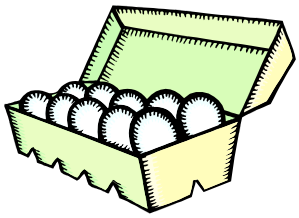
Arrays

Friday, November 9, 2007

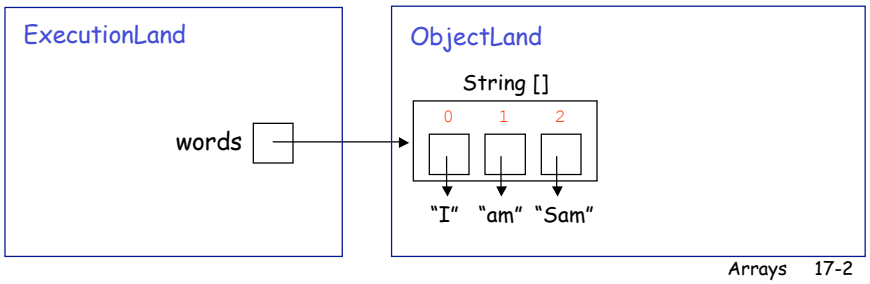


CS111 Computer Programming

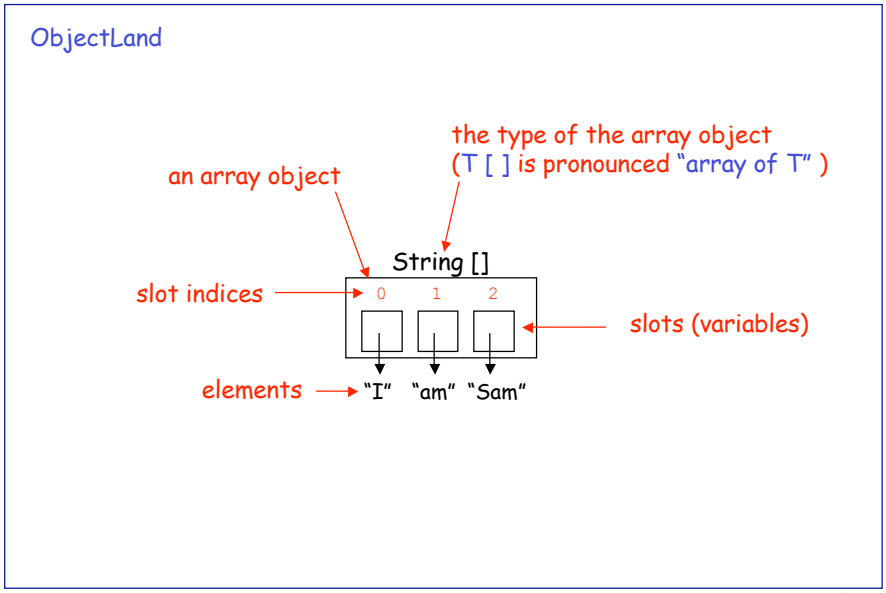
Department of Computer Science
Wellesley College

Arrays: Motivation

- Lists are great for representing a collection of values, but can only access elements in order from front to back.
- We often want an **indexed** collection where we can directly access an element at *any* index. Many languages (including Java) provide *arrays* for this purpose.
- An array is an indexed collection of variables, which we will call the **slots** of the array. The values in the slots are the **elements** of the array.

