3주차 과제 : 자바에서 사용하는 연산자들

산술 연산자

- 산술 연산자의 종류: +, -, *, /, %
- 특히 나눗셈 연산자의 경우 두 피연산자가 정수인 경우에는 정수형 나눗셈이, 실수인 경우에는 실수형 나눗셈이 진행
- 산술연산을 하면 범위가 더 큰 쪽으로 자동 캐스팅되어 연산을 수행한다.

비트 연산자

- **비트 연산자** : 각각의 비트를 대상으로 연산을 진행하는 연산자이며 피연산자는 반드시 정수여야 함

연산자	연산자의 기능	결합 방향
&	비트 단위로 AND 연산을 한다.	->
|	비트 단위로 OR 연산을 한다	->
^	비트 단위로 XOR 연산을 한다.	->
~	피연산자의 모든 비트를 반전시켜 얻은 결과를 반환한다.	<-

- **비트 시프트 연산자** : 피연산자의 비트 열을 왼쪽 또는 오른쪽으로 이동시킨 결과를 반환하는 함수

<<	피연산자의 비트 열을 왼쪽으로 이동한다. 이동에 따른 빈 공간은 0으로 채운다.
>>	피연산자의 비트 열을 오른쪽으로 이동한다. 이동에 따른 빈 공간은 음수는 1, 양수는 0으로 채운다.
>>>	피연산자의 비트 열을 오른쪽으로 이동한다. 이동에 따른 빈 공간은 0으로 채운다.

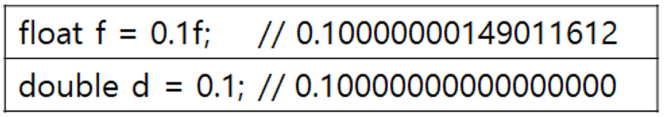
관계 연산자

- < , ≤ , > , ≥, ==, ≠ : 숫자 유형 비교를 위한 이항 연산자로, 연산자의 결과 유형은 모두 boolean

- 피연산자의 유형이 동일하지 않으면, “더 작은” 유형의 피연산자가 “더 큰” 유형으로 승격됨

```
System.out.println("7.0 == 7 : " + (7.0 == 7)); // true
```

- 하지만 float, double 의 경우는 주의해야 함
  - 실수형은 근사값으로 저장되기에 오차가 발생할 수 있기 때문이다. 0.1f는 저장할 때 2진수로 변환하는 과정에서 오차가 발생, double도 오차가 발생하지만 float 타입의 0.1f보다는 적은 오차로 저장된다.




```
float f = 0.1f;
double d = 0.1;
System.out.println(f == d); // false
```

- 피연산자 중 하나가 박스형이면 unboxing 됨
  - Unbox : 래퍼 클래스로 감싸진 데이터 → 기본 데이터 유형으로 변환

```
int primitiveInt = 5;
Integer boxedInt = new Integer(5);

System.out.print(primitiveInt == boxedInt); // true
```

논리 연산자

- 논리 연산자의 종류 : &&, ||, !
- 논리식의 결과에 따라 true 또는 false를 반환하는 연산자

```
public class Operator {
    public static void main(String[] args){
        int num1 = 0;
        int num2 = 0;
        boolean result;

        result = ((num1 += 10) < 0) && ((num2 += 10) > 0);
        System.out.println("num1: " + num1 + " num2: " + num2 + " result: " + result);
        // 실행 결과: num1: 10   num2: 0    result: false
```

```
    }
}
```

- 이는 연산의 특성 중 **Short-Circuit Evaluation(이하 SCE)**
  - **단축 평가 계산** : 첫번째 인수가 값을 결정하기에 충분하지 않은 경우에만, 두번째 인수가 평가되는 일부 프로그래밍 언어 (C, C++, Java) 등의 일부 논리연산의 계산
  - `((num1 += 10) < 0) && ((num2 += 10) > 0)` 위 연산을 진행할 때, &&의 왼편에 있는 연산이 먼저 진행되니까 num1 의 값은 10이 됨
  - 그리고 `<` 연산의 결과는 `false` 이므로 위 문장은 `false && ((num2 += 10) > 0)` 상태가 됨
  - `&&`의 오른편에 있는 연산을 진행할 차례인데, `&&`의 왼편에 `false`가 왔으니 오른편에 무엇이 오든 연산의 결과는 `false`이므로 오른편의 연산을 진행하지 않고 `&&`의 연산결과로 `false`를 반환하므로 결국 num2의 값은 변하지 않음
- &&의 왼쪽 피연산자가 `false`이면, 오른쪽 피연산자는 확인하지 않는다.
- ||의 왼쪽 피연산자가 `true`이면, 오른쪽 피연산자는 확인하지 않는다.

