워크(collections framework)	608
1.1 컬렉션 프레임워크의 핵심 인터페이스	608
1.2 ArrayList	615
1.3 LinkedList	626
1.4 Stack과 Queue	634

Contents

1.5 Iterator, ListIterator, Enumeration	644
1.6 Arrays	654
1.7 Comparator와 Comparable	658
1.8 HashSet	661
1.9 TreeSet	668
1.10 HashMap과 Hashtable	674
1.11 TreeMap	684
1.12 Properties	688
1.13 Collections	694
1.14 컬렉션 클래스 정리 & 요약	699

Chapter 12

모던 자바 기능(new Java features)

1. 지네릭스(generics)	702
1.1 지네릭스란?	702
1.2 지네릭 클래스의 선언	703
1.3 지네릭 클래스의 객체 생성과 사용	706
1.4 제한된 지네릭 클래스	709
1.5 와일드 카드	711
1.6 지네릭 메서드	717
1.7 지네릭 타입의 형변환	720
1.8 지네릭 타입의 제거	722
2. 열거형(enums)	724
2.1 열거형이란?	724
2.2 열거형의 정의와 사용	725
2.3 열거형에 멤버 추가하기	728
2.4 열거형의 이해	731
3. 애너테이션(annotation)	735
3.1 애너테이션이란?	735
3.2 표준 애너테이션	736
3.3 메타 애너테이션	744
3.4 애너테이션 타입 정의하기	748
4. 레코드(record)	754
4.1 레코드란?	754
4.2 레코드의 특징	755
4.3 레코드의 중첩	760
4.4 지네릭 레코드	762
4.5 레코드와 애너테이션	764
5. 실드 클래스(sealed class)	766
5.1 실드 클래스란?	766
5.2 실드 클래스의 제약 조건	767
5.3 실드 클래스와 switch식	769
6. 모듈(module)	774
6.1 모듈이란?	774
6.2 모듈 설명자 – module-info.java	776

6.3 이름없는 모듈과 java.base모듈	779
6.4 전이적 의존성과 순환 의존성	786
6.5 모듈의 컴파일과 실행	788
6.6 자동 모듈	793

Chapter **13****쓰레드(thread)**

1. 쓰레드	796
1.1 프로세스와 쓰레드?	796
1.2 쓰레드의 구현과 실행	798
1.3 start()와 run()	802
1.4 싱글쓰레드와 멀티쓰레드	806
1.5 쓰레드의 우선순위	812
1.6 쓰레드 그룹(thread group)	815
1.7 데몬 쓰레드(daemon thread)	818
1.8 쓰레드의 실행제어	822
2. 쓰레드의 동기화	841
2.1 synchronized를 이용한 동기화	841
2.2 wait()과 notify()	845
2.3 Lock과 Condition을 이용한 동기화	853
2.4 volatile	860
2.5 fork & join 프레임워크	862
3. 가상 쓰레드(virtual thread)	867
3.1 가상 쓰레드란?	867
3.2 가상 쓰레드의 생성과 사용	868
3.3 가상 쓰레드의 특징	869
3.4 플랫폼 쓰레드와 가상 쓰레드	871
3.5 가상 쓰레드의 상태	878
3.6 가상 쓰레드 작성시 주의사항	884
3.7 Continuation과 StackChunk	885
4. Executor와 ExecutorService	889
4.1 Executor	889
4.2 ThreadFactory	890
4.3 ExecutorService	892
4.4 쓰레드 풀(thread pool)	898
4.5 Future	903
4.6 CompletableFuture	913

Chapter **14****람다와 스트림**

1. 람다식(lambda Expression)	928
1.1 람다식이란?	928
1.2 람다식 작성하기	929
1.3 함수형 인터페이스(functional interface)	931
1.4 java.util.function패키지	936

Contents

1.5 Function의 합성과 Predicate의 결합	942
1.6 메서드 참조	946
2. 스트림(stream)	948
2.1 스트림이란?	948
2.2 스트림 만들기	953
2.3 스트림의 중간연산	958
2.4 Optional<T>와 OptionalInt	971
2.5 스트림의 최종연산	976
2.6 collect()	980
2.7 Collector구현하기	997
2.8 스트림의 변환	1000

Chapter 15

입출력(I/O)

1. 자바에서의 입출력	1004
1.1 입출력이란?	1004
1.2 스트림(stream)	1004
1.3 바이트 기반 스트림 – InputStream, OutputStream	1005
1.4 보조 스트림	1007
1.5 문자 기반 스트림 – Reader, Writer	1008
2. 바이트 기반 스트림	1010
2.1 InputStream과 OutputStream	1010
2.2 ByteArrayInputStream과 ByteArrayOutputStream	1012
2.3 FileInputStream과 FileOutputStream	1016
3. 문자 기반의 보조 스트림	1019
3.1 FilterInputStream과 FilterOutputStream	1019
3.2 BufferedInputStream과 BufferedOutputStream	1020
3.3 DataInputStream과 DataOutputStream	1023
3.4 SequenceInputStream	1029
3.5 PrintStream	1031
4. 문자 기반 스트림	1035
4.1 Reader와 Writer	1035
4.2 FileReader와 FileWriter	1037
4.3 PipedReader와 PipedWriter	1039
4.4 StringReader와 StringWriter	1041
5. 문자 기반의 보조 스트림	1042
5.1 BufferedReader와 BufferedWriter	1042
5.2 InputStreamReader와 OutputStreamWriter	1043
6. 표준 입출력과 File	1045
6.1 표준 입력력 – System.in, System.out, System.err	1045
6.2 표준 입력력의 대상변경 – setIn(), setOut(), setErr()	1047
6.3 RandomAccessFile	1049
6.4 File	1053
7. 직렬화(serialization)	1072
7.1 직렬화란?	1072

7.2 ObjectInputStream과 ObjectOutputStream	1073
7.3 직렬화 가능한 클래스 만들기 – Serializable, transient	1075
7.4 직렬화 가능한 클래스의 버전관리	1081

Chapter **16**

네트워킹(networking)

1. 네트워킹(networking)	1084
1.1 클라이언트/서버(client/server)	1084
1.2 IP주소(IP address)	1086
1.3 InetAddress	1087
1.4 URL과 URI	1089
1.5 URLConnection	1092
2. 소켓 프로그래밍	1097
2.1 TCP와 UDP	1097
2.2 TCP소켓 프로그래밍	1098
2.3 UDP소켓 프로그래밍	1116

Memo

Java

Programming
Language

Chapter 01

자바를 시작하기 전에

getting started with Java

1. 자바(Java Programming Language)

1.1 자바란?

자바는 썬 마이크로시스템즈(Sun Microsystems, 이하 썬)에서 개발하여 1996년 1월에 공식적으로 발표한 객체지향 프로그래밍 언어이며, 추후에 함수형 프로그래밍 기능이 추가되었다. 전세계적으로 가장 많이 사용되는 프로그래밍 언어 중의 하나로 약 30년동안 꾸준히 발전해왔으며 쓰이지 않는 곳을 찾기가 힘들정도로 폭넓은 분야에서 사용된다.

좁은 의미에서의 자바는 단순히 프로그래밍 언어지만, 넓은 의미에서 자바는 프로그래밍 언어 뿐만 아니라 관련된 여러 소프트웨어와 명세(specification)를 포함한다.

자바는 특정 플랫폼에 종속되지 않는 소프트웨어를 개발하고 배포하는데 필요한 모든 것을 제공한다. 그 덕분에 임베디드 시스템, 모바일 기기, 기업용 서버, 게임, 빅데이터, 인공지능에 이르기까지 아주 널리 쓰이고 있다.

자바는 SE(Standard Edition), ME(Micro Edition), EE(Enterprise Edition) 등 여리가지 종류가 있으며 대부분의 경우 자바는 ‘Java SE’를 의미한다.

| 참고 | Java EE는 2017년에 Eclipse재단으로 이전되면서 Jakarta EE로 이름이 변경되었다.



널리 사용되는 만큼 신중한 기능 개선으로 발전이 더디다는 평을 받아왔지만, 2017년 이후로 업데이트가 빨라져서 6개월마다 새로운 버전이 출시된다. 앞으로 최소한 몇년간은 자바는 기업 환경에서 가장 많이 사용되는 프로그래밍 언어의 지위를 내어주지 않을 것으로 보인다.

모던 프로그래밍 언어에서 가장 핵심적인 것은 객체지향 개념과 함수형 개념인데, 자바로 작성된 객체지향 개념과 관련된 좋은 자료가 많기 때문에 객체지향 개념을 배우기에 자바만한 언어가 없다. 빅데이터에서 많이 쓰이는 함수형 프로그래밍 언어인 스칼라(Scala)도 자바에서 발전된 것으로 자바를 잘 배워두면 여러 언어로 쉽게 확장해 나갈 수 있다.

1.2 자바의 역사

자바의 역사는 1991년에 썬의 엔지니어들에 의해서 고안된 오크(Oak)라는 언어에서부터 시작되었다. 원래 목표는 가전제품에 탑재될 소프트웨어를 만드는 것이었는데, 객체지향 언어인 C++을 확장하려다가 부족함을 느껴서 C++의 단점을 보완한 새로운 언어인 Oak를 개발하기로 결정하였다.

인터넷이 등장하자 운영체제에 독립적인 Oak가 이에 적합하다고 판단하여 개발 방향을 인터넷에 맞게 바꾸면서 이름을 자바(Java)로 변경하고, 1996년 1월에 자바의 정식 버전을 발표했다. 그 당시만 해도 자바로 작성된 애플릿(Applet)은 웹페이지에 사운드와 애니메이션 등의 멀티미디어적인 요소들을 제공할 수 있는 유일한 방법이었기 때문에 많은 인기를 얻고 단 기간에 많은 사용자층을 확보할 수 있었다.

이후에 애플릿이 매크로 미디어사의 플래시(flash)에게 밀려서 자바의 인기가 줄어들었으나 1990년 말에 웹의 폭발적인 성장으로 자바의 인기가 다시 급상승하였다. 대규모의 서버 애플리케이션의 개발에 자바가 사용됨으로써 실무에서 탄탄한 입지를 확보하게 되었다. 그러다가 2008년에 모바일 운영체제인 안드로이드에 자바가 사용되면서 자바의 활용 분야가 더욱 확대되면서 자바가 쓰이지 않는 곳이 없다고 할 정도가 되었다.

썬에서 오라클로

썬은 자바와 MySQL데이터베이스등으로 오픈소스 발전에 큰 기여를 해왔으나, 2000년대 후반에 리눅스의 발전과 닷컴 버블의 붕괴로 주 수익원이던 하드웨어 판매의 급락과 수익이 낮은 오픈소스의 특성으로 경영란에 빠지게 되었다.

결국 데이터베이스의 1인자 오라클(Oracle)이 2010년에 썬을 인수함으로써 자바와 MySQL데이터베이스를 확보하게 되었다. 오라클이 오픈소스의 수익성을 강화하면서 그 동안 무료로 사용해 왔던 자바의 유료화가 우려되었으나 기업이나 상업적 용도가 아니면 여전히 자바는 무료로 사용할 수 있다.

Oracle JDK와 OpenJDK

오라클은 자바 개발도구인 JDK(Java Development Kit)의 대부분을 Oracle OpenJDK라는 오픈소스로 jdk.java.net 사이트에서 공개하고 있으며, 이를 기반으로 여러 기업이 자신만의 JDK를 오픈소스로 개발해서 공개하고 있다.

| 참고 | 오픈소스인 OpenJDK는 조건없이 무료이고 소스가 모두 공개되어있다. <https://github.com/openjdk/jdk>

배포판	공급자	설명
Oracle OpenJDK	Oracle	오리클이 제공하는 OpenJDK의 공식 배포판. 6개월만 지원
Eclipse Temurin	Eclipse Adoptium	다양한 플랫폼을 지원하며 무료로 LTS 버전의 장기 지원
Amazon Corretto	Amazon	AWS 서비스와 통합에 적합하며 무료로 LTS 버전의 장기 지원
Azul Zulu	Azul Systems	다양한 플랫폼을 지원하며 무료로 LTS 버전의 장기 지원
Red Hat OpenJDK	Red Hat	대규모 기업 환경(RHEL)에서 안정성과 호환성이 뛰어남.

▲ 표1-1 OpenJDK의 종류

[ebook – 샘플. 무료 공유] 자바의 정석 4판 Java 21 올컬러 2025. 7. 7 출시 seong.namkung@gmail.com

오픈소스인 Oracle OpenJDK와 달리, Oracle JDK는 오픈 소스가 아니며 상업용 목적이 아닌 개인 사용자에게만 무료이다. Oracle OpenJDK는 Oracle JDK가 제공하는 JMC(Java Mission Control)와 JFR(Java Flight Recorder) 등의 상용도구가 포함되지 않는다는 점을 제외하면 Oracle JDK와 거의 동일하다.

JDK의 장기 지원 정책 – LTS, Long Term Support

기존에는 JDK의 발표 후 3년까지 보안 업데이트와 버그 수정이 무료로 지원되었으나 JDK 8부터는 LTS로 지정된 특정 버전만 ‘장기 지원(최소 8년)’을 제공하고, LTS가 아닌 버전은 다음 버전이 나오는 6개월 후면 지원이 종료된다.

학습 목적인 경우는 LTS버전이 아니어도 상관없으나, 상용 제품이나 기업 환경에서는 반드시 LTS버전의 JDK를 선택해야 한다.

| 참고 | Oracle JDK와 달리 Oracle OpenJDK는 LTS를 지원 안한다. LTS를 지원하는 OpenJDK가 필요하면 표1-1을 참고.

자바의 새로운 기능 – JDK 8 ~ 21

집필 시점인 2025년을 기준으로 JDK 21이 최신 LTS버전이고, 아래에 JDK 8부터 JDK 21까지 새로 추가된 주요 기능의 목록을 정리하였다. 자세한 내용은 목록에 적힌 책의 페이지를 참고하자.

먼저 가장 중요한 기능 몇가지를 요약하면, JDK 8에서 람다식과 스트림 API가 추가되면서 함수형 프로그래밍이 가능해졌다는 것과 JDK 9에서 G1(Garbage First)이라는 이름의 가비지 컬렉터(Garbage Collector)가 새로 추가되어 자동으로 메모리를 관리해주는 기능이 획기적으로 개선되었다. 그리고 JDK 9부터 추가된 모듈 시스템 덕분에 JDK가 모듈 단위로 잘 정리되어 6개월마다 새로운 버전을 출시하며 빠르게 변화할 수 있게 되었다.

| 참고 | JDK의 버전은 1.x 형식으로 표기하였으나 JDK 1.5부터 JDK 5와 같이 JDK x의 형식으로 변경되었다.

JDK 8 – 2014.3 LTS

- 함수형 프로그래밍 지원 – 람다식 & 스트림 API
- 새로운 날짜 및 시간 API: java.time패키지
- Optional 클래스: NullPointerException 방지

JDK 9 – 2017.9

- 모듈 시스템(JEP 261, p.774)
- 간결한 문자열(compact strings, JEP 254 p.507)
- 인터페이스에 private method 허용(p.432)
- G1을 기본 가비지 컬렉터(GC)로 변경
- REPL(Read–Eval–Print–Loop) 도구로 jshell을 제공

JDK 10 – 2018.3

- 지역 변수 타입 추론(var, JEP 286, p.54)

[ebook – 샘플. 무료 공유] 자바의 정석 4판 Java 21 올컬러 2025. 7. 7 출시 seong.namkung@gmail.com

JDK 11 – 2018.9 LTS

- HTTP Client API: 현대적인 비동기 HTTP 요청 지원
- String클래스에 메서드 추가: isBlank, lines, repeat 등(p.498)
- var를 Lambda식에 사용 가능.(JEP 323, p.929)

JDK 14 – 2020.3

- switch식(switch expressions, JEP 361, p.178)

JDK 15 – 2020.9

- 텍스트 블럭(JEP 378, p.56)

JDK 16 – 2021.3

- 레코드(records, JEP 395, p.754)
- instanceof를 위한 패턴매칭(JEP 394, p.390)

JDK 17 – 2021.9 LTS

- 실드 클래스(sealed classes, JEP 409, p.766)

JDK 18 – 2022.3

- 간단한 웹서버 제공(/bin/jwebserver, JEP 408)
- 기본 인코딩을 UTF-8로 변경(JEP 400, p.507)

JDK 21 – 2023.9 LTS

- 가상 쓰레드 (JEP 444, p.867)
- switch문을 위한 패턴매칭(JEP 441, p.769)
- 레코드 패턴(JEP 440, p.759)
- Collections Framework에 Sequenced Collections 추가 (JEP 431, p.614)

위의 목록에서 프리뷰 기능(preview features)은 제외하였다. 프리뷰 기능은 새로운 실험적 기능을 개발자의 피드백을 받기 위해 미리보기 형식으로 제공하는 것으로 향후 자바 버전에서 정식 기능이 되거나 폐기될 수 있다.

프리뷰 기능을 사용하려면 컴파일 할 때와 실행할 때와 실행할 때 아래와 같이 별도의 옵션을 지정해야 한다.

```
c:\jdk21\ch01>javac --enable-preview --release 21 PreviewTest.java
c:\jdk21\ch01>java --enable-preview PreviewTest
```

| 참고 | 버전별로 추가 및 변경된 사항에 대한 보다 자세한 내용은 아래의 링크에서 확인할 수 있다.

<https://docs.oracle.com/en/java/javase/21/language/java-language-changes-release.html>

자바 개선 제안 제도 – JEP, Java Enhancement Proposal

JDK 8부터 도입된 ‘자바 개선 제안 제도(JEP)’로, 자바에 새로운 기능의 추가나 기존 기능을 개선을 제안하는 공식 문서이다.

JEP마다 고유 번호가 있으며, OpenJDK 커뮤니티(openjdk.org)를 통해 논의, 승인, 구현 과정을 거친다. 차기 JDK버전에 어떤 JEP들을 포함시킬지 결정한다.

| 참고 | <https://openjdk.org/jeps/0>에서 JEP의 목록을 확인할 수 있다.

OpenJDK는 오픈소스답게 OpenJDK 커뮤니티와 JEP를 통해 Java 플랫폼의 발전 방향을 개발자들과 함께 고민하고 결정하며, 개발자들에게 Java의 미래를 이해할 수 있게 한다.

| 참고 | JEP와 유사한 JSR(Java Spec Request)도 있는데, JSR은 자바의 표준을 위한 제안으로 JEP보다 상위 개념으로 서로 연관되어 있다. 예를 들어 JSR 376은 JEP 261과 JEP 282에 의해 구현되었다.

1.3 자바언어의 특징

자바는 최근에 발표된 언어답게 기존의 다른 언어에는 없는 많은 장점을 가지고 있다. 그 중 대표적인 몇 가지에 대해서 알아보도록 하자.

1. 운영체제에 독립적이다.

기존의 언어는 한 운영체제에 맞게 개발된 프로그램을 다른 종류의 운영체제에 적용하기 위해서는 많은 노력이 필요하였지만, 자바에서는 더 이상 그런 노력을 하지 않아도 된다. 이것은 일종의 애뮬레이터인 자바가상머신(JVM)을 통해서 가능한 것인데, 자바 응용프로그램은 운영체제나 하드웨어가 아닌 JVM하고만 통신하고 JVM이 자바 응용프로그램으로부터 전달받은 명령을 해당 운영체제가 이해할 수 있도록 변환하여 전달한다. 자바로 작성된 프로그램은 운영체제에 독립적이지만 JVM은 운영체제에 종속적이어서 씬에서는 여러 운영체제에 설치할 수 있는 서로 다른 버전의 JVM을 제공하고 있다.

그래서 자바로 작성된 프로그램은 운영체제와 하드웨어에 관계없이 실행 가능하며 이것을 ‘한번 작성하면, 어디서나 실행된다.(Write once, run anywhere)’고 표현하기도 한다.

2. 객체지향 언어이자 함수형 언어이다.

자바는 프로그래밍의 대세로 자리 잡은 객체지향 프로그래밍언어(object-oriented programming language) 중의 하나로 객체지향개념의 특징인 상속, 캡슐화, 다형성이 잘 적용된 순수한 객체지향언어라는 평가를 받고 있다. JDK 8부터 함수형 프로그래밍을 지원하고 있어서 최신 변화의 흐름에 맞춰 지속적으로 성장해 가고 있다.

3. 비교적 배우기 쉽다.

자바의 연산자와 기본구문은 C++에서, 객체지향관련 구문은 스몰톡(small talk)이라는 객체지향언어에서 가져왔다. 이 들 언어의 장점을 취하면서 복잡하고 불필요한 부분은 과

감히 제거하여 단순화함으로서 쉽게 배울 수 있으며, 간결하고 이해하기 쉬운 코드를 작성할 수 있도록 하였다. 객체지향언어의 특징인 재사용성과 유지보수의 용이성 등의 많은 장점에도 불구하고 배우기가 어렵기 때문에 많은 사용자층을 확보하지 못했으나 자바의 간결하면서도 명료한 객체지향적 설계는 사용자들이 객체지향개념을 보다 쉽게 이해하고 활용할 수 있도록 하여 객체지향 프로그래밍의 저변확대에 크게 기여했다.

4. 자동 메모리 관리(Garbage Collection)

자바로 작성된 프로그램이 실행되면, 가비지 컬렉터(garbage collector)가 자동적으로 메모리를 관리해주기 때문에 프로그래머는 메모리를 따로 관리 하지 않아도 된다. 가비지 컬렉터가 없다면 프로그래머가 사용하지 않는 메모리를 체크하고 반환하는 일을 수동적으로 처리해야할 것이다. 자동으로 메모리를 관리한다는 것이 다소 비효율적인 면도 있지만, 프로그래머가 보다 프로그래밍에 집중할 수 있도록 도와준다.

| 참고 | JDK 9부터 성능이 크게 향상된 G1이 기본 가비지 컬렉터가 되었다. 실행시 옵션으로 가비지 컬렉터를 변경 가능

5. 네트워크와 분산처리를 지원한다.

인터넷과 대규모 분산환경을 염두에 둔 까닭인지 풍부하고 다양한 네트워크 프로그래밍 라이브러리(Java API)를 통해 비교적 짧은 시간에 네트워크 관련 프로그램을 쉽게 개발 할 수 있도록 지원한다.

6. 멀티쓰레드를 지원한다.

일반적으로 멀티쓰레드(multi-thread)의 지원은 사용되는 운영체제에 따라 구현방법도 상이하며, 처리 방식도 다르다. 그러나 자바에서 개발되는 멀티쓰레드 프로그램은 시스템과는 관계없이 구현가능하며, 관련된 라이브러리(Java API)가 제공되므로 구현이 쉽다. 그리고 여러 쓰레드에 대한 스케줄링(scheduling)을 자바 인터프리터가 담당하게 된다.

| 참고 | JDK 21부터 가상 쓰레드(virtual thread)가 추가되어 자바로 고성능, 고처리량의 서버를 만들 수 있게 되었다.

7. 동적 로딩(Dynamic Loading)을 지원한다.

보통 자바로 작성된 애플리케이션은 여러 개의 클래스로 구성되어 있다. 자바는 동적 로딩을 지원하기 때문에 실행 시에 모든 클래스가 로딩되지 않고 필요한 시점에 클래스를 로딩하여 사용할 수 있다는 장점이 있다. 그 외에도 일부 클래스가 변경되어도 전체 애플리케이션을 다시 컴파일하지 않아도 되며, 애플리케이션의 변경사항이 발생해도 비교적 적은 작업만으로도 처리할 수 있는 유연한 애플리케이션을 작성할 수 있다.

1.4 JVM(Java Virtual Machine)

JVM은 ‘Java virtual machine’을 줄인 것으로 직역하면 ‘자바를 실행하기 위한 가상 기계’라고 할 수 있다. 가상 기계라는 말이 좀 어색하겠지만 영어권에서는 컴퓨터를 머신(machine)이라고도 부르기 때문에 ‘머신’이라는 용어대신 ‘컴퓨터’를 사용해서 ‘자바를 실행하기 위한 가상 컴퓨터’라고 이해하면 좋을 것이다.

‘가상 기계(virtual machine)’는 소프트웨어로 구현된 하드웨어를 뜻하는 넓은 의미의 용어이며, 컴퓨터의 성능이 향상됨에 따라 점점 더 많은 하드웨어들이 소프트웨어화되어 컴퓨터 속으로 들어오고 있다. 그 예로는 TV와 비디오를 소프트웨어화한 윈도우 미디어 플레이어라던가, 오디오 시스템을 소프트웨어화한 윈앰프(winamp) 등이 있다.

이와 마찬가지로 ‘가상 컴퓨터(virtual computer)’는 실제 컴퓨터(하드웨어)가 아닌 소프트웨어로 구현된 컴퓨터라는 뜻으로 컴퓨터 속의 컴퓨터라고 생각하면 된다.

자바로 작성된 애플리케이션은 모두 이 가상 컴퓨터(JVM)에서만 실행되기 때문에, 자바 애플리케이션이 실행되기 위해서는 반드시 JVM이 필요하다.



▲ 그림1-1 Java애플리케이션과 일반 애플리케이션의 비교

일반 애플리케이션의 코드는 OS만 거치고 하드웨어로 전달되는데 비해 Java애플리케이션은 JVM을 한 번 더 거치기 때문에, 그리고 하드웨어에 맞게 완전히 컴파일된 상태가 아니고 실행 시에 해석(interpret)되기 때문에 속도가 느리다는 단점을 가지고 있다. 그러나 요즘엔 바이트코드(컴파일된 자바코드)를 하드웨어의 기계어로 바로 변환해주는 JIT컴파일러와 향상된 최적화 기술이 적용되어서 속도의 격차를 많이 줄였다.

그림1-1에서 볼 수 있듯이 일반 애플리케이션은 OS와 바로 맞붙어 있기 때문에 OS종속적이다. 그래서 다른 OS에서 실행시키기 위해서는 애플리케이션을 그 OS에 맞게 변경해야 한다. 반면에 Java 애플리케이션은 JVM하고만 상호작용을 하기 때문에 OS와 하드웨어에 독립적이라 다른 OS에서도 프로그램의 변경없이 실행이 가능한 것이다. 단, JVM은 OS에 종속적이기 때문에 해당 OS에서 실행가능한 JVM이 필요하다.



▲ 그림1-2 다양한 OS용 JVM

그래서 오라클에서는 일반적으로 많이 사용되는 주요 OS용 JVM을 제공하고 있고, 이렇게 함으로써 자바의 중요한 장점 중의 하나인 “Write once, run anywhere.(한 번 작성하면 어디서든 실행된다.)”이 가능하게 되는 것이다.

그랄 VM – Graal Virtual Machine

그랄 VM은 고성능, 다중언어 실행환경으로 기존의 핫스팟 VM의 JIT 컴파일러를 그랄 컴파일러로 대체한 가상 머신(Virtual Machine)이다. 자바 외에도 다양한 프로그래밍 언어 (JavaScript, python, Ruby, R 등)를 지원하고 심지어는 여러 언어를 혼합해서 코드를 작성하는 것도 가능한데, 그 이유는 그랄VM은 각 언어의 소스 코드를 인터프리터가 그랄 VM이 이해할 수 있는 중립적인 표현으로 변환해서 처리하기 때문이다.

프로그래밍 언어마다 런타임 환경 성능이 제각각이라 어떤 언어는 성능이 상대적으로 많이 떨어지기도 하는데, 그랄VM은 입력된 중간 표현을 자동으로 최적화하고 런타임에 JIT컴파일까지 해주기 때문에 때로는 네이티브 컴파일러보다 실행 성능이 나을 수 있다.

특히 서비스 및 클라우드 환경에 맞게 애플리케이션을 최적화하는 네이티브 바이너리 (native image)로 변환하는 기능을 제공하는 것이 큰 장점이다.

오라클은 2018년 5월에 그랄 VM을 처음으로 공식 발표하였으나, 보다 효율적인 개발을 위해 JDK 16부터 JDK에서 독립시켜서 별도의 JDK로 개발하고 있다. 머지않아 Oracle JDK의 기본 VM으로 포함될 것이다.

| 참고 | 그랄VM을 사용하려면 그랄VM이 포함된 JDK를 설치해야한다. <https://www.graalvm.org/downloads>

2. 자바개발환경 구축하기

2.1 자바 개발도구(JDK)설치하기

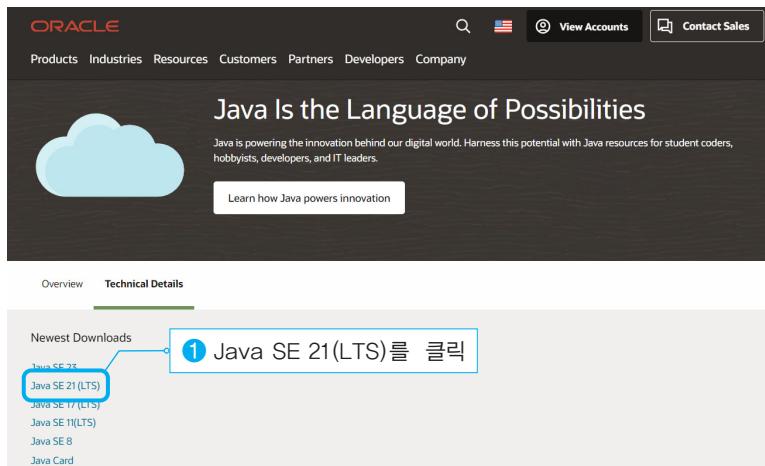
자바로 프로그래밍을 하기위해서는 먼저 JDK(Java Development Kit)를 설치해야 한다. JDK를 설치하면, 자바가상머신(Java Virtual Machine, JVM)과 자바클래스 라이브러리(Java API)외에 자바를 개발하는데 필요한 프로그램들이 설치된다.

JDK 다운로드 받기

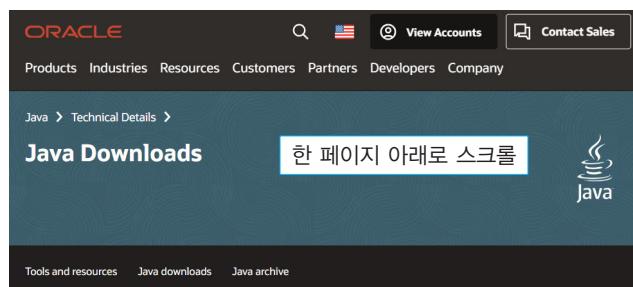
이 책을 학습하기 위해서는 JDK 21 이상의 버전이 필요하며, <http://java.sun.com>에서 다운로드 받아서 설치할 것이다. JDK를 설치하는 것만으로는 자바를 학습하기에 불편하기 때문에, 보다 편리한 개발환경을 제공하는 통합 개발 환경(IDE)가 필요한다. 본인에게 익숙한 것을 사용해도 되지만 가능하면 이 책을 학습하는 동안 인텔리제이(IntelliJ IDEA)을 추천한다. 인텔리제이를 설치하는 방법은 JDK를 설치한 다음에 설명할 것이다.

| 참고 | JDK를 설치하는 방법이 변경된 경우 <https://github.com/castello/javajungsuk4> 에서 JDK21_설치방법.pdf를 확인

1 브라우저를 열고 <http://java.sun.com>을 방문하면 아래와 같은 화면이 나온다. 아래의 화면에서 'Java SE 21(LTS)'를 클릭하자.



2 아래의 페이지가 나타나면 아래로 약간 스크롤하자. 두번째 그림처럼 다운로드 받을 JDK의 종류를 선택하는 화면이 나타난다.



윈도즈 사용자는 아래의 그림과 같이 Windows를 클릭하고, 아래의 세 번째 링크인 'x64 MSI Installer'를 클릭하면, 다운로드가 시작된다.

Java SE Development Kit 21.0.6 downloads

JDK 21 binaries are free to use in production and free to redistribute, at no cost, under the Oracle No-Fee Terms and Conditions (NFTC).

JDK 21 will receive updates under the NFTC, until September 2026, a year after the release of the next LTS. Subsequent JDK 21 updates will be licensed under the Java SE OTN License (OTN) and production use beyond the limited free grants of the OTN license will require a fee.

Platform	Product/file description	File size	Download
Linux	x64 Compressed Archive	185.92 MB	https://download.oracle.com/java/21/latest/jdk-21_windows-x64_bin.zip (sha256)
macOS	x64 Installer	164.31 MB	https://download.oracle.com/java/21/latest/jdk-21_windows-x64_bin.exe (sha256)
Windows	x64 MSI Installer	163.06 MB	https://download.oracle.com/java/21/latest/jdk-21_windows-x64_bin.msi (sha256)

맥OS 사용자는 아래의 그림에서 macOS를 클릭하고, 두 번째 링크인 'ARM64 DMG Installer'를 클릭하면, 다운로드가 시작된다.

| 참고 | Intel CPU가 장착된 컴퓨터의 경우 네 번째 링크인 '64 DMG Installer'를 클릭해야 한다.

Java SE Development Kit 21.0.6 downloads

JDK 21 binaries are free to use in production and free to redistribute, at no cost, under the Oracle No-Fee Terms and Conditions (NFTC).

JDK 21 will receive updates under the NFTC, until September 2026, a year after the release of the next LTS. Subsequent JDK 21 updates will be licensed under the Java SE OTN License (OTN) and production use beyond the limited free grants of the OTN license will require a fee.

Platform	Product/file description	File size	Download
Linux	ARM64 Compressed Archive	182.01 MB	https://download.oracle.com/java/21/latest/jdk-21_macos-aarch64_bin.tar.gz (sha256)
macOS	ARM64 DMG Installer	181.32 MB	https://download.oracle.com/java/21/latest/jdk-21_macos-aarch64_bin.dmg (sha256)
Windows	x64 Compressed Archive	184.25 MB	https://download.oracle.com/java/21/latest/jdk-21_macos-x64_bin.tar.gz (sha256)
Windows	x64 DMG Installer	183.57 MB	https://download.oracle.com/java/21/latest/jdk-21_macos-x64_bin.dmg (sha256)

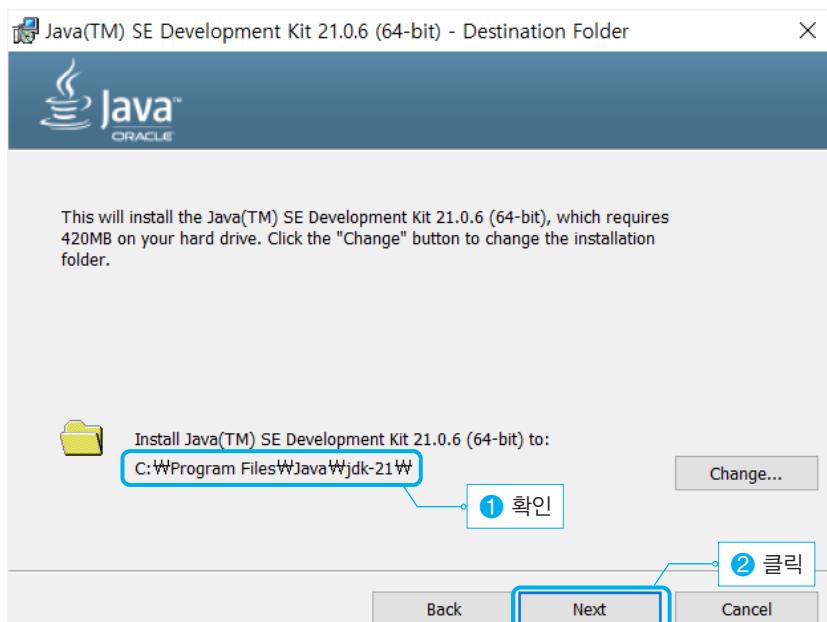
JDK 설치하기 – 윈도우즈

이제 다운로드 받은 JDK를 설치해보자. 윈도우즈에 JDK를 설치하는 방법을 먼저 설명하고 그 다음에 맥OS에 JDK를 설치하는 방법을 설명한다.

- 1 다운로드 받은 'jdk-21_windows-x64_bin.msi'를 실행하면 다음과 같은 화면을 볼 수 있다. 'Next >'버튼을 클릭하자



- 2 JDK를 설치할 위치를 묻는 화면인데, 설치될 위치를 변경하려면, 'Change...'버튼을 누르면 된다. 그냥 설치될 위치 'C:\Program Files\Java\jdk-21'만 확인하고, Next버튼을 클릭하자.



③ 아래와 같은 화면이 나타나면서 설치가 시작되는데, 잠시 후 설치가 모두 끝나고 두 번째 화면이 나타난다. 그러면 설치가 잘 끝난 것이다. 'Close'버튼을 누르자.



4 설치가 끝났으나 한가지 설정이 남았다. 제어판을 열고, 검색창에 '환경변수'라고 입력하자.

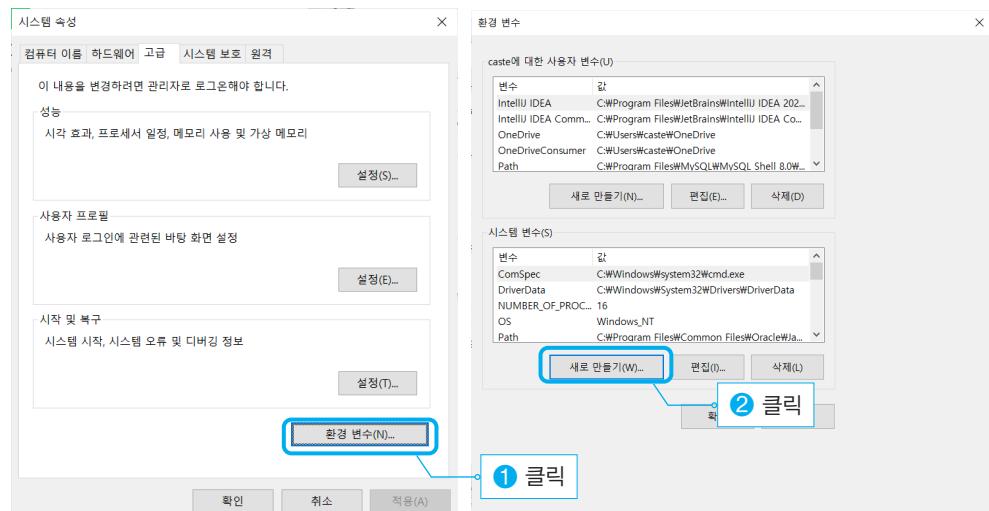
| 참고 | 제어판은 원도우키를 누르고, 찾기에 '제어판'이라고 입력하면 찾을 수 있다.



아래의 화면에서 '시스템 환경 변수 편집'을 클릭하면, '시스템 속성' 화면이 나타난다.



5 아래 왼쪽의 화면에서 '환경 변수(N)...' 버튼을 클릭하면 오른쪽과 같은 '환경 변수'화면이 나타나는데 여기서 '새로 만들기(W)...' 버튼을 누르자.

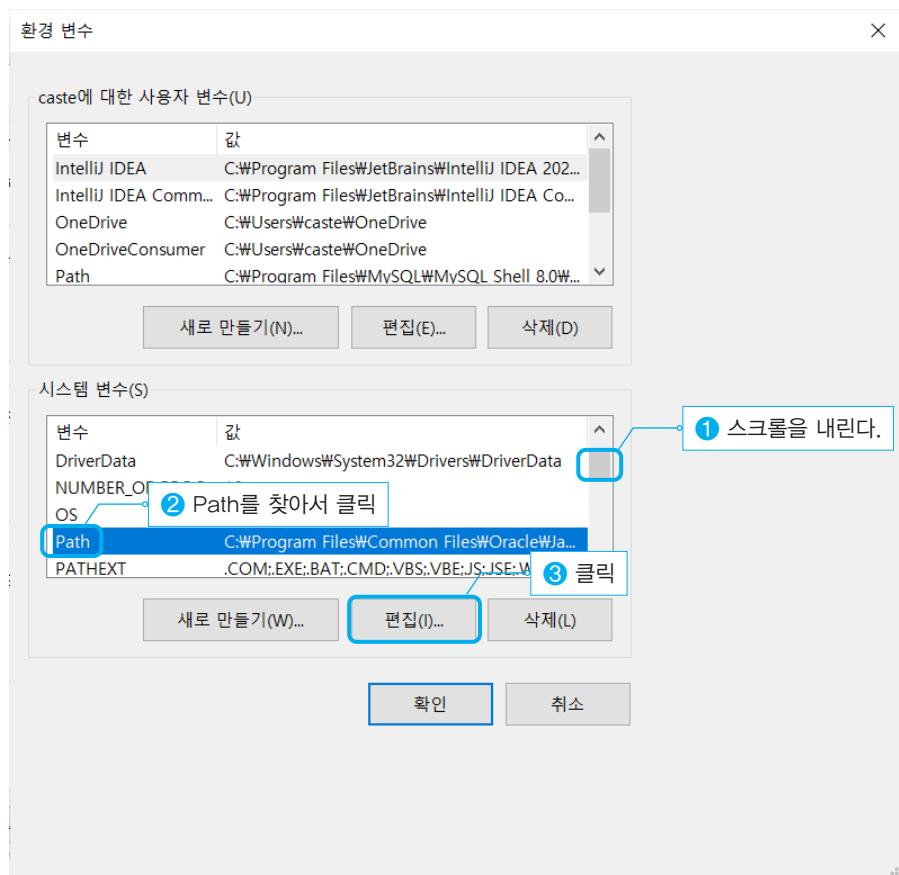


6 아래와 같은 화면이 나타나면, '변수 이름'으로 'JAVA_HOME'을 입력하고 '변수 값'에는 아까 JDK를 설치한 경로인 'C:\Program Files\Java\jdk-21'을 입력하자. 만일 JDK를 다른 곳에 설치했으면, 설치한 경로를 입력해야 한다. 입력한 내용을 다시 한번 확인하고 '확인'을 누르자.

| 참고 | '변수 값'을 직접 입력하기보다 '디렉터리 찾아보기(D)...'버튼을 눌러서 찾는 것이 확실하다.

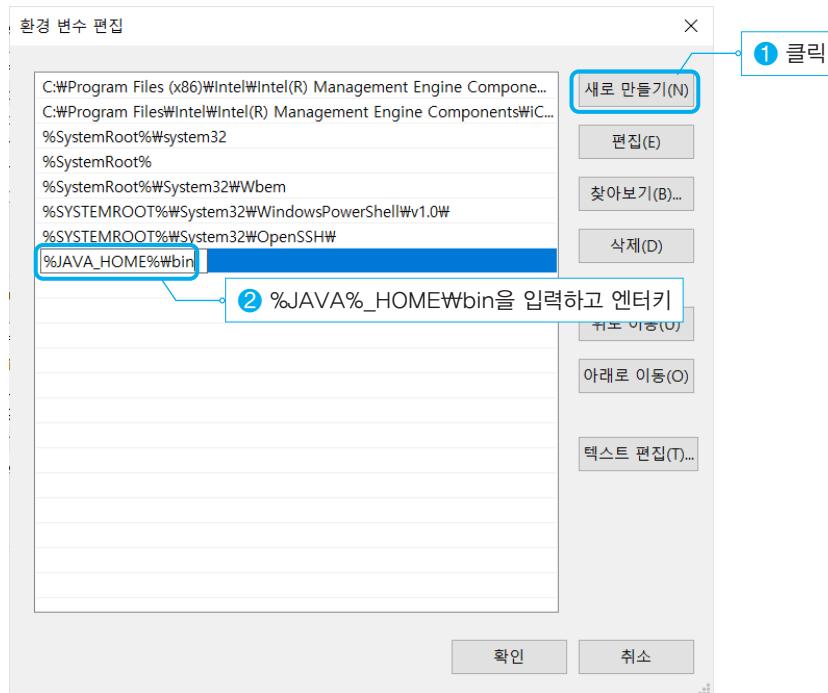


7 아래와 같은 화면이 나타나면, '시스템 변수' 목록의 스크롤바를 아래로 내리면 'Path' 항목을 찾을 수 있다. 이 항목을 클릭하고, '편집(I)...' 버튼을 누르자.

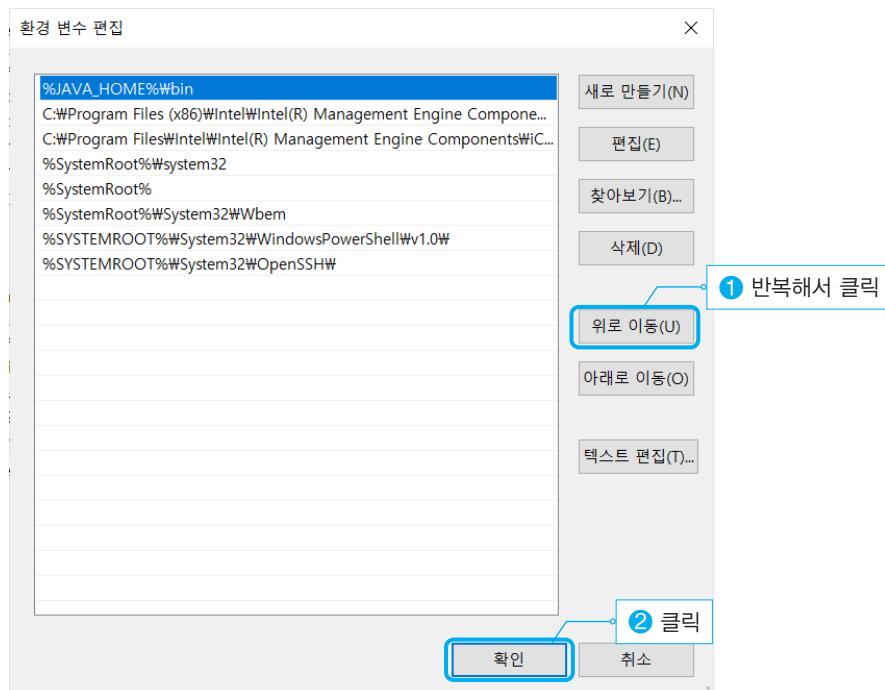


[ebook – 샘플. 무료 공유] 자바의 정석 4판 Java 21 올컬러 2025. 7. 7 출시 seong.namkung@gmail.com

- 8** 아래와 같이 새로운 화면이 열리면 우측 상단의 '새로 만들기(N)'버튼을 누르고, '%JAVA_HOME%\bin'을 입력하고 Enter키를 누르자.

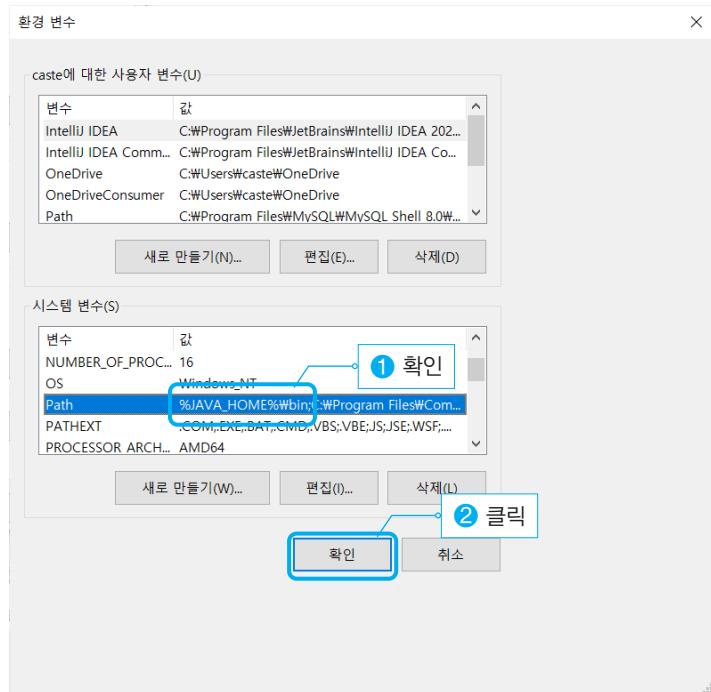


- 9** 우측의 '위로 이동(U)'버튼을 여러번 눌러서 새로 추가한 경로가 맨 위로 올라가게 하고 '확인' 버튼을 눌러서 창을 닫는다.

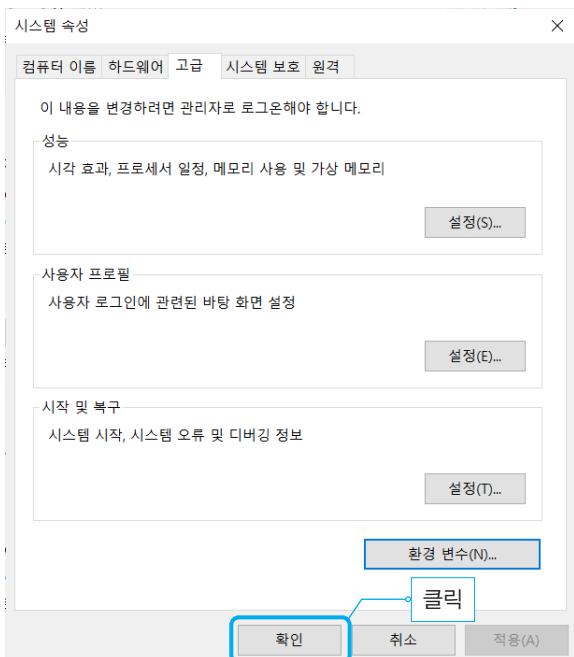


[ebook – 샘플. 무료 공유] 자바의 정석 4판 Java 21 올컬러 2025. 7. 7 출시 seong.namkung@gmail.com

10 아래의 화면에서 전에 입력한 내용이 추가된 것을 확인하고, '확인'버튼을 클릭하자.

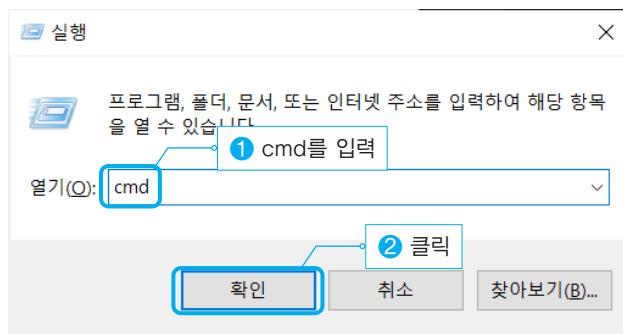


11 아래와 같은 화면에서 다시 '확인'버튼을 클릭하자.



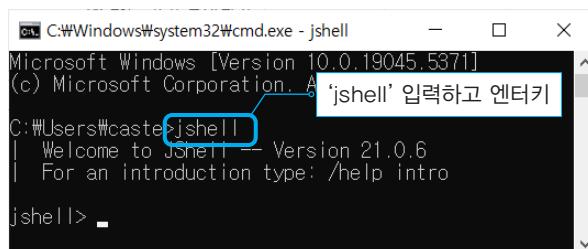
[ebook – 샘플. 무료 공유] 자바의 정석 4판 Java 21 올컬러 2025. 7. 7 출시 seong.namkung@gmail.com

12 이제 설정은 다 끝났고, 설정이 잘되었는지 확인해 보자. '윈도우키+R'을 누르면 아래와 같은 화면이 나오는데, 'cmd'라고 입력하고 '확인'버튼을 누르자.



13 새로운 창이 열리면, 'jshell'이라고 입력하고 엔터키를 누르자. 아래와 같은 결과가 나오면 설정이 잘된 것이니 창을 닫으면 JDK의 설치와 설정이 모두 끝났다.

| 참고 | 만일 'jshell은 내부 또는 ... 아닙니다.'라는 메시지가 나오면 설정이 잘못된 것이며, 4~11의 과정을 다시 반복하자.



JDK 설치하기 – 맥OS

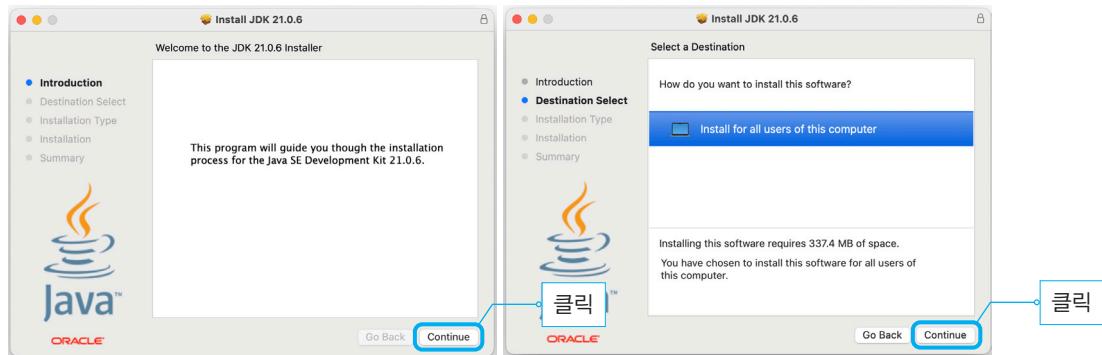
맥OS 사용자를 위한 JDK설치 방법에 대해서 알아보자.

1 앞에서 다운받은 jdk-21_macos-aarch64_bin.dmg파일을 실행하면 다음과 같은 화면이 나온다. 'JDK 21.0.6.pkg'를 더블 클릭하자.

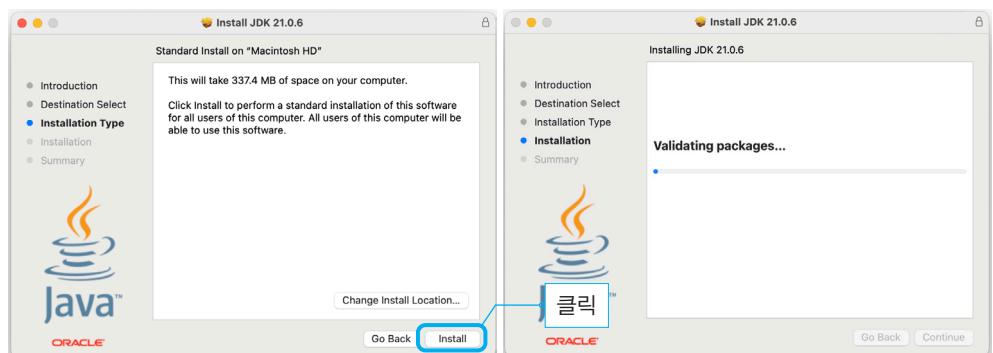


[ebook – 샘플. 무료 공유] 자바의 정석 4판 Java 21 올컬러 2025. 7. 7 출시 seong.namkung@gmail.com

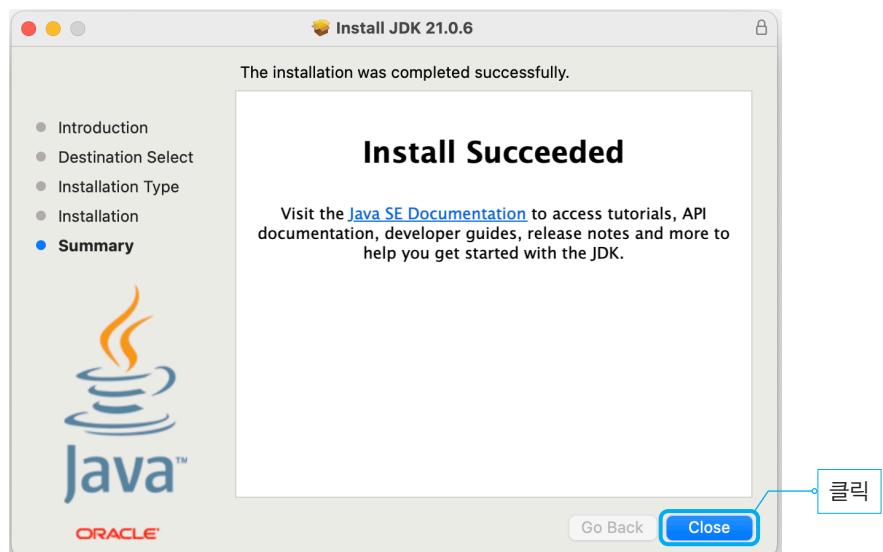
2 아래의 왼쪽 화면에서 'Continue'버튼을 클릭하면, 오른쪽 화면이 나오는데 다시 'Continue' 버튼을 클릭하자.



3 아래의 왼쪽 화면에서 'Install'버튼을 누르면 설치가 시작되면서 오른쪽과 같은 화면이 된다.



4 아래와 같은 화면이 나오면 설치가 잘 끝난 것이다. 'Close'버튼을 누르자.



[ebook – 샘플. 무료 공유] 자바의 정석 4판 Java 21 올컬러 2025. 7. 7 출시 seong.namkung@gmail.com

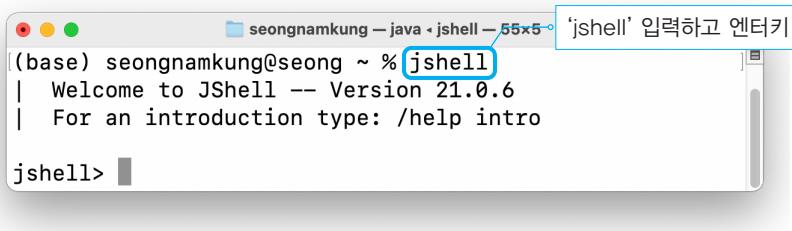
- 5 이제 설치가 끝났으니 설정을 할 차례이다. 사용자 디렉토리 아래의 '.bash_profile' 파일을 열고 아래의 두줄을 마지막에 추가하고 저장하자.

```
export JAVA_HOME=/Library/Java/JavaVirtualMachines/jdk-21.jdk/  
Contents/Home  
export PATH=$JAVA_HOME/bin:${PATH}
```

- 6 마지막으로 터미널을 열고 아래의 명령을 입력하면 변경한 설정이 반영된다.

```
%source ~/.bash_profile
```

- 7 변경한 설정이 잘 반영되었는지 확인하기 위해 아래와 같이 'jshell'을 입력하고 엔터키를 누르면 아래와 같은 결과가 나오는지 확인하자.



2.2 인텔리제이(IntelliJ IDEA) 설치하기

앞으로 통합 개발 도구인 인텔리제이(IntelliJ IDEA)를 설치하고, 간단한 예제를 실행해볼 것이다. 설치 과정이 다소 변경될 수 있으므로 앞으로 설명하는 내용이 실제 화면과 다르면 저자의 깃헙 리포(<https://github.com/castello/javajungsuk4>)에서 '인텔리제이_설치방법.pdf'에서 최신 설치방법을 확인하자.

인텔리제이는 두 가지 버전이 있는데, 무료 버전으로도 학습에 아무런 지장이 없기 때문에 무료 버전을 설치할 것이다.

- IntelliJ IDEA Ultimate – 유료, 30일 무료
- IntelliJ IDEA Community Edition – 무료

먼저 윈도우에 설치하는 방법을 설명하고, 그 다음에 MacOS에 설치하는 방법을 설명할 것이다. 본인의 OS에 맞게 설치하면 된다.

IntelliJ 다운로드하기

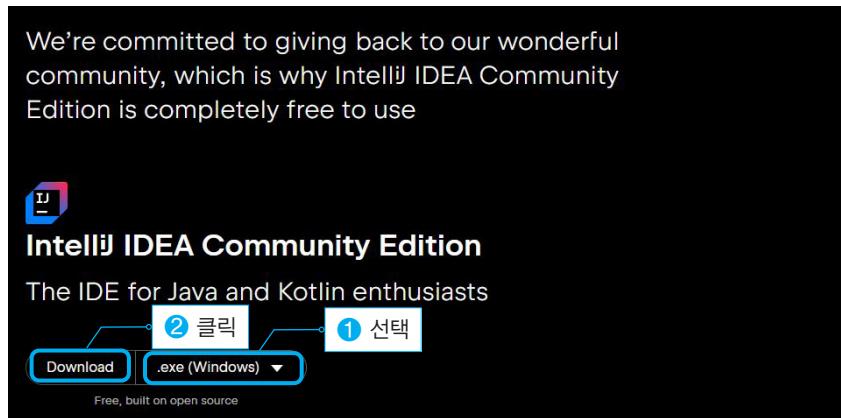
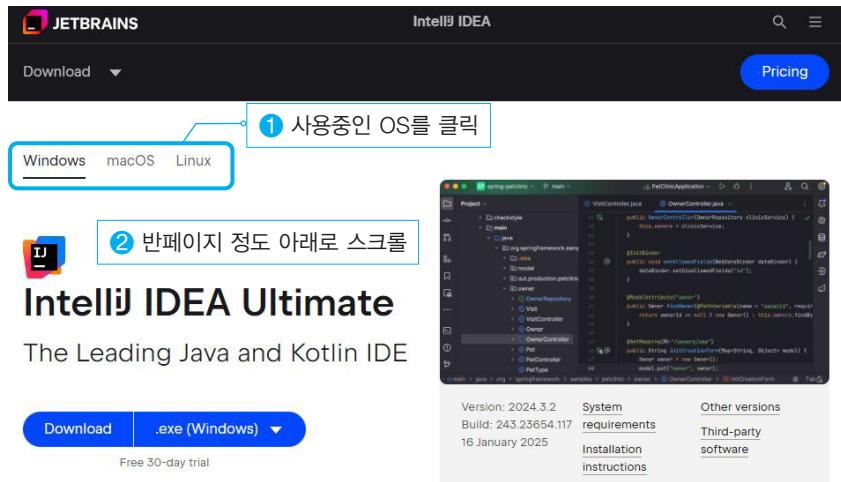
- 1 브라우저를 열고 intelliж download라고 입력하여 검색한 후, 아래의 링크를 클릭

[ebook – 샘플. 무료 공유] 자바의 정석 4판 Java 21 올컬러 2025. 7. 7 출시 seong.namkung@gmail.com



2 아래의 페이지가 나타나면 OS의 종류를 선택해서 클릭하고, 화면을 아래로 반 페이지 정도 스크롤하자. 두번째 그림처럼 IntelliJ IDEA Community Edition이 나오는데, 여기서 윈도우즈의 경우 .exe(Windows), 맥OS의 경우 .dmg(Apple Silicon)을 선택하고, 'Download'버튼을 클릭하자.

| 참고 | 맥OS인데 Intel CPU를 사용하는 경우 .dmg(Intel)을 선택하자.



[ebook – 샘플. 무료 공유] 자바의 정석 4판 Java 21 올컬러 2025. 7. 7 출시 seong.namkung@gmail.com

- ③ 아래의 페이지가 나타나면서 다운로드가 자동으로 시작된다. 만일 자동으로 다운로드가 시작되지 않으면, 아래의 수동 다운로드 링크를 클릭하자.

Thank you for downloading IntelliJ IDEA!

Your download should start shortly. If it doesn't, please use the [direct link](#).

Download and verify the file SHA-256 checksum.

Learn more about the digital signatures of JetBrains binaries.
Third-party software used by IntelliJ IDEA Ultimate Edition



수동 다운로드 링크

이제 다운로드한 파일을 실행해서 인텔리제이를 설치해 보자. 맥OS의 설치가 간단하므로 먼저 살펴보고, 그 다음에 윈도우에 설치하는 방법을 설명할 것이다.

IntelliJ 설치하기 – 맥OS

- ① 전에 다운로드 받은 'idealC-2024.3.2.2-aarch64.dmg'를 더블 클릭하면 다음과 같은 화면이 나타난다. 왼쪽의 'IntelliJ IDEA CE' 아이콘을 드래그해서 'Applications' 아이콘 위로 옮겨놓으면 된다. 설치는 이것으로 끝이다.

| 참고 | 다운받은 파일의 이름은 새로운 버전이 나오면 달라질 수 있다.



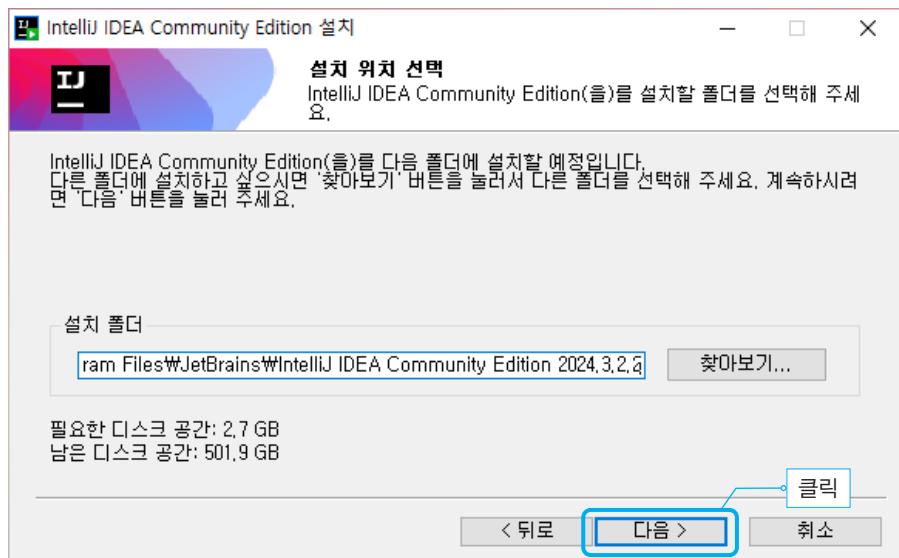
IntelliJ 설치하기 – 윈도우즈

1 전에 다운로드 받은 'idealC-2024.3.2.2.exe'를 더블 클릭하여 실행하면 다음과 같은 화면이 나오는데, '다음 >'버튼을 클릭하자.

| 참고 | 다운받은 파일의 이름은 새로운 버전이 나오면 달라질 수 있다.

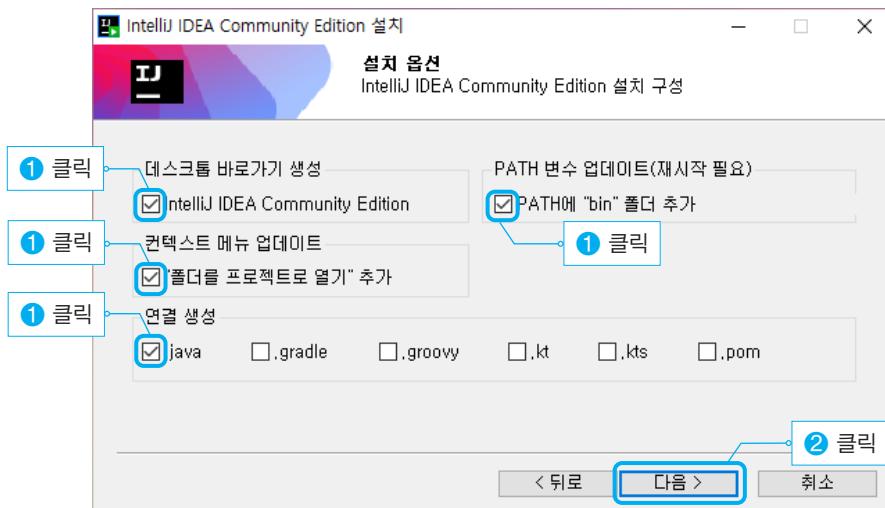


2 인텔리제이를 설치할 위치를 지정하는 화면이 나오는데, 원하는 곳으로 변경해도 되지만 특별한 이유가 없으면 기본을 지정된 위치에 설치하자. 그냥 '다음 >'버튼을 클릭하자.

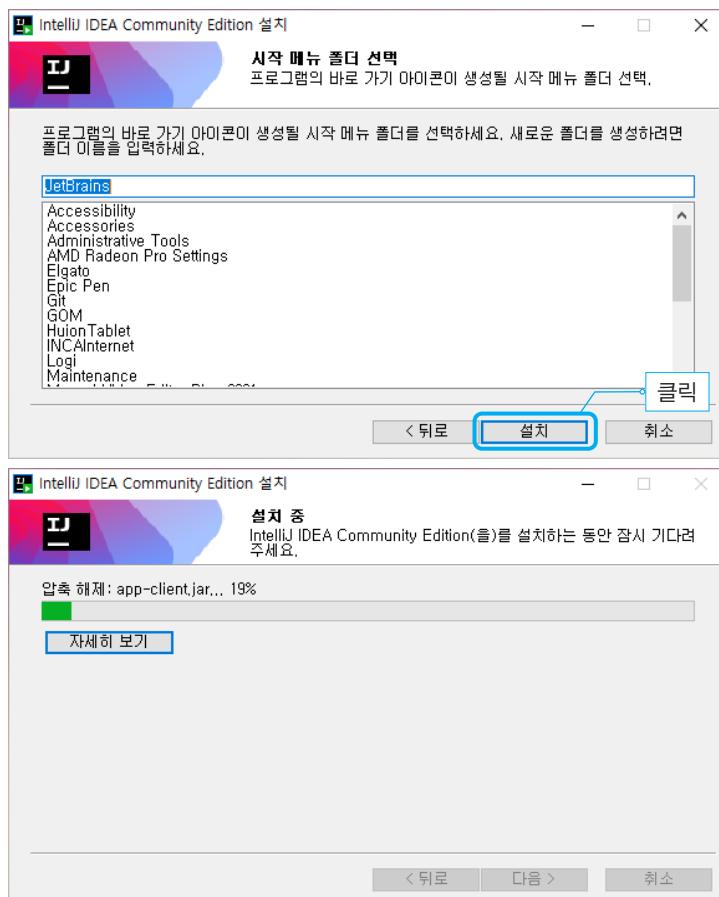


[ebook – 샘플. 무료 공유] 자바의 정석 4판 Java 21 올컬러 2025. 7. 7 출시 seong.namkung@gmail.com

③ 아래에 표시된 옵션들을 클릭해서 체크하고, ‘다음 >’버튼을 누른다.

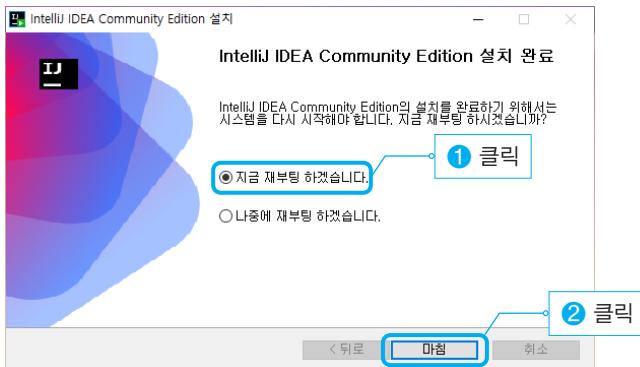


④ 바로가기 아이콘이 생성될 시작 메뉴 폴더를 선택하는 화면이다. ‘설치’버튼을 클릭하면 설치가 시작된다.



[ebook – 샘플. 무료 공유] 자바의 정석 4판 Java 21 올컬러 2025. 7. 7 출시 seong.namkung@gmail.com

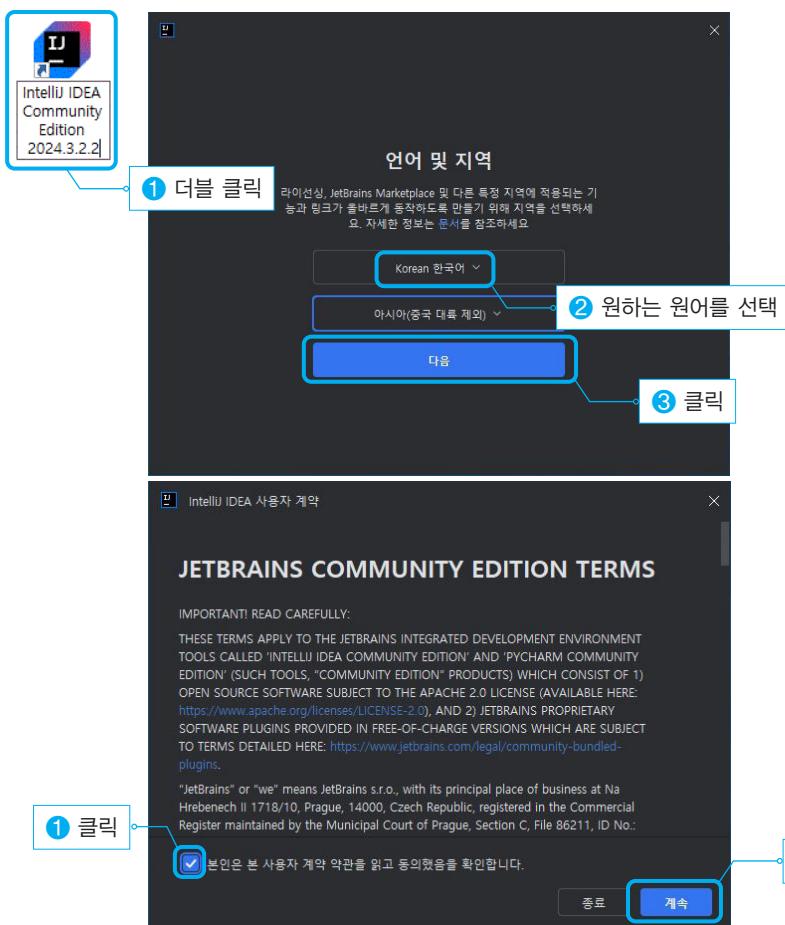
5 설치가 끝나면 아래와 같은 화면이 나타난다. ‘지금 재부팅 하겠습니다.’ 버튼을 클릭하고, ‘마침’ 버튼을 클릭하면 설치가 끝난다.



IntelliJ 실행하기 – 윈도우즈, 맥OS

1 바탕화면에 새로 생성된 아이콘을 클릭하면, 아래와 같은 화면이 나온다. 언어를 선택하고 ‘다음’ 버튼을 클릭하고, 그 다음의 사용자 계약 화면에서 체크하고 ‘계속’ 버튼을 클릭하자.

| 참고 | 경우에 따라 아래의 화면이 생략될 수도 있으며.. 언어와 지역은 나중에 변경할 수 있다.



[ebook – 샘플. 무료 공유] 자바의 정석 4판 Java 21 올컬러 2025. 7. 7 출시 seong.namkung@gmail.com

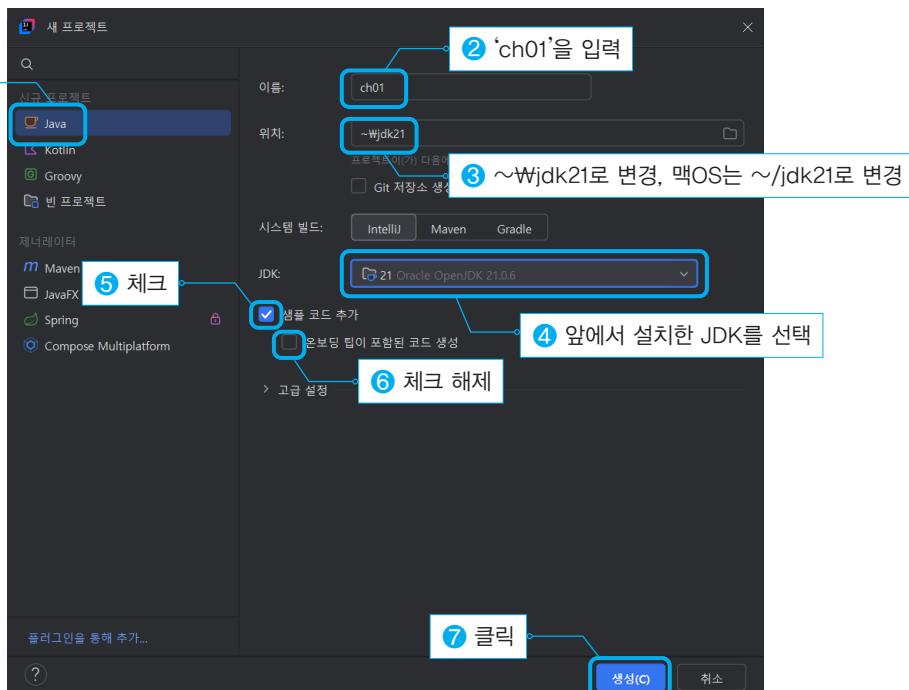
2 아래의 화면에서 좌측의 '프로젝트'를 클릭하고, 중앙의 '새 프로젝트'를 클릭하자.

| 참고 | 아래의 화면이 나타나지 않으면, 메뉴에서 File > New > Project...를 클릭하면 다음 단계의 화면이 나타난다.

| 참고 | 좌측의 '프로젝트' 아래의 '사용자 지정'을 클릭하면, 사용 언어와 테마 등의 설정을 변경할 수는 화면이 나온다.

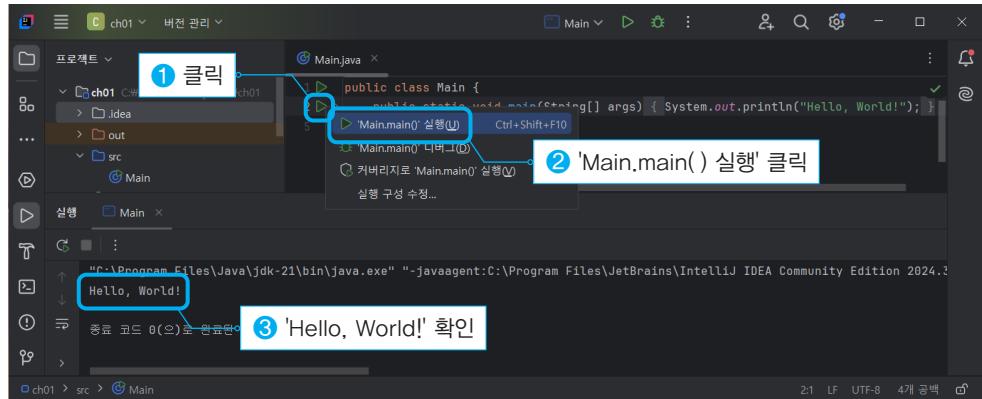


3 새로 생성할 프로젝트의 정보를 입력하는 화면이 나오는데, 프로젝트의 이름을 입력하고 경로를 변경하자. JDK는 전에 설치한 것이 자동으로 나타난다. 앞으로 챕터마다 이름만 다르게 새 프로젝트를 생성하자.

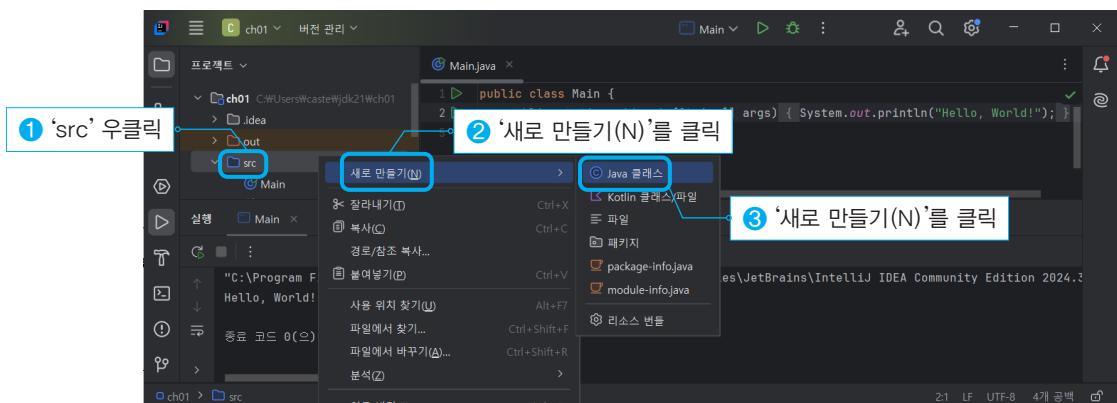


[ebook – 샘플. 무료 공유] 자바의 정석 4판 Java 21 올컬러 2025. 7. 7 출시 seong.namkung@gmail.com

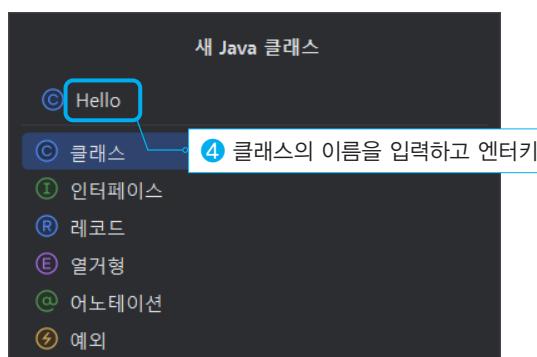
- 4 새로운 프로젝트가 생성되고 자동으로 Main.java라는 파일이 아래와 같이 생성된다. 녹색 삼각형을 클릭하면 프로그램이 실행되고 그 결과가 화면 하단의 콘솔에 'Hello, World!'가 출력된다. 이제 이걸로 인텔리제이의 설치와 설정이 모두 끝났다.



- 5 자바는 클래스 단위로 코드를 작성하며, 앞으로 새로운 예제를 작성할 때는 아래와 같이 새로운 클래스를 생성하고 코드를 작성하면 된다.



위의 화면에서 '새로 만들기(N)'를 클릭하면 아래와 같은 화면이 나오는데, 클래스 이름을 적고 엔터키를 누르면 새로운 파일이 생성된다. 생성된 파일에 예제의 내용대로 작성하고 이전 단계와 같이 녹색 삼각형을 눌르서 작성한 예제를 실행하면 된다.



3. 자바로 프로그램작성하기

3.1 Hello.java

자바로 프로그램을 개발하려면 JDK이외에도 편집기가 필요하다. 메모장과 같은 간단한 편집기도 있지만, 처음 자바를 배우는 사람들은 인텔리제이(IntelliJ IDEA)나 이클립스(eclipse)와 같이 다양하고 편리한 기능을 겸비한 고급 개발도구를 사용하는 것이 좋다.

이클립스에 비해 기능은 떨어지지만, 가볍고 간단한 편집기로 비쥬얼 스튜디오 코드(Visual Studio Code)라는 것도 있다.

| 참고 | 비쥬얼 스튜디오 코드는 <https://code.visualstudio.com/>에서 무료로 다운로드받을 수 있다.

▼ 예제 1-1/Hello.java

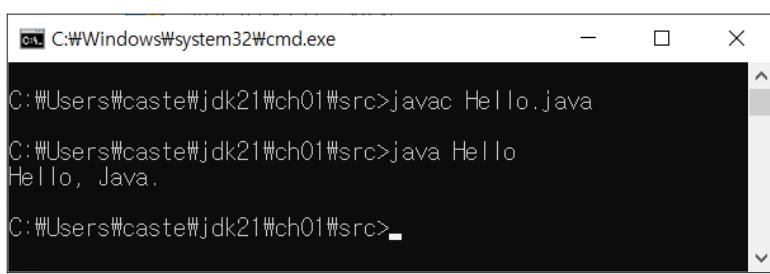
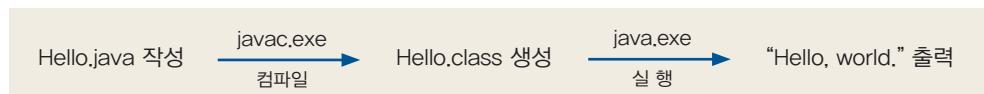
```
class Hello {
    public static void main(String[] args) {
        System.out.println("Hello, Java."); // 화면에 글자를 출력한다.
    }
}
```

▼ 실행결과
Hello, Java.

이 예제는 화면에 ‘Hello, Java.’를 출력하는 아주 간단한 프로그램이다. 이 예제를 통해서 화면에 글자를 출력하려면 어떻게 해야 하는지 쉽게 알 수 있을 것이다.

예제1-1을 편집기를 이용해서 작성한 다음 ‘Hello.java’로 저장하자. 이 때 클래스의 이름 ‘Hello’가 대소문자까지 정확히 같아야 한다.

이 예제를 실행하려면, 먼저 자바 컴파일러(javac.exe)를 사용해서 소스파일(Hello.java)로부터 클래스파일(Hello.class)을 생성해야 한다. 그 다음에 자바 인터프리터(java.exe)로 실행한다.



▲ 그림1-3 Hello.java를 컴파일하고 실행한 화면

그림1-3과 같은 결과를 얻었다면 자바로 프로그래밍할 준비가 모두 끝난 것이다. 만일 컴파일 시에 오류가 발생했다면 ‘3.2 자주 발생하는 에러와 해결방법’을 참고하자.

자바에서 모든 코드는 반드시 클래스 안에 존재해야 하며, 서로 관련된 코드들을 그룹으로 나누어 별도의 클래스를 구성하게 된다. 그리고 이 클래스들이 모여 하나의 Java 애플리케이션을 이룬다.

클래스를 작성하는 방법은 간단하다. 키워드 ‘class’ 다음에 클래스의 이름을 적고, 클래스의 시작과 끝을 의미하는 괄호{} 안에 원하는 코드를 넣으면 된다.

```
class 클래스이름 {
    /*
        모든 코드는 클래스의 블럭{} 내에 작성해야한다. (주석 제외)
    */
}
```

| 참고 | 나중에 배우게 될 package문과 import문은 예외적으로 클래스의 밖에 작성한다.

아래 코드의 ‘public static void main(String[] args)’는 main메서드의 선언부인데, 프로그램을 실행할 때 ‘java.exe’에 의해 호출될 수 있도록 미리 약속된 부분이므로 항상 똑같이 적어주어야 한다.

| 참고 | []은 배열을 의미하는 기호로 배열의 타입(type) 또는 배열의 이름 옆에 붙일 수 있다. ‘String[] args’는 String타입의 배열 args를 선언한 것이며, ‘String args[]’와 같이 쓸 수도 있다. 이 들은 같은 의미이므로 차이가 없다. 자세한 내용은 ‘5장 배열’에서 배우게 될 것이다.

```
class 클래스이름 {
    public static void main(String[] args) // main메서드의 선언부
    {
        // 실행될 문장들을 적는다.
    }
}
```

main메서드의 선언부 다음에 나오는 괄호{}는 메서드의 시작과 끝을 의미하며, 이 괄호 사이에 작업할 내용을 작성해 넣으면 된다. Java 애플리케이션은 main메서드의 호출로 시작해서 main메서드의 첫 문장부터 마지막 문장까지 수행을 마치면 종료된다.

모든 클래스가 main메서드를 가지고 있어야 하는 것은 아니지만, 하나의 Java 애플리케이션에는 main메서드를 포함한 클래스가 반드시 하나는 있어야 한다. main메서드는 Java애플리케이션의 시작점이므로 main메서드 없이는 Java 애플리케이션은 실행될 수 없기 때문이다. 작성된 Java애플리케이션을 실행할 때는 ‘java.exe’ 다음에 main메서드를 포함한 클래스의 이름을 적어줘야 한다.

하나의 소스파일에 하나의 클래스만을 정의하는 것이 보통이지만, 하나의 소스파일에 둘 이상의 클래스를 정의하는 것도 가능하다. 이 때 주의해야 할 점은 ‘소스파일의 이름은 public class의 이름과 일치해야 한다.’는 것이다. 만일 소스파일 내에 public class가 없다면, 소스파일의 이름은 소스파일 내의 어떤 클래스의 이름으로 해도 상관없다.

올바른 작성 예	설명
<pre>Hello2.java public class Hello2 {} class Hello3 {}</pre>	public class가 있는 경우, 소스파일의 이름은 반드시 public class의 이름과 일치해야한다.
<pre>Hello2.java class Hello2 {} class Hello3 {}</pre>	public class가 하나도 없는 경우, 소스파일의 이름은 'Hello2.java', 'Hello3.java' 둘 다 가능하다.

잘못된 작성 예	설명
<pre>Hello2.java public class Hello2 {} public class Hello3 {}</pre>	하나의 소스파일에 둘 이상의 public class가 존재하면 안 된다. 각 클래스를 별도의 소스파일에 나눠서 저장하던가 아니면 둘 중의 한 클래스에 public을 붙이지 않아야 한다.
<pre>Hello3.java public class Hello2 {} class Hello3 {}</pre>	소스파일의 이름이 public class의 이름과 일치하지 않는 다. 소스파일의 이름을 'Hello2.java'로 변경해야 맞다.
<pre>hello2.java public class Hello2 {} class Hello3 {}</pre>	소스파일의 이름과 public class의 이름이 일치하지 않는 다. 대소문자를 구분하므로 대소문자까지 일치해야한다. 그래서, 소스파일의 이름에서 'h'를 'H'로 바꿔야 한다.

▲ 표1-2 소스파일의 작성 예

소스파일(*.java)과 달리 클래스파일(*.class)은 클래스마다 하나씩 만들어지므로 표1-2의 ‘올바른 작성 예’에 제시된 'Hello2.java'를 컴파일하면 'Hello2.class'와 'Hello3.class' 모두 두 개의 클래스파일이 생성된다.

접근 제어자(access modifier)인 ‘public’에 대해서는 ‘7장 객체지향 프로그래밍 II’에서 자세히 배울 것이므로 여기서는 하나의 소스파일에 둘 이상의 클래스를 정의할 때 주의할 점에 대해서만 이해하고 넘어가자.

3.2 자주 발생하는 에러와 해결방법

자바로 프로그래밍을 배워나가면서 많은 수의 크고 작은 에러들을 접하게 될 것이다. 대부분의 에러는 작은 실수에서 비롯된 것들이며, 곧 익숙해져서 쉽게 대응할 수 있게 되지만 처음 배울 때는 작은 실수 하나 때문에 많은 시간을 허비하곤 한다.

그래서 자주 발생하는 기본적인 에러와 해결방법을 간단히 정리하였다. 에러가 발생하였을 때 참고하고, 그 외의 에러는 에러메시지의 일부를 인터넷에서 검색해서 찾아보면 해결책을 얻는데 도움이 될 것이다.

1. cannot find symbol 또는 cannot resolve symbol

지정된 변수나 메서드를 찾을 수 없다는 뜻으로 선언되지 않은 변수나 메서드를 사용하거나, 변수 또는 메서드의 이름을 잘못 사용한 경우에 발생한다. 자바에서는 대소문자 구분을 하기 때문에 철자 뿐 만아니라 대소문자의 일치여부도 꼼꼼하게 확인해야한다.

2. ';' expected

세미콜론 ';'이 필요한 곳에 없다는 뜻이다. 자바의 모든 문장의 끝에는 ';'을 붙여주어야 하는데 가끔 이를 잊고 실수하기 쉽다.

3. Exception in thread "main" java.lang.NoSuchMethodError: main

'main메서드를 찾을 수 없다.'는 뜻인데 실제로 클래스 내에 main메서드가 존재하지 않거나 메서드의 선언부 'public static void main(String[] args)'에 오타가 존재하는 경우에 발생한다.

이 에러의 해결방법은 main메서드가 클래스에 정의되어 있는지 확인하고, 정의되어 있다면 main메서드의 선언부에 오타가 없는지 확인한다. 자바는 대소문자를 구별하므로 대소문자의 일치여부까지 정확히 확인해야한다.

| 참고 | args는 매개변수의 이름이므로 args 대신 argv나 arg와 같이 다른 이름을 사용할 수 있다.

4. Exception in thread "main" java.lang.NoClassDefFoundError: Hello

'Hello라는 클래스를 찾을 수 없다.'는 뜻이다. 클래스 'Hello'의 철자, 특히 대소문자를 확인해보고 이상이 없으면 클래스파일 (*.class)이 생성되었는지 확인한다.

예를 들어 'Hello.java'가 정상적으로 컴파일 되었다면 클래스파일 'Hello.class'가 있어야한다. 클래스파일이 존재하는데도 동일한 메시지가 반복해서 나타난다면 클래스패스(classpath)의 설정이 바르게 되었는지 다시 확인해보자.

5. illegal start of expression

직역하면 문장(또는 수식, expression)의 앞부분이 문법에 맞지 않는다는 의미인데, 간단히 말해서 문장에 문법적 오류가 있다는 뜻이다. 팔호 '(' 나 '{'를 열고서 닫지 않거나, 수식이나 if문, for문 등에 문법적 오류가 있을 때 또는 public이나 static과 같은 키워드를 잘못 사용한 경우에도 발생한다. 에러가 발생한 곳이 문법적으로 옳은지 확인하라.

6. class, interface, or enum expected

이 메시지의 의미는 ‘키워드 class나 interface 또는 enum이 없다.’이지만, 보통 팔호 ‘{’ 또는 ‘}’의 개수가 일치하지 않는 경우에 발생한다. 열린팔호‘{’와 닫힌팔호‘}’의 개수가 같은지 확인하자.

마지막으로 한 가지 더 얘기하고 싶은 것은 에러가 발생했을 때, 어떻게 해결할 것인가에 대한 방법이다. 아주 간단하고 당연한 내용이라서 다소 실망스럽게 느껴질지도 모르지만, 막상 실제 에러가 발생했을 때 아래의 순서대로 처리해보면 도움이 될 것이다.

1. 에러 메시지를 잘 읽고 해당 부분의 코드를 살펴본다.
이상이 없으면 해당 코드의 주위(윗줄과 아래 줄)도 함께 살펴본다.
2. 그래도 이상이 없으면 에러 메시지는 잊어버리고 기본적인 부분을 재확인한다.
대부분의 에러는 사소한 것인 경우가 많다.
3. 의심이 가는 부분을 주석처리하거나 따로 떼어내서 테스트 한다.

에러 메시지가 실제 에러와는 관계없는 내용일 때도 있지만, 대부분의 경우 에러 메시지만 잘 이해해도 문제가 해결되는 경우가 많으므로 에러 해결을 위해서 제일 먼저 해야 할 일은 에러 메시지를 잘 읽는 것임을 명심하자.

3.3 자바프로그램의 실행과정

콘솔에서 아래와 같이 Java 애플리케이션을 실행시켰을 때

```
C:\jdk21\ch01\src>java Hello  
                                        ↑  
                                        main(String[] args)
```

내부적인 진행순서는 다음과 같다.

1. 프로그램의 실행에 필요한 클래스(*.class파일)를 로드한다.
2. 클래스파일을 검사한다.(파일형식, 악성코드 체크)
3. 지정된 클래스(Hello)에서 main(String[] args)를 호출한다.

main메서드의 첫 줄부터 코드가 실행되기 시작하여 마지막 코드까지 모두 실행되면 프로그램이 종료되고, 프로그램에서 사용했던 자원들은 모두 반환된다.

만일 지정된 클래스에 main메서드가 없다면 다음과 같은 에러 메시지가 나타날 것이다.

```
Exception in thread "main" java.lang.NoSuchMethodError: main
```

3.4 주석(comment)

작성하는 프로그램의 크기가 커질수록 프로그램을 이해하고 변경하는 일이 점점 어려워진다. 심지어는 자신이 작성한 프로그램도 ‘내가 왜 이렇게 작성했지?’라는 의문이 들기도 하는데, 남이 작성한 코드를 이해한다는 것은 정말 쉬운 일이 아니다.

이러한 어려움을 덜기 위해 사용하는 것이 바로 주석이다. 주석을 이용해서 프로그램 코드에 대한 설명을 적절히 덧붙여 놓으면 프로그램을 이해하는 데 많은 도움이 된다.

그 외에도 주석은 프로그램의 작성자, 작성일시, 버전과 그에 따른 변경이력 등의 정보를 제공할 목적으로 사용된다.

주석을 작성하는 방법은 다음과 같이 두 가지 방법이 있다. ‘/*’와 ‘*/’사이에 주석을 넣는 방법과 앞에 ‘//’를 붙이는 방법이 있다.

범위 주석 ‘/*’와 ‘*/’사이의 내용은 주석으로 간주된다.

한 줄 주석 ‘//’부터 라인 끝까지의 내용은 주석으로 간주된다.

| 참고 | 이 외에도 Java API문서와 같은 형식의 문서를 자동으로 만들 수 있는 주석(** ~ *)이 있지만 많이 사용되지는 않으므로 자세한 설명은 생략한다. 이 주석은 javadoc.exe에 의해서 html문서로 자동 변환되며, 보다 자세한 내용은 ‘javadoc’으로 검색하면 찾을 수 있다.

다음은 주석의 몇 가지 사용 예인데 흰색바탕으로 처리된 부분이 주석이다.

```
/*
Date : 2025. 3. 1
Source : Hello.java
Author : 남궁성
Email : seong.namkung@gmail.com
*/

class Hello
{
    public static void main(String[] args) /* 프로그램의 시작 */
    {
        System.out.println("Hello, Java."); // Hello, Java를 출력
    }
}
```

위의 코드는 예제1-1에 주석을 넣은 것인데, 컴파일러는 주석을 무시하고 건너뛰기 때문에 위의 코드를 컴파일한 결과와 예제1-1을 컴파일한 결과는 정확히 일치한다. 따라서 주석이 많다고 해서 프로그램의 성능이 떨어지는 일은 없으니 안심하고 주석을 활용하기 바란다. 코드를 작성하기 전에 미리 주석으로 자신의 생각을 정리하고 검토하는 것은 좋은 습관이다. 주석을 사용하지 말라는 주장도 있으나, 이 주장은 코드를 대충 작성하고 주석을 달지말고 주석이 필요없을 정도로 코드를 읽기 좋게 잘 작성하라는 뜻이다.

한 가지 주의해야할 점은 문자열을 의미하는 큰따옴표(“) 안에 주석이 있을 때는 주석이 아닌 문자열로 인식된다는 것이다. Hello.java를 아래와 같이 변경하여 실행해보면, 주석의 내용도 같이 출력되는 것을 확인할 수 있을 것이다.

```
class Hello
{
    public static void main(String[] args)
    {
        System.out.println("Hello, /* 이것은 주석 아님 */ world.");
        System.out.println("Hello, world. // 이것도 주석 아님");
    }
}
```

3.5 이 책으로 공부하는 방법

2000년에 처음으로 자바강의를 시작했으니까 벌써 25년이 넘는 세월이 흘렀다. 그동안 많은 학생들을 가르치면서 어떻게 하면 더 쉽게 잘 배울 수 있을까에 대한 고민을 끊임없이 해왔고 그에 대한 결실로 책도 쓰게 되었다. 여전히 많은 학생들이 자바를 공부하는데 어려움을 겪고 있고, 공부 방법에 대해 고민하는 글들이 저자가 운영하는 카페에 많이 올라왔다. 카페 회원들과 소통하면서 처음 자바를 배우는 학생들이 어떤 점을 어려워하는지 잘 알게 되었고 그 고민에 대한 나름대로의 해법을 갖게 되었다. 카페에 자바를 공부하는 방법에 대한 글도 여러 번 쓰기도 했는데, 책을 구입하고도 카페에 가입하지 않는 독자들도 있기 때문에 그동안의 공부 방법에 대한 고민을 정리해서 책에 포함시키기로 했다.

누구나 자신만의 공부 방법이 있고, 절대적인 것은 없기 때문에 여기서 제시하는 공부 방법을 참고해서 자신에게 맞는 공부 방법을 완성하기 바란다.

이 책은 크게 3부분, 1장부터 5장까지, 6장부터 9장, 그리고 나머지 부분으로 나눌 수 있다. 처음 프로그래밍 언어를 배우는 사람은 2장부터 5장을 익숙해질 때까지 반복해서 봐야하고 실습도 많이 해야 한다. 눈으로만 이해하지 말고, 반드시 모든 예제를 직접 입력해보고 실행결과를 확인해 보자. 응용이 잘 안된다고 해서 앞부분에만 머물면 안 된다. 지금 단계에서 응용이 안 되는 것은 당연하다. 기본적인 내용이 익숙해지면, 그 다음 단계인 6장으로 넘어가야 한다.

6장과 7장이 객체지향 개념의 핵심인데, 먼저 6장과 7장에 어떤 내용이 있는지 가볍게 한번 훑고 시작하자. 그 다음엔 6장을 여러 번 반복해서 보자. 6장을 이해하지 못하면 7장은 이해할 수 없기 때문이다. 7장은 좀 어려우므로 저자의 유튜브 채널(<https://www.youtube.com/@MasterNKS>)의 무료 동영상 강좌를 꼭 볼 것을 권한다.

