COMP 102.2x
Introduction to Java Programming – Part 2
Lecture One

T.C. Pong
Department of Computer Science & Engineering
HKUST

香港科技大學
THE HONG KONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

Topics to be covered

Topics to be covered in part 2 of this course:

- 2D and multidimensional arrays
- Character strings and File I/O
- Event-driven programming and GUI
- Recursion
- Abstract data types.

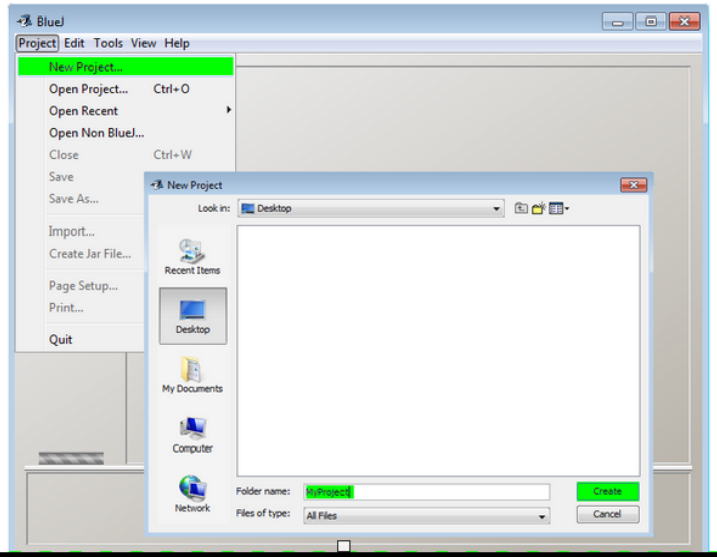
Click here to download the PDF for this tutorial.

BlueJ as IDE

CREATING A NEW BLUEJ PROJECT

Before writing a program, we will first have to create a BlueJ project for that program. Here is the procedure on how to create a new BlueJ project:

1. Launch BlueJ
2. From the menu bar, click on **Project** -> **New Project**
3. Navigate to your desired location
4. Key in your desired file name for the project folder
5. Click on the **Create** button



Part 2: Lecture 1

Topics to be covered in this lecture:

- Fundamental Principles of OOP
- 2D and multidimensional arrays
- Sorting and searching

Fundamental Principles of OOP

Three fundamental principles of object-oriented programming (OOP):

- Encapsulation: Packing data and functions into a single component (class) so as to hide implementation details.
- Inheritance: Objects in a subclass are allowed to inherit properties (including data and methods) of a superclass.
- Polymorphism: The provision of the same interface to objects of different types.

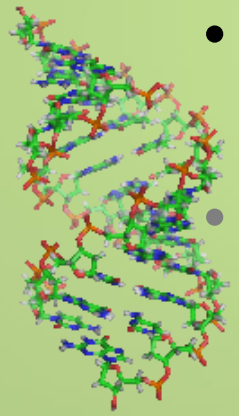
OOP: Encapsulation

- Encapsulation: Packing data and functions into a single component (class) so as to hide implementation details.
  - Only **what** a class can do is visible but not **how** it does it
- Java programs contain nothing but definitions and instantiations of classes.

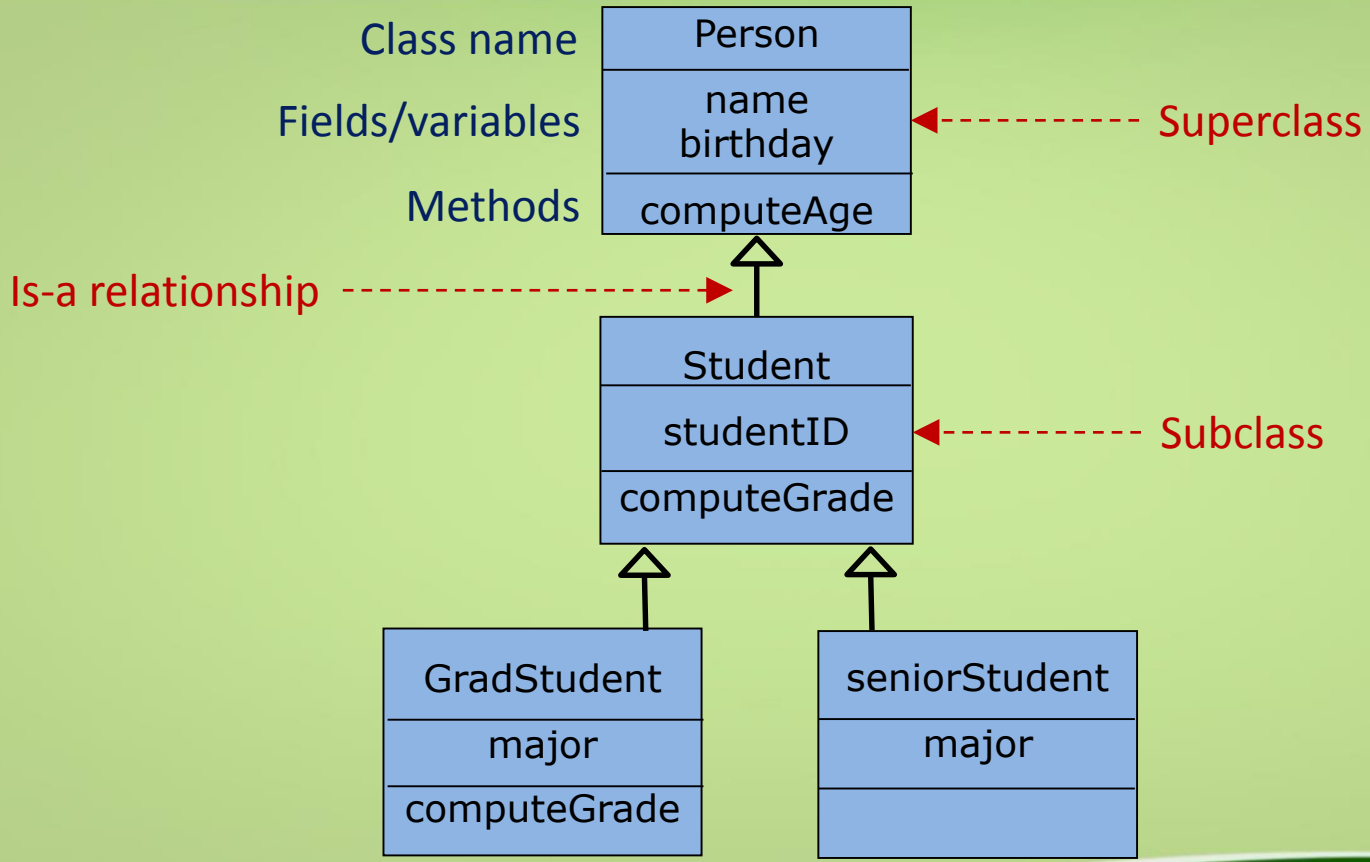
OOP: Inheritance

Three general principles of object-oriented programming:

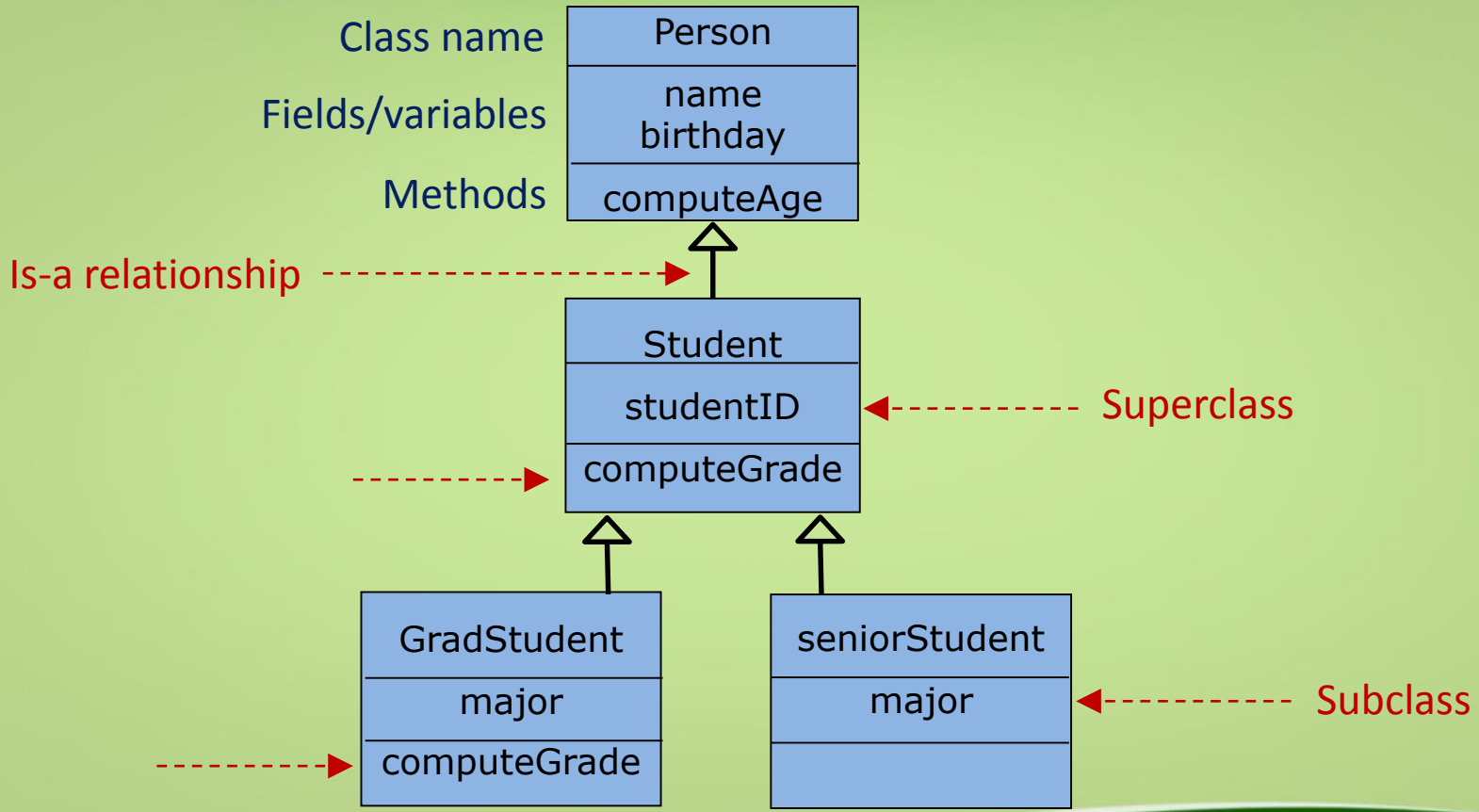
- Encapsulation: The bundling of data and functions into a simple component, and their implementation is hidden from the user.
- Inheritance: Objects in a subclass are allowed to inherit properties (including data and methods) of a superclass.
- Polymorphism: The provision of same interface for objects of different types.



Inheritance: Class Diagram



Inheritance: Class Diagram



OOP: Polymorphism

Three general principles of object-oriented programming:

- Encapsulation: The bundling of data and functions into a simple component, and their implementation is hidden from the user.
- Inheritance: Objects in a subclass are allowed to inherit properties (including data and methods) of a superclass.
- Polymorphism: The provision of same interface for objects of different types.

Subclass and Inheritance

- A subclass is a class that is derived from another class (superclass).
  - public class SubclassName extends SuperClassName
- The class **Object** is the root of the Java class hierarchy.
- A subclass inherits all the fields and methods from its superclass.
- The keyword super can be used for a subclass to invoke the constructors or methods of its superclass.

Example: Savings Account

Savings account:

- Bank account that earns interest from the account balance
- As an example, for an account with a principal of $1,000 that earns an annual interest of 10%, assuming that the interest is compounded annually.
- What would be the accumulated balance at the end of 5 years?

- 1st year: interest earned $1,000 * 10% or 100, new balance $1,100
- 2nd year: interest earned $1,100 * 10% or 110, new balance $1,210
- 3rd year: interest earned $1,210 * 10% or 121, new balance $1,331
- 4th year: interest earned $1,331 * 10% or 133.1, new balance $1,464.1
- 5th year: interest earned $1,464.1 * 10% or 146.41, new balance $1,610.51

An Example: Bank Account

```
import comp102x.IO;
/**
 * A bank account has a balance and an owner who can make
 * deposits to and withdrawals from the account.
 */
public class BankAccount {
   private double balance = 0.0;     // Initial balance is set to zero
   private String owner = "NoName";     // Name of owner

   /**
    *   Default constructor for a bank account with zero balance
    */
   public BankAccount ( ) { }
   /**
    *   Construct a balance account with a given initial balance and owner's name
    *   @param   initialBalance     the initial balance
    *   @param   name               name of owner
    */
   public BankAccount (double initialBalance, String name) {
      balance = initialBalance;
      owner = name;
   }
```

An Example: Bank Account

```
/**
 *  Method for depositing money to the bank account
 *  @param  dAmount    the amount to be deposited
 */
public void deposit(double dAmount) {
    balance = balance + dAmount;
}
/**
 *  Method for withdrawing money from the bank account
 *  @param  wAmount    the amount to be withdrawn
 */
public void withdraw(double wAmount) {
    balance = balance - wAmount;
}
/**
 *  Method for getting the current balance of the bank account
 *  @return  the current balance
 */
public double getBalance() {
    return balance;
}
```

