Names class; Class design

Finish Names example

- practice with
  - coding array algorithms
  - implementing classes
  - and using good development techniques
- incremental development
- for lookup, **remove**, insert:
  - design test cases first
  - implement code
    - code refactoring
  - test code

Class Design

- Preconditions
- Class invariants
    - representation invariants
    - testing repr. invariants

Announcements

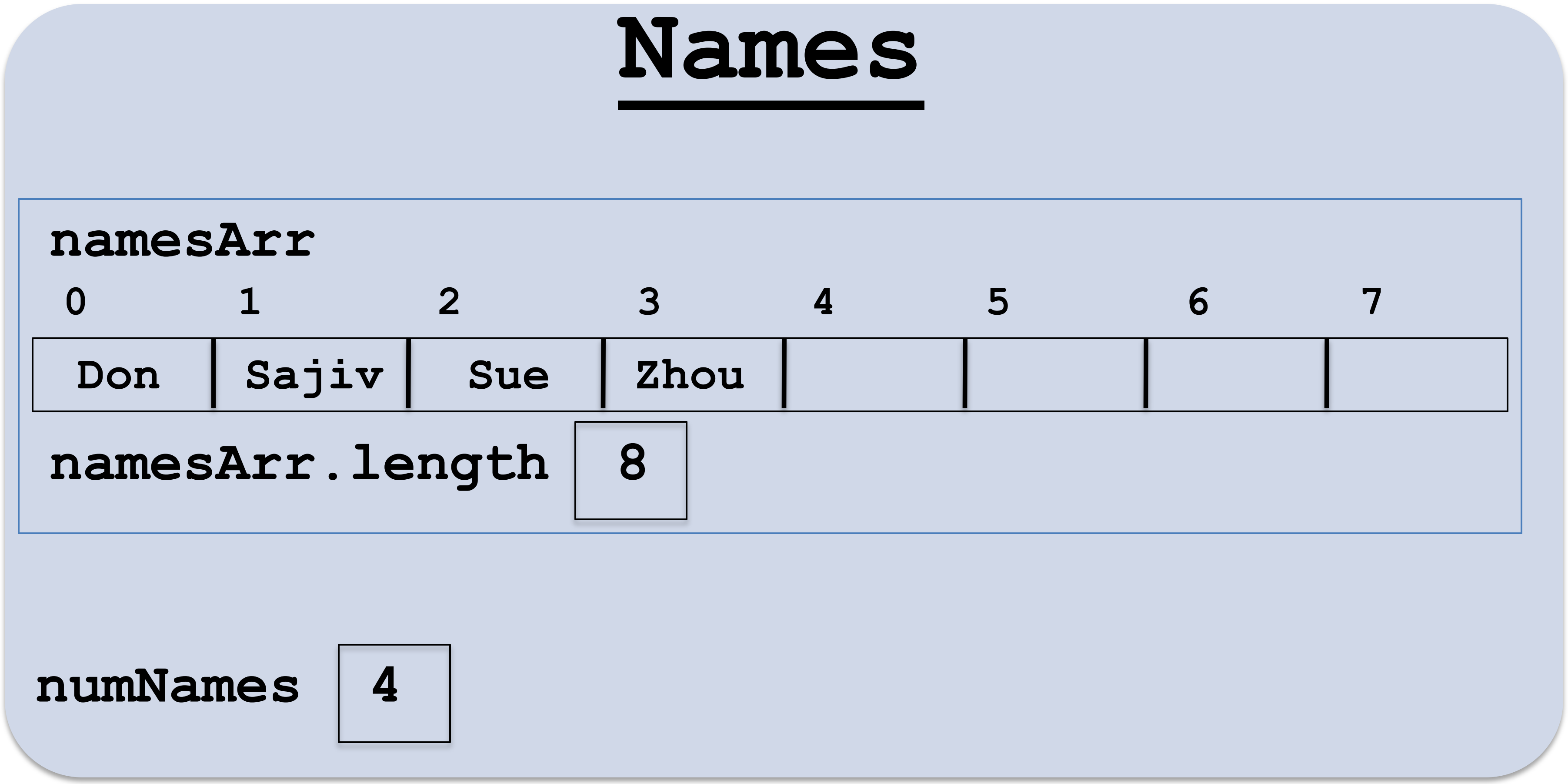
- This week's lab: milestone for PA2 (today's lecture helpful for lab)
- Midterm 1 is on Tue 9/28 9:30am – 10:50am
  - sample problems have been published
  - closed book, closed note, no electronic devices (e.g., no smartwatch)
  - bring pencils (or pens), erasers
- MT1 Remote students:
  - remote students received their detailed instructions in email
  - rehearsal exam for remote students (to check setup):
    - window to take it: 6:00pm Fri, Sept 24 PT - 6:00pm Sat, Sept 25 PT. (no zoom)
    - Spring 20 MT1 will be the rehearsal exam contents.

