Abstract Data Types

Wolfgang Schreiner
Research Institute for Symbolic Computation (RISC)
Johannes Kepler University, Linz, Austria

Wolfgang.Schreiner@risc.jku.at
http://www.risc.jku.at

Abstract Data Types

Arrays and linked lists are examples of collections.

- Collection: an object that serves as a repository for other objects.
  - Provides by various methods access to the objects stored.
  - Example: collection class `Table`.
    - Implemented with the help of an array.
- Abstract Data Type: a data structure with a defined interface.
  - ADT can be implemented by various other low-level data structures.
  - Examples: `Table`, stacks, queues.
    - Can be implemented by arrays but also by linked lists.

ADTs distinguish between interface and implementation.

Interfaces

Interface and Implementation

Many features of a class are not intended for its users.

- Interface:
  - Public part of a class that is accessible to users of the class.
    - Only legal way to interact with objects of this class.
    - Can be only changed in accordance with users of the class.
- Implementation:
  - Private part of a class that is not revealed to users of the class.
    - Only intended for internal use within the class.
    - Can be changed at any time without notifying users of the class.

Java supports this distinction by some language constructs.

Interfaces in Java

Collection of class constant and abstract object methods.

```
public interface Name
{
  public static final type name = expression; // class constant
  public type name(parameters);               // abstract method
  ...
}
```

- Interface named *Name*
  - Stored in file *Name*.java and compiled like a class.

Abstract object methods must not contain a body.

Implementation of Interfaces

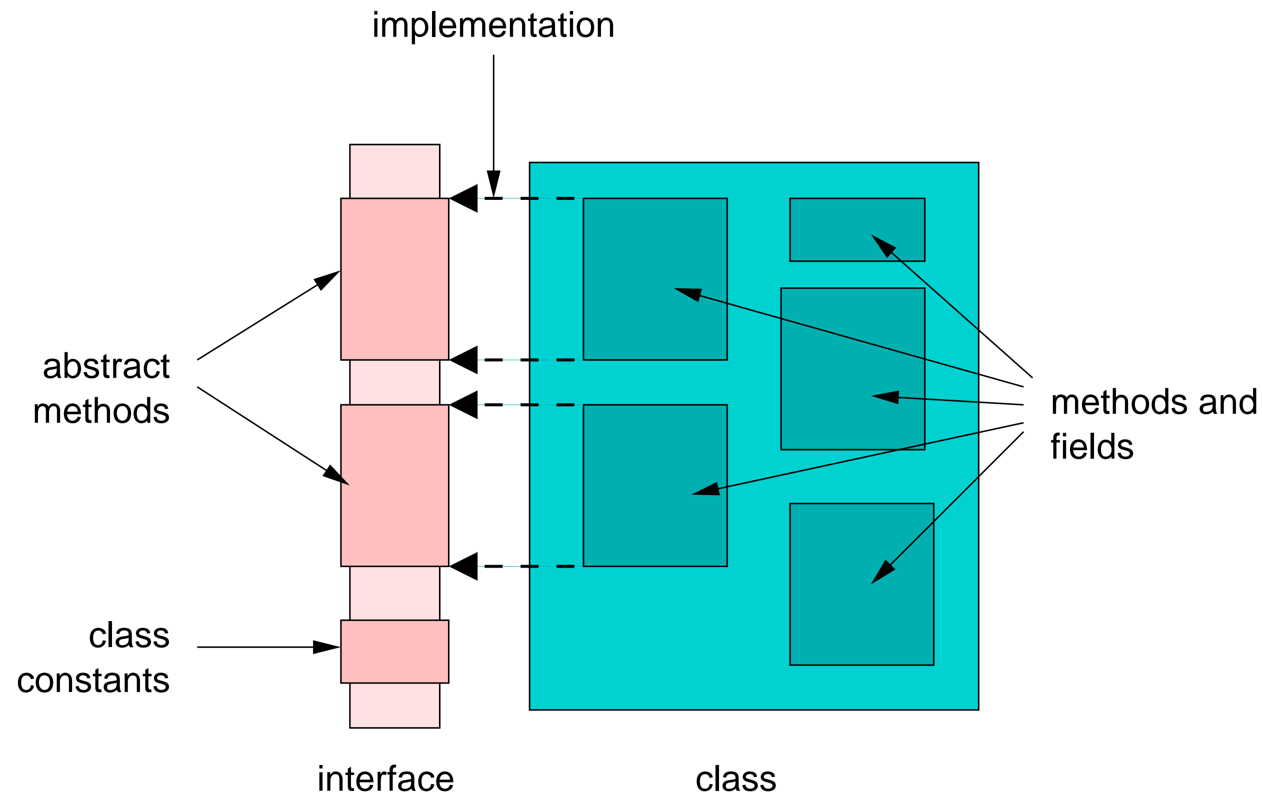
Public part of a class may implement one or more interfaces.

```
public class Class implements Name, ...
{
  public type name(parameters)
  {
    ...
  }
}
```

For every abstract method, the class must provide an implementation.

