Collaboration policy

Programs: Do not use someone else's code unless specifically authorized

Exceptions

- Code from course materials OK [cite source]
- Coding with partner OK after first assignment [stay tuned]

Where to get help

- Email (but no code in email)
- Office hours
- Lab TAs in Friend 008/009
- Bounce ideas (but not code) off classmates

Note: Programming in groups except as above is a serious violation.

Exercises: Write up your own solutions (no copying)

- working with classmates is encouraged
- checking solutions is OK

Stacks and Queues

- stacks
- dynamic resizing
- queues
- generics
- applications

Fundamental data types.

- Values: sets of objects
- Operations: insert, remove, test if empty.
- Intent is clear when we insert.
- Which item do we remove?

LIFO = "last in first out"

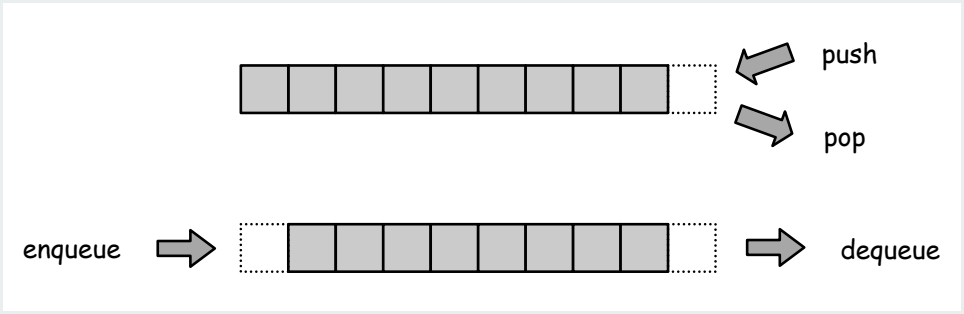
Stack.

- Remove the item most recently added.
- Analogy: cafeteria trays, Web surfing.

FIFO = "first in first out"

Queue.

- Remove the item least recently added.
- Analogy: Registrar's line.



Client, Implementation, Interface

Separate interface and implementation so as to:

- Build layers of abstraction.
- Reuse software.
- Ex: stack, queue, symbol table.

Interface: description of data type, basic operations.

Client: program using operations defined in interface.

Implementation: actual code implementing operations.

Client, Implementation, Interface

Benefits.

- Client can't know details of implementation $\Rightarrow$
  client has many implementation from which to choose.
- Implementation can't know details of client needs $\Rightarrow$
  many clients can re-use the same implementation.
- Design: creates modular, re-usable libraries.
- Performance: use optimized implementation where it matters.

Interface: description of data type, basic operations.
Client: program using operations defined in interface.
Implementation: actual code implementing operations.

- stacks
- dynamic resizing
- queues
- generics
- applications

Stacks

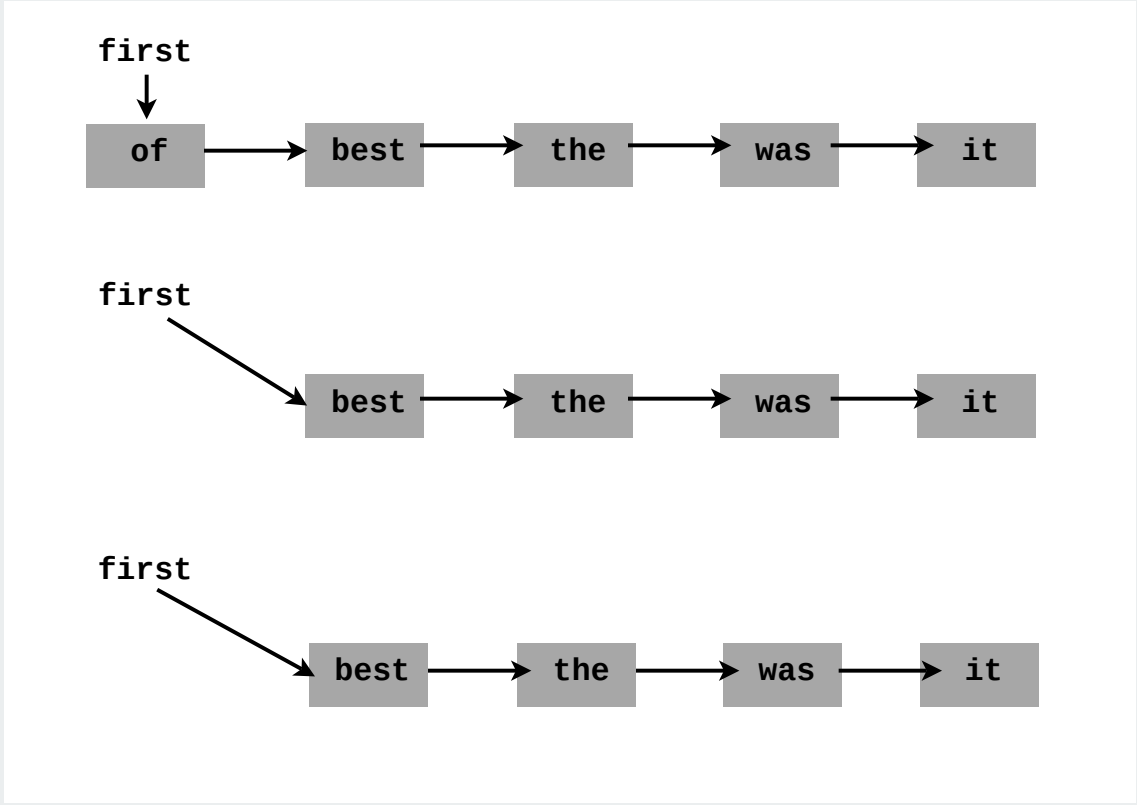
Stack operations.

