제 1 부 — 기초

3장
자료형과 변수

What a Variable Really Holds

다섯 자료형 · kind와 정밀도 · parameter 상수 · 초기화

THE IDIOM

```
use iso_fortran_env, &
    only: real64
integer, parameter &
    :: dp = real64

real(dp) :: x
x = 0.1_dp
```

학습 목표

이 장에서 익히는 네 가지

01

다섯 가지 자료형

integer, real, complex, logical, character를 구분하고 각각의 선언법을 익힌다.

02

정밀도의 선택

종류 매개변수(kind)로 변수의 정밀도를 결정하고, 어느 시스템에서든 같은 결과를 얻는다.

03

상수 다루기

parameter 속성으로 변하지 않는 값에 이름을 붙여 매직 넘버를 없앤다.

04

올바른 초기화

변수를 선언과 동시에 초기화하는 방법과, 미초기화 변수의 위험을 익힌다.

도입 문제

이 장을 마치면 풀 수 있다

1

3.14 인가, pi 인가

반지름 2.5인 원의 넓이를 계산한다. area = 3.14 * r * r 와 area = pi * r**2 중 어느 쪽이 더 정확하며, 그 정확함은 변수의 무엇으로 결정되는가?

2

0.1을 만 번 더하면

어떤 프로그램은 9999.99…를, 어떤 프로그램은 10000.000000001을 출력한다. 무엇이 이 차이를 만드는가?

3

real*8 은 무엇인가

레거시 코드의 real*8 은 무슨 뜻이며 왜 표준이 아닌가? 다른 컴퓨터에서 돌려도 같은 정밀도를 얻으려면 어떻게 해야 하는가?

세 문제의 답은 모두 자료형(type) · 정밀도(precision) · 이식성(portability)에 있다.

알아두기

32비트와 64비트는 무엇이 다른가

1바이트 = 8비트는 전 세계 하드웨어의 표준 규격이라 바뀌지 않는다. 달라지는 것은 한 번에 처리하는 데이터의 기본 단위(워드)다.

32비트

4바이트

한 번에 4바이트씩 묶어 처리한다.

2^{32} 개의 주소 → 최대 4GB 메모리

숫자 하나를 처리하려면 연산 장치를 두 번 작동시켜야 했다.

64비트

8바이트

한 번에 8바이트씩 묶어 처리한다.

2^{64} 개의 주소 → 약 1,600만 TB

8바이트 데이터를 한 번의 명령으로 곧바로 처리한다.

같은 정밀도의 결과를 내더라도, 32비트는 일을 여러 단계로 나눠 하므로 연산이 훨씬 오래 걸린다. 초대형 행렬을 메모리에 통째로 올리는 슈퍼컴퓨팅은 64비트라서 가능하다.

3.1 내장 자료형

Fortran의 다섯 가지 내장 자료형

`integer`	**정수형**	부호 있는 정수	`-3  0  42`
`real`	**실수형**	부동소수점 실수	`3.14  -0.5`
`complex`	**복소수형**	실수부·허수부 쌍	`(3.0, -4.0)`
`logical`	**논리형**	참 / 거짓	`.true.  .false.`
`character`	**문자형**	글자·문자열	`"Texas"  "Busan"`

3은 정수, 3.0은 실수 — 같아 보여도 저장 방식과 연산 규칙이 다르다. · 복소수 (a, b) 는 수학의 a + bi 를 뜻한다. · character(len=20) 은 20개의 글자 칸 — 남으면 공백, 넘치면 경고 없이 잘린다.

3.1 변수 선언

자료형을 지정하고 :: 로 이름을 나열한다

```
integer             :: count
real                :: temperature
complex             :: impedance
logical             :: is_valid
character(len=20)   :: city
character           :: initial    ! 길이 생략 = 한 글자
```

구분자 ::

자료형과 변수 이름을 명확히 분리한다. 현대 Fortran에서 권장되는 선언 문법이다.

길이 생략

character :: c 처럼 길이를 적지 않으면 한 글자(길이 1)만 담는 변수가 된다.

문자형은 길이를 가진다 — character(len=20) 에 "Busan"을 넣으면 나머지 15칸은 공백으로 채워진다. 출력할 때 trim() 으로 뒤 공백을 떼어낸다.

