CSE 21
Intro to Computing II

Lecture 12 – Final review

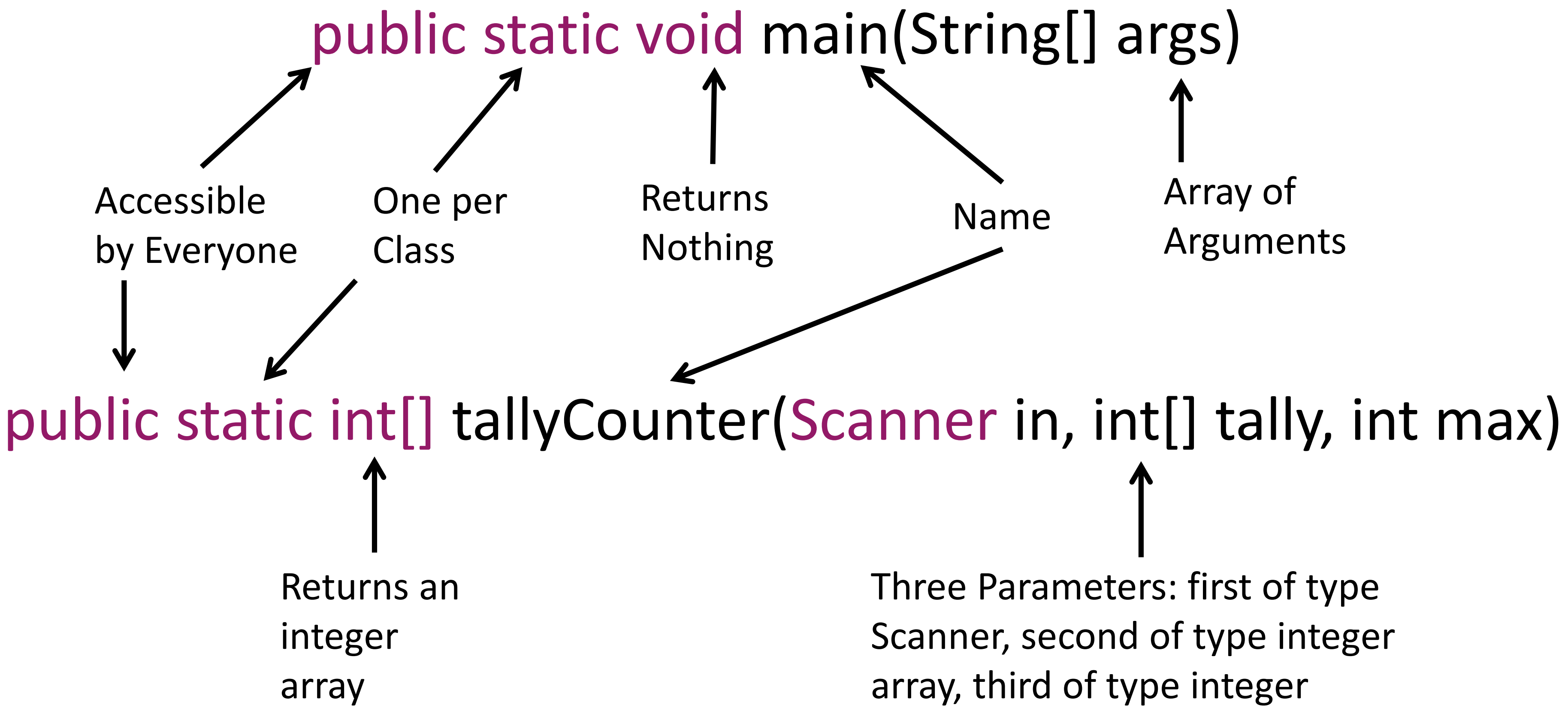
Topics

- Methods
  - Return values
  - Input arguments
  - Overloading a method
- Object Oriented Programming
  - Objects/classes
  - Instance vs class variables
  - Common methods in a class
  - Access control of variables/methods
  - Objects and pointers
  - Array of Objects

Topics

- Inheritance
  - Class hierarchy
  - Access control of members
  - What is/are inherited?
- ArrayList
  - Basic operations
    - Declaration, add, remove, insert, access
  - Used with primitives
    - Wrapper classes: Integer, Double, Character, Boolean
- File I/O
  - Scanner operations, delimiters
- Recursion
- Multi-dimensional arrays

Methods



```
public static void main(String[] args)
```

Accessible by Everyone

One per Class

Returns Nothing

Name

Array of Arguments

```
public static int[] tallyCounter(Scanner in, int[] tally, int max)
```

Returns an integer array

Three Parameters: first of type Scanner, second of type integer array, third of type integer

Sum Example

tally[0] = 13
tally[1] = 18

```
public class PreferenceMOSv2{

    // Method Declaration like variables (callee)
    public static int CombinedTally(int num1, int num2) {              #3
        System.out.println("First tally is " + num1);                  #4
        System.out.println("Second tally is " + num2);                 #5
        int total = num1 + num2;                                       #6
        return total;                                                  #7
     }

    public static void main(String[] args) {                           #1
        ...
        int sum #8 = CombinedTally(tally[0], tally[1]);   // caller    #2
        System.out.println("Total tally is " + sum);                   #9
    }
}
```

Local variables
for **total** only

Output:

```
First tally is 13
Second tally is 18
Total tally is 31
```

Sum Usage

- Want to add 3 numbers (tally[0], tally[1], tally[2])

- First Option
  - int total1 = CombinedTally(tally[1], tally[2]);
  - int total = CombinedTally(tally[0], total1);
- Second Option (Substitution)
  - int total = CombinedTally(tally[0], CombinedTally(tally[1], tally[2]));
- Third Option (Commutative +)
  - int total = CombinedTally(CombinedTally(tally[1], tally[2]), tally[0]);

How to calculate a discount?

- $10 discount if total purchase is $50 or over and an **additional** $15 discount ($25 total) if total purchase is $100 or over:
    - if >= $50 then -$10 AND if >= $100 then extra -$15
    - if >= $100 then -$25 OR if >= $50 then -$10
    - if >= $50 then -$10 OR if >= $100 then -$25

Break it down into simple logical steps!

Return styles

- If >= $50 then -$10 AND if >= $100 then extra -$15

```
discount = 0;
if (subTotal >= 50)
    discount -= 10;
if (subTotal >= 100)
    discount  -= 15;
return discount;
```

- If >= $100 then -$25 OR if >= $50 then -$10

```
if (subTotal >= 100)
    return  -25;
else if (subTotal >= 50)
    return  -10;
return 0;
```

- If >= $50 then -$10 OR if >= $100 then -$25

```
discount = 0;
if (subTotal >= 50)
    if (subTotal >= 100)
        return discount  = -25
    else
        return discount = -10;
return 0;
```

Method overloading

Are we allowed to have multiple methods of the same name???

```
public static int getAmount(Scanner input, String name) { // 1
        System.out.print("Enter the amount of " + name + ": ");
        int amount = input.nextInt();
        return amount;
}
```

2 input parameters: Scanner + String

```
public static void getAmount(Scanner input, String[] names, int[] amounts) { // 2
        for (int i = 0; i < names.length; i++) {
                System.out.print("Enter the amount of " + names[i] + " : ");
                amounts[i] = input.nextInt();
        }
}
```

3 input parameters: Scanner + String pointer + int pointer

```
public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        int sharp = getAmount(input, "Sharp");
        int brie = getAmount(input, "Brie");
        int swiss = getAmount(input, "Swiss");
        getAmount(input, names, amounts);
```

2 arguments: Scanner + String

3 arguments: Scanner + String[] + int[]

Type of arguments determines the method call!

