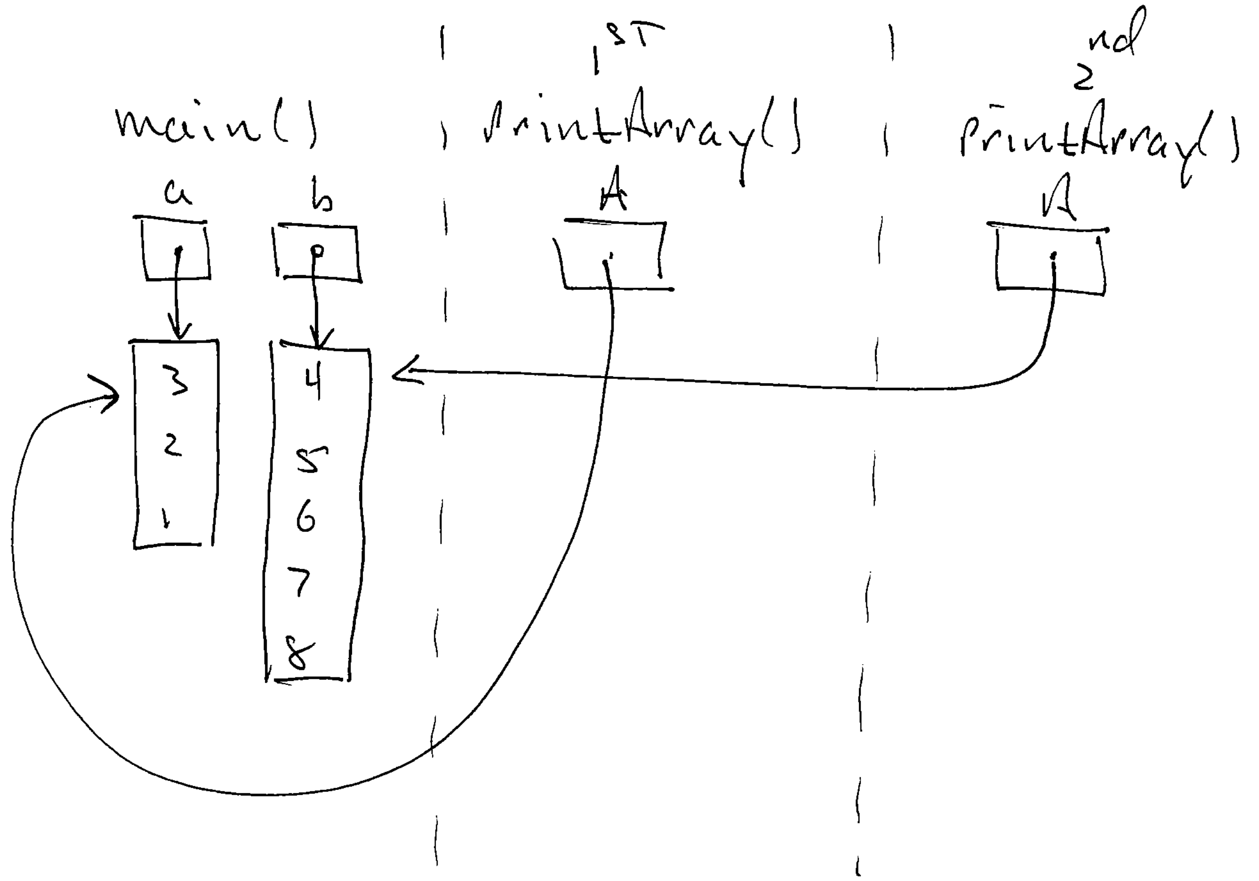


WHEN A REFERENCE VARIABLE IS PASSED TO A METHOD, WE CALL THIS PASSING BY REFERENCE. (AS OPPOSED TO PASSING BY VALUE IN WHICH DATA IS COPIED.)

THIS ALLOWS US TO WRITE METHODS THAT CHANGE THEIR OWN ARGUMENTS.

RECALL THE swap() method THAT FAILED. THE following method swaps TWO array ELEMENTS.

