

# Data Structures

Week 1 – Introduction, data and elementary data types

Dr. Ferenc Koczka · Eszterházy Károly University · 2026

# About the lecturer

---

- **Dr. Ferenc Koczka**, Assistant Professor
- Eszterházy Károly University, Eger, Hungary
- E-mail: [koczka.ferenc@uni-eszterhazy.hu](mailto:koczka.ferenc@uni-eszterhazy.hu)
- Consultation: after the classes, or by appointment via e-mail

“First, solve the problem. Then, write the code.” – John Johnson

# Course outline

---

| Week | Topic                               | Week | Topic                                      |
|------|-------------------------------------|------|--------------------------------------------|
| 1    | Introduction, elementary data types | 9    | Dynamic memory, linked list                |
| 2    | Efficiency, searching               | 10   | Sentinel, doubly linked and circular lists |
| 3    | Arrays                              | 11   | Dynamic stack and queue, hash table        |
| 4    | Strings                             | 12   | Recursion                                  |
| 5    | Sorting I (simple sorts)            | 13   | Sorting II (merge sort, quicksort)         |
| 6    | Set                                 | 14   | Trees, binary trees, traversals            |
| 7    | Stack                               | 15   | Binary search tree, heap, summary          |
| 8    | Queue                               | 16   | <b>Final test and practical exam</b>       |

Every week: **2 hours of lecture + 2 hours of practice** (Java and Go)

# Requirements and tools

---

## Assessment (week 16)

- **Final test** (theory, 90 minutes)  
true/false, multiple choice, code tracing, pseudocode
- **Practical exam** (on a computer)  
implementing a data structure in Java **or** Go

## Tools

- **Java** – JDK 17 or newer
- **Go** – 1.23 or newer
- Editor: VS Code, IntelliJ IDEA or Eclipse
- Example programs: `code/java/wXX` ,  
`code/go/wXX`

# Why do we study data structures?

## Algorithms + Data Structures = Programs

– Niklaus Wirth (1976)

- The same task is fast with a **good** data structure and unusably slow with a **bad** one
- Every programming language builds on these foundations ( `ArrayList` , `HashMap` , `Go slice` , `map` ...)
- A favourite topic of job interviews
- Later courses (databases, operating systems, artificial intelligence) assume this knowledge

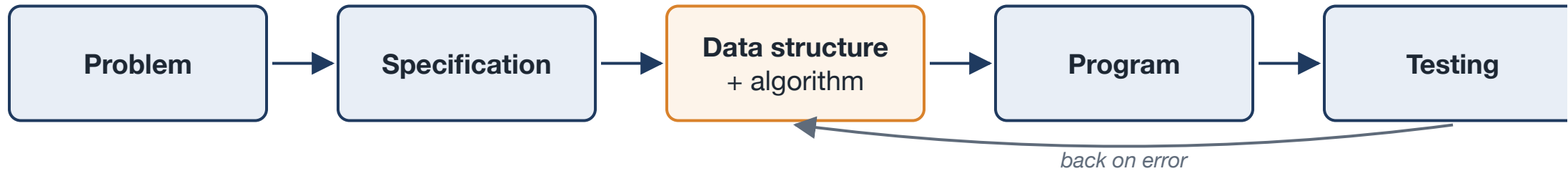
**Goal:** know the fundamental data structures, their operations and their efficiency – and be able to **implement** them.



# Problem solving and algorithms

# The steps of problem solving

---



- **Specification:** what is the input, what is the output, which conditions must hold?
- **Data structure + algorithm:** how do we store the data, and in which steps do we process it?
- This course is about the middle step

# The concept of an algorithm

---

**Algorithm:** a procedure consisting of a finite number of well-defined steps that solves a problem.

Its properties:

- **Finite** – it stops after a finite number of steps
- **Definite (deterministic)** – every step is precisely defined
- **Has input** – zero or more data items
- **Has output** – at least one result
- **Effective** – every step can actually be carried out
- **General** – it solves a whole class of problems, not just one case

# Ways to describe algorithms

---

| Tool              | Advantage                     | Disadvantage                  |
|-------------------|-------------------------------|-------------------------------|
| Plain text        | everybody understands it      | imprecise, long               |
| Flowchart         | visual                        | unreadable for large problems |
| <b>Pseudocode</b> | precise, language-independent | the notation must be learned  |
| Program code      | executable                    | language-specific details     |

In this course: **pseudocode** in the lectures and in the test, **Java and Go** in the practice classes.

