Arrays and ArrayLists

Ananda Gunawardena

Introduction

- Array is a useful and powerful aggregate data structure presence in modern programming languages
- Arrays allow us to store arbitrary sized sequences of primitive values or sequences of references to objects
- Arrays allow easy access and manipulation to the values/objects that they store
- Arrays are indexed by a sequence of integers
- classes can use arrays as instance variables to store databases of value/references

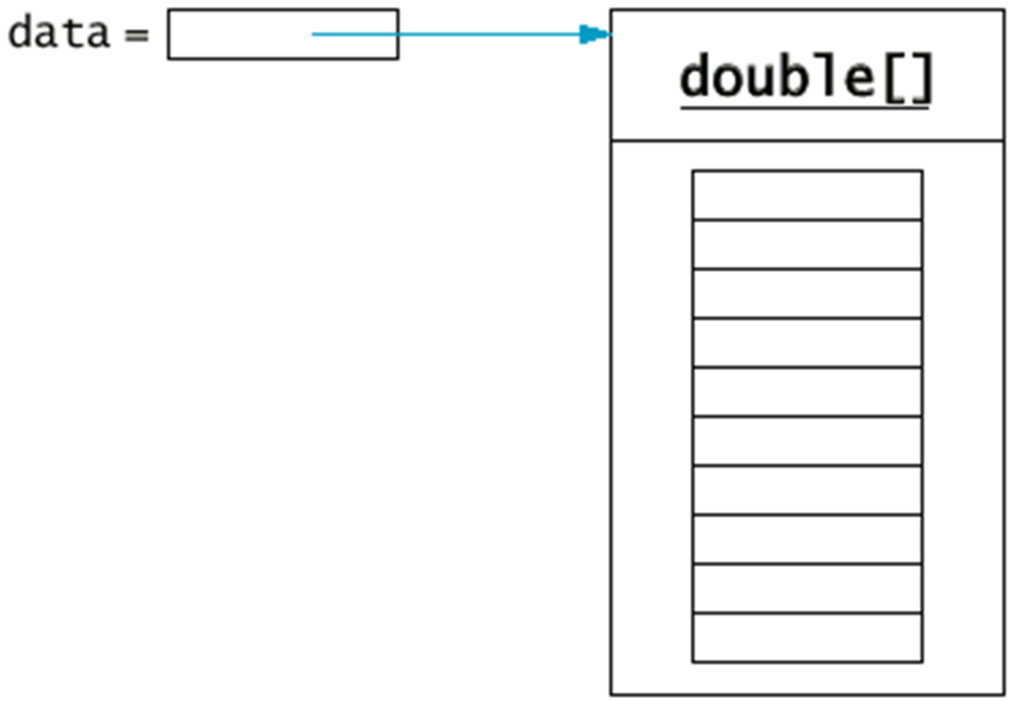
Arrays

- new is used to construct a new array:

```
new double[10]
```

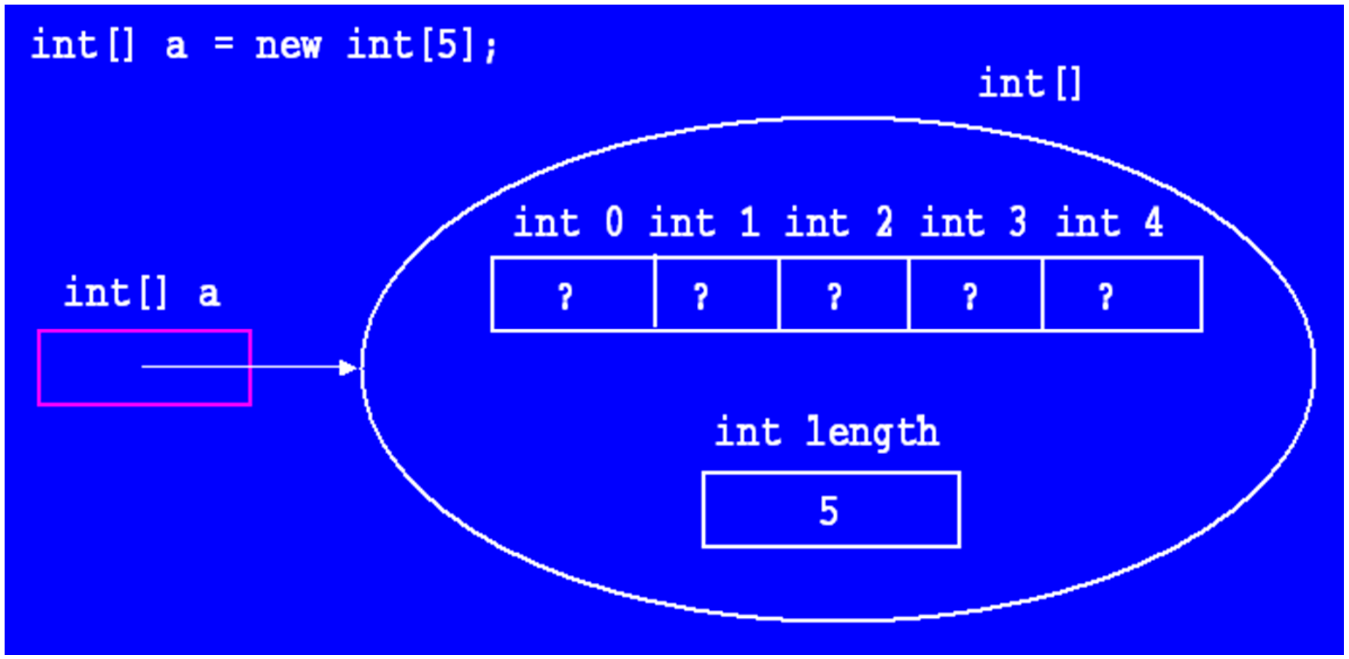
- Store 10 double type variables in an array of doubles

```
double[] data = new double[10];
```