Ex.

```
static void swap(int[] A, int i, int j) {
    int temp;
    temp = A[i];
    A[i] = A[j];
    A[j] = temp;
}
```

Now in main():

```
int[] list = { 9, 8, 7, 6, 5, 4 };

swap(list, 1, 5);
swap(list, 2, 3);
printArray(list);    // (9 4 6 7 5 8)
swap(list, 1, 6);    // throws EXCEPTION.
```

182

```
// ArrayExample2.java
// Illustrates passing array arguments to methods

class ArrayExample2{

   public static void main( String[] args ){
      int[] list = {9,8,7,6,5,4};

      swap(list, 1, 5);
      swap(list, 2, 3);
      printArray(list); // prints ( 9 4 6 7 5 8 )
      swap(list, 1, 6); // does nothing
      System.out.println(list[getMinIndex(list)]); // prints 4
   }

   static void printArray(int[] A){
      System.out.print("( ");
      for(int i=0; i<A.length; i++)
         System.out.print(A[i]+" ");
      System.out.println(")");
   }

   static void swap(int[] A, int i, int j){
      int temp, n = A.length;

      if( i>=0 && j>=0 && i<n && j<n){ // check that i and j are in range
         temp = A[i]; // do the swap if they both are
         A[i] = A[j];
         A[j] = temp;
      } // otherwise do nothing
   }

   static int getMinIndex(int[] A){
      int i, min_index=0, n=A.length;
      for(i=1; i<n; i++){
         if(A[i]<A[min_index]) min_index = i;
      }
      return min_index;
   }
}
```

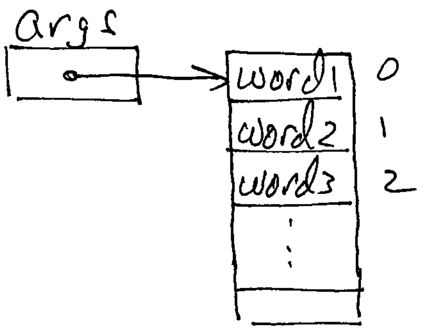
```
// Echo.java
// prints command line arguments
// compare to the unix command "echo"

class Echo{
   public static void main( String[] args ){
      for(int i=0; i<args.length; i++){
         System.out.print(args[i]+" ");
      }
      System.out.println();
   }
}
```

83

Command line ARGUMENTS

% java myProgram word1 word2 word3 . . .

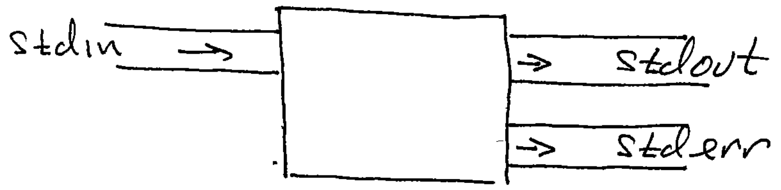
Public static void main(String[] args){
  :
}



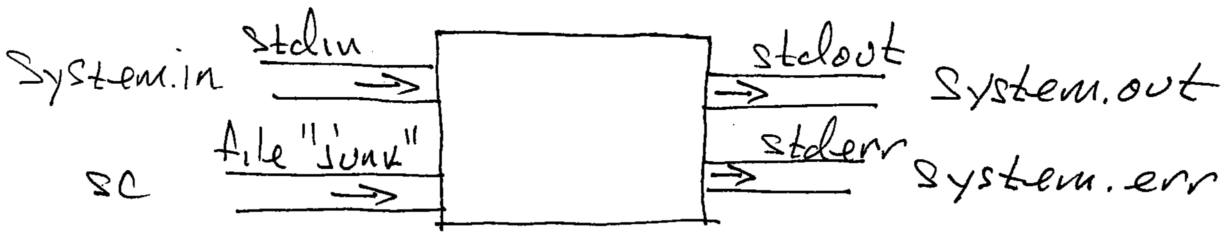
args.length = # of command line arguments.

```
////////////////////////////////////////////////////////////////////////////////
///
//
//  FileIO.java
//  Prints out the contents of the file named "junk", if it exists.  If "junk" does
not
//  exist, or cannot be accessed, throws FileNotFoundException.
//
////////////////////////////////////////////////////////////////////////////////
///

import java.util.Scanner;
import java.io.*;  // for the File class, and associated Exception classes

class FileIO{
   public static void main(String[] args) throws FileNotFoundException{
      String line;
      Scanner sc = new Scanner(new File("junk"));
                           // ^^^ This is the operation that can potentially fail by
                           // throwing FileNotFoundException.  Since main() does not
                           // catch this Exception, it must itself be declared as
                           // "throws FileNotFoundException".  Look at TryCatch.java
                           // and Cat.java to see how to catch Exceptions.

      // read lines from file and print them to standard out
      while(sc.hasNextLine()){
         line = sc.nextLine();
         System.out.println(line);
      }
   }
}
```

