Chapter 5

Queues

Quiz 2

The following 5 questions apply to the Stack abstract data type we have been reviewing in class and in the assignments.

1. What is the function of the top method?

Quiz 2

2. Both linked lists and arrays can be used to implement stacks. When might you consider using a linked-list based stack instead of an array based stack?

Provide a short explanation…

Quiz 2

3. In order to look at the bottom element of a stack of 10 elements, how many times must the pop() method be called?

Quiz 2

4. LIFO is an acronym, what does it mean?

Quiz 2

5. Using what you have learned about the stack, coding in general, and what you remember from the last 3 lectures, what will happen when you run following code segment and why.

```
public static void main(String[] args)
{
        BoundedStackInterface<Integer> myStack;
        myStack = new ArrayStack<Integer>(5);
        myStack.push(1);
        myStack.push(2);
        myStack.push(3);
        myStack.push(4);
        myStack.push(5);
        myStack.push(5);
        System.out.println(myStack.top());
}
```

HW_6_Q1

Create a recursive factorial program that prompts the used for an integer N and writes out a series of equations representing the calculation of N!

```
import java.util.Scanner;

public class Exercise1
{
  static private String currString = " = ";

  private static int factorial(int n)
  {
    if (n == 0)
    {
      currString = currString + "0!" ;
      System.out.println(currString);
      return 1;
    }
    else
    {
      currString = currString + n;
      System.out.println(currString + "!");
      currString = currString + " * ";
      return (n * factorial (n - 1));
    }
  }

  public static void main(String[] args)
  {
    Scanner conIn = new Scanner(System.in);
    int num, result;

    System.out.print("Enter integer> ");
    num = conIn.nextInt();

    System.out.print(num + "!");
    result = factorial(num);
  }
}
```

HW_6_Q2

Use the following three mathematical functions (assume $N \geq 0$):

$Sum(N) = 1 + 2 + 3 + \dots + N$

$BiPower(N) = 2^N$

$TimesFive(N) = 5N$

Define recursively: a) Sum(N), b) BiPower(N), and c) TimesFive(N)

a. Sum(N) = 0 if N = 0

N + Sum (N – 1) if N > 0

b. BiPower(N) = 1 if N = 0

2 * BiPower (N – 1) if N > 0

c. TimesFive(N) =0 if N = 0

5 + TimesFive (N – 1) if N > 0

HW_6_Q3_a,b

3a

```
import java.util.Scanner;
public class Exercise3a
{
  // Precondition: n >= 0
  // Returns the sum of 1 + 2 + ... + n
  private static int sum(int n)
  {
    if (n == 0)
       return 0;
    else
      return (n + sum(n - 1));
  }

  public static void main(String[] args)
  {
    Scanner conIn = new Scanner(System.in);
    int num;

    System.out.print("Enter an integer: ");
    num = conIn.nextInt();

    System.out.println(sum(num));
  }
}
```

3b

```
import java.util.Scanner;

public class Exercise3b
{
 // Precondition: n >= 0
  // Returns 2 to the power n
  private static int biPower(int n)
  {
    if (n == 0)
       return 1;
    else
      return (2 * biPower(n - 1));
  }

  public static void main(String[] args)
  {
    Scanner conIn = new Scanner(System.in);
    int num;

    System.out.print("Enter an integer: ");
    num = conIn.nextInt();

    System.out.println(biPower(num));
  }
}
```

HW_6_Q3_c

3c

```
import java.util.Scanner;
public class Exercise3c
{
  // Precondition: n >= 0
  // Returns the 5 * n
  private static int timesFive(int n)
  {
    if (n == 0)
       return 0;
    else
      return (5 + timesFive(n - 1));
  }

  public static void main(String[] args)
  {
    Scanner conIn = new Scanner(System.in);
    int num;

    System.out.print("Enter an integer: ");
    num = conIn.nextInt();

    System.out.println(timesFive(num));
  }
}
```

HW6_Q6

a) Identify the base case(s) of the puzzle method
b) Identify the general case(s) of the puzzle method
c)Identify the constraints on the arguments passed to the puzzle method

```
int puzzle(int base, int limit)
{
   if (base > limit)
           return -1;
   else
           if (base == limit)
                   return 1;
           else
                   return base * puzzle(base + 1, limit);
}
```

a. Base cases:
    1. when base > limit, return -1
    2. when base == limit, return 1.
b. The general case is when base < limit, return base * puzzle(base + 1, limit).
c. There are no constraints based strictly on the definition of the method, although there may be combinations of values that cause integer overflow during the computation.

HW6_Q7

Show what would be written by the following calls to the recursive method puzzle.

System.out.println(puzzle (14, 10));

a. -1

System.out.println(puzzle (4, 7));

b. 120

System.out.println(puzzle (0, 0));

c. 1

Static Variables

- Variable belonging to the class, not the object(instance)
- Initialized once, at the start of the execution .
  - Before the initialization of any instance variables
- A single copy to be shared by all instances of the class
- Can be accessed directly by the class name
  - I.e. it doesn’t need any object
- Syntax : <class-name>.<variable-name>

Static Methods

- Static methods can access only static data.
  - They can not access non-static data (instance variables)
- A static method can call only other static methods
  - This means that you can not call a non-static method from it
- A static method can be accessed directly by the class name
  - It doesn’t need any object
- Static methods cannot refer to "this" or "super" keywords
- Syntax : <class-name>.<method-name>
- The main method is static
  - it must be accessible for an application to run , before any instantiation takes place

Lab1

- Identify and comment errors in Lab1.java

Lab2 – OOP basics
Calling methods from other files

1. Using Lab2.java, create a new Java class file called recur.java with Jcreator and…

2. Move the sum method to this new class

3. In the main function, instead of calling sum(num), you must now invoke the method factorial from the class recur.

Chapter 5: The Queue ADT

5.1 – Queues

5.2 – Formal Specification

5.3 – Array-Based Implementations

5.4 – Application: Palindromes

5.5 – Application: The Card Game of War

5.6 – Link-Based Implementations

5.1 Queues

- **Queue** A structure in which elements are added to the rear and removed from the front.
  - a “first in, first out” (FIFO) structure

