

เรื่องที่ 3 อาร์เรย์ สตริง และ ฟังก์ชัน ในภาษาจาวา

ENGCE174 การเขียนโปรแกรมเชิงวัตถุ (Object-oriented programming)

อ.กิตตินันท์ น้อยมณี (mr.kittinan@rmutl.ac.th)

Java Arrays

Arrays are used to store multiple values in a single variable, instead of declaring separate variables for each value.

To declare an array, define the variable type with square brackets:

```
String[] cars;
```

We have now declared a variable that holds an array of strings. To insert values to it, you can place the values in a comma-separated list, inside curly braces:

```
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
```

Java Arrays

To create an array of integers, you could write:

```
int[] myNum = {10, 20, 30, 40};
```

Access the Elements of an Array

You can access an array element by referring to the index number.

This statement accesses the value of the first element in cars:

```
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};  
System.out.println(cars[0]);  
// Outputs Volvo
```

Note: Array indexes start with 0: [0] is the first element. [1] is the second element, etc.

Change an Array Element

To change the value of a specific element, refer to the index number:

```
cars[0] = "Opel";
```

```
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};  
cars[0] = "Opel";  
System.out.println(cars[0]);  
// Now outputs Opel instead of Volvo
```

Array Length

To find out how many elements an array has, use the `length` property:

```
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};  
System.out.println(cars.length);  
// Outputs 4
```

Test Yourself With Exercises

Create an array of type `String` called `cars`.

```
String[] cars = {"Volvo", "BMW", "Ford"};
```

Loop Through an Array

You can loop through the array elements with the **for** loop, and use the **length** property to specify how many times the loop should run.

The following example outputs all elements in the cars array:

```
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
for (int i = 0; i < cars.length; i++) {
    System.out.println(cars[i]);
}
```


Loop Through an Array with For-Each

There is also a "for-each" loop, which is used exclusively to loop through elements in arrays:

Syntax

```
for (type variable : arrayname) {  
    ...  
}
```

Loop Through an Array with For-Each

The following example outputs all elements in the cars array, using a "for-each" loop:

```
String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
for (String i : cars) {
    System.out.println(i);
}
```

Loop Through an Array with For-Each

The example above can be read like this: for each **String** element (called i - as in index) in cars, print out the value of i.

If you compare the **for** loop and for-each loop, you will see that the for-each method is easier to write, it does not require a counter (using the length property), and it is more readable.

Test Yourself With Exercises

Loop through the items in the `cars` array.

```
String[] cars = {"Volvo", "BMW", "Ford"};
for (String i : cars) {
    System.out.println(i);
}
```

Multidimensional Arrays

A multidimensional array is an array of arrays.

To create a two-dimensional array, add each array within its own set of curly braces:

```
int[][] myNumbers = { {1, 2, 3, 4}, {5, 6, 7} };
```

Multidimensional Arrays

myNumbers is now an array with two arrays as its elements.

To access the elements of the myNumbers array, specify two indexes: one for the array, and one for the element inside that array. This example accesses the third element (2) in the second array (1) of myNumbers:

```
int[][] myNumbers = { {1, 2, 3, 4}, {5, 6, 7} };  
int x = myNumbers[1][2];  
System.out.println(x); // Outputs 7
```

Multidimensional Arrays

We can also use a **for loop** inside another **for loop** to get the elements of a two-dimensional array (we still have to point to the two indexes):

```
public class Main {  
    public static void main(String[] args) {  
        int[][] myNumbers = { {1, 2, 3, 4}, {5, 6, 7} };  
        for (int i = 0; i < myNumbers.length; ++i) {  
            for(int j = 0; j < myNumbers[i].length; ++j) {  
                System.out.println(myNumbers[i][j]);  
            }  
        }  
    }  
}
```

Test Yourself With Exercises

Insert the missing part to create a two-dimensional array.

```
 myNumbers = { {1, 2, 3, 4}, {5, 6, 7} };
```


Java Methods

A method is a block of code which only runs when it is called.

You can pass data, known as parameters, into a method.

Methods are used to perform certain actions, and they are also known as functions.

Why use methods? To reuse code: define the code once, and use it many times.

