제 2 부 — 제어 구조와 입출력

5장
제어 흐름

Choices, Loops, and the Flow of Control

if · select case · do · do while · do concurrent · stop

THE SHAPE OF A LOOP

```
search: do i = 1, n
   do j = 1, n
      if (...) then
         exit search
      end if
   end do
end do search
```

학습 목표

이 장에서 익히는 네 가지

01

분기 구성

if 와 select case 로 실행 경로를 나눈다.

02

반복 제어

do 와 do while 로 반복을 만들고 cycle · exit 으로 흐름을 조절한다.

03

do concurrent 맛보기

병렬 반복의 출발점인 do concurrent 를 소개 수준에서 다룬다.

04

안전한 종료

stop 과 error stop 으로 프로그램을 의도대로 멈춘다.

도입 문제

이 장을 마치면 풀 수 있다

1

점수를 등급으로, 음수면 정지

점수를 a ~ f 등급으로 분류하고, 입력이 음수이면 계산을 멈추는 프로그램을 작성한다.

```
select case  +  error stop
```

2

수렴할 때까지 반복하고, 그림으로 확인

어떤 수열을 반복으로 누적해 원하는 정밀도에 수렴할 때까지 계산하고, 그 수렴 과정을 그림으로 확인한다.

```
do while  +  exit  +  tol
```

프로그램이 고정된 순서로만 계산된다면 복잡한 현실의 문제를 풀 수 없다 — 값에 따라 다르게 처리하고, 같은 작업을 여러 번 반복해야 한다.

5장 전체 지도

제어 흐름의 세 갈래

분기

값에 따라 길을 가른다

`if`

한 문장 또는 블록

`select case`

여러 후보와 대조

반복

같은 작업을 되풀이한다

`do`

횟수가 정해진 반복

`do while`

조건이 참인 동안

`do concurrent`

순서 없는 독립 반복

흐름 제어

루프와 프로그램을 조절한다

`cycle`

다음 바퀴로 건너뛴다

`exit`

반복을 즉시 벗어난다

`stop / error stop`

프로그램을 멈춘다

이 장은 이 세 갈래를 차례대로 훑는다.

5.1 if 구문과 분기

논리 if 와 블록 if

논리 if (logical if)

조건이 참일 때 한 줄의 명령문만 실행하는 가장 간결한 구조.

```
if (logical-expr) statement

if (temp < 0.0_dp) is_freezing = .true.
```

블록 if (block if)

실행할 명령이 여러 줄이거나 조건을 여러 갈래로 나눌 때.

```
if (expr) then
   block
[else if (expr) then ...] [else ...]
end if
```

then 과 end if 는 한 쌍

블록 if 의 시작에는 반드시 then 이 와야 하고 end if 로 닫는다. 짝이 맞지 않으면 컴파일러가 빌드를 차단한다.

else if 는 몇 개든

앞선 조건이 거짓일 때 또 다른 대안 조건을 검사하기 위해 연속해서 이어 쓸 수 있다.

else 는 최후의 보루

모든 조건이 거짓일 때 실행되며 최대 하나만 올 수 있다. 별도의 조건을 적지 않는다.

위에서부터 차례로 검사하며, 처음으로 참이 되는 단 하나의 블록만 실행한 뒤 전체 if 문을 빠져나간다.

5.1 [예제]

실수의 부호 분류 — == 를 피하는 설계

```
real(real64), parameter :: x = -3.5_real64

if (x > 0.0_real64) then
   print '(a)', "positive"
else if (x < 0.0_real64) then
   print '(a)', "negative"
else
   print '(a)', "zero"
end if
```

```
negative
```

수학적으로 실수는 양수 · 음수 · 0 이라는 세 상태뿐이다 — 두 개를 부등호로 잡고 나머지를 else 가 받는다.

