Chapter 1: Introduction

Data Structures

Digital data | Data types | Java memory model | Generics | Abstract data types | Choosing a structure | Fundamental structures

Sercan KÜLCÜ · Giresun University · Computer Engineering

INTRODUCTION

Learning Outcomes

Core concepts

Tell a data type, an abstract data type, and a data structure apart.

Choosing a structure

Choose a data structure that fits a problem's requirements.

ADTs in Java

Write an ADT as an interface and a data structure as a class; use references and generics correctly.

Fundamental structures

Know how each fundamental structure organizes data and where it is used.

Digital Data

- Numbers, text, images, sound: any information stored as 0s and 1s.
- **Examples**: text documents, image files, databases, audio recordings.



Analog sound → ADC → 0s and 1s → computer → DAC → analog sound

DIGITAL DATA

Why Digital?

Numeric

Everything is stored in binary.

`1011 0010`

Easy to store

Kept electronically, shared with ease.

`disk · cloud · network`

Fast to process

Computers work through it at high speed.

`~10⁹ operations / second`

A computer sees only 0s and 1s. Data structures arrange those bits so they are **meaningful and efficient** to use.

DIGITAL DATA

Representing Digital Data

Text

Character encodings such as ASCII and Unicode.

`'A' → 65 → 01000001`

Numbers

Integers, floating-point numbers.

`13 → 00001101 · 3.14 → IEEE 754`

Images

A color value for each pixel.

`red pixel → (R, G, B) = (255, 0, 0)`

Sound

Digital samples of the sound wave.

`CD quality → 44,100 samples per second`

Kinds of Digital Data

Kind	Description	Examples
Structured	organized by a fixed schema	relational database tables, CSV
Semi-structured	partly organized by tags	XML, JSON
Text	text meant for people to read	TXT, email
Images and graphics	visual data	JPG, PNG
Audio and music	sound recordings	MP3, WAV

Data Structures and Algorithms

Algorithm

- A step-by-step plan for carrying out a task.
- A cornerstone of computer science.

Data Structure

- A way of organizing data that fixes how it is stored.
- The groundwork for managing and processing data efficiently.

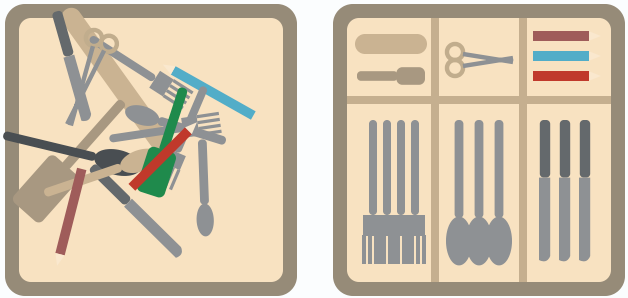
- Good algorithms and data structures make software efficient.
- Together they solve problems systematically.

Algorithms + Data Structures = Programs

Title of Niklaus Wirth's 1976 book

Data Structures

- A way to organize and store data in a computer.
- Goal: fast, efficient access to the data.
- Essential knowledge for any software developer.
- The right structure makes a program faster and leaner.

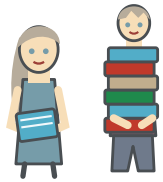


Same utensils: jumbled on the left, sorted on the right.
Where would you find what you need faster?

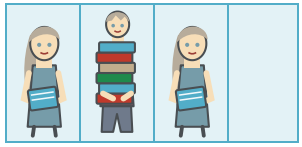
Basic Building Blocks

- **Objects** group related attributes into one unit with named fields.
- **Arrays** store items side by side in consecutive memory slots.
- **Links** let items point to one another in any directed pattern.

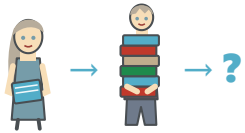
Objects



Arrays



Links



Primitive Data Types

Java type	Size	Range
`byte`	8 bits	−128 … 127
`short`	16 bits	−32,768 … 32,767
`int`	32 bits	about ±2.1 billion
`long`	64 bits	about $\pm 9.2 \cdot 10^{18}$
`float`	32 bits	floating point (≈7 digits)
`double`	64 bits	floating point (≈15 digits)
`boolean`	—	true / false
`char`	16 bits	UTF-16 code unit

- Integers, floating-point numbers, booleans, characters.
- The simplest building blocks.
- Fixed size in memory.
- Handled directly by the processor.

Data Types

- **Data type**: a set of values plus their behavior.
- **Value set**: the legal values of the type.
- **Behavior**: the operations allowed on those values.
- Not every operation makes sense for every type.
- **Type safety**: types keep errors down; their restrictions tell the programmer what is allowed.

