Lecture 22: Array Operations

Last time:

1. break / continue
2. Intro to arrays
3. Copying arrays and making arrays bigger
4. Array lengths and out-of-bounds indexing

Today:

1. Explicit array initialization
2. Arrays as arguments
3. Arrays of objects
4. Privacy leaks

CMSC 131 Spring 2007
Bonnie Dorr (adapted from Rance Cleaveland)

Project #4 —snafu = due tonight at 11pm

- The server went down right at the point when it was due!
- You have til 11pm tonight to get it submitted.
- No assistance is available for this—the extension is simply to overcome the last hour or two of the server going down, not to provide lots of extra time for people to get assistance.

Recall Arrays

- Arrays are:
  - Sequences of cells holding values of the same type (“base type”)
  - Objects (hence created using new)
- To define an array variable:

```
int[] a;  // an array with base type int
```

- To create an array object:

```
a = new int[10];
```

  - Creates an array of 10 cells
  - The base type is `int`
- To access individual array cells: use indexing
  - `a[0], a[1], …, a[9]`
  - Cells are just like variables:
    - They may be read: `x = a[3];`
    - They may be written: `a[2] = 7;`

Square Brackets: [ ]

- Three uses in Java:
  - Array variable declaration

    ```
    int[] a;
    ```
  - Array object creation

    ```
    new int[10];
    ```
  - Array indexing

    ```
    a[0]
    ```
- Don't be confused!

Alternate Declaration Syntax

- To maintain consistency with C / C++, following declaration of array variables also possible

```
int grade[];
```

  Compare to Java standard:

```
int[] grade;
```

- Java standard generally preferred ("type" emphasizes array status)
- Alternative syntax sometimes handy:

```
int grade[], i, gpa[];
```

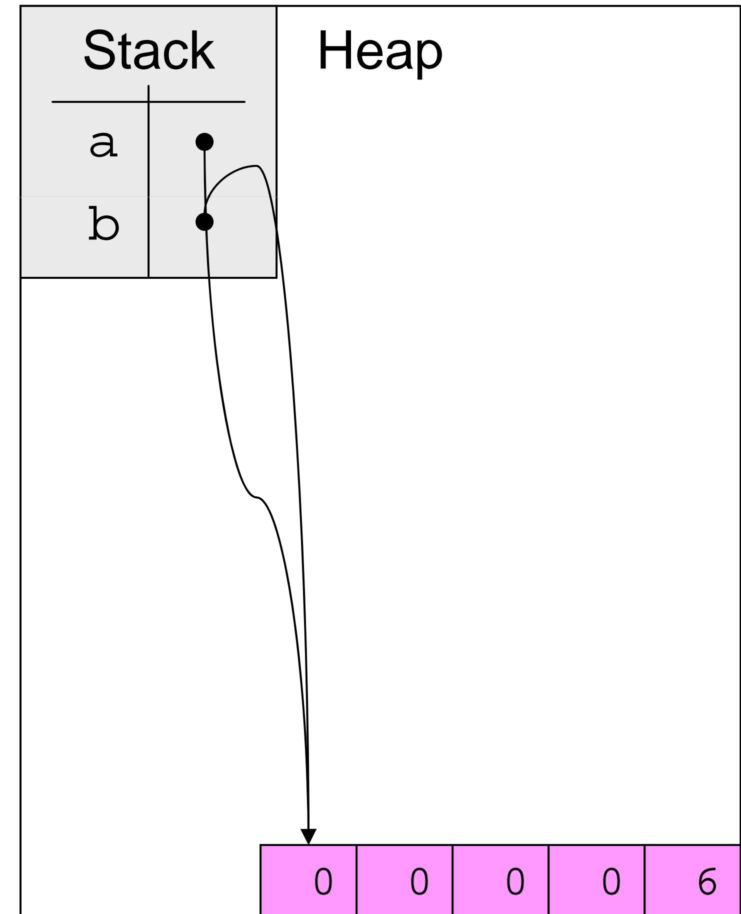
  - Declares two arrays of base type `int`: `grade`, `gpa`
  - Declares a single `int` variable: `i`

Arrays and Aliasing

- Arrays are objects, so array variables are *references*
- What happens?

```
int[] a = new int[5];
int[] b = null;
b = a;
b[4] = 6;
System.out.println(a[4]);
```

