Control Structures, Loop, Array, Methods

s

Java Conditions and If Statements

- Java supports the usual logical conditions from mathematics:
  - Less than: a < b
  - Less than or equal to: a <= b
  - Greater than: a > b
  - Greater than or equal to: a >= b
  - Equal to a == b
  - Not Equal to: a != b
- These conditions to perform different actions for different decisions.
- Java has the following conditional statements:
  - Use if to specify a block of code to be executed, if a specified condition is true
  - Use else to specify a block of code to be executed, if the same condition is false
  - Use else if to specify a new condition to test, if the first condition is false
  - Use switch to specify many alternative blocks of code to be executed

The if Statement

- Use the if statement to specify a block of Java code to be executed if a condition is true.
- Syntax:

```
if (condition) {
// block of code to be executed if the condition is true
}
```

- Example:

```
public class Main {
  public static void main(String[] args) {
    int x = 20;
    int y = 18;
    if (x > y) {
      System.out.println("x is greater than y");
    }
  }
}
```

The else Statement

- Use the else statement to specify a block of code to be executed if the condition is false.
- Syntax:

```
if (condition) {
// block of code to be executed if the condition is true
} else {
// block of code to be executed if the condition is false
}
```

- Example:

```
public class Main {
  public static void main(String[] args) {
    int time = 20;
    if (time < 18) {
      System.out.println("Good day.");
    } else {
      System.out.println("Good evening.");
    }
  }
}
```

The else if Statement

- Use the else if statement to specify a new condition if the first condition is false.
- Syntax:

```
if (condition1) {
// block of code to be executed if condition1 is true
}
else if (condition2) {
// block of code to be executed if the condition1 is false and condition2 is true
}
else {
// block of code to be executed if the condition1 is false and condition2 is false
}
```

Example:

```
public class Main {
  public static void main(String[] args) {
    int time = 22;
    if (time < 10) {
      System.out.println("Good morning.");
    } else if (time < 18) {
      System.out.println("Good day.");
    } else {
      System.out.println("Good evening.");
    }
  }
}
```

Short Hand If...Else

- There is also a short-hand if else, which is known as the **ternary operator** because it consists of three operands.
- It can be used to replace multiple lines of code with a single line, and is most often used to replace simple if else statements:
- Syntax:
  - *variable = (condition) ? expressionTrue : expressionFalse;*
- **Example**:

```
public class Main {
  public static void main(String[] args) {
    int time = 20;
    String result;
    result = (time < 18) ? "Good day." : "Good evening.";
    System.out.println(result);
  }
}
```

Java Switch Statements

- Instead of writing **many** if..else statements, you can use the switch statement.
- The switch statement selects one of many code blocks to be executed:
- Syntax:

```
switch(expression) {
case x: // code block
break;
case y: // code block
break;
default: // code block
}
```

- How it works:
  - The switch expression is evaluated once.
  - The value of the expression is compared with the values of each case.
  - If there is a match, the associated block of code is executed.
  - The break and default keywords are optional

Java Switch Statements

- *Example:*

```
public class Main {
  public static void main(String[] args) {
    int day = 4;
    switch (day) {
      case 1:
        System.out.println("Monday");
        break;
      case 2:
        System.out.println("Tuesday");
        break;
      case 3:
        System.out.println("Wednesday");
        break;
      case 4:
        System.out.println("Thursday");
        break;
      case 5:
        System.out.println("Friday");
        break;
      case 6:
        System.out.println("Saturday");
        break;
      case 7:
        System.out.println("Sunday");
        break;
    }
  }
}
```

Loops

- Loops can execute a block of code as long as a specified condition is reached.
- Loops are handy because they save time, reduce errors, and they make code more readable.
  - While
  - Do..While
  - For

While loop

- The while loop loops through a block of code as long as a specified condition is true:
- Syntax:

```
while (condition) {
// code block to be executed
}
```

Example:

```
public class Main {
  public static void main(String[] args) {
    int i = 0;
    while (i < 5) {
      System.out.println(i);
      i++;
    }
  }
}
```