# Pseudocode notation

| Notation                                                                                    | Meaning                                                            |
|---------------------------------------------------------------------------------------------|--------------------------------------------------------------------|
| <code>x ← expression</code>                                                                 | assignment                                                         |
| <code>If condition then ... else ... End if</code>                                          | selection                                                          |
| <code>While condition ... End while</code>                                                  | pre-test loop                                                      |
| <code>For i ← 1 to n ... End for</code>                                                     | counting loop                                                      |
| <code>Function Name(parameters) ... Return value ... End function</code>                    | function                                                           |
| <code>Procedure Name(parameters) ... End procedure</code>                                   | procedure (no return value)                                        |
| <code>A[i]</code>                                                                           | the <b>i</b> -th element of an array – <b>indexing starts at 1</b> |
| <code>and</code> , <code>or</code> , <code>not</code> · <code>div</code> , <code>mod</code> | logical operators · integer division, remainder                    |

In Java and Go, array indexing starts at **0** – always keep this in mind when writing code!

# Example: greatest common divisor (Euclid)

## Pseudocode

```
Function GCD(a, b)
  While b  $\neq$  0
    m  $\leftarrow$  a mod b
    a  $\leftarrow$  b
    b  $\leftarrow$  m
  End while
  Return a
End function
```

| a         | b   | a mod b |
|-----------|-----|---------|
| 1071      | 462 | 147     |
| 462       | 147 | 21      |
| 147       | 21  | 0       |
| <b>21</b> | 0   |         |

## Java

```
static int gcd(int a, int b) {
    while (b != 0) {
        int m = a % b;
        a = b;
        b = m;
    }
    return a;
}
```

## Go

```
func gcd(a, b int) int {
    for b != 0 {
        a, b = b, a%b
    }
    return a
}
```



# Data and data types

# Data and information

---

- **Data:** a recorded, processable form of some property of reality  
e.g. 37.2 , "Wuhan" , true
- **Information:** the meaning of the data in a given context  
e.g. “the patient's body temperature is 37.2 °C”
- The computer works with **data** – the meaning is given by the program (and the programmer)

## Properties of a data item

- **Name** (identifier) – how we refer to it: temperature
- **Type** – which values it can take, and what we can do with it
- **Value** – its current content
- **Address** – where it is located in memory

# The data type

---

A **data type** defines three things:

1. **Set of values** – which values it can take
2. **Operations** – what we can do with the values
3. **Representation** – how the machine stores it (how many bits, which encoding)

**Example – `int` (Java):**

values: `-2 147 483 648 ... 2 147 483 647` · operations: `+ - * / % == < ...` · representation:  
32 bits, two's complement

## Classification

- **Elementary (simple) type:** cannot be divided further – boolean, character, integer, real, pointer
- **Composite type (data structure):** built from elementary data – array, record, list ...

# Elementary types in Java and Go

| Kind                | Java                                                      | Go                                                                                                  | Size              |
|---------------------|-----------------------------------------------------------|-----------------------------------------------------------------------------------------------------|-------------------|
| boolean             | <code>boolean</code>                                      | <code>bool</code>                                                                                   | –                 |
| character           | <code>char</code> (UTF-16 code unit)                      | <code>rune</code> (= <code>int32</code> , Unicode code point)                                       | 16 / 32 bits      |
| byte                | <code>byte</code> (signed)                                | <code>byte</code> (= <code>uint8</code> , unsigned)                                                 | 8 bits            |
| integer             | <code>short</code> , <code>int</code> , <code>long</code> | <code>int8</code> , <code>int16</code> , <code>int32</code> , <code>int64</code> , <code>int</code> | 16 / 32 / 64 bits |
| unsigned integer    | –                                                         | <code>uint8</code> ... <code>uint64</code> , <code>uint</code>                                      | 8 ... 64 bits     |
| real                | <code>float</code> , <code>double</code>                  | <code>float32</code> , <code>float64</code>                                                         | 32 / 64 bits      |
| pointer / reference | reference (to objects)                                    | <code>*T</code> pointer                                                                             | 64 bits           |

In Go the size of `int` depends on the machine (almost always 64 bits today); in Java `int` is always 32 bits.

