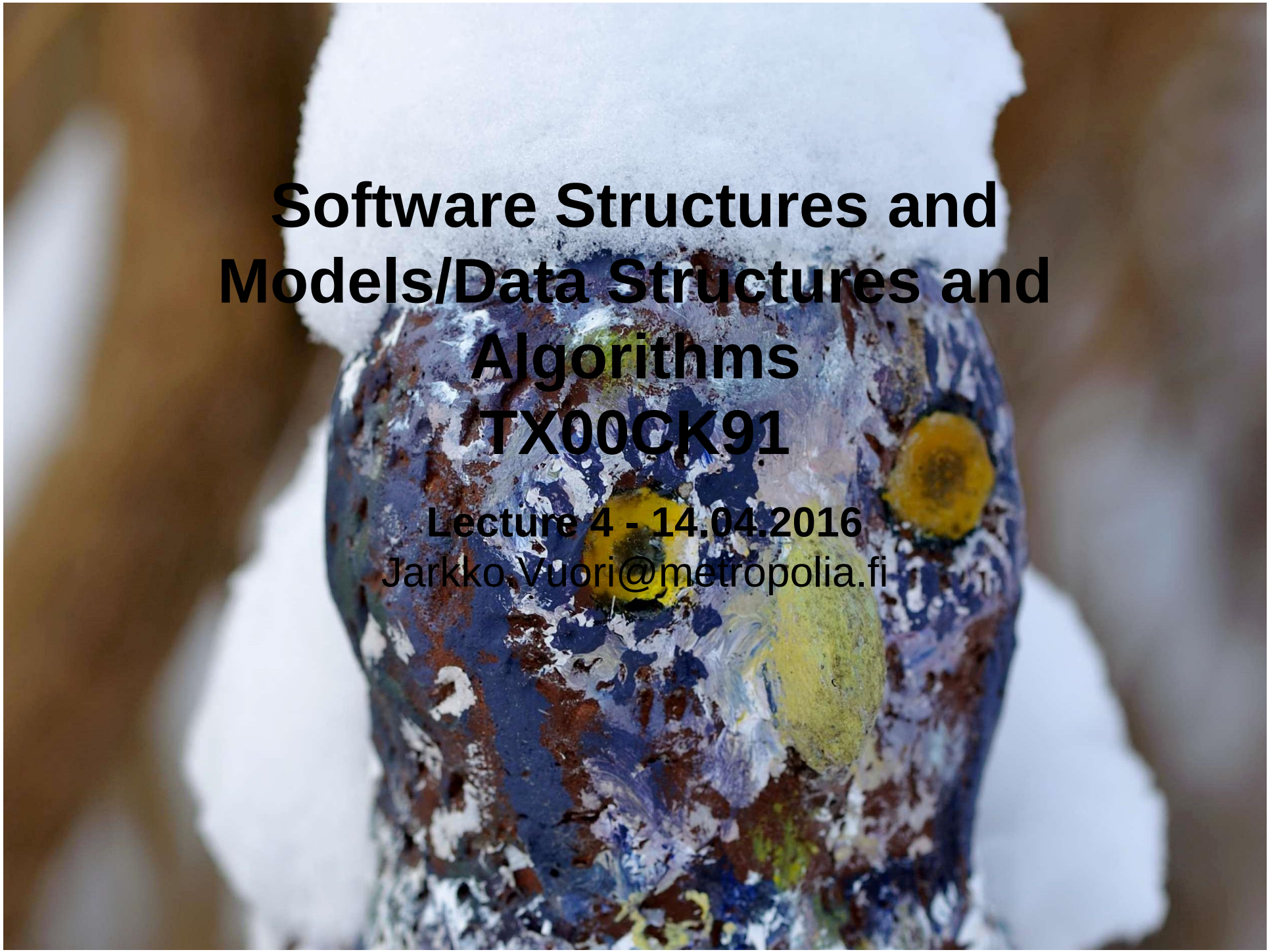


Software Structures and Models/Data Structures and Algorithms TX00CK91

Lecture 4 - 14.04.2016

Jarkko.Vuori@metropolia.fi

Java Collections Stack

- Stack is a subclass of Vector that implements a standard last-in, first-out stack
- Stack only defines the default constructor, which creates an empty stack. Stack includes all the methods defined by Vector, and adds several of its own

