Introduction to Programming

Week 4

Magnus Madsen

Week 4: Arrays

Monday

- What is an array?
- Programming with arrays

Thursday

- Higher-dimensional arrays
- Live programming

Prologue

Do you know about the study cafe?

And have you been there? Why or why not?

Quote of the Week

"In fact, my main conclusion after spending ten years of my life working on the TEX project is that software is hard. It's harder than anything else I've ever had to do."

— Donald Knuth

Epigram of the Week

"Should array indices start at 0 or 1? My compromise of 0.5 was rejected without, I thought, proper consideration."

— Stan Kelly-Bootle

Arrays

What is an array?

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

- An array could model a sequence of the 52 playing cards in a deck.
- An array could model a sequence of the 4 billion nucleotides in a DNA strand.
- An array could model a sequence of 100 students in a class.

What is an array?

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

- An array could model a sequence of the 52 playing cards in a deck.
- An array could model a sequence of the 4 billion nucleotides in a DNA strand.
- An array could model a sequence of 100 students in a class.

An **element** of an array is one of the items in the sequence.

An **index** of an element is the location in the array where the element is held.

- In most programming languages, including Java, we use **zero-based indexing.**

What is an array?

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

- An array could model a sequence of the 52 playing cards in a deck.
- An array could model a sequence of the 4 billion nucleotides in a DNA strand.
- An array could model a sequence of 100 students in a class.

An **element** of an array is one of the items in the sequence.

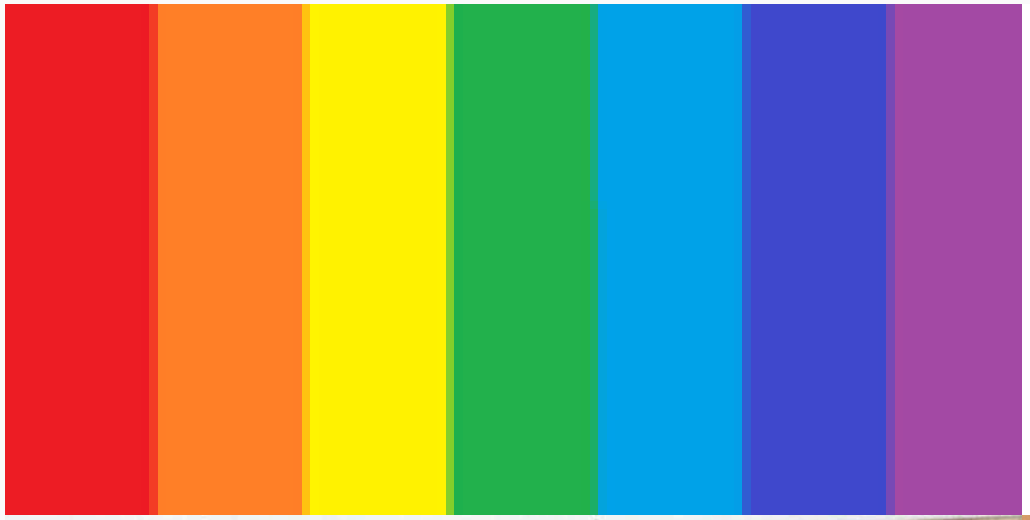
An **index** of an element is the location in the array where the element is held.

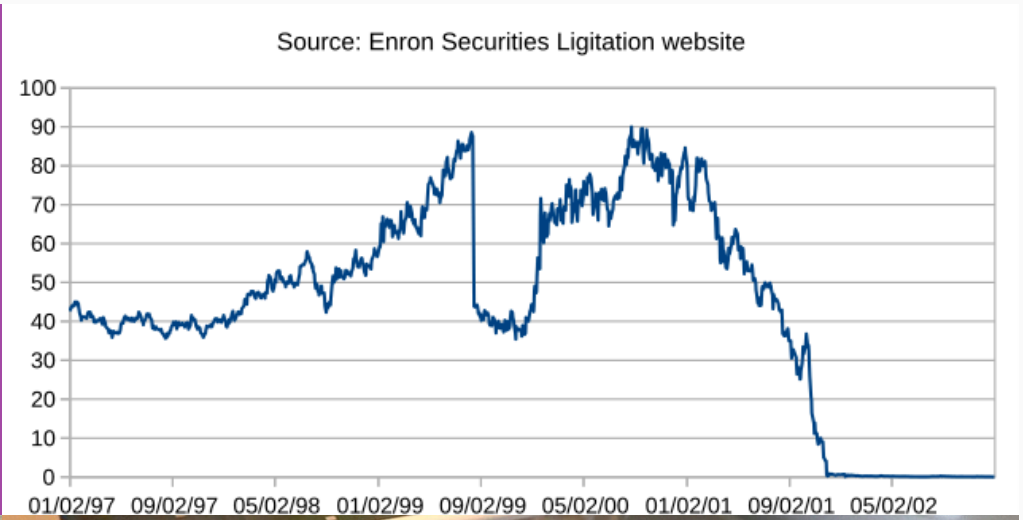
- In most programming languages, including Java, we use **zero-based indexing.**

