CHAPTER 04

상황에 따라 다르게 실행하자 제어문/루프

변수가 처리해야 하는 데이터를 의미한다면 제어문은 데이터를 처리하기 위한 로직을 작성할 때 사용합니다. 루프를 이용하면 반복 작업을 간단하게 작성할 수 있습니다.

프로그램은 결국 사람이 오랜 시간 걸려서 해야 하는 일을 컴퓨터가 대신할 수 있도록 하는 작업이고, 사람이 가장 많이 하는 것이 바로 '상황에 따른 판단'입니다. 프로그래밍에서는 이 판단을 제어문이라는 것이 하도록 합니다. 변수가 데이터를 의미한다면 제어문은 데이터를 어떻게 처리해야 하는지 결정하는 부분이라고 할 수 있겠습니다.

1 Java에서 제어문이란?

"엄마가 좋은가? 아빠가 좋은가?", "짬뽕이냐? 자장이냐?", "이 길로 갈까? 저 길로 갈까?" 등 결국 이런 구문의 공통점은 한 번의 판단으로 그 이후의 로직이 달라진다는 겁니다. 프로그래밍에서 제어문을 사용하는 상황은 다음과 같습니다.

- 만일 ~이라면의 상황에서 제어문을 활용합니다.
- 여러 가지의 선택의 상황에서 제어문을 활용합니다.

1.1 모든 제어문의 실행 여부는 true/false로 나오는 결과

Java 언어에서 제어문을 활용할 때에는 반드시 boolean으로 처리되어야만 합니다.

많은 곳에서 간과되고 있는 원칙입니다만, Java에서 제어문의 분기 기준은 모두 true/false 값을 가지는 boolean 타입의 데이터라는 것을 기억해야 합니다. 간단하게 미성년자인가 아닌가와 같은 문제라든가 프로그램에서 서버에 접근할 권한 유무와 같은 모든 판단은 true/false로 표현할 수 있다는 이야기입니다. 참고로, C 언어에서는 0과 1을 제어문의 판단 기준으로 활용할 수 있습니다만 Java에서는 반드시 true/false로 표현될 수 있어야 합니다(Java는 이런 면에서 보면 상당히 깐깐한 언어입니다). 프로그래밍이란 기본적으로 확실한 상황만을 다룹니다

프로그램적으로 말하자면 '당신이 가진 카드가 0이라면'이라든가 '당신의 메뉴는 자장면입니까?'와 같이 '예' 혹은 '아니오'로 정확하게 boolean 타입의 데이터로 표현될 수 있는 상황만이 프로그래밍에서 적합하다는 얘기입니다. 간단한 코드를 보도록 합니다.

예제

```
public class OddEven {
	public static void main(String[] args) {
		int value = 123332;
		int odd = value % 3;
	}
}
```

위의 코드에서는 value라는 변수를 선언해 주었고, odd라는 변수는 value를 3으로 나눈 나머지(%)를 의미합니다. 어떤 정수를 3으로 나누면 나머지의 값은 0, 1, 2밖에 존재하지 않습니다. 이제 이 나머지 값을 가지고 다른 메시지를 화면에 출력할 때 어떻게 사용하는지 알아봅니다.

예제

```
public class OddCheck {
	public static void main(String[] args) {
		int value = 123332;
		int odd = value % 3;
		if(odd == 0){
			System.out.println("나머지는 0");
		}else if(odd == 1){
			System.out.println("나머지는 1");
		}else{
			System.out.println("그렇다면 나머지는 2");
		}//end if
```

```
            }
}
```

```
그렇다면 나머지는 2
```

문법적인 내용을 제외하면 제어문은 어렵지는 않습니다. 우리가 실제로 많이 하는 작업이기도 하고, 순서도나 게임에서처럼 어떤 데이터의 결과에 따라 추적해가는 스타일이라고 할 수 있습니다. 다만, 결과는 반드시 true/false로 만들어져야만 한다는 사실만 기억하시기 바랍니다.

■ **제어문에서 가장 많이 쓰이는 연산자 '==(동등 연산자)'**

> 제어문에서 많이 쓰이는 연산자인 비교 연산자(〉, 〈, 〉=, 〈=, ==, !=)를 이용해서 boolean 결괏값을 구합니다. 그중에서도 '==(같다)', '!=(다르다)' 연산자를 가장 많이 사용합니다.

제어문에서 가장 많이 사용되는 연산자는 바로 == 연산자(동등 연산자)입니다. ==은 우리 말로 '같은가?'로 생각해 주시면 됩니다. 모든 연산자는 연산의 결과를 반환한다고 말씀드렸습니다. ==연산자는 비교하는 데이터 양쪽이 동일한 경우(엄밀하게 말하자면 그 안에 들어간 내용물이 같은 경우)에는 true를 반환하고, 아닌 경우에는 false를 반환해줍니다. 따라서 우리가 코드를 작성할 때는 그 결괏값을 이용해서 주로 제어문을 처리하곤 합니다. == 동등 연산자와 같이 != 연산자도 같이 사용되므로 두 연산자를 같이 알아 두시는 것이 좋습니다.

1.2 { }는 제어문과 변수의 영향력의 범위를 결정합니다.

Java 언어에서 {}은 영향력의 범위를 결정합니다. 영향을 받는 대상은 다음과 같습니다. 특히 변수와 제어문은 {}로 영향력의 경계를 명시하게 됩니다.

예제

```
public class BlockEx {
	public static void main(String[] args) {
		int a = 100;
		//만일 a가 10이라면
		if(a == 10 ){ // 제어문 영향력 시작
			System.out.println("AAAAAAAAAAA");
			System.out.println("BBBBBBBBBBB");
			System.out.println("CCCCCCCCCCC");
		}// 제어문 영향력 끝
	}
}
```

(출력 결과 없음)

Java 언어에서 {}의 위력은 절대적입니다. 아주 완벽한 울타리입니다. 위의 코드를 보면 {}의 영향력으로 묶인 코드가 실행되었습니다. 기존의 코드에서 {}를 제거하면 결과는 완전히 달라집니다.

예제

```
int a = 100;

if(a == 10)
	System.out.println("AAAAAAAAAAA");
	System.out.println("BBBBBBBBBBB");
	System.out.println("CCCCCCCCCCC");
```

```
BBBBBBBBBBB
CCCCCCCCCCC
```

이전의 코드는 a가 10이라면 실행되는 코드가 3라인이지만 변수 a의 값이 100이고 {}로 묶여 있기 때문에 아무런 결과도 실행되지 않습니다. 반면에 현재의 코드는 {}가 없는데, 이때는 무조건 아래 한 라인만이 if 구문의 영향을 받게 됩니다.

2 모든 제어문의 기초 if: 만일 ~ 한다면

모든 프로그래밍에서 판단으로 가장 많이 쓰이는 분기문은 if 구문입니다. if 구문은 어떠한 조건이 맞는다면 실행되는 방식입니다. if 구문은 실행하는 방식에 따라 세 가지로 구분됩니다.

• 단순히 프로그램 흐름의 중간에서 실행 여부를 판단하는 단순 if
• go? stop?처럼 하나의 조건으로 true와 false에 따라 분기하는(갈라지는) 경우
• 여러 개의 조건 중 맞는 하나의 조건을 찾아서 분기하는 경우

2.1 마음에 들면 이것도 실행해주세요: 단순 if

가장 단순한 if 구문의 형태는 별다른 내용 없이 다음과 같이 구성됩니다.

