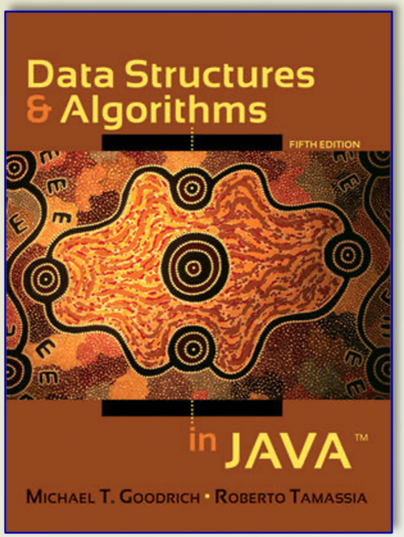
Data Structure & Algorithms in JAVA

5th edition

Michael T. Goodrich

Roberto Tamassia



Chapter 5: Stacks, Queues, and Deques

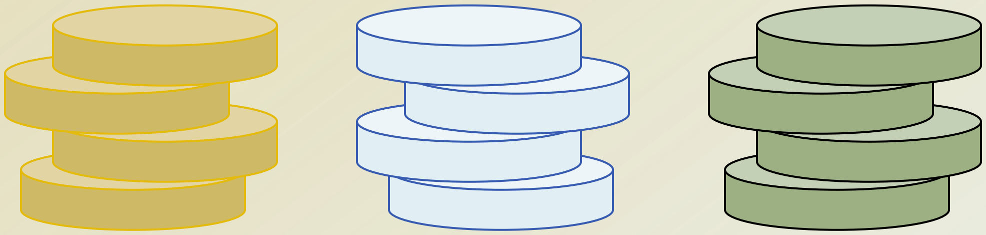
CPSC 3200

Algorithm Analysis and Advanced Data Structure

Chapter Topics

- Stacks.
- Queues.
- Double-Ended Queues.

Stacks

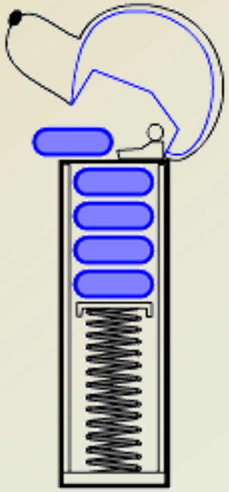


Abstract Data Types (ADTs)

- An abstract data type (ADT) is an abstraction of a data structure.
- An ADT specifies:
  - Data stored.
  - Operations on the data.
  - Error conditions associated with operations.

- Example: ADT modeling a simple stock trading system
  - The data stored are buy/sell orders.
  - The operations supported are
    - order buy(stock, shares, price).
    - order sell(stock, shares, price).
    - void cancel(order).
  - Error conditions:
    - Buy/sell a nonexistent stock.
    - Cancel a nonexistent order.

The Stack ADT



- The Stack ADT stores arbitrary objects.
- Insertions and deletions follow the **last-in first-out** scheme.
- Think of a spring-loaded plate dispenser
- Main stack operations:
  - **push(object):** inserts an element.
  - **object pop( ):** removes and returns the last inserted element.
- Auxiliary stack operations:
  - **object top():** returns the last inserted element without removing it.
  - **integer size():** returns the number of elements stored
  - **boolean isEmpty():** indicates whether no elements are stored

Example

Operation	*Output*	*Stack Content*
push(5)		
push(3)		
pop()		
push(7)		
pop()		
top()		
pop()		
pop()		
isEmpty()		
push(9)		
push(7)		
push(3)		
push(5)		
size()		
pop()		
push(8)		
pop()		
pop()		

Stack Interface in Java

- Java interface corresponding to our Stack ADT
- Requires the definition of class **EmptyStackException**
- Different from the built-in Java class java.util.Stack
- http://docs.oracle.com/javase/7/docs/api/java/util/Stack.html

