Stacks and Queues

- stacks
- dynamic resizing
- queues
- generics
- iterators
- applications

Reference: Algorithms in Java, Chapter 3, 4

Stacks and queues

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, union-find, ...

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

Client, Implementation, Interface

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

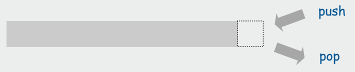
Client: program using operations defined in interface.
Implementation: actual code implementing operations.
Interface: description of data type, basic operations.

- stacks
- dynamic resizing
- queues
- generics
- iterators
- applications

Stacks

Stack operations.

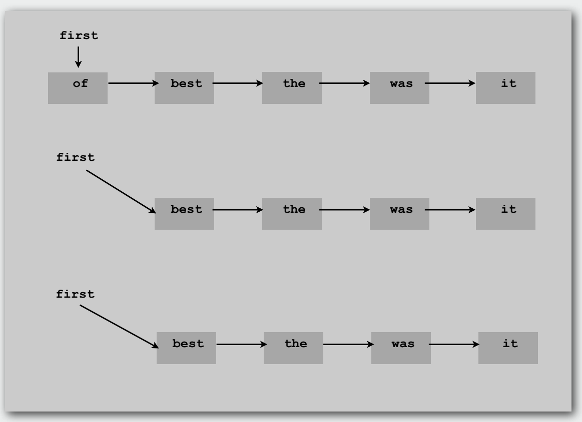
- `push()` Insert a new item onto stack.
- `pop()` Remove and return the item most recently added.
- `isEmpty()` Is the stack empty?




```
public static void main(String[] args)
{
   StackOfStrings stack = new StackOfStrings();
   while (!StdIn.isEmpty())
   {
      String item = StdIn.readString();
      if (item.equals("-")) StdOut.print(stack.pop());
      else                  stack.push(item);
   }
}
```

```
% more tobe.txt
to be or not to - be - - that - - - is

% java StackOfStrings < tobe.txt
to be not that or be
```


Stack pop: linked-list implementation

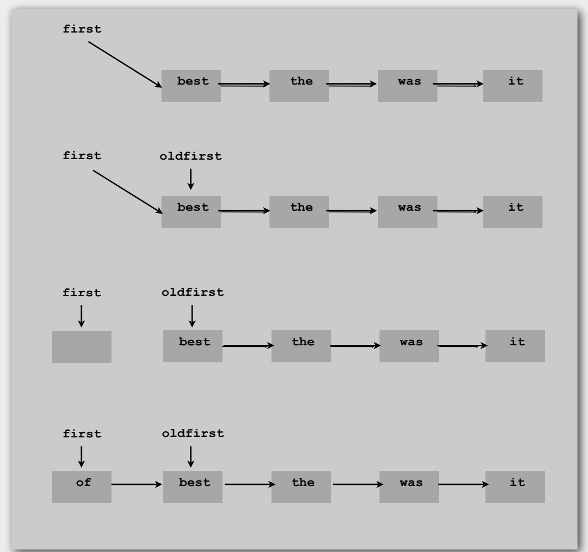


`String item = first.item;`

`first = first.next;`

`return item;`

Stack push: linked-list implementation



`Node oldfirst = first;`

`Node first = new Node();`

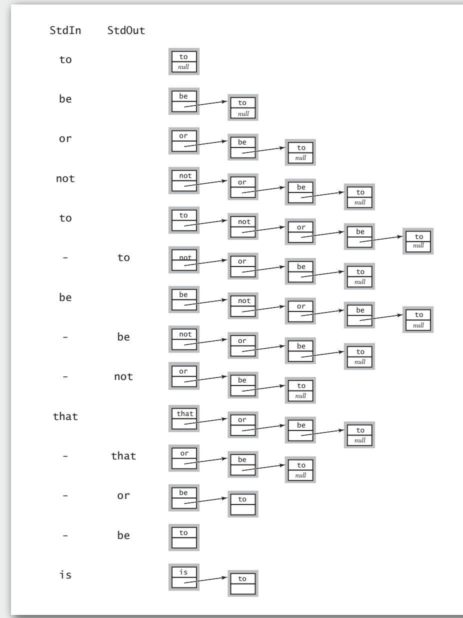
```
first.item = "of";
first.next = oldfirst;
```

