More on class design; parameter passing

- From last time:
  - a little more on Preconditions
  - do Representation Invariant example
- What does a class represent?
- Minimizing inter-method dependencies
- Choosing instance variables
  - minimizing scope
- Parameter passing

Announcements

- Completed code for Names.java and NamesTester.java is in **`$ASNLIB/public/09-21/complete`**
- Midterm 1 is on Tue 9/28 9:30am – 10:50am
  - room assignments in email from 9/20 and on piazza
  - closed book, closed note, no electronic devices (e.g., no smartwatch)
  - bring pencils (or pens), erasers
  - bring USC ID card
- 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. (zoom at the start of the session)
    - Spring 20 MT1 will be the rehearsal exam contents.

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

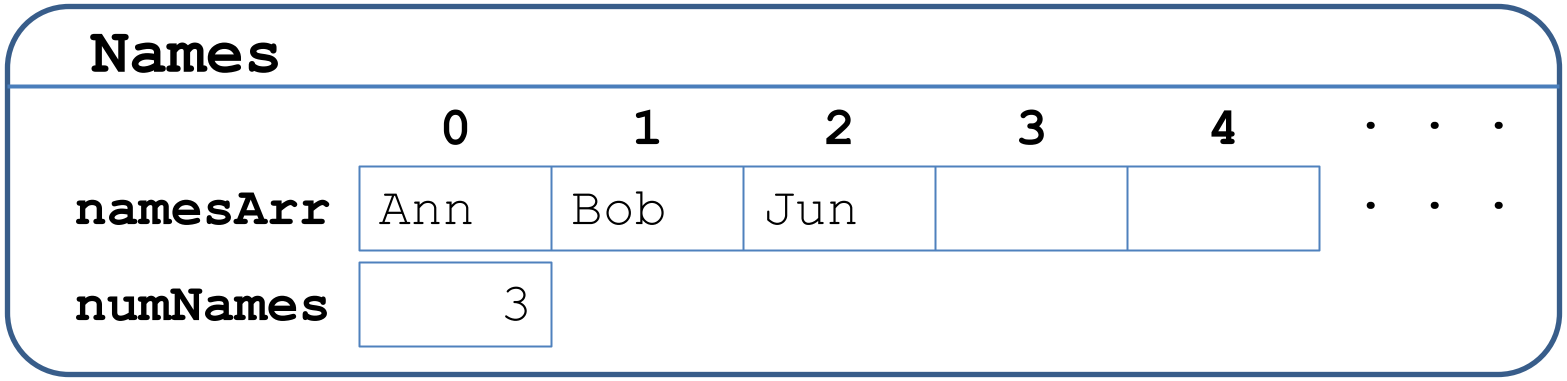
- … that uses ***ArrayList*** representation

```
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()
   */
}
```

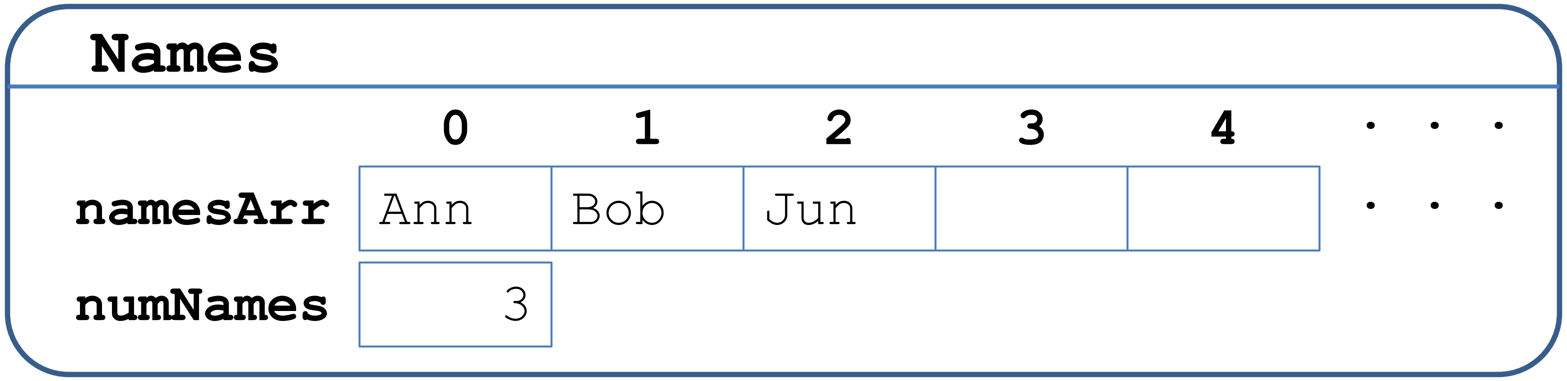
Ex 2: Repr. invariant for **`Names`** class

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

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



Ex 2 of repr. invariants (cont.)

