1495 UNIVERSITY OF ABERDEEN

JC2002 Java Programming

Day 2: Java fundamentals (CS)

Tuesday, 31 October

JC2002 Java Programming

Day 2, Session 1: Conditional structures and loops

References and learning objectives

- In the sessions today, we will continue with the basics of Java programming language
- Much of the material is based on slides from ***Java: How to Program***, chapters 2, 4, 5, 7, available via MyAberdeen
- After the theory sessions today, you should be able to:
  - Implement conditional structures and loops
  - Define and initialise arrays and ArrayLists and use them for simple tasks, like computing the sum of the elements in an array

Conditional statements

- In many situations, the program needs to make comparisons to decide what to do next
- In Java, conditional actions are usually implemented with *if … else* structure

Condition: Boolean expression (e.g. comparison) or boolean variable

```
if (condition) {
    do this
}
else {
    do that
}
```

If condition is true, do this

If condition is false, do that

Curly brackets

- Note that Java allows to omit the curly brackets when writing *if* statements containing only a single statement
- However, it is recommended always to use them to avoid bugs that are difficult to notice

✓
```
if (x>y) {
    System.out.println("x>y");
}
```

✓
```
if (x>y)
    System.out.println("x>y");
```

✗
```
if (x>y);
    System.out.println("x>y");
```

Comparisons example with *if* structure

```java
// Example of comparison with if
import java.util.Scanner; // needed for input
public class ComparisonIf {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        System.out.print("Enter x: "); int x = input.nextInt();
        System.out.print("Enter y: "); int y = input.nextInt();
        if( x == y ) {
            System.out.printf("%d == %d\n", x, y);
        }
        if( x < y ) {
            System.out.printf("%d < %d\n", x, y);
        }
        if( x > y ) {
            System.out.printf("%d > %d\n", x, y);
        }
    } // end section main
} // end class ComparisonIf
```

```
Enter x: 5
Enter y: 5
5 == 5
```

```
Enter x: 0
Enter y: 1
0 < 1
```

```
Enter x: 55
Enter y: 10
55 > 10
```

1495 UNIVERSITY OF ABERDEEN

Boolean operators

- You can combine conditions using Boolean operators ! (NOT), && (AND), and || (OR)

```
if (x > a && y > a) {
    System.out.println("x > a and y > a!");
}
if (x > a || y > a) {
    System.out.println("x > a or y > a!");
}
if (!(x > y)) {
    System.out.println("x <= y!");
}
```

Java *if* ... *else* statements

- Most Java programmers prefer to write the preceding nested *if…else* statement as:

```
if (studentGrade >= 90) {
   System.out.println("A");
}
else if (studentGrade >= 80) {
   System.out.println("B");
}
else if (studentGrade >= 70) {
   System.out.println("C");
}
else if (studentGrade >= 60) {
   System.out.println("D");
}
else {
   System.out.println("F");
}
```

Note: By convention, variable-name identifiers in Java use the *camel-case* naming convention with a lowercase first letter (e.g. **`firstNumber`**, **`studentGrade`**).

UNIVERSITY OF ABERDEEN

Java *switch* … *case* statements

- Sometimes it is reasonable to use *switch* … *case* structure instead of multiple comparisons:

```
switch (control) {
    case 0:
        do something
        break;
    case 1:
        do something else
        break;
    default:
        error handling etc.
        break;
}
```

Variable `control` will be tested for different conditions

If the variable has value 0, this will be done

If the variable has value 1, this will be done

This is done by default if none of the defined cases match

Exits the switch

Default action

Comparisons with *switch…case* structure

```java
// Example of conditional statements with case
import java.util.Scanner; // needed for input
public class TestCase {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        System.out.print("Choose 1 or 2: ");  int value = input.nextInt();
        switch(value) {
              case 1:
                  System.out.println("You chose 1!");
                  break;
              case 2:
                  System.out.println("You chose 2!");
                  break;
              default:
                  System.out.println("You did not choose 1 or 2!");
                  break;
           }
    } // end section main
} // end class TestCase
```

```
Choose 1 or 2: 1
You chose 1!
```

```
Choose 1 or 2: 2
You chose 2!
```

```
Choose 1 or 2: 3
You did not choose 1 or 2!
```

Precedence and associativity of operators

Operators	Associativity	Type
`*` `/` `%`	left to right	multiplicative
`+` `-`	left to right	additive
`<` `<=` `>`	left to right	relational
`==` `!=`	left to right	equality
`=`	right to left	assignment

Conditional operator (?:)

