Exam2 review
Method

1. *First*, you must **define** the **method**.

   How to **define a method**:  
   
   - **Write down** the steps (= statements) contained in the **method**
   - **Attach** a **name** to the steps (= statements)

   **Notes:**
   
   - You **only** need to **define** a method **once**  
     (Remember that in **Java**, you must **define the method inside** some **class**.)

2. After **defining the method**, you can then **call a method** using the **name of the method**

   - When a **method** is **called**, the **statements** inside the corresponding method are **executed**
   - When all **statements** in the method has been **executed**, the **execution** will **resume** at the **program location** of the **method call**

   This **mechanism** is called **method invocation** (an older term is **procedure call**)

   **Note:**
   
   - You can **invoke** a method **as many times as you wish**
Defining a (class) method

- Syntax used to define a method:

```
public static RETURN-TYPE method-Name ( FORMAL-PARAMETER-LIST )
{
    (variables definitions and/or statements)
}
```

Note: the construct must appear inside some class, so it will look like this:

```
public class SomeClassName
{
    ...

    public static RETURN-TYPE method-Name ( FORMAL-PARAMETER-LIST )
    {
        (variables definitions and/or statements)
    }

    ...
}
```
Example:

```java
public class ToolBox {
    public static double min(double a, double b) {
        double m = 0;
        if (a < b) {
            m = a;
        } else {
            m = b;
        }
        return m;
    }
}
```
Multiple methods with the same name in the same class

• When you use/invoke a method in your Java program, the Java compiler will use the following information to identify which method you want to use:

  – The method name (i.e., ClassName.MethodName)
  – The number and the type of the parameters that you specify in the method invocation.
Different kinds of variables

- The different kinds of variables in a Java program
  - Java has 4 different kinds of variables (See: click here)
    - Class variables
    - Instance variables
    - Local variables
    - Parameter variables

- General information on variables:
  - Each kind of variable has its own characteristics (properties)
  - The kind of variable is determined by:
    - Where the variable is defined
    - Whether the keyword static was used in the variable definition.
Passing Parameters by Values

• When you invoke a method with an argument, the value of the argument is passed to the parameter. This is referred to as *pass-by-value*. If the argument is a variable rather than a literal value, the value of the variable is passed to the parameter. The variable is not affected, regardless of the changes made to the parameter inside the method.

• Passing Parameters by reference
The *array* data structure

- An array is a *collection* (multiple) of variables where:
  - Each variable in the collection is of the same data type
  - Note: that means that the *size* (number of bytes) of each variable is the same
  - The variables are placed (stored) *consecutively* in memory

