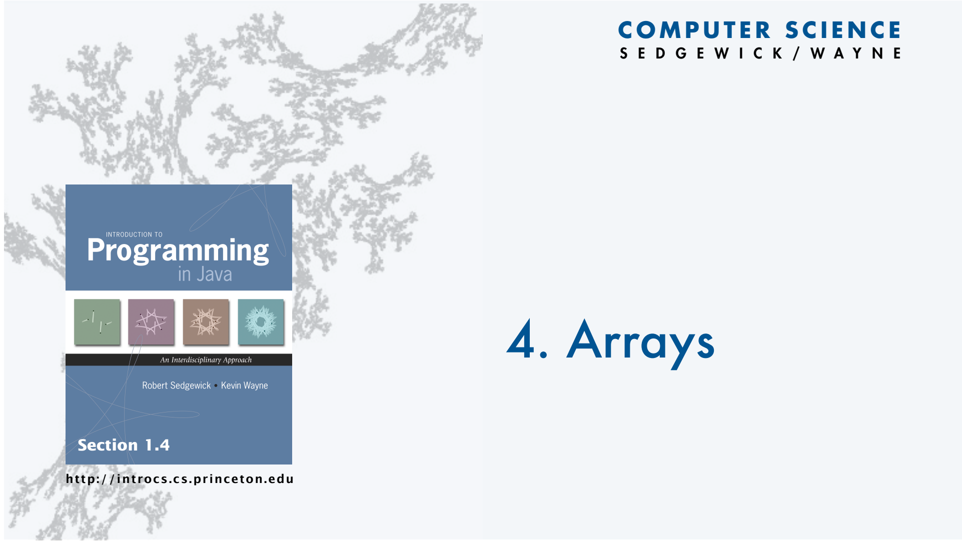


COMPUTER SCIENCE
SEDGEWICK / WAYNE

INTRODUCTION TO
Programming
in Java

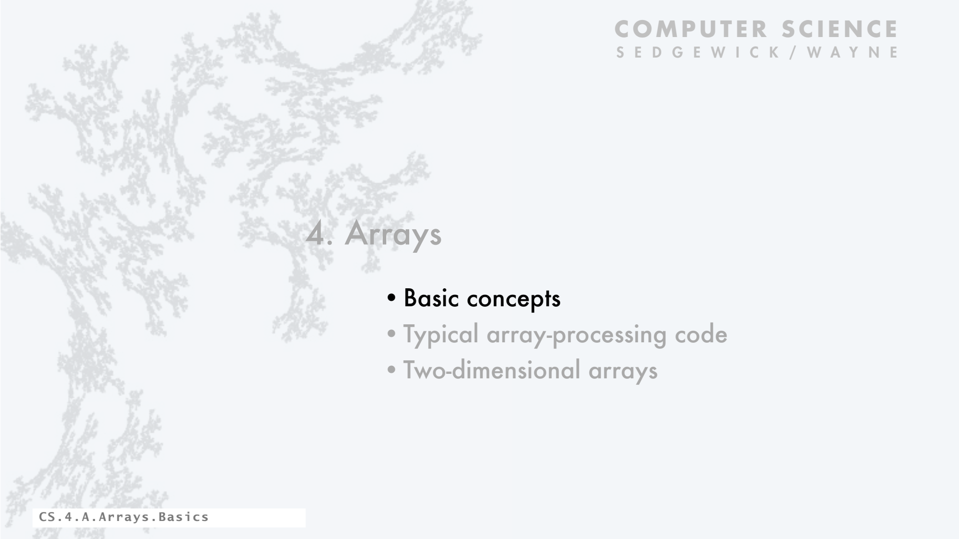
An Interdisciplinary Approach

Robert Sedgewick · Kevin Wayne

Section 1.4

http://introcs.cs.princeton.edu

4. Arrays

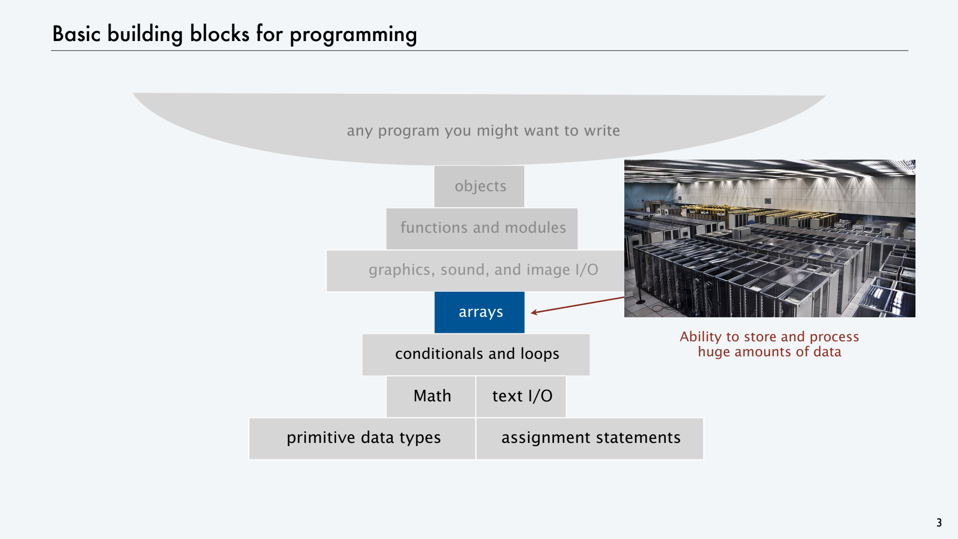


COMPUTER SCIENCE
SEDGEWICK / WAYNE

4. Arrays

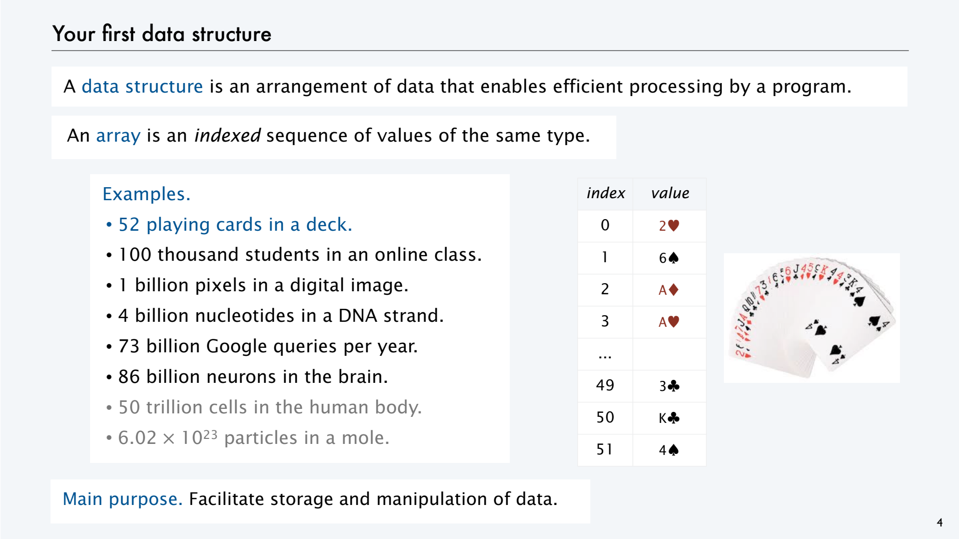
- **Basic concepts**
- Typical array-processing code
- Two-dimensional arrays

CS.4.A.Arrays.Basics



Basic building blocks for programming

Ability to store and process huge amounts of data



Your first data structure

A data structure is an arrangement of data that enables efficient processing by a program.

An array is an *indexed* sequence of values of the same type.

Examples.

- 52 playing cards in a deck.
- 100 thousand students in an online class.
- 1 billion pixels in a digital image.
- 4 billion nucleotides in a DNA strand.
- 73 billion Google queries per year.
- 86 billion neurons in the brain.