단순한 if는 로직의 흐름 속에서 하나의 추가적인 부분을 옵션으로 지정할 때 사용합니다.
• '~를 하는 데 있어서 이 부분도 실행할까?'에 해당하는 부분은 단순 if

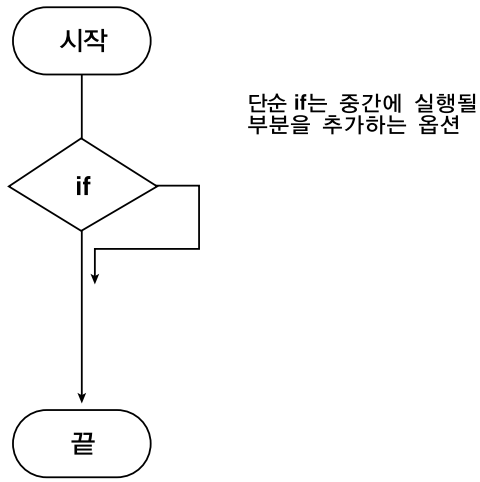

그림 1 if 제어문은 옵션

단순하게 if를 사용하는 경우는 로직이 실행되는 데 있어서 하나의 옵션으로 "맞으면 이 부분도 실행한다."라는 방식입니다. 주의할 것은 이때의 판단 조건은 반드시 true/false 값 중의 하나여야 한다는 겁니다. 이 경우에 주로 우리가 사용하는 연산자가 등가 연산(같은가 다른가를 구분하는 연산 ==, !=)이나 부등호 연산(>, >=, <, <=) 혹은 논리 연산(&&, ||) 같은 구분이 주로 사용됩니다(거듭 말하지만 모든 제어문의 판단 기준은 true/false입니다). 단순 비교는 그저 yes인가만을 비교하기 때문에 실제로 많이 쓰이지는 않습니다.

예제 | 단순 if 구문 테스트

```
public class IfEx1 {
	public static void main(String[] args) {
		int a = 10;
		int odd = a%2;
		if( odd == 0){
			System.out.println("짝수" + a);
		}
	}
}
```

```
짝수 10
```

== 연산자를 유심히 보시면 됩니다. a를 먼저 2로 나눈 나머지 연산을 먼저 수행하고, 그 값과 0이 같은 값인지를 비교합니다. 그런데 짝수가 아닐 경우는 어떻게 해야 하나요?

2.2 마음에 들면 go? 아니면 stop?: if ~ else

if ~ else는 고스톱이 딱 좋은 예입니다. 고를 외치느냐? 여기서 그만 멈추고 스톱하느냐? 판단한 만한 상황은 하나이고 이 결과를 true인지, false인지에 따라서 분기하는 방식입니다. 판단 대상도 하나이고 판단 조건도 하나인 경우입니다.

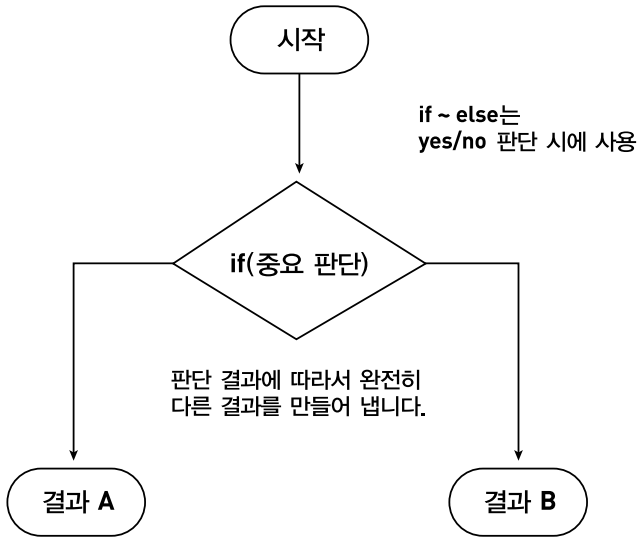

그림 2 if ~ else는 양자택일

if ~ else 구문은 양자택일의 상황에서 주로 사용됩니다.

```
if(판단 조건) {
        판단 조건이 true일 때 실행될 부분
}else {
        판단 조건이 false일 때 실행될 부분
}
```

예제

```
public class IfElseEx {
        public static void main(String[] args){
                int a = 10;
                int odd = a % 2;
                if( odd == 0){
                        System.out.println("짝수" + a);
                }else{
                        System.out.println("홀수" + a);
                }
        }
}
```

```
짝수 10
```

if 조건에서 true라고 판단되지 않을 때에는 else 부분이 실행되는 방식입니다. 좀 더 쉽게 생각해 볼만한 예로는 역시나 자장면과 짬뽕이 아닐까 싶네요. 여러 명이 모여서 "자장면 손들어봐, 안 들면 짬뽕을 주문하지 뭐..."

2.3 여러 상황 중 하나만 if ~ else if

if ~ else if는 판단 대상 하나에 대해서 여러 가지 조건을 대입해서 맞는 상황을 찾아냅니다. 즉 어떤 조건이 여러 가지 경우의 수를 발생할 수 있는 상황에서 사용됩니다. 여주인공은 한 명이고, 남자들은 여러 명인 상황을 생각해보시면 됩니다. 돈 많은 남자, 착한 남자, 잘 생긴 남자 등 판단 대상은 여성 한 명이고, 판단 조건은 돈, 성격, 외모와 같이 여러 가지로 판단할 수 있는 경우입니다. 그림으로 표현하기는 좀 어색하지만, 다음과 같이 표현할 수 있을 듯합니다.

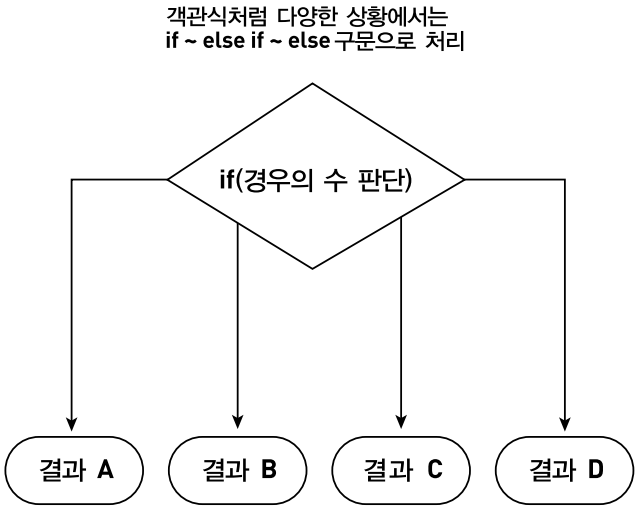

그림 1 if ~ else if 구조는 객관식

```
'if ~ else if ~ else' 구문은 다양한 선택의 상황에서 주로 사용됩니다.
if(판단 조건 A){
        판단 조건 A가 true일 때 실행될 부분
}else if(판단 조건 B){
        판단 조건 A가 false이고, 판단조건 B가 true일 때
}else if(판단 조건 C){
        판단 조건 A가 false이고, 판단 조건 B도 false이고, 판단 조건 C가 true일 때
}else{
        판단 조건 A, B, C 모두 false일 때: 기본 실행 코드(작성하고 싶지 않다면 없어도 됩니다.)
}
```

if ~ else if는 판단하는 조건을 세분화해서 처리할 수 있습니다. 시험 성적을 생각하시면 좋습니다. 만일 90점 이상이면 'A', 80점 이상이면 'B', 70점 이상이면 'C', 60점 이상이면 'D', 50점 이상일 때 'E', 그 이하는 F 학점을 받는다고 생각해 보겠습니다.

코드

```
if(grade >= 90){
        System.out.println("당신의 학점은 A");
}else if(grade >= 80){
        System.out.println("당신의 학점은 B");
}else if(grade >= 70){
        ...
}
```

이처럼 else if는 위의 조건을 만족하지 않았을 때 다음 조건에 맞는지 차곡차곡 검사하면서 실행하게 합니다. 만일 마지막의 else if까지 원하는 조건에 맞지 않는다면 최후의 else로 넘어가게 됩니다.