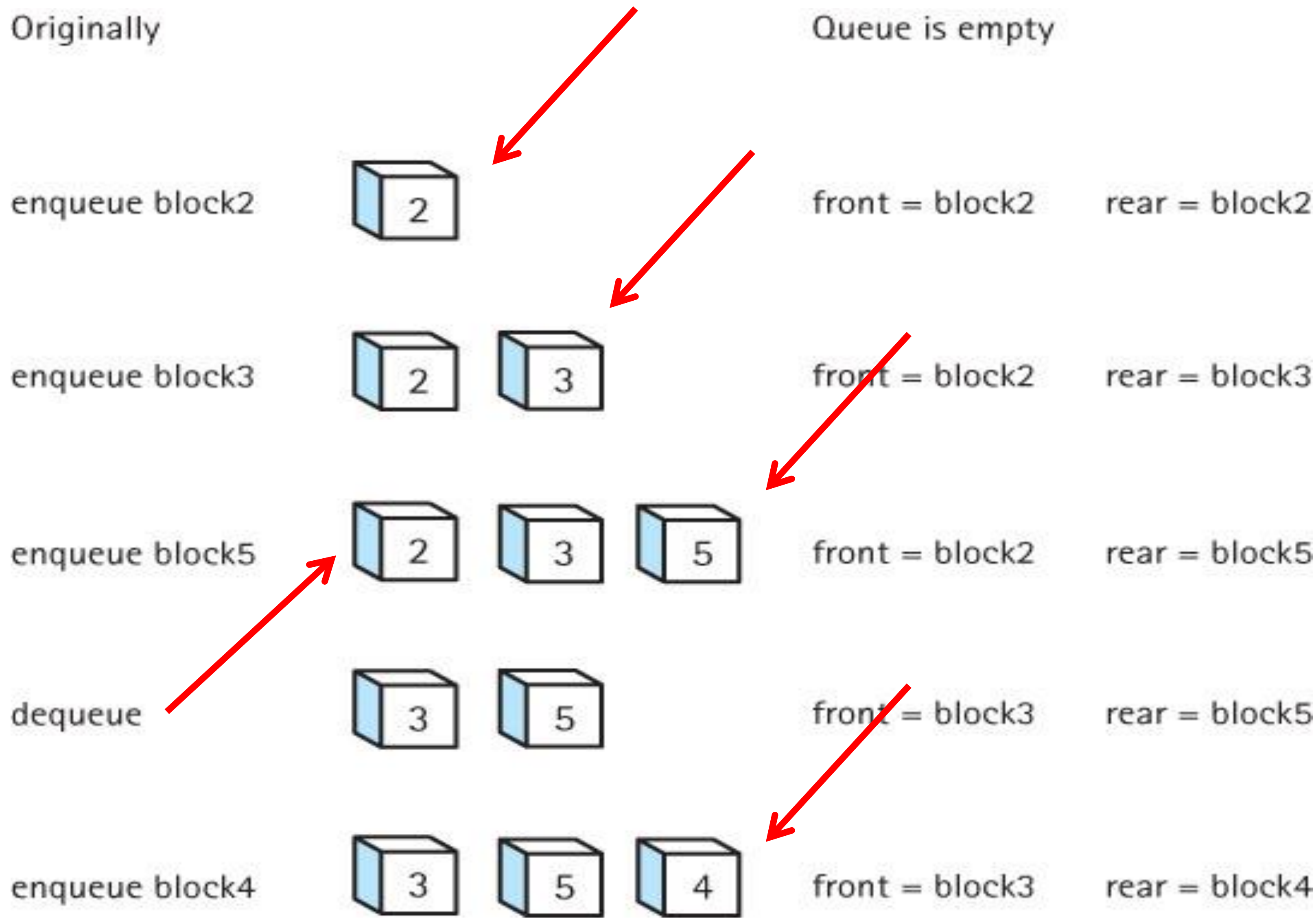
How are queues used?

- Operating systems often maintain a queue of processes
  - Ready to execute
  - Waiting for a particular event to occur
- “Holding area” is needed to manage messages between two processes, two programs, or even two systems
  - Called the buffer
  - The buffer can be implemented as a queue
- Software queue vs. Real world queue
  - Real world queues
    - Line at a cashier or at rides at Disneyland
  - Use the queue data structure simulate and analyze real world queues

Operations on Queues

- Constructor
  - new - creates an empty queue
- Transformers
  - enqueue - adds an element to the rear of a queue
  - dequeue - removes and returns the front element of the queue

Effects of Queue Operations (enqueue and dequeue)



5.2 Formal Specification (of our Queue ADT)

- Queue is FIFO
  - enqueue
    - adds an element to the rear of a queue
  - dequeue
    - removes and returns the front element

- What are the differences between the queue and the stack in terms of their methods?

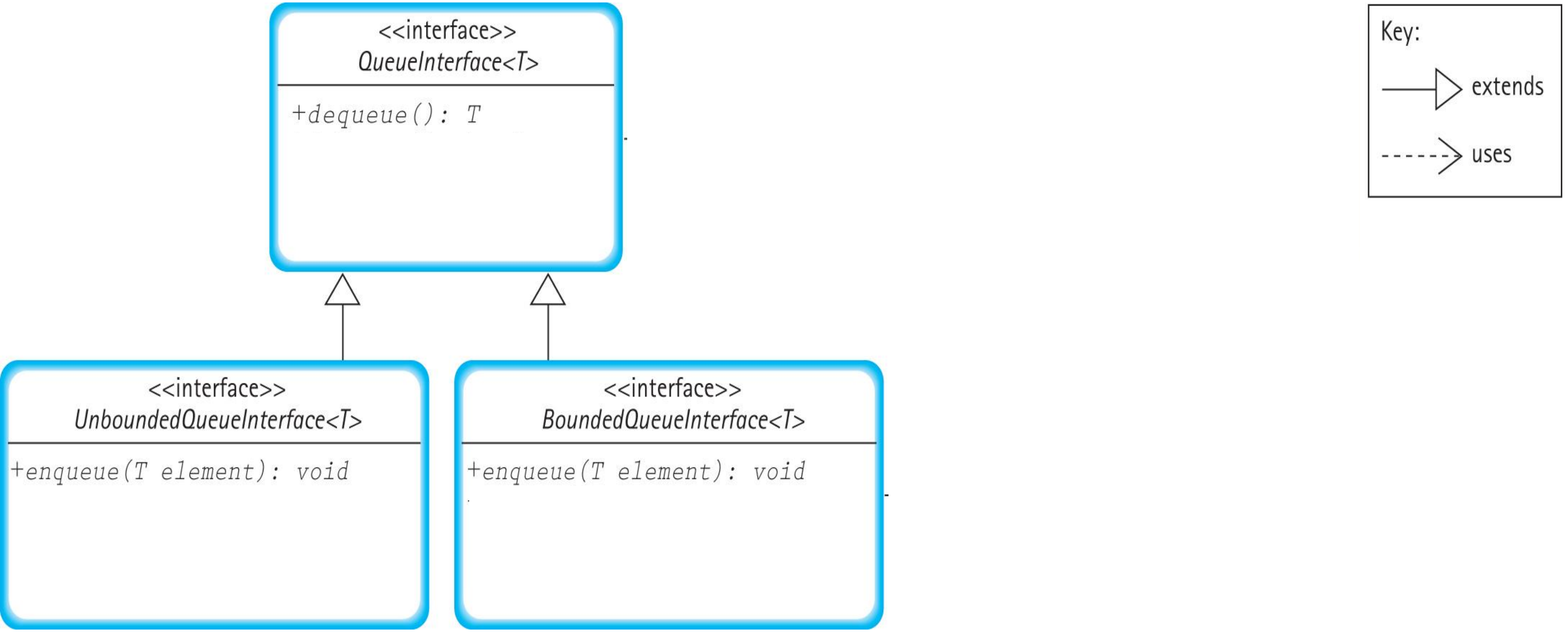
5.2 Formal Specification (of our Queue ADT)

1. enqueue method
2. dequeue method
3. constructor to create an empty queue
4. The Queue ADT should be generic
   – Allow the class of objects held by the queue to be specified upon instantiation of the queue
     - The class of objects could be a primitive data type.