Integers

Addition, subtraction, multiplication, division.

```
7 / 2 → 3
```

Strings

Comparison, concatenation, substrings.

```
"data" + "base"
```

Invalid operation

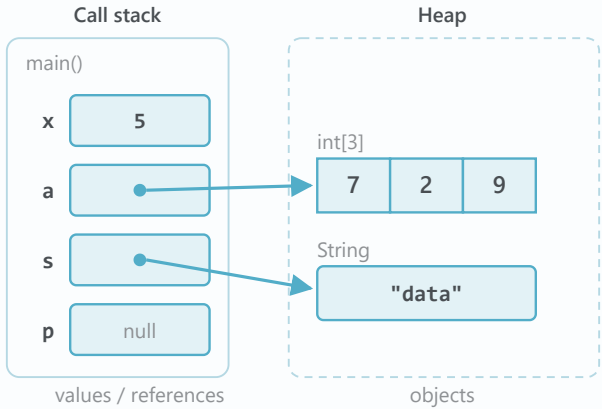
The compiler rejects it before the program runs.

```
"abc" * 2 → compile error
```

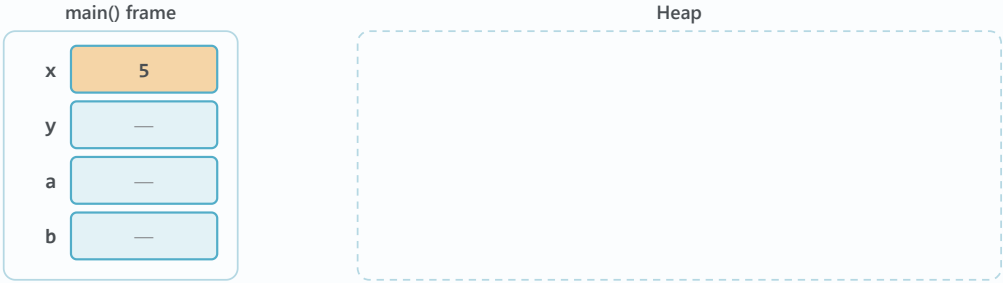
Memory in Java: Stack and Heap

- A **primitive** variable holds its value directly.
- A **reference** variable holds the **address** of an object on the heap; `new` creates the object.
- Each method call gets a stack **frame** for its local variables, discarded on return.
- The **garbage collector** reclaims unreachable objects; `p[0]` → `NullPointerException`.

```
int x = 5;                // primitive: the value
int[] a = {7, 2, 9};      // reference → array
String s = "data";        // reference → object
int[] p = null;           // refers to nothing
```



Code Trace: Value or Reference?



x = 5: a primitive; the value lives in the frame itself.

Step 1 / 8

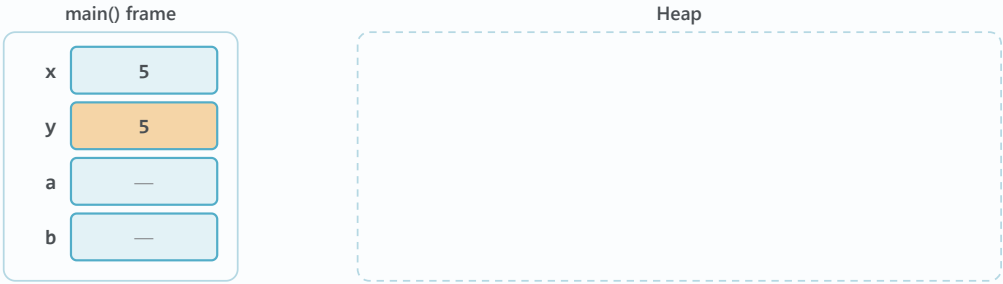
```
▶ int x = 5;
2 int y = x;
3 y = 7;
4 int[] a = {1, 2, 3};
5 int[] b = a;
6 b[0] = 9;
7 b = new int[2];
8 System.out.println(x + " " + a[0]);
```

VARIABLES

x = 5

OUTPUT

Code Trace: Value or Reference?



y = x: copies the **value**; x and y are independent.

Step 2 / 8

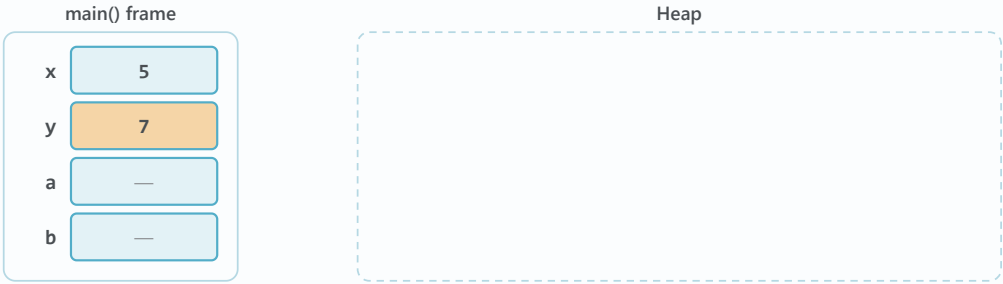
```
1 int x = 5;
▶ int y = x;
3 y = 7;
4 int[] a = {1, 2, 3};
5 int[] b = a;
6 b[0] = 9;
7 b = new int[2];
8 System.out.println(x + " " + a[0]);
```

VARIABLES

x = 5 y = 5

OUTPUT

Code Trace: Value or Reference?



y = 7: only y changes; x is still 5.

