

# Introduction to Java 3

## Methods and Arrays

# Opening Problem

Find and display the sum of integers from 1 to 10, from 20 to 30, and from 35 to 45, respectively.

# Correct: Solution 1

```
int sum = 0;  
for (int i = 1; i <= 10; i++)  
    sum += i;
```

```
System.out.println("Sum from 1 to 10 is " + sum);
```

```
sum = 0;  
for (int i = 20; i <= 30; i++)  
    sum += i;
```

```
System.out.println("Sum from 20 to 30 is " + sum);
```

```
sum = 0;  
for (int i = 35; i <= 45; i++)  
    sum += i;
```

```
System.out.println("Sum from 35 to 45 is " + sum);
```

# Better: Solution 2

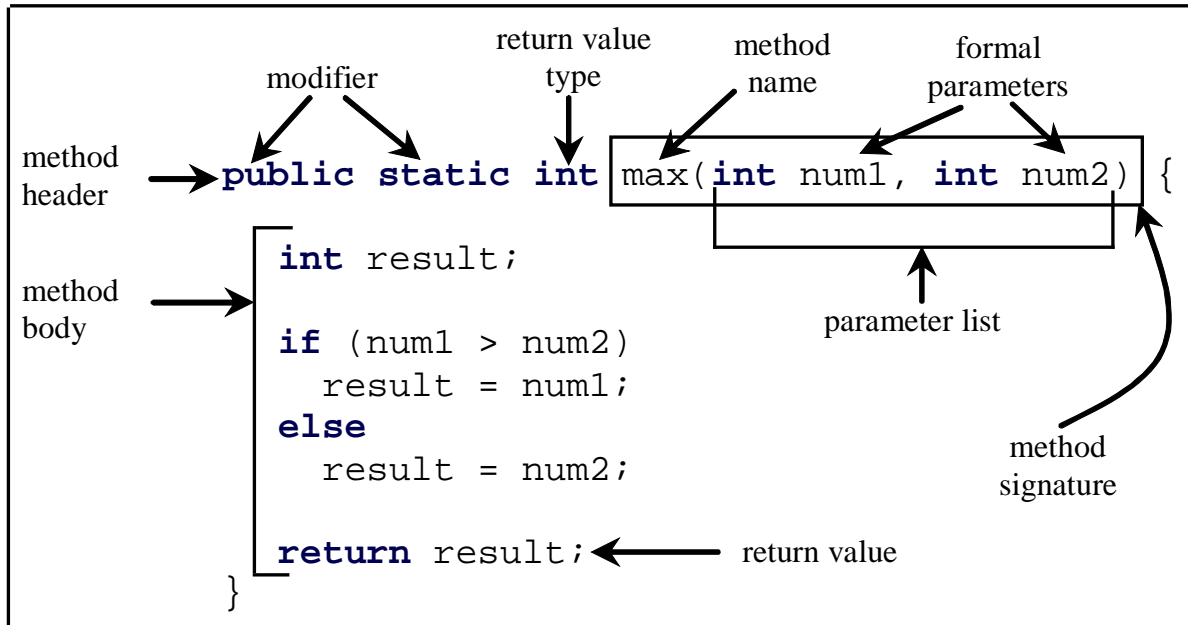
```
public static int sum(int i1, int i2) {  
    int sum = 0;  
    for (int i = i1; i <= i2; i++)  
        sum += i;  
    return sum;  
}
```

```
public static void main(String[] args) {  
    System.out.println("Sum from 1 to 10 is " + sum(1, 10));  
    System.out.println("Sum from 20 to 30 is " + sum(20, 30));  
    System.out.println("Sum from 35 to 45 is " + sum(35, 45));  
}
```

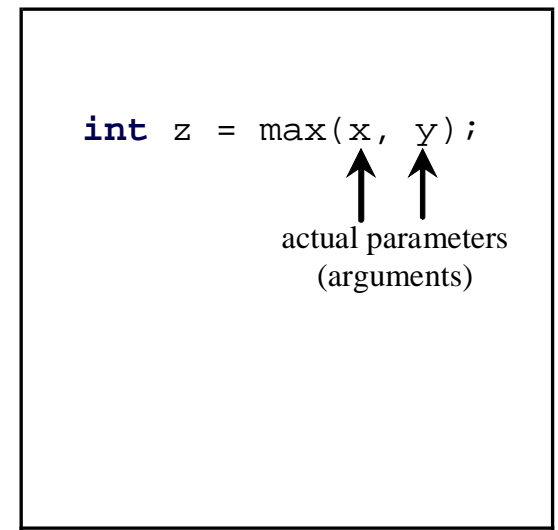
# Defining Methods

A **method** is a collection of statements that are grouped together to perform an operation.