Formal Specification for the QUEUE ADT
(same as we did for Stacks)

- Identify & address any exceptional situations
- Determine boundedness
  - Support two versions of the Queue ADT
    - Bounded & unbounded
- Define the Queue interface or interfaces
- Define the interfaces
  - `QueueInterface`:
    - features of a queue not affected by boundedness
  - `BoundedQueueInterface`:
    - features specific to a bounded queue
  - `UnboundedQueueInterface`:
    - features specific to an unbounded queue

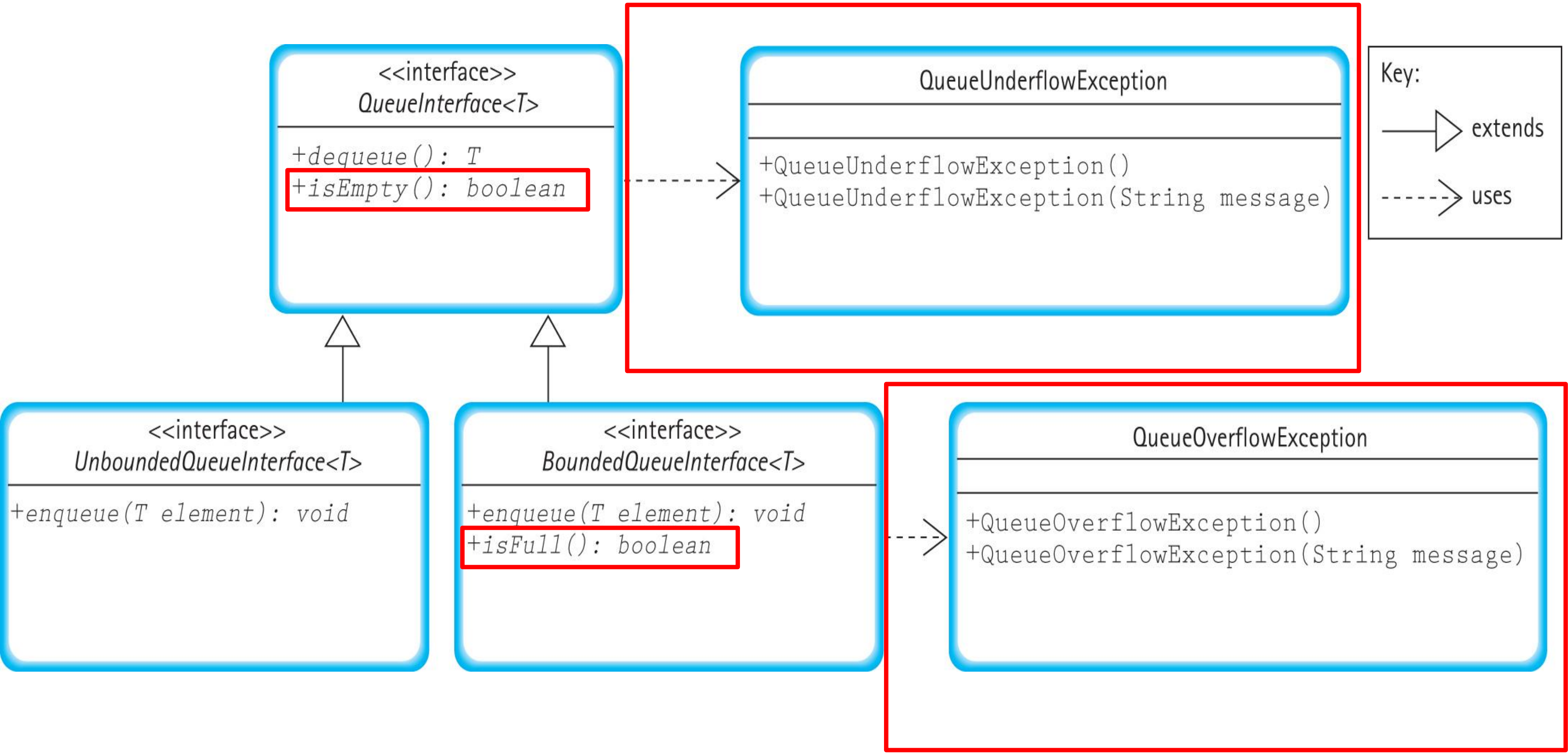
Relationships among Queue Interfaces



Exceptional Situations

- dequeue – what if the queue is empty?
  - throw a `QueueUnderflowException`
  - plus define an `isEmpty` method for use by the application
- enqueue – what if the queue is full?
  - throw a `QueueOverflowException`
  - plus define an `isFull` method for use by the application

Relationships among Queue Interfaces



QueueInterface

```
//----------------------------------------------------------------------------
// QueueInterface.java          by Dale/Joyce/Weems                 Chapter 5
//
// Interface for a class that implements a queue of Objects.
// A queue is a first-in, first-out structure.
//----------------------------------------------------------------------------

package ch05.queues;

public interface QueueInterface<T>
{
  // Throws QueueUnderflowException if this queue is empty,
  // otherwise removes front element from this queue and returns it.
  T dequeue() throws QueueUnderflowException;

  // Returns true if this queue is empty, otherwise returns false.
  boolean isEmpty();
}
```

The Remaining Queue Interfaces

The BoundedQueueInterface

```
package ch05.queues;

public interface BoundedQueueInterface<T> extends QueueInterface<T>
{
  // Throws QueueOverflowException if this queue is full,
  // otherwise adds element to the rear of this queue.
  void enqueue(T element) throws QueueOverflowException;

  // Returns true if this queue is full, otherwise returns false.
  boolean isFull();
}
```

The UnboundedQueueInterface

```
package ch05.queues;

public interface UnboundedQueueInterface<T> extends QueueInterface<T>
{
  // Adds element to the rear of this queue.
  void enqueue(T element);
}
```

5.3 Array-Based Implementations

- 2 array-based implementations of the Queue
  - bounded
  - unbounded

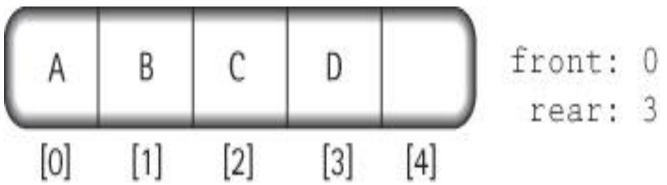
- Begin by answering the question,

  "How we will hold the queue elements in the array?"

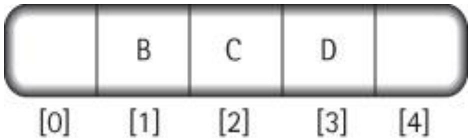
Fixed Front Design

(with an array)

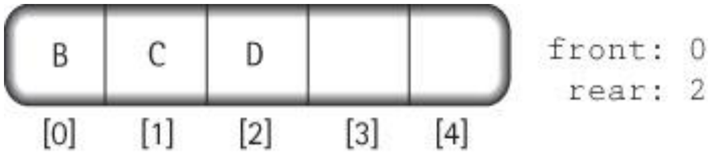
- Call enqueue() 4 times with
  - 'A', 'B', 'C', and 'D':



- Dequeue the front element:



- Move every element in the queue up one slot



We will not use it!!! Why?