- 50 trillion cells in the human body.
- 6.02×10^{23} particles in a mole.

index	*value*
0	2♥
1	6♠
2	A♦
3	A♥
...	
49	3♣
50	K♣
51	4♠

Main purpose. Facilitate storage and manipulation of data.

Processing many values of the same type

10 values, without arrays

```
double a0 = 0.0;
double a1 = 0.0;
double a2 = 0.0;
double a3 = 0.0;
double a4 = 0.0;
double a5 = 0.0;
double a6 = 0.0;
double a7 = 0.0;
double a8 = 0.0;
double a9 = 0.0;
...
a4 = 3.0;
...
a8 = 8.0;
...
double x = a4 + a8;
```

tedious and error-prone code

10 values, with an array

```
double[] a;
a = new double[10];
...
a[4] = 3.0;
...
a[8] = 8.0;
...
double x = a[4] + a[8];
```

an easy alternative

1 million values, with an array

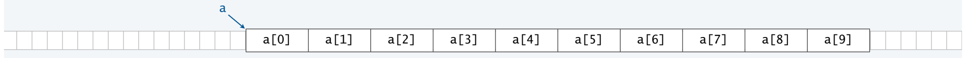
```
double[] a;
a = new double[1000000];
...
a[234567] = 3.0;
...
a[876543] = 8.0;
...
double x = a[234567] + a[876543];
```

scales to handle huge amounts of data

Memory representation of an array

An array is an indexed sequence of values of the same type.