Define a method



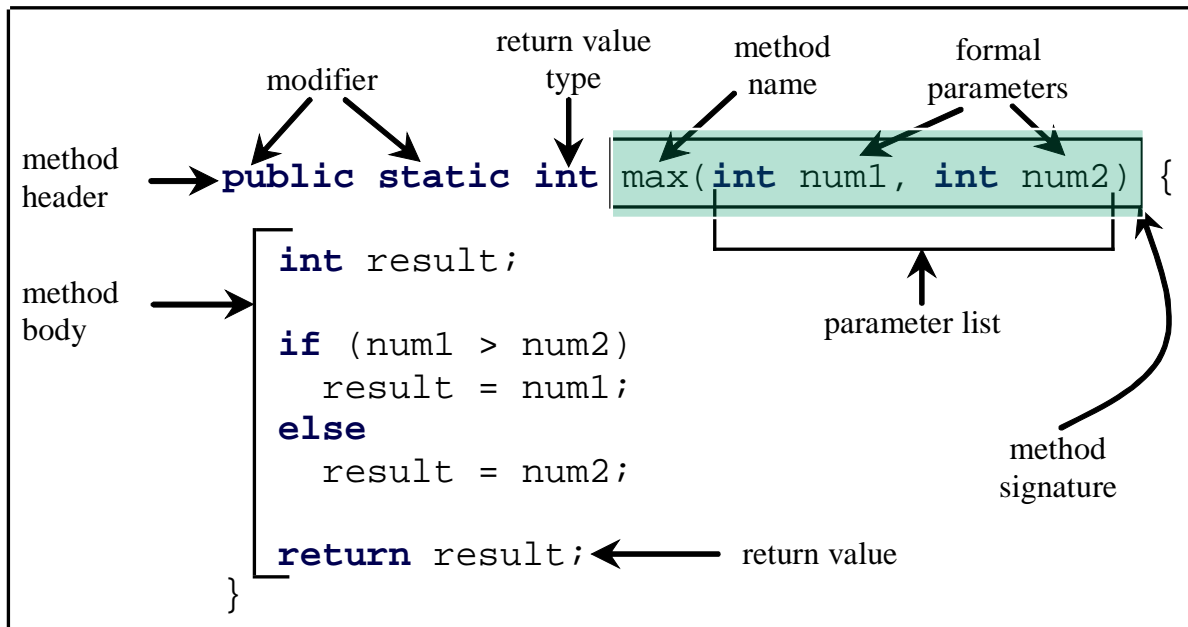
Invoke a method



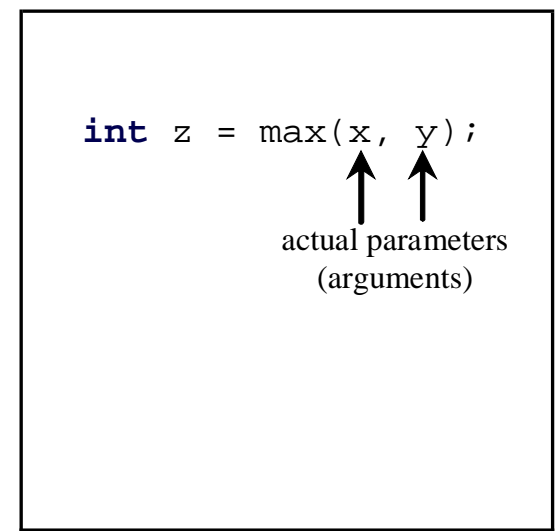
# Method Signature

*Method signature* is the combination of the method name and the parameter list.

Define a method



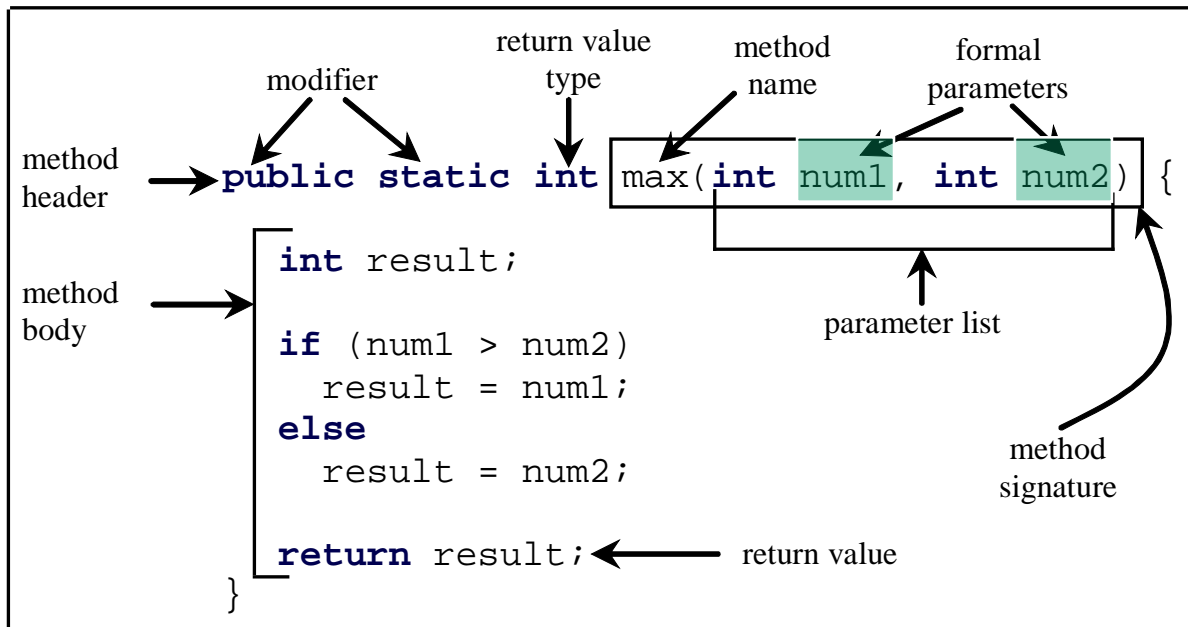
Invoke a method



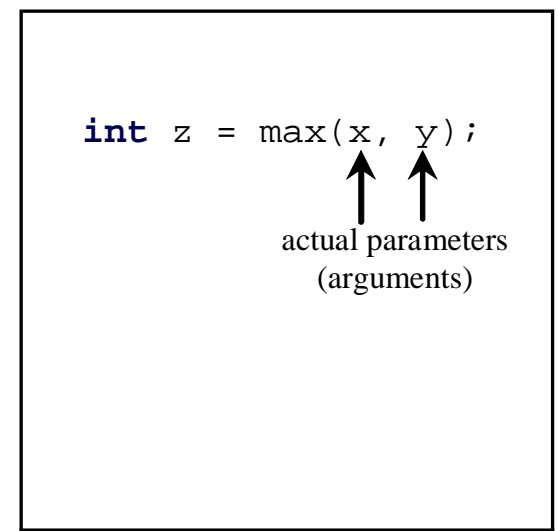
# Formal Parameters

The variables defined in the method header are known as *formal parameters*.