Example:

```
How we perceive an array:  How it is stored in memory

<table>
<thead>
<tr>
<th>Array:</th>
<th>RAM memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>23 4 6 15 5 7</td>
<td>23 4 6 15 5 7</td>
</tr>
<tr>
<td>0 1 2 3 4 5</td>
<td>Consecutive memory</td>
</tr>
<tr>
<td></td>
<td>Same data type</td>
</tr>
</tbody>
</table>
```

```
Declaring Array Variables

• `elementType[] arrayRefVar;`
• Or
• `elementType arrayRefVar[];`
Creating Arrays

• Unlike declarations for primitive data type variables, the declaration of an array variable does not allocate any space in memory for the array. It creates only a storage location for the reference to an array. If a variable does not contain a reference to an array, the value of the variable is **null**. You cannot assign elements to an array unless it has already been created. After an array variable is declared, you can create an array by using the **new** operator with the following syntax:

  – `arrayRefVar = new elementType[arraySize];`
• Declaring an array variable, creating an array, and assigning the reference of the array to the variable can be combined in one statement as:
  
  – elementType[] arrayRefVar = new elementType[arraySize];

  or

  – elementType arrayRefVar[] = new elementType[arraySize];
Array Size and Default Values

• When space for an array is allocated, the array size must be given, specifying the number of elements that can be stored in it. The size of an array cannot be changed after the array is created. Size can be obtained using \texttt{arrayRefVar.length}. For example, \texttt{myList.length} is 10.

• When an array is created, its elements are assigned the default value of 0 for the numeric primitive data types, \texttt{\textbackslash u0000} for \texttt{char} types, and \texttt{false} for \texttt{boolean} types.
Array Indexed Variables

• The array elements are accessed through the index. Array indices are 0 based; that is, they range from 0 to arrayRefVar.length-1. In the example in Figure 6.1, myList holds ten double values, and the indices are from 0 to 9.
Array Initializers

- Java has a shorthand notation, known as the *array initializer*, which combines the declaration, creation, and initialization of an array in one statement using the following syntax:
  
  ```java
  elementType[] arrayRefVar = {value0, value1, ..., valuek};
  ```
  
  For example, the statement
  
  ```java
  double[] myList = {1.9, 2.9, 3.4, 3.5};
  ```
  
  declares, creates, and initializes the array `myList` with four elements, which is equivalent to
  
  the following statements:
  
  ```java
  double[] myList = new double[4]; myList[0] = 1.9;
  myList[1] = 2.9;
  myList[2] = 3.4;
  myList[3] = 3.5;
  ```

  **Caution**
  The `new` operator is not used in the array-initializer syntax. Using an array initializer, you have to declare, create, and initialize the array all in one statement. Splitting it would cause a syntax error. Thus, the next statement is wrong:

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

However, this statement does not copy the contents of the array referenced by `list1` to `list2`, but instead merely copies the reference value from `list1` to `list2`. After this statement, `list1` and `list2` reference the same array, as shown in Figure 6.5. The array previously referenced by `list2` is no longer referenced; it becomes garbage, which will be automatically collected by the Java Virtual Machine (this process is called *garbage collection*).
There are three ways to copy arrays:

- Use a loop to copy individual elements one by one.
- Use the static `arraycopy` method in the `System` class.
- Use the `clone` method to copy arrays; this will be introduced in Chapter 15, Abstract Classes and Interfaces.

You can write a loop to copy every element from the source array to the corresponding element in the target array. The following code, for instance, copies `sourceArray` to `targetArray` using a `for` loop.

```java
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];
}
```

Another approach is to use the `arraycopy` method in the `java.lang.System` class to copy arrays instead of using a loop. The syntax for `arraycopy` is:

```java
arraycopy(sourceArray, src_pos, targetArray, tar_pos, length);
```

The parameters `src_pos` and `tar_pos` indicate the starting positions in `sourceArray` and `targetArray`, respectively. The number of elements copied from `sourceArray` to `targetArray` is indicated by `length`. For example, you can rewrite the loop using the following statement:

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

The `arraycopy` method does not allocate memory space for the target array. The target array must have already been created with its memory space allocated. After the copying takes place, `targetArray` and `sourceArray` have the same content but independent memory locations.
Passing Arrays to Methods

– When passing an array to a method, the reference of the array is passed to the method.

Java uses *pass-by-value* to pass arguments to a method. There are important differences between passing the values of variables of primitive data types and passing arrays.

- For an argument of a primitive type, the argument’s value is passed.
- For an argument of an array type, the value of the argument is a reference to an array; this reference value is passed to the method. Semantically, it can be best described as *pass-by-sharing*, that is, the array in the method is the same as the array being passed. Thus, if you change the array in the method, you will see the change outside the method.

Take the following code, for example:

```java
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

• When a method returns an array, the reference of the array is returned.

```java
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;
}
```
Variable-Length Argument Lists

You can pass a variable number of arguments of the same type to a method. The parameter in the method is declared as follows:

typeName... parameterName

```java
public class VarArgsDemo {
    public static void main(String[] args) {
        printMax(34, 3, 3, 2, 56.5);
        printMax(new double[]{1, 2, 3});
    }

    public static void printMax(double... numbers) {
        if (numbers.length == 0) {
            System.out.println("No argument passed");
            return;
        }

        double result = numbers[0];
        for (int i = 1; i < numbers.length; i++)
            if (numbers[i] > result)
                result = numbers[i];

        System.out.println("The max value is " + result);
    }
}
```
Searching Arrays

• The Linear Search Approach

```java
1 public class LinearSearch {
2     /** The method for finding a key in the list */
3     public static int linearSearch(int[] list, int key) {
4         for (int i = 0; i < list.length; i++) {
5             if (key == list[i])
6                 return i;
7         }
8     return -1;
9 }
```
• The Binary Search Approach

Binary search is the other common search approach for a list of values. For binary search to work, the elements in the array must already be ordered. Assume that the array is in ascending order. The binary search first compares the key with the element in the middle of the array. Consider the following three cases:

- If the key is less than the middle element, you need to continue to search for the key only in the first half of the array.
- If the key is equal to the middle element, the search ends with a match.
- If the key is greater than the middle element, you need to continue to search for the key only in the second half of the array.
```java
public class BinarySearch {
    /** Use binary search to find the key in the list */
    public static int binarySearch(int[] list, int key) {
        int low = 0;
        int high = list.length - 1;

        while (high >= low) {
            int mid = (low + high) / 2;
            if (key < list[mid])
                high = mid - 1;
            else if (key == list[mid])
                return mid;
            else
                low = mid + 1;
        }

        return -low - 1; // Now high < low, key not found
    }
}
```
Selection sort

- Suppose that you want to sort a list in ascending order. Selection sort finds the smallest number in the list and swaps it with the first element. It then finds the smallest number remaining and swaps it with the second element, and so on, until only a single number remains. Figure 6.12 shows how to sort the list \{2, 9, 5, 4, 8, 1, 6\} using selection sort.
Select 1 (the smallest) and swap it with 2 (the first) in the list.

The number 1 is now in the correct position and thus no longer needs to be considered.

The number 2 is now in the correct position and thus no longer needs to be considered.

The number 4 is now in the correct position and thus no longer needs to be considered.

The number 5 is now in the correct position and thus no longer needs to be considered.

The number 6 is now in the correct position and thus no longer needs to be considered.

The number 8 is now in the correct position and thus no longer needs to be considered.

Select 2 (the smallest) and swap it with 9 (the first) in the remaining list.

Select 4 (the smallest) and swap it with 5 (the first) in the remaining list.

Select 6 (the smallest) and swap it with 8 (the first) in the remaining list.

Select 8 (the smallest) and swap it with 9 (the first) in the remaining list.

Since there is only one element remaining in the list, the sort is completed.
public class SelectionSort {
    /** The method for sorting the numbers */
    public static void selectionSort(double[] list) {
        for (int i = 0; i < list.length - 1; i++) {
            // Find the minimum in the list[i..list.length-1]
            double currentMin = list[i];
            int currentMinIndex = i;

            for (int j = i + 1; j < list.length; j++) {
                if (currentMin > list[j]) {
                    currentMin = list[j];
                    currentMinIndex = j;
                }
            }

            // Swap list[i] with list[currentMinIndex] if necessary
            if (currentMinIndex != i) {
                list[currentMinIndex] = list[i];
                list[i] = currentMin;
            }
        }
    }
}
Suppose that you want to sort a list in ascending order. The insertion-sort algorithm sorts a list of values by repeatedly inserting a new element into a sorted sublist until the whole list is sorted. Figure 6.13 shows how to sort the list \{2, 9, 5, 4, 8, 1, 6\} using insertion sort.
Step 1: Initially, the sorted sublist contains the first element in the list. Insert 9 into the sublist.

Step 2: The sorted sublist is \{2, 9\}. Insert 5 into the sublist.

Step 3: The sorted sublist is \{2, 5, 9\}. Insert 4 into the sublist.

Step 4: The sorted sublist is \{2, 4, 5, 9\}. Insert 8 into the sublist.

Step 5: The sorted sublist is \{2, 4, 5, 8, 9\}. Insert 1 into the sublist.

Step 6: The sorted sublist is \{1, 2, 4, 5, 8, 9\}. Insert 6 into the sublist.

Step 7: The entire list is now sorted.

**Figure 6.13** Insertion sort repeatedly inserts a new element into a sorted sublist.

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>5</td>
<td>9</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Step 1: Save 4 to a temporary variable `currentElement`

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>5</td>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Step 2: Move `list[2]` to `list[3]`

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>5</td>
<td></td>
<td>9</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Step 3: Move `list[1]` to `list[2]`

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td></td>
<td>5</td>
<td>9</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Step 4: Assign `currentElement` to `list[1]`