Step 3 / 8

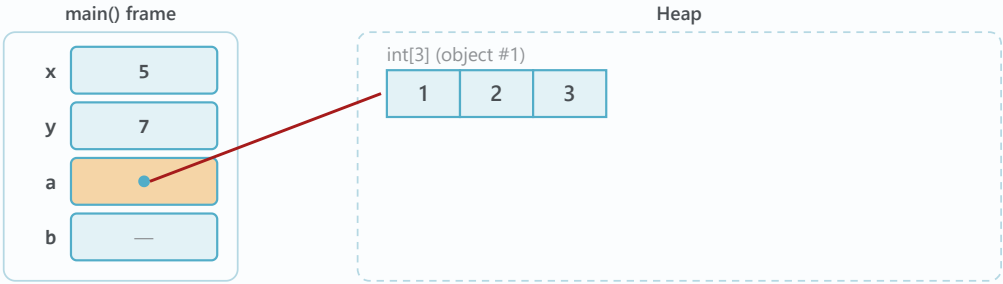
```
1 int x = 5;
2 int y = x;
▶ y = 7;
4 int[] a = {1, 2, 3};
5 int[] b = a;
6 b[0] = 9;
7 b = new int[2];
8 System.out.println(x + " " + a[0]);
```

VARIABLES

x = **5** y = **7**

OUTPUT

Code Trace: Value or Reference?



The array lives **on the heap**; **a** holds only a reference to it (the arrow).

Step 4 / 8

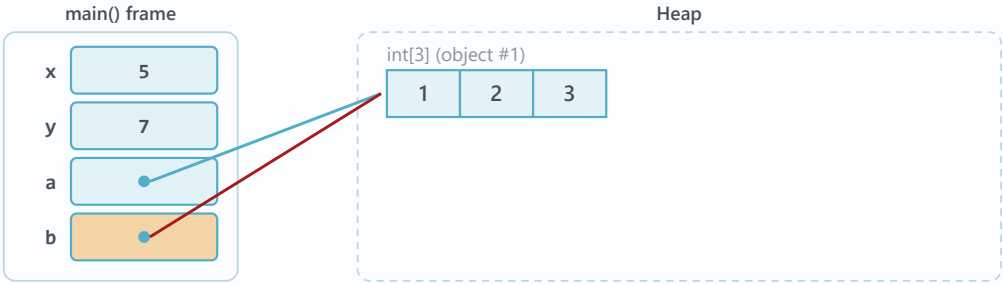
```
1 int x = 5;
2 int y = x;
3 y = 7;
▶ int[] a = {1, 2, 3};
5 int[] b = a;
6 b[0] = 9;
7 b = new int[2];
8 System.out.println(x + " " + a[0]);
```

VARIABLES

x = 5 | y = 7 | a = →#1 [1, 2, 3]

OUTPUT

Code Trace: Value or Reference?



b = a: copies the **reference**, not the array; both arrows point to one object.

Step 5 / 8

```
1 int x = 5;
2 int y = x;
3 y = 7;
4 int[] a = {1, 2, 3};
▶ int[] b = a;
6 b[0] = 9;
7 b = new int[2];
8 System.out.println(x + " " + a[0]);
```

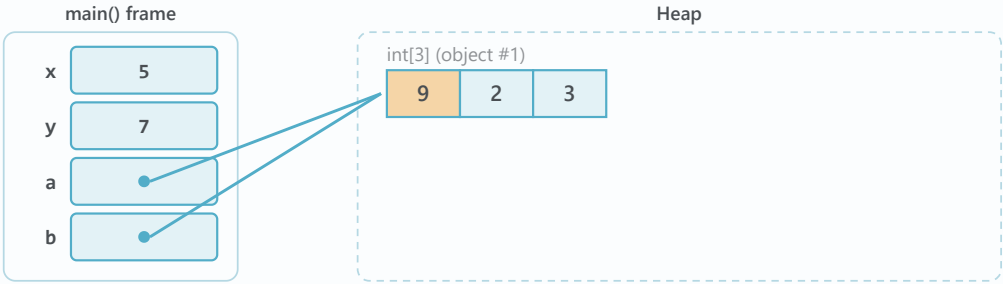
VARIABLES

`x = 5` `y = 7` `a = →#1 [1, 2, 3]`

`b = →#1 [1, 2, 3]`

OUTPUT

Code Trace: Value or Reference?



b[0] = 9: changes the object itself, so **a[0] is 9 too** (aliasing).

Step 6 / 8

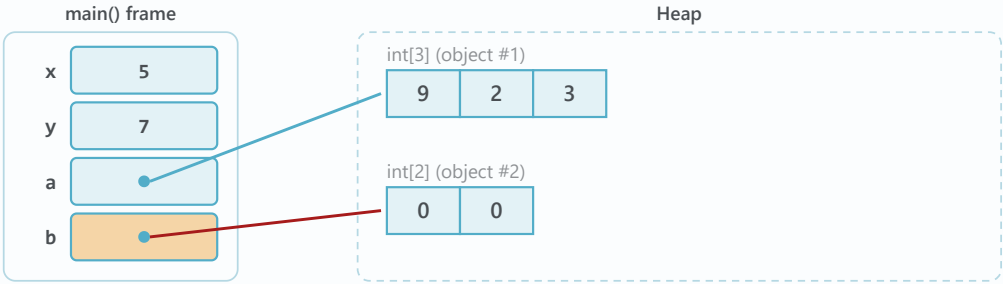
```
1 int x = 5;
2 int y = x;
3 y = 7;
4 int[] a = {1, 2, 3};
5 int[] b = a;
▶ b[0] = 9;
7 b = new int[2];
8 System.out.println(x + " " + a[0]);
```

VARIABLES

x = 5 | y = 7 | a = →#1 [9, 2, 3] | b = →#1 [9, 2, 3]

OUTPUT

Code Trace: Value or Reference?



b = new int[2]: b now points to a new, zero-filled array; a is unaffected.