흔한 유혹

대다수 입문자는 else if (x == 0.0_real64) 처럼 0 인 조건을 == 로 직접 검사하려 한다. 그러나 부동소수점을 == 로 비교하는 행위는 절단 오차 때문에 언제든 오작동할 수 있다.

안전한 설계

> 로 양수 영역, < 로 음수 영역을 먼저 걸러 낸 뒤, 그 어디에도 속하지 않는 남은 한 지점(원점)을 else 로 받아 낸다.

5.2 select case

하나의 값을 여러 후보와 대조한다

```
select case (expr)
   case (value-list)
      block
   [case (value-list)
      block]
   [case default
      block]
end select
```

if … else if 를 길게 늘어놓는 것보다

가독성이 좋고 컴파일러 최적화에도 유리하다. 처음으로 일치하는 case 블록 하나만 실행하고 end select 밖으로 빠져나간다. case default 는 if 문의 else 와 같은 역할이다.

case 후보 값의 다섯 가지 지정 방식

`case (3)`	정확히 그 값 하나와 일치
`case (1, 4, 9)`	나열된 후보 중 하나라도 일치
`case (80:89)`	80 이상 89 이하의 닫힌 범위
`case (90:)`	90 이상, 상한이 없는 열린 범위
`case (:59)`	59 이하, 하한이 없는 열린 범위

5.2 주의 사항

select case 를 쓸 수 없는 곳

1

실수형(real)은 사용 불가

값의 경계가 정확한 정수형 · 문자형 · 논리형 스칼라에서만 쓸 수 있다. 부동소수점 오차를 가지는 실수형은 case 비교 대상에서 배제된다.

2

case 안에는 오직 상수만

컴파일 시점에 값이 확정된 리터럴이나 parameter 명명 상수만 올 수 있다. 실행 중에 변하는 변수는 문법적으로 원천 봉쇄되어 있다.

3

범위 지정은 순서가 있는 형에서만

콜론(:)을 이용한 범위 지정은 정수형이나 알파벳 순서가 있는 문자형에서만 유효하다. 논리형에는 범위를 지정할 수 없다.

교집합 영역은 절대 금지 — case (80:89) 와 case (85:95) 처럼 겹치면 즉시 빌드 에러.

변수끼리 크기를 비교해야 한다면 select case 를 포기하고 if … else if 를 쓴다.

5.2 [예제]

점수를 등급으로, 코드를 요일로

정수 범위 분기 — 점수를 등급으로

```
integer, parameter :: score = 84

select case (score)
case (90:)      ; letter = "a"
case (80:89)    ; letter = "b"
case (70:79)    ; letter = "c"
case (60:69)    ; letter = "d"
case default    ; letter = "f"
end select
```

```
score 84 -> grade b
```

case (90:) 은 상한 없이 90점 이상을 모두 처리하고, case default 는 앞선 조건에 걸러지고 남은 60점 미만을 한 번에 모아 처리한다.

문자열 분기 — 코드를 요일로

```
character(len=3), parameter :: code = "wed"

select case (code)
case ("mon")          ; print *, "monday"
case ("wed")          ; print *, "wednesday"
case ("thu", "fri")   ; print *, "late week"
case default          ; print *, "weekend"
end select
```

```
wednesday
```

case ("thu", "fri") 처럼 쉼표로 여러 후보를 묶을 수 있다. 목요일이든 금요일이든 같은 블록으로 분기해 동일한 출력을 수행한다.

5.3 do 반복

횟수가 정해진 계수형 반복

```
[name:] do var = start, end [, step]
   block
end do [name]
```

반복 변수는 반드시 정수

레거시 표준에서는 실수형도 허용했으나, 부동소수점 오차로 반복 횟수가 틀어지는 버그 때문에 현대 표준에서는 정수형만 허용한다.

증감폭은 선택 사항

생략하면 1이 자동 적용된다. 음수를 넣어 값이 점차 작아지는 역방향 루프도 만들 수 있다.