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 oldfirst = first;
      first = new Node();
      first.item = item;
      first.next = oldfirst;
   }

   public String pop()
   {
      if (isEmpty()) throw new RuntimeException();
      String item = first.item;
      first = first.next;
      return item;
   }
}
```

"inner class"

Stack: linked-list trace



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]`.

s[]	it	was	the	best	of	times	*null*	*null*	*null*	*null*
	0	1	2	3	4	5	6	7	8	9
							N			capacity = 10

Stack: array implementation

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

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

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

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

   public String pop()
   {  return s[--N];  }
}
```

decrement N;
then use to index into array

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

this version avoids "loitering"

garbage collector only reclaims memory if no outstanding references

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

Stack: dynamic array implementation

Q. How to grow array?
Q. How to shrink array?

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

Too expensive.
- Need to copy all item to a new array.
- Inserting N items takes time proportional to 1 + 2 + … + N ~ $N^2/2$.

infeasible for large N

Goal. Ensure that array resizing happens infrequently.

Stack: dynamic array implementation

Q. How to grow array?
A. If array is full, create a new array of twice the size, and copy items.

"repeated doubling"

```
public StackOfStrings() {  s = new String[2];  }

public void push(String item)
{
   if (N == s.length) resize(2 * s.length);
   s[N++] = item;
}

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

1 + 2 + 4 + … + N/2 + N ~ 2N

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

Stack: dynamic array implementation

Q. How to shrink array?

First try.
- `push()`: double size of `s[]` when array is full.
- `pop()`: halve size of `s[]` when array is half full.

Too expensive
- Consider push-pop-push-pop-… sequence when array is full.
- Time proportional to N per operation.

"thrashing"

N = 5	it	was	the	best	of	*null*	*null*	*null*
N = 4	it	was	the	best				
N = 5	it	was	the	best	of	*null*	*null*	*null*
N = 4	it	was	the	best				

Stack: dynamic array implementation

Q. How to shrink array?

Efficient solution.

- `push()`: double size of `s[]` when array is full.
- `pop()`: halve size of `s[]` when array is one-quarter full.

```
public String pop()
{
   String item = s[N-1];
   s[N-1] = null;
   N--;
   if (N > 0 && N == s.length/4) resize(s.length / 2);
   s[N++] = item;
   return item;
}
```

Invariant. Array is always between 25% and 100% full.

Stack: dynamic array implementation trace

StdIn	StdOut	N	a.length	a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]
		0	1	*null*							
to		1	1	to							
be		2	2	to	be						
or		3	4	to	be	or	*null*				
not		4	4	to	be	or	not				
to		5	8	to	be	or	not	to	*null*	*null*	*null*
-	to	4	8	to	be	or	not	*null*	*null*	*null*	*null*
be		5	8	to	be	or	not	be	*null*	*null*	*null*
-	be	4	8	to	be	or	not	*null*	*null*	*null*	*null*
-	not	3	8	to	be	or	*null*	*null*	*null*	*null*	*null*
that		4	8	to	be	or	that	*null*	*null*	*null*	*null*
-	that	3	8	to	be	or	*null*	*null*	*null*	*null*	*null*
-	or	2	4	to	be	*null*	*null*				
-	be	1	2	to	*null*						
is		2	2	to	is						

Amortized analysis

Amortized analysis. Average running time per operation over a worst-case sequence of operations.

Proposition. Starting from empty data structure, any sequence of M ops takes time proportional to M.

running time for doubling stack with N elements

	worst	best	amortized
construct	1	1	1
push	N	1	1
pop	N	1	1

doubling or shrinking

Remark. WQUPC used amortized bound: starting from empty data structure, any sequence of M union and find ops takes O((M+N) log* N) time.

Stack implementations: dynamic array vs. linked List

Tradeoffs. Can implement with either array or linked list; client can use interchangeably. Which is better?

Linked list.

- Every operation takes constant time in worst-case.
- Uses extra time and space to deal with the links.

Array.

- Every operation takes constant amortized time.
- Less wasted space.

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

Queues

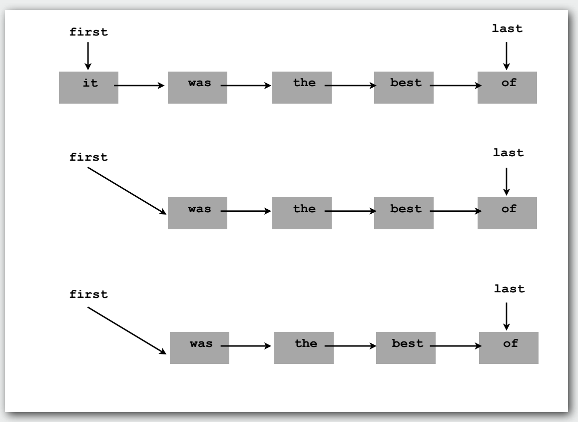
Queue operations.

- `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();
   while (!StdIn.isEmpty())
   {
      String item = StdIn.readString();
      if (item.equals("-")) StdOut.print(q.dequeue());
      else                  q.enqueue(item);
      else
   }
}
```

```
% more tobe.txt
to be or not to - be - - that - - - is

% java QueueOfStrings < tobe.txt
to be or not to be
```


