Arrays

- From last time: Finish if-else-if
- Ex: count scores
- Example of array uses
- Random access
- Syntax
- Return to counting scores example
- Arrays of objects
- Partially filled arrays
- introduction to ArrayList

Announcements

- PA1 due Wednesday 2/1
- Compiling java programs with multiple classes: **`javac *.java`**
- Students that missed the first lecture, who are not officially enrolled (e.g., on waiting list), or who have no previous programming experience need to see me after class or in office hours today

Multi-way tests

- mutually exclusive conditions:
  - if-else-if construct
  - Don't keep nesting (and indenting):

```
if (cond) {
    action1;
}
else if (cond2) {
   action2;
}
else {
   action3;
}
```

Multi-way test example

- Ex: assign letter grade based on score in course: 90, 80, 70, 60

```
public static char getGrade(int score)
```

More on control structures in the textbook

- The dangling-else problem:

  Common Error 5.3

- DeMorgan's laws:

  Special Topic 5.7

- Hand-tracing:

  Section 6.2

Consider this problem…

- read a bunch of student scores in the range 0-10 and determine how many people got each score…

- Some code to do this in ….

`ScoreCountsHard.java`

What arrays are for

- Can store a *collection* of items of the same type.
- For example:
  - points in an n-sided polygon
  - times of all runners at a track meet
  - distinct words from a story and their frequencies
  - student scores
  - all employees in a department
- also get *random access* . . .

Random access

Examples:

- can go right to track 3 on a CD
- can change individual pixels on a computer monitor
- can access the score for student #4 as fast as student #23
- can solve histogram problem (store a bunch of counts)

Array syntax

```
int[] temps;
```
array reference

```
temps = new int[10];
```
create array object

valid indices are 0 through 9

```
int aTemp = temps[3];
```
access an array elmt

```
temps[3] = 59;
```
change value of array element

```
int temp2 = temps[10];
```
run-time error

```
int len = temps.length;     // 10
```

Using a variable to index an array

```
int aNum = temps[i];
```

- What's a safer way to write this code?

Accessing an array sequentially

Let's print all the values in **`temps`** …

Return to counting scores

- Now we're ready to write better code to solve the counting scores problem:
- read a bunch of student scores and determine how many people got each score…

`ScoreCounts.java`

Arrays of objects

```
String[] names= new String[10];
```

create array of 10 String references

```
int len = names[0].length();
```
run-time error

```
names[0] = "Suzy";
```
now refers to a string object

```
String name = names[10];
```
run-time error

```
len = names[0].length();
```
ok

Elements have default initialization

- **`new Foo[10]`** all initialized to **`null`**
- **`new int[10]`** all initialized to **`0`**
- **`new boolean[10]`** all initialized to **`false`**
- Reminder:
  - Like instance variables
  - locals are **not** initialized by default

Arrays of objects (non-String ex)

- Following creates array – no Rectangle objects:

```
Rectangle[] rectArr = new Rectangle[20];
```

- Create a rectangle:

```
rectArr[0] =
```

Review: applications where we use random access

Characteristics:

- Uses random-access
- Array size known ahead of time and doesn't change

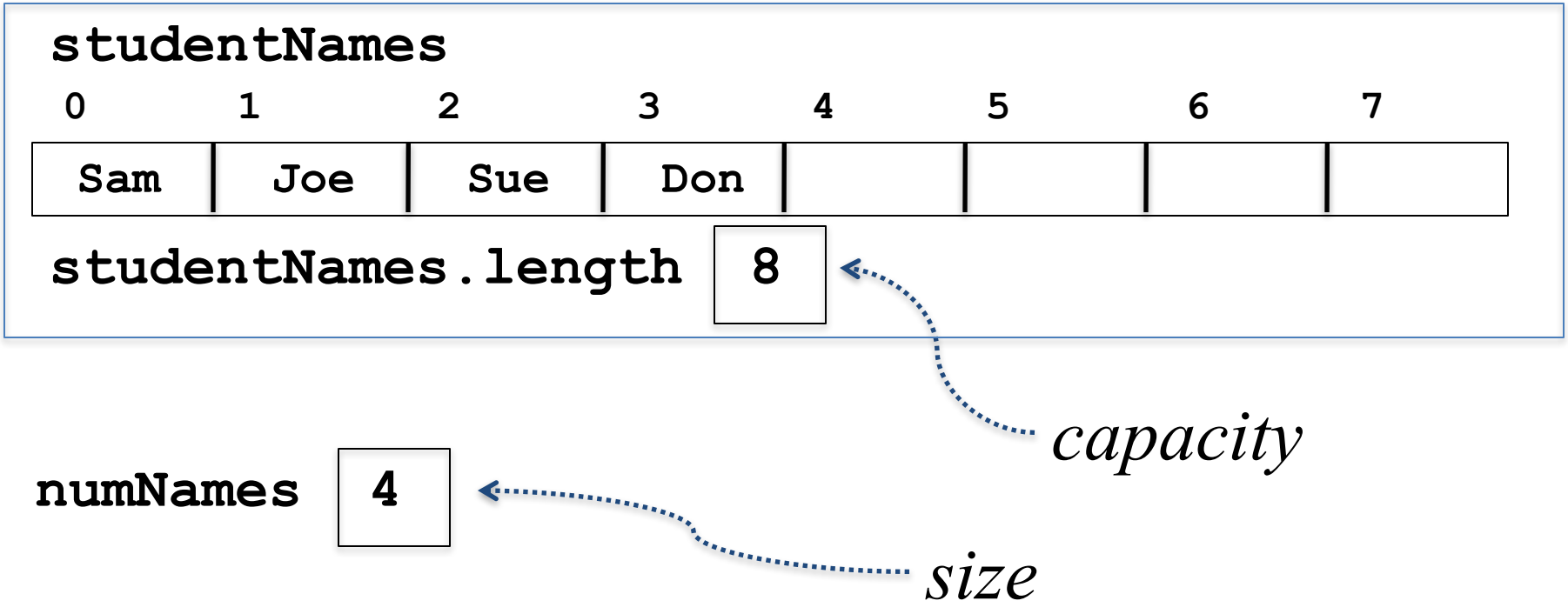
Ex: count how many people got each score (histogram)