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>4</td>
<td>5</td>
<td>9</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 6.14** A new element is inserted into a sorted sublist.
public class InsertionSort {
    /** The method for sorting the numbers */
    public static void insertionSort(double[] list) {
        for (int i = 1; i < list.length; i++) {
            /** Insert list[i] into a sorted sublist list[0..i-1] so that list[0..i] is sorted. */
            double currentElement = list[i];
            int k;
            for (k = i - 1; k >= 0 && list[k] > currentElement; k--) {
                list[k + 1] = list[k];
            }
            // Insert the current element into list[k + 1]
            list[k + 1] = currentElement;
        }
    }
}
The Arrays Class

double[] numbers = {6.0, 4.4, 1.9, 2.9, 3.4, 3.5};
java.util.Arrays.sort(numbers); // Sort the whole array

char[] chars = {'a', 'A', '4', 'F', 'D', 'P'};
java.util.Arrays.sort(chars, 1, 3); // Sort part of the array

int[] list = {2, 4, 7, 10, 11, 45, 50, 59, 60, 66, 69, 70, 79};
System.out.println("(1) Index is " +
    java.util.Arrays.binarySearch(list, 11));

You can use the equals method to check whether two arrays are equal. Two arrays are equal if they have the same contents. In the following code, list1 and list2 are equal, but list2 and list3 are not.

int[] list1 = {2, 4, 7, 10};
int[] list2 = {2, 4, 7, 10};
int[] list3 = {4, 2, 7, 10};
System.out.println(java.util.Arrays.equals(list1, list2)); // true
System.out.println(java.util.Arrays.equals(list2, list3)); // false

You can also use the toString method to return a string that represents all elements in the array. This is a quick and simple way to display all elements in the array. For example, the following code

int[] list = {2, 4, 7, 10};
System.out.println(Arrays.toString(list));

displays [2, 4, 7, 10].
Two-Dimensional Array Basics

The syntax for declaring a two-dimensional array is:

```java
elementType[][] arrayRefVar;
```

or

```java
elementType arrayRefVar[][]; // Allowed, but not preferred
```

As an example, here is how you would declare a two-dimensional array variable `matrix` of `int` values:

```java
int[][] matrix;
```

or

```java
int matrix[][]; // This style is allowed, but not preferred
```

You can create a two-dimensional array of 5-by-5 `int` values and assign it to `matrix` using this syntax:

```java
matrix = new int[5][5];
```
<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

```
matrix = new int[5][5];
matrix[2][1] = 7;
```

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

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

(b) 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;
Obtaining the Lengths of Two-Dimensional Arrays

- A two-dimensional array is actually an array in which each element is a one-dimensional array. The length of an array $x$ is the number of elements in the array, which can be obtained using $x.length$. $x[0]$, $x[1]$, ..., and $x[x.length-1]$ are arrays. Their lengths can be obtained using $x[0].length$, $x[1].length$, ..., and $x[x.length-1].length$. 
Ragged Arrays

Each row in a two-dimensional array is itself an array. Thus, the rows can have different lengths. An array of this kind is known as a ragged array. Here is an example of creating a ragged array:

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

As you can see, `triangleArray[0].length` is 5, `triangleArray[1].length` is 4, `triangleArray[2].length` is 3, `triangleArray[3].length` is 2, and `triangleArray[4].length` is 1.

If you don’t know the values in a ragged array in advance, but do know the sizes—say, the same as before—you can create a ragged array using the following syntax:

```java
int[][] triangleArray = new int[5][];
triangleArray[0] = new int[5];
triangleArray[1] = new int[4];
triangleArray[2] = new int[3];
triangleArray[3] = new int[2];
triangleArray[4] = new int[1];
```

You can now assign values to the array. For example,

```java
triangleArray[0][3] = 50;
triangleArray[4][0] = 45;
```

**Note**
The syntax `new int[5][]` for creating an array requires the first index to be specified. The syntax `new int[] []` would be wrong.
Processing Two-Dimensional Arrays

Suppose an array `matrix` is created as follows:

```java
int[][] matrix = new int[10][10];
```

The following are some examples of processing two-dimensional arrays.

1. **Initializing arrays with input values.** The following loop initializes the array with user input values:

   ```java
   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();
       }
   }
   ```

2. **Initializing arrays with random values.** The following loop initializes the array with random values between 0 and 99:

   ```java
   for (int row = 0; row < matrix.length; row++) {
       for (int column = 0; column < matrix[row].length; column++) {
           matrix[row][column] = (int)(Math.random() * 100);
       }
   }
   ```

3. **Printing arrays.** To print a two-dimensional array, you have to print each element in the array using a loop like the following:

   ```java
   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();
   ```
Multidimensional Arrays

double[][][] scores = new double[6][5][2];

You can also use the short-hand notation to create and initialize the array as follows:

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}};
}