반복해서 볼 때는 동영상을 1.5배속이나 2배속으로 보면 한 시간에 한번은 볼 수 있을 것이다. 하루에 2시간씩 5일보면, 10번은 볼 수 있다. 10번 봐도 이해가 안가면 10번 더 보자. 객체지향 개념을 이해하는데 30시간도 안 걸린다면 대성공이다.

객체지향 개념을 공부할 때 주의할 점은, 객체지향 개념 자체에 몰두하지 않아야 한다는 것이다. 이것은 완전히 샛길로 빠지는 것으로, 여러분들은 객체지향개념 언어인 ‘자바’를 배우는 것이지 객체지향 개념을 배우는 것이 아니라는 것을 잊지 말자.

6장과 7장은 반복하면 반복할수록 이해가 깊어지므로, 앞으로도 꾸준히 가볍게 복습하는 것이 좋다. 강의 내용을 요약해서 암기하자. 9장까지가 자바의 가장 기본적인 내용이므로 9장까지 마치고 나면, 2장부터 9장까지 전체적으로 한번 복습하면 좋다.

마지막으로 10장부터 16장까지는 자바의 응용부분이므로 앞부분을 충분히 이해하지 않고는 학습하기 어렵다. 이중에서 11장, 12장과 15장을 제외하고 나머지는 필요할 때 공부해도 좋다.

10장은 어떤 클래스들이 있는지 확인하고 필요할 때 책을 보고 사용할 수 있을 정도로만 공부하면 된다.

11장은 지금까지 배운 것들을 전부 활용하고 자료구조의 원리까지 들어가므로 책 전체에서 가장 어렵다. 처음엔 각 클래스의 특징과 사용법 정도만 확인하고 넘어가야 한다. 어떤 클래스들이 있고, 이 클래스들 통해서 어떤 결과를 얻을 수 있다는 정도면 충분하다. 처음부터 이장의 모든 내용을 이해하려고하면 어렵게만 느껴지고 진도도 안 나갈 것이다.

12장에서는 지네릭스가 중요한데, 예전에는 선택적으로 사용하던 기능이었지만 이제는 지네릭스를 모르고는 이해할 수 없는 코드가 많다. 지네릭스는 깊이 들어가면 어렵기 때문에 처음엔 기본적인 사용법만 익히고, 다른 장들을 공부하면서 막히는 부분을 다시 복습하는 식으로 공부하면 좋다. 애너테이션과 열거형은 경력자들의 요청으로 자세히 썼는데, 프로그래밍을 처음 배우는 사람은 기본적인 것만 이해하고 넘어가도 된다.

13장은 쓰레드에 대한 것인데, 일단 쓰레드가 어떤 것인지에 대한 감을 잡는 정도로만 공부하고, 나중에 필요할 때 자세히 보는 것이 좋다. 자바에서는 쉽게 멀티쓰레드를 구현할 수 있도록 미리 작성된 클래스들을 제공하고 있기 때문에 기본 개념만 알아도 도움이 많이 된다.

14장의 람다와 스트림은 11장, 12장과 관련이 많고 난이도가 높기 때문에 프로그래밍을 처음 배우는 사람들은 건너뛰었다가 필요할 때 추가로 학습해도 좋다.

15장은 입출력에 대한 것인데, 이 책의 후반부에서 꼭 학습해야하는 중요한 부분이다. 다른 장에 비해 실습이 재미있을 것이다.

16장은 컴퓨터간의 통신하는 방법에 대한 내용인데, 채팅 프로그램을 만드는 방법을 배운다. 15장과 관련이 있으므로 15장을 충분히 이해한 다음에 학습해야하며, 16장은 필수적으로 공부하지 않아도 되므로 건너뛰어도 좋다.

그 다음에는 안드로이드나 웹프로그래밍(JSP, Spring)을 공부하면서 하루에 한 챕터씩 꾸준히 복습해야 한다. 누구나 시간이 지나면 잊어버리기 때문에 계속 조금씩이라도 반복해서 봐야 실력이 쌓인다. 새로 배워야 할 것이 많다고 기본을 소홀히 하면 배우는 것보다 잃는 것이 더 많을 것이다. 하루에 10분이라도 반드시 시간을 내어 복습하자.

[ebook – 샘플. 무료 공유] 자바의 정석 4판 Java 21 올컬러 2025. 7. 7 출시 seong.namkung@gmail.com

연습문제에 대하여

그동안 책이 두꺼워서 힘들다는 얘기를 많이 들었다. 그래서 고민 끝에 연습문제(약 200페이지)를 별도의 pdf파일로 제공하기로 결정했다. 연습문제는 저자의 깃헙 리포(<https://github.com/castello/javajungsuk4>)에서 제공하며, 상업적인 목적이 아니면 누구나 자유롭게 배포해도 된다.

하나의 챕터(chapter, 장)를 공부하고나면 연습문제를 꼭 풀어보자. 그러나 모든 문제를 다 풀어야 하는 것은 아니다. 풀 수 있는 문제들만 풀고, 풀 수 없는 문제들은 넘어갔다가 나중에 복습할 때 다시 풀어보는 것이 좋다. 아니면 하루에 한 문제씩 틈틈이 고민하는 것도 좋다.

연습문제는 책에 있는 예제들을 응용한 것이 많기 때문에 연습문제를 풀면서 안 풀리는 문제는 해당 챕터를 뒤적거리면서 비슷한 예제가 없는지 찾아봐야 한다. 그러면서 공부가 되는 것이지, 문제만 붙들고 아무리 고민해봐야 문제도 안 풀리고 공부도 되지 않는다.

문제를 풀 때는 항상 종이에 낙서를 하는 방법을 추천한다. 어떻게 풀 것인가를 머리로만 고민하는 것보다 글과 그림으로 시각화하면 문제가 명확해지고 문제해결의 실마리를 쉽게 찾을 수 있기 때문이다.

아무리 고민해도 모르겠는 것은 연습문제 후반부에 있는 답안을 보자. 그래도 이해가 안 되면 카페에 질문을 올리면 저자가 직접 답변해 줄 것이다.

코드초보스터디카페 활용하기

네이버에 운영하고 있는 코드초보스터디는 개설한지 20년이 넘는 장수 카페이다. 회원수는 16만이 넘고 자바관련 카페 중에서 제일 오래되었다. 그동안 많은 스터디와 세미나를 해왔고, 자바 관련 질문이나 고민에 답변해주었다. 특히 본인이 집필한 책에 대한 질문은 거의 본인이 직접 답변해주었고, 다른 회원이 답변해준 내용은 바르게 답변했는지 확인한다.

제목	작성자	작성일	조회	좋아요
[공지] 자바의 정석 3판 2016년 1월 25일에 출시예정입니다. [185]	남궁성	2015.11.03	2821	0
[공지] [질의] 자동등업방법입니다. 여기에 댓글 5개달아주세요. [14462]	남궁성	2015.05.29	5997	0

[ebook – 샘플. 무료 공유] 자바의 정석 4판 Java 21 올컬러 2025. 7. 7 출시 seong.namkung@gmail.com

이 책으로 공부하면서 막히는 것이 있으면, 카페에 질문을 하기 바란다. 가끔 지식인이나 다른 사이트에 이 책에 대한 질문을 하는 것을 볼 수 있는데, 가능하면 코드초보스터디에 질문해주었으면 한다. 다른 카페에 폐를 끼치는 것이기도 하고, 책의 저자로부터 직접 답변을 받을 수 있는데, 실력이 어떤지도 모르는 사람에게 답변을 받을 이유가 없기 때문이다.

카페에는 질문 답변 외에도 많은 자료가 있으므로 카페를 찬찬히 잘 둘러보길 권한다. ‘초보프로그래머에게’나, ‘면접후기’, ‘자바소스강좌’, ‘Java1000제’ 같은 게시판은 좋은 글들이 많다.

질문 올리는 방법

책과 관련된 질문은 저자가 빠짐없이 다 확인하고 답변하기 때문에, 답변을 못 받을까봐 걱정하지 않아도 된다. 다만 질문하기 전에 유사한 질문이 없었는지 검색으로 확인하자.

책의 페이지로만 검색해도 이미 답변된 글들을 찾아서 볼 수 있으므로 답변해줄 때까지 기다리지 않아도 된다.

책과 관련되지 않은 답변은 카페에서 비교적 오래 활동해온 회원들이 해주는 경우가 많으나, 답변을 잘 받으려면, 읽는 사람 입장에서 생각하고 자신의 생각을 잘 정리해서 질문하면 된다. 수려한 문장에 맞춤법까지 완벽해야 좋은 질문이 아니고, 읽는 사람 입장에서 답변하기 쉽게 하는 질문이 좋은 질문이다.

급한 마음은 알겠지만, 자신이 올린 질문을 다시 한 번 읽어보고 어떤 작업을 하는 도중에 어떤 문제가 발생했는지를 잘 정리해보자. 그러는 과정에서 스스로 문제가 해결되는 경우도 많다. 여러 메시지가 발생하는 경우는 꼭 같이 올리도록 하고, 소스를 포함시키되 소스가 길다면, 문제가 되는 부분만 따로 떼어서 테스트할 수 있게 올리는 것이 좋다.

앞으로 프로그래밍을 계속할거라면, 카페에서 뿐만 아니라 학교 선배나, 회사 선배, 직장 동료 등 많은 사람에게 질문을 하고 배워야 한다. 실력을 빠르게 향상시키려면, 질문을 잘하는 능력은 필수적이다.

마지막으로 독자 여러분에게 하고 싶은 당부의 말은 ‘길을 잊지 말자’는 것과 ‘남과 비교하지 말자’라는 것이다. 공부하다가 막힌다고 다른 책을 보고 수학공부하고 그러지 말라는 뜻이다. 공부하다 막히면 카페에 와서 질문을 하기 바란다. 저자 본인은 항상 여러분의 질문을 20년 넘게 한결같이 같은 곳에서 기다리고 있다. 책을 읽다가 어려움이 있으면, 카페에 와서 질문을 하자. 간단한 것일지라도. 어렵다는 말만 반복하면서 질문 한번 안하는 회원들을 많이 봐왔는데, 질문을 부끄러워하지 않았으면 한다.

그리고, 사람마다 타고난 장점이 다르고 성장하는 속도가 다르다. 남들과 비교하지 말고 어제의 자신과 오늘의 자신을 비교하면서 한발 한발 나아가기 바란다.

Memo

Java

Programming Language

Chapter 02

변수
variable

1. 변수(variable)

중요한 프로그래밍 능력 중의 하나가 바로 ‘값(data)을 잘 다루는 것’이다. 값을 저장하는 공간인 변수를 잘 이해하고 활용하는 것은 그 능력을 얻기 위한 첫걸음이니 첫단추를 잘 끼워보자.

1.1 변수(variable)란?

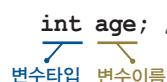
수학에서 ‘변수(變數)’를 ‘변하는 수’라고 정의하지만 프로그래밍언어에서의 변수(variable)란, 값을 저장할 수 있는 메모리상의 공간을 의미한다. 이 공간에 저장된 값은 변경될 수 있기 때문에 ‘변수’라는 수학용어의 정의와 상통하는 면이 있어서 이렇게 이름 붙여졌다.

“변수란, 단 하나의 값을 저장할 수 있는 메모리 공간.”

하나의 변수에 단 하나의 값만 저장할 수 있으므로, 새로운 값을 저장하면 기존의 값을 사라진다.

1.2 변수의 선언과 초기화

변수를 사용하려면 먼저 변수를 선언해야하는데, 변수의 선언방법은 다음과 같다.

int age; // age라는 이름의 변수를 선언

변수타입 변수이름

‘변수타입’은 변수의 ‘종류’를 지정하는 것이다. 저장하고자 하는 값의 종류에 맞게 변수의 타입을 선택해서 적어주면 된다. 변수는 값을 담기 위한 그릇이므로 어떤 값을 담을 것인지에 따라 그릇의 종류, 즉 변수의 타입이 결정된다.

위의 문장은 변수 ‘age’를 선언한다. 이 변수는 ‘나이’를 저장하기 위한 것이고, 나이는 ‘정수(integer)’이므로 변수의 타입을 ‘int’로 하였다. ‘타입(type)’에 대해서는 곧 자세히 다룰 것이므로 지금은 정수를 저장하려면 변수의 타입을 ‘int’로 한다는 정도만 알아두자.

‘변수이름’은 말 그대로 변수에 붙인 이름이다. 변수는 ‘값을 저장할 수 있는 메모리 공간’이므로 변수의 이름은 저장 공간에 이름을 붙여주는 것이다. 그래야 그 이름을 이용해서 저장 공간(변수)에 값을 저장하고, 저장된 값을 읽어오기도 할 수 있는 것이다. 당연한 얘기지만 같은 이름의 변수가 존재하면 안된다. 서로 구별될 수 있어야하기 때문이다.

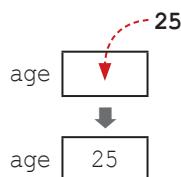
변수를 선언하면, 메모리의 빈 공간에 ‘변수타입’에 알맞은 크기의 저장 공간이 확보되고, 이 저장공간에 값을 읽고 쓰는 것은 ‘변수이름’을 통해서 가능하다. 저장 공간을 만드는 방법을 배웠으니 이제 이 저장 공간에 값을 저장하고 읽는 방법을 배워보자.

변수의 초기화

변수를 선언한 이후부터는 변수를 사용할 수 있으나, 그 전에 반드시 변수를 ‘초기화(initialization)’해야 한다. 메모리는 여러 프로그램이 공유하는 자원이므로 전에 다른 프로그램에 의해 저장된 ‘알 수 없는 값(쓰레기값, garbage value)’이 남아있을 수 있기 때문이다.

변수에 값을 저장할 때는 대입 연산자 ‘=’를 이용한다. 수학에서는 양변의 값이 같다는 뜻이지만, 자바에서는 오른쪽의 값을 왼쪽(변수)에 저장하라는 뜻이다. 그래서 대입 연산자의 왼쪽에는 반드시 변수가 와야 한다.

`int age = 25; // 변수 age를 선언하고 25로 초기화 한다.`



양쪽의 코드는 서로 같은 의미의 다른 코드이다. 변수는 한 줄에 하나씩 선언하는 것이 보통이지만, 타입이 같은 경우 콤마,’를 구분자로 여러 변수를 한 줄에 선언하기도 한다.

```
int a;
int b;
int x = 0;
int y = 0;
```



```
int a, b;
int x = 0, y = 0;
```

변수의 종류에 따라 변수의 초기화를 생략할 수 있는 경우도 있지만, 변수는 사용되기 전에 적절한 값으로 초기화 하는 것이 좋다.

| 참고 | 지역 변수는 사용되기 전에 초기화를 반드시 해야 하지만 클래스 변수와 인스턴스 변수는 초기화를 생략할 수 있다. 변수의 초기화에 대해서는 6장에서 자세히 배운다.

“변수의 초기화란, 변수를 사용하기 전에 처음으로 값을 저장하는 것”

지금까지 변수를 선언하고 값을 저장하는 방법을 알아봤으니, 이제 예제를 통해 변수에 저장된 값을 어떻게 읽어오는지 알아보자.

▼ 예제 2-1/VarEx.java

```
class VarEx {
    public static void main(String[] args) {
        int year = 0;
        int age = 14;

        System.out.println(year);
        System.out.println(age);
```

```
    year = age + 2000;    // 변수 age의 값에 2000을 더해서 변수 year에 저장
    age  = age + 1;       // 변수 age에 저장된 값을 1증가시킨다.

    System.out.print(year); // 값을 출력하고 줄바꿈을 하지않는다.
    System.out.print(age);
}

}
```



두 개의 변수 age와 year를 선언한 다음, 값을 저장하고 출력하는 간단한 예제이다. 먼저 변수의 선언부분을 보면, 변수 year와 age를 각각 0과 14로 초기화 하였다.

```
int year = 0;  
int age = 14;
```

year 0 age 14

앞에서 살펴본 것처럼 화면에 글자를 출력하려면 `println()`을 사용한다. `println()`은 값을 출력하고 줄바꿈을 하지만 `print()`은 줄바꿈을 하지 않는다는 차이가 있다.

```
→ System.out.println(age); // 변수 age의 값을 먼저 읽어와야 출력 가능  
→ System.out.println(14); // 14를 화면에 출력
```

그 다음의 문장은 변수 age에 저장된 값에 2000을 더한 다음, 그 결과를 변수 year에 저장하라는 뜻이다.

```
year = age + 2000;
```

이 문장이 처리되는 과정을 단계별로 살펴보면 다음과 같다.

```
year = age + 2000; // 변수 age의 값을 알아야 덧셈이 가능하다.  
→ year = 14 + 2000; // 변수 age에 저장된 값(14)을 읽어온다.  
→ year = 2014; // 변수 year에 2014를 저장한다.
```

위의 과정에서 알 수 있듯이 변수에 저장된 값을 사용하려면, 그저 변수의 이름만 적어주면 된다. 그리고 변수에 값을 저장하는 ‘대입 연산(=)’은 우변의 모든 계산이 끝난 후에 제일 마지막에 수행된다

다음의 코드는 변수 age에 저장된 값을 1증가 시키는데, 변수의 값을 읽어다 1을 더한 다음 다시 변수 age에 저장하라는 뜻이다

```
age = age + 1; // 변수 age의 값을 1 증가
```

위 문장이 철립되는 과정을 단계별로 살펴보면 다음과 같다.

```
age = age + 1;  
→ age = 14 + 1;    // 변수 age에 저장된 값을 읽어온다.  
→ age = 15;        // 변수 age에 15를 저장하다
```

두 변수의 값 교환하기

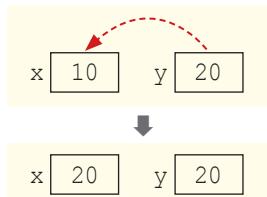
다음과 같이 변수 x, y가 있을 때, 두 변수에 담긴 값을 서로 바꾸려면 어떻게 해야 할까?
잠시 시간을 갖고 생각해보자.

```
int x = 10;
int y = 20;
```

단순하게 변수 x의 값을 y에 저장하고, y의 값을 x에 저장하면 될 것 같지만 그렇게 해서는 원하는 결과를 얻을 수 없다.

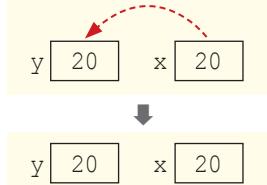
1. 변수 y에 저장된 값을 변수 x에 저장

```
x = y;
```



2. 변수 x에 저장된 값을 변수 y에 저장

```
y = x;
```



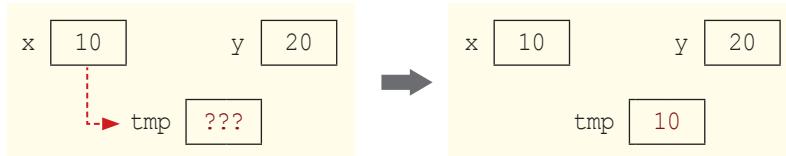
첫 번째 단계에서 y의 값을 x에 저장할 때, 이미 x의 값이 없어졌기 때문에 x의 값을 y에 저장해도 소용이 없는 것이다. 그러면 어떻게 해야 할까? 다음과 같이 변수를 하나 더 선언해서 x의 값을 위한 임시 저장소로 사용하면 된다.

```
int x = 10;
int y = 20;
int tmp; // x의 값을 임시로 저장할 변수를 선언
```

그 다음에 아래와 같은 순서로 값을 옮기면 된다.

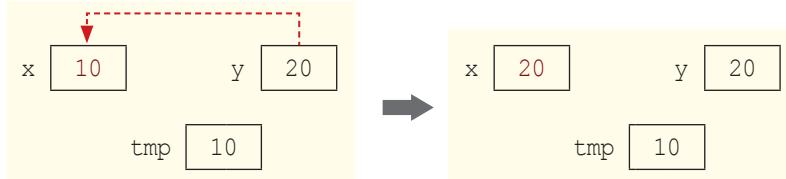
1. 변수 x에 저장된 값을 변수 tmp에 저장

```
tmp = x;
```



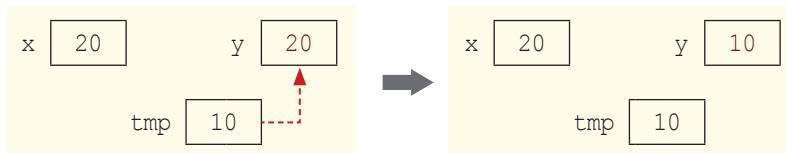
2. 변수 y에 저장된 값을 변수 x에 저장

```
x = y;
```



3. 변수 tmp에 저장된 값을 변수 y에 저장

```
y = tmp;
```



두 변수의 값을 교환하는 것은 마치 두 컵에 담긴 내용물을 바꾸려면 컵이 하나 더 필요한 것과 같다. 아직도 위의 과정이 잘 이해되지 않는다면, 한 컵에는 우유를 다른 한 컵에는 물을 담은 다음에 빈 컵을 이용해서 두 컵의 내용물을 바꾸는 일을 직접 해보자.

▼ 예제 2-2/VarEx2.java

```
class VarEx2 {
    public static void main(String[] args) {
        int x = 10, y = 20;
        int tmp = 0;

        System.out.println("x:" + x + " y:" + y);

        tmp = x;
        x = y;
        y = tmp;

        System.out.println("x:" + x + " y:" + y);
    }
}
```

▼ 실행결과

```
x:10 y:20
x:20 y:10
```

앞서 설명한 내용을 예제로 작성한 것이므로 이해하는데 어려움이 없을 것이다. 한 가지 설명할 것이 있다면 변수의 값을 출력하는 문장인데, 이 문장이 수행되는 과정을 단계별로 적어보면 다음과 같다.

```

System.out.println("x:" + x + " y:" + y);
→ System.out.println("x:" + 10 + " y:" + 20);
→ System.out.println("x:10" + " y:" + 20);
→ System.out.println("x:10 y:" + 20);
→ System.out.println("x:10 y:20");

```

덧셈 연산자 '+'는 두 값을 더하기도 하지만, 이처럼 문자열과 숫자를 하나로 결합하기도 한다. 문자열은 큰따옴표 “ ”로 묶은 연속된 문자를 말하는데, 문자열과 문자열 결합에 대한 자세한 내용은 잠시 후에 다룬다.

1.3 변수의 명명규칙

‘변수의 이름’처럼 프로그래밍에서 사용하는 모든 이름을 ‘식별자(identifier)’라고 하며, 식별자는 같은 영역 내에서 서로 구분(식별)될 수 있어야한다. 그리고 식별자를 만들 때는 다음과 같은 규칙을 지켜야 한다.

1. 대소문자가 구분되며 길이에 제한이 없다.
 - True와 true는 서로 다른 것으로 간주된다.
2. 예약어를 사용해서는 안 된다.
 - true는 예약어라서 사용할 수 없지만, True는 가능하다.
3. 숫자로 시작해서는 안 된다.
 - top10은 허용하지만, 7up은 허용되지 않는다.
4. 특수문자는 '_'와 '\$'만을 허용한다.
 - \$sharp은 허용되지만, S#arp은 허용되지 않는다.

예약어는 ‘리저브드 워드(reserved keyword)’라고 하는데, 프로그래밍언어의 구문에 사용되는 단어를 뜻한다. 그래서 예약어는 클래스나 변수, 메서드의 이름(identifier)으로 사용할 수 없다. 예약어는 앞으로 하나씩 배울 것이므로 지금은 간단히 훑어보기만 하자.

abstract	default	goto	protected	throws
assert	do	implements	public	transient
boolean	double	import	return	try
break	else	instanceof	short	void
byte	enum	int	static	volatile
case	extends	interface	strictfp	while
catch	final	long	super	–
char	finally	native	switch	
class	float	new	synchronized	
const	for	package	this	
continue	if	private	throw	

▲ 표2-1 java에서 사용되는 예약어(reserved keyword)

| 참고 | goto와 const는 실제로 사용되지 않으며, true, false, null은 예약어가 아니라 리터럴이며 이름으로 사용할 수 없다.
| 참고 | 표2-1의 마지막 항목은 언더스코어(underscore) '_'이며, JDK 9부터 예약어가 되었다.

그 외에 필수적인 것은 아니지만 자바 프로그래머들에게 권장하는 규칙들은 다음과 같다.

1. 클래스 이름의 첫 글자는 항상 대문자로 한다.
 - 변수와 메서드의 이름의 첫 글자는 항상 소문자로 한다.
2. 여러 단어로 이루어진 이름은 단어의 첫 글자를 대문자로 한다.
 - lastIndexOf, StringBuffer
3. 상수의 이름은 모두 대문자로 한다. 여러 단어로 이루어진 경우 '_'로 구분한다.
 - PI, MAX_NUMBER

위의 규칙들은 반드시 지켜야 하는 것은 아니지만, 코드를 보다 이해하기 쉽게 하기 위한 자바 개발자들 사이의 암묵적인 약속이다. 이 규칙을 따르지 않는다고 해서 문제가 되는 것은 아니지만 가능하면 지키도록 노력하자. 만일 특별한 방식으로 식별자를 작성해야 한다면 미리 규칙(coding convention)을 정해놓고 프로그램 전체에 일관되게 적용하는 것이 필요하다.

| 참고 | 자바에서는 모든 이름에 유니코드에 포함된 문자들을 사용할 수 있지만, 적어도 클래스 이름은 ASCII코드(영문자)로 하는 것이 좋다. 유니코드를 인식하지 못하는 운영체계(OS)도 있기 때문이다.

변수의 이름은 짧을수록 좋지만, 약간 길더라도 용도를 알기 쉽게 ‘의미있는 이름’으로 하는 것이 바람직하다. 변수의 선언문에 주석으로 변수에 대한 정보를 주는 것도 좋은 생각이다.

```
int curPos = 0;           // 현재 위치(current position)
int lastPos = -1;         // 마지막 위치(last position)
```

예약어와 문맥 예약어

표2-1에서 소개한 예약어는 항상 이름으로 사용할 수 없지만, 표2-2의 문맥 예약어 17개는 문맥에 따라 예약어가 되기도 하고 예약어가 아니기도 하다. JDK 8 이후로 새로 추가된 기능때문에 추가된 예약어이므로 기존 코드와 충돌을 피하기 위해 특정 문맥에서만 예약어로 동작하게 되어 있다.

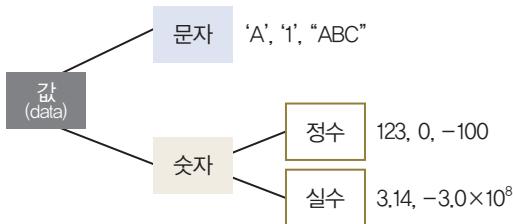
exports 9	opens 9	requires 9	uses 9	yield 14
module 9	permits 17	sealed 17	var 10	
non-sealed 17	provides 9	to 9	when 21	
open 9	record 16	transitive 9	with 9	

▲ 표2-2 java에서 사용되는 문맥 예약어 (contextual keyword)

| 참고 | 표의 숫자는 각 문맥 예약어가 추가된 JDK 버전을 의미하며, 자바 언어 명세(Java Language Spec)을 참고하였다. (<https://docs.oracle.com/javase/specs/jls/se21/html/jls-3.html#jls-3.9>)

2. 변수의 타입

우리가 주로 사용하는 값(data)의 종류(type)는 크게 ‘문자와 숫자’로 나눌 수 있으며, 숫자는 다시 ‘정수와 실수’로 나눌 수 있다.



▲ 그림2-1 값의 종류

이러한 값(data)의 종류(type)에 따라 값이 저장될 공간의 크기와 저장형식을 정의한 것이 자료형(data type)이다. 자료형에는 문자형(char), 정수형(byte, short, int, long), 실수형(float, double) 등이 있으며, 변수를 선언할 때는 저장하려는 값의 특성을 고려하여 가장 알맞은 자료형을 선택하면 된다.

기본형과 참조형

자료형은 크게 ‘기본형’과 ‘참조형’ 두 가지로 나눌 수 있는데, 기본형 변수는 실제 값(data)을 저장하는 반면, 참조형 변수는 어떤 값이 저장되어 있는 주소(memory address)를 값으로 갖는다. 자바는 C언어와 달리 참조형 변수 간의 연산을 할 수 없으므로 실제 연산에 사용되는 것은 모두 기본형 변수이다.

| 참고 | 메모리에는 1 byte 단위로 일련 번호가 붙어 있는데, 이 번호를 ‘메모리 주소(memory address)’ 또는 간단히 ‘주소’라고 한다. 객체의 주소는 객체가 저장된 메모리 주소를 뜻한다.

기본형(primitive type)

- 논리형(boolean), 문자형(char), 정수형(byte,short,int,long), 실수형(float,double)
계산을 위한 실제 값을 저장한다. 모두 8개

참조형(reference type)

- 객체의 주소를 저장한다. 8개의 기본형을 제외한 나머지 타입.

참조형 변수(또는 참조 변수)를 선언할 때는 변수의 타입으로 클래스의 이름을 사용하므로 클래스의 이름이 참조 변수의 타입이 된다. 그래서 새로운 클래스를 작성한다는 것은 새로운 참조형을 추가하는 셈이다.

다음은 참조 변수를 선언하는 방법이다. 기본형 변수와 같이 변수 이름 앞에 타입을 적어 주는데 참조 변수의 타입은 클래스의 이름이다.

클래스이름 변수이름; // 변수의 타입이 기본형이 아닌 것들은 모두 참조변수

다음은 Date클래스 타입의 참조 변수 today를 선언한 것이다. 참조 변수는 null 또는 객체의 주소를 값으로 갖으며 참조 변수의 초기화는 다음과 같이 한다.

```
Date today = new Date(); // Date객체를 생성하고, 그 주소를 today에 저장
```

객체를 생성하는 연산자 new의 결과는 생성된 객체의 주소이다. 이 주소가 대입 연산자 '='에 의해서 참조 변수 today에 저장되는 것이다. 이제 참조 변수 today를 통해서 생성된 객체를 사용할 수 있게 된다.

| 참고 | 참조형 변수는 null 또는 객체의 주소(4 byte, 0x0~0xFFFFFFFF)를 값으로 갖는다. null은 어떤 객체의 주소도 저장되어 있지 않음을 뜻한다. 단, JVM이 32 bit가 아니라 64 bit라면 참조형 변수의 크기는 8 byte가 된다.

Q. 자료형(data type)과 타입(type)의 차이가 뭔가요?

A. 기본형은 저장할 값(data)의 종류에 따라 구분되므로 기본형의 종류를 얘기할 때는 '자료형(data type)'이라는 용어를 씁니다. 그러나 참조형은 항상 '객체의 주소(4 byte 정수)'를 저장하므로 값(data)이 아닌, 객체의 종류에 의해 구분되므로 참조형 변수의 종류를 구분할 때는 '타입(type)'이라는 용어를 사용합니다. '타입(type)'이 '자료형(data type)'을 포함하는 보다 넓은 의미의 용어이므로 굳이 구분하지 않아도 됩니다.

2.1 기본형(primitive type)

기본형에는 모두 8개의 타입(자료형)이 있으며, 크게 논리형, 문자형, 정수형, 실수형으로 구분된다.

분류	타입
논리형	boolean true와 false 중 하나를 값으로 갖으며, 조건식과 논리적 계산에 사용
문자형	char 문자를 저장하는데 사용되며, 변수에 하나의 문자만 저장가능
정수형	byte, short, int, long 정수를 저장하는데 사용되며, 주로 int를 사용. byte는 이진 데이터를 다룰 때 사용되며, short은 C언어와의 호환을 위해서 추가
실수형	float, double 실수를 저장하는데 사용되며, 주로 double을 사용

▲ 표2-3 기본형의 종류

문자형인 char는 문자를 내부적으로 정수(유니코드)로 저장하기 때문에 정수형과 별반 다르지 않으며, 정수형 또는 실수형과 연산도 가능하다. 반면에 boolean은 다른 기본형과의 연산이 불가능하다. 즉, boolean을 제외한 나머지 7개의 기본형은 서로 연산과 변환이 가능하다.

정수는 가장 많이 사용되므로 타입을 4가지나 제공한다. 각 타입마다 저장할 수 있는 값의 범위가 다르므로 저장할 값의 범위에 맞는 타입을 선택하면 되지만, 일반적으로 int를 많이 사용한다. 왜냐하면, int는 CPU가 가장 효율적으로 처리할 수 있는 타입이기 때문이다. 효율적인 실행보다 메모리를 절약하려면, byte나 short을 선택하자.

| 참고 | 4개의 정수형(byte, short, int, long) 중에서 int형이 기본 자료형(default data type)이며, 실수형(float, double) 중에서는 double이 기본 자료형이다.

크기 종류	1 byte	2 byte	4 byte	8 byte
논리형	boolean			
문자형		char		
정수형	byte	short	int	long
실수형			float	double

▲ 표2-4 기본형의 종류와 크기

기본 자료형의 종류와 크기는 반드시 외워야 하며, 아래의 문장들이 도움이 될 것이다.

- ▶ boolean은 true와 false 두 가지 값만 표현할 수 있으면 되므로 가장 작은 크기인 1 byte.
- ▶ char은 자바에서 유니코드(2 byte 문자체계)를 사용하므로 2 byte.
- ▶ byte는 크기가 1 byte라서 byte.
- ▶ int(4 byte)를 기준으로 짧아서 short(2 byte), 길어서 long(8 byte). (short ↔ long)
- ▶ float는 실수값을 부동소수점(floating-point)방식으로 저장하기 때문에 float.
- ▶ double은 float보다 두 배의 크기(8 byte)를 갖기 때문에 double.

그리고 각 타입의 변수가 저장할 수 있는 값의 범위는 다음과 같다.

자료형	저장 가능한 값의 범위	크기	
		bit	byte
boolean	false, true	8	1
char	'\u0000' ~ '\uffff' (0~2 ¹⁶ -1, 0~65535)	16	2
byte	-128 ~ 127 (-2 ⁷ ~2 ⁷ -1)	8	1
short	-32,768 ~ 32,767 (-2 ¹⁵ ~2 ¹⁵ -1)	16	2
int	-2,147,483,648 ~ 2,147,483,647 (-2 ³¹ ~2 ³¹ -1, 약 ±20억)	32	4
long	-9,223,372,036,854,775,808 ~ 9,223,372,036,854,775,807 (-2 ⁶³ ~2 ⁶³ -1)	64	8
float	1.4E-45 ~ 3.4E38 (1.4×10 ⁻⁴⁵ ~3.4×10 ³⁸)	32	4
double	4.9E-324 ~ 1.8E308 (4.9×10 ⁻³²⁴ ~1.8×10 ³⁰⁸)	64	8

▲ 표2-5 기본형의 크기와 범위

| 참고 | float와 double은 양의 범위만 적은 것이다. 음의 범위는 양의 범위에 음수 부호(-)를 붙이면 된다.

각 자료형이 가질 수 있는 값의 범위를 정확히 외울 필요는 없고, 정수형(byte, short, int, long)의 경우 '-2ⁿ⁻¹ ~ 2ⁿ⁻¹-1'(n은 bit수)이라는 정도만 기억하고 있으면 된다.

예를 들어 int형의 경우 32 bit(4 byte)이므로 '-2³¹ ~ 2³¹-1'의 범위를 갖는다.

$$2^{10} = 1024 \approx 10^3 \text{이므로, } 2^{31} = 2^{10} \times 2^{10} \times 2^{10} \times 2 = 1024 \times 1024 \times 1024 \times 2 \approx 2 \times 10^9$$

따라서 int타입의 변수는 대략 10자리 수(약 20억, 2,000,000,000)의 값을 저장할 수 있다는 것을 알 수 있다. 7~9자리의 수를 계산할 때는 넉넉하게 long타입(약 19자리)으로 변수를 선언하는 것이 좋다. 연산중에 저장범위를 넘어서게 되면 원하지 않는 값을 결과로 얻게 될 것이기 때문이다.

실수형은 정수형과 저장형식이 달라서 같은 크기라도 훨씬 큰 값을 표현할 수 있으나 오차가 발생할 수 있다는 단점이 있다. 그래서 정밀도(precision)가 중요한데, 정밀도가 높을수록 발생하는 오차의 범위가 줄어든다. 아래의 표를 보면 float의 정밀도는 7자리인데, 이것은 10진수로 7자리의 수를 오차없이 저장할 수 있다는 뜻이다.

자료형	저장 가능한 값의 범위	정밀도	크기	
			bit	byte
float	$1.4E-45 \sim 3.4E38$ ($1.4 \times 10^{-45} \sim 3.4 \times 10^{38}$)	7 자리	32	4
double	$4.9E-324 \sim 1.8E308$ ($4.9 \times 10^{-324} \sim 1.8 \times 10^{308}$)	15 자리	64	8

▲ 표2-6 실수형의 범위와 정밀도

float는 약 $\pm 10^{38}$ 과 같이 큰 값을 저장할 수 있지만, 정밀도가 7자리 밖에 되지 않으므로 보다 높은 정밀도가 필요한 경우에는 변수의 타입으로 double을 선택해야한다. 이처럼 실수형에서는 저장 가능한 값의 범위뿐만 아니라 정밀도도 타입 선택의 중요한 기준이 된다.

2.2 상수와 리터럴(constant & literal)

‘상수(constant)’는 변수와 마찬가지로 ‘값을 저장할 수 있는 공간’이지만, 변수와 달리 한번 값을 저장하면 다른 값으로 변경할 수 없다. 상수를 선언하는 방법은 변수와 동일하며, 단지 변수의 타입 앞에 키워드 ‘final’을 붙이면 된다.

```
final int MAX_SPEED = 10; // 상수 MAX_SPEED를 선언 & 초기화
```

그리고 상수는 선언과 동시에 초기화하는 것이 보통이며, 한번 값이 저장되고 나면 상수의 값을 변경하는 것이 허용되지 않는다.

```
final int MAX_VALUE; // 상수 MAX_SPEED를 선언
MAX_VALUE = 100; // 상수 MAX_SPEED를 초기화
MAX_VALUE = 200; // 에러. 상수의 값은 변경할 수 없음
```

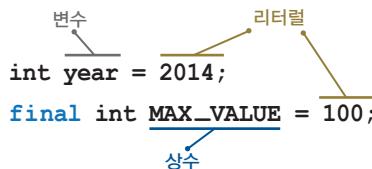
상수의 이름은 모두 대문자로 하는 것이 암묵적인 관례이며, 여러 단어로 이루어져있는 경우 ‘_’로 구분하는 것이 일반적이다.

| 참고 | JDK 6부터 상수를 선언과 동시에 초기화 하지 않아도 사용하기 전에만 초기화하면 되도록 바뀌었다. 그래도 상수는 선언과 동시에 초기화하는 습관을 들이는 것이 좋다.

리터럴(literal)

원래 12, 123, 3.14, 'A'와 같은 값들이 '상수'인데, 프로그래밍에서는 상수를 '값을 한 번 저장하면 변경할 수 없는 저장공간'으로 정의하였기 때문에 이와 구분하기 위해 상수를 다른 이름으로 불러야만 했다. 그래서 상수 대신 리터럴이라는 용어를 사용한다. 많은 사람들이 리터럴이라는 용어를 어려워하는데, 리터럴은 단지 우리가 기존에 알고 있던 '상수'의 의미하는 다른 용어일 뿐이다.

변수(variable)	하나의 값을 저장하기 위한 공간
상수(constant)	값을 한번만 저장할 수 있는 공간
리터럴(literal)	그 자체로 값을 의미하는 것



상수가 필요한 이유

아마도 이쯤에서 여러분들은 '그냥 리터럴을 직접 쓰면 될 텐데, 굳이 상수가 따로 필요한가?'라는 의문이 들 것도 같다. 먼저 다음의 코드를 보자.

```

int triangleArea = (20 * 10) / 2; // 삼각형의 면적을 구하는 공식
int rectangleArea = 20 * 10; // 사각형의 면적을 구하는 공식
    
```

위의 코드는 삼각형과 사각형의 면적을 구해서 변수에 저장한다. 이 공식을 모르는 사람은 없겠지만, 보다 복잡한 공식이라면 얘기가 달라질 것이다. 게다가 20과 10이 아닌 다른 값을 이용해서 결과를 얻고 싶다면 여러 곳을 수정해야한다.

그러면 이제 다음의 코드를 보자.

```

final int WIDTH = 20; // 폭
final int HEIGHT = 10; // 높이

int triangleArea = (WIDTH * HEIGHT) / 2; // 삼각형의 면적을 구하는 공식
int rectangleArea = WIDTH * HEIGHT; // 사각형의 면적을 구하는 공식
    
```

상수를 이용해서 기존의 코드를 변경한 것인데, 이전 코드에 비해 면적을 구하는 공식의 의미가 명확해졌다. 그리고 다른 값으로 계산할 때도 여러 곳을 수정할 필요없이 상수의 초기화만 다른 값으로 해주면 된다.

이처럼 상수는 리터럴에 '의미있는 이름'을 붙여서 코드의 이해와 수정을 쉽게 만든다.

리터럴의 타입과 접미사

변수에 타입이 있는 것처럼 리터럴에도 타입이 있다. 변수의 타입은 저장될 ‘값의 타입(리터럴의 타입)’에 의해 결정되므로, 만일 리터럴에 타입이 없다면 변수의 타입도 필요없을 것이다.

종류	리터럴	접미사
논리형	false, true	없음
정수형	123, 0b0101, 077, 0xFF, 100L	L
실수형	3.14, 3.0e8, 1.4f, 0x1.0p-1	f, d
문자형	'A', '1', '\n'	없음
문자열	"ABC", "123", "A", "true"	없음

▲ 표2-7 리터럴과 접미사

정수형과 실수형에는 여러 타입이 존재하므로, 리터럴에 접미사를 붙여서 타입을 구분한다. 정수형의 경우, long타입의 리터럴에 접미사 ‘I’ 또는 ‘L’을 붙이고, 접미사가 없으면 int타입의 리터럴이다. byte와 short타입의 리터럴은 별도로 존재하지 않으며 byte와 short타입의 변수에 값을 저장할 때는 int타입의 리터럴을 사용한다.

10진수 외에도 2, 8, 16진수로 표현된 리터럴을 변수에 저장할 수 있으며, 16진수라는 것을 표시하기 위해 리터럴 앞에 접두사‘0x’ 또는 ‘0X’를, 8진수의 경우에는 ‘0’을 붙인다.

| 참고 | 접두사‘0x’,‘0X’,‘0b’,‘0B’,‘0’의 ‘0’은 알파벳이 아니라 숫자이다. 2진 리터럴은 JDK 7부터 추가되었다.

```
int octNum = 010;           // 8진수 10, 10진수로 8
int hexNum = 0x10;          // 16진수 10, 10진수로 16
int binNum = 0b10;          // 2진수 10, 10진수로 2
```

그리고 JDK 7부터 정수형 리터럴의 중간에 구분자‘_’를 넣을 수 있어서 큰 숫자를 편하게 읽을 수 있게 되었다.

```
long big = 100_000_000_000L;      // long big = 100000000000L;
long hex = 0xFFFF_FFFF_FFFF_FFFF; // long hex = 0xFFFFFFFFFFFFFFFL;
```

실수형에서는 float타입의 리터럴에 접미사 ‘f’ 또는 ‘F’를 붙이고, double타입의 리터럴에는 접미사 ‘d’ 또는 ‘D’를 붙인다.

```
float pi      = 3.14f;           // 접미사 f 대신 F를 사용해도 된다.
double rate   = 1.618d;          // 접미사 d 대신 D를 사용해도 된다.
```

실수형 리터럴에는 접미사를 붙여서 타입을 구분하며, float타입 리터럴에는 ‘f’를, double 타입 리터럴에는 ‘d’를 붙인다. 정수형에서는 int가 기본 자료형인 것처럼 실수형에서는 double이 기본 자료형이라서 접미사‘d’는 생략이 가능하다. 실수형 리터럴인데, 접미사가 없으면 double타입 리터럴인 것이다.

```
float pi = 3.14; // 에러. float타입 변수에 double타입 리터럴 저장불가
double rate = 1.618; // OK. 접미사 d는 생략할 수 있다.
```

위의 문장에서 3.14는 접미사가 붙지 않았으므로 float타입 리터럴이 아니라 double타입 리터럴로 간주된다. 그래서 3.14가 float타입의 범위에 속한 값임에도 불구하고 컴파일 시에 에러가 발생한다. 에러를 피하려면 3.14f와 같이 접미사를 붙여야 한다.

리터럴의 접두사와 접미사는 대소문자를 구별하지 않으므로, 대문자와 소문자 중에서 어떤 것을 사용해도 상관없지만, 소문자'i'의 경우 숫자 '1'과 헷갈리기 쉬우므로 대문자인 'L'을 사용하는 것이 좋다.

| 참고 | 리터럴에 접미사가 붙는 타입은 long, float, double뿐인데, double은 생략가능하므로 long과 float의 리터럴에 접미사를 붙이는 것만 신경쓰면 된다.

리터럴에 소수점이나 10의 제곱을 나타내는 기호 E 또는 e, 그리고 접미사 f, F, d, D를 포함하고 있으면 실수형 리터럴로 간주된다.

자료형	실수형 리터럴	동등한 표현
double	10.	10.0
double	.10	0.10
float	10f	10.0f
float	3.14e3f	3140.0f
double	1e1	10.0
double	1e-3	0.001

▲ 표2-8 실수형 리터럴의 예

타입의 불일치

리터럴의 타입은 저장될 변수의 타입과 일치하는 것이 보통이지만, 타입이 달라도 저장범위가 넓은 타입에 좁은 타입의 값을 저장하는 것은 허용된다.

```
int i = 'A'; // OK. 문자'A'의 유니코드인 65가 변수 i에 저장된다.
long l = 123; // OK. int보다 long이 더 범위가 넓다.
double d = 3.14f; // OK. float보다 double이 더 범위가 넓다.
```

그러나 리터럴의 값이 변수의 타입의 범위를 넘어서거나, 리터럴의 타입이 변수의 타입보다 저장범위가 넓으면 컴파일 에러가 발생한다.

```
int i = 0x123456789; // 에러. int의 범위를 넘는 값을 저장
float f = 3.14; // 에러. float보다 double의 범위가 넓다.
```

3.14는 3.14d에서 접미사가 생략된 것으로 double타입이다. 이 값을 float타입으로 표현할 수 있지만, double타입의 리터럴이므로 float타입의 변수에 저장할 수 없다.

| 참고 | float는 접미사나 정밀도 등 신경 쓸 것이 많다. 이런 것들이 귀찮다면 그냥 double을 사용하자.

byte와 short타입의 리터럴은 따로 존재하지 않으므로 int타입의 리터럴을 사용한다. 단, 변수가 저장할 수 있는 범위에 속한 것이어야 한다.

```
byte b = 65; // OK. byte의 범위(-128~127)에 속하는 int타입 리터럴  
short s = 0x1234; // OK. short의 범위에 속하는 int타입 리터럴
```

각 타입의 범위만 알아도 충분히 판단가능한 내용이다. 값의 크기에 상관없이 double타입의 리터럴을 float타입의 변수에 저장할 수 없다는 것만 주의하자. 보다 자세한 내용은 이 장의 마지막 단원인 ‘형변환’에서 설명한다.

지역 변수의 타입 추론 – var

JDK 10부터 지역 변수(local variable)의 경우, 변수를 선언할 때 타입 대신 ‘var’를 사용할 수 있게 되었다. 변수는 값을 담기 위한 것이고, 값의 타입과 변수의 타입이 일치하는 것이 보통이기 때문에 변수의 타입을 생략해도 컴파일러가 값의 타입을 보고 변수의 타입을 추론(inference)할 수 있는 것이다.

| 참고 | 지역 변수는 메서드 내에 선언된 변수를 의미하며, 6장 이전까지의 변수는 모두 지역 변수다.

```
var year = 2024; // 아래의 문장과 동일  
int year = 2024; // 변수의 타입과 값의 타입이 일치  
                  ↓  
                  일치
```

실제 타입 대신 ‘var’를 사용하면 코드가 간결해지고, 변경에도 유리해진다. 클래스를 다른 것으로 바꿀때 변수의 타입은 변경하지 않아도 되기 때문이다.

```
LinkedHashMap map = new LinkedHashMap(); // 변경시 두 곳을 고쳐야 함  
var map = new LinkedHashMap(); // 변경시 한 곳만 고치면 됨.
```

만일 변수를 선언할 때 값을 대입하지 않거나 null을 대입하면, 변수의 타입을 추론할 수 없기 때문에 에러가 발생한다.

```
var obj; // 에러. 변수를 초기화하지 않아서 변수의 타입 추론 불가  
var obj = null; // 에러. 변수를 null로 초기화하면 변수의 타입 추론 불가
```

그리고 아래의 경우처럼 의도했던 것과 다르게 타입이 추론될 수 있으므로 조심해야 하며 이럴 때는 ‘var’대신에 실제 타입을 적어주면 된다.

```
byte b = 123; // 123의 타입은 int지만 byte의 범위를 넘지않아서 OK  
var b = 123; // byte가 아니라 int로 추론된다.
```

문자 리터럴과 문자열 리터럴

'A'와 같이 작은따옴표로 문자 하나를 감싼 것을 '문자 리터럴'이라고 한다. 두 문자 이상은 큰 따옴표로 감싸야 하며 '문자열 리터럴'이라고 한다.

| 참고 | 문자열은 '문자의 연속된 나열'이라는 뜻이며, 영어로 'string'이다.

```
char ch = 'J'; // char ch = 'Java'; 이렇게 할 수 없다.
String name = "Java"; // 변수 name에 문자열 리터럴 "Java"를 저장
```

char타입의 변수는 단 하나의 문자만 저장할 수 있으므로, 여러 문자(문자열, 0~n개 문자)를 저장하기 위해서는 String타입을 사용해야 한다.

문자열 리터럴은 ""안에 아무런 문자도 넣지 않는 것을 허용하며, 이를 빈 문자열(empty string)이라고 한다. 그러나 문자 리터럴은 반드시 "안에 하나의 문자가 있어야한다.

```
String str = ""; // OK. 내용이 없는 빈 문자열
char ch = ' '; // 에러. ''안에 반드시 하나의 문자가 필요
char ch = ' '; // OK. 공백 문자(blank)로 변수 ch를 초기화
```

사실 String은 클래스이므로 아래와 같이 객체를 생성하는 연산자 new를 사용해야 하지 만 특별히 위와 같은 표현도 허용한다.

```
String name = new String("Java"); // String객체를 생성
```

그리고 덧셈 연산자(+)를 이용하여 문자열을 결합할 수 있어서 다음과 같이 할 수 있다.

```
String name = "Ja" + "va"; // name은 "Java"
String str = name + 21; // str은 "Java21"
```

덧셈 연산자는 피연산자가 모두 숫자일 때는 두 수를 더하지만, 피연산자 중 어느 한 쪽이 String이면 나머지 한 쪽을 먼저 String으로 변환한 다음 두 String을 결합한다.

기본형과 참조형의 구별없이 어떤 타입의 변수도 문자열과 덧셈 연산을 수행하면 그 결과가 문자열이 되는 것이다.

문자열 + any type → 문자열 + 문자열 → 문자열 any type + 문자열 → 문자열 + 문자열 → 문자열
--

예를 들어 7 + "7"을 계산할 때 7이 String이 아니므로, 먼저 7을 String으로 변환한 다음 "7" + "7"을 수행하여 "77"을 결과로 얻는다. 다음은 문자열 결합의 몇 가지 예를 보여준다.

```
7 + " " → "7" + " " → "7 "
" " + 7 → " " + "7" → " 7"
7 + "7" → "7" + "7" → "77"
```

7 + 7 + "" → 14 + "" → "14" + "" → "14"
"" + 7 + 7 → "7" + 7 → "7" + "7" → "77"

true + "" → "true" + "" → "true"
null + "" → "null" + "" → "null"

덧셈 연산자는 왼쪽에서 오른쪽의 방향으로 연산을 수행하기 때문에 결합순서에 따라 결과가 달라진다는 것에 주의하자. 그리고 7과 같은 기본형 타입의 값을 문자열로 변환할 때는 아무런 내용도 없는 빈 문자열("")을 더해주면 된다는 것도 알아두자.

▼ 예제 2-3/StringEx.java

```
class StringEx {  
    public static void main(String[] args) {  
        String name = "Ja" + "va";  
        String str = name + 21;  
  
        System.out.println(name);  
        System.out.println(str);  
        System.out.println(7 + " ");  
        System.out.println(" " + 7);  
        System.out.println(7 + "");  
        System.out.println("") + 7);  
        System.out.println("") + "");  
        System.out.println(7 + 7 + "");  
        System.out.println("") + 7 + 7);  
    }  
}
```

▼ 실행결과

Java
Java21
7
7
7
7
14
77

텍스트 블럭(text blocks)

JDK 15부터 다중행 문자열(multiline string literal)을 작성할 수 있는 기능을 제공하며, ‘텍스트 블럭(text blocks)’이라고 부른다. 이 기능은 여러 줄로 이루어진 문자열을 작성하기 편리하게 도와준다.

이 전에는 여러 줄로 이루어진 문자열을 작성하려면, 아래와 같이 특수 문자들을 섞어서 문자열을 작성해야 한다. '\n'은 줄바꿈 문자이고, '\t'은 탭 문자(키보드의 tab키)이다.

| 참고 | p.78의 표2-13에 특수 문자의 목록이 있다.

```
String str = "class Main {\n" + "\tpublic String main(String args[]) {\n" + "\t\tSystem.out.println(\"Hello\");\n" + "\t}\n" + "}"
```

이렇게 작성하면 실수하기 쉽고 보기도 불편하다. 새로 추가된 텍스트 블럭을 이용하면 특수 문자를 넣지 않고 편리하면서도 보기 좋게 다중행 문자열을 작성할 수 있다. 위의 문자열을 텍스트 블럭으로 작성하면 다음과 같다.

| 참고 | 텍스트 블럭을 작성할 때 탭 키를 입력하면 4개의 공백 문자로 간주된다.

```
String str = """
class Main {
    public String main(String args[]) {
        System.out.println("Hello");
    }
}
""";
```

텍스트 블럭의 시작과 끝을 의미하는 "" 사이에 텍스트를 탭 문자나 개행문자 없이 자유롭게 작성하면 된다. 주의할 점은 아래와 같이 여는 ""의 다음 줄부터 내용이 시작해야 한다는 것이다.

```
String str1 = """Hello"""; // 예전 텍스트 블럭의 시작 "" 뒤에 내용이 오면 안됨.
String str2 = """
Hello"""; // OK. "Hello"와 동일
```

그리고 닫는 ""의 위치에 의해 들여쓰기의 기준이 된다는 것도 주의해야 한다. 아래의 코드에서 str3는 "Hello"와 닫는 ""와 들여쓰기 위치가 같으므로 "Hello"는 들여쓰기가 없다. 반면에 str4는 ""의 들여쓰기 위치가 "Hello"보다 2칸 왼쪽에 있다. 그래서 "Hello"의 왼쪽에 공백이 2칸 추가된 것이다. 참고로 몇가지 예를 더 적었는데 가볍게 보고 넘어가길 바란다.

```
String str3 = """
    Hello
"""; // "Hello\n"와 동일
String str4 = """
    Hello
"""; // "Hello\n"와 동일. 닫는 ""의 위치가 들여쓰기의 기준
String str5 = """
    Hello
""".indent(2); // str4와 동일. 들여쓰기를 숫자로 지정 가능
String str6 = """
    Hello\
"""; // "Hello"와 동일. '\'는 한 행의 개행 문자(\n)를 제거
```

2.3 형식화된 출력 – printf()

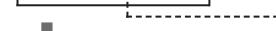
지금까지 화면에 출력할 때 println()을 써왔는데, println()은 사용하기엔 편하지만 변수의 값을 그대로 출력하므로, 값을 변환하지 않고는 다른 형식으로 출력할 수 없다. 같은 값이라도 다른 형식으로 출력하고 싶을 때가 있다. 예를 들면, 소수점 둘째자리까지만 출력한다면, 정수를 16진수나 8진수로 출력한다던가. 이럴 때 printf()를 사용하면 된다.

printf()는 ‘지시자(specifier)’를 통해 변수의 값을 여러 가지 형식으로 변환하여 출력하는 기능을 가지고 있다. ‘지시자’는 값을 어떻게 출력할 것인지를 지정해주는 역할을 한다. 정수형 변수에 저장된 값을 10진 정수로 출력할 때는 지시자 ‘%d’를 사용하며, 변수의 값을 지정된 형식으로 변환해서 지시자 대신 넣는다. 예를 들어 int 타입의 변수 age의 값이 14일 때, printf()는 지시자 ‘%d’ 대신 14를 넣어서 출력한다.

```
System.out.printf("age:%d", age);
→ System.out.printf("age:%d", 14);
→ System.out.printf("age:14"); // "age:14"가 화면에 출력된다.
```

만일 출력하려는 값이 2개라면, 지시자도 2개를 사용해야 하며 출력될 값과 지시자의 순서는 일치해야 한다. 물론 3개 이상의 값도 지시자를 지정해서 출력할 수 있으며 개수의 제한은 없다.

```
System.out.printf("age:%d year:%d", age, year);
→ System.out.printf("age:%d year:%d", 14, 2017);
```



"age:14 year:2017"이 화면에 출력된다.

println()과 달리 printf()는 출력 후 줄바꿈을 하지 않는다. 줄바꿈을 하려면 지시자 ‘%n’을 따로 넣어줘야 한다.

| 참고 | ‘%n’ 대신 ‘\n’을 사용해도 되지만, OS마다 줄바꿈 문자가 다를 수 있기 때문에 ‘%n’을 사용하는 것이 더 안전하다.

```
System.out.printf("age:%d", age); // 출력 후 줄바꿈을 하지 않는다.
System.out.printf("age:%d%n", age); // 출력 후 줄바꿈을 한다.
```

printf()의 지시자 중에서 자주 사용되는 것만 뽑아보면 다음과 같다.

| 참고 | 지시자의 전체 목록을 보려면, Java API에서 Formatter 클래스 (java.util 패키지)를 찾으면 된다.

지시자	설명
%b	불리언(boolean) 형식으로 출력
%d	10진(decimal) 정수의 형식으로 출력
%o	8진(octal) 정수의 형식으로 출력
%x, %X	16진(hexa-decimal) 정수의 형식으로 출력

%f	부동 소수점(floating-point)의 형식으로 출력
%e, %E	지수(exponent) 표현식의 형식으로 출력
%c	문자(character)로 출력
%s	문자열(string)로 출력

▲ 표2-9 자주 사용되는 printf()의 지시자

▼ 예제 2-4/PrintfEx.java

```
class PrintfEx {
    public static void main(String[] args) {
        byte b = 1;
        short s = 2;
        char c = 'A';

        int finger = 10;
        long big = 100_000_000_000L; // long big = 1000000000000L;
        long hex = 0xFFFF_FFFF_FFFF_FFFFL ;

        int octNum = 010;          // 8진수 10, 10진수로는 8
        int hexNum = 0x10;         // 16진수 10, 10진수로는 16
        int binNum = 0b10;         // 2진수 10, 10진수로는 2

        System.out.printf("b=%d%n", b);
        System.out.printf("s=%d%n", s);
        System.out.printf("c=%c, %d %n", c, (int)c);
        System.out.printf("finger=[%5d]%n", finger);
        System.out.printf("finger=[%-5d]%n", finger);
        System.out.printf("finger=[%05d]%n", finger);
        System.out.printf("big=%d%n", big);
        System.out.printf("hex=%#x%n", hex); // '#'은 접두사(16진수 0x, 8진수 0)
        System.out.printf("octNum=%o, %d%n", octNum, octNum);
        System.out.printf("hexNum=%x, %d%n", hexNum, hexNum);
        System.out.printf("binNum=%s, %d%n", Integer.toBinaryString(binNum),
                           binNum);
    }
}
```

▼ 실행결과

```
b=1
s=2
c=A, 65
finger=[ 10]
finger=[10 ]
finger=[00010]
big=100000000000
hex=0xffffffffffffffffffff
octNum=10, 8
hexNum=10, 16
binNum=10, 2
```

정수형의 값을 printf()로 출력하는 예제이다. 정수를 출력할 때는 지시자 '%d'를 사용하는데, 출력될 값이 차지할 공간을 숫자로 지정할 수 있다. 여러 값을 여러 줄로 간격 맞춰

출력할 때 꼭 필요한 기능이다. 아래의 결과를 보면 ‘0’과 ‘-’가 어떤 역할을 하는지 설명하지 않아도 알 수 있을 것이다.

```
System.out.printf("finger = [%5d]%n", finger); // finger = [ 10]
System.out.printf("finger = [%-5d]%n", finger); // finger = [10  ]
System.out.printf("finger = [%05d]%n", finger); // finger = [00010]
```

지시자 ‘%x’와 ‘%o’에 ‘#’를 사용하면 접두사 ‘0x’와 ‘0’이 각각 붙는다. 그리고 ‘%X’는 16진수에 사용되는 접두사와 영문자를 대문자로 출력한다.

```
System.out.printf("hex = %x%n", hex); // hex = ffffffff
System.out.printf("hex = %#x%n", hex); // hex = 0xffffffff
System.out.printf("hex = %#X%n", hex); // hex = 0xFFFFFFFFFFFF
```

10진수를 2진수로 출력해주는 지시자는 없기 때문에, 정수를 2진 문자열로 변환해주는 ‘Integer.toBinaryString(int i)’를 사용해야 한다. 이 메서드는 정수를 2진수로 변환해서 문자열로 반환하므로 지시자 ‘%s’를 사용했다.

```
System.out.printf("binNum=%s%n", Integer.toBinaryString(binNum));
```

그리고 C언어에서는 char타입의 값을 지시자 ‘%d’로 출력할 수 있지만, 자바에서는 허용되지 않는다. 아래와 같이 int타입으로 형변환해야만 ‘%d’로 출력할 수 있다.

```
System.out.printf("c = %c, %d %n", c, (int)c); // 형변환이 꼭 필요하다.
```

▼ 예제 2-5/PrintfEx2.java

```
class PrintfEx2 {
    public static void main(String[] args) {
        String url = "www.codechobo.com";
        float f1 = .10f; // 0.10, 1.0e-1
        float f2 = 1elf; // 10.0, 1.0e1, 1.0e+1
        float f3 = 3.14e3f;
        double d = 1.23456789;

        System.out.printf("f1=%f, %e, %g%n", f1, f1, f1);
        System.out.printf("f2=%f, %e, %g%n", f2, f2, f2);
        System.out.printf("f3=%f, %e, %g%n", f3, f3, f3);

        System.out.printf("d=%f%n", d);
        System.out.printf("d=%14.10f%n", d); // 전체 14자리 중 소수점 10자리

        System.out.printf("[12345678901234567890]%n");
        System.out.printf("[%s]%n", url);
        System.out.printf("[%20s]%n", url);
        System.out.printf("[% -20s]%n", url); // 원쪽 정렬
        System.out.printf("[%.8s]%n", url); // 원쪽에서 8글자만 출력
    }
}
```

▼ 실행결과

```
f1=0.100000, 1.000000e-01, 0.100000
f2=10.000000, 1.000000e+01, 10.0000
f3=3140.000000, 3.140000e+03, 3140.0000
d=1.234568 ← 마지막 자리 반올림됨
d= 1.2345678900
[12345678901234567890]
[www.codechobo.com]
[ www.codechobo.com ]
[www.codechobo.com ]
[www.code]
```

실수형 값의 출력에 사용되는 지시자는 '%f', '%e', '%g'가 있는데, '%f'가 주로 쓰이고 '%e'는 지수형태로 출력할 때, '%g'는 값을 간략하게 표현할 때 사용한다.

'%f'는 기본적으로 소수점 아래 6자리까지만 출력하기 때문에 소수점 아래 7자리에서 반올림한다. 그래서 1.23456789가 1.234568로 출력되었다. 그리고 다음과 같이 전체 자리수와 소수점 아래의 자리수를 지정할 수도 있다.

%전체자리.소수점아래자리 f

```
System.out.printf("d = %14.10f\n", d); // 전체 14자리 중 소수점 아래 10자리
```



소수점도 한자리를 차지하며, 소수점 아래의 빈자리는 0으로 채우고 정수의 빈자리는 공백으로 채워서 전체 자리수를 맞춘다.

| 참고 | 지시자를 '%014.10'으로 지정했다면, 양쪽 빈자리를 모두 0으로 채웠을 것이다.

지시자 '%s'에도 숫자를 추가하면 원하는 만큼의 출력공간을 확보하거나 문자열의 일부만 출력할 수 있다.

```
System.out.printf("[%s]%n", url); // 문자열의 길이만큼 출력공간을 확보  
System.out.printf("[%20s]%n", url); // 최소 20글자 출력공간 확보. (우측정렬)  
System.out.printf("[%‐20s]%n", url); // 최소 20글자 출력공간 확보. (좌측정렬)  
System.out.printf("[%.8s]%n", url); // 왼쪽에서 8글자만 출력
```

지정된 숫자보다 문자열의 길이가 작으면 빈자리는 공백으로 출력된다. 공백이 있는 경우 기본적으로 우측 끝에 문자열을 붙이지만, ‘-’를 붙이면 좌측 끝에 붙인다. 그리고 ‘.’을 붙이면 문자열의 일부만 출력할 수 있다. 숫자를 직접 바꿔가면서 다양하게 테스트 해보자.

2.4 화면에서 입력받기 – Scanner

지금까지 화면에 출력만 해왔는데, 이제 화면으로부터 입력받는 방법에 대해서 배워보자. 자바에서 화면으로부터 입력받는 방법은 여러 가지가 있으며, 점점 간단하고 편리한 방향으로 발전해 왔다. 최신 방법은 JDK 6부터 추가된 `Console` 클래스를 이용하는 것인데, 이 클래스는 인텔리제이와 같은 IDE에서 잘 동작하지 않으므로, 이와 유사한 `Scanner` 클래스를 이용해서 화면으로부터 입력받는 방법을 배워보자.

| 참고 | 화면으로부터 입력받는 방법들은 근본적으로 모두 같으므로 차이를 비교할 필요는 없다. 그저 상황에 맞는 편리한 것을 선택해서 사용하자.

화면으로부터 입력받는 방법은 아직 배우지 않은 것들을 알아야 하는데도 불구하고 본인이 직접 입력을 하면 자칫 지루해 질 수 있는 내용들이 좀 더 재미있어지지 않을까하는 생각에서 미리 소개하게 되었다. 나중에 자세히 배울 테니 지금은 이해하기보다 가져다 쓰는 정도만 알아두자.

먼저 `Scanner` 클래스를 사용하려면, 아래의 한 문장을 추가해야 한다.

```
import java.util.*; // Scanner 클래스를 사용하기 위해 추가
```

그 다음엔 `Scanner` 클래스의 객체를 생성한다.

```
Scanner scanner = new Scanner(System.in); // Scanner 클래스의 객체를 생성
```

그리고 `nextLine()`이라는 메서드를 호출하면, 입력대기 상태에 있다가 입력을 마치고 ‘엔터키(Enter)’를 누르면 입력한 내용이 문자열로 반환된다.

```
String input = scanner.nextLine(); // 입력받은 내용을 input에 저장
int num = Integer.parseInt(input); // 입력받은 내용을 int 타입의 값으로 변환
```

만일 입력받은 문자열을 숫자로 변환하려면, `Integer.parseInt()`라는 메서드를 이용해야 한다. 이 메서드는 문자열을 int 타입의 정수로 변환한다.

| 참고 | 만일 문자열을 float 타입의 값으로 변환하길 원하면, `Float.parseFloat()`를 사용해야 한다. 다른 타입으로의 변환은 p.505을 참고하자.

사실 `Scanner` 클래스에는 `nextInt()`나 `nextFloat()`와 같이 변환없이 숫자로 바로 입력받을 수 있는 메서드들이 있고, 이 메서드들을 사용하면 문자열을 숫자로 변환하는 수고는 하지 않아도 된다.

```
int num = scanner.nextInt(); // 정수를 입력받아서 변수 num에 저장
```

그러나 이 메서드들은 화면에서 연속적으로 값을 입력받아서 사용하기에 까다로울 때가 있다. 그럴 때는 모든 값을 `nextLine()`으로 입력받아서 적절히 변환하자.

▼ 예제 2-6/ScannerEx.java

```
import java.util.*;      // Scanner를 사용하기 위해 추가

class ScannerEx {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);

        System.out.print("두자리 정수를 하나 입력해주세요.>");
        String input = scanner.nextLine();
        int num = Integer.parseInt(input); // 입력받은 문자열을 숫자로 변환
//        int num = scanner.nextInt();

        System.out.println("입력내용 :" + input);
        System.out.printf("num=%d%n", num);
    }
}
```

▼ 실행결과

```
두자리 정수를 하나 입력해주세요.>22
입력내용 :22
num=22
```

만일 숫자가 아닌 문자 또는 기호를 입력하면, 입력받은 문자열을 숫자로 변환하는 과정
인 `Integer.parseInt()`에서 에러가 발생한다. 특히 공백을 입력하지 않도록 주의하자.

3. 진법

3.1 10진법과 2진법

우리는 일상생활에서 주로 사용하는 것은 10진법이다. 아마도 사람이 10개의 손가락을 가지고 있기 때문이 아닐까. 1946년에 개발된 컴퓨터인 에니악(ENIAC)은 사람에게 익숙한 10진법을 사용하도록 설계되었으나 전기회로는 전압이 불안정해서 전압을 10단계로 나누어 처리하는 데 한계가 있었다. 그래서 1950년에 개발된 에드박(EDVAC)은 단 두 가지 단계, 전기가 흐르면 1, 흐르지 않으면 0,만으로 동작하도록 설계되었고 매우 성공적이었다.

손가락의 개수가 10개인 사람에게 10진법이 적합하듯, 컴퓨터와 같은 전기회로에는 2진법이 적합한 것이다.

그 이후부터 지금까지 대부분의 컴퓨터는 2진 체계로 설계되었기 때문에, 2진법을 알지 못하면 컴퓨터의 동작원리나 데이터 처리방식을 온전히 이해할 수 없다. 지금까지 변수에 값을 저장하면 10진수로 저장되는 것처럼 설명을 하였지만, 컴퓨터는 2진수(0과 1) 밖에 모르기 때문에 아래의 오른쪽과 같이 2진수로 바뀌어 저장된다. 2진수 11001은 10진수로 25이다.

```
int age = 25; // 변수 age에 25를 저장
```



| 참고 | int타입의 크기가 4 byte이면, 32자리의 2진수로 표현해야하지만 앞의 0은 생략하였다. 0을 생략하지 않으면, '11001'이 아니라 '00000000000000000000000000011001'이다.

이처럼 2진법은 0과 1로만 데이터를 표현하기 때문에 10진법에 비해 많은 자리수를 필요로 한다. 10진수 2와 같이 작은 숫자도 2진수로 표현하려면 2자리가 필요하다. 2진수 한 자리로는 1보다 큰 값을 표현할 수 없기 때문이다.

이것은 10진수에서 9보다 큰 수를 표현하기 위해서는 두 자리의 10진수가 필요한 것과 같다.

2진수	10진수
0	0
1	1
10	2
11	3
100	4
101	5
110	6
111	7
1000	8
1001	9
1010	10

2진수는 2가 없으므로 자리 옮김이 발생해서 10이 된다.

10진수도 표현할 수 있는 제일 큰 수인 9 다음에는 자리 옮김이 발생한다.

그래서 2진수 1에 1을 더하면 2가 아닌 10이 되고, 2진수 11에 1을 더하면 12가 아닌 100이 된다. 10진수와 비교해보면 쉽게 이해가 될 것이다.

2진수	10진수
$ \begin{array}{r} 1 \quad 11 \\ + 1 \quad + 1 \\ \hline 10 \quad 100 \end{array} $	$ \begin{array}{r} 9 \quad 99 \\ + 1 \quad + 1 \\ \hline 10 \quad 100 \end{array} $

자리수가 많아지면 해도 2진수는 10진수를 온전히 표현할 수 있다. 게다가 덧셈이나 뺄셈 같은 연산도 10진수와 동일하다.

3.2 비트(bit)와 바이트(byte)

한 자리의 2진수를 ‘비트(bit, binary digit)’라고 하며, 1 비트는 컴퓨터가 값을 저장할 수 있는 최소단위이다. 그러나 1 비트는 너무 작은 단위이기 때문에 1 비트 8개를 묶어서 ‘바이트(byte)’라는 단위로 정의해서 데이터의 기본 단위로 사용한다.



▲ 그림2-2 비트, 바이트, 워드의 크기 비교

이 외에도 ‘워드(word)’라는 단위가 있는데, ‘워드(word)’는 ‘CPU가 한 번에 처리할 수 있는 데이터의 크기’를 의미한다. 그림2-2에서는 워드의 크기를 4 바이트(32 비트)라고 했지만, 사실 워드의 크기는 CPU의 성능에 따라 달라진다. 예를 들어 32 비트 CPU에서 1 워드는 32 비트(4 바이트)이고, 64 비트 CPU에서는 64 비트(8 바이트)이다.

| 참고 | nibble : 4 bit, 16진수 1자리(2진수 4자리)를 저장할 수 있는 단위

아래의 표는 1~4비트로 표현할 수 있는 값의 개수를 모두 나열한 것으로 1 비트(2진수 1 자리)로 0과 1, 모두 2개(2^1)의 값을, 2비트(2진수 2자리)로는 4개(2^2)의 값을 표현할 수 있다는 것을 알 수 있다.

| 참고 | 0001과 1은 같은 값이지만, 0001이 크기가 4 자리(4 비트)인 데이터라는 것을 강조하기 위해 빈자리를 0으로 채운 것이다.

1 bit(2개)	2 bit(4개)	3 bit(8개)	4 bit(16개)	10진수
0	00	000	0000	0
1	01	001	0001	1
	10	010	0010	2
	11	011	0011	3
		100	0100	4
		101	0101	5
		110	0110	6
		111	0111	7
			1000	8
			1001	9
			1010	10
			1011	11
			1100	12
			1101	13
			1110	14
			1111	15

▲ 표2-10 1~4비트로 표현할 수 있는 값의 개수

이를 일반화하면, n비트로 2^n 개의 값을 표현할 수 있다. 그리고 n비트로 10진수를 표현한다면, 표현가능한 10진수의 범위는 $0 \sim 2^n - 1$ 이 된다. 표 2-9의 맨 오른쪽 표를 보면, 4 비트로 모두 16개(2^4)의 값을 표현할 수 있으며, 4 비트로 10진수를 표현한다면 범위가 ' $0 \sim 15$ ($0 \sim 2^4 - 1$)'라는 것을 직접 확인할 수 있다.

n비트로 표현할 수 있는 10진수

값의 개수 : 2^n

값의 범위 : $0 \sim 2^n - 1$

| 참고 | 10진수 n자리로 표현할 수 있는 값의 범위가 ' $0 \sim 10^n - 1$ '라는 것과 비교해보면 이해가 더 쉬울 것이다. 10진수 2자리로 표현할 수 있는 값의 범위는 ' $0 \sim 10^2 - 1$ ', 즉 ' $0 \sim 99$ '가 된다.

3.3 8진법과 16진법

2진법은 오직 0과 1, 두 개의 기호만으로 값을 표현하기 때문에, 2진법으로 값을 표현하면 자리수가 상당히 길어진다는 단점이 있다. 이러한 단점을 보완하기 위해 2진법 대신 8진법이나 16진법을 사용한다.

8진수는 2진수 3자리를, 16진수는 2진수 4자리를 각각 한자리로 표현할 수 있기 때문에 자리수가 짧아져서 알아보기 쉽고 서로 간의 변환방법 또한 매우 간단하다.

2진법	8진법	10진법	16진법	2진수	8진수	10진수	16진수
0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1
2	2	2	2	10	2	2	2
3	3	3	3	11	3	3	3
4	4	4	4	100	4	4	4
5	5	5	5	101	5	5	5
6	6	6	6	110	6	6	6
7	7	7	7	111	7	7	7
8개	8개	10개	16개	1000	10	8	8
				1001	11	9	9
				1010	12	10	A
				1011	13	11	B
				1100	14	12	C
				1101	15	13	D
				1110	16	14	E
				1111	17	15	F
				10000	20	16	10

▲ 그림2-3 2, 8, 10, 16진법에 사용되는 기호

8진법은 값을 표현하는데 8개의 기호가 필요하므로 0~7의 숫자를 기호로 사용하면 되지만, 16진법은 16개의 기호가 필요하므로 0~9의 숫자만으로는 부족하다. 그래서 6개의 문자(A~F)를 추가로 사용한다. 예를 들어 16진수 A는 10진수로 10이고, F는 15이다.

2진수를 8진수, 16진수로 변환

2진수를 8진수로 변환하려면, 2진수를 뒤에서부터 3자리씩 끊어서 그에 해당하는 8진수로 바꾸면 된다. 8은 2^3 이기 때문에, 8진수 한 자리가 2진수 3자리를 대신할 수 있는 것이다. 2진수를 16진수로 변환하는 방법 역시 이와 비슷한데, 3자리가 아닌 4자리씩 끊어서 바꾼다는 점만 다르다.

| 참고 | 8진수 또는 16진수를 2진수로 변환하려면 위와 반대의 과정을 거치기만 하면 된다.

2진수	8진수
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

1010101100₍₂₎ = 1254₍₈₎ = 2AC₍₁₆₎

2진수	16진수
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	7
1000	8
1001	9
1010	A
1011	B
1100	C
1101	D
1110	E
1111	F

▲ 그림2-4 2진수를 8진수, 16진수로 변환

위의 그림은 2진수를 8진수와 16진수로 변환하는 과정을 보여준다. 2진수 1010101100는 8진수로 ‘1254’이고, 16진수로는 ‘2AC’라는 것을 알 수 있다.

3.4 정수의 진법 변환

10진수를 n진수로 변환

10진수를 다른 진수로 변환하려면, 해당 진수로 나누고 나머지 값을 옆에 적는 것을 더 이상 나눌 수 없을 때까지 반복한 다음 마지막 몫과 나머지를 아래부터 위로 순서대로 적으면 된다. 글로 설명하니까 복합한 것 같지만 사실은 쉽다. 예를 들어 10진수 46을 2진수로 변환하려면, 46을 2로 나누고 그 몫과 나머지를 아래의 그림과 같이 적는다.

$$\begin{array}{r} 2 \mid 46 \\ \hline 23 & \dots 0 \\ \text{몫} & \text{나머지} \end{array}$$

이 작업을 몫이 나누는 값인 2보다 작을 때까지 반복한다.

$$\begin{array}{r} 2 \mid 46 & \text{나머지} \\ 2 \mid 23 & \dots 0 \\ 2 \mid 11 & \dots 1 \\ 2 \mid 5 & \dots 1 \\ 2 \mid 2 & \dots 1 \\ \hline 1 & \dots 0 \end{array}$$

$46_{(10)} \rightarrow 101110_{(2)}$

이제 마지막 몫부터 나머지를 아래서 위로 순서대로 적기만 하면 2진수로 변환한 결과가 된다. 10진수를 8진수 또는 16진수로 변환하려면 2대신 8이나 16으로 나누면 된다. 즉, n 진수로 변환하려면, n으로 반복해서 나누기만 하면 되는 것이다.

$$\begin{array}{r} 8 \mid 816 & \text{나머지} \\ 8 \mid 102 & \dots 0 \\ 8 \mid 12 & \dots 6 \\ \hline 1 & \dots 4 \end{array}$$

$$816_{(10)} \rightarrow 1460_{(8)}$$

$$\begin{array}{r} 16 \mid 1615 & \text{나머지} \\ 16 \mid 100 & \dots 15(F) \\ \hline 6 & \dots 4 \end{array}$$

$$1615_{(10)} \rightarrow 64F_{(16)}$$

10진수	16진수
10	A
11	B
12	C
13	D
14	E
15	F

아래의 그림은 10진수를 10진수로 변환하는 과정을 보여준다. 10진수를 10진수로 변환하는 것이 의미는 없지만, 이 변환방법의 원리를 이해하고 기억하는데 도움이 될 것이다.

$$\begin{array}{r} 10 \mid 12345 & \text{나머지} \\ 10 \mid 1234 & \dots 5 \\ 10 \mid 123 & \dots 4 \\ 10 \mid 12 & \dots 3 \\ \hline 1 & \dots 2 \end{array}$$

$$12345_{(10)} \rightarrow 12345_{(10)}$$

n진수를 10진수로 변환

어떤 진법의 수라도 10진수로 변환하는 방법은 똑같다. 각 자리의 수에 해당 단위의 값을 곱해서 모두 더하면 된다. 예를 들어 10진수 123은 다음과 같이 풀어쓸 수 있다.

$$\begin{aligned} 123_{(10)} &= 100 + 20 + 3 \\ &= 1 \times 100 + 2 \times 10 + 3 \times 1 \\ &= 1 \times 10^2 + 2 \times 10^1 + 3 \times 10^0 \end{aligned}$$

10 ²	10 ¹	10 ⁰
1	2	3
$1 \times 10^2 + 2 \times 10^1 + 3 \times 10^0$		

마찬가지로 2진수는 다음과 같이 표현할 수 있는데, 각 자리의 단위가 10의 제곱이 아니라 2의 제곱이라는 점을 제외하면 10진수와 동일하다.

2 ⁵	2 ⁴	2 ³	2 ²	2 ¹	2 ⁰
1	0	1	1	1	0
$1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 1 \times 2^2 + 1 \times 2^1 + 0 \times 2^0$					

2의 제곱	2 ⁰	2 ¹	2 ²	2 ³	2 ⁴	2 ⁵	2 ⁶	2 ⁷	2 ⁸	2 ⁹	2 ¹⁰
10진수	1	2	4	8	16	32	64	128	256	512	1024

$$\begin{aligned} 101110_{(2)} &= 1 \times 2^5 + 0 \times 2^4 + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 \\ &= 1 \times 32 + 0 \times 16 + 1 \times 8 + 1 \times 4 + 1 \times 2 + 0 \times 1 \\ &= 32 + 8 + 4 + 2 \\ &= 46_{(10)} \end{aligned}$$

8진수와 16진수를 10진수로 변환하는 방법 역시 동일하다.

8 ³	8 ²	8 ¹	8 ⁰
1	4	6	0
$1 \times 8^3 + 4 \times 8^2 + 6 \times 8^1 + 0 \times 8^0$			

8의 제곱	8 ⁰	8 ¹	8 ²	8 ³	8 ⁴
10진수	1	8	64	512	4096

$$\begin{aligned} 1460_{(8)} &= 1 \times 8^3 + 4 \times 8^2 + 6 \times 8^1 + 0 \times 8^0 \\ &= 1 \times 512 + 4 \times 64 + 6 \times 8 + 0 \times 1 \\ &= 512 + 256 + 48 + 0 \\ &= 816_{(10)} \end{aligned}$$

$$\begin{array}{c} 16^2 \quad 16^1 \quad 16^0 \\ \boxed{6} \quad 4 \quad \boxed{F} \\ 6 \times 16^2 + 4 \times 16^1 + F \times 16^0 \end{array}$$

16의 제곱	16^0	16^1	16^2	16^3	16^4
10진수	1	16	256	4096	65536

$$\begin{aligned} 64F_{(16)} &= 6 \times 16^2 + 4 \times 16^1 + F \times 16^0 \\ &= 6 \times 256 + 4 \times 16 + F \times 1 \leftarrow F는 10진수로 15이므로 15 \times 1과 같다. \\ &= 1536 + 64 + 15 \\ &= 1615_{(10)} \end{aligned}$$

3.5 실수의 진법 변환

10진 소수점수를 2진 소수점수로 변환하는 방법

앞서 10진 정수를 2진 정수로 변환할 때, 10진수를 2로 계속 나누면서 나머지를 구했던 것을 기억할 것이다. 10진 소수점수를 2진 소수점수로 변환하는 방법은 이와 반대로 10진 소수점수에 2를 계속 곱한다.

예를 들어 10진수 0.625를 2진수로 변환하는 방법은 다음과 같다.

① 10진 소수에 2를 곱한다.

$$0.625 \times 2 = 1.25$$

② 위의 결과에서 소수부만 가져다가 다시 2를 곱한다.

$$\begin{array}{r} 0.625 \times 2 = 1.25 \\ \downarrow \\ 0.25 \times 2 = 0.5 \end{array}$$

③ ①과 ②의 과정을 소수부가 0이 될 때까지 반복한다.

$$\begin{array}{r} 0.625 \times 2 = 1.25 \\ 0.25 \times 2 = 0.5 \\ 0.5 \times 2 = 1.0 \end{array}$$

| 참고 | ③의 과정에서 소수가 0이 되지 않고 무한히 반복될 수도 있다.

위의 결과에서 정수부만을 위에서 아래로 순서대로 적고 ‘.’을 앞에 붙이면 된다.

$$\begin{array}{r} 0.625 \times 2 = 1.25 \\ 0.25 \times 2 = 0.5 \\ 0.5 \times 2 = 1.0 \\ \downarrow \\ 0.625_{(10)} \rightarrow 0.101_{(2)} \end{array}$$

참고로 10진 소수를 10진 소수로 변환하는 방법은 다음과 같다. 2대신 10을 곱할 뿐이다.
10진 소수를 2진 소수로 변환하는 방법을 기억하는데 도움이 될 것이다.

$$\begin{array}{r}
 0.625 \times 10 = 6.25 \\
 0.25 \times 10 = 2.5 \\
 0.5 \times 10 = 5.0 \\
 0.625_{(10)} \rightarrow 0.625_{(10)}
 \end{array}$$

2진 소수점수를 10진 소수점수로 변환하는 방법

그러면, 이제 2진 소수를 10진 소수로 바꿔서 $0.101_{(2)}$ 가 정말로 $0.625_{(10)}$ 인지 확인해보자.
 $0.625_{(10)}$ 를 다음과 같이 표현할 수 있듯이

	10^{-1}	10^{-2}	10^{-3}
0.	6	2	5

$$0.625_{(10)} = 6 \times 10^{-1} + 2 \times 10^{-2} + 5 \times 10^{-3}$$

$0.101_{(2)}$ 은 다음과 같이 표현할 수 있다.

	2^{-1}	2^{-2}	2^{-3}
0.	1	0	1

$$\begin{aligned}
 0.101_{(2)} &= 1 \times 2^{-1} + 0 \times 2^{-2} + 1 \times 2^{-3} \\
 &= 1 \times 0.5 + 0 \times 0.25 + 1 \times 0.125 \\
 &= 0.5 + 0.125 \\
 &= 0.625_{(10)}
 \end{aligned}$$

위의 계산과정을 통해 $0.101_{(2)}$ 이 $0.625_{(10)}$ 라는 것을 확인할 수 있다.

| 참고 | 123.456처럼 정수부가 있는 소수점수는 정수부 123과 소수점부 0.456을 따로 변환한 다음에 더하면 된다.

3.6 음수의 2진 표현 – 2의 보수법

앞서 살펴본 것과 같이 n 비트의 2진수로 표현할 수 있는 값의 개수는 모두 2^n 개이므로, 4비트의 2진수로는 모두 $2^4 (=16)$ 개의 값을 표현할 수 있다. 이 값을 모두 ‘부호없는 정수(0과 양수)’의 표현에 사용하면, 아래와 같이 ‘0부터 15까지의 정수’를 나타낼 수 있다.

#	2진수	부호없는 10진수
1	0000	최소값 → 0
2	0001	1
3	0010	2
4	0011	3
5	0100	4
6	0101	5
7	0110	6
8	0111	7
9	1000	8
10	1001	9
11	1010	10
12	1011	11
13	1100	12
14	1101	13
15	1110	14
16	1111	최대값 → 15

▲ 표2-11 4 비트로 표현할 수 있는 부호없는 10진 정수

그러면 4비트의 2진수로 부호있는 정수, 즉 양수와 음수를 모두 표현하려면 어떻게 해야 할까? 4비트 2진수의 절반인 8개는 0으로 시작하고, 나머지 절반은 1로 시작하니까, 1로 시작하는 2진수를 음수표현에 사용하자. 이렇게 하면, ‘왼쪽의 첫 번째 비트(MSB)’가 0이면 양수, 1이면 음수이므로 첫 번째 비트만으로 값의 부호를 알 수 있게 된다.

| 참고 | 2진수의 제일 왼쪽의 1 bit를 MSB(most significant bit)라고 한다.

#	2진수	부호있는 10진수
1	0000	0
2	0001	1
3	0010	2
4	0011	3
5	0100	4
6	0101	5
7	0110	6
8	0111	7
9	1000	???
10	1001	???
11	1010	???
12	1011	???
13	1100	???
14	1101	???
15	1110	???
16	1111	???

이제 위 표의 절반을 어떻게 음수로 채워야 할까? 일단 양수처럼 0부터 순차적으로 채워보자.

#	2진수	부호있는 10진수
1	0000	0
2	0001	1
3	0010	2
4	0011	3
5	0100	4
6	0101	5
7	0110	6
8	0111	최대값 → 7
9	1000	-0
10	1001	-1
11	1010	-2
12	1011	-3
13	1100	-4
14	1101	-5
15	1110	-6
16	1111	최소값 → -7

$$\begin{array}{r}
 \begin{array}{|c|c|c|c|} \hline
 0 & 1 & 0 & 1 \\ \hline
 \end{array} \leftarrow 10진수로 5 \\
 +) \quad \begin{array}{|c|c|c|c|} \hline
 1 & 1 & 0 & 1 \\ \hline
 \end{array} \leftarrow 10진수로 -5 \\
 \hline
 \end{array}$$

1 0 0 1 0 ← 자리 올림이 발생했으나
크기가 4비트라서 버려짐

음수를 이렇게 배치하면, 양수의 첫 번째 비트만 1로 바꾸면 음수가 된다는 장점이 있다. 그러나, 두 수를 더했을 때 2진수로 0이 되지 않는다는 것과 0이 두개(0, -0) 존재한다는 단점이 있다. 게다가 2진수가 증가할 때 10진 음수는 감소한다.

#	2진수	부호있는 10진수
1	0000	0
2	0001	1
3	0010	2
4	0011	3
5	0100	4
6	0101	5
7	0110	6
8	0111	최대값 → 7
9	1000	최소값 → -8
10	1001	-7
11	1010	-6
12	1011	-5
13	1100	-4
14	1101	-3
15	1110	-2
16	1111	-1

$$\begin{array}{r}
 \begin{array}{|c|c|c|c|} \hline
 0 & 1 & 0 & 1 \\ \hline
 \end{array} \leftarrow 10진수로 5 \\
 +) \quad \begin{array}{|c|c|c|c|} \hline
 1 & 0 & 1 & 1 \\ \hline
 \end{array} \leftarrow 10진수로 -5 \\
 \hline
 \end{array}$$

1 0 0 0 0 ← 자리 올림이 발생했으나
크기가 4비트라서 버려짐

• 2의 보수법에 의한 음수배치

그러나 위와 같이 ‘2의 보수법’에 의해 음수를 배치하면, 절대값이 같은 양수와 음수를 더했을 때 2진수로도 0을 결과로 얻으므로 부호를 신경쓰지 않고 덧셈할 수 있게 된다.

그리고 2진수가 증가할 때 10진 음수가 감소한다는 모순도 없어졌다. 다만, 첫 번째 비트를 바꾸는 것만으로 값의 부호를 바꿀 수 없게 되었다.

2의 보수법

어떤 수의 ‘ n 의 보수’는 더했을 때 n 이 되는 수를 말한다. 7의 ‘10의 보수’는 3이고, 3의 ‘10의 보수’는 7이다. 3과 7은 ‘10의 보수의 관계’에 있다고 한다. ‘2의 보수 관계’ 역시, 더해서 2가 되는 두 수의 관계를 말하며 10진수 2는 2진수로 ‘10’이다. 2진수로 ‘10’은 자리올림이 발생하고 0이 되는 수를 뜻한다. 그래서 ‘2의 보수 관계’에 있는 두 2진수를 더하면 ‘(자리올림이 발생하고) 0이 된다.’

0	1	0	1	← 10진수로 5
+	1	0	1	1
1	0	0	0	← 자리 올림이 발생했으나 크기가 4비트라서 버려짐

▲ 그림2-5 2의 보수 관계에 있는 두 2진수의 덧셈

위의 그림에서 알 수 있듯이 2진수 '0101'와 '1011'은 서로 '2의 보수 관계'에 있으며, 이 두 2진수를 더하면 0이 된다. 이 덧셈이 10진수로도 0이 되려면, 2진수 '0101'가 10진수로 5니까, 2진수 '1011'은 10진수로 -5이어야 한다.

2진수	부호있는 10진수
0000	0
0001	1
0010	2
0011	3
0100	4
0101	5
0110	6
0111	최대값 → 7
1000	최소값 → -8
1001	-7
1010	-6
1011	-5
1100	-4
1101	-3
1110	-2
1111	-1

2의 보수 관계

부호가 다르고
절대값이 같은 수

▲ 표2-12 2의 보수법으로 표현한 10진수

이처럼 서로 ‘2의 보수 관계’에 있는 두 2진수로 5와 -5처럼 절대값이 같고 부호가 다른 두 10진수를 표현하는 것을 ‘2의 보수법’이라고 하며, 현재 대부분의 시스템이 ‘2의 보수법’으로 부호있는 정수를 표현한다.

음수를 2진수로 표현하기

10진 음의 정수를 2진수로 변환하려면, 먼저 10진 음의 정수의 절대값을 2진수로 변환한다. 그 다음에 이 2진수의 ‘2의 보수’를 구하면 된다. 예를 들어 ‘-5’의 2진 표현을 구하는 과정은 다음과 같다.

$$\begin{array}{r} -5_{(10)} \\ \xrightarrow{\text{① 절대값}} 5_{(10)} \\ \xrightarrow{\text{② 2진수}} 0101_{(2)} \\ \xrightarrow{\text{③ 2의 보수}} 1011_{(2)} \end{array}$$

▲ 그림2-6 10진 음의 정수의 2진 표현을 구하는 과정

위의 방법은 부호가 다르고 절대값이 같은 두 정수의 2진 표현이 서로 ‘2의 보수’관계에 있다는 것을 이용한 것으로 복잡해 보이지만 간단하다. 절대값은 부호만 빼어내면 되고, 10진수를 2진수로 변환하는 방법은 이미 배웠고, ‘2의 보수’로 변환하는 방법도 쉽다.

2의 보수 구하기

서로 ‘2의 보수’의 관계에 있는 두 수를 더하면 ‘0(자리올림 발생)’이 된다. 예를 들어 2진수 ‘0101’의 ‘2의 보수’를 구하려면, ‘0101’에 어떤 수를 더하면 0이 되는지 알아내야 한다.

$$\begin{array}{c|c|c|c|c} 0 & 1 & 0 & 1 & \\ \hline + & ? & ? & ? & ? \\ \hline & 1 & 0 & 0 & 0 & 0 \end{array} \quad \text{2의 보수}$$

아래와 같이 뺄셈으로 ‘2의 보수’를 간단히 구할 수 있지만 자리수가 많아지면 뺄셈도 쉽지 않다.

$$\begin{array}{c|c|c|c|c} 1 & 0 & 0 & 0 & 0 \\ \hline - & 0 & 1 & 0 & 1 \\ \hline & 1 & 0 & 1 & 1 \end{array} \quad \text{2의 보수}$$

다행히 뺄셈보다 ‘2의 보수’를 더 간단히 구하는 방법이 있다. ‘1의 보수’를 구한 다음 1을 더한다. 그러면, 2의 보수를 구할 수 있다.

$$\boxed{2\text{의 보수} = 1\text{의 보수} + 1}$$

‘1의 보수’는 0을 1로, 1을 0으로만 바꾸면 되므로 구하기 쉽다. 예를 들어, 2진수 ‘0101’의 ‘1의 보수’는 ‘1010’이다. 여기에 1을 더하기만 하면 2의 보수가 된다.

$$\begin{array}{c} 0 & 1 & 0 & 1 \\ \downarrow & \downarrow & \downarrow & \downarrow \\ 1 & 0 & 1 & 0 & \leftarrow '0101'의 '1의 보수' \\ +) & & & 1 \\ \hline 1 & 0 & 1 & 1 & \leftarrow '0101'의 '2의 보수' \end{array}$$

4. 기본형(primitive type)

이번 단원에서는 기본형의 보다 세부적인 내용에 대해 살펴볼 것이다. 다소 깊이 있는 내용이므로 어렵다고 느낄 수도 있는데, 앞서 배운 기본형에 대한 대략적인 내용만으로도 별 부족함 없이 진도를 나갈 수 있으니까 다 이해하지 못해도 괜찮다. 그래도 언젠가는 반드시 알아야하는 내용이므로 가볍게라도 봐둘 필요는 있다.

4.1 논리형 – boolean

논리형에는 ‘boolean’ 한 가지 밖에 없다. boolean형 변수에는 true와 false 중 하나를 저장할 수 있으며 기본값(default)은 false이다.

boolean형 변수는 대답(yes/no), 스위치(on/off) 등의 논리구현에 주로 사용된다. 그리고 boolean형은 true와 false, 두 가지의 값만을 표현하면 되므로 1 bit만으로도 충분하지만, 자바에서는 데이터를 다루는 최소단위가 byte이기 때문에, boolean의 크기가 1 byte이다. 아래 문장은 power라는 boolean형 변수를 선언하고 true로 변수를 초기화 했다.

```
boolean power = true;  
boolean checked = False; // 예러. 대소문자가 구분됨. true 또는 false만 가능
```

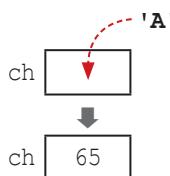
자바에서는 대소문자가 구별되기 때문에 TRUE와 true는 다른 것으로 간주된다는 것에 주의하자.

4.2 문자형 – char

문자형 역시 ‘char’ 한 가지 자료형밖에 없다. 문자를 저장하기 위한 변수를 선언할 때 사용되며, char타입의 변수는 단 하나의 문자만을 저장할 수 있다. 아래의 문장은 char타입의 변수 ch를 선언하고, 문자 ‘A’로 초기화한다.

```
char ch = 'A'; // 문자 'A'를 char타입의 변수 ch에 저장.
```

위의 문장은 변수에 ‘문자’가 저장되는 것 같지만, 사실은 문자가 아닌 ‘문자의 유니코드(정수)’가 저장된다. 컴퓨터는 숫자밖에 모르기 때문에 모든 데이터를 숫자로 변환하여 저장하는 것이다. 문자'A'의 유니코드는 65이므로, 변수 ch에는 65가 저장된다.



그래서 문자 리터럴 대신 문자의 유니코드를 직접 저장할 수도 있다. 문자 'A'의 유니코드는 10진수로 65이며, 아래의 두 문장은 동일한 결과를 얻는다.

```
char ch = 'A'; // OK. 문자 'A'를 char타입의 변수 ch에 저장.  
char ch = 65; // OK. 문자의 코드를 직접 변수 ch에 저장
```

만일 어떤 문자의 유니코드를 알고 싶으면, char형 변수에 저장된 값을 정수형(int)으로 변환하면 된다.

```
int code = (int) ch; // ch에 저장된 값을 int타입으로 변환하여 저장한다.
```

어떤 타입(type, 형)을 다른 타입으로 변환하는 것을 형변환(캐스팅, casting)이라고 하는데, 형변환에 대해서는 이 장의 마지막에 자세히 설명할 것이다. 지금은 문자의 유니코드를 알아내는 방법과, 어떤 유니코드가 어떤 문자를 나타내는가를 알아내는 방법이 있다는 것만 이해하고 넘어가자.

