UNIVERSITÉ Concordia UNIVERSITY

CENTRE FOR CONTINUING EDUCATION

DESKTOP APPLICATION DEVELOPMENT WITH JAVA – CEJV569

Lecture #8

Data Structures

Threads

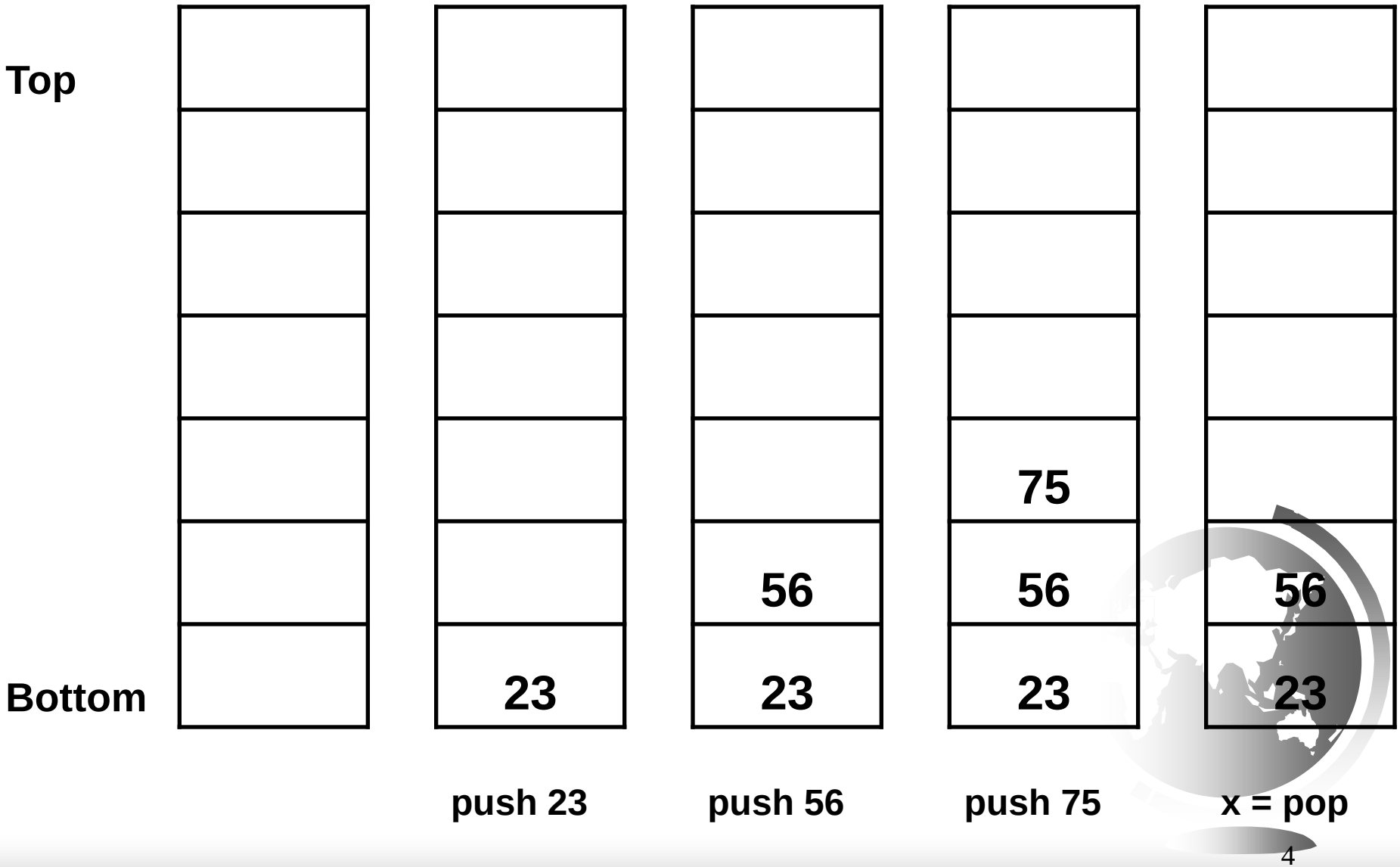
The Array

- Basic data structure common to most languages
- Allocated as a fixed size
- Supports random access to any member of the array through the subscript
- Does not support insertions or deletions
- Ideal structure when the size is constant and you only need to replace (overwrite) members

The Stack

- Structure that supports only two operations
- Add member to the stack
- Adds the member to the current end of the stack
- Called “push”
- Remove a member from the stack
- Removes the last member added
- Called “pop”
- “Last In First Out” or LIFO structure