- `push()` Insert a new item onto stack.
- `pop()` Remove and return the item most recently added.
- `isEmpty()` Is the stack empty?


```
public static void main(String[] args)
{
   StackOfStrings stack = new StackOfStrings();
   while(!StdIn.isEmpty())
   {
      String s = StdIn.readString();
      stack.push(s);
   }
   while(!stack.isEmpty())
   {
      String s = stack.pop();
      StdOut.println(s);
   }
}
```

a sample stack client

Stack pop: Linked-list implementation

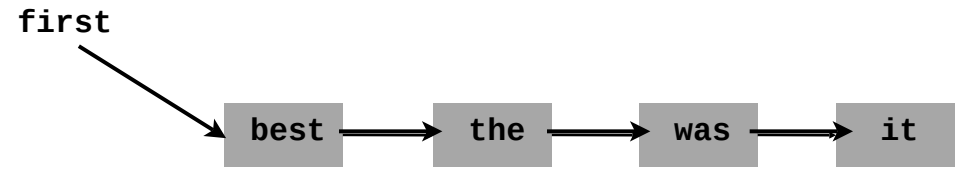



```
item = first.item;
```

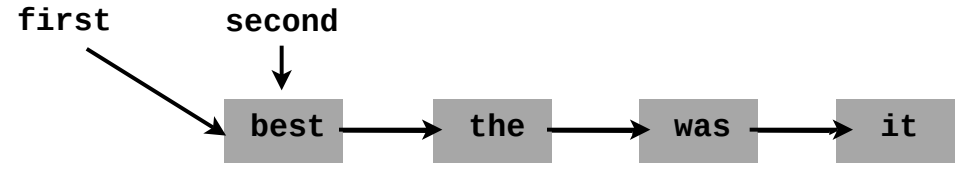
```
first = first.next;
```

```
return item;
```

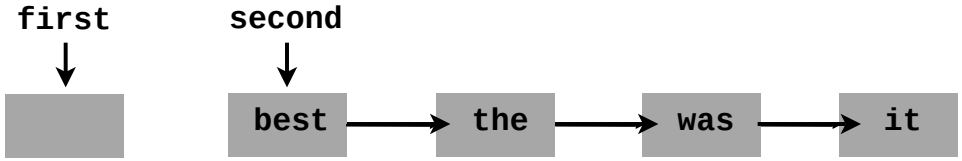
Stack push: Linked-list implementation



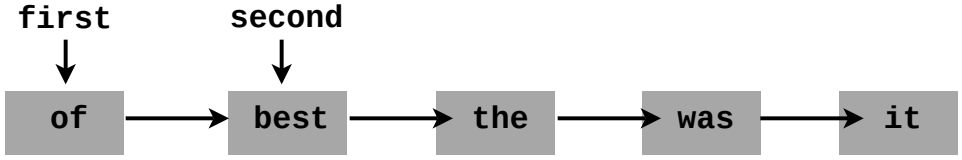

```
second = first;
```




```
first = new Node();
```




```
first.item = item;
first.next = second;
```



Stack: Linked-list implementation

```
public class StackOfStrings
{
   private Node first = null;

   private class Node
   {
      String item;
      Node next;
   }

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

   public void push(String item)
   {
      Node second = first;
      first = new Node();
      first.item = item;
      first.next = second;
   }

   public String pop()
   {
      String item = first.item;
      first = first.next;
      return item;
   }
}
```

"inner class"

Error conditions?

Example: pop() an empty stack

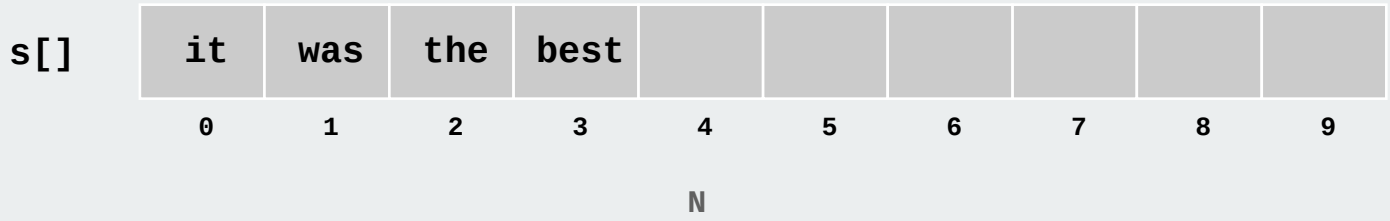
COS 217: bulletproof the code

COS 226: first find the code we want to use

Stack: Array implementation

Array implementation of a stack.

- Use array `s[]` to store `N` items on stack.
- `push()` add new item at `s[N]`.
- `pop()` remove item from `s[N-1]`.



Stack: Array implementation

```
public class StackOfStrings
{
   private String[] s;
   private int N = 0;

   public StringStack(int capacity)
   {  s = new String[capacity];  }

   public boolean isEmpty()
   { return N == 0; }

   public void push(String item)
   {  s[N++] = item;  }

   public String pop()
   {
      String item = s[N-1];
      s[N-1] = null;
      N--;
      return item;
   }

}
```

avoid loitering
(garbage collector only reclaims memory if no outstanding references)

- stacks
- **dynamic resizing**
- queues
- generics
- applications

Stack array implementation: Dynamic resizing

Q. How to grow array when capacity reached?

Q. How to shrink array (else it stays big even when stack is small)?

First try:

- **`push()`**: increase size of `s[]` by 1
- **`pop()`** : decrease size of `s[]` by 1

Too expensive

- Need to copy all of the elements to a new array.
- Inserting N elements: time proportional to $1 + 2 + \ldots + N \approx N^2/2$.

infeasible for large N

Need to guarantee that array resizing happens infrequently

Stack array implementation: Dynamic resizing

Q. How to grow array?

A. Use repeated doubling:

if array is full, create a new array of twice the size, and copy items

no-argument constructor

```
public StackOfStrings()
{ this(8); }

public void push(String item)
{
   if (N >= s.length) resize();
   s[N++] = item;
}
```

create new array
copy items to it

```
private void resize(int max)
{
   String[] dup = new String[max];
   for (int i = 0; i < N; i++)
      dup[i] = s[i];
   s = dup;
}
```

Consequence. Inserting N items takes time proportional to N (not N^2).

$8 + 16 + \ldots + N/4 + N/2 + N \approx 2N$

Stack array implementation: Dynamic resizing

Q. How (and when) to shrink array?

How: create a new array of half the size, and copy items.

When (first try): array is half full?

No, causes thrashing

(push-pop-push-pop-... sequence: time proportional to N for each op)

When (solution): array is 1/4 full (then new array is half full).

```
public String pop(String item)
{
   String item = s[--N];
   sa[N] = null;
   if (N == s.length/4)
      resize(s.length/2);
   return item;
}
```

Not `a.length/2` to avoid thrashing

Consequences.

