Lecture 4-2

Arrays, Part II

How Java Manages Arrays and Strings:

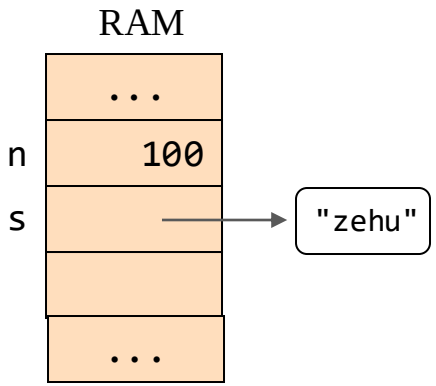
A Low-Level Perspective

How Java handles strings

high-level abstraction

```
...
int n = 100;
String s = "zehu";
```

low-level implementation

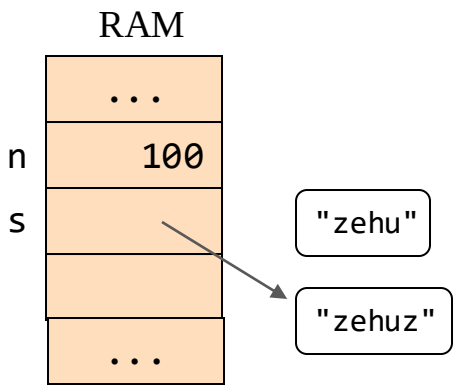


How Java handles strings

high-level abstraction

```
...
int n = 100;
String s = "zehu";
s = s + 'z';
```

low-level implementation

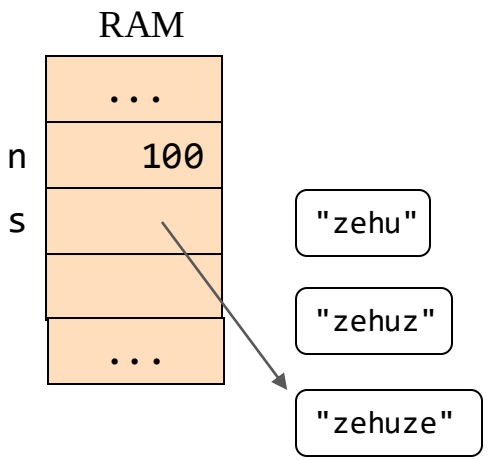


How Java handles strings

high-level abstraction

```
...
int n = 100;
String s = "zehu";
s = s + 'z';
s = s + 'e';
```

low-level implementation

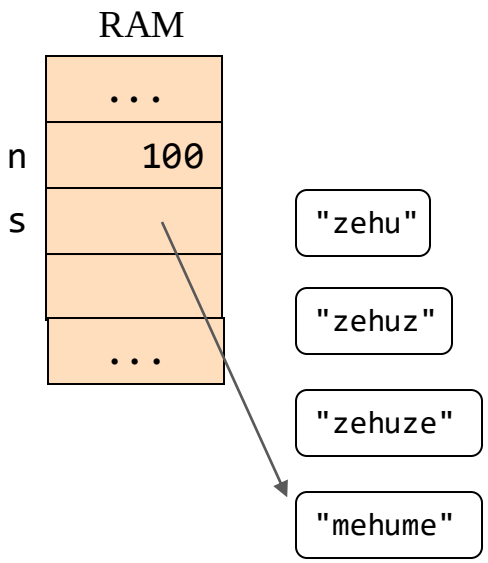


How Java handles strings

high-level abstraction

```
...
int n = 100;
String s = "zehu";
s = s + 'z';
s = s + 'e';
System.out.println(s); // zehuze
s = s.replace('z', 'm');
System.out.println(s); // mehume
...
```