integer Arrays

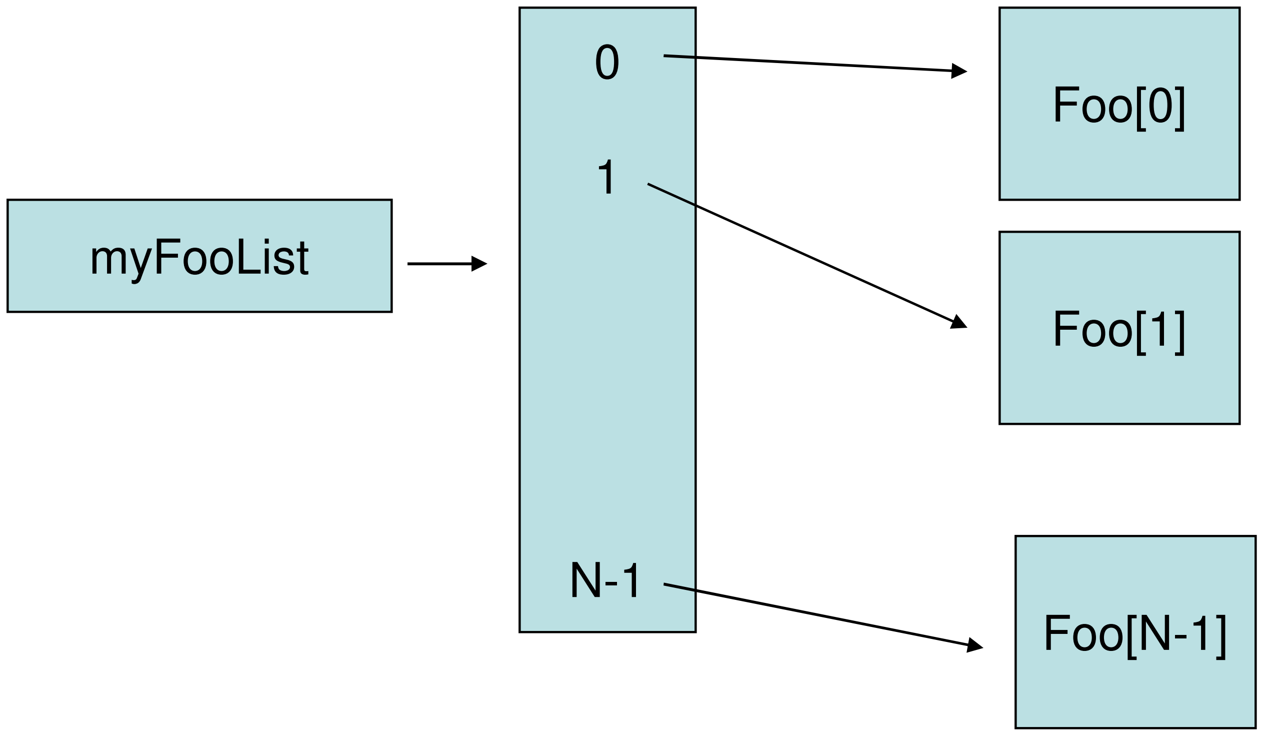
int[] A = new int[5];



Array of Object References

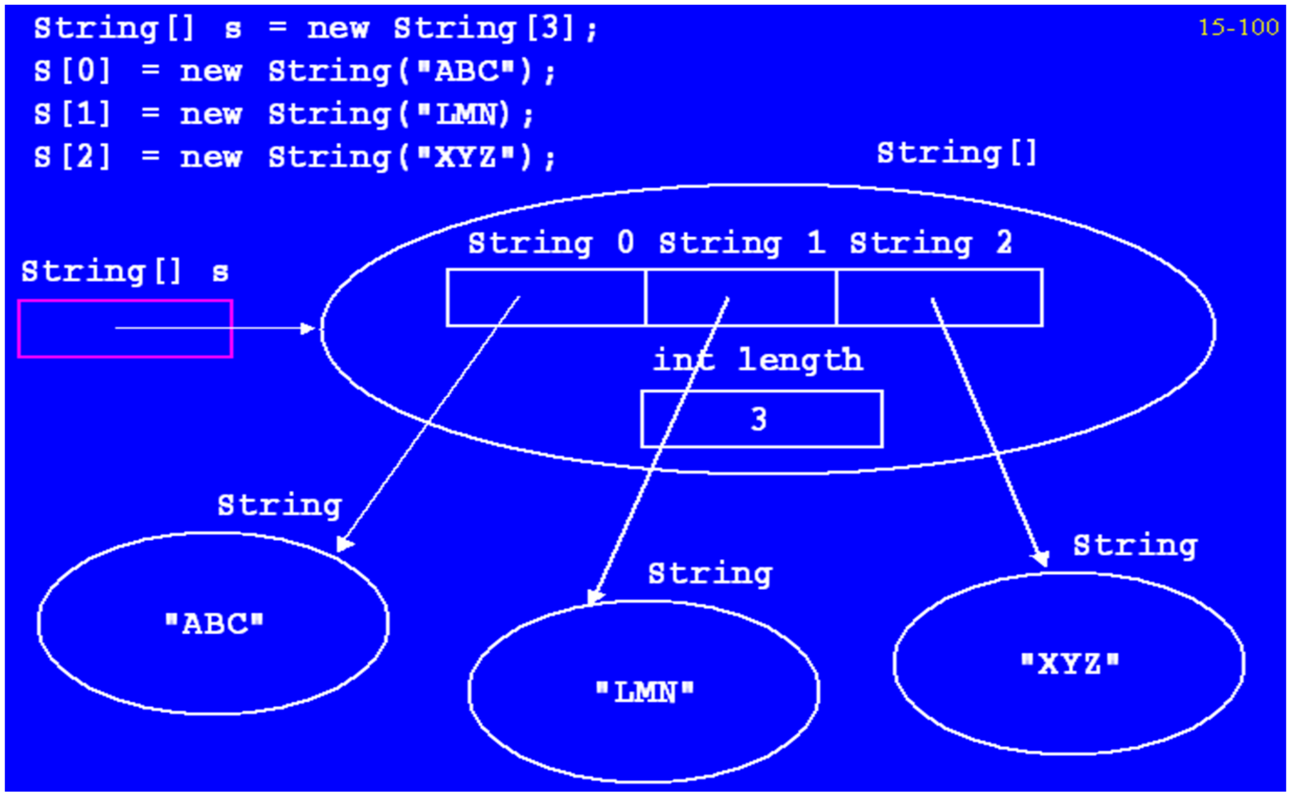
class foo() { ….}

foo[ ] myFooList = new foo[N];