The Do..While Loop

- The do…while loop is a variant of the while loop.
- This loop will execute the code block once, before checking if the condition is true, then it will repeat the loop as long as the condition is true.
- Syntax:

```
do {
// code block to be executed
} while (condition);
```

- Example:

```
public class Main {
  public static void main(String[] args) {
    int i = 0;
    do {
      System.out.println(i);
      i++;
    }
    while (i < 5);
  }
}
```

For Loop

- When you know exactly how many times you want to loop through a block of code, use the for loop instead of a while loop.
- Syntax:

```
for (statement 1; statement 2; statement 3) {
// code block to be executed
}
```

  - **Statement 1** is executed (one time) before the execution of the code block.
  - **Statement 2** defines the condition for executing the code block.
  - **Statement 3** is executed (every time) after the code block has been executed.

For Loop

- **Example1**:

```
public class Main {
  public static void main(String[] args) {
    for (int i = 0; i < 5; i++) {
      System.out.println(i);
    }
  }
}
```

- *Example2:*

```
public class Main {
  public static void main(String[] args) {
    for (int i = 0; i <= 10; i = i + 2) {
      System.out.println(i);
    }
  }
}
```

Nested For Loop

- It is also possible to place a loop inside another loop. This is called a **nested loop**.
- The "inner loop" will be executed one time for each iteration of the "outer loop“
- Example:

```
public class Main {
  public static void main(String[] args) {
    // Outer loop.
    for (int i = 1; i <= 2; i++) {
      System.out.println("Outer: " + i); // Executes 2 times

      // Inner loop
      for (int j = 1; j <= 3; j++) {
        System.out.println(" Inner: " + j); // Executes 6 times (2 * 3)
      }
    }
  }
}
```

For each Loop

- There is also a "**for-each**" loop, which is used exclusively to loop through elements in an **array**:
- Syntax

```
for (type variableName : arrayName) {
// code block to be executed
}
```

- Example:

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

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**:
- Example:
  - ***String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};***
  - ***int[] myNum = {10, 20, 30, 40};***

Arrays

- **Access the Elements of an Array**
  - You can access an array element by referring to the index number.
- **Example:**

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

Arrays

- **Change an Array Element**
  - To change the value of a specific element, refer to the index number:
- Example:

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

Arrays

- **Array Length**
  - To find out how many elements an array has, use the length property:
- Example:

```
public class Main {
  public static void main(String[] args) {
    String[] cars = {"Volvo", "BMW", "Ford", "Mazda"};
    System.out.println(cars.length);
  }
}
```

Arrays

- **Loop Through an Array**
- We can loop through the array elements with the for loop, and use the length property to specify how many times the loop should run.
- Example

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

Arrays

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

  for (*type variable* : *arrayname*) { … }
- Example:

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

Multidimensional Arrays

- A multidimensional array is an array of arrays.
- Multidimensional arrays are useful when you want to store data as a tabular form, like a table with rows and columns.
- To create a two-dimensional array, add each array within its own set of **curly braces**:
- Example:
  - ***int[][] myNumbers = { {1, 2, 3, 4}, {5, 6, 7} };***
- **myNumbers** is now an array with two arrays as its elements.

Multidimensional Arrays

- **Access 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:
- Example:

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

Multidimensional Arrays

- **Change Element Values**
- Example:

```
public class Main {
  public static void main(String[] args) {
    int[][] myNumbers = { {1, 2, 3, 4}, {5, 6, 7} };
    myNumbers[1][2] = 9;
    System.out.println(myNumbers[1][2]); / / Outputs 9 instead of 7
  }
}
```

Multidimensional Arrays

- **Loop Through a Multi-Dimensional Array**
- 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):
- Example:

```
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]);
      }
    }
  }
}
```

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:

Methods

- **Creation and Method call**

```
1.  public class Main {
2.    static void myMethod() {
3.      System.out.println("I just got executed!");
4.    }
5.    public static void main(String[] args) {
6.      myMethod();
7.    }
8.  }
```

- 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.
- void means that this method does not have a return value.
- To call a method in Java, write the method's name followed by two parentheses **()** and a semicolon**;**
- In the above example, myMethod() is used to print a text (the action), when it is called.