- any sequence of N ops takes time proportional to N
- array is always between 25% and 100% full

Stack Implementations: Array vs. Linked List

Stack implementation tradeoffs. Can implement with either array or linked list, and client can use interchangeably. Which is better?

Array.

- Most operations take constant time.
- Expensive doubling operation every once in a while.
- Any sequence of N operations (starting from empty stack) takes time proportional to N.

"amortized" bound

Linked list.

- Grows and shrinks gracefully.
- Every operation takes constant time.
- Every operation uses extra space and time to deal with references.

Bottom line: tossup for stacks
but differences are significant when other operations are added

Stack implementations: Array vs. Linked list

Which implementation is more convenient?

	array?	linked list?
return count of elements in stack		
remove the kth most recently added		
sample a random element		

- stacks
- dynamic resizing
- **queues**
- generics
- applications

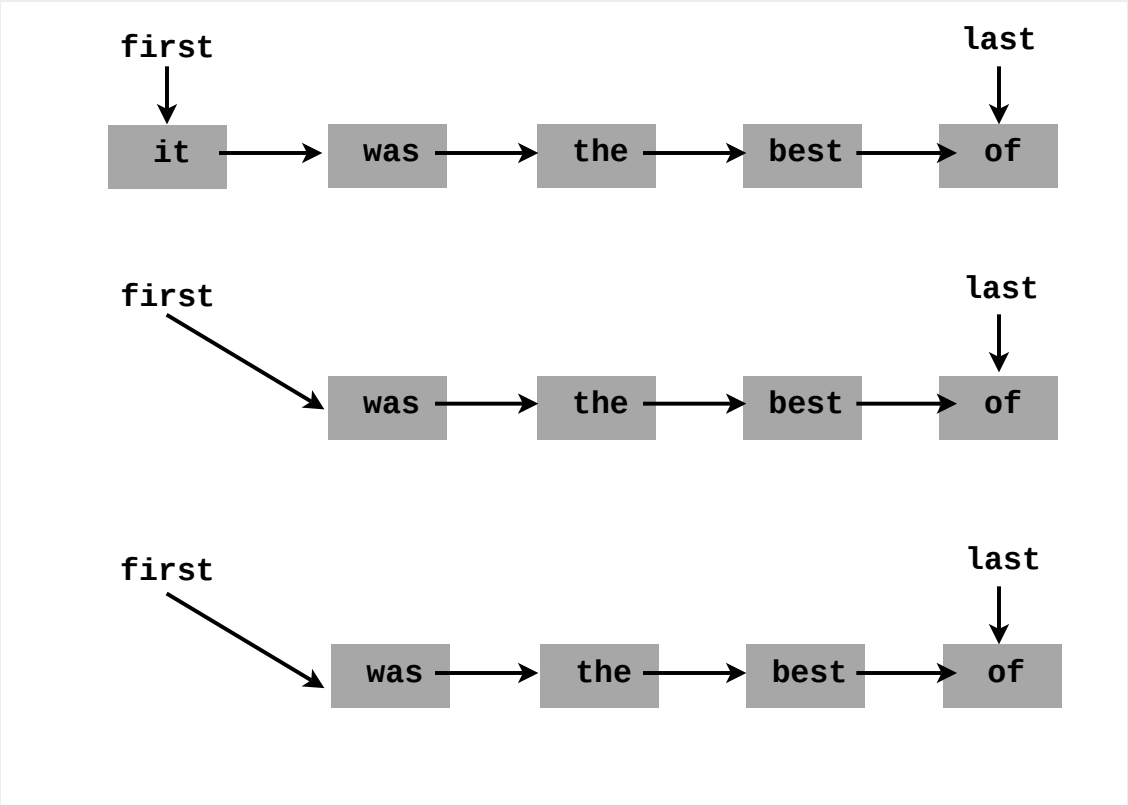
Queues

Queue operations.

- **`enqueue()`** Insert a new item onto queue.
- **`dequeue()`** Delete and return the item least recently added.
- **`isEmpty()`** Is the queue empty?

```
public static void main(String[] args)
{
   QueueOfStrings q = new QueueOfStrings();
   q.enqueue("Vertigo");
   q.enqueue("Just Lose It");
   q.enqueue("Pieces of Me");
   q.enqueue("Pieces of Me");
   System.out.println(q.dequeue());
   q.enqueue("Drop It Like It's Hot");

   while(!q.isEmpty()

      System.out.println(q.dequeue());
}
```