Create a Method

A method must be declared within a class. It is defined with the name of the method, followed by parentheses (). Java provides some pre-defined methods, such as `System.out.println()`, but you can also create your own methods to perform certain actions:

- `myMethod()` is the name of the method
- `static` means that the method belongs to the Main class and not an object of the Main class. You will learn more about objects and how to access methods through objects later in this tutorial.
- `void` means that this method does not have a return value. You will learn more about return values later in this chapter

```
public class Main {  
    static void myMethod() {  
        // code to be executed  
    }  
}
```

Call a Method

To call a method in Java, write the method's name followed by two parentheses () and a semicolon;

In the following example, `myMethod()` is used to print a text (the action), when it is called:

Inside `main`, call the `myMethod()` method:

```
public class Main {  
    static void myMethod() {  
        System.out.println("I just got executed!");  
    }  
  
    public static void main(String[] args) {  
        myMethod();  
    }  
}  
  
// Outputs "I just got executed!"
```

Call a Method

A method can also be called multiple times:

In the next chapter, [Method Parameters](#), you will learn how to pass data (parameters) into a method.

```
public class Main {  
    static void myMethod() {  
        System.out.println("I just got executed!");  
    }  
  
    public static void main(String[] args) {  
        myMethod();  
        myMethod();  
        myMethod();  
    }  
}  
  
// I just got executed!  
// I just got executed!  
// I just got executed!
```

Test Yourself With Exercises

Insert the missing part to call `myMethod` from `main`.

```
static void myMethod() {  
    System.out.println("I just got executed!");  
}  
  
public static void main(String[] args) {  
    ;  
}
```

Parameters and Arguments

Information can be passed to methods as parameter. Parameters act as variables inside the method.

Parameters are specified after the method name, inside the parentheses. You can add as many parameters as you want, just separate them with a comma.

The following example has a method that takes a `String` called `fname` as parameter. When the method is called, we pass along a first name, which is used inside the method to print the full name:

When a parameter is passed to the method, it is called an argument. So, from the example above: `fname` is a parameter, while `Liam`, `Jenny` and `Anja` are arguments.

```
public class Main {  
    static void myMethod(String fname) {  
        System.out.println(fname + " Refsnes");  
    }  
  
    public static void main(String[] args) {  
        myMethod("Liam");  
        myMethod("Jenny");  
        myMethod("Anja");  
    }  
}  
  
// Liam Refsnes  
// Jenny Refsnes  
// Anja Refsnes
```

Multiple Parameters

You can have as many parameters as you like:

Note that when you are working with multiple parameters, the method call must have the same number of arguments as there are parameters, and the arguments must be passed in the same order.

```
public class Main {  
    static void myMethod(String fname, int age) {  
        System.out.println(fname + " is " + age);  
    }  
  
    public static void main(String[] args) {  
        myMethod("Liam", 5);  
        myMethod("Jenny", 8);  
        myMethod("Anja", 31);  
    }  
}  
  
// Liam is 5  
// Jenny is 8  
// Anja is 31
```

Return Values

The **void** keyword, used in the examples above, indicates that the method should not return a value. If you want the method to return a value, you can use a primitive data type (such as **int**, **char**, etc.) instead of **void**, and use the **return** keyword inside the method:

```
public class Main {  
    static int myMethod(int x) {  
        return 5 + x;  
    }  
  
    public static void main(String[] args) {  
        System.out.println(myMethod(3));  
    }  
}  
  
// Outputs 8 (5 + 3)
```


Return Values

This example returns the sum of a method's two parameters:

```
public class Main {  
    static int myMethod(int x, int y) {  
        return x + y;  
    }  
  
    public static void main(String[] args) {  
        System.out.println(myMethod(5, 3));  
    }  
}  
  
// Outputs 8 (5 + 3)
```

Return Values

You can also store the result in a variable (recommended, as it is easier to read and maintain):

```
public class Main {  
    static int myMethod(int x, int y) {  
        return x + y;  
    }  
  
    public static void main(String[] args) {  
        int z = myMethod(5, 3);  
        System.out.println(z);  
    }  
}  
  
// Outputs 8 (5 + 3)
```

A Method with If...Else

It is common to use **if...else** statements inside methods:

```
public class Main {  
  
    // Create a checkAge() method with an integer variable called age  
    static void checkAge(int age) {  
  
        // If age is less than 18, print "access denied"  
        if (age < 18) {  
            System.out.println("Access denied - You are not old enough!");  
  
            // If age is greater than, or equal to, 18, print "access granted"  
        } else {  
            System.out.println("Access granted - You are old enough!");  
        }  
    }  
  
    }  
  
    public static void main(String[] args) {  
        checkAge(20); // Call the checkAge method and pass along an age of 20  
    }  
}  
  
// Outputs "Access granted - You are old enough!"
```

