GC01 Introductory Programming

Induction Week – Lecture 4

Dr Ghita Kouadri Mostefaoui
Department of Computer Science

Outline

- The `switch` statement
- `while` loops
- `do-while` loops
- `for` loops
- Prefix and Postfix Increment and Decrement Operators
- Nested loops
- The `Random` class

The `switch` Statement

The **`switch`** Statement

- It is often the case that you want the value of a single variable decide which branch a program should take:

```
if (x == 1)
    statement or block 1
else if (x == 2)
    statement or block 2
else if (x == 3)
    statement or block 3
else
    statement or block 4
```

- This is tedious and not very aesthetically pleasing.
- Java provides a structure that lets the value of a variable or expression decide which branch to take
  - This structure is called a **`switch`** statement.

The **`switch`** Statement

- General form of a switch statement:

```
switch (SwitchExpression) {
    case CaseExpression1:
        //One or more statements
        break;
    case CaseExpression2:
        //One or more statements
        break;
    default:
        //One or more statements
}
```

- *SwitchExpression* – a variable or expression that has to be either **char**, **byte**, **short**, **int** and **String** (since Java 7)
- **case** – keyword that begins a **case** statement (there can be any number of **case** statements)
- *CaseExpression1* – is of the same type as *SwitchExpression*.

The **`switch`** Statement

- General form of a switch statement:

```
switch (SwitchExpression) {
    case CaseExpression1:
        //One or more statements
        break;
    case CaseExpression2:
        //One or more statements
        break;
    default:
        //One or more statements
}
```

- After the statement(s) inside of a **`case`** statement's block, often the keyword **`break`** appears.
- After all of the **`case`** statements, there is the default case, which begins with the keyword **`default.`**

The **`switch`** Statement

- General form of a switch statement:

```
switch (SwitchExpression) {
    case CaseExpression1:
        //One or more statements
        break;
    case CaseExpression2:
        //One or more statements
        break;
    default:
        //One or more statements
}
```

- What this does is compare the value of *`SwitchExpression`* to each *`CaseExpression`*s.
  - If they are equal, the statements after the matching **`case`** statement are executed.
  - Once the **`break`** keyword is reached, the statements after the **`switch`** statement's block are executed.
  - If none of the *`CaseExpression`*s are equal to *`SwitchExpression`*, then the statements below the **`default`** case are executed.

The **switch** Statement

```
if (x == 1)
   y = 4;
else if (x == 2)
   y = 9;
else
   y = 22;
```

Is the same as…

```
switch (x) {
   case 1:
      y = 4;
      break;
   case 2:
      y = 9;
      break;
   default:
      y = 22;
}
```

The **`switch`** Statement Notes

- The *CaseExpression* of each **`case`** statement must be unique.
- The **`default`** section is optional.
- Again, the *SwitchExpression* and all of the *CaseExpression*s must be either **`char`**, **`byte`**, **`short`**, **`int`** **`or`** **`string`**.
- Without the **`break;`** at the end of the statements associated with a **`case`** statement, the program "falls through" to the next **`case`** statement's statements, and executes them.
  - If this is what you actually want, then leave out the **`break;`**

Loops

Loops

- So far, we've used decision structures to execute statements that follow the condition ***one or zero times***.
- What if we want the user to keep trying to put in valid input until she succeeds?
  - How would we do this with decision structures?
    - Answer: it's not possible
    - Solution: using loops
- A loop is a control structure that causes a statement or group of statements to repeat.
  - We will discuss three (possibly four) looping control structures.
    - They differ in how they control the repetition.