# Representing integers

|        |   |   |   |   |   |   |   |   |
|--------|---|---|---|---|---|---|---|---|
| 5 =    | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 1 |
| -5 =   | 1 | 1 | 1 | 1 | 1 | 0 | 1 | 1 |
| 127 =  | 0 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| -128 = | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

sign bit

byte: 8 bits

range: -128 ... 127

negation: invert bits + 1

- With  $n$  bits,  $2^n$  different values can be represented
- Signed integer (two's complement):  $-2^{n-1} \dots 2^{n-1} - 1$
- Java `int` (32 bits): -2 147 483 648 ... 2 147 483 647

# Overflow

If the result of an operation does not fit into the type, the value **“wraps around”** – without any error message!

Java

```
int max = Integer.MAX_VALUE;  
System.out.println(max + 1);  
// -2147483648  
  
byte b = 127;  
b++;           // -128
```

Go

```
var max int32 = math.MaxInt32  
max++  
fmt.Println(max)  
// -2147483648  
  
var b int8 = 127  
b++           // -128
```

Always choose a type in which even the largest expected value fits (e.g. `long` / `int64` ).

# Real numbers: floating-point representation

- Form: **sign** × **mantissa** × **2<sup>exponent</sup>** (IEEE 754 standard)
- `float` / `float32` : about 7 significant digits · `double` / `float64` : about 15–16 significant digits
- Most decimal fractions (e.g. 0.1) **cannot be represented exactly** in binary

```
double s = 0.1 + 0.2;  
System.out.println(s);  
// 0.30000000000000004  
System.out.println(s == 0.3); // false
```

**Solution:** compare with a tolerance

```
final double EPS = 1e-9;  
Math.abs(s - 0.3) < EPS    // true
```

For financial calculations we use integers (e.g. fen, cents) or the `BigDecimal` type.

# Integer division and type conversion

- The quotient of two integers is an **integer**: `7 / 2 == 3`, the remainder: `7 % 2 == 1`
- If at least one operand is real, the result is real: `7 / 2.0 == 3.5`

```
double f = 9 / 5 * celsius + 32;    // WRONG: 9 / 5 == 1
double f = 9.0 / 5 * celsius + 32;  // correct
```

## Conversion (type casting):

- **Widening** (`int` → `long`, `int` → `double`): safe, automatic in Java
- **Narrowing** (`double` → `int`, `int` → `byte`): data may be lost, only when explicitly requested  
`(int) 3.99 == 3` (truncates) · `(byte) 300 == 44` (only the lowest 8 bits remain)
- In Go **every** conversion must be written out: `float64(a) / float64(b)`

# Characters: ASCII and Unicode

- **ASCII:** 7 bits, 128 characters (English alphabet, digits, punctuation)  
`'A' = 65` , `'a' = 97` , `'0' = 48` → `'a' - 'A' == 32`
- **Unicode:** all writing systems of the world, > 150 000 characters  
`'中' = U+4E2D = 20013` · `'é' = U+00E9 = 233`
- **UTF-8:** variable-length encoding (1–4 bytes) – the default on the web and in Go

## Java

```
char c = 'A';  
System.out.println((int) c);    // 65  
char d = (char) (c + 1);        // 'B'
```

## Go

```
c := 'A'                // rune  
fmt.Println(c)           // 65  
fmt.Printf("%c", '中')   // 中
```

Strings are covered in detail in **week 4**.