▼ 예제 2-7/CharToCode.java

```
class CharToCode {  
    public static void main(String[] args) {  
        char ch = 'A'; // char ch = 65;  
        int code = (int)ch; // ch에 저장된 값을 int타입으로 변환하여 저장한다.  
  
        System.out.printf("%c=%d(%#X)%n", ch, code, code);  
  
        char hch = '가'; // char hch = 0xAC00;  
        System.out.printf("%c=%d(%#X)%n", hch, (int)hch, (int)hch);  
    }  
}
```

▼ 실행결과

```
A=65 (0X41)  
가=44032 (0XAC00)
```

실행결과를 보면, 문자 'A'의 유니코드는 65(16진수로 0x41)이고, 문자 '가'의 유니코드는 44032(16진수로 0xAC00)이라는 것을 알 수 있다. 유니코드를 알면 아래와 같이 char형 변수에 문자를 저장할 때, 문자 리터럴 대신에 유니코드를 직접 사용할 수도 있다.

```
char hch = 0xAC00; // char hch = '가';  
char hch = '\uAC00'; // 이렇게도 가능
```

특수 문자 다루기

영문자 이외에 tab이나 backspace 등의 특수문자를 저장하려면, 아래와 같이 조금 특별한 방법을 사용한다.

```
char tab = '\t'; // 변수 tab에 탭 문자를 저장
```

'\t'는 실제로는 두 문자로 이루어져 있지만 하나의 문자(탭, tab)를 의미한다. 아래의 표는 탭(tab)과 같이 특수한 문자를 어떻게 표현할 수 있는지 알려준다.

특수 문자	문자 리터럴
tab	\t
backspace	\b
form feed	\f
new line	\n
carriage return	\r
역슬래쉬(\)	\\
작은따옴표	\'
큰따옴표	\"
유니코드(16진수)문자	\u유니코드 (예: char a='\u0041')

▲ 표 2-13 특수 문자를 표현하는 방법

▼ 예제 2-8/SpecialCharEx.java

```
class SpecialCharEx {  
    public static void main(String[] args) {  
        System.out.println('\''); // ''처럼 할 수 없다.  
        System.out.println("abc\t123\b456"); // \b에 의해 3이 지워진다.  
        System.out.println('\n'); // 개행(new line) 문자 출력하고 개행  
        System.out.println("\\"Hello\\\""); // 큰따옴표를 출력하려면 이렇게 한다.  
        System.out.println("c:\\\"");  
    }  
}
```

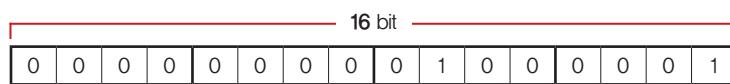
▼ 실행결과

```
'  
abc      12456  
  
"Hello"  
c:\
```

| 참고 | 한글 윈도우에서는 역슬래쉬(back slash)가 '\\' 대신 '₩'로 표시된다.

char의 저장형식

char타입의 크기는 2 byte(=16 bit)이므로, 16자리의 2진수로 표현할 수 있는 정수의 개수인 65536개($=2^{16}$)의 코드를 사용할 수 있으며, char형 변수는 이 범위 내의 코드 중 하나를 저장할 수 있다. 예를 들어, 문자 'A'를 저장하면, 아래와 같이 2진수 '0000000001000001'(10진수로 65)로 저장된다.

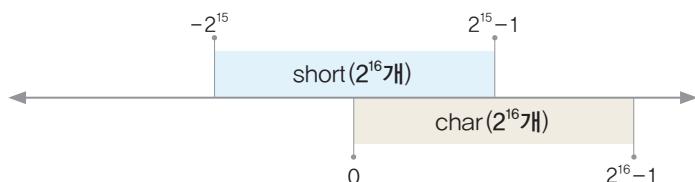


▲ 그림 2-7 char의 저장형식(문자 'A')

char타입은 문자를 저장할 변수를 선언하기 위한 것이지만, 실제로 char타입의 변수에 문자가 아닌 ‘문자의 유니코드(정수)’가 저장되고 저장형식 역시 정수형과 동일하다. 다만, 정수형과 달리 음수가 필요없으므로 저장할 수 있는 값의 범위가 다르다.

2 byte(=16 bit)로는 모두 2^{16} (=65536)개의 값을 표현할 수 있는데, char타입에 저장되는 값인 유니코드는 모두 양수(0 포함)이므로, ‘0~65535’의 범위를 가지며, 정수형인 ‘short’은 절반을 음수에 사용하므로 ‘-32768~32767’을 범위로 갖는다.

16비트로 표현할 수 있는 정수의 개수 : 2^{16} 개 (65536개)
 short타입의 범위 : $-2^{15} \sim 2^{15}-1$ (-32768~32767)
 char타입의 범위 : 0 ~ $2^{16}-1$ (0 ~ 65535)



다음과 같이 변수 ch와 s에 ‘A’와 65를 저장하면, 둘 다 2진수로 똑같은 값이 저장된다. 컴퓨터는 모든 값을 0과 1로 바꾸어 저장하기 때문이다.

```
char ch = 'A';      // char ch = 65;
short s = 65;
```

자료형	2진수	10진수
char	000000000000100001	65
short	000000000000100001	65

▲ 표2-14 char타입과 short타입의 값 비교

그런데도 두 변수의 값을 출력해보면 결과가 다르다. `println()`은 변수의 타입이 정수형이면 변수에 저장된 값을 10진수로 해석하여 출력하고, 문자형이면 저장된 숫자에 해당하는 유니코드 문자를 출력하기 때문이다.

```
System.out.println(ch);      // A가 출력된다.
System.out.println(s);      // 65가 출력된다.
```

이처럼 값은 어떻게 해석하느냐에 따라 결과가 달라지므로 값만으로는 값을 해석할 수 없다. 값의 타입까지 알아야 올바르게 해석할 수 있는 것이다. 예를 들어 ‘1231’이라는 값이 있을 때, 이 값의 타입을 모르면, 이 값을 ‘천이백삼십일’로 해석해야 할지, 아니면 12월 31일이나 12시 31분으로 해석해야 할지 알 수 없다.

인코딩과 디코딩(encoding & decoding)

컴퓨터가 숫자밖에 모르기 때문에 문자가 숫자로 변환되어 저장된다는 것은 알겠는데, 그러면 도대체 어떤 기준에 의한 것일까? 바로 아래의 오른쪽에 있는 표에 의한 것인데, 이 코드표는 ‘유니코드(unicode)’이다.

문자	유니 코드
...	...
A	65
B	66
C	67
...	...

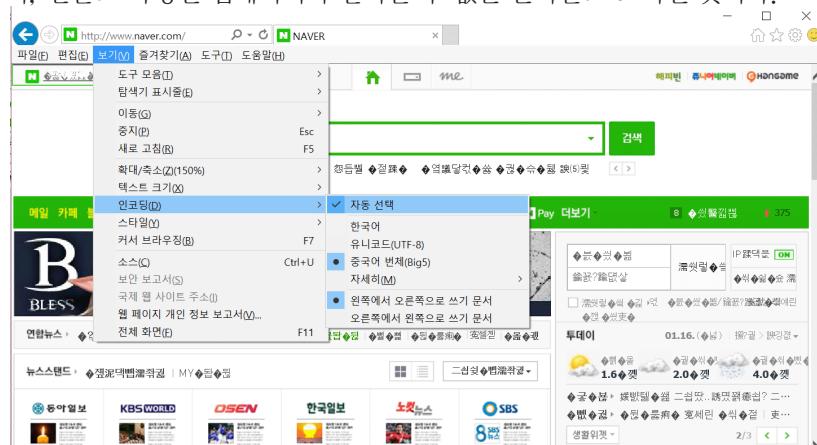
▲ 그림2-8 유니코드(unicode)를 이용한 인코딩과 디코딩

위의 그림에서 오른쪽 표를 보면, 문자 ‘A’의 유니코드가 65인 것을 알 수 있다. 그래서 문자 ‘A’를 유니코드로 인코딩하면 65가 되는 것이다. 반대로 65를 유니코드로 디코딩 하면 문자 ‘A’가 된다. 이처럼 문자를 코드로 변환하는 것을 ‘문자 인코딩(encoding)’, 그 반대로 코드를 문자로 변환하는 것을 ‘문자 디코딩(decoding)’이라고 하며, 문자를 저장할 때는 인코딩을 해서 숫자로 변환해서 저장하고, 저장된 문자를 읽어올 때는 디코딩을 해서 숫자를 원래의 문자로 되돌려야 한다.

| 참고 | ‘encode’는 ‘~을 코드화하다.’ 또는 ‘~을 암호화하다.’라는 뜻이다.

당연한 얘기지만 어떻게 인코딩을 했는지를 알아야 디코딩이 가능하다. 만일 인코딩에 사용된 코드표와 디코딩에 사용된 코드표가 다르면 엉뚱한 글자들로 바뀌어 나타날 것이다. 웹서핑을 하다가 페이지 전체가 알아볼 수 없는 이상한 글자들로 가득 찬 경험이 적어도 한두 번쯤은 있을 텐데, 그 이유는 해당 html문서의 인코딩에 사용된 코드표와 웹브라우저의 설정이 맞지 않아서이다.

대부분의 경우 웹페이지(html파일)에 인코딩 정보가 포함되어 있어서 웹브라우저가 올바르게 디코딩하지만, 웹브라우저의 인코딩 설정이 웹페이지의 인코딩과 다른 경우 글자가 알아볼 수 없게 깨져서 나타난다. 아래의 그림은 웹브라우저의 인코딩을 ‘중국어 번체’로 지정하여, 한글로 작성된 웹페이지가 알아볼 수 없는 문자들로 표시된 것이다.



▲ 그림2-9 중국어 인코딩으로 읽어서 깨진 한글 웹페이지

아스키(ASCII)

‘ASCII’는 ‘American Standard Code for Information Interchange’의 약어로 정보교환을 위한 미국 표준 코드란 뜻이다. 아스키는 128개($=2^7$)의 문자 집합(character set)을 제공하는 7 bit부호로, 처음 32개의 문자는 인쇄와 전송 제어용으로 사용되는 ‘제어문자(control character)’로 출력할 수 없고 마지막 문자(DEL)를 제외한 33번째 이후의 문자들은 출력할 수 있는 문자들로, 기호와 숫자, 영대소문자로 이루어져 있다.

아스키는 숫자 ‘0~9’, 영문자 ‘A~Z’와 ‘a~z’가 연속적으로 배치되어 있다는 특징이 있으며, 이러한 특징은 프로그래밍에서 유용하게 활용된다.

확장 아스키(Extended ASCII)와 한글

일반적으로 데이터는 byte 단위로 다뤄지는데 아스키는 7 bit이므로 1 bit가 남는다. 이 남는 공간을 활용해서 문자를 추가로 정의한 것이 ‘확장 아스키’이다. 확장 아스키에 추가된 128개의 문자는 여러 국가와 기업에서 서로의 필요에 따라 다르게 정의해서 사용한다.

‘ISO(국제표준화기구)’에서 확장 아스키의 표준을 몇 가지 발표했는데, 그 중에서 대표적인 것이 ‘ISO 8859-1’이다. 이 확장 아스키 버전은 ‘ISO Latin 1’이라고도 하는데 서유럽에서 일반적으로 사용하는 문자들을 포함하고 있다.

확장 아스키로도 표현할 수 있는 문자의 개수가 255개뿐이므로 한글을 표현하기에는 턱 없이 부족하다. 그래서 생각해낸 것이 두 개의 문자코드로 한글을 표현하는 방법이었다.

한글을 표현하는 방법은 조합형과 완성형이 있는데, 조합형은 초성, 중성, 종성을 조합하는 방식이고, 완성형은 확장 아스키의 일부 영역(162~254)에 해당하는 두 문자코드를 조합하여 한글을 표현한다. 현재 조합형은 사용되지 않고 ‘완성형(KSC 5601)’에 없는 잘 안 쓰이는 8822글자를 추가한 ‘확장 완성형(CP 949)’이 사용되는데, 이것이 바로 한글 윈도우에서 사용하는 문자 인코딩이다. 한글 윈도우에서 작성된 문서는 기본적으로 ‘CP 949(확장 완성형)’로 인코딩되어 저장된다.

코드 페이지(code page, cp)

IBM이 자사의 PC에 ‘확장 아스키’를 도입해서 사용하기 시작할 때, PC를 사용하는 지역이나 국가에 따라 여러 버전의 ‘확장 아스키’가 필요했다. IBM은 이들을 ‘코드 페이지(code page)’라 하고, 각 코드 페이지에 ‘CP xxx’와 같은 형식으로 이름을 붙였다. IBM은 MS와 같은 업체들과 협력하여 ‘코드 페이지’를 만들어내고 공유했다. 한글 윈도우는 ‘CP 949’를, 영문 윈도우는 ‘CP 437’을 사용한다.

| 참고 | 코드 페이지는 확장 아스키의 256개 문자를 어떤 숫자로 변환할 것인지를 적어놓은 ‘문자 코드표(code page)’이다.

유니코드(Unicode)

예전엔 같은 지역 내에서만 문서교환이 주를 이뤘지만, 인터넷이 발명되면서 서로 다른 지역의 다른 언어를 사용하는 컴퓨터간의 문서교환이 활발해지기 시작하자 서로 다른 문자 인코딩을 사용하는 컴퓨터간의 문서 교환에 어려움을 겪게 되었다.

이러한 어려움을 해소하고자 전 세계의 모든 문자를 하나의 통일된 문자 집합으로 표현하고자 노력하였고 그 결과가 바로 ‘유니코드’이다.

유니코드는 처음에 모든 문자를 2 byte($=2^{16}=65536$)로 표현하려했으나, 2 byte($=16$ bit)로도 부족해서 20 bit(약 100만 문자)로 확장되었다. 새로 추가된 문자들을 보충 문자(supplementary character)라고 하는데 이 문자들을 표현하기 위해서는 char타입이 아닌 int타입을 사용해야 한다. 우리가 보충문자를 쓸 일은 거의 없기 때문에 참고로만 알아두면 된다.

유니코드는 먼저 유니코드에 포함시키고자 하는 문자들의 집합을 정의하였는데, 이것을 유니코드 문자 셋(또는 캐릭터 셋, character set)이라고 한다. 그리고 이 문자 셋에 번호를 붙인 것이 유니코드 인코딩이다. 유니코드 인코딩에는 UTF-8, UTF-16, UTF-32 등 여러 가지 종류가 있는데 자바에서는 UTF-16을 사용해오다가 JDK 18부터 UTF-8로 바뀌었다. UTF-16은 모든 문자를 2 byte의 고정크기로 표현하고 UTF-8은 하나의 문자를 1~4 byte의 가변크기로 표현한다. 그리고 두 인코딩 모두 처음 128문자가 아스키와 동일하다. 아스키를 그대로 포함하고 있는 것이다.

| 참고 | 코드 포인트(code point)는 유니코드 문자 셋에 순서대로 붙인 일련번호이다. 유니코드에는 뭐라고 읽는지도 알 수 없는 문자들이 많이 포함되어 있으므로, 이 문자들은 번호(코드 포인트)를 붙여서 다루는 것이 편리하기 때문이다.

code point	유니코드 문자 셋	ASCII	UTF-8	UTF-16
...
U+0061	a	0x61	0x61	0x0061
U+0062	b	0x62	0x62	0x0062
...
U+AC00	가	–	0xEAB080	0xAC00
U+AC01	각	–	0xEAB081	0xAC01
...	...	–	...	

▲ 표 2-15 유니코드 인코딩 UTF-8과 UTF-16의 비교

모든 문자의 크기가 동일한 UTF-16이 문자를 다루기는 편리하지만, 1 byte로 표현할 수 있는 영어와 숫자가 2 byte로 표현되므로 문서의 크기가 커진다는 단점이 있다. UTF-8에서 영문과 숫자는 1 byte 그리고 한글은 3 byte로 표현되기 때문에 문서의 크기가 작지만 문자의 크기가 가변적이므로 다루기 어렵다는 단점이 있다. 인터넷에서는 전송 속도가 중요하므로, 문서의 크기가 작을수록 유리하다. 그래서 UTF-8인코딩으로 작성된 웹문서의 수가 빠르게 늘고 있다.

예전에는 한글이 2 byte인코딩을 주로 사용 해오다가 한글이 3 byte인 UTF-8인코딩을 사용하게 되면서 텍스트 파일의 크기가 커져서 불리해졌다.

4.3 정수형 – byte, short, int, long

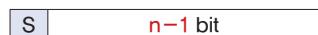
정수형에는 모두 4개의 자료형이 있으며, 각 자료형이 저장할 수 있는 값의 범위가 서로 다르다. 크기순으로 나열하면 다음과 같다. 단위는 byte이다.

byte	<	short	<	int	<	long
1		2		4		8

byte부터 long까지 1 byte부터 시작해서 2배씩 크기가 증가한다는 것을 알 수 있다. 이 중에서도 기본 자료형(default data type)은 int이다.

정수형의 저장형식과 범위

어떤 진법의 리터럴을 변수에 저장해도 실제로는 2진수로 바뀌어 저장된다. 이 2진수가 저장되는 형식은 크게 정수형과 실수형이 있으며, 정수형은 다음과 같은 형식으로 저장된다.



S : 부호 비트(양수는 0, 음수는 1)

n : 타입의 크기(단위:bit)

▲ 그림 2-10 정수형의 저장형식

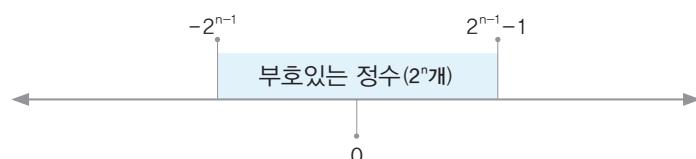
모든 정수형은 부호있는 정수이므로 왼쪽의 첫 번째 비트를 ‘부호 비트(sign bit)’로 사용하고, 나머지는 값을 표현하는데 사용한다. 그래서 n비트로 표현할 수 있는 값의 개수인 2^n 개에서, 절반인 ‘0’으로 시작하는 2^{n-1} 개의 값을 양수(0도 포함)의 표현에 사용하고, 나머지 절반인 ‘1’으로 시작하는 2^{n-1} 개의 값은 음수의 표현에 사용한다.

정수형의 저장형식(n bit)	종류	값의 개수
0 n-1 bit	0, 양수	2^{n-1} 개
1 n-1 bit	음수	2^{n-1} 개

그래서 정수형은 타입의 크기만 알면, 최대값과 최소값을 쉽게 계산해낼 수 있다.

n비트로 표현할 수 있는 정수의 개수 : 2^n 개 ($= 2^{n-1}$ 개 + 2^{n-1} 개)

n비트로 표현할 수 있는 부호있는 정수의 범위 : $-2^{n-1} \sim 2^{n-1}-1$

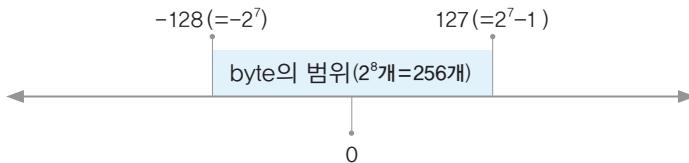


위의 범위의 최대값에서 1을 빼는 이유는 범위에 0이 포함되기 때문이다. 예를 들어 byte의 경우 크기가 1 byte(=8 bit)이므로, byte타입의 변수에 저장할 수 있는 값의 범위는 ‘-128~127’이다.

byte의 저장형식(8 비트)	종류	값의 개수
0 0 0 0 0 0 0 0 ~ 0 1 1 1 1 1 1 1	0.양수	$2^7(0, 1\sim 127)$
1 0 0 0 0 0 0 0 ~ 1 1 1 1 1 1 1 1	음수	$2^7(-128\sim -1)$

8비트로 표현할 수 있는 정수의 개수 : 2^8 개($= 2^7$ 개 + 2^7 개)

8비트로 표현할 수 있는 부호있는 정수의 범위 : $-2^7 \sim 2^7 - 1$ (-128 ~ 127)



이처럼 타입의 크기만 알면, 모든 정수형의 표현 범위를 쉽게 계산해낼 수 있다.

타입	저장 가능한 값의 범위	크기	
		bit	byte
byte	$-128 \sim 127 (-2^7 \sim 2^7 - 1)$	8	1
short	$-32,768 \sim 32,767 (-2^{15} \sim 2^{15} - 1)$	16	2
int	$-2,147,483,648 \sim 2,147,483,647 (-2^{31} \sim 2^{31} - 1, 약 \pm 20억)$	32	4
long	$-9,223,372,036,854,775,808 \sim 9,223,372,036,854,775,807 (-2^{63} \sim 2^{63} - 1)$	64	8

▲ 표 2-16 정수형의 표현범위

정수형의 선택기준

변수에 저장하려는 정수값의 범위에 따라 4개의 정수형 중에서 하나를 선택하면 되겠지만, byte나 short보다 int를 사용하도록 하자. byte와 short이 int보다 크기가 작아서 메모리를 조금 더 절약할 수는 있지만, 저장할 수 있는 값의 범위가 작은 편이라서 연산 시에 범위를 넘어서 잘못된 결과를 얻기가 쉽다.

그리고 JVM의 피연산자 스택(operand stack)이 피연산자를 4 byte 단위로 저장하기 때문에 크기가 4 byte보다 작은 자료형(byte, short)의 값을 계산할 때는 4 byte로 변환하여 연산이 수행된다. 그래서 오히려 int를 사용하는 것이 더 효율적이다.

int 타입의 크기는 4 byte(=32 bit)이므로, 표현할 수 있는 정수의 개수는 ' $2^{32} \div 4 \times 10^9$, 약 40억'이며, 표현가능한 정수의 범위는 ' $-2^{31} \sim 2^{31} - 1 \approx \pm 20억$ '이다.

$$2^{10} = 1024 \approx 10^3 \text{이므로}, 2^{32} = 2^{10} \times 2^{10} \times 2^{10} \times 2^2 = 1024 \times 1024 \times 1024 \times 4 \approx 4 \times 10^9$$

결론적으로 정수형 변수를 선언할 때는 int 타입으로 하고, int의 범위(약 ±20억)를 넘어서는 수를 다뤄야 할 때는 long을 사용하면 된다. 그리고 byte나 short은 성능보다 저장공간을 절약하는 것이 더 중요할 때 사용하자.

| 참고 | long 타입의 범위를 벗어나는 값을 다룰 때는, 실수형 타입이나 BigInteger 클래스(p.548)를 사용하면 된다.

정수형의 오버플로우

만일 4 bit 2진수의 최대값인 ‘1111’에 1을 더하면 어떤 결과를 얻을까? 4 bit의 범위를 넘어서는 값이 되기 때문에 에러가 발생할까?

$$\begin{array}{r}
 \boxed{1} \quad \boxed{1} \quad \boxed{1} \quad \boxed{1} \\
 +) \quad \boxed{0} \quad \boxed{0} \quad \boxed{0} \quad \boxed{1} \\
 \hline
 \boxed{?} \quad \boxed{?} \quad \boxed{?} \quad \boxed{?}
 \end{array}$$

원래 2진수 ‘1111’에 1을 더하면 ‘10000’이 되지만, 4 bit로는 4자리의 2진수만 저장할 수 있기 때문에 ‘0000’이 된다. 즉, 5자리의 2진수 ‘10000’중에서 하위 4 bit만 저장하게 되는 것이다. 이처럼 연산과정에서 해당 타입이 표현할 수 있는 값의 범위를 넘어서는 것을 오버플로우(overflow)라고 한다. 오버플로우가 발생했다고 해서 에러가 발생하는 것은 아니다. 다만 예상했던 결과를 얻지 못할 뿐이다. 애초부터 오버플로우가 발생하지 않게 충분한 크기의 타입을 선택해서 사용해야 한다.

10진수	2진수
$ \begin{array}{r} \boxed{9} \quad \boxed{9} \quad \boxed{9} \quad \boxed{9} \\ +) \qquad \qquad \qquad 1 \\ \hline \boxed{1} \quad \boxed{0} \quad \boxed{0} \quad \boxed{0} \end{array} $	$ \begin{array}{r} \boxed{1} \quad \boxed{1} \quad \boxed{1} \quad \boxed{1} \\ +) \qquad \qquad \qquad 1 \\ \hline \boxed{1} \quad \boxed{0} \quad \boxed{0} \quad \boxed{0} \end{array} $
	← 저장할 공간이 없어서 1은 버려짐 →

▲ 그림 2-11 4자리의 10진수와 2진수의 오버플로우

오버플로우는 ‘자동차 주행표시기(odometer)’나, ‘계수기(counter)’ 등 우리의 일상생활에서도 발견할 수 있는데, 네 자리 계수기라면 ‘0000’부터 ‘9999’까지 밖에 표현하지 못하므로 최대값인 ‘9999’ 다음의 숫자는 ‘0000’이 될 것이다. 원래는 10000이 되어야하는데 다섯 자리는 표현할 수 없어서 맨 앞의 1은 버려지기 때문이다.

그러면 이번엔 반대로 최소값인 ‘0000’에서 1을 감소시키면 어떤 결과를 얻을까? 0에서 1을 뺄 수 없으므로 ‘0000’ 앞에 저장되지 않은 1이 있다고 가정하고 뺄셈을 한다. 결과는 아래와 같이 네 자리로 표현할 수 있는 최대값이 된다.

10진수	2진수
$ \begin{array}{r} \boxed{1} \quad \boxed{0} \quad \boxed{0} \quad \boxed{0} \\ -) \qquad \qquad \qquad 1 \\ \hline \boxed{9} \quad \boxed{9} \quad \boxed{9} \quad \boxed{9} \end{array} $	$ \begin{array}{r} \boxed{1} \quad \boxed{0} \quad \boxed{0} \quad \boxed{0} \\ -) \qquad \qquad \qquad 1 \\ \hline \boxed{1} \quad \boxed{1} \quad \boxed{1} \quad \boxed{1} \end{array} $
	← 저장되지 않은 1이 있다고 가정 →

이는 마치 계수기를 거꾸로 돌리는 것과 같다. ‘0000’에서 정방향으로 돌리면 ‘0001’이 되지만 역방향으로 돌리면 ‘9999’가 되는 것이다.

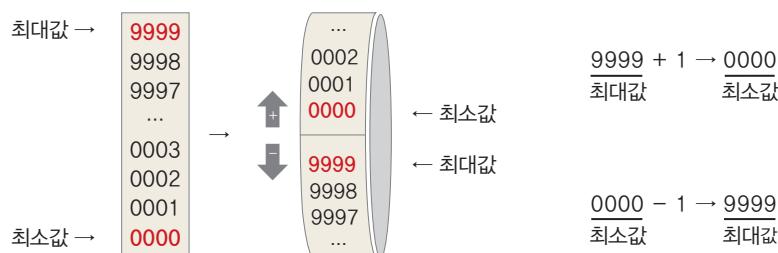
| 참고 | TV의 채널을 증가시키다가 마지막 채널에서 채널을 더 증가시키면 첫 번째 채널로 이동하고, 첫번째 채널에서 채널을 감소시키면 마지막 채널로 이동하는 것과 같다.

그래서 정수형 타입이 표현할 수 있는 최대값에 1을 더하면 최소값이 되고, 최소값에서 1을 빼면 최대값이 된다.

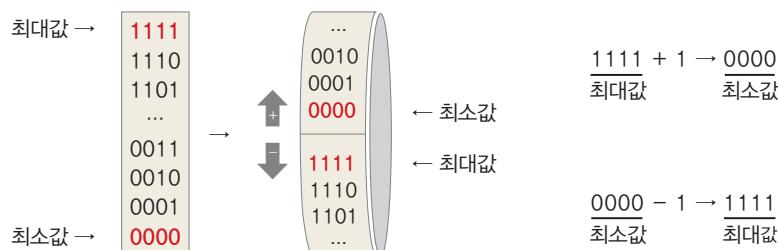
최대값 + 1 → 최소값

최소값 - 1 → 최대값

아래 그림과 같이 최소값과 최대값을 이어 놓았다고 생각하면 오버플로우의 결과를 더 이해하기 쉽다.



위의 그림을 2진수로 바꾸면 다음과 같다.



4 bit 2진수의 최소값인 ‘0000’부터 시작해서 1씩 계속 증가하다 최대값인 ‘1111’을 넘으면 다시 ‘0000’이 된다. 그래서 값을 무한히 1씩 증가시켜도 ‘0000’과 ‘1111’의 범위를 계속 반복하게 된다.

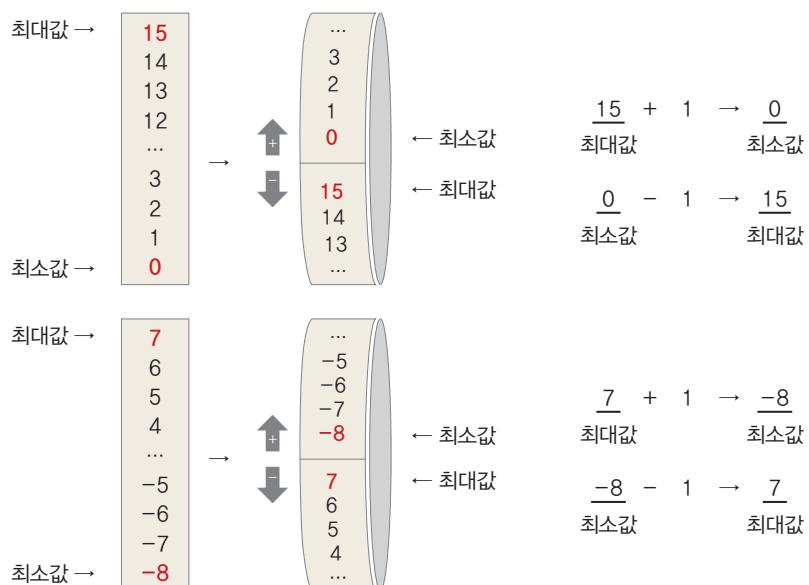
부호있는 정수의 오버플로우

부호없는 정수와 부호있는 정수는 표현범위 즉, 최대값과 최소값이 다르기 때문에 오버플로우가 발생하는 시점이 다르다. 부호없는 정수는 2진수로 '0000'이 될 때 오버플로우가 발생하고, 부호있는 정수는 부호비트가 0에서 1이 될 때 오버플로우가 발생한다.

부호없는 10진수	2진수	부호있는 10진수
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7 ← 최대값
8	1000	-8 ← 최소값 } 오버플로우 발생
9	1001	-7
10	1010	-6
11	1011	-5
12	1100	-4
13	1101	-3
14	1110	-2
최대값 → 15	1111	-1
최소값 → 0	0000	0
	0001	1
	0010	2

▲ 표 2-17 부호없는 정수와 부호있는 정수의 오버플로우

부호없는 정수의 경우 표현범위가 '0~15'이므로 이 값이 계속 반복되고, 부호있는 정수의 경우 표현범위가 '-8~7'이므로 이 값이 무한히 반복된다.



▼ 예제 2-9/OverflowEx.java

```
class OverflowEx {
    public static void main(String[] args) {
        short sMin = -32768;
        short sMax = 32767;
        char cMin = 0;
        char cMax = 65535;

        System.out.println("sMin = " + sMin);
        System.out.println("sMin-1= " + (short)(sMin-1));
        System.out.println("sMax = " + sMax);
        System.out.println("sMax+1= " + (short)(sMax+1));
        System.out.println("cMin = " + (int)cMin);
        System.out.println("cMin-1= " + (int)--cMin);
        System.out.println("cMax = " + (int)cMax);
        System.out.println("cMax+1= " + (int)++cMax);
    }
}
```

▼ 실행결과

sMin =	-32768
sMin-1=	32767
sMax =	32767
sMax+1=	-32768
cMin =	0
cMin-1=	65535
cMax =	65535
cMax+1=	0

short타입과 char타입의 최대값과 최소값에 1을 더하거나 빼 결과를 출력하였다. 실행결과를 좀더 이해하기 쉽게 정리하면 다음과 같다.

sMin - 1 → sMax	// 최소값 - 1 → 최대값
-32768	32767
sMax + 1 → sMin	// 최대값 + 1 → 최소값
32767	-32768
cMin - 1 → cMax	// 최소값 - 1 → 최대값
0	65535
cMax + 1 → cMin	// 최대값 + 1 → 최소값
65535	0

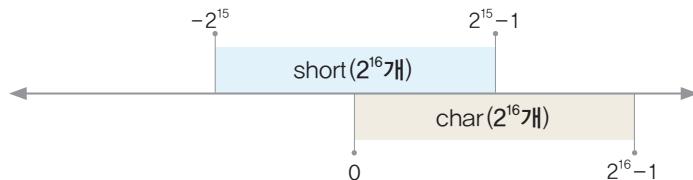
최소값에서 1을 빼면 최대값이 되고, 최대값에 1을 더하면 최소값이 된다는 것을 확인할 수 있다. 아직 이해가 안 된다면, 아래의 표가 도움이 될 것이다.

개수	부호	char	2진수(16bit)	short	부호
65536개 (2^{16} 개)	0 (1개)	최소값 → 0	0000000000000000	0	0 (1개)
	양수 ($2^{16}-1$ 개, 65535개)	1	0000000000000001	1	양수 ($2^{15}-1$ 개, 32767개)
		2	0000000000000010	2	
		3	0000000000000011	3	
		
		32765	0111111111111101	32765	
		32766	0111111111111110	32766	
		32767	0111111111111111	32767 ← 최대값	
		32768	1000000000000000	-32768 ← 최소값	음수 (2^{15} 개, 32768개)
		32769	1000000000000001	-32767	
		32770	1000000000000010	-32766	
		
		65532	1111111111111100	-4	
		65533	1111111111111101	-3	
		65534	1111111111111110	-2	
		최대값 → 65535	1111111111111111	-1	

▲ 표 2-18 char와 short의 비교

‘short’과 ‘char’의 크기는 모두 16 bit이므로 표현할 수 있는 값의 개수 역시 2^{16} 개(65536개)로 같다. 그러나 ‘short’은 이 중에서 절반(2^{15} 개=32768개)을 ‘음수’를 표현하는데 사용하고, ‘char’는 전체(2^{16} 개=65535+1개)를 ‘양수(65535개)와 0(1개)’을 표현하는데 사용한다.

16비트로 표현할 수 있는 정수의 개수 : 2^{16} 개 (65536개)
 short타입의 표현범위 : $-2^{15} \sim 2^{15}-1$ (-32768 ~ 32767)
 char타입의 표현범위 : 0 ~ $2^{16}-1$ (0 ~ 65535)



4.4 실수형 – float, double

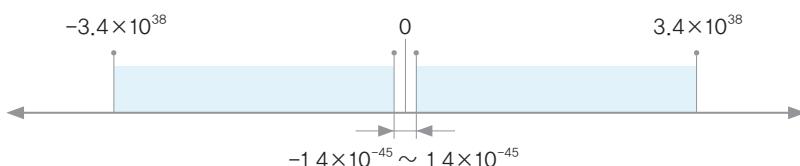
실수형의 범위와 정밀도

실수형은 실수를 저장하기 위한 타입으로 float와 double, 두 가지가 있으며 각 타입의 변수에 저장할 수 있는 값의 범위는 아래와 같다.

타입	저장 가능한 값의 범위(양수)	정밀도	크기	
			bit	byte
float	$1.4 \times 10^{-45} \sim 3.4 \times 10^{38}$	7자리	32	4
double	$4.9 \times 10^{-324} \sim 1.8 \times 10^{308}$	15자리	64	8

▲ 표2-19 실수형의 범위와 정밀도

표2-18의 범위는 ‘양의 범위’만 적은 것으로, 이 범위에 ‘-’부호를 붙이면 ‘음의 범위’가 된다. 예를 들어 float타입으로 표현가능한 음의 범위는 $-3.4 \times 10^{38} \sim -1.4 \times 10^{-45}$ 이다. float타입으로 표현가능한 양의 범위와 음의 범위를 함께 그림으로 그리면 다음과 같다.



▲ 그림2-12 float타입으로 표현할 수 있는 값의 범위

즉, float타입의 표현범위는 $-3.4 \times 10^{38} \sim 3.4 \times 10^{38}$ 이지만, $-1.4 \times 10^{-45} \sim 1.4 \times 10^{-45}$ 범위(0은 제외)의 값은 표현할 수 없다. 실수형은 소수점수도 표현해야 하므로 ‘얼마나 큰 값을 표현할 수 있는가’뿐만 아니라 ‘얼마나 0에 가깝게 표현할 수 있는가’도 중요하다.

Q. 실수형도 정수형처럼 저장할 수 있는 범위를 넘게 되면 오버플로우가 발생하나요?

A. 앞서 정수형에서 변수의 값이 표현범위를 벗어나는 것을 ‘오버플로우(overflow)’라고 배웠습니다. 실수형에서도 변수의 값이 표현범위의 최대값을 벗어나면 ‘오버플로우’가 발생하는데요. 정수형과 달리 실수형에서는 오버플로우가 발생하면 변수의 값은 ‘무한대(infinity)’가 됩니다.

그리고 정수형에는 없는 ‘언더플로우(underflow)’가 있는데, ‘언더플로우’는 실수형으로 표현할 수 없는 아주 작은 값, 즉 양의 최소값보다 작은 값이 되는 경우를 말합니다. 이 때 변수의 값은 0이 됩니다.

4 byte의 정수로는 ‘약 $\pm 2 \times 10^9$ ’의 값밖에 표현할 수 없는데, 어떻게 같은 4 byte로 ‘ $\pm 3.4 \times 10^{38}$ ’과 같이 큰 값을 표현할 수 있는 것일까? 그 이유는 바로 값을 저장하는 형식이 다르기 때문이다.

int : 1 + 31 = 32 (4 byte)

S (1)	31 bit
-------	--------

float : 1 + 8 + 23 = 32 (4 byte)

S (1)	E (8)	M (23)
-------	-------	--------

▲ 그림 2-13 int타입과 float타입의 표현형식

위 그림은 int타입과 float타입의 표현형식을 비교한 것인데, int타입은 ‘부호와 값’, 두 부분으로 이루어져 있지만, float타입과 같은 실수형은 ‘부호(S), 지수(E), 가수(M)’, 세 부분으로 이루어져 있다. 즉, ‘2의 제곱을 곱한 형태($\pm M \times 2^E$)’로 저장하기 때문에 이렇게 큰 범위의 값을 저장하는 것이 가능한 것이다.

그러나 정수형과 달리 실수형은 오차가 발생할 수 있다는 단점이 있다. 그래서 실수형에는 표현할 수 있는 값의 범위뿐만 아니라 ‘정밀도(precision)’도 중요한 요소이다.

표 2-18을 보면 float타입은 정밀도가 7자리인데, 이것은 ‘ $a \times 10^n$ ’ ($1 \leq a < 10$)의 형태로 표현된 ‘7자리의 10진수를 오차없이 저장할 수 있다’는 뜻으로 아래의 세 값은 float타입의 변수에 저장했을 때 오차없이 저장할 수 있다.

$$\begin{array}{ll} 1234.567 & = 1.234567 \times 10^3 \\ 0.00001234567 & = 1.234567 \times 10^{-5} \\ 1234567000 & = 1.234567 \times 10^9 \end{array}$$

만일 7자리 이상의 정밀도가 필요하다면, 변수의 타입을 double로 해야 한다. double타입은 float타입보다 정밀도가 약 2배인, 10진수로 15자리의 정밀도를 가지므로 float타입보다 훨씬 더 정밀하게 값을 표현할 수 있다.

실수형 값을 저장할 때, float타입이 아닌 double타입의 변수를 사용하는 경우는 대부분 저장하려는 ‘값의 범위’ 때문이 아니라 ‘보다 높은 정밀도’가 필요해서이다.

| 참고 | double이라는 이름은 float보다 약 2배(double)의 정밀도를 갖는다는 의미에서 붙여진 것이다.

연산속도의 향상이나 메모리를 절약하려면 float를 선택하고, 더 큰 값의 범위라던가 더 높은 정밀도를 필요로 한다면 double을 선택해야 한다.

▼ 예제 2-10/FFloatEx1.java

```
class FFloatEx1 {
    public static void main(String[] args) {
        float f = 9.12345678901234567890f;
        float f2 = 1.2345678901234567890f;
        double d = 9.12345678901234567890d;

        System.out.printf(" 123456789012345678901234%n");
        System.out.printf("f : %f%n", f); // 소수점 이하 6째자리까지 출력.
        System.out.printf("f : %24.20f%n", f);
        System.out.printf("f2 : %24.20f%n", f2);
        System.out.printf("d : %24.20f%n", d);
    }
}
```

▼ 실행결과

```
123456789012345678901234
f : 9.123457 ← 7자리에서 반올림되었음
f : 9.12345695495605500000
f2 : 1.23456788063049320000
d : 9.12345678901234600000
```

실수형 값을 출력할 때는 printf메서드의 지시자 '%f'를 사용한다. '%f'는 기본적으로 소수점 이하 6자리까지만 출력하므로, 7번째 자리에서 반올림되어 '9.123457'이 되었다.

```
System.out.printf("f : %f\n", f); // f : 9.123457
```

앞서 배운 것처럼, '%24.20f'는 전체 24자리 중에서 20자리는 소수점 이하의 수를 출력하라는 뜻이다.

```
System.out.printf("f : %24.20f\n", f);
```

1	2	3	4	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0
		9	.	1	2	3	4	5	6	9	5	4	9	5	6	0	5	5	0	0	0	0	0

위의 그림을 보면 실제로 저장된 값은 '9.123456954956055'인데 앞뒤의 빈자리가 공백과 0으로 채워진 것을 알 수 있다.

9.12345678901234567890 원래 값
 ↓
 9.12345695495605500000 저장된 값

float타입의 변수 f에 저장하려던 원래의 값은 '9.12345678901234567890'이지만 저장공간의 한계로 오차가 발생하여 실제 저장된 값은 '9.12345695495605500000'이다.

그러나 이 두 값이 앞의 7자리는 일치한다. 정밀도가 7자리이므로 원래의 값에서 7자리의 값만 오차없이 저장된 것이다.

```
System.out.printf("f2 : %24.20f\n", f2); // f2 : 1.23456788063049320000
```

위와 같이 간혹 원래의 값과 8자리이상 일치하는 경우도 있지만 항상 그런 것은 아니기 때문에 이런 결과를 기대해서는 안 된다.

실수형의 저장형식

앞서 언급한 바와 같이 실수형은 정수형과 저장형식이 달라서, 실수형은 값을 부동 소수점(floating-point)의 형태로 저장한다. 부동 소수점은 실수를 ' $\pm M \times 2^E$ '와 같은 형태로 표현하는 것을 말하며, 부동 소수점은 부호(Sign), 지수(Exponent), 가수(Mantissa), 모두 세 부분으로 이루어져 있다.

$$\pm M \times 2^E$$

그래서 부동소수점수는 다음과 같이 세 부분으로 나누어 저장된다.

float : 1 + 8 + 23 = 32 (4 byte)

S (1)	E (8)	M (23)
-------	-------	--------

double : 1 + 11 + 52 = 64 (8 byte)

S (1)	E (11)	M (52)
-------	--------	--------

▲ 그림 2-14 float와 double의 표현형식

위와 같은 표현형식은 IEEE754라는 표준을 따른 것인데, IEEE754는 ‘전기 전자 기술자 협회(IEEE, Institute of Electrical and Electronics Engineers)’에서 제정한 부동 소수점의 표현방법이다.

기호	의미	설명
S	부호(Sign bit)	0이면 양수, 1이면 음수
E	지수(Exponent)	부호있는 정수. 지수의 범위는 -127 ~ 128(float), -1023 ~ 1024(double)
M	가수(Mantissa)	실제값을 저장하는 부분. 10진수로 7자리(float), 15자리(double)의 정밀도로 저장 가능

▲ 표 2-20 실수 표현형식의 구성요소

1. 부호(Sign bit)

‘S’는 부호비트(sign bit)를 의미하며 1 bit이다. 이 값이 0이면 양수를, 1이면 음수를 의미한다. 정수형과 달리 ‘2의 보수법’을 사용하지 않기 때문에 양의 실수를 음의 실수로 바꾸려면 그저 부호비트만 0에서 1로 변경하면 된다.

2. 지수(Exponent)

'E'는 지수를 저장하는 부분으로 float의 경우, 8 bit의 저장공간을 갖는다. 지수는 '부호 있는 정수'이고 8 bit로는 모두 $2^8 (=256)$ 개의 값을 저장할 수 있으므로, '-127~128'의 값이 저장된다. 이 중에서 -127과 128은 '숫자 아님(NaN, Not a Number)'이나 '양의 무한대(POSITIVE_INFINITY)', '음의 무한대(NEGATIVE_INFINITY)'와 같이 특별한 값의 표현을 위해 예약되어 있으므로 실제로 사용가능한 지수의 범위는 '-126~127'이다. 그래서 지수의 최대값이 127이므로 float타입으로 표현할 수 있는 최대값은 2^{127} 이고, 10진수로 약 10^{38} 이다. 그러나 float의 최소값은 가수의 마지막 자리가 2^{-23} 이므로 지수의 최소값보다 2^{-23} 배나 더 작은 값, 약 10^{-45} 이다.

$$0.0000000000000000000000001 \times 2^{-126} = 1.0 \times 2^{-149} \approx 10^{-45}$$

3. 가수(Mantissa)

'M'은 실제 값인 가수를 저장하는 부분으로 float의 경우, 2진수 23자리를 저장할 수 있다. 2진수 23자리로는 약 7자리의 10진수를 저장할 수 있는데 이것이 바로 float의 정밀도가 된다. double은 가수를 저장할 수 있는 공간이 52자리로 float보다 약 2배이므로 double이 float보다 약 2배의 정밀도를 갖는 것이다.

부동 소수점의 오차

실수 중에는 '파이($\pi=3.141592\dots$)'와 같은 **무한 소수가 존재**하므로, 정수와 달리 실수를 저장할 때는 오차가 발생할 수 있다. 게다가 10진수가 아닌 2진수로 저장하기 때문에 10진수로는 유한 소수이더라도, **2진수로 변환하면 무한 소수가 되는 경우도 있다**. 2진수로는 10진 소수를 정확히 표현하기 어렵기 때문이다.

<u>9</u>	.	<u>1234567</u>
↓		↓
1001 . 00011111001101011011011...		

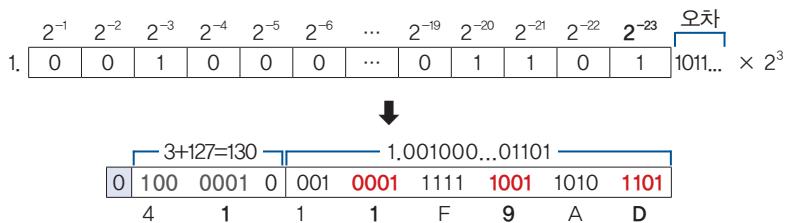
위의 그림에서 알 수 있듯이 9.1234567은 10진수로 유한 소수지만, 2진수로는 무한 소수이다. 즉, 2진수로는 이 값을 정확히 표현하지 못한다는 얘기다. 여기서부터 벌써 오차가 발생한다. 비록 2진수로 유한소수라도, 가수를 **저장할 수 있는 자리수가 한정되어 있으므로 저장되지 못하고 버려지는 값들이 있으면 오차가 발생한다**.

2진수로 변환된 실수를 저장할 때는 먼저 ' $1.\dots \times 2^n$ '의 형태로 변환하는데, 이 과정을 정규화라고 한다.

1001.00011111001101011001111...	정규화	$1.\overline{0010001111100110101101}1011\dots \times 2^3$
---------------------------------	-----	---

정규화된 2진 실수는 항상 '1.'으로 시작하기 때문에, '1.'을 제외한 23자리의 2진수가 가수(mantissa)로 저장되고 그 이후는 잘려나간다. 지수는 기저법으로 저장되기 때문에, 지수인 3에 기저인 127을 더한 130이 2진수로 변환되어 저장된다. 10진수 130은 2진수로 '10000010'이다.

| 참고 | 기저법은 '2의 보수법'처럼 부호있는 정수를 저장하는 방법이다. 저장할 때 특정값(기저)을 더했다가 읽어올 때는 다시 뺀다.



이 때 잘려나간 값들에 의해 발생할 수 있는 최대오차는 약 2^{-23} 인데, 이 값은 가수의 마지막 비트의 단위와 같다. 2^{-23} 은 10진수로 0.0000001192(약 10^{-7})이므로 float의 정밀도가 7자리라고 하는 것이다. 어떤 책에서는 '소수점이하 6자리'라고 하는데, 이것은 소수점이하의 자릿수만을 셀 것으로, 결국 같은 얘기다.

$$2^{-23} \approx \begin{array}{l} \boxed{\text{7자리}} \\ \text{0.000000}1192 \approx 10^{-7} \\ \boxed{\text{6자리}} \end{array}$$

다음의 예제를 보면, float타입의 값이 실제로 어떻게 저장되는지 직접 확인할 수 있다.

▼ 예제 2-11/FLOATTOBINEX.java

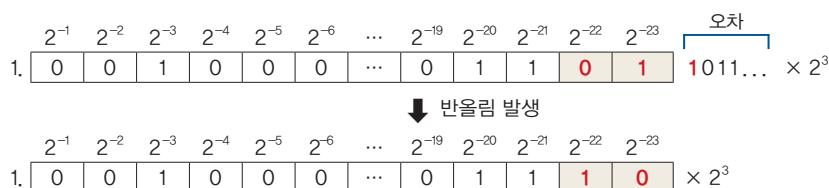
```
class FloatToIntEx {
    public static void main(String args[]) {
        float f = 9.1234567f;
        int i = Float.floatToIntBits(f);

        System.out.printf("%f%n", f);
        System.out.printf("%X%n", i); // 16진수로 출력
    } // main의 끝
}
```

▼ 실행결과
 9.123457
 4111F9AE

Float클래스의 floatToIntBits()는 float타입의 값을 int타입의 값으로 해석해서 반환한다. 반환된 값을 16진수로 출력하면, float타입의 값이 2진수로 어떻게 저장되는지 확인할 수 있다. 실행결과를 보면, 앞서 설명한 것과 달리 0x4111F9AE이다. 그 이유는 잘려나간 첫 번째 자리의 값이 1이라서 반올림되어 0x4111F9AD의 2진수 마지막 자리 두 자리의 값이 '01'에서 '10'으로 1증가했기 때문이다.

| 참고 | 10진수와 달리 2진수는 10이면, 반올림 한다. 2진수에서는 10이 절반이기 때문이다.



5. 형변환

5.1. 형변환(캐스팅, casting)이란?

모든 변수와 리터럴에는 타입이 있다는 것을 배웠다. 프로그램을 작성하다 보면 같은 타입뿐만 아니라 서로 다른 타입간의 연산을 수행해야하는 경우도 있다. 이럴 때는 연산을 수행하기 전에 타입을 일치시켜야 하는데, 변수나 리터럴의 타입을 다른 타입으로 변환하는 것을 ‘형변환(casting)’이라고 한다.

형변환이란, 변수 또는 상수의 타입을 다른 타입으로 변환하는 것

예를 들어 int타입의 값과 float타입의 값을 더하는 경우, 먼저 두 값을 같은 타입으로 즉, 둘 다 float타입으로 변환한 다음에 더해야 한다.

5.2 형변환 방법

형변환 방법은 아주 간단하다. 형변환하고자 하는 변수나 리터럴의 앞에 변환하고자 하는 타입을 괄호와 함께 붙여주기만 하면 된다.

(타입)피연산자

여기에 사용되는 괄호()는 ‘캐스트 연산자’또는 ‘형변환 연산자’라고 하며, 형변환을 ‘캐스팅(casting)’이라고도 한다.

예를 들어 다음과 같은 코드가 있을 때,

```
double d = 85.4;
int score = (int)d; // double타입의 변수 d를 int타입으로 형변환
```

두 번째 줄의 연산과정을 단계별로 살펴보면 다음과 같다.

```
int score = (int)d;
→ int score = (int)85.4;    // 변수 d의 값을 읽어 와서 형변환한다.
→ int score = 85;           // 형변환의 결과인 85를 변수 score에 저장한다.
```

이 과정에서 알 수 있듯이, 형변환 연산자는 그저 피연산자의 값을 읽어서 지정된 타입으로 형변환하고 그 결과를 반환할 뿐이다. 그래서 피연산자인 변수 d의 값은 형변환 후에도 아무런 변화가 없다.

▼ 예제 2-12/CastingEx.java

```
class CastingEx {  
    public static void main(String[] args) {  
        double d = 85.4;  
        int score = (int)d;  
  
        System.out.println("score="+score);  
        System.out.println("d="+d);  
    }  
}
```

▼ 실행결과

```
score=85  
d=85.4 ← 형변환 후에도 피연산자에는 아무런 변화가 없다.
```

기본형(primitive type)에서 boolean을 제외한 나머지 타입들은 서로 형변환이 가능하다. 그리고 기본형과 참조형 간의 형변환은 불가능하다. 참조형의 형변환은 7장에서 설명할 것이고, 여기서는 기본형의 형변환에 대해서만 다룬다.

변환	수식	결과
int → char	(char) 65	'A'
char → int	(int) 'A'	65
float → int	(int) 1.6f	1
int → float	(float) 10	10.0f

▲ 표 2-21 기본형 간의 형변환

표 2-20은 형변환의 몇 가지 예를 보여준다. float타입의 값을 int타입으로 변환할 때 소수점 이하의 값은 반올림이 아닌 버림으로 처리된다는 점을 눈여겨보자. char타입도 실제로는 정수로 저장되므로 값이 저장되는 형식은 크게 정수와 실수, 2가지 뿐이다.

5.3 정수형 간의 형변환

큰 타입에서 작은 타입으로의 변환, 예를 들어서 int타입(4 byte)의 값을 byte타입(1 byte)으로 변환하는 경우는 아래와 같이 크기의 차이만큼 잘려나간다. 그래서 경우에 따라 ‘값 손실(loss of data)’이 발생할 수 있다.

변환	2진수	10진수	값 손실																																																			
int ↓ byte	<table border="1"><tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td></tr><tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td></tr></table>	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0																				0	0	0	1	0	1	0	10 10	없음	
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	0																															
																			0	0	0	1	0	1	0																													
int ↓ byte	<table border="1"><tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td></tr><tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td><td>0</td></tr></table>	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	1	0	0																				0	0	1	0	1	1	0	300 44	있음
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	1	1	0	0																														
																			0	0	1	0	1	1	0																													

▲ 표 2-22 정수형 간의 형변환 – 큰 타입에서 작은 타입으로 변환

반대로 작은 타입에서 큰 타입으로의 변환, 예를 들어서 byte타입(1 byte)의 값을 int타입(4 byte)으로 변환하는 경우는 저장공간의 부족으로 잘려나가는 일이 없으므로 값 손실이 발생하지 않는다. 그리고 나머지 빈공간은 0 또는 1로 채워진다.

변환	2진수	10진수	값손실
byte ↓ int		10 10	없음

▲ 표2-23 정수형간의 형변환 – 작은 타입에서 큰 타입으로 변환(양수)

원래의 값을 채우고 남은 빈공간은 0으로 채우는 게 보통이지만, 변환하려는 값이 음수인 경우에는 빈 공간을 1로 채운다. 그 이유는 형변환 후에도 부호를 유지할 수 있도록 하기 위해서이다.

▲ 표 2-24 정수형간의 형변환 – 작은 타입에서 큰 타입으로 변환(음수)

| 참고 | 2의 보수법은 p.72에 있다.

▼ 예제 2-13/CastingEx2.java

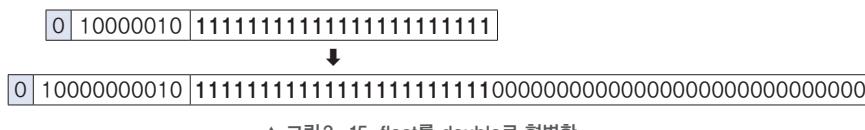
```
class CastingEx2 {  
    public static void main(String[] args) {  
        int i = 10;  
        byte b = (byte)i;  
        System.out.printf("[int -> byte] i=%d -> b=%d%n", i, b);  
  
        i = 300;  
        b = (byte)i;  
        System.out.printf("[int -> byte] i=%d -> b=%d%n", i, b);  
  
        b = 10;  
        i = (int)b;  
        System.out.printf("[byte -> int] b=%d -> i=%d%n", b, i);  
  
        b = -2;  
        i = (int)b;  
        System.out.printf("[byte -> int] b=%d -> i=%d%n", b, i);  
  
        System.out.println("i="+Integer.toBinaryString(i));  
    }  
}
```

▼ 실행결과

앞서 설명한 내용들을 확인해 보는 예제이다. 예제의 마지막에는 변수 i의 값인 -2가 2진 수로 출력된 것이다. 이처럼 'Integer.toBinaryString(int i)'라는 메서드를 이용하면, 10진 정수를 2진 정수로 변환한 문자열을 얻을 수 있다.

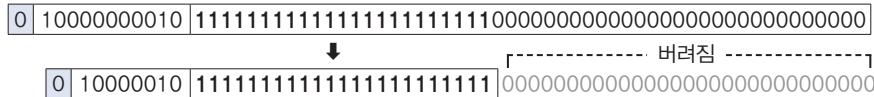
5.4 실수형 간의 형변환

실수형에서도 정수형처럼 작은 타입에서 큰 타입으로 변환하는 경우, 빈 공간을 0으로 채운다. float타입의 값을 double타입으로 변환하는 경우, 지수(E)는 float의 기저인 127을 뺀 후 double의 기저인 1023을 더해서 변환하고, 가수(M)는 float의 가수 23자리를 채우고 남은 자리를 0으로 채운다. 지수의 변화보다는 가수의 변화를 눈여겨보자.



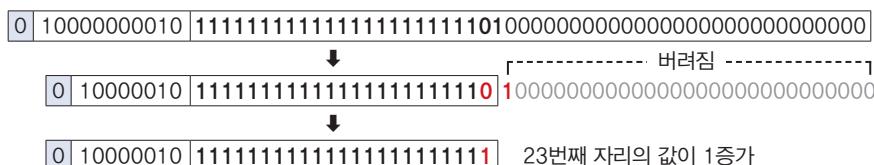
▲ 그림 2-15 float를 double로 형변환

반대로 double타입에서 float타입으로 변환하는 경우, 지수(E)는 double의 기저인 1023을 뺀 후 float의 기저인 127을 더하고 가수(M)는 double의 가수 52자리 중 23자리만 저장되고 나머지는 버려진다.



▲ 그림 2-16 double을 float로 형변환

한 가지 주의할 점은 형변환할 때 가수의 24번째 자리에서 반올림이 발생할 수 있다는 것이다. 24번째 자리의 값이 10이면, 반올림이 발생하여 23번째 자리의 값이 1증가한다.



▲ 그림2-17 double을 float로 형변환할 때 반올림이 발생하는 경우

그리고 float타입의 범위를 넘는 값을 float로 형변환하는 경우는 ‘ \pm 무한대’ 또는 ‘ ± 0 ’을 결과로 얻는다.

```
double d = 1.0e100; // float의 최대값보다 큰 값을 d에 저장( $1.0 \times 10^{100}$ )
float f = (float)d; // d의 값을 float로 형변환해서 f에 저장. f는 무한대가 된다.

double d = 1.0e-50; // float의 최소값보다 작은 값을 d에 저장( $1.0 \times 10^{-50}$ )
float f = (float)d; // f의 값은 0이 된다.
```

▼ 예제 2-14/CastingEx3.java

```
class CastingEx3 {  
    public static void main(String[] args) {  
        float f = 9.1234567f;  
        double d = 9.1234567;  
        double d2 = (double)f;  
  
        System.out.printf("f =%20.18f\n", f);  
        System.out.printf("d =%20.18f\n", d);  
        System.out.printf("d2=%20.18f\n", d2);  
    }  
}
```

▼ 실행결과

변수 f와 d에 같은 값을 저장했지만, 실제로 저장되는 값은 다르다. 변수 f에 저장된 값을 double타입으로 형변환해도 같은 그대로이다. 왜 이런 결과를 얻는지 하나씩 자세히 살펴보자.

① float f = 9.1234567f;

$$9.1234567 = 1.99 \cdot 10^0 + 0.99 \cdot 10^{-1} + 1.11 \cdot 10^{-2} + 1.11 \cdot 10^{-3} + 1.11 \cdot 10^{-4} + 1.11 \cdot 10^{-5} + 1.00 \cdot 10^{-6} + 1.11 \cdot 10^{-7} + 1.00 \cdot 10^{-8} + 1.11 \cdot 10^{-9} + 1.00 \cdot 10^{-10} + 1.00 \cdot 10^{-11} + 1.00 \cdot 10^{-12} + \dots \times 10^{-3}$$

01000010 0010001111100110101101
↓
0101110001111100011011010
반올림발생, 23번째 자리의 값이 1증가

② double d = 9.1234567;

$$9.1234567 = 1.00100011111001101011011011101110001111000110110100111001 \dots \times 2^3$$

0	10000000010	00100011111001101011011011100011111000110110100	111001...
0	10000000010	00100011111001101011011011100011111000110110101	반올림 발생

같은 값을 저장해도 float와 double의 정밀도 차이 때문에 서로 다른 값이 저장된다.

f≡9.123456954956054700

0 10000010 00100011111100110101110

d=9 123456700000000200

010000000100010001111110011010110110111011100011111000110110101

③ double d2 = (double) f;

f 0 10000010 00100011111100110101110

↓

저장할 때 이미 값이 달라졌기 때문에, 형변환을 해도 값이 같아지지 않는다.

d 0 10000000010 00100011111100110101110 10111011100011111000110110101

d2 0 10000000010 00100011111100110101110 00000000000000000000000000000000

5.5 정수형과 실수형 간의 형변환

정수형과 실수형은 저장형식이 완전히 다르기 때문에 정수형 간의 변환처럼 간단히 값을 채우고 자르는 식으로 할 수 없다. 좀 더 복잡한 변환과정을 거쳐야한다.

int : $1 + 31 = 32$ (4 byte)

S (1)	31 bit		
-------	--------	--	--

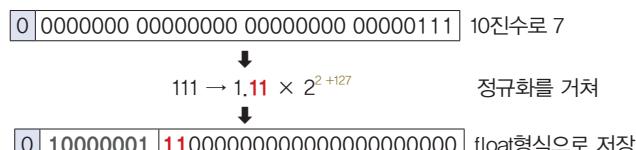
float : $1 + 8 + 23 = 32$ (4 byte)

S (1)	E (8)	M (23)
-------	-------	--------

▲ 그림2-18 int타입과 float타입의 저장형식 비교

정수형을 실수형으로 변환

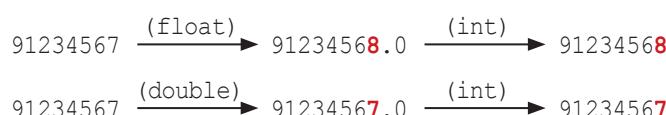
정수는 소수점이하의 값이 없으므로 비교적 변환이 간단하다. 그저 정수를 2진수로 변환한 다음 정규화를 거쳐 실수의 저장형식으로 저장될 뿐이다. 이 과정은 이미 실수형의 저장형식에서 설명했으므로 자세한 내용은 생략한다. 아래의 그림은 10진수 7을 float타입의 변수에 저장되는 과정을 보여준다.



▲ 그림2-19 정수 7이 float타입으로 형변환되는 과정

실수형은 정수형보다 훨씬 큰 저장범위를 갖기 때문에, 정수형을 실수형으로 변환하는 것은 별 무리가 없다. 정수를 2진수로 변환한 다음에 정규화해서 실수의 저장형식에 맞게 저장할 뿐이다. 한 가지 주의할 점은 실수형의 정밀도의 제한으로 인한 오차가 발생할 수 있다는 것이다.

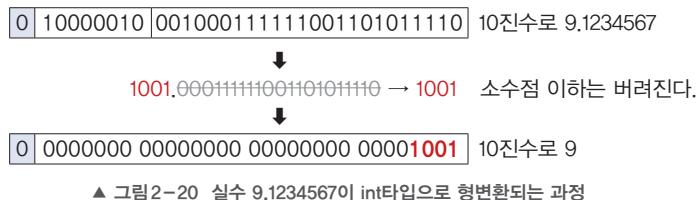
예를 들어 int의 최대값은 약 20억으로 최대 10자리의 정밀도를 요구한다. 그러나 float는 10진수로 약 7자리의 정밀도만을 제공하므로, int를 float로 변환할 때 정밀도 차이에 의한 오차가 발생할 수 있다. 그래서 10진수로 8자리 이상의 값을 실수형으로 변환할 때는 float가 아닌 double로 형변환해야 오차가 발생하지 않을 것이다.



위의 그림은 8자리의 int값을 각각 float와 double로 변환한 다음에 다시 int로 형변환했을 때 값의 변화를 보여준다. float는 정밀도가 약 7자리이므로 8자리의 정수를 저장할 때 오차가 발생하는 것을 알 수 있다. 반면에 double은 약 15자리의 정밀도를 갖기 때문에 오차없이 변환이 가능하다.

실수형을 정수형으로 변환

실수형을 정수형으로 변환하면, 실수형의 소수점이하 값은 버려진다. 정수형의 표현 형식으로 소수점 이하의 값은 표현할 수 없기 때문이다. 예를 들어 float타입의 상수 9.1234567f를 int타입으로 형변환 하면 9가 된다.



▲ 그림 2-20 실수 9.1234567f int타입으로 형변환되는 과정

그래서 실수형을 정수형으로 형변환할 때 반올림이 발생하지 않는다. 예를 들어, 실수 1.666을 int로 형변환하면, 1이 된다.

$$1.666 \xrightarrow{(\text{int})} 1$$

만일 실수의 소수점을 버리고 남은 정수가 정수형의 저장범위를 넘는 경우에는 정수의 오버플로우가 발생한 결과를 얻는다.

▼ 예제 2-15/CastingEx4.java

```
class CastingEx4 {  
    public static void main(String[] args) {  
        int i = 91234567; // 8자리의 10진수  
        float f = (float)i; // int를 float로 형변환  
        int i2 = (int)f; // float를 다시 int로 형변환  
  
        double d = (double)i; // int를 double로 형변환  
        int i3 = (int)d; // double을 다시 int로 형변환  
  
        float f2 = 1.666f;  
        int i4 = (int)f2;  
  
        System.out.printf("i=%d\n", i);  
        System.out.printf("f=%f i2=%d\n", f, i2);  
        System.out.printf("d=%f i3=%d\n", d, i3);  
        System.out.printf("(int)%f=%d\n", f2, i4);  
    }  
}
```

▼ 실행결과

```
i=91234567  
f=91234568.000000 i2=91234568  
d=91234567.000000 i3=91234567  
(int)1.666000=1
```

5.6 자동 형변환

서로 다른 타입 간의 대입이나 연산을 할 때, 형변환으로 타입을 일치시키는 것이 원칙이다. 하지만, 경우에 따라 편의상의 이유로 형변환을 생략할 수 있다. 그렇다고 해서 형변환되지 않는 것은 아니고, 컴파일러가 생략된 형변환을 자동적으로 추가한다.

```
float f = 1234; // 형변환의 생략. float f = (float)1234;와 같음.
```

위의 문장에서 우변은 int타입의 상수이고, 이 값을 저장하려는 변수의 타입은 float이다. 서로 타입이 달라서 형변환이 필요하지만 편의상 생략하였다. float타입의 변수는 1234라는 값을 저장하는데 아무런 문제가 없기 때문이다.

그러나 다음과 같이 변수가 저장할 수 있는 값의 범위보다 더 큰 값을 저장하려는 경우에 형변환을 생략하면 에러가 발생한다.

```
byte b = 1000; // 에러. byte의 범위 (-128~127)를 넘는 값을 저장.
```

에러 메시지는 ‘incompatible types: possible lossy conversion from int to byte’인데, 앞서 배운 것과 같이 큰 타입에서 작은 타입으로의 형변환은 값 손실이 발생할 수 있다는 뜻이다.

그러나 다음과 같이 명시적으로 형변환 해줬을 경우, 형변환이 프로그래머의 실수가 아닌 의도적인 것으로 간주하고 컴파일러는 에러를 발생시키지 않는다.

```
char ch = (char)1000; // 명시적 형변환. 에러가 발생하지 않는다.
```

또 다른 예로 다음과 같은 계산식에서 자주 형변환이 생략되는데, 서로 다른 두 타입의 연산에서는 먼저 타입을 일치시킨 다음에 연산을 수행해야 하므로, 연산과정에서 형변환이 자동적으로 이루어진다.

```
int i = 3;
double d = 1.0 + i; // double d = 1.0 + (double)i;에서 형변환이 생략됨
```

서로 다른 두 타입 간의 덧셈에서는 두 타입 중 표현범위가 더 넓은 타입으로 형변환하여 타입을 일치시킨 다음에 연산을 수행한다. 그렇게 하는 것이 값손실의 위험이 더 적어서 올바른 결과를 얻을 확률이 높기 때문이다.

```
double d = 1.0 + i;
→ double d = 1.0 + (double)i;
→ double d = 1.0 + (double)3; // 3을 double타입으로 형변환하면 3.0이 된다.
→ double d = 1.0 + 3.0; // double과 double의 덧셈결과 타입은 double.
→ double d = 4.0; // double + double = double
```

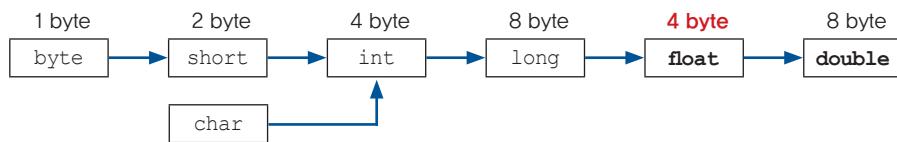
이처럼 연산과정에서 자동적으로 발생하는 형변환을 ‘산술 변환’이라고 하며, 다음 장에서 자세히 살펴볼 것이다.

자동 형변환의 규칙

형변환을 하는 이유는 주로 서로 다른 두 타입을 일치시키기 위해서인데, 형변환을 생략하면 컴파일러가 알아서 자동적으로 형변환을 한다고 했다. 그러면 컴파일러는 어떤 판단 기준으로 타입을 일치시킬까?

기존의 값을 최대한 보존할 수 있는 타입으로 자동 형변환한다.

표현범위가 좁은 타입에서 넓은 타입으로 형변환하는 경우에는 값 손실이 없으므로 두 타입 중에서 표현범위가 더 넓은 쪽으로 형변환된다.



▲ 그림2-21 기본형의 자동 형변환이 가능한 방향

그림2-21은 형변환이 가능한 7개의 기본형을 왼쪽부터 오른쪽으로 표현할 수 있는 값의 범위가 작은 것부터 큰 것의 순서로 나열한 것이다.

화살표방향으로의 변환, 즉 왼쪽에서 오른쪽으로의 변환은 형변환 연산자를 사용하지 않아도 자동 형변환이 되며, 그 반대 방향으로의 변환은 반드시 형변환 연산자를 써줘야 한다.

보통 자료형의 크기가 큰 것일수록 값의 표현범위가 크기 마련이지만, 실수형은 정수형과 값을 표현하는 방식이 다르기 때문에 같은 크기일지라도 실수형이 정수형보다 훨씬 더 큰 표현 범위를 갖기 때문에 float와 double이 같은 크기인 int와 long보다 오른쪽에 위치한다.

| 참고 | 정수형을 실수형으로 형변환하는 경우, 정밀도의 한계로 인한 오차가 발생할 수 있다.

char와 short은 둘 다 2 byte의 크기로 크기가 같지만, char의 범위는 ‘0~ $2^{16}-1$ (0~65535)’이고 short의 범위는 ‘ $-2^{15} \sim 2^{15}-1$ (-32768~32767)’이므로 서로 범위가 달라서 둘 중 어느 쪽으로의 형변환도 값 손실이 발생할 수 있으므로 자동 형변환이 수행될 수 없다.

1. boolean을 제외한 나머지 7개의 기본형은 서로 형변환이 가능하다.
2. 기본형과 참조형은 서로 형변환할 수 없다.
3. 서로 다른 타입의 변수 간의 연산은 형변환을 하는 것이 원칙이지만,
값의 범위가 작은 타입에서 큰 타입으로의 형변환은 생략할 수 있다.

자주 사용되는 타입 간의 변환

기본적인 타입 간의 변환은 프로그램에서 자주 사용되므로 반드시 기억해 두자.

1. 숫자를 문자로 변환 – 숫자에 '0'을 더한다.

`(char) (3 + '0') → '3'`

2. 문자를 숫자로 변환 – 문자에서 '0'을 뺀다.

`'3' - '0' → 3`

3. 숫자를 문자열로 변환 – 숫자에 빈 문자열("")을 더한다.

`3 + "" → "3"`

4. 문자열을 숫자로 변환 – Integer.parseInt(), Double.parseDouble()을 사용한다.

`Integer.parseInt("3") → 3`

`Double.parseDouble("3.14") → 3.14`

5. 문자열을 문자로 변환 – charAt(0)을 사용한다.

`"3".charAt(0) → '3'`

6. 문자를 문자열로 변환 – 빈 문자열("")을 더한다.

`'3' + "" → "3"`

▼ 예제 2-16/CastingEx5.java

```
class CastingEx5 {
    public static void main(String[] args) {
        String str = "3";
        System.out.println(str.charAt(0) - '0');
        System.out.println('3' - '0' + 1);
        System.out.println(Integer.parseInt("3") + 1);
        System.out.println("3" + 1);
        System.out.println((char) (3 + '0'));
    }
}
```

▼ 실행결과
3
4
4
31
3

| 참고 | 연습문제는 깃헙(<https://github.com/castello/javajungsuk4>)에서 PDF파일로 제공

Memo

Java

Programming Language

Chapter 03

연산자

operator

1. 연산자(operator)

연산자는 ‘연산을 수행하는 기호’를 말한다. 예를 들어 ‘+’기호는 덧셈 연산을 수행하며, ‘덧셈 연산자’라고 한다. 자바에서는 사칙연산(+, -, *, /)을 비롯해서 다양한 연산자들을 제공한다. 처음에는 배워야할 연산자가 많아 보이지만, 자주 쓰이는 것들을 중심으로 하나씩 배워나가면 쉽게 익숙해질 것이다.

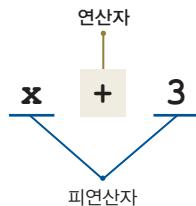
1.1 연산자와 피연산자

연산자가 연산을 수행하려면 반드시 연산의 대상이 있어야하는데, 이것을 ‘피연산자(operand)’라고 한다. 피연산자로 상수, 변수 또는 식(式) 등을 사용할 수 있다.

연산자(operator) 연산을 수행하는 기호(+,-,*,/ 등)

피연산자(operand) 연산자의 작업 대상(변수, 상수, 리터럴, 수식)

다음과 같이 ‘ $x + 3$ ’이라는 식(式)이 있을 때, ‘+’는 두 피연산자를 더해서 그 결과를 반환하는 덧셈 연산자이고, 변수 x 와 상수 3은 이 연산자의 피연산자이다.



이처럼 덧셈연산자 ‘+’는 두 값을 더한 결과를 반환하므로, 두 개의 피연산자를 필요로 한다. 대부분의 연산자는 이처럼 두 개의 피연산자를 필요로 하며, 하나 또는 세 개의 피연산자를 필요로 하는 연산자도 있다. 연산자는 피연산자로 연산을 수행하고 나면 항상 결과값을 반환한다. 예를 들어 x 의 값이 5일 때, 덧셈연산 ‘ $x + 3$ ’의 결과값은 8이 된다.

1.2 식(式)과 대입 연산자

연산자와 피연산자를 조합하여 계산하고자하는 바를 표현한 것을 ‘식(式, expression)’이라고 한다. 그리고 식을 계산하여 결과를 얻는 것을 ‘식을 평가(evaluation)한다’고 한다. 하나의 식을 평가(계산)하면, 단 하나의 결과를 얻는다. 만일 x 의 값이 5라면, 아래의 식을 평가한 결과는 23이 된다.

4 * x + 3

작성한 식을 프로그램에 포함시키려면, 식의 끝에 ‘;’를 붙여서 문장으로 만들어야 한다.

4 * x + 3; // 문장(statement)

예를 들어 변수 x의 값이 5일 때, 위의 문장은 다음과 같은 과정으로 처리된다.

```
4 * x + 3;
→ 4 * 5 + 3;
→ 23;           // 결과를 얻었지만 쓰이지 않고 사라진다.
```

식이 평가되어 23이라는 결과를 얻었지만, 이 값이 어디에도 쓰이지 않고 사라지기 때문에 이 문장은 아무런 의미가 없다. 그래서 아래와 같이 대입 연산자 '='를 사용해서 변수와 같이 값을 저장할 수 있는 공간에 결과를 저장해야 한다.

```
y = 4 * x + 3;
→ y = 4 * 5 + 3;
→ y = 23;           // 식의 평가결과가 변수 y에 저장된다.
```

그 다음에는 변수 y에 저장된 값을 다른 곳에 사용하거나 화면에 출력함으로써 의미있는 결과를 얻을 수 있다.

```
y = 4 * x + 3;
System.out.println(y); // 변수 y의 값을 화면에 출력
```

만일 식의 평가결과를 출력하기만 원할 뿐, 이 값을 다른 곳에 사용하지 않을 것이라면 다음과 같이 변수에 저장하지 않고 println메서드의 팔호() 안에 직접 식을 써도 된다.

```
System.out.println(4 * x + 3);
→ System.out.println(23); // 23이 화면에 출력된다.
```

1.3 연산자의 종류

배워야 할 연산자의 개수가 많아서 부담스러울 수 있는데, 기능이 비슷한 것들끼리 묶어놓고 보면 몇 종류 안 된다.

종류	연산자	설명
산술 연산자	+ - * / % << >>	사칙 연산(+,-,*,/.)과 나머지 연산(%)
비교 연산자	> < >= <= == !=	크고 작음과 같고 다른을 비교
논리 연산자	&& ! & ^ ~	'그리고(AND)'와 '또는(OR)'으로 조건을 연결
대입 연산자	=	우변의 값을 좌변에 저장
기 타	(type) ?: instanceof	형변환 연산자, 삼항 연산자, instanceof연산자

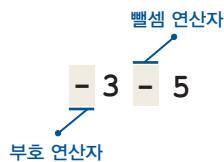
▲ 표3-1 연산자의 기능별 분류

| 참고 | (type)은 '형변환 연산자'를 의미한다.

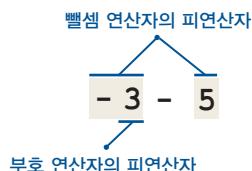
연산자는 위의 표에서 알 수 있는 것처럼, 크게 산술, 비교, 논리, 대입 4가지로 나눌 수 있다. 산술, 비교, 대입 연산자는 이미 알고 있는 것들이고, 논리 연산자도 쉽게 이해가 될 것이다.

피연산자의 개수에 의한 분류

피연산자의 개수로 연산자를 분류하기도 하는데, 피연산자의 개수가 하나면 ‘단항 연산자’, 두 개면 ‘이항 연산자’, 세 개면 ‘삼항 연산자’라고 부른다. 대부분의 연산자는 ‘이항 연산자’이고, 삼항 연산자는 오직 ‘?:’ 하나뿐이다.



위의 식에는 두 개의 연산자가 포함되어 있는데, 둘 다 같은 기호‘-’로 나타내지만 엄연히 다른 연산자이다. 왼쪽의 것은 ‘부호 연산자’이고, 오른쪽의 것은 ‘뺄셈 연산자’이다. 이처럼 서로 다른 연산자의 기호가 같은 경우도 있는데, 이럴 때는 피연산자의 개수로 구분이 가능하다.

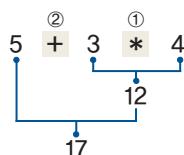


‘부호 연산자’는 단항 연산자로 피연산자가 ‘3’ 한 개뿐이지만, ‘뺄셈 연산자’는 이항 연산자로 피연산자가 ‘-3’과 ‘5’ 두 개이다.

이처럼 연산자를 기능별, 피연산자의 개수별로 나누어 분류하는 것은 곧이어 배우게 될 ‘연산자의 우선순위’ 때문이기도 하다. 연산자마다 우선순위가 다르지만, 같은 종류의 연산자들은 우선순위가 비슷하기 때문에 각 종류별로 우선순위를 외우면 기억하기 더 쉽다.

1.4 연산자의 우선순위와 결합규칙

식에 사용된 연산자가 둘 이상인 경우, 연산자의 우선순위에 의해서 연산순서가 결정된다. 곱셈과 나눗셈(*, /)은 덧셈과 뺄셈(+, -)보다 우선순위가 높다는 것은 이미 수학에서 배워서 알고 있을 것이다. 그래서 아래의 식은 ‘ $3 * 4$ ’가 먼저 계산된 다음, 그 결과에 5를 더해서 17을 결과로 얻는다.



이처럼 연산자의 우선순위는 대부분 상식적인 선에서 해결된다. 아래의 표를 통해 이 사실을 한번 확인해보자.

식	설명
$-x + 3$	단항 연산자가 이항 연산자보다 우선순위가 높다. 그래서 x 의 부호를 바꾼 다음 덧셈이 수행된다. 여기서 ‘-’는 뺄셈 연산자가 아니라 부호 연산자이다.
$x + 3 * y$	곱셈과 나눗셈이 덧셈과 뺄셈보다 우선순위가 높다. 그래서 ‘ $3 * y$ ’가 먼저 계산된다.
$x + 3 > y - 2$	비교 연산자(>)보다 산술 연산자 ‘+’와 ‘-’가 먼저 수행된다. 그래서 ‘ $x + 3$ ’과 ‘ $y - 2$ ’가 먼저 계산된 다음에 ‘>’가 수행된다.
$x > 3 \&& x < 5$	논리 연산자 ‘&&’보다 비교 연산자가 먼저 수행된다. 그래서 ‘ $x > 3$ ’와 ‘ $x < 5$ ’가 먼저 계산된 다음에 ‘&&’가 수행된다. 식의 의미는 ‘ x 가 3보다 크고 5보다 작다’이다.
<code>result = x + y * 3;</code>	대입 연산자는 연산자 중에서 제일 우선순위가 낮다. 그래서 우변의 최종 연산결과가 변수 <code>result</code> 에 저장된다.

▲ 표3-2 연산자 우선순위의 예와 설명

표3-2에서 설명을 가린 채로 왼쪽의 식만 보고 어떤 순서로 연산이 수행될지 생각해보자. 그리 어렵지 않게 연산순서를 알아낼 수 있을 것이다. 실제 프로그래밍에서 사용되는 대부분의 식은 이처럼 상식적으로 판단이 가능한 수준이다.

상식만으로 판단하기 쉽지 않은 우선순위 몇 가지를 아래의 표에 정리하였다. 아직 배우지 않은 연산자들이니까 지금은 가볍게 보고, 이 장을 다 배운 후에 다시 확인하자.

식	설명
$x \ll 2 + 1$	쉬프트 연산자(<<)는 덧셈 연산자보다 우선순위가 낮다. 그래서 원쪽의 식은 ‘ $x \ll (2 + 1)$ ’과 같다.
<code>data & 0xFF == 0</code>	비트 연산자(&)는 비교 연산자(==)보다 우선순위가 낮으므로 비교연산 후에 비트연산이 수행된다. 그래서 원쪽의 식은 ‘ <code>data & (0xFF == 0)</code> ’과 같다.
$x < -1 x > 3 \&& x < 5$	논리 연산자 중에서 AND를 의미하는 ‘&’와 ‘&&’가 OR를 의미하는 ‘ ’와 ‘ ’보다 우선순위가 높다. 이처럼 수식에 AND와 OR가 함께 사용되는 경우는 다음과 같이 괄호를 사용해서 우선순위를 명확히 하는 것이 좋다. $x < -1 (x > 3 \&& x < 5)$

▲ 표3-3 주의해야 할 연산자 우선순위의 예와 설명

우리가 알고 있는 일반적인 수학 상식과 표3-3에 정리된 내용정도면 연산자의 우선순위 문제는 해결된다. 만일 우선순위가 확실하지 않다면, 먼저 계산되어야하는 부분을 괄호로 묶어주면 된다. 괄호 안의 계산식이 먼저 계산될 것이 확실하기 때문이다.

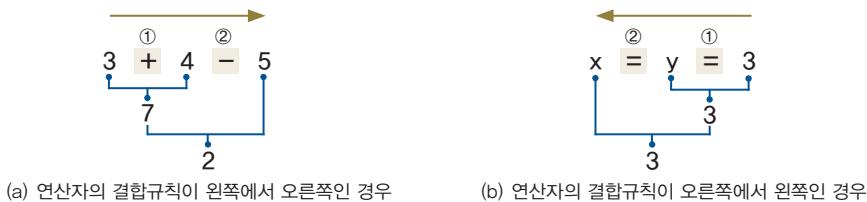
| 참고 | 괄호는 연산자가 아니다. 연산자의 우선순위를 임의로 지정할 때 사용하는 기호일 뿐이다.

연산자의 결합규칙

하나의 식에 같은 우선순위의 연산자들이 여러 개 있는 경우, 어떤 순서로 연산을 수행할까? 우선순위가 같다고 해서 아무거나 먼저 처리하는 것은 아니고 나름대로의 규칙을 가지고 있는데, 그 규칙을 ‘연산자의 결합규칙’이라고 한다.

연산자의 결합규칙은 연산자마다 다르지만, 대부분 왼쪽에서 오른쪽의 순서로 연산을 수행하고, 단항 연산자와 대입 연산자만 그 반대로, 오른쪽에서 왼쪽의 순서로, 연산을 수행한다.

그림3-1의 (a)에서 수식 ‘ $3 + 4 - 5$ ’는 덧셈연산자‘+’의 결합방향이 왼쪽에서 오른쪽이므로 수식의 왼쪽에 있는 ‘ $3 + 4$ ’를 먼저 계산하고, 그 다음에 ‘ $3 + 4$ ’의 연산결과인 7과 5의 뺄셈을 수행한다.



▲ 그림3-1 연산자의 결합규칙

그림 3-1의 (b)에서 대입 연산자는 연산자의 결합규칙이 오른쪽에서 왼쪽이므로 수식 ‘ $x = y = 3$ ’에서 오른쪽의 대입연산자부터 처리한다. 따라서 ‘ $y = 3$ ’이 먼저 수행되어서 y 에 3이 저장되고 그 다음에 ‘ $x = 3$ ’이 수행되어 x 에도 3이 저장된다.

| 참고 | ‘ $x = y = 3$;’은 ‘ $y=3$;’과 ‘ $x=3$;’을 하나의 문장으로 합쳐놓은 것으로 이해해도 좋다.

$$\begin{aligned} &x = \color{red}{y = 3} \\ \rightarrow &x = \color{red}{3} \\ \rightarrow &3 \end{aligned}$$

앞서 모든 연산자는 연산결과를 갖는다고 했는데, 대입연산자도 예외는 아니다. 대입연산자는 우변의 값을 좌변에 저장하고, 저장된 값을 연산결과로 반환한다. 그래서 ‘ $y=3$ ’의 연산결과가 3이 되는 것이다.

지금까지 배운 연산자의 우선순위에 대해서 정리하면 다음과 같다.

1. 산술 > 비교 > 논리 > 대입. 대입은 제일 마지막에 수행된다.
2. 단항(1) > 이항(2) > 삼항(3). 단항 연산자의 우선순위가 이항 연산자보다 높다.
3. 단항 연산자와 대입 연산자를 제외한 모든 연산의 진행방향은 왼쪽에서 오른쪽이다.

표3-3과 이 내용만 잘 기억하고 있어도, 대부분의 연산자 우선순위 문제는 해결할 수 있을 것이다. 그렇지 않을 때는 표3-4를 참고하자.

종 류	결합규칙	연산자	우선순위
단항 연산자	←	++ -- + - ~ ! (type)	높음
산술 연산자	→	* / %	
	→	+ -	
	→	<< >>	
비교 연산자	→	< > <= >= instanceof	
	→	== !=	
논리 연산자	→	&	
	→	^	
	→		
	→	&&	
	→		
삼항 연산자	→	? :	
대입 연산자	←	= += -= *= /= %=<<= >>= &= ^= =	낮음

▲ 표 3-4 연산자의 우선순위와 결합규칙

| 주의 | 단항연산자에 있는 '+'와 '-'는 부호연산자이고, '(type)'은 형변환 연산자이다.

| 참고 | instanceof는 객체의 타입을 확인하는데 사용되는 연산자이다. 7장 객체지향개념에서 설명한다.

시험에서는 연산자의 우선순위와 결합규칙으로 어려운 문제가 나올 수 있겠지만, 실제 프로그래밍에서는 앞서 정리한 3가지 내용과 몇 가지 예외적인 것들만 기억하는 것만으로도 충분하다.

1.5 산술 변환(usual arithmetic conversion)

이항 연산자는 두 피연산자의 타입이 일치해야 연산이 가능하므로, 피연산자의 타입이 서로 다르다면 연산 전에 형변환 연산자로 타입을 일치시켜야 한다. 예를 들어 int타입과 float타입을 덧셈하는 경우, 형변환 연산자를 사용해서 피연산자의 타입을 둘 다 int 또는 float로 일치시켜야 한다.

```
int i = 10;
float f = 20.0f;

float result = f + (float)i; // 형변환으로 두 피연산자의 타입을 일치
```

대부분의 경우, 두 피연산자의 타입 중에서 더 큰 타입으로 일치시키는데, 그 이유는 작은 타입으로 형변환하면 원래의 값이 손실될 가능성이 있기 때문이다. 앞서 배운 것과 같이 작은 타입에서 큰 타입으로 형변환하는 경우, 자동적으로 형변환되므로 형변환 연산자를 생략할 수 있다.

```
float result = f + i; // 큰 타입으로 형변환시, 형변환연산자 생략 가능
```

이처럼 연산 전에 피연산자 타입의 일치를 위해 자동 형변환되는 것을 ‘산술 변환’ 또는 ‘일반 산술 변환’이라 하며, 이 변환은 이항 연산에서만 아니라 단항 연산에서도 일어난다. ‘산술 변환’의 규칙은 다음과 같다.

- ① 두 피연산자의 타입을 같게 일치시킨다.(보다 큰 타입으로 일치)

```
long + int → long + long → long
float + int → float + float → float
double + float → double + double → double
```

- ② 피연산자의 타입이 int보다 작은 타입이면 int로 변환된다.

```
byte + short → int + int → int
char + short → int + int → int
```

| 참고 | 모든 연산에서 ‘산술 변환’이 일어나지만, 쉬프트 연산자(<<, >>)와 증감 연산자(++, --)는 예외이다.

첫 번째 규칙은 앞서 자동 형변환에서 배운 것처럼 피연산자의 값손실을 최소화하기 위한 것이고, 두 번째 규칙은 정수형의 기본 타입인 int가 가장 효율적으로 처리할 수 있는 타입이기 때문에, 그리고 int보다 작은 타입, 예를 들면 char나 short의 표현범위가 좁아서 연산중에 오버플로우(overflow)가 발생할 가능성이 높기 때문에 만들어진 것이다.

여기서 한 가지 주목해야 할 점은 연산결과의 타입이다. 연산결과의 타입은 피연산자의 타입과 일치한다. 예를 들어 int와 int의 나눗셈 연산결과는 int이다. float와 double과 같은 실수형이 아니기 때문에 소수점 이하는 버려진다. 그래서 아래의 식 ‘5 나누기 2’의 결과가 2.5가 아닌 2이다.

```
int / int → int
5 / 2 → 2
```

위의 식에서 2.5라는 실수를 결과로 얻으려면, 피연산자 중 어느 한 쪽을 float와 같은 실수형으로 형변환해야 한다. 그러면, 다른 한 쪽은 산술 변환의 첫 번째 규칙에 의해 자동적으로 형변환되어 두 피연산자 모두 실수형이 되고, 연산결과 역시 실수형의 값을 얻을 수 있다.

```
int / (float)int → int / float → float / float → float
5 / (float)2 → 5 / 2.0f → 5.0f / 2.0f → 2.5f
```

결국 산술 변환이란, 용어가 좀 거창하지만, 연산 직전에 발생하는 자동 형변환일 뿐이다. 아래의 두 가지 규칙만 잘 기억해두자.

산술 변환이란? 연산 수행 직전에 발생하는 피연산자의 자동 형변환

- ① 두 피연산자의 타입을 같게 일치시킨다(보다 큰 타입으로 일치).
- ② 피연산자의 타입이 int보다 작은 타입이면 int로 변환된다.

2. 단항 연산자

2.1 증감 연산자 ++ --

증감연산자는 피연산자에 저장된 값을 1 증가 또는 감소시킨다. 증감연산자의 피연산자로 정수와 실수가 모두 가능하지만, 상수는 값을 변경할 수 없으므로 가능하지 않다.

앞서 형변환에서 설명한 것과 같이 대부분의 연산자는 피연산자의 값을 읽어서 연산에 사용할 뿐, 피연산자의 타입이나 값을 변경시키지 않는다. 오직 대입연산자와 증감연산자만 피연산자의 값을 변경한다.

| 참고 | 증감연산자는 일반 산술 변환에 의한 자동 형변환이 발생하지 않으며, 연산결과의 타입은 피연산자의 타입과 같다.

증가 연산자(++) 피연산자의 값을 1 증가시킨다.

감소 연산자(--) 피연산자의 값을 1 감소시킨다.

일반적으로 단항 연산자는 피연산자의 왼쪽에 위치하지만, 증가 연산자 ‘++’와 감소 연산자 ‘--’는 양쪽 모두 가능하다. 피연산자의 왼쪽에 위치하면 ‘전위형(prefix)’, 오른쪽에 위치하면 ‘후위형(postfix)’이라고 한다.

전위형과 후위형 모두 피연산자의 값을 1 증가 또는 감소시키지만, 증감연산자가 수식이나 메서드 호출에 포함된 경우 전위형일 때와 후위형일 때의 결과가 다르다.

타입	설명	사용예
전위형	값이 참조되기 전에 증가시킨다.	j = ++i;
후위형	값이 참조된 후에 증가시킨다.	j = i++;

▲ 표3-5 전위형과 후위형의 비교

그러나 ‘+i;’와 ‘i++;’처럼 증감연산자가 수식이나 메서드 호출에 포함되지 않고 독립적인 하나의 문장으로 쓰인 경우에는 전위형과 후위형의 차이가 없다.

```
+i;      // i의 값을 1 증가시킨다.  
i++;    // 위의 문장과 차이가 없다.
```

▼ 예제 3-1/OperatorEx.java

```
class OperatorEx {
    public static void main(String args[]) {
        int i=5;
        i++;           // i=i+1;과 같은 의미이다. ++i;로 바꿔 써도 결과는 같다.
        System.out.println(i);

        i=5;          // 결과를 비교하기 위해 i값을 다시 5로 변경.
        ++i;
        System.out.println(i);
    }
}
```

▼ 실행결과

6
6

i의 값을 증가시킨 후 출력할 때, 한번은 전위형(`++i`)을 사용했고, 또 한 번은 후위형(`i++`)을 사용했다. 결과를 보면 두 번 모두 i의 초기값 5에서 1이 증가된 6이 출력됨을 알 수 있다.

이 경우에는 어떤 수식에 포함된 것이 아니라 단독적으로 사용된 것이기 때문에, 증감연산자(`++`)를 피연산자의 왼쪽에 사용한 경우(전위형, `++i`)와 오른쪽에 사용한 경우(후위형, `i++`)의 차이가 없다.

그러나 다른 수식에 포함되거나 메서드의 매개변수로 사용된 경우, 즉 단독으로 사용되지 않은 경우 전위형(`++i`)과 후위형(`i++`)의 결과는 다르다. 다음의 예제를 보자.

▼ 예제 3-2/OperatorEx2.java

```
class OperatorEx2 {
    public static void main(String args[]) {
        int i=5, j=0;

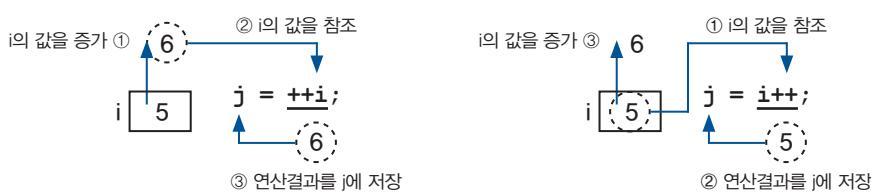
        j = i++;
        System.out.println("j=i++; 실행 후, i=" + i + ", j=" + j);

        i=5;          // 결과를 비교하기 위해, i와 j의 값을 다시 5와 0으로 변경
        j=0;

        j = ++i;
        System.out.println("j=++i; 실행 후, i=" + i + ", j=" + j);
    }
}
```

▼ 실행결과
j=i++; 실행 후, i=6, j=5
j=++i; 실행 후, i=6, j=6

실행결과를 보면 i의 값은 두 경우 모두 1이 증가되어 6이 되지만, j의 값은 그렇지 않다. 식을 계산하기 위해서는 식에 포함된 변수의 값을 읽어 와야 하는데, 전위형은 변수(피연산자)의 값을 먼저 증가시킨 후에 변수의 값을 읽어오는 반면, 후위형은 변수의 값을 먼저 읽어온 후에 값을 증가시킨다.



▲ 그림3-2 전위형과 후위형의 연산과정 비교

전위형 '`j=++i;`'에서는 i의 값을 증가시킨 후에 읽어오므로 i의 값이 5에서 6으로 증가된 후에 이 값이 j에 저장되며, 후위형 '`j=i++;`'에서는 i값인 5를 먼저 읽어온 다음에 i를 증가시키니까 j에 5가 저장된다.

증감 연산자가 포함된 식을 이해하기 어려울 때는 다음과 같이 증감 연산자를 따로 떼어내면 이해하기가 쉬워진다. 전위형의 경우 증감연산자를 식의 이전으로,

j = ++i; // 전위형

++i; // 증가 후에
j = i; // 참조하여 대입

후위형의 경우 증감연산자를 식의 이후로 떼어내면 된다.

j = i++; // 후위형

j = i; // 참조하여 대입 후에
i++; // 증가

다음은 메서드 호출에 증감연산자가 사용된 예이다.

▼ 예제 3-3/OperatorEx3.java

```
class OperatorEx3 {
    public static void main(String args[]) {
        int i=5, j=5;
        System.out.println(i++);
        System.out.println(++j);
        System.out.println("i = " + i + ", j = " + j);
    }
}
```

▼ 실행결과

5
6
i = 6, j = 6

i는 값이 증가되기 전에 참조되므로 println()에게 i에 저장된 값 5를 넘겨주고 나서 i의 값이 증가하기 때문에 5가 출력되고, j의 경우 j에 저장된 값을 증가 시킨 후에 println()에게 값을 넘겨주므로 6이 출력된다. 결과적으로는 i, j 모두 1씩 증가되어 6이 된다.

아래의 왼쪽 코드에서 증감연산자를 따로 떼어내면 오른쪽과 같은 코드가 된다.

