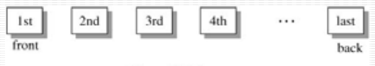
Data Structures for Java
William H. Ford
William R. Topp

Chapter 15
Queues and Priority Queues

Bret Ford

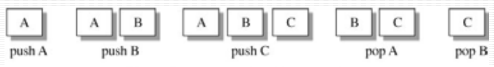
Queue Collection

- A queue is a list of items that allows for access only at the two ends of the sequence, called the front and back of the queue. An item enters at the back and exits from the front.



Queue Operations

- Queue operations are restricted to the ends of the list, called front and back.
  push(item) adds item at the back
  pop() removes element from the front
  peek() accesses value at the front



Queue Operations (2)

- An item removed (pop) from the queue is the first element that was added (push) into the queue. A queue has FIFO (first-in-first-out) ordering.
- Queue inserts followed by queue deletions maintain the order of the elements

Queue Interface

- The generic Queue interface defines a restricted set of collection operations that access and update elements only at the end of the list.

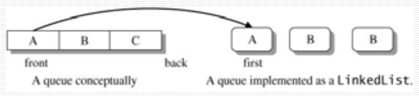
interface Queue<T>	**ds.util**
boolean	**isEmpty**() Returns true if this collection contains no elements and false if the collection has at least 1 element.
T	**peek**() Returns the element at the front of the queue. If empty, throws a NoSuchElementException.

Queue Interface (end)

interface Queue<T>	**ds.util**
T	**pop**() Removes the element at the front of the queue and returns its value. If the queue is empty, throws a NoSuchElementException.
void	**push**(T item) Inserts item at the back of the queue.
int	**size**() Returns the number of elements in this queue

LinkedQueue Class

- The Queue interface is an adapter which defines a restricted set of list methods. A Queue class can be efficiently implemented using a linked list as the storage structure. The LinkedQueue class uses a LinkedList collection and composition.



LinkedQueue Class (2)

```
public class LinkedQueue<T> implements Queue<T>
{
   private LinkedList<T> qlist = null;
   public LinkedQueue ()
   {
      qlist = new LinkedList<T>();
   }
   . . .
}
```

Implementing method pop()

- Method has runtime efficiency O(1)

```
public T pop()
{
   // if the queue is empty, throw
   // NoSuchElementException
   if (isEmpty())
      throw new NoSuchElementException(
            "LinkedQueue pop(): queue empty");

   // remove and return the first element in the list
   return qlist.removeFirst();
}
```

Program 15.1

- The program implements an interview scheduler that is a queue of Time24 objects. Output lists the times and the potential length of each interview.

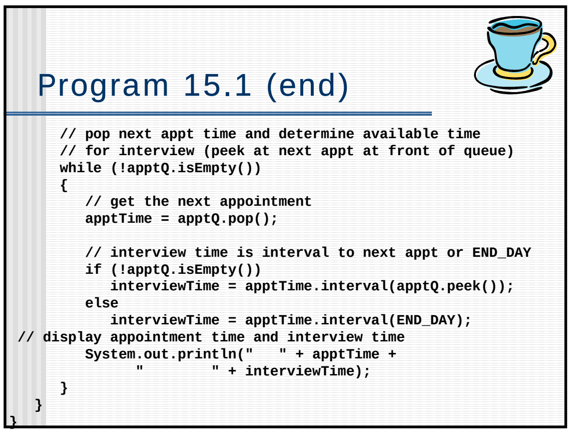
```
import java.io.*;
import java.util.Scanner;
import ds.util.LinkedQueue;
import ds.time.Time24;

public class Program15_1
{
```

Program 15.1 (2)

```
public static void main(String[] args)
   throws IOException
   {
      final Time24 END_DAY = new Time24(17,00);
      String apptStr;

      // time interval from current appt to next appt
      Time24 apptTime = null, interviewTime = null;

      // input stream to read times from file "appt.dat"
      Scanner input = new Scanner(
            new FileReader("appt.dat"));
```

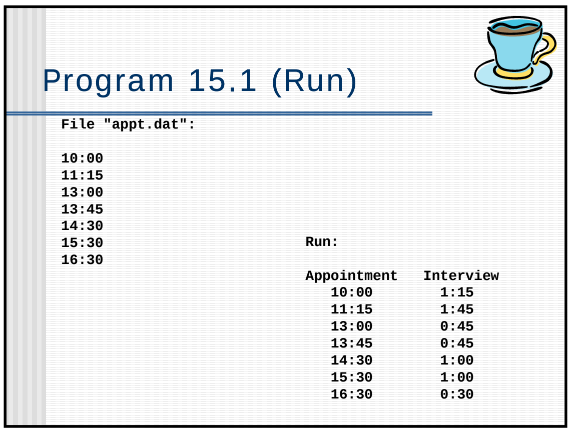
Program 15.1 (3)

```
// queue to hold appointment time for job applicants
LinkedQueue<Time24> apptQ = new
      LinkedQueue<Time24>();

// construct the queue by appt times as
// strings from file; use parseTime to
// convert to Time24 object
while (input.hasNext())
{
   apptStr = input.nextLine();
   apptQ.push(Time24.parseTime(apptStr));
}

// output the day's appointment schedule
System.out.println("Appointment   Interview");
```