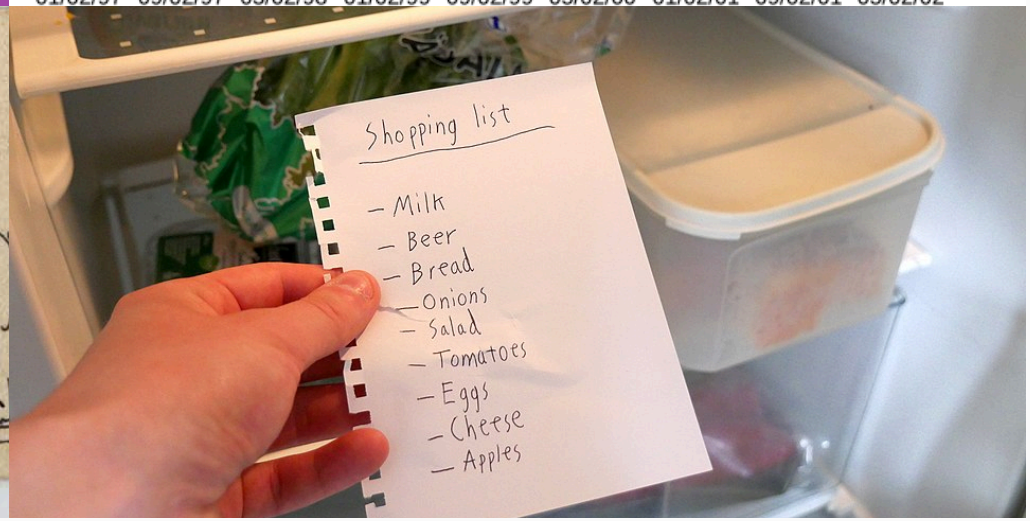
For example, here is an array of fruits:

Indices	0	1	2	3	4
Elements	apple	banana	cucumber	durian	elderberry

One dimensional arrays







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*
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 };`

no need to use a loop like

```
for (int i = 0; i < 1000; i+
   a[i] = 0.0;
```

BUT cost of creating an array is proportional to its length.

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

Example: Arrays in Java

Colors:

```
String[] colors = {"red", "orange", "yellow", "green", "blue", "indigo", "violet"};
```

Example: Arrays in Java

Colors:

```
String[] colors = {"red", "orange", "yellow", "green", "blue", "indigo", "violet"};
```

Stock prices:

```
double[] prices = new double[365];    // daily prices for one year
prices[100] = 45.50;                  // price on day 100
```

Example: Arrays in Java

Colors:

```
String[] colors = {"red", "orange", "yellow", "green", "blue", "indigo", "violet"};
```

Stock prices:

```
double[] prices = new double[365];    // daily prices for one year
prices[100] = 45.50;                  // price on day 100
```

People:

```
String[] queue = new String[10];      // people in line
queue[0] = "Alice";                   // first person in line
```

Example: Arrays in Java

Colors:

```
String[] colors = {"red", "orange", "yellow", "green", "blue", "indigo", "violet"};
```

Stock prices:

```
double[] prices = new double[365];    // daily prices for one year
prices[100] = 45.50;                  // price on day 100
```

People:

```
String[] queue = new String[10];      // people in line
queue[0] = "Alice";                   // first person in line
```

Shopping items:

```
String[] groceries = {"milk", "bread", "eggs", "apples", "cheese"};
```

Example: Printing an array (1/2)

We can print **an array** as follows:

```java
import java.util.Arrays;

public class PrintArray {
    public static void main(String[] args) {
        String[] colors = {"apple", "pear"};
        System.out.println(Arrays.toString(colors));
    }
}
```

```
java PrintArray
```

```
[apple, pear]
```

Example: Printing an array (2/2)

We can print **the elements** of an array as follows:

```java
public class PrintArray {
    public static void main(String[] args) {
        String[] colors = {"apple", "pear"};
        for(int i = 0; i < colors.length; i++){
            System.out.println(colors[i]);
        }
    }
}
```

```
java PrintArray
```

```
apple
pear
```

Example: Accessing command-line arguments

We have already seen how we can access command-line arguments:

```
public class Greet {
    public static void main(String[] args) {
        String first = args[0];
        String last = args[1];
        System.out.println("Hello " + first + " " + last + "!");
    }
}
```

Example: Creating an array with N random values

We can create an array with random numbers and print it:

```
import java.util.Arrays;

public class RandomArray {
    public static void main(String[] args) {
        int length = Integer.parseInt(args[0]);
        double[] a = new double[length];
        for (int i = 0; i < length; i++) {
            a[i] = Math.random();
        }
        System.out.println(Arrays.toString(a));
    }
}
```

Off-by-one errors

Arrays in Java are **zero-indexed**.

The first element in the sequence is at index `0`, the second at index `1`, and so on.

- Starting at zero is convenient for some arithmetic operations, but can lead to trouble.

Off-by-one errors

Arrays in Java are **zero-indexed**.

The first element in the sequence is at index `0`, the second at index `1`, and so on.

- Starting at zero is convenient for some arithmetic operations, but can lead to trouble.

Suppose we are writing a banking application with this simple interface:

```
Press a number to send $100.
1. Your friend Elaine
2. Your friend Kramer
3. Your enemy Newman
```

Elements	Elaine	Kramer	Newman
Indices	0	1	2

This interface is *1-indexed*, but the array we store the accounts in is *0-indexed*.

If the user selects `2` to send money to Kramer, and the programmer uses `2` to index into the array, we'll send money to the wrong person! Oops!

Array length

Every array has a **length** which we can access with the syntax `array.length`.

Indices	0	1	2	3	4
Elements	apple	banana	cucumber	durian	elderberry

This array has length 5. **But its last index is `4`!**

> **i** The last element of an array is always at position `array.length - 1`.

Q:

What does this program print?

```
String[] a = {"Apple", "Pear", "Orange"};
for (int i = 0; i <= a.length; i++) {
    System.out.println(a[i]);
}
```

Example: Find the maximum value

We can find the **maximum value** of an array of numbers:

```
double max = Double.NEGATIVE_INFINITY;
for (int i = 0; i < a.length; i++) {
    if (a[i] > max) {
        max = a[i];
    }
}
```

Q: Why do we initialize `max` to `Double.NEGATIVE_INFINITY`?