A Stack



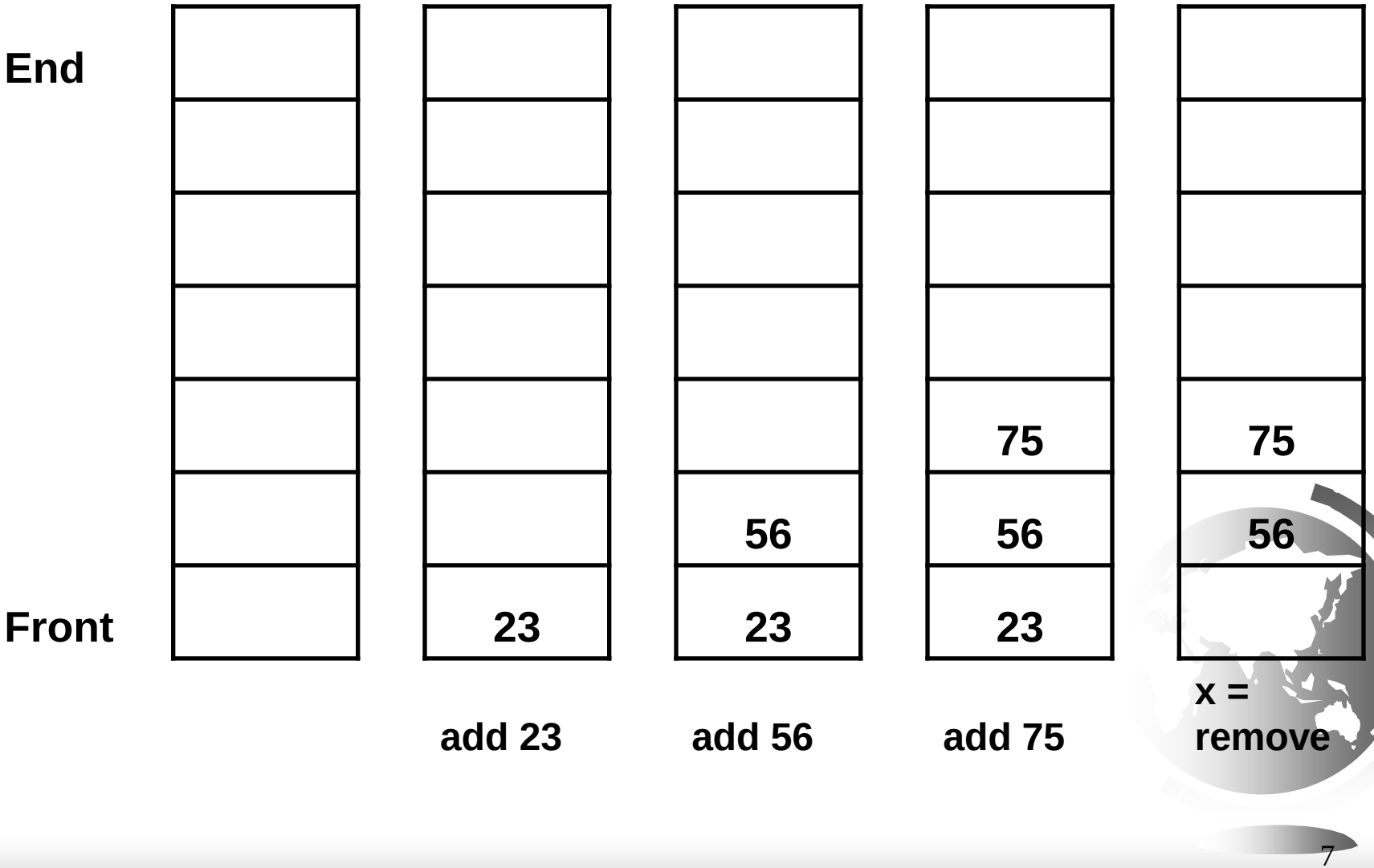
The Stack Interface

- void push(T t);
- T pop();
- T peek();
- int size();

The Queue

- Related to a stack
- New members are added to the end
- Called “enqueue” akin to push
- Members can only be removed from the front of the structure
- Called “dequeue” which is akin to pop but at the front
- “First In First Out” or FIFO structure

A Queue



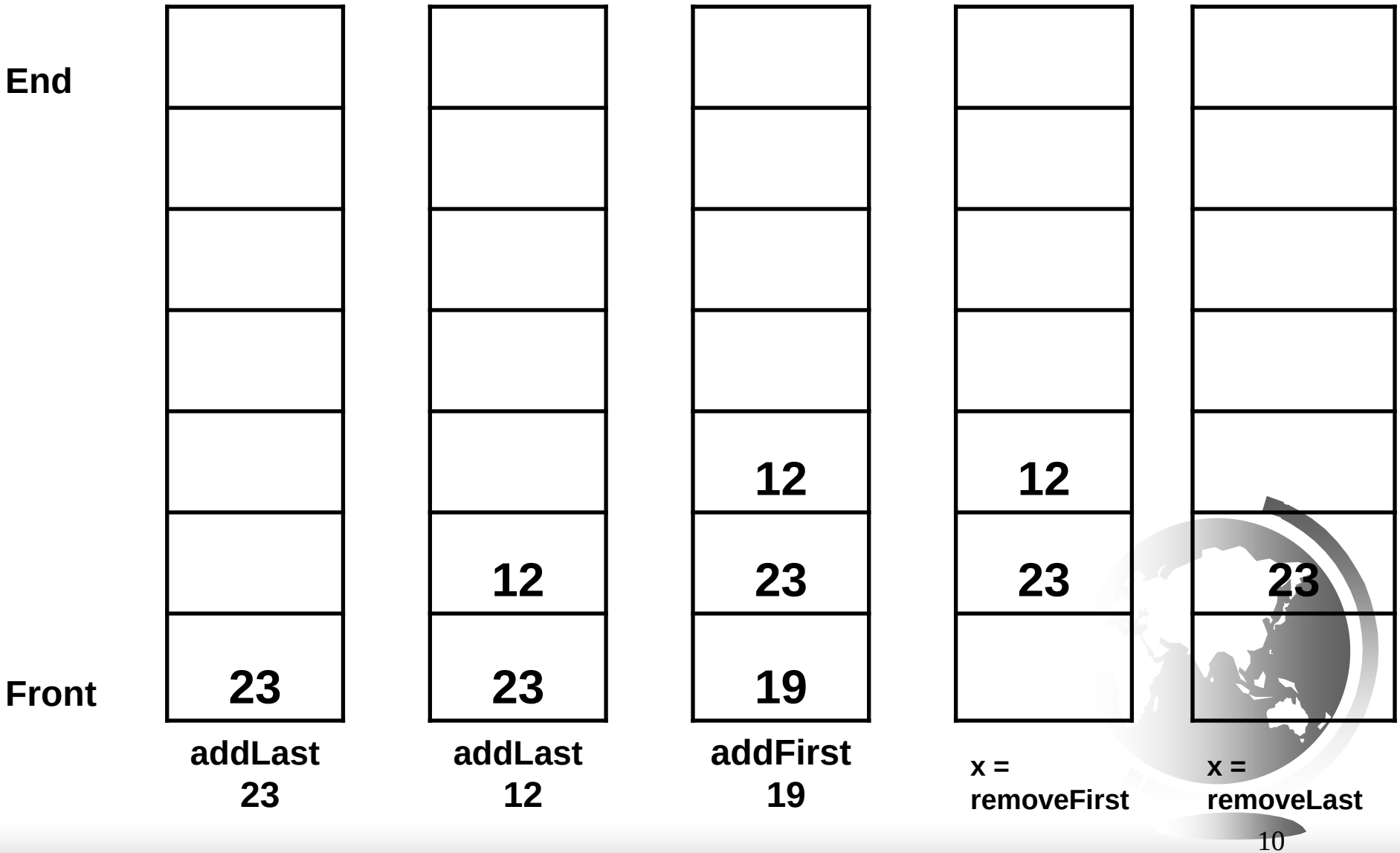
The Queue Interface

- void add(T t); // add to end
- T remove(); // remove from front
- T element(); // peek at front
- int size();

The Deque

- Accepts additions or removals from either end
- ‘push’ and ‘pop’ from either end

The Deque


End
Front
23
12
23
12
23
19
12
23
23
addLast
23
addLast
12
addFirst
19
x =
removeFirst
x =
removeLast