Array parameter in Methods

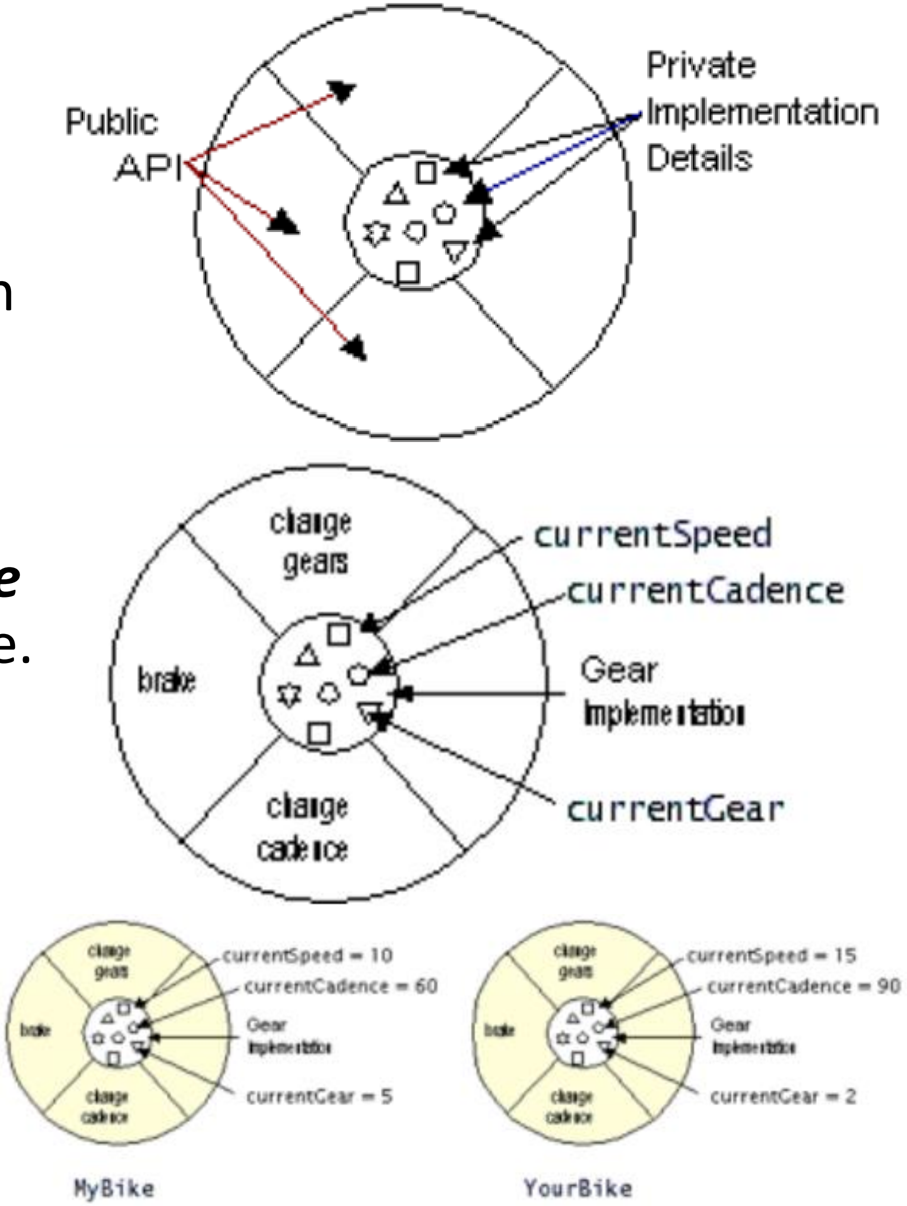
```
public static void sumAll(int[] subTotal, int max) {
        for (int i = 1; i <= max; i++)
                if(subTotal[i] == 0)
                        subTotal[i] = subTotal[i-1] + i;
}

public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        int[] sumAllArr = new int[1000];
        int repeat = 0;
        do {
                System.out.print("Enter the max number for sumAll: ");
                int max = input.nextInt();
                sumAll(sumAllArr, max);
                for (int i = 0; i <= max; i++)
                        System.out.println("Sumall of " + i + " is " + sumAllArr[i]);
                System.out.print("Repeat this program? (1 for yes)");
                repeat = input.nextInt();
        } while (repeat == 1);
}
```

OOP Concepts

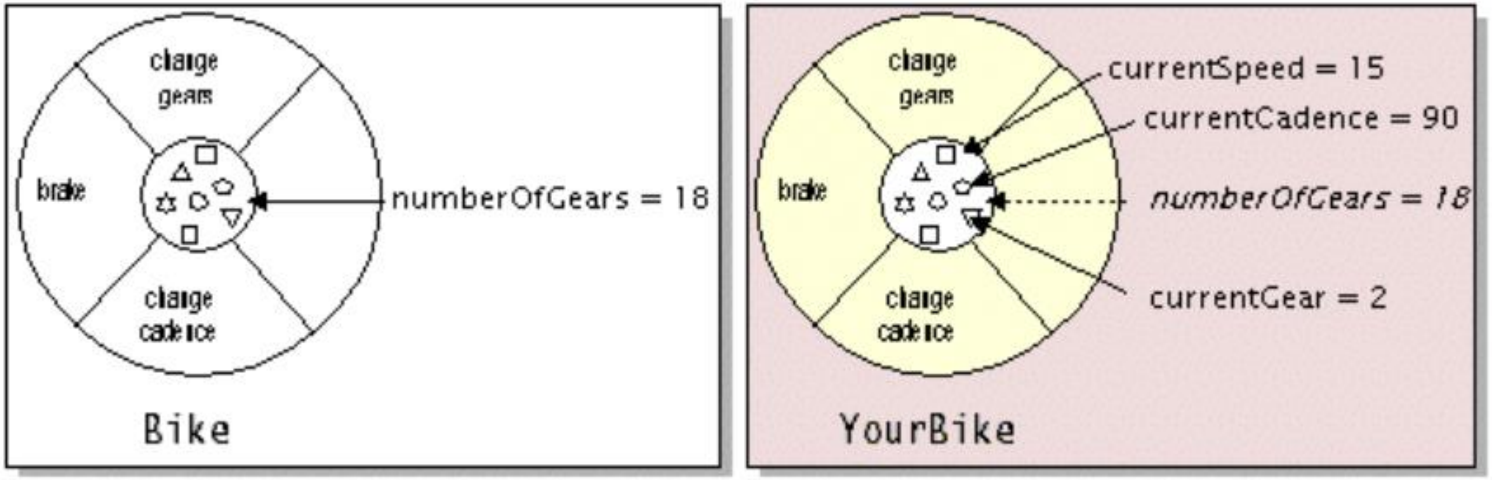
- Objects: consists of some internal data and operations that manipulate that data
  - It can help to think of an object as a "thing"
- (Member) Variables or Fields: names for the data in objects
  - A named place to store some information (state) pertaining to the object, that may or may not change
  - Variables can be instance or class (static)
- (Member) Methods: a procedure for the object
  - Something that the object can do
  - It is best if only methods are public and not variables – that is, other objects don't access variables directly
    - More flexibility (when inheriting, error checking)
    - Equally efficient (in most cases)
- Classes: factories for "generating" objects
- Package: a set of related classes
  - This is how you find existing code
- Project: a set of packages/classes that solve a problem (also a set of files on your computer)