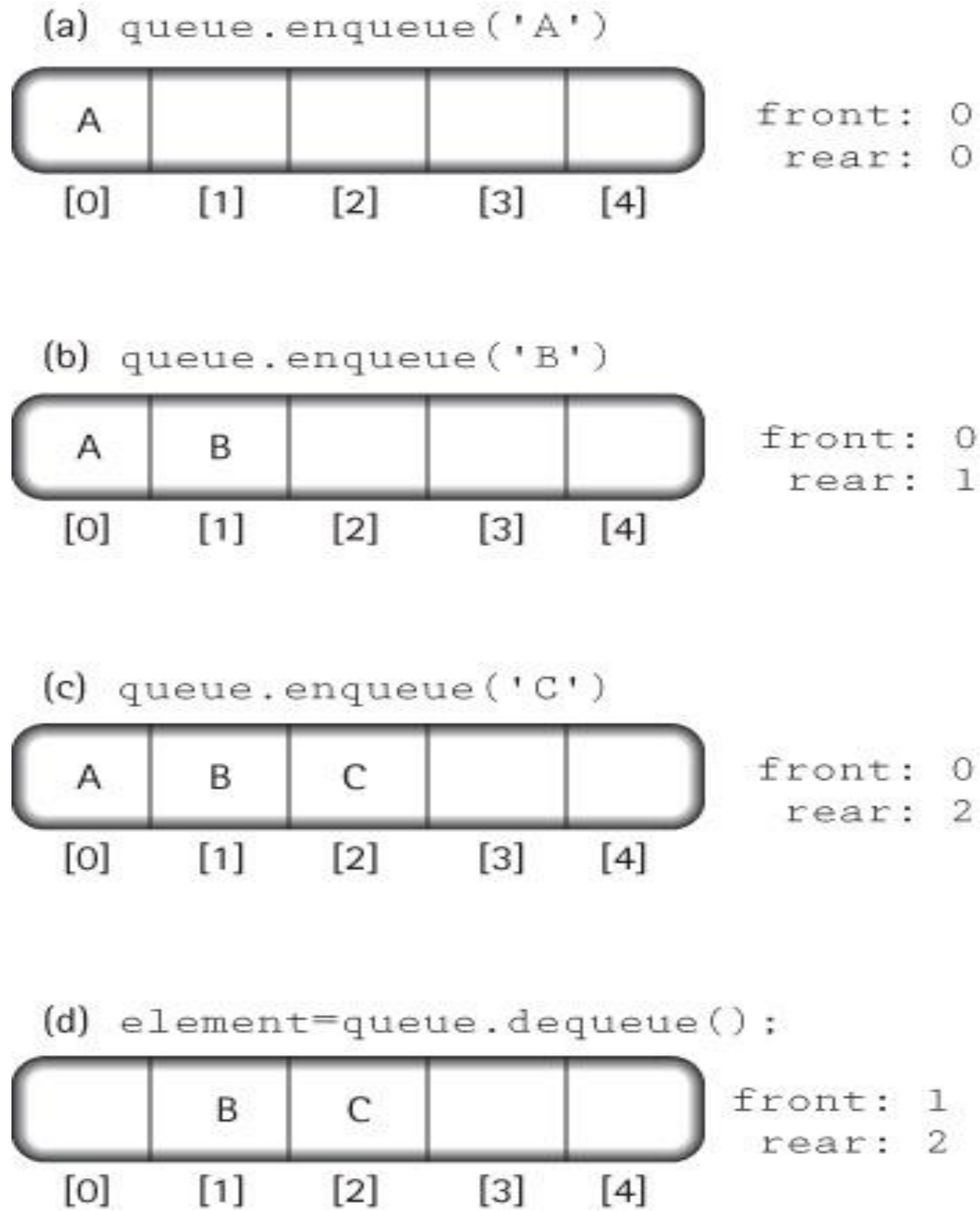
This implementation of dequeue is inefficient!

Floating Front Design

We use this approach!

Is this confusing?

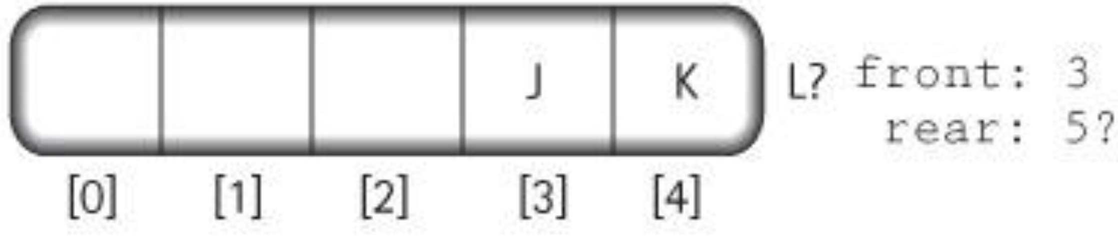
How might it work?



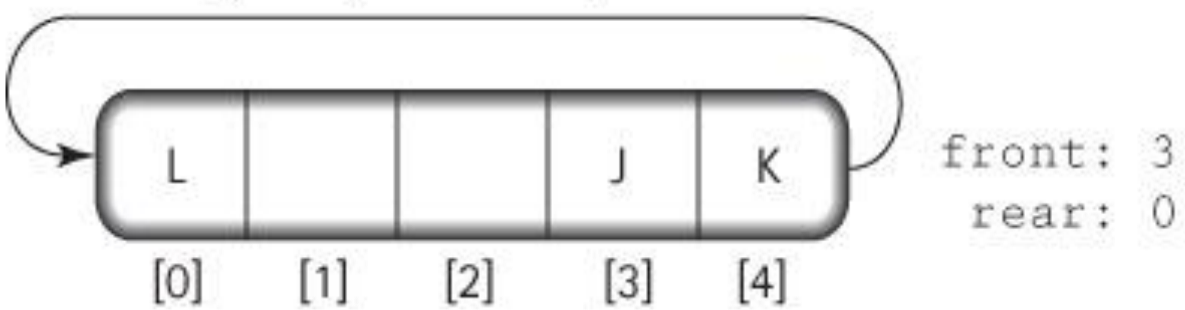
And the solution is…

Wrap Around with Floating Front Design

(a) There is no room at the end of the array



(b) Using the array as a circular structure, we can wrap the queue around to the beginning of the array



The `ArrayBndQueue` Class

```
package ch05.queues;

public class ArrayBndQueue<T> implements BoundedQueueInterface<T>
{
  protected final int DEFCAP = 100; // default capacity
  protected T[] queue;              // array that holds queue elements
  protected int numElements = 0;    // number of elements in the queue
  protected int front = 0;          // index of front of queue
  protected int rear;               // index of rear of queue

  public ArrayBndQueue()
  {
    queue = (T[]) new Object[DEFCAP];
    rear =  DEFCAP - 1;
  }

  public ArrayBndQueue(int maxSize)
  {
    queue = (T[]) new Object[maxSize];
    rear =  maxSize - 1;
  }
```

`ArrayBndQueue` Class
The enqueue operation

```
public void enqueue(T element)
// Throws QueueOverflowException if this queue is full,
// otherwise adds element to the rear of this queue.
{
  if (isFull())
    throw new QueueOverflowException("Enqueue attempted on a full queue.");
  else
  {
    rear = (rear + 1) % queue.length;
    queue[rear] = element;
    numElements = numElements + 1;
  }
}
```

`ArrayBndQueue` Class
The dequeue operation

```
public T dequeue()
// Throws QueueUnderflowException if this queue is empty,
// otherwise removes front element from this queue and returns it.
{
  if (isEmpty())
    throw new QueueUnderflowException("Dequeue attempted on empty queue.");
  else
  {
    T toReturn = queue[front];
    queue[front] = null;
    front = (front + 1) % queue.length;
    numElements = numElements - 1;
    return toReturn;
  }
}
```