- `6` is printed
  - a, b are aliases
  - Modifying `b[4]` also modifies `a[4]`



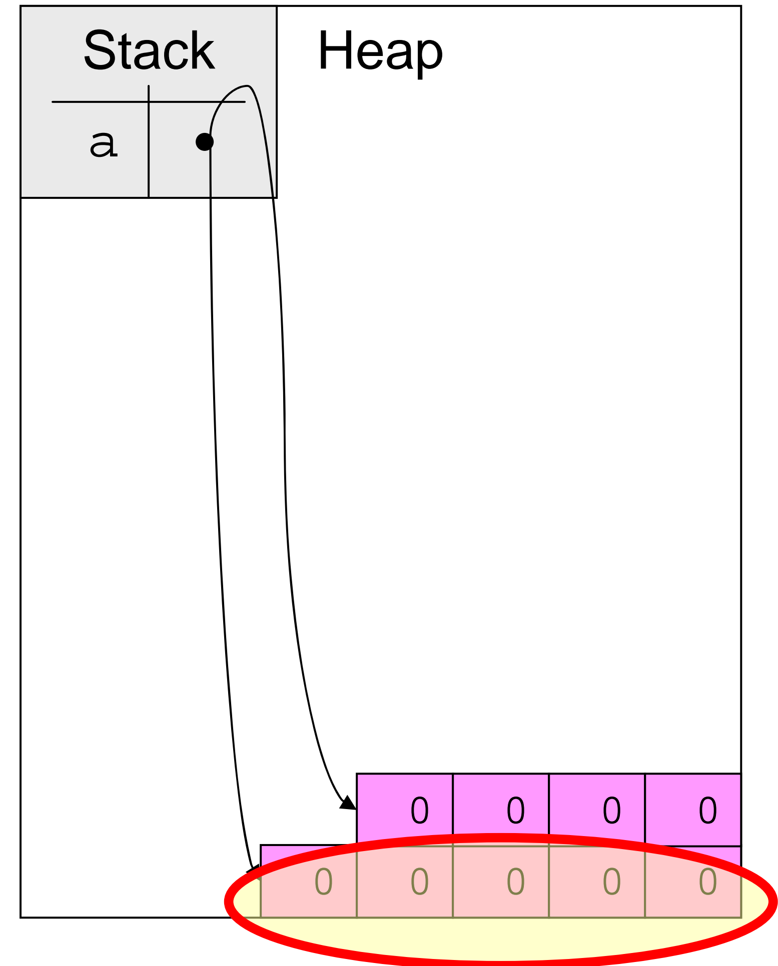
Array and Garbage

- What happens?

```
int[] a = new int[5];
```

```
a = new int [4];
```

- Garbage!
  - `a` is assigned address of new array object
  - Old array object no longer accessible