3.1 [예제]

다섯 자료형을 선언하고 출력하기

```
program types_intro
   implicit none
   integer           :: count
   real              :: temperature
   complex           :: impedance
   logical           :: is_valid
   character(len=20) :: city

   count = 42;  temperature = 36.5
   impedance = (3.0, -4.0)
   is_valid  = .true.
   city      = "Busan"

   print *, "city =", trim(city)
end program types_intro
```

```
 count       =          42
 temperature =   36.5000000
 impedance   = (3.00000000,-4.00000000)
 is_valid    = T          city = Busan
```

logical 의 표기

값을 넣을 때는 양쪽에 마침표를 붙인 .true. / .false. 를 쓰고, 화면에는 T 또는 F 로 나온다.

complex 의 의미

(3.0, -4.0) 은 수학의 3.0 − 4.0i 를 나타낸다.

trim() 의 역할

문자열 뒤에 남은 빈 공백을 떼어내고 글자만 남기는 내장 함수다 (14장).

3.2 종류 매개변수(kind)

같은 real 이라도 정밀도가 다르다

단정밀도

real32

single precision

유효숫자 약 7자리

메모리 4 바이트

배정밀도

real64

double precision

유효숫자 약 15자리

메모리 8 바이트

과학·공학 계산은 배정밀도가 기본이다

복잡한 연산을 반복하면 누적 오차가 빠르게 커진다. 레거시에서 흔한 real*8 표기는 Fortran 표준이 아니므로, 최신 컴파일러에서 에러를 내거나 정밀도 문제를 일으킨다.

표준의 접근법: 바이트 크기를 직접 지정하지 말고 “몇 자리 정밀도와 어떤 지수 범위가 필요한가”를 선언하면, 컴파일러가 그 시스템에 맞는 종류를 찾아 준다.

3.2 kind 값을 얻는 두 가지 표준 방법

이식성 우선 · 명시성 우선

1 selected_real_kind(p, r)

이식성 우선

"유효숫자 p자리 이상, 십진 지수 범위 r 이상을 보장하는 종류를 달라"고 컴파일러에 요청하는 내장 함수다. 다른 컴퓨터·운영체제로 옮겨 컴파일해도 같은 정밀도를 보장받는다.

만족하지 못하면 음수를 돌려준다

- `-1` 정밀도 `p` 만 부족
- `-2` 지수 범위 `r` 만 부족
- `-3` 둘 다 부족

2 iso_fortran_env

명시성 우선

표준 라이브러리 모듈을 불러와 real32(4바이트)·real64(8바이트)처럼 비트 폭을 직접 지정한다. 다루는 데이터의 크기를 직관적으로 드러내므로 코드의 명시성이 높아진다.

특히 유용한 경우

- 다른 언어와 데이터를 주고받을 때
- 바이트 크기를 맞춰야 하는 바이너리 입출력

두 방식 모두 표준이다 — 정수형에는 selected_int_kind(r) 를 같은 방식으로 쓴다.

3.2 관용구

종류 값에 이름을 붙여 한 곳에서 관리한다

```
use iso_fortran_env, only: real64
integer, parameter :: dp = real64      ! 종류 값에 이름을 붙인다
real(dp) :: 변수목록                    ! 그 종류로 변수 선언

! 또는 정밀도·범위 요청 방식
integer, parameter :: dp = selected_real_kind(15, 307)
```

왜 숫자를 직접 쓰면 안 되는가

real(8) 처럼 종류 값을 하드코딩하면, 그 숫자 8이 8바이트를 뜻한다는 보장이 없다. 종류 식별 숫자는 컴파일러와 시스템에 따라 달라질 수 있기 때문이다. 반드시 real64 나 selected_real_kind 로 이름을 얻어 쓴다.

real64 는 사실 정수형 상수

이름 때문에 자료형으로 오해하기 쉽지만, 표준 모듈 안에 정의된 정수형 상수다. 대부분의 시스템에서 이 값은 8이다.

parameter 로 별명 붙이기

매번 real(real64) 라고 쓰는 건 번거롭고 실수를 부른다. dp 라는 상수로 고정해 두면 한 곳만 고쳐 전체 정밀도를 바꿀 수 있다.

dp 는 세계 공통의 약속

double precision 의 앞 글자를 딴 표준적인 약어로, 과학 계산 현업에서 널리 통용된다.