`ArrayBndQueue` Class Remaining Queue Operations

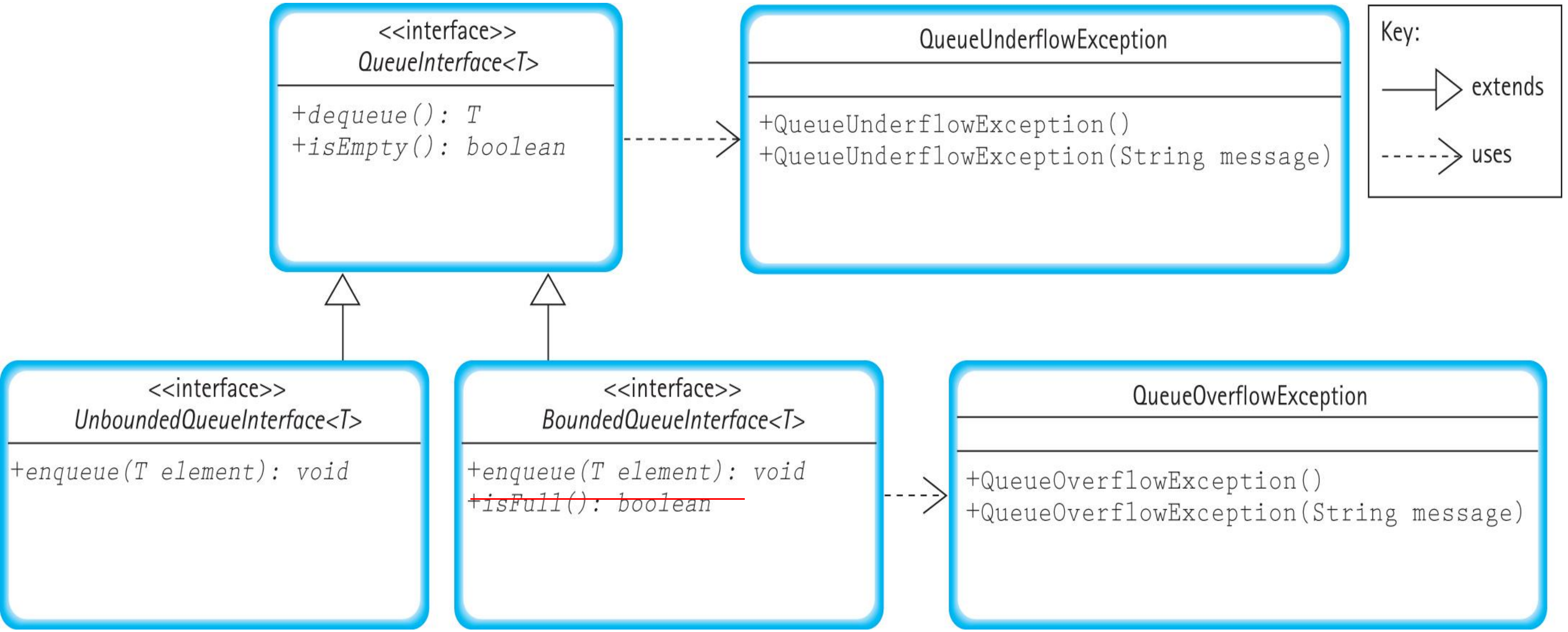
```
public boolean isEmpty()
// Returns true if this queue is empty, otherwise returns false
{
  return (numElements == 0);
}

public boolean isFull()
// Returns true if this queue is full, otherwise returns false.
{
  return (numElements == queue.length);
}
```

Relationships among Queue Interfaces

If we were to make an

***ArrayUnbndQueue** Class*

what method would not be needed?



What else would be needed?

A dynamically expanding array!

The `ArrayUnbndQueue` Class

- The trick:
  1. Create a new / larger array
  2. Copy the queue into the new array

  - Enlarging the array is not the same as enqueing
    - Implement a separate `enlarge` method
    - Instantiate a new array with a size equal to the current capacity plus the original capacity
    - Copy contents from the smaller array into the larger one
    - `isFull` is no longer required by the Unbounded Queue Interface
      - Drop isFull from the class

  - `dequeue` and `isEmpty` methods are unchanged

The `ArrayUnbndQueue` Class

```
package ch05.queues;

public class ArrayUnbndQueue<T> implements UnboundedQueueInterface<T>
{
  protected final int DEFCAP = 100; // default capacity
  protected T[] queue;              // array that holds queue elements
  protected int origCap;            // original capacity
  protected int numElements = 0;    // number of elements n the queue
  protected int front = 0;          // index of front of queue
  protected int rear = -1;          // index of rear of queue

  public ArrayUnbndQueue()
  {
    queue = (T[]) new Object[DEFCAP];
    rear = DEFCAP - 1;
    origCap = DEFCAP;
  }

  public ArrayUnbndQueue(int origCap)
  {
    queue = (T[]) new Object[origCap];
    rear = origCap - 1;
    this.origCap = origCap;
  }
```

Why do we use this.?

The `ArrayUnbndQueue` Class
The enlarge operation

```
private void enlarge()
// Increments the capacity of the queue by an amount
// equal to the original capacity.
{
  // create the larger array
  T[] larger = (T[]) new Object[queue.length + origCap];

  // copy the contents from the smaller array into the larger array
  int currSmaller = front;
  for (int currLarger = 0; currLarger < numElements; currLarger++)
  {
    larger[currLarger] = queue[currSmaller];
    currSmaller = (currSmaller + 1) % queue.length;
  }

  // update instance variables
  queue = larger;
  front = 0;
  rear = numElements - 1;
}
```

The `ArrayUnbndQueue` Class
The enqueue operation

```
public void enqueue(T element)
// Adds element to the rear of this queue.
{
  if (numElements == queue.length)
    enlarge();
  rear = (rear + 1) % queue.length;
  queue[rear] = element;
  numElements = numElements + 1;
}
```

5.4 Application: Palindromes

- Examples
  - A tribute to Teddy Roosevelt, who orchestrated the creation of the Panama Canal:
    - A man, a plan, a canal—Panama!
  - Allegedly muttered by Napoleon Bonaparte upon his exile to the island of Elba:
    - Able was I ere, I saw Elba.
  - Or a more well known and easier palindrome
    - wet stew
- Our goal is to write a program that identifies Palindromic strings
  - we ignore blanks, punctuation and the case of letters

The `Balanced` Class

- Create a class called `Palindrome`
  - has a single exported static method `test`
- `test`
  - Input: a candidate string argument
  - Output: returns a boolean value indicating whether the string is a palindrome
- `test` is static
  - invoke it using the name of the class rather than instantiating an Object of the class
- `test` uses both the stack and queue data structures

The `test` method approach

- The `test` method creates a stack and a queue
- It then repeatedly
  - pushes each input letter onto the stack
  - and also enqueues the letter onto the queue
- It discards any non-letter characters
- To simplify the comparison
  - push and enqueue only lowercase versions of the characters
- Once the chars of the candidate string have been processed `test` repeatedly
  - pops a letter from the stack
  - and dequeues a letter from the queue
- As long as there are no mismatches for each accepted character, we have a palindrome and `test` returns true, otherwise `test` returns false