# The boolean type and logical operations

| a | b | a and b | a or b | not a |
|---|---|---------|--------|-------|
| F | F | F       | F      | T     |
| F | T | F       | T      | T     |
| T | F | F       | T      | F     |
| T | T | T       | T      | F     |

| Operation    | Java | Go |
|--------------|------|----|
| and          | &&   | && |
| or           |      |    |
| not          | !    | !  |
| exclusive or | ^    | != |

- **Short-circuit evaluation:** in `a && b`, if `a` is false, `b` is not evaluated at all  
`if (i < n && t[i] > 0)` – never indexes past the end of the array!

# Literals, variables, constants

---

- **Literal:** a value written directly into the program text

42 , 0x2A , 0b101010 , 3.14 , 2.5e3 , 'A' , "hello" , true

- **Variable:** a named storage location whose value can change while the program runs
- **Constant:** a named value that **cannot change**

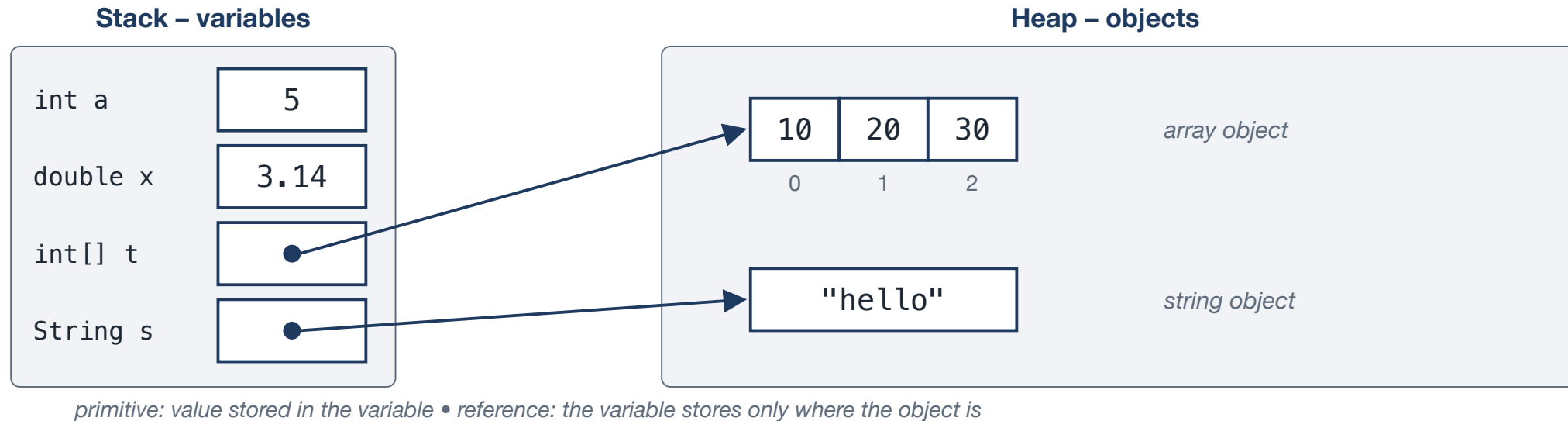
## Java

```
int count = 0;           // variable
final int DAYS = 7;      // constant
long big = 9_000_000_000L; // L: long
float f = 2.5f;          // f: float
```

## Go

```
var count int           // 0 is the zero value
n := 10                  // short declaration
const Days = 7           // constant
var f float32 = 2.5
```

# Pointers and references



- **Pointer:** a data item whose value is a **memory address**
- With a pointer we access another data item indirectly
- Special value: **null / nil** – “points nowhere”

# References in Java, pointers in Go

## Java

- Primitive type ( `int` , `double` ...): the variable stores **the value**
- Object (array, `String` , own class): the variable stores a **reference**
- No pointer arithmetic, the address is not visible

```
int[] a = {10, 20, 30};
int[] b = a;      // the same array!
b[0] = 99;
System.out.println(a[0]); // 99
```

## Go

- Every type is copied by value – even an array ( `[3]int` )
- Pointer: `&x` is the address, `*p` is the value pointed to
- A slice ( `[]int` ) and a `map` contain a reference internally

```
x := 5
p := &x      // p stores the address
*p = 6
fmt.Println(x) // 6
```

Pointers/references make **dynamic data structures** possible (linked list, tree) – see from week 9.



