Lecture 12
# Operator Precedence and Associativity

## Table 3.10: Operator Precedence Chart

<table>
<thead>
<tr>
<th>Precedence</th>
<th>Operator</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><code>var++</code> and <code>var--</code> (Postfix)</td>
</tr>
<tr>
<td></td>
<td><code>+</code>, <code>-</code> (Unary plus and minus), <code>++var</code> and <code>--var</code> (Prefix)</td>
</tr>
<tr>
<td></td>
<td>(type) (Casting)</td>
</tr>
<tr>
<td></td>
<td>!(Not)</td>
</tr>
<tr>
<td></td>
<td><code>*</code>, <code>/</code>, <code>%</code> (Multiplication, division, and remainder)</td>
</tr>
<tr>
<td></td>
<td><code>+</code>, <code>-</code> (Binary addition and subtraction)</td>
</tr>
<tr>
<td></td>
<td><code>&lt;</code>, <code>&lt;=</code>, <code>&gt;</code>, <code>&gt;=</code> (Comparison)</td>
</tr>
<tr>
<td></td>
<td><code>==</code>, <code>!=</code> (Equality)</td>
</tr>
<tr>
<td></td>
<td><code>^</code> (Exclusive OR)</td>
</tr>
<tr>
<td></td>
<td><code>&amp;&amp;</code> (AND)</td>
</tr>
<tr>
<td></td>
<td>`</td>
</tr>
<tr>
<td></td>
<td><code>=</code>, <code>+=</code>, <code>-=</code> (Assignment operator)</td>
</tr>
</tbody>
</table>
Evaluate the following expressions

• true || true && false
• true && true || false
• 2 * 2 - 3 > 2 && 4 - 2 > 5
• 2 * 2 - 3 > 2 || 4 - 2 > 5
Conditional Expressions

- syntax

```java
boolean-expression ? expression1 : expression2;

```java
if (x > 0) y = (x > 0) ? 1 : -1;
y = 1;
else y = -1;

System.out.println((num % 2 == 0) ? "num is even" : "num is odd");

```java
public class Test {
    public static void main(String[] args) {
        java.util.Scanner input = new java.util.Scanner(System.in);
        double x = input.nextDouble();
        double y = input.nextDouble();
        double z = input.nextDouble();
        System.out.println((x < y && y < z) ? "sorted" : "not sorted");
    }
}
```
Formatting Console Output

double amount = 12618.98;
double interestRate = 0.0013;

double interest = amount * interestRate;
System.out.println("Interest is " + interest);

Interest is 16.404674

Because the interest amount is currency, it is desirable to display only two digits after the decimal point. To do this, you can write the code as follows:

double amount = 12618.98;
double interestRate = 0.0013;
double interest = amount * interestRate;
System.out.println("Interest is "+ (int)(interest * 100) / 100.0);

Interest is 16.4

However, the format is still not correct. There should be two digits after the decimal point: **16.40** rather than **16.4**. You can fix it by using the **printf** method, like this:
double amount = 12618.98;
double interestRate = 0.0013;
double interest = amount * interestRate;
System.out.printf("Interest is %4.2f", interest);

Interest is 16.40

The syntax to invoke this method is

System.out.printf(format, item1, item2, ..., itemk)

where format is a string that may consist of substrings and format specifiers.

A format specifier specifies how an item should be displayed. An item may be a numeric value, a character, a Boolean value, or a string. A simple format specifier consists of a percent sign (%) followed by a conversion code. Table 3.8 lists some frequently used simple format specifiers.

<table>
<thead>
<tr>
<th>FormatSpecifier</th>
<th>Output</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>%b</td>
<td>a Boolean value</td>
<td>true or false</td>
</tr>
<tr>
<td>%c</td>
<td>a character</td>
<td>‘a’</td>
</tr>
<tr>
<td>%d</td>
<td>a decimal integer</td>
<td>200</td>
</tr>
<tr>
<td>%f</td>
<td>a floating-point number</td>
<td>45.460000</td>
</tr>
<tr>
<td>%e</td>
<td>a number in standard scientific notation</td>
<td>4.556000e+01</td>
</tr>
<tr>
<td>%s</td>
<td>a string</td>
<td>“Java is cool”</td>
</tr>
</tbody>
</table>
```java
int count = 5;
double amount = 45.56;
System.out.printf("count is \%d and amount is \%f", count, amount);
```

