Programming and Software Development
COMP90041

Lecture 6

Arrays

NOTE: Some of the Material in these slides are adopted from
* Lectures Notes prepared by Dr. Peter Schachte and
* the Textbook resources

Udaya Parampalli & Thuan Pham
Semester 1 2021, 12-04-2021

- Static methods and static variables
  - The Math class and wrapper classes
  - Automatic boxing and unboxing mechanism

- References and class parameters
  - Variables and Memory
  - Using and misusing references

Review: Week 5

- Introduction to arrays
  - Creating and accessing arrays
  - Use for loops with arrays
- Arrays and references
- Programming with arrays
  - Sorting
- ArrayList
- Multidimensional arrays

Outline

- **Introduction to arrays**
  - **Creating and accessing arrays**
  - **Use for loops with arrays**
- Arrays and references
- Programming with arrays
  - Sorting
- ArrayList
- Multidimensional arrays

Outline

```java
import java.util.Scanner;

public class Scores {
    public static void main(String[] args) {
        Scanner keyboard = new Scanner(System.in);

        double var1, var2, var3, var4, var5, max;

        System.out.println("Enter 5 scores:");
        var1 = keyboard.nextDouble();
        max = var1;
        var2 = keyboard.nextDouble();
        max = (var2 > max)?var2:max;
        var3 = keyboard.nextDouble();
        max = (var3 > max)?var3:max;
        var4 = keyboard.nextDouble();
        max = (var4 > max)?var4:max;
        var5 = keyboard.nextDouble();
        max = (var5 > max)?var5:max;

        System.out.println("The highest score is " + max);
        System.out.println("The scores are:");

        System.out.println("Score 1 differs from max by " + (max - var1));
        System.out.println("Score 2 differs from max by " + (max - var2));
        System.out.println("Score 3 differs from max by " + (max - var3));
        System.out.println("Score 4 differs from max by " + (max - var4));
        System.out.println("Score 5 differs from max by " + (max - var5));
    }
}
```

Why do we need arrays?

- An ***array*** is a data structure used to process a collection of data that is all of the **same type**
  - An array behaves like a numbered list of variables with a uniform naming mechanism
  - It has a part that does not change: the name of the array
  - It has a part that can change: an integer in square brackets
  - For example, given five scores:

score[0], score[1], score[2], score[3], score[4]

Introduction to Arrays

- An array that behaves like this collection of variables, all of type **`double`**, can be created using one statement as follows:

```
double[] score = new double[5];
```

- Or using two statements:

```
double[] score;
score = new double[5];
```

  - The first statement declares the variable **`score`** to be of the array type **`double[]`**
  - The second statement creates an array with five numbered variables of type **`double`** and makes the variable **`score`** a name for the array

Creating and Accessing Arrays

- The individual variables that together make up the array are called *indexed variables*
  - They can also be called ***subscripted variables*** or ***elements*** of the array
  - The number in square brackets is called an ***index*** or ***subscript***
  - In Java, *indices must be numbered starting with **0**, and nothing else*

score[0], score[1], score[2], score[3], score[4]

Creating and Accessing Arrays

- The number of indexed variables in an array is called the ***length*** or ***size*** of the array
- When an array is created, the length of the array is given in square brackets after the array type
- The indexed variables are then numbered starting with **0**, and ending with the integer that is *one less than the length of the array*

score[0], score[1], score[2], score[3], score[4]

Creating and Accessing Arrays

double[] score = new double[5];

- A variable may be used in place of the integer index (i.e., in place of the integer ***5*** above)
  - The value of this variable can then be read from the keyboard
  - This enables the size of the array to be determined when the program is run

double[] score = new double[count];

- An array can have indexed variables of any type, including any class type
- All of the indexed variables in a single array must be of the same type, called the ***base type*** of the array

Creating and Accessing Arrays

Declaring and Creating an Array

- An array is declared and created in almost the same way that objects are declared and created:

```
BaseType[] ArrayName = new BaseType[size];
```

  - The `size` may be given as an expression that evaluates to a nonnegative integer, for example, an `int` variable

```
char[] line = new char[80];
double[] reading = new double[count];
Person[] specimen = new Person[100];
```

Example: ArrayOfScores.java

```java
import java.util.Scanner;

public class ArrayOfScores
{
  /**
  Reads in 5 scores and shows how much each
  score differs from the highest score.
  */
  public static void main(String[] args)
  {
    Scanner keyboard = new Scanner(System.in);
    double[] score = new double[5];
    int index;
    double max;

    System.out.println("Enter 5 scores:");
    score[0] = keyboard.nextDouble();
    max = score[0];
    for (index = 1; index < 5; index++)
    {
      score[index] = keyboard.nextDouble();
      if (score[index] > max)
        max = score[index];
      //max is the largest of the values score[0], ..., score[index]
    }

    System.out.println("The highest score is " + max);
    System.out.println("The scores are:");
    for (index = 0; index < 5; index++)
      System.out.println(score[index] +
                         " differs from max by " + (max - score[index]));
  }
}
```

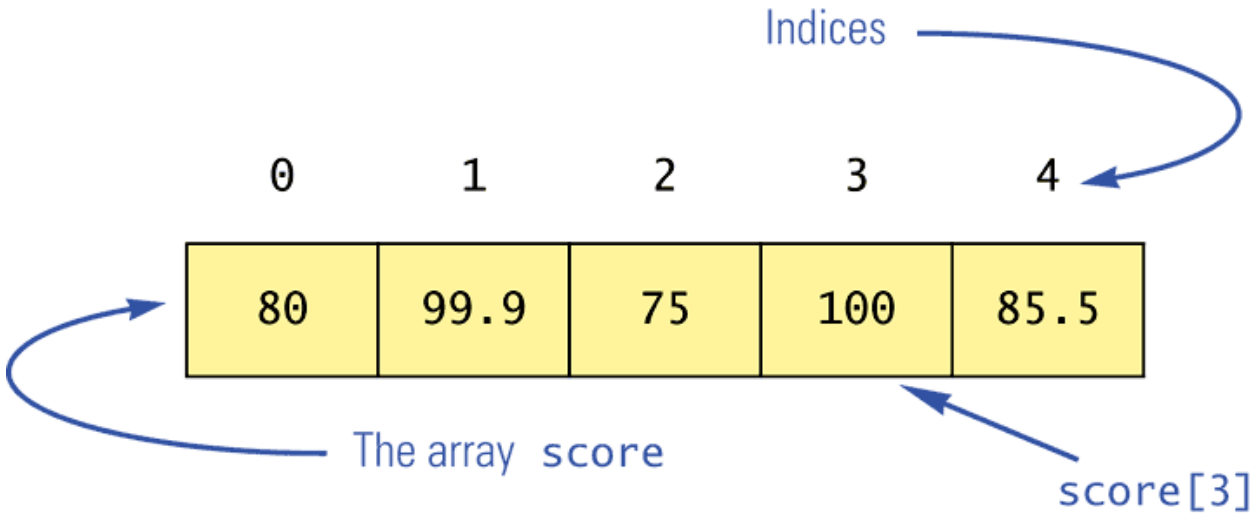
Example: ArrayOfScores.java

- Each array element can be used ***just like any other single variable*** by referring to it using an indexed expression: `score[0]`
- The array itself (i.e., the entire collection of indexed variables) can be referred to using the array name (without any square brackets): `score`
- An array index can be computed when a program is run
  - It may be represented by a variable: `score[index]`
  - It may be represented by an expression that evaluates to a suitable integer: `score[next + 1]`

Referring to Arrays and Array Elements

- The **`for`** loop is ideally suited for performing array manipulations:

```
for  (index = 0; index < 5; index++)
System.out.println(score[index] +
   " differs from max by " +
   (max-score[index]) );
```



Using the score Array in a Program

- Square brackets can be used to create a type name:
  double[] score;
- Square brackets can be used with an integer value as part of the special syntax Java uses to create a new array:
  score = new double[5];
- Square brackets can be used to name an indexed variable of an array:
  max = score[0];