Dequeue: Linked List Implementation

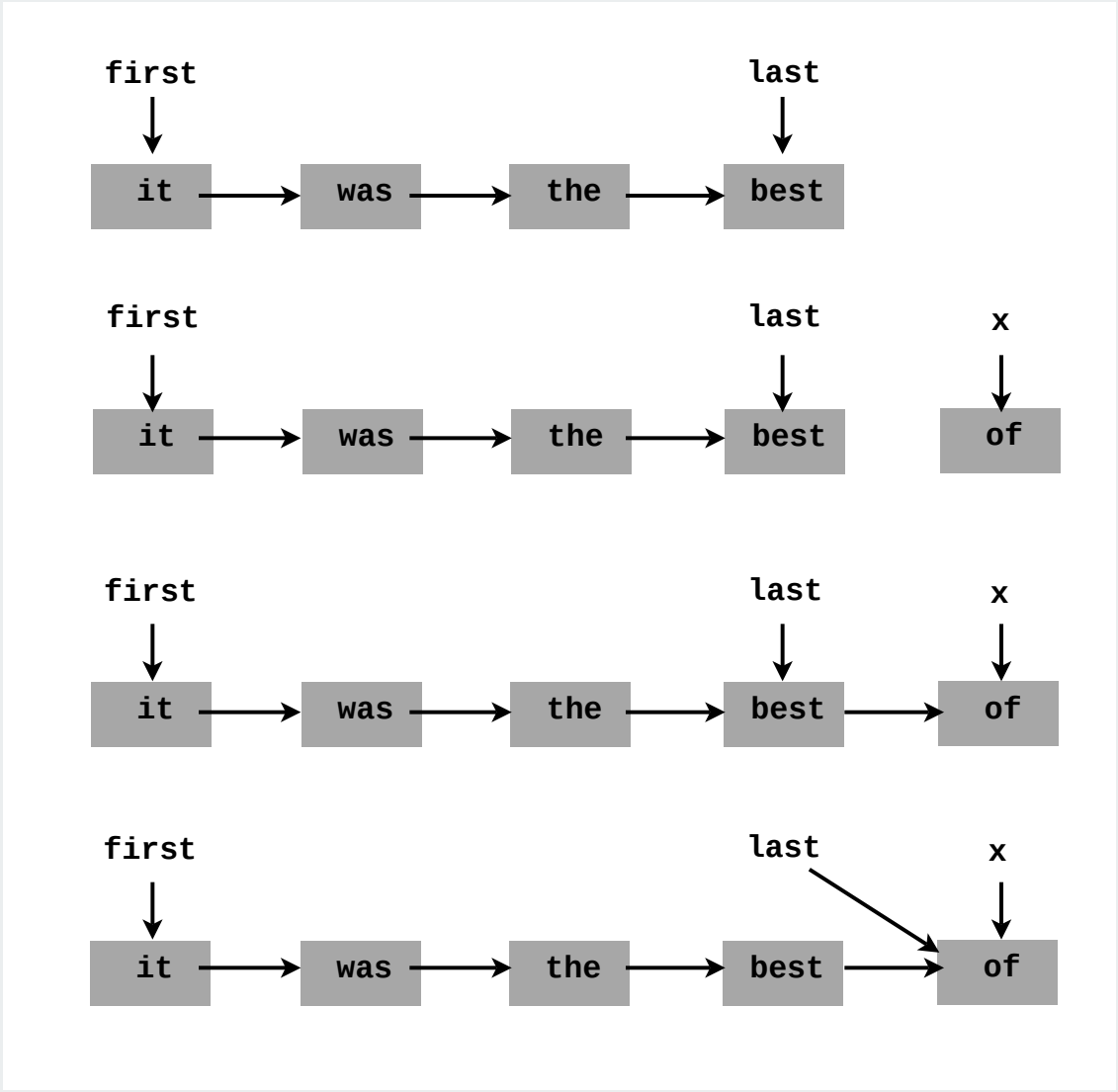


Aside:

dequeue (pronounced "DQ") means "remove from a queue"

deque (pronounced "deck") is a data structure (see PA 1)

Enqueue: Linked List Implementation




```
x = new Node();
x.item = item;
x.next = null;
```

```
last.next = x;
```

```
last = x;
```

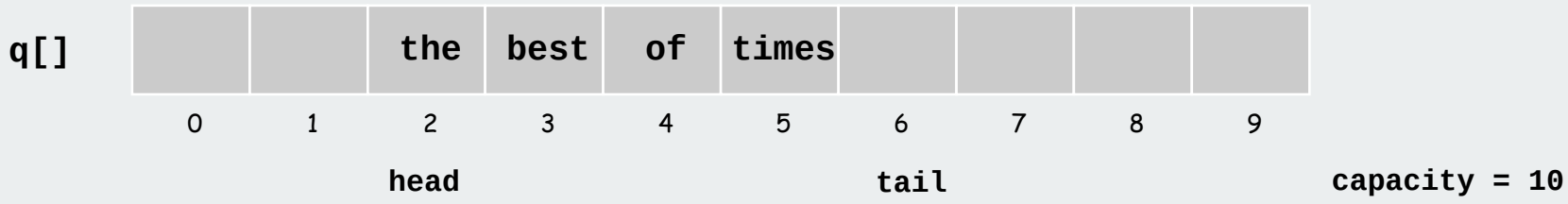
Queue: Linked List Implementation

```
public class QueueOfStrings
{
   private Node first;
   private Node last;

   private class Node
   { String item; Node next; }

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

   public void enqueue(String item)
   {
      Node x = new Node();
      x.item = item;
      x.next = null;
      if (isEmpty()) { first      = x; last = x; }
      else           { last.next = x; last = x; }
   }

   public String dequeue()
   {
      String item = first.item;
      first       = first.next;
      return item;
   }
}
```

Queue: Array implementation

Array implementation of a queue.

- Use array `q[]` to store items on queue.
- `enqueue()`: add new object at `q[tail]`.
- `dequeue()`: remove object from `q[head]`.
- Update `head` and `tail` modulo the `capacity`.



[details: good exercise or exam question]

- stacks
- dynamic resizing
- queues
- **generics**
- applications

Generics (parameterized data types)

We implemented: `StackOfStrings`, `QueueOfStrings`.

We also want: `StackOfURLs`, `QueueOfCustomers`, etc?

Attempt 1. Implement a separate stack class for each type.

- Rewriting code is tedious and error-prone.
- Maintaining cut-and-pasted code is tedious and error-prone.

@#$*! most reasonable approach until Java 1.5 [hence, used in AlgsJava]

Stack of Objects

We implemented: **StackOfStrings**, **QueueOfStrings**.

We also want: **StackOfURLs**, **QueueOfCustomers**, etc?

Attempt 2. Implement a stack with items of type **Object**.

- Casting is required in client.
- Casting is error-prone: run-time error if types mismatch.

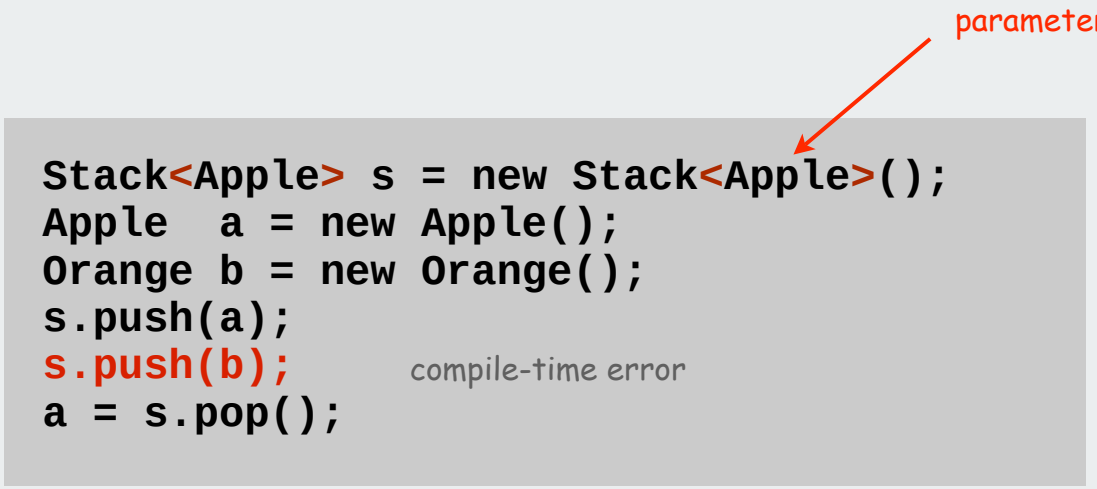
```
Stack  s = new Stack();
Apple  a = new Apple();
Orange b = new Orange();
s.push(a);
s.push(b);
a = (Apple) (s.pop());
```