루프 이름은 가독성을 위해

중첩 루프에서 어떤 do 문이 어디서 닫히는지 명확히 식별할 수 있어 실무에서 적극적으로 쓴다.

trip count — 반복 횟수는 진입 직전에 고정된다

```
(end - start + step) / step
```

한 번 고정되면, 루프 안쪽에서 start 나 end 로 쓰인 변수의 값을 바꾸더라도 이미 결정된 반복 횟수는 달라지지 않는다.

[흔한 실수] 실수 반복 변수

do x = 0.0, 1.0, 0.1 은 현행 표준에서 삭제된 기능이라 컴파일되지 않는다. 정수로 세고 x = real(i, real64) / real(n, real64) 처럼 값을 계산한다.

5.3 [예제]

반복의 기초와 계승 계산

1부터 5까지 2씩 증가

```
do i = 1, 5, 2
   print *, "현재 값: ", i
end do
```

```
 현재 값:  1
 현재 값:  3
 현재 값:  5
```

공식에 대입하면

(5 − 1 + 2) / 2 = 3 이므로 총 3번의 반복이 진입 직전에 고정된다. 그 결과 i 는 1, 3, 5 로 증가하며 블록을 수행한 뒤 루프를 탈출한다.

12의 계승 — 오버플로 방어

```
use iso_fortran_env, only: int64
integer(int64) :: result

result = 1_int64
do k = 2, n
   result = result * k
end do
```

정수 넘침 방어

계승은 숫자가 조금만 커져도 기하급수적으로 늘어난다. 기본 32비트는 약 21억이 한계라 금세 음수로 뒤집힌다.

표준 종류 지정 사수

integer*8 을 배제하고 iso_fortran_env 의 int64 를 써서 표준 규격과 이식성을 동시에 충족했다.

5.3 do while

횟수를 모를 때, 조건이 참인 동안

```
[name:] do while (logical-expr)
   block
end do [name]
```

매 바퀴 맨 위에서 평가

반복이 새로 시작되기 직전에 논리식을 평가하고, 참일 때만 내부 블록이 실행된다.

무한 루프 방지는 필수

블록 내부에 논리식의 결과를 변경하는 로직이 반드시 있어야 한다. 없으면 조건이 영원히 참으로 남는다.

수치 시뮬레이션에서 주로 쓰이는 두 상황

수치 모델의 수렴 판정

뉴턴-랩슨 기법이나 비선형 방정식의 반복 해법에서, 잔차(residual)가 허용 한계치 아래로 떨어져 값이 수렴할 때까지 반복하는 구조에 가장 적합하다.

입력 · 스트리밍 데이터 대기

외부 관측 장비나 파일로부터 실시간 데이터를 받아오거나, 사용자가 종료 명령을 내리기 전까지 대기하는 상시 구동 시스템에 유용하다.

5.3 [예제]

반감기 — 임계값 아래로 떨어질 때까지

```
real(real64), parameter :: threshold = 1.0e-3_real64

value = 1.0_real64;  steps = 0
do while (value > threshold)
   value = value * 0.5_real64
   steps = steps + 1
end do
```

```
below threshold after 10 halvings
```

매 바퀴 값을 반으로 자르다 보면 10번째에 $(1/2)^{10} = 1/1024 \approx 0.00097656$ 이 되어 임계값 0.001 보다 작아지므로, 조건이 거짓이 되며 루프를 무사히 탈출한다.

핵심은 부등호

종료를 판정하는 경계선을 == 을 사용한 동치 비교가 아니라 > 부등호로 설정한 것이 이 예제의 전부다.

[흔한 실수] 부동소수점 동치 종료

do while (value /= threshold) 처럼 동치 연산으로 반복을 제어하면 무한 루프에 빠질 수 있다. 소수점 아래 먼 자리가 10^{-17} 만큼만 어긋나도 컴퓨터는 "두 값은 다르다"고 판단해 영원히 탈출하지 못한다.