# Data structures

# What is a data structure?

---

A **data structure** is a collection of logically related data elements, defined by

1. its **elements** (and their types),
2. the **relationships** between the elements,
3. the **operations** that can be performed.

**Example – a class list:** elements: names (strings) · relationship: alphabetical order (everybody has a predecessor and a successor) · operations: search, insert a new name, delete a name

Data structures are classified according to several criteria ►

# Classification: type and number of elements

---

## By the type of elements

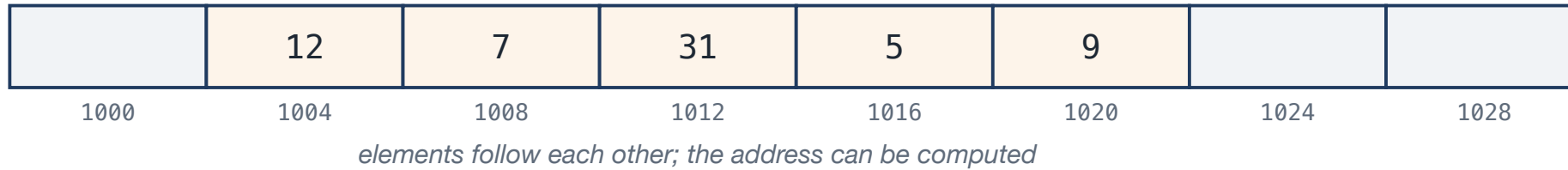
- **Homogeneous:** all elements have the same type – array, string, set
- **Heterogeneous:** the elements may have different types – record ( `class` / `struct` )

## By the number of elements

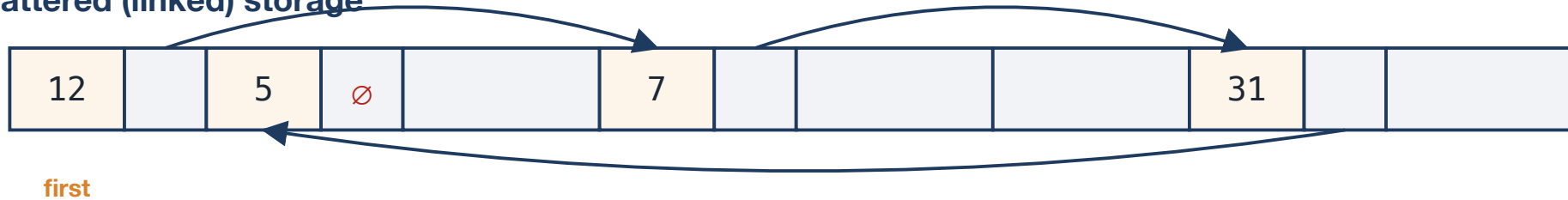
- **Static:** the number of elements is fixed at creation and does not change later  
Java array `new int[10]` , Go array `[10]int`
- **Dynamic:** the number of elements can change at run time  
linked list, `ArrayList` , Go `slice` , tree

# Classification: way of storage

## Contiguous storage



## Scattered (linked) storage



- **Contiguous:** the address of the  $i$ -th element can be computed:  $\text{base address} + (i - 1) \cdot \text{element size}$
- **Scattered:** each element stores the location of the next one (pointer) – no shifting needed when inserting