A computer's memory is *also* an indexed sequence of memory locations. ← stay tuned for many details
- Each primitive type value occupies a fixed number of locations.
- *Array values are stored in contiguous locations.*



Critical concepts
- The array name a refers to the first value in the array.
- Indices start at 0.
- Given i, the operation of accessing the value a[i] is extremely efficient.
- The assignment b = a makes the names b and a refer to the same array. ← it does *not* copy the array, as with primitive types (stay tuned for details)

Java language support for arrays

Basic support

operation	typical code
Declare an array	`double[] a;`
Create an array of a given length	`a = new double[1000];`
Refer to an array entry by index	`a[i] = b[j] + c[k];`
Refer to the length of an array	`a.length;`

Initialization options

operation	typical code

Explicitly set all entries to some value	`for (int i = 0; i < a.length; i++)`
	`a[i] = 0.0;`

Default initialization to 0 for numeric types	`a = new double[1000];`
Declare, create and initialize in one statement	`double[] a = new double[1000];`
Initialize to literal values	`double[] x = { 0.3, 0.6, 0.1 };`

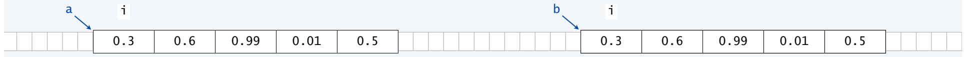
equivalent in Java

cost of creating an array is proportional to its length.

Copying an array

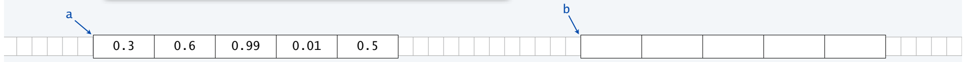
To copy an array, create a new array , then copy all the values.

```
double[] b = new double[a.length];
for (int i = 0; i < a.length; i++)
   b[i] = a[i];
```



Important note: The code b = a does *not* copy an array (it makes b and a refer to the same array).

```
double[] b = new double[a.length];
b = a;
```



Pop quiz 1 on arrays

Q. What does the following code print?

```
public class PQarray1
{
    public static void main(String[] args)
    {
        int[] a;
        int[] b = new int[a.length];

        b = a;
        for (int i = 1; i < b.length; i++)
            b[i] = i;

        for (int i = 0; i < a.length; i++)
            System.out.print(a[i] + " ");
        System.out.println();

        for (int i = 0; i < b.length; i++)
            System.out.print(b[i] + " ");
        System.out.println();
    }
}
```

Programming with arrays: typical examples

For brevity, N is a.length and b.length in all this code.

Access command-line args in system array

```
int stake  = Integer.parseInt(args[0]);
int goal   = Integer.parseInt(args[1]);
int trials = Integer.parseInt(args[2]);
```

Copy to another array

```
double[] b = new double[N];
for (int i = 0; i < N; i++)
   b[i] = a[i];
```

Create an array with N random values

```
double[] a = new double[N];
for (int i = 0; i < N; i++)
   a[i] = Math.random();
```

Print array values, one per line

```
for (int i = 0; i < N; i++)
  System.out.println(a[i]);
```

Compute the average of array values

```
double sum = 0.0;
for (int i = 0; i < N; i++)
   sum += a[i];
double average = sum / N;
```

Find the maximum of array values

```
double max = a[0];
for (int i = 1; i < N; i++)
   if (a[i] > max) max = a[i];
```

Programming with arrays: typical bugs

Array index out of bounds

```
double[] a = new double[10];
for (int i = 1; i <= 10; i++)
  a[i] = Math.random();
```

No a[10] (and a[0] unused)

Uninitialized array

```
double[] a;
for (int i = 0; i < 9; i++)
  a[i] = Math.random();
```

Never created the array

Undeclared variable

```
a = new double[10];
for (int i = 0; i < 10; i++)
  a[i] = Math.random();
```

What type of data does a refer to?

COMPUTER SCIENCE
SEDGEWICK / WAYNE

4. Arrays

- Basic concepts
- **Examples of array-processing code**
- Two-dimensional arrays

CS.4.B.Arrays.Examples

Example of array use: create a deck of cards

Define three arrays

- Ranks.
- Suits.
- Full deck.

```
String[] rank = {"2", "3", "4", "5", "6", "7", "8", "9", "10", "J", "Q", "K", "A" };
```

```
String[] suit = { "♣ ", "♦ ", "♥ ", "♠ " };
```

```
String[] deck[52];
```

Use nested for loops to put all the cards in the deck.

```
for (int j = 0; j < 4; j++)
   for (int i = 0; i < 13; i++)
      deck[i + 13*j] = rank[i] + suit[j];
```

j	0	1	2	3
suit	♣	♦	♥	♠

i	0	1	2	3	4	5	6	7	8	9	10	11	12
rank	2	3	4	5	6	7	8	9	10	J	Q	K	A

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	...
deck	2♣	3♣	4♣	5♣	6♣	7♣	8♣	9♣	10♣	J♣	Q♣	K♣	A♣	2♦	3♦	4♦	5♦	6♦	7♦	8♦	9♦	...