5.4 cycle 과 exit

루프 안에서 흐름을 비틀기

cycle

현재 반복의 남은 명령문을 전부 무시하고, 루프의 맨 위(다음 바퀴)로 즉시 이동한다.

exit

잔여 반복 수와 상관없이 반복문 자체를 즉시 빠져나간다.

[예제] 3의 배수 건너뛰기

```
integer, parameter :: n = 12, skip_factor = 3

do i = 1, n
   if (mod(i, skip_factor) == 0) cycle
   print '(i0)', i
end do
```

```
1  2  4  5  7  8  10  11
```

무슨 일이 일어났나

i 가 3, 6, 9, 12 일 때 조건이 참이 되어 cycle 이 실행된다. 아래 print 로 내려가지 않고 다음 바퀴로 넘어간다.

매직 넘버 제거

반복 한계선 n 과 건너뛸 인수 skip_factor 를 parameter 상수로 정의한 설계도 눈여겨보자.

5.4 명명 구문

중첩 반복을 한 번에 빠져나가기

아무 조치 없이 exit 나 cycle 을 부르면, 컴퓨터는 자신이 속한 가장 가까운 안쪽 루프 하나에만 작용한다.

```
search: do i = 1, n
   do j = 1, n
      if (i * j > threshold) then
         print *, "first pair:", i, j, i*j
         exit search
      end if
   end do
end do search
```

```
first pair: i=5, j=9, product=45
```

이름이 없었다면

가장 가까운 안쪽 j 반복만 탈출한 뒤, 바깥 i 반복으로 넘어가 i=6 부터 계산을 징검다리처럼 이어 갔을 것이다.

이름을 붙이면

exit search 는 안쪽 j 반복만이 아니라 이름이 search 인 바깥 i 반복까지 지체 없이 한 번에 벗어난다. 중첩이 깊어져도 설계 의도가 분명히 드러난다.

5.5 do concurrent

“이 루프는 순서가 중요하지 않다”는 선언

```
do concurrent (index = lower:upper [, mask-expr])
   block
end do
```

`index`	루프를 제어하는 반복 변수
`lower:upper`	계산을 수행할 정수 범위의 하한과 상한
`mask-expr`	선택 사항. 이 조건이 참인 인덱스 단계만 골라 실행하는 필터

컴파일러가 안심하고 자동 벡터화와 멀티 코어 병렬화를 최고 수준으로 적용할 수 있게 된다.

실행 순서 미보장

반복 변수가 순서대로 실행된다고 가정해서는 안 된다. 컴파일러는 하드웨어 환경에 따라 순서를 임의로 뒤섞거나 조율할 수 있다.

독립성 보장 — 의존성 금지

루프 내부에서 다른 반복 단계의 값을 참조하거나 수정하는 행위는 엄격히 금지된다. i-1 의 결과를 가져다 쓰면 정의되지 않은 동작이 된다.

5.5 [예제]

독립 반복의 형태, 그리고 금지 사항

```
do concurrent (i = 1:5)
   print '(a, i0, a, i0)', "i = ", i, ", i*i = ", i * i
end do
```

```
          i = 1, i*i = 1    ...    i = 5, i*i = 25
```

반복 간 의존성 제로

i 가 3일 때의 3*3=9 를 구하려고 i 가 1이나 2였을 때의 결과를 가져다 쓰지 않는다.

출력 순서의 무작위성

위 결과가 1~5로 정렬된 것은 루프가 작아 단일 스레드가 순식간에 처리했기 때문이다. 대규모 병렬에서는 순서가 전혀 보장되지 않는다.

블록 내부에 넣을 수 없는 것들

흐름 제어 명령 금지

return · exit · cycle 처럼 제어권을 강제로 쥐고 흔드는 문장

입출력 · 외부 호출 제한

print · write 같은 I/O 명령, 독립성이 검증되지 않은 외부 프로시저

데이터 의존성 유발 금지

계산 결과가 꼬리를 물고 이어지는 모든 형태의 의존성 수식

[흔한 실수] total = total + i 처럼 이전 결과 위에 누적하는 수식에는 절대 쓰지 않는다.