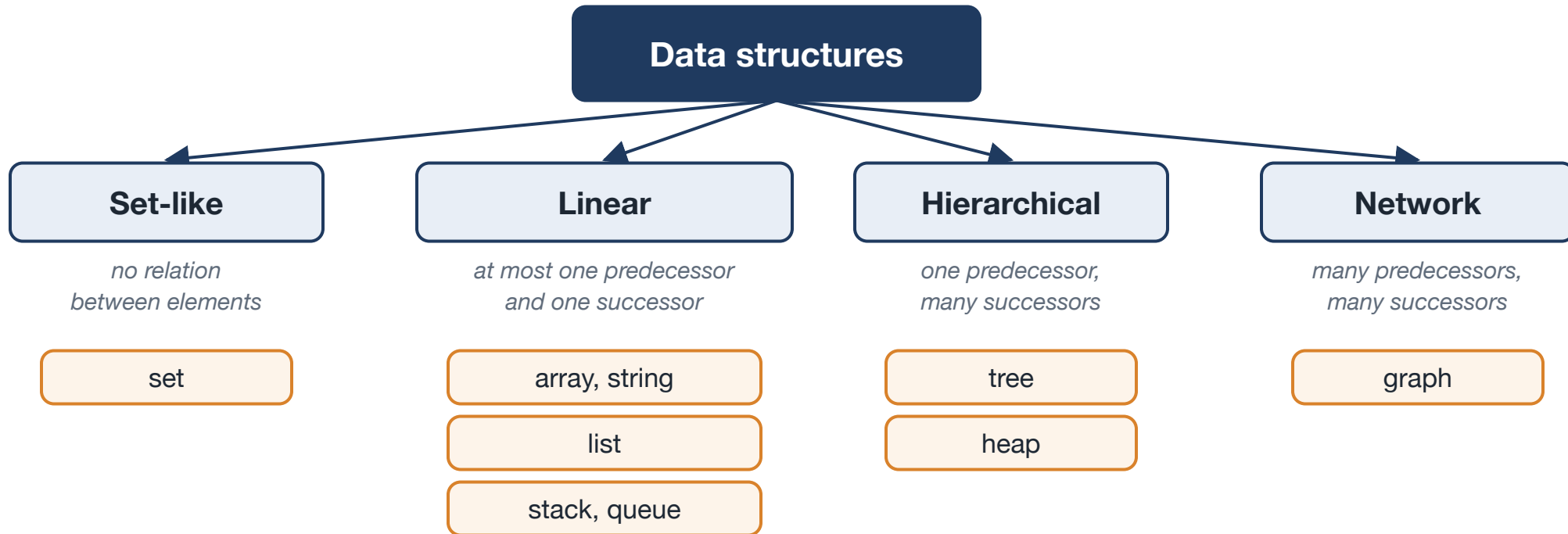
# Classification: access to the elements

---

- **Direct (random) access:** every element can be reached in the same time, by index  
array: `t[1000]` is as fast as `t[0]`
- **Sequential access:** an element can only be reached by passing through the ones before it  
linked list, (magnetic tape)
- **Associative (content-based) access:** an element is reached by a key  
hash table: `map["Wuhan"]`
- **Restricted access:** only a specific element is accessible  
stack (only the top), queue (only the front)

# Classification: relationships between elements

---



# Overview: the data structures of the course

---

| Data structure | Elements      | Size     | Storage                 | Access              | Relationship | Week     |
|----------------|---------------|----------|-------------------------|---------------------|--------------|----------|
| array          | homogeneous   | static   | contiguous              | direct              | linear       | 3        |
| string         | homogeneous   | static*  | contiguous              | direct              | linear       | 4        |
| set            | homogeneous   | variable | several                 | –                   | none         | 6        |
| stack, queue   | homogeneous   | dynamic  | contiguous or scattered | restricted          | linear       | 7, 8, 11 |
| linked list    | homogeneous   | dynamic  | scattered               | sequential          | linear       | 9, 10    |
| hash table     | homogeneous   | dynamic  | mixed                   | associative         | none         | 11       |
| binary tree    | homogeneous   | dynamic  | scattered (or array)    | sequential          | hierarchical | 14, 15   |
| record         | heterogeneous | static   | contiguous              | direct (field name) | –            | –        |

\* In Java and Go strings are immutable.