Partially filled array

- Ex: store data about all students in the class
- Characteristics…
  - Don't know how many students there will be ahead of time
  - Students may add or drop
  - Uses mostly sequential access

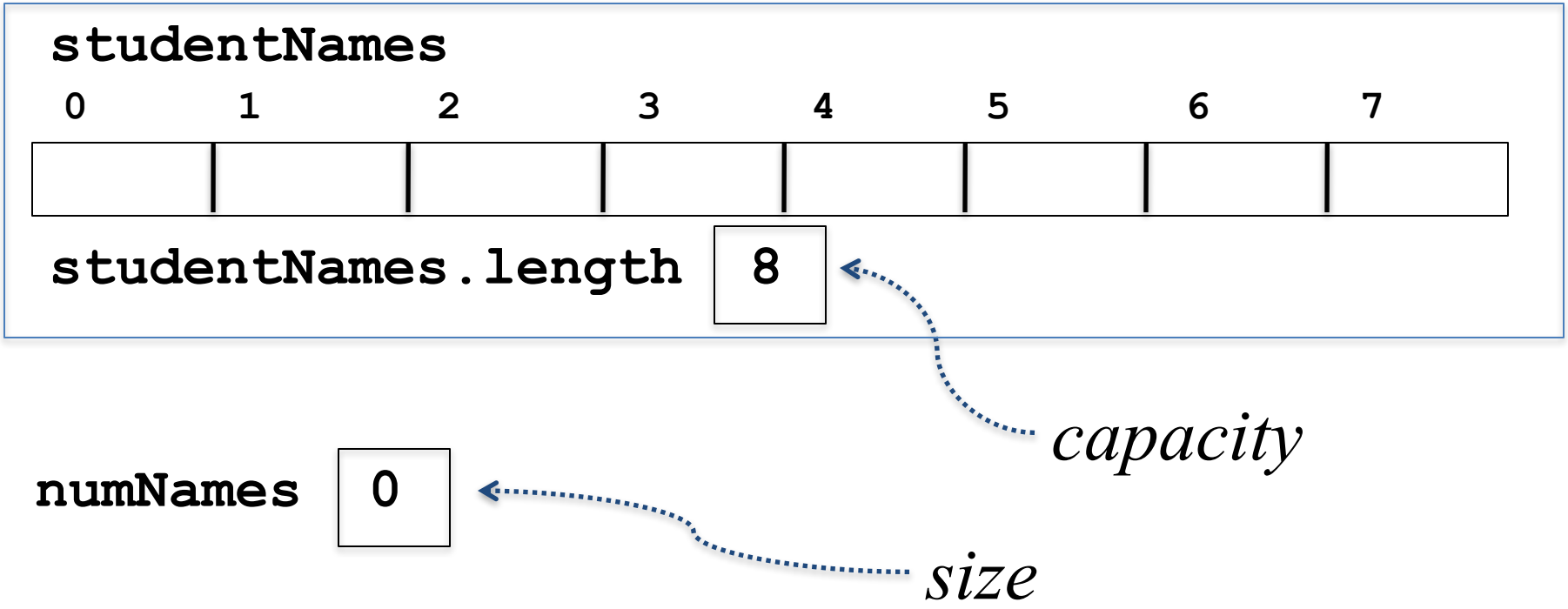
- Use a partially filled array

Ex: partially filled array of student names



code to add a new student to the end:

Empty partially filled array of student names



example initialization:

```
String[] studentNames = new String[8];
int numNames = 0;
```

Difficulties of partially filled array

- have to guess necessary capacity ahead of time
- have to keep two variables in sync: **`numNames`** and **`studentNames`**
- What if we run out of space?
  - have to allocate a bigger array
  - copy all the elements from smaller array to bigger array
  - **`Arrays.copyOf`** (discussed in section 7.3.9 can help with this)
- Common use of arrays, so …

`ArrayList` class

- Hides the code to take care of messy details of partially-filled array:
- Keeps track of how full array is:

  ```
  arrList.size()
  ```

- Makes array bigger as necessary:

  ```
  arrList.add("Joe");
  ```

  adds Joe to the *end* of the partially-filled array
- Accessing individual elements by index still uses random access (fast): **`get`**, **`set`**