RECALL EVERY PROGRAM COMES EQUIPPED WITH 3 'DATA STREAMS'



YOU CAN OPEN ANY NUMBER OF OTHER STREAMS TO OR FROM VARIOUS DEVICES. THE ABOVE EXAMPLE CAUSES SC TO REFER (i.e. POINT TO) A FILE CALLED "JUNK"



IF THE FILE "JUNK" DOES NOT EXIST IN ONES CURRENT WORKING DIRECTORY THE CREATION OF A NEW FILE OBJECT WILL FAIL!

new File("junk")

THROWING A FileNotFoundException.

JAVA EXCEPTIONS FALL INTO TWO CATEGORIES.

- <u>UNCHECKED EXCEPTIONS</u>
  - REPRESENT DEFECTS IN THE PROGRAM, i.e. THINGS WHICH ARE THE "PROGRAMMERS FAULT" SO TO SPEAK.
  - SUBCLASSES OF RuntimeException
  - methods ARE <u>NOT</u> OBLIGED TO HANDLE THESE EXCEPTIONS

- <u>CHECKED EXCEPTIONS</u>
  - REPRESENT ERRORS OUTSIDE THE CONTROL OF THE PROGRAM, i.e. <u>NOT</u> THE "PROGRAMMERS FAULT".
  - SUBCLASSES OF EXCEPTION (NOT RuntimeException
  - methods ARE REQUIRED TO HANDLE THESE EXCEPTIONS BY EITHER:

(1) THROW THE EXCEPTION TO ITS CALLER AND DECLARE ITSELF TO DO SO:

```
--- method(...) throws SomeException{
 :
}
```

OR (2) CATCH THE EXCEPTION AND HANDLE PROBLEM LOCALLY.

IN FileIO.java, main() TOOK THE FIRST OPTION. THE SECOND OPTION IS IMPLEMENTED BY THE try-catch STATEMENT!

```
try{
   SOME OPERATION THAT MIGHT THROW
   AN EXCEPTION;
} Catch (ExceptionType ExceptionVariable) {
   Do something TO RECOVER, OR PERHAPS
   QUIT IN A SPECIAL WAY;
}
```

SOME OPERATIONS HAVE THE POTENTIAL OF THROWING TWO OR MORE DISTINCT EXCEPTION TYPES.

IN THIS CASE WE MIGHT HAVE MULTIPLE catch clauses:

```
try {
    something THAT might throw multiple
    EXCEPTIONS;
} catch (ExceptionType1 e1) {
    code that deals with e1;
} catch (ExceptionType2 e2) {
    code that deals with e2;
}
```

A FEW REMARKS:

- THE try-catch CONSTRUCT LOOKS A LITTLE LIKE if-else, BUT ITS BEST NOT TO THINK OF IT THIS WAY.
- try must HAVE AT LEAST ONE ACCOMPANYING CATCH.
- THE BRACES {..} ARE REQUIRED.