예제

```
public class IfElesIfEx {
        public static void main(String[] args) {
                int grade = 65;

                if(grade >= 90){
                        System.out.println("당신의 학점은 A");
                }else if(grade >= 80){
                        System.out.println("당신의 학점은 B");
                }else if(grade >= 70){
                        System.out.println("당신의 학점은 C");
                }else if(grade >= 60){
                        System.out.println("당신의 학점은 D");
                }else if(grade >= 50){
                        System.out.println("당신의 학점은 E");
                }
        }
}
```

```
당신의 학점은 D
```

2.4 if ~ else if에서 이런 건 주의: 만족하는 것 하나만 실행됩니다.

'if ~ else if ~ else'를 사용할 때에는 반드시 다음 규칙을 따르도록 합니다.
- 갈수록 범위가 넓어지도록 작성합니다.
- 마지막의 else는 최후의 수단에 해당하는 코드를 넣습니다.
- 공통으로 실행될 코드는 반드시 if 구문 바깥쪽에 작성합니다.
- 참고로 부등호를 사용하는 경우에는 〉 방향으로 작성해 주면 조금 더 가독성을 높여 줄 수 있습니다.

가끔 if ~ else if로 코딩해 놓은 어이없는 결과가 발생하는 코드들이 종종 있습니다. 정작 코드를 만든 본인은 정확하게 만들었다고 생각하는데 실제로는 원하는 데로 결과가 나오지 않는 겁니다. 우선 아래의 코드를 보시겠습니다. 아래와 같은 코드는 부분으로 나누어서 보면 바로 잘못된 부분이 보이지만 몇천 라인의 코드라면 잘못된 부분을 찾기 어려운 코드입니다.

코드

```
int grade = 93;

if(grade > 50){
        System.out.println("턱걸이로 합격");
}else if(grade > 70){
        System.out.println("안정권으로 합격");
}else if(grade > 90){
        System.out.println("우수한 성적으로 합격");
}
```

위 코드의 실행 결과는 어떨까요? 점수가 90 이상이 되니까 당연히 '우수한 성적으로 합격'이라는 메시지가 화면에 출력된다고 생각하실 수 있겠지만, 잠시만 코드를 유심히 봐 주시면, 'if (grade 〉 50)'에도 결과가 true인 조건임을 알 수 있습니다. 사실 위의 코드를 실행하면 결과는 '턱걸이로 합격'이라는 결과가 나옵니다. if ~ else if는 첫 번째 조건을 만족하지 않았을 때 다음 조건, 그 다음 조건과 같은 순서로 진행되기 때문에 조건을 어디서 만족하는가가 가장 중요합니다.

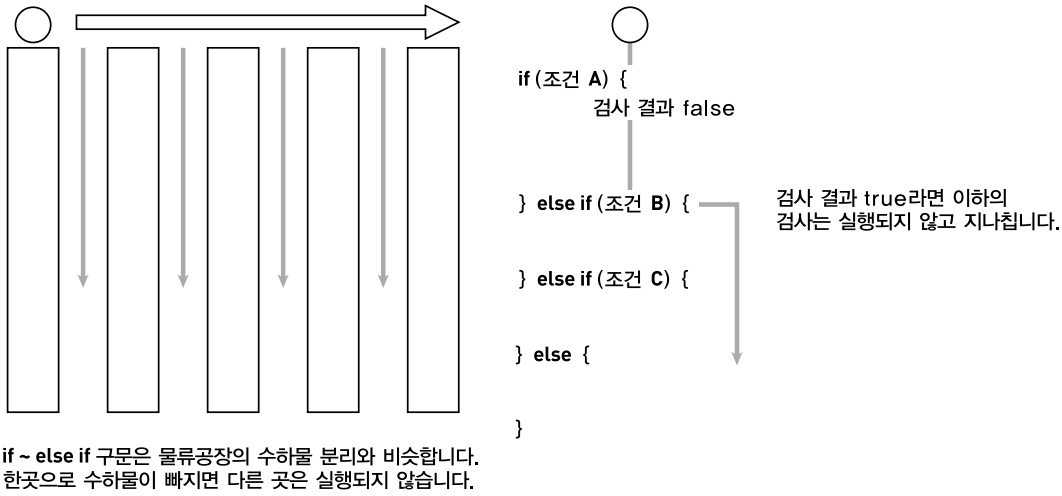

그림 4 작은 if ~ else if에서 주의점

2.5 if에서 { }의 의미

이전에도 얘기했듯이 if에서 {}의 의미는 영향력을 행사하는 부분을 의미합니다. {}가 많은 라인을 차지할수록 많은 코드가 실행된다는 의미입니다.

참고

들여쓰기 습관과 주석 처리

아마도 프로그래밍을 처음 하는 사람과 어느 정도 프로그래밍을 해 본 사람들이 가장 큰 차이를 보이는 부분이 들여쓰기 같은 사소한 부분일 것입니다. 들여쓰기는 {}를 이용하거나 하지 않거나 상관없이 중요한 습관입니다. 결국에 프로그래밍이라는 것은 개발자와 컴퓨터 사이의 커뮤니케이션인 동시에 개발자와 다른 개발자들 사이에 의사소통 수단입니다. 모든 언어를 사용할 때 들여쓰기 습관을 반드시 가지도록 하세요. 주석 처리는 주로 if, for, while 등과 같이 블록을 많이 쓰는 곳에서 들여쓰기와 함께 현재 종료한 } 가 어느 부분인지를 명확하게 하려고 사용합니다. 다음 코드를 봅시다.

예제

```
16:                }else if(grade >= 50){
17:                        System.out.println("당신의 학점은 E");
18:                } // end if
19:
20:        } // end main
21:  } // end class
```

}가 3번이 나오는 데 들여쓰기와 주석 처리가 되어 있기 때문에 어떤 구문의 블록이 종료되는지를 정확하게 알아볼 수 있습니다.

■ **if 구문의 { }와 변수: 변수는 { }라는 경계선을 갖습니다.**

if 구문에서 주의해야 하는 또 다른 내용 중의 하나는 if 구문 안에서 선언되는 변수입니다. 다음의 코드는 컴파일 에러가 발생하는 코드입니다. 원인이 무엇일까요?

예제 | if 구문에서 변수의 선언 시에는 { }를 조심하세요.

```
public class IfVariables {
        public static void main(String[] args) {
                int x = 10;
                int odd = x % 2;
                if(odd == 0){
                        int value = 100; // { }안에서 선언된 변수
                }else{
                        value = 200; // 컴파일 에러
                }
        }// end main
}
```

위의 코드는 'value = 200;' 구문에서 컴파일 에러가 발생합니다. 코드를 유심히 보면 value라는 변수가 최초로 선언된 부분이 if 구문 안쪽인 것을 보실 수 있습니다. 즉, 위의 코드에서 value라는 변수는 if {}안쪽에서만 사용될 수 있는 변수입니다. 변수의 생명은 주로 {}와 관련이 있기 때문에 어떤 변수가 특정 {}안쪽에 선언되면 외부에서는 쉽게 사용할 수 없게 됩니다.

2.6 간단한 if ~ else 테스트: 홀짝 만들기

주사위를 아실 겁니다. 주사위는 임의의 1에서 6에 해당하는 숫자를 발생시켜 주는 도구입니다. 여러분이 이번에 만들 프로그램은 바로 그러한 주사위처럼 어떤 숫자를 만들어서 홀수인지 짝수인지를 판단하는 겁니다. 우선 100 미만의 어떤 숫자를 만들어 보도록 합니다.

예제

```
public class OddEven {
	public static void main(String[] args) {
		int value = (int)(Math.random() * 100);
		System.out.println(value);
	}
}
```

Math.random( )은 0에서 0.9999...까지의 double에 해당하는 숫자를 발생시키고, 이것에 100을 곱했으니 0에서 99.999...의 숫자가 만들어집니다. 이제 이 숫자를 2로 나누면 나머지의 값은 0이나 1이 됩니다.