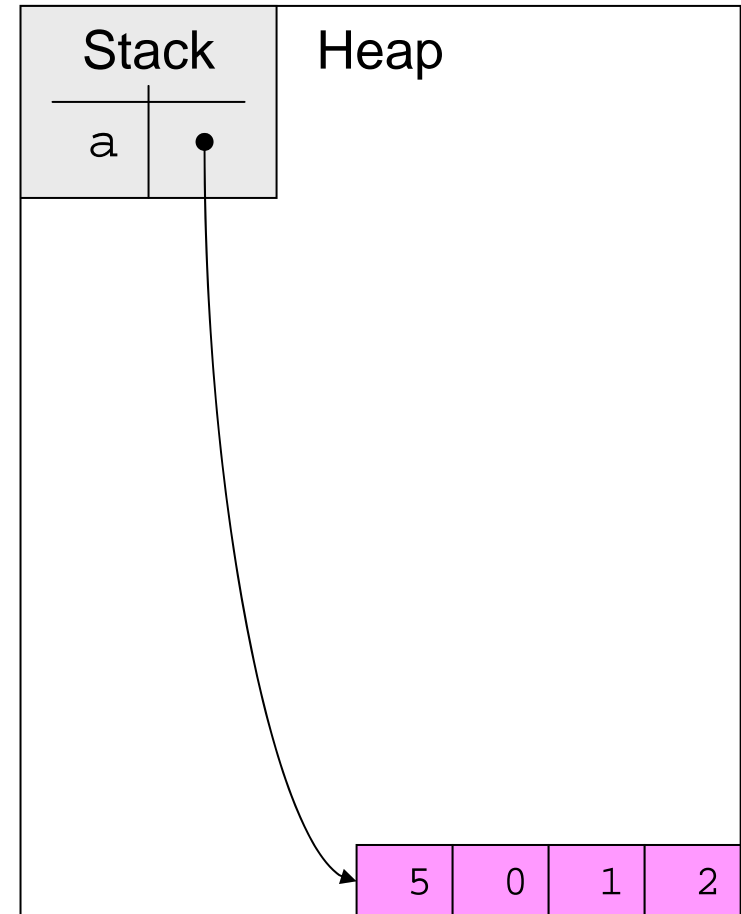
Array Initializers

- Arrays may be initialized at declaration time!

  `int[] a = {5,0,1,2};`

- Java:
  - counts elements (here, 4);
  - creates correct size of array
  - copies elements into array
  - returns reference to array

See Array Example 3



Array Arguments

- **Array and array elements**: can be passed as parameters to methods, just like primitive types and object references.
- **Passing a Single Element**: A single element of an array can be passed to any method as if it were a **simple variable**.

  Example: Suppose **fooBar(z)** expects an argument z of type **double**.

```
double[ ] a = new double[10];
int i = …
// … initializations omitted
fooBar( a[3]  );       // passes the value of a[3] to fooBar
fooBar( a[i] );        // evaluates a[i] and passes the value to fooBar
```

- **Passing an Entire Array**: You can pass an entire array as an argument. As with objects, this just passes the **reference**.
  - If the method expects an array of type double, then you can pass it an array of doubles **of any size**.
  - **Promotion does not apply**. E.g., you cannot pass it an array of int's.

Arrays As Arguments

- Arrays = objects
- Array variables = references
- Array cells = variables
- Both can be used as arguments to methods
  - Array cells: passed just like variables
  - Array arguments: passed just like objects
    - Reference to array is passed in
    - If the method expects an array of doubles, an array of doubles of any size can be passed
    - Promotion does not apply. You cannot pass an int array

Example

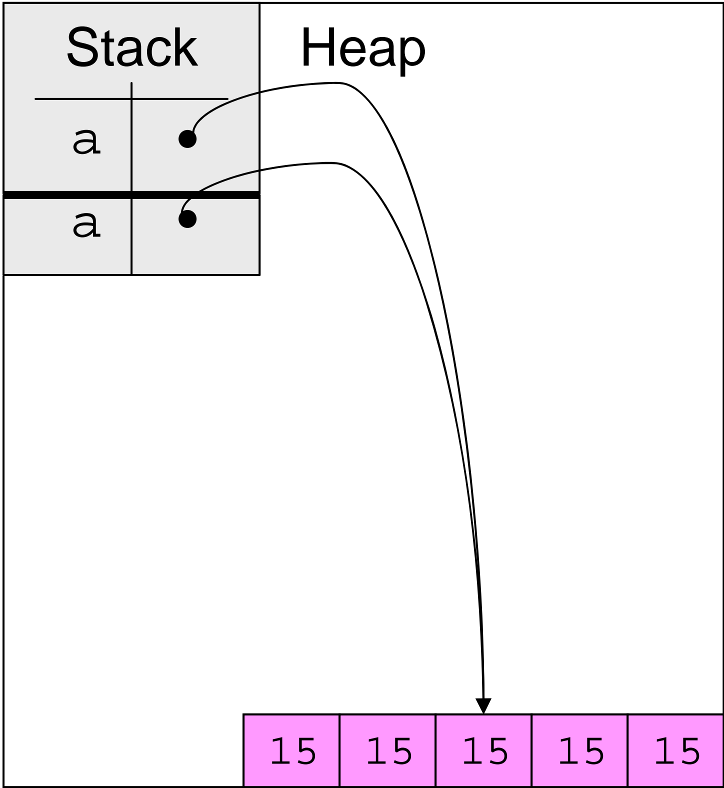
```java
public static void triple(int a) {
    a *= 3;
}

public static void triple(int[] a) {
    for (int i = 0; i < a.length; i++)
        a[i] *= 3;
}

public static void main(String[] args) {
    int[] a = new int[5];
    for (int i = 0; i < a.length; i++)
        a[i] = 5;

    triple(a[0]);
    System.out.println("After int triple, a[0] is " + a[0]);

    triple(a);
    System.out.println("After array triple, a[0] is " + a[0]);
}
```