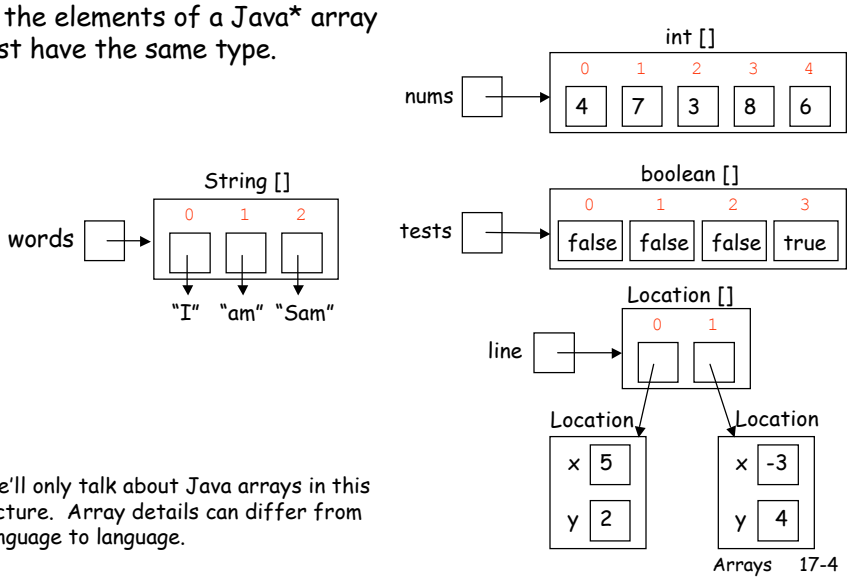


Anatomy of an Array Diagram



Arrays Are Homogeneous Collections

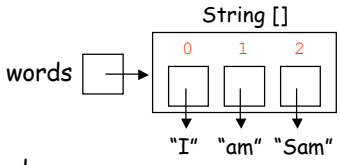
All the elements of a Java* array must have the same type.



* We'll only talk about Java arrays in this lecture. Array details can differ from language to language.

Arrays Have a Fixed Length

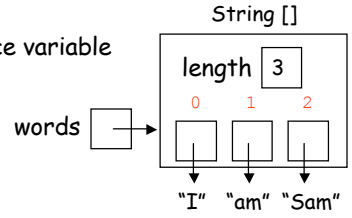
The **length** of an array is the number of slots
= the number of elements
= one more than the largest index.
E.g., the length of words is 3.



An array's length is **fixed**: once an array is created, its length cannot be changed.

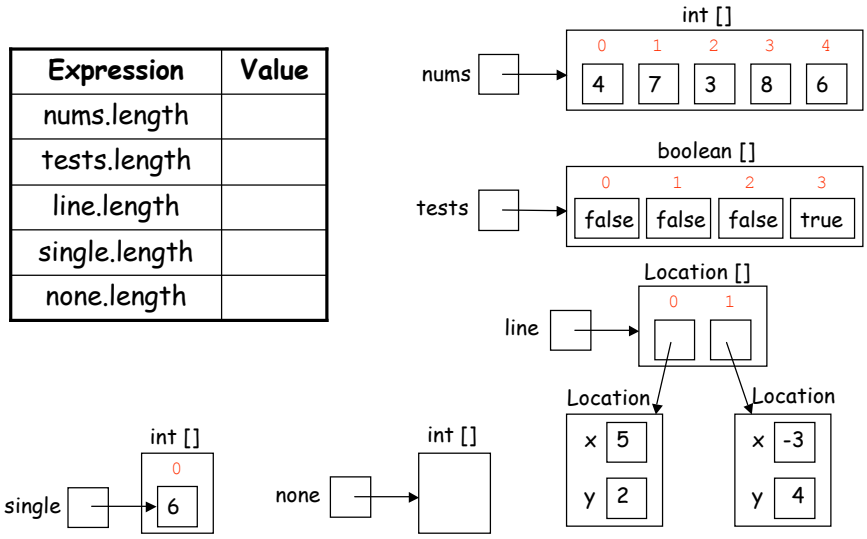
If a denotes an array, a.length stands for its length.
E.g. words.length evaluates to 3.

It's as if each array object has a length instance variable
We usually do not explicitly show this variable.
Its value cannot be changed.