System.out.println(i++);
System.out.println(++j);

System.out.println(i);
i++;
++j;
System.out.println(j);

증감연산자를 사용하면 코드가 간결해지지만, 지나치면 코드가 복잡해서 이해하기 어려워지기도 한다. 예를 들어 x의 값이 5일 때, 아래 식이 수행된 후의 x의 값은 얼마일까?

x = x++ - ++x; // x의 값은 -1? -2?

생각보다 쉽게 답을 내기 어려울 것이다. 실제 프로그래밍에서는 이러한 코드를 작성할 일이 없고, 이렇게 작성하는 것은 바람직하지 않다. 하나의 식에서 증감연산자의 사용을 최소화하고, 식에 두 번 이상 포함된 변수에 증감 연산자를 사용하는 것은 피해야 한다. 감소 연산자(--)는 피연산자의 값을 1 감소시킨다는 것만 빼면 증가 연산자와 동일하다.

2.2 부호 연산자 + -

부호 연산자‘-’는 피연산자의 부호를 반대로 변경한 결과를 반환한다. 피연산자가 음수면 양수, 양수면 음수가 연산의 결과가 된다. 부호연산자‘+’는 하는 일이 없으며, 쓰이는 경우도 거의 없다. 부호연산자‘-’가 있으니까 형식적으로 ‘+’를 추가해 놓은 것뿐이다.

부호 연산자는 boolean형과 char형을 제외한 기본형에만 사용할 수 있다.

| 참고 | 부호연산자는 덧셈, 뺄셈연산자와 같은 기호를 쓰지만 다른 연산자이다. 기호는 같아도 피연산자의 개수가 달라서 구별이 가능하다.

▼ 예제 3-4/OperatorEx4.java

```
class OperatorEx4 {
    public static void main(String[] args) {
        int i = -10;
        i = +i;
        System.out.println(i);

        i = -10;
        i = -i;
        System.out.println(i);
    }
}
```

▼ 실행결과

-10
10

지금까지 소개한 연산자 외에도 ‘단항 연산자’가 더 있지만, 편의상 관련된 이항연산자와 함께 놓았다. 논리부정 연산자‘!’는 5.1 논리 연산자‘에서’, 비트전환 연산자‘~’는 5.2 비트 연산자에서 설명한다.

3. 산술 연산자

산술 연산자에는 사칙 연산자(+, -, *, /)와 나머지 연산자(%)가 있다. 사칙연산은 일상생활에서 자주 사용하는 익숙한 것이라 그리 어렵지 않을 것이다. 다만 몇 가지 주의할 사항들이 있는데, 그 것들을 중심으로 설명할 것이다.

3.1 사칙 연산자 + - * /

사칙 연산자, 덧셈(+), 뺄셈(-), 곱셈(*), 나눗셈(/)은 아마도 프로그래밍에 가장 많이 사용되는 연산자들 일 것이다. 여러분들이 이미 알고 있는 것처럼 곱셈(*), 나눗셈(/), 나머지(%) 연산자가 덧셈(+), 뺄셈(-)연산자보다 우선순위가 높으므로 먼저 처리된다.

그리고 피연산자가 정수형인 경우, 나누는 수로 0을 사용할 수 없다. 만일 0으로 나눈다면, 실행 시에 에러가 발생할 것이다.

▼ 예제 3–5/OperatorEx5.java

```
class OperatorEx5 {
    public static void main(String args[]) {
        int a = 10;
        int b = 4;

        System.out.printf("%d + %d = %d%n", a, b, a + b);
        System.out.printf("%d - %d = %d%n", a, b, a - b);
        System.out.printf("%d * %d = %d%n", a, b, a * b);
        System.out.printf("%d / %d = %d%n", a, b, a / b);
        System.out.printf("%d / %f = %f%n", a, (float)b, a / (float)b);
    }
}
```

▼ 실행결과

```
10 + 4 = 14
10 - 4 = 6
10 * 4 = 40
10 / 4 = 2
10 / 4.000000 = 2.500000
```

두 변수 a와 b에 각각 10과 4를 저장하여 사칙연산을 수행하고 그 결과를 출력하는 예제이다. 한 가지 눈여겨볼 것은 10을 4로 나눈 결과가 2.5가 아닌 2라는 것이다.

int	int	int
10	/ 4	→ 2

// 소수점 이하는 버려진다.

나누기 연산자의 두 피연산자가 모두 int타입인 경우, 연산결과 역시 int타입이다. 그래서 실제 연산결과는 2.5일지라도 int타입의 값인 2를 결과로 얻는다. int타입은 소수점을 저장하지 못하므로 정수만 남고 소수점 이하는 버려지기 때문이다. 이 때, 반올림이 발생하지 않는다는 것에 주의하자.

그래서 올바른 연산결과를 얻기 위해서는 두 피연산자 중 어느 한 쪽을 실수형으로 형변환해야 한다. 그래야만 다른 한 쪽도 같이 실수형으로 자동 형변환되어 결국 실수형의 값을 결과로 얻는다.

```
int   float      float   float      float
    / 4.0f   →  10.0f / 4.0f   →  2.5f
```

위의 연산과정을 보면, 두 피연산자의 타입이 일치하지 않으므로 int타입보다 범위가 넓은 float타입으로 일치시킨 후에 연산을 수행하는 것을 알 수 있다. 이제 float타입과 float타입의 연산이므로 연산결과 역시 float타입이다.

```
System.out.println(3/0); // 실행하면, 오류(ArithmeticException) 발생!!!
System.out.println(3/0.0); // Infinity가 출력됨
```

그리고 피연산자가 정수형인 경우, 나누는 수로 0을 사용할 수 없다. 만일 0으로 나누면, 컴파일은 정상적으로 되지만 실행 시 오류(ArithmeticException)가 발생한다.

부동 소수점값인 0.0f, 0.0d로 나누는 것은 가능하지만 그 결과는 Infinity(무한대)이다. 나눗셈 연산자 '/'와 나머지 연산자 '%'의 피연산자가 무한대(Infinity) 또는 0.0인 경우의 결과를 표로 정리해 놓았다. 그리 중요한 것은 아니니까 참고만 하자.

| 참고 | NaN은 'Not a Number'를 줄인 것으로, 숫자가 아니라는 뜻이다.

x	y	x / y	x % y
유한수	±0.0	±Infinity	NaN
유한수	±Infinity	±0.0	x
±0.0	±0.0	NaN	NaN
±Infinity	유한수	±Infinity	NaN
±Infinity	±Infinity	NaN	NaN

▲ 표3-6 피연산자가 유한수가 아닌 경우의 연산결과

▼ 예제 3-6/OperatorEx6.java

```
class OperatorEx6 {
    public static void main(String[] args) {
        byte a = 10;
        byte b = 20;
        byte c = a + b; •—————
        System.out.println(c);
    }
}
```

컴파일 에러가 발생한다.
 명시적으로 형변환이 필요하다.
 byte c = (byte)(a+b);

▼ 실행결과

OperatorEx6.java:5: error: incompatible types: possible
 lossy conversion from int to byte
 byte c = a + b;
 ^
 1 error

이 예제를 컴파일하면 위와 같은 에러가 발생한다. 발생한 위치는 5번째 줄이다. a와 b는 모두 int형보다 작은 byte형이기 때문에 연산자 '+'는 이 두 개의 피연산자들의 자료형을 int형으로 변환한 다음 연산(덧셈)을 수행한다.

그래서 'a+b'의 연산결과는 byte형이 아닌 int형(4 byte)인 것이다. 4 byte의 값을 1 byte의 변수에 형변환없이 저장하려고 했기 때문에 에러가 발생하는 것이다.

크기가 작은 자료형의 변수를 큰 자료형의 변수에 저장할 때는 자동으로 형변환(type conversion, casting)되지만, 반대로 큰 자료형의 값을 작은 자료형의 변수에 저장하려면 명시적으로 형변환 연산자를 사용해서 변환해주어야 한다.

예제의 5번째 줄 'byte c = a + b;'를 'byte c = (byte)(a + b);'와 같이 변경해야 컴파일 에러가 발생하지 않는다.

▼ 예제 3-7/OperatorEx7.java

```
class OperatorEx7 {
    public static void main(String[] args) {
        byte a = 10;
        byte b = 30;
        byte c = (byte) (a * b);
        System.out.println(c);
    }
}
```

▼ 실행결과

44

이 예제를 실행하면 44가 화면에 출력된다. '10 * 30'의 결과는 300이지만, 형변환(캐스팅, casting)에서 배운 것처럼, 큰 자료형에서 작은 자료형으로 변환하면 데이터의 손실이 발생하므로 값이 바뀔 수 있다. 300은 byte형의 범위를 넘기 때문에 byte형으로 변환하면 데이터 손실이 발생하여 결국 44가 byte형 변수 c에 저장된다.

아래의 표3-7에서 알 수 있듯이 byte형(1 byte)에서 int형(4 byte)으로 변환하는 것은 2진수 8자리에서 32자리로 변환하는 것이기 때문에 자료 손실이 일어나지 않는다. 원래 8 자리는 그대로 보존하고 나머지는 모두 0으로 채운다. 음수인 경우에는 부호를 유지하기 위해 0 대신 1로 채운다.

반대로 int형을 byte형으로 변환하는 경우 앞의 24자리를 없애고 하위 8자리(1 byte)만을 보존한다. 저장된 값이 10인 경우 값이 작아서 상위 24자리를 잘라내도 원래 값을 유지하는데 지장이 없지만, byte형의 범위인 '-128~127'의 범위를 넘는 int형의 값을 byte형으로 변환하면, 원래의 값이 보존되지 않고 byte형의 범위 중 한 값을 가지게 된다. 이러한 값 손실을 예방하기 위해서는 충분히 큰 자료형을 사용해야 한다.

변환	2진수	10진수	값손실		
byte ↓ int	<table border="1"><tr><td>0 1 0 1 0</td></tr></table>	0 1 0 1 0	10 10	없음	
0 1 0 1 0					
int ↓ byte	<table border="1"><tr><td>0 1 0 1 0</td><td>0 0 1 0 1 1 0 0</td></tr></table>	0 1 0 1 0	0 0 1 0 1 1 0 0	300 44	있음
0 1 0 1 0	0 0 1 0 1 1 0 0				

▲ 표 3-7 byte와 int 간의 형변환

▼ 예제 3-8/OperatorEx8.java

```
class OperatorEx8 {
    public static void main(String args[]) {
        int a = 1_000_000;      // 1,000,000    1백만
        int b = 2_000_000;      // 2,000,000    2백만

        long c = a * b;        // a * b = 2,000,000,000,000 ?
        System.out.println(c);
    }
}
```

▼ 실행결과
-1454759936

식 ‘ $a * b$ ’의 결과 값을 담는 변수 c 의 자료형이 long타입(8 byte)이기 때문에 2×10^{12} 을 저장하기에 충분하므로 ‘2000000000000’이 출력될 것 같지만, 결과는 전혀 다른 값이 출력된다.

그 이유는 int타입과 int타입의 연산결과는 int타입이기 때문이다. ‘ $a * b$ ’의 결과가 이미 int타입의 값(-1454759936)이므로 long형으로 자동 형변환되어도 같은 변하지 않는다.

```
long c = a * b;
→ long c = 1000000 * 2000000;
→ long c = -1454759936;
```

올바른 결과를 얻으려면 아래와 같이 변수 a 또는 b 의 타입을 ‘long’으로 형변환해야 한다.

```
long c = (long)a * b;
→ long c = (long)1000000 * 2000000;
→ long c = 1000000L * 2000000;
→ long c = 1000000L * 2000000L;
→ long c = 2000000000000L;
```

▼ 예제 3-9/OperatorEx9.java

```
class OperatorEx9 {
    public static void main(String args[]) {
        long a = 1_000_000 * 1_000_000;
        long b = 1_000_000 * 1_000_000L;

        System.out.println("a=" + a);
        System.out.println("b=" + b);
    }
}
```

▼ 실행결과
a=-727379968
b=1000000000000

이 예제는 예제3-8과 비슷한 내용의 예제인데, ‘ $1000000 * 1000000$ ’의 결과가 10^{12} 임에도 불구하고, -727379968이라는 결과가 출력되었다. 그 이유는 int타입과 int타입의 연산결과는 int타입인데, 연산결과가 int타입의 최대값인 약 2×10^9 을 넘으므로 오버플로우(overflow)가 발생했기 때문이다. 이미 오버플로우가 발생한 값을 아무리 long타입의 변수에 저장을 해도 소용이 없다.

```

int      int      int
1000000 * 1000000 → -727379968 오버플로우 발생!!!

```

```

int      long      long      long      long
1000000 * 1000000L → 1000000L * 1000000L → 100000000000000L

```

그러나 ‘`1000000 * 1000000L`’은 `int`타입과 `long`타입의 연산이기 때문에 그 결과가 `long` 타입이다. `long`타입은 연산결과인 10^{12} 을 저장할 수 있는 타입이므로 올바른 결과를 얻을 수 있다.

▼ 예제 3-10/OperatorEx10.java

```

class OperatorEx10 {
    public static void main(String args[]) {
        int a = 1000000;

        int result1 = a * a / a;      // 1000000 * 1000000 / 1000000
        int result2 = a / a * a;      // 1000000 / 1000000 * 1000000

        System.out.printf("%d * %d / %d = %d%n", a, a, a, result1);
        System.out.printf("%d / %d * %d = %d%n", a, a, a, result2);
    }
}

```

▼ 실행결과

```

1000000 * 1000000 / 1000000 = -727
1000000 / 1000000 * 1000000 = 1000000

```

`1,000,000`에 `1,000,000`을 먼저 곱한 후에 나누는 것과 먼저 나눈 후에 곱하는 것의 연산 결과가 다르다는 것을 알 수 있다. 먼저 곱하는 경우 `int`의 범위를 넘어서기 때문에 예상했던 것과 다른 결과가 나왔다.

```

1000000 * 1000000 / 1000000
→ -727379968 / 1000000          오버플로우 발생!!!
→ -727

1000000 / 1000000 * 1000000
→ 1 * 1000000
→ 1000000

```

이처럼 같은 의미의 식이라도 연산의 순서에 따라서 다른 결과를 얻을 수 있다는 것에 주의하자.

▼ 예제 3-11/OperatorEx11.java

```
class OperatorEx11 {
    public static void main(String args[]) {
        char a = 'a';
        char d = 'd';
        char zero = '0';
        char two = '2';

        System.out.printf("%c - %c = %d\n", d, a, d - a); // 'd'-'a'=3
        System.out.printf("%c - %c = %d\n", two, zero, two - zero);
        System.out.printf("%c=%d\n", a, (int)a);
        System.out.printf("%c=%d\n", d, (int)d);
        System.out.printf("%c=%d\n", zero, (int)zero);
        System.out.printf("%c=%d\n", two, (int)two);
    }
}
```

▼ 실행결과

```
'd' - 'a' = 3
'2' - '0' = 2
'a'=97
'd'=100
'0'=48
'2'=50
```

사칙연산의 피연산자로 숫자뿐만 아니라 문자도 가능하다. 문자는 실제로 해당 문자의 유니코드(부호없는 정수)로 바뀌어 저장되므로 문자간의 사칙연산은 정수간의 연산과 동일하다. 주로 문자간의 뺄셈을 하는 경우가 대부분이며, 문자 '2'를 숫자로 변환하려면 다음과 같이 문자 '0'을 빼주면 된다.

$$'2' - '0' \rightarrow 50 - 48 \rightarrow 2$$

문자 '2'의 유니코드는 50이고, 문자 '0'은 48이므로, 두 문자간의 뺄셈은 2를 결과로 얻는다. 아래의 표는 유니코드의 일부인데, '0'~'9'까지의 문자가 연속적으로 배치되어 있는 것을 알 수 있다. 그렇기 때문에 해당 문자에서 '0'을 빼주면 숫자로 변환되는 것이다.

문자	코드	문자	코드	문자	코드
0	48	A	65	a	97
1	49	B	66	b	98
2	50	C	67	c	99
3	51	D	68	d	100
4	52	E	69	e	101
5	53
6	54	W	87	w	119
7	55	X	88	x	120
8	56	Y	89	y	121
9	57	Z	90	z	122

▲ 표3-8 숫자와 영문자의 유니코드

'A'~'Z'와 'a'~'z' 역시 연속적으로 배치되어 있기 때문에, 문자 'd'에서 문자 'a'를 빼면 다음과 같이 처리된다.

$$'d' - 'a' \rightarrow 100 - 97 \rightarrow 3$$

▼ 예제 3-12/OperatorEx12.java

```
class OperatorEx12 {
    public static void main(String[] args) {
        char c1 = 'a';           // c1에는 문자 'a'의 코드값인 97이 저장된다.
        char c2 = c1;            // c1에 저장되어 있는 값이 c2에 저장된다.
        char c3 = ' ';
        // c3를 공백으로 초기화 한다.

        int i = c1 + 1;          // 'a'+1 → 97+1 → 98
        c3 = (char) (c1 + 1);   •————— 덧셈연산 c1+1의 결과가 int이므로
        c2++;
        c2++;

        System.out.println("i=" + i);
        System.out.println("c2=" + c2);
        System.out.println("c3=" + c3);
    }
}
```

▼ 실행결과
i=98
c2=c
c3=b

'c1+1'을 계산할 때, c1이 char형이므로 int형으로 변환한 후 덧셈연산을 수행하게 된다. c1에 저장되어 있는 코드값이 변환되어 int형 값이 되는 것이다. 따라서 'c1+1'은 '97+1'이 되고 결과적으로 int형 변수 i에는 98이 저장된다.

'c2++'은 형변환없이 c2에 저장되어 있는 값을 1 증가시키므로, 예제에서는 원래 저장되어 있던 값인 97이 1씩 두 번 증가되어 99가 된다. 코드값이 10진수로 99인 문자는 'c'이다. 따라서 c2를 출력하면, 'c'가 화면에 나타나는 것이다.

| 참고 | c2++; 대신에 c2=c2+1;을 사용하면 에러가 발생할 것이다. c2+1의 연산결과는 int형이며, 그 결과를 다시 c2에 담으려면 형변환 연산자를 사용하여 char형으로 형변환해야 하기 때문이다.

▼ 예제 3-13/OperatorEx13.java

```
class OperatorEx13 {
    public static void main(String[] args) {
        char c1 = 'a';

        //     char c2 = c1+1;           // 라인 5 : 컴파일 에러발생!!!
        //     char c2 = 'a'+1;          // 라인 6 : 컴파일 에러없음

        System.out.println(c2);
    }
}
```

▼ 실행결과
b

이 예제를 컴파일 하면 오류가 발생하지 않고 실행도 올바른 결과를 얻는다. 덧셈 연산자와 같은 이항 연산자는 int보다 작은 타입의 피연산자를 int로 자동 형변환한다고 배웠는데, 어째서 아래의 코드처럼 형변환을 해주지 않고도 문제가 없는 것일까?

```
char c2 = (char) ('a'+1);
```

그것은 바로 ‘a’+1이 리터럴 간의 연산이기 때문이다. 상수 또는 리터럴 간의 연산은 실행 과정동안 변하는 값이 아니기 때문에, 컴파일 시에 컴파일러가 계산해서 그 결과로 대체 함으로써 코드를 보다 효율적으로 만든다.

표3-9에서 알 수 있듯이 컴파일러가 미리 덧셈연산을 수행하기 때문에 실행 시에는 덧셈 연산이 수행되지 않는다. 그저 덧셈연산결과인 문자 ‘b’를 변수 c2에 저장할 뿐이다.

컴파일 전의 코드	컴파일 후의 코드
char c2 = 'a'+1; int sec = 60 * 60 * 24;	char c2 = 'b'; int sec = 86400;

▲ 표3-9 컴파일러에 의해서 최적화된 코드의 비교

그러나 라인 5와 같이 수식에 변수가 들어가 있는 경우에는 컴파일러가 미리 계산을 할 수 없기 때문에 아래의 오른쪽 코드와 같이 형변환을 해주어야 한다. 그렇지 않으면 컴파일 에러가 발생한다.



일부러 뺀 한 리터럴 연산을 풀어쓸 필요는 없지만, 코드의 가독성과 유지보수를 위해서 그렇게 하는 경우가 있다. 표3-9에서 int타입의 변수 sec에 하루(day)를 초(秒, second) 단위로 변환한 값을 저장하는 코드를 보면, ‘86400’이라는 값보다는 ‘60*60*24’와 같이 적어주는 것이 이해하기도 쉽고 오류가 발생할 여지가 적다. 나중에 반나절(12시간)로 값을 변경해야한다면 계산할 필요없이 ‘60*60*12’로 변경하면 되기 때문이다. 이렇게 풀어 써도 결국 컴파일러에 의해서 미리 계산되기 때문에 실행 시의 성능차이는 없다.

▼ 예제 3-14/OperatorEx14.java

```

class OperatorEx14 {
    public static void main(String[] args) {
        char c = 'a';
        for(int i=0; i<26; i++) {      // 블럭{} 안의 문장을 26번을 반복한다.
            System.out.print(c++);
        }
        System.out.println(); // 줄바꿈을 한다.

        c = 'A';
        for(int i=0; i<26; i++) {      // 블럭{} 안의 문장을 26번을 반복한다.
            System.out.print(c++);
        }
        System.out.println();

        c='0';
        for(int i=0; i<10; i++) {     // 블럭{} 안의 문장을 10번을 반복한다.
            System.out.print(c++);
        }
    }
}
  
```

```

    }
    System.out.println();
}
}

```

▼ 실행결과

abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
0123456789

| 참고 | println메서드는 값을 출력하고 줄을 바꾸지만, print메서드는 줄을 바꾸지 않고 출력한다. 매개변수없이 println메서드를 호출하면, 아무 것도 출력하지 않고 단순히 줄을 바꾸고 다음 줄의 처음으로 출력위치를 이동시킨다.

위의 예제를 실행하면, 문자 a부터 시작해서 26개의 문자를 출력하고, 또 문자 A부터 시작해서 26개의 문자, 0부터 9까지 10개의 문자를 출력한다. 소문자 a부터 z까지, 그리고 대문자 A부터 Z까지, 숫자 0부터 9까지 연속적으로 코드가 지정되어 있기 때문에 이런 결과가 나타난다.

문자 a의 코드값은 10진수로 97, b의 코드값은 98, c의 코드값은 99, ..., z의 코드값은 122이며, 문자 A의 코드값은 10진수로 65, B의 코드값은 66, C의 코드값은 67, ..., Z의 코드값은 90이다. 그리고 문자 0의 코드값은 10진수로 48이다.

이 사실을 이용하면 대문자를 소문자로 소문자를 대문자로 변환하는 프로그램을 작성할 수 있다.

| 참고 | 대문자와 소문자 간의 코드값 차이는 10진수로 32이다.

▼ 예제 3-15/OperatorEx15.java

```

class OperatorEx15 {
    public static void main(String[] args) {
        char lowerCase = 'a';
        char upperCase = (char)(lowerCase - 32);
        System.out.println(upperCase);
    }
}

```

▼ 실행결과

A

소문자를 대문자로 변경하려면, 대문자 A가 소문자 a보다 코드값이 32가 적으므로 소문자 a의 코드값에서 32를 빼면 되고, 반대로 대문자를 소문자로 변환하려면 대문자의 코드값에 32를 더해주면 된다.

| 참고 | char형과 int형 간의 뺄셈연산 결과는 int형이므로, 연산 후 char형으로 다시 형변환해야 한다는 것을 잊지 말자.

▼ 예제 3-16/OperatorEx16.java

```

class OperatorEx16 {
    public static void main(String[] args) {
        float pi = 3.141592f;
        float shortPi = (int)(pi * 1000) / 1000f;
        System.out.println(shortPi);
    }
}

```

▼ 실행결과

3.141

int형 간의 나눗셈 ‘int / int’를 수행하면 결과가 float나 double이 아닌 int임에 주의하라. 그리고 나눗셈의 결과를 반올림을 하는 것이 아니라 버린다는 점도 꼭 기억하자. 예를 들어 ‘3 / 2’의 결과는 1.5 또는 2가 아니라 1이다.

이 예제는 나눗셈 연산자의 이러한 성질을 이용해서 실수형 변수 pi의 값을 소수점 넷째 자리까지만 빼내는 방법을 보여 준다.

```
(int) (pi * 1000) / 1000f;
```

위의 수식에서 제일 먼저 수행되는 것은 괄호 안의 ‘pi * 1000’이다. pi가 float이고 1000이 정수형이니까 연산의 결과는 float인 3141.592f가 된다.

```
(int) (3141.592f) / 1000f;
```

그 다음으로는 단항연산자인 형변환 연산자의 형변환이 수행된다. 3141.592f를 int로 변환하면 3141을 얻는다. 소수점 이하는 반올림 없이 버려진다.

```
3141 / 1000f;
```

int와 float의 연산이므로, int가 float로 변환된 다음, float와 float의 연산이 수행된다.

```
3141.0f / 1000f → 3.141f
```

float와 float의 나눗셈이므로 결과는 float인 3.141f가 된다.

| 참고 | 1000f는 1000.0f와 같다.

그렇다면 버림이 아닌 반올림이 되도록 하려면 어떻게 해야 할까? 다음 예제가 그 방법을 알려준다.

▼ 예제 3-17/OperatorEx17.java

```
class OperatorEx17 {
    public static void main(String args[]) {
        double pi = 3.141592;
        double shortPi = (int) (pi * 1000 + 0.5) / 1000.0;

        System.out.println(shortPi);
    }
}
```

▼ 실행결과

3.142

이 예제는 소수점 넷째자리에서 반올림하는 방법을 보여준다. 이전 예제와 다른 점은 반올림을 위해 0.5를 더해 준다는 것이다.

```
(int) (pi * 1000 + 0.5) / 1000.0
```

위의 수식에서 제일 먼저 수행되는 것은 팔호 안의 ‘`pi * 1000`’이다. `pi`가 `double`이고 1000이 정수형이니까 연산의 결과는 `double`인 3141.592가 된다. 그리고 여기에 0.5를 더하면 3142.092가 된다.

```
(int) (3141.592 + 0.5) / 1000.0
→ (int) (3142.092) / 1000.0
```

그 다음엔 형변환 연산자에 의해서 형변환되어 3142.092가 3142가 된다.

```
3142 / 1000.0
```

`int`과 `double`의 연산이므로, `int`가 `double`로 변환된 다음, `double`과 `double`의 연산이 수행된다.

```
3142.0 / 1000.0 → 3.142
```

`double`과 `double`의 나눗셈이므로 결과는 `double`인 3.142가 된다. 만일 1000.0이 아닌 1000으로 나누었다면, 3.142가 아닌 3을 결과로 얻었을 것이다.

`Math.round()`를 사용하면 좀 더 간단히 반올림할 수 있다. 다음의 예제는 이 메서드의 사용법을 알려준다.

▼ 예제 3-18/OperatorEx18.java

```
class OperatorEx18 {
    public static void main(String args[]) {
        double pi = 3.141592;
        double shortPi = Math.round(pi * 1000) / 1000.0;
        System.out.println(shortPi);
    }
}
```

▼ 실행결과
3.142

이 예제의 결과는 `pi`의 값을 소수점 넷째 자리인 5에서 반올림을 해서 3.142가 출력되었다. `round`메서드는 매개변수로 받은 값을 소수점 첫째자리에서 반올림을 하고 그 결과를 정수로 돌려주는 메서드이다. 그래서 `Math.round(3141.592)`의 결과는 3142이다.

```
Math.round(pi * 1000) / 1000.0
→ Math.round(3.141592 * 1000) / 1000.0
→ Math.round(3141.592) / 1000.0
→ 3142 / 1000.0
→ 3.142
```

3.2 나머지 연산자 %

나머지 연산자는 왼쪽의 피연산자를 오른쪽 피연산자로 나누고 낸 나머지 값을 결과로 반환하는 연산자이다. 그리고 나눗셈에서처럼 나누는 수(오른쪽 피연산자)로 0을 사용할 수 없다는 점에 주의하자. 나머지 연산자는 주로 짹수, 홀수 또는 배수 검사 등에 주로 사용된다.

▼ 예제 3-19/OperatorEx19.java

```
class OperatorEx19 {
    public static void main(String args[]) {
        int x = 10;
        int y = 8;

        System.out.printf("%d을 %d로 나누면, %n", x, y);
        System.out.printf("몫은 %d이고, 나머지는 %d입니다.%n", x / y, x % y);
    }
}
```

▼ 실행결과

```
10을 8로 나누면,
몫은 10이고, 나머지는 2입니다.
```

나눗셈 연산자와 나머지 연산자를 이용해서 몫과 나머지를 구하는 예제이다. 간단한 예제라서 따로 설명하지 않아도 이해하는 데 어려움이 없을 것이다.

▼ 예제 3-20/OperatorEx20.java

```
class OperatorEx20 {
    public static void main(String[] args) {
        System.out.println(-10%8);
        System.out.println(10%-8);
        System.out.println(-10%-8);
    }
}
```

▼ 실행결과

```
-2
2
-2
```

나머지 연산자(%)는 나누는 수로 음수도 허용한다. 그러나 부호는 무시되므로 결과는 음수의 절대값으로 나눈 나머지와 결과가 같다.

```
System.out.println(10 % 8); // 10을 8로 나눈 나머지 2가 출력된다.
System.out.println(10% -8); // 위와 같은 결과를 얻는다.
```

그냥 피연산자의 부호를 모두 무시하고, 나머지 연산을 한 결과에 왼쪽 피연산자(나눠지는 수)의 부호를 붙이면 된다.

4. 비교 연산자

비교 연산자는 두 피연산자를 비교하는 데 사용되는 연산자다. 주로 조건문과 반복문의 조건식에 사용되며, 연산결과는 오직 true와 false 둘 중의 하나이다.

비교 연산자 역시 이항 연산자이므로 비교하는 피연산자의 타입이 서로 다를 경우에는 자료형의 범위가 큰 쪽으로 자동 형변환하여 피연산자의 타입을 일치시킨 후에 비교한다는 점에 주의하자.

4.1 대소비교 연산자 < > <= >=

두 피연산자의 값의 크기를 비교하는 연산자이다. 참이면 true를, 거짓이면 false를 결과로 반환한다. 기본형 중에서는 boolean형을 제외한 나머지 자료형에 다 사용할 수 있지만 참조형에는 사용할 수 없다.

비교연산자	연산결과
>	좌변 값이 크면, true 아니면 false
<	좌변 값이 작으면, true 아니면 false
>=	좌변 값이 크거나 같으면, true 아니면 false
<=	좌변 값이 작거나 같으면, true 아니면 false

▲ 표 3-10 대소비교 연산자의 종류와 연산결과

4.2 등가비교 연산자 == !=

두 피연산자의 값이 같은지 또는 다른지를 비교하는 연산자이다. 대소비교 연산자(<, >, <=, >=)와는 달리, 기본형은 물론 참조형, 즉 모든 자료형에 사용할 수 있다. 기본형의 경우 변수에 저장되어 있는 값이 같은지를 알 수 있고, 참조형의 경우 객체의 주소값을 저장하기 때문에 두 개의 피연산자(참조변수)가 같은 객체를 가리키고 있는지를 알 수 있다.

기본형과 참조형은 서로 형변환이 가능하지 않기 때문에 등가비교 연산자(==, !=)로 기본형과 참조형을 비교할 수 없다.

비교연산자	연산결과
==	두 값이 같으면, true 아니면 false
!=	두 값이 다르면, true 아니면 false

▲ 표 3-11 등가비교 연산자의 종류와 연산결과

비교연산자는 수학기호와 유사한 기호와 의미를 가지고 있으므로 이해하는데 별 어려움이 없을 것이다. 한 가지 다른 점은 ‘두 값이 같다’는 의미로 ‘=’가 아닌 ‘==’를 사용한다는 것인데, ‘=’는 이미 배운 것과 같이 변수에 값을 저장할 때 사용하는 ‘대입연산자’이기 때문에 ‘==’로 두 값이 같은지 비교하는 연산자를 표현한다.

|주의 | '>=와 같이 두 개의 기호로 이루어진 연산자는 '='와 같이 기호의 순서를 바꾸거나 '>=와 같이 중간에 공백이 들어가면 안 된다.

▼ 예제 3-21/OperatorEx21.java

```
class OperatorEx21 {
    public static void main(String args[]) {
        System.out.printf("10 == 10.0f \t %b%n", 10==10.0f);
        System.out.printf("'0'== 0 \t %b%n", '0'== 0);
        System.out.printf("'A'== 65 \t %b%n", 'A'== 65);
        System.out.printf("'A' > 'B' \t %b%n", 'A' > 'B');
        System.out.printf("'A'+1 != 'B' \t %b%n", 'A'+1 != 'B');
    }
}
```

▼ 실행결과

10 == 10.0f	true
'0'== 0	false
'A'== 65	true
'A' > 'B'	false
'A'+1 != 'B'	false

비교 연산자도 이항 연산자이므로 연산을 수행하기 전에 형변환을 통해 두 피연산자의 타입을 같게 맞춘 다음 피연산자를 비교한다. 10==10.0f에서 10은 int타입이고 10.0f는 float타입이므로, 10을 float로 변환한 다음에 비교한다. 두 값이 10.0f로 같으므로 결과로 true를 얻게 된다.

```
10 == 10.0f
→ 10.0f == 10.0f
→ true
```

문자 'A'의 유니코드는 10진수로 65이고, 'B'는 66, '0'은 48이므로 나머지 식들은 다음과 같은 과정으로 연산된다.

```
'0' == 0 → 48 == 0 → false
'A' == 65 → 65 == 65 → true

'A' > 'B' → 65 > 66 → false
'A'+1 != 'B' → 65+1 != 66 → 66 != 66 → false
```

▼ 예제 3-22/OperatorEx22.java

```
class OperatorEx22 {
    public static void main(String args[]) {
        float f = 0.1f;
        double d = 0.1;
        double d2 = (double)f;

        System.out.printf("10.0==10.0f %b%n", 10.0==10.0f);
        System.out.printf("0.1==0.1f %b%n", 0.1==0.1f);
        System.out.printf("f =%19.17f%n", f);
        System.out.printf("d =%19.17f%n", d);
        System.out.printf("d2=%19.17f%n", d2);
    }
}
```

```
System.out.printf("d==f %b%n", d==f);
System.out.printf("d==d2 %b%n", d==d2);
System.out.printf("d2==f %b%n", d2==f);
System.out.printf("(float)d==f %b%n",
                  (float)d==f);
}
}
```

▼ 실행결과

10.0==10.0f true
0.1==0.1f false
f =0.10000000149011612
d =0.10000000000000000
d2=0.10000000149011612
d==f false
d==d2 false
d2==f true
(float)d==f true

이 예제의 결과를 보고 다소 혼란스러울 것이다. ‘10.0==10.0f’는 true인데 ‘0.1==0.1f’는 false라니 이해하기 어렵다. 왜 이런 결과를 얻는 것일까? 그것은 정수형과 달리 실수형은 근사값으로 저장되므로 오차가 발생할 수 있기 때문이다.

10.0f는 오차없이 저장할 수 있는 값이라서 double로 형변환해도 그대로 10.0이 되지만, 0.1f는 저장할 때 2진수로 변환하는 과정에서 오차가 발생한다. double타입의 상수인 0.1도 저장되는 과정에서 오차가 발생하지만, float타입의 리터럴인 0.1f보다 적은 오차로 저장된다.

float f = 0.1f; // f에 0.10000000149011612로 저장된다.
double d = 0.1; // d에 0.10000000000000001로 저장된다.

이미 앞서 배운 것처럼 float타입의 값을 double타입으로 형변환하면, 부호와 지수는 달라지지 않고 그저 가수의 빈자리를 0으로 채울 뿐이므로 0.1f를 double타입으로 형변환해도 그 값은 전혀 달라지지 않는다. 즉, float타입의 값을 정밀도가 더 높은 double타입으로 형변환했다고 해서 오차가 적어지는 것이 아니라는 얘기다.

| 참고 | 그림3-3에서 지수부의 값도 달라진 것처럼 보이지만, float타입과 double타입의 기저(bias)의 차이에 의한 것일 뿐 달라지지 않았다.



▲ 그림 3-3 float타입의 값을 double타입으로 변환

그래서 식 `'d'=='f'`가 연산되는 과정을 단계별로 살펴보면 다음과 같다. 최종결과는 `false`이다. 변수 `f`를 `double`타입으로 형변환해도 값이 변하지 않았음에 주목하자.

```
d == f
→ d ==(double)f
→ 0.1000000000000001 == (double)0.1000000149011612
→ 0.1000000000000001 == 0.1000000149011612
→ false
```

마찬가지로 변수 d2에 변수 f의 값을 double로 형변환해서 저장해도, 직전에 설명한 것과 같이 f의 값이 그대로 d2에 저장된다. 그래서 'd2==f'의 결과가 true가 되는 것이다.

```
double d2 = (double)f;
→ double d2 = (double)0.10000000149011612;
→ double d2 = 0.10000000149011612;

d2 == f
→ 0.10000000149011612 == 0.10000000149011612
→ true
```

그러면 float타입의 값과 double타입의 값을 비교하려면 어떻게 해야 하는 걸까? double 타입의 값을 float타입으로 형변환한 다음에 비교해야 한다. 그래야만 올바른 결과를 얻을 수 있다. 또는, 어느 정도의 오차는 무시하고 두 타입의 값을 앞에서 몇 자리만 잘라서 비교할 수도 있다.

```
(float)d == f
→ (float)0.1000000000000001 == 0.10000000149011612
→ 0.10000000149011612 == 0.10000000149011612
→ true
```

문자열의 비교

두 문자열을 비교할 때는, 비교 연산자 '==' 대신 equals()라는 메서드를 사용해야 한다. 비교 연산자는 두 문자열이 완전히 같은 것인지 비교할 뿐이므로, 문자열의 내용이 같은지 비교하기 위해서는 equals()를 사용하는 것이다. equals()는 비교하는 두 문자열이 같으면 true를, 다르면 false를 반환한다.

```
String str = new String("abc");

// equals()는 두 문자열의 내용이 같으면 true, 다르면 false
boolean result = str.equals("abc"); // 내용이 같으므로 result에 true가 저장됨
```

원래 String은 클래스이므로, 아래와 같이 new를 사용해서 객체를 생성해야 한다.

```
String str = new String("abc"); // String클래스의 객체를 생성
String str = "abc";           // 위의 문장을 간단히 표현
```

그러나 특별히 String만 new를 사용하지 않고, 위와 같이 간단히 쓸 수 있게 허용한다. 위 두 문장은 거의 같지만, 한 가지 차이점이 있는데, 이에 대해서는 '9장 java.lang 패키지와 유용한 클래스'에서 설명한다. 지금은 문자열을 비교할 때 비교 연산자가 아니라 equals()를 사용해야 한다는 것만 알면 된다.

▼ 예제 3-23/OperatorEx23.java

```
class OperatorEx23 {
    public static void main(String[] args) {
        String str1 = "abc";
        String str2 = new String("abc");

        System.out.printf("\\"abc\\"==\\"abc\\" ? %b%n", "abc"=="abc");
        System.out.printf(" str1==\\"abc\\" ? %b%n", str1=="abc");
        System.out.printf(" str2==\\"abc\\" ? %b%n", str2=="abc");
        System.out.printf("str1.equals(\"abc\") ? %b%n",
                           str1.equals("abc"));
        System.out.printf("str2.equals(\"abc\") ? %b%n",
                           str2.equals("abc"));
        System.out.printf("str2.equals(\"ABC\") ? %b%n",
                           str2.equals("ABC"));
        System.out.printf("str2.equalsIgnoreCase(\"ABC\") ? %b%n",
                           str2.equalsIgnoreCase("ABC"));
    }
}
```

▼ 실행결과

```
"abc"=="abc" ? true
str1=="abc" ? true
str2=="abc" ? false
str1.equals("abc") ? true
str2.equals("abc") ? true
str2.equals("ABC") ? false
str2.equalsIgnoreCase("ABC") ? true
```

str2와 “abc”의 내용이 같은데도 ‘==’로 비교하면, false를 결과로 얻는다. 내용은 같지만 서로 다른 객체라서 그렇다. 그러나 equals()는 객체가 달라도 내용이 같으면 true를 반환한다. 그래서 문자열을 비교할 때는 항상 equals()를 사용해야 한다는 것을 기억하자.

만일 대소문자를 구별하지 않고 비교하고 싶으면, equals() 대신 equalsIgnoreCase()를 사용하면 된다.

5. 논리 연산자

‘x가 4보다 작다’라는 조건은 비교연산자를 써서 ‘ $x < 4$ ’와 같이 표현할 수 있다. 그러면, x가 4보다 작거나 또는 10보다 크다’와 같이 두 개의 조건이 결합된 경우는 어떻게 표현해야 할까? 이 때 사용하는 것이 ‘논리 연산자’이다. 논리 연산자는 둘 이상의 조건을 ‘그리고(AND)’나 ‘또는(OR)’으로 연결하여 하나의 식으로 표현할 수 있게 해준다.

5.1 논리 연산자 – `&&`, `||`, `!`

논리 연산자 ‘`&&`’은 우리말로 ‘그리고(AND)’에 해당하며, 두 피연산자가 모두 true일 때만 true를 결과로 얻는다. ‘`||`’은 ‘또는(OR)’에 해당하며, 두 피연산자 중 어느 한 쪽만 true이어도 true를 결과로 얻는다. 그리고 논리 연산자는 피연산자로 boolean형 또는 boolean형 값을 결과로 하는 조건식만을 허용한다.

 (OR결합)	피연산자 중 어느 한 쪽만 true이면 true를 결과로 얻는다.
&& (AND결합)	피연산자 양쪽 모두 true이어야 true를 결과로 얻는다.

| 참고 | ‘|’는 한글 키보드에서 Enter키의 바로 위에 있다.

논리 연산자의 피연산자가 ‘참(true)인 경우’와 ‘거짓(false)인 경우’의 연산결과를 표로 나타내면 다음과 같다.

x	y	$x \parallel y$	$x \&\& y$
true	true	true	true
true	false	true	false
false	true	true	false
false	false	false	false

▲ 표 3-12 논리 연산자의 연산결과

이제 자주 사용될만한 몇 가지 예를 통해서 논리연산자가 실제로 어떻게 사용되고, 주의해야 할 점은 어떤 것들이 있는지 살펴보자.

① x는 10보다 크고, 20보다 작다.

‘ $x > 10$ ’와 ‘ $x < 20$ ’가 ‘그리고(and)’로 연결된 조건이므로 다음과 같이 쓸 수 있다.

$x > 10 \&\& x < 20$

‘ $x > 10$ ’는 ‘ $10 < x$ ’와 같으므로 다음과 같이 쓸 수도 있다. 보통은 변수를 왼쪽에 쓰지만 이런 경우 가독성측면에서 보면 아래의 식이 더 나을 수 있다.

$10 < x \&\& x < 20$

그렇다고 해서 위의 식에서 논리연산자를 생략하고 ‘ $10 < x < 20$ ’과 같이 표현하는 것은 허용되지 않는다.

② i는 2의 배수 또는 3의 배수이다.

어떤 수가 2의 배수라는 얘기는 2로 나누었을 때 나머지가 0이라는 뜻이다. 그래서 나머지연산의 결과가 0인지 확인하면 된다. ‘또는’으로 두 조건이 연결되었으므로 논리 연산자 ‘ $\mid\mid$ ’(OR)를 사용해야 한다.

```
i%2 == 0 || i%3 == 0
```

i의 값이 8일 때, 위의 식은 다음과 같은 과정으로 연산된다.

```
i%2 == 0 || i%3 == 0
→ 8%2 == 0 || 8%3 == 0
→ 0 == 0 || 2 == 0
→ true || false
→ true
```

③ i는 2의 배수 또는 3의 배수지만 6의 배수는 아니다.

이전 조건에 6의 배수를 제외하는 조건이 더 붙었다. 6의 배수가 아니어야 한다는 조건은 ‘ $i \% 6 != 0$ ’이고, 이 조건을 ‘ $\&\&$ (AND)’로 연결해야 한다.

```
( i%2==0 || i%3==0 ) && i%6!=0
```

위의 식에 괄호를 사용한 이유는 ‘ $\&\&$ ’가 ‘ $\mid\mid$ ’보다 우선순위가 높기 때문이다. 만일 괄호를 사용하지 않으면 ‘ $\&\&$ ’를 먼저 연산한다. 다음의 두 식은 동일하다.

```
i%2==0 || i%3==0 && i%6!=0
i%2==0 || (i%3==0 && i%6!=0)
```

이처럼 하나의 식에 ‘ $\&\&$ ’와 ‘ $\mid\mid$ ’가 같이 포함된 경우, ‘ $\&\&$ ’가 먼저 연산되어야 하는 경우라도 괄호를 사용해서 우선순위를 명확히 해주는 것이 좋다.

④ 문자 ch는 숫자('0'~'9')이다.

사용자로부터 입력된 문자가 숫자('0'~'9')인지 확인하는 식은 다음과 같이 쓸 수 있다.

```
'0' <= ch && ch <= '9'
```

유니코드에서 문자 ‘0’부터 ‘9’까지 연속적으로 배치되어 있기 때문에 가능한 식이다. 문자 ‘0’부터 ‘9’까지 유니코드는 10진수로 다음과 같다.

문자	'0'	'1'	'2'	'3'	'4'	'5'	'6'	'7'	'8'	'9'
문자코드	48	49	50	51	52	53	54	55	56	57

그래서 ch의 값이 '5'인 경우 위의 식은 다음과 같은 과정으로 연산된다.

```
'0' <= ch && ch <= '9'
→ '0' <= '5' && '5' <= '9'
→ 48 <= 53 && 53 <= 57
→      true && true
→          true
```

⑤ 문자 ch는 대문자 또는 소문자이다.

④의 경우와 마찬가지로 문자 'a'부터 'z'까지, 그리고 'A'부터 'Z'까지도 연속적으로 배치되어 있으므로 문자 ch가 대문자 또는 소문자인지 확인하는 식은 다음과 같이 쓸 수 있다.

```
('a' <= ch && ch <= 'z') || ('A' <= ch && ch <= 'Z')
```

이제 예제를 통해서 직접 확인해보자.

▼ 예제 3-24/OperatorEx24.java

```
class OperatorEx24 {
    public static void main(String args[]) {
        int x = 0;
        char ch = ' ';

        x = 15;
        System.out.printf("x=%2d, 10 < x && x < 20 =%b%n", x,
                           10 < x && x < 20);
        x = 6;
        System.out.printf("x=%2d, x%2==0 || x%3==0 && x%6!=0 =%b%n",
                           x, x%2==0 || x%3==0 && x%6!=0);
        System.out.printf("x=%2d, (x%2==0 || x%3==0) && x%6!=0 =%b%n",
                           x, (x%2==0 || x%3==0) && x%6!=0);
        ch='1';
        System.out.printf("ch='%c', '0' <= ch && ch <= '9' =%b%n", ch,
                           '0' <= ch && ch <= '9');
        ch='a';
        System.out.printf("ch='%c', 'a' <= ch && ch <= 'z' =%b%n", ch,
                           'a' <= ch && ch <= 'z');
        ch='A';
        System.out.printf("ch='%c', 'A' <= ch && ch <= 'Z' =%b%n", ch,
                           'A' <= ch && ch <= 'Z');
        ch='q';
        System.out.printf("ch='%c', ch=='q' || ch=='Q' =%b%n", ch,
                           ch=='q' || ch=='Q');
    }
}
```

▼ 실행결과

```
x=15, 10 < x && x < 20 =true
x= 6, x%2==0 || x%3==0 && x%6!=0 =true
x= 6, (x%2==0 || x%3==0) && x%6!=0 =false
ch='1', '0' <= ch && ch <= '9' =true
ch='a', 'a' <= ch && ch <= 'z' =true
ch='A', 'A' <= ch && ch <= 'Z' =true
ch='q', ch=='q' || ch=='Q' =true
```

지금까지 논리 연산자에 대해 배운 내용들을 확인할 수 있는 간단한 예제이다. 변수의 값과 조건식을 다양하게 변경해가면서 실행하여, 결과가 예측과 일치하는지 확인해 보자.

▼ 예제 3-25/OperatorEx25.java

```
import java.util.*; // Scanner클래스를 사용하기 위해 추가

class OperatorEx25 {
    public static void main(String args[]) {
        Scanner scanner = new Scanner(System.in);
        char ch = ' ';

        System.out.print("문자를 하나 입력하세요.>");
        String input = scanner.nextLine();
        ch = input.charAt(0);

        if('0'<= ch && ch <= '9') {
            System.out.println("입력하신 문자는 숫자입니다.");
        }

        if(('a'<= ch && ch <= 'z') || ('A'<= ch && ch <= 'Z')) {
            System.out.println("입력하신 문자는 영문자입니다.");
        }
    } // main
}
```

▼ 실행결과 1

문자를 하나 입력하세요.>7
입력하신 문자는 숫자입니다.

▼ 실행결과 2

문자를 하나 입력하세요.>a
입력하신 문자는 영문자입니다.

이 예제는 사용자로부터 하나의 문자를 입력받아서 숫자인지 영문자인지 확인한다. 조건문 if는 팔호()안의 연산결과가 참인 경우 블럭{}내의 문장을 수행한다. 그래서 아래의 코드는 '0'<=ch && ch <='9'가 참일 때, 화면에 ‘입력하신 문자는 숫자입니다.’라고 출력한다.

```
if('0'<= ch && ch <= '9') {
    System.out.printf("입력하신 문자는 숫자입니다.");
}
```

조건문 if에 대해서는 다음 장에서 자세히 배울 것이므로 가볍게 보고 넘어가자.

효율적인 연산(short circuit evaluation)

논리 연산자의 또 다른 특징은 효율적인 연산을 한다는 것이다. OR연산 ‘||’의 경우, 두 피연산자 중 어느 한 쪽만 ‘참’이어도 전체 연산결과가 ‘참’이므로 좌측 피연산자가 ‘true(참)’이면, 우측 피연산자의 값은 평가하지 않는다.

x	y	$x \parallel y$
true	true	true
true	false	true
false	true	true
false	false	false

‘x가 true이면, $x \parallel y$ 는 항상 true이다.’

AND연산 ‘&&’의 경우도 마찬가지로 어느 한쪽만 ‘거짓(false)’이어도 전체 연산결과가 ‘거짓(false)’이므로 좌측 피연산자가 ‘거짓(false)’이면, 우측 피연산자는 평가하지 않는다.

x	y	$x \&\& y$
true	true	true
true	false	false
false	true	false
false	false	false

‘x가 false이면, $x \&\& y$ 는 항상 false이다.’

그래서 같은 조건식이라도 피연산자의 위치에 따라서 연산속도가 달라질 수 있는 것이다. OR연산 ‘||’의 경우에는 연산결과가 ‘참’일 확률이 높은 피연산자를 연산자의 왼쪽에 놓아야 더 빠른 연산결과를 얻을 수 있다.

`('a' <= ch && ch <= 'z') || ('A' <= ch && ch <= 'Z')`

위의 식은 문자 ch가 소문자 또는 대문자인지 확인하는 것인데, 이 식에서 문자 ch가 소문자인 조건을 대문자인 조건보다 왼쪽에 놓았다. 그 이유는 사용자로부터 문자 ch를 입력받을 때, 사용자가 대문자보다 소문자를 입력할 확률이 높다고 판단했기 때문이다. 실제로 사용자가 소문자를 더 자주 입력한다면, 이 식은 더 효율적으로 처리될 것이다.

▼ 예제 3-26/OperatorEx26.java

```
class OperatorEx26 {
    public static void main(String[] args) {
        int a = 5;
        int b = 0;

        System.out.printf("a=%d, b=%d%n", a, b);
        System.out.printf("a!=0 || ++b!=0 = %b%n", a!=0 || ++b!=0);
        System.out.printf("a=%d, b=%d%n", a, b);
```

```

        System.out.printf("a==0 && ++b!=0 = %b%n", a==0 && ++b!=0);
        System.out.printf("a=%d, b=%d%n", a, b);
    } // main의 끝
}

```

▼ 실행결과

```

a=5, b=0
a!=0 || ++b!=0 = true
a=5, b=0
a==0 && ++b!=0 = false
a=5, b=0

```

논리 연산자가 효율적인 연산을 하는지 확인하는 예제이다. 변수 b에 증감 연산자 ‘++’을 사용해서 우측 피연산자가 처리되면, b의 값이 증가하도록 했다.

그러나 실행결과에서 알 수 있듯이, 두 번의 논리연산 후에도 b의 값은 여전히 0인 채로 남아있다. ‘|| (OR)’의 경우는 좌측 피연산자($a \neq 0$)가 참이라서, 그리고 ‘&& (AND)’의 경우는 좌측 피연산자($a == 0$)가 거짓이라서 우측 피연산자를 평가하지 않았기 때문이다.

논리 부정 연산자 !

이 연산자는 피연산자가 true이면 false를, false면 true를 결과로 반환한다. 간단히 말해서, true와 false를 반대로 바꾸는 것이다.

x	!x
true	false
false	true

▲ 표3-13 논리 부정 연산자의 연산결과

어떤 값에 논리 부정 연산자 ‘!’를 반복적으로 적용하면, 참과 거짓이 차례대로 반복된다. 이 연산자의 이러한 성질을 이용하면, 한번 누르면 켜지고, 다시 한 번 누르면 꺼지는 TV의 전원버튼과 같은 ‘토글 버튼(toggle button)’을 논리적으로 구현할 수 있다.

false(거짓, off) → ! → true(참, on) → ! → false(거짓, off) → ! → true(참, on) → ! → ...

논리 부정 연산자 ‘!’가 주로 사용되는 곳은 조건문과 반복문의 조건식이며, 이 연산자를 잘 사용하면 조건식이 보다 이해하기 쉬워진다. 예를 들어 ‘문자 ch는 소문자가 아니다’라는 조건을 아래의 왼쪽과 같이 쓰기보다 오른쪽과 같이 논리부정연산자 ‘!’를 사용하는 쪽이 알기 쉽다.

ch < 'a' || ch > 'z'



! ('a' <= ch && ch <= 'z')

위와 같이 논리부정연산자 ‘!’를 적절히 사용해서 보다 이해하기 쉬운 식이 되도록 노력하자.

▼ 예제 3-27/OperatorEx27.java

```
class OperatorEx27 {
    public static void main(String[] args) {
        boolean b = true;
        char ch = 'C';

        System.out.printf("b=%b%n", b);
        System.out.printf("!b=%b%n", !b);
        System.out.printf("!!b=%b%n", !!b);
        System.out.printf("!!!b=%b%n", !!!b);
        System.out.println();

        System.out.printf("ch=%c%n", ch);
        System.out.printf("ch < 'a' || ch > 'z'=%b%n",
                         ch < 'a' || ch > 'z');
        System.out.printf("!( 'a'<=ch && ch<='z ')=%b%n",
                         !( 'a'<= ch && ch<='z' ));
        System.out.printf("  'a'<=ch && ch<='z' =%b%n",
                         'a'<=ch && ch<='z');

    } // main의 끝
}
```

▼ 실행결과

```
b=true
!b=false
!!b=true
!!!b=false

ch=C
ch < 'a' || ch > 'z'=true
!( 'a'<=ch && ch<='z ')=true
  'a'<=ch && ch<='z' =false
```

식 ‘!!b’가 평가되는 과정은 아래와 같다. 단항연산자는 결합방향이 오른쪽에서 왼쪽이므로 피연산자와 가까운 것부터 먼저 연산된다. 그래서 피연산자인 b와 가까운 논리 부정 연산자 ‘!’가 먼저 수행되어 false를 결과로 얻는다. 그리고 이 값에 다시 ‘!’연산을 수행하므로 true를 결과로 얻는다.

!!b	
→ !true	가까운 연산자가 먼저 연산된다.
→ !false	!true 의 결과는 false 이다.
→ true	!false 의 결과는 true 이다.

5.2 비트 연산자 & | ^ ~ << >>

비트 연산자는 피연산자를 비트단위로 논리 연산한다. 피연산자를 이진수로 표현했을 때의 각 자리를 아래의 규칙에 따라 표3-14와 같이 연산을 수행하며, 피연산자로 실수는 허용하지 않는다. 정수(문자 포함)만 허용된다.

| (OR연산자) 피연산자 중 한 쪽의 값이 1이면, 1을 결과로 얻는다. 그 외에는 0을 얻는다.

& (AND연산자) 피연산자 양 쪽이 모두 1이어야만 1을 결과로 얻는다. 그 외에는 0을 얻는다.

^ (XOR연산자) 피연산자의 값이 서로 다를 때만 1을 결과로 얻는다. 같을 때는 0을 얻는다.

x	y	$x \mid y$	$x \& y$	$x \wedge y$
1	1	1	1	0
1	0	1	0	1
0	1	1	0	1
0	0	0	0	0

▲ 표3-14 비트 연산자의 연산결과

| 참고 | 연산자 '^'는 배타적 XOR(exclusive OR)라고 하며, 피연산자의 값이 서로 다른 경우, 즉 배타적인 경우에만 참(1)을 결과로 얻는다.

비트OR연산자 '|'는 주로 특정 비트의 값을 변경할 때 사용한다. 아래의 식은 피연산자 0xAB의 마지막 4 bit를 'F'로 변경하는 방법을 보여준다.

식	2진수	16진수																
0xAB 0xF = 0xAF	<table border="1" style="display: inline-table; vertical-align: middle;"> <tr><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td></tr> </table>) <table border="1" style="display: inline-table; vertical-align: middle;"> <tr><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td></tr> </table>	1	0	1	0	1	0	1	1	0	0	0	0	1	1	1	1	0xAB 0xF
1	0	1	0	1	0	1	1											
0	0	0	0	1	1	1	1											
	<table border="1" style="display: inline-table; vertical-align: middle;"> <tr><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td></tr> </table>	1	0	1	0	1	1	1	1	0xAF								
1	0	1	0	1	1	1	1											

비트AND연산자 '&'는 주로 특정 비트의 값을 뽑아낼 때 사용한다. 아래의 식에서는 피연산자의 마지막 4 bit가 어떤 값인지 알아내는데 사용되었다.

식	2진수	16진수																
0xAB & 0xF = 0xB	<table border="1" style="display: inline-table; vertical-align: middle;"> <tr><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td></tr> </table> &) <table border="1" style="display: inline-table; vertical-align: middle;"> <tr><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td></tr> </table>	1	0	1	0	1	0	1	1	0	0	0	0	1	1	1	1	0xAB 0xF
1	0	1	0	1	0	1	1											
0	0	0	0	1	1	1	1											
	<table border="1" style="display: inline-table; vertical-align: middle;"> <tr><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td></tr> </table>	0	0	0	0	1	0	1	1	0xB								
0	0	0	0	1	0	1	1											

비트XOR연산자 '^'는 두 피연산자의 비트가 다를 때만 1이 된다. 그리고 같은 값으로 두고 XOR연산을 수행하면 원래의 값으로 돌아오는 특징이 있어서 간단한 암호화에 사용된다.

식	2진수	16진수								
$0xAB \wedge 0xF = 0xA4$	<table border="1"> <tr><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td></tr> </table>	1	0	1	0	1	0	1	1	0xAB
1	0	1	0	1	0	1	1			
<table border="1"> <tr><td>^)</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td></tr> </table>	^)	0	0	0	0	1	1	1	1	0xF
^)	0	0	0	0	1	1	1	1		
<table border="1"> <tr><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td></tr> </table>	1	0	1	0	0	1	0	0	0xA4	
1	0	1	0	0	1	0	0			
$0xA4 \wedge 0xF = 0xAB$	<table border="1"> <tr><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>1</td><td>1</td><td>1</td></tr> </table>	0	0	0	0	1	1	1	1	0XF
0	0	0	0	1	1	1	1			
<table border="1"> <tr><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td></tr> </table>	1	0	1	0	0	1	0	0	0xA4	
1	0	1	0	0	1	0	0			
<table border="1"> <tr><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td></tr> </table>	1	0	1	0	1	0	1	1	0xAB	
1	0	1	0	1	0	1	1			

지금까지 이해를 돋기 위해 2진수를 8자리로 표현하였지만, 사실은 int타입(4 byte)간의 연산이라 32자리로 표현하는 것이 맞다. 그리고 비트연산에서도 피연산자의 타입을 일치시키는 ‘산술 변환’이 일어날 수 있다.

▼ 예제 3-28/OperatorEx28.java

```
class OperatorEx28 {
    public static void main(String[] args) {
        int x = 0xAB, y = 0xF;

        System.out.printf("x = %#X \t\t%s%n", x, toBinaryString(x));
        System.out.printf("y = %#X \t\t%s%n", y, toBinaryString(y));
        System.out.printf("%#X | %#X = %#X \t\t%s%n" ,
                           x, y, x | y, toBinaryString(x | y));
        System.out.printf("%#X & %#X = %#X \t\t%s%n" ,
                           x, y, x & y, toBinaryString(x & y));
        System.out.printf("%#X ^ %#X = %#X \t\t%s%n" ,
                           x, y, x ^ y, toBinaryString(x ^ y));
        System.out.printf("%#X ^ %#X ^ %#X = %#X %s%n" ,
                           x, y, y, x ^ y ^ y, toBinaryString(x ^ y ^ y));
    } // main의 끝

    static String toBinaryString(int x) { // 10진 정수를 2진수로 변환하는 메서드
        String zero = "00000000000000000000000000000000";
        String tmp = zero + Integer.toBinaryString(x);
        return tmp.substring(tmp.length()-32);
    }
}
```

비트연산의 결과를 2진수로 출력하기 위해 `toBinaryString()`이라는 메서드를 작성해서 사용하였다. 이 메서드는 4 byte의 정수를 32자리의 2진수로 변환한다. 이 메서드가 어떻게 동작하는지를 이해하기에는 아직 배워야 할 것들이 많으므로 설명은 생략한다.

비트 전환 연산자 ~

이 연산자는 피연산자를 2진수로 표현했을 때, 0은 1로, 1은 0으로 바꾼다. 논리부정 연산자 '!'와 유사하다.

x	$\sim x$
1	0
0	1

▲ 표 3-15 비트전환 연산자의 2진 연산결과

비트 전환 연산자 '~'에 의해 '비트 전환'되고 나면, 부호있는 타입의 피연산자는 부호가 반대로 변경된다. 즉, 피연산자의 '1의 보수'를 얻을 수 있는 것이다. 그래서 비트 전환 연산자를 '1의 보수'연산자라고도 한다.

2진수	10진수																											
<table border="1" style="margin-left: auto; margin-right: auto;"> <tr><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td></tr> <tr><td>↓</td><td>↓</td><td>↓</td><td>↓</td><td>↓</td><td>↓</td><td>↓</td><td>↓</td></tr> <tr><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td></tr> </table>	0	0	0	0	1	0	1	0	↓	↓	↓	↓	↓	↓	↓	↓	1	1	1	1	0	1	0	1	<table border="1" style="margin-left: auto; margin-right: auto;"> <tr><td>10</td></tr> <tr><td>↓</td></tr> <tr><td>-11</td></tr> </table>	10	↓	-11
0	0	0	0	1	0	1	0																					
↓	↓	↓	↓	↓	↓	↓	↓																					
1	1	1	1	0	1	0	1																					
10																												
↓																												
-11																												

▲ 표 3-16 byte타입의 비트전환연산

예를 들어 10진수 10을 비트전환 연산한 결과는 -11이고, 이 값은 10의 '1의 보수'이다. 이미 배운 것과 같이 1의 보수에 1을 더하면 음수가 되므로 -11에 1을 더하면 -10이 되고 -11은 10의 '1의 보수'가 맞다는 것을 확인할 수 있다.

2진수	10진수								
<table border="1" style="margin-left: auto; margin-right: auto;"> <tr><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td></tr> </table>	0	0	0	0	1	0	1	0	10
0	0	0	0	1	0	1	0		
<table border="1" style="margin-left: auto; margin-right: auto;"> <tr><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td></tr> </table>	1	1	1	1	0	1	0	1	-11
1	1	1	1	0	1	0	1		
<table border="1" style="margin-left: auto; margin-right: auto;"> <tr><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td></tr> </table>	1	1	1	1	0	1	0	1	-11
1	1	1	1	0	1	0	1		
<table border="1" style="margin-left: auto; margin-right: auto;"> <tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>1</td></tr> </table>	0	0	0	0	0	0	0	1	+)
0	0	0	0	0	0	0	1		
<table border="1" style="margin-left: auto; margin-right: auto;"> <tr><td>1</td><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td><td>1</td><td>1</td></tr> </table>	1	1	1	1	1	0	1	1	-10
1	1	1	1	1	0	1	1		

▲ 표 3-17 음수를 2진수로 표현하는 방법

위의 표에서는 연산결과를 8자리의 2진수로 표현했지만, 비트 전환 연산자는 피연산자의 타입이 int보다 작으면 int로 자동 형변환(산술 변환) 후에 연산하기 때문에 연산결과는 32자리의 2진수이다.



▼ 예제 3-29/OperatorEx29.java

```

class OperatorEx29 {
    public static void main(String[] args) {
        byte p = 10;
        byte n = -10;

        System.out.printf(" p =%d \t%s%n", p, toBinaryString(p));
        System.out.printf("~p =%d \t%s%n", ~p, toBinaryString(~p));
        System.out.printf("~p+1=%d \t%s%n", ~p+1, toBinaryString(~p+1));
        System.out.printf("~~p =%d \t%s%n", ~~p, toBinaryString(~~p));
        System.out.println();
        System.out.printf(" n =%d%n", n);
        System.out.printf(" ~(n-1)=%d%n", ~(n-1));
    } // main의 끝

    // 10진 정수를 2진수로 변환하는 메서드
    static String toBinaryString(int x) {
        String zero = "00000000000000000000000000000000";
        String tmp = zero + Integer.toBinaryString(x);
        return tmp.substring(tmp.length()-32);
    }
}

```

▼ 실행결과

p =10	00000000000000000000000000000001010
~p ==-11	1111111111111111111111111111110101
~p+1=-10	1111111111111111111111111111110110
~~p =10	00000000000000000000000000000001010
 n ==-10	
~(n-1)=10	

결과를 보면, 어떤 양의 정수에 대한 음의 정수를 얻으려면 어떻게 해야 하는지 알 수 있다. 양의 정수 p가 있을 때, p에 대한 음의 정수를 얻으려면 ‘~p+1’을 계산하면 된다. 이 사실을 통해서 -10을 2진수로 어떻게 표현할 수 있는지 알 수 있을 것이다.

반대로 음의 정수 n이 있을 때, n에 대한 양의 정수를 얻으려면 ‘~(n-1)’을 계산하면 된다. 물론 부호연산자‘-’를 사용하면 되므로, 이렇게 복잡하게 변환하지 않는다. 참고로만 알아두자.

‘~~p’는 변수 p에 비트 전환 연산을 두 번 적용한 것인데, 1을 0으로 바꿨다가 다시 0을 1로 바꾸므로 원래의 값이 된다. 그러나 연산결과의 타입이 byte가 아니라 int라는 것에 주의하자.

쉬프트 연산자 << >>

이 연산자는 피연산자의 각 자리(2진수로 표현했을 때)를 ‘오른쪽(>>)’ 또는 ‘왼쪽(<<)’으로 이동(shift)한다고 해서 ‘쉬프트 연산자(shift operator)’라고 이름 붙여졌다.

예를 들어 ‘8 << 2’는 왼쪽 피연산자인 10진수 8의 2진수를 왼쪽으로 2자리 이동한다.

이 때, 자리이동으로 저장범위를 벗어난 값들은 버려지고 빈자리는 0으로 채워진다. 이 과정을 그림과 함께 단계별로 살펴보면 다음과 같다.

- ① 10진수 8은 2진수로 ‘00001000’이다.

0	0	0	0	1	0	0	0
---	---	---	---	---	---	---	---

- ② ‘8 << 2’은 10진수 8의 2진수를 왼쪽으로 2자리 이동시킨다.

0	0	0	0	1	0	0	0		
---	---	---	---	---	---	---	---	--	--

- ③ 자리이동으로 인해 저장범위를 벗어난 값은 버려지고, 빈자리는 0으로 채워진다.

0	0	0	0	1	0	0	0	0	0

- ④ ‘8 << 2’의 결과는 2진수로 ‘00100000’이 된다.(10진수로 32)

0	0	1	0	0	0	0	0
---	---	---	---	---	---	---	---

‘<<’연산자의 경우, 피연산자의 부호에 상관없이 각 자리를 왼쪽으로 이동시키며 빈칸을 0으로만 채우면 되지만, ‘>>’연산자는 오른쪽으로 이동시키기 때문에 부호있는 정수는 부호를 유지하기 위해 왼쪽 피연산자가 음수인 경우 빈자리를 1로 채운다. 물론 양수일 때는 0으로 채운다.

-8 >> 2 → -2

1	1	1	1	1	1	0	0	0
			1	1	1	1	1	0
1	1	1	1	1	1	1	0	0
1	1	1	1	1	1	1	0	0

8 >> 2 → 2

0	0	0	0	1	0	0	0	
			0	0	0	0	1	0
0	0	0	0	0	0	0	1	0
0	0	0	0	0	0	0	1	0

쉬프트 연산자의 좌측 피연산자는 산술변환이 적용되어 int보다 작은 타입은 int타입으로 자동 변환되고 연산결과 역시 int타입이 된다. 그러나 쉬프트 연산자는 다른 이항연산자들과 달리 피연산자의 타입을 일치시킬 필요가 없기 때문에 우측 피연산자에는 산술변환이 적용되지 않는다.

아래의 표는 쉬프트 연산의 결과를 2진수와 10진수로 나타낸 것이다. 각 쉬프트 연산의 결과를 비교해보자.

수식	자리이동	연산결과	
		2진수	10진수
$8 \gg 0$	없음	00000000 00000000 00000000 00001000	8
$8 \gg 1$	오른쪽으로 한 번	00000000 00000000 00000000 00000100	4
$8 \gg 2$	오른쪽으로 두 번	00000000 00000000 00000000 00000010	2
$-8 \gg 0$	없음	11111111 11111111 11111111 11111000	-8
$-8 \gg 1$	오른쪽으로 한 번	11111111 11111111 11111111 11111100	-4
$-8 \gg 2$	오른쪽으로 두 번	11111111 11111111 11111111 11111110	-2
$8 \ll 0$	없음	00000000 00000000 00000000 00001000	8
$8 \ll 1$	왼쪽으로 한 번	00000000 00000000 00000000 00010000	16
$8 \ll 2$	왼쪽으로 두 번	00000000 00000000 00000000 00100000	32
$-8 \ll 0$	없음	11111111 11111111 11111111 11111000	-8
$-8 \ll 1$	왼쪽으로 한 번	11111111 11111111 11111111 11110000	-16
$-8 \ll 2$	왼쪽으로 두 번	11111111 11111111 11111111 11100000	-32

▲ 표3-18 쉬프트 연산의 예

위의 표를 보면, 2진수 n자리를 왼쪽으로 이동하면 피연산자를 2^n 으로 곱한 결과를, 오른쪽으로 이동하면 피연산자를 2^n 으로 나눈 결과를 얻는다는 것을 알 수 있다.

$x \ll n$ 은 $x * 2^n$ 의 결과와 같다.
 $x \gg n$ 은 $x / 2^n$ 의 결과와 같다.

2진수로의 자리이동이 왜 2^n 으로 곱하거나 나눈 결과와 같은지 이해가 안가면, 10진수로 자리이동을 해보자. 예를 들어 10진수 123을 왼쪽으로 2자리 이동하면 12300이 되는데, 이 값은 123에 10^2 으로 곱한 결과라는 것을 알 수 있다. 이 값을 다시 오른쪽으로 2자리 이동하면 123이 되고 이 값은 12300을 10^2 으로 나눈 결과가 된다. 이처럼 10진수에서의 자리이동은 10^n 으로 곱하거나 나눈 결과를 얻는다. 단지 2진수이기 때문에 자리이동시 2^n 으로 곱하거나 나눈 결과를 얻는 것뿐이다.

그리고, ‘ $x \ll n$ ’ 또는 ‘ $x \gg n$ ’에서, n의 값이 자료형의 bit수 보다 크면, 자료형의 bit수로 나눈 나머지만큼만 이동한다. 예를 들어 int타입이 4 byte(=32 bit)인 경우, 자리수를 32번 바꾸면 결국 제자리로 돌아오기 때문에, ‘ $8 \gg 32$ ’는 아무 일도 하지 않는다. ‘ $8 \gg 34$ ’는 34를 32로 나눈 나머지인 2만큼만 이동하는 ‘ $8 \gg 2$ ’를 수행한다. 당연히 n은 정수만 가능하며 음수인 경우, 부호없는 정수로 자동 변환된다.

곱셈이나 나눗셈 연산자를 사용하면 같은 결과를 얻을 수 있는데, 굳이 쉬프트 연산자를 제공하는 이유 무엇일까? 그 이유는 속도 때문이다.

예를 들어 ‘ $8 \gg 2$ ’의 결과는 ‘ $8 / 4$ ’의 결과와 같다. 하지만, ‘ $8 / 4$ ’를 연산하는데 걸리는 시간보다 ‘ $8 \gg 2$ ’를 연산하는데 걸리는 시간이 더 적게 걸린다. 다시 말하면, ‘ \gg ’ 또는

'<<' 연산자를 사용하는 것이 나눗셈 '/' 또는 곱셈 '*' 연산자 보다 더 빠르다.

그러나 프로그램의 실행속도도 중요하지만 프로그램을 개발할 때 코드의 가독성(readability)도 중요하다. 쉬프트 연산자가 속도가 빠르긴 해도 곱셈이나 나눗셈 연산자보다는 가독성이 떨어질 것이다. 쉬프트 연산자보다 곱셈 또는 나눗셈 연산자를 주로 사용하고, 보다 빠른 실행속도가 요구되어지는 곳만 쉬프트 연산자를 사용하는 것이 좋다.

▼ 예제 3-30/OperatorEx30.java

```
class OperatorEx30 {
    // 10진 정수를 2진수로 변환하는 메서드
    static String toBinaryString(int x) {
        String zero = "00000000000000000000000000000000";
        String tmp = zero + Integer.toBinaryString(x);
        return tmp.substring(tmp.length()-32);
    }

    public static void main(String[] args) {
        int dec = 8;
        System.out.printf("%d >> %d = %4d \t%s%n",
                           dec, 0, dec >> 0, toBinaryString(dec >> 0));
        System.out.printf("%d >> %d = %4d \t%s%n",
                           dec, 1, dec >> 1, toBinaryString(dec >> 1));
        System.out.printf("%d >> %d = %4d \t%s%n",
                           dec, 2, dec >> 2, toBinaryString(dec >> 2));

        System.out.printf("%d << %d = %4d \t%s%n",
                           dec, 0, dec << 0, toBinaryString(dec << 0));
        System.out.printf("%d << %d = %4d \t%s%n",
                           dec, 1, dec << 1, toBinaryString(dec << 1));
        System.out.printf("%d << %d = %4d \t%s%n",
                           dec, 2, dec << 2, toBinaryString(dec << 2));
        System.out.println();

        dec = -8;
        System.out.printf("%d >> %d = %4d \t%s%n",
                           dec, 0, dec >> 0, toBinaryString(dec >> 0));
        System.out.printf("%d >> %d = %4d \t%s%n",
                           dec, 1, dec >> 1, toBinaryString(dec >> 1));
        System.out.printf("%d >> %d = %4d \t%s%n",
                           dec, 2, dec >> 2, toBinaryString(dec >> 2));

        System.out.printf("%d << %d = %4d \t%s%n",
                           dec, 0, dec << 0, toBinaryString(dec << 0));
        System.out.printf("%d << %d = %4d \t%s%n",
                           dec, 1, dec << 1, toBinaryString(dec << 1));
        System.out.printf("%d << %d = %4d \t%s%n",
                           dec, 2, dec << 2, toBinaryString(dec << 2));
        System.out.println();
    }
}
```

```

    dec = 8;
    System.out.printf("%d >> %2d = %4d \t%s%n",
                      dec, 0, dec >> 0, toBinaryString(dec >> 0));
    System.out.printf("%d >> %2d = %4d \t%s%n",
                      dec, 32, dec >> 32, toBinaryString(dec >> 32));
} // main의 끝
}

```

▼ 실행결과

```

8 >> 0 =     8  000000000000000000000000000000001000
8 >> 1 =     4  00000000000000000000000000000000100
8 >> 2 =     2  0000000000000000000000000000000010
8 << 0 =     8  000000000000000000000000000000001000
8 << 1 =    16  0000000000000000000000000000000010000
8 << 2 =    32  00000000000000000000000000000000100000

-8 >> 0 =    -8  11111111111111111111111111111111000
-8 >> 1 =    -4  1111111111111111111111111111111100
-8 >> 2 =    -2  1111111111111111111111111111111110
-8 << 0 =    -8  11111111111111111111111111111111000
-8 << 1 =   -16  111111111111111111111111111111110000
-8 << 2 =   -32  1111111111111111111111111111111100000

8 >> 0 =     8  000000000000000000000000000000001000
8 >> 32 =    8  000000000000000000000000000000001000

```

표3-18의 내용을 직접 확인할 수 있는 예제이다. 값을 바꿔가며 결과를 확인해보자.

▼ 예제 3-31/OperatorEx31.java

```

class OperatorEx31 {
    public static void main(String[] args) {
        int dec = 1234;
        int hex = 0xABCD;
        int mask = 0xF;

        System.out.printf("hex=%X%n", hex);
        System.out.printf("%X%n", hex & mask);

        hex = hex >> 4;
        System.out.printf("%X%n", hex & mask);

        hex = hex >> 4;
        System.out.printf("%X%n", hex & mask);
    } // main의 끝
}

```

▼ 실행결과

```

hex=ABCD
D
C
B
A

```

쉬프트 연산자와 비트AND연산자를 이용해서 16진수를 끝에서부터 한자리씩 뽑아내는 예제이다. 비트AND연산자는 두 bit가 모두 1일 때만 1이 되므로 0xABCD와 0x000F를 비트AND연산하면 다음과 같이 마지막 자리만 남고 나머지자리는 모두 0이 된다.

A	B	C	D
1010	1011	1100	1101
0000	0000	0000	1111
&)	0	0	F
	0000	0000	1101
	0	0	D

그 다음엔 쉬프트 연산자로 0xABCD를 2진수로 4자리를 오른쪽으로 이동한다. 2진수 4자리는 16진수로 한자리에 해당하므로 0xABCD는 0x0ABC가 된다.

0xABCD \gg 4 \rightarrow 0x0ABC

A	B	C	D	0	A	B	C
1010	1011	1100	1101	→	0000	1010	1011

| 참고 | 0xABCD의 왼쪽 첫 번째 비트가 1인데 왜 0으로 채워졌을까? 정수 상수는 int타입(4 byte)이므로 0xABCD는 사실 0x0000ABCD에서 앞의 0이 생략된 것인가 때문이다.

0x0ABC에 다시 0xF로 비트AND연산을 수행하면, 0xABCD의 오른쪽 끝에서 두번째 자리인 0xC를 결과로 얻을 수 있다.

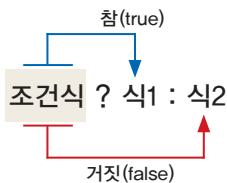
0	A	B	C
0000	1010	1011	1100
0000	0000	0000	1111
&)	0	0	F
	0000	0000	0000
	0	0	0
			C

위 과정을 반복하면 16진수 0xABCD의 각 자리를 하나씩 얻을 수 있다.

6. 그 외의 연산자

6.1 조건 연산자 ?: :

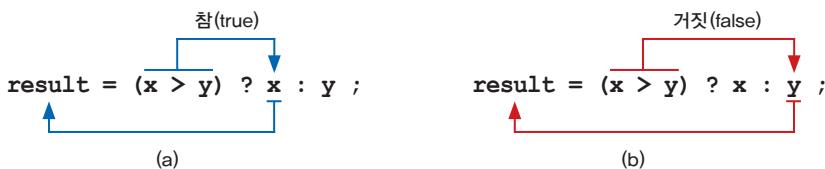
조건 연산자는 조건식, 식1, 식2 모두 세 개의 피연산자를 필요로 하는 삼항 연산자이며, 삼항 연산자는 조건 연산자 하나뿐이다.



조건 연산자는 첫 번째 피연산자인 조건식의 평가결과에 따라 다른 결과를 반환한다. 조건식의 평가결과가 true이면 식1이, false이면 식2가 연산결과가 된다. 가독성을 높이기 위해 조건식을 팔호()로 둘러싸는 경우가 많지만 필수는 아니다.

```
result = (x > y) ? x : y ;
```

위의 문장에서 식 ‘ $x > y$ ’의 결과가 true이면, 변수 result에는 x의 값이 저장되고, false이면 y의 값이 저장된다.



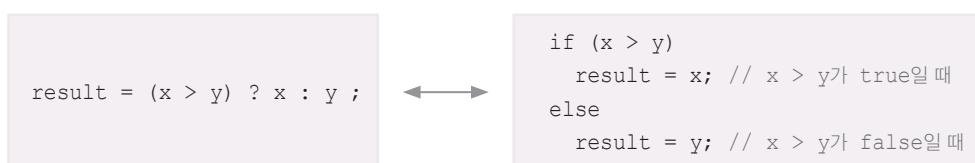
▲ 그림 3-4 조건식의 결과가 참(true)일 때(a)와 거짓(false)일 때(b)

만일 x의 값이 5, y의 값이 3이라면, 이 식은 다음과 같은 과정으로 계산된다.

```

result = (x > y) ? x : y ;
→ result = (5 > 3) ? 5 : 3 ;
→ result = (true) ? 5 : 3 ;      조건식이 true(참)이므로 연산결과는 5
→ result = 5;
    
```

조건 연산자는 조건문인 if문으로 바꿔 쓸 수 있으며, if문 대신 조건 연산자를 사용하면 코드를 보다 간단히 할 수 있다. 아래 왼쪽의 조건 연산자가 쓰인 문장을 if문으로 바꾸면 오른쪽과 같다.



아직 if문을 배우지 않았지만, 조건 연산자를 사용하는 것이 if문보다 간략하다는 것은 한 눈에 알 수 있다.

조건 연산자를 중첩해서 사용하면 셋 이상 중의 하나를 결과로 얻을 수 있다. 아래의 식은 x의 값이 양수면 1, 0이면 0, 음수면 -1, 즉 셋 중의 하나를 결과로 반환한다.

```
result = x > 0 ? 1 : (x == 0 ? 0 : -1);
```

조건 연산자의 결합규칙이 오른쪽에서 왼쪽이므로 괄호가 없어도 되지만 가독성을 높이기 위해 사용했다. 만일 x의 값이 3이라면, 위의 식은 아래와 같은 과정으로 처리된다.

```
result = x > 0 ? 1 : (x == 0 ? 0 : -1);
→ result = x > 0 ? 1 : (3 == 0 ? 0 : -1);
→ result = x > 0 ? 1 : (false ? 0 : -1); 조건식이 false이므로, 연산결과는 식2
→ result = 3 > 0 ? 1 : -1;
→ result = true ? 1 : -1; 조건식이 true이므로, 연산결과는 식1
→ result = 1;
```

조건 연산자를 여러 번 중첩하면 코드가 간략해지긴 하지만, 가독성이 떨어지므로 꼭 필요한 경우에 한번 정도만 중첩하는 것이 좋다.

그리고 조건 연산자의 식1과 식2, 이 두 피연산자의 타입이 다른 경우, 이항 연산자처럼 산술 변환이 발생한다.

```
x = x + (mod < 0.5 ? 0 : 0.5) 0과 0.5의 타입이 다르다.
→ x = x + (mod < 0.5 ? 0.0 : 0.5) 0이 0.0으로 변환되었다.
```

위의 식에서 조건 연산자의 피연산자 0과 0.5의 타입이 다르므로, 자동 형변환이 일어나서 double타입으로 통일되고 연산결과 역시 double타입이 된다.

▼ 예제 3-32/OperatorEx32.java

```
class OperatorEx32 {
    public static void main(String args[]) {
        int x, y, z;
        int absX, absY, absZ;
        char signX, signY, signZ;

        x = 10;
        y = -5;
        z = 0;

        absX = x >= 0 ? x : -x; // x의 값이 음수이면, 양수로 만든다.
        absY = y >= 0 ? y : -y;
        absZ = z >= 0 ? z : -z;
```

```

signX = x > 0 ? '+' : ( x==0 ? ' ' : '-' ); // 조건 연산자를 중첩
signY = y > 0 ? '+' : ( y==0 ? ' ' : '-' );
signZ = z > 0 ? '+' : ( z==0 ? ' ' : '-' );

System.out.printf("x=%c%d%n", signX, absX);
System.out.printf("y=%c%d%n", signY, absY);
System.out.printf("z=%c%d%n", signZ, absZ);
}

```

▼ 실행결과
x=+10
y=-5
z= 0

조건 연산자를 이용해서 변수의 절대값을 구한 후, 부호를 붙여 출력하는 예제이다. 간단해서 이해하는 데 별 어려움은 없을 것이다.

6.2 대입 연산자 = op=

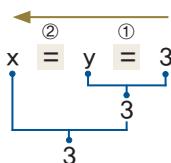
대입 연산자는 변수와 같은 저장공간에 값 또는 수식의 연산결과를 저장하는데 사용된다. 이 연산자는 오른쪽 피연산자의 값(식이라면 평가값)을 왼쪽 피연산자에 저장한다. 그리고 저장된 값을 연산결과로 반환한다. 예를 들어, 아래의 문장은 변수 x에 3을 저장하고, 연산결과인 3을 화면에 출력한다.

```

System.out.println(x = 3); // 변수 x에 3이 저장되고
→ System.out.println(3); // 연산결과인 3이 출력된다.

```

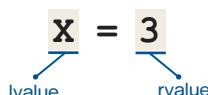
대입 연산자는 연산자들 중에서 가장 낮은 우선순위를 가지고 있기 때문에 식에서 제일 나중에 수행된다. 그리고 앞서 배운 것처럼 연산 진행 방향이 오른쪽에서 왼쪽이기 때문에 ‘x=y=3;’에서 ‘y=3’이 먼저 수행되고 그 다음에 ‘x=y’가 수행된다.



▲ 그림3-5 대입 연산자의 결합방향(오른쪽에서 왼쪽)

lvalue와 rvalue

대입 연산자의 왼쪽 피연산자를 ‘lvalue(left value)’이라 하고, 오른쪽 피연산자를 ‘rvalue(right value)’라고 한다.



▲ 그림3-6 대입 연산자의 lvalue(왼쪽 값)와 rvalue(오른쪽 값)

대입연산자의 rvalue는 변수뿐만 아니라 식이나 상수 등이 모두 가능한 반면, lvalue는 반드시 변수처럼 값을 변경할 수 있는 것이어야 한다. 그래서 리터럴이나 상수같이 값을 저장할 수 없는 것들은 lvalue가 될 수 없다.

```

int i = 0;
3 = i + 3;           // 에러. lvalue가 값을 저장할 수 있는 공간이 아니다.
i + 3 = i;          // 에러. lvalue의 연산결과는 리터럴(i+3 → 0+3 → 3)

final int MAX = 3;   // 변수 앞에 키워드 final을 붙이면 상수가 된다.
MAX = 10;             // 에러. 상수(MAX)에 새로운 값을 저장할 수 없다.

```

앞서 배운 것과 같이 변수 앞에 키워드 ‘final’을 붙이면 상수가 된다. 상수에 한 번 저장된 값은 바꿀 수 없다.

복합 대입 연산자

대입 연산자는 다른 연산자(op)와 결합하여 ‘op=’와 같은 방식으로 사용될 수 있다. 예를 들면, ‘i = i + 3’은 ‘i += 3’과 같이 표현될 수 있다. 그리고 결합된 두 연산자는 반드시 공백없이 붙여 써야 한다.

op=	=
i +=3;	i = i + 3;
i -= 3;	i = i - 3;
i *= 3;	i = i * 3;
i /= 3;	i = i / 3;
i %= 3;	i = i % 3;
i <= 3;	i = i << 3;
i >= 3;	i = i >> 3;
i &= 3;	i = i & 3;
i ^= 3;	i = i ^ 3;
i = 3;	i = i 3;
i *= 10 + j;	i = i * (10 + j);

▲ 표3-19 복합 대입 연산자의 종류

표3-19의 왼쪽은 복합 연산자의 사용 예이고, 오른쪽은 대입 연산자를 이용한 왼쪽과 동일한 의미의 식이다. 복합 연산자가 잘 익숙해지지 않는다면, 오른쪽과 같은 형태의 식을 사용하다가 점차 왼쪽의 형태로 바꿔가도록 하자.

한 가지 주의할 점은 표3-19의 마지막 줄처럼, 대입연산자의 우변이 둘 이상의 항으로 이루어져 있는 경우이다. ‘i *= 10 + j;’를 ‘i = i * 10 + j;’와 같은 것으로 오해하지 않도록 하자.

| 참고 | 연습문제는깃헙(<https://github.com/castello/javajungsuk4>)에서 PDF파일로 제공

Memo

Java

Programming
Language

Chapter 04

조건문과 반복문

if, switch, for, while statement

지금까지는 코드의 실행흐름이 무조건 위에서 아래로 한 문장씩 순차적으로 진행되었지만 때로는 조건에 따라 문장을 건너뛰고, 때로는 같은 문장을 반복해서 수행해야 할 때가 있다. 이처럼 프로그램의 흐름(flow)을 바꾸는 역할을 하는 문장들을 ‘제어문(flow control statement)’이라고 한다. 제어문에는 ‘조건문과 반복문’이 있는데, 조건문은 조건에 따라 다른 문장이 수행되도록 하고, 반복문은 특정 문장들을 반복해서 수행한다.

1. 조건문 – if, switch

조건문은 조건식과 조건에 따라 실행될 블럭{}으로 구성되며, 조건식의 결과에 따라 실행할 문장이 달라져서 프로그램의 실행 흐름을 바꿀 수 있다.

조건문은 if문과 switch문, 두 가지가 있으며 주로 if문이 사용된다. 처리할 경우의 수가 많을 때는 if문보다 switch문이 효율적이지만, switch문은 if문보다 제약이 많다.

1.1 if문

if문은 가장 기본적인 조건문이며, 다음과 같이 ‘조건식’과 ‘블럭{}’로 이루어져 있다. ‘if’의 뜻이 ‘만일 ~이라면...’이므로 ‘만일(if) 조건식이 참(true)이면 블럭{} 안의 문장들을 실행하라.’라는 의미로 이해하면 된다.

```
if (조건식) {
    // 조건식이 참(true) 일 때 수행될 문장들을 적는다.
}
```

만일 다음과 같은 if문이 있을 때, 조건식 ‘score > 60’이 참(true)이면 블럭{} 안의 문장이 실행되어 화면에 “합격입니다.”라고 출력되고 거짓(false)이면, if문 다음의 문장으로 넘어간다.

```
if (score > 60) {
    System.out.println("합격입니다.");
}
```

위 if문의 조건식이 평가되는 과정을 단계별로 살펴보면 다음과 같다. 변수 score의 값을 80으로 가정하였다.

score > 60 → 80 > 60 → true	조건식이 참(true) 이므로 블럭{} 안의 문장이 실행된다.
-----------------------------------	------------------------------------

위 조건식의 결과는 ‘true’이므로 if문 팔호{} 안의 문장이 실행된다. 만일 조건식의 결과가 ‘false’이면, 팔호{} 안의 문장은 실행되지 않을 것이다.

조건식

if문에 사용되는 조건식은 일반적으로 비교 연산자와 논리 연산자로 구성된다. 이미 연산자를 배울 때 살펴보았지만, 복습 차원에서 몇 개를 골라 표로 정리했다.

조건식	조건식이 참일 조건
<code>90 <= x && x <= 100</code>	정수 x가 90이상 100이하일 때
<code>x < 0 x > 100</code>	정수 x가 0보다 작거나 100보다 클 때
<code>x%3==0 && x%2!=0</code>	정수 x가 3의 배수지만, 2의 배수는 아닐 때
<code>ch=='y' ch=='Y'</code>	문자 ch가 'y' 또는 'Y'일 때
<code>ch==' ' ch=='\t' ch=='\n'</code>	문자 ch가 공백이거나 탭 또는 개행 문자일 때
<code>'A' <= ch && ch <= 'Z'</code>	문자 ch가 대문자일 때
<code>'a' <= ch && ch <= 'z'</code>	문자 ch가 소문자일 때
<code>'0' <= ch && ch <= '9'</code>	문자 ch가 숫자일 때
<code>str.equals ("yes")</code>	문자열 str의 내용이 "yes"일 때(대소문자 구분)
<code>str.equalsIgnoreCase ("yes")</code>	문자열 str의 내용이 "yes"일 때(대소문자 구분안함)

▲ 표4-1 자주 사용되는 조건식

조건식을 작성할 때 실수하기 쉬운 것이, 등가비교 연산자 '==' 대신 대입 연산자 '='를 사용하는 것이다. 예를 들어 'x가 0일 때 참'인 조건식은 'x==0'인데, 아래와 같이 실수로 'x=0'이라고 적는 경우가 있다.

```
if (x = 0) { ... }      x에 0이 저장되고, 결과는 0이 된다.  
→ if (0) { ... }       결과가 true 또는 false가 아니므로 에러가 발생한다.
```

자바에서 조건식의 결과는 반드시 true 또는 false이어야 한다는 것을 잊지 말자.

▼ 예제 4-1/FlowEx.java

```
class FlowEx {
    public static void main(String[] args) {
        int x= 0;
        System.out.printf("x=%d 일 때, 참인 것은%n", x);

        if(x==0)      System.out.println("x==0");
        if(x!=0)      System.out.println("x!=0");
        if(!(x==0))  System.out.println("!(x==0)");
        if(!(x!=0))  System.out.println("!(x!=0)");

        x = 1;
        System.out.printf("x=%d 일 때, 참인 것은%n", x);

        if(x==0)      System.out.println("x==0");
        if(x!=0)      System.out.println("x!=0");
        if(!(x==0))  System.out.println("!(x==0)");
        if(!(x!=0))  System.out.println("!(x!=0)");
    }
}
```

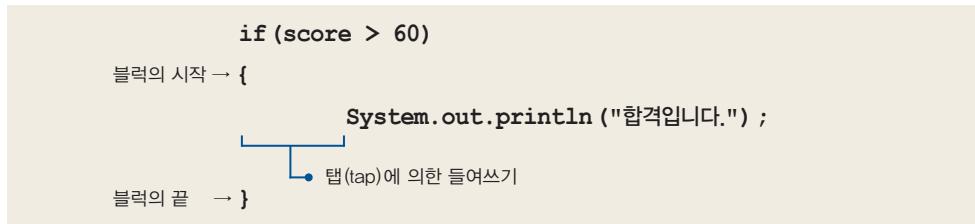
▼ 실행결과

x=0 일 때, 참인 것은
x==0
! (x!=0)
x=1 일 때, 참인 것은
x!=0
! (x==0)

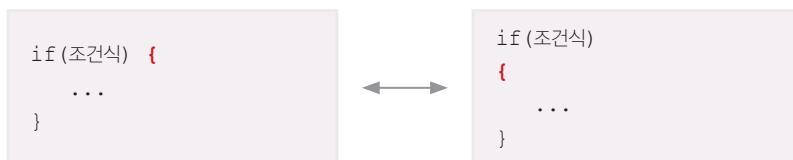
블럭{}

괄호{}로 여러 문장을 하나의 단위로 묶을 수 있는데, 이것을 ‘블럭(block)’이라고 한다. 블럭은 ‘{’로 시작해서, ‘}’로 끝나는데, ‘}’ 다음에 문장의 끝을 의미하는 ‘;’을 붙이지 않는다는 것에 주의하자.

블럭 내의 문장들은 텐(tab)으로 들여쓰기(indentation)를 해서 블럭 안에 속한 문장이 라는 것을 알기 쉽게 해주는 것이 좋다. 텐(tab)은 키보드의 맨 왼쪽에 있다.



블럭의 시작을 의미하는 ‘{’의 위치는 아래와 같이 두 가지 스타일이 있는데, 각 스타일마다 장단점이 있으므로 본인의 취향에 맞는 것으로 선택해서 사용하자. 왼쪽의 스타일은 라인의 수가 짧아진다는 장점이, 오른쪽의 스타일은 블럭의 시작과 끝을 찾기 쉽다는 장점이 있다.



블럭 안에는 보통 여러 문장을 넣지만, 한 문장만 넣거나 아무런 문장도 넣지 않을 수 있다. 만일 블럭 내의 문장이 하나뿐 일 때는 아래와 같이 괄호{}를 생략할 수 있다.

```

if(score > 60)
    System.out.println("합격입니다.");

```

또는 아래와 같이 한 줄로 쓸 수도 있다.

```

if(score > 60) System.out.println("합격입니다.");

```

이처럼 블럭 내의 문장이 하나뿐인 경우 괄호{}를 생략할 수 있지만 가능하면 생략하지 않고 사용하는 것이 바람직하다. 나중에 새로운 문장들이 추가되면 괄호{}로 문장들을 감싸 주어야 하는데, 이 때 괄호{}를 추가하는 것을 잊기 쉽기 때문이다.

```

if (score > 60)
    System.out.println("합격입니다."); // 문장1 if문에 속한 문장
    System.out.println("축하드립니다."); // 문장2. if문에 속한 문장이 아님

```

if문에 새로운 코드 한 줄이 추가되었지만, 괄호{}로 묶지 않았기 때문에 ‘문장2’는 if문에 속한 문장이 아니다. 들여쓰기를 했기 때문에 if문에 속한 것으로 착각하기 쉽지만 단지 들여쓰기를 했다고 if문에 속한 문장이 되는 것은 아니다.

아래와 같이 괄호{}로 묶어줘야만 두 문장 모두 if문에 속한 문장이 된다.

```
if (score > 60) {
    System.out.println("합격입니다."); // 문장1 if문에 속한 문장
    System.out.println("축하드립니다."); // 문장2. if문에 속한 문장
}
```

지금은 ‘이런 실수를 누가할까?’라는 생각이 들겠지만, 에러를 못 찾아서 한참을 고생하다가 이렇게 어처구니없는 곳에서 실수가 발견되곤 한다. 대부분의 실수는 늘 쉬운 곳에 있다는 것을 잊지 말자.

▼ 예제 4-2/FlowEx2.java

```
import java.util.*; // Scanner클래스를 사용하기 위해 추가

class FlowEx2 {
    public static void main(String[] args) {
        int input;

        System.out.print("숫자를 하나 입력하세요.>");
        Scanner scanner = new Scanner(System.in);
        String tmp = scanner.nextLine(); // 화면을 통해 입력받은 내용을 tmp에 저장
        input = Integer.parseInt(tmp); // 입력받은 문자열(tmp)을 숫자로 변환

        if(input==0) {
            System.out.println("입력하신 숫자는 0입니다.");
        }

        if(input!=0)
            System.out.println("입력하신 숫자는 0이 아닙니다.");
            System.out.printf("입력하신 숫자는 %d입니다.", input);
    } // main의 끝
}
```

▼ 실행결과 1

```
숫자를 하나 입력하세요.>3
입력하신 숫자는 0이 아닙니다.
입력하신 숫자는 3입니다.
```

▼ 실행결과 2

```
숫자를 하나 입력하세요.>0
입력하신 숫자는 0입니다.
입력하신 숫자는 0입니다.
```

두 번째 if문은 괄호{}를 생략했기 때문에, 조건식 바로 다음에 오는 하나의 문장만 if문에 속하게 된다. 그래서 실행결과를 보면 세 번째 출력문이 항상 출력된다.

```
if(input!=0)
    System.out.println("입력하신 숫자는 0이 아닙니다.");
    System.out.printf("입력하신 숫자는 %d입니다.", input); // if문 밖의 문장
```

만일 두 문장 모두 if문에 속하게 하려면 괄호{}로 묶으면 된다. 예제의 두 번째 if문에 괄호{}를 추가하여 다음과 같은 실행결과가 나오는지 확인해보자.

