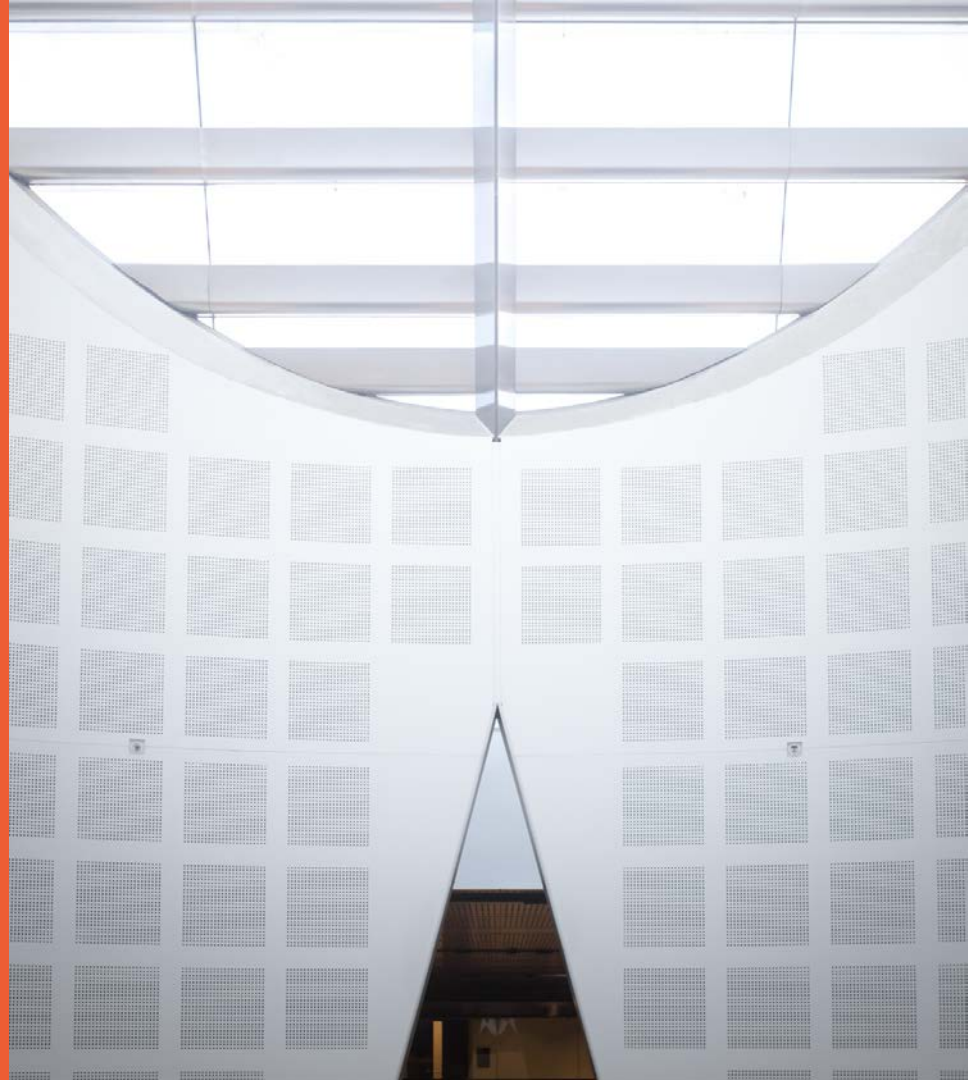
COMP9103: Software Development in Java

W3: Control Structures and Arrays

Presented by
Dr Ali Anaissi
School of Computer Science

THE UNIVERSITY OF SYDNEY



Control Structures

Control Structures

- Java provides the same set of control structures as in C and C#
- *Important*: the value used in the conditional expressions must be a **boolean**.