Length Examples

Expression	Value
nums.length	
tests.length	
line.length	
single.length	
none.length	

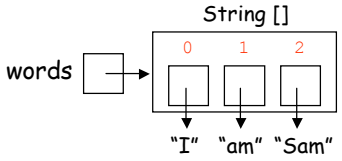


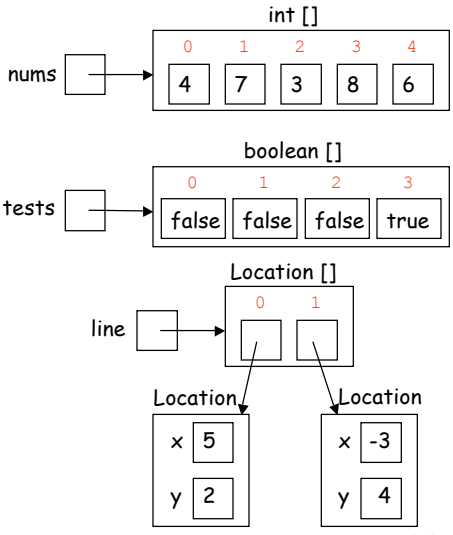
Array Indexing

If a denotes an array and i denotes an integer, the notation a[i] (pronounced "a sub i") refers to the ith slot of a.

In most contexts, a[i] evaluates to the contents of the slot:

```
words[1] // evaluates to "am"
nums[4] // evaluates to 6
tests[1+2] // evaluates to true
line[0].x // evaluates to 5
```



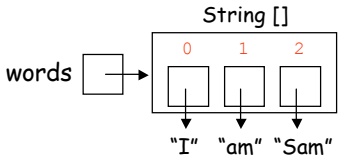


ArrayIndexOutOfBoundsException

An ArrayIndexOutOfBoundsException occurs when an invalid index is used.

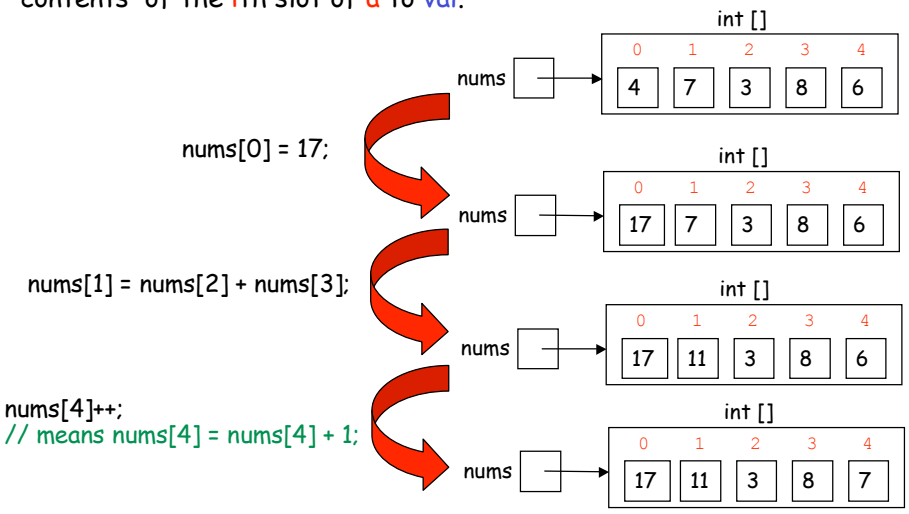
Examples:

```
words[-1]
words[4]
words[3] (gotcha!)
words[words.length] (gotcha!)
```



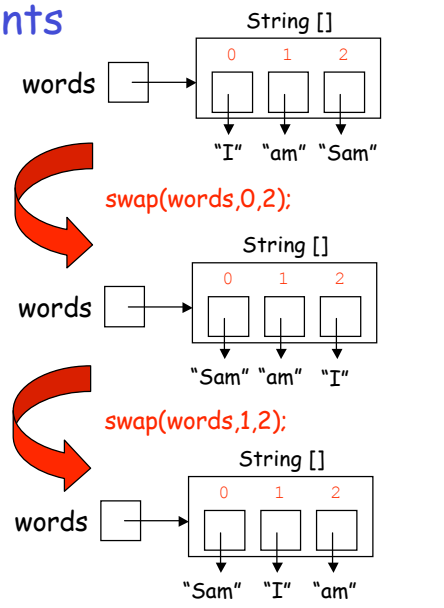
Array Slots are Mutable

The assignment a[i] = val changes the contents of the ith slot of a to val.