Define a method



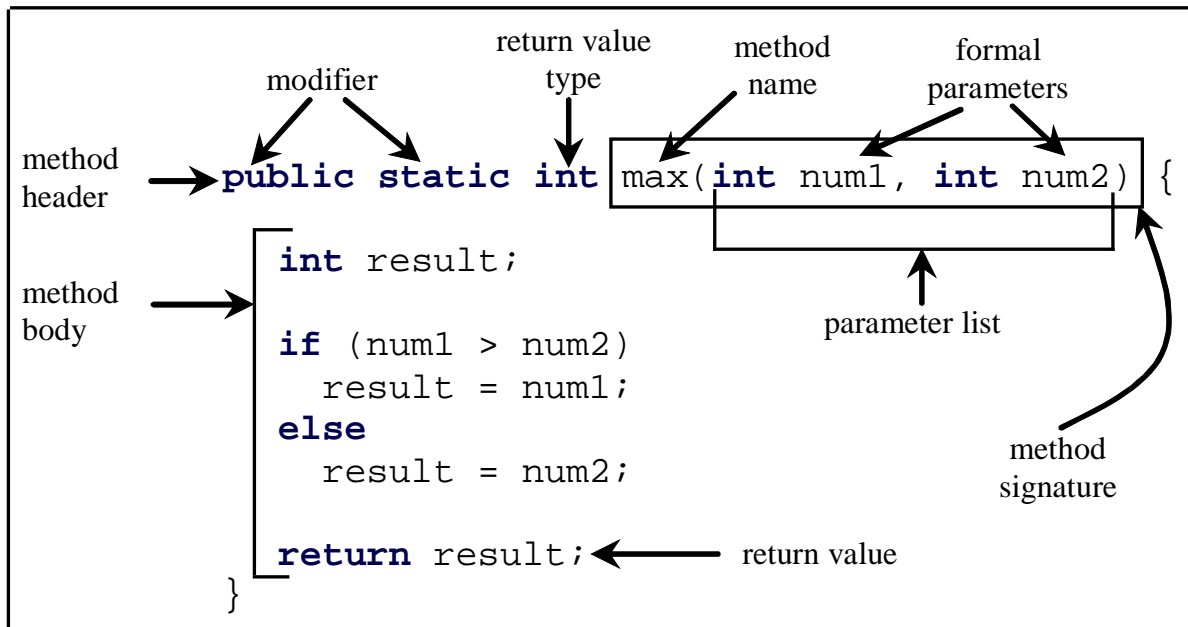
Invoke a method



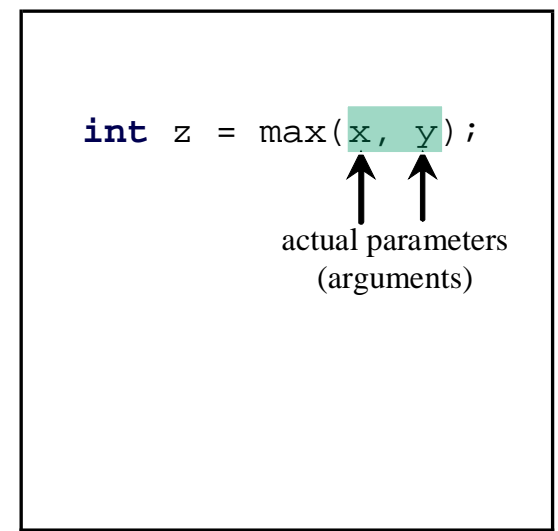
# Actual Parameters

When a method is invoked, you pass 0 or more values to the method. These values are referred to as *actual parameters or arguments*.

Define a method



Invoke a method

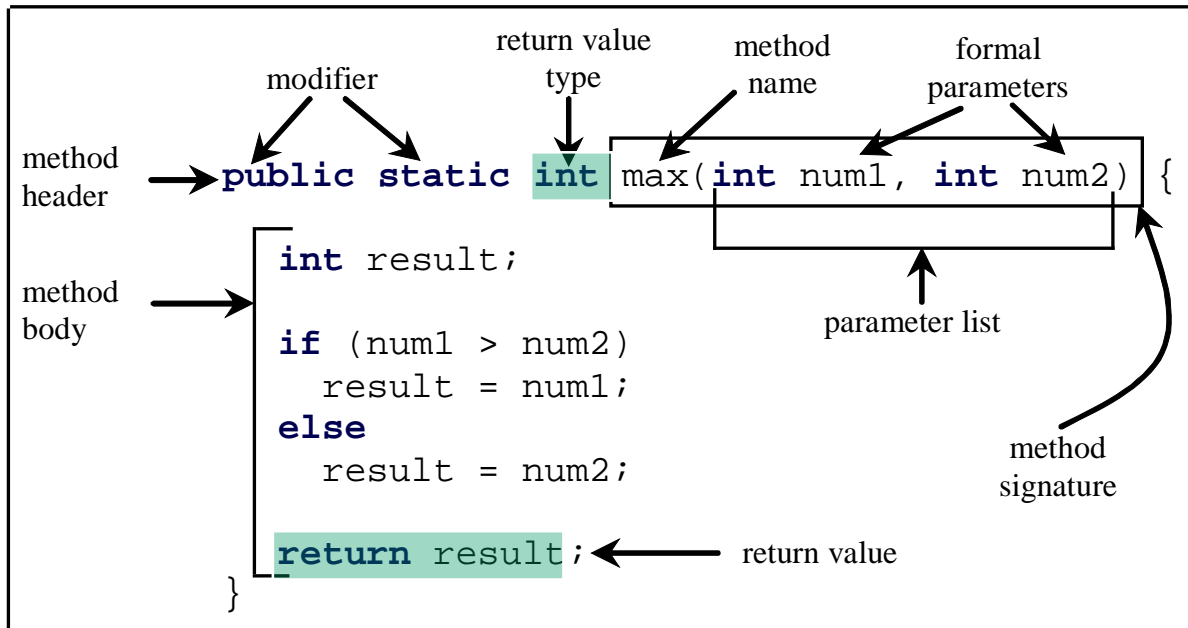




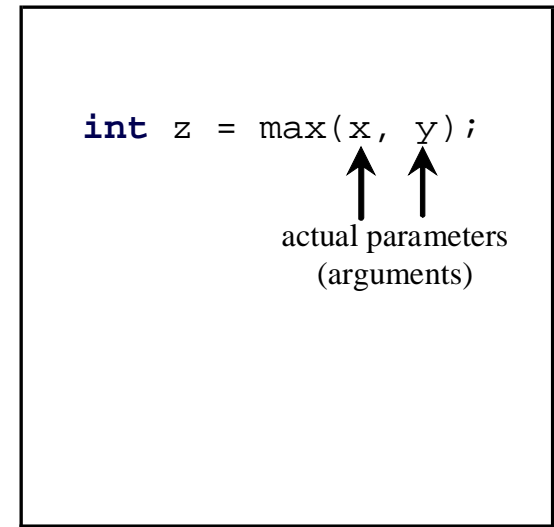
# Return Value Type

A method may return a value. The **return value type** is the data type of the value the method returns. If the method does not return a value, the **return value type** is the keyword **void**. For example, the **return value type** in the **max** method is **int**.