low-level implementation

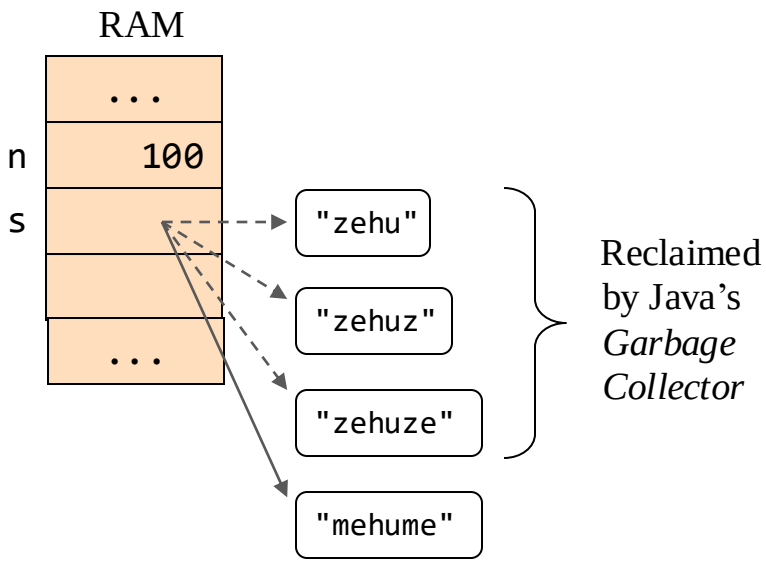


How Java handles strings

high-level abstraction

```
...
int n = 100;
String s = "zehu";
s = s + 'z';
s = s + 'e';
System.out.println(s); // zehuze
s = s.replace('z', 'm');
System.out.println(s); // mehume
...
```

low-level implementation



Garbage Collector

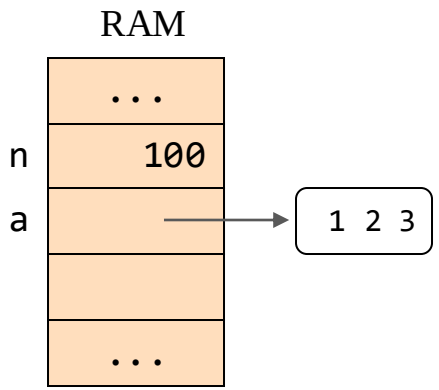
- A process that runs in the background (part of Java's runtime system)
- Collects "orphan" objects (arrays, strings, …) that have no variables pointing at them
- Recycles the memory held by these objects.

How Java handles arrays

high-level abstraction

```
...
int n = 100;
int[] a = {1, 2, 3};




// Returns the elements of arr, reversed
public static int[] reverse(int[] arr) {
  // Code omitted
}
...
```

low-level implementation

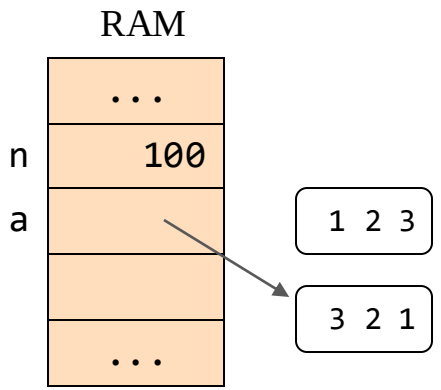


How Java handles arrays

high-level abstraction

```
...
int n = 100;
int[] a = {1, 2, 3};
a = reverse(a);




// Returns the elements of arr, reversed
public static int[] reverse(int[] arr) {
  // Code omitted
}
...
```

low-level implementation

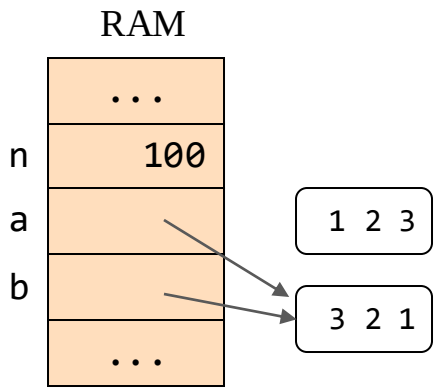


How Java handles arrays

high-level abstraction

```
...
int n = 100;
int[] a = {1, 2, 3};
a = reverse(a);
...
int[] b = a;



// Returns the elements of arr, reversed
public static int[] reverse(int[] arr) {
  // Code omitted
}
...
```