알아두기

과거에는 컴퓨터마다 계산 결과가 달랐다

"32비트와 64비트에서 계산한 값이 미세하게 달라 기상 예보가 틀어졌다" — 하드웨어 결함이 아니라 정밀도와 연산 특성 때문이다.

1

기본값 자료형의 크기 변화

32비트 시절에는 정밀도를 지정하지 않으면 실수를 4바이트 단정밀도로 처리하는 경우가 많았다. 64비트로 전환되면서 많은 컴파일러가 기본값을 8바이트 배정밀도로 올렸다. 소스 코드가 같아도 어디서 컴파일하느냐에 따라 유효숫자 자릿수가 달라진 것이다.

2

컴파일러 최적화와 반올림 차이

나눗셈·삼각함수·로그를 연산할 때 최하위 비트의 반올림 처리 방식은 컴파일러와 CPU 명령어 집합에 따라 미세하게 다르다. 기상 모델처럼 나비효과가 극단적으로 작용하는 영역에서는, 소수점 아래 수십 번째 자리의 오차도 수백만 번의 시간 전진을 거치며 눈덩이처럼 증폭된다.

그래서 selected_real_kind 나 real64 로 정밀도를 명시적으로 고정하는 선언부 작성이 필수다.

3.2 [예제]

kind 값을 직접 확인하기

```
program kind_demo
   use iso_fortran_env, only: real32, real64, int32, int64
   implicit none
   integer, parameter :: sp = selected_real_kind(6, 37)
   integer, parameter :: dp = selected_real_kind(15, 307)
   real(sp) :: x_single
   real(dp) :: x_double

   x_single = 1.0_sp / 3.0_sp
   x_double = 1.0_dp / 3.0_dp

   print *, "  sp =", sp, " real32 =", real32
   print *, "  dp =", dp, " real64 =", real64
end program kind_demo
```

```
 sp =  4   real32 =  4        dp =  8   real64 =  8
```

거대한 인덱스나 요소 개수를 다룰 때는 int64 를 써야 데이터 유실을 막는다.

종류 값의 일치

두 방식이 같은 하드웨어 표현형을 가리킨다. 다만 이 숫자 4나 8 자체를 코드에 하드코딩해서는 안 된다.

정수형의 한계도 함께 확인하자

int32 약 21억

2 147 483 647

int64 약 922경

9 223 372 036 854 775 807

3.2 조회 함수

형의 성질을 직접 물어본다

		단정밀도 real32	배정밀도 real64
precision	유효숫자 자릿수	6	15
range	십진 지수 범위	37	307
1.0 / 3.0	실제 계산 결과	0.333333343	0.33333333333333331
epsilon	구분 가능한 최소 간격	약 1.2E-07	2.2204460492503131E-016
huge	표현 가능한 최댓값	약 3.4E+38	1.7976931348623157E+308
tiny	표현 가능한 최소 양수	약 1.2E-38	2.2250738585072014E-308

epsilon 은 두 실수를 비교하거나 오차 범위를 정할 때의 물리적 기준을 알려 준다 — 배정밀도에서는 16번째 자리부터 구분이 무의미하다.

3.3 리터럴

코드에 그대로 적어 넣은 고정 데이터

`42`	기본 정수형
`3.14`	기본 단정밀도 실수
`3.14_dp`	배정밀도 실수 — 접미사를 붙인다
`(3.0, -4.0)`	복소수 — (실수부, 허수부)
`.true.  .false.`	논리 — 양옆에 마침표
`"Busan"`	문자 — 따옴표로 감싼다

[흔한 실수] 접미사가 없는 실수 리터럴

3.14 나 1.0 처럼 접미사가 없는 실수 리터럴은 컴파일러가 단정밀도로 해석한다. 배정밀도 변수에 대입하더라도 덜 정밀한 값이 먼저 만들어진 뒤 들어가므로, 대형 수치 계산에서 반올림 오차 버그의 주범이 된다. 고정밀도가 필요하면 반드시 3.14_dp 처럼 꼬리표를 붙인다.

3.3 명명 상수

매직 넘버를 없애는 parameter 속성

매직 넘버 (magic number)

3.141592, 110 처럼 의미를 알 수 없는 숫자가 코드 곳곳에 흩어져 가독성과 유지보수를 해치는 것.