Control Structures

Java provides three control flow elements:

1. **Sequential**
   - Statements are executed in the order they are written
   - All the codes we have seen so far are sequential

2. **Branching (Decision making) -- Conditionals**
   - Provides computer programs with ability to make decisions and to carry out different actions according to different conditions

3. **Repetition (Iteration) – Loops**
   - When the given condition satisfied, execute a block of statements repeatedly

Conditionals

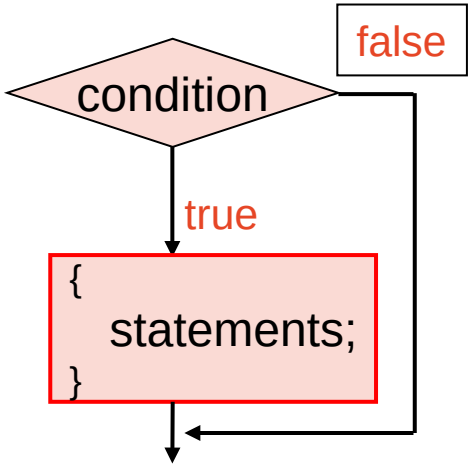
The *if* Structure

- The *if* structure allows to run statement(s) depending on a ***condition***
- The *if* structure has the syntax as:

condition is a boolean expression

```
if (condition){
  java statement(s);
}
```

if the condition is ***true,*** execute the statement(s)
if the condition is ***false***, the statement(s) will ***not*** be executed



- **if and only if** the *condition* is ***true,*** the {statement(s)} block is to be executed

Example: The *if* Statement

```
public class IFStatement {
/*    demonstrate how to use if structure
 */
      public static void main(String[] args) {
      //reads a number from the command-line input
      //print the output comments accordingly

        final int ZERO = 0;
        int input=Integer.parseInt(args[0]);
        System.out.print("the input " + input + " is a ");

        if (input < ZERO){
                System.out.print("negative ");
        }
                System.out.println("number");
}
```

```
java IFStatement -52
the input -52 is a negative number
```

```
java IFStatement 52
the input 52 is a number
```

The *if/else* Structure

– The syntax of ***if/else*** structure :

```
if (condition){
    statements_T
}
else{
    statements_F;
}
```

- If the *condition* is ***true***, execute {statement_T};
- Otherwise, (indicating the *condition* is ***false***) then execute {statement_F}

Example: Quadratic.java

- Print the real roots of $ax^2+bx+c=0$
- Condition for real roots: $b^2-4ac>=0$ *(non-negative)*
- Solution: ***if-else*** is used for testing the condition, and then print output accordingly

```
public class Quadratic { //calculate real roots for any user-defined (user inputs for a, b and c ) quadratic function
        public static void main(String[] args) {
                double a=Double.parseDouble(args[0]);
                double b=Double.parseDouble(args[1]);
                double c=Double.parseDouble(args[2]);
                double discriminant=b*b-4.0*a*c;
                if (discriminant < 0.0) //no real roots
                {
                        System.out.println("No real roots");
                }
                else //calculate the roots
                {
                        System.out.println((-b+Math.sqrt(discriminant))/2.0*a);
                        System.out.println((-b-Math.sqrt(discriminant))/2.0*a);
                }
        }
}
```

Note: Comparing Floating-Point Numbers

- Consider this code:

```
double r = Math.sqrt(2);
double d = r * r -2;
if (d == 0)
    System.out.println("sqrt(2)squared minus 2 is 0");
else
    System.out.println("sqrt(2)squared minus 2 is not 0 but " + d);
```

- It prints: *sqrt(2)squared minus 2 is not 0 but 4.440892098500626E-16*
- **Avoid using (Don't use!) "==" to compare floating-point numbers; instead, testing whether they are *close enough*:**

```
final double EPSILON = 1E-14;
if (Math.abs(x - y) <= EPSILON)
// x is approximately equal to y
//ε is a small number such as 1.0e-14
```