Test Yourself With Exercises

Add a `fname` parameter of type `String` to `myMethod`, and output "John Doe":

```
static void myMethod(   ) {  
    System.out.println(  + " Doe" );  
}  
  
public static void main(String[] args) {  
    myMethod( "John" );  
}
```

Method Overloading

With method overloading, multiple methods can have the same name with different parameters:

```
int myMethod(int x)
float myMethod(float x)
double myMethod(double x, double y)
```

Method Overloading

Consider the following example, which has two methods that add numbers of different type:

```
static int plusMethodInt(int x, int y) {  
    return x + y;  
}  
  
static double plusMethodDouble(double x, double y) {  
    return x + y;  
}  
  
public static void main(String[] args) {  
    int myNum1 = plusMethodInt(8, 5);  
    double myNum2 = plusMethodDouble(4.3, 6.26);  
    System.out.println("int: " + myNum1);  
    System.out.println("double: " + myNum2);  
}
```

Method Overloading

Instead of defining two methods that should do the same thing, it is better to overload one.

In the example below, we overload the `plusMethod` method to work for both `int` and `double`:

Note: Multiple methods can have the same name as long as the number and/or type of parameters are different.

```
static int plusMethod(int x, int y) {  
    return x + y;  
}
```

```
static double plusMethod(double x, double y) {  
    return x + y;  
}
```

```
public static void main(String[] args) {  
    int myNum1 = plusMethod(8, 5);  
    double myNum2 = plusMethod(4.3, 6.26);  
    System.out.println("int: " + myNum1);  
    System.out.println("double: " + myNum2);  
}
```

Java Scope

In Java, variables are only accessible inside the region they are created. This is called scope.

Method Scope

Variables declared directly inside a method are available anywhere in the method following the line of code in which they were declared:

```
public class Main {  
    public static void main(String[] args) {  
  
        // Code here CANNOT use x  
  
        int x = 100;  
  
        // Code here can use x  
        System.out.println(x);  
    }  
}
```

Block Scope

A block of code refers to all of the code between curly braces `{}`.

Variables declared inside blocks of code are only accessible by the code between the curly braces, which follows the line in which the variable was declared:

A block of code may exist on its own or it can belong to an `if`, `while` or `for` statement. In the case of `for` statements, variables declared in the statement itself are also available inside the block's scope.

```
public class Main {  
    public static void main(String[] args) {  
  
        // Code here CANNOT use x  
  
        { // This is a block  
  
            // Code here CANNOT use x  
  
            int x = 100;  
  
            // Code here CAN use x  
            System.out.println(x);  
  
        } // The block ends here  
  
        // Code here CANNOT use x  
  
    }  
}
```

Java Recursion

Recursion is the technique of making a function call itself. This technique provides a way to break complicated problems down into simple problems which are easier to solve.

Recursion may be a bit difficult to understand. The best way to figure out how it works is to experiment with it.

Recursion Example

Adding two numbers together is easy to do, but adding a range of numbers is more complicated. In the following example, recursion is used to add a range of numbers together by breaking it down into the simple task of adding two numbers:

Use recursion to add all of the numbers up to 10.

```
public class Main {  
    public static void main(String[] args) {  
        int result = sum(10);  
        System.out.println(result);  
    }  
    public static int sum(int k) {  
        if (k > 0) {  
            return k + sum(k - 1);  
        } else {  
            return 0;  
        }  
    }  
}
```

55

Example Explained

When the `sum()` function is called, it adds parameter `k` to the sum of all numbers smaller than `k` and returns the result. When `k` becomes 0, the function just returns 0. When running, the program follows these steps:

```
10 + sum(9)
10 + ( 9 + sum(8) )
10 + ( 9 + ( 8 + sum(7) ) )
...
10 + 9 + 8 + 7 + 6 + 5 + 4 + 3 + 2 + 1 + sum(0)
10 + 9 + 8 + 7 + 6 + 5 + 4 + 3 + 2 + 1 + 0
```

Since the function does not call itself when `k` is 0, the program stops there and returns the result.

Halting Condition

Just as loops can run into the problem of infinite looping, recursive functions can run into the problem of infinite recursion. Infinite recursion is when the function never stops calling itself. Every recursive function should have a halting condition, which is the condition where the function stops calling itself. In the previous example, the halting condition is when the parameter **k** becomes 0.

