DM550 / DM857
Introduction to Programming

Peter Schneider-Kamp

petersk@imada.sdu.dk

http://imada.sdu.dk/~petersk/DM550/

http://imada.sdu.dk/~petersk/DM857/

JAVA PROJECT

Organizational Details

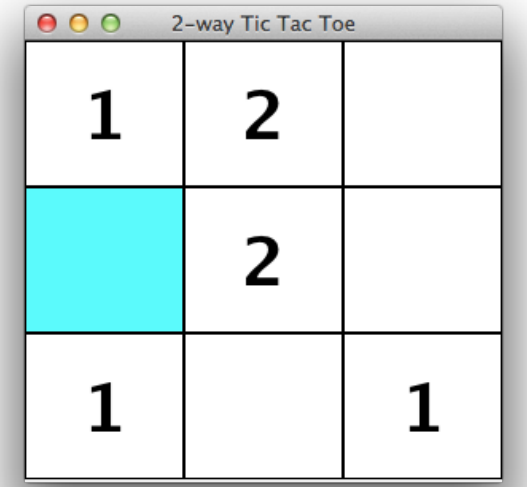
- exam = Python project + Java project + 150 assessment points
- Java project in groups of 3 (write for other sizes)

- Select 1 out of 3 different tracks
  - Connect Four, Go, Puzzle Games
- Select 1 out of 3 different user interfaces
  - Android app, CLI, Swing GUI

- Deliverables:
  - Written 10-20 page report as specified in project description
  - Handed in electronically & physically
  - Deadline: Friday, January 12, 23:59

Tasks 1-3/4: Tic Tac Toe

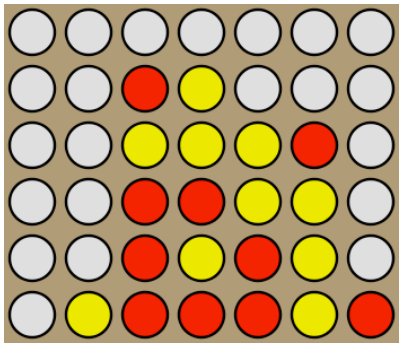
- Tic Tac Toe: simple 2 player board game played on a 3 x 3 grid
- extended rules for n-way Tic Tac Toe:
  - n players
  - (n+1) x (n+1) grid
  - 3 marks in a row, column, diagonal

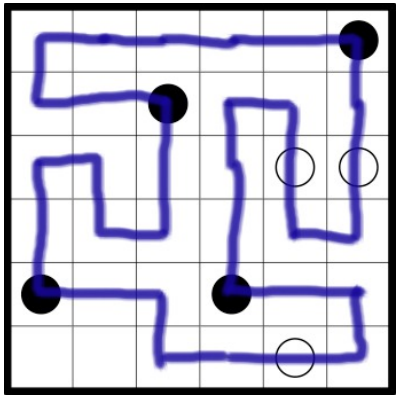


- **Goal:** complete an implementation of n-way Tic Tac Toe
- **Challenges:** Interfaces, UI, Array Programming

Task 5, 6 or 7: Beyond Tic Tac Toe

- Task 5: Connect Four (obligatory level – easiest)
  - implementable as a slight modification and generalization of 2-way Tic Tac Toe
- Task 6: Go (supplementary level – medium difficulty)
  - rich board game in a league with chess
- Task 7: Puzzle Games (challenge level – hardest)
  - Sudoku, Kakuro, Masyu, Nurikabe, Kuromasu, Fillomino, Battleship, Sokoban
  - generator, user interface, solver





ABSTRACT DATA TYPES FOR STACKS & QUEUES

Stacks

- stacks are special sequences, where elements are only added and removed at one end
- imagine a stack of paper on a desk
- many uses:
  - postfix calculator
  - activation records
  - depth-first tree traversals
  - …
- basic stack operations are
  - looking at the top of the stack
  - removing the top-most element
  - adding an element to the top of the stack

