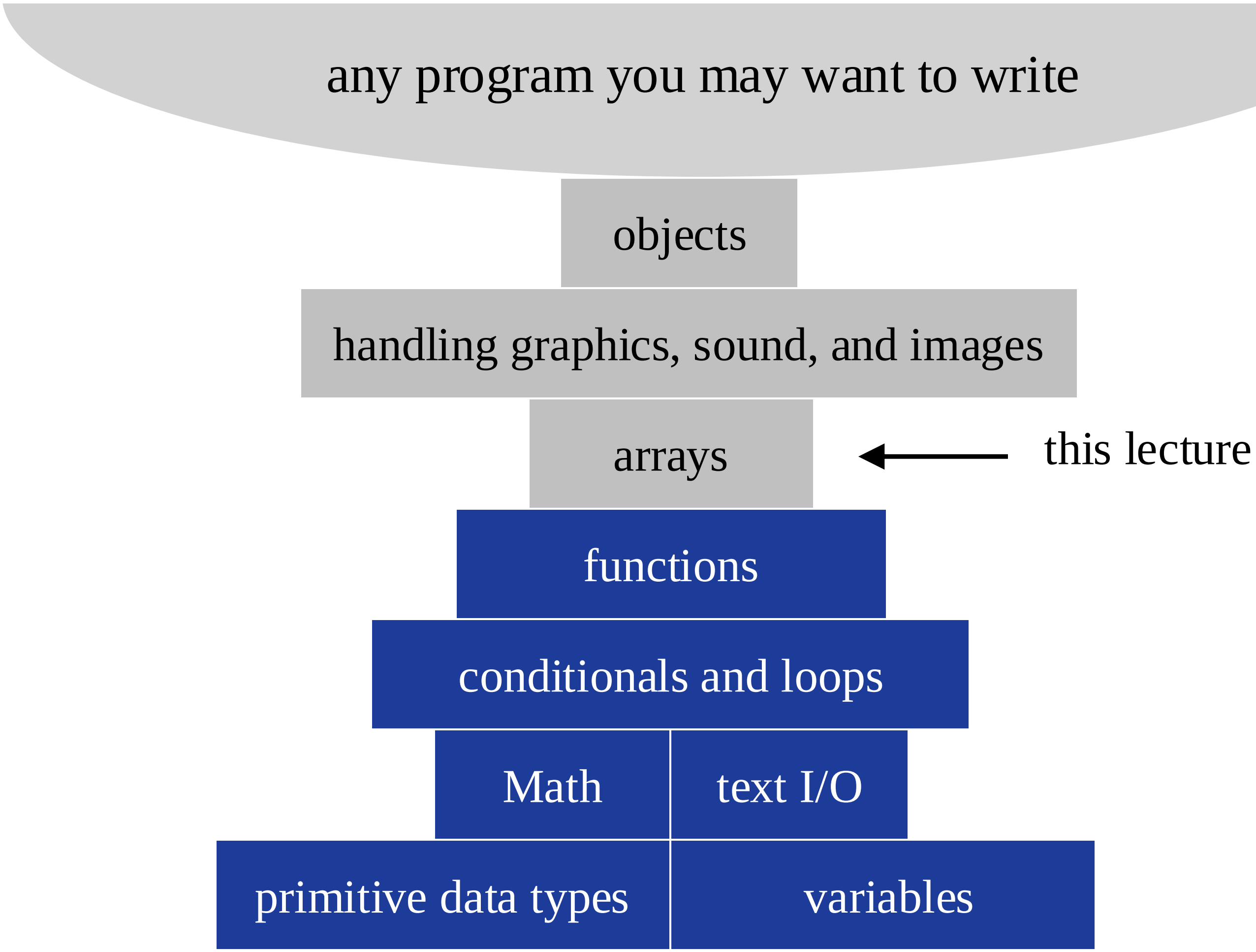
Introduction to Computer Science

Reichman University

Lecture 4-1

Arrays, Part I

The big picture



any program you may want to write

objects

handling graphics, sound, and images

arrays

this lecture

functions

conditionals and loops

Math

text I/O

primitive data types

variables

Arrays

	0	1	2	3	4	5	6	7	8	9	...
`dna` array:	A	C	A	C	G	G	T	C	G	T	...

Purpose: Storing and processing a *sequence of values*

Examples:

- 50,000 letters in a DNA segment
- 1,000 stock prices
- 100,000 common English words
- 300 students enrolled in a course
- Etc.

Array: Data set of a fixed size, stored in the computer’s main memory (RAM)

Arrays

	0	1	2	3	4	5	6	7	8	9	…
`dna` array:	A	C	A	C	G	G	T	C	G	T	…

<u>Typical queries</u>

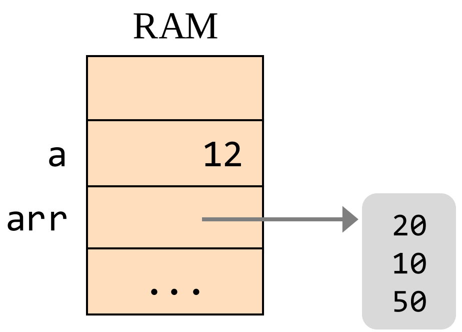
- Which value appears in location 512 ?
- What is the location of the first / last occurrence of `G` ?
- How many times `T` appears in the array?
- Does the pattern `ATG` appear in the array?
- Does the pattern `C?T` appear (where `?` is any character) ?
- Given two arrays `dna1` and `dna2` of the same length,
  in what percentage of the locations `dna1` and `dna2` have identical letters?
- …

Array variables

<u>Abstraction</u> (Java)

```
// a:  a primitive variable
int a = 12;

// arr:  an array variable
int[] arr = {20, 10, 50};
```

<u>Implementation</u>



(stored somewhere else in the RAM)