run-time error

Generics

Generics. Parameterize stack by a single type.

- Avoid casting in both client and implementation.
- Discover type mismatch errors at compile-time instead of run-time.



Guiding principles.

- Welcome compile-time errors
- Avoid run-time errors

Why?

Generic Stack: Linked List Implementation

```
public class StackOfStrings
{
   private Node first = null;

   private class Node
   {
      String item;
      Node next;
   }

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

   public void push(String item)
   {
      Node second = first;
      first = new Node();
      first.item = item;
      first.next = second;
   }

   public String pop()
   {
      String item = first.item;
      first = first.next;
      return item;
   }
}
```

```
public class Stack<Item>
{
   private Node first = null;

   private class Node
   {
      Item item;
      Node next;
   }

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

   public void push(Item item)
   {
      Node second = first;
      first = new Node();
      first.item = item;
      first.next = second;
   }

   public Item pop()
   {
      Item item = first.item;
      first = first.next;
      return item;
   }
}
```

Generic type name

Generic stack: array implementation

The way it should be.

```
public class Stack<Item>
{
   private Item[] s;
   private int N = 0;

   public Stack(int cap)
   {  s = new Item[cap];  }

   public boolean isEmpty()
   { return N == 0; }

   public void push(Item item)
   {  s[N++] = item;  }

   public String pop()
   {
      Item item = s[N-1];
      s[N-1] = null;
      N--;
      return item;
   }

}
```

```
public class StackOfStrings
{
   private String[] s;
   private int N = 0;

   public StackOfStrings(int cap)
   {  s = new String[cap];  }

   public boolean isEmpty()
   { return N == 0; }

   public void push(String item)
   {  s[N++] = item;  }

   public String pop()
   {
      String item = s[N-1];
      s[N-1] = null;
      N--;
      return item;
   }

}
```

@#$*! generic array creation not allowed in Java

Generic stack: array implementation

The way it is: an ugly cast in the implementation.

```
public class Stack<Item>
{
   private Item[] s;
   private int N = 0;

   public Stack(int cap)
   { s = (Item[]) new Object[cap]; }

   public boolean isEmpty()
   { return N == 0; }

   public void push(Item item)
   {  s[N++] = item;  }

   public String pop()
   {
      Item item = s[N-1];
      s[N-1] = null;
      N--;
      return item;
   }

}
```

the ugly cast

Number of casts in good code: 0

Generic data types: autoboxing

Generic stack implementation is object-based.

What to do about primitive types?

Wrapper type.

- Each primitive type has a wrapper object type.
- Ex: `Integer` is wrapper type for `int`.

Autoboxing. Automatic cast between a primitive type and its wrapper.

Syntactic sugar. Behind-the-scenes casting.

```
Stack<Integer> s = new Stack<Integer>();
s.push(17);         // s.push(new Integer(17));
int a = s.pop();    // int a = ((int) s.pop()).intValue();
```

Bottom line: Client code can use generic stack for any type of data

- stacks
- dynamic resizing
- queues
- generics
- **applications**

Stack Applications

Real world applications.

- Parsing in a compiler.
- Java virtual machine.
- Undo in a word processor.
- Back button in a Web browser.
- PostScript language for printers.
- Implementing function calls in a compiler.

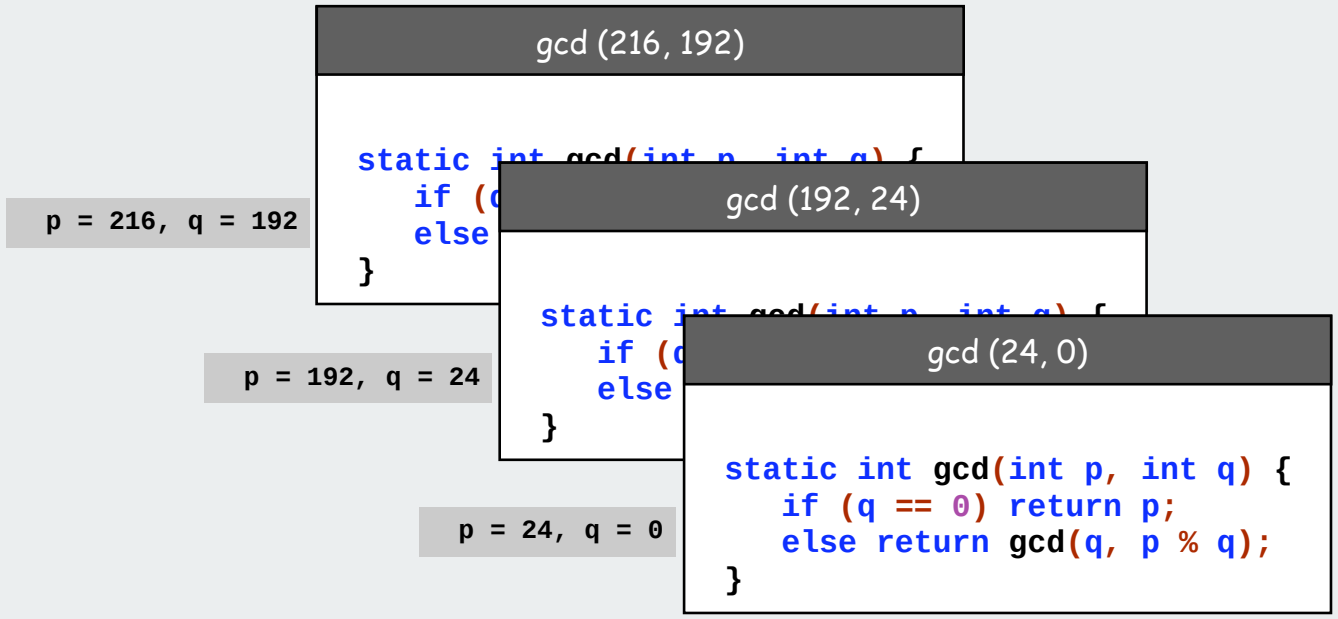
Function Calls

How a compiler implements functions.