코드

```
int odd = value%2;
```

이제 남은 것은 odd라는 변수의 값이 0인지 1인지를 판별하는 작업을 진행해주면 됩니다.

예제 | 홀짝 프로그램

```
public class OddEven {
	public static void main(String[] args) {
		int value = (int)(Math.random() * 100);
		System.out.println(value);
		int odd = value%2;
		if(odd == 1){
			System.out.println("홀수네요.");
		}else{
			System.out.println("짝수네요.");
		}//end if
	}
}
```

```
31
홀수네요.
```

물론 실행한 결과는 매번 달라질 수 있습니다. 간단한 변수와 if ~ else만으로도 꽤 긴 프로그램을 작성할 수 있습니다. 가장 쉽게 생각해볼 수 있는 예제가 사칙연산을 하는 계산기 프로그램입니다.

예제 | if를 이용하는 계산 프로그램

```
public class OperTest {
	public static void main(String[] args) {

		float a = 10;
		float b = 3;
		float result;
		char oper = '+';

		if(oper == '+'){
			result = a + b;
		}else if(oper == '-'){
			result = a - b;
		}else if(oper == '*'){
			result = a * b;
		}else {
			result = a/b;
		}
		System.out.println("연산 결과 :" + result);
	}
}
```

이번에는 if ~ else를 이용하되 char를 연산의 기호로 삼아서 작성한 코드입니다. result라는 변수를 선언하고 모든 결과를 result 값을 변경하는 데에만 사용했습니다. 마지막에서 한 번만 결과를 출력해 주고 있습니다.

3 if ~ else를 간단하게 쓰려는 시도 switch

switch는 if ~ else if의 변형입니다. 솔직히 말하자면 switch 구문으로 작성해도 Java의 내부적

으로는 if ~ else if로 다시 변형되어 실행됩니다. 속도로 따지자면 if ~ else if가 더 나은 방법이긴 합니다만, 좀 더 간단하게 switch 구문을 사용하기도 합니다(가끔 어떤 책들을 보면 switch를 권장하는 책들도 있습니다만 개인적으로는 그다지 추천하고 싶지는 않습니다. 이유는 잠시 후에 설명드리겠습니다).

switch는 if ~ else if와 같은 기능을 하지만 약간 가독성을 높이는 효과가 있습니다.
- 숫자, char, enum 을 기준으로 제어합니다(enum은 지금 다루지 않습니다).
- JDK1.7에서는 문자열의 switch 처리가 추가될 예정입니다.

JDK1.6의 경우에는 switch는 간단한 숫자(int)와 간단한 문자(char)를 구분으로 해서 돌아갑니다. switch 구문을 알려면 case와 break의 사용법을 같이 알아야만 합니다. switch 구문의 문법은 다음과 같습니다.

```
switch(구분 기준값) {
case 판단 값 :
        true일 때의 로직 ;
break; ← 여기까지만 실행하겠다는 선언(경계선)
case 판단 값 :
        true일 때의 로직 ;
break;
...
default :
        아무 조건에 해당하지 않을 때의 로직
}
```

switch 구문은 case가 if ~ else if의 역할을 수행합니다. 각 판단 값은 case라는 단어에 의해서 단위가 구분됩니다. 여기서 또 하나 주의하실 점은 Java의 모든 라인을 끝낼 때에는 세미콜론(;)을 쓰지만, case 문은 콜론(:)으로 끝난다는 점입니다. 또한, 각 case 문을 끝낼 때에는 break 문을 이용하는데 break에 대해서는 루프를 설명할 때 자세히 보겠습니다만, 우선 여기서는 실행을 벗어난다는 의미로 이해하시면 되겠습니다. if ~ else 구문은 일반적인 경우라면 switch 구문으로 바꿀 수 있습니다.

코드

```
char c = 'A';
if( c == 'A'){
        System.out.println("AAAAAAAAAAA");
}else if(c == 'B'){
        System.out.println("BBBBBBBBBBB");
}else if(c == 'C'){
        System.out.println("CCCCCCCCCCCC");
}else {
        System.out.println("unknown");
}
```

```
switch(c){
case 'A':
        System.out.println("AAAAAAAAAAA");
        break;
case 'B':
        System.out.println("BBBBBBBBBBB");
        break;
case 'C':
        System.out.println("CCCCCCCCCCCC");
        break;
default:
        System.out.println("unknown");
}
```

위의 코드에서 보는 것처럼 switch 구문을 활용하면 확실히 가독성이라는 부분에서는 좀 나아집니다. switch 구문을 적용하기 위해서 '날씨'를 예제로 하나 만들어 보겠습니다. 현재 온도를 10으로 나눈 값을 기준으로 해서 몫이 1일 경우에는 쌀쌀한 날씨, 2일 경우에는 따뜻한 날씨, 3일 경우에는 더운 날씨라고 해 보겠습니다. 만일 1, 2, 3이 아니라면 그냥 말하기 어려운 날씨라고 하겠습니다.

예제 | switch 구문을 이용한 분기

```
public class SwitchEx {
        public static void main(String[] args) {
                int grade = 35;

                //정수와 정수의 연산은 정수만 나온다(3.5이지만 결과는 3).
                int odd = grade / 10;
```

```
                switch(odd){
                case 1:
                        System.out.println("쌀쌀한 날씨입니다.");
                        break;
                case 2:
                        System.out.println("따뜻한 날씨입니다.");
                        break;
                case 3:
                        System.out.println("좀 더운 날씨입니다.");
                        break;
                default:
                        System.out.println("뭐라 말씀드리기 어려운 날씨네요.");
                }
        }
}
```

■ **switch 문 지나치게 좋아하지 마세요: 단점 많은 switch**

일부 Java 서적을 보면 switch 문이 보기 간결하고, 좋은 방법의 코딩이라고 칭송하는 책들도 많이 봐왔습니다만, 다음과 같은 몇 가지 단점을 보시고 나면 생각이 달라지실 겁니다.

- switch는 부등호 연산이나, 논리 연산, 심지어 더하기, 빼기, 곱하기, 나누기처럼 산술연산마저도 사용할 수 없다. 즉 정확한 값이 나오는 숫자나 글자만을 사용하기 때문에 코딩에 많은 제약이 따른다.
- break 문을 빼먹으면 다음 case까지 적용된다. 이때 컴파일 시점에는 아무 이상 없이 지나치기 때문에 코드 양이 많아지면 구분하기 어렵다.
- 세미콜론과 콜론의 차이는 구분하기 쉽지 않다. Java의 모든 코드가 세미콜론(;)으로 끝나지만 유독 switch만은 콜론(:)을 사용하므로 몇백 라인의 코드를 메모장이나 vi 편집기로 열어서 코드를 보면서 숨어 있는 콜론 표시를 찾는 일은 너무나 힘든 일이다.

예제 | break가 없는 switch는 위험하다.

```
public class SwitchEx {
	public static void main(String[] args) {
		int grade = 10;

		//정수와 정수의 연산은 정수만 나온다.
		int odd = grade / 10;

		switch(odd){
		case 1:
			System.out.println("쌀쌀한 날씨입니다.");
			//break;
		case 2:
			System.out.println("따뜻한 날씨입니다.");
			//break;
		case 3:
			System.out.println("좀 더운 날씨입니다.");
			//break;
		default:
			System.out.println("뭐라 말씀드리기 어려운 날씨네요.");
		}
	}
}
```

```
쌀쌀한 날씨입니다.
따뜻한 날씨입니다.
좀 더운 날씨입니다.
뭐라 말씀드리기 어려운 날씨네요.
```

온도를 10도로 조정하고 모든 break 문을 주석으로 수정한 후에 실행되는 결과를 보시면 느끼실 겁니다. 사실 break는 이후에 다시 설명하겠지만, 일종의 실행을 중지하는 기능이기 때문에 중지되지 않은 로직은 무작정 계속 실행되어 위에서처럼 어처구니없는 결과가 발생합니다. 간단한 코딩에서 이게 뭐 그리 큰 문제인가라고 생각하는 분들이 계실 수도 있습니다만, 실제로 각 case 문 안에 있는 로직이 몇백 라인을 넘어갈 때에는 심각하게 생각해볼 만합니다.