Stack ADT: Specification

- data are arbitrary objects of class E
- operations are defined by the following interface

```
public interface Stack<E> {
    public boolean isEmpty();          // is stack empty?
    public E peek();                   // look at top element
    public E pop();                    // remove top element
    public void push(E elem);          // add top element
}
```

Stack ADT: Design 1

- Design 1: use dynamic array
  - the top of the stack is the end of the list
  - in other words, num specifies the top position
  - pushing corresponds to adding at the end
  - poping corresponds to removing at the end

Stack ADT: Implementation 1

- Implementation 1:

```
public class DynamicArrayStack<E> implements Stack<E> {
    private int limit;              // maximal number of elements
    private E[] data;               // elements of the list
    private int num = 0;            // current number of elements
    public DynamicArrayStack(int limit) {
        this.limit = limit;
        this.data = (E[]) new Object[limit];
    }
    public boolean isEmpty() {  return this.num == 0;  }
    …
}
```

Stack ADT: Implementation 1

- Implementation 1 (continued):

```
public class DynamicArrayStack<E> implements Stack<E> {   …
    public E peek() {
        if (this.isEmpty()) {  throw new RuntimeException("es");  }
        return this.data[this.num-1];
    }
    public E pop() {
        E result = this.peek();
        num--;
        return result;
    }   …
}
```

Stack ADT: Implementation 1

- Implementation 1 (continued):

```
public class DynamicArrayStack<E> implements Stack<E> {   …
    public void push(E elem) {
        if (this.num >= this.limit) {
            E[] newData = (E[]) new Object[2*this.limit];
            for (int j = 0; j < limit; j++) {  newData[j] = data[j];  }
            this.data = newData;
            this.limit *= 2;
        }
        this.data[num++] = elem;
    }
}
```

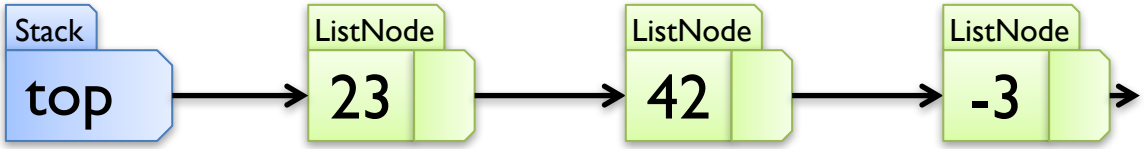
Stack ADT: Design & Implement. 2

- Design 2: reuse dynamic array list (ArrayList<E>)
- Implementation 2:

```
public class ArrayListStack<E> implements Stack<E> {
    private List<E> list = new ArrayList<E>();
    public boolean isEmpty() {  return this.list.isEmpty();  }
    public E peek() {  return this.list.get(this.list.size()-1);  }
    public E pop() {  return this.list.remove(this.list.size()-1);  }
    public void push(E elem) {  this.list.add(elem);  }
}
```

Stack ADT: Design 3

- Design 3: use recursive data structure
  - linked lists have cheap insert and remove operations
  - adding at the end requires running to the end
  - represent top as the beginning of the "list"
- reuse linked list node class (ListNode<E>)
- with dynamic arrays, sometimes need to copy full array
- with linked list, always constant time operations



Stack ADT: Implementation 3

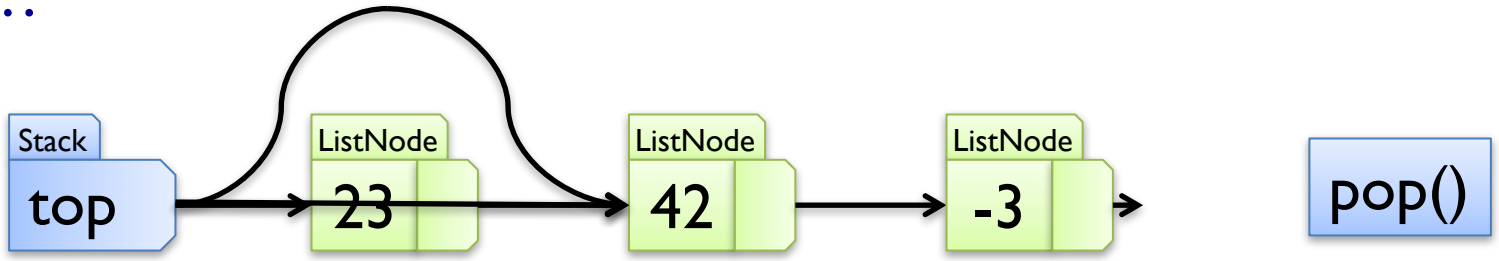
- Implementation 3:

```
public class LinkedStack<E> implements Stack<E> {
    private ListNode<E> top = null;    // top of the stack
    public boolean isEmpty() {  return this.top == null;  }
    public E peek() {
        if (this.isEmpty()) {  throw new RuntimeException("es");  }
        return this.top.get(0);
    }
    …
}
```