instanceof

- 자바의 대표적인 type introspection (실행 시간에 객체의 타입이나 속성을 검사하는 프로그램의 능력을 뜻함)
- 객체 타입을 확인하는데 사용하는 연산자
- 객체가 특정 클래스/인터페이스 유형인지 여부를 확인하고, true / false 로 알려줌
- `null` 은 어떤 것의 instance 도 아님

```
class A {}

class B extends A {}

public static void main(String[] args) {
    A a = new A();
    B b = new B();

    System.out.println(a instanceof A); // true
    System.out.println(b instanceof A); // true
    System.out.println(a instanceof B); // false
    System.out.println(b instanceof B); // true
    System.out.println(null instanceof Object); // false
}
```

assignment(=) operator

- 할당 연산자 : 변수에 값을 할당(저장) 하는데 사용
- 단순 할당 연산자 : '='
- 복합 할당 연산자 : '=' 를 제외한 나머지

대입 연산자	설명
=	왼쪽의 피연산자에 오른쪽의 피연산자를 대입함.
+=	왼쪽의 피연산자에 오른쪽의 피연산자를 더한 후, 그 결괏값을 왼쪽의 피연산자에 대입함.
-=	왼쪽의 피연산자에서 오른쪽의 피연산자를 뺀 후, 그 결괏값을 왼쪽의 피연산자에 대입함.
*=	왼쪽의 피연산자에 오른쪽의 피연산자를 곱한 후, 그 결괏값을 왼쪽의 피연산자에 대입함.
/=	왼쪽의 피연산자를 오른쪽의 피연산자로 나눈 후, 그 결괏값을 왼쪽의 피연산자에 대입함.
%=	왼쪽의 피연산자를 오른쪽의 피연산자로 나눈 후, 그 나머지를 왼쪽의 피연산자에 대입함.
&=	왼쪽의 피연산자를 오른쪽의 피연산자와 비트 AND 연산한 후, 그 결괏값을 왼쪽의 피연산자에 대입함.
|=	왼쪽의 피연산자를 오른쪽의 피연산자와 비트 OR 연산한 후, 그 결괏값을 왼쪽의 피연산자에 대입함.
^=	왼쪽의 피연산자를 오른쪽의 피연산자와 비트 XOR 연산한 후, 그 결괏값을 왼쪽의 피연산자에 대입함.
<<=	왼쪽의 피연산자를 오른쪽의 피연산자만큼 왼쪽 시프트한 후, 그 결괏값을 왼쪽의 피연산자에 대입함.
>>=	왼쪽의 피연산자를 오른쪽의 피연산자만큼 부호를 유지하며 오른쪽 시프트한 후, 그 결괏값을 왼쪽의 피연산자에 대입함.
>>>=	왼쪽의 피연산자를 오른쪽의 피연산자만큼 부호에 상관없이 오른쪽 시프트한 후, 그 결괏값을 왼쪽의 피연산자에 대입함.

- 'a = b = c'의 경우, 오른쪽에서 왼쪽으로 수행. 'a = (b = c)'.

화살표(->) 연산자

- 람다 연산자는 람다식을 도입하는 데 사용되는 연산자
- `(인자, ...) -> { /* 메소드 바디 */ }`

```
// 두 개의 숫자 중 큰 수 출력하기
@FunctionalInterface
public interface  MyNumber {
    int getMax(int num1, int num2);
}

public static void main(String[] args) {
    MyNumber max = (x, y) -> (x >= y) ? x : y;
    System.out.println(max.getMax(10, 30));
}
```