Classes/Objects

- A class is the **"blueprint"** or **"factory"** that defines the variables and methods common to all objects of a certain kind.
- Objects are instances of a class type.
- Methods isolate, or ***encapsulate*** the data inside from the outside.
  - Other objects ask about this object's state via methods.
- After you have your Bike class, you can create any number of bike objects!



Instance vs. Class (static) variables



Class

Instance of a Class

- A ***class variable*** (aka ***static variable***) is shared by all instances of the same class.
  - Unlike ***instance variables*** that can be different for each instance.
  - E.g., suppose all bikes had the same number of gears. If we made this a class variable, and we wanted to change it, it would change for ALL bikes.

```
static int numGears;
```

Common Methods in a Class

- Methods common to many classes
  - *Constructors* are called if you ask for a ***new*** object
    - Java provides a ***default*** constructor (with no arguments)
  - *Accessors*, or "get methods", or "getters" are used to read/retrieve the values of instance variables
    - Including predicate methods returning booleans
  - *Mutators*, or "set methods", or "setters" are used to set the values of instance variables
  - **toString** method creates a String representation of the contents of the object
    - **System.out.println(obj)** calls object's **toString**
    - public String toString() { … }

Date Class Definition

```
public class Date {
        public int day;
        public int month;
        public int year;
        public Date() {                                     // Constructor 1
                day = month = year = 0;
        }
        public Date(int year) {                             // Constructor 2
                day = month = 0;
                this.year = year;
        }
        public Date(int year, int month) {                  // Constructor 3
                day = 0;
                this.month = month;
                this.year = year;
        }
        public Date(int year, int month, int day) {         // Constructor 4
                this.day = day;
                this.month = month;
                this.year = year;
        }
}
```

We use “**this**” to explicitly access instance variables.

The "this" implicit parameter

- Compiler converts

  objectReference.method(…);

  To

  method(objectReference, …);
- Implicitly-passed object reference is accessible via **this**
- Useful when method parameter and member variable have the same name

```
public class Date {
        public int day;
        public int month;
        public int year;
        public Date(int year) {
                day = month = 0;
                this.year = year;
        }
}
```

Accessors and Mutators

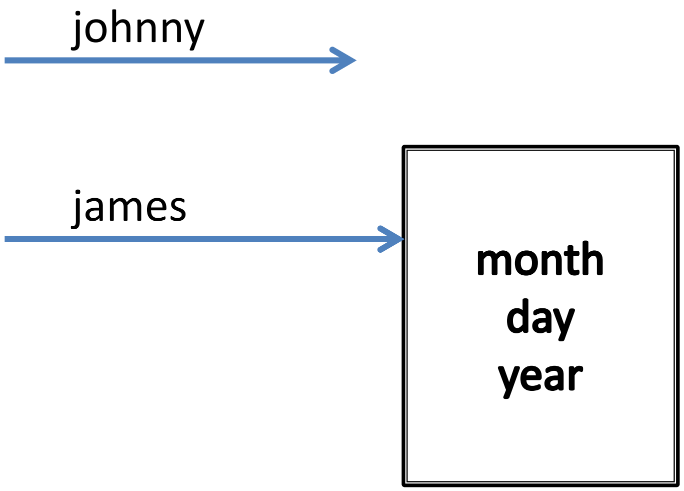
```
public class Date {
    private int month;
    private int day;
    private int year;

    public void setMonth(int month) {
        if (month > 0 && month <= 12)
                this.month = month;
        else
                System.out.println("Invalid month");
    }
    public int getMonth() {
        return month;
    }

}
```

Compile-time error

```
Date johnny = new Date( );

// instead of johnny.month = 7;
johnny.setMonth(7);  // method call

// month is a variable
System.out.println("Birth month " + johnny.month);
```

Objects and Pointers

- Date johnny;
- Date james = new Date();
- johnny = james; ???



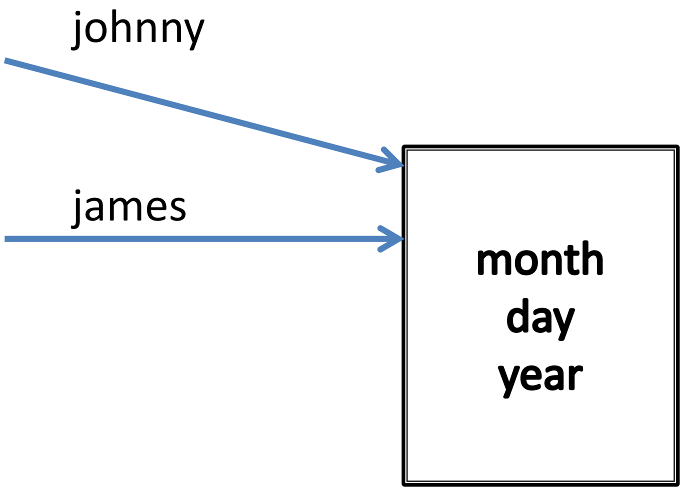
johnny

james

month
day
year

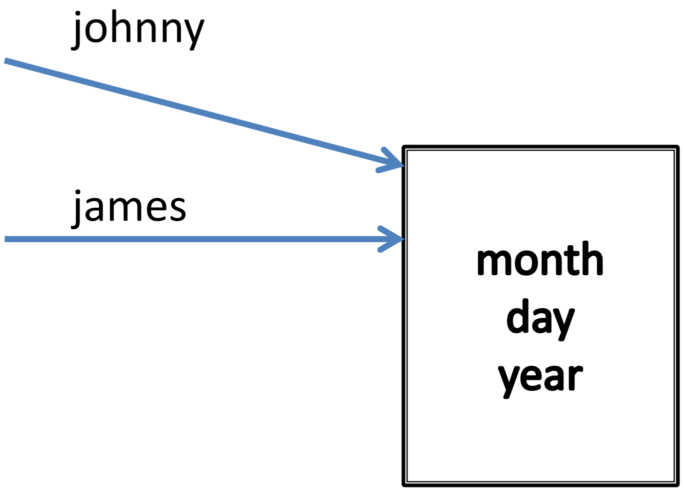
Objects and Pointers

- Date johnny;
- Date james = new Date();
- johnny = james;