Stack.java

Exceptions

- Attempting the execution of an operation of ADT may sometimes cause an error condition, called an exception.
- Exceptions are said to be "thrown" by an operation that cannot be executed.
- In the Stack ADT, operations **pop** and **top** cannot be performed if the stack is empty.
- Attempting the execution of **pop** or **top** on an empty stack throws an EmptyStackException

Applications of Stacks

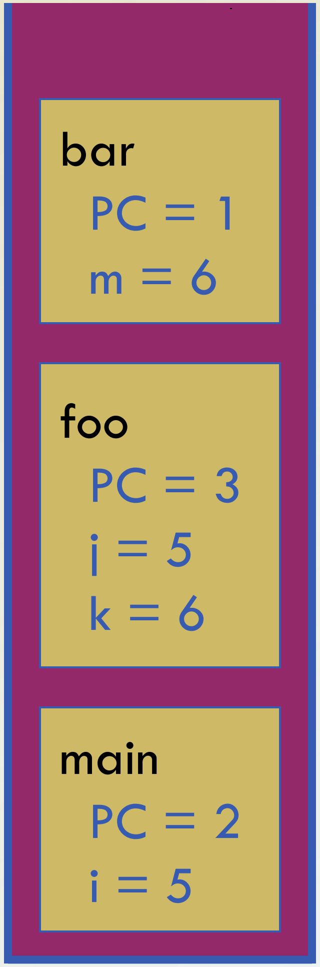
- **Direct applications**
  - Page-visited history in a Web browser.
  - Undo sequence in a text editor.
  - Chain of method calls in the Java Virtual Machine.

- **Indirect applications**
  - Auxiliary data structure for algorithms.
  - Component of other data structures.

Method Stack in the JVM

- The Java Virtual Machine (JVM) keeps track of the chain of active methods with a stack.
- When a method is called, the JVM **pushes** on the stack a frame containing
  - **Local variables** and **return value**
  - Program counter, keeping track of the statement being executed
- When a method ends, its frame is **popped** from the stack and control is passed to the method on top of the stack
- Allows for recursion

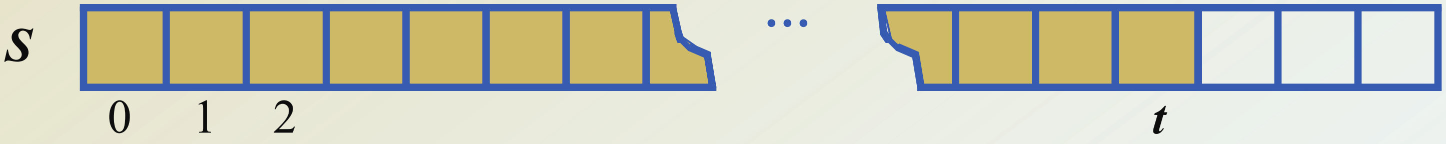
```
main()
{
    int i = 5;
    foo( i );
}

foo(int j)
{
    int k;
    k = j+1;
    bar( k );
}

bar(int m)
{
    …
}
```



Array-based Stack

- A simple way of implementing the Stack ADT uses an **array**.
- We add elements from left to right.
- A variable keeps track of the index of the top element.

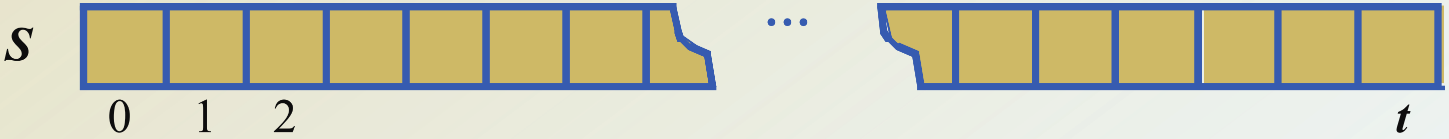
```
Algorithm size( )
    return t + 1

Algorithm pop( )
    if isEmpty( ) then
        throw EmptyStackException
    else
        t ← t − 1
        return S[t + 1]
```



Array-based Stack (cont.)