An Example: Bank Account

```
import comp102x.IO;
/**
 * SavingsAccount is a subclass of BankAccount.
 */
public class SavingsAccount extends BankAccount {
    double interestRate;
    /**
     *   Constructor that makes use of the constructor from super class
     */
    public SavingsAccount (double initialBalance, String name, double rate)  {
        super(initialBalance, name);
        interestRate = rate;
    }
```

```
// -  A subclass inherits all the members including fields and
//    methods from its superclass.
// -  Constructors of the superclass are NOT inherited by subclasses
//     but can be invoked from the subclass.
```

An Example: Bank Account

```
/**
   *  compoundInterest computes the compound interest for a given duration
   *
   *  @param   duration        the number of times the interest is to be compounded
   */
public void compoundInterest (int duration)   {
    for (int i =1; i <= duration; i++) {
        double currentBalance = getBalance();
        deposit(currentBalance * interestRate);}
    }

public void setInterestRate(double rate) {
    interestRate = rate;
    }
}
```

Interest earned for the i^{th} period

```
// Formula for computing compound interest:
//      P ( 1 + r ) n
```

Annotation

- Introduced in Java 5
- Predefined annotations provide information to the compiler to detect errors or suppress warnings
- Most predefined annotations have no effect on code
- Syntax
  - *@AnnotationName*
- Some predefined annotations in Java
  - @Override, @SuppressWarnings

An Example: Modified Checking Account

```
/**
 * CheckingAccount is a subclass of BankAccount.
 * A fee is charged for each withdrawal from a CheckingAccount
 */

public class CheckingAccount extends BankAccount {

   // instance variables perChequeFee is the fee charged per cheque
   private double perChequeFee;

   /**
    * Constructor for objects of class CheckingAccount
    */
   public CheckingAccount(double initialBalance, String name, double fee) {

      super(initialBalance, name); // constructor from the superclass is called
      perChequeFee = fee;
   }
```

An Example: Investment Account

```
    /**
     * The method withdraw withdraws with wAmount plus a fee from
     * CheckingAcount
     *
     * @param  wAmount   the amount to be withdraw from the account
     */

    @Override
    public void withdrawal(double wAmount) {

        // a fee is added to each withdrawal
        super.withdraw(wAmount + perChequeFee);

    }

    public void showBalance() {
        System.out.println("Balance: " + super.getBalance());
    }

}
```

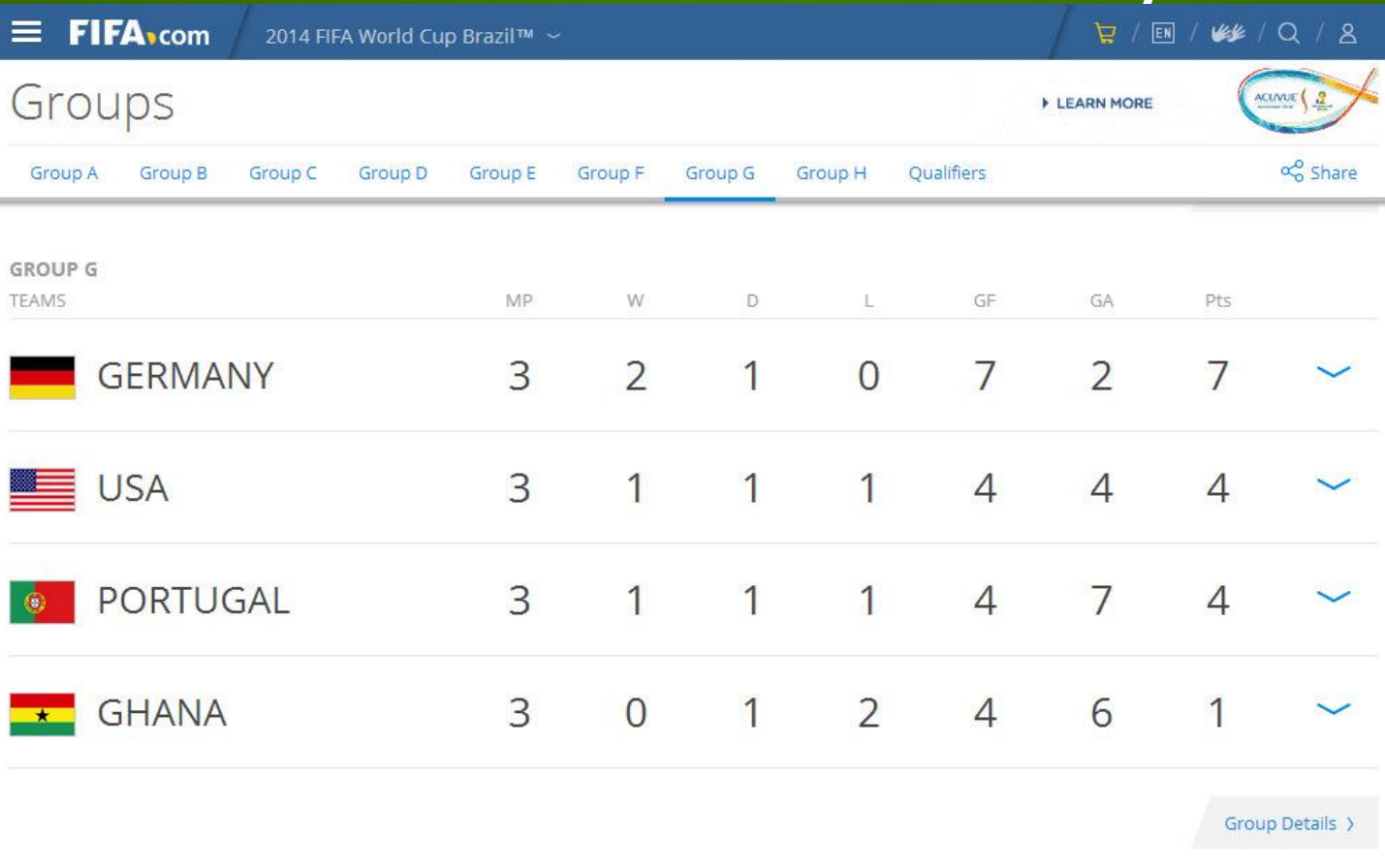
Error detected during compilation.