- Function call: push local environment and return address.
- Return: pop return address and local environment.

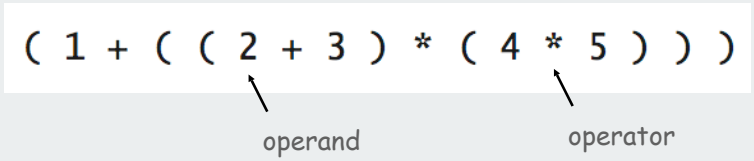
Recursive function. Function that calls itself.

Note. Can always use an explicit stack to remove recursion.



Arithmetic Expression Evaluation

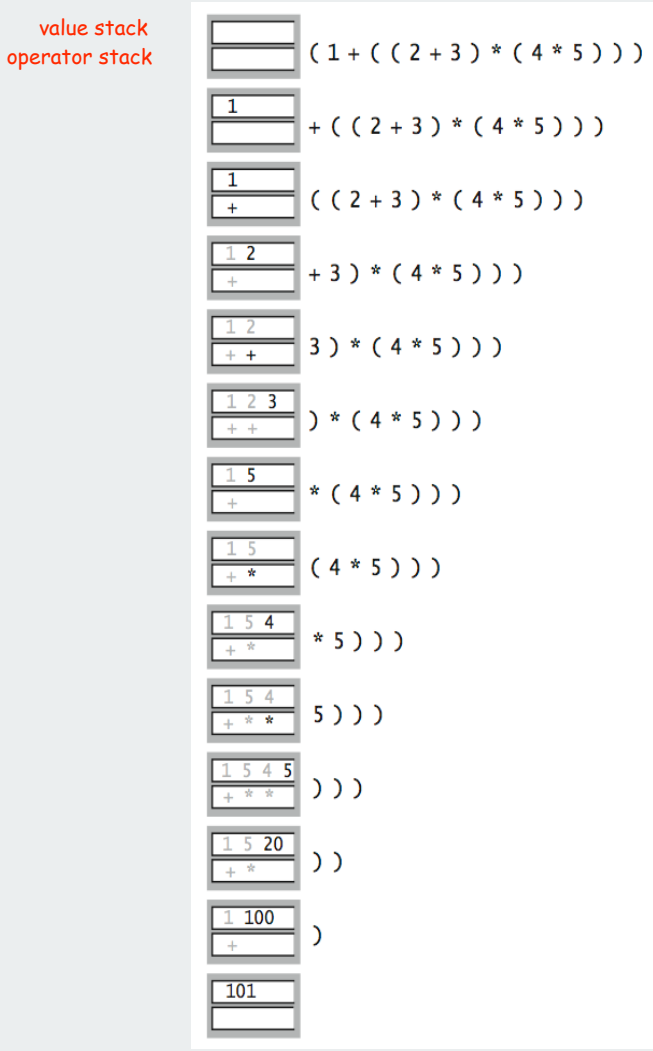
Goal. Evaluate infix expressions.



Two-stack algorithm. [E. W. Dijkstra]

- Value: push onto the value stack.
- Operator: push onto the operator stack.
- Left parens: ignore.
- Right parens: pop operator and two values; push the result of applying that operator to those values onto the operand stack.

Context. An interpreter!



Arithmetic Expression Evaluation

```
public class Evaluate {
   public static void main(String[] args) {
      Stack<String> ops  = new Stack<String>();
      Stack<Double> vals = new Stack<Double>();
      while (!StdIn.isEmpty()) {
         String s = StdIn.readString();
         if      (s.equals("("))               ;
         else if (s.equals("+"))    ops.push(s);
         else if (s.equals("*"))    ops.push(s);
         else if (s.equals(")")) {
            String op = ops.pop();
            if      (op.equals("+")) vals.push(vals.pop() + vals.pop());
            else if (op.equals("*")) vals.push(vals.pop() * vals.pop());
         }
         else vals.push(Double.parseDouble(s));
      }
      StdOut.println(vals.pop());
   }
}
```

```
% java Evaluate
( 1 + ( ( 2 + 3 ) * ( 4 * 5 ) ) )
101.0
```

Note: Old books have two-pass algorithm because generics were not available!

Correctness

Why correct?

When algorithm encounters an operator surrounded by two values within parentheses, it leaves the result on the value stack.

```
( 1 + ( ( 2 + 3 ) * ( 4 * 5 ) ) )
```

as if the original input were:

```
( 1 + ( 5 * ( 4 * 5 ) ) )
```

Repeating the argument:

```
( 1 + ( 5 * 20 ) )
( 1 + 100 )
101
```

Extensions. More ops, precedence order, associativity.

```
1 + (2 - 3 - 4) * 5 * sqrt(6 + 7)
```

Stack-based programming languages

Observation 1.

Remarkably, the 2-stack algorithm computes the same value if the operator occurs after the two values.

```
( 1 ( ( 2 3 + ) ( 4 5 * ) * ) + )
```

Observation 2.

All of the parentheses are redundant!

```
1 2 3 + 4 5 * * +
```

Jan Lukasiewicz

Bottom line. Postfix or "reverse Polish" notation.

Applications. Postscript, Forth, calculators, Java virtual machine, ...