**Table 3.9** Examples of Specifying Width and Precision

<table>
<thead>
<tr>
<th>Example</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>%5c</td>
<td>Output the character and add four spaces before the character item, because the width is 5.</td>
</tr>
<tr>
<td>%6b</td>
<td>Output the Boolean value and add one space before the false value and two spaces before the true value.</td>
</tr>
<tr>
<td>%5d</td>
<td>Output the integer item with width at least 5. If the number of digits in the item is &lt;5, add spaces before the number. If the number of digits in the item is &gt;5, the width is automatically increased.</td>
</tr>
<tr>
<td>%10.2f</td>
<td>Output the floating-point item with width at least 10 including a decimal point and two digits after the point. Thus there are 7 digits allocated before the decimal point. If the number of digits before the decimal point in the item is &lt;7, add spaces before the number. If the number of digits before the decimal point in the item is &gt;7, the width is automatically increased.</td>
</tr>
<tr>
<td>%10.2e</td>
<td>Output the floating-point item with width at least 10 including a decimal point, two digits after the point and the exponent part. If the displayed number in scientific notation has width less than 10, add spaces before the number.</td>
</tr>
<tr>
<td>%12s</td>
<td>Output the string with width at least 12 characters. If the string item has fewer than 12 characters, add spaces before the string. If the string item has more than 12 characters, the width is automatically increased.</td>
</tr>
</tbody>
</table>
System.out.printf("%3d%2s%3.2f\n", 1234, "Java", 51.6653);

displays

1234#Java#51.67

By default, the output is right justified. You can put the minus sign (–) in the format specifier to specify that the item is left justified in the output within the specified field. For example, the following statements

System.out.printf("%8d%8s%8.1f\n", 1234, "Java", 5.63);
System.out.printf("%-8d%-8s%-8.1f \n", 1234, "Java", 5.63);

display

<table>
<thead>
<tr>
<th>8</th>
<th>8</th>
<th>8</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>□□□□ 1234 □□□□ Java □□□□ 5.6</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>□□□□ 1234 □□□□ Java □□□□ 5.6</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

where the square box (□) denotes a blank space.
Programming Errors

• Syntax Errors
  – Errors that are detected by the compiler are called syntax errors or compile errors. Syntax errors result from errors in code construction, such as mistyping a keyword, omitting some necessary punctuation, or using an opening brace without a corresponding closing brace. These errors are usually easy to detect, because the compiler tells you where they are and what caused them.
• Runtime Errors
  
  – Runtime errors are errors that cause a program to terminate abnormally. They occur while a program is running if the environment detects an operation that is impossible to carry out. Input mistakes typically cause runtime errors.
• Logic Errors
  – Logic errors are called bugs. The process of finding and correcting errors is called debugging.
The while-statement

- Syntax of the while-statement:

```
while ( loop-continuation-condition )
  ONE-statement
```

Body of the while-loop

Explanation:

- The keyword while announces (to the Java compiler) that we started an while-statement
- A conditional clause ( LOOP-CONTINUATION-CONDITION ) follows the keyword while
  - The LOOP-CONTINUATION-CONDITION is a Boolean expression (exactly the same as in the condition clause of an if-statement)
  - This is the condition of the while-statement
- Following the loop-continuation-condition clause, you can write (only) one statement
  - This is the body of the while-statement
  - The body will be executed as long as the loop-continuation-condition is true !!!
○ Fact:

- The body of any loop-statement will almost always contain multiple statements

○ Common practice in while-loops:

- Use a block as body of loop-statements

○ A typical while-loop looks like this:

```plaintext
while ( loop-continuation-condition )
{
    statement1
    statement2
    ....
}
```

Body of the while-loop
Flow chart representing a while-statement:

- **loop-cont-condition**
  - false
  - true
  - Body of while-loop

Execution repeats after body is done

- statement1
- statement2
- ...
```java
public class While01 {
    
    public static void main(String[] args) {
        int a;

        a = 1;

        while (a <= 10) {  // While-statement
            System.out.println(a);  // Print a
            a++;  // Increment a
        }

        System.out.println("Done");
        System.out.println("Exit: a = " + a);
    }
}
```
The loop invariant

- Property of loops
  - An important property of loops
    - In each iteration of a loop statement, the entire loop body is executed

Implication:
- Because the loop body is executed many times, to ensure correctness, the same condition must hold at the start of each iteration !!!