Implementation of Interfaces

Objects can “hide” behind a “shield” represented by the interface.



Example

```
public interface Container
{
  public static final int size = 100;
  public void addElement(int x);
  public int getElement(int i);
}
```

```
public class Array implements Container
{
  public int[] a = new int[size];
  public int l = 0;

  public void setElement(int i, int x)
  { a[i] = x; }

  public void addElement(int x)
  { setElement(l, x);
    l++;
  }

  public int getElement(int i)
  { return a[i]; }
}
```

Polymorphism via Interfaces

What are Java interfaces good for?

- Interface name may be used to declare the type of a variable.
  - Polymorphism: the variable value may be an object of any class that implements the interface.
  - We may use the interface methods and forget the actual type of the object.
- Use of polymorphic variable:

```
Container c = new Array();
c.addElement(1);
```

- Illegal use of polymorphic variable:

```
c.setElement(0, 1);
int l = c.l;
Container d = new Container();
```

Interfaces may be used in type declarations, not in object creations.

Polymorphic Functions

A polymorphic function has a parameter with an interface type.

```
public int getFirst(Container c)
{
  return c.getElement(0);
}

Container a = new Array();
a.addElement(1);
int i = getFirst(a);

Container l = new List();
l.addElement(1);
int i = getFirst(l);
```

```
public interface Container
{ ...
}

public class Array implements Container
{ ...
}

public class List implements Container
{ ...
}
```

Example

Interfaces help to avoid code modifications.

- Original code:

```
Container c = new Array();
c.addElement(1);
int i = getFirst(c);
```

- Changed code:

```
Container c = new List();
... (unchanged)
```

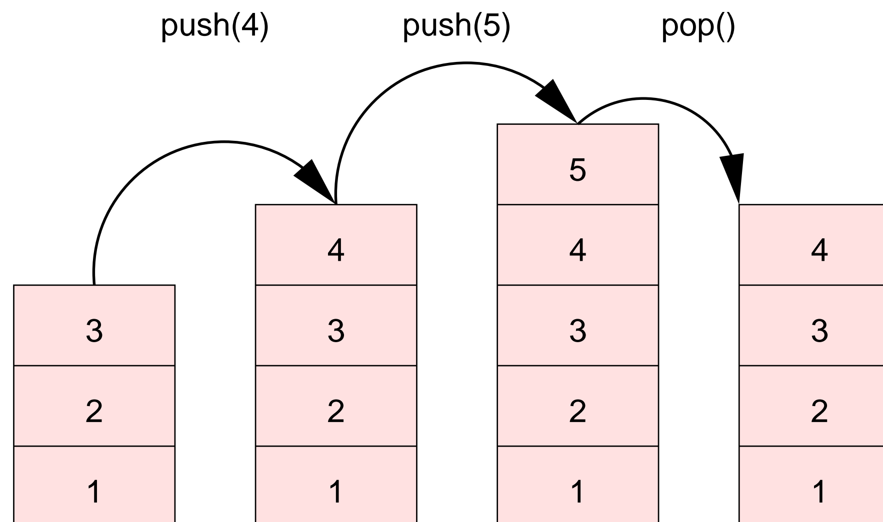
Actual type only used in object creation.