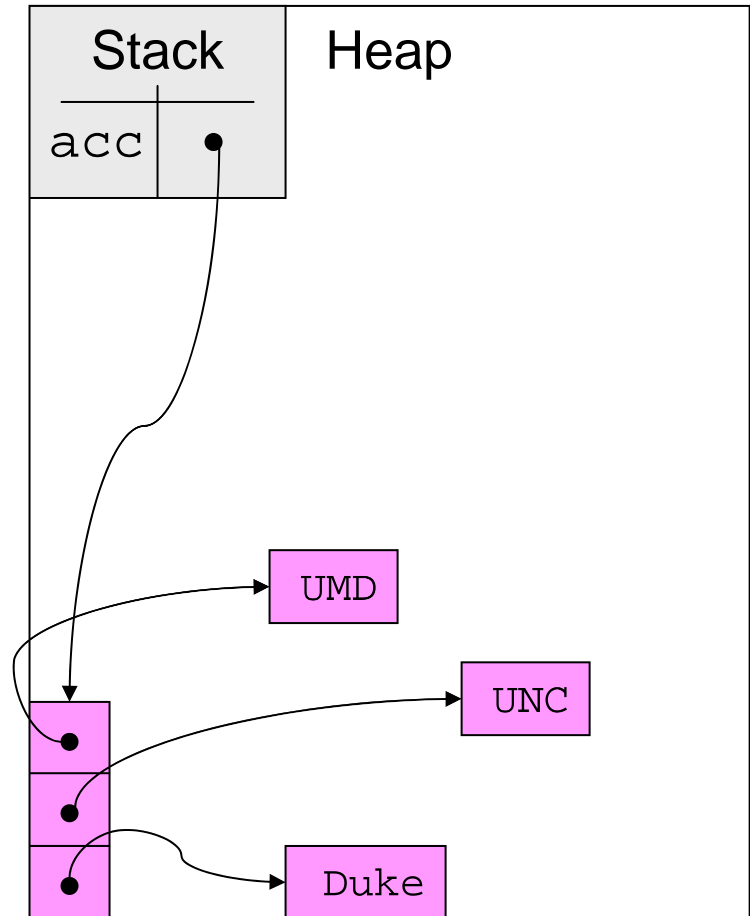


See Array Example 4

Output:
5
15

Arrays of Objects

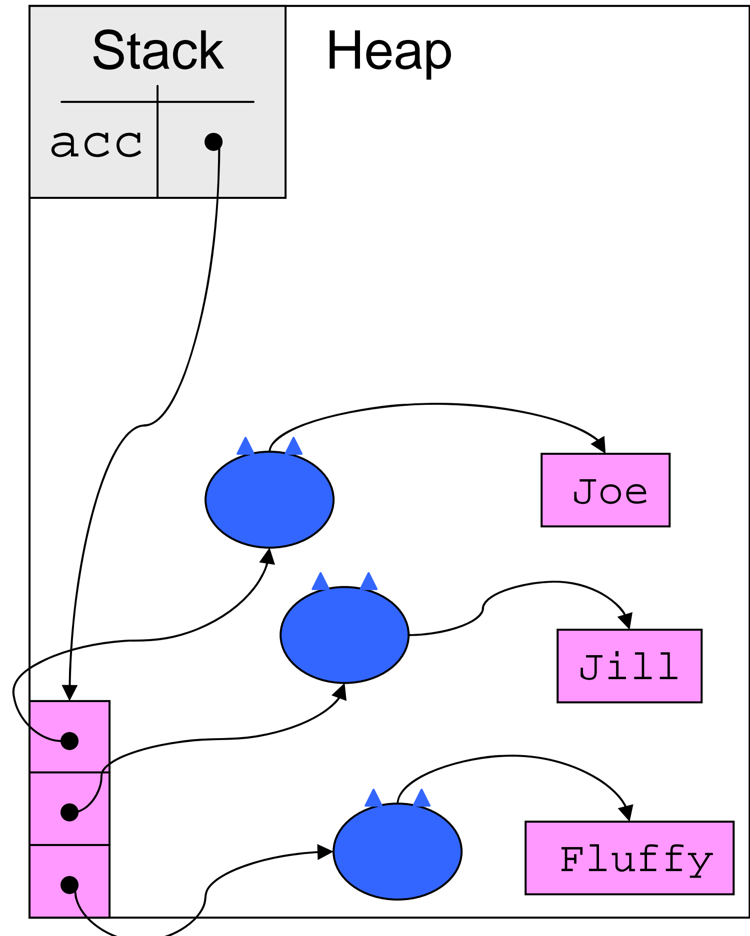
- Class types can also be base types of arrays
  - e.g.
    `String[] acc = new String[3];`
  - Array cells store references to objects
- Array initializers can also be used
  `String[] acc = {"UMD", "UNC", "Duke"};`