Step 7 / 8

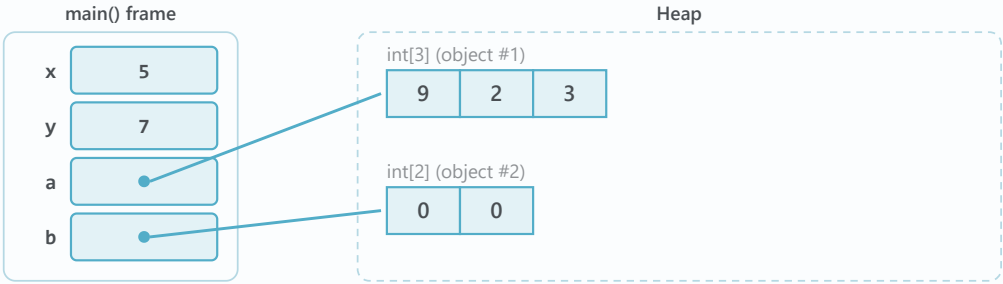
```
1 int x = 5;
2 int y = x;
3 y = 7;
4 int[] a = {1, 2, 3};
5 int[] b = a;
6 b[0] = 9;
▶ b = new int[2];
8 System.out.println(x + " " + a[0]);
```

VARIABLES

x = 5 | y = 7 | a = →#1 [9, 2, 3] | b = →#2 [0, 0]

OUTPUT

Code Trace: Value or Reference?



Output: **5 9**. Rule: **=** copies the value of a primitive and the address of a reference.

Step 8 / 8

```
1 int x = 5;
2 int y = x;
3 y = 7;
4 int[] a = {1, 2, 3};
5 int[] b = a;
6 b[0] = 9;
7 b = new int[2];
▶ System.out.println(x + " " + a[0]);
```

VARIABLES

`x = 5` `y = 7` `a = →#1 [9, 2, 3]` `b = →#2 [0, 0]`

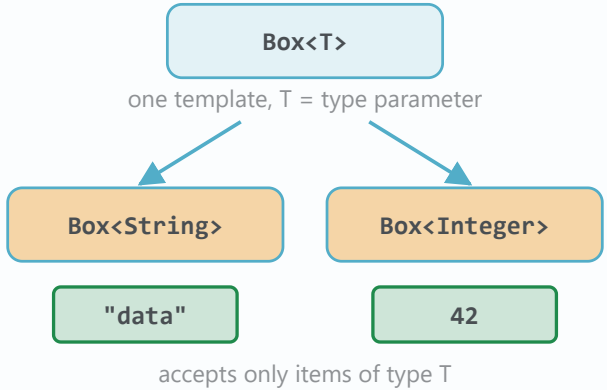
OUTPUT

```
5 9
```

Generics: One Structure for Every Type

```java
public class Box<T> {
  private T item;
  public void put(T x) { item = x; }
  public T get() { return item; }
}

Box<String> b = new Box<>();
b.put("data");
String s = b.get();      // no cast needed
Box<Integer> m = new Box<>();
m.put(42);               // int → Integer
// m.put("x");           // COMPILE ERROR
```



- Types are checked at **compile time**, so a wrong type never reaches run time.
- T must be a reference type: no `Box<int>`; use `Integer`.
- Java's collections are generic: `ArrayList<E>`, `Map<K,V>`.

Non-Primitive Data Structures

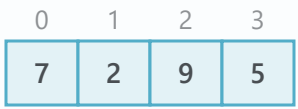
More complex structures, such as:

Linear Arrays, lists	**Linked** Linked lists	**Hierarchical** Trees
Network Graphs	**Indexed** Hash tables	**...** Sets, maps

- More complex; can combine several primitive types.
- Most can grow and shrink (arrays cannot).
- Built for specialized operations.

Data Structures at a Glance

The same four values (7, 2, 9, 5) arranged eight ways, each with its own strengths and weaknesses:



a[2] → 9 (one step)

Array

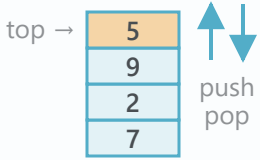
contiguous cells, O(1) access by index, fixed size



reach item i: O(n)

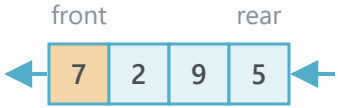
Linked list

nodes joined by references, flexible size



Stack

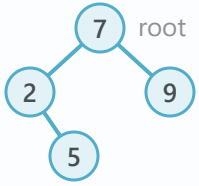
LIFO: push and pop at the top only



out at front · in at rear

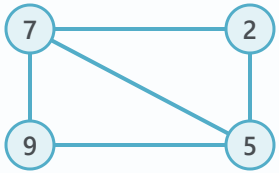
Queue

FIFO: first in, first out



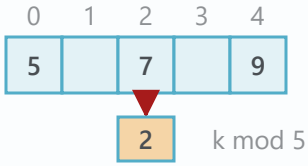
Tree

hierarchy: root, children, leaves



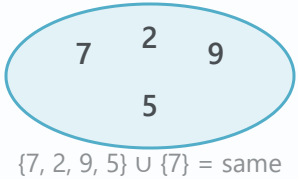
Graph

nodes + edges, many-to-many



Hash table

h(k) picks the bucket; O(1) on average



{7, 2, 9, 5} ∪ {7} = same

Set / Map

unique elements / key → value

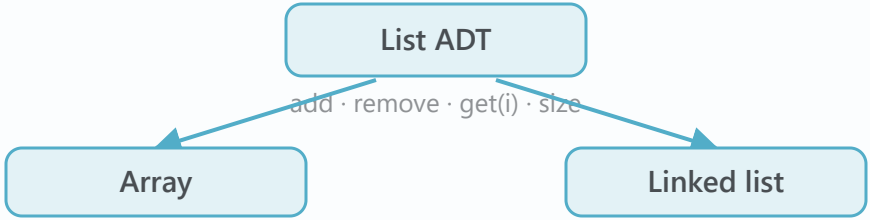
Abstract Data Types vs. Data Structures

Abstract Data Type (ADT)