Objects and Pointers

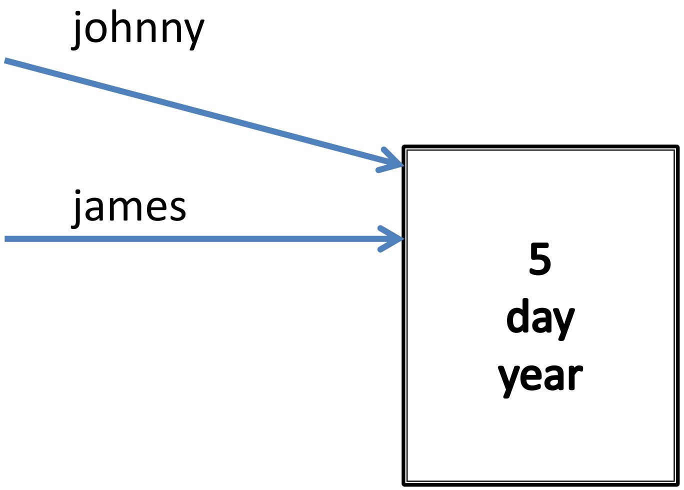
- Date johnny;
- Date james = new Date();
- johnny = james;
- james.setMonth(5); ???



Objects and Pointers

- Date johnny;
- Date james = new Date();
- johnny = james;
- james.setMonth(5);
- johnny.getMonth(); ???

5



johnny

james

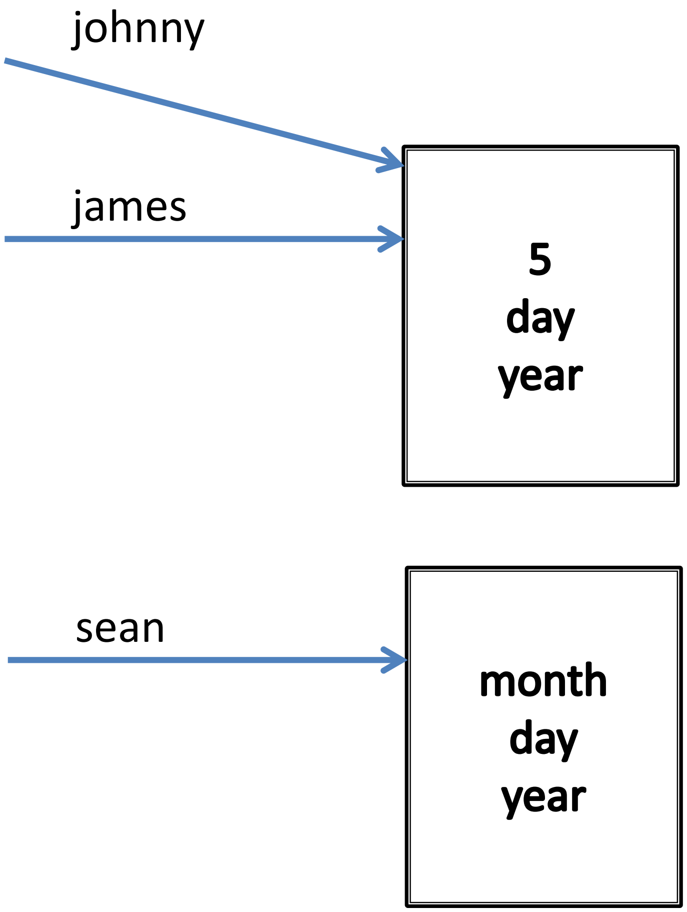
5

day

year

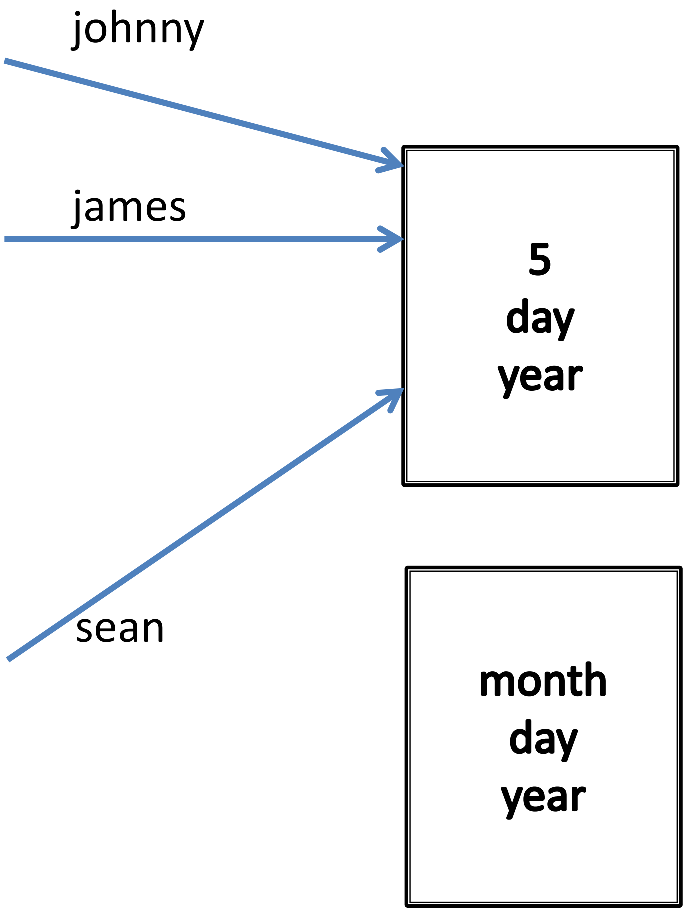
Objects and Pointers

- Date johnny;
- Date james = new Date();
- johnny = james;
- james.setMonth(5);
- Date sean = new Date();
- sean = james; ???



Objects and Pointers

- Date johnny;
- Date james = new Date();
- johnny = james;
- james.setMonth(5);
- Date sean = new Date();
- sean = james;

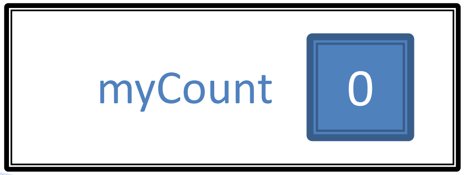


Array of Objects

- Date johnny = new Date();
  - Creates an object pointed to by variable johnny
- Date[] birthdays = new Date[MAX];
  - Creates MAX # of Date pointers
  - Does not have objects yet
  - Not valid to use birthdays[0].setMonth(12) yet
  - Statement creates MAX # of entries
- birthdays[0] = new Date();
  - Now we can access
  - birthdays[0].setMonth(12);
- Need to instantiate two things for arrays (new)
  - Pointers using Square brackets
  - Objects using parenthesis