88

```
// TryCatch.java
// Does a safe division by catching an ArithmeticException

import java.util.Scanner;

class TryCatch{
   public static void main( String[] args ){
      Scanner sc = new Scanner(System.in);
      int a, b, c=0; // See what happens if c is not initialized here
                     // Why is it a syntax error?

      System.out.print("Enter dividend and divisor: ");
      a = sc.nextInt();
      b = sc.nextInt();

      // Here is a try-catch statement to perform integer division.  Enter
      // 0 as the divisor to cause an ArithmeticException.
      try{
         c = a/b;
      }catch(ArithmeticException e){
         System.err.println(e.getMessage());
         System.exit(1);
      }

      System.out.println("Integer quotient = " + c);
   }
}
```

NOTE THAT IN THIS EXAMPLE, THE EXCEPTION TYPE ArithmeticException IS AN UNCHECKED EXCEPTION, SO CATCHING IT WAS NOT NECESSARY.

EVERY EXCEPTION CLASS HAS A getMessage() method WHICH RETURNS A STRING SPELLING OUT A PARTICULAR MESSAGE.

```
////////////////////////////////////////////////////////////////////////////////////
///
//
//  ReadIntegers.java
//
//  Opens the file specified on the command line, then picks out any positive integers
//  in that file and prints them to standard out.  Ignores negative integers, zero,
//  double values, and non-numeric strings.  Try this out on the accompanying files
//  numbers1.dat, numbers2.dat, and junk.
//
////////////////////////////////////////////////////////////////////////////////////
///

import java.util.Scanner;
import java.io.*;

class ReadIntegers{
   public static void main( String[] args ){
      Scanner sc = null;
      int n;

      // check command line argument and open file for reading
      if(args.length!=1)
         Usage();
      try{
         sc = new Scanner(new File(args[0])); // possibly throw FileNotFoundException
      }catch(FileNotFoundException e){
         System.err.println(e.getMessage());
         Usage();
      }

      // Read from the file given on the command line.  Copy any positive integers
      // to standard out, and skip over any other strings
      while(sc.hasNext()){
         if(sc.hasNextInt()){
            n = sc.nextInt(); // read and assign integer token in file
            if(n>0)
               System.out.println(n); // print it only if its positive
         }else{
            sc.next(); // otherwise discard the string
         }
      }
   }


   // Usage()
   // prints usage message then quits
   static void Usage(){
      System.err.println("Usage: java ReadIntegers input_file");
      System.exit(1);
   }

}
```

FileNotFoundException is A CHECKED EXCEPTION AND SO MUST BE EITHER CAUGHT, OR THROWN UP TO THE CALLER.

```
////////////////////////////////////////////////////////////////////////////////
///
//
//  Cat.java
//  Prints out the files listed as command line arguments.
//  Compare to the unix command cat.
//
//  Usage: java Cat file1 file1 file3 ...
//
////////////////////////////////////////////////////////////////////////////////
///

import java.util.Scanner;
import java.io.*;  // for the File class, and associated Exception classes

class Cat{
   public static void main(String[] args){
      Scanner sc;
      String line;

      // step through command line arguments
      for(int i=0; i<args.length; i++){

         // open file, catch resulting FileNotFoundException if necessary
         try{
            sc = new Scanner(new File(args[i]));
         }
         catch(FileNotFoundException e){
            // Notice nothing is actually done with the Exception object e.
            // Instead, we recover and continue the loop.
            System.err.println("Cat: " + args[i] + ": No such file or directory");
            continue;
         }
         // if execution reaches this point, sc can read from the file args[i]

         // read lines from file and print them to standard out
         while(sc.hasNextLine()){
            line = sc.nextLine();
            System.out.println(line);
         }

         // continue lands here
      }
   }
}
```

90

IN THIS EXAMPLE WE DO NOT QUIT, BUT INSTEAD RECOVER, PRINT A MESSAGE (OTHER THAN e.getMessage()) THEN GO TO THE NEXT FILE.