- Lists, stacks, queues, sets, maps, graphs…
- Defines which operations exist (the behavior).
- Ignores implementation details.

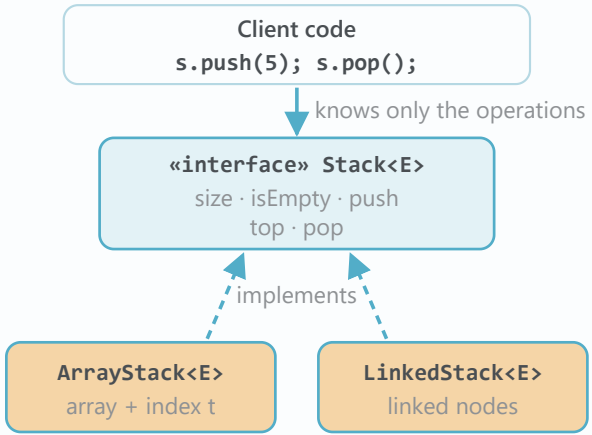
Data Structure

- Linked lists, arrays, adjacency matrices…
- Concrete structures that implement ADTs.
- Focus on memory and performance details.



From ADT to Java: The `interface`

```
public interface Stack<E> {
  int size();          // number of elements
  boolean isEmpty();   // is it empty?
  void push(E e);      // add on top
  E top();             // return the top element
  E pop();             // remove and return the top
}
```

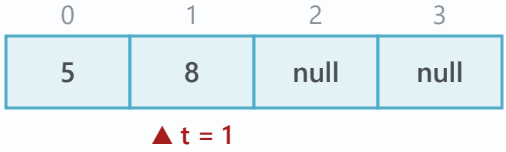


- **interface** = the ADT's contract: *what* it does.
- **class** = the data structure: *how* it does it.
- One ADT, many possible implementations.

`Stack<Integer> s = new ArrayStack<>(10);` — to swap implementations, change only this line; client code stays the same.

One Implementation: `ArrayStack`

```
public class ArrayStack<E> implements Stack<E> {
  private E[] data;          // the elements
  private int t = -1;        // index of the top element
  public ArrayStack(int capacity) {
    data = (E[]) new Object[capacity];
  }
  public int size() { return t + 1; }
  public boolean isEmpty() { return t == -1; }
  public E top() {
    if (isEmpty()) return null;
    return data[t];
  }
  // push and pop
}
```



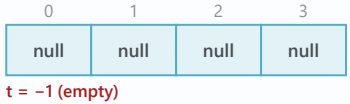
`size() = t + 1 = 2`

Operation	Time
size, isEmpty, top	O(1)
push, pop	O(1)
Memory	O(N) (capacity)

- Java can't create a generic array directly, hence `(E[]) new Object[..]`.
- Fixed capacity: `push` throws once the array is full.

Code Trace: `push` / `pop`

```
1  public void push(E e) {
2    if (size() == data.length)
3      throw new IllegalStateException("full");
4    data[++t] = e;
5  }
6  public E pop() {
7    if (isEmpty()) return null;
8    E e = data[t];
9    data[t--] = null;      // let GC reclaim it
10   return e;
11 }
▶ s.push(5); s.push(8);
13 System.out.println(s.pop());
14 s.push(3);
```



s is an **empty** ArrayStack with capacity 4 (t = −1). Call `push(5)`.

Step 1 / 16

VARIABLES

`t = -1` `data = [null, null, null, null]`

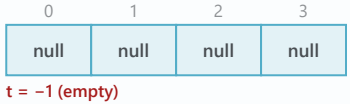
CALL STACK

`main()`

OUTPUT

Code Trace: `push` / `pop`

```
1  public void push(E e) {
▶    if (size() == data.length)
3      throw new IllegalStateException("full");
4    data[++t] = e;
5  }
6  public E pop() {
7    if (isEmpty()) return null;
8    E e = data[t];
9    data[t--] = null;      // let GC reclaim it
10   return e;
11 }
12 s.push(5); s.push(8);
13 System.out.println(s.pop());
14 s.push(3);
```



size() = 0 ≠ 4 → not full.

Step 2 / 16

VARIABLES

`t = -1` `data = [null, null, null, null]` `e = 5`

CALL STACK

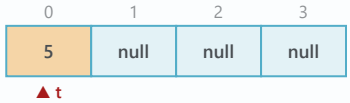
`push(5)`

`main()`

OUTPUT

Code Trace: `push` / `pop`

```
1  public void push(E e) {
2    if (size() == data.length)
3      throw new IllegalStateException("full");
▶    data[++t] = e;
5  }
6  public E pop() {
7    if (isEmpty()) return null;
8    E e = data[t];
9    data[t--] = null;      // let GC reclaim it
10   return e;
11 }
12 s.push(5); s.push(8);
13 System.out.println(s.pop());
14 s.push(3);
```



++t → t = 0, then **data[0] = 5**. Increment first, then write.