Example: Compute the average

We can compute the average of an array of numbers:

```
double sum = 0.0;
for (int i = 0; i < a.length; i++) {
    sum = sum + a[i];
}
double average = sum / a.length;
```

Q: What happens if the array is empty?

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;
```

Example: Reversing an array (1/2)

We can reverse an array:

```
String[] input = {"apple", "banana", "cherry", "pear"};
String[] output = new String[input.length];

for (int i = 0; i < input.length; i++) {
    output[i] = input[input.length - 1 - i];
}
```

Example: Reversing an array (1/2)

We can reverse an array:

```
String[] input = {"apple", "banana", "cherry", "pear"};
String[] output = new String[input.length];

for (int i = 0; i < input.length; i++) {
    output[i] = input[input.length - 1 - i];
}
```

Q: What happens if I write `input[output.length - 1 - i]`?

Example: Reversing an array (2/2)

We can also reverse an array **in-place**:

```
String[] fruits = {"apple", "banana", "cherry", "pear"};

for (int i = 0; i < fruits.length / 2; i++) {
    String tmp = fruits[i];
    fruits[i] = fruits[fruits.length - 1 - i];
    fruits[fruits.length - 1 - i] = tmp;
}
```

Default array initialization

When you create an array, Java automatically initializes it with **default values**:

```
double[] a = new double[100]; // All elements are initialized to 0.0
System.out.println(a[0]);     // Prints 0.0
System.out.println(a[1]);     // Prints 0.0
...
```

Q:

What does this program print?

```
boolean[] a = new boolean[100];
System.out.println(a[0])
```

Relationship between arrays and memory

Arrays are stored as *contiguous blocks* in memory.

When we create an array, we allocate a block of memory and store two pieces of information about it:

- where the array is stored
- the length of the array.

```
int[] a = { 3, 1, 4, 1, 5 };
```

We can think of `a` as containing the address that points to the actual content of the array, stored elsewhere in a contiguous block of memory.

Memory		
address	***content***	***note***
…		
123	523	*address of* `a`
124	8	*length of* `a`
…		
523	3	`a[0]`
524	1	`a[1]`
525	4	`a[2]`
526	1	`a[3]`
527	5	`a[4]`
…		

Q:

What is the difference between:

```
int[] numbers = {1, 2, 3, 4, 5};
for (int i = 0; i < a.length; i++) {
    System.out.println(a[i]);
}
```

and

```
int[] numbers = {1, 2, 3, 4, 5};
System.out.println(Arrays.toString(numbers));
```

Example: Simplifying repetitive code

We can use arrays to simplify repetitive code. Here is a long chain of if-then-else:

```
public class Month {
    public static void main(String[] args) {
        int m = Integer.parseInt(args[0]);
        if (m == 1) System.out.println("Jan");
        else if (m == 2) System.out.println("Feb");
        else if (m == 3) System.out.println("Mar");
        else if (m == 4) System.out.println("Apr");
        ...
        else if (m == 12) System.out.println("Dec");
    }
}
```

Example: Simplifying repetitive code

We can use an array to implement a more elegant solution:

```
public class MonthName {
    public static void main(String[] args) {
        int m = Integer.parseInt(args[0]);
        String[] MONTHS = {
            "", "Jan", "Feb", "Mar", "Apr", "May", "Jun",
            "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"
        };
        System.out.println(MONTHS[m]);
    }
}
```

Trick: We intentionally waste index 0 to make `MONTHS[1]` correspond to January.

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 = new String[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];
```

better style to use rank.length and suit.length
clearer in lecture to use 4 and 13

	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

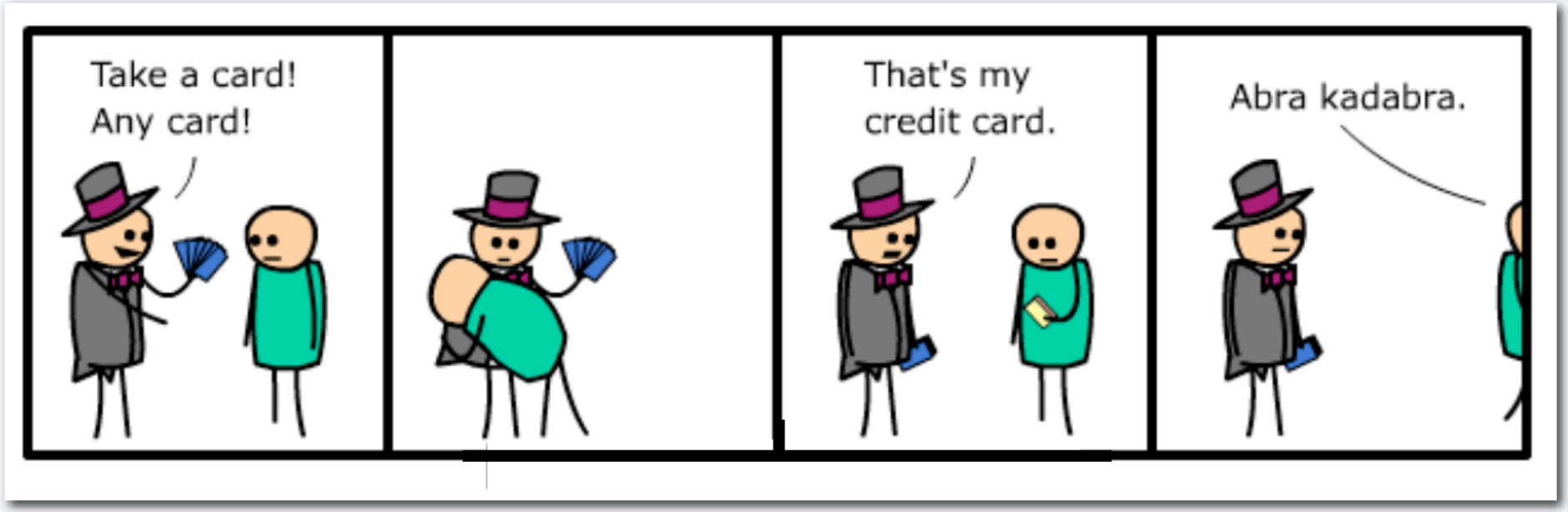
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	...
deck	2♣	3♣	4♣	5♣	6♣	7♣	8♣	9♣	10♣	J♣	Q♣	K♣	A♣	2♦	3♦	4♦	5♦	6♦	7♦	8♦	9♦	...