low-level implementation

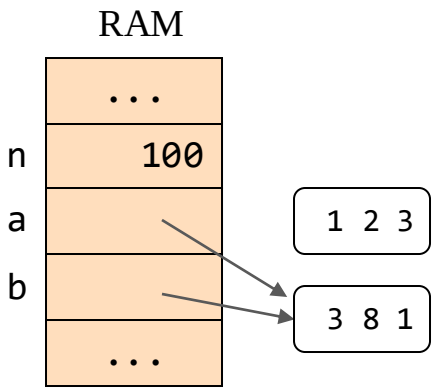


How Java handles arrays

high-level abstraction

```
...
int n = 100;
int[] a = {1, 2, 3};
a = reverse(a);
...
int[] b = a;
b[1] = 8;


// Returns the elements of arr, reversed
public static int[] reverse(int[] arr) {
  // Code omitted
}
...
```

low-level implementation



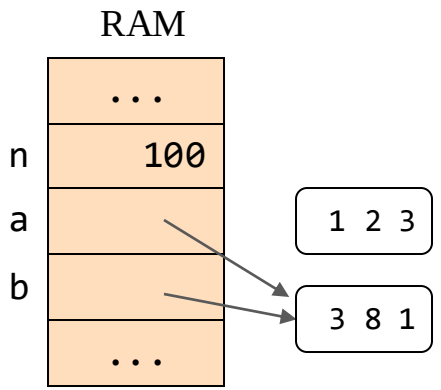
How Java handles arrays

high-level abstraction

```
...
int n = 100;
int[] a = {1, 2, 3};
a = reverse(a);
...
int[] b = a;
b[1] = 8;
System.out.println(a[1]); // 8
...
// Returns the elements of arr, reversed
public static int[] reverse(int[] arr) {
  // Code omitted
}
...
```

dangerous code

low-level implementation

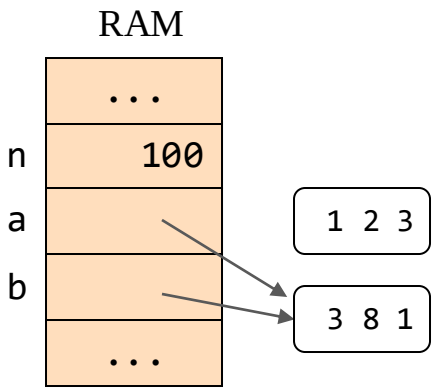


How Java handles arrays

high-level abstraction

```
...
int n = 100;
int[] a = {1, 2, 3};
a = reverse(a);
...
int[] b = a;
b[1] = 8;
System.out.println(a[1]); // 8
...
// Returns the elements of arr, reversed
public static int[] reverse(int[] arr) {
  // Code omitted
}
...
```

low-level implementation



Recap

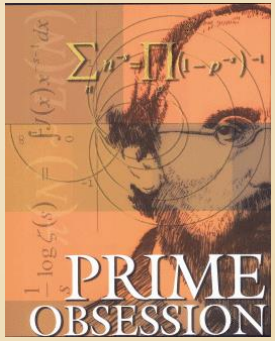
String variables and array variables are *references* (contain addresses)

- Arrays: Changes to the array elements remain "in place" (within the same memory block)
- String: Changes to the string result in creating a new string (a new memory block).

Introduction to Computer Science
Shimon Schocken
IDC Herzliya

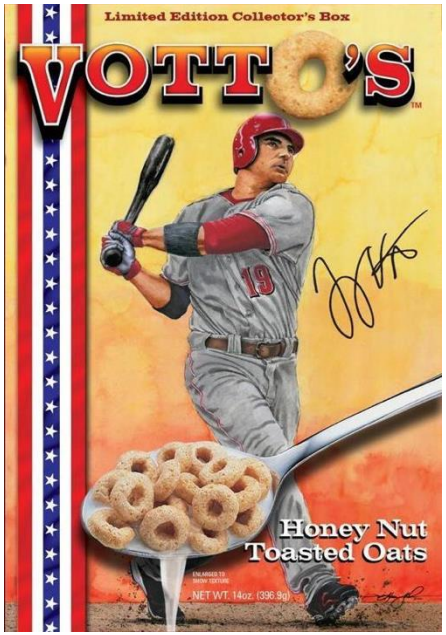
Lecture 4-2

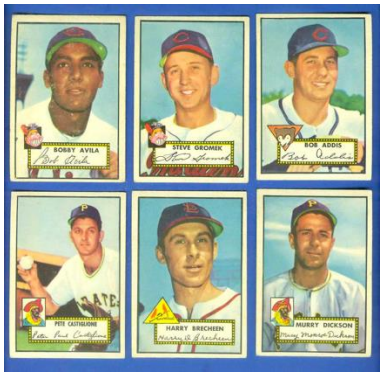
Arrays, Part III



Arrays in action: Application examples

Application example: Coupon collector problem





Example

- There are 100 different baseball cards.
- Each cereal box contains one card.