Define a method



Invoke a method

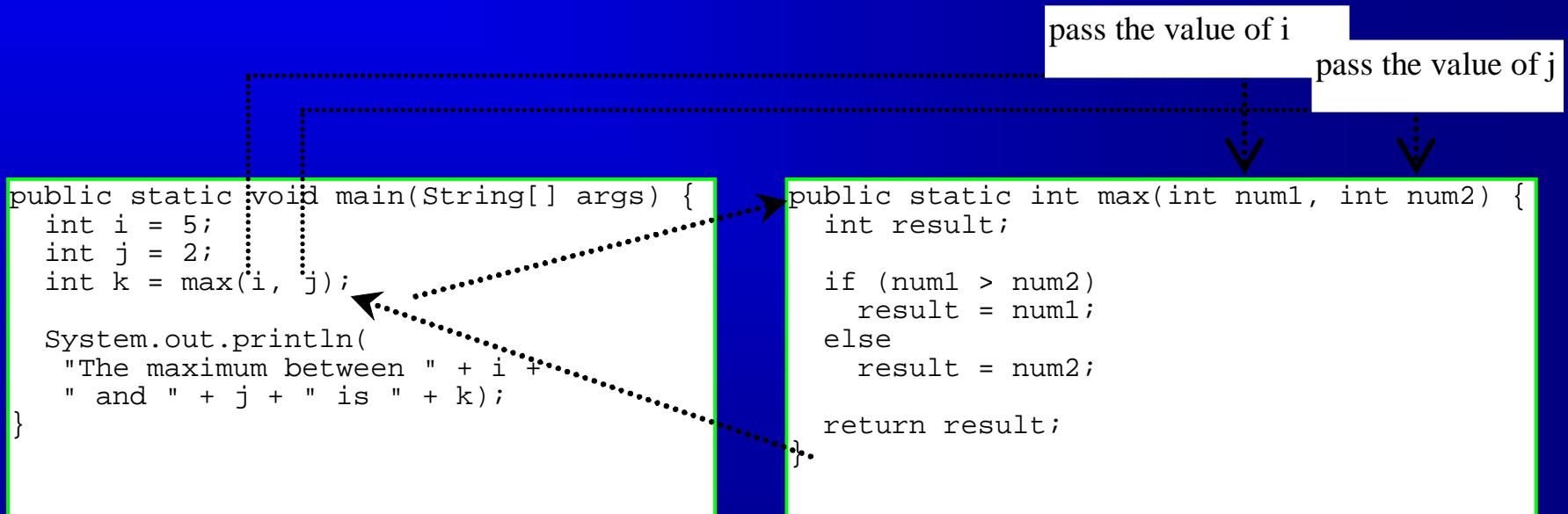


# Calling Methods

Testing the `max` method. This program demonstrates calling a method `max` to return the largest of the `int` values

```
public class TestMax {  
    /** Main method */  
    public static void main(String[] args) {  
        int i = 5;  
        int j = 2;  
        int k = max(i, j);  
        System.out.println("The maximum between " + i +  
            " and " + j + " is " + k);  
    }  
  
    /** Return the max between two numbers */  
    public static int max(int num1, int num2) {  
        int result;  
  
        if (num1 > num2)  
            result = num1;  
        else  
            result = num2;  
  
        return result;  
    }  
}
```

# Calling Methods, cont.



# CAUTION

A **return** statement is required for a value-returning method. The method shown below in (a) is logically correct, but it has a compilation error because the Java compiler thinks it possible that this method does not return any value.

```
public static int sign(int n) {  
    if (n > 0)  
        return 1;  
    else if (n == 0)  
        return 0;  
    else if (n < 0)  
        return -1;  
}
```

(a)

Should be

```
public static int sign(int n) {  
    if (n > 0)  
        return 1;  
    else if (n == 0)  
        return 0;  
    else  
        return -1;  
}
```

(b)

To fix this problem, delete *if* ( $n < 0$ ) in (a), so that the compiler will see a **return** statement to be reached regardless of how the **if** statement is evaluated.

# void Method Example

This type of method does not return a value. The method performs some actions.

```
public class TestVoidMethod {
    public static void main(String[] args) {
        System.out.print("The grade is ");
        printGrade(78.5);

        System.out.print("The grade is ");
        printGrade(59.5);
    }

    public static void printGrade(double score) {
        if (score >= 90.0) {
            System.out.println('A');
        }
        else if (score >= 80.0) {
            System.out.println('B');
        }
        else if (score >= 70.0) {
            System.out.println('C');
        }
        else if (score >= 60.0) {
            System.out.println('D');
        }
        else {
            System.out.println('F');
        }
    }
}
```

# Overloading Methods

## Overloading the `max` Method

```
public static double max(double num1, double num2){  
    if (num1 > num2)  
        return num1;  
    else  
        return num2;  
}
```

# Ambiguous Invocation

Sometimes there may be two or more possible matches for an invocation of a method, but the compiler cannot determine the most specific match. This is referred to as *ambiguous invocation*. Ambiguous invocation is a compilation error.

# Ambiguous Invocation

```
public class AmbiguousOverloading {  
    public static void main(String[] args) {  
        System.out.println(max(1, 2));  
    }  
  
    public static double max(int num1, double num2) {  
        if (num1 > num2)  
            return num1;  
        else  
            return num2;  
    }  
  
    public static double max(double num1, int num2) {  
        if (num1 > num2)  
            return num1;  
        else  
            return num2;  
    }  
}
```



# Scope of Local Variables

- A **local variable**: a variable defined inside a method (or block).
- **Scope**: the part of the program where the variable can be referenced.
- The scope of a local variable starts from its declaration and continues to the end of the block that contains the variable.
- A local variable must be declared before it can be used.

# Scope of Local Variables, cont.

You can declare a local variable with the same name multiple times in different non-nesting blocks in a method, but you cannot declare a local variable twice in nested blocks.

# Scope of Local Variables, cont.

A variable declared in the initial action part of a **for** loop header has its scope in the entire loop. But a variable declared inside a **for** loop body has its scope limited in the loop body from its declaration and to the end of the block that contains the variable.

```
public static void method1() {  
    .  
    .  
    for (int i = 1; i < 10; i++) {  
        .  
        .  
        int j;  
        .  
        .  
        .  
    }  
}
```

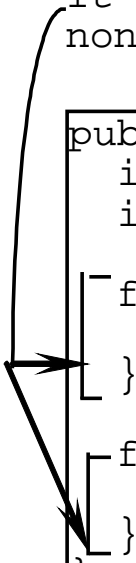
The scope of i →

The scope of j →

# Scope of Local Variables, cont.

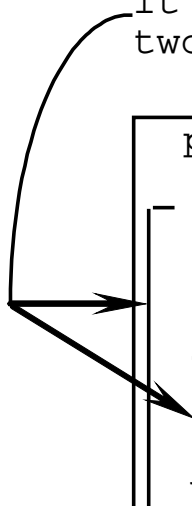
It is fine to declare `i` in two non-nesting blocks

```
public static void method1() {  
    int x = 1;  
    int y = 1;  
  
    for (int i = 1; i < 10; i++) {  
        x += i;  
    }  
  
    for (int i = 1; i < 10; i++) {  
        y += i;  
    }  
}
```



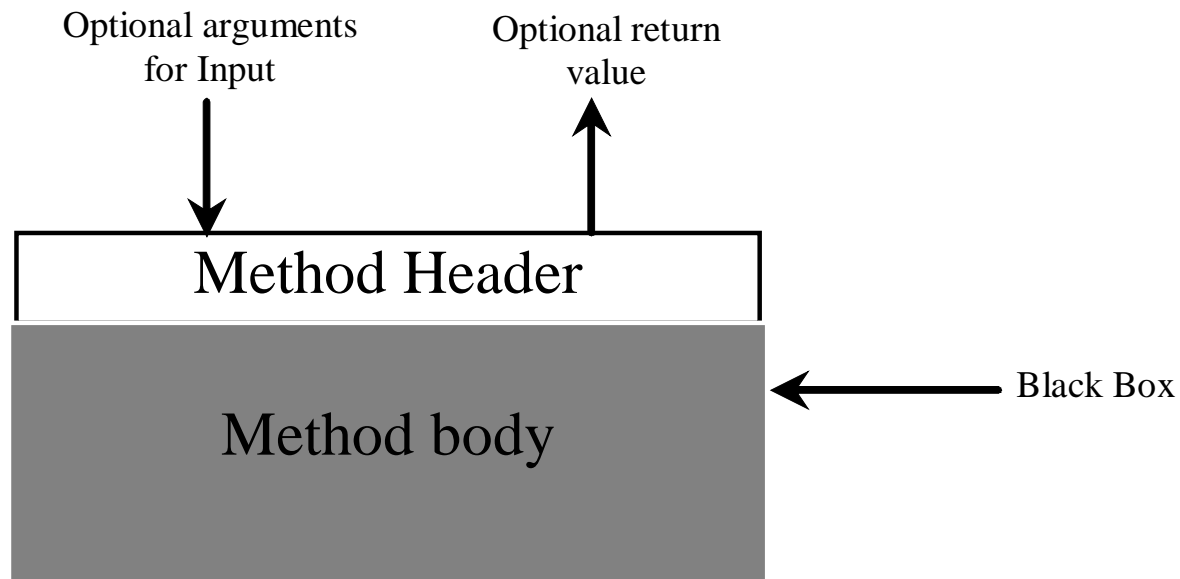
It is wrong to declare `i` in two nesting blocks

```
public static void method2() {  
    int i = 1;  
    int sum = 0;  
  
    for (int i = 1; i < 10; i++) {  
        sum += i;  
    }  
}
```



# Method Abstraction

You can think of the method body as a black box that contains the detailed implementation for the method.