MORE ON ARRAYS

STUDY THE FOLLOWING EXAMPLES FROM THE BOOK

- SelectionSort.java (P. 130)
  SORTS AN ARRAY OF INTEGERS.
- BinarySearch.java (P. 133)
  SEARCH FOR A TARGET IN A SORTED ARRAY.
- Primes.java (P. 135)
  IMPLEMENTS SIEVE OF ERATOSTHENES TO FIND PRIMES. USES AN ARRAY OF booleans:

```
boolean[] isPrime = new boolean[100];
```

AND FILLS IT WITH 'true' or 'false' IN SUCH A WAY THAT

```
isPrime[n] == true
```

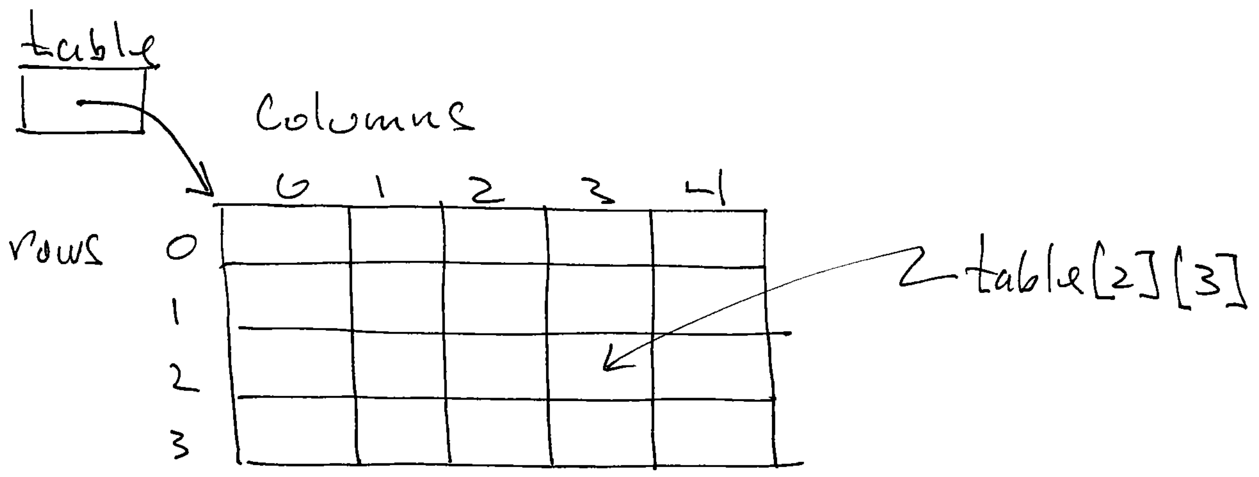
IF AND only IF n is PRIME.

MULTIDIMENSIONAL ARRAYS

Ex

```
String[] list;      // 1-dim array
double[][] table;   // 2-dim array
int[][][] cube;     // 3-dim
```

```
table = new double[4][5];
```



AS FOR 1-dimensional arrays, WE CAN DECLARE AND ALLOCATE IN ONE STEP:

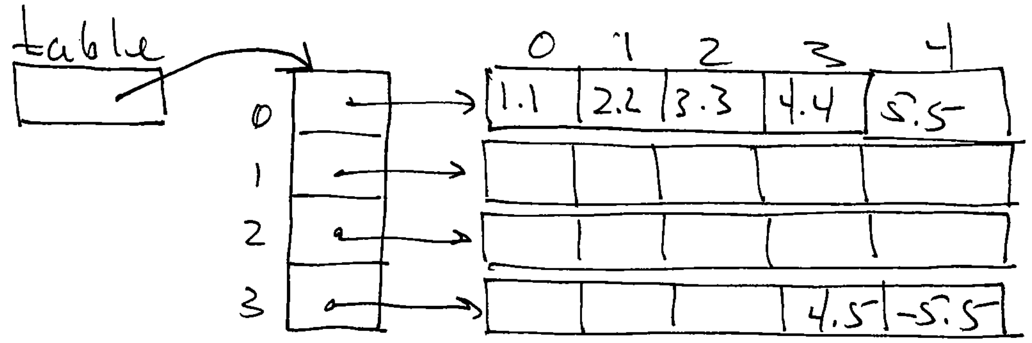
```
double[][] table = new double[4][5];
```

THIS CREATES THE SAME PICTURE AS ABOVE.