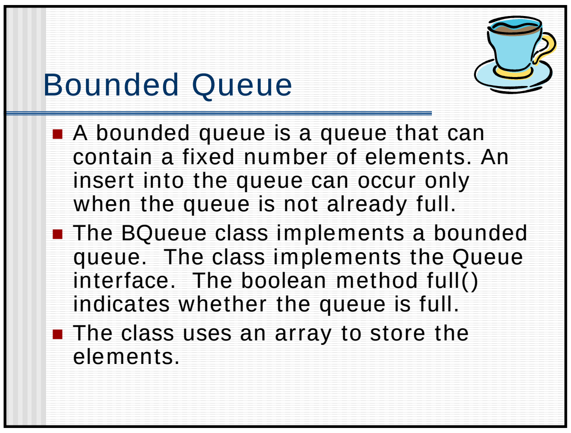
Program 15.1 (end)

```
      // pop next appt time and determine available time
      // for interview (peek at next appt at front of queue)
      while (!apptQ.isEmpty())
      {
         // get the next appointment
         apptTime = apptQ.pop();

         // interview time is interval to next appt or END_DAY
         if (!apptQ.isEmpty())
            interviewTime = apptTime.interval(apptQ.peek());
         else
            interviewTime = apptTime.interval(END_DAY);
 // display appointment time and interview time
         System.out.println("   " + apptTime +
               "         " + interviewTime);
      }
   }
}
```



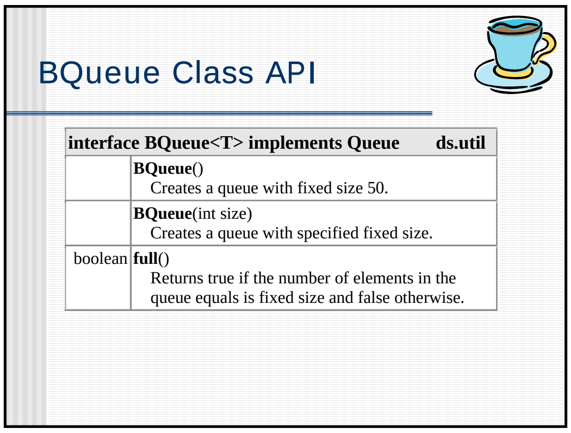
Program 15.1 (Run)

```
File "appt.dat":

10:00
11:15
13:00
13:45
14:30
15:30                        Run:
16:30
                             Appointment   Interview
                                10:00        1:15
                                11:15        1:45
                                13:00        0:45
                                13:45        0:45
                                14:30        1:00
                                15:30        1:00
                                16:30        0:30
```



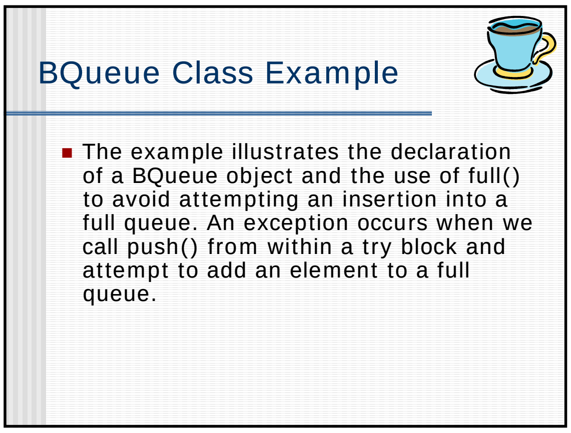
Bounded Queue

- A bounded queue is a queue that can contain a fixed number of elements. An insert into the queue can occur only when the queue is not already full.
- The BQueue class implements a bounded queue. The class implements the Queue interface. The boolean method full() indicates whether the queue is full.
- The class uses an array to store the elements.



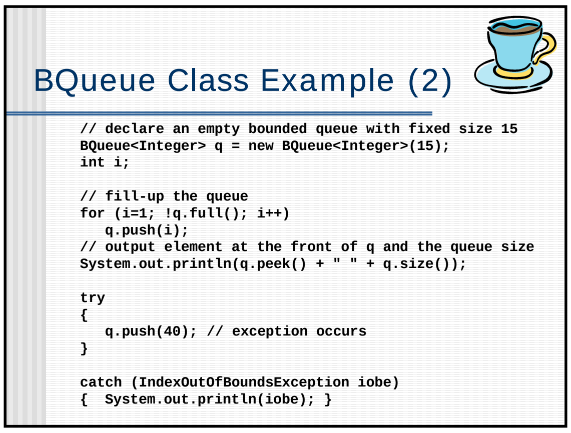
BQueue Class API

interface BQueue<T> implements Queue	**ds.util**
	BQueue() Creates a queue with fixed size 50.
	BQueue(int size) Creates a queue with specified fixed size.
boolean	**full**() Returns true if the number of elements in the queue equals is fixed size and false otherwise.



