Mahidol University
Faculty of Information
and Communication Technology

LECTURE 04
Array and ArrayList

ITCS123 Object Oriented Programming

Dr. Siripen Pongpaichet
Dr. Petch Sajjacholapunt
Asst. Prof. Dr. Ananta Srisuphab

Semester 2/2023 14 January 2024

Recap – Lecture 03

- Object & Class
- Implementing Class
  - Instance variables (or instance fields or attributes)
  - Different Kinds of Methods in a class
    - Constructor
    - Accessor Method (e.g., getter)
    - Mutator Method (e.g., setter)
    - Instane Methods
    - Static Methods
- Encapsulation and scope of variables
  - Access specifiers: public, protected, default, private

You can check your lab assignment score on MyCourses by selecting “Grade” menu

Note that each lab has 2 points

In the past week, how many **hours** you spent on coding outside the classroom?

Outcomes of this lecture

- Can explain the different between **Array** vs **ArrayList**
- Can demonstrate how to **construct** and **initialize** both Array and ArrayList
- Can demonstrate how to store **primitive data type** variables and **reference data type** variables in Array and ArrayList
- Can implement a program to **access** and **change** values or elements in both Array and ArrayList
- Can implement a program to **pass** Array as arguments to methods
- Can use Array, ArrayList, and 2 dimensional arrays (**2D Arrays**) to implement **basic algorithms**

1. Array

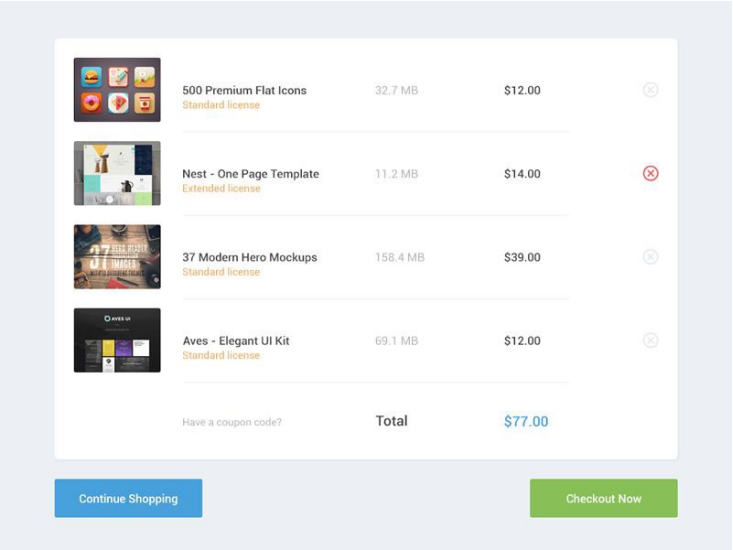
- Introduction to Array
- Initializing Array of Primitive Type
- Initializing Array of Objects
- Accessing Array of Primitive Type and Objects
- Passing Array as Arguments to Methods

Recall: Primitives vs. objects

- In java, a variable can be either
  - **Primitive**
    - Has well-defined set of possible values
    - Primitive types are lowercase and shown as `keyword` in Eclipse.
    - E.g. int, double, char, boolean
  - **Reference**
    - Similar to C/C++, You can think of it as a pointer to an object.
    - An **object** is a class instance or an **array**.
    - Yes, an array is an object!

Introduction to Array

- Array: a container of values/objects of the same data type
  - e.g., shopping cart contains value of items, waiting queue contains value of person, etc.
- After creation its length is fixed
- Variable declared as an array is object reference



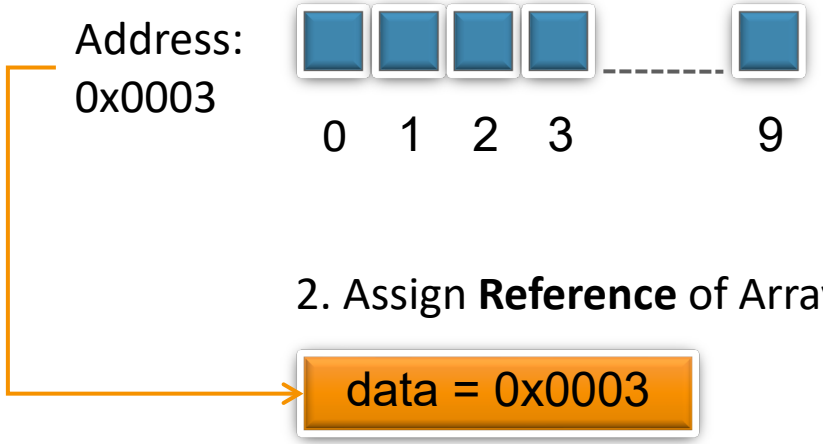
1.1 Syntax – Construct & Declaration

```
// =========== Syntax ===========
// 1. Construction
new typeName[length];    // empty values
{ v1, v2, v3, v4 };      // with values

// 2. Variable declaration
typeName[] variableName;

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

char[] grade = {'A’, 'B', 'C', 'D’};
```

Array behave like an object.

The array variable stores the **address** pointing to the array.

1. Construct Array of length '10'



2. Assign **Reference** of Array to `data` variable

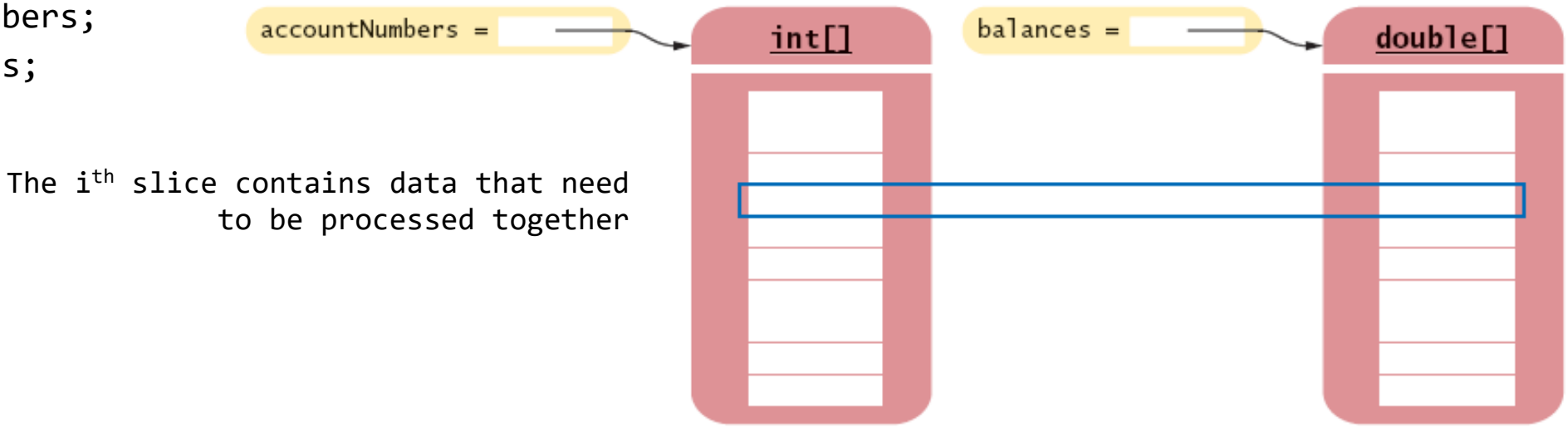
1.2 Avoid Parallel Arrays

Parallel arrays are a set of arrays that contain data that need to be process together at the same element.

This is **NOT** the OOP ways.

e.g., parallel arrays of bank account information

```
int[] accountNumbers;
double[] balances;
```

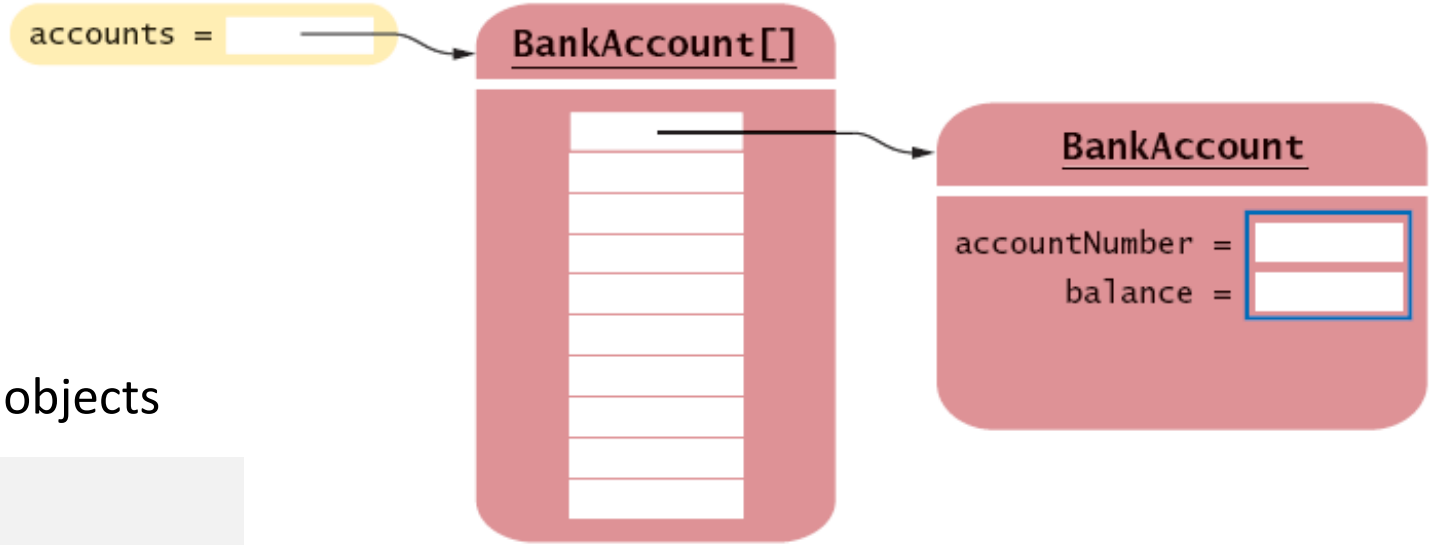


Avoid Parallel Arrays

Make Parallel Arrays into Arrays of Objects

First, Create BankAccount Class containing needed information such as *accountNumber* and *balance*,

```
public class BankAccount {
    private int accountNumber;
    private double balance;
    // . . .
}
```



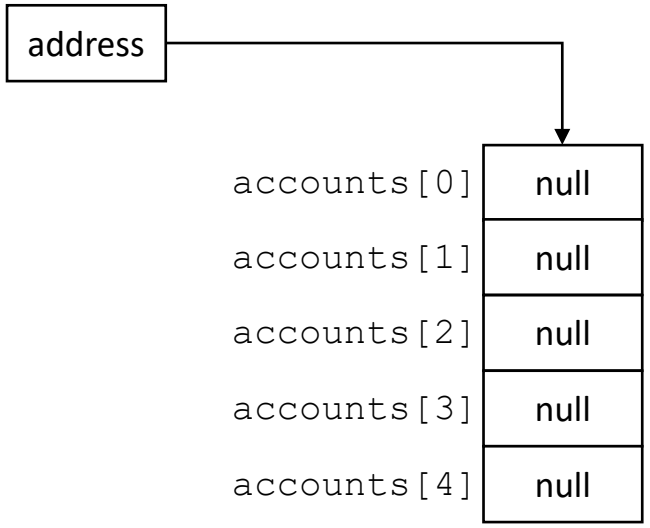
and then create an array of BankAccount objects

```
// accounts array

BankAccount[] accounts = new BankAccount[10];
```

1.3 Initializing Array of Objects

```
BankAccount[] accounts = new BankAccount[5];
```

The **accounts** variable holds the address of an **BankAccount** array.



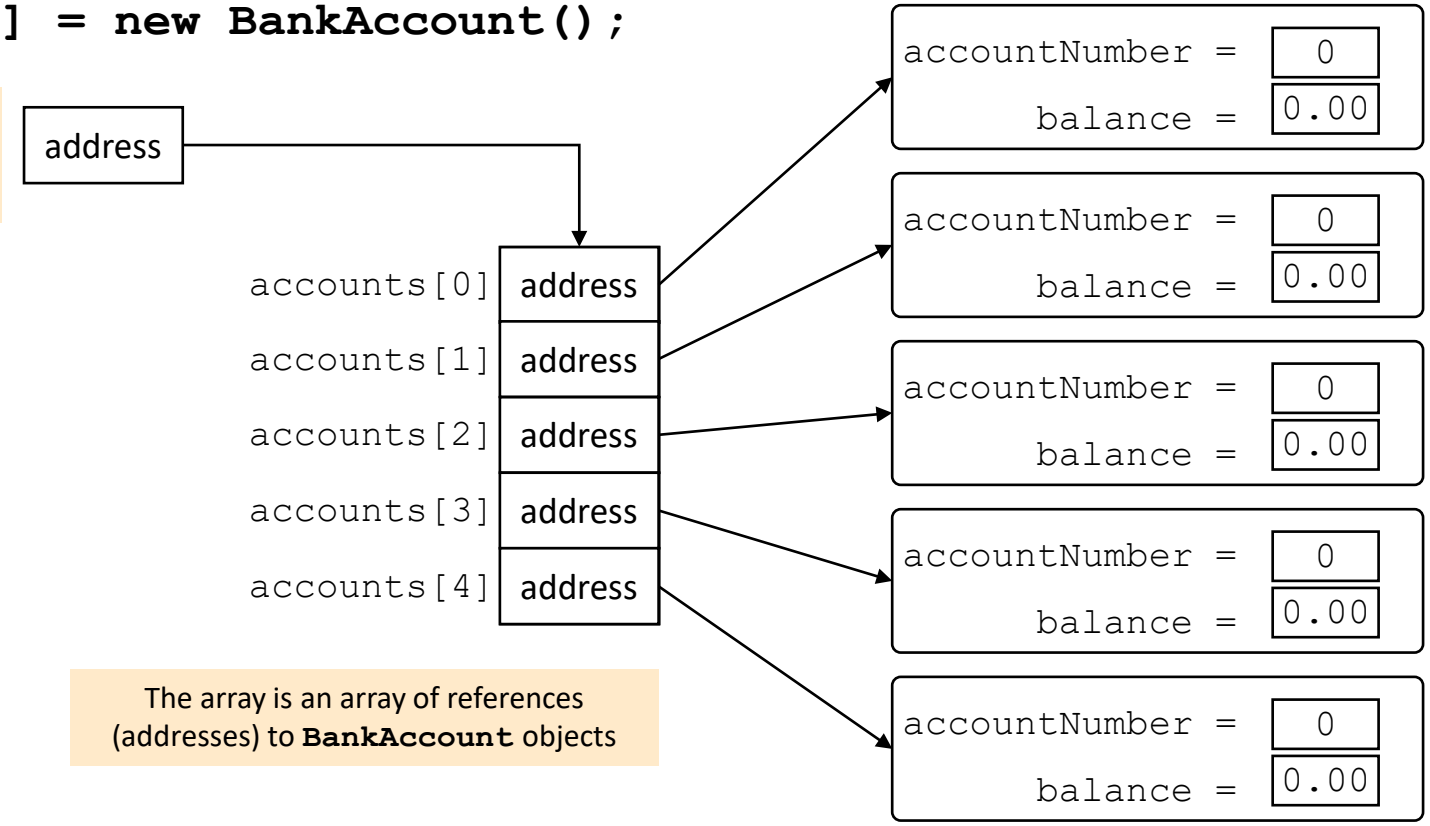
null is a special value which is a default value for any reference variable

The array is an array of references (addresses) to **BankAccount** objects

Initializing Array of Objects

```
BankAccount[] accounts = new BankAccount[5];

for (int i = 0; i < accounts.length; i++)

        accounts[i] = new BankAccount();
```

The **accounts** variable holds the address of an **BankAccount** array.



The array is an array of references (addresses) to **BankAccount** objects

1.4 Accessing Array Elements

Use [ ] to access an element

Assigning values

```
data[2] = 29.95;
accounts[7] = new BankAccount(1021);
```

Using the value stored:

```
double d = data [4];
BankAccount ba = accounts[7];
System.out.println("The value of this data item is " + data[4]);
System.out.println("The balance of this account is " + accounts[7].getBalance());
```

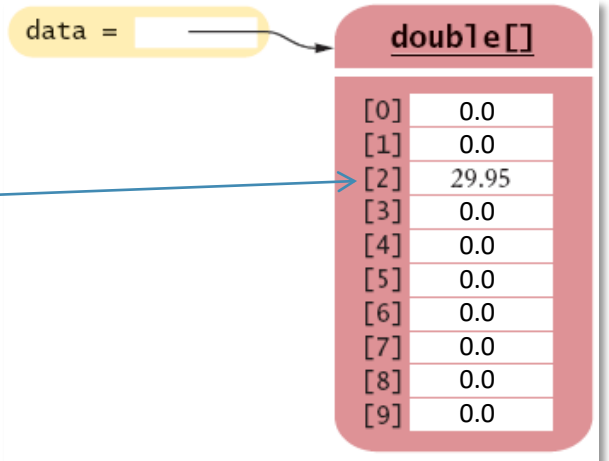


Figure 2 Storing a Value in an Array

Accessing Array Elements

To get array size

```
//Syntax
variablename.length
// Example
data.length
accounts.length
```

(length is a property of an array.
It is NOT a method!)

When array is created without defining values, all values are initialized to the default values depending on the array's data type:

- Numbers: 0 (for `int`) or 0.0 (for `double`)
- Boolean: `false`
- Object References: `null`

```
// Example
int nums = new int[5];
System.out.println(nums[3]);  // 0
```

Tip! In stead of using a number as a size declarator, it is common practice to use a 'final' variable.

```
final int ARRAY_SIZE = 10;
int[] numbers = new int[ARRAY_SIZE];
```

Common Errors: Accessing Array Elements

- Accessing a **nonexistent element** results in a bounds error
  - Index values range from 0 to length – 1

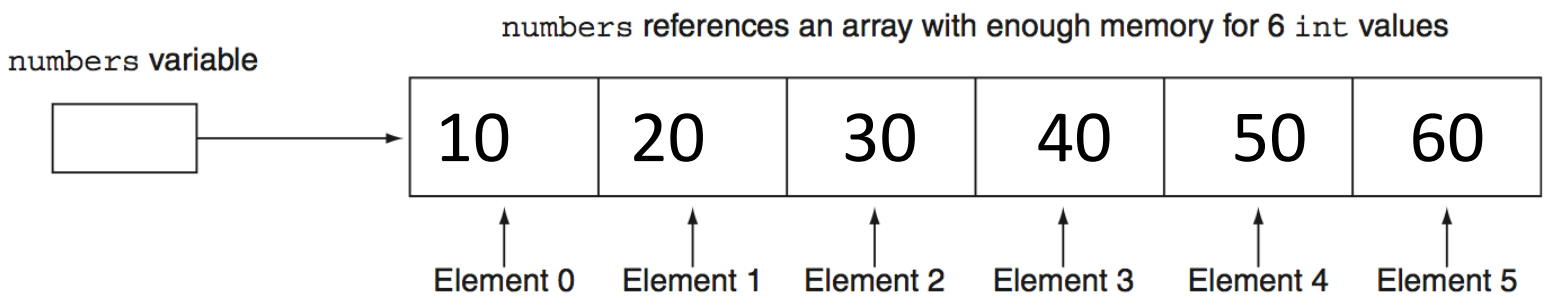
```
double[] data = new double[10];
data[10] = 29.95;      // 'Run-time ERROR'
// java.lang.ArrayIndexOutOfBoundsException: 10
```

- Accessing **uninitialized Arrays**

```
double[] data;
data[0] = 29.95;       // 'Compile-time ERROR'
// The local variable data may not have been initialized
```

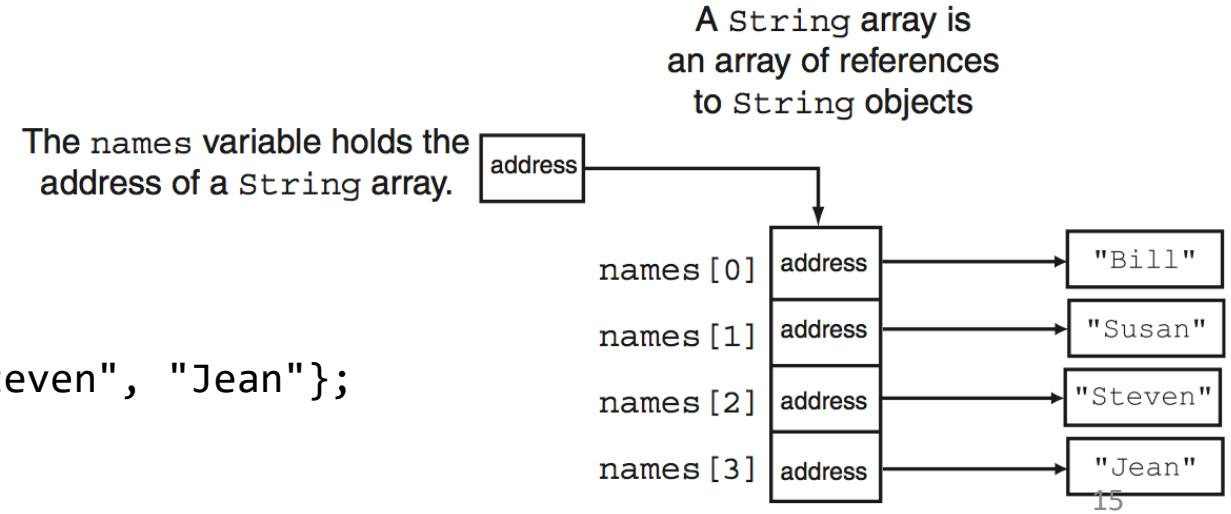
Summary: Primitive Type VS Objects

```
int[] numbers = new int[6];
numbers[0] = 10;
numbers[1] = 20;
numbers[2] = 30;
numbers[3] = 40;
numbers[4] = 50;
numbers[5] = 60;
```




```
int[] numbers = {10,20,30,40,50,60};
```

```
String[] names = new String[4];
names[0] = "Bill";
names[1] = "Susan";
names[2] = "Steven";
names[3] = "Jean";

String[] names = {"Bill", "Susan", "Steven", "Jean"};
```



Self Check

What elements does the data array contain after the following statements?

```
double[] data = new double[5];
for (int i = 0; i < data.length; i++) {
    data[i] = i * i;
}
```

Which choice is the correct answer

a)	0	0	0	0	0	
b)	0	1	4	9	16	
c)	0	1	4	9	16	25
d)	1	4	9	16	25	

Self Check

What do the following program segments print? Matching the results with the provided code.

a)
```
double[] a = new double[10];
System.out.println(a[0]);
```

b)
```
double[] b = new double[10];
System.out.println(b[10]);
```

c)
```
double[] c;
System.out.println(c[0]);
```

1) a run-time error:
   array index out of bounds

2) a compile-time error:
   array is not initialized

3) 0.0

1.5 Array in Methods

- Does the following `swap` method work? Why or why not?

```
public static void main(String[] args) {
    int a = 7;
    int b = 35;

    // swap a with b?
    swap(a, b);

    System.out.println(a + " " + b);
}

public static void swap(int a, int b) {
    int temp = a;
    a = b;
    b = temp;
}
```

// OUTPUT ??

Primitive: Value semantic

- **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
```

Primitive Types Variable as Parameter

- When you pass a variable of primitive type as an argument to a method, the method gets a copy of value stored in the variable.
- Any modification made to the variable will not be visible to the caller

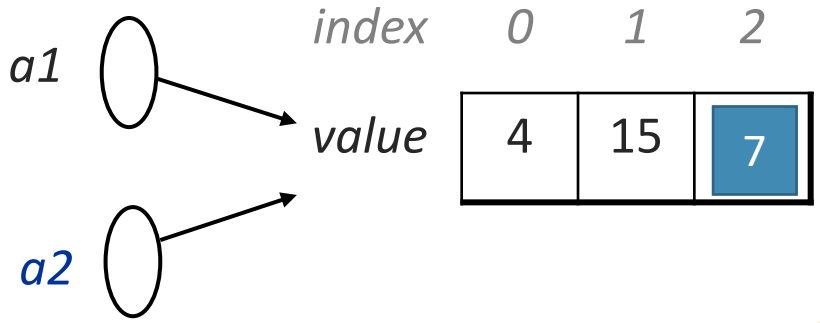
```
public void increaseNum(int n) {
    n++;
}
```

```
int n = 5;
increaeseNum(n);
System.out.println(n);      // OUTPUT?
```

Object: Reference semantic

- **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. BE CAREFUL!!!
  - Array variable is an object variable, so it behaves the same way.

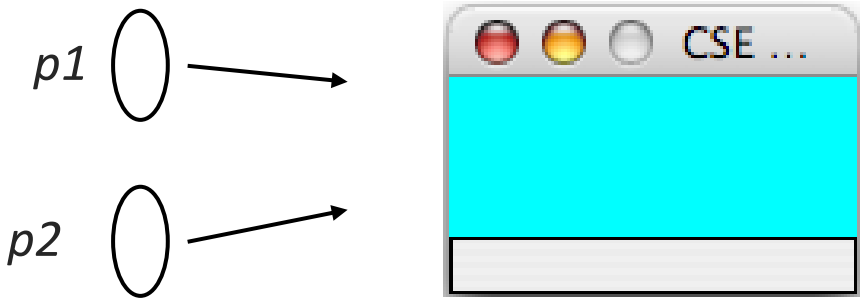
```
int[] a1 = {4, 15, 8};
int[] a2 = a1;     // refer to same array as a1
a2[0] = 7;
System.out.println(a1[0]);        // 7
```



Why Reference Semantic?

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

```
Panel p1 = new Panel(80, 50);
Panel p2 = p1;    // same window
p2.setBackground(Color.CYAN);
```

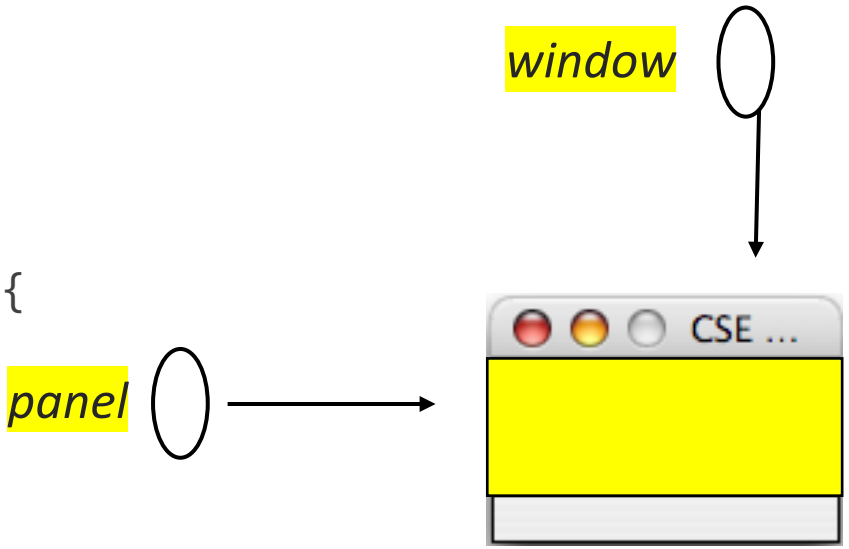


Note that Panel is a user-defined class

Objects as parameters

- 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.

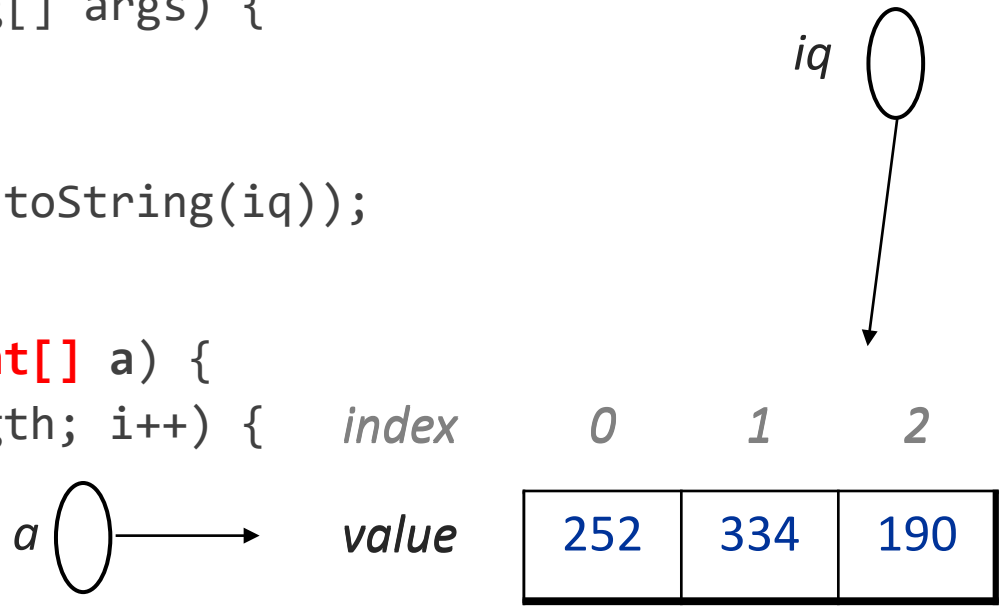
```
public static void main(String[] args) {
    Panel window = new Panel(80, 50);
    window.setBackground(Color.YELLOW);
    example(window);
}

public static void example(Panel panel) {
    panel.setBackground(Color.CYAN);
    ...
}
```



Arrays as Parameter: Arrays pass by reference

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

```
public static void main(String[] args) {
    int[] iq = {126, 167, 95};
    increase(iq);
    System.out.println(Arrays.toString(iq));
}

public static void increase(int[] a) {
    for (int i = 0; i < a.length; i++) {
        a[i] = a[i] * 2;
    }
}
```



Output: `[252, 334, 190]`

```java
public class BankAccount {
    public int accountNumber;
    public double balance;

    public BankAccount(int accNum, double balance){
        this.accountNumber = accNum;
        this.balance = balance;
    }

    public void change(int num){
        System.out.println("Inside increase");

        // change value of primitive type
        num = 888;
        System.out.println("num = " + num);

        // change value of reference type
        this.balance = 999;
        System.out.println("BankAccount = " + this.balance);
    }

    public static void main(String[] args){
        BankAccount myAccount = new BankAccount(1, 100);
        int amount = 10;
        myAccount.change(amount);
        //BankAccount.change(amount, myAccount);
        System.out.println("Inside main");
        System.out.println("amount = " + amount);
        System.out.println("BankAccount = " + myAccount.balance);
    }
}
```

Account Number
Balance

No.1, $100

OUTPUT

Inside increase
num = 888
BankAccount = 999.0
Inside main
amount = 10
BankAccount = 999.0

Primitive type – method gets a copy of value

Object type – method gets the object's reference

2. Simple Array Algorithms

- Enhanced `for` Loop
- Finding Minimum, Maximum, and Average
- Comparing Arrays
- Copying Array
- Inserting and Deleting Element to and from Array at specific index
- Growing Array

2.1 The Generalized `for` Loop

- Traverses all elements of a collection:

```
double[] data = . . .;
double sum = 0;
for (double e : data) {
   sum = sum + e;
}
```

You should read this loop as "for each e in data"

In each iteration, the variable is assigned the next element of the collection. Then the statement is executed.

- Traditional alternative:

```
double[] data = . . .;
double sum = 0;
for (int i = 0; i < data.length; i++)
{
   double e = data[i];
   sum = sum + e;
}
```

Self Check

Write a "for each" loop that prints all elements in the array `data`.

Why is the "for each" loop not an appropriate shortcut for the following ordinary `for` loop?

```
for (int i = 0; i < data.length; i++)
    data[i] = i * i;
```

2.2 Finding Min, Max, Average

```
final int ARRAY_SIZE = 50;
int[] numbers = new int[ARRAY_SIZE];

int highest = numbers[0];
for (int index = 1; index < numbers.length; index++) {
    if (numbers[index] > highest)
        highest = numbers[index];
}

int lowest = numbers[0];
for (int index = 1; index < numbers.length; index++) {
    if (numbers[index] < lowest)
        lowest = numbers[index];
}

double total = 0;
for (int index = 1; index < numbers.length; index++) {
    total += numbers[index];
}
double average = total/numbers.length;
```

2.3 Comparing Arrays

```
int[] firstArray = { 5, 10, 15, 20, 25 };
int[] secondArray = { 5, 10, 15, 20, 25 };
if (firstArray == secondArray)   // This is a mistake. Why?
      System.out.println("The arrays are the same.");
else
      System.out.println("The arrays are not the same.");
```

```
boolean arraysEqual = true;           // Flag variable
int index = 0;                        // Loop control variable

// First determine whether the arrays are the same size.
if (firstArray.length != secondArray.length)
      arraysEqual = false;

// Next determine whether the elements contain the same data.
while (arraysEqual && index < firstArray.length){
      if (firstArray[index] != secondArray[index])
            arraysEqual = false;
      else
            index++;
}
if (arraysEqual)
      System.out.println("The arrays are equal.");
else
      System.out.println("The arrays are not equal.");
```

2.4 Copying Arrays: Copying Array References

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

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

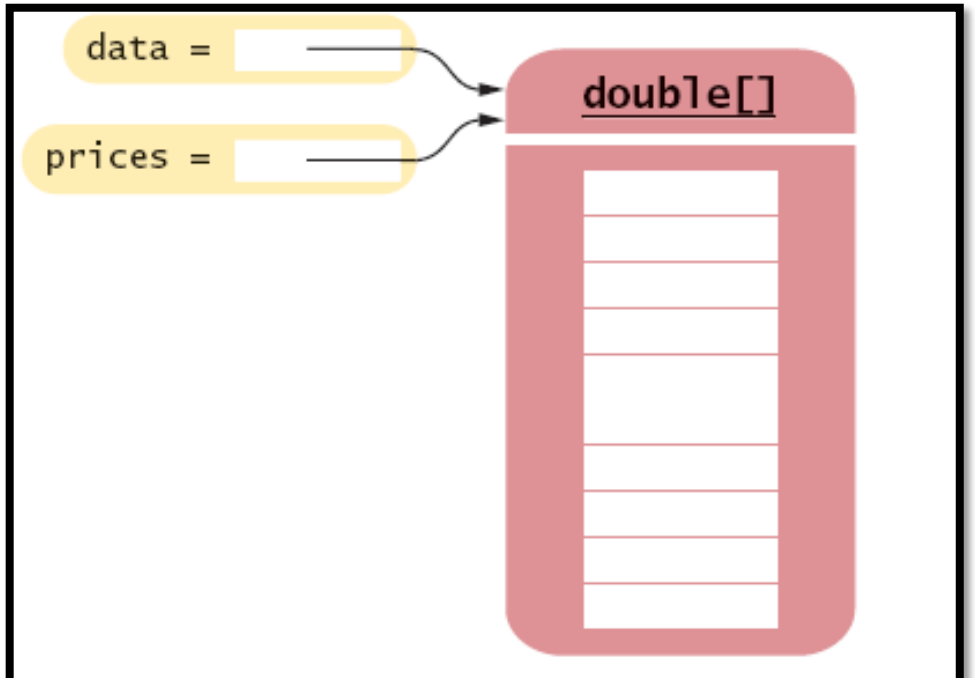


Figure 7 Two References to the Same Array

Copying Arrays: Cloning Arrays

Use `clone` to make true copy (Return from a method `clone` is an `"Object"` type.

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

Don't forget to cast

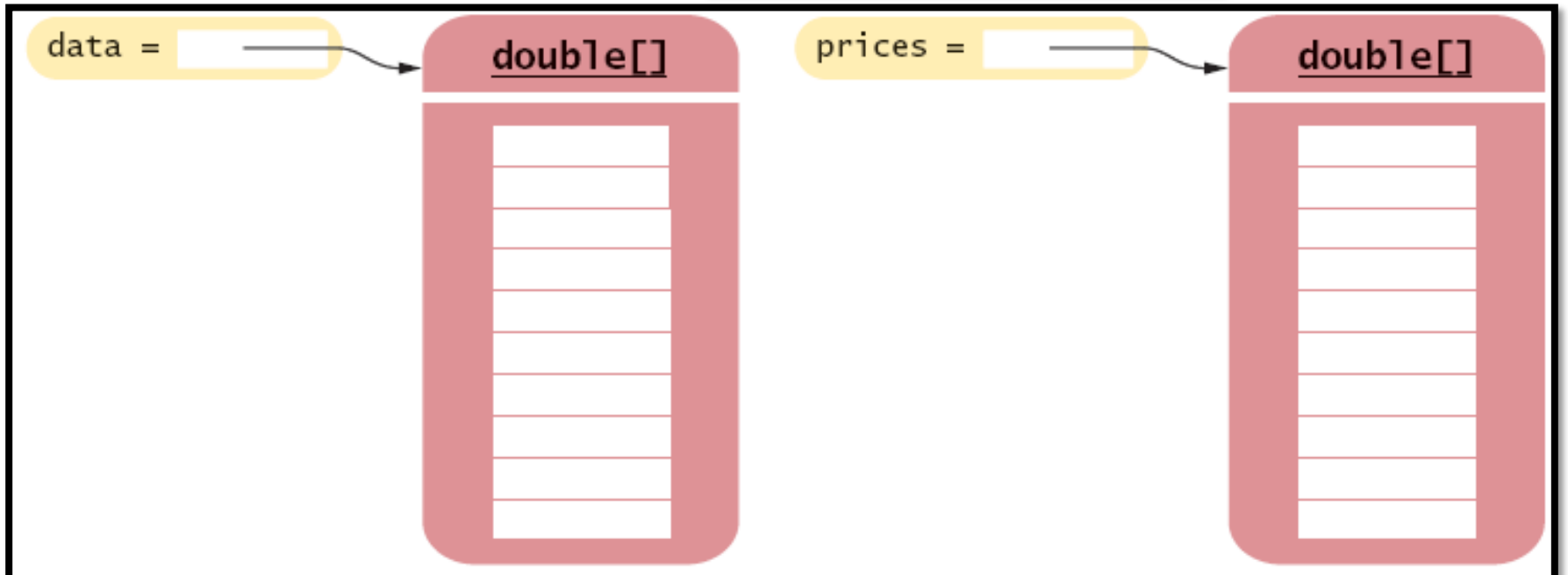


Figure 8 Cloning an Array

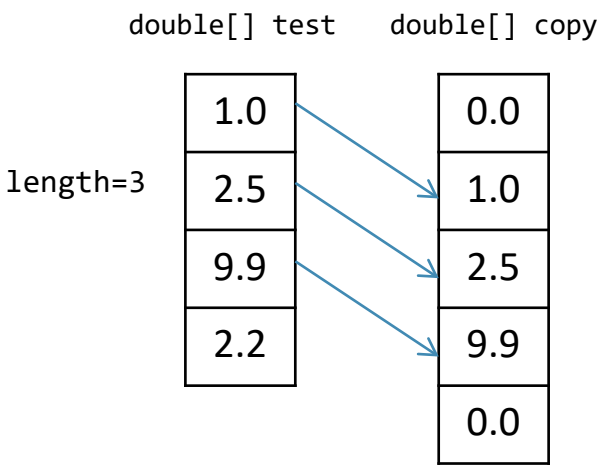
Copying Array Elements

- `System.arraycopy`:
- Copies an array from the specified source array, beginning at the specified position, to the specified position of the destination array.

```
System.arraycopy(src, srcPos, dest, destPos, length);
```

- Parameters:
  - **src** the source array.
  - **srcPos** starting position in the source array.
  - **dest** the destination array.
  - **destPos** starting position in the destination data.
  - **length** the number of array elements to be copied.

```
double[] test = {1.0, 2.5, 9.9, 2.2};
double[] copy = new double[5];
System.arraycopy(test, 0, copy, 1, 3);
for(double x: copy){
    System.out.println(x);
}
```



Output

```
0.0
1.0
2.5
9.9
0.0
```

what if `src` and `dest` array are the same?

2.5 Inserting an Element to an Array

First move all elements from i onward one position up

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

Then insert the new value

```
data[i] = x;
```

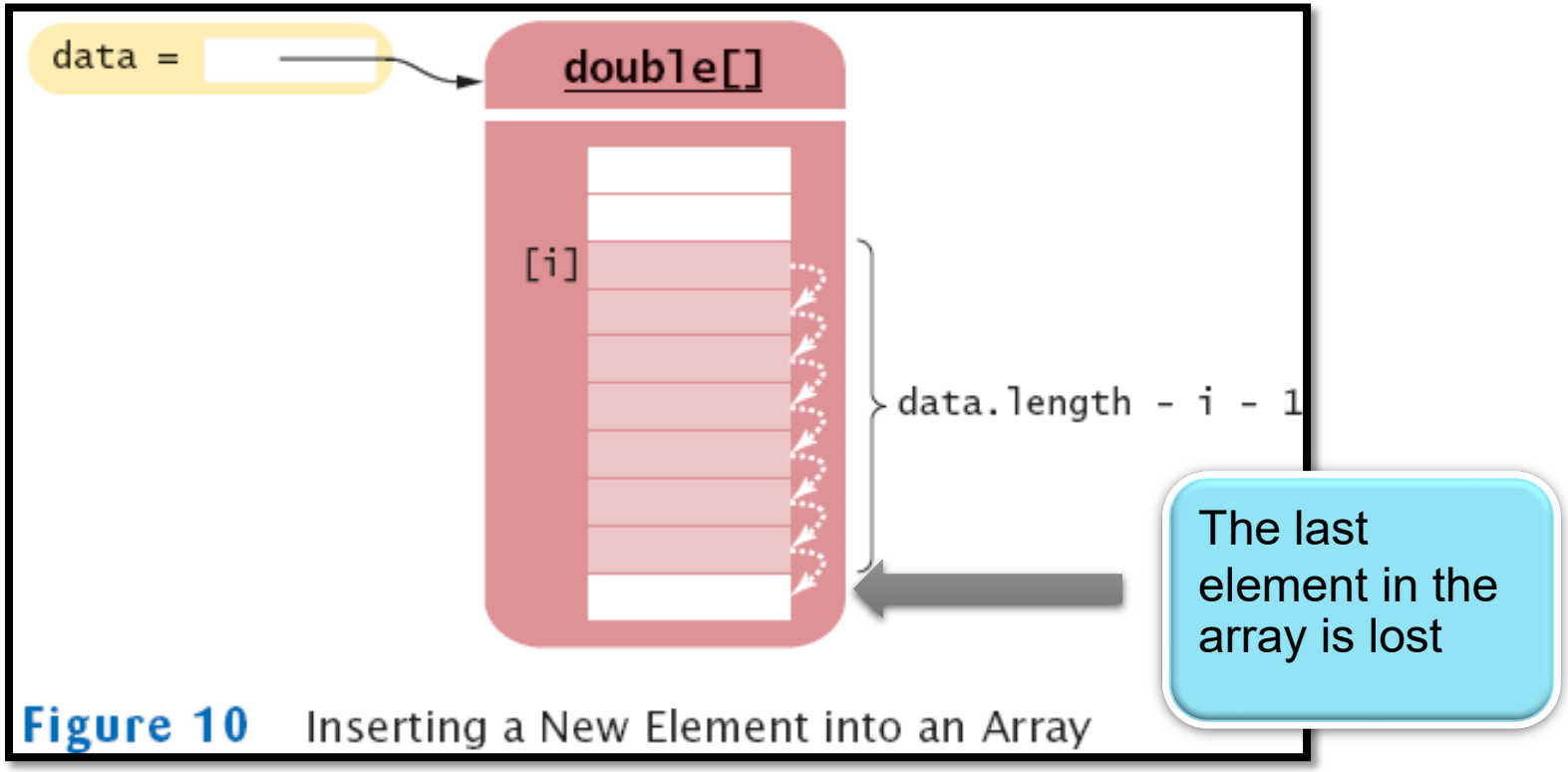


Figure 10 Inserting a New Element into an Array

2.5 Deleting an Element from an Array

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

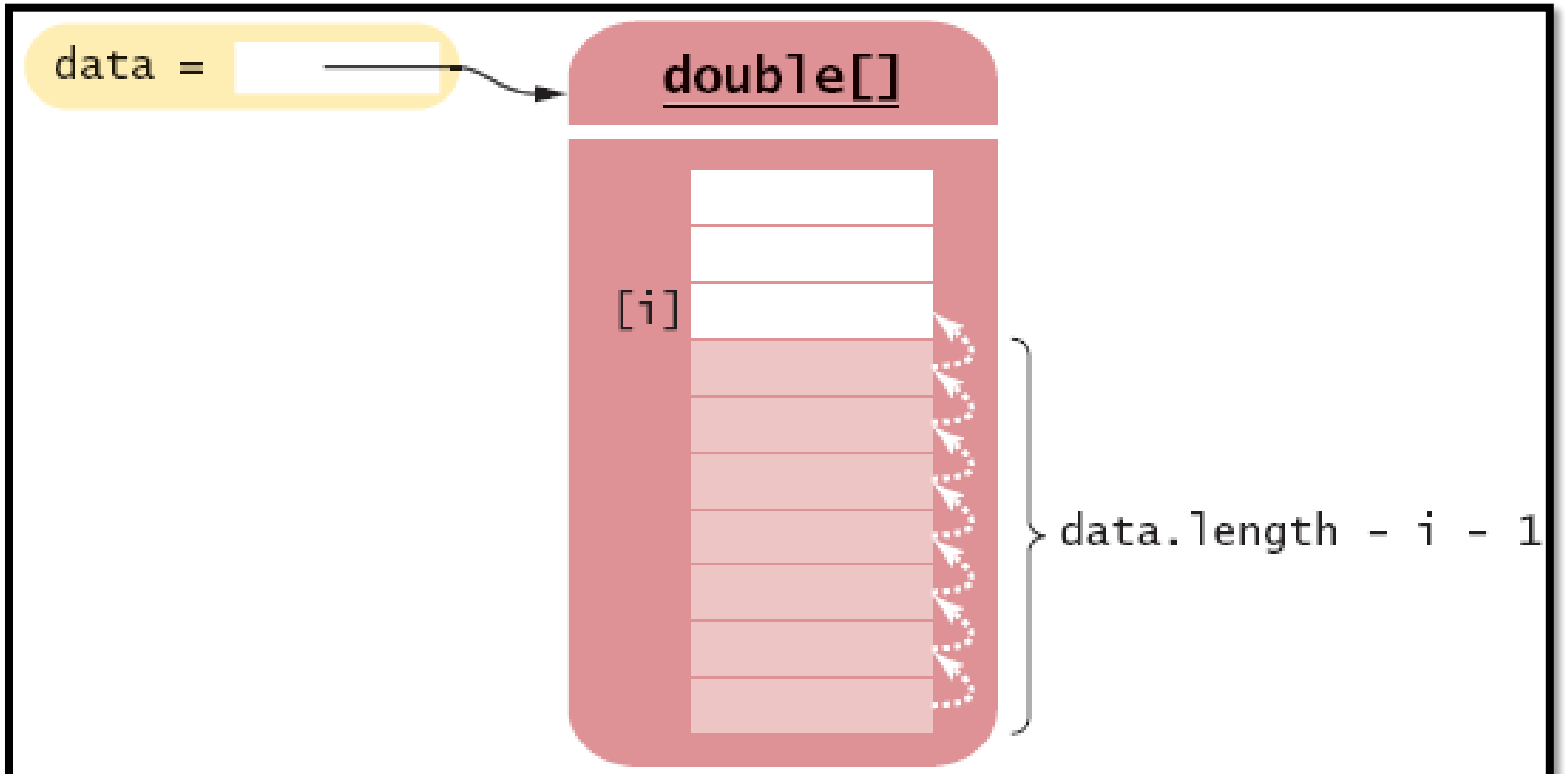


Figure 11 Removing an Element from an Array

2.5 Growing an Array

- If the array is full and you need more space, you can grow the array:

- Create a new, larger array:
  `double[] newData = new double[2 * data.length];` 1

- Copy all elements into the new array:
  `System.arraycopy(data, 0, newData, 0, data.length);` 2

- Store the reference to the new array in the array variable:
  `data = newData;` 3

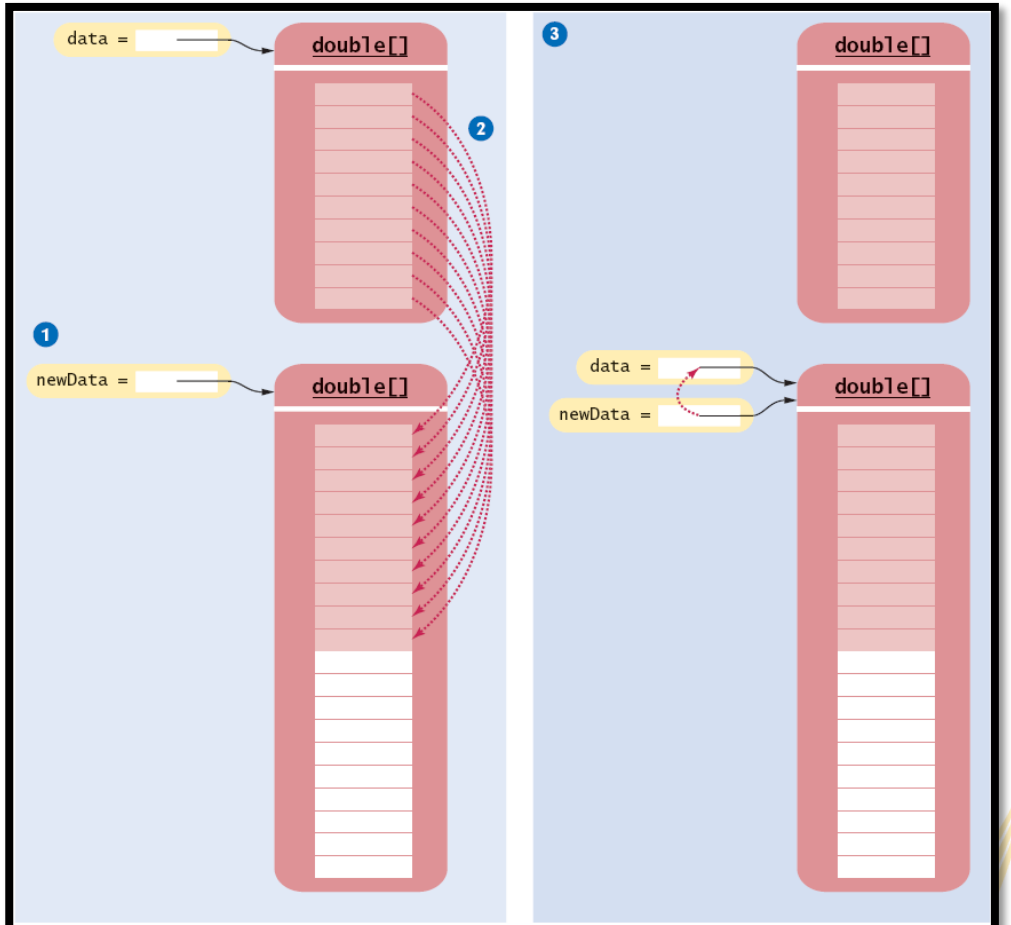


Figure 12 Growing an Array

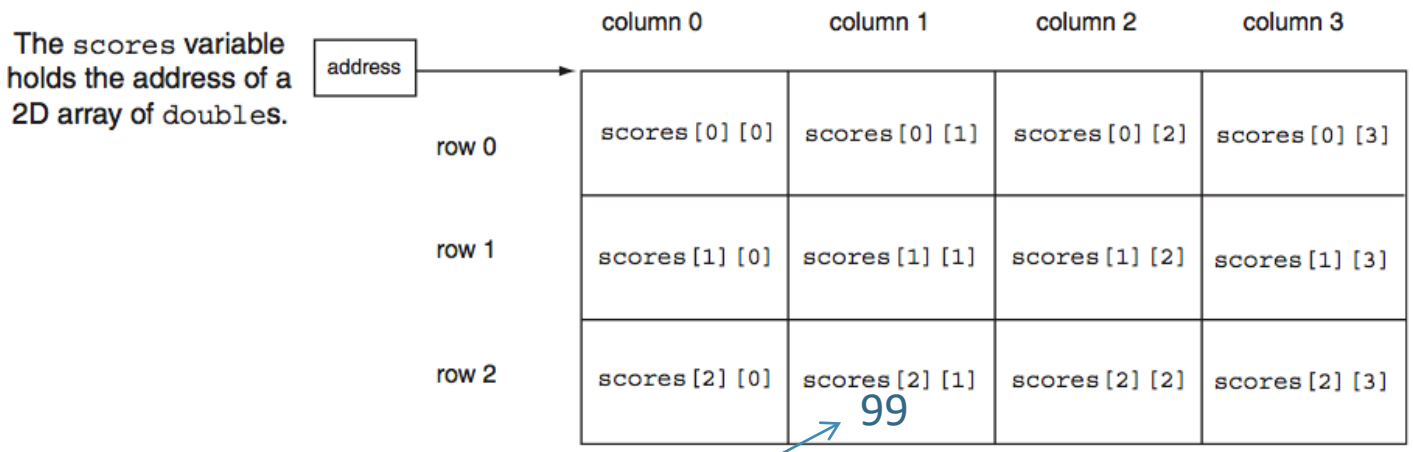
3. Two Dimensional Arrays

- Constructing and Initializing 2D Arrays
- Traversing 2D Arrays
- `length` field in 2D Arrays
- Summing Values in 2DArrays
- Three or More Dimensional Arrays
- Tic-Tac-Toe Board Game

3.1 Constructing and Initializing 2D Arrays

When constructing a two-dimensional array,
you specify how many rows and columns you need:

```
double[][] scores = new double[3][4];
```

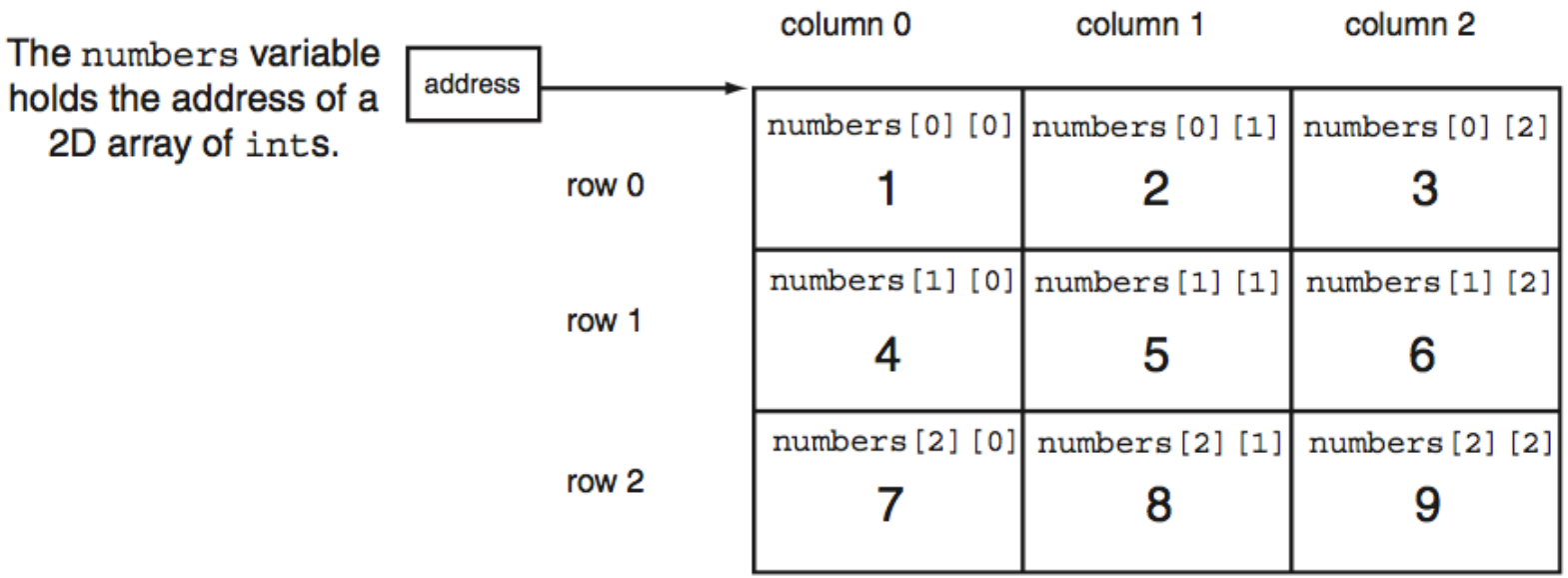


- You access elements with an index pair `variable_name[i][j]`

```
scores[2][1] = 99;
```

Initializing

```
int[][] numbers = { {1, 2, 3}, {4, 5, 6}, {7, 8, 9} };

int[][] numbers = { {1, 2, 3},
                    {4, 5, 6},
                    {7, 8, 9} };
```



Example: A Tic-Tac-Toe Board

Full source code for Tic-TacT-oe board game application: including **class `TicTacToe` (TicTacToe.java)** and **class TicTacToeRunner (TicTacToeRunner.java)** can be found in **Appendix A** at the end of this presentation

```
final int ROWS = 3;
final int COLUMNS = 3;
String[][] board = new String[ROWS][COLUMNS];

board[1][1] = "X";
board[2][1] = "O";
```

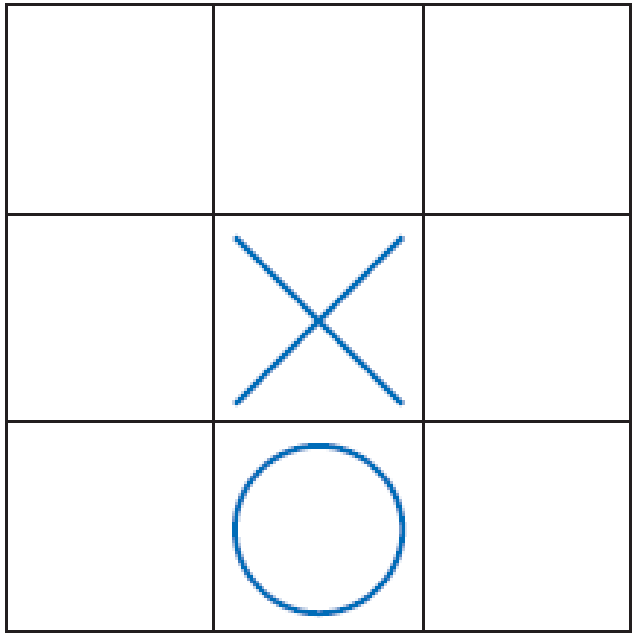


Figure 6
A Tic-Tac-Toe Board

3.2 Traversing Two-Dimensional Arrays

It is common to use two nested loops when filling or searching:

```
for (int i = 0; i < ROWS; i++)
   for (int j = 0; j < COLUMNS; j++)
      board[i][j] = " ";
```

Traversing Two-Dimensional Arrays

- prompts the user to enter a score, once for each element in the scores array.

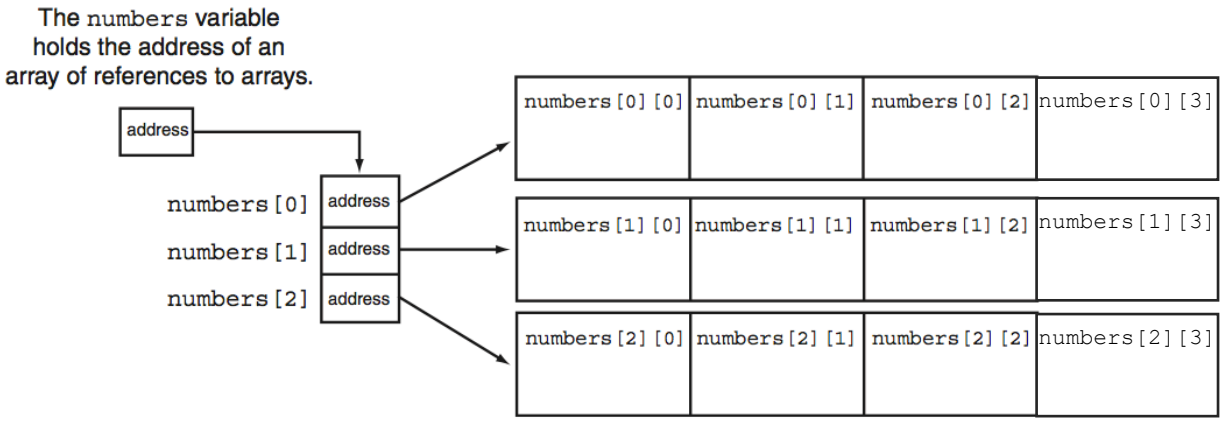
```
final int ROWS = 3;
final int COLS = 4;
double[][] scores = new double[ROWS][COLS];
double number;
Scanner keyboard = new Scanner(System.in);
for (int row = 0; row < ROWS; row++){
    for (int col = 0; col < COLS; col++) {
        System.out.print("Enter a score: ");
        number = keyboard.nextDouble();
        scores[row][col] = number;
    }
}
```

- displays all the elements in the scores array.

```
for (int row = 0; row < ROWS; row++) {
    for (int col = 0; col < COLS; col++) {
        System.out.println(scores[row][col]);
    }
}
```

3.3 length field in 2D Array

```
int[][] numbers = { { 1, 2, 3, 4 },{ 5, 6, 7, 8 }, { 9, 10, 11, 12 } };

// Display the number of rows.
System.out.println("The number of rows is " + numbers.length);

// Display the number of columns in each row.
for (int index = 0; index < numbers.length; index++) {
    System.out.println("The number of columns " + "in row " + index + " is "
                       + numbers[index].length);
}

// Display the number of columns
System.out.println("The number of rows is " + numbers[0].length);
```



3.4 Summing Array

- Summing the Rows of a Two-Dimensional Array

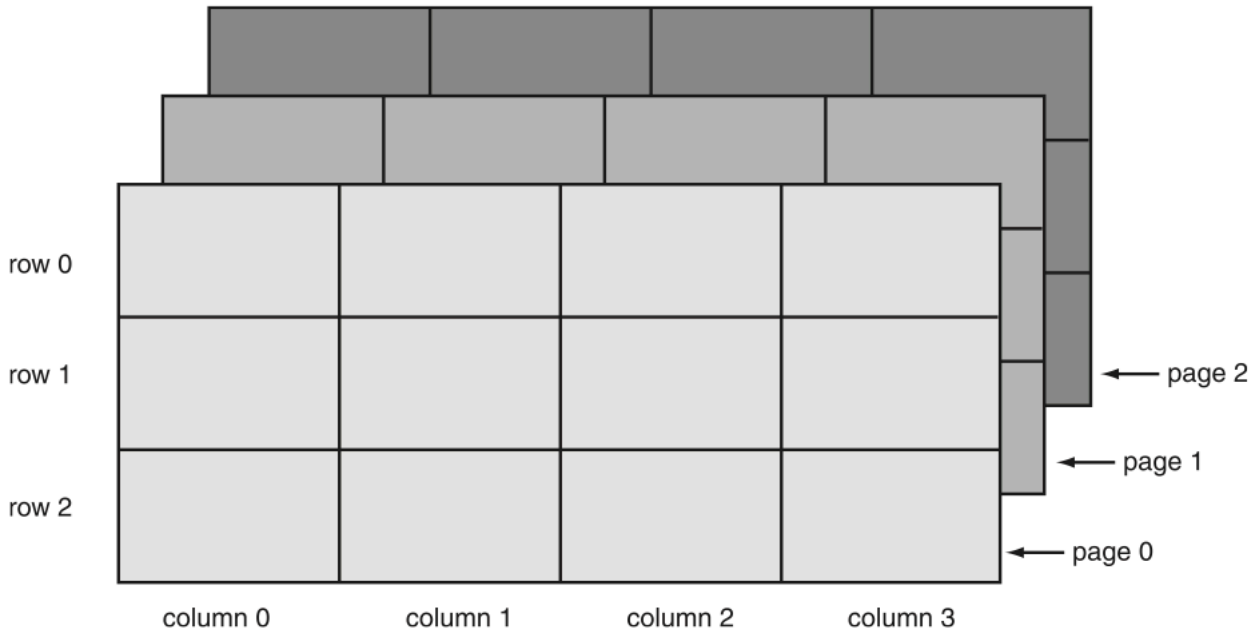
```
int[][] numbers = { { 1, 2, 3, 4 }, { 5, 6, 7, 8 }, { 9, 10, 11, 12 } };
int total = 0; // Start the accumulator at 0.
for (int row = 0; row < numbers.length; row++) {
     total = 0;     // Set the accumulator to 0.

// Sum a row.
     for (int col = 0; col < numbers[row].length; col++)
          total += numbers[row][col]; // Display the row's total.
     System.out.println("Total of row " + row + " is " + total);
}
```

- Summing the Columns of a Two-Dimensional Array

```
int total; // Accumulator
for (int col = 0; col < numbers[0].length; col++) {
     total = 0;     // Set the accumulator to 0.

     // Sum a column.
     for (int row = 0; row < numbers.length; row++)
          total += numbers[row][col];
     // Display the column's total.
     System.out.println("Total of column " + col + " is " + total);
}
```

3.5 Arrays with Three or More Dimensions

```
double[][][] seats = new double[3][4][3];
```



Self Check

How do you declare and initialize a 4-by-4 array of integers?

```
a) int array[][] = new int[4][4];
b) int[][] array = new int[][];
c) int [4][4] array = new int[][];
d) int[][] array = new int[4][4];
```

Which option is correct?

How do you count the number of spaces in the tic-tac-toe board?

```
int count = 0;
 for (int i = 0; i <= ROWS; i++)
    for (int j = 0; j <= COLUMNS; j++)
       if (board[i][j] == ' ')
         count++;
```

Suppose ROWS is number of rows,
And COLUMNS is number of columns.
Can you fix this code?

4. ArrayList

- Introduction to ArrayList
- Retrieving Element
- Changing and Inserting Element
- Wrappers and Auto Boxing

4.1 Introduction to ArrayList

- The `ArrayList` class manages a sequence of objects
- Its length is dynamic. Can grow and shrink as needed
- `ArrayList` class supplies methods for many common tasks, such as inserting and removing elements
- The `ArrayList` class is a generic class:
  `ArrayList<T>` collects objects of type `T`:

Generic class is a class that can be used with many different types of data

```
ArrayList<BankAccount> accounts = new ArrayList<BankAccount>();
accounts.add(new BankAccount(1001));
accounts.add(new BankAccount(1015));
accounts.add(new BankAccount(1022));
```

Add an object to the end of the array list

- The size of the array list increases after each call to add

ArrayList

- **`size`** method yields number of elements (current size of the array list)

```
int i = accounts.size(); // i = 3;
```

- Cannot use primitive types as type parameters
  - There is NO `ArrayList<int>` or `ArrayList<double>`
  - But, there are `ArrayList<Integer>` and `ArrayList<Double>`

- To use the array list, you have to import
  - `import java.util.ArrayList;`

4.2 Retrieving ArrayList Elements

- Use **get** method
- Index starts at 0

```
BankAccount anAccount = accounts.get(2);
// gets the third element of the array list
```

- Bounds error if index is out of range

```
int i = accounts.size();

anAccount = accounts.get(i); // Error
//legal index values are 0. . .i-1
```

4.3 Changing and Inserting Elements

- Use method **set** to change an existing value

```
BankAccount anAccount = new BankAccount(1729);
accounts.set(2, anAccount);
```

  - Set position 2 of the accounts array list to anAccount
  - Overwriting whatever value was there before

- Use method add to add a new object to the end of the array list

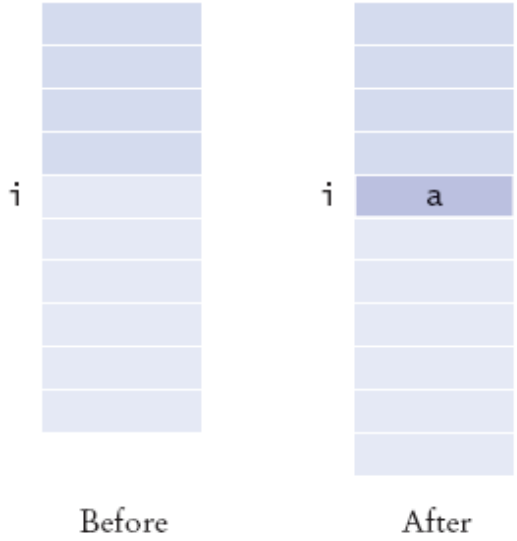
```
accounts.add(anAccount);
```

- Use method `add` to insert a new value before the index

```
accounts.add(i, a);
```

Insert at position i

Inserted object



Removing Elements

- remove removes an element at an index
- `accounts.remove(i)`

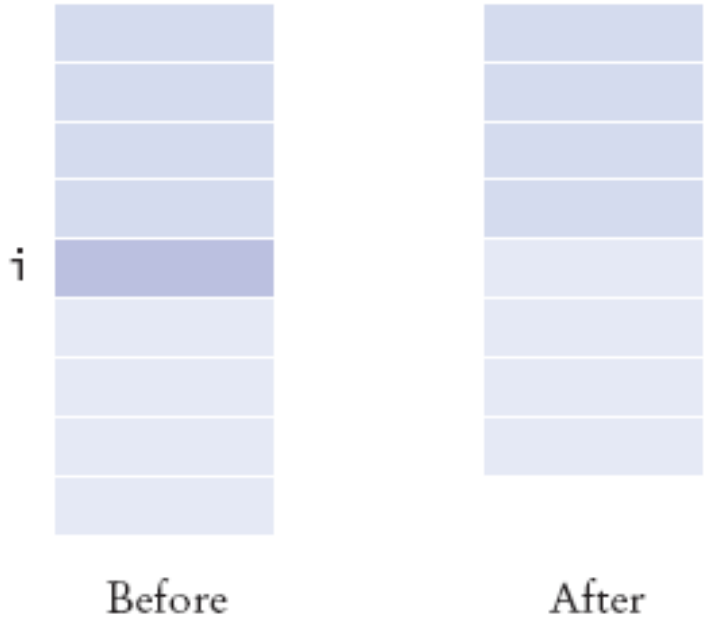


Figure 4 Removing an Element from the Middle of an Array List

```java
int[] numArr = new int[5];
numArr[0] = 2;
numArr[2] = 5;            //numArr[5] = 10; //ERROR

System.out.println("----------- Array --------------");
for(int i = 0; i < numArr.length; i++){
    System.out.println("value at index " + i + " is " + numArr[i]);
}
```

```
----------- Array -------------
value at index 0 is 2
value at index 1 is 0
value at index 2 is 5
value at index 3 is 0
value at index 4 is 0
```

```java
List<String> nameList = new ArrayList<String>();
nameList.add("a");
nameList.add("b");

System.out.println("----------- List --------------");
for(int i = 0; i < nameList.size(); i++){
    System.out.println("name at index "+i+" is "+ nameList.get(i));
}
```

```
----------- List -------------
name at index 0 is a
name at index 1 is b
```

```java
class Card{
    String name;
    int hp;

    public Card(String name, int hp){
        this.name = name;
        this.hp = hp;
    }

    public int getHP(){
        return this.hp;
    }

    public void setHP(int hp){
        this.hp = hp;
    }

    public String toString(){
        return "name: " + this.name + ", hp: " + this.hp;
    }
}
```

```java
Card[] cardArray = new Card[5];
cardArray[0] = new Card("Mario", 5000);
cardArray[2] = new Card("Pikachu", 7000);

System.out.println("----------- Array --------------");
for(int i = 0; i < cardArray.length; i++){
    System.out.println("value at index " + i + " is " + cardArray[i].toString());
}
```

```
----------- Array -------------
value at index 0 is name: Mario, hp: 5000
value at index 1 is null
value at index 2 is name: Pikachu, hp: 7000
value at index 3 is null
value at index 4 is null
```

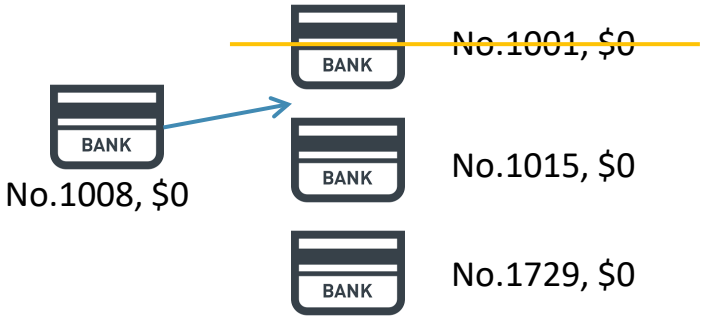
```java
List<Card> cardList = new ArrayList<Card>();
cardList.add(new Card("Doraemon", 10000));
cardList.add(new Card("Nobita", 200));

System.out.println("----------- List --------------");
for(int i = 0; i < cardList.size(); i++){
  System.out.println("card at index " + i + " is " + cardList.get(i).toString());
}
```

```
----------- List -------------
name at index 0 is name: Doraemon, hp: 10000
name at index 1 is name: Nobita, hp: 200
```

File ArrayListTester.java

```
01: import java.util.ArrayList;
02:
03: /**
04:    This program tests the ArrayList class.
05: */
06: public class ArrayListTester
07: {
08:    public static void main(String[] args)
09:    {
10:       ArrayList<BankAccount> accounts
11:             = new ArrayList<BankAccount>();
12:       accounts.add(new BankAccount(1001));
13:       accounts.add(new BankAccount(1015));
14:       accounts.add(new BankAccount(1729));
15:       accounts.add(1, new BankAccount(1008));
16:       accounts.remove(0);
17:
18:       System.out.println("Size: " + accounts.size());
19:       System.out.println("Expected: 3");
20:       BankAccount first = accounts.get(0);
```

Full source code for **Class BankAccount (BankAccount.java)** can be found in **Appendix A** at the end of this presentation



File ArrayListTester.java (cont.)

```
21:         System.out.println("First account number: "
22:               + first.getAccountNumber());
23:         System.out.println("Expected: 1008");
24:         BankAccount last = accounts.get(accounts.size() - 1);
25:         System.out.println("Last account number: "
26:               + last.getAccountNumber());
27:         System.out.println("Expected: 1729");
28:      }
29: }
```

No.1008, $0

No.1015, $0

No.1729, $0

Output:

```
Size: 3
Expected: 3
First account number: 1008
Expected: 1008
Last account number: 1729
Expected: 1729
```

Self Check

How do you construct an array of 10 strings? An array list of strings?

What is the content of `names` after the following statements?

```
ArrayList<String> names = new ArrayList<String>();
names.add("A");
names.add(0, "B");
names.add("C");
names.remove(1);
```

Self Check

How do you construct an array of 10 strings? An array list of strings?

Answer:

```
new String[10];
new ArrayList<String>();
```

What is the content of `names` after the following statements?

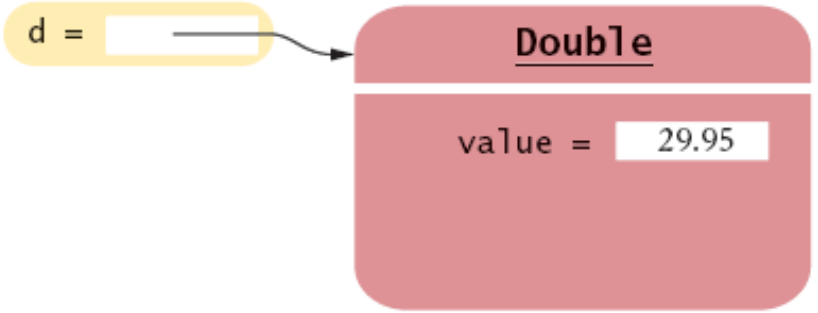
```
ArrayList<String> names = new ArrayList<String>();
names.add("A");
names.add(0, "B");
names.add("C");
names.remove(1);
```

Answer: `names` contains the strings `"B"` and `"C"` at positions 0 and 1

4.4 Wrappers & Auto-Boxing

- You cannot insert primitive types directly into array lists
- To treat primitive type values as objects, you must use **wrapper classes:**

```
// How to use wrapper class
Double d = 29.95;
// OR
Double d = new Double(29.95);
```




```
// ArrayList
ArrayList<double> data = new ArrayList<double>();  // ERROR!!

ArrayList<Double> data = new ArrayList<Double>();  // OK
data.add(29.95);
double d = data.get(0);
```

Wrappers

There are wrapper classes for all eight primitive types:

Primitive Type	Wrapper Class
byte	Byte
boolean	Boolean
char	Character
double	Double
float	Float
int	Integer
long	Long
short	Short

They are called wrappers because they are small classes that has a primitive value instead. This is similar to a gift that we wrap with a thin paper around it.

Auto-boxing

- Auto-boxing: Starting with Java 5.0, **conversion between primitive types and the corresponding wrapper classes** is **automatic**.

```
Double wrapper = 29.95;
// auto-boxing; same as Double wrapper = new Double(29.95);

double x = wrapper;
// auto-unboxing; same as double x = wrapper.doubleValue();

// ArrayList
ArrayList<Double> values = new ArrayList<Double>();
values.add(42.0);
double theAnswer = values.get(0);
```

- Auto-boxing even works inside arithmetic expressions
  `Double e = wrapper + 1;`
- Means:
  - *auto-unbox d into a* `double`
  - *add 1*
  - *auto-box the result into a new* `Double`
  - *store a reference to the newly created wrapper object in e*

Self Check

What is the difference between the types `double` and `Double`?

Suppose data is an `ArrayList<Double>` of size > 0. How do you increment the element with index 0?

Self Check

What is the difference between the types `double` and `Double`?

Answer: `double` is one of the eight primitive types. `Double` is a class type.

Suppose data is an `ArrayList<Double>` of size > 0. How do you increment the element with index 0?

Answer: `data.set(0, data.get(0) + 1);`

5. ArrayList Algorithms

- Enhanced `for` Loop
- Adding a New Object
- Counting Matched Objects
- Finding a Given Object
- Getting Maximum or Minimum Values

The Generalized `for` Loop

- Works for `ArrayLists` too:

```
ArrayList<BankAccount> accounts = . . . ;
double sum = 0;
for (BankAccount a : accounts)
{
   sum = sum + a.getBalance();
}
```

- Equivalent to the following ordinary `for` loop:

```
double sum = 0;
for (int i = 0; i < accounts.size(); i++)
{
   BankAccount a = accounts.get(i);
   sum = sum + a.getBalance();
}
```

File BankTester.java

Full source code for **Class Bank (Bank.java)** can be found in **Appendix A** at the end of this presentation

```
/**
   This program tests the Bank class.
*/
public class BankTester
{
   public static void main(String[] args)
   {
      Bank firstBankOfJava = new Bank();

      // add new bank accounts
      firstBankOfJava.addAccount(new BankAccount(1001, 20000));
      firstBankOfJava.addAccount(new BankAccount(1015, 10000));
      firstBankOfJava.addAccount(new BankAccount(1729, 15000));

      double threshold = 15000;
      // count number of account(s) that have balance at least 15000
      int c = firstBankOfJava.count(threshold);
      System.out.println("Count: " + c);
      System.out.println("Expected: 2");

      int accountNumber = 1015;
```

1

2

File BankTester.java (Cont.)

```
        // find bank account of a given account number
    BankAccount a = firstBankOfJava.find(accountNumber);
    if (a == null)
       System.out.println("No matching account");
    else
       System.out.println("Balance of matching account: " + a.getBalance());
    System.out.println("Expected: 10000");

        // find account number that have the highest balance
    BankAccount max = firstBankOfJava.getMaximum();
    System.out.println("Account with largest balance: "
            + max.getAccountNumber());
    System.out.println("Expected: 1001");
  }
}
```

3

4

No.1001, $20,000

No.1015, $10,000

No.1729, $15,000

Output:

```
Count: 2
Expected: 2
Balance of matching account: 10000.0
Expected: 10000
Account with largest balance: 1001
Expected: 1001
```

Counting Matches

Check all elements and count the matches until you reach the end of the array list.

```
public class Bank
{
     private ArrayList<BankAccount> accounts;

    public void addAccount(BankAccount a)
     {
          accounts.add(a);
     }

     public int count(double atLeast)
     {
      int matches = 0;
      for (BankAccount a : accounts)
          {
         if (a.getBalance() >= atLeast)
                   matches++;         // Found a match
      }
      return matches;
   }
   . . .
}
```

Finding a Value

Check all elements until you have found a match.

```
public class Bank
{
   public BankAccount find(int accountNumber)
   {
      for (BankAccount a : accounts)
      {
         if (a.getAccountNumber() == accountNumber)
               // Found a match return a;
      }
      return null; // No match in the entire array list
   }
   . . .
}
```

Finding the Maximum or Minimum

- Initialize a candidate with the starting element
- Compare candidate with remaining elements
- Update it if you find a larger or smaller value
- Example:

```
public BankAccount getMaximum(){
    BankAccount largestYet = accounts.get(0);
     for (int i = 1; i < accounts.size(); i++)
     {
         BankAccount a = accounts.get(i);
         if (a.getBalance() > largestYet.getBalance())
            largestYet = a;
     }
     return largestYet;
}
```

- Works only if there is at least one element in the array list . . .
- If list is empty, return `null`:

```
if (accounts.size() == 0) return null;
BankAccount largestYet = accounts.get(0);
   . . .
```

Self Check

What does the `find` method do if there are two bank accounts with a matching account number?

Answer: It returns the first match that it finds.

Would it be possible to use a "for each" loop in the `getMaximum` method?

Answer: Yes, but the first comparison would always fail.

Appendix A: Full Source Code

File BankAccount.java


```
01: /**
02:    A bank account has a balance that can be changed by
03:    deposits and withdrawals.
04: */
05: public class BankAccount
06: {
07:    /**
08:       Constructs a bank account with a zero balance
09:       @param anAccountNumber the account number for this account
10:    */
11:    public BankAccount(int anAccountNumber)
12:    {
13:       accountNumber = anAccountNumber;
14:       balance = 0;
15:    }
16:
17:    /**
18:       Constructs a bank account with a given balance
19:       @param anAccountNumber the account number for this account
20:       @param initialBalance the initial balance
21:    */
```

File BankAccount.java (Cont.)


```
22:     public BankAccount(int anAccountNumber, double initialBalance)
23:     {
24:        accountNumber = anAccountNumber;
25:        balance = initialBalance;
26:     }
27:
28:     /**
29:        Gets the account number of this bank account.
30:        @return the account number
31:     */
32:     public int getAccountNumber()
33:     {
34:        return accountNumber;
35:     }
36:
37:     /**
38:        Deposits money into the bank account.
39:        @param amount the amount to deposit
40:     */
41:     public void deposit(double amount)
42:     {
43:        double newBalance = balance + amount;
44:        balance = newBalance;
45:     }
```

File BankAccount.java (Cont.)


```java

    /**
       Withdraws money from the bank account.
       @param amount the amount to withdraw
    */
    public void withdraw(double amount)
    {
       double newBalance = balance - amount;
       balance = newBalance;
    }

    /**
       Gets the current balance of the bank account.
       @return the current balance
    */
    public double getBalance()
    {
       return balance;
    }

    private int accountNumber;
    private double balance;
}
```

File Bank.java


```
01: import java.util.ArrayList;
02:
03: /**
04:    This bank contains a collection of bank accounts.
05: */
06: public class Bank
07: {
08:    /**
09:       Constructs a bank with no bank accounts.
10:    */
11:    public Bank()
12:    {
13:       accounts = new ArrayList<BankAccount>();
14:    }
15:
16:    /**
17:       Adds an account to this bank.
18:       @param a the account to add
19:    */
20:    public void addAccount(BankAccount a)
21:    {
22:       accounts.add(a);
23:    }
```

File Bank.java (cont.)


```
24:
25:     /**
26:        Gets the sum of the balances of all accounts in this bank.
27:        @return the sum of the balances
28:     */
29:     public double getTotalBalance()
30:     {
31:        double total = 0;
32:        for (BankAccount a : accounts)
33:        {
34:           total = total + a.getBalance();
35:        }
36:        return total;
37:     }
38:
39:     /**
40:        Counts the number of bank accounts whose balance is at
41:        least a given value.
42:        @param atLeast the balance required to count an account
43:        @return the number of accounts having least the given balance
44:     */
45:     public int count(double atLeast)
46:     {
```

File Bank.java (cont.)


```
47:         int matches = 0;
48:         for (BankAccount a : accounts)
49:         {
50:            if (a.getBalance() >= atLeast) matches++; // Found a match
51:         }
52:         return matches;
53:      }
54:
55:      /**
56:         Finds a bank account with a given number.
57:         @param accountNumber the number to find
58:         @return the account with the given number, or null if there
59:         is no such account
60:      */
61:      public BankAccount find(int accountNumber)
62:      {
63:         for (BankAccount a : accounts)
64:         {
65:            if (a.getAccountNumber() == accountNumber) // Found a match
66:               return a;
67:         }
68:         return null; // No match in the entire array list
69:      }
70:
```

File Bank.java (cont.)


```
71:     /**
72:        Gets the bank account with the largest balance.
73:        @return the account with the largest balance, or null if the
74:        bank has no accounts
75:     */
76:     public BankAccount getMaximum()
77:     {
78:        if (accounts.size() == 0) return null;
79:        BankAccount largestYet = accounts.get(0);
80:        for (int i = 1; i < accounts.size(); i++)
81:        {
82:           BankAccount a = accounts.get(i);
83:           if (a.getBalance() > largestYet.getBalance())
84:              largestYet = a;
85:        }
86:        return largestYet;
87:     }
88:
89:     private ArrayList<BankAccount> accounts;
90: }
```

File TicTacToe.java


```
01: /**
02:    A 3 x 3 tic-tac-toe board.
03: */
04: public class TicTacToe
05: {
06:    /**
07:       Constructs an empty board.
08:    */
09:    public TicTacToe()
10:    {
11:       board = new String[ROWS][COLUMNS];
12:       // Fill with spaces
13:       for (int i = 0; i < ROWS; i++)
14:          for (int j = 0; j < COLUMNS; j++)
15:             board[i][j] = " ";
16:    }
17:
18:    /**
19:       Sets a field in the board. The field must be unoccupied.
20:       @param i the row index
21:       @param j the column index
22:       @param player the player ("x" or "o")
23:    */
```

File TicTacToe.java (Cont.)


```
24:     public void set(int i, int j, String player)
25:     {
26:        if (board[i][j].equals(" "))
27:           board[i][j] = player;
28:     }
29:
30:     /**
31:        Creates a string representation of the board, such as
32:        |x  o|
33:        |  x |
34:        |   o|
35:        @return the string representation
36:     */
37:     public String toString()
38:     {
39:        String r = "";
40:        for (int i = 0; i < ROWS; i++)
41:        {
42:           r = r + "|";
43:           for (int j = 0; j < COLUMNS; j++)
44:              r = r + board[i][j];
45:           r = r + "|\n";
```

File TicTacToe.java (Cont.)


```
        }
        return r;
    }

    private String[][] board;
    private static final int ROWS = 3;
    private static final int COLUMNS = 3;
}
```

File TicTacToeRunner.java


```
01: import java.util.Scanner;
02:
03: /**
04:    This program runs a TicTacToe game. It prompts the
05:    user to set positions on the board and prints out the
06:    result.
07: */
08: public class TicTacToeRunner
09: {
10:    public static void main(String[] args)
11:    {
12:       Scanner in = new Scanner(System.in);
13:       String player = "x";
14:       TicTacToe game = new TicTacToe();
15:       boolean done = false;
16:       while (!done)
17:       {
18:          System.out.print(game.toString());
19:          System.out.print(
20:                "Row for " + player + " (-1 to exit): ");
21:          int row = in.nextInt();
22:          if (row < 0) done = true;
23:          else
24:          {
```

File TicTacToeRunner.java (Cont.)


```
            System.out.print("Column for " + player + ": ");
            int column = in.nextInt();
            game.set(row, column, player);
            if (player.equals("x"))
               player = "o";
            else
               player = "x";
         }
      }
   }
}
```