Example: **`Names`** class

- Stores a list of unique names in alphabetical order.
- Allows look-up, insert, and removal of names in the list.
- Uses partially-filled array representation

- **`Names.java`** has a partial implementation
- **`MinNamesTester.java`** is a program to test that subset.

Names representation



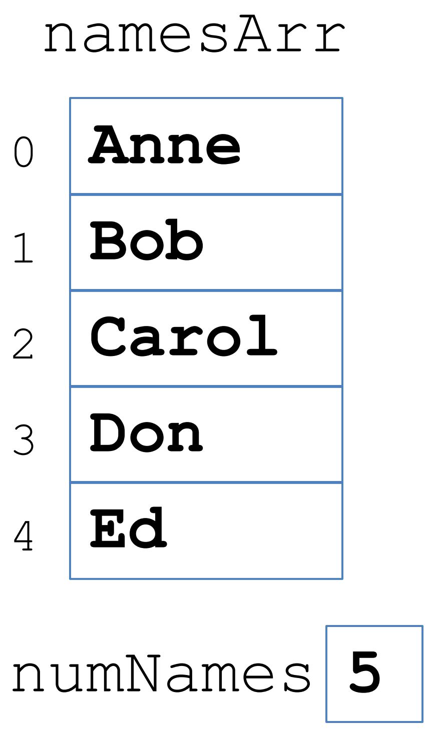
Reuse code to test remove

```
public static void testRemove() {
   Names names = new Names();
   names.loadNames();
   System.out.println("Attempt remove: Scotty");
   boolean removed = names.remove("Scotty");
   if (!removed) {
      System.out.println("Scotty was not present");
   }
   System.out.println(
     "Names in list [exp: Anne Bob Carol Don Ed]: ");
   names.printNames();
   System.out.println(
             "Number of names in list [exp: 5]: "
             + names.numNames());
}
```

Implementing remove: outline

Removes **`target`** from names object, and returns **`true`**.
If **`target`** wasn't present in names, returns **`false`** and no change made to names.

```
public boolean remove(String target) {
```



Minimize amount of code

- Reuse lookup loop?
- It returns boolean
- Refactor!

New helper function

```
/**
    lookupLoc returns index of target in namesArr
    or NOT_FOUND if it is not present
*/
private int lookupLoc(String target)
```

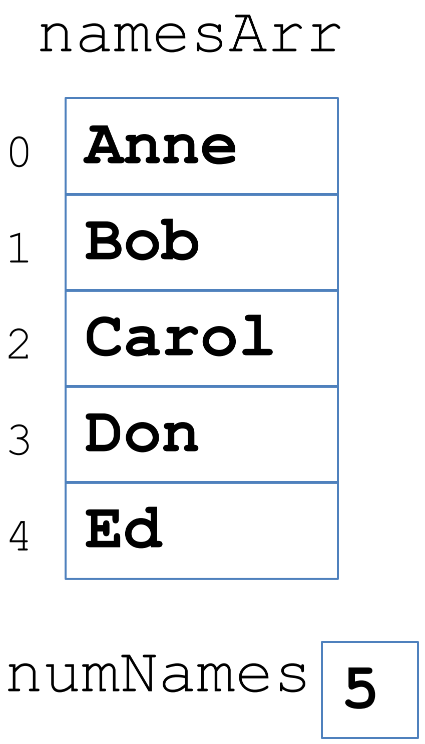
Refactored `lookup` that uses `lookupLoc`

```
public boolean lookup(String target)
```