Step 3 / 16

VARIABLES

`t = 0` `data = [5, null, null, null]` `e = 5`

CALL STACK

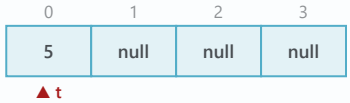
`push(5)`

`main()`

OUTPUT

Code Trace: `push` / `pop`

```
1  public void push(E e) {
2    if (size() == data.length)
3      throw new IllegalStateException("full");
4    data[++t] = e;
5  }
6  public E pop() {
7    if (isEmpty()) return null;
8    E e = data[t];
9    data[t--] = null;      // let GC reclaim it
10   return e;
11 }
▶ s.push(5); s.push(8);
13 System.out.println(s.pop());
14 s.push(3);
```



Call `push(8)`.

Step 4 / 16

VARIABLES

`t = 0` `data = [5, null, null, null]`

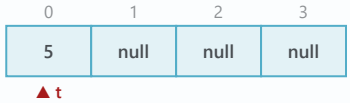
CALL STACK

`main()`

OUTPUT

Code Trace: `push` / `pop`

```
1  public void push(E e) {
▶    if (size() == data.length)
3      throw new IllegalStateException("full");
4    data[++t] = e;
5  }
6  public E pop() {
7    if (isEmpty()) return null;
8    E e = data[t];
9    data[t--] = null;      // let GC reclaim it
10   return e;
11 }
12 s.push(5); s.push(8);
13 System.out.println(s.pop());
14 s.push(3);
```



size() = 1 ≠ 4 → not full.

Step 5 / 16

VARIABLES

`t = 0` `data = [5, null, null, null]` `e = 8`

CALL STACK

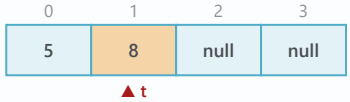
`push(8)`

`main()`

OUTPUT

Code Trace: `push` / `pop`

```
1  public void push(E e) {
2    if (size() == data.length)
3      throw new IllegalStateException("full");
▶    data[++t] = e;
5  }
6  public E pop() {
7    if (isEmpty()) return null;
8    E e = data[t];
9    data[t--] = null;      // let GC reclaim it
10   return e;
11 }
12 s.push(5); s.push(8);
13 System.out.println(s.pop());
14 s.push(3);
```



++t → t = 1, then **data[1] = 8**. Increment first, then write.

Step 6 / 16

VARIABLES

`t = 1` `data = [5, 8, null, null]` `e = 8`

CALL STACK

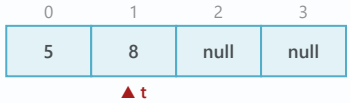
`push(8)`

`main()`

OUTPUT

Code Trace: `push` / `pop`

```
1  public void push(E e) {
2    if (size() == data.length)
3      throw new IllegalStateException("full");
4    data[++t] = e;
5  }
6  public E pop() {
7    if (isEmpty()) return null;
8    E e = data[t];
9    data[t--] = null;      // let GC reclaim it
10   return e;
11 }
12 s.push(5); s.push(8);
▶ System.out.println(s.pop());
14 s.push(3);
```



Call `pop()`; the result will be printed.

Step 7 / 16

VARIABLES

`t = 1` `data = [5, 8, null, null]`

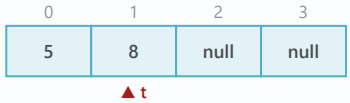
CALL STACK

`main()`

OUTPUT

Code Trace: `push` / `pop`

```
1  public void push(E e) {
2    if (size() == data.length)
3      throw new IllegalStateException("full");
4    data[++t] = e;
5  }
6  public E pop() {
▶    if (isEmpty()) return null;
8    E e = data[t];
9    data[t--] = null;      // let GC reclaim it
10   return e;
11 }
12 s.push(5); s.push(8);
13 System.out.println(s.pop());
14 s.push(3);
```



t = 1 → not empty; go on.

Step 8 / 16

VARIABLES

`t = 1` `data = [5, 8, null, null]`

CALL STACK

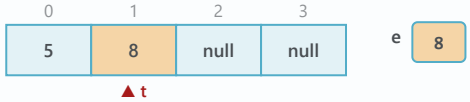
`pop()`

`main()`

OUTPUT

Code Trace: `push` / `pop`

```
1  public void push(E e) {
2    if (size() == data.length)
3      throw new IllegalStateException("full");
4    data[++t] = e;
5  }
6  public E pop() {
7    if (isEmpty()) return null;
▶   E e = data[t];
9    data[t--] = null;      // let GC reclaim it
10   return e;
11 }
12 s.push(5); s.push(8);
13 System.out.println(s.pop());
14 s.push(3);
```



e = data[1] = 8: the most recently added element (LIFO).