Example of array use: create a deck of cards

```
public class Deck
{
   public static void main(String[] args)
   {
      String[] rank = {"2", "3", "4", "5", "6", "7", "8", "9", "10",
                       "J", "Q", "K", "A" };
      String[] suit = { "♣ ", "♦ ", "♥ ", "♠ " };
      String[] deck = new String[52];

      for (int j = 0; j < 4; j++)
         for (int i = 0; i < 13; i++)
            deck[i + 13*j] = rank[i] + suit[j];

      for (int i = 0; i < 52; i++)
         System.out.print(deck[i]);
      System.out.println();
   }
}
```

no color in Unicode;
artistic license for lecture

```
% java Deck
2♣ 3♣ 4♣ 5♣ 6♣ 7♣ 8♣ 9♣ 10♣ J♣ Q♣ K♣ A♣
2♦ 3♦ 4♦ 5♦ 6♦ 7♦ 8♦ 9♦ 10♦ J♦ Q♦ K♦ A♦
2♥ 3♥ 4♥ 5♥ 6♥ 7♥ 8♥ 9♥ 10♥ J♥ Q♥ K♥ A♥
2♠ 3♠ 4♠ 5♠ 6♠ 7♠ 8♠ 9♠ 10♠ J♠ Q♠ K♠ A♠
%
```

Pop quiz 2 on arrays

Q. What happens if the order of the for loops in Deck is switched?

```
for (int j = 0; j < 4; j++)
   for (int i = 0; i < 13; i++)
      deck[i + 13*j] = rank[i] + suit[j];
```

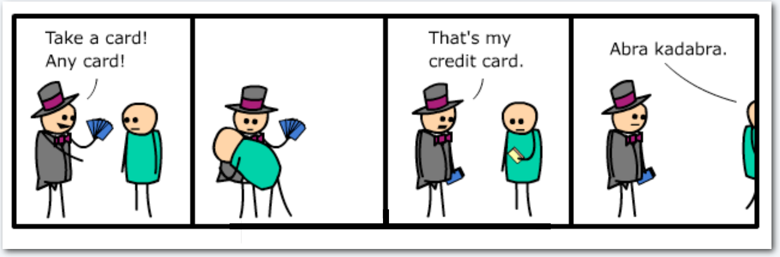
→

```
for (int i = 0; i < 13; i++)
   for (int j = 0; j < 4; j++)
      deck[i + 13*j] = rank[i] + suit[j];
```

Pop quiz 3 on arrays

Q. Change Deck to put the cards in rank order in the array.

```
% java Deck
2♣ 2♦ 2♥ 2♠ 3♣ 3♦  3♥  3♠  4♣ 4♦ 4♥ 4♠ 5♣ 5♦ 5♥ 5♠ 6♣ 6♦ 6♥ 6♠ 7♣ 7♦ 7♥ 7♠ 8♣ 8♦
8♥ 8♠ 9♣ 9♦ 9♥ 9♠ 10♣ 10♦ 10♥ 10♠ J♣ J♦ J♥ J♠ Q♣ Q♦ Q♥ Q♠ K♣ K♦ K♥ K♠ A♣ A♦ A♥ A♠
%
```



Array application: take a card, any card

Problem: Print a random sequence of *N* cards.

Algorithm
Take *N* from the command line and do the following *N* times
- Calculate a random index *p* between 0 and 51.
- Print deck[p].

Implementation: Add this code instead of printing deck in Deck.

each value between 0 and 51 equally likely

```
for (int i = 0; i < N; i++)
   System.out.println(deck[(int) (Math.random() * 52)]);
```

Note: Same method is effective for printing a random sequence from any data collection.

Array application: random sequence of cards

```
public class DrawCards
{
   public static void main(String[] args)
   {
      int N = Integer.parseInt(args[0]);

      String[] rank = {"2", "3", "4", "5", "6", "7", "8", "9",
                          "10", "J", "Q", "K", "A" };
      String[] suit = { "♣ ", "♦ ", "♥ ", "♠ " };
      String[] deck = new String[52];

      for (int i = 0; i < 13; i++)
         for (int j = 0; j < 4; j++)
            deck[i + 13*j] = rank[i] + " of " + suit[j];

      for (int i = 0; i < N; i++)
         System.out.print(deck[(int) (Math.random() * 52)]);
      System.out.println();
   }
}
```

```
% java DrawCards 10
6♥ K♦ 10♠ 8♦ 9♦ 9♥ 6♦ 10♠ 3♣ 5♦
```

appears twice

```
% java DrawCards 10
2♦ A♠ 5♣ A♣ 10♣ Q♦ K♣ K♠ A♣ A♦
```

```
% java DrawCards 10
6♠ 10♦ 4♥ A♣ K♥ Q♠ K♠ 7♣ 5♦ Q♠
```

```
% java DrawCards 10
A♣ J♣ 5♥ K♥ Q♣ 5♥ 9♦ 9♣ 6♠ K♥
```

Note: Sample is *with* replacement (same card can appear multiple times).

Array application: shuffle and deal from a deck of cards

Problem: Print *N* random cards from a deck.

Algorithm: Shuffle the deck, then deal.
- Consider each card index *i* from 0 to 51.
  - Calculate a random index *p* between *i* and 51.
  - Exchange deck[i] with deck[p]
- Print the first *N* cards in the deck.