The **`while`** Loop

General Form:

`while` (*`BooleanExpression`*)

`Statement or Block`

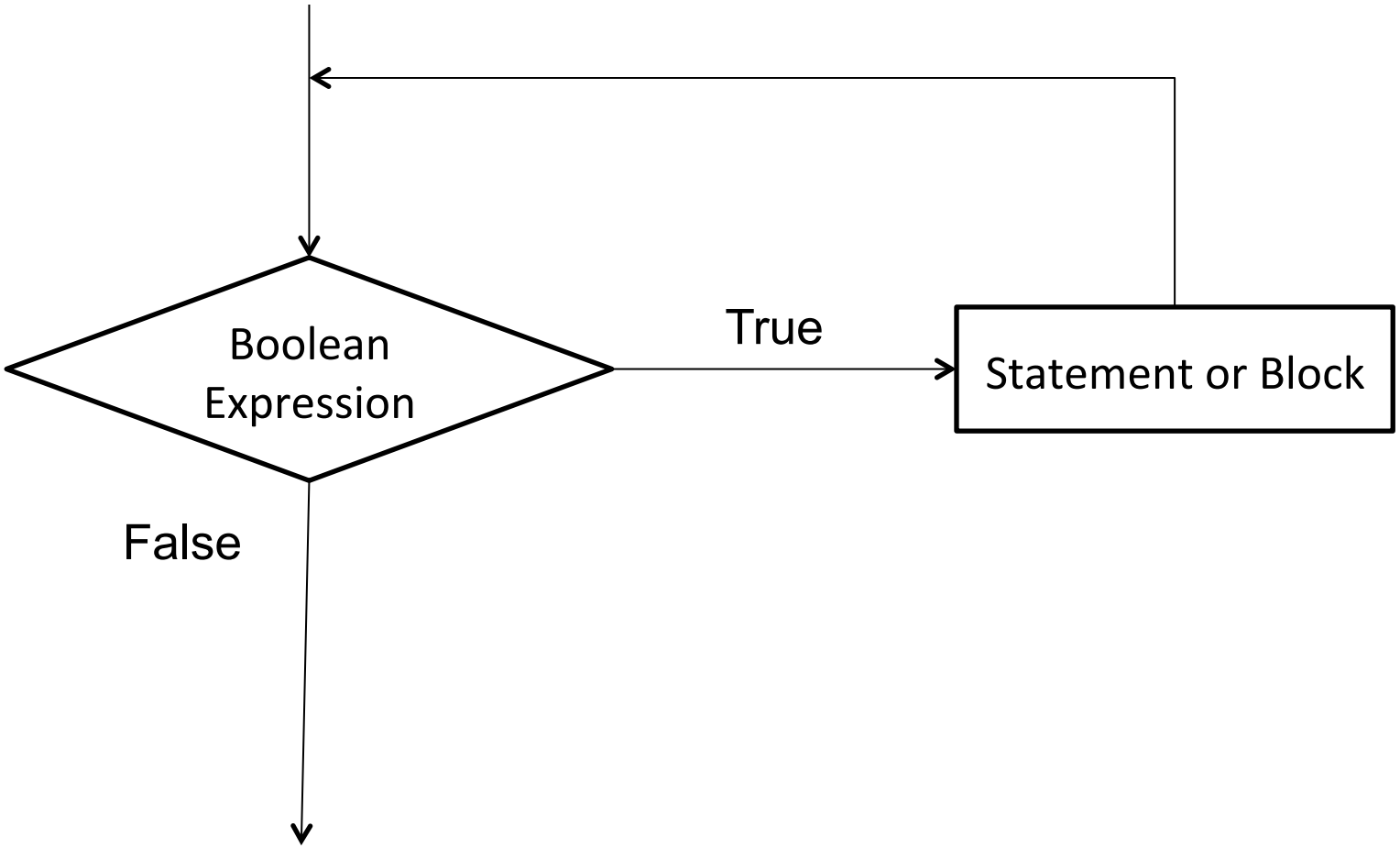
- First, the *`BooleanExpression`* is tested
  - If it is **`true`**, the *`Statement or Block`* is executed
    - After the *`Statement or Block`* is done executing, the *`BooleanExpression`* is tested again
      - If it is still **`true`**, the *`Statement or Block`* is executed again
        - » This continues until the test of the *`BooleanExpression`* results in **`false`**.

While example Loop

```
int count = 1;
while (count < 11) {
        System.out.println("Count is: " + count);
        count++;
   }
```

- Here, `count` is called a loop control variable.
  - A loop control variable determines how many times a loop repeats.
- Each repetition of a loop is called an iteration.
- The a while loop is known as a pretest loop, because it tests the boolean expression before it executes the statements in its body.
  - Note: This implies that if the boolean expression is not initially **`true`**, the body is never executed.

