Lecture 12: 2D Arrays

AP Computer Science Principles

Semantics

Value semantics

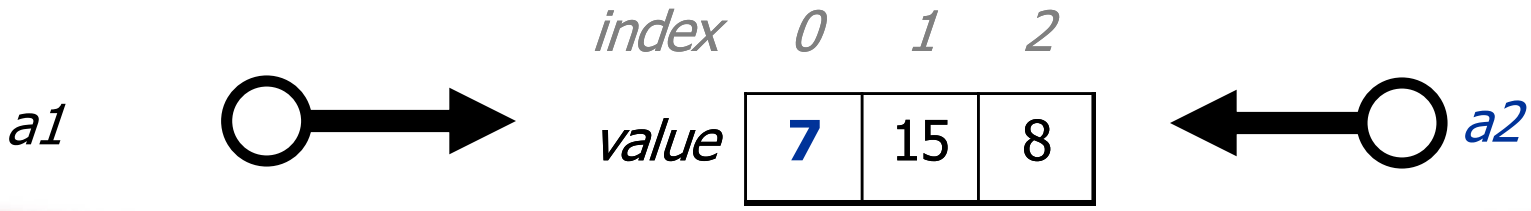
- **value semantics**: Behavior where values are copied when assigned, passed as parameters, or returned.
  - All primitive types in Java use value semantics.
  - When one variable is assigned to another, its value is copied.
  - Modifying the value of one variable does not affect others.

```
int x = 5;
int y = x;      // x = 5, y = 5
y = 17;         // x = 5, y = 17
x = 8;          // x = 8, y = 17
```

Reference semantics (objects)

- **reference semantics**: Behavior where variables actually store the address of an object in memory.
  - When one variable is assigned to another, the object is *not* copied; both variables refer to the *same object*.
  - Modifying the value of one variable *will* affect others.

```
int[] a1 = {4, 15, 8};
int[] a2 = a1;   // refer to same array as a1
a2[0] = 7;
println(a1); // [7, 15, 8]
```



Objects as parameters

- Arrays and objects(except String) use reference semantics. Why?
  - *efficiency.* Copying large objects slows down a program.
  - *sharing.* It's useful to share an object's data among methods.

- When an object is passed as a parameter, the object is *not* copied. The parameter refers to the same object.
  - If the parameter is modified, it *will* affect the original object.

- Arrays are passed as parameters by *reference.*
  - Changes made in the method are also seen by the caller.

Value Semantics

Example:

```
void setup() {
      int x = 2;
      triple(x);
      println(x); // 2

}
void triple(int number) {
      number = number * 3; // 6
}
```

Value Semantics

String uses value semantics like primitive types.

Example:

```
void setup() {

      String str = "hi";
      twice(str);
      println(str); // "hi"
}
void twice(String str) {
      str = str + str;
      println(str); // "hihi"
}
```

Reference Semantics

Example:

```
void setup(){

      int[] arr = {0,1,2,3};
      triple(arr);
      println(arr);
      //{0,3,6,9}
}
void triple(int[] numbers) {
      for (int i = 0; i < numbers.length; i++) {
          numbers[i] = numbers[i] * 3;
      }
}
```

Nested Loops

Nested loops

- **nested loop**: A loop placed inside another loop.

```
for (int i = 1; i <= 5; i++) {
    for (int j = 1; j <= 10; j++) {
        print("*");
    }
    println();    // to end the line
}
```

- Output:

```
**********
**********
**********
**********
**********
```

- The outer loop repeats 5 times; the inner one 10 times.
  - "sets and reps" exercise analogy

Nested `for` loop exercise

- What is the output of the following nested `for` loops?

```
for (int i = 1; i <= 5; i++) {
    for (int j = 1; j <= i; j++) {
        print("*");
    }
    println();
}
```

- Output:

```
*
**
***
****
*****
```

Nested `for` loop exercise

- What is the output of the following nested `for` loops?

```
for (int i = 1; i <= 5; i++) {
    for (int j = 1; j <= i; j++) {
        print(i);
    }
    println();
}
```