Stack ADT: Implementation 3

- Implementation 3 (continued):

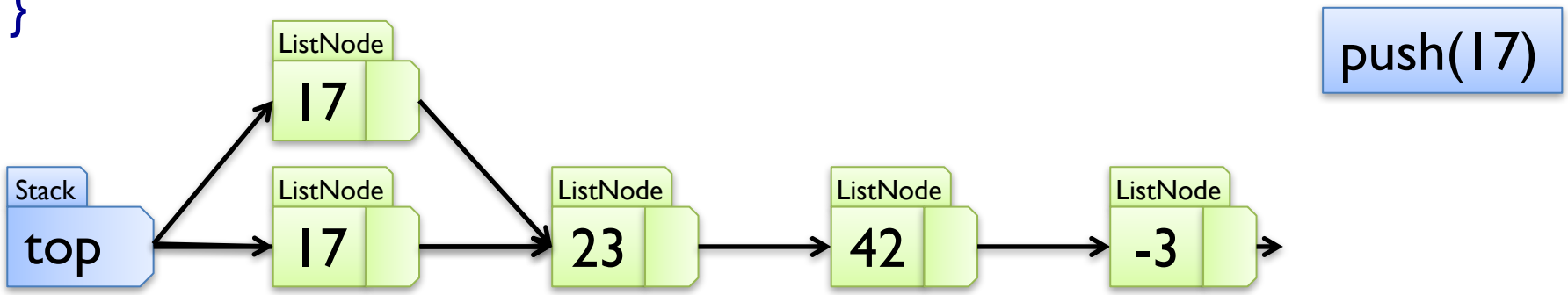
```
public class LinkedStack<E> implements Stack<E> {
    …
    public E pop() {
        E result = this.peek();
        this.top = this.top.getNext();
        return result;
    }
    …
}
```



Stack ADT: Implementation 3

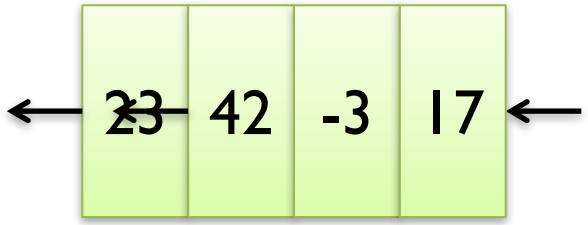
- Implementation 3 (continued):

```
public class LinkedStack<E> implements Stack<E> {
    private ListNode<E> top = null;    // top of the stack
    …
    public void push(E elem) {
        this.top = new ListNode<E>(elem, this.top);
    }
}
```



Queues

- queues are special sequences, where elements are added on one and removed at the other end
- imagine a waiting line in the supermarket
- many uses:
  - network send/receive buffers
  - process scheduling
  - breadth-first tree traversals
  - …
- basic queue operations are
  - looking at the beginning of the queue
  - removing the first element
  - adding an element to the end of the queue



Queue ADT: Specification

- data are arbitrary objects of class E
- operations are defined by the following interface

```
public interface Queue<E> {
    public boolean isEmpty();          // is queue empty?
    public E peek();                   // look at first element
    public E poll();                   // remove first element
    public boolean offer(E elem);      // true, if element added
            // at end of queue; false, if queue is full
}
```