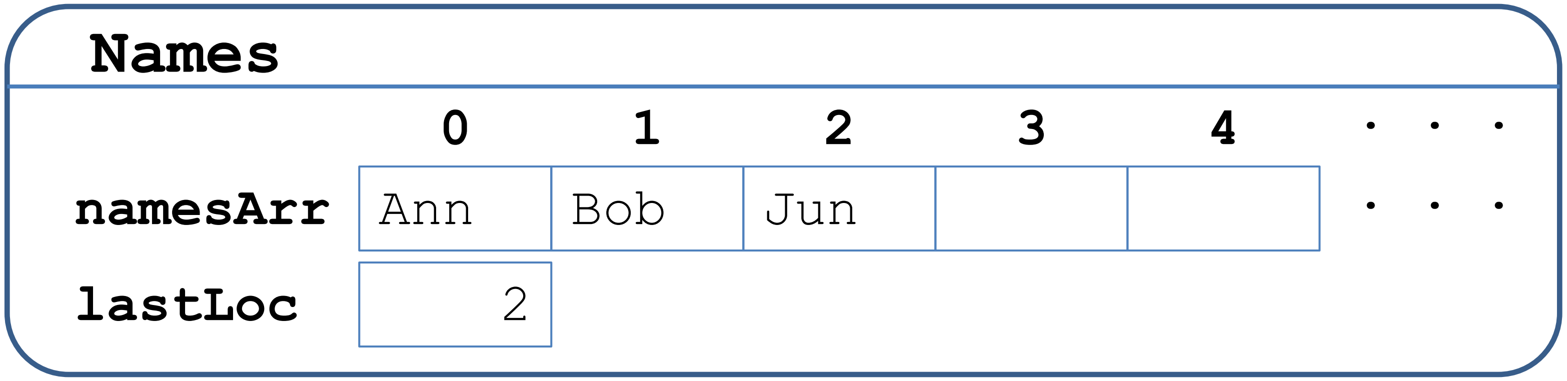


repr. invariant:

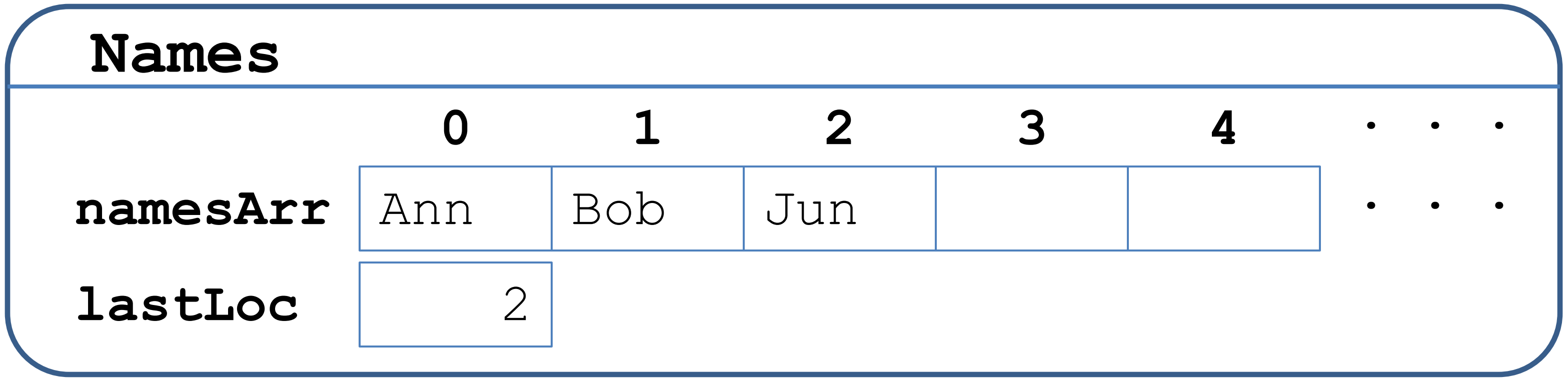
- **`numNames is the number of names`**
- **`0 <= numNames <= namesArr.length`**
- **`if numNames > 0, the names are in namesArr`**
  `locs: 0 <= loc < numNames`
- **`names are in alphabetical order`**
- **`names are unique`**

Different representation with same data types

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



Practice with representation invariant



- representation invariant:
- lastLoc +1 **is the number of names**
- **valid range of lastLoc:** [-1, nameArr.length-1]
- **if** lastloc >= 0**, the names are in namesArr locations:** [0, lastloc]
- **names are in alphabetical order**
- **names are unique**

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.