Implementing remove

Removes **`target`** from names object, and returns **`true`**. If **`target`** wasn't present in names, returns **`false`** and no change made to names.

```
   public boolean remove(String target) {
```



Class design: Method preconditions

- a restriction on how a method can be called
  - Ex (from book): in **`BankAccount`** class

    `void deposit(double amount)`

  Precondition:

- document any preconditions in the method comment
- why not

  "amount must be type double" ?

Method contract

- client must satisfy precondition
- a contract between client code and method:
  - if you call the function this way,
    we guarantee it will do what we say it does
  - otherwise, behavior is undefined
- avoid performing duplicate checks between client and method code

POLL: preconditions

- **`BankAccount`** example

```
void deposit(double amount)
```

Precondition: **`amount > 0`**

Asynchronous participation: [Link to Preconditions poll]()

What should method do?

- a call that violates the precond is incorrect (remember: undefined results)
- Java **`assert`** statement is useful:

```
assert amount > 0;
```

Restrictions on implicit parameter

```
x.foo();
```

Another reason for a precond:

- restriction on *when* certain methods can be called
  - object can be in different states
- Illegal to call **`next()`** when **`Scanner`** has no more input (eof in lab4)
- **`PRE: hasNext() is true`**
- Try to minimize them

Your Precondition comments

- Two ways to document at the top of a method:
- Javadoc style (next to param in question):

```
@param amount
        the amount of money to deposit,
        must be > 0
```

- Or state all preconditions on separate line:

```
PRE: amount > 0
```

Class Invariants

- a statement about an object that's always true between method calls:
  - true after constructor
  - true after every mutator
  - (therefore, also true before every method call)
- interface invariant: true from client view
- representation invariant: true about object representation

Interface Invariants

- sometimes related to preconditions
- Example in book: **`BankAccount`**

  ```
  Invariant: getBalance() >= 0
  ```
- would document in overall class comment
- For **`CoinTossSimulator`** class:

  ```
  Invariant: getNumTrials() =
          getTwoHeads() + getTwoTails() +
          getHeadTails()
  ```
- For **`Names`** class

  ```
  Invariant: names are in alphabetical order
                  and are unique
  ```

Representation invariants

- a statement about the *internal object representation* that's always true between method calls:
  - true after constructor
  - true after every mutator
  - (therefore, also true before every method call)
- describes valid internal state of the object
  - any restrictions on what can be in instance variables
  - any relationships between values in different instance variables

Ex: Repr. invar. for **`Names`** class

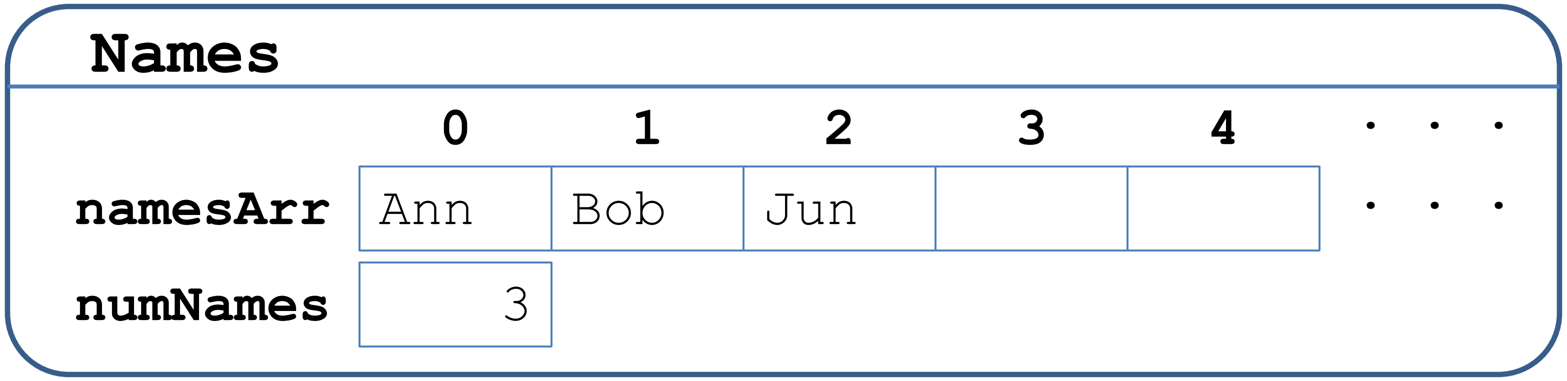
- … that uses ***ArrayList*** representation

```
class Names {
   . . .
   private ArrayList<String> namesArr;
   /* Representation invariant:
      -- names are unique
      -- names are in alphabetical order in namesArr
      -- number of names stored is namesArr.size()
   */
}
```