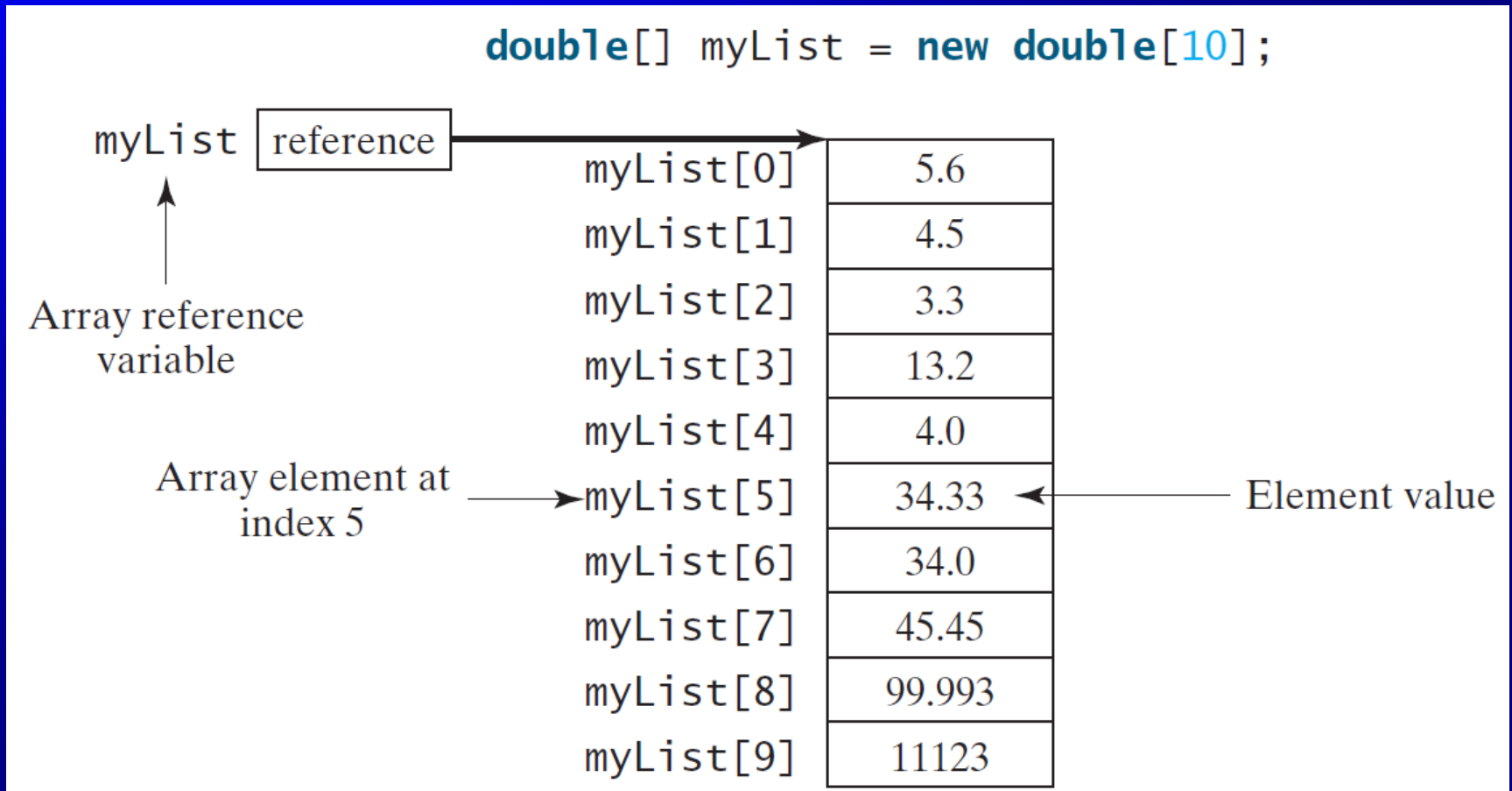


# Benefits of Methods

- Methods can be used to **reduce redundant coding** and **enable code reuse**. Write a method once and reuse it anywhere.
- **Information hiding**. Hide the implementation from the user.
- **Reduce complexity**. Methods can also be used to modularize code and improve the quality of the program.

# Introducing Arrays

Array is a data structure that represents a collection of the same types of data.



# Declaring Array Variables

- `datatype[] arrayRefVar;`

Example:

```
double[] myList;
```

- `datatype arrayRefVar[];` // This style is allowed, but not preferred

Example:

```
double myList[];
```



# Creating Arrays

```
arrayRefVar = new datatype[arraySize];
```

Example:

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

`myList[0]` references the first element in the array.

`myList[9]` references the last element in the array.

# Declaring and Creating in One Step

- `datatype[] arrayRefVar = new  
datatype[arraySize];`

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

- `datatype arrayRefVar[] = new  
datatype[arraySize];`

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

# The Length of an Array

Once an array is created, its size is fixed. It cannot be changed. You can find its size using

`arrayRefVar.length`

For example,

`myList.length` returns 10

# Default Values

When an array is created, its elements are assigned the default value of :

- 0 for the numeric primitive data types,
- '\u0000' for char types, and
- false for boolean types.

# Indexed Variables

The array elements are accessed through the index. The array indices are *0-based*, i.e., it starts from 0 to `arrayRefVar.length-1`.

Each element in the array is represented using the following syntax, known as an *indexed variable*:

```
arrayRefVar[index];
```

# Using Indexed Variables

After an array is created, an **indexed variable** can be used in the same way as a regular variable. For example, the following code adds the value in `myList[0]` and `myList[1]` to `myList[2]`.

```
myList[2] = myList[0] + myList[1];
```

# Array Initializers

Declaring, creating, initializing in one step:

```
double[] myList = {1.9, 2.9, 3.4, 3.5};
```

This shorthand syntax must be in one statement.