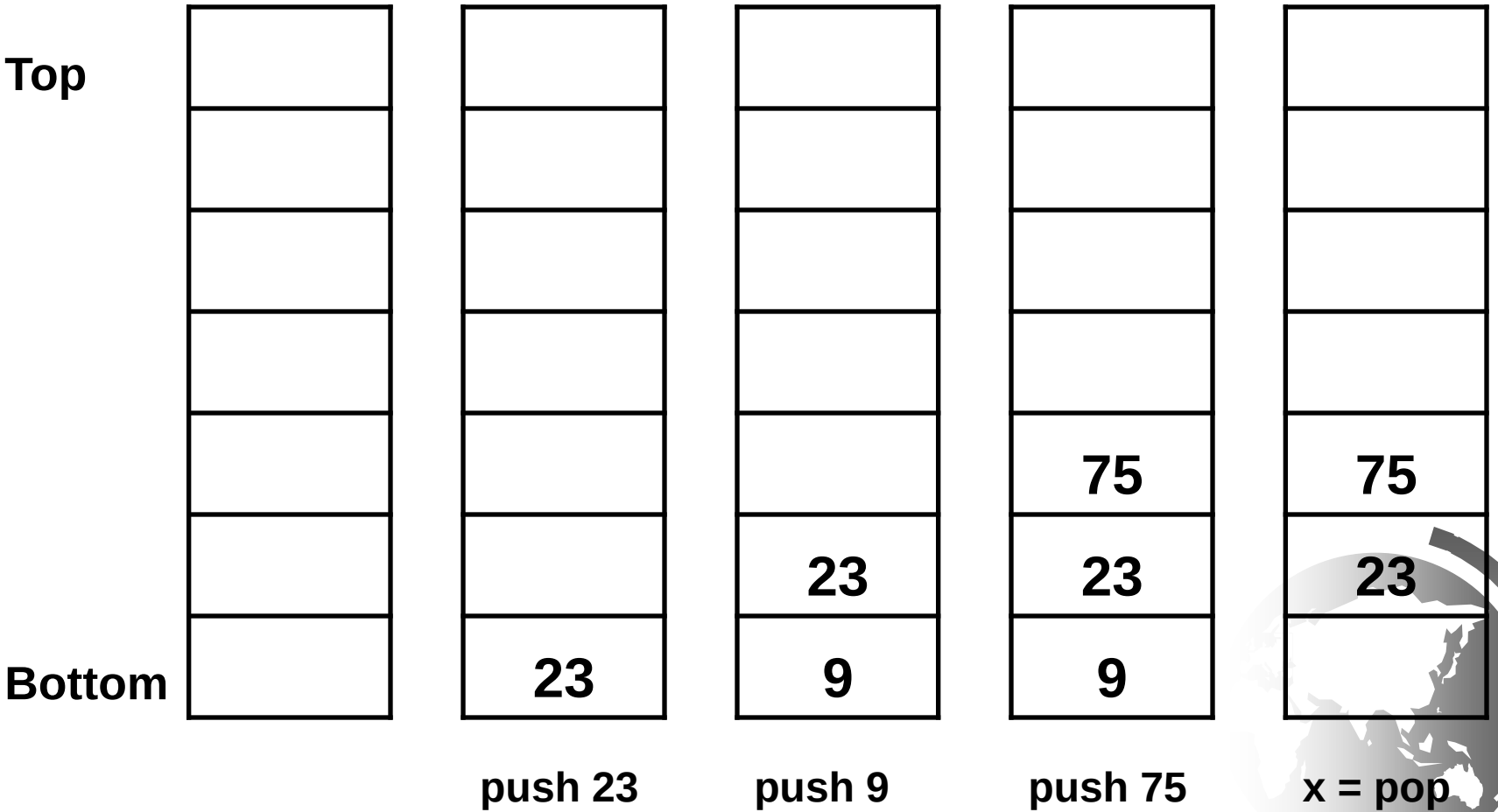
The Deque Interface

- void addFirst(T t); // add to front
- void addLast(T t); // add to end
- T removeFirst(); // remove from front
- T removeLast(); // remove from end
- T getFirst(); // peek at front
- T getLast(); // peek at end
- int size();

The Priority Queue

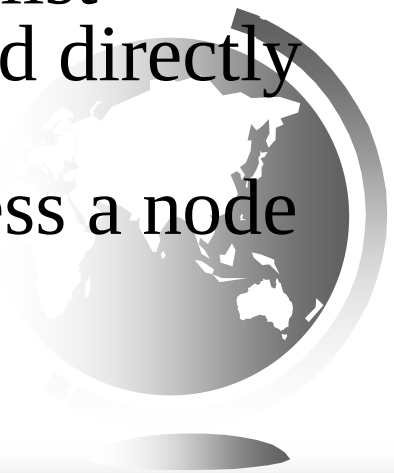
- Similar to normal queue except that it is maintained in a specific order
- Data is stored in a priority queue in pairs
- One item is the data and the second item is the priority
- Ensures that items of the highest priority are removed from the front of the queue