Queue dequeue: linked list implementation

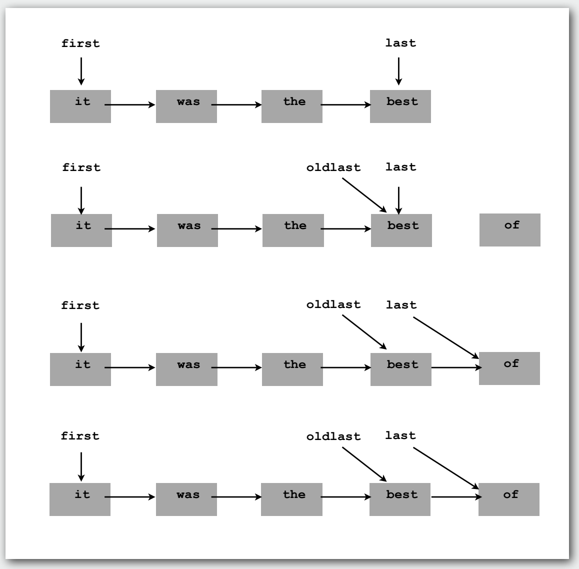



```
String item = first.item;
```

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

```
return item;
```


Queue enqueue: linked list implementation




```
Node oldlast = last;
```

```
Node last  = new Node();
last.item = "of";
last.next = null;
```

```
oldlast.next = last;
```

Queue: linked list implementation

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

   private class Node
   { String item; Node next; }

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

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

   public String dequeue()
   {
      String item = first.item;
      first       = first.next;
      if (isEmpty()) last = null;
      return item;
   }
}
```


Queue: dynamic array implementation

Array implementation of a queue.

- Use array `q[]` to store items in queue.
- `enqueue()`: add new item at `q[tail]`.
- `dequeue()`: remove item from `q[head]`.
- Update `head` and `tail` modulo the `capacity`.
- Add repeated doubling and shrinking.

q[]	*null*	*null*	the	best	of	times	*null*	*null*	*null*	*null*
	0	1	2	3	4	5	6	7	8	9
			head				tail			capacity = 10

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

Parameterized stack

We implemented: `StackOfStrings`.
We also want: `StackOfURLs`, `StackOfCustomers`, `StackOfInts`, 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]

Parameterized stack

We implemented: `StackOfStrings`.

We also want: `StackOfURLs`, `StackOfCustomers`, `StackOfInts`, 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.

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

run-time error

Parameterized stack

We implemented: `StackOfStrings`.

We also want: `StackOfURLs`, `StackOfCustomers`, `StackOfInts`, etc?

Attempt 3. Java generics.

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

type parameter

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

compile-time error

Guiding principles. Welcome compile-time errors; avoid run-time errors.

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 oldfirst = first;
      first = new Node();
      first.item = item;
      first.next = oldfirst;
   }

   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 oldfirst = first;
      first = new Node();
      first.item = item;
      first.next = oldfirst;
   }

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

generic type name

Generic stack: array implementation

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

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

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

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

   public String pop()
   {  return s[--N];  }
}
```