Stacks

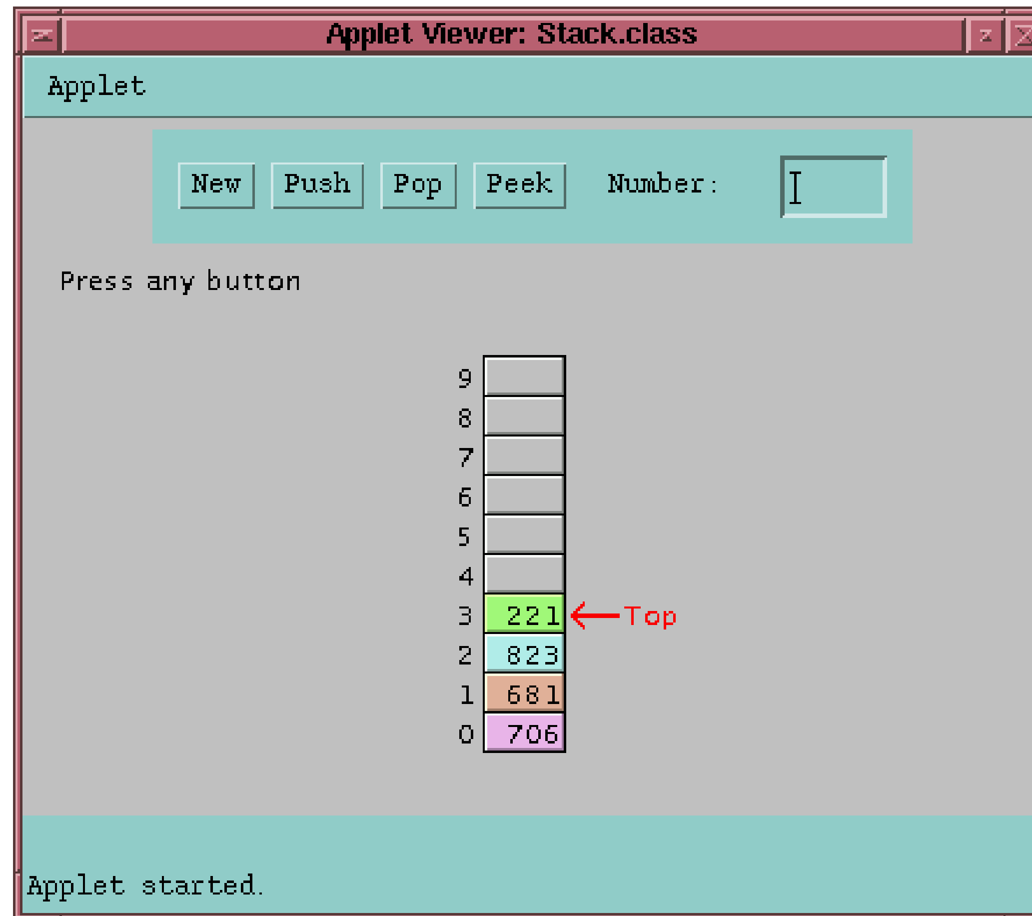
Stacks

A stack is a pile of items.

- Access only to item to top of stack.
  - Push: add a new item to the top.
  - Pop: remove an item from the top.
  - Last item pushed is the first item popped (LIFO: "last in, first out").



Java Applet



Stack Interface

```
public interface Stack
{
  public boolean isEmpty();
  public void push(int t);
  public int pop();
  public int top();
}
```

For method specifications, see the lecture notes.

Stack Implementation as a List

```
public class ListStack implements Stack
{
  public Node head;

  public boolean isEmpty() { return head == null; }
  public void push(int t)
  { Node node = new Node(t, head);
    head = node;
  }
  public int pop()
  { int value = top();
    head = head.next;
    return value;
  }
  public int top() { return head.value; }
}
```

Stack Implementation as an Array

```
public class ArrayStack implements Stack
{
  ...
}
```

- Java standard library provides a class `Stack`.
  - Implemented on top of class `Vector`.
  - Objects can store an arbitrary number of elements.

See the Java documentation.

Stack Use

```
Stack s = new ListStack();
s.push(3); s.push(4); s.push(5);
int v0 = s.pop(); System.out.println(v0);
int v1 = s.pop(); System.out.println(v1);
```

```
5
4
```

Example: Parentheses Matching

Take a string of parentheses and return true if parentheses match.

```
(([()[]][])[]) → true
(([()[])[])[]) → false
```

- Idea: use a stack to record open parantheses.
  - When we encounter an open parentheses, we push it on the stack.
  - When we encounter a closing parenthesis, we pop one element off the stack.
    - Check whether the closing parenthesis matches the popped one.

Simple solution to an otherwise difficult problem.