- The loop invariant:
  - In order for the loop statement to execute correctly for every iteration:
    - Some property/relationship must be valid at the start of every iteration of the loop
    - In fact, the statements inside the loop body must make sure this pre-condition will be satisfied at the start of the next iteration !

Note:
- I have to be vague on what the exact condition is...
- Because the property is different for different loops

- The technical term for this property that is true at the start of every iteration of the loop is:
  - the loop invariant
Examples of *loop invariant*

- **Loop 1**: find all divisors of the number *n*

```java
x = 1;
while (x <= n) // Run x = 1, 2, ..., n
{
    Invariant:
    We have found all divisors of n that are < x
    x = the current divisor that we will try

    if (n % x == 0)
    {
        // x is a divisor of n
        System.out.println(x); // Print x (because it's a divisor)
    }

    x++; // Make sure we move to the next number !!!
    // or else: infinite loop !!!
}

Invariant when loop ends:
We have found all divisors of n that are <= n
```

**Note:**
- The *loop invariant* must be *true* at the start of every iteration !!!

- Furthermore: the *loop invariant* is *also true* when the loop exits

Because:
- The loop *exits* when the *loop-continuation-condition* is false.
- In other words, the program was *ready* to *start the next iteration* of the *loop*
- Because we are starting a *new iteration*, the *loop invariant* *must* then be *true* !!!
Infinite loop

• Infinite loop = a loop statement that does not end
• You must type control-C in the terminal window to terminate the Java program when you do run it!
• The while-body must contain statements that will affect (change) the outcome of the loop-continuation-condition of the while-loop !!!
Controlling a Loop with a Sentinel Value

- Another common technique for controlling a loop is to designate a special value when reading and processing a set of values. This special input value, known as a sentinel value, signifies the end of the input. A loop that uses a sentinel value to control its execution is called a sentinel-controlled loop.

- Example: read integers until the input is 0.
Common errors with while-loops

- **Common error 1**: bogus semicolon

  Example:

  ```java
  public class Error01 {
    public static void main(String[] args) {
      int a;
      a = 1;
      while (a <= 10) ; // BOGUS semicolon!
      {  
        System.out.println(a); // Print a
        a++;
        // Increment a
      }
      System.out.println("Done");
      System.out.println("Exit: a = " + a);
    }
  }
  ```
○ **Common error 2:** forgetting to use statement block

**Example:**

```java
public class Error02 {
    public static void main(String[] args) {
        int a;

        a = 1;

        while (a <= 10) {
            System.out.println(a); // Print a
            a++; // Increment a
            System.out.println("Done");
        }
        System.out.println("Exit: a = " + a);
    }
}
```
The break and continue statements

- **Introduction**
  - There are 2 special statements that can affect the execution of loop statements (such as a while-statement).
  - The special statements are:
    - break
    - continue

We will study their meaning and how to use these special statements inside the while-statement.
- **The break statement**
  - **Syntax:**
    ```
    break;
    ```
  - **Effect:**
    - When the `break statement` is *executed* inside a `loop-statement`, the `loop-statement` is terminated *immediately*
    - The *execution* of the program will *continue* with the `statement following the loop-statement`
  - **Schematically:**
    ```
    while ( loop-continuation-condition )
    {
      statement1
      statement2
      ....
      break;
      ....
    }
    ```
    *Execution proceeds to here*
    ```
    statement following the while loop
    ```
The continue statement

Syntax:

```
continue;
```

Schematically:

```
while ( loop-continuation-condition )
{
    statement1
    statement2
    ....
    continue;
    ....
}
```

Execution proceeds to here

```
statement following the while loop
```
Using the *while*-statement to process data files

- File (or *data file*)
  - What is a *file*:
    - *File* = an *electronic* document stored inside a computer system that contains *information* (data)
    - A *file* can be created by *humans* using a computer program called an *editor* (e.g., *gedit*)
    - A *file* can also be created when a *computer program* needs to *store its output data*.