It is helpful to see a variety of different examples to better understand the concept. In this example, the function adds a range of numbers between a start and an end. The halting condition for this recursive function is when end is not greater than start:

Halting Condition

Use recursion to add all of the numbers between 5 to 10.

The developer should be very careful with recursion as it can be quite easy to slip into writing a function which never terminates, or one that uses excess amounts of memory or processor power. However, when written correctly recursion can be a very efficient and mathematically-elegant approach to programming.

```
public class Main {  
    public static void main(String[] args) {  
        int result = sum(5, 10);  
        System.out.println(result);  
    }  
    public static int sum(int start, int end) {  
        if (end > start) {  
            return end + sum(start, end - 1);  
        } else {  
            return end;  
        }  
    }  
}
```

45

Java ArrayList

The `ArrayList` class is a resizable [array](#), which can be found in the `java.util` package.

The difference between a built-in array and an `ArrayList` in Java, is that the size of an array cannot be modified (if you want to add or remove elements to/from an array, you have to create a new one). While elements can be added and removed from an `ArrayList` whenever you want. The syntax is also slightly different:

Create an `ArrayList` object called `cars` that will store strings:

```
import java.util.ArrayList; // import the ArrayList class

ArrayList<String> cars = new ArrayList<String>(); // Create an ArrayList object
```


Add Items

The `ArrayList` class has many useful methods. For example, to add elements to the `ArrayList`, use the `add()` method:

```
import java.util.ArrayList;

public class Main {
    public static void main(String[] args) {
        ArrayList<String> cars = new ArrayList<String>();
        cars.add("Volvo");
        cars.add("BMW");
        cars.add("Ford");
        cars.add("Mazda");
        System.out.println(cars);
    }
}
```

Access an Item

To access an element in the `ArrayList`, use the `get()` method and refer to the index number:



```
cars.get(0);
```

Remember: Array indexes start with 0: `[0]` is the first element. `[1]` is the second element, etc.

Change an Item

To modify an element, use the `set()` method and refer to the index number:

```
cars.set(0, "Opel");
```

```
import java.util.ArrayList;

public class Main {
    public static void main(String[] args) {
        ArrayList<String> cars = new ArrayList<String>();
        cars.add("Volvo");
        cars.add("BMW");
        cars.add("Ford");
        cars.add("Mazda");
        cars.set(0, "Opel");
        System.out.println(cars);
    }
}
```

```
[Opel, BMW, Ford, Mazda]
```

Remove an Item

To remove an element, use the `remove()` method and refer to the index number:

```
cars.remove(0);
```

```
import java.util.ArrayList;
```

```
public class Main {  
    public static void main(String[] args) {  
        ArrayList<String> cars = new ArrayList<String>();  
        cars.add("Volvo");  
        cars.add("BMW");  
        cars.add("Ford");  
        cars.add("Mazda");  
        cars.remove(0);  
        System.out.println(cars);  
    }  
}
```

[BMW, Ford, Mazda]

Remove an Item

To remove all the elements in the `ArrayList`, use the `clear()` method:

```
cars.clear();
```

```
import java.util.ArrayList;

public class Main {
    public static void main(String[] args) {
        ArrayList<String> cars = new ArrayList<String>();
        cars.add("Volvo");
        cars.add("BMW");
        cars.add("Ford");
        cars.add("Mazda");
        cars.clear();
        System.out.println(cars);
    }
}
```



```
[]
```

ArrayList Size

To find out how many elements an ArrayList have, use the **size** method:

```
cars.size();

import java.util.ArrayList;

public class Main {
    public static void main(String[] args) {
        ArrayList<String> cars = new ArrayList<String>();
        cars.add("Volvo");
        cars.add("BMW");
        cars.add("Ford");
        cars.add("Mazda");
        System.out.println(cars.size());
    }
}
```

Loop Through an ArrayList

Loop through the elements of an `ArrayList` with a `for` loop, and use the `size()` method to specify how many times the loop should run:

```
public class Main {  
    public static void main(String[] args) {  
        ArrayList<String> cars = new ArrayList<String>();  
        cars.add("Volvo");  
        cars.add("BMW");  
        cars.add("Ford");  
        cars.add("Mazda");  
        for (int i = 0; i < cars.size(); i++) {  
            System.out.println(cars.get(i));  
        }  
    }  
}
```