3 ways to use square brackets []

The length Instance Variable

- An array is considered to be an object
- Since other objects can have instance variables, so can arrays
- Every array has exactly one instance variable named ***length***
  - When an array is created, the instance variable **length** is automatically set equal to its size
  - The value of **length** cannot be changed (other than by creating an entirely new array with **new**)

    double[] score = new double[5];
  - Given **score** above, **score.length** has a value of 5

- Array indices always start with **0**, and always end with the integer that is one less than the size of the array
  - The most common programming error made when using arrays is attempting to use a nonexistent array index
- When an index expression evaluates to some value other than those allowed by the array declaration, the index is said to be *out of bounds*
  - An out of bounds index will cause a program to terminate with a ***run-time error message***
  - Array indices get out of bounds most commonly at the *first* or *last* iteration of a loop that processes the array: Be sure to test for this! (off-by-one error)

Pitfall: Array Index Out of Bounds

Initializing Arrays

- An array can be initialized when it is declared
  - Values for the indexed variables are enclosed in braces, and separated by commas
  - The array size is automatically set to the number of values in the braces

```
int[] age = {2, 12, 1};//no need to new
```

  - Given `age` above, `age.length` has a value of 3

- Another way of initializing an array is by using a **`for`** loop

```
double[] reading = new double[100];
int index;
for (index = 0; index < reading.length; index++)
  reading[index] = 42.0;
```

- If the elements of an array are not initialized explicitly, they will ***automatically be initialized*** to the default value for their base type

Initializing Arrays

- Introduction to arrays
  - Creating and accessing arrays
  - Use for loops with arrays
- **Arrays and references**
- Programming with arrays
  - Sorting
- ArrayList
- Multidimensional arrays

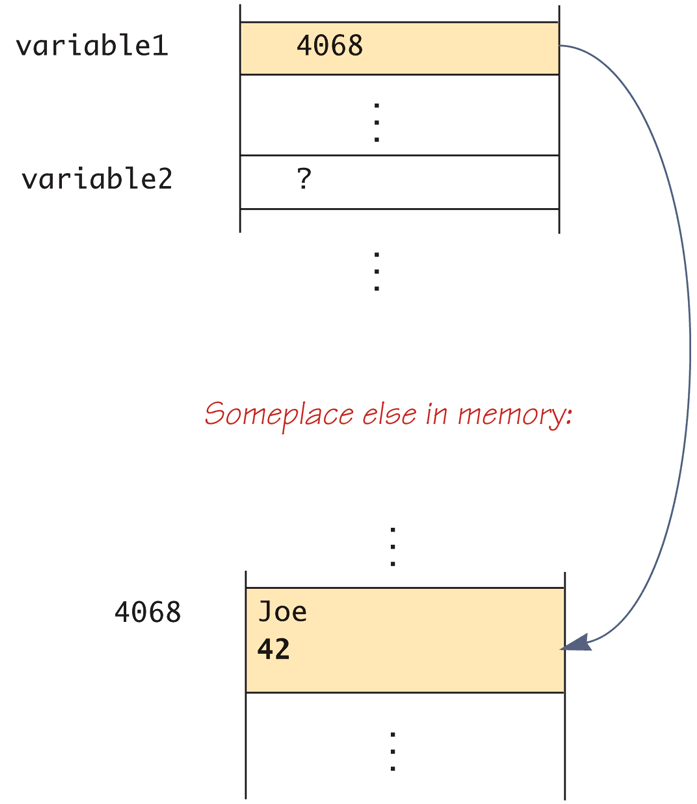
Outline

- Like class types, a variable of an array type holds a ***reference***
  - Arrays are objects
  - A variable of an array type holds the address of where the array object is stored in memory
  - Array types are (usually) considered to be class types

Arrays and References

Display 5.13 Assignment Operator with Class Type Variables

```
ToyClass variable1 = new ToyClass("Joe", 42);
ToyClass variable2;
```



We do not know what memory address (reference) is stored in the variable `variable1`. *Let's say it is* `4068`. *The exact number does not matter.*

Note that you can think of

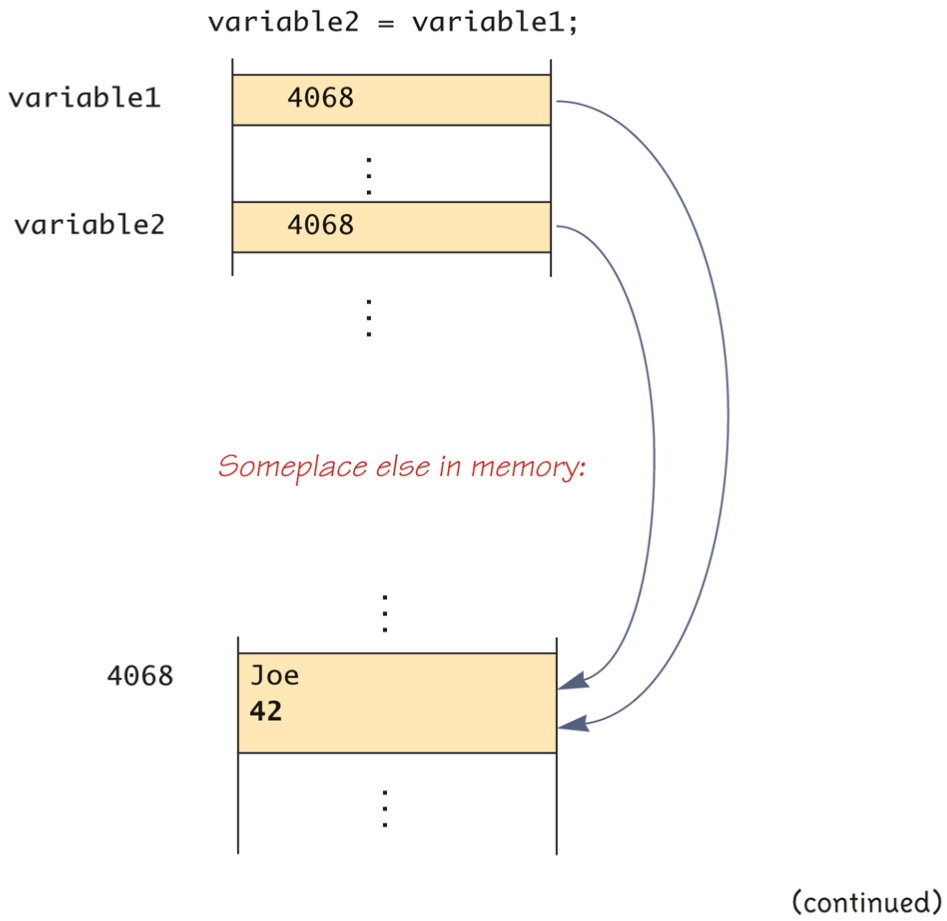
```
new ToyClass("Joe", 42)
```

as returning a reference.