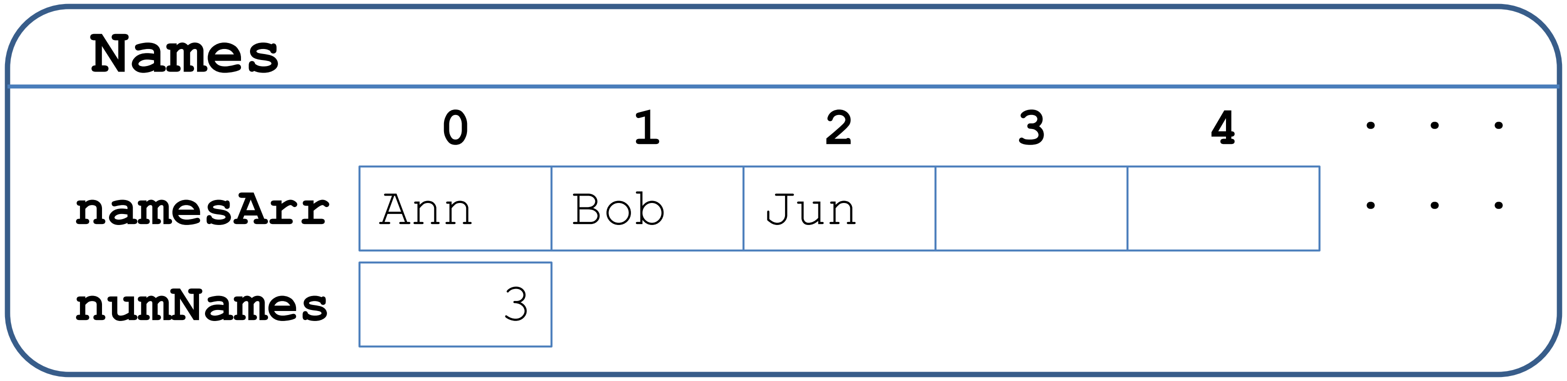
Ex 2: Repr. invariant for **`Names`** class

- … that uses ***partially filled array*** representation

```
class Names {
   . . .
   private String[] namesArr;
   private int numNames;
}
```



Ex 2 of repr. invariants (cont.)