The Priority Queue

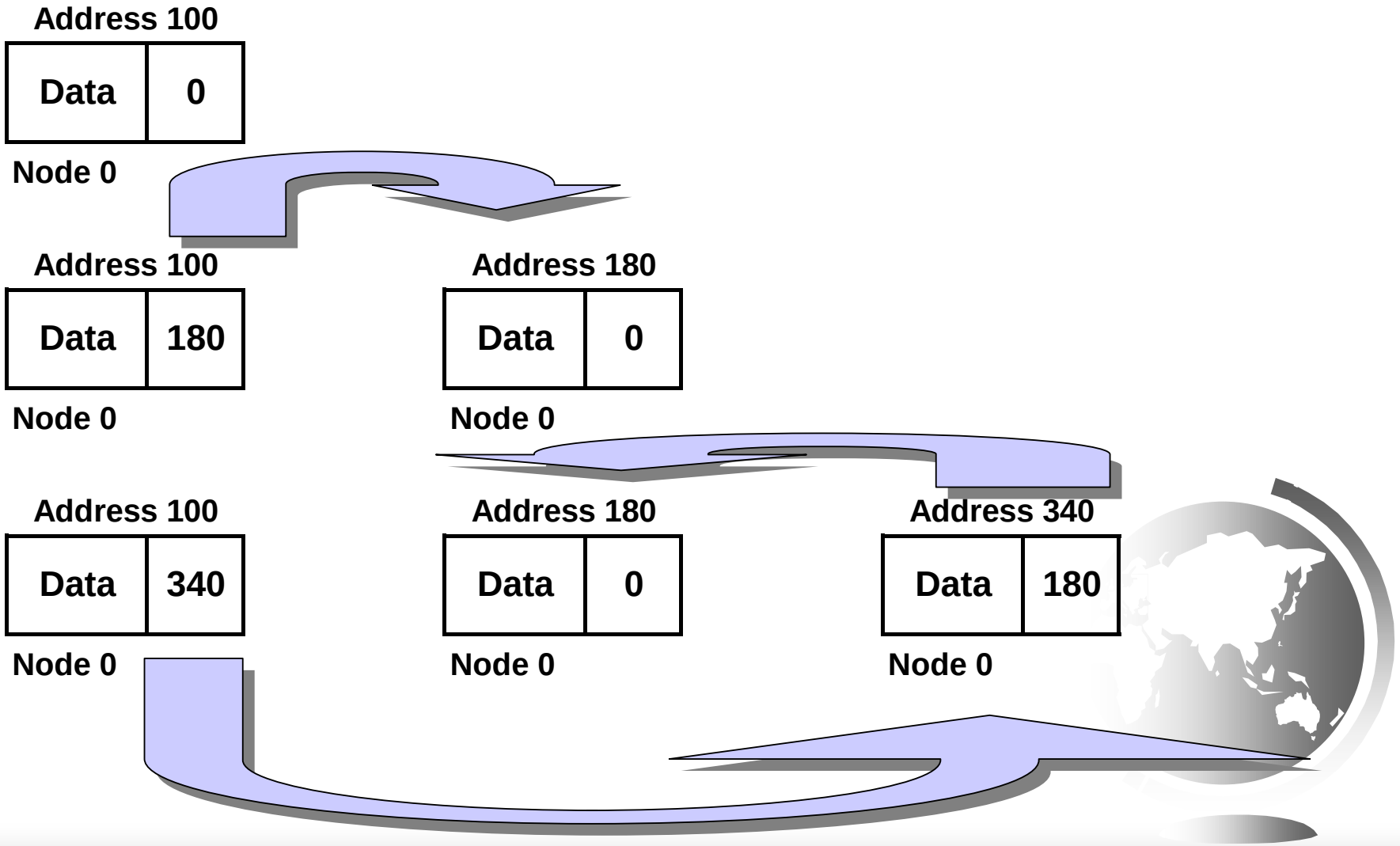


The Single Linked List

- Stores data in units commonly called “nodes”
- Nodes are created dynamically when ever an item is added to the list
- To connect the nodes there is a variable in every node that contains the address of the next node in the sequence
- Nodes are not adjacent therefore a list supports insertions and deletions anywhere in the list
- Drawback is that a list cannot be accessed directly with a subscript
- Must always be searched linearly to access a node



The Single Linked List

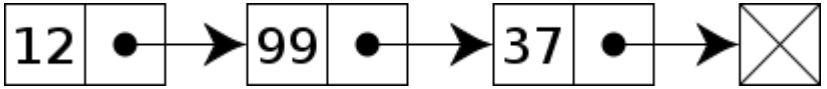


The Double Linked List

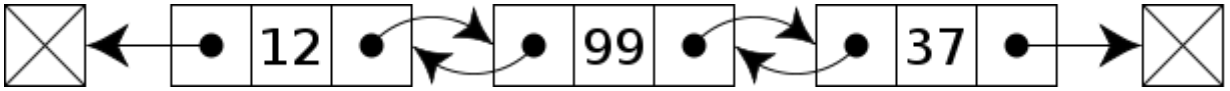
- Every node has a pointer to both the next node and the previous node
- Permits the list to be searched in both directions

The Double Linked List

Linked List



Double Link List

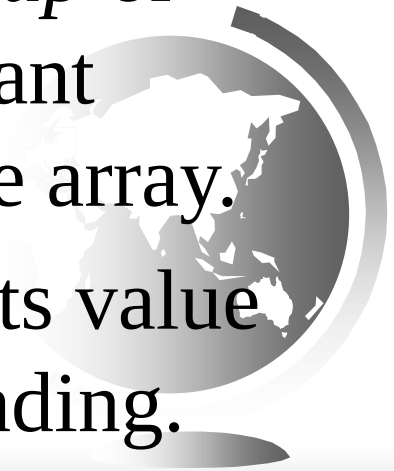


Set

- A **set** is a collection of certain values without any particular order, and no repeated values.
- It corresponds with the mathematical concept of set, but with the restriction that it has to be finite.
- Disregarding sequence, and the fact that there are no repeated values, it is the same as a list

Associative Array

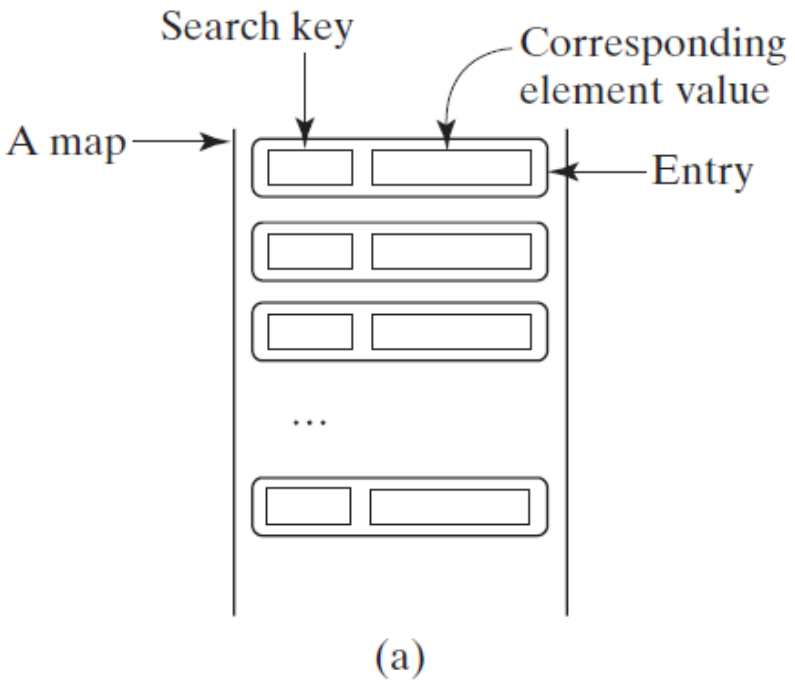
- An **associative array** is composed of a collection of keys and a collection of values, where each key is associated with one value.
- The operation of finding the value associated with a key is called a *lookup* or indexing, and this is the most important operation supported by an associative array.
- The relationship between a key and its value is sometimes called a mapping or binding.



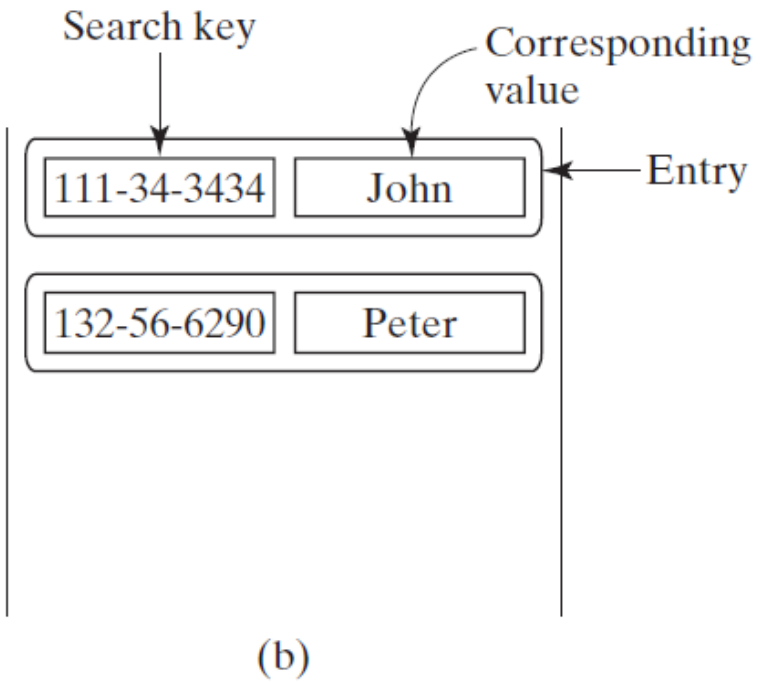
Hash

- A **hash table**, or a **hash map**, is a data structure that associates *keys* with *values*.
- The primary operation it supports efficiently is a *lookup*
- Given a key (e.g. a person's name), find the corresponding value (e.g. that person's telephone number).
- It works by transforming the key using a hash function into a *hash*, a number that the hash table uses to locate the desired value.