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

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

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

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

   public Item pop()
   {  return s[--N];  }
}
```

the way it should be

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

Generic stack: array implementation

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

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

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

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

   public String pop()
   {  return s[--N];  }
}
```

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

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

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

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

   public Item pop()
   {  return s[--N];  }
}
```

the way it is

the ugly cast

Generic data types: autoboxing

Q. 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.

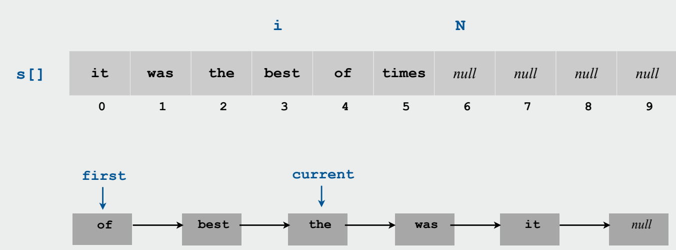
Syntactic sugar. Behind-the-scenes casting.

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

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

Iteration

Design challenge. Support iteration over stack items by client, without revealing the internal representation of the stack.



Java solution. Make stack `Iterable`.

Iterators

Q. What is an `Iterable` ?
A. Has a method that returns an `Iterator`.

```
public interface Iterable<Item>
{
   Iterator<Item> iterator();
}
```

Q. What is an `Iterator` ?
A. Has methods `hasNext()` and `next()`.

```
public interface Iterator<Item>
{
   boolean hasNext();
   Item next();
   void remove();   ← optional; use at your own risk
}
```

Q. Why make data structures `Iterable` ?
A. Java supports elegant client code.

"foreach" statement

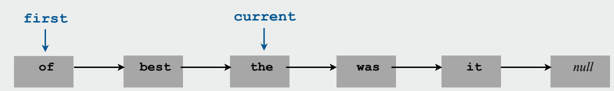
```
for (String s : stack)
   StdOut.println(s);
```

equivalent code

```
Iterator<String> i = stack.iterator();
while (i.hasNext())
{
   String s = i.next();
   StdOut.println(s);
}
```


Stack iterator: linked list implementation

```
import java.util.Iterator;

public class Stack<Item> implements Iterable<Item>
{
    ...

    public Iterator<Item> iterator() { return new ListIterator(); }

    private class ListIterator implements Iterator<Item>
    {
        private Node current = first;

        public boolean hasNext() {  return current != null;  }
        public void remove()     {  /* not supported */      }
        public Item next()
        {
            Item item = current.item;
            current   = current.next;
            return item;
        }
    }
}
```



Stack iterator: array implementation

```
import java.util.Iterator;

public class Stack<Item> implements Iterable<Item>
{
    …

    public Iterator<Item> iterator() { return new ArrayIterator(); }

    private class ArrayIterator implements Iterator<Item>
    {
        private int i = N;