- *Conditional operator* (`?:`) is a shorthand of *if…else*
  - *Ternary operator* (takes *three* operands)
- Operands and `?:` form a *conditional expression*
  - Operand to the left of the `?` is a *boolean expression*—evaluates to a boolean value (`true` or `false`)
  - Second operand (between the `?` and `:`) is the value if the boolean expression is `true`
  - Third operand (to the right of the `:`) is the value if the boolean expression evaluates to `false`

```
System.out.println(studentGrade >= 60 ? "Passed" : "Failed");
```

<-- Boolean expression -->   <-- if true -->   <-- if false -->

Example of conditional operator

```java
// Example of conditional operator
import java.util.Scanner; // needed for input
public class ClassCond {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        System.out.print("Write your age: ");
        int age = input.nextInt();
        String str = age < 18 ? "minor" : "adult";
        System.out.printf("You are %s!\n", str);
    } // end section main
} // end class ClassCond
```

```
Write your age: 10
You are minor!
```

```
Write your age: 50
You are adult!
```

- Conditional operator allows you to write compact code, but beware: it can make code difficult to read and prone to bugs!

Java iteration statement *while*

- In some situations, the program needs to repeat an action many times if the condition remains true
- In Java, *while* iteration statement can be used for this

Condition: Boolean expression (e.g. comparison) or boolean variable

```
while (condition) {
    do this
}
continue here
```

If condition is true, do this, then test condition again, if it is still true, repeat doing this

If condition is false, just continue here

Example of using *while* iteration statement

- Find the first power of 3 larger than 100:

```
product = 3;
while (product <= 100)
    product = 3 * product;
```

- Each iteration multiplies `product` by 3, so `product` takes on the values 9, 27, 81 and 243 successively
- When `product` becomes 243, `product <= 100` becomes false
- Iteration terminates, the final value of `product` is 243
- Program execution continues with the next statement after the `while` statement

Example of *while* loop

```java
// Example of while loop
import java.util.Scanner; // needed for input
public class ClassAverage {
    public static void main(String[] args) {
        Scanner input = new Scanner(System.in);
        int total = 0; // initialize sum of grades
        int gradeCounter = 0; // initialize number of grades

        while (gradeCounter <= 10) { // loop ten times
            System.out.print ("Enter grade: ");
            int grade = input.nextInt();
            total = total + grade;
            gradeCounter = gradeCounter + 1;
        }
        System.out.printf("Average: %d%n", total/gradeCounter);
    } // end section main
} // end class ClassAverage
```

Java *do … while* iteration statement

- The iteration statement do…`while` is similar to `while` statement
- In the `while` statement, loop-continuation condition tested at the *beginning* of the loop, ***before*** executing the loop's body; if the condition is *false*, the body *never* executes
- The do…`while` statement tests the loop-continuation condition ***after*** executing the loop's body; therefore, *the body always executes at least once*
- When a do…`while` loop terminates, execution continues with the next statement in sequence

Example of *do…while* loop

```java
// Example of do..while loop
public class DoWhileTest {
    public static void main(String[] args) {
        int counter = 1; // initialize counter

        do {
            System.out.printf("%d ", counter);
            ++counter;
        } while (counter <= 10);

        System.out.println();
    } // end section main
} // end class DoWhileTest
```

```
1 2 3 4 5 6 7 8 9 10
```

Java *for* loop

- *For* loops are common in many programming languages
- In Java, *for* loops have the following header components:

Initialization expression

Test expression

Update expression

```
for( int counter = 1; counter <= 10; counter++ )
```

Initializes control variable with value 1 when the loop is started

Loop is repeated if the test expression is true

Update expression updates the control variable in each iteration

Example of *for* loop

```java
// Example of for loop
public class ForTest {
    public static void main(String[] args) {

        for (int counter = 1; counter <= 10; counter++) {
            System.out.printf("%d ", counter);
        }

        System.out.println();
    } // end section main
} // end class ForTest
```

```
1 2 3 4 5 6 7 8 9 10
```

1495 UNIVERSITY OF ABERDEEN

Questions, comments?

JC2002 Java Programming

Day 2, Session 4: Arrays and array lists

What are arrays?