# Declaring, creating, initializing Using the Shorthand Notation

```
double[] myList = {1.9, 2.9, 3.4, 3.5};
```

This shorthand notation is equivalent to the following statements:

```
double[] myList = new double[4];
```

```
myList[0] = 1.9;
```

```
myList[1] = 2.9;
```

```
myList[2] = 3.4;
```

```
myList[3] = 3.5;
```



# CAUTION

Using the shorthand notation, you have to declare, create, and initialize the array all in one statement. Splitting it would cause a syntax error.

For example, the following is wrong:

```
double[] myList;
```

```
myList = {1.9, 2.9, 3.4, 3.5};
```

# Initializing arrays with input values

```
java.util.Scanner input = new java.util.Scanner(System.in);  
  
System.out.print("Enter " + myList.length + " values: ");  
  
for (int i = 0; i < myList.length; i++)  
    myList[i] = input.nextDouble();
```

# Printing arrays

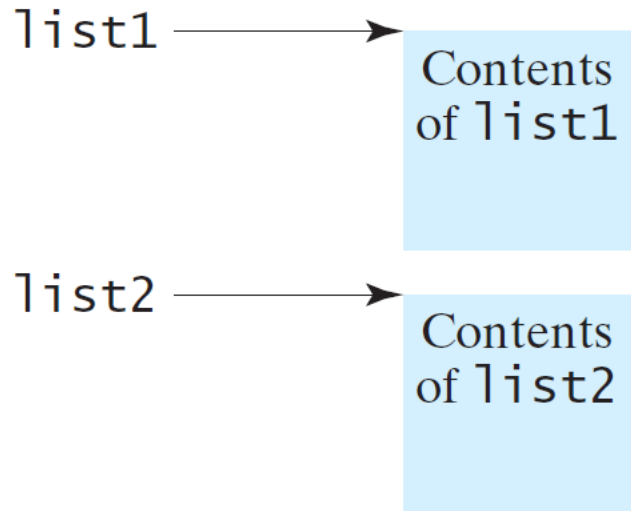
```
for (int i = 0; i < myList.length; i++) {  
    System.out.print(myList[i] + " ");  
}
```

# Copying Arrays

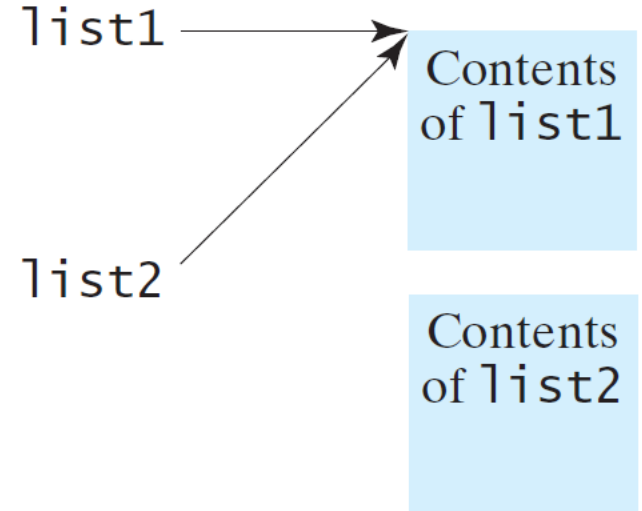
Often, in a program, you need to duplicate an array or a part of an array. In such cases you could attempt to use the assignment statement (`=`), as follows:

```
list2 = list1;
```

*Before the assignment*  
`list2 = list1;`



*After the assignment*  
`list2 = list1;`



# Copying Arrays

Using a loop:

```
int[] sourceArray = {2, 3, 1, 5, 10};  
int[] targetArray = new int[sourceArray.length];  
  
for (int i = 0; i < sourceArray.length; i++)  
    targetArray[i] = sourceArray[i];
```

# The arraycopy Utility

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

Parameters:

**src** - the source array.

**srcPos** - starting position in the source array.

**dest** - the destination array.

**destPos** - starting position in the destination data.

**length** - the number of array elements to be copied.

Example:

```
System.arraycopy(sourceArray, 0,  
                  targetArray, 0, sourceArray.length);
```

# Passing Arrays to Methods

```
public static void printArray(int[] array) {  
    for (int i = 0; i < array.length; i++) {  
        System.out.print(array[i] + " ");  
    }  
}
```

Invoke the method

```
int[] list = {3, 1, 2, 6, 4, 2};  
printArray(list);
```

Invoke the method

```
printArray(new int[]{3, 1, 2, 6, 4, 2});
```

Anonymous array

# Anonymous Array

The statement

```
printArray(new int[]{3, 1, 2, 6, 4, 2});
```

creates an array using the following syntax:

```
new dataType[]{literal0, literal1, ..., literalk};
```

There is no explicit reference variable for the array. Such array is called an *anonymous array*.



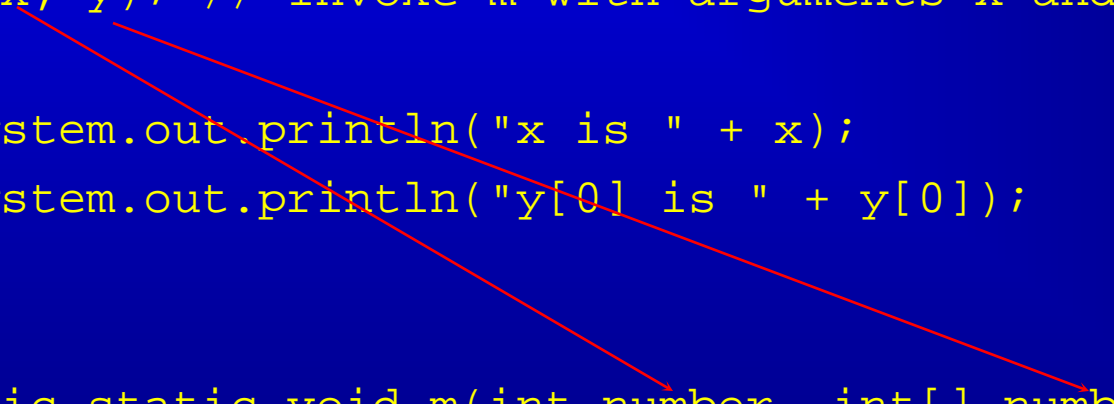
# Passing Arrays to Methods

There are important differences between passing a value of variables of primitive data types and passing arrays.

- For a parameter of a primitive type value, the **actual value** is passed. Changing the value of the local parameter inside the method does not affect the value of the variable outside the method.
- For a parameter of an array type, the value of the parameter **contains a reference** to an array; this reference is passed to the method. Any changes to the array that occur inside the method body will affect the original array that was passed as the argument.