4 루프(Loop): 지정된 만큼 실행하라.

아주 짧은 코드로 여러 번 실행되는 코드를 작성할 때 사용하는 구문을 루프(반복문)이라고 합니다.
- for 루프: 주로 횟수가 지정된 경우에 사용합니다.
- while 루프: 얼마나 반복하는지를 결정하지 못하는 경우에 주로 쓰입니다.

여러분이 화면에 'HelloWorld'를 열 번 출력하는 프로그램을 만들려면 간단히 Copy & Paste를 이용해서 만들어 낼 수 있을 겁니다.

```
System.out.println("HelloWorld");
System.out.println("HelloWorld");
System.out.println("HelloWorld");
...
```

위를 열 번 복사해서 붙여 넣기(Copy & Paste)를 하면 됩니다. 그런데 여러분에게 100번이나 1,000번을 찍으라고 한다면 상황이 달라집니다. 프로그램을 작성하다 보면 반복적으로 처리해야 하는 작업이 꽤 많습니다. 예를 들어 모든 성적의 합을 구한다든가, 모든 회원에게 전화를 건다든가, 단체 메일을 발송한다든가 등 이런 반복적인 작업을 매번 코드로 작성하지 않고 사용하는 것이 바로 루프(순환문)입니다. Java에서 쓰이는 순환 구문 중에서 반드시 알아 두셔야 하는 것은 for 루프와 while 루프 구문입니다.

4.1 for 루프: 지정된 횟수만큼의 순환

순환문 중에 대표주자를 뽑으라면 단연 for 루프입니다. for 루프는 주로 지정된 만큼의 로직을 반복적으로 실행할 때 사용합니다. for 루프는 다음과 같이 구성됩니다. 영어의 for를 '~하는 동안'이라고 해석하는 것과 동일합니다.

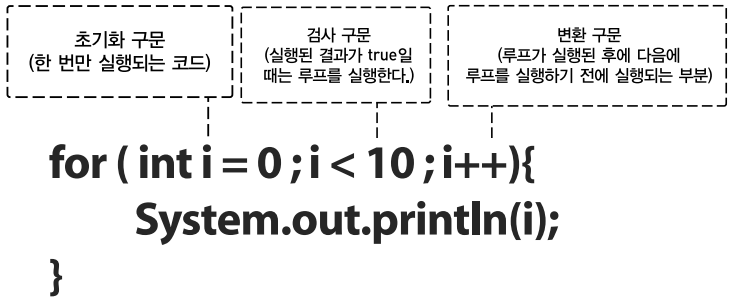

그림 5

- 초기화 구문: 순환문을 시작할 때 준비하는 조건을 의미합니다. 공백으로 비워 둘 수 있다. 주로 루프를 돌면서 몇 번이나 돌고 있는지를 세는 count 값의 의미를 가진다(옵션).
- 검사 조건: 루프가 다음에서 실행되어야 하는지를 검사하는 부분이다. 이 부분은 true/false로 판단할 수 있으면 된다. 공백으로 비워둘 수 있다(옵션).
- 변환 조건: 루프가 계속 실행되면서 무언가 변화가 일어나야 한다면 처리하는 부분을 말한다. 예를 들어 실행될 때마다 카운트 값이 증가하던가, 특정한 변수의 값이 조절될 때 사용하는 부분이다(옵션).

4.1.1 for 루프의 초기화 구문: 루프를 실행하기 전 준비 과정

초기화 조건은 루프를 실행하기 전에 준비 작업을 하는 부분입니다. 이런 준비 과정으로 가장 많이 쓰이는 형태는 루프가 실행되기 위한 어떤 변수를 선언하거나, 반드시 먼저 실행되는 코드를 넣는 것입니다.

■ 변수를 선언하는 초기화 구문

코드

```
for( int i = 0; i < 10; i++){ // i라는 변수의 선언을 목적으로 사용합니다.
        System.out.println(i);
}
```

이럴 때 i라는 변수는 for 루프의 {}의 영향을 받습니다. 따라서 i라는 변수는 for 루프 안에서만 사용하는 변수가 됩니다.

■ **특정 코드를 먼저 실행하게 하는 초기화 구문**

자주 사용되지는 않습니다만 개념을 정리하는 차원에서 보면 for 루프의 초기화 구문은 다음과 같이 사용할 수도 있습니다.

예제 초기화 구문에는 루프가 돌기 전에 실행되는 코드를 넣습니다.

```
public class ForEx1 {
	public static void main(String[] args) {
		int a = 10;
		for( System.out.println("AAAA"); a < 20; a++ ){
			System.out.println("BBBB");
		}
	}
}
```

```
AAAA
BBBB
BBBB
이하 생략
```

소스를 보면 "AAAA"가 가장 먼저 한 번 실행되는 모습을 볼 수 있습니다.

4.1.2 for 루프의 검사 조건: true면 실행하라.

루프에서는 검사 조건이 가장 핵심입니다. 간단히 말해서 모든 루프는 검사 조건을 true로 판단할 수 있으면 구문을 실행합니다.

```
for(int i = 0;  true ; i++){ ... }
```

검사 조건이 true라면 {} 안의 내용은 무한히 반복됩니다. 위에서는 아예 검사 조건을 주지 않았습니다. 검사 조건을 주지 않으면 그냥 지나쳐 버리기 때문에 이 역시 계속 실행되게 됩니다(실무에서 이런 코드로 작성하지는 않습니다 다만 원리를 설명하기 위해서 사용했습니다).

4.1.3 for 루프의 변환 조건: 밑에까지 다녀왔으니까.

변환 조건은 루프의 {} 내부가 실행되고 나서 다시 루프의 검사 조건으로 가기 직전에 실행되는 구문입니다.

```
for(int i = 0; i < 10; i++){ ... }
```

루프를 한 번이라도 실행해야 변환 조건을 거칩니다. 주로 검사 조건에서 필요한 변수의 값을 변경하는 데 사용합니다.

```
for(int i = 0; i < 10; ){ ... }
```

위와 같이 변환 조건이 없는 경우는 어떨까요? 루프를 돌면서 변경되는 값이 없으므로 'i < 10'이라는 검사 조건은 항상 true이므로 계속 실행됩니다.

참고

i++과 ++i

사실 '++i'라고 써도 상관 없으며(실제로는 아주 미세하게 ++i가 더 빠릅니다만) 'i = i+1'과 같이 작성해도 상관없습니다. 다만, 'i++'이라고 쓰는 게 조금 더 직관적이라는 이유에서 사용합니다.

4.1.4 for 루프의 실행 순서: 이렇게 해서 계속 반복합니다.

for 루프에서 반드시 이해해야 하는 것이 두 가지가 있다면 하나는 위에서 설명한 for 루프의 기본 구문이고, 다른 하나는 실행 순서입니다. 루프가 실행되면 어떤 결과가 화면에 출력되는가에 대한 설명입니다.

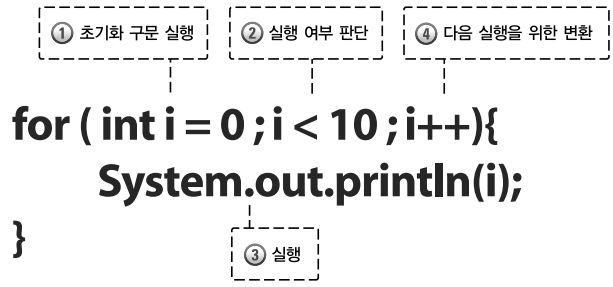

그림 6

결국, 위의 루프를 화면으로 출력해 보면 0부터 값이 출력되기 시작합니다. 맨 마지막은 10이 출

력될까요? 9가 출력될까요? 검사 조건을 자세히 보시기 바랍니다. 검사 조건의 결과가 10보다 작을 때만 true가 되기 때문에 10이 되면 검사 조건의 결과가 false가 되므로 출력되지 않습니다.