(continued)

Assignment Operator with Class Type Variables (Part 1 of 3)

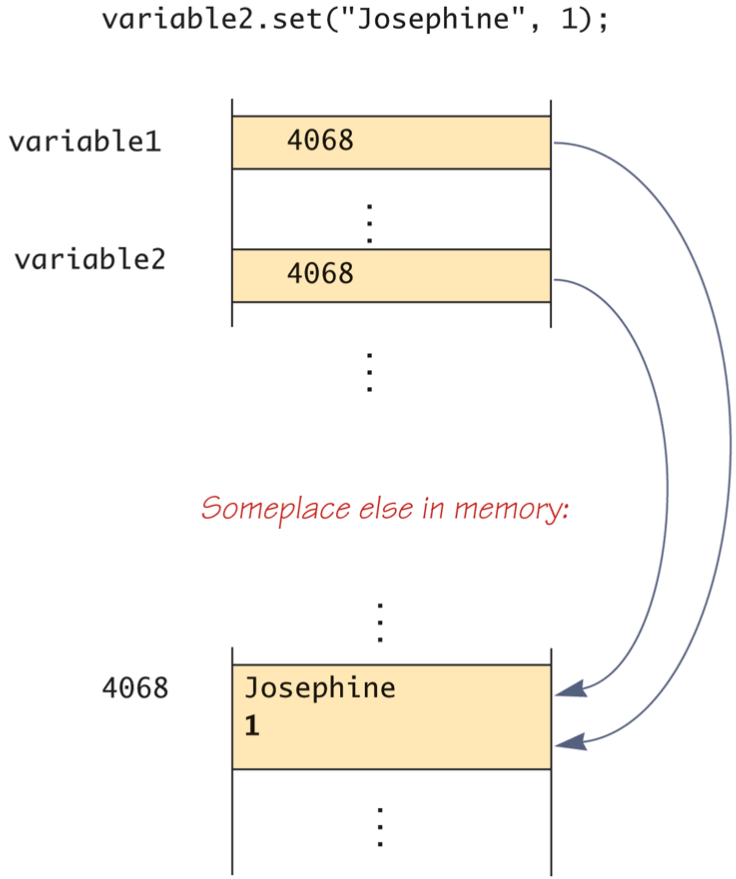
Display 5.13 Assignment Operator with Class Type Variables



(continued)

Assignment Operator with Class Type Variables (Part 2 of 3)

Display 5.13 **Assignment Operator with Class Type Variables**



Assignment Operator with Class Type Variables (Part 3 of 3)

Arrays are Objects

- An array can be viewed as a collection of indexed variables
- An array can also be viewed as a single item whose value is a collection of values of a base type
  - An array variable names the array as a single item
    double[] a;
  - A **new** expression creates an array object and stores the object in memory
    new double[10]
  - An assignment statement places a reference to the memory address of an array object in the array variable
    a = new double[10];

- The previous steps can be combined into one statement
  `double[] a = new double[10];`
- Note that the **`new`** expression that creates an array invokes a constructor that uses a nonstandard syntax
- Note also that as a result of the assignment statement above, **`a`** contains a single value: a memory address or *reference*
- Since an array is a reference type, the behavior of arrays with respect to assignment (**`=`**), equality testing (**`==`**), and parameter passing are the same as that described for classes

Arrays are Objects

- The base type of an array can be a class type
  `Date[] holidayList = new Date[20];`
- The above example creates 20 indexed variables of type **`Date`**
  - It does **NOT** create 20 objects of the class **`Date`**
  - Each of these indexed variables are automatically initialized to **`null`**
  - Any attempt to reference any them at this point would result in a "null pointer exception" error message

Pitfall: Arrays with a Class Base Type

- Like any other object, each of the indexed variables requires a separate invocation of a constructor using **`new`** (singly, or perhaps using a **`for`** loop) to create an object to reference

```
holidayList[0] = new Date();
        . . .
holidayList[19] = new Date();
```

OR

```
for (int i = 0; i < holidayList.length; i++)
  holidayList[i] = new Date();
```

- Each of the indexed variables can now be referenced since each holds the memory address of a **`Date`** object

Pitfall: Arrays with a Class Base Type

Intentionally blank

- Both ***array indexed variables*** and ***entire arrays*** can be used as arguments to methods
  - An indexed variable can be an argument to a method in exactly the same way that any variable of the array base type can be an argument

Array Parameters

```
double n = 0.0;
double[] a = new double[10];//all elements
            //are initialized to 0.0
int i = 3;
```

- Given `myMethod` which takes one argument of type `double`, then all of the following are legal:

```
myMethod(n);//n evaluates to 0.0
myMethod(a[3]);//a[3] evaluates to 0.0
myMethod(a[i]);//i evaluates to 3,
            //a[3] evaluates to 0.0
void myMethod(double a){
        a= a+1;
}
```

In which case can we change the value of the argument?

Array Parameters

- An argument to a method may be an entire array
- Array arguments behave like objects of a class
  - Therefore, a method can change the values stored in the indexed variables of an array argument
- A method with an array parameter must specify the base type of the array only

  BaseType[]
  - It does not specify the length of the array

Array Parameters

- The following method, **`doubleElements`**, specifies an array of **`double`** as its single argument:

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

Array Parameters

- Arrays of double may be defined as follows:
  double[] a = new double[10];
  double[] b = new double[30];
- Given the arrays above, the method **`doubleElements`** from class **`SampleClass`** can be invoked as follows:
  SampleClass.doubleElements(a);
  SampleClass.doubleElements(b);
  - Note that no square brackets are used when an entire array is given as an argument
  - Note also that a method that specifies an array for a parameter can take an ***array of any length*** as an argument

Array Parameters

- Because an array variable contains the memory address of the array it names, the assignment operator (**=**) only copies this memory address
  - It does not copy the values of each indexed variable
  - Using the assignment operator will make two array variables be different names for the same array

    b = a;
  - The memory address in **a** is now the same as the memory address in **b**: They reference the same array

Pitfall: Use of = and == with Arrays

- For the same reason, the equality operator **(==)** only tests two arrays to see if they are stored in the same location in the computer's memory
  - It does not test two arrays to see if they contain the same values
    (a == b)
  - The result of the above **`boolean`** expression will be **`true`** if **`a`** and **`b`** share the same memory address (and, therefore, reference the same array), and **`false`** otherwise

Pitfall: Use of = and == with Arrays

- In the same way that an **`equals`** method can be defined for a class, an ***`equalsArray`*** method can be defined for a type of array
  - This is how two arrays must be tested to see if they contain the same elements
  - The following method tests two integer arrays to see if they contain the same integer values

Pitfall: Use of = and == with Arrays

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

Pitfall: Use of = and == with Arrays

- The heading for the **`main`** method of a program has a parameter for an array of **`String`**
  - It is usually called ***`args`*** by convention
    public static void main(String[] *`args`*)
  - Note that since ***`args`*** is a parameter, it could be replaced by any other non-keyword identifier
- If a Java program is run without giving an argument to **`main`**, then a default empty array of strings is automatically provided

Arguments for the Method main

- Here is a program that expects three string arguments:

```
public class SomeProgram
{
  public static void main(String[] args)
  {
    System.out.println(args[0] + " " +
                       args[2] + args[1]);
  }
}
```

- Note that if it needed numbers, it would have to convert them from strings first

Arguments for the Method main

Arguments for the Method main

- If a program requires that the **`main`** method be provided an array of strings argument, each element must be provided from the command line when the program is run

  java SomeProgram Hi ! there

  - This will set **`args[0]`** to "Hi", **`args[1]`** to "!", and **`args[2]`** to "there"
  - It will also set **`args.length`** to 3
- When **`SomeProgram`** is run as shown, its output will be:

  Hi there!

- In Java, a method may also return an array
  - The return type is specified in the same way that an array parameter is specified

```
public static int[]  incrementArray(int[] a, int increment)
{
  int[] temp = new int[a.length];
  int i;
  for (i = 0; i < a.length; i++) {
    temp[i] = a[i] + increment;
   }
  return temp;
}
```

Methods that return an array

- Introduction to arrays
  - Creating and accessing arrays
  - Use for loops with arrays
- Arrays and references
- **Programming with arrays**
  - **Sorting**
- ArrayList
- Multidimensional arrays

Outline

- The exact size needed for an array is not always known when a program is written, or it may vary from one run of the program to another
- A common way to handle this is to declare the array to be of the largest size that the program could possibly need
- Care must then be taken to keep track of how much of the array is actually used
  - An indexed variable that has not been given a meaningful value ***must never be referenced***

Partially Filled Arrays

- A variable can be used to keep track of how many elements are currently stored in an array
  - For example, given the variable ***count***, the elements of the array **someArray** will range from positions **someArray[0]** through **someArray[*count* – 1]**
  - Note that the variable **count** will be used to process the partially filled array instead of **someArray.length**
  - Note also that this variable (**count**) must be an argument to any method that manipulates the partially filled array