- boolean empty()
  - Tests if this stack is empty. Returns true if the stack is empty, and returns false if the stack contains elements
- Object peek( )
  - Returns the element on the top of the stack, but does not remove it
- Object pop( )
  - Returns the element on the top of the stack, removing it in the process
- Object push(Object element)
  - Pushes element onto the stack. element is also returned
- int search(Object element)
  - Searches for element in the stack. If found, its offset from the top of the stack is returned. Otherwise, -1 is returned

Java Collections Stack

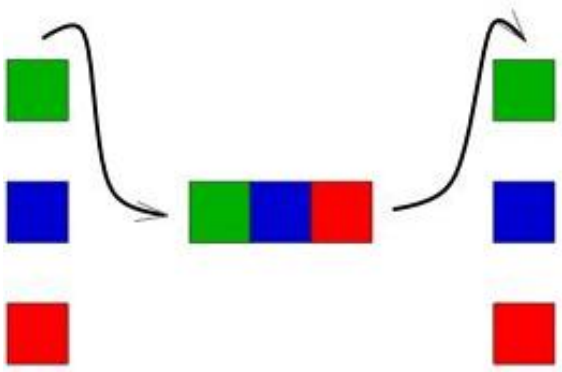
- Program on the right produces the following output

```
stack: [ ]
push(42)
stack: [42]
push(66)
stack: [42, 66]
push(99)
stack: [42, 66, 99]
pop -> 99
stack: [42, 66]
pop -> 66
stack: [42]
pop -> 42
stack: [ ]
pop -> empty stack
```

```
import java.util.*;

public class StackDemo {

   static void showpush(Stack st, int a) {
      st.push(new Integer(a));
      System.out.println("push(" + a + ")");
      System.out.println("stack: " + st);
   }

   static void showpop(Stack st) {
      System.out.print("pop -> ");
      Integer a = (Integer) st.pop();
      System.out.println(a);
      System.out.println("stack: " + st);
   }

   public static void main(String args[]) {
      Stack st = new Stack();
      System.out.println("stack: " + st);
      showpush(st, 42);
      showpush(st, 66);
      showpush(st, 99);
      showpop(st);
      showpop(st);
      showpop(st);
      try {
         showpop(st);
      } catch (EmptyStackException e) {
         System.out.println("empty stack");
      }
   }
}
```