        public boolean hasNext() {  return i > 0;        }
        public void remove()     {  /* not supported */  }
        public Item next()       {  return s[--i];       }
    }
}
```

					i			N			
s[]	it	was	the	best	of	times	*null*	*null*	*null*	*null*
	0	1	2	3	4	5	6	7	8	9

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.

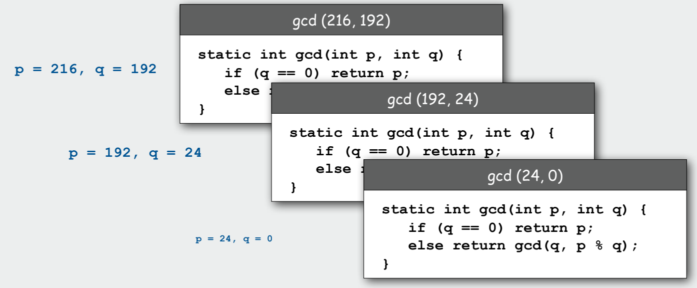
Function calls

How a compiler implements a function.

- 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.

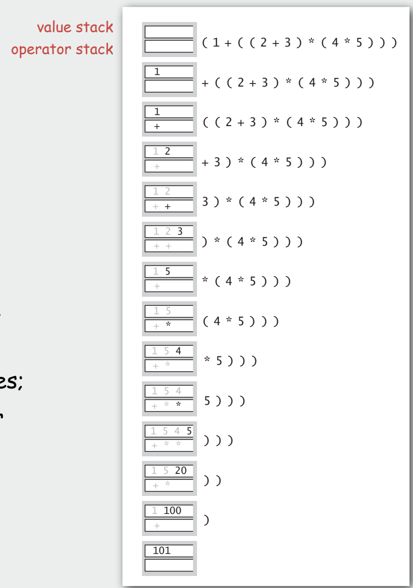
`( 1 + ( ( 2 + 3 ) * ( 4 * 5 ) ) )`

operand   operator

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
```

Correctness

Q. Why correct?

A. 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.

Stack-based programming languages

Observation 1. 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, ...

PostScript

Page description language.

- Explicit stack.
- Full computational model
- Graphics engine.

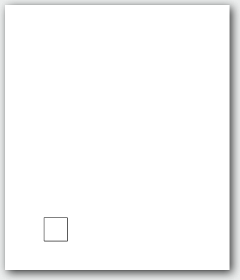
Basics.

- `%!`: "I am a PostScript program."
- Literal: "push me on the stack."
- Function calls take arguments 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
```

its output



PostScript

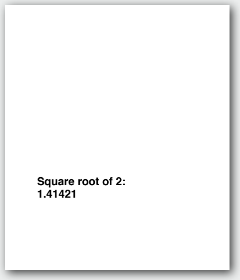
Data types.

- basic: integer, floating point, boolean, ...
- Graphics: font, path, curve, ...
- Full set of built-in operators.

Text and strings.

- Full font support.
- `show` (display a string, using current font). ← `System.out.print()`
- `cvs` (convert anything to a string). ← `toString()`

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



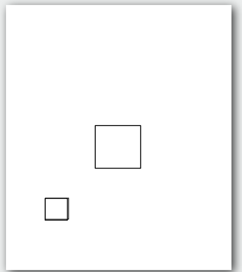
Variables (and functions).

- Identifiers start with /.
- def operator associates id with value.
- Braces.
- args on stack.

function definition

```
%!
/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
```

function calls



For loop.

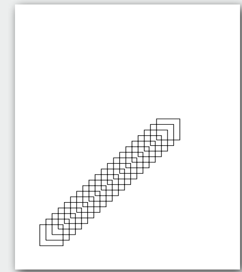
- "from, increment, to" on stack.
- Loop body in braces.
- `for` operator.

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

If-else conditional.

- Boolean on stack.
- Alternatives in braces.
- `if` operator.

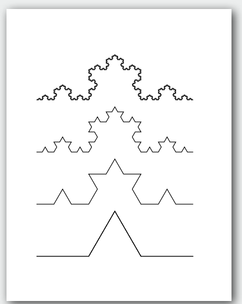
... (hundreds of operators)



Application 1. All figures in Algorithms in Java

Application 2. Deluxe version of `StdDraw` also saves to PostScript for vector graphics.

```
%!
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.

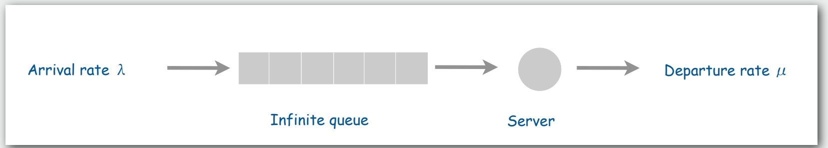
- Traffic analysis.
- Waiting times of customers at call center.
- Determining number of cashiers to have at a supermarket.

M/M/1 queuing model

M/M/1 queue.