Example

[()[]]

Next Character	Stack Contents
[	[
(	[(
)	[
[	[[
]	[
]	

Example

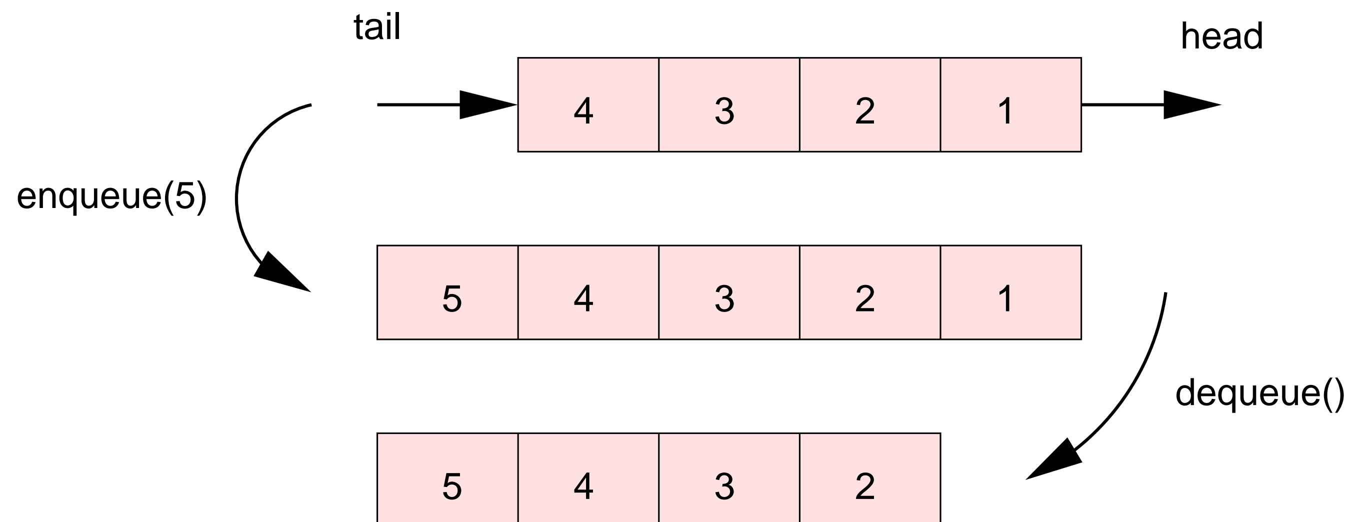
```
public static boolean checkParens(String word)
{ int n = word.length();
  Stack stack = new ListStack();
  for (int i = 0; i < n; i++)
  { char ch = word.charAt(i);
    switch (ch)
    { case '(': case '[':
        stack.push(ch);
        break;
      case ')': case ']':
        if (stack.isEmpty()) return false;
        char ch0 = (char)stack.pop();
        if (ch == ')' && ch0 != '(') return false;
        if (ch == ']' && ch0 != '[') return false;
        break;
      default: break;
    }
  }
  return stack.isEmpty();
}
```