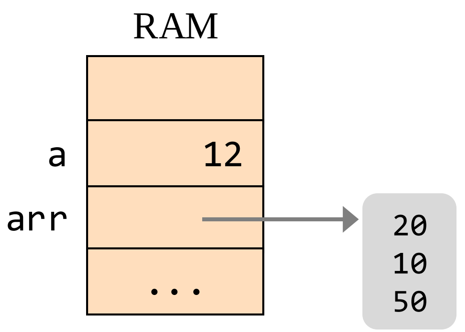
- Variables that have primitive types (like `int`) store *values*
- Variables that have array types (like `int[]`) store *addresses* in memory

Array variables

Abstraction (Java)

```
// a:  a primitive variable
int a = 12;

// arr:  an array variable
int[] arr = {20, 10, 50};
```

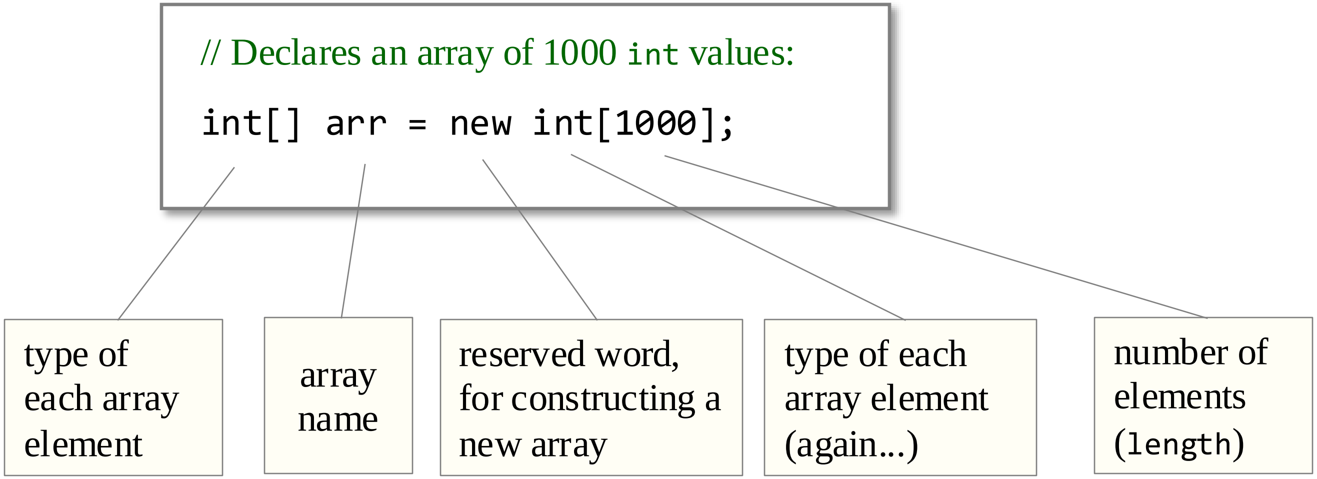
Implementation



(stored somewhere else in the RAM)

- Variables that have primitive types (like `int`) store *values*
- Variables that have array types (like `int[]`) store *addresses* in memory
- That’s why array variables are sometimes called:
  - ○ *reference variables*
  - ○ *references*
  - ○ *pointers.*

Array construction



```
// Declares an array of 1000 int values:

int[] arr = new int[1000];
```

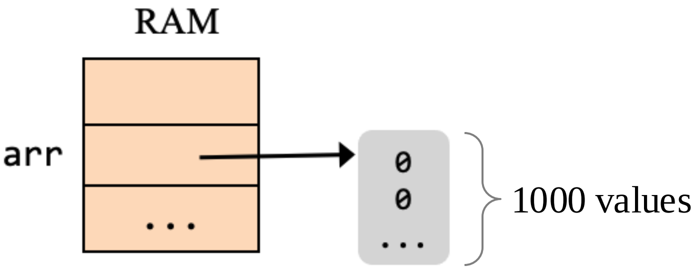
type of each array element

array name

reserved word, for constructing a new array

type of each array element (again...)

number of elements (`length`)



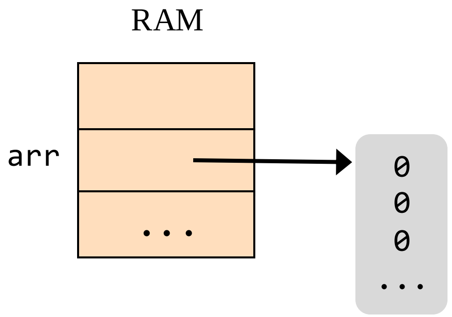
Array elements are initialized according to the array data type:

- `int`, `long`, `char`: `0`
- `double`: `0.0`
- `boolean`: `false`

Array construction: three versions / options

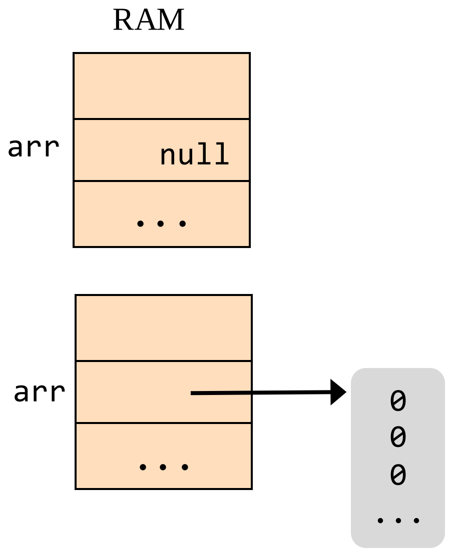
One-stage declaration and construction:

```
// Declares a 1000-element array, initialized with 0's:
int[] arr = new int[1000];
```



