Stacks and Queues

Stacks are LIFO

Queues are FIFO

Client code only performs basic operations

Implementation abstracted away from interface- details are completely separated away

Implementation doesn’t know details from client- only understands the implementation

This modular programming style allows reusable libraries and better performance

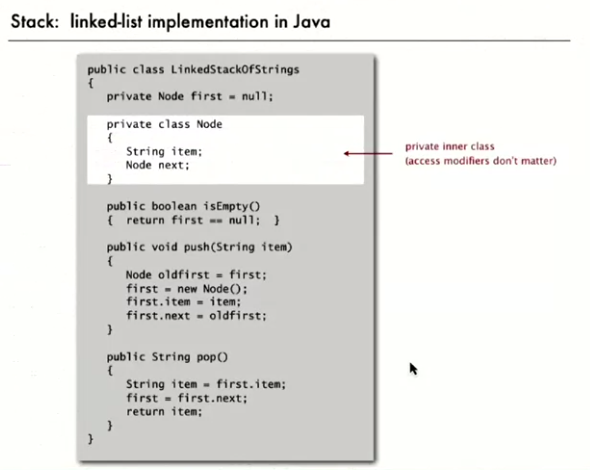
Stacks

Push: insert a object onto a stack

Pop: remove and return string most recently added

isEmpty: is the stack empty?

Size: number of objects on the stack

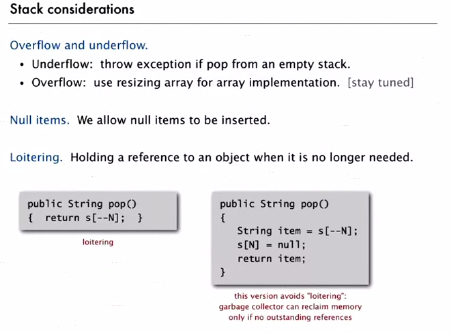
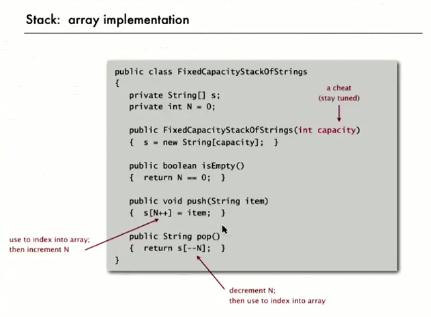


Can also implement a stack with arrays

Index N is the top of the stack where the next item will go. With array s[] we store N items.

Push() adds a new item at s[N]

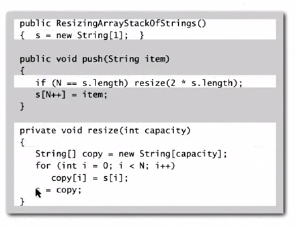
Pop() removes item from s[N-1]



Resizing arrays

This will make array stacks more flexible (instead of specifying a static array size)

**Repeated doubling:** when the array fills up, create a new array of twice the size and copy items over



Shrink array to 50% when 25% full. If you shrink to 50% at 50%, this can lead to thrashing. In other words, quadratic time for operations where the stack repeatedly doubles and halves arrays on the border of 50% with many calls to push() and pop().

Linked lists stacks are slower:

* Every operation takes constant time in the worst case
* It uses extra time and space to deal with the links
* Guarantees every *operation* is fast- but not the overall time

Resizing array stacks

* Every operation takes constant amortized time (total time averaged over the whole process)
* There’s less wasted space
* Guarantees a better overall time- but some steps may slow due to doubling/halving

Trade off with resizing array stacks is that doubling and halving operations slow the process. Therefore, if you can’t accept any delays in performance and need to count on constant times (e.g. while landing an airplane, can’t wait for a ‘random’ doubling/halving program event).

Queues

Similar API to that of stacks, but names and semantics are different:

Queue() : create an empty queue

Enqueuer(item): insert into queue (at front)

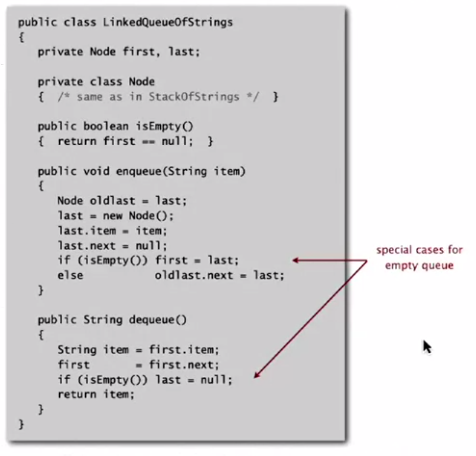
Dequeuer(): remove and return item least recently added (from end)

isEmpty(): is the queue empty?

Size(): what is the size of the queue?