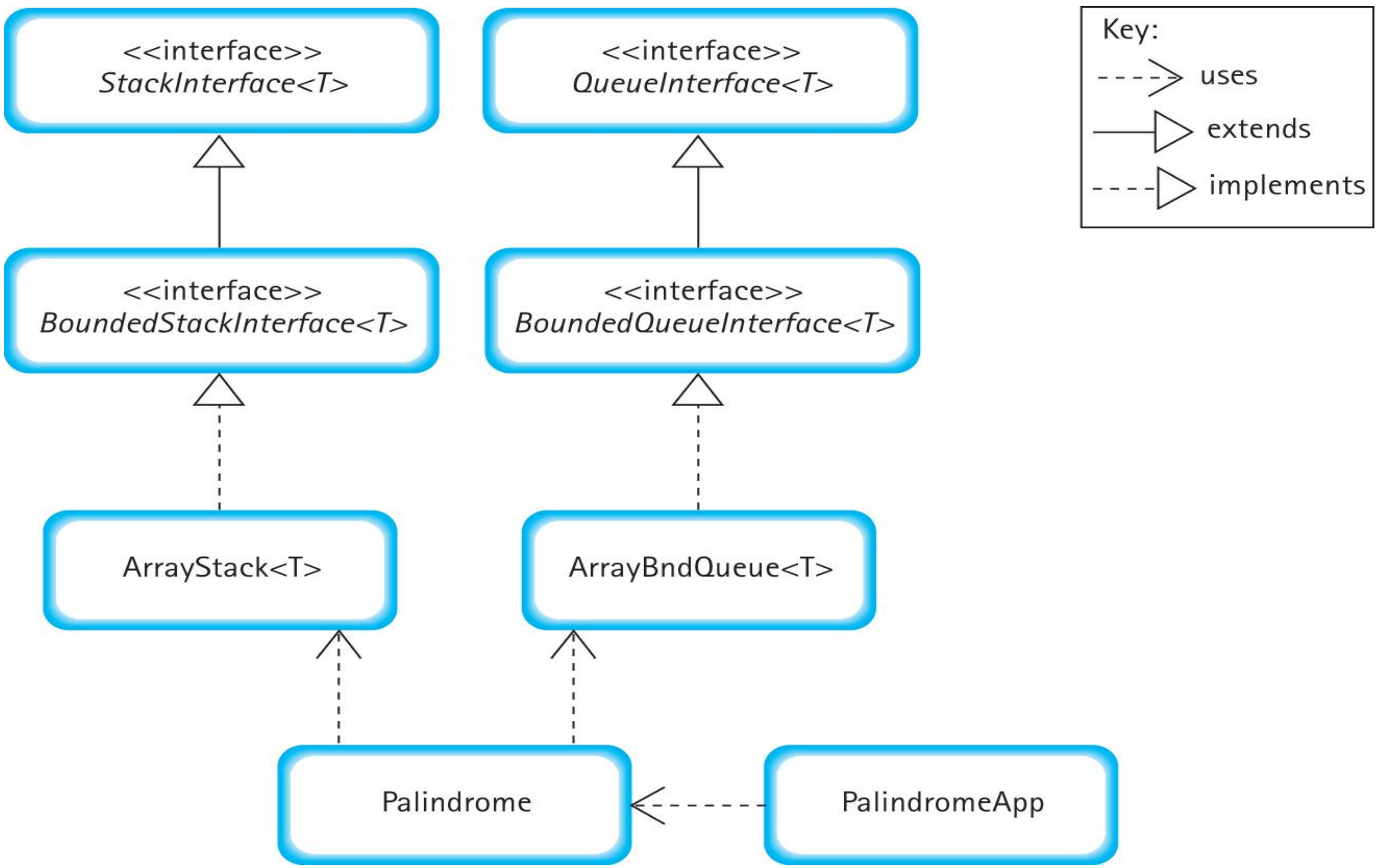
Test for Palindrome (String candidate) the psuedocode

```
Create a new stack
Create a new queue

for each character in candidate
  if the character is a letter
    Change the character to lowercase
    Push the character onto the stack
    Enqueue the character onto the queue

Set stillPalindrome to true

while (there are still more characters in the structures
       && stillPalindrome)
  Pop character1 from the stack
  Dequeue character2 from the queue
  if (character1 != character2)
    Set stillPalindrome to false

return (stillPalindrome)
```

Code and Demo

- Code walkthrough
  - Palindrome.java
  - PalindromeApp.java
- Demo
- SimplePalindrome

Program Architecture



5.5 Application: The Card Game of War

- A deck of cards is shuffled and dealt to two players
- The players repeatedly battle with their cards
  - A battle consists of each player placing the top card from their hand, face up, on the table.
  - Whoever has the higher of the two cards wins the cards
  - A tied battle means war!
- In a war
  - Each player adds three more cards to the prize pile, and then turns up another battle card.
  - Whoever wins this battle wins the entire prize pile
  - If that battle is also a tie then there is another war …
- The card game continues until one of the players runs out of cards, either during a regular battle or during a war. That player loses.

Our War Program

- Simulates several games of war and tells us, on average, how many battles are waged in each game
- Models player's hands and the prize pile as queues
- Simulates the game by coordinating the dequeing and enqueing operations among the two hands and the prize pile according to the rules
- Requires two input values - the number of games to simulate and the maximum number of battles allowed before a game is discontinued
- Output from the program consists of
  - the number of discontinued games
  - the number of completed games
  - the average number of battles waged in completed games

The `RankCardDeck` Class

- Create a class called `RankCardDeck`
  - Models a deck of cards
- Since we are only interested in card “ranks”
  - Do not model suits
- Three methods are exported
  - `shuffle` – randomizes card order
  - `hasMoreCards` – returns a boolean
  - `nextCard` – returns an int

The `WarGame` Class

- Create another class called `WarGame`
  - Simulates a game of War
- The constructor requires an argument indicating the maximum number of battles to allow before discontinuing the game
- The class exports two methods
  - `play` simulates a game until it is finished or discontinued
    - Returns true if the game finished normally
    - Returns false if the game was discontinued
  - `getNumBattles`
    - Returns the number of battles waged in the most recent game