▼ 실행결과

```
숫자를 하나 입력하세요.>0
입력하신 숫자는 0입니다.
```

1.2 if–else문

if문의 변형인 if–else문의 구조는 다음과 같다. if문에 ‘else블럭’이 추가되었다. ‘else’의 뜻이 ‘그 밖의 다른’이므로 조건식의 결과가 참이 아닐 때, 즉 거짓일 때 else블럭의 문장을 수행하라는 뜻이다.

```
if (조건식) {
    // 조건식이 참(true) 일 때 수행될 문장들을 적는다.
} else {
    // 조건식이 거짓(false) 일 때 수행될 문장들을 적는다.
}
```

조건식의 결과에 따라 이 두 개의 블럭{} 중 어느 한 블럭{}의 내용이 수행되고 전체 if문을 벗어나게 된다. 두 블럭{}의 내용이 모두 수행되거나, 모두 수행되지 않는 경우는 있을 수 없다. 아래 왼 쪽의 두 개의 if문을 if–else문으로 바꾸면 오른쪽과 같다.

```
if(input==0) {
    System.out.println("0입니다.");
}
if(input!=0) {
    System.out.println("0이 아닙니다.");
}
```



```
if(input==0) {
    System.out.println("0입니다.");
} else {
    System.out.println("0이 아닙니다.");
}
```

왼쪽 코드의 두 조건식은 어느 한 쪽이 참이면 다른 한 쪽이 거짓인 상반된 관계에 있기 때문에 오른 쪽과 같이 if–else문으로 바꿀 수 있는 것이지, 두 개의 if문을 항상 if–else 문으로 바꿀 수 있는 것은 아니다.

그리고 왼 쪽의 코드는 두 개의 조건식을 계산해야하지만, if–else문을 사용한 오른쪽의 코드는 하나의 조건식만 계산하므로 더 효율적이다.

▼ 예제 4-3/FlowEx3.java

```
import java.util.*; // Scanner클래스를 사용하기 위해 추가
class FlowEx3 {
    public static void main(String[] args) {
        System.out.print("숫자를 하나 입력하세요.>");
    }
}
```

```

Scanner scanner = new Scanner(System.in);
int input = scanner.nextInt(); // 화면을 통해 입력받은 숫자를 input에 저장

if(input==0) {
    System.out.println("입력하신 숫자는 0입니다.");
} else { // input!=0인 경우
    System.out.println("입력하신 숫자는 0이 아닙니다.");
}
} // main의 끝
}

▼ 실행결과 1
숫자를 하나 입력하세요.>5
입력하신 숫자는 0이 아닙니다.

▼ 실행결과 2
숫자를 하나 입력하세요.>0
입력하신 숫자는 0입니다.

```

if-else문 역시 블럭 내의 문장이 하나뿐인 경우 아래와 같이 괄호{}를 생략할 수 있다.

```

if(input == 0)
    System.out.println("입력하신 숫자는 0입니다.");
else
    System.out.println("입력하신 숫자는 0이 아닙니다.");

```

1.3 if-else if문

if-else문은 경우의 수가 둘 일 때 그 중 하나가 수행되는 구조인데, 경우의 수가 셋 이상인 경우에는 어떻게 해야 할까? 그럴 때는 한 문장에 여러 개의 조건식을 쓸 수 있는 ‘if-else if’문을 사용하면 된다.

```

if (조건식1) {
    // 조건식1의 연산 결과가 참일 때 실행될 문장들을 적는다.
} else if (조건식2) {
    // 조건식2의 연산 결과가 참일 때 실행될 문장들을 적는다.
} else if (조건식3) {          // 여러 개의 else if를 사용할 수 있다.
    // 조건식3의 연산 결과가 참일 때 실행될 문장들을 적는다.
} else { // 마지막은 보통 else블럭으로 끝나며, else블럭은 생략가능하다.
    // 위의 어느 조건식도 만족하지 않을 때 실행될 문장들을 적는다.
}

```

첫 번째 조건식부터 순서대로 평가해서 결과가 참인 조건식을 만나면, 해당 블럭{}만 실행하고 ‘if-else if’문 전체를 벗어난다.

만일 결과가 참인 조건식이 하나도 없으면, 마지막에 있는 else블럭의 문장들이 실행된다. 그리고 else블럭은 생략이 가능하다. else블럭이 생략되었을 때는 if-else if문의 어떤 블럭도 수행되지 않을 수 있다.

예를 들어 다음과 같은 if-else if문이 있을 때, 변수 score의 값이 85라면, 다음의 과정으로 처리된다.

```

    거짓
    ①
    if(score >=90)
        85
    {
        grade = 'A';
    } else if(score >=80)
        85
    {
        grade = 'B';
    } else if(score >=70)
    {
        grade = 'C';
    } else {
        grade = 'D';
    }
    ③

```

① 결과가 참인 조건식을 만날 때까지 첫 번째 조건식부터 순서대로 평가한다.

(첫 번째 조건식이 거짓이면, 두 번째 조건식으로 넘어간다.)

② 참인 조건식을 만나면, 해당 블럭{}의 문장들을 실행한다.

③ if-else if문 전체를 빠져나온다.

▼ 예제 4-4/FlowEx4.java

```

import java.util.*;
class FlowEx4 {
    public static void main(String[] args) {
        int score = 0; // 점수를 저장하기 위한 변수
        char grade = ' '; // 학점을 저장하기 위한 변수. 공백으로 초기화한다.

        System.out.print("점수를 입력하세요.>");
        Scanner scanner = new Scanner(System.in);
        score = scanner.nextInt(); // 화면을 통해 입력받은 숫자를 score에 저장

        if (score >= 90) { // score가 90점 보다 같거나 크면 A학점
            grade = 'A';
        } else if (score >=80) { // score가 80점 보다 같거나 크면 B학점
            grade = 'B';
        } else if (score >=70) { // score가 70점 보다 같거나 크면 C학점
            grade = 'C';
        } else { // 나머지는 D학점
            grade = 'D';
        }
        System.out.println("당신의 학점은 "+ grade +"입니다.");
    }
}

```

▼ 실행결과 1
점수를 입력하세요.>70
당신의 학점은 C입니다.

▼ 실행결과 2
점수를 입력하세요.>63
당신의 학점은 D입니다.

점수를 입력하면, 그에 해당하는 학점을 출력하는 간단한 예제이다. 여기서 한 가지 눈여겨봐야할 것은 두 번째와 세 번째 조건식이다.

```
if (score >= 90) {
    grade = 'A';
} else if (80 <= score && score < 90) { // 80 ≤ score < 90
    grade = 'B';
} else if (70 <= score && score < 80) { // 70 ≤ score < 80
    grade = 'C';
} else { // score < 70
    grade = 'D';
}
```

점수가 90점 미만이고, 80점 이상인 사람에게 ‘B’학점을 주는 조건이라면, 위의 코드에서처럼 조건식이 ‘80 <= score && score < 90’이 되어야 하는 것이 아닌가?

그럼에도 불구하고, 두 번째 조건식을 ‘score >= 80’이라고 쓸 수 있는 것은 첫 번째 조건식인 ‘score >= 90’이 거짓이기 때문이다. ‘score >= 90’이 거짓이라는 것은 ‘score < 90’이 참이라는 뜻이므로 두 번째 조건식에서 ‘score < 90’이라는 조건을 중복해서 확인할 필요가 없다. 세 번째 조건식도 같은 이유로, ‘70 <= score && score < 80’이 아닌, ‘score >= 70’과 같이 간단히 쓸 수 있다.

```
if (score >= 90) {
    grade = 'A';
} else if (80 <= score && score < 90 ) {
    grade = 'B';
} else if (70 <= score && score < 80 ) {
    grade = 'C';
} else {
    grade = 'D';
}
```

```
if (score >= 90) {
    grade = 'A';
} else if (score >=80) {
    grade = 'B';
} else if (score >=70) {
    grade = 'C';
} else {
    grade = 'D';
}
```



그래서 만일 아래와 같이 왼쪽의 if–else if문을 별개의 if문으로 떼어낸다면, 오른쪽과 같이 조건식이 달라져야 한다.

```
if (score >= 90) {
    grade = 'A';
} else if (score >=80) {
    grade = 'B';
} else if (score >=70) {
    grade = 'C';
} else {
    grade = 'D';
}
```

```
if (score >= 90) {
    grade = 'A';
}

if (80 <= score && score < 90) {
    grade = 'B';
}

if (70 <= score && score < 80) {
    grade = 'C';
}

if (score < 70) {
    grade = 'D';
}
```



if–else if문이 여러 개의 if문을 합쳐놓은 것이긴 하지만, 조건식을 바꾸지 않고 여러 개의 if문으로 조개놓기만 하면 전혀 다른 코드가 된다는 점에 유의하자.

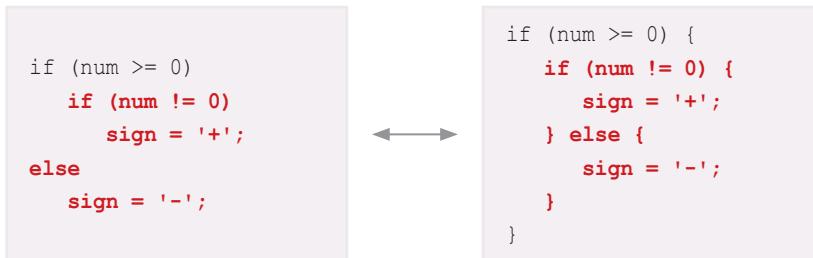
1.4 중첩 if문

if문의 블럭 내에 또 다른 if문을 포함시키는 것이 가능하며 이것을 중첩 if문이라고 부르며 중첩의 횟수에는 거의 제한이 없다.

```
if (조건식1) {
    // 조건식1의 연산 결과가 true일 때 실행될 문장들을 적는다.
    if (조건식2) {
        // 조건식1과 조건식2가 모두 true일 때 실행될 문장들
    } else {
        // 조건식1이 true이고, 조건식2가 false일 때 실행되는 문장들
    }
} else {
    // 조건식1이 false일 때 실행되는 문장들
}
```

위와 같이 내부의 if문은 외부의 if문보다 안쪽으로 들여쓰기를 해서 두 if문의 범위가 명확히 구분될 수 있도록 작성하는 것이 좋다.

중첩if문에서는 팔호{}의 생략에 더욱 조심해야 한다. 바깥쪽의 if문과 안쪽의 if문이 서로 엉켜서 if문과 else블럭의 관계가 의도한 바와 다르게 형성될 수도 있기 때문이다.



```
if (num >= 0)
    if (num != 0)
        sign = '+';
    else
        sign = '-';
}
if (num >= 0) {
    if (num != 0) {
        sign = '+';
    } else {
        sign = '-';
    }
}
```

왼쪽 코드는 언뜻 보기에도 else블럭이 바깥쪽의 if문에 속한 것처럼 보이지만, 팔호가 생략되었을 때 else블럭은 가까운 if문에 속한 것으로 간주되므로 실제로는 오른쪽과 같이 안쪽 if문의 else블럭이 되어버린다. 이제 else블럭은 어떤 경우에도 실행될 수 없다. 그래서 아래와 같이 팔호{}를 넣어서 if블럭과 else블럭의 관계를 확실히 해주는 것이 좋다.

```
if (num >= 0) {
    if (num != 0)
        sign = '+';
} else {
    sign = '-';
}
```

▼ 예제 4-5/FlowEx5.java

```

import java.util.*;

class FlowEx5 {
    public static void main(String[] args) {
        int score = 0;
        char grade = ' ', opt = '0';

        System.out.print("점수를 입력해주세요.>");

        Scanner scanner = new Scanner(System.in);
        score = scanner.nextInt(); // 화면을 통해 입력받은 점수를 score에 저장

        System.out.printf("당신의 점수는 %d입니다.%n", score);

        if (score >= 90) {           // score가 90점 보다 같거나 크면 A학점(grade)
            grade = 'A';
            if (score >= 98) {       // 90점 이상 중에서도 98점 이상은 A+
                opt = '+';
            } else if (score < 94) { // 90점 이상 94점 미만은 A-
                opt = '-';
            }
        } else if (score >= 80){     // score가 80점 보다 같거나 크면 B학점(grade)
            grade = 'B';
            if (score >= 88) {
                opt = '+';
            } else if (score < 84) {
                opt = '-';
            }
        } else {                     // 나머지는 C학점(grade)
            grade = 'C';
        }
        System.out.printf("당신의 학점은 %c%c입니다.%n", grade, opt);
    }
}

```

▼ 실행결과 1

점수를 입력해주세요.>100
당신의 점수는 100입니다.
당신의 학점은 A+입니다.

▼ 실행결과 2

점수를 입력해주세요. >81
당신의 점수는 81입니다.
당신의 학점은 B-입니다.

▼ 실행결과 3

점수를 입력해주세요. >85
당신의 점수는 85입니다.
당신의 학점은 B0입니다.

위 예제는 모두 3개의 if문으로 이루어져 있으며 if문 안에 또 다른 2개의 if문을 포함하고 있는 모습을 하고 있다. 제일 바깥쪽에 있는 if문에서 점수에 따라 학점(grade)을 결정하고, 내부의 if문에서는 학점을 더 세부적으로 나누어서 평가를 하고 그 결과를 출력한다. 외부 if문의 조건식에 의해 한번 걸려졌기 때문에 내부 if문의 조건식은 더 간단해 질 수 있다.

원래는 아래의 원쪽과 같이 써야하는데, '90 <= score'라는 조건이 이미 외부 if문의 조건식과 동일하므로 오른쪽의 조건식처럼 간단히 쓸 수 있는 것이다.

```

if (score >= 90) {
    grade = 'A';
    if (score >= 98) {
        opt = '+';
    } else if (90 <= score && score < 94) {
        opt = '-';
    }
    ...
}

```

```

if (score >= 90) {
    grade = 'A';
    if (score >= 98) {
        opt = '+';
    } else if (score < 94) {
        opt = '-';
    }
    ...
}

```

아래 오른쪽의 if–else if문은 else블럭이 생략되었는데, 만일 생략되지 않았다면 왼쪽과 같은 코드가 될 것이다. 변수 opt를 선언할 때 이미 '0'으로 초기화했기 때문에 굳이 else 블럭을 쓸 필요가 없는 것이다.

```

char opt = '0';
...
if (score >= 98) {
    opt = '+';
} else if (score < 94) {
    opt = '-';
} else { // 94 ≤ score < 98
    opt = '0';
}

```

```

char opt = '0';
...
if (score >= 98) {
    opt = '+';
} else if (score < 94) {
    opt = '-';
}

```

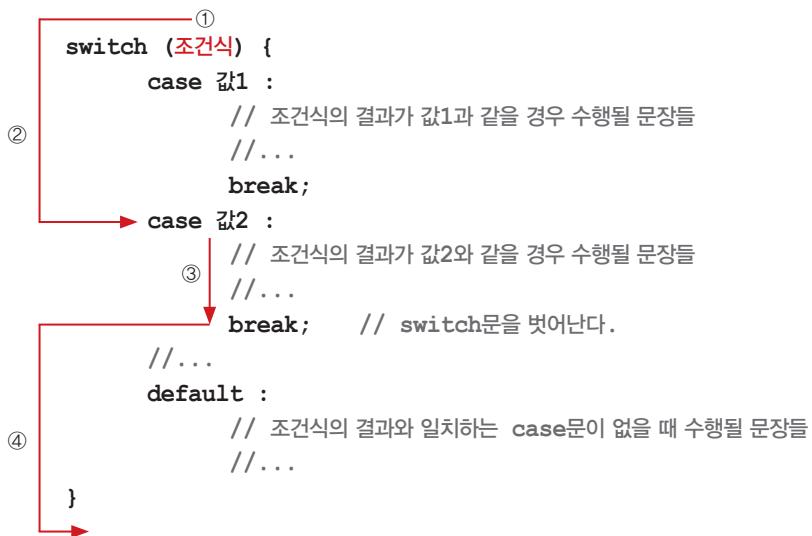
1.5 switch문

if문은 조건식의 결과가 참과 거짓, 두 가지 밖에 없기 때문에 경우의 수가 많아질수록 else-if를 계속 추가해야하므로 조건식이 많아져서 복잡해지고, 여러 개의 조건식을 계산해야하므로 처리시간도 많이 걸린다.

이러한 if문과 달리 switch문은 단 하나의 조건식으로 많은 경우의 수를 처리할 수 있고, 표현도 간결하므로 알아보기 쉽다. 그래서 처리할 경우의 수가 많은 경우에는 if문보다 switch문으로 작성하는 것이 좋다. 다만 switch문은 제약조건이 있기 때문에, 경우의 수가 많아도 어쩔 수 없이 if문으로 작성해야 하는 경우가 있다.

switch문은 조건식을 먼저 계산한 다음, 그 결과와 일치하는 case문으로 이동한다. 이동한 case문 아래에 있는 문장들을 수행하며, break문을 만나면 전체 switch문을 빠져나가게 된다.

- ① 조건식을 계산한다.
- ② 조건식의 결과와 일치하는 case문으로 이동한다.
- ③ 이후의 문장들을 실행한다.
- ④ break문이나 switch문의 끝을 만나면 switch문 전체를 빠져나간다.



만일 조건식의 결과와 일치하는 case문이 하나도 없는 경우에 default문으로 이동한다. default문은 if문의 else블럭과 같은 역할을 한다고 보면 된다. default문의 위치는 어디 라도 상관없으나 보통 마지막에 놓기 때문에 break문을 쓰지 않아도 된다.

switch문에서 break문은 각 case문의 영역을 구분하는 역할을 하는데, 만일 break문을 생략하면 case문 사이의 구분이 없어지므로 다른 break문을 만나거나 switch문 블럭{}의 끝을 만날 때까지 나오는 모든 문장들을 수행한다. 이러한 이유로 각 case문의 마지막에 break문을 빼먹는 실수를 하지 않도록 주의해야한다.

그러나 경우에 따라서는 다음과 같이 고의적으로 break문을 생략하는 경우도 있다.

```

switch (level) {
    case 3 :
        grantDelete(); // 삭제권한을 준다.
    case 2 :
        grantWrite(); // 쓰기권한을 준다.
    case 1 :
        grantRead(); // 읽기권한을 준다.
}

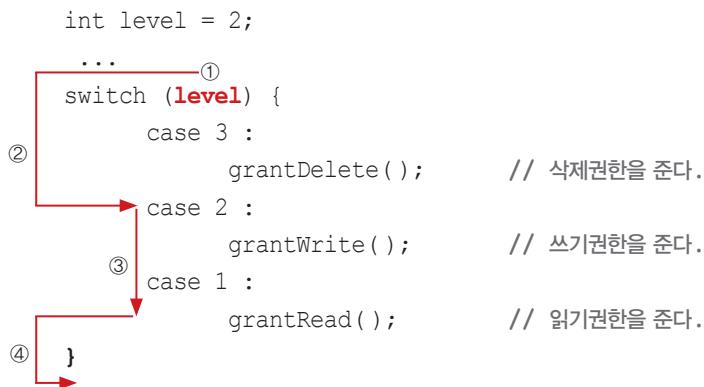
```

| 참고 | 위의 코드는 사용자에게 읽기, 쓰기, 삭제권한을 주는 기능의 `grantRead()`, `grantWrite()`, `grantDelete()`가 존재한다는 가정 하에 작성되었다.

위의 코드는 전체 코드가 아닌 코드의 일부를 발췌한 것인데, 회원제로 운영되는 웹사이트에서 많이 사용될 만한 코드이다.

로그인한 사용자의 등급(level)을 체크하여, 등급에 맞는 권한을 부여하는 방식으로 되어 있다. 제일 높은 등급인 3을 가진 사용자는 `grantDelete()`, `grantWrite()`, `grantRead()`가 모두 수행되어 읽기, 쓰기, 삭제 권한까지 모두 갖게 되고, 제일 낮은 등급인 1을 가진 사용자는 읽기 권한만을 갖게 된다.

예를 들어 변수 level의 값이 2라면, 다음과 같은 흐름으로 진행된다.



변수 level의 값이 2이므로 조건식의 결과는 2가 되고, 이와 일치하는 case문인 ‘case 2 :’로 이동한다. break문이 없으므로 ‘case 2 :’에 속한 grantWrite()뿐만 아니라 ‘case 1 :’에 속한 grantRead()까지 수행되고 더 이상 문장이 없으므로 switch문을 빠져나온다.

switch문의 제약조건

switch문의 조건식은 결과값이 반드시 정수이어야 하며, 이 값과 일치하는 case문으로 이동하기 때문에 case문의 값 역시 정수이어야 한다. 그리고 중복되지 않아야 한다. 같은 값의 case문이 여러 개이면, 어디로 이동해야할지 알 수 없기 때문이다.

계다가 case문의 값은 반드시 상수이어야 한다. 변수나 실수, 문자열은 case문의 값으로 사용할 수 없다.

switch문의 제약조건

1. switch문의 조건식 결과는 정수 값(long 타입 제외) 또는 참조형(객체 주소)만 허용
2. case문에는 정수 값, 문자열 리터럴 또는 참조형만 가능(중복 불가)

| 참고 | JDK 14부터 switch문의 조건식에 int 타입과 참조형이 가능하다. JDK 7이전에는 int 타입만 가능했다.

```

public static void main(String[] args) {
    int num = 10;
    final int ONE = 1;
    ...
    switch(result) {
        case '1':                  // OK. 문자 리터럴 (49와 동일)
        case ONE:                 // OK. 정수 상수
        case "YES":                // OK. 문자열 리터럴. JDK 7부터 허용
        case Double d:              // OK. 참조형. JDK 14부터 허용
        case num:                  // 에러. 변수는 불가
        case 1.0:                  // 에러. 실수도 불가
        ...
    }
}

```

문자 '1'은 정수 49와 동등하므로 문제가 없고, ONE은 정수가 아닌 것처럼 보이지만, 'final'이 붙은 정수 상수이므로 case문의 값으로 적합하다. 그러나 변수나 실수 리터럴은 case문의 값으로 적합하지 않다. JDK 14부터 switch문의 조건식에 참조형 값을 허용하면서 case문에도 Double과 같은 참조형이 가능해졌다. 다만 타입만 적으면 안되고 변수 선언하듯이 'Double d'와 같은 형식으로 적어야 한다.

참조형은 앞으로 자세히 배울 것이니 지금은 가볍게 보고 넘어가자.

▼ 예제 4-6/FlowEx6.java

```
import java.util.*;
class FlowEx6 {
    public static void main(String[] args) {
        System.out.print("현재 월을 입력하세요.>");
        Scanner scanner = new Scanner(System.in);
        int month = scanner.nextInt(); // 화면을 통해 입력받은 숫자를 month에 저장
        switch(month) {
            case 3:
            case 4:
            case 5:
                System.out.println("현재의 계절은 봄입니다.");
                break;
            case 6: case 7: case 8:
                System.out.println("현재의 계절은 여름입니다.");
                break;
            case 9: case 10: case 11:
                System.out.println("현재의 계절은 가을입니다.");
                break;
            default:
                case 12: case 1: case 2:
                    System.out.println("현재의 계절은 겨울입니다.");
        }
    } // main의 끝
}
```

▼ 실행결과

```
현재 월을 입력하세요.>3
현재의 계절은 봄입니다.
```

현재 몇 월인지 입력받아서 해당하는 계절을 출력하는 예제이다. 간단한 예제이므로 별로 설명할 것은 없다 case문은 한 줄에 하나씩 쓰던, 한 줄에 붙여서 쓰던 상관없다.

```
case 3:
case 4:
case 5:
    System.out.println("현재의 계
절은 ...");
    break;
```



```
case 3: case 4: case 5:
System.out.println("현재의 계
절은 ...");
break;
```

그리고 예제의 switch문을 if문으로 변경하면 다음과 같다.

```

if(month == 3 || month == 4 || month == 5) {
    System.out.println("현재의 계절은 봄입니다.");
} else if(month == 6 || month == 7 || month == 8) {
    System.out.println("현재의 계절은 여름입니다.");
} else if(month == 9 || month == 10 || month == 11) {
    System.out.println("현재의 계절은 가을입니다.");
} else { // if(month == 12 || month == 1 || month == 2)
    System.out.println("현재의 계절은 겨울입니다.");
}

```

두 문장을 비교해보면, 이 예제에서는 if문보다 switch문이 더 알아보기 쉽고 간결하다는 것을 알 수 있다.

▼ 예제 4-7/FlowEx7.java

```

import java.util.*;
class FlowEx7 {
    public static void main(String[] args) {
        System.out.print("가위(1), 바위(2), 보(3) 중 하나를 입력하세요.>");
        Scanner scanner = new Scanner(System.in);
        int user = scanner.nextInt(); // 화면을 통해 입력받은 숫자를 user에 저장
        int com = (int)(Math.random() * 3) + 1; // 1, 2, 3중 하나가 com에 저장됨

        System.out.println("당신은 " + user + "입니다.");
        System.out.println("컴은 " + com + "입니다.");

        switch(user-com) {
            case 2: case -1:
                System.out.println("당신이 졌습니다.");
                break;
            case 1: case -2:
                System.out.println("당신이 이겼습니다.");
                break;
            case 0:
                System.out.println("비겼습니다.");
                break;
        } // 마지막 문장이므로 break를 사용할 필요가 없다.
    } // main의 끝
}

```

▼ 실행결과 1

```

가위(1), 바위(2), 보(3) 중 하나를 입력하세요.>1
당신은 1입니다.
컴은 1입니다.
비겼습니다.

```

▼ 실행결과 2

```

가위(1), 바위(2), 보(3) 중 하나를 입력하세요.>3
당신은 3입니다.
컴은 2입니다.
당신이 이겼습니다.

```

이 예제는 컴퓨터와 사용자가 가위바위보를 하는 간단한 게임이다. 사용자로부터 1(가위), 2(바위), 3(보) 중의 하나를 입력받고, 컴퓨터는 1, 2, 3 중에서 하나를 임의로 선택한다.

난수(임의의 수)를 얻기 위해서 Math.random()을 사용했는데, 이 메서드는 0.0과 1.0사이의 범위에 속하는 하나의 double값을 반환한다. 0.0은 범위에 포함되고 1.0은 포함되지 않는다.

```
0.0 <= Math.random() < 1.0
```

만일 1과 3 사이의 정수를 구하기를 원한다면, 다음과 같은 과정으로 난수를 구하는 식을 얻을 수 있다.

1. 각 변에 3을 곱한다.

```
0.0 * 3 <= Math.random() * 3 < 1.0 * 3  
0.0 <= Math.random() * 3 < 3.0
```

2. 각 변을 int형으로 변환한다.

```
(int) 0.0 <= (int) (Math.random() * 3) < (int) 3.0  
0 <= (int) (Math.random() * 3) < 3
```

3. 각 변에 1을 더한다.

```
0 + 1 <= (int) (Math.random() * 3) + 1 < 3 + 1  
1 <= (int) (Math.random() * 3) + 1 < 4
```

자, 이제는 1과 3사이의 정수 중 하나를 얻을 수 있다. 1은 포함되고 4는 포함되지 않는다.

| 참고 | 순서 2와 3을 바꿔서, 각 변에 1을 먼저 더한 다음 int형으로 변환해도 같은 결과를 얻는다.

위와 같이 식을 변환해가며 범위를 조절하면 원하는 범위의 값을 얻을 수 있다. 주사위를 던졌을 때 나타나는 임의의 값을 얻기 위해서는 3대신 6을 곱하면 된다. 그렇게 하면 1과 6사이의 값을 얻어낼 수 있을 것이다.

이제 사용자가 입력한 값(user)하고 컴퓨터가 생성한 난수(com)하고 비교해서 가위바위보의 승부결과를 판단해야 한다. 두 값 모두 3가지 값이 가능하므로, 아래의 표와 같이 모두 9가지의 경우의 수를 처리해야한다.

		com	가위(1)	바위(2)	보(3)
		user	가위(1)	무승부	컴승
		가위(1)	무승부	컴승	유저승
		바위(2)	유저승	무승부	컴승
		보(3)	컴승	유저승	무승부

user에서 com의 값을 빼면, 아래의 표와 같은 결과를 얻는다. 무승부는 0, 컴퓨터 승리는 -1과 2, 사용자의 승리는 1, -2이다.

com user	가위(1)	바위(2)	보(3)
가위(1)	0	-1	-2
바위(2)	1	0	-1
보(3)	2	1	0

경우의 수가 9개에서 5개로 줄었다. 그리고 이 값들은 모두 정수이므로 switch문으로 처리가 가능하다.

```
switch(user - com) {
    case 2: case -1:
        System.out.println("당신이 졌습니다.");
        break;
    case 1: case -2:
        System.out.println("당신이 이겼습니다.");
        break;
    case 0:
        System.out.println("비겼습니다.");
        break; // 마지막 문장이므로 break를 사용할 필요가 없다.
}
```

▼ 예제 4-8/FlowEx8.java

```
import java.util.*;
class FlowEx8 {
    public static void main(String[] args) {
        System.out.print("당신의 주민번호를 입력하세요. (011231-1111222)>");
        Scanner scanner = new Scanner(System.in);
        String regNo = scanner.nextLine();

        char gender = regNo.charAt(7); // 입력받은 번호의 8번째 문자를 gender에 저장

        switch(gender) {
            case '1': case '3':
                System.out.println("당신은 남자입니다.");
                break;
            case '2': case '4':
                System.out.println("당신은 여자입니다.");
                break;
            default:
                System.out.println("유효하지 않은 주민등록번호입니다.");
        }
    } // main의 끝
}
```

▼ 실행결과

당신의 주민번호를 입력하세요. (011231-1111222)>**110101-2111222**
 당신은 여자입니다.

주민등록번호를 입력받아서 성별을 출력하는 예제이다. 주민등록번호 뒷 번호의 첫 자리의 값은 성별을 의미하는데, 그 값이 1, 3이면 남자, 2, 4이면 여자를 의미한다. 입력받은 주민등록번호는 char배열 regNo에 저장되며, 이 배열에서 성별을 의미하는 값은 8번째에 저장되어 있다.

```
gender = regNo.charAt(7); // 입력받은 번호의 8번째 문자를 gender에 저장
```

문자열에 저장된 문자는 ‘문자열.charAt(index)’로 가져올 수 있는데, index는 연속된 정수이며 1이 아닌 0부터 시작한다. 그래서 8번째 문자는 regNo.charAt(8)이 아닌 regNo.charAt(7)이다.

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13
regNo	'1'	'1'	'0'	'1'	'0'	'1'	'-'	2	'1'	'1'	'1'	'2'	'2'	'2'

char는 하나의 문자를 다루기 위한 타입이지만, char타입의 값은 사실 문자가 아닌 정수(유니코드)로 저장되기 때문에 이처럼 char타입의 값도 switch문의 조건식과 case문에 사용할 수 있다.

▼ 예제 4-9/FlowEx9.java

```
import java.util.*;

class FlowEx9 {
    public static void main(String[] args) {
        char grade = ' ';

        System.out.print("당신의 점수를 입력하세요. (1~100)>");

        Scanner scanner = new Scanner(System.in);
        int score = scanner.nextInt(); // 화면을 통해 입력받은 숫자를 score에 저장

        switch(score) {
            case 100: case 99: case 98: case 97: case 96:
            case 95: case 94: case 93: case 92: case 91: case 90:
                grade = 'A';
                break;
            case 89: case 88: case 87: case 86: case 85:
            case 84: case 83: case 82: case 81: case 80:
                grade = 'B';
                break;
            case 79: case 78: case 77: case 76: case 75:
            case 74: case 73: case 72: case 71: case 70:
                grade = 'C';
                break;
            default :
                grade = 'F';
        } // end of switch
    }
}
```

```

        System.out.println("당신의 학점은 "+ grade +"입니다.");
    }
}

```

▼ 실행결과 1

당신의 점수를 입력하세요. (1~100)>82
당신의 학점은 B입니다.

▼ 실행결과 2

당신의 점수를 입력하세요. (1~100)>69
당신의 학점은 F입니다.

이 예제는 예제4-4의 if문을 switch문을 이용해서 변경한 것이다. 이 예제를 if문을 이용해서 구현하려면, 조건식이 4개가 필요하며, 최대 4번의 조건식을 계산해야 한다.

반면에 switch문은 조건식을 1번만 계산하면 되므로 더 빠르다. 그러나 case문이 너무 많아져서 좋지 않은 코드가 되었다. 반드시 속도를 더 향상시켜야 한다면 복잡하더라도 switch문을 선택해야겠지만, 그렇지 않다면 이런 경우 if문이 더 적합하다.

▼ 예제 4-10/FlowEx10.java

```

import java.util.*;

class FlowEx10 {
    public static void main(String[] args) {
        int score = 0;
        char grade = ' ';

        System.out.print("당신의 점수를 입력하세요. (1~100)>");

        Scanner scanner = new Scanner(System.in);
        String tmp = scanner.nextLine(); // 화면을 통해 입력받은 내용을 tmp에 저장
        score = Integer.parseInt(tmp); // 입력받은 문자열(tmp)를 숫자로 변환

        switch(score/10) {
            case 10:
            case 9 :
                grade = 'A';
                break;
            case 8 :
                grade = 'B';
                break;
            case 7 :
                grade = 'C';
                break;
            default :
                grade = 'F';
        } // end of switch

        System.out.println("당신의 학점은 "+ grade +"입니다.");
    } // main의 끝
}

```

▼ 실행결과

당신의 점수를 입력하세요. (1~100)>82
당신의 학점은 B입니다.

이전 예제에 기교를 부려서 보다 간결하게 작성한 예제이다. score를 10으로 나누면, 전에 배운 것과 같이 ‘int / int’의 결과는 int이기 때문에, 예를 들어 ‘88/10’은 8.8이 아니라 8을 얻는다. 따라서 80과 89사이의 숫자들은 10으로 나누면 결과가 8이 된다. 마찬가지로 70과 79사이의 숫자들은 10으로 나누면 7이 된다.

이처럼 switch문에서는 조건식을 잘 만들어서 case문의 갯수를 줄이는 것이 중요하다.

switch문의 중첩

if문처럼 switch문도 중첩이 가능하다. 아래의 예제는 보기만 해도 별도의 설명이 필요 없을 것이다. 한 가지 주의할 점은 중첩 switch문에서 break문을 빼먹기 쉽다는 것이다.

▼ 예제 4-11/FlowEx11.java

```
import java.util.*;
class FlowEx11 {
    public static void main(String[] args) {
        System.out.print("당신의 주민번호를 입력하세요. (011231-1111222) >");

        Scanner scanner = new Scanner(System.in);
        String regNo = scanner.nextLine();
        char gender = regNo.charAt(7); // 입력받은 번호의 8번째 문자를 gender에 저장

        switch(gender) {
            case '1': case '3':
                switch(gender) {
                    case '1':
                        System.out.println("당신은 2000년 이전에 출생한 남자입니다.");
                        break;
                    case '3':
                        System.out.println("당신은 2000년 이후에 출생한 남자입니다.");
                        break; // 이 break문을 빼먹지 않도록 주의
                }
            case '2': case '4':
                switch(gender) {
                    case '2':
                        System.out.println("당신은 2000년 이전에 출생한 여자입니다.");
                        break;
                    case '4':
                        System.out.println("당신은 2000년 이후에 출생한 여자입니다.");
                        break;
                }
                break;
            default:
                System.out.println("유효하지 않은 주민등록번호입니다.");
        }
    } // main의 끝
}
```

▼ 실행결과

당신의 주민번호를 입력하세요. (011231-1111222) > 010101-4111222
당신은 2000년 이후에 출생한 여자입니다.

switch식(switch expression)

JDK 14부터 switch문의 기능이 확장되어서, 전통적으로 문장(statement)이었던 switch 문이 식(expression)으로도 쓰일 수 있다. 예제4-10의 switch문을 switch식으로 바꾸면 다음과 같다.

```
char grade = switch(score/10) {
    case 9, 10 -> 'A'; // 콤마(,)로 여러 case를 합칠 수 있다.
    case 8      -> 'B'; // break;가 없어도 다음 case로 넘어가지 않는다.
    case 7      -> 'C';
    default     -> 'F';
}; ← 끝에 세미콜론을 붙이는것을 잊지말 것
```

코드가 훨씬 간결해졌다. 예전의 switch문과 비슷하게 작성하는 것도 여전히 가능하다. 하지만 break문 대신 yield문을 사용해서 대입할 값을 지정해야 한다.

| 참고 | yield는 var처럼 상황에 따라 예약어가 되는 문맥 예약어라서 switch식 내에서만 예약어이다.

```
char grade = switch(score/10) {
    case 10 :
    case 9 :
        yield 'A'; // score가 90이면, grade에 'A'를 대입
    case 8 : yield 'B';
    case 7 : yield 'C';
    default : yield 'F';
};
```

아무래도 화살표(->)를 사용하는 쪽이 break나 yield를 붙이지 않아도 되니 간결하다. 앞서 break문을 생략하는 코드를 소개했지만, break문을 넣어야하는데 있는 경우가 많아서 의도한 것인지 실수인지 알기 어려워서 break문을 아예 쓰지 않는 쪽으로 변경되었다.

```
char grade = switch(score/10) {
    case 9, 10 -> { // 여러 문장은 괄호{}로 묶는다.
        System.out.println("score = " + score);
        yield 'A'; // yield를 생략할 수 없다.
    }
    case 8      -> 'B';
    ...
}
```

case문에 해당하는 문장이 여럿일 때는 괄호{}로 묶는다. break문은 쓰지 않으며 yield는 생략할 수 없다. switch식은 어떤 조건에서도 반드시 결과값을 반환해야 하므로 모든 경우를 다 처리할 수 있어야 한다.

```

char grade = switch(score/10) {
    case 9, 10 -> 'A';
    case 8      -> 'B';
    case 7      -> 'C';
//    default     -> 'F'; // 이 문장이 없으면 예외. 값을 반환할 수 없는 경우가 존재
};

```

switch문은 모던 프로그래밍 언어에서 활용도가 높고 점점 복잡해지고 있다. 이 외에도 더 많은 기능이 있지만 진도에 맞춰 추가로 설명할 것이다.

처음 배울 때는 if문만으로도 프로그래밍이 충분히 가능하므로 switch문에 대한 부담은 갖지 않길 바란다. 다음은 예제4-10의 switch문을 식으로 변경한 것이다.

▼ 예제 4-12/FlowEx12.java

```

import java.util.*;

class FlowEx12 {
    public static void main(String[] args) {
        int score = 0;

        System.out.print("당신의 점수를 입력하세요. (1~100)>");

        Scanner scanner = new Scanner(System.in);
        String tmp = scanner.nextLine(); // 화면을 통해 입력받은 내용을 tmp에 저장
        score = Integer.parseInt(tmp); // 입력받은 문자열(tmp)를 숫자로 변환

        char grade = switch(score/10) {
            case 9, 10 -> 'A';
            case 8      -> 'B';
            case 7      -> 'C';
            default     -> 'F';
        };

        System.out.println("당신의 학점은 "+ grade +"입니다.");
    } // main의 끝
}

```

▼ 실행결과
당신의 점수를 입력하세요. (1~100)> 82
당신의 학점은 B입니다.

2. 반복문 – for, while, do-while

반복문은 어떤 작업이 반복적으로 실행되게 할 때 사용되며, 반복문의 종류로는 for문과 while문, 그리고 while문의 변형인 do-while문이 있다.

for문이나 while문에 속한 문장은 조건에 따라 한 번도 실행되지 않을 수 있지만 do-while문에 속한 문장은 무조건 최소한 한 번은 실행될 것이 보장된다. 반복문은 주어진 조건을 만족하는 동안 주어진 문장들을 반복적으로 실행하므로 조건식을 포함하며, if문과 마찬가지로 조건식의 결과가 true이면 참이고, false면 거짓으로 간주된다.

for문과 while문은 구조와 기능이 유사하여 항상 서로 변환이 가능하기 때문에 반복문을 작성해야 할 때 for문과 while문 중 어느 쪽을 선택해도 좋으나 for문은 주로 반복 횟수를 알고 있을 때 사용한다.

2.1 for문

for문은 반복 횟수를 알고 있을 때 적합하다. 구조가 조금 복잡하지만 직관적이라 오히려 이해하기 쉽다. 자세한 설명에 앞서 가장 기본적인 for문의 예를 하나 소개할까 한다. 아래의 for문은 블럭{} 내의 문장을 5번 반복한다. 즉, “I can do it.”이라는 문장이 5번 출력된다.

```

 1부터      5까지      1씩 증가
    ↓          ↓          ↓
for(int i=1;i<=5;i++) { // i=1,2,3,4,5
    System.out.println("I can do it.");
}

```

변수 i에 1을 저장한 다음, 매 반복마다 i의 값을 1씩 증가시킨다. 그러다가 i의 값이 5를 넘으면 조건식 ‘*i<=5*’가 거짓이 되어 반복을 마치게 된다. i의 값이 1부터 5까지 1씩 증가하니까 모두 5번 반복한다. 만일 10번 반복하기를 원한다면, 5를 10으로 바꾸기만 하면 된다.

이제 for문에 대해 좀 더 자세히 배운 다음에 예제를 통해 자주 사용되는 for문의 형태를 가장 쉬운 것부터 조금씩 난이도를 높여가며 소개할 것이다.

for문의 구조와 수행순서

for문은 아래와 같이 ‘초기화’, ‘조건식’, ‘증감식’, ‘블럭{}’, 모두 4부분으로 이루어져 있으며, 조건식이 참인 동안 블럭{} 내의 문장들을 반복하다 거짓이 되면 반복문을 벗어난다.

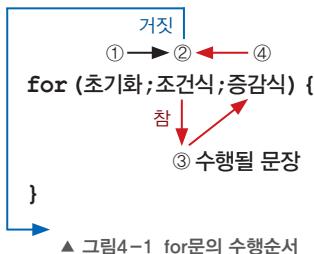
```

for (초기화; 조건식; 증감식) {
    // 조건식이 참일 때 수행될 문장들을 적는다.
}

```

| 참고 | 반복하려는 문장이 단 하나일 때는 괄호{}를 생략할 수 있다.

제일 먼저 ‘①초기화’가 수행되고, 그 이후부터는 조건식이 참인 동안 ‘②조건식 → ③수행될 문장 → ④증감식’의 순서로 반복된다. 그러다가 ②조건식이 거짓이 되면, for문 전체를 빠져나가게 된다.



▲ 그림4-1 for문의 수행순서

초기화

반복문에 사용될 변수를 초기화하는 부분이며 처음에 한번만 수행된다. 보통 변수 하나로 for문을 제어하지만, 둘 이상의 변수가 필요할 때는 아래와 같이 콤마','를 구분자로 변수를 초기화하면 된다. 단, 두 변수의 타입이 같아야 한다.

```
for(int i=1;i<=10;i++) { ... }      // 변수 i의 값을 1로 초기화 한다.  
for(int i=1,j=0;i<=10;i++) { ... } // int타입의 변수 i와 j를 선언하고 초기화
```

조건식

조건식의 값이 참(true)이면 반복을 계속하고, 거짓(false)이면 반복을 중단하고 for문을 벗어난다. for의 뜻이 ‘~하는 동안’이므로 조건식이 ‘참인 동안’ 반복을 계속한다고 생각하면 쉽다.

```
for(int i = 1;i<=10;i++) { ... } // 'i<=10'가 참인 동안 블럭{} 안의 문장들을 반복
```

조건식을 잘못 작성하면 블럭{} 내의 문장이 한 번도 수행되지 않거나 영원히 반복되는 무한반복에 빠지기 쉬우므로 주의해야 한다. 반복문에서 문제가 발생하는 대부분은 조건문이며, 블럭{} 안에 ‘System.out.println(i);’만 넣어도 간단히 확인 가능하다.

증감식

반복문을 제어하는 변수의 값을 증가 또는 감소시키는 식이다. 매 반복마다 변수의 값이 증감식에 의해 점차 변하다가 결국 조건식이 거짓이 되어 for문을 벗어나게 된다.

```
for(int i = 1;i<=10;i++) { ... } // 1부터 10까지 1씩 증가  
for(int i = 10;i>=1;i--) { ... } // 10부터 1까지 1씩 감소  
for(int i = 1;i<=10;i+=2) { ... } // 1부터 10까지 2씩 증가  
for(int i = 1;i<=10;i*=3) { ... } // 1부터 10까지 3배씩 증가
```

증감식도 쉼표','를 이용해서 두 문장 이상을 하나로 연결해서 쓸 수 있다.

```
for(int i=1, j=10;i<=10;i++, j--) { ... } // i는 1부터 10까지 1씩 증가하고
                                                // j는 10부터 1까지 1씩 감소한다.
```

지금까지 살펴본 이 3가지 요소는 생략할 수 있으며, 심지어 모두 생략 가능하다.

```
for(;;) { ... } // 초기화, 조건식, 증감식 모두 생략. 조건식은 참이 된다.
```

조건식이 생략된 경우, 참(true)으로 간주되어 무한 반복문이 된다. 대신 블럭{} 안에 if문을 넣어서 특정 조건을 만족하면 for문을 빠져 나오게 해야 한다. 무한 반복문에 대해서는 곧 자세히 다룰 것이다.

이제 다양한 예제를 통해 for문을 어떻게 활용하는지 알아보자.

▼ 예제 4-13/FlowEx13.java

```
class FlowEx13 {
    public static void main(String[] args) {
        for(int i=1;i<=5;i++)
            System.out.println(i); // i의 값을 출력한다.

        for(int i=1;i<=5;i++)
            System.out.print(i); // print()를 쓰면 가로로 출력된다.

        System.out.println();
    }
}
```

▼ 실행결과

1
2
3
4
5
12345

1부터 5까지 세로로 한번, 가로로 한번 출력하는 간단한 예제이다. 아래의 표를 보면 i의 값이 변화함에 따라 조건식의 결과가 어떻게 되는지 알 수 있다.

i	i <= 5
1	1 <= 5 → true 참
2	2 <= 5 → true 참
3	3 <= 5 → true 참
4	4 <= 5 → true 참
5	5 <= 5 → true 참
6	6 <= 5 → false 거짓, 반복종료

사실 i의 값은 1부터 6까지 변하지만, i값이 6일 때 조건식이 '6<=5'가 되고, 이 식의 결과는 거짓(false)이므로 for문을 벗어나기 때문에 6은 출력되지 않는다.

▼ 예제 4-14/FlowEx14.java

```
class FlowEx14 {
    public static void main(String[] args) {
        int sum = 0; // 합계를 저장하기 위한 변수.

        for(int i=1; i <= 10; i++) {
            sum += i; // sum = sum + i;
            System.out.printf("1부터 %d 까지의 합: %d%n",
                               i, sum);
        }
    } // main의 끝
}
```

▼ 실행결과

1부터 1 까지의 합: 1
1부터 2 까지의 합: 3
1부터 3 까지의 합: 6
1부터 4 까지의 합: 10
1부터 5 까지의 합: 15
1부터 6 까지의 합: 21
1부터 7 까지의 합: 28
1부터 8 까지의 합: 36
1부터 9 까지의 합: 45
1부터 10 까지의 합: 55

1부터 10까지의 합을 구하는 예제이다. 변수 i를 1부터 10까지 변화시키면서 i를 sum에 계속 더해서 누적시킨다. 그 과정을 출력했으므로 어렵지 않게 이해할 수 있을 것이다.

i	sum = sum + i
1	1 = 0 + 1
2	3 = 1 + 2
3	6 = 3 + 3
4	10 = 6 + 4
5	15 = 10 + 5
6	21 = 15 + 6
7	28 = 21 + 7
8	36 = 28 + 8
9	45 = 36 + 9
10	55 = 45 + 10

이와 같이 매 반복마다 변수들의 값이 어떻게 변하는지 적어보면 이해하기 쉬워진다. 반복회수가 많은 경우에는 일부 구간만 적어보면 된다.

▼ 예제 4-15/FlowEx15.java

```
class FlowEx15 {
    public static void main(String[] args) {
        for(int i=1,j=10;i<=10;i++,j--)
            System.out.printf("%d \t %d%n", i, j);
    }
}
```

▼ 실행결과

1	10
2	9
3	8
4	7
5	6
6	5
7	4
8	3
9	2
10	1

for문에 i와 j, 두 개의 변수로 i는 1부터 10까지 증가시키는 동시에, j는 10부터 1까지 감소시키며 출력한다. 하나의 for문에 두 개의 변수를 이용해서 출력하는 예를 보여준 것인데, 사실 아래처럼 하나의 변수로도 같은 결과를 얻을 수 있다.

```
for(int i=1;i<=10;i++) {
    System.out.printf("%d \t %d\n", i, 11 - i);
}
```

실행결과에서 i와 j의 관계를 살펴보면, i와 j를 더한 값이 11로 일정하다는 것을 알 수 있다. 이 사실을 이용하면 j는 '11 - i'가 된다. 그래서 j대신 '11 - i'라는 식을 사용할 수 있는 것이다.

$$\begin{aligned} i + j &= 11 \\ j &= 11 - i \end{aligned}$$

아무래도 for문에 사용되는 변수의 수가 적은 것이 더 효율적이고 간단하다.

▼ 예제 4-16/FlowEx16.java

```
class FlowEx16 {
    public static void main(String[] args) {
        System.out.println("i \t 2*i \t 2*i-1 \t i*i \t 11-i \t i%3 \t i/3");
        System.out.println("-----");
        for(int i=1;i<=10;i++)
            System.out.printf("%d \t %d \t %d \t %d \t %d \t %d \t %d\n",
                i, 2*i, 2*i-1, i*i, 11-i, i%3, i/3);
    }
}
```

▼ 실행결과						
i	2*i	2*i-1	i*i	11-i	i%3	i/3
1	2	1	1	10	1	0
2	4	3	4	9	2	0
3	6	5	9	8	0	1
4	8	7	16	7	1	1
5	10	9	25	6	2	1
6	12	11	36	5	0	2
7	14	13	49	4	1	2
8	16	15	64	3	2	2
9	18	17	81	2	0	3
10	20	19	100	1	1	3

변수 i의 값이 1부터 10까지 변하는 동안, 다양한 연산자로 짝수($2*i$), 홀수($2*i+1$), 제곱($i*i$), 역순($11-i$), 순환($i%3$), 반복($i/3$)을 구하는 방법을 보여준다.

나머지 연산자 '%'를 이용하면 특정 범위의 값들이 순환하면서 반복되는 결과를 얻을 수 있다는 것과 나누기 연산자 '/'는 같은 값이 연속적으로 반복되게 할 수 있다는 점을 눈여겨보자.

중첩 for문

if문 안에 또 다른 if문을 넣을 수 있는 것처럼, for문 안에 또 다른 for문을 포함시키는 것도 가능하다. 그리고 중첩의 횟수는 거의 제한이 없다. 중첩 for문을 설명하는데 별찍기 만큼 좋은 것은 없다. 일단 가장 쉬운 것부터 시작해보자.

만일 다음과 같이 5행 10열의 별 '*'을 찍으려면 어떻게 해야 할까?

```
*****
*****
*****
*****
*****
```

가장 간단한 방법은 다음과 같이 한 줄씩 5번 출력하는 것이다.

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

그러나 우리는 for문을 배웠으니, 다음과 같이 간단히 할 수 있다.

```
for(int i=1;i<=5;i++) {
    System.out.println("*****"); // 10개의 별을 출력한다.
}
```

'System.out.println("*****");' 역시 반복적인 일을 하는 문장이니 for문으로 바꿀 수 있다. 이 문장을 for문으로 바꾸면 다음과 같다.

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



```
for(int j=1;j<=10;j++) {
    System.out.print("*");
}
System.out.println();
```

왼쪽의 문장 대신 오른쪽의 for문을 이전의 for문에 넣으면 다음과 같이 두 개의 for문이 중첩된 형태가 된다.

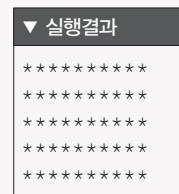
```
for(int i=1;i<=5;i++) {
    System.out.println("*****");
}
```



```
for(int i=1;i<=5;i++) {
    for(int j=1;j<=10;j++) {
        System.out.print("*");
    }
    System.out.println();
}
```

▼ 예제 4-17/FlowEx17.java

```
class FlowEx17 {
    public static void main(String[] args) {
        for(int i=1;i<=5;i++) {
            for(int j=1;j<=10;j++) {
                System.out.print("*");
            }
            System.out.println();
        }
    } // main의 끝
}
```



이번엔 다음과 같은 삼각형 모양의 별을 출력해보자.

```
*
**
***
****
*****

```

앞서 배운 바와 같이 가로로 출력하려면, println메서드 대신 print메서드로 출력하면 된다. 아래의 for문은 ‘*****’을 출력하고 줄 바꿈을 한다.

```
for(int j = 1;j <= 5;j++) {
    System.out.print("*"); // *****을 출력한다.
}
System.out.println(); // 줄 바꿈을 한다.
```

따라서 다음과 같이 코드를 작성하면, 우리가 원하는 결과를 얻을 수 있다.

```
for(int j = 1;j <= 1;j++) {System.out.print("*");} System.out.println(); // *
for(int j = 1;j <= 2;j++) {System.out.print("*");} System.out.println(); // **
for(int j = 1;j <= 3;j++) {System.out.print("*");} System.out.println(); // ***
for(int j = 1;j <= 4;j++) {System.out.print("*");} System.out.println(); // ****
for(int j = 1;j <= 5;j++) {System.out.print("*");} System.out.println(); // *****
```

위 문장들을 잘 보면 조건식의 숫자만 변할 뿐 나머지는 같다. 똑같은 내용이 반복되는데 반복문으로 간단히 처리할 방법이 없을까? 이럴 때는 한 문장의 조건식에 숫자 대신 변수 i를 넣고, 이 문장을 i의 값이 1부터 5까지 증가하는 for문 안에 넣으면 된다.

```
for(int i = 1;i <= 5;i++) {
    for(int j = 1;j <= i;j++) { System.out.print("*");} System.out.println();
}
```

위의 코드는 for문으로 1부터 5까지 출력하는 것과 근본적으로 같다.

이제 위의 코드로 우리가 원하는 결과를 얻을 수 있는지 직접 확인해 보자.

▼ 예제 4-18/FlowEx18.java

사용자로부터 라인의 수를 입력받아 별을 출력하도록 약간 수정하였다.

다음은 반복문의 단골손님인 구구단을 출력하는 예제이다. 예제4-16을 약간만 변경한 것이다.

▼ 예제 4-19/FlowEx19.java

```
class FlowEx19 {
    public static void main(String[] args) {
        for(int i=2;i<=9;i++) {
            for(int j=1;j<=9;j++) {
                System.out.printf("%d x %d = %d\n",i,j,i*j);
            }
        }
    } // main의 끝
}
```

▼ 실행결과
2 x 1 = 2
2 x 2 = 4
2 x 3 = 6
2 x 4 = 8
2 x 5 = 10
2 x 6 = 12
2 x 7 = 14
2 x 8 = 16
2 x 9 = 18
3 x 1 = 3
3 x 2 = 6
...중간생략...
9 x 7 = 63
9 x 8 = 72
9 x 9 = 81

반복문을 중첩해서 구구단을 출력하는 예제이다. 안쪽 for문은 하나의 단을 출력하고, 바깥쪽 for문은 안쪽 for문을 8번(2단~9단) 반복해서 출력한다. 실행결과를 보면 중첩된 for문이 어떤 순서로 반복되는지 알 수 있다. 바깥쪽 for문이 한번 반복될 때마다 안쪽 for문의 모든 반복이 끝나고서야 바깥쪽 for문의 다음 반복으로 넘어간다.

안쪽 for문의 내부에 문장이 하나뿐이므로 다음과 같이 괄호를 생략할 수 있다.

```
for(int i=2;i<=9;i++) {
    for(int j=1;j<=9;j++)
        System.out.printf("%d x %d = %d\n", i, j, i*j);
}
```

바깥쪽 for문에게는 안 쪽 for문 전체가 하나의 문장이므로 다음과 같이 바깥쪽 for문의 괄호{}도 생략이 가능하다.

```
for(int i=2;i<=9;i++)
    for(int j=1;j<=9;j++)
        System.out.printf("%d x %d = %d\n", i, j, i*j);
```

되도록이면 괄호{}를 사용하는 것이 좋지만 너무 많아도 복잡하므로 이처럼 간략하게 생략하는 것도 좋다.

▼ 예제 4-20/FlowEx20.java

```
class FlowEx20 {
    public static void main(String[] args) {
        for(int i=1;i<=3;i++)
            for(int j=1;j<=3;j++)
                for(int k=1;k<=3;k++)
                    System.out.println(""+i+j+k);
    } // main의 끝
}
```

▼ 실행결과
111
112
113
121
122
123
...중간생략...
323
331
332
333

3개의 반복문이 중첩되어 있는 경우 어떤 순서로 반복이 수행되는지를 눈으로 직접 확인할 수 있는 예제이다. 실행결과를 잘 살펴보자. 각 반복문이 3번씩 반복하므로 모두 27번 ($3 \times 3 \times 3 = 27$)이 반복된다. i, j, k가 각각 1, 2, 3일 때 식 " $+i+j+k$ "는 아래와 같이 계산된다.

$$\begin{aligned} "+i+j+k &\rightarrow "+\textcolor{red}{1}+2+3 \rightarrow "1"+2+3 \rightarrow "12"+3 \rightarrow "123" \end{aligned}$$

| 참고 | 위 과정이 이해되지 않는다면, p.55를 참고하자.

▼ 예제 4-21/FlowEx21.java

```
class FlowEx21 {
    public static void main(String[] args) {
        for(int i=1;i<=5;i++) {
            for(int j=1;j<=5;j++) {
                System.out.printf("[%d,%d]",i,j);
            }
            System.out.println();
        }
    } // main의 끝
}
```

▼ 실행결과

```
[1,1][1,2][1,3][1,4][1,5]
[2,1][2,2][2,3][2,4][2,5]
[3,1][3,2][3,3][3,4][3,5]
[4,1][4,2][4,3][4,4][4,5]
[5,1][5,2][5,3][5,4][5,5]
```

2중 반복문을 이용해서 i와 j를 1부터 5까지 1씩 증가시키면서 i와 j의 값을 쌍으로 출력하였다. 이 2중 for문 안에 if문을 넣어서 조건에 맞는 쌍만 출력함으로써 다양한 모양을 만들어 낼 수 있다. 다음의 예제를 보자.

▼ 예제 4-22/FlowEx22.java

```
class FlowEx22 {
    public static void main(String[] args) {
        for(int i=1;i<=5;i++) {
            for(int j=1;j<=5;j++) {
                if(i==j) {
                    System.out.printf("[%d,%d]", i, j);
                } else {
                    System.out.printf("%5c", ' ');
                }
            }
            System.out.println();
        }
    } // main의 끝
}
```

▼ 실행결과

```
[1,1]
[2,2]
[3,3]
[4,4]
[5,5]
```

바로 전 예제의 2중 for문에 if문을 넣어서 조건식 ‘*i==j*’를 만족하는 경우에만 i와 j의 값을 출력하고 나머지는 공백을 출력하였다.

i==j

```
[1,1][1,2][1,3][1,4][1,5]
[2,1][2,2][2,3][2,4][2,5]
[3,1][3,2][3,3][3,4][3,5]
[4,1][4,2][4,3][4,4][4,5]
[5,1][5,2][5,3][5,4][5,5]
```

if문의 조건식을 다르게 하면, 다양한 모양의 출력결과를 얻어낼 수 있다. 여기서 숫자쌍 대신 별‘*’을 찍으면 별찍기가 되므로, 이 예제를 이용하면 별찍기가 한결 쉬워진다. 출력되어야 할 별의 위치에 해당하는 숫자쌍을 모두 표시한 다음 이 숫자쌍들의 공통점을 찾아서 조건식으로 표현하면 된다.

향상된 for문(enhanced for statement)

JDK 5부터 배열과 컬렉션에 저장된 요소에 접근할 때 기존보다 편리한 방법으로 처리할 수 있도록 for문의 새로운 문법이 추가되었다.

```
for( 타입 변수명 : 배열 또는 컬렉션) { // 조건식이 없다!!!
    // 반복할 문장
}
```

위의 문장에서 타입은 배열 또는 컬렉션의 요소의 타입이어야 한다. 배열 또는 컬렉션에 저장된 값이 매 반복마다 하나씩 순서대로 읽혀서 변수에 저장된다. 그리고 반복문의 팔호{}내에서는 이 변수를 사용해서 코드를 작성한다.

```
int[] arr = {10,20,30,40,50};
```

배열 arr을 위와 같이 선언했을 때, 이 배열의 모든 요소를 출력하는 for문은 아래와 같다.

<pre>for(int i=0; i < arr.length; i++) { System.out.println(arr[i]); }</pre>		<pre>for(int tmp : arr) { System.out.println(tmp); }</pre>
---	--	--

위의 왼쪽은 일반적인 for문으로, 그리고 오른쪽은 향상된 for문으로 작성되었다. 두 for문은 동등하며, 향상된 for문은 골치거리인 조건식이 없어서 문제가 발생할 확률이 줄어든다. 그러나 향상된 for문은 배열이나 컬렉션에 저장된 요소들을 읽어오는 용도로만 사용할 수 있다는 제약이 있다.

작은 배열을 배우지 않아서 좀 어렵게 느껴질 수도 있는데, 지금은 이해하지 못해도 괜찮다. 이런 형태의 for문도 있다는 정도만 알아두어도 앞으로 진도를 나가면서 자연스럽게 익숙해질 것이다.

▼ 예제 4-23/FlowEx23.java

```
class FlowEx23 {
    public static void main(String[] args) {
        int[] arr = {10,20,30,40,50};
        int sum = 0;

        for(int i=0;i<arr.length;i++)
            System.out.printf("%d ", arr[i]);
        System.out.println();

        for(int tmp : arr) {
            System.out.printf("%d ", tmp);
            sum += tmp;
        }
        System.out.println();
        System.out.println("sum="+sum);
    } // main의 끝
}
```

▼ 실행결과

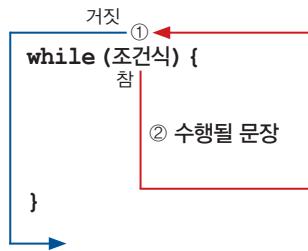
```
10 20 30 40 50
10 20 30 40 50
sum=150
```

2.2 while문

for문에 비해 while문은 구조가 간단하다. if문처럼 조건식과 블럭{}만으로 이루어져 있다. 다만 if문과 달리 while문은 조건식이 ‘참(true)인 동안’, 즉 조건식이 거짓이 될 때까지 블럭{} 내의 문장을 반복한다.

```
while (조건식) {
    // 조건식의 연산결과가 참(true)인 동안, 반복될 문장들을 적는다.
}
```

while문은 먼저 조건식을 평가해서 조건식이 거짓이면 문장 전체를 벗어나고, 참이면 블럭{} 내의 문장을 실행하고 다시 조건식으로 돌아간다. 조건식이 거짓이 될 때까지 이 과정이 계속 반복된다.



▲ 그림4-2 while문의 수행순서

- ① 조건식이 참(true)이면 블럭{}안으로 들어가고, 거짓(false)이면 while문을 벗어난다.
- ② 블럭{}의 문장을 실행하고 다시 조건식으로 돌아간다.

for문과 while문의 비교

1부터 10까지의 정수를 순서대로 출력하는 for문을 while문으로 변경하면 아래 오른쪽과 같다.

```
// 1.초기화, 2.조건식, 3.증감식
for(int i=1;i<=10;i++) {
    System.out.println(i);
}
```

```
int i=1; // 1.초기화

while(i<=10) { // 2.조건식
    System.out.println(i);
    i++; // 3.증감식
}
```

위의 두 코드는 완전히 동일하다. for문은 초기화, 조건식, 증감식을 한 곳에 모아 놓은 것 일 뿐, while문과 다르지 않다. 그래서 for문과 while문은 항상 서로 변환이 가능하다.

그래도 이 경우 for문이 더 간결하고 알아보기 쉽다. 만일 초기화나 증감식이 필요하지 않은 경우라면, while문이 더 적합할 것이다. 앞으로 소개할 예제들은 for문보다 while문이 더 적합한 것들이다.

while문의 조건식은 생략불가

한 가지 주의할 점은 for문과 달리 while문의 조건식은 생략할 수 없다는 것이다.

```
while( ) { // 에러. 조건식이 없음.  
    ...  
}
```

그래서 while문의 조건식이 항상 참이 되도록 하려면 반드시 true를 넣어야 한다. 다음의 두 반복문은 블럭{} 내의 문장을 무한 반복한다.

<pre>for(;;) { // 조건식이 항상 참 ... }</pre>		<pre>while(true) { // 조건식이 항상 참 ... }</pre>
---	--	---

| 참고 | 무한 반복문은 반드시 블럭{} 안에 조건문을 넣어서 특정 조건을 만족하면 무한 반복문을 벗어나도록 해야 한다.

▼ 예제 4-24/FlowEx24.java

```
class FlowEx24 {  
    public static void main(String[] args) {  
        int i= 5;  
  
        while(i--!=0) {  
            System.out.println(i + " - I can do it.");  
        }  
    } // main의 끝  
}
```

▼ 실행결과

```
4 - I can do it.  
3 - I can do it.  
2 - I can do it.  
1 - I can do it.  
0 - I can do it.
```

변수 i의 값만큼 블럭{}을 반복하는 예제이다. i의 값이 5이므로 ‘I can do it.’이 모두 5번 (4~0) 출력되었다. while문의 조건식은 i의 값이 0이 아닌 동안만 참이 되고, i의 값이 매 반복마다 1씩 감소하다 0이 되면 조건식은 거짓이 되어 while문을 벗어난다.

‘i--’가 후위형이므로 조건식이 평가된 후에 i의 값이 감소된다는 점에 주의하자. 그래서 실행결과에서 i의 값이 5~1이 아닌 4~0으로 출력된 것이다.

아직 이해가 잘 안 간다면 아래 오른쪽과 같이 조건식에서 감소 연산자‘--’를 분리해보자. 좀 더 이해하기 쉬운 코드가 될 것이다.

<pre>while(i--!=0) { System.out.println(i); }</pre>		<pre>while(i!=0) { i--; System.out.println(i); }</pre>
---	--	--

그러나 ‘--i’와 같은 전위형은 감소 연산자가 조건식에서 분리되면 while문을 벗어나기 때문에 전과 다른 문장이 된다. 아래 오른쪽의 while문은 반복을 거듭해도 i의 값이 감소하지 않아서 조건식이 결코 거짓이 될 수 없다.

```
while(--i!=0) {
    System.out.println(i);
}
```



```
--i; // while문을 벗어남
while(i!=0) {
    System.out.println(i);
}
```

예제4-24에서 'i--' 대신 '--i'를 사용한다면, 어떤 결과를 얻을지 예측해보고 예제를 수정하여 직접 결과를 확인해 보자.

▼ 예제 4-25/FlowEx25.java

```
class FlowEx25 {
    public static void main(String[] args) {
        int i=11;

        System.out.println("카운트 다운을 시작합니다.");
        while(i--!=0) {
            System.out.println(i);

            for(long j=0;j<5_000_000_000L;j++) {
                ;
            }
            System.out.println("GAME OVER");
        }
    }
}
```

▼ 실행결과

카운트 다운을 시작합니다.
10
9
8
7
6
5
4
3
2
1
0
GAME OVER

10부터 0까지 1씩 감소시켜가면서 출력을 하되, for문으로 매 출력마다 약간의 시간이 지연되도록 했다. 컴퓨터의 성능에 따라 지연되는 시간이 달라지므로 지연시간이 너무 짧거나 길면 반복횟수를 적절히 변경하자.

```
for(long j = 0;j < 5_000_000_000L;j++) {
    ;           // 아무런 내용도 없는 빈 문장
}
```

이 for문의 블럭{} 내에는 아무 일도 하지 않는 빈 문장';'하나만 있을 뿐 그 외에는 아무 것도 없다. 그저 조건식과 증감식을 2,000,000,000번 반복하면서 시간을 보낼 뿐이다.

블럭 내에 문장이 하나뿐일 때 괄호{}를 생략할 수 있으므로 위의 for문을 다음과 같이 바꿀 수 있다.

```
for(long j = 0;j < 5_000_000_000L;j++);
```

또는 아래와 같이 괄호{}를 써주고 빈 문장';'을 없앨 수 있다. 블럭{}안에는 문장을 넣지 않아도 되기 때문이다.

```
for(long j = 0;j < 5_000_000_000L;j++) {}
```

간혹 실수로 다음과 같이 코드를 작성하는 경우가 있는데, 이럴 때는 빈 문장 ';'만 for문에 속한 것으로 간주되어 블럭{}은 반복되지 않는다. 단 한번만 수행된다.

```
for (i = 1; i <= 10; i++) ;           // 빈 문장 ';'을 10번 반복한다.
{
    System.out.println("i = " + i);   // i = 11이 출력된다.
}
```

for문은 i의 값이 11일 때, 조건식이 거짓이 되어 반복을 마치므로 i의 값은 11이 출력된다.

▼ 예제 4-26/FlowEx26.java

```
import java.util.*;

class FlowEx26 {
    public static void main(String[] args) {
        int num = 0, sum = 0;
        System.out.print("숫자를 입력하세요. (예:12345)>");

        Scanner scanner = new Scanner(System.in);
        String tmp = scanner.nextLine(); // 화면을 통해 입력받은 내용을 tmp에 저장
        num = Integer.parseInt(tmp);     // 입력받은 문자열(tmp)을 숫자로 변환

        while (num != 0) {
            // num을 10으로 나눈 나머지를 sum에 더함
            sum += num % 10;           // sum = sum + num%10;
            System.out.printf("sum=%3d num=%d\n", sum, num);

            num /= 10;    // num = num / 10;  num을 10으로 나눈 값을 다시 num에 저장
        }

        System.out.println("각 자리수의 합:" + sum);
    }
}
```

▼ 실행결과

```
숫자를 입력하세요. (예:12345)>12345
sum = 5 num = 12345
sum = 9 num = 1234
sum = 12 num = 123
sum = 14 num = 12
sum = 15 num = 1
각 자리수의 합:15
```

사용자로부터 숫자를 입력받고, 이 숫자의 각 자리의 합을 구하는 예제이다. 실행결과에서 알 수 있듯이 12345를 입력하면, 결과는 15($1+2+3+4+5=15$)이다.

어떤 수를 10으로 나머지 연산하면 마지막 자리를 얻을 수 있다. 그리고 10으로 나누면 마지막 한자리가 제거된다.

$$\begin{aligned} 12345 \% 10 &\rightarrow 5 \\ 12345 / 10 &\rightarrow 1234 \end{aligned}$$

그래서 입력 받은 숫자 num을 0이 될 때까지 반복해서 10으로 나누가면서, 10으로 나머지 연산을 하면 num의 모든 자리를 얻을 수 있다. 이 과정을 단계별로 살펴보면 다음과 같다.

num	num%10	sum = sum + num%10 (sum+=num%10)	num = num / 10 (num/=10)
12345	5	5 = 0 + 5	1234 = 12345 / 10
1234	4	9 = 5 + 4	123 = 1234 / 10
123	3	12 = 9 + 3	12 = 123 / 10
12	2	14 = 12 + 2	1 = 12 / 10
1	1	15 = 14 + 1	0 = 1 / 10
0	–	–	–

num의 값은 ‘num/=10’에 의해 한자리씩 줄어들다가 0이 되면, while문의 조건식이 거짓이 되어 반복을 멈춘다.

▼ 예제 4-27/FlowEx27.java

```
class FlowEx27 {
    public static void main(String[] args) {
        int sum = 0;
        int i = 0;

        // i를 1씩 증가시켜서 sum에 계속 더해나간다.
        while((sum += ++i) <= 100) {
            System.out.printf("%d - %d%n", i, sum);
        }
    } // main의 끝
}
```

▼ 실행결과

```
1 - 1
2 - 3
3 - 6
4 - 10
5 - 15
6 - 21
7 - 28
8 - 36
9 - 45
10 - 55
11 - 66
12 - 78
13 - 91
```

1부터 몇까지 더하면 누적합계가 100을 넘지 않는 제일 큰 수가 되는지 알아내는 예제이다. 이전에 비해 조건식‘(sum+=++i) <=100’이 복잡한데, 아래의 두 식을 하나로 합쳐놓은 것이라고 생각하면 이해하기 쉬울 것이다.

```
sum += ++i      // i의 값을 1 증가시켜서 sum에 누적
sum <= 100      // sum의 값이 100보다 작거나 같은지 확인
```

식이 좀 복잡하긴 해도 예제가 전체적으로 간결하다. 자바를 배우다보면 이와 같은 식들이 자주 등장하므로 익숙해져야 한다.

▼ 예제 4-28/FlowEx28.java

```

import java.util.*;

class FlowEx28 {
    public static void main(String[] args) {
        int num;
        int sum = 0;
        boolean flag = true; // while문의 조건식으로 사용될 변수
        Scanner scanner = new Scanner(System.in);

        System.out.println("합계를 구할 숫자를 입력하세요. (끝내려면 0을 입력) ");

        while(flag) { // flag의 값이 true이므로 조건식은 참이 된다.
            System.out.print(">>");

            String tmp = scanner.nextLine();
            num = Integer.parseInt(tmp);

            if(num!=0) {
                sum += num; // num이 0이 아니면, sum에 더한다.
            } else {
                flag = false; // num이 0이면, flag의 값을 false로 변경.
            }
        } // while문의 끝
        System.out.println("합계:"+ sum);
    }
}

```

▼ 실행결과

```

합계를 구할 숫자를 입력하세요. (끝내려면 0을 입력)
>>100
>>200
>>300
>>400
>>0
합계:1000

```

사용자로부터 반복해서 숫자를 입력받다가 0을 입력하면 입력을 마치고 총 합을 출력하는 예제이다. while문의 조건식으로 변수 flag를 사용했는데, 처음엔 flag에 true를 저장해서 계속 반복을 하다가 사용자가 0을 입력하면 flag의 값을 false로 바꿔서 반복을 멈추게 한다.

```

while(flag) { // flag의 값이 true이므로 조건식은 참이 된다.
    System.out.print(">>");

    Scanner scanner = new Scanner(System.in);
    String tmp = scanner.nextLine();
    num = Integer.parseInt(tmp);

    if(num!=0) {
        sum += num; // num이 0이 아니면, sum에 더한다.
    } else {
        flag = false; // num이 0이면, flag의 값을 false로 변경.
    }
}

```

while문의 조건식이 상수는 아니지만, 변수가 고정된 값을 유지하므로 무한 반복문과 같다. 그래서 특정 조건을 만족할 때 반복을 멈추게 하는 if문이 반복문 안에 꼭 필요하다.

2.3 do-while문

do-while문은 while문의 변형으로 기본적인 구조는 while문과 같으나 조건식과 블럭{}의 순서를 바꿔놓은 것이다. 그래서 블럭{}을 먼저 실행한 후에 조건식을 평가한다.

while문은 조건식의 결과에 따라 블럭{}이 한 번도 실행되지 않을 수 있지만, do-while문은 최소한 한번은 실행될 것을 보장한다.

```
do {
    // 조건식의 연산결과가 참일 때 실행될 문장들을 적는다.
} while (조건식); ← 끝에 ';'을 잊지 않도록 주의
```

그리 많이 쓰이지는 않지만, 다음의 예제처럼 반복적으로 사용자의 입력을 받아서 처리할 때 유용하다.

▼ 예제 4-29/FlowEx29.java

```
import java.util.*;

class FlowEx29 {
    public static void main(String[] args) {
        int input = 0, answer = 0;

        answer = (int)(Math.random() * 100) + 1; // 1~100사이의 임의의 수를 저장
        Scanner scanner = new Scanner(System.in);

        do {
            System.out.print("1과 100사이의 정수를 입력하세요.>");
            input = scanner.nextInt();

            if(input > answer) {
                System.out.println("더 작은 수로 다시 시도해보세요.");
            } else if(input < answer) {
                System.out.println("더 큰 수로 다시 시도해보세요.");
            }
        } while(input!=answer);
        System.out.println("정답입니다.");
    }
}
```

▼ 실행결과

```
1과 100사이의 정수를 입력하세요.>50
더 작은 값으로 다시 시도해보세요.
1과 100사이의 정수를 입력하세요.>25
더 작은 값으로 다시 시도해보세요.
1과 100사이의 정수를 입력하세요.>12
더 큰 값으로 다시 시도해보세요.
1과 100사이의 정수를 입력하세요.>21
정답입니다.
```

`Math.random()`을 이용해서 1과 100사이의 임의의 수를 변수 `answer`에 저장하고, 이 값을 맞출 때까지 반복하는 예제이다. 사용자 입력인 `input`이 변수 `answer`의 값과 다른 동안 반복하다가 두 값이 같으면 반복을 벗어난다.

▼ 예제 4-30/FlowEx30.java

```
class FlowEx30 {
    public static void main(String[] args) {
        for(int i=1;i<=100;i++) {
            System.out.printf("i=%d ", i);

            int tmp = i;

            do {
                // tmp%10이 3의 배수인지 확인(0 제외)
                if(tmp%10%3==0 && tmp%10!=0)
                    System.out.print("짝");
                // tmp /= 10은 tmp = tmp / 10과 동일
            } while((tmp/=10)!=0);

            System.out.println();
        }
    } // main
}
```

▼ 실행결과
i=1
i=2
i=3 짝
i=4
i=5
i=6 짝
...
i=97 짝
i=98 짝
i=99 짹 짹
i=100

숫자 중에 3의 배수(3, 6, 9)가 포함되어 있으면, 포함된 개수만큼 박수를 치는 369게임을 1부터 100까지 출력하는 예제이다. 숫자의 각 자리를 확인해야 하므로 예제4-26에서처럼 10으로 나누고 10으로 나머지 연산을 한다. 그러나 이 작업은 변수 `i`에 직접하면 안되고 다른 변수에 저장해서 처리해야 한다. 변수 `i`는 for문의 반복을 제어하는데 사용하는 변수이기 때문이다.

```
int tmp = i; // i의 값을 다른 변수에 저장한다.
```

```
do {
    if(tmp%10%3 == 0 && tmp%10 != 0) // tmp%10이 3의 배수인지 확인(0은 제외)
        System.out.print("짝");
} while((tmp/=10) != 0);
```

예를 들어 `i`의 값이 97일 때, do-while문이 반복되는 동안 변수 `tmp`의 값은 식 '`tmp/=10`'에 의해 다음과 같이 변화된다.

tmp	tmp%10	tmp%10%3	tmp = tmp / 10
97	7	1	$9 = 97 / 10$
9	9	0	$0 = 9 / 10$

두 번째 반복에서만 if문의 조건식 ‘tmp%10%3==0 && tmp%10!=0’을 만족시키므로 ‘짝’이 한번 출력된다는 것을 알 수 있다. 위의 표에서 알 수 있듯이 tmp%10은 tmp의 끝자리이다. 식 ‘tmp%10%3==0’은 tmp의 끝자리가 3의 배수인지 확인하기 위한 것이다. 이 식은 tmp%10의 값이 0일 때도 참이므로, 식 ‘tmp%10!=0’을 ‘&&’로 연결해서 tmp%10의 값이 0인 경우를 제외해야 한다.

2.4 break문

앞서 switch문에서 break문에 대해 배웠던 것을 기억할 것이다. 반복문에서도 break문을 사용할 수 있는데, switch문에서 그랬던 것처럼, break문은 자신이 포함된 가장 가까운 반복문을 벗어난다. 주로 if문과 함께 사용되어 특정 조건을 만족하면 반복문을 벗어나게 한다.

▼ 예제 4-31/FlowEx31.java

```
class FlowEx31 {
    public static void main(String[] args) {
        int sum = 0;
        int i = 0;

        while(true) {
            if(sum > 100)
                • break;
            ++i;
            sum += i;
        } // end of while
        System.out.println("i=" + i);
        System.out.println("sum=" + sum);
    }
}
```

break문이 실행되면 이 부분은
실행되지 않고 while문을 완전히 벗어난다.

▼ 실행결과

```
i=14
sum=105
```

숫자를 1부터 계속 더해 나가서 몇까지 더하면 합이 100을 넘는지 알아내는 예제이다. i의 값을 1부터 1씩 계속 증가시켜며 더해서 sum에 저장한다. sum의 값이 100을 넘으면 if문의 조건식이 참이므로 break문이 실행되어 자신이 속한 반복문을 즉시 벗어난다.

이처럼 무한 반복문에는 조건문과 break문이 항상 같이 사용된다. 그렇지 않으면 무한히 반복되기 때문에 프로그램이 종료되지 않을 것이다.

| 참고 | sum += i;와 ++i; 두 문장을 sum += ++i;과 같이 한 문장으로 줄여 쓸 수 있다. 예제4-27과 비교해보자.

2.5 continue문

continue문은 반복문 내에서만 사용될 수 있으며, 반복이 진행되는 도중에 continue문을 만나면 반복문의 끝으로 이동하여 다음 반복으로 넘어간다. for문의 경우 증감식으로 이동하며, while문과 do-while문의 경우 조건식으로 이동한다.

continue문은 반복문 전체를 벗어나지 않고 다음 반복을 계속 수행한다는 점이 break 문과 다르다. 주로 if문과 함께 사용되어 특정 조건을 만족하는 경우에 continue문 이후의 문장들을 수행하지 않고 다음 반복으로 넘어가서 계속 진행하도록 한다.

전체 반복 중에 특정조건을 만족하는 경우를 제외할 때 유용하다.

▼ 예제 4-32/FlowEx32.java

```
class FlowEx32 {
    public static void main(String[] args) {
        for(int i=0;i <= 10;i++) {
            if (i%3==0)
                • continue; •
            System.out.println(i);
        }
    }
}
```

▼ 실행결과

```
1
2
4
5
7
8
10
```

1과 10사이의 숫자를 출력하되 그 중에서 3의 배수인 것은 제외하도록 하였다. i의 값이 3의 배수인 경우, if문의 조건식 ‘i%3==0’은 참이 되어 continue문에 의해 반복문의 블럭 끝’으로 이동된다. 즉, continue문과 반복문 블럭의 끝‘}’ 사이의 문장들을 건너뛰고 반복을 이어가는 것이다.

▼ 예제 4-33/FlowEx33.java

```
import java.util.*;

class FlowEx33 {
    public static void main(String[] args) {
        int menu = 0;
        int num = 0;

        Scanner scanner = new Scanner(System.in);
```

```

while(true) {
    System.out.println("(1) square");
    System.out.println("(2) square root");
    System.out.println("(3) log");
    System.out.print("원하는 메뉴(1~3)를 선택하세요. (종료:0) >");

    String tmp = scanner.nextLine(); // 화면에서 입력받은 내용을 tmp에 저장
    menu = Integer.parseInt(tmp); // 입력받은 문자열(tmp)을 숫자로 변환

    if(menu==0) {
        System.out.println("프로그램을 종료합니다.");
        break;
    } else if (!(1 <= menu && menu <= 3)) {
        System.out.println("메뉴를 잘못 선택하셨습니다. (종료는 0)");
        continue;
    }

    System.out.println("선택하신 메뉴는 " + menu + "번입니다.");
}
} // main의 끝
}

```

▼ 실행결과

```

(1) square
(2) square root
(3) log
원하는 메뉴(1~3)를 선택하세요. (종료:0) >4
메뉴를 잘못 선택하셨습니다. (종료는 0)
(1) square
(2) square root
(3) log
원하는 메뉴(1~3)를 선택하세요. (종료:0) >1
선택하신 메뉴는 1번입니다.
(1) square
(2) square root
(3) log
원하는 메뉴(1~3)를 선택하세요. (종료:0) >0
프로그램을 종료합니다.

```

메뉴를 보여주고 선택하게 하는 예제이다. 메뉴를 잘못 선택한 경우, continue문으로 다시 메뉴를 보여주고, 종료(0)를 선택한 경우 break문으로 반복을 벗어나 프로그램이 종료되도록 했다. 이 예제는 메뉴를 보여주고 선택하는 것을 반복하는 것 외에 별다른 기능이 없지만, 곧 이 예제를 좀 더 쓸 만한 것으로 발전시킬 것이다.

2.6 이름 붙은 반복문

break문은 근접한 단 하나의 반복문만 벗어날 수 있기 때문에, 여러 개의 반복문이 중첩된 경우에는 break문으로 중첩 반복문을 완전히 벗어날 수 없다. 이때는 중첩 반복문 앞에 이름을 붙이고 break문과 continue문에 이름을 지정해 줌으로써 하나 이상의 반복문을 벗어나거나 반복을 건너뛸 수 있다.

▼ 예제 4-34/FlowEx34.java

```
class FlowEx34 {
    public static void main(String[] args) {
        // for문에 Loop1이라는 이름을 붙였다.
        Loop1 : for(int i=2;i <=9;i++) {
            for(int j=1;j <=9;j++) {
                if(j==5)
                    • break Loop1;
                // • break;
                continue Loop1; •
                continue; •
                System.out.println(i+"*" + j + "=" + i*j);
            } // end of for j ←
            System.out.println();
        } // end of Loop1 ←
    }
}
```

▼ 실행결과

```
2*1=2
2*2=4
2*3=6
2*4=8
```

구구단을 출력하는 예제이다. 제일 바깥에 있는 for문에 Loop1이라는 이름을 붙였다. 그리고 j가 5일 때 break문을 수행하도록 했다. 반복문의 이름이 지정되지 않은 break문은 자신이 속한 하나의 반복문만 벗어날 수 있지만, 지금처럼 반복문에 이름을 붙여 주고 break문에 반복문 이름을 지정해주면 둘 이상의 반복문도 벗어날 수 있다.

j가 5일 때 반복문 Loop1을 벗어나게 했으므로 2단의 4번째 줄까지 밖에 출력되지 않았다. 만일 반복문의 이름이 지정되지 않은 break문이었다면 2단부터 9단까지 모두 네 줄씩 출력되었을 것이다.

예제에서는 ‘break Loop1;’ 아래의 세 문장들을 주석처리하였다. 이 네 문장(2개의 break문과 2개의 continue문) 중의 하나를 선택하고 선택한 문장을 제외한 나머지는 주석처리한 다음, 어떤 결과를 얻을지 예측하고 실행한 후에 예측한 결과와 비교해보자.

| 참고 | continue Loop1;과 같은 문장을 쓸 일은 거의 없을 테니 무시해도 좋다.

▼ 예제 4-35/FlowEx35.java

```

import java.util.*;

class FlowEx35 {
    public static void main(String[] args) {
        int menu = 0, num = 0;

        Scanner scanner = new Scanner(System.in);

        outer:
        while(true) {
            System.out.println("(1) square");
            System.out.println("(2) square root");
            System.out.println("(3) log");
            System.out.print("원하는 메뉴(1~3)를 선택하세요. (종료:0)>");

            String tmp = scanner.nextLine(); // 화면에서 입력받은 내용을 tmp에 저장
            menu = Integer.parseInt(tmp); // 입력받은 문자열(tmp)을 숫자로 변환

            if(menu==0) {
                System.out.println("프로그램을 종료합니다.");
                break;
            } else if (!(1<= menu && menu <= 3)) {
                System.out.println("메뉴를 잘못 선택하셨습니다. (종료는 0)");
                continue;
            }

            for(;;) {
                System.out.print("계산할 값을 입력하세요. (계산 종료:0, 전체 종료:99)>");
                tmp = scanner.nextLine(); // 화면에서 입력받은 내용을 tmp에 저장
                num = Integer.parseInt(tmp); // 입력받은 문자열(tmp)을 숫자로 변환

                if(num==0)
                    break; // 계산 종료. for문을 벗어난다.

                if(num==99)
                    break outer; // 전체 종료. for문과 while문을 모두 벗어난다.

                switch(menu) {
                    case 1:
                        System.out.println("result="+ num*num);
                        break;
                    case 2:
                        System.out.println("result="+ Math.sqrt(num));
                        break;
                    case 3:
                        System.out.println("result="+ Math.log(num));
                        break;
                }
            } // for(;;)
        } // while의 끝
    } // main의 끝
}

```

▼ 실행결과

```
(1) square
(2) square root
(3) log
원하는 메뉴 (1~3)를 선택하세요. (종료:0) >1
계산할 값을 입력하세요. (계산 종료:0, 전체 종료:99) >2
result=4
계산할 값을 입력하세요. (계산 종료:0, 전체 종료:99) >3
result=9
계산할 값을 입력하세요. (계산 종료:0, 전체 종료:99) >0
(1) square
(2) square root
(3) log
원하는 메뉴 (1~3)를 선택하세요. (종료:0) >2
계산할 값을 입력하세요. (종료:0, 전체종료:99) >4
result=2.0
계산할 값을 입력하세요. (종료:0, 전체종료:99) >99
```

이 예제는 예제4-33을 발전시킨 것으로 메뉴를 선택하면 해당 연산을 반복할 수 있게 for문을 추가하였다. 이 예제를 실행해서 다양하게 테스트한 후에 분석하면 더 이해하기 쉬울 것이다.

아래와 같이 반복문만 빼놓고 보면, 무한 반복문인 while문 안에 또 다른 무한 반복문인 for문이 중첩된 구조라는 것을 알 수 있다. while문은 메뉴를 반복해서 선택할 수 있게 해주고, for문은 선택된 메뉴의 작업을 반복해서 할 수 있게 해준다.

```
outer:
while(true) {
    ...
    for(;;) {
        ...
        if(num==0) // 계산 종료. for문을 벗어난다.
            • break;
        if(num==99) // 전체 종료. for문과 while문 모두 벗어난다.
            • break outer;
        ...
    } // for(;;)
} // while(true)
```

선택된 메뉴에서 0을 입력하면 break문으로 for문을 벗어나서 다른 메뉴를 선택할 수 있게 되고, 99를 입력하면 'break outer;'에 의해 for문과 while문 모두를 벗어나 프로그램이 종료된다.

| 참고 | 연습문제는 깃헙(<https://github.com/castello/javajungsuk4>)에서 PDF파일로 제공

Java

Programming Language

Chapter 05

배열
array

1. 배열(array)

1.1 배열(array)이란?

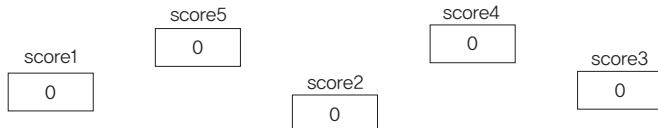
같은 타입의 여러 변수를 하나로 묶은 것을 ‘배열(array)’이라고 한다. 많은 양의 데이터를 저장하기 위해서, 예를 들어 10,000개의 데이터를 저장하기 위해 변수를 그만큼 선언해야 한다면 상상하는 것만으로도 상당히 곤혹스러울 것이다.

이런 경우에 배열을 사용하면 많은 양의 데이터를 손쉽게 다룰 수 있다.

“배열은 같은 타입의 여러 변수를 하나로 묶은 것”

여기서 중요한 것은 ‘같은 타입’이어야 한다는 것이다. 서로 다른 타입의 변수들로 구성된 배열은 만들 수 없다. 한 학급의 시험점수를 저장하고자 할 때가 배열을 사용하기 좋은 예이다. 만일 배열을 사용하지 않는다면 학생 5명의 점수를 저장하기 위해서 아래와 같이 5개의 변수를 선언해야 할 것이다.

```
int score1, score2, score3, score4, score5 ;
```



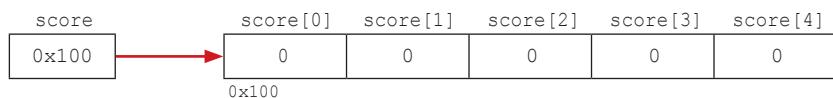
▲ 그림5-1 메모리에 생성된 변수들

변수 대신 배열을 이용하면 다음과 같이 간단히 처리할 수 있다. 변수의 선언과 달리 달리 야할 데이터의 수가 아무리 많아도 단지 배열의 길이만 바꾸면 된다.

```
int[] score = new int[5]; // 5개의 int 값을 저장할 수 있는 배열을 생성한다.
```

아래의 그림은 위의 코드가 실행되어 생성된 배열을 그림으로 나타낸 것이다. 값을 저장할 수 있는 공간은 score[0]부터 score[4]까지 모두 5개이며, 변수 score는 배열을 다루는데 필요한 참조 변수일 뿐 값을 저장하기 위한 공간은 아니다.

| 참고 | 배열처럼 여러 변수를 하나로 묶은 것을 객체라고 하며, 객체를 다루려면 참조 변수가 필요하다.



▲ 그림5-2 메모리에 생성된 배열

위의 그림에서 알 수 있듯이, 변수와 달리 배열은 각 저장공간이 연속적으로 배치되어 있다는 특징이 있다.

1.2 배열의 선언과 생성

배열을 선언하는 방법은 간단하다. 원하는 타입의 변수를 선언하고 변수 또는 타입에 배열을 의미하는 대괄호[]를 붙이면 된다. 대괄호[]는 타입 뒤에 붙여도 되고 변수이름 뒤에 붙여도 되는데, 저자의 경우 대괄호를 타입에 붙이는 쪽을 선호한다. 대괄호가 타입의 일부라고 보기 때문이다.

선언 방법	선언 예
타입[] 변수이름;	int[] score; String[] name;
타입 변수이름[];	int score[]; String name[];

▲ 표 5-1 배열의 선언 방법과 선언 예

배열의 생성

배열을 선언한 다음에 배열을 생성해야 한다. 배열을 선언하는 것은 단지 생성된 배열을 다루기 위한 참조변수를 위한 공간이 만들어질 뿐이고, 배열을 생성해야만 비로소 값을 저장할 공간이 만들어진다. 배열을 생성하려면 연산자 ‘new’와 함께 배열의 타입과 길이를 지정해 주어야 한다.

```
타입[] 변수이름;           // 배열을 선언(배열을 다루기 위한 참조 변수 선언)
변수이름 = new 타입[길이]; // 배열을 생성(실제 저장 공간을 생성)
```

아래의 코드는 ‘길이가 5인 int배열’을 생성한다.

```
int[] score;           // int타입의 배열을 다루기 위한 참조 변수 score 선언
score = new int[5];    // int타입의 값 5개를 저장할 수 있는 배열
```

다음과 같이 배열의 선언과 생성을 동시에 하면 간략히 한 줄로 할 수 있는데, 대부분의 경우 이렇게 한다.

```
int[] score = new int[5]; // 배열의 선언과 생성을 동시에
```

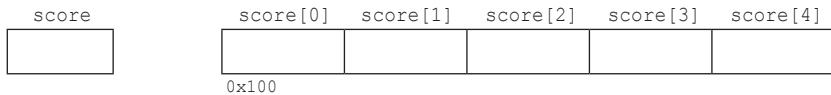
이제 배열의 선언과 생성 과정을 단계별로 그림과 함께 자세히 살펴보자.

1. int[] score;
int타입 배열 참조변수 score를 선언한다. 데이터를 저장할 수 있는 공간은 아직 마련되지 않았다.

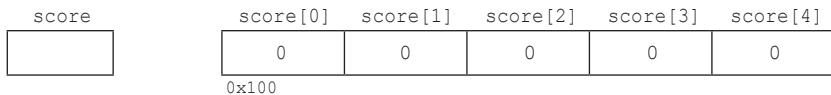


```
2. score = new int[5];
```

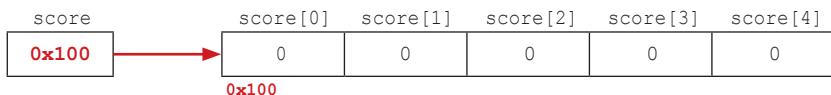
연산자 ‘new’에 의해서 메모리의 빈 공간에 5개의 int타입 값을 저장할 수 있는 공간이 마련된다.



그리고 각 배열 요소는 자동적으로 int의 기본값(default)인 0으로 초기화된다.



끝으로 대입 연산자‘=’에 의해 배열의 주소가 int배열 참조 변수 score에 저장된다.



| 참고 | 배열이 주소 0x100번지에 생성되었다고 가정한 그림이다.

참조 변수 score를 통해서만 배열에 값을 저장하거나 읽어 올 수 있다. 이 배열은 ‘길이가 5인 int배열’이며, 참조 변수의 이름을 따서 ‘배열 score’라고 부르자.

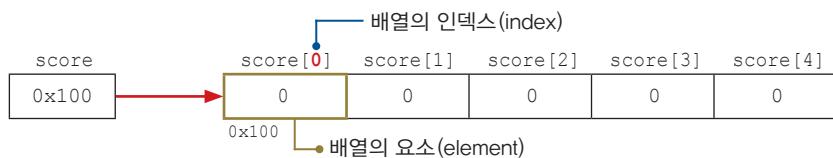
1.3 배열의 길이와 인덱스

생성된 배열의 각 저장공간을 ‘배열의 요소(element)’라고 하며, ‘배열이름[인덱스]’의 형식으로 배열의 요소에 접근한다. 인덱스(index)는 배열의 요소마다 붙여진 일련번호로 각 요소를 구별하는데 사용된다. 우리가 변수의 이름을 지을 때 score1, score2, score3과 같이 번호를 붙이는 것과 비슷하다고 할 수 있다. 다만 인덱스는 1이 아닌 0부터 시작한다.

인덱스(index)의 범위는 0 ~ 배열길이-1

예를 들어 길이가 5인 배열은 모두 5개의 요소(저장공간)를 가지며 인덱스의 범위는 1부터 5까지가 아닌 0부터 4까지, 즉 0, 1, 2, 3, 4가 된다.

```
int[] score = new int[5]; // 길이가 5인 int배열
```



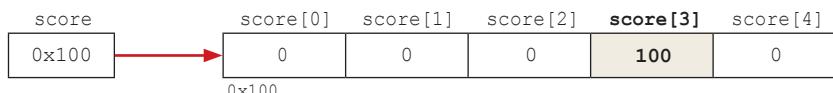
▲ 그림5-3 배열의 요소와 인덱스

배열에 값을 저장하고 읽어오는 방법은 변수와 같다. 변수이름 대신 ‘배열이름[인덱스]’를 사용한다는 점만 다르다.

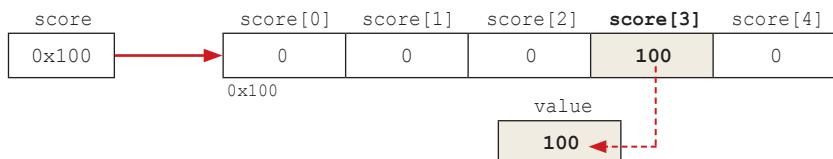
```
score[3] = 100;           // 배열 score의 4번째 요소에 100을 저장한다.
int value = score[3];    // 배열 score의 4번째 요소에 저장된 값을 읽어서 value에 저장
```

위의 코드를 단계별로 살펴보면 다음과 같다.

