CS415
INTRODUCTION TO COMPUTER SCIENCE
FALL 2017

15 2D ARRAYS
CHAPTER 13

PREVIEW

- Review
  - basic one-dimensional arrays
  - Vector
  - ArrayList
- Two-dimensional arrays
- Class wrappers for primitive types
  - Integer, Float, Double

JAVA ARRAY SUPPORT

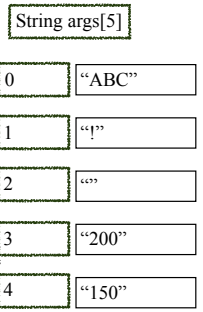
- Basic Java arrays have a fixed length; they are difficult to use if you don't know that length when you need to create the array.
- Java's *Vector* and *ArrayList* classes create dynamic length arrays: they can grow and shrink as needed during execution.

BASIC JAVA ARRAYS

- Array is simplest data structure
- A collection of n objects of the same type
- Entries in the array can be accessed by an integer index using the notation:

  args[i]

  where i ranges from 0 to n-1



VECTOR AND ARRAYLIST

- *Vector* was part of the first release of Java; *ArrayList* appeared in version 1.2 as part of the *Collection* framework of classes.
- The *Collection* interface provides a common interface to a variety of data structures that implement a collection of objects.
- *Vector* has been retrofitted to implement *Collection*.

BASIC ARRAY EXAMPLE

- Create an ellipse for each Point in wPts array

```
Point[] wPts = getWayPoints();

Ellipse[] wEll = new Ellipse[ wPts.length ];

for ( int i = 0; i < wPts.length; i++ )
{
  Point next = wPt[ i ];
  wEll[i] = new Ellipse( next.x, next.y );
}
```


VECTOR EXAMPLE

- Create an ellipse for each Point in wPts Vector

```
Vector<Point> wPts = getWayPoints();

Vector<Ellipse> wEll = new Vector<Ellipse>();

for (int i = 0; i < wPts.size(); i++ )
{
  Point next = wPts.get( i );
  wEll.add( new Ellipse( next.x, next.y ));
}
```


VECTOR WITH ITERATOR

- Create an ellipse for each Point in wPts Vector

```
Vector<Point> wPts = getWayPoints();

Vector<Ellipse> wEll = new
Vector<Ellipse>();

Iterator<Point> iter = wPts.listIterator();
while ( iter.hasNext() )
{
  Point next = iter.next()
  wEll.add( new Ellipse( next.x, next.y ));
}
```

VECTOR WITH FOR EACH LOOP

```
Vector<Point> wPts = getWayPoints();

Vector<Ellipse> wEll = new Vector<Ellipse>();


for ( Point next : wPts )
{
  wEll.add( new Ellipse( next.x, next.y ));
}
```


TWO-DIMENSIONAL ARRAYS

- There are many cases where the data for an application is naturally organized as a 2-d array
  - Many game boards: chess, checkers, MineSweeper, etc.
  - Images: photos, satellite, medical data
  - Geographic data based on latitude/longitude
  - many, many more examples, especially in science and engineering research

JAVA 2D ARRAYS

- Java declares 2D arrays with the notation:

```
int[][]  vals = new int[ nRows ][ nCols ];

Tile[][] board = new Tile[ nRows ][ nCols ];
```

- Access to 2D array elements extends 1D access

```
for ( int r = 0; r < nRows; r++ )
{
   for ( int c = 0; c < nCols; c++ )
       System.out.print( vals[ r ][ c ] + " " );
   System.out.println();
}
```

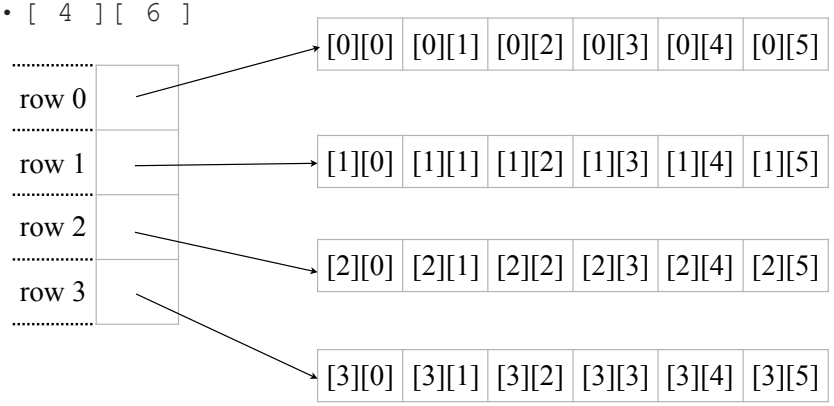

ARRAY CONCEPTUAL LAYOUT

- Notation: [4][6] means 4 rows, 6 columns
  - The entries in the figure below refer to the indexes needed to reference the value stored at that location.