알아두기

거대 격자 연산에서 빛을 발한다

do concurrent 의 진정한 성능은 거대한 격자 공간의 물리량 배열을 요소별로 한 번에 후처리할 때 나타난다.

일반 do 루프

1번 격자점, 2번 격자점, 3번 격자점을 순서대로 계산한다. 앞의 루프가 끝날 때까지 뒤의 루프가 기다려야 한다.

순차 실행

do concurrent

"순서가 상관없다"고 선언하므로, 컴파일러가 격자 공간을 조각내어 여러 CPU 코어에 동시에 할당한다.

동시 실행

기상 수치 시뮬레이션의 예

3차원 격자점마다 온도 · 습도 · 풍속 같은 물리량 배열을 수백만 개씩 쌓아 두고 계산한다. 이 중 다른 격자점의 결과와 무관하게 그 지점의 데이터만으로 끝나는 작업(단위 변환 등)이 상당량 존재한다. 1번 코어는 북태평양 격자, 2번 코어는 아시아 대륙 격자를 나눠 맡아도 문제가 없는 구조다.

반복 간에 데이터를 주고받는 행위가 원천 차단되므로, 여러 코어가 한 메모리를 동시에 건드리는 데이터 경합(data race) 위험이 없다.

5.6 stop · error stop

루프가 아니라 프로그램을 멈춘다

`stop [stop-code]`

정상 종료

의도된 흐름에 따라 프로그램을 끝낸다. 코드를 생략하거나 0을 주면 운영체제에 정상 종료 상태(0)를 돌려준다.

`error stop [stop-code]`

오류 종료

예측하지 못한 결함이나 치명적 버그가 발생했음을 운영체제에 알린다. 0이 아닌 오류 상태 코드(흔히 1 이상)를 넘겨준다.

종료 코드는 운영체제와의 소통 수단이다

stop-code 는 정수 상수 또는 문자 상수여야 한다. 이 코드를 지정하는 이유는 프로그램을 실행한 주체(리눅스 셸 스크립트 등)에게 "이 프로그램이 어떤 상태로 끝났는지"를 명확히 전달하기 위해서다. 후속 자동화 스크립트는 이 값을 보고 다음 단계로 넘어갈지, 전체 공정을 중단하고 알림을 보낼지를 판정한다.

exit 는 반복을 벗어나고, stop 은 프로그램을 끝낸다 — 둘을 혼동하지 않는다.

5.6 [예제]

정상 종료와 오류 종료

조건 만족 시 정상 종료

```
do i = 1, items
   print '(a, i0)', "item ", i
   if (i == 2) then
      print '(a)', "stopping early"
      stop 0
   end if
end do
```

```
item 1    item 2    stopping early    STOP 0
```

루프 한계선은 3이지만 i 가 2가 되는 순간 stop 0 을 만난다. 남은 루프나 하단 코드와 상관없이 즉시 제어권을 내려놓는다.