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♠
%
```


Take a card!
Any card!
That's my
credit card.
Abra kadabra.

Q:

What can you say about these program?

```
double[] prices = new double[3];
prices[42] = 10;
```

```
int[] counts = new int[3];
counts[0] = 10.5;
```

```
int[] counts = new int[1_000_000_000_000_000];
```

The Birthday Paradox

The **Birthday Paradox**: How many people do you need in a room before there's a 50% chance that two share a birthday?

The Birthday Paradox

The **Birthday Paradox**: How many people do you need in a room before there's a 50% chance that two share a birthday?

The surprising answer (i.e. "paradox") is that you need just **23**!

Example: Birthday Paradox Simulation

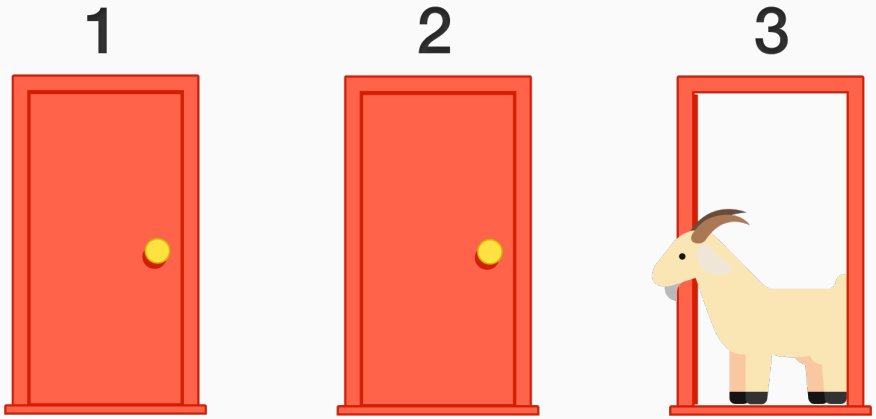
```
int people = Integer.parseInt(args[0]);      // Number of people in room
int trials = Integer.parseInt(args[1]);      // Number of simulations
int matches = 0;
for (int trial = 0; trial < trials; trial++) {
    int[] birthdayOf = new int[people];       // Array of birthdays
    for (int i = 0; i < people; i++) {
        birthdayOf[i] = random.nextInt(365);  // Assign random birthday
    }
    boolean shareBirthday = false;
    for (int i = 0; i < people; i++) {
        for (int j = 0; j < people; j++) {
            if (i != j && birthdayOf[i] == birthdayOf[j]) {
                shareBirthday = true;
            }
        }
    }
    if (shareBirthday) { matches++; }
}
System.out.println("Probability: " + (100.0 * matches / trials) + "%");
```

The Monty Hall Problem

The **Monty Hall Problem** is a famous probability puzzle.

You are in front of **three doors**: Behind one is a 🚗 (prize), behind the other two are 🐐.

You pick door number **3** and it is opened:



The game show host offers you to switch doors. Do you accept? Why / why not?

Example: Monty Hall Simulation

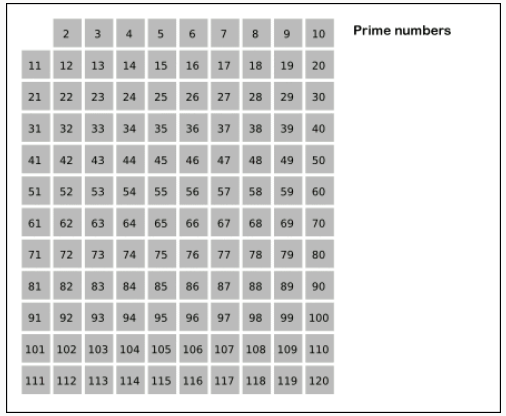
We can simulate thousands of games to verify that switching doors wins 66% of the time!

```
Random random = new Random();
int trials = Integer.parseInt(args[0]);     // Number of games to simulate
int switchWins = 0;
int stayWins = 0;
for (int i = 0; i < trials; i++) {
    boolean[] doors = new boolean[3];       // Three doors (false = goat, true = car)
    int prizeDoor = random.nextInt(3);      // Randomly place the car
    doors[prizeDoor] = true;
    int choice = random.nextInt(3);         // Player picks a random door
    // Host reveals a goat behind some door that is not our choice.
    if (doors[choice]) {
        stayWins++;                         // Player picked the car initially
    } else {
        switchWins++;                       // Player picked a goat, switching wins
    }
}
System.out.println("Stay wins: " + (100.0 * stayWins / trials) + "%");
System.out.println("Switch wins: " + (100.0 * switchWins / trials) + "%");
```

Example: Sieve of Eratosthenes

The **Sieve of Eratosthenes** finds all prime numbers up to a given limit.