Implementation

```
for (int i = 0; i < 52; i++)
{
   int p = i + (int) (Math.random() * (52-i));
   String t = deck[p];
   deck[p] = deck[i];
   deck[i] = t;
}
for (int i = 0; i < N; i++) System.out.print(deck[i]);
System.out.println();
```

each value between *i* and 51 equally likely

Array application: shuffle a deck of 10 cards (trace)

```
for (int i = 0; i < 10; i++)
{
   int p = i + (int) (Math.random() * (10-i));
   String t = deck[p];
   deck[p] = deck[i];
   deck[i] = t;
}
```

i	p	deck 0	1	2	3	4	5	6	7	8	9
		2♣	3♣	4♣	5♣	6♣	7♣	8♣	9♣	10♣	J♣
0	7	9♣	3♣	4♣	5♣	6♣	7♣	8♣	2♣	10♣	J♣
1	3	9♣	5♣	4♣	3♣	6♣	7♣	8♣	2♣	10♣	J♣
2	9	9♣	5♣	J♣	3♣	6♣	7♣	8♣	2♣	10♣	4♣
3	9	9♣	5♣	J♣	4♣	6♣	7♣	8♣	2♣	10♣	3♣
4	6	9♣	5♣	J♣	4♣	8♣	7♣	6♣	2♣	10♣	3♣
5	9	9♣	5♣	J♣	4♣	8♣	3♣	6♣	2♣	10♣	7♣
6	8	9♣	5♣	J♣	4♣	8♣	3♣	10♣	2♣	6♣	7♣
7	9	9♣	5♣	J♣	4♣	8♣	3♣	10♣	7♣	6♣	2♣
8	8	9♣	5♣	J♣	4♣	8♣	3♣	10♣	7♣	6♣	2♣
9	9	9♣	5♣	J♣	4♣	8♣	3♣	10♣	7♣	6♣	2♣

Q. Why does this method work?

- Uses only exchanges, so the deck after the shuffle has the same cards as before.
- $N-i$ equally likely values for deck[i].
- Therefore $N\times(N-1)\times(N-1)\ldots\times2\times1 = N!$ equally likely values for deck[].

Initial order is immaterial.

Note: Same method is effective for randomly rearranging any type of data.

Array application: shuffle and deal from a deck of cards

```
public class DealCards
{
    public static void main(String[] args)
    {
        int N = Integer.parseInt(args[0]);

        String[] rank = {"2", "3", "4", "5", "6", "7", "8", "9",
                         "10", "J", "Q", "K", "A" };
        String[] suit = { "♣ ", "♦ ", "♥ ", "♠ " };
        String[] deck = new String[52];

        for (int i = 0; i < 13; i++)
            for (int j = 0; j < 4; j++)
                deck[i + 13*j] = rank[i] + suit[j];

        for (int i = 0; i < 52; i++)
            {
                int p = i + (int) (Math.random() * (52-i));
                String t = deck[p];
                deck[p] = deck[i];
                deck[i] = t;
            }

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

random poker hand

```
% java DealCards 5
9♣ Q♥ 6♥ 4♦ 2♠
```

random bridge hand

```
% java DealCards 13
3♠ 4♥ 10♦ 6♥ 6♦ 2♠ 9♣ 8♠ A♠ 3♥ 9♠ 5♠ Q♥
```

Coupon collector

Coupon collector problem

- M different types of coupons.
- Collector acquires random coupons, one at a time, each type equally likely.

Q. What is the expected number of coupons needed to acquire a full collection?

Example: Collect all ranks in a random sequence of cards ($M = 13$).

Sequence

9♣	5♠	8♥	10♦	2♠	A♠	10♥	Q♦	3♠	9♥	5♦	9♣	7♦	2♦	8♣	6♣	Q♥	K♣	10♥	A♦	4♦	J♥

Collection

2	3	4	5	6	7	8	9	10	J	Q	K	A
2♠	3♠	4♦	5♠	6♣	7♦	8♥	9♣	10♦	J♥	Q♦	K♣	A♠
2♦			5♦			8♣	9♥	10♥		Q♥		A♦
							9♣	10♥				

22 cards needed to complete collection

Array application: coupon collector

Coupon collector simulation

- Generate random int values between 0 and $M-1$.
- Count number used to generate each value at least once.

Key to the implementation

- Create a boolean array of length M. (Initially all false by default.)
- When r generated, check the rth value in the array.
  - If true, ignore it (not new).
  - If false, count it as new (and set rth entry to true)

```
public class Coupon
{
   public static void main(String[] args)
   {
      int M = Integer.parseInt(args[0]);
      int cardcnt = 0; // number of cards collected
      int cnt = 0;     // number of distinct cards

      boolean[] found = new boolean[M];
      while (cnt < M)
      {
         int r = (int) (Math.random() * M);
         cardcnt++;
         if (!found[r])
         {
            cnt++;
            found[r] = true;
         }
      }

      System.out.println(cardcnt);
   }
}
```

```
% java Coupon 13
46

% java Coupon 13
22

% java Coupon 13
54

% java Coupon 13
27
```

Array application: coupon collector (trace for M = 6)