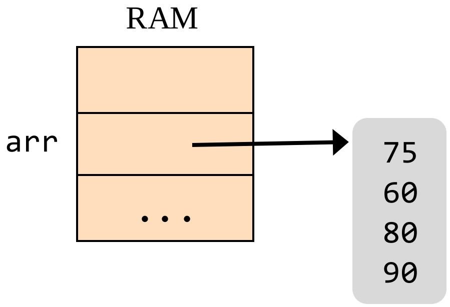
Declare first, construct later:

```
// Declares a reference variable, initialized to null:
int[] arr;
...

// Later in the program …
// Constructs the array, and makes the variable arr refer to it:
arr = new int[1000];
...
```



One-stage declaration, construction, and initialization:

```
// Declares a 5-element array, and initializes it with values:
int[] arr = {75, 60, 80, 60, 90};
```



Array processing example: DNA

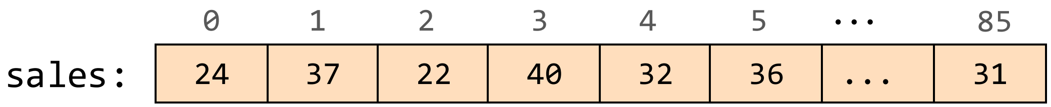
	0	1	2	3	4	5	6	7	8	9	…
dna array:	A	C	A	C	G	G	T	C	G	T	…

```
// Normally, we’ll read the DNA data from a file.
// For testing purposes, we often use a small example:
char[] dna = {'A','C','A','C','G','G','T','C','G','T'};

// Which base appears in location 3?
System.out.println(dna[3]);  // prints C

// Mutation
dna[1] = 'G';

// Mutation: switches bases 2 and 3
char temp = dna[2];
dna[2] = dna[3];
dna[3] = temp;

// Prints the array
for (int i = 0; i < dna.length; i++) {
    System.out.print(dna[i] + " ");
}
```

```
C

A G C A G G T C G T
```

Each array has a `length` property that holds how many elements the array has

Array processing example: Sales reporting



	0	1	2	3	4	5	...	85
sales:	24	37	22	40	32	36	...	31

sales of coffee machines
in 86 regions

```
public class ArrayDemo {
   public static void main(String[] args) {

      // Builds a small array, for testing purposes
      int[] sales = {24, 37, 22, 40, 32, 36};
      ...
      // Computes and prints the sales average
      int sum = 0;
      for (int i = 0; i < sales.length; i++) {
           sum += sales[i];  // sum = sum + sales[i];
      }
      System.out.println("The sales average is " + sum / sales.length);
      ...
```

```
% java ArrayDemo

The sales average is 31
```

Array processing example: Sales reporting

	0	1	2	3	4	5	...	85
sales:	24	37	22	40	32	36	...	31

sales of coffee machines
in 86 regions

```
public class ArrayDemo {
   public static void main(String[] args) {

       // Builds a small array, for testing purposes
       int[] sales = {24, 37, 22, 40, 32, 36};
       ...
       // Prints the largest sales figure
       int max = sales[0];
       for (int i = 0; i < sales.length; i++) {
           if (sales[i] > max) {
              max = sales[i];
           }
       }
       System.out.println("Largest sale figure: " + max);
       ...
```

```
...
Largest sales figure: 40
```

Array processing example: Sales reporting

	0	1	2	3	4	5	...	85
sales:	24	37	22	40	32	36	...	31

sales of coffee machines
in 86 regions

```
public class ArrayDemo {
   public static void main(String[] args) {

       // Builds a small array, for testing purposes
       int[] sales = {24, 37, 22, 40, 32, 36};

       ...

       // Prints the least sales figure, and its index
       int min = sales[0];
       int minIndex = 0;
       for (int i = 0; i < sales.length; i++) {
          if (sales[i] < min) {
              min = sales[i];
              minIndex = i;
          }
        }
        System.out.println("Region " + minIndex +
                           " had the least sales, with " + min + " units sold");

        ...
```

```
...
Region 2 had the least sales, with 22 units sold
```

Array processing example: Sales reporting

	0	1	2	3	4	5	...	85
sales:	24	37	22	40	32	36	...	31

sales of coffee machines
in 86 regions

```
public class ArrayDemo {
   public static void main(String[] args) {

       // Builds a small array, for testing purposes
       int[] sales = {24, 37, 22, 40, 32, 36};

       ...

       // Increases all sales by 10%
       for (int i = 0; i < sales.length; i++) {
           sales[i] = (int) (sales[i] * 1.1);
       }

       // Prints all the sales
       for (int i = 0; i < sales.length; i++) {
           System.out.print(sales[i] + " ");
       }
       ...
```

```
...

26 40 24 44 35 39
```

A library of array processing functions

```
public class MyArrays {
   public static void main(String[] args) {
      int[] x = {5, 3, 2}; // for testing
      println(x);
      System.out.println(sum(x));
      System.out.println(average(x));
   }

   /** Returns the sum of the elements of the array */
   public static int sum(int[] arr) {
      return 0;
   }

    /** Returns the average of the elements of the array */
    public static double average(int[] arr) {
       return 0;
    }

    /** Prints the array, and then a new line */
    public static void println(int[] arr) {
    }

    // More array functions...
}
```

Executable class skeleton

Common array processing tasks

- Sum
- Average
- Min / max
- Print
- Reverse
- etc.