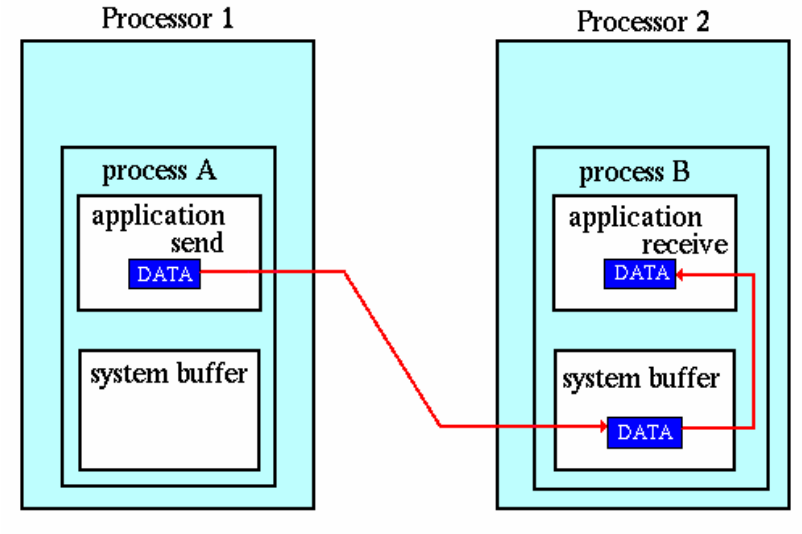
Queue, the definition

- Queue (Jono) is a container that preserves order (order of entering) and to which the following operations are defined
  1. initialize (initializes the queue)
  2. enqueue (add an element to the end of the queue)
  3. dequeue (removes an element from the beginning of the queue)
  4. is_empty (test whether the queue is emtpy)
- Why we need the queue, why stack and list are not enough?
  - Stack will reverse the ordering
  - List can have whatever ordering, queue has only the order of entering
    - And therefore queue is often more efficient
- We call often the queue as FIFO (First-In First-Out) buffer



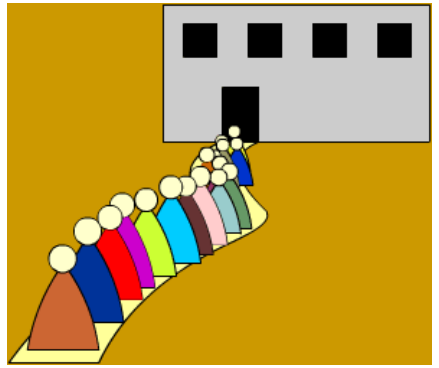
Queue, application areas

- Applications
  - Buffering (puskurointi) between different processes
    - Between NIC (network interface card) and the operating system
    - Keyboard buffer
  - Buffering is needed because those processes (or devices) operate at different speed
    - because the other process is occasionally producing more data than the other is consuming (they have different rates)
- Why a queue, why not list in everywhere?
  - Because queue can be implemented more efficiently (than a list)
    - Very important when buffering data transmissions between high-speed processes
    - There exist even hardware chips for data buffering



Queue, implementation

- Simple array implementation
  - When we remove elements from the queue, we move other elements forward (like queues in supermarkets)
  - Moving the elements can be ”heavy” operation if there are many elements and the elements are large in size
    - We always try to avoid unnecessary (where no “action” to the data is done) data movement from the one memory location to the other
      - This causes unnecessary traffic to the data bus which is already loaded (fetching instructions and operands from the memory)



Queue, implementation

- We'll implement the queue in different ways
- This is our test application for the queue
  - It is possible to enter characters to the queue, and every time a character is enqued, the content of the whole queue is shown

```
public static void main(String[] args) {
    simpleQueue<Character> queue = new simpleQueue();
    Character              item;

    System.out.println("Enter a letter to push onto stack");
    System.out.println("or digit 1 to dequeue a letter");
    System.out.println("Return to end the program\n");
    try {
        item = new Character((char)System.in.read());
        while (item.compareTo(new Character('\n')) != 0) {
            if (item.compareTo(new Character('1')) == 0)
                System.out.println("A letter dequeued " + queue.dequeue());
            else
                queue.enqueue(item);
            System.out.print("Queue content: [");
            queue.print();
            System.out.println("]");

            item = new Character((char)System.in.read());
        }
        System.out.println();
    } catch(Exception e) {
        System.out.println("Exception " + e);
    }
}
```