```
boolean[] found = new boolean[M];
while (cnt < M)
{
   int r = (int) (Math.random() * M);
   cardcnt++;
   if (!found[r])
   {
      cnt++;
      found[r] = true;
   }
}
```

r	found						cnt	cardcnt
	0	1	2	3	4	5		
	F	F	F	F	F	F	0	0
2	F	F	T	F	F	F	1	1
0	T	F	T	F	F	F	2	2
4	T	F	T	F	T	F	3	3
0	T	F	T	F	T	F	3	4
1	T	T	T	F	T	F	4	5
2	T	T	T	F	T	F	4	6
5	T	T	T	F	T	T	5	7
0	T	T	T	F	T	T	5	8
1	T	T	T	F	T	T	5	9
3	T	T	T	T	T	T	6	10

Simulation, randomness, and analysis (revisited)

Coupon collector problem

- *M* different types of coupons.
- Collector acquires random coupons, one at a time, each type equally likely.

Q. What is the expected number of coupons needed to acquire a full collection?

Pierre-Simon Laplace
1749-1827

A. (known via mathematical analysis for centuries) About $M \ln M + .57721M$.

type	*M*	*expected wait*
playing card suits	4	8
playing card ranks	13	41
baseball cards	1200	9201
Magic™ cards	12534	125508

```
% java Coupon 4
11

% java Coupon 13
38

% java Coupon 1200
8789

% java Coupon 12534
125671
```

Remarks

- Computer simulation can help validate mathematical analysis.
- Computer simulation can also validate software behavior.

Example: Is `Math.random()` simulating randomness?

Simulation, randomness, and analysis (revisited)

Once simulation is debugged, experimental evidence is easy to obtain.

Gambler's ruin simulation, previous lecture

```
public class Gambler
{
    public static void main(String[] args)
    {
      int stake  = Integer.parseInt(args[0]);
      int goal   = Integer.parseInt(args[1]);
      int trials = Integer.parseInt(args[2]);

      int wins   = 0;
      for (int i = 0; i < trials; i++)
      {
         int t = stake;
         while (t > 0 && t < goal)
         {
            if (Math.random() < 0.5) t++;
            else                     t--;
         }
         if (t == goal) wins++;
      }
      System.out.println(wins + " wins of " + trials);
   }
}
```

Analogous code for coupon collector, this lecture

```
public class Collector
{
    public static void main(String[] args)
    {
      int M = Integer.parseInt(args[0]);
      int trials = Integer.parseInt(args[1]);
      int cardcnt = 0;
      boolean[] found;

      for (int i = 0; i < trials; i++)
      {
         int cnt = 0;
         found = new boolean[M];
         while (cnt < M)
         {
            int r = (int) (Math.random() * M);
            cardcnt++;
            if (!found[r])
               { cnt++; found[r] = true; }
         }
      }
      System.out.println(cardcnt/trials);
    }
}
```

Simulation, randomness, and analysis (revisited)

Coupon collector problem

- *M* different types of coupons.
- Collector acquires random coupons, one at a time, each type equally likely.

Q. What is the expected number of coupons needed to acquire a full collection?

Predicted by mathematical analysis

type	*M*	$M \ln M + .57721M$
playing card suits	4	8
playing card ranks	13	41
playing cards	52	236
baseball cards	1200	9201
magic cards	12534	125508

Observed by computer simulation

```
% java Collector 4 1000000
8

% java Collector 13 1000000
41

% java Collector 52 100000
236

% java Collector 1200 10000
9176

% java Collector 12534 1000
125920
```

Hypothesis. Centuries-old analysis is correct and `Math.random()` simulates randomness.

COMPUTER SCIENCE
SEDGEWICK / WAYNE

4. Arrays

- Basic concepts
- Examples of array-processing code
- **Two-dimensional arrays**

CS.4.C.Arrays.2D

Two-dimensional arrays

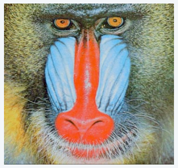
A two-dimensional array is a *doubly-indexed* sequence of values of the same type.

Examples
- Matrices in math calculations.
- Grades for students in an online class.
- Outcomes of scientific experiments.
- Transactions for bank customers.
- Pixels in a digital image.
- Geographic data
- ...

grade

student ID

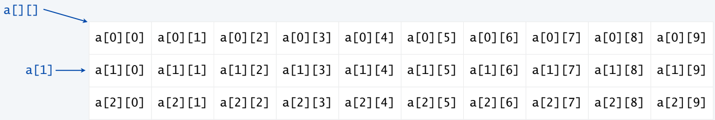
	0	1	2	3	4	5	...
0	A	A	C	B	A	C	
1	B	B	B	B	A	A	
2	C	D	D	B	C	A	
3	A	A	A	A	A	A	
4	C	C	B	C	B	B	
5	A	A	A	B	A	A	
...							



Main purpose. Facilitate storage and manipulation of data.

Java language support for two-dimensional arrays (basic support)

operation	*typical code*
Declare a two-dimensional array	`double[][] a;`
Create a two-dimensional array of a given length	`a = new double[1000][1000];`
Refer to an array entry by index	`a[i][j] = b[i][j] * c[j][k];`
Refer to the number of rows	`a.length;`
Refer to the number of columns	`a[i].length;` ← can be different for each row
Refer to row *i*	`a[i]` ← no way to refer to column j



a[0][0]	a[0][1]	a[0][2]	a[0][3]	a[0][4]	a[0][5]	a[0][6]	a[0][7]	a[0][8]	a[0][9]
a[1][0]	a[1][1]	a[1][2]	a[1][3]	a[1][4]	a[1][5]	a[1][6]	a[1][7]	a[1][8]	a[1][9]
a[2][0]	a[2][1]	a[2][2]	a[2][3]	a[2][4]	a[2][5]	a[2][6]	a[2][7]	a[2][8]	a[2][9]

Java language support for two-dimensional arrays (initialization)

operation	*typical code*

Explicitly set all entries to 0	`for (int i = 0; i < a.length; i++)`
	`   for (int j = 0; j < a[i].length; j++)`
	`      a[i][j] = 0.0;`

Default initialization to 0 for numeric types	`a = new double[1000][1000];`
Declare, create and initialize in a single statement	`double[][] a = new double[1000][1000];`

Initialize to literal values	`double[][] p =`
	`{`
	`   { .92, .02, .02, .02, .02 },`
	`   { .02, .92, .32, .32, .32 },`
	`   { .02, .02, .02, .92, .02 },`
	`   { .92, .02, .02, .02, .02 },`
	`   { .47, .02, .47, .02, .02 },`
	`};`

equivalent in Java

cost of creating an array is proportional to its size.

Application of arrays: vector and matrix calculations

Mathematical abstraction: vector
Java implementation: 1D array

Mathematical abstraction: matrix
Java implementation: 2D array

Vector addition