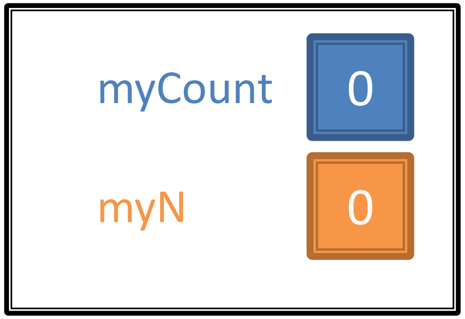
Counter Class Example

```
public class Counter {
        private int myCount;
        public Counter() {
                myCount = 0;
        }
        public void increment(){
                myCount++;
        }
        public void reset() {
                myCount = 0;
        }
        public int value() {
                return myCount;
        }
}
```



myCount 0

```
public class ModNCounter extends Counter {

        private int myN;
        public ModNCounter(int n){
                myN = n;
        }
        public int value(){
                // Cycles from 0 to (myN – 1)
                return myCount % myN;
        }
        public int max(){
                return myN-1;
        }
}
```



myCount 0

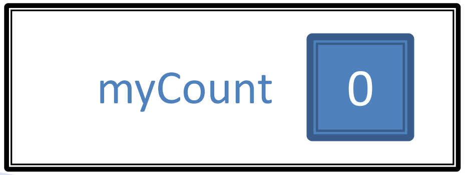
myN 0

Protected Access Specifier

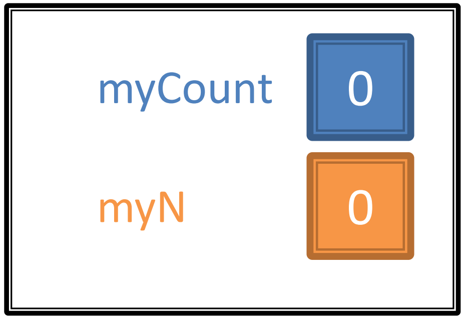
- As written, *ModNCounter* will not compile!
- The *myCount* variable is private (only accessible in the *Counter* class)
- We can fix this by making it **protected**:
  - Only classes that "extend" *Counter* can access its protected variables/methods
- Three different Access types:
  - **public**: any class can read/modify
  - **protected**: only this class, classes within the same package, and subclass descendants can read/modify
  - **private**: only this class can read/modify
  - **No modifier**: Only this class and classes within same package can read/modify. No access by subclasses.

Counter Class Example

```
public class Counter {
    protected int myCount;
    public Counter() {
        myCount = 0;
    }
    public void increment(){
        myCount++;
    }
    public void reset() {
        myCount = 0;
    }
    public int value() {
        return myCount;
    }
}
```



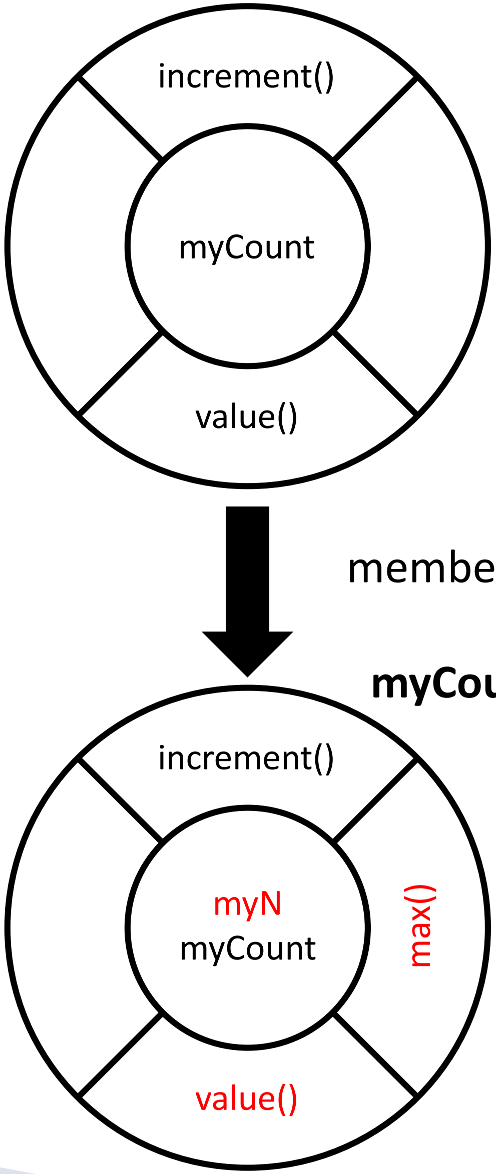
```
public class ModNCounter extends Counter {

    private int myN;
    public ModNCounter(int n){
        myN = n;
    }
    public int value(){
        // Cycles from 0 to (myN – 1)
        return myCount % myN;
    }
    public int max(){
        return myN-1;
    }
}
```



Inheritance

Superclass

class Counter



Subclass inherits members from superclass (public or protected)

myCount, increment(), value()

Subclass

class ModNCounter

Testing Equality of Objects

- To check whether two *Counters* are equal:

```
public boolean equals (Object c) {
        return this.myCount == ((Counter) c).myCount;
} //Checks if myCounts are the same.
```

Down cast to Counter type

- Overriding equals for *ModNCounter*:

```
public boolean equals (Object o) {
        ModNCounter mc = (ModNCounter) o;
        return (this.myCount == mc.myCount && this.myN == mc.myN);
} //Checks if myCounts and myN are the same.
```

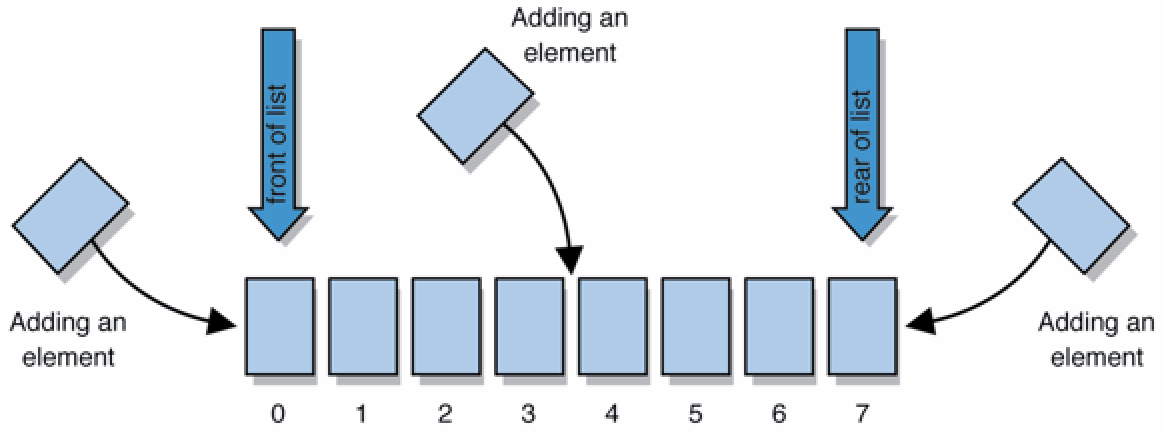
A new pointer pointing at the same (typecasted) object

ArrayList: Problems with Arrays

- The size is pre-defined
  - It cannot be changed once declared.
  - We can initialize it with a large size: *int[1000]*, but memory will be wasted if not all spaces are used.
- Difficult to insert or delete elements
  - Elements need to be shifted around when new elements are inserted or existing elements are deleted.