The `while` Loop Flowchart



The **do-while** Loop

General Form:

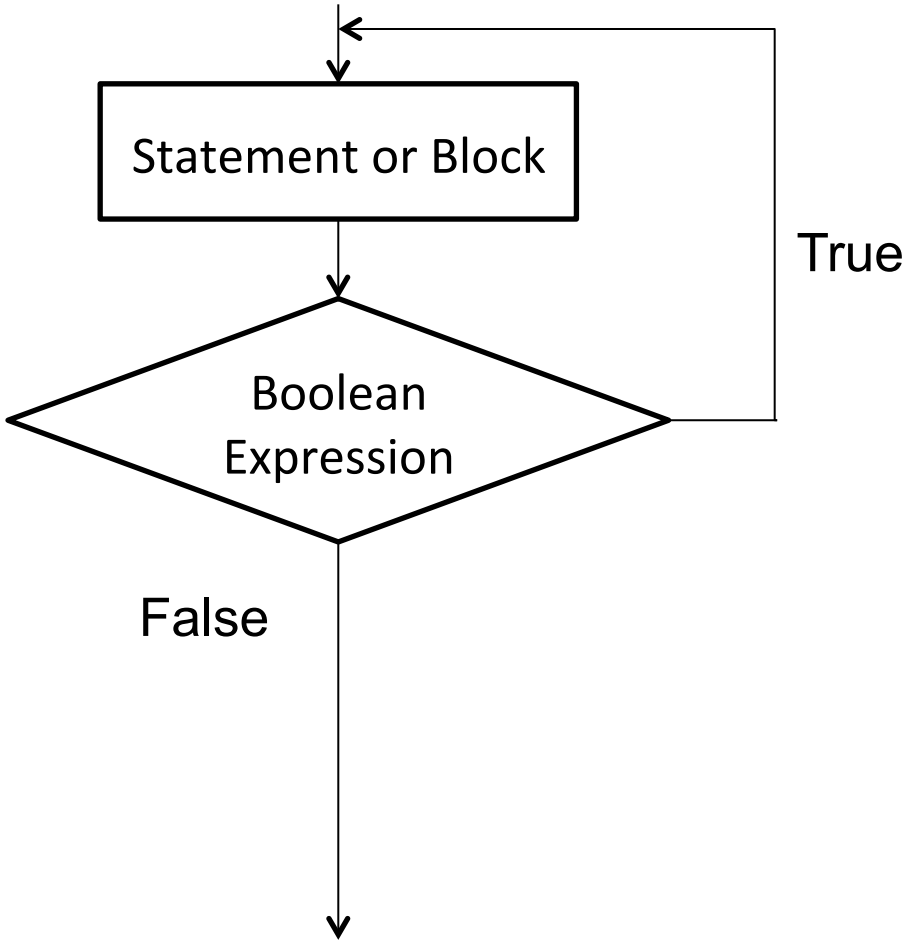
```
do
     Statement or Block
while (BooleanExpression);
```

- Here, the *Statement or Block* is executed first
- Next, the *BooleanExpression* is tested
  - If true, the *Statement or Block* is executed
    - Then the *BooleanExpression* is tested
      - This continues until the *BooleanExpression* is false.
- Again, this is a ***posttest*** loop, meaning the *BooleanExpression* is tested at the end.
  - Note that this means the *Statement or Block* will ALWAYS be executed ***at least once***.
- Also not the semicolon at the end of the last line.

do-while Example

```
int count = 1;
        do {
            System.out.println("Count is: "
+ count);
            count++;
        } while (count < 11);
```

The `do-while` Loop Flowchart



The `for` Loop

- **`while`** and **`do-while`** are conditionally-controlled loops.
  - A Conditionally-Controlled Loop executes as long as a particular condition exists.
- However, sometimes you know exactly how many iterations a loop must perform.
  - A loop that repeats a specific number of times is called a count-controlled loop.
  - For example, you may ask for information about the 12 months about a year.
  - Java provides a structure specifically for count-controlled loop called the **`for`** loop.