① `score[3] = 100;` // 배열 score의 4번째 요소에 100을 저장한다.



② `int value = score[3];` // 배열 score의 4번째 요소의 값을 읽어서 value에 저장.



배열의 다른 장점은 index로 상수 대신 변수나 수식도 사용할 수 있다는 것이다. 그래서 왼쪽의 코드를 오른쪽과 같이 for문을 이용해서 간단히 할 수 있다. 오른쪽 코드는 index로 상수대신 변수 i를 사용하고, for문으로 변수 i의 값을 0부터 4까지 증가시킨다.



for문의 제어변수 i는 배열의 index로 사용하기에 딱 알맞아서, 배열을 다룰 때 for문은 거의 필수적이다. 만일 아래와 같이 괄호[] 안에 수식이 포함된 경우, 이 수식이 먼저 계산된다. 그래야만 배열의 몇 번째 요소인지 알 수 있기 때문이다.

```
int tmp = score[i+1];
```

예를 들어 `score[3]`의 값이 100이고, 변수 i의 값이 2일 때, 위의 문장은 다음과 같이 계산된다.

```
int tmp = score[i+1];
→ int tmp = score[2+1];
→ int tmp = score[3];
→ int tmp = 100;
```

배열을 다룰 때 한 가지 주의할 점은 index의 범위를 벗어난 값을 index로 사용하지 않아야 한다는 것이다. 예를 들어 다음과 같이 길이가 5인 배열이 선언되어 있을 때, index의 범위는 0~4이다. 이 때, 이 범위를 벗어나는 값인 5를 index로 사용하면 안된다는 얘기다.

```
int[] score = new int[5]; // 길이가 5인 int 배열. index의 범위는 0~4
...
score[5] = 100; // index의 범위(0~4)를 벗어난 값을 index로 사용.
```

유효한 범위를 벗어난 값을 index로 사용하는 것은 가장 흔한 실수이다. 그러나 컴파일러는 이러한 실수를 걸러주지 못한다. 왜냐하면 배열의 index로 변수를 많이 사용하는데, 변수의 값은 실행 시에 대입되므로 컴파일러는 이 값의 범위를 확인할 수 없다.

그래서 유효한 범위의 값을 index로 사용하는 것은 전적으로 프로그래머의 책임이며, 유효하지 않은 값을 index로 사용하면, 무사히 컴파일을 마쳤더라도 실행 시에 에러 (ArrayIndexOutOfBoundsException)가 발생한다.

▼ 예제 5-1/ArrayEx.java

```
class ArrayEx {
    public static void main(String[] args) {
        int[] score = new int[5];
        int k = 1;

        score[0] = 50;
        score[1] = 60;
        score[k+1] = 70;    // score[2] = 70
        score[3] = 80;
        score[4] = 90;

        int tmp = score[k+2] + score[4]; // int tmp = score[3] + score[4]

        // for문으로 배열의 모든 요소를 출력한다.
        for(int i=0; i < 5; i++) {
            System.out.printf("score[%d]:%d%n", i, score[i]);
        }

        System.out.printf("tmp:%d%n", tmp);
        System.out.printf("score[%d]:%d%n", 7, score[7]); // index의 범위를 벗어난 값
    } // main
}
```

▼ 실행결과

```
score[0]:50
score[1]:60
score[2]:70
score[3]:80
score[4]:90
tmp:170
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 7
at ArrayEx1.main(ArrayEx.java:20)
```

[ebook – 샘플. 무료 공유] 자바의 정석 4판 Java 21 올컬러 2025. 7. 7 출시 seong.namkung@gmail.com

앞서 배운 내용을 직접 확인할 수 있는 간단한 예제이다. 배열 score는 길이가 5이므로 index의 범위가 0~4인데, 일부러 이 범위에 속하지 않는 7을 배열의 index로 지정해서 값을 출력해보았다. 컴파일 시에는 아무런 문제가 없지만, 실행 시에는 아래와 같은 에러가 발생하였다.

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 7
at ArrayEx.main(ArrayEx.java:20)
```

위 메시지는 배열의 인덱스가 유효한 범위를 넘었다는 뜻이다.

배열의 길이

앞서 배운 것과 같이 배열을 생성할 때 괄호[]안에 배열의 길이를 적어야 하는데, 배열의 길이는 배열 요소의 개수, 즉 값을 저장할 수 있는 공간의 개수다.

당연히 배열의 길이는 양의 정수이어야 하며 최대값은 int타입의 최대값, 약 20억이다. 실제로 이렇게 큰 배열을 생성하는 경우는 꽤 드물니까 배열의 길이는 거의 제약이 없다고 할 수 있다.

```
타입[] 배열이름 = new 타입[길이];
int[] arr     = new int[5];    // 길이가 5인 int배열
```

그런데 길이가 0인 배열도 생성이 가능하다. 길이가 0이라는 얘기는 값을 저장할 수 있는 공간이 하나도 없다는 뜻인데, 이런 배열을 생성하는 것이 무슨 의미가 있을까?

```
int[] arr = new int[0]; // 길이가 0인 배열도 생성이 가능하다!!!
```

그래도 프로그래밍을 하다보면 길이가 0인 배열이 필요한 상황이 있고 나름 유용하다. 앞으로 진도를 나가면서 길이가 0인 배열이 필요한 상황을 만나게 될 것이므로 지금은 자바에서 ‘배열의 길이가 0일 수도 있다.’는 것만 기억하자.

배열의 길이는 int범위의 양의 정수(0 포함)이어야 한다.

배열이름.length

배열의 길이는 ‘배열이름.length’를 통해서 알 수 있다. 아래의 코드에서 배열 arr의 길이가 5이므로 arr.length의 값도 5이다.

```
int[] arr = new int[5]; // 길이가 5인 int배열
int tmp   = arr.length; // arr.length의 값은 5이고 tmp에 5가 저장된다.
```

배열은 한번 생성하면 길이를 변경할 수 없기 때문에, ‘배열이름.length’는 상수다. 즉, 값을 읽을 수만 있을 뿐 변경할 수 없다.

[ebook – 샘플. 무료 공유] 자바의 정석 4판 Java 21 올컬러 2025. 7. 7 출시 seong.namkung@gmail.com

```
int[] arr = new int[5];
arr.length = 10;           // 에러. 배열의 길이는 변경할 수 없음.
```

아래의 코드는 배열의 각 요소를 for문을 이용해서 출력한다. 여기서 배열 score의 길이는 6이며, index의 범위는 0~5이다.

```
int[] score = new int[6]; // 배열의 길이는 6이고 index 범위는 0~5

for (int i=0; i < 6; i++) {
    System.out.println(score[i]);
}
```

이 때 코드를 다음과 같이 변경하여 배열의 길이를 줄인다면, 유효한 index의 범위는 0~4가 된다.

```
int[] score = new int[5]; // 배열의 길이를 6에서 5로 변경.

for (int i=0; i < 6; i++) { // 실수로 조건식을 변경하지 않음
    System.out.println(score[i]); // 에러 발생!!!
}
```

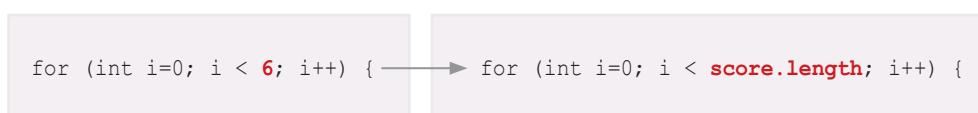
배열의 길이가 변경되었으니 for문의 조건식도 변경해야 한다. 그런데 이것을 잊고 실행하면 for문은 배열의 유효한 index 범위인 0~4를 넘어 0부터 5까지 반복하기 때문에 6번 째 반복에서 예외(index가 유효한 범위를 벗어났다는 에러)가 발생하여 비정상적으로 종료될 것이다.

그래서 for문의 조건식에 배열의 길이를 대신 ‘배열이름.length’를 적는 것이 좋다.

```
int[] score = new int[5]; // 배열의 길이를 6에서 5로 변경

for (int i=0; i < score.length; i++) { // 조건식을 변경하지 않아도 됨
    System.out.println(score[i]);
}
```

‘배열이름.length’는 배열의 길이가 변경되면 자동적으로 같이 변경되므로, 배열과 함께 사용되는 for문의 조건식을 일일이 변경해주지 않아도 된다.



for문뿐만 아니라, 모든 경우에 배열의 길이를 직접 적어주는 것보다 ‘배열이름.length’를 사용하는 것이 코드의 관리가 쉽고 에러가 발생할 확률이 적어진다.

배열의 길이 변경하기

배열은 한번 선언되고 나면 길이를 변경할 수 없다고 배웠는데, 그렇다면 배열에 저장할 공간이 부족한 경우에는 어떻게 해야 할까? 더 큰 길이의 새로운 배열을 생성한 다음, 기존의 배열에 저장된 값들을 새로운 배열에 복사하면 된다.

배열의 길이를 변경하는 방법 :

1. 더 큰 배열을 새로 생성한다.
2. 기존 배열의 내용을 새로운 배열에 복사한다.

이런 작업들은 비용이 많이 들기 때문에, 처음부터 배열의 길이를 넉넉하게 잡아줘서 새로 배열을 생성해야하는 일이 가능한 적게 발생하도록 해야 한다. 그렇다고 배열의 길이를 너무 크게 잡으면 메모리를 낭비하게 되므로, 기존의 2배의 길이로 생성하는 것이 보통이다.

보다 자세한 내용은 ‘1.5 배열의 복사’에서 다룬다.

1.4 배열의 초기화

배열은 생성과 동시에 자동으로 자신의 타입에 해당하는 기본값으로 초기화되므로 배열을 사용하기 전에 따로 초기화를 해주지 않아도 되지만, 원하는 값을 저장하려면 아래와 같이 각 요소마다 값을 지정해 줘야한다.

```
int[] score = new int[5];           // 길이가 5인 int형 배열을 생성한다.
score[0] = 50;                      // 각 요소에 직접 값을 저장한다.
score[1] = 60;
score[2] = 70;
score[3] = 80;
score[4] = 90;
```

배열의 길이가 큰 경우에는 이렇게 요소 하나하나에 값을 지정하기 보다는 for문을 사용하는 것이 좋다. 위의 코드를 for문을 이용해서 바꾸면 다음과 같다.

```
int[] score = new int[5];           // 길이가 5인 int형 배열을 생성한다.

for(int i=0; i < score.length; i++)
    score[i] = i * 10 + 50;
```

그러나 for문으로 배열을 초기화하려면, 저장하려는 값에 일정한 규칙이 있어야만 가능하기 때문에 자바에서는 다음과 같이 배열을 간단히 초기화 할 수 있는 방법을 제공한다.

```
int[] score = new int[]{ 50, 60, 70, 80, 90}; // 배열의 생성과 초기화를 동시에
```

저장할 값을 팔호{} 안에 쉼표로 구분해서 나열하면 되며, 팔호{} 안의 값의 개수에 의해 배열의 길이가 자동으로 결정되기 때문에 팔호[] 안에 배열의 길이는 적지 않는다.

```
int[] score = new int[] { 50, 60, 70, 80, 90};  
int[] score = { 50, 60, 70, 80, 90}; // new int[]를 생략할 수 있음
```

심지어는 위와 같이 ‘new 타입[]’을 생략하여 코드를 더 간단히 할 수도 있다. 아무래도 생략된 형태의 코드가 더 간단하므로 자주 사용된다. 다만 다음과 같이 배열의 선언과 생성을 따로 하는 경우에는 생략할 수 없다는 것만 주의하자.

```
int[] score;  
score = new int[] { 50, 60, 70, 80, 90}; // OK  
score = { 50, 60, 70, 80, 90}; // 에러. new int[]를 생략할 수 없음
```

또 다른 예로, 아래와 같이 매개변수로 int배열을 받는 add메서드가 정의되어 있고 이 메서드를 호출해야 할 경우 역시 ‘new 타입[]’을 생략할 수 없으며, 이유는 같다.

```
int add(int[] arr) { /* 내용 생략 */ } // add메서드  
int result = add(new int[] { 100, 90, 80, 70, 60}); // OK  
int result = add({ 100, 90, 80, 70, 60}); // 에러. new int[]를 생략할 수 없음
```

그리고 팔호{} 안에 아무 것도 넣지 않으면, 길이가 0인 배열이 생성된다. 배열을 가리키는 참조변수를 null대신 길이가 0인 배열로 초기화하기도 한다. 아래의 세 문장은 모두 길이가 0인 배열을 생성한다.

```
int[] score = new int[0]; // 길이가 0인 배열  
int[] score = new int[]{}; // 길이가 0인 배열  
int[] score = {} ; // 길이가 0인 배열, new int[]가 생략됨
```

배열의 출력

배열을 초기화할 때 for문을 사용하듯이, 배열에 저장된 값을 확인할 때도 다음과 같이 for문을 사용하면 된다.

```
int[] iArr = { 100, 95, 80, 70, 60 };  
// 배열의 요소를 순서대로 하나씩 출력  
for(int i=0;i < iArr.length; i++) {  
    System.out.println(iArr[i]);  
}
```

println메서드는 출력 후에 줄 바꿈을 하므로, 여러 줄에 출력되어 보기 불편할 때가 있다. 그럴 때는 다음과 같이 출력 후에 줄 바꿈을 하지 않는 print메서드를 사용하자.

[ebook – 샘플. 무료 공유] 자바의 정석 4판 Java 21 올컬러 2025. 7. 7 출시 seong.namkung@gmail.com

```
int[] iArr = { 100, 95, 80, 70, 60 };

for(int i=0;i < iArr.length; i++) {
    System.out.print(iArr[i]+","); // 각 요소간의 구별을 위해 쉼표를 넣는다.
}
System.out.println(); // 다음 출력이 바로 이어지지 않도록 줄 바꿈을 한다.
```

더 간단한 방법은 ‘Arrays.toString(배열이름)’메서드를 사용하는 것이다. 이 메서드는 배열의 모든 요소를 [첫번째 요소, 두번째 요소, ...]와 같은 형식의 문자열로 만들어서 반환한다. 이 메서드와 관련된 자세한 내용은 진도를 나가면서 자연스럽게 알게 될 것들이므로 지금은 이 메서드를 이용하면 배열의 내용을 쉽게 확인할 수 있다는 것만 알아두자.

| 참고 | Arrays.toString()을 사용하려면, ‘import java.util.*;’를 추가해야 한다.

```
int[] iArr = { 100, 95, 80, 70, 60 };
// 배열 iArr의 모든 요소를 출력한다. [100, 95, 80, 70, 60]이 출력된다.
System.out.println(Arrays.toString(iArr));
```

만일 iArr의 값을 바로 출력하면 어떻게 될까? iArr은 참조변수니까 변수에 저장된 값, 즉 ‘배열의 주소’가 출력될 것으로 생각했다면 지금까지 잘 이해하고 있는 것이다.

그러나 이러한 예상과는 달리 ‘타입@주소’의 형식으로 출력된다. ‘[I’는 1차원 int배열이라는 의미이고, '@’뒤에 나오는 16진수는 배열의 주소인데 실제 주소가 아닌 내부 주소이다. 이 내용은 지금 진도와 맞지 않는 내용이므로 가볍게 참고만 하고, 배열을 가리키는 참조변수를 출력해봐야 별로 얻을 정보가 없다는 정도만 기억하자.

```
// 배열을 가리키는 참조변수 iArr의 값을 출력한다.
System.out.println(iArr); // [I@14318bb와 같은 형식의 문자열이 출력된다.
```

예외적으로 char배열은 println메서드로 출력하면 각 요소가 구분자없이 그대로 출력되는데, 이것은 println메서드가 char배열일 때만 이렇게 동작하도록 작성되었기 때문이다.

```
char[] chArr = { 'a', 'b', 'c', 'd' };
System.out.println(chArr); // abcd가 출력된다.
```

▼ 예제 5-2/ArrayEx2.java

```
import java.util.*; // Arrays.toString()을 사용하기 위해 추가

class ArrayEx2 {
    public static void main(String[] args) {
        int[] iArr1 = new int[10];
        int[] iArr2 = new int[10];
        int[] iArr3 = new int[]{100, 95, 80, 70, 60};
        int[] iArr3 = {100, 95, 80, 70, 60};
        char[] chArr = {'a', 'b', 'c', 'd'};
```

[ebook – 샘플. 무료 공유] 자바의 정석 4판 Java 21 올컬러 2025. 7. 7 출시 seong.namkung@gmail.com

```

for(int i=0; i < iArr1.length; i++ ) {
    iArr1[i] = i + 1; // 1~10의 숫자를 순서대로 배열에 넣는다.
}

for(int i=0; i < iArr2.length; i++ ) {
    iArr2[i] = (int)(Math.random()*10) + 1; // 1~10의 값을 배열에 저장
}

// 배열에 저장된 값들을 출력한다.
for(int i=0; i < iArr1.length;i++) {
    System.out.print(iArr1[i]+",");
}
System.out.println();
System.out.println(Arrays.toString(iArr2));
System.out.println(Arrays.toString(iArr3));
System.out.println(Arrays.toString(chArr));
System.out.println(iArr3);
System.out.println(chArr);
}
}

```

▼ 실행결과

```

1,2,3,4,5,6,7,8,9,10,
[3, 4, 8, 10, 1, 10, 6, 2, 7, 1]
[100, 95, 80, 70, 60]
[a, b, c, d]
[I@14318bb ← 실행할 때 마다 달라질 수 있다.
abcd

```

1.5 배열의 복사

배열은 한번 생성하면 그 길이를 변경할 수 없기 때문에 더 많은 저장공간이 필요하면 보다 큰 배열을 새로 만들고 이전 배열로부터 내용을 복사해야 한다.

배열을 복사하는 방법은 두 가지가 있는데, 먼저 for문을 이용해서 배열을 복사하는 방법은 다음과 같다.

```

int[] arr = {1,2,3,4,5}; // arr.length는 5
...
int[] tmp = new int[arr.length*2]; // 1. arr보다 길이가 2배인 배열 생성
for(int i=0; i < arr.length;i++) // 2. arr의 내용을 tmp에 복사
    tmp[i] = arr[i]; // arr[i]의 값을 tmp[i]에 저장
arr = tmp; // 3. 참조변수 arr이 새로운 배열을 가리키게 한다.

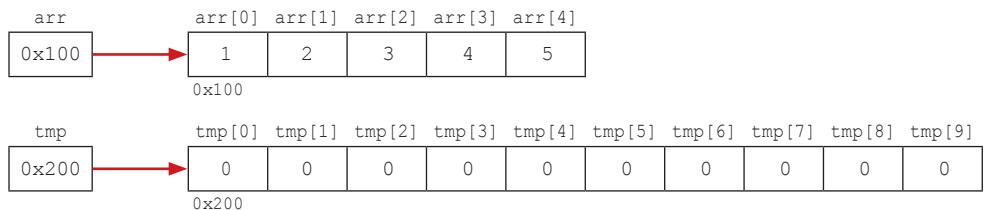
```

이 작업은 꽤 비용이 많이 들기 때문에, 처음부터 배열의 길이를 넉넉하게 잡아줘서 새로 배열을 생성해야하는 상황이 가능한 적게 발생하도록 해야 한다. 그렇다고 배열의 길이를 너무 크게 잡으면 메모리를 낭비하게 되므로, 위의 코드에서처럼 기존의 2배정도의 길이로 배열을 생성하는 것이 좋다.

이 과정을 단계별로 그림과 함께 살펴보면 다음과 같다.

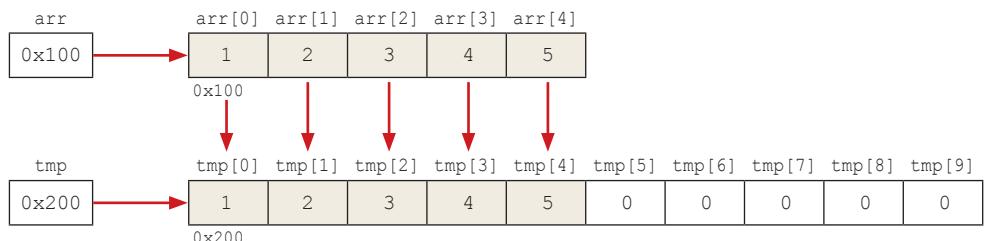
1. 배열 arr의 길이인 arr.length의 값이 5이므로 길이가 10인 int배열 tmp가 생성되고, 배열 tmp의 각 요소는 int의 기본값인 0으로 초기화된다.

```
int[] tmp = new int[arr.length*2];
→ int[] tmp = new int[5*2];
→ int[] tmp = new int[10];
```



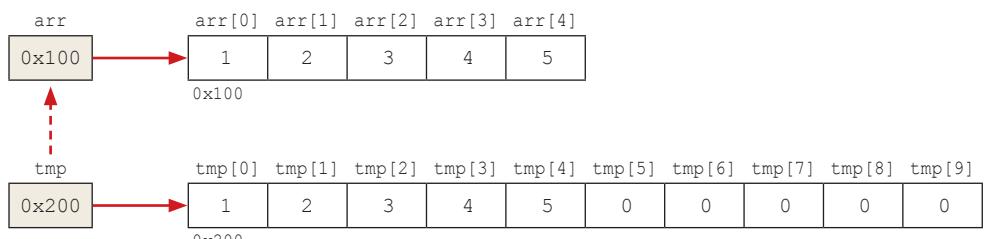
2. for문을 이용해서 배열 arr의 모든 요소에 저장된 값을 하나씩 배열 tmp에 복사한다.

```
for(int i=0; i < arr.length;i++)
    tmp[i] = arr[i];
```



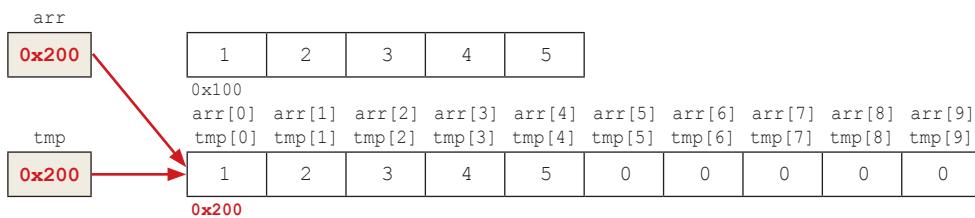
3. 참조변수 arr에 참조변수 tmp의 값을 저장한다. arr의 값은 0x100에서 0x200으로 바뀌고, arr은 배열 tmp를 가리키게 된다.

```
arr = tmp; // 변수 tmp에 저장된 값을 변수 arr에 저장한다.
```



결국 참조변수 arr과 tmp는 같은 배열을 가리키게 된다. 즉, 배열 arr과 배열 tmp는 이름만 다를 뿐 동일한 배열이다. 그리고 전에 arr이 가리키던 배열은 더 이상 사용할 수 없게 된다.

| 참고 | 배열은 참조 변수를 통해서만 접근할 수 있으므로, 참조 변수가 없는 배열은 사용할 수 없다. 이렇게 쓸모없게 된 배열은 JVM의 가비지 컬렉터(garbage collector)에 의해 자동으로 메모리에서 제거된다.



아직 참조 변수에 대해서 자세히 배우지 않았기 때문에 이해하기 어려울 수 있다. 지금은 배열의 길이를 변경할 때 이런 식으로 처리한다는 정도만 이해해도 충분하다.

▼ 예제 5-3/ArrayEx3.java

```
class ArrayEx3 {
    public static void main(String[] args) {
        int[] arr = new int[5];

        // 배열 arr에 1~5를 저장한다.
        for(int i=0; i < arr.length;i++)
            arr[i] = i + 1;

        System.out.println("[변경전]");
        System.out.println("arr.length:"+arr.length);
        for(int i=0; i < arr.length;i++)
            System.out.println("arr["+i+"]:"+arr[i]);

        int[] tmp = new int[arr.length*2];

        // 배열 arr에 저장된 값들을 배열 tmp에 복사한다.
        for(int i=0; i < arr.length;i++)
            tmp[i] = arr[i];

        arr = tmp; // tmp에 저장된 값을 arr에 저장한다.

        System.out.println("[변경후]");
        System.out.println("arr.length:"+arr.length);
        for(int i=0; i < arr.length;i++)
            System.out.println("arr["+i+"]:"+arr[i]);
    }
}
```

▼ 실행결과
[변경전]
arr.length:5
arr[0]:1
arr[1]:2
arr[2]:3
arr[3]:4
arr[4]:5
[변경후]
arr.length:10
arr[0]:1
arr[1]:2
arr[2]:3
arr[3]:4
arr[4]:5
arr[5]:0
arr[6]:0
arr[7]:0
arr[8]:0
arr[9]:0

System.arraycopy()를 이용한 배열의 복사

for문 대신 System 클래스의 `arraycopy()`를 사용하면 보다 간단하고 빠르게 배열을 복사할 수 있다. for문은 배열의 요소 하나하나에 접근해서 복사하지만, `arraycopy()`는 지정된 범위의 값들을 한 번에 통째로 복사한다. 각 요소들이 연속적으로 저장되어 있다는 배열의 특성때문에 이렇게 처리하는 것이 가능한 것이다.

배열의 복사는 for문보다 `System.arraycopy()`를 사용하는 것이 빠르다.

이전 예제에서 배열의 복사에 사용된 for문을 arraycopy()로 바꾸면 다음과 같다.

```
for(int i = 0; i < num.length;i++) { newNum[i] = num[i]; }
```



```
System.arraycopy(num, 0, newNum, 0, num.length);
```

arraycopy()를 호출할 때는 어느 배열의 몇 번째 요소에서 어느 배열로 몇 번째 요소로 몇 개의 값을 복사할 것인지 지정해줘야 하는데, 다음과 같이 생각하면 이해하기 쉽다.

```
System.arraycopy(num, 0, newNum, 0, num.length);
```

num[0]에서 newNum[0]으로 num.length개의 데이터를 복사

배열 num의 내용을 배열 newNum으로, 배열 num의 첫 번째 요소(num[0])부터 시작해서 num.length개의 데이터를 newNum의 첫 번째 요소(newNum[0])에 복사한다.

이때 복사하려는 배열의 위치가 적절하지 못하여 복사하려는 내용보다 여유 공간이 적으면 에러(ArrayIndexOutOfBoundsException)가 발생한다.

| 참고 | Arrays.copyOf()나 Arrays.copyOfRange()로도 배열을 복사할 수 있다. p.654

▼ 예제 5-4/ArrayEx4.java

```
class ArrayEx4 {
    public static void main(String[] args) {
        char[] abc = { 'A', 'B', 'C', 'D' };
        char[] num = { '0', '1', '2', '3', '4', '5', '6', '7', '8', '9' };
        System.out.println(abc);
        System.out.println(num);

        // 배열 abc와 num을 붙여서 하나의 배열(result)로 만든다.
        char[] result = new char[abc.length+num.length];
        System.arraycopy(abc, 0, result, 0, abc.length);
        System.arraycopy(num, 0, result, abc.length, num.length);
        System.out.println(result);

        // 배열 abc를 배열 num의 첫 번째 위치부터 배열 abc의 길이만큼 복사
        System.arraycopy(abc, 0, num, 0, abc.length);
        System.out.println(num);

        // number의 인덱스6 위치에 3개를 복사
        System.arraycopy(abc, 0, num, 6, 3);
        System.out.println(num);
    }
}
```

▼ 실행결과
ABCD
0123456789
ABCD0123456789
ABCD456789
ABCD45ABC9

다른 배열과 달리 char배열은 for문을 사용하지 않고도 print()나 println()으로 배열에 저장된 모든 문자를 출력할 수 있다.

1.6 배열의 활용

지금까지 배열의 기본적인 내용은 모두 살펴보았는데, 아직 배열을 어떻게 활용해야 할지 감이 잘 오지 않을 것이다. 이제 다양한 예제를 통해서 배열을 어떻게 활용하는지 배워보자.

[예제5-5] 총합과 평균	배열의 모든 요소를 더해서 총합과 평균을 구한다.
[예제5-6] 최대값과 최소값	배열의 요소 중에서 제일 큰 값과 제일 작은 값을 찾는다.
[예제5-7,8] 섞기(shuffle)	배열의 요소의 순서를 반복해서 바꾼다.(카드섞기, 로또번호생성)
[예제5-9] 임의의 값으로 배열 채우기	연속 또는 불연속적인 값들로 배열을 초기화 한다.
[예제5-10] 정렬하기(sort)	오름차순, 내림차순으로 배열을 정렬
[예제5-11] 빈도수 구하기	배열에 어떤 값이 몇 개 저장되어 있는지 세어서 보여준다.

이 외에도 배열의 활용방법은 무궁무진하지만, 이 예제들만 잘 배워두면 앞으로 배울 더 높은 수준의 활용을 이해하는데 별 어려움이 없을 것이다.

▼ 예제 5-5/ArrayEx5.java

```
class ArrayEx5 {
    public static void main(String[] args) {
        int sum = 0;          // 총점을 저장하기 위한 변수
        float average = 0f; // 평균을 저장하기 위한 변수

        int[] score = {100, 88, 100, 100, 90};

        for (int i=0; i < score.length; i++ ) {
            sum += score[i]; ●
        }
        average = sum / (float)score.length; // 계산결과를 float로 얻기 위해서 형변환

        System.out.println("총점 : " + sum);
        System.out.println("평균 : " + average);
    }
}
```

반복문으로 배열에 저장되어 있는 값을 모두 더한다.

▼ 실행결과

총점 : 478
평균 : 95.6

for문을 이용해서 배열에 저장된 값을 모두 더한 결과를 배열의 개수로 나누어서 평균을 구하는 예제이다. 평균을 구하기 위해 전체 합을 배열의 길이인 score.length로 나누었다.

이 때 int와 int 간의 연산은 int를 결과로 얻기 때문에 정확한 평균값을 얻지 못하므로 score.length를 float로 변환후에 나누었다.

$$\begin{aligned} 478 / 5 &\rightarrow 95 \\ 478 / (\textbf{float})5 &\rightarrow 478 / 5.0f \rightarrow 478.0f / 5.0f \rightarrow \mathbf{95.6f} \end{aligned}$$

[ebook – 샘플. 무료 공유] 자바의 정석 4판 Java 21 올컬러 2025. 7. 7 출시 seong.namkung@gmail.com

▼ 예제 5-6/ArrayEx6.java

```
class ArrayEx6 {
    public static void main(String[] args) {
        int[] score = { 79, 88, 91, 33, 100, 55, 95};

        int max = score[0]; // 배열의 첫 번째 값으로 최대값을 초기화
        int min = score[0]; // 배열의 첫 번째 값으로 최소값을 초기화

        for(int i=1; i < score.length;i++) {
            if(score[i] > max) {
                max = score[i];
            } else if(score[i] < min) {
                min = score[i];
            }
        } // end of for

        System.out.println("최대값 :" + max);
        System.out.println("최소값 :" + min);
    } // end of main
} // end of class
```

배열의 두 번째 요소부터
읽기 위해서 변수 i의 값을
1로 초기화 했다.

▼ 실행결과

최대값 :100
최소값 :33

배열에 저장된 값 중에서 최대값과 최소값을 구하는 예제이다. 배열의 첫 번째 요소 ‘score[0]’의 값으로 변수 max와 min을 초기화 하였다.

그 다음 반복문을 통해서 배열의 두 번째 요소 ‘score[1]’부터 max와 비교하기 시작한다. 만일 배열에 담긴 값이 max에 저장된 값보다 크다면, 이 값을 max에 저장한다.

이런 식으로 배열의 마지막 요소까지 비교하고 나면 max에는 배열에 담긴 값 중에서 최대값이 저장된다. 최소값 min도 같은 방식으로 얻을 수 있다.

▼ 예제 5-7/ArrayEx7.java

```
class ArrayEx7 {
    public static void main(String[] args) {
        int[] numArr = new int[10];

        for (int i=0; i < numArr.length; i++ ) {
            numArr[i] = i; // 배열을 0~9의 숫자로 초기화
            System.out.print(numArr[i]);
        }
        System.out.println();

        for (int i=0; i < 100; i++ ) {
            int n = (int)(Math.random() * 10); // 0~9중의 한 값을 임의로 얻는다.
            int tmp = numArr[0];
            numArr[0] = numArr[n];
            numArr[n] = tmp;
        }

        for (int i=0; i < numArr.length; i++ )
            System.out.print(numArr[i]);
    } // main의 끝
}
```

numArr[0]과 numArr[n]의 값을
서로 바꾼다.

▼ 실행결과

0123456789
5827164930

| 참고 | Math.random() 때문에 실행 할 때마다 결과가 다를 수 있다.

[ebook – 샘플. 무료 공유] 자바의 정석 4판 Java 21 올컬러 2025. 7. 7 출시 seong.namkung@gmail.com

길이가 10인 배열 numArr을 생성하고 0~9를 순서대로 저장하여 출력한다. 그 다음 random()을 이용해서 배열의 임의의 위치에 있는 값과 배열의 첫 번째 요소인 'numArr[0]'의 값을 교환하는 일을 100번 반복한다.

만일 random()을 통해 얻은 값 n이 3이라면, 왼쪽의 코드는 오른쪽처럼 될 것이다.

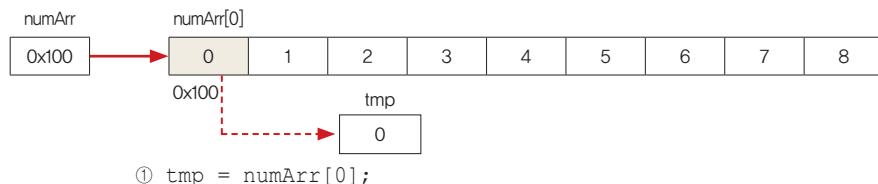
```
int tmp = numArr[0];
numArr[0] = numArr[n];
numArr[n] = tmp;
```



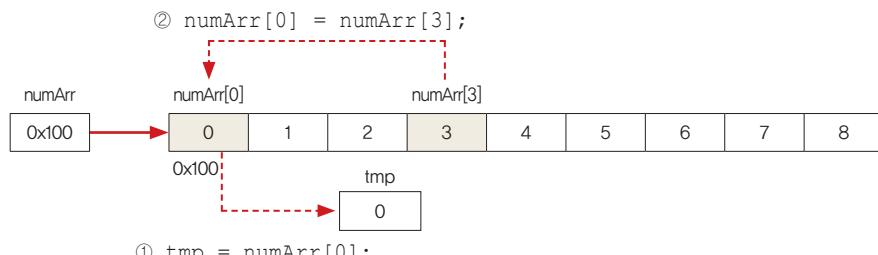
```
int tmp = numArr[0];
numArr[0] = numArr[3];
numArr[3] = tmp;
```

오른쪽의 코드는 numArr[0]과 numArr[3]에 저장된 값을 서로 바꾸는 일을 한다. 두 캡에 담긴 내용물을 서로 바꾸려면, 하나의 빈 캡이 더 필요한 것처럼. 두 변수에 저장된 값을 서로 바꾸려면, 별도의 저장공간이 하나 더 필요하다. 여기서는 변수 tmp가 빈 캡의 역할을 한다. 위의 코드를 단계별로 그림을 그려보면 다음과 같다.

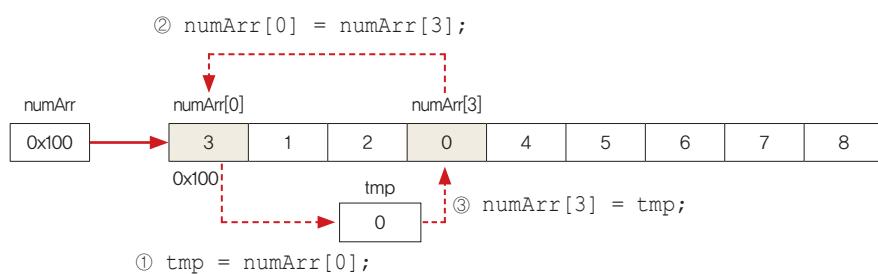
1. `tmp = numArr[0]; // numArr[0]의 값을 변수 tmp에 저장한다.`



2. `numArr[0] = numArr[3]; // numArr[3]의 값을 numArr[0]에 저장한다.`



3. `numArr[3] = tmp; // tmp의 값을 numArr[3]에 저장한다.`



이 작업을 반복적으로 수행하면, 배열 numArr의 값들이 섞이게 된다. 이 예제를 응용하면 카드게임에서 카드 한 벌을 생성하여 초기화한 다음, 카드를 섞는 것과 같은 일을 할 수 있다.

[ebook – 샘플. 무료 공유] 자바의 정석 4판 Java 21 올컬러 2025. 7. 7 출시 seong.namkung@gmail.com

가끔 배열 numArr에 중복된 값이 생길 수도 있지 않느냐는 질문을 받는데, 애초에 중복된 값이 없는 배열에서 값들의 위치만 서로 바꾸는 것이므로 중복된 값이 나올 수가 없다. 종이에 숫자를 써서 직접 해보면 확실히 이해가 될 것이다.

▼ 예제 5-8/ArrayEx8.java

```
class ArrayEx8 {
    public static void main(String[] args) {
        int[] ball = new int[45]; // 45개의 정수를 저장하기 위한 배열 생성.

        // 배열의 각 요소에 1~45의 값을 저장한다.
        for(int i=0; i < ball.length; i++)
            ball[i] = i+1; // ball[0]에 1이 저장된다.

        int temp = 0; // 두 값을 바꾸는데 사용할 임시변수
        int j = 0; // 임의의 값을 얻어서 저장할 변수

        // 배열의 i번째 요소와 임의의 요소에 저장된 값을 서로 바꿔서 배열을 섞는다.
        // 0번째부터 5번째 요소까지 모두 6개만 바꾼다.
        for(int i=0; i < 6; i++) {
            j = (int)(Math.random() * 45); // 0~44범위의 임의의 값을 얻는다.
            temp = ball[i];
            ball[i] = ball[j];
            ball[j] = temp;
        }

        // 배열 ball의 앞에서부터 6개의 요소를 출력한다.
        for(int i=0; i < 6; i++)
            System.out.printf("ball[%d]=%d\n", i, ball[i]);
    }
}
```

▼ 실행결과

```
ball[0]=40
ball[1]=12
ball[2]=19
ball[3]=39
ball[4]=29
ball[5]=3
```

로또번호를 생성하는 예제이다. 길이가 45인 배열에 1부터 45까지의 값을 담은 다음 반복문을 이용해서 ball[i]와 임의의 위치에 있는 값과 자리를 바꾸는 것을 6번 반복한다. 이것은 마치 1부터 45까지의 번호가 쓰인 카드를 잘 섞은 다음 맨 위의 6장을 꺼내는 것과 같다고 할 수 있다.

45개의 요소 중에서 앞에 6개의 요소만 임의의 위치에 있는 요소와 자리를 바꾸면 된다.

```
// 배열의 인덱스가 i인 요소와 임의의 요소 j에 저장된 값을 서로 바꿔서 배열을 섞는다.
// 0 번째부터 5번째 요소까지 모두 6개만 바꾼다.
for(int i=0; i < 6; i++) {
    j = (int)(Math.random() * 45); // 0~44범위의 임의의 값을 얻는다.
    temp = ball[i];
    ball[i] = ball[j];
    ball[j] = temp;
}
```

▼ 실행결과

```
ball[0]=40
ball[1]=12
ball[2]=19
ball[3]=39
ball[4]=29
ball[5]=3
```

사실 예제5-7도 100번씩이나 반복할 필요가 없다. 다음과 같이 변경하면 더 효율적이다.

[ebook – 샘플. 무료 공유] 자바의 정석 4판 Java 21 올컬러 2025. 7. 7 출시 seong.namkung@gmail.com

```

for (int i=0; i < numArr.length; i++) {
    int n = (int)(Math.random() * 10); // 0~9중의 한 값을 임의로 얻는다.
    int tmp = numArr[i];
    numArr[i] = numArr[n];
    numArr[n] = tmp;
}

```

numArr[i]와 numArr[n]의
값을 서로 바꾼다.

임의의 값으로 배열 채우기

배열을 연속적인 범위의 임의의 값으로 채우는 것은 다음과 같이 random()으로 쉽게 가능하다.

```

for(i=0;i<arr.length;i++) {
    arr[i] = (int)(Math.random()*5); // 0~4범위의 임의의 값을 저장
}

```

그러면, 불연속적인 범위의 값들로 배열을 채우는 것은 어떻게 해야 할까? 배열을 하나 더 사용하면 된다. 먼저 배열 code에 불연속적인 값을 담고, 임의로 선택된 index에 저장된 값으로 배열 arr의 요소들을 하나씩 채우면 되는 것이다. 저장된 값에 상관없이 배열의 index는 항상 연속적이기 때문이다. 다음의 예제를 보자.

▼ 예제 5-9/ArrayEx9.java

```

import java.util.*; // Arrays.toString()을 사용하기 위해 추가

class ArrayEx9 {
    public static void main(String[] args) {
        int[] code = { -4, -1, 3, 6, 11 }; // 불연속적인 값으로 구성된 배열
        int[] arr = new int[10];

        for (int i=0; i < arr.length; i++) {
            int tmp = (int)(Math.random() * code.length);
            arr[i] = code[tmp];
        }

        System.out.println(Arrays.toString(arr));
    } // main의 끝
}

```

▼ 실행결과

[-4, -4, -1, -1, 3, 6, 3, 3, 11, 3] ← 실행할 때마다 달라진다.

배열 code의 길이가 5이므로 code.length의 값은 5가 된다. 따라서 변수 tmp에는 0~4범위에 속한 임의의 정수가 저장되는데, 이 범위는 배열 code의 index의 범위와 일치한다.

```

int tmp = (int)(Math.random() * code.length);
→ int tmp = (int)(Math.random() * 5); // tmp에 0, 1, 2, 3, 4중의 하나가 저장된다.

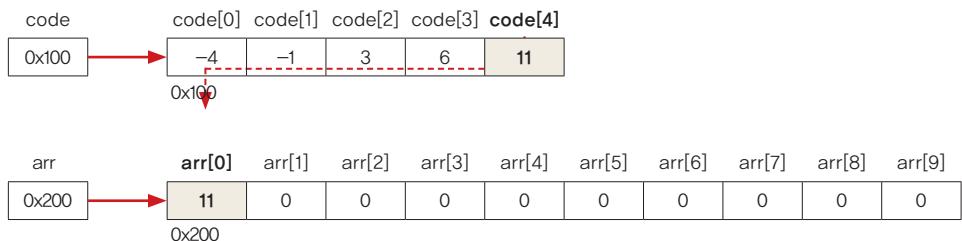
```

만일 i의 값이 0이고, tmp의 값이 4라면 다음과 같이 계산되어 arr[0]에는 code[4]의 값

인 11이 저장된다.

```
arr[i] = code[tmp];
→ arr[0] = code[4];      // code[4]는 배열 code의 5번째 요소이므로 11이다.
→ arr[0] = 11;           // arr[0]에 11이 저장된다.
```

위의 상황을 그림으로 그려보면 다음과 같다.



이와 같은 과정이 반복

▼ 예제 5-10/ArrayEx10.java

```
class ArrayEx10 {
    public static void main(String[] args) {
        int[] numArr = new int[10];

        for (int i=0; i < numArr.length; i++ ) {
            System.out.print(numArr[i] = (int)(Math.random() * 10));
        }
        System.out.println();

        for (int i=0; i < numArr.length-1 ; i++ ) {
            boolean changed = false; // 자리바꿈이 발생했는지를 체크

            for (int j=0; j < numArr.length-1-i ;j++) {
                if(numArr[j] > numArr[j+1]) { // 옆의 값이 작으면 서로 바꾼다.
                    int temp = numArr[j];
                    numArr[j] = numArr[j+1];
                    numArr[j+1] = temp;
                    changed = true; // 자리바꿈이 발생했으니 changed를 true로.
                }
            } // end for j

            if (!changed) break; // 자리바꿈이 없으면 반복문을 벗어난다.

            for(int k=0; k<numArr.length;k++)
                System.out.print(numArr[k]); // 정렬된 결과를 출력
            System.out.println();
        } // end for i
    } // main의 끝
}
```

▼ 실행결과

```
1344213843
1342134438
1321344348
1213343448
1123334448
```

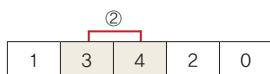
길이가 10인 배열에 0과 9사이의 임의의 값으로 채운 다음, 버블정렬 알고리즘을 통해서 크기순으로 정렬하는 예제이다. 이 알고리즘의 정렬방법은 아주 간단하다. 배열의 길이가 n 일 때, 배열의 첫 번째부터 $n-1$ 까지의 요소에 대해, 바로 옆의 요소의 값과 크기를 비교하여 자리바꿈을 반복하는 것이다.

```
for (int j = 0; j < numArr.length-1-i; j++) {
    // numArr[j]와 바로 옆의 요소 numArr[j+1]을 비교
    if(numArr[j] > numArr[j+1]) {    // 왼쪽의 값이 크면 서로 바꾼다.
        int tmp      = numArr[j];
        numArr[j]    = numArr[j+1];
        numArr[j+1] = tmp;
    }
}
```

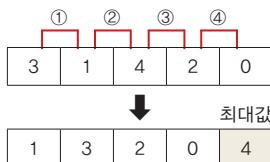
예를 들어 다음과 같이 길이가 5인 int배열이 있을 때, 첫 번째와 두 번째 요소의 값을 비교해서 왼쪽 요소의 값이 크면 두 값의 위치를 바꾸고, 그렇지 않으면 바꾸지 않는다.



위의 그림에서 왼쪽의 값이 크므로 두 값의 자리를 바꾼다.

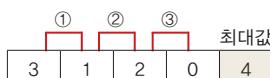


두 번째 비교에서는 왼쪽의 값이 작으므로 두 값의 자리를 바꾸지 않는다. 이러한 작업을 배열의 끝에 도달할 때 까지 반복하면 배열에서 제일 큰 값이 배열의 마지막 값이 된다.



비교횟수는 모두 4번이며, 이 값은 배열의 길이보다 1이 작은 값($\text{numArr.length}-1$)이다. 즉, 배열의 길이가 5라면, 4번만 비교하면 된다는 뜻이다. 나머지 값들이 아직 정렬되지 않았으므로 비교작업을 배열의 첫 번째 요소부터 다시 해야 한다.

그러나 처음과 달리 이번엔 세 번만 비교하면 된다. 배열의 마지막 요소는 최대값이므로 비교할 필요가 없기 때문이다.



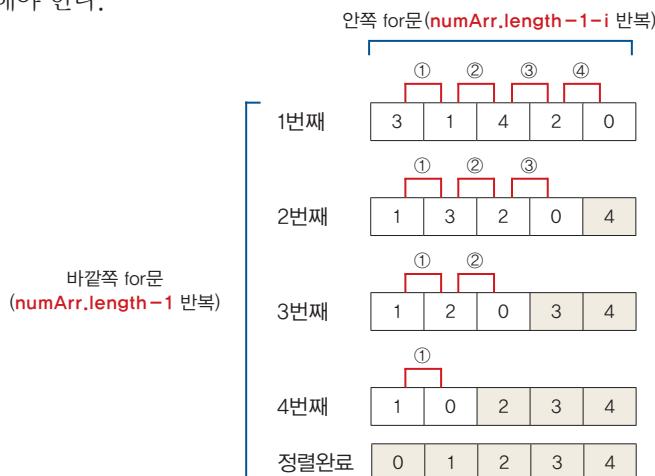
이처럼 비교작업(아래의 for문)을 반복할수록 비교할 범위는 하나씩 줄어든다. 그래서 원래는 배열의 길이에서 1이 작은 ‘ $\text{numArr.length}-1$ ’번을 비교해야 하는데, 매 반복마다 비교횟수가 1씩 줄어들기 때문에 바깥쪽 for문의 제어변수 i 를 빼주는 것이다.

```

for (int j = 0; j < numArr.length-1-i; j++) {
    // numArr[j]와 바로 옆의 요소 numArr[j+1]을 비교한다.
    if (numArr[j] > numArr[j+1]) { // 왼쪽의 값이 크면 서로 바꾼다.
        int tmp      = numArr[j];
        numArr[j]    = numArr[j+1];
        numArr[j+1] = tmp;
    }
}

```

위의 작업이 한번 수행되는 것만으로는 정렬이 되지 않기 때문에 아래의 그림에서 알 수 있는 것처럼, 비교작업(위의 for문)을 모두 4번, 즉, ‘배열의 길이-1’번 만큼 반복해서 비교해야 한다.



그래서 바깥쪽 for문의 조건식이 ‘`numArr.length-1`’이어야 하는 것이다.

```

for (int i = 0; i < numArr.length-1; i++) {
    changed = false; // 매 반복마다 changed를 false로 초기화

    for (int j = 0; j < numArr.length-1-i; j++) {
        if (numArr[j] > numArr[j+1]) { // 옆의 값이 작으면 서로 바꾼다.
            int tmp      = numArr[j];
            numArr[j]    = numArr[j+1];
            numArr[j+1] = tmp;

            changed = true; // 자리바꿈이 발생했으니 changed를 true로 바꾼다.
        }
    } // end for j

    if (!changed) break; // 자리바꿈이 없으면 반복문을 벗어난다.

    for (int k = 0; k < numArr.length; k++)
        System.out.print(numArr[k]); // 정렬된 결과를 출력
    System.out.println();
} // end of for i

```

보다 효율적인 작업을 위해 changed라는 boolean형 변수를 두어서 자리바꿈이 없으면 break문을 수행하여 정렬을 마치도록 했다. 자리바꿈이 없다는 것은 정렬이 완료되었음을 뜻하기 때문이다.

이 정렬 방법을 ‘버블 정렬(bubble sort)’라고 하는데, 비효율적이지만 가장 간단하다.

```
System.out.print(numArr[i] = (int)(Math.random() * 10));
```

그리고 위의 문장은 아래의 두 문장을 하나로 합친 것이다.

```
numArr[i] = (int)(Math.random() * 10);
System.out.print(numArr[i]);
```

▼ 예제 5-11/ArrayEx11.java

```
class ArrayEx11 {
    public static void main(String[] args) {
        int[] numArr = new int[10];
        int[] counter = new int[10];

        for (int i=0; i < numArr.length; i++ ) {
            numArr[i] = (int)(Math.random() * 10); // 0~9의 임의의 수를 배열에 저장
            System.out.print(numArr[i]);
        }
        System.out.println();

        for (int i=0; i < numArr.length; i++ ) {
            counter[numArr[i]]++;
        }

        for (int i=0; i < numArr.length; i++ ) {
            System.out.println( i +"의 개수 :" + counter[i]);
        }
    } // main의 끝
}
```

▼ 실행결과	
4446579753	
0의 개수 :0	
1의 개수 :0	
2의 개수 :0	
3의 개수 :1	
4의 개수 :3	
5의 개수 :2	
6의 개수 :1	
7의 개수 :2	
8의 개수 :0	
9의 개수 :1	

길이가 10인 배열을 만들고 0~9의 임의의 값으로 초기화 한다. 그리고 이 배열에 저장된 각 숫자가 몇 번 반복해서 나타나는지를 세어서 배열 counter에 담고 화면에 출력한다.

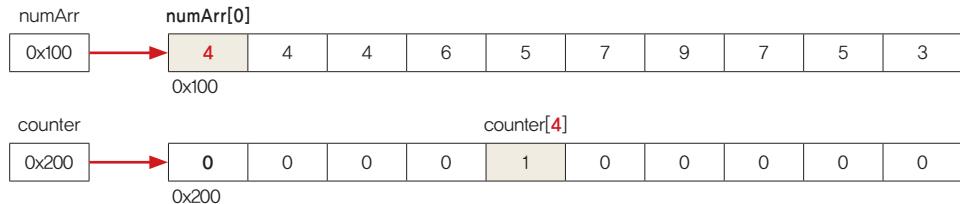
간단한 예제라서 아래의 코드만 이해하면 나머지는 별 어려움이 없을 것이다.

```
for (int i=0; i < numArr.length ; i++ ) {
    counter[numArr[i]]++;
}
```

random()을 사용했기 때문에 실행할 때마다 결과가 달라지겠지만, 실행결과를 토대로 계산과정을 단계별로 살펴보면 다음과 같다.

[ebook – 샘플. 무료 공유] 자바의 정석 4판 Java 21 올컬러 2025. 7. 7 출시 seong.namkung@gmail.com

```
counter[numArr[i]]++; // i의 값이 0인 경우를 가정하면,
→ counter[numArr[0]]++; // numArr[0]의 값은 4이다.
→ counter[4]++; // counter[4]의 값을 1증가시킨다.
```



배열 counter에서, 배열 numArr에 저장된 값과 일치하는 인덱스의 요소에 저장된 값을 1증가시킨다. 위의 그림에서는 numArr[0]에 4가 저장되어있으므로 배열 counter의 인덱스가 4인 요소에 저장된 값이 0에서 1로 증가되었다. 이 과정이 반복되고 나면, 배열 counter의 각 요소에는 해당 인덱스의 값이 몇 번 나타났는지 알 수 있는 값이 저장된다.

배열을 이용한 카운팅은 반장 선거처럼 값의 범위가 제한적일 때만 사용할 수 있다는 단점이 있지만, 정렬과 중복제거를 아주 빠르게 처리할 수 있다. 어떻게하면 카운팅 결과가 저장된 배열로 부터 정렬과 중복제거를 할 수 있을지 생각해보자.

| 플래시동영상 | 예제5-11에 대한 설명은 압축된 소스파일의 '/flash/Array.exe'에서 자세히 볼 수 있다.

2. String배열

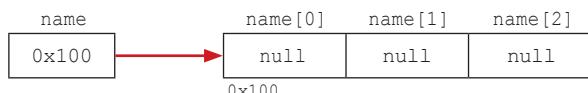
2.1 String배열의 선언과 생성

배열의 타입이 String인 경우에도 int배열의 선언과 생성방법은 다르지 않다. 예를 들어 3개의 문자열(String)을 담을 수 있는 배열을 생성하는 문장은 다음과 같다.

```
String[] name = new String[3]; // 3개의 문자열을 담을 수 있는 배열을 생성
```

위의 문장을 수행한 결과를 그림으로 표현하면 다음과 같다. 3개의 String타입의 참조변수를 저장하기 위한 공간이 마련되고 참조형 변수의 기본값은 null이므로 각 요소의 값은 null로 초기화 된다.

| 참고 | null은 참조 변수가 어떠한 객체도 가리키고 있지 않다는 뜻이다.



참고로 변수의 타입에 따른 기본값은 다음과 같다.

타입	기본값
boolean	false
char	'\u0000'
byte, short, int	0
long	0L
float	0.0f
double	0.0d 또는 0.0
참조형	null

▲ 표 5–2 타입에 따른 변수의 기본값(default value)

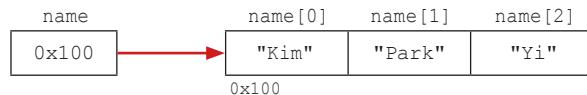
2.2 String배열의 초기화

초기화 역시 int배열과 동일한 방법으로 한다. 아래와 같이 배열의 각 요소에 문자열을 지정하면 된다.

```
String[] name = new String[3]; // 길이가 3인 String배열을 생성
name[0] = "Kim";
name[1] = "Park";
name[2] = "Yi";
```

또는 팔호 {}를 사용해서 다음과 같이 간단히 초기화 할 수도 있다.

```
String[] name = new String[]{"Kim", "Park", "Yi"};
String[] name = { "Kim", "Park", "Yi"}; // new String[]를 생략가능
```

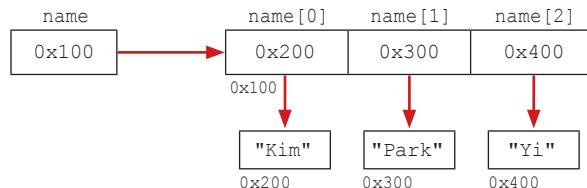


▲ 그림5-4 초기화된 String배열

특별히 String클래스만 “Kim”과 같이 큰따옴표만으로 간략히 표현하는 것이 허용되지만, 원래 String은 클래스이므로 아래의 왼쪽처럼 new연산자를 통해 객체를 생성해야한다.

<pre>String[] name = new String[3]; name[0] = new String("Kim"); name[1] = new String("Park"); name[2] = new String("Yi");</pre>		<pre>String[] name = new String[3]; name[0] = "Kim"; name[1] = "Park"; name[2] = "Yi";</pre>
--	--	--

그림5-4도 편의상 간략히 그린 것이며, 원래는 아래와 같이 그려야 더 정확한 그림이다.



배열에 실제 객체가 아닌 객체의 주소가 저장되어 있는 것을 볼 수 있다. 이처럼, 기본형 배열이 아닌 경우, 즉, 참조형 배열의 경우 배열에 저장되는 것은 객체의 주소이다. 참조형 배열을 객체 배열이라고도 하는데, 다음 장인 ‘6장 객체지향개념 1’에서 배울 것이다.

| 참고 | 모든 참조 변수에는 객체가 메모리에 저장된 주소 또는 null이 저장된다.

▼ 예제 5-12/ArrayEx12.java

```
class ArrayEx12 {
    public static void main(String[] args) {
        String[] names = {"Kim", "Park", "Yi"};

        for(int i=0; i < names.length;i++)
            System.out.println("names["+i+"]: "+names[i]);

        String tmp = names[2]; // 배열 names의 세 번째요소를 tmp에 저장
        System.out.println("tmp:"+tmp);
        names[0] = "Yu"; // 배열 names의 첫 번째 요소를 "Yu"로 변경

        for(String str : names) // 향상된 for문
            System.out.println(str);
    } // main
}
```

▼ 실행결과
names[0]:Kim names[1]:Park names[2]:Yi tmp:Yi Yu Park Yi

▼ 예제 5-13/ArrayEx13.java

```

class ArrayEx13 {
    public static void main(String[] args) {
        char[] hex = { 'C', 'A', 'F', 'E'};

        String[] binary = { "0000", "0001", "0010", "0011"
                           , "0100", "0101", "0110", "0111"
                           , "1000", "1001", "1010", "1011"
                           , "1100", "1101", "1110", "1111" };

        String result="";

        for (int i=0; i < hex.length ; i++ ) {
            if(hex[i] >='0' && hex[i] <='9') {
                result += binary[hex[i]-'0']; // '8'-'0'의 결과는 8
            } else { // A~F이면
                result += binary[hex[i]-'A'+10]; // 'C'-'A'의 결과는 2
            }
        }
        // String(char[] value)
        System.out.println("hex:"+ new String(hex));
        System.out.println("binary:"+result);
    }
}

```

▼ 실행결과

```
hex:CAFE
binary:1100101011111110
```

16진수를 2진수로 변환하는 예제이다. 먼저 변환하고자 하는 16진수를 배열 hex에 나열 한다. 16진수에는 A~F까지 6개의 문자가 포함되므로 char배열로 처리하였다. 그리고 문자열 배열 binary에는 이진수 '0000'부터 '1111'(16진수로 0~F)까지 모두 16개의 값을 문자열로 저장하였다.

for문을 이용해서 배열 hex에 저장된 문자를 하나씩 읽어서 그에 해당하는 이진수 표현을 배열 binary에서 얻어 result에 덧붙이고 그 결과를 화면에 출력한다.

```
result += binary[hex[i]-'A'+10];
```

i의 값이 0일 때, hex[0]의 값은 'C'이므로, 위의 문장은 다음과 같은 과정으로 계산된다.

```

→ result += binary[hex[0]-'A'+10]; // hex[0]은 'C'
→ result += binary['C'-'A'+10]; // 'C'-'A' → 67-65 → 2
→ result += binary[2+10];
→ result += binary[12];
→ result += "1100";

```

2.3 char배열과 String클래스

지금까지 여러 문자, 즉 문자열을 저장할 때 String타입의 변수를 사용했다. 사실 문자열은 ‘문자를 연이어 늘어놓은 것’이라는 뜻으로 char배열과 같은 뜻이다. 그런데 자바에서 char배열이 아닌 String클래스로 문자열을 처리하는 이유는 String클래스가 char배열에 여러 가지 기능을 추가하여 확장한 것이기 때문이다.

그래서 char배열보다 String클래스를 사용하는 것이 문자열을 다루기 더 편리하다.

String클래스는 char배열에 기능(메서드)을 추가한 것이다.

객체지향개념이 나오기 이전의 언어들은 데이터와 기능을 따로 다루었지만, 객체지향언어에서는 데이터와 그에 관련된 기능을 하나의 클래스에 묶어서 다룰 수 있게 한다. 즉, 서로 관련된 것들끼리 데이터와 기능을 구분하지 않고 함께 묶는 것이다.

| 참고 | 여기서 ‘기능’은 함수를 의미하며, 메서드는 객체지향 언어에서 ‘함수’ 대신 사용하는 용어일 뿐 함수와 같은 뜻이다.

char배열과 String클래스의 한 가지 중요한 차이가 있는데, String객체(문자열)는 읽을 수만 있을 뿐 내용을 변경할 수 없다는 것이다.

```
String str = "Java";
str = str + "21";           // "Java21"이라는 새로운 문자열이 str에 저장된다.
System.out.println(str);   // "Java21"
```

위의 문장에서 문자열 str의 내용이 변경되는 것 같지만, 문자열은 변경할 수 없으므로 새로운 내용의 문자열이 생성된다.

| 참고 | 변경 가능한 문자열을 다룰 때는 StringBuffer클래스(p.508)를 사용하면 된다.

String클래스의 주요 메서드

String클래스는 상당히 많은 문자열 관련 메서드를 제공하지만, 가장 기본적인 몇 가지만 살펴보고 나머지는 9장에서 자세히 설명할 것이다. 지금은 원하는 결과를 얻으려면 어떻게 코드를 작성하는지만 이해하자.

메서드	설명
char charAt(int index)	문자열에서 해당 위치(index)에 있는 문자를 반환한다.
int length()	문자열의 길이를 반환한다.
String substring(int from, int to)	문자열에서 해당 범위(from~to)에 있는 문자열을 반환한다. (to는 범위에 포함되지 않음)
boolean equals(Object obj)	문자열의 내용이 같은지 확인한다. 같으면 true, 다르면 false
char[] toCharArray()	문자열을 문자 배열(char[])로 변환해서 반환한다.

▲ 표5-3 String클래스의 주요 메서드

charAt()은 문자열에서 지정된 index에 있는 문자 하나를 가져온다. 배열에서 ‘배열이름 [index]’로 index에 위치한 값을 가져오는 것과 같다고 생각하면 된다. 배열과 마찬가지로 charAt()의 index는 0부터 시작한다.

```
String str = "ABCDE";
char ch = str.charAt(3); // 문자열 str의 4번째 문자 'D' 를 ch에 저장.
```

index	0	1	2	3	4
문자	A	B	C	D	E

substring()은 문자열의 일부를 뽑아낼 수 있다. 주의할 것은 범위의 끝은 포함되지 않는다는 것이다. 예를 들어, index의 범위가 1~4라면 4는 범위에 포함되지 않는다.

```
String str = "012345";
String tmp = str.substring(1,4); // str에서 index범위 1~4의 문자들을 반환
System.out.println(tmp); // "123"이 출력된다.
```

equals()는 이미 앞에서 간단히 배웠는데, 문자열의 내용이 같은지 다른지 확인할 때 사용한다. 기본형 변수의 값을 비교하는 경우 ‘==’연산자를 사용하지만, 문자열의 내용을 비교할 때는 equals()를 사용해야 한다. 그리고 이 메서드는 대소문자를 구분한다는 점에 주의하자. 대소문자를 구분하지 않고 비교하려면 equals()대신 equalsIgnoreCase()를 사용해야 한다.

```
String str = "abc";
if(str.equals("abc")) { // str과 "abc"의 내용이 같으면 true
    ...
}
```

char배열과 String의 변환

가끔 char배열을 String으로 변환하거나, 또는 그 반대로 변환해야하는 경우가 있다. 그럴 때 다음의 코드를 사용하자.

```
char[] chArr = { 'A', 'B', 'C' };
String str = new String(chArr); // char배열 → String
char[] tmp = str.toCharArray(); // String → char배열
```

▼ 예제 5-14/ArrayEx14.java

```
class ArrayEx14 {
    public static void main(String[] args) {
        String src = "ABCDE";
        for(int i=0; i < src.length(); i++) {
            char ch = src.charAt(i); // src의 i번째 문자를 ch에 저장
            System.out.println("src.charAt("+i+"):"+ ch);
        }
    }
}
```

```
// String을 char[]로 변환
char[] chArr = src.toCharArray();

// char배열(char[])을 출력
System.out.println(chArr);
}
```

▼ 실행결과

```
src.charAt(0):A
src.charAt(1):B
src.charAt(2):C
src.charAt(3):D
src.charAt(4):E
ABCDE
```

String의 charAt()을 사용하는 방법을 보여주는 예제이다. charAt(int index)은 문자열의 index번째 위치에 있는 문자를 반환한다. idx의 값은 배열처럼 0부터 시작한다는 것에 주의하자.

그리고 println()로 문자 배열을 출력하면 문자 배열의 모든 문자를 순서대로 붙여서 출력한다.

▼ 예제 5-15/ArrayEx15.java

```
class ArrayEx15 {
    public static void main(String[] args) {
        String source = "SOSHELP";
        String[] Morse = {".-.", "-...", "-.-.", "-..", ".",
            "...-", "--.", "...-", "...-", "-.-", ".--", ".-.", "...-", "-.-",
            "-.-", "-.-.", "-.-", "-.-", "-.-", "-.-", "-.-", "-.-", "-.-"};
        String result = "";

        for (int i=0; i < source.length(); i++) {
            result += Morse[source.charAt(i) - 'A'];
        }
        System.out.println("source:" + source);
        System.out.println("morse:" + result);
    }
}
```

▼ 실행결과

```
source:SOSHELP
morse:...---.....-....-
```

문자열을 모尔斯(morse)부호로 변환하는 예제이다. 이전의 16진수를 2진수로 변환하는 예제와 같지만, 이번엔 char배열 대신 String을 사용했다.

String의 문자 개수는 length()로 얻을 수 있고, charAt(int index)는 String의 index 번째 문자를 반환한다. 그래서 for문의 조건식에 length()를 사용하고 charAt(int index)를 통해서 source에서 문자를 하나씩 차례대로 읽어 올 수 있다.

```
result += Morse[source.charAt(i) - 'A']; // i가 0일 때
→ result += Morse[source.charAt(0) - 'A']; // source.charAt(0)는 첫 번째 문자
→ result += Morse['S' - 'A']; // 'S' - 'A' → 83 - 65 → 18
→ result += Morse[18];
→ result += "..."; // result = result + "...";
```

2.4 커맨드 라인을 통해 입력받기

Scanner클래스의 nextLine()외에도 화면을 통해 사용자로부터 값을 입력받을 수 있는 간단한 방법이 있다. 바로 커맨드라인을 이용한 방법인데, 프로그램을 실행할 때 클래스 이름 뒤에 공백문자로 구분하여 여러 개의 문자열을 프로그램에 전달 할 수 있다.

만일 실행할 프로그램의 main메서드가 담긴 클래스의 이름이 MainTest라고 가정하면 다음과 같이 실행할 수 있을 것이다.

```
c:\...\ch05> java MainTest abc 123
```

커맨드라인을 통해 입력된 두 문자열 “abc”와 “123”은 String배열에 담겨서 MainTest 클래스의 main메서드의 매개변수(args)에 전달된다. 그리고는 main메서드 내에서 args[0], args[1]과 같은 방식으로 커맨드라인으로부터 전달받은 문자열에 접근할 수 있다. 여기서 args[0]은 “abc”이고 args[1]은 “123”이 된다.

▼ 예제 5-16/ArrayEx16.java

```
class ArrayEx16 {
    public static void main(String[] args) {
        System.out.println("매개변수의 개수:"+args.length);
        for(int i=0;i< args.length;i++) {
            System.out.println("args["+ i + "] = \""+ args[i] + "\"");
        }
    }
}
```

▼ 실행결과

```
C:\...\cd \Users\userid\jdk21\ch05\out\production\ch05
C:\...\ch05>java ArrayEx16 abc 123 "Hello world"
매개변수의 개수:3
args[0] = "abc"
args[1] = "123"
args[2] = "Hello world"
C:\...\ch05>java ArrayEx16 ← 매개변수를 입력하지 않았다.
매개변수의 개수:0
```

| 참고 | 실행결과에서 '...'은 경로를 짧게 생략해서 표현한 것이며, 'userid'는 사용자에 따라 다를 수 있다.

커맨드라인에 입력된 매개변수는 공백문자로 구분하기 때문에 입력값에 공백이 있는 경우 큰따옴표("")로 감싸주어야 한다. 그리고 커맨드라인에서 숫자를 입력해도 숫자가 아닌 문자열로 처리된다는 것에 주의해야한다. 문자열 “123”을 숫자 123으로 바꾸려면 다음과 같이 한다.

```
int num = Integer.parseInt("123"); // 변수 num에 숫자 123이 저장된다.
```

그리고 커맨드라인에 매개변수를 입력하지 않으면 크기가 0인 배열이 생성되어 args.length의 값은 0이 된다. 앞서 배운 것처럼 이렇게 크기가 0인 배열을 생성하는 것도 가능하다. 만일 입력된 매개변수가 없다고 해서 배열을 생성하지 않으면 참조변수 args의 값은 null이 될 것이고, 배열 args를 사용하는 모든 코드에서 에러가 발생할 것이다. 이러한 에러를 피하려면, 다음과 같이 main메서드에 if문을 추가해야 한다.

[ebook – 샘플. 무료 공유] 자바의 정석 4판 Java 21 올컬러 2025. 7. 7 출시 seong.namkung@gmail.com

```
public static void main(String[] args) {
    if(args != null) { // args가 null이 아닐 때만 괄호{}의 문장들을 수행
        System.out.println("매개변수의 개수:"+args.length);
        for(int i = 0;i< args.length;i++) {
            System.out.println("args["+ i + "] = "+ args[i]);
        }
    }
}
```

그러나 입력된 매개변수가 없을 때, JVM이 null 대신 크기가 0인 배열을 생성해서 args에 전달하도록 구현되어 있어서 우리는 이러한 수고를 덜게 되었다.

▼ 예제 5-17/ArrayEx17.java

```
class ArrayEx17 {
    public static void main(String[] args) {
        if (args.length != 3) { // 입력된 값의 개수가 3개가 아니면,
            System.out.println("usage: java ArrayEx17 NUM1 OP NUM2");
            System.exit(0); // 프로그램을 종료한다.
        }

        int num1 = Integer.parseInt(args[0]); // 문자열을 숫자로 변환한다.
        char op = args[1].charAt(0); // 문자열을 문자(char)로 변환한다.
        int num2 = Integer.parseInt(args[2]);
        int result = 0;

        switch(op) { // switch문의 수식으로 char타입의 변수도 가능하다.
            case '+':
                result = num1 + num2;
                break;
            case '-':
                result = num1 - num2;
                break;
            case 'x':
                result = num1 * num2;
                break;
            case '/':
                result = num1 / num2;
                break;
            default :
                System.out.println("지원되지 않는 연산입니다.");
        }
        System.out.println("결과:"+result);
    }
}
```

▼ 실행결과

```
C:\...\ch05>java ArrayEx17
usage: java ArrayEx17 NUM1 OP NUM2
C:\...\ch05>java ArrayEx17 10 + 3
결과:13
C:\...\ch05>java ArrayEx17 10 x 3
결과:30
```

| 참고 | 실행결과에서 '...'은 경로(\Users\userid\jdk21\ch05\out\production)를 짧게 생략해서 표현한 것이며, 'userid'는 사용자에 따라 다를 수 있다. 'cd'명령을 이용해서 해당 경로로 이동한 다음에 실행해야 한다.

화면으로부터 사칙연산을 수행하는 수식을 입력받아서 계산하여 그 결과를 보여주는 예제이다. 커맨드라인으로부터 입력받은 데이터는 모두 문자열이므로 숫자와 문자로 변환해야 필요하며, Integer.parseInt()를 사용했다.

[ebook – 샘플. 무료 공유] 자바의 정석 4판 Java 21 올컬러 2025. 7. 7 출시 seong.namkung@gmail.com

3. 다차원 배열

지금까지 우리가 배운 배열은 1차원 배열인데, 2차원 이상의 배열, 즉 다차원(多次元, multi-dimensional) 배열도 가능하다. 메모리의 용량이 허용하는 한, 차원의 제한은 없지만, 주로 1, 2차원 배열이 사용되므로 2차원 배열만 잘 이해하면 3차원 이상의 배열도 어렵지 않게 다룰 수 있으므로 2차원 배열에 대해서 중점적으로 배울 것이다.

3.1 2차원 배열의 선언과 인덱스

2차원 배열을 선언하는 방법은 1차원 배열과 같다. 다만 괄호[]가 하나 더 들어갈 뿐이다.

선언 방법	선언 예
타입[][] 변수이름;	int[][] score;
타입 변수이름[][];	int score[][];
타입[] 변수이름[];	int[] score[];

▲ 표 5-4 2차원 배열의 선언

| 참고 | 3차원이상의 고차원 배열의 선언은 대괄호[]의 개수를 차원의 수만큼 추가해 주기만 하면 된다.

2차원 배열은 주로 테이블 형태의 데이터를 담는데 사용되며, 만일 4행 3열의 데이터를 담기 위한 배열을 생성하려면 다음과 같이 한다.

```
int[][] score = new int[4][3]; // 4행 3열의 2차원 배열을 생성한다.
```

위 문장이 수행되면 아래의 그림처럼 4행 3열의 데이터, 모두 12개의 int값을 저장할 수 있는 공간이 마련된다.



위의 그림에서는 각 요소, 즉 저장 공간의 타입을 적어놓은 것이고, 실제로는 배열 요소의 타입인 int의 기본값인 0이 저장된다. 배열을 생성하면, 배열의 각 요소에는 배열 요소 타입의 기본값이 자동적으로 저장된다.

2차원 배열의 index

2차원 배열은 행(row)과 열(column)로 구성되어 있기 때문에 index도 행과 열에 각각 하나씩 존재한다. ‘행index’의 범위는 ‘0~행의 길이-1’이고 ‘열index’의 범위는 ‘0~열의 길이-1’이다. 그리고 2차원 배열의 각 요소에 접근하는 방법은 ‘배열이름[행index][열index]’이다.

[ebook – 샘플. 무료 공유] 자바의 정석 4판 Java 21 올컬러 2025. 7. 7 출시 seong.namkung@gmail.com

만일 다음과 같이 배열 score를 생성하면, score[0][0]부터 score[3][2]까지 모두 12개($4 \times 3 = 12$)의 int값을 저장할 수 있는 공간이 마련되고, 각 배열 요소에 접근할 수 있는 방법은 아래의 그림과 같다.

```
int[][] score = new int[4][3]; // 4행 3열의 2차원 배열 score를 생성
```

열 index(0 ~ 열의 길이-1)			
0	1	2	
행 index (0 ~ 행의 길이-1)	score[0][0]	score[0][1]	score[0][2]
	score[1][0]	score[1][1]	score[1][2]
	score[2][0]	score[2][1]	score[2][2]
	score[3][0]	score[3][1]	score[3][2]

배열 score의 1행 1열에 100을 저장하고, 이 값을 출력하려면 다음과 같이 하면 된다.

```
score[0][0] = 100; // 배열 score의 1행 1열에 100을 저장
System.out.println(score[0][0]); // 배열 score의 1행 1열의 값을 출력
```

3.2 2차원 배열의 초기화

2차원 배열도 팔호{}를 사용해서 생성과 초기화를 동시에 할 수 있다. 다만, 1차원 배열보다 팔호{}를 한번 더 써서 행별로 구분해 준다.

```
int[][] arr = new int[][]{ {1, 2, 3}, {4, 5, 6} };
int[][] arr = { {1, 2, 3}, {4, 5, 6} }; // new int[][]가 생략됨
```

크기가 작은 배열은 위와 같이 간단히 한 줄로 써도 되지만, 가능하면 다음과 같이 행별로 줄 바꿈을 해주는 것이 보기도 좋고 이해하기 쉽다.

```
int[][] arr = {
    {1, 2, 3},
    {4, 5, 6}
};
```

만일 아래와 같은 테이블 형태의 데이터를 배열에 저장하려면,

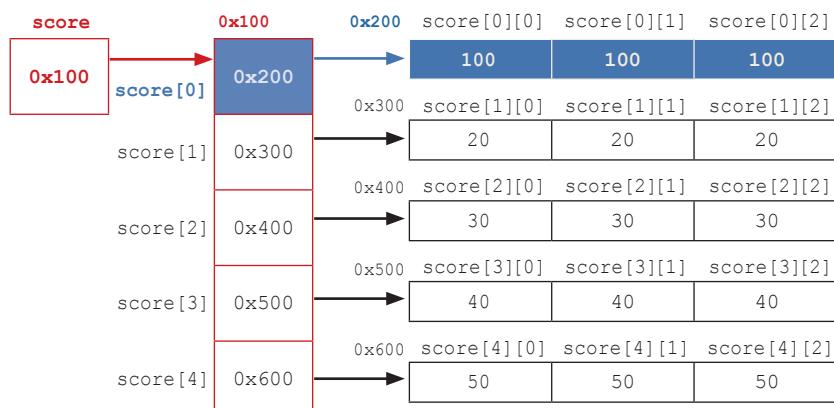
	국어	영어	수학
1	100	100	100
2	20	20	20
3	30	30	30
4	40	40	40
5	50	50	50

[ebook – 샘플. 무료 공유] 자바의 정석 4판 Java 21 올컬러 2025. 7. 7 출시 seong.namkung@gmail.com

다음과 같이 하면 된다.

```
int[][] score = {
    {100, 100, 100}
, {20, 20, 20}
, {30, 30, 30}
, {40, 40, 40}
, {50, 50, 50}
};
```

위 문장이 수행된 후에, 2차원 배열 score가 메모리에 어떤 형태로 만들어지는지 그려보면 다음과 같다.v



▲ 그림5-5 2차원 배열의 구조

그림5-5에서 알 수 있듯이 2차원 배열은 ‘배열의 배열’로 구성되어 있다. 여러 개의 1차원 배열을 뭍어서 또 하나의 배열로 만든 것이다. 그러면, 여기서 score.length의 값은 얼마일까?

배열 참조변수 score가 참조하고 있는 배열의 길이가 얼마인가를 세어보면 될 것이다. 정답은 5이다. 그리고 score[0].length은 배열 참조변수 score[0]이 참조하고 있는 배열의 길이이므로 3이다.

같은 이유로 score[1].length, score[2].length, score[3].length, score[4].length의 값 역시 모두 3이다.

만일 for문을 이용해서 2차원 배열을 초기화한다면 다음과 같을 것이다.

```
// 2차원 배열 score의 모든 요소를 10으로 초기화한다.
for (int i = 0; i < score.length; i++) {
    for (int j = 0; j < score[i].length; j++) {
        score[i][j] = 10;
    }
}
```

[ebook – 샘플. 무료 공유] 자바의 정석 4판 Java 21 올컬러 2025. 7. 7 출시 seong.namkung@gmail.com

▼ 예제 5-18/ArrayEx18.java

```
class ArrayEx18 {
    public static void main(String[] args) {
        int[][] score = {
            { 100, 100, 100 },
            { 20, 20, 20 },
            { 30, 30, 30 },
            { 40, 40, 40 }
        };
        int sum = 0;

        for(int i=0;i < score.length;i++) {
            for(int j=0;j < score[i].length;j++) {
                System.out.printf("score[%d] [%d]=%d%n", i, j, score[i][j]);
            }
        }

        for (int[] tmp : score) {
            for (int i : tmp) {
                sum += i;
            }
        }

        System.out.println("sum="+sum);
    }
}
```

▼ 실행결과

```
score[0][0]=100
score[0][1]=100
score[0][2]=100
score[1][0]=20
score[1][1]=20
score[1][2]=20
score[2][0]=30
score[2][1]=30
score[2][2]=30
score[3][0]=40
score[3][1]=40
score[3][2]=40
sum=570
```

2차원 배열 score의 모든 요소의 합을 구하고, 출력하는 예제이다. 하나의 이중 for문으로 처리가 가능한 작업이지만, 향상된 for문으로 2차원 배열의 모든 요소를 읽어오는 방법을 보여주기 위해 출력과 합계를 따로 처리하였다.

```
for (int i : score) { // 예러. 2차원 배열 score의 각 요소는 1차원 배열
    sum += i;
}
```

이렇게 간단히 되면 좋겠지만, 2차원 배열 score의 각 요소는 1차원 배열이므로 아래와 같이 for문을 하나 더 추가해야 한다.

```
for (int[] tmp : score) { // score의 각 요소(1차원 배열 주소)를 tmp에 저장
    for (int i : tmp) { // tmp는 1차원 배열을 가리키는 참조변수
        sum += i;
    }
}
```

| 참고 | 향상된 for문으로 배열의 각 요소의 값을 읽을 수 있지만, 배열에 저장된 값은 변경할 수 없다.

[ebook – 샘플. 무료 공유] 자바의 정석 4판 Java 21 올컬러 2025. 7. 7 출시 seong.namkung@gmail.com

▼ 예제 5-19/ArrayEx19.java

```

class ArrayEx19 {
    public static void main(String[] args) {
        int[][] score = {
            { 100, 100, 100}
            , { 20, 20, 20}
            , { 30, 30, 30}
            , { 40, 40, 40}
            , { 50, 50, 50}
        };
        // 과목별 총점
        int korTotal = 0, engTotal = 0, mathTotal = 0;

        System.out.println("번호 국어 영어 수학 총점 평균 ");
        System.out.println("===== ");

        for(int i=0;i < score.length;i++) {
            int sum = 0;          // 개인별 총점
            float avg = 0.0f;     // 개인별 평균

            korTotal += score[i][0];
            engTotal += score[i][1];
            mathTotal += score[i][2];
            System.out.printf("%3d", i+1);

            for(int j=0;j < score[i].length;j++) {
                sum += score[i][j];
                System.out.printf("%5d", score[i][j]);
            }

            avg = sum/(float)score[i].length; // 평균계산
            System.out.printf("%5d %.1f\n", sum, avg);
        }

        System.out.println("===== ");
        System.out.printf("총점:%3d %4d %4d\n",korTotal,engTotal,mathTotal);
    }
}

```

▼ 실행결과

번호	국어	영어	수학	총점	평균
1	100	100	100	300	100.0
2	20	20	20	60	20.0
3	30	30	30	90	30.0
4	40	40	40	120	40.0
5	50	50	50	150	50.0
<hr/>					
총점: 240 240 240					

5명의 학생의 세 과목 점수를 더해서 각 학생의 총점과 평균을 계산하고, 과목별 총점을 계산하는 예제이다. 간단한 예제이므로 자세한 설명은 생략한다.

3.3 가변 배열

자바는 2차원 이상의 배열을 ‘배열의 배열’의 형태로 처리한다는 사실을 이용하면 보다 자유로운 형태의 배열을 구성할 수 있다.

2차원 이상의 다차원 배열을 생성할 때 전체 배열 차수 중 마지막 차수의 길이를 지정하지 않고, 추후에 각기 다른 길이의 배열을 생성함으로써 고정된 형태가 아닌 보다 유동적인 가변 배열을 구성할 수 있다.

만일 다음과 같이 ‘ 5×3 ’길이의 2차원 배열 score를 생성하는 코드가 있을 때,

```
int[][] score = new int[5][3]; // 5행 3열의 2차원 배열 생성
```

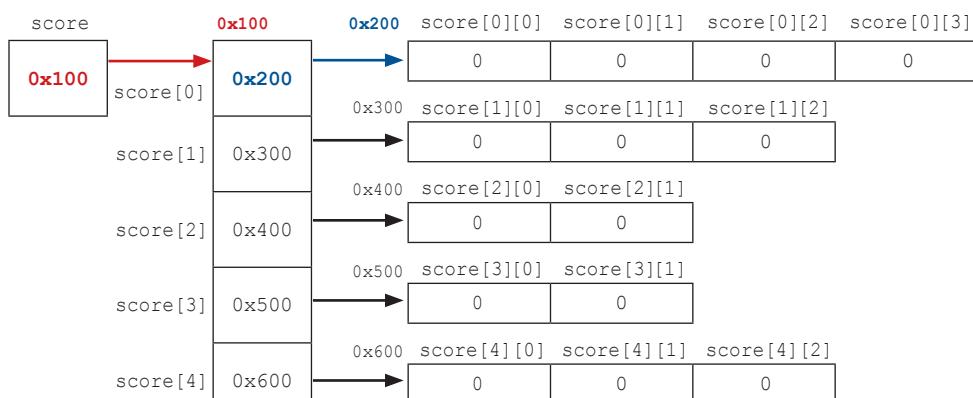
위 코드를 다음과 같이 표현 할 수 있다.

```
int[][] score = new int[5][]; // 두 번째 차원의 길이는 지정하지 않는다.
score[0] = new int[3];
score[1] = new int[3];
score[2] = new int[3];
score[3] = new int[3];
score[4] = new int[3];
```

첫 번째 코드와 같이 2차원 배열을 생성하면 직사각형 테이블 형태의 고정적인 배열만 생성할 수 있지만, 두 번째 코드와 같이 2차원 배열을 생성하면 다음과 같이 각 행마다 다른 길이의 배열을 생성하는 것이 가능하다.

```
int[][] score = new int[5][];
score[0] = new int[4];
score[1] = new int[3];
score[2] = new int[2];
score[3] = new int[2];
score[4] = new int[3];
```

위의 코드에 의해서 생성된 2차원 배열을 그림으로 표현하면 다음과 같다.



▲ 그림5-6 가변 배열의 구조

score.length의 값은 여전히 5지만, 일반적인 2차원 배열과 달리 score[0].length의 값은 4이고 score[1].length의 값은 3으로 서로 다르다.

가변배열 역시 중괄호{}를 이용해서 다음과 같이 생성과 초기화를 동시에 하는 것이 가능하다.

```
int[][] score = {
    {100, 100, 100, 100}
    , {20, 20, 20}
    , {30, 30}
    , {40, 40}
    , {50, 50, 50}
};
```

| 플래시동영상 | /flash/MultiDim.exe을 보면 가변 배열의 생성과정을 자세히 볼 수 있다.

3.4 다차원 배열의 활용

다차원 배열의 대표적인 예제들을 몇 가지 골라보았다. 이 예제들만 잘 이해해도 다차원 배열을 활용하는데 별 어려움이 없을 것이다.

[예제5–20] 좌표에 X표하기 입력한 2차원 좌표의 위치에 X를 표시

[예제5–21] 빙고 빙고판을 만들고 입력받은 숫자를 빙고판에서 지운다.

[예제5–22] 행렬의 곱셈 두 행렬(matrix)을 곱한 결과를 출력

[예제5–23] 단어 맞추기 영어 단어를 보여주고, 뜻을 맞추는 게임

이 예제에 추가하면 좋을만한 기능은 없는지 고민해보고, 조금씩 단계별로 발전시켜보면 좋은 공부가 될 것이다.

▼ 예제 5–20/MultiArrEx.java

```
import java.util.*;

class MultiArrEx {
    public static void main(String[] args) {
        final int SIZE = 10;
        int x = 0, y = 0;

        char[][] board = new char[SIZE][SIZE];
        byte[][] shipBoard = {
            // 1 2 3 4 5 6 7 8 9
            { 0, 0, 0, 0, 0, 0, 1, 0, 0 }, // 1
            { 1, 1, 1, 1, 0, 0, 1, 0, 0 }, // 2
            { 0, 0, 0, 0, 0, 0, 1, 0, 0 }, // 3
            { 0, 0, 0, 0, 0, 0, 1, 0, 0 }, // 4
            { 0, 0, 0, 0, 0, 0, 0, 0, 0 }, // 5
            { 1, 1, 0, 1, 0, 0, 0, 0, 0 }, // 6
            { 0, 0, 0, 1, 0, 0, 0, 0, 0 }, // 7
            { 0, 0, 0, 1, 0, 0, 0, 0, 0 }, // 8
            { 0, 0, 0, 0, 0, 1, 1, 1, 0 }, // 9
        };
    }
}
```

```

// 1행에 행번호를, 1열에 열번호를 저장한다.
for(int i=1;i<SIZE;i++)
    board[0][i] = board[i][0] = (char)(i+'0');

Scanner scanner = new Scanner(System.in);

while(true) {
    System.out.print("좌표를 입력하세요. (종료는 00)>");
    String input = scanner.nextLine(); // 화면입력받은 내용을 input에 저장

    if(input.length()==2) { // 두 글자를 입력한 경우
        x = input.charAt(0) - '0'; // 문자를 숫자로 변환
        y = input.charAt(1) - '0';

        if(x==0 && y==0) // x와 y가 모두 0인 경우 종료
            break;
    }

    if(input.length()!=2 || x<0 || x>=SIZE || y<0 || y>=SIZE) {
        System.out.println("잘못된 입력입니다. 다시 입력해주세요.");
        continue;
    }

    // shipBoard[x-1][y-1]의 값이 1이면, 'O'를 board[x][y]에 저장한다.
    board[x][y] = shipBoard[x-1][y-1]==1 ? 'O' : 'X';

    // 배열 board의 내용을 화면에 출력한다.
    for(int i=0;i<SIZE;i++)
        System.out.println(board[i]); // board[i]는 1차원 배열
    System.out.println();
}
} // main의 끝
}

```

▼ 실행결과

좌표를 입력하세요. (종료는 00)>**1010**

잘못된 입력입니다. 다시 입력해주세요.

좌표를 입력하세요. (종료는 00)>**33**

123456789

1

2

3 X

4

5

6

7

8

9

좌표를 입력하세요. (종료는 00)>**00**

돌이 마주 앉아 다양한 크기의 배를 상대방이 알지 못하게 배치한 다음, 번갈아가며 좌표를 불러서 상대방의 배의 위치를 알아내는 게임을 간단히 하여 예제로 만들어 보았다.

2차원 char배열 board는 입력한 좌표를 표시하기 위한 것이고, 2차원 byte배열 shipBoard에는 상대방의 배의 위치를 저장한다. 0은 바다이고, 1은 배가 있는 것이다.

```

char[][] board = new char[SIZE][SIZE];
byte[][] shipBoard = {
    // 1 2 3 4 5 6 7 8 9
    { 0, 0, 0, 0, 0, 0, 1, 0, 0 }, // 1
    { 1, 1, 1, 1, 0, 0, 1, 0, 0 }, // 2
    { 0, 0, 0, 0, 0, 0, 1, 0, 0 }, // 3
    { 0, 0, 0, 0, 0, 0, 1, 0, 0 }, // 4
    { 0, 0, 0, 0, 0, 0, 0, 0, 0 }, // 5
    { 1, 1, 0, 1, 0, 0, 0, 0, 0 }, // 6
    { 0, 0, 0, 1, 0, 0, 0, 0, 0 }, // 7
    { 0, 0, 0, 1, 0, 0, 0, 0, 0 }, // 8
    { 0, 0, 0, 0, 0, 1, 1, 1, 0 } // 9
};

```

배열 board는 좌표를 쉽게 입력하기 위한 행번호와 열번호가 필요하다. 그래서 배열 board가 배열 shipBoard보다 행과 열의 길이가 1씩 큰 것이다.

```

// 1행에 행번호를, 1열에 열번호를 저장한다.
for(int i=1;i<SIZE;i++) {
    board[0][i] = board[i][0] = (char)(i+'0'); // (char)(1+'0') → '1'
                                                                // (char)(2+'0') → '2'
}

```

board는 char배열이므로, 숫자를 문자로 변환하여 저장해야한다. 그래서 변수 i에 문자 '0'을 더한다. 숫자 1에 문자 '0'을 더하면 문자 '1'이 된다. 그 다음엔 무한 반복문으로 좌표를 반복해서 입력받는다. 입력받은 좌표 x, y에 저장된 값이 1이면, board[x][y]에 'O'를 저장하고, 1이 아니면 'X'를 저장한다. 배열 board와 shipBoard간에는 좌표가 가로, 세로로 각각 1씩 차이가 난다는 점을 잊지 말자.

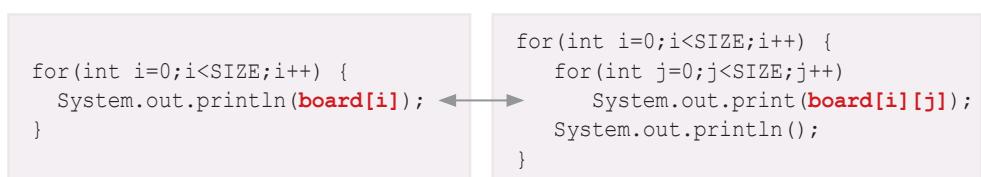
```

while(true) {
    ...
    board[x][y] = shipBoard[x-1][y-1] == 1 ? 'O':'X';
}

```

그리고 2차원 char배열의 각 요소는 1차원 배열이므로 아래의 원쪽과 같이 간단히 출력할 수 있다. 원래는 오른쪽과 같이 배열의 각 요소를 하나씩 출력해야하는데, println메서드로 1차원 char배열의 참조변수를 출력하면, 배열의 모든 요소를 한 줄로 출력한다.

| 참고 | 그림5–5에서 알 수 있듯이 2차원 배열의 각 요소는 1차원 배열의 참조변수 역할을 한다.



println메서드가 모든 1차원 배열을 이렇게 출력할 수 있는 것은 아니고, char배열인 경우만 가능하다.

▼ 예제 5-21/MultiArrEx2.java

```

import java.util.*;

class MultiArrEx2 {
    public static void main(String[] args) {
        final int SIZE = 5;
        int x = 0, y = 0, num = 0;

        int[][] bingo = new int[SIZE][SIZE];
        Scanner scanner = new Scanner(System.in);

        // 배열의 모든 요소를 1부터 SIZE*SIZE까지의 숫자로 초기화
        for(int i=0;i<SIZE;i++) {
            for(int j=0;j<SIZE;j++)
                bingo[i][j] = i*SIZE + j + 1;

        // 배열에 저장된 값을 뒤섞는다. (shuffle)
        for(int i=0;i<SIZE;i++) {
            for(int j=0;j<SIZE;j++) {
                x = (int)(Math.random() * SIZE);
                y = (int)(Math.random() * SIZE);

                // bingo[i][j]와 임의로 선택된 값(bingo[x][y])을 바꾼다.
                int tmp = bingo[i][j];
                bingo[i][j] = bingo[x][y];
                bingo[x][y] = tmp;
            }
        }

        do {
            for(int i=0;i<SIZE;i++) {
                for(int j=0;j<SIZE;j++)
                    System.out.printf("%2d ", bingo[i][j]);
                System.out.println();
            }
            System.out.println();

            System.out.printf("1~%d의 숫자를 입력하세요. (종료:0) >", SIZE*SIZE);
            String tmp = scanner.nextLine(); // 화면에서 입력받은 내용을 tmp에 저장
            num = Integer.parseInt(tmp); // 입력받은 문자열(tmp)을 숫자로 변환

            // 입력받은 숫자와 같은 숫자가 저장된 요소를 찾아서 0을 저장
            outer:
            for(int i=0;i<SIZE;i++) {
                for(int j=0;j<SIZE;j++) {
                    if(bingo[i][j]==num) {
                        bingo[i][j] = 0;
                        break outer; // 2중 반복문을 벗어난다.
                    }
                }
            }

            } while(num!=0);
        } // main의 끝
    }
}

```

▼ 실행결과

```
9 22  2 12 17
6   1 25  3  5
16  7 11 19 23
14 10 21 13  8
 4 15 20 24 18
```

1~25의 숫자를 입력하세요. (종료:0) >**1**

```
9 22  2 12 17
6   0 25  3  5
16  7 11 19 23
14 10 21 13  8
 4 15 20 24 18
```

1~25의 숫자를 입력하세요. (종료:0) >**9**

```
0 22  2 12 17
6   0 25  3  5
16  7 11 19 23
14 10 21 13  8
 4 15 20 24 18
```

1~25의 숫자를 입력하세요. (종료:0) >**0**

5×5크기의 빙고판에 1~25의 숫자를 차례로 저장한 다음에, Math.random()을 이용해서 저장된 값의 위치를 섞는다. 여기까지의 과정은 지금까지 여러 번 반복된 것이므로 자세한 설명은 생략한다.

그 다음에 사용자로부터 숫자를 입력받아서 일치하는 숫자가 빙고판에 있으면 해당 숫자를 0으로 바꾼다.

```
// 입력받은 숫자와 같은 숫자가 저장된 요소를 찾아서 0을 저장
outer:
for(int i = 0;i<SIZE;i++) {
    for(int j = 0;j < SIZE;j++) {
        if(bingo[i][j] == num) {
            bingo[i][j] = 0;           // 일치하는 숫자를 찾으면 0으로 변경
            break outer;           // 2중 반복문을 벗어난다.
        }
    }
}
```

입력받은 숫자와 일치하는 숫자를 빙고판에서 찾는 방법은 간단하다. 배열의 첫 번째 요소부터 순서대로 하나씩 비교하다 일치하는 숫자를 찾으면, 값을 0으로 바꾸고 break문으로 반복문을 빠져나오면 된다. 이중 반복문이므로, 이름 붙은 break문을 사용해야 한다.

▼ 예제 5-22/MultiArrEx3.java

```

class MultiArrEx3 {
    public static void main(String[] args) {
        int[][] m1 = {
            {1, 2, 3},
            {4, 5, 6}
        };

        int[][] m2 = {
            {1, 2},
            {3, 4},
            {5, 6}
        };

        final int ROW      = m1.length;      // m1의 행 길이
        final int COL      = m2[0].length;    // m2의 열 길이
        final int M2_ROW   = m2.length;      // m2의 행 길이

        int[][] m3 = new int[ROW][COL];

        // 행렬곱 m1 x m2의 결과를 m3에 저장
        for(int i=0;i<ROW;i++)
            for(int j=0;j<COL;j++)
                for(int k=0;k<M2_ROW;k++)
                    m3[i][j] += m1[i][k] * m2[k][j];

        // 행렬 m3를 출력
        for(int i=0;i<ROW;i++) {
            for(int j=0;j<COL;j++) {
                System.out.printf("%3d ", m3[i][j]);
            }
            System.out.println();
        }
    } // main의 끝
}

```

▼ 실행결과
22 28
49 64

수학에서 두 개의 행렬(matrix) m1과 m2가 있을 때, 이 두 행렬을 곱한 결과인 행렬 m3는 아래와 같이 정의된다.

| 참고 | 학교 과제로 자주 출제되기 때문에 넓은 예제다. 어렵게 느껴진다면 넘어가도 좋다.

$$\begin{array}{ccc}
 m1 & m2 & m3 \\
 \left(\begin{array}{ccc} A0 & A1 & A2 \\ B0 & B1 & B2 \end{array} \right) \times \left(\begin{array}{cc} a0 & a1 \\ b0 & b1 \\ c0 & c1 \end{array} \right) = \left(\begin{array}{cc} A0 \times a0 + A1 \times b0 + A2 \times c0 & A0 \times a1 + A1 \times b1 + A2 \times c1 \\ B0 \times a0 + B1 \times b0 + B2 \times c0 & B0 \times a1 + B1 \times b1 + B2 \times c1 \end{array} \right)
 \end{array}$$

두 행렬의 곱셈이 가능하려면, m1의 열의 길이와 m2의 행의 길이가 같아야 한다는 조건이 있다. 위의 경우에는 m1이 2행 3열이고, m2가 3행 2열이므로 곱셈이 가능하다.

그리고 곱셈연산의 결과인 행렬 m3의 행의 길이는 m1의 행의 길이와 같고, 열의 길이는 m2의 열의 길이와 같다. 2행 3열인 행렬과 3행 2열인 행렬을 곱하면 결과는 2행 2열의 행렬이 되는 것이다.

지금까지의 내용은 수학에서 정의된 행렬의 곱셈에 대한 정의와 규칙일 뿐, 왜 그렇게 되는지 따지지 말자. 이에 맞게 코드를 작성하는 것만 생각하자. 위의 행렬의 곱셈 공식을 2 차원 배열로 표현하면 아래와 같다.

$$\begin{array}{c}
 \text{m1} \\
 \begin{array}{|c|c|c|} \hline
 m1[0][0] & m1[0][1] & m1[0][2] \\ \hline
 m1[1][0] & m1[1][1] & m1[1][2] \\ \hline
 \end{array}
 \end{array}
 \times
 \begin{array}{c}
 \text{m2} \\
 \begin{array}{|c|c|} \hline
 m2[0][0] & m2[0][1] \\ \hline
 m2[1][0] & m2[1][1] \\ \hline
 m2[2][0] & m2[2][1] \\ \hline
 \end{array}
 \end{array}
 =
 \begin{array}{c}
 \text{m3} \\
 \begin{array}{|c|c|} \hline
 m3[0][0] & m3[0][1] \\ \hline
 m3[1][0] & m3[1][1] \\ \hline
 \end{array}
 \end{array}$$

그리고 행렬 곱셈의 결과인 행렬 m3의 각 요소들은 아래와 같이 계산된다.

$$\begin{aligned}
 m3[0][0] &= m1[0][0] * m2[0][0] \\
 &\quad + m1[0][1] * m2[1][0] \\
 &\quad + m1[0][2] * m2[2][0]; \\
 m3[0][1] &= m1[0][0] * m2[0][1] \\
 &\quad + m1[0][1] * m2[1][1] \\
 &\quad + m1[0][2] * m2[2][1]; \\
 m3[1][0] &= m1[1][0] * m2[0][0] \\
 &\quad + m1[1][1] * m2[1][0] \\
 &\quad + m1[1][2] * m2[2][0]; \\
 m3[1][1] &= m1[1][0] * m2[0][1] \\
 &\quad + m1[1][1] * m2[1][1] \\
 &\quad + m1[1][2] * m2[2][1];
 \end{aligned}$$

위의 문장들을 자세히 들여다보면, 행렬 m3의 행index가 행렬 m1의 행index와 일치하고, m3의 열index가 m2의 열index와 일치한다는 것을 알 수 있다. 그래서 위의 문장들은 다음과 같이 2중 for문으로 대체할 수 있다.