Queue ADT: Design & Implement. I

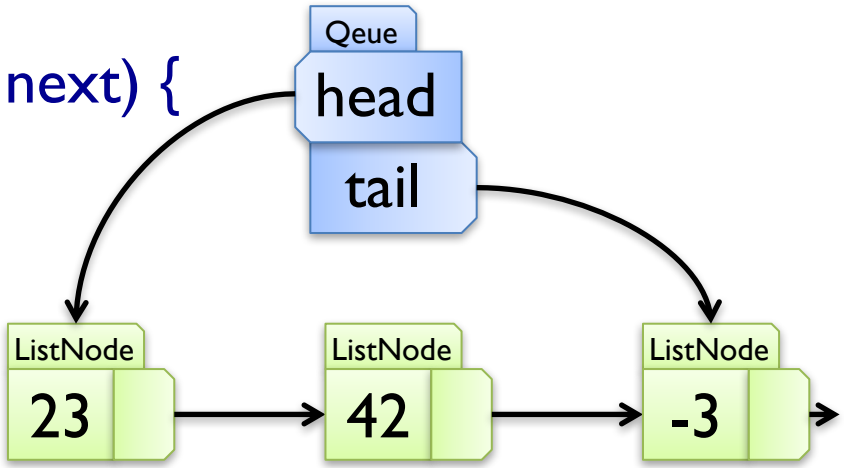
- Design 1: reuse dynamic array list (ArrayList<E>)
- Implementation 1:

```
public class ArrayListQueue<E> implements Queue<E> {
    private List<E> list = new ArrayList<E>();
    public boolean isEmpty() {  return this.list.isEmpty();  }
    public E peek() {  return this.list.get(0);  }
    public E poll() {  return this.list.remove(0);  }
    public boolean offer(E elem) {
        this.list.add(elem);
        return true;
    }
}
```

Queue ADT: Design & Implement. 2

- Design 2: use recursive data structure
  - use two references instead of one
  - one reference to end of queue
  - one reference to beginning of queue
- reuse & extend linked list node class (ListNode<E>)
- Implementation 2:

```
public class ListNode<E> {   …
    public void setNext(ListNode<E> next) {
        this.next = next;
    }
}
```



Queue ADT: Implementation 2

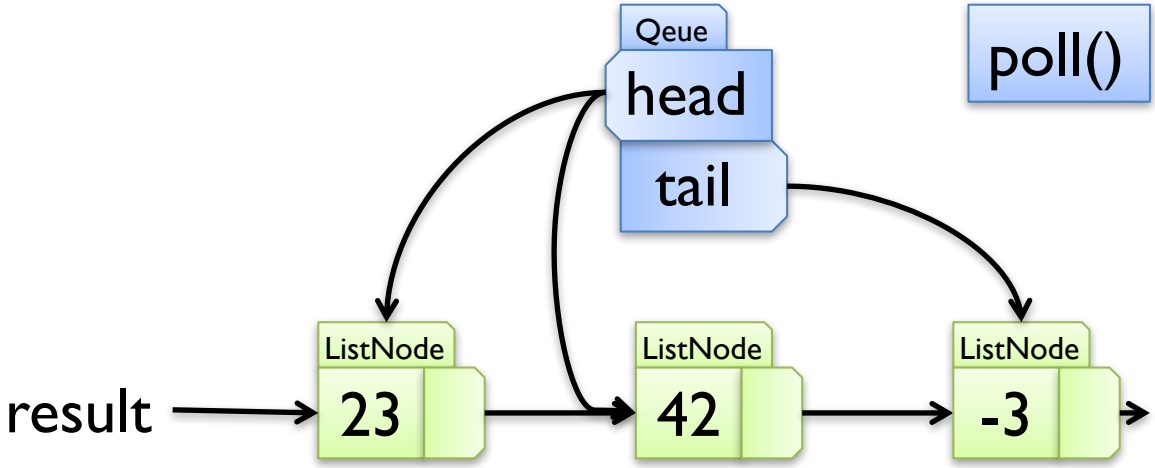
- Implementation 2 (continued):

```
public class LinkedQueue<E> implements Queue<E> {
    private ListNode<E> head = null;          // beginning
    private ListNode<E> tail = null;          // end
    public boolean isEmpty() {
        return this.head == null;
    }
    public E peek() {
        return this.head.get(0);
    }
    …
}
```