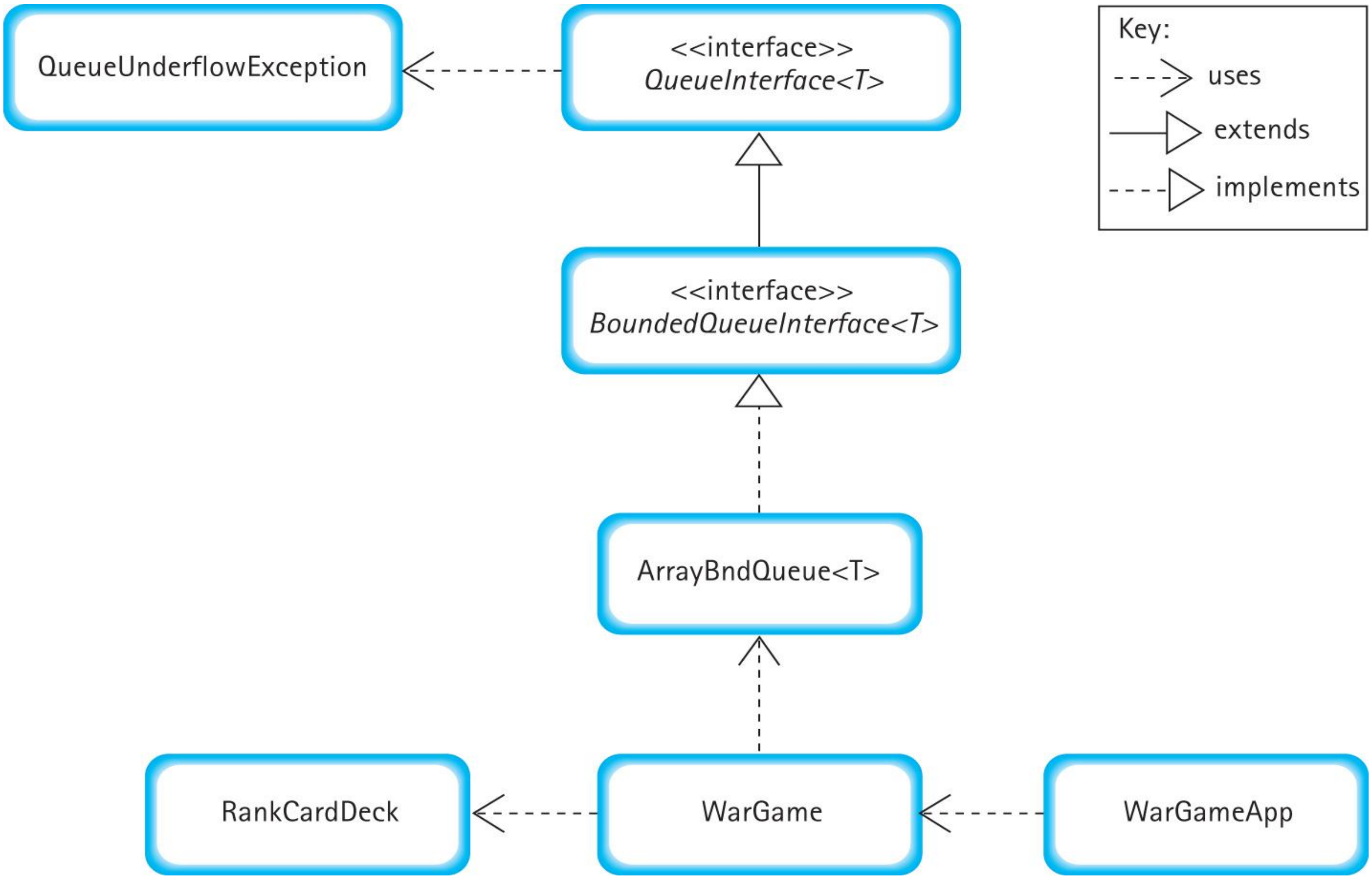
The `play` method

- Creates three queues
  - player1’s hand
  - player2’s hand
  - the prize pile of cards
- Shuffles the deck of cards and "deals" them
- Repeatedly calls a `battle` method
  - Enacting the battle between the two players
  - Continues until the game is over or discontinued
- the `battle` method is recursive – do you see why?

The `battle` method

```
battle()
Get player1's card from player1's hand
Put player1's card in the prize pile
Get player2's card from player2's hand
Put player2's card in the prize pile
if (player1's card > player2's card)
{
  remove all the cards from the prize pile
    and put them in player1's hand
}
else
  if (player2's card > player1's card)
  {
    remove all the cards from the prize pile
      and put them in player2's hand
  }
  else   // war!
  {
    each player puts three cards in the prize pile
    battle()
  }
```

Program Architecture



Code and Demo and Lab

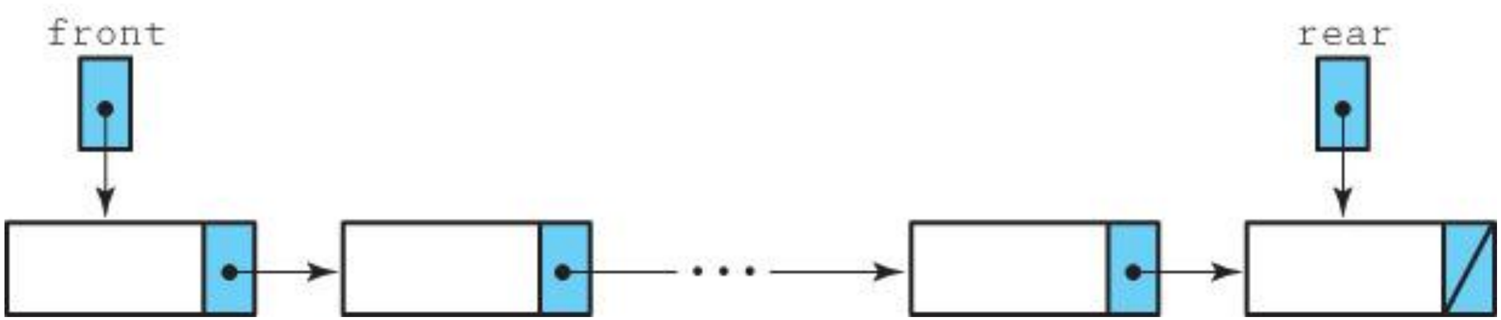
- Can the game end in a tie?
  - Yes
  - Example deck: K 2 4 6
    - Player1 gets: K 4
    - Player2 gets: 2 6
  - Too many ties / incomplete games!!!
- How to avoid ties in real life
  - Shuffle the hands of player1 and player2 after each card has been played
- Lab: for 3 points extra credit implement a queueShuffle method in WarGame.java.
  - You might accomplish this by implementing a queueShuffle() method after a fixed amount of battles in the play() method

5.6 Linked-Based Implementations

- Link-based implementations of the Unbounded Queue ADT
  - discuss a second link-based approach
- Use the same `LLNode` class we used for the linked implementation of stacks.
- Compare queue implementation approaches.

The LinkedUnbndQueue Class

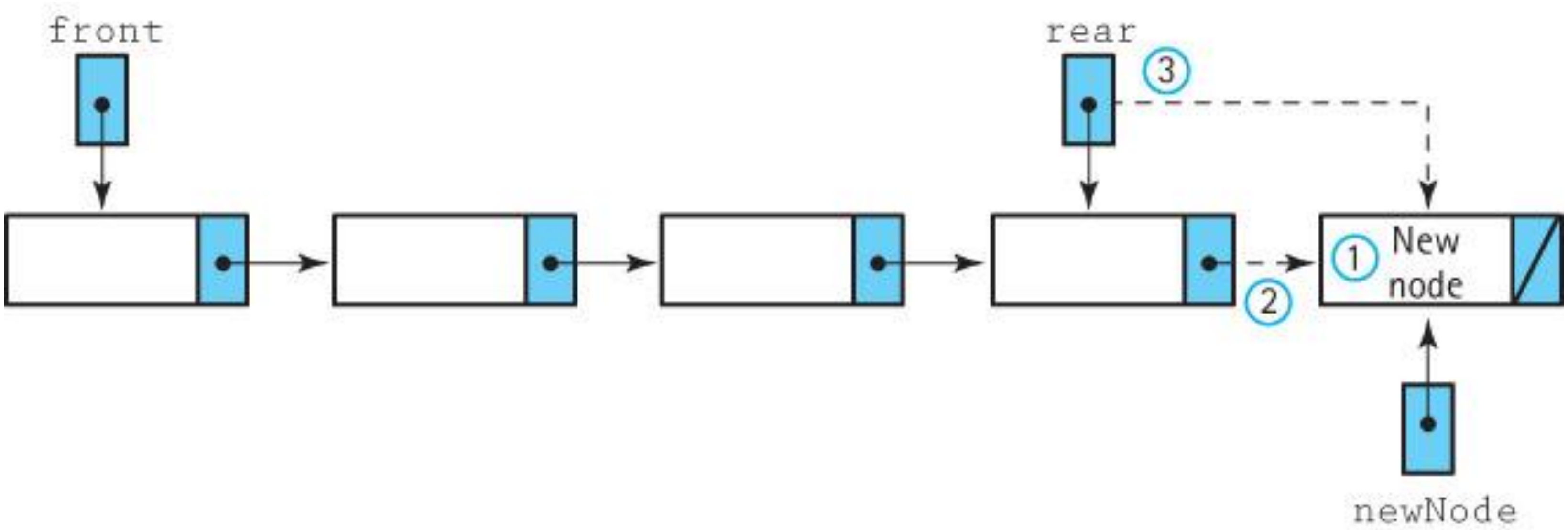
```
package ch05.queues;

import support.LLNode;

public class LinkedUnbndQueue<T> implements UnboundedQueueInterface<T>
{
  protected LLNode<T> front;   // reference to the front of this queue
  protected LLNode<T> rear;    // reference to the rear of this queue
  public LinkedUnbndQueue()
  {
    front = null;
    rear = null;
  }
. . .
```