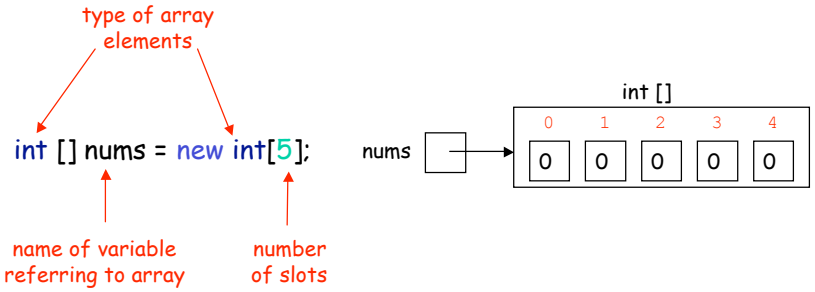


Swapping Array Elements

```
public static void swap
   (String [] strings, int i, int j) {




}
```



Array Creation



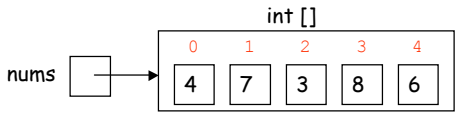
Slots are initially filled with the default value for element type:

- 0 for int
- 0.0 for double
- **false** for boolean
- **null** for any Object

Initializing an Array

Can change default slot values with assignments.

```
int [] nums= new int[5];
nums[0] = 4;
nums[1] = 7;
nums[2] = 3;
nums[3] = 8;
nums[4] = 6;
```



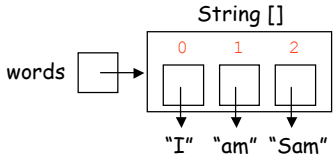
The above code can be abbreviated as:

```
int [ ] nums = new int [ ] {4,7,3,8,6};
```

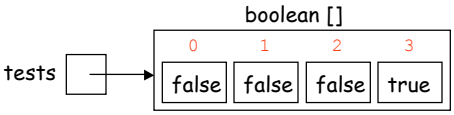
This form can be used wherever an array expression is expected.

Array Elements can be Any Type

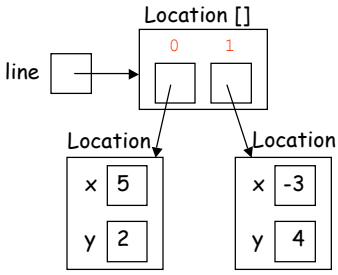
```
String [] words =
   new String [] {"I", "am", "Sam"};
```




```
boolean [] tests =
   new boolean []
    {false, false, false, true};
```




```
Location [] line =
   new Location []
    {new Location(5,2),
     new Location(-3,4)};
```



Pop Quiz! What Does This Code Print?

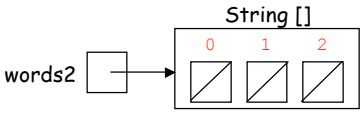
```
Buggle [] bugs = new Buggle [] {new Buggle(), new Buggle()};
bugs[0].forward(3);
bugs[1].left();
Buggle becky = new Buggle();
becky.setPosition(bugs[0].getPosition());
becky.setHeading(bugs[1].getHeading());
bugs[0] = becky;
bugs[1] = bugs[0];
bugs[0].forward(2);
System.out.println(bugs[1].getPosition());
```