How many cereal boxes do you have to buy until you collect all 100 cards?

Coupon collector problem

How many times do you have to draw random integers between 0 and *N*-1 until every integer has been drawn at least once?

N

```
% java CouponCollector 100
587

% java CouponCollector 100
538

% java CouponCollector 100
580
```

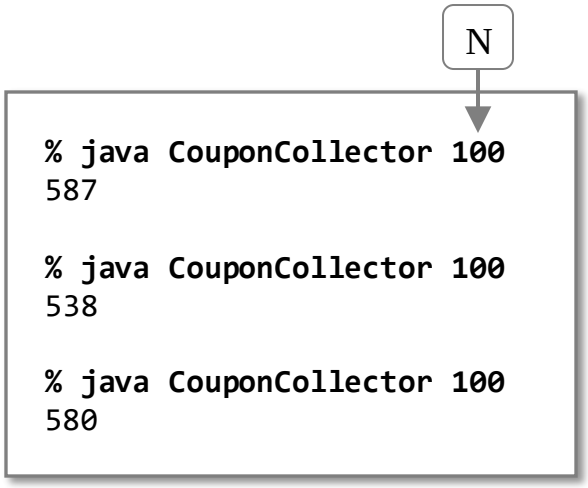
Coupon collector problem

Algorithm

```
found = boolean array[0, ..., N-1]   // all false
count = 0
nDistinct = 0  // Counts how many distinct values where drawn so far
while nDistinct < N
    draw a random int r from 0 to N-1
    count++
    if found[r] is false:
        found[r] = true
        nDistinct++
return count
// And hope that the loop will terminate…
```

Coupon collector problem

How many times do you have to draw random integers between 0 and *N*-1 until every integer has been drawn at least once?



Coupon collector problem: implementation

```
// Computes How many times you have to draw random integers between
// 0 and N-1 until all integers 0,1,2 ,…, N-1 have been drawn
public class CouponCollector {
   public static void main(String[] args) {
      int N = Integer.parseInt(args[0]);
      int count = 0;      // number of values drawn so far
      int nDistinct = 0; // number of distinct values drawn so far

      boolean[] found = new boolean[N];
      for (int i = 0; i < N; i++) found[i] = false;

      // Runs the simulation (note: this loop may never terminate)
      while (nDistinct < N) {
         // Draws a random integer between 0 and N-1, and updates the relevant counters
         int val = (int) (Math.random() * N);
         count++;
         if (found[val] == false) {
            found[val] = true;
            nDistinct++;
         }
      }

      // All the numbers between 0 and N-1 have been collected!
      System.out.println(count);
   }
}
```