2D and Multidimensional Arrays

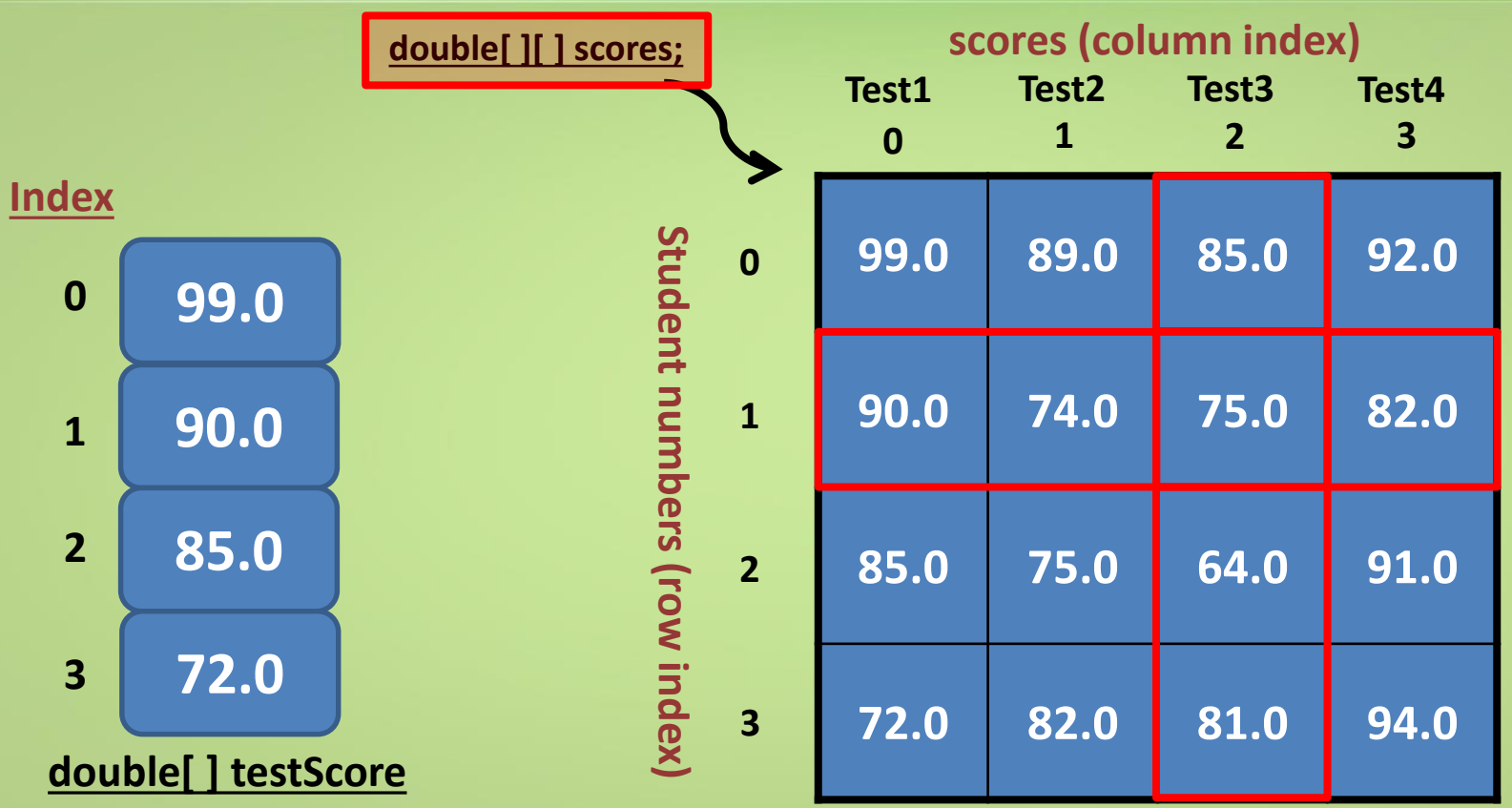
Two-dimensional Array

- The idea of one-dimensional array can be extended to two-dimensional
- A R x C two dimensional array can be illustrated as a table with R rows and C columns
- Example: The following scores could be stored for each student in a course:
  - Exam score
  - Homework score
  - Lab score
  - Final score

Two-dimensional Array



Example: Students' test scores

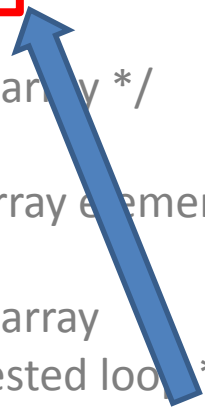


Example: Scores

```
public class Scores {
   /* 1. A 2D array instance variable */

   /* 2. Initialize a 2D array */

   /* 3. Access a 2D array element */

   /* 4. Traverse a 2D array
                 using a nested loop */

}
```

Example: Declare Scores

```
public class Scores {
/* 1. A 2D array instance variable */
   private double [ ][ ] scores ;

   /* 2. Initialize a 2D array */

   /* 3. Access a 2D array element */

   /* 4. Traverse a 2D array
               using a nested loop */

}
```



Define an instance variable of a 2D array
Syntax:
DataType[ ][ ] nameOfTheVariable;

Initializing Scores

```
public class Scores {
   /* 1. A 2D array instance variable */
   private double [ ][ ] scores ;

   public void initializeAllScores( ) {

        /* 2. Initialize a 2D array */

   }

   /* 3. Access a 2D array element */
   /* 4. Traverse a 2D array using a nested loop */
}
```

Initializing Scores

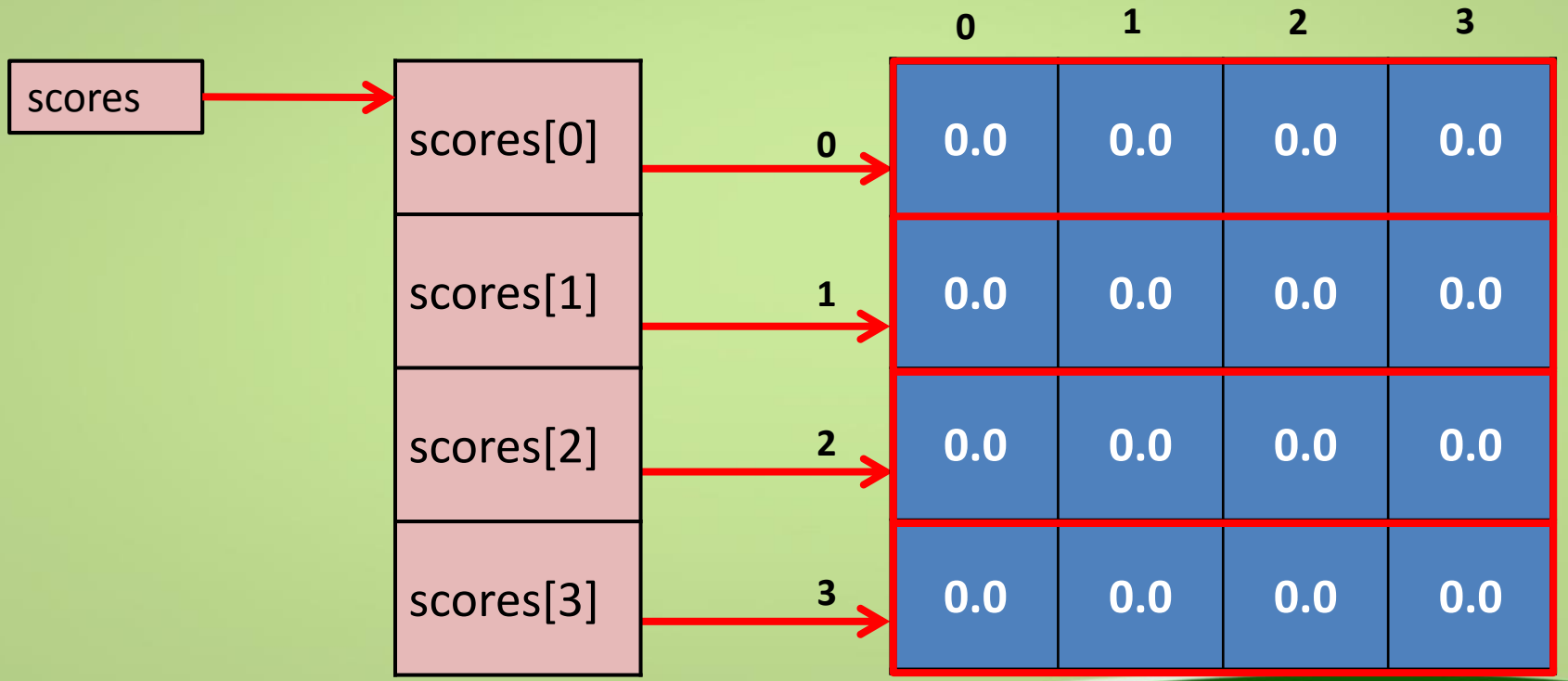
```
public void initializeAllScores( ) {

 scores = new double[4][4];

}
```