잘못된 입력에서 오류 종료

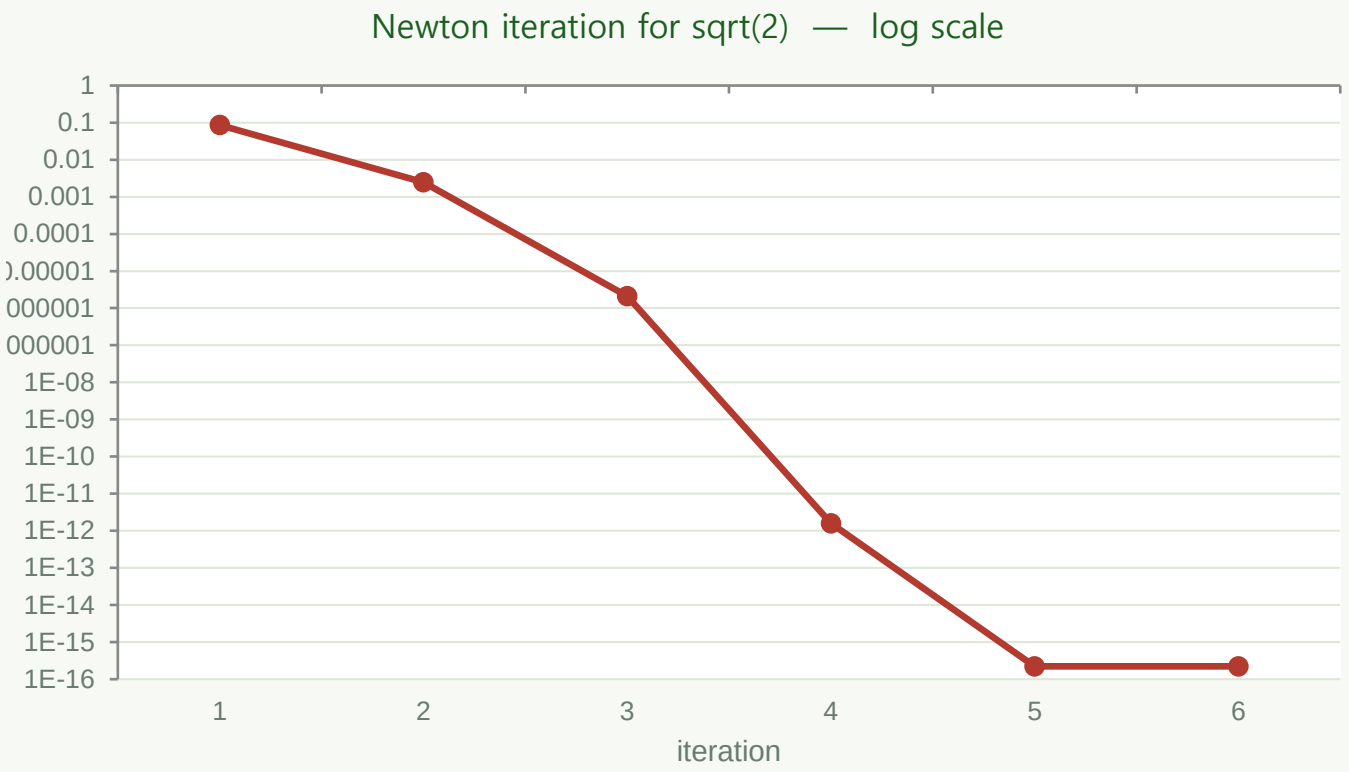
```
integer, parameter :: n = -5

if (n < 0) then
   print '(a)', "error: n must be non-negative"
   error stop 1
end if

print '(a, i0)', "processing n = ", n
```

```
error: n must be non-negative
ERROR STOP 1    Error termination.
```

방어적 데이터 검증 — 처리할 수 없는 값을 발견하면 명확한 원인을 출력한 뒤 급제동한다. 하단의 메인 연산 구문은 실행되지 않는다.

5.3 [예제] 도입 문제 2의 답

뉴턴 반복 — 여섯 바퀴 만의 수렴




```
do iter = 1, max_iter
   x_next = 0.5_real64 * (x + a / x)
   if (abs(x_next - x) < tol) exit
   x = x_next
end do
```

이중 안전장치

종료 판정은 두 근사값의 차이로 하고, 계수 do 의 max_iter 로 무한 루프를 막는다.

2차 수렴 (quadratic convergence)

한 바퀴마다 올바른 유효 자릿수가 대략 2배씩 늘어난다.

단 6바퀴 만에 배정밀도의 정밀도 한계선(10^{-15})을 넘어 수렴했다 — 수천 번을 돌아도 더딘 라이프니츠 급수와 대조된다.