- The array storing the stack elements may become full.
- A **push** operation will then throw a FullStackException
  - Limitation of the array-based implementation.
  - Not intrinsic to the Stack ADT.

```
Algorithm push(o)
  if $t = S.length - 1$ then
    throw FullStackException
  else
    $t \leftarrow t + 1$
    $S[t] \leftarrow o$
```



Performance and Limitations

- **Performance**
  - Let $\boldsymbol{n}$ be the number of elements in the stack
  - The **space** used is $\boldsymbol{O}(\boldsymbol{n})$
  - Each operation **runs** in time $\boldsymbol{O}(1)$
- **Limitations**
  - The maximum size of the stack must be defined a priori and cannot be changed.
  - Trying to push a new element into a full stack causes an implementation-specific exception.

Array-based Stack in Java

`ArrayStack.java`

Parentheses Matching

- Each “(”, “{”, or “[” must be paired with a matching “)”, “}”, or “[”
  - correct: ( )(( )){([( )])}
  - correct: ((( )(( )){([( )])}
  - incorrect:)(( )){([( )])}
  - incorrect: ({[ ])}
  - incorrect: (

Parentheses Matching Algorithm

Algorithm ParenMatch(*X*,*n*):

Input: An array *X* of *n* tokens, each of which is either a grouping symbol, a variable, an arithmetic operator, or a number

Output: **true** if and only if all the grouping symbols in *X* match

```
Let S be an empty stack
for i=0 to n-1 do
    if X[i] is an opening grouping symbol then
            S.push(X[i])
    else if X[i] is a closing grouping symbol then
            if S.isEmpty() then
                    return false {nothing to match with}
            if S.pop() does not match the type of X[i] then
                    return false {wrong type}
if S.isEmpty() then
    return true {every symbol matched}
else return false {some symbols were never matched}
```

HTML Tag Matching

- For fully-correct HTML, each <name> should pair with a matching </name>

```
<body>
<center>
<h1> The Little Boat </h1>
</center>
<p> The storm tossed the little
boat like a cheap sneaker in an
old washing machine. The three
drunken fishermen were used to
such treatment, of course, but
not the tree salesman, who even as
a stowaway now felt that he
had overpaid for the voyage. </p>
<ol>
<li> Will the salesman die? </li>
<li> What color is the boat? </li>
<li> And what about Naomi? </li>
</ol>
</body>
```

The Little Boat

The storm tossed the little boat like a cheap sneaker in an old washing machine. The three drunken fishermen were used to such treatment, of course, but not the tree salesman, who even as a stowaway now felt that he had overpaid for the voyage.

1. Will the salesman die?
2. What color is the boat?
3. And what about Naomi?

HTML.java

Evaluating Arithmetic Expressions

Slide by Matt Stallmann included with permission.

14 – 3 * 2 + 7 = (14 – (3 * 2)) + 7

Operator precedence

* has precedence over +/–

Associativity

operators of the same precedence group
evaluated from left to right
Example: (x – y) + z rather than x – (y + z)

Idea: push each operator on the stack, but first pop and perform higher and *equal* precedence operations.

Algorithm for Evaluating Expressions

Slide by Matt Stallmann included with permission.

Two stakcs:

- **opStk** holds operators
- **valStk** holds values
- Use **$** as special **“end of input”** token with lowest precedence

```
Algorithm doOp()
    x ← valStk.pop();
    y ← valStk.pop();
    op ← opStk.pop();
    valStk.push( y op x )

Algorithm repeatOps( refOp ):
 while ( valStk.size() > 1 ∧
          prec(refOp) ≤
          prec(opStk.top())
    doOp()
```