The enqueue operation

Enqueue (element)

```
1. Create a node for the new element
2. Insert the new node at the rear of the queue
3. Update the reference to the rear of the queue
```



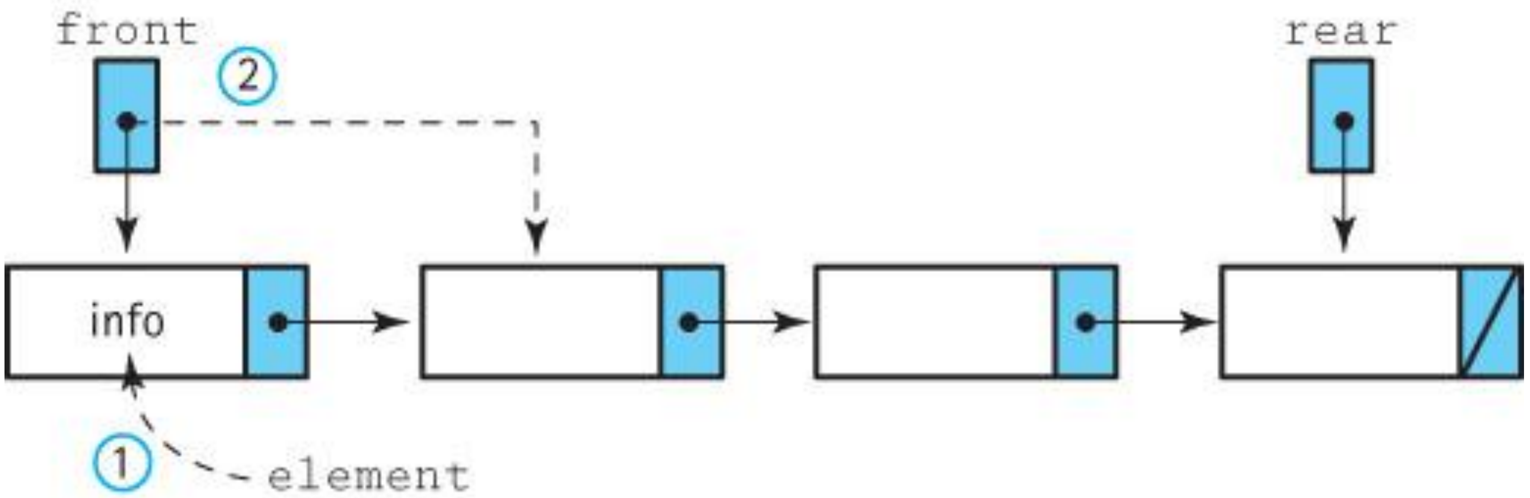
Code for the enqueue method

```
public void enqueue(T element)
// Adds element to the rear of this queue.
{
  LLNode<T> newNode = new LLNode<T>(element);
  if (rear == null)
    front = newNode;
  else
    rear.setLink(newNode);
  rear = newNode;
}
```

The dequeue operation

`Dequeue: returns Object`

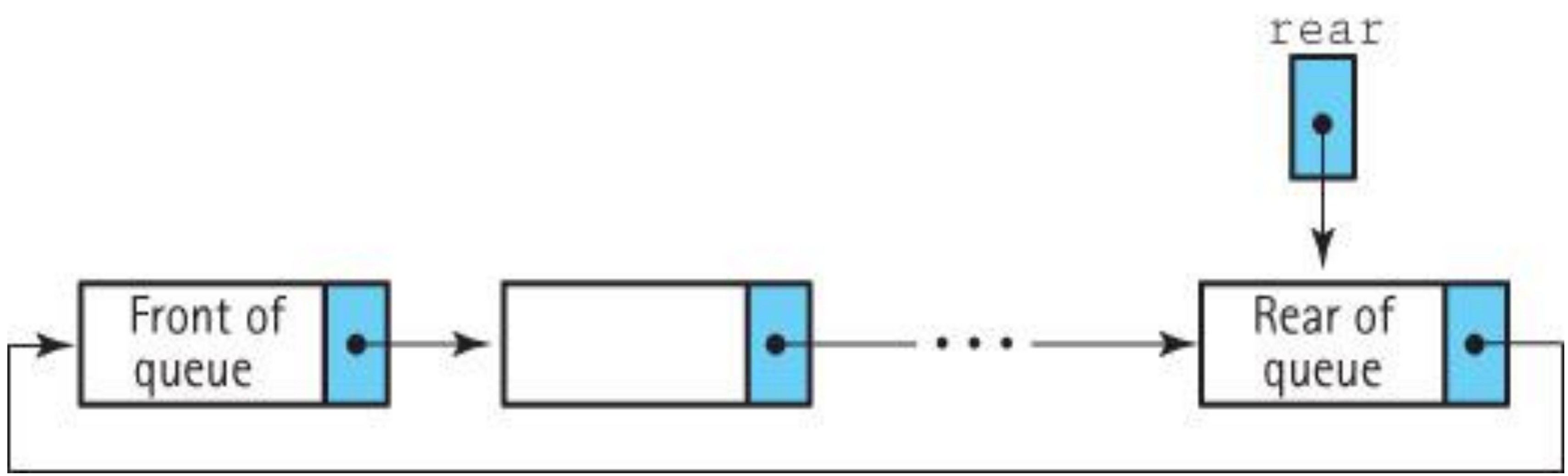
```
1. Set element to the information in the front node
2. Remove the front node from the queue
   if the queue is empty
      Set the rear to null
   return element
```



Code for the dequeue method

```
// Throws QueueUnderflowException if this queue is empty,
// otherwise removes front element from this queue and returns it.
public T dequeue()
{
  if (isEmpty())
    throw new QueueUnderflowException("Dequeue attempted on empty queue.");
  else
  {
    T element;
    element = front.getInfo();
    front = front.getLink();
    if (front == null)
      rear = null;
    return element;
  }
}
```

An Alternative Approach - A Circular Linked Queue



Comparing Queue Implementations

- Storage Size
  - Array-based: takes the same amount of memory, no matter how many array slots are actually used, proportional to current capacity
  - Link-based: takes space proportional to actual size of the queue (but each element requires more space than with array approach)
- Operation efficiency
  - All operations, for each approach, are O(1)
  - Except for the Constructors:
    - Array-based: O(N)
    - Link-based: O(1)
- Special Case – For the `ArrayUnbndQueue` the size "penalty" can be minimized but the `enlarge` method is O(N)

Homework

6(**a** and **b**), **14**, **32**

Keep reviewing sorting methods on Wikipedia
http://en.wikipedia.org/wiki/Sorting_algorithm

If you choose to do a final project, you will be competing 5 of these algorithms with data files that I provide to you.

If you choose to do the final, it will be a take home exam. You will not be given questions and answers, however you will have 1 week to complete it and e-mail it to me… That means it will be long, and yes there will be coding involved, the coding section will account for ~40% of the final exam.