# Operations on data structures

---

| Operation              | Meaning                                  |
|------------------------|------------------------------------------|
| Creation               | producing an empty structure             |
| Insertion              | adding a new element                     |
| Deletion               | removing an element                      |
| Modification           | changing the value of an element         |
| Search                 | finding an element with a given property |
| Traversal (processing) | visiting every element once              |
| Sorting                | putting the elements in order            |

A data structure is good for a task if its **most frequent operations are fast**.

# Abstract data type (ADT)

---

- **ADT**: a data structure defined by its **operations** – **what** it can do, not **how**
- The implementation (representation) can be hidden and replaced

## Java – interface

```
interface Stack {  
    void push(int x);  
    int pop();  
    boolean isEmpty();  
}
```

## Go – interface

```
type Stack interface {  
    Push(x int)  
    Pop() int  
    IsEmpty() bool  
}
```

The same stack can be implemented with an **array** (week 7) or a **linked list** (week 11) – the user's code does not change.

# Summary

---

- Programs = **algorithms + data structures**
- An algorithm is a finite, well-defined sequence of steps; we describe it in **pseudocode** (indexing from 1)
- A **data type** = set of values + operations + representation
- Representations have limits: **overflow, floating-point inaccuracy, integer division**
- A **pointer/reference** stores a memory address
- Classification of data structures: homogeneous/heterogeneous · static/dynamic · contiguous/scattered · direct/sequential access · linear/hierarchical/network

**Next week:** How do we measure the efficiency of an algorithm? Linear and binary search.

# Review questions

---

True or false?

1. An array is a heterogeneous data structure.
2. The elements of a linked list are accessed sequentially.
3. In Java, `Integer.MAX_VALUE + 1` causes a run-time error.
4. In Java, the value of the expression `0.1 + 0.2 == 0.3` is false.
5. The value of a pointer is a memory address.
6. The values of the elements of a static data structure cannot change.



# Practice

# Setting up the environment

---

## Java

```
java -version          # 17 or newer
cd code/java/w01
java Types.java        # compile + run
```

or the classic way:

```
javac Types.java
java Types
```

## Go

```
go version             # 1.23 or newer
cd code/go
go run ./w01/types
```

Formatting and checking:

```
gofmt -w .
go vet ./...
```

## Demo programs

| Program             | Java                       | Go                        | What does it show?                                   |
|---------------------|----------------------------|---------------------------|------------------------------------------------------|
| Elementary types    | <code>Types.java</code>    | <code>w01/types</code>    | ranges, literals, character codes, constants         |
| Pitfalls            | <code>Pitfalls.java</code> | <code>w01/pitfalls</code> | overflow, integer division, $0.1 + 0.2$ , conversion |
| Value and reference | <code>ValueRef.java</code> | <code>w01/valueref</code> | parameter passing, aliasing, null/nil, swap          |
| Euclid              | <code>Gcd.java</code>      | <code>w01/gcd</code>      | pseudocode → code, printing the steps                |

Run each of them, and **before** running, guess what it will print! We will discuss the surprising lines.

## Exercises

---

1. **Number of bits:** write a function that returns how many bits are needed to store an integer  $n \geq 0$ ! ( 255  $\rightarrow$  8 , 256  $\rightarrow$  9 )
2. **Binary form:** print the binary form of a positive integer by repeated division by 2! ( 50  $\rightarrow$  110010 )
3. **Upper case:** convert a lower-case letter to upper case using its character code (without library functions)!
4. **Comparing reals:** write an `almostEqual(x, y)` function with a tolerance!
5. **Truth table:** print the truth tables of `and` , `or` , `not` and `exclusive or` !
6. **Overflow check:** decide **before adding** whether `x + y` would overflow ( `int` / `int32` )!

Solutions: `Exercises01.java` , `w01/exercises`

## Homework

---

1. What is the value of `(byte) 200` in Java, and of `int8(200)` in Go? Compute it by hand, then check it!
2. How many different values can a 16-bit unsigned integer store? What is its largest value?
3. Write a program that prints every character of the text `"Hello, 世界"` together with its Unicode code!
4. Find everyday examples of a **stack**, a **queue** and a **tree** structure!