Null Pointers

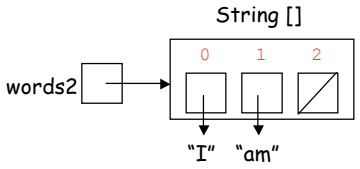
The default array value for Objects is **null** (the null pointer), which represents "no object".

```
String [] words2 = new String [3];
```

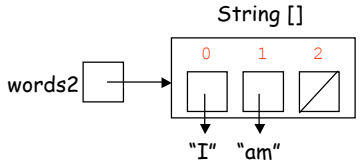
We depict variables holding the null pointer value as a box with a slash.




```
words2[0] = "I";
words2[1] = "am";
words2[0] == null // evaluates to false
words2[2] == null // evaluates to true;
```



NullPointerException



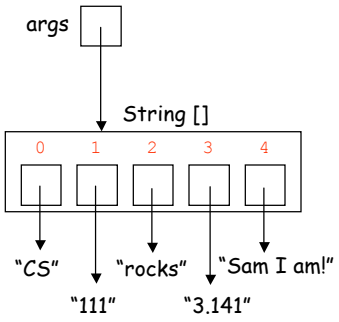
Any attempt to send a message to the null pointer value results in a NullPointerException.

```
words2[1].length() // evaluates to 2
                   //  (length of string "am", not of array)
words2[2].length() // generates a NullPointerException
```

NullPointerExceptions due to uninitialized variables are extremely common and will haunt you from now on.

main() Arguments Explained!

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




```
> java PrintArgs CS 111 rocks 3.141 "Sam I am!"
 args[0] = CS
 args[1] = 111
 args[2] = rocks
 args[3] = 3.141
 args[4] = Sam I am!
```


toString() and fromString()

The default toString() instance method for an array doesn't display its elements. Instead, it gives something unintelligible:

```
nums.toString() // evaluates to "[I@e1225d"
"nums = " + nums // evaluates to "nums = [I@e1225d"
```

The CS111 classes IntArray and StringArray provide toString() class methods that are more sensible:

```
IntArray.toString(nums) // evaluates to "{4,7,3,8,6}"
StringArray.toString(words) // evaluates to "{I,am,Sam}"
StringArray.toString(words2) // evaluates to "{I,am,null}"
```

They also provide fromString() class methods for creating arrays from strings written in this format:

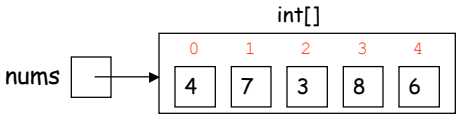
```
int [] nums = IntArray.fromString("{4,7,3,8,6}");
String [] words = StringArray.fromString("{I,am,Sam}");
```

split()

The split() instance method of the String class is useful for creating string arrays:

```
String [ ] seuss = "I do not like green eggs and ham".split(" ");
StringArray.toString(seuss)
  // evaluates to "{I,do,not,like,green,eggs,and,ham}"
```


Summing an Array



Want IntArray.sum(nums) → 28

Iteration Table

i	result
0	0
1	4
2	11
3	14
4	22
5	28

```
// Assume this is in the class IntArray
public static int sum (int [ ] a) {




}
```

Testing sum()

Can test interactively in the Dr. Java Interaction Pane:

```
> IntArray.sum(new int [] {4,7,3,8,6})
28
```

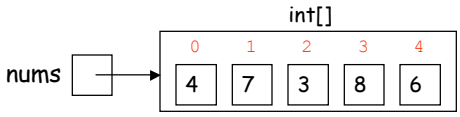
Can include hardwired test arrays in a main() method:

```
public static void main (String [] args) {
   String [ ] test = new int [ ] {4,7,3,8,6};
   System.out.println("sum(" + IntArray.toString(test)+ ") = " + sum(test));
}
```