Queues

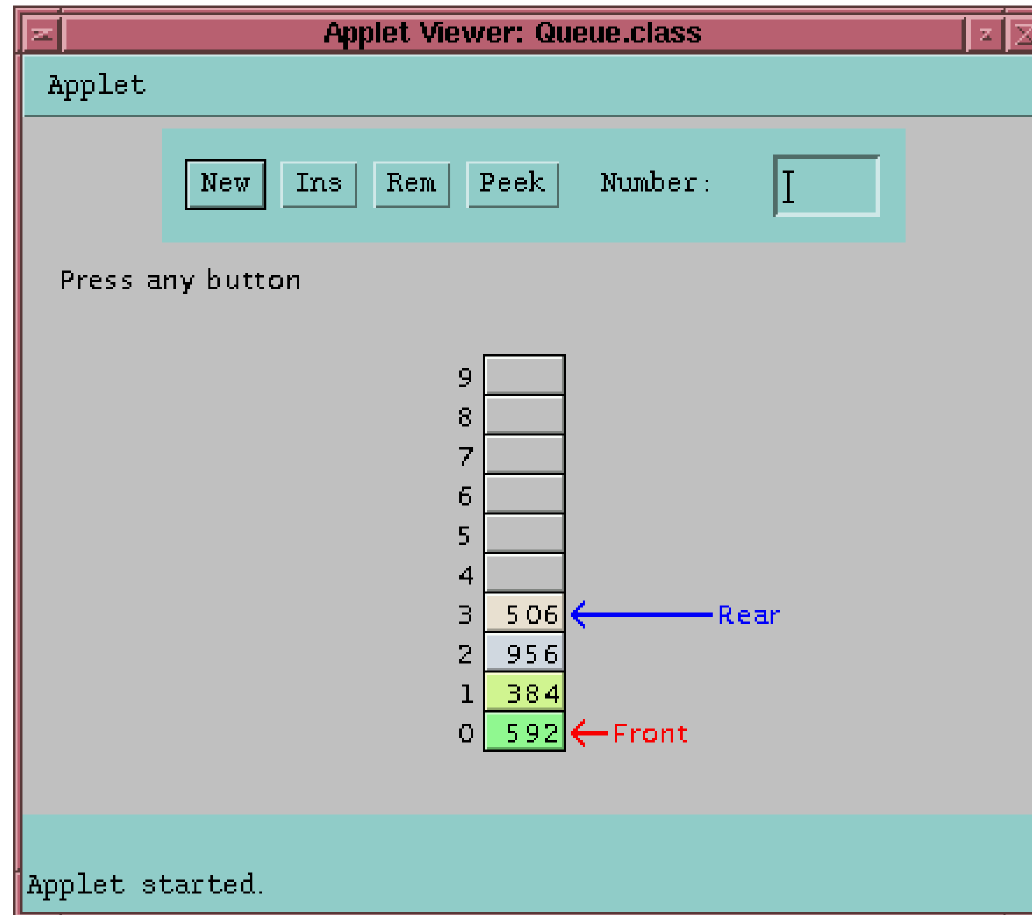
Queues

A queue is a sequence of elements.

- Accessed from both ends of the sequence.
  - Enqueue: new elements are added at the tail.
  - Dequeue: elements are removed from the head.
  - First item enqueued is the first item dequeued (FIFO: "first in, first out").



Java Applet



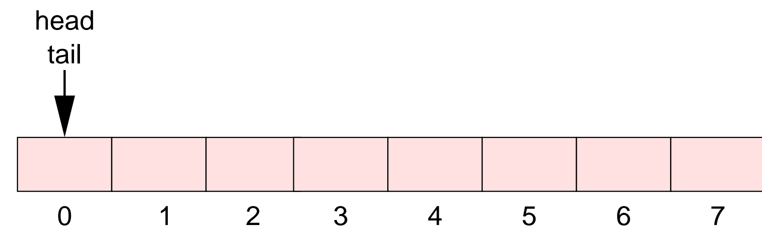
Queue Interface

```
public interface Queue
{
  public boolean isEmpty();
  public void enqueue(int e);
  public int dequeue();
  public int peek();
}
```

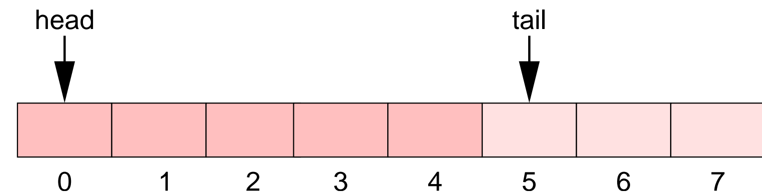
For method specifications, see the lecture notes.

Queue Implementation as an Array

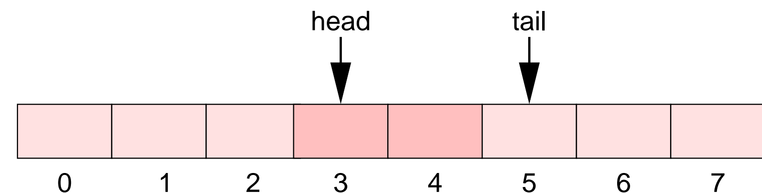
- Two indices describe the queue:
  - *head* refers to the position of the head element of the queue.
  - *tail* refers to the first free array slot where new elements can be enqueued.
- Initial state:



- 5 elements inserted:



- 3 elements removed:

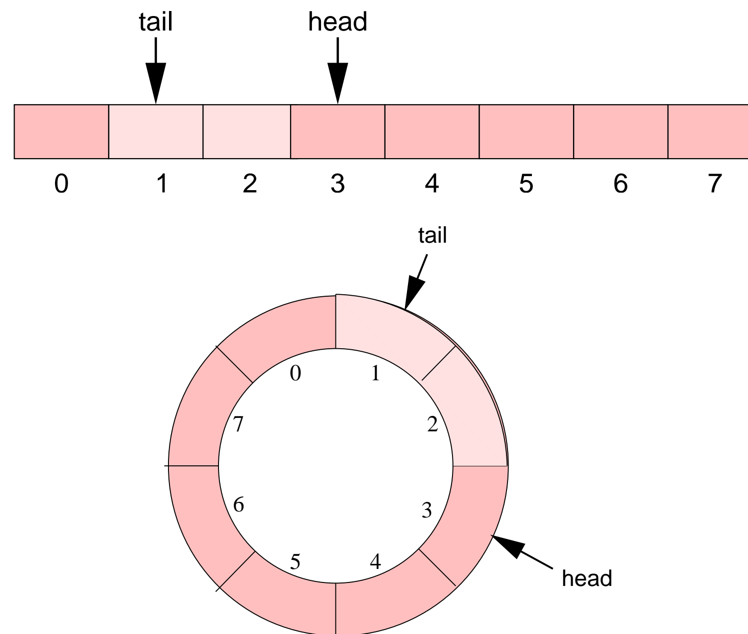


How to insert four more elements?

Circular Array

If an index exceeds one end, it moves back in at the other end.

- 4 elements inserted:



Arbitary many queue operations can be performed.

Queue Implementation as an Array

```
public class ArrayQueue implements Queue
{
  public int[] a;
  public int head = 0;
  public int tail = 0;
  public int count = 0;

  public ArrayQueue(int n)
  {
    a = new int[n];
  }

  public int isEmpty() { return count == 0; }
  ...
```

Queue Implementation as an Array

```
  public void enqueue(int value)
  { if (count == a.length) System.exit(-1);
    count = count+1;
    a[tail] = value;
    tail = (tail+1) % a.length;
  }

  public int dequeue()
  { count = count-1;
    int value = a[head];
    head = (head+1) % a.length;
    return value;
  }

  public int peek() { return a[head]; }
}
```

Queue Use

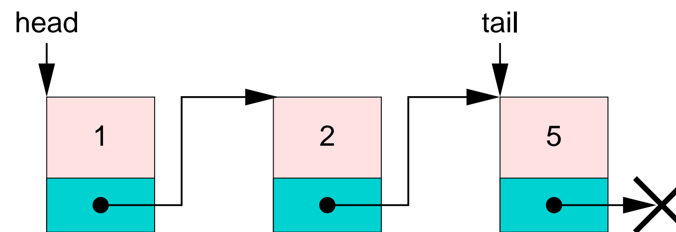
```
Queue q = new ArrayQueue(8);
q.enqueue(2); q.enqueue(3); q.enqueue(5);
int v0 = s.dequeue(); System.out.println(v0);
int v1 = s.dequeue(); System.out.println(v1);
```

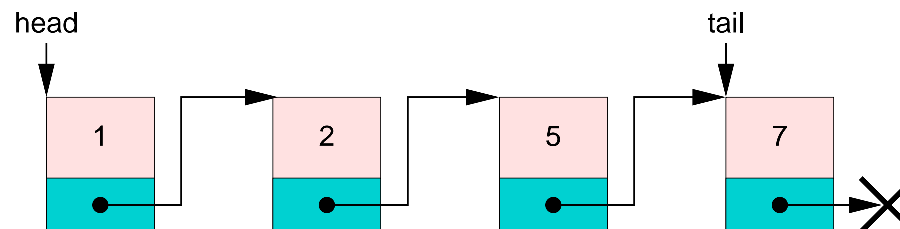
```
2
3
```

Queue Implementation as a List

Problem: linked list can be only accessed at one end.

- Need two pointers to maintain the queue.
  - $head$ refers to the first list node.
  - $tail$ refers to the last list node.
  - Enqueue: link element to last node and update $tail$.





Queue Implementation as a List

```
public class ListQueue implements Queue
{
  public Node head;
  public Node tail;

  public boolean isEmpty()
  {
    return head == null;
  }

  public int peek()
  { return head.value; }

  ...
}
```

Queue Implementation as a List

```
public void enqueue(int e)
{ Node node = new Node(e);
  if (head == null)
    head = node;
  else
    tail.next = node;
  tail = node;
}

public int dequeue()
{ int value = head.value;
  head = head.next;
  if (head == null) tail = null;
  return value;
}
```

Comparison of Queue Implementations

- Cyclic arrays:
  - Simple static data-structure.
  - Very efficient enqueue and dequeue operations.
  - Only finite capacity (if array is not re-allocated).
  - Used in operating systems to buffer events.
    - Events are processed in the order in which they have arrived.
- Linked lists:
  - Simple enqueue and dequeue operations.
  - Number of elements is unbounded.
  - Enqueue operations are more costly.
    - A new list node is required for every enqueue operation.
  - Used for application software.