Multiple Alternatives: Sequence of Comparisons

```
if (condition_1) {
    statement_1;
}

else if (condition_2) {
    statement_2;
}

else if     . . .
…
else {
    statement_n;
}
```

Example: Multiple Alternatives

```
public class Earthquake
{//measure how severe an earthquake is and print the corresponding warning
    public static void main(String [] args)
    {
        double richter = Double.parseDouble(args[0]);
        String r;
        if (richter >= 8.0)          r = "Most structures fall";
        else if (richter >= 7.0)   r = "Many buildings destroyed";
        else if (richter >= 6.0)   r = "Many buildings considerably damaged, some collapse";
        else if (richter >= 4.5)   r = "Damage to poorly constructed buildings";
        else if (richter >= 3.5)   r = "Felt by many people, no destruction";
        else if (richter >= 0.0)   r = "Generally not felt by people";
        else                       r = "Negative numbers are not valid";
        System.out.println(r);

    }
}
```

The *switch* Statement: Another Way for Multibranch

– ***switch*** statement is to compare a single value against several constant alternatives.

```
switch (single_Control_Variable)
{
   case Constant_No0:
         statements ...
         break;
   case Constant_No1:
         statements ...
         break;
   case ......
   default:
         statements...
         break;
}
```

Type of *Control_Variable* can be:

- **Integer (byte, short, int, long); or**
- **character (char)**
- **String**

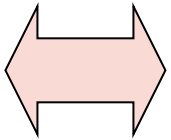
Not floating value, Not Boolean value

- When *break* statement is encountered, control leaves the *switch block*
- If a *break* statement is not used, the flow of control will continue to next case

Do not omit ***break***!

Example: The *switch* Statement

```
int digit=Integer.parseInt(args[0]);

if (digit == 1) System.out.print("one");

else if (digit == 2) System.out.print("two");

else if (digit == 3) System.out.print("three") ; else
if (digit == 4) System.out.print("four");

else if (digit == 5) System.out.print("five") ;

else if (digit == 6) System.out.print("six") ;

else if (digit == 7) System.out.print("seven") ;

else if (digit == 8) System.out.print("eight") ;

else if (digit == 9) System.out.print("nine") ;

else System.out.print("error") ;
```



```
int digit=Integer.parseInt(args[0]);

switch (digit)

{

        case 1: System.out.print("one"); break;

        case 2: System.out.print("two"); break;

        case 3: System.out.print("three"); break;

        case 4: System.out.print("four"); break;

        case 5: System.out.print("five"); break;

        case 6: System.out.print("six"); break;

        case 7: System.out.print("seven"); break;

        case 8: System.out.print("eight"); break;

        case 9: System.out.print("nine"); break;

        default: System.out.print("error"); break;

}
```

Loops

Repetition structure

- **Loop** is a programming structure that repeats an action for a certain number of times according to a Boolean condition
- **Body of loop** is the part to be repeated
- **Iteration** is every repetition of the loop body

- Three types of loops in Java:
  - ***while***
  - ***do-while***
  - ***for***

while Loop

– As long as the condition is true, a ***while*** structure executes a block of code (loop body) repeatedly.

```
while (condition)  {
        //loop body to be repeated
}
```

- The condition is evaluated first and if it is true, the loop body will be executed
- This procedure will be repeated until the condition becomes false

while Loop -- Tracing

- When you trace a loop, you keep track of the current line of code and the current values of the variables.
- Whenever a variable's value changes, you cross out the old value and write in the new value.

```
int i = 1;
int sum =0;
while (i <= 5) {
    sum=sum+i;
    i++;
}
```

sum	i	i <= 5
0	1	true
1	2	true
3	3	true
6	4	true
10	5	true
15	6	false

Common Error: Infinite Loops

- The body of a loop **must eventually make the loop condition false**, otherwise, the loop will keep running (infinite loop) and needs user's intervention to terminate it.

this is an infinite loop!