Class is a single concept

- Class should represent a single concept
- An object in the real world
  - (or from math, or a software artifact)
- E.g., Point, Rectangle, Bar, Paycheck
  - Methods all relate to that single concept:
  - get info about the object (accessor)
  - manipulate the object (mutator)
- Can make multiple instances of the class

A bad class design

```
class MyProgAssgtClass {
    public void doStep1() { . . . }
    public void doStep2() { . . . }
    public void doStep3() { . . . }
    // instance variables are effectively
    // "global" vars
}
```

- Can you make multiple instances of the object?
- What is the data abstraction it represents?

Minimizing inter-method dependencies

- Inter-method dependencies:
  - Generally want to be able to call methods in any order. e.g., Names: lookup, insert, remove
  - Minimize the different states object can be in

Some objects naturally have multiple states

- Have to think through what they are and transitions between them
- Ex: cash register class from Ch. 3 (and lab 3)

Choosing instance variables

- For implementor: Instance variables are the input to every method.
- Need a clear understanding of what values are for, and how they are interrelated

POLL: Choosing instance variables

Suppose we had the following **`CoinTossSimulator`** instance variables. Which of them can we *eliminate*?

```
private int totNumTrials; // total since last reset
int currNumTrials; // total for this run
int numHeadsTails;
int numTailsTails;
int numHeadsHeads;
int i;               // which trial we are on
Random generator;
boolean doneReset; // have we done a reset?
```

Asynchronous participation: [Link to Instance Variables poll]()

A general principle:

- "principle of locality"
- Minimize scope of variables / methods
  - public vs. private
  - instance var vs. local var
  - method scope vs. loop body scope

- Also one of our style guidelines for the class

Minimize scope: another example

- Proposed solution for reuse **`lookup`** code: Adding a data member so **`remove`** could use **`lookup`**:

```
class Names {
  private String[] namesArr;
  private int numNames;
  private int locationFound;
                      // when is this init'd?
  . . .

  public boolean lookup(…) {
     . . . locationFound = . . .
  }
  public boolean remove(…) {
     . . . lookup(…);
     i = locationFound; . . .
  }
  . . .

}
```

Second example (cont.)

- Reminder: improved solution
- private helper method

```
class Names {
  private String[] namesArr;
  private int numNames;
  ~~private int locationFound;~~
  . . .
  public boolean lookup(…) { …lookupLoc(…)… }
  public boolean remove(…) {…lookupLoc(…)… }
  private int lookupLoc(…) {  }
  . . .
}
```

Choosing instance variables (cont.)

- Scenario: use an **`ArrayList`** representation for **`Names`** class.
- Suppose we had the following **`Names`** instance variables:

```
ArrayList<String> namesArr;
int numNames;
```

- Why is this not ideal?

Review of instance variables

- For implementer: Instance variables are the input to every method.
  - want to minimize how many
  - and how many different states they can be in
- Need a clear understanding of what values are for, any restrictions on them, and how they are interrelated
- Explicit statement of the last two is the

  representation invariant

Parameter passing in Java

- All Java parameters are passed by value.
- Value and reference semantics also apply to parameter-passing rules:
  - Primitive types use value semantics
  - Object types (and arrays) use reference semantics
- Let's see what this means . . .

Parameter passing in Java: primitive types

- all parameters passed by value. E.g.,

```
public static void foo(int x) {
    x = 0;
}
```

has no effect on caller:

```
int y = 10;
foo(y); // y unchanged
System.out.println(y);   // 10
```