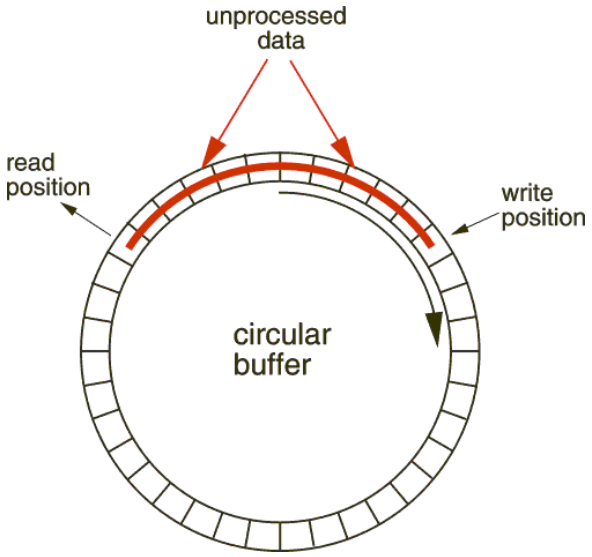
Queue, simple array implementation

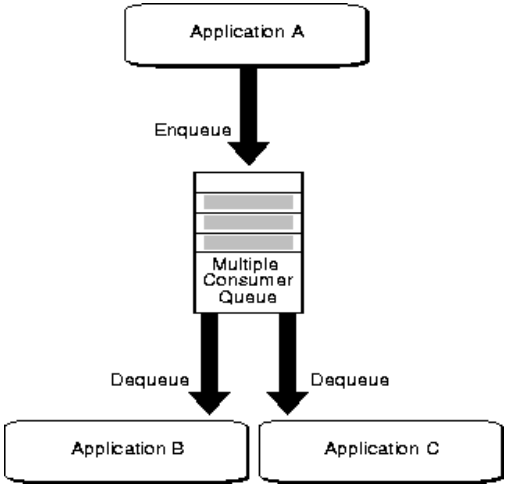
- This simple method stores items directly to the array
  - First element of the queue will always be at the first location
  - Last element of the queue, and the insertion point of the new element to be added, is at the end of the queue
    - And the number_of_items will tell us where is the end of the queue
  - When dequeueing, we remove the first element of the array and then copy element by element the elements after the removed element (to fill the space of the previous first element)
- The problem of this implementation is the moving of elements when we remove the first element of the queue
- Also the size of the queue is fixed

```
public class simpleQueue<T extends Comparable<T>> {
    private static final int MAXN = 10;
    private int number_of_items;
    private T[] array;

    public simpleQueue() {
        number_of_items = 0;
        array = (T[])new Comparable[MAXN];
    }

    public boolean enqueue(T item) {
        if (number_of_items >= MAXN)
            return false;
        else
            array[number_of_items++] = item;
            return true;
    }

    public T dequeue() {
        if (number_of_items == 0)
            return null;
        else {
            T item = array[0];
            for (int i = 0; i < number_of_items-1; i++)
                array[i] = array[i+1];
            number_of_items--;
            return item;
        }
    }

    public void print() {
        for (int i = 0; i < number_of_items; i++)
            System.out.print(array[i] + " ");
    }
```

Queue, circular implementation

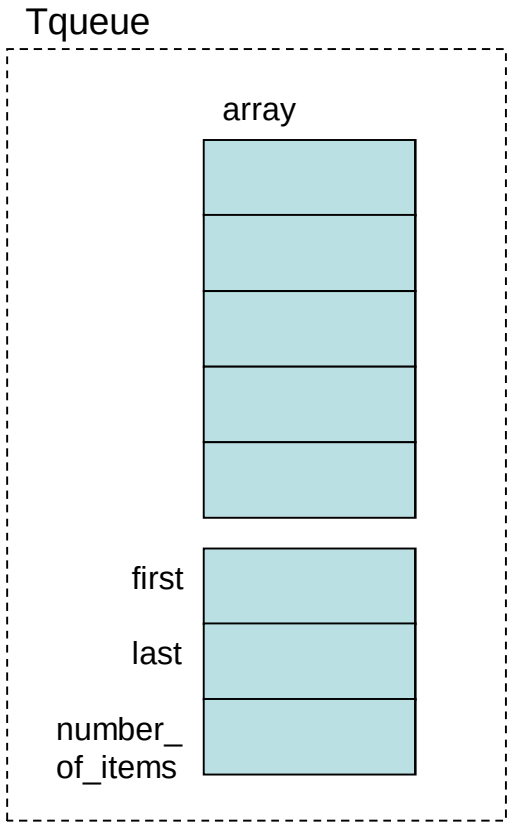
- Circular queue (buffer) with array implementation
  - Unnecessary data movement is avoided
  - We need variables which tell to what place we should insert a new item (write) and from what place we should remove an item (read)
  - Note that the physical implementation of the queue is different than logical implementation
    - Now we don't need data movement anymore, beginning of the queue and end of the queue moves their places
- The size of the queue is still fixed