BQueue Class Example

- The example illustrates the declaration of a BQueue object and the use of full() to avoid attempting an insertion into a full queue. An exception occurs when we call push() from within a try block and attempt to add an element to a full queue.



BQueue Class Example (2)

```
// declare an empty bounded queue with fixed size 15
BQueue<Integer> q = new BQueue<Integer>(15);
int i;

// fill-up the queue
for (i=1; !q.full(); i++)
   q.push(i);
// output element at the front of q and the queue size
System.out.println(q.peek() + " " + q.size());

try
{
   q.push(40); // exception occurs
}

catch (IndexOutOfBoundsException iobe)
{  System.out.println(iobe); }
```

BQueue Class Example (end)

```
Output:
    1 15
    java.lang.IndexOutOfBoundsException: BQueue push(): queue full
```

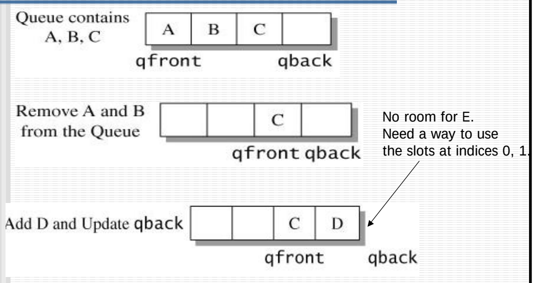
BQueue Class

```
public class BQueue<T> implements Queue<T>
{
   // array holding the queue elements
   private T[] queueArray;
   // index of the front and back of the queue
   private int qfront, qback;
   // the capacity of the queue and the current size
   private int qcapacity, qcount;

   // create an empty bounded queue with specified size
   public BQueue(int size)
   {
      qcapacity = size;
      queueArray = (T[])new Object[qcapacity];
      qfront = 0;
      qback = 0;
      qcount = 0;
   }
```

BQueue Class (end)

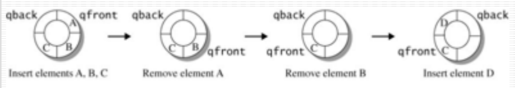
```
   public BQueue()
   {
      // called non-default constructor with capacity = 50
      BQueue(50);
   }

   < method full() and methods in the Queue interface >
}
```

BQueue Class Implementation



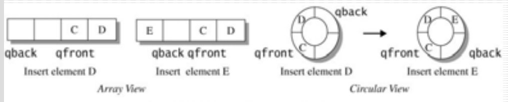
BQueue Implementation (2)

- Think of the queue as a circular sequence with a series of slots that allow element to enter in a clockwise fashion. The element at index qfront exits the queue and an element enters the queue at index qback.



BQueue Implementation (3)

- Treating the array as a circular sequence involves updating qfront and qback to cycle back to the front the array as soon as they move past the end of the array.



```
Move qback forward: qback = (qback + 1) % qcapacity;
Move qfront forward:qfront = (qfront + 1) % qcapacity;
```

BQueue full()

```
public boolean full()
{
   return qcount == qcapacity;
}
```

BQueue push()

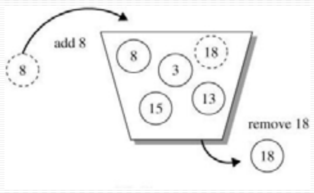
```
public void push(T item)
{
   // is queue full? if so, throw an
   // IndexOutOfBoundsException
   if (qcount == qcapacity)
      throw new IndexOutOfBoundsException(
            "BQueue push(): queue full");

   // insert into the circular queue
   queueArray[qback] = item;
   qback = (qback+1) % qcapacity;

   // increment the queue size
   qcount++;
}
```

BQueue pop()

```
public T pop()
{
   // if queue is empty, throw a NoSuchElementException
   if (count == 0)
      throw new NoSuchElementException(
            "BQueue pop(): empty queue");

   // save the front of the queue
   T queueFront = queueArray[qfront];

   // perform a circular queue deletion
   qfront = (qfront+1) % qcapacity;

   // decrement the queue size
   qcount--;
   // return the front
   return queueFront;
}
```

Priority Queue Collection

- A priority queue is a collection in which all elements have a comparison (priority) ordering.
- It provides only simple access and update operations where a deletion always removes the element of highest priority



PQueue Interface

- The generic PQueue resembles a queue with the same method names.

interface PQueue<T>	ds.util
boolean	**isEmpty**() Returns true if the priority queue is empty and false otherwise.
T	**peek**() Returns the value of the highest-priority item. If empty, throws a NoSuchElementException.