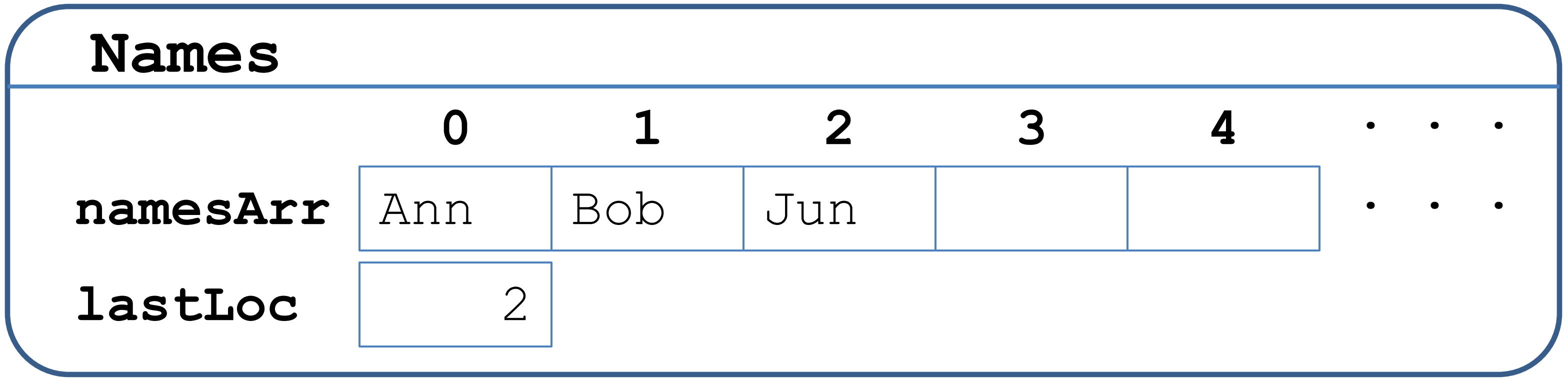


repr. invariant:

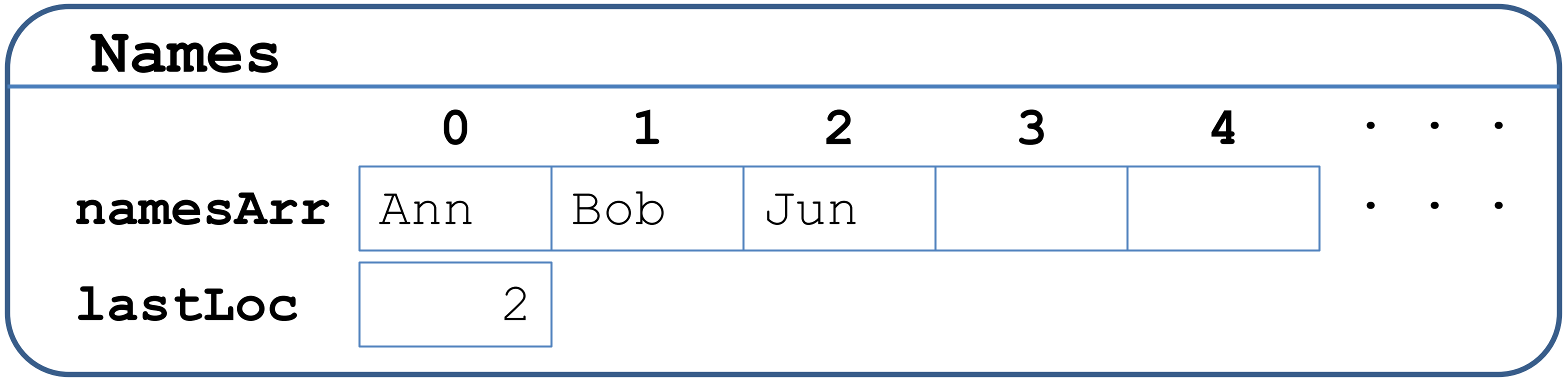
- **`numNames is the number of names`**
- **`0 <= numNames <= namesArr.length`**
- **`if numNames > 0, the names are in namesArr`**
  `locs: 0 <= loc < numNames`
- **`names are in alphabetical order`**
- **`names are unique`**

Different invar. with same data types

```
class Names {
   . . .
   private String[] namesArr;
   private int lastLoc;
}
```



Different invariant (cont.)



- representation invariant:

Testing representation invariants

- Can use **`assert`** for sanity checks.
- One kind of sanity check:

  check representation invariant
- Write a *private* method:

```
boolean isValidObject()
```

- at end of every method:

```
assert isValidObject();
```

- You will be doing this in pa2.