Array of Strings

- **An array of Strings**
  - **String[] s = new String[]{"ABC", "LMN", "XYZ"};**



```
String[] s = new String[3];
s[0] = new String("ABC");
s[1] = new String("LMN);
s[2] = new String("XYZ");
```

Array of Bytes

- We can create array of bytes and perhaps return them from a method
- public byte[ ] foo(){

```
      byte[]  temp = new byte[10];
      for (int i=0;i<10;i++)
         temp[i] = new Byte(i);
       return temp;
}
```

Arrays

- Arrays have *fixed length*
- Arrays have element of specific type or references to `Objects`
- Operator [ ] is used to access array elements
  `data[4] = 29.95;`
- Use length attribute to get array length.
  - `data.length`. (Not a method!)

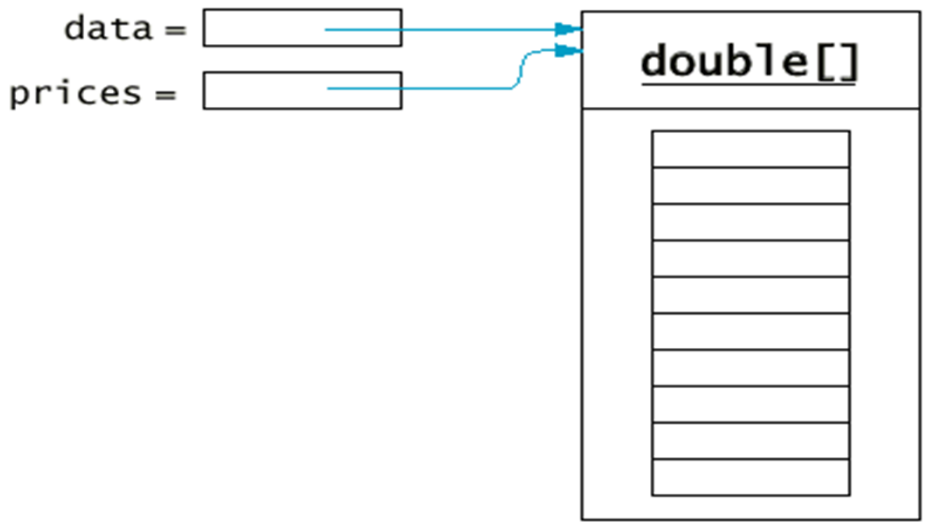
Array

- is a homogeneous data structure: each of its members stores the same type (either primitive or reference)
- the indices go from 0 to one less than the length of the array
- each array object stores a **public final int length** instance variable that stores the length of the array
- we can access the value stored in this field, in the example above, by writing **a.length**

Copying Arrays

Copying an array reference yields a second reference to the same array

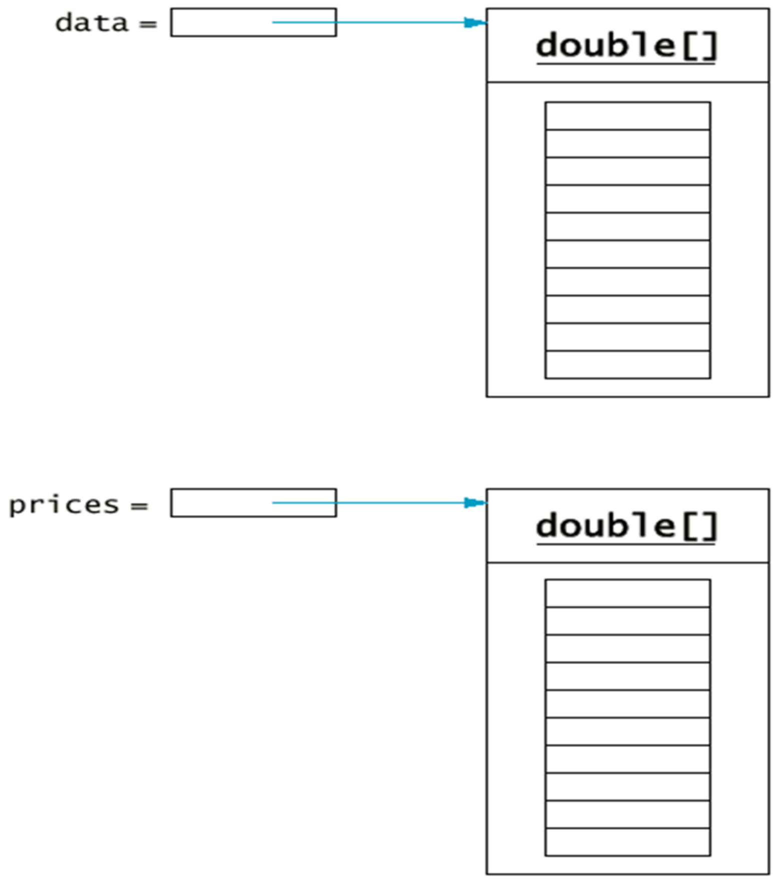
```
    double[] data = new double[10];
      // fill array . . .
      double[] prices = data;
```