Hash Map


Search key
Corresponding element value
A map
Entry
...

(a)


Search key
Corresponding value
Entry
111-34-3434
John
132-56-6290
Peter

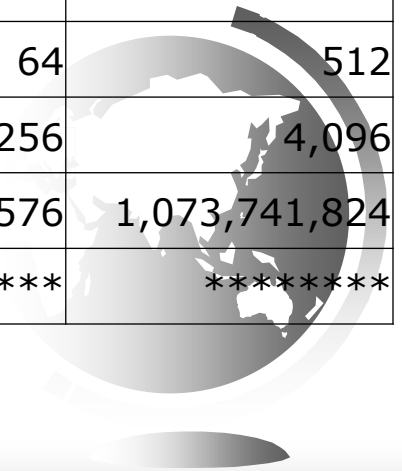
(b)

Measuring Performance

- In Computer Science it is necessary to measure the performance of a data structure or compiled code
- An indication of the time it takes to perform an operation relative to some measurable value
- It is indicated using **Big O Notation**
- To access an element of an array we say it performs at **O(1)**
- Regardless of the size of an array it will always take the same amount of time to access an element
- **O(1)** means in a constant length of time

Size Matters

	constant	*logarithmic*	*linear*		*quadratic*	*cubic*
n	**O(1)**	**O(log N)**	**O(N)**	**O(N log N)**	**O(N^2)**	**O(N^3)**
1	1	1	1	1	1	1
2	1	1	2	2	4	8
4	1	2	4	8	16	64
8	1	3	8	24	64	512
16	1	4	16	64	256	4,096
1,024	1	10	1,024	10,240	1,048,576	1,073,741,824
1,048,576	1	20	1,048,576	20,971,520	****	********



Does anyone really have that much data?

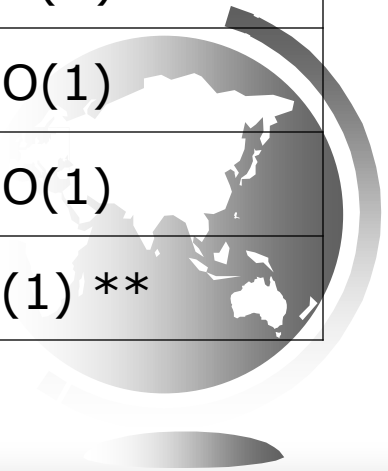
- It's hard to find a digital camera that that has fewer than a million pixels (1 mega-pixel).
- These images are processed and displayed on the screen.
- The algorithms that do this had better not be $O(N^2)$!
- If it took one microsecond (1 millionth of a second) to process each pixel, an $O(N^2)$ algorithm would take more than a week to finish processing a 1 megapixel image, and more than three months to process a 3 megapixel image (note the rate of increase is definitely not linear).

Does anyone really have that much data?

- Another example is sound.
- CD audio samples are 16 bits, sampled 44,100times per second for each of two channels.
- A typical 3 minute song consists of about 8 million data points.
- You had better choose the right algorithm to process this data.

Algorithms

Algorithm	array ArrayList	LinkedList
access front	O(1)	O(1)
access back	O(1)	O(1)
access middle	O(1)	O(N)
insert at front	O(N)	O(1)
insert at back	O(1)	O(1)
insert in middle	O(N)	O(1) **



Map Performance

	HashMap
get	O(1)
put	O(1)
contains	O(1)
remove	O(1)

Exercise 27

1) Create two priority queues, {"**George**", "**Jim**", "**John**", "**Blake**", "**Kevin**", "**Michael**"} and {"**George**", "**Katie**", "**Kevin**", "**Michelle**", "**Ryan**"}, and find their union, difference, and intersection.
2) A Java program contains various pairs of grouping symbols, such as:
   - Parentheses: **(** and **)**
   - Braces: **{** and **}**
   - Brackets: **[** and **]**

   – Note that the grouping symbols cannot overlap. For example, **(a{b)}** is illegal. Write a program to check whether a Java source-code file has correct pairs of grouping symbols.