명명 상수 (named constant)

한 번 정하면 실행 중 절대 바꿀 수 없는 값에 이름을 붙인다. 컴파일 시점에 값이 고정된다.

```
자료형, parameter :: 이름 = 초기식

real(dp), parameter :: pi     = 3.141592653589793_dp
real(dp), parameter :: two_pi = 2.0_dp * pi
character(len=*), parameter :: name = "Wave Solver"
```

바꾸려 하면 컴파일 오류

실행 도중 값을 바꾸려 시도하면 컴파일러가 즉시 차단한다.

상수끼리 조합할 수 있다

이미 값을 아는 다른 명명 상수를 연산식으로 대입할 수 있다 (two_pi = 2.0_dp * pi).

parameter 로 상수를 엄격히 정의하는 습관은 대규모 수치 모델에서 정확도와 수치적 안정성을 지키는 핵심이다.

3.3 [예제]

명명 상수로 원 넓이 구하기

```
program constants
   use iso_fortran_env, only: real64
   implicit none
   integer, parameter :: dp = real64

   real(dp), parameter :: pi     = 3.141592653589793_dp
   real(dp), parameter :: two_pi = 2.0_dp * pi
   integer,  parameter :: max_iter = 1000
   character(len=*), parameter :: app_name = "Wave Solver"

   real(dp) :: radius, area
   radius = 2.5_dp
   area   = pi * radius**2
end program constants
```

```
 pi    :    3.1415926535897931
 area  :    19.634954084936208
```

도입 문제 1의 답

3.14 라는 단정밀도 값을 그대로 썼다면 소수점 몇 자리 가지 못해 오차가 쌓인다. pi 를 배정밀도 명명 상수로 두면 같은 식으로 15자리까지 신뢰할 수 있는 넓이를 얻는다.

이 예제가 보여 주는 것

- dp 한 곳에서 정밀도를 관리
- pi 로 매직 넘버 제거
- 상수끼리 조합한 two_pi
- len=* 로 길이 자동 결정

3.4 변수 선언과 초기화

선언과 동시에 값을 채울 수 있다

```
자료형[, 속성목록] :: 이름[ = 초기식][, 이름2 = 초기식2 ...]

integer  :: i = 0
real(dp) :: total = 0.0_dp
integer, parameter :: n = 100
```

속성목록에 들어가는 것

- parameter — 바꿀 수 없는 상수
- allocatable — 동적 할당 (9장)
- dimension — 배열 크기 (7장)

미초기화 변수의 값은 0이 아니다

컴파일러·운영체제·당시 메모리 상태에 따라 그 방에 남아 있던 무작위 쓰레기 값이 들어갈 수 있다. 결과가 실행할 때마다 달라지는 함정에 빠지므로, 연산에 쓰기 전에 반드시 값을 지정한다.

단정밀도 리터럴을 넣으면 정밀도를 잃는다

real(dp) :: x = 0.1 로 선언하면 컴파일러가 0.1을 32비트로 먼저 만든 뒤 64비트 공간에 넣는다. 이미 버림 오차가 생겨 0.1이 정확히 들어가지 않는다. 반드시 0.1_dp 로 쓴다.

3.4 [예제]

같은 배정밀도 변수, 다른 결과

```
integer, parameter :: dp = real64
real(dp) :: wrong, right

wrong = 0.1      ! 단정밀도로 만들어진 뒤 넓혀짐
right = 0.1_dp   ! 처음부터 배정밀도

print *, "diff =", abs(wrong - right)
```

wrong	0.10000000149011612
right	0.10000000000000001
diff	1.4901161138336505E-009

두 변수 모두 같은 64비트 방에 선언되었는데도 담긴 값이 다르다.