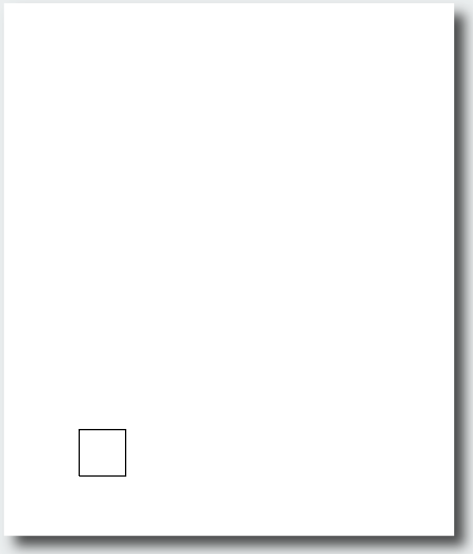
Stack-based programming languages: PostScript

Page description language

- explicit stack
- full computational model
- graphics engine

Basics

- %!: "I am a PostScript program"
- literal: "push me on the stack"
- function calls take args from stack
- turtle graphics built in



a PostScript program

```
%!
72 72 moveto
0 72 rlineto
72 0 rlineto
0 -72 rlineto
-72 0 rlineto
2 setlinewidth
stroke
```

Stack-based programming languages: PostScript

Data types

- basic: integer, floating point, boolean, ...
- graphics: font, path, ...
- full set of built-in operators

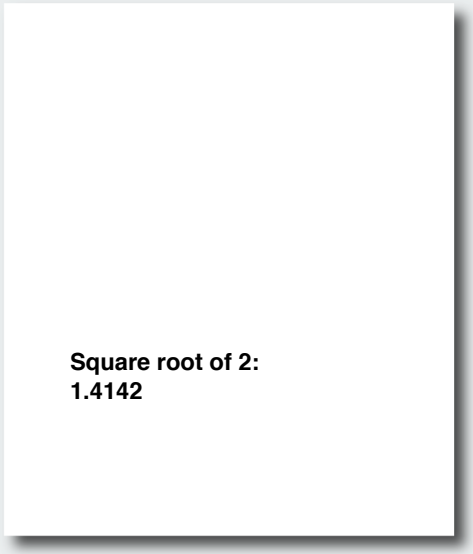
Text and strings

- full font support
- **show** (display a string, using current font)
- **cvs** (convert anything to a string)

like System.out.print()

like toString()

```
%!
/Helvetica-Bold findfont 16 scalefont setfont
72 168 moveto
(Square root of 2:) show
72 144 moveto
2 sqrt 10 string cvs show
```



Stack-based programming languages: PostScript

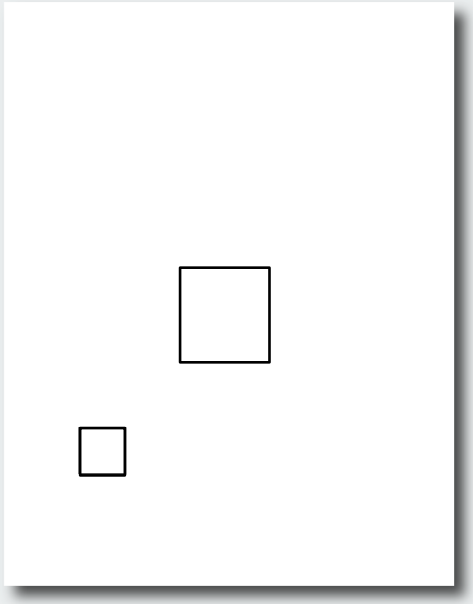
Variables (and functions)

- identifiers start with /
- def operator associates id with value
- braces
- args on stack

function definition →

function calls →

```
%!
/box
{
  /sz exch def
  0 sz rlineto
  sz 0 rlineto
  0 sz neg rlineto
  sz neg 0 rlineto
} def

72 144 moveto
72 box
288 288 moveto
144 box
2 setlinewidth
stroke
```

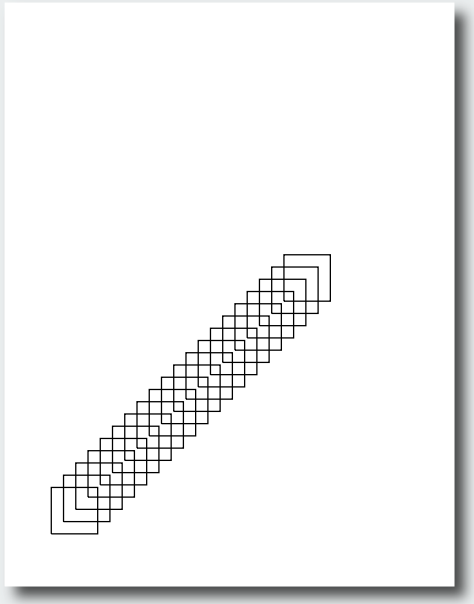


Stack-based programming languages: PostScript

for loop

- "from, increment, to" on stack
- loop body in braces
- **`for`** operator

```
1 1 20
{ 19 mul dup 2 add moveto 72 box }
for
```



if-else

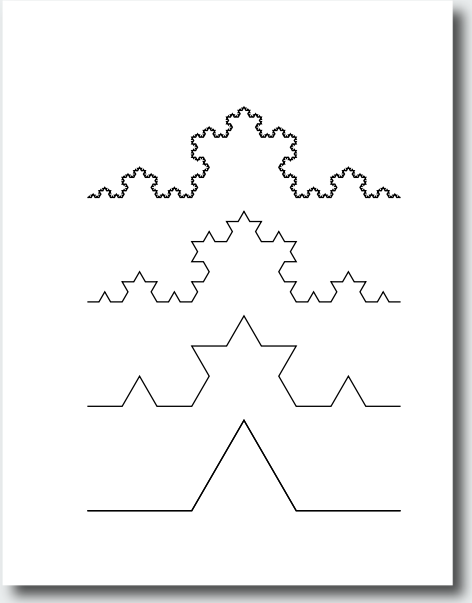
- boolean on stack
- alternatives in braces
- **`if`** operator

... (hundreds of operators)

Stack-based programming languages: PostScript

An application: all figures in Algorithms in Java

```
%!
72 72 translate

/kochR
  {
    2 copy ge { dup 0 rlineto }
      {
        3 div
        2 copy kochR 60 rotate
        2 copy kochR -120 rotate
        2 copy kochR 60 rotate
        2 copy kochR
      } ifelse
    pop pop
  } def

0   0 moveto   81 243 kochR
0  81 moveto   27 243 kochR
0 162 moveto    9 243 kochR
0 243 moveto    1 243 kochR
stroke
```



See page 218

Queue applications

Familiar applications.