```

for(int i=0;i<2;i++) { // i = 0, 1
    for(int j=0;j<2;j++) { // j = 0, 1
        m3[i][j] = m1[i][0] * m2[0][j]
                    + m1[i][1] * m2[1][j]
                    + m1[i][2] * m2[2][j];
    }
}

```

위의 문장을 잘 보면 여전히 반복되는 부분이 있다. m1의 열index와 m2의 행index가 동일하게 0부터 2까지 1씩 증가한다. 이 부분을 또 하나의 for문으로 바꾸면 다음과 같다.

```

for(int i=0;i<2;i++) {
    for(int j=0;j<2;j++) {
        for(int k=0;k<3;k++) { // k = 0, 1, 2
            m3[i][j] += m1[i][k] * m2[k][j];
        }
    }
}

```

[ebook – 샘플. 무료 공유] 자바의 정석 4판 Java 21 올컬러 2025. 7. 7 출시 seong.namkung@gmail.com

행렬 m1과 m2의 길이가 달라져도 행렬 m3가 계산될 수 있도록, 배열 m3의 크기와 for 문의 조건식이 동적으로 계산되게 하였다.

```
final int ROW      = m1.length;           // m1의 행 길이(m3의 행 길이)
final int COL      = m2[0].length;         // m2의 열 길이(m3의 열 길이)
final int M2_ROW   = m2.length;           // m2의 행 길이

int[][] m3 = new int[ROW][COL];
```

두 행렬의 곱셈이 가능하려면, 배열 m1의 열의 길이와 배열 m2의 행의 길이가 일치해야 한다는 것에 주의하자.

▼ 예제 5-23/MultiArrEx4.java

```
import java.util.*;

class MultiArrEx4{
    public static void main(String[] args) {
        String[][] words = {
            {"chair", "의자"},           // words[0][0], words[0][1]
            {"computer", "컴퓨터"},       // words[1][0], words[1][1]
            {"integer", "정수"}          // words[2][0], words[2][1]
        };

        Scanner scanner = new Scanner(System.in);

        for(int i=0;i<words.length;i++) {
            System.out.printf("Q%d. %s의 뜻은?", i+1, words[i][0]);

            String tmp = scanner.nextLine();

            if(tmp.equals(words[i][1])) {
                System.out.printf("정답입니다.%n%n");
            } else {
                System.out.printf("틀렸습니다. 정답은 %s입니다.%n%n", words[i][1]);
            }
        } // for
    } // main의 끝
}
```

▼ 실행결과

Q1. chair의 뜻은?dm1wk
틀렸습니다. 정답은 의자입니다.

Q2. computer의 뜻은?컴퓨터
정답입니다.

Q3. integer의 뜻은?정수
정답입니다.

[ebook – 샘플. 무료 공유] 자바의 정석 4판 Java 21 올컬러 2025. 7. 7 출시 seong.namkung@gmail.com

영단어를 보여주고 단어의 뜻을 맞추는 예제이다. words[i][0]은 문제이고, words[i][1]은 답이다. words[i][0]을 화면에 보여주고, 입력받은 답은 tmp에 저장한다.

```
System.out.printf("Q%d. %s의 뜻은?", i+1, words[i][0]);  
String tmp = scanner.nextLine();
```

그 다음엔 equals()로 tmp와 words[i][1]을 비교해서 정답인지 확인한다.

```
if(tmp.equals(words[i][1])) {  
    System.out.printf("정답입니다.%n%n");  
} else {  
    System.out.printf("틀렸습니다. 정답은 %s입니다.%n%n", words[i][1]);  
}
```

| 참고 | 연습문제는 깃헙(<https://github.com/castello/javajungsuk4>)에서 PDF파일로 제공

Java

Programming
Language

Chapter 06

객체지향 프로그래밍 I

Object-oriented Programming I

1. 객체지향언어

1.1 객체지향 언어의 역사

요즘은 컴퓨터의 눈부신 발전으로 활용 폭이 넓고 다양해져서 컴퓨터가 사용되지 않는 분야가 없을 정도지만, 초기에는 주로 과학실험이나 미사일 발사실험과 같은 모의실험(simulation)을 목적으로 사용되었다. 이 시절의 과학자들은 모의실험을 위해 실제 세계와 유사한 가상 세계를 컴퓨터 속에 구현하고자 노력하였으며 이러한 노력은 객체지향이론을 탄생시켰다.

객체지향이론의 기본 개념은 '실제 세계는 사물(객체)로 이루어져 있으며, 발생하는 모든 사건들은 사물간의 상호작용이다.'라는 것이다. 실제 사물의 속성과 기능을 분석한 다음, 데이터(변수)와 함수로 정의함으로써 실제 세계를 컴퓨터 속에 옮겨 놓은 것과 같은 가상 세계를 구현하고 이 가상세계에서 모의실험을 함으로써 많은 시간과 비용을 절약할 수 있었다. 객체지향이론은 상속, 캡슐화, 추상화 개념을 중심으로 점차 구체적으로 발전되었으며 1960년대 중반에 객체지향이론을 프로그래밍언어에 적용한 시뮬라(Simula)라는 최초의 객체지향언어가 탄생하였다.

그 당시에는 FORTRAN이나 COBOL과 같은 절차적 언어들이 주류를 이루었으며, 객체지향언어는 널리 사용되지 못하고 있었다. 1980년대 중반에 C++을 비롯하여 여러 객체지향언어가 발표되면서 객체지향언어가 본격적으로 개발자들의 관심을 끌기 시작하였지만 여전히 사용자층이 넓지 못했다.

그러나 프로그램의 규모가 점점 커지고 사용자들의 요구가 빠르게 변화해가는 상황을 절차적 언어로는 극복하기 어렵다는 한계를 느끼고 객체지향언어를 이용한 개발방법론이 대안으로 떠오르게 되면서 조금씩 입지를 넓혀가고 있었다.

자바가 1995년에 발표되고 1990년대 말에 인터넷의 발전과 함께 크게 유행하면서 객체지향언어는 이제 프로그래밍언어의 주류로 자리 잡았다.

1.2 객체지향언어

객체지향언어는 기존의 프로그래밍언어와 다른 전혀 새로운 것이 아니라, 기존의 프로그래밍 언어에 몇 가지 새로운 규칙을 추가한 보다 발전된 형태의 것이다. 이러한 규칙들을 이용해서 코드 간에 서로 관계를 맺어 줌으로써 보다 유기적으로 프로그램을 구성하는 것이 가능해졌다. 기존의 프로그래밍 언어에 익숙한 사람이라면 자바의 객체지향적인 부분만 새로 배우면 된다. 다만 절차적 언어에 익숙한 프로그래밍 습관을 객체지향적으로 바꾸도록 노력해야 할 것이다. 객체지향언어의 주요특징은 다음과 같다.

1. 코드의 재사용성이 높다.

새로운 코드를 작성할 때 기존의 코드를 이용하여 쉽게 작성할 수 있다.

2. 코드의 관리가 용이하다.

코드간의 관계를 이용해서 적은 노력으로 쉽게 코드를 변경할 수 있다.

3. 신뢰성이 높은 프로그래밍을 가능하게 한다.

제어자와 메서드를 이용해서 데이터를 보호하고 올바른 값을 유지하도록 하며, 코드의 중복을 제거하여 코드의 불일치로 인한 오동작을 방지할 수 있다.

객체지향언어의 가장 큰 장점은 '코드의 재사용성이 높고 변경에 유리하다.'는 것이다. 이러한 객체지향언어의 장점은 프로그램의 개발과 유지보수에 드는 시간과 비용을 획기적으로 개선하였다.

앞으로 상속, 다형성과 같은 객체지향개념을 학습할 때 재사용성과 유지보수 그리고 중복된 코드의 제거, 이 세 가지 관점에서 보면 보다 쉽게 이해할 수 있을 것이다.

객체지향 프로그래밍은 프로그래머에게 거시적 관점에서 설계할 수 있는 능력을 요구하기 때문에 객체지향개념을 이해했다 하더라도 자바의 객체지향적 장점을 충분히 활용한 프로그램을 작성하기란 쉽지 않을 것이다.

너무 객체지향개념에 얹매여서 고민하기보다는 일단 프로그램을 기능적으로 완성한 다음 어떻게 하면 보다 객체지향적으로 코드를 개선할 수 있을지를 고민하여 점차 개선해나가는 것이 좋다.

이러한 경험들이 축적되어야 프로그램을 객체지향적으로 설계할 수 있는 능력이 길러지는 것이지 처음부터 이론을 많이 안다고 해서 좋은 설계를 할 수 있는 것은 아니다.

2. 클래스와 객체

2.1 클래스와 객체의 정의와 용도

클래스란 '객체를 정의한 것.' 또는 '객체의 설계도'라고 할 수 있다. 클래스는 객체를 생성하는데 사용되며, 객체는 클래스에 정의된 대로 생성된다.

클래스의 정의 클래스란 객체를 정의해 놓은 것이다.

클래스의 용도 클래스는 객체를 생성하는데 사용된다.

객체의 사전적인 정의는, '실제로 존재하는 것'이다. 우리가 주변에서 볼 수 있는 책상, 의자, 자동차와 같은 사물들이 곧 객체이다. 객체지향이론에서는 사물과 같은 유형적인 것뿐만 아니라, 개념이나 논리와 같은 무형적인 것들도 객체로 간주한다.

프로그래밍관점에서 객체는 클래스에 정의된 대로 메모리에 생성된 것을 뜻한다.

객체의 정의 실제로 존재하는 것. 사물 또는 개념

객체의 용도 객체가 가지고 있는 기능과 속성에 따라 다름

유형의 객체 책상, 의자, 자동차, TV와 같은 사물

무형의 객체 수학공식, 프로그램 에러와 같은 논리나 개념

클래스와 객체의 관계를 우리가 살고 있는 실생활에서 예를 들면, 제품 설계도와 제품과의 관계라고 할 수 있다. 예를 들면, TV설계도(클래스)는 TV라는 제품(객체)을 정의한 것이며, TV(객체)를 만드는데 사용된다.

또한 클래스는 단지 객체를 생성하는데 사용될 뿐, 객체 그 자체는 아니다. 우리가 원하는 기능의 객체를 사용하기 위해서는 먼저 클래스로부터 객체를 생성하는 과정이 선행되어야 한다.

우리가 TV를 보기 위해서는, TV(객체)가 필요한 것이지 TV설계도(클래스)가 필요한 것은 아니며, TV설계도(클래스)는 단지 TV라는 제품(객체)을 만드는데 사용될 뿐이다.

그리고 TV설계도를 통해 TV가 만들어진 후에야 사용할 수 있다. 프로그래밍에서는 먼저 클래스를 작성한 다음, 클래스로부터 객체를 생성하여 사용한다.

| 참고 | 객체를 사용한다는 것은 객체가 가지고 있는 속성과 기능을 사용한다는 뜻이다.

클래스	객체
제품 설계도	제품
TV 설계도	TV
붕어빵 기계	붕어빵

▲ 표6-1 클래스와 객체의 예

클래스를 정의하고 클래스를 통해 객체를 생성하는 이유는 설계도를 통해서 제품을 만드는 이유와 같다. 하나의 설계도만 잘 만들어 놓으면 제품을 만드는 일이 쉬워진다. 제품을 만들 때마다 매번 고민할 필요없이 설계도대로 만들면 되기 때문이다.

설계도 없이 제품을 만든다고 생각해보라. 복잡한 제품일수록 설계도 없이 제품을 만든다는 것은 상상할 수도 없을 것이다.

이와 마찬가지로 클래스를 한번만 잘 만들어 놓기만 하면, 매번 객체를 생성할 때마다 어떻게 객체를 만들어야 할지를 고민하지 않아도 된다. 그냥 클래스로부터 객체를 생성해서 사용하기만 하면 되는 것이다.

JDK(Java Development Kit)에서는 프로그래밍을 위해 많은 수의 유용한 클래스(Java API)를 기본적으로 제공하고 있으며, 우리는 이 클래스들을 이용해서 원하는 기능의 프로그램을 보다 쉽게 작성할 수 있다.

2.2 객체와 인스턴스

클래스로부터 객체를 만드는 과정을 클래스의 인스턴스화(instantiate)라고 하며, 어떤 클래스로부터 만들어진 객체를 그 클래스의 인스턴스(instance)라고 한다.

예를 들면, Tv클래스로부터 만들어진 객체를 Tv클래스의 인스턴스라고 한다. 결국 인스턴스는 객체와 같은 의미이지만, 객체는 모든 인스턴스를 대표하는 포괄적인 의미를 갖고 있으며, 인스턴스는 어떤 클래스로부터 만들어진 것인지를 강조하는 보다 구체적인 의미를 갖고 있다.

[ebook – 샘플. 무료 공유] 자바의 정석 4판 Java 21 올컬러 2025. 7. 7 출시 seong.namkung@gmail.com

예를 들면, ‘책상은 인스턴스다.’라고 하기 보다는 ‘책상은 객체다.’라는 쪽이, ‘책상은 책상 클래스의 객체이다.’라고 하기 보다는 ‘책상은 책상 클래스의 인스턴스다.’라고 하는 것이 더 자연스럽다.

인스턴스와 객체는 같은 의미이므로 두 용어의 사용을 엄격히 구분할 필요는 없지만, 위의 예에서 본 것과 같이 문맥에 따라 구별하여 사용하는 것이 좋다.



2.3 객체의 구성요소 – 속성과 기능

객체는 속성과 기능, 두 종류의 구성요소로 이루어져 있으며, 일반적으로 객체는 다수의 속성과 다수의 기능을 갖는다. 즉, 객체는 속성과 기능의 집합이라고 할 수 있다. 그리고 객체가 가지고 있는 속성과 기능을 그 객체의 멤버(구성원, member)라 한다.

클래스란 객체를 정의한 것이므로 클래스에는 객체의 모든 속성과 기능이 정의되어 있다. 클래스로부터 객체를 생성하면, 클래스에 정의된 속성과 기능을 가진 객체가 만들어지는 것이다.

속성과 기능은 아래와 같이 같은 뜻의 여러 가지 용어가 있으며, 앞으로 이 중에서도 ‘속성’보다는 ‘멤버 변수’를, ‘기능’보다는 ‘메서드’를 주로 사용할 것이다.

속성(property) 멤버 변수(member variable), 특성(attribute), 필드(field), 상태(state)

기능(function) 메서드(method), 함수(function), 행위(behavior)

보다 쉽게 이해할 수 있도록 TV를 예로 들어보자. TV의 속성으로는 전원상태, 크기, 길이, 높이, 색상, 볼륨, 채널과 같은 것들이 있으며, 기능으로는 켜기, 끄기, 볼륨 높이기, 채널 변경하기 등이 있다.



속성	크기, 길이, 높이, 색상, 볼륨, 채널 등
기능	켜기, 끄기, 볼륨 높이기, 볼륨 낮추기, 채널 변경하기 등

▲ 그림 6-1 TV의 속성과 기능

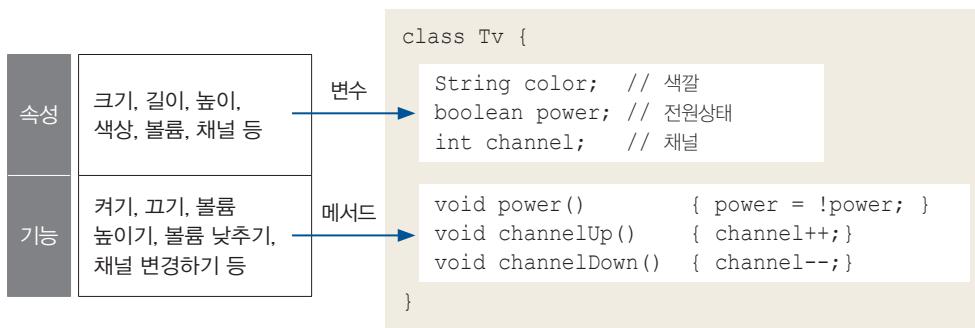
객체지향 프로그래밍에서는 속성과 기능을 각각 변수와 메서드로 표현한다.

속성(property) → 멤버 변수(member variable)
기능(function) → 메서드(method)

채널 → int channel
채널 높이기 → channelUp() { ... }

[ebook – 샘플. 무료 공유] 자바의 정석 4판 Java 21 올컬러 2025. 7. 7 출시 seong.namkung@gmail.com

위에서 분석한 내용을 토대로 Tv클래스를 만들어 보면 다음과 같다.



| 참고 | 멤버 변수와 메서드를 선언할 때 순서는 관계없지만, 일반적으로 메서드보다 멤버 변수를 먼저 선언하고 멤버 변수는 멤버 변수끼리 메서드는 메서드끼리 모아 놓는 것이 일반적이다.

실제 TV가 갖는 기능과 속성은 이 외에도 더 있지만, 프로그래밍에 필요한 속성과 기능만을 선택하여 클래스를 작성하면 된다.

각 변수의 타입은 속성의 값에 알맞은 것을 선택해야 한다. 전원상태(power)의 경우, on과 off 두 가지 값을 가질 수 있으므로 boolean으로 선언했다.

power()의 ‘power = !power;’ 이 문장에서 power의 값이 true면 false로, false면 true로 변경하는 일을 한다. power의 값에 관계없이 항상 반대의 값으로 변경해주면 되므로 굳이 if문을 사용할 필요가 없다. 참고로 if문을 사용하여 코드를 작성하면 다음과 같다.

```
if (power) // if (power == true)
    power = false;
else
    power = true;
```

2.4 인스턴스의 생성과 사용

Tv클래스를 선언한 것은 Tv설계도를 작성한 것에 불과하므로, Tv인스턴스를 생성해야 제품(Tv)을 사용할 수 있다. 클래스로부터 인스턴스를 생성하는 방법은 여러가지가 있지만 일반적으로는 다음과 같이 한다.

```
클래스명 변수명; // 클래스의 객체를 참조하기 위한 참조 변수(리모콘)를 선언
변수명 = new 클래스명(); // 클래스의 객체를 생성 후, 객체의 주소를 참조 변수에 저장(연결)

Tv t; // Tv클래스 타입의 참조 변수 t를 선언
t = new Tv(); // Tv인스턴스를 생성한 후, 생성된 Tv인스턴스의 주소를 t에 저장
```

▼ 예제 6-1/TvEx.java

```

class Tv {
    // Tv의 속성(멤버변수)
    String color;           // 색상
    boolean power;          // 전원상태(on/off)
    int channel;            // 채널

    // Tv의 기능(메서드)
    void power() { power = !power; } // TV를 켜거나 끄는 기능을 하는 메서드
    void channelUp() { ++channel; } // TV의 채널을 높이는 기능을 하는 메서드
    void channelDown() { --channel; } // TV의 채널을 낮추는 기능을 하는 메서드
}

class TvEx {
    public static void main(String args[]) {
        Tv t;                  // Tv인스턴스를 참조하기 위한 변수 t를 선언
        t = new Tv();           // Tv인스턴스를 생성
        t.channel = 7;          // Tv인스턴스의 멤버변수 channel의 값을 7로 한다.
        t.channelDown();        // Tv인스턴스의 메서드 channelDown()을 호출한다.
        System.out.println("현재 채널은 " + t.channel + " 입니다.");
    }
}

```

▼ 실행결과
현재 채널은 6 입니다.

이 예제는 `Tv`클래스로부터 인스턴스를 생성하고 인스턴스의 속성(`channel`)과 메서드(`channelDown()`)를 사용하는 방법을 보여 주는 것이다. 이 예제를 그림과 함께 단계별로 자세히 살펴보도록 하자.

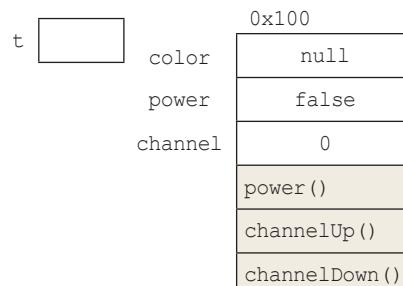
1. `Tv t;`

`Tv`클래스 타입의 참조변수 `t`를 선언한다. 메모리에 참조 변수 `t`를 위한 공간이 마련된다. 아직 인스턴스가 생성되지 않았으므로 참조 변수로 아무것도 할 수 없다.

2. `t = new Tv();`

연산자 `new`에 의해 `Tv`클래스의 인스턴스가 메모리의 빈 공간에 생성된다. 주소가 `0x100`인 곳에 생성되었다고 가정하자. 이 때, 멤버 변수는 각 자료형에 해당하는 기본값으로 초기화 된다.

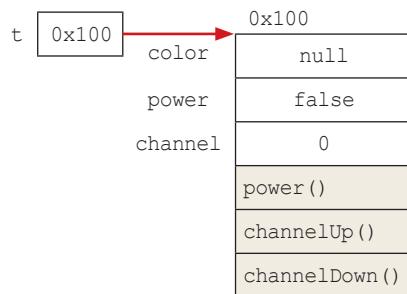
`color`은 참조형이므로 `null`로, `power`는 `boolean`이므로 `false`로, 그리고 `channel`은 `int`이므로 0으로 초기화 된다.



[ebook – 샘플. 무료 공유] 자바의 정석 4판 Java 21 올컬러 2025. 7. 7 출시 seong.namkung@gmail.com

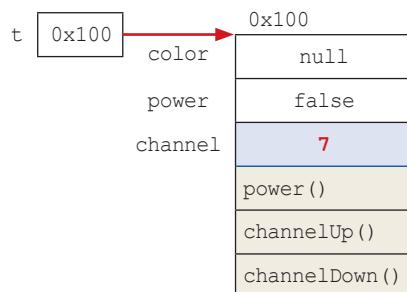
그 다음에는 대입 연산자(=)에 의해서 생성된 객체의 주소가 참조 변수 t에 저장된다. 이제는 참조 변수 t를 통해 Tv인스턴스에 접근할 수 있다. 인스턴스를 다루기 위해서는 참조 변수가 반드시 필요하다.

| 참고 | 아래 그림에서의 화살표는 참조 변수 t가 Tv인스턴스를 참조하고 있다는 것을 알기 쉽게 하기 위해 추가한 상징적인 것이다. 이 때, 참조 변수 t가 Tv인스턴스를 '가리키고 있다' 또는 '참조하고 있다'라고 한다.



3. `t.channel = 7 ;`

참조 변수 t에 저장된 주소에 있는 인스턴스의 멤버 변수 `channel`에 7을 저장한다. 여기서 알 수 있는 것처럼, 인스턴스의 멤버 변수(속성)를 사용하려면 '참조변수.멤버변수'와 같이 하면 된다.

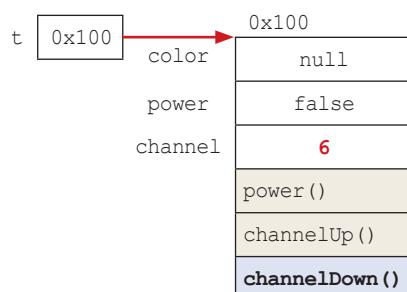


4. `t.channelDown();`

참조 변수 t가 참조하고 있는 `Tv`인스턴스의 `channelDown`메서드를 호출한다. `channel Down`메서드는 멤버 변수 `channel`에 저장되어 있는 값을 1 감소시킨다.

```
void channelDown() { --channel; }
```

`channelDown()`에 의해서 `channel`의 값은 7에서 6이 된다.



[ebook – 샘플. 무료 공유] 자바의 정석 4판 Java 21 올컬러 2025. 7. 7 출시 seong.namkung@gmail.com

[ebook – 샘플. 무료 공유] 자바의 정석 4판 Java 21 올컬러 2025. 7. 7 출시 seong.namkung@gmail.com

5. `System.out.println("현재 채널은 " + t.channel + " 입니다.");`

참조 변수 t가 참조하고 있는 Tv인스턴스의 멤버 변수 channel에 저장되어 있는 값을 출력한다.

현재 channel의 값은 6이므로 '현재 채널은 6 입니다.'가 화면에 출력된다.

인스턴스와 참조 변수의 관계는 마치 우리가 일상생활에서 사용하는 TV와 TV리모콘의 관계와 같다. TV리모콘(참조 변수)을 사용하여 TV(인스턴스)를 다루기 때문이다. 다른 점은 인스턴스는 오직 참조 변수를 통해서만 다룰 수 있다는 것이다.

그리고 TV를 사용하려면 TV 리모콘을 사용해야하고, 에어콘을 사용하려면, 에어콘 리모콘을 사용해야하는 것처럼 Tv인스턴스를 사용하려면, Tv클래스 타입의 참조 변수가 필요하다.

인스턴스는 참조 변수를 통해서만 다룰 수 있으며,

참조 변수의 타입은 인스턴스의 타입과 일치해야 한다.

▼ 예제 6-2/TvEx2.java

```
class Tv2 {
    // Tv2의 속성(멤버 변수)
    String color;           // 색상
    boolean power;          // 전원상태(on/off)
    int channel;            // 채널

    // Tv2의 기능(메서드)
    void power() { power = !power; } // TV를 켜거나 끄는 기능을 하는 메서드
    void channelUp() { ++channel; } // TV의 채널을 높이는 기능을 하는 메서드
    void channelDown() { --channel; } // TV의 채널을 낮추는 기능을 하는 메서드
}

class TvEx2 {
    public static void main(String args[]) {
        Tv2 t1 = new Tv2(); // Tv2 t1; t1 = new Tv2(); 를 한 문장으로 가능
        Tv2 t2 = new Tv2();
        System.out.println("t1의 channel값은 " + t1.channel + "입니다.");
        System.out.println("t2의 channel값은 " + t2.channel + "입니다.");

        t1.channel = 7; // channel 값을 7으로 한다.
        System.out.println("t1의 channel값을 7로 변경하였습니다.");

        System.out.println("t1의 channel값은 " + t1.channel + "입니다.");
        System.out.println("t2의 channel값은 " + t2.channel + "입니다.");
    }
}
```

▼ 실행결과

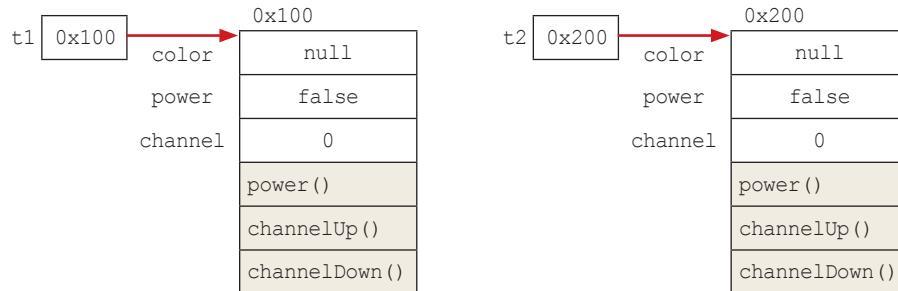
```
t1의 channel값은 0입니다.
t2의 channel값은 0입니다.
t1의 channel값을 7로 변경하였습니다.
t1의 channel값은 7입니다.
t2의 channel값은 0입니다.
```

[ebook – 샘플. 무료 공유] 자바의 정석 4판 Java 21 올컬러 2025. 7. 7 출시 seong.namkung@gmail.com

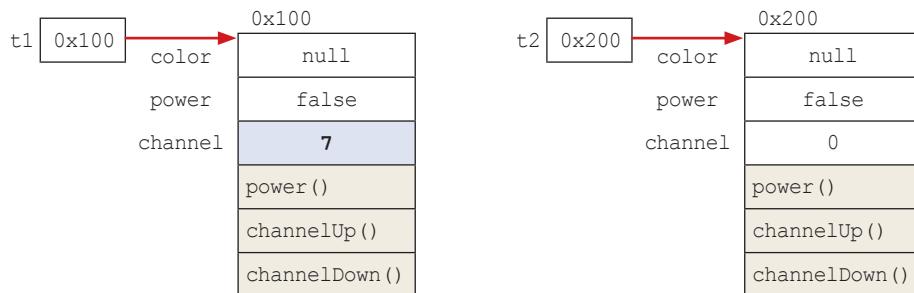
위의 예제는 Tv클래스의 인스턴스 t1과 t2를 생성한 후에, 인스턴스 t1의 멤버 변수인 channel의 값을 변경하였다.

| 참고 | 참조 변수 t1이 가리키고(참조하고) 있는 인스턴스를 간단히 인스턴스 t1이라고 했다.

1. `Tv2 t1 = new Tv2();
Tv2 t2 = new Tv2();`



2. `t1.channel = 7;` // t1이 가리키고 있는 인스턴스의 멤버 변수 channel의 값을 7로 변경



같은 클래스로부터 생성되었어도 각 인스턴스의 속성(멤버 변수)은 서로 다른 값을 유지할 수 있으며, 메서드의 내용은 모든 인스턴스에서 동일하다.

▼ 예제 6-3/TvEx3.java

```
class Tv3 {
    // Tv3의 속성(멤버 변수)
    String color;           // 색상
    boolean power;          // 전원상태(on/off)
    int channel;            // 채널

    // Tv3의 기능(메서드)
    void power() { power = !power; } // TV를 켜거나 끄는 기능을 하는 메서드
    void channelUp() { ++channel; } // TV의 채널을 높이는 기능을 하는 메서드
    void channelDown() { --channel; } // TV의 채널을 낮추는 기능을 하는 메서드
}

class Tv3 {
    public static void main(String args[]) {
        Tv3 t1 = new Tv3();
        Tv3 t2 = new Tv3();
        System.out.println("t1의 channel값은 " + t1.channel + "입니다.");
        System.out.println("t2의 channel값은 " + t2.channel + "입니다.");
    }
}
```

```

        t2 = t1;           // t1에 저장된 값(주소)을 t2에 저장
        t1.channel = 7; // channel 값을 7로 변경
        System.out.println("t1의 channel값은 7로 변경하였습니다.");
        System.out.println("t1의 channel값은 " + t1.channel + "입니다.");
        System.out.println("t2의 channel값은 " + t2.channel + "입니다.");
    }
}

```

▼ 실행결과

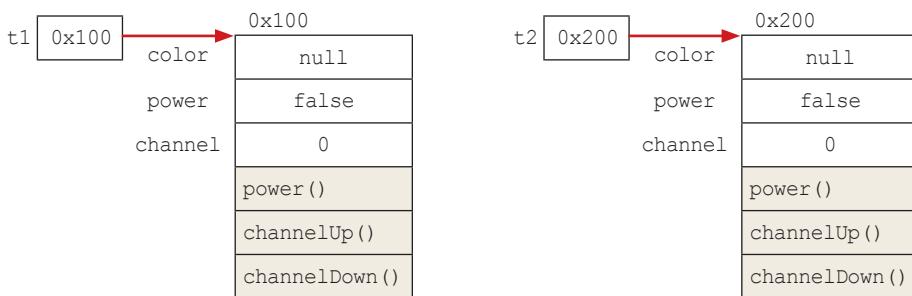
```

t1의 channel값은 0입니다.
t2의 channel값은 0입니다.
t1의 channel값을 7로 변경하였습니다.
t1의 channel값은 7입니다.
t2의 channel값은 7입니다.

```

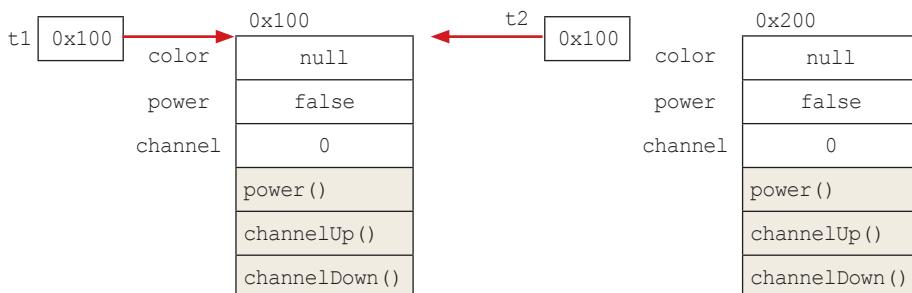
이 예제의 실행과정을 그림과 함께 설명하면 다음과 같다.

1. `Tv3 t1 = new Tv3();`
`Tv3 t2 = new Tv3();`



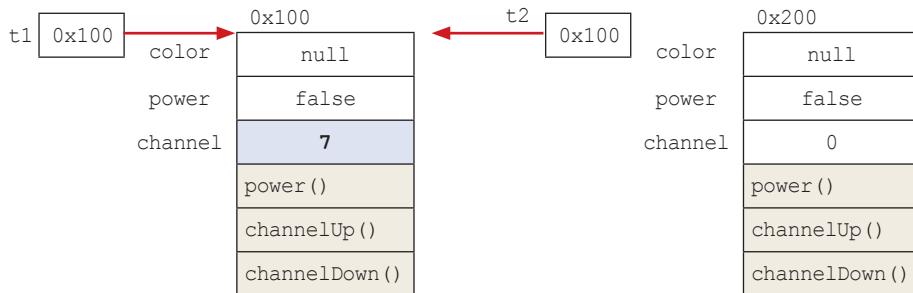
2. `t2 = t1;` // t1이 저장하고 있는 값(주소)을 t2에 저장한다.

t1은 참조 변수이므로, 인스턴스의 주소를 저장하고 있다. 이 문장이 실행되면, t2가 가지고 있던 값을 잃어버리게 되고 t1에 저장되어 있던 값이 t2에 저장된다. 그러면 t2 역시 t1이 참조하고 있던 인스턴스를 같이 참조하게 되고, t2가 원래 참조하고 있던 인스턴스는 더 이상 사용할 수 없게 된다.



| 참고 | 자신을 참조하고 있는 참조 변수가 하나도 없는 인스턴스는 더 이상 사용될 수 없으므로 ‘가비지 컬렉터(garbage collector)’에 의해서 자동으로 메모리에서 제거된다.

3. `t1.channel = 7;` // channel 값을 7로 변경



4. `System.out.println("t1의 channel값은 "+t1.channel+"입니다.");`

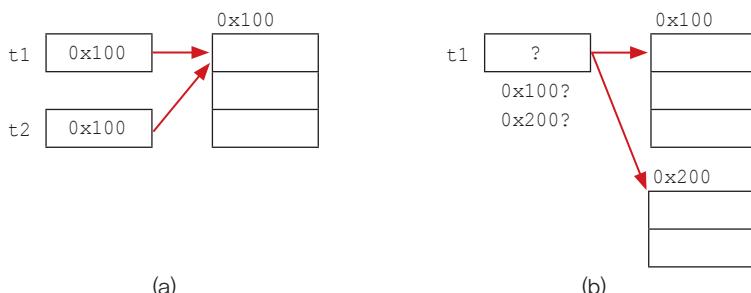
`System.out.println("t2의 channel값은 "+t2.channel+"입니다.");`

이제 t1, t2 모두 같은 Tv인스턴스를 가리키기 때문에 t1.channel과 t2.channel의 값은 7이며 다음과 같은 결과가 화면에 출력된다.

t1의 channel값은 7입니다.

t2의 channel값은 7입니다.

이 예제에서 알 수 있듯이, 하나의 참조 변수에 하나의 값(주소)만이 저장될 수 있으므로 둘 이상의 참조 변수가 하나의 인스턴스를 가리키는(참조하는) 것은 가능하지만 하나의 참조 변수로 여러 개의 인스턴스를 가리키는 것은 가능하지 않다.



- (a) 하나의 인스턴스를 여러 개의 참조 변수가 가리키는 경우(가능)
- (b) 여러 인스턴스를 하나의 참조 변수가 가리키는 경우(불가능)

▲ 그림 6-2 참조 변수와 인스턴스의 관계

2.5 객체 배열

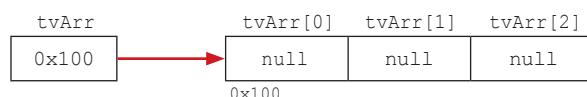
다수의 객체를 다뤄야 할 때, 배열을 이용하면 편리할 것이다. 객체 역시 배열로 다루는 것이 가능하며, 이를 ‘객체 배열’이라고 한다. 그렇다고 객체 배열 안에 객체가 저장되는 것은 아니고, 객체의 주소가 저장된다. 사실 객체 배열은 참조 변수들을 하나로 묶은 참조

변수 배열인 것이다.



길이가 3인 객체 배열 tvArr을 아래와 같이 생성하면, 각 요소는 참조 변수의 기본값인 null로 자동 초기화 된다. 그리고 이 객체 배열은 3개의 객체, 엄밀히 말하면 객체의 주소,를 저장할 수 있다.

```
Tv[] tvArr = new Tv[3]; // 길이가 3인 Tv타입의 참조 변수 배열
```



위의 그림에서 알 수 있듯이 객체 배열을 생성하는 것은, 그저 객체를 다루기 위한 참조 변수들이 만들어진 것일 뿐, 아직 객체가 생성되지 않았다. 객체를 생성해서 객체 배열의 각 요소에 저장하는 것을 잊으면 안 된다. 지금은 이런 실수를 안 할 것 같지만, 객체 배열에서 제일 많이 받는 질문이 객체 배열만 생성해 놓고 ‘분명히 객체를 생성했는데, 에러가 발생한다.’는 것이다.

```
Tv[] tvArr = new Tv[3]; // 참조 변수 배열(객체 배열)을 생성
```

```
// 객체를 생성해서 배열의 각 요소에 저장
tvArr[0] = new Tv();
tvArr[1] = new Tv();
tvArr[2] = new Tv();
```

배열의 초기화 블럭을 사용하면, 다음과 같이 한 줄로 간단히 할 수 있다.

```
Tv[] tvArr = { new Tv(), new Tv(), new Tv() };
```

다뤄야할 객체의 수가 많을 때는 for문을 사용하면 된다.

```
Tv[] tvArr = new Tv[100];

for(int i=0;i<tvArr.length;i++) {
    tvArr[i] = new Tv();
}
```

모든 배열이 그렇듯이 객체 배열도 같은 타입의 객체만 저장할 수 있다. 그러면, 여러 종

▼ 예제 6-4/TvEx4.java

```

class TvEx4 {
    public static void main(String args[]) {
        Tv4[] tvArr = new Tv4[3]; // 길이가 3인 Tv객체 배열

        // Tv객체를 생성해서 Tv객체 배열의 각 요소에 저장
        for(int i=0; i < tvArr.length;i++) {
            tvArr[i] = new Tv4();
            tvArr[i].channel = i+10; // tvArr[i]의 channel에 i+10을 저장
        }

        for(int i=0; i < tvArr.length;i++) {
            tvArr[i].channelUp(); // tvArr[i]의 메서드를 호출. 채널이 1증가
            System.out.printf("tvArr[%d].channel=%d\n",i,
                tvArr[i].channel);
        }
    } // main의 끝
}

class Tv4 {
    String color;           // 색상
    boolean power;          // 전원상태(on/off)
    int channel;            // 채널

    void power() { power = !power; }
    void channelUp() { ++channel; }
    void channelDown() { --channel; }
}

```

▼ 실행결과

```

tvArr[0].channel=11
tvArr[1].channel=12
tvArr[2].channel=13

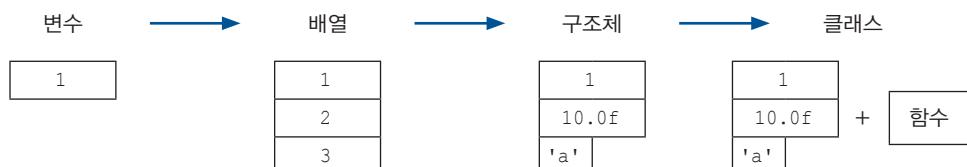
```

2.6 클래스의 또 다른 정의

클래스는 ‘객체를 생성하기 위한 틀’이며 ‘클래스는 속성과 기능으로 정의되어있다.’고 했다. 이것은 객체지향이론 관점에서의 정의이고, 이번엔 프로그래밍적인 관점에서 클래스의 정의와 의미를 살펴보자.

1. 클래스 – 데이터와 함수의 결합

프로그래밍언어에서 데이터 처리를 위한 데이터 저장형태의 발전과정은 다음과 같다.



▲ 그림 6-3 데이터 저장 개념의 발전 과정

1. **변수** 하나의 데이터를 저장할 수 있는 공간
2. **배열** 같은 종류의 여러 데이터를 하나의 집합으로 저장할 수 있는 공간
3. **구조체** 서로 관련된 여러 데이터를 종류에 관계없이 하나의 집합으로 저장할 수 있는 공간
4. **클래스** 데이터와 함수의 결합(구조체 + 함수)

하나의 데이터를 저장하기 위해 변수를, 그리고 같은 종류의 데이터를 보다 효율적으로 다루기 위해서 배열이라는 개념을 도입했으며, 후에는 구조체(structure)가 등장하여 자료형의 종류에 상관없이 서로 관계가 깊은 변수들을 하나로 묶어서 다룰 수 있게 했다.

그동안 데이터와 함수가 서로 관계가 없는 것처럼 데이터는 데이터끼리, 함수는 함수끼리 따로 다루어져 왔지만, 사실 함수는 주로 데이터를 가지고 작업을 하기 때문에 많은 경우에 있어서 데이터와 함수는 관계가 깊다.

그래서 자바와 같은 객체지향언어에서는 변수(데이터)와 함수를 하나의 클래스에 정의하여 서로 관계가 깊은 변수와 함수들을 함께 다룰 수 있게 했다.

서로 관련된 변수들을 정의하고 이들에 대한 작업을 수행하는 함수들을 함께 정의한 것이 바로 클래스이다. C언어에서는 문자열을 문자의 배열로 다루지만, 자바에서는 String이라는 클래스로 문자열을 다룬다. 문자열을 단순히 문자의 배열로 정의하지 않고 클래스로 정의한 이유는 문자열과 문자열을 다루는데 필요한 함수들을 함께 묶기 위해서이다.

```
public final class String implements java.io.Serializable, Comparable {
    private char[] value;           // 문자열을 저장하기 위한 공간

    public String replace(char oldChar, char newChar) {
        ...
        char[] val = value;        // 같은 클래스 내의 변수를 사용해서 작업을 한다.
        ...
    }
    ...
}
```

위 코드는 String클래스의 실제 소스의 일부이다. 클래스 내부에 value라는 문자 배열이 선언되어 있고, 문자열을 다루는 데 필요한 함수들을 함께 정의해 놓았다. 문자열의 일부를 뽑아내는 함수나 문자열의 길이를 알아내는 함수들은 항상 작업 대상으로 문자열을 필요로 하기 때문에 문자열과 깊은 관계에 있어서 함께 정의하였다.

이렇게 하면 변수와 함수가 서로 연결되어 작업이 간단하고 명료해진다.

2. 클래스 – 사용자정의 타입(user-defined type)

프로그래밍언어에서 제공하는 기본형(primitive type)외에 프로그래머가 서로 관련된 변수들을 묶어서 하나의 타입으로 새로 추가하는 것을 사용자정의 타입(user-defined type)이라고 한다.

많은 프로그래밍언어에서 사용자정의 타입을 정의할 수 있는 방법을 제공하고 있으며 자바와 같은 객체지향언어에서는 클래스가 곧 사용자 정의 타입이다. 기본형의 개수는 8

[ebook – 샘플. 무료 공유] 자바의 정석 4판 Java 21 올컬러 2025. 7. 7 출시 seong.namkung@gmail.com

개로 정해져 있지만 참조형의 개수가 정해져 있지 않은 이유는 이처럼 프로그래머가 새로운 타입을 추가할 수 있기 때문이다.

```
int hour;           // 시간을 표현하기 위한 변수
int minute;        // 분을 표현하기 위한 변수
float second;      // 초를 표현하기 위한 변수, 1/100초까지 표현하려고 float로 했다.
```

하나의 시간을 표현하기 위해 위와 같이 3개의 변수를 선언하였다. 만일 3개의 시간을 다뤄야 한다면 다음과 같이 해야 할 것이다.

```
int     hour1, hour2, hour3;
int     minute1, minute2, minute3;
float   second1, second2, second3;
```

이처럼 다뤄야 하는 시간의 개수가 늘어날 때마다 시, 분, 초를 위한 변수를 추가해야 하는데 데이터의 개수가 많으면 이런 식으로는 곤란하다.

```
int[]   hour    = new int[3];
int[]   minute  = new int[3];
float[] second  = new float[3];
```

위와 같이 배열로 처리하면 다뤄야 하는 시간 데이터의 개수가 늘어나더라도 배열의 크기만 변경해주면 되므로, 변수를 매번 새로 선언해줘야 하는 불편함과 복잡함은 없어졌다. 그러나 하나의 시간을 구성하는 시, 분, 초가 분리되어 있어서 실행 중에 시, 분, 초가 따로 섞여서 옮바르지 않은 데이터가 될 가능성이 있다. 이런 경우 시, 분, 초를 하나로 묶는 사용자정의 타입, 즉 클래스를 정의하여 사용하는 것이 좋다.

```
class Time { // 세 개의 변수를 하나로 묶는 새로운 타입을 정의
    int   hour;
    int   minute;
    float second;
}
```

위의 코드는 시, 분, 초를 저장하기 위한 세 변수를 멤버 변수로 갖는 Time클래스를 정의한 것이다. 사용자정의 타입 Time클래스를 이용해서 코드를 작성하면 아래와 같다.

비객체지향적 코드	객체지향적 코드
<pre>int hour1, hour2, hour3; int minute1, minute2, minute3; float second1, second2, second3;</pre>	<pre>Time t1 = new Time(); Time t2 = new Time(); Time t3 = new Time();</pre>
<pre>int[] hour = new int[3]; int[] minute = new int[3]; float[] second = new float[3];</pre>	<pre>Time[] t = new Time[3]; t[0] = new Time(); t[1] = new Time(); t[2] = new Time();</pre>

▲ 표 6-2 비객체지향적 코드와 객체지향적 코드의 비교

[ebook – 샘플. 무료 공유] 자바의 정석 4판 Java 21 올컬러 2025. 7. 7 출시 seong.namkung@gmail.com

[ebook – 샘플. 무료 공유] 자바의 정석 4판 Java 21 올컬러 2025. 7. 7 출시 seong.namkung@gmail.com

이제 시, 분, 초가 하나의 단위로 묶여서 다루어지기 때문에 다른 시간 데이터와 섞이는 일은 없겠지만, 시간 데이터에는 다음과 같은 추가적인 제약조건이 있다.

1. 시, 분, 초는 모두 0보다 크거나 같아야 한다.
2. 시의 범위는 0~23, 분과 초의 범위는 0~59 이다.

이러한 조건들이 모두 프로그램 코드에 반영될 때, 보다 정확한 데이터를 유지할 수 있을 것이다. 객체지향언어가 아닌 언어에서는 이러한 추가적인 조건들을 반영하기가 어렵다.

그러나 객체지향언어에서는 제어자와 메서드를 이용해서 이러한 조건들을 코드에 쉽게 반영할 수 있다.

아직 제어자에 대해서 배우지는 않았지만, 위의 조건들을 반영하여 Time클래스를 작성해 보았다. 가볍게 참고만 하기 바란다.

```
public class Time {
    private int hour; // 제어자 private를 붙이면 외부에서 접근 불가
    private int minute;
    private float second;

    public int getHour() { return hour; }
    public int getMinute() { return minute; }
    public float getSecond() { return second; }

    public void setHour(int h) { // public메서드로만 hour 변경 가능
        if (h < 0 || h > 23) return;
        hour = h; // 조건에 맞는 값일 때만 변경 가능
    }

    public void setMinute(int m) {
        if (m < 0 || m > 59) return;
        minute = m;
    }

    public void setSecond(float s) {
        if (s < 0.0f || s > 59.99f) return;
        second = s;
    }
}
```

제어자를 이용해서 변수의 값을 직접 변경하지 못하도록 하고 대신 메서드를 통해서 값을 변경하도록 작성하였다. 값을 변경할 때 지정된 값의 유효성을 if문으로 점검한 다음에 유효한 값일 경우에만 변경된다..

이 외에도 시간의 차를 구하는 메서드와 같이 시간과 관련된 메서드를 추가로 정의하여 Time클래스를 향상시켜 보는 것도 좋은 프로그래밍 공부거리가 될 것이다.

[ebook – 샘플. 무료 공유] 자바의 정석 4판 Java 21 올컬러 2025. 7. 7 출시 seong.namkung@gmail.com

3. 변수와 메서드

3.1 선언위치에 따른 변수의 종류

변수는 클래스 변수, 인스턴스 변수, 지역 변수 모두 세 종류가 있다. 변수의 종류를 결정짓는 중요한 요소는 '변수의 선언된 위치'이므로 변수의 종류를 파악하려면, 변수가 어느 영역에 선언되었는지 확인하는 것이 중요하다.

영역은 클래스 영역과 메서드 영역 두 가지뿐이며, 메서드 영역이 아니면 클래스 영역이라고 생각하면 쉽다. 메서드 영역에 선언된 변수는 지역 변수(lv)이며, 그 외에는 모두 멤버 변수다. 멤버 변수 중 static이 붙은 것은 클래스 변수(cv), 붙지 않은 것은 인스턴스 변수(iv)이다. 그리고 인스턴스 변수와 클래스 변수의 타입으로 var를 사용할 수 없다.

```
class MyClass
{
    int iv;          // 인스턴스 변수
    static int cv;   // 클래스 변수(static 변수, 공유 변수)

    void method()
    {
        int lv = 0;  // 지역 변수
    }
}
```

변수의 종류	선언 위치	생성 시기
클래스 변수 (class variable)	클래스 영역	클래스가 메모리에 올라갈 때
인스턴스 변수 (instance variable)		인스턴스가 생성될 때
지역 변수 (local variable)	클래스 영역 이외의 영역 (메서드, 생성자, 초기화 블럭 내부)	변수 선언문이 수행될 때

▲ 표 6-3 변수의 종류와 특징

1. 인스턴스 변수(iv, instance variable)

클래스 영역에 선언되며, 인스턴스를 생성할 때 만들어진다. 그래서 인스턴스 변수(iv)를 사용하려면 먼저 인스턴스를 생성해야 한다.

각 인스턴스는 독립적인 저장 공간을 가지므로 서로 다른 값을 가질 수 있다. 인스턴스마다 고유한 상태를 유지해야하는 속성의 경우, 인스턴스 변수(iv)로 선언한다.

2. 클래스 변수(cv, class variable)

클래스 변수는 인스턴스 변수 앞에 static을 붙인 것이다. 인스턴스 변수와 달리, 클래스 변수는 모든 인스턴스가 저장 공간(변수)을 공유한다. 같은 클래스의 모든 인스턴스들이 공통적인 값을 유지해야하는 속성의 경우, 클래스 변수로 선언해야 한다.

[ebook – 샘플. 무료 공유] 자바의 정석 4판 Java 21 올컬러 2025. 7. 7 출시 seong.namkung@gmail.com

클래스 변수는 인스턴스를 생성하지 않고도 언제라도 바로 사용할 수 있으며, ‘클래스 이름.클래스 변수’와 같은 형식으로 사용한다. 예를 들어 클래스 MyClass의 클래스 변수 cv를 사용하려면 ‘MyClass.cv’와 같이 하면 된다.

클래스 변수는 클래스가 메모리에 ‘로딩/loading’될 때 자동 생성되어 프로그램이 종료될 때까지 유지되며, public을 앞에 붙이면 같은 프로그램 내에서 어디서나 접근할 수 있는 ‘전역 변수(global variable)’의 성격을 갖는다.

| 참고 | 참조 변수의 선언이나 객체의 생성과 같이 클래스의 정보가 필요할 때, 클래스는 메모리에 로딩된다.

3. 지역 변수(local variable, lv)

메서드 내에 선언되어 메서드 내에서만 사용 가능하며, 메서드가 호출되면 만들어졌다가 메서드가 종료되면 소멸된다. for문 또는 while문의 블럭 내에 선언된 지역 변수는, 지역 변수가 선언된 블럭{} 내에서만 사용 가능하며, 블럭{}을 벗어나면 소멸되어 사용할 수 없게 된다. 우리가 6장 이전에 선언한 변수들은 모두 main메서드의 지역 변수다.

3.2 클래스 변수와 인스턴스 변수

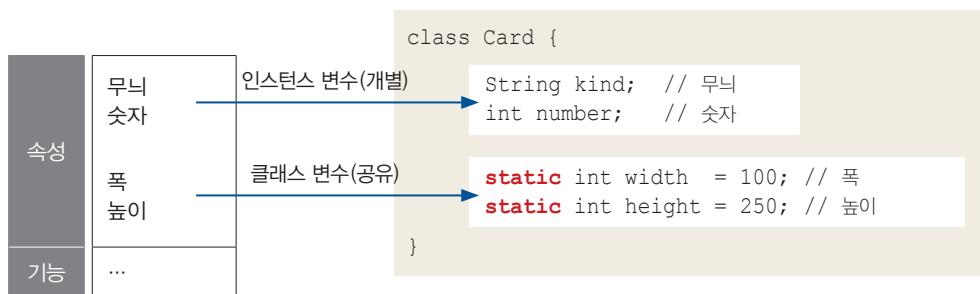
클래스 변수와 인스턴스 변수의 차이를 이해하기 위한 예로, 카드 게임의 카드를 클래스로 정의해보자.



▲ 그림 6-4 게임 카드

카드 클래스를 작성하기 위해 먼저 카드를 분석해서 속성과 기능을 알아내야 한다. 속성으로는 카드의 무늬, 숫자, 폭, 높이 정도를 생각할 수 있을 것이다.

이 중에서 어떤 속성을 클래스 변수로 선언할지, 또 어떤 속성들을 인스턴스 변수로 선언할 것인지 한번 생각해보자.



[ebook – 샘플. 무료 공유] 자바의 정석 4판 Java 21 올컬러 2025. 7. 7 출시 seong.namkung@gmail.com

[ebook – 샘플. 무료 공유] 자바의 정석 4판 Java 21 올컬러 2025. 7. 7 출시 seong.namkung@gmail.com

각 Card인스턴스는 자신만의 무늬(kind)와 숫자(number)를 유지하고 있어야 하므로 이들을 인스턴스 변수로 선언하였고, 각 카드의 폭(width)과 높이(height)는 모든 인스턴스가 공통적으로 같은 값을 유지해야하므로 클래스 변수로 선언하였다.

이렇게 하면 카드의 폭을 변경해야 할 때, 모든 카드의 width값을 변경하지 않고 한 카드의 width값만 변경해도 모든 카드의 width값이 변경된다.

▼ 예제 6-5/CardEx.java

```
class CardEx {
    public static void main(String args[]) {
        System.out.println("Card.width = " + Card.width);
        System.out.println("Card.height = " + Card.height);

        Card c1 = new Card();
        c1.kind = "Heart";
        c1.number = 7;

        Card c2 = new Card();
        c2.kind = "Spade";
        c2.number = 4;
    }

    System.out.println("c1은 " + c1.kind + ", " + c1.number
                      + "이며, 크기는 (" + c1.width + ", " + c1.height + ")" );
    System.out.println("c2는 " + c2.kind + ", " + c2.number
                      + "이며, 크기는 (" + c2.width + ", " + c2.height + ")" );
    System.out.println("c1의 width와 height를 각각 50, 80으로 변경합니다.");
    c1.width = 50;
    c1.height = 80;
}

class Card {
    String kind;
    int number;
    static int width = 100;
    static int height = 250;
}
```

클래스 변수(static변수)는 객체생성 없이 '클래스 이름.클래스 변수'로 직접 사용 가능하다.

인스턴스 변수의 값을 변경한다.

클래스 변수의 값을 변경한다.

▼ 실행결과

Card.width = 100
 Card.height = 250
 c1은 Heart, 7이며, 크기는 (100, 250)
 c2는 Spade, 4이며, 크기는 (100, 250)
 c1의 width와 height를 각각 50, 80으로 변경합니다.
 c1은 Heart, 7이며, 크기는 (50, 80)
 c2는 Spade, 4이며, 크기는 (50, 80)

| 플레이어 영상 | MemberVar.exe는 예제6-5의 실행 과정을 설명과 함께 보여준다.

Card클래스의 클래스 변수(static변수)인 width, height는 Card클래스의 인스턴스를 생성하지 않고도 '클래스 이름.클래스 변수'와 같은 방식으로 사용할 수 있다.

Card인스턴스인 c1과 c2는 클래스 변수인 width와 height를 공유하기 때문에, c1의 width와 height를 변경하면 c2의 width와 height값도 바뀐 것과 같은 결과를 얻는다.

Card.width, c1.width, c2.width는 모두 같은 저장 공간을 참조하므로 항상 같은 값을 갖게 된다.

[ebook – 샘플. 무료 공유] 자바의 정석 4판 Java 21 올컬러 2025. 7. 7 출시 seong.namkung@gmail.com

[ebook – 샘플. 무료 공유] 자바의 정석 4판 Java 21 올컬러 2025. 7. 7 출시 seong.namkung@gmail.com

클래스 변수를 사용할 때는 Card.width와 같이 ‘클래스 이름.클래스 변수’의 형태로 하는 것이 좋다. 참조 변수 c1, c2를 통해서도 클래스 변수를 사용할 수 있지만 이렇게 하면 클래스 변수를 인스턴스 변수로 오해하기 쉽기 때문이다.

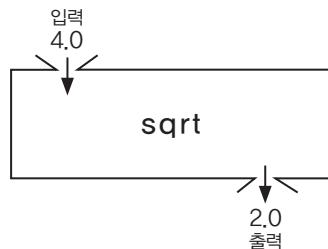
인스턴스 변수는 인스턴스가 생성될 때마다 생성되므로 다른 값을 갖지만,
클래스 변수는 모든 인스턴스가 하나의 저장공간을 공유하므로, 같은 값을 갖는다.

3.3 메서드

‘메서드(method)’는 특정 작업을 수행하는 문장들을 하나로 묶은 것이다. 기본적으로 수학의 함수와 유사하며, 어떤 값을 입력하면 이 값으로 작업을 수행해서 결과를 반환한다.

예를 들어 제곱근을 구하는 메서드 ‘Math.sqrt()’는 4.0을 입력하면, 2.0을 결과로 반환한다.

| 참고 | 메서드는 입력값 또는 출력값(결과값)이 없을 수도 있으며, 심지어는 입력값과 출력값이 모두 없을 수도 있다.



그저 메서드가 작업을 수행하는데 필요한 값만 주고 원하는 결과를 얻으면 될 뿐, 이 메서드가 내부적으로 어떤 과정을 거쳐 결과를 만들어내는지 전혀 몰라도 된다. 즉, 메서드에 넣을 값(입력)과 반환하는 결과(출력)만 알면 되는 것이다. 그래서 메서드를 내부가 보이지 않는 ‘블랙 박스(black box)’라고도 한다.

sqrt() 외에도 지금까지 빈번히 사용해온 println()이나 random()과 같은 메서드들 역시 내부적으로 어떻게 동작하는지 몰라도 사용하는데 아무런 어려움이 없었다.

메서드를 사용하는 이유

메서드를 통해서 얻는 이점은 여러 가지가 있지만 그 중에서 대표적인 세 가지가 있다. 아직 메서드를 배우기 전이므로 여기서 나열하는 이점들이 잘 와닿지 않을 수 있지만, 이 이점들을 염두에 두고 있으면 메서드를 더 깊게 이해하는데 도움이 될 것이다.

1. 높은 재사용성(reusability)

이미 Java API에서 제공하는 메서드들을 사용하며 경험한 것처럼 한번 만들어 놓은 메서드는 몇 번이고 호출할 수 있으며, 다른 프로그램에서도 사용이 가능하다.

2. 중복된 코드의 제거

프로그램을 작성하다보면, 같은 내용의 문장들이 여러 곳에 반복해서 나타나곤 한다. 이렇게 반복되는 문장을 끓어서 하나의 메서드로 작성해 놓으면, 반복되는 문장을 대신 메서드를 호출하는 문장으로 대체할 수 있다. 그러면, 전체 소스 코드의 길이도 짧아지고, 변경사항이 발생했을 때 수정해야 할 코드의 양이 줄어들어 오류가 발생할 가능성도 함께 줄어든다. 아래의 코드는 예제5-10을 약간 변경한 것인데, 배열을 출력하는 같은 내용의 문장이 두 번 반복된다.

```
public static void main(String args[]) {
    ...
    for (int i=0;i<10;i++)
        numArr[i] = (int) (Math.random()*10);

    for (int i=0;i<10;i++)
        System.out.printf("%d", numArr[i]);
    System.out.println();
    ...
    for (int i=0;i<10;i++)
        System.out.printf("%d", numArr[i]);
    System.out.println();
}
```

같은 내용의 코드가 반복됨.
변경할 때도 두 곳을 모두 수정해야 함.

이처럼 같은 내용의 문장들이 반복되면, 소스 코드도 길어지고 해당 문장에서 변경 사항이 발생했을 때 여러 곳을 고쳐야 하므로 일도 많아지고 오류를 만들어낼 가능성도 높다.
아래와 같이 printArr이라는 메서드를 만들어서 이전의 코드에 적용하면, 다음과 같다.

```
static void printArr(int[] numArr) {
    for(int i = 0;i<10;i++)
        System.out.printf("%d", numArr[i]);
    System.out.println();
}

public static void main(String args[]) {
    ...
    for(int i = 0;i<10;i++)
        numArr[i] = (int) (Math.random()*10);

    printArr(numArr); // 배열을 출력
    ...
    printArr(numArr); // 배열을 출력
}
```

이제 여기만 변경하면 된다.

반복되는 코드 대신 메서드를 호출

이처럼 반복적으로 나타나는 문장들을 메서드로 만들어 사용하면 코드의 중복이 제거되고, 변경사항이 발생했을 때 이 메서드만 수정하면 되므로 관리도 쉽고 오류의 발생 가능성도 낮아진다.

아직 메서드에 대해 배우지 않아서 이해가지 않는 부분이 있겠지만, 지금은 메서드의 장점으로 이런 것들이 있다는 정도만 이해하자.

3. 프로그램의 구조화

지금까지는 main메서드 안에 모든 문장을 넣는 식으로 프로그램을 작성해왔다. 길어야 100줄 정도 밖에 안 되는 작은 프로그램을 작성할 때는 이렇게 해도 별 문제가 없지만, 몇 천 줄, 몇 만 줄이 넘는 프로그램도 이런 식으로 작성할 수는 없다. 큰 규모의 프로그램에서는 문장들을 작업 단위로 나눠서 여러 개의 메서드에 담아 프로그램의 구조를 단순화시키는 것이 필수적이다. 예를 들어서 예제5-10을 작업 단위로 나누어서 메서드로 만들면, main메서드가 아래와 같이 간단해 진다.

```
public static void main(String args[]) {
    int[] numArr = new int[10];

    initArr(numArr); // 1. 배열을 초기화
    printArr(numArr); // 2. 배열을 출력
    sortArr(numArr); // 3. 배열을 정렬
    printArr(numArr); // 4. 배열을 출력
}
```

이처럼 main메서드는 프로그램의 전체 흐름이 한눈에 들어올 정도로 단순하게 구조화하는 것이 좋다. 그래야 나중에 프로그램에 문제가 발생해도 해당 부분을 쉽게 찾아서 해결 할 수 있다.

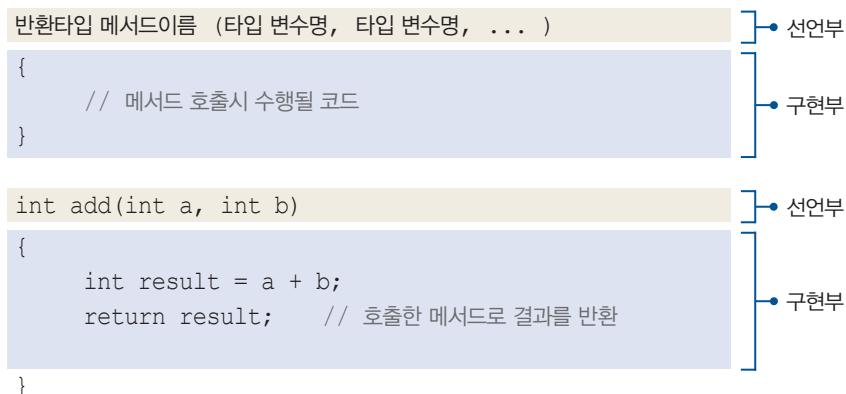
처음에 프로그램을 설계할 때 내용이 없는 메서드를 작업 단위로 만들어 놓고, 하나씩 완성해가는 것도 프로그램을 구조화하는 좋은 방법이다. 아래의 코드는 성적처리 프로그램을 설계한 것인데, 처음에 showMenu메서드를 호출해서 메뉴를 보여주고 선택한 메뉴에 따라 다른 작업을 하도록 작성하였다.

```
static int showMenu()      /* 나중에 내용을 완성한다.*/ }
static void inputRecord()  /* 나중에 내용을 완성한다.*/ }
static void changeRecord() /* 나중에 내용을 완성한다.*/ }
static void deleteRecord() /* 나중에 내용을 완성한다.*/ }
static void searchRecord() /* 나중에 내용을 완성한다.*/ }
static void showRecordList() /* 나중에 내용을 완성한다.*/ }

public static void main(String args[]) {
    ...
    switch(showMenu()) {
        case 1: inputRecord(); break; // 데이터를 입력받는 메서드
        case 2: changeRecord(); break; // 데이터를 변경하는 메서드
        case 3: deleteRecord(); break; // 데이터를 삭제하는 메서드
        case 4: searchRecord(); break; // 데이터를 검색하는 메서드
        default: showRecordList(); // 데이터의 목록을 보여주는 메서드
    }
}
```

3.4 메서드의 선언과 구현

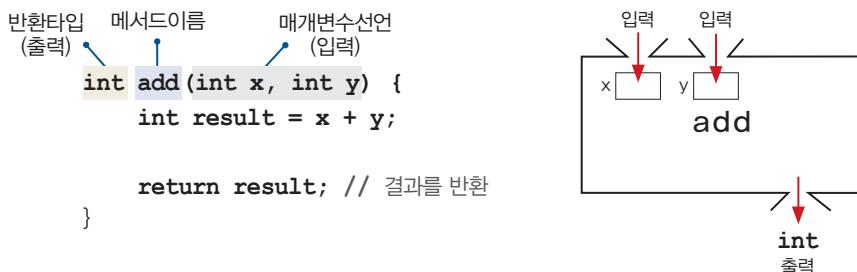
메서드는 크게 두 부분, ‘선언부(header, 머리)’와 ‘구현부(body, 몸통)’로 이루어져 있다. 메서드를 정의한다는 것은 선언부와 구현부를 작성하는 것을 뜻하며 다음과 같은 형식으로 메서드를 정의한다.



메서드 선언부(method declaration, 메서드 머리)

메서드 선언부는 ‘메서드의 이름’과 ‘매개변수 선언’, 그리고 ‘반환타입’으로 구성되어 있으며, 메서드가 작업을 수행하기 위해 어떤 값을 필요로 하고 작업의 결과로 어떤 타입의 값을 반환하는지에 대한 정보를 제공한다.

예를 들어 아래에 정의된 메서드 add는 두 개의 정수를 입력받아서, 두 값을 더한 결과(int타입의 값)를 반환한다.



메서드의 선언부는 후에 변경할 일이 생기지 않게 신중히 작성해야한다. 메서드의 선언부가 변경되면, 그 메서드가 호출되는 모든 곳을 같이 변경해야 하기 때문이다.

매개변수 선언(parameter declaration)

매개변수는 메서드가 작업을 수행하는데 필요한 값들(입력)을 제공받기 위한 것이며, 필요한 값의 개수만큼 변수를 선언하며 쉼표','를 구분자로 사용한다. 주의할 점은 일반적인 변수 선언과 달리 두 변수의 타입이 같아도 변수의 타입을 생략할 수 없다는 것이다.

```
int add(int x, int y) { ... } // OK.
int add(int x, y) { ... } // 에러. 매개변수 y의 타입이 없다.
```

선언할 수 있는 매개변수의 개수는 거의 제한이 없지만, 만일 입력해야 할 값의 개수가 많은 경우에는 배열이나 참조 변수를 사용하면 된다. 만일 값을 전혀 입력받을 필요가 없다면 팔호() 안에 아무 것도 적지 않는다.

| 참고 | 매개변수도 메서드 내에 선언된 것으로 간주되므로 '지역 변수(local variable)'이다.

메서드의 이름(method name)

메서드의 이름도 앞서 배운 변수의 명명규칙대로 작성하면 된다. 메서드는 특정 작업을 수행하므로 메서드의 이름은 'add'처럼 동사인 경우가 많으며, 이름만으로도 메서드의 기능을 쉽게 알 수 있게 의미있는 이름을 짓도록 노력해야 한다.

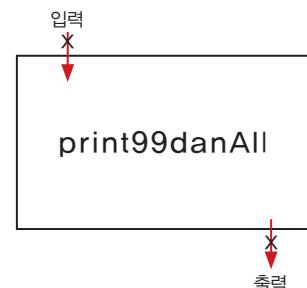
반환타입(return type)

메서드의 작업수행 결과(출력)인 '반환값(return value)'의 타입을 적는다. 반환값이 없는 경우 반환타입으로 'void'를 적어야한다.

아래에 정의된 메서드 'print99danAll'은 구구단 전체를 출력한다. 작업을 수행하는데 필요한 값도, 작업수행의 결과인 반환값도 없다. 그래서 반환타입이 'void'이다.

| 참고 | 'void'는 '아무 것도 없음'을 의미한다.

```
void print99danAll() {
    for(int i=1;i<=9;i++) {
        for(int j=2;j<=9;j++) {
            System.out.print(j+"*"+i+" = "+(j*i)+" ");
        }
        System.out.println();
    }
}
```



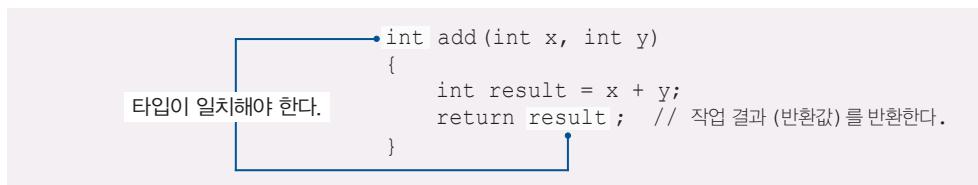
메서드의 구현부(method body, 메서드 몸통)

메서드의 선언부 다음에 오는 팔호{}를 '메서드의 구현부'라고 하는데, 여기에 메서드를 호출했을 때 수행될 문장들을 넣는다. 우리가 그동안 작성한 문장들은 모두 main 메서드의 구현부{}에 속한 것들이었으므로 지금까지 하던 대로만 하면 된다.

return문

메서드의 반환타입이 'void'가 아닌 경우, 구현부{} 안에 'return 반환값;'이 반드시 포함되어 있어야 한다. 이 문장은 작업을 수행한 결과인 반환값을 호출한 메서드로 전달하는데, 이 값의 타입은 반환타입과 일치하거나 자동 형변환이 가능한 것이어야 한다.

여러 변수를 선언할 수 있는 매개변수와 달리 return문은 단 하나의 값만 반환할 수 있는데, 메서드로의 입력(매개변수)은 여러 개일 수 있어도 출력(반환값)은 없거나 하나만 허용한다.



위의 코드에서 ‘return result;’는 변수 result에 저장된 값을 호출한 메서드로 반환한다. 변수 result의 타입이 int이므로 메서드 add의 반환타입과 일치하는 것을 알 수 있다.

지역 변수(local variable)

메서드 내에 선언된 변수들은 그 메서드 내에서만 사용할 수 있으므로 서로 다른 메서드라면 같은 이름의 변수를 선언해도 된다. 이처럼 메서드 내에 선언된 변수를 ‘지역 변수 (local variable)’라고 한다.

| 참고 | 매개변수도 메서드 내에 선언된 것으로 간주되므로 지역 변수이다.

```

int add(int x, int y) {
    int result = x + y;
    return result;
}

int multiply(int x, int y) {
    int result = x * y;
    return result;
}

```

위에 정의된 메서드 add와 multiply에 각기 선언된 변수, x, y, result는 이름만 같을 뿐 서로 다른 변수이다. 이처럼 영역이 다르면 구분할 수 있으므로 이름이 같아도 된다.

3.5 메서드의 호출

메서드를 정의했어도 호출하지 않으면 아무 일도 일어나지 않는다. 메서드를 호출해야만 구현부{}의 문장들이 수행된다. 메서드를 호출하는 방법은 다음과 같다.

| 참고 | main 메서드는 프로그램 실행 시 JVM에 의해 자동으로 호출된다.

메서드이름(값1, 값2, ...); // 메서드를 호출하는 방법

```

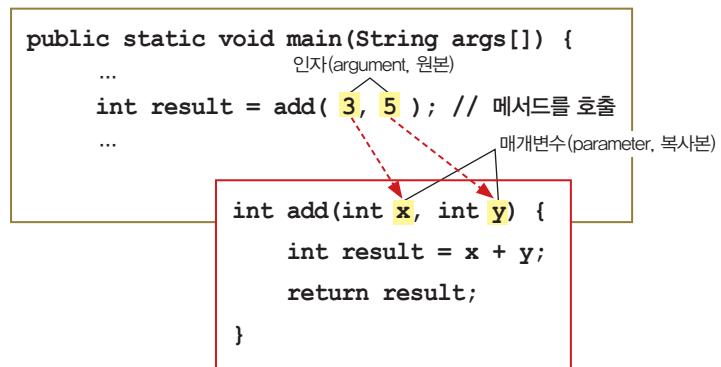
print99danAll();           // void print99danAll ()을 호출
int result = add(3, 5);   // int add(int x, int y)를 호출하고, 결과를 result에 저장

```

인자(argument)와 매개변수(parameter)

메서드를 호출할 때 괄호()안에 지정해준 값들을 ‘인자(argument)’라고 하는데, 인자의 개수와 순서는 호출된 메서드에 선언된 매개변수와 일치해야 한다.

그리고 인자는 메서드가 호출되면서 매개변수에 대입되므로, 인자의 타입은 매개변수의 타입과 일치하거나 자동 형변환이 가능한 것이어야 한다.



만일 아래와 같이 메서드에 선언된 매개변수의 개수보다 많은 값을 괄호()에 넣거나 타입이 다른 값을 넣으면 컴파일러가 에러를 발생시킨다.

```

int result = add(1, 2, 3); // 에러. 메서드에 선언된 매개변수의 개수가 다름
int result = add(1.0, 2.0); // 에러. 메서드에 선언된 매개변수의 타입이 다름

```

반환타입이 void가 아닌 경우, 메서드가 작업을 수행하고 반환한 값을 대입 연산자로 변수에 저장하는 것이 보통이지만, 원하지 않으면 저장하지 않아도 된다.

```

int result = add(3, 5); // int add(int x, int y)의 호출 결과를 result에 저장
add(3, 5);             // OK. 메서드 add가 반환한 결과를 사용하지 않아도 된다.

```

메서드의 실행흐름

같은 클래스 내의 메서드끼리 참조변수를 사용하지 않고도 서로 호출이 가능하지만 static 메서드는 같은 클래스 내의 인스턴스 메서드를 호출할 수 없다.

다음은 두 개의 값을 매개변수로 받아서 사칙연산을 수행하는 4개의 메서드를 가진 MyMath클래스를 정의한 것이다.

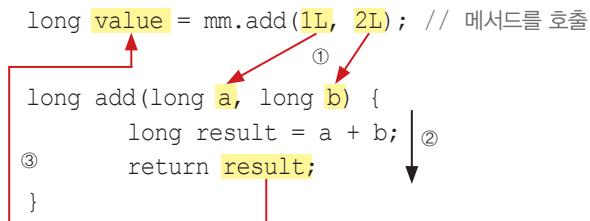
```

class MyMath {
    long add(long a, long b) {
        long result = a+b;
        return result;
        //    return a + b; // 위의 두 줄을 한 줄로 간단히 할 수 있다.
    }
    long subtract(long a, long b)      {    return a - b;    }
    long multiply(long a, long b)     {    return a * b;    }
    double divide(double a, double b) {    return a / b;    }
}

```

MyMath클래스의 ‘add(long a, long b)’를 호출하려면 먼저 ‘MyMath mm = new MyMath();’로, MyMath클래스의 인스턴스를 생성해야 한다.

```
MyMath mm = new MyMath(); // 먼저 인스턴스를 생성
```



① 메서드 add를 호출한다. 호출시 지정한 1L, 2L이 매개변수 a, b에 각각 복사(대입)된다.

② 괄호{ }안에 있는 문장들이 순서대로 수행된다.

③ 괄호{ }안의 모든 문장이 실행되거나 return문을 만나면, 호출한 메서드(main메서드)로 되돌아와서 이후의 문장들을 실행한다.

메서드가 호출되면 지금까지 실행 중이던 메서드는 실행을 잠시 멈추고 호출된 메서드의 문장들이 실행된다. 호출된 메서드의 작업이 모두 끝나면, 다시 호출한 메서드로 돌아와 이후의 문장들을 실행한다.

위의 그림에서는 편의상 메서드 add의 호출결과가 바로 value에 저장되는 것처럼 그렸지만, 사실은 호출한 자리를 반환값이 대신하고 대입연산자에 의해 이 값이 변수 value에 저장된다.

```
long value = add(1L, 2L);
→ long value = 3L;
```

add메서드의 매개변수의 타입이 long이므로 호출할 때 long 또는 long으로 자동 형변환 가능한 값을 제공해야 한다. 제공된 값은 메서드의 매개변수로 복사된다. 위의 코드에서는 1L과 2L의 값이 long타입의 매개변수 a와 b에 각각 복사된다.

메서드는 호출시 넘겨받은 값으로 작업을 수행하고 그 결과를 반환하면서 종료된다. 반환된 값은 대입 연산자에 의해서 변수 value에 저장된다. 메서드의 결과를 저장하기 위한 변수 value역시 반환값과 같은 타입이거나 반환값이 자동 형변환되어 저장될 수 있는 타입이어야 한다.

▼ 예제 6-6/MyMathEx.java

```
class MyMathEx {
    public static void main(String args[]) {
        MyMath mm = new MyMath();
        long result1 = mm.add(5L, 3L);
```

```

long result2 = mm.subtract(5L, 3L);
long result3 = mm.multiply(5L, 3L);
double result4 = mm.divide(5L, 3L); •
System.out.println("add(5L, 3L) = " + result1);
System.out.println("subtract(5L, 3L) = " + result2);
System.out.println("multiply(5L, 3L) = " + result3);
System.out.println("divide(5L, 3L) = " + result4);
}

class MyMath {
    long add(long a, long b) {
        long result = a+b;
        return result;
        // return a + b; // 위의 두 줄을 한 줄로 간단히 할 수 있다.
    }

    long subtract(long a, long b) { return a - b; }
    long multiply(long a, long b) { return a * b; }
    double divide(double a, double b) {
        return a / b;
    }
}

```

double 대신 long값으로 호출하였다. 이 값은 double로 자동 형변환된다.

▼ 실행결과

```

add(5L, 3L) = 8
subtract(5L, 3L) = 2
multiply(5L, 3L) = 15
divide(5L, 3L) = 1.6666666666666667

```

사칙연산을 위한 4개의 메서드가 정의되어 있는 MyMath 클래스를 이용한 예제이다. 이 예제를 통해서 클래스에 선언된 메서드를 어떻게 호출하는지 알 수 있을 것이다.

여기서 눈여겨봐야 할 곳은 divide(double a, double b)를 호출하는 부분이다. divide 메서드에 선언된 매개변수 타입은 double형인데, 이와 다른 long형의 값인 5L과 3L을 사용해서 호출하는 것이 가능하다.

```

double result4 = mm.divide( 5L , 3L ); // double 대신 long을 지정
double divide(double a, double b) { // long이 double로 자동 형변환
    return a / b; // return 5.0 / 3.0;
}

```

호출 시에 입력된 값은 메서드의 매개변수에 대입되는 값이므로, long형의 값을 double형 변수에 저장하는 것과 같아서 ‘double a = 5L;’을 수행 했을 때와 같이 long형의 값인 5L은 double형 값인 5.0으로 자동 형변환되어 divide의 매개변수 a에 저장된다.

그래서, divide 메서드에 두 개의 정수값(5L, 3L)을 입력하여 호출하였음에도 불구하고 연산결과가 double형의 값이 된다.

이와 마찬가지로 add(long a, long b) 메서드에도 매개변수 a, b에 int형의 값을 넣어 add(5, 3)과 같이 호출하는 것이 가능하다.

3.6 return문

return문은 현재 실행중인 메서드를 종료하고 호출한 메서드로 되돌아간다. 지금까지 반환값이 있을 때만 return문을 썼지만, 원래는 반환값의 유무에 관계없이 모든 메서드에는 적어도 하나의 return문이 있어야 한다. 그런데도 반환타입이 void인 경우, return문 없이도 아무런 문제가 없었던 이유는 컴파일러가 메서드의 마지막에 ‘return;’을 자동적으로 추가해주었기 때문이다.

```
void printGugudan(int dan) {
    for(int i = 1; i <= 9; i++) {
        System.out.printf("%d * %d = %d%n", dan, i, dan * i);
    }
    // return; // 반환 타입이 void이므로 생략 가능. 컴파일러가 자동 추가
}
```

그러나 반환타입이 void가 아닌 경우, 즉 반환값이 있는 경우, 반드시 return문이 있어야 한다. return문이 없으면 컴파일 에러(error: missing return statement)가 발생한다.

```
int multiply(int x, int y) {
    int result = x * y;

    return result; // 반환타입이 void가 아니므로 생략 불가
}
```

아래의 코드는 두 값 중에서 큰 값을 반환하는 메서드이다. 이 메서드의 리턴타입이 int이고 int타입의 값을 반환하는 return문이 있지만, return문이 없다는 에러가 발생한다. 왜냐하면 if문 조건식의 결과에 따라 return문이 실행되지 않을 수도 있기 때문이다.

```
int max(int a, int b) {
    if(a > b)
        return a; // 에러. 조건식이 참일 때만 실행된다.
}
```

그래서 이런 경우 다음과 같이 if문의 else블럭에 return문을 추가해서, 항상 결과값이 반환되도록 해야 한다. 이런 실수를 안할 것 같지만 코드가 길어지면 장담할 수 없다.

```
int max(int a, int b) {
    if(a > b)
        return a;
    else
        return b;
}
```

반환값(return value)

return문의 반환값으로 주로 변수가 오긴 하지만 항상 그런 것은 아니다. 아래 왼쪽의 코드는 오른쪽과 같이 간략히 할 수 있는데, 오른쪽의 코드는 return문의 반환값으로 ‘ $x+y$ ’라는 식(式)이 적혀있다. 그렇다고 해서 식이 반환되는 것은 아니고, 이 식을 계산한 결과가 반환된다.

```
int add(int x, int y) {
    int result = x + y;
    return result;
}
```

```
int add(int x, int y) {
    return x + y;
}
```

| 참고 | 수학에서처럼, result의 값이 ' $x+y$ '와 같으므로 result대신 ' $x+y$ '를 쓸 수 있다고 생각하면 이해하기 쉽다.

예를 들어 매개변수 x와 y의 값이 각각 3과 5라면, ‘return $x+y$;’는 다음과 같은 계산과정을 거쳐서 반환값은 8이 된다.

```
return x + y;
→ return 3 + 5;
→ return 8;
```

아래의 diff메서드는 두 개의 정수를 받아서 그 차이를 절대값으로 반환한다. 오른쪽 코드 역시 메서드를 반환하는 것이 아니라 메서드 abs를 호출하고, 그 결과를 받아서 반환한다. 메서드 abs의 반환타입이 메서드 diff의 반환타입과 일치하기 때문에 가능하다는 것에 주의하자.

```
int diff(int x, int y) {
    int result = abs(x-y);
    return result;
}
```

```
int diff(int x, int y) {
    return abs(x-y);
}
```

간단한 메서드의 경우 if문 대신 조건 연산자를 사용하기도 한다. 메서드 abs는 입력받은 정수의 부호를 판단해서 음수일 경우 부호 연산자(−)를 사용해서 양수로 반환한다.

```
int abs(int x) {
    if(x >= 0) {
        return x;
    } else {
        return -x;
    }
}
```

```
int abs(int x) {
    return x >= 0 ? x : -x;
}
```

매개변수의 유효성 검사

메서드의 구현부 {}를 작성할 때, 제일 먼저 해야 하는 일이 매개변수의 값이 적절한 것인지 확인하는 것이다. 메서드를 작성하는 사람은 ‘호출하는 쪽에서 알아서 값을 넘겨주겠지.’라고 생각하면 안된다. 타입만 맞으면 어떤 값도 매개변수를 통해 넘어올 수 있기 때문에, 가능한 모든 경우의 수에 대해 고민하고 그에 대비한 코드를 작성해야 한다.

아래에 정의된 메서드 divide는 매개변수 x를 y로 나눈 결과를 실수(float타입)로 반환하는데, 0으로 나누는 것은 금지되어 있기 때문에 계산 전에 y의 값이 0인지 확인해야 한다.

만일 y의 값이 0이면, 나누기를 할 수 없으므로 return문으로 작업을 중단하고 메서드를 빠져나와야 한다. 그렇지 않으면, 나누기를 하는 문장에서 에러가 발생하여 프로그램이 비정상적으로 종료된다.

```
float divide(int x, int y) {
    // 작업을 하기 전에 나누는 수(y)가 0인지 확인한다.
    if(y == 0) {
        System.out.println("0으로 나눌 수 없습니다.");
        return 0; // 매개변수가 유효하지 않으므로 메서드를 종료한다.
    }

    return x / (float)y;
}
```

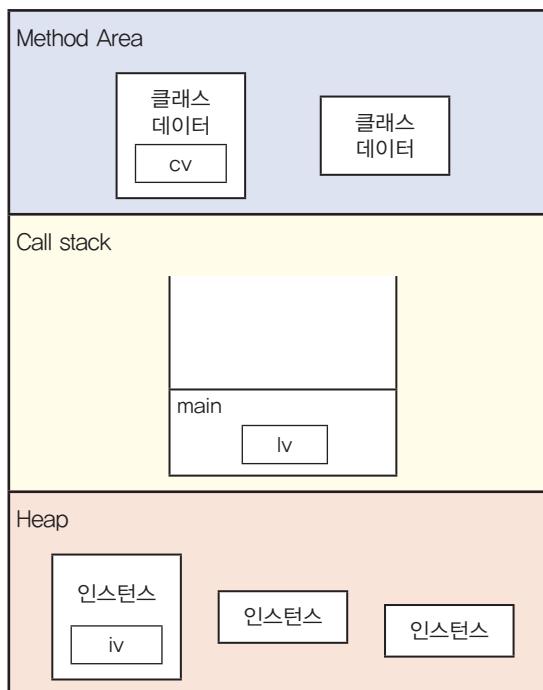
적절하지 않은 값이 매개변수를 통해 넘어온다면 매개변수의 값을 보정하던가, 보정하는 것이 불가능하다면 return문을 사용해서 작업을 중단하고 호출한 메서드로 되돌아가야 한다.

이 책의 많은 예제에서 코드를 단순화하기 위해서 유효성 검사를 생략한 경우가 많지만 여러분들이 메서드를 작성할 때는 매개변수의 유효성 검사하는 코드를 반드시 넣길 바란다. 매개변수의 유효성 검사는 메서드의 작성에 있어서 간과하기 쉬운 중요한 부분이다.

3.7 JVM의 메모리 구조

자바 프로그램이 실행되면, JVM은 시스템으로부터 프로그램을 수행하는데 필요한 메모리를 할당받고 JVM은 이 메모리를 용도에 따라 여러 영역으로 나누어 관리한다.

그 중 3가지 주요 영역(method area, call stack, heap)에 대해서 알아보자.



▲ 그림 6-5 JVM의 메모리 구조

| 참고 | cv는 클래스 변수, iv는 인스턴스 변수, lv는 지역 변수를 뜻한다.

1. 메서드 영역(method area)

- 프로그램 실행 중 어떤 클래스가 필요하면, JVM은 해당 클래스의 클래스파일(*.class)을 읽어서 분석하여 클래스에 대한 정보(클래스 데이터)를 이곳에 저장한다. 이 때, 그 클래스의 클래스 변수(class variable)도 이 영역에 함께 생성된다.

2. 힙(heap)

- 인스턴스가 생성되는 공간. 프로그램 실행 중에 인스턴스는 모두 이곳에 생성된다.
즉, 인스턴스 변수(instance variable)가 생성되는 공간이다.

3. 호출 스택(call stack 또는 execution stack)

- 호출 스택은 메서드의 작업에 필요한 메모리 공간을 제공한다. 메서드가 호출되면, 호출스택에 호출된 메서드를 위한 메모리가 할당되며, 이 메모리는 메서드가 작업을 수행하는 동안 지역 변수(매개변수 포함)와 연산의 결과를 저장하는데 사용된다. 그리고 메서드가 종료되면 할당되었던 메모리 공간은 자동으로 반환된다.

메서드가 호출되면 호출 스택에 메서드를 위한 작업 공간이 마련된다. 이 작업 공간은 메서드마다 서로 구별되며 지역 변수가 저장된다.

처음 호출된 메서드를 위한 작업 공간이 호출스택의 맨 밑에 마련되고, 메서드가 수행 중에 다른 메서드를 호출하면, 자신의 바로 위에 호출된 메서드를 위한 공간이 마련된다.

이 때 호출한 메서드(caller)는 수행을 멈추고, 호출된 메서드(callee)가 수행되기 시작한다. 호출된 메서드가 수행을 마치면, 이 메서드가 사용하던 메모리공간이 호출 스택에서 제거되며, 멈추었던 메서드(caller)는 다시 수행을 계속하게 된다.

호출스택의 제일 상위에 위치하는 메서드가 현재 실행 중인 메서드이며, 나머지는 대기 상태에 있게 된다.

따라서, 호출스택을 조사해 보면 메서드 간의 호출관계와 현재 수행중인 메서드가 어느 것인지 알 수 있다. 호출스택의 특징을 정리해보면 다음과 같다.

- 메서드가 호출되면 수행에 필요한 메모리를 스택에 할당받는다.
- 메서드가 수행을 마치고나면 할당받은 메모리가 스택에서 자동 제거된다.
- 호출 스택의 제일 위에 있는 메서드만 현재 실행 중인 메서드이다.(나머지는 대기중)
- 아래에 있는 메서드가 바로 위의 메서드를 호출한 메서드이다.

반환타입(return type)이 있는 메서드는 종료되면서 결과값을 자신을 호출한 메서드(caller)에게 반환한다. 대기상태에 있던 호출한 메서드(caller)는 넘겨받은 반환값으로 수행을 계속 진행하게 된다.

▼ 예제 6-7/CallStackEx.java

```
class CallStackEx {
    public static void main(String[] args) {
        firstMethod(); // static메서드는 객체 생성없이 호출 가능
    }

    static void firstMethod() {
        secondMethod();
    }

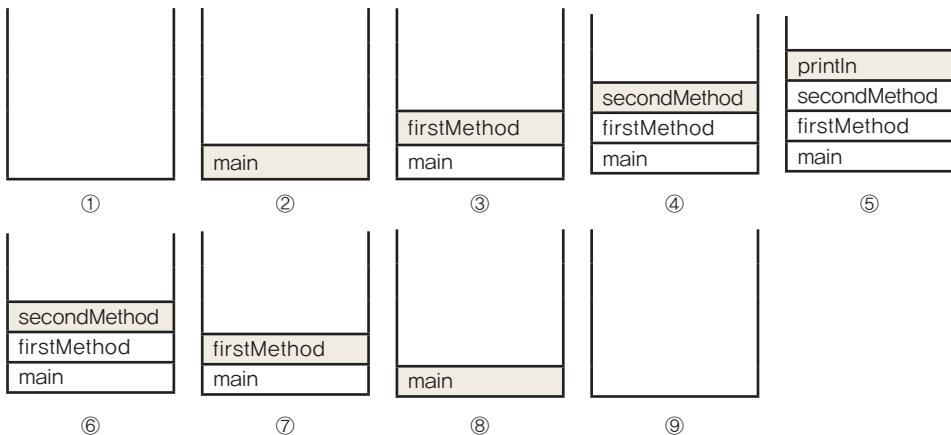
    static void secondMethod() {
        System.out.println("secondMethod()");
    }
}
```

▼ 실행결과

secondMethod()

main()이 firstMethod()를 호출하고 firstMethod()는 secondMethod()를 호출한다. 객체를 생성하지 않고도 메서드를 호출할 수 있으려면, 메서드 앞에 ‘static’을 붙여야 한다. 잠시 후에 자세히 설명할 것이므로 지금은 이정도만 알아두자.

위의 예제를 실행시켰을 때, 프로그램이 수행되는 동안 호출스택의 변화를 그림과 함께 살펴보도록 하자.



▲ 그림 6-6 예제6-7의 실행 시 호출스택의 변화

①~② 위의 예제를 실행시키면, JVM에 의해 main메서드가 호출되면서 프로그램이 시작. 이 때, 호출 스택에는 main메서드를 위한 메모리 공간이 할당되고 main메서드의 코드가 수행되기 시작.

③ main메서드가 firstMethod()를 호출. 아직 main메서드가 끝난 것은 아니므로 main메서드는 대기 중 firstMethod()가 끝나야 main메서드의 나머지 문장들을 수행할 수 있기 때문.

④ firstMethod()가 secondMethod()를 호출. firstMethod()는 secondMethod()가 끝날 때까지 대기 중.

⑤ secondMethod()에서 println()을 호출했다. println()에 의해 'secondMethod()'가 화면에 출력된다.

⑥ println()의 수행이 완료되어 호출 스택에서 사라지고 자신을 호출한 secondMethod()로 되돌아간다. 대기 중이던 secondMethod()는 println()을 호출한 이후부터 수행을 재개한다.

⑦ secondMethod()에 더 이상 수행할 코드가 없으므로 종료. 자신을 호출한 firstMethod()로 돌아간다.

⑧ firstMethod()도 더 이상 수행할 코드가 없으므로 종료. 자신을 호출한 main메서드로 돌아간다.

⑨ main메서드도 더 이상 수행할 코드가 없으므로 종료. 호출 스택은 완전히 비워지고 프로그램은 종료.

▼ 예제 6-8/CallStackEx2.java

```
class CallStackEx2 {
    public static void main(String[] args) {
        System.out.println("main(String[] args)이 시작되었음.");
        firstMethod();
        System.out.println("main(String[] args)이 끝났음.");
    }

    static void firstMethod() {
        System.out.println("firstMethod()이 시작되었음.");
        secondMethod();
        System.out.println("firstMethod()이 끝났음.");
    }

    static void secondMethod() {
        System.out.println("secondMethod()이 시작되었음.");
        System.out.println("secondMethod()이 끝났음.");
    }
}
```

▼ 실행결과

```
main(String[] args)이 시작되었음.
firstMethod()이 시작되었음.
secondMethod()이 시작되었음.
secondMethod()이 끝났음.
firstMethod()이 끝났음.
main(String[] args)이 끝났음.
```

예제6-7에 출력문을 추가해서 각 메서드의 시작과 종료의 순서를 확인하는 예제이다. 그림6-6과 함께 다시 한 번 호출과정을 확인해보자.

3.8 기본형 매개변수와 참조형 매개변수

자바에서는 메서드를 호출할 때 매개변수로 지정한 값을 메서드의 매개변수에 복사해서 넘겨준다. 매개변수의 타입이 기본형(primitive type)일 때는 기본형 값이 복사되겠지만, 참조형(reference type)이면 인스턴스의 주소가 복사된다.

메서드의 매개변수를 기본형으로 선언하면 단순히 저장된 값만 얻지만, 참조형으로 선언하면 값이 저장된 곳의 주소를 알 수 있기 때문에 값을 읽어 오는 것은 물론 값을 변경하는 것도 가능하다.

기본형 매개변수 변수의 값을 읽기만 할 수 있다.(read only)

참조형 매개변수 변수의 값을 읽고 변경할 수 있다.(read & write)

▼ 예제 6-9/PrimitiveParamEx.java

```
class Data { int x; }

class PrimitiveParamEx {
    public static void main(String[] args) {
        Data d = new Data();
        d.x = 10;
        System.out.println("main() : x = " + d.x);

        change(d.x);
        System.out.println("After change(d.x)");
        System.out.println("main() : x = " + d.x);
    }

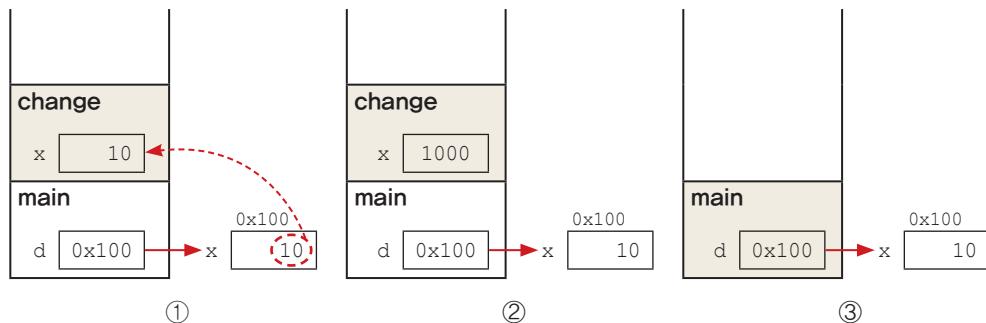
    static void change(int x) { // 기본형 매개변수
        x = 1000;
        System.out.println("change() : x = " + x);
    }
}
```

▼ 실행결과

```
main() : x = 10
change() : x = 1000
After change(d.x)
main() : x = 10
```

| 플레이동영상 | /flash/PrimitiveParam.exe를 보면 예제6-9의 실행과정을 자세히 볼 수 있다.

change메서드에서 main메서드로부터 넘겨받은 d.x의 값을 1000으로 변경했는데도 main메서드에서는 d.x의 값이 그대로이다. 왜 이런 결과가 나오는지 아래의 그림으로 확인해 보자.



- ① change메서드가 호출되면서 'd.x'가 change메서드의 매개변수 x에 복사됨
- ② change메서드에서 x의 값을 1000으로 변경
- ③ change메서드가 종료되면서 매개변수 x는 스택에서 제거됨

'd.x'의 값이 변경된 것이 아니라, change메서드의 매개변수 x의 값이 변경된 것이다. 즉, 원본이 아닌 복사본이 변경된 것이라 원본에는 아무런 영향을 미치지 못한다. 이처럼 기본형 매개변수는 변수의 값을 읽을 수 있지만 변경할 수는 없다.

▼ 예제 6-10/ReferenceParamEx.java

```
class Data2 { int x; }

class ReferenceParamEx {
    public static void main(String[] args) {
        Data2 d = new Data2();
        d.x = 10;
        System.out.println("main() : x = " + d.x);

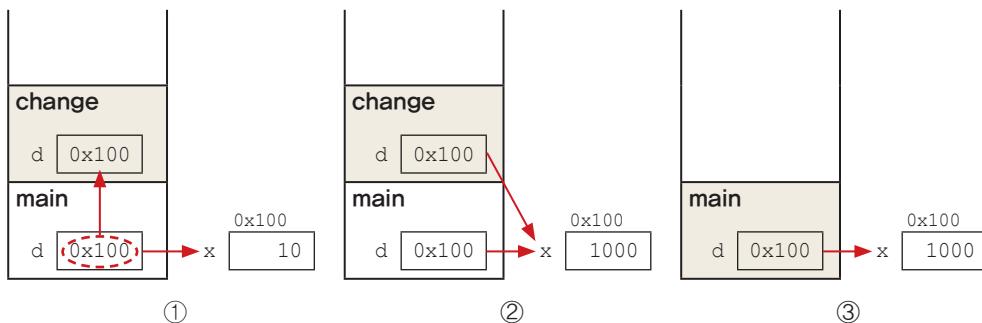
        change(d);
        System.out.println("After change(d)");
        System.out.println("main() : x = " + d.x);
    }

    static void change(Data2 d) { // 참조형 매개변수
        d.x = 1000;
        System.out.println("change() : x = " + d.x);
    }
}
```

▼ 실행결과
main() : x = 10 change() : x = 1000 After change(d) main() : x = 1000

| 플래시동영상 | ReferenceParam.exe를 보면 예제6-10의 실행과정을 자세히 볼 수 있다.

이전 예제와 달리 change메서드를 호출한 후에 d.x의 값이 변경되었다. change메서드의 매개변수가 참조형이므로 값이 아니라 '값이 저장된 주소'를 change메서드에게 넘겨주었기 때문에 값을 읽어오는 것뿐만 아니라 변경하는 것도 가능하다.



① change메서드가 호출되면서 참조 변수 d의 값(주소)이 매개변수 d에 복사됨.

이제 매개변수 d에 저장된 주소값으로 x에 접근이 가능

② change메서드에서 매개변수 d로 x의 값을 1000으로 변경

③ change메서드가 종료되면서 매개변수 d는 스택에서 제거됨

이전 예제와 달리, change메서드의 매개변수를 참조형으로 선언했기 때문에, x의 값이 아닌 주소가 매개변수 d에 복사되었다. 이제 main메서드의 참조변수 d와 change메서드의 참조변수 d는 같은 객체를 가리키게 된다. 그래서 매개변수 d로 x의 값을 읽는 것과 변경하는 것이 모두 가능한 것이다. 이 두 예제의 차이를 이해하는 것은 매우 중요하다.

▼ 예제 6-11/ReferenceParamEx2.java

```
class ReferenceParamEx2 {
    public static void main(String[] args)
    {
        int[] x = {10}; // 크기가 1인 배열. x[0] = 10;
        System.out.println("main() : x = " + x[0]);

        change(x);
        System.out.println("After change(x)");
        System.out.println("main() : x = " + x[0]);
    }

    static void change(int[] x) { // 참조형 매개변수
        x[0] = 1000;
        System.out.println("change() : x = " + x[0]);
    }
}
```

▼ 실행결과

```
main() : x = 10
change() : x = 1000
After change(x)
main() : x = 1000
```

이전의 참조형 매개변수 예제를 Data2클래스의 인스턴스 대신 길이가 1인 배열 x를 사용하도록 변경한 것이다. 배열도 객체와 같이 참조변수를 통해 데이터가 저장된 공간에 접근한다는 것을 이미 배웠다. 이전 예제의 Data2클래스 타입의 참조변수 d와 같이 변수 x도 int배열 타입의 참조변수이기 때문에 같은 결과를 얻는다.

임시적으로 간단히 처리할 때는 별도의 클래스를 선언하는 것보다 이처럼 배열을 이용하는 것도 좋은 방법이다.

▼ 예제 6-12/ReferenceParamEx3.java

```

class ReferenceParamEx3 {
    public static void main(String[] args) {
        int[] arr = new int[] {3,2,1,6,5,4};

        printArr(arr); // 배열의 모든 요소를 출력
        sortArr(arr); // 배열을 정렬
        printArr(arr); // 정렬후 결과를 출력
        System.out.println("sum="+sumArr(arr)); // 배열의 총합을 출력
    }

    static void printArr(int[] arr) { // 배열의 모든 요소를 출력
        System.out.print("[");
        for(int i : arr) // 향상된 for문
            System.out.print(i+",");
        System.out.println("]");
    }

    static int sumArr(int[] arr) { // 배열의 모든 요소의 합을 반환
        int sum = 0;
        for(int i=0;i<arr.length;i++)
            sum += arr[i];
        return sum;
    }

    static void sortArr(int[] arr) { // 배열을 오름차순으로 정렬
        for(int i=0;i<arr.length-1;i++)
            for(int j=0;j<arr.length-1-i;j++) {
                if(arr[j] > arr[j+1]) {
                    int tmp = arr[j];
                    arr[j] = arr[j+1];
                    arr[j+1] = tmp;
                }
            } // sortArr(int[] arr)
    }
}

```

▼ 실행결과
[3,2,1,6,5,4]
[1,2,3,4,5,6]
sum=21

메서드로 배열을 다루는 여러 가지 방법을 보여주는 예제이다. 매개변수의 타입이 배열이니까, 참조형 매개변수이다. 그래서 sortArr메서드에서 정렬한 것이 원래의 배열에 영향을 미친다. 그 외에는 따로 설명하지 않아도 충분히 이해가 될 것이다.

▼ 예제 6-13/ReturnEx.java

```

class ReturnEx {
    public static void main(String[] args) {
        ReturnEx r = new ReturnEx();

        int result = r.add(3,5);
        System.out.println(result);

        int[] result2 = {0}; // 배열을 생성하고 result2[0]의 값을 0으로 초기화
        r.add(3,5,result2); // 배열을 add메서드의 매개변수로 전달
        System.out.println(result2[0]);
    }
}

```

```

int add(int a, int b) {
    return a + b;
}

void add(int a, int b, int[] result) {
    result[0] = a + b; // 매개변수로 넘겨받은 배열에 연산결과를 저장
}

```

▼ 실행결과
8
8

이 예제는 반환값이 있는 메서드를 반환값이 없는 메서드로 바꾸는 방법을 보여준다. 앞서 배운 참조형 매개변수를 활용하면 반환값이 없어도 메서드의 실행결과를 얻어 올 수 있다.

