CIS 110: Introduction to Computer Programming

Lecture 19
Array Wrap-up
(§ 7.2-7.3)

Outline

- Side note: bits and integral representations
- The null value
- Multi-dimensional arrays

Bits and integral representations

What is actually "in" a variable?

- So far, we've worked at this level of abstraction for variables:

- But computers don't physically store "42" in memory

Digital representation of data

- Recall from chapter 1: a computer stores all data digitally, i.e., as a binary number (base 2).

- This is because of the natural of the electromagnetic *switches* that store the data.
  - Either "on" (value 1) or "off" (value 0).

Java Integers

- Java ints are 32-bit values.
  - Each binary digit is called a *bit*.

- Reasoning about numbers in binary is tedious so we usually use decimal (base 10) instead.

Color components

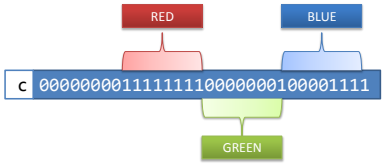
- A color component of a pixel is made up of 3 numbers, one for each color.

R	11111111
G	00000000
B	10001111

- Each component can take on a value from 0-255 (i.e., 8 bits, or 2^8 values).

Bit-packing

- We can store each component as an int, but that is wasteful.
  - A component uses 8 bits, but an int uses 32 bits.
- Instead, let's pack all 3 values into a single int!



The null value

Arrays of objects

- We've made arrays of *primitive types* so far.

```
int x[] = { 0, 1, 2 };
```
x → [0, 1, 2]

- We can make arrays of *object types* as well.

```
String y[] = new String[3];
```
y → [?, ?, ?]

A question of initialization

- When we make an array of primitive type, we initialize its elements to "zero values".

```
int arr1[] = new int[3];
```
arr1 → [0, 0, 0]

```
double arr2[] = new double[3];
```
arr2 → [0.0, 0.0, 0.0]

```
boolean arr3[] = new boolean[3];
```
arr4 → [false, false, false]

- What is the zero value of a String? In general any object type?

Null means "no reference"

- We specify that "this variable is not referencing an object" by using the *null* value.

```
String s1 = "hello";
```
s1 → "Hello"

```
String s2 = null;
```
s2 →

- Null is the "zero value" for all objects types.

```
String y[] = new String[3];
```
y → [?, ?, ?]

The dreaded NullPointerException

- When you try to call a method/access a field of an null reference, you get a *NullPointerException*.

```
String s2 = null;
s2.length();
```

s2 →

What String? s2 isn't referencing a String… AHHHHHH

"Null References: The Billion Dollar Mistake"

- Abstract of Tony Hoare's talk about null references (QCon London 2009):

> I call it my billion-dollar mistake. It was the invention of the null reference in 1965. At that time, I was designing the first comprehensive type system for references in an object oriented language (ALGOL W). My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. **This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.**

Multi-dimensional arrays

Arrays of Arrays

- We can declare arrays of *primitive types*.

```
int x[] = { 0, 1, 2 };
```

x → [0, 1, 2]

- We can declare arrays of *object types*.

```
String[] y = { "hi", "bye" };
```

y → [,] → "hi", "bye"

- But arrays are objects, so we can declare *arrays of arrays*.

```
int[][] z = new int[3][3];
```


Rectangular two-dimensional arrays

```
int[][] z = new int[3][3];
```

- Really an array of arrays

z → [, ,] → [0, 0, 0], [0, 0, 0], [0, 0, 0]

- Conceptually a *grid*

z → [0, 0, 0]
[0, 0, 0]
[0, 0, 0]

Accessing two-dimensional arrays

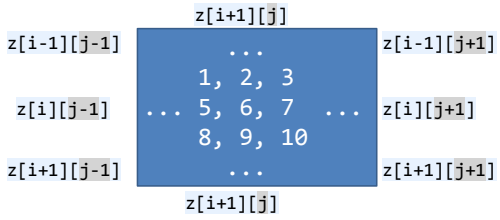
z → [0, 0, 0]
[0, 0, 0]
[0, 5, 0]

```
int[][] z = new int[3][3];
z[2][1] = 5;
```

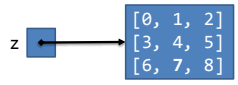
- Remember zero-based indexing.
- Think of z[i][j] as "ith row, jth column"

Say hi to you neighbors

Say the middle element is `z[i][j]...`



Layers upon layers




```
int z[][] = new int[3][3];

... z ...      // The entire array
... z[1] ...   // A single row (an array itself)
... z[1][2] ... // A single element
```


Traversing two-dimensional arrays

```
int z[][] = new int[3][4];

// for each row...
for (int i = 0; i < z.length; i++) {
  // for each column...
  for (int j = 0; j < z[i].length; j++) {
    // print out the element
    System.out.println(z[i][j]);
  }
}
```

```
[0, 1, 2]
[3, 4, 5]
[6, 7, 8]
[9, 0, 1]
```


N-dimension arrays

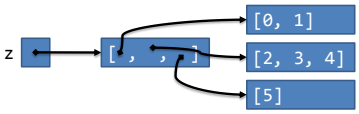
- We can have as many dimensions as we want!

```
int threeDims[][][] = new int[3][4][5];
int fourDims[][][][] = new int[5][7][9][10];
```

- Same logic as before applies!
- How we interpret the dimensions is irrelevant as long as we are consistent.

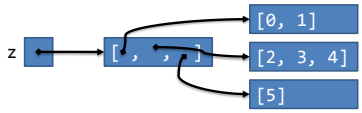
Jagged arrays

- Array rows need not have the same length, i.e., they can be *jagged*.



Initializing jagged arrays

- We have to *manually create jagged arrays* by initializing each row at a time.




```
int[][] z = new int[3][];
z[0] = { 0, 1 };
z[1] = { 2, 3, 4 };
z[2] = new int[1];
z[2][0] = 5;
```

- Otherwise they behave like rectangular arrays!

The Arrays helper class (in java.util)

```
// Returns the string representation of arr suitable for
// printing, e.g., [0, 2, 3, 4, 5]
Arrays.toString(arr);

// Returns true if the elements of the array are pairwise equals
Arrays.equals(arr1, arr2);

// Fills the array with the given value
Arrays.fill(arr, value);

// Returns a copy of the given array with the specified length,
// either truncating elements or filling with zero-values to
// meet that length
Arrays.copyOf(arr, len);

// toString and equals variants that work for multi-dimensional arrays
Arrays.deepToString(arr);
Arrays.deepEquals(arr1, arr2);
```