- iTunes playlist.
- Data buffers (iPod, TiVo).
- Asynchronous data transfer (file IO, pipes, sockets).
- Dispensing requests on a shared resource (printer, processor).

Simulations of the real world.

- Traffic analysis.
- Waiting times of customers at call center.
- Determining number of cashiers to have at a supermarket.

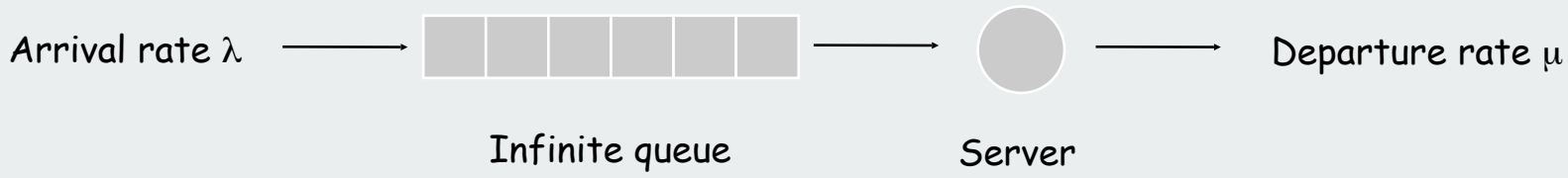
M/D/1 queuing model

M/D/1 queue.

- Customers are serviced at fixed rate of μ per minute.
- Customers arrive according to Poisson process at rate of λ per minute.

inter-arrival time has exponential distribution

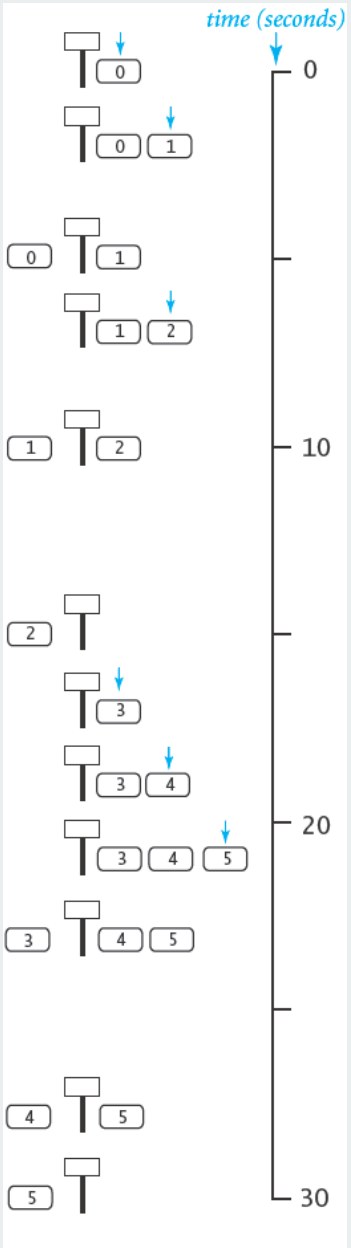
$$\Pr[X \le x] = 1 - e^{-\lambda x}$$



Q. What is average wait time W of a customer?

Q. What is average number of customers L in system?

M/D/1 queuing model: example



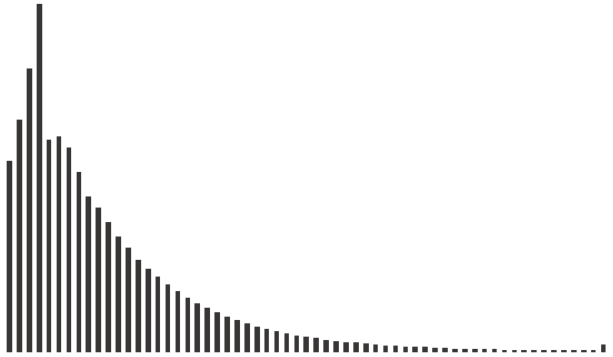
	arrival	departure	wait
0	0	5	5
1	2	10	8
2	7	15	8
3	17	23	6
4	19	28	9
5	21	30	9

M/D/1 queuing model: experiments and analysis

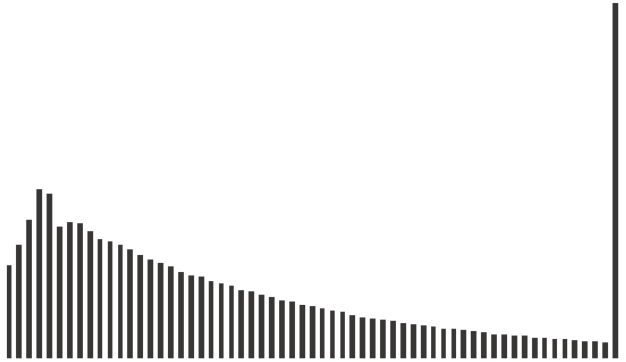
Observation.

As service rate μ approaches arrival rate λ, service goes to h***.

```
% java MD1Queue .167 .25
```



```
% java MD1Queue .167 .22
```



Queueing theory (see ORFE 309).

$$W = \frac{\lambda}{2\mu(\mu-\lambda)} + \frac{1}{\mu}, \quad L = \lambda W$$

Little's Law

wait time W and queue length L approach infinity as service rate approaches arrival rate

M/D/1 queuing model: event-based simulation

```
public class MD1Queue
{
   public static void main(String[] args)
   {
      double lambda = Double.parseDouble(args[0]);   // arrival rate
      double mu     = Double.parseDouble(args[1]);   // service rate
      Histogram hist = new Histogram(60);
      Queue<Double> q = new Queue<Double>();
      double nextArrival = StdRandom.exp(lambda);
      double nextService = 1/mu;
      while (true)
      {
         while (nextArrival < nextService)
         {
            q.enqueue(nextArrival);
            nextArrival += StdRandom.exp(lambda);
         }
         double wait = nextService - q.dequeue();
         hist.addDataPoint(Math.min(60,  (int) (wait)));
         if (!q.isEmpty())
            nextService = nextArrival + 1/mu;
         else
            nextService = nextService + 1/mu;
      }
   }
}
```