4.1.5 for 루프를 이용해서 0에서 10까지 찍어 보기, 10에서부터 카운트다운하기

모든 언어의 for 루프에서 나오는 가장 단순한 예제입니다. 하지만, 루프를 전혀 실행해본 적이 없으시다면 한 번쯤 실습해보실 필요가 있습니다. 이번에 예제는 0에서 10까지 증가하는 루프와 10에서부터 0으로 감소하는 루프를 만들어 보도록 하겠습니다.

예제 0에서 10까지 출력

```
public class ForEx2 {
	public static void main(String[] args) {
		for(int i = 0; i <= 10; i++){
			System.out.println(i);
		}
	}
}
```

```
0
1
2
3
4
5
6
7
8
9
10
```

루프에서 i<= 10까지로 검사 조건을 주었다는 걸 주의해서 보기 바랍니다. 0에서 10까지의 출력하는 것은 위에서 루프의 시작과 끝을 생각해 보시면 됩니다. 반대로 10에서부터 카운트다운하는 기능은 어떤 방식으로 만들어야 할까요? for 루프를 구성하는 세 가지 조건을 생각해 보겠습니다.

- 초기 조건: 숫자 10에서부터 시작하면 될 듯하니 'int i = 10;'이라고 초기 조건을 주면 될 것이다.
- 검사 조건: 검사 조건은 당연히 현재 변하는 값이 0보다 크거나 같은 경우 true로 판단하면 된다(i <= 0).

• 변환 조건: 위에서 만든 코드가 루프를 순환할 때마다 1씩 증가하였다면, 현재는 1씩 감소해야 하므로 i--가 된다.

예제 | 10에서 0까지 출력

```
public class ForEx3 {
	public static void main(String[] args) {
		for(int i = 10; i >= 0 ; i--){
			System.out.println(i);
		}
	}
}
```

```
10
9
8
7
6
5
4
3
2
1
0
```

위의 두 예제는 초기 조건, 검사 조건, 변환 조건의 개념을 잘 이해하셨다면 그다지 어렵지 않게 이해할 수 있으리라 생각합니다. 조금 다른 문제를 생각해 보겠습니다.

4.1.6 10까지의 숫자 중 홀수만 출력

루프를 도는 데 있어서 홀수의 결과만 출력할 수 있는 방법은 어떤 것이 있을까요? 많은 방법이 있겠지만 간단하게 루프가 1에서 시작해서 10까지 도는 동안, 2씩 증가시키는 방법이 있고(1, 3, 5, 7, 9), 또 하나는 전체를 순환하는 데 있어서 if 구문을 이용해 짝수일 경우에는 출력하지 않도록 하는 방법도 고려해 볼 수 있습니다. 하지만, 모든 루프를 돌면서 홀수 값을 확인하려면 실제로 많은 루프를 순환하기 때문에(홀수 짝수 포함해서 결국 2배로 많은 루프를 돌게 됩니다.) 성능면에서 좋지 않을 듯합니다.

• 시작 조건: 홀수니까 1에서 시작하는 것이 좋다. (int i = 1)

• 검사 조건: 10보다 작은 수 일 경우에만 사용하므로 (i < 10)
• 변환 조건: 1에서부터 루프를 순환할 때마다 2씩 증가하므로 (i = i +2)

조금 낯선 부분이 바로 마지막 변환 조건 부분입니다. 자동 증감 연산자는 i++ 하게 되면 실제로 i 값이 증가합니다. 하지만, 일반적인 변수 선언으로 생각해 보면 'i = i+1;'이라는 연산과 i++은 동일한 결과를 만들어 냅니다(변수의 자동증감 부분을 참고해주세요). 따라서 위의 내용을 코드로 만들면 다음과 같은 소스를 작성할 수 있습니다.

예제 | 변환 조건을 이용한 홀수 출력

```
public class ForEx4 {
	public static void main(String[] args) {
		for(int i = 1; i < 10; i= i+2){
			System.out.println(i);
		}
	}
}
```

```
1
3
5
7
9
```

4.1.7 for 루프를 이용해서 1에서 10까지의 합 구하기

앞에서도 말씀드렸듯이 기본 자료형과 제어문(루프 포함)은 모든 프로그래밍의 기본입니다. 변수를 선언할 수도 있고, 사용할 수도 있고, 연산자도 알고, 루프나 if와 같은 제어문도 학습했으니 이제는 1에서 10까지의 합을 구하는 것과 같은 누적되는 계산을 해 볼 차례입니다. 결론부터 말하자면 1에서 10까지의 합은 55입니다. 그럼 이 결과를 만드는 프로그램을 어떻게 만들어야 하는지 생각해 보겠습니다.

• 당연히 루프는 돌아야 한다: 1에서 10까지 값이 계속해서 변경되어야 하기 때문
• for(int i = 1; i < 10; i++){ ... }에서 매번 바뀌는 결괏값을 저장해야 한다. 바뀌는 값(변경되는 데이터를 저장하려면 변수가 선언되어야 한다. 변수 설명 부분을 참고)을 저장하려면 변수를 선언해야 한다.

• 선언된 변수에는 매번 루프에 의해서 변경되는 값이 계속 쌓여야 한다.

```
sum = 0;
sum = sum + 1; → 기존의 sum 변수에서 값을 꺼내서 더한 후에 다시 보관하는 작업을 반복
sum = sum + 2;
sum ...
sum  = sum + 10;
```

위의 코드를 보면서 루프를 돌면서 적용되도록 코드를 수정하면 다음과 같은 결과가 됩니다.

예제 1에서 10까지의 합을 구하는 for 루프

```
public class ForEx5 {
	public static void main(String[] args) {
		int sum = 0;
		for(int i = 1; i <= 10; i++){
			sum = sum + i;
			System.out.println("sum : " + sum);
		}
		System.out.println("total: " + sum);
	}
}
```

```
sum : 1
sum : 3
sum : 6
sum : 10
sum : 15
sum : 21
sum : 28
sum : 36
sum : 45
sum : 55
total: 55
```

4.2 while 루프: 몇 번 돌아야 하는지 모른다면

while 루프는 for 루프와 더불어서 가장 많이 사용되는 루프 구문입니다. while 루프는 특이하게도 검사 조건 하나만 가지고 실행됩니다.

while 구문은 단순히 검사 조건 하나만을 가집니다.

```
while(검사 조건){
        검사 조건이 true인 동안 실행될 로직
}
```

while 구문에는 초기 조건이나 변환 조건이 없다는 점을 유심히 봐 두셔야 합니다. 초기 조건이나 검사 조건은 결국 프로그래머들이 알아서 작업해야 하는 부분입니다. for 루프에 비하면 명확하지 않다는 느낌입니다. 오직 중간에 검사 조건만 있으므로 while 구문은 정확한 판단 시보다는 불확실한 경우에 많이 사용됩니다. for 루프를 while 루프로 변경해보면 그 차이점을 알 수 있습니다.

코드 | 0에서 10까지의 출력을 while 루프로 변경

```
public class whileEx {
        public static void main(String[] args) {
                int i = 0; // 초기 조건이 루프 밖으로
                while(i < 10){
                        System.out.println(i);
                        i++; // 변환 조건이 루프 안으로
                }//end while
        }
}
```

위의 코드를 보면 for 루프에서의 초기 조건 'int i = 0;'이 while 루프의 앞쪽으로 올라옵니다. 루프를 실행할 때의 검사 조건은 for 루프와 동일합니다. for 루프에서는 마지막 부분에 루프가 실행되면서 변화되는 변환 조건이 들어가는 반면에 while 루프는 검사 조건만 있으므로 실행되면서 변환하는 로직은 결국 while 루프 안쪽에서 직접 작성할 수밖에 없습니다.