It makes sense to build a library that features these services to any array

```
% java MyArrays
(Println will print nothing)
0
0
```

A library of array processing functions

```
public class MyArrays {
   public static void main(String[] args) {
      int[] x = {5, 3, 2}; // for testing
      println(x);
      System.out.println(sum(x));
      System.out.println(average(x));
   }

   /** Returns the sum of the elements of the array */
   public static int sum(int[] arr) {
      int sum = 0;
      for (int i = 0; i < arr.length; i++) {
         sum = sum + arr[i];
      }
      return sum;
    }

    /** Returns the average of the elements of the array */
    public static double average(int[] arr) {
       return ((double) sum(arr)) / arr.length);
    }

    /** Prints the array, and then a new line */
    public static void println(int[] arr) {
       for (int i = 0; i < arr.length; i++) {
          System.out.print(arr[i] + " ");
       }
       System.out.println();
    }
}
```

<u>Common array processing tasks</u>

- Sum
- Average
- Min / max
- Print
- Reverse
- etc.

```
% java MyArrays
5 3 2
10
3.3333333333333335
```

Arrays, part I

- ✓ Basic concepts
- ✓ Array processing examples
- ➡ Mutability
- More array processing examples
  - ➢ Letter frequency
  - ➢ Monte Carlo simulation
  - ➢ Reversing an array
- Side effects

Mutable / Immutable

Mutable object

The object's state can be changed

(example: whiteboard)

Immutable object

The object state cannot be changed

(example: sent email)

In programming:

Some variables are *mutable*, some are *immutable* – it depends both on the variable and on the context in which it is used.

Mutable / Immutable

```
public class MutateDemo {
   public static void main(String args[]) {
      char   a1 = 'm';
      char[] a2 = {'m', '&', 'm'};

      System.out.println(a1);  mutate1(a1);  System.out.println(a1);

      println(a2);             mutate2(a2);  println(a2);

   }

   public static void mutate1(char x) {
      x = 'b';
      System.out.println(x);
   }

   public static void mutate2(char[] x) {
      x[0] = 'b'; x[2] = 'b';
      println(x);
   }


   /** Prints the array, and then a new line */
   public static void println(char[] arr) {
      // See previous slides
   }
}
```

Meaningless functions, designed to demo when a function can, or cannot, change the arguments passed to it

Mutable / Immutable

```java
public class MutateDemo {
   public static void main(String args[]) {
      char   a1 = 'm';
      char[] a2 = {'m', '&', 'm'};

      System.out.println(a1);  mutate1(a1);  System.out.println(a1);
      println(a2);             mutate2(a2);  println(a2);
   }

   public static void mutate1(char x) {
      x = 'b';
      System.out.println(x);
   }

   public static void mutate2(char[] x) {
      x[0] = 'b'; x[2] = 'b';
      println(x);
   }


   /** Prints the array, and then a new line */
   public static void println(char[] arr) {
      // See previous slides
   }
}
```

```
% java MutateDemo
m
b
m      (unchanged)
```

Explanation

When passing a *primitive variable* to a function, what is being passed is not the variable, but the variable's *value;*

`mutate1` has no access to the variable; It cannot change it.

"Call by value".

Mutable / Immutable

```
public class MutateDemo {
   public static void main(String args[]) {
      char   a1 = 'm';
      char[] a2 = {'m', '&', 'm'};

      System.out.println(a1);  mutate1(a1);  System.out.println(a1);
      println(a2);             mutate2(a2);  println(a2);
   }

   public static void mutate1(char x) {
      x = 'b';
      System.out.println(x);
   }

   public static void mutate2(char[] x) {
      x[0] = 'b'; x[2] = 'b';
      println(x);
   }


   /** Prints the array, and then a new line */
   public static void println(char[] arr) {
      // See previous slides
   }
}
```

```
% java MutateDemo
m
b
m     (unchanged)
m&m
b&b
b&b   (changed)
```

<u>Explanation</u>

When passing an *array variable* to a function, what is being passed is the *reference* (base address of the array in memory)

`mutate2` has access to the array elements;
it can change them.

“Call by reference”.

Arrays, part I

✓ Basic concepts

✓ Array processing examples

✓ Mutability

- More array processing examples
  - ➡ Letter frequency
  - ➢ Monte Carlo simulation
  - ➢ Reversing an array
- Side effects

Letter frequency

```
// Computes the frequency of the characters A, T, G and C in a given DNA string
public class CharCount1 {
   public static void main(String args[]) {
```

```
% java CharCount1 AATTTGCATTC
A appears 3 times
T appears 5 times
G appears 1 times
C appears 2 times
```

Letter frequency

```
// Computes the frequency of the characters A, T, G and C in a given DNA string
public class CharCount1 {
   public static void main(String args[]) {
      String str = args[0];
      char[] bases = {'A','T','G','C'};
      int[] freq = new int[bases.length];
```

bases:

0	1	2	3
A	T	G	C

freq:

0	1	2	3
0	0	0	0

Algorithm

```
for i = 0 … str.length
   for j = 0 … bases.length
      if str[i] = bases[j]
         freq[j]++
```

```
% java CharCount1 AATTTGCATTC
A appears 3 times
T appears 5 times
G appears 1 times
C appears 2 times
```

Letter frequency

```
// Computes the frequency of the characters A, T, G and C in a given DNA string
public class CharCount1 {
   public static void main(String args[]) {
      String str = args[0];
      char[] bases = {'A','T','G','C'};
      int[] freq = new int[bases.length];

      // Scans the string and updates frequency counters
      for (int i = 0; i < str.length(); i++) {
         for (int j = 0; j < bases.length; j++) {
            if (str.charAt(i) == bases[j]) {
               freq[j]++;
            }
         }
      }

      // Prints the frequency counters
      for (int i = 0; i < freq.length; i++) {
         System.out.println(bases[i] + " appears " + freq[i] + " times");
      }
   }
}
```

```
        0   1   2   3
bases: [A | T | G | C]

        0   1   2   3
 freq: [0 | 0 | 0 | 0]
```

Notice the syntax difference between accessing the `length` and elements of a *string*, and accessing the `length` and elements of an *array*

```
% java CharCount1 AATTTGCATTC
A appears 3 times
T appears 5 times
G appears 1 times
C appears 2 times
```