```
public class PrimeSieve {
    public static void main(String[] args) {
        int n = Integer.parseInt(args[0]);
        boolean[] isComposite = new boolean[n + 1];
        for (int i = 2; i <= n; i++) {
            for (int j = 2; j * i <= n; j++) {
                isComposite[j * i] = true;
        }}
        for (int i = 2; i <= n; i++) {
            if (!isComposite[i]) {
                System.out.print(i + " ");
        }}
        System.out.println();
}}
```




```
$ java PrimeSieve 20
```

```
2 3 5 7 11 13 17 19
```

Off the Edge of the World

What is `ArrayIndexOutOfBoundsException`?

An **ArrayIndexOutOfBoundsException** is an error that occurs when you try to access an array with an invalid index.

- Arrays in Java are **zero-indexed**: valid indices range from `0` to `array.length - 1`.
- Accessing index `-1` or `array.length` (or higher) throws an exception.
- The program **crashes** at runtime when this happens.
- This is a **runtime** error, not a **compile**-time error.
- Always ensure your indices are within the valid range: `0 <= index < array.length`.

> **i** Java does *not* allow negative indices or wrap-around indexing.

Example: ArrayIndexOutOfBoundsException

```java
int[] numbers = {10, 20, 30};

// These are valid accesses.
System.out.println(numbers[0]);  // ✓ Valid: prints 10
System.out.println(numbers[2]);  // ✓ Valid: prints 30

// These will cause ArrayIndexOutOfBoundsException:
System.out.println(numbers[3]);  // ✗ Index too high
System.out.println(numbers[-1]); // ✗ Negative index
```

```
10
30
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException:
    Index 3 out of bounds for length 3
```

The `Arrays` Class

The `Arrays` Class

Java provides a built-in `Arrays` module with many useful **utility methods**:

- Remember to import it: `import java.util.Arrays`.
- Provides static methods for common array operations.
- Makes array manipulation easier and more efficient.
- Works with arrays of all primitive types and reference types.

Note: In real-life you always check to see if `Arrays` has what you need.

- But in the course the default expectation is that you write your loops yourself.

Overview of the `Arrays` Module

Method	Description
`binarySearch(array, key)`	Searches for a key in a sorted array
`copyOf(array, length)`	Copies the array to a new array of specified length
`copyOfRange(array, from, to)`	Copies a range of the array
`equals(array1, array2)`	Checks if two arrays are equal
`fill(array, value)`	Fills the array with the specified value
`sort(array)`	Sorts the array in ascending order
`toString(array)`	Returns a string representation of the array
...	...

Example: Sorting and Searching

We can use `Arrays` to sort an array and then search for values:

```java
int[] numbers = {42, 17, 3, 23, 67};

// Sort the array
Arrays.sort(numbers);
System.out.println("Sorted:");
System.out.println(Arrays.toString(numbers));

// Binary search for a value
int index = Arrays.binarySearch(numbers, 42);
if (index >= 0) {
    System.out.println("Found 42 at index: " + index);
} else {
    System.out.println("42 not found");
}
```

```
$ java SortAndSearch
```

```
Sorted:
[3, 17, 23, 42, 67]
Found 42 at index: 3
```

Example: Array Equality and Copying

We can use `Arrays` to compare and copy arrays:

```java
int[] a = {1, 2, 3};
int[] b = {1, 2, 3};
int[] c = a;

// Reference equality vs content equality:
System.out.println("a == b: " + (a == b));
System.out.println("a == c: " + (a == c));
System.out.println("Arrays.equals(a, b): " +
                   Arrays.equals(a, b));
// Copying arrays:
int[] original = {10, 20, 30, 40, 50};
int[] copy1 = Arrays.copyOf(original, 3);
int[] copy2 = Arrays.copyOf(original, 7);
int[] range = Arrays.copyOfRange(original, 1, 4);

System.out.println("copy1: " + Arrays.toString(copy1));
System.out.println("copy2: " + Arrays.toString(copy2));
System.out.println("range: " + Arrays.toString(range));
```

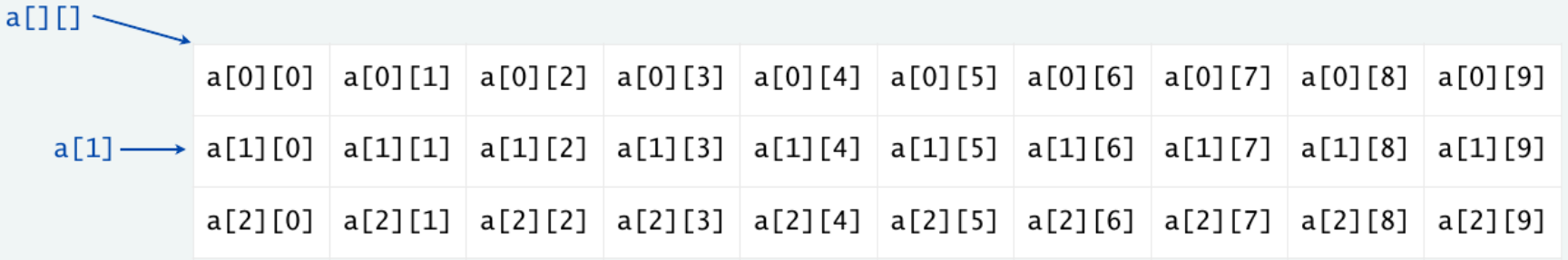
```
$ java ArrayCopy
```