double[ ][ ] scores	0	1	2	3
0	0.0	0.0	0.0	0.0
1	0.0	0.0	0.0	0.0
2	0.0	0.0	0.0	0.0
3	0.0	0.0	0.0	0.0

An array of 4 rows and 4 columns is created

2D array as an Array of 1D arrays

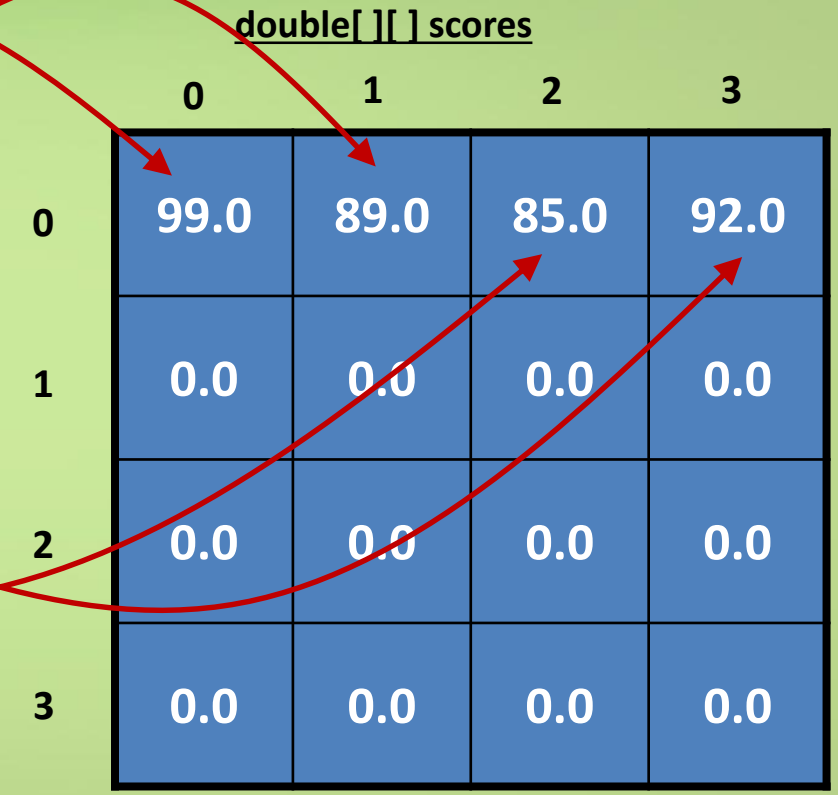
- Each row can be visualized as a 1D array
  - **double[ ][ ] scores = new double[4][4];**



Initializing Scores

```
public void initializeAllScores() {
scores = new double[4][4];

scores[0][0] = 99.0; scores[0][1] = 89.0;
scores[0][2] = 85.0; scores[0][3] = 92.0;

}
```

The first index is the row index and the second is the column index. Both indices start from 0



Initializing Scores

```
public void initializeAllScores() {
   scores = new double[4][4];

    scores[0][0] = 99.0; scores[0][1] = 89.0;
    scores[0][2] = 85.0; scores[0][3] = 92.0;

    scores[1][0] = 90.0; scores[1][1] = 74.0;
    scores[1][2] = 75.0; scores[1][3] = 82.0;
}
```

Initializing the second row

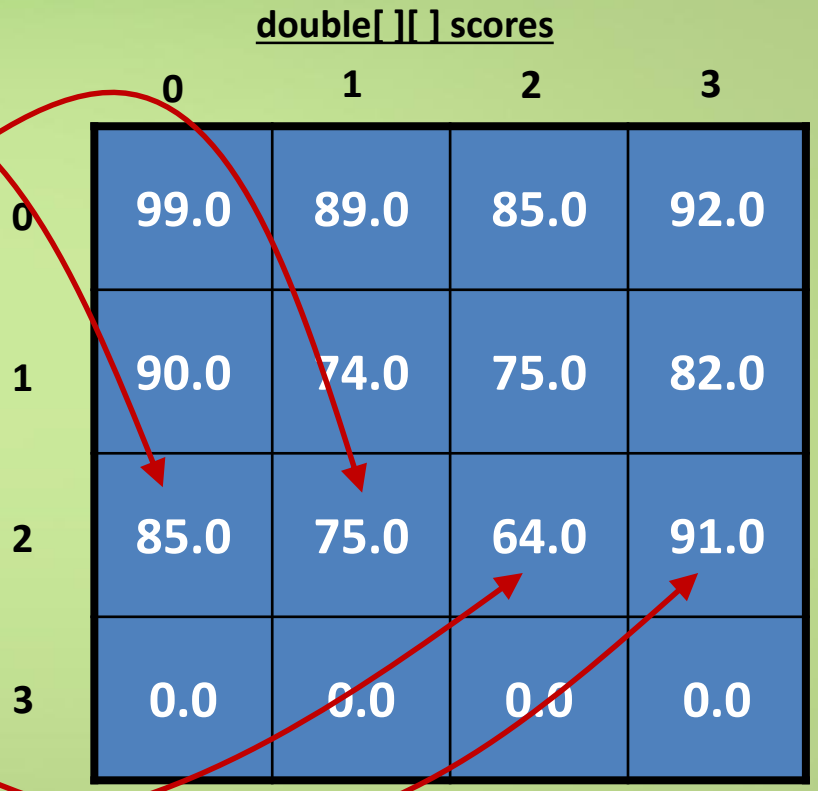
double[ ][ ] fares

	0	1	2	3
0	99.0	89.0	85.0	92.0
1	90.0	74.0	75.0	82.0
2	0.0	0.0	0.0	0.0
3	0.0	0.0	0.0	0.0