■ for 루프와 while 루프의 차이

항목	for 루프	while 루프
초기화 구문	주로 'int i = 0;'과 같이 준다. 옵션이긴 하지만 거의 선언하는 것이 일반적이다.	없음
검사 구문	옵션이긴 하지만 주로 선언한다.	반드시 필요
변환 구문	옵션이긴 하지만 주로 선언한다.	없음
사용되는 경우	루프를 명확하게 시작과 끝을 아는 경우에 주로 사용 (루프의 숫자가 명시적인 경우에는 for 루프가 일반적)	몇 번이나 실행되는지 짐작하기가 어려운 경우에 사용

표

아직은 배열이나 자료구조에 대해 이야기하지 않았기 때문에 우선은 while은 루프의 실행 숫자를 알 수 없을 때 적합하다고만 생각하시기 바랍니다.

4.3 영원히 실행되는 무한 루프

for 루프나 while 루프 모두 검사 조건이 있습니다. 간단히 말하자면 루프를 돌아야 하는가 혹은 멈춰야 하는가를 결정해주는 부분입니다. 그런데 조금만 더 깊이 생각해보면 모든 검사 조건은 true/false로 판단되는 조건입니다. 즉 다음과 같은 for 루프나 while 루프는 무한히 실행되는 루프라는 겁니다.

코드

```
while(true){
        System.out.println("Hello World");
}
```

항상 true이므로 계속 실행합니다.

코드

```
for(int i = 0; true ; i++){ // 검사조건이 무조건 true
        System.out.println("Hello World");
}
혹은
for(int i = 0;     ; i++){ // 검사를 하지 않음
        System.out.println("Hello World");
}
```

이런 무한 루프는 루프의 아래쪽의 모든 코드를 실행시키지 못하므로 프로그래밍에서는 항상 주의해서 작성해야 합니다.

4.4 break, continue: 루프를 컨트롤하는 구문

살다가 보면 늘 예상하지 못하는 문제에 봉착할 수 있습니다. 연산이 잘못되어 루프의 검사 조건이 항상 true가 나오게 되어서 무한 루프에 빠진다든가, 한두 번 정도만 루프를 돌려도 충분한데 괜히 모든 데이터에 대해서 루프를 돌게 된다든가 등 이러한 경우에 대비해서 루프를 중간에 컨트롤할 수 있는 방법을 제공합니다.

4.4.1 break: 어떤 순간에도 벗어날 수 있습니다.

break는 루프에서 완전히 탈출할 때 사용합니다. 무한 루프이건 뭐건 간에 상관없습니다. 실행되다가 언제든지 break를 만나면 루프에서 벗어날 수 있습니다.

> break는 루프를 벗어날 때 사용합니다. 아무리 많은 { }가 있더라도 관계없이 가장 가까운 루프에서 벗어나게 할 때 사용합니다.

예제 | break를 이용하여 루프 중간에서 중지하는 경우

```
public class BreakEx {
	public static void main(String[] args) {
		for(int i = 0;  i < 100; i++){
			if(i == 5){
				System.out.println("i 값이 5이므로 벗어납니다.");
				break;
			}//end if
			System.out.println(i);
		}//end for
	}
}
```

```
0
1
2
3
4
i 값이 5이므로 벗어납니다.
```

코드를 보면 break 구문이 if ~ else의 {} 안에 있음에도 불구하고 루프를 벗어나는 역할을 하는 것을 볼 수 있습니다.

4.4.2 continue: 무조건 루프를 위로 다시 올려줍니다.

continue는 원래 "루프를 계속해서 실행한다."라는 의미이지만, 간편히 생각하기 위해서 무조건 루프를 다시 위로 올린다고 생각하면 편리합니다.

continue에 대해서는 영어로만 해석해보자면 당연히 '계속~'이라는 뜻입니다만, 실제로 continue를 소스에 적용해 보면 그런 개념으로 이해하는 것보다는 루프를 다시 실행하게끔 위로 보내는 기능으로 이해하는 게 더 편합니다. 앞쪽에서 만든 break 예제를 continue로 변경하고 루프를 100이 아니라 10까지만 실행하게 만들어 보겠습니다.

예제 continue는 루프를 건너뛰게 합니다.

```
public class ContinueEx {

	public static void main(String[] args) {
		for(int i = 0;  i < 10; i++){
			if(i == 5){
				System.out.println("i 값이 5이므로 continue");
				continue;
			}//end if
			System.out.println(i);
		}//end for
	}
}
```

i의 값이 5일 때는 무조건 위로 갑니다.

```
0
1
2
3
4
i 값이 5이므로 continue
6
7
8
9
```

실행되는 결과를 보시면 5일 경우에만 숫자가 출력되지 않는 걸 보실 수 있습니다. continue를 만나면 루프는 현재보다 아래에 있는 로직을 실행하지 않고 바로 다시 위로 가서 루프를 순환하게 됩니다. 따라서 계속이라는 원래의 의미로 이해하는 것보다는 '무조건 위로'라는 뜻으로 이해하는 것이 더 적합합니다.

4.4.3 continue 활용 팁

continue를 적절히 사용하면 보다 편리하게 코드를 작성할 수 있습니다. 다음의 코드를 생각해봅시다.

코드

```
public class ContinueExample {
	public static void main(String[] args) {
		for(int i = 0; i <= 10; i++){
			System.out.println(i);
		}
	}
}
```

예를 들어 위의 코드처럼 0에서 10을 순환하는 루프에서 홀수의 합을 따로 계산하고, 짝수의 합을 구한다고 생각해봅시다.

예제 | 홀,짝, 모든 수의 합을 계산하는 if ~ else

```
public class ContinueExample {
	public static void main(String[] args) {
		//홀수의 합
		int oddSum = 0;
		//짝수의 합
		int evenSum = 0;
		//모든 합
		int totalSum = 0;

		for(int i = 0; i <= 10; i++){
			System.out.println(i);
			totalSum = totalSum + i;
			if( i % 2 == 0){
				evenSum = evenSum + i;
			}else if(i % 2 == 1){
				oddSum = oddSum + i;
```

```
                }
            }
            System.out.println("홀수: " + oddSum );
            System.out.println("짝수: " + evenSum);
            System.out.println("총합: " + totalSum);
        }
}
```

지금은 단순히 홀수, 짝수이지만 if 때문에 코드가 더욱 복잡해 집니다. 위의 코드를 continue로 변경하면 다음과 같은 형태가 됩니다.

코드

```
// 단순화된 홀짝의 합
for(int i = 0; i <= 10; i++){
        System.out.println(i);
        totalSum = totalSum + i;
        //짝수일때는 그냥 다시 루프로
        if( i % 2 == 0){
                evenSum = evenSum + i;
                continue; // 짝수일 때는 다시 루프 위쪽을 실행
        }
        oddSum = oddSum + i;
}
```

짝수일 때는 계산하고 다시 루프를 위로 올려 버리는 방식을 사용하면 코드를 조금 더 간결하게 작성할 수 있습니다.

5 do ~ while: 무조건 실행하고 조건을 따지는 방식

do~ while은 다른 루프들과는 달리 무조건 저지르고 보는 스타일입니다. 마치 비행 청소년 같다고나 할까요? 아무 생각 없습니다. 우선 실행할 로직을 실행해 두고 차후에 while의 판단 조건을 가지고 계속 할까 말까 고민하는 방식입니다. do~ while은 루프 중에서 가장 사용빈도가 낮습니다. 우선을 구문 자체는 다음과 같이 작성합니다.

```
do {
        반드시 한번은 실행되어야 하는 구문
}while(판단 조건);
```

예제

```
public class DoWhileEx {
        public static void main(String[] args) {
                do{
                        System.out.println("반드시 한번은 실행");
                }while(false);
        }
}
```

do이므로 무조건 실행합니다.