Partially Filled Arrays

- The standard Java libraries include a number of collection classes
  - Classes whose objects store a collection of values
- Ordinary **`for`** loops cannot cycle through the elements in a collection object
  - Unlike array elements, collection object elements are not normally associated with indices
- However, there is a new kind of **`for`** loop, first available in Java 5.0, called a ***`for`**-each loop* or *enhanced **`for`** loop*
- This kind of loop can cycle through each element in a collection even though the elements are not indexed

The "for each" Loop

- Although an ordinary **`for`** loop cannot cycle through the elements of a collection class, an enhanced **`for`** loop can cycle through the elements of an array
- The general syntax for a **`for`**-each loop statement used with an array is

```
for (ArrayBaseType VariableName : ArrayName)
        Statement
```

- The above **`for`**-each line should be read as "for each **`VariableName`** in **`ArrayName`** do the following:"
  - Note that **`VariableName`** must be declared within the **`for`**-each loop, not before
  - Note also that a colon (not a semicolon) is used after **`VariableName`**

The "for each" Loop

The "for each" Loop

- The for-each loop can make code cleaner and less error prone
- If the indexed variable in a **`for`** loop is used only as a way to cycle through the elements, then it would be preferable to change it to a for-each loop
  - For example:

```
double sum = 0.0;
for (int i = 0; i < a.length; i++)
    sum += a[i];
```

  - Can be changed to:

```
double sum = 0.0;
for (double element : a)
    sum += element;
```

- Note that the for-each syntax is simpler and quite easy to understand
- Don't use for-each loop for arrays of primitive types if you want to update elements' values

- Starting with Java 5.0, methods can be defined that take any number of arguments
- Essentially, it is implemented by taking in an array as argument, but the job of placing values in the array is done automatically
  - The values for the array are given as arguments
  - Java automatically creates an array and places the arguments in the array
  - Note that arguments corresponding to regular parameters are handled in the usual way

Methods with a variable number of parameters

- Such a method has as the last item on its parameter list a *vararg specification* of the form:

  Type... ArrayName

  - Note the three dots called an *ellipsis* that must be included as part of the vararg specification syntax

- Following the arguments for regular parameters are any number of arguments of the type given in the vararg specification
  - These arguments are automatically placed in an array
  - This array can be used in the method definition
  - Note that a vararg specification allows any number of arguments, including zero

Methods with a variable number of parameters

Display 6.7 Method with a Variable Number of Parameters

```
1  public class UtilityClass
2  {
3      /**
4       Returns the largest of any number of int values.
5      */
6     public static int max(int... arg)
7      {
8          if (arg.length == 0)
9          {
10            System.out.println("Fatal Error: maximum of zero values.");
11            System.exit(0);
12         }

13         int largest = arg[0];
14         for (int i = 1; i < arg.length; i++)
15             if (arg[i] > largest)
16                 largest = arg[i];
17         return largest;
18     }
19 }
20
```

This is the file `UtilityClass.java`.

(continued)

Methods with a variable number of parameters

- If an accessor method does return the contents of an array, special care must be taken
  - Just as when an accessor returns a reference to any private object

```
public double[] getArray()
{
  return anArray;//BAD!
}
```

  - The example above will result in a *privacy leak*

Privacy Leaks with Array Instance Variables

- The previous accessor method would simply return a reference to the array **`anArray`** itself
- Instead, an accessor method should return a reference to a *deep copy* of the private array object
  - Below, both **`a`** and **`count`** are instance variables of the class containing the **`getArray`** method

```
public double[] getArray()
{
  double[] temp = new double[count];
  for (int i = 0; i < count; i++)
    temp[i] = a[i];
  return temp
}
```

Privacy Leaks with Array Instance Variables

- If a private instance variable is an array that has a class as its base type, then copies must be made of each class object in the array when the array is copied:

```
public ClassType[] getArray()
{
  ClassType[] temp = new ClassType[count];
  for (int i = 0; i < count; i++)
    temp[i] = new ClassType(someArray[i]);
  return temp;
}
```

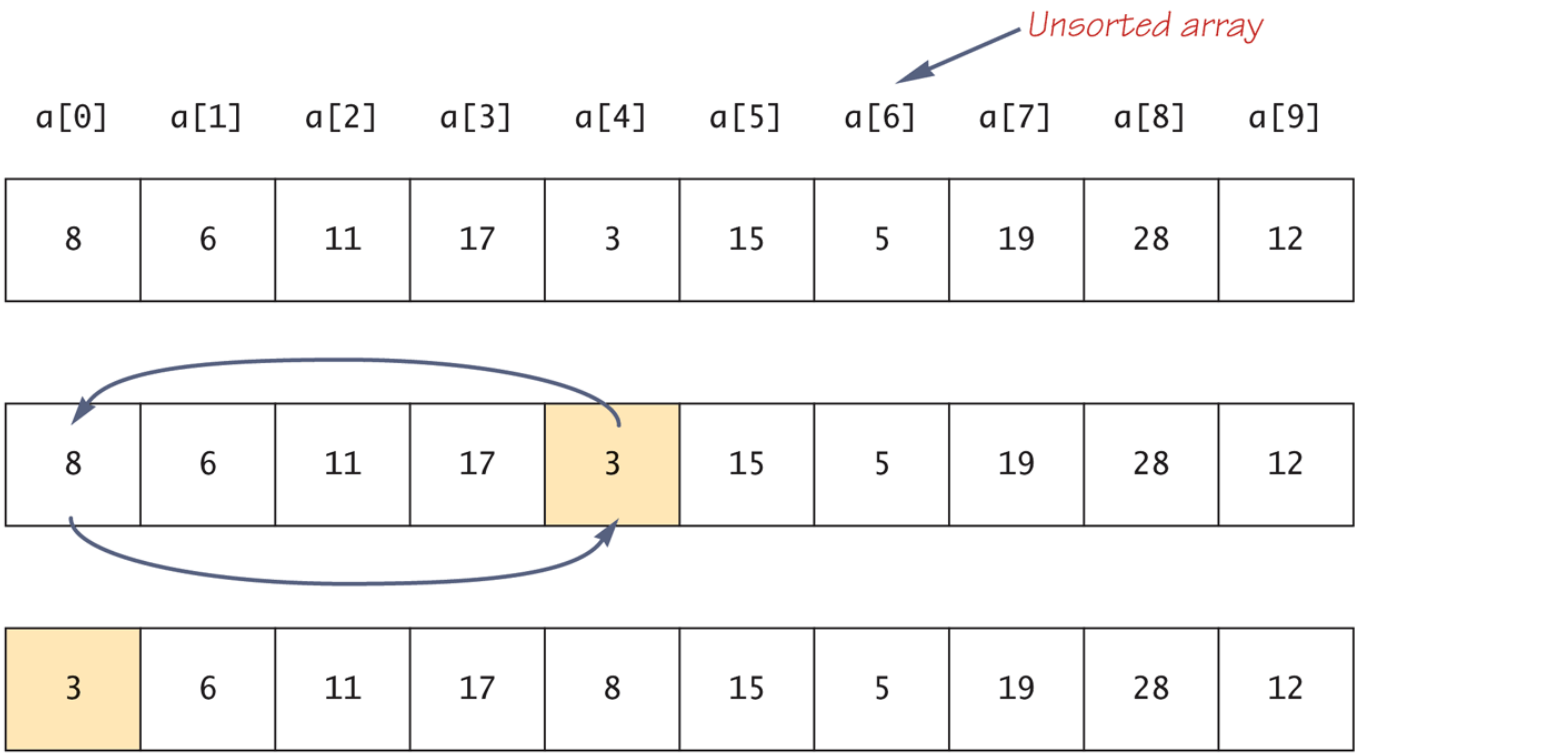
Privacy Leaks with Array Instance Variables