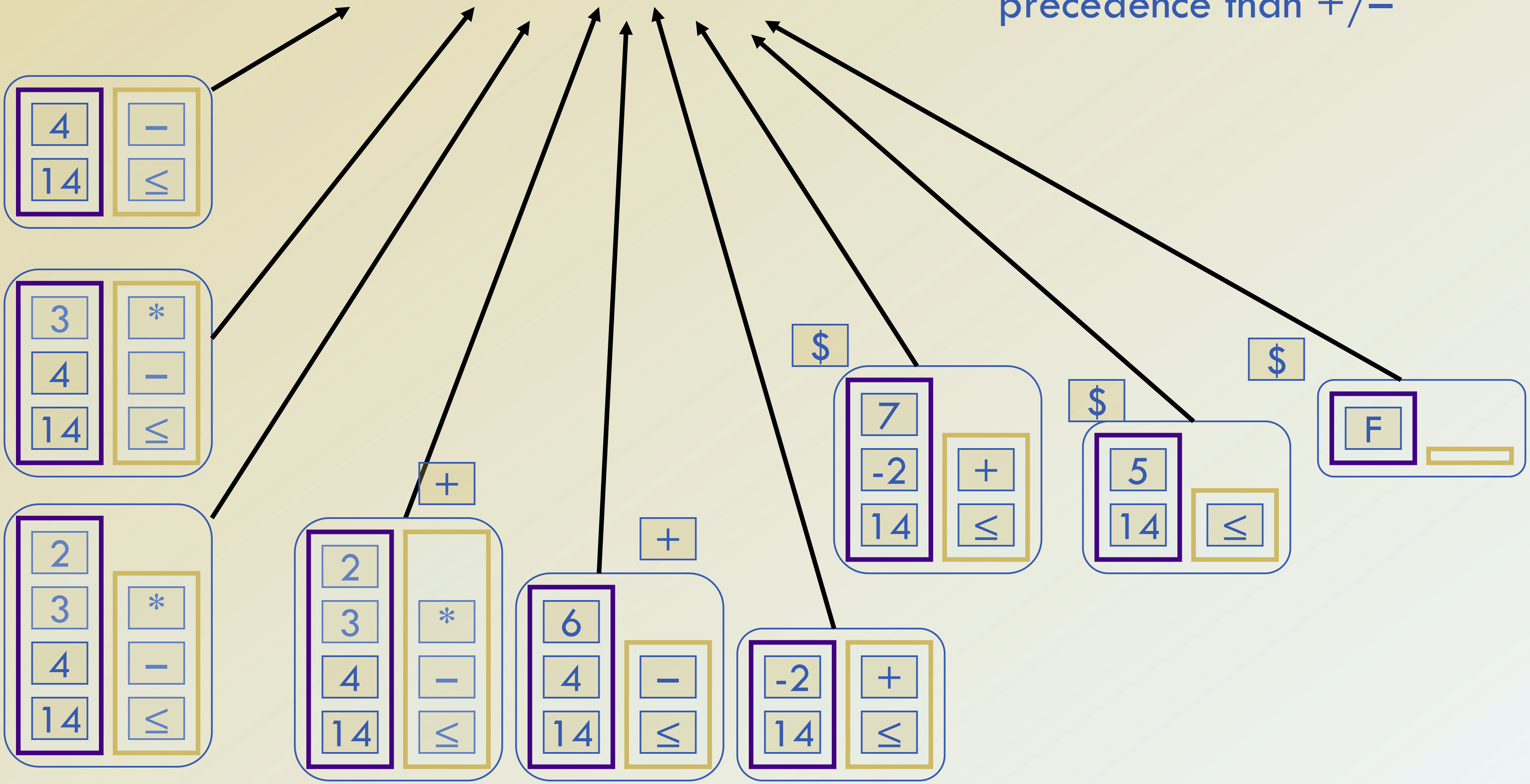
```
Algorithm EvalExp( )
    Input: a stream of tokens
        representing an arithmetic
        expression (with numbers)
    Output: the value of the expression

while there’s another token z
  if isNumber(z) then
        valStk.push(z)
  else
        repeatOps(z);
        opStk.push(z)
repeatOps($);
return valStk.top()
```