Methods

- **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.
- We can add as many parameters as you want, just separate them with a comma.

Methods

- **Parameters and Arguments**

```
1.  public class Main {
2.    static void myMethod(String fname) {
3.      System.out.println("Hello " + fname);
4.    }
5.    public static void main(String[] args) {
6.      myMethod("World!");
7.      myMethod("Ram");
8.      myMethod("Sita");
9.    }
10. }
```

- The above 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:

Methods

- Multiple Parameters

```
1.  public class Main {
2.    static void myMethod(String fname, int age) {
3.      System.out.println(fname + " is " + age +" years old");
4.    }
5.    public static void main(String[] args) {
6.      myMethod("Ram", 5);
7.      myMethod("Aram", 8);
8.      myMethod("Sitaram", 31);
9.    }
10. }
```

Methods

- **Return Values**
- The void keyword, used in the examples above, indicates that the method should not return a value.
- If we 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:

```
1.  public class Main {
2.    static int myMethod(int x) {
3.      return 5 + x;
4.    }
5.    public static void main(String[] args) {
6.      System.out.println(myMethod(3));
7.    }
8.  }
```

Methods

- **Return Values: Examples**

```
1.  public class Main {
2.    static int myMethod(int x, int y) {
3.      return x + y;
4.    }
5.    public static void main(String[] args) {
6.      System.out.println(myMethod(5, 3));
7.    }
8.  }
```

```
1.  public class Main {
2.    static int myMethod(int x, int y) {
3.      return x + y;
4.    }
5.    public static void main(String[] args) {
6.      int z = myMethod(5, 3);
7.      System.out.println(z);
8.    }
9.  }
```

Methods

- A Method with control statements

```
1.  public class Main {
2.    static void checkAge(int age) {
3.      if (age < 18) {
4.        System.out.println("You can not cast vote");
5.      } else {
6.        System.out.println("You can cast vote");
7.      }
8.    }
9.    public static void main(String[] args) {
10.     checkAge(20);
11.   }
12. }
```

Methods

- Method Overloading
- With **method overloading**, multiple methods can have the same name with different parameters:
- Here we have two methods to do the same arithmetic addition:

```
1.  public class Main {
2.    static int plusMethodInt(int x, int y) {
3.      return x + y;
4.    }
5.    static double plusMethodDouble(double x, double y) {
6.      return x + y;
7.    }
8.    public static void main(String[] args) {
9.      int myNum1 = plusMethodInt(8, 5);
10.     double myNum2 = plusMethodDouble(4.3, 6.26);
11.     System.out.println("int: " + myNum1);
12.     System.out.println("double: " + myNum2);
13.   }
14. }
```

Methods

- Method Overloading
- With **method overloading**, multiple methods can have the same name with different parameters:
- Instead of defining two methods that should do the same thing, it is better to overload one.
- Example:

```
1.  public class Main {
2.    static int plusMethod(int x, int y) {
3.      return x + y;
4.    }
5.    static double plusMethod(double x, double y) {
6.      return x + y;
7.    }
8.    public static void main(String[] args) {
9.      int myNum1 = plusMethod(8, 5);
10.     double myNum2 = plusMethod(4.3, 6.26);
11.     System.out.println("int: " + myNum1);
12.     System.out.println("double: " + myNum2);
13.   }
14. }
```

Methods

- 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:

```
1.  public class Main {
2.    public static void main(String[] args) {
3.      // Code here cannot use x
4.      int x = 100;
5.      // Code here can use x
6.      System.out.println(x);
7.    }
8.  }
```

Methods

- **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:

```
1.  public class Main {
2.    public static void main(String[] args) {
3.      // Code here CANNOT use x
4.      { // This is a block
5.        // Code here CANNOT use x
6.        int x = 100;
7.        // Code here CAN use x
8.        System.out.println(x);
9.      } // The block ends here
10.   // Code here CANNOT use x
11.   }
12. }
```