Queue ADT: Implementation 2

- Implementation 2 (continued):

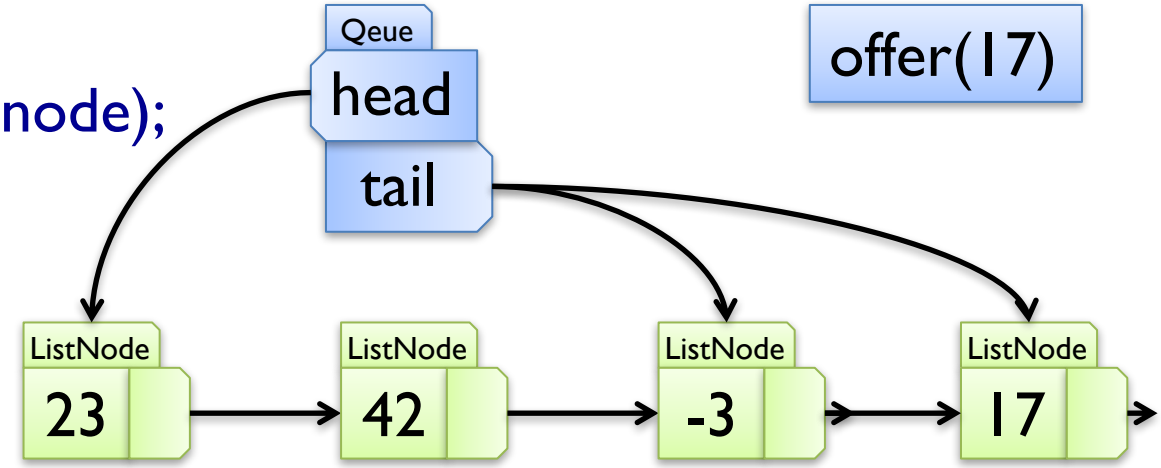
```
public class LinkedQueue<E> implements Queue<E> {   …
    public E poll() {
        E result = this.peek();
        this.head = this.head.getNext();
        if (this.head == null) {
            this.tail = null;
        }
        return result;
    }
    …
}
```



Queue ADT: Implementation 2

- Implementation 2 (continued):

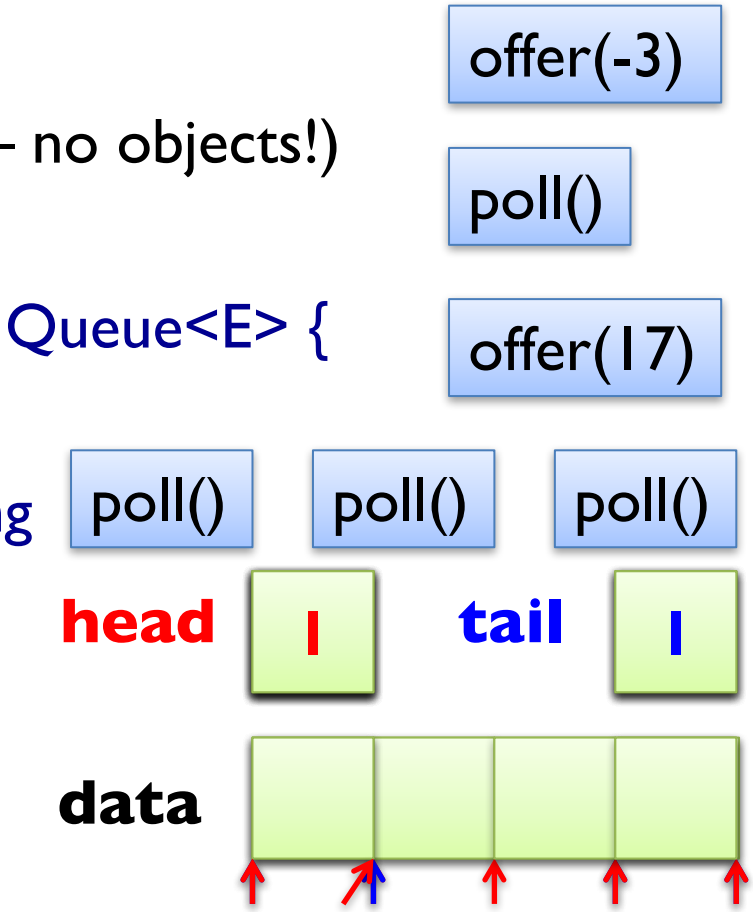
```
public class LinkedQueue<E> implements Queue<E> {   …
    public boolean offer(E elem) {
        ListNode<E> node = new ListNode<E>(elem, null);
        if (this.head == null) {
            this.head = this.tail = node;
        } else {
            this.tail.setNext(node);
            this.tail = node;
        }
        return true;
    }  }
```