- Output:

```
1
22
333
4444
55555
```

Nested `for` loop exercise

- What is the output of the following nested `for` loops?

```
for (int i = 1; i <= 5; i++) {
    for (int j = i; j <= 5; j++) {
        print(i);
    }
    println();
}
```

- Output:

```
11111
2222
333
44
5
```

Common errors

```
for (int i = 1; i <= 5; i++) {
    for (int j = 1; i <= 10; j++) {
        print("*");
    }
    println();
}
```

```
for (int i = 1; i <= 5; i++) {
    for (int j = 1; j <= 10; i++) {
        print("*");
    }
    println();
}
```

Both of the above sets of code produce *infinite loops*:

2-Dimensional Arrays

2D Array

0	0	0	0
0	0	0	0
0	0	0	0

```
int[][] matrix=new int[3][4];
```
//3 rows, 4 columns
//initialized to 0.

2D Array

2	0	0	0
0	0	-6	0
0	7	0	0

```
matrix[0][0]=2;
matrix[1][2]=-6;
matrix[2][1]=7;
```

Declare and Initialize

Declaring and initializing 2D arrays.

```
int[][] table; //2D array of ints, null reference


//one way to initialize 2D array.
float[][] matrix=new float[4][5];
//4 rows, 5 columns
//initialized all to 0.0


String[][] strs=new String[2][5];
//strs reference 2x5 array of
//String objects. Each element is
// null
```

Initializer List

//another way to initialize 2D array is through a list.

```
int[] array1={1,4,3};

int[][] mat={{3,4,5}, {6,7,8}}; //2 rows, 3 columns
```

3	4	5
6	7	8

Array of Arrays

- A 2D array is implemented as an array of row arrays. Each row is a one-dimensional array of elements. Suppose that mat is the matrix

3	-4	1	2
6	0	8	1
-2	9	1	7

Then mat is an array of three arrays:

mat[0] is the one-dimensional array {3,-4,1,2}.

mat[1] is the one-dimensional array {6,0,8,1}.

mat[2] is the one-dimensional array {-2,9,1,7}.

mat.length is the number of rows.

Array of Arrays

3	-4	1	2
6	0	8	1
-2	9	1	7

- mat.length is the number of rows. In this case, it equals 3 because there are three row-arrays in mat.
- For each k, where 0<=k<mat.length, mat[k].length is the number of elements in that row, namely the number of columns. For example mat[0].length is the number of elements in row 0. In this case, mat[k].length=4 for all k.
- Java allows “jagged arrays” where each row array may have different lengths. However, we won’t used jagged arrays in this class.

Initializer List

```
int[][] mat={{3,4,5},{1,2},{0,1,-3,5}};
```

```
mat[0]={3,4,5}
mat[1]={1,2}
mat[2]={0,1,-3,5}
```

```
mat.length=3
mat[0].length=3
mat[1].length=2
mat[2].length=4
```

Row-Column Traversal

Suppose that `mat` is a 2D array initialized with integers. Use nested for loop to print out the elements of the array.

```
for(int i=0;i<mat.length;i++){
   for(int j=0;j<mat[i].length;j++)
       print(mat[i][j]+ " ");
   println();
}
```

Row-by-Row

Suppose the following method has been implemented.

```
void printArray(int[] array)
{/*implementation not shown*/}
```

Use it to print out the 2D array `mat`.

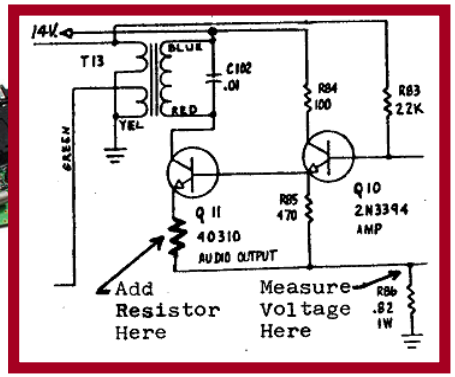
```
for(int i=0;i<mat.length;i++){
   printArray(mat[i]); //mat[i] is row i of mat
   println();
}
```

Encapsulation

Encapsulation

- **encapsulation**: Hiding implementation details from clients.
  - Encapsulation forces ***abstraction***.
    - separates external view (behavior) from internal view (state)
    - protects the integrity of an object's data
    - **Abstraction:** You can use something without knowing how it works.