right

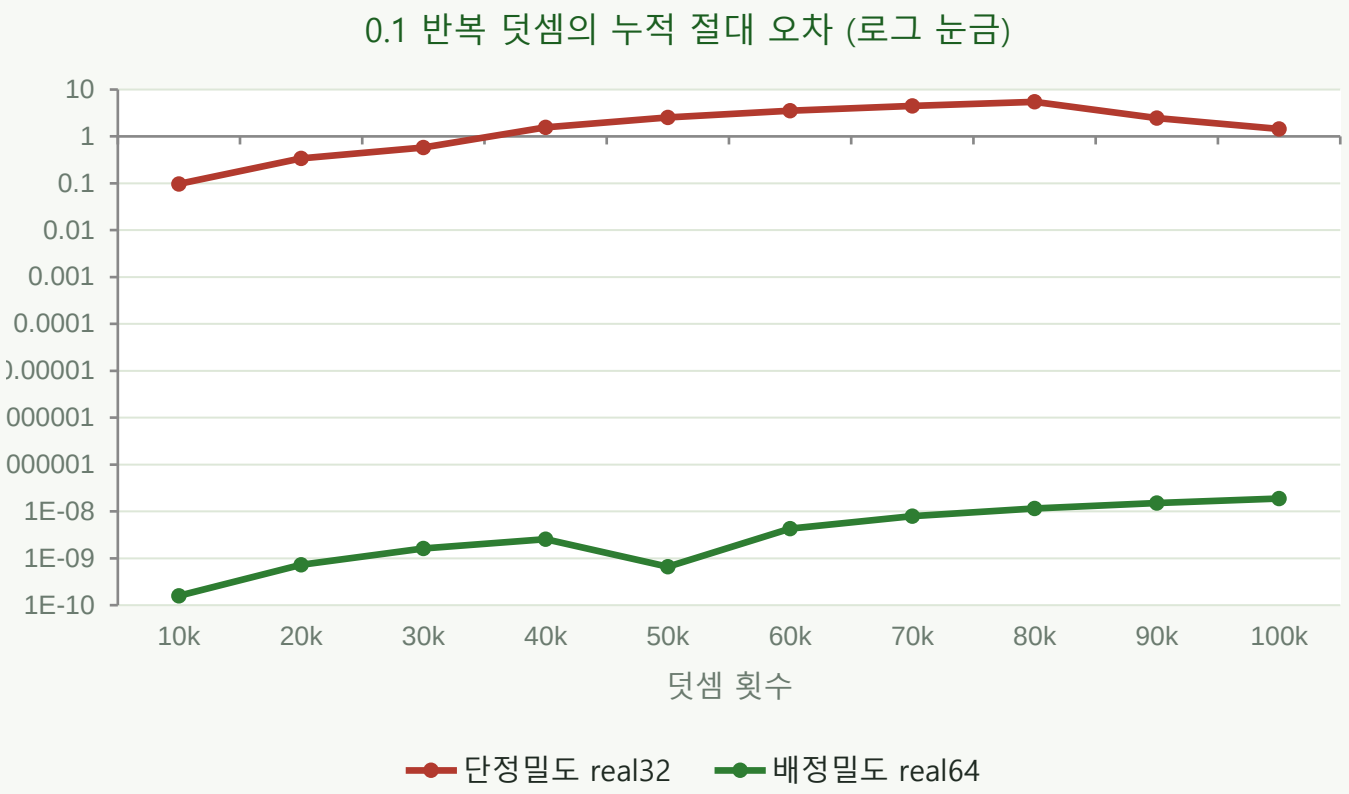
처음부터 배정밀도로 명시했기 때문에 소수점 16번째 자리 근처까지 오차 없이 0.1을 유지한다.

wrong

접미사를 빠뜨리는 순간 32비트로 잘린 값이 들어간다. 약 1.49E-9 의 틈이지만, 수억 번의 누적 루프에서는 눈덩이처럼 불어난다.

3.4 [예제] 정밀도의 실전 비교

0.1을 십만 번 더하면 어떻게 되는가



단정밀도

오차가 가파르게 우상향해, 십만 번을 더한 지점에서 절대 오차가 약 1.4에 이른다. 참값이 10000.0이므로 정수 자리를 통째로 틀린 것이다.

배정밀도

십만 번에 이를 때까지 오차가 10^{-8} 안팎의 미세한 영역에 안정적으로 묶여 있다.

세로축은 10의 거듭제곱 단위의 로그 눈금이다 — 두 곡선 사이의 간격이 정밀한 시뮬레이션에서 배정밀도를 기본으로 쓰는 이유다.