```
double[] c = new double[N];
for (int i = 0; i < N; i++)
   c[i] = a[i] + b[i];
```

$$\begin{bmatrix} .30 & .60 & .10 \end{bmatrix} + \begin{bmatrix} .50 & .10 & .40 \end{bmatrix} = \begin{bmatrix} .80 & .70 & .50 \end{bmatrix}$$

Matrix addition

```
double[][] c = new double[N][N];
for (int i = 0; i < N; i++)
   for (int j = 0; j < N; j++)
      c[i][j] = a[i][j] + b[i][j];
```

$$\begin{bmatrix} .70 & .20 & .10 \\ .30 & .60 & .10 \\ .50 & .10 & .40 \end{bmatrix} + \begin{bmatrix} .80 & .30 & .50 \\ .10 & .40 & .10 \\ .10 & .30 & .40 \end{bmatrix} = \begin{bmatrix} 1.5 & .50 & .60 \\ .40 & 1.0 & .20 \\ .60 & .40 & .80 \end{bmatrix}$$

Application of arrays: vector and matrix calculations

Mathematical abstraction: vector
Java implementation: 1D array

Mathematical abstraction: matrix
Java implementation: 2D array

Vector dot product

```
double sum = 0.0;
for (int i = 0; i < N; i++)
   sum = sum + a[i]*b[i];
```

$$\begin{bmatrix} .30 & .60 & .10 \end{bmatrix} \cdot \begin{bmatrix} .50 & .10 & .40 \end{bmatrix} = \begin{bmatrix} .25 \end{bmatrix}$$

i	x[i]	y[i]	x[i]*y[i]	sum
0	0.30	0.50	0.15	0.15
1	0.60	0.10	0.06	0.21
2	0.10	0.40	0.04	0.25

end-of-loop trace

Matrix multiplication

```
double[][] c = new double[N][N];
for (int i = 0; i < N; i++)
   for (int j = 0; j < N; j++)
      for (int k = 0; k < N; k++)
         c[i][j] += a[i][k] * b[k][j];
```

$$\begin{bmatrix} .70 & .20 & .10 \\ \boxed{.30 \;\; .60 \;\; .10} \\ .50 & .10 & .40 \end{bmatrix} * \begin{bmatrix} .80 & .30 & .50 \\ .10 & .40 & .10 \\ .10 & .30 & .40 \end{bmatrix} = \begin{bmatrix} .59 & .32 & .41 \\ .31 & .36 & \boxed{.25} \\ .45 & .31 & .42 \end{bmatrix}$$

Pop quiz 4 on arrays

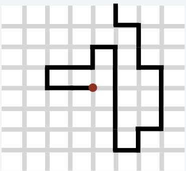
Q. How many multiplications to multiply two *N*-by-*N* matrices?