	col 0	col 1	col 2	col 3	col 4	col 5
row 0	[0][0]	[0][1]	[0][2]	[0][3]	[0][4]	[0][5]
row 1	[1][0]	[1][1]	[1][2]	[1][3]	[1][4]	[1][5]
row 2	[2][0]	[2][1]	[2][2]	[2][3]	[2][4]	[2][5]
row 2	[3][0]	[3][1]	[3][2]	[3][3]	[3][4]	[3][5]

JAVA 2D ARRAY ACTUAL LAYOUT

- Java implements a 2D array as an array of arrays
  - `[ 4 ][ 6 ]`



ARRAY INDEX NAMING

- It's natural to traverse a 2D array with an *outer* **for** loop over the 1st index and the *inner* **for** loop over the 2nd index:

```
for ( int r = 0; r < nRows; r++ )
{
  for ( int c = 0; c < nCols; c++ )
    System.out.print( vals[ r ][ c ] + " " );
  System.out.println();
}
```

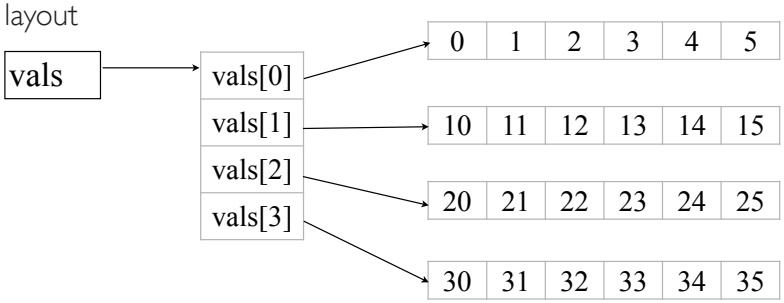
- Thus, a "row" of output corresponds to the 1st index, and a "column" to the 2nd index

2D INT ARRAY CODE EXAMPLE

```
// Create a 2d int array where each entry's value
//   is 10 * row number + column number

int nRows = 4, nCols = 6;
int[][] vals = new int[ nRows ][ nCols ];

for ( int r = 0; r < nRows; r++ )
{
  for ( int c = 0; c < nCols; c++ )
  {
    vals[ r ][ c ] = r * 10 + c;
    System.out.print( vals[ r ][ c ] + " " );
  }
  System.out.println();
}
```


EXAMPLE DETAILS

- Memory layout



- Output

```
0 1 2 3 4 5
10 11 12 13 14 15
20 21 22 23 24 25
30 31 32 33 34 35
```

Each value should be r*10 + c

REVERSE THE FOR LOOP NESTING

- If we reverse for loops in previous example
  - Memory layout unchanged
  - Output changes!

```
// Reverse for loop nesting
for ( int c = 0; c < nCols; c++ )
{
  for ( int r = 0; r < nRows; r++ )
  {
    vals[ r ][ c ] = r * 10 + c;
    System.out.print( vals[ r ][ c ] + " " );
  }
  System.out.println();
}
```

Output

```
0  10  20  30
1  11  21  31
2  12  22  32
3  13  23  33
4  14  24  34
5  15  25  35
```


ROW MAJOR VS COLUMN MAJOR ORDER

- By convention, the first index is called the *row* index and the second index is called the *column* index
- We say that Java stores its arrays in *row major order*, which is pretty apparent from *Java 2D Array Actual Layout* slide, but not reflected in the previous output!
- Caveat: the visual layout of the rows and columns is entirely a figment of our imagination -- OR the way the program prints or displays the information

GRAPHICS CONVENTION

- Graphics and array conventions are inconsistent
  - For array indices the first index indicates row and the second column:
    i, j means row i and column j
  - For graphic coordinates the first coordinate indicates the x coordinate (column) and the second the y coordinate (row)
    i, j means the column i and row j

GRAPHICS PROGRAMMING WITH JAVA ARRAYS

- Need to keep the code as obvious as possible
  - Be careful with variable naming by being very clear about *x, y* versus *row, col*
  - If you want indexes to be [x][y], create arrays with the 1st index being the xSize and the second the ySize:

```
int [][] pixels = new int[ xSize ][ ySize ];
```

- Do this for all arrays that map to the *x,y* coordinate system
- This is what the book means by "use column major" order.