Parameter passing: object references

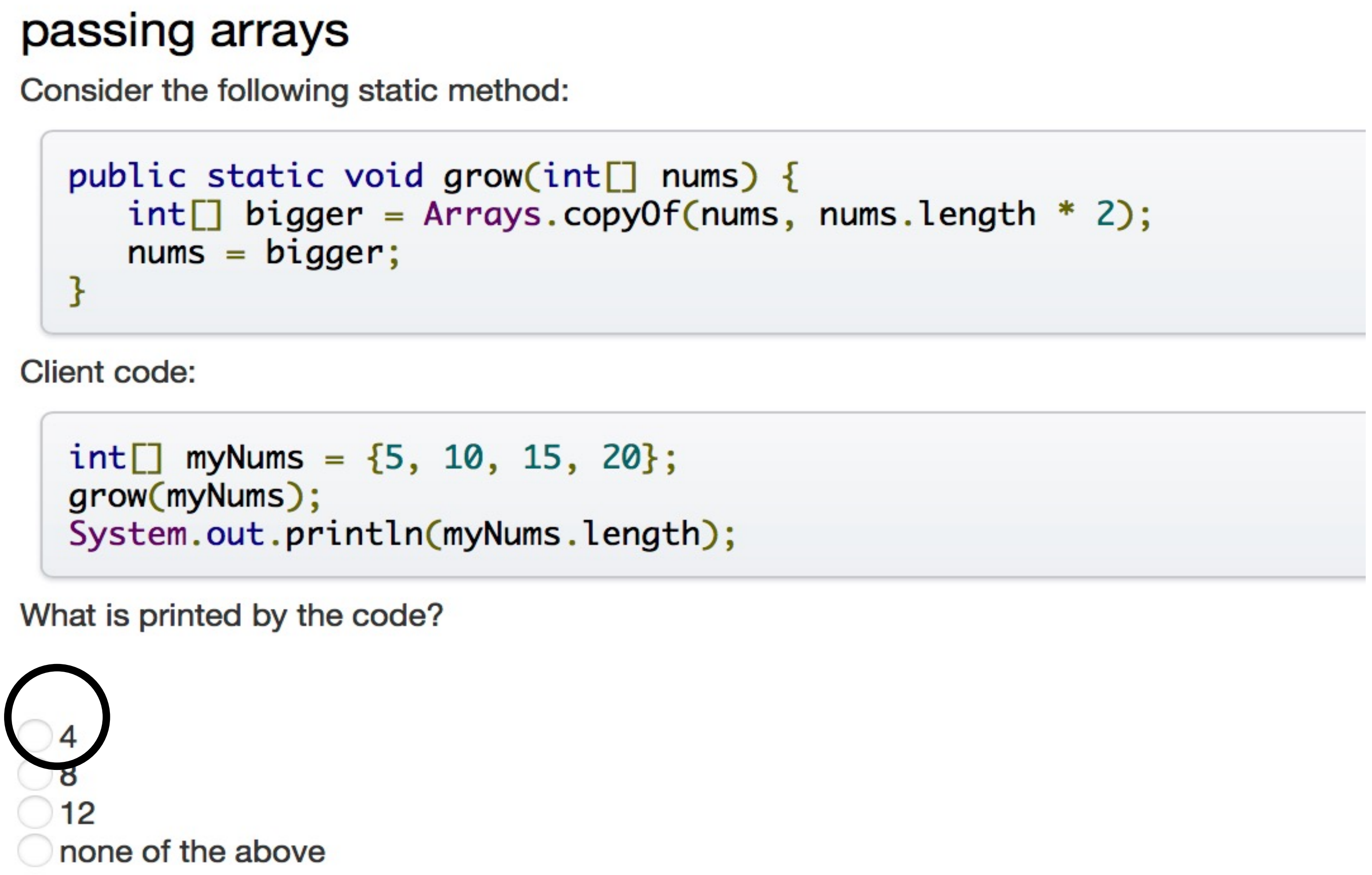
- for objects and arrays, the object *reference* is passed by value. E.g.

```
public static void foo(BankAccount account) {
   account = null;
}
```

has no effect on caller:

```
BankAccount myAccount = new BankAccount(100);
foo(myAccount);
int bal = myAccount.getBalance();  // 100
```

Poll: passing arrays



passing arrays

Consider the following static method:

```java
public static void grow(int[] nums) {
   int[] bigger = Arrays.copyOf(nums, nums.length * 2);
   nums = bigger;
}
```

Client code:

```java
int[] myNums = {5, 10, 15, 20};
grow(myNums);
System.out.println(myNums.length);
```

What is printed by the code?

- [✓ 4]
- 8
- 12
- none of the above

Asynchronous participation: Link to Passing Arrays poll

Passing object references by value

- Method can't change *which* object **`myAccount`** refers to
- But it could still change what's *inside* the object
  by calling one of its mutators:

```
public static void evil(BankAccount account) {
   account.withdraw(account.getBalance());
}
```

- Call:

```
BankAccount myAccount = new BankAccount(100);
evil(myAccount);
int bal = myAccount.getBalance();
```

How to “change” a primitive var in a method

Can *use return value* to update a single variable:

```
public static int incr(int x) {
  return x+1;
}
```

Sample call:

```
int x = 5;
x = incr(x);
```

Similar idea with immutable object:

```
String s = “foobar”;
s = s.substring(3);
```

Example: *cannot* write a swap method in Java

Method definition:

```
public static void swap(int x, int y) {
  int temp = x;
  x = y;
  y = temp;
}
```

Sample call:

```
int a = 5;
int b = 10;
swap(a, b);
```