Algorithm on an Example Expression

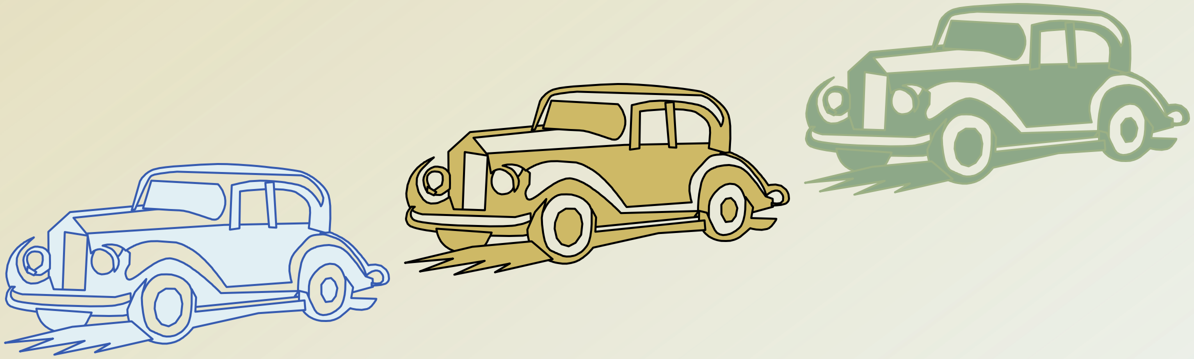
Slide by Matt Stallmann included with permission.

14 ≤ 4 – 3 * 2 + 7

Operator ≤ has lower precedence than +/–



Queues



The Queue ADT

- The Queue ADT stores arbitrary objects.
- Insertions and deletions follow the **first-in first-out** scheme.
- **Insertions are at the rear** of the queue and **removals are at the front** of the queue.
- Main queue operations:
  - **enqueue(object):** inserts an element at the end of the queue.
  - **object dequeue( ):** removes and returns the element at the front of the queue.
- Auxiliary queue operations:
  - **object front( ):** returns the element at the front without removing it.
  - **integer size( ):** returns the number of elements stored
  - **boolean isEmpty( ):** indicates whether no elements are stored
- Exceptions
  - Attempting the execution of dequeue or front on an empty queue throws an EmptyQueueException

Example

Operation	*Output*	*Queue Content*
enqueue(5)		
enqueue(3)		
dequeue()		
enqueue(7)		
dequeue()		
front()		
dequeue()		
dequeue()		
isEmpty()		
enqueue(9)		
enqueue(7)		
size()		
enqueue(3)		
enqueue(5)		
dequeue()		

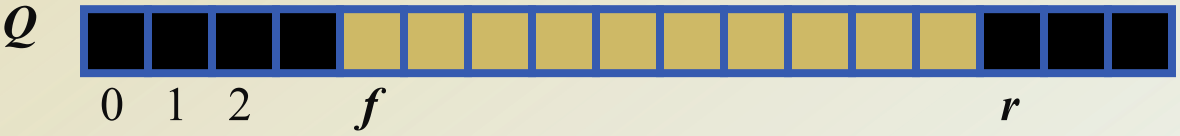
Applications of Queues

- **Direct applications**
  - Waiting lists, bureaucracy,
  - Access to shared resources (e.g., printer).
  - Multiprogramming.
- **Indirect applications**
  - Auxiliary data structure for algorithms.
  - Component of other data structures.

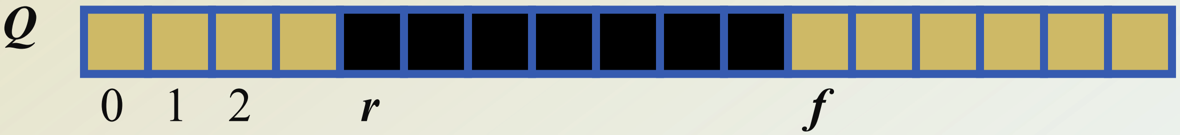
Array-based Queue

- Use an array of size *N* in a circular fashion.
- Two variables keep track of the front and rear
  - ***f*** - index of the front element
  - ***r*-**index immediately past the rear element
- Array location ***r*** is kept empty.