```
a == b: false
a == c: true
Arrays.equals(a, b): true

copy1: [10, 20, 30]
copy2: [10, 20, 30, 40, 50, 0, 0]
range: [20, 30, 40]
```

Higher-dimensional arrays

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 3-by-10 array

Java language support for two-dimensional arrays (initialization)

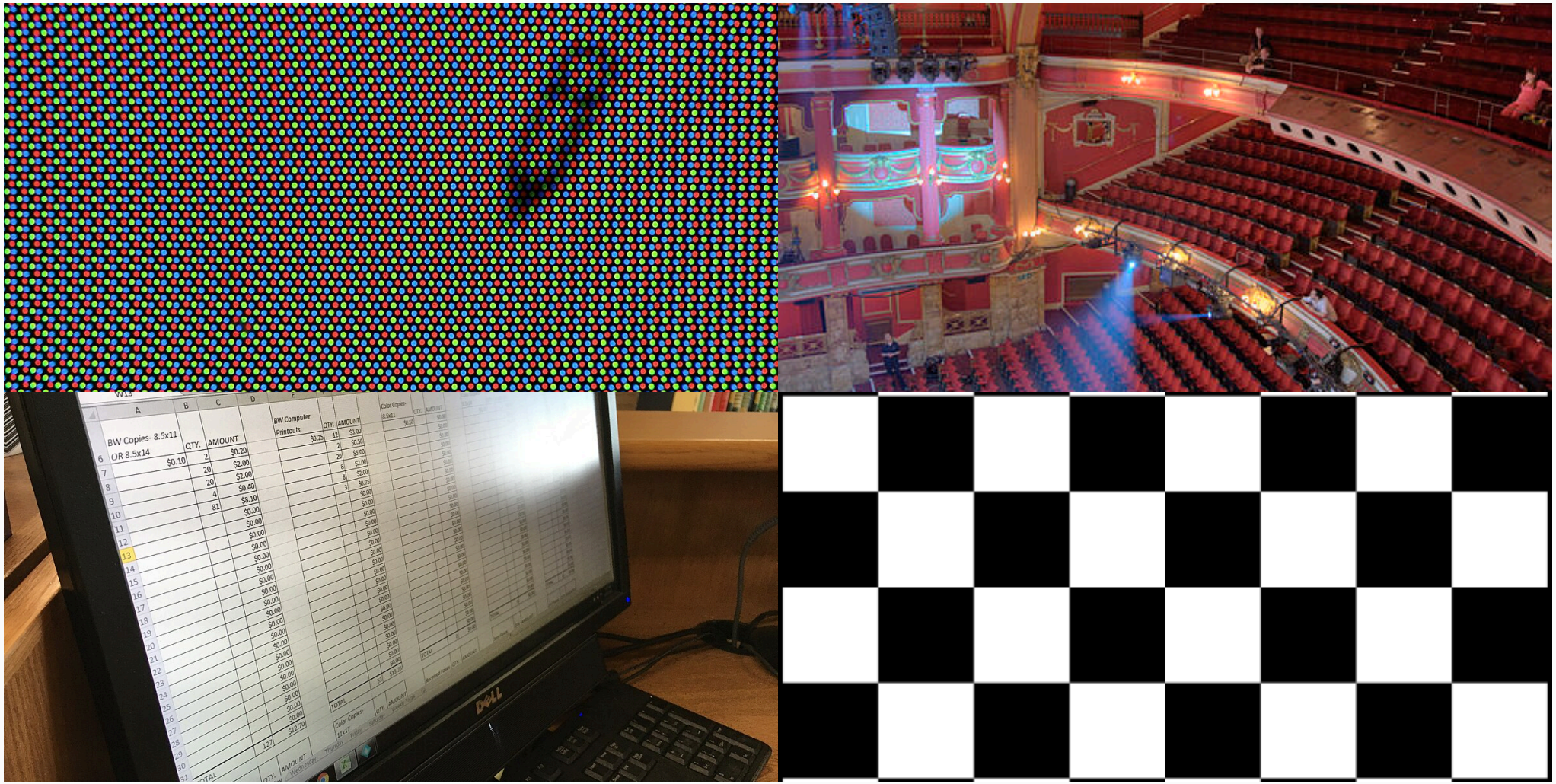
operation	*typical code*
Default initialization to 0 for numeric types	`a = new double[1000][1000];`
Declare, create and initialize in a single statement	`double[][] a = new double[1000][1000];`
Initialize to literal values	`double[][] p = { { .92, .02, .02, .02, .02 }, { .02, .92, .32, .32, .32 }, { .02, .02, .02, .92, .02 }, { .92, .02, .02, .02, .02 }, { .47, .02, .47, .02, .02 }, };`

no need to use nested loops like

```
for (int i = 0; i < 1000; i++)
   for (int j = 0; j < 1000; j++)
    a[i][j] = 0.0;
```

BUT cost of creating an array is proportional to its size.

Examples of 2D arrays



Example: Higher-Dimensional Arrays in Java

Pixel colors in an image:

```
int[][] pixels = new int[800][600];  // width x height
pixels[100][200] = 0xFF0000;         // red pixel at (100, 200)
```

Example: Higher-Dimensional Arrays in Java

Pixel colors in an image:

```
int[][] pixels = new int[800][600];  // width x height
pixels[100][200] = 0xFF0000;         // red pixel at (100, 200)
```

Theater seating chart:

```
boolean[][] seats = new boolean[30][40];  // 30 rows, 40 seats each
seats[15][20] = true;                     // seat 20 in row 15 is occupied
```

Example: Higher-Dimensional Arrays in Java

Pixel colors in an image:

```
int[][] pixels = new int[800][600];  // width x height
pixels[100][200] = 0xFF0000;         // red pixel at (100, 200)
```

Theater seating chart:

```
boolean[][] seats = new boolean[30][40];  // 30 rows, 40 seats each
seats[15][20] = true;                     // seat 20 in row 15 is occupied
```

Chess board positions:

```
String[][] board = new String[8][8];    // 8 x 8 board
board[0][0] = "rook";                   // black rook at a8
```

Example: Higher-Dimensional Arrays in Java

Pixel colors in an image:

```
int[][] pixels = new int[800][600];  // width x height
pixels[100][200] = 0xFF0000;         // red pixel at (100, 200)
```

Theater seating chart:

```
boolean[][] seats = new boolean[30][40];  // 30 rows, 40 seats each
seats[15][20] = true;                     // seat 20 in row 15 is occupied
```

Chess board positions:

```
String[][] board = new String[8][8];    // 8 x 8 board
board[0][0] = "rook";                   // black rook at a8
```

Spreadsheet cells:

```
double[][] sheet = new double[100][26];  // 100 rows, 26 columns (A-Z)
sheet[2][4] = 42.5;                      // value 42.5 in cell E3
```

Example: Matrix addition

We can add two matrices together by adding their elements point-wise.