Initializing Scores

```
public void initializeAllScores() {
 scores = new double[4][4];

  scores[0][0] = 99.0; scores[0][1] = 89.0;
  scores[0][2] = 85.0; scores[0][3] = 92.0;
  scores[1][0] = 90.0; scores[1][1] = 74.0;
  scores[1][2] = 75.0; scores[1][3] = 82.0;
  scores[2][0] = 85.0; scores[2][1] = 75.0;
  scores[2][2] = 64.0; scores[2][3] = 91.0;

 }
```

Initializing the third row



Initializing Scores

```
public void initializeAllScores() {
 scores = new double[4][4];
   scores[0][0] = 99.0; scores[0][1] = 89.0;
   scores[0][2] = 85.0; scores[0][3] = 92.0;
   scores[1][0] = 90.0; scores[1][1] = 74.0;
   scores[1][2] = 75.0; scores[1][3] = 82.0;
   scores[2][0] = 85.0; scores[2][1] = 75.0;
   scores[2][2] = 64.0; scores[2][3] = 91.0;

   scores[3][0] = 72.0; scores[3][1] = 82.0;
   scores[3][2] = 81.0; scores[3][3] = 94.0;

}
```

Initializing the forth row

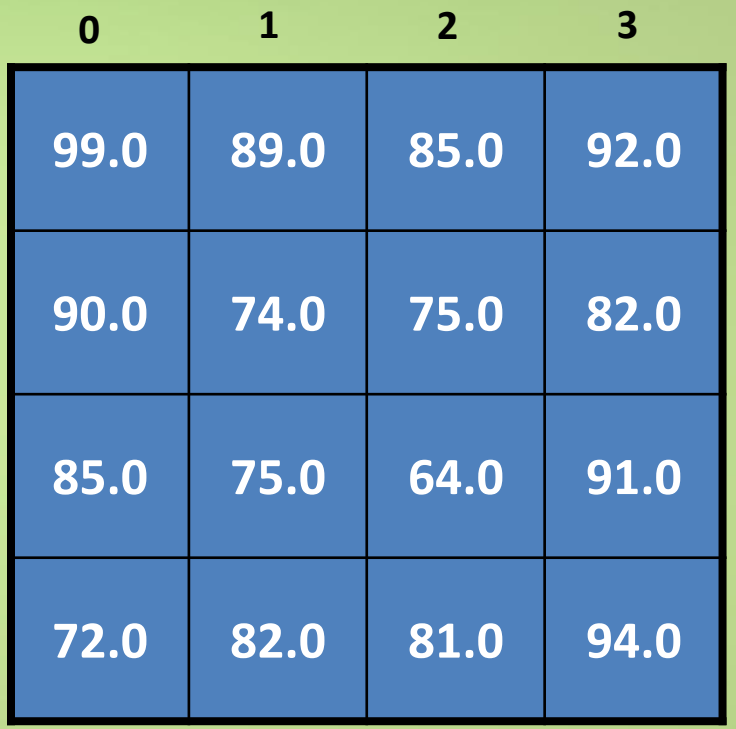
double[ ][ ] scores

	0	1	2	3
0	99.0	89.0	85.0	92.0
1	90.0	74.0	75.0	82.0
2	85.0	75.0	64.0	91.0
3	72.0	82.0	81.0	94.0

Shorthand notation for a 2D array

- Declare, define and initialize a 2D array using a single statement:

```
double [ ][ ] scores = {
    {99.0, 89.0, 85.0, 92.0},
    {90.0, 74.0, 75.0, 82.0},
    {85.0, 75.0, 64.0, 91.0},
    {72.0, 82.0, 81.0, 94.0}
};
```



Caution on the shorthand syntax

- This shorthand syntax **must be in one statement**

```
double [ ][ ] scores = {
    {99.0, 89.0, 85.0, 92.0},
    {90.0, 74.0, 75.0, 82.0},
    {85.0, 75.0, 64.0, 91.0},
    {72.0, 82.0, 81.0, 94.0}
};
```

```
double [ ][ ] scores;
scores = {
    {99.0, 89.0, 85.0, 92.0},
    {90.0, 74.0, 75.0, 82.0},
    {85.0, 75.0, 64.0, 91.0},
    {72.0, 82.0, 81.0, 94.0}
};
```

Access a 2D array element

```
public class Scores{

   /* 1. A 2D array instance variable */
   private double[ ][ ] scores;
   public void initializeAllScores() {   /* 2. Initialize a 2D array */    }

   public double getScoreByIndices(int rowIndex, int colIndex) {
         /* 3. Access a 2D array element */
    }

   /* 4. Traverse a 2D array
                 using a nested loop */

}
```

Access an element in a 2D array

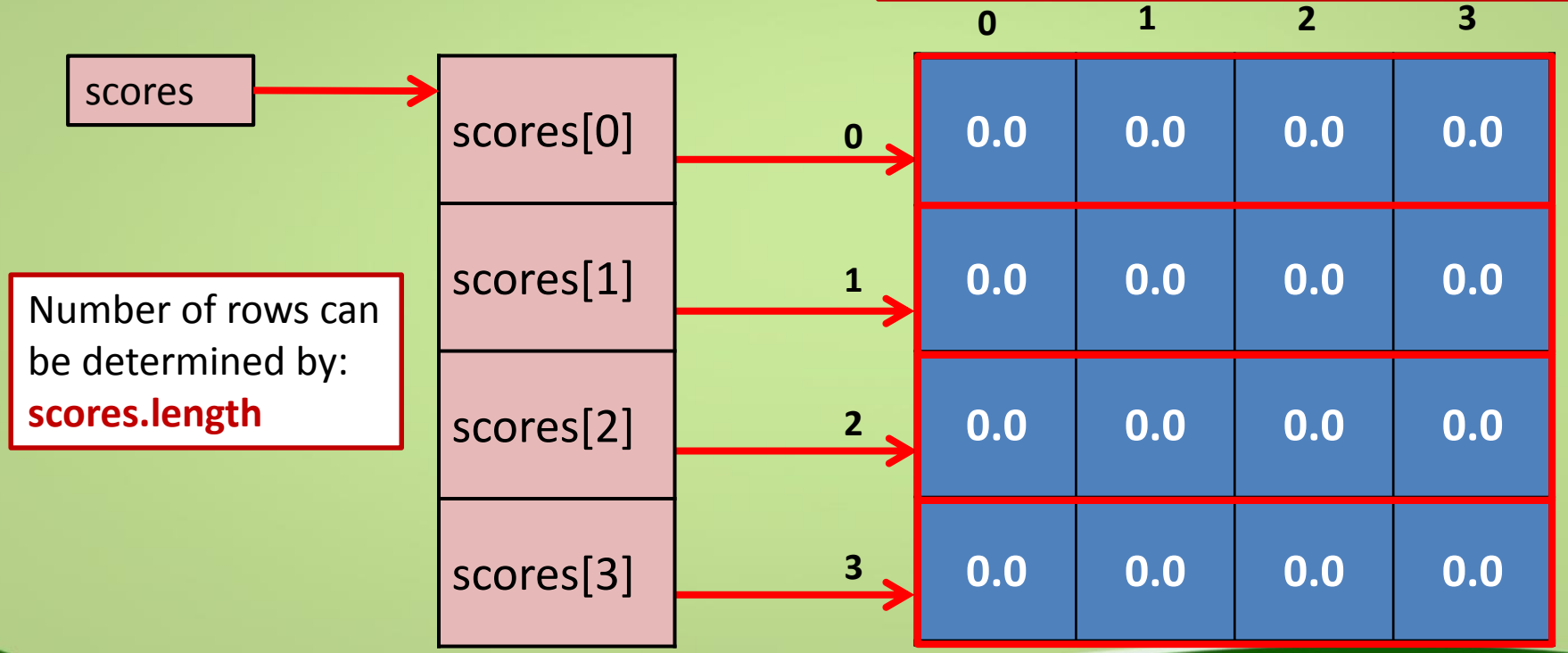
```
public double getScoreByIndices(int rowIndex, int colIndex)
{

    int numOfRows = scores.length;
    int numOfCols = scores[0].length;

}
```