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

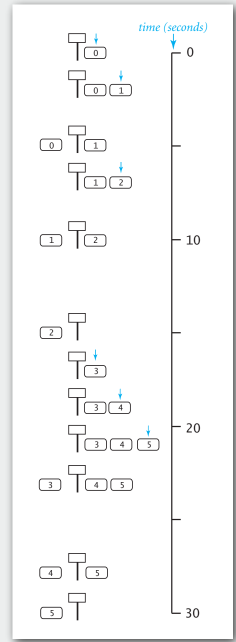
interarrival time has exponential distribution $\Pr[X \le x] = 1 - e^{-\lambda x}$
service time has exponential distribution $\Pr[X \le x] = 1 - e^{-\mu x}$



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

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

M/M/1 queuing model: example simulation



	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

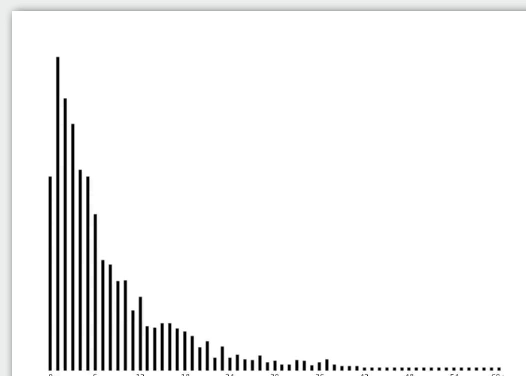
M/M/1 queuing model: event-based simulation

```
public class MM1Queue
{
    public static void main(String[] args) {
        double lambda = Double.parseDouble(args[0]);   // arrival rate
        double mu     = Double.parseDouble(args[1]);   // service rate
        double nextArrival = StdRandom.exp(lambda);
        double nextService = nextArrival + StdRandom.exp(mu);

        Queue<Double> queue = new Queue<Double>();
        Histogram hist = new Histogram("M/D/1 Queue", 60);

        while (true)
        {
            // next event is an arrival
            while (nextArrival < nextService)
            {
                queue.enqueue(nextArrival);
                nextArrival += StdRandom.exp(lambda);
            }

            // next event is a service completion
            double arrival = queue.dequeue();
            double wait = nextService - arrival;
            hist.addDataPoint(Math.min(60,  (int)  (Math.round(wait))));
            if (queue.isEmpty()) nextService = nextArrival + StdRandom.exp(mu);
            else                 nextService = nextService + StdRandom.exp(mu);
        }
    }
}
```


M/M/1 queuing model: experiments

Observation. If service rate μ is much larger than arrival rate λ, customers gets good service.

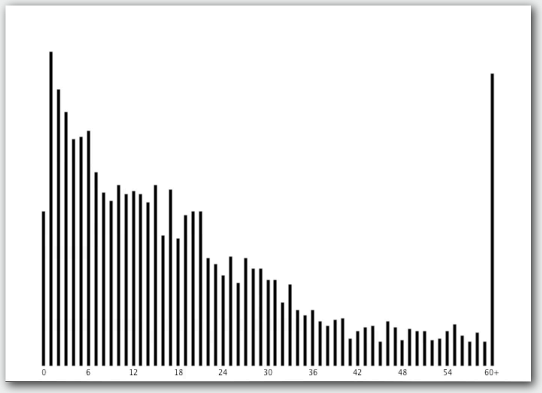
```
% java MM1Queue .2 .333
```



M/M/1 queuing model: experiments

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

```
% java MM1Queue .2 .25
```



M/M/1 queuing model: experiments

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

```
% java MM1Queue .2 .21
```

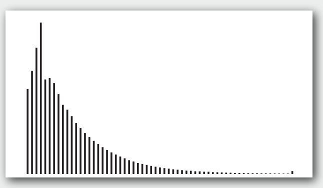
M/M/1 queuing model: analysis

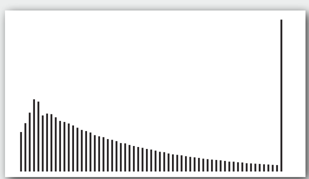
M/M/1 queue. Exact formulas known.

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

Little's Law

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





More complicated queueing models. Event-based simulation essential!

Queueing theory. See ORFE 309.