Abstraction is one of the 7 big ideas in AP Computer Science Principles.



Private fields

A field that cannot be accessed from outside the class

```
private type name;
```

- Examples:

```
private int id;
private String name;
```

- Client code won't compile if it accesses private fields.

In Processing, the convention is to AVOID using private variables to keep things simple.

Private fields

In the file Student.pde(Object class).

```
class Student{
      String name;
      private int ID;
      Student(String n, int id){
        …
      }
      …
}
```

Private fields

In the file Driver.pde(Driver/client class).

```
Student s1;
void setup(){
     s1 = new Student("John Smith", 3454);
}
void draw(){
     println(s1.name); //ok
     println(s1.ID); //error, private

}
```

Accessing private state

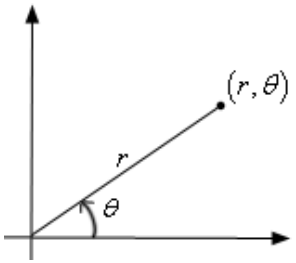
```
// A "read-only" access to the x field ("accessor")
public int getID() {
    return ID;
}
// Allows clients to change the x field ("mutator")
public void setID(int newID) {
    ID = newID;
}
```

- Client code will look more like this:

```
println(s1.getID());
s1.setID(14343);
```

Benefits of encapsulation

- **Abstraction** between object and clients
- Protects object from unwanted access
  - Example: Can't fraudulently increase an `Account`'s balance.
- Can change the class implementation later
  - Example: `Ball` could be rewritten in polar coordinates (r, θ) with the same methods.



- Can constrain objects' state (**invariants**)
  - Example: Only allow `Account`s with non-negative balance.
  - Example: Only allow `Date`s with a month from 1-12.

Lab 1

Write the following methods.

`sum`: Write method `sum` which accepts a 2D array of integers and returns the sum of all of the elements. Use row-column traversal method.(Nested for loop)

`rowSum: rowSum` accepts two parameters: a 2D array of integers and an integer `row`. `rowSum` returns the sum of the integers of elements in the row given by `row`.

`sum2`: This method is the same as `sum` above but **you must use `rowSum` in your code. (One for loop)**

Lab 1

Write the following methods.

`largest` accepts a 2D array of integers and returns the largest value. Use row-column traversal method to examine each value.(Nested for loop)

`largestByRow` accepts two parameters: a 2D array of integers and an integer `row`. `largestByRow` returns the largest value in the row given by `row`.

`largest2` accepts a 2D array of integers and returns the largest value. **You must call `largestByRow`. (One for loop)**

Lab 1

`printTranspose`: Given 2D array of integers, print the transpose of the array. The transpose of a 2D array is a new array whose rows are the columns of the original array. DO NOT CREATE A NEW ARRAY!

If `mat={{1,2,3},{4,5,6}}; printTranspose(mat)` will print:

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

Lab 1 Outline

Write the setup() method to test your methods.

```
void setup(){


}
int sum(int[][] mat){…}
int rowSum(int[][] mat, int row){…}
int sum2(int[][] mat){…}
int largest(int[][] mat){…}
int largestByRow(int[][] mat, int row){…}
int largest2(int[][] mat){…}
void printTranspose(int[][] mat){…}
```

Homework

1) Read and reread these lecture notes.

1) Complete the lab.

References

Part of this lecture is taken from the following book.

1) Stuart Reges and Marty Stepp. Building Java Programs: A Back to Basics Approach. Pearson Education. 2008.