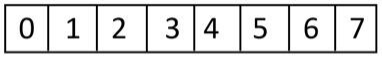


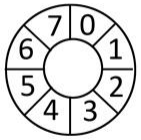


Queue, circular array implementation

- Elements to be queued are stored to the array
- variable last gives the index of the previously stored value (write position)
  - Value of -1 means an empty queue
- Variable first gives the index of the first element in the queue (read position)
- When inserting a new element to the queue, we advance last variable to point to the next element, and then we write the element to that new index
- Elements are removed from the queue at index position marked by the variable first
- When advancing first or last variable, if we go to the outside of the array, we set the variable to the index of the first element in array (normally 0); we look the array as a ring
- Variable number_of_items gives the current number of elements in the queue
  - Because the number of item is difficult to calculate from the first and last variables







Queue, circular array implementation

- Circular buffer is implemented by testing the index to the end of the buffer and resetting the index if necessary (`if (++last > MAXN-1) last = 0`)
  - Sometimes lenght of the buffer is set to be 2^n, then we can get rid of the testing operation (`last = (last+1) & 2^n-1`)

```
public class circularQueue<T extends Comparable<T>> {
    private static final int MAXN = 10;
    private int first, last, number_of_items;
    private T[] array;

    public circularQueue() {
        first = 0;
        last = -1;
        number_of_items = 0;
        array = (T[])new Comparable[MAXN];
    }

    public boolean enqueue(T item) {
        if (number_of_items >= MAXN)
            return false;
        else
            if (++last > MAXN-1) last = 0;
            array[last] = item;
            number_of_items++;
            return true;
    }

    public T dequeue() {
        if (number_of_items == 0)
            return null;
        else {
            T item = array[first++];
            if (first > MAXN-1) first = 0;
            number_of_items--;
            return item;
        }
    }

    public void print() {
        int i;

        i = first;
        for (int n = 0; n < number_of_items; n++) {
            System.out.print(array[i] + " ");
            if (++i > MAXN-1) i = 0;
        }
    }
```

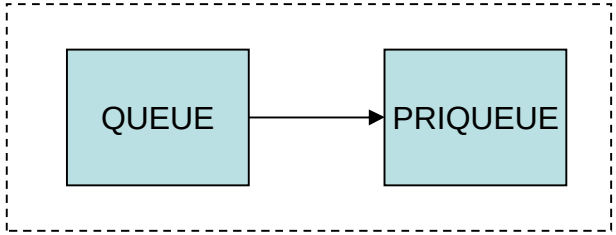
Queue interface in Java Collections

- Queue in Java collections is an interface you need to instantiate a concrete implementation of the interface in order to use it
- You can choose between the following Queue implementations in the Java Collections API:
  - java.util.LinkedList
  - java.util.PriorityQueue
- LinkedList is a pretty standard queue implementation

```
import java.util.*;

public class collectionsQueue<T extends Comparable<T>> {
    public static void main(String[] args) {
        Queue<Character> queue = new LinkedList<Character>();

        Character              item;

        System.out.println("Enter a letter to push onto stack");
        System.out.println("or digit 1 to dequeue a letter");
        System.out.println("Return to end the program\n");
        try {
            item = new Character((char)System.in.read());
            while (item.compareTo(new Character('\n')) != 0) {
                if (item.compareTo(new Character('1')) == 0)
                    System.out.println("A letter dequeued " + queue.remove());
                else
                    queue.add(item);

                System.out.print("Queue content: [");
                for(Character anObject : queue)
                    System.out.print(anObject);
                System.out.println("]");

                item = new Character((char)System.in.read());
            }
            System.out.println();
        } catch(Exception e) {
            System.out.println("Exception " + e);
        }
    }
}
```