Maintain two references (to front and end of the list)

Linked list implementation:



Array implementation API

Use array q[] to store N items

Enqueue() adds new items at q[tail]

Dequeuer() removes items at q[head]

Update head and tail modulo *capacity*

Add resizing array

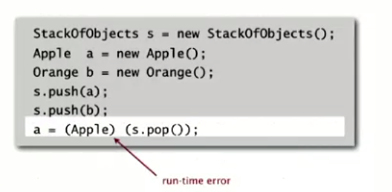


\*\*Once you get capacity you have to resize to 0

Using stacks, queues, etc with different data types:

Options:

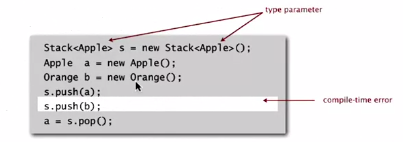
* Create a new class and rename data types to new data type
* Cast the data types to new data type



Problem: client code does casting and create bugs

Last option:

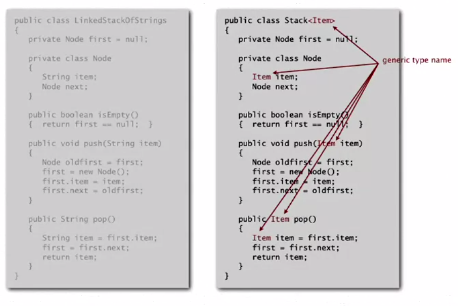
GENERICS



Why?

* Avoid casting in a client
* Discover mismatch errors in compile-time instead of run-time

With generics we can replace (for example) ‘strings’ with another generic type name:]



We declare different types in signatures and/or parameters, then in angle brackets in class declaration

Doesn’t work so well with arrays- there is no generic array declaration in Java, so we need to cast



**Good code minimizes casts**

Generic types are for objects, casting down from an array of objects

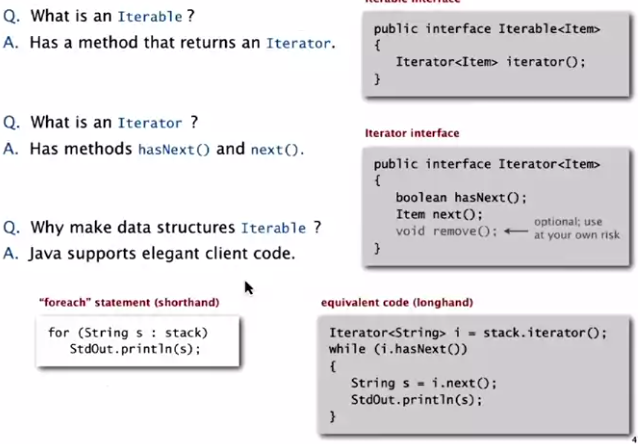
We need to use wrapper types for primitives ( <Integer> )

**Autoboxing** will automatically cast between primitive types and wrappers

Iteration

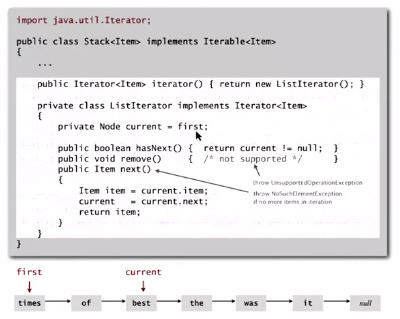
**Iterable** is a class that has a method that returns an iterator

And **iterator** is a class that has methods hasNext() and next()

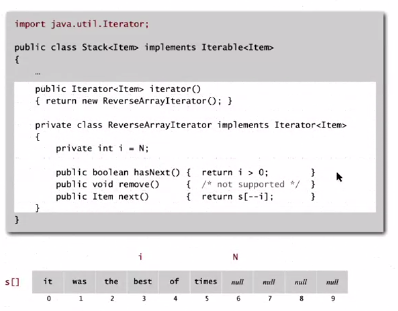


‘implements Iterable'

Linked list implementation



Array implementation



Bags

When you don’t care about the order of insertion and removal and order of iteration

Bag API

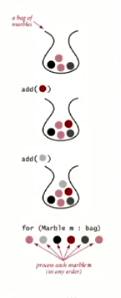
Public class Bag<Item> implements Iterable<Item>

Bag() creates empty bag

Void add(Item x) inserts an item into the bag

Int size() number of items in the bag

Iterable<Item> iterator() iterator for all items in bag



***\*\* Many data structures are found in existing (Java) libraries, but…***

*Java has stacks, queues (utilizing resizing arrays), but unfortunately the bloated API has too many operations to appeal to too many use cases. Performance usually suffers even for simple clients.*

Stack and queue applications

Some applications

* Back button on browsers (places you’ve been are stored on a stack)
* Using ‘undo’ in a word processor
* Compilers implement functions using stacks:
  + Function calls push local environment and return address
  + Pop returns address and local environment
  + Recursive functions heavily use stacks