Get the number of rows and the number of columns
In this example, we assume that all rows have the same number of elements

2D array as an Array of 1D arrays

- Each row can be visualized as a 1D array
  - **double[ ][ ] scores = new double[4][4];**

Number of columns can be determined by:
scores[i].length



Number of rows can be determined by:
scores.length

Access an element in a 2D array

```
public double getScoreByIndices(int rowIndex, int colIndex)
{

    int numOfRows = scores.length;
    int numOfCols = scores[0].length;

}
```

Get the number of rows and the number of columns
In this example, we assume that all rows have the same number of elements

Access an element in a 2D array

```
public double getScoreByIndices(int rowIndex, int colIndex) {
        int numOfRows = scores.length;
        int numOfCols = scores[0].length;

        if ( rowIndex < 0 || rowIndex >= numOfRows )
            return -1.0;
        if ( colIndex < 0 || colIndex >= numOfCols )
            return -1.0;

}
```

If the row index or the column index is invalid, return -1.0

Access an element in a 2D array

```
public double getScoreByIndices(int rowIndex, int colIndex) {
        int numOfRows = scores.length;
        int numOfCols = scores[0].length;

        if ( rowIndex < 0 || rowIndex >= numOfRows )
           return -1.0;
        if ( colIndex < 0 || colIndex >= numOfCols )
           return -1.0;

        return scores[rowIndex][colIndex];
}
```

Return the score located by the rowIndex (the first index) and the colIndex (the second index)

Traverse a 2D Array using a Nested Loop

```
public class Scores {
   /* 1. A 2D array instance variable */
   private double[ ][ ] scores;
   public void initializeAllScores() {   /* 2. Initialize a 2D array */    }
   public double getScoreByIndices(int rowIndex, int colIndex) {
         /* 3. Access a 2D array element */
    }

   public void printAllScores() {
         /* 4. Traverse a 2D array using a nested loop */
   }

}
```

Traversing a 2D array using a nested loop

```
public void printAllScores() {

 int numOfRows = scores.length;
 int numOfCols = scores[0].length;

}
```

Get the number of rows and the number of columns. In this example, both are 4.

double[ ][ ] fares

	0	1	2	3
0	99.0	89.0	85.0	92.0
1	90.0	74.0	75.0	82.0
2	85.0	75.0	64.0	91.0
3	72.0	82.0	81.0	94.0

Traversing a 2D array using a nested loop

```
public void printAllScores() {
 int numOfRows = scores.length;
 int numOfCols = scores[0].length;
 for ( int r=0; r<numOfRows; r++) {
   IO.output("Row " + r + " : ");
   for (int c=0; c<numOfCols; c++) {
      IO.output(getScoreByIndices(r,c) + " " );
    } // for loop c
    IO.outputln(" " );
 } // for loop r
} // end of the method
```

r = 0

c = 3

double[ ][ ] scores

	0	1	2	3
0	99.0	89.0	85.0	92.0
1	90.0	74.0	75.0	82.0
2	85.0	75.0	64.0	91.0
3	72.0	82.0	81.0	94.0

Traversing a 2D array using a nested loop

```
public void printAllScores() {
 int numOfRows = scores.length;
 int numOfCols = scores[0].length;
 for ( int r=0; r<numOfRows; r++) {
   IO.output("Row " + r + " : ");
   for (int c=0; c<numOfCols; c++) {
      IO.output(getScoreByIndices(r,c) + " " );
    } // for loop c
    IO.outputln(" " );
 } // for loop r
} // end of the method
```

r = 1

c = 3

double[ ][ ] scores

	0	1	2	3
0	99.0	89.0	85.0	92.0
1	90.0	74.0	75.0	82.0
2	85.0	75.0	64.0	91.0
3	72.0	82.0	81.0	94.0

Traversing a 2D array using a nested loop

```
public void printAllScores() {
 int numOfRows = scores.length;
 int numOfCols = scores[0].length;
 for ( int r=0; r<numOfRows; r++) {
   IO.output("Row " + r + " : ");
   for (int c=0; c<numOfCols; c++) {
      IO.output(getScoreByIndices(r,c) + " " );
    } // for loop c
    IO.outputln(" " );
 } // for loop r
}  // end of the method
```