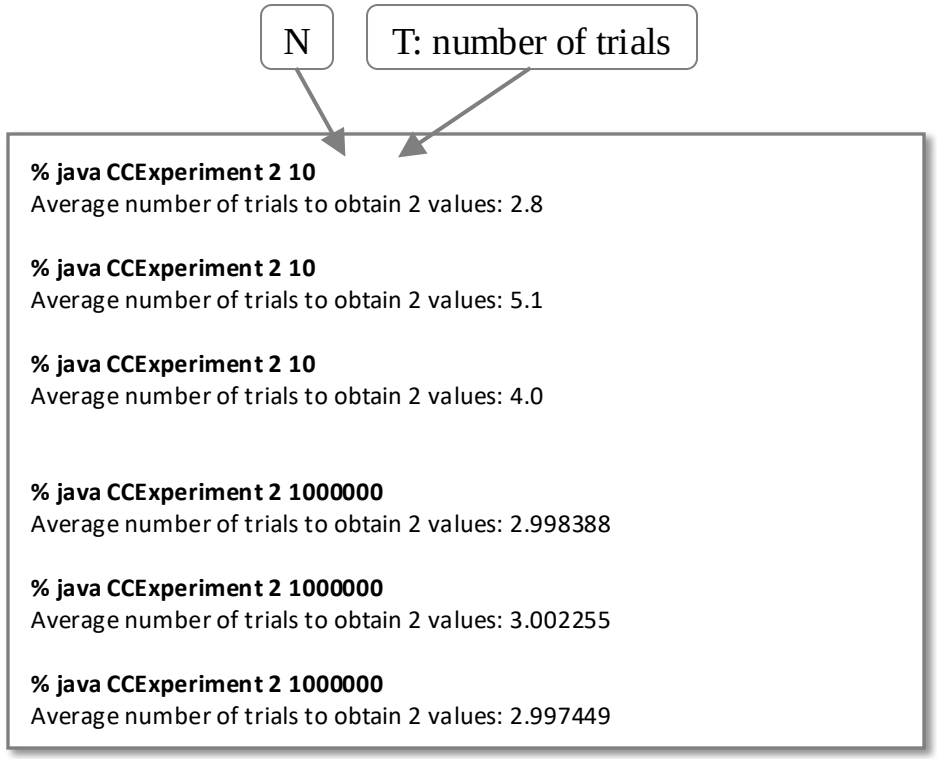
N

```
% java CouponCollector 100
587

% java CouponCollector 100
538

% java CouponCollector 100
580
```

Coupon collector: Experiment

N = 10,
high variance

N = 100000,
low variance



Example of the "Law of Large Numbers"

Coupon collector: Experiment

```
public class CCExperiment {

   // Computes how many times you have to draw random integers between 0 and N-1 until every integer has been drawn
   public static int couponCollector(int N) {
      int count = 0;       // number of values drawn so far
      int nDistinct = 0;  // number of distinct values drawn so far
      boolean[] found = new boolean[N];
      for (int i=0; i<N; i++) found[i] = false;
      // Runs the simulation
      while (nDistinct < N) {
         // Draws a random integer between 0 and N-1 and updates
         //  the relevant counters
         int val = (int) (Math.random() * N);
         count++;
         if (!found[val]) {
             nDistinct++;
             found[val] = true;
         }
      }
      // All the values between 0 and N-1 have been collected!
      return count;
}
```

% java CCExperiment 2 10 // 10 = number of trials
Average number of trials to obtain 2 values: 2.8

% java CCExperiment 2 10
Average number of trials to obtain 2 values: 5.1

% java CCExperiment 2 10
Average number of trials to obtain 2 values: 4.0

% java CCExperiment 2 1000000
Average number of trials to obtain 2 values: 2.998388

% java CCExperiment 2 1000000
Average number of trials to obtain 2 values: 3.002255

% java CCExperiment 2 1000000
Average number of trials to obtain 2 values: 2.997449

Coupon collector: Experiment

```
public class CCExperiment {
   public static void main(String[] args) {
      final int N = Integer.parseInt(args[0]);   // number of values ("final": cannot be mutated)
      final int T = Integer.parseInt(args[1]);   // number of trials
      double sum = 0;
      for (int t = 0; t < T; t++) {
         sum = sum + couponCollector(N);
      }
      System.out.println("Average number of trials to obtain " + N + " values: " + sum / T );
   }
   // Computes how many times you have to draw random integers between 0 and N-1 until every integer has been drawn
   public static int couponCollector(int N) {
      int count = 0;      // number of values drawn so far
      int nDistinct = 0;  // number of distinct values drawn so far
      boolean[] found = new boolean[N];
      for (int i=0; i<N; i++) found[i] = false;
      // Runs the simulation
      while (nDistinct < N) {
         // Draws a random integer between 0 and N-1 and updates
         //  the relevant counters
         int val = (int) (Math.random() * N);
         count++;
         if (!found[val]) {
             nDistinct++;
             found[val] = true;
         }
      }
      // All the values between 0 and N-1 have been collected
      return count;
}
```

% java CCExperiment 2 10 // 10 = number of trials
Average number of trials to obtain 2 values: 2.8