Step 9 / 16

VARIABLES

`t = 1` `data = [5, 8, null, null]` `e = 8`

CALL STACK

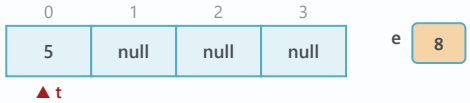
`pop()`

`main()`

OUTPUT

Code Trace: `push` / `pop`

```
1  public void push(E e) {
2    if (size() == data.length)
3      throw new IllegalStateException("full");
4    data[++t] = e;
5  }
6  public E pop() {
7    if (isEmpty()) return null;
8    E e = data[t];
▶   data[t--] = null;     // let GC reclaim it
10   return e;
11 }
12 s.push(5); s.push(8);
13 System.out.println(s.pop());
14 s.push(3);
```



Clear the cell to **null** so the object can be collected; **t--** → t = 0.

Step 10 / 16

VARIABLES

`t = 0` `data = [5, null, null, null]` `e = 8`

CALL STACK

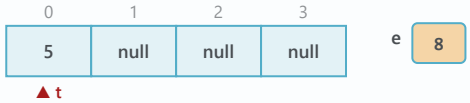
`pop()`

`main()`

OUTPUT

Code Trace: `push` / `pop`

```
1  public void push(E e) {
2    if (size() == data.length)
3      throw new IllegalStateException("full");
4    data[++t] = e;
5  }
6  public E pop() {
7    if (isEmpty()) return null;
8    E e = data[t];
9    data[t--] = null;     // let GC reclaim it
▶    return e;
11 }
12 s.push(5); s.push(8);
13 System.out.println(s.pop());
14 s.push(3);
```



return 8: the call's frame is removed.

Step 11 / 16

VARIABLES

`t = 0` `data = [5, null, null, null]` `e = 8`

CALL STACK

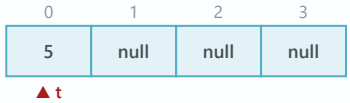
`pop()`

`main()`

OUTPUT

Code Trace: `push` / `pop`

```
1  public void push(E e) {
2    if (size() == data.length)
3      throw new IllegalStateException("full");
4    data[++t] = e;
5  }
6  public E pop() {
7    if (isEmpty()) return null;
8    E e = data[t];
9    data[t--] = null;      // let GC reclaim it
10   return e;
11 }
12 s.push(5); s.push(8);
▶ System.out.println(s.pop());
14 s.push(3);
```



8 is printed.

Step 12 / 16

VARIABLES

`t = 0` `data = [5, null, null, null]`

CALL STACK

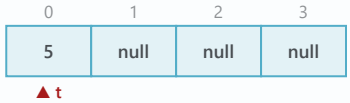
`main()`

OUTPUT

```
8
```

Code Trace: `push` / `pop`

```
1  public void push(E e) {
2    if (size() == data.length)
3      throw new IllegalStateException("full");
4    data[++t] = e;
5  }
6  public E pop() {
7    if (isEmpty()) return null;
8    E e = data[t];
9    data[t--] = null;      // let GC reclaim it
10   return e;
11 }
12 s.push(5); s.push(8);
13 System.out.println(s.pop());
▶  s.push(3);
```



`push(3)` : cell 1, now free, gets reused.

Step 13 / 16

VARIABLES

`t = 0` `data = [5, null, null, null]`

CALL STACK

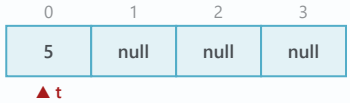
`main()`

OUTPUT

```
8
```

Code Trace: `push` / `pop`

```
1  public void push(E e) {
▶    if (size() == data.length)
3      throw new IllegalStateException("full");
4    data[++t] = e;
5  }
6  public E pop() {
7    if (isEmpty()) return null;
8    E e = data[t];
9    data[t--] = null;      // let GC reclaim it
10   return e;
11 }
12 s.push(5); s.push(8);
13 System.out.println(s.pop());
14 s.push(3);
```



size() = 1 ≠ 4 → not full.

Step 14 / 16

VARIABLES

`t = 0` `data = [5, null, null, null]` `e = 3`

CALL STACK

`push(3)`

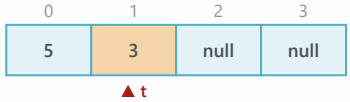
`main()`

OUTPUT

```
8
```

Code Trace: `push` / `pop`

```
1  public void push(E e) {
2    if (size() == data.length)
3      throw new IllegalStateException("full");
▶    data[++t] = e;
5  }
6  public E pop() {
7    if (isEmpty()) return null;
8    E e = data[t];
9    data[t--] = null;      // let GC reclaim it
10   return e;
11 }
12 s.push(5); s.push(8);
13 System.out.println(s.pop());
14 s.push(3);
```



++t → t = 1, then **data[1] = 3**. Increment first, then write.

Step 15 / 16

VARIABLES

`t = 1` `data = [5, 3, null, null]` `e = 3`

CALL STACK

`push(3)`

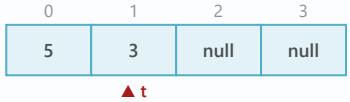
`main()`

OUTPUT

```
8
```

Code Trace: `push` / `pop`

```
1  public void push(E e) {
2    if (size() == data.length)
3      throw new IllegalStateException("full");
4    data[++t] = e;
5  }
6  public E pop() {
7    if (isEmpty()) return null;
8    E e = data[t];
9    data[t--] = null;      // let GC reclaim it
10   return e;
11 }
12 s.push(5); s.push(8);
13 System.out.println(s.pop());
▶ s.push(3);
```



Final state: [5, 3], t = 1. Each operation takes a fixed number of steps, whatever n is → **O(1)**.