```
int[][] matrix1 = {{1, 2}, {3, 4}};
int[][] matrix2 = {{5, 6}, {7, 8}};
int[][] result = new int[2][2];

for (int i = 0; i < 2; i++) {
    for (int j = 0; j < 2; j++) {
        result[i][j] = matrix1[i][j] + matrix2[i][j];
    }
}
```

Q: What shape should the matrices have?

Example: Matrix multiplication

We can multiply two matrices together with a triple-nested loop.

```
int[][] matrix1 = {{1, 2}, {3, 4}};
int[][] matrix2 = {{5, 6}, {7, 8}};
int[][] result = new int[2][2];

for (int i = 0; i < 2; i++) {
    for (int j = 0; j < 2; j++) {
        for (int k = 0; k < 2; k++) {
            result[i][j] += matrix1[i][k] * matrix2[k][j];
        }
    }
}
```

Q: What shape should the matrices have?

Example: Matrix symmetry

We can check if a matrix is symmetric.

```
int[][] matrix = {{1, 2, 3}, {2, 4, 5}, {3, 5, 6}};
boolean isSymmetric = true;
for (int i = 0; i < matrix.length; i++) {
    for (int j = 0; j < matrix[i].length; j++) {
        if (matrix[i][j] != matrix[j][i]) {
            isSymmetric = false;
        }
    }
}
if (isSymmetric) {
    System.out.println("Matrix is symmetric");
} else {
    System.out.println("Matrix is not symmetric");
}
```

Example: Multiplication table

We can generate a multiplication table:

```
int size = 10;
int[][] table = new int[size][size];
for (int i = 0; i < size; i++) {
    for (int j = 0; j < size; j++) {
        table[i][j] = (i + 1) * (j + 1);
    }
}
for (int i = 0; i < size; i++) {
    for (int j = 0; j < size; j++) {
        System.out.printf("%2d", table[i][j]);
    }
    System.out.println();
}
```

Multiplication Table (8 × 8)

```
    |  1   2   3   4   5   6   7   8
  --+--------------------------------
  1 |  1   2   3   4   5   6   7   8
  2 |  2   4   6   8  10  12  14  16
  3 |  3   6   9  12  15  18  21  24
  4 |  4   8  12  16  20  24  28  32
  5 |  5  10  15  20  25  30  35  40
  6 |  6  12  18  24  30  36  42  48
  7 |  7  14  21  28  35  42  49  56
  8 |  8  16  24  32  40  48  56  64
```

Q:

What is wrong here?

```
int sum = Integer.MIN_VALUE;
for (int i = 1; i <= a.length; i--) {
    sum += sum + a[i++];
    i = i + 1;
}
double average = (double) (sum / a.length);
```

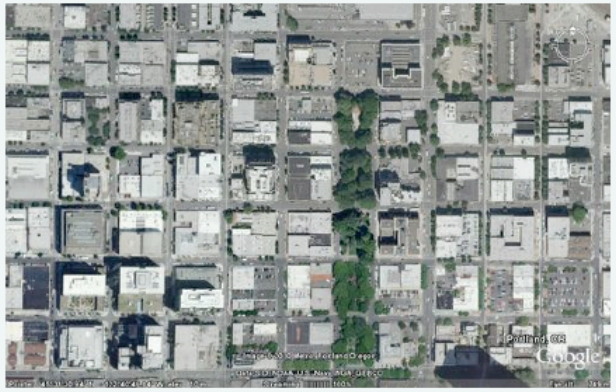
Example: Movie theater booking

We can manage seat reservations in a movie theater:

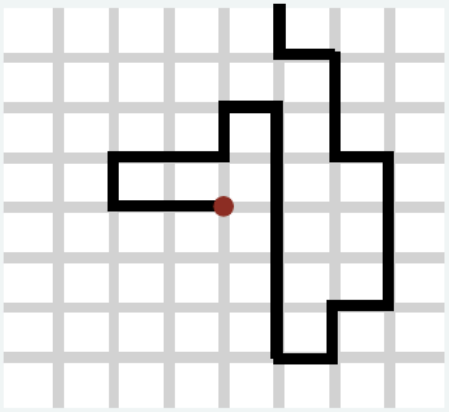
```
String[][] seats = new String[8][12]; // 8 rows, 12 seats each

// Initialize all seats as available
for (int row = 0; row < seats.length; row++) {
    for (int seat = 0; seat < seats[row].length; seat++) {
        seats[row][seat] = "O";
    }
}

// Book some seats
seats[3][5] = "X"; // Row 4, Seat 6
seats[3][6] = "X"; // Row 4, Seat 7
seats[5][8] = "X"; // Row 6, Seat 9