% java CCExperiment 2 10
Average number of trials to obtain 2 values: 5.1

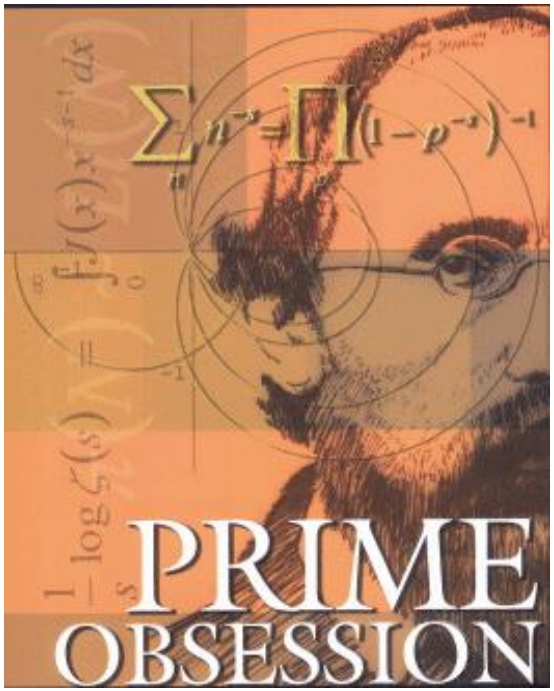
% java CCExperiment 2 10
Average number of trials to obtain 2 values: 4.0

% java CCExperiment 2 1000000
Average number of trials to obtain 2 values: 2.998388

% java CCExperiment 2 1000000
Average number of trials to obtain 2 values: 3.002255

% java CCExperiment 2 1000000
Average number of trials to obtain 2 values: 2.997449

Application example: Prime Numbers



Prime numbers

Prime: an integer > 1 whose only divisors (aka *factors*) are 1 and itself.

2, 3, 5, 7, 11, 13, 17, ..., 1013, ..., 2398120761, ..., ?

October 12, 2024

Luke Durant finds the largest known prime number: $2^{136279841} - 1$,

(a number that has 41,024,320 digits). Spent $2M on the project.

The thinning of the primes

Prime: an integer > 1 whose only divisors (aka *factors*) are 1 and itself.

The thinning of the primes

- The primes between 1 and 20: 2, 3, 5, 7, 11, 13, 17
- The primes between 980 and 1000: 983, 991, 997
- The primes between 9980 and 10000: none

Observation: As we go further along the number line,
the primes seem to *thin out* (though they never run out)

Proof: One of the most important open problems in mathematics

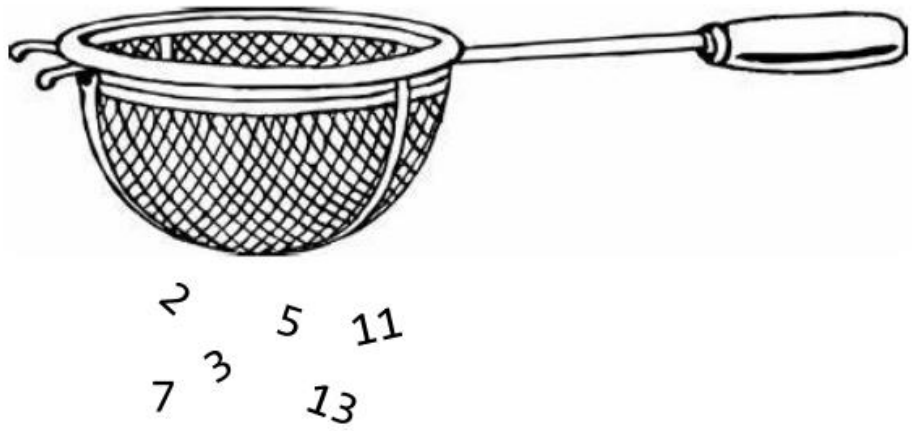
Illustration: Next few slides

Sieve of Eratosthenes: Algorithm

An algorithm for finding all the primes numbers up to a given number *n*

Discovered 2,200 years ago, still being used by modern computers

Well known for its simplicity and efficiency.



Sieve of Eratosthenes: Algorithm (~200 BCE)

initialize: build a Boolean array of size $n + 1$ and set all the elements with index > 1 to `true`:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	
isPrime	F	F	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	...

Cross out all the multiples of p = 2:

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	
isPrime	F	F	T	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	F	T	...

Cross out all the multiples of the next prime p = 3 (the next index which was not crossed out already):

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	
isPrime	F	F	T	T	F	T	F	T	F	F	F	T	F	T	F	F	F	T	F	T	...