# Simple Example

```
public class Test {  
    public static void main(String[] args) {  
        int x = 1; // x represents an int value  
        int[] y = new int[10]; // y represents an array of int values  
  
        m(x, y); // Invoke m with arguments x and y  
  
        System.out.println("x is " + x);  
        System.out.println("y[0] is " + y[0]);  
    }  
  
    public static void m(int number, int[] numbers) {  
        number = 1001; // Assign a new value to number  
        numbers[0] = 5555; // Assign a new value to numbers[0]  
    }  
}
```



# Returning an Array from a Method

```
public static int[] reverse(int[] list) {  
    int[] result = new int[list.length];  
  
    for (int i = 0, j = result.length - 1;  
        i < list.length; i++, j--) {  
        result[j] = list[i];  
    }  
  
    return result;  
}
```

```
int[] list1 = {1, 2, 3, 4, 5, 6};  
int[] list2 = reverse(list1);
```

# Multidimensional Arrays

# Motivations

Thus far, you have used one-dimensional arrays to model linear collections of elements. You can use a two-dimensional array to represent a matrix or a table. For example, the following table that describes the distances between the cities can be represented using a two-dimensional array.

Distance Table (in miles)							
	Chicago	Boston	New York	Atlanta	Miami	Dallas	Houston
Chicago	0	983	787	714	1375	967	1087
Boston	983	0	214	1102	1763	1723	1842
New York	787	214	0	888	1549	1548	1627
Atlanta	714	1102	888	0	661	781	810
Miami	1375	1763	1549	661	0	1426	1187
Dallas	967	1723	1548	781	1426	0	239
Houston	1087	1842	1627	810	1187	239	0

# Declare/Create Two-dimensional Arrays

```
// Declare array ref var
dataType[][] refVar;

// Create array and assign its reference to variable
refVar = new dataType[10][10];

// Combine declaration and creation in one statement
dataType[][] refVar = new dataType[10][10];

// Alternative syntax
dataType refVar[][] = new dataType[10][10];
```

# Declaring Variables of Two-dimensional Arrays and Creating Two-dimensional Arrays

```
int[][] matrix = new int[10][10];  
    or  
int matrix[][] = new int[10][10];  
matrix[0][0] = 3;  
  
for (int i = 0; i < matrix.length; i++)  
    for (int j = 0; j < matrix[i].length; j++)  
        matrix[i][j] = (int)(Math.random() * 1000);  
  
double[][] x;
```

# Declaring, Creating, and Initializing Using Shorthand Notations

You can also use an array initializer to declare, create and initialize a two-dimensional array. For example,

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

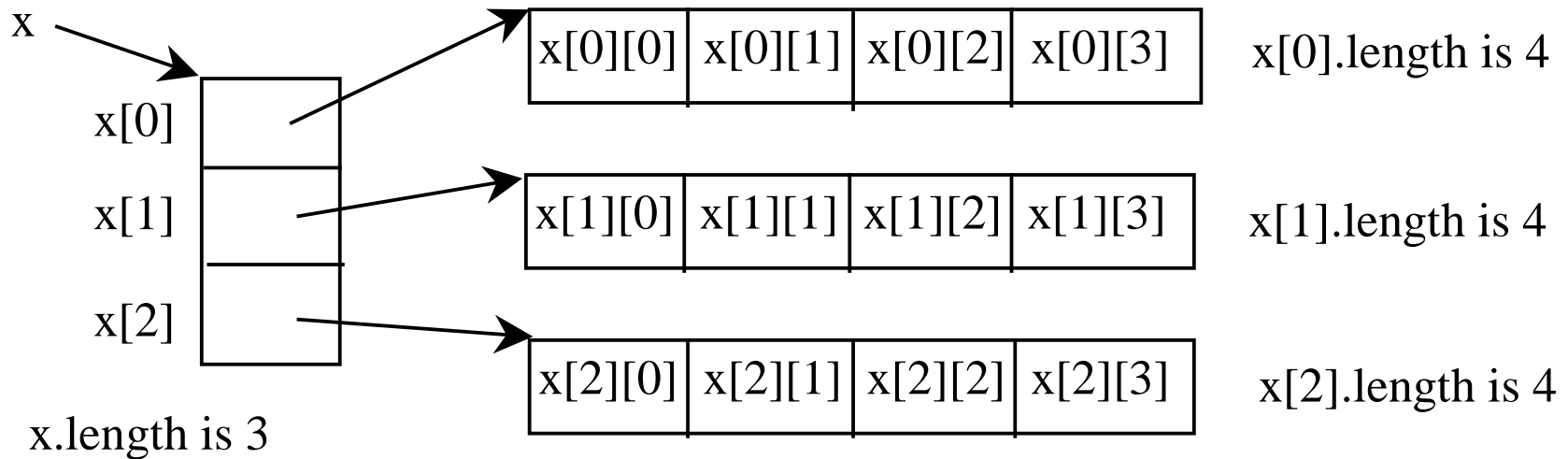
Same as

```
int[][] array = new int[4][3];  
array[0][0] = 1; array[0][1] = 2; array[0][2] = 3;  
array[1][0] = 4; array[1][1] = 5; array[1][2] = 6;  
array[2][0] = 7; array[2][1] = 8; array[2][2] = 9;  
array[3][0] = 10; array[3][1] = 11; array[3][2] = 12;
```



# Lengths of Two-dimensional Arrays

```
int[][] x = new int[3][4];
```



# Lengths of Two-dimensional Arrays, cont.

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

array.length

array[0].length

array[1].length

array[2].length

array[3].length

array[4].length

ArrayIndexOutOfBoundsException

# Ragged Arrays

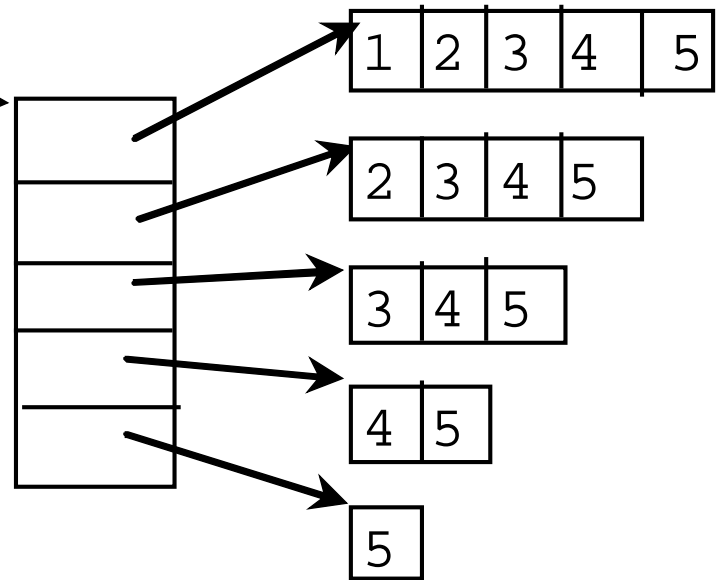
Each row in a two-dimensional array is itself an array. So, the rows can have different lengths. Such an array is known as a *ragged array*. For example,

```
int[][] matrix = {  
    {1, 2, 3, 4, 5},  
    {2, 3, 4, 5},  
    {3, 4, 5},  
    {4, 5},  
    {5}  
};
```

```
matrix.length is 5  
matrix[0].length is 5  
matrix[1].length is 4  
matrix[2].length is 3  
matrix[3].length is 2  
matrix[4].length is 1
```