Threads

Concept

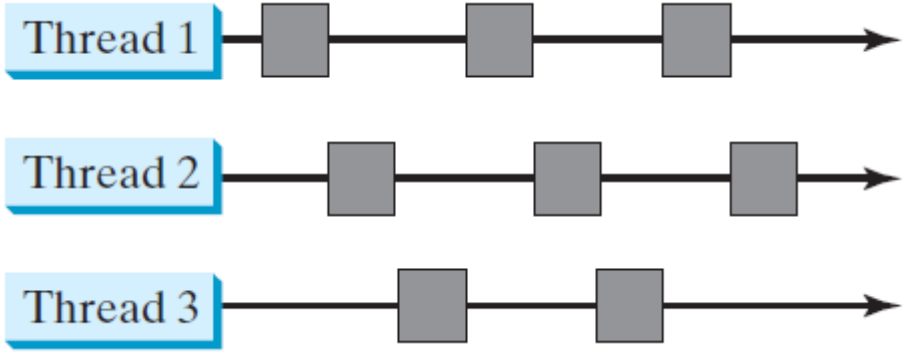
Parallel Programming

Usage

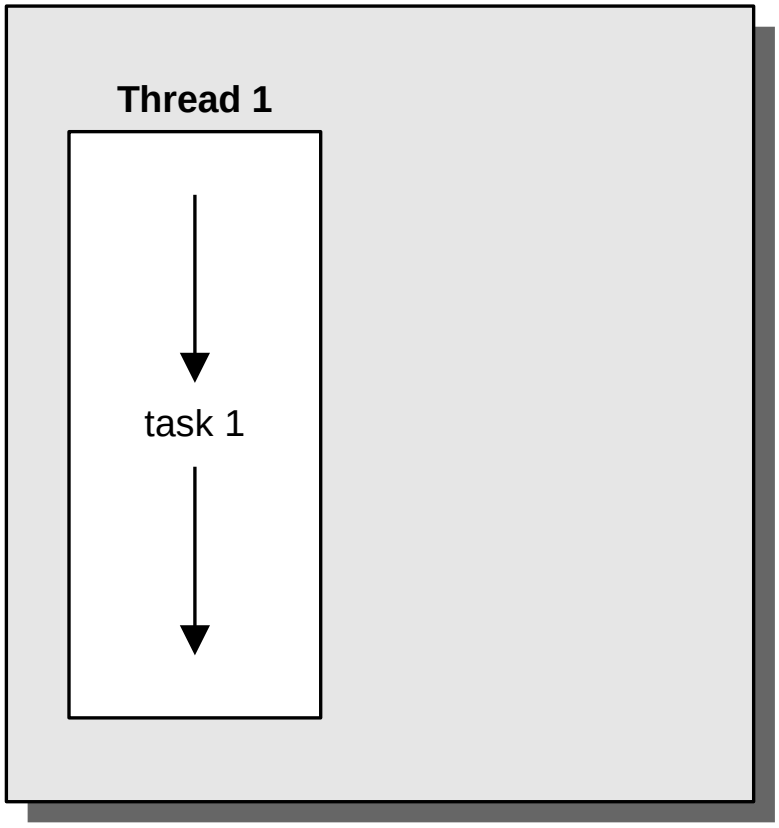
Threads Concept

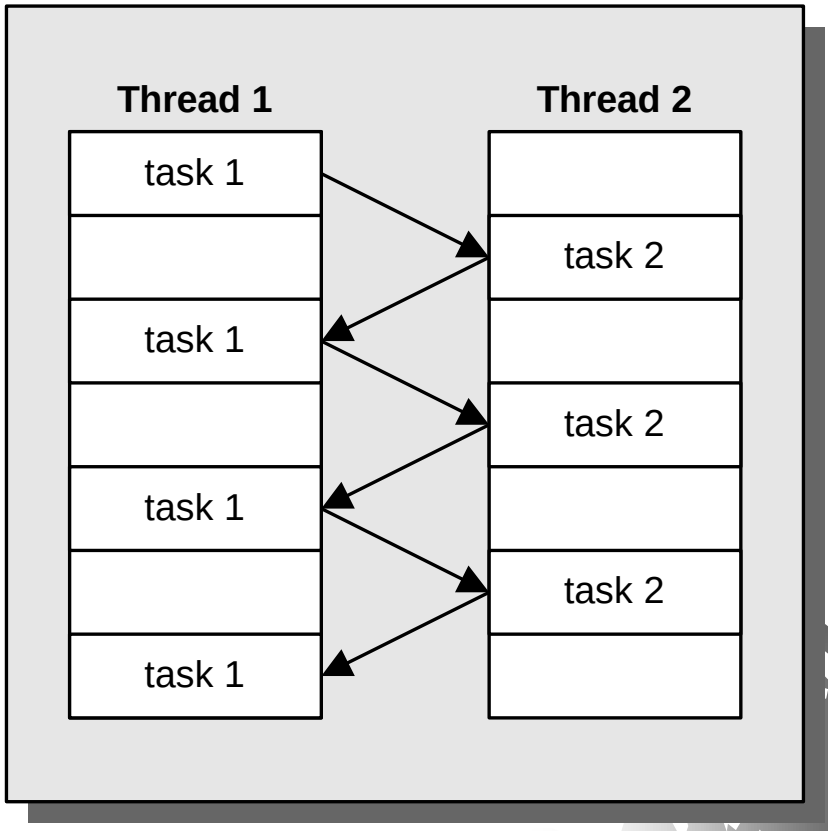
Multiple threads on multiple CPUs

Multiple threads sharing a single CPU



How using threads can improve performance





Concurrency

- Turn a program into separate, independently running subtasks
- Each of these is called a *thread*
- Code as if each thread runs by itself and has the CPU to itself
- Underlying mechanism is dividing up the CPU time

Concurrency

- *Process* is a self-contained running program with its own address space
- Behaves as if it is the only process running
- Periodically the CPU switches from one task to another
- *Thread* is a single sequential flow of control within a process
- Single process can have multiple concurrently executing threads

Motivation

- Produce a responsive user interface.
- Consider a program that ignores user input and becomes unresponsive
- Conventional methods cannot continue performing their operations and at the same time return control to the rest of the program

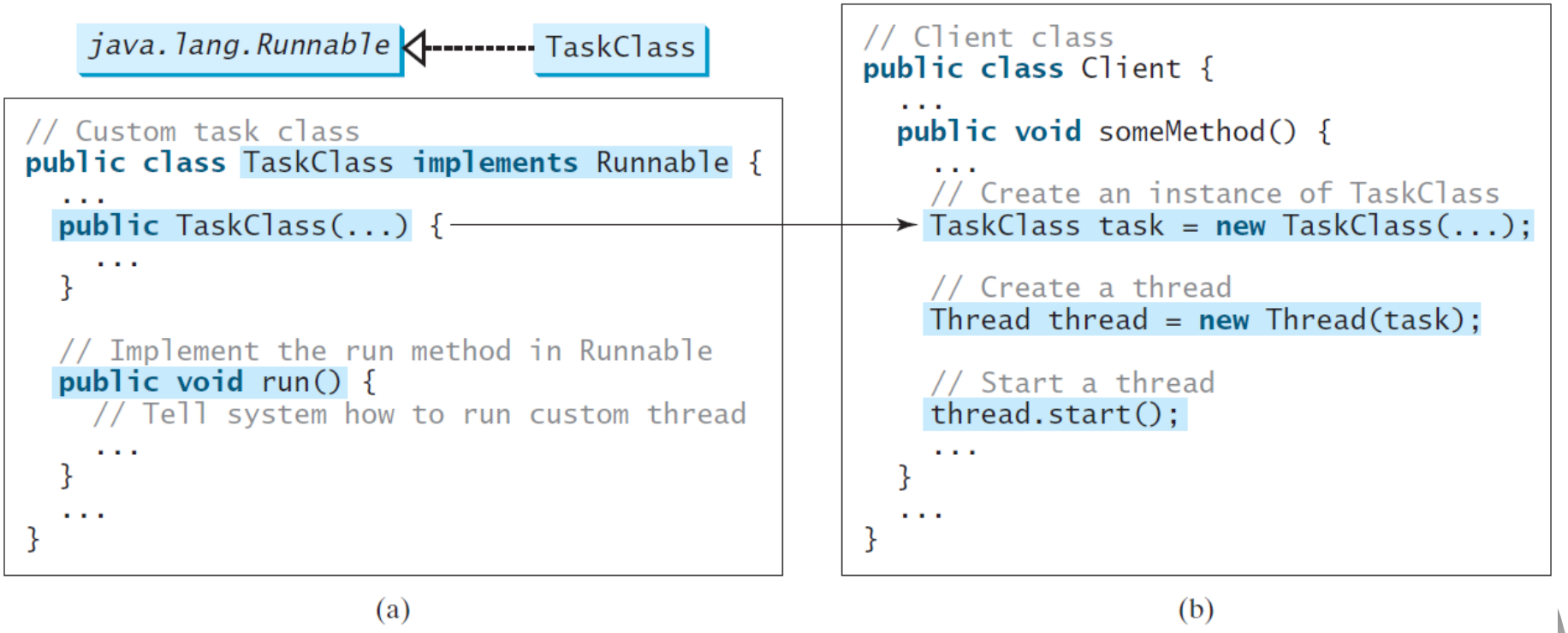
Motivation

- Programs with many threads must be able to run on a single-CPU machine
- Must also be possible to write the same program without using any threads
- Problems such as simulation are very difficult to solve without concurrency

Motivation

- Threading models allow for the juggling of several operations at the same
- CPU will cycle around and give each thread some of its time
- Each thread behaves as if it constantly has the CPU to itself
- CPU's time is actually sliced between all the threads

Creating Tasks and Threads



Basic threads

- Simplest way to create a thread is to inherit from **java.lang.Thread**
- Most important method for **Thread** is **run( )**
- Must override this method to make the thread do your bidding
- **run( )** is the code that will be executed "simultaneously" with the other threads in a program.