Cross out all the multiples of the next prime p = 5 (the next index which was not crossed out already):

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	
isPrime	F	F	T	T	F	T	F	T	F	F	F	T	F	T	F	F	F	T	F	T	...

Keep incrementing p (skipping indexes that were already crossed) until p > sqrt(n)

When done: the indexes of all the surviving `true` values are primes:

			2	3		5		7				11		13				17		19	
isPrime	F	F	T	T	F	T	F	T	F	F	F	T	F	T	F	F	F	T	F	T	...

Optimization note: We stop at $\sqrt{n}$ because of the following observation:
if $x = a \times b$, then $a \leq \sqrt{x} \leq$ b .

Sieve of Eratosthenes: Implementation

```
public class Primes {
    public static void main(String[] args) {
        final int N = Integer.parseInt(args[0]);
        // Put your code here
    }
}
```

```
% java Primes 25
2
3
5
7
11
13
17
19
23
There are 9 primes between 2 and 25. (36% are primes)
```

The thinning of the primes

Proposition

As we go further along the number line, the primes thin out (become more and more rare)

```
% java Primes 100
There are 25 primes between 2 and 100. (25% are primes)

% java Primes 100000
There are 9592 primes between 2 and 100000. (9% are primes)

% java Primes 1000000000
There are 50847534 primes between 2 and 1000000000. (5% are primes)
```

Conclusion

The experiment seems to support the proposition.

Application example: Shuffling

Tasks

- Building a deck of cards
- Shuffling a deck of cards

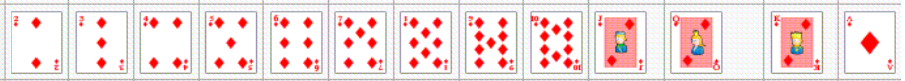
Building a deck of cards

```
public class Deck {
   public static void main(String[] args) {
      // Builds a deck of cards
      String[] rank = {"2", "3", "4", "5", "6", "7", "8", "9",
                       "10", "Jack", "Queen", "King", "Ace"};
      String[] suit = {"Clubs", "Diamonds", "Hearts", "Spades"};
      ...
```

```
% java Deck
2 of Clubs
2 of Diamonds
2 of Hearts
2 of Spades
3 of Clubs
3 of Diamonds
3 of Hearts
3 of Spades
4 of Clubs
4 of Diamonds
4 of Hearts
4 of Spades
5 of Clubs
5 of Diamonds
...
Ace of Clubs
Ace of Diamonds
Ace of Hearts
Ace of Spades

4 of Hearts
```

Builds and prints a deck of cards,
then selects and prints a random card

rank

suit	0	1	2	3	4	...
	2	3	4	5	6	7	8	9	10	Jack	Queen	King	Ace
0	Clubs												
1	Diamonds												
2	Hearts												
3	Spades												

Building a deck of cards

```
public class Deck {
   public static void main(String[] args) {
      // Builds a deck of cards
      String[] rank = {"2", "3", "4", "5", "6", "7", "8", "9",
                       "10", "Jack", "Queen", "King", "Ace"};
      String[] suit = {"Clubs", "Diamonds", "Hearts", "Spades"};

      String[] deck = new String[52];
      for (int i = 0; i < rank.length; i++)
          for (int j = 0; j < suit.length; j++)
               deck[4 * i + j] = rank[i] + " of " + suit[j];
```

```
% java Deck
2 of Clubs
2 of Diamonds
2 of Hearts
2 of Spades
3 of Clubs
3 of Diamonds
3 of Hearts
3 of Spades
4 of Clubs
4 of Diamonds
4 of Hearts
4 of Spades
5 of Clubs
5 of Diamonds
...
Ace of Clubs
Ace of Diamonds
Ace of Hearts
Ace of Spades

4 of Hearts
```

