CS102

Introduction to data structures, algorithms, and object-oriented programming

April 4, 2016
Java Parameterized (generic) Types

1. Makes more bugs detectable at compile time

2. Types of objects in a container are specified by arguments inside <>s.

3. Eliminates need for some type-casting.
ArrayList\<E\>

Parameterized type that removes the complexity of dealing with objects in a container.

ArrayList is implemented as a dynamic array because each item must be accessed quickly.

A dynamic array has an initial fixed size that is incremented (usually doubled) when capacity is exceeded.
ArrayList<E> Class Methods

- boolean add(E): appends item to end of list
- void add(i, E): inserts the item E at position i
- boolean contains(o): Returns true if o in list
- E get(i): Returns element at position i in list
- E indexOf(o): Returns index position of o in list
- E size(): Returns number of objects in list
- E remove(i): Removes and returns element at position i
Abstract Data Types

The term abstract data type, or ADT, refers to a set of possible values and a set of operations on those values, without any specification of how the values are to be represented or how the operations are to be implemented.

There are often several different ways to implement the same abstract data type.
ArrayList<E> Class

Notice that all stack operations can be implemented with an ArrayList (suppose it's called arrly):

You can choose to implement a stack with an ArrayList. You can choose where you want the top of the stack

- push(item) = arrly.add(0,item)
- pop() = arrly.remove(0)
- isEmpty() = arrly.size()==0
- etc.
Writing your own generic types

A generic type is an interface that uses a capital letter in which the actual type is substituted for the generic type.

When you create the interface, you specify where the type should occur by using <> wherever the type name should be specified. The single capital letters are conventional for specifying a generic type.
import java.util.*;
public interface Stack<T> {
    // return number of elements in stack
    public int size();
    // return true if stack has 0 elements
    public boolean isEmpty();
    // return the top element and throw exception if stack.isEmpty()
    public T peek() throws StackEmptyException;
    // add a new item to the top of the stack
    public void push(T item);
    // remove and return the top element and throw exception if empty
    public T pop() throws StackEmptyException;
}

A stack is called a LIFO data structure, meaning "Last In, First Out".
Writing your own Exception Classes

• The StackEmptyException class. Better to make this a class on its own because it may be used in an interface and a class.

```java
public class StackEmptyException extends RuntimeException {
    public ArgumentOutOfRangeException(String err) {
        super(err);
    }
}
```
The Queue Abstract Data Type
The Queue ADT

• The Queue ADT stores arbitrary objects
• Insertions and deletions follow the first-in first-out (FIFO) scheme
• Insertions are at the rear of the queue and removals are at the front of the queue
• Main queue operations:
  – enqueue(Object o): inserts element o at the rear of the queue
  – Object dequeue(): removes and returns the element at the front of the queue
The Queue ADT

• Auxiliary queue operations:
  – Object *front*(): returns the element at the front without removing it
  – int *size*(): returns the number of elements stored
  – boolean *isEmpty*(): indicates whether no elements are stored

• Exceptions
  – Attempting the execution of dequeue or front on an empty queue throws an *QueueEmptyException*
Applications of Queues

• Direct applications
  – Waiting lists, bureaucracy, TSA lines
  – Access to shared resources (e.g., printer)

• Indirect applications
  – Auxiliary data structure for algorithms
  – Component of other data structures
Queue Interface in Java

- Java interface corresponding to the Queue ADT
- Requires the definition of class QueueEmptyException

```java
public interface Queue<T> {
    public int size();
    public boolean isEmpty();
    public T front() throws QueueEmptyException;
    public void enqueue(T item);
    public T dequeue() throws QueueEmptyException;
}
```
Array-based Queue

- Use an array of size $N$ in a circular fashion
- Two variables keep track of the front and rear
  - $f$ index of the front element
  - $r$ index immediately past the rear element
- Array location $r$ is kept empty, holds position for next element to be enqueued
Queue Operations

- Use the modulo operator (remainder of division) when resetting front and rear and making your queue robust to wrap-around.

```
Algorithm size()
    return (N - f + r) mod N

Algorithm isEmpty()
    return (f = r)
```
Queue Operations (cont.)

- Operation enqueue throws an exception if the array is full
- This exception is implementation-dependent; not needed if implementing with ArrayList.

Algorithm `enqueue(o)`
```plaintext
if size() = N - 1 then
    throw QueueFullException
else
    Q[r] ← o
    r ← (r + 1) mod N
```

![Diagram of queue operations](image)
Queue Operations (cont.)

- Operation dequeue throws an exception if the queue is empty
- This exception is specified in the queue ADT

Algorithm `dequeue()`

```
if isEmpty() then
    throw QueueEmptyException
else
    o ← Q[f]
    f ← (f + 1) mod N
    return o
```
Growable ArrayList Queue

• An ArrayList has enough flexibility to implement the stack or queue ADT
• For an ArrayList called arrQueue, the head of the queue could be at position 0 and the tail could be at position arrQueue.size()-1
The Deque ADT

• The Deque ADT stores arbitrary objects
• The term deque comes from double-ended queue
• Insertions and deletions are allowed at both ends
• Main deque operations:
  – void insertFirst(object): inserts object at the front
  – void insertLast(object): inserts object at the rear
  – object removeFirst(): remove and return element at front
  – object removeLast(): remove and return element at the rear
a) roll(4, 1)

```
E  D  C  B  A
S  D  C  B  E
D  C  B  E  A
```

```
D.enqueue(S.pop())
(operations repeated 4 times)
```

```
D.enqueue(D.dequeue())
(operations repeated 1 time);
S.push(D.removeLast())
(operations repeated 4 times)
```

b) roll(3, 2)

```
E  D  C  B  A
S  C  E  D  B
A  C  E  D  B
```

```
D.enqueue(S.pop())
(operations repeated 3 times)
```

```
D.enqueue(D.dequeue())
(operations repeated 2 times);
S.push(D.removeLast())
(operations repeated 3 times)
```

c) roll(2, 4)

```
E  D  C  B  A
S  E  D  C  B
A  E  D  C  B
```

```
D.enqueue(S.pop())
(operations repeated 2 times)
```

```
D.enqueue(D.dequeue())
(operations repeated 2 times);
S.push(D.removeLast())
(operations repeated 4 times)
```

```
D.enqueue(D.dequeue())
(operations repeated 4 times);
S.push(D.removeLast())
(operations repeated 3 times)
```