Loop Through an ArrayList

You can also loop through an `ArrayList` with the for-each loop:

```
public class Main {  
    public static void main(String[] args) {  
        ArrayList<String> cars = new ArrayList<String>();  
        cars.add("Volvo");  
        cars.add("BMW");  
        cars.add("Ford");  
        cars.add("Mazda");  
        for (String i : cars) {  
            System.out.println(i);  
        }  
    }  
}
```


Other Types

Elements in an `ArrayList` are actually objects. In the examples above, we created elements (objects) of type "String". Remember that a String in Java is an object (not a primitive type). To use other types, such as `int`, you must specify an equivalent [wrapper class](#):

`Integer`. For other primitive types, use: `Boolean` for boolean, `Character` for char, `Double` for double, etc:

Create an `ArrayList` to store numbers (add elements of type `Integer`):

```
import java.util.ArrayList;

public class Main {
    public static void main(String[] args) {
        ArrayList<Integer> myNumbers = new ArrayList<Integer>();
        myNumbers.add(10);
        myNumbers.add(15);
        myNumbers.add(20);
        myNumbers.add(25);
        for (int i : myNumbers) {
            System.out.println(i);
        }
    }
}
```


Sort an ArrayList

Another useful class in the `java.util` package is the `Collections` class, which include the `sort()` method for sorting lists alphabetically or numerically:

Sort an ArrayList of Strings:

```
import java.util.ArrayList;
import java.util.Collections; // Import the Collections class

public class Main {
    public static void main(String[] args) {
        ArrayList<String> cars = new ArrayList<String>();
        cars.add("Volvo");
        cars.add("BMW");
        cars.add("Ford");
        cars.add("Mazda");
        Collections.sort(cars); // Sort cars
        for (String i : cars) {
            System.out.println(i);
        }
    }
}
```



Sort an ArrayList

Sort an ArrayList of Integers:

```
import java.util.ArrayList;
import java.util.Collections; // Import the Collections class

public class Main {
    public static void main(String[] args) {
        ArrayList<Integer> myNumbers = new ArrayList<Integer>();
        myNumbers.add(33);
        myNumbers.add(15);
        myNumbers.add(20);
        myNumbers.add(34);
        myNumbers.add(8);
        myNumbers.add(12);

        Collections.sort(myNumbers); // Sort myNumbers

        for (int i : myNumbers) {
            System.out.println(i);
        }
    }
}
```

8

12

15

20

33

34

Java LinkedList

In the previous chapter, you learned about the [ArrayList](#) class. The [LinkedList](#) class is almost identical to the [ArrayList](#):

```
// Import the LinkedList class
import java.util.LinkedList;

public class Main {
    public static void main(String[] args) {
        LinkedList<String> cars = new LinkedList<String>();
        cars.add("Volvo");
        cars.add("BMW");
        cars.add("Ford");
        cars.add("Mazda");
        System.out.println(cars);
    }
}
```

[Volvo, BMW, Ford, Mazda]

ArrayList vs. LinkedList

The `LinkedList` class is a collection which can contain many objects of the same type, just like the `ArrayList`.

The `LinkedList` class has all of the same methods as the `ArrayList` class because they both implement the `List` interface. This means that you can add items, change items, remove items and clear the list in the same way.

However, while the `ArrayList` class and the `LinkedList` class can be used in the same way, they are built very differently.

How the ArrayList works

The `ArrayList` class has a regular array inside it. When an element is added, it is placed into the array. If the array is not big enough, a new, larger array is created to replace the old one and the old one is removed.

How the LinkedList works

The `LinkedList` stores its items in "containers." The list has a link to the first container and each container has a link to the next container in the list. To add an element to the list, the element is placed into a new container and that container is linked to one of the other containers in the list.

ArrayList vs. LinkedList

When To Use

Use an **ArrayList** for storing and accessing data, and **LinkedList** to manipulate data.

LinkedList Methods

For many cases, the **ArrayList** is more efficient as it is common to need access to random items in the list, but the **LinkedList** provides several methods to do certain operations more efficiently:

Method	Description
<code>addFirst()</code>	Adds an item to the beginning of the list.
<code>addLast()</code>	Add an item to the end of the list
<code>removeFirst()</code>	Remove an item from the beginning of the list.
<code>removeLast()</code>	Remove an item from the end of the list
<code>getFirst()</code>	Get the item at the beginning of the list
<code>getLast()</code>	Get the item at the end of the list