Example: Using the `Runnable` Interface to Create and Launch Threads

- Objective: Create and run three threads:
  - The first thread prints the letter *a* 100 times.
  - The second thread prints the letter *b* 100 times.
  - The third thread prints the integers 1 through 100.

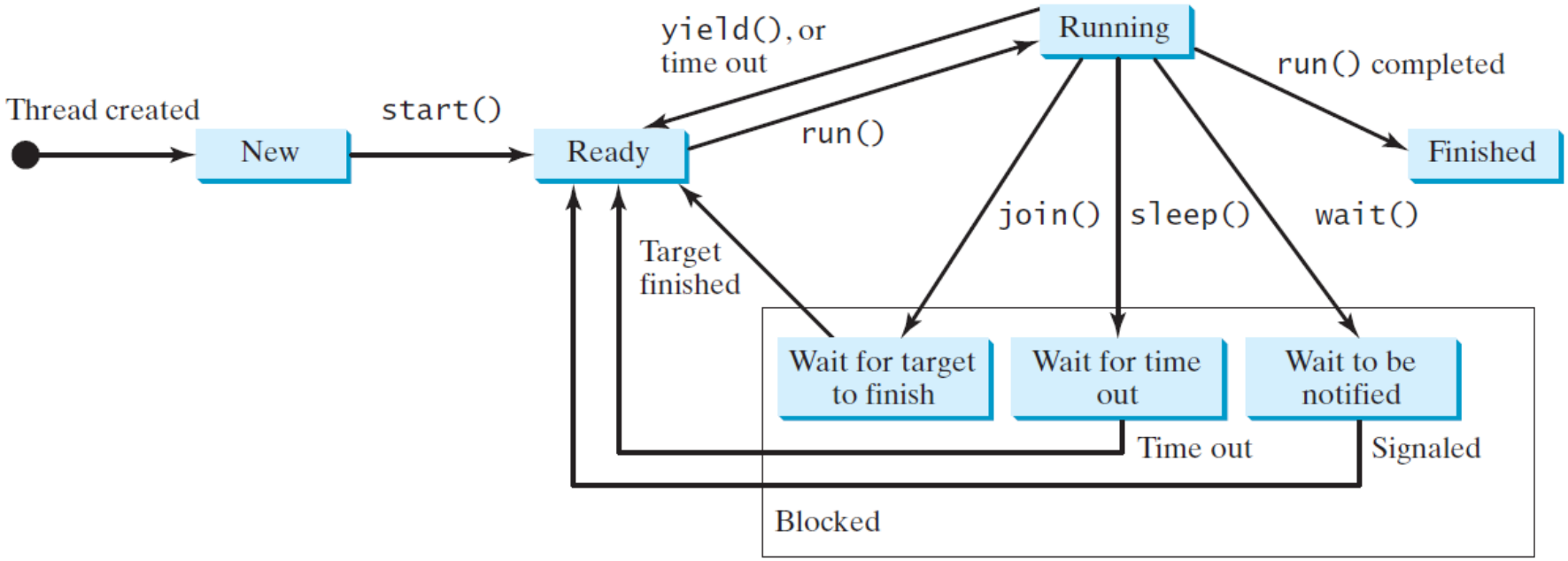
TaskThreadDemo

Basic threads

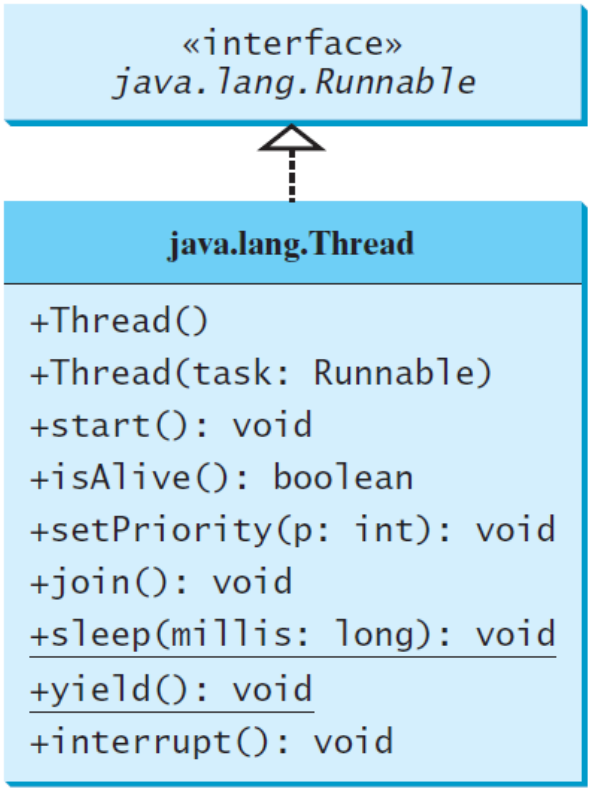
1. Constructor called to build the object
2. Constructor calls **start( )** to configure the thread
3. Thread execution mechanism calls **run( )**

- ☞ Without a call to **start( )** the thread will never be started

The life cycle of a thread



The Thread Class



`+Thread()`	Creates an empty thread.
`+Thread(task: Runnable)`	Creates a thread for a specified task.
`+start(): void`	Starts the thread that causes the `run()` method to be invoked by the JVM.
`+isAlive(): boolean`	Tests whether the thread is currently running.
`+setPriority(p: int): void`	Sets priority `p` (ranging from 1 to 10) for this thread.
`+join(): void`	Waits for this thread to finish.
`+sleep(millis: long): void`	Puts a thread to sleep for a specified time in milliseconds.
`+yield(): void`	Causes a thread to pause temporarily and allow other threads to execute.
`+interrupt(): void`	Interrupts this thread.