normal configuration



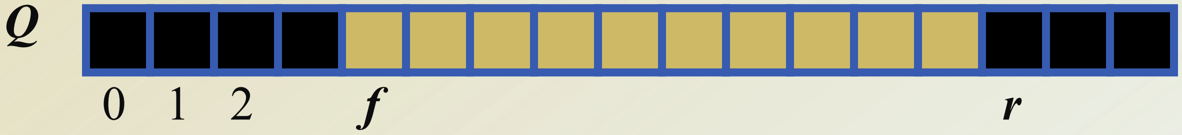
wrapped-around configuration

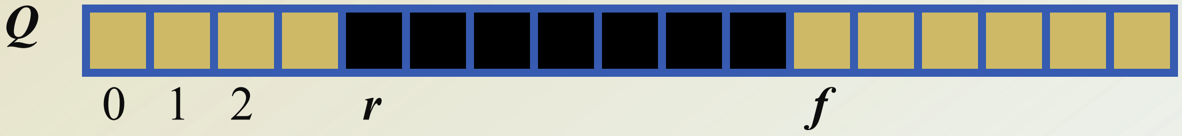


Queue Operations

- We use the modulo operator (remainder of division)

```
Algorithm size( )
    return $(N - f + r) \bmod N$

Algorithm isEmpty( )
    return $(f = r)$
```

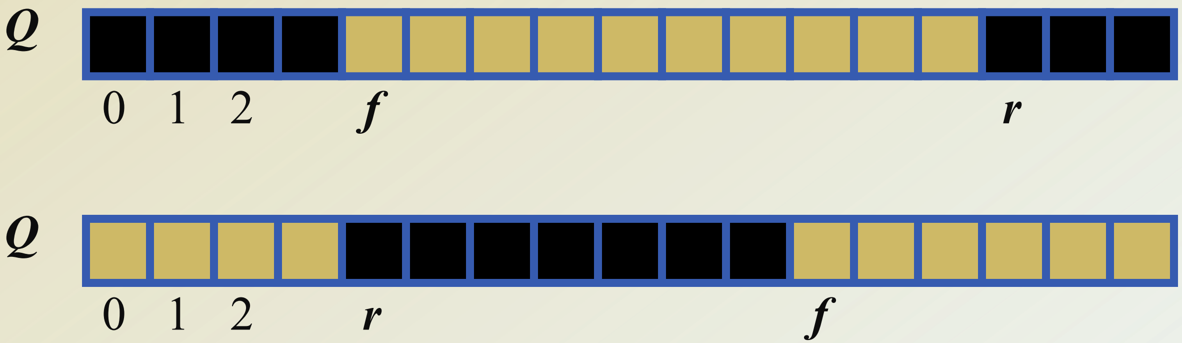




Queue Operations (cont.)

- Operation enqueue throws an exception if the array is full
- This exception is implementation-dependent

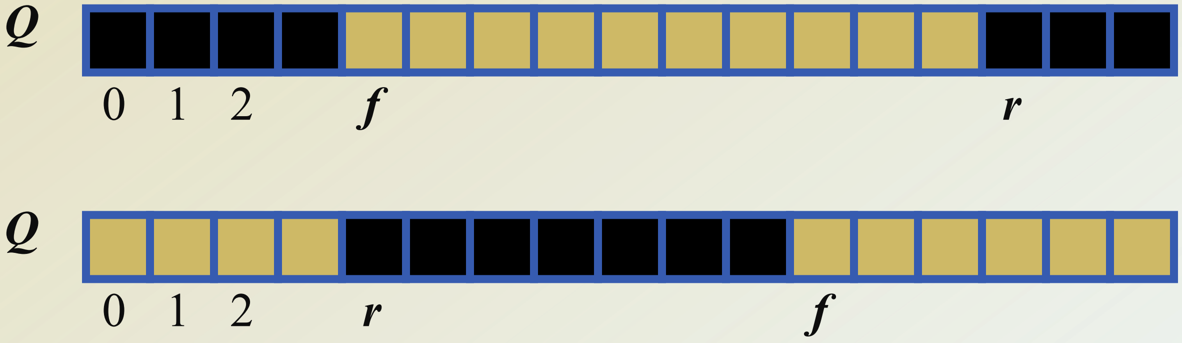
```
Algorithm enqueue( o )
  if size( ) = N – 1 then
    throw FullQueueException
  else
    Q[ r ] ← o
    r ← (r + 1) mod N
```