실행 조건은 false이므로 이후에는 실행하지 않습니다.

반드시 한 번은 실행

위의 코드를 보시면 판단 조건이 false입니다. 하지만, 실행 결과를 보면 반드시 한 번은 실행되어서 나옵니다. 무조건 한 번은 실행하며 나머지는 while과 동일합니다. 하지만, 실제로는 do ~ while은 많이 쓰이지 않습니다. 우선은 while이라는 판단 기준이 마지막에 있기 때문에 코드를 볼 때 for 루프처럼 직관적으로 알아볼 수가 없습니다. 또한, while 루프를 이용하더라도 변수를 활용하면 같은 효과를 얻을 수 있습니다.

코드

```
// do ~ while 대신에 while을 사용하는 것이 더 보기 편합니다.
boolean start = true;
while(start){
        System.out.println("실행");
        start = false;
}
```

보시기에는 어떠실지 모르겠지만, 위처럼 작성하는 로직이 더 많다고 알아 두시면 됩니다.

6 주사위 게임 만들기

이번에는 여러분과 컴퓨터가 주사위를 굴리는 게임을 한번 만들어보도록 합니다.

6.1 주사위 게임의 실행 방식

1_ "화면에 주사위를 굴릴까요?"라는 메시지가 보입니다.

2_ 사용자는 Enter 를 누릅니다.

3_ 화면에 사용자가 뽑은 번호가 출력됩니다.

4_ "컴퓨터가 주사위를 굴립니다. 실행할까요?"라는 메시지가 출력됩니다.

5_ 사용자는 Enter 를 누릅니다.

6_ 컴퓨터가 만들어 낸 주사위 숫자가 출력되고, 사용자가 더 높은 수이면 "You Win!", 낮은 수이면 "You Lose!", 비기면 "Draw"라는 메시지를 출력합니다.

처음부터 실행하면 다음과 같은 화면이 됩니다.

```
안녕하세요 ^^
D I C E  G A M E !!!!!
주사위를 굴려 볼까요? (사용자가 Enter 를 누를 때까지 대기)
당신의 숫자는 : 4
컴퓨터가 주사위를 굴려 볼까요? (사용자가 Enter 를 누를 때까지 대기)
컴퓨터의 숫자는 : 6
You LOSE!
```

6.2 배운 내용과 생각해야 하는 내용

주사위 게임을 만들고 싶다면 우선은 아는 것과 모르는 것을 구분할 필요가 있습니다. 아는 것과 모르는 것을 표로 만들어 보시면 많은 도움이 됩니다.

■ 배운 내용 중에서

- 화면에 메시지 뿌리는 것
- Random이라는 것을 이용하면 임의의 숫자를 만들 수 있다는 것
- if ~ else를 이용해서 두 개의 숫자를 비교해서 메시지를 출력하는 것

■ 배우지 않은 내용 찾아내기

- 사용자가 Enter 를 누르는 동작(변수에서 비슷한 것을 잠시 본 적이 있는 듯하기도)
- 정확하게 1에서 6까지의 숫자를 만드는 방법

6.3 입력한 문자열을 알아내는 Scanner의 nextLine( )

Scanner는 어떤 곳에서 흘러들어오는 데이터(스트림이라고 합니다)를 우리가 쉽게 알아낼 때 사용하는 장치입니다. 흘러들어오는 데이터라는 것을 쉽게 생각하자면 마치 집까지 이어진 수도관과 비슷하다고 할 수 있을 것 같습니다만, 자세한 얘기는 나중에 입출력을 프로그래밍할 때 다루도록 하겠습니다.

Scanner s = new Scanner(System.in);

Scanner에는 여러 가지 기능이 있습니다. 변수를 공부할 때에는 nextInt( )를 이용해 보았습니다.

nextInt( ) – 키보드에서 입력되는 글자를 숫자로 바꿔주는 기능

화면에서 Enter 를 누르면 주로 라인이 변경됩니다. Scanner에는 이렇게 입력된 라인을 읽어들이는 기능을 가지고 있습니다. nextLine( )은 입력된 데이터를 문자로 만들어 줍니다. 간단하게 무언가 메시지를 입력받도록 하겠습니다.

예제 키보드로 입력한 문자열을 인식하는 Scanner의 nextLine( )

```
import java.util.*;
public class ReadLineEx {
        public static void main(String[] args) {
                Scanner s = new Scanner(System.in);
                System.out.println("아무거나 입력해 보시죠^^");
                System.out.println(s.nextLine());
        }
}
```

```
아무거나 입력해 보시죠^^
하하하 ← 입력한 데이터
하하하
```

코드를 실행할 때 주의해서 봐야 하는 부분은 사용자가 Enter 를 누르기 전까지 프로그램이 잠시 멈추게 된다는 겁니다(이것을 blocked되었다고 얘기합니다만, 대기 상태라고 알고 계시면 됩니다).

6.3.1 1에서 6까지의 숫자를 만드는 방법

1에서 6까지의 숫자를 만드는 방법은 상대적으로 좀 쉽습니다. Random에 있는 nextInt(숫자)를 생각하면 됩니다. nextInt(6)을 하게 되면 0에서부터 6 미만의 수(0~ 5)를 발생하게 되니까 발생한 수에 1을 더하면 됩니다.

```
Random r = new Random( );
int num = r.nextInt(6) + 1 ;
```

6.3.2 순서를 세우고 조립하기

1_ DiceGame 클래스를 선언하고 main 메소드를 만들어줍니다.

2_ 준비물을 선언해둡니다.

A. 사용자의 키보드 입력을 받아들이기 위한 Scanner

B. 임의의 숫자를 발생시키는 Random

예제

```
import java.util.Random;
import java.util.Scanner;

public class DiceGame {
        public static void main(String[] args) {
                //준비 도구
                Scanner scanner = new Scanner(System.in);
                Random random = new Random();
```

3_ 사용자에게 주사위 게임에 대한 메시지를 알려주고, 사용자에게 Enter 를 입력하도록 해주는 메시지를 출력합니다.

예제

```
                //인사말
                System.out.println("안녕하세요 ^^");
                System.out.println(" D I C E  G A M E !!!!! ");

                //사용자가 주사위를 굴리는 부분
                System.out.println("주사위를 굴려 볼까요?");
```

4_ 사용자가 Enter 를 입력하는 것을 감지하고 주사위의 숫자를 발생합니다. 발생한 숫자를 화면에 출력해줍니다.

예제

```
//사용자가 엔터를 치면 다음 부분이 실행
System.out.println( scanner.nextLine());

//사용자의 주사위 숫자 1- 6이 나와야 한다.
int userNumber = random.nextInt(6) + 1;

System.out.println("당신의 숫자는 : " + userNumber);
```

5_ 이제 컴퓨터가 주사위를 굴리도록 다시 한번 메시지를 출력해주고 Enter 를 누르면 숫자를 발생시킵니다.

예제

```
//컴퓨터가 주사위를 굴리는 부분
System.out.println("컴퓨터가 주사위를 굴려 볼까요?");
//사용자가 엔터를 치면 다음 부분이 실행
System.out.println( scanner.nextLine());

//컴퓨터의 주사위 숫자 1~6이 나와야 한다.
int comNumber = random.nextInt(6) + 1;

System.out.println("컴퓨터의 숫자는 : " + comNumber);
```

6_ 마지막으로 사용자의 수와 컴퓨터가 가지는 숫자를 비교합니다.

예제

```
if(userNumber > comNumber){
        System.out.println("You WIN!!");
}else if(userNumber == comNumber){
        System.out.println("DRAW");
}else{
        System.out.println("You LOSE!");
}
```