List of Objects

- An ordered sequence of elements:
  - each element is accessible by a 0-based **index**
  - a list has a **size** (number of elements that have been added)
  - elements can be added to the front, back, or elsewhere
  - in Java, a list can be represented as an **ArrayList** object



Contents of a List

- Rather than creating an array of boxes, create an object that represents a "list" of items. (initially an empty list.)

  { }

- You can add items to the list.
  - The default behavior is to add to the end of the list.

    {first, second, third, forth}

- The list object keeps track of the element values that have been added to it, their order, indexes, and its total size.
  - Think of an "array list" as an automatically resizing array object.
  - Internally, the list is implemented using an array and a size field.

ArrayList Methods (1)

add(**value**)	appends value at end of list
add(**index**, **value**)	inserts given value just before the given index, shifting subsequent values to the right
clear()	removes all elements of the list
indexOf(**value**)	returns first index where given value is found in list (-1 if not found)
get(**index**)	returns the value at given index
remove(**index**)	removes/returns value at given index, shifting subsequent values to the left
set(**index**, **value**)	replaces value at given index with given value
size()	returns the number of elements in list
toString()	returns a string representation of the list such as "[3, 42, -7, 15]"

ArrayList Methods (2)

addAll(**list**)
addAll(**index**, **list**)
	adds all elements from the given list to this list
	(at the end of the list, or inserts them at the given index)

contains(**value**)	returns true if given value is found somewhere in this list
containsAll(**list**)	returns true if this list contains every element from given list
equals(**list**)	returns true if given other list contains the same elements
lastIndexOf(**value**)	returns last index if value is found in list (-1 if not found)
remove(**value**)	finds and removes the given value from this list
removeAll(**list**)	removes any elements found in the given list from this list
retainAll(**list**)	removes any elements not found in given list from this list
subList(**from**, **to**)	returns the sub-portion of the list between indexes **from** (inclusive) and **to** (exclusive)
toArray()	returns the elements in this list as an array

ArrayList vs. Array

- Construction

```
String[] names = new String[5];
ArrayList<String> list = new ArrayList<String>();
```

- Storing a value

```
names[0] = "Daniel";
list.add("Daniel");
```

- Retrieving a value

```
String s = names[0];
String s = list.get(0);
```

Using index values to access contents

Scanners

- Read from User:
  - Scanner kdb = new Scanner (System.in);
  - Pass System.in as parameter to Scanner constructor
- String **s1** = "This is an example";
- Scanner line = new Scanner (**s1**);
  - Can pass in a String to Scanner constructor as well
- kdb.next();   // get next input word
- line.next();   // also gets next input word
- line.hasNext() ; // check if there is another word

Parsing Strings

```
String s1 = "This is an example";
Scanner line = new Scanner (s1);
while (line.hasNext()) {
     System.out.println(line.next());
}
```

- Delimiting character is space: ' '
- OUTPUT:

```
This
is
an
example
```

Parsing Strings with a Delimiter

```
String s1 = "This,is,an,example";
Scanner line = new Scanner (s1);
line.useDelimiter("[,]");
while (line.hasNext()) {
     System.out.println(line.next());
}
```

- Delimiting character is comma: ','
- OUTPUT:

```
This
is
an
example
```

Parsing Strings with Multiple Delimiters

```
String s1 = "+This,is+an,example";
Scanner line = new Scanner (s1);
line.useDelimiter("[,+]");
while (line.hasNext()) {
    System.out.println(line.next());
}
```

- Delimiting characters are comma and plus: ',' and '+'
- OUTPUT:

```
This
is
an
example
```

Reading File line by line

```
System.out.print("Enter the file name: ");
Scanner kdb = new Scanner(System.in);
String filename = kdb.next();

try { // TRY it out
    Scanner input = new Scanner (new FileReader(filename));
    while (input.hasNextLine()) {
        Scanner line = new Scanner(input.nextLine());
        line.useDelimiter("[\t\r]"); // Tab delimited file
        while (line.hasNext())
            System.out.print(line.next()); // Read each token
        System.out.println();  // Done reading one line
    }
    input.close();
} catch (FileNotFoundException e){ // ERROR : Catch
    System.out.println(e);
} catch (NoSuchElementException e) { // ERROR : Catch
    System.out.println(e);
}
```

2 scanner objects!

1 for reading the whole file, 1 for reading each line.

Different Scanner Methods

```
while (input.hasNextLine()) {
        Scanner line = new Scanner(input.nextLine());
        line.useDelimiter("[\t\r]");

        short s = line.nextShort();

        int i = line.nextInt();

        double d = line.nextDouble();

        float f = line.nextFloat();

        String str = line.next();
        char c = line.next().charAt(0);

        String rest = line.nextLine();
}
```

Example File Out

```
String filename = "Result.txt";

try {
        FileWriter output = new FileWriter(filename);
        String outstr = "";
        for (int i = 0; i < arr.length; i++) {
                outstr = (arr[i] + "\t");
                output.write(outputstr);
        }
        output.close();
} catch (Exception e) {
        System.out.println(e);
}
```

Two Versions of Number Summation

- Iterative (loop)

```
subTotal = 0;
for (int i = 1; i <= max ; i++) {
    subTotal += i;
}
```

- Recursive

```
public static int sumAll(int n) {
    if (n == 0)
        return 0;
    else
        return n + sumAll(n - 1);
}
```

Call the method again with a new argument

Declaration and Invocation

```
public static long sumAll(int n) { // Declaration
    System.out.println("sumAll " + n);
    if (n == 0)
        return 0;
    else
        return n + sumAll(n - 1);
}

public static void main(String[] args) {
    System.out.println("sumAll output for 5 is " + sumAll(5)); // Invoke
    System.out.println("sumAll output for 10 is " + sumAll(10));
    System.out.println("sumAll output for 20 is " + sumAll(20));
    System.out.println("sumAll output for 15 is " + sumAll(15));
    System.out.println();
}
```