Step 16 / 16

VARIABLES

`t = 1` `data = [5, 3, null, null]`

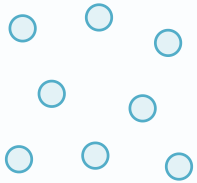
CALL STACK

`main()`

OUTPUT

```
8
```

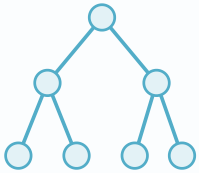
Organizing Data



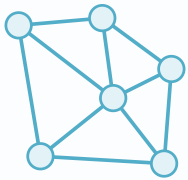
Unordered



Linear



Tree



Graph

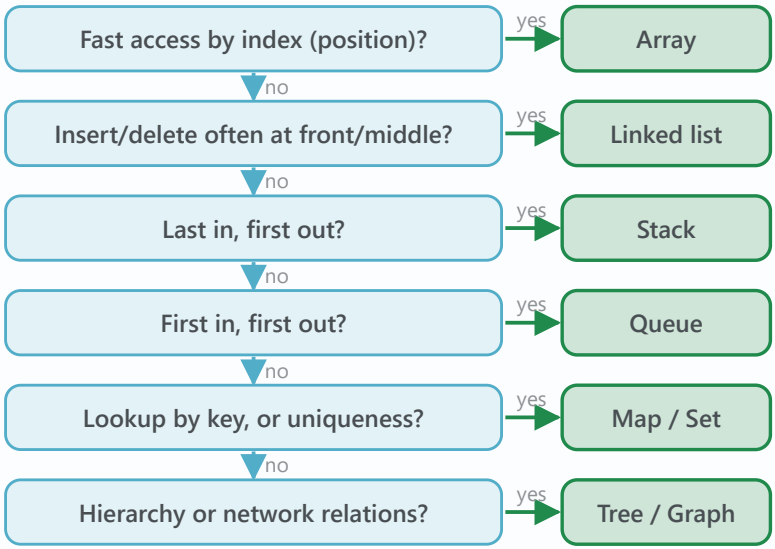
Arrangement	Relationship	Everyday example
Unordered	none	students in a classroom
Linear	one-to-one	students waiting in line
Tree	one-to-many	a family tree
Graph	many-to-many	a road map

Choosing a Data Structure

A few questions drive the choice:

- Can the structure be filled all at once, up front?
- Will elements be deleted?
- Does insertion order matter?
- Can the elements be sorted?
- Are elements processed in a fixed order, or is random access needed?

Contents won't change? Use a **static** structure. Frequent insertions and deletions? Use a **dynamic** one.



Fundamental Data Structures: An Overview

Structure	Key idea	Typical uses
Array	contiguous cells; O(1) access by index, fixed size	tables, matrices; the base of other structures
Linked list	nodes joined by references; flexible size	frequent insertion and deletion
Stack	LIFO: add / remove only at the top	undo, the call stack, bracket matching
Queue	FIFO: add at the rear, remove at the front	print queue, task queue, BFS
Tree	hierarchy: root, children, leaves	file systems, HTML (DOM), indexes
Priority queue	highest priority always first (heap)	task scheduling, emergency-room triage
Map / hash table	key → value; h(k) picks the bucket, O(1) on average	student ID → record, caches, word counts
Set	distinct elements; fast membership tests	removing duplicates, union and intersection
Graph	nodes + edges; many-to-many relations	road maps, social networks, shortest paths

SUMMARY

Summary

Topic	Key points
Digital data	text, numbers, images, and sound all become 0s and 1s, each under its own encoding
Data type	a value set plus operations; type safety catches errors at compile time
Primitive / reference	a primitive holds its value; a reference points to a heap object, and assignment copies only the reference
Generics	`Box<T>` : one class for every reference type; `Integer`, not `int`
ADT / data structure	the ADT says what (interface); the data structure says how (class)
Choosing a structure	depends on the access pattern, how often we insert / delete, ordering needs, and the kind of relationship

SUMMARY

Sample Problems

1. Which data structure fits "undo" in a text editor? Why?
2. Which structure lets a printer print documents in the order they arrive?
3. Which structure gives fast lookup of a student's record by student ID?
4. How would you model a city map to find the shortest route between two points?
5. Which structure would you use to remove duplicates from a list?
6. In Java, is the stack ADT written as an interface or a class? Which is `ArrayStack`?

▶ **Show answers**

SUMMARY

Exercises

1. After `int[] a = {1, 2}; int[] b = a; b[1] = 7;`, what is `a[1]`? Why?
2. Why doesn't `Box<int> b;` compile? What should you write instead?
3. Write the character `'B'` and the number 13 as 8-bit patterns.
4. After `String s = "data"; String t = s; t = t + "!";`, what is `s`? Why?
5. Starting from an empty `ArrayStack` with capacity 3, run push(1), push(2), pop(), push(3), push(4). What are t and the array at the end?
6. A music player needs a playlist in order, a "recently played" list, and fast search by song title. Which structures fit?

▶ **Answers / hints**

SUMMARY

Further Reading

- Goodrich, Tamassia, and Goldwasser, *Data Structures and Algorithms in Java*, 6th ed., 2014.
- Wirth, *Algorithms + Data Structures = Programs*, 1976.
- Oracle, Java SE 17 API documentation.

The End

Chapter 1: Introduction

Next deck: Chapter 2: Algorithm Analysis · Study questions · Glossary