- A sort method takes in an array parameter **a**, and rearranges the elements in **a**, so that after the method call is finished, the elements of **a** are sorted in ascending order
- A *selection sort* accomplishes this by using the following algorithm:

```
for (int index = 0; index < count; index++)
  Place the indexth smallest element in
  a[index]
```

Sorting an Array

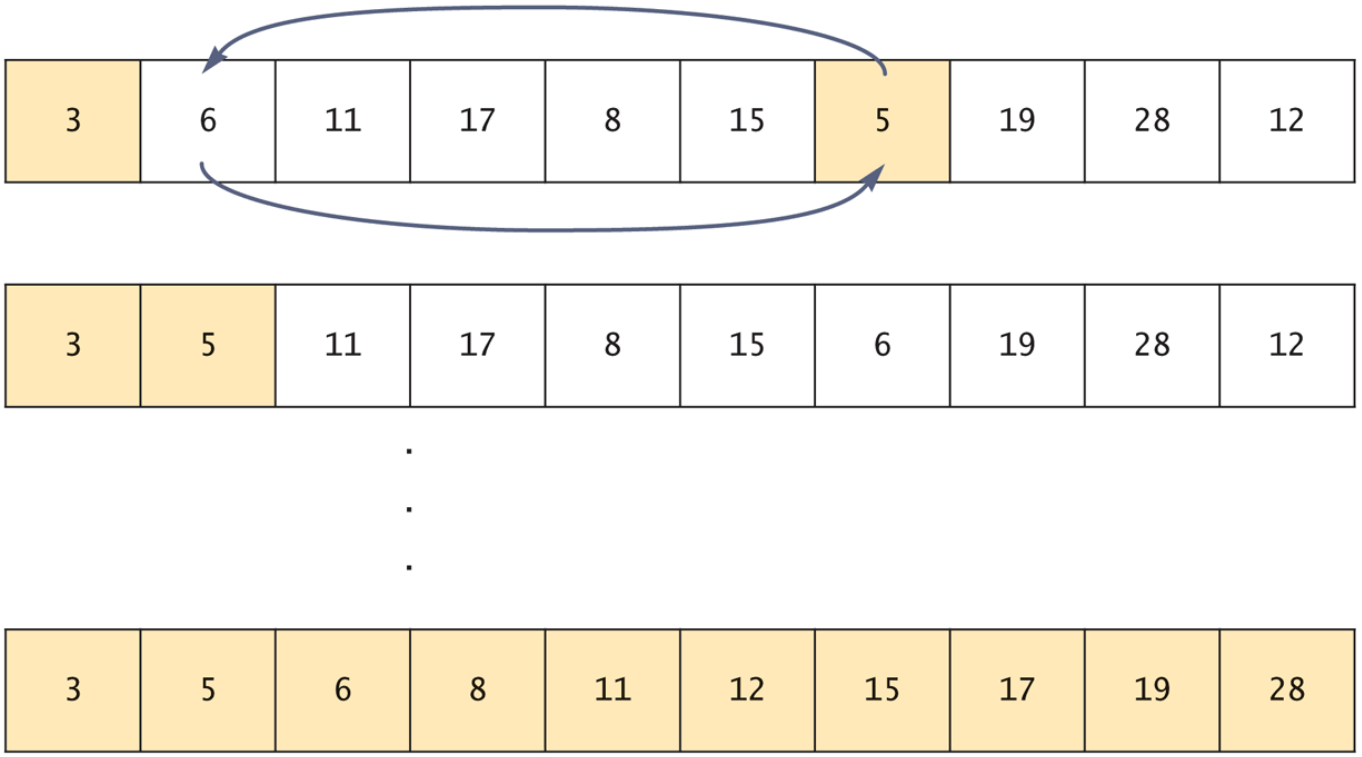
Display 6.10 Selection Sort



(continued)

Selection Sort (Part 1 of 2)

Display 6.10 **Selection Sort**



Selection Sort (Part 2 of 2)

```
public class SelectionSort
{
  /**
  Precondition: count <= a.length;
    The first count indexed variables have
    values.
  Action: Sorts a so that a[0] <= a[1] <=
    ... <= a[count - 1].
  */
```

Example: SelectionSort.java

SelectionSort Class (Part 1 of 5)

```
public static void sort(double[] a, int count)
{
  int index, indexOfNextSmallest;
  for (index = 0; index < count - 1; index++)
  {//Place the correct value in a[index]:
    indexOfNextSmallest =
            indexOfSmallest(index, a, count);
    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.
  }
}
```

SelectionSort Class (Part 2 of 5)

```
/**
Returns the index of the smallest value among
a[startIndex], a[startIndex+1], ...
a[numberUsed - 1]
*/

private static int indexOfSmallest(int
        startIndex, double[] a, int count)
{
  double min = a[startIndex];
  int indexOfMin = startIndex;
  int index;
```

SelectionSort Class (Part 3 of 5)

```
    for (index = startIndex + 1; index < count; index++)
    if (a[index] < min)
    {
      min = a[index];
      indexOfMin = index;
      //min is smallest of a[startIndex] through
      //a[index]
    }
  return indexOfMin;
}
```

SelectionSort Class (Part 4 of 5)

```
 /**
  Precondition: i and j are legal indices for
    the array a.
  Postcondition: Values of a[i] and a[j] have
    been interchanged.
  */
  private static void interchange(int i, int j,
                                  double[] a)
  {
    double temp;
    temp = a[i];
    a[i] = a[j];
    a[j] = temp; //original value of a[i]
  }
}
```

SelectionSort Class (Part 5 of 5)

Enumerated Types

- Starting with version 5.0, Java permits enumerated types
  - An enumerated type is a type in which all the values are given in a (typically) short list
- The definition of an enumerated type is normally placed outside of all methods in the same place that named constants are defined:

  enum TypeName {VALUE_1, VALUE_2, …, VALUE_N};
  - Note that a value of an enumerated type is a kind of named constant and so, by convention, is spelled with all ***uppercase*** letters
  - As with any other type, variables can be declared of an enumerated type

- Given the following definition of an enumerated type:
  enum WorkDay {MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY};
- A variable of this type can be declared as follows:
  WorkDay meetingDay, availableDay;
- The value of a variable of this type can be set to one of the values listed in the definition of the type, or else to the special value **`null`**:

```
meetingDay = WorkDay.THURSDAY;
availableDay = null;
```

Enumerated Types Example

Enumerated Types Usage

- Just like other types, variable of this type can be declared and initialized at the same time:

```
WorkDay meetingDay = WorkDay.THURSDAY;
```

  - Note that the value of an enumerated type must be prefaced with the name of the type
- The value of a variable or constant of an enumerated type can be output using **`println`**
  - The code:
    System.out.println(meetingDay);
  - Will produce the following output:
    THURSDAY
  - As will the code:
    System.out.println(WorkDay.THURSDAY);
  - Note that the type name **`WorkDay`** is not output

- Although they may look like `String` values, values of an enumerated type are not `String` values
- However, they can be used for tasks which could be done by `String` values and, in some cases, work better
  - Using a `String` variable allows the possibility of setting the variable to a nonsense value
  - Using an enumerated type variable ***constrains*** the possible values for that variable
  - An error message will result if an attempt is made to give an enumerated type variable a value that is not defined for its type

Enumerated Types Usage

- Two variables or constants of an enumerated type can be compared using the **`equals`** method or the **==** operator
- However, the **==** operator has a nicer syntax

```
if (meetingDay == availableDay)
        System.out.println("Meeting will be on
schedule.");
if (meetingDay == WorkDay.THURSDAY)
        System.out.println("Long weekend!);
```

Enumerated Types Usage

Display 6.13 An Enumerated Type

```
1  public class EnumDemo
2  {
3      enum WorkDay {MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY};
4      public static void main(String[] args)
5      {
6          WorkDay startDay = WorkDay.MONDAY;
7          WorkDay endDay = WorkDay.FRIDAY;
8          System.out.println("Work starts on " + startDay);
9          System.out.println("Work ends on " + endDay);
10     }
11 }
```