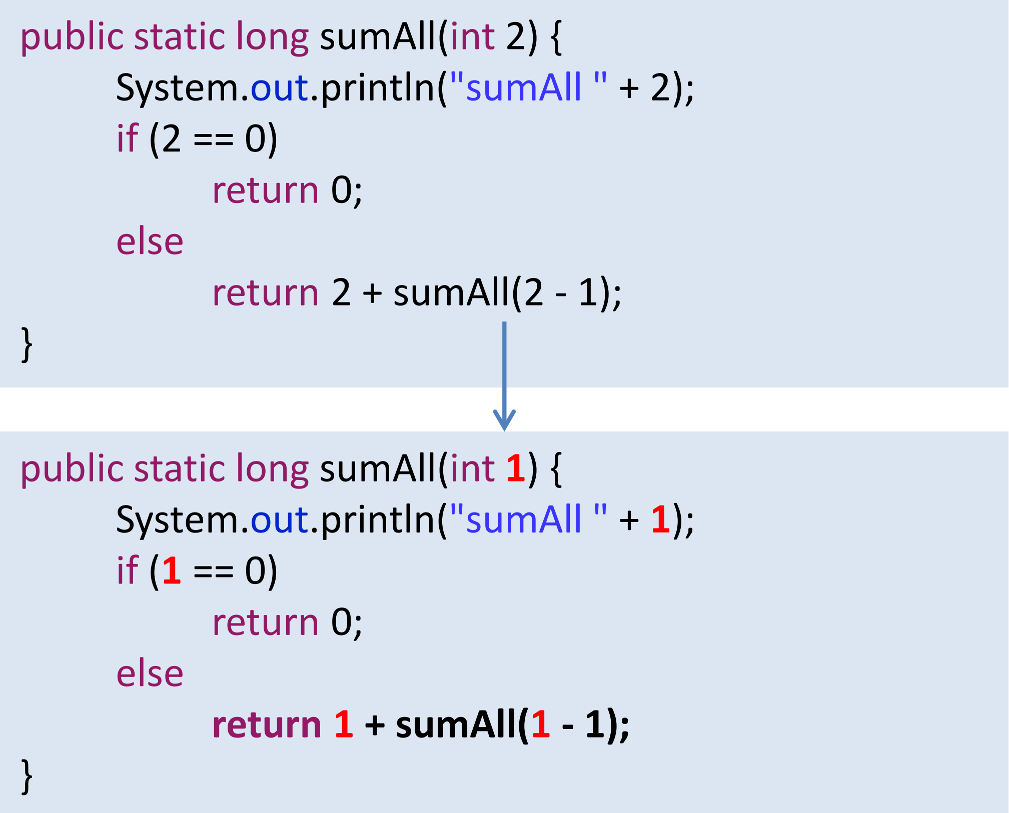
Call sumAll(2)

```
public static long sumAll(int 2) {
        System.out.println("sumAll " + 2);
        if (2 == 0)
                return 0;
        else
                return 2 + sumAll(2 - 1);
}
```

OUTPUT:

```
sumAll 2
```

Call sumAll(2)



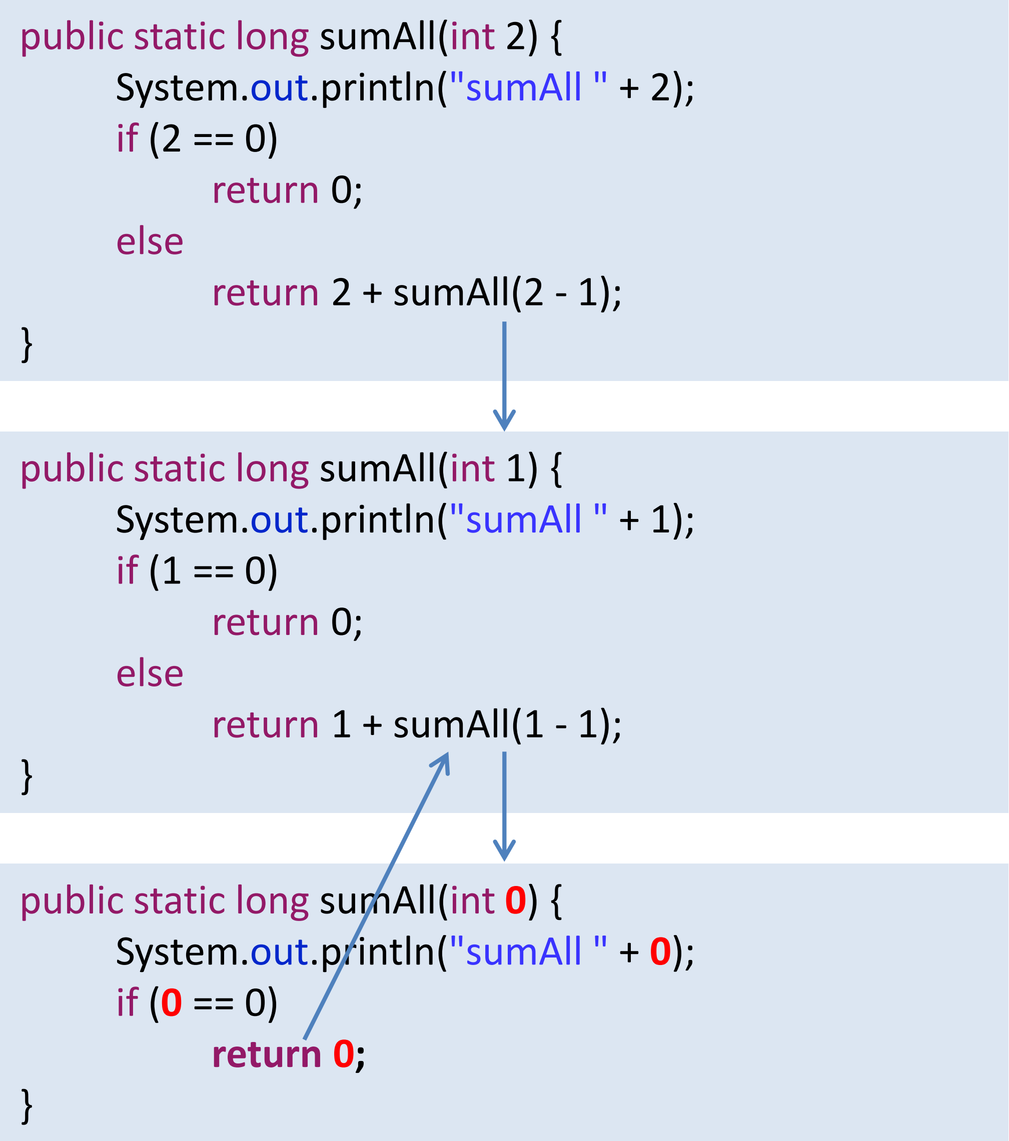
```
public static long sumAll(int 2) {
        System.out.println("sumAll " + 2);
        if (2 == 0)
                return 0;
        else
                return 2 + sumAll(2 - 1);
}
```

```
public static long sumAll(int 1) {
        System.out.println("sumAll " + 1);
        if (1 == 0)
                return 0;
        else
                return 1 + sumAll(1 - 1);
}
```

OUTPUT:

```
sumAll 2
sumAll 1
```

Call sumAll(2)



```
public static long sumAll(int 2) {
        System.out.println("sumAll " + 2);
        if (2 == 0)
                return 0;
        else
                return 2 + sumAll(2 - 1);
}
```