The Static sleep(milliseconds) Method

The sleep(long mills) method puts the thread to sleep for the specified time in milliseconds. For example, suppose you modify the code in TaskThreadDemo.java as follows:

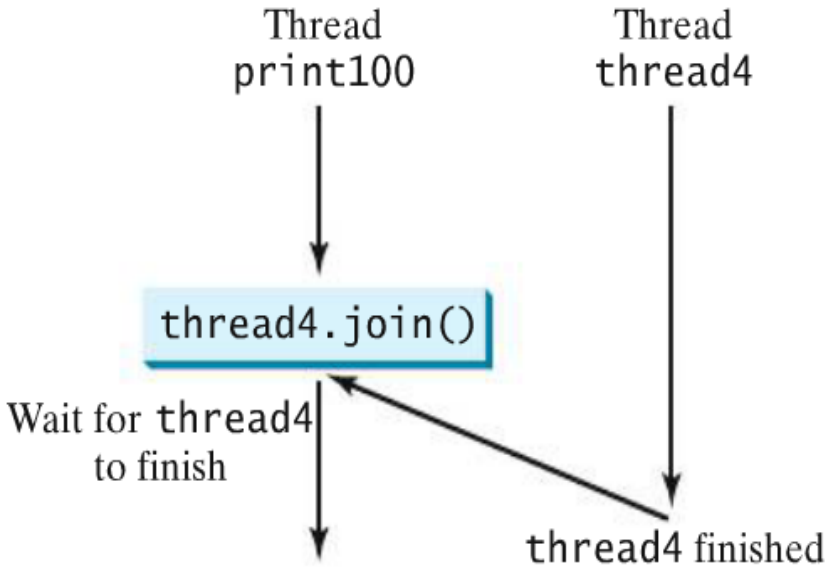
```
public void run() {
  for (int i = 1; i < times; i++) {
    System.out.print(" " + charToPrint);
    try {
      if (i >= 50) Thread.sleep(1);
    }
    catch (InterruptedException ex) {
    }
  }
}
```

Every time a number (>= 50) is printed, the thread is put to sleep for 1 millisecond.

The join() Method

You can use the join() method to force one thread to wait for another thread to finish. For example, suppose you modify the code in TaskThreadDemo.java as follows:

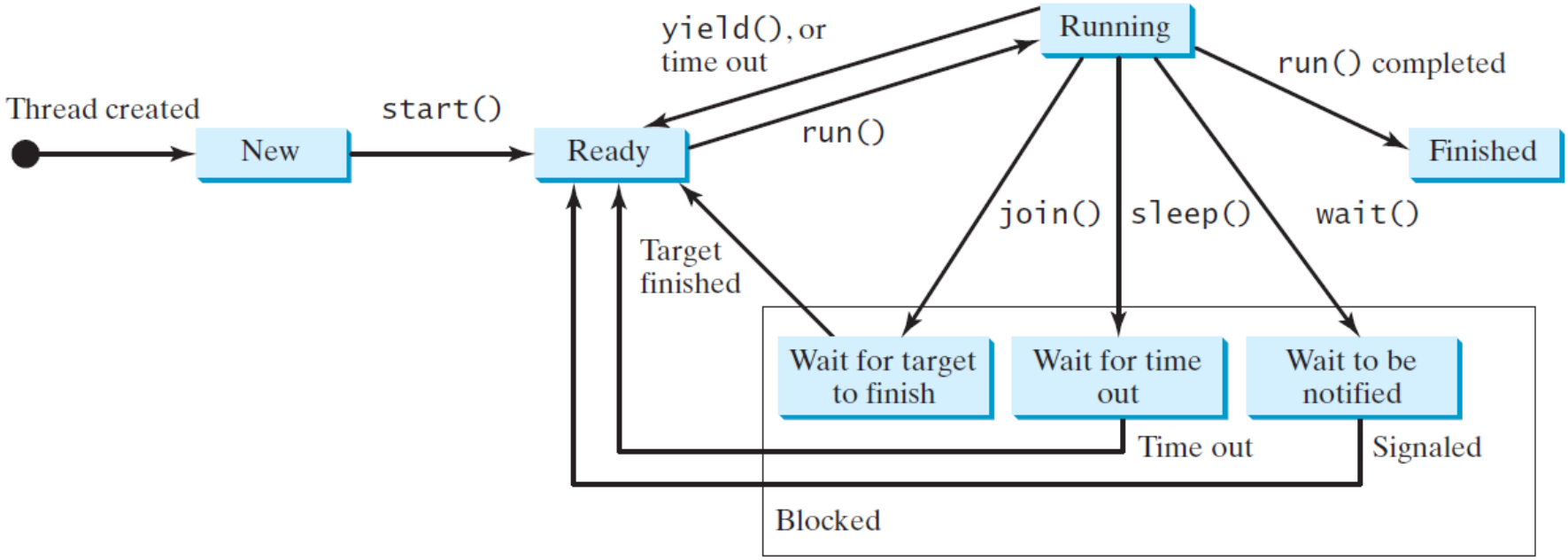
```
public void run() {
  Thread thread4 = new Thread(
    new PrintChar('c', 40));
  thread4.start();
  try {
    for (int i = 1; i <= lastNum; i++) {
      System.out.print(" " + i);
      if (i == 50) thread4.join();
    }
  }
  catch (InterruptedException ex) {
  }
```



The numbers after 50 are printed after thread printA is finished.

isAlive(), interrupt(), and isInterrupted()

- The isAlive() method is used to find out the state of a thread. It returns true if a thread is in the Ready, Blocked, or Running state; it returns false if a thread is new and has not started or if it is finished.



isAlive(), interrupt(), and isInterrupted()

- The interrupt() method interrupts a thread in the following way:
  - If a thread is currently in the Ready or Running state, its interrupted flag is set;
  - if a thread is currently blocked, it is awakened and enters the Ready state, and an java.io.InterruptedException is thrown.

- The isInterrupt() method tests whether the thread is interrupted.

The deprecated stop(), suspend(), and resume() Methods

NOTE:

- The Thread class also contains the stop(), suspend(), and resume() methods.
- As of Java 2, these methods are *deprecated* (or *outdated*) because they are known to be inherently unsafe.
- You should assign null to a Thread variable to indicate that it is stopped rather than use the stop() method.

Order of Execution

- Cannot control the order of execution of threads
- Do not use threads at all if order is required
- Write cooperative routines that hand control to each other in a specified order

Priority

- Tells the scheduler how important this thread is
- Order that the CPU attends to threads is indeterminate
- Scheduler leans toward the one with the highest priority first
- Doesn’t mean that threads with lower priority aren’t run
- Can’t get deadlocked because of priorities.
- Lower priority threads just tend to run less often

Thread Priority

- Each thread is assigned a default priority of `Thread.NORM_PRIORITY`. You can reset the priority using `setPriority(int priority)`.

- Some constants for priorities include
  `Thread.MIN_PRIORITY`
  `Thread.MAX_PRIORITY`
  `Thread.NORM_PRIORITY`

Exercise 28

- Create and run three threads:
  - The first thread prints the letter *a* 100 times.
  - The second thread prints the letter *b* 100 times.
  - The third thread prints the integers 1 through 100.
  - After writing your code, display the output in a text area, as shown in the following Figure: