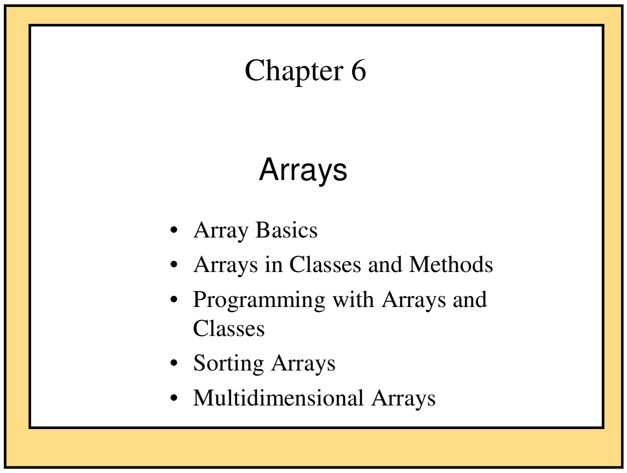
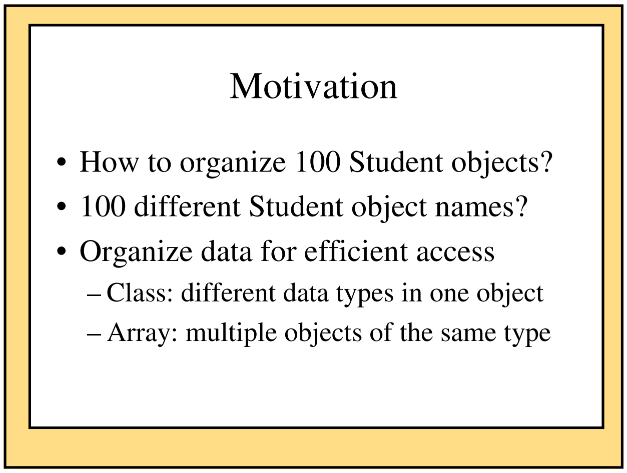


Chapter 6

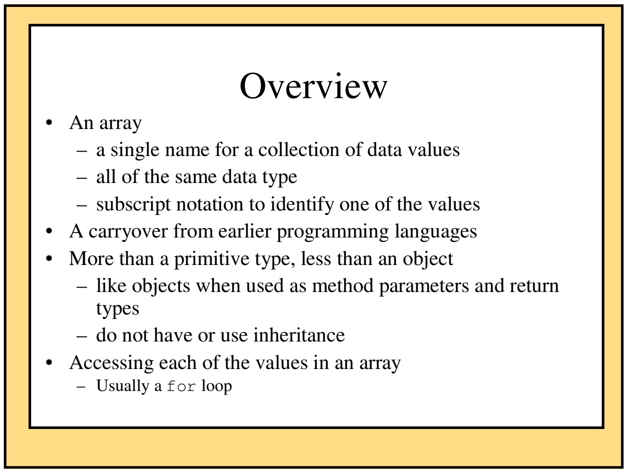
Arrays

- Array Basics
- Arrays in Classes and Methods
- Programming with Arrays and Classes
- Sorting Arrays
- Multidimensional Arrays



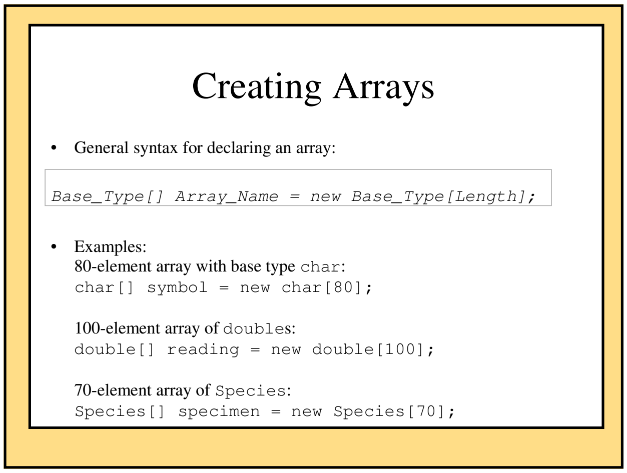
Motivation

- How to organize 100 Student objects?
- 100 different Student object names?
- Organize data for efficient access
  - Class: different data types in one object
  - Array: multiple objects of the same type



Overview

- An array
  - a single name for a collection of data values
  - all of the same data type
  - subscript notation to identify one of the values
- A carryover from earlier programming languages
- More than a primitive type, less than an object
  - like objects when used as method parameters and return types
  - do not have or use inheritance
- Accessing each of the values in an array
  - Usually a `for` loop



Creating Arrays

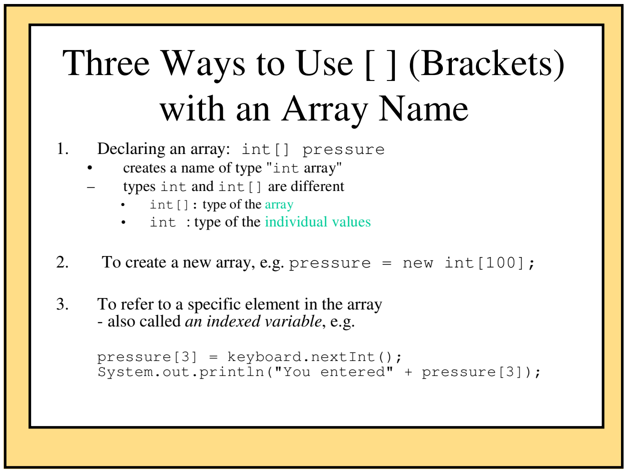
- General syntax for declaring an array:

```
Base_Type[] Array_Name = new Base_Type[Length];
```

- Examples:
  80-element array with base type `char`:
  `char[] symbol = new char[80];`

  100-element array of `double`s:
  `double[] reading = new double[100];`

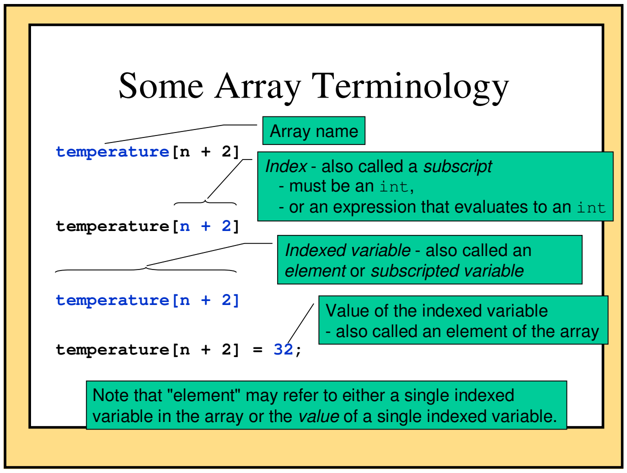
  70-element array of `Species`:
  `Species[] specimen = new Species[70];`



Three Ways to Use [ ] (Brackets) with an Array Name

1. Declaring an array: `int[] pressure`
   - creates a name of type "`int` array"
   - types `int` and `int[]` are different
     - `int[]`: type of the array
     - `int` : type of the individual values
2. To create a new array, e.g. `pressure = new int[100];`
3. To refer to a specific element in the array
   - also called *an indexed variable*, e.g.

```
pressure[3] = keyboard.nextInt();
System.out.println("You entered" + pressure[3]);
```



Some Array Terminology

temperature[n + 2]

Array name

temperature[n + 2]

Index - also called a *subscript*
- must be an `int`,
- or an expression that evaluates to an `int`

temperature[n + 2]

Indexed variable - also called an *element* or *subscripted variable*

temperature[n + 2] = 32;

Value of the indexed variable
- also called an element of the array

Note that "element" may refer to either a single indexed variable in the array or the *value* of a single indexed variable.

Array Length

- Specified by the number in brackets when created with `new`
  - *maximum* number of elements the array can hold
  - storage is allocated whether or not the elements are assigned values
- the attribute `length`,

```
        Species[] entry = new Species[20];
         System.out.println(entry.length);
```

- The `length` attribute is established in the declaration and cannot be changed unless the array is redeclared

Subscript Range

- Array subscripts use zero-numbering
  - the first element has subscript 0
  - the second element has subscript 1
  - etc. - the nth element has subscript n-1
  - the last element has subscript `length-1`
- For example: an int array with 4 elements

Subscript:	0	1	2	3
Value:	97	86	92	71

Subscript out of Range Error

- Using a subscript larger than `length-1` causes a *run time* (not a compiler) error
  - an `ArrayOutOfBoundsException` is thrown
    - you do not need to catch it
    - you need to fix the problem and recompile your code
- Other programming languages, e.g. C and C++, do not even cause a run time error!
  - one of the most dangerous characteristics of these languages is that they allow out of bounds array indices.

Array Length Specified at Run-time

```
// array length specified at compile-time
int[] array1 = new int[10];


// array length specified at run-time
// calculate size…
int size = …;
int[] array2 = new int[size];
```

Programming Tip: Use Singular Array Names

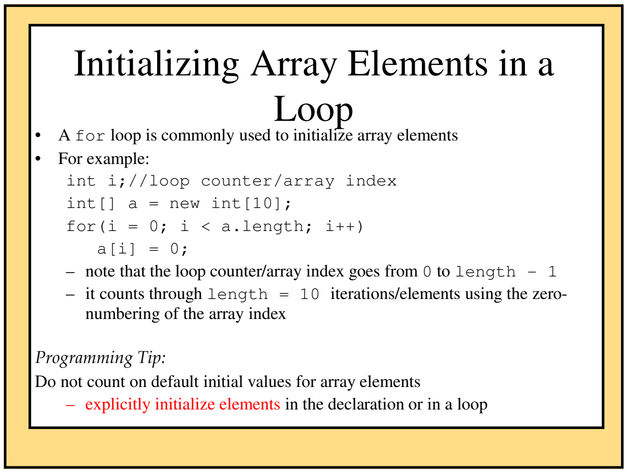
- Using singular rather than plural names for arrays improves readability
- Although the array contains many elements the most common use of the name will be with a subscript, which references a *single* value.
- It is easier to read:
  - `score[3]` than
  - `scores[3]`

Initializing an Array's Values in Its Declaration

- can be initialized by putting a comma-separated list in braces
- Uninitialized elements will be assigned some default value, e.g. 0 for `int` arrays (**explicit initialization** is recommended)
- The length of an array is automatically determined when the values are explicitly initialized in the declaration
- For example:

```
double[] reading = {5.1, 3.02, 9.65};
System.out.println(reading.length);
```

- displays `3`, the length of the array `reading`



Initializing Array Elements in a Loop

- A `for` loop is commonly used to initialize array elements
- For example:

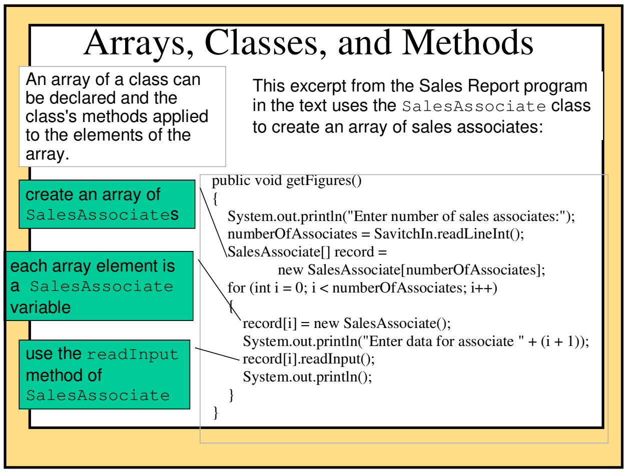
```
int i;//loop counter/array index
int[] a = new int[10];
for(i = 0; i < a.length; i++)
    a[i] = 0;
```

  - note that the loop counter/array index goes from `0` to `length - 1`
  - it counts through `length = 10` iterations/elements using the zero-numbering of the array index

Programming Tip:

Do not count on default initial values for array elements

- explicitly initialize elements in the declaration or in a loop



Arrays, Classes, and Methods

An array of a class can be declared and the class's methods applied to the elements of the array.

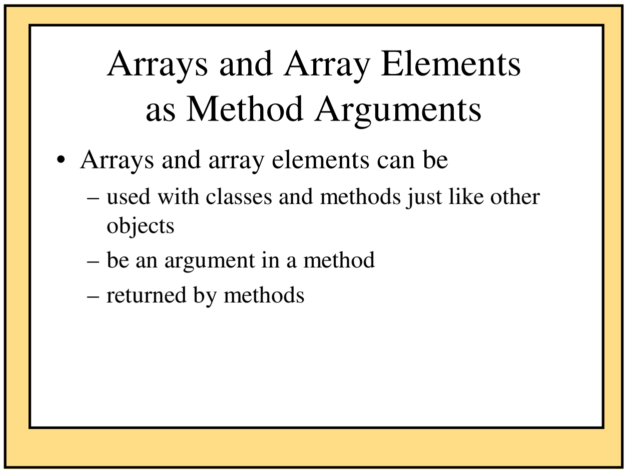
This excerpt from the Sales Report program in the text uses the `SalesAssociate` class to create an array of sales associates:

create an array of `SalesAssociate`s

each array element is a `SalesAssociate` variable

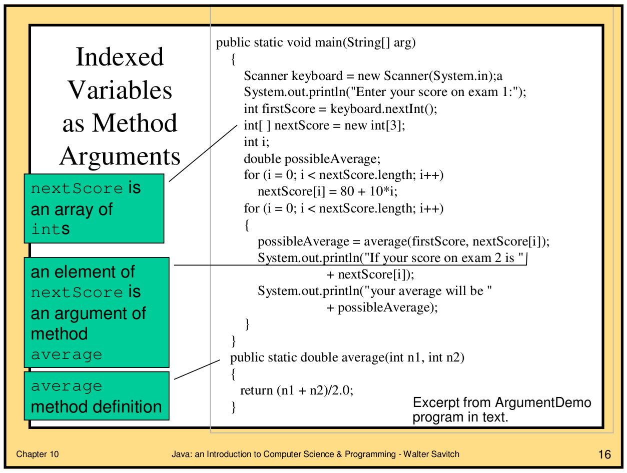
use the `readInput` method of `SalesAssociate`

```
public void getFigures()
{
    System.out.println("Enter number of sales associates:");
    numberOfAssociates = SavitchIn.readLineInt();
    SalesAssociate[] record =
              new SalesAssociate[numberOfAssociates];
    for (int i = 0; i < numberOfAssociates; i++)
    {
        record[i] = new SalesAssociate();
        System.out.println("Enter data for associate " + (i + 1));
        record[i].readInput();
        System.out.println();
    }
}
```



Arrays and Array Elements as Method Arguments

- Arrays and array elements can be
  - used with classes and methods just like other objects
  - be an argument in a method
  - returned by methods



Indexed Variables as Method Arguments

`nextScore` is an array of `int`s

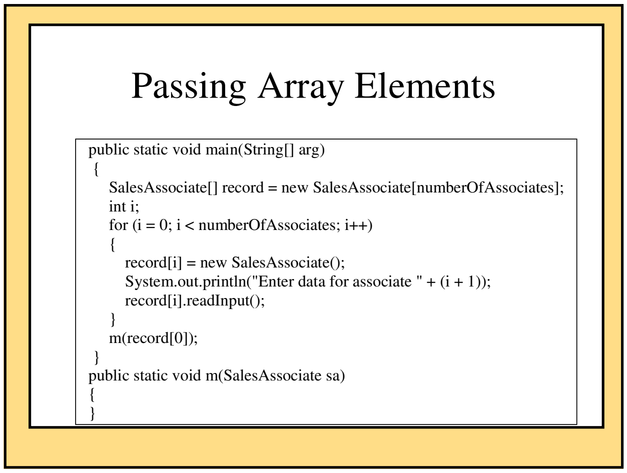
an element of `nextScore` is an argument of method `average`

`average` method definition

```
public static void main(String[] arg)
  {
    Scanner keyboard = new Scanner(System.in);a
    System.out.println("Enter your score on exam 1:");
    int firstScore = keyboard.nextInt();
    int[ ] nextScore = new int[3];
    int i;
    double possibleAverage;
    for (i = 0; i < nextScore.length; i++)
      nextScore[i] = 80 + 10*i;
    for (i = 0; i < nextScore.length; i++)
    {
      possibleAverage = average(firstScore, nextScore[i]);
      System.out.println("If your score on exam 2 is "
                    + nextScore[i]);
      System.out.println("your average will be "
                    + possibleAverage);
    }
  }
  public static double average(int n1, int n2)
  {
    return (n1 + n2)/2.0;
  }
```

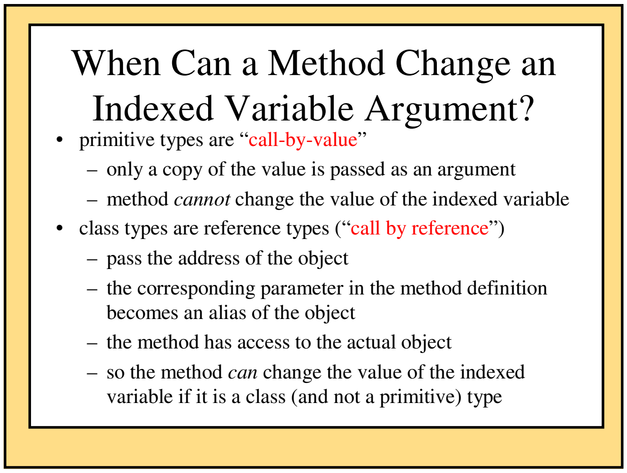
Excerpt from ArgumentDemo program in text.

Chapter 10 Java: an Introduction to Computer Science & Programming - Walter Savitch 16



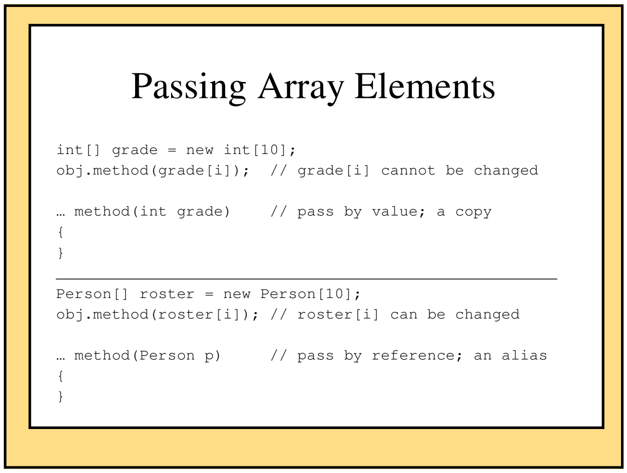
Passing Array Elements

```
public static void main(String[] arg)
 {
   SalesAssociate[] record = new SalesAssociate[numberOfAssociates];
   int i;
   for (i = 0; i < numberOfAssociates; i++)
   {
      record[i] = new SalesAssociate();
      System.out.println("Enter data for associate " + (i + 1));
      record[i].readInput();
   }
   m(record[0]);
 }
public static void m(SalesAssociate sa)
{
}
```



When Can a Method Change an Indexed Variable Argument?

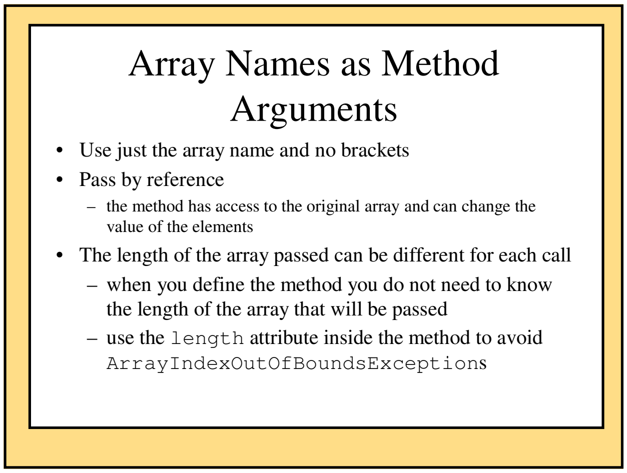
- primitive types are “call-by-value”
  - only a copy of the value is passed as an argument
  - method *cannot* change the value of the indexed variable
- class types are reference types (“call by reference”)
  - pass the address of the object
  - the corresponding parameter in the method definition becomes an alias of the object
  - the method has access to the actual object
  - so the method *can* change the value of the indexed variable if it is a class (and not a primitive) type



Passing Array Elements

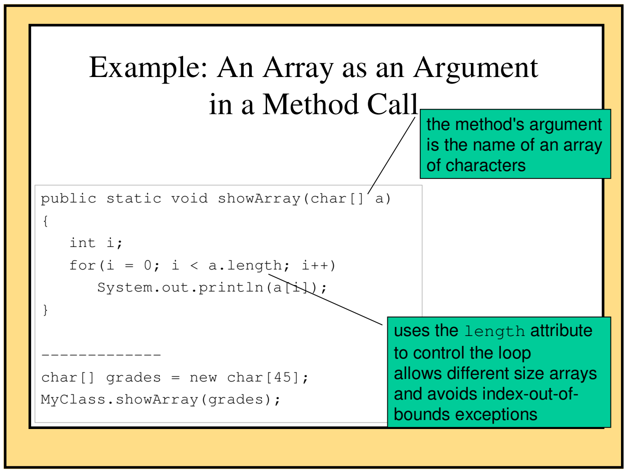
```
int[] grade = new int[10];
obj.method(grade[i]);  // grade[i] cannot be changed

… method(int grade)    // pass by value; a copy
{
}
```

```
Person[] roster = new Person[10];
obj.method(roster[i]); // roster[i] can be changed

… method(Person p)     // pass by reference; an alias
{
}
```



Array Names as Method Arguments

- Use just the array name and no brackets
- Pass by reference
  - the method has access to the original array and can change the value of the elements
- The length of the array passed can be different for each call
  - when you define the method you do not need to know the length of the array that will be passed
  - use the `length` attribute inside the method to avoid `ArrayIndexOutOfBoundsExceptions`

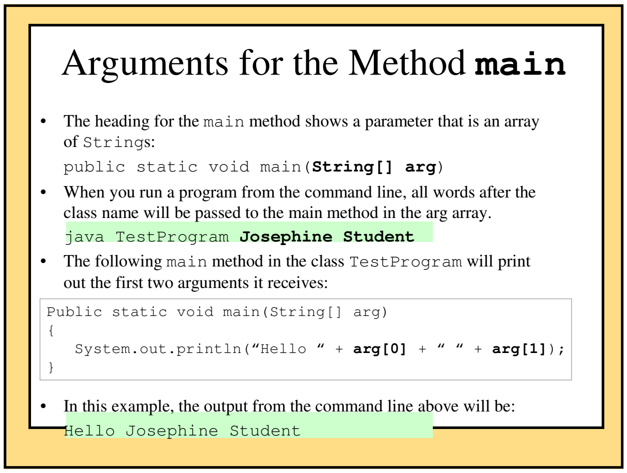


Example: An Array as an Argument in a Method Call

```
public static void showArray(char[] a)
{
   int i;
   for(i = 0; i < a.length; i++)
      System.out.println(a[i]);
}

-------------
char[] grades = new char[45];
MyClass.showArray(grades);
```

the method's argument is the name of an array of characters

uses the `length` attribute to control the loop allows different size arrays and avoids index-out-of-bounds exceptions



Arguments for the Method **`main`**

- The heading for the `main` method shows a parameter that is an array of `String`s:

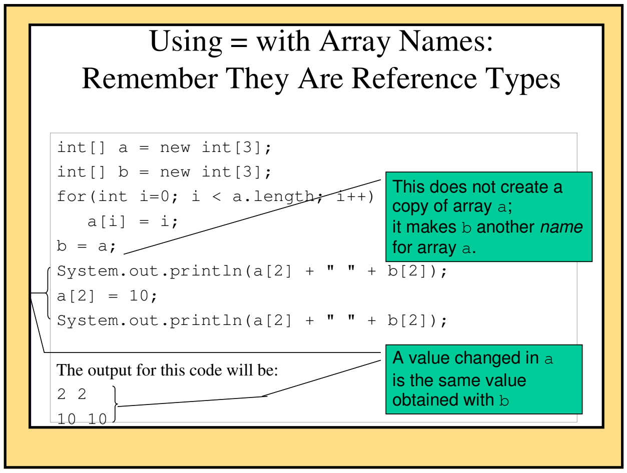
  `public static void main(`**`String[] arg`**`)`
- When you run a program from the command line, all words after the class name will be passed to the main method in the arg array.

  `java TestProgram `**`Josephine Student`**
- The following `main` method in the class `TestProgram` will print out the first two arguments it receives:

```
Public static void main(String[] arg)
{
   System.out.println(“Hello ” + arg[0] + “ ” + arg[1]);
}
```

- In this example, the output from the command line above will be:

  `Hello Josephine Student`



Using = with Array Names: Remember They Are Reference Types

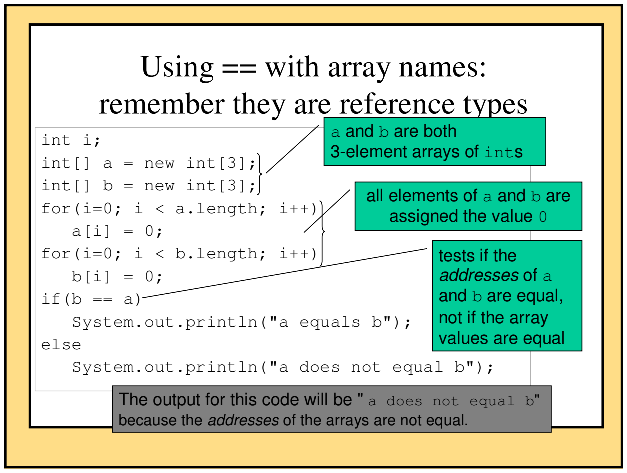
```
int[] a = new int[3];
int[] b = new int[3];
for(int i=0; i < a.length; i++)
   a[i] = i;
b = a;
System.out.println(a[2] + " " + b[2]);
a[2] = 10;
System.out.println(a[2] + " " + b[2]);
```

This does not create a copy of array a; it makes b another *name* for array a.

The output for this code will be:

```
2 2
10 10
```

A value changed in a is the same value obtained with b



Using == with array names: remember they are reference types

```
int i;
int[] a = new int[3];
int[] b = new int[3];
for(i=0; i < a.length; i++)
   a[i] = 0;
for(i=0; i < b.length; i++)
   b[i] = 0;
if(b == a)
   System.out.println("a equals b");
else
   System.out.println("a does not equal b");
```

a and b are both 3-element arrays of ints

all elements of a and b are assigned the value 0

tests if the *addresses* of a and b are equal, not if the array values are equal

The output for this code will be "a does not equal b" because the *addresses* of the arrays are not equal.

Behavior of Three Operations

	Primitive Type	Class Type	Entire Array	Array Element
Assignment (=)	Copy content	Copy address	Copy address	Depends on primitive/ class type
Equality (==)	Compare content	Compare address	Compare address	Depends on primitive/ class type
Parameter Passing	*Pass by value* (content)	*Pass by reference* (address)	*Pass by reference* (address)	Depends on primitive/ class type

Testing Two Arrays for Equality

- To test two arrays for equality you need to define an `equals` method that returns true if and only the arrays have the same length and all corresponding values are equal

```
public static boolean equals(int[] a,
                             int[] b)
   {
       boolean match = false;
       if (a.length == b.length)
       {
           match = true; //tentatively
           int i = 0;
           while (match && (i < a.length))
           {
               if (a[i] != b[i])
                   match = false;
               i++;
           }
       }
       return match;
    }
```

Methods that Return an Array

- the address of the array is passed
- The local array name within the method is just another name for the original array

```
public class returnArrayDemo
  {
      public static void main(String arg[])
      {
          char[] c;
          c = vowels();
          for(int i = 0; i < c.length; i++)
             System.out.println(c[i]);
       }
       public static char[] vowels()
       {
           char[] newArray = new char[5];
           newArray[0] = 'a';
           newArray[1] = 'e';
           newArray[2] = 'i';
           newArray[3] = 'o';
           newArray[4] = 'u';
           return newArray;
       }
    }
```

`c`, `newArray`, and the return type of `vowels` are all the same type: `char []`

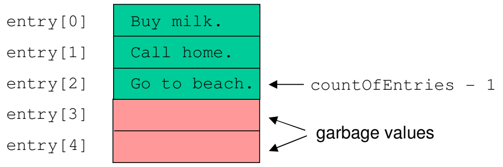
Wrapper Classes for Arrays

- Arrays can be made into objects by creating a wrapper class
  - similar to wrapper classes for primitive types
- In the wrapper class:
  - make an array an attribute
  - define constructors
  - define accessor methods to read and write element values and parameters
- The text shows an example of creating a wrapper class for an array of objects of type `OneWayNoRepeatsList`
  - the wrapper class defines two constructors plus the following methods: `addItem`, `full`, `empty`, `entryAt`, `atLastEntry`, `onList`, `maximumNumberOfEntries`, `numberOfEntries`, and `eraseList`

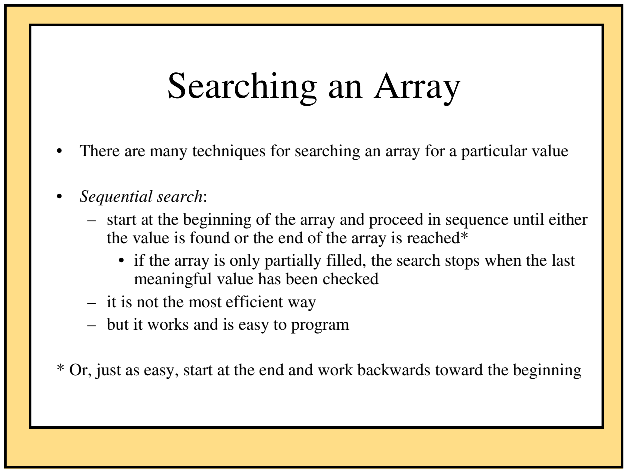
Partially Filled Arrays

- Sometimes only part of an array has been filled with data
- Array elements always contain something
  - elements which have not been written to
    - contain unknown (*garbage*) data so you should avoid reading them
- There is no automatic mechanism to detect how many elements have been filled
  - *you*, the programmer need to keep track!
- An example: the instance variable `countOfEntries` (in the class `OneWayNoRepeatsList`) is incremented every time `addItem` is called (see the text)

Example of a Partially Filled Array



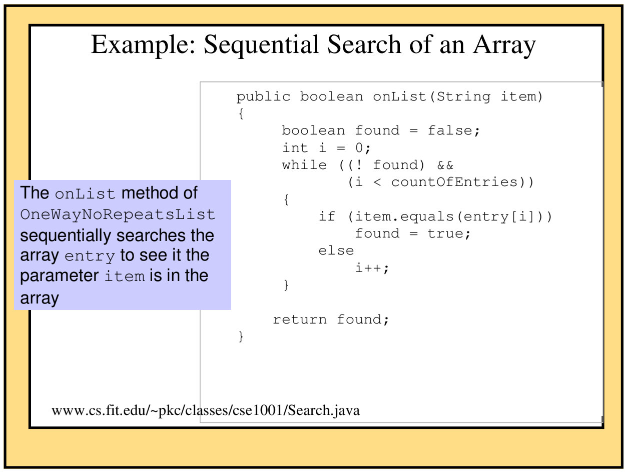
`countOfEntries` has a value of `3`.
`entry.length` has a value of `5`.



Searching an Array

- There are many techniques for searching an array for a particular value
- *Sequential search*:
  - – start at the beginning of the array and proceed in sequence until either the value is found or the end of the array is reached*
    - if the array is only partially filled, the search stops when the last meaningful value has been checked
  - – it is not the most efficient way
  - – but it works and is easy to program

* Or, just as easy, start at the end and work backwards toward the beginning

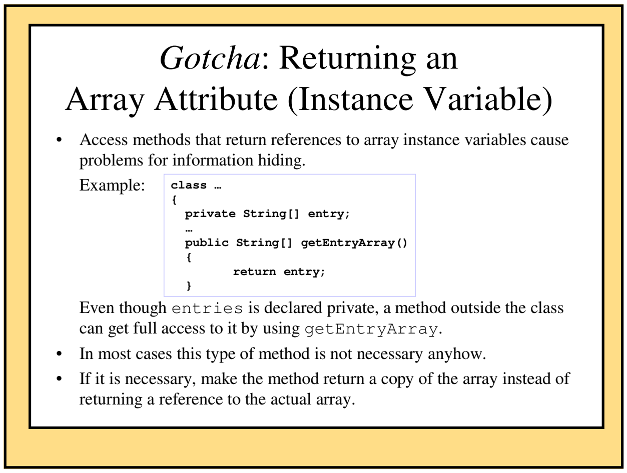


Example: Sequential Search of an Array

The `onList` method of `OneWayNoRepeatsList` sequentially searches the array `entry` to see it the parameter `item` is in the array

```
public boolean onList(String item)
{
     boolean found = false;
     int i = 0;
     while ((! found) &&
              (i < countOfEntries))
     {
          if (item.equals(entry[i]))
              found = true;
          else
              i++;
     }

     return found;
}
```

www.cs.fit.edu/~pkc/classes/cse1001/Search.java



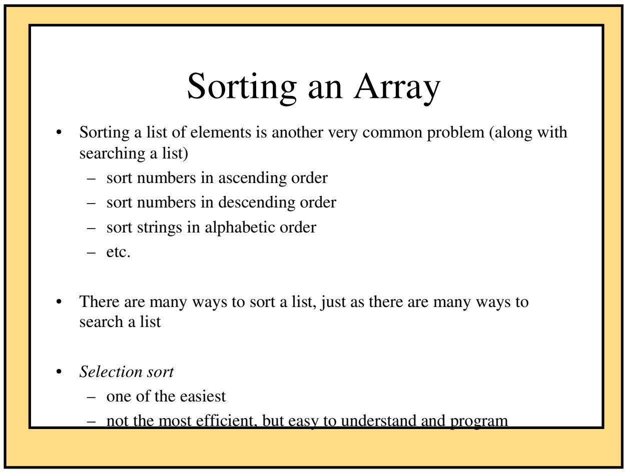
Gotcha: Returning an Array Attribute (Instance Variable)

- Access methods that return references to array instance variables cause problems for information hiding.

  Example:

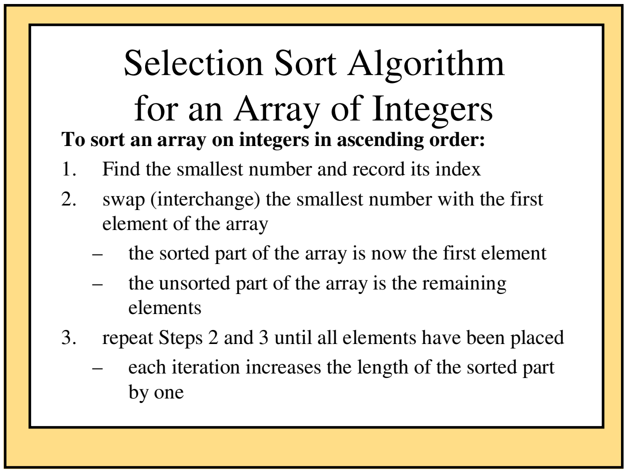
  ```
  class …
  {
    private String[] entry;
    …
    public String[] getEntryArray()
    {
          return entry;
    }
  ```

  Even though `entries` is declared private, a method outside the class can get full access to it by using `getEntryArray`.
- In most cases this type of method is not necessary anyhow.
- If it is necessary, make the method return a copy of the array instead of returning a reference to the actual array.



Sorting an Array

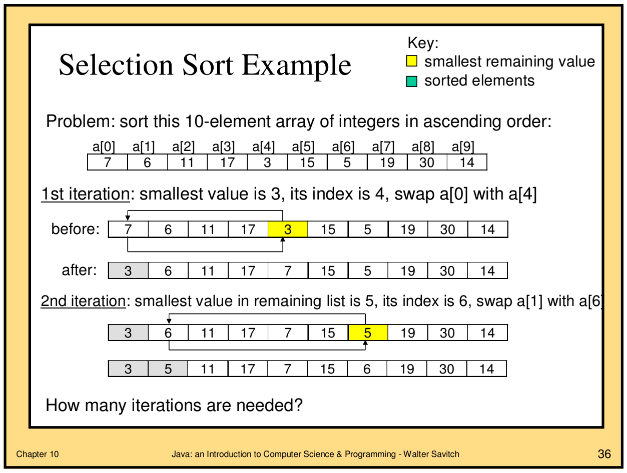
- Sorting a list of elements is another very common problem (along with searching a list)
  - – sort numbers in ascending order
  - – sort numbers in descending order
  - – sort strings in alphabetic order
  - – etc.
- There are many ways to sort a list, just as there are many ways to search a list
- *Selection sort*
  - – one of the easiest
  - – not the most efficient, but easy to understand and program



Selection Sort Algorithm for an Array of Integers

To sort an array on integers in ascending order:

1. Find the smallest number and record its index
2. swap (interchange) the smallest number with the first element of the array
   - – the sorted part of the array is now the first element
   - – the unsorted part of the array is the remaining elements
3. repeat Steps 2 and 3 until all elements have been placed
   - – each iteration increases the length of the sorted part by one



Selection Sort Example

Key:
🟨 smallest remaining value
🟩 sorted elements

Problem: sort this 10-element array of integers in ascending order:

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
7	6	11	17	3	15	5	19	30	14

1st iteration: smallest value is 3, its index is 4, swap a[0] with a[4]

before:	7	6	11	17	3	15	5	19	30	14
after:	3	6	11	17	7	15	5	19	30	14

2nd iteration: smallest value in remaining list is 5, its index is 6, swap a[1] with a[6

3	6	11	17	7	15	5	19	30	14
3	5	11	17	7	15	6	19	30	14

How many iterations are needed?

Example: Selection Sort

- Notice the precondition: every array element has a value
- may have duplicate values
- broken down into smaller tasks
  - "find the index of the smallest value"
  - "interchange two elements"
  - `private` because they are helper methods (users are not expected to call them directly)

Selection Sort Code

```
/*************************************
*Precondition:
*Every indexed variable of the array a has a value.
*Action: Sorts the array a so that
*a[0] <= a[1] <= ... <= a[a.length - 1].
**************************************************/
public static void sort(int[] a)
{
   int index, indexOfNextSmallest;
   for (index = 0; index < a.length - 1; index++)
   {//Place the correct value in a[index]:
       indexOfNextSmallest = indexOfSmallest(index, a);
       interchange(index,indexOfNextSmallest, a);
       //a[0] <= a[1] <=...<= a[index] and these are
       //the smallest of the original array elements.
       //The remaining positions contain the rest of
       //the original array elements.
   }
}
```

Insertion Sort

- Basic Idea:
  - Keeping expanding the sorted portion by one
  - Insert the next element into the right position in the sorted portion
- Algorithm:
  1. Start with one element [is it sorted?] – sorted portion
  2. While the sorted portion is not the entire array
     1. Find the right position in the sorted portion for the next element
     2. Insert the element
     3. If necessary, move the other elements down
     4. Expand the sorted portion by one

Insertion Sort: An example

- First iteration
  - Before: **[5]**, 3, 4, 9, 2
  - After: **[**3, **5]**, 4, 9, 2
- Second iteration
  - Before: **[3, 5]**, 4, 9, 2
  - After: **[3**, 4, **5]**, 9, 2
- Third iteration
  - Before: **[3, 4, 5]**, 9, 2
  - After: **[3, 4, 5**, 9**]**, 2
- Fourth iteration
  - Before: **[3, 4, 5, 9]**, 2
  - After: **[**2, **3, 4, 5, 9]**

Bubble Sort

- Basic Idea:
  - Expand the sorted portion one by one
  - "Sink" the largest element to the bottom after comparing adjacent elements
  - The smaller items "bubble" up
- Algorithm:
  - While the unsorted portion has more than one element
    - Compare adjacent elements
    - Swap elements if out of order
    - Largest element at the bottom, reduce the unsorted portion by one

Bubble Sort: An example

- First Iteration:
  - [5, 3], 4, 9, 2 → [3, 5], 4, 9, 2
  - 3, [5, 4], 9, 2 → 3, [4, 5], 9, 2
  - 3, 4, [5, 9], 2 → 3, 4, [5, 9], 2
  - 3, 4, 5, [9, 2] → 3, 4, 5, [2, 9]
- Second Iteration:
  - [3, 4], 5, 2, **9** → [3, 4], 5, 2, **9**
  - 3, [4, 5], 2, **9** → 3, [4, 5], 2, **9**
  - 3, 4, [5, 2], **9** → 3, 4, [2, 5], **9**
- Third Iteration:
  - [3, 4], 2, **5, 9** → [3, 4], 2, **5, 9**
  - 3, [4, 2], **5, 9** → 3, [2, 4], **5, 9**
- Fourth Iteration:
  - [3, 2], **4, 5, 9** → [2, 3], **4, 5, 9**

How to Compare Algorithms in Efficiency (speed)

- Empirical Analysis
  - Wall-clock time
  - CPU time
  - Can you predict performance before implementing the algorithm?
- Theoretical Analysis
  - Approximation by counting important operations
  - Mathematical functions based on input size (N)

How Fast/Slow Can It Get? (10G Hz, assume 10^{10} operations/sec)

N	$N\log_2 N$	N^2	2^N
10	33	100	1,024
100 (10^{-8} sec)	664	10,000	1.3 x 10^{30} (4 x10^{12} years)
1,000	9,966	1,000,000	Forever??
10,000	132,877	100,000,000	Eternity??

Theoretical Analysis (Sorting)

- Counting important operations
  - Comparisons (array elements)
    - >, <, …
  - Swaps/moves (array elements)
    - 1 swap has 3 moves
- Comparison is the more important operation—could be expensive
- Size of input (N) = Number of array elements
- Three cases for analysis
  - Worst case (interesting, popular analysis)
  - Best case (not so interesting)
  - Average case (discussed in another course)

Selection Sort

- Comparisons
  - $N - 1$ iterations
  - First iteration: how many comparisons?
  - Second iteration: how many comparisons?
  - $(N - 1) + (N - 2) + \ldots + 2 + 1 = N(N\text{-}1)/2 = (N^2 - N)/2$
- Moves (worst case: every element is in the wrong location)
  - $N - 1$ iterations
  - First iteration: how many swaps/moves?
  - Second iteration: how many swaps/moves?
  - $(N - 1)$ x $3 = 3N - 3$

Insertion Sort

- Comparisons (worst case: correct order)
  - $N - 1$ iterations
  - First iteration: how many comparisons?
  - Second iteration: how many comparisons?
  - $1 + 2 + \ldots + (N - 2) + (N - 1) = N(N\text{-}1)/2 = (N^2 - N)/2$
- Moves (worst case: reverse order)
  - $N - 1$ iterations
  - First iteration: how many moves?
  - Second iteration: how many moves?
  - $3 + 4 + \ldots + N + (N + 1) = (N + 4)(N - 1)/2 = (N^2 + 3N - 4)/2$

Bubble Sort

- Comparisons
  - $N - 1$ iterations
  - First iteration: how many comparisons?
  - Second iteration: how many comparisons?
  - $(N - 1) + (N - 2) + \ldots + 2 + 1 = N(N\text{-}1)/2 = (N^2 - N)/2$
- Moves (worst case: reverse order)
  - $N - 1$ iterations
  - First iteration: how many swaps/moves?
  - Second iteration: how many swaps/moves?
  - $[(N - 1) + (N - 2) + \ldots + 2 + 1]$ x $3 = 3N(N\text{-}1)/2 = (3N^2 - 3N)/2$

Summary of Worst-case Analysis

	Comparisons (more important)	Moves
Selection	$(N^2 – N)/2$	$3N - 3$
Insertion	$(N^2 – N)/2$	$(N^2 + 3N - 4)/2$
Bubble	$(N^2 – N)/2$	$(3N^2 – 3N)/2$

Sorting Algorithm Tradeoffs

- Easy to understand algorithms
  - not very efficient
  - less likely to have mistakes
  - require less time to code, test, and debug
  - Selection, Insertion, Bubble Sorting algorithms
  - Bubble Sort is the easiest to implement
- Complicated but more efficient
  - useful when performance is a major issue
  - programming project for Chapter 11 describes a more efficient sorting algorithm

"Getting the wrong result is always inefficient."

Multidimensional Arrays

- Arrays with more than one index
  - number of dimensions = number of indexes
- Arrays with more than two dimensions are a simple extension of two-dimensional (2-D) arrays
- A 2-D array corresponds to a table or grid
  - one dimension is the row
  - the other dimension is the column
  - cell: an intersection of a row and column
  - an array element corresponds to a cell in the table

Table as a 2-Dimensional Array

- The table assumes a starting balance of $1000
- First dimension: row identifier - Year
- Second dimension: column identifier - percentage
- Cell contains balance for the year (row) and percentage (column)
- Balance for year 4, rate 7.00% = $1311

Balances for Various Interest Rates Compounded Annually (Rounded to Whole Dollar Amounts)

Year	5.00%	5.50%	6.00%	6.50%	7.00%	7.50%
1	$1050	$1055	$1060	$1065	$1070	$1075
2	$1103	$1113	$1124	$1134	$1145	$1156
3	$1158	$1174	$1191	$1208	$1225	$1242
4	$1216	$1239	$1262	$1286	$1311	$1335
5	$1276	$1307	$1338	$1370	$1403	$1436
…	…	…	…	…	…	…

Table as a 2-D Array

Column Index 4
(5th column)

Indexes	0	1	2	3	4	5
0	$1050	$1055	$1060	$1065	$1070	$1075
1	$1103	$1113	$1124	$1134	$1145	$1156
2	$1158	$1174	$1191	$1208	$1225	$1242
3	$1216	$1239	$1262	$1286	$1311	$1335
4	$1276	$1307	$1338	$1370	$1403	$1436
…	…	…	…	…	…	…

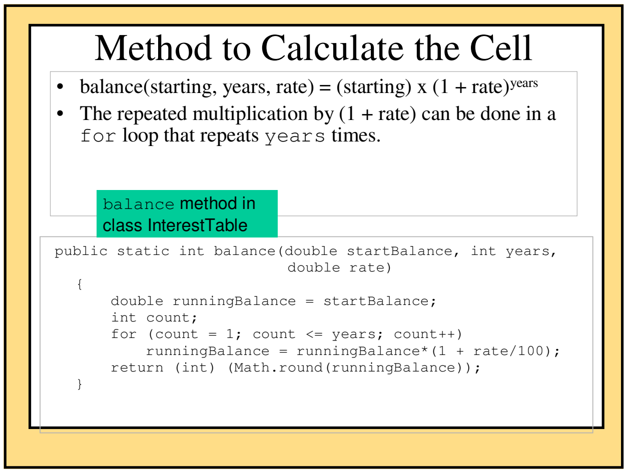
Row Index 3
(4th row)

- Generalizing to two indexes: [row][column]
- First dimension: row index
- Second dimension: column index
- Cell contains balance for the year/row and percentage/column
- All indexes use zero-numbering
  - Balance[3][4] = cell in 4th row (year = 4) and 5th column (7.50%)
  - Balance[3][4] = $1311 (shown in yellow)

Java Code to Create a 2-D Array

- Syntax for 2-D arrays is similar to 1-D arrays
- Declare a 2-D array of `int`s named `table`
  - the table should have ten rows and six columns

```
int[][] table = new int[10][6];
```

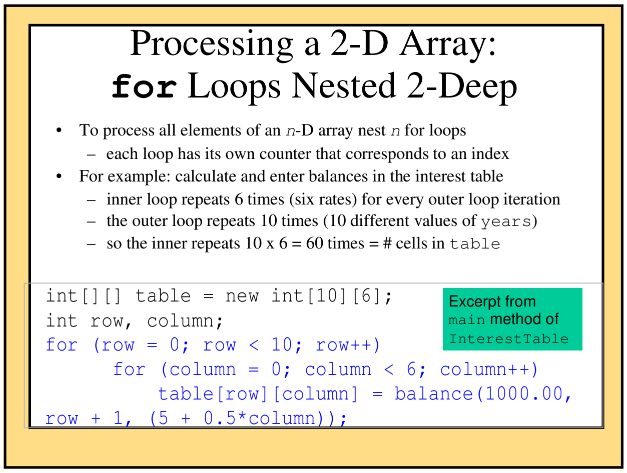


Method to Calculate the Cell

- balance(starting, years, rate) = (starting) x (1 + rate)^{years}
- The repeated multiplication by (1 + rate) can be done in a `for` loop that repeats `years` times.

`balance` method in class InterestTable

```
public static int balance(double startBalance, int years,
                                double rate)
   {
       double runningBalance = startBalance;
       int count;
       for (count = 1; count <= years; count++)
           runningBalance = runningBalance*(1 + rate/100);
       return (int) (Math.round(runningBalance));
   }
```

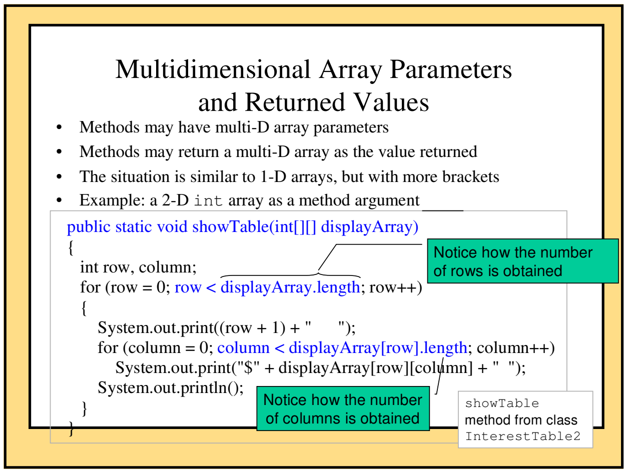


Processing a 2-D Array: `for` Loops Nested 2-Deep

- To process all elements of an *n*-D array nest *n* for loops
  - each loop has its own counter that corresponds to an index
- For example: calculate and enter balances in the interest table
  - inner loop repeats 6 times (six rates) for every outer loop iteration
  - the outer loop repeats 10 times (10 different values of `years`)
  - so the inner repeats 10 x 6 = 60 times = # cells in `table`

```
int[][] table = new int[10][6];
int row, column;
for (row = 0; row < 10; row++)
      for (column = 0; column < 6; column++)
          table[row][column] = balance(1000.00,
row + 1, (5 + 0.5*column));
```

Excerpt from `main` method of `InterestTable`



Multidimensional Array Parameters and Returned Values

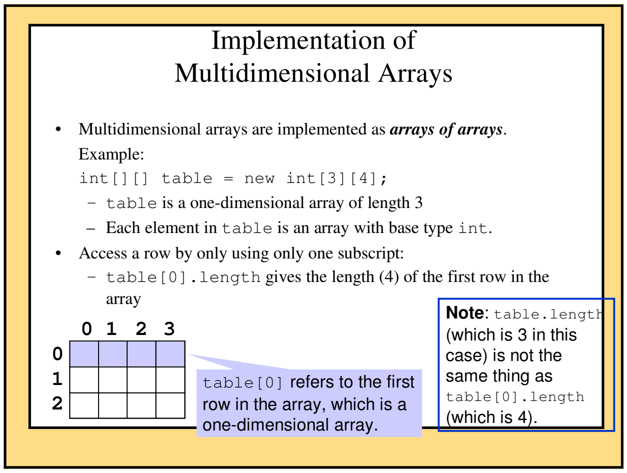
- Methods may have multi-D array parameters
- Methods may return a multi-D array as the value returned
- The situation is similar to 1-D arrays, but with more brackets
- Example: a 2-D `int` array as a method argument

```
public static void showTable(int[][] displayArray)
{
  int row, column;
  for (row = 0; row < displayArray.length; row++)
  {
     System.out.print((row + 1) + "      ");
     for (column = 0; column < displayArray[row].length; column++)
        System.out.print("$" + displayArray[row][column] + "  ");
     System.out.println();
  }
}
```

Notice how the number of rows is obtained

Notice how the number of columns is obtained

`showTable` method from class `InterestTable2`



Implementation of Multidimensional Arrays

- Multidimensional arrays are implemented as ***arrays of arrays***.

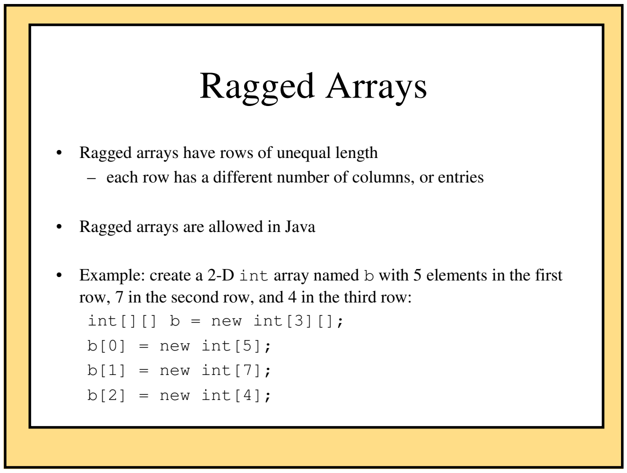
  Example:

  `int[][] table = new int[3][4];`
  - `table` is a one-dimensional array of length 3
  - Each element in `table` is an array with base type `int`.
- Access a row by only using only one subscript:
  - `table[0].length` gives the length (4) of the first row in the array

	0	1	2	3
0				
1				
2				

`table[0]` refers to the first row in the array, which is a one-dimensional array.

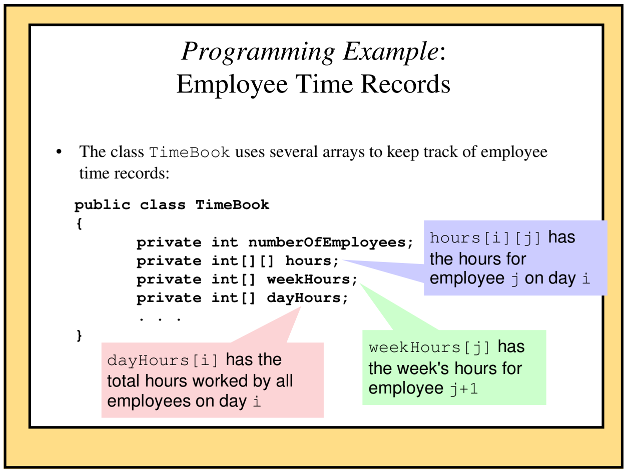
Note: `table.length` (which is 3 in this case) is not the same thing as `table[0].length` (which is 4).



Ragged Arrays

- Ragged arrays have rows of unequal length
  - each row has a different number of columns, or entries
- Ragged arrays are allowed in Java
- Example: create a 2-D `int` array named `b` with 5 elements in the first row, 7 in the second row, and 4 in the third row:

```
int[][] b = new int[3][];
b[0] = new int[5];
b[1] = new int[7];
b[2] = new int[4];
```



Programming Example: Employee Time Records

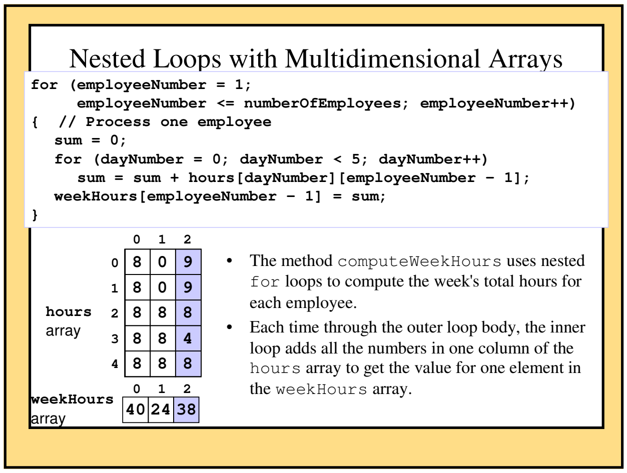
- The class `TimeBook` uses several arrays to keep track of employee time records:

```
public class TimeBook
{
        private int numberOfEmployees;
        private int[][] hours;
        private int[] weekHours;
        private int[] dayHours;
        . . .
}
```

`hours[i][j]` has the hours for employee `j` on day `i`

`weekHours[j]` has the week's hours for employee `j+1`

`dayHours[i]` has the total hours worked by all employees on day `i`



Nested Loops with Multidimensional Arrays

```
for (employeeNumber = 1;
      employeeNumber <= numberOfEmployees; employeeNumber++)
{  // Process one employee
   sum = 0;
   for (dayNumber = 0; dayNumber < 5; dayNumber++)
      sum = sum + hours[dayNumber][employeeNumber - 1];
   weekHours[employeeNumber - 1] = sum;
}
```

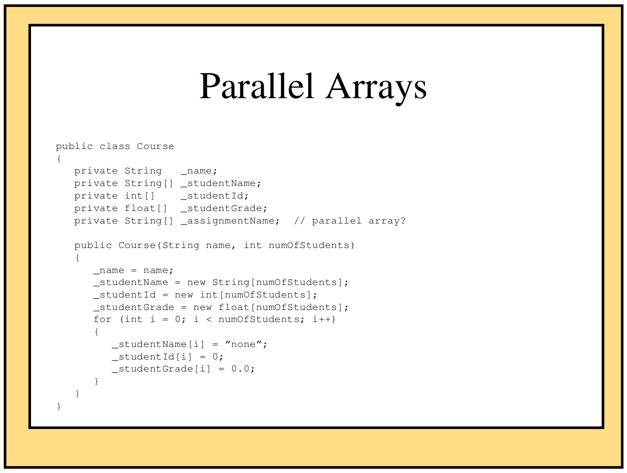
hours array

	0	1	2
0	8	0	9
1	8	0	9
2	8	8	8
3	8	8	4
4	8	8	8

weekHours array

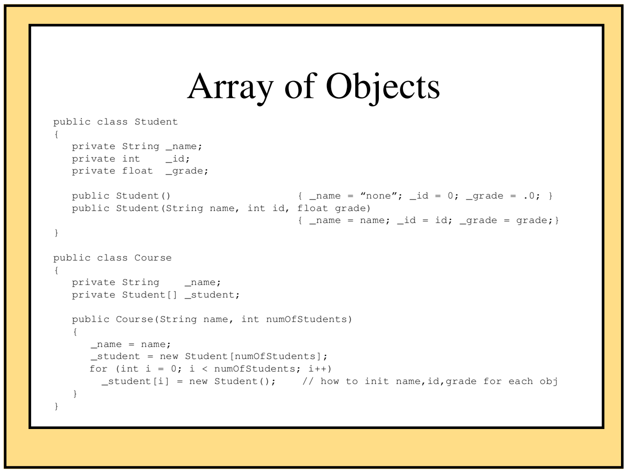
0	1	2
40	24	38

- The method `computeWeekHours` uses nested `for` loops to compute the week's total hours for each employee.
- Each time through the outer loop body, the inner loop adds all the numbers in one column of the `hours` array to get the value for one element in the `weekHours` array.



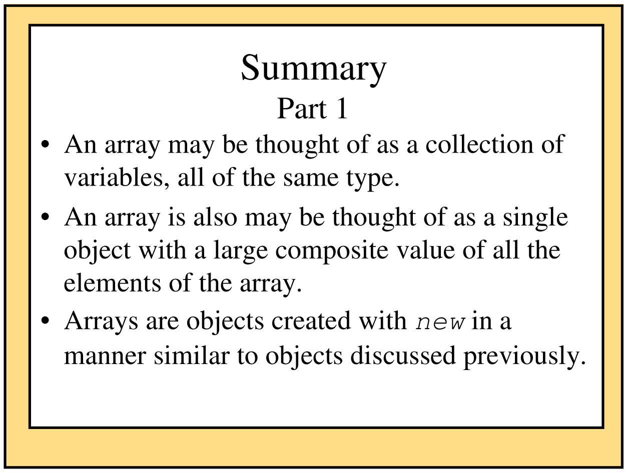
Parallel Arrays

```
public class Course
{
   private String   _name;
   private String[] _studentName;
   private int[]    _studentId;
   private float[]  _studentGrade;
   private String[] _assignmentName;  // parallel array?

   public Course(String name, int numOfStudents)
   {
      _name = name;
      _studentName = new String[numOfStudents];
      _studentId = new int[numOfStudents];
      _studentGrade = new float[numOfStudents];
      for (int i = 0; i < numOfStudents; i++)
      {
         _studentName[i] = “none”;
         _studentId[i] = 0;
         _studentGrade[i] = 0.0;
      }
   }
}
```



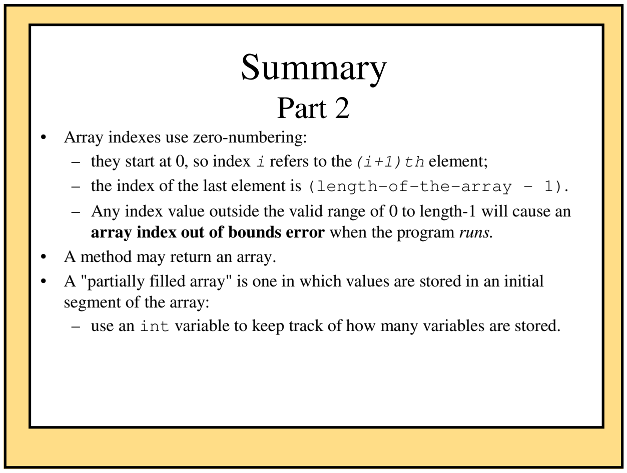
Array of Objects

```
public class Student
{
   private String _name;
   private int    _id;
   private float  _grade;

   public Student()                     { _name = “none”; _id = 0; _grade = .0; }
   public Student(String name, int id, float grade)
                                        { _name = name; _id = id; _grade = grade;}
}

public class Course
{
   private String    _name;
   private Student[] _student;

   public Course(String name, int numOfStudents)
   {
      _name = name;
      _student = new Student[numOfStudents];
      for (int i = 0; i < numOfStudents; i++)
        _student[i] = new Student();    // how to init name,id,grade for each obj
   }
}
```



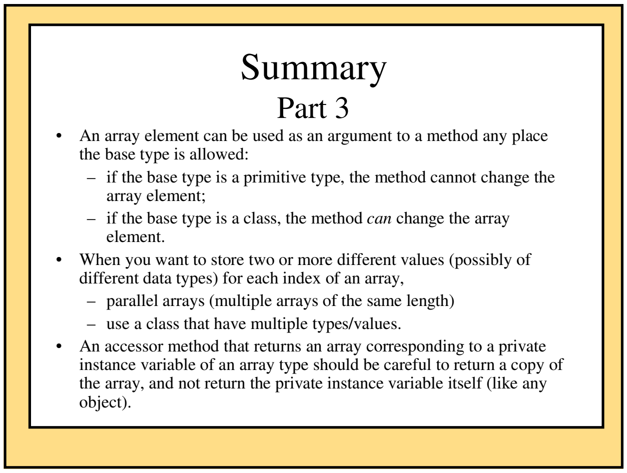
Summary
Part 1

- An array may be thought of as a collection of variables, all of the same type.
- An array is also may be thought of as a single object with a large composite value of all the elements of the array.
- Arrays are objects created with *`new`* in a manner similar to objects discussed previously.



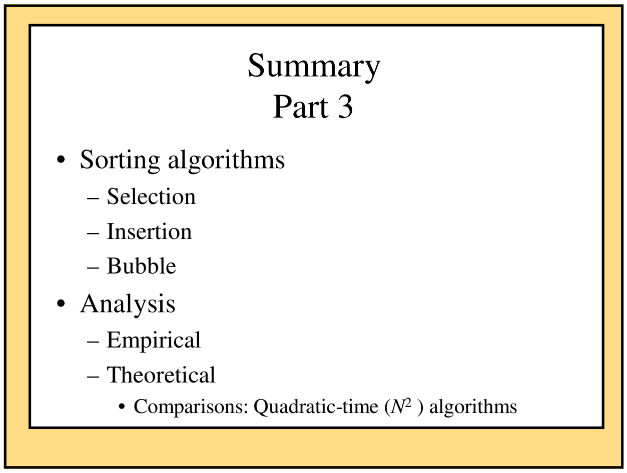
Summary
Part 2

- Array indexes use zero-numbering:
  - they start at 0, so index *`i`* refers to the *`(i+1)th`* element;
  - the index of the last element is `(length-of-the-array - 1)`.
  - Any index value outside the valid range of 0 to length-1 will cause an **array index out of bounds error** when the program *runs.*
- A method may return an array.
- A "partially filled array" is one in which values are stored in an initial segment of the array:
  - use an `int` variable to keep track of how many variables are stored.



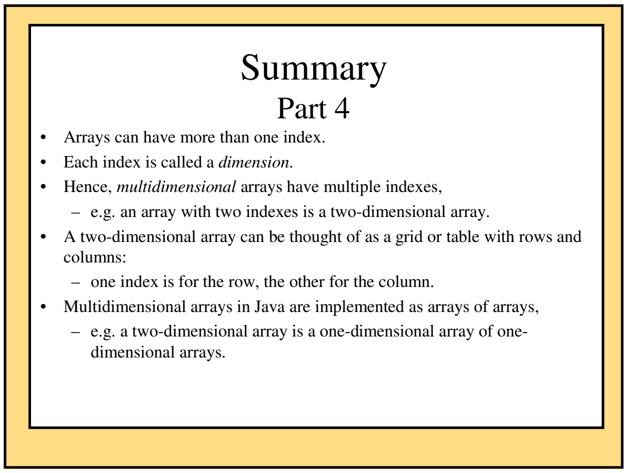
Summary
Part 3

- An array element can be used as an argument to a method any place the base type is allowed:
  - if the base type is a primitive type, the method cannot change the array element;
  - if the base type is a class, the method *can* change the array element.
- When you want to store two or more different values (possibly of different data types) for each index of an array,
  - parallel arrays (multiple arrays of the same length)
  - use a class that have multiple types/values.
- An accessor method that returns an array corresponding to a private instance variable of an array type should be careful to return a copy of the array, and not return the private instance variable itself (like any object).



Summary
Part 3

- Sorting algorithms
  - Selection
  - Insertion
  - Bubble
- Analysis
  - Empirical
  - Theoretical
    - Comparisons: Quadratic-time (N^2) algorithms



Summary
Part 4

- Arrays can have more than one index.
- Each index is called a *dimension*.
- Hence, *multidimensional* arrays have multiple indexes,
  - e.g. an array with two indexes is a two-dimensional array.
- A two-dimensional array can be thought of as a grid or table with rows and columns:
  - one index is for the row, the other for the column.
- Multidimensional arrays in Java are implemented as arrays of arrays,
  - e.g. a two-dimensional array is a one-dimensional array of one-dimensional arrays.