The `for` Loop

- General Form of a for loop:

```
for(Initialization; Test; Update)
    Statement or Block
```

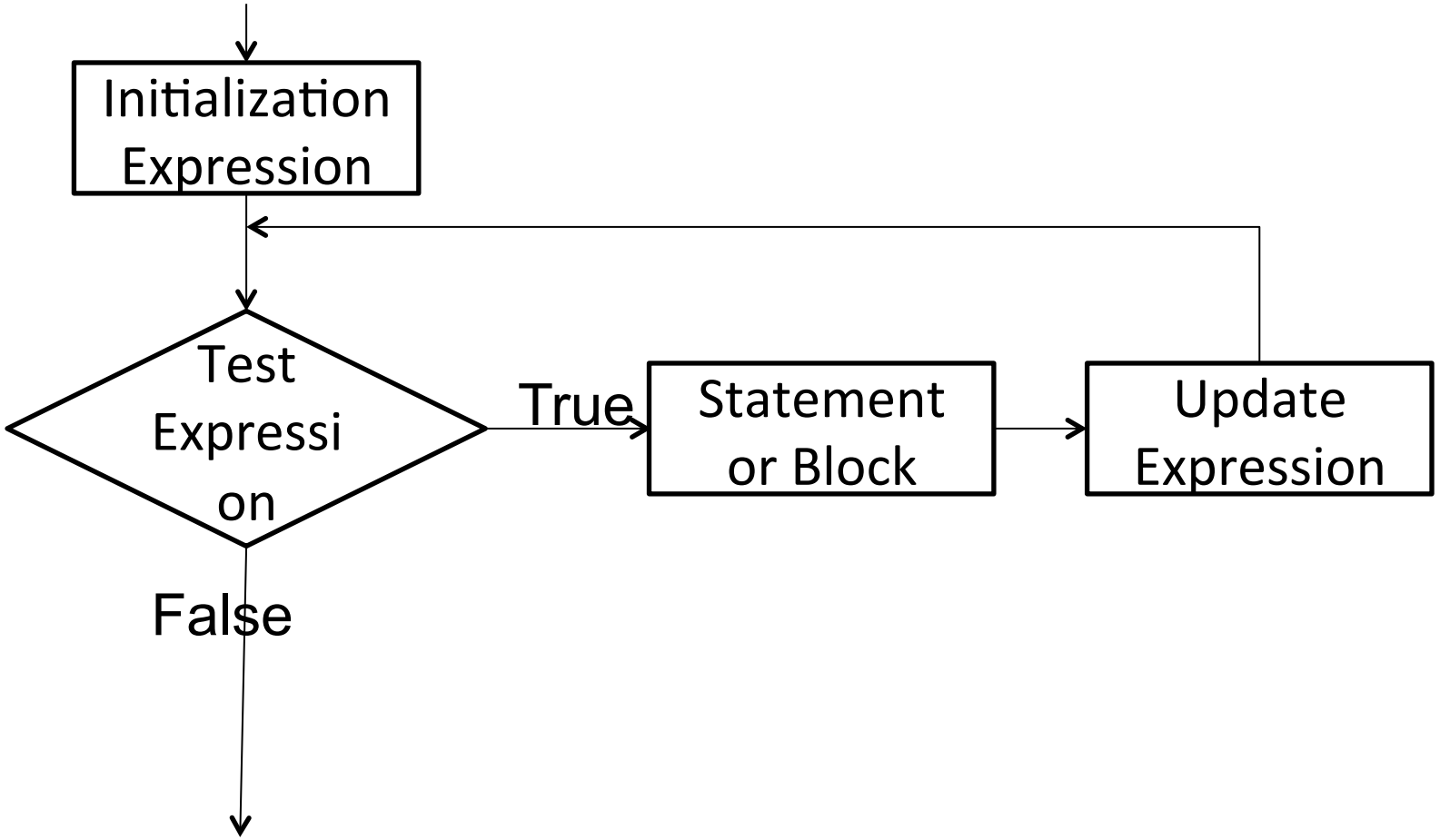
  - *Initialization* – an initialization expression that happens once when the loop is first reached.
    - Normally used to initialize the control variable
  - *Test* – boolean expression known as the test expression that controls the execution of the loop.
    - As long as this is **true**, the loop with iterate again
    - Note: the **for** loop is a pretest loop
  - *Update* – expression known as the update expression that executes at the end of every iteration
    - Usually used to change the control variable.