Can read test arrays from the command line:

```
public static void main (String [] args) {
   String [ ] test = new int [args.length];
   for (int i = 0; i < args.length; i++) {
      test[i] = Integer.parseInt(args[i]);
   }
   System.out.println("sum(" + IntArray.toString(test) + ") = " + sum(test));
}
```


Summing from Top Down



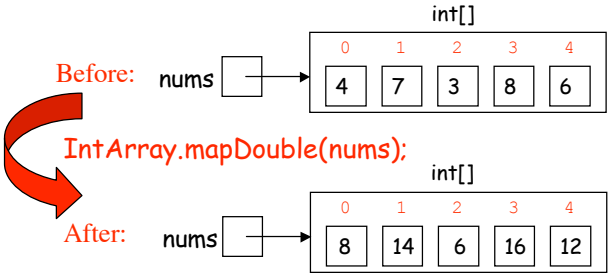
Want IntArray.sum(nums) → 28

Iteration Table

i	result
4	0
3	6
2	14
1	17
0	24
-1	28

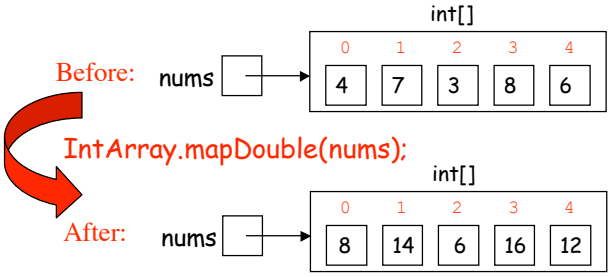
```
// Assume this is in the class IntArray
public static int sum (int [ ] a) {




}
```

Doubling the Elements of an Array



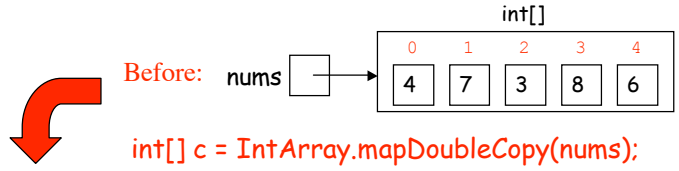

```
// Assume this is in the class IntArray
public static void mapDouble (int [] a) {




}
```


Destructive Mapping Idiom

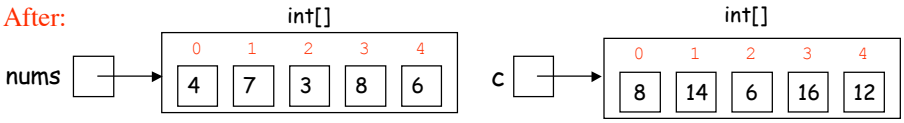


- mapDouble() illustrates the **mapping idiom** for arrays.
- It is **destructive** in the sense that it replaces the original values in the array slots.

Nondestructive (Copying) Mapper

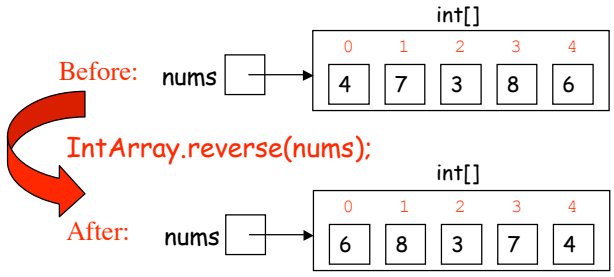



```
int[] c = IntArray.mapDoubleCopy(nums);
```




```
// Assume this is in the class IntArray
public static int[] mapDoubleCopy (int [] a) {




}
```