Letter frequency

```
// Computes the frequency of the characters A, T, G and C in a given DNA string
public class CharCount2 {
   public static void main(String args[]) {
      String str = args[0];
      int[] freq = new int[4];

      // Scans the string; for each character, if the character
      // appears in "ATGC", increments its frequency counter.
      for (int i = 0; i < str.length(); i++) {
          freq["ATGC".indexOf(str.charAt(i))]++;
      }

      // Prints the frequency results (Same as previous slide)
      ...
      }
   }
}
```

another approach

<u>This solution</u>

- Less code
- Same efficiency: (`indexOf` also uses a loop)
- Less readable

```
% java CharCount1 AATTTGCATTC
A appears 3 times
T appears 5 times
G appears 1 times
C appears 2 times
```

Letter frequency

```
// Computes the frequency of the characters A, T, G and C in a given DNA string
public class CharCount2 {
   public static void main(String args[]) {
      String str = args[0];
      int[] freq = new int[4];

      // Scans the string; for each character, if the character
      // appears in "ATGC", increments its frequency counter.
      for (int i = 0; i < str.length(); i++) {
          freq["ATGC".indexOf(str.charAt(i))]++;
      }

      // Prints the frequency results (Same as previous slide)
      ...
      }
   }
}
```

another approach

This solution

- Less code
- Same efficiency: (`indexOf` also uses a loop)
- Less readable

Third solution (self exercise)

Use four counter variables. (less fancy, more efficient, more readable).

Arrays, part I

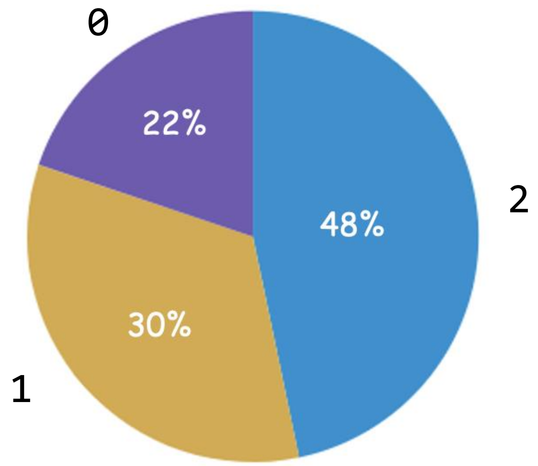
- Basic concepts
- Array processing examples
- Mutability
- More array processing examples
  - ➢ Letter frequency
  - ➡ Monte Carlo simulation
  - ➢ Reversing an array
- Side effects

Monte Carlo simulation

Monte Carlo simulation:

Generating pseudo-random values from
a given Probability Density Function

Monte Carlo simulation: Example



One of three possible events (0, 1, 2) happens, randomly:

0 occurs 0.22 of the time

1 occurs 0.30 of the times

2 occurs 0.48 of the times

Task: Generate *N* events from this probability distribution

```
% java MyRandom 100000

0 occurred 0.219901 of the time
1 occurred 0.300052 of the time
2 occurred 0.480047 of the time
```

Probability Density Functions

The setting

- N possible and mutually-exclusive events can happen
- The events are denoted 0, 1, 2, 3, ..., N-1
- Each event occurs with a given probability

Probability Density Function (PDF)

$$\left.\begin{array}{l} p(0) = .1 \\ p(1) = .3 \\ p(2) = .5 \\ p(3) = .1 \end{array}\right\}$$

example of a PDF
describing the likelihood
of N = 4 possible events

Cumulative Distribution Function (CDF)

$P(0) = p(0) =$ .1

$P(1) = p(0 \text{ or } 1) =$ .1 + .3 = .4

$P(2) = p(0 \text{ or } 1 \text{ or } 2) =$ .1 + .3 + .5 = .9

$P(3) = p(0 \text{ or } 1 \text{ or } 2 \text{ or } 3) =$ .1 + .3 + .5 + .1 = 1.0

```
/** Creates a CDF from a given PDF */
public static double[] CDF(double[] p) {
    double[] P = new double[p.length];
    P[0] = p[0];
    for (int i = 1; i < p.length; i++)  {
        P[i] = P[i-1] + p[i];
    }
    return P;
}
```

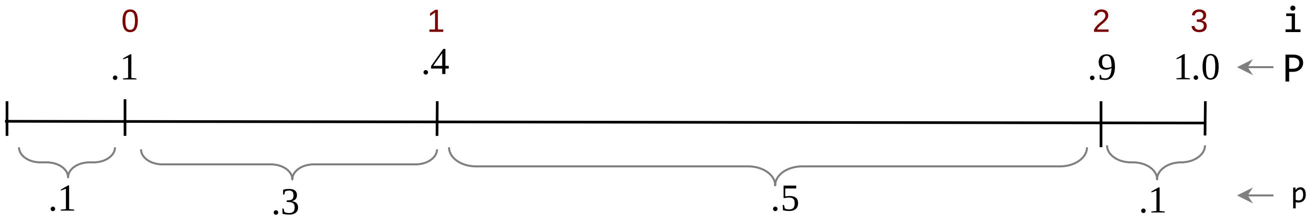
Generating pseudo-random values from a given distribution

Task: Generate events that have a given probability

Example: Generate values from {0,1,2,3} where $p(0)=.1$, $p(1)=.3$, $p(2)=.5$, $p(3)=.1$

Method: 0. Given: $p = \begin{array}{cccc} 0 & 1 & 2 & 3 \\ \hline .1 & .3 & .5 & .1 \end{array}$

1. Compute the CDF:



0 1 2 3 i

.1 .4 .9 1.0 ← P

.1 .3 .5 .1 ← p

2. Generate a random number r in the range [0,1)

3. for $i = 0, \ldots, N-1$: if $(r < P[i])$ return i

```
/** Generates a random integer 0,1,...n-1 from a given CDF of size n. */
public static int rnd(double[] P) {
    // draws a random number in [0,1), and returns where it falls in the CDF
    double r = Math.random();
    for (int i = 0; i < P.length; i++)  {
        if (r <= P[i]) return i;
    }
    return 0;   // compilation requirement
}
```