```

int add(int a, int b) {
    return a + b;
} ←→ void add(int a, int b, int[] result) {
        result[0] = a + b;
    }

```

메서드는 단 하나의 값만을 반환할 수 있지만 이것을 응용하면 여러 개의 값을 반환받을 수 있다.

3.9 참조형 반환타입

매개변수뿐만 아니라 반환타입도 참조형이 가능하다. 반환타입이 참조형이라는 것은 반환하는 값의 타입이 참조형이라는 얘긴데, 모든 참조형 타입의 값은 ‘객체의 주소’이므로 그저 정수값이 반환되는 것일 뿐 특별할 것이 없다. 일단 예제부터 먼저 살펴보자.

▼ 예제 6-14/ReferenceReturnEx.java

```

class Data3 { int x; }

class ReferenceReturnEx {
    public static void main(String[] args) {
        Data3 d = new Data3();
        d.x = 10;

        Data3 d2 = copy(d);
        System.out.println("d.x =" + d.x);
        System.out.println("d2.x=" + d2.x);
    }

    static Data3 copy(Data3 d) { // 참조형 반환타입. 객체의 주소(정수)를 반환
        Data3 tmp = new Data3();
        tmp.x = d.x;

        return tmp;
    }
}

```

▼ 실행결과
d.x =10
d2.x=10

copy메서드는 새로운 객체를 생성한 후에, 매개변수로 넘겨받은 객체에 저장된 값을 복사해서 반환한다. 반환하려는 값이 Data3객체의 주소이므로 반환 타입이 ‘Data3’이다.

[ebook – 샘플. 무료 공유] 자바의 정석 4판 Java 21 올컬러 2025. 7. 7 출시 seong.namkung@gmail.com

```

static Data3 copy(Data d) {
    Data3 tmp = new Data3(); // 새로운 객체 tmp를 생성
    tmp.x = d.x;           // d.x의 값을 tmp.x에 복사

    return tmp; // 복사한 객체의 주소를 반환
}

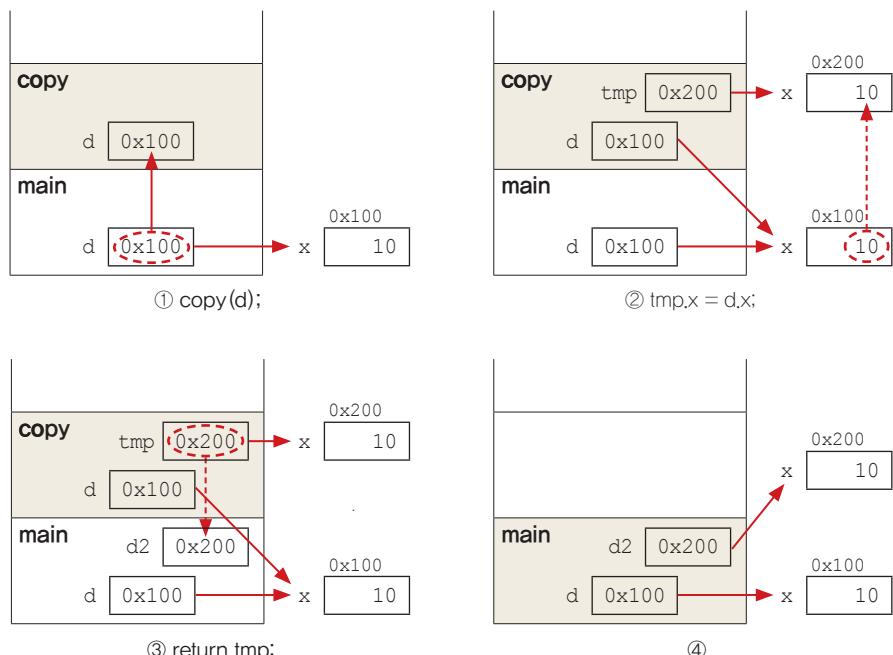
```

이 메서드의 반환타입이 'Data3'이므로, 호출결과를 저장하는 변수의 타입 역시 'Data3'타입의 참조변수이어야 한다.

```
Data3 d2 = copy(d); // static Data3 copy(Data d)
```

copy메서드 내에서 생성한 객체를 main메서드에서 사용할 수 있으려면, 이렇게 새로운 객체의 주소를 반환해야 한다. 그렇지 않으면, copy메서드가 종료되면서 새로운 객체의 참조가 사라지기 때문에 더 이상 이 객체를 사용할 방법이 없다.

copy메서드가 호출된 직후부터 종료까지의 과정을 단계별로 살펴보면 다음과 같다.



- ① copy메서드를 호출하면서 참조변수 d의 값이 매개변수 d에 복사된다.
- ② 새로운 객체를 생성한 다음, d.x에 저장된 값을 tmp.x에 복사한다.
- ③ copy메서드가 종료되면서 반환한 tmp의 값은 참조변수 d2에 저장된다.
- ④ copy메서드가 종료되어 tmp가 사라졌지만, d2로 새로운 객체를 다룰 수 있다.

반환타입이 '참조형'이라는 것은

메서드가 '객체의 주소'를 반환한다는 것을 의미한다.

[ebook – 샘플. 무료 공유] 자바의 정석 4판 Java 21 올컬러 2025. 7. 7 출시 seong.namkung@gmail.com

3.10 재귀 호출(recursive call)

메서드의 내부에서 메서드 자신을 다시 호출하는 것을 ‘재귀 호출(recursive call)’이라 하고, 재귀 호출을 하는 메서드를 ‘재귀 메서드’라 한다.

```
void method() {
    method(); // 재귀 호출. 메서드 자신을 호출한다.
}
```

어떻게 메서드가 자기자신을 호출할 수 있는지 의아하겠지만, 메서드 입장에서는 자기 자신을 호출하는 것과 다른 메서드를 호출하는 것의 차이가 없다. ‘메서드 호출’이라는 것이 그저 특정 위치에 저장되어 있는 명령들을 수행하는 것일 뿐이기 때문이다.

호출된 메서드는 ‘값’에 의한 호출(call by value)을 통해, 원래의 값이 아닌 복사된 값으로 작업하기 때문에 호출한 메서드와 관계없이 독립적인 작업수행이 가능하다.

그런데 위의 코드처럼 오로지 재귀 호출뿐이면, 무한히 자기 자신을 호출하기 때문에 무한반복에 빠지게 된다. 무한반복문이 조건문과 함께 사용되어야하는 것처럼, 재귀 호출도 조건문이 필연적으로 따라다닌다.

```
void method(int n) {
    if(n == 0)
        return; // n의 값이 0일 때, 메서드를 종료한다.
    System.out.println(n);

    method(--n); // 재귀 호출. method(int n)을 호출
}
```

이 코드는 매개변수 n을 1씩 감소시켜가면서 재귀 호출을 하다가 n의 값이 0이 되면 재귀 호출을 중단하게 된다. 재귀호출은 반복문과 유사한 점이 많으며, 대부분의 재귀 호출은 반복문으로 작성하는 것이 가능하다. 위의 코드를 반복문으로 작성하면 다음의 오른쪽 코드와 같다.

```
void method(int n) {
    if(n==0) return;
    System.out.println(n);
    method(--n); // 재귀 호출
}
```



```
void method(int n) {
    while(n!=0) {
        System.out.println(n--);
    }
}
```

반복문은 그저 같은 문장을 반복해서 수행하는 것이지만, 메서드를 호출하는 것은 반복문 보다 몇 가지 과정, 예를 들면 매개변수 복사와 종료 후 복귀할 주소저장 등, 이 추가로 필요하기 때문에 반복문보다 재귀 호출의 수행시간이 더 오래 걸린다.

[ebook – 샘플. 무료 공유] 자바의 정석 4판 Java 21 올컬러 2025. 7. 7 출시 seong.namkung@gmail.com

그렇다면 ‘왜? 굳이 반복문 대신 재귀 호출을 사용할까’. 그 이유는 바로 재귀 호출이 주는 간결함 때문이다. 몇 겹의 반복문과 조건문으로 복잡하게 작성된 코드가 재귀 호출로 작성하면 보다 단순한 구조로 바뀔 수도 있다. 아무리 효율적이라도 알아보기 힘들게 작성하는 것보다 다소 비효율적이어도 알아보기 쉽게 작성하는 것이 논리적 오류가 발생할 확률도 줄어들고 나중에 수정하기도 좋다.

어떤 작업을 반복적으로 처리해야 한다면, 먼저 반복문으로 작성해보고 너무 복잡하면 재귀 호출로 간단히 할 수 없는지 고민해볼 필요가 있다. 재귀 호출은 비효율적이므로 재귀 호출에 드는 비용보다 재귀 호출의 간결함이 주는 이득이 충분히 큰 경우에만 사용해야 한다는 것도 잊지 말자.

대표적인 재귀 호출의 예는 팩토리얼(factorial)을 구하는 것이다. 팩토리얼은 한 숫자가 1이 될 때까지 1씩 감소시켜가면서 계속해서 곱해 나가는 것인데, $n!(n\text{은 양의 정수})$ 과 같이 표현한다. 예를 들면, ' $5! = 5 * 4 * 3 * 2 * 1 = 120$ '이다.

팩토리얼을 수학적으로 표현하면 다음과 같다.

$$f(n) = n * f(n-1), \text{ 단 } f(1) = 1$$

다음 예제는 위의 함수를 자바로 구현한 것이다.

▼ 예제 6-15/FactorialEx.java

```
class FactorialEx {
    public static void main(String args[]) {
        int result = factorial(4);
        System.out.println(result);
    }

    static int factorial(int n) {
        int result = 0;
        if (n == 1)
            result = 1;
        else
            result = n * factorial(n-1); // 다시 메서드 자신을 호출
        return result;
    }
}
```

▼ 실행결과
24

| 플래시동영상 | /flash/RecursiveCall.exe를 보면 예제6-15의 실행과정을 자세히 볼 수 있다.

위 예제는 팩토리얼을 계산하는 메서드를 구현하고 테스트하는 것이다. `factorial()`이 static메서드이므로 인스턴스를 생성하지 않고 직접 호출할 수 있다. 그리고 main메서드와 같은 클래스에 있기 때문에 static메서드를 호출할 때 클래스이름을 생략하는 것이 가능하다. 그래서 'FactorialEx.factorial(4)' 대신 'factorial(4)'와 같이 하였다.

예제6-15는 실행과정을 플래시 동영상으로 보여주기 위해 작성한 것이라 코드가 길어졌는데, 좀 더 간단히 하면 다음과 같이 쓸 수 있다.

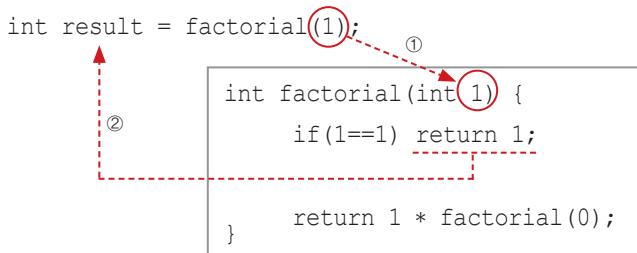
[ebook – 샘플. 무료 공유] 자바의 정석 4판 Java 21 올컬러 2025. 7. 7 출시 seong.namkung@gmail.com

```
static int factorial(int n) {
    if(n == 1) return 1;
    return n * factorial(n-1);
}
```

이해하기 어려운 코드는 변수에 직접 값을 대입해보면 알기 쉬워진다. 만일 매개변수 n의 값이 3이라면, n대신 3을 직접 대입해보자. 오른쪽의 코드처럼 된다.



같은 방법으로 main메서드에서 factorial(1)을 호출했을 때의 실행과정을 살펴보자. 아래의 그림은 매개변수 n 대신 1을 대입한 것이다.



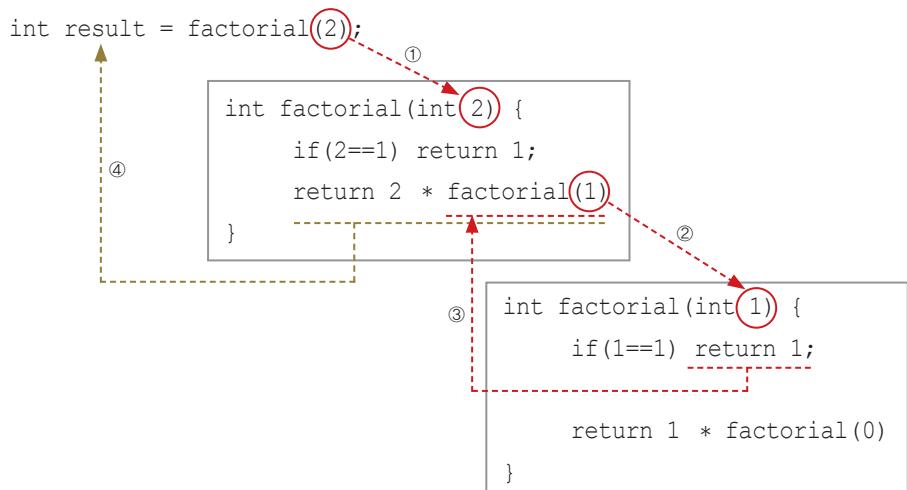
if문의 조건식이 참이 되어 1을 반환하기 때문에, 재귀 호출이 일어나기도 전에 factorial 메서드는 종료된다. 그리고 result에는 ‘factorial(1)’의 반환값인 1이 저장된다. 위의 그림에서는 반환값이 바로 result에 저장되는 것처럼 되어 있지만, 정확히는 아래와 같이 반환 값이 메서드의 호출을 대신한다.

```
int result = factorial(1);
→ int result = 1;
```

이번엔 ‘factorial(2)’를 호출했을 때의 실행과정도 살펴보자. 매개변수의 값이 1이 아니므로 조건식이 거짓이 되어 그 다음 문장인 ‘return 2 * factorial(1);’이 수행되고, 이 식을 계산하는 과정에서 다시 factorial(1)이 호출된다.

- ① factorial(2)를 호출하면서 매개변수 n에 2가 복사된다.
- ② ‘return 2 * factorial(1);’을 계산하려면 factorial(1)을 호출한 결과가 필요하다.
그래서 factorial(1)이 호출되고 매개변수 n에 1이 복사된다.
- ③ if문의 조건식이 참이므로 1을 반환하면서 메서드는 종료된다. 그리고 factorial(1)을 호출한 곳으로 되돌아간다.
- ④ 이제 factorial(1)의 결과값인 1을 얻었으므로, return문이 다음의 과정으로 계산된다.

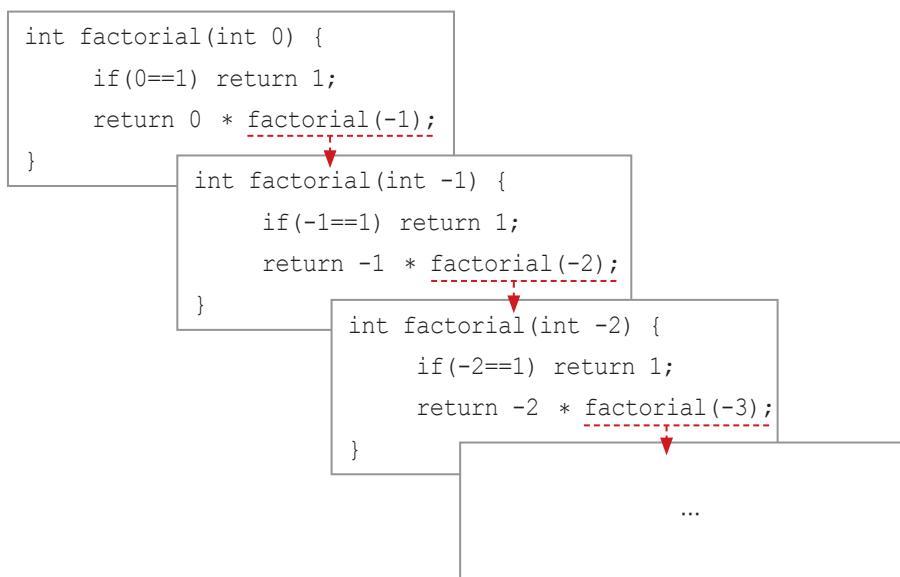
```
return 2 * factorial(1);
→ return 2 * 1;
→ return 2;
```



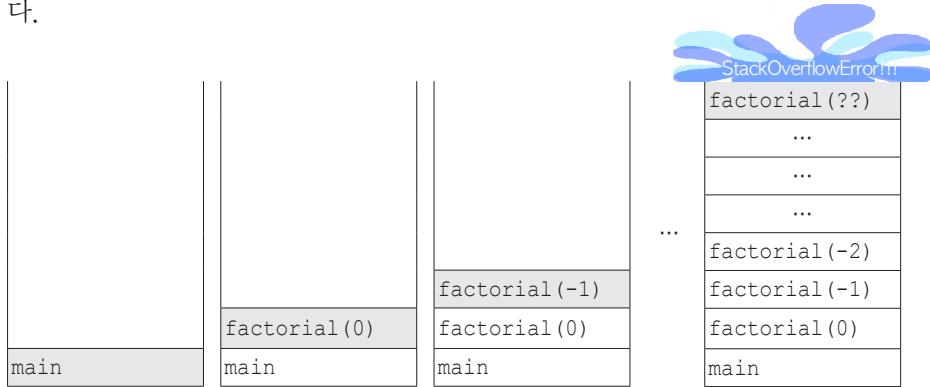
`factorial(2)`는 종료되면서 결과값 2를 반환하고, 이 값은 변수 `result`에 저장된다.

```
int result = factorial(2);  
→ int result = 2;
```

이제 매개변수 n의 값이 커도 어떤 과정으로 계산되는지 이해할 수 있으리라 생각한다. 그런데 만일 factorial()의 매개변수 n의 값이 0이면 어떻게 될까? 또는 100,000과 같은 큰 수이면 어떻게 될까? 예제를 변경해서 실행해보면 쉽게 알 수 있겠지만 그 전에 잠시 시간을 갖고 어떻게 될 것인지 예측해보자. 먼저 매개변수 n의 값이 0인 경우는 if문의 조건식이 절대 참이 될 수 없기 때문에 계속해서 재귀호출만 일어날 뿐 메서드가 종료되지 않으므로 스택에 계속 데이터가 쌓여만 간다.



어느 시점에 이르러서는 결국 스택의 저장 한계를 넘게 되고, ‘스택오버플로우 에러(Stack OverflowError)’가 발생한다. 매개변수 n의 값이 100,000과 같이 큰 경우에도 마찬가지다.



이처럼 우리가 메서드를 작성할 때, ‘호출하는 사람이 당연히 알아서 적절한 값을 인자로 주겠지.’라는 막연한 믿음을 가지면 안 되고, 어떤 값이 들어와도 에러없이 처리되는 견고한 코드를 작성해야 한다. 그래서 ‘매개변수의 유효성 검사’가 중요한 것이다.

```
static int factorial(int n) {
    if(n <= 0 || n > 12) return -1; // 매개변수 n의 유효성 검사를 추가
    if(n == 1) return 1;

    return n * factorial(n-1);
}
```

매개변수 n의 상한값을 12로 정한 이유는 $13!$ 의 값이 factorial()의 반환타입인 int의 최대값(약20억)보다 크기 때문이다. 만일 그 이상의 값을 구하고 싶으면 반환타입을 int보다 큰 long이나 BigInteger로 변경하면 된다.

| 참고 | BigInteger는 배열을 이용해서 무한히 큰 정수를 다룰 수 있게 해주는 클래스로 p.548에서 자세히 배운다.

참고로 재귀 메서드 factorial을 반복문으로 작성하면 다음과 같다.

```
int factorial(int n) {
    if(n==1) return 1;
    return n * factorial(n-1);
} → int factorial(int n) {
    int result = 1;
    while(n!=0)
        result *= n--;
    return result;
}
```

while문으로 작성한 오른쪽의 코드는 왼쪽의 재귀 호출과 달리 많은 수의 반복에도 ‘스택 오버플로우 에러’와 같은 메모리 부족 문제를 겪지 않을 뿐만 아니라 속도도 빠르다.

[ebook – 샘플. 무료 공유] 자바의 정석 4판 Java 21 올컬러 2025. 7. 7 출시 seong.namkung@gmail.com

▼ 예제 6-16/FactorialEx2.java

```

class FactorialEx2 {
    static long factorial(int n) {
        if(n<=0 || n>20) return -1; // 매개변수의 유효성 검사.

        if(n<=1) return 1;
        return n * factorial(n-1);
    }

    public static void main(String args[]) {
        int n = 21;
        long result = 0;

        for(int i = 1; i <= n; i++) {
            result = factorial(i);

            if(result===-1) {
                System.out.printf("유효하지 않은 값입니다. (0<n<=20) :%d%n", n);
                break;
            }

            System.out.printf("%2d!=%20d%n", i, result);
        }
    } // main의 끝
}

```

▼ 실행결과

```

1!=          1
2!=          2
3!=          6
4!=         24
5!=        120
6!=       720
7!=      5040
8!=     40320
9!=    362880
10!=   3628800
11!=  39916800
12!= 479001600
13!= 6227020800
14!= 87178291200
15!= 1307674368000
16!= 20922789888000
17!= 355687428096000
18!= 6402373705728000
19!= 121645100408832000
20!= 2432902008176640000
유효하지 않은 값입니다. (0<n<=20) :21

```

이전 예제 6-15에 매개변수의 유효성을 검사하는 코드를 추가해서 factorial()의 매개변수 n이 음수이거나 20보다 크면 -1을 반환하도록 하였다.

[ebook – 샘플. 무료 공유] 자바의 정석 4판 Java 21 올컬러 2025. 7. 7 출시 seong.namkung@gmail.com

▼ 예제 6-17/MainEx.java

```
class MainEx {
    public static void main(String args[]) {
        main(null); // 재귀 호출. 자기 자신을 다시 호출
    }
}
```

▼ 실행결과

```
java.lang.StackOverflowError
at MainEx.main(MainEx.java:3)
at MainEx.main(MainEx.java:3)
...
at MainEx.main(MainEx.java:3)
at MainEx.main(MainEx.java:3)
```

main메서드 역시 자기 자신을 호출하는 것이 가능하며 아무런 조건도 없이 계속해서 자기 자신을 다시 호출하기 때문에 무한 호출에 빠지게 된다.

main메서드가 종료되지 않고 호출 스택에 계속해서 쌓이게 되므로 결국 호출 스택의 메모리 한계를 넘게 되고 StackOverflowError가 발생하여 프로그램은 비정상적으로 종료된다.

▼ 예제 6-18/PowerEx.java

```
class PowerEx {
    public static void main(String[] args) {
        int x = 2;
        int n = 5;
        long result = 0;

        for(int i=1; i<=n; i++) {
            result += power(x, i);
        }

        System.out.println(result);
    }

    static long power(int x, int n) {
        if(n==1) return x;
        return x * power(x, n-1);
    }
}
```

▼ 실행결과

62

x^1 부터 x^n 까지의 합을 구하는 예제이다. 재귀 호출로 x^n 을 구하는 power()를 작성하였다. x는 2, n은 5로 계산했기 때문에 $2^1+2^2+2^3+2^4+2^5$ 의 결과인 62가 출력되었다.

x의 n제곱을 계산하는 메서드는 다음과 같이 정의할 수 있는데, 이 메서드 역시 메서드의 정의에 자신을 포함하는 재귀 메서드이다.

$$f(x, n) = x * f(x, n-1), \text{ 단 } f(x, 1) = x$$

[ebook – 샘플. 무료 공유] 자바의 정석 4판 Java 21 올컬러 2025. 7. 7 출시 seong.namkung@gmail.com

예를 들어 2의 4제곱을 구해보자. x는 2이고, n은 4이므로 각각을 메서드에 대입하면 다음과 같이 된다.

$$f(2, 4) = 2 * f(2, 3)$$

$f(2, 3)$ 은 ‘ $2 * f(2, 2)$ ’이므로, 위의 식에 $f(2, 3)$ 대신 ‘ $2 * f(2, 2)$ ’를 대입하면 다음과 같다.

$$\begin{aligned} f(2, 4) &= 2 * \textcolor{red}{f(2, 3)} \\ f(2, 4) &= 2 * 2 * \textcolor{red}{f(2, 2)} \end{aligned}$$

같은 방식으로 반복해서 계산하면 다음과 같다. $f(2, 1)$ 은 2라는 것에 주의하자.

$$\begin{aligned} f(2, 4) &= 2 * \textcolor{red}{f(2, 3)} \\ \rightarrow f(2, 4) &= 2 * 2 * \textcolor{red}{f(2, 2)} \\ \rightarrow f(2, 4) &= 2 * 2 * 2 * \textcolor{red}{f(2, 1)} \\ \rightarrow f(2, 4) &= 2 * 2 * 2 * 2 \end{aligned}$$

이 메서드도 재귀 호출이 아닌 반복문으로 처리하는 것이 가능하다. 재귀 호출의 예를 보여주기 위해 재귀 메서드로 작성한 것 뿐이다.

3.11 클래스 메서드와 인스턴스 메서드

변수에서 그랬던 것과 같이, 메서드 앞에 static이 붙어 있으면 클래스 메서드이고 붙어 있지 않으면 인스턴스 메서드이다.

클래스 메서드도 클래스 변수처럼, 객체를 생성하지 않고도 ‘클래스이름.메서드이름(매개변수)’와 같은 식으로 호출이 가능하다. 반면에 인스턴스 메서드는 반드시 객체를 생성해야만 호출할 수 있다. 그렇다면 클래스를 정의할 때, 어느 경우에 static을 사용해서 클래스 메서드로 정의해야하는 것일까?

클래스는 ‘데이터(변수)와 데이터에 관련된 메서드의 집합’이므로, 같은 클래스 내에 있는 메서드와 멤버변수는 아주 밀접한 관계가 있다.

인스턴스 메서드는 인스턴스 변수와 관련된 작업을 하는, 즉 메서드의 작업을 수행하는데 인스턴스 변수를 필요로 하는 메서드이다. 그런데 인스턴스 변수는 인스턴스(객체)를 생성해야만 만들어지므로 인스턴스 메서드 역시 인스턴스를 생성해야 호출할 수 있다.

반면에 메서드 중에서 인스턴스와 관계없는(인스턴스 변수나 인스턴스 메서드를 사용하지 않는) 메서드를 클래스 메서드(static메서드)로 정의한다.

물론 인스턴스 변수를 사용하지 않는다고 해서 반드시 클래스 메서드로 정의해야하는 것은 아니지만 특별한 이유가 없는 한 클래스 메서드로 정의한다.

인스턴스 메서드 – 인스턴스(iv의 집합)가 필요한 메서드

클래스 메서드 – 인스턴스(iv의 집합)가 필요한 메서드

[ebook – 샘플. 무료 공유] 자바의 정석 4판 Java 21 올컬러 2025. 7. 7 출시 seong.namkung@gmail.com

1. 멤버변수 중 모든 인스턴스에 공통으로 사용하는 것에 static을 붙인다.

- 인스턴스는 서로 독립적이기 때문에 각 인스턴스의 변수(iv)는 서로 다른 값을 유지한다. 그러나 모든 인스턴스에서 같은 값이 유지되어야 하는 변수는 static을 붙여서 클래스 변수로 정의한다.

2. 클래스 변수(static변수)는 인스턴스를 생성하지 않아도 사용할 수 있다.

- static이 붙은 변수(클래스 변수, cv)는 클래스가 메모리에 올라갈 때 자동으로 생성되기 때문이다.

3. 클래스 메서드(static메서드)는 인스턴스 변수를 사용할 수 없다.

- 인스턴스 변수는 인스턴스가 반드시 존재해야만 사용할 수 있는데, 클래스 메서드(static이 붙은 메서드)는 인스턴스 생성 없이 호출가능하므로 클래스 메서드가 호출되었을 때 인스턴스가 존재하지 않을 수도 있다. 그래서 클래스 메서드에서 인스턴스변수의 사용을 금지한다.
반면에 인스턴스 변수나 인스턴스 메서드는 static변수나 static메서드를 사용하는 것이 항상 가능하다. 인스턴스 변수가 존재한다는 것은 static변수가 이미 메모리에 존재한다는 것을 의미하기 때문이다.

4. 메서드 내에서 인스턴스 변수를 사용하지 않는다면, static을 붙이는 것을 고려한다.

- 메서드 내에서 인스턴스 변수가 필요하다면, static을 붙일 수 없다. 반대로 인스턴스 변수가 필요하지 않다면 static을 붙여서 성능을 높이자.
static을 안 붙인 메서드(인스턴스메서드)는 실행 시 호출되어야 할 메서드를 찾는 과정이 추가적으로 필요하기 때문에 시간이 더 걸린다.

- 멤버 변수 중 모든 인스턴스에 공통된 값을 가져야 하는 것에 static을 붙이자.
- 작성한 메서드 중에서 인스턴스 변수나 인스턴스 메서드를 사용하지 않는 메서드에 static을 붙일 것을 고려한다.

| 참고 | random()과 같은 Math클래스의 메서드는 모두 static이다. Math클래스에는 인스턴스 변수가 하나도 없다.

▼ 예제 6-19/MyMathEx2.java

```

class MyMath2 {
    long a, b; // 인스턴스 변수

    // 인스턴스 변수 a, b만을 이용해서 작업하므로 매개변수가 필요없다.
    long add() { return a + b; } // a, b는 인스턴스 변수
    long subtract() { return a - b; }
    long multiply() { return a * b; }
    double divide() { return a / b; }

    // 인스턴스 변수와 관계없이 매개변수만으로 작업이 가능하다.
    static long add(long a, long b) { return a + b; } // a, b는 지역 변수
    static long subtract(long a, long b) { return a - b; }
    static long multiply(long a, long b) { return a * b; }
    static double divide(double a, double b) { return a / b; }
}

class MyMathEx2 {
    public static void main(String args[]) {
        // 클래스 메서드 호출. 인스턴스 생성없이 호출 가능
        System.out.println(MyMath2.add(200L, 100L));
        System.out.println(MyMath2.subtract(200L, 100L));
        System.out.println(MyMath2.multiply(200L, 100L));
        System.out.println(MyMath2.divide(200.0, 100.0));

        MyMath2 mm = new MyMath2(); // 인스턴스를 생성
        mm.a = 200L;
        mm.b = 100L;

        // 인스턴스 메서드는 객체생성 후에만 호출이 가능함.
        System.out.println(mm.add());
        System.out.println(mm.subtract());
        System.out.println(mm.multiply());
        System.out.println(mm.divide());
    }
}

```

▼ 실행결과
300
100
20000
2.0
300
100
20000
2.0

인스턴스 메서드인 add(), subtract(), multiply(), divide()는 인스턴스 변수인 a와 b만으로 작업이 가능하기 때문에, 매개변수가 필요없어서 팔호()에 매개변수를 선언하지 않았다.

반면에 add(long a, long b), subtract(long a, long b) 등은 인스턴스 변수 없이 매개 변수만으로 작업을 수행하기 때문에 static을 붙여서 클래스 메서드로 선언하였다.

그래서 MyMathEx2의 main메서드에서 보면, 클래스 메서드는 객체생성없이 바로 호출이 가능했고, 인스턴스 메서드는 MyMath2클래스의 인스턴스를 생성한 후에야 호출이 가능했다.

이 예제를 통해서 어떤 경우 인스턴스 메서드로, 또는 클래스 메서드로 선언해야 하는지, 그리고 그 차이를 이해하는 것은 매우 중요하다.

3.12 클래스 멤버와 인스턴스 멤버간의 참조와 호출

같은 클래스의 멤버들은 별도의 인스턴스를 생성하지 않고도 서로 참조 또는 호출이 가능하다. 단, 클래스 멤버가 인스턴스 멤버를 참조 또는 호출하고자 하는 경우에는 인스턴스를 생성해야 한다.

그 이유는 인스턴스 멤버가 존재하는 시점에 클래스 멤버는 항상 존재하지만, 클래스 멤버가 존재하는 시점에 인스턴스 멤버가 존재하지 않을 수도 있기 때문이다.

| 참고 | 인스턴스 멤버란 인스턴스 변수와 인스턴스 메서드를 의미한다.

```
class TestClass {
    void instanceMethod() {}           // 인스턴스 메서드
    static void staticMethod() {}      // static메서드

    void instanceMethod2() {           // 인스턴스 메서드
        instanceMethod();             // 다른 인스턴스 메서드를 호출한다.
        staticMethod();               // static메서드를 호출한다.
    }

    static void staticMethod2() { // static메서드
        instanceMethod();          // 에러. 인스턴스 메서드를 호출할 수 없다.
        staticMethod();              // static메서드는 호출 할 수 있다.
    }
} // end of class
```

위의 코드는 같은 클래스 내의 인스턴스 메서드와 static메서드 간의 호출에 대해서 설명하고 있다. 같은 클래스 내의 메서드는 서로 객체의 생성이나 참조변수 없이 직접 호출이 가능하지만 static메서드는 인스턴스 메서드를 호출할 수 없다.

```
class TestClass2 {
    int iv;                      // 인스턴스 변수
    static int cv;                // 클래스 변수

    void instanceMethod() {       // 인스턴스 메서드
        System.out.println(iv);   // 인스턴스 변수를 사용할 수 있다.
        System.out.println(cv);   // 클래스 변수를 사용할 수 있다.
    }

    static void staticMethod() { // static메서드
        System.out.println(iv); // 에러. 인스턴스 변수를 사용할 수 없다.
        System.out.println(cv); // 클래스 변수는 사용할 수 있다.
    }
} // end of class
```

이번엔 변수와 메서드간의 호출에 대해서 살펴보자. 메서드간의 호출과 마찬가지로 인스턴스 메서드는 인스턴스 변수를 사용할 수 있지만, static메서드는 인스턴스 변수를 사용할 수 없다.

▼ 예제 6-20/MemberCall.java

```

class MemberCall {
    int iv = 10;
    static int cv = 20;

    int iv2 = cv;
    // static int cv2 = iv;           // 에러. 클래스 변수는 인스턴스 변수를 사용할 수 없음.
    static int cv2 = new MemberCall().iv; // 이처럼 객체를 생성해야 사용가능.

    static void staticMethod1() {
        System.out.println(cv);
    }
    // System.out.println(iv); // 에러. 클래스 메서드에서 인스턴스 변수를 사용불가.
    MemberCall c = new MemberCall();
    System.out.println(c.iv); // 객체를 생성한 후에야 인스턴스 변수의 참조가능.
}

void instanceMethod1() {
    System.out.println(cv);
    System.out.println(iv); // 인스턴스 메서드에서는 인스턴스 변수를 바로 사용가능.
}

static void staticMethod2() {
    staticMethod1();
}
// instanceMethod1(); // 에러. 클래스 메서드에서는 인스턴스 메서드를 호출할 수 없음.
MemberCall c = new MemberCall();
c.instanceMethod1(); // 인스턴스를 생성한 후에야 호출할 수 있음.

void instanceMethod2() { // 인스턴스 메서드에서는 인스턴스 메서드와 클래스 메서드
    staticMethod1(); // 모두 인스턴스 생성없이 바로 호출이 가능하다.
    instanceMethod1();
}
}

```

클래스 멤버(클래스 변수와 클래스 메서드)는 언제나 참조 또는 호출이 가능하기 때문에 인스턴스 멤버가 클래스 멤버를 사용하는 것은 아무런 문제가 안 된다. 클래스 멤버간의 참조 또는 호출 역시 아무런 문제가 없다.

그러나, 인스턴스 멤버(인스턴스 변수와 인스턴스 메서드)는 반드시 객체를 생성한 후에만 참조 또는 호출이 가능하기 때문에 클래스 멤버가 인스턴스 멤버를 참조, 호출하기 위해서는 객체를 생성하여야 한다.

하지만, 인스턴스 멤버간의 호출에는 아무런 문제가 없다. 하나의 인스턴스 멤버가 존재한다는 것은 인스턴스가 이미 생성되어있다는 것을 의미하며, 즉 다른 인스턴스 멤버들도 모두 존재하기 때문이다.

실제로는 같은 클래스 내에서 클래스 멤버가 인스턴스 멤버를 참조 또는 호출해야 하는 경우는 드물다. 만일 그런 경우가 발생한다면, 인스턴스 메서드로 작성해야 할 메서드를 클래스 메서드로 한 것은 아닌지 생각해봐야 한다.

■ 알아두면 좋아요!

수학의 대입법처럼, `c = new MemberCall()`이므로 `c.instanceMethod1();`에서 `c`대신 `new MemberCall()`을 대입하여 사용할 수 있다.

```
MemberCall c = new MemberCall();
int result = c.instanceMethod1();
```

그래서 위의 두 줄을 다음과 같이 한 줄로 할 수 있다.

```
int result = new MemberCall().instanceMethod1();
```

대신 참조변수를 선언하지 않았기 때문에 생성된 MemberCall인스턴스는 더 이상 사용할 수 없다.

4. 오버로딩(overloading)

4.1 오버로딩이란?

메서드도 변수와 마찬가지로 같은 클래스 내에서 서로 구별될 수 있어야 하므로 각기 다른 이름을 가져야 한다. 그러나 자바에서는 한 클래스 내에 이미 같은 이름의 메서드가 있어도 매개변수의 개수 또는 타입이 다르면, 같은 이름의 메서드를 여러 개 정의할 수 있다.

이처럼 한 클래스 내에 같은 이름의 메서드를 여러 개 정의하는 것을 ‘메서드 오버로딩(method overloading)’ 또는 간단히 ‘오버로딩(overloading)’이라 한다.

오버로딩(overloading)의 사전적 의미는 ‘과적하다.’ 즉, 많이 싣는 것을 뜻한다. 보통 하나의 메서드 이름에 하나의 기능을 구현하는데, 하나의 메서드 이름으로 여러 기능을 구현하기 때문에 붙여진 이름다. 앞으로 ‘메서드 오버로딩’을 간단히 ‘오버로딩’이라고 하자.

4.2 오버로딩의 조건

같은 이름의 메서드를 정의한다고 해서 무조건 오버로딩인 것은 아니다. 오버로딩이 성립하기 위해서는 다음과 같은 조건을 만족해야 한다.

1. 메서드 이름이 같아야 한다.
2. 매개변수의 개수 또는 타입이 달라야 한다.
3. 반환타입은 상관없다.

비록 메서드의 이름이 같아도 매개변수가 다르면 서로 구별될 수 있기 때문에 오버로딩이 가능한 것이다. 위의 조건을 만족시키지 못하는 메서드는 중복 정의로 간주되어 컴파일 시에 에러가 발생한다. 그리고 오버로딩된 메서드들은 매개변수에 의해서만 구별될 수 있으므로 반환 타입은 오버로딩을 구현하는데 아무런 영향을 주지 못한다는 것에 주의하자.

4.3 오버로딩의 예

오버로딩의 예로 가장 대표적인 것은 `println`메서드이다. 지금까지 `println`메서드에 꽤나 안에 값만 지정해주면 화면에 출력하는데 아무런 어려움이 없었다.

사실은 `println`메서드를 호출할 때 매개변수로 지정하는 값의 타입에 따라서 호출되는 `println`메서드가 달라진다.

`PrintStream`클래스에는 어떤 종류의 매개변수를 지정해도 출력할 수 있도록 아래와 같이 10개의 오버로딩된 `println`메서드가 정의되어 있다.

```
void println()
void println(boolean x)
void println(char x)
void println(char[] x)
void println(double x)
void println(float x)
void println(int x)
void println(long x)
void println(Object x)
void println(String x)
```

println메서드를 호출할 때 매개변수로 넘겨주는 값의 타입에 따라서 위의 오버로딩된 메서드들 중의 하나가 선택되어 실행되는 것이다.

몇 가지 예를 들어 오버로딩에 대해 자세히 설명하고자 한다.

[보기 1]

```
int add(int a, int b) { return a+b; }
int add(int x, int y) { return x+y; }
```

위의 두 메서드는 매개변수의 이름만 다를 뿐 매개변수의 타입이 같기 때문에 오버로딩이 성립하지 않는다. 매개변수의 이름이 다르면 메서드 내에서 사용되는 변수의 이름이 달라질 뿐, 아무런 의미가 없다. 그래서 이 두 메서드는 완전히 동일한 것이다. 마치 수학에서 ' $f(x) = x + 1$ '과 ' $f(a) = a + 1$ '이 같은 표현인 것과 같다.

컴파일하면, ‘add(int,int) is already defined(이미 같은 메서드가 정의되었다.)’라는 메시지가 나타날 것이다.

[보기 2]

```
int add(int a, int b) { return a+b; }
long add(int a, int b) { return (long)(a + b); }
```

이번 경우는 리턴타입만 다른 경우이다. 매개변수의 타입과 개수가 일치하기 때문에 add(3,3)과 같이 호출하였을 때 어떤 메서드가 호출된 것인지 결정할 수 없기 때문에 오버로딩으로 간주되지 않는다.

이 경우 역시 컴파일하면, ‘add(int,int) is already defined(이미 같은 메서드가 정의되었다.)’라는 메시지가 나타날 것이다.

[보기 3]

```
long add(int a, long b) { return a+b; }
long add(long a, int b) { return a+b; }
```

두 메서드 모두 int형과 long형 매개변수가 하나씩 선언되어 있지만, 서로 순서가 다른 경우이다. 이 경우에는 호출 시 매개변수의 값에 의해 호출될 메서드가 구분될 수 있으므로 오버로딩으로 간주한다.

[ebook – 샘플. 무료 공유] 자바의 정석 4판 Java 21 올컬러 2025. 7. 7 출시 seong.namkung@gmail.com

이처럼 매개변수의 순서만을 다르게 하여 오버로딩을 구현하면, 사용자가 매개변수의 순서를 외우지 않아도 되는 장점이 있지만, 오히려 단점이 될 수도 있기 때문에 주의해야 한다.

예를 들어 add(3,3L)과 같이 호출하면 첫 번째 메서드가, add(3L, 3)과 같이 호출하면 두 번째 메서드가 호출된다. 단, add(3,3)과 같이 호출할 수 없다. 이렇게 호출할 경우, 두 메서드 중 어느 메서드를 호출한 것인지 알 수 없기 때문에 컴파일 에러가 발생한다.

[보기 4]

```
int add(int a, int b) { return a+b; }
long add(long a, long b) { return a+b; }
long add(int[] a) { // 배열의 모든 요소의 합을 반환
    long result = 0;
    for(int i=0; i < a.length; i++) {
        result += a[i];
    }
    return result;
}
```

위 메서드들은 모두 바르게 오버로딩되어 있다. 정의된 매개변수가 서로 다른 해도, 세 메서드 모두 매개변수로 넘겨받은 값을 더해서 그 결과를 돌려주는 일을 한다.

같은 일을 하지만 매개변수를 달리해야하는 경우에, 이와 같이 이름은 같고 매개변수를 다르게 하여 오버로딩을 구현한다.

4.4 오버로딩의 장점

지금까지 오버로딩의 정의와 성립하기 위한 조건을 알아보았다. 그렇다면 오버로딩을 구현함으로써 얻는 이득은 무엇인가에 대해서 생각해보자.

만일 메서드도 변수처럼 단지 이름만으로 구별된다면, 한 클래스내의 모든 메서드들은 이름이 달라야한다. 그렇다면, 이전에 예로 들었던 10가지의 println메서드들은 각기 다른 이름을 가져야 한다.

예를 들면, 아래와 같은 방식으로 메서드 이름이 변경되어야 할 것이다.

```
void println()
void printlnBoolean(boolean x)
void printlnChar(char x)
void printlnDouble(double x)
void printlnString(String x)
```

모두 근본적으로는 같은 기능을 하는 메서드지만, 서로 다른 이름을 가져야 하기 때문에 메서드를 작성하는 쪽은 이름을 짓기 어렵고, 메서드를 사용하는 쪽도 이름을 일일이 구분해서 기억해야하기 때문에 서로 부담이 된다.

[ebook – 샘플. 무료 공유] 자바의 정석 4판 Java 21 올컬러 2025. 7. 7 출시 seong.namkung@gmail.com

[ebook – 샘플. 무료 공유] 자바의 정석 4판 Java 21 올컬러 2025. 7. 7 출시 seong.namkung@gmail.com

하지만 오버로딩으로 여러 메서드들이 println이라는 하나의 이름으로 정의될 수 있다면, println이라는 이름만 기억하면 되므로 기억하기도 쉽고 이름도 짧게 할 수 있어서 유동의 가능성을 줄일 수 있다. 그리고 메서드의 이름만 보고도 ‘이 메서드들은 이름이 같으니, 같은 기능을 하겠구나.’라고 쉽게 예측할 수 있게 된다.

또 하나의 장점은 메서드의 이름을 절약할 수 있다는 것이다. 하나의 이름으로 여러 개의 메서드를 정의할 수 있으니, 메서드의 이름을 짓는데 고민을 덜 수 있는 동시에 사용되었어야 할 메서드 이름을 다른 메서드의 이름으로 사용할 수 있기 때문이다.

▼ 예제 6-21/OverloadingEx.java

```
class OverloadingEx {
    public static void main(String args[]) {
        MyMath3 mm = new MyMath3();
        System.out.println("mm.add(3, 3) 결과: " + mm.add(3, 3));
        System.out.println("mm.add(3L, 3) 결과: " + mm.add(3L, 3));
        System.out.println("mm.add(3, 3L) 결과: " + mm.add(3, 3L));
        System.out.println("mm.add(3L, 3L) 결과: " + mm.add(3L, 3L));

        int[] a = {100, 200, 300};
        System.out.println("mm.add(a) 결과: " + mm.add(a));
    }
}

class MyMath3 {
    int add(int a, int b) {
        System.out.print("int add(int a, int b) - ");
        return a+b;
    }

    long add(int a, long b) {
        System.out.print("long add(int a, long b) - ");
        return a+b;
    }

    long add(long a, int b) {
        System.out.print("long add(long a, int b) - ");
        return a+b;
    }

    long add(long a, long b) {
        System.out.print("long add(long a, long b) - ");
        return a+b;
    }

    int add(int[] a) { // 배열의 모든 요소의 합을 결과로 돌려준다.
        System.out.print("int add(int[] a) - ");
        int result = 0;
        for(int i=0; i < a.length;i++) {
            result += a[i];
        }
        return result;
    }
}
```

[ebook – 샘플. 무료 공유] 자바의 정석 4판 Java 21 올컬러 2025. 7. 7 출시 seong.namkung@gmail.com

▼ 실행결과

```
int add(int a, int b) - mm.add(3, 3) 결과:6
long add(long a, int b) - mm.add(3L, 3) 결과: 6
long add(int a, long b) - mm.add(3, 3L) 결과: 6
long add(long a, long b) - mm.add(3L, 3L) 결과: 6
int add(int[] a) - mm.add(a) 결과: 600
```

| 참고 | add(3L, 3), add(3, 3L), add(3L, 3L)의 결과는 모두 6L이지만, System.out.println(6L);을 수행하면 6이 출력된다. 리터럴의 접미사는 출력되지 않기 때문이다.

실행결과의 출력순서를 보고 의아할 수도 있다. 어떻게 add메서드가 println메서드보다 먼저 출력될 수 있는가?

```
System.out.println("mm.add(3, 3) 결과:" + mm.add(3,3));
```

println메서드가 결과를 출력하려면, add메서드의 결과가 먼저 계산되어야 하기 때문이다. 간단히 위의 문장이 아래의 두 문장을 하나로 합친 것이라고 생각하면 이해가 쉬울 것이다.

```
int result = mm.add(3,3);
System.out.println("mm.add(3, 3) 결과:" + result);
```

4.5 가변인자(varargs)와 오버로딩

기존에는 메서드의 매개변수 개수가 고정적이었으나 JDK 5부터 동적으로 지정해 줄 수 있게 되었으며, 이 기능을 ‘가변인자(variable arguments)’라고 한다.

가변인자는 ‘타입... 변수명’과 같은 형식으로 선언하며, PrintStream클래스의 printf()가 대표적인 예이다.

```
public PrintStream printf(String format, Object... args) { ... }
```

위와 같이 가변인자 외에도 매개변수가 더 있다면, 가변인자를 매개변수 중에서 제일 마지막에 선언해야 한다. 그렇지 않으면, 컴파일 에러가 발생한다. 가변인자인지 아닌지를 구별할 방법이 없기 때문에 허용하지 않는 것이다.

```
// 컴파일 에러 발생 - 가변인자는 항상 마지막 매개변수이어야 한다.
public PrintStream printf(Object... args, String format) {
    ...
}
```

[ebook – 샘플. 무료 공유] 자바의 정석 4판 Java 21 올컬러 2025. 7. 7 출시 seong.namkung@gmail.com

만일 여러 문자열을 하나로 결합하여 반환하는 concatenate메서드를 작성한다면, 아래와 같이 매개변수의 개수를 다르게 해서 여러 개의 메서드를 작성해야 할 것이다.

```
String concatenate(String s1, String s2) { ... }  
String concatenate(String s1, String s2, String s3) { ... }  
String concatenate(String s1, String s2, String s3, String s4){ ... }
```

이럴 때, 가변인자를 사용하면 메서드 하나로 간단히 대체할 수 있다.

```
String concatenate(String... str) { ... }
```

이 메서드를 호출할 때는 아래와 같이 인자의 개수를 가변적으로 할 수 있다. 심지어는 인자가 아예 없어도 되고 배열도 인자가 될 수 있다.

```
System.out.println(concatenate()); // 인자가 없음  
System.out.println(concatenate("a")); // 인자가 하나  
System.out.println(concatenate("a", "b")); // 인자가 둘  
System.out.println(concatenate(new String[] {"A", "B"})); // 배열도 가능
```

이쯤에서 아마도 눈치를 챘을 것이다. 그렇다. 가변인자는 내부적으로 배열을 이용하는 것이다. 그래서 가변인자가 선언된 메서드를 호출할 때마다 배열이 새로 생성된다. 가변인자가 편리하지만, 이런 비효율이 숨어있으므로 꼭 필요한 경우에만 가변인자를 사용하자.

그러면 가변인자는 아래와 같이 매개변수의 타입을 배열로 하는 것과 어떤 차이가 있는가?

```
String concatenate(String[] str) { ... }  
  
String result = concatenate(new String[0]); // 인자로 배열을 지정  
String result = concatenate(null); // 인자로 null을 지정  
String result = concatenate(); // 에러. 인자가 필요함.
```

매개변수의 타입을 배열로 하면, 반드시 인자를 지정해 줘야하기 때문에, 위의 코드에서처럼 인자를 생략할 수 없다. 그래서 null이나 길이가 0인 배열을 인자로 지정해줘야 하는 불편함이 있다.

| 참고 | C언어와 달리 자바에서는 길이가 0인 배열이 허용된다.

가변인자를 오버로딩할 때 한 가지 주의해야 할 점이 있는데, 먼저 예제부터 살펴보자.

[ebook – 샘플. 무료 공유] 자바의 정석 4판 Java 21 올컬러 2025. 7. 7 출시 seong.namkung@gmail.com

▼ 예제 6-22/VarArgsEx.java

```

class VarArgsEx {
    public static void main(String[] args) {
        String[] strArr = { "100", "200", "300" };

        System.out.println(concatenate("", "100", "200", "300"));
        System.out.println(concatenate("-", strArr));
        System.out.println(concatenate(", ", new String[]{"1", "2", "3"}));
        System.out.println("[ "+concatenate(", ", new String[]{})+"]");
        System.out.println("[ "+concatenate(", ")+" ]");
    }

    static String concatenate(String delim, String... args) {
        String result = "";

        for(String str : args) {
            result += str + delim;
        }

        return result;
    }

    /*
     * static String concatenate(String... args) {
     *     return concatenate("", args);
     * }
    */
} // class

```

▼ 실행결과

```

100200300
100-200-300-
1,2,3,
[]
[]

```

concatenate메서드는 매개변수로 입력된 문자열에 구분자를 사이에 포함시켜 결합해서 반환한다. 가변인자로 매개변수를 선언했기 때문에 문자열을 개수의 제약없이 매개변수로 지정할 수 있다.

```

String[] strArr = new String[]{"100", "200", "300"};
System.out.println(concatenate("-", strArr));

```

위의 두 문장을 하나로 합치면 아래와 같이 쓸 수 있다.

```
System.out.println(concatenate("-", new String[]{"100", "200", "300"}));
```

그러나 아래와 같은 문장은 허용되지 않는다는 것에 주의하자.

```
System.out.println(concatenate("-", {"100", "200", "300"}));
```

위의 예제에서는 주석처리하였지만, concatenate메서드의 또 다른 오버로딩된 메서드가 있다.

```

static String concatenate(String delim, String... args) {
    String result = "";

    for(String str : args) {
        result += str + delim;
    }

    return result;
}

static String concatenate(String... args) {
    return concatenate("",args);
}

```

이 두 메서드는 별 문제가 없어 보이지만 위의 예제에서 주석을 풀고 컴파일을 하면 아래와 같이 컴파일에러가 발생한다.

```

VarArgsEx.java:5: error: reference to concatenate is ambiguous
    System.out.println(concatenate("-", "100", "200", "300"));
                           ^
both method concatenate(String, String...) in VarArgsEx and method
concatenate(String...) in VarArgsEx match
1 error

```

에러의 내용을 살펴보면 두 오버로딩된 메서드가 구분되지 않아서 발생하는 것임을 알 수 있다. 가변인자를 선언한 메서드를 오버로딩하면, 메서드를 호출했을 때 이와 같이 구별되지 못하는 경우가 발생하기 쉽기 때문에 주의해야 한다. 가능하면 가변인자를 사용한 메서드는 오버로딩하지 않는 것이 좋다.

5. 생성자(Constructor)

5.1 생성자란?

생성자는 인스턴스가 생성될 때 호출되는 ‘인스턴스 초기화 메서드’이다. 따라서 인스턴스 변수의 초기화 작업에 주로 사용되며, 인스턴스 생성 시에 실행되어야 할 작업을 위해서도 사용된다.

| 참고 | 인스턴스 초기화란, 인스턴스 변수들을 초기화하는 것을 뜻한다.

생성자도 일종의 메서드지만 리턴값이 없다는 점이 일반적인 메서드와 다르다. 그리고 생성자 앞에 리턴값이 없음을 뜻하는 키워드 void를 붙이지 않는다. 생성자의 조건은 다음과 같다.

1. 생성자의 이름은 클래스의 이름과 같아야 한다.
2. 생성자는 리턴 값이 없다.

| 참고 | 모든 생성자가 예외없이 리턴값이 없으므로 void를 붙이지 않는 것이다.

생성자는 다음과 같이 정의한다. 생성자도 오버로딩이 가능하므로 하나의 클래스에 여러 개의 생성자가 존재할 수 있다.

```
클래스이름 (타입 변수명, 타입 변수명, ... ) {
    // 인스턴스 생성 시 수행될 코드,
    // 주로 인스턴스 변수의 초기화 코드를 적는다.
}

class Card {
    Card() {           // 매개변수가 없는 생성자.
        ...
    }

    Card(String k, int num) {           // 매개변수가 있는 생성자.
        ...
    }
}
```

연산자 new가 인스턴스를 생성하는 것이지 생성자가 인스턴스를 생성하는 것이 아니다. 생성자라는 용어 때문에 오해하기 쉬운데, 생성자는 단순히 인스턴스 변수들의 초기화에 사용되는 조금 특별한 메서드일 뿐이다. 생성자가 갖는 몇 가지 특징만 제외하면 일반적인 메서드와 다르지 않다.

Card클래스의 인스턴스를 생성하는 코드를 예로 들어, 수행되는 과정을 단계별로 나누어 보면 다음과 같다.

```
Card c = new Card();
```

1. 연산자 new에 의해서 메모리(heap)에 Card클래스의 인스턴스가 생성된다.
2. 생성자 Card()가 호출되어 수행된다.
3. 생성된 Card인스턴스의 주소가 참조변수 c에 저장된다.

지금까지 인스턴스를 생성하기 위해 사용해왔던 ‘클래스이름()’이 바로 생성자였던 것이다. 인스턴스를 생성할 때는 반드시 클래스 내에 정의된 생성자 중의 하나를 선택하여 지정해주어야 한다.

5.2 기본 생성자(default constructor)

지금까지는 생성자를 모르고도 프로그래밍을 해 왔지만, 사실 모든 클래스에는 반드시 하나 이상의 생성자가 정의되어 있어야 한다.

그러나 지금까지 클래스에 생성자를 정의하지 않고도 인스턴스를 생성할 수 있었던 이유는 컴파일러가 제공하는 ‘기본 생성자(default constructor)’ 덕분이었다.

컴파일 할 때, 소스파일(*.java)의 클래스에 생성자가 하나도 정의되지 않은 경우 컴파일러는 자동적으로 아래와 같은 내용의 기본 생성자를 추가한다.

```
클래스이름 () { }  
Card() { }
```

컴파일러가 자동적으로 추가해주는 기본 생성자는 이와 같이 매개변수도 없고 아무런 내용도 없는 아주 간단한 것이다.

그동안 우리는 인스턴스를 생성할 때 컴파일러가 제공한 기본 생성자를 사용해왔던 것이다. 특별히 인스턴스 초기화 작업이 요구되어지지 않는다면 생성자를 정의하지 않고 컴파일러가 제공하는 기본 생성자를 사용하는 것도 좋다.

| 참고 | 클래스의 ‘접근 제어자(access modifier)’가 public인 경우에는 기본 생성자로 ‘public 클래스이름() {}’이 추가된다.

▼ 예제 6-23/ConstructorEx.java

```
class Data_1 {  
    int value;  
}
```

[ebook – 샘플. 무료 공유] 자바의 정석 4판 Java 21 올컬러 2025. 7. 7 출시 seong.namkung@gmail.com

```

class Data_2 {
    int value;

    Data_2(int x) {      // 매개변수가 있는 생성자.
        value = x;
    }
}

class ConstructorEx {
    public static void main(String[] args) {
        Data_1 d1 = new Data_1();
        Data_2 d2 = new Data_2();      // compile error 발생
    }
}

```

▼ 실행결과

```

ConstructorEx.java:15: cannot resolve symbol
symbol : constructor Data_2 ()
location: class Data_2
        Data_2 d2 = new Data_2();      // compile error 발생
^
1 error

```

이 예제를 컴파일 하면 위와 같은 에러 메시지가 나타난다. 이것은 Data_2에서 Data_2()라는 생성자를 찾을 수 없다는 내용인데, Data_2에 생성자 Data_2()가 정의되어 있지 않기 때문에 에러가 발생한 것이다.

Data_1의 인스턴스를 생성하는 코드에는 에러가 없는데, Data_2의 인스턴스를 생성하는 코드에서 에러가 발생하는 이유는 무엇일까?

그 이유는 Data_1에는 정의되어 있는 생성자가 하나도 없으므로 컴파일러가 기본 생성자를 추가해주었지만, Data_2에는 이미 생성자 Data_2(int x)가 정의되어 있으므로 기본 생성자가 추가되지 않았기 때문이다.

컴파일러가 자동적으로 기본 생성자를 추가해주는 경우는 ‘클래스 내에 생성자가 하나도 없을 때’뿐이라는 것을 명심해야 한다.

```

Data_1 d1 = new Data_1();
Data_2 d2 = new Data_2(); // 에러

```



```

Data_1 d1 = new Data_1();
Data_2 d2 = new Data_2(10); // OK

```

이 예제에서 컴파일 에러가 발생하지 않도록 하기 위해서는 위의 오른쪽 코드와 같이 Data_2의 인스턴스를 생성할 때 생성자 Data_2(int x)를 사용하던가, 아니면 클래스 Data_2에 생성자 Data_2()를 추가로 정의해줘야 한다.

기본 생성자가 컴파일러에 의해서 자동 추가되는 경우는
 생성자가 하나도 없을 때 뿐이다.

[ebook – 샘플. 무료 공유] 자바의 정석 4판 Java 21 올컬러 2025. 7. 7 출시 seong.namkung@gmail.com

5.3 매개변수가 있는 생성자

생성자도 메서드처럼 매개변수를 선언하여 호출 시 값을 넘겨받아서 인스턴스의 초기화 작업에 사용할 수 있다. 인스턴스마다 각기 다른 값으로 초기화되어야 하는 경우가 많기 때문에 매개변수를 사용한 초기화는 매우 유용하다.

아래의 코드는 자동차를 클래스로 정의한 것인데, 단순히 color, gearType, door 세 개의 인스턴스 변수와 두 개의 생성자를 가지고 있다.

```
class Car {
    String color;           // 색상
    String gearType;        // 변속기 종류 - auto(자동), manual(수동)
    int door;               // 문의 개수

    Car() {}   // 기본 생성자
    Car(String c, String g, int d) { // 매개변수가 있는 생성자
        color = c;
        gearType = g;
        door = d;
    }
}
```

Car인스턴스를 생성할 때, 생성자 Car()를 사용하면, 인스턴스를 생성한 후에 인스턴스 변수들을 따로 초기화해야 하지만, 매개변수가 있는 생성자 Car(String color, String gearType, int door)를 이용하면 인스턴스를 생성하는 동시에 원하는 값으로 초기화를 할 수 있다.

인스턴스를 생성한 다음에 인스턴스 변수의 값을 변경하는 것보다 매개변수를 갖는 생성자를 사용하는 것이 코드를 보다 짧고 변경에 유리하게 만든다.

```
Car c = new Car();
c.color = "white";
c.gearType = "auto";
c.door = 4;
```

→

```
Car c = new Car("white", "auto", 4);
```

위의 양쪽 코드 모두 같은 내용이지만, 오른쪽의 코드가 더 간결하고 직관적이다. 이처럼 클래스를 작성할 때 다양한 생성자를 제공함으로써 인스턴스 생성 후에 추가로 초기화를 하지 않아도 되게 하는 것이 좋다.

▼ 예제 6-24/CarEx.java

```
class Car {
    String color;           // 색상
    String gearType;        // 변속기 종류 - auto(자동), manual(수동)
    int door;               // 문의 개수

    Car() {}
```

```

Car(String c, String g, int d) {
    color = c;
    gearType = g;
    door = d;
}

class CarEx {
    public static void main(String[] args) {
        Car c1 = new Car();
        c1.color = "white";
        c1.gearType = "auto";
        c1.door = 4;

        Car c2 = new Car("white", "auto", 4);

        System.out.println("c1의 color=" + c1.color + ", gearType="
                           + c1.gearType + ", door=" + c1.door);
        System.out.println("c2의 color=" + c2.color + ", gearType="
                           + c2.gearType + ", door=" + c2.door);
    }
}

```

▼ 실행결과

```
c1의 color=white, gearType=auto, door=4
c2의 color=white, gearType=auto, door=4
```

5.4 생성자에서 다른 생성자 호출하기 – this(), this

같은 클래스의 멤버들 간에 서로 호출할 수 있는 것처럼 생성자도 서로 호출이 가능하다. 단, 다음의 두 조건을 만족시켜야 한다.

- 생성자의 이름으로 클래스이름 대신 this를 사용한다.
- 다른 생성자를 호출할 때 반드시 첫 줄에서만 호출이 가능하다.

다음의 코드는 생성자를 작성할 때 지켜야하는 두 조건을 모두 만족시키지 못했기 때문에 에러가 발생한다.

```

Car(String color) {
    door = 5; // 첫 번째 줄
    Car(color, "auto", 4); // 에러1. 생성자의 두 번째 줄에서 다른 생성자 호출
} // 에러2. this(color, "auto", 4);로 해야함

```

생성자 내에서 다른 생성자를 호출할 때는 클래스이름인 ‘Car’ 대신 ‘this’를 사용해야하는데 그러지 않아서 에러이고, 또 다른 에러는 생성자 호출이 첫 번째 줄이 아닌 두 번째 줄이기 때문에 에러이다.

생성자에서 다른 생성자를 첫 줄에서만 호출이 가능하도록 한 이유는 생성자 내에서 초기화 작업 중에 다른 생성자를 호출하면, 호출된 다른 생성자 내에서도 멤버변수들의 값을 초기화를 할 것이므로 이전의 초기화 작업이 무의미해질 수 있기 때문이다.

▼ 예제 6-25/CarEx2.java

```

class Car2 {
    String color;          // 색상
    String gearType;       // 변속기 종류 - auto(자동), manual(수동)
    int door;              // 문의 개수

    Car2() {
        this("white", "auto", 4); •
    }

    Car2(String color) {
        this(color, "auto", 4);
    }

    Car2(String color, String gearType, int door) {
        this.color = color;
        this.gearType = gearType;
        this.door = door;
    }
}

class CarEx2 {
    public static void main(String[] args) {
        Car2 c1 = new Car2();
        Car2 c2 = new Car2("blue");

        System.out.println("c1의 color=" + c1.color + ", gearType="
                           + c1.gearType + ", door=" + c1.door);
        System.out.println("c2의 color=" + c2.color + ", gearType="
                           + c2.gearType + ", door=" + c2.door);
    }
}

```

Car(String color, String gearType, int door)를 호출

▼ 실행결과

c1의 color=white, gearType=auto, door=4
c2의 color=blue, gearType=auto, door=4

생성자 Car2()에서 또 다른 생성자 Car2(String color, String gearType, int door)를 호출하였다. 이처럼 생성자간의 호출에는 생성자의 이름 대신 this를 사용해야 하므로 ‘Car2’ 대신 ‘this’를 사용했다. 그리고 생성자 Car2()의 첫째 줄에서 호출하였다는 점을 눈여겨보기 바란다.

```

Car2() {
    color = "white";
    gearType = "auto";
    door = 4;
}

→

Car2() {
    this("white", "auto", 4);
}

```

위 코드는 양쪽 모두 같은 일을 하지만 오른쪽 코드는 생성자 Car2(String color, String gearType, int door)로 더 간략히 한 것이다. Car2 c1 = new Car2();와 같이 생성자 Car2()로 Car2인스턴스를 생성한 경우에, 인스턴스변수 color는 “white”, gearType은 “auto”, door는 4로 초기화 되도록 하였다.

[ebook – 샘플. 무료 공유] 자바의 정석 4판 Java 21 올컬러 2025. 7. 7 출시 seong.namkung@gmail.com

이것은 마치 실생활에서 자동차(Car인스턴스)를 생산할 때, 아무런 옵션도 주지 않으면, 기본적으로 흰색(white)에 자동변속기어(auto) 그리고 문의 개수가 4개인 자동차가 생산되도록 하는 것에 비유할 수 있다.

같은 클래스 내의 생성자들은 일반적으로 서로 관계가 깊어서 이처럼 서로 호출하도록 하면 코드의 중복을 줄이고 더 좋은 코드를 얻을 수 있다. 그리고 수정이 필요한 경우에도 보다 적게 변경하면 되므로 유지보수가 쉬워진다.

```
Car2(String c, String g, int d) {
    color = c;
    gearType = g;
    door = d;
}

Car2(String color, String gearType,
      int door) {
    this.color = color;
    this.gearType = gearType;
    this.door = door;
}
```

왼쪽 코드의 ‘color = c;’는 생성자의 매개변수로 선언된 지역변수 c의 값을 인스턴스변수 color에 저장한다. 이 때 변수 color와 c는 이름만으로도 서로 구별되므로 아무런 문제가 없다.

하지만, 오른쪽 코드처럼 생성자의 매개변수로 선언된 변수의 이름이 color로 인스턴스 변수 color와 같을 경우에는 이름만으로 두 변수가 서로 구별이 안 된다. 이런 경우에는 인스턴스 변수 앞에 ‘this’를 붙이면 구별이 가능하다.

`this.color`은 인스턴스 변수이고, `color`은 생성자의 매개변수로 선언된 지역 변수이다. 왼쪽 코드와 같이 매개변수 이름을 다르게 하는 것 보다 ‘this’로 구별되게 하는 것이 의미가 더 명확하고 이해하기 쉽다.

‘this’는 참조 변수이며 인스턴스 자신을 가리킨다. 참조변수를 통해 인스턴스의 멤버에 접근할 수 있는 것처럼, ‘this’로 인스턴스 변수에 접근할 수 있는 것이다.

하지만, ‘this’를 사용할 수 있는 것은 인스턴스 멤버뿐이다. static메서드(클래스 메서드)에서는 인스턴스 멤버들을 사용할 수 없는 것처럼, ‘this’ 역시 사용할 수 없다. 왜냐하면, static메서드는 인스턴스를 생성하지 않고도 호출될 수 있으므로 static메서드가 호출된 시점에 인스턴스(this)가 존재하지 않을 수도 있기 때문이다.

생성자를 포함한 모든 인스턴스 메서드에는 인스턴스 자신을 가리키는 참조변수 ‘this’가 지역 변수로 숨겨진 채로 존재한다. 일반적으로 인스턴스 메서드는 인스턴스와 관련된 작업을 하기 때문에 인스턴스의 정보가 필요하지만, static메서드는 인스턴스와 관련 없는 작업을 하므로 인스턴스에 대한 정보가 필요없다.

this 인스턴스 자신을 가리키는 참조 변수, 인스턴스의 주소가 저장되어 있으며
모든 인스턴스 메서드에 지역변수로 숨겨진 채로 존재한다.

this(), this(매개변수) 생성자, 같은 클래스의 다른 생성자를 호출할 때 사용한다.

| 참고 | `this`와 `this()`는 생긴 것 만 비슷할 뿐 전혀 다른 것이다. `this`는 ‘참조 변수’이고, `this()`는 ‘생성자’이다.

[ebook – 샘플. 무료 공유] 자바의 정석 4판 Java 21 올컬러 2025. 7. 7 출시 seong.namkung@gmail.com

5.5 생성자를 이용한 인스턴스의 복사

현재 사용하고 있는 인스턴스와 같은 상태를 갖는 인스턴스를 하나 더 만들고자 할 때 생성자를 이용할 수 있다. 두 인스턴스가 같은 상태를 갖는다는 것은 두 인스턴스의 모든 인스턴스 변수(상태)가 동일한 값을 갖고 있다는 것을 뜻한다.

하나의 클래스로부터 생성된 모든 인스턴스의 메서드와 클래스 변수는 서로 동일하기 때문에 인스턴스간의 차이는, 인스턴스마다 각기 다른 값을 가질 수 있는 인스턴스 변수뿐이다.

```
Car(Car c) {
    color      = c.color;
    gearType  = c.gearType;
    door       = c.door;
}
```

위의 코드는 Car클래스의 참조 변수를 매개변수로 선언한 생성자이다. 매개변수로 넘겨진 참조 변수가 가리키는 Car인스턴스의 인스턴스 변수인 color, gearType, door의 값을 인스턴스 자신으로 복사하는 것이다.

이렇게 하면 어떤 인스턴스의 상태를 전혀 알지 못해도 똑같은 상태의 인스턴스를 추가로 생성할 수 있다. Java API의 많은 클래스들이 인스턴스의 복사를 위한 생성자를 제공하고 있다.

| 참고 | Object클래스에 정의된 clone()을 이용하면 간단히 인스턴스를 복사할 수 있다. p.486

▼ 예제 6-26/CarEx3.java

```
class Car3 {
    String color;          // 색상
    String gearType;       // 변속기 종류 - auto(자동), manual(수동)
    int door;              // 문의 개수

    Car3() {
        this("white", "auto", 4);
    }

    Car3(Car3 c) { // 인스턴스의 복사를 위한 생성자.
        color      = c.color;
        gearType  = c.gearType;
        door       = c.door;
    }

    Car3(String color, String gearType, int door) {
        this.color      = color;
        this.gearType  = gearType;
        this.door       = door;
    }
}
```

```

class CarEx3 {
    public static void main(String[] args) {
        Car3 c1 = new Car3();
        Car3 c2 = new Car3(c1); // c1의 복사본 c2를 생성한다.
        System.out.println("c1의 color=" + c1.color + ", gearType="
                           + c1.gearType + ", door=" + c1.door);
        System.out.println("c2의 color=" + c2.color + ", gearType="
                           + c2.gearType + ", door=" + c2.door);
        c1.door=100; // c1의 인스턴스 변수 door의 값을 변경한다.
        System.out.println("c1.door=100; 수행 후");
        System.out.println("c1의 color=" + c1.color + ", gearType="
                           + c1.gearType + ", door=" + c1.door);
        System.out.println("c2의 color=" + c2.color + ", gearType="
                           + c2.gearType + ", door=" + c2.door);
    }
}

```

▼ 실행결과

```

c1의 color=white, gearType=auto, door=4
c2의 color=white, gearType=auto, door=4
c1.door=100; 수행 후
c1의 color=white, gearType=auto, door=100
c2의 color=white, gearType=auto, door=4

```

인스턴스 c2는 c1을 복사하여 생성된 것이므로 서로 같은 상태를 갖지만, 서로 독립적으로 메모리 공간에 존재하는 별도의 인스턴스이므로 c1의 값들이 변경되어도 c2는 영향을 받지 않는다.

생성자 ‘Car(Car c)’는 아래와 같이 다른 생성자인 ‘Car(String color, String gearType, int door)’를 호출하는 것이 바람직하다. 무작정 새로 코드를 작성하는 것보다 기존의 코드를 최대한 활용하도록 고민해야 한다.

```

Car(Car c) {
    color      = c.color;
    gearType   = c.gearType;
    door       = c.door;
}

Car(Car c) {
    // Car(String color, String gearType, int door)
    this(c.color, c.gearType, c.door);
}

```

지금까지 생성자에 대해서 모르고도 자바프로그래밍이 가능했던 것을 생각한다면, 생성자는 그리 중요하지 않은 것으로 생각될지도 모른다. 하지만, 지금까지 본 것처럼 생성자를 잘 활용하면 보다 간결하고 직관적인 코드를 작성할 수 있을 것이다.

인스턴스를 생성할 때는 다음의 2가지 사항을 결정해야 한다.

1. 클래스 – 어떤 클래스의 인스턴스를 생성할 것인가?
2. 생성자 – 선택한 클래스의 어떤 생성자로 인스턴스를 생성할 것인가?

6 변수의 초기화

6.1 변수의 초기화

변수를 선언하고 처음으로 값을 저장하는 것을 ‘변수의 초기화’라고 한다. 변수의 초기화는 경우에 따라서 필수적이기도 하고 선택적이기도 하지만, 가능하면 선언과 동시에 적절한 값으로 초기화 하는 것이 바람직하다.

멤버 변수는 초기화를 하지 않아도 자동적으로 변수의 자료형에 맞는 기본값으로 초기화가 이루어지므로 초기화하지 않고 사용할 수 있지만, 지역변수는 사용하기 전에 반드시 초기화해야 한다. 인스턴스 변수와 클래스 변수는 초기화가 필수가 아니라서 타입 추론(var)을 허용하지 않는다.

```
class InitTest {
    int x;                      // 인스턴스 변수. 타입 추론(var) 불가
    int y = x;                  // 인스턴스 변수. 타입 추론(var) 불가

    void method1() {
        int i;                  // 지역 변수
        int j = i;              // 에러. 지역 변수를 초기화하지 않고 사용
    }
}
```

위의 코드에서 x, y는 인스턴스 변수이고, i, j는 지역변수이다. 그 중 x와 i는 선언만 하고 초기화를 하지 않았다. 그리고 y를 초기화하는데 x를 사용하였고, j를 초기화하는데 i를 사용하였다.

인스턴스 변수 x는 초기화를 해주지 않아도 자동적으로 int형의 기본값인 0으로 초기화되므로, ‘int y = x;’와 같이 할 수 있다. x의 값이 0이므로 y역시 0이 저장된다.

하지만, method1()의 지역 변수 i는 자동적으로 초기화되지 않으므로, 초기화되지 않은 상태에서 변수 j를 초기화하는데 사용될 수 없다. 컴파일하면, 에러가 발생한다.

**멤버 변수(클래스 변수와 인스턴스 변수)와 배열의 초기화는 선택이지만,
지역 변수의 초기화는 필수이다.**

참고로 각 타입의 기본값(default value)은 다음과 같다.

자료형	기본값
boolean	false
char	'\u0000'
byte, short, int	0
long	0L
float	0.0f
double	0.0d 또는 0.0
참조형	null

변수의 초기화에 대한 예를 몇 가지 더 살펴보자.

선언예	설명
int i=10; int j=10;	int형 변수 i를 선언하고 10으로 초기화 한다. int형 변수 j를 선언하고 10으로 초기화 한다.
int i=10, j=10;	같은 타입의 변수는 콤마(.)를 사용해서 함께 선언하거나 초기화할 수 있다.
int i=10, long j=0;	에러. 타입이 다른 변수는 함께 선언하거나 초기화할 수 없다.
int i=10; int j=i;	변수 i에 저장된 값으로 변수 j를 초기화 한다. 변수 j는 i의 값인 10으로 초기화 된다.
int j=i; int i=10;	에러. 변수 i가 선언되기 전에 i를 사용할 수 없다.

▲ 표 6-4 다양한 초기화 방법

멤버 변수의 초기화는 지역 변수와 달리 여러 가지 방법이 있는데 앞으로 멤버 변수의 초기화에 대한 모든 방법에 대해 비교, 정리할 것이다.

▶ 멤버 변수의 초기화 방법

1. 명시적 초기화(explicit initialization)
2. 생성자(constructor)
3. 초기화 블럭(initialization block)
 - 인스턴스 초기화 블럭 : 인스턴스 변수를 초기화하는데 사용.
 - 클래스 초기화 블럭 : 클래스 변수를 초기화하는데 사용.

6.2 명시적 초기화(explicit initialization)

변수를 선언과 동시에 초기화하는 것을 명시적 초기화라고 한다. 가장 기본적이면서도 간단한 초기화 방법이므로 여러 초기화 방법 중에서 가장 우선적으로 고려되어야 한다.

```
class Car {
    int door = 4;           // 기본형(primitive type) 변수의 초기화
    Engine e = new Engine(); // 참조형(reference type) 변수의 초기화

    ...
}
```

명시적 초기화가 간단하고 명료하긴 하지만, 보다 복잡한 초기화가 필요할 때는 ‘초기화 블럭(initialization block)’ 또는 생성자를 사용해야 한다.

6.3 초기화 블럭(initialization block)

초기화 블럭에는 ‘클래스 초기화 블럭’과 ‘인스턴스 초기화 블럭’ 두 가지 종류가 있다. 클래스 초기화 블럭은 클래스 변수의 초기화에 사용되고, 인스턴스 초기화 블럭은 인스턴스 변수의 초기화에 사용된다.

클래스 초기화 블럭	클래스 변수의 복잡한 초기화에 사용
인스턴스 초기화 블럭	인스턴스 변수의 복잡한 초기화에 사용

초기화 블럭을 작성하려면, 인스턴스 초기화 블럭은 단순히 클래스 내에 블럭{} 만들고 그 안에 코드를 작성하기만 하면 된다. 그리고 클래스 초기화 블럭은 인스턴스 초기화 블럭 앞에 단순히 static을 붙이기만 하면 된다.

초기화 블럭 내에는 조건문, 반복문, 예외처리구문 등을 자유롭게 사용할 수 있으므로, 초기화 작업이 복잡하여 명시적 초기화만으로는 부족한 경우 초기화 블럭을 사용한다.

```
class InitBlock {
    static { /* 클래스 초기화 블럭입니다. */ }

    { /* 인스턴스 초기화 블럭입니다. */ }

    // ...
}
```

클래스 초기화 블럭은 클래스가 메모리에 처음 로딩될 때 한번만 수행되며, 인스턴스 초기화 블럭은 생성자와 같이 인스턴스를 생성할 때마다 수행된다.

그리고 생성자보다 인스턴스 초기화 블럭이 먼저 수행된다는 사실도 기억해두자.

| 참고 | 클래스가 처음 로딩될 때 클래스 변수들이 자동적으로 메모리에 만들어지고, 곧바로 클래스 초기화 블럭이 클래스 변수들을 초기화한다.

인스턴스 변수의 초기화는 주로 생성자를 사용하고, 인스턴스 초기화 블럭은 모든 생성자에서 공통으로 수행돼야 하는 코드를 넣는데 사용한다.

```
Car() {
    count++;
    serialNo = count;
    color = "White";
    gearType = "Auto";
}

Car(String color, String gearType) {
    count++;
    serialNo = count;
    this.color = color;
    this.gearType = gearType;
}
```

[ebook – 샘플. 무료 공유] 자바의 정석 4판 Java 21 올컬러 2025. 7. 7 출시 seong.namkung@gmail.com

예를 들면, 위와 같이 클래스의 모든 생성자에 공통으로 수행되어야 하는 문장들이 있을 때, 이 문장들을 모든 생성자에 넣기보다 아래와 같이 인스턴스 블럭에 넣어주면 코드가 보다 간결해진다.

```

{
    count++;
    serialNo = count; } ] 인스턴스 초기화 블럭
}

Car() {
    color = "White";
    gearType = "Auto";
}

Car(String color, String gearType) {
    this.color = color;
    this.gearType = gearType;
}

```

이처럼 코드의 중복을 제거하는 것은 코드의 신뢰성을 높여 주고, 오류의 발생 가능성을 줄여 준다는 장점이 있다. 즉, 재사용성을 높이고 중복을 제거하는 것, 이것이 바로 객체지향프로그래밍이 추구하는 궁극적인 목표이다.

프로그래머는 이와 같은 객체지향언어의 요소들을 잘 이해하고 활용하여 코드의 중복을 최대한 제거하기 위해서 노력해야 한다.

▼ 예제 6-27/BlockEx.java

```

class BlockEx {
    static {
        System.out.println("static { }"); } ] 클래스 초기화 블럭
    }

    System.out.println("{ }"); } ] 인스턴스 초기화 블럭

    public BlockEx() {
        System.out.println("생성자");
    }

    public static void main(String args[]) {
        System.out.println("BlockEx be = new BlockEx(); ");
        BlockEx be = new BlockEx();

        System.out.println("BlockEx be2 = new BlockEx(); ");
        BlockEx be2 = new BlockEx();
    }
}

```

▼ 실행결과

```

static { }
BlockEx be = new BlockEx();
{ }
생성자
BlockEx be2 = new BlockEx();
{ }
생성자

```

[ebook – 샘플. 무료 공유] 자바의 정석 4판 Java 21 올컬러 2025. 7. 7 출시 seong.namkung@gmail.com

예제가 실행되면서 BlockEx가 메모리에 로딩될 때, 클래스 초기화 블럭이 가장 먼저 수행되어 'static {}'이 화면에 출력된다. 그 다음에 main메서드가 수행되어 BlockEx인스턴스가 생성되면서 인스턴스 초기화 블럭이 먼저 수행되고, 끝으로 생성자가 수행된다.

위의 실행결과에서 알 수 있듯이 클래스 초기화 블럭은 처음 메모리에 로딩될 때 한번만 수행되었지만, 인스턴스 초기화 블럭과 생성자는 인스턴스가 생성될 때 마다 수행되었다.

▼ 예제 6-28/StaticBlockEx.java

```
class StaticBlockEx {
    static int[] arr = new int[10];

    static {
        for(int i=0;i<arr.length;i++) {
            // 1과 10사이의 임의의 값을 배열 arr에 저장
            arr[i] = (int)(Math.random()*10) + 1;
        }
    }

    public static void main(String args[]) {
        for(int i=0; i<arr.length;i++)
            System.out.println("arr["+i+"] :" + arr[i]);
    }
}
```

▼ 실행결과
arr[0] :4
arr[1] :8
arr[2] :7
arr[3] :2
arr[4] :2
arr[5] :10
arr[6] :7
arr[7] :10
arr[8] :1
arr[9] :7

명시적 초기화를 통해 배열 arr을 생성하고, 클래스 초기화 블럭을 이용해서 배열의 각 요소들을 임의의 값으로 채우도록 했다.

이처럼 배열이나 예외처리가 필요한 복잡한 초기화 작업은 명시적 초기화만으로 할 수 없다. 이런 경우에 추가적으로 클래스 초기화 블럭을 사용해야 한다.

| 참고 | 인스턴스 변수의 복잡한 초기화는 생성자 또는 인스턴스 초기화 블럭을 사용한다.

6.4 멤버변수의 초기화 시기와 순서

지금까지 멤버변수를 초기화하는 방법에 대해서 알아봤다. 이제는 초기화가 수행되는 시기와 순서에 대해서 정리해보도록 하자.

클래스변수의 초기화 시점 클래스가 처음 로딩될 때 단 한번 초기화 된다.

인스턴스변수의 초기화 시점 인스턴스가 생성될 때마다 각 인스턴스별로 초기화가 이루어진다

클래스변수의 초기화 순서 기본값 → 명시적 초기화 → 클래스 초기화 블럭

인스턴스변수의 초기화 순서 기본값 → 명시적 초기화 → 인스턴스 초기화 블럭 → 생성자

프로그램 실행도중 클래스에 대한 정보가 요구될 때, 클래스는 메모리에 로딩된다. 예를 들면, 클래스 멤버를 사용했을 때, 인스턴스를 생성할 때 등이 이에 해당한다.

하지만, 해당 클래스가 이미 메모리에 로딩되어 있다면, 또다시 로딩하지 않는다. 물론 초기화도 다시 수행되지 않는다.

[ebook – 샘플. 무료 공유] 자바의 정석 4판 Java 21 올컬러 2025. 7. 7 출시 seong.namkung@gmail.com

| 참고 | 클래스의 로딩 시기는 JVM의 종류에 따라 좀 다를 수 있다. 클래스가 필요할 때 메모리에 로딩하도록 설계가 되어 있는 것도 있고, 실행효율을 높이기 위해서 사용될 클래스들을 프로그램이 시작될 때 미리 로딩하도록 되어있는 것도 있다.

```
class InitTest {
    static int cv = 1;           명시적 초기화
    int iv = 1;                  (explicit initialization)

    static {          cv = 2;      } // 클래스 초기화 블럭
    {          iv = 2;      }       // 인스턴스 초기화 블럭

    InitTest () {
        iv = 3;
    }
}
```

| 플래시동영상 | Initialization.exe에 초기화 과정에 대한 보다 자세한 설명이 있다.

위의 InitTest클래스는 클래스변수(cv)와 인스턴스변수(iv)를 각각 하나씩 가지고 있다. ‘new InitTest();’와 같이 하여 인스턴스를 생성했을 때, cv와 iv가 초기화되어가는 과정을 단계별로 자세히 살펴보도록 하자.

클래스 초기화			인스턴스 초기화				
기본값	명시적 초기화	클래스 초기화 블럭	기본값	명시적 초기화	인스턴스 초기화 블럭	생성자	
cv 0	cv 1	cv 2	cv 2	cv 2	cv 2	cv 2	
			iv 0	iv 1	iv 2	iv 3	
1	2	3	4	5	6	7	

- ▶ **클래스 변수 초기화 (1~3)** : 클래스가 처음 메모리에 로딩될 때 차례대로 수행됨.
- ▶ **인스턴스 변수 초기화(4~7)** : 인스턴스를 생성할 때 차례대로 수행됨

| 중요 | 클래스 변수는 항상 인스턴스 변수보다 항상 먼저 생성되고 초기화된다.

1. cv가 메모리(method area)에 생성되고, cv에는 int형의 기본값인 0이 cv에 저장된다.
2. 그 다음에는 명시적 초기화(int cv=1)에 의해서 cv에 1이 저장된다.
3. 마지막으로 클래스 초기화 블럭(cv=2)이 수행되어 cv에는 2가 저장된다.
4. InitTest클래스의 인스턴스가 생성되면서 iv가 메모리(heap)에 존재하게 된다.
iv 역시 int형 변수이므로 기본값 0이 저장된다.
5. 명시적 초기화에 의해서 iv에 1이 저장되고
6. 인스턴스 초기화 블럭이 수행되어 iv에 2가 저장된다.
7. 마지막으로 생성자가 수행되어 iv에는 3이 저장된다.

[ebook – 샘플. 무료 공유] 자바의 정석 4판 Java 21 올컬러 2025. 7. 7 출시 seong.namkung@gmail.com

▼ 예제 6-29/ProductEx.java

```
class Product {
    static int count = 0; // 생성된 인스턴스의 수를 저장하기 위한 변수
    int serialNo; // 인스턴스 고유의 번호

    {
        ++count;
        serialNo = count;
    }

    public Product() {} // 기본 생성자, 생략 가능
}

class ProductEx {
    public static void main(String args[]) {
        Product p1 = new Product();
        Product p2 = new Product();
        Product p3 = new Product();

        System.out.println("p1의 제품번호(serial no)는 " + p1.serialNo);
        System.out.println("p2의 제품번호(serial no)는 " + p2.serialNo);
        System.out.println("p3의 제품번호(serial no)는 " + p3.serialNo);
        System.out.println("생산된 제품의 수는 모두 "+Product.count+"개 입니다.");
    }
}
```

▼ 실행결과

p1의 제품번호 (serial no)는 1
p2의 제품번호 (serial no)는 2
p3의 제품번호 (serial no)는 3
생산된 제품의 수는 모두 3개입니다.

공장에서 제품을 생산할 때 제품마다 생산 일련번호(serial no)를 부여하는 것과 같이 Product 클래스의 인스턴스가 고유의 일련번호(serialNo)를 갖도록 하였다.

Product 클래스의 인스턴스를 생성할 때마다 인스턴스 블럭이 수행되어, 클래스 변수 count의 값을 1증가시키 다음, 그 값을 인스턴스 변수 serialNo에 저장한다.

이렇게 함으로써 새로 생성되는 인스턴스는 이전에 생성된 인스턴스보다 1이 증가된 serialNo값을 갖게 된다.

생성자가 하나 밖에 없기 때문에 인스턴스 블럭 대신, Product 클래스의 생성자를 사용해도 결과는 같지만, 코드의 의미상 모든 생성자에서 공통으로 수행되어야하는 내용이기 때문에 이스턴스 블럭을 사용하였다.

만일 count를 인스턴스 변수로 선언했다면, 인스턴스가 생성될 때마다 0으로 초기화 될 것이므로 모든 Product인스턴스의 serialNo값은 항상 1이 될 것이다.

▼ 예제 6-30/DocumentEx.java

```

class Document {
    static int count = 0;
    String name;           // 문서명(Document name)

    Document() {          // 문서 생성 시 문서명을 지정하지 않았을 때
        this("제목없음" + ++count);
    }

    Document(String name) {
        this.name = name;
        System.out.println("문서 " + this.name + "가 생성되었습니다.");
    }
}

class DocumentEx {
    public static void main(String args[]) {
        Document d1 = new Document();
        Document d2 = new Document("자바.txt");
        Document d3 = new Document();
        Document d4 = new Document();
    }
}

```

▼ 실행결과

문서 제목없음1가 생성되었습니다.
 문서 자바.txt가 생성되었습니다.
 문서 제목없음2가 생성되었습니다.
 문서 제목없음3가 생성되었습니다.

바로 이전의 일련번호 예제를 응용한 것으로, 워드프로세서나 문서 편집기에 이와 유사한 코드가 사용된다. 문서(Document)를 생성할 때, 문서의 이름을 지정하면 그 이름의 문서가 생성되지만, 문서의 이름을 지정하지 않으면 프로그램이 일정한 규칙을 적용해서 자동으로 이름을 결정한다.

예를 들면, ‘제목없음1’, ‘제목없음2’, ‘제목없음3’ ... 과 같은 식으로 문서의 이름이 결정된다. 문서의 이름은 서로 구별될 수 있어야 하기 때문이다.

| 참고 | 연습문제는 깃헙(<https://github.com/castello/javajungsuk4>)에서 PDF파일로 제공

Memo