r = 3

c = 3

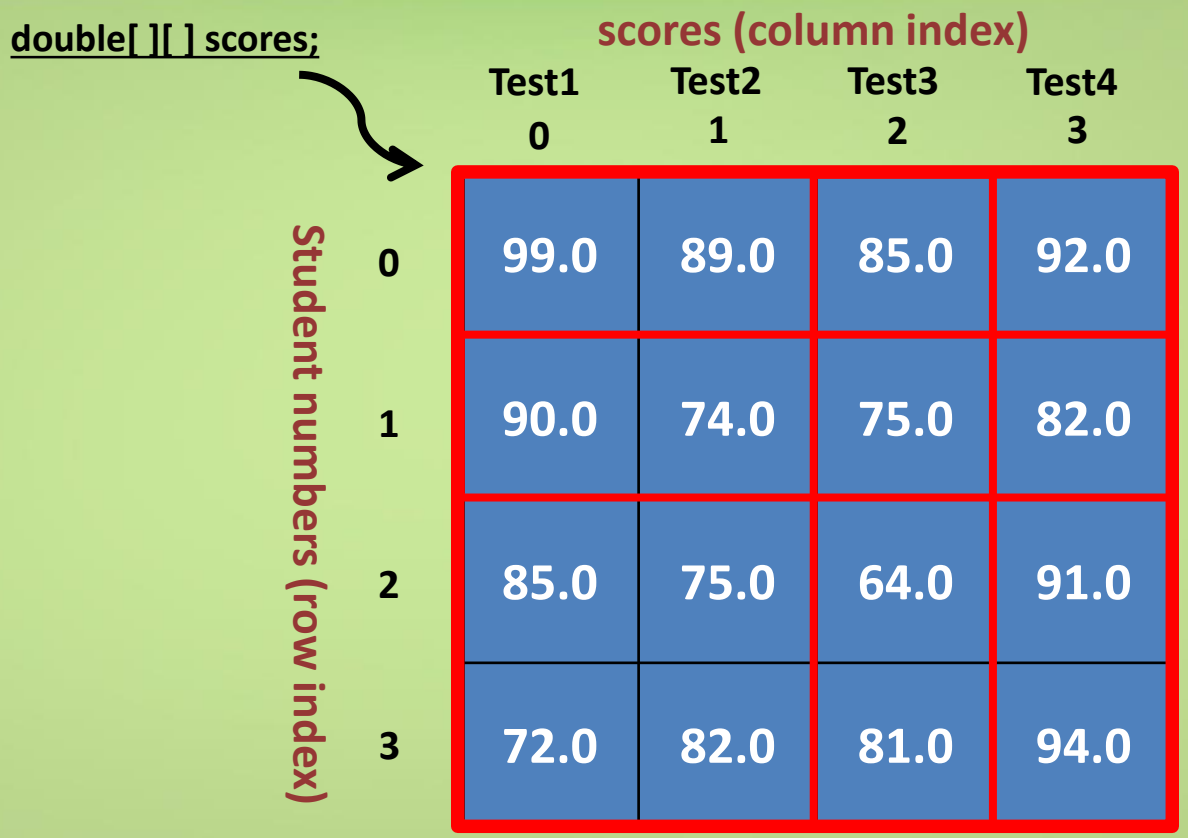
double[ ][ ] scores

	0	1	2	3
0	99.0	89.0	85.0	92.0
1	90.0	74.0	75.0	82.0
2	85.0	75.0	64.0	91.0
3	72.0	82.0	81.0	94.0

Compute Average

```
/*
 * aveScore computes the average of the values in an array
 */
public double aveScore( ) {
    double sum = 0;      // for storing the cumulative sum
    int size = scoreArray.length;  // size of the array

    for (int i = 0; i < size; i++)
       sum = sum + scoreArray[i];

    return sum / size;
  }
```

Example: Students' test scores



Compute Average by Row

```
/*
 * aveByRow computes the row average of an array
 */
public double aveByRow( int row ) {
    double sum = 0;      // for storing the cumulative sum
    int numOfCols = scores[row].length;

    for (int c = 0; c < numOfCols; c++)
      sum = sum + scores[row][c];

    return sum / numOfCols;
}
```

Compute Average by Column

```
/*
 * aveByCol computes the column average an array
 */
public double aveByCol( int col ) {
    double sum = 0;      // for storing the cumulative sum
    int numOfRows = scores.length;

    for (int r = 0; r < numOfRows; r++)
        sum = sum + scores[r][col];

    return sum / numOfRowls;
}
```

Find Maximum

```
/*
  * maxIndex finds the location of the largest values in an array
  * up to index size - 1
 */
public int maxIndex(int size){
    int mIndex = 0;   // index for the current maximum
    if (size > scoreArray.length) size = scoreArray.length;

    for (int i = 0; i < size; i++) {
       if (scoreArray[i] > scoreArray[mIndex]) mIndex = i;
     }
    return mIndex;
  }
```

Find Maximum 2D

```
/*
  * maxRowIndex finds the location of the largest values for a
  * given column in a 2D array
 */
 public int maxRowIndex(int col, int size){
     int mIndex = 0;   // index for the current maximum
     if (size > scores.length) size = scores.length;

     for (int i = 0; i < size; i++) {
         if (scores[i][col] > scores[mIndex][col]) mIndex = i;
     }
     return mIndex;
}
```

Sorting and Searching

Sorting

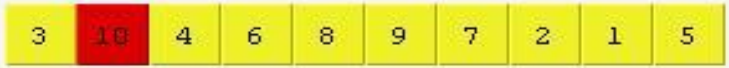
- Sorting is the process of arranging a list of items in certain order, e.g. numerical order or lexicographical order.
- Many applications require sorting:
  - To order a group of students according to their names, ID, and examinations scores.
  - To arrange a list of events in chronological order.
  - To facilitate the search for information, e.g. dictionary or phone books.
  - To display a list of webpages based on their popularity, e.g. number of hits.

Selection Sort

- Selection sort performs sorting by repeatedly finding the largest element in the unsorted portion of the array and then placing it to the end of this unsorted portion until the whole array is sorted.
- **Algorithm**
  - Define the entire array as unsorted at the beginning
  - While the unsorted portion of the array has more than one element:
    - Find its largest element
    - Swap with last element
    - Reduce the unsorted portion of the array by 1

Selection Sort - Microsoft Internet Explorer

Array



Description

Search through the array, find the largest element (10), and swap it with the last element (5), in the unprocessed portion of the array since they are different.

Note that the whole array is unprocessed at the beginning (which is in yellow).

Algorithm

1. Define the "unprocessed" portion of the array.

2. While the unprocessed portion of the array has more than one element:

2.1 Find largest element

2.2 Swap with last element if they are different

2.3 Reduce unprocessed portion of the array by 1

Change Array

Prograr

Selection Sort

```
/*
 * Use selection sort to arrange the array in ascending order
 */
public void selectSort () {
    int maxPos;  // index for the largest element in unsorted array

    for (int i = scoreArray.length-1; i > 0; i--) {
        maxPos = maxIndex(i+1);        // find the largest element
        swap (scoreArray, maxPos, i); // swap the largest and last
                                      // elements of unsorted portion
    }
}
```

Searching
and
break and continue statements

Using Break and Continue

- Two statements: **break** and **continue** can be used in all 3 types of loops
- Usage of break statement
  - Conditionally terminate and exit the loop
- Usage of continue statement
  - Conditionally skip the remaining statements in the loop body and start the next iteration

Example

```
int[ ] intArray = {90,78,100,90,65};
int value = 100;  // value to search for
int size = intArray.length;
boolean found = false;
int i;
for (i = 0; i < size; i++) {
    if (intArray[i] == value) {
            found = true;
            break;
    }
}
if (found)
    IO.outputln("The value was found at index" + i);
else
    IO.outputln{“The value was not found”);
```

Example

```
int[ ] intArray = {90,78,100,90,65};
int value = 90;  // value to search for
int size = intArray.length;
int nTimes = 0;
int i;
for (i = 0; i < size; i++) {
    if (intArray[i] != value) continue;
    // actions for each occurrence of value
    nTimes++;
}
IO.outputln("The value was found " + nTimes + " times.");
```