Generating pseudo-random values from a given distribution: Testing

```java
public class MyRandom {
    public static void main(String args[]) {
        // Tests the CDF and rnd functions by generating events and observing their actual distribution
        // The array p represents a Probability Density Function (PDF):
        // p[i] represents the probability that event i occurs.
        double[] p = {.2, .2, .6};
```

```
% java MyRandom 100000
Expected distribution:
0 should occur 0.2 of the time
1 should occur 0.2 of the time
2 should occur 0.6 of the time
Actual distribution after 100000 trials:
0 occurred 0.200039 of the time
1 occurred 0.199882 of the time
2 occurred 0.600079 of the time
```

Generating pseudo-random values from a given distribution: Testing

```
public class MyRandom {
    public static void main(String args[]) {
        // Tests the CDF and rnd functions by generating events and observing their actual distribution
        // The array p represents a Probability Density Function (PDF):
        // p[i] represents the probability that event i occurs.
        double[] p = {.2, .2, .6};

        //// Rest of main function code... Next slide

    /** Creates and returns a Cumulative Distribution Function from a given distribution function. */
    public static double[] CDF(double[] p) {
        double[] P = new double[p.length];
        P[0] = p[0];
        for (int i = 1; i < p.length; i++)  {
            P[i] = P[i-1] + p[i];
        }
        return P;
    }

    /** Generates a random integer 0,1,...n-1 from a given CDF of size n. */
    public static int rnd(double[] P) {
        // draws a random number in [0,1),
        // and returns where it falls in the CDF
        double r = Math.random();
        for (int i = 0; i < P.length; i++)  {
            if (r <= P[i]) return i;
        }
        return 0;   // compilation requirement
    }

    // More random functions can come here (serving various needs)

}
```

```
% java MyRandom 100000

Expected distribution:

0 should occur 0.2 of the time
1 should occur 0.2 of the time
2 should occur 0.6 of the time

Actual distribution after 100000 trials:

0 occurred 0.200039 of the time
1 occurred 0.199882 of the time
2 occurred 0.600079 of the time
```

Generating pseudo-random values from a given distribution: Testing

```
public class MyRandom {
    public static void main(String args[]) {
        // Tests the CDF and rnd functions by generating events and observing their actual distribution
        // The array p represents a Probability Density Function (PDF):
        // p[i] represents the probability that event i occurs.
        double[] p = {.2, .2, .6};
```

```
% java MyRandom 100000
Expected distribution:
0 should occur 0.2 of the time
1 should occur 0.2 of the time
2 should occur 0.6 of the time
Actual distribution after 100000 trials:
0 occurred 0.200039 of the time
1 occurred 0.199882 of the time
2 occurred 0.600079 of the time
```

Generating pseudo-random values from a given distribution: Testing

```
public class MyRandom {
    public static void main(String args[]) {
        // Tests the CDF and rnd functions by generating events and observing their actual distribution
        // The array p represents a Probability Density Function (PDF):
        // p[i] represents the probability that event i occurs.
        double[] p = {.2, .2, .6};

        // Prints the probability distribution
        System.out.println("Expected distribution:\n");
        for (int i = 0; i < p.length; i++) {
            System.out.println(i + " should occur " + p[i] + " of the time");
        }

        // Number of trials
        int T = Integer.parseInt(args[0]);
        // Stores how many times each event occurred
        int[] count = new int[p.length];

        // Creates the Cumulative Distribution Function of p
        double[] P = CDF(p);

        // Generates T random values, and counts how many times each value occurred.
        for (int t = 0; t < T; t++) {
            count[rnd(P)]++;
        }

        System.out.println("\nActual distribution after " + T + " trials:\n");
        for (int i = 0; i < count.length; i++) {
            System.out.println(i + " occurred " +  ((double) count[i] / T) + " of the time");
    } //// Class code continues with the CDF and rnd functions (previous slide)
```

```
% java MyRandom 100000
Expected distribution:

0 should occur 0.2 of the time
1 should occur 0.2 of the time
2 should occur 0.6 of the time

Actual distribution after 100000 trials:

0 occurred 0.200039 of the time
1 occurred 0.199882 of the time
2 occurred 0.600079 of the time
```

The law of large numbers

```
% java MyRandom 10

Expected distribution:

0 should occur 0.2 of the time
1 should occur 0.2 of the time
2 should occur 0.6 of the time

Actual distribution after 10 trials:

0 occurred 0.3 of the time
1 occurred 0.0 of the time
2 occurred 0.7 of the time
```

```
% java MyRandom 20

Expected distribution:

0 should occur 0.2 of the time
1 should occur 0.2 of the time
2 should occur 0.6 of the time

Actual distribution after 20 trials:

0 occurred 0.35 of the time
1 occurred 0.05 of the time
2 occurred 0.6 of the time
```

```
% java MyRandom 100

Expected distribution:

0 should occur 0.2 of the time
1 should occur 0.2 of the time
2 should occur 0.6 of the time

Actual distribution after 100 trials:

0 occurred 0.23 of the time
1 occurred 0.18 of the time
2 occurred 0.59 of the time
```