- A common usage of the *while*-statement
  - One of the *common usage* of the *while*-statement is *file processing*
  - We will study how to:
    - Open a file
    - Read all the items in the file, and do some *simple processing* on the data items.
• Opening a data file

  ○ How to *open a file* in Java:

    ```java
    File myFile;       // Define a "File" type variable to receive
    // the opened file
    myFile = new File("Path-name-of-the-file");     // Open the file
    ```

    The *variable* `myFile` contains *information* about the *opened file*

    The *variable* `myFile` will be used in *read operations*

  • Scanning an *opened file*

    ○ *Reading data* from an *opened file* is a *very complex task*

    ○ *Fortunately:*

      - An *opened file* behaves exactly like a *keyboard* (which is represented by the *variable* `System.in` in Java)

    ○ We can *use the method* in the scanner class to help us *read and convert* the data into the *desired encoding*
- **Read from a `data file`:**

  **Construct a `scanner object` using an `opened file`:**

  ```java
  /* ------------------------------------------
  Open a data file
  ------------------------------------------ */
  File myFile;  // Define a "File" type variable to receive
                // the opened file
  
  myFile = new File("Path-name-of-the-file");  // Open the file

  /* ------------------------------------------
  Construct a Scanner from the opened file "myFile"
  ------------------------------------------ */
  Scanner in;  // Define a Scanner typed variable

  in = new Scanner(myFile);  // Construct a Scanner that read
                               // data from opened file"myFile"
  
  /* From this point onwards, you can use */

  - `in.nextDouble()` to read a `floating point number` from the `data file`
  - `in.nextInt()` to read an `integer number` from the `data file`
  - `in.next()` to read a `string (word)` from the `data file`
• **Checking for available input data**

  ○ There are **very useful methods** available in the **scanner class** to **test** if the **input file** is **empty (exhausted)** or not.

  ○ **Check function** for **input availability** on **scanner typed variable in**:

    ```
    ┌ in.hasNextDouble() ────────────────────────────────────────────────────────────────────────────────────────────┐
    │ returns true if the scanner object in contains at least one double typed value in the input.            │
    │ returns false otherwise (no more double typed value in the input.)                                   │
    └───────────────────────────────────────────────────────────────────────────────────────────────────────────┘
    ```

    ```
    ┌ in.hasNextInt() ───────────────────────────────────────────────────────────────────────────────────┐
    │ returns true if the scanner object in contains at least one int typed value in the input.         │
    │ returns false otherwise (no more int typed value in the input.)                                  │
    └──────────────────────────────────────────────────────────────────────────────────────────────────────┘
    ```

    ```
    ┌ in.hasNext() ───────────────────────────────────────────────────────────────────────────────────┐
    │ returns true if the scanner object in contains at least one String typed value in the input.     │
    │ returns false otherwise (no more String typed value in the input.)                             │
    └──────────────────────────────────────────────────────────────────────────────────────────────────────┘
    ```
Programming example 1: print the content of a data file

Java program:

```java
import java.io.*;
import java.util.Scanner;

public class File01
{
    public static void main(String[] args) throws IOException
    {
        File myFile = new File("inpl"); // Open file "inpl"
        Scanner in = new Scanner(myFile); // Make Scanner obj with opened file

        String x; // Variable to receive a string

        while ( in.hasNext() )
        {
            x = in.next(); // Read a string (word)
            System.out.println(x); // Print string read
        }

        System.out.println("Done");
    }
}
```
Programming example 2: sum the (floating point) data in a data file

○ Java program:

```java
import java.io.*;
import java.util.Scanner;

public class File02
{
    public static void main(String[] args) throws IOException
    {
        File myFile = new File("inp2");  // Open file "inp2"
        Scanner in = new Scanner(myFile); // Make Scanner obj with opened file

        double x;          // Variable to receive a floating point number
        double sum;        // Running sum

        sum = 0.0;          // Initialize ("clear the slate")

        while ( in.hasNextDouble() )
        {
            x = in.nextDouble();     // Read a floating point number
            sum = sum + x;           // Add to the running sum
        }

        System.out.println("Sum = " + sum);
    }
}
```
Programming example 3: compute the average of (floating point) data in a data file

```java
import java.io.*;
import java.util.Scanner;

public class File03 {
  public static void main(String[] args) throws IOException {
    File myFile = new File("inp2"); // Open file "inp2"
    Scanner in = new Scanner(myFile); // Make Scanner obj with opened file

    double x;                     // Variable to receive a floating point number
    double sum;                   // Running sum
    int N;                        // # items added ***** New code

    sum = 0.0;                     // Initialize ("clear the slate")
    N = 0;                        // No items added ***** New code

    while (in.hasNextDouble()) {
      x = in.nextDouble();        // Read a floating point number
      sum = sum + x;               // Add to the running sum
      N++;                         // One more item added ***** New code
    }

    System.out.println("Average = " + sum/N);
  }
}
```