오류 학습

실수 반복 변수의 금기와 정수 스케일링

금지된 코드

```
real(real64) :: x

do x = 0.0_real64, 1.0_real64, 0.1_real64
   print '(f6.2)', x
end do
```

```
Error: Deleted feature: Loop variable
  at (1) must be integer
```

왜 금지되었나

0.1 은 이진 부동소수점에서 무한소수라 미세한 절단 오차를 품은 채 저장된다. 이를 계속 누적해 더하면 마지막 바퀴에서 값이 1.0000000000000002 처럼 비대해져, 컴파일러가 끝값을 초과했다고 판단해 루프를 한 바퀴 일찍 끝내 버린다.

정석적인 해법 — 정수 스케일링

```
integer, parameter :: steps = 10
integer :: i
real(real64) :: x

do i = 0, steps
   x = real(i, real64) / real(steps, real64)
end do
```

```
0.00  0.10  0.20  ...  0.90  1.00
```

루프 수는 오차가 없는 정수로 세고, 원하는 실수 값은 매 바퀴마다 정수 인덱스로부터 다시 도출한다. 반복 횟수는 steps + 1 로 정확히 고정되고, x 는 이전 바퀴의 누적 오차에 영향받지 않는다.

5장 정리

요약

분기와 반복

- 논리 if (cond) statement — 한 문장만 조건 실행.
- 블록 if ... else if ... else ... end if — 처음 참인 가지 하나만 실행.
- select case — 정수·문자·논리 스칼라를 상수 후보와 비교. 범위와 다중값 가능, 중복 금지.
- 계수 do — var 는 정수, 반복 횟수는 진입 시 고정(trip count).
- do while — 횟수를 모르는 반복. 종료 조건은 부등호로.
- do concurrent — 의존성 없는 독립 반복, 실행 순서 미보장.

흐름 제어와 원칙

- cycle — 현재 반복의 남은 본문을 건너뛰고 다음 바퀴로.
- exit — 반복을 즉시 벗어남. 이름을 붙이면 바깥 반복까지 제어.
- stop — 정상 종료(상태 0 또는 지정 코드).
- error stop — 오류 종료(0이 아닌 상태).
- 부동소수점은 == 로 비교하지 말고 부등호·허용 오차로 판정한다.
- 반복 변수는 정수로 세고, 실수 값은 정수 인덱스로부터 계산한다.

한 줄 요약: 분기는 처음 참인 하나만, 반복은 정수로 세고, 종료는 부등호로 판정한다.

연습 문제

1. 정수 parameter 하나를 받아 짝수면 "even", 홀수면 "odd" 를 출력하는 프로그램을 if 로 작성한다.

2. 1부터 50까지의 합을 계수 do 로 구해 출력한다.

3. select case 로 1~7을 요일 이름으로 바꿔 출력한다. 범위 밖은 case default 로 "invalid".

4. do while 로 어떤 정수가 1이 될 때까지 2로 나눈 횟수를 센다. 나눠떨어지지 않으면 멈춘다.

5. 1부터 30까지 출력하되, 5의 배수는 cycle 로 건너뛴다.

6. 계수 do 로 2**0 부터 2**10 까지를 차례로 출력한다.

7. 1~100 중 3의 배수의 합과 5의 배수의 합을 각각 구해 출력한다 (if 와 mod 사용).

8. 중첩 do 로 구구단(2~9단)을 출력하되, 곱이 50을 넘는 항은 cycle 로 건너뛴다.

9. 2부터 차례로 나눠 보며 약수를 처음 찾으면 exit 로 멈춰 소수 여부를 판정한다.

10. 등비수열 부분합을 do while 로 누적하되, 새 항이 1.0e-6 보다 작아지면 멈춘다 (r = 0.5).

다음 장에서는 입출력 — 화면 출력과 파일 읽기·쓰기, 그리고 형식 지정을 다룬다.