SAMPLE DIALOGUE

```
Work starts on MONDAY
Work ends on FRIDAY
```

An Enumerated Type

- Enumerated types can be used to control a **`switch`** statement
  - The **`switch`** control expression uses a variable of an enumerated type
  - Case labels are the ***unqualified*** values of the same enumerated type
- The enumerated type control variable is set by using the static method **`valueOf`** to convert an input string to a value of the enumerated type
  - The input string must contain all upper case letters, or be converted to all upper case letters using the **`toUpperCase`** method

Enumerated Types in switch statements

Display 6.16 Enumerated Type in a `switch` Statement

```
1  import java.util.Scanner;
2
3  public class EnumSwitchDemo
4  {
5      enum Flavor {VANILLA, CHOCOLATE, STRAWBERRY};

6      public static void main(String[] args)
7      {
8          Flavor favorite = null;
9          Scanner keyboard = new Scanner(System.in);
```

(continued)

Example (Part 1 of 3)

Display 6.16 Enumerated Type in a switch Statement

```
10          System.out.println("What is your favorite flavor?");
11          String answer = keyboard.next();
12          answer = answer.toUpperCase();
13          favorite = Flavor.valueOf(answer);

14          switch (favorite)
15          {
16              case VANILLA:
17                  System.out.println("Classic");
18                  break;
19              case CHOCOLATE:
20                  System.out.println("Rich");
21                  break;
22              default:
23                  System.out.println("I bet you said STRAWBERRY.");
24                  break;
25          }
26      }
27  }
```

The case labels must have just the name of the value without the type name and dot.

(continued)

Example (Part 2 of 3)

Display 6.16 **Enumerated Type in a `switch` Statement**

SAMPLE DIALOGUE

```
What is your favorite flavor?
Vanilla
Classic
```

SAMPLE DIALOGUE

```
What is your favorite flavor?
STRAWBERRY
I bet you said STRAWBERRY.
```

SAMPLE DIALOGUE

```
What is your favorite flavor?
PISTACHIO
```

This input causes the program to end and issue an error message.

Example (Part 3 of 3)

- Introduction to arrays
  - Creating and accessing arrays
  - Use for loops with arrays
- Arrays and references
- Programming with arrays
  - Sorting
- **ArrayList**
- Multidimensional arrays

Outline

- **ArrayList is a class in the standard Java libraries**
  - Unlike arrays, which have a fixed length once they have been created, an **`ArrayList`** is an object that can **grow** and **shrink** while your program is running
- In general, an **`ArrayList`** serves the same purpose as an array, except that an **`ArrayList`** can ***change length*** while the program is running
- The class **`ArrayList`** is implemented using an array as a private instance variable
  - When this hidden array is full, a new larger hidden array is created and the data is transferred to this new array

The ArrayList Class

Why not always use an **`ArrayList`** instead of an array?

1. An **`ArrayList`** is **less efficient** than an array
2. It does not have the convenient **square bracket** notation
3. The base type of an **`ArrayList`** must be a class type or interface type (or other reference type): it cannot be a primitive type

– This last point is less of a problem now that Java provides automatic boxing and unboxing of primitives

The ArrayList Class

- In order to make use of the **`ArrayList`** class, it must first be imported from the package **`java.util`**
- An **`ArrayList`** is created and named in the same way as object of any class, except that you specify the base type as follows:

```
ArrayList<BaseType> aList =
   new ArrayList<BaseType>();
```

Compare with array:

```
double[] score= new double[5];
```

Using the ArrayList Class

- An **initial capacity** can be specified when creating an **`ArrayList`** as well
  - The following code creates an **`ArrayList`** that stores objects of the base type **`String`** with an initial capacity of 20 items

```
ArrayList<String> list =
    new ArrayList<String>(20);
```

  - Specifying an initial capacity does not limit the size to which an **`ArrayList`** can eventually grow
- Note that the base type of an ArrayList is specified as a ***type parameter***

Using the ArrayList Class

- The **`add`** method is used to set an element for the first time in an **`ArrayList`**

```
list.add("something");
```

  - The method name **`add`** is overloaded
  - There is also a two argument version that allows an item to be added at any currently used index position or at the first unused position

Using the ArrayList Class

- The `size` method is used to find out how many indices already have elements in the `ArrayList`

```
int howMany = list.size();
```

- The `set` method is used to replace any existing element, and the `get` method is used to access the value of any existing element

```
list.set(index, "something else");
String thing = list.get(index);
```

Using the ArrayList Class

- The simplest **`add`** method has a single parameter for the element to be added to the end of the ArrayList
  - **list.add("one");**
  - **list.add("two");**
  - **list.add("three");**

Tip: Summary of Adding to an ArrayList

- An element can be added at an already occupied list position by using the two-parameter version of **`add`**
- This causes the new element to be placed at the index specified, and every other member of the **`ArrayList`** to be **moved up** by one position
- **list.add(0,"Zero");**

Tip: Summary of Adding to an ArrayList

- The tools for manipulating arrays consist only of the square brackets and the instance variable `length`
- **ArrayLists, however, come with a selection of powerful methods that can do many of the things for which code would have to be written in order to do them using arrays**

Methods in the Class ArrayList

- The **`ArrayList`** class is an example of a *collection* class
- Starting with version 5.0, Java has added a new kind of for loop called a ***for-each*** or ***enhanced for*** **loop**
  - This kind of loop has been designed to cycle through all the elements in a collection (like an **`ArrayList`**)

The "For Each" Loop

Display 14.2 **A for-each Loop Used with an ArrayList**

```
1  import java.util.ArrayList;
2  import java.util.Scanner;

3  public class ArrayListDemo
4  {
5     public static void main(String[] args)
6     {
7        ArrayList<String> toDoList = new ArrayList<String>(20);
8        System.out.println(
9                        "Enter list entries, when prompted.");
10       boolean done = false;
11       String next = null;
12       String answer;
13       Scanner keyboard = new Scanner(System.in);
```

(continued)

Example (Part 1 of 3)

Display 14.2 **A for-each Loop Used with an ArrayList**

```
14          while (! done)
15          {
16              System.out.println("Input an entry:");
17              next = keyboard.nextLine();
18              toDoList.add(next);

19              System.out.print("More items for the list? ");
20              answer = keyboard.nextLine();
21              if (!(answer.equalsIgnoreCase("yes")))
22                    done = true;
23          }

24          System.out.println("The list contains:");
25          for (String entry : toDoList)
26               System.out.println(entry);
27      }
28  }
29
```

(continued)

Example (Part 2 of 3)

Display 14.2 **A for-each Loop Used with an `ArrayList`**

SAMPLE DIALOGUE

```
Enter list entries, when prompted.
Input an entry:
Practice Dancing.
More items for the list? yes
Input an entry:
Buy tickets.
More items for the list? yes
Input an entry:
Pack clothes.
More items for the list? no
The list contains:
Practice Dancing.
Buy tickets.
Pack clothes.
```

Example (Part 3 of 3)

Display 14.3 **Golf Score Program**

```
1  import java.util.ArrayList;
2  import java.util.Scanner;

3  public class GolfScores
4  {
5      /**
6       Shows differences between each of a list of golf scores and their average.
7      */
8      public static void main(String[] args)
9      {
10       ArrayList<Double> score = new ArrayList<Double>();

11         System.out.println("This program reads golf scores and shows");
12         System.out.println("how much each differs from the average.");

13         System.out.println("Enter golf scores:");
14         fillArrayList(score);
15         showDifference(score);
16     }
```

Parameters of type `ArrayList<Double>()` *are handled just like any other class parameter.*

(continued)

Golf Score Program (Part 1 of 6)

Display 14.3 **Golf Score Program**

```
17      /**
18       Reads values into the array a.
19      */
20      public static void fillArrayList(ArrayList<Double> a)
21      {
22          System.out.println("Enter a list of nonnegative numbers.");
23          System.out.println("Mark the end of the list with a negative number.");
24          Scanner keyboard = new Scanner(System.in);
```

(continued)

Golf Score Program (Part 2 of 6)

Display 14.3 Golf Score Program

```
25          double next;
26          int index = 0;
27          next = keyboard.nextDouble();
28          while (next >= 0)
29          {
30              a.add(next);
31              next = keyboard.nextDouble();
32          }
33      }
34      /**
35        Returns the average of numbers in a.
36      */
37      public static double computeAverage(ArrayList<Double> a)
38      {
39          double total = 0;
40          for (Double element : a)
41              total = total + element;
```

Because of automatic boxing, we can treat values of type `double` *as if their type were* `Double`.