3항 연산자

- 조건식 ? 반환값1 : 반환값2

연산자 우선 순위

연산자	연산 방향	우선 순위
증감(++, --), 부호(+, -), 비트(~), 논리(!)	←	높음
산술(*, /, %)	→	
산술(+, -)	→	
쉬프트(<<, >>, >>>)	→	
비교(<, >, <=, >=, instanceof)	→	
비교(==, !=)	→	
논리(&)	→	
논리(^)	→	
논리(|)	→	
논리(&&)	→	
논리(||)	→	
조건(?:)	→	
대입(=, +=, -=, *=, /=, %=, &=, ^=, |=, <<=, >>=, >>>=)	←	낮음

Java 13. switch 연산자

- JDK 12,13 의 preview 로 공개됐고 14버전에 정식 추가됨
- 기존의 case … : 를 대체하는 case … → 를 사용해 불필요한 코드 개선 가능

```
// 기존
switch (day) {
    case MONDAY:
    case FRIDAY:
    case SUNDAY:
        System.out.println(6);
        break;
    case TUESDAY:
        System.out.println(7);
        break;
    case THURSDAY:
    case SATURDAY:
        System.out.println(8);
        break;
    case WEDNESDAY:
        System.out.println(9);
        break;
}

// case L1, L2 ... -> 에 N개의 label 동시 작성 가능
```

```
switch (day) {
    case MONDAY, FRIDAY, SUNDAY -> System.out.println(6);
    case TUESDAY                -> System.out.println(7);
    case THURSDAY, SATURDAY     -> System.out.println(8);
    case WEDNESDAY              -> System.out.println(9);
}
```

- 자바 13부터는 break문 대신 `yield` 구문 사용 가능
    - yield 는 함수의 return 과 비슷하다고 할 수 있는데, 해당 switch 블록의 특정 값을 switch 의 결과값으로 반환하는 것

```
Day day = Day.WEDNESDAY;
int numLetters = switch (day) {
    case MONDAY:
    case FRIDAY:
    case SUNDAY:
        System.out.println(6);
        yield 6;
    case TUESDAY:
        System.out.println(7);
        yield 7;
    case THURSDAY:
    case SATURDAY:
        System.out.println(8);
        yield 8;
    case WEDNESDAY:
        System.out.println(9);
        yield 9;
    default:
        throw new IllegalStateException("Invalid day: " + day);
};
System.out.println(numLetters);
```

- 화살표 연산자를 사용할 경우에는 yield 가 빠지면, 컴파일 에러가 발생하여 미리 알 수 있기 때문에 화살표 연산자를 사용할 것을 권장함

```
Day day = Day.WEDNESDAY;
int numLetters = switch (day) {
    case MONDAY, FRIDAY, SUNDAY -> {
        System.out.println(6);
        yield 6;
    }
    case TUESDAY -> {
        System.out.println(7);
        yield 7;
    }
    case THURSDAY, SATURDAY -> {
        System.out.println(8);
        yield 8;
    }
```

```
    case WEDNESDAY -> {
        System.out.println(9);
        yield 9;
    }
    default -> {
        throw new IllegalStateException("Invalid day: " + day);
    }
};
```

- yield 는 항상 switch 블록 내부에서만 사용됨

```
public void test(Day day) {
        int cnt = switch (day) {
            case MONDAY -> 0;
            case TUESDAY -> yield 1; // 에러. yield는 block 안에서만 유효
            case WEDNESDAY -> { yield 2;} // ok
            default -> 0;
        }
}
```

advanced

자바 8 이후로 변경 사항 정리

▼ Java 8

- `람다 표현식`
  - 메소드로 전달할 수 있는 익명함수를 단순한 문법으로 표기한 것

```
// 익명 클래스로 Runnable 을 구현
Thread thread = new Thread(new Runnable() {
    @Override
    public void run() {
        System.out.println("Start to new thread!");
    }
});

thread.start();


// 람다 표현식으로 단순하게 표현
Thread thread = new Thread(() -> System.out.println("Start to new thread!"));
```

- `함수형 인터페이스`
  - 단 하나의 추상메소드를 갖는 인터페이스
  - 컴파일러가 람다의 타입을 추론할 수 있도록 정보를 제공하는 역할

- **Supplier** : 매개변수를 받지 않고 특정 타입의 결과를 리턴하는 “공급자”

```
@FunctionalInterface
public interface Supplier<T> {
  T get();
}
```

이를 활용하면 값을 그냥 전달하는 게 아니라, Supplier 를 이용해서 단순히 값을 가져오기 전에 **중간에 로직을 추가해서 전달 가능**

```
public double square(double d) {
  return Math.pow(d, 2);
}

double squre = square(3); // 9.0
```

```
public double squareLazy(Supplier<Double> lazyValue) {
  return Math.pow(lazyValue.get(), 2);
}

Supplier<Double> lazyValue = () -> {
  // 1000ms 딜레이를 줌
  Uninterruptibles.sleepUninterruptibly(1000, TimeUnit.MILLISECONDS);
  return 3d;
};

Double squre = squareLazy(lazyValue); // 9.0
```

- **Consumer** : 매개변수를 받고, 리턴하지는 않고 처리하는 “소비자”

```
default void forEach(Consumer<? super T> action) { ... }

List<String> names = Arrays.asList("John", "Freddy", "Samuel");
names.forEach(name -> System.out.println("Hello, " + name));
```

- **Predicates** : 특정 타입의 매개변수를 받아 boolean 값을 리턴하는 함수형 인터페이스

```
// Stream 클래스의 filter 메소드
Stream<T> filter(Predicate<? super T> predicate);

List<String> names = Arrays.asList("Angela", "Aaron", "Bob", "Claire", "David");

List<String> namesWithA = names.stream()
  .filter(name -> name.startsWith("A"))
```

```
    .collect(Collectors.toList());

System.out.println(namesWithA); // [Angela, Aaron]
```

- `Stream API`
  - 컬렉션을 편리하게 처리하는 방법 제공하는 API
  - "책들 중 니체가 작성한 책의 ISBN 정보가 필요합니다. 정렬은 책 이름을 기준으로 해주세요."

```
books.sort(Comparator.comparing(Book::getName));

List<String> booksWrittenByNietzsche = new ArrayList<>();
for (Book book : books) {
    if (book.getAuthor().equals("Friedrich Nietzsche")) {
        booksWrittenByNietzsche.add(book.getIsbn());
    }
}
```

```
List<String> booksWrittenByNietzsche =
            books.stream()
                .filter(book -> book.getAuthor().equals("Friedrich Nietzsche"))
                .sorted(Comparator.comparing(Book::getName))
                .map(Book::getIsbn)
                .collect(Collectors.toList());
```

  - 스트림의 기능을 거치면, 연산 이전의 값을 저장하지 않고 연산된 값만 새롭게 반환
    → 재사용이 필요하다면 변수에 저장해두고 사용해야 함
  - 중간 연산과 종료 연산으로 구분됨
    - 중간 연산 : Stream<T> 형태의 스트림 반환
    - 종료 연산 : 그렇지 않으면 종료 연산

- `Optional`

▼ Java 9

- **불변 컬렉션 생성 메소드**
  - 자바 8까지는 불변 리스트를 만들기 위해, 가변 리스트를 먼저 만든 후에 Collections.unmodifiableList() 를 사용하여 불변으로 변환시켜줘야 했음

```
List<String> fruits = new ArrayList<>();

fruits.add("Apple");
fruits.add("Banana");
fruits = Collections.unmodifiableList(fruits);

fruits.add("Lemon"); // UnsupportedOperationException
```

    - List.of() / Set.of() / Map.of() 라는 새로운 팩토리 메소드가 추가되어, 담고자 하는 값을 바로 전달해 불변 컬렉션 만들 수 있음

- try-with-resources 문이 개선됨
    - 자원 할당을 try 밖에서 하면 그 변수를 가지고 바로 try문 안에서 사용할 수는 없어서 따로 try안에서 새로운 변수를 선언하여 사용해야 했다
    - final or effectively final (초기화된 이후 절대 바뀌지 않는 변수) 이 적용되어 변수를 바로 참조할 수 있음

```
public void testOne() throws FileNotFoundException {
    BufferedReader brs = new BufferedReader(new FileReader("C:\\number.txt"));
    try (BufferedReader br = brs) {
          String str;
          while ((str = br.readLine()) != null) {
            System.out.println(str);
          }
    } catch (IOException e) {
          e.printStackTrace();
    }
}

public void testOne() throws FileNotFoundException {
    BufferedReader br = new BufferedReader(new FileReader("C:\\number.txt"));
    try (br) {
          String str;
          while ((str = br.readLine()) != null) {
            System.out.println(str);
          }
    } catch (IOException e) {
          e.printStackTrace();
    }
}
```