- Example 1:

```
int years = 0;
while (years < 10)  {
    double interest = balance * rate / 100;
    balance = balance + interest;
}
```

Change the control condition in the loop body!

years++;

- Example 2:

```
int years = 10;
while (years > 0)  {
    years++;
    double interest = balance * rate / 100;
    balance = balance + interest;
}
```

years--;

// Oops, the condition always be true
// the loop will not stop……

do-while Loop

– Executes loop body at least once:

```
do {
    loop body
} while (condition);
```

Do/Execute the loop body once initially and then evaluate the condition; while condition is true, the loop body will be repeated

Example: *do-while* Loop

do loop

```
do
{
   System.out.println(i);
   i=i-1;
} while (i > 0);
```

while loop

```
while (i> 0)
{
   System.out.println(i);
   i=i-1;
}
```

do-while loop executes <u>loop body at least once</u>

What is the output when initial value of i is: 5, 0, -5?

For loop

```
for (initialization; condition; update) {
    loop body
}
```

Step 0: The **initialization** is executed **only once** before the loop begins
Step 1: If the condition is true, the **loop body** is executed
Step 2: The **update** is executed at the end of each iteration
Step 1 & 2 will be repeated until the condition becomes false

Equivalent to

```
initialization;
while (condition)
{
    loop body;
    update;
}
```

For loop

```
int sum = 0;
for (int i = 1; i <= 10; i++) //sum=1+2+3+…+10

    sum = sum + i;
System.out.println(sum);
```

↔

```
int i = 1;
int sum =0;
while (i <= 10) {
    sum=sum+i;
    i++;
}
System.out.println(sum);
```

Variable Scope

- The **scope of a variable** is the section of the program where the variable is defined.
- Generally, the scope of a variable is comprised of the statements that follow the declaration in the same block as its declaration.
- **A variable is visible or accessible only within its scope**
- For instance, in a typical *for* loop, the incrementing variable is not available for use beyond the loop structure

Example: Variable Scope

```
for (int i=1; i<=4; i++ )
    System.out.println(i);

System.out.println(i);
```

Loop body & the scope of incrementing variable **i**

//out of scope---error!

A semicolon that shouldn't be there:

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

    sum = sum + i;

System.out.println(sum);
```

Loop body become empty statement and will do nothing

i is out of the scope and cannot be accessed

Nesting

- With nesting, you can compose loops and conditionals to build programs to solve complex problems.
  - Nest conditionals within conditionals
  - Nest loops within loops
  - Nest conditionals within loops

Nesting

- Create triangle pattern with nested loops:
- Loop through rows

```
for (int i = 1; i <= n; i++)
{
    // make each specific row
}
```

- Make a specific row via another loop

```
for (int j = 1; j <= i; j++)
      System.out.print("* ");
System.out.println();
```

Change to a new line

```
*
* *
* * *
* * * *
```

Put loops together → Nested loops

```
for (int i = 1; i <= n; i++)
 {
    for (int j = 1; j <= i; j++)
          System.out.print("* ");
    System.out.println();
 }
```

Loop (break/continue)

```
class BreakContinue {
    public static void main(String[] args) {
        for (int counter = 0;counter < 10;counter++) {
            // start a new iteration if the counter is odd
            if (counter % 2 == 1) continue;
            // abandon the loop if the counter is equal to 8
            if (counter == 8) break;
            // print the value
            System.out.print(counter +" ");
        }
        System.out.print("done.");
    }
}
```