# Ragged Arrays, cont.

```
int[][] triangleArray = {  
    {1, 2, 3, 4, 5},  
    {2, 3, 4, 5},  
    {3, 4, 5},  
    {4, 5},  
    {5}  
};
```



# Initializing arrays with input values

```
java.util.Scanner input = new Scanner(System.in);
```

```
System.out.println("Enter " + matrix.length + " rows and " +  
    matrix[0].length + " columns: ");
```

```
for (int row = 0; row < matrix.length; row++) {  
    for (int column = 0; column < matrix[row].length; column++) {  
        matrix[row][column] = input.nextInt();  
    }  
}
```

# Printing arrays

```
for (int row = 0; row < matrix.length; row++) {  
    for (int column = 0; column < matrix[row].length; column++) {  
        System.out.print(matrix[row][column] + " ");  
    }  
  
    System.out.println();  
}
```

# Summing All Elements

```
int total = 0;
for (int row = 0; row < matrix.length; row++) {
    for (int column = 0; column < matrix[row].length; column++) {
        total += matrix[row][column];
    }
}
```

# Summing Elements by Row

```
for (int row = 0; row < matrix.length; row++) {  
    int total = 0;  
  
    for (int column = 0; column < matrix[row].length; column++) {  
        total += matrix[row][column];  
    }  
}
```



# Summing Elements by Column

```
for (int column = 0; column < matrix[0].length; column++) {  
    int total = 0;  
    for (int row = 0; row < matrix.length; row++)  
        total += matrix[row][column];  
    System.out.println("Sum for column " + column + " is "  
        + total);  
}
```

# Passing/Returning Two-Dimensional Arrays to Methods

```
1  import java.util.Scanner;
2
3  public class PassTwoDimensionalArray {
4      public static void main(String[] args) {
5          int[][] m = getArray(); // Get an array
6
7          // Display sum of elements
8          System.out.println("\nSum of all elements is " + sum(m));
9      }
10
11     public static int[][] getArray() {
12         // Create a Scanner
13         Scanner input = new Scanner(System.in);
14
15         // Enter array values
16         int[][] m = new int[3][4];
17         System.out.println("Enter " + m.length + " rows and "
18             + m[0].length + " columns: ");
19         for (int i = 0; i < m.length; i++)
20             for (int j = 0; j < m[i].length; j++)
21                 m[i][j] = input.nextInt();
22
23         return m;
24     }
25
26     public static int sum(int[][] m) {
27         int total = 0;
28         for (int row = 0; row < m.length; row++) {
29             for (int column = 0; column < m[row].length; column++) {
30                 total += m[row][column];
31             }
32         }
33
34         return total;
35     }
36 }
```

# Multidimensional Arrays

In Java, you can create **n-dimensional** arrays for any integer **n**.

The way you declare two-dimensional array variables and create two-dimensional arrays can be generalized to declare n-dimensional array variables and create n-dimensional arrays for  $n \geq 3$ .

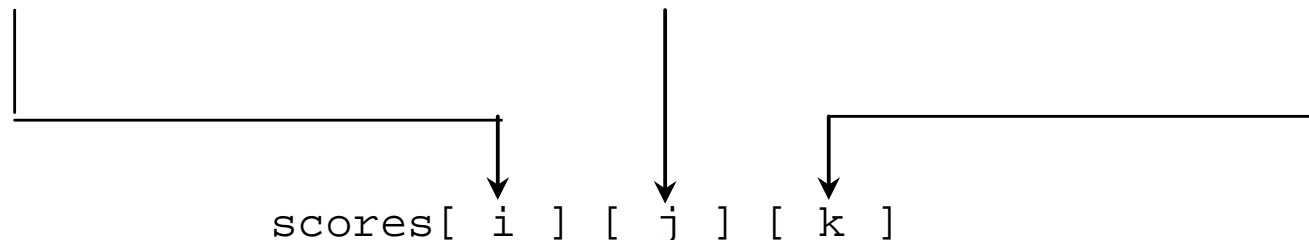
# Multidimensional Arrays

```
double[][][] scores = {  
    {{7.5, 20.5}, {9.0, 22.5}, {15, 33.5}, {13, 21.5}, {15, 2.5}},  
    {{4.5, 21.5}, {9.0, 22.5}, {15, 34.5}, {12, 20.5}, {14, 9.5}},  
    {{6.5, 30.5}, {9.4, 10.5}, {11, 33.5}, {11, 23.5}, {10, 2.5}},  
    {{6.5, 23.5}, {9.4, 32.5}, {13, 34.5}, {11, 20.5}, {16, 7.5}},  
    {{8.5, 26.5}, {9.4, 52.5}, {13, 36.5}, {13, 24.5}, {16, 2.5}},  
    {{9.5, 20.5}, {9.4, 42.5}, {13, 31.5}, {12, 20.5}, {16, 6.5}}};
```

Which student

Which exam

Multiple-choice or essay



# Programming Challenge

*(Sum the digits in an integer)* Write a method that computes the sum of the digits in an integer. Use the following method header:

```
public static int sumDigits(long n)
```

For example, `sumDigits(234)` returns `9` ( $2 + 3 + 4$ ). (*Hint:* Use the `%` operator to extract digits, and the `/` operator to remove the extracted digit. For instance, to extract 4 from 234, use `234 % 10` ( $= 4$ ). To remove 4 from 234, use `234 / 10` ( $= 23$ ). Use a loop to repeatedly extract and remove the digit until all the digits are extracted. Write a test program that prompts the user to enter an integer and displays the sum of all its digits.

# Programming Challenge

*(Palindrome integer)* Write the methods with the following headers

```
// Return the reversal of an integer, i.e., reverse(456) returns 654  
public static int reverse(int number)
```

```
// Return true if number is a palindrome  
public static boolean isPalindrome(int number)
```

Use the `reverse` method to implement `isPalindrome`. A number is a palindrome if its reversal is the same as itself. Write a test program that prompts the user to enter an integer and reports whether the integer is a palindrome.

# Programming Challenge

Create an Array with ten random numbers, find the biggest number in the array, compute their average, and find out how many numbers are above the average.

# Programming Challenge

- Generate 100 lowercase letters randomly and assign to an array of characters.
- Count and display the occurrence of each letter in the array.



# Programming Challenge

## **Chips and Salsa**

Write a program that lets a maker of chips and salsa keep track of their sales for five different types of salsa they produce: mild, medium, sweet, hot, and zesty. It should use two parallel five-element arrays: an array of strings that holds the five salsa names and an array of integers that holds the number of jars sold during the past month for each salsa type. The salsa names should be stored using an initialization list at the time the name array is created. The program should prompt the user to enter the number of jars sold for each type. Once this sales data has been entered, the program should produce a report that displays sales for each salsa type, total sales, and the names of the highest selling and lowest selling products.

*Input Validation: Do not accept negative values for number of jars sold.*