A for-each loop is the nicest way to cycle through all the elements in an `ArrayList`.

(continued)

Golf Score Program (Part 3 of 6)

Display 14.3 Golf Score Program

```
42          int numberOfScores = a.size();
43          if (numberOfScores > 0)
44          {
45              return (total/numberOfScores);
46          }
47          else
48          {
49              System.out.println("ERROR: Trying to average 0 numbers.");
50              System.out.println("computeAverage returns 0.");
51              return 0;
52          }
53      }
```

(continued)

Golf Score Program (Part 4 of 6)

Display 14.3 Golf Score Program

```
54      /**
55        Gives screen output showing how much each of the elements
56        in a differ from their average.
57      */
58      public static void showDifference(ArrayList<Double> a)
59      {
60          double average = computeAverage(a);
61          System.out.println("Average of the " + a.size()
62                                              + " scores = " + average);
63          System.out.println("The scores are:");
64          for (Double element : a)
65              System.out.println(element + " differs from average by "
66                                              + (element – average));
67      }
68  }
```

(continued)

Golf Score Program (Part 5 of 6)

Display 14.3 **Golf Score Program**

SAMPLE DIALOGUE

```
This program reads golf scores and shows
how much each differs from the average.
Enter golf scores:
Enter a list of nonnegative numbers.
Mark the end of the list with a negative number.
69  74  68  -1
Average of the 3 scores = 70.3333
The scores are:
69.0 differs from average by -1.33333
74.0 differs from average by 3.66667
68.0 differs from average by -2.33333
```

Golf Score Program (Part 6 of 6)

- An **`ArrayList`** automatically increases its capacity when needed
  - However, the capacity may increase beyond what a program requires
  - In addition, although an **`ArrayList`** grows automatically when needed, it does not shrink automatically
- If an **`ArrayList`** has a large amount of excess capacity, an invocation of the method **`trimToSize`** will shrink the capacity of the **`ArrayList`** down to the size needed

Tip: Use trimToSize to save memory

- Introduction to arrays
  - Creating and accessing arrays
  - Use for loops with arrays
- Arrays and references
- Programming with arrays
  - Sorting
- ArrayList
- **Multidimensional arrays**

Outline

- It is sometimes useful to have an array with more than one index
- Multidimensional arrays are declared and created in basically the same way as one-dimensional arrays
  - You simply use as many square brackets as there are indices
  - Each index must be enclosed in its own brackets

```
double[][]table = new double[100][10];
int[][][] figure = new int[10][20][30];
Person[][] = new Person[10][100];
```

Multidimensional Arrays

Multidimensional Arrays

- Multidimensional arrays may have any number of indices, but perhaps the most common number is two
  - Two-dimensional array can be visualized as a two-dimensional display with the first index giving the row, and the second index giving the column

```
char[][] a = new char[5][12];
```

  - Note that, like a one-dimensional array, each element of a multidimensional array is just a variable of the base type (in this case, `char`)

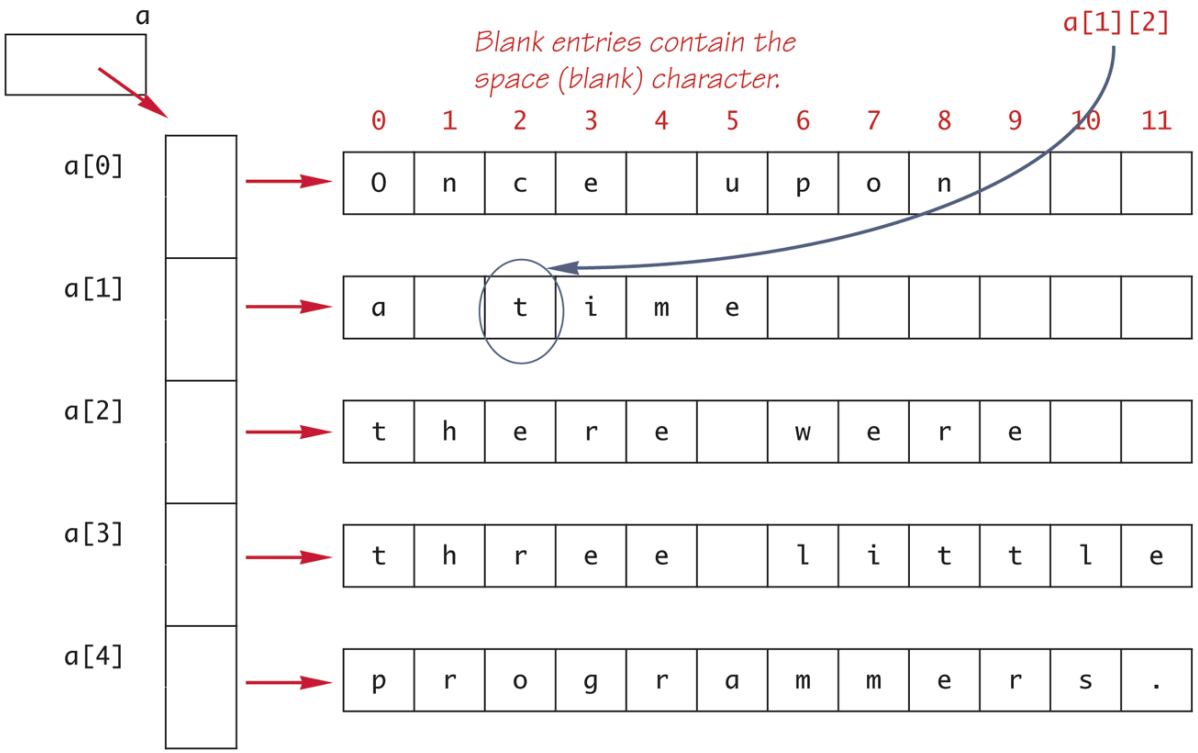
- In Java, a two-dimensional array, such as **`a`**, is actually an array of arrays
  - The array **`a`** contains a reference to a one-dimensional array of size 5 with a base type of **`char[]`**
  - Each indexed variable (**`a[0]`**, **`a[1]`**, etc.) contains a reference to a one-dimensional array of size 12, also with a base type of **`char[]`**
- A three-dimensional array is an array of arrays of arrays, and so forth for higher dimensions

Multidimensional Arrays

Display 6.17 Two-Dimensional Array as an Array of Arrays

```
char[][] a = new char[5][12];
```

Code that fills the array is not shown.