Destructive Array Reversal



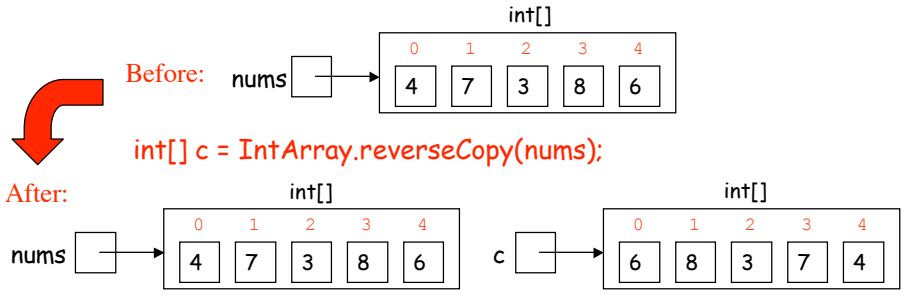

```
// Assume this is in the class IntArray
public static void reverse (int [] a) {




}
```

A Broken Array Reversal

The following destructive reverse method attempts to reverse an array using swap(). Unfortunately, it has a bug. Please fix the method so that it works.

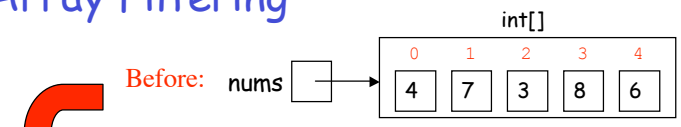
```
// Assume this is in the class IntArray
public static void reverse (int [] a) {
   for (int i = 0; i < a.length; i++) {
      IntArray.swap(a, i, a.length – 1 - i);
   }
}
```


Nondestructive Array Reversal

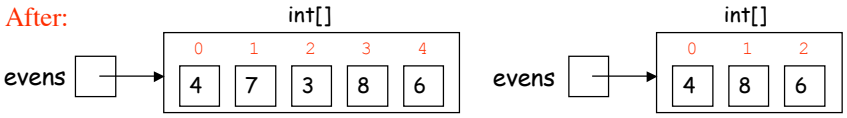



```
// Assume this is in the class IntArray
public static int[] reverseCopy (int [] a) {




}
```

Array Filtering



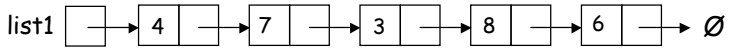
int[] evens = IntArray.filterEvensCopy(nums);



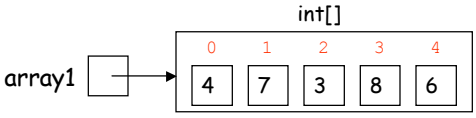

```
// Assume this is in the class IntArray
public static int[] filterEvensCopy (int [] a) {




}
```


Accessing Elements: Lists vs. Arrays

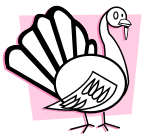
Accessing the ith element of a list takes time proportional to i.



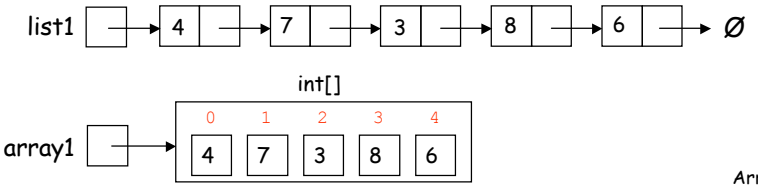
Accessing the ith element of an array can be done directly.
The time it takes is the same for any i.



Good news & bad news



	Lists	Arrays
Find first element		
Find last element		
Add element to front		
Add element to end		
Find length		
Mutable		
Work for diff types		



Java Arrays: Summary of Properties

- Arrays are **fixed-length** indexed collections of elements. Once created, the size of an array cannot change.
- Elements are **0-indexed** (i.e., indices start at 0, not 1).
- Every element can be **accessed cheaply**, independent of index.
- Arrays are **homogeneous** collections: all elements must have the same type.
- There is a **special concise syntax** for creating, accessing and updating arrays.
- **Dynamic (run-time) index checking** ensures that indices of all array operations are valid.
- Arrays are objects that are **not instances of any class**.
- **Arrays** are the answer to the following analogy question:

  Recursion : lists :: iteration : ???