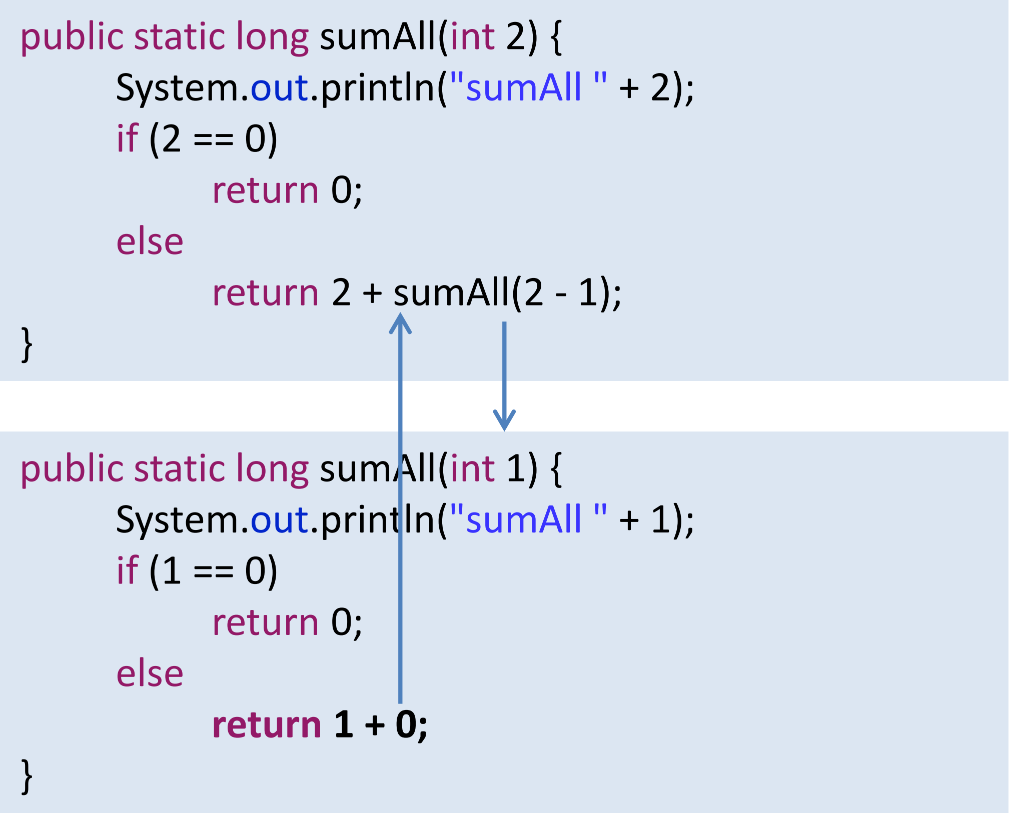
```
public static long sumAll(int 1) {
        System.out.println("sumAll " + 1);
        if (1 == 0)
                return 0;
        else
                return 1 + sumAll(1 - 1);
}
```

```
public static long sumAll(int 0) {
        System.out.println("sumAll " + 0);
        if (0 == 0)
                return 0;
}
```

OUTPUT:

```
sumAll 2
sumAll 1
sumAll 0
```

Call sumAll(2)



```
public static long sumAll(int 2) {
        System.out.println("sumAll " + 2);
        if (2 == 0)
                return 0;
        else
                return 2 + sumAll(2 - 1);
}
```

```
public static long sumAll(int 1) {
        System.out.println("sumAll " + 1);
        if (1 == 0)
                return 0;
        else
                return 1 + 0;
}
```

OUTPUT:

```
sumAll 2
sumAll 1
sumAll 0
```

Call sumAll(2)

```
public static long sumAll(int 2) {
        System.out.println("sumAll " + 2);
        if (2 == 0)
                return 0;
        else
                return 2 + 1;
}
```

```
public static void main(String[] args) {
        System.out.println("sumAll of 2 is " + sumAll(2));
}
```

OUTPUT:

```
sumAll 2
sumAll 1
sumAll 0

sumAll of 2 is 3
```

2D Arrays

- Example:

```
double[][] a = new double[3][5];
for ( r = 0; r < 3; r++ ) {
  for ( c = 0; c < 5; c++ ){
    a[r][c] = r*c; // Mult table
  }
}
```

Indices	0	1	2	3	4
0					
1					
2				?	

a[0][0]
a[0][1]
a[0][2]
a[0][3]
a[0][4]

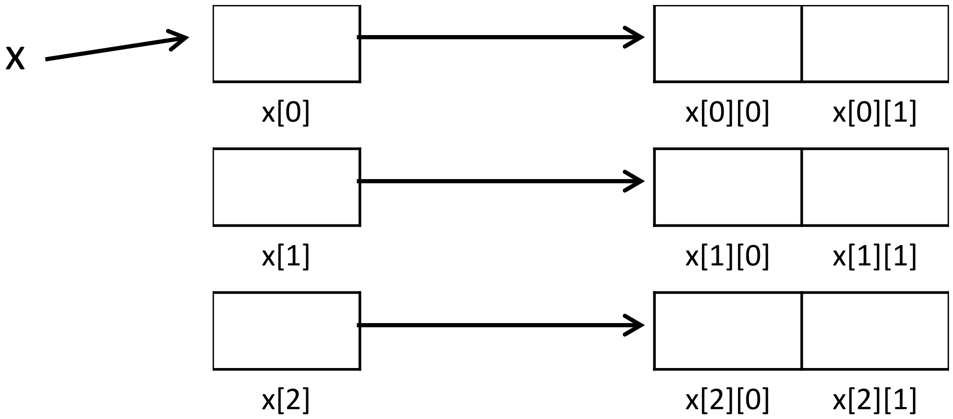
a[1][0]
a[1][1]
a[1][2]
a[1][3]
a[1][4]

a[2][0]
a[2][1]
a[2][2]
a[2][3]
a[2][4]

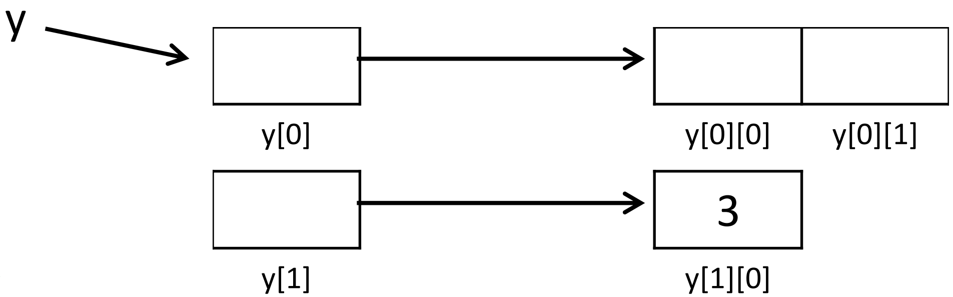
2D Arrays: Rows with diff Columns

- Not all rows have to have the same # of cols:

```
int [][] x =
  new int [3][2];
//3 rows and 2 cols
```



```
int [][] y =
  new int [2][];
y[0] = new int[2];
y[1] = new int[1];
y[1][0] = 3;
//2 rows: 2 and 1 cols!
```


y
y[0]
y[0][0]
y[0][1]
y[1]
3
y[1][0]