GRAPHICS PROGRAMMING WITH JAVA ARRAYS

- Example
  - Display a 2D array of rectangles

```
// size of rectangles is 50 X 50
// location of rectangle[0][0] is x,y

for ( int r = 0; r < nRows; r++ )
   for ( int c = 0; c < nCols; c++ )
      rects[ r ][ c ].setLocation( x + c * 50, y + r * 50    )
```


WHEELS IMAGE CLASS

- *Wheels* has a very basic *Image* class:
  - *Image(String fileName)* -- creates an *Image* object from a jpeg or png file
  - *setLocation, setRotation* are implemented
  - Extends *AbstractShape*, so can extend it to get mouse events.
- Built on *java.awt.image.BufferedImage*

STACKS

- Arrays and Vectors are data structures.
- Collections of values organized in some way.
- The Stack is a data structure
- A stack can be used in any situations where the last object that we added to the collection is the first one we want to remove from the collection (LIFO)

STACKS

- Stacks are useful whenever an algorithm calls for a "postponed obligation"
- The last postponed obligation is the first one we want to return to.
- An important example is the system's "execution stack"

EXAMPLE: EXECUTION STACKS

- An important example is the systems "execution stack"
  - For each method called an "activation record" for the method is placed on the top of the execution stack
  - If another method is called from the current method its activation is put on the stack.
  - When that method terminates we remove its record from the stack and return to the previous method

STACK API

- Stack( )
  Creates an empty Stack.
- boolean empty()
  Tests if this stack is empty.
- E peek( )
  Looks at the object at the top of this stack without removing it from the stack.
- E pop( )
  Removes the object at the top of this stack and returns that object as the value of this function.
- E push(E item)
  Pushes an item onto the top of this stack.

STACK EXAMPLE (ORDER REVERSED)

```
Stack<Color> stack = new Stack<Color>( );

stack.push( Color.BLUE );
stack.push( Color.WHITE );
stack.push( Color.RED );

while( ! stack.empty( ) )
{
     Color temp = stack.pop( );
     System.out.println( temp );
}
```

```
OUTPUT:
java.awt.Color[ 255, 0, 0 ]
java.awt.Color[ 255, 255, 255 ]
java.awt.Color[ 0, 0, 255 ]
```


PRIMITIVE TYPES IN CLASSES

- Java's primitive types, like *int* and *float*, are efficient, but have severe limitations compared to class objects; e.g., there is no polymorphism.
- Java includes *wrapper* classes that encapsulate the basic functionality of each of the primitive types into a class (and adds additional functionality):

  Integer, Float, Double, Long, Short, Byte, Character

INTEGER CLASS

- Some key methods

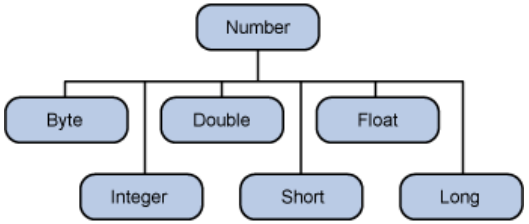
Integer(int)	Construct Integer from an int
Integer(String)	Construct Integer from String that is a valid integer representation. Throws NumberFormatException
int getIntValue();	return the *int* value of this Integer object
float getFloatValue()	return the *float* value of this Integer object
double getDoubleValue()	return the *double* value of this Integer object
short getShortValue()	return the value of this Integer object as a *short.* It may involve truncation or rounding.
byte getByteValue()	return value as *byte*; truncation or rounding possible
long getLongValue()	return value as a long.

IMPLICIT CONVERSION

```
import java.util.*;

public class VecNum
{
    public static void main( String argv[ ] )
    {
    Vector<Double> v = new Vector<Double>( );
    v.add( 1.5 );
    v.add( 2.5 );

    double sum = 0;
    for( Double iw: v )
        sum +=  iw ;

    System.out.println( sum );
   }
}
```


OTHER NUMBER CLASSES

- Float, Double, Long, Short, Byte classes are similar to the Integer class. They all have the same get<type>( ) methods as Integer.
- In fact, there is a class hierarchy; all the *get<type>()* methods are defined in the *Number* class.



USING NUMBER CLASSES

- Suppose you have a command language in which different commands have different types and numbers of parameters.
  - Rather than have each command implement its own Scanner and parsing algorithm, create one method that parses all arguments of all commands and returns an Vector of Number objects:
    - Vector<Number> getArgs(String line)

GETARGS(STRING)

```
public Vector<Number> getArgs( String line )
{
  Vector<Number> ret = new Vector<Number>();
  Scanner s = new Scanner( line );
  while ( s.hasNext() )
 {
    if ( s.hasNextByte() ) // test order important!
      ret.add( new Byte( s.nextByte() ));
    else if ( s.hasNextShort() )
      ret.add( new Short( s.nextShort() ));
    . . .
    else if ( s.hasNextDouble() )
      ret.add( new Double( s.nextDouble() ));
    else
      error( “Argument not a number: “ + s.next());
  }
  return ret;
}
```


REVIEW

- One-dimensional arrays
  - basic arrays
  - Vector and ArrayList
- Two-dimensional arrays
- Number wrapper classes