오류 학습

입문자가 자주 범하는 다섯 가지 실수

1 배정밀도 변수에 단정밀도 리터럴

x = 0.1 은 32비트로 먼저 만들어진다. 유실된 정밀도는 되찾을 수 없다.

```
x = 0.1_dp
```

2 비표준 real*8 표기법

컴파일러 개별 확장일 뿐 표준이 아니다. -std=f2018 에서 빌드가 차단된다.

```
real(dp) :: x
```

3 무리한 정밀도 요청

selected_real_kind(40) 은 -1 을 돌려주고, 컴파일러는 즉시 빌드를 멈춘다.

```
Kind -1 not supported
```

4 미초기화 변수를 0으로 짐작

선언만 한 방에는 이전의 쓰레기 값이 남아 있다. 누적 변수는 반드시 초기화한다.

```
total = 0.0_dp
```

5 정수 오버플로

int32 의 한계 2147483647 를 넘으면 값이 큰 음수로 뒤집히는데, 경고조차 없다.

```
int64 를 쓴다
```

가장 위험한 것은 4번과 5번이다 — 컴파일러가 아무 말도 해 주지 않는다.

알아두기

빌드 환경을 로그에 남겨라

```
use iso_fortran_env, only: compiler_version, compiler_options
print '(a)', compiler_version()
print '(a)', compiler_options()
```

compiler_version()

컴파일러의 정확한 이름과 버전을 문자열로 반환한다.

compiler_options()

적용한 최적화 옵션과 표준 준수 플래그를 반환한다.

과학 계산의 신뢰성을 확보하는 일종의 실험 노트 기록 습관이다.

```
====================================
[BUILD ENVIRONMENT METADATA]
Compiler Version : gcc 13.2.0
Compiler Options : -O2 -std=f2018 -Wall
Execution Date   : 2026-07-10 13:05:00
====================================

[SIMULATION RESULTS]
Step 100: Error = 1.4523E-08
```

왜 기록하는가

컴파일러 고유의 최적화에 따라 부동소수점 끝자리가 달라질 수 있다. 어떤 컴파일러의 몇 버전으로 어떤 옵션을 주었는지 남겨 두면 전 세계 어디서든 같은 결과를 재현할 수 있다.

3장 정리

요약

자료형과 정밀도

- 다섯 내장 자료형: integer, real, complex, logical, character.
- 실수의 정밀도는 kind 가 결정하며, iso_fortran_env 의 real64(배정밀도)를 기본으로 쓴다.
- selected_real_kind(p, r) 로 정밀도·범위를 요청할 수도 있다. 종류 값에는 dp 처럼 이름을 붙여 한 곳에서 관리한다.
- precision · range · epsilon · huge · tiny 로 형의 성질을 직접 확인할 수 있다.

리터럴과 변수

- 리터럴에도 종류가 있다 — 접미사 없는 3.14 는 단정밀도다. 배정밀도가 필요하면 3.14_dp.
- parameter 속성으로 명명 상수를 만들어 매직 넘버를 없앤다.
- 변수는 선언과 동시에 초기화할 수 있으나, 미초기화 값은 0 이 아니다.
- int32 의 한계는 약 21억 — 거대한 인덱스에는 int64 를 쓴다.

한 줄 요약: 정밀도는 kind 로 고정하고, 리터럴에도 _dp 를 붙이고, 상수는 parameter 로 이름을 준다.

연습 문제

1. 정수·단정밀도·배정밀도·논리형 변수를 하나씩 선언해 값을 넣고 모두 출력하라.

2. int8, int16, int32, int64 의 huge 값을 출력하라. 어떤 종류가 약 21억, 어떤 것이 약 922경인가?

3. z = (3.0_dp, 4.0_dp) 를 선언하고 real(z) · aimag(z) · abs(z) 를 출력하라.

4. pi 를 배정밀도 명명 상수로 두고 반지름 1.0·2.0·3.0 인 원의 넓이를 세 줄로 출력하라.

5. character(len=*) 명명 상수에 자기 이름을 담고 len 함수로 길이를 함께 출력하라.

6. selected_real_kind(6) 과 real32, selected_real_kind(15) 와 real64 가 같은 값인지 확인하라.

7. a = 0.1 (접미사 없이), b = 0.1_dp 를 넣고 a - b 를 출력하라. 0이 아닌 이유를 주석으로 쓰라.

8. precision 과 range 를 단·배정밀도에 적용해 자릿수와 지수 폭을 표 형태로 출력하라.

9. c = 36.5_dp 를 화씨로 바꿔 출력하라. 모든 리터럴에 _dp 를 붙여야 하는 이유를 적어라.

10. 5 / 2 와 5.0_dp / 2.0_dp 를 각각 출력하고, 두 결과가 다른 이유를 주석으로 설명하라.

다음 장에서는 연산자와 수식, 형변환과 연산자 우선순위를 다룬다.