Arrays of Objects (continued)

- Class types can also be base types of arrays
  - e.g.
    `String[] acc = new String[3];`
  - Array cells store references to objects
- Array initializers can also be used
  `String[] acc = {"UMD", "UNC", "Duke"};`
- More complicated example than strings: Cat objects
- Expressions can also appear in initializers

```
Cat[] kennel = {
    new Cat("Joe"),
    new Cat("Jill"),
    new Cat("Fluffy")
};
```



Privacy Leaks

- Suppose we create a mutable class:

```
public class MutableThing {
    …
    public void mutateMe() {…};
}
```

- Then let's create a new class called Foo that has an instance variable of type mutableThing:

```
public class Foo {
    private MutableThing q = new MutableThing();
    …
```

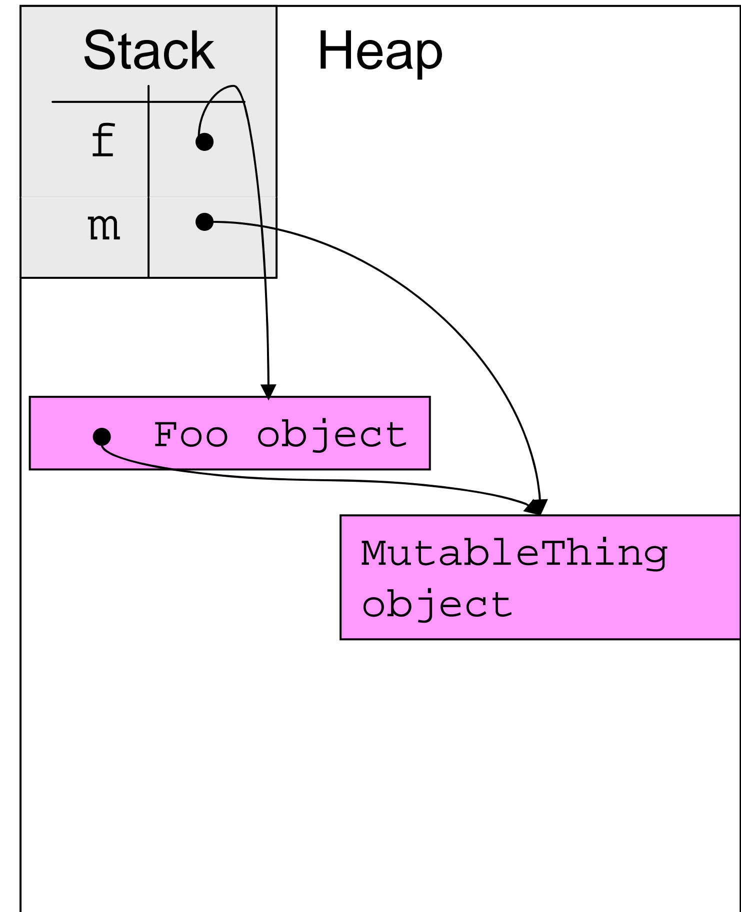
- Inside the class Foo, suppose we write an accessor for this instance variable:

```
    …
    public MutableThing getQ(){
      return q;
    }
}
```

- Instance variables are private, meaning they should only be changed using Foo methods
- Since no Foo methods involve set, Foo objects "should" be immutable
- Are they?

Privacy Leaks (continued)

- Consider following code
```
Foo f = new Foo ();
MutableThing m = f.getQ();
m.mutateMe();
```
- After this executes, what happens?
- This phenomenon is called a privacy leak
  - Private instance variables can be modified outside class
  - Behavior is due to aliasing



Fixing Privacy Leaks

- Return copies of objects referenced by instance variables
- To fix `getQ` method in `Foo`:

```
MutableThing getQ(){
    return new MutableThing(q);
}
```

  - This returns a copy of q
  - Changes made to this copy will not affect original