- An array is a group of variables (called elements or components), containing values that all have the same type
- Arrays are objects (*reference types*) , whereas the elements of an array can be either *primitive types* or *reference types* (including arrays)
    - Remember: the primitive types are `boolean, byte, char, short, int, long, float` and `double`; all the other types are *reference types*
- Arrays *remain the same length* once they're created

Declaring and creating arrays

- Array objects
  - You specify the element type and the number of elements in an *array-creation expression*, which returns a reference that can be stored in an array variable
- Declaration and array-creation expression for an array of 12 `int` elements:

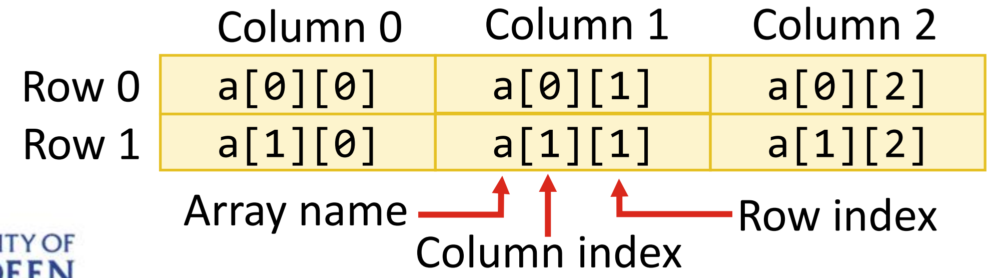
```
int[] c = new int[12];
```

- It can also be performed in two steps as follows:

```
int[] c; // declare the array variable
c = new int[12]; // create the array
```

Multidimensional arrays

- Java does *not* support multidimensional arrays directly
  - To achieve the same effect, we can specify one-dimensional arrays whose elements are also one-dimensional arrays, etc.
- Two-dimensional arrays are often used to represent tables of values with data arranged in rows and columns
  - An array with *m* rows and *n* columns is called an *m-by-n array*
- Each table element is identified with two indices
  - By convention, the first index is the row and the second is the column



	Column 0	Column 1	Column 2
Row 0	a[0][0]	a[0][1]	a[0][2]
Row 1	a[1][0]	a[1][1]	a[1][2]

Multidimensional arrays can have ***more than*** two dimensions!

Array example 1

- One-dimensional array, initialise the elements of an array to default values of zero

```
public class InitArray1 {
      public static void main(String[] args) {
            // declare variable array and initialize it with an array object
            int[] array = new int[10]; // create the array object

            System.out.printf("%s%8s%n", "Index", "Value"); // column headings

            // output each array element's value
            for (int counter = 0; counter < array.length; counter++) {
                  System.out.printf("%5d%8d%n", counter, array[counter]);
            }
      }
}
```

Array example 2

- One-dimensional array, initialise the elements of an array with an array initialiser

```java
public class InitArray2 {
      public static void main(String[] args) {
            // initializer list specifies the initial value for each element
            int[] array = {32, 27, 64, 18, 95, 14, 90, 70, 60, 37};

            System.out.printf("%s%8s\n", "Index", "Value"); // column headings

            // output each array element's value
            for (int counter = 0; counter < array.length; counter++) {
                  System.out.printf("%5d%8d%n", counter, array[counter]);
            }
      }
}
```

Array example 3

- One-dimensional array, compute the sum of the elements of an array

```java
public class SumArray{
      public static void main(String[] args) {
            int[] array = {87, 68, 94, 100, 83, 78, 85, 91, 76, 87};
            int total = 0;

            // add each element's value to total
            for (int counter = 0; counter < array.length; counter++) {
                  total += array[counter];
            }

            System.out.printf("Total of array elements: %d%n", total);
      }
}
```

Array example 4 (1)

- One-dimensional array, passing arrays and individual array elements to methods

```java
public class PassArray {
      // main creates array and calls modifyArray and modifyElement
      public static void main(String[] args) {
            int[] array = {1, 2, 3, 4, 5};

            System.out.printf(
                  "Effects of passing reference to entire array:%n" +
                  "The values of the original array are:%n");

            // output original array elements
            for (int value : array) {
                  System.out.printf(" %d", value);
            }
```

Array example 4 (2)

```
            modifyArray(array); // pass array reference
            System.out.printf("%n%nThe values of the modified array are:%n");

            // output modified array elements
            for (int value : array) {
                  System.out.printf(" %d", value);
            }

            System.out.printf(
                  "%n%nEffects of passing array element value:%n" +
                  "array[3] before modifyElement: %d%n", array[3]);

            modifyElement(array[3]); // attempt to modify array[3]
            System.out.printf(
                  "array[3] after modifyElement: %d%n", array[3]);
            }
```

Array example 4 (3)

```java
        // multiply each element of an array by 2
        public static void modifyArray(int[] array2) {
              for (int counter = 0; counter < array2.length; counter++) {
                    array2[counter] *= 2;
              }
        }

        // multiply argument by 2
        public static modifyElement(int element) {
              element *= 2;
              System.out.printf(
                    "Value of element in modifyElement: %d%n", element);
        }
  }
```