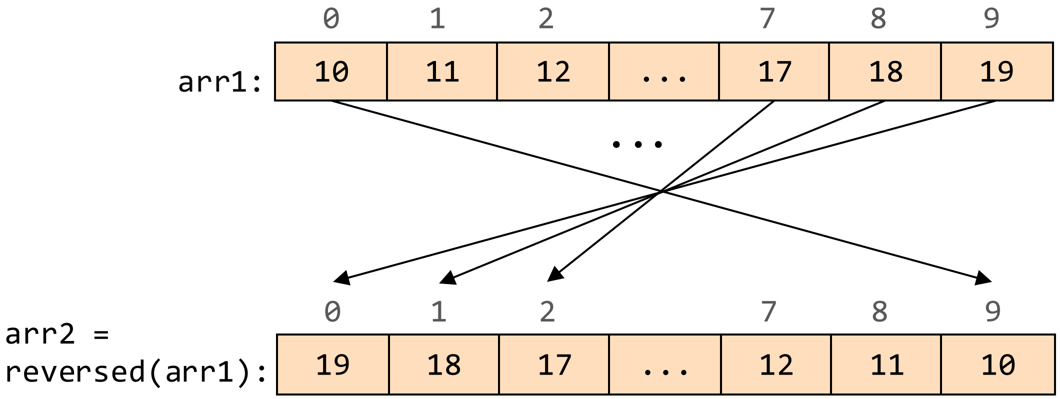
```
% java MyRandom 100000

Expected distribution:

0 should occur 0.2 of the time
1 should occur 0.2 of the time
2 should occur 0.6 of the time

Actual distribution after 100000 trials:

0 occurred 0.200039 of the time
1 occurred 0.199882 of the time
2 occurred 0.600079 of the time
```

Law of large numbers: As we increase the number of independent trials,
the average of the results gets closer to the expected average.

Arrays, part I

- Basic concepts
- Array processing examples
- Mutability
- More array processing examples
    - ➢ Letter frequency
    - ➢ Monte Carlo simulation
    - ➡ Reversing an array
- Side effects

Reversing an array



arr1:

0	1	2		7	8	9
10	11	12	...	17	18	19

arr2 = reversed(arr1):

0	1	2		7	8	9
19	18	17	...	12	11	10

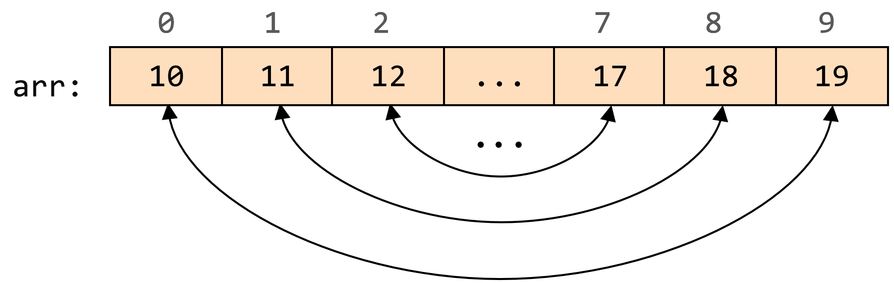
We store the result in a new array

Algorithm:

```
arr2[0] = arr1[9]
arr2[1] = arr1[8]
arr2[2] = arr1[7]
...
arr2[i] = arr1[N – i - 1]   (N = arr1.length)
...
```

Do this as long as `i < N`

Reversing an array (in place)



	0	1	2		7	8	9
arr:	10	11	12	...	17	18	19

	0	1	2		7	8	9
arr:	19	18	17	...	12	11	10

We store the result in the *original array*

Algorithm:

Switch the values of `arr[0]` and `arr[9]`

Switch the values of `arr[1]` and `arr[8]`

Switch the values of `arr[2]` and `arr[7]`

...

Switch the values of `arr[i]` and `arr[N – i - 1]` `(N = arr.length)`

...

Do this as long as `i < N / 2`

```
// Switches the values of array elements:
int temp = arr[i];
arr[i] = arr[j];
arr[j] = temp;
```

Reversing an array

```
public class MyArrays {
   public static void main(String[] args) {
      int[] x = {5, 3, 2}; // for testing
      println(x); // uses an array printing function






   /** Returns an array which is the reverse of the given array */
   public static int[] reversed(int[] arr) {








   /** Reverses the order of elements in the given array.
    *  Side effect: the original array is mutated. */
   public static void reverseInPlace(int[] arr) {







}
```

```
% java MyArrays
5 3 2  (x)
```

Returns a new array, containing the elements of the given array, reversed

Reverses the given array

Reversing an array

```
public class MyArrays {
   public static void main(String[] args) {
      int[] x = {5, 3, 2};
      println(x);
      println(reversed(x));
      println(x);
      reverseInPlace(x);
      println(x);
   }

   /** Returns an array which is the reverse of the given array */
   public static int[] reversed(int[] arr) {









   /** Reverses the order of elements in the given array.
    *  Side effect: the original array is mutated. */
   public static void reverseInPlace(int[] arr) {









}
```

```
% java MyArrays
5 3 2  (x)
2 3 5   (values of x, reversed)
5 3 2   (original x hasn’t changed)
2 3 5   (original x has changed)
```

Returns a new array, containing the elements of the given array, reversed

Reverses the given array

Reversing an array

```
public class MyArrays {
   public static void main(String[] args) {
      int[] x = {5, 3, 2};
      println(x);
      println(reversed(x));
      println(x);
      reverseInPlace(x);
      println(x);
   }

   /** Returns an array which is the reverse of the given array */
  public static int[] reversed(int[] arr) {









   /** Reverses the order of elements in the given array.
    *  Side effect: the original array is mutated. */
   public static void reverseInPlace(int[] arr) {









}
```

```
% java MyArrays
5 3 2  (x)
2 3 5   (values of x, reversed)
5 3 2   (original x hasn’t changed)
2 3 5   (original x has changed)
```

Returns a new array, containing the elements of the given array, reversed

Reverses the given array

Reversing an array

```
public class MyArrays {
   public static void main(String[] args) {
      int[] x = {5, 3, 2};
      println(x);
      println(reversed(x));
      println(x);
      reverseInPlace(x);
      println(x);
   }

   /** Returns an array which is the reverse of the given array */
  public static int[] reversed(int[] arr) {







   /** Reverses the order of elements in the given array.
    *  Side effect: the original array is mutated. */
   public static void reverseInPlace(int[] arr) {







}
```

```
% java MyArrays
5 3 2  (x)
2 3 5   (values of x, reversed)
5 3 2   (original x hasn't changed)
2 3 5   (original x has changed)
```