```
double[][] c = new double[N][N];
for (int i = 0; i < N; i++)
   for (int j = 0; j < N; j++)
      for (int k = 0; k < N; k++)
         c[i][j] += a[i][k] * b[k][j];
```

1. N
2. N^2
3. N^3
4. N^4

Self-avoiding random walks

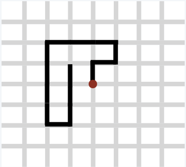
A dog walks around at random in a city, never revisiting any intersection.



escape

Q. Does the dog escape?

Model: a random process in an *N*-by-*N* lattice
- Start in the middle.
- Move to a random neighboring intersection but *do not revisit any intersection.*
- Outcome 1 (escape): reach edge of lattice.
- Outcome 2 (dead end): no unvisited neighbors.

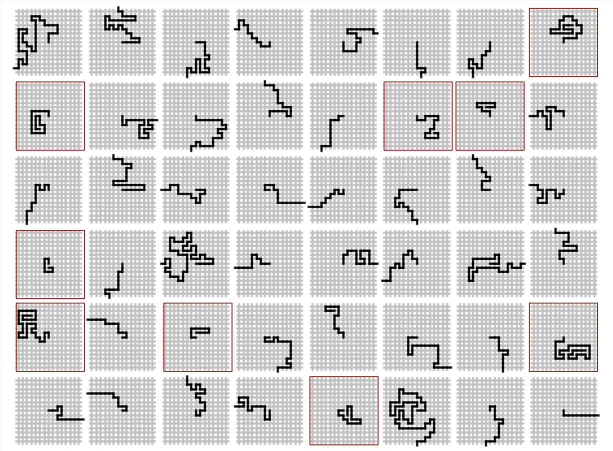


dead end

Q. What are the chances of reaching a dead end?

Approach: Use Monte Carlo simulation, recording visited positions in an *N*-by-*N* array.

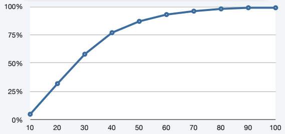
Self-avoiding random walks



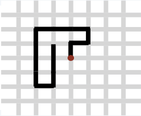
Application of 2D arrays: self-avoiding random walks

```
public class SelfAvoidingWalk
{
   public static void main(String[] args)
   {
      int N = Integer.parseInt(args[0]);
      int trials = Integer.parseInt(args[1]);
      int deadEnds = 0;
      for (int t = 0; t < trials; t++)
      {
         boolean[][] a = new boolean[N][N];
         int x = N/2, y = N/2;

         while (x > 0 && x < N-1 && y > 0 && y < N-1)
         {
            if (a[x-1][y] && a[x+1][y] && a[x][y-1] && a[x][y+1])
            {  deadEnds++;  break; }

            a[x][y] = true;
            double r = Math.random();
            if      (r < 0.25) { if (!a[x+1][y]) x++; }
            else if (r < 0.50) { if (!a[x-1][y]) x--; }
            else if (r < 0.75) { if (!a[x][y+1]) y++; }
            else if (r < 1.00) { if (!a[x][y-1]) y--; }
         }
      }
      System.out.println(100*deadEnds/trials + "% dead ends");
   }
}
```

```
% java SelfAvoidingWalk 10 100000
5% dead ends
% java SelfAvoidingWalk 20 100000
32% dead ends
% java SelfAvoidingWalk 30 100000
58% dead ends
% java SelfAvoidingWalk 40 100000
77% dead ends
% java SelfAvoidingWalk 50 100000
87% dead ends
% java SelfAvoidingWalk 60 100000
93% dead ends
% java SelfAvoidingWalk 70 100000
96% dead ends
% java SelfAvoidingWalk 80 100000
98% dead ends
% java SelfAvoidingWalk 90 100000
99% dead ends
% java SelfAvoidingWalk 100 100000
99% dead ends
```



Simulation, randomness, and analysis (revisited again)



Self-avoiding walk in an *N*-by-*N* lattice

- Start in the middle.
- Move to a random neighboring intersection (do not revisit any intersection).

Applications

- Model the behavior of solvents and polymers.
- Model the physics of magnetic materials.
- (many other physical phenomena)

Paul Flory
1910-1985
Nobel Prize 1974

Q. What is the probability of reaching a dead end?

A. Nobody knows (despite decades of study). ← Mathematicians and physics researchers cannot solve the problem.

A. 99+% for *N* >100 (clear from simulations). ← YOU can!

Computational models play an essential role in modern scientific research.

Remark: Computer simulation is often the *only* effective way to study a scientific phenomenon.

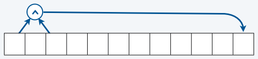
Your first data structure

Arrays: A basic building block in programming

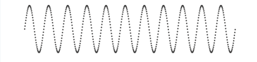
- They enable storage of large amounts of data (values all of the same type).
- With an index, a program can instantly access a given value.
- Efficiency derives from low-level computer hardware organization (stay tuned).

Some applications in this course where *you* will use arrays:

LFSR

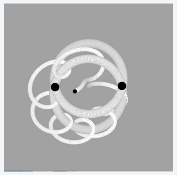


digital audio

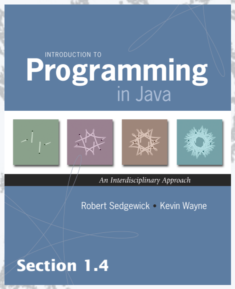


digital images

N-body simulation



COMPUTER SCIENCE
SEDGEWICK / WAYNE



INTRODUCTION TO
Programming
in Java

An Interdisciplinary Approach

Robert Sedgewick • Kevin Wayne

Section 1.4

http://introcs.cs.princeton.edu

4. Arrays