Queue ADT: Design & Implement. 3

- Design 3: use a fixed length array
  - use two indices denoting beginning and end
  - wrap around end of array
- very efficient (memory and runtime – no objects!)
- Implementation 3:

```
public class RingQueue<E> implements Queue<E> {
    private int limit;
    private int head = 0;          // beginning
    private int tail = 0;          // end
    private E[] data;
    …
}
```



Queue ADT: Implementation 3

- Implementation 3 (continued):

```
public class RingQueue<E> implements Queue<E> {   …
    private int head = 0;       // beginning
    private int tail = 0;       // end
    private E[] data;
    public boolean isEmpty() {  return this.head == this.tail;  }
    public E peek() {
        if (this.isEmpty()) {  throw new RuntimeException("eq");  }
        return this.data[this.head];
    }
    …
}
```

Queue ADT: Implementation 3

- Implementation 3 (continued):

```
public class RingQueue<E> implements Queue<E> {
    …
    public E poll() {
        E result = this.peek();
        this.head = (this.head+1) % this.limit;
        return result;
    }
    …
}
```

Queue ADT: Implementation 3

- Implementation 3 (continued):

```
public class RingQueue<E> implements Queue<E> {   …
    public boolean offer(E elem) {
        int newTail = (this.tail+1) % this.limit;
        if (newTail == this.head) {
            return false;                // full
        }
        this.data[this.tail] = elem;
        this.tail = newTail;
        return true;
    }   …
}
```

COLLECTION CLASSES FOR STACKS & QUEUES

Queue ADT: Specification & Implem.

- interface Queue<E> extends Collection<E>
- data are arbitrary objects of type E
- defines additional operations over Collection<E>:
  - public boolean offer(E e); // alternative name to add
  - public E peek(); // return head
  - public E element(); // alternative name to peek
  - public E poll(); // remove and return head
- extended again by interface Deque<E> providing support for adding AND removing at both ends
- Implementations:
  - ArrayDeque – with offer == offerLast and poll == pollFirst
  - LinkedList – only useful, when not a pure Queue

Stack ADT: Specification & Implem.

- class Stack<E> implements Collection<E>
- data are arbitrary objects of type E
- defines additional operations over Collection<E>:
  - public E push(E e); // add on top of stack
  - public E peek(); // return top element
  - public E pop(); // remove and return top
  - public int search(Object o); // return 1-based index
- superseded by interface Deque<E> providing support for adding AND removing at both ends
- Alternative Implementations:
  - ArrayDeque – with push == addFirst and pop == removeFirst

Deque ADT: Specification & Implem.

- interface Deque<E> extends Collection<E>
- data are arbitrary objects of type E
- defines additional operations over Collection<E>:
  - addFirst, offerFirst, addLast, offerLast
  - removeFirst, pollFirst, removeLast, pollLast
  - getFirst, peekFirst, getLast, peekLast
- add*, remove*, get* throw exceptions
- offer*, poll*, peek* return special value
- Implementations:
  - ArrayDeque – fast and preferred
  - LinkedList – only use when more than Deque needed