Abstract Data Types (ADTs): Stacks and Queues

Brief Discussion of Interfaces and Abstract Classes

Because abstract date types tend to employ the use of interfaces, it seems beneficial to discuss them before continuing on with abstract data types. Let us consider how many ways there are to solve a particular problem with computer science. Even just consider the different method headers that are available. No normal person has the patience to sift all the different types of methods to figure out how to use them all. Maybe if there were a way to force everyone to use the exact same method headers so that if you know what one method is supposed to do, you know what all of them are supposed to do. You can think of interfaces as a way to do that.

Interfaces usually just serve as code templates and are only composed of method headers and empty method bodies (abstract methods). Any class that implements an interface will be forced to override the empty methods. The typical driver is not so interested in how the car works under the hood, but just that it will act like a car. The same can be said of a general user of some class. Some of the skeleton code provided for some of the homework assignments can be considered examples of this if they were declared to be interfaces.

A concept similar to an interface is an abstract class. An ***abstract class*** can be thought of a cross between an interface and a normal class, because it contains regular methods and can contain, though not necessarily, abstract methods. It is important to note that it is possible for a class to implement more than one interface, but it can only extend one class, abstract, regular or something else. Abstract classes and interfaces also cannot be instantiated.

For more information and examples, please visit:
http://java.sun.com/docs/books/tutorial/java/IandI/abstract.html.

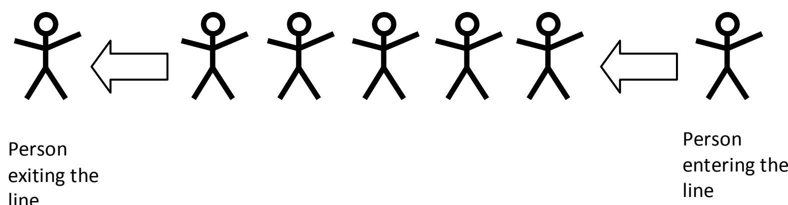
Abstract Data Types

An ***abstract data type (ADT)*** is a theoretical set of specifications of a data set and the set of operations that can be performed on the data within a set. A data type is termed abstract when it is independent of various concrete implementations.

Abstract Data Types: Stacks and Queues

Queues

A common abstract data type is a queue. A real life example of a queue is a line of people waiting for some event. The first person in line will be served first, while the last person last.

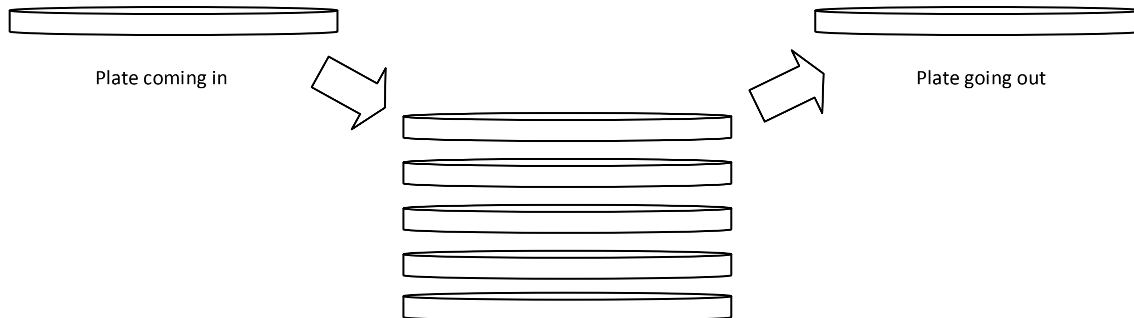


A ***queue*** is a first in, first out (FIFO) structure or in the other sense, a last in, last out (LILO) structure. A queue is sometimes generalized as a structure where insertion (enqueue,

pronounced N-Q) occur at one end and removal (dequeue, pronounced D-Q) occurs at the other end.

Stacks

Another abstract data type is a stack. A good example of a stack is a stack of dishes in the dining hall. You cannot get the one on the bottom unless you pick up all the ones on top of it.



A ***stack*** is a first in, last out (FILO) structure, or a last in, first out (LIFO) structure. A stack is sometimes generalized a structure with insertion (push) and removal (pop) all at the same end.

Abstract data types and interfaces

What do abstract data types have to do with interfaces? Back track to the definition of an abstract data type. An abstract data type has a set of data and a set of operations. An interface is what will dictate what operations are necessary to the data type.

Stacks and queues in action

Given this set of numbers {1, 2, 3, 4, 5} put them all in a queue first and then remove them. Second try putting them into a stack and then removing them. Compare the differences between a queue and a stack. Note a queue is sometimes denoted with <-||<- or ->||-> with arrowheads on opposite sides, while a stack is ||-><- or -><-||.

Set: { 1, 2, 3 }
Queue: <-||<-

enqueue(1)
Set: { 2, 3 }
Queue: <-| 1 |<-

enqueue(2)
Set: { 3 }
Queue: <-| 1 2 |<-

enqueue(3)
Set: { }
Queue: <-| 1 2 3 |<-

dequeue()
returns: 1
Queue: <-| 2 3 |<-

dequeue()
returns: 2
Queue: <-| 3 |<-

dequeue()
returns: 3
Queue: <-||<-

Set: { 1, 2, 3 }
Stack: | |-><-

push(1)
Set: { 2, 3 }
Stack: | 1 |-><-

push(2)
Set: { 3 }
Stack: | 1 2 |-><-

push(3)
Set: {}
Stack: | 1 2 3 |-><-

pop()
returns 3
Stack: | 1 2 |-><-

pop()
returns 2
Stack: | 1 |-><-

pop()
returns 1
Stack: | |-><-

Notice the difference in the removal of elements from a stack compared to a queue, but also notice that this difference only exists because of the way the arrows were chosen.

More Practice Problems

For the following sets, insert the elements into a queue and remove the elements again. Do the same for a stack. Be sure to indicate your arrows clearly.

1. Set = { 4, 7, 2, 8, 1, 5, 9 }

2. Set = {the, lazy, dog, jumped, over, the, fence }

3. Set = {aa, bb, cc, dd, ee}