Queue Operations (cont.)

- Operation dequeue throws an exception if the queue is empty.
- This exception is specified in the queue ADT.

```
Algorithm dequeue( )
  if isEmpty( ) then
    throw EmptyQueueException
  else
    $o \leftarrow Q[f]$
    $f \leftarrow (f + 1) \bmod N$
    return $o$
```



Queue Interface in Java

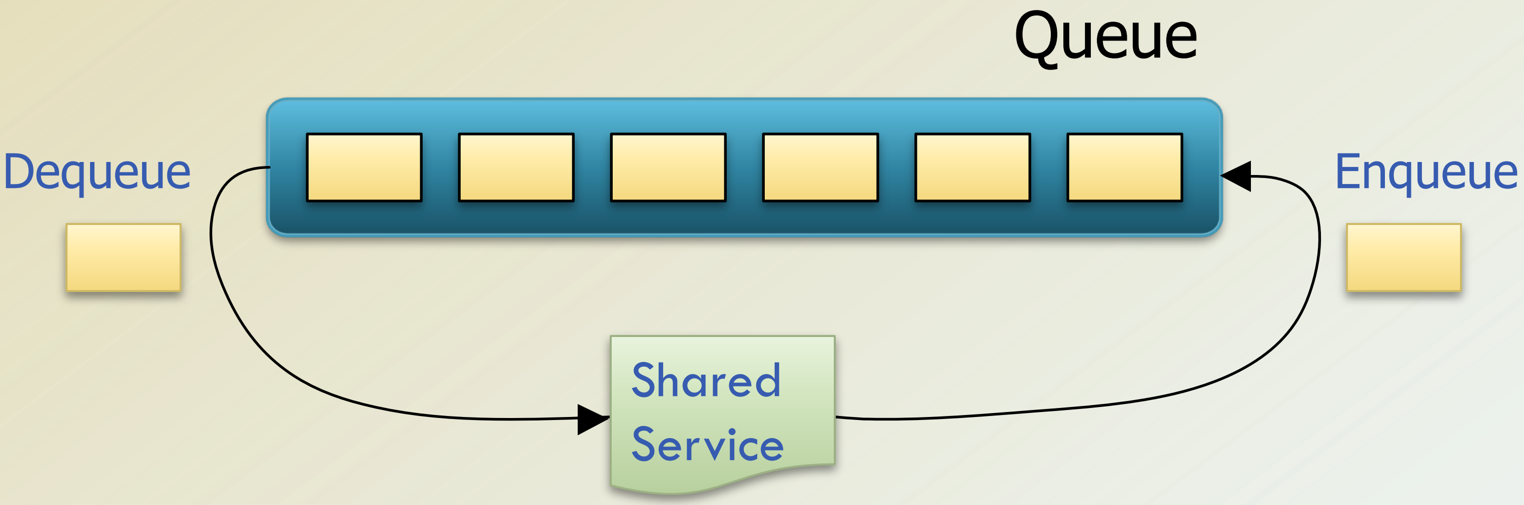
- Java interface corresponding to our Queue ADT.
- Requires the definition of class **`EmptyQueueException`**
- No corresponding built-in Java class.

`Queue.java`

Application: Round Robin Schedulers

- We can implement a round robin scheduler using a queue Q by repeatedly performing the following steps:

1. e = Q.dequeue()
2. Service element e
3. Q.enqueue(e)



End of Chapter 5