PQueue Interface (end)

interface PQueue<T>	ds.util
T	**pop**() Removes the highest priority item from the queue and returns its value. If it is empty, throws a NoSuchElementException.
void	**push**(T item) Inserts item into the priority queue.
int	**size**() Returns the number of elements in this priority queue

HeapPQueue Class

- The collection class HeapPQueue implements the PQueue interface.
  - By default, the element of highest priority is the one with the largest value (a maximum priority queue); that is, if x and y are two elements in a priority queue and x > y, then x has higher priority than y.

HeapPQueue Class Example

```
// create an empty priority queue of generic type String
HeapPQueue<String> pq = new HeapPQueue<String>();
int n;
pq.push("green");
pq.push("red");
pq.push("blue");
// output the size and element with the highest priority
System.out.println(pq.size() + "  " + pq.peek());
// use pop() to clear the collection and list elements in
// priority (descending) order
while (!pq.isEmpty())
       System.out.print(pq.pop() + "  ");
```

```
Output:
   3  red
   red  green  blue
```

Support Services Pool

- The application processes job requests to a company support service pool. A request has a job ID, a job status, and a time requirement.
- JobStatus is an enum with a listing of employee categories with values that allows for comparison of objects.
- The JobRequest class implements Comparable and describes job objects.

Support Services Pool (2)

```
enum JobStatus
{
       clerk (0), manager (1), director(2),
      president(3);
       int jsValue;
       JobStatus(int value) { jsValue = value; }
       public int value() { return jsValue; }
}
```

Support Services Pool (3)

class JOBREQUEST implements Comparable<JobRequest>	
	Constructors
	JobRequest (JobStatus status, int ID, int time) Creates an object with the specified arguments.
	Methods
int	**getJobID**() Returns the ID for this object.
int	**getJobStatus**() Returns the status for this object.

Support Services Pool (4)

class JOBREQUEST implements Comparable<JobRequest>	
int	**getJobTime**() Returns the time for this object in minutes.
static JobRequest	**readJob**(Scanner sc) Reads a job from the scanner with the form status jobID jobTime; Returns a JobRequest object or null is input is requests beyond end of file.
String	**toString**() Returns a string that represents a job in the format "<status name> <ID> <time>".
int	**compareTo**(JobRequest item) Compare the current object's jobStatus with the jobStatus of item.

Program 15.3

- The program processes job requests with different employee statuses. Output lists the jobs by status along with their total time.

Program 15.3 (2)

```
import java.io.*;
import java.util.Scanner;
import ds.util.HeapPQueue;

public class Program15_3
{
   public static void main(String[] args)
   throws IOException
   {
      // handle job requests
      HeapPQueue<JobRequest> jobPool =
            new HeapPQueue<JobRequest>();

      // job requests are read from file "job.dat"
      Scanner sc = new Scanner(new FileReader(
            "job.dat"));
```

Program 15.3 (3)

```
      // time spent working for each category
      // of employee
      // initial time 0 for each category
      int[] jobServicesUse = {0,0,0,0};
      JobRequest job = null;

      // read file; insert each job into
      // priority queue
      while ((job = JobRequest.readJob(sc)) != null)
         jobPool.push(job);

      // delete jobs from priority queue
      // and output information
      System.out.println("Category     Job ID" +
            "    Job Time");
```

Program 15.3 (4)

```
      while (!jobPool.isEmpty())
      {
         // remove a job from the priority
         // queue and output it
         job = (JobRequest)jobPool.pop();
         System.out.println(job);

         // accumulate job time for the
         // category of employee
         jobServicesUse[job.getStatus().value()] +=
               job.getJobTime();
      }
      System.out.println();

      writeJobSummary(jobServicesUse);
   }
```

Program 15.3 (end)

```
   private static void writeJobSummary(
   int[] jobServicesUse)
   {
      System.out.println("Total Pool Usage");
      System.out.println("   President    " +
            jobServicesUse[3]);
      System.out.println("   Director     " +
            jobServicesUse[2]);
      System.out.println("   Manager      " +
            jobServicesUse[1]);
      System.out.println("   Clerk        " +
            jobServicesUse[0]);
   }
}
```

Program 15.3 (Run)

```
Run

Category     Job ID    Job Time
President     303        25
President     306        50
Director      300        20
Director      307        70
Director      310        60
Director      302        40
Manager       311        30
Manager       304        10
Manager       305        40
Clerk         308        20                      Total Pool Usage
Clerk         309        20                         President    75
Clerk         301        30                         Director     190
                                                    Manager      80
                                                    Clerk        70
```