Cloning Arrays

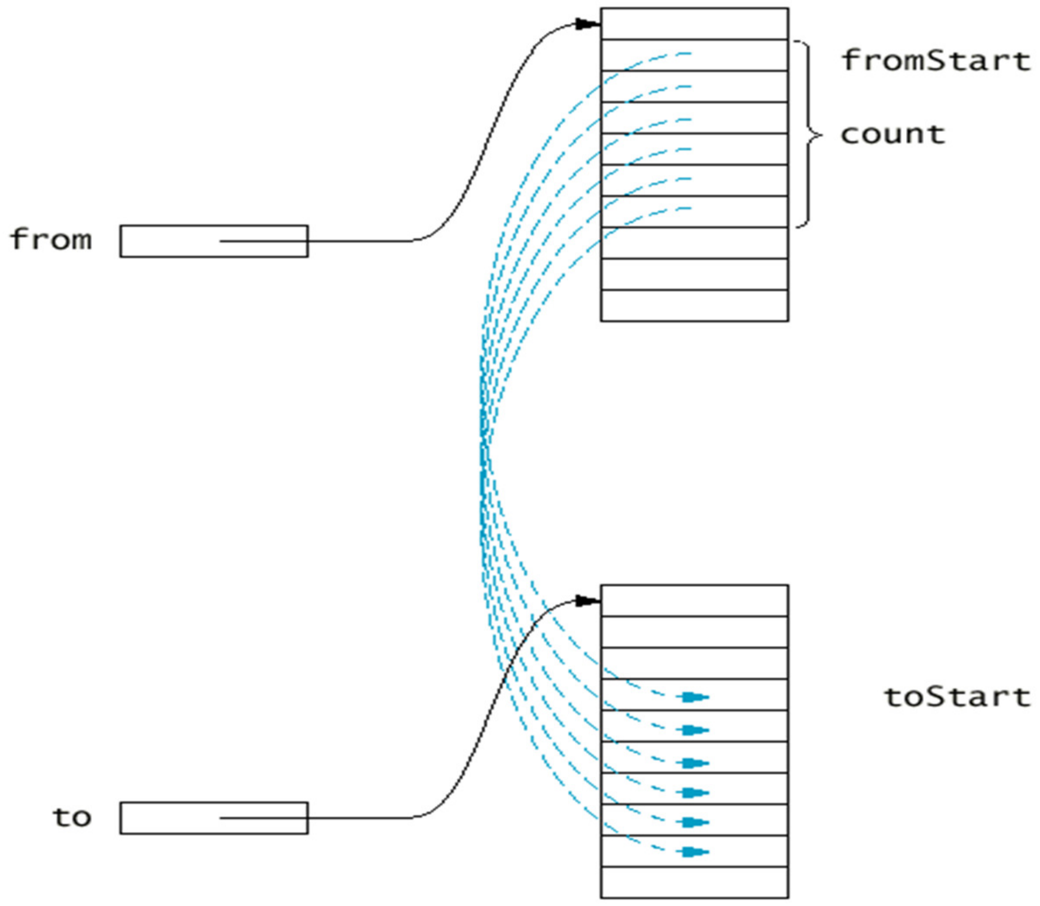
- **Use `clone` to make true copy**

```
double[] prices = (double[])data.clone();
```



Copying Array Elements

- System.arraycopy(from, fromStart, to, toStart, count);



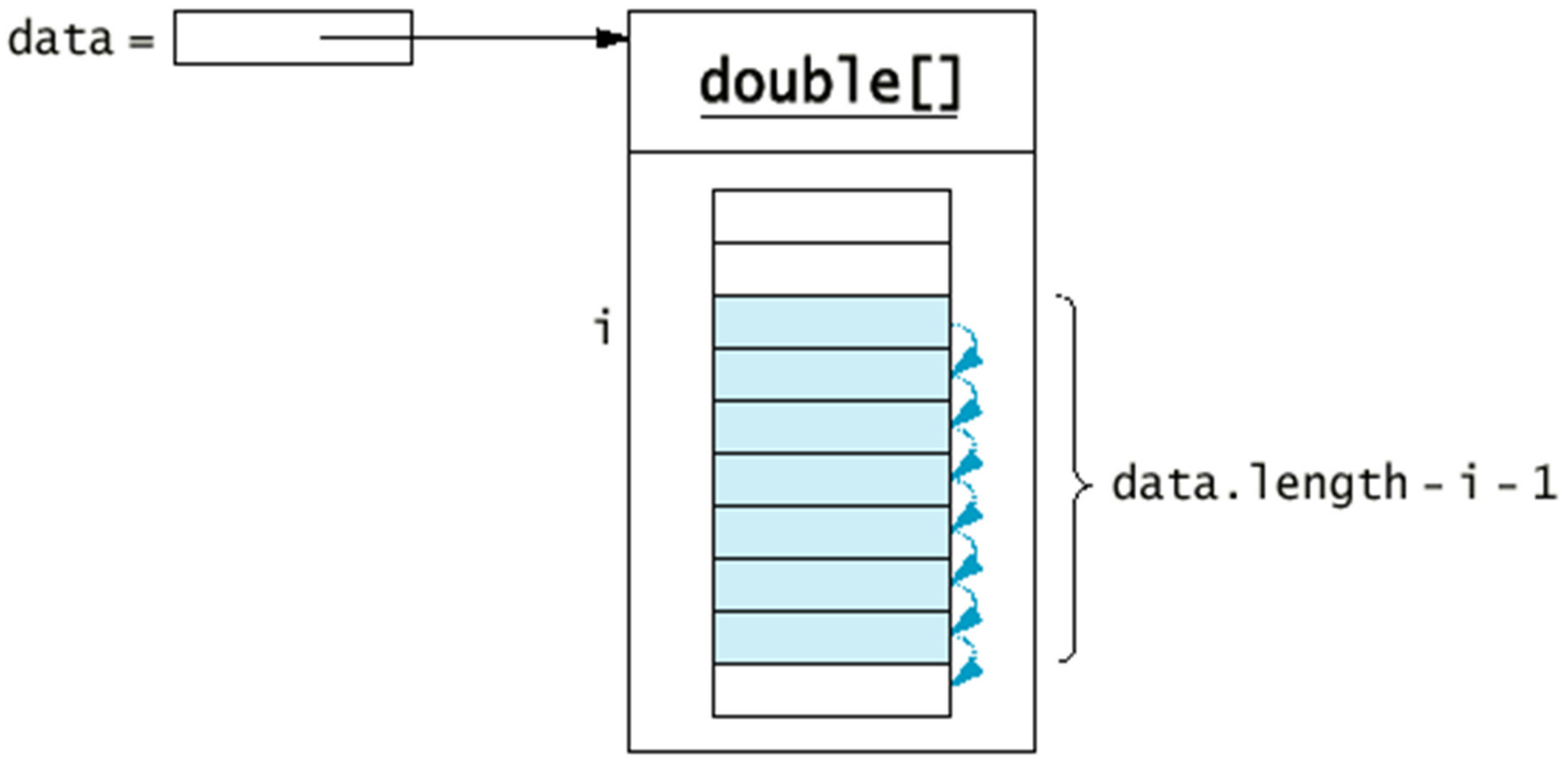
Question

- Write your own version of arraycopy for int arrays

```
public static void arraycopy(int[] from,int fromstart, int[] to, int tostart, int count) {












}
```

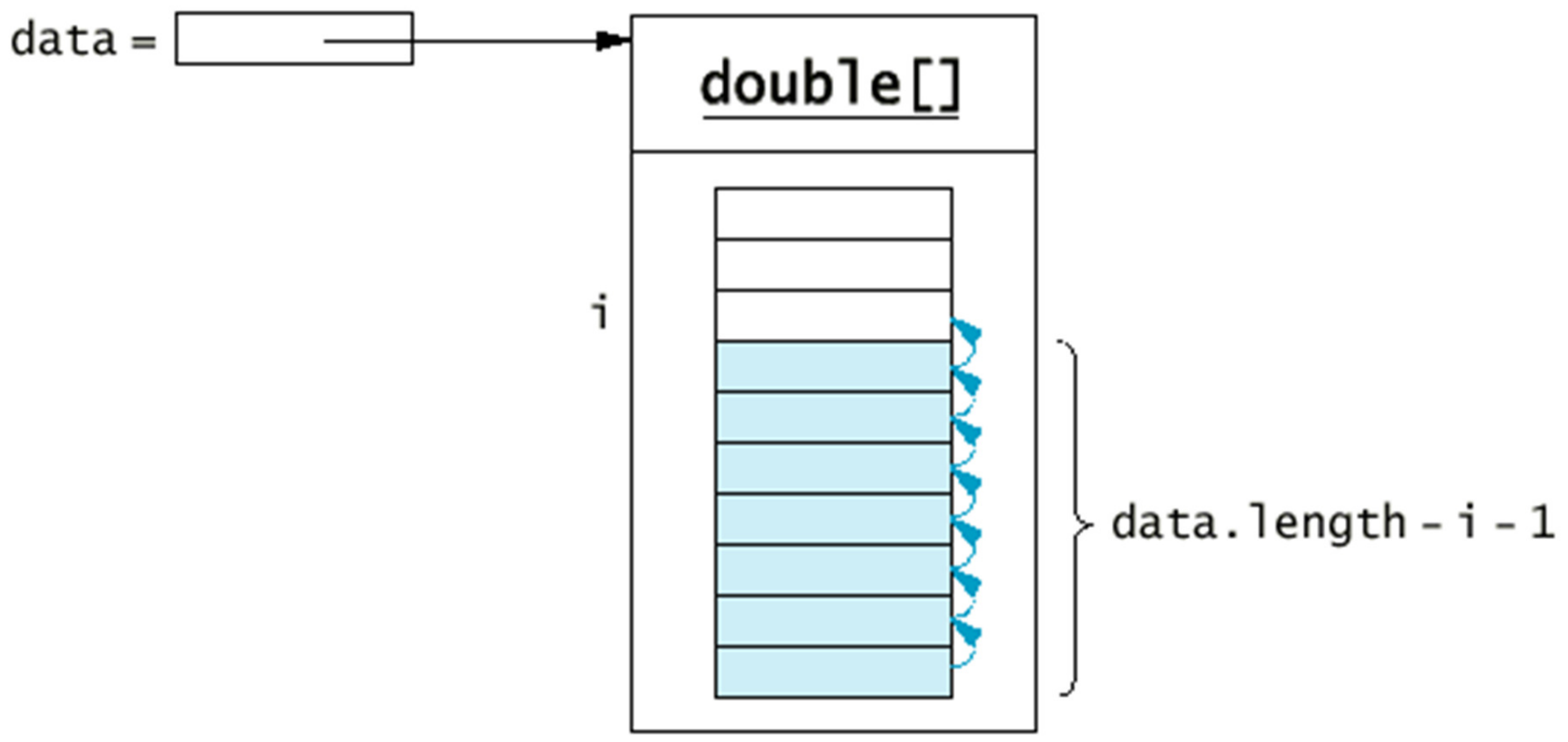
Shifting and Copying

- `System.arraycopy(data, i, data, i + 1, data.length - i - 1);`
  `data[i] = x;`



More Shifting

```
System.arraycopy(data, i + 1, data, i, data.length - i - 1);
```



Shifting Elements

- Shift all elements to Right by 1 starting at index i

- Shift all elements left by 1 starting at index i (i>0)

Swapping Array Elements

- Suppose you want to swap two elements in the array, say entries with indices i and j. Assuming we are dealing with an array of ints
  - int temp = A[i]; // save a copy of A[i] in temp
  - A [i] = A[j];   // copy the content of A[j] to A[i]
  - A[j] = temp;   // copy the content of temp to A[j]
- Note that : A[i]= A[j] and A[j] = A[i] do not swap content
- Exercise: Reverse an array using swaps

Accessing Arrays

- int[] a = new int[]{4, 2, 0, 1, 3};
- system.out.println(a[0]);
- if (a[5] == 0) ...some statement
- if the value computed for the index is less than 0, or greater than OR EQUAL TO the length of the array
  - trying to access the member at an illegal index causes Java to throw the
  - ArrayIndexOutOfBoundsException which contains a message showing what index was attempted to be accessed

Partially Filled Arrays

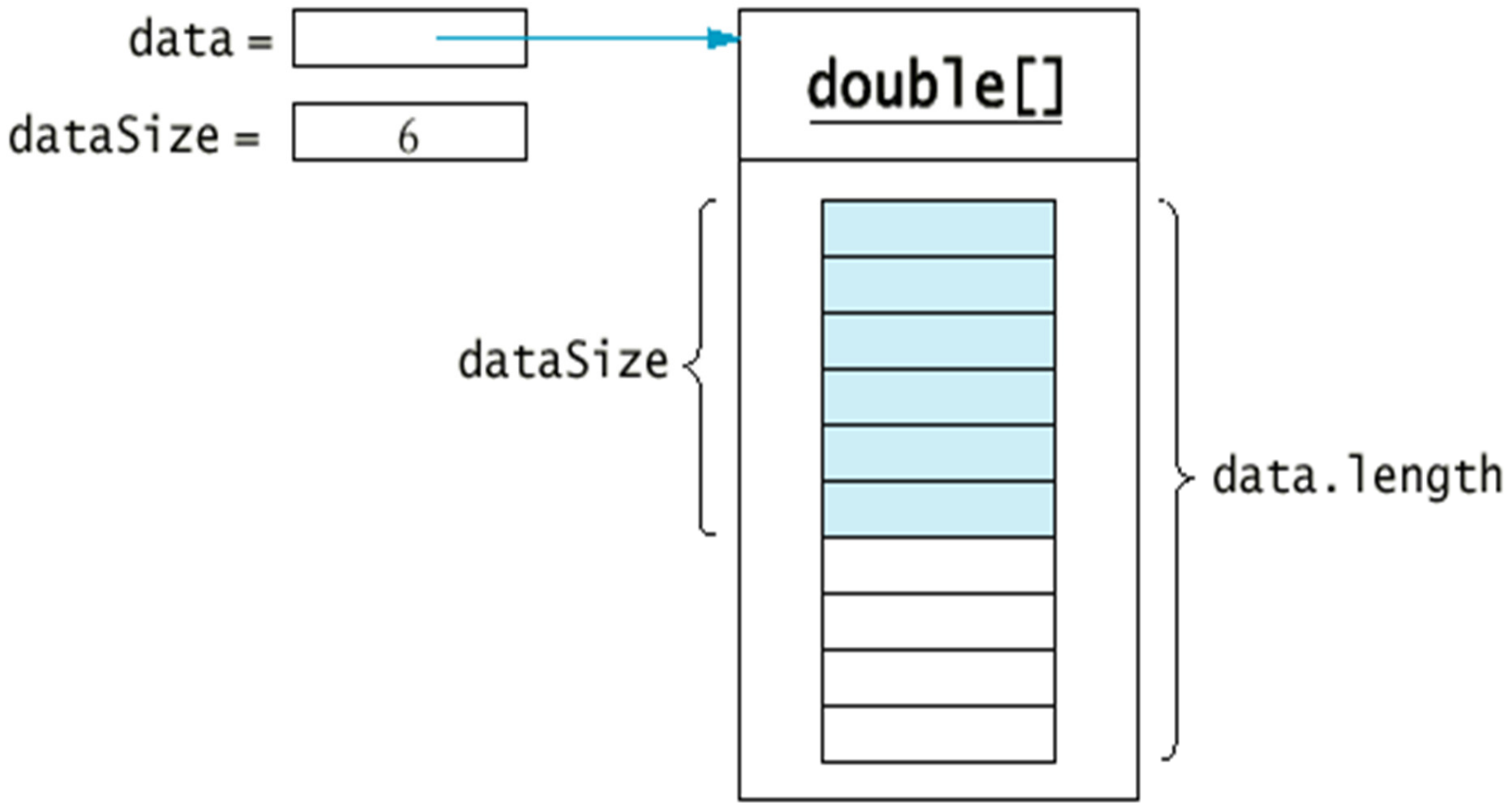
- Array.length = maximum capacity of the array
- Usually, array is partially filled
- Need companion variable to keep track of current size

```
final int capacity = 100;
double[] data = new double[capacity];
int size = 0;
```

- Update *size* as array is filled:

```
data[size] = x;
size++;
```

Partially Filled Arrays



Partially Filled Arrays

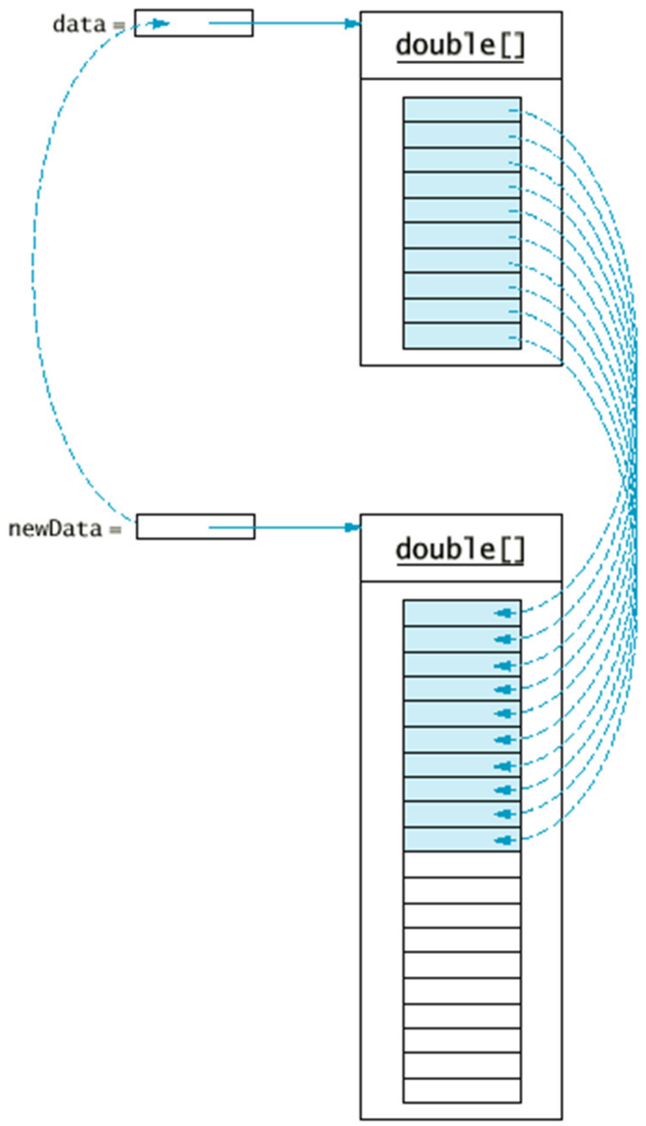
- Remember to stop at `dataSize-1` when looking at array elements:

```
for (int i = 0; i < dataSize; i++)
   sum = sum + data[i];
```

- Be careful not to overfill the array

```
if (dataSize >= data.length)
   System.out.println("Sorry--array full");
```

Resizing an Array



Dynamic Arrays

- Arrays are typically static structures
- However we can design a new array class that is dynamic (that is, you never run out of space)
- Java already has a dynamic array class called ArrayList
- See Java API for arrayList class
  - http://java.sun.com/j2se/1.5/docs/api/java/util/ArrayList.html

Dynamic Array Class

```
public class myArrayList {
    int capacity;
    ……
    public void add(Object O){





    }
…
}
```

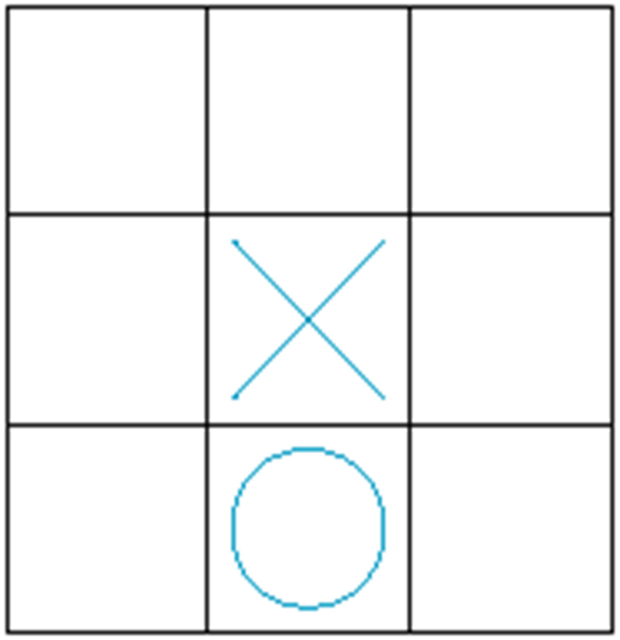
Multidimensional Arrays

Dimensions

- Some application solutions require tables with multiple dimensions
  - Modeling a matrix require a 2-dimensional array or table
  - Modeling an application that require 3-dimensional array
    - Example: in Graphics, representing a point (x, y, z)

Two-Dimensional Arrays

- Matrix with rows and columns
- Example: Tic Tac Toe board



```
char[][] board = new char[3][3];
board[i][j] = 'x';
```

Matrix Access

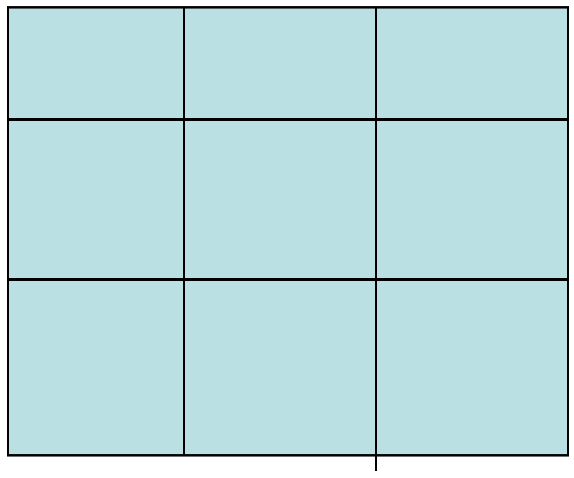
- A matrix or 2D array require access using two indices.

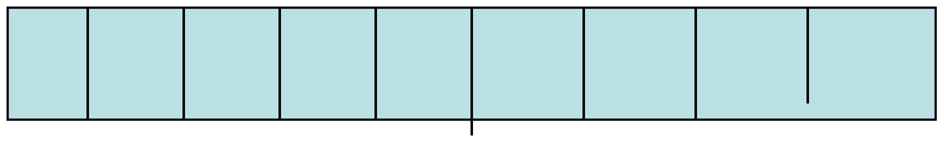
Example:
```
for (int i=0; i<N; i++)
    for (int j=0; j<N; j++)
        A[i][j] = 0;
```

- Write a method that converts a matrix A to a matrix B, where B[i][j] = A[j][i]

Memory Allocation

- Java (and many other language compilers) allocate memory for 2D arrays as an array of 1D arrays





ArrayLists

java.util

Class ArrayList<E>

java.lang.Object

  java.util.AbstractCollection<E>

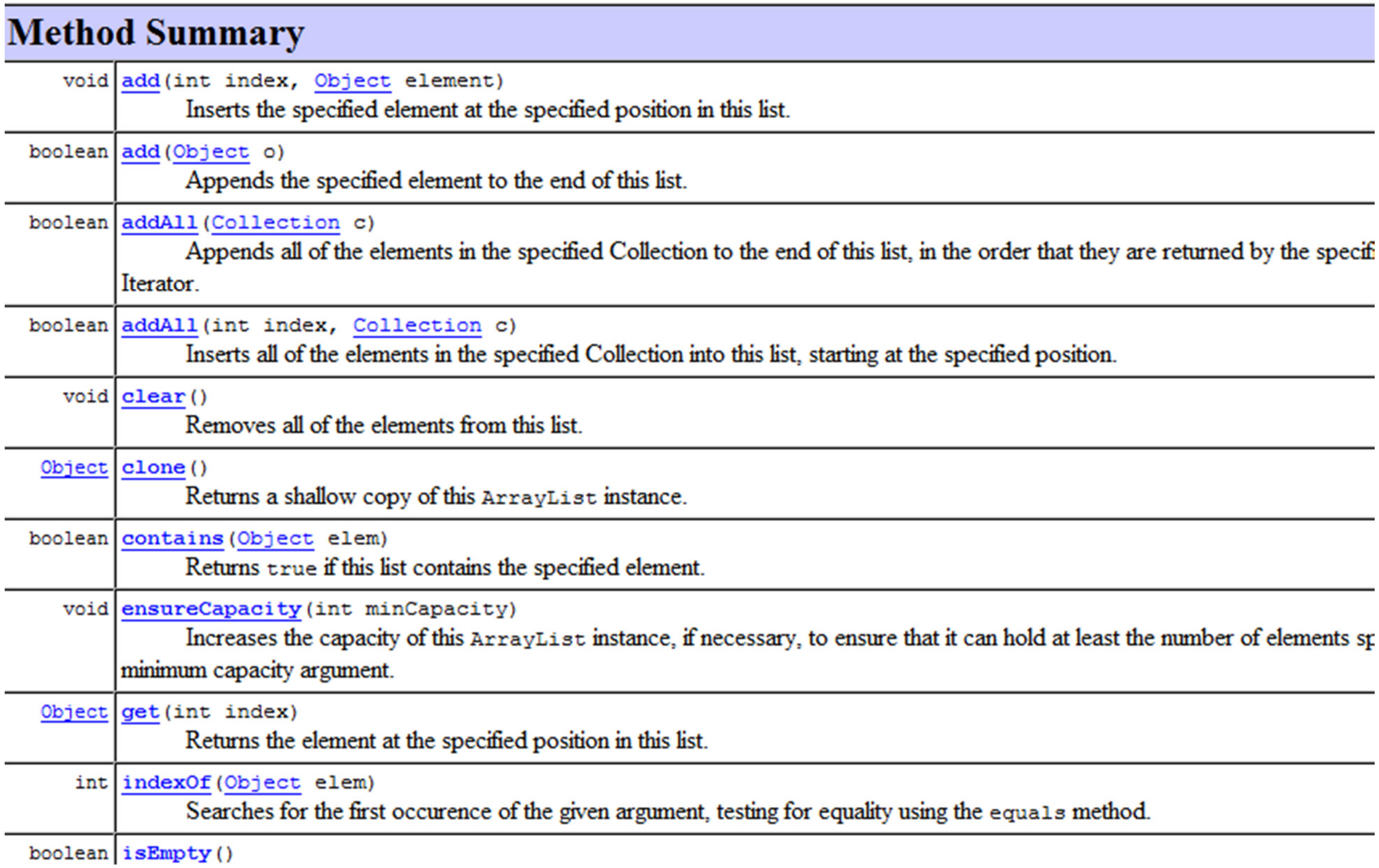
      java.util.AbstractList<E>

          java.util.ArrayList<E>

ArrayList

- Resizable-array implementation of the List interface. Implements all optional list operations, and permits all elements, including null. In addition to implementing the List interface, this class provides methods to manipulate the size of the array that is used internally to store the list.
- Each ArrayList instance has a *capacity*. The capacity is the size of the array used to store the elements in the list. It is always at least as large as the list size. As elements are added to an ArrayList, its capacity grows automatically. The details of the growth policy are not specified beyond the fact that adding an element has constant amortized time cost.
- An application can increase the capacity of an ArrayList instance before adding a large number of elements using the ensureCapacity operation. This may reduce the amount of incremental reallocation.

ARRAYLIST METHODS



Method Summary	
void	add(int index, Object element) Inserts the specified element at the specified position in this list.
boolean	add(Object o) Appends the specified element to the end of this list.
boolean	addAll(Collection c) Appends all of the elements in the specified Collection to the end of this list, in the order that they are returned by the specif Iterator.
boolean	addAll(int index, Collection c) Inserts all of the elements in the specified Collection into this list, starting at the specified position.
void	clear() Removes all of the elements from this list.
Object	clone() Returns a shallow copy of this `ArrayList` instance.
boolean	contains(Object elem) Returns `true` if this list contains the specified element.
void	ensureCapacity(int minCapacity) Increases the capacity of this `ArrayList` instance, if necessary, to ensure that it can hold at least the number of elements sp minimum capacity argument.
Object	get(int index) Returns the element at the specified position in this list.
int	indexOf(Object elem) Searches for the first occurence of the given argument, testing for equality using the `equals` method.
boolean	isEmpty()

See API for More

Summary

- Arrays are homogeneous linear structures with direct access to content
- Arrays are simple and easy to use in applications
- Most applications require use of an array
  - Sorting, searching