`for` Loop Example

```
for(int count = 0; count < 5; count++)
    System.out.println("Hello!");
```

- This will print “Hello!” 5 times.
- First, `count` is initialized to 0.
  - `count` is often called a counter variable because it keeps count of the number of iterations.
- Then, `count < 5` is tested.
- It is **`true`** so the body is executed.
- Then, `count` is incremented.
- This happens 5 times until `count = 5` which makes `count < 5` **`false`**.
- Note that `count` is declared inside of the loop header, this makes it have block-level scope in the loop.
  - This implies that it can be used in the body of the loop.

`for` Loop Flowchart



The `for` Loop Notes

- Remember: the **`for`** loop is a pretest loop.
- Use the update expression to modify the control variable, not a statement in the body of the loop
- You can use any statement as the update expression:

```
count--
count += 2
```

- You can declare the loop control variable outside of the loop header, and it's scope will not be limited to the loop.

```
int count;
for(count= 0; count < 5; count++)
   System.out.println("Hello!");
count = 99;
```

Increment and Decrement Operators

Prefix and Postfix Increment and Decrement Operators

- We talked about the `++` and `--` operators before

  `x++`

  `x--`

  - These are known as <u>postfix</u> increment and decrement operators, because they are placed after the variable.
  - There is also <u>prefix</u> increment and decrement operators:
    - `++x`
    - `--x`
    - What's the difference?
      - » **<u>When</u>** the increment or decrement takes place.

```
int x = 1, y;
y = x++;
```

  - `y` is 1 `x` is 2.
    - The increment operator happened **after** the assignment operator.

```
int x = 1, y;
y = ++x;
```

  - `y` is 2 `x` is 2.
    - The increment operator happened **before** the assignment operator.

Nested Loops

Nested Loops

- Just like in `if` statements, loops can be nested.
- This is required when a repetition of statements itself must be repeated a number of times.

Nested Loops Example

```
public class NestedLoops
{
    public static void main(String [] args)
    {
        for (int i=1; i<=9; i++)
        {
            System.out.println();
            for (int j=1; j<=i; j++)
            {
                System.out.print(j);
            }
        }
        System.out.println();
    }
}
```

`break` and `continue`

- Java provides two keywords that can be used to modify the normal iteration of a loop:
  - **`break`** – when encountered in a loop, the loop stops and the program execution jumps to the statement immediately following the loop block.
  - **`continue`** – when encountered in a loop, the current iteration is skipped.

break and **continue**

```
System.out.println ("starting loop:");
for (int n = 0; n < 7; ++n)
{
    System.out.println ("in loop: " + n);
    if (n == 2) {
        continue;
    }
    System.out.println ("   survived first guard");
    if (n == 4) {
        break;
    }
    System.out.println ("   survived second guard");
    // continue at head of loop
}
// break out of loop
System.out.println ("end of loop or exit via break");
```

Java `Random` class

The `Random` Class

- Some application require randomly generated numbers
- The Java API provides a class called Random that does exactly that.
- Need to import it:

```
import java.util.Random;
```

- To create an object:

```
Random identifier = new Random();
```

- The random class provides many methods for generating random numbers, namely:
  - `nextDouble()` – Returns the next random number as a `double` between 0.0 and 1.0.
  - `nextInt()` – Returns the next random number as an `int` within in the range of `int` (-2,147,483,648 to 2,147,483,648)
  - `nextInt(int n)` – Returns the next random number as an `int` within in the range of 0 and `n`.

Random class example

```
import java.util.Random;

/** Generate 10 random integers in the range 0..99. */
public final class RandomInteger {

  public static final void main(String... aArgs){
    System.out.println("Generating 10 random integers in range 0..99.");

    //note a single Random object is reused here
    Random randomGenerator = new Random();

    for (int i = 1; i <= 10; ++i){
      int randomInt = randomGenerator.nextInt(100);
      System.out.println("Generated : " + randomInt);
    }

    System.out.println("Done.");
  }
}
```