```
java BreakContinue
0 2 4 6 done.
```

Arrays

Arrays

- **Array** is a collection of items of same type in a sequential (or list-like) structure
- The elements in an array are related
  - syntactically by being the same type (e.g., *int*, *double*, …)
  - semantically by being concerned with the same concept
    - Examples.
      - » 1 million characters in a book.
      - » 52 playing cards in a deck.
      - » 10 million audio samples in an MP3 file.
      - » 4 billion nucleotides in a DNA strand.
      - » 73 billion Google queries per year.

Make Arrays

- **Step 1:** ***Declare*** the array name and type
  - e.g. double [ ] **data;**
  - public static void main (**String[ ] args**)…
- **Step 2:** ***Create*** the array by using ***new*** followed by
  - array **type**, and
  - **[size/length/capacity]** of the array
  - e.g.
- Step 1 & 2 can be combined as: *double*[ ] **data** = ***new*** *double*[10];

type [ ] arrayName

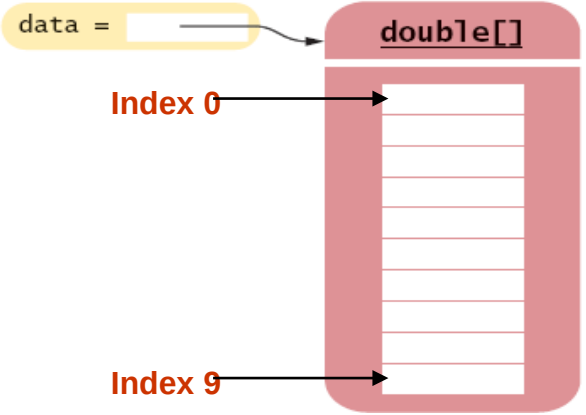
new type[size];

```
int N = 10;
double[] data;
data= new double[N];
```

```
double[] data= new double[N];
```

Arrays

- An array is a data structure that has a **single name** (eg *data*) and a **single type** (eg *double*)
- The member of an array is called **array element** and the elements are listed in order indicated by ***index***
- An **index** belongs to the range of *0* to array *length-1* ***
- A particular **array element is referenced and accessed** using the array name followed by the index in brackets (e.g., *data[i]*)



1st element: data[0]

2nd element: data[1]

……

10th element: data[9]

Arrays

- Get array length/size as ***arrayName.length*** (***Not a method & without ())
  - **Array length** = Array Capacity = maximum number of elements can be stored in the array

```
double[] data = new double[10];
//data.length is 10 (elements indexed from 0 to 9)
```

- Watch out array boundaries

```
double[ ] data = new double[10];
data[10] = 29.95; // ERROR – out of boundary (index range 0~9)!!!
```

- Limitation: primitive arrays have fixed length

Initialization of Arrays

- Default initialization: When an array is created, all elements are initialized depending on array type:
  - Numbers: 0
  - Boolean: *false*
- An array can be initialized by user-defined values:

```
data[0]=28.5;
data[1]=10.3;
```

- Initialization at declaration:

```
double [ ] temperature={28.5, 10.3, 22.2, -12.8};
```

Note:
the new operator is NOT used
No size is specified: the size of the array is determined by the number of the items in the initialization list
The initialization list can only be used at array declaration

Array Manipulations

Array Manipulations

- **Traversal:**
  - Aim: to access/manipulate **every element in the array**

The array has *data.length* elements but be aware that indices range from 0 to data.length-1

```
double[] data = new double[20];
for (int i=0; i<data.length; i++) {

   //do something with each data[i]
}
```

Note: the for loop goes from **i=0 to i<data.length**

Array Manipulations: Findmax

- Aim: to find the **index of the first "largest" element in the array**
  - Suppose data is an array of int type

0	4
1	11
2	13
3	17
4	35
5	15

maxIndex=0;
i=1

0	4
1	11
2	13
3	17
4	35
5	15

maxIndex=1;
i=2

0	4
1	11
2	13
3	17
4	35
5	15

maxIndex=2;
i=3

0	4
1	11
2	13
3	17
4	35
5	15

maxIndex=3;
i=4

0	4
1	11
2	13
3	17
4	35
5	15

maxIndex=4;
i=5