(continued)

2-D Array as an Array of Arrays (Part 1 of 2)

Display 6.17 Two-Dimensional Array as an Array of Arrays

We will see that these can and should be replaced with expressions involving the `length` *instance variable.*

```
int row, column;
for (row = 0; row < 5; row++)
{
    for (column = 0; column < 12; column++)
       System.out.print(a[row][column]);
    System.out.println();
}
```

Produces the following output:

```
Once upon
a time
there were
three little
programmers.
```

2-D Array as an Array of Arrays (Part 2 of 2)

```
char[][] page = new char[30][100];
```

- The instance variable **`length`** does not give the total number of indexed variables in a two-dimensional array
  - Because a two-dimensional array is actually an array of arrays, the instance variable **`length`** gives the number of first indices (or "rows") in the array
    - **`page.length`** is equal to 30
  - For the same reason, the number of second indices (or "columns") for a given "row" is given by referencing **`length`** for that *"row" variable*
    - **`page[0].length`** is equal to 100

Using the length instance variable

- The following program demonstrates how a nested **`for`** loop can be used to process a two-dimensional array
  - Note how each **`length`** instance variable is used

```
int row, column;
for (row = 0; row < page.length; row++)
  for (column = 0; column < page[row].length;  column++)
    page[row][column] = 'Z';
```

Using the length instance variable

- Each row in a two-dimensional array need not have the same number of elements
  - Different rows can have different numbers of columns
- An array that has a different number of elements per row it is called a *ragged array*

Ragged Arrays

```
double[][] a = new double[3][5];
```

- The above line is equivalent to the following:

```
double [][] a;
a = new double[3][]; //Note below
a[0] = new double[5];
a[1] = new double[5];
a[2] = new double[5];
```

  - Note that the second line makes **`a`** the name of an array with room for 3 entries, each of which can be an array of **`doubles`** *that can be of any length*
  - The next 3 lines each create an array of doubles of size 5

Ragged Arrays

```
double [][] a;
a = new double[3][];
```

- Since the above line does not specify the size of `a[0]`, `a[1]`, or `a[2]`, each could be made a different size instead:

```
a[0] = new double[5];
a[1] = new double[10];
a[2] = new double[4];
```

Ragged Arrays

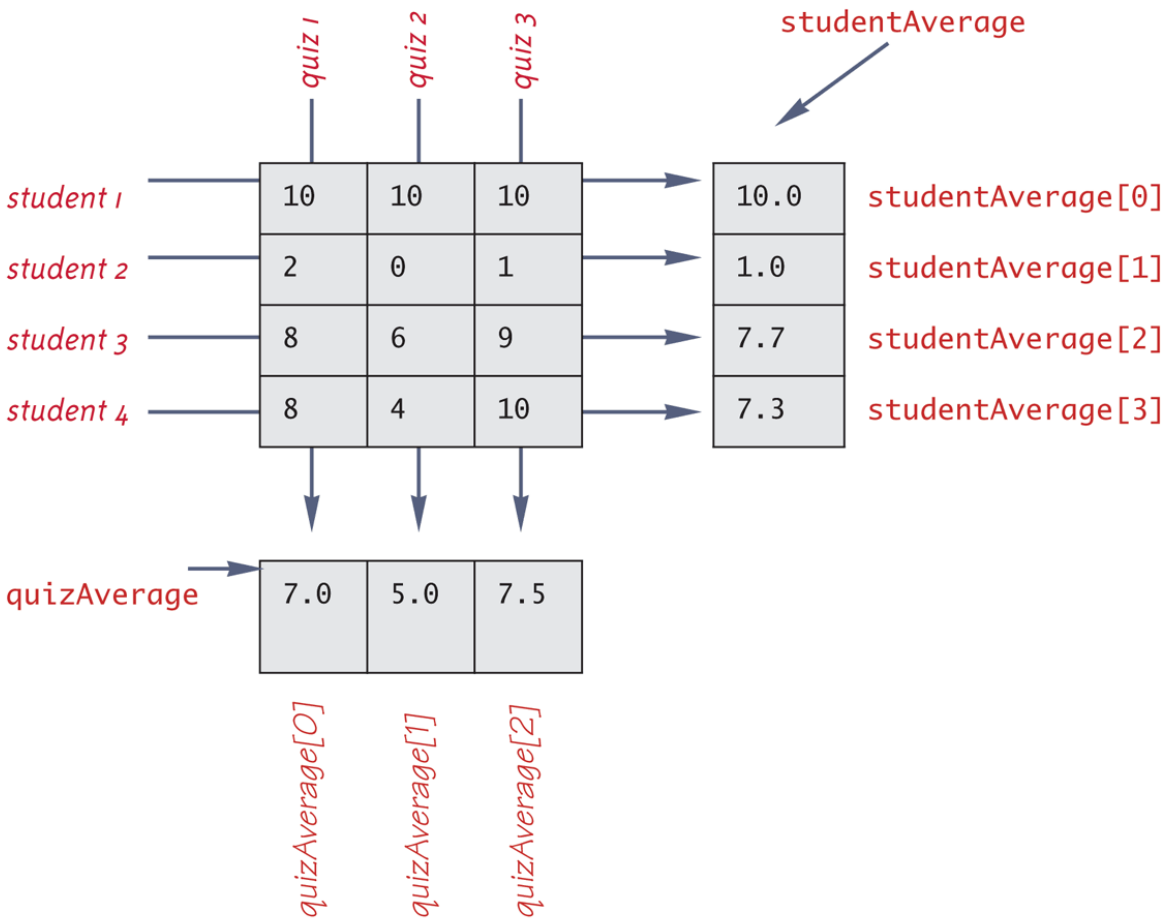
- As an example of using arrays in a program, a class **`GradeBook`** is used to process quiz scores
- Objects of this class have three instance variables
  - **grade**: **a two-dimensional array that records the grade of each student on each quiz**
  - **studentAverage**: **an array used to record the average quiz score for each student**
  - **quizAverage**: **an array used to record the average score for each quiz**

A Grade Book Class

- The score that student 1 received on quiz number 3 is recorded in **`grade[0][2]`**
- The average quiz grade for student 2 is recorded in **`studentAverage[1]`**
- The average score for quiz 3 is recorded in **`quizAverage[2]`**
- Note the relationship between the three arrays

A Grade Book Class

Display 6.19 **The Two-Dimensional Array grade**



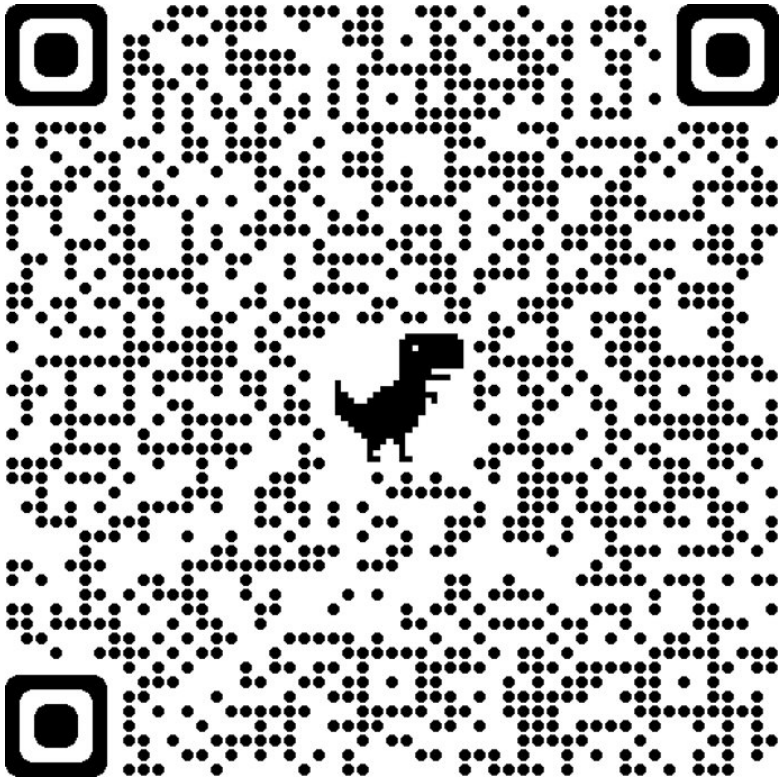
A Grade Book Class

- Introduction to arrays
  - Creating and accessing arrays
  - Use for loops with arrays
- Arrays and references
- Programming with arrays
  - Sorting
- ArrayList
- Multidimensional arrays

Learning Outcomes

- Which moment or experience from COMP90041 this week was significant or important to you?
- Why do you think this experience was significant
  - Examine your experience. Why do you care?)
- What insights have you had?
  - What can you learn from the experience?)
- How is this experience going to help you in the future?
- What questions have come up for you?

Class Reflections



http://go.unimelb.edu.au/5o8i.

Class Reflections