WE CAN ALSO DECLARE, ALLOCATE, AND INITIALIZE ALL IN ONE STEP:

```
double[][] table = {{1.1, 2.2, 3.3, 4.4, 5.5},
                    {1.2, 2.3, 3.4, 4.5, 5.6},
                    {0.0, 0.1, 0.2, 0.3, 0.4},
                    {-1.1, 2.3, -0.2, 4.5, -5.5}};
```

CREATES:

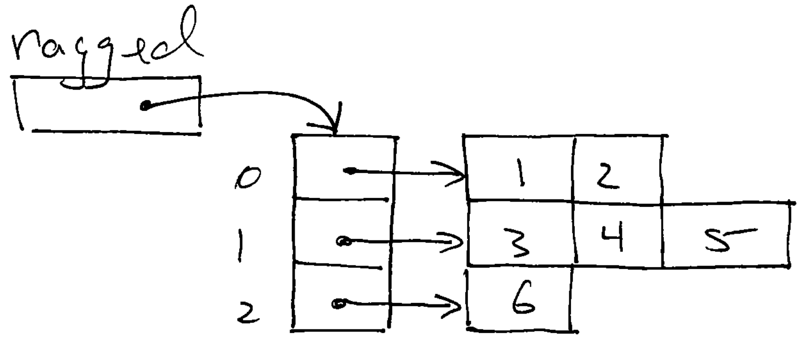


THUS A 2-dim ARRAY is literally AN ARRAY OF ARRAYS. NOTE THAT EACH OF THE ARRAYS IN THE ARRAY (OF ARRAYS) NEED NOT BE THE SAME LENGTH.

Ex

int[][] ragged = {{1, 2}, {3, 4, 5}, {6}};

CREATES THE following PICTURE IN MEMORY:



WE CAN Print SUCH AN ARRAY WITH A SPECIALIZED Print method!

```
static void Print2DimArray(int[][] A){
    for(int i=0; i<A.length; i++){
        for(int j=0; j<A[i].length; j++){
            System.out.print(A[i][j]+" ");
        }
        System.out.println();
    }
}
```

NOW THE CALL IN main();
Print2DimArray(ragged);

Prints OUT:

```
1 2
3 4 5
6
```

- READ EXAMPLE! <u>THE GAME OF LIFE</u> PP. 140-148 OF THE TEXT.

PROGRAMMING ASSIGNMENT 4 WILL BE AN AMBITIOUS PROJECT TO SOLVE SUDOKU PUZZLES.

read stuff on:
- www.websudoku.com
- www.sudoku.com
- WIKIPEDIA ARTICLES ON SUDOKU

YOUR PROGRAM WILL MANIPULATE SEVERAL 2 & 3 DIMENSIONAL ARRAYS TO KEEP TRACK OF PUZZLE PROGRESS.

S[][] is AN int array REPRESENTING THE SUDOKU PUZZLE ITSELF AND P[][][] is AN ARRAY OF "POSSIBILITIES" AS FOLLOWS

$$P[i][j][k] = \begin{cases} 0 & \text{IF IT IS KNOWN THAT } S[i][j] \neq k \\ 1 & \text{OTHERWISE} \end{cases}$$

YOUR PROGRAM WILL PROCEED BY MAKING SEVERAL PASSES OVER ARRAY P.

ON EACH PASS, DETERMINE WHICH CELLS IN ARRAY S HAVE ONLY ONE POSSIBLE CANDIDATE, AND FILL THAT INTO S, THEN USE THESE NEW ENTRIES IN S TO FURTHER REDUCE THE POSSIBILITIES FOR NEIGHBORING CELLS, AND RECORD THIS INFORMATION IN ARRAY P BY CHANGING SOME 1s TO 0s.

MORE DETAIL WILL BE GIVEN IN PA4 PROJECT DESCRIPTION, BUT YOU CAN GET A HEAD START BY WRITING FUNCTIONS TO

- READ IN A 2-dim int array from a file
- Print A 2-dim int array TO stdout.

Ex.

5	3	.	.	7	.	.	.	.
6	.	.	1	9	5	.	.	.
.	9	8	.	.	.	.	6	.
8	.	.	.	6	.	.	.	3
4	.	.	8	.	3	.	.	1
7	.	.	.	2	.	.	.	6
.	6	.	.	.	.	2	8	.
.	.	.	4	1	9	.	.	5
.	.	.	.	8	.	.	7	9

Solution

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9