```
int maxIndex=0;            // a variable to remember the index of max element
for(int i=1; i<data.length; i=i+1)//go through every element in array except the 1st one
{
    if (data[i] > data[maxIndex])
    {
        maxIndex=i;
    }
} // maxIndex contains the index of the first largest element in the array
```

Self-testing: How to find minimal value in an array?

Array Manipulations: Reverse the elements in an array

– Reverse elements in a given array:

0	4
1	11
2	13
3	17
4	35
5	15

i=0

0	15
1	11
2	13
3	17
4	35
5	4

i=1

0	15
1	35
2	13
3	17
4	11
5	4

i=2

0	15
1	35
2	17
3	13
4	11
5	4

result

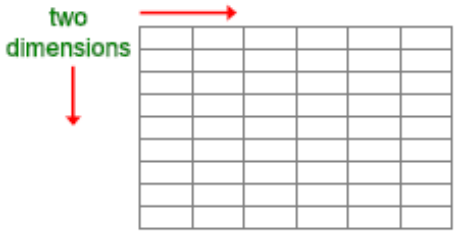
```
int N=a.length;
for (int i=0; i<N/2; i++)  //??? Can we change the stop condition to i<N here? Why?
{
        double temp=a[i];
        a[i]=a[N-1-i];
        a[N-1-i]=temp;
}
```

Self-testing: how to judge whether an array is symmetric?

Two-Dimensional Arrays

Two-Dimensional Arrays

- 2D Arrays:
  - table of students' grades
  - gray values of each picture element (pixel) in 2D image



- When constructing a two-dimensional array, we need to specify how many rows and columns in the array:

```
int rows = 3;
int columns = 4;
double[ ][ ] a = new double[rows][columns];
```

Declare the array variable

Create the array

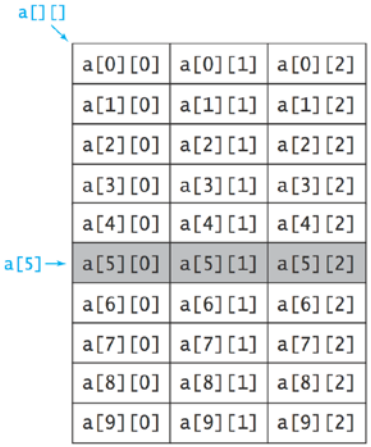
(allocate the memory space for the array)

Two-Dimensional Arrays

- **Array access:** use ***a[i][j]*** to access element at <u>row *i* and column *j*</u>.

- **Zero-based indexing.** Row and column indices start at **0**.
- To process all the elements in a two-dimensional array, you have to use two levels of loops: one loop nested inside the other

```
int m = 10;
int n = 3;
double[][] a = new double[m][n];
for (int i = 0; i < m; i++) {
    for (int j = 0; j < n; j++) {
        a[i][j] = 2.0;
    }
}
```

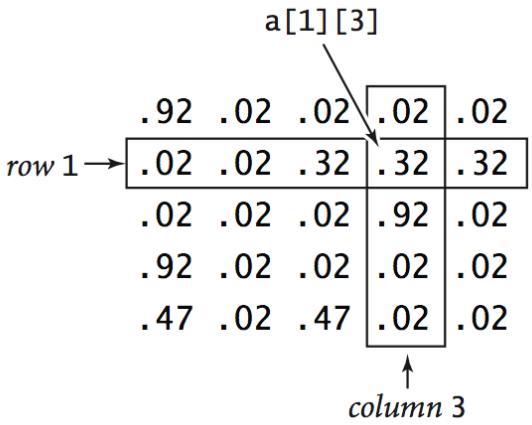


A 10-by-3 array

Setting 2D Array Values at Compile Time

– Initialize 2D array by <u>initialization list</u>: each row is initialized as a 1D array

```
double[][] a =
{
    { .02, .92, .02, .02, .02 },
    { .02, .02, .32, .32, .32 },
    { .02, .02, .02, .92, .02 },
    { .92, .02, .02, .02, .02 },
    { .47, .02, .47, .02, .02 },
};
```



Questions?