Building a deck of cards

```
public class Deck {
   public static void main(String[] args) {
      // Builds a deck of cards
      String[] rank = {"2", "3", "4", "5", "6", "7", "8", "9",
                       "10", "Jack", "Queen", "King", "Ace"};
      String[] suit = {"Clubs", "Diamonds", "Hearts", "Spades"};

      String[] deck = new String[52];
      for (int i = 0; i < rank.length; i++)
          for (int j = 0; j < suit.length; j++)
               deck[4 * i + j] = rank[i] + " of " + suit[j];

       // Prints the deck
       for (int i = 0; i < 52; i++) {
           System.out.println(deck[i]);
       }
       System.out.println();

       // Selects and prints a random card
       System.out.println(deck[(int) (Math.random() * 52)]);
       ...
```

```
% java Deck
2 of Clubs
2 of Diamonds
2 of Hearts
2 of Spades
3 of Clubs
3 of Diamonds
3 of Hearts
3 of Spades
4 of Clubs
4 of Diamonds
4 of Hearts
4 of Spades
5 of Clubs
5 of Diamonds
...
Ace of Clubs
Ace of Diamonds
Ace of Hearts
Ace of Spades

4 of Hearts
```

Shuffling

Each time you run this program,
it outputs a shuffled deck of cards

```
% java Deck

10 of
King
2 of
3 of
4 of
Queen
2 of
7 of
6 of
Queen
3 of
Jack
6 of
8 of
9 of
...
6 of
```

```
% java Deck

3 of
6 of
3 of
Jack
8 of
4 of
2 of
7 of
6 of
10 o
King
2 of
3 of
Jack
6 of
...
4 of
```

```
% java Deck

5 of Clubs
Jack of Hearts
9 of Spades
10 of Spades
9 of Clubs
7 of Spades
6 of Diamonds
7 of Hearts
7 of Clubs
5 of Spades
4 of Spades
Queen of Diamonds
5 of Diamonds
Jack of Clubs
Ace of Hearts
...
10 of Hearts
```

Shuffling

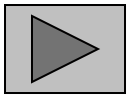
Goal: given an array `deck` of size N,
rearrange its elements in random order.

Shuffling algorithm:

for $i = 0 \ldots N-1$:

Pick a random int r from i to $N-1$

Swap cards `deck[`i`]` and `deck[`r`]`



```
// Shuffles the deck
for (int i = 0; i < N-1; i++) {
    // Selects a random number between i and N-1
    int r = i + (int) (Math.random() * (N-i));
    // Swaps cards r and card i
    String temp = deck[r];
    deck[r] = deck[i];
    deck[i] = temp;
}
```

```
% java Deck

5 of Clubs
Jack of Hearts
9 of Spades
10 of Spades
9 of Clubs
7 of Spades
6 of Diamonds
7 of Hearts
7 of Clubs
5 of Spades
4 of Spades
Queen of Diamonds
5 of Diamonds
Jack of Clubs
Ace of Hearts
...
10 of Hearts
```

Final version

```
public class Deck {
   public static void main(String[] args) {
      // Builds a deck of cards
      String[] suit = { "Clubs", "Diamonds", "Hearts", "Spades" };
      String[] rank = { "2", "3", "4", "5", "6", "7", "8", "9",
                        "10", "Jack", "Queen", "King", "Ace"   };
      int Nsuit = suit.length
      int Nrank = rank.length;
      int N = Nsuit * Nrank;

      String[] deck = new String[N];
      for (int i = 0; i < Nrank; i++)
          for (int j = 0; j < Nsuit; j++)
              deck[Nsuit * i + j] = rank[i] + " of " + suit[j];

      // Shuffles the deck
      for (int i = 0; i < N-1; i++) {
           int r = i + (int) (Math.random() * (N-i));
           String temp = deck[r];
           deck[r] = deck[i];
           deck[i] = temp;
      }

      // Prints the shuffled deck
      for (int i = 0; i < N; i++)
           System.out.println(deck[i]);
   }
}
```

```
% java Deck

5 of Clubs
Jack of Hearts
9 of Spades
10 of Spades
9 of Clubs
7 of Spades
6 of Diamonds
7 of Hearts
7 of Clubs
5 of Spades
4 of Spades
Queen of Diamonds
5 of Diamonds
Jack of Clubs
Ace of Hearts
...
10 of Hearts
```

End comment

How can we tell that a deck is well shuffled?

End comment

How PlanetPoker.com went broke:

In on-line poker, the card shuffling is done by the game software.

PlanetPoker.com used a naïve shuffling algorithm.

Clever players learned to exploit the weakness, and the company went bankrupt.

The article