▼ Java 10

- **Local-Variable Type Inference**
    - var 를 사용하여 생성할 변수 타입 추론이 가능

→ 회사코드 언급해보기

- **Optional.orElseThrow() Method**
  - get() 메서드의 대안으로써 본질적으로 동일한 새로운 메서드. Optional이 값을 보유하면 반환된다. 그렇지 않으면 NoSuchElementException이 발생한다.
- 루트 인증서
  - cacerts 라고 불리는 JDK의 keystore에는 보안 프로토콜에 사용되는 루트 인증서 세트를 넣을 수 있는 곳이 존재했지만, 비어있었고 사용자가 직접 문서화된 루트 인증서 세트를 채워야 했다. **Oracle은 Java 10** 을 통해 OpenJDK 빌드를 개발자에게 더 매력적으로 만들고 이러한 빌드와 Oracle JDK 빌드 간의 차이를 줄이기 위해 Oracle의 Java SE 루트 CA 프로그램에서 **루트 인증서** 를 **오픈 소스로 제공**

▼ **Java 11**

- lambda 에 대한 지역 변수 구문
  - 람다 매개 변수 에서 지역 변수 구문 (*var* 키워드) 사용에 대한 지원 추가
    이 기능을 사용하여 타입 어노테이션 정의와 같이 지역 변수에 수정자 적용 가능

```
List<String> sampleList = Arrays.asList("Java", "Kotlin");
String resultString = sampleList.stream()
  .map((@Nonnull var x) -> x.toUpperCase())
  .collect(Collectors.joining(", "));
assertThat(resultString).isEqualTo("JAVA, KOTLIN");
```

- java 파일 실행 방법
  - 이 버전의 주요 변경 사항은 **더 이상 명시적으로 *javac로* Java 소스 파일을 컴파일 할 필요가 없다는 것**

▼ **Java 15**

- Text block 기능 추가
  - 기존 JSON 문자열을 직접 생성할 때 이스케이프 처리로 가독성이 떨어지던 문제 개선

```
// Before
private static void oldStyle() {
    String text = "{\n" +
                  "  \"name\": \"John Doe\",\n" +
                  "  \"age\": 45,\n" +
                  "  \"address\": \"Doe Street, 23, Java Town\"\n" +
                  "}";
    System.out.println(text);
}
```

```
// After
private static void jsonBlock() {
    String text = """
            {
              "name": "John Doe",
              "age": 45,
              "address": "Doe Street, 23, Java Town"
            }
            """;
    System.out.println(text);
}
```

▼ Java 16

- Record 가 정식적으로 Java 언어에 추가됨
- Pattern Matching

▼ Java 17

- Instance of 변경
  - instanceof 사용 시 객체 instance 확인 → 지역 변수 선언하여 객체 Casting 후 사용하는 과정을 단축

```
// 기존 instanceof 사용 방식 #1
if (shape instanceof Rectangle) {
    Rectangle r = (Rectangle) shape; // Casting
    return 2 * r.length() + 2 * r.width();
} else if (shape instanceof Circle) {
    Circle c = (Circle) shape;       // Casting
    return 2 * c.radius() * Math.PI;
}

// 새로운 instanceof 사용 방식 #1
if (shape instanceof Rectangle r) {
    return 2 * r.length() + 2 * r.width();
} else if (shape instanceof Circle c) {
    return 2 * c.radius() * Math.PI;
}
----------------------------------------------------------------------
// 기존 instanceof 사용 방식 #2
public boolean equals(Object o) {
    return (o instanceof CaseInsensitiveString) &&
        ((CaseInsensitiveString) o).s.equalsIgnoreCase(s);
}
// 새로운 instanceof 사용 방식 #2
public boolean equals(Object o) {
    return (o instanceof CaseInsensitiveString cis) &&
        cis.s.equalsIgnoreCase(s);
}
```

    - 변수의 scope 주의 필요 : 기본적으로 지역변수여서, local variable scope 을 따름

```
if (shape instanceof Rectangle r) {
  // r 사용가능
  // c 사용불가능
  return 2 * r.length() + 2 * r.width();
} else if (shape instanceof Circle c) {
  // r 사용불가능
  // c 사용가능
  return 2 * c.radius() * Math.PI;
}
```

- Record Class 추가