Returns a new array, containing the elements of the given array, reversed

Reverses the given array

Reversing an array

```
public class MyArrays {
   public static void main(String[] args) {
      int[] x = {5, 3, 2};
      println(x);
      println(reversed(x));
      println(x);
      reverseInPlace(x);
      println(x);
   }

   /** Returns an array which is the reverse of the given array */
  public static int[] reversed(int[] arr) {








   /** Reverses the order of elements in the given array.
    *  Side effect: the original array is mutated. */
   public static void reverseInPlace(int[] arr) {







}
```

```
% java MyArrays
5 3 2  (x)
2 3 5   (values of x, reversed)
5 3 2   (original x hasn’t changed)
2 3 5   (original x has changed)
```

Returns a new array, containing the elements of the given array, reversed

Reverses the given array

Reversing an array

```
public class MyArrays {
   public static void main(String[] args) {
      int[] x = {5, 3, 2};
      println(x);
      println(reversed(x));
      println(x);
      reverseInPlace(x);
      println(x);
   }

   /** Returns an array which is the reverse of the given array */
   public static int[] reversed(int[] arr) {
      int N = arr.length;
      int[] reversed = new int[N];
      for (int i = 0; i < N; i++) {
          reversed[i] = arr[N – i – 1];
      }
      return reversed; // returns the new array
   }

   /** Reverses the order of elements in the given array.
    *  Side effect: the original array is mutated. */
   public static void reverseInPlace(int[] arr) {
      int N = arr.length;
      for (int i = 0; i < N / 2; i++) {
         int temp = arr[i];
         arr[i] = arr[N – i – 1];
         arr[N – i – 1] = temp;
      }
   }
}
```

```
% java MyArrays
5 3 2  (x)
2 3 5   (values of x, reversed)
5 3 2   (original x hasn't changed)
2 3 5   (original x has changed)
```

Arrays, part I

- Basic concepts
- Array processing examples
- Mutability
- More array processing examples
  - ➢ Letter frequency
  - ➢ Monte Carlo simulation
  - ➢ Reversing an array

➡ Side effects

Side effects (same as last slide)

```
public class MyArrays {
   public static void main(String[] args) {
      int[] x = {5, 3, 2};
      println(x);
      println(reversed(x));
      println(x);
      reverseInPlace(x);
      println(x);
   }

   /** Returns an array which is the reverse of the given array */
  public static int[] reversed(int[] arr) {
     int N = arr.length;
     int[] reversed = new int[N];
     for (int i = 0; i < N; i++) {
        reversed[i] = arr[N – i – 1];
     }
     return reversed; // returns the new array
  }

  /** Reverses the order of elements in the given array.
   *  Side effect: the original array is mutated. */
  public static void reverseInPlace(int[] arr) {
     int N = arr.length;
     for (int i = 0; i < N / 2; i++) {
        int temp = arr[i];
        arr[i] = arr[N – i – 1];
        arr[N – i – 1] = temp;
     }
  }
}
```

```
% java MyArrays
5 3 2  (x)
2 3 5   (values of x, reversed)
5 3 2   (original x hasn't changed)
2 3 5   (original x has changed)
```

This function *has no side-effects*

This function ***has a side-effect:***

- It changes the given array
- In doing so, it changes the world of the caller (which may well be in another class)
- **Let the caller beware!**

Side effects

```
public class MyArrays {
   public static void main(String[] args) {
      int[] x = {5, 3, 2};
      println(x);
      println(reversed(x));
      println(x);
      reverseInPlace(x);
      println(x);
   }

   /** Returns an array which is the reverse of the given array */
   public static int[] reversed(int[] arr) {
      int N = arr.length;
      int[] reversed = new int[N];
      for (int i = 0; i < N; i++) {
          reversed[i] = arr[N - i - 1];
      }
      return reversed;
   }

   /** Reverses the order of elements in the given array.
    *  Side effect: the original array is mutated. */
   public static void reverseInPlace(int[] arr) {
      int N = arr.length;
      for (int i = 0; i < N / 2; i++) {
         int temp = arr[i];
         arr[i] = arr[N - i - 1];
         arr[N - i - 1] = temp;
      }
   }
}
```

Observations

- Functions can mutate variables in their scope (locals and parameters). This practice is normal, and safe
- Functions can also mutate variables outside their scope (like arrays that are passed as arguments). This practice is possible, but dangerous.

In this example

- The unsafe `reverseInPlace` function is not really needed.
- If a caller (like `main`) wants to reverse an array (say `x`), it can use the code:

  `x = reversed(x);`

  (following this action, x will refer to the address of the new array returned by the function).
- Safest solution:

  `int[] y = reversed(x);`

Side effects

```
public class MyArrays {
   public static void main(String[] args) {
      int[] x = {5, 3, 2};
      println(x);
      println(reversed(x));
      println(x);
      reverseInPlace(x);
      println(x);
   }

   /** Returns an array which is the reverse of the given array */
   public static int[] reversed(int[] arr) {
      int N = arr.length;
      int[] reversed = new int[N];
      for (int i = 0; i < N; i++) {
          reversed[i] = arr[N - i - 1];
      }
      return reversed;
   }

   /** Reverses the order of elements in the given array.
    *  Side effect: the original array is mutated. */
   public static void reverseInPlace(int[] arr) {
      int N = arr.length;
      for (int i = 0; i < N / 2; i++) {
         int temp = arr[i];
         arr[i] = arr[N - i - 1];
         arr[N - i - 1] = temp;
      }
   }
}
```

Observations

- Functions can mutate variables in their scope (locals and parameters).
  This practice is normal, and safe.
- Functions can also mutate variables outside their scope (like arrays that are passed as arguments). This practice is possible, but dangerous.

Best practice

Try to avoid using / writing functions that have side effects

If you have to write a function that has a side-effect:

- Document clearly (in the function API) how the function changes the world of the caller
- Use a function name that describes / informs about its side effect.