Priority Queue

- We can build new abstract datatypes using existing (abstract) datatypes
- Priority queue is a queue, where elements have (some) priority
  - Priority queue is a new abstract data type which is based on an existing abstract datatype Queue
- When an element is added to the queue, it will be placed to the end of the queue if all elements already in the queue have higher priority than the new element
- New element will be added to the beginning of the queue, if there exists no element with higher (or same) priority
- The rule is: new element bypasses all lower priority elements
- Priority queue can be used in operating system with a scheduler
  - Higher priority task will be processed before lower priority tasks
- Priority queue can be implemented in many ways
  - Here we have an implementation that is built with separate “normal” queues. We have three priorities (1, 2 and 3)



And this still looks as a queue to the outside world

Priority Queue

- When enqueing, elements are placed to queues based on priority
- When dequeuing, elements are read first from the higher priority queue
  - If it is empty, the next lower priority queue is tried
- Implementation of the priority queue is not visible to the outside
  - Visible are only those functions that are the same type than original queue (init, enque, dequeue)

```
public boolean enqueue(T item, int pri) {
    bool status = false;

    switch (pri) {
    case 1: status = queue1.enqueue(item);
        break;
    case 2: status = queue2.enqueue(item);
        break;
    case 3: status = queue3.enqueue(item);
        break;
    }
    return (status);
}

public T dequeue() {
    if (!queue1.empty()) {
        queue1.dequeue(item);
        return (true);
    }
    if (!queue2.empty()) {
        queue2.dequeue(item);
        return (true);
    }
    if (!queue3.empty()) {
        queue3.dequeue(item);
        return (true);
    }
    return (false);
}

public void print() {
     queue1.print();
     queue2.print();
     queue3.print();
}
```

Performance measurement

- Java contains relative high-performance timer function
  - That make it possible to create an easy function execution time measurement system
- Because we run our programs in virtualized multitasking environment, execution time measurement is not very accurate (other users consume also processor time)
  - Sometimes it is possible to reduce errors by making multiple measurements and taking the best

```
import java.io.*;

public class testTiming {
    private static final int MAXN = 1000;

    private static final int N = 10;
    private static final int N0 = 1000;
    private static final int STEP = 1000;

    public static long test(int n) {
        simpleQueue<Character> queue = new simpleQueue();
        Character[]            items = new Character[MAXN];

    for (int i = 0; i < MAXN; i++)
            items[i] = new Character(Character.forDigit(i, 10));

        long tic = System.currentTimeMillis();
        for (int k = 0; k < n; k++) {
            for (int i = 0; i < MAXN; i++)
                queue.enqueue(items[i]);

            for (int i = 0; i < MAXN; i++)
                queue.dequeue();
        }
        long tac = System.currentTimeMillis();

        long elapsedTime = tac - tic;
        return elapsedTime;
    }

    public static void main(String[] args) {
        PrintWriter writer = null;

        try {
            writer = new PrintWriter("results.csv", "UTF-8");

            int n = N0;
            for (int i = 0; i < N; i++) {
                long time = test(n);
                writer.println(n + ";" + time);
                System.out.println("N: " + n + " Time: " + time + " ms");
                n = n + STEP;
            }
        } catch(Exception e) {
            System.out.println("Exception " + e);
        } finally {
            if (writer != null) writer.close();
        }
    }
}
```

Performance measurement

- Demo program on the previous slide produces csv-files
- On the example right, csv-file is generated which can be easily analyzed in Excel
  - This demonstras that the simpleQueue class (array implementation of Queue data structure) has execution time class O(n), i.e. linear dependent to the number of items inserted to the queue