Array example 4 (4)

- Program output

```
Effects of passing reference to entire array:
The values of the original array are:
 1 2 3 4 5

The values of the modified array are:
 2 4 6 8 10

Effects of passing array element value:
array[3] before modifyElement: 8
Value of element in modifyElement: 16
array[3] after modifyElement: 8
```

Array example 5 (1)

- Initialising two-dimensional arrays

```java
public class Init2DArray {
     // create and output two-dimensional arrays
     public static void main(String[] args) {
          int[][] array1 = {{1, 2, 3}, {4, 5, 6}};
          int[][] array2 = {{1, 2}, {3}, {4, 5, 6}};

          System.out.println("Values in array1 by row are");
          outputArray(array1); // displays array1 by row

          System.out.printf("%nValues in array2 by row are%n");
          outputArray(array2); // displays array2 by row
     }
```

Array example 5 (2)

```java
        // output rows and columns of a two-dimensional array
        public static void outputArray(int[][] array) {
            // loop through array's rows
            for (int row = 0; row < array.length; row++) {
                // loop through columns of current row
                for (int column = 0; column < array[row].length; column++) {
                    System.out.printf("%d ", array[row][column]);
                }
                System.out.println();
            }
        }
}
```

Array example 5 (3)

- Program output

```
Values in array1 by row are
1 2 3
4 5 6

Values in array2 by row are
1 2
3
4 5 6
```

Class ArrayList

- Arrays do not change their size at execution time to accommodate additional elements
  - If you need to increase array size during execution, you need to declare and initialize it again
- For arrays with dynamic size, you can use class `ArrayList<T>` in package `java.util`
  - Note that T is a placeholder for the type of element stored in the `ArrayList` object. Classes with this kind of placeholder that can be used with any type are called *generic classes*; we will discuss more about generic classes later in this course

ArrayList methods

Method	Description
add	Adds an element to the end or at the specific index of the ArrayList.
clear	Removes all the elements from the ArrayList.
contains	Returns true if the ArrayList contains the specified element, otherwise returns false.
get	Returns the element at the specified index.
indexOf	Returns the index of the first occurrence of the specified element in the ArrayList.
remove	Removes the first occurrence of the specified value or the element at the specified index.
size	Returns the number of elements stored in the ArrayList.
trimToSize	Trims the capacity of the ArrayList to its current size.

ArrayList example (1)

- Generic `ArrayList<E>` example

```java
import java.util.ArrayList;

public class ArrayListCollection {
      public static void main(String[] args) {
            // create a new ArrayList of Strings with an initial capacity of 10
            ArrayList<String> items = new ArrayList<String>();

            items.add("red"); // append an item to the list
            items.add(0, "yellow"); // insert "yellow" at index 0

            // header
            System.out.print(
                  "Display list contents with counter-controlled loop:");
```

ArrayList example (2)

```
// display the colors in the list
for (int i = 0; i < items.size(); i++) {
      System.out.printf(" %s", items.get(i));
}

// display colors using enhanced for in the display method
display(items,
      "\nDisplay list contents with enhanced for statement:");

items.add("green"); // add "green" to the end of the list
items.add("yellow"); // add "yellow" to the end of the list
display(items, "List with two new elements:");

items.remove("yellow"); // remove the first "yellow"
display(items, "Remove first instance of yellow:");

items.remove(1); // remove item at index 1
display(items, "Remove second list element (green):");
```

1495 UNIVERSITY OF ABERDEEN

ArrayList example (3)

```
            // check if a value is in the List
            System.out.printf("\"red\" is %sin the list\n",
            items.contains("red") ? "": "not ");

            // display number of elements in the List
            System.out.printf("Size: %s%n", items.size());
        }

        // display the ArrayList's elements on the console
        public static void display(ArrayList<String> items, String header) {
            System.out.printf(header); // display header

            // display each element in items
            for (String item : items) {
                System.out.printf(" %s", item);
            }
            System.out.println();
        }
}
```

ArrayList example (4)

- Program output

```
Display list contents with counter-controlled loop: yellow red
Display list contents with enhanced for statement: yellow red
List with two new elements: yellow red green yellow
Remove first instance of yellow: red green yellow
Remove second list element (green): red yeallow
"red" is in the list
Size: 2
```

Summary

- Basic structures of Java programs introduced
  - Implementing conditional statements: `if...else`, `switch...case`, ternary operator `? :`
  - Implementing loops: `while`, `do...while`, `for`
  - Defining and using arrays and `ArrayLists`

Questions, comments?

1495 UNIVERSITY OF ABERDEEN