// Print seating chart
System.out.println("Movie Theater Seating (O = Available, X = Booked):");
for (int row = 0; row < seats.length; row++) {
    System.out.print("Row " + (row + 1) + ": ");
    for (int seat = 0; seat < seats[row].length; seat++) {
        System.out.print(seats[row][seat] + " ");
    }
    System.out.println();
}
```

Self-avoiding random walks

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



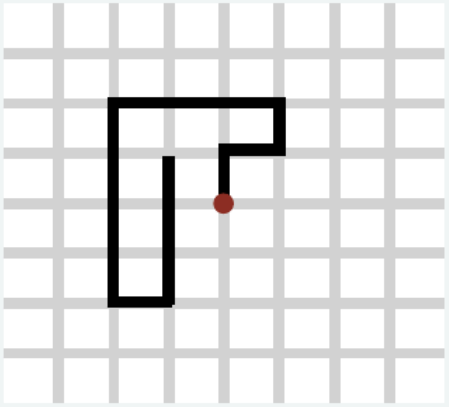
Q. Does the dog escape?



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

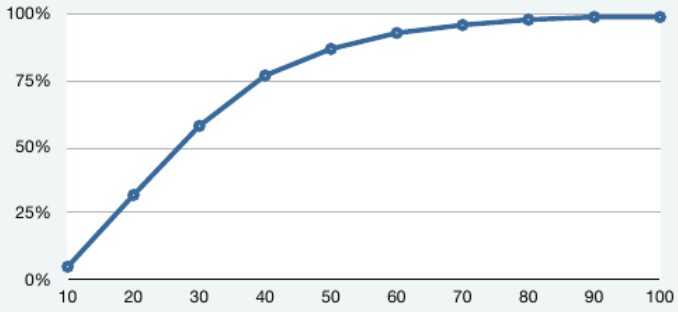
Q. What are the chances of reaching a dead end?

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

Application of 2D arrays: self-avoiding random walks

```
public class SelfAvoidingWalker
{
   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");
   }
}
```

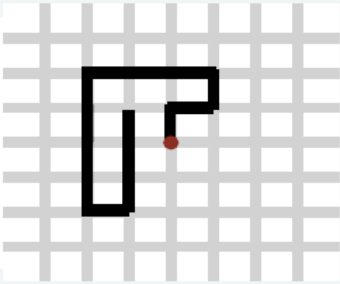
```
% java SelfAvoidingWalker 10 100000
5% dead ends

% java SelfAvoidingWalker 20 100000
32% dead ends

% java SelfAvoidingWalker 30 100000
58% dead ends

% java SelfAvoidingWalker 40 100000
77% dead ends

% java SelfAvoidingWalker 50 100000
87% dead ends

% java SelfAvoidingWalker 60 100000
93% dead ends

% java SelfAvoidingWalker 70 100000
96% dead ends

% java SelfAvoidingWalker 80 100000
98% dead ends

% java SelfAvoidingWalker 90 100000
99% dead ends

% java SelfAvoidingWalker 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.

Even higher-dimensional arrays

3D Arrays: Scientific simulations (modeling 3D space with x, y, z coordinates)

```
// 3D temperature field
double[][][] temperature = new double[100][100][50];
```

4D Arrays: Video processing (width, height, time, color channels)

```
// RGB video frames over time
int[][][][] video = new int[1920][1080][3600][3];
```

> **i** Higher-dimensional arrays are useful for representing complex data structures in scientific computing, graphics, and machine learning.

Imperative programming

Assignments, conditionals, loops, and arrays are the basic blocks of simple imperative programming.

You are now a programmer!

Epilogue

Compiler Error of the Week (1/2)

What is the problem here?:

```
int[] arr = {0.0, 1.0, 2.0};
int result = 0;
for (int i = 0; i < 3; i++) {
    result += arr[i];
}
System.out.println(result);
```

Compiler Error of the Week (1/2)

What is the problem here?:

```
int[] arr = {0.0, 1.0, 2.0};
int result = 0;
for (int i = 0; i < 3; i++) {
    result += arr[i];
}
System.out.println(result);
```

```
java: incompatible types: possible lossy conversion from double to int
```

We expected `arr` to be an `int` array, but we provided a `double` array.

Compiler Error of the Week (2/2)

What is the problem here?:

```
int[] arr = {0, 3, 5, 4, 2};
int result = 0;
for (double i = 0.0; i < 5.0; i += 1.0) {
    result += arr[i];
}
System.out.println(result);
```

Compiler Error of the Week (2/2)

What is the problem here?:

```
int[] arr = {0, 3, 5, 4, 2};
int result = 0;
for (double i = 0.0; i < 5.0; i += 1.0) {
    result += arr[i];
}
System.out.println(result);
```

```
java: incompatible types: possible lossy conversion from double to int
```

The index of an array should be an `int` but we provided a `double`.

Live Programming

Live programming

- Array addition, multiplication, transpose
- Birthday Paradox
- Monty Hall Problem
- AIOOB and Large Arrays

Sources for images and slides

- https://introcs.cs.princeton.edu/java/lectures/
- https://commons.wikimedia.org/wiki/File:Rainbow_colors_(red,_orange,_yellow,_green,_blue,_indigo,_violet).png
- https://commons.wikimedia.org/wiki/File:Enron_closing_stock,_1997-2002.svg
- https://commons.wikimedia.org/wiki/File:Sergei_Eisenstein._The_Queue.jpg
- https://commons.wikimedia.org/wiki/File:Shopping_list_20170612.jpg
- https://brilliant.org/wiki/monty-hall-problem/
- https://commons.wikimedia.org/wiki/File:Sieve_of_Eratosthenes_animation.gif
- https://commons.wikimedia.org/wiki/File:CRT_pixels_close-up.JPG
- https://commons.wikimedia.org/wiki/File:Bristol_Hippodrome_Auditorium_Seating.jpg
- https://commons.wikimedia.org/wiki/File:Chess_Board.svg
- https://commons.wikimedia.org/wiki/File:Closeup_of_Excel_Spreadsheet_template_to_track_printouts_(29911005444).jpg
- knowyourmeme.com/memes/brent-rambo